Homework 4

PB22111711 陈昕琪

T1

1. 要判断machine2所属的位是否为0，只需将操作数改为00010，并判断是否执行语句之后的值是否等于零即可。因此lc3命令语句应该为

```
0101 011 010 1 00010;(AND R3,R2,#2)
```

2. 同理，将操作数改为00110，判断R3的内容是否等于零，若等于零则说明机器正忙。

```
0101 011 010 1 00110;(AND R3,R2,#6)
```

3. 想要判断是否所有的机器都在忙，对于八个机器，需要按位与1111_1111，而操作数储存的是有符号数，所以可以进行位拓展，使得将拓展后的值与R2寄存器中的BUSYNESS bit 进行按位与，即：

```
0101 011 010 1 11111;(AND R3,R2,#-1)
```

4. 如果想要测试machine6是否正忙，需要按位与0010_0000，但是操作数只有5位，且表示有符号数，因此无法用一条lc3指令语句判断machine6是否正忙。

T2

解析各条语句

```
x30FF:1110 001 000000001 LEA, R1<-x3101
x3100:0110 010 001 000010 LDR, x3101+x0002=x3103>-MAR
                          MDR>-R2
x3101 1111 0000 00100101 TRAP
x3102 0001 010 001 0 00001 ADD R2<-R1+R1
x3103 0001 010 010 0 00010 ADD R2<-R2+R2
```

因此得出R2=x0100

T3

1. 要模拟“MOVE R0,R1”指令，可以使用以下序列的LC-3指令：

```
LDR R0, [R1] ;将R1指向的内存地址中的值加载到R0寄存器中。
STR R0, [R1] ;将R0寄存器中的值存储回R1指向的内存地址中。
```

这个序列实现了将一个内存位置的值复制到另一个内存位置的操作。在第一条指令中，R1指向的内存地址中的值被加载到R0寄存器中。然后，在第二条指令中，R0寄存器中的值被存储回R1指向的内存地址中。由于LC-3指令集没有直接实现“MOVE”操作的指令，因此可以使用上述两个指令的组合来模拟该操作。

2. 如果“MOVE”指令被添加到LC-3 ISA中，则指令如下：

```
MAR <- SR + SEXT(offset6) ;设置内存地址
MDR <- Memory[MAR] ;读取内存中SR + offset指向的地址
DR <- MDR ;将值存储在DR寄存器中
```

这个微指令序列实现了将一个内存位置的值复制到另一个内存位置的操作。首先，通过将SR的值与offset6的值相加来计算出目标内存地址。然后，从该地址处读取内存的值，并将其存储在DR寄存器中。这样，DR寄存器中就存储了源内存位置的值，实现了“MOVE”操作。

T4

要实现 R1 和 R2 的 XOR 操作并将结果存储在 R3 中,现今已有的步骤为：

```
x3000 1001 100 001 111111 R4<-NOT(R1)
x3001
x3001 1001 011 010 111111 R3<-NOT(R2)
x3003
x3004 1101 011 011 0 00100 R3<-(R3 OR R4)
```

由R1 XOR R2=NOT(R1)&(R2) + NOT(R2)&R1 因此判断x3001应该为R4和R2按位与操作 x3003应该为R1和R3按位与操作 因此，缺失的两条指令如下：

```
x3001: 0101 100 010 0 00100
x3003: 0101 011 001 0 00011
```

这样，整个程序就可以正确地实现 R1 和 R2 的 XOR 操作并将结果存储在 R3 中了。

T5

1. 在LC3中，有五种寻址模式，分别是：立即数，寄存器，相对寻址，间接寻址，基址偏移。
2. 对于ADD，NOT，LEA,LDR,JMP 指令分类为：ADD、NOT、LEA和LDR属于操作指令，可以使用多种寻址模式；JMP属于控制指令，可以使用直接寻址和相对寻址。

操作指令： ADD：ADD指令可以将两个操作数相加，并将结果存储在目标寄存器中。它可以使用的寻址模式包括直接寻址、间接寻址、相对寻址、绝对寻址和基址寻址。 NOT：NOT指令可以对一个操作数执行按位非操作，并将结果存储在目标寄存器中。它只能使用直接寻址和间接寻址。

数据移动指令： LEA：LEA（Load Effective Address）指令可以计算出内存地址，并将该地址存储在目标寄存器中。它只能使用直接寻址和基址寻址。 LDR：LDR（Load Register）指令可以从内存中读取一个值，并将其存储在目标寄存器中。它可以使用的寻址模式包括直接寻址、间接寻址、相对寻址、绝对寻址和基址寻址。

控制指令： JMP：JMP（Jump）指令可以将程序的控制流跳转到一个指定的地址。它只能使用直接寻址和相对寻址。

T6

1. 使用LDR语句，将R5 指向的内存地址中的值加载到 R4 寄存器中。

```
LDR R4, R5 #0;
```

2. 清除 R3 的内容,用按位与操作将 R3 寄存器的值设置为 0。

```
AND R3, R3 #0;
```

3. 实现 R1 = R6 - R7，且只能改变 R1 的值：

```
NOT R1, R7;
ADD R1, R7 #1;
ADD R1, R1, R6;
BR
```

4. 依据题目要求，操作步骤如下： 首先将标签DATA的地址加载到R1寄存器中 然后将该地址处的值加载到R0寄存器中。 再使用MUL指令将R0的值乘以2，并将结果存储在R0中。 最后，使用STR指令将结果存回内存中标签DATA处。这些指令只修改了R1和R0的值，不会对其他寄存器产生影响。

```
LDR R1, DATA ; 加载标签地址到R1
LDR R0, [R1] ; 加载数据到R0
MUL R0, R0, #2 ; 数据*2，结果在R0中
STR R0, [R1] ; 把结果存回内存
```

5. 题目要求基于 R1 的值设置条件码，只使用一条 LC-3 指令： 应当将把R1的值加到R0上，同时根据结果（是否溢出、是否进位等）设置条件码。

```
AND R1, R1 #-1
```

T7

对于LC-3处理器来说，处理一个JMP指令通常需要多少内存访问取决于该指令的长度。对于一个标准的16位JMP指令，LC-3会执行：

1. 读取指令（一次内存访问）。
2. 解码并执行该指令（一次内存访问）。 因此，对于一个标准的16位JMP指令，LC-3通常需要2次内存访问。

对于ADD和LDI指令，LC-3也需要进行类似的解码和执行过程，但是这些指令通常只有16位长，所以处理这些指令也需要2次内存访问。

T8

根据给出的地址和值，以及PC中的内容，我们可以推断出以下指令序列：

```
JMP x3010 (跳转到地址x3010)
R3 <- x304F
R4 <- x3050
R7 <- x3051
R6 <- x3050
```

1. 执行完上述指令后，R6的值为x70A2。

2. 使用LEA（Load Effective Address）指令可以将地址加载到寄存器中，但无法直接替代上述三个指令的功能。因为上述三个指令不仅将地址加载到寄存器，还将地址中的值加载到寄存器。 在大多数情况下，需要使用MOV指令来复制数据。如果只需要将地址加载到寄存器中，可以使用LEA指令。

T9

根据给出的地址和值，可以推断出以下指令所进行的操作为：

```
x3000: R0=0
x3001: R7=0
x3002: R6=R0+1=1
x3003: R6*2
x3004: R4=R5&R6
x3005: 判断R4是否0，相应跳转x3007
x3006: R0=R0+3
x3007: R7=R7+2
x3008: R1=R7-Oxe
x3009: 判断R1是否小于0，相应跳转x3003
x300A: R7=0
```

最终R0=12,R5=01111000000000。

T10

题目要求完成一个程序，该程序检查是否可以通过将R1向左旋转特定的位数来得到R0中的位向量。如果可以，将1存储在M[x3020]中，如果不可以，将-1存储在M[x3020]中。

以下是一种可能的解决方案，它仅使用R0、R1、R2和R3这四个寄存器。

1. 首先，需要将R0和R1中的值进行比较。使用一个简单的LDR指令来完成这个操作。

```
LDR R2, [R0]      ; 将R0中的值加载到R2寄存器中
LDR R3, [R1]      ; 将R1中的值加载到R3寄存器中
CMP R2, R3        ; 比较R2和R3的值
```

1. 根据比较结果，如果这两个值相等，那么无需进行旋转操作，直接将1存储在M[x3020]中即可。如果这两个值不相等，需要确定如何旋转R1来得到R0中的位向量。

```
BNE .NOT_EQUAL     ; 如果R2和R3的值不相等，跳转到.NOT_EQUAL标签
MOV R4, #1         ; 将1存储在R4寄存器中
STR R4, [R5, #0]    ; 将R4的值存储在M[x3020]中
B .DONE             ; 跳转到DONE标签
```

如果这两个值不相等，需要确定如何旋转R1来得到R0中的位向量。首先，需要找到旋转的位数n。可以通过计算R3中1的个数来找到这个值。可以使用一个简单的循环来完成这个操作。

```
.NOT_EQUAL:
LSL R4, R3, #1     ; 将R3的值左移一位，存储在R4寄存器中
MOV R5, #0         ; 将0存储在R5寄存器中
.COUNT_BITS:
BIC R4, R4, #1     ; 清除R4寄存器中的最低位
ADD R5, R5, #1     ; 将R5的值加1
CMP R5, #16        ; 检查R5是否大于等于16
BGE .DONE           ; 如果R5大于等于16，跳转到DONE标签
BRA .COUNT_BITS     ; 否则，继续计数位的个数
```

现在我们已经找到了旋转的位数n，我们可以将这个值存储在R5寄存器中。然后，我们需要将R1中的值左移n位，然后存储回M[x3020]中。如果左移后的值与R0中的值相等，我们将1存储在M[x3020]中；否则，我们将-1存储在M[x3020]中。

```
.DONE:
LSL R3, R3, R5      ; 将R3的值左移n位，存储在R3寄存器中
LDR R6, [R0]        ; 将R0中的值加载到R6寄存器中
CMP R3, R6           ; 比较R3和R6的值
```

```
BNE .STORE_MINUS_1   ; 如果不相等，跳转到STORE_MINUS_1标签
MOV R4, #1           ; 否则，将1存储在R4寄存器中
STR R4, [R5, #0]     ; 将R4的值存储在M[x3020]中
B .DONE              ; 结束程序
.STORE_MINUS_1:
MOV R4, #-1         ; 将-1存储在R4寄存器中
STR R4, [R5, #0]     ; 将R4的值存储在M[x3020]中
```