硬件课程设计（Ⅰ）

单周期 CPU 设计报告

班级（班号） 9171062301

姓　　名 王烨欢

学　　号 917106840143

任 课 教 师 杜姗姗

南京理工大学计算机科学与工程学院

2020 年 9 月 18 日

目录

第一章 单周期 CPU 的设计原理

为实现单周期 CPU，本文首先研究了单周期 CPU 的相关理论和工作原理。本章首先简要介绍单周期 CPU 的基本概念，然后对 CPU 的工作原理进行简要分析，以便为单周期 CPU 的设计和开发提供理论基础。

1.1 单周期 CPU 概述

中央处理器，即 CPU，作为计算机系统的运算和控制核心，是信息处理、程序运行的最终执行单元。在 CPU 内部，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。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。

1.2 CPU 工作原理

CPU 在处理指令时，一般需要经过以下几个步骤：

(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。

CPU 的指令处理过程如下：

图 1 CPU 指令处理过程

单周期 CPU，是在一个时钟周期内完成这五个阶段的处理。

第二章 单周期 CPU 的设计内容

2.1 指令系统的设计

2.1.1 概述

本文所设计的单周期 CPU 的指令系统采用类似 MIPS 的设计风格，包括以下四类指令：

(1) 运算类指令；

(2) 传送类指令；

(3) 存储类指令；

(4) 控制类指令；

其中，所有指令的操作码部分用 4 位二进制表示，寄存器编号用 3 位二进制表示。在下述的具体设计表示中，以助记符表示的是汇编指令；以代码表示的则是二进制机器指令。

2.1.2 运算类指令的设计

运算类指令共有 8 条，具体的指令功能与指令格式如下：

(1) add rd,rs,rt

0000	rs(3 位)	rt(3 位)	rd(3 位)	reserved(未用)

功能：将寄存器 rs 和 rt 中的值相加并将结果写回寄存器 rd。

reserved 为预留部分，即未用，一般填“0”。

(2) addi rt,rs,immediate

0001	rs(3 位)	rt(3 位)	immediate(6 位)

功能：将立即数 immediate 进行符号扩展后再和寄存器 rs 中的值相

加并将结果写回寄存器 rt。

(3) sub rd,rs,rt

0010	rs(3 位)	rt(3 位)	rd(3 位)	reserved(未用)

功能：用寄存器 rs 中的值减去 rt 中的值并将结果写回寄存器 rd。reserved 为预留部分，即未用，一般填“0”。

(4) or rd,rs,rt

0011	rs(3 位)	rt(3 位)	rd(3 位)	reserved(未用)

功能：将寄存器 rs 和 rt 中的值进行按位或运算并将结果写回寄存器 rd。reserved 为预留部分，即未用，一般填“0”。

(5) ori rt,rs,immediate

0100	rs(3 位)	rt(3 位)	immediate(6 位)

功能：将立即数 immediate 进行“0”扩展后再和寄存器 rs 中的值进行按位或运算并将结果写回寄存器 rt。

(6) and rd,rs,rt

0101	rs(3 位)	rt(3 位)	rd(3 位)	reserved(未用)

功能：将寄存器 rs 和 rt 中的值进行按位与运算并将结果写回寄存器 rd。reserved 为预留部分，即未用，一般填“0”。

(7) sll rd,rt,sa

0110	未用(3 位)	rt(3 位)	rd(3 位)	sa(3 位)

功能：先将 sa 表示的立即数进行“0”拓展，然后将寄存器 rt 中的值左移 sa 位并将结果写回寄存器 rd。

(8) slt rd,rs,rt

0111	rs(3 位)	rt(3 位)	rd(3 位)	reserved(未用)

功能：将寄存器 rs 和 rt 中的值作为带符号数进行比较，若 rs 中的值更小，则将寄存器 rd 中的值置 1，否则置 0。

2.1.3 传送类指令的设计

传送类指令共有 2 条，具体的指令功能与指令格式如下：

(1) mov rd,rt

1000	未用(3 位)	rt(3 位)	rd(3 位)	reserved(未用)

功能：将寄存器 rt 中的值传送到寄存器 rd。reserved 为预留部分，即未用，一般填“0”。

(2) movi rt,immediate

1001	未用(3 位)	rt(3 位)	immediate(6 位)

功能：将立即数 immediate 进行符号扩展后传送到寄存器 rt。

2.1.4 存储类指令的设计

存储类指令共有 2 条，具体的指令功能与指令格式如下：

(1) sw rt,immediate(rs)

1010	rs(3 位)	rt(3 位)	immediate(6 位)

功能：将寄存器 rt 中的值存储到内存中，存储单元的地址为寄存器 rs 中的值加上经符号拓展的立即数 immediate。

(2) lw rt,immediate(rs)

1011	rs(3 位)	rt(3 位)	immediate(6 位)

功能：将内存中的值加载到寄存器 rt 中，存储单元的地址为寄存器 rs 中的值加上经符号拓展的立即数 immediate。

2.1.5 控制类指令的设计

控制类指令共有 4 条，具体的指令功能与指令格式如下：

(1) beq rs,rt,immediate

1100	rs(3 位)	rt(3 位)	immediate(6 位)

功能：比较寄存器 rs 和 rt 中的值，若相等则跳转到目标处执行，否则继续执行下一条指令。跳转的目标地址与当前指令的下一条指令的相对位移由立即数 immediate 给出。

(2) bgtz rs,immediate

1101	rs(3 位)	000	immediate(6 位)

功能：判断寄存器 rs 中的值是否大于 0，若大于 0 则跳转到目标处执行，否则继续执行下一条指令。跳转的目标地址与当前指令的下一条指令的相对位移由立即数 immediate 给出。

(3) j addr

1110	addr(12 位)

功能：执行无条件跳转，跳转的目的地址组成如下：高 3 位为 PC 中值的高 3 位，第 1-12 位由 addr 给出，最低位为 0。

(4) halt

1111	000000000000(12 位)

功能：停机指令，不改变 PC 的值，PC 保持不变。

2.2 整体框架的设计

本文所设计的单周期 CPU 的整体框架主要包括七部分：程序计数器、指令寄存器、寄存器组、算术逻辑单元、数据存储器、控制单元和顶层模块。具体框架如下：

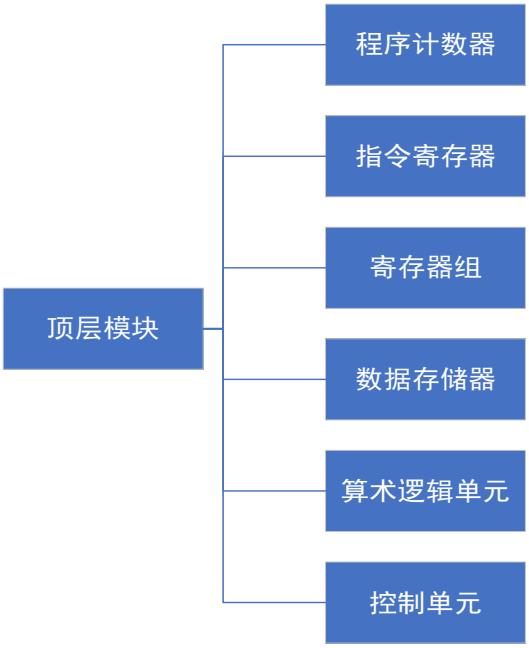


图 2 整体框架图

2.3 数据通路的设计

本文所设计的单周期 CPU 的数据通路如下：

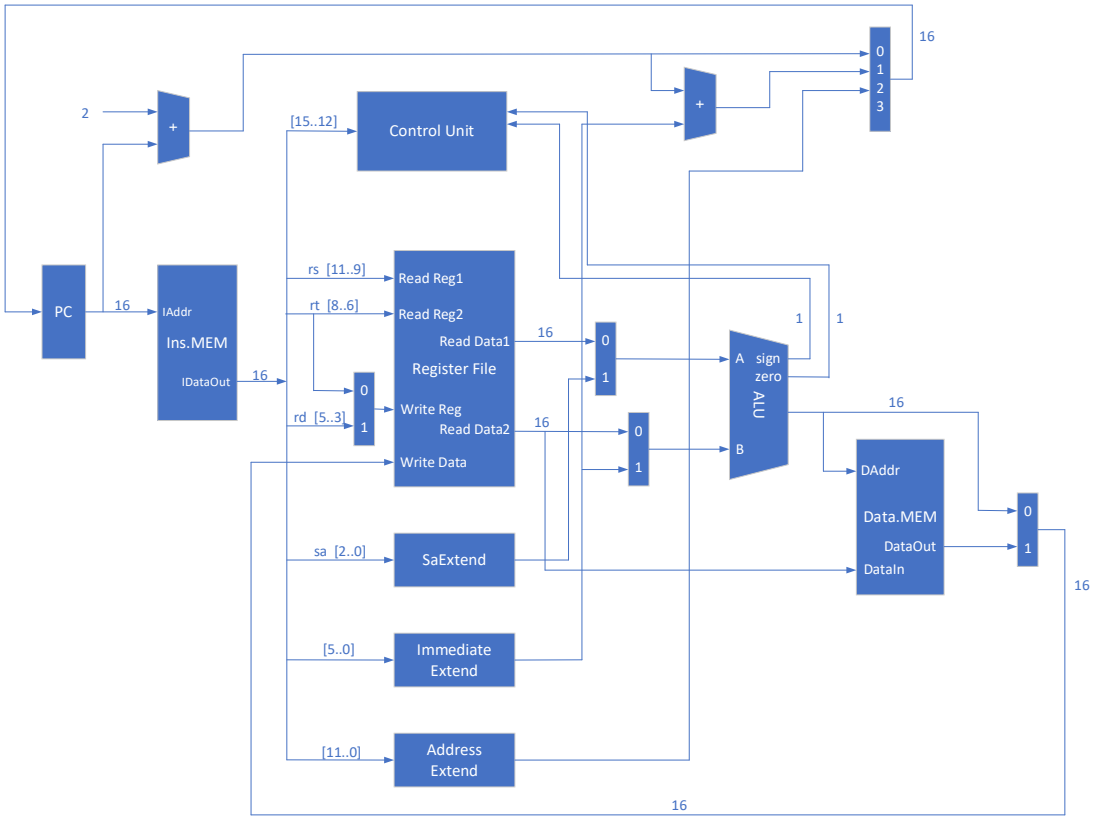


图 3 数据通路图

其中 Ins.MEM 为指令存储器，Data.MEM 为数据存储器。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 Register File，先给出寄存器编号，读操作时，输出端就直接输出相应数据；而在写操作时，当写使能信号为 1 时，在时钟边沿触发将数据写入寄存器。

主要模块接口说明如下：

(1) Instruction Memory：指令存储器。

IAddr，指令存储器地址输入端口

IDataOut，指令存储器数据输出端口（指令代码输出端口）

InsMemRW，指令存储器读写控制信号，为 0 写，为 1 读

(2) Data Memory：数据存储器。

DAddr，数据存储器地址输入端口

DataIn，数据存储器数据输入端口

DataOut，数据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

(3) 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端口，其地址来源为 rt 或 rd 字段

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

(4) ALU：算术逻辑单元。

A，操作数 A 输入端口

B，操作数 B 输入端口

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

2.4 控制信号的设计

本文所设计的单周期 CPU 的各控制信号如下：

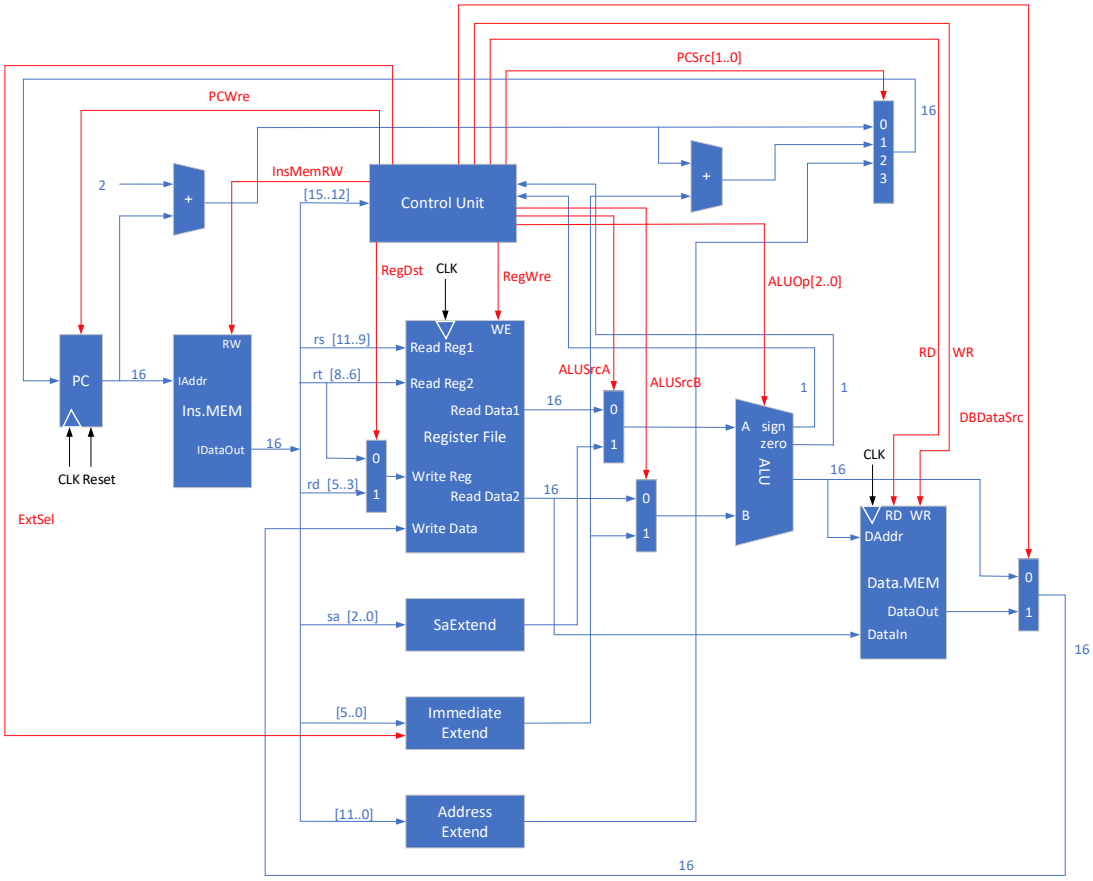


图 4 控制信号图

各控制信号的功能如下：

控制信号名	状态“0”	状态“1”
Reset	初始化 PC 为 0	PC 接收新地址
PCWre	PC 不更改，相关指令：halt	PC 更改，相关指令：除指令 halt 外
ALUSrcA	来自寄存器堆 data1 输出，相关指令：add、addi、sub、or、ori、and、slt、mov、movi、beq、bgtz、sw、lw	来自经过“0”拓展的移位数 sa，即{{13{0}},sa}，相关指令：sll
ALUSrcB	来自寄存器堆 data2 输出，相关指令：add、sub、or、and、sll、slt、mov、beq、bgtz	来自 sign 或 zero 扩展的立即数，相关指令：addi、ori、movi、sw、lw
DBDataSrc	来自 ALU 运算结果的输出，相关指令：add、addi、sub、or、ori、and、sll、slt、mov、movi	来自数据存储器（Data MEM）的输出，相关指令：lw
RegWre	寄存器组写不使能，相关指令：sw、beq、bgtz、j、halt	寄存器组写使能，相关指令：add、addi、sub、or、ori、and、sll、slt、mov、movi、lw
InsMemRW	写指令存储器	读指令存储器
RD	读数据存储器，相关指令：lw	无操作

WR	写数据存储器，相关指令：sw	无操作
RegDst	写寄存器组寄存器的地址，来自 rt 字段，相关指令：addi、ori、movi、lw	写寄存器组寄存器的地址，来自 rd 字段，相关指令：add、sub、or、and、sll、slt、mov
ExtSel	(zero-extend)**immediate**（0 扩展），相关指令：ori	(sign-extend)**immediate**（符号扩展），相关指令：addi、movi、sw、lw、beq、bgtz
PCSrc[1..0]	00：pc<－pc+2，相关指令：add、addi、sub、or、ori、and、sll、slt、sw、lw、beq(zero=0)、bgtz(sign=1，或 zero=1)； 01：pc<－pc+2+(sign-extend)**immediate**，相关指令：beq(zero=1)、bgtz(sign=0，zero=0)； 10：pc<－{(pc+2)[15..13],addr[12..1],0}，相关指令：j； 11：未用	
ALUOp[2..0]	ALU 8 种运算功能选择(000-111)，具体见 ALU 功能表	

ALU 运算功能表如下：

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

110	if (A<B && (A[15] == B[15]))
	Y = 1;
	else if (A[15] && !B[15] Y = 1;
	else Y = 0;
		比较 A 与 B
		带符号

111	Y = B	直送

在具体进行控制时，PC 的改变是在时钟上升沿进行的，指令执行的结果总是在时钟下降沿保存到寄存器和存储器中，这样稳定性较好。

第三章 单周期 CPU 的具体实现

3.1 底层模块的实现

单周期 CPU 每个时钟周期可被划分为五个阶段，对应的有五个最关键的底层模块：IF 阶段对应 PC，ID 阶段对应 InstructionMemory 和 RegisterFile，EXE 阶段对应 ALU，MEM 阶段对应 DataMemory，WB 阶段也对应 RegisterFile。除此之外，还需要生成控制信号的控制单元（ControlUnit）等。下面具体介绍上述各主要模块的设计与实现。

3.1.1 程序计数器 PC

PC 为时序逻辑，在时钟上升沿到来时，若 PCWre 为 1，则输出下一条待处理的指令地址，否则不输出新的指令地址。若 Reset 为 0，则设置输出的指令地址为 0。具体代码如下：

```
1.  module PC(
2.      input clk,
3.      input [15:0] PCin,
4.      input PCWre,
5.      input Reset,
6.      output reg [15:0] PCout
7.  );
8.
9.      initial begin
10.         PCout <= 0;
11.     end
12.
13.     always@(posedge clk) begin
14.         if(Reset == 0) begin
15.             PCout <= 0;
16.         end
17.         else if(PCWre == 0) begin
18.             PCout <= PCout;
```

```
        end
        else begin
            PCout <= PCin;
        end
    end

endmodule
```

3.1.2 指令存储器 InstructionMemory

InstructionMemory 为组合逻辑，内含 128 个字节的 mem。mem 中存放要执行的测试程序段的机器码，使用 initial 语句中的 $readmemb 伪指令从 Instructions.txt 文件中读取出来。当输入一个 16 位长的指令地址，输出对应地址中的 16 位机器指令。具体代码如下：

```
module InsMemory(
    input InsMemRW,
    input [15:0] address,
    output reg [15:0] DataOut
);

    reg [7:0] mem [0:127];

    initial begin
        DataOut = 16'b1111000000000000;
        $readmemb("Instructions.txt", mem);
    end

    always@(*) begin
        DataOut[15:8] <= mem[address];
        DataOut[7:0] <= mem[address+1];
    end

endmodule
```

3.1.3 寄存器组 RegisterFile

RegisterFile 中定义了一个含有 8 个寄存器的寄存器组，设计逻辑包括两部分：一方面，读寄存器组为组合逻辑，当输入一个 3 位长的寄存器编号时，输出对应寄存器中的数值。另一方面，写寄存器组是时序逻辑，在时钟下降沿到来时触发写入。具体代码如下：

```
1.  module RegFile(
2.      input CLK,
3.      input RST,
4.      input RegWre,
5.      input [2:0] ReadReg1,
6.      input [2:0] ReadReg2,
7.      input [2:0] WriteReg,
8.      input [15:0] WriteData,
9.      output [15:0] ReadData1,
10.     output [15:0] ReadData2
11. );
12.
13.     reg [15:0] regFile[0:7];
14.
15.     integer i;
16.
17.     assign ReadData1 = regFile[ReadReg1];
18.     assign ReadData2 = regFile[ReadReg2];
19.
20.     always @ (negedge CLK) begin
21.         if (RST == 0) begin
22.             for(i=1;i<8;i=i+1)
23.                 regFile[i] <= 0;
24.         end
25.         else if(RegWre == 1 && WriteReg != 0) begin
26.             regFile[WriteReg] <= WriteData;
27.         end
28.     end
29.
30. endmodule
```

3.1.4 算术逻辑单元 ALU

ALU 为组合逻辑，根据控制信号对两个操作数进行相应的运算，并输出结果。具体代码如下：

```
1.  module ALU(
2.      input [2:0] ALUopcode,
3.      input [15:0] rega,
4.      input [15:0] regb,
5.      output reg [15:0] result,
6.      output zero,
7.      output sign
8.  );
9.
10.     assign zero = (result==0)?1:0;
11.     assign sign = result[15];
12.
13.     always @( ALUopcode or rega or regb ) begin
14.         case (ALUopcode)
15.             3'b000 : result = rega + regb;
16.             3'b001 : result = rega - regb;
17.             3'b010 : result = regb << rega;
18.             3'b011 : result = rega | regb;
19.             3'b100 : result = rega & regb;
20.             3'b101 : result = (rega < regb)?1:0;
21.             3'b110 : begin
22.                 if (rega<regb &&(( rega[15] == 0 && regb[15]==0) ||
23.                 (rega[15] == 1 && regb[15]==1))) result = 1;
24.                 else if (rega[15] == 0 && regb[15]==1) result = 0;
25.                 else if ( rega[15] == 1 && regb[15]==0) result = 1;
26.                 else result = 0;
27.             end
28.             3'b111 : result = regb;
29.         endcase
30.     end
31.
32. endmodule
```

3.1.5 数据存储器 DataMemory

DataMemory 中包含 128 个字节的 ram，设计逻辑包括两部分：一方面，读存储器为组合逻辑，当输入一个 16 位长的数据地址时，输出对应地址中的 16 位数据。另一方面，写存储器是时序逻辑，在时钟下降沿到来时触发写入。具体代码如下：

```
1.  module DataMemory(
2.      input clk,
3.      input [15:0] address,
4.      input RD,
5.      input WR,
6.      input [15:0] DataIn,
7.      output [15:0] DataOut
8.  );
9.
10.     reg [7:0] ram[0:127];
11.
12.     integer i;
13.
14.     initial begin;
15.         for(i=0;i<128;i=i+1)
16.             ram[i]<=0;
17.     end
18.
19.     assign DataOut[7:0] = (RD == 0)? ram[address+1]:8'bz;
20.     assign DataOut[15:8] = (RD == 0)? ram[address]:8'bz;
21.
22.     always@(negedge clk) begin
23.         if(WR == 0) begin
24.             if(address>=0 && address<128) begin
25.                 ram[address] <= DataIn[15:8];
26.                 ram[address+1] <= DataIn[7:0];
27.             end
28.         end
29.     end
30.
31. endmodule
```

3.1.6 控制单元 ControlUnit

ControlUnit 为组合逻辑，将机器码中的操作码（opcode）转换为各个控制信号，从而控制不同的指令在不同的数据通路中传输。具体代码如下：

```
1.  module ControlUnit(
2.      input [3:0] opcode,
3.      input zero,
4.      input sign,
5.      output reg PCWre,
6.      output reg ALUSrcA,
7.      output reg ALUSrcB,
8.      output reg DBDataSrc,
9.      output reg RegWre,
10.     output reg InsMemRW,
11.     output reg RD,
12.     output reg WR,
13.     output reg RegDst,
14.     output reg ExtSel,
15.     output reg [1:0] PCSrc,
16.     output reg [2:0] ALUOp
17. );
18.
19.     initial begin
20.         RD = 1;
21.         WR = 1;
22.         RegWre = 0;
23.         PCWre = 0;
24.         InsMemRW = 1;
25.     end
26.
27.     always@ (opcode) begin
28.         case(opcode)
29.             4'b0000:begin // add
30.                 PCWre = 1;
31.                 ALUSrcA = 0;
32.                 ALUSrcB = 0;
33.                 DBDataSrc = 0;
34.                 RegWre = 1;
35.                 InsMemRW = 1;
```

```
36.                 RD = 1;
37.                 WR = 1;
38.                 RegDst = 1;
39.                 ALUOp = 3'b000;
40.             end
41.             4'b0001:begin //addi
42.                 PCWre = 1;
43.                 ALUSrcA = 0;
44.                 ALUSrcB = 1;
45.                 DBDataSrc = 0;
46.                 RegWre = 1;
47.                 InsMemRW = 1;
48.                 RD = 1;
49.                 WR = 1;
50.                 RegDst = 0;
51.                 ExtSel = 1;
52.                 ALUOp = 3'b000;
53.             end
54.             4'b0010:begin //sub
55.                 PCWre = 1;
56.                 ALUSrcA = 0;
57.                 ALUSrcB = 0;
58.                 DBDataSrc = 0;
59.                 RegWre = 1;
60.                 InsMemRW = 1;
61.                 RD = 1;
62.                 WR = 1;
63.                 RegDst = 1;
64.                 ALUOp = 3'b001;
65.             end
66.             4'b0011:begin // or
67.                 PCWre = 1;
68.                 ALUSrcA = 0;
69.                 ALUSrcB = 0;
70.                 DBDataSrc = 0;
71.                 RegWre = 1;
72.                 InsMemRW = 1;
73.                 RD = 1;
74.                 WR = 1;
75.                 RegDst = 1;
76.                 ALUOp = 3'b011;
77.             end
78.             4'b0100:begin // ori
79.                 PCWre = 1;
```

```
                ALUSrcA = 0;
                ALUSrcB = 1;
                DBDataSrc = 0;
                RegWre = 1;
                InsMemRW = 1;
                RD = 1;
                WR = 1;
                RegDst = 0;
                ExtSel = 0;
                ALUOp = 3'b011;
            end
            4'b0101:begin //and
                PCWre = 1;
                ALUSrcA = 0;
                ALUSrcB = 0;
                DBDataSrc = 0;
                RegWre = 1;
                InsMemRW = 1;
                RD = 1;
                WR = 1;
                 RegDst = 1;
                ALUOp = 3'b100;
             end
             4'b0110:begin //sll
                 PCWre = 1;
                 ALUSrcA = 1;
                 ALUSrcB = 0;
                 DBDataSrc = 0;
                 RegWre = 1;
                 InsMemRW = 1;
                 RD = 1;
                 WR = 1;
                 RegDst = 1;
                 ALUOp = 3'b010;
             end
             4'b0111:begin //slt
                 PCWre = 1;
                 ALUSrcA = 0;
                 ALUSrcB = 0;
                 DBDataSrc = 0;
                 RegWre = 1;
                 InsMemRW = 1;
                 RD = 1;
                 WR = 1;
```

```
124.                RegDst = 1;
125.                ALUOp = 3'b110;
126.            end
127.            4'b1000:begin //mov
128.                PCWre = 1;
129.                ALUSrcA = 0;
130.                ALUSrcB = 0;
131.                DBDataSrc = 0;
132.                RegWre = 1;
133.                InsMemRW = 1;
134.                RD = 1;
135.                WR = 1;
136.                RegDst = 1;
137.                ExtSel = 1;
138.                ALUOp = 3'b111;
139.            end
140.            4'b1001:begin //movi
141.                PCWre = 1;
142.                ALUSrcA = 0;
143.                ALUSrcB = 1;
144.                DBDataSrc = 0;
145.                RegWre = 1;
146.                InsMemRW = 1;
147.                RD = 1;
148.                WR = 1;
149.                RegDst = 0;
150.                ExtSel = 1;
151.                ALUOp = 3'b111;
152.            end
153.            4'b1010:begin //sw
154.                PCWre = 1;
155.                ALUSrcA = 0;
156.                ALUSrcB = 1;
157.                RegWre = 0;
158.                InsMemRW = 1;
159.                RD = 1;
160.                WR = 0;
161.                ExtSel =1;
162.                ALUOp = 3'b000;
163.            end
164.            4'b1011:begin //lw
165.                PCWre = 1;
166.                ALUSrcA = 0;
167.                ALUSrcB = 1;
```

```
168.                 DBDataSrc = 1;
169.                 RegWre = 1;
170.                 InsMemRW = 1;
171.                 RD = 0;
172.                 WR = 1;
173.                 RegDst = 0;
174.                 ExtSel = 1;
175.                 ALUOp = 3'b000;
176.             end
177.             4'b1100:begin //beq
178.                 PCWre = 1;
179.                 ALUSrcA = 0;
180.                 ALUSrcB = 0;
181.                 RegWre = 0;
182.                 InsMemRW = 1;
183.                 RD = 1;
184.                 WR = 1;
185.                 ExtSel = 1;
186.                 ALUOp = 3'b001;
187.             end
188.             4'b1101:begin  //bgtz
189.                 PCWre = 1;
190.                 ALUSrcA = 0;
191.                 ALUSrcB = 0;
192.                 RegWre = 0;
193.                 InsMemRW = 1;
194.                 RD = 1;
195.                 WR = 1;
196.                 ExtSel = 1;
197.                 ALUOp = 3'b001;
198.             end
199.             4'b1110:begin //j
200.                 PCWre = 1;
201.                 RegWre = 0;
202.                 InsMemRW = 1;
203.                 RD = 1;
204.                 WR = 1;
205.                 ALUOp = 3'b010;
206.             end
207.             4'b1111:begin //halt
208.                 PCWre = 0;
209.                 RegWre = 0;
210.                 InsMemRW = 1;
211.                 RD = 1;
```

```
212.                 WR = 1;
213.             end
214.             default:begin
215.                 RD = 1;
216.                 WR = 1;
217.                 RegWre = 0;
218.                 InsMemRW = 0;
219.             end
220.         endcase
221.     end
222.
223.     always@(opcode or zero or sign) begin
224.         if(opcode == 4'b1110) // j
225.             PCSrc = 2'b10;
226.         else if(opcode == 4'b1100) begin
227.             if(zero == 1)
228.                 PCSrc = 2'b01;
229.             else
230.                 PCSrc = 2'b00;
231.         end
232.         else if(opcode == 4'b1101) begin
233.             if(zero == 0 && sign == 0)
234.                 PCSrc = 2'b01;
235.             else
236.                 PCSrc = 2'b00;
237.         end
238.         else begin
239.             PCSrc = 2'b00;
240.         end
241.     end
242.
243. endmodule
```

3.2 顶层模块的实现

在顶层模块中，通过实例化各个底层模块，并使用导线将它们按照数据通路图连接起来，构成单周期 CPU 的完整结构，具体原理图如下：

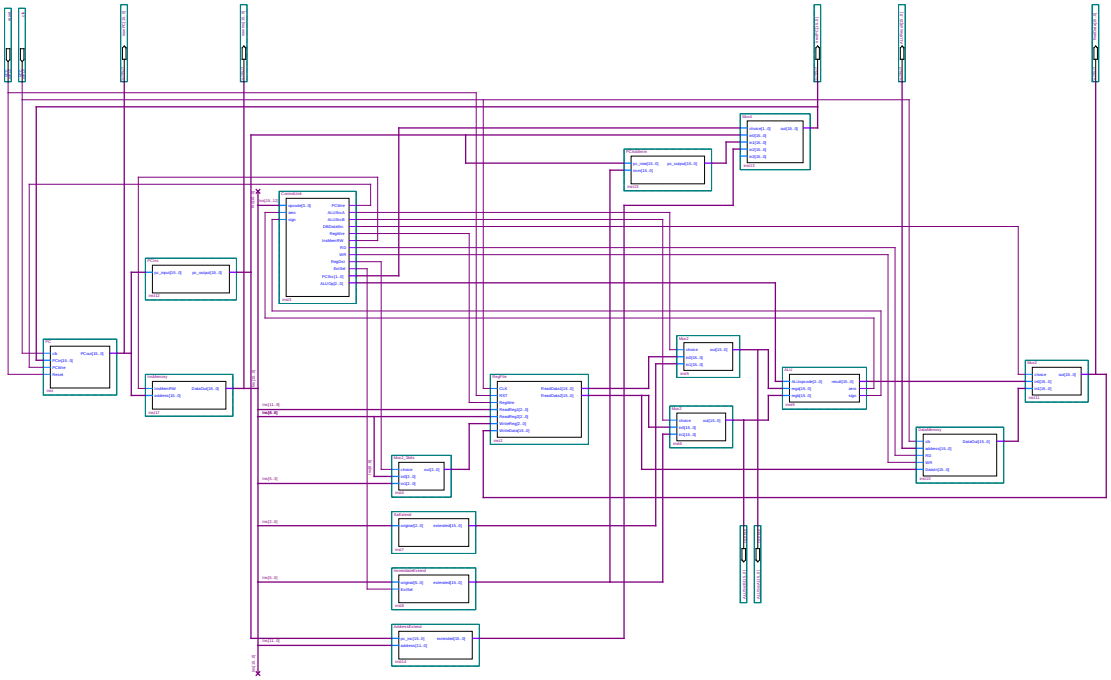


图 5 顶层原理图

第四章 单周期 CPU 的仿真验证

根据上述实现的单周期 CPU，利用指令系统中所有的指令设计一个测试程序，对该 CPU 进行如下仿真验证。

4.1 测试程序

使用该指令系统中所有指令，设计出如下一段具有实际意义的测试程序。该程序计算数字 1 到 5 的和以及按位或的和，并将两者中较大者存储到内存中地址为 14 的存储单元中，最后读出该存储单元中的值。具体指令序列如下：

地址	汇编程序		机器指令代码（二进制）
0x0000	movi	$0,0	10010000 00000000
0x0002	mov	$1,$0	10000000 00001000
0x0004	and	$3,$1,$0	01010010 00011000
0x0006	addi	$2,$0,6	00010000 10000110
0x0008	ori	$3,$0,1	01000000 11000001
0x000A	add	$5,$0,$3	00000000 11101000
0x000C	or	$6,$1,$3	00110010 11110000
0x000E	mov	$0,$5	10000001 01000000
0x0010	mov	$1,$6	10000001 10001000
0x0012	addi	$4,$3,1	00010111 00000001
0x0014	mov	$3,$4	10000001 00011000
0x0016	beq	$3,$2,1	11000110 10000001
0x0018	j	5	11100000 00000101
0x001A	slt	$5,$0,$1	01110000 01101000
0x001C	movi	$4,4	10010001 00000100
0x001E	sub	$6,$2,$4	00100101 00110000
0x0020	sll	$7,$6,1	01100001 10111001
0x0022	bgtz	$5,2	11011010 00000010
0x0024	sw	$0,10($7)	10101110 00001010
0x0026	j	21	11100000 00010101
0x0028	sw	$1,10($7)	10101110 01001010
0x002A	lw	$6,10($7)	10111111 10001010
0x002C	halt		11110000 00000000

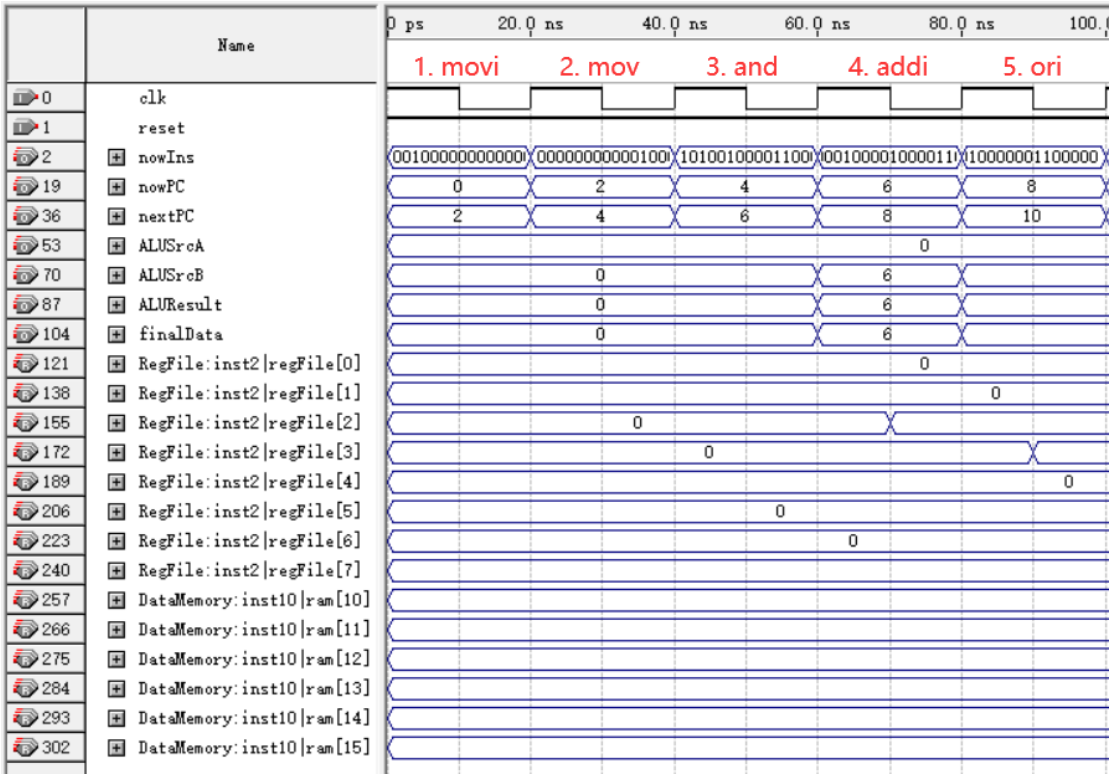
4.2 波形仿真

在仿真过程中，设置如下输入和输出，并显示如下寄存器中的内容：

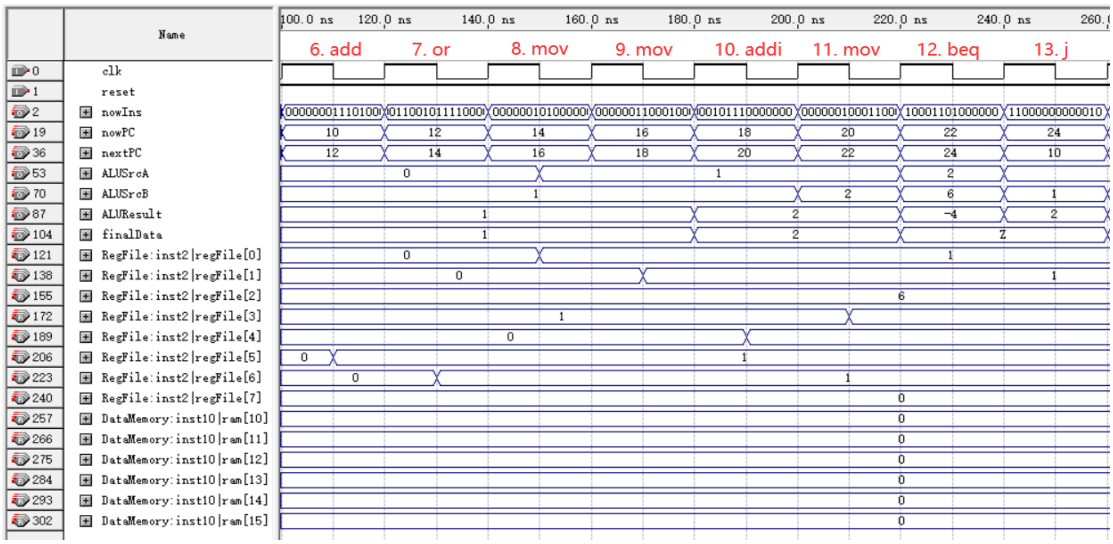
输入	clk	CPU 时钟信号，周期为 10ns
	reset	CPU 复位信号，0 为复位
输出	nowIns	当前指令
	nextPC	下一条指令地址
	nowPC	当前 PC 地址
	ALUSrcA	ALU 操作数 A
	ALUSrcB	ALU 操作数
	ALUResult	ALU 运算结果
	finalData	写回的结果
寄存器	RegFile[0]-RegFile[7]	寄存器组
	DataMemory[10]-DataMemory[15]	数据存储器

仿真结果如下：

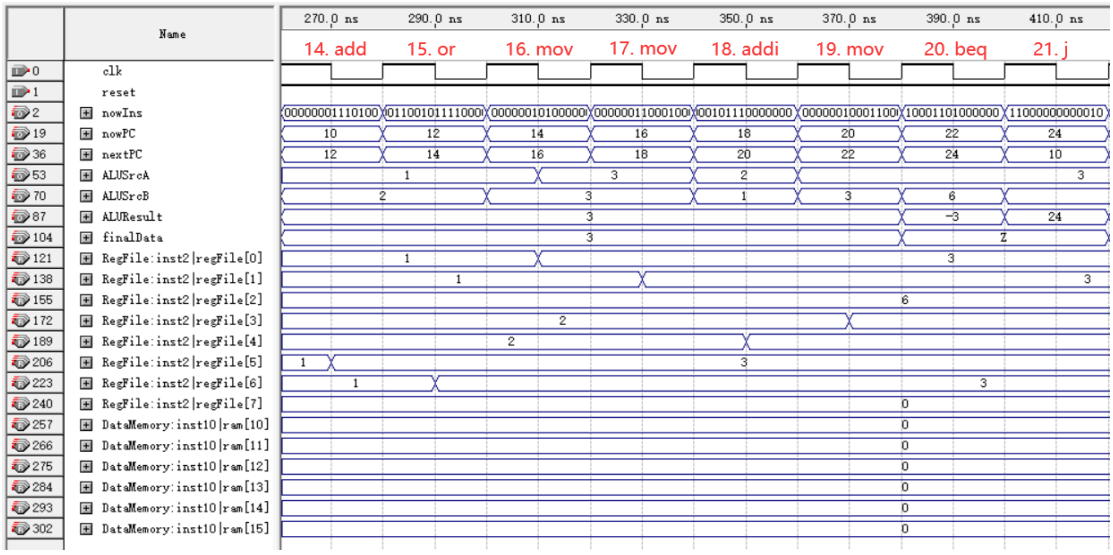
(1) movi $0,0 至 ori $3,$0,1



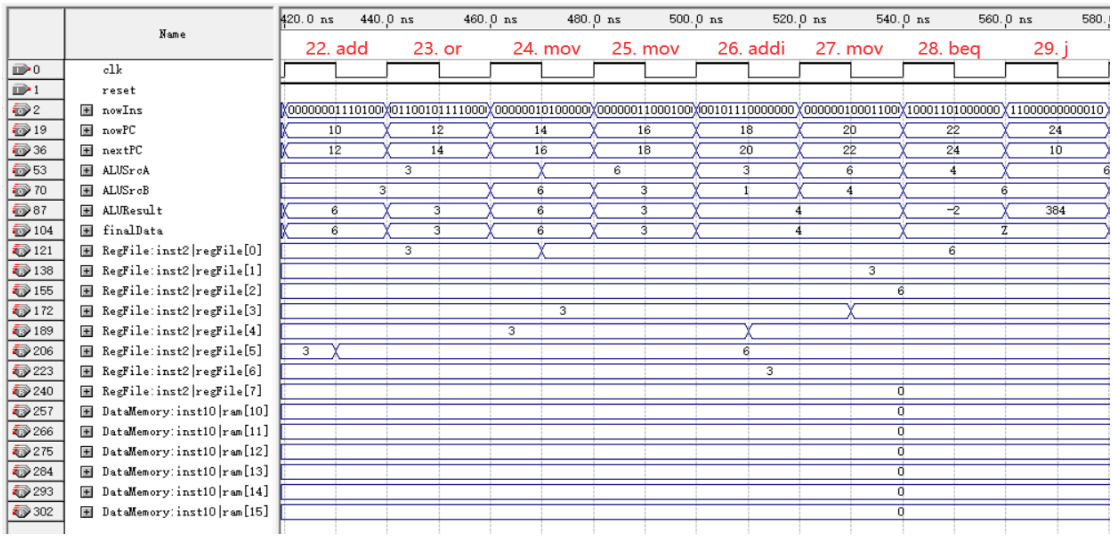
(2) add $5,$0,$3 至 j 5（第一轮循环）



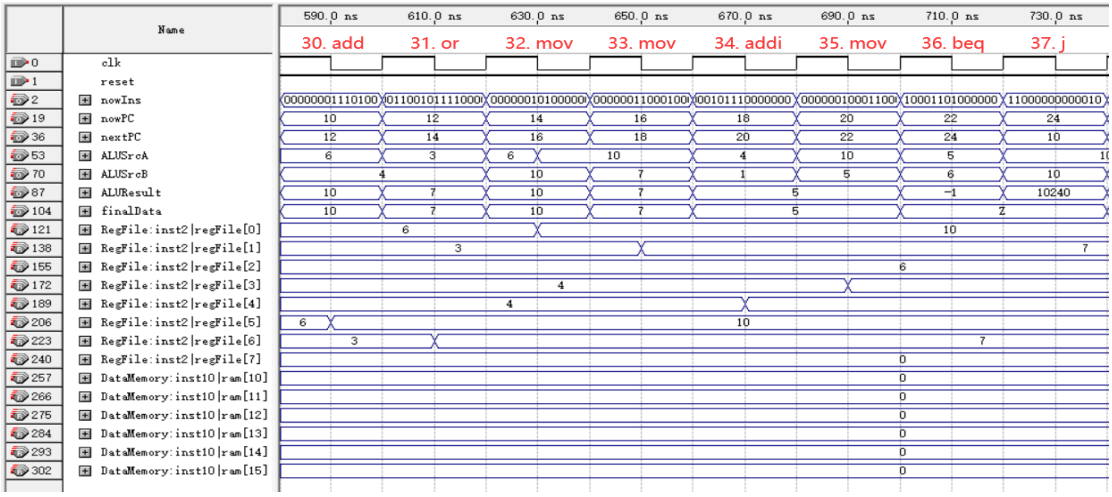
(3) add $5,$0,$3 至 j 5（第二轮循环）



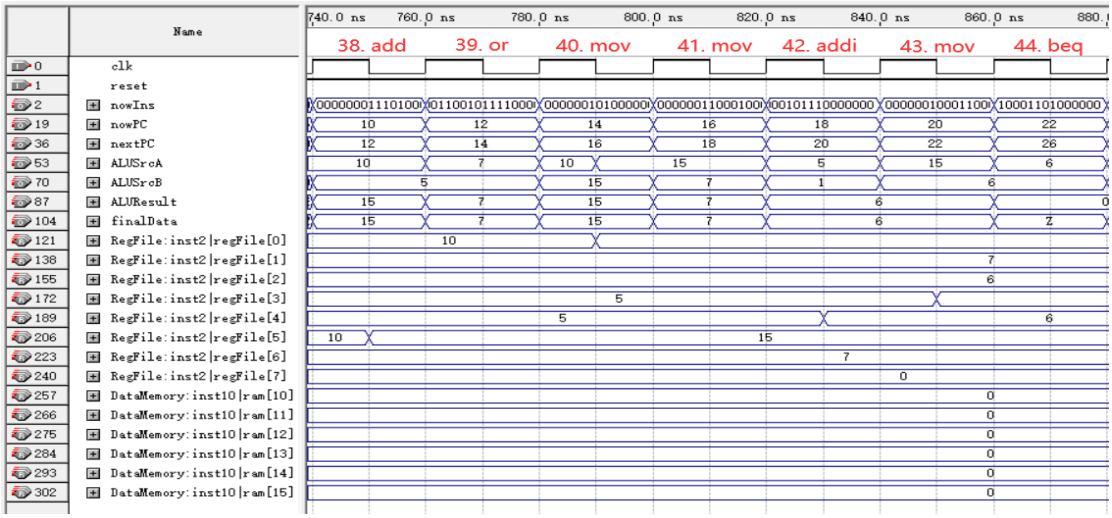
(4) add $5,$0,$3 至 j 5（第三轮循环）



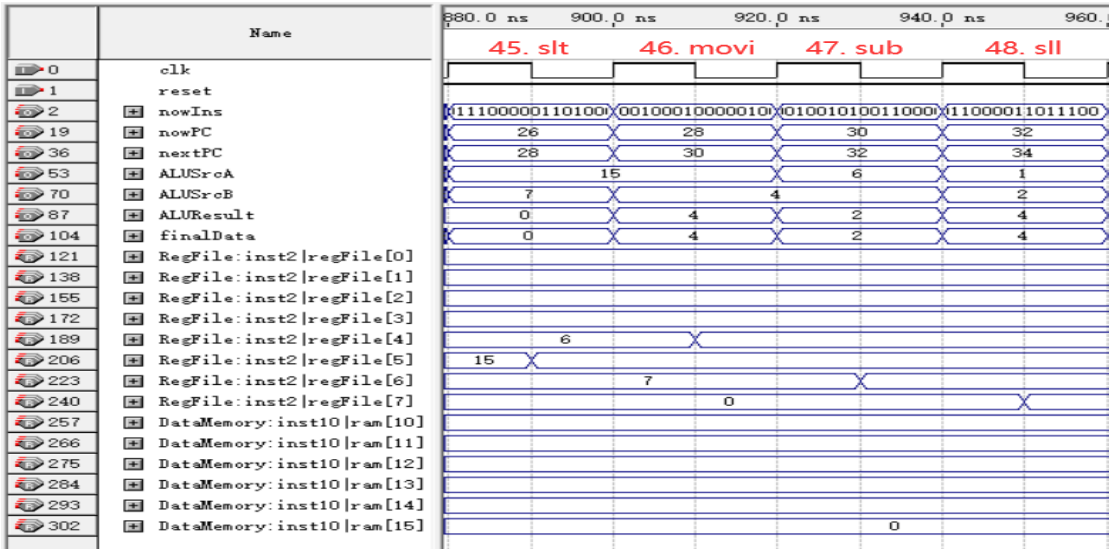
(5) add $5,$0,$3 至 j 5（第四轮循环）



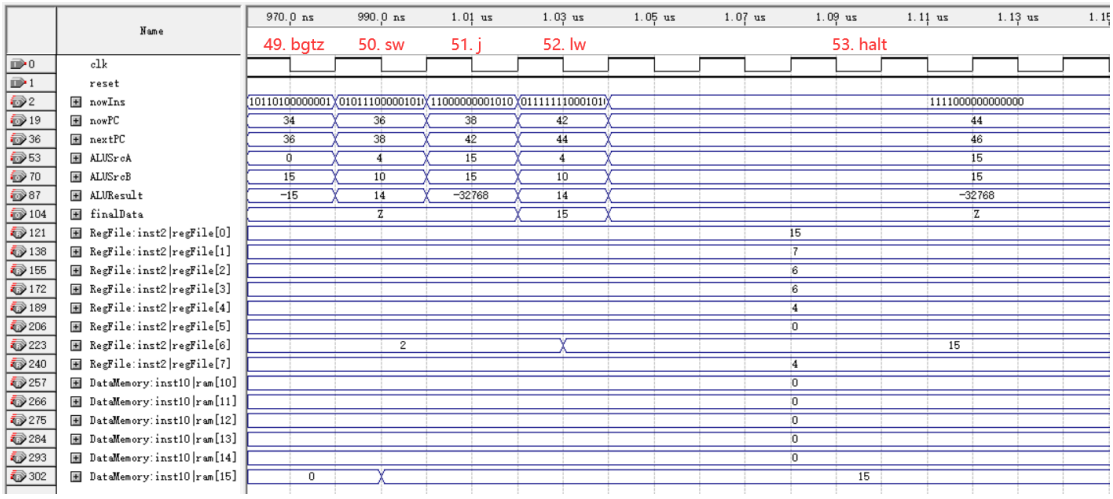
(6) add $5,$0,$3 至 beq $3,$2,1（第五轮循环）



(7) slt $5,$0,$1 至 sll $7,$6,1



(8) bgtz $5,2 至 halt



第五章 总结与心得

5.1 遇到的问题与解决方法

(1) 问题：

在设计顶层原理图时，对从指令存储器中读出的 16 位指令，需要将它的不同数据位传递给不同的部件进行相应的操作，但由于之前从未涉及过这一操作，导致遇到了一些困难。

解决方法：

通过上网搜索相关的知识，从他人的博客中学到了在顶层原理图中对数据线进行分线的操作：先悬空画一条数据线，作为总线，将它命名为 Ins[15..0]，然后从这条线再引出一条分线，并将它命名为 Ins[15..12]，即可获得总线上数据的高 4 位。以此类推，即可获得总线上数据的不同位。

(2) 问题：

在处理 j 指令时，最初直接将 PC 的高 4 位和指令中的 12 位绝对地址进行拼接获得物理绝对地址，导致出错。

解决方法：

在计算目标地址时，由于指令长度为 16 位，而 j 指令中给出的是按逻辑地址表示的绝对地址，所以按物理地址表示的绝对地址应该是由逻辑绝对地址左移 1 位获得，所以物理绝对地址的构成应该为：PC 的高 3 位、j 指令中给出的 12 位地址以及最低位的“0”。

5.2 心得体会

通过这次单周期CPU的设计与实现，我收获颇丰。首先，我领会到了在整个实践过程中全局观念的重要性。单周期CPU设计这一实践是一项比较大的工程项目，因此，在开始实现之前，必须要有全局观，绝对不能一开始就编写各模块的代码。在编写代码之前，要确定好CPU的指令系统，包括在指令中用几位表示操作码，各个指令的格式等。另外还应该清楚将要设计的CPU的大致框架：熟悉CPU内部结构、熟悉单周期CPU的数据通路图、理解PC、指令存储器、寄存器堆、ALU、数据存储器的工作原理等。只有清楚了这些，在真正实现时，才不会手忙脚乱。其次，通过这次实践，我对Verilog语言的熟练度大大提高，对阻塞赋值与非阻塞赋值、过程赋值语句与持续赋值语句、wire类型与reg类型等概念有了更清晰的认识，也学会了在合适的地方使用合适的表达式和语句，这对我之后的硬件设计与实现无疑有很大的帮助。最后，通过这次对单周期CPU的完整实现，我对CPU，特别是单周期 CPU 有了更加深刻的理解，对 CPU 的认识不再局限于课本知识，而是更真切地感受到了各部件的协同配合以及 CPU 的工作模式。同时，我也体会到了设计以及实现CPU的艰辛，这锻炼了我的学习探索能力，激励着我更进一步探究CPU中的奥妙。