



**2021~2022学年 第一学期**

**《系 统 硬 件 综 合 设 计》**

设 计 报 告

班 级\_\_\_\_\_\_\_\_物联网工程19-2班\_\_\_\_\_\_\_

学 号 \_\_\_\_\_\_\_\_2019217819\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

姓 名\_\_\_\_\_\_\_\_\_\_\_付炎平\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

成 绩 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

2022年 1月

**目录**

[1. 设计目的和要求 3](#_Toc7776)

[2. 设计内容 3](#_Toc17618)

[2.1 MIPS32架构的指令说明及本文设计 3](#_Toc21787)

[2.2 本文设计 4](#_Toc29203)

[2.3 MIPS32架构寻址 7](#_Toc26525)

[3. 设计原理 7](#_Toc16346)

[4. 具体实现 8](#_Toc29187)

[4.1整体架构设计 8](#_Toc14999)

[4.2.1 Instructions Memory 9](#_Toc21661)

[4.2.2 Regfile 10](#_Toc14456)

[4.2.3 ALU 10](#_Toc3989)

[4.2.4 Data Memory 11](#_Toc30949)

[4.2.5 control 12](#_Toc19697)

[4.3 数据流介绍展示 13](#_Toc19717)

[4.3.1 状态转移 13](#_Toc5009)

[4.3.2 控制信号的产生 14](#_Toc31046)

[4.3.3 ALU运算 16](#_Toc13069)

[5. 调试及结果展示 17](#_Toc16439)

[5.1 打开工程 17](#_Toc14246)

[5.2 查看命令 17](#_Toc18753)

[5.3 使用Vivado仿真 18](#_Toc28330)

[5.4 生产约束文件 19](#_Toc27125)

[5.5 生成bitstream 20](#_Toc1777)

[5.6 开启并连接开发板 21](#_Toc20241)

[5.7 结果显示 21](#_Toc30312)

[6. 总结 23](#_Toc10858)

**说明**

此次课设我们组仅有我一人，课设从设计、编写代码实现到后期调试、下载到开发版、报告撰写全部由我一人独立完成，共实现16条指令：add、sub、addi、 ori、sll、slt、mov、lw、sw、beq、j、jr、or、and、jal、halt。

**多周期MIPS架构CPU的设计与实现**

# 1. 设计目的和要求

通过设计并实现支持16条指令的多周期MIPS32架构的CPU，进一步理解和掌握多周期CPU设计的基本原理和步骤。

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：完成多周期CPU设计，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. 设计内容

## 2.1 MIPS32架构的指令说明及本文设计

**R类型**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| op | rs | rt | rd | sa | func |

**I类型**

|  |  |  |  |
| --- | --- | --- | --- |
| 31 26 | 25 21 | 20 16 | 15 0 |
| op | rs | rt | immediate |

**J类型**

|  |  |  |
| --- | --- | --- |
| 31 26 | 25 21 | 20 0 |
| op | rs | address |

MIPS32架构中的所有指令都是32位，也就是32个0、1编码连在一起表示一条指令，有三种指令格式。如图所示：其中op是指令码、func是功能码。

(1)**R类型**：具体操作由op、func结合指定，rs和rt是源寄存器的编号，rd是目的寄存器的编号，比如假设目的寄存器是S3,那么对应的rd就是00011(此处是二进制数。MIPS32架构中有32个通用寄存器，使用5位编码就可以全部表示，所以rs、rt、rd的宽度都是5位。sa只有在移位指令中使用，用来指定移位位数。

(2)**I类型：**具体操作由op指定，指令的低16位是立即数，运算时要将其扩展至32位，然后作为其中一个源操作数参与运算。

(3)**J类型：**具体操作由op指定，一般是跳转指令，低26位是字地址，用于产生跳转的目标地址。

## 2.2 本文设计

（1）**R类型**：

① add：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| 000000 | rs | rt | rd | 00000 | 100001 |

进行的操作：R[rd]←R[rs]+R[rt]; PC←PC+4

② sub：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| 000001 | rs | rt | rd | 00000 | 100011 |

进行的操作：R[rd]←R[rs]-R[rt]; PC←PC+4

③ or：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| 010000 | rs | rt | rd | 00000 | 100011 |

进行的操作：R[rd]←R[rs]|R[rt]; PC←PC+4

④ and：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| 010001 | rs | rt | rd | 00000 | 100011 |

进行的操作：R[rd]←R[rs]&R[rt]; PC←PC+4

⑤ sll：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| 011000 | rs | 00000 | rd | sa | 100011 |

进行的操作：R[rd]←R[rs]<<(zero-extend)sa; PC←PC+4

⑥ slt：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| 100111 | rs | rt | rd | 00000 | 100011 |

进行的操作：if R[rs]<R[rt] R[rd]=1; else R[rd]=0 PC←PC+4

⑦ mov：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| 100111 | rs | rt | rd | 00000 | 100011 |

进行的操作：R[rd]<－R[rs] + $0 PC←PC+4

（2）**I类型**

① addi：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 21 | | | 20 16 | | 15 0 |
| 000010 | | rs | rt | | imm16 | |

进行的操作：R[rt]←R[rs]+zero\_ext(imm16); PC←PC+4

② lw：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 21 | | | 20 16 | | 15 0 |
| 110001 | | rs | rt | | imm16 | |

进行的操作：R[rt]←Mem(R[rs]+sign\_ext(imm16)); PC←PC+4

③ sw：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 21 | | | 20 16 | | 15 0 |
| 110000 | | rs | rt | | imm16 | |

进行的操作：Mem(R[rs]+sign\_ext(imm16))←R[rt]; PC←PC+4

④ beq：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 21 | | | 20 16 | | 15 0 |
| 110100 | | rs | rt | | imm16 | |

进行的操作：if(R[rs] == R[rt]) then PC←PC+4+[sign\_ext(imm16)]

else PC←PC+4

⑤ lui：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 21 | | | 20 16 | | 15 0 |
| 001111 | | rs | rt | | imm16 | |

进行的操作：R[rt][31:16]←imm16 , R[rt][15:0]←0; PC←PC+4

⑥ ori：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 21 | | | 20 16 | | 15 0 |
| 010010 | | rs | rt | | imm16 | |

进行的操作：R[rt]←R[rs]|zero\_ext(imm16); R[rt][15:0]←0;

PC←PC+4

（3）**J类型**

① j addr：

|  |  |  |
| --- | --- | --- |
| 31 26 | 25 0 | |
| 111000 | | imm25 |

进行的操作：PC←{PC[31:28], imm25, 2’b0}

② jal addr：

|  |  |  |
| --- | --- | --- |
| 31 26 | 25 0 | |
| 111010 | | imm25 |

进行的操作：调用子程序，pc←{(pc+4)[31..28],addr[27..2],0,0;; $31<-pc+4

③ j r：

|  |  |  |
| --- | --- | --- |
| 31 26 | 25 0 | |
| 111001 | | rs(5位) 未用 未用 保留 |

进行的操作：PC←rs，跳转。

③ halt(停机指令)：

|  |  |  |
| --- | --- | --- |
| 31 26 | 25 0 | |
| 111111 | | 26位0 |

PC不变

## **2.3 MIPS32架构寻址**

MIPS32架构的寻址模式有四种：寄存器寻址、立即数寻址、寄存器相对寻址和PC相对寻址。在我们的指令中，R类型的add、sub、or、and、sll等指令采用寄存器寻址；I类型的ori、lw、sw、lui等指令采用寄存器相对寻址，beq指令除此之外还采用了PC相对寻址；J、Jr等指令采用了PC相对寻址。

# 3. 设计原理

多周期CPU指的是将整个CPU的执行过程分成几个阶段，每个阶段用一个时钟去完成，然后开始下一条指令的执行，而每种指令执行时所用的时钟数不尽相同，这就是所谓的多周期CPU。

CPU在处理指令时，一般需要经过以下几个阶段：

**取指令(IF)**：根据程序计数器pc中的指令地址，从存储器中取出一条指令，同时，pc根据指令字长度自动递增产生下一条指令所需要的指令地址，但遇到“地址转移”指令时，则控制器把“转移地址”送入pc，当然得到的“地址”需要做些变换才送入pc。

**指令译码(ID)**：对取指令操作中得到的指令进行分析并译码，确定这条指令需要完成的操作，从而产生相应的操作控制信号，用于驱动执行状态中的各种操作。

**指令执行(EXE)：**根据指令译码得到的操作控制信号，具体地执行指令动作，然后转移到结果写回状态。

**存储器访问(MEM)：**所有需要访问存储器的操作都将在这个步骤中执行，该步骤给出存储器的数据地址，把数据写入到存储器中数据地址所指定的存储单元或者从存储器中得到数据地址单元中的数据。

**结果写回(WB)：**指令执行的结果或者访问存储器中得到的数据写回相应的目的寄存器中。

设计中就按照这五个阶段进行设计，这样一条指令的执行最长需要五个(小)时钟周期才能完成，但具体情况要根据该条指令的情况而定，有些指令不需要五个时钟周期的，这就是多周期的CPU。

# 4. 具体实现

## 4.1整体架构设计

（1）设计架构如图所示：

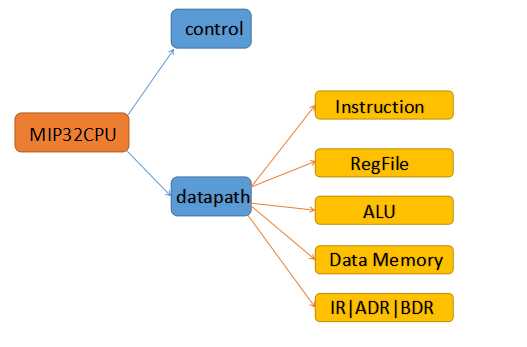


图4-1设计架构

整体分为控制模块与数据流模块两部分；控制模块为数据流模块提供控制信号，同时控制状态转移；数据流模块分为指令模块、寄存器模块、运算器模块、存储器模块、结果保存模块五大部分，分别实现了不同功能。

（2）整体架构

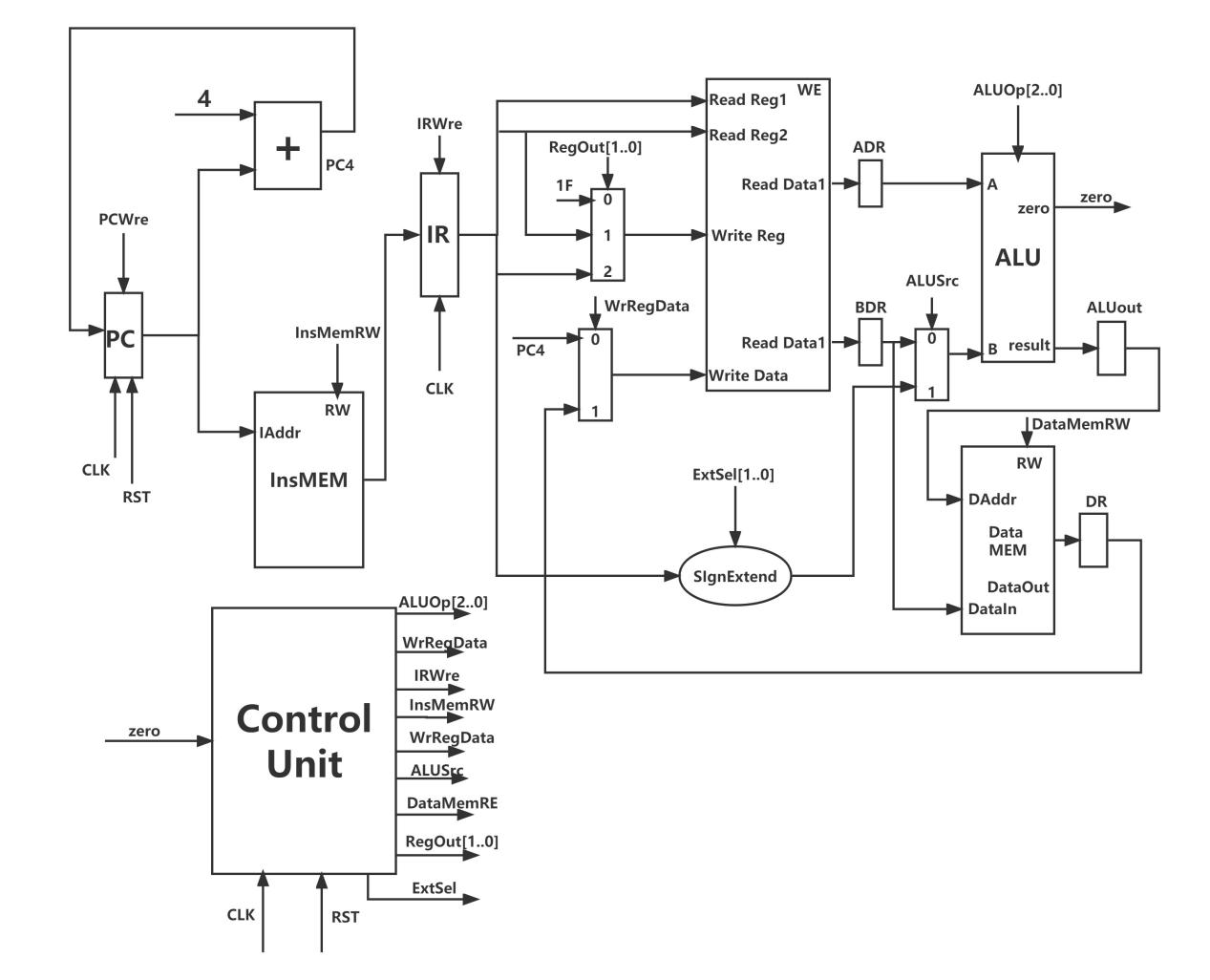


图4-2整体架构

这是多周期cpu各个模块的整体架构图，其中ControlUnit为控制器模块，主要用于产生各种控制信号来使其他模块能够按照多周期CPU的设计逻辑有序进行，并且在各个模块的后面，都有相应数据寄存器来暂时存储该模块现在的状态，从而为下一个状态转换更好的做出准备。

**4.2 分模块设计**

### 4.2.1 Instructions Memory

|  |  |  |  |
| --- | --- | --- | --- |
| **接口名** | **宽度(bit)** | **输入\输出** | **作用** |
| Iaddr | 32 | 输入 | 指令地址输入端口 |
| DataIn | 32 | 输入 | 存储器数据输入端口 |
| Instructions | 32 | 输出 | 存储器数据输出端口 |
| RW | 1 | 输入 | 指令存储器读写控制信号 |

表4-1 Instructions模块的接口描述

Iaddr是指令地址的输入端口，指令存储器根据输入PC的值取出相应的32位的指令，并输出，同时吧PC加4。

### 4.2.2 Regfile

|  |  |  |  |
| --- | --- | --- | --- |
| **接口名** | **宽度(bit)** | **输入\输出** | **作用** |
| RegWr | 1 | 输入 | 写使能信号 |
| Read Reg1 | 5 | 输入 | 读入的第一个寄存器编号 |
| Read Reg2 | 5 | 输入 | 读入的第二个寄存器编号 |
| RW | 5 | 输入 | 写入的寄存器编号 |
| Write Data | 32 | 输入 | 写入的数据 |
| Read Data1 | 32 | 输出 | 从第一个寄存器中读出的数据 |
| Read Data2 | 32 | 输出 | 从第二个寄存器中读出的数据 |

表4-2 Regfile模块的接口描述

Regfile模块是寄存器相关模块，实现寄存器的读写功能。因为共有32个寄存器，需要5位来对寄存器编号进行编码，故有关编号的输入都是5位的。其中，Read Reg1、Read Reg2寄存器的信息分别由Read Data1、Read Data2端口输出，因有些指令会利用到立即数，故Read Data2端口与立即数通过控制信号进行二选一的选择。RW、RegWr、Write Data是与写有关的端口，RegWr控制是否进行写操作；RegWr端口的输入是一个二选一选择器的输出，该二选一选择器是通过控制信号判断rt为需要输入的寄存器编号还是rs为需要输入的寄存器编号。

### 4.2.3 ALU

|  |  |  |  |
| --- | --- | --- | --- |
| **接口名** | **宽度(bit)** | **输入\输出** | **作用** |
| ALUOp | 2 | 输入 | 控制ALU中进行的操作 |
| A | 32 | 输入 | 操作数1 |
| B | 32 | 输入 | 操作数2 |
| result | 32 | 输出 | 运算结果 |
| zero | 32 | 输出 | 运算结果 |

表4-3 ALU模块的接口描述

ALU模块是运算相关的模块，由ALUOp控制信号控制两个操作数进行的数据操作，result与zero均为答案输出，只是继续传递的方向不同，zero是向Instruction模块传递，主要是为在beq指令时根据运算结果控制PC的变化情况，ALU功能运算表如下：

|  |  |  |
| --- | --- | --- |
| ALUOp[2..0] | 功能 | 描述 |
| 000 | Y=A+B | 加 |
| 001 | Y=A-B | 减 |
| 010 | Y=(A-B)?1:0 | 比较A与B不带符号 |
| 011 | Y=(((ra<rb)&&(ra[31]==rb[31]))||((ra[31)==1&&rb[31]==0)))?1:0 | 比较A与B带符号 |
| 100 | Y=B<<A | B左移A位 |
| 101 | Y = A ∨ B | 或 |
| 110 | Y = A ∧ B | 与 |
| 111 | Y = A ⊕ B | 异或 |

### 4.2.4 Data Memory

|  |  |  |  |
| --- | --- | --- | --- |
| **接口名** | **宽度(bit)** | **输入\输出** | **作用** |
| WR | 1 | 输入 | 写使能信号 |
| Daddr | 32 | 输入 | 需要访问的地址 |
| Data In | 32 | 输入 | 需要写入的数据 |
| Data Out | 32 | 输出 | 读出的数据 |
| RD | 1 | 输入 | 读使能信号 |

表4-4 Data Memory模块的接口描述

Data Memory是与存储器相关的模块。在进行读操作时WR信号置0，Daddr端口输入需要访问的地址，由Data Out端口输出访问的数据；在进行写操作时WR信号置1，数据由Data In端口读入并写入到Daddr端口输入的地址中。

### 4.2.5 control

|  |  |  |
| --- | --- | --- |
| **控制信号** | 状态“0” | 状态“1” |
| RST | 对于 PC，初始化 PC 为程序首地址 | 对于 PC，PC 接收下一条指令地址 |
| PCWre | PC 不更改，相关指令：halt | PC 更改，相关指令：除指令 halt 外 |
| ALUSrc | 来自寄存器堆 data 输出，相关指令：add、sub、and、slt、sll | 来自 sign 或 zero 扩展的立即数，相关指令：add、ani、ori、beq lw、sw |
| IRWre | IR(指令寄存器)不更改 | IR 寄存器写使能 |
| ExtSel | (zero-extend)immediate，相关指令：andi、xori、ori | (sign-extend)immediate，相关指令： addiu、slti、lw、sw、beq |
| WrRegData | 写入寄存器组寄存器的数据来自pc+4(pc4），相关指令：jal | 写入寄存器组寄存器的数据来自 ALU 运算结果或存储器读出的数据 |
| InsMemRW | 写指令存储器 | 读指令存储器 |
| RegOut[1..0] | 写寄存器组寄存器的地址 |  |

表4-5 control模块控制信号与指令对应关系

control模块是整个CPU运行的核心，表格中列出了控制信号与相关指令的对应关系。CU通过改变控制信号来控制状态转移，每个状态代表一个时钟周期，同时需要D触发器保存当前状态。

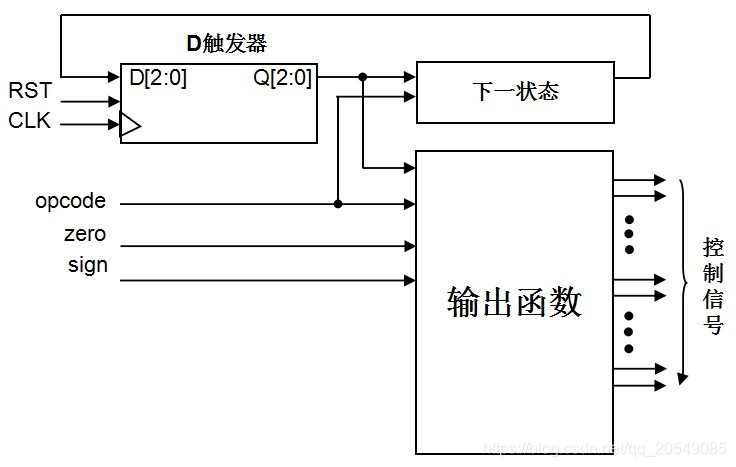


图4-2-1 D触发器电路图

三个 D 触发器用于保存当前状态，是时序逻辑电路，RST 用于初始化状态“000“，另外两个部分都是组合逻辑电路，一个用于产生下一个阶段的状态，另一个用于产生每个阶段的控制信号。从图上可看出，下个状态取决于指令操作码和当前状态；而每个阶段的控制信号取决于指令操作码、当前状态和反映运算结果的状态 zero 标志和符号 sign 标志。

## 4.3 数据流介绍展示

此处结合代码展示多周期具有代表性状态转移，以及ALU在进行运算时CU给各个控制信号的赋值，同时展示ALU运算时根据取出指令不同的操作码来进行不同的运算。

### 4.3.1 状态转移

1. 控制信号根据Opcode进行控制当前状态到下一个状态的转换，状态转换与Opcode[0..3]对应关系如图所示：

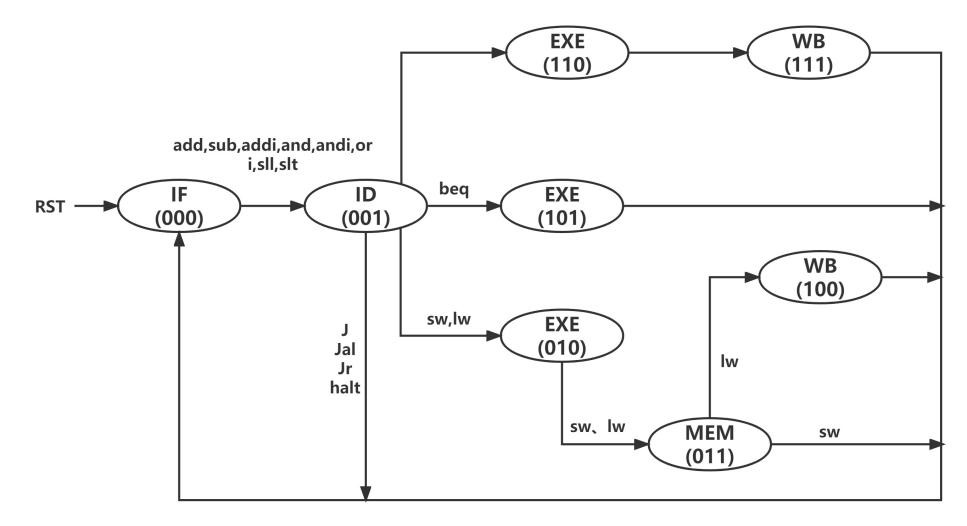


图4-3状态转移图

1. 状态的转换主要有当前状态和操作码opcode决定，根据上图的状态转换图，我们实现相应的代码:

*parameter [2:0] IF = 3'b000, // IF状态*

*ID = 3'b001, // ID状态*

*aEXE = 3'b110, // 第一条分支的EXE状态*

*bEXE = 3'b101, // 第二条分支的EXE状态*

*cEXE = 3'b010, // 第三条分支的EXE状态*

*MEM = 3'b011, // MEM状态*

*aWB = 3'b111, // 第一个分支的WB状态*

*cWB = 3'b100; // 第三个分支的WB状态*

*always @(i\_state or opcode) begin*

*case (i\_state)*

*IF: next\_state = ID;*

*ID: begin*

*case (opcode[5:3])*

*3'b110: begin*

*if (opcode == 6'b110100) next\_state = bEXE; // beq指令*

*else next\_state = cEXE; // sw, lw指令*

*end*

*3'b111: next\_state = IF;*

*// j, jal, jr, halt指令*

*default: next\_state = aEXE; // add, sub等指令*

*endcase*

*end*

*aEXE: next\_state = aWB;*

*bEXE: next\_state = IF;*

*cEXE: next\_state = MEM;*

*MEM: begin*

*if (opcode == 6'b110001) next\_state = cWB;*

*// lw指令*

*else next\_state = IF; // sw指令*

*end*

*aWB: next\_state = IF;*

*cWB: next\_state = IF;*

*default: next\_state = IF;*

*endcase*

3) 从代码中可以看出来，我们使用不同的二进制代码来代表不同的状态， 当当前状态为IF时,下一个状态一定是ID，当当前状态为ID时，下一 个状态为执行状态，但是执行状态有三种情况，分别为aEXE、bEXE、cEXE， 主要是对了三种不同类型的命令，分别为：① j、jal、jr、halt；② lw、 sw；③ sw、lw;在这三种不同的执行状态的下一个状态也不同，分别对 应aWB、cWB、IF、MEM，对于lw指令的下一个状态为cWB，其余指令的 下一个状态均为IF，从IF状态回到IF状态，相当于一个指令周期。

4) 状态的转移有的是无条件的，例如从 IF 状态转移到 ID 就是无条件 的；有些是有条件的，例如 EXE 状态之后不止一个状态，到底转向哪 个状态由该指令功能，即指令操作码决定。

### 4.3.2 控制信号的产生

1. CU根据当前的状态和取出指令的操作码来产生不同的控制信号，从而使各个模块之前能够协调一致的运行，并同时能够按照各个指令进行状态转换。
2. 根据情况对不同控制信号的分配的代码如下：

*always @(state) begin*

*// 对PCWre定值*

*if (state == IF && opcode != halt) PCWre = 1;*

*else PCWre = 0;*

*// 对InsMemRW定值*

*InsMemRW = 1;*

*// 对IRWre定值*

*if (state == IF) IRWre = 1;*

*else IRWre = 0;*

*// 对WrRegData定值*

*if (state == aWB || state == cWB) WrRegData = 1;*

*else WrRegData = 0;*

*// 对RegWre定值*

*if (state == aWB || state == cWB || opcode == jal) RegWre = 1;*

*else RegWre = 0;*

*// 对ALUSrcA定值*

*if (opcode == sll) ALUSrcA = 1;*

*else ALUSrcA = 0;*

*// 对ALUSrcB定值*

*if (opcode == addi || opcode == ori || opcode == slti|| opcode == sw || opcode == lw) ALUSrcB = 1;*

*else ALUSrcB = 0;*

*// 对DataMemRW定值*

*if (state == MEM && opcode == sw) DataMemRW = 1;*

*else DataMemRW = 0;*

*// 对 DBDataSrc定值*

*if (state == cWB) DBDataSrc = 1;*

*else DBDataSrc = 0;*

*// 对ExtSel定值*

*if (opcode == ori) ExtSel = 2'b01;*

*else if (opcode == sll) ExtSel = 2'b00;*

*else ExtSel = 2'b10;*

*// 对RegDst定值*

*if (opcode == jal) RegDst = 2'b00;*

*else if (opcode == addi || opcode == ori || opcode == lw) RegDst = 2'b01;*

*else RegDst = 2'b10;*

*// 对PCSrc定值*

*case(opcode)*

*j: PCSrc = 2'b11;*

*jal: PCSrc = 2'b11;*

*jr: PCSrc = 2'b10;*

*beq: begin*

*if (zero) PCSrc = 2'b01;*

*else PCSrc = 2'b00;*

*end*

*default: PCSrc = 2'b00;*

*endcase*

1. 可以从上述代码看出，控制信号主要根据当前状态的不同以及需要执行指令的不同而产生，在多周期CPU中，一个时钟周期只能执行一个状态，而每个状态仅仅是完成某个指令其中的一部分，当下一个时钟周期到来时，我们要相应的切换下一个状态，这需要根据当前状态和指令的操作码来决定。

### 4.3.3 ALU运算

1. CPU在执行指令时，往往要进行一些算术逻辑运算，而进行运算的主要场所是在我们的算数逻辑单元ALU中，在我们的指令中有加减法运算指令：add、sub、addi、subi，逻辑运算指令：and、or、ori，移位运算指令：sll，slt；不同的运算是根据控制单元CU产生的ALUOp控制信号决定的。
2. CU产生的ALUOp控制信号代码如下所示：

*case(opcode)*

*sub: ALUOp = 3'b011;*

*Or: ALUOp = 3'b101;*

*And: ALUOp = 3'b110;*

*ori: ALUOp = 3'b101;*

*slt: ALUOp = 3'b010;*

*slti: ALUOp = 3'b111;*

*sll: ALUOp = 3'b100;*

*beq: ALUOp = 3'b001;*

*default: ALUOp = 3'b000;*

*endcase*

1. ALU根据不同的ALUOp进行不同的运算，代码如下所示：

*always @(A or B or ALUOp) begin*

*case(ALUOp)*

*3'b000: result = A + B;*

*3'b001: result = A - B;*

*3'b010: begin*

*if (A < B &&(( A[31] == 0 && B[31]==0) || (A[31] == 1 && B[31]==1))) result = 1;*

*else if (A[31] == 0 && B[31]==1) result = 0;*

*else if (A[31] == 1 && B[31]==0) result = 1;*

*else result = 0;*

*end*

*3'b011: result = (A < B ? 1 : 0);*

*3'b100: result = B << A;*

*3'b101: result = A | B;*

*3'b110: result = A & B;*

*3'b111: result = (~A & B) | (A & ~B);*

*default: result = 0;*

*endcase*

1. 从上述代码可以看出来两个操作数在进行不同的运算时主要根据ALUOp来决定，并且最后都是把运算结果保存到result中传送到寄存器或者存储器中。

# 调试及结果展示

## **打开工程**

打开vivado并打开工程并查看开发板型号，我们的开发板型号是xc7a35t-csg324。

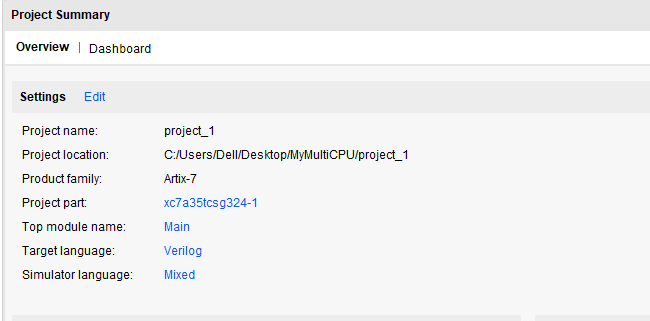


图5-1 Vivado工程图

## **查看命令**

查看Instructions.txt中的指令，其中一条指令为32位，而存储器中存储单元是一个字节8位，所以一条指令占4个存储单元，下图所示存储的依次指令为：add、sub、beg、ori、or。

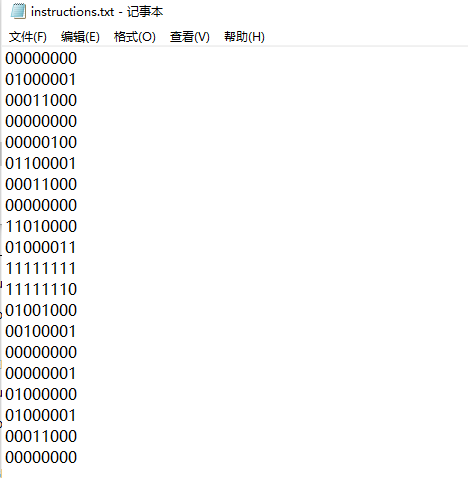


图5-2 指令二进制码

## **使用Vivado仿真**

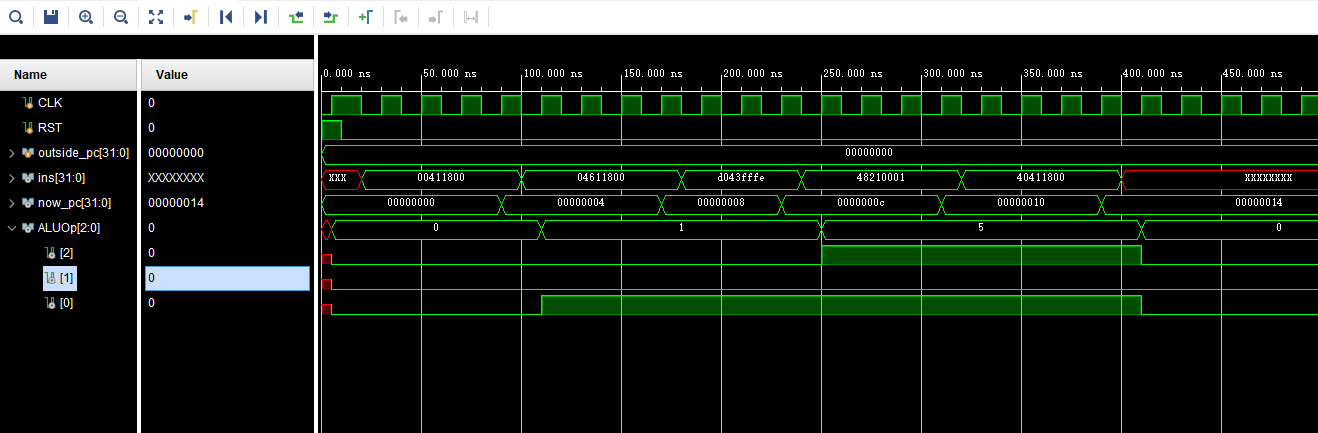
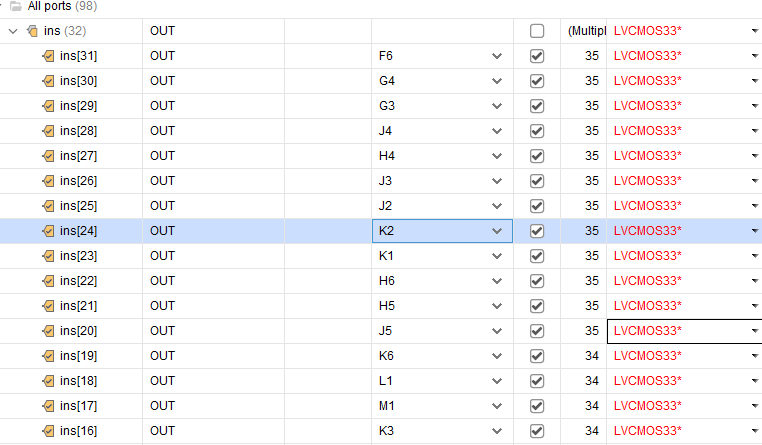


图5-3 Vivado仿真图

其中now\_pc为当前PC的值，ins为取出的指令16进制的值。从仿真图中可以看出，每过若干个时钟周期，PC会加4，为取出一条指令，而取出指令所需要的时钟周期的不同主要是因为不同指令的状态个数不同，同时也很好的验证了上述所说的执行指令的顺序，其中ALUOP的值依次为：000，001，001，101，101，对应指令：add、sub、beg、ori、or。

## **生产约束文件**

这里我利用了Vivado提供的绑定功能，将Ego板子上的16位LED灯和代码中取出的32位寄存器变量ins高16位绑定，并把时钟CLK和开关R1绑定，RST和开关N4绑定。



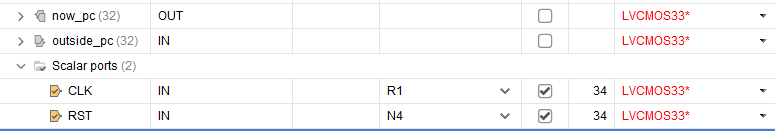


图5-4-1 端口变量绑定图

生成的约束文件如下：

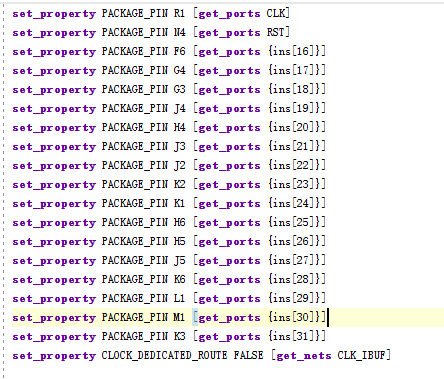


图5-4-2 约束文件图

## **生成bitstream**

在这里需要配置一个bitstream的文件，然后生成bitstream。

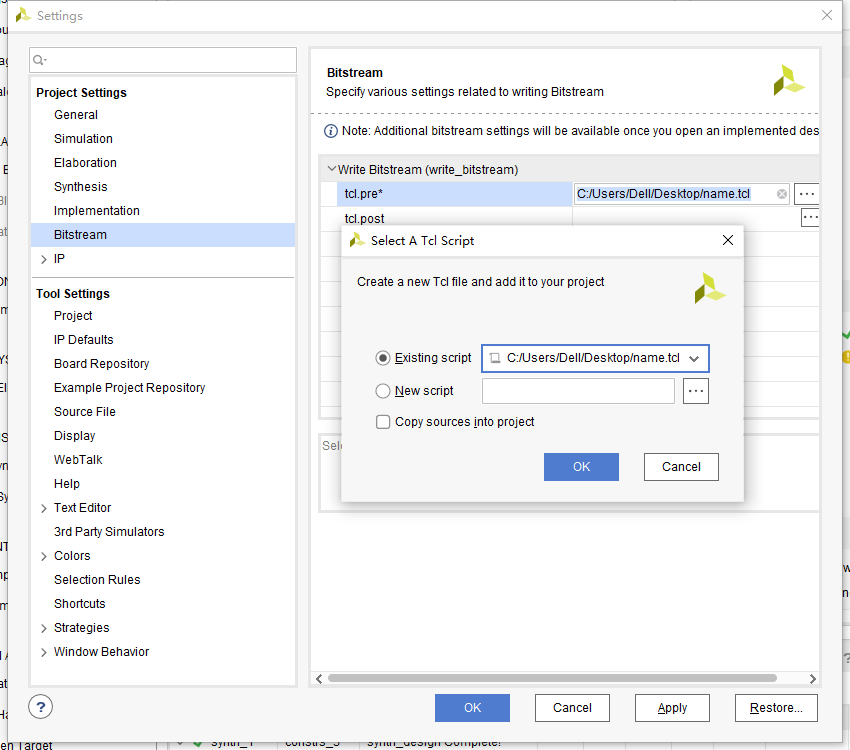


图5-5 bitstream配置图

## **开启并连接开发板**

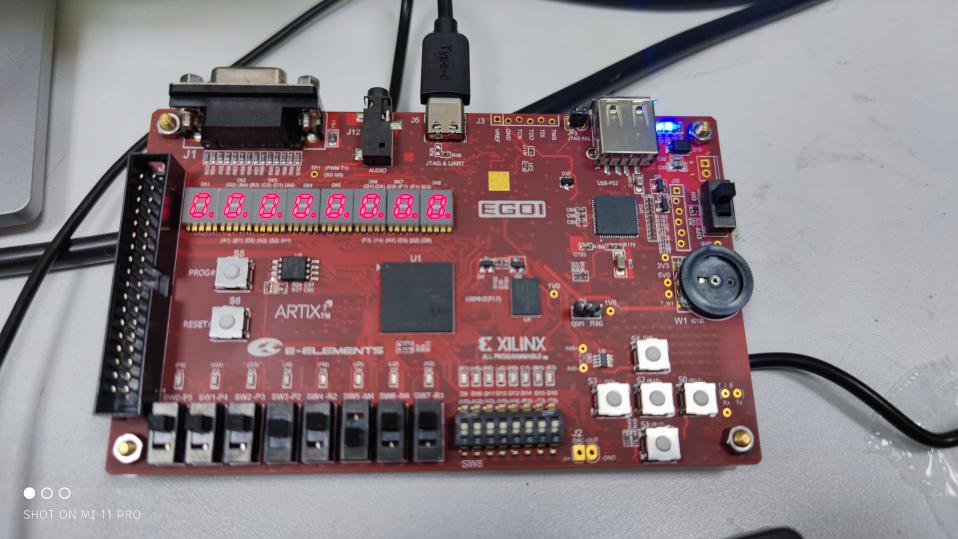


图5-6 FPGA开发板

## **结果显示**

连接成功后，将生成的bitstream烧录在板子上并启动开发板，通过来回波动R1模拟时钟周期的变化，来看LED改变的效果，其中16个LED灯D1至D6代表取出指令的16至31位，取出的前四条指令FPGA图片如图所示：

① add 指令高16位：0000000001000001；

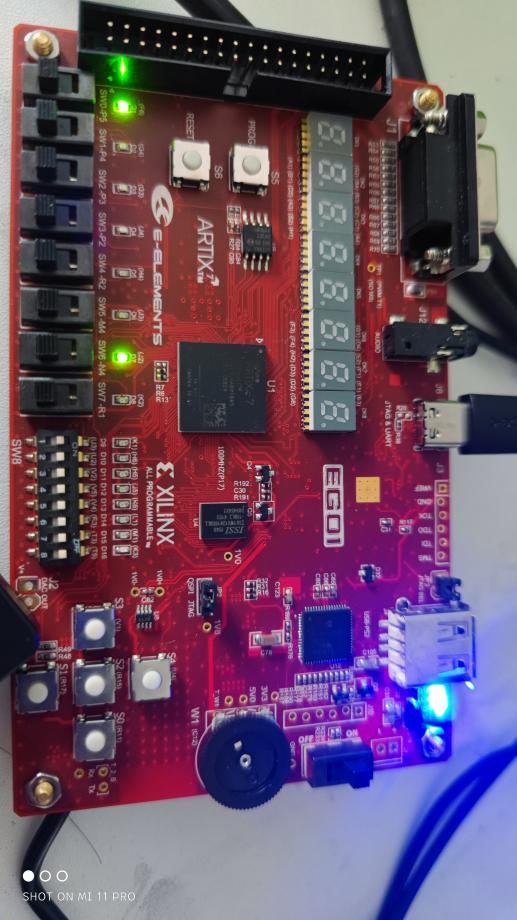


图5-7-1 add指令图片

② sub指令高16位：0000010001100001；

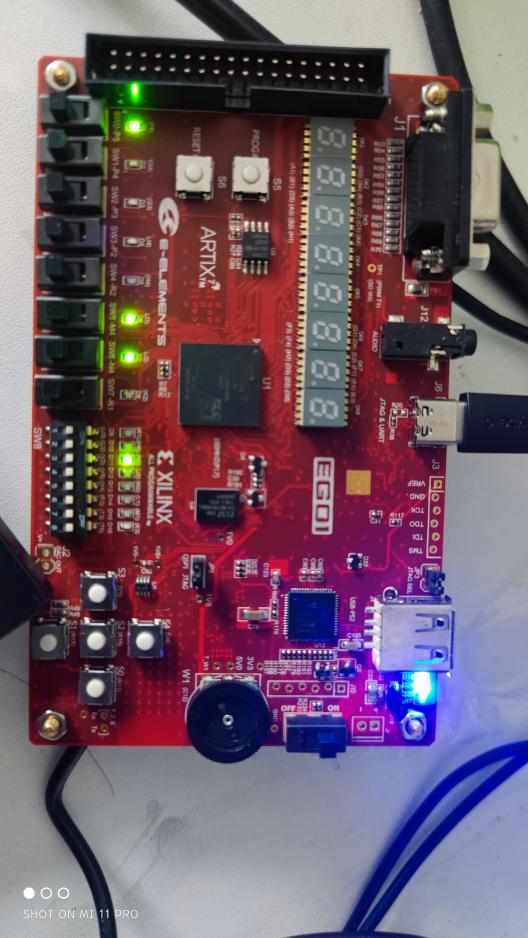


图5-7-2 sub指令图片

③ beg指令高16位：1101000001000011；

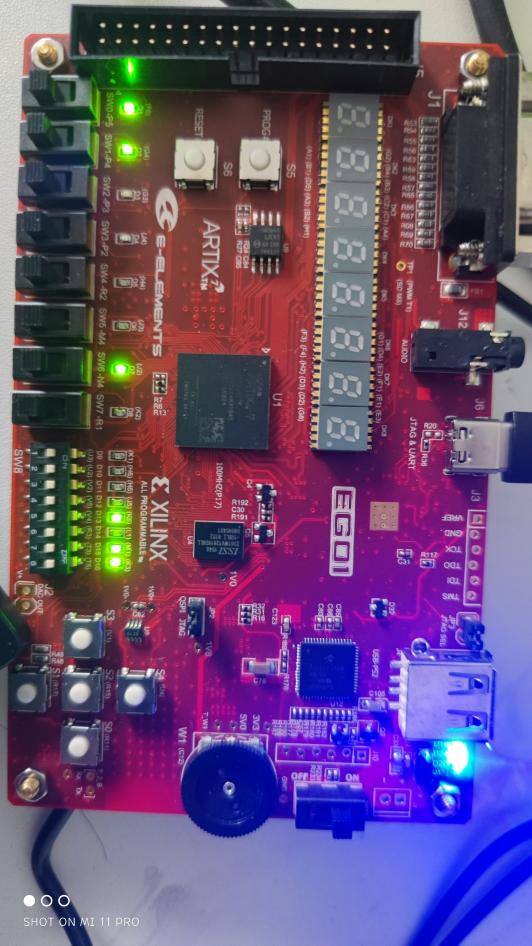


图5-7-3 beg指令图片

④ ori指令高16位：0100100000100001；

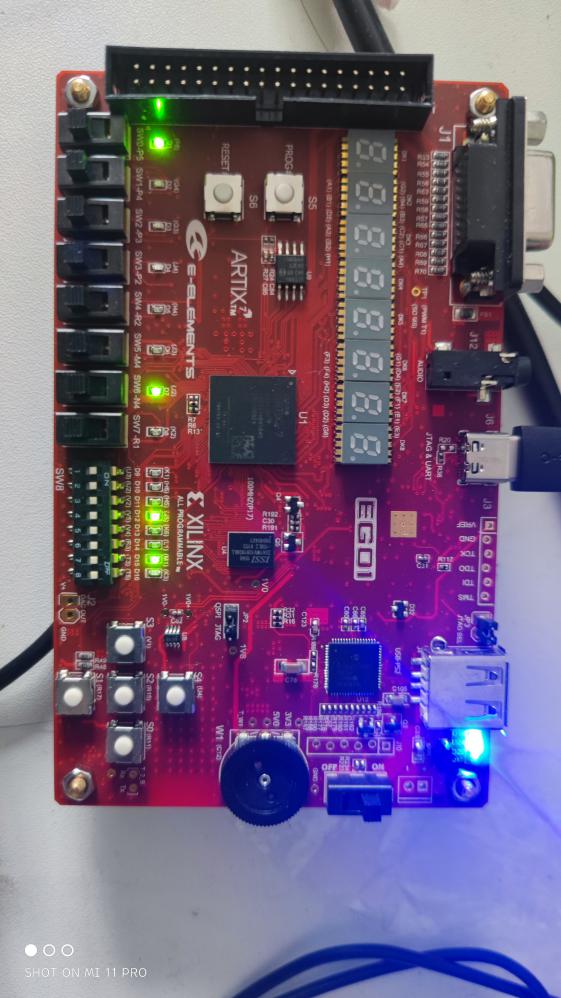


图5-7-4 beg指令图片

# 总结

通过此次课程设计，让我对CPU的设计有了一个更加深入的理解，并且明白了多周期CPU和单周期CPU的区别，让我想起了在计算机组成原理里面学习的知识，让我掌握了CPU执行的各个阶段，包括取指令、指令译码、指令执行、存储器访问、结果写回，让我对这些阶段有了一个更加深入的理解，以及多周期CPU中每条指令执行的各个状态间的转移，控制器发出不同控制信号来使CPU能够按照取出指令的含义执行，这些内容都让我印象非常深刻。

同时我学会了如何使用Vivado，在仿真图中如何观测各个寄存器变量的值，并且让我学会了verilog硬件描述语言，这跟我们之前学习的编程语言有一定的差别，但也有一定的相似之处，让我对硬件描述语言产生了兴趣，最主要的是从画整体框架图到设计各个模块再到在vivado中编写verilog代码以及调试，这中间遇到了很多的问题，稍有不慎调试就不成功，通过不断的修改bug，我学会了如何分析出现的问题并且上百度查找解决方案，这些都是我不断学习和摸索的过程，提高了我的动手实践能力。

让我收获最多的是，最后能够通过vivado连接开发板，并且通过编写约束文件把程序中的寄存器变量和开发板的某些端口对应起来，生成bit流后烧录在开发板上，通过按键来控制时钟，开发板按照verilog程序闪烁对应的LED灯,这时候让我硬件有了更加深入的理解。

从多周期CPU开始设计到最后烧录在开发板上，我用了将近一周的时间，学会了很多知识，同时也让我对多周期CPU有了更加深入的理解，这次课程设计让我收获很多，我也希望CPU课程设计能够发展的越来越好。

**参考文献：**

[]