摘要

本次硬件课程设计基于 Verilog 语言实现了简易 16 位 CPU。指令系统包含了运算类指令、传送类指令、储存类指令、控制类指令。借助 Quartus II 9.0 软件实现程序的编写和结果仿真。

关键词: CPU 设计，Verilog

目录

第 1 章　设计目的与要求

1.1　设计目的

1. 掌握 CPU 数据通路图的构成、原理及其设计方法

2. 掌握 CPU 的实现方法，代码实现方法

3. 认识和掌握指令与 CPU 的关系

1.2　设计要求

设计一个 16 位的 CPU。指令系统的指令条数不能少于 10，指令格式不限。

1.3　实验平台

Quartus II 9.0

1.4　实现语言

硬件描述语言——Verilog HDL。

第 2 章　设计原理

2.1　单周期 CPU

单周期 CPU 指的是一条指令的执行在一个时钟周期内完成，然后开始下一条指令的执行，即一条指令用一个时钟周期完成。电平从低到高变化的瞬间称为时钟上升沿，两个相邻时钟上升沿之间的时间间隔称为一个时钟周期。时钟周期一般也称振荡周期。

2.2　指令的处理过程

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。单周期 CPU，是在一个时钟周期内完成这五个阶段的处理。

图 2.1: 单周期 CPU 指令处理过程

第 3 章　指令系统

3.1　指令结构

opcode(6 位)	rs(5 位)	rt(5 位)
rd(5 位)/immediate(16 位)		

- op：为操作码；
- rs：为第 1 个源操作数寄存器；
- rt：为第 2 个源操作数寄存器，或目的操作数寄存器；
- rd：为目的操作数寄存器；
- 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（Laod）/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。

3.2　指令功能

3.2.1　运算类指令

- **add rd , rs, rt**

  功能：rd←rs + rt。reserved 为预留部分，即未用，一般填“0”。

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

- **addi rt , rs ,immediate**

  功能：rt←rs + immediate。

000001	rs(5 位)	rt(5 位)
immediate(16 位)		

- **sub rd , rs , rt**

  功能：rd←rs - rt。

000010	rs(5 位)	rt(5 位)
rd(5 位)	reserved	

- **ori rt , rs ,immediate**

  功能：rt←rs | immediate。

010000	rs(5 位)	rt(5 位)
immediate(16 位)		

- **and rd , rs , rt**

  功能：rd←rs & rt；逻辑与运算。

010001	rs(5 位)	rt(5 位)
rd(5 位)	reserved	

- **or rd , rs , rt**

  功能：rd←rs | rt；逻辑或运算。

010010	rs(5 位)	rt(5 位)
rd(5 位)	reserved	

3.2.2 传送类指令

- **move rd , rs**

  功能：rd←rs + $0 ； $0=$zero=0。

100000	rs(5 位)	00000
rd(5 位)	reserved	

3.2.3 储存类指令

- **sw rt ,immediate(rs) 写存储器**

  功能：memory[rs+ immediate]←rt。

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

- **lw rt , immediate(rs) 读存储器**

  功能：rt ← memory[rs + immediate]。

100111	rs(5 位)	rt(5 位)
immediate(16 位)		

3.2.4 控制类指令

- **beq rs,rt,immediate**

  功能：if(rs=rt) pc←pc +8 + immediate «3。

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

- **jmp immediate**

  功能：强制跳转指令 pc←pc +8 + immediate «3。

110001	00000	00000
immediate(16 位)		

3.2.5 停机指令

- **halt**

  功能：停机；不改变 PC 的值，PC 保持不变。

111111	00000	00000
0000000000000000		

第 4 章　详细设计与实现

4.1　整体框架

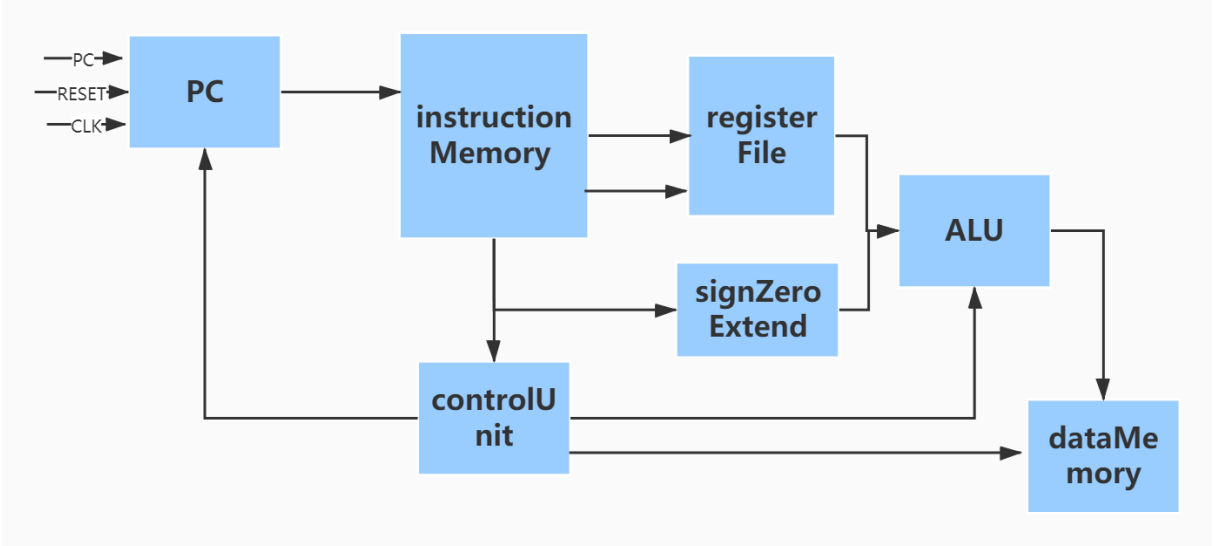

图 4.1: 整体框架图

4.2　数据通路

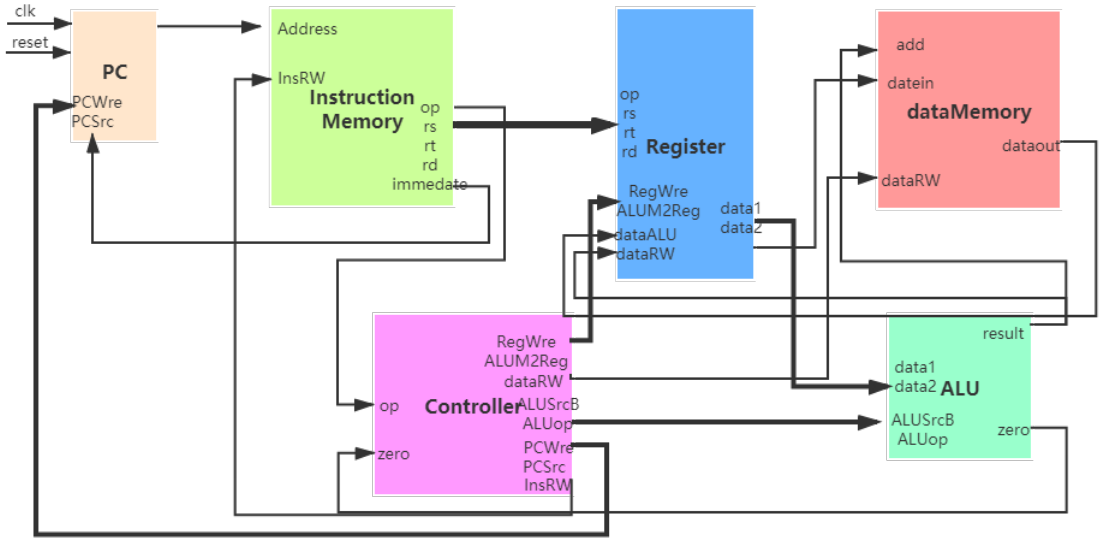

图 4.2: 数据通路图

4.3 各功能模块设计

程序主要由六个功能模块组成。项目文件目录结构如图所示。

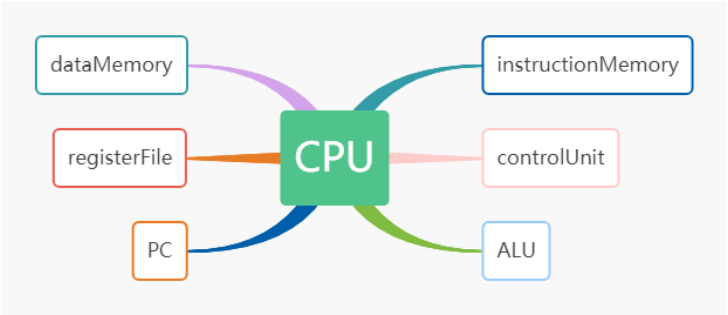

图 4.3: 程序的主要功能模块

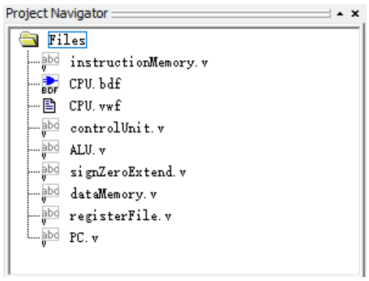

图 4.4: 项目文件目录结构

4.3.1 顶层文件

顶层文件 CPU.bdf 连接各个功能模块。

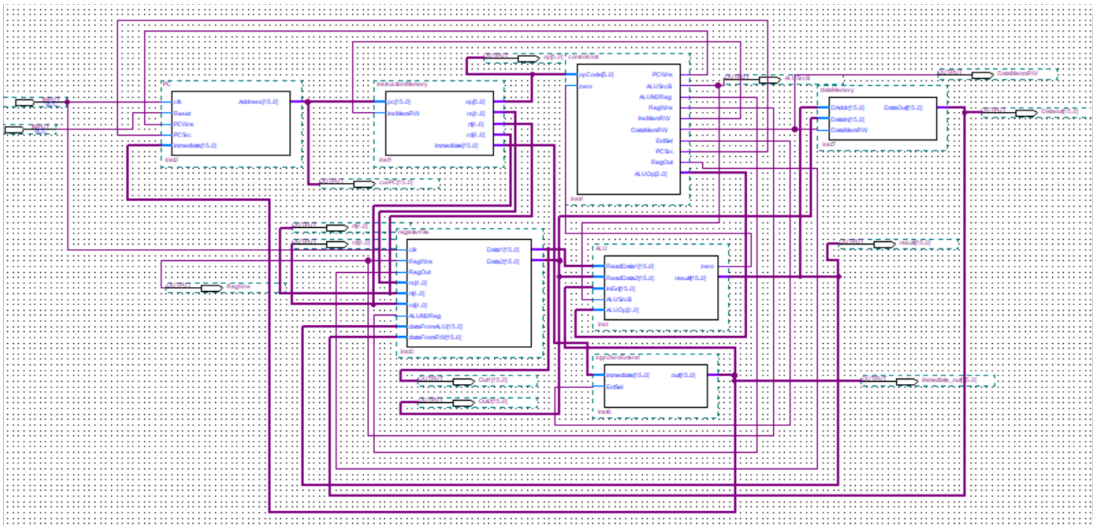

图 4.5: 顶层原理图

4.3.2 PC.v

根据上一条指令修改 PC，得到新的 PC 值，使得指令继续执行。

表 4.1: PC.v 输入输出信号功能

输入/输出	信号名	功能
输入	clk	时钟信号
输入	Reset	开关信号，为“0”不执行
输入	PCWre	P 为“1”时 PC 更改，P 为“0”时 PC 不更改
输入	PCSrc	为“1”代表跳转指令
输入	immediate	立即数，跳转指令可能会用到。
输出	Address	下一条执行指令地址

```
1  `timescale 1ns / 1ps
2  module PC(clk, Reset, PCWre, PCSrc, immediate, Address);
3          input clk, Reset, PCWre, PCSrc;
4          input [15:0] immediate;
5          output [15:0] Address;
6          reg [15:0] Address;
7
8          always @(posedge clk or negedge Reset)
9          begin
10                 if (Reset == 0) begin
11                         Address = 0;
12                 end
13                 else if (PCWre) begin
14                         if (PCSrc) Address = Address + 8 +
                               immediate*8;
15                         else Address = Address + 8;
16                 end
17         end
18 endmodule
```

4.3.3 instructionMemory.v

指令存储单元。根据当前的 PC 地址得到对应的 opCode，rs，rt，rd 以及 immediate。

表 4.2: instructionMemory.v 输入输出信号功能

输入/输出	信号名	功能
输入	pc	取值地址
输入	InsMemRW	读指令存储器 (Ins. Data)，初始化为 0。“1”时代表，写指令，暂时无此功能。
输出	op	操作码
输出	rs	第 1 个源操作数寄存器
输出	rt	第 2 个源操作数寄存器
输出	rd	目的操作数寄存器
输出	immediate	立即数

```
1  `timescale 1ns / 1ps
2  module instructionMemory(
3          input [15:0] pc,
4          input InsMemRW,
5          output [5:0] op,
6          output [4:0] rs, rt, rd,
7          output [15:0] immediate);
8          wire [15:0] mem[0:32];
9
10         //指令初始化
11
12         // output
13         assign op = mem[pc[6:2]][15:10];
14         assign rs = mem[pc[6:2]][9:5];
15         assign rt = mem[pc[6:2]][4:0];
16         assign rd = mem[pc[6:2]+1][15:11];
17         assign immediate = mem[pc[6:2]+1][15:0];
18 endmodule
```

4.3.4 registerFile.v

寄存器文件单元的功能是接收 instructionMemory 中的 rs，rt 等作为输入，输出对应寄存器的数据，从而达到取寄存器里的数据的目的。在其内部实现的过程中，为了防止 0 号寄存器写入数据需要在 writeReg 的时候多加入一个判断条件，即 writeReg 不等于 0 时写入数据。

表 4.3: registerFile.v 输入输出信号功能

输入/输出	信号名	功能
输入	clk	取值地址
输入	RegWre	“1”代表写寄存器
输入	RegOut	选择写 rd 或 rt
输入	rs	第 1 个源操作数寄存器
输入	rt	第 2 个源操作数寄存器
输入	rd	目的操作数寄存器
输入	immediate	立即数
输入	dataFromALU	来自 ALU 的计算结果 result
输入	dataFromRW	来自存储器的值
输出	Data1	rs 寄存器中的值
输出	Data2	rt 寄存器中的值

```
1  `timescale 1ns / 1ps
2  module registerFile(clk, RegWre, RegOut, rs, rt, rd, ALUM2Reg,
       dataFromALU, dataFromRW, Data1, Data2);
3          input clk, RegOut, RegWre, ALUM2Reg;
4          input [4:0] rs, rt, rd;
5          input [15:0] dataFromALU, dataFromRW;
6          output [15:0] Data1, Data2;
7          wire [4:0] writeReg;
8          wire [15:0] writeData;
9          assign writeReg = RegOut? rd : rt;
10         assign writeData = ALUM2Reg? dataFromRW : dataFromALU;
11
12         reg [15:0] register[0:31];
13         integer i;
14         initial begin
```

```
15                  register[0] = 0;
16          end
17          // output: 随register变化而变化
18          // Data1 为ALU运算时的A，当指令为sll时，A的值从立即数
                的16位中获得
19          // Data2 位ALU运算中的B，其值始终是为rt
20          assign Data1 = register[rs];
21          assign Data2 = register[rt];
22          // Write Reg
23          always @(posedge clk) begin
24          // writeReg != 0 是确保零号寄存器不会改变的作用
25                  if (writeReg && RegWre) begin
26                          register[writeReg] = writeData;
27                  end
28          end
29  endmodule
```

4.3.5 ALU.v

计算单元接收寄存器的数据和控制信号作为输入，将计算结果输出。

表 4.4: ALU.v 输入输出信号功能

输入/输出	信号名	功能
输入	ReadData1	rs 中的值
输入	ReadData2	rt 中的值
输入	inExt	立即数
输入	ALUSrcB	“0”代表 ReadData2 做运算，“1”代表立即数做运算
输入	ALUOp	计算功能
输出	zero	“1”代表结果为 0，“0”代表结果不为 0
输出	result	计算结果

```
1  `timescale 1ns / 1ps
2
3  module ALU(ReadData1, ReadData2, inExt, ALUSrcB, ALUOp, zero,
       result);
4          input [15:0] ReadData1, ReadData2, inExt;
5          input ALUSrcB;
6          input [2:0] ALUOp;
7          output zero;
8          output [15:0] result;
9          reg zero;
10         reg [15:0] result;
11         wire [15:0] B;
12         assign B = ALUSrcB? inExt : ReadData2;
13         always @(ReadData1 or ReadData2 or inExt or ALUSrcB or
               ALUOp or B)
14                 begin
15                         case(ALUOp)
16             //..根据ALUop 将ReadData1和B做运算
17                         //结果保存在result中，并根据结果修改
                               zero
18                         endcase
19                 end
20 endmodule
```

表 4.5: ALU 功能

ALUop	功能	描述
000	A+B	加
001	A-B	减
010	B-A	减
011	A B	或
100	A B	与

4.3.6 controlUnit.v

输入指令操作码，得到一系列控制信号。

表 4.6: controlUnit.v 输入输出信号功能

输入/输出	信号名	功能
输入	opCode	指令操作码
输入	zero	零位
输出	PCWre	“0”代表 PC 不更改，相关指令：halt
输出	ALUSrcB	“0”代表 ReadData2 做运算，“1”代表立即数做运算
输出	RegWre	“1”代表寄存器组写使能
输出	InsMemRW	“0”代表读指令存储器 (Ins. Data)，初始化为 0
输出	DataMemRW	“0”代表读数据存储器，“1”代表写数据存储器
输出	PCSrc	“1”代表跳转指令
输出	RegOut	“0”代表写寄存器组寄存器的地址来自 rt 字段，“1”代表写寄存器组寄存器的地址来自 rd 字段，
输出	ALUOp	ALU 运算功能选择

```
1  `timescale 1ns / 1ps
2  module controlUnit(opCode, zero, PCWre, ALUSrcB, ALUM2Reg,
       RegWre, InsMemRW, DataMemRW, ExtSel, PCSrc, RegOut, ALUOp);
3      input [5:0] opCode;
4          input zero;
5          output PCWre, ALUSrcB, ALUM2Reg, RegWre, InsMemRW,
               DataMemRW, ExtSel, PCSrc, RegOut;
6          output[2:0] ALUOp;
7          assign PCWre = (opCode == 6'b111111)? 0 : 1;
8          assign ALUSrcB = (opCode == 6'b000001 || opCode == 6'
               b010000 || opCode == 6'b100110 || opCode == 6'
               b100111)? 1 : 0;
9          assign ALUM2Reg = (opCode == 6'b100111)? 1 : 0;
10         assign RegWre = (opCode == 6'b100110 ||opCode==6'
```

```
            b110000 || opCode == 6'b111111)? 0 : 1;
11          assign InsMemRW = 0;
12          assign DataMemRW = (opCode == 6'b100110)? 1 : 0;
13          assign ExtSel = (opCode == 6'b010000)? 0 : 1;
14          assign PCSrc = ((opCode == 6'b110000 && zero == 1) ||
              opCode == 6'b110001)? 1 : 0;
15          assign RegOut = (opCode == 6'b000001 || opCode == 6'
              b010000 || opCode == 6'b100111)? 0 : 1;
16          assign ALUOp[2] = (opCode == 6'b010001)? 1 : 0;
17          assign ALUOp[1] = (opCode == 6'b010000 || opCode == 6
              'b010010)? 1 : 0;
18          assign ALUOp[0] = (opCode == 6'b000010 || opCode == 6
              'b010000 || opCode == 6'b010010 || opCode == 6'
              b110000)? 1 : 0;
19  endmodule
```

4.3.7 dataMemory.v

读写数据存储器。

表 4.7: dataMemory.v 输入输出信号功能

输入/输出	信号名	功能
输入	DAddr	数据寄存器的地址
输入	DataIn	将要写入数据寄存器的数据
输入	DataMemRW	“0”代表读数据，“1”代表写数据
输出	DataOut	根据地址读出的数据

```
1  `timescale 1ns / 1ps
2  module dataMemory(DAddr, DataIn, DataMemRW, DataOut);
3          input [15:0] DAddr, DataIn;
4          input DataMemRW;
5          output reg [15:0] DataOut;
6          reg[15:0] memory[31:0];
7
8          // read data
9          always @(DataMemRW) begin
10                 if (DataMemRW == 0) //assign
11                 DataOut = memory[DAddr];
12         end
13
14         // write data
15         integer i;
16         always @(DataMemRW or DAddr or DataIn)
17                 begin
18                         if (DataMemRW) memory[DAddr] = DataIn;
19                 end
20 endmodule
```

4.4 RTL 原理图

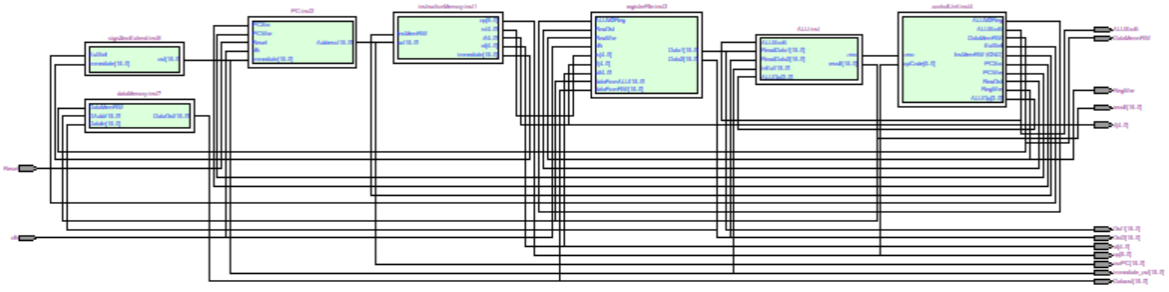

图 4.6: RTL 原理图

第 5 章　结果测试

5.1　测试指令集

表 5.1: 测试指令

汇编程序	op(6)	rs(5)	rt(5)	rd(5)/immediate(16)
ori $1,$0,6	010000	00000	00001	0000000000000110
addi $2,$0,9	000001	00000	00010	0000000000001001
beq $1,$2,3	110000	00001	00010	0000000000000011
sw $1,1($2)	100110	00010	00001	0000000000000001
lw $2,10($0)	100111	00001	00010	0000000000001010
beq $1,$2,-4	110000	00001	00010	1111111111111100
move $3,$2	100000	00001	00000	0001100000000000
add $4,$1,$3	000000	00001	00011	0010000000000000
sub $5,$2,$1	000010	00010	00001	0010100000000000
and $6,$3,$5	010001	00011	00101	0011000000000000
jmp 1	110001	00000	00000	0000000000000001
sub $3,$2,$1	000010	00010	00001	0001100000000000
add $4,$1,$3	000000	00001	00011	0010100000000000
halt	111111	00000	00000	0000000000000000

5.2　仿真波形

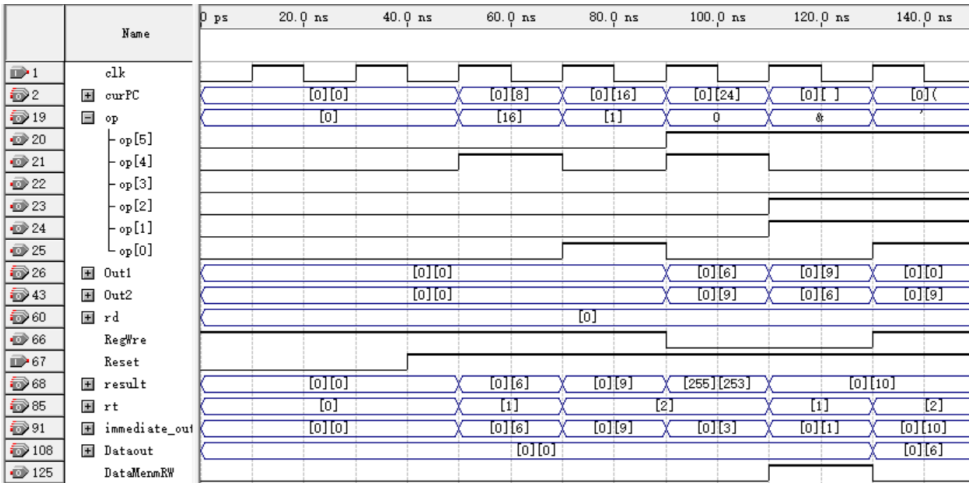

图 5.1: 仿真波形结果 1

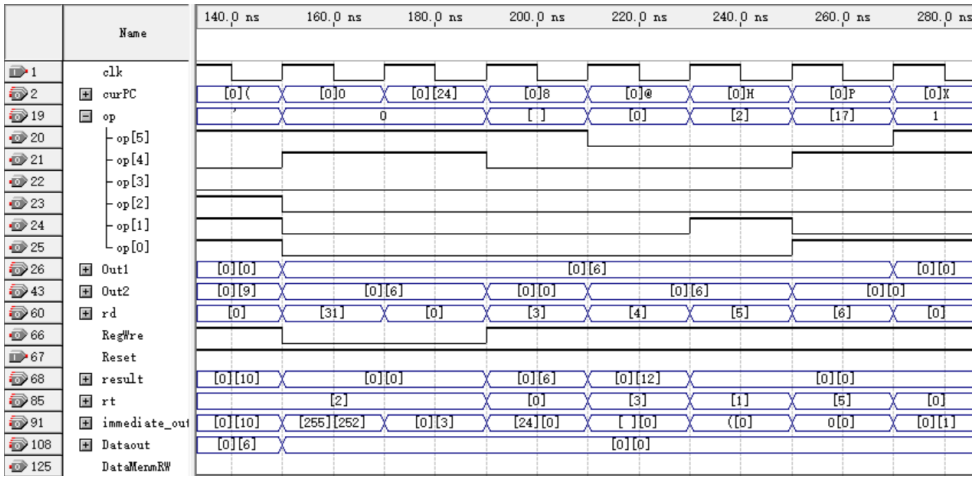

图 5.2: 仿真波形结果 2

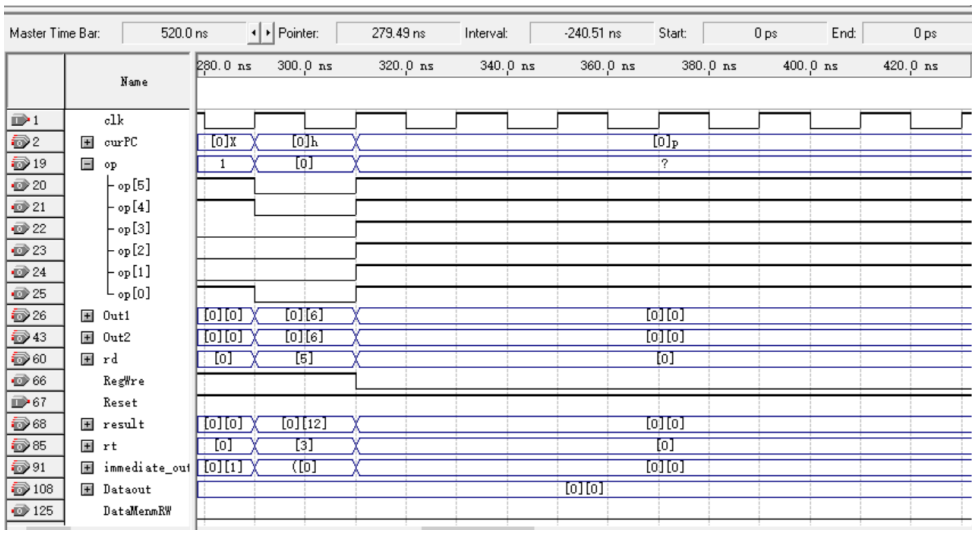

图 5.3: 仿真波形结果 3

第 6 章　总结

6.1　遇到的问题及解决

1. 不理解 wire reg 这些的具体用法，经常出现语法错误

   解决：Wire 主要起信号间连接作用，用以构成信号的传递或者形成组合逻辑。因为没有时序限定，wire 的赋值语句通常和其他 block 语句并行执行。Wire 不保存状态，它的值可以随时改变，不受时钟信号限制。除了可以在 module 内声明，所有 module 的 input 和 output 默认都是 wire 型的。

   Reg 是寄存器的抽象表达，作用类似通常编程语言中的变量，可以储存数值，作为参与表达式的运算，通常负责时序逻辑，以串行方式执行。Reg 可以保存输出状态。状态改变通常在下一个时钟信号边沿翻转时进行。

2. Out1 与 Out2 没有结果

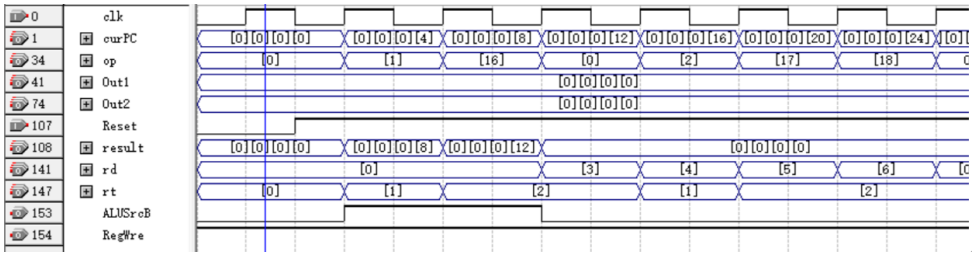

图 6.1

   解决：clk 忘记连上。连线还是应该要仔细。

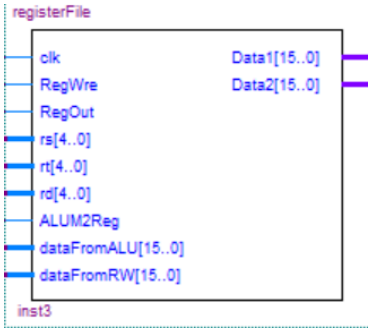

图 6.2

3. <= 和 = 的区别

   解决：<= 是非阻塞式赋值，= 是阻塞式赋值。当 = 这个赋值语句执行的时候是不允许有其它语句执行的，这就是阻塞的原因。而非阻塞赋值，例如 a<=b; 当这个赋值语句执行的时候是不阻碍其它语句执行的。

6.2 心得

经过本次实验，我运用到了数字电路和计算机组成原理的相关知识，极大地提高了我对书本上知识的理解和运用能力，并锻炼了动手能力。对 CPU 的指令执行流程有了更加深入的体会。

模型机的设计制作过程还算顺利，指令系统和线路图的设计花费了比较多的时间，尤其是线路布局很伤脑筋，但是总归是克服了困难，制作出这样一个简单的单周期单总线的模型机，如果以后有精力，我希望能继续尝试设计更先进的多周期或者流水线型 CPU 模型机，以进一步提高专业能力。

最后，想做一件事，就立马去做。你会发现没那么困难，往往最大的困难，是心里的困难。