实验六：LC3 系统-数据通路模块设计

1. 实验目的

1.1. 学习设计数据通路的前置 Chisel 语法。

1.2. 掌握 LC3 数据通路的关键设计

2. 实验内容

2.1. 阅读本实验前置 Chisel 语法。

2.2. 根据实验指导设计数据通路关键组件。

3. 实验步骤

3.1. Chisel 语法学习

（1）Mux

实验三中介绍了 switch 生成选择器，Chisel 中对应一个最简单的二选一可以采用 Mux 生成选择器，其具体用法为 val aMux = Mux(cond, singal1, singal2) cond 为真时，aMux 连线到 singal1，cond 为假时，aMux 连线到 singal2

（2）MuxLookup

对于多选择器，可以采用多个 Mux 级联起来如 Mux(cond1,Mux(cond2,s1,s2),s3) 其次可以采用 MuxLookup 查找表。代码（src/test/scala/example/Lookup.scala）如下：

```scala
// 示例MuxLookup模块
class Lookup extends Module {
  val io = IO(new Bundle {
    val ina     = Input(UInt(4.W))
    val inb     = Input(UInt(4.W))
    val inc     = Input(UInt(4.W))
    val ind     = Input(UInt(4.W))
    val select  = Input(UInt(2.W))
    val result  = Output(UInt(4.W))
  })

  io.result := MuxLookup(io.select, 0.U, Seq(
    0.U -> io.ina,
    1.U -> io.inb,
    2.U -> io.inc,
    3.U -> io.ind
  ))

}
```

```scala
// 示例测试
class LookupTest extends AnyFlatSpec
  with ChiselScalatestTester {

  it should "test lookup" in {
    test(new Lookup) { c =>
      c.io.ina.poke(1.U)
      c.io.inb.poke(2.U)
      c.io.inc.poke(4.U)
      c.io.ind.poke(8.U)
      c.io.select.poke(0.U)
      println(s"select:${c.io.select.peek}, "+s"result:${c.io.result.peek}")
      c.io.select.poke(1.U)
      println(s"select:${c.io.select.peek}, "+s"result:${c.io.result.peek}")
      c.io.select.poke(2.U)
      println(s"select:${c.io.select.peek}, "+s"result:${c.io.result.peek}")
      c.io.select.poke(3.U)
      println(s"select:${c.io.select.peek}, "+s"result:${c.io.result.peek}")
    }
  }

}
```

其中 MuxLookup 第一个参数为选择信号，第二个参数为默认信号，第三个参数为一个映射表，映射表根据第一个参数选择出对应信号连到 result。

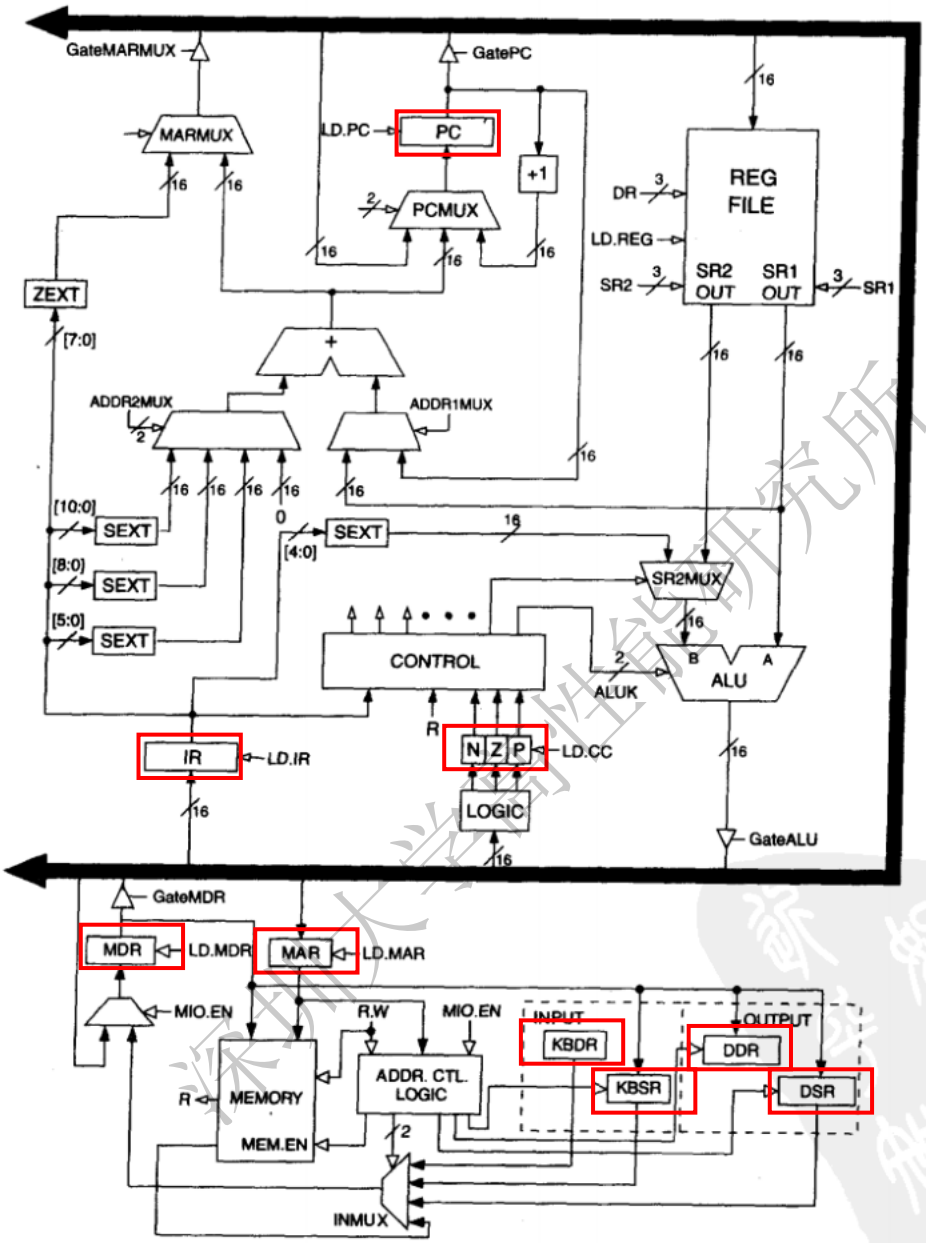
键入命令 `mill chisel_lc3.test.testOnly example.LookupTest` 可见结果如下：

```
LookupTest:
select:UInt<2>(0), result:UInt<4>(1)
select:UInt<2>(1), result:UInt<4>(2)
select:UInt<2>(2), result:UInt<4>(4)
select:UInt<2>(3), result:UInt<4>(8)
```

3.2. 数据通路设计

3.2.1. 定义关键寄存器与线

在 LC3 的数据通路中，有很多寄存器如 PC、IR、MAR、MDR 等，还有关键线网如经过 IR 寄存器译码出来的立即数等。这些寄存器和线组成基本的数据通路。



图C-3 LC-3数据通路

从上图数据通路提取关键寄存器和线如下表所示：

寄存器	描述	寄存器	描述
PC	程序计数器	IR	指令寄存器
MAR	地址寄存器	MDR	数据寄存器
KBDR	键盘输入数据寄存器	KBSR	键盘状态寄存器
DDR	显示器输出数据寄存器	DSR	显示器状态
N Z P	负 零 正 寄存器		

关键线定义如下表：

线	描述
ADDR1MUX	ADDR1MUX 选择器输出
ADDR2MUX	ADDR2MUX 选择器输出
PCMUX	PCMUX 选择器输出
DRMUX	DRMUX 选择器输出
SR1MUX	SR1MUX 选择器输出
SR2MUX	SR2MUX 选择器输出
VectorMUX	VectorMUX 选择器输出
PSRMUX	PSRMUX 选择器输出
GATEOUT	总线输出端
addrOut	ADDR1MUX 与 ADDR2MUX 相加输出
aluOut	ALU 输出
r1Data	寄存器堆读数据端口 1
r2Data	寄存器堆读数据端口 2
offset5	IR 寄存器低 5 位作符号扩展成 16 位
offset6	IR 寄存器低 6 位作符号扩展成 16 位
offset9	IR 寄存器低 9 位作符号扩展成 16 位
offset11	IR 寄存器低 11 位作符号扩展成 16 位
offset8	IR 寄存器低 8 位作零号扩展成 16 位

寄存器与线定义代码如下图所示：

```
// src/main/scala/LC3/DataPath.scala
val PC  = RegInit("h3000".U(16.W))
val IR  = RegInit(0.U(16.W))
val MAR = RegInit(0.U(16.W))
val MDR = RegInit(0.U(16.W))
val PSR = RegInit(0.U(16.W))

val KBDR = RegInit(0.U(16.W))
val KBSR = RegInit(0.U(16.W))
val DDR  = RegInit(0.U(16.W))
val DSR  = RegInit(0.U(16.W))

val BEN = RegInit(false.B)
val N = RegInit(false.B)
val P = RegInit(false.B)
val Z = RegInit(true.B)
```

```
val ADDR1MUX  = Wire(UInt(16.W))
val ADDR2MUX  = Wire(UInt(16.W))

val PCMUX     = Wire(UInt(16.W))
val DRMUX     = Wire(UInt(16.W))
val SR1MUX    = Wire(UInt(16.W))
val SR2MUX    = Wire(UInt(16.W))
val SPMUX     = Wire(UInt(16.W))
val MARMUX    = Wire(UInt(16.W))
val VectorMUX = Wire(UInt(16.W))
val PSRMUX    = Wire(UInt(16.W))
val GATEOUT   = Wire(UInt(16.W))
val addrOut   = Wire(UInt(16.W))
val aluOut    = Wire(UInt(16.W))
val r1Data    = WireInit(0.U(16.W))
val r2Data    = WireInit(0.U(16.W))

// IR Decode
val offset5  = SignExt(IR(4,0),  16)
val offset6  = SignExt(IR(5,0),  16)
val offset9  = SignExt(IR(8,0),  16)
val offset11 = SignExt(IR(10,0), 16)
val offset8  = ZeroExt(IR(7,0),  16)
```

3.2.2. 选择器连线

完成寄存器与线定义后，需要根据控制通路的信号对所定义的寄存器和线进行连接。以下图红框为例 ADDR1MUX 为 2 选 1 选择器，第一个输入为 PC，第二个输入为寄存器堆中的第一个读数据，选择信号在代码中为 SIG.ADDR1_MUX。因此这个选择器可以用一个 Mux 来描述，即:

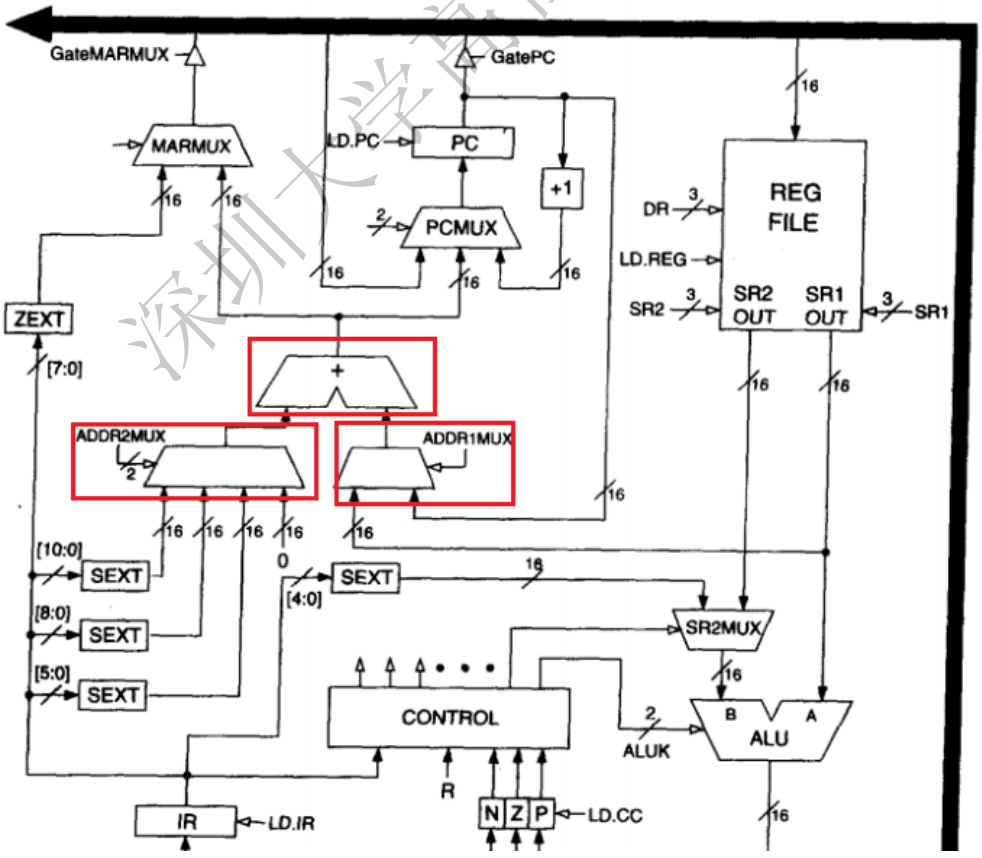
ADDR1MUX := Mux(SIG.ADDR1_MUX, r1Data, PC)

要注意 r1Data 在前还是 PC 在前，这要根据信号控制表（全表在文末附录），因此控制信号（代码中为 SIG.ADDR1_MUX）ADDR1MUX=0 时选择 PC，否则选 r1Data

```
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]]
```

同理 ADDR2MUX 为四选一选择器，可以用 MuxLookup 描述：

```
ADDR2MUX := MuxLookup(SIG.ADDR2_MUX, 0.U, Seq(
  0.U -> 0.U,
  1.U -> offset6,
  2.U -> offset9,
  3.U -> offset11
))
```



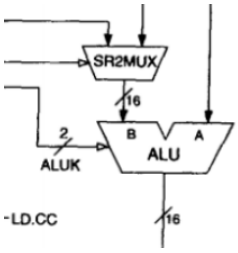
最后用一个加法语句描述加法器 `addrOut := ADDR1MUX + ADDR2MUX`

任务一：在 Datapath.scala 中完成选择器 DRMUX、SR1MUX、SR2MUX、MARMUX 的连接。

其中，SR2MUX 在数据通路图中没有指示，而是根据 LC3 指令实现，即根据 IR(5)判断第二个操作数是来自寄存器还是立即数。

3.2.3. 子模块例化与端口连接

在 Chisel 中使用 Module(new submodule)来例化子模块。对于下图 ALU 子模块的例化与端口连接代码如下：

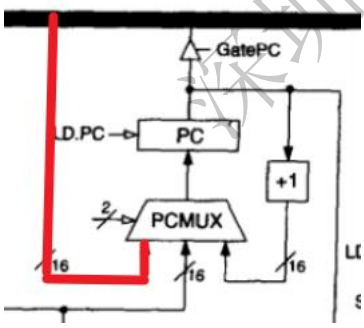



```
val alu = Module(new ALU)
alu.io.ina := r1Data
alu.io.inb := SR2MUX
alu.io.op := SIG.ALUK
aluOut := alu.io.out
```

任务二：在 Datapath.scala 中完成选择器例化寄存器堆，并连接端口

3.2.4. 总线

LC3 数据通路中只有一根总线，总线有多个输入端和多个输入端。在源码中定义 GATEOUT 线作为总线输出，其可以连接到多个线或寄存器如图中的 MAR, IR 数据均来自总线输出，因此可以接这些寄存器和线与 GATEOUT 相连。如 PCMUX 其中一个输入来自于总线 GATEOUT。




```
PCMUX := MuxLookup(SIG.PC_MUX, RESET_PC, Seq(
  0.U -> Mux(PC===0.U, RESET_PC,  PC + 1.U),
  1.U -> GATEOUT,
  2.U -> addrOut
))
```

那么总线的输入如何确定呢？由于总线上一次只能传送一个数据，因此多个输入端口同时只有一个端口提供数据，若多个数据在总线上传输则会冲突。而哪一个端口提供数据是由控制信号确定的。这里的 Gate*信号含义为，如果为 1，则对应信号提供数据。从总线的特性可以，控制信号最多只有一个为 1。因此总线可以看作一个多选一选择器，在 Chisel 中可以用 when select 等语句来实现。

```
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
```

任务三：在 Datapath.scala 中完总线连接（可用 when 实现）

其中 Vector（中断向量）在源码中未实现，可用零代替。

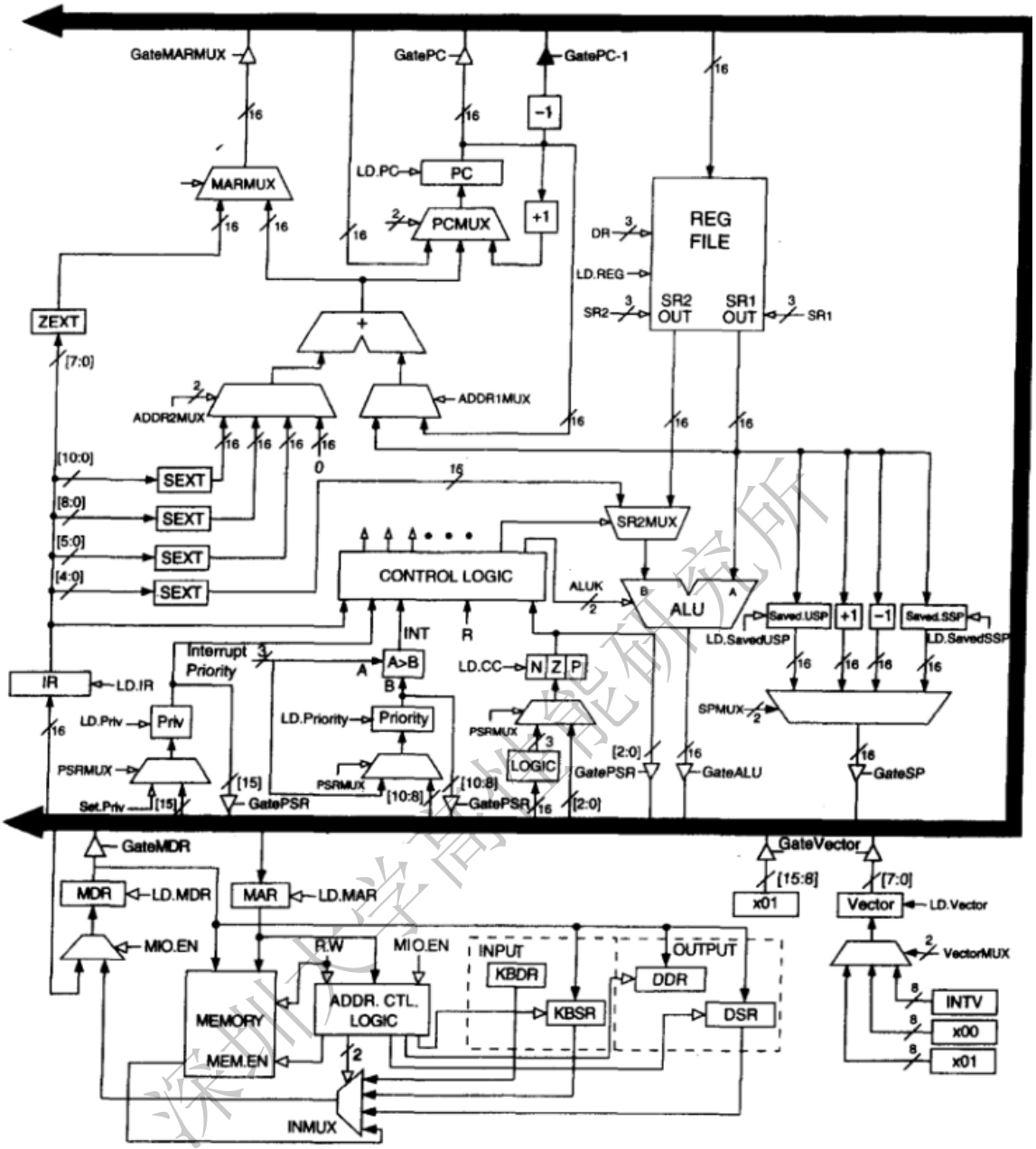
3.2.5. 阅读完整数据通路

完成上述任务后，对数据通路关键部分有所了解。结合数据通路图阅读源码。

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	



图C-8　LC-3数据通路完整图（包括中断控制）

参考答案：

任务一：

注：SR2MUX 在数据通路图中没有指示，而是根据 LC3 指令实现，即根据 IR(5)判断第二个操作数是来自寄存器还是立即数。

```
DRMUX := MuxLookup(SIG.DR_MUX, IR(11,9), Seq(
  0.U -> IR(11,9),
  1.U -> R7,
  2.U -> SP
))

SR1MUX := MuxLookup(SIG.SR1_MUX, IR(11,9), Seq(
  0.U -> IR(11,9),
  1.U -> IR(8,6),
  2.U -> SP
))

SR2MUX := Mux(IR(5), offset5, r2Data)

MARMUX := Mux(SIG.MAR_MUX, addrOut, offset8)
```

任务二：

```
/*********** Regfile Interface ***************/
val regfile = Module(new Regfile)
regfile.io.wen    := SIG.LD_REG
regfile.io.wAddr  := DRMUX
regfile.io.r1Addr := SR1MUX
regfile.io.r2Addr := IR(2, 0)
regfile.io.wData  := GATEOUT
r1Data := regfile.io.r1Data
r2Data := regfile.io.r2Data
```

任务三：

注：第一行是默认值。

```
GATEOUT := PC
when(SIG.GATE_PC){ GATEOUT := PC }
when(SIG.GATE_MDR){ GATEOUT := MDR }
when(SIG.GATE_ALU){ GATEOUT := aluOut }
when(SIG.GATE_MARMUX){ GATEOUT := MARMUX}
when(SIG.GATE_VECTOR){ GATEOUT := Cat(1.U(8.W), 0.U) }
when(SIG.GATE_PC1){ GATEOUT := PC - 1.U }
when(SIG.GATE_PSR){ GATEOUT := Cat(Seq(0.U(13.W),PSRMUX)) }
when(SIG.GATE_SP){ GATEOUT := SPMUX }
```