ToyMips——经典的五级流水线 CPU 实现

刘明杰	程恩华	谭子轩	朱亮
062020119	062010418	072020114	032040113

日期：2022 年 12 月 31 日

目录

1 简介

本项目实现了经典的五级流水线 MIPS，完成了 MIPS-32 手册中包含的主要指令。项目地址在https://github.com/caaatch22/ToyMips。除了规定的 57 条指令外，ToyMips 还完成了 `teq`, `tge` 等自陷相关的 12 条指令，移动指令中的 `movn`, `movz` 以及用于原子操作的 `ll`, `sc` **共 73 条指令**。具体指令如下表:

表 1: 实现的指令

指令功能类型	包含指令	详尽测试
算数运算指令	add, addi, addu,addiu, sub,subu, slt,slti,sltu,sltiu, mult,multu,div,divu	✓
逻辑运算指令	and,andi,or,ori,lui,nor,xor,xori	✓
移位操作指令	sll,sllv,srl,srlv,sra,srav	✓
分支跳转指令	beq,bne,bgez,bgtz,blez,bltz,bgezal,bltzal,j,jr,jal,jalr	✓
数据移动指令	mfhi, mflo,mthi,mtlo, movn, movz	✓
加载存储指令	lb, lbu, lh, lhu, lw, sb, sh, sw, ll, sc	✓
自陷、异常相关	teq, tge, tgeu, tlt, tltu,tne,teqi,tgei,tgeiu,tlti,tltiu,tnei, eret,mfc0,mtc0	
空指令	nop	✓

2 项目构建

本项目除了 CPU 设计外，还实现了一个自动测试框架。需要在 linux 环境下使用 MIPS32 交叉编译工具。

2.1 安装依赖

安装 `iverilog` 以及 `gtkwave`(可选的)

```
apt-get install iverilog
apt-get install gtkwave
```

安装 `GCC cross compilation tool chain`。我们需要 MIPS 交叉编译工具从汇编代码生成机器码。

```
wget https://sourcery.mentor.com/GNUToolchain/package12725/public/mips-sde-elf/mips
    -2014.05-24-mips-sde-elf-i686-pc-linux-gnu.tar.bz2

tar jxvf mips-2014.05-24-mips-sde-elf-i686-pc-linux-gnu.tar.bz2

rm mips-2014.05-24-mips-sde-elf-i686-pc-linux-gnu.tar.bz2
mv mips-2014.05 ~
```

对于 64-bit Linux,

```
sudo dpkg --add-architecture i386
sudo apt-get update
```

```
sudo apt-get install libc6:i386 libncurses5:i386 libstdc++6:i386
```

可以从 github 上下载代码：

```
git clone git@github.com:caaatch22/ToyMips.git
```

2.2 使用

使用我们的测试框架

```
cd test
make all
```

测试框架包括以下测试单元：

- test-arithmetic
- test-logic
- test-shift
- test-move
- test-div
- test-jump
- test-branch
- test-mem
- test-forwarding

你也可以单独的对他们进行测试：

```
cd test
make <test-name>
```

你可以使用 gtkwave 查看生成的波形文件 `<test-name>/dump.vcd`。

3 流水线 CPU 设计

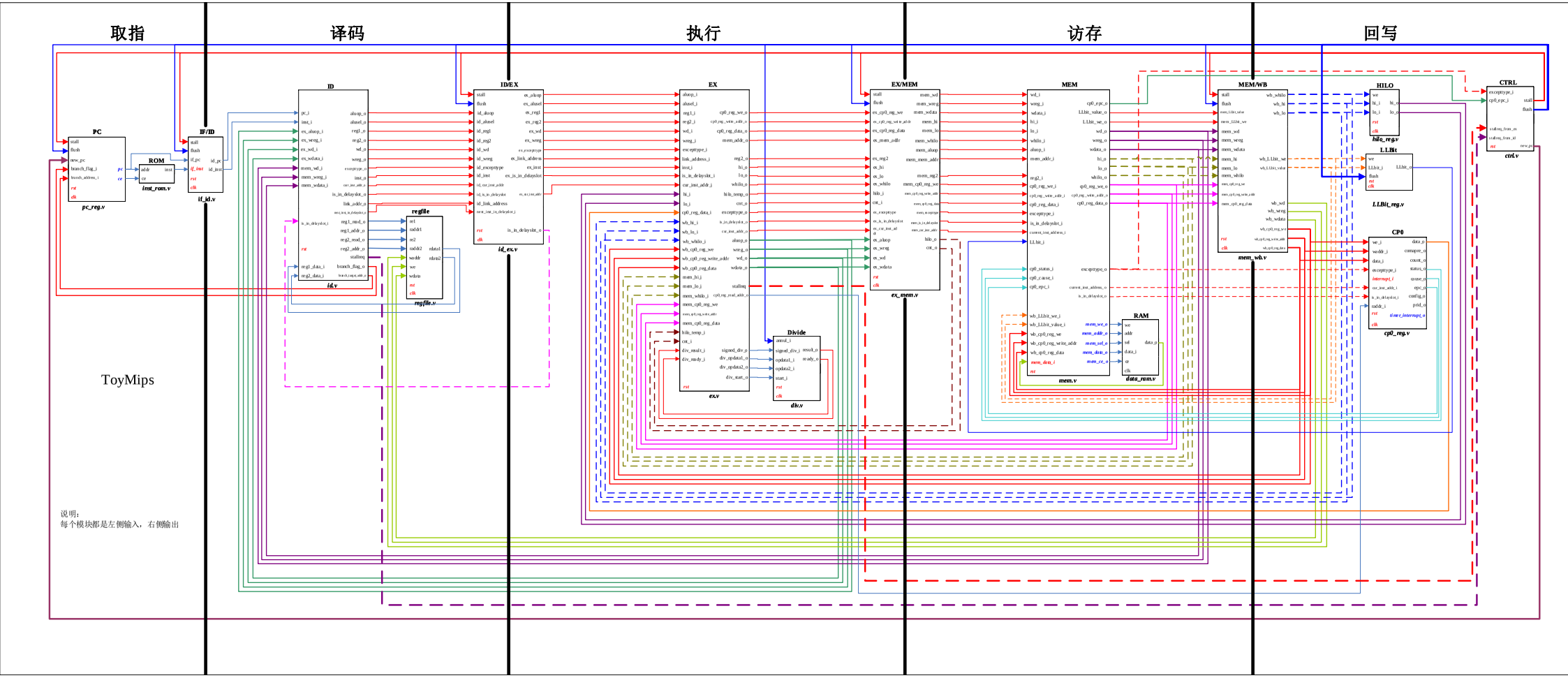
CPU 大致可以拆为五个阶段。它们分别是：Instruction Fetch，Instruction Decode，Execute，Memory，Write Back。

其中，各个阶段的任务是：

- IF: 从 Instruction Memory 中获取一条指令。
- ID：对指令进行解码，决定之后几个周期元器件的执行操作。
- EX：通过 ALU 进行计算。如果是分支指令，在这一阶段可以决定是否跳转。
- MEM：读写内存。
- WB：把数据写回 Register File。

3.1 总体设计

模块之间的连接图如下：


取指
译码
执行
访存
回写
PC
pc_reg.v
ROM
inst_rom.v
IF/ID
if_id.v
ID
id.v
regfile
regfile.v
ID/EX
id_ex.v
EX
ex.v
Divide
div.v
EX/MEM
ex_mem.v
MEM
mem.v
RAM
data_ram.v
MEM/WB
mem_wb.v
HILO
hilo_reg.v
LLbit
LLbit_reg.v
CP0
cp0_reg.v
CTRL
ctrl.v
ToyMips
说明；
每个模块都是左侧输入，右侧输出

3.2 部分模块接口与实现

接下来简要介绍数据通路中一些具有代表性的模块。分别有 ID 阶段，HI_LO 寄存器模块，div 模块以及 ctrl 模块。ID 阶段代表了五个流水线数据通路类似的部分，因此对 EX, MEM 不再赘述。HILO 则表示了一批非阶段寄存器的相关设计。div 模块为一个特殊设计。ctrl 模块则控制流水线暂停以及 EPC 相关。

阶段寄存器由于只是起到过渡作用，将不会写出。

3.2.1 ID 阶段

我们将分几次介绍 ID 模块的信号输入输出引脚。

基础信号

```
module id (
    input                   rst,
    input [`InstAddrBus]    pc_i,
    input [`InstBus]        inst_i,
    // read from regfile
    input [`RegBus]         reg1_data_i,
    input [`RegBus]         reg2_data_i,

    // send to regfile
    output reg              reg1_read_o,   // enable signal to regfile
    output reg              reg2_read_o,
    output reg[`RegAddrBus] reg1_addr_o,
    output reg[`RegAddrBus] reg2_addr_o,

    // send to exe
    output reg[`AluOpBus]   aluop_o,
    output reg[`AluSelBus]  alusel_o,
    output reg[`RegBus]     reg1_o,
    output reg[`RegBus]     reg2_o,
    output reg[`RegAddrBus] wd_o,
    output reg              wreg_o,
    // ... 未全部列出
);
```

以上为没有考虑数据冒险，load-store，以及异常相关时候的 ID 模块输入输出。除了最基本的数据、地址外，我们还增加了 `wreg_o` **是否写信号**。

解决相关 data hazard

```
module id (
    // ... 上接
    // read from excution phase to solve data hazard
    input                   ex_wreg_i,
    input [`RegBus]         ex_wdata_i,
    input [`RegAddrBus]     ex_wd_i,
```

```
    // read from mem phase to solve data hazard
    input                   mem_wreg_i,
    input [`RegBus]         mem_wdata_i,
    input [`RegAddrBus]     mem_wd_i,
);
```

我们从 ex，mem 阶段引入的一些数据，以解决数据冒险。具体可以见第 4 部分专门介绍 Hazard 处理。需要说明的是，我们并未专门采用一个 `forwarding` 模块来处理信号。这样的缺点是使得模块间的耦合增高。不利于维护，但优点是少了专门 forward 相关逻辑电路，可以减少延时，从而提升主频。

id 阶段还有分支、异常相关引脚，会放在后面专门的部分解释。

3.2.2 ctrl 模块

ctrl 模块的引脚如下：

```
module ctrl (
    input               rst,
    input               stallreq_from_id,
    input               stallreq_from_ex,

    input [31:0]        excepttype_i,
    input [`RegBus]     cp0_epc_i,


    output reg[`RegBus] new_pc,
    output reg          flush,

    // stall[0] = 1'b1: pc stay still
    // stall[1] = 1'b1: if pause
    // stall[2] = 1'b1: id pause
    // stall[3] = 1'b1: ex pause
    // stall[4] = 1'b1: mem pause
    // stall[5] = 1'b1: wb pause
    output reg[5:0]     stall
);
```

ctrl 模块的作用主要有两个：

- 接受来自 id, ex 阶段的暂停请求，并控制流水线暂停。
- 接受来自 CP0 分析的异常种类信号，给出 EPC 并控制流水线内寄存器的刷新。

接受暂停请求主要是为了控制结构冒险，这会在第四部分专门解释；与异常相关会在第五部分专门解释。以下说明该模块是如何控制流水线暂停的：

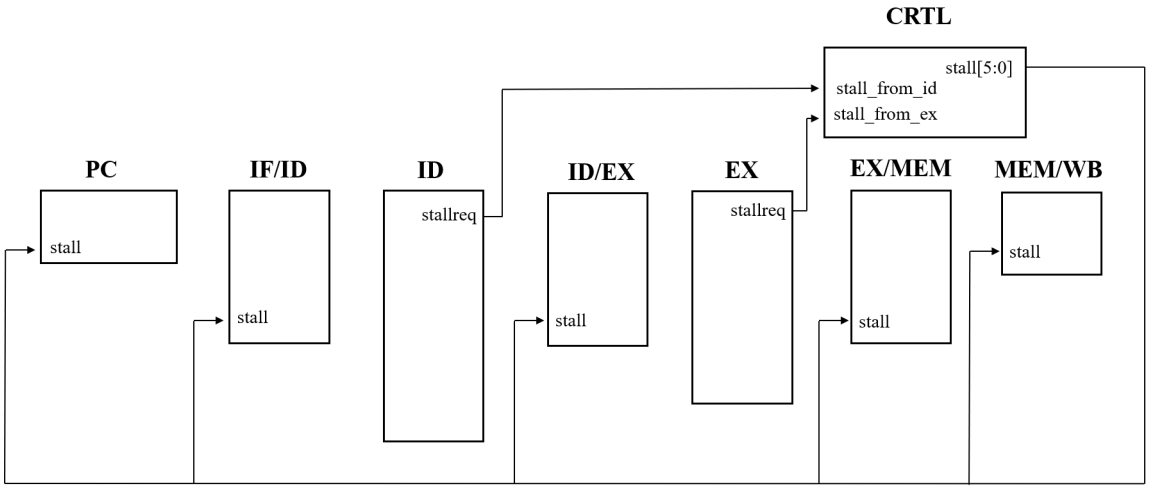


图 1: control 模块示意图

在接受 `stall_req` 后，给出 stall 信号。`stall[5:0]` 是一个宽度为 6 的信号，含义为：

- stall[0] = 1'b1: pc stay still
- stall[1] = 1'b1: if pause
- stall[2] = 1'b1: id pause
- stall[3] = 1'b1: ex pause
- stall[4] = 1'b1: mem pause
- stall[5] = 1'b1: wb pause

对应处理程序如下：

```
always @(*) begin
    if(rst == `RstEnable)
        stall  <= 6'b000000;
    else if(stallreq_from_ex == `Stop)
        stall  <= 6'b001111;
    else if(stallreq_from_id == `Stop)
        stall  <= 6'b000111;
    else
        stall  <= 6'b000000;
    end
end
```

1. 当 EX 阶段请求暂停时，则 IF,ID,EX 暂停，MEM,WB 阶段继续，故 stall 设置为 6'b001111
2. 当 ID 阶段请求暂停时，则 IF,ID 暂停，EX, MEM, WB 阶段继续，故 stall 设置为 6'b000111

而当各阶段寄存器得到 stall 信号为 1'b1 时，会将接下来的相关寄存器变为 `` `ZeroWord ``。

3.2.3 div 模块

为什么要引入专门的除法模块呢，虽然我们可以直接利用 verilog 中的 '/'，但是这样组合逻辑的延迟极高，为了适应这个延迟，时钟周期必须大于这个延迟。导致主频降低，所以 ToyMips

采用多周期除法模块。当进行除法时，我们会将流水线暂停，等待 div 模块的结果。本项目采用试商法实现除法，参考了一生一芯项目中的实现。其状态机描述如下：

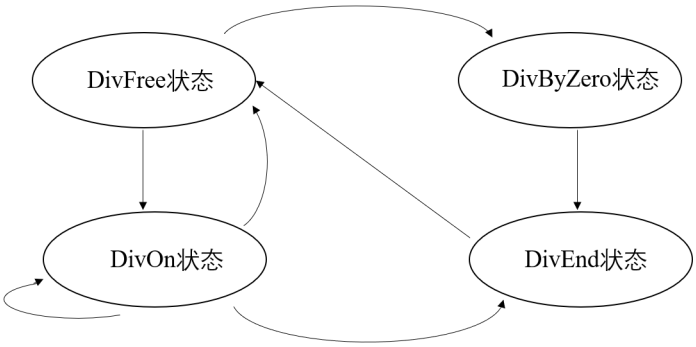


图 2: 除法状态机

说实话，该除法的具体内容也没有细看，所以对具体代码不做介绍，目前就是当他为黑匣子再用。后面要是有空希望还能实现以下多周期的乘法，例如采用华莱士树 +booth 的形式实现。

3.2.4 HILO 寄存器

```
module hilo_reg(

    input             clk,
    input             rst,

    // write port
    input             we,
    input [`RegBus]   hi_i,
    input [`RegBus]   lo_i,

  // read port
  output reg[`RegBus] hi_o,
  output reg[`RegBus] lo_o

);

  always @ (posedge clk) begin
    if (rst == `RstEnable) begin
        hi_o <= `ZeroWord;
        lo_o <= `ZeroWord;
    end else if((we == `WriteEnable)) begin
        hi_o <= hi_i;
        lo_o <= lo_i;
        // for debug
        $display("reg:hi<=%h",hi_i);
        $display("reg:lo<=%h",lo_i);
    end
  end
```

```
endmodule
```

这一部分比较简单，但表示了一类寄存器模块的一般模式，包括在 Register 文件夹下的 CP0_reg, pc_reg, regfile 等。

4 Hazard 处理

4.1 Data Hazard

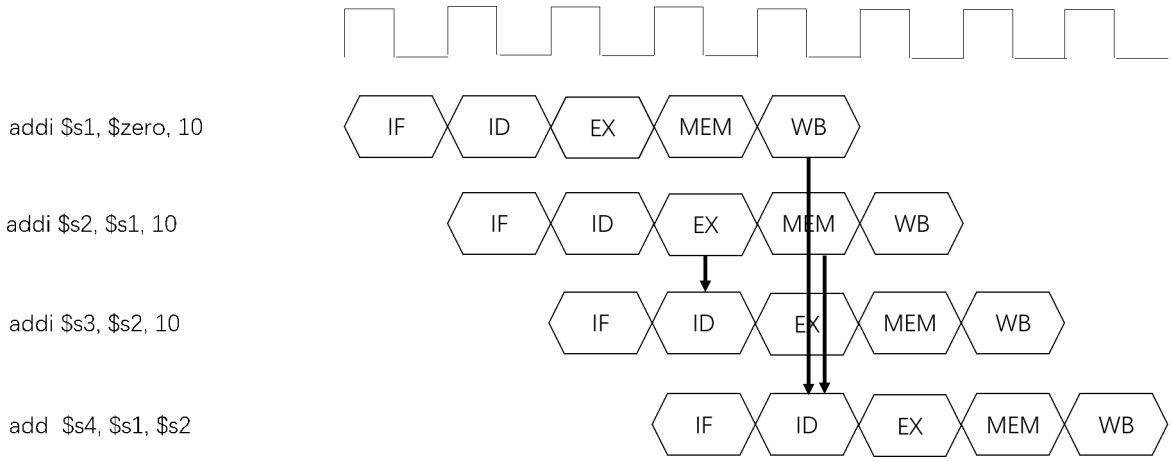


图 3: 数据冒险示意

上图给出了一个 data hazard 的示例。我们需要将 wb, mem, ex 阶段的一些结果通过旁路传到 id 阶段。需要注意的是，对于 EX –> ID 这条旁路，数据必须在 EX 阶段得到结果。这也就导致对 load 相关的指令不适用。Load 最早在 MEM 阶段才能得到结果。

为了解决 load 相关冒险，采用了流水线暂停的设计，也就是此时 id 阶段会申请流水线暂停。

4.2 Control Hazard

Control Hazard 可能在两个模块中被检测到。

- j, jal, jr 无条件跳转导致的 Control Hazard，在 ID 阶段被监测。
- beq, bne, bgez, bgtz, blez, bltz 预测错误导致的跳转，在 EX 阶段被检测。

在 mips32 中，实际上规定采用了**延迟槽 (delay slot)** 的设计解决此类冒险：意思是，转移指令的下一条，总是在延迟槽中。即下一条指令，无论分支、跳转发不发生，下一条指令总是执行（通过 mips 编译器调整了指令的顺序，将一条转移之前的指令（且与转移是否发生的数据无关）放在转移的下一条）。但是由于我们测试是直接给了机器码来测的，并没有这种优化。。。所以我写了一个判断，若发生跳转，则将延迟槽中的指令变成 NOP。另外，我们还需要经分支指令提前到 id 阶段就进行判断。

5 协处理器 CP0 及异常处理

5.1 CP0

查询 MIPS 手册可以知道，协处理器 CP0 中有很多相关的寄存器。其中大部分都是和缓存、MMU、TLB 以及调试相关。目前本项目并未实现这些功能，所以不会全部实现。根据手册，我们将会实现的相关寄存器如下：

表 2: cp0 内的寄存器

标号	寄存器名称	功能描述
9	Count	定时中断控制
11	Status	处理器状态和控制寄存器，决定 CPU 特权等级，使能哪些中断等字段
13	Cause	保存上一次异常原因
14	EPC	保存上一次异常时的程序计数器
15	PRId	处理器版本和标志
16	Config	配置寄存器，用来设置 CPU 参数

具体功能：

1. Count 寄存器和 Compare 寄存器共同工作。Count 寄存器会不停地计数，计数频率与 CPU 时钟频率相当。当 Count 寄存器中的数等于 Compare 寄存器中的数时，会产生定时中断。这也是 MIPS 中产生中断的方式。两者都是可读可写。
2. status 寄存器有 31 位，我们将实现以下功能：

表 3: Status 寄存器

位数	标志	功能描述
[31:28]	CU3CU0	Coprocessor Uasbility，分别表示 CP3 CP0 是否可用，可用时为 1；我们将本字段设置为 4'b0001
[15:8]	IM7-IM0	表示是否屏蔽相应中断 (Interrupt Mask), 0 表示屏蔽，1 表示不屏蔽。MIPS 处理器可以有 8 个中断源，对应 IM 字段的 8 位，其中 6 个中断源是处理器外部硬件中断，另外 2 个是软件中断, 中断是否被处理器响应由 Status 寄存器与 Cause 寄存器共同决定, 如果 Satus 寄存器的 IM 字段与 Cause 寄存器的 IP 字段的相应位都为 1，且 Suans 寄存器的 IE 字段也为 1 时，处理器才响应相应中断。
[1]	EXL	表示是否处于异常级，异常发生时，该字段为 1，处理器会进入内核模式
[0]	IE	表示是否是能中断，1 表示中断使能，0 表示中断禁止

3. Cause 寄存器主要记录最近一次异常发生原因，也控制软件中断请求。

表 4: Cause 寄存器

位数	标志	功能描述
[31]	BD	当异常指令在延迟槽内时，该字段设为 1
[7:2]	IP7-IP2	IP[7:2] 分别表示 5 号到 0 号硬件中断，IP[1:0] 对应 1 号，0 号软件中断
[6:2]	ExtCode	5 位，有 32 位编码，表示中断原因。大部分和 TLB 等相关，对我们有用的有： 8: Sys 系统调用指令 Syscall 10: RI 未定义指令引起异常 12: Ov 整数溢出 13: Tr 自陷指令引起异常
[0]	IE	表示是否是能中断，1 表示中断使能，0 表示中断禁止

4. EPC 寄存器，存储异常返回地址。可读可写

5.2 异常处理

了解了 CP0 相关后，我们来看会触发异常的情况。该项目实现的异常包括以下 6 种：

- 硬件复位
- 中断（包括软件中断，时钟中断）
- syscall 系统调用
- 无效指令
- 溢出
- 自限指令引发异常

发生异常后，理论上会跳转到指定的异常处理程序，但由于并非直接上板测试，所以我们会暂时采取打印异常信息的方式进行。并同时将 PC 置位 PC+4。

在异常处理中，我们需要完成一个**精确异常**的处理。当一个异常发生后，系统的顺序执行会被中断，此时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例程，异常处理结束后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，所以对于异常发生时，流水线上没有执行完的指令必须记住它处于流水线的哪一一个阶段，以便异常处理结束后能恢复执行，这便是精确异常。

这看起来似乎很直观，但在流水线 CPU 中，这意味着我们不可以简单的按照流水线的顺序执行异常。例如：若 lw 将在访存阶段发生异常，假设它的下一条指令为无效指令，会在阶段就被找出，而此时 lw 还在 ex 阶段，并未发现异常。则会导致后出现的异常先被响应的情况。

为了避免上述情况的发生，先发生的异常并不会被立即处理，而是标记异常事件。我们会统一在 wb 时，在 CP0 模块中处理异常。

6 测试

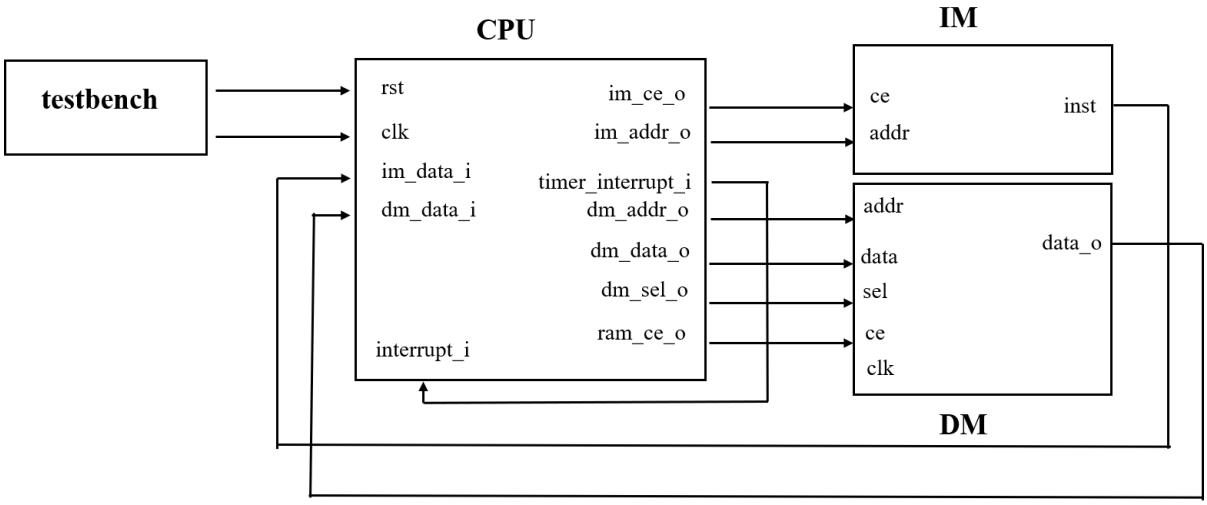


图 4: sopc

我们利用上图的激励信号开始测试。本项目通过交叉编译工具实现了半自动测试，只需要写汇编代码，以及对应的预期寄存器堆中的变化就可以验证正确性。在 test 目录下使用 `make all` 可验证；验证结果如下：

```
ubuntu@VM-4-14-ubuntu ~/D/T/test (main)> make all
./test.sh logic
==> start testing logic.
==> test logic passed.
./test.sh shift
==> start testing shift.
==> test shift passed.
./test.sh move
==> start testing move.
==> test move passed.
./test.sh arithmetic
==> start testing arithmetic.
==> test arithmetic passed.
./test.sh div
==> start testing div.
==> test div passed.
./test.sh jump
==> start testing jump.
==> test jump passed.
./test.sh branch
==> start testing branch.
==> test branch passed.
./test.sh mem
==> start testing mem.
==> test mem passed.
./test.sh forwarding
==> start testing forwarding.
==> test forwarding passed.
==> all tests passed.
```

图 5: 测试结果

我们写了很多测试样例，以 branch 相关指令为例，生成的波形图如下：

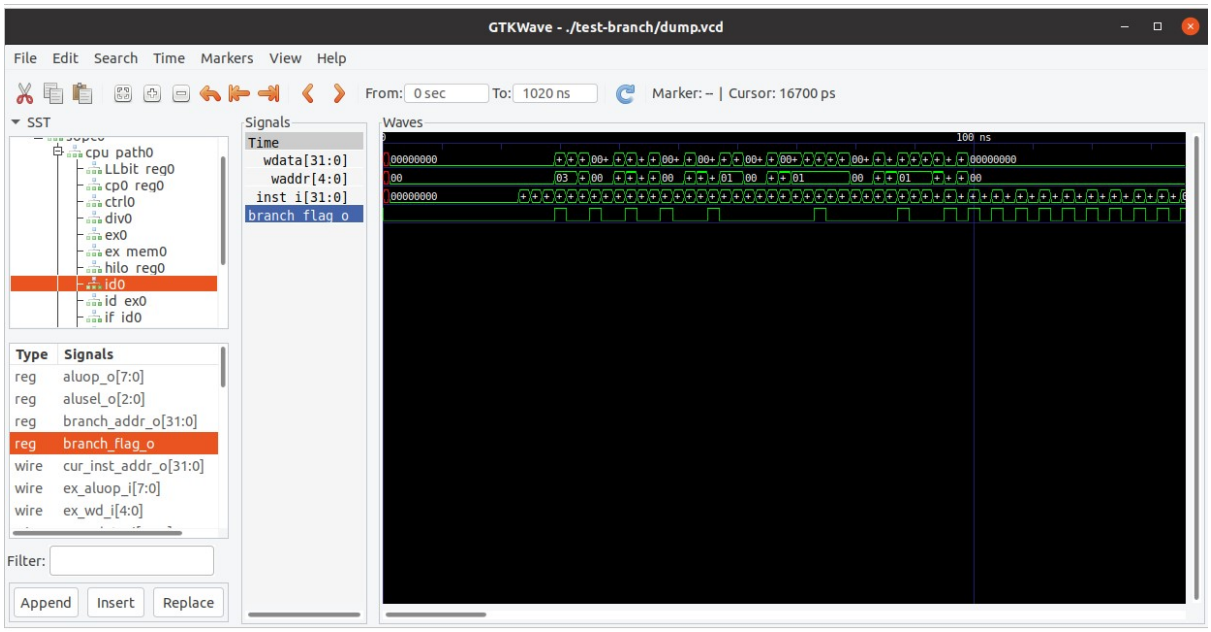


图 6: 测试 branch 生成的波形图

7 其他

7.1 小组分工

- 062020119 刘明杰：负责主要模块编写，包括 if, id, ex, mem, wb，以及参与 CP0 编写，测试系统编写，报告编写
- 062010418 程恩华: 参与异常相关设计，编写阶段寄存器，参与测试系统编写，提供测试数据。
- 072020114 谭子轩: 参与异常相关设计，提供测试数据，报告编写
- 032040113 朱亮: 报告编写，提供测试数据

7.2 reference