实验报告

实验名称：单周期处理器实验

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

1. 掌握单周期 CPU 数据通路图的构成、原理及其设计方法;
2. 掌握单周期 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;
4. 掌握测试单周期 CPU 的方法。

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，其中主要包含 alu 和寄存器堆(实验 5 已完成)，存储器（实验 6 已经完成），PC 和 controller（实验 7 已经完成，其中 Controller 包含两部分，分别为 main_decoder，alu_decoder），adder、mux2、signext、sl2(其中 adder、mux2 数字逻辑课程已实现，相关的源代码模板在实验 5 已经提供，signext、sl2 参见实验原理)。
- 指令存储器 inst_mem(Single Port Rom)，数据存储器 data_mem(Single Port Ram)；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 top 文件，需兼容 top 文件端口设定。
- 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 实验模块结构

|–top.v	设计顶层文件，已提供。
|—-mips.v	MIPS 软核顶层文件，将 Controller 与 Datapath 连接，已提供
|———controller.v	控制器模块，参照（改写）实 7 代码。
|———maindec.v	Main decoder 模块，负责译码得到各个组件的控制信号。
|———aludec.v	ALU Decoder 模块，负责译码得到 ALU 控制信号。
|———datapath.v	数据通路模块，自行实现
|———pc.v	PC 模块，参照（改写）实验 7 代码
|———alu.v	ALU 模块，参照（改写）实验 5 代码
|———sl2.v	移位模块，参考《其他组件实现.pdf》
|———signext.v	有符号扩展模块，参考《其他组件实现.pdf》
|———mux2.v	二选一选择器，自行实现
|———regfile.v	寄存器堆，参照基础模块代码，实验 5 已经提供
|———adder.v	加法器，参照基础模块代码，实验 5 已经提供
|—-inst_ram.ip	RAM IP，通过 Block memory generator 进行实例化，见实验 6
|—-data_ram.ip	RAM IP，通过 Block memory generator 进行实例化，见实验 6

二、实验原理

5.1 实验总体框架及单周期处理器连线图

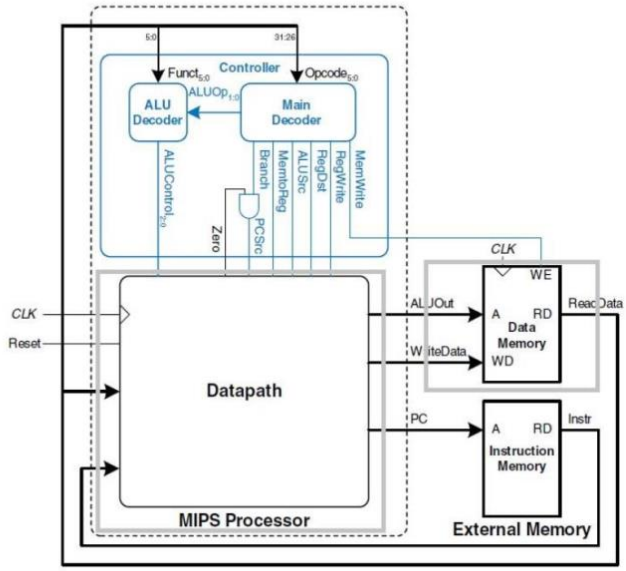


图 1: 单周期 CPU 框架图

如图 1，完整的单周期 CPU 实现框架图。图中有些部分我们之前已经在实验 5,6,7 中已经完成（参见 4.2 中的描述），继续完成 datapath 模块，即可将单周期 MIPS 软核完整实现。

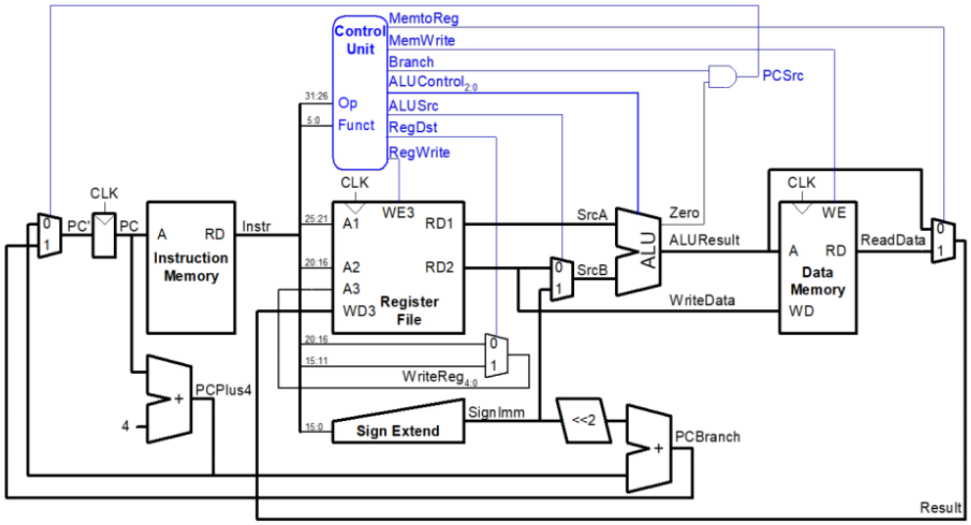


图 2: 单周期 CPU 系统连线图

图 2 为单周期的完整系统连线图，涵盖 *add*、*and*、*sub*、*or*、*slt*、*beq*、*j*、*lw*、*sw*、*addi* 等指令。本次实验仅实现上述几类指令，完整的 MIPS 指令集将与硬件综合设计中完善，此处以掌握数据通路分析为主要目的。

三、实验设计

Inst_mem 例化

```
59  ENTITY inst_mem IS
60    PORT (
61      clka : IN STD_LOGIC;
62      addra : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
63      douta : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
64    );
65  END inst_mem;
```

Data_mem 例化

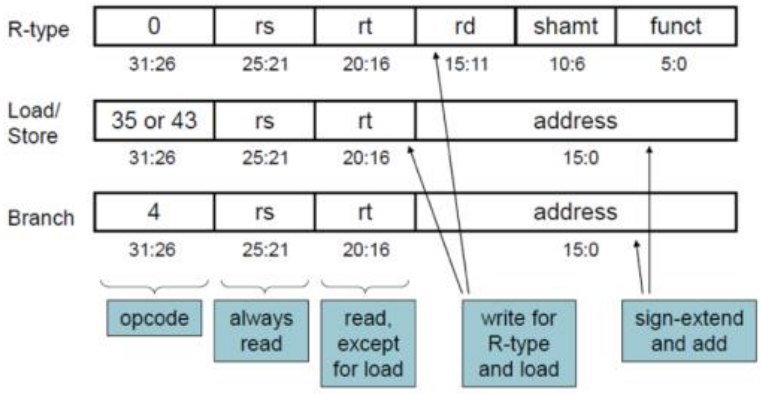
```
59  ENTITY data_mem IS
60    PORT (
61      clka : IN STD_LOGIC;
62      wea : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
63      addra : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
64      dina : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
65      douta : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
66    );
67  END data_mem;
```

Mips.v

```
1  `timescale 1ns / 1ps
2
3  module mips(
4      input wire clk,rst,
5      output wire[31:0] pc,
6      input wire[31:0] instr,
7      output wire memwrite,
8      output wire[31:0] aluout,writedata,
9      input wire[31:0] readdata
10     );
11
12     wire memtoreg,alusrc,regdst,regwrite,jump,pcsrc,zero,overflow;
13     wire[2:0] alucontrol;
14
15     controller c(instr[31:26],instr[5:0],zero,memtoreg,
16         memwrite,pcsrc,alusrc,regdst,regwrite,jump,alucontrol);
17     datapath dp(clk,rst,memtoreg,pcsrc,alusrc,
18         regdst,regwrite,jump,alucontrol,overflow,zero,pc,instr,aluout,writedata,readdata);
19
20 endmodule
```

controller.v

指令译码原理



根据指令的高 6 位得到 op，低 6 位得到 funct，还有一个 ALU 的零输出（这里默认为 1）作为控制器的输入，得到一系列控制信号

```
`timescale 1ns / 1ps

module controller(
    input wire[5:0] op,
    input wire[5:0] funct,
    input wire zero,
    output wire memtoreg,memwrite,
    output wire pcsrc,alusrc,
    output wire regdst,regwrite,
    output wire jump,branch,
    output wire[2:0] alucontrol
    );
    wire [1:0] aluop;

    maindec md(op,memtoreg,memwrite,branch,alusrc,regdst,regwrite,jump,aluop);
    aludec ad(funct,aluop,alucontrol);

    assign pcsrc = branch & zero;
endmodule

```

maindec.v

instruction	op5:0	regwrite	regdst	alusrc	branch	memWrite	memtoReg	aluop1:0
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01
addi	001000	1	0	1	0	0	0	00
j	000010	0	X	X	X	0	X	XX

根据参考信号表，可以设计出该模块代码

```
`timescale 1ns / 1ps

module maindec(
    input wire[5:0] op,
    output wire memtoreg,memwrite,
    output wire branch,alusrc,
    output wire regdst,regwrite,
    output wire jump,
    output wire[1:0] aluop
    );
    reg[8:0] controls;
    assign {regwrite,regdst,alusrc,branch,memwrite,memtoreg,jump,aluop} = controls;
    always @(*) begin
       case(op)
           6'b000000:controls <= 9'b110000010;//R-TYPE
           6'b100011:controls <= 9'b101001000;//LW
           6'b101011:controls <= 9'b001010000;//SW
           6'b000100:controls <= 9'b000100001;//BEQ
           6'b001000:controls <= 9'b101000000;//ADDI
           6'b000010:controls <= 9'b000000100;//J
           default  :controls <= 9'b000000000;//illegal op
       endcase
    end
endmodule
```

aludec.v

opcode	aluop	operation	funct	alu function	alu control
lw	00	Load word	XXXXXX	Add	010
sw	00	Store word	XXXXXX	Add	010
beq	01	Branch equal	XXXXXX	Subtact	110
R-type	10	Add	100000	qdd	010
		subtract	100010	Subtract	110
		and	100100	And	000
		or	100101	Or	001
		set-on-less-than	101010	SLT	111

根据上表，用 funct 和 aluop 可得出 alucontrol

```
`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

Pcnext 可能是加 4，可能是 branch 指令后的地址，也可能是 j 指令后的地址。通过两个二选一选择器可以得到正确的 pcnext

```
`timescale 1ns / 1ps

module datapath(
    input wire clk,rst,
    input wire memtoreg,pcsrc,
    input wire alusrc,regdst,
    input wire regwrite,jump,
    input wire[2:0] alucontrol,
    output wire overflow,zero,
    output wire[31:0] pc,
    input wire[31:0] instr,
    output wire[31:0] aluout,writedata,
    input wire[31:0] readdata
    );
    wire [4:0] writereg;
    wire [31:0] pcnext,pcnextbr,pcplus4,pcbranch;
    wire [31:0] signimm,signimmsh,srca,srcb,result;
    // next PC
    flopr#(32) pcreg(clk,rst,pcnext,pc);// 除非rst，否则在时钟跳变时pc = pcnext
    adder pcadd1(pc,32'b100,pcplus4);// pcplus4 = pc + 4
    sl2 immsh(signimm,signimmsh);// signimmsh = signimm << 2
    adder pcadd2(pcplus4,signimmsh,pcbranch);// pcbranch = pcplus4 + signimmsh
    mux2 #(32) pcbrmux(pcplus4,pcbranch,pcsrc,pcnextbr);// 如果pcsrc = 1，那么pcnextbr = pcbranch（跳转发生），否则pcnextbr = pcplus4(正常运行)
    mux2 #(32) pcmux(pcnextbr,{pcplus4[31:28],instr[25:0],2'b00},jump,pcnext);//如果jump = 1，pcnext由立即数*4和当前地址高4位组成，否则pcnext = pcnextbr
```

寄存器的两个输出一个肯定是 ALU 的一个操作数，通过判断指令类型选择 ALU 的第二个输入是寄存器的另一个输出还是立即数。根据 regdst 得到正确的写地址，根据 memtoreg 选择寄存器写入的数据是 ALU 的输出还是 datamemory 的 readdata。

```
    // register
    regfile rf(clk,regwrite,instr[25:21],instr[20:16],writereg,result,srca,writedata);// 由寄存器获得srca（ALU第一个输入）和writedata
    mux2 #(5) wrmux(instr[20:16],instr[15:11],regdst,writereg);// 根据regdst得到正确的写地址
    mux2 #(32) resmux(aluout,readdata,memtoreg,result);// 根据memtoreg选择写入的数据result是alu的输出还是datamemory的readdata
    signext se(instr[15:0],signimm);// signimm是instr低16位的32位扩展
    // ALU
    mux2 #(32) srcbmux(writedata,signimm,alusrc,srcb);// 根据alusrc选择srcb（ALU第二个输入）是来自寄存器的writedata还是signimm（立即数扩展）
    alu alu(srca,srcb,alucontrol,aluout,overflow,zero);// alu计算，zero用于控制器判断是否发生跳转
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
```

Adder.v

```
`timescale 1ns / 1ps

module adder(
    input wire[31:0] a,b,
    output wire[31:0] y
    );

    assign y = a + b;
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

```
`timescale 1ns / 1ps

module signext(
    input wire[15:0] a,
    output wire[31:0] y
    );

    assign y = {{16{a[15]}},a};
endmodule
```

Flopr.v

```
`timescale 1ns / 1ps

module flopr #(parameter WIDTH = 8)(
    input wire clk,rst,
    input wire[WIDTH-1:0] d,
    output reg[WIDTH-1:0] q
    );
    always @(negedge clk,posedge rst) begin
        if(rst) begin
            q <= 0;
        end else begin
            q <= d;
        end
    end
endmodule

```

Mux2.v

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

Alu.v

```
`timescale 1ns / 1ps

module alu(
    input wire[31:0] a,b,
    input wire[2:0] op,
    output reg[31:0] y,
    output reg overflow,
    output wire zero
    );

    wire[31:0] s,bout;
    assign bout = op[2] ? ~b : b;
    assign s = a + bout + op[2];
    always @(*) begin
        case (op[1:0])
            2'b00: y <= a & bout;
            2'b01: y <= a | bout;
            2'b10: y <= s;
            2'b11: y <= s[31];
            default : y <= 32'b0;
        endcase
    end
    assign zero = (y == 32'b0);

    always @(*) begin
        case (op[2:1])
            2'b01:overflow <= a[31] & b[31] & ~s[31] |
                              ~a[31] & ~b[31] & s[31];
            2'b11:overflow <= ~a[31] & b[31] & s[31] |
                              a[31] & ~b[31] & ~s[31];
            default : overflow <= 1'b0;
        endcase
    end
endmodule
```

top. v

```
`timescale 1ns / 1ps

module top(
    input wire clk,rst,
    output wire[31:0] writedata,dataadr,
    output wire memwrite
    );
    wire[31:0] pc,instr,readdata;
    mips mips(clk,rst,pc,instr,memwrite,dataadr,writedata,readdata);
    //create imem and dmem by yourself
    inst_mem imem(clk,pc[9:2],instr);
    data_mem dmem(~clk,memwrite,dataadr[9:2],writedata,readdata);
endmodule
```

testbench. v

```
`timescale 1ns / 1ps

module testbench();
    reg clk;
    reg rst;

    wire[31:0] writedata,dataadr;
    wire memwrite;

    top dut(clk,rst,writedata,dataadr,memwrite);

    initial begin
        rst <= 1;
        #200;
        rst <= 0;
    end

    always begin
        clk <= 1;
        #10;
        clk <= 0;
        #10;

    end
```

```
25
26      always @(negedge clk) begin
27          if(memwrite) begin
28              if(dataadr === 84 & writedata === 7) begin
29                  /* code */
30                  $display("Simulation succeeded");
31                  $stop;
32              end
33          end
34      end
35  endmodule
36
```

四、实验结果与分析

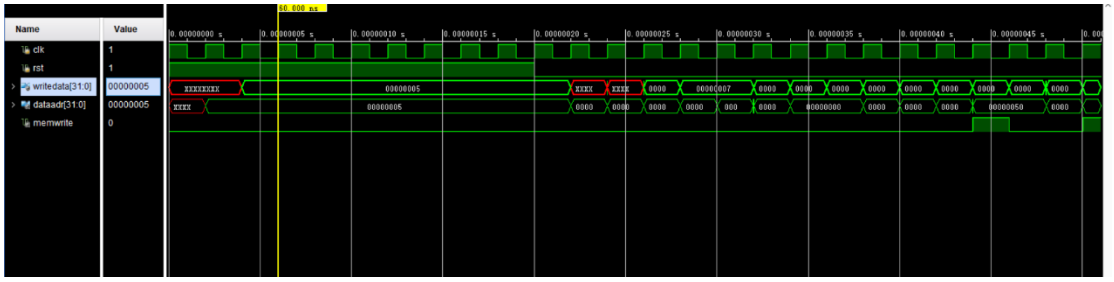
运行仿真程序

```
26      always @(negedge clk) begin
27          if(memwrite) begin
28              if(dataadr === 84 & writedata === 7) begin
29                  /* code */
30                  $display("Simulation succeeded");
31                  $stop;
32              end
33          end
34      end
35  endmodule
36
```

```
Block Memory Generator module loading initial data...
Block Memory Generator data initialization complete.
Block Memory Generator module testbench.dut.imem.inst.native_mem_module.blk_mem_gen_v8_4_1_inst is using a behavioral m
Block Memory Generator module loading initial data...
Block Memory Generator data initialization complete.
Block Memory Generator module testbench.dut.dmem.inst.native_mem_module.blk_mem_gen_v8_4_1_inst is using a behavioral m
Simulation succeeded
INFO: [USF-XSim-96] XSim completed. Design snapshot 'testbench_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:03 ; elapsed = 00:00:09 . Memory (MB): peak = 1125.215 ; gain = 0.000
```

说明仿真程序正确运行！

波形图概览

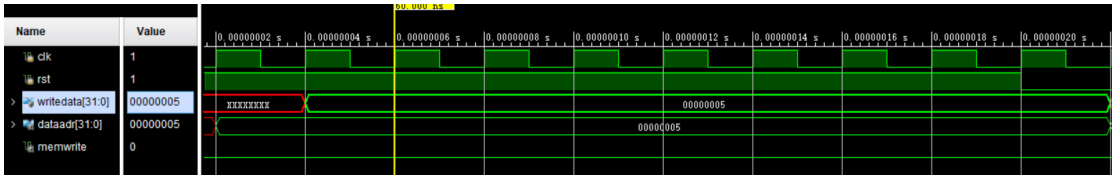


Writedata：第二个操作数的值/data memory 中存储的数据

Dataadr：ALU 的计算结果

Memwrite：当执行 sw 指令时是 1

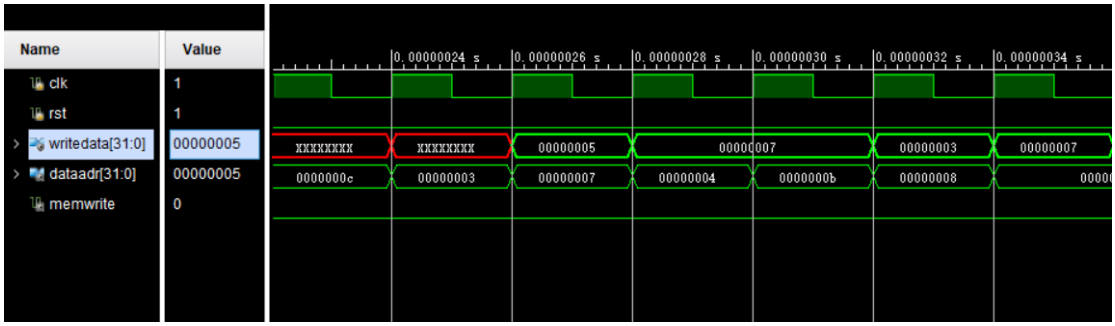
下面将根据详细的波形分析程序是如何正确运行的



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

dataadr = 5（ALU 算术结果）



addi $3,$0,12　　200c000c　　$3 = 12

dataadr = 12（ALU 算术结果）

addi $7,$3,-9　　2067fff7　　$7 = 3

dataadr = 3（ALU 算术结果）

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

writedata = 5(第二个操作数$2 = 5)

dataadr = 7（ALU 算术结果 3 or 5 = 7）

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

writedata = 7（第二个操作数$4 = 7）

dataadr = 4（ALU 算术结果 12 and 7 = 4）

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

writedata = 7（第二个操作数$4 = 7）

dataadr = 11（ALU 算术结果 4 + 7 = 11）

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

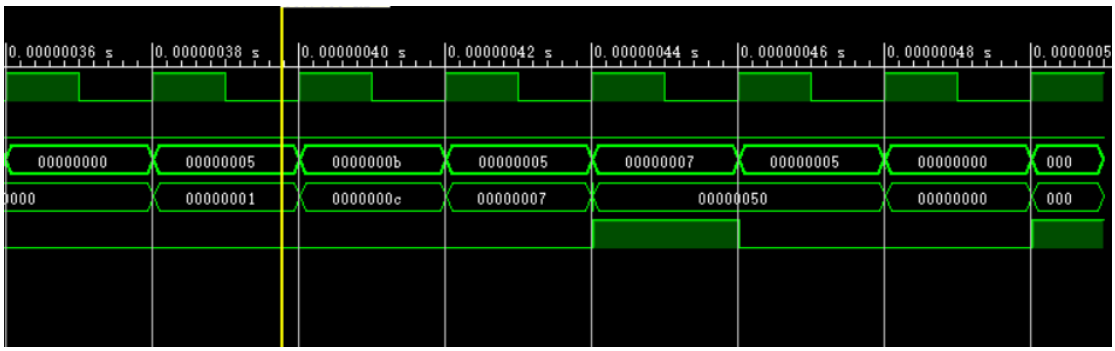
writedata = 3（第二个操作数$7 = 3）

dataadr = 8（ALU 算术结果 11 - 3 = 8）

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

writedata = 7（第二个操作数$4 = 7）

dataadr = 0（ALU 算术结果 12 < 7 = 0）



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

writedata = 0（第二个操作数$0 = 0）

dataadr = 0（ALU 算术结果 0 - 0 = 0）

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

writedata = 5（第二个操作数$2 = 5）

dataadr = 1（ALU 算术结果 3 < 5 = 1）

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

writedata = 11（第二个操作数$5 = 11）

dataadr = 12（ALU 算术结果 1 + 11 = 12）

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

writedata = 5（第二个操作数$2 = 5）

dataadr = 7（ALU 算术结果 12 - 5 = 7）

sw $2,68($3)　　ac670044　　[80] = 7

writedata = 7（$7 原本存储数据为 7）

dataadr = 80（ALU 算术结果 68 + 3 * 4 = 80）

memwrite = 1

lw $2,80($0)　　8c020050　　$2 = [80] = 7

writedata = 5（$2 原本存储数据为 5）

dataadr = 80（ALU 算术结果 80 + 0 * 4 = 80）

j end　　08000011　　跳至 end

writedata = 0

dataadr = 0

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

sw $2,84($0)　　ac020054　　[84] = 7

writedata = 7（$2 原本存储数据为 7）

dataadr = 84（ALU 算术结果 84 + 0 * 4 = 84）

memwrite = 1

程序运行完毕！

五、实验总结

单周期 CPU 实验是前几次实验的综合。需要对各模块 PC，register，ALU，controller，inst_mem，data_mem 的工作熟练掌握，也要对各指令如何运行十分了解，才能在编写复杂的 datapath 数据通路时不会出错。datapath 中用到了许多变量，刚开始我经常搞混淆，导致出现了很多奇怪的错误。最终经过一遍遍的检查，我才终于得到最终的正确程序。这次实验让我掌握了单周期 CPU 数据通路图的构成、原理及其设计方法，也掌握了单周期 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。