实验报告

实验名称：多周期处理器实验

院系：计算机学院 19 级计算机科学与技术

班级：　计科 2 班

姓名：　施天予

指导老师：陈刚

一、 实验内容

4.1 实验要求

先阅读实验原理部分，实现下列模块，并按照下列要求完成实验：

- Datapath，基于单周期处理器 CPU 的数据通路代码进行改造，在各个需要插入寄存器组 IR, MDR, A,B, ALUOut（系统框图**红色部分**），注意其中 IR 和 PC 寄存器都需要进行使能控制。用于控制在当前指令没有结束的时候，禁止下一条指令导入。
- 在多周期处理器里面，由于指令存储器 inst_mem(Single Port Rom)和数据存储器 data_mem(Single Port Ram)是在不同周期里面使用的，因此可以分时使用将两个存储器合并；同实验 8 一样，使用 BlockMemory Generator IP 构造指令，注意考虑 PC 地址位数统一。 (参考实验 6)
- 参照实验原理，将上述模块依指令执行顺序连接。
- 实验给出仿真程序，最终以仿真输出结果判断是否成功实现要求指令。仿真具体参照，参照实验 1 基础操作，以及《Basys3 入门指导手册》。

4.2 实验步骤

1. 根据控制器在每个周期的输出控制信号，根据状态机设计控制器；
2. 从实验 8 中，导入各个模块，并按照如图 1 进行改写 datapath；
3. 根据实验 6 使用 Block Memory，生成统一的 memory；
4. 参考实验原理，连接各模块；
5. 导入顶层文件及仿真文件，运行仿真；

二、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。

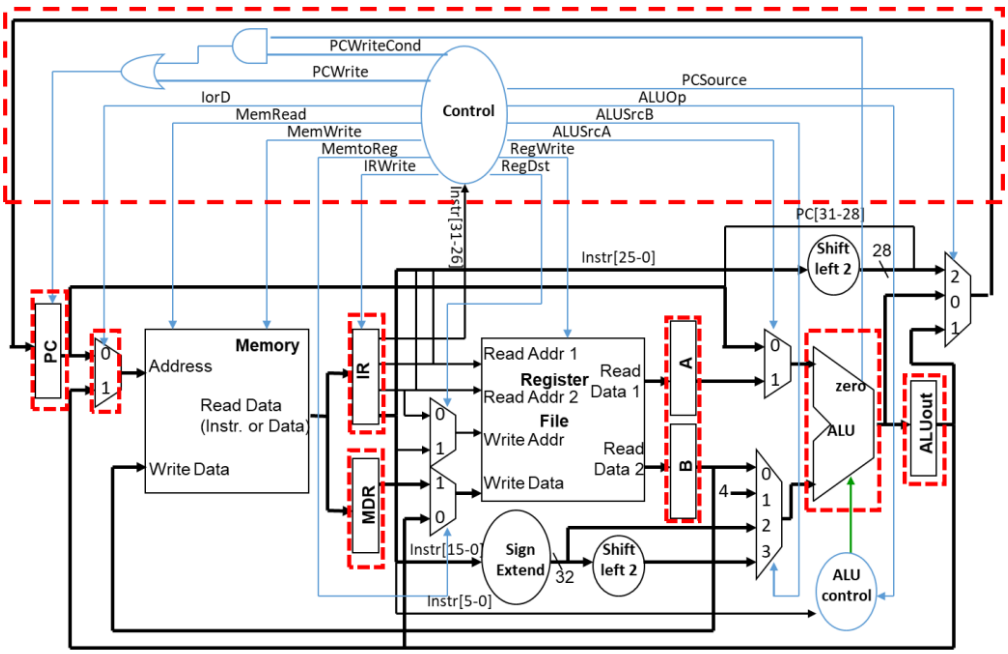
(3) 指令执行(EXE)：根据指令译码得到的操作控制信号，具体地执行指令动作，然后转移到结果写回状态。

(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。

5.1 实验总体框架及多周期 datapath 连线图

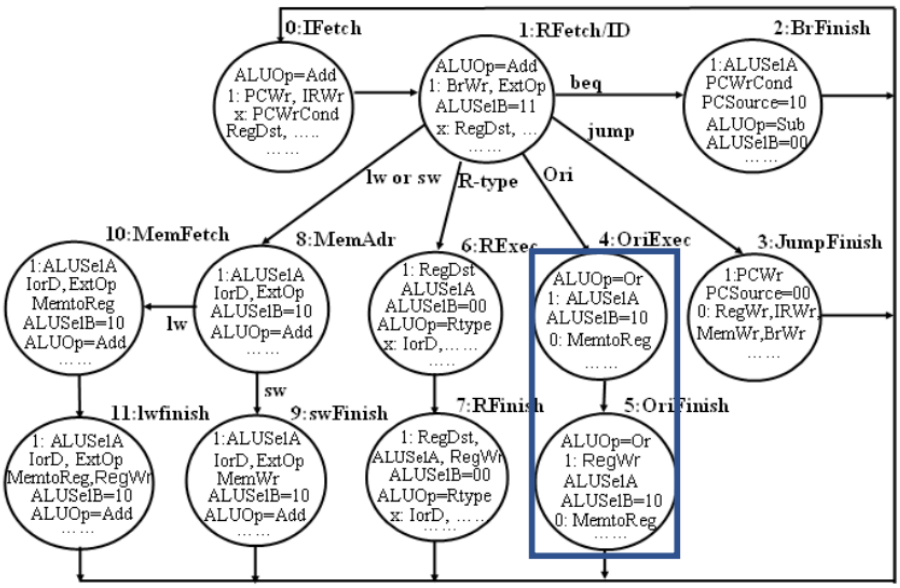


控制信号作用

控制信号名	描述
PCWrite	PCWrite =0，PC 不更改；PCWrite =1, PC 更改
ALUSrcA	ALUSrcA=0, ALU 的输入来自 PC; ALUSrcA=1, ALU 的输入来自寄存器堆;
ALUSrcB	ALUSrcB=00，ALU 的输入来自寄存器堆; ALUSrcB=01，ALU 的输入等于 4；用于做 PC+4; ALUSrcB=10，ALU 的输入来自与立即数; ALUSrcB=11，ALU 的输入来自与立即数迁移，主要用于 beq 指令;
MemtoReg	MemtoReg=0，来自 ALU 的输出写入 registerfile，例如 R-type 指令 MemtoReg=1，来自 Memory 出写入 registerfile，例如 lw 指令

RegWritem	Registerfile 写入控制信号，0 为读，1 为写入
RegDst	寄存器写入地址是 rd 还是 rt (和单周期类似) 注意：简单起见，我们未考虑 jal 指令
MemWrite	读写指令和存储存储器
IRWrite	IRWre =0；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。
ExtOp	(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< - pc+4+(sign-extend)immediate ×4，相关指令：beq(zero=1)、bne(zero=0)、bltz(sign=1); 10：pc< - {pc[31:28],addr[27:2],2'b00}，相关指令：j、jal;
ALUOp[2..0]	ALU 8 种运算功能选择(000-111)，见实验 8
PCWriteCond	Beq 信号的 PC 更新旁路信号
IorD	选择选指令存储器还是数据存储器

状态机



三、实验设计

根据控制信号作用表，可以设计出各状态的控制信号真值表。

ExtOp 在实验中并未用到，所以真值表中未出现。因为本次实验运行的程序中没有 ori 指令，有 addi 指令，所以我将 OriExec 和 OriFinish 改成 AddiExec 和 AddiFinish。它们都是立即数的计算，控制信号基本一致，唯一的不同是 ALUOp 从 OR 改成了 ADD。所以 ALUOp 共有 ADD，SUB，RTYPE 三种可能，只需 2 位即可，我们可分别设为 00，01，10。

这样，我们有 PCwrite，ALUSrcA，ALUSrcB，MemtoReg，RegWrite，RegDst，MemWrite，IRWrite，PCSrc，ALUOp，PCWriteCond，IorD 共 12 个控制信号，一共 15 位。

控制信号表

	I Fetch	R Fetch	Br Finish	Jump Finish	Addi Exec	Addi Finish
PCwrite	1	0	0	1	0	0
ALUSrcA	0	0	1	0	1	1
ALUSrcB	01	11	00	XX	10	10
MemtoReg	X	X	X	X	0	0
RegWrite	0	0	0	0	0	1
RegDst	X	X	X	X	0	0
MemWrite	0	0	0	0	0	0
IRWrite	1	0	0	0	0	0
PCSrc	00	XX	01	10	XX	XX
ALUOp	ADD	ADD	SUB	ADD	ADD	ADD
PCWriteCond	X	0	1	0	0	0
IorD	0	X	X	X	X	X

	R Exec	R Finish	Mem Adr	Sw Finish	Mem Fetch	Lw Finish
PCwrite	0	0	0	0	0	0
ALUSrcA	1	1	1	1	1	1
ALUSrcB	00	00	10	10	10	10
MemtoReg	X	0	X	X	1	1
RegWrite	0	1	0	0	0	1

RegDst	1	1	X	X	0	0
MemWrite	0	0	0	1	0	0
IRWrite	0	0	0	0	0	0
PCSrc	XX	XX	XX	XX	XX	XX
ALUOp	RTYPE	RTYPR	ADD	ADD	ADD	ADD
PCWriteCond	0	0	0	0	0	0
IorD	X	X	1	1	1	1

以下是具体代码模块

idmem 例化

```
ENTITY idmem IS
  PORT (
    clka : IN STD_LOGIC;
    wea : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
    addra : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
    dina : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
    douta : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
  );
END idmem;
```

top.v

```
`timescale 1ns / 1ps

module top(
    input wire clk, reset,
    output wire [31:0] writedata, adr,
    output wire memwrite,
    output wire [14:0]controls
    );
    wire [31:0] readdata;
    // instantiate processor and memory
    mips mips(clk, reset, adr, writedata, memwrite,readdata,controls);
    idmem idmem(clk, memwrite, adr[9:2], writedata,readdata);
endmodule
```

Mips.v

```
`timescale 1ns / 1ps

module mips(
    input wire clk, reset,
    output wire [31:0] adr, writedata,
    output wire memwrite,
    input wire [31:0] readdata,
    output wire [14:0] controls
    );
    wire zero, pcen, irwrite, regwrite,alusrca, iord, memtoreg, regdst;
    wire [1:0] alusrcb;
    wire [1:0] pcsrc;
    wire [2:0] alucontrol;
    wire [5:0] op,funct;
    controller c(clk,reset,op,funct,zero,pcen,memwrite,irwrite,regwrite,
        alusrca,iord,memtoreg,regdst,alusrcb,pcsrc,alucontrol,controls);
    datapath dp(clk,reset,pcen,irwrite,regwrite,alusrca,iord,memtoreg,regdst,
        alusrcb,pcsrc,alucontrol,zero,adr,writedata,readdata,op,funct);
endmodule
```

controller.v

```
`timescale 1ns / 1ps

module controller(
    input wire clk, reset,
    input wire [5:0] op, funct,
    input wire zero,
    output wire pcen, memwrite, irwrite, regwrite,
    output wire alusrca, iord, memtoreg, regdst,
    output wire [1:0] alusrcb,
    output wire [1:0] pcsrc,
    output wire [2:0] alucontrol,
    output wire [14:0] controls
    );
    wire [1:0] aluop;
    wire pcwritecond, pcwrite;
    maindec md(clk, reset, op, pcwrite, alusrca, alusrcb, memtoreg, regwrite,
        regdst, memwrite, irwrite, pcsrc, aluop, pcwritecond, iord,controls);
    aludec ad(funct, aluop, alucontrol);
    assign pcen = pcwrite | (pcwritecond & zero);
endmodule
```

maindec.v

由之前的状态机图，可设计出具体代码

```
`timescale 1ns / 1ps

module maindec(
    input wire clk, reset,
    input wire [5:0] op,
    output wire pcwrite, alusrca,
    output wire [1:0] alusrcb,
    output wire memtoreg, regwrite, regdst, memwrite, irwrite,
    output wire [1:0] pcsrc,
    output wire [1:0] aluop,
    output wire pcwritecond, iord,
    output reg [14:0] controls
    );
    localparam IFetch=0, RFetch=1, BrFinish=2, JumpFinish=3, AddiExec=4,
        AddiFinish=5, RExec=6, RFinish=7, MemAdr=8, swFinish=9, MemFetch=10, lwFinish=11;
    localparam RTYPE=6'b000000, LW=6'b100011, SW=6'b101011,
        BEQ=6'b000100, ADDI=6'b001000, J=6'b000010, ORI=6'b001101;
    reg [3:0] state, nextstate;
    always@(negedge clk or posedge reset)
        begin
            if (reset==1) state <= IFetch;
            else state <= nextstate;
        end

    // next state logic
    always @( * )
        case(state)
            IFetch: nextstate <= RFetch;
            RFetch: case(op)
                LW: nextstate <= MemAdr;
                SW: nextstate <= MemAdr;
                RTYPE: nextstate <= RExec;
                BEQ: nextstate <= BrFinish;
                ADDI: nextstate <= AddiExec;
                J: nextstate <= JumpFinish;
                default: nextstate <= IFetch;
            endcase
            BrFinish: nextstate <= IFetch;
            JumpFinish: nextstate <= IFetch;
            AddiExec: nextstate <= AddiFinish;
            AddiFinish: nextstate <= IFetch;
            RExec: nextstate <= RFinish;
            RFinish: nextstate <= IFetch;
            MemAdr: case(op)
                //RTYPE: nextstate <= swFinish;//
                LW: nextstate <= MemFetch;
                SW: nextstate <= swFinish;
                default: nextstate <= IFetch;
            endcase
```

```
            swFinish: nextstate <= IFetch;
            MemFetch: nextstate <= lwFinish;
            lwFinish: nextstate <= IFetch;
            default: nextstate <= IFetch;
        endcase
    // output logic
    assign {pcwrite, alusrca, alusrcb, memtoreg, regwrite, regdst,
        memwrite, irwrite,  pcsrc, aluop, pcwritecond, iord} = controls;
    always @( * )
        case(state)
            IFetch: controls <= 15'b100100001000000;
            RFetch: controls <= 15'b001100000000000;
            BrFinish: controls <= 15'b010000000010110;
            JumpFinish: controls <= 15'b100000000100000;
            AddiExec: controls <= 15'b011000000000000;
            AddiFinish: controls <= 15'b011001000000000;
            RExec: controls <= 15'b010000100001000;
            RFinish: controls <= 15'b010001100001000;
            MemAdr: controls <= 15'b011000000000001;
            swFinish: controls <= 15'b011000010000001;
            MemFetch: controls <= 15'b011010000000001;
            lwFinish: controls <= 15'b011011000000001;
            default: controls <= 15'b000000000000000;
        endcase
endmodule
```

aludec.v

```
`timescale 1ns / 1ps

module aludec(
    input wire[5:0] funct,
    input wire[1:0] aluop,
    output reg[2:0] alucontrol
    );
    always @(*) begin
       case(aluop)
           2'b00 : alucontrol <= 3'b010;//add (for lw/sw/addi/j)
           2'b01 : alucontrol <= 3'b110;//sub (for beq)
           default : case (funct)
               6'b100000:alucontrol <= 3'b010;//add
               6'b100010:alucontrol <= 3'b110;//sub
               6'b100100:alucontrol <= 3'b000;//and
               6'b100101:alucontrol <= 3'b001;//or
               6'b101010:alucontrol <= 3'b111;//slt
               default:  alucontrol <= 3'b000;
           endcase
       endcase

    end
endmodule
```

Datapath.v

```
`timescale 1ns / 1ps

module datapath(
    input wire clk, reset,
    input wire pcen, irwrite, regwrite,
    input wire alusrca, iord, memtoreg, regdst,
    input wire [1:0] alusrcb,
    input wire [1:0] pcsrc,
    input wire [2:0] alucontrol,
    output wire zero,
    output wire [31:0] adr, writedata,
    input wire [31:0] readdata,
    output wire [5:0] op,funct
    );
    wire [4:0] writereg;
    wire [31:0] pcnext, pc;
    wire [31:0] instr, data, srca, srcb;
    wire [31:0] a;
    wire [31:0] aluresult, aluout;
    wire [31:0] signimm; // the sign-extended imm
    wire [31:0] signimmsh; // the sign-extended imm << 2
    wire [31:0] wd3, rd1, rd2;
    assign op = instr[31:26];
    assign funct = instr[5:0];

    // datapath
    flopenr #(32) pcreg(clk, reset, pcen, pcnext, pc);
    mux2 #(32) adrmux(pc, aluout, iord, adr);// 由iord控制adr选择pc还是aluout
    flopenr #(32) instrreg(clk, reset, irwrite, readdata, instr);
    // 寄存器IR 在irwrite==1时触发instr = readdata
    flopr #(32) datareg(clk, reset, readdata, data);
    // 寄存器MDR 触发data = readdata
    mux2 #(5) regdstmux(instr[20:16],instr[15:11], regdst, writereg);
    mux2 #(32) wdmux(aluout, data, memtoreg, wd3);
    regfile rf(clk, regwrite, instr[25:21],instr[20:16],writereg, wd3, rd1, rd2);
    signext se(instr[15:0], signimm);
    sl2 immsh(signimm, signimmsh);
    flopr #(32) areg(clk, reset, rd1, a);// 寄存器A 触发a = rd1
    flopr #(32) breg(clk, reset, rd2, writedata);// 寄存器B 触发writedata = rd2
    mux2 #(32) srcamux(pc, a, alusrca, srca);// 由alusrca控制srca选择a还是pc
    mux4 #(32) srcbmux(writedata, 32'b100, signimm, signimmsh, alusrcb, srcb);
    // 由alusrcb控制srcb选择writedata，4，signimm还是signimmsh
    alu alu(srca, srcb, alucontrol, aluresult, zero);
    flopr #(32) alureg(clk, reset, aluresult, aluout);// 寄存器ALUout 触发aluout = aluresult
    mux3 #(32) pcmux(aluresult, aluout,{pc[31:28], instr[25:0], 2'b00},pcsrc, pcnext);
    // 由pcsrc控制pcnext选择aluresult，aluout还是j指令的跳转地址
endmodule
```

Regfile.v

```
`timescale 1ns / 1ps

module regfile(
    input wire clk,
    input wire we3,
    input wire[4:0] ra1,ra2,wa3,
    input wire[31:0] wd3,
    output wire[31:0] rd1,rd2
    );

    reg [31:0] rf[31:0];

    always @(posedge clk) begin
        if(we3) begin
            rf[wa3] <= wd3;
        end
    end

    assign rd1 = (ra1 != 0) ? rf[ra1] : 0;
    assign rd2 = (ra2 != 0) ? rf[ra2] : 0;
endmodule
```

S12.v

```
`timescale 1ns / 1ps

module s12(
    input wire[31:0] a,
    output wire[31:0] y
    );
    assign y = {a[29:0],2'b00};
endmodule
```

Signext.v

```
1  `timescale 1ns / 1ps
2
3  module signext(
4      input wire[15:0] a,
5      output wire[31:0] y
6      );
7
8      assign y = {{16{a[15]}},a};
9  endmodule
```

Alu.v

```
1      `timescale 1ns / 1ps
2
3      module alu(
4          input wire[31:0] a,b,
5          input wire[2:0] op,
6          output reg[31:0] y,
7          output wire zero
8          );
9
10         wire[31:0] s,bout;
11         assign bout = op[2] ? ~b : b;
12         assign s = a + bout + op[2];
13         always @(*) begin
14             case (op[1:0])
15                 2'b00: y <= a & bout;
16                 2'b01: y <= a | bout;
17                 2'b10: y <= s;
18                 2'b11: y <= s[31];
19                 default : y <= 32'b0;
20             endcase
21         end
22         assign zero = (y == 32'b0);
23     endmodule
```

Flopr.v

```
1   `timescale 1ns / 1ps
2
3   module flopr # (parameter WIDTH = 8)(
4       input wire clk,rst,
5       input wire[WIDTH-1:0] d,
6       output reg[WIDTH-1:0] q
7       );
8       always @(negedge clk,posedge rst) begin
9           if(rst) begin
10              q <= 0;
11          end else begin
12              q <= d;
13          end
14      end
15  endmodule
```

Flopenr.v

```
1   `timescale 1ns / 1ps
2
3   module flopenr # (parameter WIDTH = 8)(
4       input wire clk,rst,
5       input wire en,
6       input wire[WIDTH-1:0] d,
7       output reg[WIDTH-1:0] q
8       );
9       always @(negedge clk,posedge rst) begin
10          if(rst) begin
11              q <= 0;
12          end else begin
13              if (en)
14                  q <= d;
15              else
16                  q <= q;
17          end
18      end
19  endmodule
```

Mux2.v

```
1  `timescale 1ns / 1ps
2  module mux2 #(parameter WIDTH = 8)(
3      input wire[WIDTH-1:0] d0,d1,
4      input wire s,
5      output wire[WIDTH-1:0] y
6      );
7      assign y = s ? d1: d0;
8  endmodule
```

Mux3.v

```
1      `timescale 1ns / 1ps
2
3      module mux3 #(parameter WIDTH = 8)(
4          input wire [WIDTH-1:0] d0, d1, d2,
5          input wire [1:0] s,
6          output reg [WIDTH-1:0] y
7          );
8          always @( * )
9          case(s)
10             2'b00: y <= d0;
11             2'b01: y <= d1;
12             2'b10: y <= d2;
13         endcase
14     endmodule
```

Mux4.v

```
1      `timescale 1ns / 1ps
2
3      module mux4 #(parameter WIDTH = 8)(
4          input wire [WIDTH-1:0] d0, d1, d2, d3,
5          input wire [1:0] s,
6          output reg [WIDTH-1:0] y
7          );
8          always @( * )
9          case(s)
10             2'b00: y <= d0;
11             2'b01: y <= d1;
12             2'b10: y <= d2;
13             2'b11: y <= d3;
14         endcase
15     endmodule
```

testbench.v

```
`timescale 1ns / 1ps

module testbench();
    reg clk;
    reg rst;
    wire[31:0] writedata,adr;
    wire memwrite;
    wire [14:0]controls;
    top dut(clk,rst,writedata,adr,memwrite,controls);
    initial begin
        rst <= 1;
        #100;
        rst <= 0;
    end
    // 缩短时钟周期至1/5，以起到多周期加快运行速度的作用
    always begin
        clk <= 1;
        #2;
        clk <= 0;
        #2;
    end
     always @(negedge clk) begin
         if(memwrite) begin
             if(writedata === 7 & adr === 84) begin
                 $display("Simulation succeeded");
                 $stop;
             end
         end
     end
endmodule
```

四、实验结果与分析

为方便实验结果的观察，我把 Controls（12 个控制信号，15 位）也作为输出，这样可以看到程序正运行到哪个阶段。

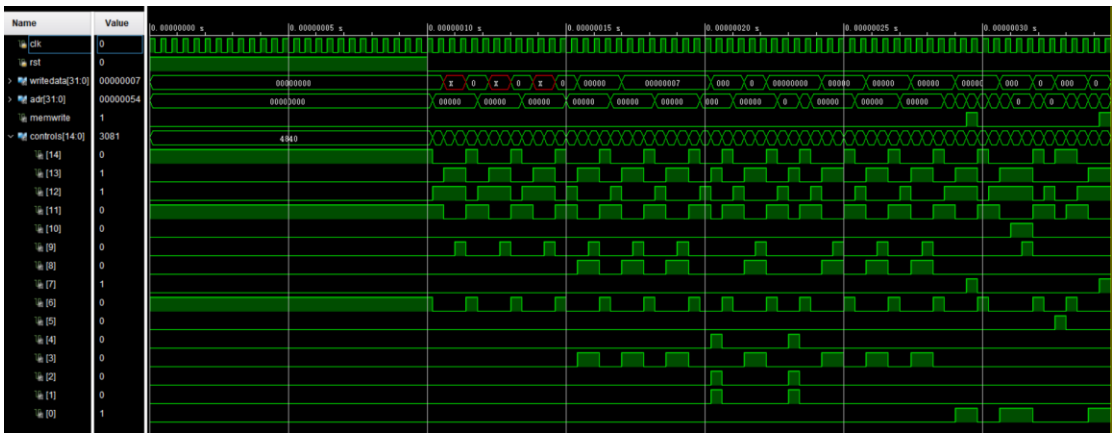
运行仿真程序

```
23          if(memwrite) begin
24              if(writedata === 7 & adr === 84) begin
25                  $display("Simulation succeeded");
26                  $stop;
27              end
```

```
Block Memory Generator module testbench.dut.idmem.inst.native_mem_module.blk_mem_ge
Simulation succeeded
INFO: [USF-XSim-96] XSim completed. Design snapshot 'testbench_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
```

说明仿真程序正确运行！

波形图概览



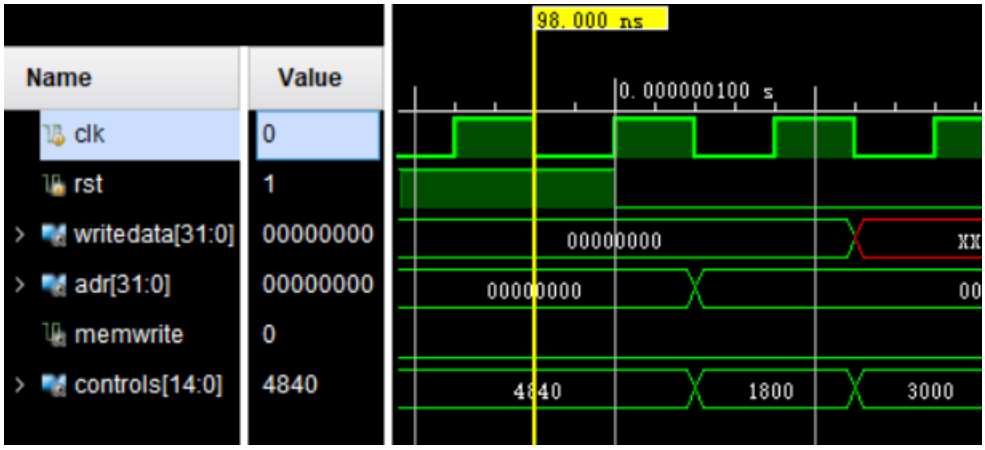
Writedata：第二个 ALU 操作数的值 / memory 中存储的数据

Dataadr：PC 下一条地址（未跳变） / memory 的地址

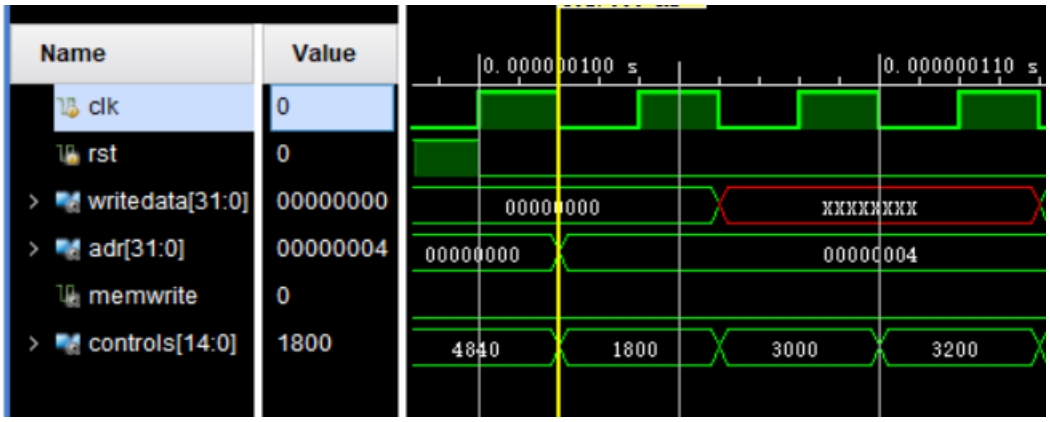
Memwrite：当执行 sw 指令时是 1

Controls：12 个控制信号，共 15 位。例如图中 4840（16 进制）代表正运行到 IFetch 阶段

下面将根据详细的波形分析程序是如何正确运行的



程序开始运行时 rst=1，直到 rst=0 时开始正常执行



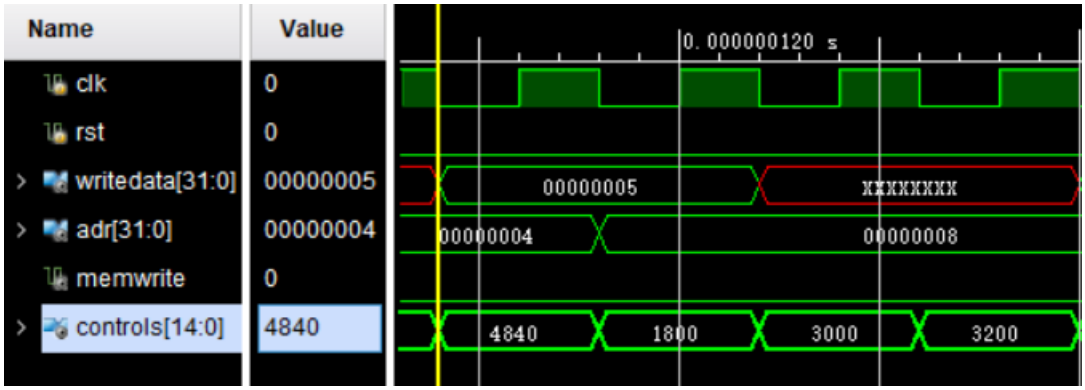
addi $2,$0,5　　20020005　　$2 = 5

IFetch（4840）：adr = 0（正运行指令的地址）

RFetch（1800）：adr = 4（下一条指令的地址）

AddiExec（3000）

AddiFinish（3200）



addi $3,$0,12　　200c000c　　$3 = 12

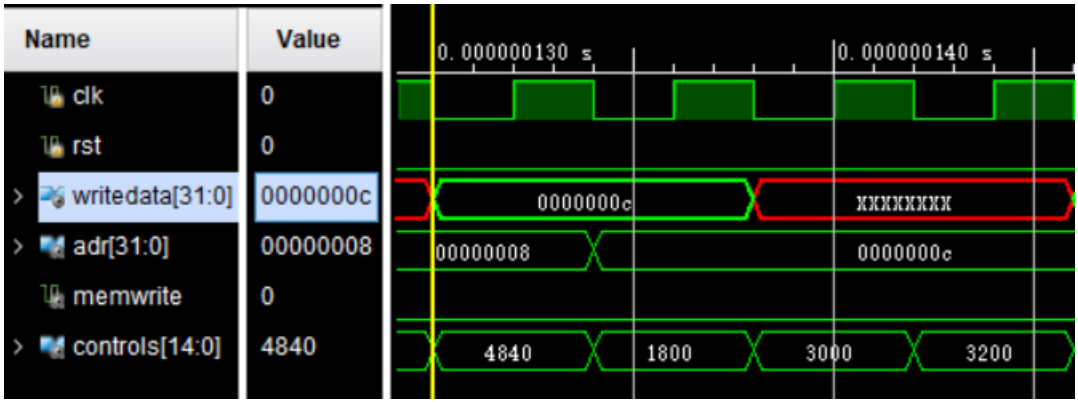
IFetch（4840）：adr = 4（正运行指令的地址）

Writedata = 5（上条指令计算$2 = 5）

RFetch（1800）：adr = 8（下一条指令的地址）

AddiExec（3000）

AddiFinish（3200）



addi $7,$3,-9　　2067fff7　　$7 = 3

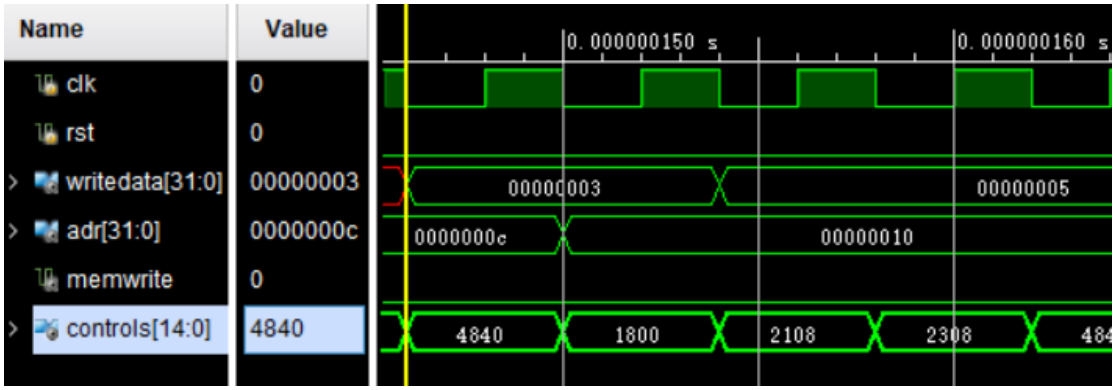
IFetch（4840）：adr = 8（正运行指令的地址）

Writedata = 12（上条指令计算$3 = 12）

RFetch（1800）：adr = c（下一条指令的地址）

AddiExec（3000）

AddiFinish（3200）



or $4,$7,$2　　00e22025　　$4 = 3 or 5 = 7

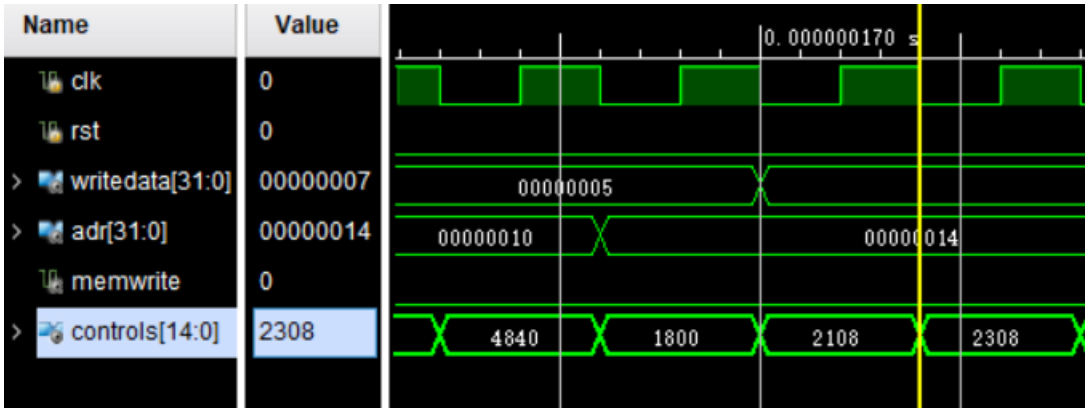
IFetch（4840）：adr = c（正运行指令的地址）

Writedata = 3（上条指令计算$7 = 3）

RFetch（1800）：adr = 10（下一条指令的地址）

RExec（2108）：writedata = 5（第二个 ALU 操作数为 5）

RFinish（2308）



and $5,$3,$4　　00642824　　$5 = 12 and 7 = 4

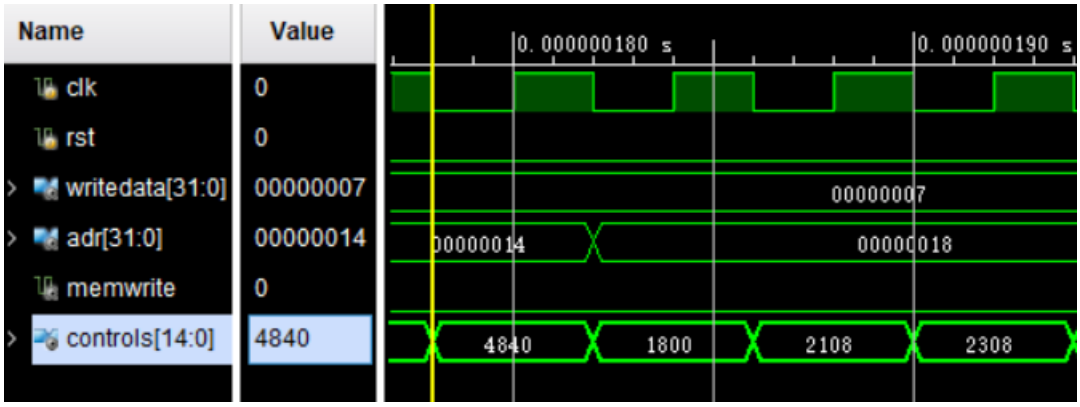
IFetch（4840）：adr = 10（正运行指令的地址）

Writedata = 5（上条指令为 5）

RFetch（1800）：adr = 14（下一条指令的地址）

RExec（2108）：writedata = 7（第二个 ALU 操作数为 5）

RFinish（2308）



add $5,$5,$4　　00a42820　　$5 = 4 + 7 = 11

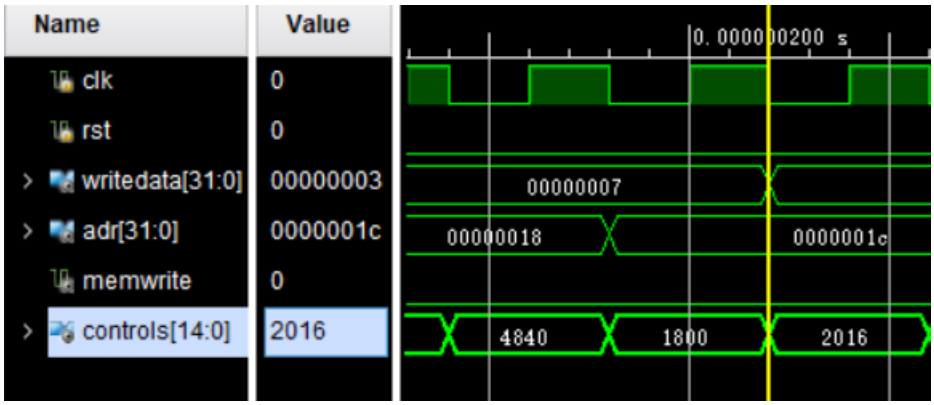
IFetch（4840）：adr = 14（正运行指令的地址）

Writedata = 7（上条指令为 7）

RFetch（1800）：adr = 18（下一条指令的地址）

RExec（2108）：writedata = 7（第二个 ALU 操作数为 7）

RFinish（2308）



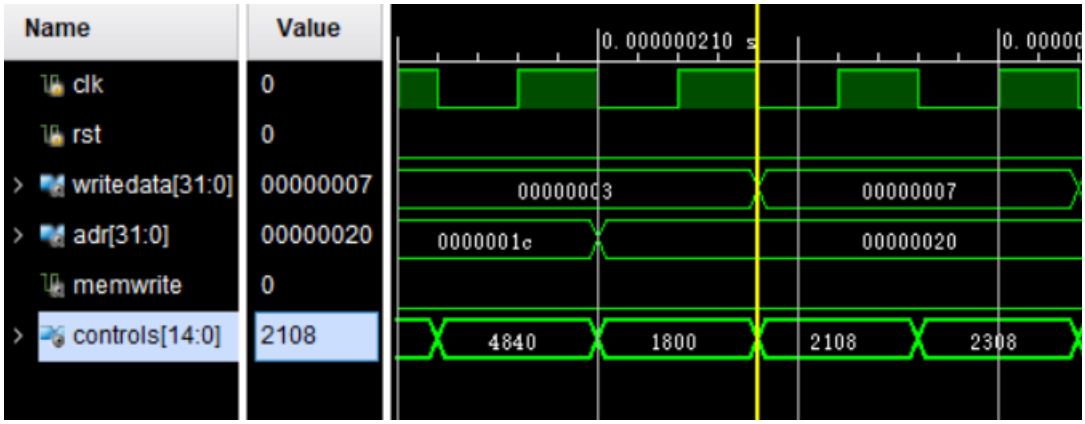
beq $5,$7,end　　10a70001　　11!=3 不发生 branch

IFetch（4840）：adr = 18（正运行指令的地址）

Writedata = 7（上条指令为 7）

RFetch（1800）：adr = 1c（下一条指令的地址）

BrFinish（2016）：writedata = 3（第二个 ALU 操作数为 3）



slt $4,$3,$4　　0064202a　　$4 = 12 < 7 = 0

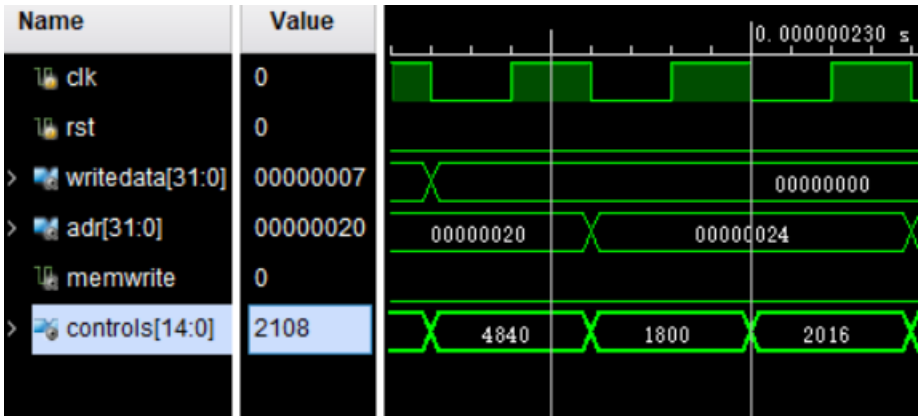
IFetch（4840）：adr = 1c（正运行指令的地址）

Writedata = 3（上条指令为 3）

RFetch（1800）：adr = 20（下一条指令的地址）

RExec（2108）：writedata = 7（第二个 ALU 操作数为 7）

RFinish（2308）

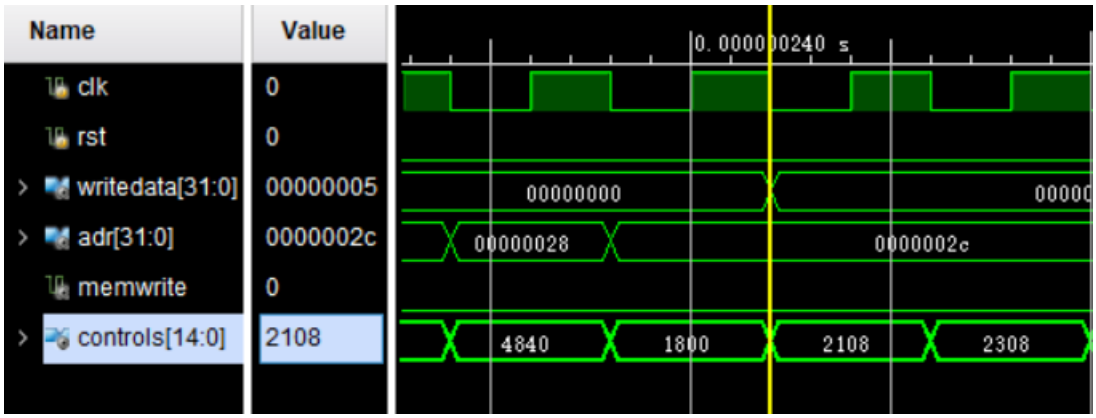


beq $4,$0,around 10800001 0==0 发生 branch 跳至 around

IFetch（4840）：adr = 20（正运行指令的地址）

RFetch（1800）：adr = 24（下条指令地址，其实跳变至 28）

BrFinish（2016）：writedata = 0（第二个 ALU 操作数为 3）



slt $4,$7,$2 00e2202a $4 = 3 < 5 = 1

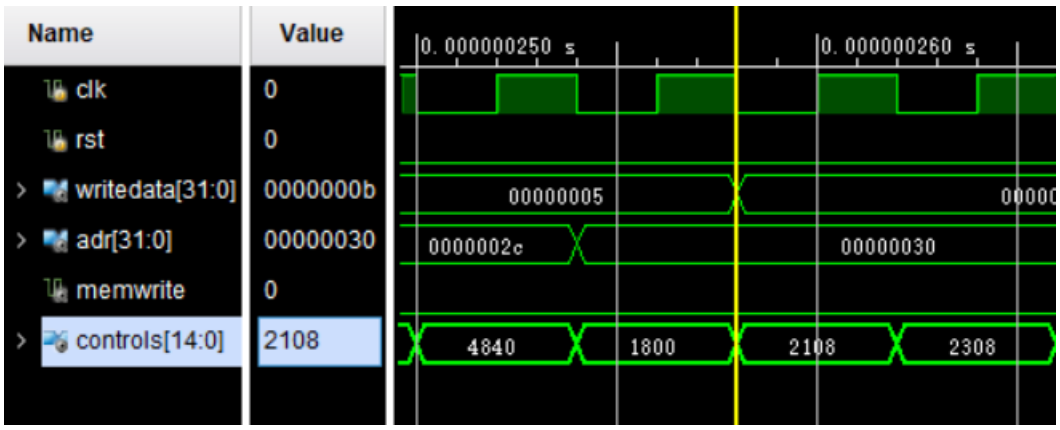
IFetch（4840）：adr = 28（正运行指令的地址）

Writedata = 0（上条指令为 0）

RFetch（1800）：adr = 2c（下一条指令的地址）

RExec（2108）：writedata = 5（第二个 ALU 操作数为 5）

RFinish（2308）



add $7,$4,$5　　00853820　　$7 = 1 + 11 = 12

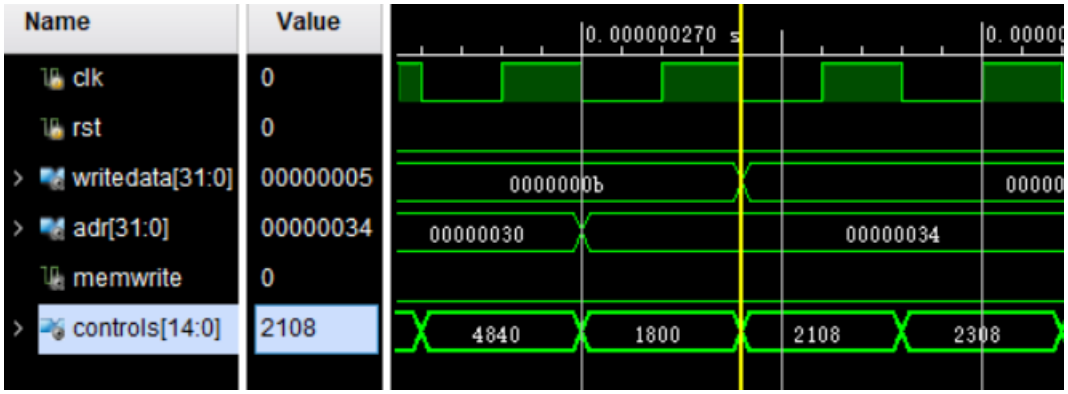
IFetch（4840）：adr = 2c（正运行指令的地址）

Writedata = 5（上条指令为 5）

RFetch（1800）：adr = 30（下一条指令的地址）

RExec（2108）：writedata = b（第二个 ALU 操作数为 11）

RFinish（2308）



sub $7,$7,$2　　00e23822　　$7 = 12 - 5 = 7

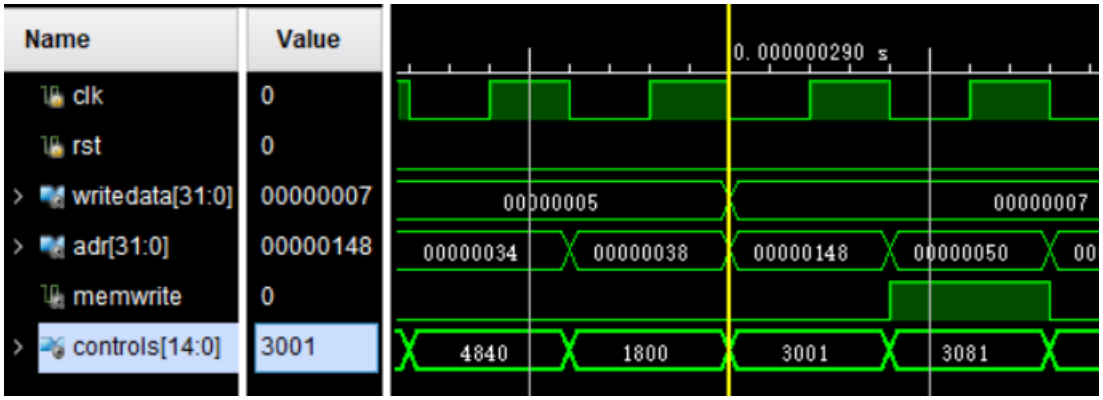
IFetch（4840）：adr = 30（正运行指令的地址）

Writedata = b（上条指令为 11）

RFetch（1800）：adr = 34（下一条指令的地址）

RExec（2108）：writedata = 5（第二个 ALU 操作数为 5）

RFinish（2308）



sw $7,68($3)　　ac670044　　[80] = 7

IFetch（4840）：adr = 34（正运行指令的地址）

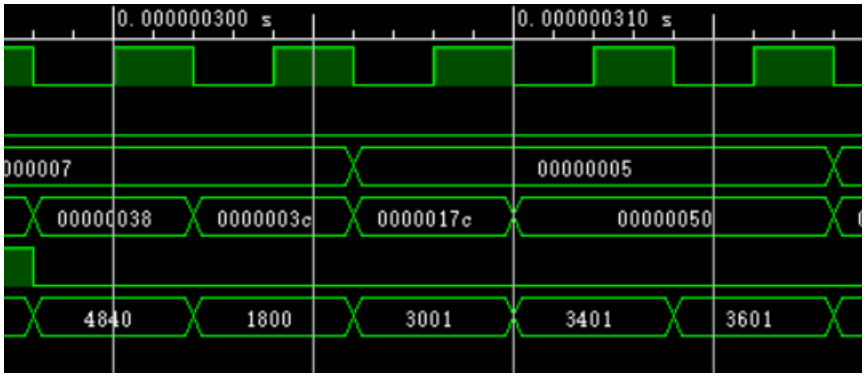
Writedata = 5（上条指令为 5）

RFetch（1800）：adr = 38（下一条指令的地址）

MemAdr（3001）：writedata = 7（$7 = 7）

swFinish（3081）：adr = 50（即十进制 68 + 4 * 3 = 80）

memwrite = 1



lw $2,80($0)　　8c020050　　$2 = [80] = 7

IFetch（4840）：adr = 38（正运行指令的地址）

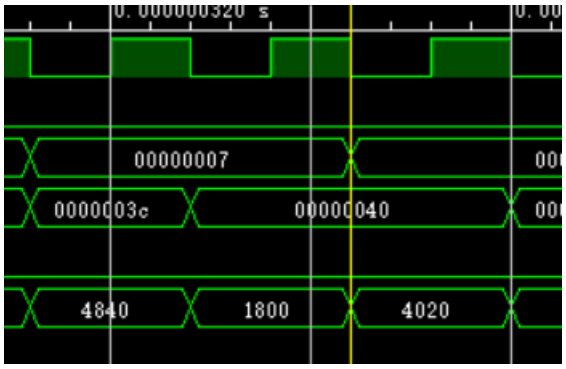
Writedata = 7（上条指令为 7）

RFetch（1800）：adr = 3c（下一条指令的地址）

MemAdr（3001）：writedata = 5（$2 原本为 5）

MemFetch（3401）：adr = 50（即十进制 80 + 4 * 0 = 80）

LwFinish（3601）：写回寄存器



j end　　08000011　　跳至 end

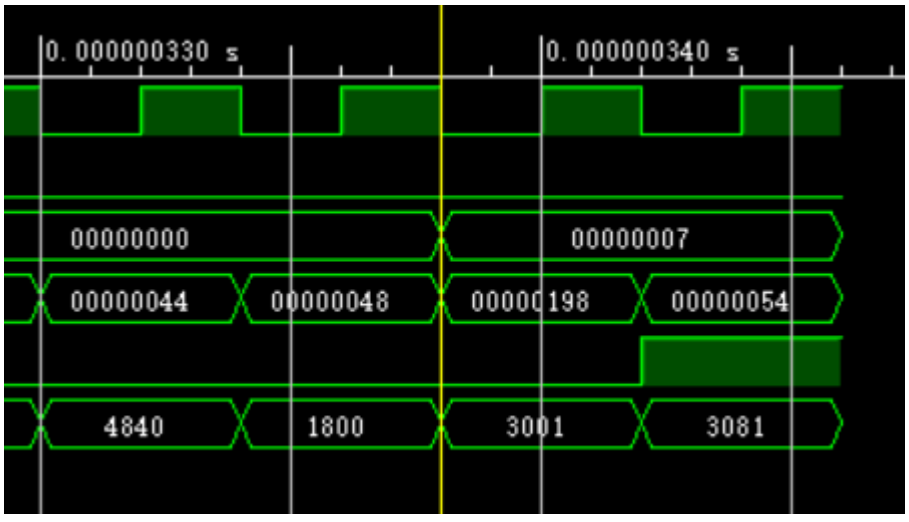
IFetch（4840）：adr = 3c（正运行指令的地址）

Writedata = 7（上条指令$2 = [80] = 7）

RFetch（1800）：adr = 40（下条指令地址，其实跳变至 44）

JumpFinish（4020）：此时计算出下条指令地址 44

（00010001<<2 = 01000100（44）刚好是 end 的地址）



sw $2,84($0)　　ac020054　　[84] = 7

IFetch（4840）：adr = 44（正运行指令的地址，j 指令跳转）

RFetch（1800）：adr = 48（下一条指令的地址，其实没了）

MemAdr（3001）：writedata = 7（$2 = 7）

swFinish（3081）：adr = 54（即十进制 84 + 4 * 0 = 84）

memwrite = 1

程序运行完毕!

五、实验总结

多周期 CPU 实验相比于单周期 CPU 大体的模块基本相同，但也有许多不同之处。首先就是控制信号变成了状态机，根据指令的不同类型跳变到不同的状态，不同状态的各个控制信号也各不相同。因为指令存储器和数据存储器在不同周期里使用，所以可将它们合并成 idmem。实验中我发现了一些细节的差错，比如老师的 PPT 上不小心把 lwFinish 状态的 RegWrite 错写成了 0，其实应该是 1，导致我一开始寄存器没办法写回。

实验中 deBug 的过程是痛苦的，我花了十个小时才发现我的一个错误。就是我在 mips 模块直接把 op 和 funct 用 readdata [31:26]和 readdata[5:0]代入，这样 pc 一跳变 op 和 funct 全乱了。其实应该在 datapath 模块把 readdata 通过 flopenr 存为 instr 后，再给 op 赋值 instr[31:26]，给 funct 赋值 instr[5:0]。

经过一遍遍的检查，我终于得到最终的正确程序。这次实验让我掌握了多周期 CPU 数据通路图的构成、原理及其设计方法，也掌握了多周期 CPU 的实现方法，代码实现方法，并且认识和掌握了指令与 CPU 的关系，让我受益匪浅。