中山大學
SUN YAT-SEN UNIVERSITY

《计算机组成原理实验》

实验报告

（实验三）

学 院 名 称	：	数据科学与计算机学院
专业（班级）	：	17 软件工程 2 班
学 生 姓 名	：	郑佳豪
学 号	：	16305204
时 间	：	2018 年 12 月 26 日

成 绩 ：

实验三 ： 多周期CPU设计与实现

一. 实验目的

1. 认识和掌握多周期数据通路图的构成、原理及其设计方法；
2. 掌握多周期CPU的实现方法，代码实现方法；
3. 编写一个编译器，将MIPS汇编程序编译为二进制机器码；
4. 掌握多周期CPU的测试方法；
5. 掌握多周期CPU的实现方法。

二. 实验内容

设计一个多周期 CPU，该 CPU 至少能实现以下指令功能操作。需设计的指令与格式如下：

==>算术运算指令

（1）add rd, rs, rt

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

功能：rd<－rs + rt。

（2）sub rd, rs, rt

000001	rs(5 位)	rt(5 位)	rd(5 位)	reserved

完成功能：rd<－rs － rt。

（3）addiu rt, rs, **immediate**

000010	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt<－rs + (sign-extend)**immediate**。

==>逻辑运算指令

（4）and rd , rs , rt

010000	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs & rt；逻辑与运算。

（5）andi rt , rs ,**immediate**

010001	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt←rs & (zero-extend)**immediate**；**immediate** 做“0”扩展再参加“与”运算。

（6）ori rt , rs ,**immediate**

010010	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt←rs | (zero-extend)**immediate**；**immediate** 做“0”扩展再参加“或”运算。

（7）xori　rt , rs , **immediate**

010011	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt←rs ⊕ (zero-extend)**immediate；immediate** 做“0”扩展再参加“异或”运算。

==>移位指令

（8）sll　rd, rt,sa

011000	未用	rt(5 位)	rd(5 位)	sa	reserved

功能：rd<－rt<<(zero-extend)sa，左移 sa 位 ，(zero-extend)sa。

==>比较指令

（9）slti　rt, rs,immediate　　带符号

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

功能：if (rs <(sign-extend)**immediate)**　rt =1 else　rt=0，　具体请看表 2 ALU 运算功能表，带符号。

（10）slt　rd, rs, rt　　带符号

100111	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：if (rs<rt)　rd =1 else　rd=0，具体请看表 2 ALU 运算功能表，带符号。

==>存储器读写指令

（11）sw　rt, **immediate**(rs)

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

功能：memory[rs+ (sign-extend)**immediate**]<－rt。即将 rt 寄存器的内容保存到 rs 寄存器内容和立即数符号扩展后的数相加作为地址的内存单元中。

（12）lw　rt, **immediate**(rs)

110001	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt <－ memory[rs + (sign-extend)**immediate**]。即读取 rs 寄存器内容和立即数符号扩展后的数相加作为地址的内存单元中的数，然后保存到 rt 寄存器中。

==>分支指令

(13) beq　rs,rt, **immediate** (说明：**immediate 从 pc+4 开始和转移到的指令之间间隔条数**)

110100	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：if(rs=rt) pc <－pc + 4 + ((sign-extend)**immediate** <<2) else pc <－pc + 4。

(14) bne　rs,rt, **immediate** (说明：**immediate 从 pc+4 开始和转移到的指令之间间隔条数**)

110101	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：if(rs!=rt) pc <－pc + 4 + ((sign-extend)**immediate** <<2) else pc <－pc + 4。

（15）bltz　rs,**immediate**

110110	rs(5 位)	00000	**immediate**

功能：if(rs<$0) pc←pc + 4 +((sign-extend)**immediate** <<2)　else pc ←pc + 4。

==>跳转指令

（16）j　addr

111000	addr[27:2]

功能：pc <－{(pc+4)[31:28],addr[27:2],2'b00}，跳转。

说明：由于 MIPS32 的指令代码长度占 4 个字节，所以指令地址二进制数最低 2 位均为 0，将指令地址放进指令代码中时，可省掉！这样，除了最高 6 位操作码外，还有 26 位可用于存放地址，事实上，可存放 28 位地址，剩下最高 4 位由 pc+4 最高 4 位拼接上。

（17）jr　rs

111001	rs(5 位)	未用	未用	reserved

功能：pc <－ rs，跳转。

==>调用子程序指令

（18）jal　addr

111010	addr[27:2]

功能：调用子程序，pc <－ {(pc+4)[31:28],addr[27:2],2'b00}；$31<－pc+4，返回地址设置；子程序返回，需用指令 jr $31。跳转地址的形成同 j addr 指令。

==>停机指令

（19）halt (停机指令)

111111	00000000000000000000000000(26 位)

不改变 pc 的值，pc 保持不变。

在本文档中，提供的相关内容对于设计可能不足或甚至有错误，希望同学们在设计过程中如发现有问题，请你们自行改正，进一步补充、完善。谢谢！

三．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。

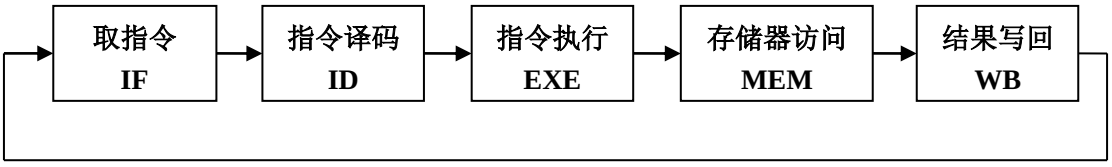
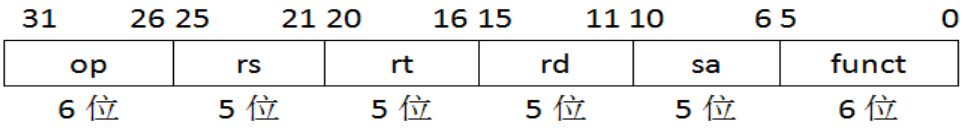


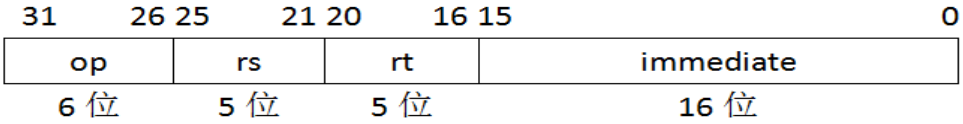
图 1　多周期 CPU 指令处理过程

MIPS 指令的三种格式:

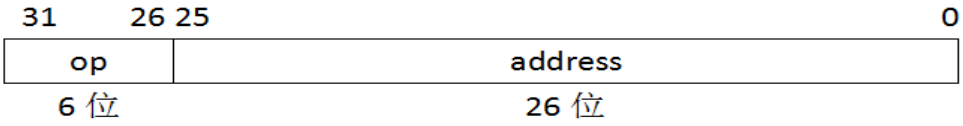
R 类型：



I 类型：



J 类型：



其中，

op：为操作码；

rs：为第 1 个源操作数寄存器，寄存器地址（编号）是 00000~11111，00~1F；

rt：为第 2 个源操作数寄存器，或目的操作数寄存器，寄存器地址（同上）；

rd：为目的操作数寄存器，寄存器地址（同上）；

sa：为位移量（shift amt），移位指令用于指定移多少位；

funct：为功能码，在寄存器类型指令中（R 类型）用来指定指令的功能；

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（Load）/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；

address：为地址。

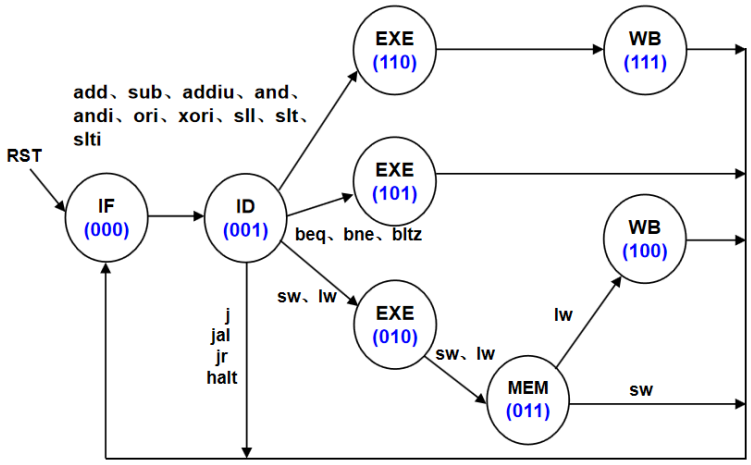


图 2 多周期 CPU 状态转移图

状态的转移有的是无条件的，例如从 sIF 状态转移到 sID 就是无条件的；有些是有条件的，例如 sEXE 状态之后不止一个状态，到底转向哪个状态由该指令功能，即指令操作码决定。每个状态代表一个时钟周期。

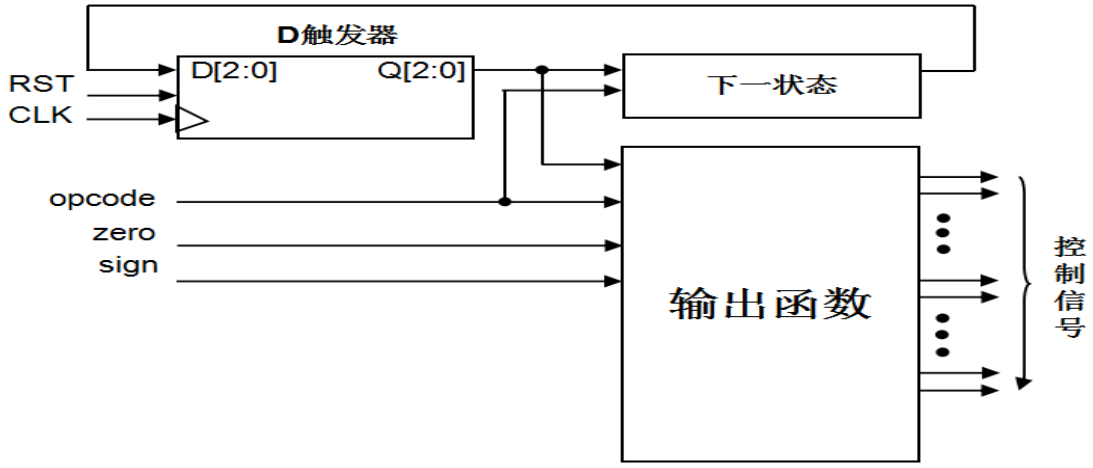


图 3 多周期 CPU 控制部件的原理结构图

图 3 是多周期 CPU 控制部件的电路结构，三个 D 触发器用于保存当前状态，是时序逻辑电路，RST 用于初始化状态“000“，另外两个部分都是组合逻辑电路，一个用于产生下一个阶段的状态，另一个用于产生每个阶段的控制信号。从图上可看出，下个状态取决于指令操作码和当前状态；而每个阶段的控制信号取决于指令操作码、当前状态和反映运算结果的状态 zero 标志和符号 sign 标志。

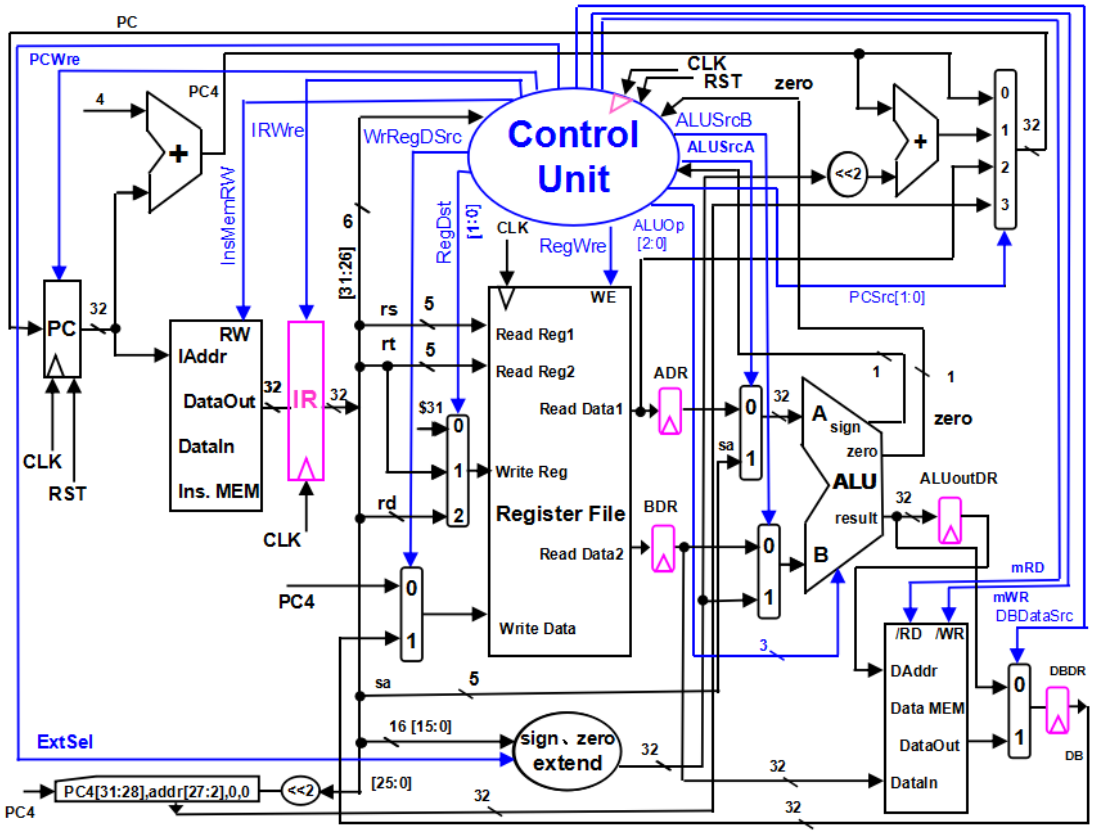


图 4　多周期 CPU 数据通路和控制线路图

图 4 是一个简单的基本上能够在多周期 CPU 上完成所要求设计的指令功能的数据通路和必要的控制线路图。其中指令和数据各存储在不同存储器中，即有指令存储器和数据存储

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，给出寄存器地址（编号），读操作时不需要时钟信号，输出端就直接输出相应数据；而在写操作时，在 WE 使能信号为 1 时，在时钟边沿触发将数据写入寄存器。图中控制信号功能如表 1 所示，表 2 是 ALU 运算功能表。

特别提示，图上增加 IR 指令寄存器，目的是使指令代码保持稳定，pc 写使能控制信号 PCWre，是确保 pc 适时修改，原因都是和多周期工作的 CPU 有关。ADR、BDR、ALUoutDR、DBDR 四个寄存器**不需要写使能信号，**其作用是切分数据通路，将大组合逻辑切分为若干个小组合逻辑，大延迟变为多个分段小延迟。

表 1 控制信号作用

控制信号名	状态“0”	状态“1”
RST	对于 PC，初始化 PC 为程序首地址	对于 PC，PC 接收下一条指令地址
PCWre	PC 不更改，相关指令：halt，另外，除‘000’状态之外，其余状态慎改 PC 的值。	PC 更改，相关指令：除指令 halt 外，另外，在‘000’状态时，修改 PC 的值合适。
ALUSrcA	来自寄存器堆 data1 输出，相关指令：add、sub、addiu、and、andi、ori、xori、slt、slti、sw、lw、beq、bne、bltz	来自移位数 sa，同时，进行(zero-extend)sa，即 {{27{1'b0},sa}，相关指令：sll
ALUSrcB	来自寄存器堆 data2 输出，相关指令：add、sub、and、slt、sll、beq、bne、bltz	来自 sign 或 zero 扩展的立即数，相关指令：addiu、andi、ori、xori、slti、lw、sw
DBDataSrc	来自 ALU 运算结果的输出，相关指令：add、sub、addiu、and、andi、ori、xori、sll、slt、slti	来自数据存储器（Data MEM）的输出，相关指令：lw
RegWre	无写寄存器组寄存器，相关指令：beq、bne、bltz、j、sw、jr、halt	寄存器组寄存器写使能，相关指令：add、sub、addiu、and、andi、ori、xori、sll、slt、slti、lw、jal
WrRegDSrc	写入寄存器组寄存器的数据来自 pc+4(pc4），相关指令：jal，写 $31	写入寄存器组寄存器的数据来自 ALU 运算结果或存储器读出的数据，相关指令：add、addiu、sub、and、andi、ori、xori、sll、slt、slti、lw
InsMemRW	写指令存储器	读指令存储器(Ins. Data)
mRD	存储器输出高阻态	**读数据存储器，相关指令：lw**
mWR	无操作	**写数据存储器，相关指令：sw**
IRWre	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。
ExtSel	(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<－rs，相关指令：jr； 11：pc<－{pc[31:28],addr[27:2],2'b00}，相关指令：j、jal；
RegDst[1..0]	写寄存器组寄存器的地址，来自： 00：0x1F($31)，相关指令：jal，用于保存返回地址（$31<-pc+4）； 01：rt 字段，相关指令：addiu、andi、ori、xori、slti、lw； 10：rd 字段，相关指令：add、sub、and、slt、sll； 11：未用；
ALUOp[2..0]	ALU 8 种运算功能选择(000-111)，看功能表

相关部件及引脚说明：

Instruction Memory：指令存储器

Iaddr，指令地址输入端口

DataIn，存储器数据输入端口

DataOut，存储器数据输出端口

RW，指令存储器读写控制信号，为 0 写，为 1 读

Data Memory：数据存储器

Daddr，数据地址输入端口

DataIn，存储器数据输入端口

DataOut，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

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，其地址输入端口（rt、rd）

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

IR：　指令寄存器，用于存放正在执行的指令代码

ALU： 算术**逻辑单元**

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

表 2 ALU 运算功能表

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<B 不带符号
110	Y=(((A<B)&&(A[31] == B[31])) ||((A[31]==1&& B[31] == 0))) ? 1:0	比较 A<B 带符号
111	Y = A ⊕ B	异或

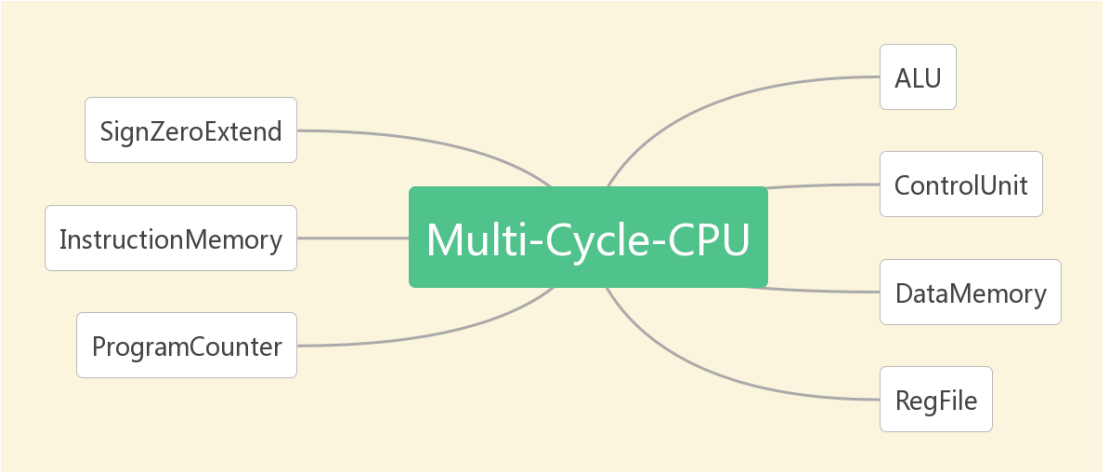
值得注意的问题，设计时，用模块化、层次化的思想方法设计，关于如何划分模块、如何整合成一个系统等等，是必须认真考虑的问题。

四. 实验器材

电脑一台，Xilinx Vivado 软件一套，Basys3板一块

五. 实验过程与结果

1. 设计思想



参照上面提及到的数据通路，我将项目拆分为上图所示的几个部分。Multi-Cycle-CPU为顶层模块，用来将其余模块联系。

- ControlUnit：控制数据路线
- ALU：负责算术逻辑运算
- DataMemory：数据存储器
- SignZeroExtend：对立即数进行零扩展和符号扩展
- InstructionMemory：指令存储器
- Program counter：负责程序要处理指令的提供
- RegFile：寄存器堆

2. 模块设计

1) PC

该模块负责为处理器提供下一条要处理指令的地址。本模块划分为三大模块：PC、PC_Source 和 PC_Jumper 模块。PC_Source 模块控制PC地址数据的来源，PC_Jumper 模块则负责跳转地址的计算，PC 模块则根据前两个模块的响应，提供正确的下一条指令的地址。

```
`timescale 1ns / 1ps

module PC(input CLK, input RST, input PCWre,
  input [31:0] pc, output reg [31:0] IAddr = 0,
  output [31:0] PC4
);
  always @(posedge CLK or negedge RST) begin
    if (!RST) begin
      IAddr <= 0;
    end
    else if (PCWre) begin
      IAddr <= pc;
    end
  end
  assign PC4 = IAddr + 4;
endmodule

module PC_Source (input [31:0] PC4, input [31:0] PC_Brench,
  input [31:0] PC_JumpRigister, input [31:0] PC_Jump,
  input [1:0] PC_Src,output reg [31:0] pc
);
  always @(*) begin
    case(PC_Src)
      2'b00: pc <= PC4;
      2'b01: pc <= (PC_Brench << 2) + PC4;
      2'b10: pc <= PC_JumpRigister;
      2'b11: pc <= PC_Jump;
    endcase
  end
endmodule

module PC_Jumper (input [31:0] PC4, input [25:0] pc,
  output [31:0] PC_Jump
);
  assign PC_Jump[31:28] = PC4[31:28];
  assign PC_Jump[27:0] = pc << 2;
endmodule
```

2) InstructionMemory

该模块根据当前的PC地址，得到对应地址的指令。

此模块整合了 IR 寄存器。

```
`timescale 1ns / 1ps

module IM(input CLK, input [31:0] IAddr, input RW,
  input IRWre, output reg [31:0] IROut
);
  reg [7:0] Mem[0:127];
  reg [31:0] IDataOut;
  initial begin
    $readmemb("D:/Multi-Cycle-CPU/data/instructions.txt", Mem);
  end
  always @(*) begin
    if (RW == 1) begin
      IDataOut[31:24] <= Mem[IAddr];
      IDataOut[23:16] <= Mem[IAddr + 1];
      IDataOut[15:8] <= Mem[IAddr + 2];
      IDataOut[7:0] <= Mem[IAddr + 3];
    end
  end
  always @(posedge CLK) begin
    if (IRWre == 1) begin
        IROut[31:0] <= IDataOut[31:0];
    end
  end
endmodule
```

3) SignZeroExtend

该模块提供将16位立即数扩展到32位的功能。

```
`timescale 1ns / 1ps

module SZE(
    input ExtSel,
    input [15:0] Origin,
    output [31:0] Extended
);

    assign Extended[15:0] = Origin[15:0];
    assign Extended[31:16] = (ExtSel && Origin[15]) ? 16'hffff : 16'h0000;

endmodule
```

4) RegFile

该模块为其余模块提供读取和写入寄存器的功能。

```
`timescale 1ns / 1ps

module RF(input CLK, input WE, input WrRegDSrc,
    input [1:0] RegDst, input [4:0] RReg1,
    input [4:0] RReg2, input [4:0] rd,
    input [31:0] DB, input [31:0] PC4,
    output reg [31:0] ReadData1, output reg [31:0] ReadData2
);
    reg [4:0] WReg;
    wire [31:0] WriteData;
    always @(*) begin
        case(RegDst)
            2'b00:
                WReg <= 5'b11111;
            2'b01:
                WReg <= RReg2;
            2'b10:
                WReg <= rd;
        endcase
    end
    assign WriteData = (WrRegDSrc) ? DB : PC4;
    reg [31:0] Regs[0:31];
    integer i = 0;
    initial begin
      repeat(32) begin
        Regs[i] = 0;
        i = i + 1;
      end
    end
    always @(*) begin
        ReadData1 <= Regs[RReg1];
        ReadData2 <= Regs[RReg2];
    end
    always @(negedge CLK) begin
      if (WE) begin
        Regs[WReg] <= WriteData;
      end
    end
endmodule
```

5) DataMemory

该模块能提供读取、写入数据的功能，在时钟下降沿且RD为1时，表现为读取数据到DB；在时钟下降沿且WR为1时，表现为写入数据到RAM。

此模板整合了 DRDR 寄存器。

```
`timescale 1ns / 1ps

module DM(input CLK, input RD, input WR,
  input DBDataSrc, input [31:0] result,
  input [31:0] DAddr,input [31:0] DataIn,
  output reg [31:0] DB
);
  wire [31:0] DataOut;
  reg [7:0] Data_Mem[0:127];
  integer i = 0;
  initial begin
    repeat(128) begin
        Data_Mem[i] = 0;
        i = i + 1;
    end
  end
  assign DataOut[7:0] = (RD) ? Data_Mem[DAddr + 3] : 8'bz;
  assign DataOut[15:8] = (RD) ? Data_Mem[DAddr + 2] : 8'bz;
  assign DataOut[23:16] = (RD) ? Data_Mem[DAddr + 1] : 8'bz
; assign DataOut[31:24] = (RD) ? Data_Mem[DAddr] : 8'bz;
  always @(negedge CLK) begin
    if (WR) begin
      Data_Mem[DAddr] <= DataIn[31:24];
      Data_Mem[DAddr + 1] <= DataIn[23:16];
      Data_Mem[DAddr + 2] <= DataIn[15:8];
      Data_Mem[DAddr + 3] <= DataIn[7:0];
    end
  end
  always @(*) begin
    DB <= (DBDataSrc) ? DataOut : result;
  end
endmodule
```

6) ALU

该模块接收寄存器的数据和控制信号的输入，将运算后的结果输出。

此模块整合了 ADR 和 BDR 寄存器。

```
`timescale 1ns / 1ps

module ALU(
  input ALUSrcA,
  input ALUSrcB,
  input [31:0] ReadData1,
  input [31:0] ReadData2,
  input [4:0] sa,
  input [31:0] sze,
  input [2:0] ALUOp,
  output zero,
  output sign,
  output reg [31:0] result
);

  wire [31:0] A;
  wire [31:0] B;

  assign A[4:0] = ALUSrcA ? sa : ReadData1[4:0];
  assign A[31:5] = ALUSrcA ? 0 : ReadData1[31:5];
  assign B = ALUSrcB ? sze : ReadData2;
  assign zero = result == 0;
  assign sign = result[31];

  always @(*) begin
    case(ALUOp)
      3'b000: result = A + B;
      3'b001: result = A - B;
      3'b010: result = B << A;
      3'b011: result = A | B;
      3'b100: result = A & B;
      3'b101: result = (A < B) ? 1 : 0;
      3'b110: result = ((A < B && A[31] == B[31]) || (A[31] == 1 && B[31] == 0)) ? 1 : 0
;     3'b111: result = A ^ B;
    endcase
  end

endmodule
```

7) CU

该模块控制数据通路的路线，即控制其余各模块的数据通路。由于代码量较大和篇幅的限制，此处不贴出具体代码，具体可参照附件。

3. 模块测试

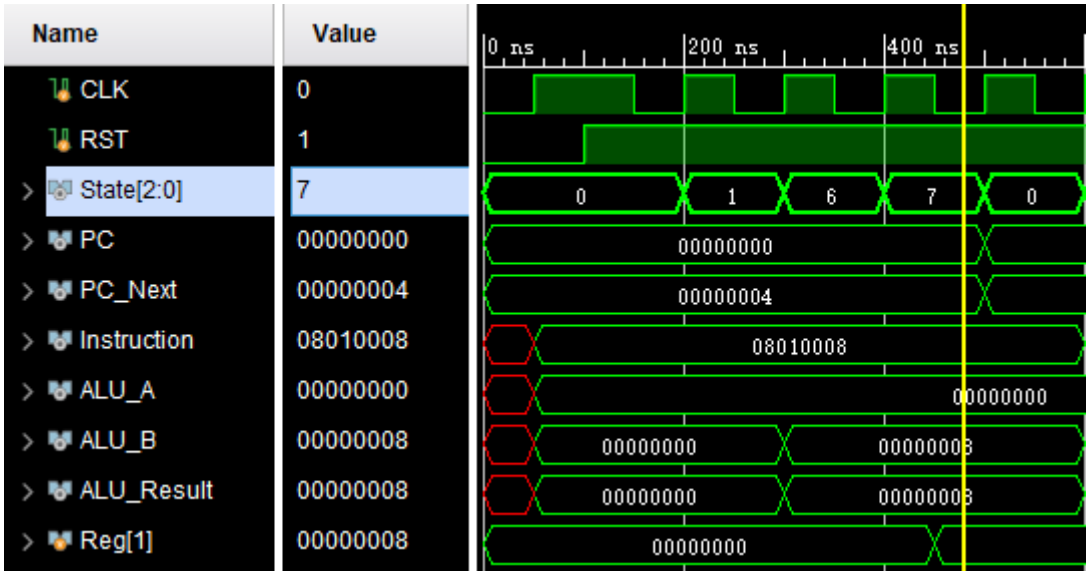
1) 测试代码

地址	汇编程序	指令代码					
		op（6）	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码	
0x00000000	addiu $1,$0,8	000010	00000	00001	0000 0000 0000 1000	=	08010008
0x00000004	ori $2,$0,2	010010	00000	00010	0000 0000 0000 0010	=	48020002
0x00000008	xori $3,**$2**,8	010011	00010	00011	0000 0000 0000 1000	=	4C430008
0x0000000C	sub $4,**$3**,$1	000001	00011	00001	0010 0000 0000 0000	=	04612000
0x00000010	and $5,**$4**,$2	010000	00100	00010	0010 1000 0000 0000	=	40822800
0x00000014	**sll $5,$5,2**	011000	00000	00101	0010 1000 1000 0000	=	60052880
0x00000018	**beq $5,$1,-2(=,转 14)**	110100	00101	00001	1111 1111 1111 1110	=	D0A1FFFE
0x0000001C	**jal 0x0000050**	111010	00000	00000	0000 0000 0101 0000	=	E8000050
0x00000020	**slt $8,$13,$1**	100111	01101	00001	0100 0000 0000 0000	=	9DA14000
0x00000024	**addiu $14,$0,-2**	000010	00000	01110	1111 1111 1111 1110	=	080EFFFE
0x00000028	**slt $9,$8,$14**	100111	01000	01110	0100 1000 0000 0000	=	9D0E4800
0x0000002C	**slti $10,$9,2**	100110	01001	01010	0000 0000 0000 0010	=	992A0002
0x00000030	**slti $11,$10,0**	100110	01010	01011	0000 0000 0000 0000	=	994B0000
0x00000034	add $11,**$11**,$10	000000	010110	01010	0101 1000 0000 0000	=	016A5800
0x00000038	**bne $11,$2,-2 (≠,转 34)**	110101	010110	00010	1111 1111 1111 1110	=	D562FFFE
0x0000003C	addiu $12,$0,-2	000010	00000	01100	1111 1111 1111 1110	=	080CFFFE
0x00000040	addiu $12,$12,1	000010	01100	01100	0000 0000 0000 0001	=	098C0001
0x00000044	**bltz $12,-2 (<0,转 40)**	110110	01100	00000	1111 1111 1111 1110	=	D980FFFE
0x00000048	andi $12,$2,2	010001	00010	01100	0000 0000 0000 0010	=	444C0002
0x0000004C	**j 0x000005C**	111000	00000	00000	0000 0000 0101 1100	=	E000005C
0x00000050	sw $2,4($1)	110000	00001	00010	0000 0000 0000 0100	=	C0220004
0x00000054	lw $13,4($1)	110001	00001	01101	0000 0000 0000 0100	=	C42D0004
0x00000058	**jr $31**	111001	11111	00000	0000 0000 0000 0000	=	E7E00000
0x0000005C	**halt**	111111	00000	00000	0000 0000 0000 0000	=	FC000000

2) 功能验证

i.

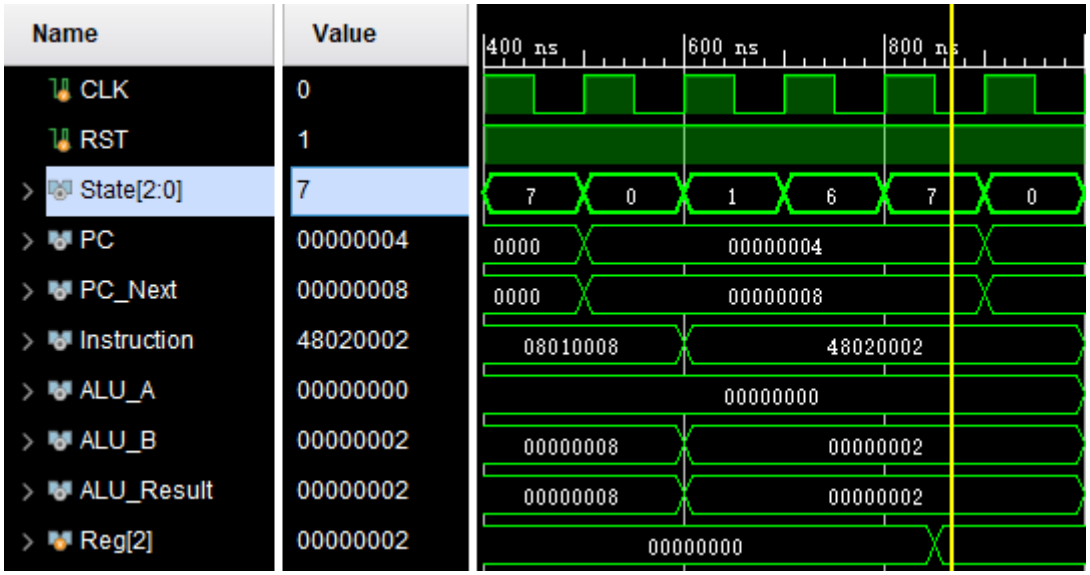
0x00000000	addiu $1,$0,8	$1=8	000010	00000	00001	0000 0000 0000 1000	08010008



该指令表示为与无符号立即数的加法运算，ALU_A为寄存器$0的内容（0），ALU_B为立即数8，ALU运算结果为8。最后，ALU运算结果会被写入寄存器$1中，其内容为8。

ii.

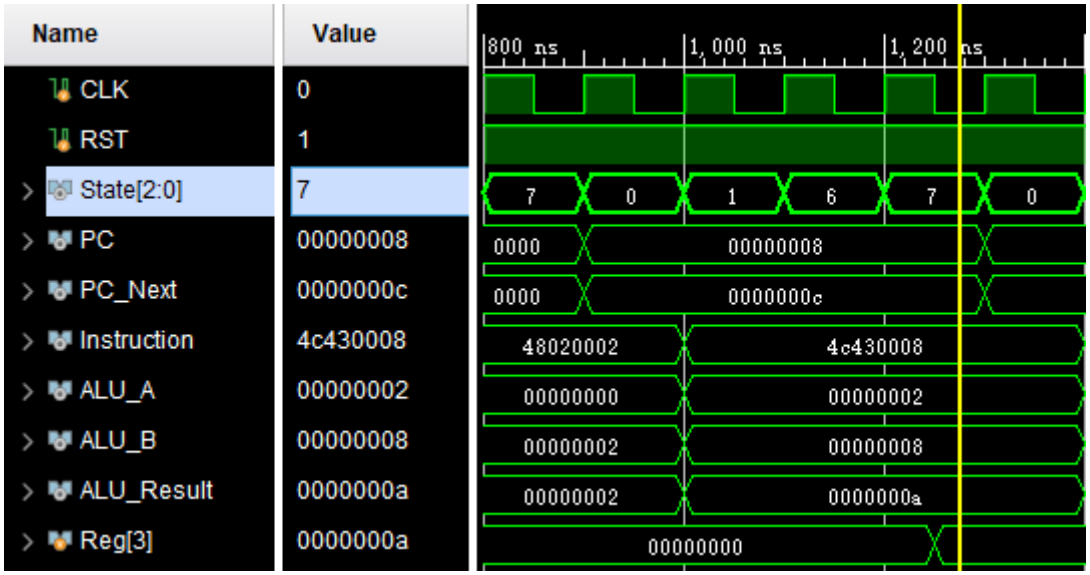
0x00000004	ori $2,$0,2	$2=2	010010	00000	00010	0000 0000 0000 0010	48020002



该指令表示与立即数的或运算，ALU_A为寄存器$0的内容（0），ALU_B为立即数2，ALU运算结果为2。最后，ALU运算结果会被写入寄存器$2中，其内容为2。

iii.

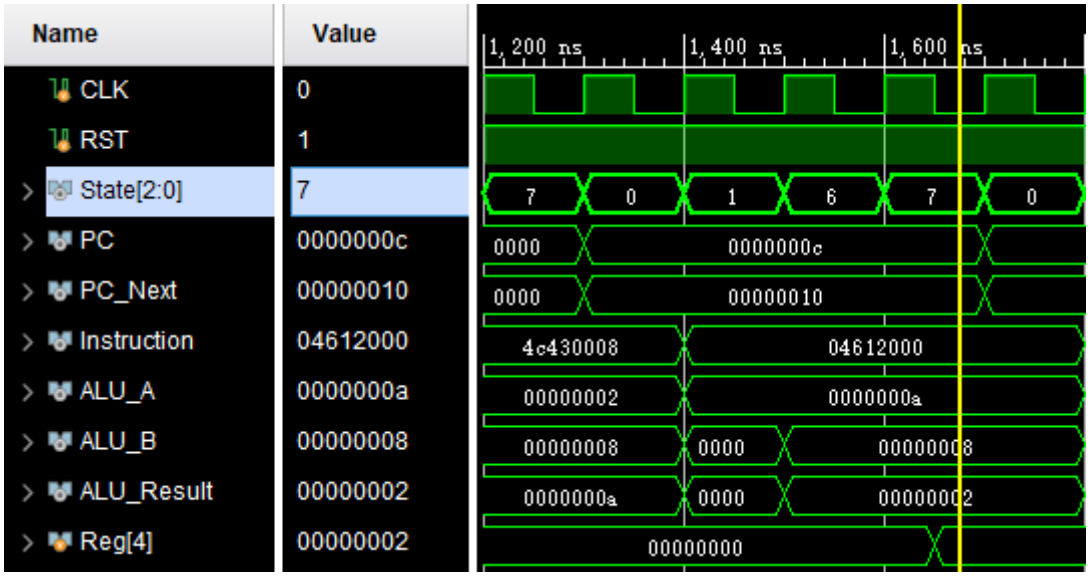
0x00000008	xori $3,$2,8	$3=10	010011	00010	00011	0000 0000 0000 1000	4C430008



该指令表示与立即数的异或运算，ALU_A为寄存器$2的内容（2），ALU_B为立即数8，ALU运算结果为10。最后，ALU运算结果会被写入寄存器$3中，其内容为10。

iv.

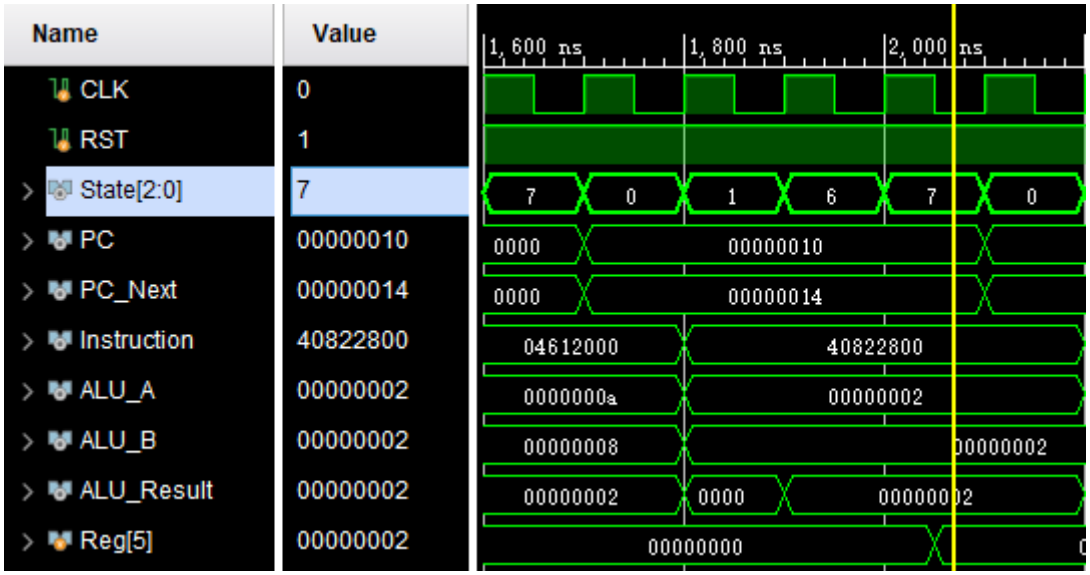
0x0000000C	sub $4,$3,$1	$4=2	000001	00011	00001	0010 0000 0000 0000	04612000



该指令表示减法运算，ALU_A为寄存器$3的内容（10），ALU_B为寄存器$1的内容（8），ALU运算结果为2。最后，ALU运算结果会被写入寄存器$4中，其内容为2。

v.

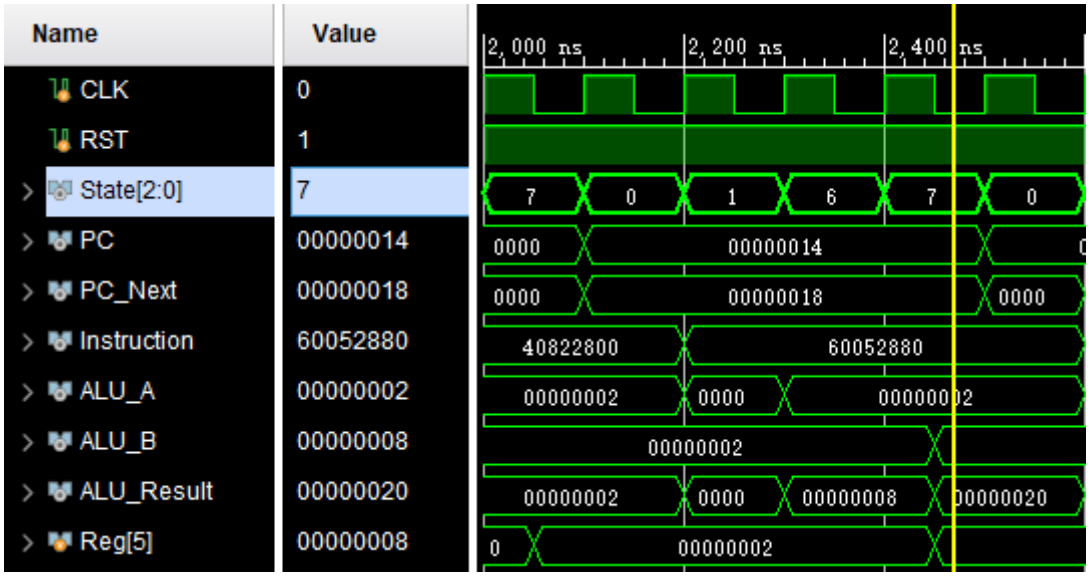
0x00000010	and $5,$4,$2	$5=1	010000	00100	00010	0010 1000 0000 0000	40822800



该指令表示与运算，ALU_A为寄存器$4的内容（2），ALU_B为寄存器$2的内容（2），ALU运算结果为2。最后，ALU运算结果会被写入寄存器$5中，其内容为2。

vi.

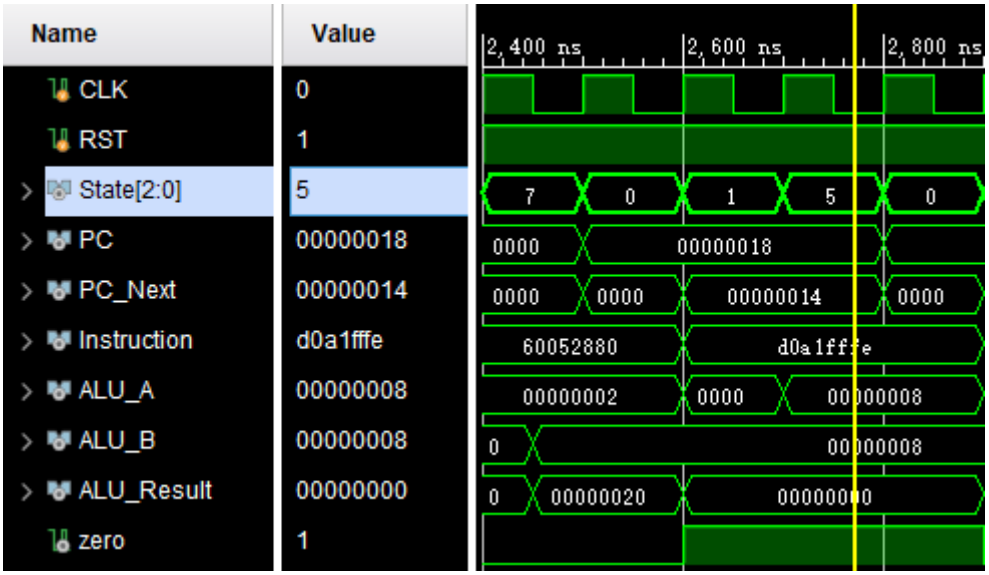
0x00000014	sll $5,$5,2	$5=8	011000	00000	00101	0010 1000 1000 0000	60052880



该指令表示左移位运算，ALU_A为寄存器$5的内容（2），ALU_B为立即数2，ALU运算结果为8。最后，ALU运算结果会被写入寄存器$5中，其内容为8。

vii.

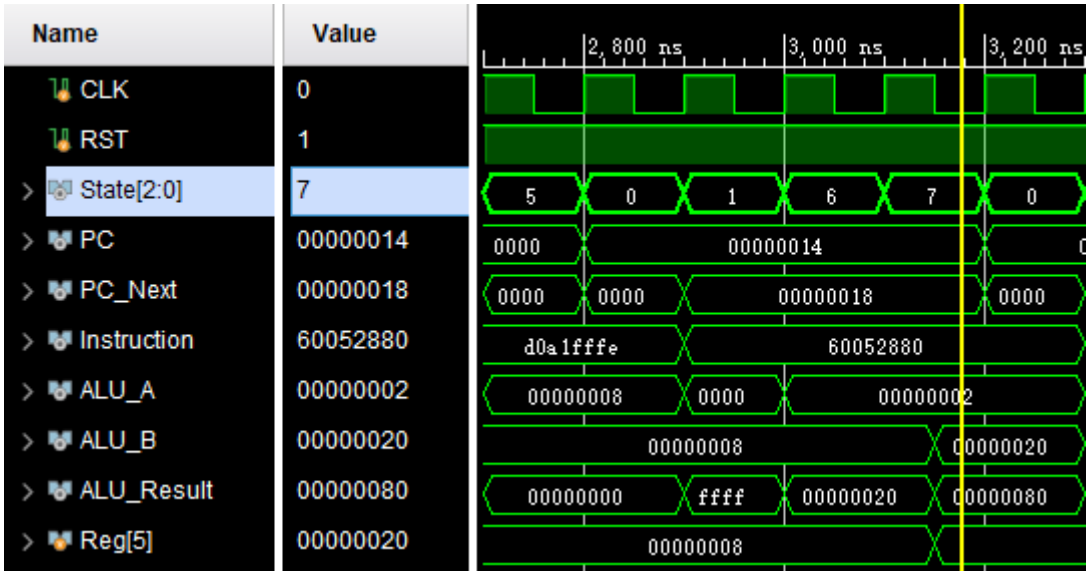
0x00000018	beq $5,$1,-2(=，转 14)	跳转至 14	110100	00101	00001	1111 1111 1111 1110	D0A1FFFE



该指令表示条件跳转，ALU_A为寄存器$5的内容（8），ALU_B为寄存器$1的内容（8），ALU运算结果zero为1，表示$5和$1内容相等，故下一条指令地址为nextPC=currentPC+immediate<<2，即跳转至0x00000014。

viii.

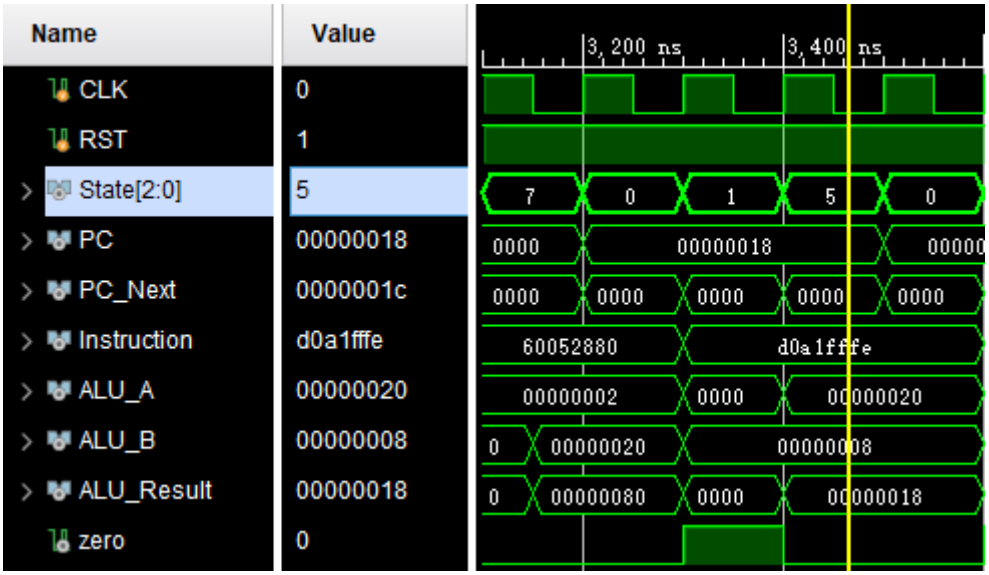
0x00000014	sll $5,$5,2	$5=32	011000	00000	00101	0010 1000 1000 0000	60052280



该指令表示左移位运算，ALU_A为寄存器$5的内容（8），ALU_B为立即数2，ALU运算结果为32。最后，ALU运算结果会被写入寄存器$5中，其内容为32。

ix.

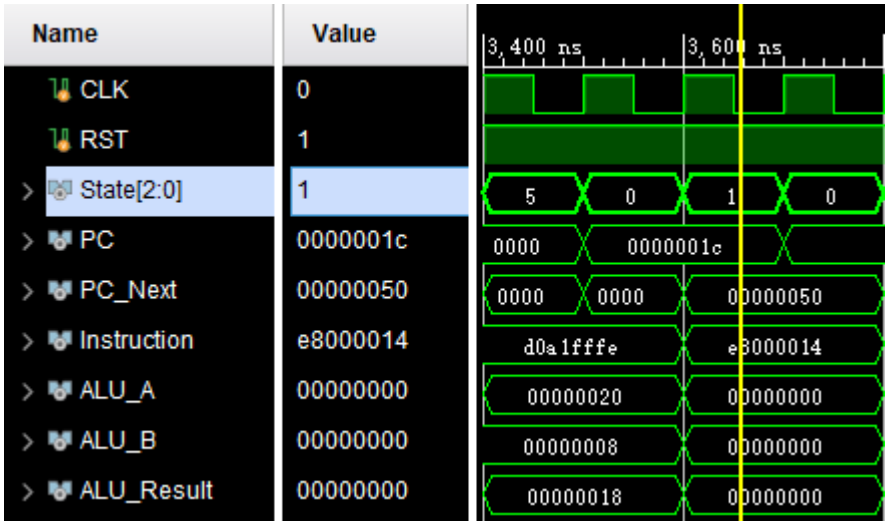
0x00000018	beq $5,$1,-2(=，转 14)	执行 1C	110100	00101	00001	1111 1111 1111 1110	D0A1FFFE



该指令表示条件跳转，ALU_A为寄存器$5的内容（32），ALU_B为寄存器$1的内容（8），ALU运算结果zero为0，表示$5和$1内容不相等，故下一条指令地址为nextPC=currentPC+4，即跳转至0x0000001C。

x.

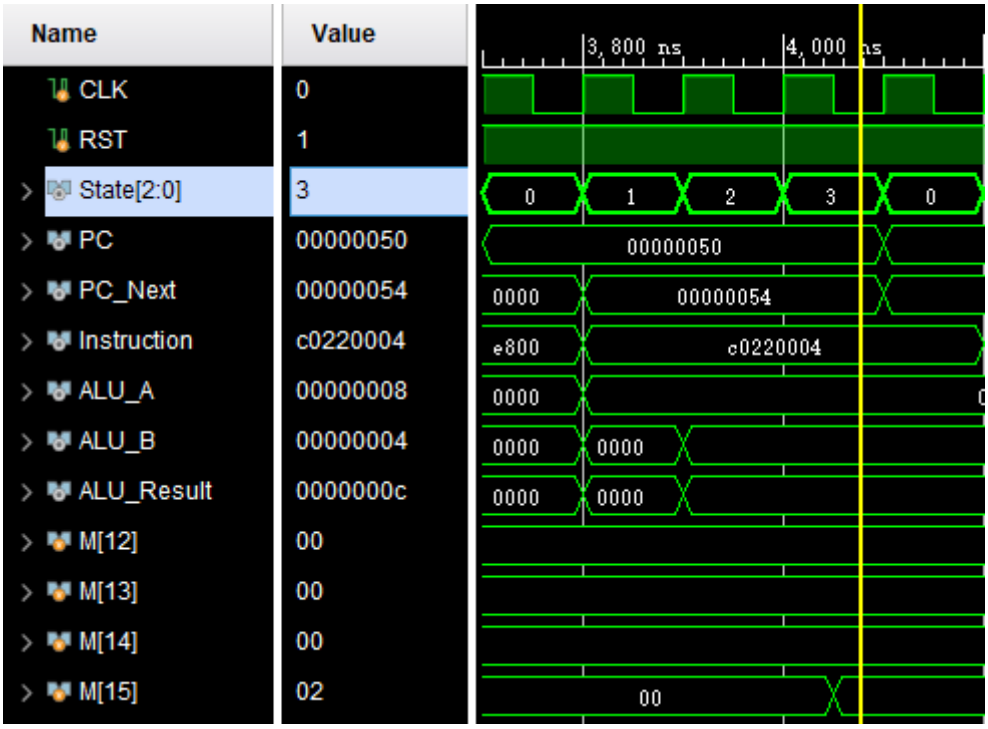
0x0000001C	jal 0x00000050	地址跳转	111010	00000	00000	0000 0000 0101 0000	E8000014



该指令表示无条件跳转，故下一条指令地址为0x00000050。

xi.

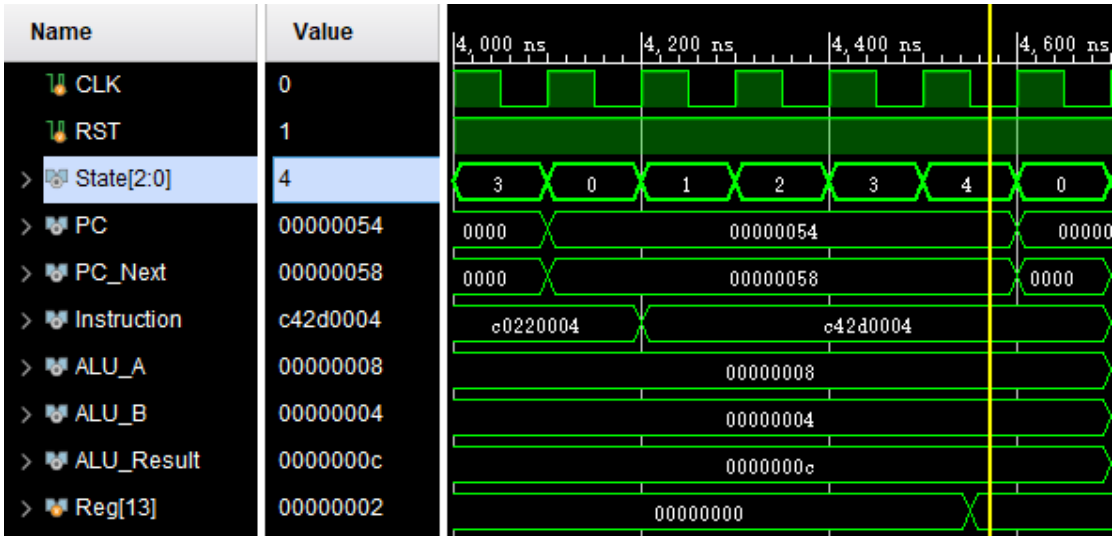
0x00000050	sw $2,4($1)	M[12-15]=0002	110000	00001	00010	0000 0000 0000 0100	C0220004



该指令表示写入存储器，即将寄存器$2的内容（2）写入到地址为4+($1)的内存单元，即12号内存单元。由于使用了大端存储，故M[12-15]=0002。

xii.

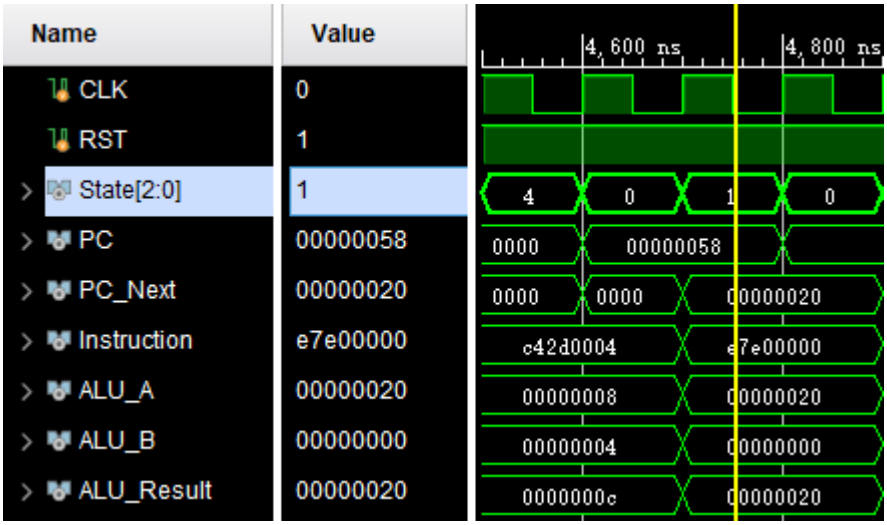
0x00000054	lw $13,4($1)	$13=2	110001	00001	01101	0000 0000 0000 0100	C42D0004



该指令表示读取存储器，即读取地址为4+($1)的内存单元至寄存器$13。故最后内存单元被读取至$13，其内容为2。

xiii.

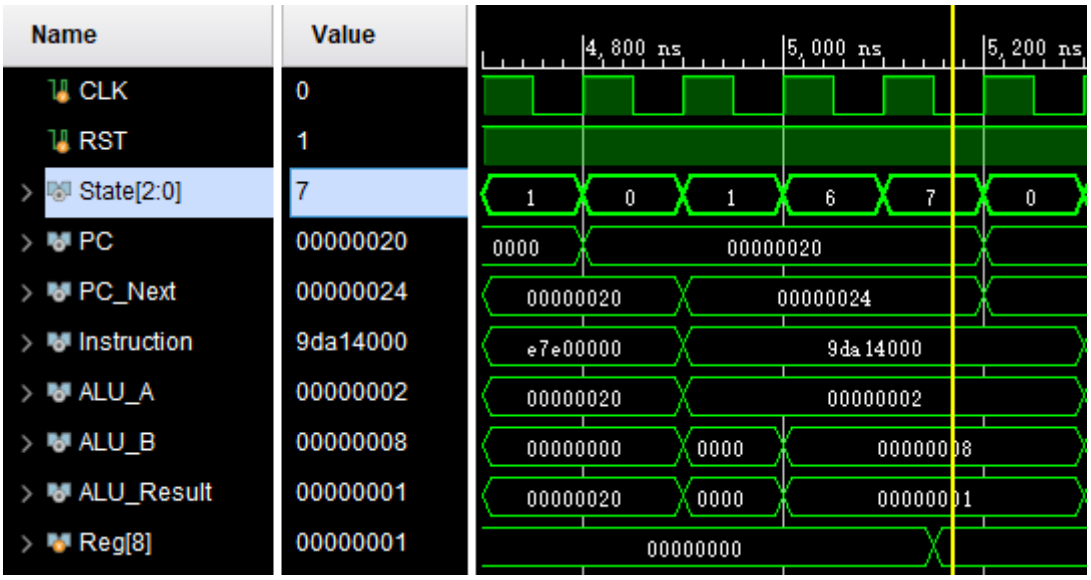
0x00000058	jr $31	跳转返回	111001	11111	00000	0000 0000 0000 0000	E7E00000



该指令表示跳转返回指令，故下一指令地址为0x00000020。

xiv.

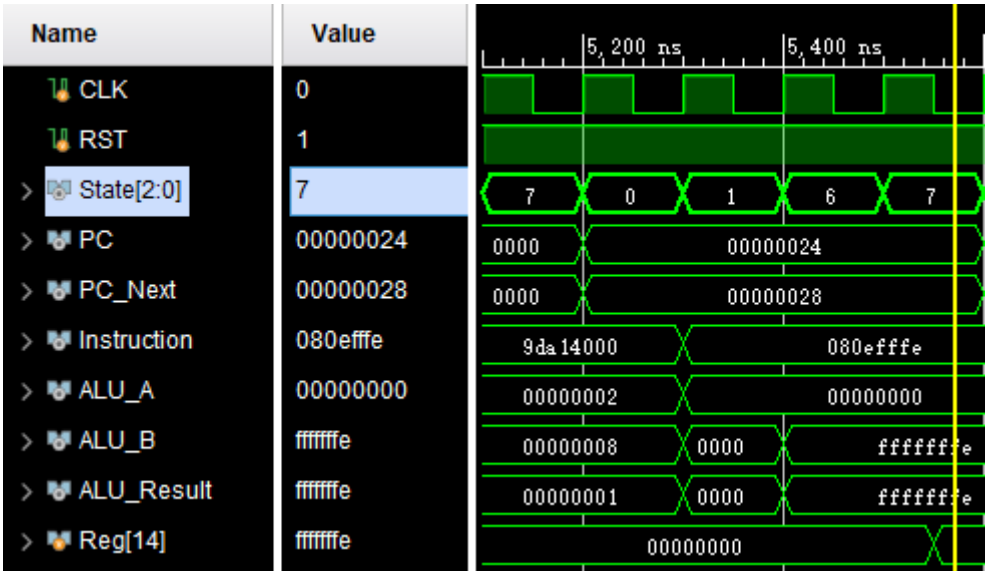
0x00000020	slt $8,$13,$1	$8=1	100111	01101	00001	0100 0000 0000 0000	9DA14000



该指令表示比较运算，ALU_A为寄存器$13的内容（2），ALU_B为寄存器$1的内容（8），由于$13<$1，故ALU运算结果为1。最后，ALU运算结果会被写入寄存器$8中，其内容为1。

xv.

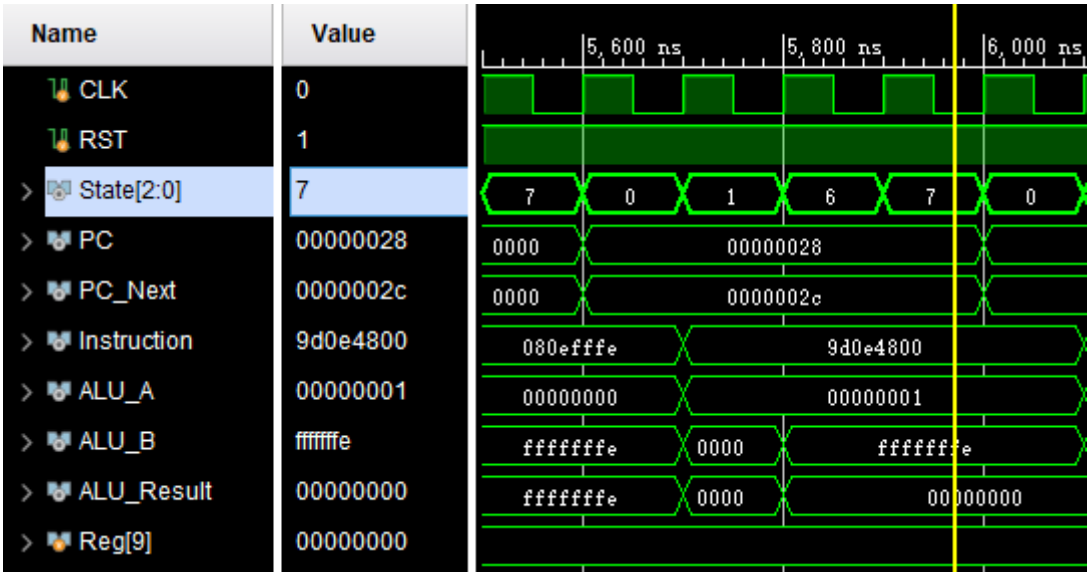
0x00000024	addiu $14,$0,-2	$14=-2	000010	00000	01110	1111 1111 1111 1110	080EFFFE



该指令表示与无符号立即数的加法运算，ALU_A为寄存器$0的内容（0），ALU_B为立即数-2，ALU运算结果为-2。最后，ALU运算结果会被写入寄存器$14中，其内容为-2。

xvi.

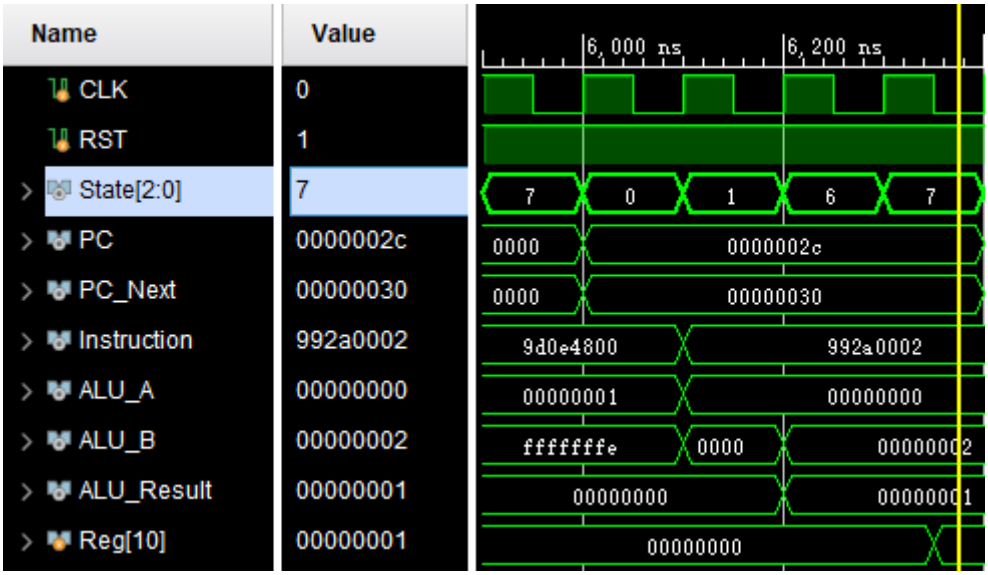
0x00000028	slt $9,$8,$14	$9=0	100111	01000	01110	0100 1000 0000 0000	9D0E4800



该指令表示比较运算，ALU_A为寄存器$8的内容（1），ALU_B为寄存器$14的内容（-2），由于$8>$14，故ALU运算结果为0。最后，ALU运算结果会被写入寄存器$9中，其内容为0。

xvii.

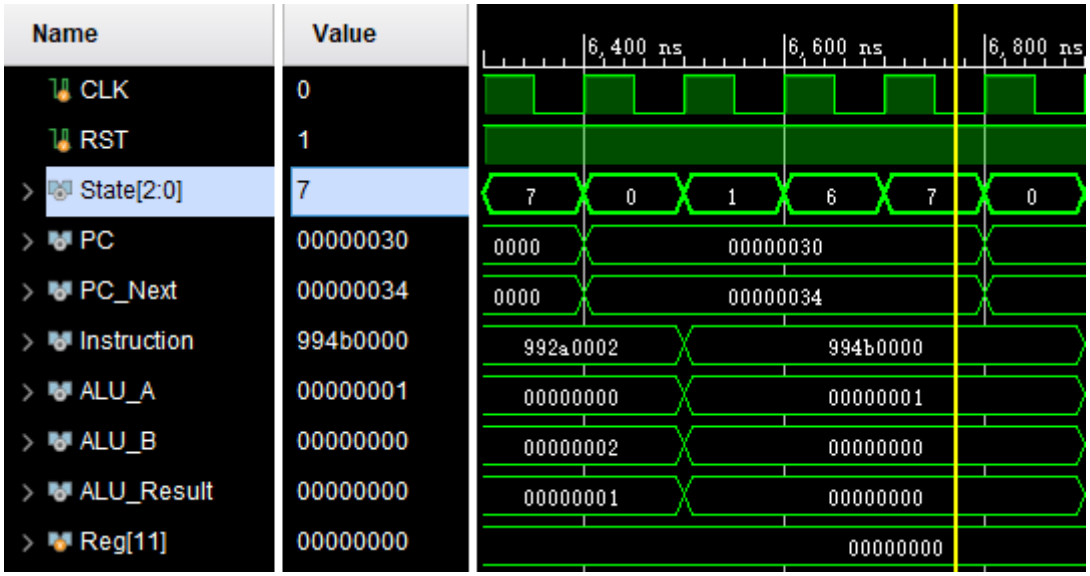
0x0000002C	slit $10,$9,2	$10=1	100110	01001	01010	0000 0000 0000 0010	992A0002



该指令表示与立即数的比较运算，ALU_A为寄存器$9的内容（0），ALU_B为立即数2，由于$9<2，故ALU运算结果为1。最后，ALU运算结果会被写入寄存器$10中，其内容为1。

xviii.

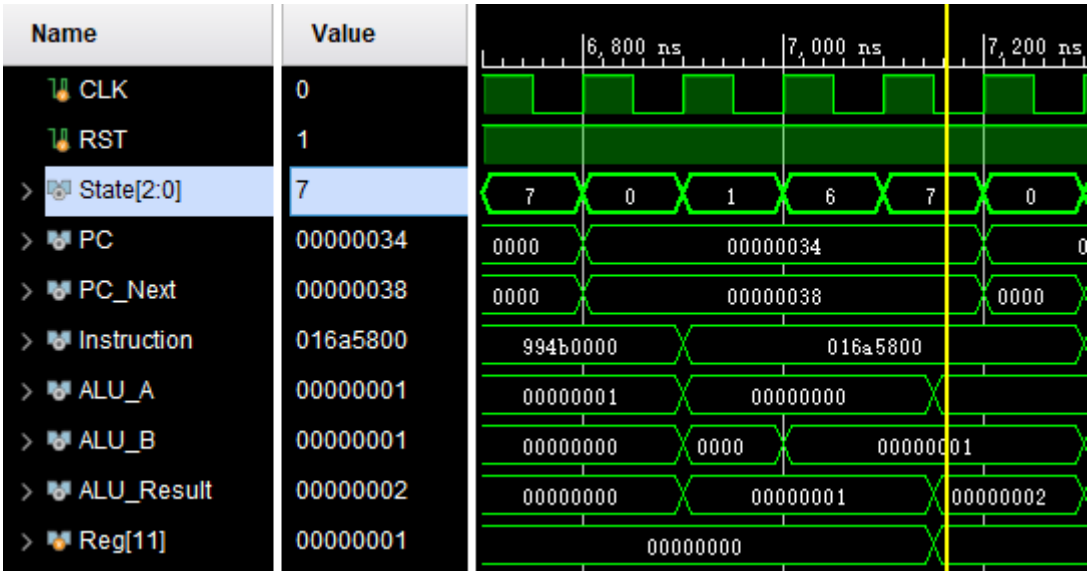
0x00000030	slit $11,$10,0	$11=0	100110	01010	01011	0000 0000 0000 0000	994B0000



该指令表示与立即数的比较运算，ALU_A为寄存器$10的内容（1），ALU_B为立即数0，由于$10>0，故ALU运算结果为0。最后，ALU运算结果会被写入寄存器$11中，其内容为0。

xix.

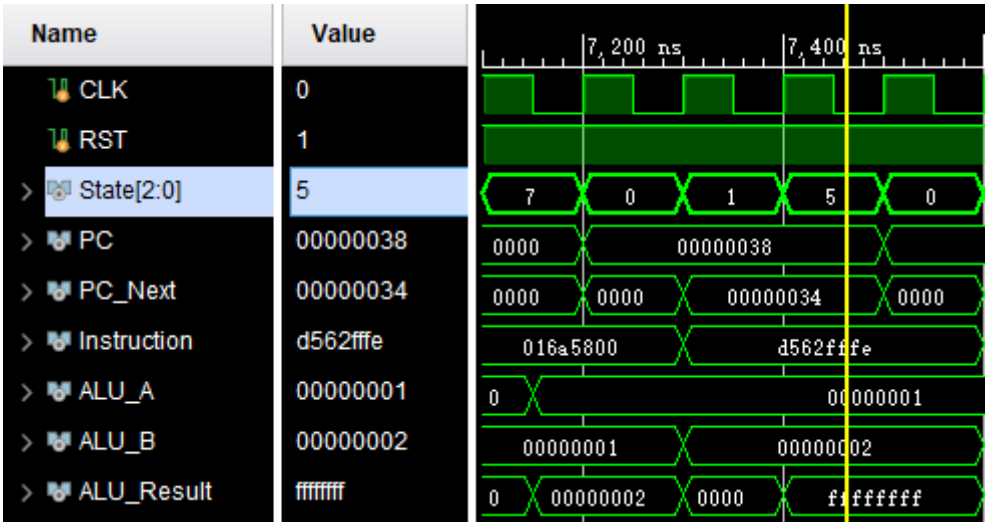
0x00000034	add $11,$11,$10	$11=1	000000	01010	01010	0101 1000 0000 0000	016A5800



该指令表示加法运算，ALU_A为寄存器$11的内容（0），ALU_B为寄存器$10的内容（1），ALU运算结果为1。最后，ALU运算结果会被写入寄存器$11中，其内容为1。

xx.

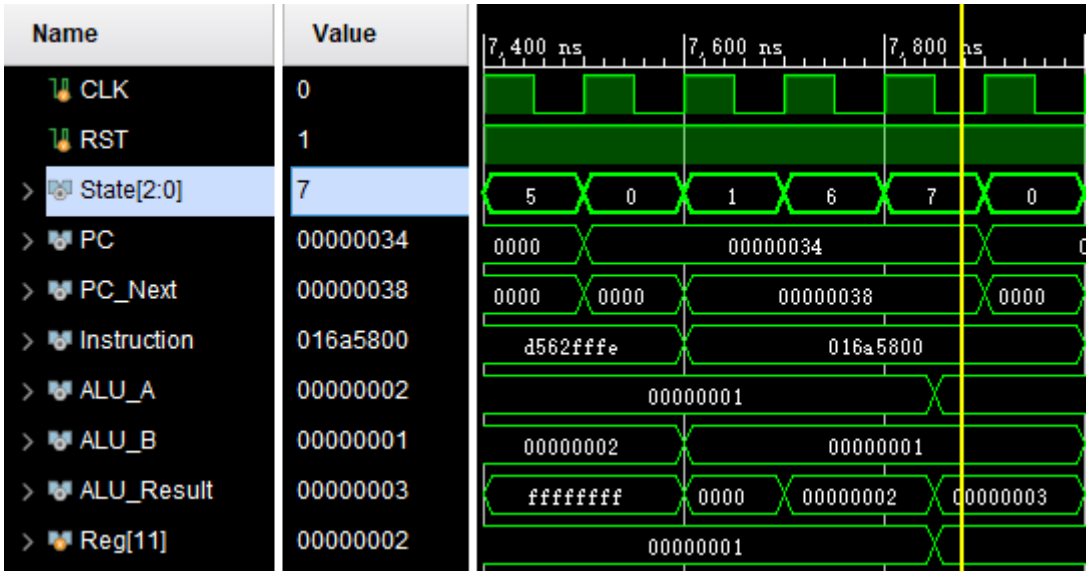
0x00000038	bne $11,$2,-2(≠，转 34)	跳转至 34	110101	01011	00010	1111 1111 1111 1110	D562FFFE



该指令表示条件跳转，ALU_A为寄存器$11的内容（1），ALU_B为寄存器$2的内容（2），ALU运算结果zero不为0，表示$11和$2内容不相等，故下一条指令地址为nextPC=currentPC+immediate<<2，即跳转至0x00000034。

xxi.

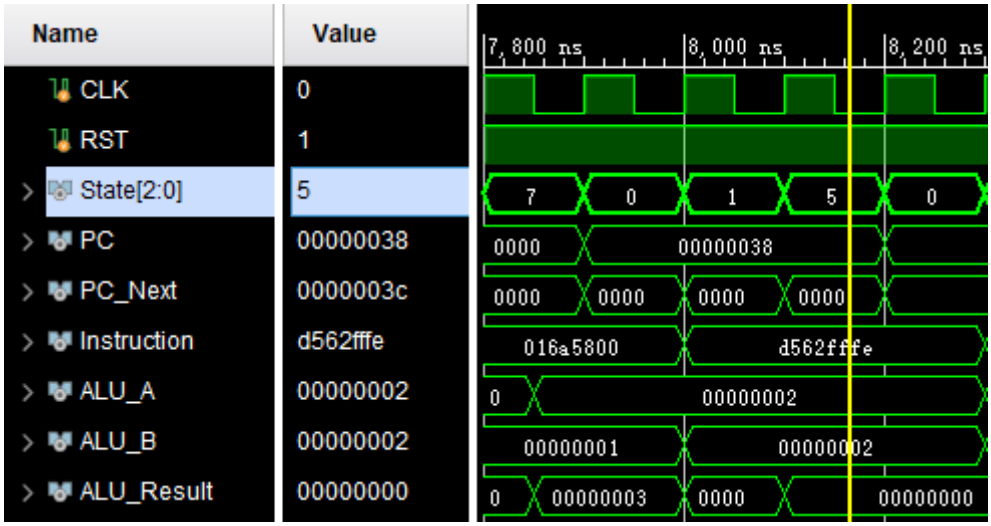
0x00000034	add $11,$11,$10	$11=1	000000	01010	01010	0101 1000 0000 0000	016A5800



该指令表示加法运算，ALU_A为寄存器$11的内容（1），ALU_B为寄存器$10的内容（1），ALU运算结果为2。最后，ALU运算结果会被写入寄存器$11中，其内容为2。

xxii.

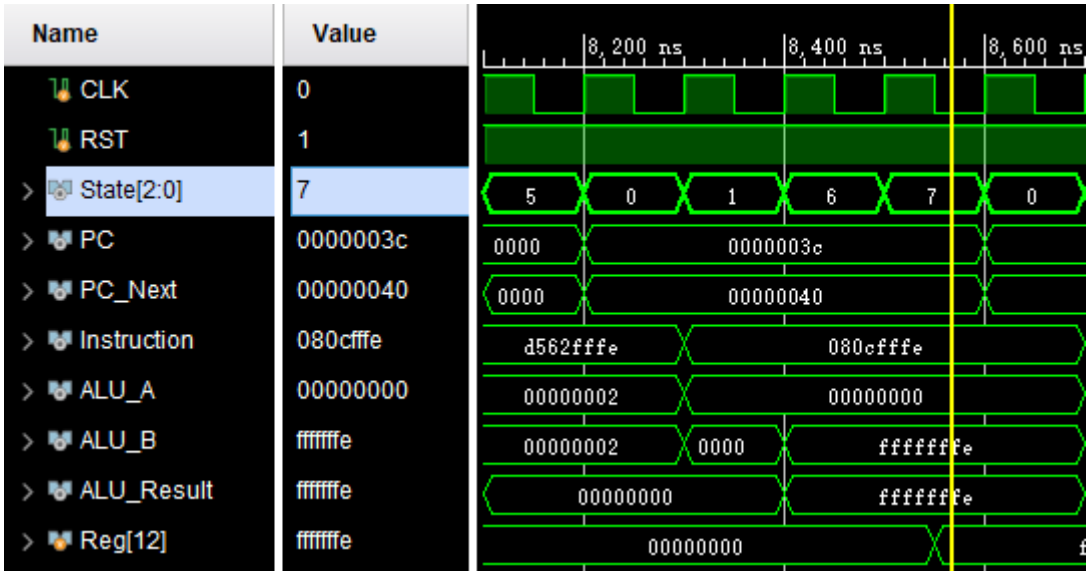
0x00000038	bne $11,$2,-2(≠，转 34)	跳转至 34	110101	01011	00010	1111 1111 1111 1110	D562FFFE



该指令表示条件跳转，ALU_A为寄存器$11的内容（2），ALU_B为寄存器$2的内容（2），ALU运算结果zero为0，表示$11和$2内容相等，故下一条指令地址为nextPC=currentPC+4，即跳转至0x0000003C。

xxiii.

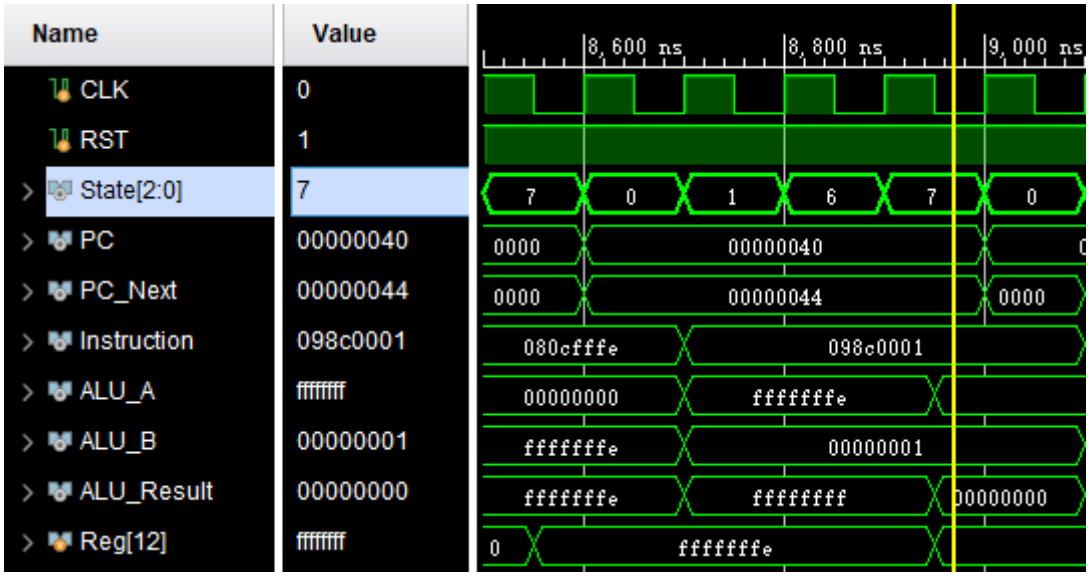
0x0000003C	addiu $12,$0,-2	$12=-2	000010	00000	01100	1111 1111 1111 1110	080CFFFE



该指令表示与无符号立即数的加法运算，ALU_A为寄存器$0的内容（0），ALU_B为立即数-2，ALU运算结果为-2。最后，ALU运算结果会被写入寄存器$12中，其内容为-2。

xxiv.

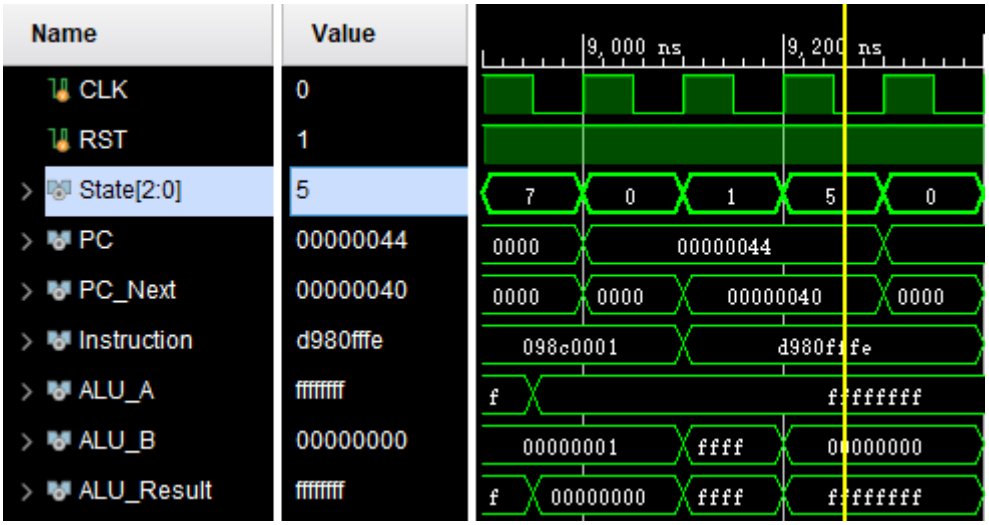
0x00000040	addiu $12,$12,1	$12=-1	000010	01100	01100	0000 0000 0000 0001	098C0001



该指令表示与无符号立即数的加法运算，ALU_A为寄存器$12的内容（-2），ALU_B为立即数1，ALU运算结果为-1。最后，ALU运算结果会被写入寄存器$12中，其内容为-1。

xxv.

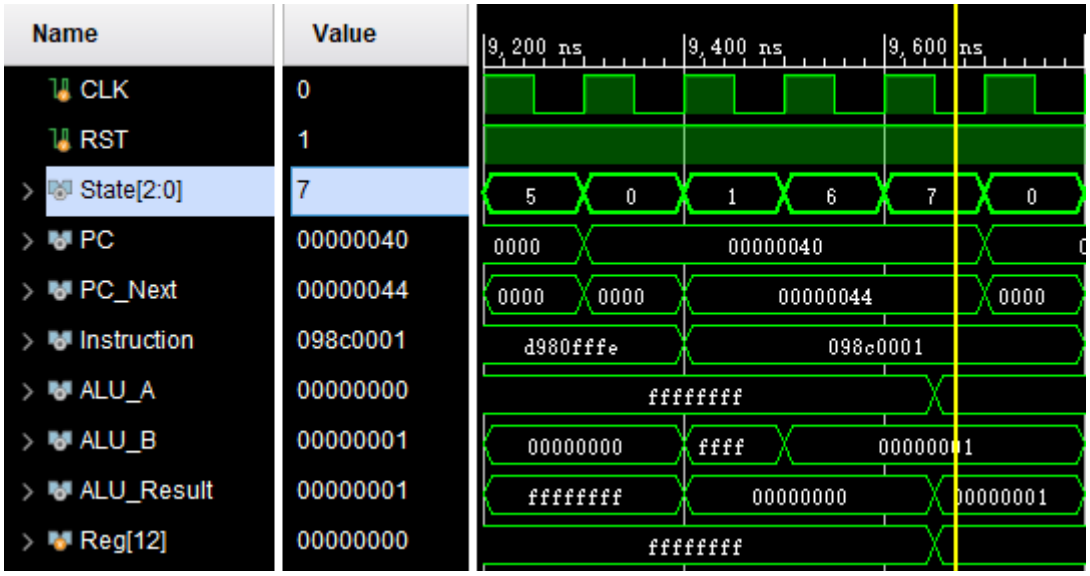
0x00000044	bltz $12,-2(<0，转 40)	跳转至 40	110110	01100	00000	1111 1111 1111 1110	D980FFFE



该指令表示条件跳转，由于寄存器$12的内容（-1）小于0，即$12<0，故下一条指令地址为nextPC=currentPC+immediate<<2，即跳转至0x00000040。

xxvi.

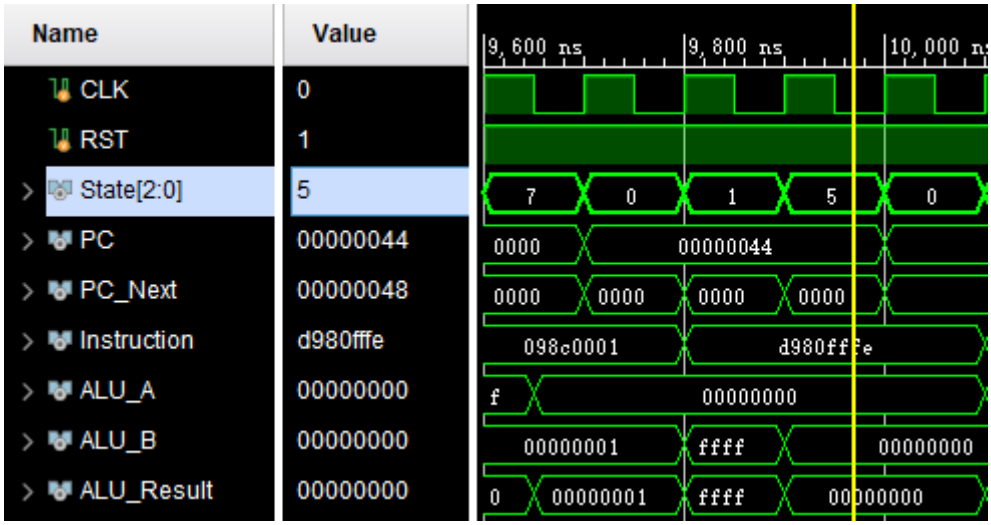
0x00000040	addiu $12,$12,1	$12=0	000010	01100	01100	0000 0000 0000 0001	098C0001



该指令表示与无符号立即数的加法运算，ALU_A为寄存器$12的内容（-1），ALU_B为立即数1，ALU运算结果为0。最后，ALU运算结果会被写入寄存器$12中，其内容为0。

xxvii.

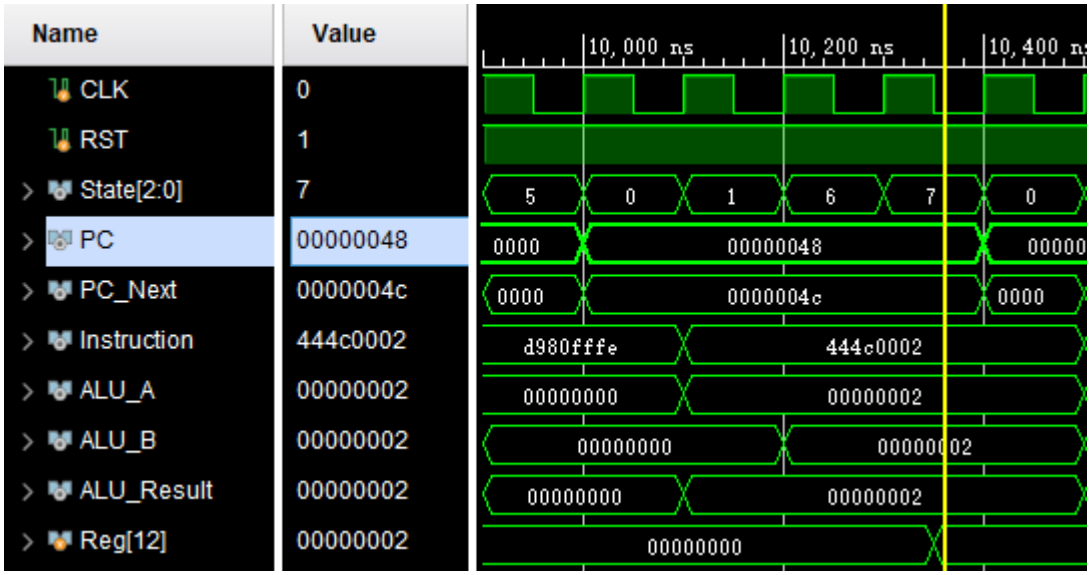
0x00000044	bltz $12,-2(<0，转 40)	执行 48	110110	01100	00000	1111 1111 1111 1110	D980FFFE



该指令表示条件跳转，由于寄存器$12的内容（0）等于0，即$12=0，故下一条指令地址为nextPC=currentPC+4，即跳转至0x00000048。

xxviii.

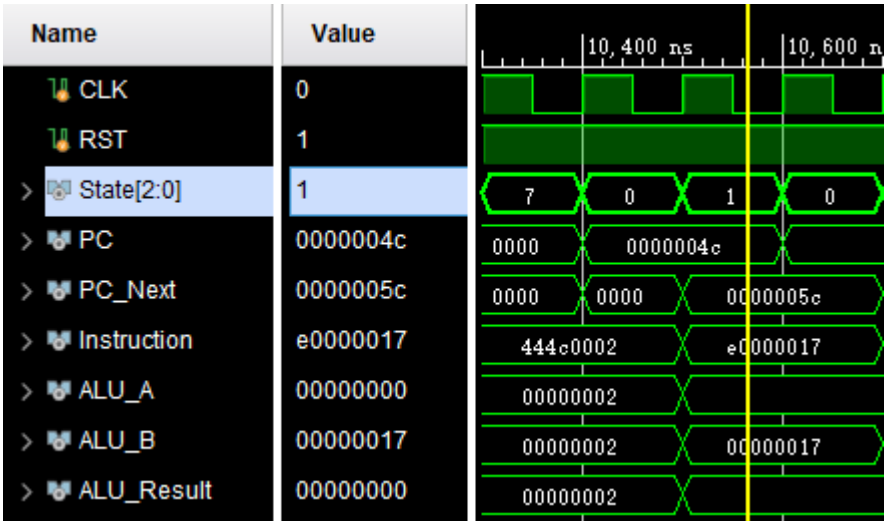
0x00000048	and $12,$2,2	$12=2	010001	00010	01100	0000 0000 0000 0010	444C0002



该指令表示与立即数的与运算，ALU_A为寄存器$2的内容（2），ALU_B为立即数2，ALU运算结果为2。最后，ALU运算结果会被写入寄存器$12中，其内容为2。

xxix.

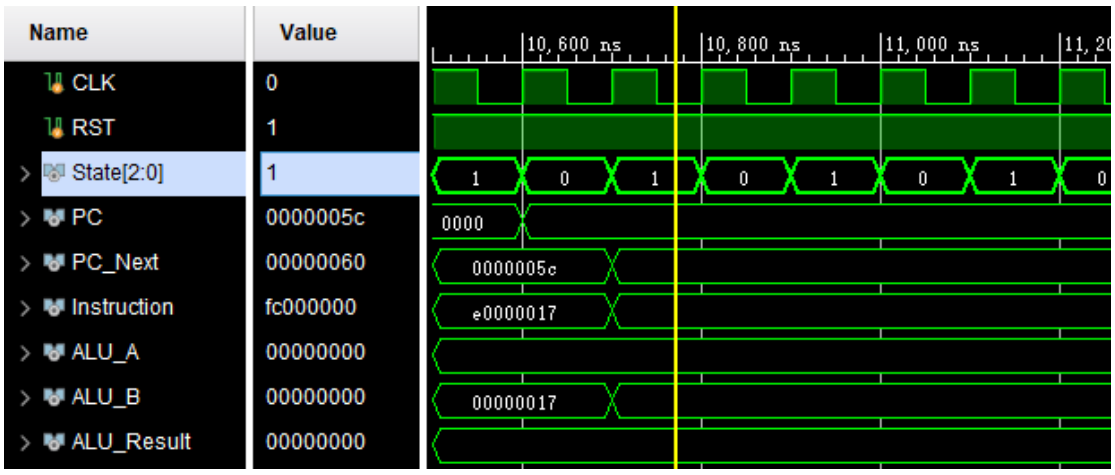
0x0000004C	j 0x0000005C	地址跳转	111000	00000	00000	0000 0000 0101 1100	E000005C



该指令表示无条件跳转，故下一条指令地址为0x0000005C。

xxx.

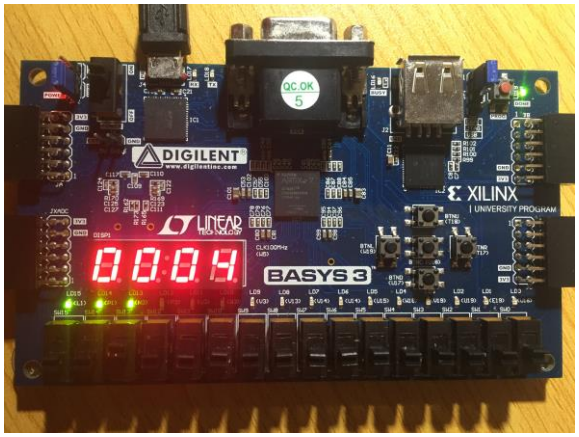
0x0000005C	halt	停机	111111	00000	00000	0000 0000 0000 0000	FC000000



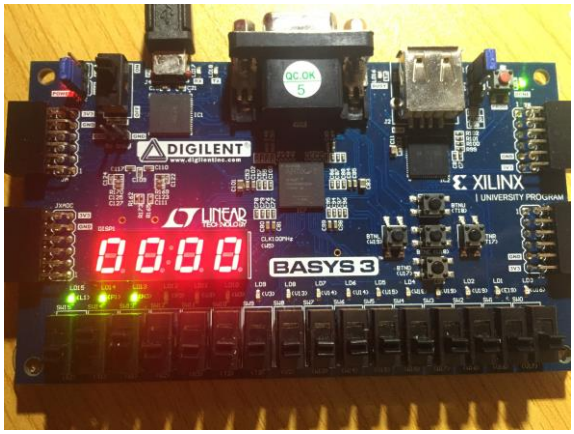
该指令表示停机，执行完毕后，PC停止改变。

六. Basys3 测试结果

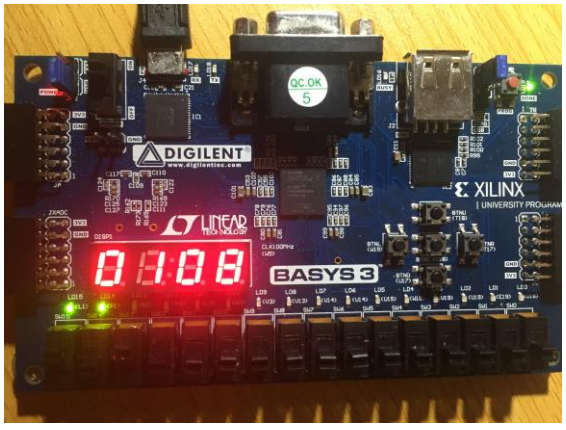
- Switch=00：显示当前PC和下条PC值。左边为当前PC地址00h，右边为下条PC地址04h。



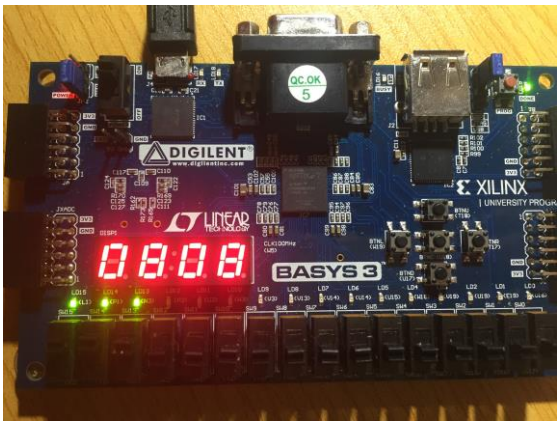
- Switch=01：显示RS寄存器地址和RS寄存器数据。左边为当前指令的RS寄存器编号00，右边为其内容00h。



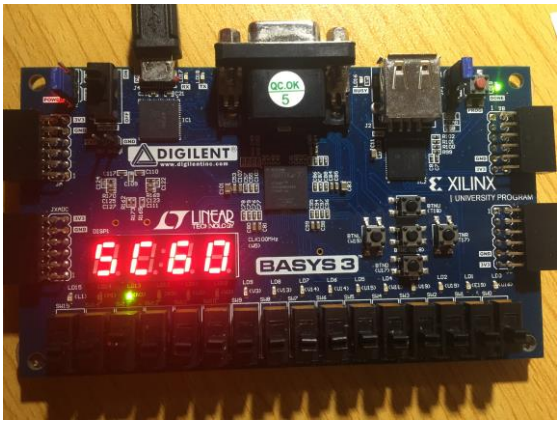
- Switch=10：显示RT寄存器地址和RT寄存器数据。左边为当前指令的RT寄存器编号01，右边为其内容08h。



- **Switch=11：显示ALU输出结果和DB总线数据。左边为ALU运算结果08h，右边为DB总线数据08h。**



- **执行 halt 指令后，PC保持5Ch不变。**



七. 编译器实现

实现内容

在本次实验的目的中， 有一项“编写一个编译器，将MIPS汇编程序编译为二进制机器码”。为完成该实验目的，我使用C++作为开发语言，编写了一个简单的MIPS编译程序。附件中有:

- MIPS-Compiler.cc 为编译器源码文件
- in 为程序的输入文件，格式为一行一条指令，可使用逗号或空格在行内进行分割
- out 为程序的输出文件，每条MIPS指令将生成32位机器代码

附件代码均成功通过测试。

代码实现

具体代码可参见附件，下面列出关键代码：

```
string decode(string str) {
  vector<string> tokens;
  split(str, ", ", tokens);
  string rlt = "";
  A("add", "000000")
  A("sub", "000001")
  A("and", "010000")
  A("slt", "100111")
  B("addiu", "000010", 2, 1)
  B("andi", "010001", 2, 1)
  B("ori", "010010", 2, 1)
  B("xori", "010011", 2, 1)
  B("slti", "100110", 2, 1)
  B("beq", "110100", 1, 2)
  B("bne", "110101", 1, 2)
  C("sw", "110000")
  C("lw", "110001")
  D("j", "111000")
  D("jal", "111010")
  if (tokens[0] == "sll")
    rlt = "011000" + format("0", 5) + format(bin(search(tokens[2])), 5) +
          format(bin(search(tokens[1])), 5) +
          format(bin(search(tokens[3])), 5) + format("0", 6);
  if (tokens[0] == "bltz")
    rlt = "110110" + format(bin(search(tokens[1])), 5) + format("0", 5) +
          format(bin(tokens[2]), 16);
  if (tokens[0] == "jr")
    rlt = "111001" + format(bin(search(tokens[1])), 5) + format("0", 21);
  if (tokens[0] == "halt") rlt = "111111" + format("0", 26);
  return rlt;
}
```

测试示例

in

```
addiu $1, $0, 8
ori $2, $0, 2
xori $3, $2, 8
sub $4, $3, $1
and $5, $4, $2
sll $5, $5, 2
beq $5, $1, -2
jal 0x0000050
slt $8, $13, $1
addiu $14, $0, -2
slt $9, $8, $14
slti $10, $9, 2
slti $11, $10, 0
add $11, $11, $10
bne $11, $2, -2
addiu $12, $0, -2
addiu $12, $12, 1
bltz $12, -2
andi $12, $2, 2
j 0x000005c
sw $2, 4($1)
lw $13, 4($1)
jr $31
halt
```

out

```
00001000000000001000000000001000
01001000000000010000000000000010
01001100010000110000000000001000
00000100011000010010000000000000
01000000100000100010100000000000
01100000000001010010100010000000
11010000101000011111111111111110
11101000000000000000000001010000
10011101101000010100000000000000
00001000000011101111111111111110
10011101000011100100100000000000
10011001001010100000000000000010
10011001010010110000000000000000
00000001011010100101100000000000
11010101011000101111111111111110
00001000000011001111111111111110
00001001100011000000000000000001
11011001100000001111111111111110
01000100010011000000000000000010
11100000000000000000000001011100
11000000001000100000000000000100
11000100001011010000000000000100
11100111111000000000000000000000
11111100000000000000000000000000
```

八. 实验心得

总体来说，本次设计多周期CPU的实验是成功的。由于在之前的设计单周期CPU实验吸取了很多经验教训，我在这次的实验中没有犯低级错误。尽管如此，我还是遇见了一些值得关注的问题：

- 数据通路和控制线路是CPU设计的核心内容，因此从数据通路图入手CPU设计，能够提高我们的开发效率，加深我们对CPU的认识。
- 在读取本地指令文件时，Vivado要求使用绝对路径，且路径中不能有中文名称，相对路径是一定不能使用的。
- 各模块的频率需正确设置，显示选择模块的频率应是防抖按键输出的频率，数码管的频率应该是分频后的频率。数码管的正常显示，需要充分利用余晖效应，因此时钟分频的设置非常关键，应该设置到1000Hz。
- Basys3的数码管控制按键，需要进行防抖处理。
- 烧录实验板时，务必检查其型号，否则会出现端口分配失败的错误。在本次实验中，我差点再次犯下此低级错误。

通过这次多周期CPU设计实验，我基本掌握了多周期CPU的设计和测试方法，对单周期和多周期CPU的区别与联系有了更为深刻的认识。

经过这学期的三次计算机组成原理实验，我对计算机底层知识的理解得到了巨大的扩充。现在回顾这三次实验，虽然自己走了不少弯路，花了很长时间去debug，但这些都是值得的，因为在经历这些困难后，我对计算机组成原理有了更为深刻的认识，能够更为运用Verilog开发硬件，能够更为熟悉地使用 Vivado 工具，更重要的是，我的独立思考和解决问题的能力得到了提升。