数字逻辑与处理器多周期处理器作业报告

2019011008 无 92 刘雪枫

文件说明：

InstAndDataMemory_1.v：存放“汇编程序分析-1”的代码；

InstAndDataMemory_setsub.v：存放 setsub 测试代码；

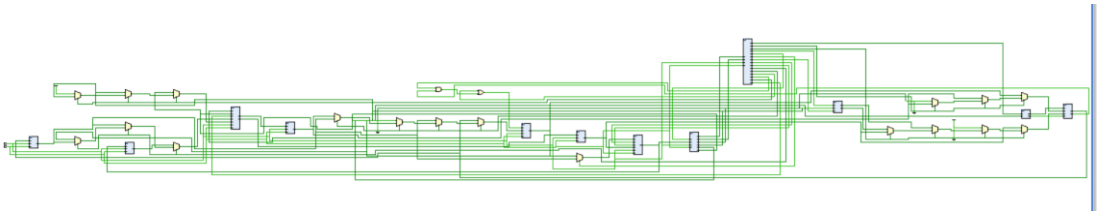
InstAndDataMemory_2.v：存放“汇编程序分析-2”的代码；

InstAndDataMemory_err.v：存放溢出测试代码

上述四个文件不能一起放入工程中，需要将执行的代码复制到 InstAndDataMemory.v 中。

1. 数据通路设计

编写代码见附件，Vivado 显示的数据通路如下：



包含的多路选择器有：

(1) PCSource 控制的多路选择器，用于选择 PC 的下一个值

(2) ALUSrcA 和 ALUSrcB 控制的多路选择器，用于选择 ALU 两个操作数的来源

(3) RegDst 控制的多路选择器，用于选择写入的寄存器

(4) MemtoReg 控制的多路选择器，用于选择写入寄存器的数据

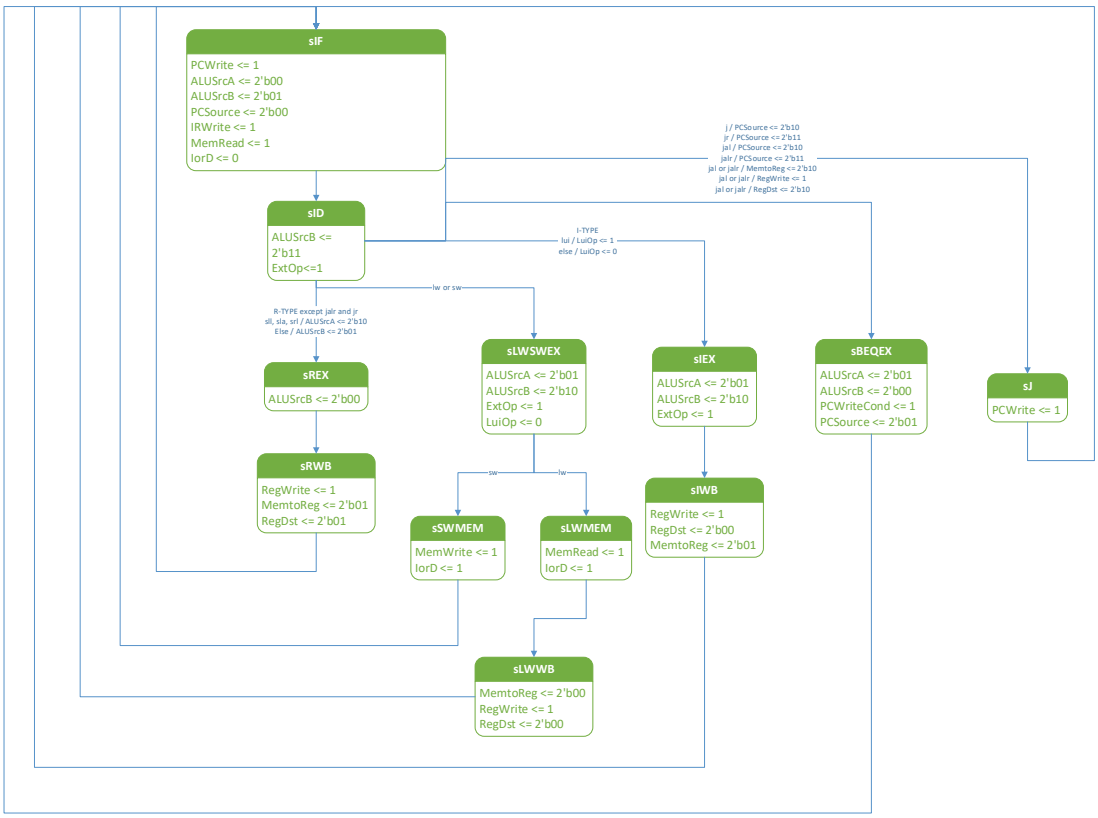
(5) IorD 控制的多路选择器，用于选择读取或写入指令存储器还是数据存储器

包含的寄存器有：

指令寄存器 InstReg 用于存储指令、MemDataReg 用于储存上一个周期读到的存储器中的数据、Read_data_A_Reg 与 Read_data_B_Reg 用于储存寄存器堆

中读到的两个寄存器的值、ALUOut 寄存器用于存储上一次 ALU 计算得到的结果

2. 控制信号分析与有限状态机实现
   a) 2.1 控制信号具体功能
      (1) PCWrite 控制 PC 寄存器是否可以写入
      (2) PCWriteCond 控制是否是比较分支指令写入 PC 寄存器的阶段
      (3) IorD 控制当前读写的是指令还是数据
      (4) MemWrite 控制是否写入可以存储器
      (5) MemRead 控制是否可以从存储器中读数据
      (6) IRWrite 控制指令存储器是否可以写入
      (7) MemtoReg 控制写入寄存器堆的数据来源
      (8) RegDst 控制写入的寄存器编号的来源
      (9) ExtOp 控制立即数是否是有符号运算
      (10) LuiOp 控制该立即数运算是否是 lui 指令
      (11) ALUSrcA 和 ALUSrcB 控制 ALU 运算的两个操作数的来源
      (12) ALUOp 控制 ALU 进行的运算
      (13) PCSource 控制要写入 PC 寄存器的值的来源
   b) 2.2 状态转移图中用到的指令说明：
      R-TYPE except jalr and jr：add, addu, sub, subu, and, or, xor, nor, sll, srl, sra, slt, sltu
      I-TYPE：lui, addi, addiu, andi, sltiu
      unsigned: addiu, sltiu
      状态转移图如下：



c) 2.3 代码编写见附件

3. ALU 功能拓展

a) 3.1 增加 setsub 指令：将其作为 r 型指令，功能吗 Funct 定义为 0x19 即可

b) 3.2 画出真值表：

A	B	Result
0	0	0
0	1	0
1	0	1
1	1	0

结果 Result=AB’

c) 3.3 汇编程序如下：

```
lui $t0, 0xabcd
addiu $t0, $t0, 0x1234
lui $t1, 0xcdef
addiu $t1, $t1, 0x3456
```

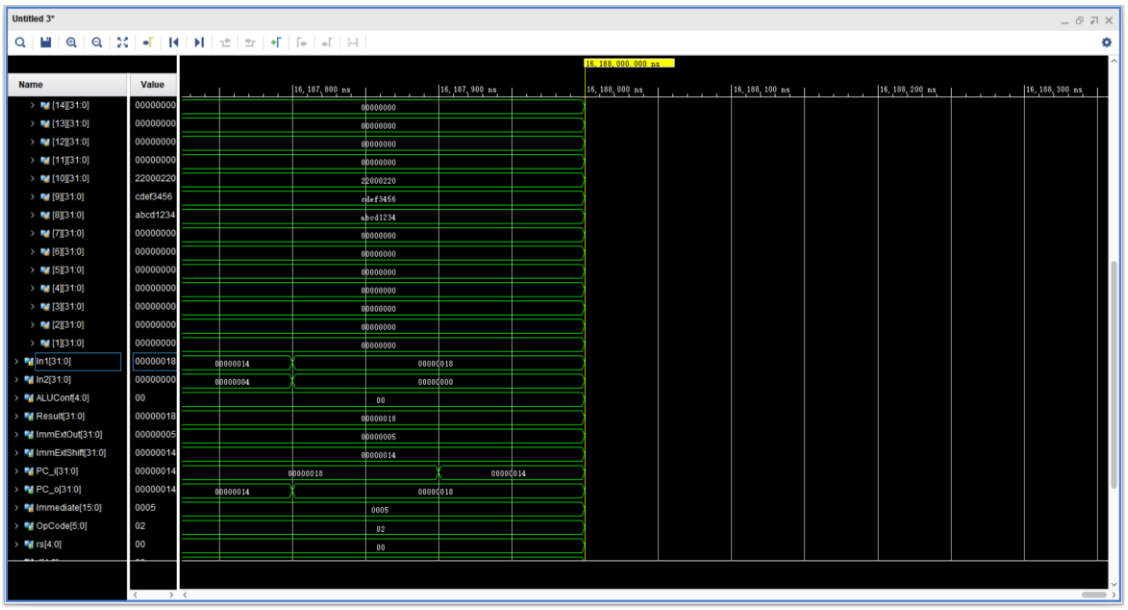
setsub $t2, $t0, $t1

Loop:

j Loop

该程序已经放入 InstAndDataMemory_setsub.v 文件当中。

翻译为机器码并进行行为级仿真，得到结果：



可以看到确实得到了正确的结果 0x22000220

4. 汇编程序分析-1

程序放入 InstAndDataMemory_1.v 文件当中

a) 该程序计算过程：

先将$a0 置为 0x00002f5b，再将$a1 置为 0xffffcfc7，然后计算$a2=$a1<<16，即$a2=0xcfc70000，然后计算算术右移$a3=$a2>>16，即$a3=0xffffcfc7。然后比较$a3 与$a1 的值，两者相等，因此分支。

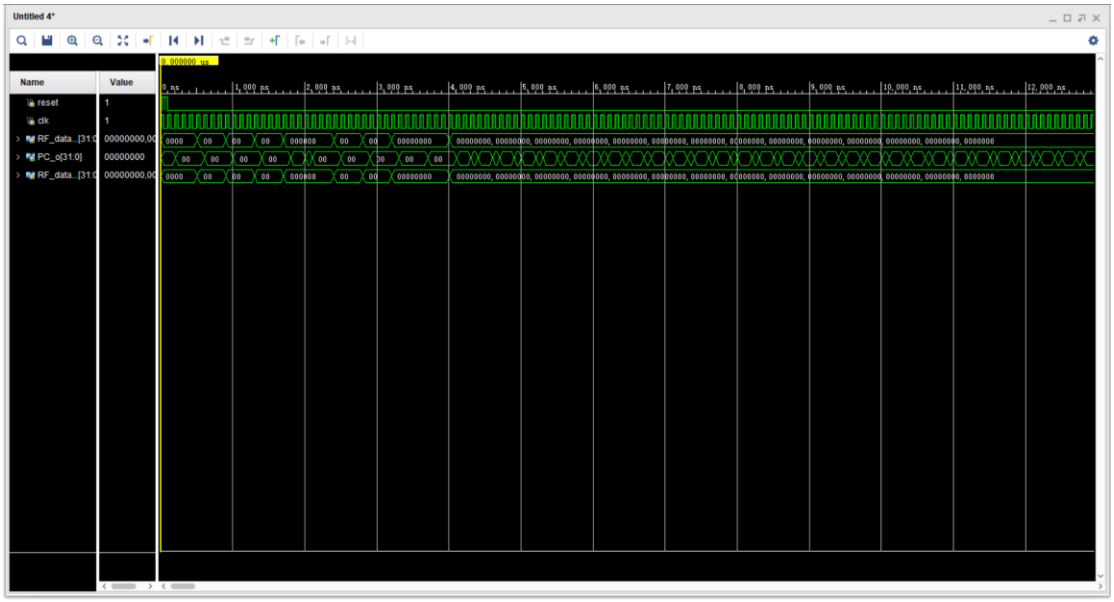
计算$t0=$a2+$a0=0xcfc72f5b

计算算术右移$t0=t0>>8=0xffcfc72f，计算$t2=0xffffd0a5

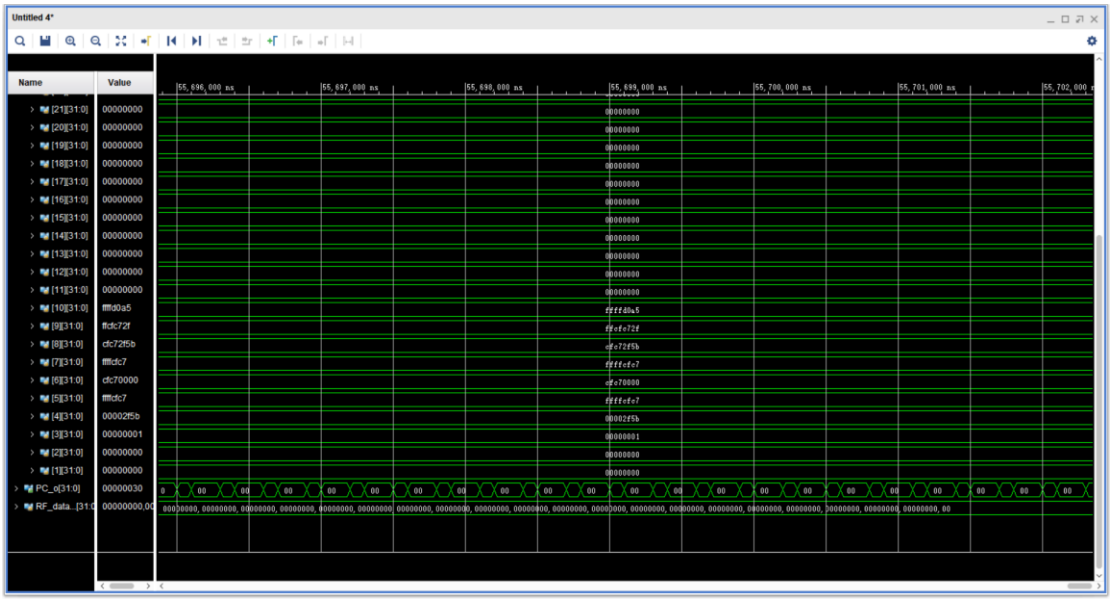
此时$a0==0x00002f5b，而$t2==0xffffd0a5。比较$a0<$t2，有符号比较为假，将$v0 置为 0；无符号比较为真，将$v1 置为 1

b) 进行行为级仿真，得到结果：

总体结果：



最终稳定时结果：



可以看到寄存器内的值均为期望得到的值，仿真正确。

5. 汇编程序分析-2

a) 作用是计算从 1~n 的和的二倍，储存在$v0 寄存器中

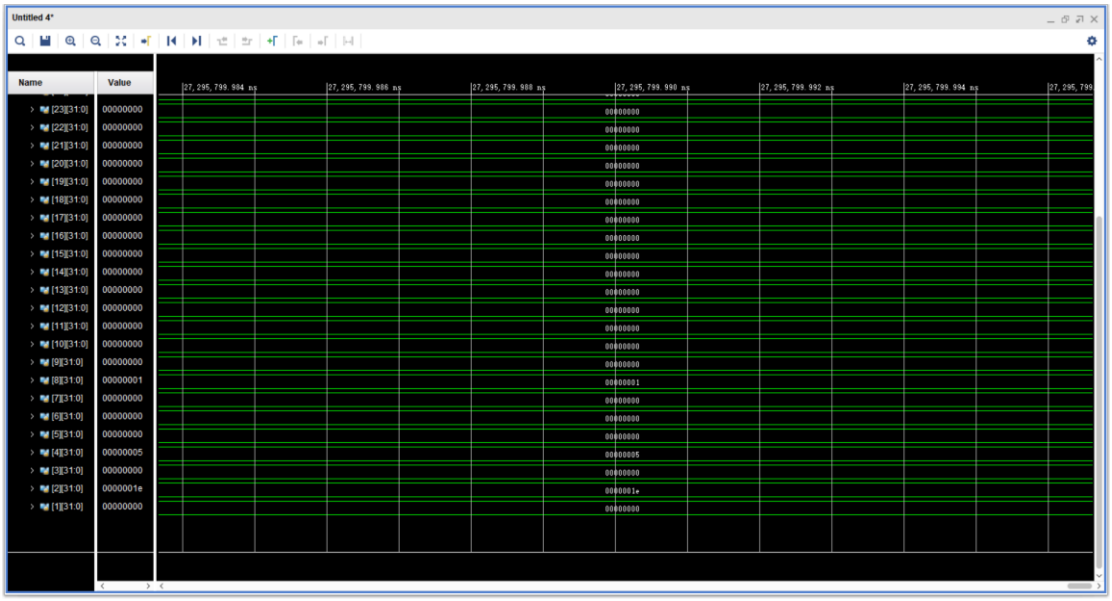
程序注释如下：

```
addi $a0, $zero, 5          # $a0 = 5
xor $v0, $zero, $zero       # $v0 = 0
jal sum                     # call function sum: save $ra
Loop:
```

```
        beq $zero, $zero, Loop    # loop forever
sum:
        addi $sp, $sp, -8          # push stack
        sw $ra, 4($sp)              # save $ra
        sw $a0, 0($sp)              # save parameter $a0
        slti $t0, $a0, 1           # if ($a0 < 1) $t0 = 1; else $t0 = 0
        beq $t0, $zero, L1         # if ($t0 == 0) (means $a0 >= 1) goto L1
        addi $sp, $sp, 8           # pop stack
        jr $ra                      # return $v0
L1:
        add $v0, $a0, $v0                  # $v0 = $a0 + $v0
        addi $a0, $a0, -1                  # parameter $a0 = $a0 - 1
        jal sum                             # call sum
        lw $a0, 0($sp)                     # restore $a0
        lw $ra, 4($sp)                     # restore $ra
        addi $sp, $sp, 8                   # pop stack
        add $v0, $a0, $v0                  # return value: $v0 + $a0
    jr $ra                                 # return
```

b) 5.2 汇编翻译成机器码后已经附在 InstAndDataMemory_2.v 文件当中

c) 5.3 进行行为级仿真，得到最终结果如下：



结果$v0 的结果是 0x1e，即十进制的 30；$a0 的结果是 5，均符合预期

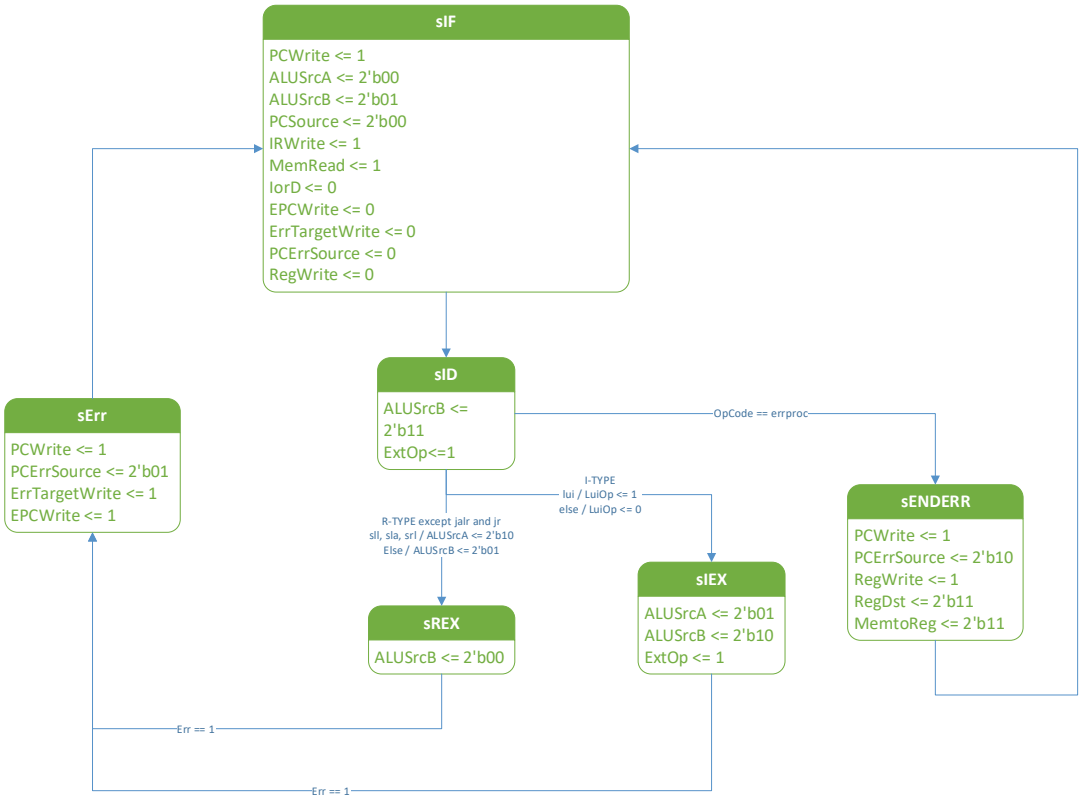
d) 5.4 PC 的变化总体上是每次递增 4 的，但是每次调用 sum 时，PC 都会先递增 4 后再改变为 sum 的地址，即函数调用，此外运行到 beq 指令时，在前 5 次函数调用时，PC 也会进行写入，跳转到 L1。$a0 寄存器先是从 5 递减到 0，再从 0 逐渐增加回 5。$v0 寄存器则是逐渐递增的：每次加 5、4、3、2、1、2、3、4、5，最终到达 30；$sp 寄存器的值显示逐渐减小，即每次递归调用时进行压栈，然后逐渐增大，即函数退出时退栈。$ra 最开始写入时地址为 Loop 处的地址值，即 0x0000000c，此时是第一次调用 sum，然后变成 sum 处的地址值，即 0x00000038，这是因为之后的每次调用（jal 指令）都是 sum 函数中的，然后又变回 0x0000000c，这时是最先调用的 sum 函数恢复了$ra 寄存器。

6. 异常处理

a) 6.1 控制信号：EPCWrite 控制 EPC 是否可以写入；ErrTargetWrite 控制 ErrorTarget 是否可以写入

b) 6.2 指令设计：errproc 的 OpCode 为 1，其余均为 0

状态转移图：



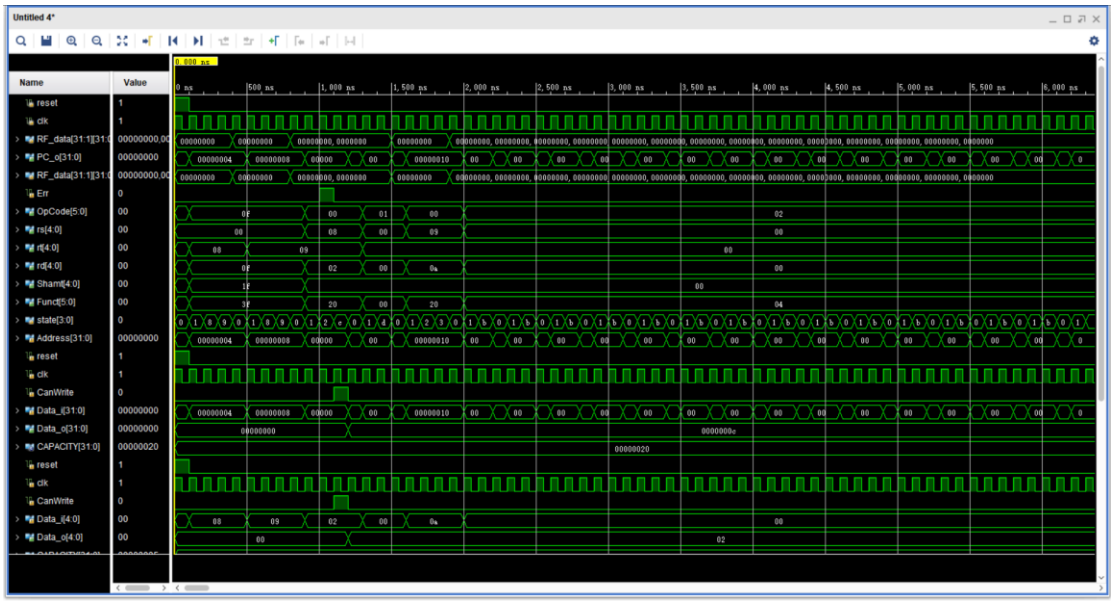
c) 6.3 应加入的模块已经编写进代码中。编写的汇编测试程序为：

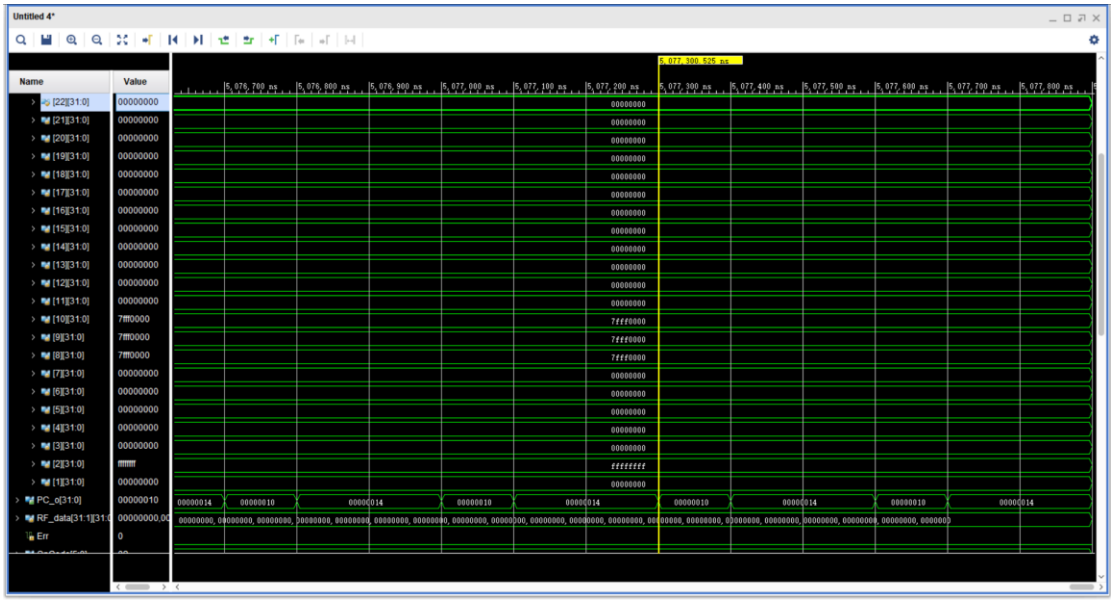
```
        lui $t0, 0x7fff
        lui $t1, 0x7fff
        add $v0, $t0, $t1
        add $t2, $t1, $zero
Loop:
    j Loop
……
    errproc      # 异常处理程序
```

该程序第三行应溢出，因此$v0 的值应为 0xffffffff，程序到达死循环后，$t0、$t1、$t2 的值均应为 0x7fff0000，而$v0 的值应为 0xffffffff

转为机器码后的程序存放在 InstAndDataMemory_err.v 中。

仿真后，得到结果：





可以看到，得到的寄存器中的值与预期的完全一致，因此确实实现了异常处理功能。