



《计算机组成原理实验》 实验报告

(实验三)

学 院 名 称 : 数据科学与计算机学院

专业 (班级) : 18 软件工程 2 班

时 间 : 2019 年 月 日

组 员 : 张楚明 18342125

成绩：

实验三：多周期 CPU 设计与实现

一、实验目的

- (1) 认识和掌握多周期数据通路图的构成、原理及其设计方法；
- (2) 掌握多周期 CPU 的实现方法，代码实现方法；
- (3) 编写一个编译器，将 MIPS 汇编程序编译为二进制机器码；
- (4) 掌握多周期 CPU 的测试方法；
- (5) 掌握多周期 CPU 的实现方法。

二、实验内容

设计一个多周期 CPU，该 CPU 至少能实现以下指令功能操作。需设计的指令与格式如下：

==>算术运算指令

(1) add rd, rs, rt

000000	rs(5 位)	rt(5 位)	rd(5 位)	reserved
--------	---------	---------	---------	----------

功能：rd ← rs + rt。

(2) sub rd, rs, rt

000001	rs(5 位)	rt(5 位)	rd(5 位)	reserved
--------	---------	---------	---------	----------

完成功能：rd ← rs - rt。

(3) addiu rt, rs, immediate

000010	rs(5 位)	rt(5 位)	immediate(16 位)
--------	---------	---------	-----------------

功能：rt ← rs + (sign-extend)immediate。

==>逻辑运算指令

(4) and rd, rs, rt

010000	rs(5 位)	rt(5 位)	rd(5 位)	reserved
--------	---------	---------	---------	----------

功能：rd ← rs & rt；逻辑与运算。

(5) andi rt, rs, immediate

010001	rs(5 位)	rt(5 位)	immediate(16 位)
--------	---------	---------	-----------------

功能：rt ← rs & (zero-extend)immediate；immediate 做“0”扩展再参加“与”运算。

(6) ori rt, rs, immediate

010010	rs(5 位)	rt(5 位)	immediate(16 位)
--------	---------	---------	-----------------

功能：rt ← rs | (zero-extend)immediate；immediate 做“0”扩展再参加“或”运算。

(7) xori rt, rs, immediate

010011	rs(5 位)	rt(5 位)	immediate(16 位)
--------	---------	---------	-----------------

功能：rt ← rs ⊕ (zero-extend)immediate；immediate 做“0”扩展再参加“异或”运算。

==>移位指令

(8) sll rd, rt, sa

011000	未用	rt(5 位)	rd(5 位)	sa	reserved
--------	----	---------	---------	----	----------

功能: $rd \leftarrow rt \ll (\text{zero-extend})sa$, 左移 sa 位, $(\text{zero-extend})sa$ 。

==>比较指令

(9) slti rt, rs, immediate 带符号

100110	rs(5 位)	rt(5 位)	immediate(16 位)
--------	---------	---------	-----------------

功能: if (rs < (sign-extend)immediate) rt = 1 else rt = 0, 具体请看表 2 ALU 运算功能表, 带符号。

(10) slt rd, rs, rt 带符号

100111	rs(5 位)	rt(5 位)	rd(5 位)	reserved
--------	---------	---------	---------	----------

功能: if (rs < rt) rd = 1 else rd = 0, 具体请看表 2 ALU 运算功能表, 带符号。

==>存储器读写指令

(11) sw rt, immediate(rs)

110000	rs(5 位)	rt(5 位)	immediate(16 位)
--------	---------	---------	-----------------

功能: $\text{memory}[rs + (\text{sign-extend})immediate] \leftarrow rt$ 。即将 rt 寄存器的内容保存到 rs 寄存器内容和立即数符号扩展后的数相加作为地址的内存单元中。

(12) lw rt, immediate(rs)

110001	rs(5 位)	rt(5 位)	immediate(16 位)
--------	---------	---------	-----------------

功能: $rt \leftarrow \text{memory}[rs + (\text{sign-extend})immediate]$ 。即读取 rs 寄存器内容和立即数符号扩展后的数相加作为地址的内存单元中的数, 然后保存到 rt 寄存器中。

==>分支指令

(13) beq rs, rt, immediate (说明: immediate 从 pc+4 开始和转移到的指令之间间隔条数)

110100	rs(5 位)	rt(5 位)	immediate(16 位)
--------	---------	---------	-----------------

功能: if (rs = rt) $pc \leftarrow pc + 4 + ((\text{sign-extend})immediate \ll 2)$ else $pc \leftarrow pc + 4$ 。

(14) bne rs, rt, immediate (说明: immediate 从 pc+4 开始和转移到的指令之间间隔条数)

110101	rs(5 位)	rt(5 位)	immediate(16 位)
--------	---------	---------	-----------------

功能: if (rs != rt) $pc \leftarrow pc + 4 + ((\text{sign-extend})immediate \ll 2)$ else $pc \leftarrow pc + 4$ 。

(15) bltz rs, immediate

110110	rs(5 位)	00000	immediate
--------	---------	-------	-----------

功能: if (rs < \$0) $pc \leftarrow pc + 4 + ((\text{sign-extend})immediate \ll 2)$ else $pc \leftarrow pc + 4$ 。

==>跳转指令

(16) j addr

111000	addr[27:2]				
--------	------------	--	--	--	--

功能: $pc \leftarrow \{(pc+4)[31:28], \text{addr}[27:2], 2'b00\}$, 跳转。

说明：由于 MIPS32 的指令代码长度占 4 个字节，所以指令地址二进制数最低 2 位均为 0，将指令地址放进指令代码中时，可省掉！这样，除了最高 6 位操作码外，还有 26 位可用于存放地址，事实上，可存放 28 位地址，剩下最高 4 位由 pc+4 最高 4 位拼接上。

(17) jr rs

111001	rs(5 位)	未用	未用	reserved
--------	---------	----	----	----------

功能：pc ← rs，跳转。

==>调用子程序指令

(18) jal addr

111010	addr[27:2]
--------	------------

功能：调用子程序，pc ← {(pc+4)[31:28],addr[27:2],2'b00}; \$31←pc+4，返回地址设置；子程序返回，需用指令 jr \$31。跳转地址的形成同 j addr 指令。

==>停机指令

(19) halt (停机指令)

111111	0000000000000000000000000000(26 位)
--------	------------------------------------

不改变 pc 的值，pc 保持不变。

三、实验原理

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：

- (1) 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。
- (2) 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。
- (3) 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。
- (4) 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。
- (5) 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。

实验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，但具体情况怎样？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。



图 1 多周期 CPU 指令处理过程

MIPS 指令的三种格式:

R 类型:

31	26 25	21 20	16 15	11 10	6 5	0
op	rs	rt	rd	sa	funct	
6 位	5 位	5 位	5 位	5 位	6 位	

I 类型:

31	26 25	21 20	16 15	0
op	rs	rt	immediate	
6 位	5 位	5 位	16 位	

J 类型:

31	26 25	0
op	address	
6 位	26 位	

其中,

op: 为操作码;

rs: 为第 1 个源操作数寄存器, 寄存器地址 (编号) 是 00000~11111, 00~1F;

rt: 为第 2 个源操作数寄存器, 或目的操作数寄存器, 寄存器地址 (同上);

rd: 为目的操作数寄存器, 寄存器地址 (同上);

sa: 为位移量 (shift amt), 移位指令用于指定移多少位;

funct: 为功能码, 在寄存器类型指令中 (R 类型) 用来指定指令的功能;

immediate: 为 16 位立即数, 用作无符号的逻辑操作数、有符号的算术操作数、数据加载 (Load) / 数据保存 (Store) 指令的数据地址字节偏移量和分支指令中相对程序计数器 (PC) 的有符号偏移量;

address: 为地址。

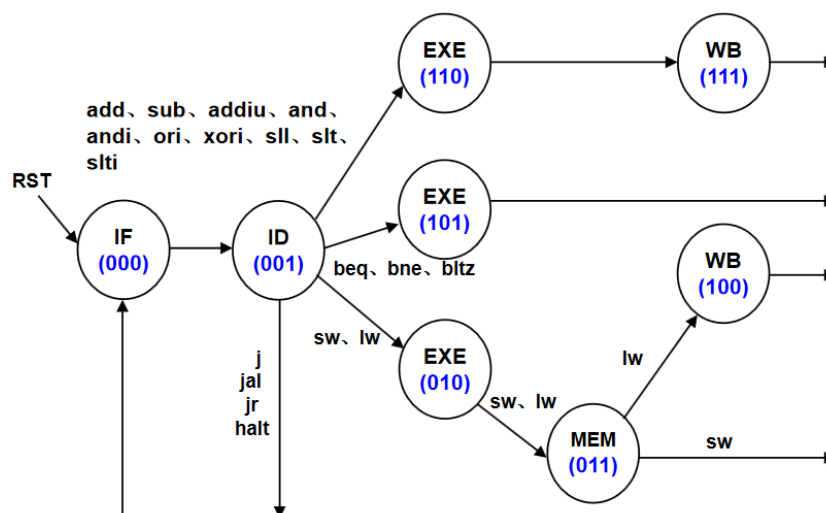


图 2 多周期 CPU 状态转移图

状态的转移有的是无条件的, 例如从 sIF 状态转移到 sID 就是无条件的; 有些是有条件的, 例如 sEXE 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。每个状态代表一个时钟周期。

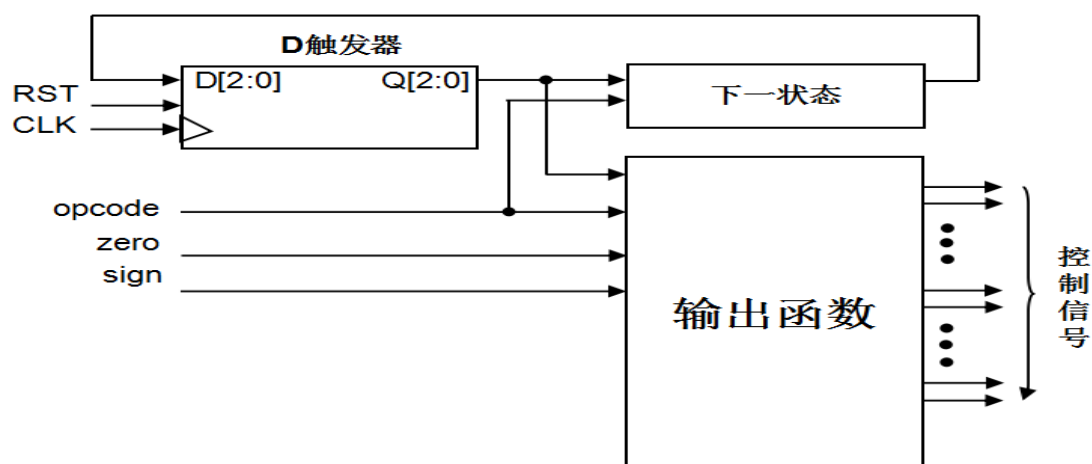


图 3 多周期 CPU 控制部件的原理结构图

图 3 是多周期 CPU 控制部件的电路结构，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。

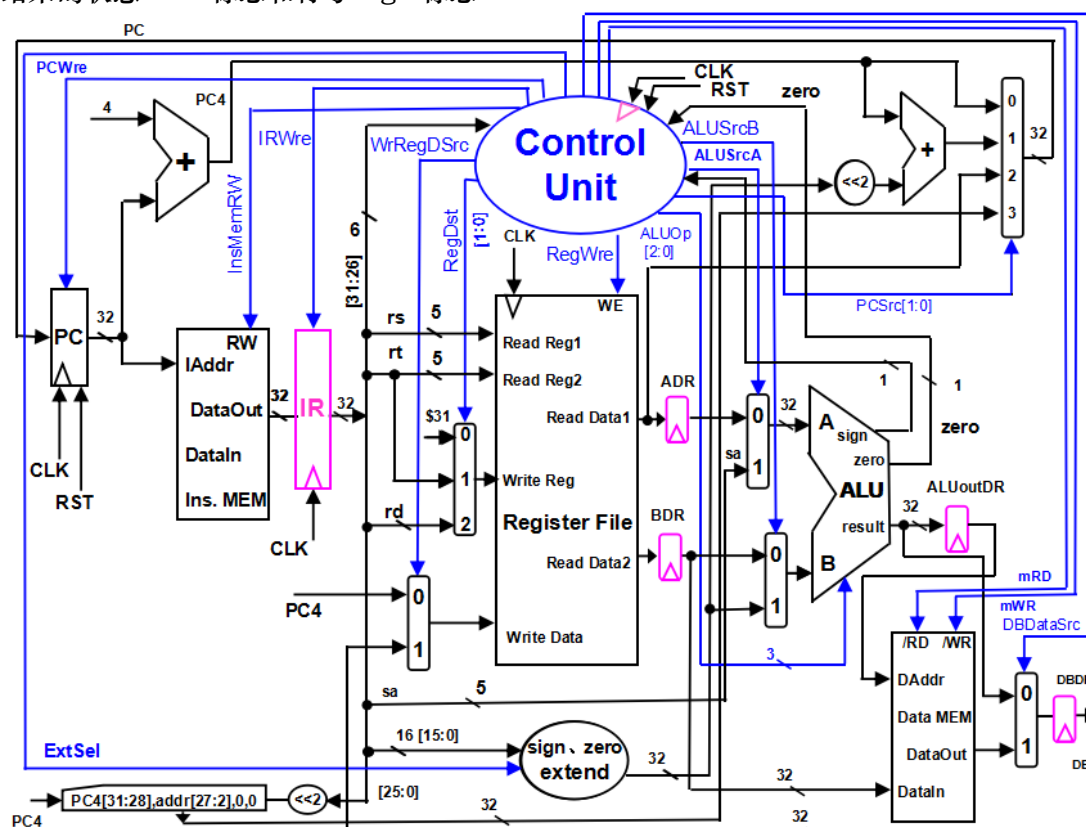


图 4 多周期 CPU 数据通路和控制线路图

图 4 是一个简单的基本上能够在多周期 CPU 上完成所要求设计的指令功能的数据通路和必要的控制线路图。其中指令和数据各存储在不同存储器中，即有指令存储器和数据存储器。访问存储器时，先给出内存地址，然后由读或写信号控制操作。对于寄存器组，给出寄

寄存器地址（编号），读操作时不需要时钟信号，输出端就直接输出相应数据；而在写操作时，在 WE 使能信号为 1 时，在时钟边沿触发将数据写入寄存器。图中控制信号功能如表 1 所示，表 2 是 ALU 运算功能表。

特别提示，图上增加 IR 指令寄存器，目的是使指令代码保持稳定，pc 写使能控制信号 PCWre，是确保 pc 适时修改，原因都是和多周期工作的 CPU 有关。ADR、BDR、ALUoutDR、DBDR 四个寄存器不需要写使能信号，其作用是切分数据通路，将大组合逻辑切分为若干小组合逻辑，大延迟变为多个分段小延迟。

表 1 控制信号作用

控制信号名	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, 另外, 除 ‘000’ 状态之外, 其余状态慎改 PC 的值。	PC 更改, 相关指令: 除指令 halt 外, 另外, 在 ‘000’ 状态时, 修改 PC 的值合适。
ALUSrcA	来自寄存器堆 data1 输出, 相关指令: add、sub、addiu、and、andi、ori、xori、slt、slti、sw、lw、beq、bne、bltz	来自移位数 sa, 同时, 进行 (zero-extend)sa, 即 $\{27\{1'b0\}, sa\}$, 相关指令: sll
ALUSrcB	来自寄存器堆 data2 输出, 相关指令: add、sub、and、slt、sll、beq、bne、bltz	来自 sign 或 zero 扩展的立即数, 相关指令: addiu、andi、ori、xori、slti、lw、sw
DBDataSrc	来自 ALU 运算结果的输出, 相关指令: add、sub、addiu、and、andi、ori、xori、sll、slt、slti	来自数据存储器 (Data MEM) 的输出, 相关指令: lw
RegWre	无写寄存器组寄存器, 相关指令: beq、bne、bltz、j、sw、jr、halt	寄存器组寄存器写使能, 相关指令: add、sub、addiu、and、andi、ori、xori、sll、slt、slti、lw、jal
WrRegDSrc	写入寄存器组寄存器的数据来自 pc+4(pc4), 相关指令: jal, 写 \$31	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, 相关指令: add、addiu、sub、and、andi、ori、xori、sll、slt、slti、lw
InsMemRW	写指令存储器	读指令存储器(Ins. Data)
mRD	存储器输出高阻态	读数据存储器, 相关指令: lw
mWR	无操作	写数据存储器, 相关指令: sw
IRWre	IR(指令寄存器)不更改	IR 寄存器写使能。向指令存储器发出读指令代码后, 这个信号也接着发出, 在时钟上升沿, IR 接收从指令存储器送来的指令代码。与每条指令都相关。
ExtSel	(zero-extend)immediate, 相关指令: andi、xori、ori;	(sign-extend)immediate, 相关指令: addiu、slti、lw、sw、beq、bne、bltz;
PCSrc[1..0]	00: $pc < -pc+4$, 相关指令: add、addiu、sub、and、andi、ori、xori、slt、slti、sll、sw、lw、beq(zero=0)、bne(zero=1)、bltz(sign=0);	

	01: $pc \leftarrow -pc + 4 + (\text{sign-extend})\text{immediate} \times 4$, 相关指令: beq(zero=1)、bne(zero=0)、bltz(sign=1); 10: $pc \leftarrow -rs$, 相关指令: jr; 11: $pc \leftarrow \{pc[31:28], \text{addr}[27:2], 2'b00\}$, 相关指令: j、jal;
RegDst[1..0]	写寄存器组寄存器的地址, 来自: 00: 0x1F(\$31), 相关指令: jal, 用于保存返回地址 ($\$31 \leftarrow -pc + 4$); 01: rt 字段, 相关指令: addiu、andi、ori、xori、slti、lw; 10: rd 字段, 相关指令: add、sub、and、slt、sll; 11: 未用;
ALUOp[2..0]	ALU 8 种运算功能选择(000-111), 看功能表

相关部件及引脚说明:

Instruction Memory: 指令存储器

- Iaddr, 指令地址输入端口
- DataIn, 存储器数据输入端口
- DataOut, 存储器数据输出端口
- RW, 指令存储器读写控制信号, 为 0 写, 为 1 读

Data Memory: 数据存储器

- Daddr, 数据地址输入端口
- DataIn, 存储器数据输入端口
- DataOut, 存储器数据输出端口
- /RD, 数据存储器读控制信号, 为 0 读
- /WR, 数据存储器写控制信号, 为 0 写

Register File: 寄存器组

- Read Reg1, rs 寄存器地址输入端口
- Read Reg2, rt 寄存器地址输入端口
- Write Reg, 将数据写入的寄存器, 其地址输入端口 (rt、rd)
- Write Data, 写入寄存器的数据输入端口
- Read Data1, rs 寄存器数据输出端口
- Read Data2, rt 寄存器数据输出端口
- WE, 写使能信号, 为 1 时, 在时钟边沿触发写入

IR: 指令寄存器, 用于存放正在执行的指令代码

ALU: 算术逻辑单元

- result, ALU 运算结果
- zero, 运算结果标志, 结果为 0, 则 zero=1; 否则 zero=0
- sign, 运算结果标志, 结果最高位为 0, 则 sign=0, 正数; 否则, sign=1, 负数

表 2 ALU 运算功能表

ALUOp[2..0]	功能	描述
000	$Y = A + B$	加
001	$Y = A - B$	减
010	$Y = B \ll A$	B 左移 A 位
011	$Y = A \vee B$	或
100	$Y = A \wedge B$	与

101	$Y = (A < B) ? 1 : 0$	比较 $A < B$ 不带符号
110	$Y = (((A < B) \& \& (A[31] == B[31])) \vee ((A[31] == 1 \& \& B[31] == 0))) ? 1 : 0$	比较 $A < B$ 带符号
111	$Y = A \oplus B$	异或

值得注意的问题，设计时，用模块化、层次化的思想方法设计，关于如何划分模块、如何整合成一个系统等等，是必须认真考虑的问题。

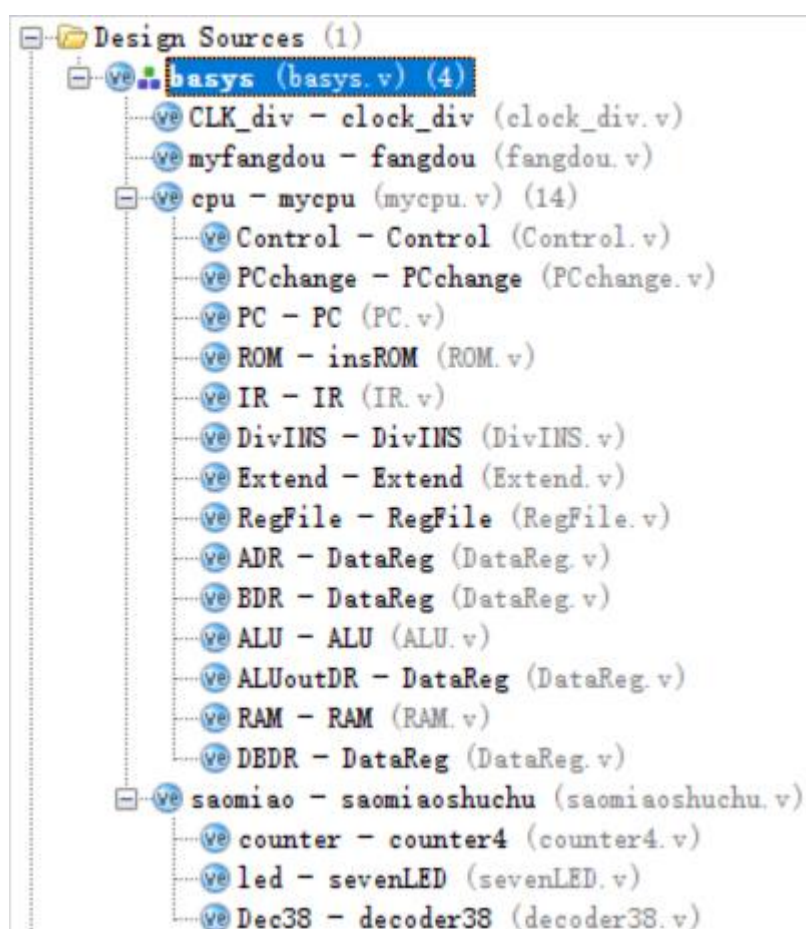
四、实验器材

电脑一台，Xilinx Vivado 软件一套，Basys3板一块。

五、实验过程与结果

1. CPU设计的思想方法（流程图和控制信号表见第三部分：实验原理）

(1) 大致思路同单周期CPU，分模块设计。分为PC，寄存器，寄存器堆，ALU，控制单元，RAM，ROM，扩展等模块，最后构建顶层模块将底层模块串联封装。具体模块如下：



(2) 主要模块代码如下

①分解指令模块：由单周期设计的经验可得，先将指令分解方便后来的设计与数据处理

```
module DivINS(  
    input [31:0] instruction,  
    output reg[5:0] op,  
    output reg[4:0] rs,  
    output reg[4:0] rt,  
    output reg[4:0] rd,  
    output reg[4:0] sa,  
    output reg[15:0] immediate,  
    output reg[25:0] addr  
);  
  
    initial begin  
        op = 5'b00000;  
        rs = 5'b00000;  
        rt = 5'b00000;  
        rd = 5'b00000;  
    end  
  
    always@(instruction)  
    begin  
        op = instruction[31:26];  
        rs = instruction[25:21];  
        rt = instruction[20:16];  
        rd = instruction[15:11];  
        sa = instruction[10:6];  
        immediate = instruction[15:0];  
        addr = instruction[25:0];  
    end  
endmodule
```

②PC 转变模块：由单周期经验可得，PC 转换为测试时的重点，也是 TA 检查的重点。将 4 种 PC 转变都写到一个模块，更加方便模块化与理解代码

```
module PCchange(
    input Reset,
    input [1:0] PCSrc,
    input [31:0] IMM,
    input [25:0] addr,
    input [31:0] curPC,
    input [31:0] rs,
    output reg[31:0] nextPC
);

initial begin
    nextPC = 0;
end

reg [31:0] pc;

always@(*)
begin
    if(!Reset) begin
        nextPC = 0;
    end
    else begin
        pc = curPC + 4;
        case(PCSrc)
            2'b00: nextPC = curPC + 4; //add 4
            2'b01: nextPC = curPC + 4 + IMM * 4; //beq, bne
            2'b10: nextPC = rs; //jalr
            2'b11: nextPC = {pc[31:28], addr, 2'b00}; //j
        endcase
    end
end
```

③寄存器模块（上升沿触发）

```
module DataReg(
    input CLK,
    input [31:0] In,
    output reg[31:0] Out
);

initial begin
    Out = 0;
end

always@(posedge CLK) begin
    Out <= In;
end

endmodule
```

④IR 模块：主要是使指令保持稳定执行，采用上升沿触发，是为了区别 PC 写的下降沿。

```
module IR(
    input [31:0] instruction,
    input CLK,
    input IRWe,
    output reg[31:0] IRInstruction
);
initial begin
    IRInstruction = 0;
end
always@(posedge CLK)
begin
    if(IRWe) begin
        IRInstruction <= instruction;
    end
end
endmodule
```

⑤PC 模块：上升沿触发，记录 curPC 和 nextPC，主要用于 PC 的控制

```
module PC(
    input CLK,
    input Reset,
    input PCWe,
    input [31:0] nextPC,
    output reg[31:0] curPC
);

initial begin
    curPC = 32'b0;
end

always@(posedge CLK or negedge Reset)
begin
    if(!Reset)
    begin
        curPC <= 0;
    end
    else
    begin
        if(PCWe)
        begin
            curPC = nextPC;
        end
        else
        begin
            curPC = curPC; //停机
        end
    end
end
```

⑥ALU 模块：用于实现 ALU 逻辑算数运算，同单周期 CPU 类似

```

module ALU(
    input ALUSrcA,
    input ALUSrcB,
    input [31:0] ReadData1,
    input [31:0] ReadData2,
    input [4:0] sa,
    input [31:0] extend,
    input [2:0] ALUOp,
    output reg zero,
    output reg[31:0] result
);

reg [31:0] A;
reg [31:0] B;

initial begin
    result = 0;
    zero = 0;
end

always@(*)
begin
    A = (ALUSrcA == 0) ? ReadData1 : sa;
    B = (ALUSrcB == 0) ? ReadData2 : extend;

    case(ALUOp)
        3'b000: result = A + B;
        3'b001: result = A - B;
        3'b010: result = B << A;
        3'b011: result = A | B;
        3'b100: result = A & B;
        3'b101: result = (A < B) ? 1 : 0;
        3'b110: result = (((A < B) && (A[31] == B[31])) || ((A[31] == 1 && B[31] == 0))) ? 1 : 0;
        3'b111: result = A ^ B;
    endcase
    zero = (result == 0) ? 1 : 0;
end
endmodule

```

⑦Control 模块：最重要底层模块，由不同状态给出对应的不同的控制信号

```

module Control(
    input CLK,
    input Reset,
    input zero,
    input [5:0] op,
    output reg IRWr,
    output reg PCWr,
    output reg ExtSel,
    output reg InsMemRW,
    output reg WrRegDSrc,
    output reg [1:0] RegDst,
    output reg RegWr,
    output reg ALUSrcA,
    output reg ALUSrcB,
    output reg [1:0] PCSrc,
    output reg [2:0] ALUOp,
    output reg mRD,
    output reg mWR,
    output reg DBDataSrc
);

reg [2:0] state, nextState;
parameter [2:0] iniState = 3'b111,
    sIF = 3'b000,
    sID = 3'b001,
    sEXE = 3'b010,
    sMEM = 3'b100,
    sWB = 3'b011;

initial begin
    state = iniState;
    PCWr = 0;
    InsMemRW = 0;
    IRWr = 0;
    RegWr = 0; ;
    ExtSel = 0;
    PCSrc = 2'b00;
    RegDst = 2'b11;
    ALUOp = 0;
    ExtSel = 0;
    WrRegDSrc = 0;
    ALUSrcA = 0;
    ALUSrcB = 0;
    DBDataSrc = 0;
    mRD = 0;
    mWR = 0;
end

always@(posedge CLK) begin
    if(!Reset) begin
        state <= sIF;
    end else begin
        state <= nextState;
    end
end

always@(state or op or zero) begin
    case(state)
        iniState : nextState = sIF;
        sIF: nextState = sID;
        sID: begin
            case(op[5:3])
                3'b111: nextState = sIF;
                default: nextState = sEXE;
            endcase
        end
        sEXE: begin
            if((op == 6'b110100) || (op == 6'b110101) || (op == 6'b110110)) begin
                nextState = sIF;
            end else if (op == 6'b110000 || op == 6'b110001) begin
                //sw, lw
                nextState = sMEM;
            end else begin
                nextState = sWB;
            end
        end
        sMEM: begin
            if(op == 6'b110000) begin
                //sw
                nextState = sIF;
            end else begin
                //lw
                nextState = sWB;
            end
        end
        sWB: nextState = sIF;
    endcase
end

```

A. B.

C.

D.

```

if(nextState == sIF && op != 6'b111111 && state != iniState)
begin
    PCWr = 1;
    InsMemRW = 1;
end else begin
    PCWr = 0;
    InsMemRW = 0;
end

if(state == sIF || nextState == sID) begin
    IRWr = 1;
end else begin
    IRWr = 0;
end

if(op == 6'b011000)
begin
    ALUSrcA = 1;
end
else begin
    ALUSrcA = 0;
end

if(op == 6'b000010 || op == 6'b010001 || op == 6'b010010 || op == 6'b010011 || op == 6'b110000 || op == 6'b110001 || op == 6'b100110)
begin
    ALUSrcB = 1;
end else
begin

```

E.

```

    ALUSrcB = 0;
end

if(op == 6'b110001)
begin
    DEDataSrc = 1;
end
else begin
    DEDataSrc = 0;
end

if((state == sWB && op != 6'b110100 && op != 6'b110101 && op != 6'b110000 && op != 6'b110110) || (op == 6'b110101 && state == sID))
begin
    RegWr = 1;
    if(op == 6'b110101)
begin
        WrRegDsrc = 0;
        RegDst = 2'b00;
end
else begin
        WrRegDsrc = 1;
        if(op == 6'b000010 || op == 6'b010001 || op == 6'b010010 || op == 6'b010011 || op == 6'b100110 || op == 6'b110001)
begin
            RegDst = 2'b01;
end
else begin
            RegDst = 2'b10;
end
end
end

```

F.

```

end else begin
    RegWrite = 0;
end

if(op != 6'b111111)
    InsMemRW = 1;
mRD = (op == 6'b110001) ? 1 : 0;

mWR = (state == sMEM && op == 6'b110000) ? 1 : 0;

ExtSel = (op == 6'b000010 || op == 6'b110001 || op == 6'b110101 || op == 6'b110000 || op == 6'b110100 || op == 6'b110110) ? 1 : 0;

if(op == 6'b111001)
    begin
        PCSrc = 2'b10;
    end
else if((op == 6'b110100 && zero==1) || (op == 6'b110101 && zero==0) || (op == 6'b110110 && !zero))
    begin
        PCSrc = 2'b01;
    end
else if(op == 6'b111010 || op == 6'b111000)
    begin
        PCSrc = 2'b11;
    end
else begin
        PCSrc = 2'b00;
    end
end

```

G.

```

case(op)
    6'b000000: ALUOp = 3'b000; //add
    6'b000010: ALUOp = 3'b000; // addi
    6'b010011: ALUOp = 3'b111; // xori
    6'b010010: ALUOp = 3'b011; // ori

    6'b010000: ALUOp = 3'b100; // and
    6'b000001: ALUOp = 3'b001; // sub
    6'b010001: ALUOp = 3'b100; // andi
    6'b011000: ALUOp = 3'b010; // sll
    6'b110100: ALUOp = 3'b001; // beq
    6'b110101: ALUOp = 3'b001; // bne
    6'b100111: ALUOp = 3'b110; // slt
    6'b100110: ALUOp = 3'b110; // slti
    6'b110110: ALUOp = 3'b001; // bltz
    6'b110000: ALUOp = 3'b000; //sw
    6'b110001: ALUOp = 3'b000; //lw

endcase
end
endmodule

```


(3) 测试程序段

地址 (周期数)	汇编程序	指令代码					16 进制数代码	
		op(6)	rs(5)	rt(5)	rd(5)/immediate (16)			
0x00000000 (4)	addiu \$1,\$0,8	000010	00000	00001	0000 0000 0000 1000	=	08010008	
0x00000004 (4)	ori \$2,\$0,2	010010	00000	00010	0000 0000 0000 0010	=	48020002	
0x00000008 (4)	xori \$3,\$2,8	010011	00010	00011	0000 0000 0000 1000	=	4C430008	
0x0000000C (4)	sub \$4,\$3,\$1	000001	00011	00001	00100 000 0000 0000	=	04612000	
0x00000010 (4)	and \$5,\$4,\$2	010000	00100	00010	00101 000 0000 0000	=	40822800	
0x00000014 (4)	sll \$5,\$5,2	011000	00000	00101	00101 00010 00 0000	=	60052880	
0x00000018 (2)	beq \$5,\$1,-2(=,转 14)	110100	00101	00001	1111 1111 1111 1110	=	D0A1FFFE	
0x0000001C (1)	jal 0x0000050	111010	00000	00000	0000 0000 0101 0000	=	E8000050	
0x00000020 (4)	slt \$8,\$13,\$1	100111	01101	00001	01000 000 0000 0000	=	9DA14000	
0x00000024 (4)	addiu \$14,\$0,-2	000010	00000	01110	1111 1111 1111 1110	=	080EFFFF	
0x00000028 (4)	slt \$9,\$8,\$14	100111	01000	01110	01001 000 0000 0000	=	9D0E4800	
0x0000002C (4)	slti \$10,\$9,2	100110	01001	01010	0000 0000 0000 0010	=	992A0002	
0x00000030 (4)	slti \$11,\$10,0	100110	01010	01011	0000 0000 0000 0000	=	994B0000	
0x00000034 (4)	add \$11,\$11,\$10	000000	01011	01010	01011 000 0000 0000	=	016A5800	
0x00000038 (2)	bne \$11,\$2,-2(≠,转 34)	110101	01011	00010	1111 1111 1111 1110	=	D562FFFE	
0x0000003C (4)	addiu \$12,\$0,-2	000010	00000	01100	1111 1111 1111 1110	=	080CFFFF	
0x00000040 (4)	addiu \$12,\$12,1	000010	01100	01100	0000 0000 0000 0001	=	098C0001	
0x00000044 (2)	bltz \$12,-2(<0,转 40)	110110	01100	00000	1111 1111 1111 1110	=	D980FFFE	
0x00000048 (4)	andi \$12,\$2,2	010001	00010	01100	0000 0000 0000 0010	=	444C0002	
0x0000004C (1)	j 0x000005C	111000	00000	00000	0000 0000 0101 1100	=	E000005C	
0x00000050 (4)	sw \$2,4(\$1)	110000	00001	00010	0000 0000 0000 0100	=	C0220004	
0x00000054 (5)	lw \$13,4(\$1)	110001	00001	01101	0000 0000 0000 0100	=	C42D0004	
0x00000058 (1)	jr \$31	111001	11111	00000	0000 0000 0000 0000	=	E7E00000	
0x0000005C	halt	111111	00000	00000	0000 0000 0000 0000	=	FC000000	

2. 通过波形图检验

(1) 第1~5条指令

A. addiu \$1, \$0, 8

0x00000000	addiu \$1,\$0,8	000010	00000	00001	0000 0000 0000 1000	=	08010008
------------	-----------------	--------	-------	-------	---------------------	---	----------

当前 PC 地址 (即 curPC) 为 0x00000000。PCSrc 信号为 00, 顺序执行;

rs 和 rt 分别为 0 号和 1 号, 执行指令后值分别为 0 和 8

ALUOp 为 000, 做加法运算, 运算结果为 8。ALUSrcA 信号为 0, ALU 的输入 A 为 rs 寄存器的值; ALUSrcB 信号为 1, ALU 的输入 B 为符号扩展 (ExtSel 信号 1) 后的立即数; RegDst 信号为 0, 因此目的寄存器是 rt; DBDataSrc 信号为 0, 写回寄存器的值来自 ALU 的输出。该指令不涉及数据存储器, 因此 mRD 和 mWR 均为 0。

B. ori \$2, \$0, 2

0x00000004	ori \$2,\$0,2	010010	00000	00010	0000 0000 0000 0010	=	48020002
------------	---------------	--------	-------	-------	---------------------	---	----------

当前 PC 为 0x00000004。PCSrc 为 00，顺序执行；rs 和 rt 分别为 0 号和 2 号，执行后值分别为 0 和 2；

ALUOp 为 003，做逻辑或运算，结果为 2。ALUSrcA 信号为 0，输入 rs 寄存器的值；

ALUSrcB 信号为 1，输入零扩展(ExtSel 信号为 0)后的立即数；DBDataSrc 信号为 0，写回寄存器的值来自 ALU 的输出。该指令不涉及数据存储器，因此 mRD 和 mWR 均为 0。

C. xori \$3,\$2,8

0x00000008	xori \$3,\$2,8	010011	00010	00011	0000 0000 0000 1000	=	4C430008
------------	----------------	--------	-------	-------	---------------------	---	----------

当前 PC 为 0x00000008。PCSrc 为 00，顺序执行；rs 和 rt 分别是 2 号和 3 号，执行前后值不变，分别为 2 和 10；

ALUOp 为 001，做异或

RegDst 信号为 1，因此目的寄存器由 rd 指定而不是 rt；DBDataSrc 信号为 0，写回寄存器的值来自 ALU 的输出。

该指令不涉及数据存储器，因此 mRD 和 mWR 均为 0。

D. ub \$4,\$3,\$1

0x0000000C	sub \$4,\$3,\$1	000001	00011	00001	00100 000 0000 0000	=	04612000
------------	-----------------	--------	-------	-------	---------------------	---	----------

当前 PC 为 0x0000000c。PCSrc 为 00，顺序执行；rs 和 rt 分别是 3 号和 1 号，值分别为 10 和 8；rd 是 4 号，值为 2

ALUOp 为 001，做减法运算，结果为 10-8=2。ALUSrcA 和 ALUSrcB 信号均为 0，因此 ALU 的输入来源都是寄存器堆；

RegDst 信号为 1，因此目的寄存器由 rd 指定而不是 rt；DBDataSrc 信号为 0，写回寄存器的值来自 ALU 的输出。该指令不涉及数据存储器，因此 mRD 和 mWR 均为 0。

E. and \$5, \$4, \$2

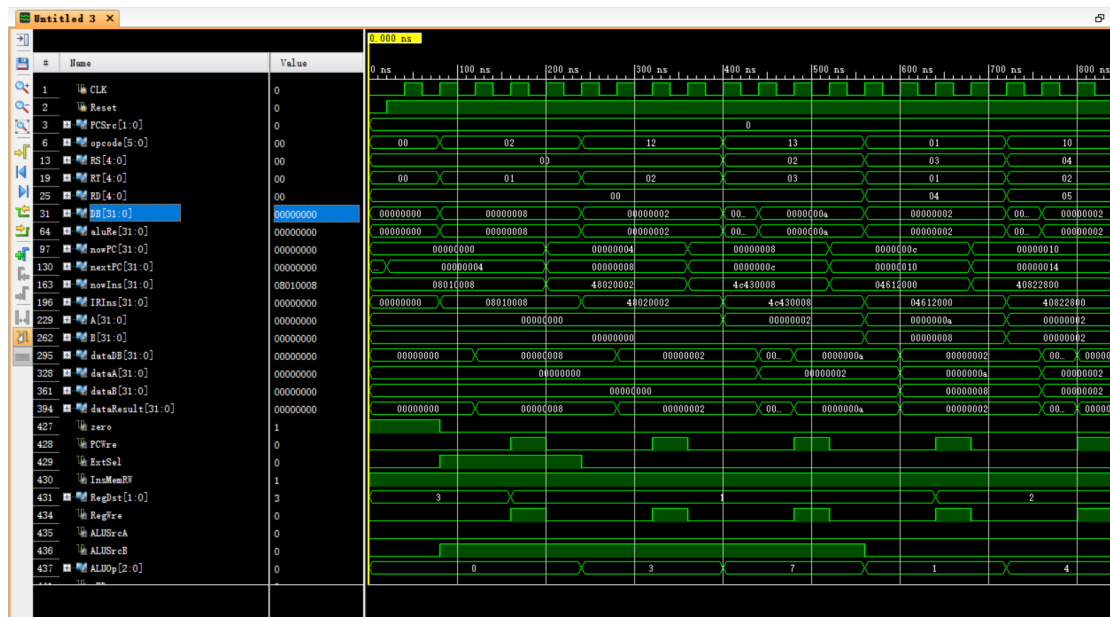
0x00000010	and \$5,\$4,\$2	010000	00100	00010	00101 000 0000 0000	=	40822800
------------	-----------------	--------	-------	-------	---------------------	---	----------

当前 PC 为 0x00000010，PCSrc 为 00，顺序执行；rs 和 rt 分别是 4 号和 2 号，值分别为 2 和 2；

ALUOp 为 100，做逻辑与运算，结果为 2&2=2。ALUSrcA 和 ALUSrcB 信号均为 0，因此 ALU 的输入来源都是寄存器堆；

RegDst 信号为 1，因此目的寄存器由 rd 指定而不是 rt；DBDataSrc 为 0，写回寄存器的值来自 ALU 的输出。该指令不涉及数据存储器，因此 mRD 和 mWR 均为 0。

指令 1~5 的波形图



(2) 第6~8条指令

F. sll \$5,\$5,2

0x00000014	sll \$5,\$5,2	011000	00000	00101	00101 00010 00 0000	=>	60052880
------------	---------------	--------	-------	-------	---------------------	----	----------

当前 PC 为 0x00000018, PCSrc 为 0, 顺序执行; rt 寄存器为 5 号, 其值为 $2 \ll 2 = 16$;

ALUOp 为 010, 做逻辑左移操作, 结果为 16; ALUSrcA 信号为 1, 输入为经零扩展 ALUSrcB 信号为 0, 输入 rt 寄存器的值;

RegDst 信号为 1, 因此目的寄存器由 rd 指定; DBDataSrc 信号为 0, 写回寄存器的值来自 ALU 的输出。

该指令不涉及数据存储器, 因此 mRD 和 mWR 均为 0。

G. beq \$5,\$1,-2(=,转14)

0x00000018	beq \$5,\$1,-2(=,转14)	110100	00101	00001	1111 1111 1111 1110	=>	D0A1FFFE
------------	-----------------------	--------	-------	-------	---------------------	----	----------

当前 PC 为 0x00000018, 这是跳转指令。rs 和 rt 分别为 1 号和 5 号, 值分别为 16 和 8, 不满足跳转条件, 不跳转

ALUOp 为 001, 做减法运算, 结果为 $16 - 8 \neq 0$, ALU 的 zero 输出为 0, 表示运算数不相等, beq 条件为假。因此顺序执行

该指令不涉及写回寄存器堆, 也不涉及数据存储器, 因此 RegWre、mRD、mWR 信号均为 0。

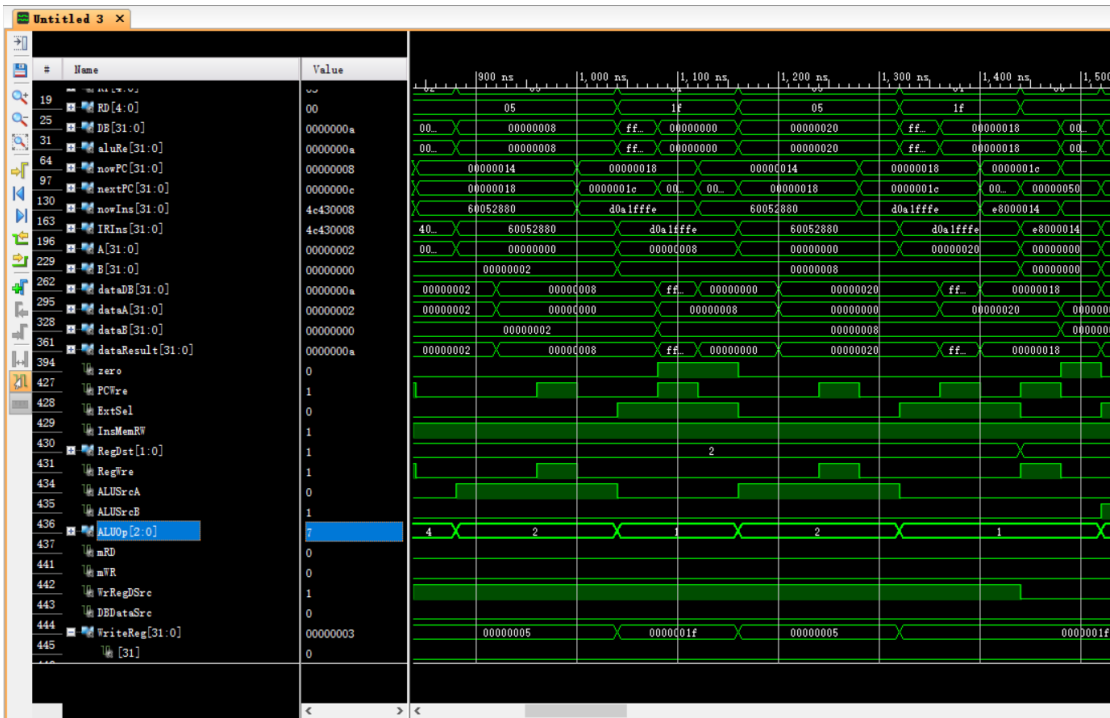
H. jal 0x00000050

0x0000001C	jal 0x00000050	111010	00000	00000	0000 0000 0101 0000	=>	E8000050
------------	----------------	--------	-------	-------	---------------------	----	----------

当前 PC 为 0x0000001C。跳转目标地址是 32 位的, 其高 4 位和当前 PC 的高 4 位相同, 低 28 位是指令的低 26 位左移两位的结果, 这个地址就是 0x00000050。从波形图也可以看到, nextPC 为 0x00000050。31 号寄存

器保存顺序执行的下一条指令，即值为 0x00000020
J 型指令不涉及寄存器堆、数据存储器、ALU，无关控制信号不做讨论。

指令 6~8 的波形图



(3) 第9~13条指令

I. slt \$8,\$13,\$1

0x00000020	slt \$8,\$13,\$1	100111	01101	00001	01000 000 0000 0000	=	9DA14000
------------	------------------	--------	-------	-------	---------------------	---	----------

当前 PC 为 0x00000020。PCSrc 为 00，顺序执行； rs 和 rt 分别为 13 号
和 1 号，执行后值分别为 0 和 8； rd 为 8 号，值为 0
ALUOp 为 110，做带符号比较操作，结果为 0<8=0。
该指令不涉及数据存储器，因此 mRD 和 mWR 均为 0。

J. addiu \$14,\$0,-2

0x00000024	addiu \$14,\$0,-2	000010	00000	01110	1111 1111 1111 1110	=	080EFFFF
------------	-------------------	--------	-------	-------	---------------------	---	----------

当前 PC 地址为 0x00000024。PCSrc 信号为 00，顺序执行； rs 和 rt 分别
为 0 号和 14 号，执行指令后值分别为 0 和 -2
ALUOp 为 000，做加法运算，运算结果为 -2。ALUSrcA 信号为 0，ALU 的输入 A
为 rs 寄存器的值；ALUSrcB 信号为 1，ALU 的输入 B 为符号扩展（ExtSel 信号 1）
后的立即数；
RegDst 信号为 0，因此目的寄存器是 rt；DBDataSrc 信号为 0，写回寄存
器的值来自 ALU 的输出。该指令不涉及数据存储器，因此 mRD 和 mWR
均为 0。

K. slt \$9,\$8,\$14

0x00000028	slt \$9,\$8,\$14	100111	01000	01110	01001 000 0000 0000	=	9D0E4800
------------	------------------	--------	-------	-------	---------------------	---	----------

当前 PC 为 0x00000028。PCSrc 为 00，顺序执行；rs 和 rt 分别为 8 号和 14 号，执行后值分别为 1 和 -2；rd 为 9 号，值为 1

ALUOp 为 110，做带符号比较操作，结果为 $1 < -2 = 1$ 。

该指令不涉及数据存储器，因此 mRD 和 mWR 均为 0。

L. slti \$10,\$9,2

0x0000002C	slti \$10,\$9,2	100110	01001	01010	0000 0000 0000 0010	=	992A0002
------------	-----------------	--------	-------	-------	---------------------	---	----------

当前 PC 为 0x0000002C。PCSrc 为 00，顺序执行；rs 和 rt 分别为 9 号和 10 号，执行后值分别为 1 和 0；

ALUOp 为 110，做带符号比较操作，结果为 $1 < 2 = 0$ 。ALUSrcA 信号为 0，输入 rs 寄存器的值；ALUSrcB 信号为 1，输入符号扩展（ExtSel 信号为 1）后的立即数；RegDst 信号为 1，因此目的寄存器是 rt；DBDataSrc 信号为 0，写回寄存器的值来自 ALU 的输出。

该指令不涉及数据存储器，因此 mRD 和 mWR 均为 0。

M. slti \$11,\$10,0

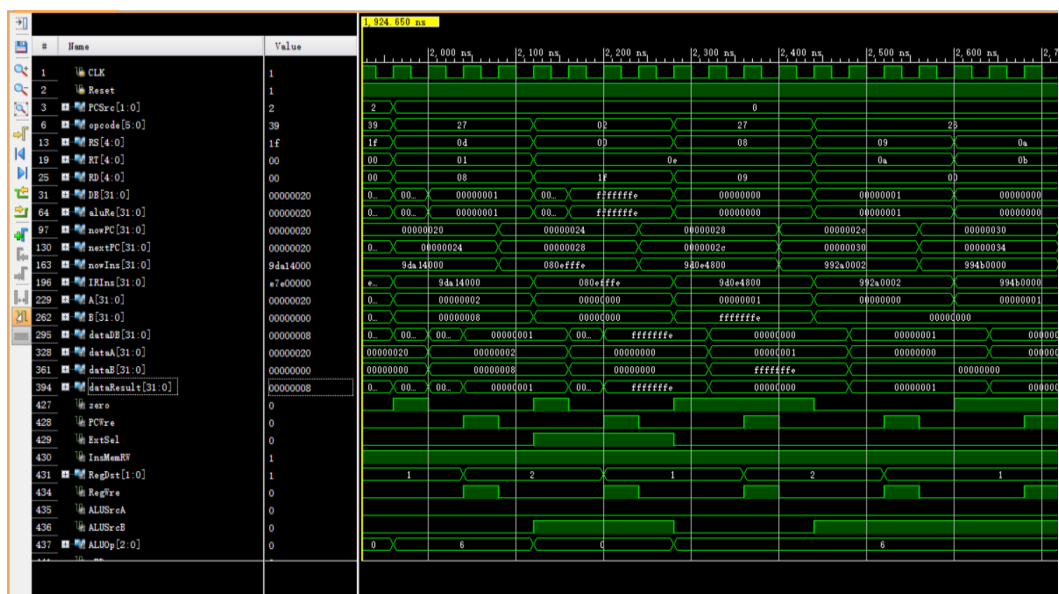
0x00000030	slti \$11,\$10,0	100110	01010	01011	0000 0000 0000 0000	=	994B0000
------------	------------------	--------	-------	-------	---------------------	---	----------

当前 PC 为 0x00000030。PCSrc 为 00，顺序执行；rs 和 rt 分别为 10 号和 11 号，执行后值分别为 0 和 1；

ALUOp 为 110，做带符号比较操作，结果为 $0 < 0 = 1$ 。ALUSrcA 信号为 0，输入 rs 寄存器的值；ALUSrcB 信号为 1，输入符号扩展（ExtSel 信号为 1）后的立即数；RegDst 信号为 1，因此目的寄存器是 rt；DBDataSrc 信号为 0，写回寄存器的值来自 ALU 的输出。

该指令不涉及数据存储器，因此 mRD 和 mWR 均为 0。

指令 9~13 的波形图



(4) 第14~16条指令

N. add \$11,\$11,\$10

0x00000034 | add \$11,\$11,\$10 | 000000 | 01011 | 01010 | 01011 000 0000 0000 | = | 016A5800 |

当前 PC 为 0x00000034。PCSrc 为 00，顺序执行；rs 和 rt 分别是 11 号和 10 号，执行前后值不变，分别为 1 和 0；

ALUOp 为 000，做加法运算，结果为 $1+0=1$ 。ALUSrcA 和 ALUSrcB 信号均为 0，因此 ALU 的输入来源都是寄存器堆；

RegDst 信号为 1，因此目的寄存器由 rd 指定而不是 rt；DBDataSrc 信号为 0，写回寄存器的值来自 ALU 的输出。

该指令不涉及数据存储器，因此 mRD 和 mWR 均为 0。

O. bne \$11,\$2,-2 (≠,转34)

0x00000038 | bne \$11,\$2,-2 (≠,转 34) | 110101 | 01011 | 00010 | 1111 1111 1111 1110 | = | D562FFFE |

当前 PC 为 0x00000038。这是跳转指令。rs 和 rt 分别是 2 号和 11 号，值分别为 2 和 0；

ALUOp 为 001，做减法运算，结果为 $2-2!=1$ 。此结果不是 0，故 ALU 的 zero 输出为 0，这表示比较结果不相等，bne 条件为真，PC 跳转到 0x00000034。由波形图可以看出，当前指令地址为 0x00000038，下一条指令的地址为 00000034，这证明跳转确实发生了。该指令不涉及写回寄存器堆，也不涉及数据存储器，因此 RegWre、mRD、mWR 信号均为 0。

P. addiu \$12,\$0,-2

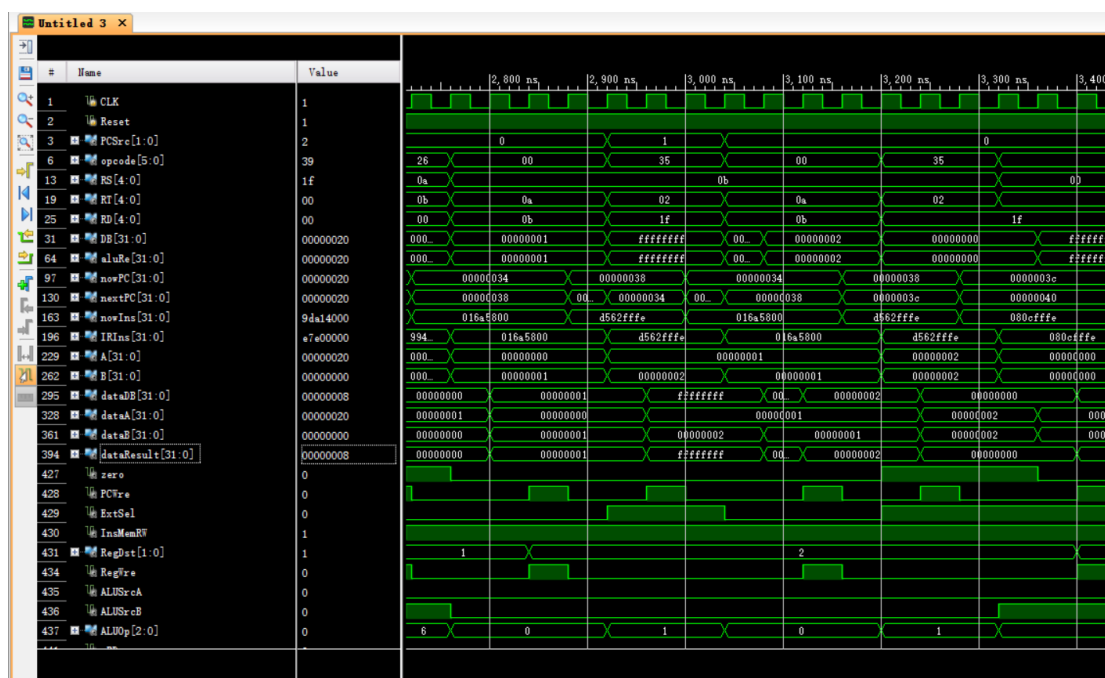
0x0000003C | addiu \$12,\$0,-2 | 000010 | 00000 | 01100 | 1111 1111 1111 1110 | = | 080CFFFE |

当前 PC 地址为 0x0000003C。PCSrc 信号为 00，顺序执行；rs 和 rt 分别为 0 号和 12 号，执行指令后值分别为 0 和 -2

ALUOp 为 000，做加法运算，运算结果为 -2。ALUSrcA 信号为 0，ALU 的输入 A 为 rs 寄存器的值；ALUSrcB 信号为 1，ALU 的输入 B 为符号扩展 (ExtSel 信号 1) 后的立即数；

RegDst 信号为 0，因此目的寄存器是 rt；DBDataSrc 信号为 0，写回寄存器的值来自 ALU 的输出。该指令不涉及数据存储器，因此 mRD 和 mWR 均为 0。

指令14~16的波形图



(5) 第17~19条指令

Q. addiu \$12,\$12,1

0x00000040 | addiu \$12,\$12,1 | 000010 | 01100 | 01100 | 0000 0000 0000 0001 | => 098C0001 |

当前 PC 地址为 0x0000003C。PCSrc 信号为 00，顺序执行；rs 和 rt 都为 12 号，执行指令前后值分别为-2 和-1

ALUOp 为 000，做加法运算，运算结果为-1。ALUSrcA 信号为 0，ALU 的输入 A 为 rs 寄存器的值；ALUSrcB 信号为 1，ALU 的输入 B 为符号扩展（ExtSel 信号 1）后的立即数；

RegDst 信号为 0，因此目的寄存器是 rt；DBDataSrc 信号为 0，写回寄存器的值来自 ALU 的输出。该指令不涉及数据存储器，因此 mRD 和 mWR 均为 0。

R. bltz \$12,-2 (<0,转40)

0x00000044 | bltz \$12,-2 (<0,转 40) | 110110 | 01100 | 00000 | 1111 1111 1111 1110 | => D980FFFE |

当前 PC 为 0x00000044，PCSrc 为 00，顺序执行；rs 和 rt 分别为 12 号和 0 号，值分别为-1 和 0；

ALUOp 为 000，做加法运算，这是因为将运算数与 0 相加结果仍为本身，这样就可以利用 ALU 的 sign 输出。ALUSrcA 和 ALUSrcB 信号都为 0，ALU 的操作数都来自寄存器堆。ALU 将算出-1+0=-1，因此 sign 输出为 1，bltz 条件为真。因此 PCSrc 为 10，表示 nextIAddr 为 currentIAddr+4-2×4，下一步将跳转到分支去。由波形图可以看出，当前指令地址为 0x00000044，

下一条指令地址为 0x00000040，因此确实发生了跳转。

该指令不涉及写回寄存器堆，也不涉及数据存储器，因此 RegWre、mRD、mWR 信号均为 0。

S. andi \$12,\$2,2

0x00000048	andi \$12,\$2,2	010001	00010	01100	0000 0000 0000 0010	=	444C0002
------------	-----------------	--------	-------	-------	---------------------	---	----------

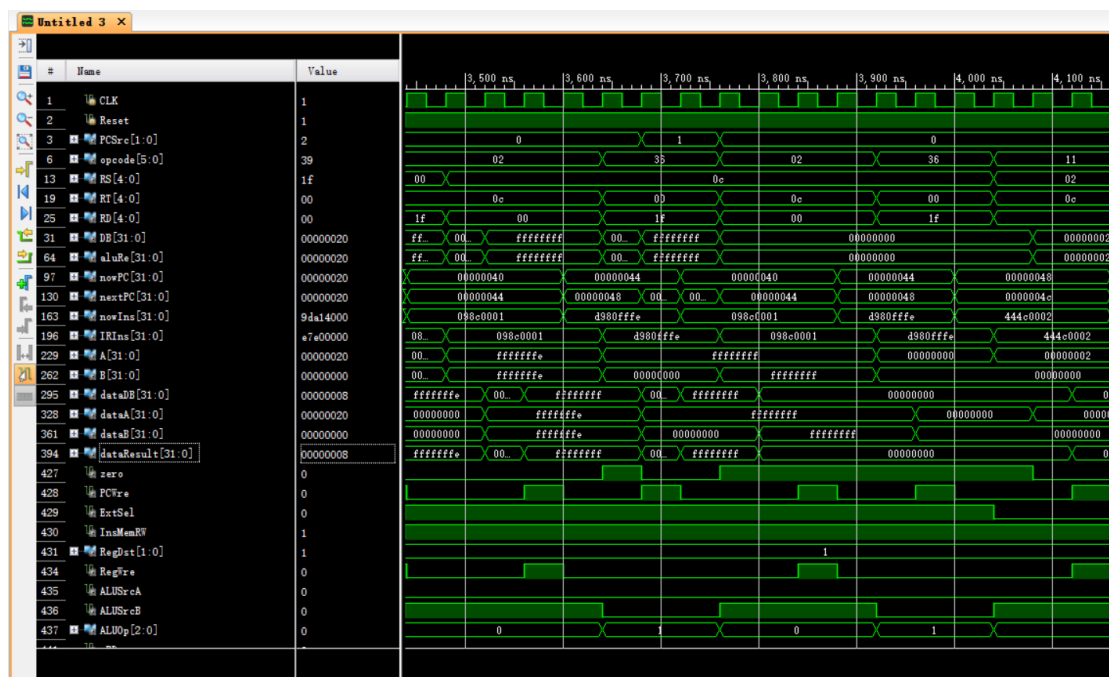
当前 PC 为 0x00000048。PCSrc 为 00，顺序执行；rs 和 rt 分别为 2 号和 12 号，执行后值分别为 2 和 -1；

ALUOp 为 100，做逻辑与运算，结果为 2。ALUSrcA 信号为 0，输入 rs 寄存器的值；ALUSrcB 信号为 1，输入零扩展（ExtSel 信号为 0）后的立即数；

RegDst 信号为 0，因此目的寄存器是 rt；DBDataSrc 信号为 0，写回寄存器的值来自 ALU 的输出。

该指令不涉及数据存储器，因此 mRD 和 mWR 均为 0。

指令 17~19 的波形图



(6)第20、24条指令（由于21~23指令由于之前的跳转已经被执行过，所以会断开）

T. j 0x0000005C

0x0000004C	j 0x0000005C	111000	00000	00000	0000 0000 0101 1100	=	E000005C
------------	--------------	--------	-------	-------	---------------------	---	----------

当前 PC 为 0x0000004C。PCSrc 为 10，为无条件跳转。跳转目标地址是 32 位的，其高 4 位和当前 PC 的高 4 位相同，低 28 位是指令的低 26 位左移两位的结果，这个地址就是 0x0000005C。从波形图也可以看到，nextIPC 为 0x0000005C。

J 型指令不涉及寄存器堆、数据存储器、ALU，无关控制信号不做讨论。

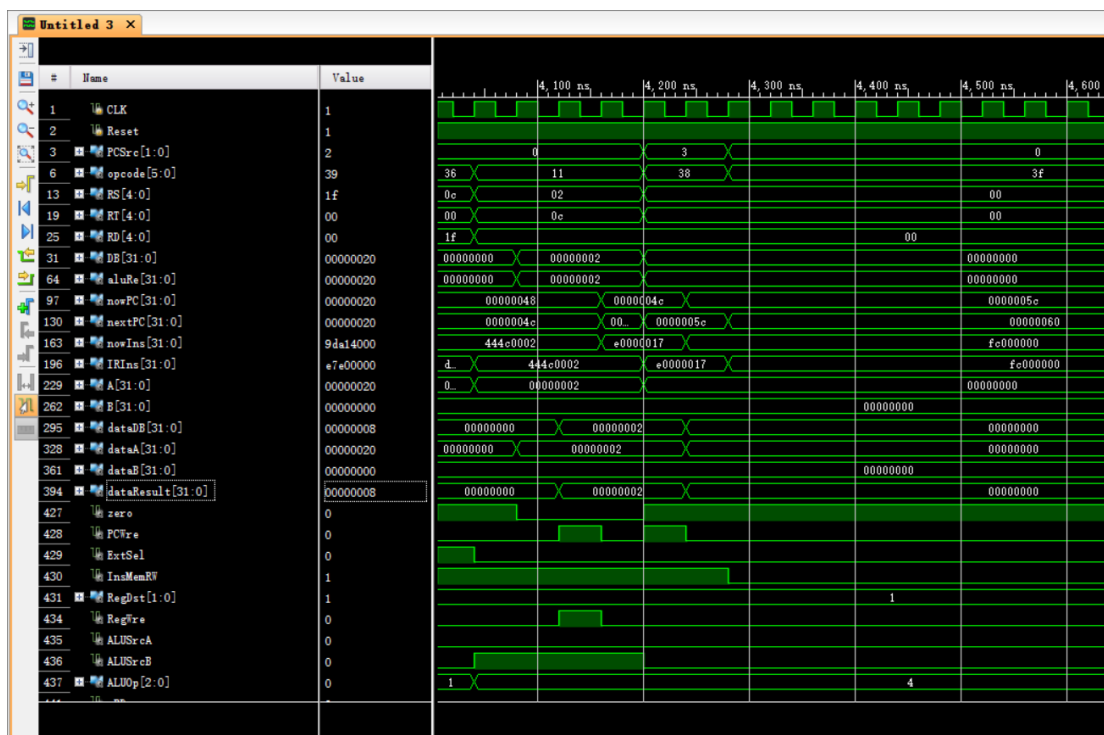
X. halt

`0x0000005C` | `halt` | `111111` | `00000` | `00000` | `0000 0000 0000 0000` | = | `FC000000`

当前 PC 为 0x0000005C，PCSrc 为 11，因此 nextPC 于 curPC 相同。halt 指令使得 PC 保持不变，从波形图可以看出，4300ns 以后，所有信号、数值都保持不变了。

该指令不涉及其他操作，无关信号不做讨论。

指令 20、24 的波形图



(7) 第21~23条指令

U. sw \$2,4(\$1)

`0x00000050` | `sw $2,4($1)` | `110000` | `00001` | `00010` | `0000 0000 0000 0100` | = | `C0220004`

当前 PC 为 0x00000030，PCSrc 为 00，顺序执行；rs 和 rt 分别为 1 号和 2 号，值分别为 8 和 2；

ALUOp 为 000，做加法运算；ALUSrcA 为 0，运算数 A 为寄存器堆的 rs 中的值，

ALUSrcB 为 1，表示运算数 B 为符号扩展（ExtSel 为 1）后的立即数；最后 ALU 运算出来的结果为 8+4=12，表示数据存储器地址。

此时 mWR 信号变为 1，数据存储器写使能有效，根据数据通路图便可得知，在时钟下降沿到来时，rt 寄存器的值（即 8）被写入数据存储器，写入地址即是 ALU 计算结果，即 12。

RegWre 信号为 0，结果不需写回寄存器堆。

V. lw \$13,4(\$1)

`0x00000054` | `lw $13,4($1)` | `110001` | `00001` | `01101` | `0000 0000 0000 0100` | = | `C42D0004`

当前 PC 为 0x00000054, PCSrc 为 00, 顺序执行; rs 为 1 号, 其值为 8;
rt 为 13 号, 在执行前值为 0, 执行后为 2.

ALUOp 为 000, 做加法运算; ALUSrcA 为 0, 运算数 A 为寄存器堆的 rs 中的值, ALUSrcB 为 1, 表示运算数 B 为符号扩展 (ExtSel 为 1) 后的立即数; 最后 ALU 运算出来的结果为 $8+4=12$, 表示数据存储器地址。此时 mWR 信号变为 0, 不可写; mRD 信号变为 1, 数据存储器读使能有效;

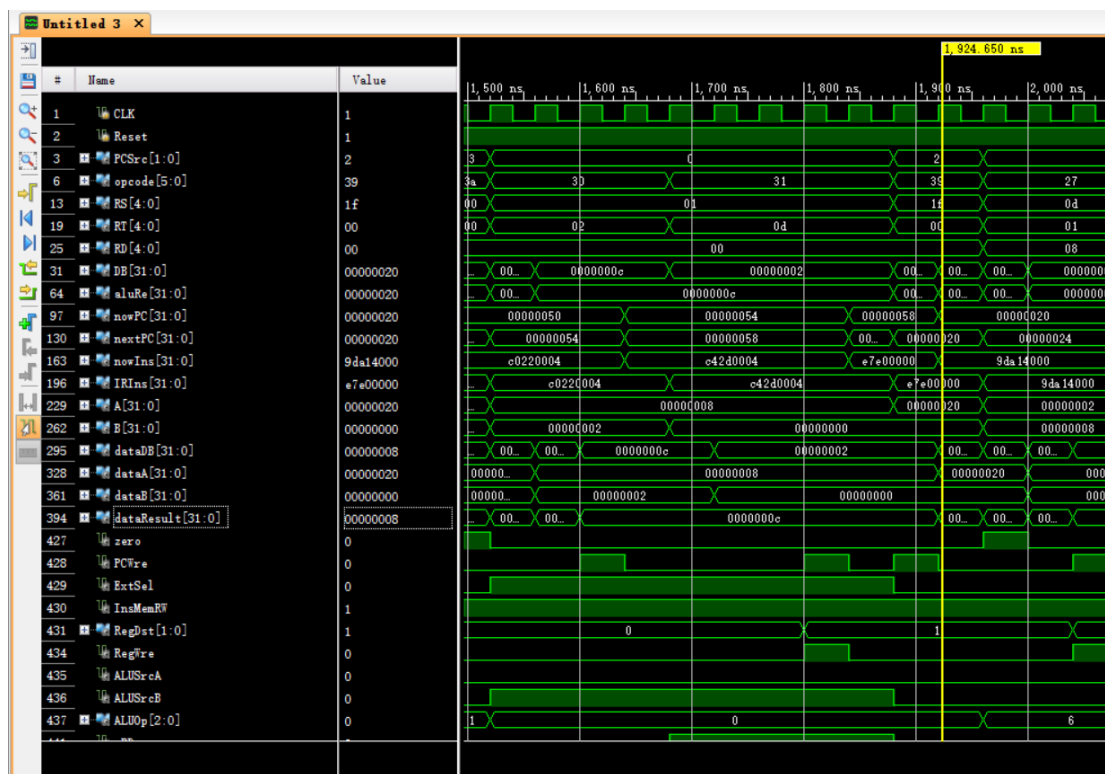
DBDataSrc 信号为 1, 数据总线 (DataBus) 上的数据来自数据存储器的输出 RegWre 信号为 1, 数据总线上的数据被写入寄存器堆。

W. jr \$31

0x00000058 jr \$31 111001 11111 00000 0000 0000 0000 0000 = E7E00000

当前 PC 为 0x00000058。PCSrc 为 10, 跳转回 31 号寄存器所存地址, 就是 0x00000020。从波形图也可以看到, nextPC 为 0x00000050。

指令 21~23 的波形图



3. 在Basy3板上实现多周期CPU

(1) 思想方法

根据要求, 在板上显示当前PC、下一条PC、rs、rt数据及地址、ALU结果、DB总线等数据。

所以先写出显示七段数码管的模块, 再将封装好的CPU模块和显示模块进行封装, 就可以综合、实现并烧写到板子上了。

(2) 操作方法

A. 设计7段数码管对应输出

```

module sevenLED(input [3:0] num, output reg [7:0] dispcode );
always @( num )
begin
    case (num)
        4'b0000 : dispcode = 8'b1100_0000; //0: '0'-亮灯, '1'-熄灯
        4'b0001 : dispcode = 8'b1111_1001; //1
        4'b0010 : dispcode = 8'b1010_0100; //2
        4'b0011 : dispcode = 8'b1011_0000; //3
        4'b0100 : dispcode = 8'b1001_1001; //4
        4'b0101 : dispcode = 8'b1001_0010; //5
        4'b0110 : dispcode = 8'b1000_0010; //6
        4'b0111 : dispcode = 8'b1101_1000; //7
        4'b1000 : dispcode = 8'b1000_0000; //8
        4'b1001 : dispcode = 8'b1001_0000; //9
        4'b1010 : dispcode = 8'b1000_1000; //A
        4'b1011 : dispcode = 8'b1000_0011; //b
        4'b1100 : dispcode = 8'b1100_0110; //C
        4'b1101 : dispcode = 8'b1010_0001; //d
        4'b1110 : dispcode = 8'b1000_0110; //E
        4'b1111 : dispcode = 8'b1000_1110; //F
        default : dispcode = 8'b1111_1111; //不亮
    endcase
end endmodule

```

B. 4位计数器

```

module counter4(
    input clk, reset,
    output reg [1:0] count
);

always @ (posedge clk or negedge reset) begin
    if(reset == 0) begin
        count <= 2'b00; //复位
    end
    else begin
        if(count == 2'b11)
            count <= 2'b00;
        else begin
            count <= count + 2'b01; // 加 1 计数
        end
    end
end
endmodule

```

C. 38译码器

```

module decoder38(
    input A,
    input B,
    input C,
    input EN,
    output reg [7:0] Y_Out
);
always@(A or B or C or EN) begin
    if(!EN)
        Y_Out = 8'b0000_0000;
    else begin
        case({C,B,A})
            3'b000: Y_Out = 8'b0000_0001;
            3'b001: Y_Out = 8'b0000_0010;
            3'b010: Y_Out = 8'b0000_0100;
            3'b011: Y_Out = 8'b0000_1000;
            3'b100: Y_Out = 8'b0001_0000;
            3'b101: Y_Out = 8'b0010_0000;
            3'b110: Y_Out = 8'b0100_0000;
            3'b111: Y_Out = 8'b1000_0000;
            default: Y_Out = 8'b0000_0000;
        endcase
    end
end
endmodule

```

D. 时钟分频器

```

module clock_div(
    input clk,
    output reg clk_sys = 0
);
    parameter cnt_sim = 4;
    parameter cnt_1Hz = 49999999;
    parameter cnt_100Hz = 499999;
    parameter cnt_1kHz = 49999;
    integer div_counter = 0; // 分频计数器
    always @(posedge clk) begin
        if(div_counter >= cnt_1kHz) begin // 用于生成 比特流文件 下载用
            clk_sys <= ~clk_sys; // 电平反向
            div_counter <= 0;
        end
        else begin
            div_counter <= div_counter + 1;
        end
    end
end
endmodule

```

E. 按键消抖（重点难点——实验与现实需要的结合）

由于硬件原因，按板上的按钮时提供的电平变化可能不是单一的、稳定的上升沿或下降沿，因此需要对按钮进行消抖，其原理是按下按钮后延迟 20ms 才接受下一次按下。

```
module fangdou(
    input clk,
    input reset,    // 复位
    input key_in,  // 按键
    output key_out  // 输出
);

    reg key3, key2, key1;
    always @(posedge /*clk_100Hz*/ clk or negedge reset) begin
        if(!reset)
            begin
                key3 <= 1'b1;
                key2 <= 1'b1;
                key1 <= 1'b1;
            end
        else
            begin
                key3 <= key2;
                key2 <= key1;
                key1 <= key_in;
            end
        end
    end

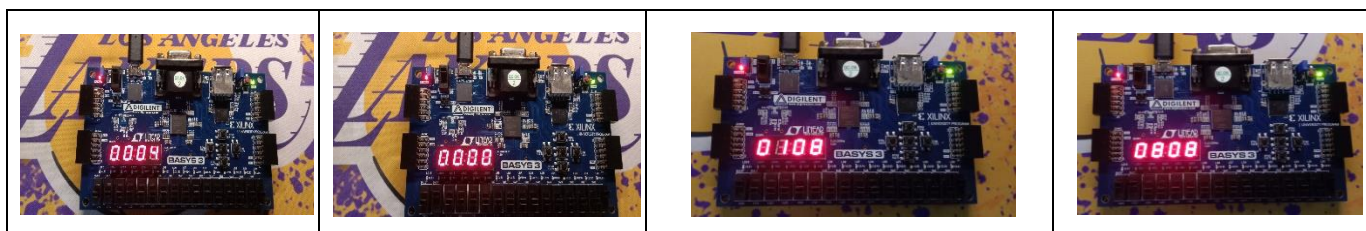
    assign key_out = key3 & key2 & key1;

endmodule
```



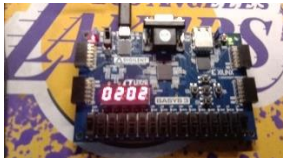
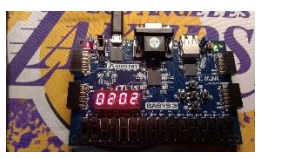
(3) 贴图展示(前五条指令)—有些相同图片（如PC）源自单周期实验，所以导致亮度不一，其中ALU结果DB总线为单周期时的显示要求。另因为前五条指令的state阶段一样，和其他类型指令的state在最后统一展示

① addiu \$1,\$0,8



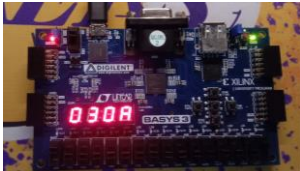
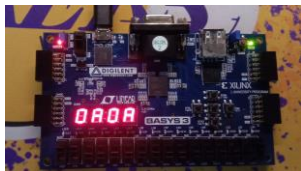
当前 PC:下一 PC	rs 地址:rs 数据	rt 地址:rt 数据	ALU 结果:DB 总线
-------------	-------------	-------------	--------------




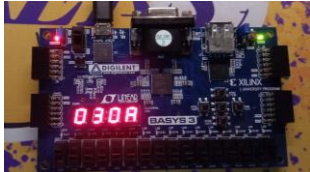
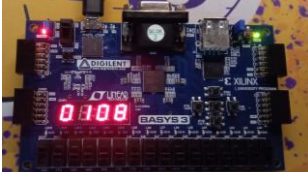

② ori \$2,\$0,2

当前 PC:下一 PC	rs 地址:rs 数据	rt 地址:rt 数据	ALU 结果:DB 总线
			

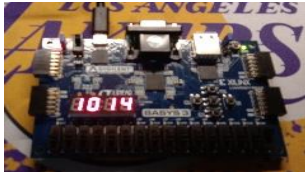

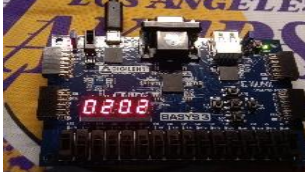
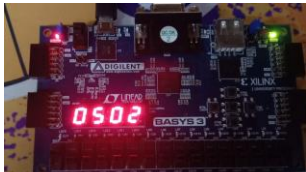
③ xori \$3, \$2, 8

当前 PC:下一 PC	rs 地址:rs 数据	rt 地址:rt 数据	ALU 结果:DB 总线
			

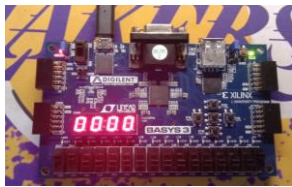
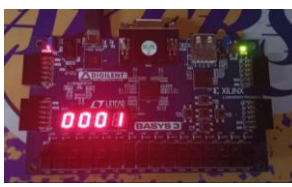
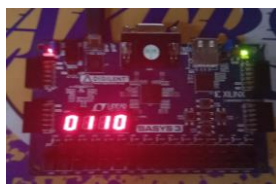
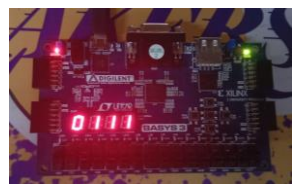
④ sub \$4,\$3,\$1

当前 PC:下一 PC	rs 地址:rs 数据	rt 地址:rt 数据	ALU 结果:DB 总线
			

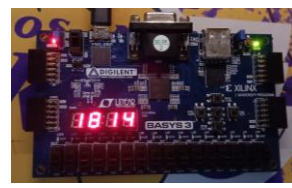
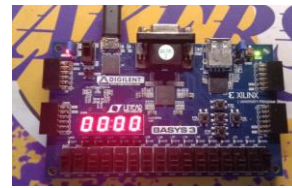

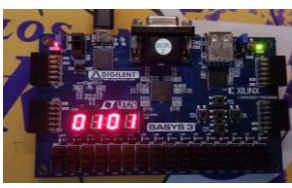
⑤ and \$5,\$4,\$2

当前 PC:下一 PC	rs 地址:rs 数据	rt 地址:rt 数据	ALU 结果:DB 总线
			


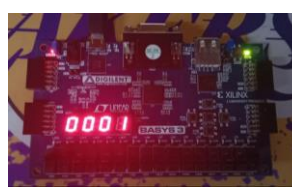
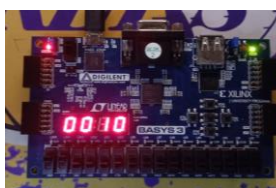
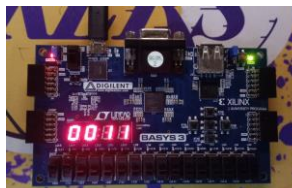
⑥ State 展示一 (add、sub、addiu、and、andi、ori、xori、sll、slt、slti)

IF	ID	EXE	WB
			






⑦ State 展示二 (beq、bne、bltz)

beq \$5,\$1,-2	IF	ID	EXE
			

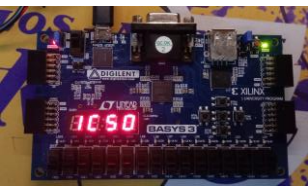
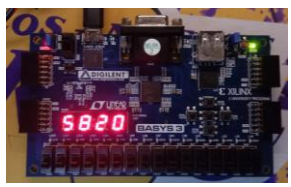
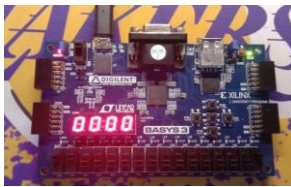

⑧ State 展示三 (sw)

IF	ID	EXE	MEM
			

⑨ State 展示四 (lw)

IF	ID	EXE	MEM	WB
				

⑩ State 展示五 (J、jal、jr、halt)

jal 0x0000050	jr \$31	IF	ID
			

六、实验心得

1.意识到了总结经验的重要性,经过单周期CPU的历练,多周期CPU写起来就较为顺手,但是由于总结不够,遇到问题只是感觉这个问题很熟悉,在设计单周期CPU时也出现过,而不能立刻解决,总要回过头看单周期的实验才能想起解决思路。

2.代码书写要规范,特别是关键地方要给出注释,也是承接第一点的经验教训,好在单周期CPU的关键代码注释较为全面,所以读起来简单且很快就能接上当时的思路。

3.不同的学习阶段要借助不同的学习工具。比如清晰简洁的图示在模块化设计时显得尤为重要,而在要同时处理很多个信号的状态转换时真值表能帮助我们快速而准确地处理好信号之间的转换,且能大大降低出错率。

4.单周期CPU和多周期CPU的设计思路还是会有不一样的地方,设计以及解决问题时不能陷入一种僵化思维。比如在单周期CPU里面即使不单独做一个指令切割分段模块也能较好地完成,但是多周期CPU如果不设计一个单独的指令切割分段模块则会导致处理情况的复杂性极大增加。

大二机计组实验课课程总结:

实验课做了三个实验,我的心态也是随着这三个实验的时间进展慢慢变化的。

① 第一个实验时MIPS汇编语言的应用,我选择了用MIPS汇编语言实现冒泡排序。

该实验现在回过头来看比较简单,但是在当时对于刚接触汇编语言的我来说这还是个不小的挑战,难点在于寄存器的分配,以及不同种类寄存器的特征,是否会自动保存决定了要不要压栈,还有0号、sp、gp等特殊寄存器的使用。汇编语言写成的冒泡排序是在第二个循环里面没有用控制第一次循环的变量,这也让我对冒泡多种多样的写法有了更进一步的认识。从初学时的恐惧,到熟悉汇编编程的基本操作、基本指令之后的欣慰以及最后自己写出多个版本的汇编冒泡的成就感令我久久不能忘怀。现在下半学期开设了单片机课程,里面用的是arm指令集汇编,当我能轻松且很快的入手arm指令集时我才惊讶的发现汇编指令集多为大同小异,学习了其中一种便能很快消化其他种类的指令集。这也让我感觉到了满满的收获。越体会到汇编编程的不易,我对写出编译器的前辈就越敬佩,更进一步想到第一批将二进制机器码转为汇编语言的初代程序员的不易而深受鼓舞与激励。

② 第二个实验便是早在大一时就听学长学姐们整日念叨并为此叫苦不迭的单周CPU设计。我个人比较习惯有作业都尽早做完,于是我从刚开始完全不知从何下手硬啃了

前两个星期，才总算勉强写出了能仿真的项目。之后便遇到了最大的困难——我的程序烧写到开发板时总是不能正常跳转，这一情况整整卡了我一个多星期，期间我又为自己不成熟的代码模块化吃尽了苦头甚至几度重新规划模块，遵循老师的建议该细分的细分该整合的整合。期间的痛苦不堪回首，看到个别同学借助“祖传代码”不劳而获时，我也曾动摇，最后还是咬牙坚持了下来，并且在Ta的指导下发现了不能成功跳转的问题——从第一条指令开始我就没有成功读进去。最后我要小聪明的做法——在最开始加一条空指令实现清零的目的还是被Ta发现并且指正——这种做法不是长久之计，不利于对CPU设计的理解和学习。这一阶段可谓是备受打击，也深刻体会到CPU设计的不易，由此对这块小小的CPU以及其设计者深感敬佩。

③ 第三个实验是多周期CPU，在这个阶段由于有了前面单周期CPU的基础相对较为轻松。也顺利地解决了单周期CPU留下来的不能正确跳转的问题（只要在刚开始阶段反转时钟进行一次清零即可）。多周期CPU设计的相对顺利的完成，是对我上一个实验中所受的种种煎熬的最好回报。这个对我来说较大型的项目，也第一次让我感受到了作为一个程序员做一天打代码的乐趣所在。记得老九君说过，程序员学得越深越不喜欢打游戏，因为游戏就是一些随机事件的组合体，这种完成任务式的快感已经不能满足曾经设计过项目、游戏的一个以创造感为满足的程序员了。程序员的使命就是创造，这种造物主的感觉只属于上帝和程序员。

最后，课程虽然结束了，但是这门课带给我的不仅是汇编以及CPU设计，更是打码前的规划设计，打码时的耐得住寂寞，忍得住诱惑的耐心和毅力，也是打码后的及时总结反思这种优秀程序员必备的品质。