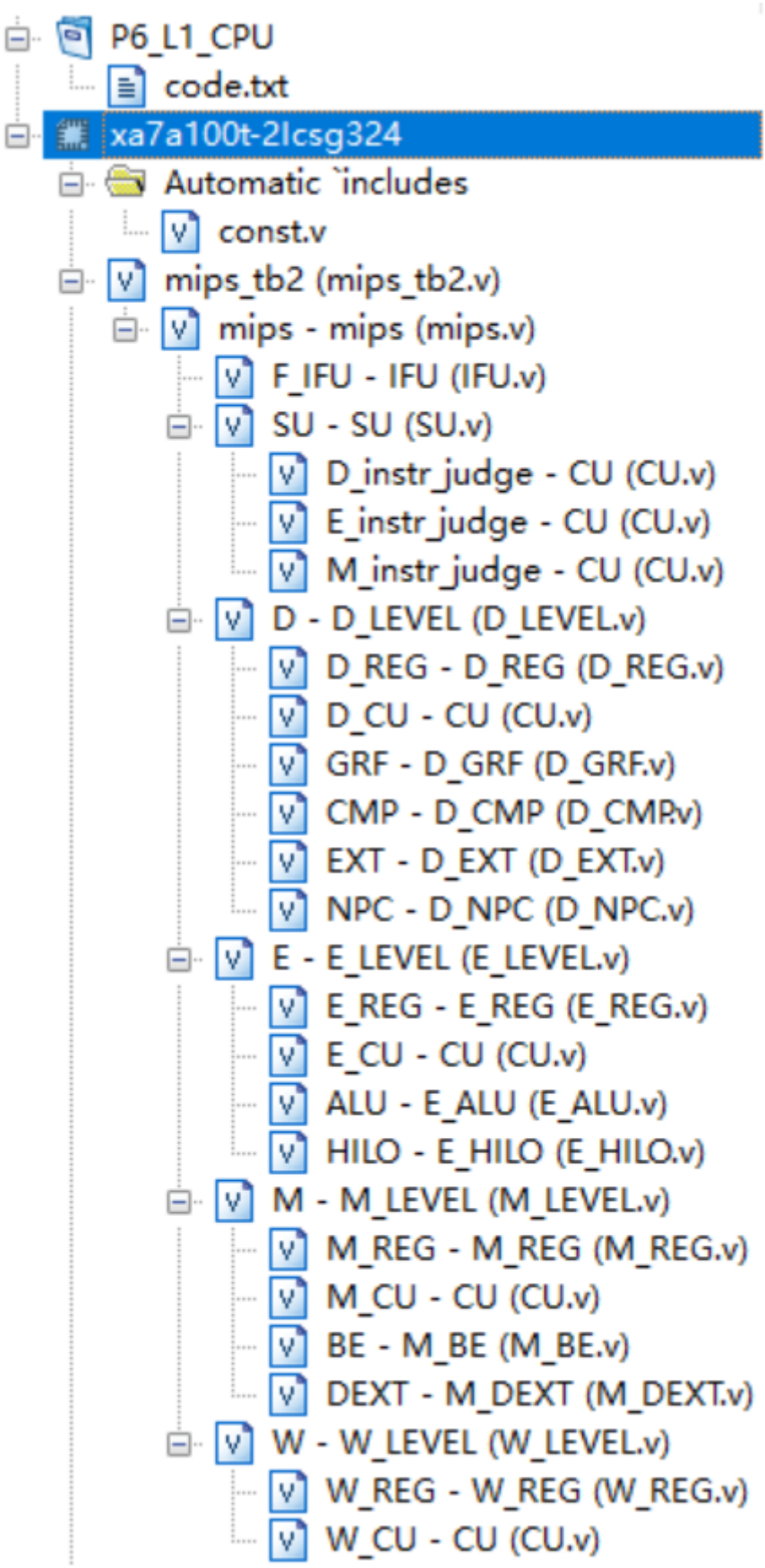
Verilog流水线CPU设计-MIPS-C3

一、CPU设计方案综述

（一）总体设计概述

本CPU为Logisim实现的32位5级流水线MIPS-CPU，支持指令集包含`{LB、LBU、LH、LHU、LW、SB、SH、SW、ADD、ADDU、 SUB、 SUBU、 MULT、 MULTU、 DIV、 DIVU、 SLL、 SRL、 SRA、 SLLV、 SRLV、SRAV、AND、OR、XOR、NOR、ADDI、ADDIU、ANDI、ORI、 XORI、LUI、SLT、SLTI、SLTIU、 SLTU、BEQ、BNE、BLEZ、BGTZ、 BLTZ、BGEZ、J、JAL、JALR、JR、MFHI、MFLO、MTHI、MTLO}`，并采用存储器外置的形式。为实现相关功能，CPU主要包含IFC、NPC、NPC、IM、GRF、EXT、ALU、BE、DEXT、CU、SU等模块。遵循形式化建模综合方法完成设计与实现。

（二）关键模块定义



1.暂停控制器-SU

新增乘除槽相关指令产生stall_hilo、md、mf、mt等信号

```
`include "const.v"

module SU(
    input [31:0] D_instr,
```

```
    input [31:0] E_instr,
    input [31:0] M_instr,
     input [1:0] E_ACmpB,
     input [1:0] M_ACmpB,
     input [1:0] E_ACmp0,
     input [1:0] M_ACmp0,
     input E_Busy,
    output stall
    );

    wire [1:0] Tuse_RS, Tuse_RT;
    wire [1:0] Tnew_E, Tnew_M;
    wire stall_rs_e, stall_rs_m, stall_rs;
    wire stall_rt_e, stall_rt_m, stall_rt;

    assign stall_rs_e = (Tuse_RS < Tnew_E) && D_instr[25:21] && (D_instr[25:21]
== E_RFA3) && E_RFWr;
    assign stall_rs_m = (Tuse_RS < Tnew_M) && D_instr[25:21] && (D_instr[25:21]
== M_RFA3) && M_RFWr;
    assign stall_rs = stall_rs_e | stall_rs_m ;

    assign stall_rt_e = (Tuse_RT < Tnew_E) && D_instr[20:16] && (D_instr[20:16]
== E_RFA3) && E_RFWr;
    assign stall_rt_m = (Tuse_RT < Tnew_M) && D_instr[20:16] && (D_instr[20:16]
== M_RFA3) && M_RFWr;
    assign stall_rt = stall_rt_e | stall_rt_m ;

    assign stall_hilo = E_Busy && (D_md | D_mf | D_mt) ;

    assign stall = stall_rs | stall_rt | stall_hilo ;

// ============== D级 ====================//
    wire D_cal_r, D_cal_i, D_load, D_store, D_branch, D_j_reg, D_md, D_mf, D_mt;
    CU D_instr_judge (
    .op(D_instr[31:26]),
    .funct(D_instr[5:0]),
    .rt_op(D_instr[20:16]),
     //output
    .cal_r(D_cal_r),
    .cal_i(D_cal_i),
    .load(D_load),
    .store(D_store),
    .branch(D_branch),
    .j_reg(D_j_reg),
     .md(D_md),
     .mf(D_mf),
     .mt(D_mt)
    );

    assign Tuse_RS = (D_cal_r | D_cal_i | D_load | D_store | D_md | D_mt) ?
2'd1:
                         (D_branch | D_j_reg) ? 2'd0 : 2'd3 ;
    assign Tuse_RT = (D_store) ? 2'd2 :
                         (D_cal_r | D_md) ? 2'd1 :
                         (D_branch)? 2'd0 : 2'd3 ;

// ============== E级 ====================//
    wire E_cal_r, E_cal_i, E_load, E_mf;
```

```
    wire E_RFWr;
    wire [1:0] E_GRFA3Sel;
    wire [4:0] E_RFA3;
    CU E_instr_judge (
    .op(E_instr[31:26]),
    .funct(E_instr[5:0]),
    .rt_op(E_instr[20:16]),
     .ACmpB(E_ACmpB),
     .ACmp0(E_ACmp0),
     //output
    .cal_r(E_cal_r),
    .cal_i(E_cal_i),
    .load(E_load),
     .mf(E_mf),

    .RFWr(E_RFWr),
    .GRFA3Sel(E_GRFA3Sel)
    );

    assign E_RFA3 = (E_GRFA3Sel == `GRFA3_rt) ? E_instr[20:16] :
                        (E_GRFA3Sel == `GRFA3_rd) ? E_instr[15:11] :
                        (E_GRFA3Sel == `GRFA3_31) ? 5'd31 :
                        (E_GRFA3Sel == `GRFA3_00) ? 5'd0  :
                          E_instr[20:16];
    assign Tnew_E = (E_load) ? 2'd2 :
                        (E_cal_r | E_cal_i | E_mf) ? 2'd1 :
                           2'd0;

// ============== M级 ====================//
    wire M_load;
    wire M_RFWr;
    wire [1:0] M_GRFA3Sel;
    wire [4:0] M_RFA3;
    CU M_instr_judge (
    .op(M_instr[31:26]),
    .funct(M_instr[5:0]),
    .rt_op(M_instr[20:16]),
     .ACmpB(M_ACmpB),
     .ACmp0(M_ACmp0),
     //output
    .load(M_load),

    .RFWr(M_RFWr),
    .GRFA3Sel(M_GRFA3Sel)
    );

    assign M_RFA3 = (M_GRFA3Sel == `GRFA3_rt) ? M_instr[20:16] :
                        (M_GRFA3Sel == `GRFA3_rd) ? M_instr[15:11] :
                        (M_GRFA3Sel == `GRFA3_31) ? 5'd31 :
                        (M_GRFA3Sel == `GRFA3_00) ? 5'd0  :
                          M_instr[20:16];
    assign Tnew_M = M_load ? 2'd1 : 2'd0 ;

endmodule
```

2.转发模块

分散在各级，列举典型例子:

E_LEVEL 发出转发数据:

```
    wire [3:0] ALUOp;
    wire [1:0] ALUBSel;
    wire [1:0] GRFA3Sel,GRFWDSel;//解析回写控制信号
    wire j_link; //解析当前指令类型，以判断是否以产生数据可以转发
    CU E_CU (
    .op(IR_out[31:26]),
    .funct(IR_out[5:0]),
    .rt_op(IR_out[20:16]),
     //output
    .ALUOp(ALUOp),
    .ALUBSel(ALUBSel),

     .RFWr(E_RFWr_out),
    .GRFA3Sel(GRFA3Sel),
    .GRFWDSel(GRFWDSel),
     //指令检测
     .j_link(j_link)
    );

    assign E_RFA3_out = (GRFA3Sel == `GRFA3_rt) ? IR_out[20:16] :
                        (GRFA3Sel == `GRFA3_rd) ? IR_out[15:11] :
                        (GRFA3Sel == `GRFA3_31) ? 5'd31 :
                          IR_out[20:16];
    assign E_RFWD_out = (GRFWDSel == `GRFWD_pc8) ? PC_out + 8 :
                            0 ;
    assign E_Forward_Ready_out = j_link ;
```

E_LEVEL 接收并选择转发数据

```
    wire [31:0] ALU_A, ALU_B;
    wire [31:0] FWD_RS, FWD_RT;
    assign FWD_RS = (IR_out[25:21] == 0) ? 0 :
                    (IR_out[25:21] == M_RFA3_in && M_RFWr_in &&
M_Forward_Ready_in) ? M_RFWD_in :
                    (IR_out[25:21] == W_RFA3_in && W_RFWr_in &&
W_Forward_Ready_in) ? W_RFWD_in :
                     V1_out ;
    assign FWD_RT = (IR_out[20:16] == 0) ? 0 :
                    (IR_out[20:16] == M_RFA3_in && M_RFWr_in &&
M_Forward_Ready_in) ? M_RFWD_in :
                    (IR_out[20:16] == W_RFA3_in && W_RFWr_in &&
W_Forward_Ready_in) ? W_RFWD_in :
                     V2_out0 ;
    assign ALU_A = FWD_RS ;
    assign ALU_B =  (ALUBSel == `ALUB_rd2) ? FWD_RT         :
                    (ALUBSel == `ALUB_imm) ? EXT_out        :
                    (ALUBSel == `ALUB_shamt) ? IR_out[10:6] : 0 ;

    assign V2_out = FWD_RT; //转发处理后正确的数据才能传到下一级
```

（三） 重要机制实现方法

1.跳转

这里确实容易出错， 因为寄存器的原因导致了PC和NPC的分离，若使用D级PC信号会导致NPC的滞后而产生类似指令回流错觉的错误。因而得用F级PC的信号做PC正常累加。其他的同单周期CPU设置NPCOp的方法。

2.流水线延迟槽

jal的改存PC+8进入寄存器

3.转发

E、M、W级都具有完整指令，将其解析出所有写寄存器的信息（不管是不是写入指令）一起转发到相应D、E、M共5处转发位点。设置转发判断相关条件已控制是否选择转发的信号

4.暂停

AT法分析Tnew和Tuse并比较，当Tuse < Tnew且当前指令是冲突指令（需要转发）时，需要暂停

T_use < T_new : 暂停

指令	T_use	
	rs([25:21])	rt([20:16]])
cal_r	1	1
cal_i	1	
load	1	
store	1	2
branch	0	0
j_reg	0	0
j_link		
j_addr		

D			E			M
指令	源寄存器	T_use	Tnew_E			Tnew_M
			cal_r (rd)	cal_i (rt)	load (rt)	load (rt)
			1	1	2	1
cal_r	rs/rt	1			stall	
cal_i	rs	1			stall	
load	rs	1			stall	
store	rs	1			stall	
store	rt	2				
branch	rs/rt	0	stall	stall	stall	stall
j_reg	rs	0	stall	stall	stall	stall

5.存储器外置

完成信号的传输；DM处设置byteen完成按字节存储和指令控制信号传输

二、测试方案

（一）手动少量测试

新增指令功能性测试

```
ori $a1, $0, 2
ori $a2, $0, 4
ori $a3, $0, 6

sll $t1, $t0, 4
sllv $t2, $t1, $a1
srl $t3, $t2, 2
srlv $t4, $t3, $a2
sra $t5, $t4, 16
srav $t6, $t5, $a2
```

```
sll $t7, $t6, 20
sllv $t8, $t7, $a3
sra $t9, $t8, 16
### cal_r：shift类功能测试

lui $t0, 0x5678
ori $t0, $t0, 0x1234
lui $t1, 0xffff

mult $t0, $t1
mfhi $a0
mflo $a1
addu $t2, $a1, $a0

multu $t0, $t1
mfhi $a0
mflo $a1
addu $t2, $a1, $a0

div $t1, $t0
mfhi $a2
mflo $a3
addu $t3, $a2, $a3

divu $t1, $t0
mfhi $a2
mflo $a3
addu $t3, $a2, $a3

mtlo $t3
mthi $t2
mfhi $s0
mflo $s1
### 乘除功能测试
```

加载代码到ise的code.txt中

```
system("java -jar Mars_perfect.jar db nc mc CompactDataAtZero 8000 dump .text
HexText P5_LO_CPU/code.txt self_test.asm > msg_out_mars.txt");
```

在mips_tb.v相应模块中进行文件操作，导出文件后在进行比较

```
void fileCmp(){
    int cnt=0, flag = 1;
    string tmp_cpu,tmp_mars;

    ifstream cpufile("msg_out_cpu.txt");
    ifstream marfile("msg_out_mars.txt");
    ofstream cmpfile("msg_cmp.txt");

    while(getline(cpufile,tmp_cpu) && getline(marfile,tmp_mars)){
        cnt++;
        if(tmp_cpu != tmp_mars){
            flag = 0;
            cmpfile << "mismatch - line" << cnt << "\t: ";
            cmpfile << "[cpu] " << tmp_cpu << "\t[mars] " << tmp_mars << endl;
```

```
        }
    }
    if(flag) cmpfile << "no error";
}

int main(){
    fileCmp();
    return 0;
}
```

（二）自动批量测试

搭配了自动批量测试文件，p5的稍微修一下就好

```
import os

mars_dir = 'D:\\0_Personal_File\\Grade2_01_Autumn\\CO\\P5\\Mars_perfect.jar'
hexcode_dir = 'D:\\0_Personal_File\\Grade2_01_Autumn\\CO\\P5\\code.txt'
mipscode_dir = ''
standard_outdir =
'D:\\0_Personal_File\\Grade2_01_Autumn\\CO\\P5\\ans_out_mars.txt'

walk =
os.walk('D:\\0_Personal_File\\Grade2_01_Autumn\\CO\\P5\\P5tester\\analysis\\work
\\P5_data')
fcmp = open('ans_cmp.txt','w')
epoch = 0
for path1, docu_list, file_list in walk:
    for file_name in file_list:
        if('.asm' in file_name):
            mipscode_dir = os.path.join(path1, file_name)

            print(epoch + 1,mipscode_dir)

            epoch = epoch + 1
            os.system('java -jar ' + mars_dir + ' ' + mipscode_dir + ' nc mc
CompactDataAtZero a dump .text HexText ' + hexcode_dir) #编译出十六进制文件
            os.system('java -jar ' + mars_dir + ' ' + mipscode_dir + ' db nc mc
CompactDataAtZero 8000 >' + standard_outdir)

            print('mars finished')

            #进行编译
            os.environ['XILINX'] = 'D:\\Xilinx\\14.7\\ISE_DS\\ISE'
            fuse_dir = 'D:\\Xilinx\\14.7\\ISE_DS\\ISE\\bin\\nt64\\fuse'
            prj_dir =
'D:\\0_Personal_File\\Grade2_01_Autumn\\CO\\P5\\P5_LO_CPU\\mips_tb_beh.prj'
            tcl_dir =
'D:\\0_Personal_File\\Grade2_01_Autumn\\CO\\P5\\P5_LO_CPU\\conf.tcl'

            print('ise file prepared')

            #使用编译产生的可执行文件，输出CPU的答案
            os.system(fuse_dir + ' -nodebug -prj ' + prj_dir + ' -o testX.exe
mips_tb > log.txt')
```

```
            os.system('testX.exe -nolog -tclbatch '+ tcl_dir + '>
ans_out_cpu.txt')

            print('ise finished')

            #比对两个文件
            temp = open('ans_out_cpu.txt', 'r')
            useful = temp.readlines()[8:]
            newtemp = open('ans_out_cpu_new.txt', 'w')
            newtemp.writelines(useful)
            temp.close()
            newtemp.close()

            temp = open('ans_out_mars.txt', 'r')
            useful = temp.readlines()
            while useful[0][0:9] < '@00003000' :
                del useful[0]
            newtemp = open('ans_out_mars_new.txt', 'w')
            newtemp.writelines(useful)
            temp.close()
            newtemp.close()

            file1 = open('ans_out_cpu_new.txt')
            l1 = file1.readlines()
            file2 = open('ans_out_mars_new.txt')
            l2 = file2.readlines()

            new_l1 = []
            new_l2 = []

            for i in range(len(l1)):
                if(l1[i][0] == '@'):
                    new_l1.append(l1[i])
            for i in range(len(l2)):
                if(l2[i][0] == '@'):
                    new_l2.append(l2[i])

            same = True
            if(len(new_l1) != len(new_l2)):
                same = False
            else:
                for i in range(len(new_l1)):
                    if(new_l1[i].strip() != new_l2[i].strip()):
                        print(i,'\n', new_l1[i], new_l2[i])
                        same = False
            if(same == False):
                print('Failure in: ', file_name, '  epoch: {}, result =
{}'.format(epoch, same))
                fcmp.write('Failure in: ' + file_name + '  epoch: {}, result =
{}\n'.format(epoch, same))
                break

            print('epoch: {}, result = {}'.format(epoch, same))
            fcmp.write('epoch: {}, result = {}\n'.format(epoch, same))

fcmp.close()
```

三、 思考题

- **为什么需要有单独的乘除法部件而不是整合进 ALU？为何需要有独立的 HI、LO 寄存器？**

  乘除运算慢（仿真时设置5/10周期得出结果），若放进ALU中，整个流水线都得等待乘除运算直至结束。会大大降低流水线的速度。独立出来，则使得在进行乘除运算时仍可以其他指令的运算，不至于大幅降低流水线速度。

- **参照你对延迟槽的理解，试解释“乘除槽”。**

  槽相当于脱离了流水线本身独立存在，目的便是防止流水线速度被慢速运算过多拖累

- **举例说明并分析何时按字节访问内存相对于按字访问内存性能上更有优势。（Hint： 考虑 C 语言中字符串的情况）**

  字符串读写。由于char类型按字节存储，因而修改相应字符类型的数据时，按字节访存更有优势

- **在本实验中你遇到了哪些不同指令类型组合产生的冲突？你又是如何解决的？相应的测试样例是什么样的？**

  指令总归类

```
    assign cal_r  = add | addu | sub | subu | slt | sltu | _and | _or | _xor
| _nor |
                        sll | sllv | srl | srlv | sra | srav ;
    assign cal_i  = addi | addiu | slti | sltiu | andi | ori | xori | lui ;
    assign md     = mult | multu | div  | divu ;
    assign mt     = mthi | mtlo ; //新增
    assign mf     = mfhi | mflo ; //新增
    assign load   = lw | lh | lb | lhu | lbu ;
    assign store  = sw | sh | sb ;
    assign branch = beq | bne | bgtz | blez | bgez | bltz | bgezal ;
    assign j_addr = j | jal ;
    assign j_link = jal | jalr | bgezal ;
    assign j_reg  = jr | jalr ;
```

  新增指令其实没产生新冲突，我将mf类指令乘除槽处输出结果和ALU计算结果通过MUX得到E级流出信号，所以无需增加新冲突；感　　觉反正P5已经把冲突写全了；添加了新的暂停规则

- **为了对抗复杂性你采取了哪些抽象和规范手段？这些手段在译码和处理数据冲突的时候有什么样的特点与帮助？**

  指令归类：无需重复设置转发和暂停，考虑更全面也更简单

  命名规则化：增加可读性