Homework 4

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.

Example 2.11

Suppose we have eight machines that we want to monitor with respect to their availability. We can keep track of them with an eight-bit BUSYNESS bit vector, where a bit is 1 if the unit is free and 0 if the unit is busy. The bits are labeled, from right to left, from 0 to 7.

The BUSYNESS bit vector 11000010 corresponds to the situation where only units 7, 6, and 1 are free and therefore available for work assignment.

Suppose work is assigned to unit 7. We update our BUSYNESS bit vector by performing the logical AND, where our two sources are the current bit vector 11000010 and the bit mask 01111111. The purpose of the bit mask is to clear bit 7 of the BUSYNESS bit vector, while leaving alone the values corresponding to all the other units. The result is the bit vector 01000010, indicating that unit 7 is now busy.

Suppose unit 5 finishes its task and becomes idle. We can update the BUSYNESS bit vector by performing the logical OR of it with the bit mask 00100000. The result is 01100010, indicating that unit 5 is now available.

（用每一位代表一个机器，0忙1闲）

1. Write an LC-3 instruction that determines whether machine 2 is busy.

AND R3,R2,#4 `0101 011 010 1 00100` ，如果结果是 0，则 2 号机器处于忙碌状态

2. Write an LC-3 instruction that determines whether both machines 2 and 3 are busy.

`0101 011 010 1 01100` ，如果结果为 0，则 2 号和 3 号机器均处于忙碌状态

3. Write an LC-3 instruction that determines whether all of the machines are busy.

`0101 011 010 1 11111` 或者 `0101 011 010 0 00 010` ，如果结果为 0，则全部 8 台机器均处 于忙碌状态，此处要求 `R2` 的前 8 位均为 0

4. Can you write an LC-3 instruction that determines whether machine 6 is busy? Is there a problem here?

不能使用一条指令，因为 0101 的立即数最多只有五位，但可以考虑先将需要的掩码 `0000 0000 0100 0000` 存入寄存器，再使用 `0101` (`AND`) 的寄存器与模式，则可以检测6号机器的工作状态

T2

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

Address	Value	
x30FF	1110 0010 0000 0001	LEA R1, #1 (R1 = X3101)
x3100	0110 0100 0100 0010	LDR R2, R1, #2 (R2 = MEM[X3103]=X1482)
x3101	1111 0000 0010 0101	TRAP x25
x3102	0001 0100 0100 0001	x1441
x3103	0001 0100 1000 0010	x1482

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

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.)

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

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` ?

```
MAR <- SR
MDR <- Memory[MAR]
MAR <- DR
Memory[MAR] <- MDR
```

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.

$a \oplus b = a \cdot \bar{b} + \bar{a} \cdot b$

Address	Instruction	
x3000	1001 100 001 111111	NOT R4, R1
x3001	**0101 100 100 000 010**	AND R4, R4, R2
x3002	1001 011 010 111111	NOT R3, R2
x3003	**0101 011 011 000 001**	AND R3, R3, R1
x3004	1101 011 011 000 100	OR R3, R4, R3

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.

Addressing modes: register, immediate, PC-relative, indirect, Base+offset

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

T6

1. Write a **single** LC3 assembly instruction that copies the content of `R5` to `R4`.

   ADD R4, R5, #0 or AND R4, R5, #-1

2. Write a **single** LC3 assembly instruction that clears the content of `R3`. (i.e. `R3 = 0`)

   AND R3, R3, #0

3. Write **3** LC3 assembly instructions that does `R1=R6-R7`.
   - You are ONLY allowed to change the value of `R1`.
   - You may assume that the initial value of `R1` is 0.

   NOT R1, R7,
   ADD R1, R1, #1‘
   ADD R1, R1, R6

4. Write **3** LC3 assembly instructions that multiply the value at label `DATA` by 2. (`Mem[DATA] = Mem[DATA] * 2`)
   - You are ONLY allowed to change the value of `R1`.
   - You don't need to restore or clear the value of the register you used.
   - No need to consider overflow.

LD R1, DATA

ADD R1, R1, R1
ST R1, DATA

---

5. Set condition codes based on the value of `R1` using only **one** LC-3 instruction.
   - You are not allowed to change any value in the registers.

ADD R1, R1, #0 or
AND R1, R1, #-1（xFFFF）

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?

- 对于 `JMP` 指令， LC-3 需要 1 次内存访问。由于其内部内部存储的为下一条指令相对于当前地址的偏移量，除了本身的取指之外无须再进行内存访问。
- 对于 `ADD` 指令， LC-3 需要 1 次内存访问。其第一个操作数必须从寄存器中获得，第二个操 作数可以为立即数或者使用寄存器寻址，故除了本身的取指外无须再进行内存访问。
- 对于 `LDI` 指令， LC-3 需要 3 次内存访问。第一次是本身的取指，第二次是获取内存中 incremented PC + offset 位置的值，第三次是用那个值再去内存中取值。

T8

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

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

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

Address	Value
x3010	1110 0110 0011 1110LEA R3,62
x3011	0110 1000 1100 0001LDR R6,R3,1
x3012	0110 1111 0000 0001LDR R7,R4,1
x3013	0110 1101 1111 1111LDR R6,R7,-1

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` .

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

若使用 `LEA` ，由于偏移量位数不足，无法计算得到目标值.

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.

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

Address	Value	Comments
x3000	0101 0000 0010 0000	AND R0, R0, #0
x3001	0101 1111 1110 0000	AND R7, R7, #0
x3002	0001 1100 0010 0001	ADD R6, R0, #1
x3003	0001 1101 1000 0110	ADD R6, R6, R6
x3004	0101 1001 0100 0110	AND R4, R5, R6
x3005	0000 0100 0000 0001	BRz x3007
x3006	0001 0000 0010 0001	ADD R0, R0, #3
x3007	0001 1111 1110 0010	ADD R7, R7, #2
x3008	0001 0011 1111 0010	ADD R1, R7, #-14
x3009	0000 1001 1111 1001	BRn x3003
x300A	0101 1111 1110 0000	AND R7, R7, #0

反推得到 R5 的第 2 位到第 8 位 (最低位为第 1 位) 中有且仅有 4 位为 1.

```
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;
```

T10

`R0` and `R1` contain 16-bit bit vectors. The program below determines if rotating `R1` left by n bits produces the same bit vector that is in `R0`. If yes, the program stores the value n in M[x3020]. If not, the program stores -1 to M[x3020].

Rotating left a bit vector by one bit consists of left shifting the bit vector one bit, and then loading into bit[0] the bit that was shifted out of bit[15].

For example, rotating left **111**1000011110000 by 3 bits produces 1000011110000**111**.

Your job: Complete the program below by supplying the missing instructions so it stores n in location M[x3020] if rotating left `R1` by n bits produces the bit vector in `R0`, and store -1 if it is not possible to produce the bit vector of `R0` by rotating left `R1`. You are required to only use four registers: `R0`, `R1`, `R2`, and `R3`.

> Hint: The highest bit determines whether a 2's complement is positive or negative.

Address	Value
x3000	1001 000 000 111111NOT R0,R0
x3001	0001 000 000 1 00001ADD R0,R0,1
x3002	0101 010 010 1 00000AND R2,R2,0
x3003	0001 011 000 0 00 001ADD R3,R0,R1
x3004	**0000 010 000001100** BRz x3011
x3005	0001 010 010 1 00001ADD R2,R2,1
x3006	**0001 011 010 1 10000** ADD R3, R2, #-16
x3007	0000 010 000000111BRZ X300F
x3008	0101 001 001 1 11111AND R1,R1,-1
x3009	0000 100 000000010BRN X300C
x300A	0001 001 001 0 00 001ADD R1,R1,R1
x300B	0000 111 111110111BRNZP X3003
x300C	**0001 001 001 0 00 001** ADD R1, R1, R1
x300D	**0001 001 001 1 00001** ADD R1, R1, #1
x300E	0000 111 111110100BRNZP X3003
x300F	0101 010 010 1 00000AND R2,R2,0
x3010	0001 010 010 1 11111ADD R2,R2,-1
x3011	**0011 010 000001110** ST R2, x3020
x3012	1111 0000 0010 0101 HALT

依次分析每行命令。

- x3000, x3001 将 R0 中的值取负数存在 R0 中，这用来判断 R1 和 R0 是否相同，也就是 用 R1 加上 -R0 是否为 0 来判断
- x3002 将 R2 中的值置为 0
- x3003 将 R3 中的值置为 R0 和 R1 的和，也就是 R1 减去原来的 R0，那么接下来的 x3004 应该是一个判断是否为 0 跳转的命令，跳转目标现在还不知道
- x3005 是 R2 加 1，之后 x3006 未知，而 x3007 判断若 x3006 结果为 0 则跳转到 x300F. x300F, x3010 将 R2 清空后减 1，可以看出这是判断已经不可能让原本的 R1 左移成 R0 后将结果 -1 存在 R2 里，则 x3011 将 R2 中值存到 x3020 中，即 ST R2, x3020，这时也可以判断之前的 x3004 应跳转到这里，也就是 BRz x3011

- x3008, x3009 若 R1 为负数则跳转到 x300C, x300D 然后跳转到 x3003
- 若 R1 为正数， x300A 将 R1 翻倍后跳转到 x3003，否则经 x300C, x300D 后跳转到 x3003，而此时为 R1 为负数情况，也就是最高位为 1，所以将 R1 翻倍后要在最低位加 1，这样就实现了左移 1 位的操作
- 最后思考剩下的 x3006，因为左移 16 位就回到本身，所以如果 R2 加到 16 就说明已经不可能左移成 R0 了，所以 x3006 应该是判断 R2 是否为 16，也就是 ADD R3, R2, #-16