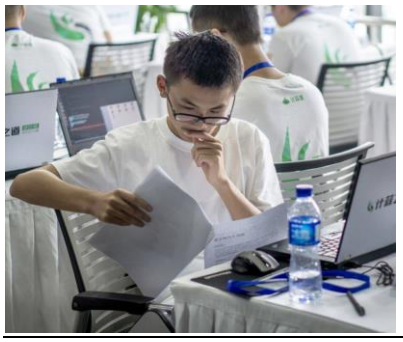
实验九--多周期 IP 核集成 CPU 实验报告

姓名：蒋仕彪 学号：3170102587 专业：求是科学班（计算机）1701

课程名称：计算机组成实验 同组学生姓名：

实验时间：2019-5-14 实验地点：紫金港东 4-509 指导老师：马德

个人形象照：



一、实验目的和要求

1. 深入理解 CPU 结构
2. 学习 CPU 性能优化:多周期
3. 建立多周期 CPU 测试应用环境
4. IP 核深入应用

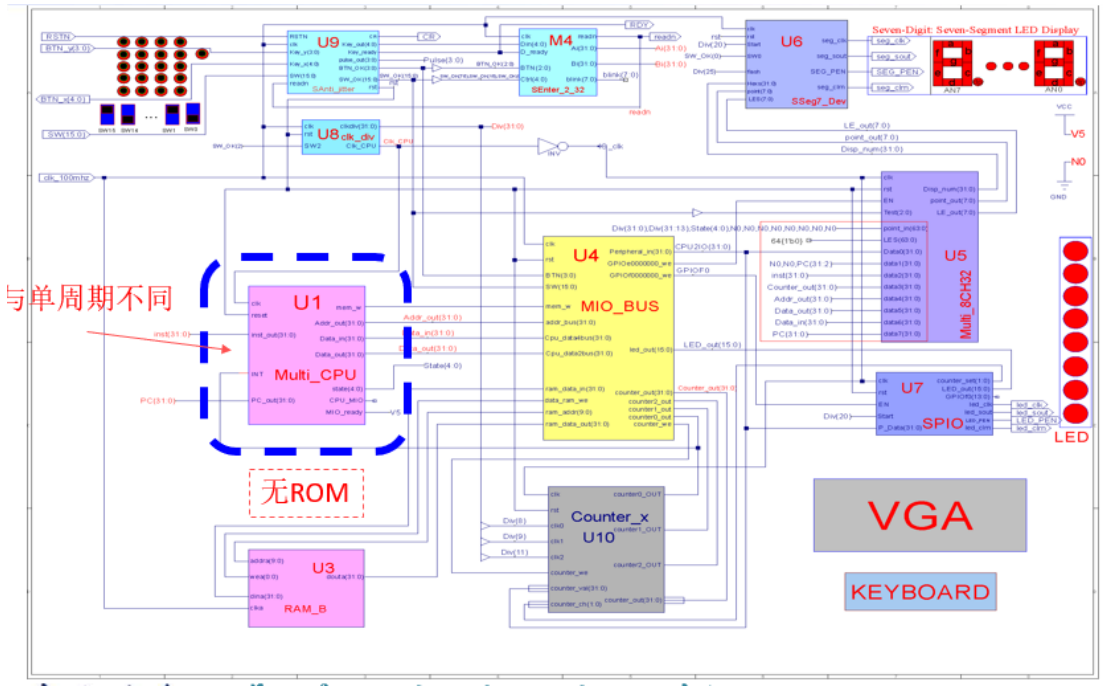
二、实验任务和原理

2.1 实验任务

1. 搭建多周期 CPU 测试应用环境
    - ■ 用结构描述重建 Exp03 顶层模块
        - □ 用多周期 CPU 核替换单周期
2. 用数据通路和控制器核集成替换 CPU 核
        - □ 除 CPU 外复用 Exp03 的部件模块

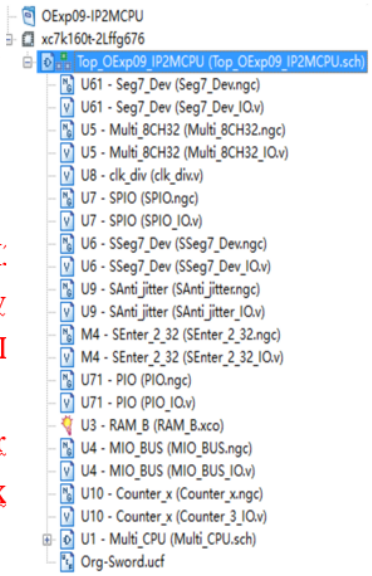
2.2 实验原理

2.2.1 多周期处理器测试框架或 SOC 总览



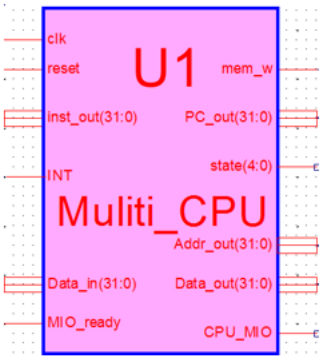
2.2.2 系统分解为九个子模块

- 用此9个模块，用结构描述建立SOC测试构架
- 集成实现多周期处理器SOC
  - U1： MCPU -Muliti-CPU
  - U2： ROM -ROM_D
  - U3： RAM -RAM_B
  - U4： 总线(含外设3~4) -MIO_BUS
  - U5： 7段显示接口 -Multi_8CH
  - U6： 外设1- 7段显示设备 -SSeg_Dev
  - U7： 外设2-GPIO接口及LED -SPI
  - U8： 辅助模块一，通用分频模块 -clk_div
  - U9： 辅助模块二，机械去抖模块 -SAnti_jitter
  - U10：通用计数器 -Counter_x
- 以上除U1外与单周期共享



2.2.3 （核心）U1-多周期 CPU 模块：Multi-CPU

- MIPS 构架
  - RISC体系结构
  - 三种指令类型
- SOC测试模块的处理核心
  - 由数据通路和控制器二个核组成
    - 本实验直接用2个IP核集成
    - 本实验也可先用CPU IP核
      - 调试通过后再用2个IP集成
  - 本实验可用IP Core- U1
    - 核调用模块Multi_CPU.ngc
    - 核接口信号模块(空文档)：Multi_CPU.v
    - 核模块符号文档：Multi_SCPU.sym



2.2.4 CPU 核接口空模块-Multi_CPU.v

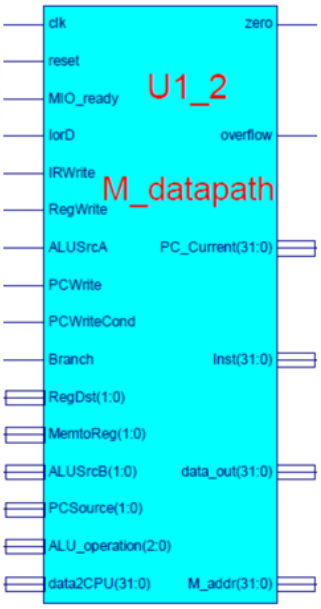
```
module    Muliti_CPU ( input wire clk,
                       input wire reset,
                       input wire MIO_ready,      // be used：=1

                       output wire[31:0]PC_out, //Test
                       output[31:0] inst_out,       //TEST
                       output wire mem_w,           //存储器读写控制
                       output wire[31:0]Addr_out,//数据空间访问地址
                       output wire[31:0]Data_out//数据输出总线
                       input wire [31:0]Data_in, //数据输入总线
                       output wire CPU_MIO,     // Be used
                       input wire  INT              //中断
                       output[4:0]state             //Test
                       );
endmodule
```

注意与单周期区别

2.2.5 CPU 部件之一 数据通路：M_datapath

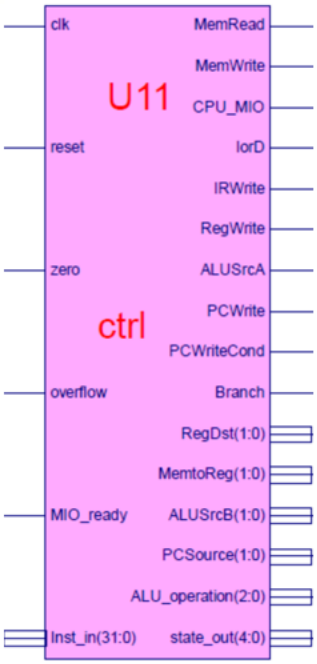
MUX选择更多输入以兼容扩展

- 本实验用IP 软核- **M_datapath**
  - 核调用模块M_atapath.ngc
  - 核接口信号模块：M_datapath.v
  - 核模块符号文档：M_datapath.sym
- 重要信号
  - Inst：指令寄存器输出
  - PC_Current：当前PC(PC+4)
  - M_addr：存储器地址
  - Branch(教材中的beq)：
    - =1：beq
    - =0：bne
  - PCWriteCond：Branch指令



2.2.6 CPU 部件之二-控制器：ctrl

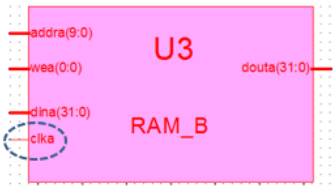
- 本实验用IP 软核- **ctrl**
  - 核调用模块ctrl.ngc
  - 核接口信号模块(空文档)：ctrl.v
  - 核模块符号文档：ctrl.sym
- 重要信号
  - MIO_ready：外设就绪
    - =0 CPU等待
    - =1 CPU正常运行
    - 本实验恒等于1
  - Inst_in：指令输入，来自IR输出
  - State_out：状态编码，用于测试



2.2.7 U3-指令代码存储模块：RAM_B

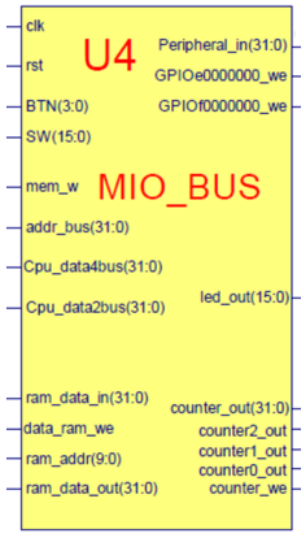
用Distributed Memory Generator没有clk信号
请编辑删除clka引脚。SP3平台用不用

- **RAM_B**
  - 将Lab3的ROM和RAM合并
    - 数据代码存储共享
  - FPGA内部存储器
    - Block Memory Generator或Distributed Memory Generator
  - 容量与Lab3的RAM_B相同
    - 1024×32bit
  - 核模块符号文档：RAM_B.sym
    - 自动生成符号不规则，需要修整
- **本实验需要重新生成IP固核**
  - RAM初始化文档：**mem.coe**
    - **代码与数据合并在一个存储器中**
  - 核调用模块RAM_B.xco
    - 生成后自动调用关联，不需要空文档



2.2.8 U4-总线接口模块：MIO_BUS

- **MIO_BUS**
  - CPU与外部数据交换接口模块
  - 本课程实验将数据交换电路合并成一个模块
    - 非常简单，但非标准，扩展不方便
    - 后继课程采用标准总线
      - Wishbone总线
- **基本功能**
  - 数据存储、Seg7、SW、BTN和LED等接口
- **本实验用IP 软核- U4**
  - 核调用模块MIO_BUS.ngc
  - 核接口信号模块(空文档)：MIO_BUS.v
  - 核模块符号文档：MIO_BUS.sym



注：其余原理与单周期一样。

三、主要仪器设备

1. 计算机（Intel Core i5 以上，4GB 内存以上）系统　　1 套
2. 计算机软硬件课程贯通教学实验系统　　1 套
3. Xilinx ISE14.4 及以上开发工具　　1 套

四、操作方法与实验步骤

4.1 设计工程：OExp09-IP2MCPU

◎ **建立 CPU 调试、测试和应用环境**

- 顶层用 HDL 实现，调用 IP 核模块
  - 模块名：Top_OExp09_IP2MCPU.sch

◎ **SOC 集成技术实现测试系统构架**

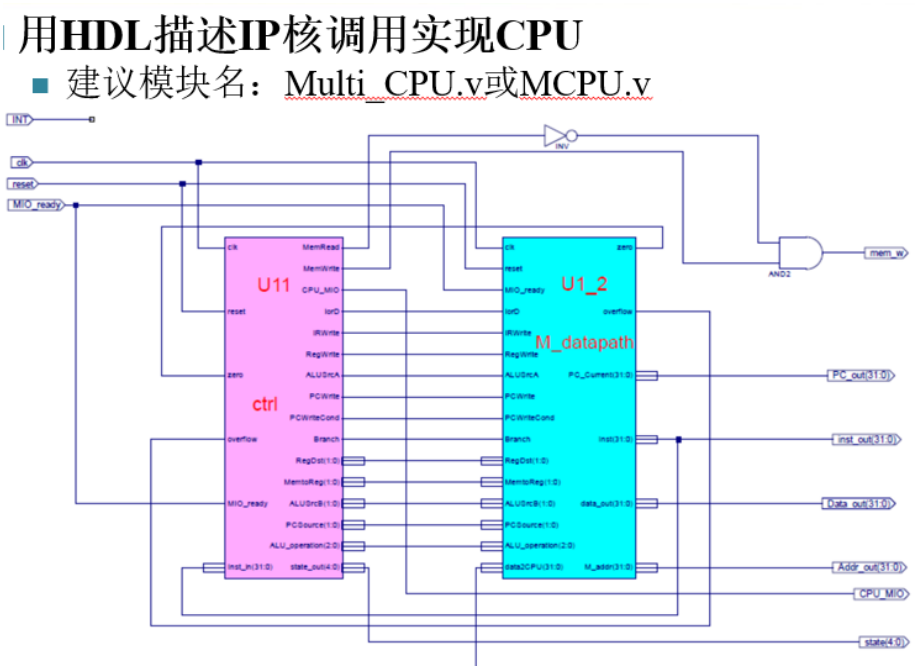
- 复用实验三的模块，除 SCPU 外
  - CPU (第三方 IP 核)：U1
  - RAM (ISE 构建 IP 核)：U3
  - 总线(第三方 IP 核)：U4
  - 八数据通路模块(实验一 Multi_8CH32)：U5
  - 七段显示模块(实验二 SSeg7_Dev IP)：U6
  - LED 显示模块(实验二 SPIO 模块)：U7
  - 通用分频模块(clk_div)：U8
  - 开关去抖模块(IP 核)：U9
  - 数据输入模块(IP 核)：M4(目前没有使用)

◎ **分解 CPU 为二个 IP 核**

- SOC 调试通过后用二个 IP 核构建 MCPU

4.2 设计多周期 CPU 核调用模块

- 用 HDL 描述 IP 核调用实现 CPU
  - 建议模块名：Multi_CPU.v 或 MCPU.v

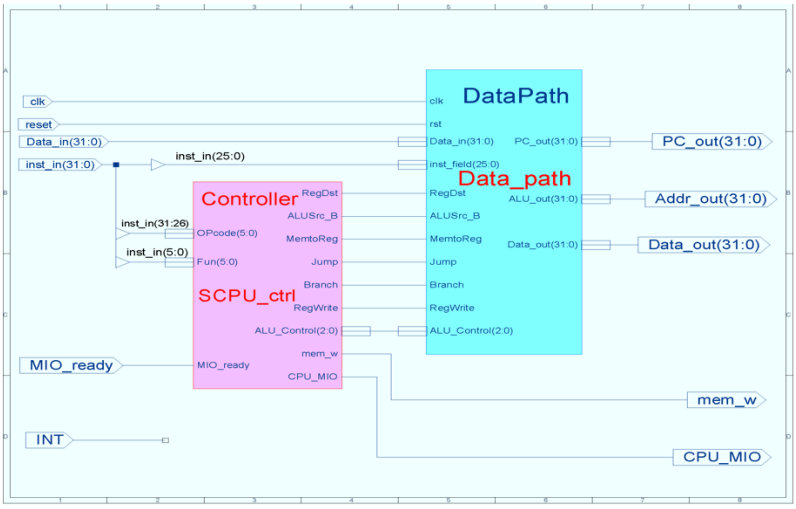


4.3 核集成 CPU 描述结构参考

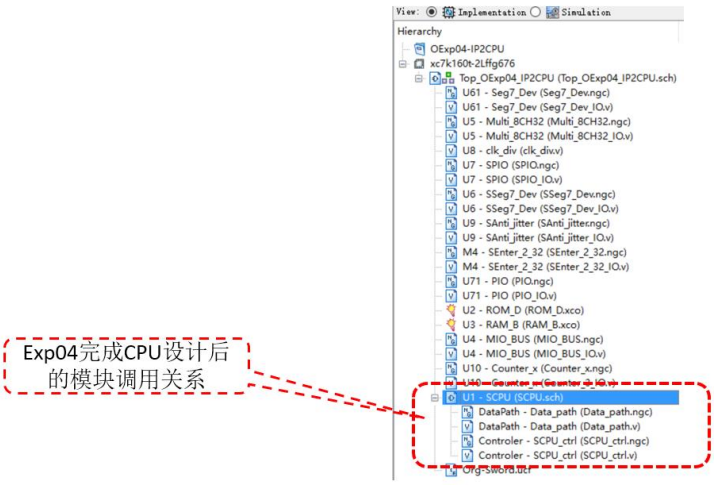
```
module    Muliti_CPU(input clk,                    //muliti_CPU
          input reset,
          input MIO_ready,
          output[31:0] PC_out,   //TEST
          output[31:0] inst_out,  //TEST
          output mem_w,
          output[31:0] Addr_out,
          output[31:0] Data_out,
          input [31:0] Data_in,
          output CPU_MIO,
          input INT,
          output[4:0]state          //Test
         );
```

4.3 用原理图设计 CPU

用逻辑原理图输入CPU设计



4.4 最终调用模块结构



五、实验结果与分析

5.1 multiCPU 顶层模块展示

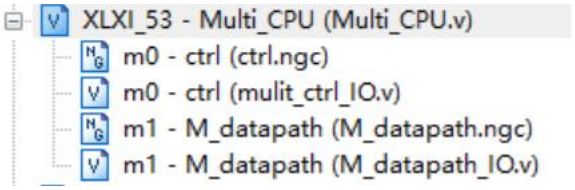
本实验的核心是：首次建立多周期的 CPU。
因为是第一次建立，这次只需把 ctrl 和 datapath 的带接口的 ngc 拼起来即可。
以前的 CPU 顶层都是画图的，为了使所有部件都代码化，这次我选择用代码写。

下面展示一下 **multi_CPU.v**

```
module   Multi_CPU (
            input wire clk,
            input wire reset,
            input wire MIO_ready,
            input wire  INT,
            output wire[31:0]PC_out,
            output[31:0] inst_out,
            output wire mem_w,
            output wire[31:0]Addr_out,
            output wire[31:0]Data_out,
            input wire [31:0]Data_in,
            output wire CPU_MIO,
            output[4:0]state
            );
    wire zero, overflow, MemRead, MemWrite;
    wire IorD, IRWrite, RegWrite, ALUSrcA, PCWrite, PCWriteCond,
Branch;
    wire [1:0] RegDst, MemtoReg, ALUSrcB, PCSource;
    wire [2:0] ALU_operation;
    wire [4:0] status_out;
    assign mem_w = (~MemRead) && MemWrite;
    ctrl m0
(.clk(clk), .reset(reset), .zero(zero), .overflow(overflow), .MIO_ready
(MIO_ready),.Inst_in(inst_out), .MemRead(MemRead), .MemWrite(MemWrite),
.CPU_MIO(CPU_MIO), .IorD(IorD), .IRWrite(IRWrite), .RegWrite(RegWrite),
.ALUSrcA(ALUSrcA), .PCWrite(PCWrite), .PCWriteCond(PCWriteCond), .Branc
h(Branch), .RegDst(RegDst), .MemtoReg(MemtoReg), .ALUSrcB(ALUSrcB), .PC
Source(PCSource), .ALU_operation(ALU_operation), .state_out(state));
    M_datapath
m1(.clk(clk), .reset(reset), .MIO_ready(MIO_ready), .IorD(IorD), .IRWri
te(IRWrite),.RegWrite(RegWrite), .ALUSrcA(ALUSrcA), .PCWrite(PCWrite),
.PCWriteCond(PCWriteCond), .Branch(Branch), .RegDst(RegDst), .MemtoReg(
MemtoReg), .ALUSrcB(ALUSrcB), .PCSource(PCSource), .ALU_operation(ALU_o
```

```
peration), .data2CPU(Data_in), .zero(zero), .overflow(overflow), .PC_Cu
rrent(PC_out), .Inst(inst_out), .data_out(Data_out), .M_addr(Addr_out))
;
endmodule
```

两个关键部件 SCPU_ctrl 和 Data_path 依然是调用 ngc 的。接口如下：

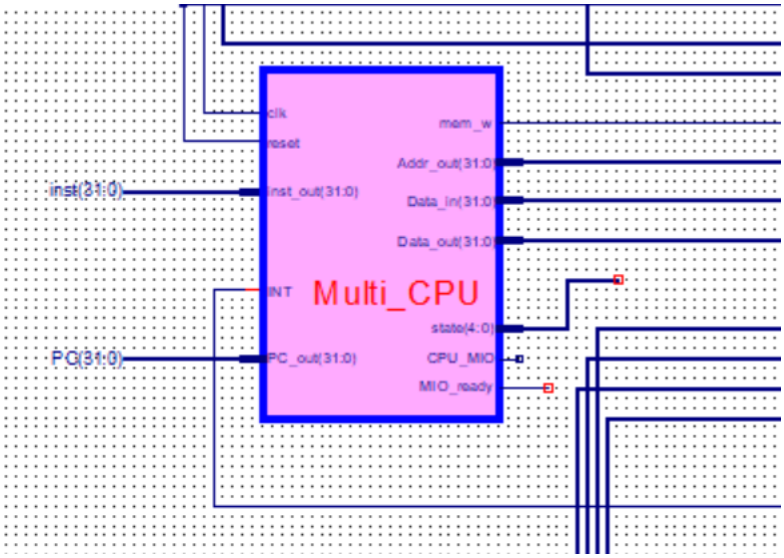


5.2 搭建顶层模块

只需把单周期实验的顶层模块拿过来加以改动即可。

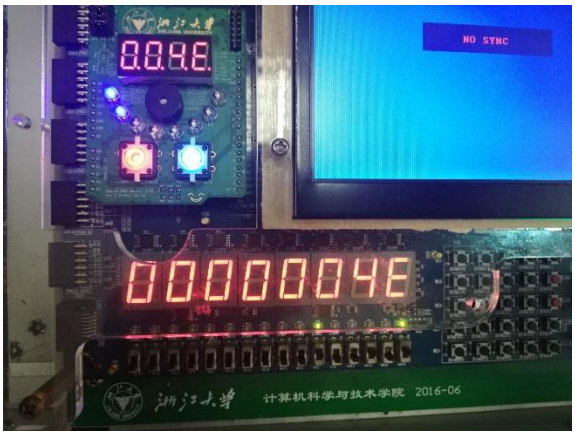
因为顶层模块接线比较复杂，我依然采用了画图的形式（而且只需做小小改动，换一下接线即可）

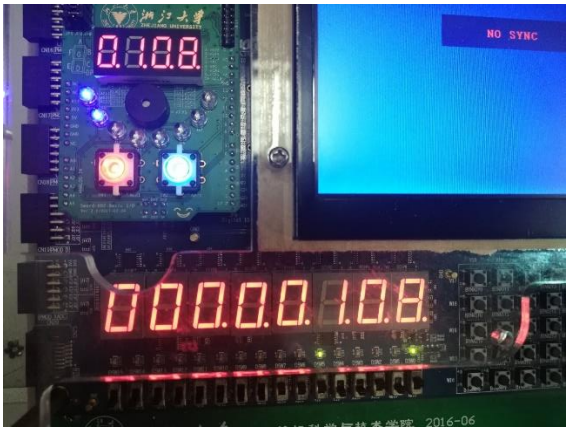
但是 sch 还是有一个不方便的地方，每当我 CPU 符号重新生成（比如少写了一个接口），在顶层模块中的接口会全部乱掉，又要重新接了。



5.3 实验现象一

SW[0]=1, SW[2]=1，SW[7:5]=111。
输出 CPU 指令字节地址 PC_out。



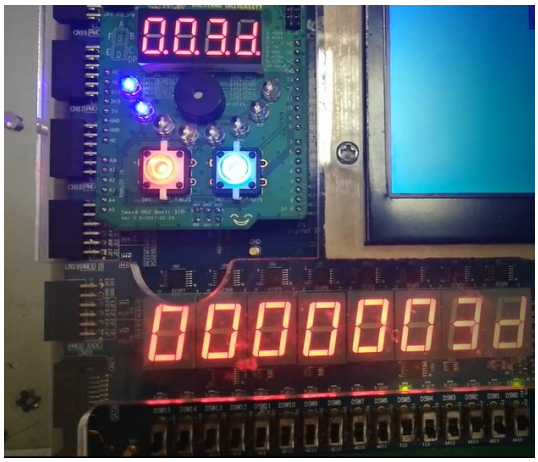


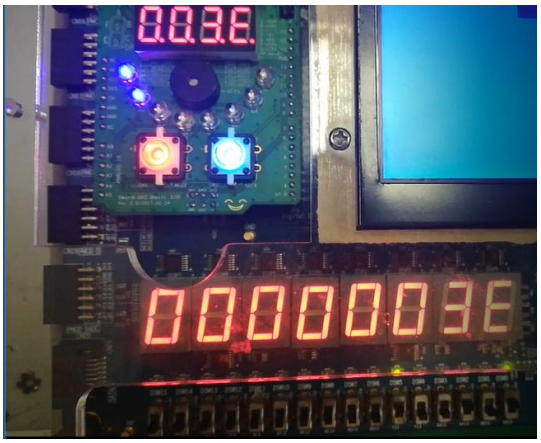
□ 实验现象 1

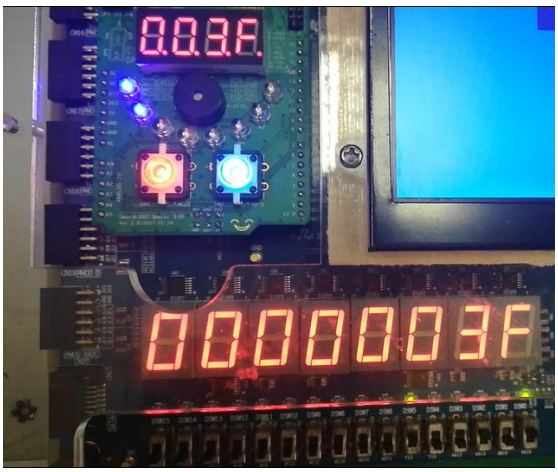
5.4 实验现象二

SW[0]=1, SW[2]=1，SW[7:5]=001。

输出 CPU 指令字地址 PC_out[31:2]。





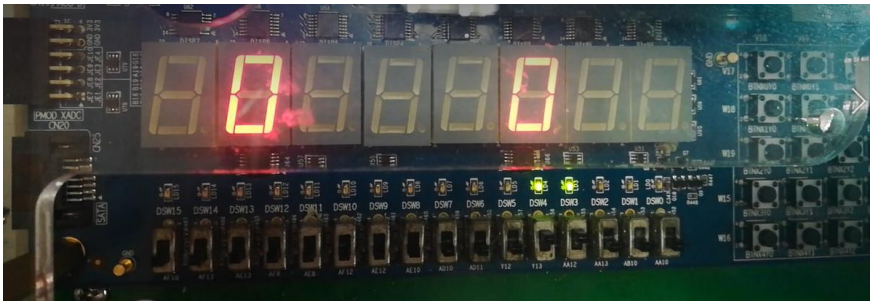


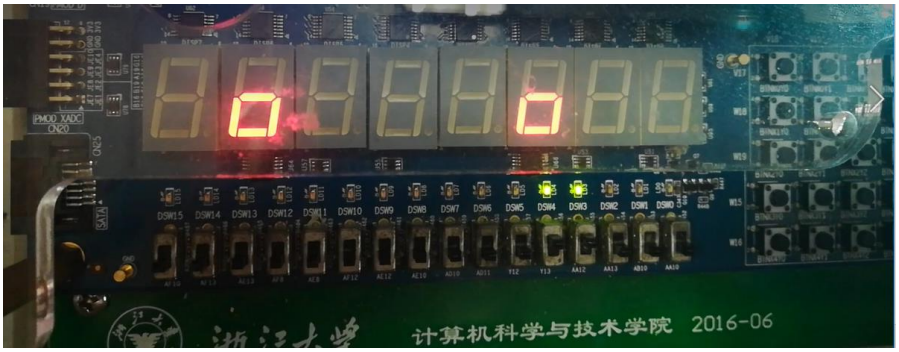
□ 实验现象 2

5.5 实验现象三

SW[0]=0, SW[3]=1，SW[4]=1。

矩形框跳舞。



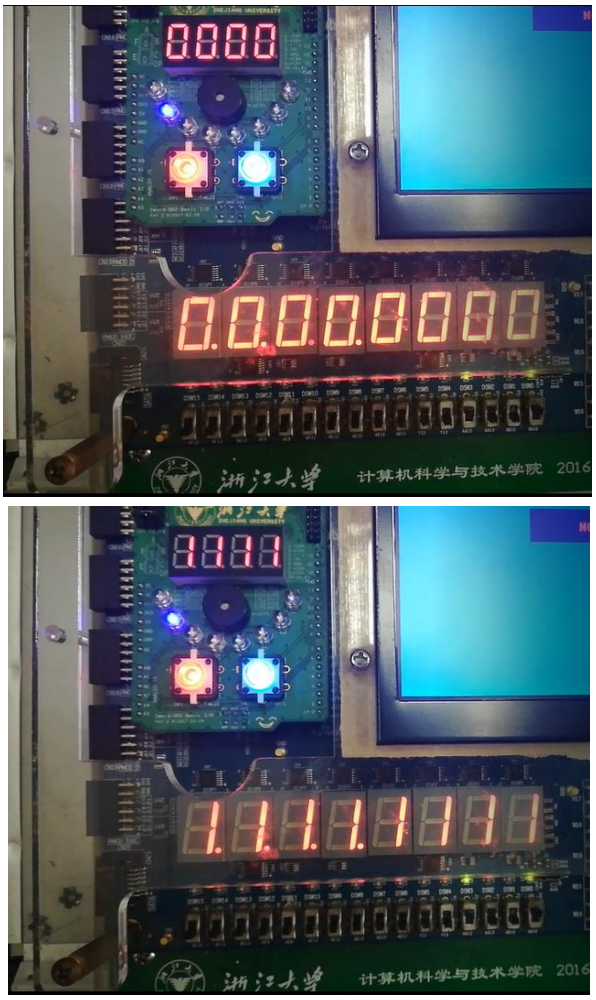


◘ 实验现象 3

5.4 实验现象四

SW[0]=1, SW[3]=1。

全速时钟。



□ 实验现象 4

六、讨论、心得

随着理论的推进，实验也终于从单周期到多周期了。

从单周期换到多周期，**体现在数位板上的一个最大的特点是：运行每一条指令的时间是不一样的。**其实也很好理解，由理论课学到的知识，SW 和 LW 很慢，需要 5 步；不同的指令计算需要 4 步；J 指令只要 3 步。不同指令步数不同，运行速度自然不同。

理论上来说，多周期会比单周期速度快，但是实际上速度差不多。好像是因为多周期和单周期的时钟频率都差不多，而且比较低，多周期无法体现优势。

本次实验还算比较简单。因为是多周期的第一次实验，

此外，多周期的 CPU 会比单周期稍微复杂一些。

实验十-- CPU 设计之数据通路

姓名：蒋仕彪 学号：3170102587 专业：求是科学班（计算机）1701

课程名称：计算机组成实验 同组学生姓名：

实验时间：2019-5-21 实验地点：紫金港东 4-509 指导老师：马德

一、实验目的和要求

1. 深入运用寄存器传输控制技术
2. 深入掌握 CPU 的核心：数据通路组成与原理
3. 设计多周期数据通路
4. 测试方案的设计
5. 测试程序的设计

二、实验任务和原理

2.1 实验任务

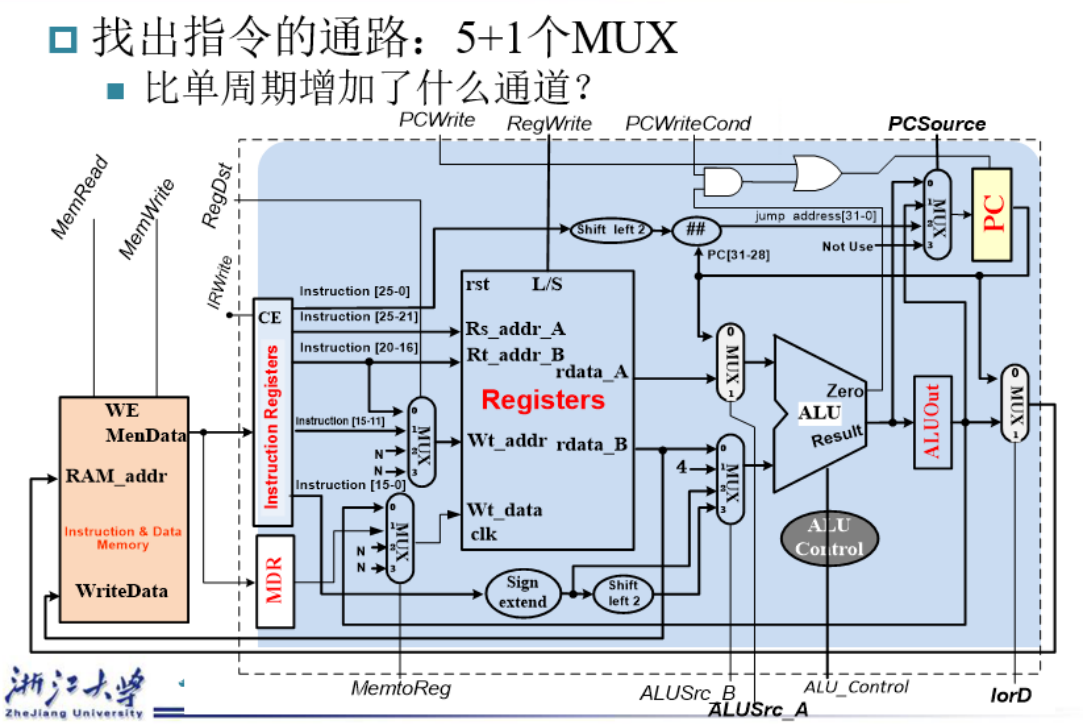
1. 设计 9^{+}条指令的多周期数据通路
   - ■ 设计多周期数据通路逻辑原理图

   R-Type：add，sub，and，or，slt，nor*；
   I-Type： lw，sw，beq；
   J-Type： J
   - ■ 用硬件描述语言设计实现数据通路
   - □ ALU 和 Regs 调用 Exp04 设计的模块
   - ■ 替换 Exp09 数据通路核
2. 数据通路测试
   - ■ 设计测试方案与测试程序

   通路测试：I-格式通路、R-格式通路

2.2 实验原理

2.2.1 多周期数据通路结构



2.2.2 多周期控制信号定义

信号	源数目	功能定义	赋值0时动作	赋值1时动作
ALUScrA	?	ALU端口A输入选择	选通PC	选通A暂存器
ALUSrc_B		选择ALU端口B数据来源	01:常数4 00:选能寄存器B 10:扩展后的数据 11:扩展左移的数据;	
RegDst	?	寄存器写地址选择(考虑扩展)	选择指令rt域	选择指令rs域
MemtoReg	?	寄存器写数据选择(考虑扩展)	选通暂存器F	选通暂存器MDR
IorD	?	(新增)选择存储器地址端addr的来源	选通PC	选通暂存器F
PCSource	?	(新增)选择打入PC的数据来源	00:ALU输出,01:选择F暂存器 10:jump	
PCWriteCond	?	(新增)设置PC的工作模式	未使用	将PC置为写模式
IRWrite	?	(新增)设置IR工作模式	未使用	将IR置为写模式
Branch	?	Beq指示(考虑Bne扩展)	选择PC+4	转移地址
RegWrite	-	寄存器写控制		写模式
MemWrite	-	存储器写控制		
MemRead	-	存储器读控制		
ALU_Control	000- 111	3位ALU操作控制	参考表 Exp04	Exp04

三、主要仪器设备

1. 计算机（Intel Core i5 以上，4GB 内存以上）系统　　1 套
2. 计算机软硬件课程贯通教学实验系统　　1 套
3. Xilinx ISE14.4 及以上开发工具　　1 套

四、操作方法与实验步骤

4.1 设计工程：OExp10-MDP

◎ **设计 CPU 之数据通路**

- 根据理论课分析讨论设计 9+条指令的数据通路
- 仿真测试 M_Datapath.v 模块

◎ **集成替换验证通过的数据通路模块**

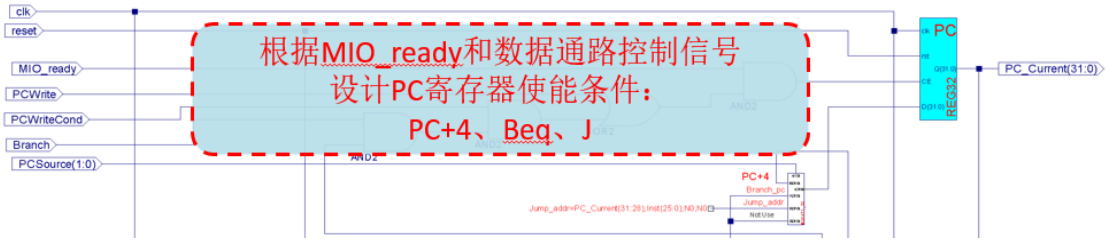
- 替换实验四(Exp09)中的 M_Datapath.ngc 核
- 顶层模块延用 Exp09
  - ⊙ 模块名：Top_OExp09_MDP.v

◎ **测试数据通路模块**

- 设计测试程序(MIPS 汇编)测试：
- ALU 功能
- I-指令通路
- R-指令通路

4.4 设计要点

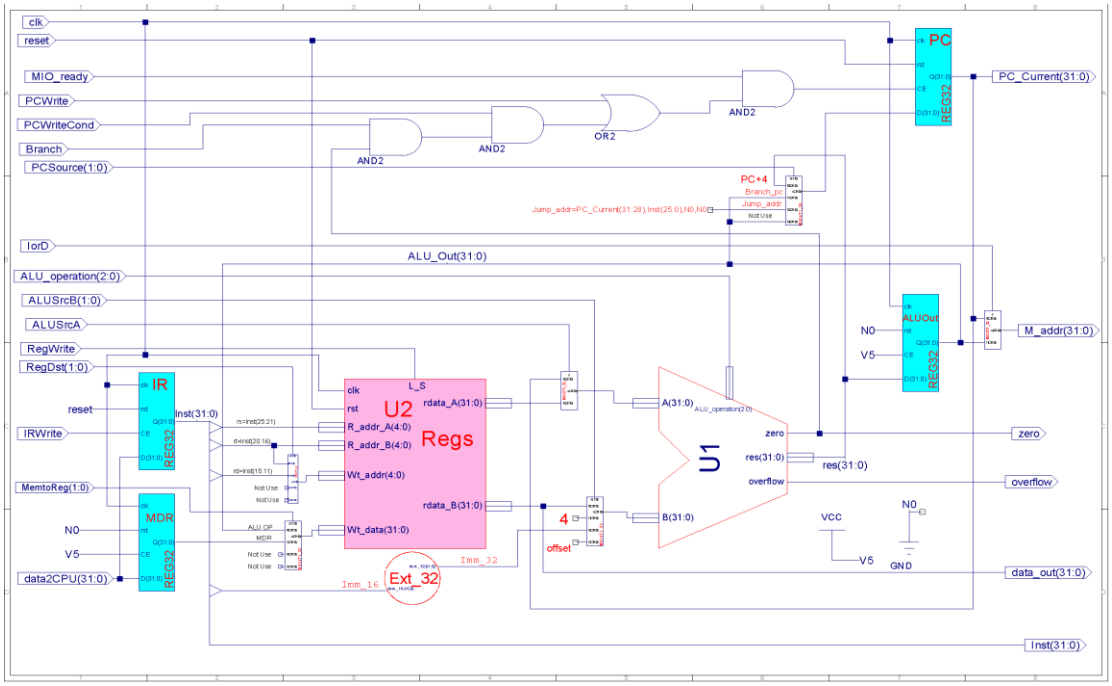
- **建立 DataPath HDL 输入模板**
- **设计 PC 通路**
  - 调用 REG32 和 MUX4T1_32 模块



  - 参考描述

```
assign      CE = MIO_ready ?&& (PCWrite || (PCWriteCond && zero&&Branch));
                  res(31:0)   ALU_Out(31:0)  PC_Current(31:28),Inst(25:0),N0,N0
MUX4T1_32  MUX6(.I0(PC+4?), .I1(Beq?), .I2(Jump?), .I3(No Use), .o(PC_next));
REG32      PC(.clk(???), .rst(???), .CE(CE), .D(PC_next), .Q(PC_Current));
        ……          clk      reset
```

4.3 完整支持 9+条指令的数据通路参考描述

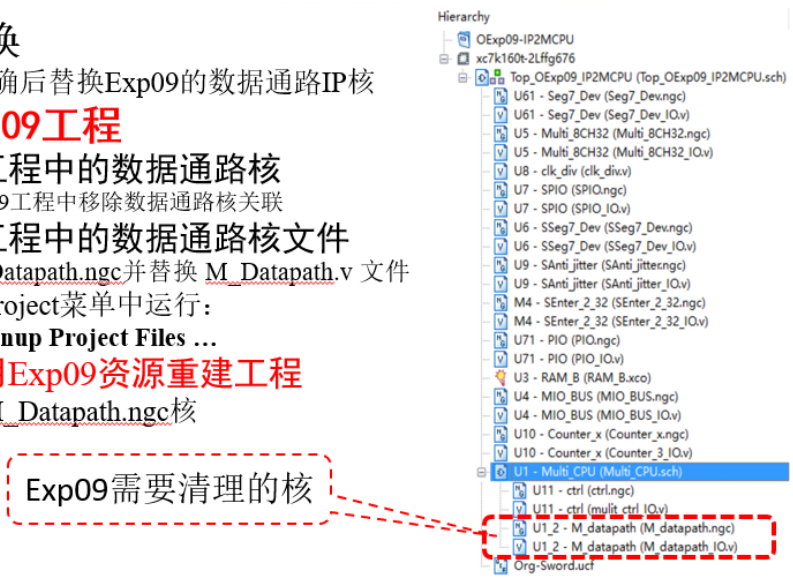


4.4 仿真测试

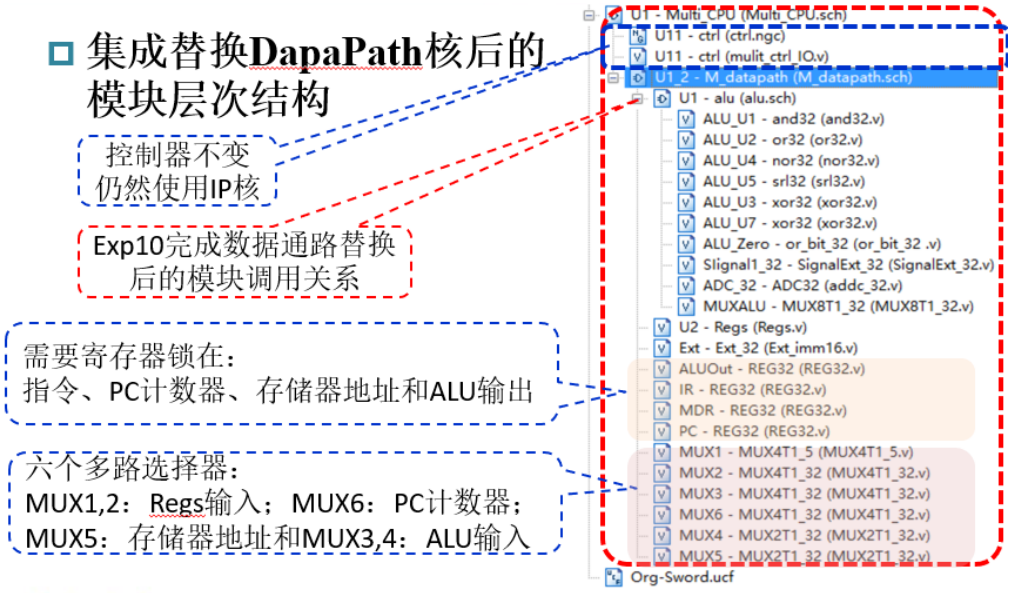
- **数据通路仿真调试**
  - 参考实验五，注意多周期时序
  - M_Datapath 模块仿真
    - 语法检查没有 Errors 和 warnings 后仿真测试
    - 仿真激励代码设计要点
      - 只做功能性测试，不做性能和完备性测试
      - **通路功能测试**
        - » 选择 9 条指令所有可能通路的代表指令
        - » 激励输入：
          - 计算出不同指令控制信号、代表数据和时序
          - clk、rst
      - **ALU 功能测试**
        - » 选择 add、and、sub、or、nor、slt 指令
          - 计算出对应指令的输入控制信号和代表数据
        - » 选择 Beq 比较、Load 和 Stroe 测试地址计算
      - **Regs 功能测试**
        - » add 指令代表作寄存器遍历测试

4.5 M_Datapath 替换集成

- 集成替换
  - 仿真正确后替换Exp09的数据通路IP核
- 清理Exp09工程
  - 移除工程中的数据通路核
    - Exp09工程中移除数据通路核关联
  - 删除工程中的数据通路核文件
    - M_Datapath.ngc并替换 M_Datapath.v 文件
    - 在Project菜单中运行：
      Cleanup Project Files …
  - 建议用Exp09资源重建工程
    - 除M_Datapath.ngc核



- 集成替换**DapaPath**核后的模块层次结构



五、实验结果与分析

5.1 M_datapath.v 源码展示

ALU，regs 等部件在单周期时已经写过，可以直接拿过来。

所以本次实验的核心是设计多周期的 datapath。

多周期的数据通路连线会十分复杂，而且容易出错，于是我选择写代码的方式。一个要注意的地方是，很多单周期中具有的接口，在多周期处要拓宽成 2 位或者 3 位。

```
module M_datapath(input clk,
                  input reset,

                  input MIO_ready,
                  input IorD,
                  input IRWrite,
                  input[1:0] RegDst,
                  input RegWrite,
                  input[1:0]MemtoReg,
                  input ALUSrcA,

                  input[1:0]ALUSrcB,
                  input[1:0]PCSource,
                  input PCWrite,
                  input PCWriteCond,
                  input Branch,
                  input[2:0]ALU_operation,

                  output[31:0]PC_Current,
                  input[31:0]data2CPU,
                  output[31:0]Inst,
                  output[31:0]data_out,
                  output[31:0]M_addr,

                  output zero,
                  output overflow
                 );
    wire V5,N0;
    assign V5 = 1'b1;
    assign N0 = 1'b0;

    wire [31:0] Q, ALU_out;
    wire [31:0] RdataA, RdataB;
```

```
        wire [31:0] imm32, offset, four, res;
        wire [31:0] Jump_addr, Prenode;

        REG32 IR
(.clk(clk), .rst(reset), .CE(IRWrite), .D(data2CPU), .Q(Inst));
        REG32 MDR(.clk(clk), .rst(N0), .CE(V5), .D(data2CPU), .Q(Q));

        Regs U2
(.clk(clk), .rst(reset), .R_addr_A(Inst[25:21]), .R_addr_B(Inst[20:16])
,.Wt_addr(RegDst[0] ? Inst[15:11] : Inst[20:16]),.
Wt_data(MemtoReg[0] ? Q :ALU_out),.L_S(RegWrite), .rdata_A(RdataA),
.rdata_B(RdataB));

        assign data_out = RdataB;
        assign imm32={Inst[15], Inst[15], Inst[15], Inst[15], Inst[15],
Inst[15], Inst[15], Inst[15], Inst[15], Inst[15], Inst[15], Inst[15],
Inst[15], Inst[15], Inst[15], Inst[15], Inst[15:0]};
        assign four = 32'h00000004;
        assign offset = {imm32[29:0], N0, N0};

        ALU U1(.A(ALUSrcA ? RdataA : PC_Current), .B(ALUSrcB[1] ?
(ALUSrcB[0] ? offset : imm32) : (ALUSrcB[0] ? four :
RdataB)), .ALU_operation(ALU_operation), .zero(zero), .res(res), .overf
low(overflow));
        REG32
ALUOut(.clk(clk), .rst(N0), .CE(V5), .D(res), .Q(ALU_out));

        assign Jump_addr = {PC_Current[31:28], Inst[25:0], N0, N0};
        assign Prenode = PCSource[1] ? (PCSource[0] ? ALU_out :
Jump_addr) : (PCSource[0] ? ALU_out : res);

        REG32 PC(.clk(clk), .rst(reset), .CE(MIO_ready && ((Branch &&
zero && PCWriteCond) || PCWrite)), .D(Prenode), .Q(PC_Current));

        assign M_addr = IorD ? ALU_out : PC_Current;

endmodule
```

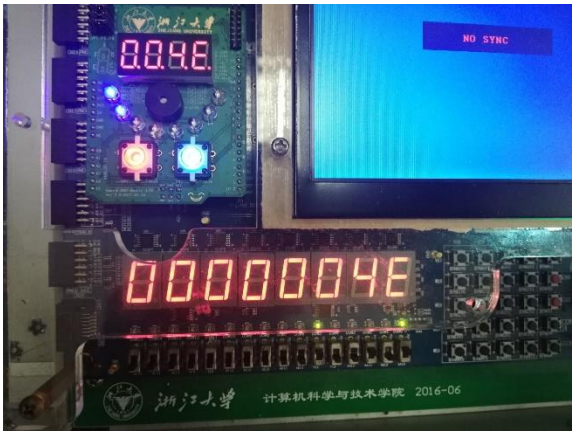
5.2 Multi_CPU.v 源码展示

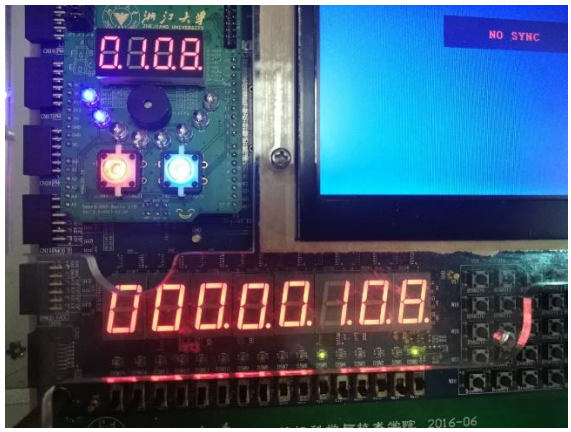
因为 multi_CPU.v 我是采用代码描述的，几乎不用修改就可以兼容 m_datapath.v。

```
module   Multi_CPU (
            input wire clk,
            input wire reset,
            input wire MIO_ready,
            input wire  INT,
            output wire[31:0]PC_out,
            output[31:0] inst_out,
            output wire mem_w,
            output wire[31:0]Addr_out,
            output wire[31:0]Data_out,
            input wire [31:0]Data_in,
            output wire CPU_MIO,
            output[4:0]state
            );
    wire zero, overflow, MemRead, MemWrite;
    wire IorD, IRWrite, RegWrite, ALUSrcA, PCWrite, PCWriteCond,
Branch;
    wire [1:0] RegDst, MemtoReg, ALUSrcB, PCSource;
    wire [2:0] ALU_operation;
    wire [4:0] status_out;
    assign mem_w = (~MemRead) && MemWrite;
    ctrl m0
(.clk(clk), .reset(reset), .zero(zero), .overflow(overflow), .MIO_ready
(MIO_ready),.Inst_in(inst_out), .MemRead(MemRead), .MemWrite(MemWrite),
.CPU_MIO(CPU_MIO), .IorD(IorD), .IRWrite(IRWrite), .RegWrite(RegWrite),
.ALUSrcA(ALUSrcA), .PCWrite(PCWrite), .PCWriteCond(PCWriteCond), .Branc
h(Branch), .RegDst(RegDst), .MemtoReg(MemtoReg), .ALUSrcB(ALUSrcB), .PC
Source(PCSource), .ALU_operation(ALU_operation), .state_out(state));
    M_datapath
m1(.clk(clk), .reset(reset), .MIO_ready(MIO_ready), .IorD(IorD), .IRWri
te(IRWrite),.RegWrite(RegWrite), .ALUSrcA(ALUSrcA), .PCWrite(PCWrite),
.PCWriteCond(PCWriteCond), .Branch(Branch), .RegDst(RegDst), .MemtoReg(
MemtoReg), .ALUSrcB(ALUSrcB), .PCSource(PCSource), .ALU_operation(ALU_o
peration), .data2CPU(Data_in), .zero(zero), .overflow(overflow), .PC_Cu
rrent(PC_out), .Inst(inst_out), .data_out(Data_out), .M_addr(Addr_out))
;
endmodule
```

5.3 实验现象一

SW[0]=1, SW[2]=1，SW[7:5]=111。
输出 CPU 指令字节地址 PC_out。

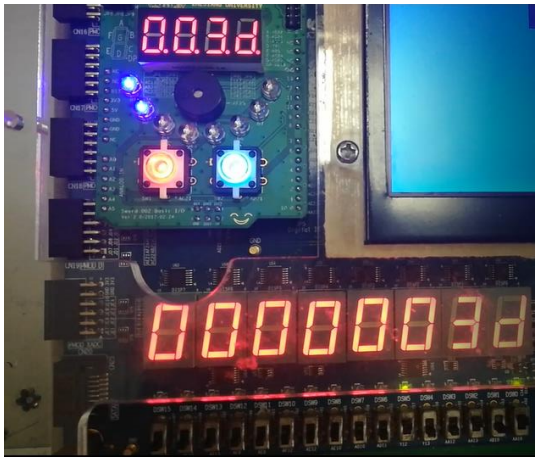


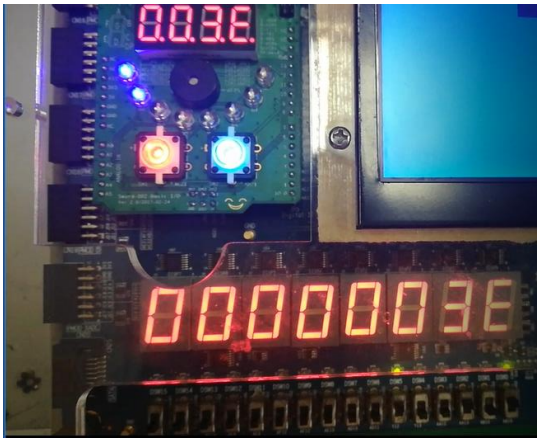


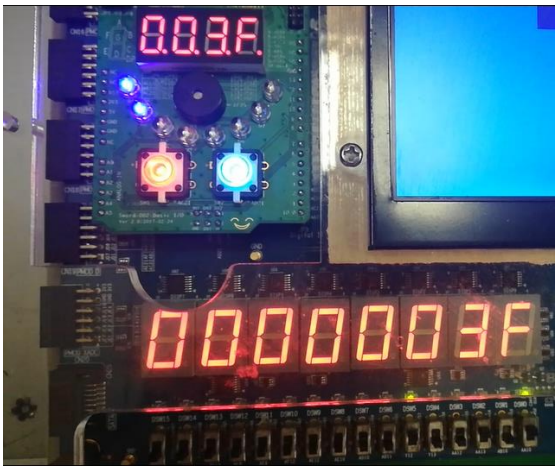
□ 实验现象 1

5.4 实验现象二

SW[0]=1, SW[2]=1，SW[7:5]=001。
输出 CPU 指令字地址 PC_out[31:2]。



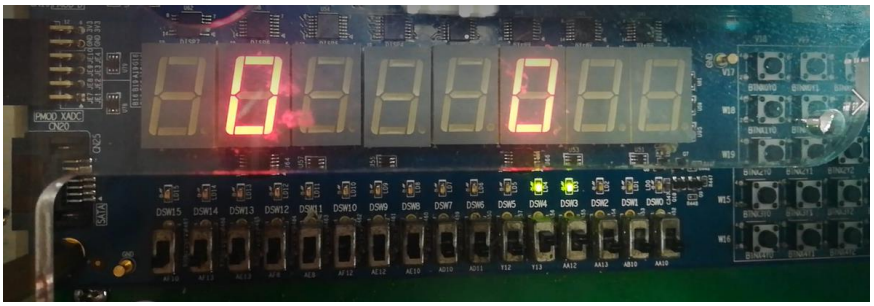


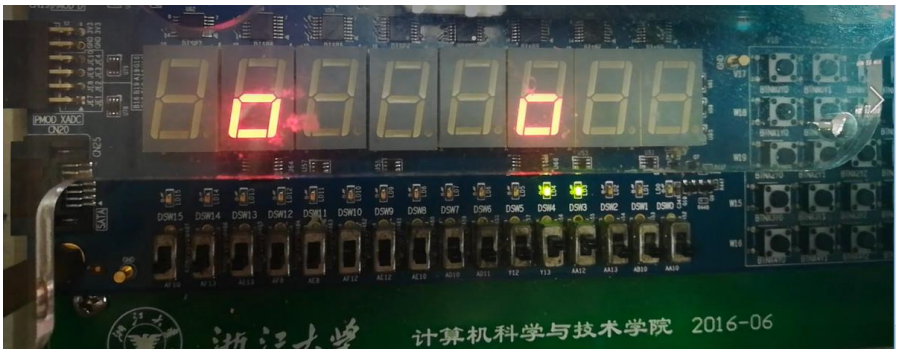


□ 实验现象 2

5.5 实验现象三

SW[0]=0, SW[3]=1，SW[4]=1。
矩形框跳舞。



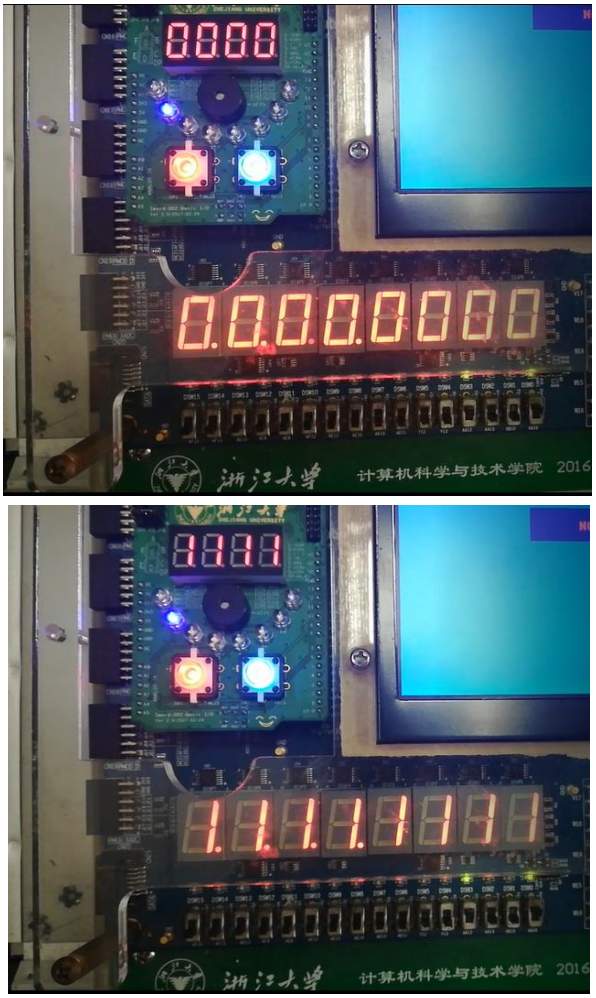


- 实验现象 3

5.6 实验现象四

SW[0]=1, SW[3]=1。

全速时钟。



- 实验现象 4

六、讨论、心得

上次实验是写好了 MCPU 的顶层模块，这次就是先拆解 Data_path 了。感觉实验设计的老师煞费苦心，设计的实验也蛮不错的。这和单周期的套路也很一般。

在最近的一次验收中，马老师突然路过我旁边，问我多周期输出的 PC_out 和单周期有什么不一样。那时候我正放的是跑马灯程序的 PC_out，我突然愣住了，感觉没有什么不一样的变化。他在旁边斥责了我一会后，我才突然意识到，多周期的 PC_out 执行的速度不一样，因为有些指令只要 3 步就行，有些却要五步。

感觉马老师煞费苦心想强调这一点，我却还是没有很快意识到>_<。

我也再一次深刻地认识到，写代码比画图优秀多了，不但易于移植，每次底层模块有一个小改动（比如增加了一个接口），顶层模块也几乎不用做出修改。

实验十一--多周期 CPU 设计-控制器设计

姓名：蒋仕彪 学号：3170102587 专业：求是科学班（计算机）1701

课程名称：计算机组成实验 同组学生姓名：

实验时间：2019-5-28 实验地点：紫金港东 4-509 指导老师：马德

一、实验目的和要求

1. 运用寄存器传输控制技术
2. 掌握 CPU 的核心：指令执行过程与控制流关系
3. 设计控制器
4. 学习测试方案的设计
5. 学习测试程序的设计

二、实验任务和原理

2.1 实验任务

1. 设计 9^{+}条指令的控制器
    - 用硬件描述语言设计实现控制器
        - 此实验在 Exp10 的基础上完成
2. 设计控制器测试方案：
    - OP 译码测试：R-格式、访存指令、分支指令，转移指令
    - 运算控制测试：Function 译码测试
3. 设计控制器测试程序

2.2 实验原理

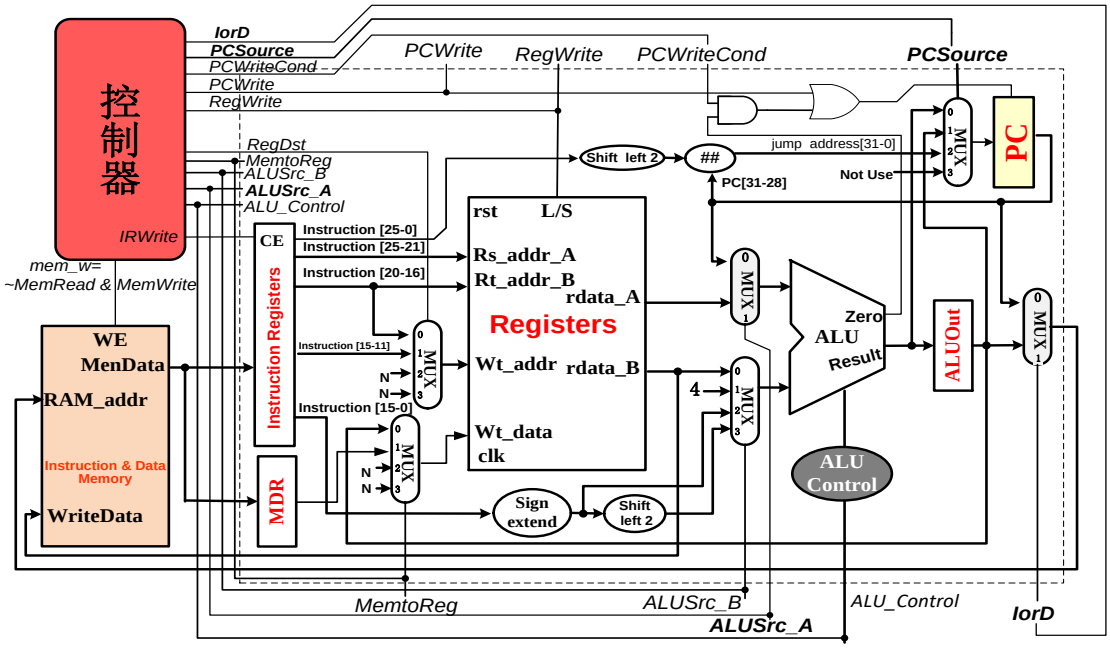
2.2.1 控制器设计方案

- **控制器实现有多种方法**
    - 状态转换实现：
        - 状态表→状态方程→激励方程→HDL 描述
        - 或状态表→ HDL 行为描述
    - 激励方程和输出信号：

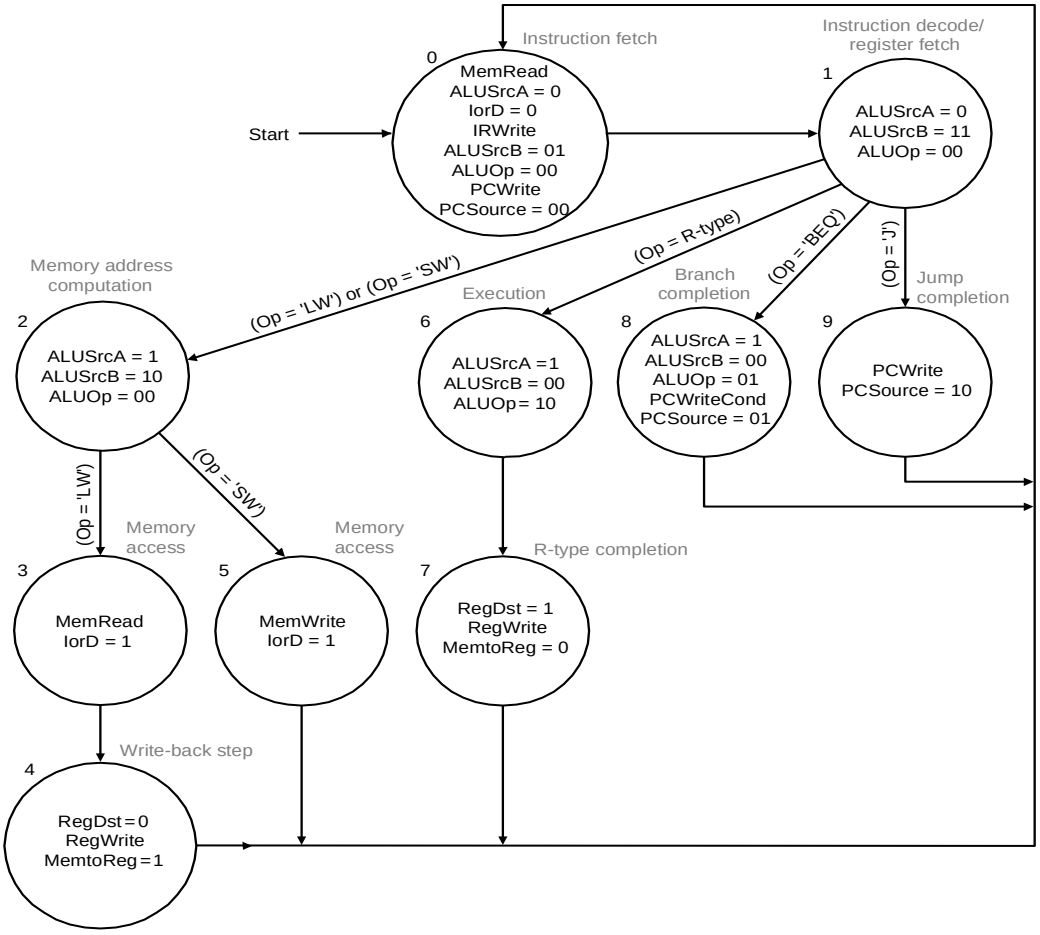
- HDL 直接描述
- ROM/ PLA (教材光盘)
- MUX
- 门电路

- **这里根据时序电路的一般设计流程分析**
  - 实现时可以选用任意一种方法。建议：
    - 9+条指令不是非常复杂，用激励方程 HDL 描述实现
    - 指令较多时用 HDL 直接描述状态表实现

2.2.2 控制器与控制对象



2.2.3 9⁺指令的状态机：根据设计指令画出



2.2.4 状态转换表/次态表

- **根据状态图和输入变量 OP_0~OP_5 写出状态转换表**
  - 4 个状态变量共有 16 个状态，其中 1010~1111 六个状态为非工作状态
  - 操作码有 6 个变量，共 2^6=64 个最小项组合，只有 5 种组合为有效输入，其余为无效输入，可作任意项考虑。

序号	现 态	输 入(指令操作码)	次 态		备注
	Q_{3n} Q_{2n} Q_{1n} Q_{0n}	Op5 Op4 Op3 Op2 Op1 Op0	Q_{3n+1} Q_{2n+1} Q_{1n+1} Q_{0n+1}		
0	0 0 0 0	x x x x x x	0 0 0 1	1	Op无关
1	0 0 0 1	0 0 0 0 0 0	0 1 1 0	6	R-type
		1 0 x 0 1 1	0 0 1 0	2	L/S
		0 0 0 1 0 0	1 0 0 0	8	Beq
		0 0 0 0 1 0	1 0 0 1	9	Jump
2	0 0 1 0	1 0 0 0 1 1	0 0 1 1	3	Load
		1 0 1 0 1 1	0 1 0 1	5	Store
3	0 0 1 1	1 0 0 0 1 1	0 1 0 0	4	Load
4	0 1 0 0	1 0 0 0 1 1	0 0 0 0	0	Load
5	0 1 0 1	1 0 1 0 1 1	0 0 0 0	0	Store
6	0 1 1 0	0 0 0 0 0 0	0 1 1 1	7	R-type
7	0 1 1 1	0 0 0 0 0 0	0 0 0 0	0	R-type
8	1 0 0 0	0 0 0 1 0 0	0 0 0 0	0	Beq
9	1 0 0 1	0 0 0 0 1 0	0 0 0 0	0	Jump

2.2.5 状态方程

- 根据状态转换表可以写出状态方程如下：

$Q_{3n+1} = State1\ (Beq + Jump)$

$Q_{2n+1} = State1\ Rtype + State2\ Store + State3\ Load + State6\ Rtype$

$Q_{1n+1} = State1\ (Rtype + LS) + State2\ Load + State6\ Rtype$

$Q_{0n+1} = State0 + State1\ Jump + State2\ Load + State2\ Store + State6\ Rtype$

$$Q_{3n+1} = \overline{Q}_{3n}\overline{Q}_{2n}\overline{Q}_{1n}Q_{0n}(\overline{O}p_5\overline{O}p_4\overline{O}p_3Op_2\overline{O}p_1\overline{O}p_0 + \overline{O}p_5\overline{O}p_4\overline{O}p_3\overline{O}p_2Op_1\overline{O}p_0)$$

$$Q_{2n+1} = \overline{Q}_{3n}\overline{Q}_{2n}\overline{Q}_{1n}Q_{0n}\overline{O}p_5\overline{O}p_4\overline{O}p_3\overline{O}p_2\overline{O}p_1\overline{O}p_0 + \overline{Q}_{3n}\overline{Q}_{2n}Q_{1n}\overline{Q}_{0n}Op_5\overline{O}p_4Op_3\overline{O}p_2Op_1Op_0 + \overline{Q}_{3n}\overline{Q}_{2n}Q_{1n}Q_{0n}Op_5\overline{O}p_4\overline{O}p_3\overline{O}p_2Op_1Op_0 + \overline{Q}_{3n}Q_{2n}Q_{1n}\overline{Q}_{0n}\overline{O}p_5\overline{O}p_4\overline{O}p_3\overline{O}p_2\overline{O}p_1\overline{O}p_0$$

$$Q_{1n+1} = \overline{Q}_{3n}\overline{Q}_{2n}\overline{Q}_{1n}Q_{0n}(\overline{O}p_5\overline{O}p_4\overline{O}p_3\overline{O}p_2\overline{O}p_1\overline{O}p_0 + Op_5\overline{O}p_4X\overline{O}p_2Op_1Op_0) + \overline{Q}_{3n}\overline{Q}_{2n}Q_{1n}\overline{Q}_{0n}Op_5\overline{O}p_4\overline{O}p_3\overline{O}p_2Op_1Op_0 + \overline{Q}_{3n}\overline{Q}_{2n}Q_{1n}Q_{0n}Op_5\overline{O}p_4\overline{O}p_3\overline{O}p_2\overline{O}p_1\overline{O}p_0$$

$$Q_{0n+1} = \overline{Q}_{3n}\overline{Q}_{2n}\overline{Q}_{1n}\overline{Q}_{0n} + \overline{Q}_{3n}\overline{Q}_{2n}\overline{Q}_{1n}Q_{0n}\overline{O}p_5\overline{O}p_4\overline{O}p_3\overline{O}p_2Op_1\overline{O}p_0 + \overline{Q}_{3n}\overline{Q}_{2n}Q_{1n}\overline{Q}_{0n}Op_5\overline{O}p_4\overline{O}p_3\overline{O}p_2Op_1Op_0 + \overline{Q}_{3n}\overline{Q}_{2n}Q_{1n}\overline{Q}_{0n}Op_5\overline{O}p_4Op_3\overline{O}p_2Op_1Op_0 + \overline{Q}_{3n}Q_{2n}Q_{1n}\overline{Q}_{0n}\overline{O}p_5\overline{O}p_4\overline{O}p_3\overline{O}p_2\overline{O}p_1\overline{O}p_0$$

- Op_4全是“0”可以简化，但意义不大

2.2.6 激励方程

- 根据D触发器特征方程和状态方程可得D触发器激励函数：

$$D_3 = \overline{Q}_{3n}\overline{Q}_{2n}\overline{Q}_{1n}Q_{0n}(\overline{O}p_5\overline{O}p_4\overline{O}p_3Op_2\overline{O}p_1\overline{O}p_0 + \overline{O}p_5\overline{O}p_4\overline{O}p_3\overline{O}p_2Op_1\overline{O}p_0)$$

$$D_2 = \overline{Q}_{3n}\overline{Q}_{2n}\overline{Q}_{1n}Q_{0n}\overline{O}p_5\overline{O}p_4\overline{O}p_3\overline{O}p_2\overline{O}p_1\overline{O}p_0 + \overline{Q}_{3n}\overline{Q}_{2n}Q_{1n}\overline{Q}_{0n}Op_5\overline{O}p_4Op_3\overline{O}p_2Op_1Op_0 + \overline{Q}_{3n}\overline{Q}_{2n}Q_{1n}Q_{0n}Op_5\overline{O}p_4\overline{O}p_3\overline{O}p_2Op_1Op_0 + \overline{Q}_{3n}Q_{2n}Q_{1n}\overline{Q}_{0n}\overline{O}p_5\overline{O}p_4\overline{O}p_3\overline{O}p_2\overline{O}p_1\overline{O}p_0$$

$$D_1 = \overline{Q}_{3n}\overline{Q}_{2n}\overline{Q}_{1n}Q_{0n}(\overline{O}p_5\overline{O}p_4\overline{O}p_3\overline{O}p_2\overline{O}p_1\overline{O}p_0 + Op_5\overline{O}p_4X\overline{O}p_2Op_1Op_0) + \overline{Q}_{3n}\overline{Q}_{2n}Q_{1n}\overline{Q}_{0n}Op_5\overline{O}p_4\overline{O}p_3\overline{O}p_2Op_1Op_0 + \overline{Q}_{3n}\overline{Q}_{2n}Q_{1n}Q_{0n}\overline{O}p_5\overline{O}p_4\overline{O}p_3\overline{O}p_2\overline{O}p_1\overline{O}p_0$$

$$D_0 = \overline{Q}_{3n}\overline{Q}_{2n}\overline{Q}_{1n}\overline{Q}_{0n} + \overline{Q}_{3n}\overline{Q}_{2n}\overline{Q}_{1n}Q_{0n}\overline{O}p_5\overline{O}p_4\overline{O}p_3\overline{O}p_2Op_1\overline{O}p_0 + \overline{Q}_{3n}\overline{Q}_{2n}Q_{1n}\overline{Q}_{0n}Op_5\overline{O}p_4\overline{O}p_3\overline{O}p_2Op_1Op_0 + \overline{Q}_{3n}\overline{Q}_{2n}Q_{1n}\overline{Q}_{0n}Op_5\overline{O}p_4Op_3\overline{O}p_2Op_1Op_0 + \overline{Q}_{3n}Q_{2n}Q_{1n}\overline{Q}_{0n}\overline{O}p_5\overline{O}p_4\overline{O}p_3\overline{O}p_2\overline{O}p_1\overline{O}p_0$$

$D_3 = State1\ (Beq + Jump)$

$D_2 = State1\ Rtype + State2\ Store + State3\ Load + State6\ Rtype$

$D_1 = State1\ (Rtype + LS) + State2\ Load + State6\ Rtype$

$D_0 = State0 + State1\ Jump + State2\ Load + State2\ Store + State6\ Rtype$

- 这一步完成了状态转换的设计
  - 根据D触发器激励方程可以画出状态转换逻辑
  - 现代工程设计已经不需要自己求解
    - EDA综合器会自动综合

2.2.7 状态激励表

- 也可以根据D触发器特征和状态表得到状态激励表
  - 和状态表类同，仅输出是触发器输入D，一般用状态方程配方
  - 输出真值信号太多，需要另列
    - 采用Moore状态机，输出仅与状态有关

序号	现 态 $Q_{3n}\ Q_{2n}\ Q_{1n}\ Q_{0n}$	输 入(指令操作码) Op5 Op4 Op3 Op2 Op1 Op0	D触发器输入 $D_3\ D_2\ D_1\ D_0$	输出另列	备注
0	0 0 0 0	x x x x x x	0 0 0 1		Op无关
1	0 0 0 1	0 0 0 0 0 0	0 1 1 0		R-type
		1 0 x 0 1 1	0 0 1 0		L/S
		0 0 0 1 0 0	1 0 0 0		Beq
		0 0 0 0 1 0	1 0 0 1		Jump
2	0 0 1 0	1 0 0 0 1 1	0 0 1 1		Load
		1 0 1 0 1 1	0 1 0 1		Store
3	0 0 1 1	1 0 0 0 1 1	0 1 0 0		Load
4	0 1 0 0	1 0 0 0 1 1	0 0 0 0		Load
5	0 1 0 1	1 0 1 0 1 1	0 0 0 0		Store
6	0 1 1 0	0 0 0 0 0 0	0 1 1 1		R-type
7	0 1 1 1	0 0 0 0 0 0	0 0 0 0		R-type
8	1 0 0 0	0 0 0 1 0 0	0 0 0 0		Beq
9	1 0 0 1	0 0 0 0 1 0	0 0 0 0		Jump

2.2.8 输出信号真值表(状态激励表另列部分)

- 根据状态图和多周期数据通路控制要求信号真值表如下：

输入$Q_{3n}\ Q_{2n}\ Q_{1n}\ Q_{0n}$（当前状态—现态）										输出控制信号
0000 IF	0001 ID	0010 MEN-Ex	0011 MEN-RD	0100 LW_WB	0101 MEM_W	0110 R_Exc	0111 R_WB	1000 Beq_Exc	1001 J	
1	0	0	0	0	0	0	0	0	1	PCWrite
0	0	0	0	0	0	0	0	1	0	PCWriteCond
0	0	0	1	0	1	0	0	0	0	IorD
1	0	0	1	0	0	0	0	0	0	MemRead
0	0	0	0	0	1	0	0	0	0	MemWrite
1	0	0	0	0	0	0	0	0	0	IRWrite
0	0	0	0	1	0	0	0	0	0	MemtoReg
0	0	0	0	0	0	0	0	0	1	PCSource1
0	0	0	0	0	0	0	0	1	0	PCSource0
0	0	0	0	0	0	1	0	0	0	ALUOp1
0	0	0	0	0	0	0	0	1	0	ALUOp0
0	1	1	0	0	0	0	0	0	0	ALUSrcB1
1	1	0	0	0	0	0	0	0	0	ALUSrcB0
0	0	1	0	0	0	1	0	1	0	ALUSrcA
0	0	0	0	1	0	0	1	0	0	RegWrite
0	0	0	0	0	0	0	1	0	0	RegDst

2.2.9 多周期控制信号定义：

实验十定义的数据通路控制信号

信号	源数目	功能定义	赋值0时动作	赋值1时动作
ALUScrA ALUSrc_B	?	ALU端口A、B输入选择		
RegDst	?	寄存器写地址选择(考虑扩展)		
MemtoReg	?	寄存器写数据选择(考虑扩展)		
IorD	?	新增	请填写信号赋值时对应操作	
PCSource	?	新增		
PCWriteCond	?	新增		
.....	?	新增		
Branch	?	Beq指示(考虑Bne扩展)		
RegWrite	-	寄存器写控制		
MemWrite	-	存储器写控制		
MemRead	-	存储器读控制		
ALU_Control	000- 111	3位ALU操作控制	参考表 Exp04	Exp04

2.2.10 兼容 Exp10 的数据通路完善输出信号真值表

状态 输出信号	0000 IF	0001 ID	0010 MEM-Ex	0011 MEM-RD	0100 LW_WB	0101 MEM_W	0110 R_Exc	0111 R_WB	1000 Beq_Exc	1001 J
PCWrite	1	0	0	0	0	0	0	0	0	1
PCWriteCond	0	0	0	0	0	0	0	0	1	0
IorD	0	0	0	1	0	1	0	0	0	0
MemRead	1	0	0	1	0	0	0	0	0	0
MemWrite	0	0	0	0	0	1	0	0	0	0
IRWrite	1	0	0	0	0	0	0	0	0	0
MemtoReg	00	00	00	00	01	00	00	00	00	00
PCSource1	0	0	0	0	0	0	0	0	0	1
PCSource0	0	0	0	0	0	0	0	0	1	0
ALUSrcA	0	0	1	0	0	0	1	0	1	0
ALUSrcB1	0	1	1	0	0	0	0	0	0	0
ALUSrcB0	1	1	0	0	0	0	0	0	0	0
RegWrite	0	0	0	0	1	0	0	1	0	0
RegDst	00	00	00	00	00	00	00	01	00	00
Branch	0	0	0	0	0	0	0	0	1	0
ALUOp1	0	0	0	0	0	0	1	0	0	0
ALUOp0	0	0	0	0	0	0	0	0	1	0
MEM_IO	0	0	0	1	0	1	0	0	0	0

三、主要仪器设备

1. 计算机（Intel Core i5 以上，4GB 内存以上）系统　　1 套
2. 计算机软硬件课程贯通教学实验系统　　1 套
3. Xilinx ISE14.4 及以上开发工具　　1 套

四、操作方法与实验步骤

4.1 设计工程 OExp11-OwnMCPU

◎ **设计多周期 CPU 之控制器**

- 根据 Exp10 数据通路及指令设计状态图和状态真值表
- 根据状态表完成控制器电路，HDL 描述实现
  - ⊙ 必须根据状态表结构描述或激励方程描述实现
- 仿真测试控制器模块

◎ **集成替换验证后的控制器模块**

- 替换实验十(Exp10)中的 ctrl.ngc 核
- 顶层模块延用 Exp10：模块名：Top_OExp10_OwnMCPU.v

◎ **测试控制器模块**

- 设计测试程序(MIPS 汇编)测试：
- OP 译码测试：
  - ⊙ R-格式、访存指令、分支指令，转移指令
- 运算控制测试：Function 译码测试

4.2 设计要点

- **设计主控制器模块**
  - 完成输出信号真值表
    - 用 HDL 直接描述实现状态转换并输出控制信号
    - 或用激励方程实现状态转换并输出控制信号
- **设计 ALU 操作译码**
  - 分离出单周期的 ALU 译码模块并修改调用
  - 使用 DEMO 作功能初步调试
    - ALU 必须运算包含“nor”操作
    - 否则需要修改或重新设计调试程序
- **仿真主控制器电路模块**
  - 可以单独或合并仿真，但最后要合并为一个控制模块

五、实验结果与分析

5.1 ctrl 实现的两大思路选择

设计多周期的 ctrl.v 是比较复杂的。我看了 PPT 后总结了一下，一共有两种大的方向来实现这个模块：

① 对于每一种状态，用 verilog 语言来直接描述不同输入下状态机之间的转换。

② 无脑列出每一种状态被得到的函数表达式（一定一堆 and 最后 or 起来），并化简。

① 和 ② 在 PPT 中都有一些提示给出，我最终选择了①。因为 ① 虽然代码长，但理解和调试起来很清晰。②虽然代码短，一旦写错后，调起来就是一个噩梦。

5.2 一个小模块：ALU_decoder.v 的实现

```
module ALU_Decoder(
    input [1:0] ALUop,
    input [5:0] Fun,
    output reg [2:0] ALU_Control
);
parameter AND = 3'b000, OR  = 3'b001, ADD = 3'b010, SUB = 3'b110,
            NOR = 3'b100, SLT = 3'b111, XOR = 3'b011, SRL = 3'b101;
always @ * begin
    case(ALUop)
        2'b00: ALU_Control = 3'b010;
        2'b01: ALU_Control = 3'b110;
        2'b10:
                case (Fun[5:0])
                    6'b100000: ALU_Control = ADD;
                    6'b100010: ALU_Control = SUB;
                    6'b100100: ALU_Control = AND;
                    6'b100101: ALU_Control = OR;
                    6'b100111: ALU_Control = NOR;
                    6'b101010: ALU_Control = SLT;
                    6'b000010: ALU_Control = SRL;
                    6'b000000: ALU_Control = XOR;
                    default:   ALU_Control = ADD;
                endcase
        2'b11: ALU_Control = 3'b111;           //slti
    endcase
end
endmodule
```

5.3 multi_ctrl.v 模块实现

在这个模块里，一共分为几个步骤：

① 用 ALU_decoder 翻译出丢给 ALU 的指令。

② 定义每一个状态的二进制编码。

③ 定义每一个状态的控制信号。

④ 定义状态之间的转移。

由于 PPT 里已经给出了所有状态的常数，我只需写好状态之间的转移函数，并把他们连接在一起即可。

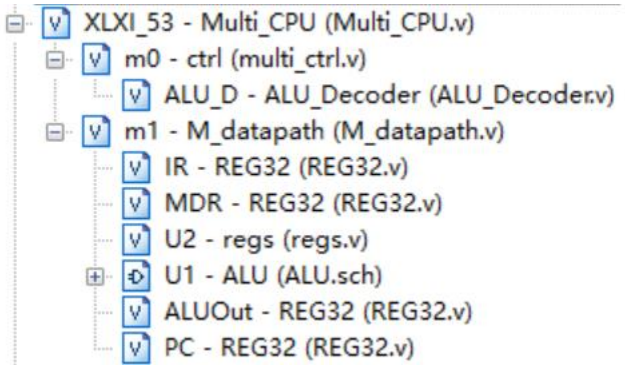
```
module ctrl(input  clk,
                input  reset,
                input  [31:0] Inst_in,
                input  zero,
                input  overflow,
                input  MIO_ready,
                output reg MemRead,
                output reg MemWrite,
                output [2:0]ALU_operation,
                output [4:0]state_out,

                output reg CPU_MIO,
                output reg IorD,
                output reg IRWrite,
                output reg [1:0]RegDst,
                output reg RegWrite,
                output reg [1:0]MemtoReg,
                output reg ALUSrcA,
                output reg [1:0]ALUSrcB,
                output reg [1:0]PCSource,
                output reg PCWrite,
                output reg PCWriteCond,
                output reg Branch
                );
reg [3:0] state;
reg [1:0] ALUop;

ALU_Decoder
ALU_D(.ALUop(ALUop), .Fun(Inst_in[5:0]), .ALU_Control(ALU_operation));

parameter IF = 4'b0000, ID = 4'b0001, Mem_Ex = 4'b0010,
```

```
       Mem_RD = 4'b0011, LW_WB =   4'b0100,  Mem_W = 4'b0101,
       R_Exc = 4'b0110,   R_WB = 4'b0111, Beq_Exc = 4'b1000,
       J = 4'b1001,  Error = 4'b1111;

`define Datapath_signals {PCWrite, PCWriteCond,IorD, MemRead,
MemWrite,IRWrite, MemtoReg, PCSource, ALUSrcA, ALUSrcB, RegWrite,
RegDst, Branch, ALUop, CPU_MIO}

parameter  value0 = 20'b10010100000010000000,
           value1 = 20'b00000000000110000000,
           value2 = 20'b00000000001100000000,
           value3 = 20'b00110000000000000001,
           value4 = 20'b00000001000001000000,
           value5 = 20'b00101000000000000001,
           value6 = 20'b00000000001000000100,
           value7 = 20'b00000000000001010000,
           value8 = 20'b01000000011000001010,
           value9 = 20'b10000000100000000000;

always @ * begin
      case (state)             //state
           IF:        `Datapath_signals = value0;
           ID:    `Datapath_signals = value1;
           Mem_Ex:  `Datapath_signals = value2;
           Mem_RD:  `Datapath_signals = value3;
           LW_WB:  `Datapath_signals = value4;
           Mem_W:  `Datapath_signals = value5;
           R_Exc:  `Datapath_signals = value6;
           R_WB:      `Datapath_signals = value7;
           Beq_Exc: `Datapath_signals = value8;
           J:       `Datapath_signals = value9;
           default: `Datapath_signals = value0;
     endcase
end

always @ (posedge clk or posedge reset)
   if (reset==1) state <= IF;
   else
        case (state)
           IF:     if (MIO_ready) state <= ID; else state <= IF;
           ID:     case (Inst_in[31:26])
                          6'b000000: state <= R_Exc;  //R-type OP
                          6'b100011: state <= Mem_Ex;     //Lw
                          6'b101011: state <= Mem_Ex;   //Sw
```

```
                            6'b000100: state <= Beq_Exc;  //Beq
                            6'b000010: state <= J;        //J
                            default:   state <= Error;
                        endcase
            Mem_Ex:  if (Inst_in[31:26] == 6'b100011) state <= Mem_RD;
else state <= Mem_W;
            Mem_RD:  state <= LW_WB;
            LW_WB:   state <= IF;
            Mem_W:   state <= IF;
            R_Exc:   state <= R_WB;
            R_WB:    state <= IF;
            Beq_Exc: state <= IF;
            J:       state <= IF;
            Error:   state <= Error;
            default: state <= Error;
      endcase
endmodule
```

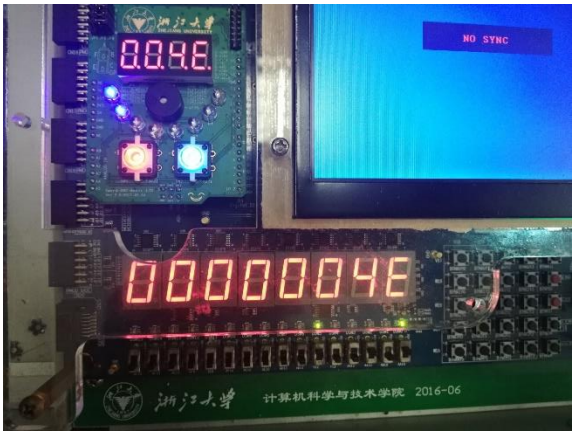
5.4 最终整个 CPU 的架构

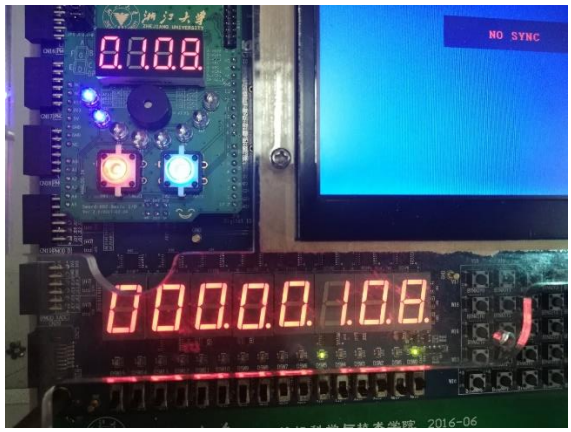
到实验十一位置，我们完整地把多周期的 CPU 给搭出来了。
以下是顶层框架。



5.4 实验现象一

SW[0]=1, SW[2]=1，SW[7:5]=111。
输出 CPU 指令字节地址 PC_out。

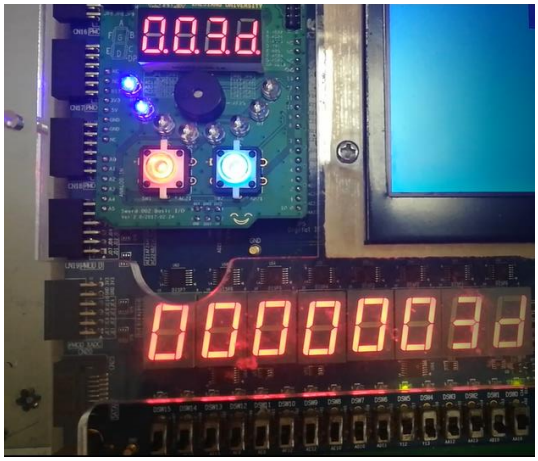


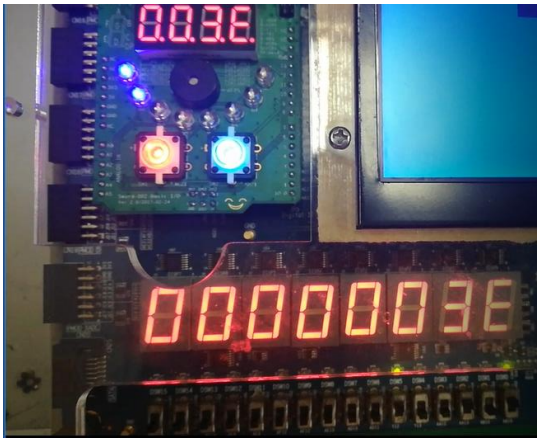


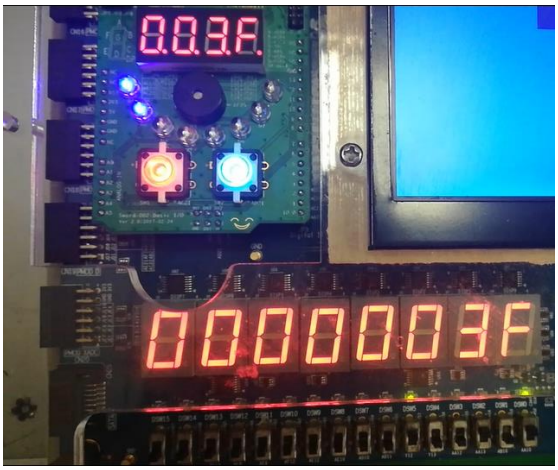
实验现象 1

5.5 实验现象二

SW[0]=1, SW[2]=1，SW[7:5]=001。
输出 CPU 指令字地址 PC_out[31:2]。



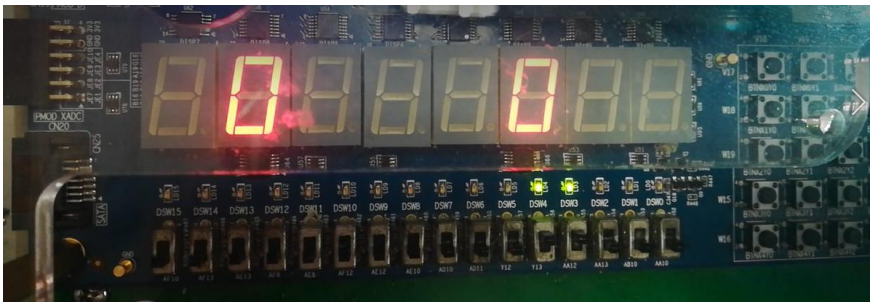


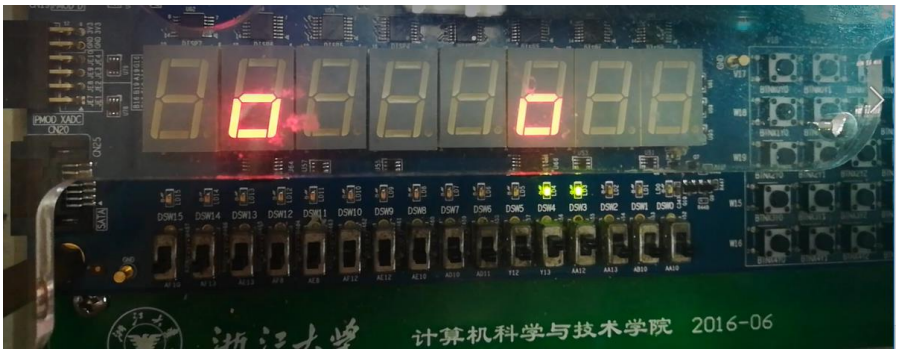


□ 实验现象 2

5.6 实验现象三

SW[0]=0, SW[3]=1，SW[4]=1。
矩形框跳舞。

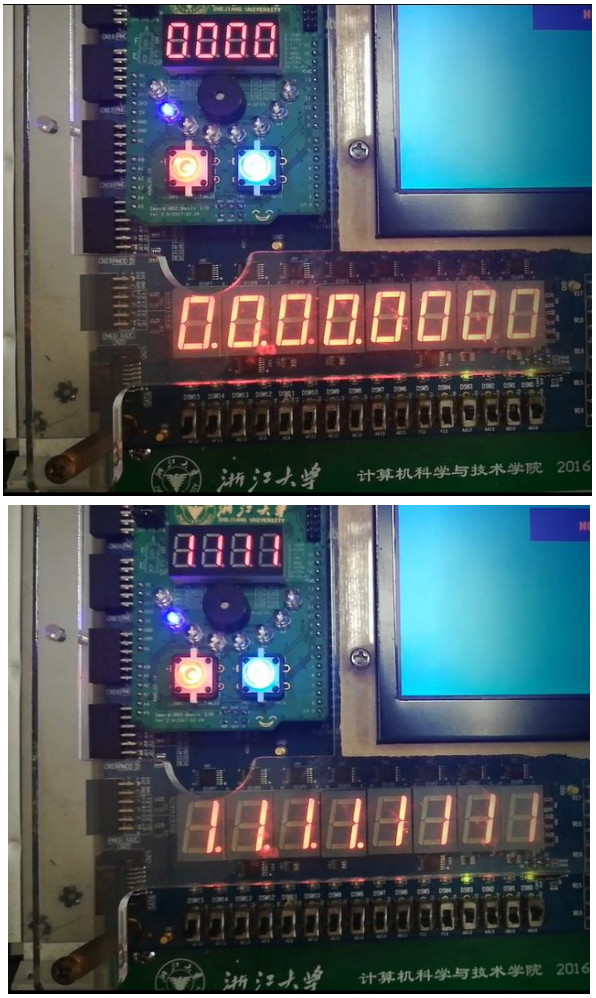




□ 实验现象 3

5.7 实验现象四

SW[0]=1, SW[3]=1。
全速时钟。



- 实验现象 4

六、讨论、心得

实验 9 连了 MCPU 顶层模块，实验 10 拆解了 M_datapath，现在终于把 multictrl.v 也拆解了。这次的 SCPU_ctrl 比上次的 Data_path 要好玩多了。因为上次主要死板地连线，而这次是我们自己来设计一下状态机的转移和 ctrl.v 最终的设计。

不得不说，本次实验也是三次实验里难度最大的。因为在多周期 CPU 里引入了状态机的概念。显然不同状态的控制信号不一样。我们不仅要对不同状态的控制信号进行预设，还要设计状态之间的转移，稍有不慎，跑马灯就跑不起来了。

不过经历了本学期的实验，我对 CPU 架构和 verilog 语言有了更深一步的认识。所以这次实验我刚写完，就有惊无险地跑出想要的图案了。