六、功能验证

以上是本次实验中我们所做的设计与实现。接下来，我们将对关键内容进行一一验证。

6.1 指令正确性测试

我们设计的 CPU 支持多达 76 条指令。为了简化流程，所有的指令测试均采用 Vivado 仿真度方式进行。

对于基本的一部分指令，我们采用了程序模拟验证。这一部分的实现与 Lab5 中完全相同。

对于扩展的指令，其验证用的汇编程序基本思路为：针对每一条指令，在 x5、x6 中存入所需要的源操作数，在 x11 中存入预期结果。接下来执行相应的待测指令，将计算结果存入 x5 中。最后，程序比较 x5 和 x11 中的数值，若不相等则跳转到程序末尾的 Fail 部分。换而言之，**如果程序的执行顺序是从头到尾依次执行，则可以视作待测指令完全正确。**我们将基于此进行指令正确性测试。

以下是指令正确性测试的过程。

6.1.1 基本指令正确性测试

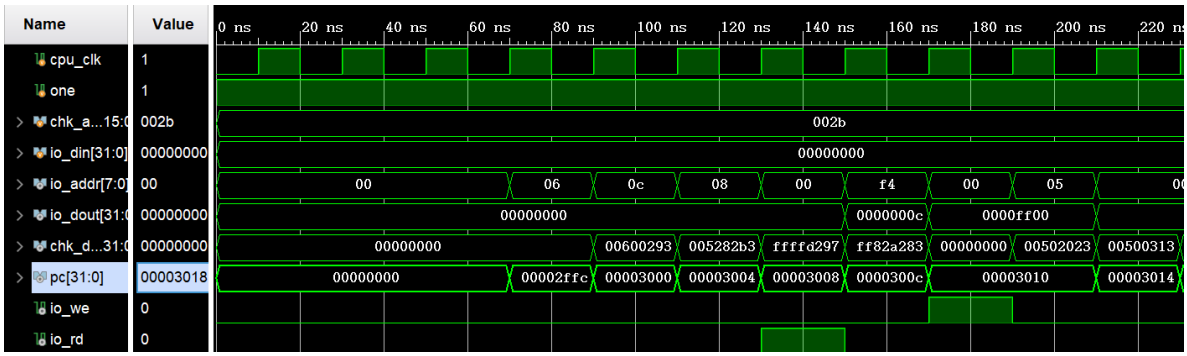
编写如下的汇编程序：

```
# This is the instruction test program
.data
base: .word 0xFF00

.text
#addi
addi t0, x0, 6

#add
add t0, t0, t0

#lw and auipc
lw t0, base

#sw
sw t0, 0(x0)

#beq, bnq, blt, bltu
addi t1, x0, 5
label1:
blt t1, x0, label2
addi t1, x0, -6
bne t1, t1, label2
beq t1, t1, label1
label2:
bltu t1, x0, label1

#jal, jalr

jal s0, label3
addi x0, x0, 0
label3:
```

```
jalr s1, 12(s0)
addi x0, x0, 0

#lui
lui t2 0xFEDCB
```

注意到上面的程序没有验证所有的运算指令。（实际上将其中最开始的 addi add 指令换成相应的其他运算指令结果依然是正确的）在本实验报告中我们仅以 addi add 版本作为结果展示。**设置 Debug 查看内容为 WB 段指令内容， PC 输出 WB 段 PC。**仿真结果如下：

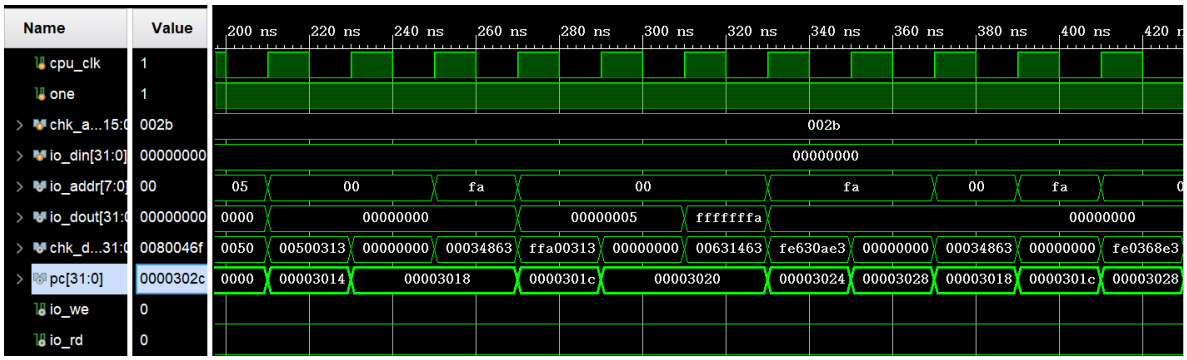
PART I



PC	指令名称	指令代码	执行指令功能	对应输出结果(时钟下降沿)
3000	addi	00600293	将 x5 加上立即数6	x5 = 00000006
3004	add	005282b3	将 x5 加上自身	x5 = 0000000c
3008	auipc	ffffd297	将 PC 减去 0x3000 后存入 x5	x5 = 00000008
300c	lw	ff82a283	从 MEM[x5 - 8] 处读取数值存入 x5	x5 = 0000FF00
3010	sw	00502023	将 x5 的值存入 MEM[0]	MEM[0] = 0000FF00

注意到仿真波形图中 PC = 3010 处被插入了一条空指令。这是因为 lw 与 其后的 sw 产生数据相关所导致的。

PART II



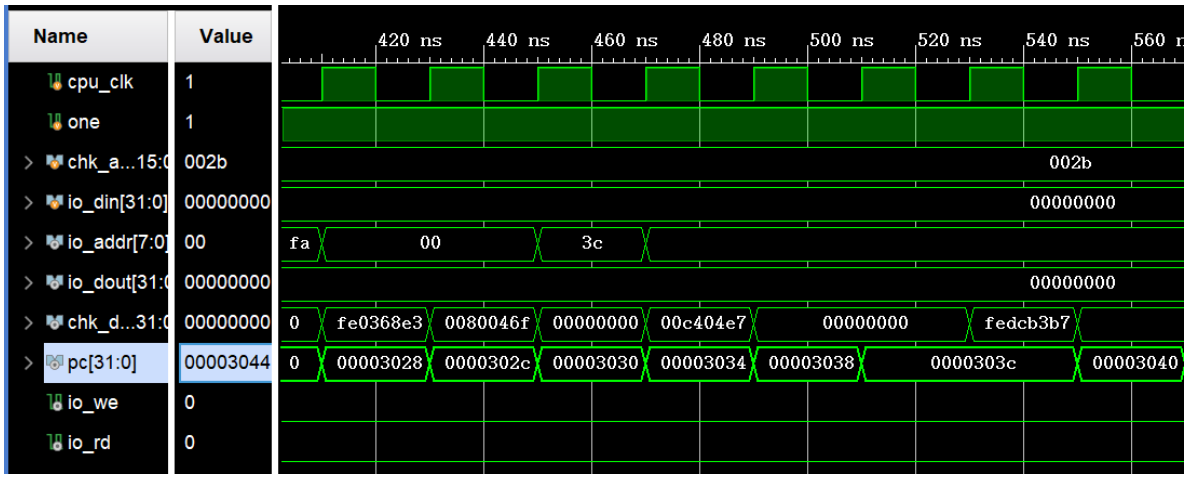
这一部分对应的指令为：

0x00003014	0x00500313	addi x6, x0, 5	19: addi t1, x0, 5
0x00003018	0x00034863	blt x6, x0, 0x00000010	21: blt t1, x0, label2
0x0000301c	0xffa00313	addi x6, x0, 0xfffffffa	22: addi t1, x0, -6
0x00003020	0x00631463	bne x6, x6, 0x00000008	23: bne t1, t1, label2
0x00003024	0xfe630ae3	beq x6, x6, 0xfffffff4	24: beq t1, t1, label1
0x00003028	0xfe0368e3	bltu x6, x0, 0xfffffff0	26: bltu t1, x0, label1
0x0000302c	0x0080046f	jal x8, 0x00000008	30: jal s0, label3

可以看到，指令执行的逻辑为 $5 < 0?\ \mathrm{N} \to -1 \neq -1?\ \mathrm{N} \to -1 = -1?\ \mathrm{Y} \to -1 < 0?\ \mathrm{Y} \to -1 <_u 0?\ \mathrm{N}$。由此可见，所有的条件跳转指令执行结果均符合预期。

此外，在 **PC=3018, 3020** 处均插入了空指令，这是因为条件跳转指令与其之前的运算指令产生了数据相关；在 **PC=3028, 301C** 处均进行了指令清空，这是因为分支预测失败所进行的流水线一个周期停顿操作。综上所述，所有的分支跳转指令执行结果完全正确。

PART III



这一部分对应的指令为：

0x0000302c	0x0080046f	jal x8, 0x00000008	30: jal s0, label3
0x00003030	0x00000013	addi x0, x0, 0	31: addi x0, x0, 0
0x00003034	0x00c404e7	jalr x9, x8, 12	33: jalr s1, 12(s0)
0x00003038	0x00000013	addi x0, x0, 0	34: addi x0, x0, 0
0x0000303c	0xfedcb3b7	lui x7, 0x000fedcb	37: lui t2 0xFEDCB

可以看到，程序运行时跳过了中间的 **PC=3030, 3038** 两条指令（插入了空指令），且相应的 PC 值已经被正确存储。对于 **PC=3034** 处的 jalr 指令，后续的 **PC=3038, 303C** 两条指令都被清除。所有的停顿操作均符合预期。

在这一部分里，我们验证了 CPU 执行 addi、add、lw、sw、bne、beq、bltu、jal、jalr、lui 指令的正确性。后续的正确性测试中这部分指令将视作正确指令。

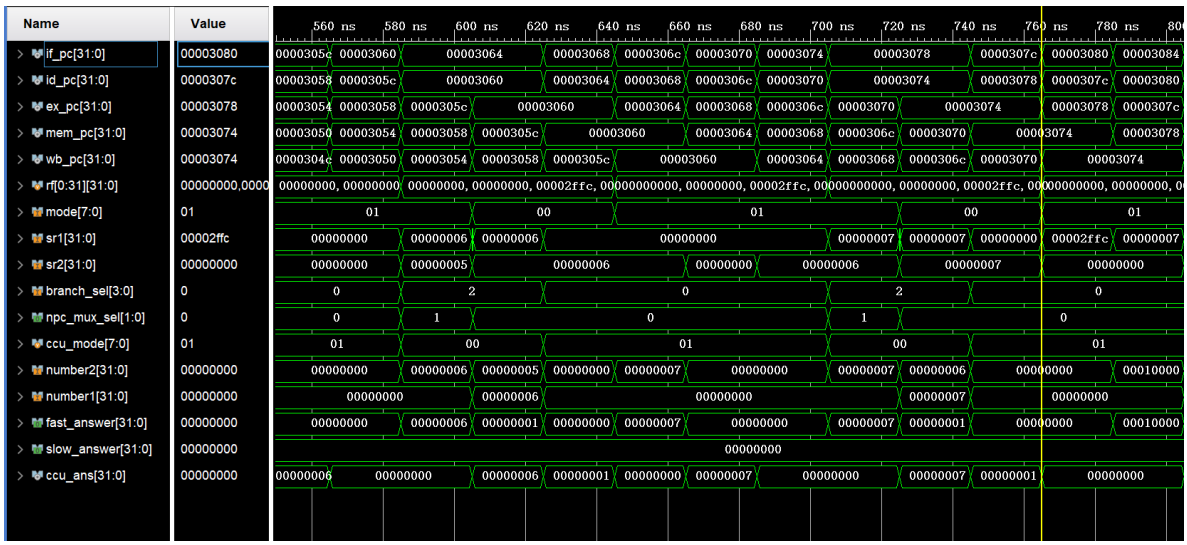
6.1.2 访存指令正确性测试

编写如下的汇编程序：

```
# suppose that our former instructions are correct
# in the following test, if test fail we will show 0xffff
# otherwise we will show 0x0000

# test lb
li x5, 0x2
sw x5, 0(x0)
lb x5, 0(x0)
li x7, 0x2
bne x5, x7, fail

# test lbu
li x5, 0x3
sw x5, 0(x0)
lbu x5, 0(x0)
li x7, 0x3
bne x5, x7, fail

# test lh
li x5, 0x4
sw x5, 0(x0)
lh x5, 0(x0)
li x7, 0x4
```

```
bne x5, x7, fail

# test lhu
li x5, 0x5
sw x5, 0(x0)
lhu x5, 0(x0)
li x7, 0x5
bne x5, x7, fail

# test ld
li x5, 0x2
sw x5, 0(x0)
ld x5, 0(x0)
li x7, 0x2
bne x5, x7, fail

# test sd
li x5, 0x6
sd x5, 0(x0)
lw x5, 0(x0)
li x7, 0x6
bne x5, x7, fail

# test sb
li x5, 0x6
sb x5, 0(x0)
lw x5, 0(x0)
li x7, 0x6
bne x5, x7, fail

# test sh
li x5, 0x7
sh x5, 0(x0)
lw x5, 0(x0)
li x7, 0x7
bne x5, x7, fail

nop

fail:
li x5, 0xffff
sw x5, 0(x0)
```

测试结果如下：



其中 0x3080 为最后 Fail 处的 li 指令。可以看到，程序的执行流程是顺序的，这说明本部分指令测试完全正确。

6.1.3 ALU 指令正确性测试——第一部分

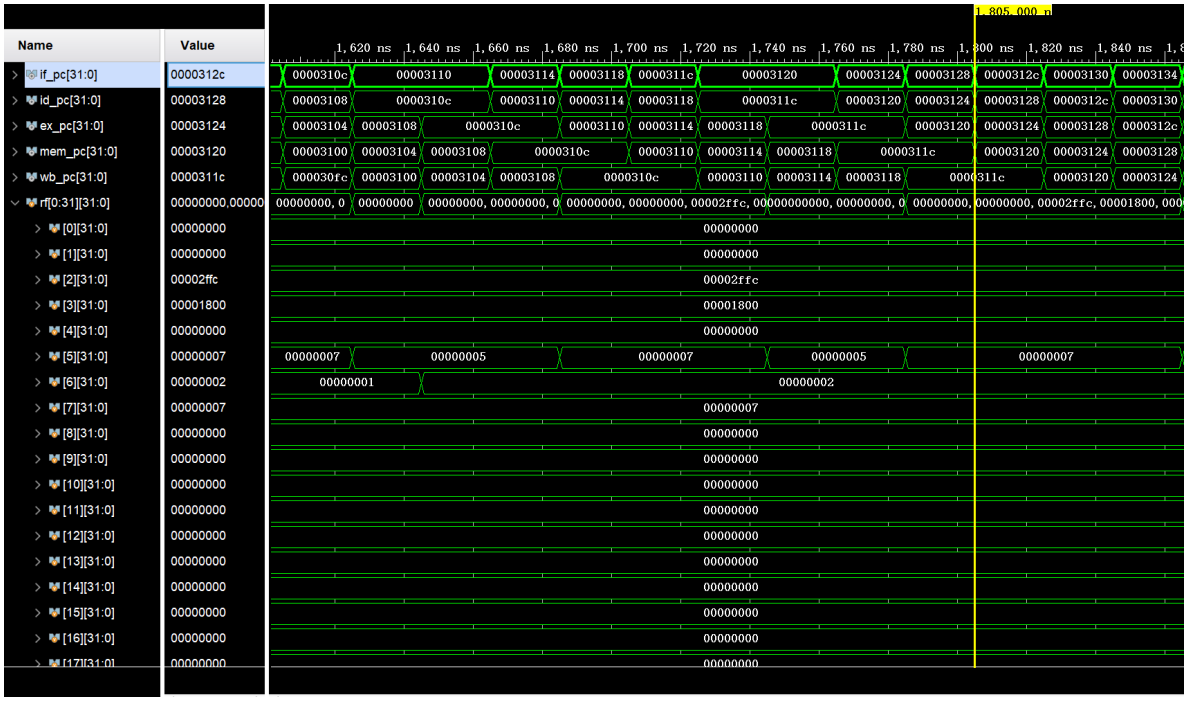
编写如下的汇编程序：

```
# suppose that our former instructions are correct
# in the following test, if test fail we will show 0xffff
# otherwise we will show 0x00ff

# test and
li x5, 0xf
li x6, 0x1
li x7, 0x1
and x5, x5, x6
bne x5, x7, fail

# test andi
li x5, 0xf
li x7, 0x1
andi x5, x5, 0x1
bne x5, x7, fail

# test or
li x5, 0x1
li x6, 0x2
li x7, 0x3
or x5, x5, x6
bne x5, x7, fail

# test ori
li x5, 0x1
li x7, 0x3
ori x5, x5, 0x2
bne x5, x7, fail

# test sll
li x5, 0x1
li x6, 0x2
```

```
li x7, 0x4
sll x5, x5, x6
bne x5, x7, fail

# test slli
li x5, 0x1
li x7, 0x4
slli x5, x5, 0x2
bne x5, x7, fail

# test slt
li x5, 0x3
li x6, 0x2
li x7, 0x2
slt x5, x5, x6
bne x5, x7, fail

# test slti
li x5, 0x3
li x7, 0x2
slti x5, x5, 0x2
bne x5, x7, fail

# test sltu
li x5, 0x3
li x6, 0x2
li x7, 0x2
sltu x5, x5, x6
bne x5, x7, fail

# test sltiu
li x5, 0x3
li x7, 0x2
sltiu x5, x5, 0x2
bne x5, x7, fail

# test sra
li x5, 0xf
li x6, 0x1
li x7, 0x7
sra x5, x5, x6
bne x5, x7, fail

# test srai
li x5, 0xf
li x7, 0x7
srai x5, x5, 0x1
bne x5, x7, fail

# test srl
li x5, 0xf
li x6, 0x1
li x7, 0x7
srl x5, x5, x6
bne x5, x7, fail

# test srli
li x5, 0xf
```

```
li x7, 0x7
srli x5, x5, 0x1
bne x5, x7, fail

# test xor
li x5, 0x5
li x6, 0x2
li x7, 0x7
xor x5, x5, x6
bne x5, x7, fail

# test xori
li x5, 0x5
li x7, 0x7
xori x5, x5, 0x2
bne x5, x7, fail

nop

fail:
li x5, 0xfff
sw x5, 0(x0)
```

测试结果如下：



其中 0x3124 为最后 Fail 处的 li 指令。可以看到，程序的执行流程是顺序的，这说明本部分指令测试完全正确。

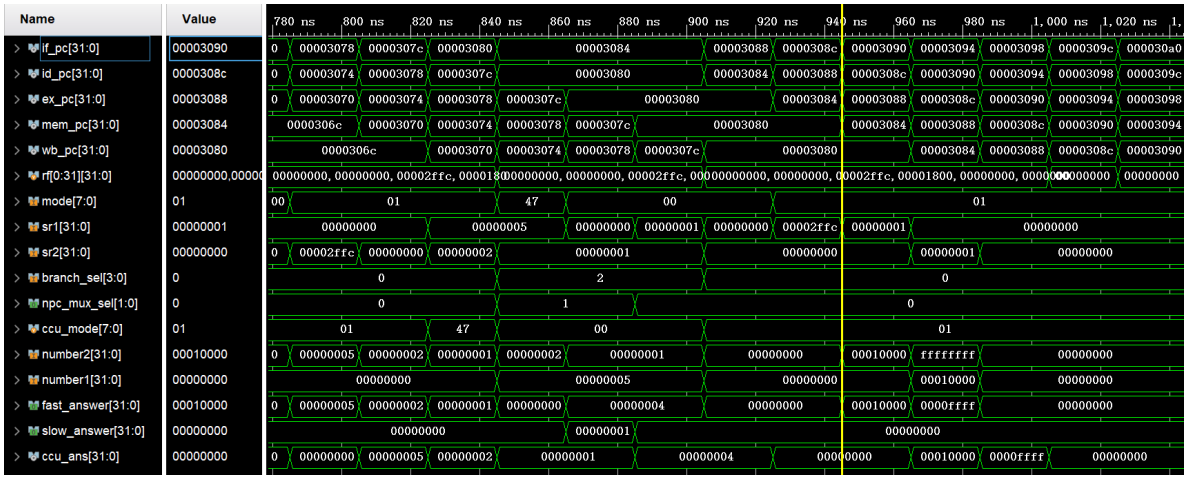
6.1.4 ALU 指令正确性测试——第二部分

编写如下的汇编程序：

```
# suppose that our former instructions are correct
# in the following test, if test fail we will show 0xffff
# otherwise we will show 0x0000
```

```
# test div
li x5, 0x5
li x6, 0x2
li x7, 0x2
div x5, x5, x6
bne x5, x7, fail

# test divu
li x5, 0x5
li x6, 0x2
li x7, 0x2
divu x5, x5, x6
bne x5, x7, fail

# test mul
li x5, 10
li x6, 11
li x7, 110
mul x5, x5, x6
bne x5, x7, fail

# test mulh
li x5, 0x7fffffff
li x6, 0x233333
li x7, 0x119999
mulh x5, x5, x6
bne x5, x7, fail

# test rem
li x5, 0x5
li x6, 0x2
li x7, 0x1
rem x5, x5, x6
bne x5, x7, fail

# test remu
li x5, 0x5
li x6, 0x2
li x7, 0x1
remu x5, x5, x6
bne x5, x7, fail

nop

fail:
li x5, 0xffff
sw x5, 0(x0)
```

测试结果如下：



其中 0x3088 为最后 Fail 处的 li 指令。可以看到，程序的执行流程是顺序的，这说明本部分指令测试完全正确。

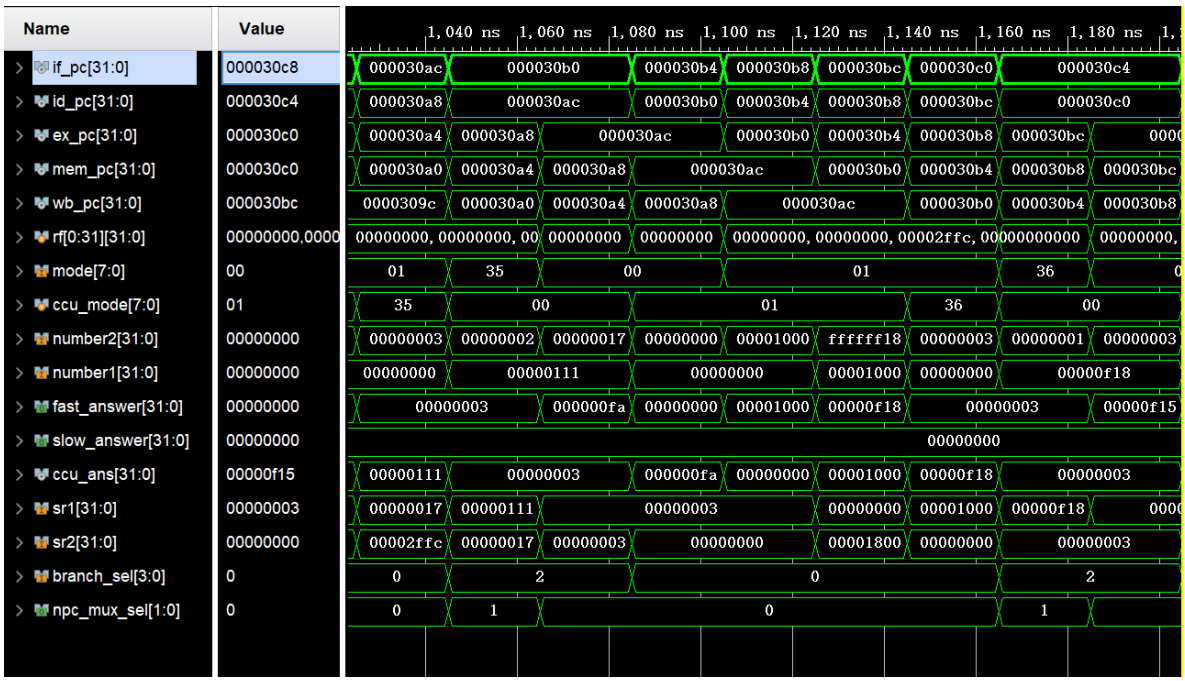
6.1.5 BALU 指令正确性测试——第一部分

编写如下的汇编程序：

```
# RISCV B扩展指令测试程序
.text
#bclr
li x5 0x0FF
li x6 0x003
li x11 0x0F7
bclr x5, x5, x6
bne x5, x11, Fail

#bclri
li x5 0x0FF
li x11 0x0F7
bclri x5, x5, 3
bne x5, x11, Fail

#bext
li x5 0x123
li x6 0x003
li x11 0x000
bext x5, x5, x6
bne x5, x11, Fail

#bexti
li x5 0x123
li x11 0x000
bexti x5, x5, 3
bne x5, x11, Fail

#binv
li x5 0x123
li x6 0x003
li x11 0x12b
binv x5, x5, x6
bne x5, x11, Fail
```

```
#binvi
li x5 0x123
li x11 0x12b
binvi x5, x5, 3
bne x5, x11, Fail

#bset
li x5 0x000
li x6 0x003
li x11 0x008
bset x5, x5, x6
bne x5, x11, Fail

#bseti
li x5 0x000
li x11 0x008
bseti x5, x5,3
bne x5, x11, Fail

#clz
li x5 0x123
li x11 0x017
clz x5, x5
bne x5, x11, Fail

#cpop
li x5 0x111
li x11 0x003
cpop x5, x5
bne x5, x11, Fail

#ctz
li x5 0xF18
li x11 0x003
ctz x5, x5
bne x5, x11, Fail


Fail:
li x10 0xFFF
```

测试结果如下：



其中 0x30c4 为最后 Fail 处的 li 指令。可以看到，程序的执行流程是顺序的，这说明本部分指令测试完全正确。

6.1.6 BALU 指令正确性测试——第二部分

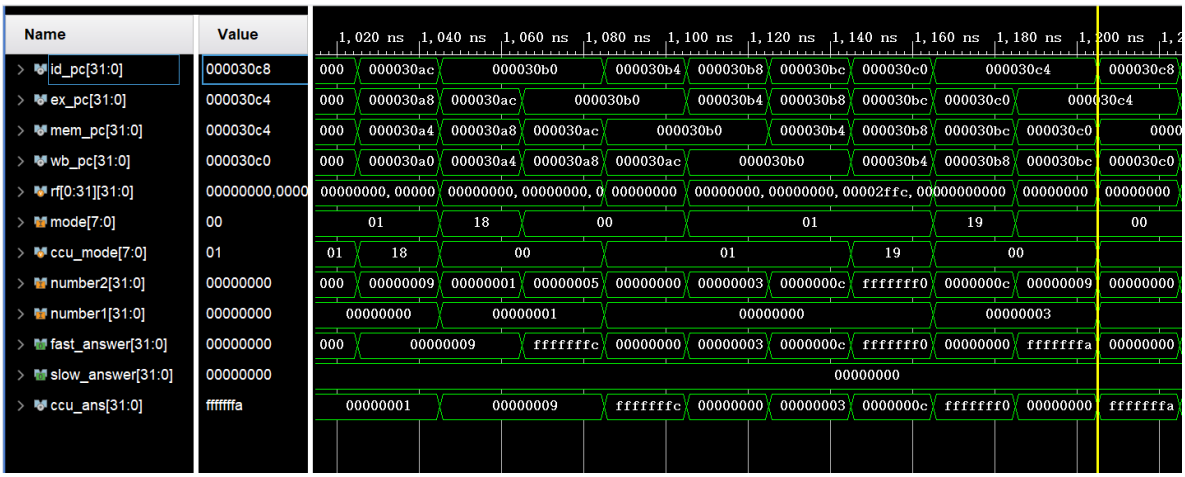
编写如下的汇编程序：

```
# suppose that our former instructions are correct
# in the following test, if test fail we will show 0xffff
# otherwise we will show 0x0000

# test andn
li x5, 0x1
li x6, 0x2
li x7, 0x1
andn x5, x5, x6
bne x5, x7, fail


# test orn
li x5, 0x1
li x6, 0x0
li x7, 0xffffffff
orn x5, x5, x6

bne x5, x7, fail


# test max
li x5, 0x1
li x6, 0x2
li x7, 0x2
max x5, x5, x6
bne x5, x7, fail
```

```
# test maxu
li x5, 0x1
li x6, 0x2
li x7, 0x2
maxu x5, x5, x6
bne x5, x7, fail


# test min
li x5, 0x2
li x6, 0x1
li x7, 0x1
min x5, x5, x6
bne x5, x7, fail


# test minu
li x5, 0x2
li x6, 0x1
li x7, 0x1
minu x5, x5, x6
bne x5, x7, fail


# test sh1add
li x5, 0x1
li x6, 0x1
li x7, 0x3
sh1add x5, x5, x6
bne x5, x7, fail


# test sh2add
li x5, 0x1
li x6, 0x1
li x7, 0x5
sh2add x5, x5, x6
bne x5, x7, fail


# test sh3add
li x5, 0x1
li x6, 0x1
li x7, 0x9
sh3add x5, x5, x6
bne x5, x7, fail

# test xnor
li x5, 0x3
li x6, 0xc
li x7, 0xfffffff0
xnor x5, x5, x6
bne x5, x7, fail

nop
fail:
li x5, 0xffff
```

```
sw x5, 0(x0)
```

测试结果如下：



其中 0x30cc 为最后 Fail 处的 li 指令。可以看到，程序的执行流程是顺序的，这说明本部分指令测试完全正确。

综上所述，RISCV 32I B M 的拓展指令的正确性可以保证。

6.2 迷宫游戏测试

测试结果详细请见演示视频。

七、总结与展望

本次实验中，我们对 CPU 进行了的功能与结构扩展，使其可以支持更多的指令，为汇编程序的编写带来了极大的便利性。换而言之，指令集的扩充在一定程度上提高了 CPU 的运行效率。

CPU 的中断与异常处理功能允许我们采用更为高效与先进的中断而不是轮询进行 I/O。这样的设计可以更好地隔离用户程序与外部输入。等待外部输入的同时，用户程序可以继续自己的进程而不是卡在原地进行等待。

CSR 的物理与虚拟实现方式允许我们自主定义所需要的 CSR，极大地丰富了硬件设计的可拓展性。

本次实验初期考虑但并没有加入的内容包括但不限于：

- RISCV 32F 浮点数扩展。浮点扩展是现代计算机几何计算与数值分析的重要一环。浮点数的引入可以大大增加程序的计算完备性。然而，由于浮点数运算需要消耗更多的时钟周期来完成，为了不影响 CPU 的性能，实现浮点运算操作需要设计单独的浮点数流水线。双流水线的引入会带来大量的额外操作与判断，短期内很难较为完整地实现。
- 迷宫的 VGA 计时与地图切换。理论上数据存储器的扩展可以让我们显示更多的地图。通过特定点的坐标判断可以实现地图的切换。这样可以极大地扩展用户体验。然而由于时间等因素的限制，我们最终没有实现这样的功能。计时器也是出于时间因素选择了数码管显示的方式。

希望我们的这次实验可以为后来的学弟学妹们提供参考，也以本次实验作为 2022 春季学期计算机组成原理(H) 的完美收官。