

同濟大學

TONGJI UNIVERSITY

《计算机系统实验》

实验报告

实验名称

实验一：CPU 改造、下板

实验成员

何锦洋 2251934

日期

二零二五年 四月 十一日

实验一报告

要求：

[attachment:46c5c792-7780-4bbf-ada6-2dd767accfe4:实验1、Cpu改造实验.pdf](#)

1、实验目的

本项目基于经典MIPS架构实现五级流水线CPU设计，包含取指（IF）、译码（ID）、执行（EX）、访存（MEM）和写回（WB）五个核心阶段。通过模块化开发深入理解各单元功能：IF模块实现PC计数器与指令存储器交互；ID模块完成指令解码与寄存器堆读写控制；EX模块集成ALU运算单元及旁路转发逻辑；MEM模块管理数据存储器访问与load/store指令处理；WB模块实现结果回写与数据冲突解决。

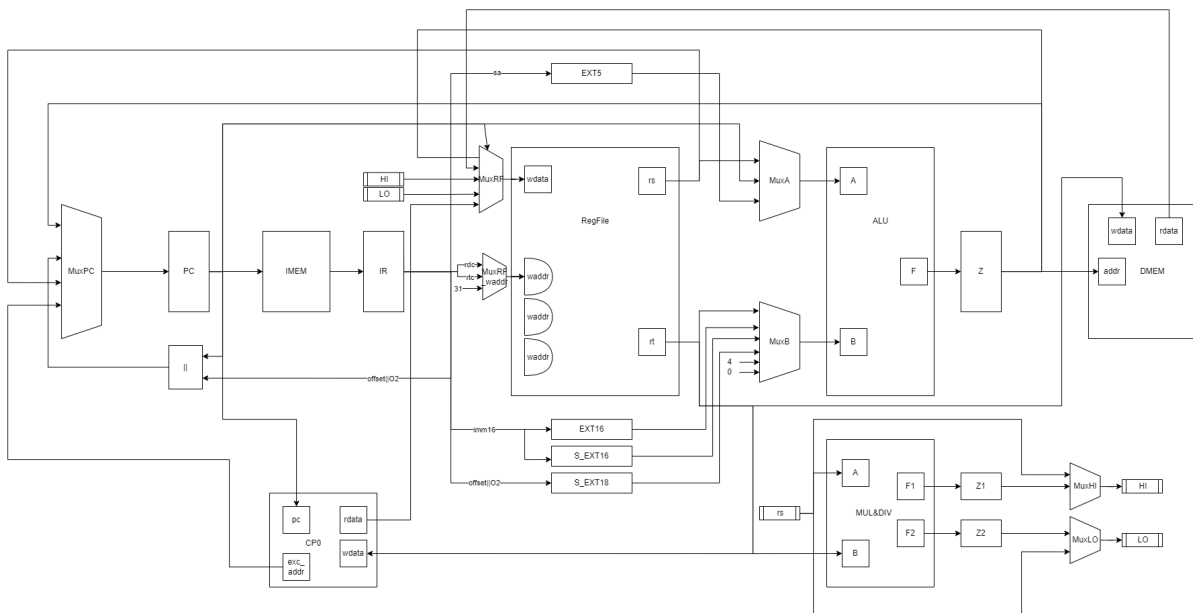
在Verilog实现过程中，重点攻克三大技术难点：

- 流水线冲突处理：**通过插入气泡（Stall）与数据前递（Forwarding）机制，解决RAW/WAR数据依赖问题
- 异常处理架构：**设计四级优先级中断控制器（INTC），支持精确异常处理模式
- 存储层次优化：**采用哈佛架构分离指令/数据存储器，总线宽度优化为32bit

本实现严格遵循教学CPU开发规范，模块间通过标准Wishbone总线互联，预留OS移植所需的MMU接口与特权寄存器组（CP0）。通过该实践，不仅掌握了Verilog的阻塞/非阻塞赋值、时序约束等关键语法，更深入理解了硬件描述语言（HDL）与真实硬件电路（ASIC）的映射关系，为后续实现中断驱动、虚拟内存等操作系统核心功能奠定硬件基础。

2、实验内容

总体架构设计



在原有的54条指令的流水线CPU的基础上进行架构设计

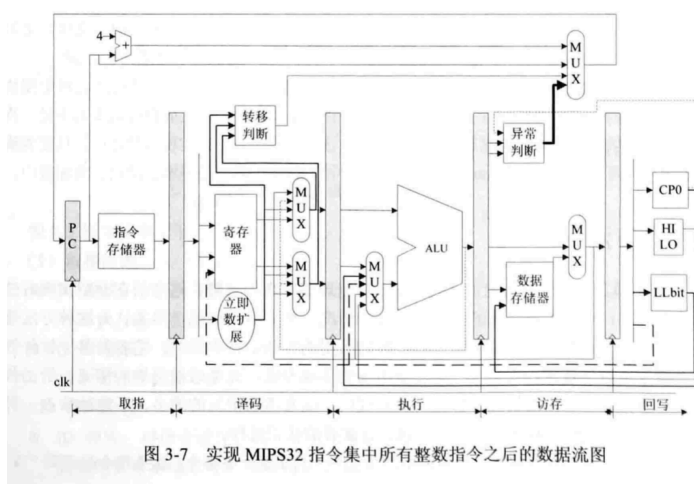
保留的设计

- 五级整数流水线，分别是：取指、译码、执行、访存、回写。
- 哈佛结构，分开的指令、数据接口。
- 32个32位整数寄存器。
- 大端模式。
- 支持延迟转移。
- 采用IP核存储指令
- 指令起始地址为 `0x00400000` ，数据起始地址为 `0x10010000`

改动和新增的设计

- 向量化异常处理，支持精确异常处理。
- 支持6个外部中断。
- 具有32bit数据、地址总线宽度。
- 能实现单周期乘法。
- 新增LLbit，异常判断板块

- 优化流水线结构，大多数指令可以在一个时钟周期完成
- 寻址方式为按字节寻址，数据存储器采用4个8位的存储器构成



增加指令的实现

移动指令

movn: 判断rt寄存器的值，若不为零，则将rs赋值给rd。movz相反

实现思路：译码得到rt寄存器的同时判断是否为零，是否要写。执行阶段确定是否写，写的数据

算术操作指令

madd, maddu: 操作数相乘后和HI, LO相加

msub, musbu: 操作数相乘后和HI, LO相减

实现思路：为了提高流水线效率，将乘累加和乘累减分为两个周期实现，一个周期实现乘法，另一个周期实现加减法。为了使得不同指令实现周期数不同，增加流水线暂停机制，以及运算阶段内部计数功能。使得在处理多周期指令时能暂停流水线，并且能标记当前执行状态。

转移指令

转移指令分为无条件转移和条件转移，为了降低延迟槽大小，对于所有的转移指令的转移地址判断都移动至译码阶段。

ID阶段：对于转移指令且满足条件，输出branch_target_address，branch_flag到IF阶段，此外还会输出next_inst_in_delayslot给ID/EX阶段，并且会在下一个周期输入回ID表示此时在ID的指令是延迟槽指令

部分指令要求存储转移地址，新增link_address输出

加载存储指令

8条加载指令：lb, lbu, lh, lhu, ll, lw, lwl, lwr

6条存储指令：sb, sc, sh, sw, swl, swr

实现思路：将指令传递到执行阶段，根据指令计算加载和存储的地址，在访存阶段进行相应的加载和存储操作

当前假设访存可以在一个周期内完成

异常相关指令

异常涉及异常的判断，CPO状态的读写，地址的跳转等步骤

为了实现异常按照指令执行顺序处理而不是按照异常发生顺序处理，采用精确异常的设计方式。

精确异常概念：当异常发生在指令A时，A之前所有指令正常执行完，A之后的指令被取消。

实现思路：在流水线各个阶段收集异常信息（ID：异常指令，返回指令，EX：自陷，溢出，MEM：中断，并传递到流水线访存阶段。

访存阶段结合CPO相关寄存器值，判断异常是否需要处理，如果要转入入口程序，清除流水线上除写回阶段的全部信息，同时修改CPO相关寄存器值。

其他指令

空指令：nop和ssnop

ssnop可以确保单独占用一个发射周期

很有意思的一点是：nop和ssnop的功能码与sll一致，所以在译码的时候可以就当作sll翻译，意思就是对零号寄存器左移并存回零号寄存器。因此这两个指令无需额外实现，只要保证零号寄存器始终为零即可

sync：保证加载、存储操作的顺序。由于OpenMIPS顺序是固定的，所以该指令无意义等同于nop

pref：缓存预取，也等同于nop

3、实验步骤

完成各个模块的编写

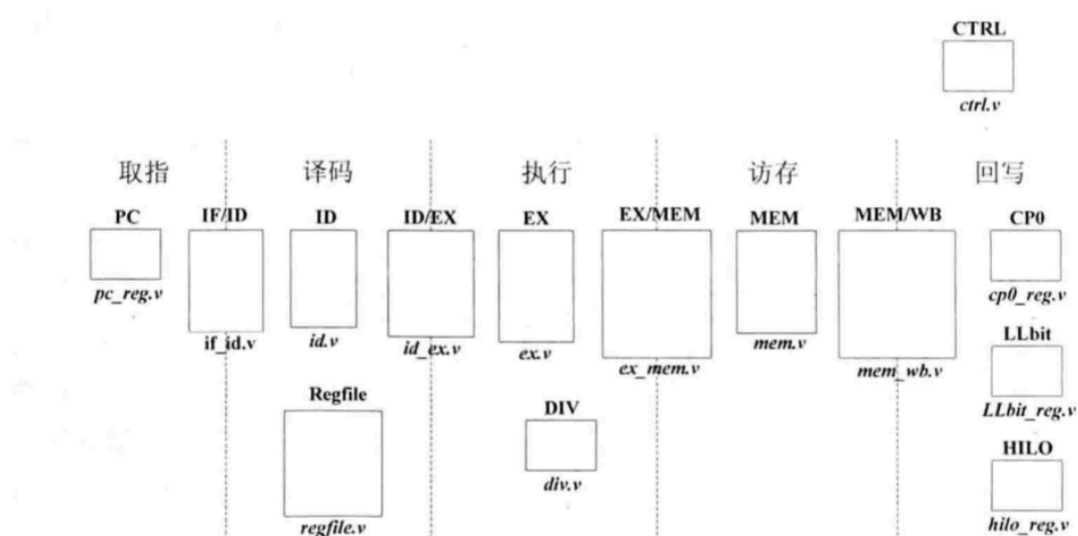
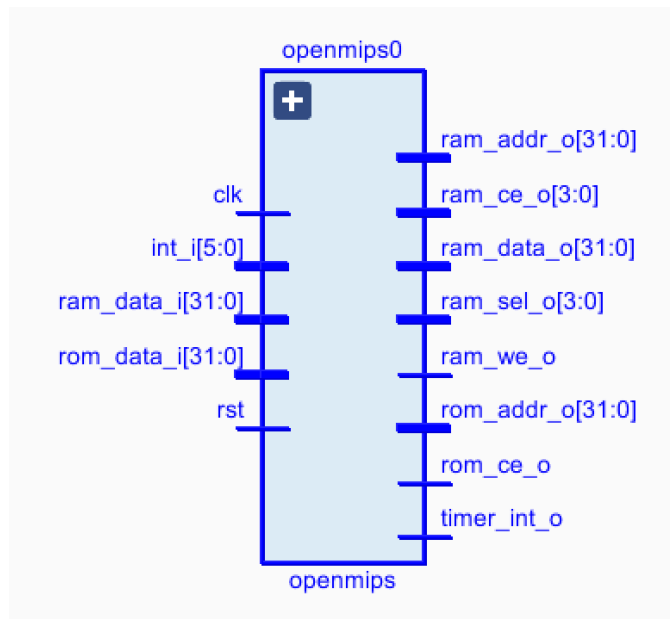


图 3-5 OpenMIPS 流水线各个阶段的模块、对应的文件

顶层模块接口示例

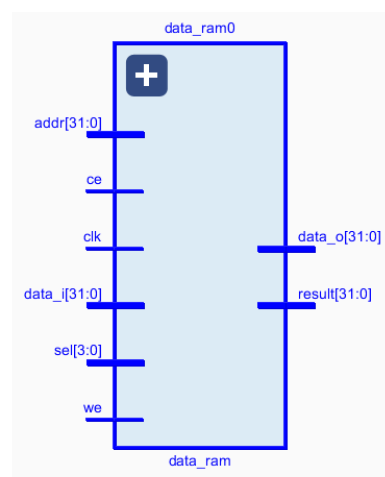
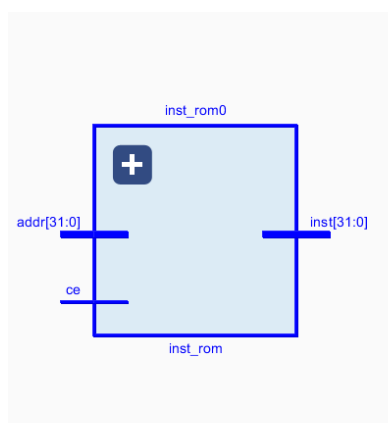
CPU接口：



指令存储器和数据存储器接口

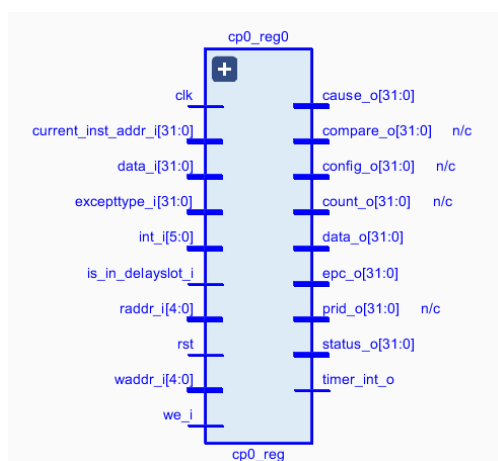
```
inst_rom inst_rom0(
    .ce(rom_ce),
    .addr(relative_inst_addr),
    .inst(inst)
);

data_ram data_ram0(
    .clk(clk),
    .ce(mem_ce_i),
    .we(mem_we_i),
    .addr(relative_mem_addr),
    .sel(mem_sel_i),
    .data_i(mem_data_i),
    .data_o(mem_data_o),
    .result(result_o)
);
```



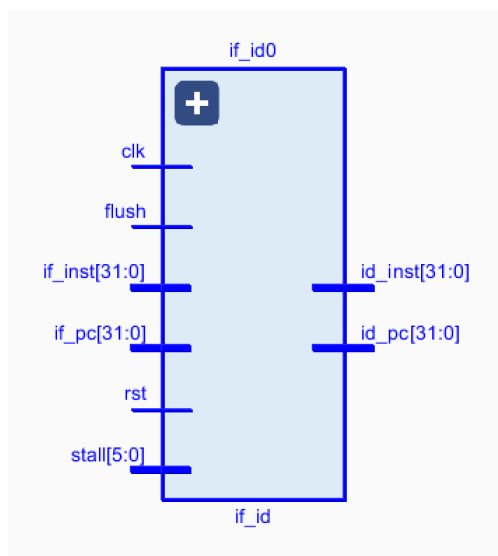
部分模块接口示例

CP0



接口名	作用
rst	复位信号
clk	时钟信号
raddr_i	要读取的CPO中寄存器的地址
int_i	6个外部硬件中断输入
we_i	是否要写CPO中的寄存器
waddr_i	要写的CPO中寄存器的地址
wdata_i	要写入CPO中寄存器的数据
data_o	读出的CPO中某个寄存器的值
count_o	Count寄存器的值
compare 0	Compare寄存器的值
status_o	Status寄存器的值
cause_o	Cause寄存器的值
epc_o	EPC寄存器的值
config_o	Config寄存器的值
prid_o	PRId 寄存器的值
timer_int_o	是否有定时中断发生

IF_ID



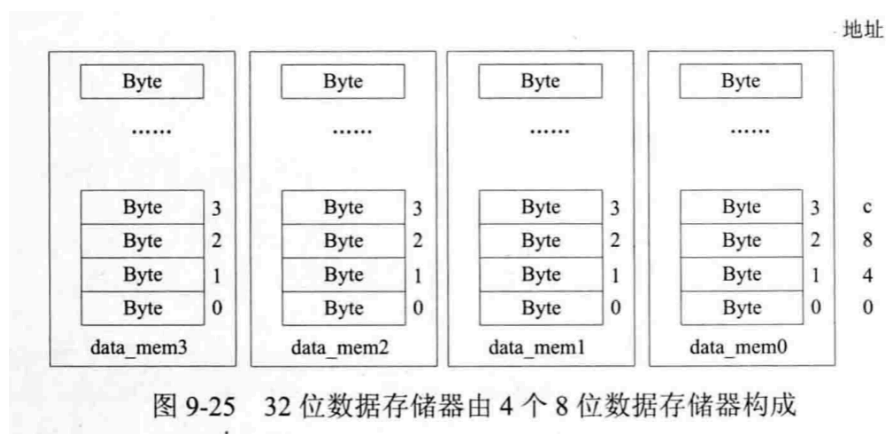
接口名	作用
<code>rst</code>	复位信号
<code>clk</code>	时钟信号
<code>if_pc</code>	取指阶段取出的指令对应的地址
<code>if_inst</code>	取指阶段取出的指令
<code>stall</code>	取指阶段是否暂停
<code>flush</code>	流水线清除信号
<code>id_pc</code>	译码阶段的指令对应的地址
<code>id_inst</code>	译码阶段的指令

寻址实现

OpenMIPS框架选择32位作为机器码，同时也是指令码，寻址方式上OpenMIPS采用按字节方式寻址，

存储器

参考Wishbone设计，MEM模块向存储器传递的`mem_addr`实际上只有[31:2]位有用，最低的两位用于标记`mem_sel`表示取字，半字...



实现按字节寻址，有四个数据存储器，因此对于存储器索引而言，只有当地址增加4，索引增加1。因此地址mem_addr后两位不重要置为0，每次会读取32位的字，每次mem_sel宽度是4，分别对应总线的四个字节

eg：当向存储器写半字时，首先根据指令子类型aluop从寄存器中取半字，重复填充mem_data，并且设置mem_sel，然后存储器根据mem_sel从mem_data中取数，并且存入指定位置

```
if (sel[3] == 1'b1) data_mem3[addr[`DataMemNumLog2+1:2]] <= data_i[31:24];
if (sel[2] == 1'b1) data_mem2[addr[`DataMemNumLog2+1:2]] <= data_i[23:16];
if (sel[1] == 1'b1) data_mem1[addr[`DataMemNumLog2+1:2]] <= data_i[15:8];
if (sel[0] == 1'b1) data_mem0[addr[`DataMemNumLog2+1:2]] <= data_i[7:0];
```

流水线实现

通过四个中间模块传递寄存器的值，实现指令在各个部件中按流水线的方式运行

流水线相关问题：

- 结构相关：存储器访问冲突
- 数据相关：前后指令之间有依赖

- 控制相关：分支指令

数据相关

写后读 RAW

若写和读指令间相隔两个指令，则写指令的WB阶段和读指令的ID阶段重合，为了避免数据相关，在RegFiles中修改一个分支，若写的地址和读的地址相同时，读的数据将直接是写的数据

对于相邻和间隔一个的读写指令，如果写指令的结果是在EX阶段计算得到，则读指令的ID阶段可以从相邻指令的EX阶段或者间隔指令的MEM阶段获得数据

实现：

新增控制信号：是否要写目的寄存器wreg_o、要写的目的寄存器地址wd_o、要写入目的寄存器的数据wdata_o等信息送到译码阶段

对于mf，mt指令

mf从特殊寄存器中读数据是在EX阶段读，mt向特殊寄存器写数据是在WB阶段，因此相邻指令以及间隔一条指令会有数据相关问题，解决方法是增加从MEM，WB到EX阶段的数据通路。

实现：

新增whilo, hi, lo信号。EX阶段判断这些信号，EX阶段最终给出的HI，LO输出就是当前指令 需要在WB阶段写入HILO寄存器的值（HILO都会被写入，所以要同时准备）

其他相关问题

- 读后写 WAR：先读再写的指令若在读之前就写了则导致读不正确
- 写后写 WAW：由于写只发生在WB阶段，因此OpenMIPS不会出现该数据相关问题

Load 相关

分支判断在ID阶段，判断依赖寄存器，通过EX得到的可以通过重定向解决，但是通过访存得到的寄存器的值无法解决冲突

因此解决方法：在ID检查当前指令和上一条指令是否存在load相关，如果存在，就让译码，取指暂停一次。由于访存和执行仍能继续，所以下一个时钟周期，上一条指令在访存阶段，而当前指令仍在译码阶段。然后将访存阶段的数据前推到译码阶段（已经实现了）

流水线暂停机制

对于乘除指令，需要暂停流水线，也就是需要保持取指令地址PC的值不变，同时流水线各个阶段寄存器的值不变（改进方法是可以访存，回写继续执行）

添加CTRL模块，接受各个阶段（ID，EX）传来的流水线暂停请求，从而控制流水线各阶段运行

对于相邻的两个阶段，若左阶段不暂停，取左阶段传来的，若左阶段暂停右阶段继续，则取空指令作为右阶段的输入，否则若都暂停，则取自身的值。

```
if (stall[1] == `Stop && stall[2] == `NoStop) begin
    id_pc  <= `ZeroWord;
    id_inst <= `ZeroWord;
end
else begin
    if (stall[1] == `NoStop) begin
        id_pc  <= if_pc;
        id_inst <= if_inst;
    end
end
end
```

4、实验结果

测试和调试过程

前仿真

通过系统函数从文本文件中读取指令集，通过系统函数输出部分提示性内容

```
$display("Loading memory from inst_rom.data...");  
$readmemh("E:\\DigitCircuit\\computer_componont\\OpenMIPS\\CPU89_code\\inst_rom.data", inst_mem);
```

终端输出提示：

```
# run 4000ns  
Loading memory from inst_rom.data...  
INFO: [USF-XSim-96] XSim completed. Design snapshot 'openmips_min_sopc_tb_behav' loaded.  
INFO: [USF-XSim-97] XSim simulation ran for 4000ns
```

选择跟踪所有信号，并通过vivado自带的仿真器观察各个信号的波形图，与书中相关测试用例以及程序代码一一比对

【比对结果】

测试用例

将老师提供的测试汇编程序通过MARS编译运行

EditExecute

Text Segment

Bkpt	Address	Code	Basic	Source
	0x0040017c	0x0c100285jal	0x00400a14	123: jal func_test9
	0x00400180	0x00000000nop		124: nop
	0x00400184	0x0c100070jal	0x004001c0	125: jal func_reset
	0x00400188	0x00000000nop		126: nop
	0x0040018c	0x0c1002a1jal	0x00400a84	127: jal func_test10
	0x00400190	0x00000000nop		128: nop
	0x00400194	0x0c100070jal	0x004001c0	129: jal func_reset
	0x00400198	0x00000000nop		130: nop
	0x0040019c	0x0c100300jal	0x00400c00	131: jal func_test11
	0x004001a0	0x00000000nop		132: nop
	0x004001a4	0x0c100070jal	0x004001c0	133: jal func_reset
	0x004001a8	0x00000000nop		134: nop
	0x004001ac	0x20050001addi	\$5,\$0,0x00000001	136: addi \$5,\$0,1
	0x004001b0	0x3c011001lui	\$1,\$1,0x00001001	137: sw \$5,ANSCODE
	0x004001b4	0xac250000sw	\$5,0x00000000(\$1)	
	0x004001b8	0x0810006ej	0x004001b8	139: j dead_loop

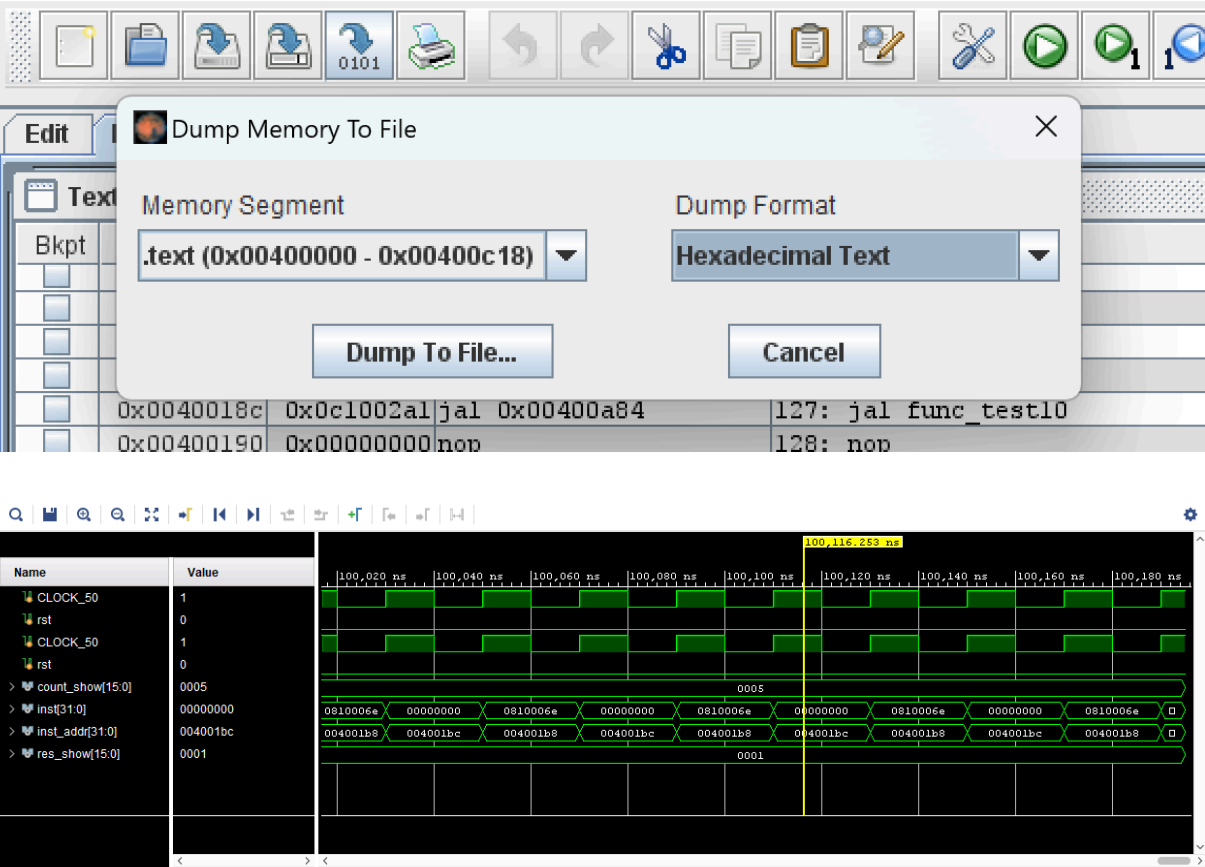
Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0xffffffff	0x070e0000	0x0b0a0000	0x0f0e0d0c	0x13120000	0x00000014	0x0000000f	0x00000015
0x10010020	0x0000001c	0x00000024	0x0000002d	0x00000037	0x00000042	0x0000004e	0x0000005b	0x00000069
0x10010040	0x00000078	0x00000088	0x00000099	0x000000ab	0x000000b6	0x000000c2	0x000000e7	0x000000fd
0x10010060	0x00000114	0x0000012c	0x00000145	0x0000015f	0x0000017a	0x00000196	0x000001b3	0x000001d1
0x10010080	0x000001f0	0x00000210	0x00000231	0x00000253	0x00000276	0x0000029a	0x000002bf	0x000002e5
0x100100a0	0x0000030c	0x00000334	0x0000035d	0x00000387	0x000003b2	0x000003de	0x0000040b	0x00000439
0x100100c0	0x00000468	0x00000498	0x000004c9	0x000004fb	0x0000052e	0x00000562	0x00000597	0x000005cd
0x100100e0	0x00000604	0x0000063c	0x00000675	0x000006af	0x000006ea	0x00000701	0x00000734	0x0000076a
0x10010100	0x00000713	0x0000074f	0x00000792	0x000007d7	0x00000814	0x00000862	0x000008b1	0x000008e6
0x10010120	0x000008c7	0x000008eb	0x00000912	0x0000095c	0x000009af	0x000009e9	0x00000a38	0x00000a87
0x10010140	0x00000a2b	0x00000a77	0x00000ab6	0x00000af8	0x00000b3d	0x00000b95	0x00000be0	0x00000c31
0x10010160	0x00000c4f	0x00000c93	0x00000cd1	0x00000d17	0x00000d6d	0x00000db3	0x00000e09	0x00000e6f
0x10010180	0x00000e73	0x00000ebf	0x00000f03	0x00000f60	0x00000fb2	0x00000ff9	0x0000105a	0x000010b9
0x100101a0	0x0000109b	0x000010e6	0x00001132	0x00001190	0x000011f3	0x0000125c	0x000012c5	0x00001337

可以看到Mars运行的结果是第零号寄存器的值为 **0xffffffff8**，即表示十进制数的-8。说明Mars在第8类也就是中断异常处理指令发生错误，并且程序陷入死循环。因此该测试用例无法与Mars比对结果。

波形图

接着将老师提供的测试程序通过Mars转为机器码，经过仿真观察波形和预期一致



预期效果：最终程序陷入死循环，并且存储器相对地址为0处的输出（计数+函数返回值）为预期的变化的计数过程以及函数返回1（合法的返回值）

综合与下板

IP核

对于指令存储器的构建，从以系统函数读取方式更改为创建IP核的方式

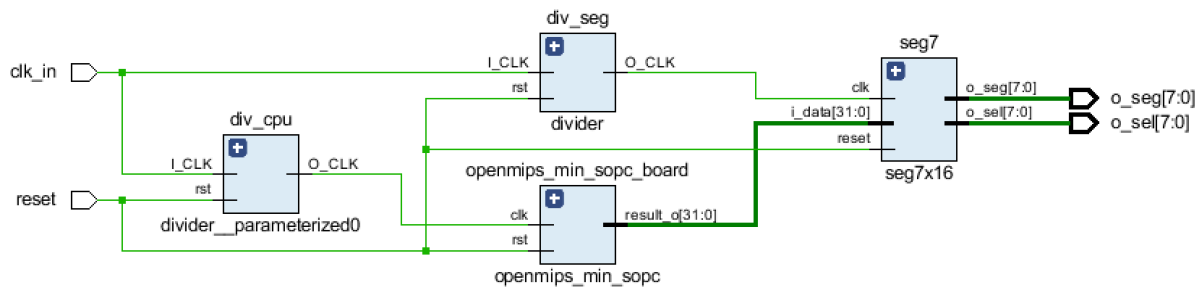
```

wire [`InstBus] rom_o;
irom irom (
    .a(addr[`InstMemNumLog2+1:2]),
    .spo(rom_o)
);

```

修改顶层模块

引入新模块：分频器和七段数码管



约束文件

编写约束文件，将时钟信号与系统时钟连接，reset信号与系统管脚相连，七段数码管与系统数字显示管脚相连

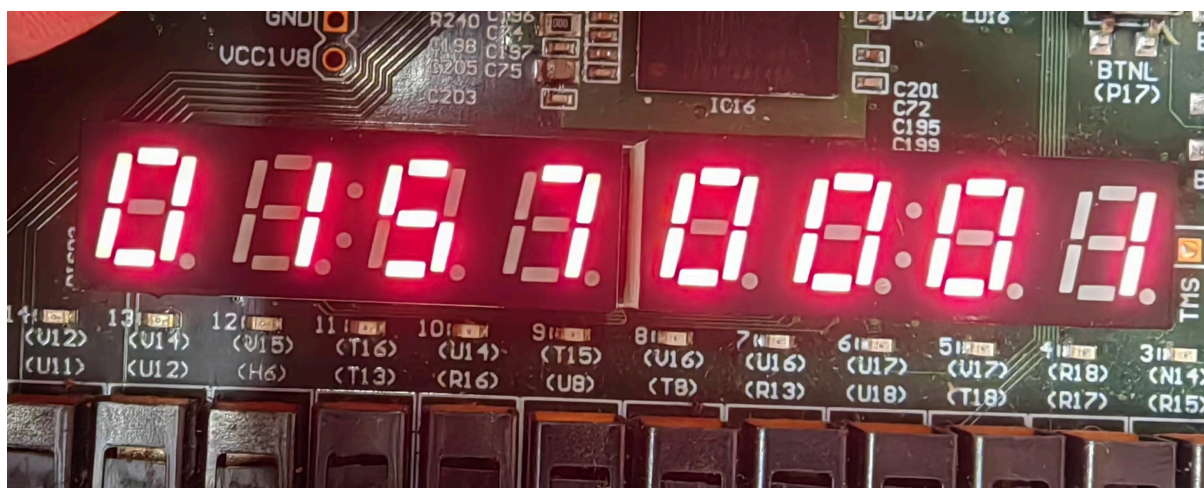
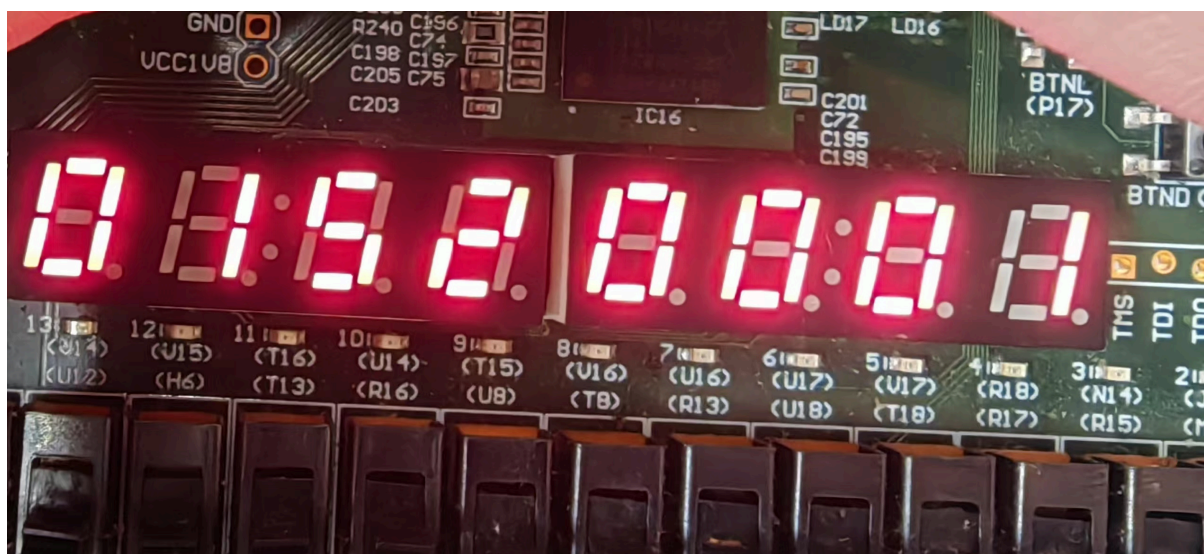
```

create_clock -period 100.000 -name clk_pin -waveform {0.000 50.000} [ge
t_ports clk_in]
set_input_delay -clock [get_clocks *] 1.000 [get_ports reset]
set_output_delay -clock [get_clocks *] 0.000 [get_ports -filter { NAME =~
"*" && DIRECTION == "OUT" }]

set_property PACKAGE_PIN E3 [get_ports clk_in]
set_property PACKAGE_PIN N17 [get_ports reset]
...

```

下板结果



说明：在程序运行到死循环之前，显示的值（存储器0号）都是存储器的初始值0。当程序运行到死循环时

- 显示的低16位为1，代表函数的返回值为1，也就是11条测试用例均通过。
- 显示的高16位为不断递增的数，其表示随着时钟中断不断增加的时钟中断计数

5、实验总结

一、关键问题与解决方案

1. 异常入口地址配置错误

现象与影响：

系统启动后程序陷入死循环，经交叉编译工具Mars比对发现异常处理程序未正确触发。

根因分析：

异常入口地址默认指向主程序入口地址（0x00400000），导致异常处理与正常执行流冲突。

解决方案：

- 遵循MIPS32规范，将异常入口地址重定向至0x00400004，确保与主程序隔离。
- 在协处理器CP0中实现 `EPC`（异常程序计数器）和 `Cause` 寄存器，记录异常类型及返回地址。
- 添加专用异常返回指令 `ERET`，通过硬件逻辑自动恢复 `EPC` 至PC寄存器。

2. 加载-跳转流水线冲突

冲突场景：

`LW $t0, 0($s0)`（MEM阶段写回）后紧跟 `BEQ $t0, $zero, label`（ID阶段读取\$t0），导致条件判断值未更新。

解决方案：

- **流水线暂停机制：**检测到数据冲突时插入2个NOP周期，关键逻辑如下：

```
else if (stallreq_from_ex == `Stop) begin
    stall <= 6'b001111; // 访存，回写不暂停
    flush <= 1'b0;
end
else if (stallreq_from_id == `Stop) begin
    stall <= 6'b000111; // 执行，访存，回写不暂停
    flush <= 1'b0;
end
else begin
    stall <= 6'b000000;
    flush <= 1'b0;
```

```
new_pc <= `TextBegin;  
end // if
```

二、实践心得与理论认知

在本次基于OpenMIPS架构的CPU设计实践中，我深刻理解了计算机体系结构的核心原理与工程实现方法。通过实现89条MIPS指令集，我对协处理器CP0的中断异常处理机制有了系统认知。CP0通过状态寄存器与异常程序计数器（EPC）的协同工作，实现了精确异常处理与中断嵌套，其优先级判断逻辑与现场保护机制显著提升了系统可靠性。在存储器设计环节，不同位宽存储体的联合编址方案验证了哈佛架构的优势，指令存储器32位等宽存取与数据存储器的字节寻址能力，通过地址对齐电路与符号扩展模块的配合，有效平衡了存储效率与访问灵活性。

寻址方式的实现过程揭示了地址映射的本质特征。基址寻址时地址寄存器的动态偏移计算、PC相对寻址的分支延迟槽设计，都需要精确的地址生成与存储体索引转换。特别是在实现load/store指令时，字节/半字/字访问模式通过地址低两位的译码实现了存储体选择，这种硬件级地址解码机制使我直观认识到对齐访问对性能的影响。HI/LO寄存器组的扩展应用体现了MIPS架构的独特设计思想，在实现乘累加指令时，这两个专用寄存器通过多周期操作实现了64位中间结果的暂存，其硬件互锁机制确保了运算流水线的正确性。

在工程实现层面，Verilog的结构化编程方法展现出显著优势。采用always块配合case语句实现的指令译码器，通过层次化状态机控制的数据通路，相比数据流描述具有更好的可维护性。自顶向下的模块化设计策略大幅降低了开发复杂度：首先构建五级流水线框架，再逐级实现取指、译码、执行等模块，每个阶段通过仿真验证后再进行集成。这种分阶段推进的开发模式，使得在实现延迟槽处理、数据前推等复杂机制时能快速定位问题。通过本次实践，我不仅掌握了CPU设计的核心方法，更培养了硬件描述语言工程化实现的系统思维，为后续复杂系统开发奠定了坚实基础。