合肥工業大學

2021~2022 学年 第二学期

《系统硬件综合设计》

设计报告

目　录

0 写在前面

本次课程设计我完成的是一个基于 mips32 的具有五级流水的，包含了 21 条指令的指令集的 cpu，实现了 mips32 三种基本类型的指令，解决了在流水过程中的数据相关和控制相关，并且对每条指令进行一个仿真测试，可以完成一些基本程序的运行，并将 cpu 综合到开发板上面，通过 led 灯将指令流水展示出来，最后写了一个函数来对 cpu 进行一个上板测试。

该课程设计涉及的相关课程很多，主要的有丁贤庆老师的数字逻辑课程，阙夏老师的计算机组成原理课程，李建华老师的计算机体系结构课程，张本宏老师的汇编语言课程，需要自己学习的知识有 Verilog 硬件语言和 FPGA 实验板的相关知识。

整个的设计过程全部由我一个人独立完成，之所以选择一个人来完成是因为这样可以让我思路逻辑更加清晰，并且可以对 cpu 有一个更全面的认识，对我未来发展有很大帮助。

1 设计要求

基于先修课程，根据系统设计思想，使用硬件描述语言设计实现一款基于 MIPS32，ARM，RISC-V 或者自定义指令集的微处理器（CPU）。

要求：完成单周期 CPU 设计，或多周期 CPU 设计，或 5 级流水线 CPU 设计（递进式、难度依次提升。所有学生必须至少完成单周期 CPU 的设计工作），并将设计的 CPU 下载至 FPGA 开发板（ego-1）上运行。以此贯穿数字逻辑、计算机组成原理、计算机体系结构课程，实现从逻辑门至完整 CPU 处理器的设计[1]。

2 设计思路

2.1 数据流通路

本次课程设计完成的 cpu 总共有十二个部分，分别包括了 PC 寄存器、id 译码模块、ex 执行模块、mem 访存模块、寄存器堆、流水线暂停模块、指令存储器和数据存储器，以及四个流水线寄存器共同组成，在后面将一一对每个模块进行介绍，实验过程中参考了雷思磊的《自己动手写 cpu》[1]，这本书对我刚开始独立完成 cpu 框架的搭建极为重要，为我打开了设计 cpu 的大门，完成了基于 openmips 的 cpu 的设计，本次课设所实现的 cpu 数据通路如下：

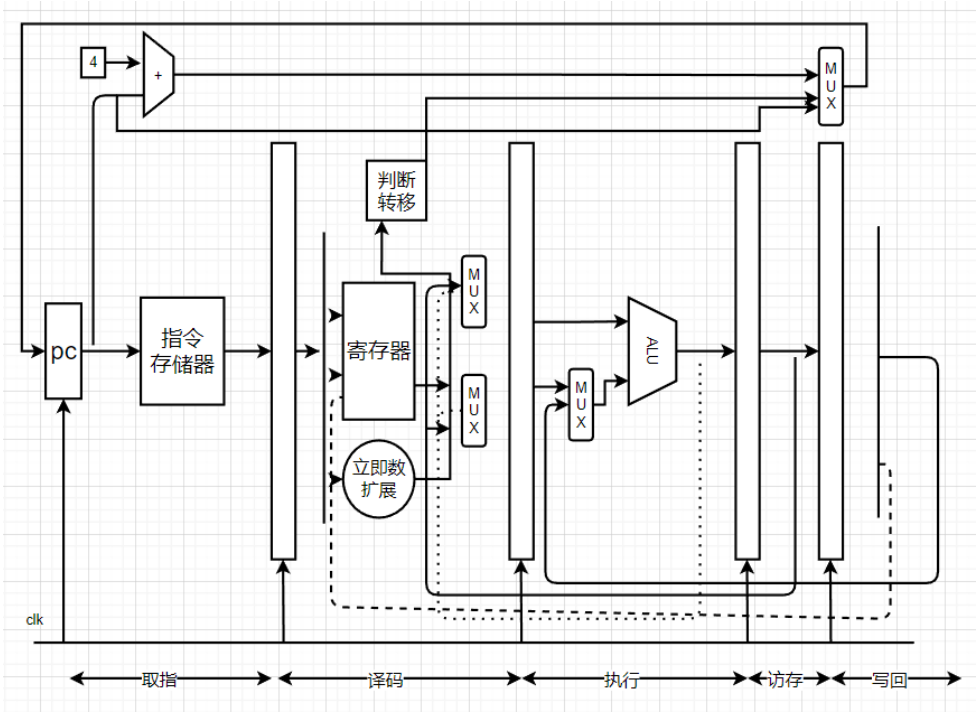


图 1 cpu 数据通路图

以此构成了五级流水线的 mips32cpu，各个阶段完成的主要工作如下：

1. **取指:**取出指令存储器中的指令，PC 值递增，准备取下一条指令。

2. **译码:**对指令进行译码，依据译码结果，从 32 个通用寄存器中取出源操作数，有的指令要求两个源操作数都是寄存器的值，比如 or 指令，有的指令要求其中一个源操作数是指令中立即数的扩展，比如 ori 指令，所以这里有两个复用器，用于依据指令要求，确定参与运算的操作数，最终确定的两个操作数会送到执行阶段。

3. **执行**阶段:依据译码阶段送入的源操作数、操作码，进行运算，对于 ori 指令而言，就是进行逻辑“或”运算，运算结果传递到访存阶段。

4. **访存**阶段:对于 ori 指令，在访存阶段没有任何操作，直接将运算结果向下传递到回写阶段。

5. **回写**阶段:将运算结果保存到目的寄存器。

对 cpu 进行封装，封装后的 cpu 效果如下：

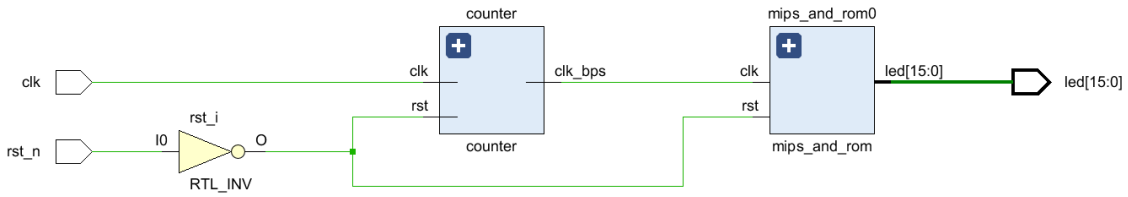


图 2 封装效果图 1

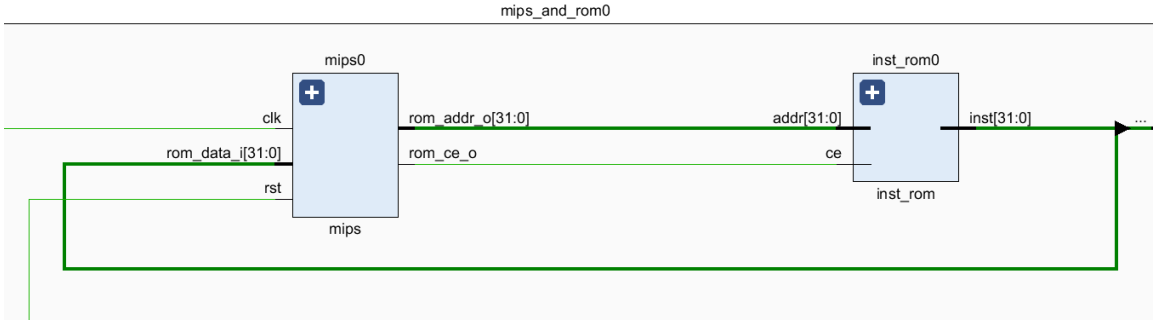


图 3 封装效果图 2

2.2 模块介绍

2.2.1 基本核心模块

（1）PC 寄存器

PC 的作用是给出指令地址，其接口如下：

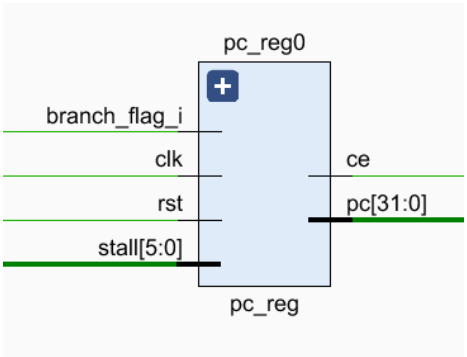


图 4 PC 寄存器

接口名	宽度(bit)	输入/输出	作 用
rst	1	输入	复位信号
clk	1	输入	时钟信号

pc	32	输出	要读取的指令地址
branch_flag_i	1	输入	判断是否是跳转
ce	1	输出	指令存储器使能信号
stall	5	输入	判断 pc 寄存器是否暂停

表 1 PC 寄存器接口

（2）id 模块（译码）

ID 模块的作用是对指令进行译码，得到最终运算的类型、子类型、源操作数 1、源操作数 2、要写入的目的寄存器地址等信息，其中运算类型指的是逻辑运算、移位运算、算术运算等，子类型指的是更加详细的运算类型。

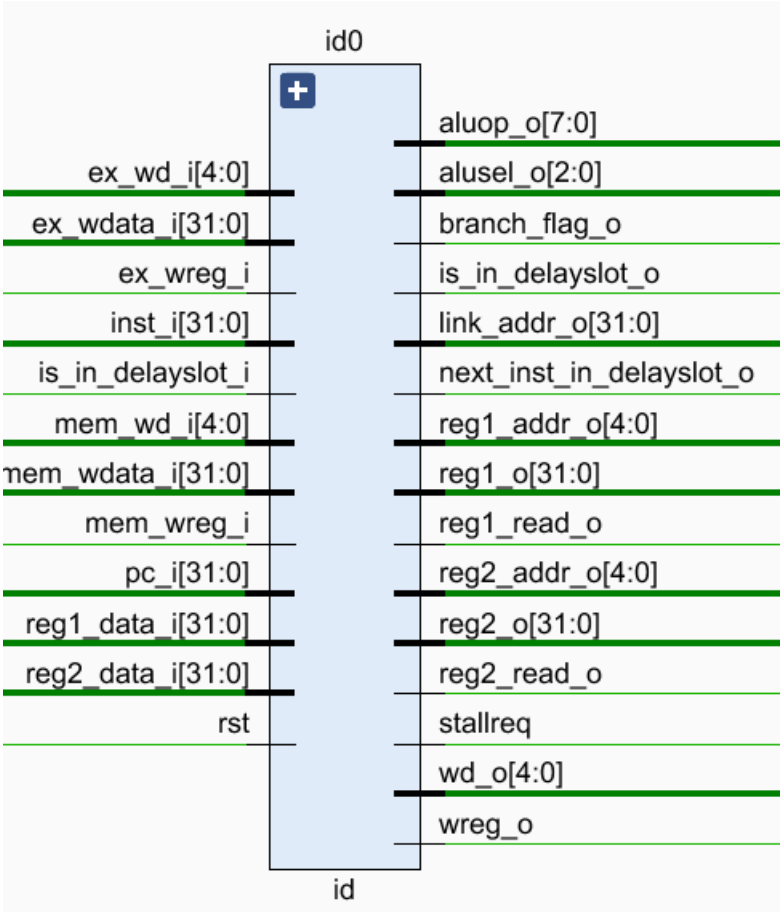


图 5 id 模块

接口名	宽度(bit)	输入/输出	作 用
rst	1	输入	复位信号
pc_i	32	输入	指令地址
reg1_data_i	32	输入	从 Regfile 输入的第一个读寄存器端口的输入

reg2_ data_i	32	输入	从 Regfile 输入的第二个读寄存器端口的输入
ex_wd_i	5	输入	ex 阶段数据相关的地址
ex_wdata_i	32	输入	ex 阶段数据相关的数据
ex_wreg_i	1	输入	ex 阶段是否需要写入
mem_wd_i	5	输入	mem 阶段数据相关的地址
mem_wdata_i	32	输入	mem 阶段数据相关的数据
mem_wreg_i	1	输入	mem 阶段是否需要写入
inst_i	32	输入	需要译码的指令
reg1_read_o	1	输出	Regfile 模块的第一个读寄存器端口读使能信号
reg2_read_o	1	输出	Regfile 模块的第二个读寄存器端口读使能信号
reg1_addr_o	5	输出	Regfile 模块的第一个读寄存器端口读地址信号
reg2_addr_o	5	输出	Regfile 模块的第二个读寄存器端口读地址信号
aluop_o	8	输出	译码阶段的指令要进行的运算的子类型
alusel_o	3	输出	译码阶段的指令要进行的运算的类型
reg1_o	32	输出	译码阶段的指令要进行的运算的源操作数 1
reg2_o	32	输出	译码阶段的指令要进行的运算的源操作数 2
wd_o	5	输出	译码阶段的指令要写入的目的寄存器地址
wreg_o	1	输出	译码阶段的指令是否有要写入的目的寄存器
is_in_delayslot_i	1	输入	是否是延迟槽指令
branch_flag_o	1	输出	该条指令是否需要跳转
is_in_delayslot_o	1	输出	是否是延迟槽指令
link_addr_o	32	输出	跳转地址
next_inst_in_delayslot_o	32	输出	下一条指令是否是延迟槽指令
stallreq	1	输出	是否需要流水线暂停

表 2 id 模块接口

（3）ex 模块（执行）

EX 模块会从 ID/EX 模块得到运算类型 alusel_i、运算子类型 aluop_i、源操作数 regl_i、源操作数 reg2_i、要写的目的寄存器地址 wd_i，EX 模块会依据这些数据进行运算。

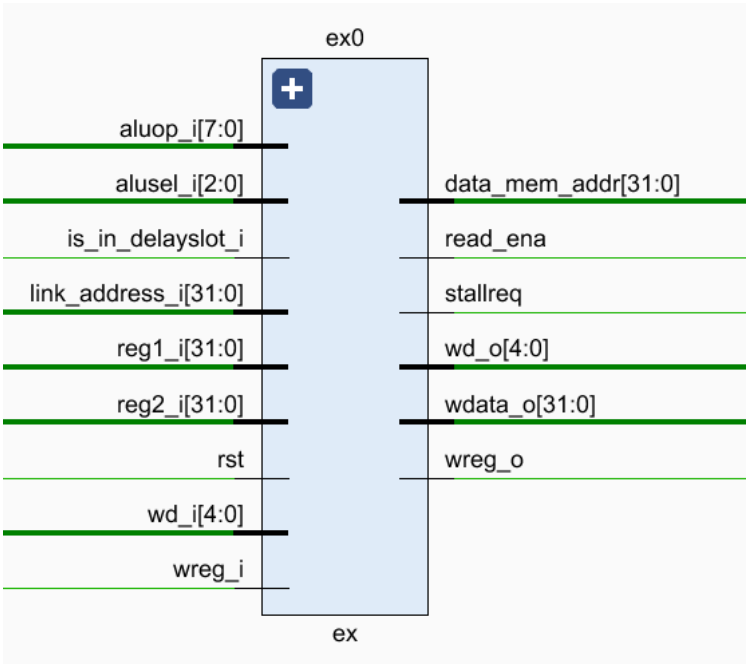


图 6 ex 模块

接口名	宽度(bit)	输入/输出	作 用
aluop_i	8	输入	指令要进行的运算的子类型
alusel_i	3	输入	指令要进行的运算的类型
is_in_delayslot_i	1	输入	是否是延迟槽指令
link_address_i	32	输入	跳转指令地址
reg1_i	32	输入	操作数 1 的值
reg2_i	32	输入	操作数 2 的值
rst	1	输入	复位信号
wd_i	5	输入	写入的寄存器地址
wreg_i	1	输入	是否写寄存器或写回存储器
data_mem_addr	32	输出	写回的数据存储器地址
read_ena	1	输出	数据存储器使能
stallreq	1	输出	是否需要流水线暂停
wd_o	5	输出	写入寄存器的地址
wdata_o	32	输出	需要写入的数据
wreg_o	1	输出	写使能

表 3 ex 模块接口

（4）mem 模块

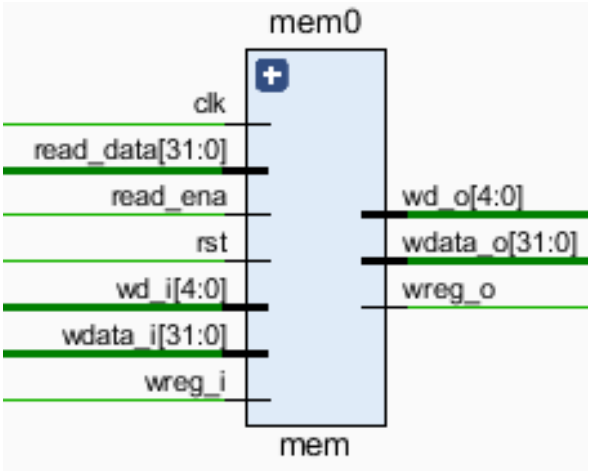


图 7 mem 模块

接口名	宽度(bit)	输入/输出	作 用
clk	1	输入	时钟信号
read_data	32	输入	访存时读的数据
read_ena	1	输入	数据存储器使能
rst	1	输入	复位信号
wd_i	5	输入	写寄存器的地址
wdata_i	32	输入	需要写的数据或读内存地址
wreg_i	1	输入	写使能
wd_o	5	输出	写入寄存器的地址
wdata_o	32	输出	写入寄存器的值
wreg_o	1	输出	输出写使能

（5）regfile 寄存器堆

Regfile 模块实现了 32 个 32 位通用整数寄存器，可以同时进行两个寄存器的读操作和一个寄存器的写操作。

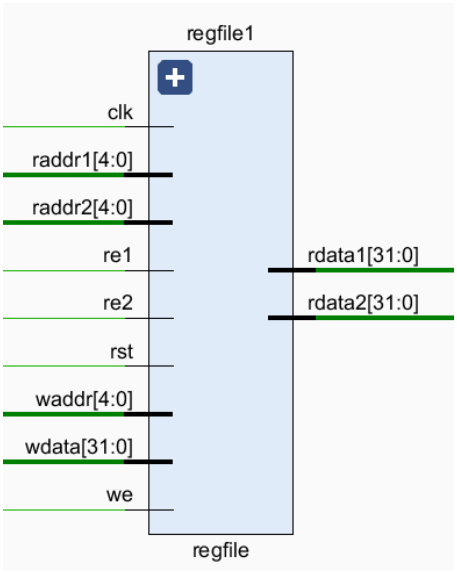


图 8 寄存器堆模块

接口名	宽度(bit)	输入/输出	作 用
clk	1	输入	时钟信号
raddr1	5	输入	1 号读端口地址
raddr2	5	输入	2 号读端口地址
re1	1	输入	1 号读端口使能
re2	1	输入	2 号读端口使能
rst	1	输入	复位信号
waddr	5	输入	写端口地址
wdata	32	输入	写入数据值
we	1	输入	写使能
rdata1	32	输出	1 号数据输出端口
rdata2	32	输出	2 号数据输出端口

2.2.2 流水线寄存器

（1）取指译码阶段

IF/ID 模块的作用是暂时保存取指阶段取得的指令，以及对应的指令地址，并在下一个时钟传递到译码阶段。

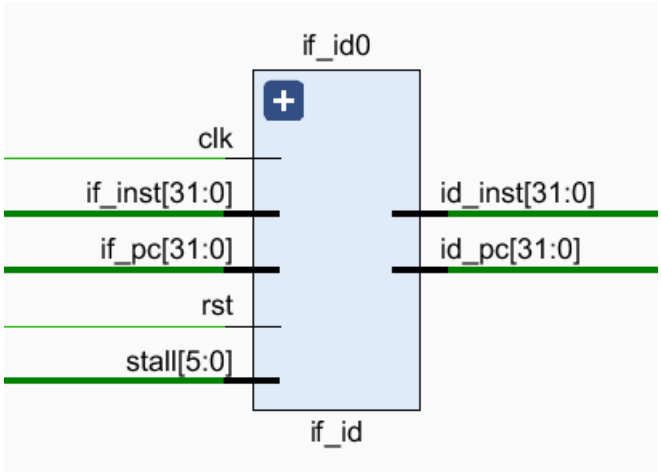


图 9 if_id 模块

接口名	宽度(bit)	输入/输出	作 用
clk	1	输入	时钟信号
if_inst	32	输入	取指阶段输入的机器指令
if_pc	32	输入	取指阶段输入的指令地址
rst	1	输入	复位信号
stall	6	输入	流水线暂停信号
id_inst	32	输出	输出到译码阶段的机器指令
id_pc	32	输出	输出到译码阶段的指令地址

表 4 if_id 模块

（2）译码执行阶段

ID_EX 模块作用是将译码阶段取得的运算类型、源操作数、要写的目的寄存器地址等结果，在下一个时钟传递到流水线执行阶段。

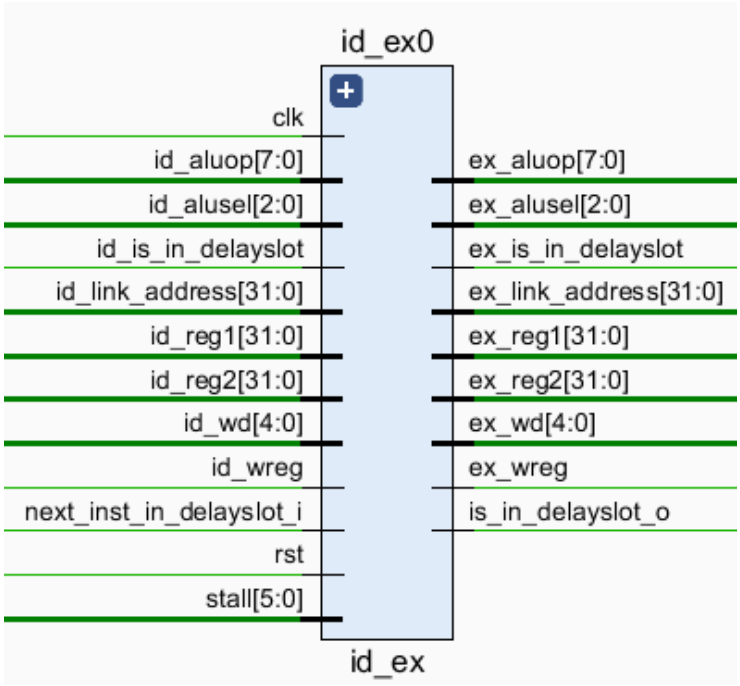


图 10 id_ex 模块

接口名	宽度(bit)	输入/输出	作 用
clk	1	输入	时钟信号
id_aluop	8	输入	译码阶段的指令要进行的运算的子类型
id_alusel	3	输入	译码阶段的指令要进行的运算的类型
id_is_in_delayslot	1	输入	译码阶段的判断是否是延迟槽指令
id_link_address	32	输入	译码阶段传来的跳转地址
id_reg1	32	输入	译码阶段传来的操作数一
id_reg2	32	输入	译码阶段传来的操作数二
id_wd	5	输入	译码段传来的要写入的寄存器地址
id_wreg	1	输入	是否要写入寄存器
next_inst_in_delayslot_i	1	输入	下一个指令是否再延迟槽
rst	1	输入	复位信号
stall	6	输入	流水线暂停信号
ex_aluop	8	输出	执行阶段要执行的运算的子类型
ex_alusel	3	输出	执行阶段要执行的运算的类型
ex_is_in_delayslot	1	输出	执行阶段判断是否是延迟槽指令
ex_link_address	32	输出	执行阶段传入的跳转地址
ex_reg1	32	输出	执行阶段的操作数 1
ex_reg2	32	输出	执行阶段的操作数 2
ex_wd	5	输出	执行阶段要写入的寄存器地址

ex_wreg	1	输出	执行阶段是否要写入寄存器
is_in_delayslot	1	输出	是否是延迟指令

表 5 id_ex 模块接口

（3）执行访存阶段

EX/MEM 模块作用是将执行阶段取得的运算结果，在下一个时钟传递到流水线访存阶段。

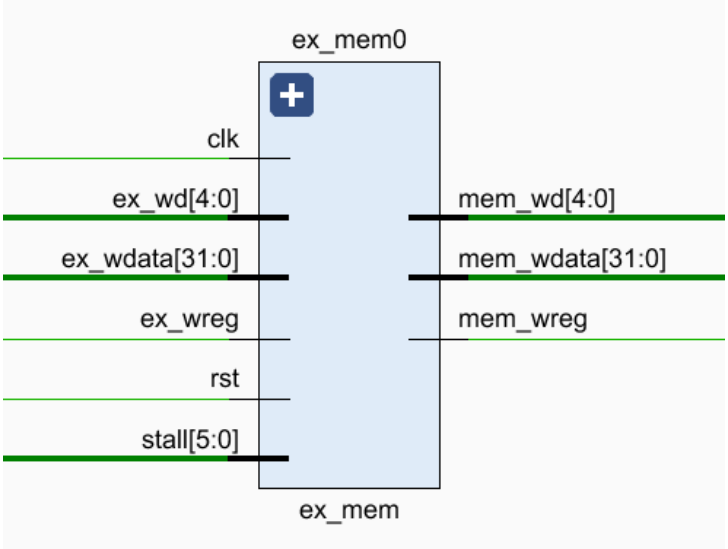


图 11 ex_mem 模块图

接口名	宽度(bit)	输入/输出	作 用
clk	1	输入	时钟信号
ex_wd	5	输入	执行阶段传来的需要写入的寄存器地址
ex_wdata	32	输入	执行阶段传来的需要写入的值
ex_wreg	1	输入	执行阶段传来的判断是否需要写入寄存器
rst	1	输入	复位信号
stall	6	输入	流水线暂停信号
mem_wd	5	输出	访存阶段需要写入寄存器的地址
mem_wdata	32	输出	访存阶段需要写入寄存器的值
mem_wreg	1	输出	是否要写入寄存器

表 6 ex_mem 模块接口

（4）访存写回阶段

MEM/WB 模块的作用是将访存阶段的运算结果，在下一个时钟传递到回写阶段。

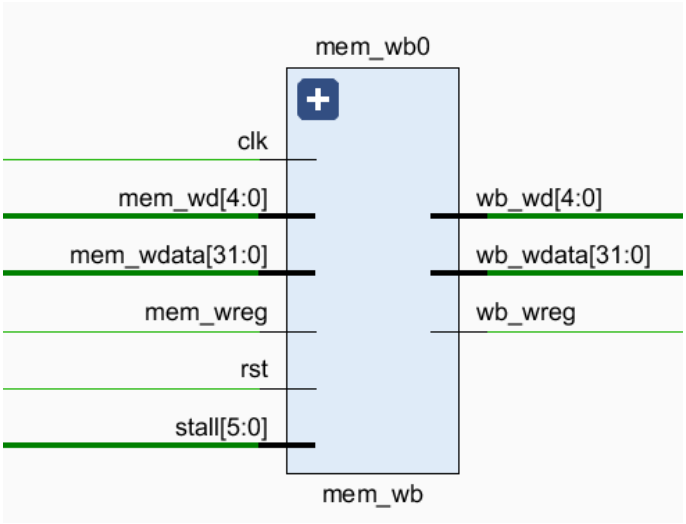


图 12 mem_wb 模块

接口名	宽度(bit)	输入/输出	作 用
clk	1	输入	时钟信号
rst	1	输入	复位信号
mem_wd	5	输入	访存阶段需要写入寄存器的地址
mem_wdata	32	输入	访存阶段需要写入寄存器的值
mem_wreg	1	输入	是否要写入寄存器
stall	6	输入	流水线暂停信号
wb_wd	5	输出	写回阶段需要写入的寄存器地址
wb_wdata	32	输出	写回阶段需要写回的值
wb_wreg	1	输出	写回阶段是否需要写入寄存器

表 7 mem_wb 模块接口

2.2.3 其他相关模块

（1）流水线暂停模块

假如位于流水线第 n 阶段的指令需要多个时钟周期，进而请求流水线暂停，那么需保持取指令地址 PC 的值不变，同时保持流水线第 n 阶段、第 n 阶段之前的各个阶段的寄存器不变，而第 n 阶段后面的指令继续运行，因此需要设计流水线暂停模块。

CTRL 模块的输入来自 ID、EX 模块的请求暂停信号 stallreq，对于本次课程设计设计的 cpu 而言，只有译码、执行阶段可能会有暂停请求，取指、访存阶段都没

有暂停请求，因为指令读取、数据存储器的读/写操作都可以在一个时钟周期内完成。

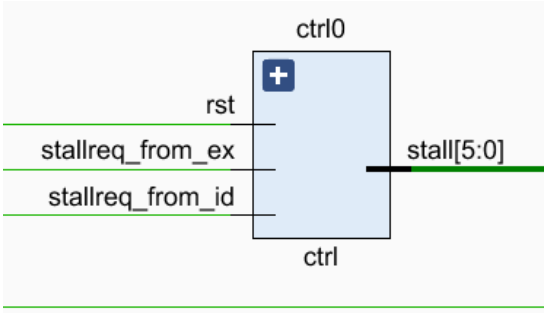


图 13 流水线暂停模块

接口名	宽度(bit)	输入/输出	作 用
rst	1	输入	复位信号
stallreq_from_ex	1	输入	执行阶段传来的请求流水线暂停信号
stallreq_from_id	1	输入	译码阶段传来的请求流水线暂停信号
stall	6	输出	输出流水线暂停信号

表 8 流水线暂停模块

（2）数据存储器

数据存储器的构成，使用了 vivado 当中的 IP 核构成，将数据都存储在.coe 文件当中，这样是因为自己动手写的 rom 指令存储器或者数据存储器实际可用的会非常的小，所以最好调用板子上已有的存储器，这样可以更好的加载操作系统或者其他大的程序，容量也很大。

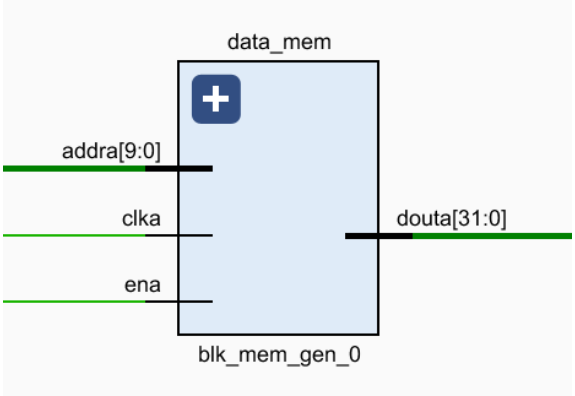


图 14 数据存储器模块

接口名	宽度(bit)	输入/输出	作 用
addra	10	输入	访存的值的数据存储器的值
clka	1	输入	时钟信号
ena	1	输入	使能信号
douta	32	输出	输出的对应地址中的数据

表 9 数据存储器模块

（3）指令存储器

指令存储器利用 verilog 语言编程实现，容量大小为 128KB。

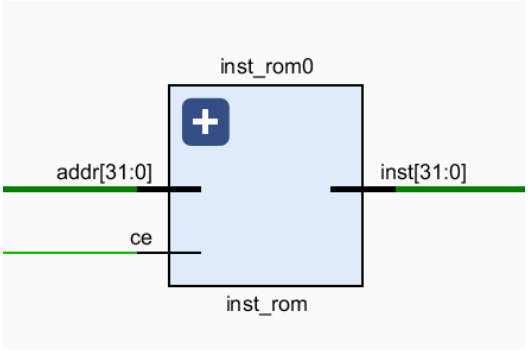


图 15 指令存储器模块

接口名	宽度(bit)	输入/输出	作 用
addr	32	输入	取指令的地址
ce	1	输入	使能信号
inst	32	输出	取出的指令

表 10 指令存储器模块

2.2.4 顶层封装模块

顶层封装模块主要是将上面所有实现的流水线各个阶段的模块进行例化、连接，将所有部件连接成一个整体，构成一个五级的流水线 cpu，从而使得结构更加完整，构成一个完整的能够准确执行一些指令的 cpu，以上所有模块的具体源码实现可以见附件，报告中将不再粘贴源码。

利用 vivado 所生成的数据通路图如下：

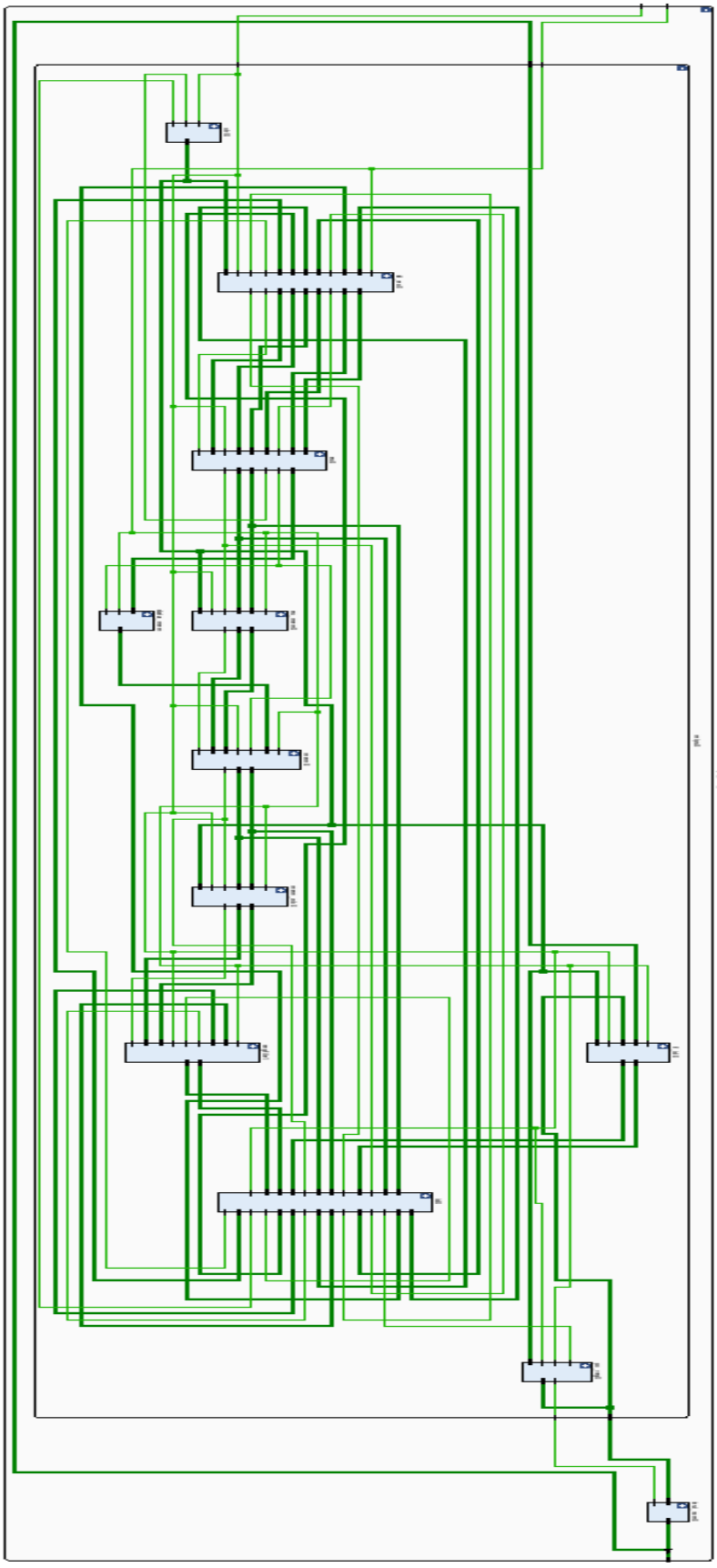


图 16 数据通路图（vivado）

3 支持的 21 条指令

本次 CPU 的设计实现了一些基本指令的实现，可以完成所有基本的算术运算和逻辑运算，由于时间原因，有一些运算类型类似但有略微差别的指令我没有再实现，例如 addu 和 add、subu 和 sub、sltu 和 slt 等等。

具体实现的指令有：

①R 型指令： OR AND XOR NOR SLL SLR SRA ADD SUB SLT MUL DIV JR

注：这里的乘法指令 MUL 和除法指令 DIV 我在实现的过程中，由于利用二进制乘除法来进行实现有一定的难度，需要考虑的条件有很多，所以我都进行了一个简化，直接利用 verilog 语言里面的* / 表达式实现的，但这样的 cpu 在实际过程中是不建议的因为直接利用 verilog 乘除来实现会带来很大的开销。

②I 型指令：ORI ANDI XORI ADDI LW

③J 型指令：J

下面将分别对这三种类型的指令进行具体分析。

3.1 R 型指令

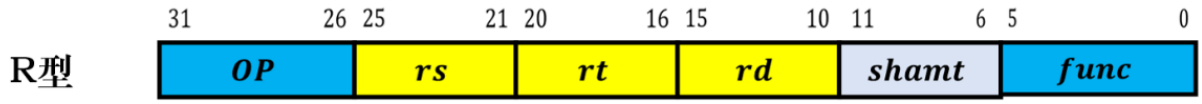


图 17 R 型指令机器码格式

①OR 指令

指令名	OP	Rs	Rt	Rd	shamt	func
OR	6op000001	5Rs	5Rt	5Rd	00000	Fun000010

作用：Rd<-Rs|Rt

将地址为 rs 的通用寄存器的值与地址为 rt 的通用寄存器的值进行逻辑“或”运算，运算结果保存到地址为 rd 的通用寄存器中。

②AND 指令

指令名	OP	Rs	Rt	Rd	shamt	func
AND	6op000001	5Rs	5Rt	5Rd	00000	Fun000001

作用：Rd<- Rs&Rt

将地址为 rs 的通用寄存器值与地址为 rt 的通用寄存器的值进行逻辑“与”运算，将结果送回 rd 寄存器。

③XOR 指令

指令名	OP	Rs	Rt	Rd	shamt	func
XOR	6op000001	5Rs	5Rt	5Rd	00000	Fun000011

作用：Rd<- Rs⊕Rt

将地址为 rs 的通用寄存器值与地址为 rt 的通用寄存器的值进行逻辑“异或”运算，将结果送回 rd 寄存器。

④NOR 指令

指令名	OP	Rs	Rt	Rd	shamt	func
NOR	6op000001	5Rs	5Rt	5Rd	00000	Fun000100

作用：Rd<- Rs⊙Rt

将地址为 rs 的通用寄存器值与地址为 rt 的通用寄存器的值进行逻辑“同或”运算，将结果送回 rd 寄存器。

⑤SLL

指令名	OP	Rs	Rt	Rd	shamt	func
SLL	6op000001	00000	5Rt	5Rd	sa	Fun000101

作用：Rd<- Rt<<sa

将 rt 寄存器当中的值逻辑左移 sa 位后，将结果送回 rd 寄存器。

⑥SLR

指令名	OP	Rs	Rt	Rd	shamt	func
SRL	6op000001	00000	5Rt	5Rd	sa	Fun000110

作用：Rd<- Rt>>sa

将 rt 寄存器当中的值逻辑右移 sa 位后，将结果送回 rd 寄存器。

⑦SRA

指令名	OP	Rs	Rt	Rd	shamt	func
SRA	6op000001	00000	5Rt	5Rd	sa	Fun000111

作用：Rd<- Rt>>sa（算术右移）

将 rt 寄存器当中的值算术右移 sa 位后，将结果送回 rd 寄存器。

⑧ADD

指令名	OP	Rs	Rt	Rd	shamt	func
ADD	6op000001	5Rs	5Rt	5Rd	00000	Fun001000

作用：Rd<- Rs+Rt

将地址为 rs 的通用寄存器值与地址为 rt 的通用寄存器的值进行“加”运算，将结果送回 rd 寄存器，不对是否溢出进行判断。

⑨SUB

指令名	OP	Rs	Rt	Rd	shamt	func
SUB	6op000001	5Rs	5Rt	5Rd	00000	Fun001001

作用：Rd<- Rs+Rt

将地址为 rs 的通用寄存器值与地址为 rt 的通用寄存器的值进行“减”运算，将结果送回 rd 寄存器，不对是否溢出进行判断。

⑩SLT

指令名	OP	Rs	Rt	Rd	shamt	func
SLT	6op000001	5Rs	5Rt	5Rd	00000	Fun001010

作用：Rd <- (Rs < Rt)

将地址为 rs 的通用寄存器的值与地址为 rt 的通用寄存器的值按照有符号数进行比较，如果前者小于后者，那么将 1 保存到地址为 rd 的通用寄存器中;反之，将 0 保存到地址为 rd 的通用寄存器中。

⑪MUL

指令名	OP	Rs	Rt	Rd	shamt	func
MUL	6op011100	5Rs	5Rt	5Rd	00000	Fun001011

作用：Rd<- Rs*Rt

将地址为 rs 的通用寄存器值与地址为 rt 的通用寄存器的值进行“乘法”运算，将结果送回 rd 寄存器，只保留低 32 位，高 32 位会丢失。（若想实现数据保留需要设计 HI、LO 寄存器，暂未实现。）

⑫DIV

指令名	OP	Rs	Rt	Rd	shamt	func
DIV	6op011100	5Rs	5Rt	5Rd	00000	Fun001100

作用：Rd<- Rs/Rt

将地址为 rs 的通用寄存器值与地址为 rt 的通用寄存器的值进行“除法”运算，将结果送回 rd 寄存器，只能实现整除，小数后部分会丢失。（商和余数都保留需

要借助 HI、LO 寄存器，暂未实现。）

⑬JR 指令

指令名	OP	Rs	Rt	Rd	shamt	func
JR	6op000001	5Rs	00000	00000	00000	Fun001101

作用：pc<-rs

将地址为 rs 的通用寄存器的值赋给寄存器 PC，作为新的指令地址。

3.2 I 型指令

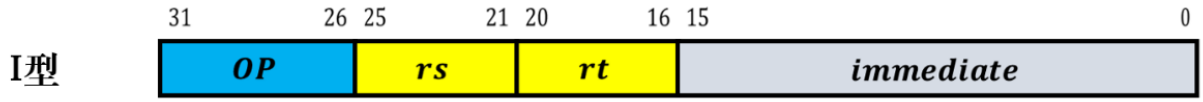


图 18 I 型指令机器码格式

⑭ORI 指令

指令名	OP	Rs	Rt	imm
ORI	6op001101	5Rs	5Rt	imm

作用：Rt<- Rs|imm

将地址为 rs 的通用寄存器值与立即数 imm 进行逻辑“或”运算，将结果送回 rt 寄存器。

⑮ANDI 指令

指令名	OP	Rs	Rt	imm
ANDI	6op001100	5Rs	5Rt	imm

作用：Rt<- Rs&imm

将地址为 rs 的通用寄存器值与立即数 imm 进行逻辑“与”运算，将结果送回 rt 寄存器。

⑯XORI 指令

指令名	OP	Rs	Rt	imm
XORI	6op001110	5Rs	5Rt	**XORI**

作用：Rt<- Rs⊕imm

将地址为 rs 的通用寄存器值与立即数 imm 进行逻辑“异或”运算，将结果送回 rt 寄存器。

⑰ADDI 指令

指令名	OP	Rs	Rt	imm
ADDI	6op001000	5Rs	5Rt	imm

作用：Rt<- Rs+imm

将地址为 rs 的通用寄存器值与立即数 imm 进行“加法”运算，将结果送回 rt 寄存器。

⑱BEQ 指令

指令名	OP	Rs	Rt	imm
BEQ	6op000100	5Rs	5Rt	offset

作用：如果 rs == rt 则跳转 offset，否则不跳转。

⑲B 指令

指令名	OP	Rs	Rt	imm
B	6op000100	00000	00000	offset

作用：直接跳转到 offset 偏移地址。

⑳LW 指令

指令名	OP	Rs	Rt	imm
lw	6op100011	5base	5Rt	offset

作用：从内存中指定的加载地址处，读取一个字，保存到地址为 rt 的通用寄存器中。该指令有地址对齐要求，要求加载地址的最低两位为 00。

3.3 J 型指令

图 19 J 型指令机器码格式

㉑J 指令

指令名	OP	address
J	6op000010	26target

作用： pc <- (pc+4)[31,28] || target || ‘00’

转移到新的指令地址，其中新指令地址的低 28 位是指令中的 target 左移两位的值，新指令地址的高 4 位是跳转指令后面延迟槽指令的地址高 4 位。

4 流水线相关的处理

流水线中的相关分为以下三种类型。

（1）结构相关:指的是在指令执行的过程中，由于硬件资源满足不了指令执行的要求，发生硬件资源冲突而产生的相关，在本次课程设计中暂不考虑。

（2）数据相关:指的是在流水线中执行的几条指令中，一条指令依赖于前面指令的执行结果。

（3）控制相关:指的是流水线中的分支指令或者其他需要改写 PC 的指令造成的相关。

4.1 数据相关

流水线的数据相关又分为三种情况：RAW、WAR、WAW，根据 mips 五级流水线的特点并不存在 WAR 和 WAW 相关，所以需要解决的只有 RAW（写后读）相关，解决这个数据相关主要运用的方法就是**数据前推**。

具体做法就是，保存 ex、mem、wb 阶段的结果传到 id 当中去，如果译码阶段使用到了，就直接利用，这样就解决了其对应**相邻指令、相隔一条指令、相隔两条指令**的数据相关，核心代码如下：

```
1.      end else if ((reg2_read_o == 1'b1)&&(ex_wreg_i == 1'b1)//相邻
2.                      &&(ex_wd_i == reg2_addr_o)) begin
3.          reg2_o<=ex_wdata_i;
4.      end else if((reg2_read_o == 1'b1)&&(mem_wreg_i == 1'b1)//隔 1 条
5.                      &&(mem_wd_i == reg2_addr_o)) begin
6.          reg2_o<=mem_wdata_i;
7.      end else if(reg2_read_o==1'b1)begin//间隔两条
8.          reg2_o <= reg2_data_i;// Regfile 读端口 2 的输出值
```

4.2 控制相关

控制相关是指流水线中的转移指令或者其他需要改写 PC 的指令造成的相关。这些指令改写了 PC 的值，所以导致后面已经进入流水线的几条指令无效。

解决控制相关的一个方法就是设置延迟槽，所谓延迟槽就是指跳转指令后面的那一条指令，延迟槽指令需要放入和该跳转指令不相关的指令，以至于不影响整个 cpu 的运行，而那一条指令的放入**不是由 cpu 决定的，是由编译器决定的**，所以这里我只是简单的实现了延迟槽模块，在 id 阶段就进行判断是否要进行跳转，

只需要添加几个接口判断该指令是否是延迟槽指令即可。

```
            if(branch_flag_i == `Branch) begin
                pc <= branch_target_address_i;
            end
            else begin
                pc <= pc+4;
            end
```

5 实验结果分析

在完成所有的模块设计并完成封装之后，接下来对已经编写的 cpu 来进行一个测试，测试 cpu 的各项性能是否符合要求，主要将其分为两部分，一个是对 cpu 的仿真分析，编写 testbench 对 cpu 进行一个仿真结果分析，根据仿真波形来判断结果是否准确，确保每条指令的可行性，另一个是对 cpu 的上板调试，将 cpu 烧写在 fpga 开发版上，利用开发板上的一些基本的按键、led 灯等接口将 cpu 的运行过程进行一个可视化，从而达到测试 cpu 的一个目的。

5.1 仿真结果分析

对 cpu 进行一个结果仿真，设置一个 20ns 的时钟，即每 10ns 时钟翻转一次，在 200ns 的时刻复位信号 rst 取消使能，然后将 rst 和 clk 信号传入 cpu 当中，cpu 即可运行起来，编写仿真代码 testbench 如下：

```
`timescale 1ns / 1ps
`include "defines.v"
module textbench(
        output wire[15:0] led
        );

    reg     CLOCK_50;
    reg     rst;
    initial begin
    CLOCK_50 = 1'b0;
    forever #10 CLOCK_50 = ~CLOCK_50;
    end

  initial begin
     rst = `RstEnable;
     #200 rst= `RstDisable;
     #1000 $stop;
```

```
    end

    mips_and_rom mips_and_rom0(
    .clk(CLOCK_50),
    .rst(rst),
          .led(led)
   );
endmodule
```

5.1.1 指令分析

根据上面编写的机器指令格式编写一串测试上述大部分指令的代码，然后放在指令存储器当中，对其进行一个仿真，其中进行测试的机器代码如下图所示：

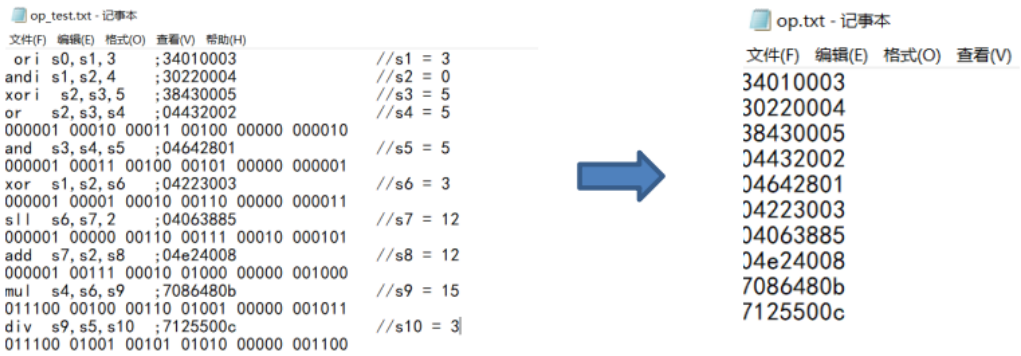


图 20 用来测试的机器代码

上面测试的机器代码存储在 op.txt 文件当中，在 cpu 运行过程中指令存储器 inst_rom 会读取文本文件当中的值，然后一个一个的送入到 pc 当中进行测试。

```
        initial $readmemh("E:\\MIPS_CPU\\op.txt ",inst_mem);
```

总共测试了十条指令的结果，其中相似的指令没有再重复测试，测试的理论结果的值已经注释到文件的旁边，仿真波形测试结果如下：

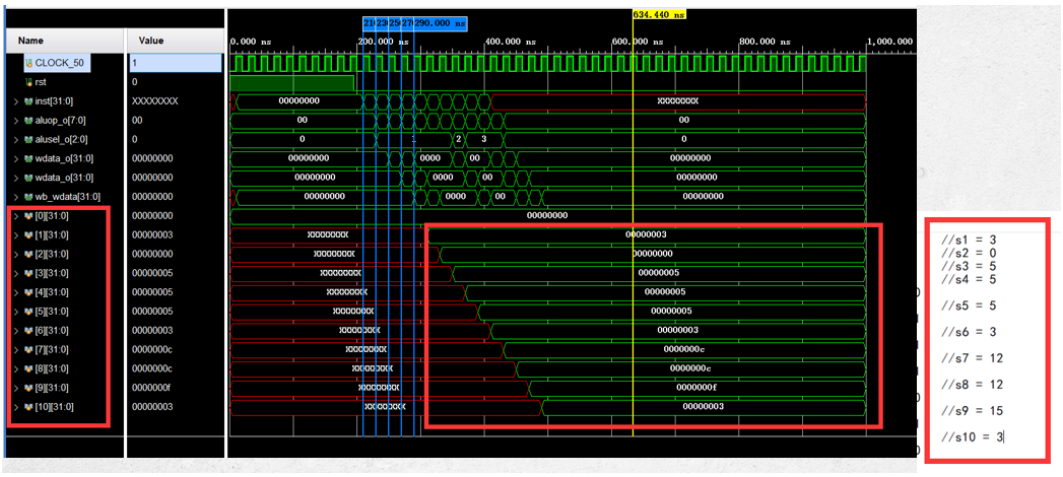


图 21 指令测试结果

从仿真波形中，我们将前十个寄存器的值在仿真波形当中显示出来，分别代表了寄存器$0-$10 的值，然后根据仿真结果与实现写好的寄存器当中的值进行一一比对，发现结果是正确的，说明 cpu 指令运行过程中是没问题的。

5.1.2 流水线分析

在运行测试指令的过程中，将 cpu 运行过程中五个流水线阶段的中间变量都取出来，在仿真波形当中展现出来，然后在仿真波形图当中进行一个标记，最后得到的效果如下：

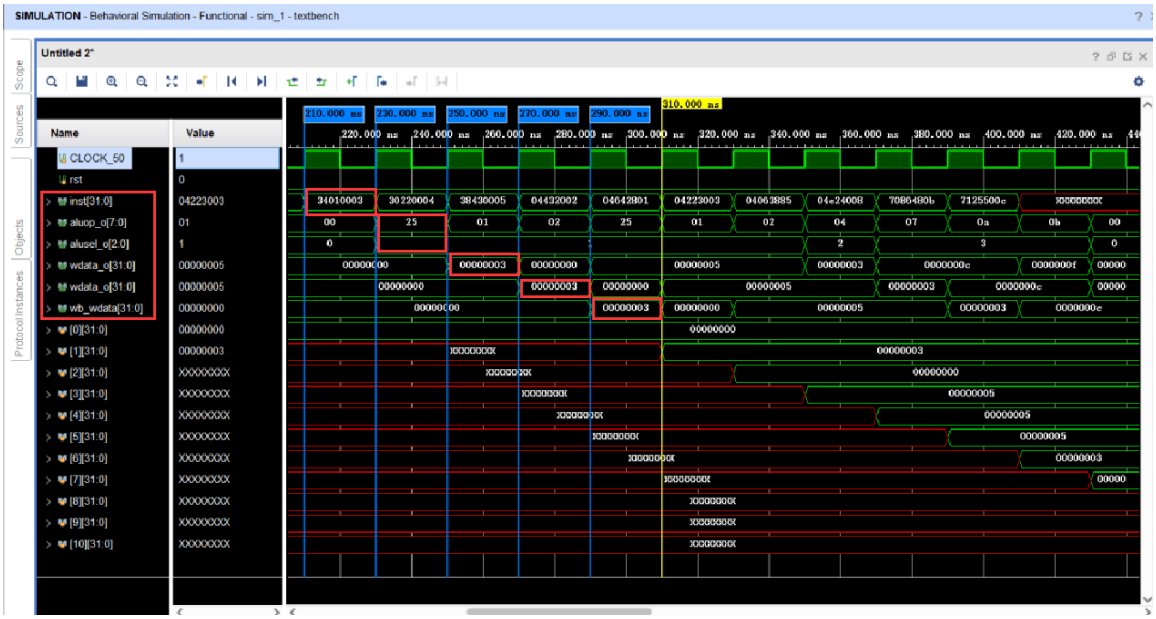


图 22 流水线效果图

根据测试结果可以很明显的看出 cpu 的流水线运行的过程，测试结果中我分别取的是取指阶段的机器指令的值 inst、译码阶段的 alu_op 运算子类型和 alu_sel 运算类型的值、执行阶段需要写入源操作数的值 wdata_o、访存阶段需要写入源操作

数的值 wdata_o 和写回阶段需要写回的值 wb_data。

5.2 上板调试

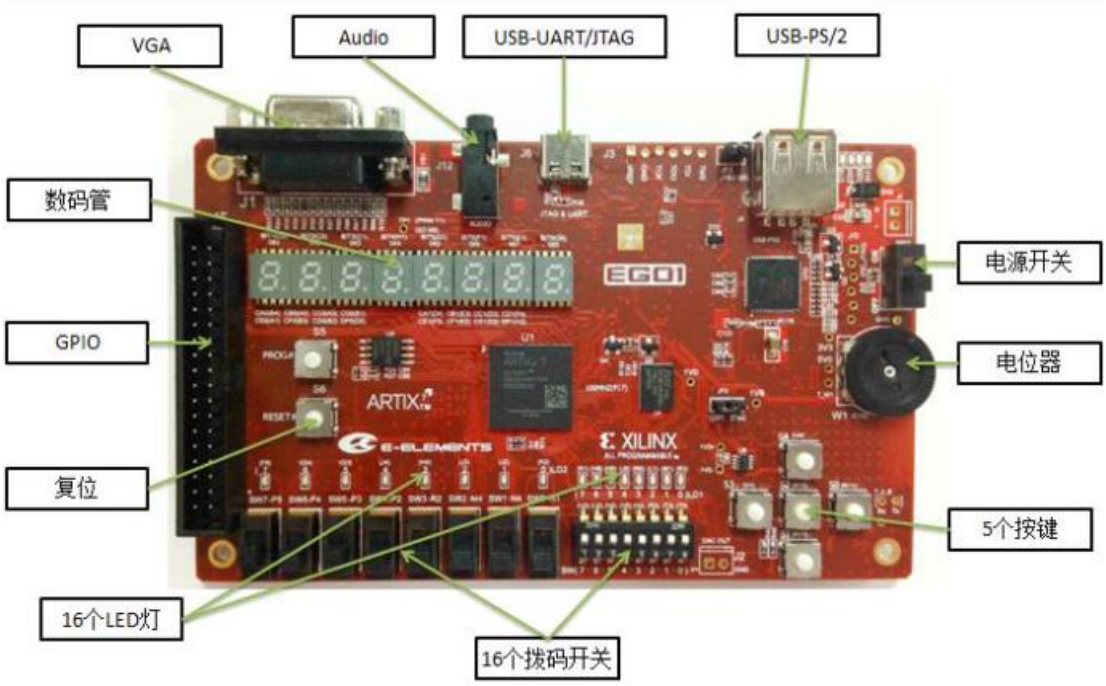


图 23 fpga 开发板示意图

实验使用的开发板主要是基于 Xilinx Artix-7 FPGA 研发的 EGO1 配备的 FPGA (XC7A35T-1CSG324C)，通过 vivado 连接 fpga 开发板进行上板调试的过程老师在课程群里面已经给出教程，这里我就不再赘述，也可以参考 csdn 一个博主写的类似板子的调试步骤，简单易懂，在此感谢该博主让我快速上手调试过程，链接如下：

https://blog.csdn.net/weixin_43529932/article/details/106983712

本次实验主要利用开发板进行了如下两个测试：

①verilog 编写慢时钟，将 cpu 取指令的过程通过 16 个 led 灯在开发板上显示出来，代表了指令的高十六位，将 cpu 取指令的过程动态的展示了出来。

②编写机器代码，计算函数 f(x,y) = x^2 + y + 2 的值，然后烧写到通过 16 个 led 灯展现出来，利用拨码实现简单的交互式运行。

5.2.1 指令的动态显示

首先，将预先写好的所有指令的机器码放到 inst_rom.txt 文件当中，供指令存储器读取，文件内容如下：

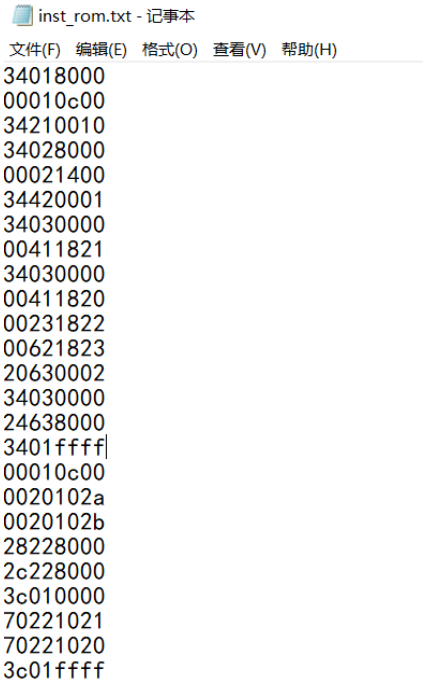
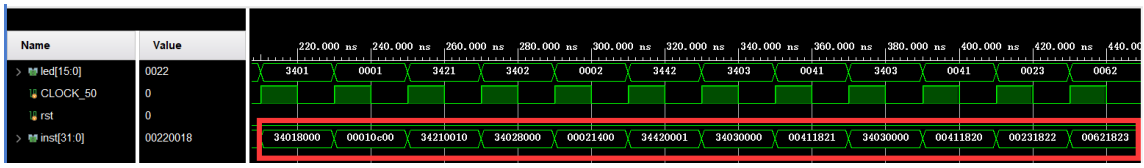


图 24 inst_rom.txt 文件内容

先对其结果进行一个简单的仿真，观察其仿真波形，看是否 cpu 能够顺利将预先写好的指令读入 pc 寄存器当中，仿真结果如下：



根据仿真结果可以看出，cpu 读出的值与 txt 文件当中的值是一样的，仿真波形没有什么问题。

接下来就是对慢时钟的 verilog 编写，由于板子上的时钟是 100MHz，直接利用的话 cpu 运行速度是非常快的，无法观察其结果，所以需要编写一个 1s 的慢时钟。

刚开始我编写慢时钟时，直接在 textbench 中写了一个一秒的时钟传入 cpu 当中，但是上板调试的过程中发现，不论怎样，在 fpga 开发板上始终只能显示第一条指令的结果在 led 灯上，这个问题我想了很长很长的时间都没有发现问题所在，最后我通过学习老师在课设群里面发的 EGO1 实验当中，发现了“流水灯实现”这一个实现，里面用到了慢时钟实现流水灯，我才找到原因，原来是先要设置一个计数器进行计数，到一定数量之后再对 clk 进行一个翻转，最终将结果传入 cpu 当中，才能够实现，这个问题花了我一天的时间去解决，过程可以说是十分煎熬了。具体代码如下：

conter.v

```
module counter(
 input clk,
   input rst,
   output clk_bps
   );
    reg [13:0]cnt_first,cnt_second;
    always @( posedge clk or posedge rst )
      if( rst )
      cnt_first <= 14'd0;
     else if( cnt_first == 14'd10000 )
      cnt_first <= 14'd0;
     else
      cnt_first <= cnt_first + 1'b1;
    always @( posedge clk or posedge rst )
     if( rst )
      cnt_second <= 14'd0;
     else if( cnt_second == 14'd10000 )
      cnt_second <= 14'd0;
     else if( cnt_first == 14'd10000 )
      cnt_second <= cnt_second + 1'b1;
    assign clk_bps = cnt_second == 14'd10000 ? 1'b1 : 1'b0;
endmodule
```

testbench.v

```
`include "defines.v"
module textbench(
        input clk,
    input rst_n,
        output wire[15:0] led
        );
       wire clk_bps;
   wire rst;
   assign rst = ~rst_n;
  counter counter(
    .clk( clk ),
    .rst( rst ),
    .clk_bps( clk_bps )
   );
  mips_and_rom mips_and_rom0(
  .clk(clk_bps),
  .rst(rst) ,
    .led(led)
 );
endmodule
```

配置完成的接口图如下：

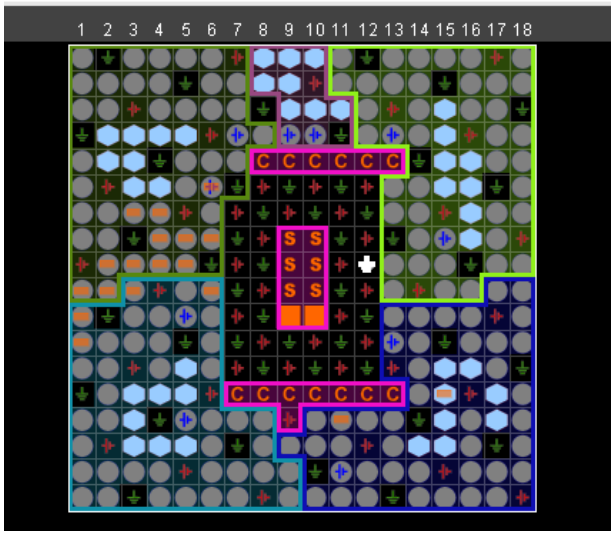


图 25 接口图

实验结果如下：

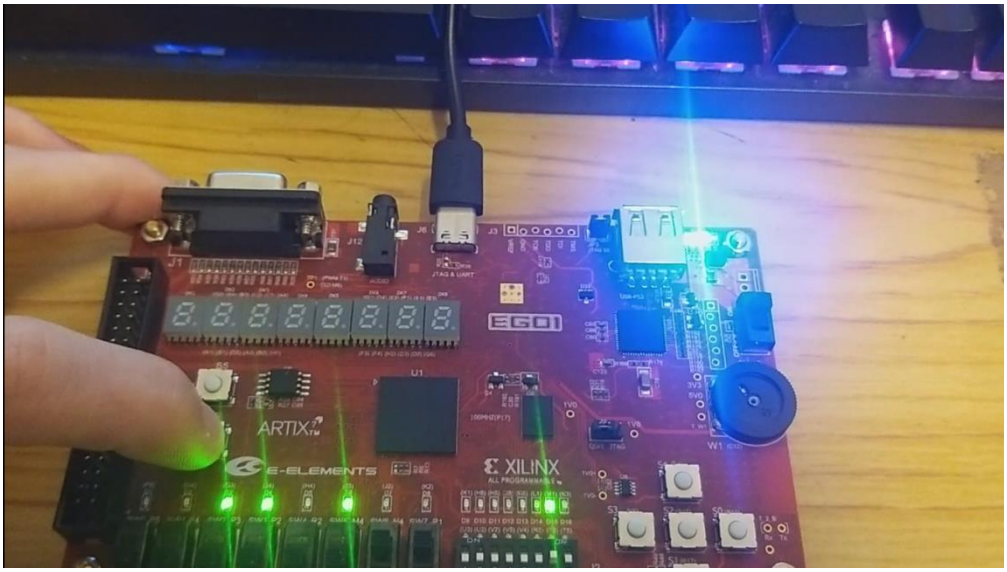


图 26 效果演示图 1

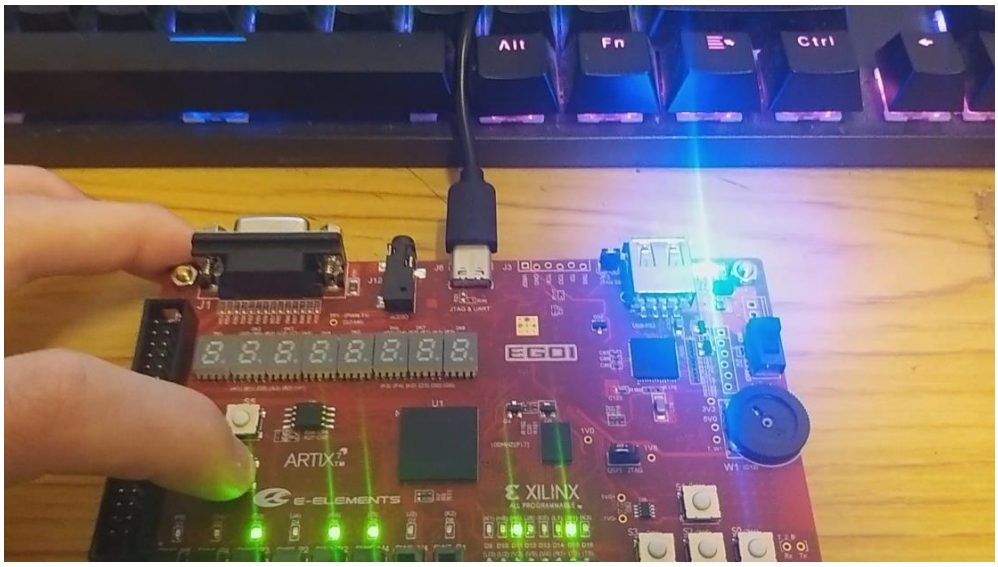


图 27 效果演示图 2

测试效果已经在验收的过程中为老师演示过了，这里就不再展示了，只放两张具有代表性的图片，分别代表了不同时钟周期指令的高 16 位的值在 16 个 led 上显示。

5.2.2 函数值的计算

为了使实现的 cpu 能够在板子上更好的展现出来，利用第二种调试方案，将 RST 按钮复用为时钟信号，也就是给 CLK 以 RST 的默认引脚 P15，这样就能实现按一次 FPGA 板上的 RST 按钮，就运行一个时钟周期，但是需要注意的一点是，利用按键来实现时钟周期时，一定要进行**按键的防抖动处理，不然按下一次按钮，可能走的是两个或者多个周期，而不是一个周期。**

老样子，首先我们在板子上调试之前先进行一个仿真测试，可以有效的避免一些问题，毕竟烧写在板子上测试一次所需要的时间还是挺长的，测试的机器指令代码如下：

```
34010003 //ori s1,s0,0x0003给寄存器s1赋值x=3

34020004  //ori s2,s0,0x0004 给寄存器s2赋值y=4

7021180b  //011100 00001 00001 00011 00000 001011
        //mul s1,s1,s3 求x的平方放入s3中

04431808  //000001 00010 00011 00011 00000 001000
        //add s2,s3,s3 再加上y放入s3中

20630002 //001000 00011 00011 0000000000000010
        //addi s3,s3,2 最后加上立即数2


完成了函数F(x,y) = x^2+y+2 的编写，在开发板上实现
```

a.txt - 记事本

文件(F) 编辑(E) 格式(O)

```
34010003
34020004
7021180b
04431808
20630002
```

图 28 测试的机器代码

这里我们先将寄存器 1 的值赋值为 3，寄存器 2 的值赋值为 4，可以知道计算函数值的结果为 15，存入寄存器 3 当中，仿真结果如下：

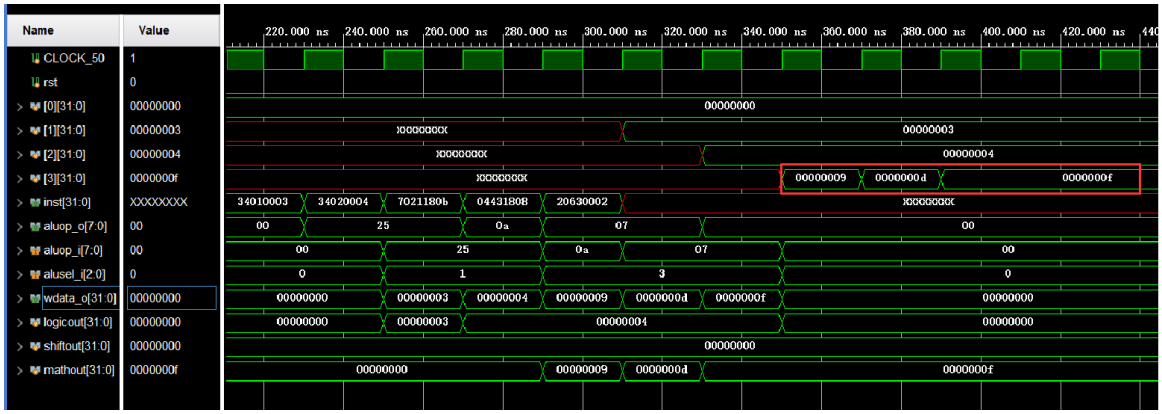


图 29 仿真结果图

由仿真结果可以看出，寄存器 3 当中的值为 15，与实现计算的结果一致，没有问题，接下来就进行上板调试。

引脚接入的过程很简单，只需要将防抖动处理后的 clk 信号接到 P15 上面，从而达到按一次走一个时钟周期的效果。

需要注意的是，上板调试时，将机器指令的前两条赋值指令删去，然后将 R1 和 R2 两个寄存器分别接到开发板的前四个和后四个拨码上面，从而实现可以交互式的改变寄存器的值，将 R3 的值利用 led 灯显示出来，按下 rst 按钮之后，便可实现函数值的重新计算，结果已成功向老师演示，效果图如下：

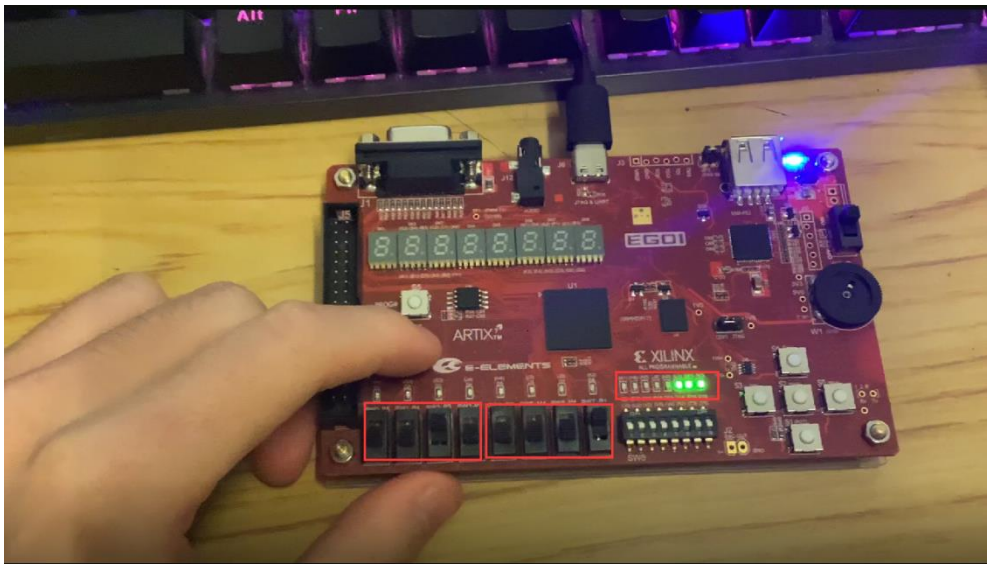


图 30 效果演示图 1

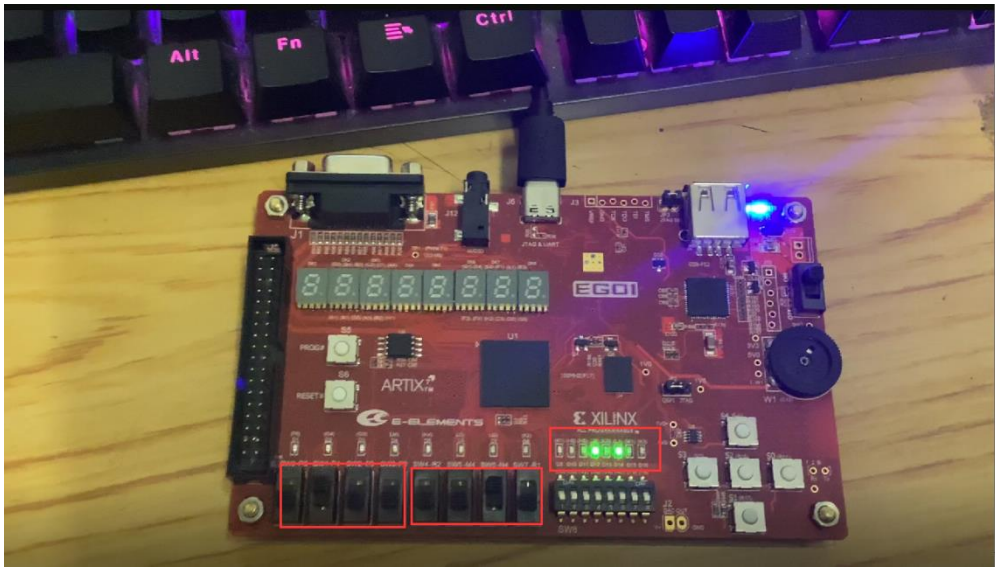


图 31 效果演示图 2

6 总结

6.1 一些小问题

①在初始化指令存储器时，使用了 initial 过程语句，initial 过程语句只执行一次，通常用于仿真模块中对激励向量的描述，或用于给变量赋初值，是面向模拟仿真的过程语句，通常不能被综合工具支持，所以如果要将本章实现的 OpenMIPS 处理器使用综合工具进行综合,那么需要修改这里初始化指令存储器的方法。

②细心真的非常的重要，由于我在写寄存器堆的时候，else if 条件少加了一个零号寄存器恒置为零的一个操作，导致仿真的过程中一直都在出一些问题，光 debug 的时间就花了我好几个小时，所以用 verilog 语言写 cpu 的时候一定要小心谨慎。

③目前的 PC 的取值只有三种情况：

情况一:PC 等于 PC+4。这属于一般情况，每个时钟周期 PC 加 4，指向下一条指令。

情况二:PC 保持不变。当流水线暂停的时候，就会发生这种情况。

情况三:PC 等于转移判断的结果。如果是转移指令，且满足转移条件，那么会将转移目标地址赋给 PC。

④设计跳转指令的时候需要注意，因为是**按字节寻址 pc+4**，pc 指令地址后面还需要补上两个 00。

6.2 心得体会

这次课程设计由于特殊情况，是在家中完成的，而且是一个人完成，对我来说是一次很大的考验，我刚开始没什么思路，所以就一直在网上找一些相关的教程，找一些相关的教学视濒进行学习，我从最基本的开始学习，先学习 verilog 语言的一些基本语法，虽然在上学期学习过，但是已经有些遗忘，毕竟直接用 verilog 语言编写出来一个 cpu 对我来说还是很有难度的，然后就是 cpu 的整个设计逻辑，搞清楚了之后才开始动手。

这里需要再次感谢的是《自己动手写 cpu》这本书以及 B 站重庆大学的 cpu 设计教程视频

https://www.bilibili.com/video/BV1pK4y1C7es/?spm_id_from=333.337.search-card.all.click

对我帮助很大，一开始动手时毫无头绪，但是跟着这本书的第四章以及教学视频，一步步的搭建框架，虽然只是一个框架，而且只能实现一条 ORI 指令，但是框架搭建起来以后整个逻辑就清晰很多了，我不断的往里面加指令，增加功能，让我的 cpu 更加的完善，最终独自完成了整个相对完整的一个 cpu 的设计。

同时，这次综合设计让我更加深刻的理解的 cpu 的工作过程，CPU 的本质就是一个只会“解释指令”，并且死板地执行指令的莫得感情的机器，有一个比喻说得好：硬件是计算机的躯体，OS 才是灵魂。它的主要结构包括译码器、ALU 以及各种寄存器。我们会给具体指令集中的指令规定固定的格式，以便让译码器理解代码，其实原理很简单，CPU 译码的过程相当于就是查字典，在规定的有限的指令集内找到与输入的机器码相匹配的情况，并执行相应的操作。

最后在本次系统硬件综合设计中，我充分将所学到的理论知识与实践相结合起来，真正的掌握了一个 CPU 设计的全部模块与步骤，对之前学过的很多知识有了更具象，更深刻的理解了，更具体的感悟都穿插在实验报告的每个部分里面，与实验内容是相结合的。

报告的最后感谢一下本次课程设计的老师们，验收过程中有不会的问题老师也认真为我讲解原理，让我对 cpu 有更深层次的理解。

参考文献

[1] 雷思磊.自己动手写 CPU. 北京: 电子工业出版社，2014.