实验五：LC3 系统-控制器模块设计

1. 实验目的

1.1. 掌握 LC3 系统中控制器模块的设计

1.2. 掌握使用单元测试验证 LC3 系统中的控制器模块。

2. 实验内容

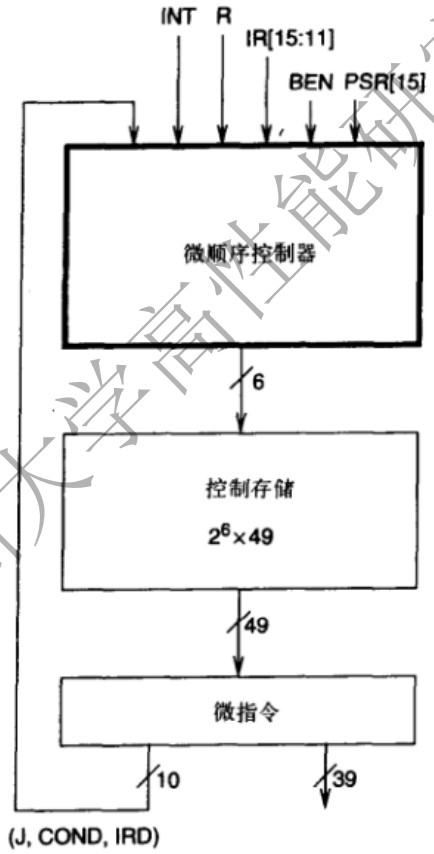
2.1. 根据实验指导设计控制器模块。

2.2. 根据实验指导对控制器模块进行单元测试。

3. 实验步骤

3.1. 控制器的设计

Controller 是计算机中最为核心的一个部件，控制着机器的运行行为。 在 LC3 中，Controller 的功能为接收上一周期的运算结果，根据上周期的状态生成本周期的状态，最后根据生成的状态查表得到控制信号（微指令），并传送给数据通路和存储器，让其做出对应的工作。原理如图所示：



图C-4 控制结构的微编程实现方法：结构图

其中状态转换以及状态编号在《计算机系统概论》p369 和 p377。而控制信号（微指令）在 p371-372。每一个状态对应一个控制信号列表（该表见文末附录），可视为一个二维数组，该二位数组即微指令表。

控制信号：信号长度为 49bits，微指令表的容量则为状态数*49bits，信号含义如下（详细含义见文末附录）：

信号名称	含义
LD.*	该信号为1时,寄存器 * 的值才能被改变。
Gate*	该信号为1时，寄存器 * 才能在在总线上传输（通常通过总线传到另一个寄存器）。由于总线只能传输一个信号，所有Gate *同时只能有一个信号为1。
*Mux	数据选择时的选择信号
ALUK	ALU的操作类型
MIO.EN	是否要读写内存
R.W	内存操作读还是写
Set.Priv	LC3执行模式

举例：

1. 在状态机号 35 时，操作为 IR<-MDR，因此 LD.IR 和 GateMDR 都为 1
2. ADDR1MUX 为 0 选择 PC，为 1 时选择 BaseR

接口定义：根据控制器定义图，可以提取出模块的输入输入信号。除了模块级别的输入输出信号，对于整个 LC3 系统需要接收两个信号代表工作状态。

接口方向	接口信号	含义
in	sig	J, COND, IDR的按位拼接
in	INT	中断信号
in	R	内存控制，为1代表读写完
in	IR	区分指令的比特位
in	BEN	上一个运算的符号情况
in	PSR	机器执行模式
out	signalEntry	输出控制信号（见文末附录）

接口方向	接口信号	含义
in	work	运行状态
in	end	结束状态

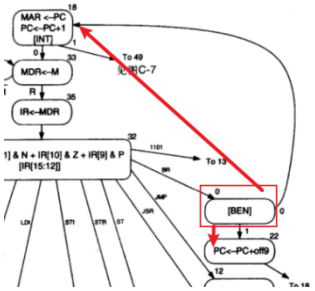
根据《计算机系统概论》p369 和 p377 列出所有的状态转移。根据当周期的状态在微指令表中索引出控制信号。根据上述接口定义，可以对模块进行定义，代码如下，SignalEntry 中表示了每个状态所包含的信号。

```scala
// 输出接口类定义（src/main/scala/LC3/controller.scala）
class signalEntry extends Bundle {
  val LD_MAR      = Bool()
  val LD_MDR      = Bool()
  val LD_IR       = Bool()
  val LD_BEN      = Bool()
  val LD_REG      = Bool()
  val LD_CC       = Bool()
  val LD_PC       = Bool()
  val LD_PRIV     = Bool()
  val LD_SAVEDSSP = Bool()
  val LD_SAVEDUSP = Bool()
  val LD_VECTOR   = Bool()
  val GATE_PC     = Bool()
  val GATE_MDR    = Bool()
  val GATE_ALU    = Bool()
  val GATE_MARMUX = Bool()
  val GATE_VECTOR = Bool()
  val GATE_PC1    = Bool()
  val GATE_PSR    = Bool()
  val GATE_SP     = Bool()
  val PC_MUX      = UInt(2.W)
  val DR_MUX      = UInt(2.W)
  val SR1_MUX     = UInt(2.W)
  val ADDR1_MUX   = Bool()
  val ADDR2_MUX   = UInt(2.W)
  val SP_MUX      = UInt(2.W)
  val MAR_MUX     = Bool()
  val VECTOR_MUX  = UInt(2.W)
  val PSR_MUX     = Bool()
  val ALUK        = UInt(2.W)
  val MIO_EN      = Bool()
  val R_W         = Bool()
  val SET_PRIV    = Bool()
}

// 输入接口类定义（src/main/scala/LC3/datapath.scala）
class FeedBack extends Bundle {
  val sig = Output(UInt(10.W))    // control signal. sig[9:4]: j   sig[3:1]: cond   sig[0]: ird
  val int = Output(Bool())        // high priority device request
  val r   = Output(Bool())        // ready: memory operations is finished
  val ir  = Output(UInt(4.W))     // opcode
  val ben = Output(Bool())        // br can be executed
  val psr = Output(Bool())        // privilege: supervisor or user
}

// 控制器模块定义（src/main/scala/LC3/controller.scala）
class Controller extends Module {
  val io = IO(new Bundle{
    val in  = Flipped(new FeedBack)
    val out = Output(new signalEntry)     // output control signal

    val work = Input(Bool())
    val end = Input(Bool())
  })
  // 模块内逻辑，待补充
}
```


代码中 Flipped 会将输入变成输出，输出变成输入。到此为止，控制器的“外壳”已经定义好，接下来控制器的内部需要对输入的信号进行处理得到输出。其原理为一个状态机，状态机根据本次的状态号和条件得到下一个状态号，然后从对应状态号中获取控制信号。状态转移示例代码如下：

```
// 实验五 任务一
// 请在下方填写 1~59的状态转移，其中17 19 46 53 55~58 没有对应状态
when(io.work && !io.end){
  switch (state) {              // 控制状态机
    // 此处为示例：当前状态为0，下一状态根据ben信号转移，若为真，则下一状态为22，否则为18
    is (0.U) { state := Mux(ben, 22.U, 18.U) }

  }
}
```

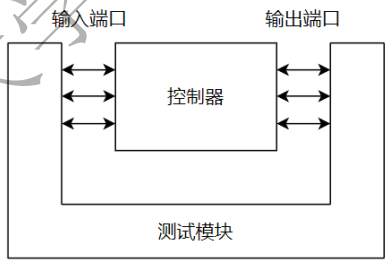
控制器判断当前 state，如过 state 为 0，则判断 ben 信号，如果为真则 state 寄存器更新为 22.U，如果为假则则 state 寄存器更新为 18.U，即下图的部分



实验五-实验一：上文给出 0 号状态转移示例，参照状态转移图在 Controller.scala 中填写 1~59 的状态转移，其中 17 19 46 53 55~58 没有对应状态，可以省略。

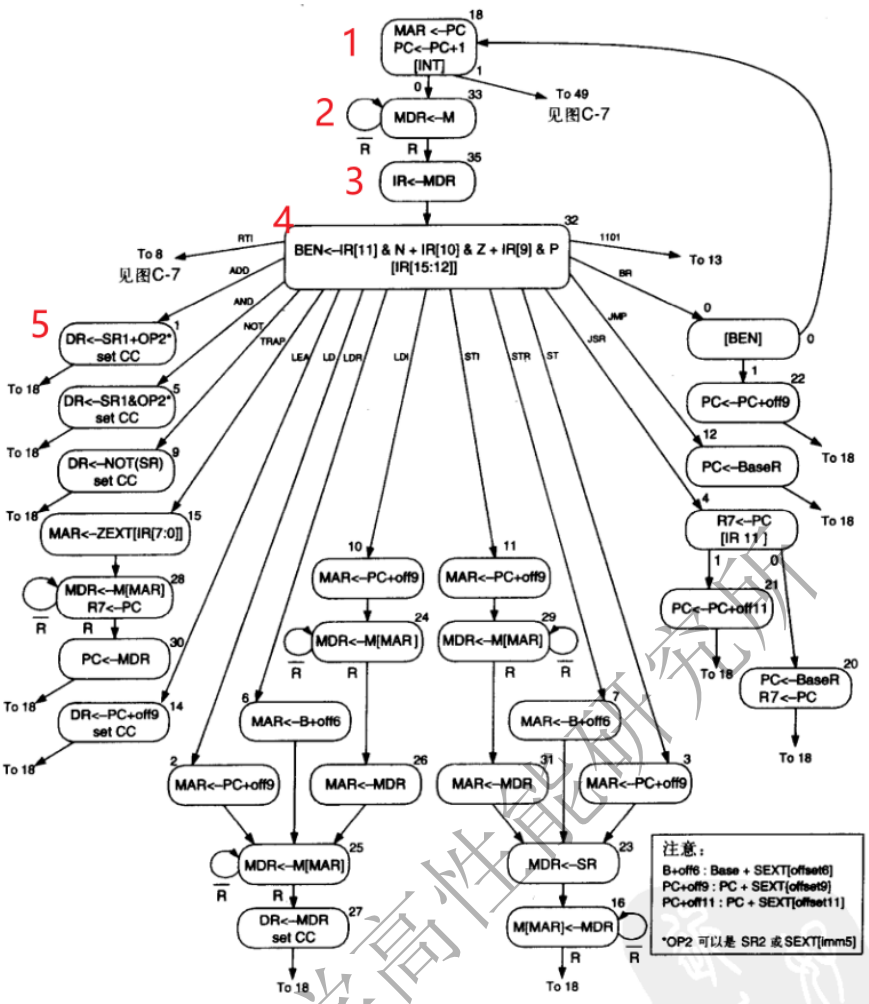
3.2. 控制器的验证

单元测试是一种快捷的仿真手段，主要用于对模块的输入端口输入信号激励，并能够获取输出端口的信号值，通过判断获取输出值的对错来保证模块设计的正确性。其原理如下图所示。



建立单元测试可以在不运行整个 LC3 情况下独立运行控制器，搭建步骤如下：
①通过模块端口把状态机设置成运行状态
②通过模块输入状态机转移的条件
③观察状态机转移结果。

我们以 add 指令在 LC3 的状态为例，状态转移顺序为 18, 33, 35, 32, 1 最后转移回到 18。



图C-2 LC-3状态机

我们搭建控制器的思路如下：

①启动控制器，即设置 work 和 end 端口，此时状态机会被初始化。
②然后激励对应状态转移的条件，如 18->33 的条件为 int 信号是 0
③如此类推，直至状态回到 18 号，即一条指令执行完成。

注意，每完成一次激励需要让时钟走一拍，让状态寄存器更新值。控制器测试 and 指令代码如下：

```scala
// src/test/scala/ControllerTest.scala

class ControllerTest extends AnyFlatSpecwith ChiselScalatestTester  // chiseltest 类和依赖
{
    behavior of "Controller"

    it should "test state machine" in {   //测试用例
        test(new Controller) { c =>

            // 初始状态
            c.io.work.poke(true.B)
            c.io.end.poke(false.B)
            c.clock.step()
            println(s"io.state=${c.io.state.peek}")    // 初始为18

            // add指令状态转移
            c.io.in.int.poke(false.B)                  // 转移条件：int为0
            c.clock.step()                             // 此处为让时钟过过一拍，让寄存器才能写入新值
            println(s"io.state=${c.io.state.peek}")    // 转移为33

            c.io.in.r.poke(true.B)                     // 转移条件：r为1
            c.clock.step()
            println(s"io.state=${c.io.state.peek}")    // 转移为35

            c.clock.step()
            println(s"io.state=${c.io.state.peek}")    // 转移为32

            c.io.in.ir.poke(1.U)                       // 转移条件：ir为1
            c.clock.step()
            println(s"io.state=${c.io.state.peek}")    // 转移为1

            c.clock.step()
            println(s"io.state=${c.io.state.peek}")    // 指令结束，转移为18

        }
    }
}
```

执行以下命令观察测试结果

```
mill -i chisel_lc3.test.testOnly -o -s LC3.ControllerTest
```

```
ControllerTest:
Controller
io.state=UInt<6>(18)
io.state=UInt<6>(33)
io.state=UInt<6>(35)
io.state=UInt<6>(32)
io.state=UInt<6>(1)
io.state=UInt<6>(18)
- should test state machine
```

实验五-任务二：根据上述示例，自选三条指令的状态转移进行测试

提示：注意每个状态转移时需要输入什么条件

附录：

表C-1 数据通路控制信号

信号	数值	
LD.MAR/1:	NO,LOAD	
LD.MDR/1:	NO,LOAD	
LD.IR/1:	NO,LOAD	
LD.BEN/1:	NO,LOAD	
LD.REG/1:	NO,LOAD	
LD.CC/1:	NO,LOAD	
LD.PC/1:	NO,LOAD	
LD.Priv/1:	NO,LOAD	
LD.SavedSSP/1:	NO,LOAD	
LD.SavedUSP/1:	NO,LOAD	
LD.Vector/1:	NO,LOAD	
GatePC/1:	NO,YES	
GateMDR/1:	NO,YES	
GateALU/1:	NO,YES	
GateMARMUX/1:	NO,YES	
GateVector/1:	NO,YES	
GatePC-1/1:	NO,YES	
GatePSR/1:	NO,YES	
GateSP/1:	NO,YES	
PCMUX/2:	PC+1	;select pc+1
	BUS	;select value from bus
	ADDER	;select output of address adder
DRMUX/2:	11.9	;destination IR[11:9]
	R7	;destination R7
	SP	;destination R6
SR1MUX/2:	11.9	;source IR[11:9]
	8.6	;source IR[8:6]
	SP	;source R6
ADDR1MUX/1:	PC,BaseR	
ADDR2MUX/2:	ZERO	;select the value zero
	offset6	;select SEXT[IR[5:0]]
	PCoffset9	;select SEXT[IR[8:0]]
	PCoffset11	;select SEXT[IR[10:0]]
SPMUX/2:	SP+1	;select stack pointer+1
	SP-1	;select stack pointer-1
	Saved SSP	;select saved Supervisor Stack Pointer
	Saved USP	;select saved User Stack Pointer
MARMUX/1:	7.0	;select ZEXT[IR[7:0]]
	ADDER	;select output of address adder
VectorMUX/2:	INTV	
	Priv.exception	
	Opc.exception	
PSRMUX/1:	individual settings, BUS	
ALUK/2:	ADD,AND,NOT,PASSA	
MIO.EN/1:	NO, YES	

状态号	LD	Gate	Mux	other
00	00000000000	00000000	000000000000000	00000
01	00001100000	00100000	000001000000000	00000
02	10000000000	00010000	000000010001000	00000
03	10000000000	00010000	000000010001000	00000
04	00001000000	10000000	000100000000000	00000
05	00001100000	00100000	000001000000000	01000
06	10000000000	00010000	000001101001000	00000
07	10000000000	00010000	000001101001000	00000
08	10000000000	00100000	000010000000000	11000
09	00001100000	00100000	000001000000000	10000
10	10000000000	00010000	000000010001000	00000
11	10000000000	00010000	000000010001000	00000
12	00000010000	00000000	100001100000000	00000
13	01000001001	00000010	000000000000100	00000
14	00001100000	00010000	000000010001000	00000
15	10000000000	00010000	000000000000000	00000
16	00000000000	00000000	000000000000000	00110
17	00000000000	00000000	000000000000000	00000
18	10000010000	10000000	000000000000000	00000
19	00000000000	00000000	000000000000000	00000
20	00001010000	10000000	100101100000000	00000
21	00000010000	00000000	100000110000000	00000
22	00000010000	00000000	100000010000000	00000
23	01000000000	00100000	000000000000000	11000
24	01000000000	00000000	000000000000000	00100
25	01000000000	00000000	000000000000000	00100
26	10000000000	01000000	000000000000000	00000
27	00001100000	01000000	000000000000000	00000
28	01001000000	10000000	000100000000000	00100
29	01000000000	00000000	000000000000000	00100
30	00000010000	01000000	010000000000000	00000
31	10000000000	01000000	000000000000000	00000
32	00010000000	00000000	000000000000000	00000
33	01000000000	00000000	000000000000000	00100
34	00001000000	00000001	001010000000000	00000
35	00100000000	01000000	000000000000000	00000
36	01000000000	00000000	000000000000000	00100
37	10001000000	00000001	001010000010000	00000
38	00000010000	01000000	010000000000000	00000
39	10001000000	00000001	001010000000000	00000
40	01000000000	00000000	000000000000000	00100
41	00000000000	00000000	000000000000001	00110
42	00000101000	01000000	000000000000000	00000

```
43      01000000000_00000100_000000000000000_00000
44      01000001001_00000010_000000000000010_00000
45      00001000010_00000001_001010000100000_00000
46      00000000000_00000000_000000000000000_00000
47      10001000000_00000001_001010000100000_00000
48      00000000000_00000000_000000000000000_00110
49      01000001001_00000010_000000000000000_00000
50      10000000000_00001000_000000000000000_00000
51      00000000000_00000000_000000000000000_00000
52      01000000000_00000000_000000000000000_00100
53      00000000000_00000000_000000000000000_00000
54      00000010000_01000000_010000000000000_00000
55      00000000000_00000000_000000000000000_00000
56      00000000000_00000000_000000000000000_00000
57      00000000000_00000000_000000000000000_00000
58      00000000000_00000000_000000000000000_00000
59      00001000100_00000001_001010000110000_00000
```