MIPS 单周期 CPU 设计实验报告

佴瑞乾 2018011716 自 84

目录

1 简单指令集 MIPS 单周期 CPU 数据通路

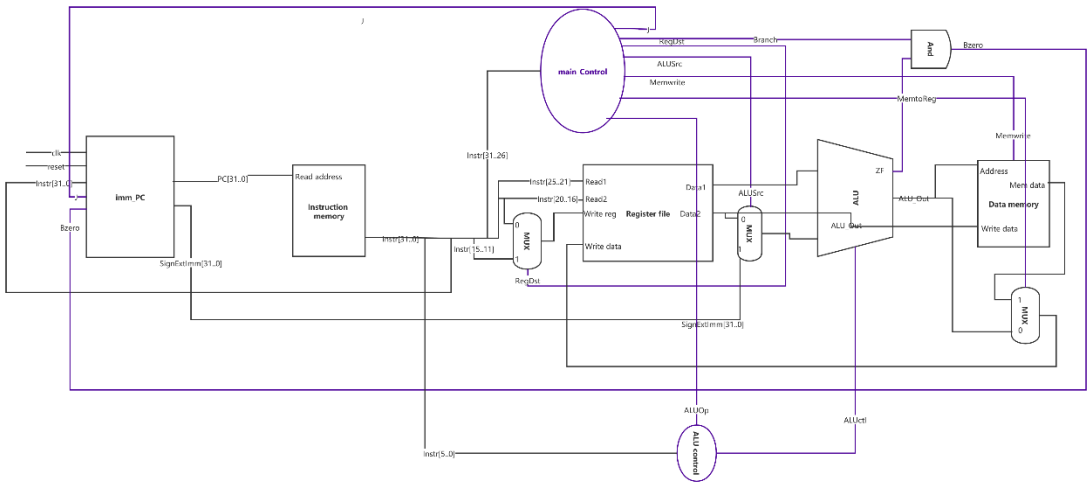


Figure 1 **简单指令集** MIPS **单周期** CPU **数据通路图**

MIPS 单周期数据通路如 Figure 1 所示。数据通路主要参考了教材图 4.17，其中，imm_PC 负责更新 PC 和对指令中的立即数进行符号扩展和无符号扩展。其他模块功能与教材图 4.17 相同。

2模块结构与作用

2.1 顶层模块结构

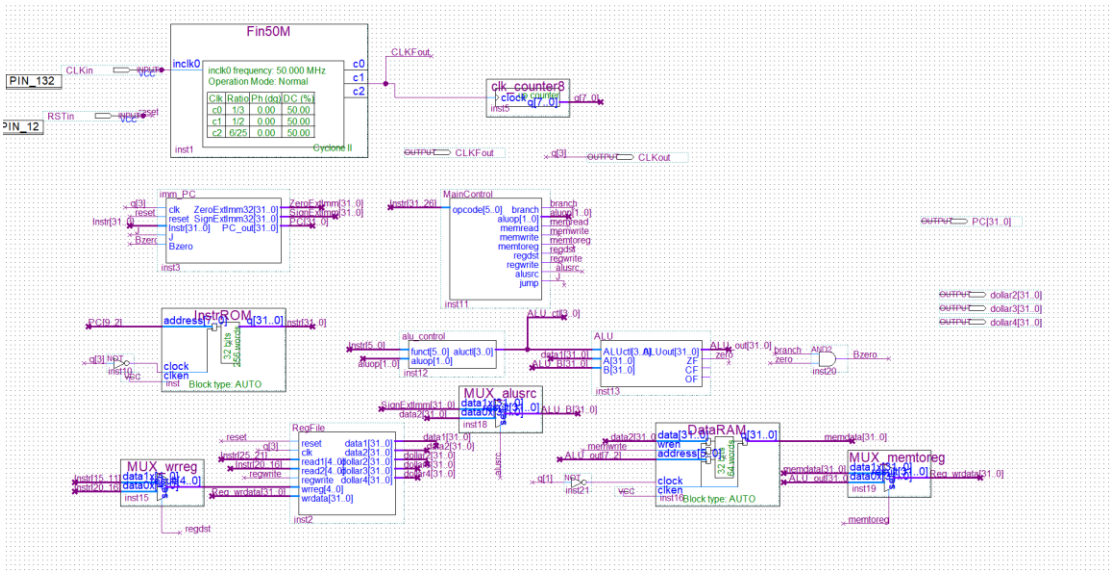


Figure 2 顶层模块结构

顶层模块结构如 Figure 2 所示。

顶层模块主要分为分频器(Fin50M,clk_counter8),imm_PC,指令存储器(InstrROM),主控制模块(MainControl),寄存器堆(RegFile), ALU 控制模块(alu_control),ALU,数据寄存器(DataRAM)和一些多选器，一个与门构成。

2.2 功能模块作用

2.2.1 分频模块

- **输入：**晶振 CLKin
- **输出：**计数器技术结果 q[7..0]
- **功能及实现思路：**分频模块由宏器件分频器和计数器构成。宏器件输出 25MHz 信号，再通过 8 位计数器实现分频并提供多种频率选择。实验采用计数器输出 q[3]作为主频，q[1]作为 DataRAM 的读写时钟。

2.2.2 imm_PC

- **输入：**时钟 clk，异步置零信号 reset，当前指令 Instr[31..0]，跳转控制信号 J，分支控制信号 Bzero.
- **输出：**程序计数器 PC_out，零扩展立即数 ZeroExtImm[31..0]，符号扩展立即数 SignExtImm[31..0].

- **功能及实现思路：**imm_PC 模块将输入指令中的立即数进行零扩展和符号扩展后分别输出。并通过控制信号 Bzero,J 来计算下一个 PC.若 Bzero 和 J 均为 0，则执行 PC+4；若 Bzero=1，则说明 beq 指令比较结果为相等，需要跳转，则 PC <= {{14{Instr[15]}},Instr[15:0],2'b0}；若 J=1，则说明需要跳转，则 PC <= {(PC+4)[31:28],Instr[25:0],2'b0}.这里的 reset 是异步置零端，reset 为高电平时，PC=0.

```
1.    module imm_PC(
2.           input wire        clk,
3.             input wire        reset,
4.             input wire [31:0] Instr,
5.             input wire        J,
6.             input wire        Bzero,
7.             output wire [31:0] ZeroExtImm32,
8.             output wire [31:0] SignExtImm32,
9.             output reg [31:0] PC_out,
10.            output wire [31:0] Next_PC);
11.
12.
13.            reg [31:0] TempPC;
14.
15.
16.
17.
18.            initial
19.            begin
20.               //TempPC <= 32'b0;
21.               PC_out <= 32'b0;
22.
23.            end
24.
25.            assign ZeroExtImm32[31:0] = {16'b0,Instr[15:0]};/*zero extend*
/
26.            assign SignExtImm32[31:0] = {{16{Instr[15]}},Instr[15:0]};/*si
gn extend*/
27.            assign Next_PC[31:0] = PC_out+32'b0100;
28.
29.
30.
31.            always @ (posedge clk or posedge reset)begin
32.               if (reset==1)
33.                   PC_out <= 32'b0;
34.                else if(J==0&&Bzero==0)
35.                  PC_out <= Next_PC;
36.                else if(J==1)
37.                   PC_out <= {Next_PC[31:28],Instr[25:0],2'b0};
```

```
38.                 else if(Bzero==1)
39.                     PC_out <= {{14{Instr[15]}},Instr[15:0],2'b0};
40.             end
41.
42.  endmodule
43.
44.
```

2.2.3 指令寄存器(InstrROM)

- **输入：**指令地址 address[7..0],时钟 clk,时钟使能信号 clken
- **输出：**ROM 存储内容 q[31..0]
- **功能及实现思路：**指令寄存器模块输入程序计数器 PC 输出当前需要执行的指令。指令寄存器使用 ROM 宏器件实现。考虑时钟上升沿到达到 PC 更新有一定的延迟，故将主频时钟 q[3]取反后作为指令寄存器的时钟。

2.2.4 主控制模块(MainControl)

- **输入：**指令的 opcode[5..0]
- **输出：**控制信号
- **功能及实现思路：**主控制器通过 opcode 为每个指令匹配控制信号从而设置正确的数据通路。Opcode 与控制信号的真值表如下。

表格 1 主控制模块真值表

	lw	sw	add	sub	and	or	slt	beq	j
aluop[1..0]	2'b10	2'b00	2'b10	2'b10	2'b10	2'b10	2'b10	2'b01	2'b10
alusrc	0	1	0	0	0	0	0	0	0
branch	0	0	0	0	0	0	0	1	0
memread	0	1	0	0	0	0	0	0	0
memtoreg	0	1	0	0	0	0	0	0	0
memwrite	0	0	0	0	0	0	0	0	0
regdst	1	0	1	1	1	1	1	1	1
regwrite	1	1	1	1	1	1	1	0	1
jump	0	0	0	0	0	0	0	0	1

通过真值表可以看出，对于 R-Format 指令，MainControl 输出的信号完全相同。

```
1.   module MainControl(
2.           input  wire [5:0]   opcode,
3.           output reg          branch,
4.           output reg [1:0]    aluop,
```

```
5.            output reg          memread, memwrite, memtoreg,
6.            output reg          regdst, regwrite, alusrc,
7.            output reg          jump);
8.
9.        always @(*) begin
10.
11.           aluop   <= 2'b10;
12.           alusrc      <= 1'b0;
13.           branch  <= 1'b0;
14.           memread     <= 1'b0;
15.           memtoreg    <= 1'b0;
16.           memwrite    <= 1'b0;
17.           regdst      <= 1'b1;
18.           regwrite    <= 1'b1;
19.           jump        <= 1'b0;
20.
21.           case (opcode)
22.               6'b100011: begin    /* lw */
23.                   memread  <= 1'b1;
24.                   regdst   <= 1'b0;
25.                   memtoreg <= 1'b1;
26.                   aluop <= 2'b00;
27.                   alusrc   <= 1'b1;
28.               end
29.               6'b000100: begin    /* beq */
30.                   aluop <= 2'b01;
31.                   branch <= 1'b1;
32.                   regwrite  <= 1'b0;
33.               end
34.               6'b101011: begin    /* sw */
35.                   memwrite <= 1'b1;
36.                   aluop <= 2'b00;
37.                   alusrc   <= 1'b1;
38.                   regwrite <= 1'b0;
39.               end
40.               6'b000000: begin    /* R-
Format Instructions:add,sub,and,or,slt */
41.               end
42.               6'b000010: begin    /* j */
43.                   jump <= 1'b1;
44.               end
45.           endcase
46.       end
47.   endmodule
```

2.2.5 寄存器堆(RegFile)

- **输入：**时钟 clk，异步初始信号 reset，读寄存器编号 read1[4..0],read2[4..0]，写寄存器编号 wrreg[4..0]，需要写入的数据 wrdata[31..0]，写使能信号 regwr.
- **输出：**read1 存储的数据 data1[31..0]，read2 存储的数据 data2[31..0].
- **功能及实现思路：**寄存器堆可以读取寄存器存储内容并在时钟上升沿来临时写入寄存器。异步初始化信号 reset 可以初始化寄存器内容便于调试。

```
1.   module RegFile(
2.          input wire        reset,
3.            input wire          clk,
4.            input wire  [4:0]   read1, read2,
5.            output wire [31:0]  data1, data2,
6.            output wire [31:0]   dollar2,dollar3,dollar4,
7.            input wire          regwrite,
8.            input wire  [4:0]   wrreg,
9.            input wire  [31:0]  wrdata);
10.
11.          reg [31:0] mem [0:31];// 32-bit memory with 32 $s
12.
13.       reg [31:0] _data1, _data2;
14.
15.       initial
16.       begin
17.
18.           begin
19.              mem[2][31:0] <= 32'b0;
20.               mem[3][31:0] <= 32'b1;
21.               mem[4][31:0] <= 32'b010;
22.           end
23.       end
24.
25.
26.
27.
28.       always @(*) begin
29.           if (read1 == 5'b0)/*$0=0*/
30.               _data1 = 32'b0;
31.           else
32.               _data1 = mem[read1][31:0];
33.       end
34.
35.       always @(*) begin
36.           if (read2 == 5'b0)
```

```
            _data2 = 32'b0;
        else
            _data2 = mem[read2][31:0];
    end

    assign data1 = _data1;
    assign data2 = _data2;
    assign dollar2 = mem[2];
    assign dollar3 = mem[3];
    assign dollar4 = mem[4];

    always @(posedge clk or posedge reset) begin

        if (reset==1)
        begin
           mem[2][31:0] <= 32'b0;
            mem[3][31:0] <= 32'b1;
            mem[4][31:0] <= 32'b010;
        end
        else if (regwrite && wrreg != 5'd0&&reset!=1) begin

            mem[wrreg] <= wrdata;
        end
    end
endmodule
```

2.2.6 ALU 控制模块(alu_control)

- **输入：**指令 funct[5..0],aluop[1..0]
- **输出：**ALU 控制信号 ALU_ctl[3..0]
- **功能及实现思路：**通过指令 funct 和 aluop 来匹配 ALU 的运算。当 aluop=0，这时的指令应是 I-Format 的 lw 或 sw，ALU 需要执行加运算；当 aluop=1，这时的指令应是 beq，ALU 需要执行减运算；当 aluop=2，这时执行的是 R-Format 的指令，需要通过 funct 来确定执行何种运算。实验时规定默认情况 ALU 执行加运算。ALU 控制模块的真值表如下。

表格 2ALU 控制模块真值表

	lw	sw	add	sub	and	or	slt	beq	j
aluop[1..0]	2'b10	2'b00	2'b10	2'b10	2'b10	2'b10	2'b10	2'b01	2'b10
funct[5..0]	2'h23	2'h2b	2'h20	2'h22	2'h24	2'h25	2'h2a	2'h04	2'h02
ALUctl[3..0]	4'b0010	4'b0010	4'b0010	4'b0110	4'b0000	4'b0001	4'b0111	4'b0110	x

```
module alu_control(
        input wire [5:0] funct,
      input wire [1:0] aluop,
```

```
4.          output reg [3:0] aluctl);
5.
6.      reg [5:0] _aluctl;
7.
8.      always @(*) begin
9.          case(funct)
10.             6'b100000:
11.             begin
12.             _aluctl = 4'b0010;   /* add */
13.             end
14.             6'b100010:
15.             begin
16.             _aluctl = 4'b0110;   /* sub */
17.             end
18.             6'b100100:
19.             begin
20.             _aluctl = 4'b0000;   /* and */
21.             end
22.             6'b100101:
23.             begin
24.             _aluctl = 4'b0001;   /* or */
25.             end
26.             6'b101010:
27.             begin
28.             _aluctl = 4'b0111;  /* slt */
29.             end
30.             default:
31.             begin
32.             _aluctl = 4'b0;
33.             end
34.         endcase
35.     end
36.
37.
38.     always @(*) begin
39.         case(aluop)
40.             2'b00: aluctl = 4'd2;   /* add */
41.             2'b01: aluctl = 4'd6;   /* sub */
42.             2'b10: aluctl = _aluctl;
43.             default: aluctl = 0;
44.         endcase
45.     end
46.
47.
```

```
48.
49.   endmodule
```

2.2.7 ALU

- **输入：**
  ALUctl：4 位控制信号，用于控制 ALU 执行加减与逻辑运算
  A,B：32 位有符号操作数
- **输出：**
  ALUout：32 位运算结果
  ZF：零标志
  OF：溢出标志
  CF：进位输出
- **功能及实现思路：**ALU 模块完全复用了实验二的设计。本实验设输入 A,B 均为有符号数。通过 ALUctl 来确定对操作数 A,B 进行何种操作，结果通过 ALUout 输出，当结果为零时零标志 ZF=1.

```
1.    module ALU (ALUctl,A,B,ALUout,ZF,CF,OF);
2.        input [3:0] ALUctl;
3.         input  signed [31:0] A,B;
4.         output reg signed[31:0] ALUout;
5.         output  ZF;
6.         output  CF;
7.         output  reg OF;
8.         reg extra;
9.         reg signed [31:0] C;
10.
11.
12.        assign ZF = (ALUout==0);
13.        assign CF = OF ^ (ALUctl==6);
14.        always @(ALUctl,A,B)
15.        begin
16.         case(ALUctl)
17.             0 :
18.               begin
19.                OF <= 0;
20.                ALUout <= A&B;
21.               end
22.             1 :
23.               begin
24.                 OF <= 0;
25.                 ALUout <= A|B;
26.               end
27.             2 :
```

```
28.               begin
29.                 {extra,ALUout} = {A[31],A}+{B[31],B};
30.                 OF = extra ^ ALUout[31];
31.               end
32.             6 :
33.               begin
34.                  C=-1*B;
35.                 {extra,ALUout} = {A[31],A}+{C[31],C};
36.                 OF = extra ^ ALUout[31];
37.               end
38.             7 :
39.               begin
40.                 OF <= 0;
41.                 ALUout <= A<B?1:0;
42.               end
43.             12 :
44.               begin
45.                 OF <= 0;
46.                 ALUout <= ~(A|B);
47.               end
48.              default:
49.               begin
50.                 OF <= 0;
51.               end
52.         endcase
53.       end
54.  endmodule
```

2.2.8 数据寄存器(DataRAM)

- **输入：**写入的数据 data[31..0]，写使能信号 wren，地址输入 address[5..0]，时钟 clk，时钟使能信号 clken.
- **输出：**q[31..0].
- **功能及实现思路：**数据寄存器可以实现读写数据的功能。采用 RAM 宏器件实现，其中，考虑到一个主频周期内读取数据寄存器的时间应当在读取指令寄存器以后且应在本周期以内，时钟 clk 输入为取反后 4 倍主频的时钟。

2.2.9 多选器

2.2.9.1 MUX_wwreg

- **data1:**Instr[15..11]

- **data0:**Instr[20..16]
- **控制信号：**regdst
- **功能：**选择写回寄存器堆的寄存器编号是$rt 还是$rd.

2.2.9.2 MUX_alusrc

- **data1:**SignExtImm[31..0]
- **data0:**data2[31..0]
- **控制信号：**alusrc
- **功能：**选择 ALU 的第二个操作数是$rt 还是符号扩展的立即数.

2.2.9.3 MUX_memtoreg

- **data1:**memdata[31..0]
- **data0:**ALUout[31..0]
- **控制信号：**memtoreg
- **功能：**选择写回寄存器的数据是 ALU 计算结果还是数据寄存器输出.

2.2.10 与门

- **输入：**控制分支信号 branch,ALU 输出的零标志 ZF.
- **输出：**控制是否跳转的信号 Bzero.
- **功能：**执行 beq 指令时，只有当比较结果相等，即 ALU 零标志为 1 时执行跳转。

3 分析仿真结果

3.1 仿真文件结构

为观察指令执行情况及 CPU 运行时的时序，仿真文件主要通过产生 50MHz 的时钟激励信号来仿真 CPU 运行。为与 signal tap 配合检验，仿真文件还在 CPU 运行时将 reset 置 1 后置 0.

```
1.  `timescale 10ns / 1ps
2.
3.  module SingleCyleCPU_tb;
4.     reg      RSTin;
5.      reg      CLKin;
6.      wire     CLKout;
7.      wire     CLKFout;
8.      wire       branch;
9.      wire [1:0] aluop;
```

```
10.       wire        memread;
11.       wire        memwrite;
12.       wire        memtoreg;
13.       wire        regdst;
14.       wire        regwrite;
15.       wire        alusrc;
16.       wire        J;
17.       wire [31:0]Instr;
18.       wire [31:0]PC;
19.       wire [31:0]SignExtImm;
20.       wire [31:0]data1;
21.       wire [31:0]data2;
22.       wire [3:0] ALU_ctl;
23.       wire [31:0]ALU_out;
24.       wire        zero;
25.       wire [31:0]memdata;
26.       wire [31:0]dollar2,dollar3,dollar4;
27.
28.       SingleCycleCPU CPU(.RSTin(RSTin),
29.                          .CLKin(CLKin),
30.                          .branch(branch),
31.                          .aluop(aluop),
32.                          .memread(memread),
33.                          .memwrite(memwrite),
34.                          .memtoreg(memtoreg),
35.                          .regdst(regdst),
36.                          .regwrite(regwrite),
37.                          .alusrc(alusrc),
38.                          .J(J),
39.                          .Instr(Instr),
40.                          .PC(PC),
41.                          .SignExtImm(SignExtImm),
42.                          .data1(data1),
43.                          .data2(data2),
44.                          .ALU_ctl(ALU_ctl),
45.                          .ALU_out(ALU_out),
46.                          .zero(zero),
47.                          .memdata(memdata),
48.                                .CLKout(CLKout),
49.                                .CLKFout(CLKFout),
50.                                .dollar2(dollar2),
51.                                .dollar3(dollar3),
52.                                .dollar4(dollar4));
53.
```

```
54.       initial
55.       begin
56.
57.         #0 RSTin = 0;
58.         #0 CLKin = 0;
59.
60.       end
61.
62.       always #1 CLKin = ~CLKin;
63.
64.       initial
65.       begin
66.
67.       #200 RSTin = 1;
68.       #4  RSTin =0 ;
69.
70.       end
71.
72.  endmodule
```

3.2 仿真结果分析

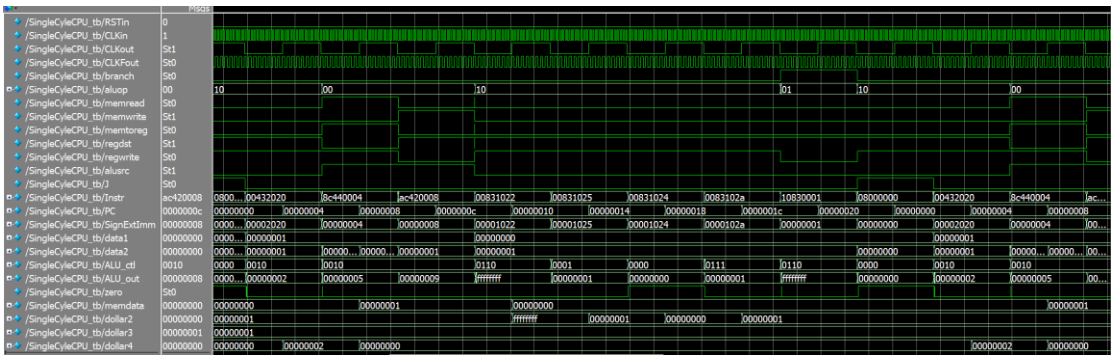


Figure 3 **仿真结果**

由 PC 和 Instr 仿真结果可知，程序执行语句顺序正确，且跳转、分支语句执行正确。

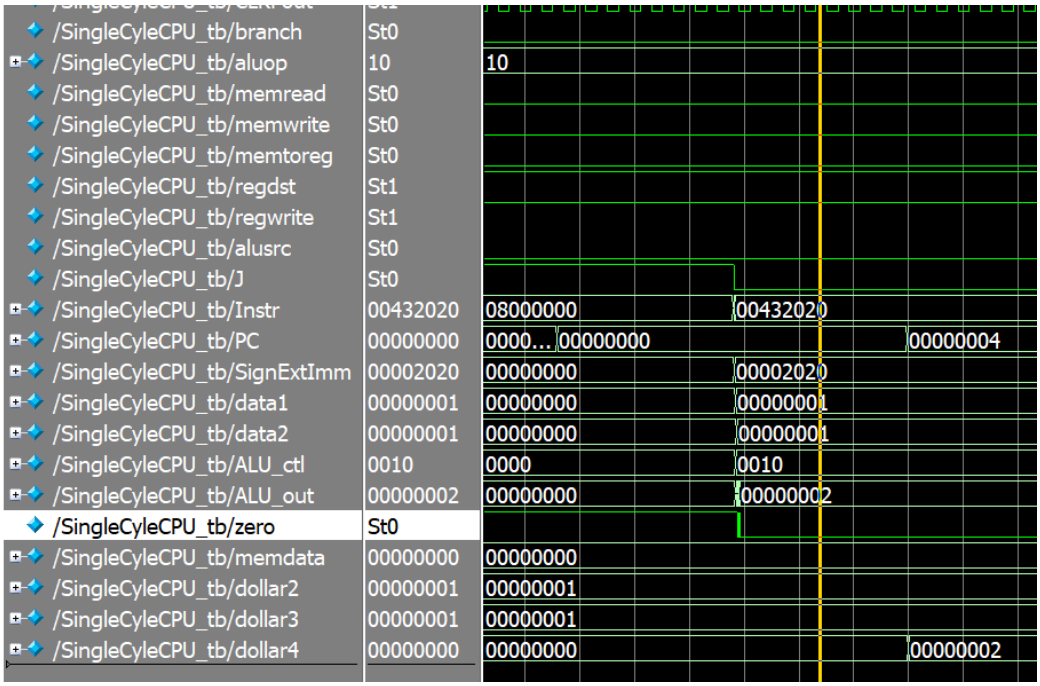


Figure 4 add $4,$2,$3 **语句执行情况**

观察 Figure 4 可知，执行 add 语句时，PC 更新后 Instr 更新，之后各个控制信号更新，然后寄存器堆输出更新，接着 ALU 输出结果更新。再下一个时钟上升沿来临时，运算结果被写回寄存器。

再看寄存器计算结果，$4=$2+$3=1+1=2，运算结果正确。

综上，add 语句执行正确。

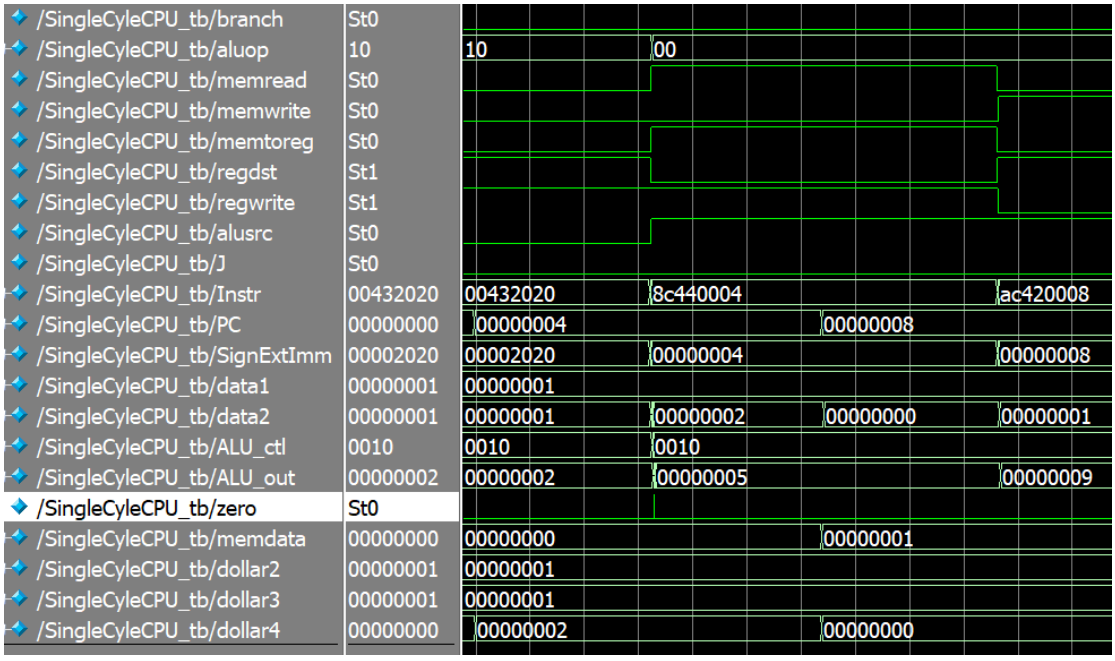


Figure 5 lw $4,4($2)**语句执行情况**

观察 Figure 5 可知，执行 lw 语句时，PC 更新后 Instr,SignExtImm 更新，之后各个控制信号更新，然后寄存器堆输出更新，接着 ALU 输出结果更新，之后数据寄存器输出更新。再下一个时钟上升沿来临时，运算结果被写回寄存器。

再看寄存器数据变化，$4 的值被更新为 0，故 lw 语句执行正确。

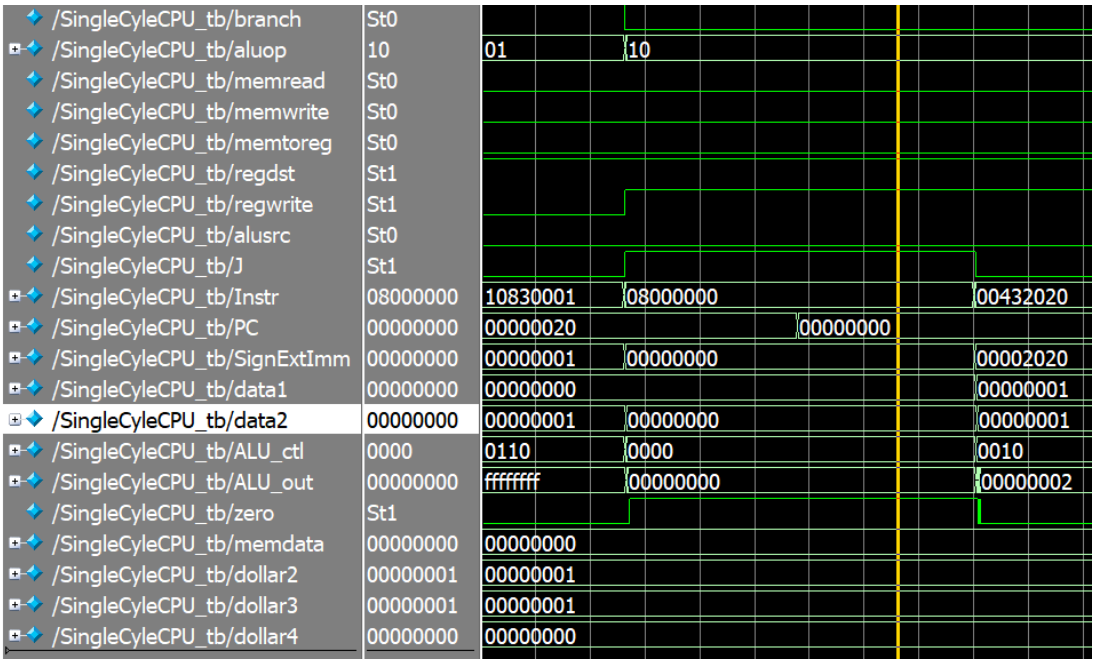


Figure 6 j main **语句执行情况**

观察 Figure 6 可知，PC 更新后 Instr 更新，之后各个控制信号更新。再下一个时钟上升沿来临时，PC 被更新为跳转地址。

再看 PC 数值变化，语句执行结束后 PC 被更新为 8'h00000000，为 main 对应地址，故语句执行正确。

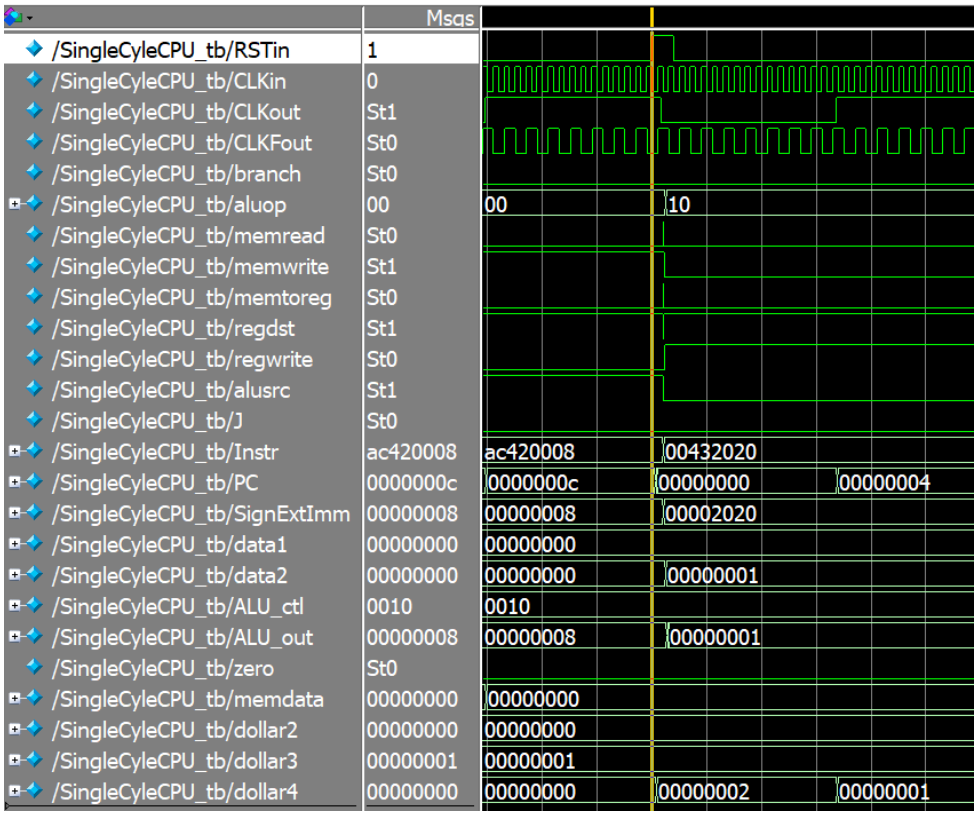


Figure 7reset **信号仿真**

观察 Figure 7 可知，reset 被置 1 后，PC 立即被初始化为零，$2,$3,$4 也被更新为初值。说明 reset 功能正常。

4分析 signalTap Ⅱ测量结果

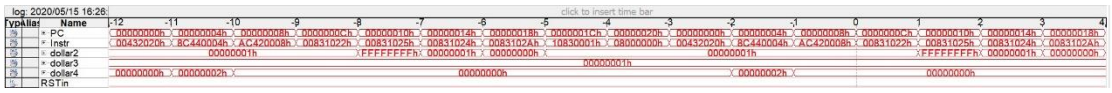


Figure 8 signalTap Ⅱ测量结果

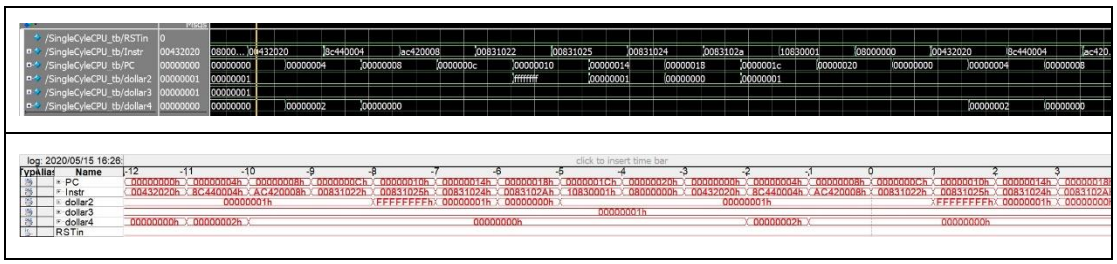
由 PC 和 Instr 运行结果可知，程序执行语句顺序正确，且跳转、分支语句执行正确。

PC=0 时，执行语句 add $4,$2,$3，观察执行结果，$4=$2+$3=1+1=2，运行正确。

PC=4 时，执行语句 lw $4,4($2)，观察执行结果，$4 存储数据被更新为零，运行正确。

PC=8'h20 时，执行语句 j main，观察执行结果，PC 值被更新为 0，运行正确。

表格 3 signalTap Ⅱ执行结果与 modelsim 仿真结果对比



对比 modelsim 仿真数据，可以看出仿真与实际运行结果相符合。故程序在 FPGA 板上运行正确。

5实验中出现的 bug 及解决办法

- **BUG:**进行 modelsim 仿真时，刚开始已知不能出现正确的波形，各个输出一直显示 0 或高阻态。modelsim script 报错显示分频器时钟输入错误。
- **解决办法：**检查仿真文件，发现时钟激励信号的频率不是 50MHz。由于使用的是宏器件，这要求分频器的时钟输入信号是 50MHz 才能正常输出。