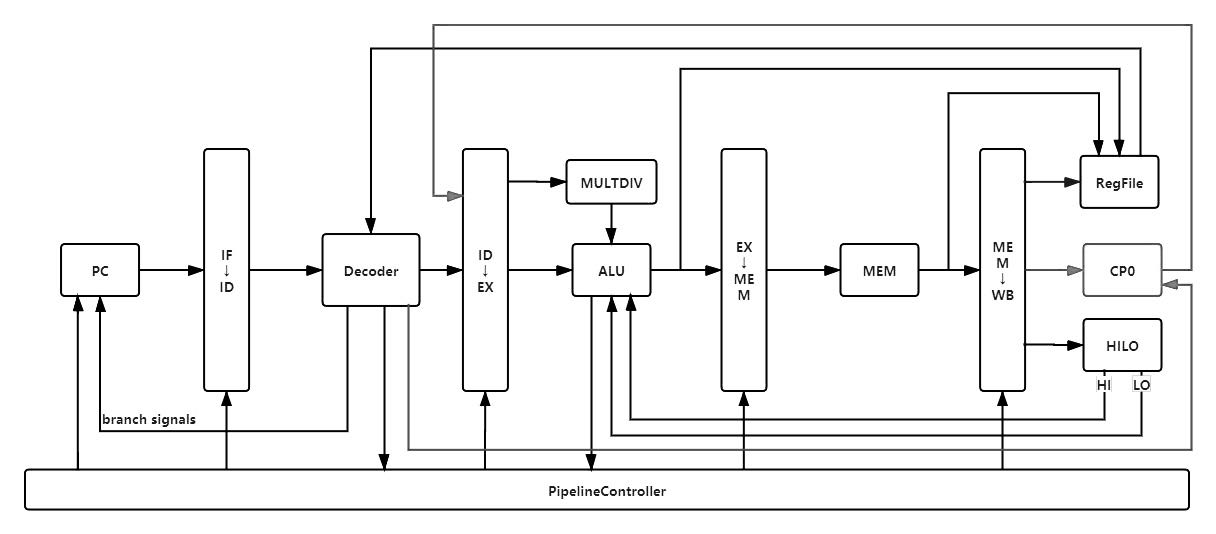
**Part 3 创新实验（演示系统介绍）实验分组编号：XXX（填写实验箱分组中组号）**

**系统概述：~~实现了什么功能，扩展了哪些内容，如何测试，有何发现。~~**

**本小组实现了复杂指令的拓展：div，divu,mult,multu,mfhi,mflo,mthi,mtlo，syscall,break等，并成功通过了n1-n88测试点。**

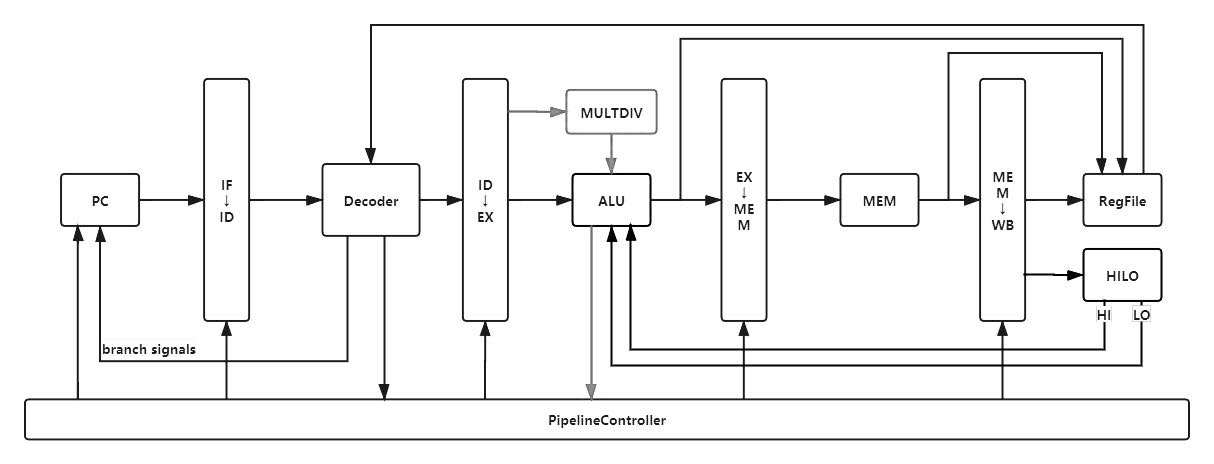
**1 系统总体架构图**

**【图片】**

****

**2 设计过程**

**3.2.1乘除法运算**

****

**（一）乘除法运算器**

我们小组分别使用Multipler和Divider Generate这两个IP核来实现乘法器与除法器的构建。但是所设置的乘除法器都是无符号的且无法在当前周期得到结果，所以在整合的时候需要对有符号的情况进行判断，对有符号乘除法的结果进行修正，并且增加运算完成的提示信号。

module MultDiv(

//接口定义

input clk,

input rst,

input flush,

input [`FUNCT\_BUS] funct,

input [`DATA\_BUS] operand\_1,

input [`DATA\_BUS] operand\_2,

input [`DATA\_BUS] hi,

input [`DATA\_BUS] lo,

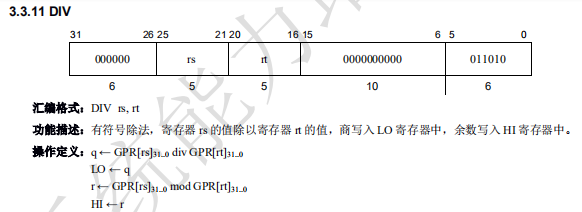
output done,

output reg[`DOUBLE\_DATA\_BUS] result

);

**（二）乘除法指令拓展（以DIV指令为例）**

**指令格式**

****

格式：DIV rs, rt

目标：有符号除法，rs的值除以rt的值，将商写入LO寄存器，余数写入HI寄存器

描述：q<-GPR[rs]31..0 div GPR[rt]31..0

LO<-q

r<-GPR[rs]31..0 mod GPR[rt]31..0

HI<-r

**实现代码**

DIV指令需要读取寄存器rs,rt,但是由于DIV指令为R型指令并且Opcode为special，所以不需要在ID模块生成寄存器读地址。只需修改/include/funct：

`define FUNCT\_DIV 6'b011010

随后，根据指令描述，我们需要将除法器计算出的结果写入LO和HI寄存器（hilo

写使能置1）故修改/cpu/core/stage/ex/ex.v来实现

// hilo写使能置1，hi\_out,lo\_out写入计算结果

always @(\*) begin

case (funct)

`FUNCT\_MULT, `FUNCT\_MULTU,

`FUNCT\_DIV, `FUNCT\_DIVU: begin

hilo\_write\_en <= 1;

hi\_out <= mult\_div\_result[63:32];

lo\_out <= mult\_div\_result[31: 0];

end

endcase

end

由于除法器的计算过程中无法在一个周期内结束，所以需要在除法进行计算时，暂停流水。

故修改/cpu/core/stage/ex/ex.v使得stall\_request <= !mult\_div\_done完成pc、if、id、ex的暂停。

//EX 阶段在计算完成之前发出流水线暂停请求

always @(\*) begin

case (funct)

`FUNCT\_MULT, `FUNCT\_MULTU,

`FUNCT\_DIV, `FUNCT\_DIVU: begin

stall\_request <= !mult\_div\_done;

end

default: stall\_request <= 0;

endcase

end

在完成EX的暂停信号后，我们便需要对/cpu/core/pipeline/PipelineController.v进行修改

//增加接口request\_from\_ex

input request\_from\_ex,

…

…

//暂停pc、if、id、ex

else if (request\_from\_ex) begin

stall <= 6'b001111;

end

这样在除法器完成计算后(request\_from\_ex=0)，又可以继续流水，这样DIV指令就拓展完成了。

**3.2.2数据移位**

**（一）HILO寄存器**

MFHI、MFLO、MTHI、MTLO，均为对 HILO 寄存器进行读写，同时HILO寄存器在WB模块，故在/core/storage中加设HILO.v文件

// write channel

input write\_en,

input [`DATA\_BUS] hi\_i,

input [`DATA\_BUS] lo\_i,

output [`DATA\_BUS] hi\_o,

output [`DATA\_BUS] lo\_o

…

…

reg [`DATA\_BUS] hi;

reg [`DATA\_BUS] lo;

assign hi\_o = hi;

assign lo\_o = lo;

always @(posedge clk) begin

if(rst) begin

hi <= 0;

lo <= 0;

end else if (write\_en) begin

hi <= hi\_i;

lo <= lo\_i;

end

end

由于是在WB 阶段写入HILO 寄存器，而在EX 阶段对 HILO 寄存器内数据进行读取和使用，所以存在RAW这种数据相关问题。故我们小组采用数据前递来解决此冲突，在/core/storage文件夹中，增加HILOReadProxy.v文件,将HILO中的结果提前传递给 ID 模块，与RegReadProxy相似。

input [`DATA\_BUS] hi\_i,

input [`DATA\_BUS] lo\_i,

input mem\_hilo\_write\_en,

input [`DATA\_BUS] mem\_hi\_i,

input [`DATA\_BUS] mem\_lo\_i,

input wb\_hilo\_write\_en,

input [`DATA\_BUS] wb\_hi\_i,

input [`DATA\_BUS] wb\_lo\_i,

output [`DATA\_BUS] hi\_o,

output [`DATA\_BUS] lo\_o

…

assign hi\_o = mem\_hilo\_write\_en ? mem\_hi\_i :

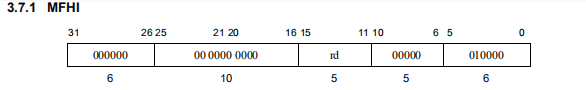
(wb\_hilo\_write\_en ? wb\_hi\_i : hi\_i);

assign lo\_o = mem\_hilo\_write\_en ? mem\_lo\_i :

(wb\_hilo\_write\_en ? wb\_lo\_i : lo\_i);

**（二）数据移动指令拓展**

**指令格式（MFHI）**

****

格式：MFHI rd

目标：将HI寄存器的值写入寄存器rd中

描述：GPR[rd]<-HI

**实现代码**

MFHI指令需要写入寄存器rd,但是由于MFHI 为R 型指令且指令操作码Opcode为special，所以不需要在ID模块修改，只需修改/include/funct

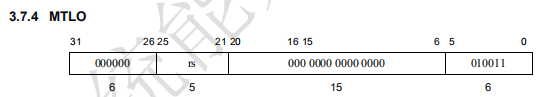
`define FUNCT\_MFHI 6'b010000

随后，HILO 部件会将进行了数据前递处理后的 hi、lo 寄存器数据传输给 EX 阶段，所以我们只需对/cpu/core/stage/ex/ex.v进行修改，我们将HI寄存器的值写入rd寄存器

`FUNCT\_MFHI: result <= hi\_in;

result信号会在流水线上传递下去，一直到WB级写回rt寄存器，这样MFHI指令就拓展完成了。

**指令格式（MTLO）**

****

格式：MTLO rs

目标：将寄存器rs的值写入HI寄存器中

描述：LO<-GPR[rs]

**实现代码**

MTLO指令需要读取寄存器rs,但是由于MTLO为R 型指令且指令操作码Opcode为special，所以不需要在ID模块修改，只需修改/include/funct

`define FUNCT\_MTLO 6'b010011

由于HIFO部件的设计，我们需要增加为 HILO 的写使能、hi 寄存器的写数据、lo 寄存器的写数据这三个信号量，同时由于无法区别写入的具体寄存器，故需要对hi、lo同时写入，对/cpu/core/stage/ex/ex.v进行修改，

`FUNCT\_MTLO: begin

hilo\_write\_en <= 1;

hi\_o <= hi\_i;

lo\_o <= operand\_1;

end

default: begin

hilo\_write\_en <= 0;

hi\_o <= hi\_i;

lo\_o <= lo\_i;

end

这样便完成了对MTLO指令的拓展

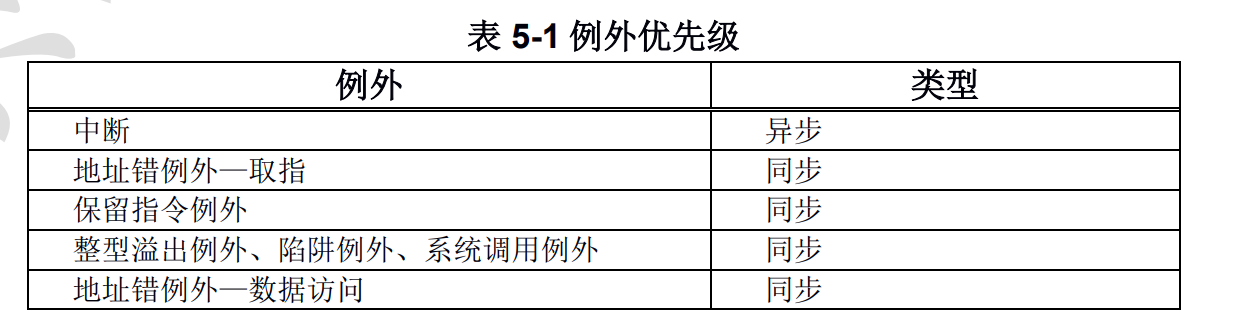
**（三）异常指令拓展**

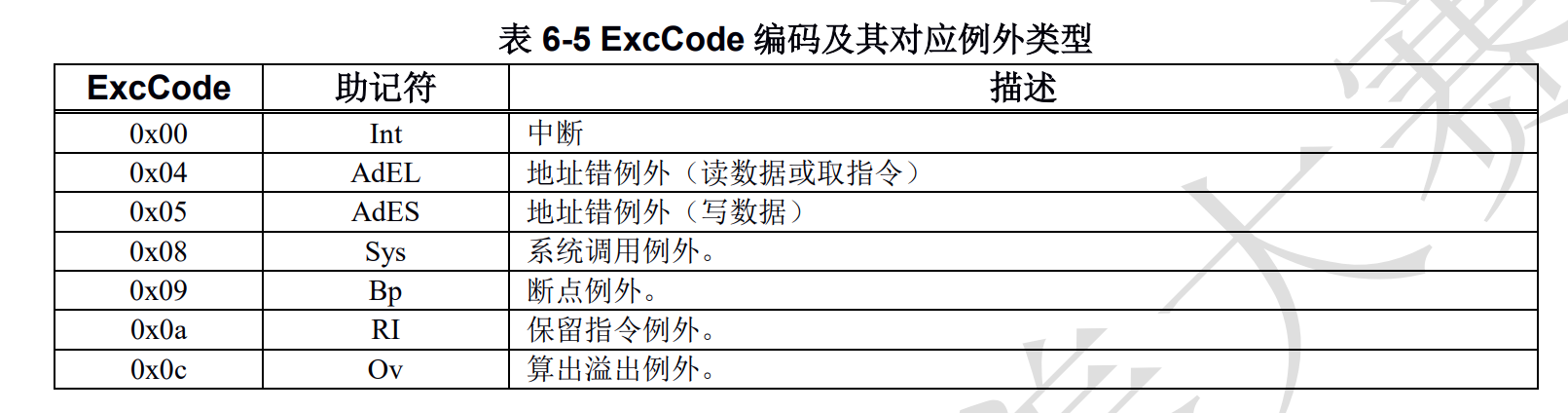
**例外处理**

**精确异常：**当一个异常发生的时候，系统的顺序执行会被中断，此时会有若干条指令处于流水线上的不同阶段，处理器会转移到异常处理历程，异常处理结束后会返回原程序继续执行，因为不希望异常处理例程破坏原程序的正常执行，对于异常发生时，流水线上没有执行完的指令就必须记住它所处于的阶段，以便异常处理结束后能够恢复执行，这就是精确异常。

**1、例外优先级**

当一条指令同时满足多个例外触发条件时，优先触发优先级高的例外。

****



在自己设计的TinyMIPS中，关于例外类型的实现在MEM.v中

**always @(\*) begin**

**if(int\_occured && int\_enabled) begin**

**exception\_type\_out <= `EXC\_TYPE\_INT;**

**end**

**else if (|current\_pc\_addr\_in[1:0]) begin**

**exception\_type\_out <= `EXC\_TYPE\_IF;**

**end**

**else if (exception\_type\_in[`EXC\_TYPE\_POS\_RI]) begin**

**exception\_type\_out <= `EXC\_TYPE\_RI;**

**end**

**else if (exception\_type\_in[`EXC\_TYPE\_POS\_OV]) begin**

**exception\_type\_out <= `EXC\_TYPE\_OV;**

**end**

**else if (exception\_type\_in[`EXC\_TYPE\_POS\_BP]) begin**

**exception\_type\_out <= `EXC\_TYPE\_BP;**

**end**

**else if (exception\_type\_in[`EXC\_TYPE\_POS\_SYS]) begin**

**exception\_type\_out <= `EXC\_TYPE\_SYS;**

**end**

**else if (adel\_flag) begin**

**exception\_type\_out <= `EXC\_TYPE\_ADEL;**

**end**

**else if (ades\_flag) begin**

**exception\_type\_out <= `EXC\_TYPE\_ADES;**

**end**

**else if (exception\_type\_in[`EXC\_TYPE\_POS\_ERET]) begin**

**exception\_type\_out <= `EXC\_TYPE\_ERET;**

**end**

**else begin**

**exception\_type\_out <= `EXC\_TYPE\_NULL;**

**end**

**end**

**2、处理器硬件响应一般型过程**

（1）当CP0.Status.EXL为0时，更新CP0.EPC。将例外处理返回后重新开始执行的指令PC填入到CP0.EPC寄存器中。如果发生例外的指令不在分支延迟槽中，则重新开始执行的指令PC就等于发生例外的指令的PC。否则重新开始执行的指令PC等于发生例外的指令的PC-4。

**assign count\_o  = reg\_count[32:1];**

**assign status\_o = reg\_status;**

**assign cause\_o = reg\_cause;**

**assign epc\_o = reg\_epc;**

**assign exc\_epc = delayslot\_flag ? current\_pc\_addr - 4 : current\_pc\_addr;**

（2）当EXL为0时，更新CP0.Cause寄存器的BD位：如果发生例外的指令在分支延迟槽中，则将CP0.Cause.BD置为1.

（3）更新CP0.Status.EXL位：将其置为1

（4）进入约定的例外入口重新取指，软件开始执行例外处理程序。

**case (exception\_type)**

**`EXC\_TYPE\_INT: begin**

**reg\_epc <= exc\_epc;**

**reg\_cause[`CP0\_SEG\_BD] <= delayslot\_flag;**

**reg\_status[`CP0\_SEG\_EXL] <= 1;**

**reg\_cause[`CP0\_SEG\_EXCCODE] <= `CP0\_EXCCODE\_INT;**

**end**

**`EXC\_TYPE\_IF, `EXC\_TYPE\_ADEL: begin**

**reg\_epc <= exc\_epc;**

**reg\_cause[`CP0\_SEG\_BD] <= delayslot\_flag;**

**reg\_badvaddr <= cp0\_badvaddr\_write\_data;**

**reg\_status[`CP0\_SEG\_EXL] <= 1;**

**reg\_cause[`CP0\_SEG\_EXCCODE] <= `CP0\_EXCCODE\_ADEL;**

**end**

**`EXC\_TYPE\_RI: begin**

**reg\_epc <= exc\_epc;**

**reg\_cause[`CP0\_SEG\_BD] <= delayslot\_flag;**

**reg\_status[`CP0\_SEG\_EXL] <= 1;**

**reg\_cause[`CP0\_SEG\_EXCCODE] <= `CP0\_EXCCODE\_RI;**

**end**

**`EXC\_TYPE\_OV: begin**

**reg\_epc <= exc\_epc;**

**reg\_cause[`CP0\_SEG\_BD] <= delayslot\_flag;**

**reg\_status[`CP0\_SEG\_EXL] <= 1;**

**reg\_cause[`CP0\_SEG\_EXCCODE] <= `CP0\_EXCCODE\_OV;**

**end**

**`EXC\_TYPE\_BP: begin**

**reg\_epc <= exc\_epc;**

**reg\_cause[`CP0\_SEG\_BD] <= delayslot\_flag;**

**reg\_status[`CP0\_SEG\_EXL] <= 1;**

**reg\_cause[`CP0\_SEG\_EXCCODE] <= `CP0\_EXCCODE\_BP;**

**end**

**`EXC\_TYPE\_SYS: begin**

**reg\_epc <= exc\_epc;**

**reg\_cause[`CP0\_SEG\_BD] <= delayslot\_flag;**

**reg\_status[`CP0\_SEG\_EXL] <= 1;**

**reg\_cause[`CP0\_SEG\_EXCCODE] <= `CP0\_EXCCODE\_SYS;**

**end**

**`EXC\_TYPE\_ADES: begin**

**reg\_epc <= exc\_epc;**

**reg\_cause[`CP0\_SEG\_BD] <= delayslot\_flag;**

**reg\_badvaddr <= cp0\_badvaddr\_write\_data;**

**reg\_status[`CP0\_SEG\_EXL] <= 1;**

**reg\_cause[`CP0\_SEG\_EXCCODE] <= `CP0\_EXCCODE\_ADES;**

**end**

**`EXC\_TYPE\_ERET: begin**

**reg\_status[`CP0\_SEG\_EXL] <= 0;**

**end**

**default:;**

**endcase**

**end**

**end**

**3、例外类型：**

**（1）中断例外：**

**当为屏蔽的中断到来时，触发中断例外。**



**// MEM.v**

**assign int\_occured = |(cp0\_cause\_in[`CP0\_SEG\_INT] & cp0\_status\_in[`CP0\_SEG\_IM])**

**assign int\_enabled = !cp0\_status\_in[`CP0\_SEG\_EXL] && cp0\_status\_in[`CP0\_SEG\_IE]**

**if(int\_occured && int\_enabled) begin**

**exception\_type\_out <= `EXC\_TYPE\_INT;**

**end**

**// CP0.v**

**case (exception\_type)**

**`EXC\_TYPE\_INT: begin**

**reg\_epc <= exc\_epc;**

**reg\_cause[`CP0\_SEG\_BD] <= delayslot\_flag;**

**reg\_status[`CP0\_SEG\_EXL] <= 1;**

**reg\_cause[`CP0\_SEG\_EXCCODE] <= `CP0\_EXCCODE\_INT;**

**end**

**（2）地址错例外**

当发生下列条件时触发地址错例外：

字 load/store 指令，其访问地址不对齐于字边界。

半字 load/store 指令，其访问地址不对齐与半字边界。

取指 PC 不对齐于字边界。

**// adel, ades**

**always @(\*) begin**

**if(|current\_pc\_addr\_in[1:0]) begin**

**{adel\_flag, ades\_flag} <= 2'b10;**

**cp0\_badvaddr\_write\_data\_out <= current\_pc\_addr\_in;**

**end**

**else if(mem\_sel\_in == 4'b0011 && address[0]) begin**

**{adel\_flag, ades\_flag} <= {mem\_read\_flag\_in, mem\_write\_flag\_in};**

**cp0\_badvaddr\_write\_data\_out <= address;**

**end**

**else if (mem\_sel\_in == 4'b1111 && |address[1:0]) begin**

**{adel\_flag, ades\_flag} <= {mem\_read\_flag\_in, mem\_write\_flag\_in};**

**cp0\_badvaddr\_write\_data\_out <= address;**

**end**

**else begin**

**{adel\_flag, ades\_flag} <= 2'b00;**

**cp0\_badvaddr\_write\_data\_out <= 0;**

**end**

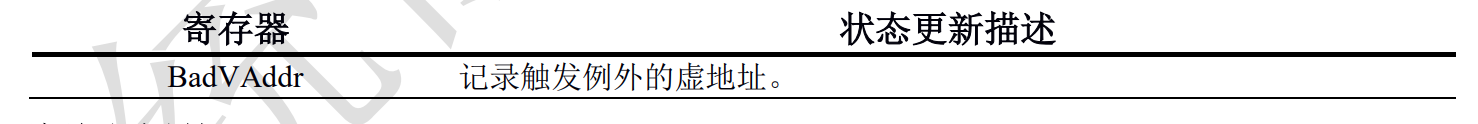
**end**

**控制寄存器 Cause 的 ExcCode 域：**

**0x04(ADEL)： 取指或读数据**

**0x05(ADES): 写数据**

**响应例外时的额外硬件状态更新：**



**（3）整型溢出例外**

**// Exception.v**

**assign overflow\_flag = ((op == `OP\_SPECIAL && (funct == `FUNCT\_ADD || funct == `FUNCT\_SUB)) || (op == `OP\_ADDI)) ? 1 : 0;**

**// ID.v**

**wire invalid\_inst\_flag, overflow\_flag;**

**assign exception\_type ={**

**eret\_flag, /\* ADE \*/ 1'b0,**

**syscall\_flag, break\_flag, /\* TP \*/ 1'b0,**

**overflow\_flag, invalid\_inst\_flag, /\* IF \*/ 1'b0};**

**// EX.v**

**wire overflow\_sum = ((!operand\_1[31] && !operand\_2\_mux[31]) && result\_sum[31]) || ((operand\_1[31] && operand\_2\_mux[31]) && (!result\_sum[31]));**

**wire overflow\_exc = exception\_type\_in[`EXC\_TYPE\_POS\_OV] ? overflow\_sum : 0;**

**assign exception\_type\_out = {exception\_type\_in[7:3], overflow\_exc, exception\_type\_in[1:0]};**

**//CP0.v**

**`EXC\_TYPE\_OV: begin**

**reg\_epc <= exc\_epc;**

**reg\_cause[`CP0\_SEG\_BD] <= delayslot\_flag;**

**reg\_status[`CP0\_SEG\_EXL] <= 1;**

**reg\_cause[`CP0\_SEG\_EXCCODE] <= `CP0\_EXCCODE\_OV;**

**end**

当一条 ADD、ADDI 或 SUB 指令执行结果溢出时，触发整型溢出例外。

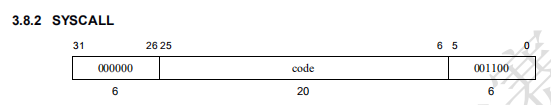
**控制寄存器 Cause 的 ExcCode 域：**

0x0c(Ov)

响应例外时的额外硬件状态更新：

无

**（4）系统调用例外（Syscall）**

****

格式：SYSCALL

目标：触发系统调用例外

描述：SignalException(SystemCall)

**实现代码**

SYSCALL指令操作码Opcode为special，所以不需要在ID模块修改，只需修改/include/funct/funct.v

`define FUNCT\_SYSCALL 6'b001100

首先我们需要对ID阶段进行修改，增加一个关于异常指令译码的模块ExceptionGen

……

assign syscall\_flag = (op == `OP\_SPECIAL && funct == `FUNCT\_SYSCALL)? 1:0;

……

之后对EX阶段进行修改

// exception signal

input syscall\_flag\_in,

output syscall\_flag\_out,

……

assign syscall\_flag\_out = syscall\_flag\_in;

然后，对MEM阶段进行修改

input syscall\_flag\_in,

output syscall\_flag\_out,

……

// exception\_type\_o

always @(\*) begin

…

else if (exception\_type\_in[`EXC\_TYPE\_POS\_SYS]) begin

exception\_type\_out <= `EXC\_TYPE\_SYS;

end

…

修改PipelineController 模块

// generate the exception handle signals

assign flush = stall\_all? 0:(exception\_type != `EXC\_TYPE\_NULL)?1:0;

always @(\*) begin

if(exception\_type == `EXC\_TYPE\_ERET) begin

exc\_pc <= cp0\_epc;

end

else if (exception\_type != `EXC\_TYPE\_NULL) begin

exc\_pc <= 32'hbfc0\_0380;

end

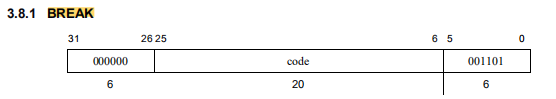
else begin

exc\_pc <= 32'hbfc0\_0000;

end

end

**（5）断点例外（Break）**

****

格式：BREAK

功能：触发断点例外。

操作定义：SignalException(Breakpoint)

**代码实现**

在funct.v中添加

`define FUNCT\_BREAK 6'b001101

在译码阶段需要对异常相关指令进行译码

//ExceptionGen.v

output break\_flag,

assign break\_flag = (op == `OP\_SPECIAL && funct == `FUNCT\_BREAK)?1: 0;

EX阶段

//EX.v

input break\_flag\_in,

output break\_flag\_out,

…

assign break\_flag\_out = break\_flag\_in;

MEM阶段

//MEM.v

input break\_flag\_in,

output break\_flag\_out,

…

assign break\_flag\_out = break\_flag\_in;

**（6）保留指令例外**

当执行一条未实现的指令时，触发保留指令例外。

**控制寄存器 Cause 的 ExcCode 域：**

0x0a(RI)

响应例外时的额外硬件状态更新：

无

**// Exception.v**

**always @(\*) begin**

**case (op)**

**`OP\_SPECIAL: begin**

**case (funct)**

**`FUNCT\_SLL, `FUNCT\_SRL, `FUNCT\_SRA, `FUNCT\_SLLV,**

**`FUNCT\_SRLV, `FUNCT\_SRAV, `FUNCT\_JR, `FUNCT\_JALR,**

**`FUNCT\_MOVN, `FUNCT\_MOVZ, `FUNCT\_ADD, `FUNCT\_SUB,**

**`FUNCT\_MFHI, `FUNCT\_MTHI, `FUNCT\_MFLO, `FUNCT\_MTLO,**

**`FUNCT\_MULT, `FUNCT\_MULTU, `FUNCT\_DIV, `FUNCT\_DIVU,**

**`FUNCT\_ADDU, `FUNCT\_SUBU, `FUNCT\_AND, `FUNCT\_OR,**

**`FUNCT\_XOR, `FUNCT\_NOR, `FUNCT\_SLT, `FUNCT\_SLTU,**

**`FUNCT\_SYSCALL, `FUNCT\_BREAK: begin**

**invalid\_inst\_flag <= 0;**

**end**

**default: invalid\_inst\_flag <= 1;**

**endcase**

**end**

**`OP\_REGIMM: begin**

**case (rt)**

**`REGIMM\_BLTZ, `REGIMM\_BLTZAL,**

**`REGIMM\_BGEZ, `REGIMM\_BGEZAL: begin**

**invalid\_inst\_flag <= 0;**

**end**

**default: invalid\_inst\_flag <= 1;**

**endcase**

**end**

**`OP\_CP0: begin**

**case (rs)**

**`CP0\_MFC0, `CP0\_MTC0, `CP0\_ERET: begin**

**invalid\_inst\_flag <= 0;**

**end**

**default: invalid\_inst\_flag <= 1;**

**endcase**

**end**

**`OP\_J, `OP\_JAL, `OP\_BEQ, `OP\_BNE, `OP\_BLEZ, `OP\_BGTZ,**

**`OP\_ADDIU, `OP\_SLTI, `OP\_SLTIU, `OP\_ANDI, `OP\_ORI,**

**`OP\_XORI, `OP\_LUI, `OP\_LB, `OP\_LH, `OP\_LW, `OP\_LBU,**

**`OP\_LHU, `OP\_SB, `OP\_SH, `OP\_SW, `OP\_ADDI: begin**

**invalid\_inst\_flag <= 0;**

**end**

**default: invalid\_inst\_flag <= 1;**

**endcase**

**end**

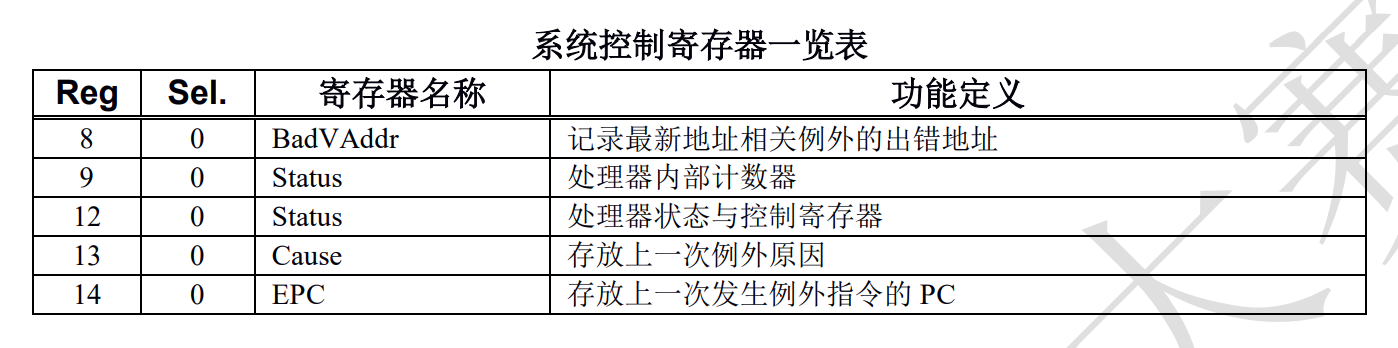
**// MEM.v**

**else if (exception\_type\_in[`EXC\_TYPE\_POS\_RI]) begin**

**exception\_type\_out <= `EXC\_TYPE\_RI;**

**end**

**系统控制寄存器**



**以下为CP0.v中**

**always @(posedge clk) begin**

**if (rst) begin**

**reg\_badvaddr <= 32'h0;**

**reg\_count <= 33'h0;**

**reg\_status <= `CP0\_REG\_STATUS\_VALUE;**

**reg\_cause <= 32'h0;**

**reg\_epc <= 32'h0;**

**end**

**else begin**

**reg\_count <= reg\_count + 1;**

**reg\_cause[15:10] <= interrupt\_i;**

**if (cp0\_write\_en) begin**

**case (cp0\_write\_addr)**

**`CP0\_REG\_COUNT: reg\_count <= {cp0\_write\_data, 1'b0};**

**`CP0\_REG\_STATUS: begin**

**reg\_status[22] <= cp0\_write\_data[22];**

**reg\_status[15:8] <= cp0\_write\_data[15:8];**

**reg\_status[1:0] <= cp0\_write\_data[1:0];**

**end**

**`CP0\_REG\_CAUSE: reg\_cause[9:8] <= cp0\_write\_data[9:8];**

**`CP0\_REG\_EPC: reg\_epc <= cp0\_write\_data;**

**default: ;**

**endcase**

**end**

**case (exception\_type)**

**`EXC\_TYPE\_INT: begin**

**reg\_epc <= exc\_epc;**

**reg\_cause[`CP0\_SEG\_BD] <= delayslot\_flag;**

**reg\_status[`CP0\_SEG\_EXL] <= 1;**

**reg\_cause[`CP0\_SEG\_EXCCODE] <= `CP0\_EXCCODE\_INT;**

**end**

**`EXC\_TYPE\_IF, `EXC\_TYPE\_ADEL: begin**

**reg\_epc <= exc\_epc;**

**reg\_cause[`CP0\_SEG\_BD] <= delayslot\_flag;**

**reg\_badvaddr <= cp0\_badvaddr\_write\_data;**

**reg\_status[`CP0\_SEG\_EXL] <= 1;**

**reg\_cause[`CP0\_SEG\_EXCCODE] <= `CP0\_EXCCODE\_ADEL;**

**end**

**`EXC\_TYPE\_RI: begin**

**reg\_epc <= exc\_epc;**

**reg\_cause[`CP0\_SEG\_BD] <= delayslot\_flag;**

**reg\_status[`CP0\_SEG\_EXL] <= 1;**

**reg\_cause[`CP0\_SEG\_EXCCODE] <= `CP0\_EXCCODE\_RI;**

**end**

**`EXC\_TYPE\_OV: begin**

**reg\_epc <= exc\_epc;**

**reg\_cause[`CP0\_SEG\_BD] <= delayslot\_flag;**

**reg\_status[`CP0\_SEG\_EXL] <= 1;**

**reg\_cause[`CP0\_SEG\_EXCCODE] <= `CP0\_EXCCODE\_OV;**

**end**

**`EXC\_TYPE\_BP: begin**

**reg\_epc <= exc\_epc;**

**reg\_cause[`CP0\_SEG\_BD] <= delayslot\_flag;**

**reg\_status[`CP0\_SEG\_EXL] <= 1;**

**reg\_cause[`CP0\_SEG\_EXCCODE] <= `CP0\_EXCCODE\_BP;**

**end**

**`EXC\_TYPE\_SYS: begin**

**reg\_epc <= exc\_epc;**

**reg\_cause[`CP0\_SEG\_BD] <= delayslot\_flag;**

**reg\_status[`CP0\_SEG\_EXL] <= 1;**

**reg\_cause[`CP0\_SEG\_EXCCODE] <= `CP0\_EXCCODE\_SYS;**

**end**

**`EXC\_TYPE\_ADES: begin**

**reg\_epc <= exc\_epc;**

**reg\_cause[`CP0\_SEG\_BD] <= delayslot\_flag;**

**reg\_badvaddr <= cp0\_badvaddr\_write\_data;**

**reg\_status[`CP0\_SEG\_EXL] <= 1;**

**reg\_cause[`CP0\_SEG\_EXCCODE] <= `CP0\_EXCCODE\_ADES;**

**end**

**`EXC\_TYPE\_ERET: begin**

**reg\_status[`CP0\_SEG\_EXL] <= 0;**

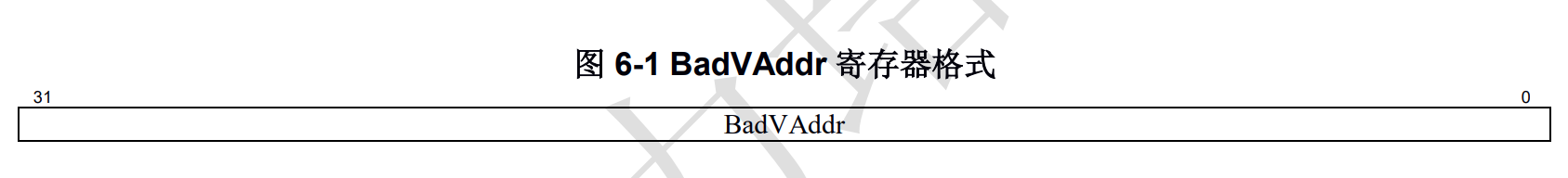
**end**

**default:;**

**endcase**

**end**

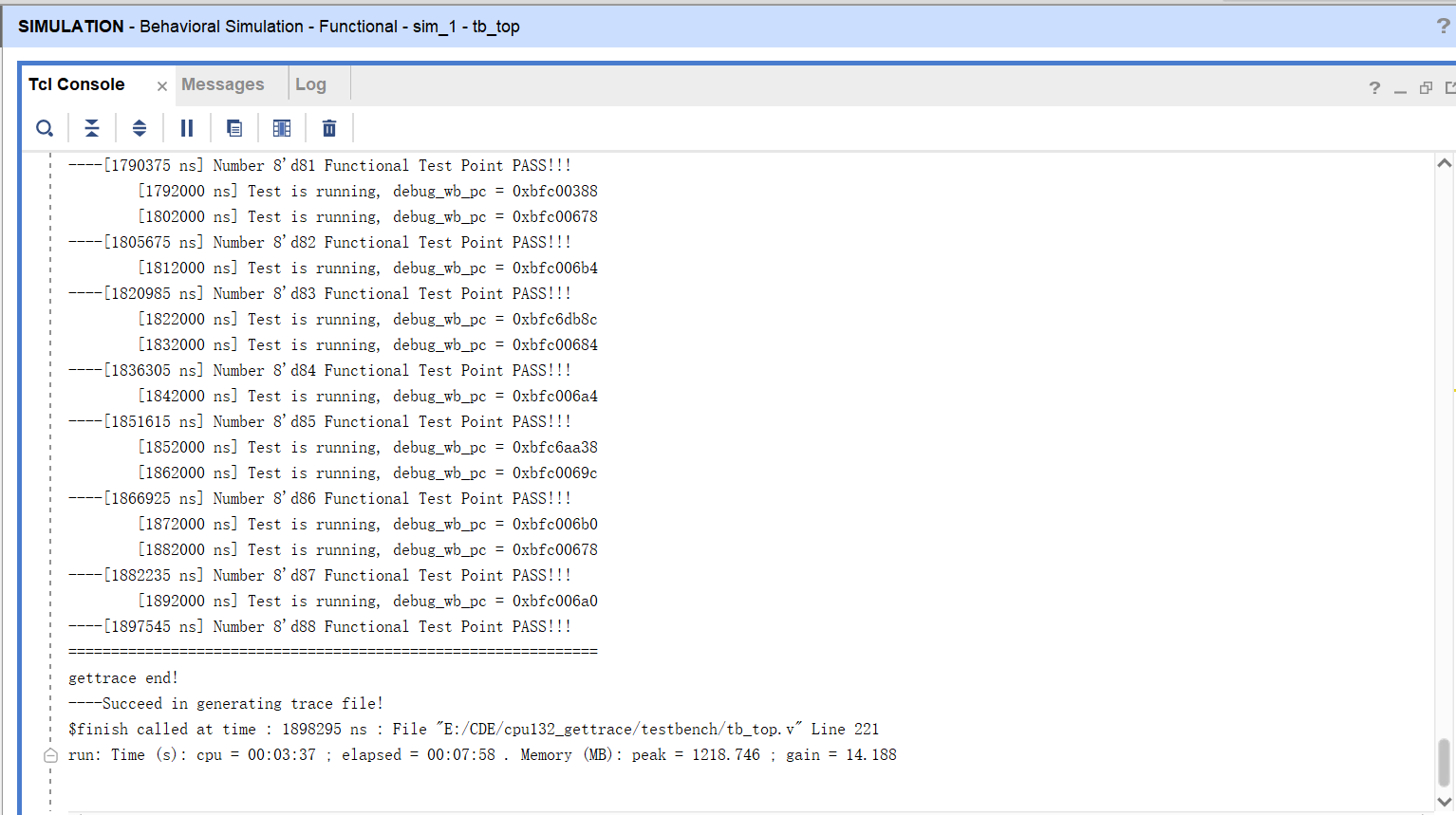
**end**

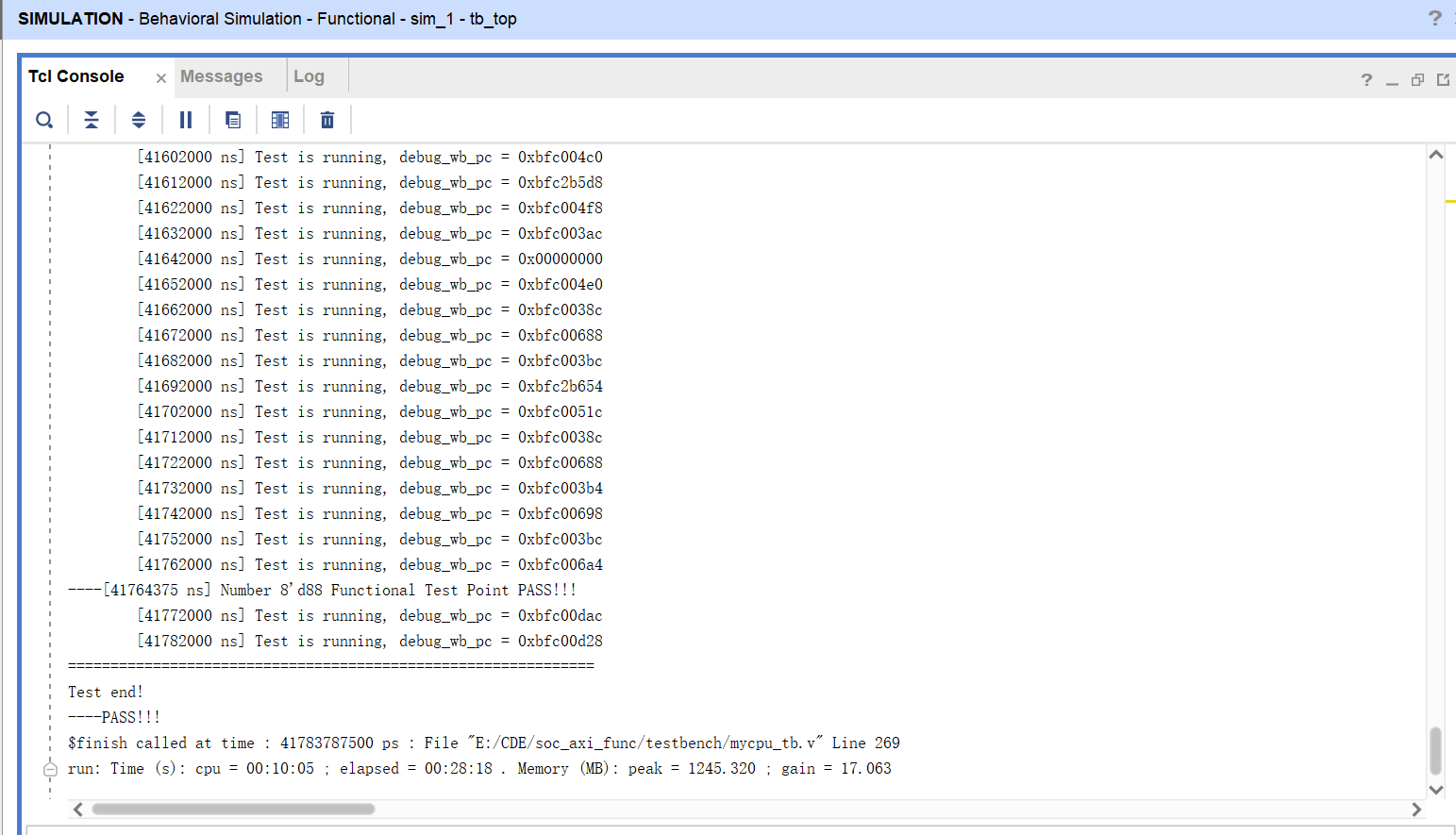




**3 实验现象及分析**

**通过trace比对的方法通过88个测试点**





**4 组内成员主要工作及贡献比例**

如：

张三 负责集成cache模块 50%

李四 负责测试 20%

王五 负责扩展异常指令XXX 30%