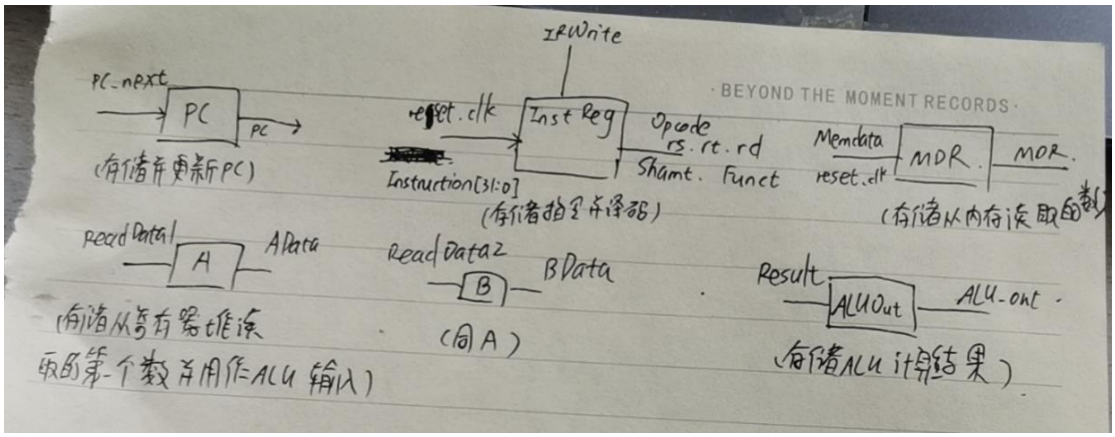
多周期大作业实验报告

Xuan

1.1 将上述基础功能电路模块进行实例化和连接，完成多周期处理器的整体数 据通路设计，即完成 MultiCycleCPU.v 中相关 Verilog 代码的编写。请写 出数据通路中包含的寄存器（除寄存器堆之外）和多路选择器，并简要说 明它们的功能。

MultiCycleCPU.v 见文件

寄存器及其功能如下



代码：PC PC1(.reset(reset), .clk(clk), .PCWrite(PCWrite_all), .PC_i(PC_next), .PC_o(PC));

InstRegInstReg1(.reset(reset), .clk(clk), .IRWrite(IRWrite), .Instruction(Mem_data), .OpCode(OpCode), .rs(rs), .rt(rt), .rd(rd),.Shamt(Shamt), .Funct(Funct));

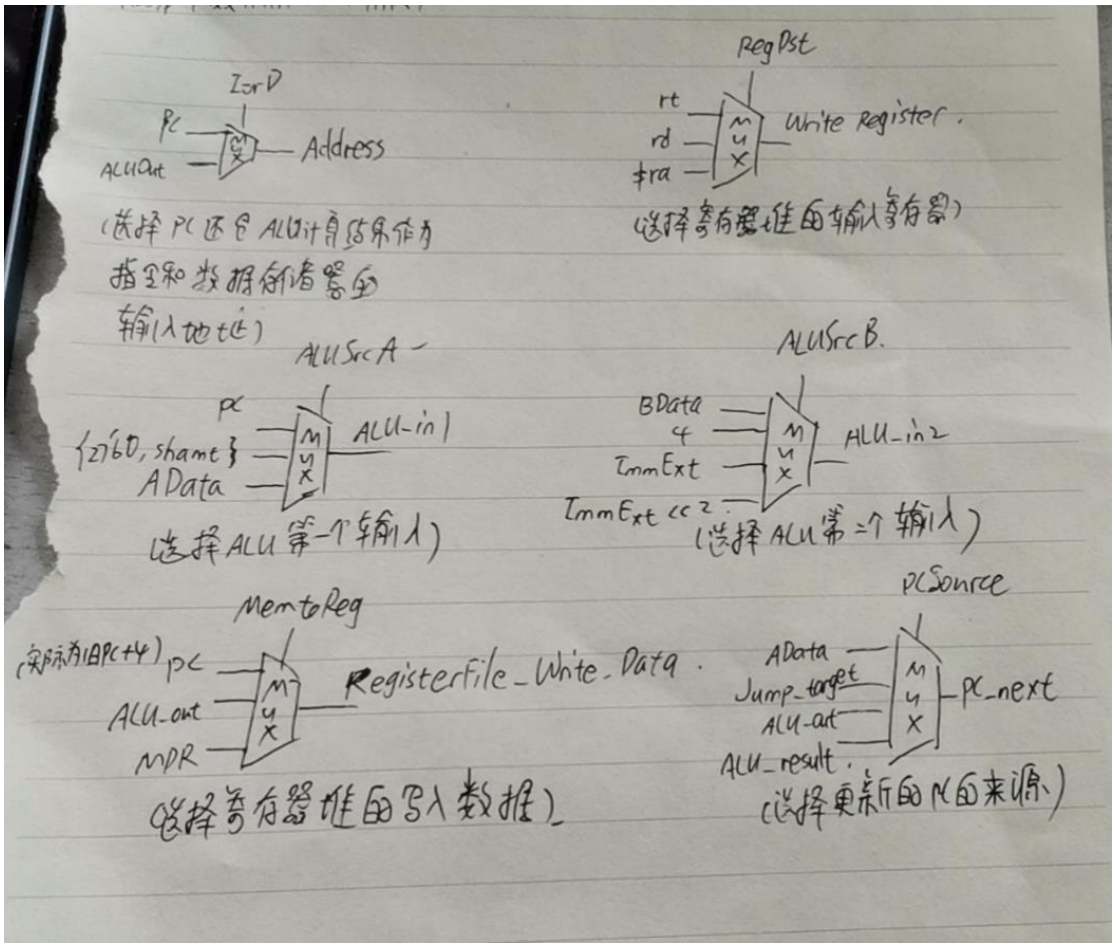
RegTemp MemoryReg(.reset(reset), .clk(clk), .Data_i(Mem_data), .Data_o(MDR));

RegTemp AReg(.reset(reset), .clk(clk), .Data_i(Read_data1), .Data_o(AData));

RegTemp BReg(.reset(reset), .clk(clk), .Data_i(Read_data2), .Data_o(BData));

RegTemp ALUOUTReg(.reset(reset), .clk(clk), .Data_i(ALU_Result), .Data_o(ALU_out));

多路选择器及其功能如下



代码：assign Address=(IorD==0)?PC:ALU_out;//PC 后的二路选择器

assign Write_register = (RegDst == 2'b00)? rt: (RegDst == 2'b01)? rd: 5'b11111;

assign ALU_in1 = (ALUSrcA==2'b10)? {27'b0, Shamt}: ((ALUSrcA==2'b01)?AData:PC);//ALU 第一输入前的三路选择器

assign ALU_in2 = (ALUSrcB==2'b11)?ImmExtShift: ((ALUSrcB==2'b10)?ImmExtOut: //ALU 第二输入前的四路选择器

assign RFWrite_data=(MemtoReg==2'b10)?PC:((MemtoReg==2'b01)?ALU_out:MDR);//寄存器堆写入数据前的多路选择器,ID 阶段 PC+4 写入$ra

assign PC_next = (PCSource == 2'b00)? ALU_Result: ((PCSource == 2'b01)?ALU_out:((PCSource == 2'b10)?Jump_target: AData));//新的 PC 的来源的选择信号

2.1 基于上述设计的数据通路和 Controller.v 中的控制信号，请写出每个控制 信号所实现的具体控制功能。

PCWrite：PC 寄存器的写使能信号；0-不能写 PC；1-允许写 PC。

PCWriteCond:Branch 指令对 PC 寄存器的写使能信号；0-不是 Branch 指令；1-是 Branch 指令

IorD:控制PC后的二路选择器的选择信号;0-内存的address=PC;1-内存的address=ALUOut;

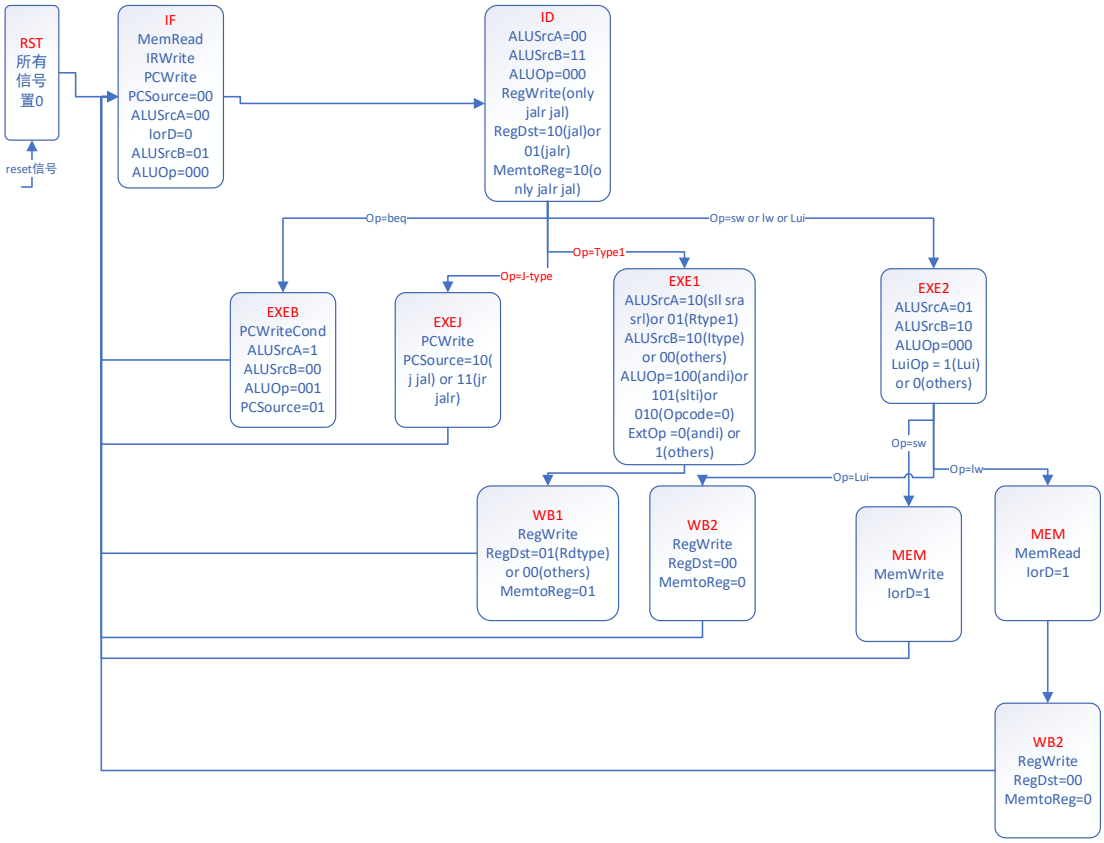
MemWrite:指令和数据内存的写使能信号；0-不能写入 write data;1-允许写入

MemRead：指令和数据内存的读使能信号；0-Mem_data=0 即不能读取内存数据;1-允许读取内存数据

IRWrite：指令寄存器的写使能信号；0-不能写指令寄存器,防止读内存时更改了指令；1-允

许写指令寄存器。
MemtoReg[1:0]:控制寄存器堆的写入数据的选择信号；
10-PC+4; (jal 和 jalr 指令时为 PC+4) 01-ALUOut;else--MDR;
RegDst [1:0]：控制寄存器堆的写入寄存器的选择信号；00-rt;01-rd;else-$ra;
RegWrite：寄存器堆的写使能信号；0-不能写入寄存器堆；1-允许写寄存器堆。
ExtOp：符号拓展信号；1-有符号拓展；0-零拓展
LuiOp：lui 指令的移位信号；1-是 lui 指令，拓展和移位单元会输出未拓展的立即数的左移 16 位；0-不是 lui 指令，拓展和移位单元会输出拓展后的立即数
ALUSrcA[1:0] :alu 第一个输入前的多路选择器的选择信号； 10: 5bit 移位量的 32bit 拓展（用于 sll,sra 等指令）; 01-寄存器 A 的数据；else-PC;
ALUSrcB[1:0]: alu 第二个输入前的多路选择器的选择信号；00-寄存器 B 的数据；01-4；（用于计算 PC+4）；10-拓展后的立即数（用于 I 型和 lui 指令等）；11-拓展后的立即数左移两位（用于计算 beq 地址即 ALUOut <= PC + (sign-extended(IR[15:0] << 2)）
ALUOp[3:0]:选择 ALU 进行的操作；3'b001-aluSUB; 3'b100-aluAND;3'b101-aluSLT;3'b010-aluFunct（即 R 型指令，还要根据 Funct 来确定）;else-aluADD
PCSource:选择新的 PC 的来源的选择信号；00-PC+4 即 ALU 的 result;01-ALUOut 寄存器的数据（用于 beq 指令）；10- {PC[31:28], IR[25:0], 2'b00}（用于 j 和 jal）；11-AData(即$rs 的地址数据，用于 jr 和 jalr 指令)

2.2 依据对 MIPS 指令集和多周期处理器的理解，完成控制器 Controller 中有 限状态机的设计，并画出状态转移图。要求状态转移列出具体的指令类型 （合并的指令，比如 R-type，需要提前列出所有的 R-type 指令），并且状 态图中列出最关键的控制信号。
Type1:add addu sub subu addi addiu and or xor nor andi sll srl sra slt sltu slti sltiu setsub
J-type: j jal jr jalr
Rtype1: add addu sub subu and or xor nor slt sltu jr jalr setsub
Itype：lw sw lui addi addiu andi slti sltiu
Rdtype:add addu sub subu and or xor nor sll srl sra slti sltiu jalr setsub
默认（即没在状态中写出的信号）ALUSrcA=01 PCSource=00 ALUSrcB=00 ExtOp=1 LuiOp=0 RegWrite=0 RegDst=00 MemtoReg=00 IRWrite=0 MemRead=0 MemWrite=0 IorD=0 PCWriteCond=0 PCWrite=0 ALUOp=000
状态图如下：



2.3 基于上述设计的有限状态机和控制信号分析，完成控制器 Controller.v 中 相关 Verilog 代码的编写。

Controller.v 见文件

3 ALU 功能拓展 对于一个元素个数为 32 的全集，使用如下数据格式表征一个集合：对全集中的 32 个元素进行序号分配（0-31），使用 32-bit 寄存器 S 表示某集合，S 中的第 i 位表示该集合是否包含第 i 个元素（0 为不包含，1 为包含）。下面将修改 ALU 电 路实现如下功能：输入集合 A 与集合 B（输入数据存储在寄存器中），计算两个集 合的差，即 A-B。

3.1 假设该指令为 setsub，请说明该指令的类型和并自行定义对应的机器码字 段内容（与现有指令不冲突即可）。

R 型指令，setsub $rd,$rs,$rt($rs 为 A,$rt 为 B)

OpCode=6'b0,Shamt=5'b0,Funct=0x28,则机器码为

{6'b0,rs,rt,rd,5'b0,6'h0x28}

3.2 阅读 ALU.v、ALUControl.v 和 Controller.v，请写出你的设计思路，并完 成 ALU 电路模块中相关的 Verilog 代码修改。

因为 A-B=A 交(~B) 所以直接在 ALU.v 加 Result<= In1&(~In2);//setsub

因为是 R 型指令，所以 Controller.v 中仍然 ALUOp[2:0] = 3'b010;//have Funct

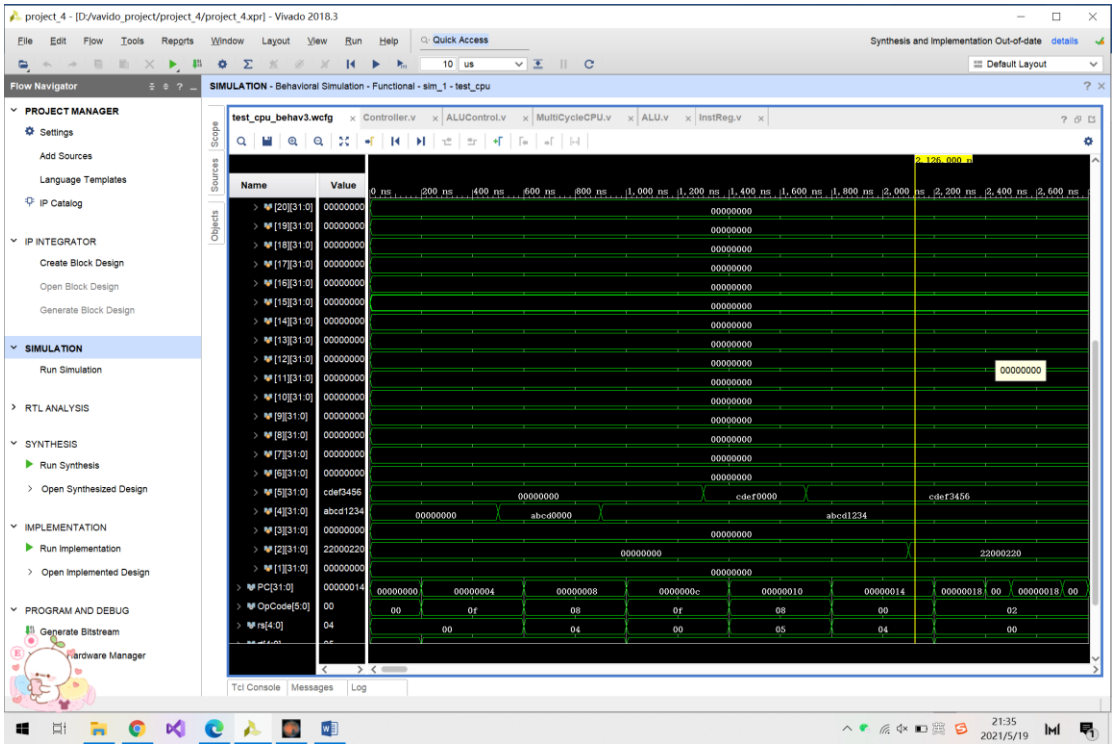
还要相应修改 Rdtype 和 Rtype1

assign

Rdtype=OpCode==6'h00&&(Funct==6'h20||Funct==6'h21||Funct==6'h22||Funct==6'h23||Funct==6'h24||Funct==6'h25

||Funct==6'h26||Funct==6'h27||Funct==6'h2a||Funct==6'h2b||Funct==6'h00||Funct==6'h02||

Funct==6'h03||Funct==6'h09||Funct==6'h28);
assign
Rtype1=OpCode==6'h00&&(Funct==6'h20||Funct==6'h21||Funct==6'h22||Funct==6'h23||Funct==6'h24||Funct==6'h25
||Funct==6'h26||Funct==6'h27||Funct==6'h2a||Funct==6'h2b||Funct==6'h08||Funct==6'h09||Funct==6'h28);
但要在 ALUControl.v 中 Funct 的选择中加 6'b10_1000: aluFunct <= aluSETSUB;//setsub
ALU.v、ALUControl.v 和 Controller.v 见 code 中的文件
3.3 假设输入集合 A 为 0xABCD1234，输入集合 B 为 0xCDEF3456，请自行设计测试指令序列并修改指令存储器文件(InstAndDataMemory.v)，使用 ModelSim 或 Vivado 等仿真软件进行仿真，给出仿真结果示例以说明 ALU 功能拓展的 正确性。
装入 32 位立即数需两条指令（lui 和 addi） 设计的指令如下：
lui $a0, 0xABCD
addi $a0, $a0, 0x1234
lui $a1, 0xCDEF
addi $a1, $a1, 0x3456
setsub $v0 $a0 $a1
Loop:j Loop
机器码为：（修改后的 InstAndDataMemory.v 为 code 中的 InstAndDataMemory3.v）
3c04abcd
20841234
3c05cdef
20a53456
00851028
08000005
A= 10101011110011010001001000110100
B= 11001101111011110011010001010110
结果应为 00100010000000000000001000100000=0x22000220 与仿真结果相符
仿真结果如下（RF_data[2]=0x22000220）



4 汇编程序分析-1

4.1 这段程序运行足够长时间后，寄存器$a0 到$a3，$t0 到 t2，$v0 到$v1 中的 值应该是多少？写出计算过程。

$a0=$zero+0x2f5b =0x2f5b(12123);$a1=$zero +0xffffcfc7=0xffffcfc7(-12345)

$a2=$a1<<16=0xcfc70000;$a3=$a2>>16=0xffffcfc7(算术右移)

因为$a3==$a1,beq 指令执行后跳到 L1：add $t0, $a2, $a0

$t0=$a2+$a0=0xcfc72f5b

$t1=$t0>>8=0xffcfc72f(算术右移)

$t2=$zero+0xffffd0a5=0xffffd0a5(-12123)

因为$a0>$t2(有符号比较),所以 slt 指令后$v0=0;

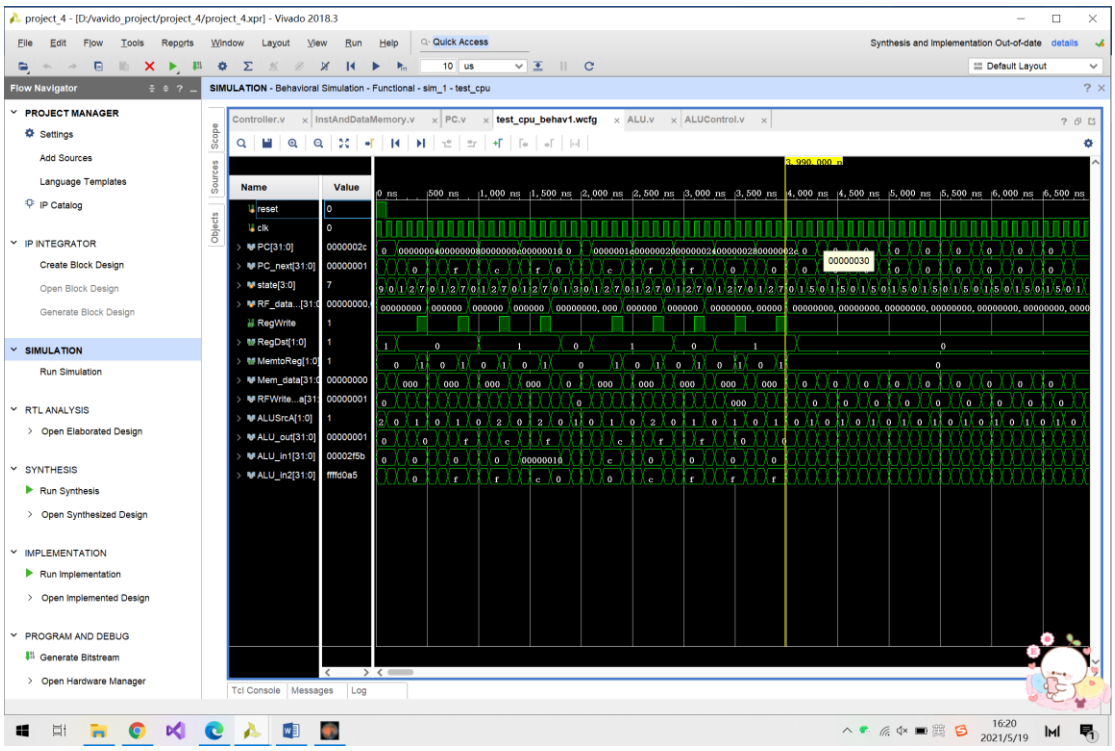
因为$a0<$t2(无符号比较),所以 sltu 指令后$v1=1;

4.2 使用 ModelSim 或 Vivado 等仿真软件进行仿真，顶层仿真模块为 test_cpu.v。请给出仿真截图，验证计算结果和所设计的多周期处理器的 功能正确性。

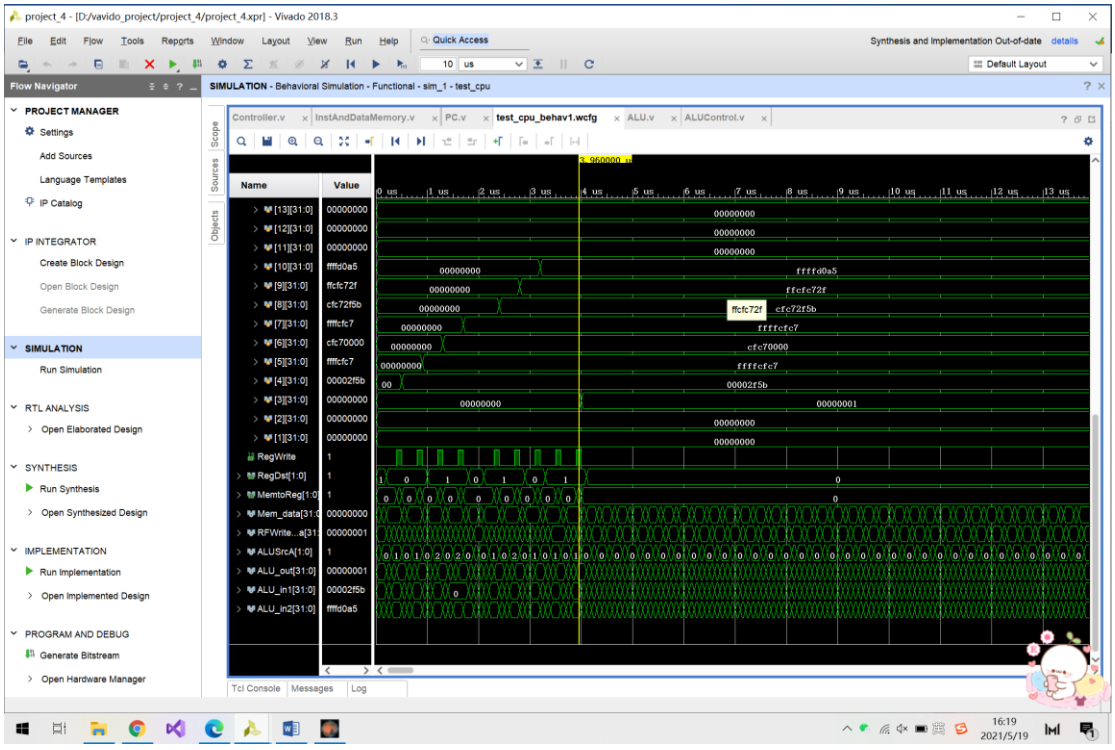
PC 寄存器和状态的变化如下图（光标处为 j Loop）

（InstAndDataMemory.v 为 code 中的 InstAndDataMemory1.v）

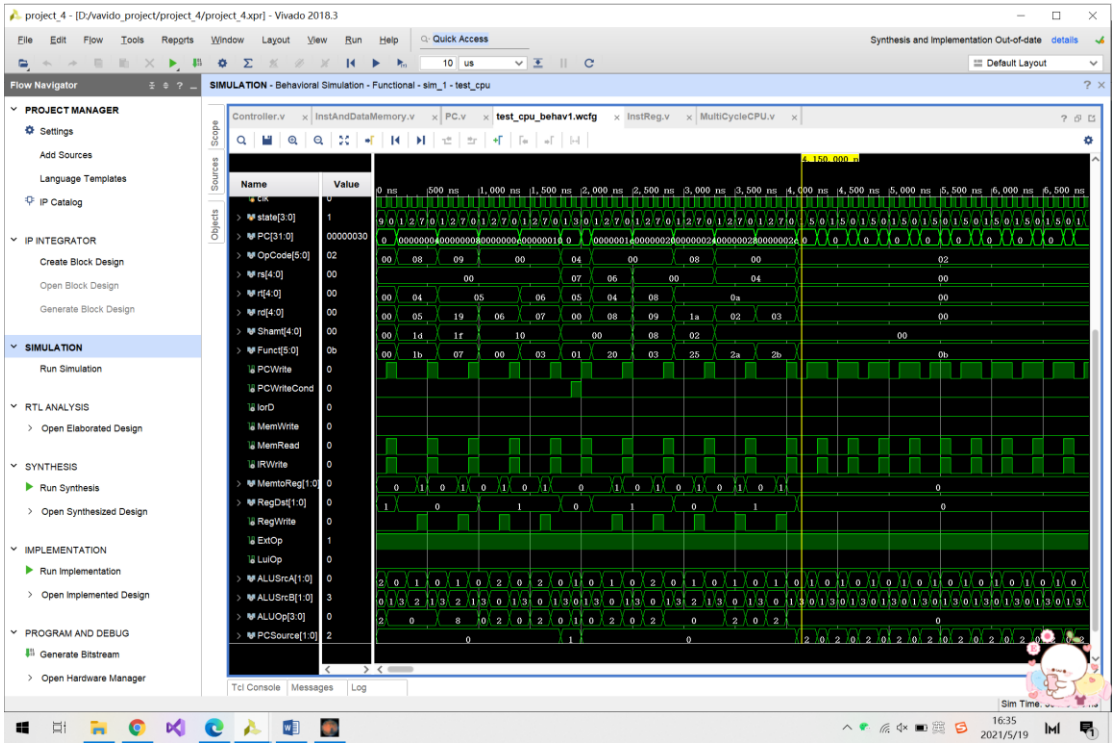
仿真波形见 test_cpu_behav1.wcfg



寄存器堆内各寄存器的变化情况如下图，与 mars 运行结果一致，最终$v0=0,$v1=1;然后循环 j Loop 指令



指令、状态和控制信号变化情况如下



5 汇编程序分析-2

5.1 如果第一行的 5 是任意正整数 n，这段程序能实现什么功能？Loop，sum， L1 各有什么作用？为每一句代码添加注释。

能实现计算 n(n+1)的功能；（$v0=n(n+1)）

L1 可通过$v0=n+(n-1)+…+1+1+2+…+n=n(n+1)实现 n(n+1)

Sum 可判断 n 是否等于 0，作为 n=0 时的 return 0

Loop:表示执行完最后一句程序

addi $a0, $zero, 5#输入参数 n

xor $v0, $zero, $zero#初始化$v0

jal sum#跳回来时表示程序执行结束 jump to sum $ra points Loop

Loop:

beq $zero, $zero, Loop#程序结束

sum:

addi $sp, $sp, -8#分配栈空间

sw $ra, 4($sp)#save $ra in ($sp)[4]

sw $a0, 0($sp)# save $a0 in ($sp)[0]

#保护现场

slti $t0, $a0, 1#if($a0<1)$t0=1

beq $t0, $zero, L1#if($a0>=1) jump to L1

addi $sp, $sp, 8#恢复栈空间

jr $ra#n=0 时的 return

L1:

add $v0, $a0, $v0#sum+=n

addi $a0, $a0, -1#n-1

jal sum#f(n-1)

lw $a0, 0($sp)

lw $ra, 4($sp)#恢复现场
addi $sp, $sp, 8#恢复栈空间
add $v0, $a0, $v0#sum=f(n-1)+n;
jr $ra#return sum
5.2 将这段汇编翻译成机器码并写出，完成 InstAndDataMemory.v 的修改。
机器码为
20040005 addi
00001026 xor
0c000004 jal
1000ffff beq
23bdfff8 addi
afbf0004
afa40000
28880001
11000002 beq
23bd0008
03e00008
00821020
2084ffff
0c000004
8fa40000
8fbf0004
23bd0008
00821020
03e00008
修改后的 InstAndDataMemory.v 为 code 中的 InstAndDataMemory2.v，如下

```
module InstAndDataMemory(reset, clk, Address, Write_data, MemRead, MemWrite, Mem_data);
    //Input Clock Signals
    input reset;
    input clk;
    //Input Data Signals
    input [31:0] Address;
    input [31:0] Write_data;
    //Input Control Signals
    input MemRead;
    input MemWrite;
    //Output Data
    output [31:0] Mem_data;

    parameter RAM_SIZE = 256;
    parameter RAM_SIZE_BIT = 8;
    parameter RAM_INST_SIZE = 32;
```

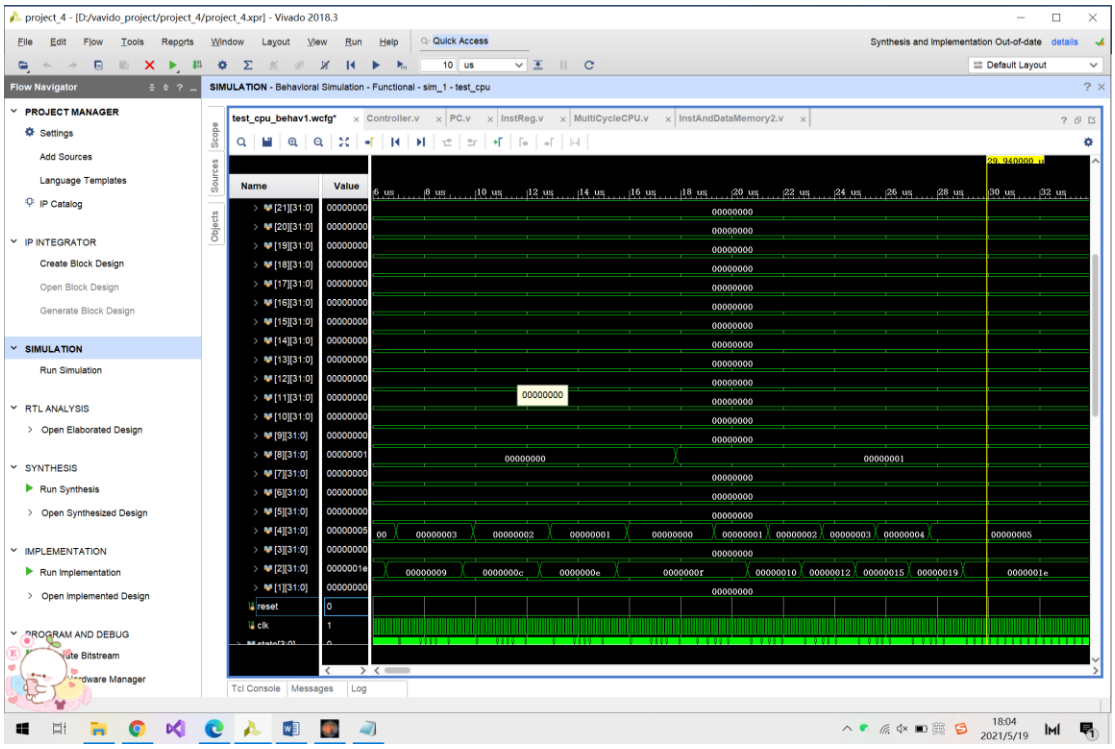
```
    reg [31:0] RAM_data[RAM_SIZE - 1: 0];

    //read data
    assign Mem_data = MemRead? RAM_data[Address[RAM_SIZE_BIT + 1:2]]: 32'h00000000;

    //write data
    integer i;
    always @(posedge reset or posedge clk) begin
        if (reset) begin
            // init instruction memory
            // jal 机器码要减去 0x00100000,beq 不用
            RAM_data[8'd0] <= 32'h20040005;
            RAM_data[8'd1] <= 32'h00001026;
            RAM_data[8'd2] <= 32'h0c000004;
            RAM_data[8'd3] <= 32'h1000ffff;
            RAM_data[8'd4] <= 32'h23bdfff8;
            RAM_data[8'd5] <= 32'hafbf0004;
            RAM_data[8'd6] <= 32'hafa40000;
            RAM_data[8'd7] <= 32'h28880001;
            RAM_data[8'd8] <= 32'h11000002;
            RAM_data[8'd9] <= 32'h23bd0008;
            RAM_data[8'd10] <= 32'h03e00008;
            RAM_data[8'd11] <= 32'h00821020;
            RAM_data[8'd12] <= 32'h2084ffff;
            RAM_data[8'd13] <= 32'h0c000004;
            RAM_data[8'd14] <= 32'h8fa40000;
            RAM_data[8'd15] <= 32'h8fbf0004;
            RAM_data[8'd16] <= 32'h23bd0008;
            RAM_data[8'd17] <= 32'h00821020;
            RAM_data[8'd18] <= 32'h03e00008;
            //init instruction memory
            //reset data memory
            for (i = RAM_INST_SIZE - 1; i < RAM_SIZE; i = i + 1)
                RAM_data[i] <= 32'h00000000;
        end else if (MemWrite) begin
            RAM_data[Address[RAM_SIZE_BIT + 1:2]] <= Write_data;
        end
    end

endmodule
```

5.3 使用 ModelSim 或 Vivado 等仿真软件进行仿真。运行足够长时间后，寄存 器$a0，$v0 的值是多少？和你预期的程序功能是否一致？

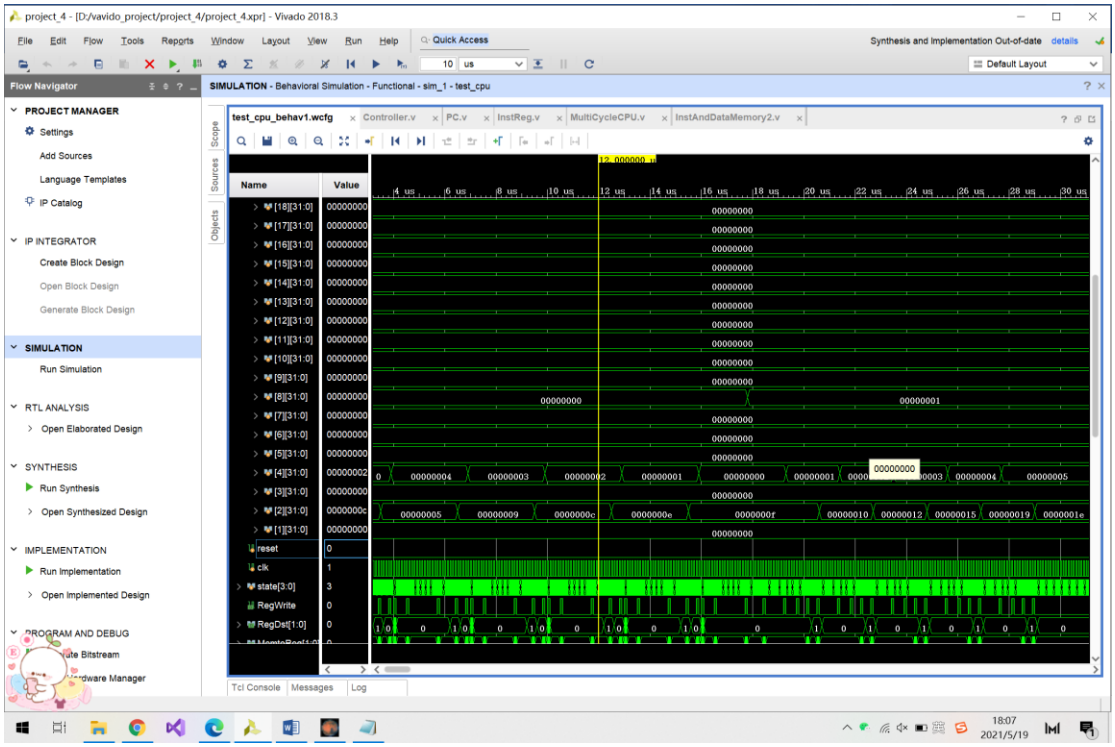
$a0=5,$v0=0x1e=30=5*（5+1）,与预期一致。

如图，最终结果为光标处的$2=0x1e,$4=5;$t0=1

仿真波形见 test_cpu_behav2.wcfg



5.4 观察、描述并解释 PC（只考虑 PC 寄存器写入时的变化）、$a0、$v0、$sp、 $ra 如何变化。



$v0 变化：0->5->9->12->14->15->16->18->21->25->30;

因为等价的程序是

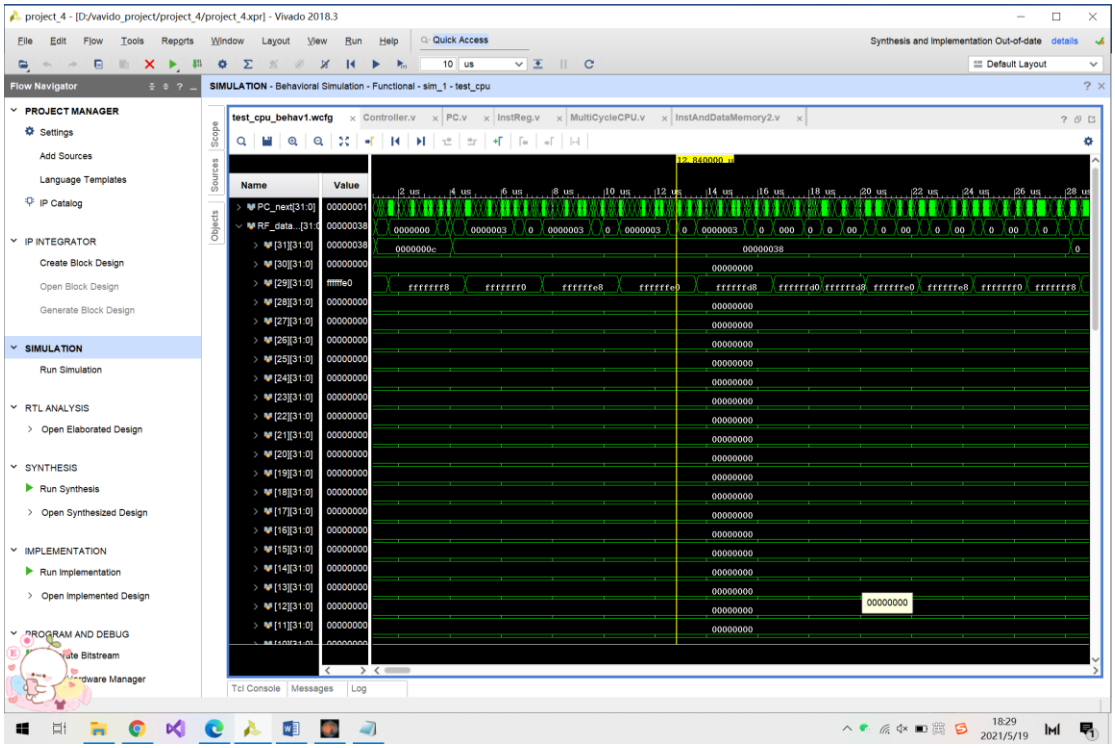
Int sum=0;

Int n=5;

f(n,sum);

Void f(int n,int sum){
if(n>0){
sum+=n;
f(n-1);
sum+=n;
}
所以$v0（即为 sum）先加 5,4,3,2,1，再依次加 1,2,3,4,5,
n 先递减进栈，后递增出栈，所以变化为$a0：5->4->3->2->1->0->1->2->3->4->5

$sp 初始值为 0，先分配栈空间（$sp=$sp-8）使要保存的变量（$ra,,$a0）进栈,共六层栈;每进一层栈$sp-=8；一直到 0xfffffffd0;后出栈（$sp=$sp+8）以恢复现场，又逐渐加 8 一直恢复到$sp=0
$sp 变化如下图

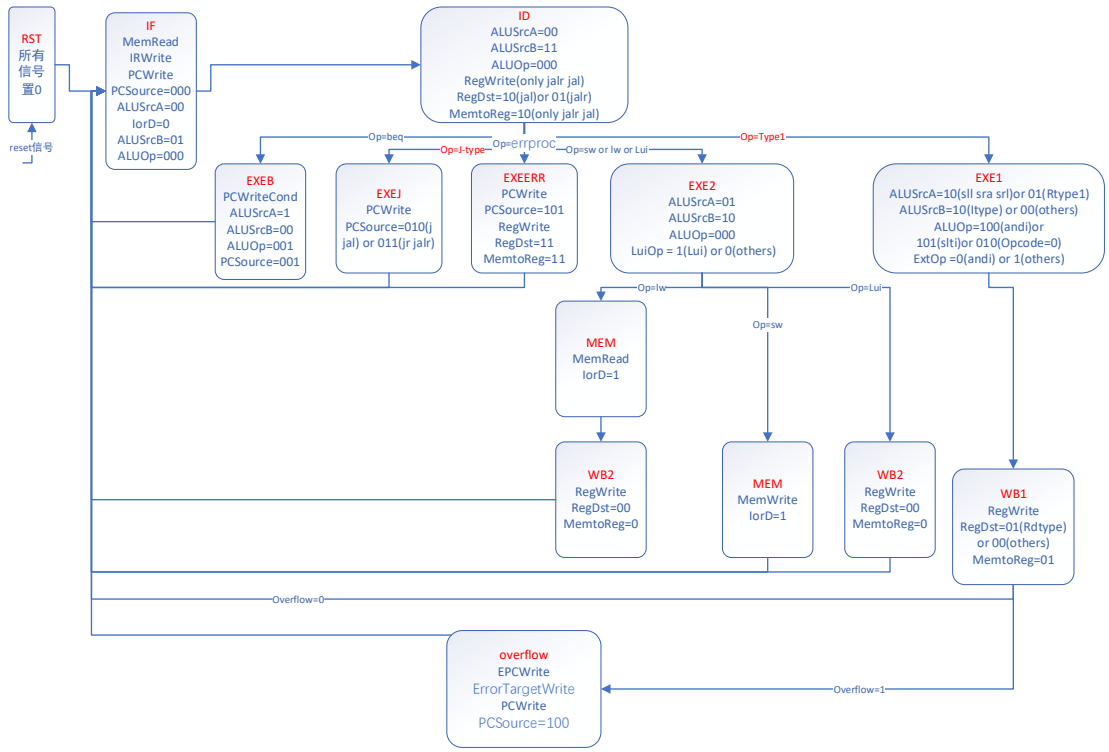


$ra 变化：0->0x0c->0x38->0x0c
第一个 jal 出现在
0x00000008：jal sum
0x0000000c：Loop:beq $zero, $zero, Loop
所以$ra 先变成 jal 的下一条指令（即 beq 指令）的地址 0x0c
第二个 jal 出现在：
0x00000034：jal sum
0x00000038：lw $a0, 0($sp)
所以$ra 变成 lw 的地址 0x38，持续六层栈的时间后，通过恢复现场出栈又变成开始进栈时保存的 0x0c
程序运行到最后一条指令不断在 LOOP 循环，所以$ra 最后保持 0x0c

6 异常处理（附加思考题，3 分，注：大作业总分 15 分） 试添加电路模块和异常处理指令，并修改数据通路，以实现对算术溢出的异常处理，具体要求如下：
（为方便查看，附加题的所有文件放进了 code(附加题)文件夹中，code 文件夹为不含附加题的文件）
6.1 1）增加 EPC 寄存器，保存异常指令的下一条指令地址（注：由于本次大作业没有与操作系统进行交互，因此无法修改异常指令，异常处理完成后直接跳过异常指令）；
由于在每条指令执行的第一个周期(取指令)将 PC+4 写 入了 PC，所以写入 EPC 的地址应该是 PC。
代码见 code 中的 EPC.v
2）增加 ErrorTarget 寄存器，当发生溢出异常时，保存算术运算的目标寄存器地址；
代码见 code 中的 ErrorTarget.v
3）增加异常处理程序的入口地址为 0x7c， 该地址存储了一条异常处理指令 errproc，该指令将 0xffffffff 写入目标寄存器地址（缓存在 ErrorTarget 寄存器中）；
入口地址为 0111_1100
而指令存储器中 Mem_data = MemRead? RAM_data[Address[RAM_SIZE_BIT + 1:2]]: 32'h00000000;（parameter RAM_SIZE_BIT = 8;）
所以该地址为 RAM_data[Address [9:2]]= RAM_data[8'b00011111],
同时应修改 parameter RAM_INST_SIZE = 33;

4）完成异常处理后，继续执行后续的指令程序。根据上述要求，请写出需要添加的控制信号和具体的控制功能。
EPCWrite——EPC 写入使能 ； 1-EPC 写入；0-EPC 不能写入
ErrorTargetWrite——ErrorTargetWrite 写入使能 1- ErrorTargetWrite 写入； 0-ErrorTargetWrite 不能写入
PCSource——3 位，PC 源选择信号。000-PC+4 即 ALU 的 result;001-ALUOut 寄存器的数据（用于 beq 指令）；010- {PC[31:28], IR[25:0], 2'b00}（用于 j 和 jal）；011-AData(即$rs 的地址数据，用于 jr 和 jalr 指令)
100- 0x7c；（overflow） 101-EPC(errproc 指令)
overflow——alu 输出的溢出检测信号；1-发生溢出；0-没有溢出
MemtoReg[1:0]:控制寄存器堆的写入数据的选择信号；
10-PC+4; (jal 和 jalr 指令时为 PC+4) 01-ALUOut;00--MDR;11: 0xffffffff(errproc 指令)
RegDst [1:0]：控制寄存器堆的写入寄存器的选择信号； 00-rt;01-rd;10-$ra;11-ErrorTarget(errproc 指令)

6.2 请完成异常处理指令 errproc 的设计（包括指令格式和状态机），并画出执行该指令时的状态转移图（不需要画出其他指令，但是需要包括所有的关键控制信号）。
Errproc={Opcode(6'h0x10),Rs(5'b0),Rt(5'b0),Rd(5'b0),Shamt(5'b0),Funct(6'b0)}
即 0100_0000_0000_0000_0000_0000_0000_0000
机器码为 0x40000000
在 mips 指令表中，add,addi，sub 需要溢出检测，addu、addiu、subu 以及其他指令不需要溢出检测。(slt 等指令用的是 Result <= {31'h00000000, Sign? lt_signed: (In1 < In2)}已检测符号，所以不需溢出检测)



6.3 完成数据通路的修改及电路 Verilog 代码 RTL 实现，自行设计汇编程序，给出异常处理的仿真结果并说明算术溢出异常处理的功能正确性。

加法存在溢出：正数与负数相加一定不会溢出；

正数与正数相加（两输入符号位均为 0），结果为负数（输出符号位为 1）

负数与负数相加（两输入符号位均为 1），结果为正数（输出符号位为 0）

减法存在溢出：

正数减负数（In1 符号位为 0，In2 符号位为 1），结果为负数（输出符号位为 1）

负数减正数（In1 符号位为 1，In2 符号位为 0），结果为正数（输出符号位为 0）

设计汇编程序:输入 1: 1000_0000_0000_0000_0000_0000_0000_0000（0x80000000）:-2^31

输入 2：0000_0000_0000_0000_0000_0000_0000_0001 (0x00000001):1

Sub 结果 0111_1111_1111_1111_1111_1111_1111_1111 (0x7fffffff)：2^31-1 溢出，与实际结果不符

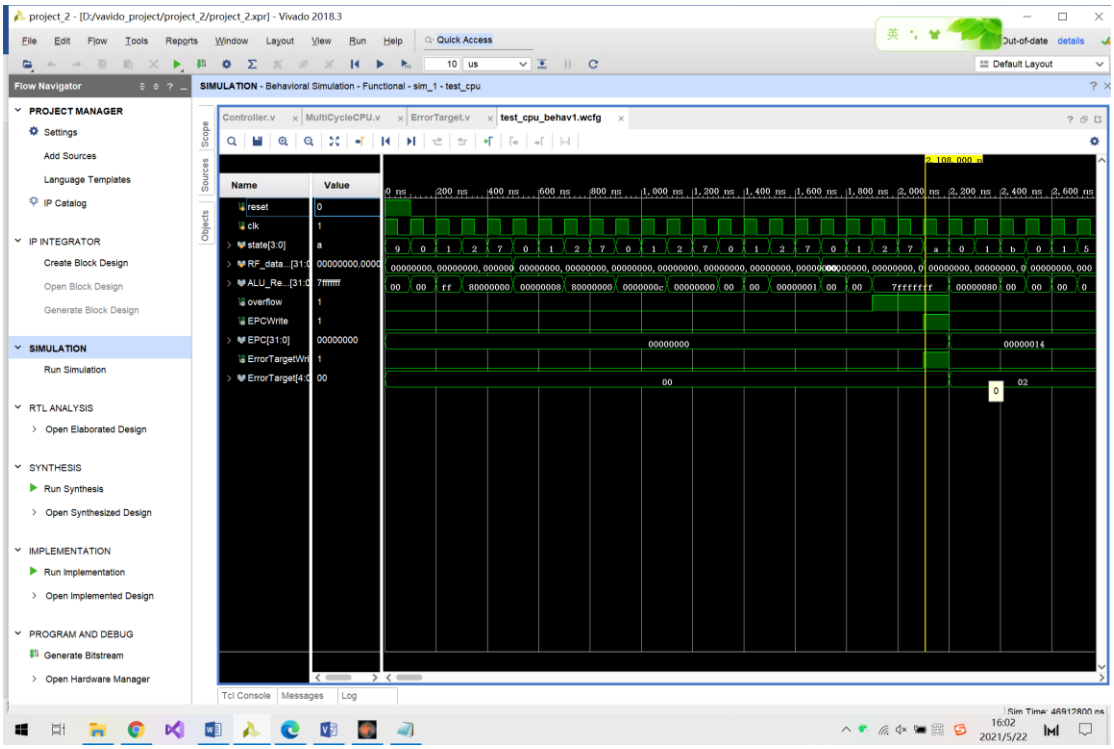
设计指令：

```
lui $a0, 0x8000
addi $a0, $a0, 0#-2^31
lui $a1, 0
addi $a1, $a1, 1#1
sub $v0 $a0 $a1
Loop:j Loop
```

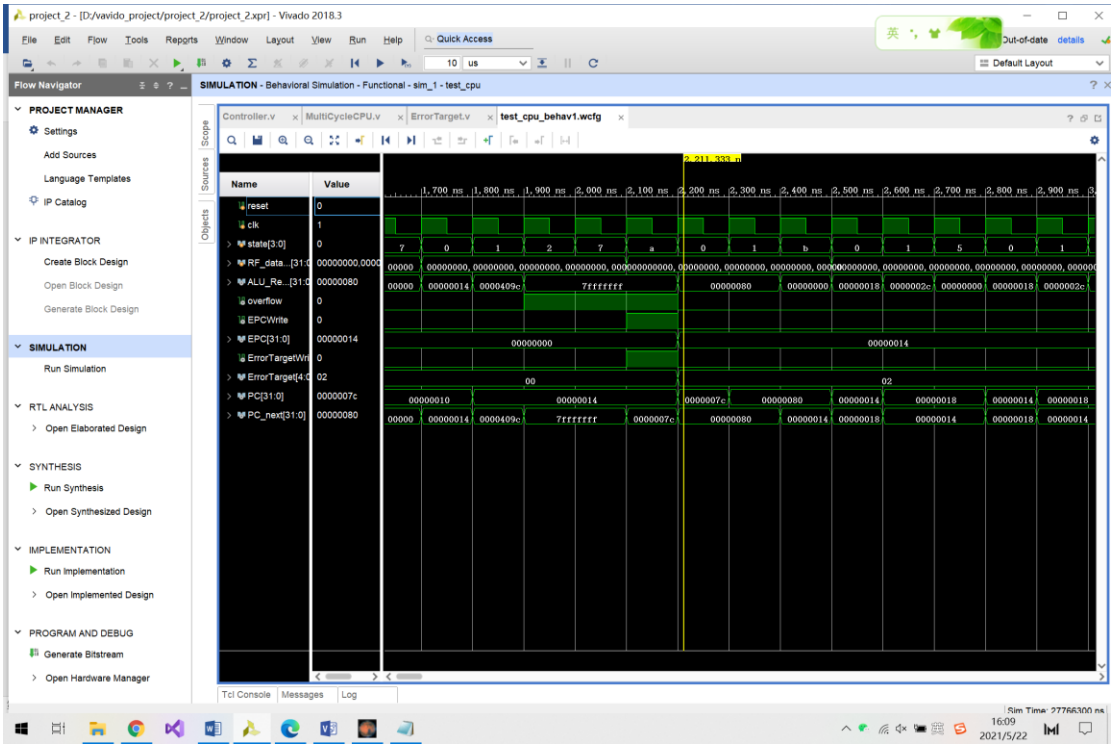
机器码为：（修改后的 InstAndDataMemory.v 为 code 中的 InstAndDataMemory4.v）

```
3c048000
20840000
3c050000
20a50001
00851022
08000005
```

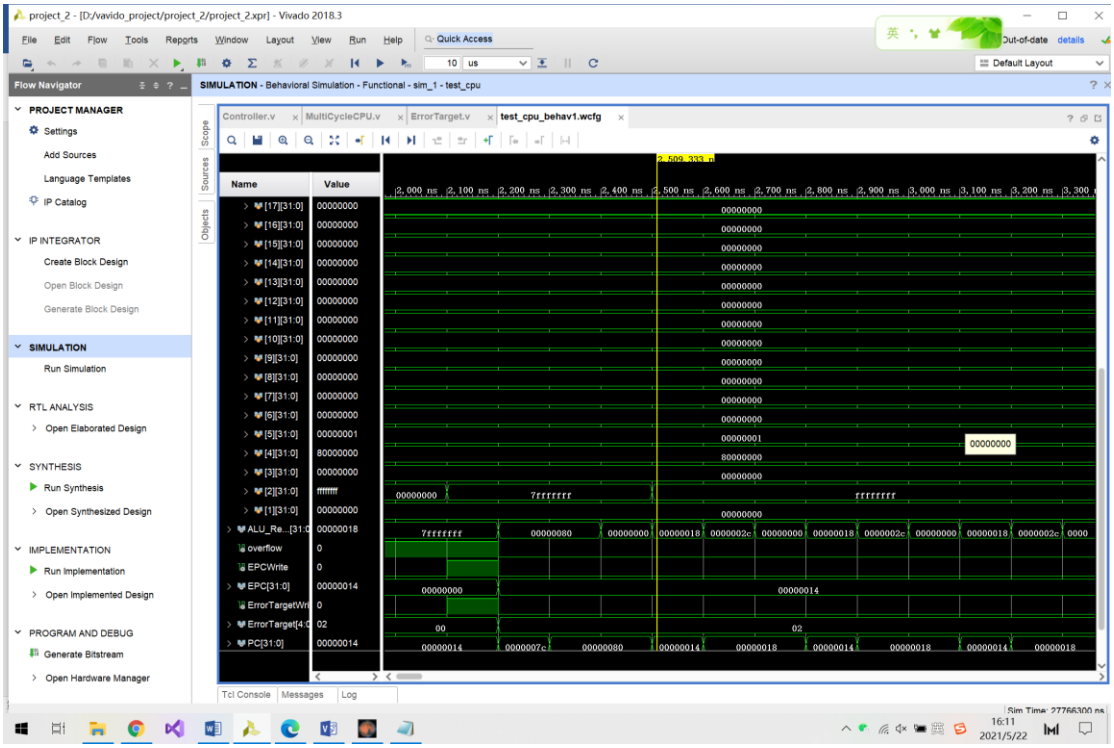
修改了 Alu.v， MultiCycleCPU.v, Controller.v;增加了 EPC.v, ErrorTarget.v
仿真结果如下：符合设计要求



如图，光标处发生溢出，alu_result=0x7fffffff,overflow=1,状态 state 转移至 a(即 soverflow),此状态下 EPCWrite 和 ErrorTargetWrite=1, EPC 记录了发生异常的指令的下一条指令地址 0x14, ErrorTarget 记录了算术运算的目标寄存器地址$2($v0),



然后 PC 更新为 0x7c（如图光标处）执行 errproc 指令，状态转移至 b(EXEERR),把 0xffffffff 写入$2(如下图光标处)，且 PC 更新为 EPC 保存的地址即发生异常的指令的下一条指令 0x14



设计汇编程序 2 验证 add:

lui $a0, 0xabcd

addi $a0, $a0, 0x1234#-2^31

lui $a1, 0xcdef

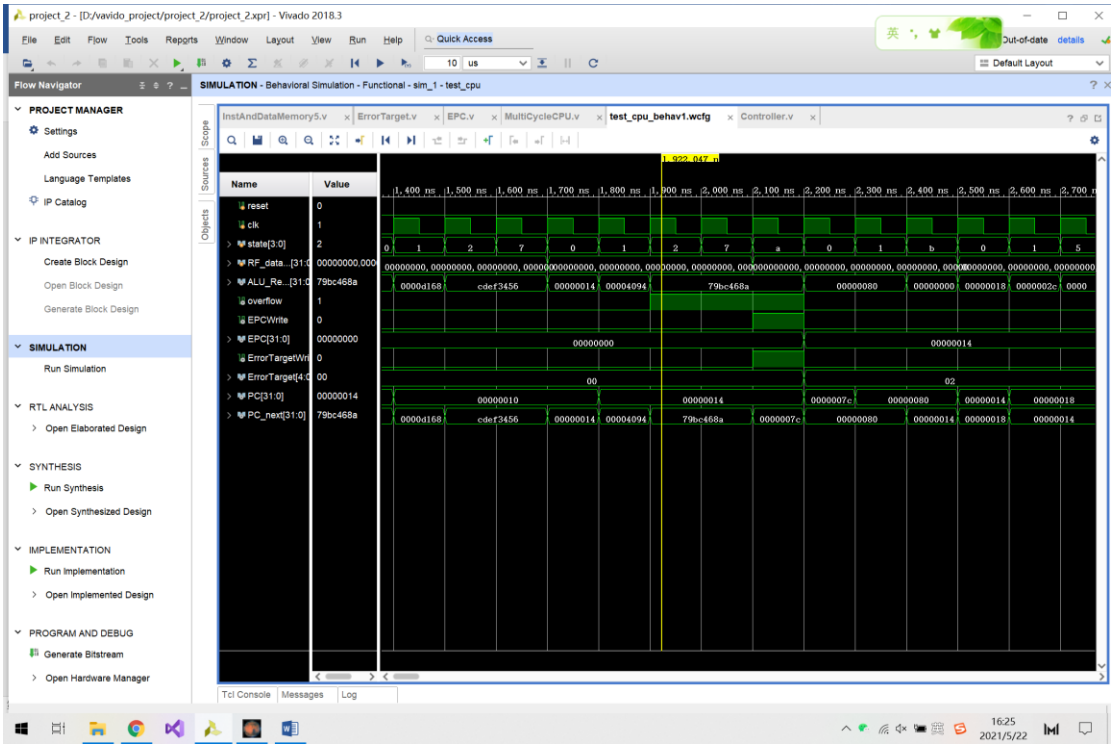
addi $a1, $a1, 0x3456#1

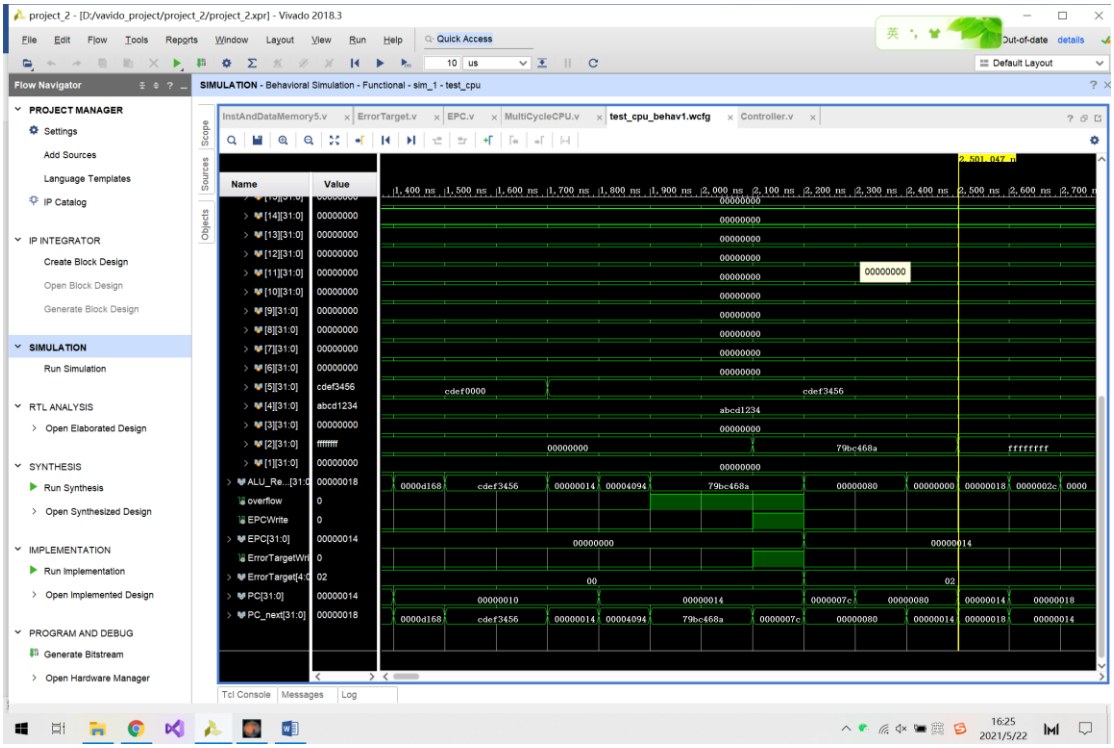
add $v0 $a0 $a1

Loop:j Loop

机器码见（修改后的 InstAndDataMemory.v 为 code 中的 InstAndDataMemory5.v）

仿真结果如下:


project_2 - [D:/vavido_project/project_2/project_2.xpr] - Vivado 2018.3
File Edit Flow Tools Reports Window Layout View Run Help
Quick Access
Out-of-date details
Default Layout
Flow Navigator
PROJECT MANAGER
Settings
Add Sources
Language Templates
IP Catalog
IP INTEGRATOR
Create Block Design
Open Block Design
Generate Block Design
SIMULATION
Run Simulation
RTL ANALYSIS
Open Elaborated Design
SYNTHESIS
Run Synthesis
Open Synthesized Design
IMPLEMENTATION
Run Implementation
Open Implemented Design
PROGRAM AND DEBUG
Generate Bitstream
Open Hardware Manager
SIMULATION - Behavioral Simulation - Functional - sim_1 - test_cpu
InstAndDataMemory5.v
ErrorTarget.v
EPC.v
MultiCycleCPU.v
test_cpu_behav1.wcfg
Controller.v
Name
Value
reset
clk
state[3:0]
RF_data...[31:0]
ALU_Re...[31:0]
overflow
EPCWrite
EPC[31:0]
ErrorTargetWri
ErrorTarget[4:0]
PC[31:0]
PC_next[31:0]
Tcl Console
Messages
Log
16:25
2021/5/22


project_2 - [D:/vavido_project/project_2/project_2.xpr] - Vivado 2018.3
File Edit Flow Tools Reports Window Layout View Run Help
Quick Access
Out-of-date details
Default Layout
Flow Navigator
PROJECT MANAGER
Settings
Add Sources
Language Templates
IP Catalog
IP INTEGRATOR
Create Block Design
Open Block Design
Generate Block Design
SIMULATION
Run Simulation
RTL ANALYSIS
Open Elaborated Design
SYNTHESIS
Run Synthesis
Open Synthesized Design
IMPLEMENTATION
Run Implementation
Open Implemented Design
PROGRAM AND DEBUG
Generate Bitstream
Open Hardware Manager
SIMULATION - Behavioral Simulation - Functional - sim_1 - test_cpu
InstAndDataMemory5.v
ErrorTarget.v
EPC.v
MultiCycleCPU.v
test_cpu_behav1.wcfg
Controller.v
Name
Value
Tcl Console
Messages
Log
16:25
2021/5/22