《计算机组成原理与接口技术实验》

实验报告

（实验二）

学 院 名 称 ：	数据科学与计算机学院
专业（班级）：	16 软件工程四（7）班
学 生 姓 名 ：	袁之浩
学　　号 ：	16340282
时　　间 ：	2018　年　5　月　20　日

成 绩 ：

实验二 ： 单周期CPU设计与实现

一. 实验目的

(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；
(5) 掌握单周期 CPU 的实现方法。

二. 实验内容

设计一个单周期 CPU，该 CPU 至少能实现以下指令功能操作。指令与格式如下：

==> 算术运算指令

（1）**add rd , rs, rt** （**说明：以助记符表示，是汇编指令；以代码表示，是机器指令**）

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

功能：rd←rs + rt。**reserved 为预留部分，即未用，一般填“0”。**

（2）addi rt , rs ,**immediate**

000001	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt←rs + (sign-extend)**immediate；immediate** 符号扩展再参加“加”运算。

（3）sub rd , rs , rt

000010	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs - rt

==> 逻辑运算指令

（4）ori rt , rs ,**immediate**

010000	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。

（5）and rd , rs , rt

010001	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs & rt；逻辑与运算。

（6）or rd , rs , rt

010010	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs | rt；逻辑或运算。

==>移位指令

（7）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

（8）slti　rt, rs,immediate　　带符号

011011	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

==> 存储器读/写指令

（9）sw　rt ,**immediate**(rs)　写存储器

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

功能：memory[rs+ (sign-extend)**immediate**]←rt；**immediate** 符号扩展再相加。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。

（10) lw　rt , **immediate**(rs)　读存储器

100111	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：rt ← memory[rs + (sign-extend)**immediate**]；**immediate** 符号扩展再相加。

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

（11）beq　rs,rt,**immediate**

110000	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

特别说明：**immediate 是从 PC+4 地址开始和转移到的指令之间指令条数。immediate** 符号扩展之后左移 2 位再相加。为什么要左移 2 位？由于跳转到的指令地址肯定是 4 的倍数（每条指令占 4 个字节），最低两位是“00”，因此将 **immediate** 放进指令码中的时候，是右移了 2 位的，也就是以上说的“指令之间指令条数”。

（12）bne　rs,rt,**immediate**

110001	rs(5 位)	rt(5 位)	**immediate**

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

特别说明：与 beq 不同点是，不等时转移，相等时顺序执行。

==>跳转指令

（13）j　addr

111000	addr[27..2]

功能：pc <－{(pc+4)[31..28],addr[27..2],2{0}}，无条件跳转。

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 位地址了，剩下最高 4 位由 pc+4 最高 4 位拼接上。

==> 停机指令

（14）halt

111111	00000000000000000000000000(26 位)

功能：停机；不改变 PC 的值，PC 保持不变。

三. 实验原理

单周期CPU指的是一条指令的执行在一个时钟周期内完成，然后开始下一条指令的执行，即一条指令用一个时钟周期完成。

单周期 CPU，是在一个时钟周期内完成这五个阶段的处理。

图 1　单周期 CPU 指令处理过程

MIPS 指令的三种格式：

R 类型：

31–26	25–21	20–16	15–11	10–6	5–0
op	rs	rt	rd	sa	funct
6 位	5 位	5 位	5 位	5 位	6 位

I 类型：

31–26	25–21	20–16	15–0
op	rs	rt	immediate
6 位	5 位	5 位	16 位

J 类型：

31–26	25–0
op	address
6 位	26 位

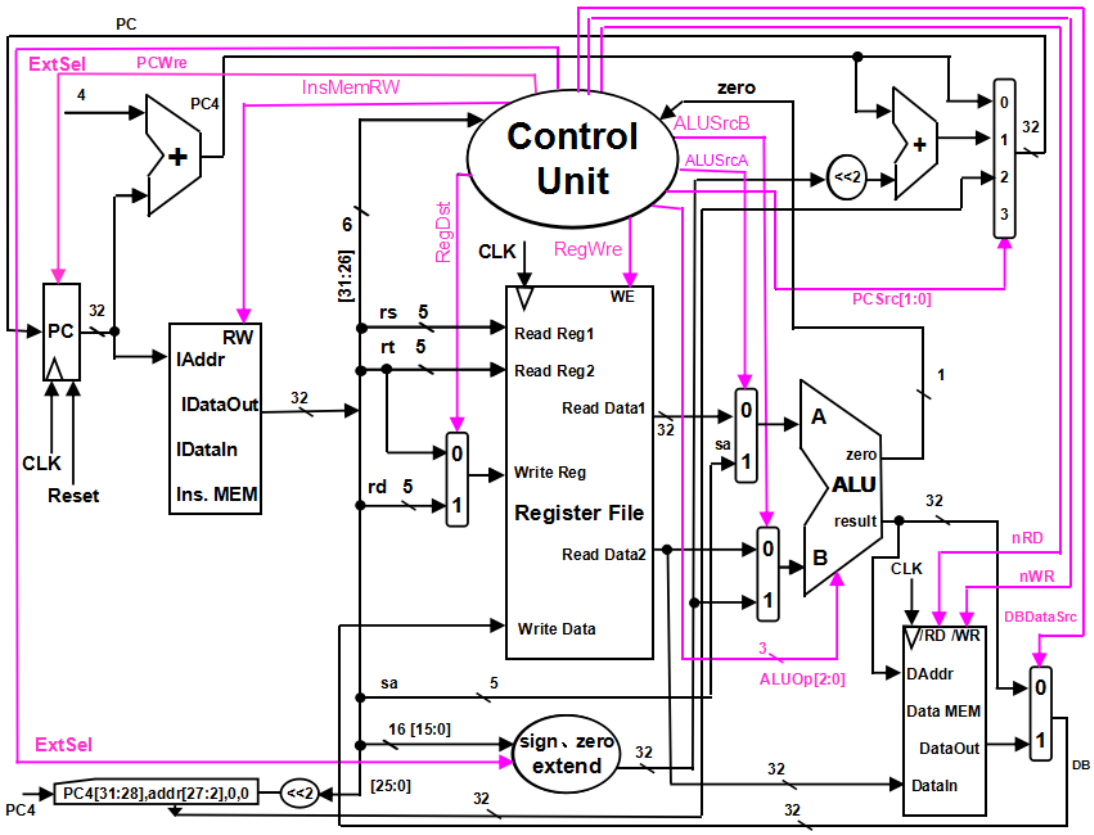


图 2　单周期 CPU 数据通路和控制线路图

表 1 控制信号的作用

控制信号名	状态“0”	状态“1”
Reset	初始化 PC 为 0	PC 接收新地址
PCWre	PC 不更改，相关指令：halt	PC 更改，相关指令：除指令 halt 外
ALUSrcA	来自寄存器堆 data1 输出，相关指令：add、sub、addi、or、and、ori、beq、bne、slti、sw、lw	来自移位数 sa，同时，进行(zero-extend)sa，即 {{27{0}},sa}，相关指令：sll
ALUSrcB	来自寄存器堆 data2 输出，相关指令：add、sub、or、and、sll、beq、bne	来自 sign 或 zero 扩展的立即数，相关指令：addi、ori、slti、sw、lw
DBDataSrc	来自 ALU 运算结果的输出，相关指令：add、addi、sub、ori、or、and、slti、sll	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、sw、halt、j	寄存器组写使能，相关指令：add、addi、sub、ori、or、and、slti、sll、lw
InsMemRW	写指令存储器	读指令存储器(Ins. Data)
nRD	输出高阻态	**读数据存储器，相关指令：lw**
nWR	无操作	**写数据存储器，相关指令：sw**
RegDst	写寄存器组寄存器的地址，来自 rt 字段，相关指令：addi、ori、lw、slti	写寄存器组寄存器的地址，来自 rd 字段，相关指令：add、sub、and、or、sll

ExtSel	(zero-extend)immediate（0 扩展），相关指令：ori	(sign-extend)immediate（符号扩展），相关指令：addi、slti、sw、lw、beq、bne
PCSrc[1..0]	00：pc<－pc+4，相关指令：add、addi、sub、or、ori、and、slti、sll、sw、lw、beq(zero=0)、bne(zero=1); 01：pc<－pc+4+(sign-extend)immediate，相关指令：beq(zero=1)、bne(zero=0); 10：pc<－{(pc+4)[31:28],addr[27:2],2{0}}，相关指令：j; 11：未用	
ALUOp[2..0]	ALU 8 种运算功能选择(000-111)，看功能表	

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

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

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.流程图说明了CPU指令处理过程的步骤

(1) 取指令(IF)：根据程序计数器 PC 中的指令地址，从存储器中取出一条指令，同时，PC 根据指令字长度自动递增产生下一条指令所需要的指令地址，但遇到“地址转移”指令时，则控制器把“转移地址”送入 PC，当然得到的“地址”需要做些变换才送入 PC。

这里需要一个三选一选择器，根据信号 PCSrc[1..0]的值选择哪一个数据进入 PC，三个值分别来自 PC+4，加法器（beq，bne）和立即数拓展（j)。

(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。

PC 数据送入 ROM，得到指令送入 Control Unit 中，产生各种控制信号。

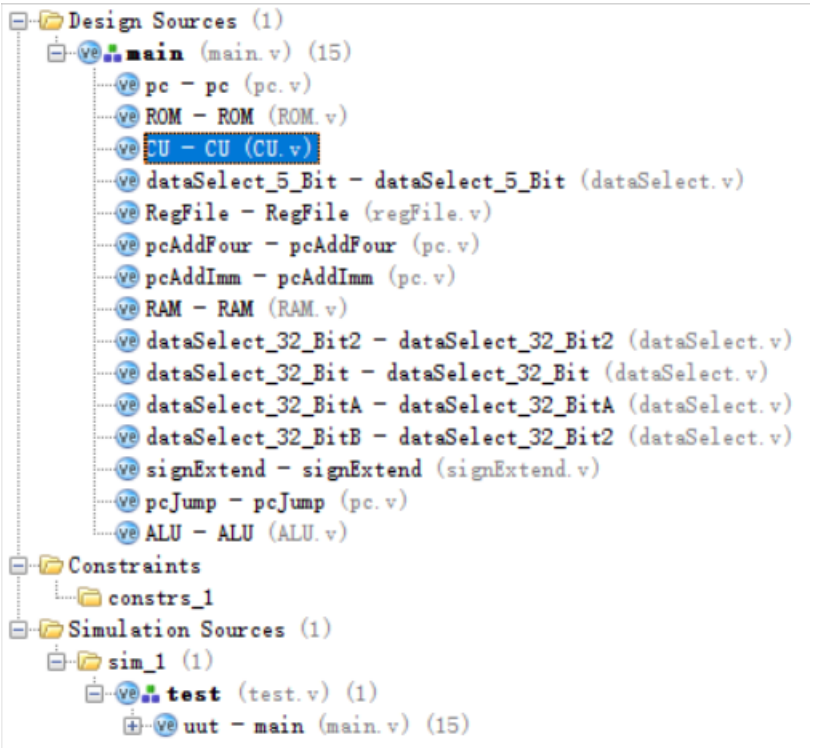
(3) 指令执行(EXE)：根据指令译码得到的操作控制信号，具体地执行指令动作，然后转移到结果写回状态。

包括 RegFile、signExtend 和 ALU 等，根据 CU 的控制信号进行各种计算。

(4) 存储器访问(MEM)：所有需要访问存储器的操作都将在这个步骤中执行，该步骤给出存储器的数据地址，把数据写入到存储器中数据地址所指定的存储单元或者从存储器中得到数据地址单元中的数据。

(5) 结果写回(WB)：指令执行的结果或者访问存储器中得到的数据写回相应的目的寄存器中。

CPU设计思路：先根据数据通路图和后面的表格，设计出每一个模块，再在顶层模块中将它们连起来。



CU：控制单元，根据指令产生控制信号。使用case语句块选择。

```
always@(decode or zero) begin
    case(decode)
     6'b000000:              //add
     begin
        ALUSrcA=0;
        ALUSrcB=0;
        DBDataSrc=0;
        RegWire=1;
        RegDst=1;
        ExtSel=1;
        pcSrc=2'b00;
        ALUOp=3'b000;
     end
```

pc模块生成下一条指令地址，包括PC+4，加上立即数，和直接跳转

```
module pcAddFour(in_pc, out_pc);
    input wire [31:0] in_pc;
    output wire [31:0] out_pc;
    assign out_pc[31:0] = in_pc[31:0] + 4;
endmodule

module pcAddImm(
    input [31:0] now_pc,
    input [31:0] addNum,
    output wire [31:0] out_pc
    );
    assign out_pc = now_pc + (addNum * 4);     //□□□□
endmodule

module pcJump(
    input wire [31:0] pc4,
    input wire [31:0] addr,
    output reg [31:0] out
    );
    always @(pc4 or addr) begin
        out <= { pc4[31:28],addr[27:2],{2{1'b0}} };
    end
```

ROM模块：将txt文件中的指令加载到内存中。

```
module ROM(
    input rd,                    // □1□□
    input [31:0] addr,
    output reg [31:0] dataOut
    );

     reg [7:0] mem [0:99];    // mem □□□□100□□□□□□□□□□□8□□□□□□□□

     initial
         $readmemb ("D:/vivado/cpu_design/rom_data.txt", mem);

     always @( addr or rd)
         if (rd)begin
             dataOut[7:0] = mem[addr + 3];
             dataOut[15:8] = mem[addr + 2];
             dataOut[23:16] = mem[addr + 1];
             dataOut[31:24] = mem[addr];
         end
endmodule
```

数据选择模块：根据Ctrl信号选择输出，包括二选一，三选一，5位的和32位的。

```
module dataSelect_5_Bit(
    input [4:0] A,
    input [4:0] B,
    input Ctrl,
    output [4:0] S
    );
    assign S = ( Ctrl == 1'b0 ? A : B);
endmodule

module dataSelect_32_Bit(
    input [31:0] A,
    input [31:0] B,
    input [31:0] C,
    input [1:0] Ctrl,
    output reg [31:0] S
    );
    always @(Ctrl or A or B or C) begin
        case(Ctrl)
         2'b00: S=A;
         2'b01: S=B;
         2'b10: S=C;
```

RegFile模块：保存了一组32位的寄存器，根据控制信号和输入地址进行存取。

```
module RegFile(
      input CLK,
      input RST,
      input RegWre,
      input [4:0] ReadReg1,
      input [4:0] ReadReg2,
      input [4:0] WriteReg,
      input [31:0] WriteData,
      output [31:0] ReadData1,
      output [31:0] ReadData2
      );
      reg [31:0] regFile[1:31]; //  □□□□□□□□□reg□□
      integer i;

      assign  ReadData1 = (ReadReg1 == 0) ? 0 : regFile[ReadReg1];   // □□□□□□
      assign  ReadData2 = (ReadReg2 == 0) ? 0 : regFile[ReadReg2];

      always @ (negedge CLK or negedge RST) begin   // □□□□□□□□□□
          if (RST==0) begin
               for(i=1;i<32;i=i+1)
               regFile[i] <= 0;
          end
          else if(RegWre == 1 && WriteReg != 0)      // WriteReg != 0□$0□□□□□□□□
               regFile[WriteReg] <= WriteData;            // □□□□
      end
endmodule
```

RAM模块：作为存储器使用，以八位为单位申请了一个数组，根据写入信号在时钟下降沿将数据写入数组对应地址。

```
module RAM(
     input clk,
     input [31:0] address,
     input [31:0] writeData, // [31:24], [23:16], [15:8], [7:0]
     input nRD,              // 0000000001,00000
     input nWR,              // 0000001 0000
     output [31:0] Dataout
     );


     reg [7:0] RAM [0:60]; //

    // 
     assign Dataout[7:0]   = (nRD==1)?RAM[address + 3]:8'bz; // z 
     assign Dataout[15:8]  = (nRD==1)?RAM[address + 2]:8'bz;
     assign Dataout[23:16] = (nRD==1)?RAM[address + 1]:8'bz;
     assign Dataout[31:24] = (nRD==1)?RAM[address ]:8'bz;

    // 
     always@( negedge clk ) begin
        if( nWR ) begin
            RAM[address]   <= writeData[31:24];
            RAM[address+1] <= writeData[23:16];
            RAM[address+2] <= writeData[15:8];
            RAM[address+3] <= writeData[7:0];
        end
     end
endmodule
```

signExtend模块：符号拓展，零拓展在前面补零，符号拓展在前面补最高位的值

```
module signExtend(
    input wire [15:0] in_num,
    input wire extSel,
    output reg [31:0] out_num
    );
    initial begin
        out_num=0;
    end
    always @(in_num or extSel) begin
        if (extSel) begin
            out_num <= { {16{in_num[15]}}, in_num[15:0] };
        end
        else begin
            out_num <= { {16{1'b0}}, in_num[15:0] };
        end
    end

endmodule
```

ALU模块：算逻运算模块，根据之前的功能表执行相应运算。result为运算结果，zero作为标志位，用于跳转指令的比较操作。

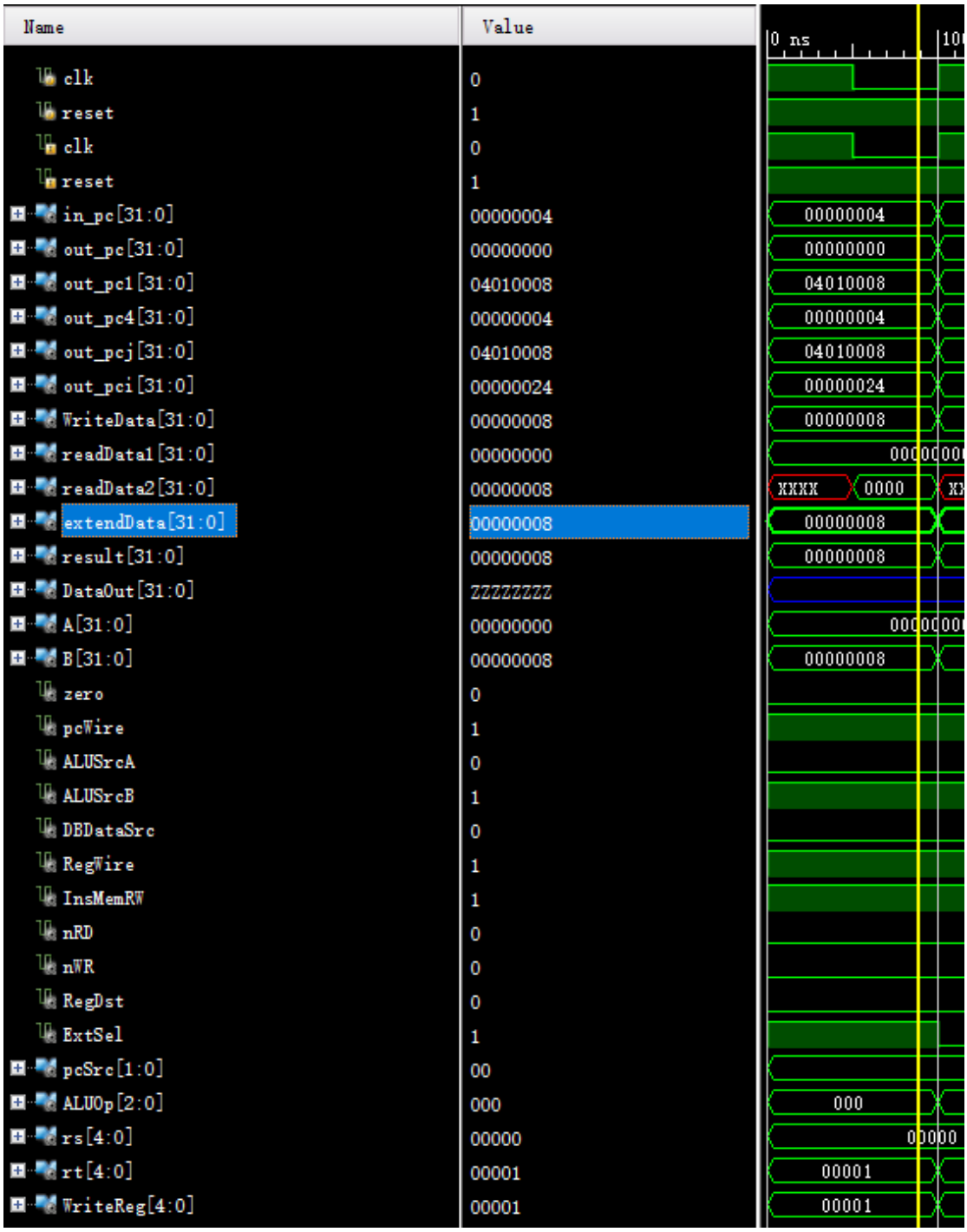
```
module ALU(
    input [2:0] ALUopcode,
    input [31:0] rega,
    input [31:0] regb,
    output reg [31:0] result,
    output zero
    );

    assign zero = (result==0)?1:0;
    always @( ALUopcode or rega or regb ) begin
        case (ALUopcode)
          3'b000 : result = rega + regb;
          3'b001 : result = rega - regb;
          3'b010 : result = regb << rega;
          3'b011 : result = rega | regb;
          3'b100 : result = rega & regb;
          3'b101 : result = (rega > regb)?1:0;
          3'b110 : begin                          // □□□□□
                     if ((rega<regb) &&( rega[31] == regb[31]))
                         result = 1;
                     else if ( rega[31]  && !regb[31])  result = 1;
                     else result = 0;
                   end
         default : begin
                   result = 32'h00000000;
                   $display (" no match");
                end
         endcase
     end
endmodule
```

最后根据数据通路图在main模块里将它们用线连起来。

```
module main(
    input wire clk,
    input wire reset
    );

    wire [31:0] in_pc,out_pc,out_pc1,out_pc4,out_pcj,out_pci,WriteData,readData1,readData2,extendData,result,DataOut,A,B;
    wire zero,pcWire,ALUSrcA,ALUSrcB,DBDataSrc,RegWire,InsMemRW,nRD,nWR,RegDst,ExtSel;
    wire [1:0] pcSrc;
    wire [2:0] ALUOp;
    wire [4:0] rs,rt,WriteReg;

    assign rs=out_pc1[25:21];
    assign rt=out_pc1[20:16];

    pc pc(clk, reset, in_pc, out_pc, pcWire);
    ROM ROM(InsMemRW,out_pc,out_pc1);
    CU CU(out_pc1[31:26],zero,pcWire,ALUSrcA,ALUSrcB,DBDataSrc,RegWire,InsMemRW,nRD,nWR,RegDst,ExtSel,pcSrc,ALUOp);
    dataSelect_5_Bit dataSelect_5_Bit(rt,out_pc1[15:11],RegDst,WriteReg);
    RegFile RegFile(clk,reset,RegWire,rs,rt,WriteReg,WriteData,readData1,readData2);
    pcAddFour pcAddFour(out_pc, out_pc4);
    pcAddImm pcAddImm(out_pc4,extendData,out_pci);
    RAM RAM(clk,result,readData2,nRD,nWR,DataOut);
    dataSelect_32_Bit2 dataSelect_32_Bit2(result,DataOut,DBDataSrc,WriteData);
    dataSelect_32_Bit dataSelect_32_Bit(out_pc4,out_pci,out_pcj,pcSrc,in_pc);

    dataSelect_32_BitA dataSelect_32_BitA(readData1,out_pc1[10:6],ALUSrcA,A);
    dataSelect_32_Bit2 dataSelect_32_BitB(readData2,extendData,ALUSrcB,B);
    signExtend signExtend(out_pc1[15:0],ExtSel,extendData);
    pcJump pcJump(out_pc4,out_pc1,out_pcj);
    ALU ALU(ALUOp,A,B,result,zero);
endmodule
```

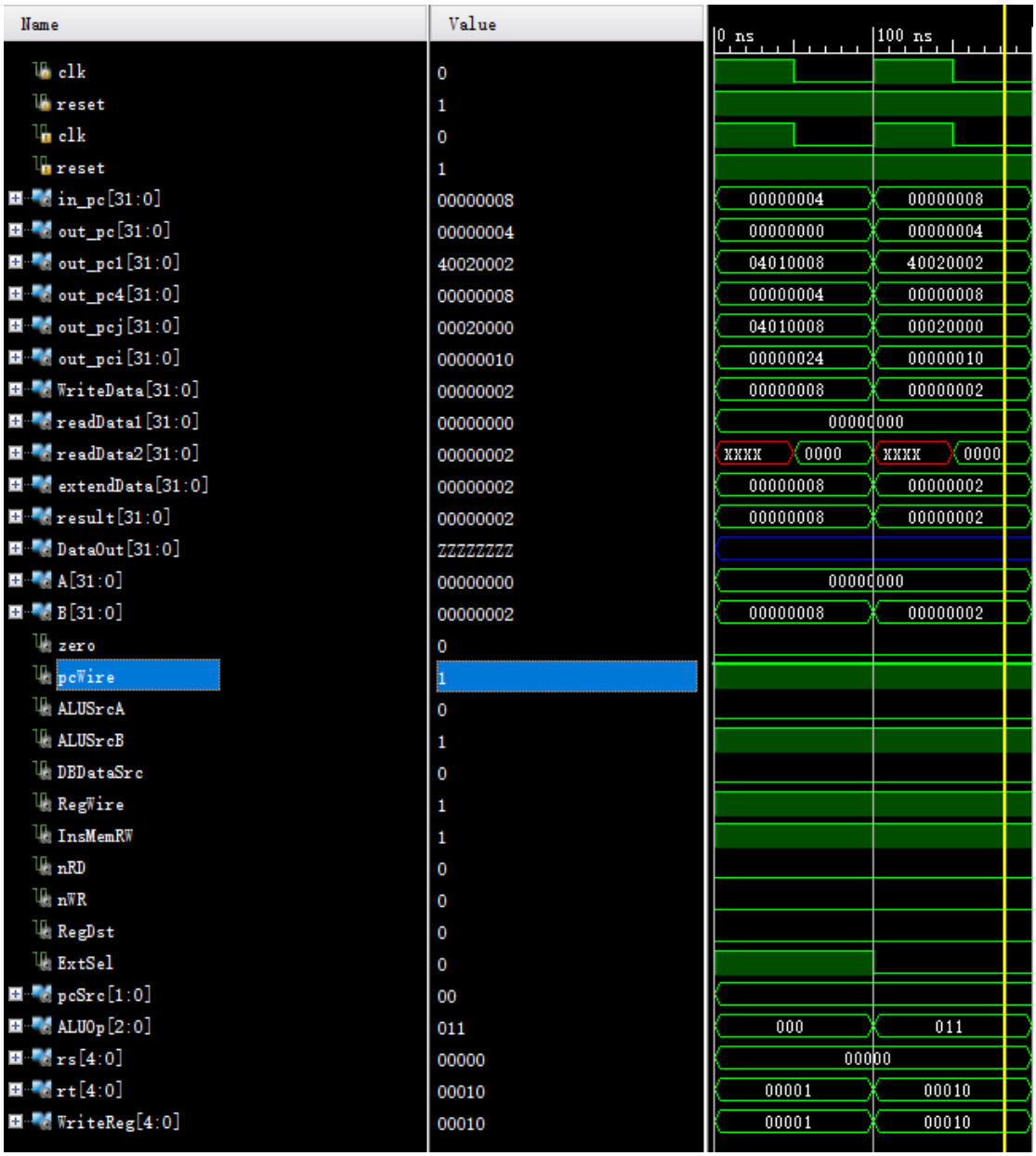
2.验证CPU的正确性，每条指令波形和说明。

0x00000000	addi　$1,$0,8	**000001**	**00000**	**00001**	**0000 0000 0000 1000**	=	04010008



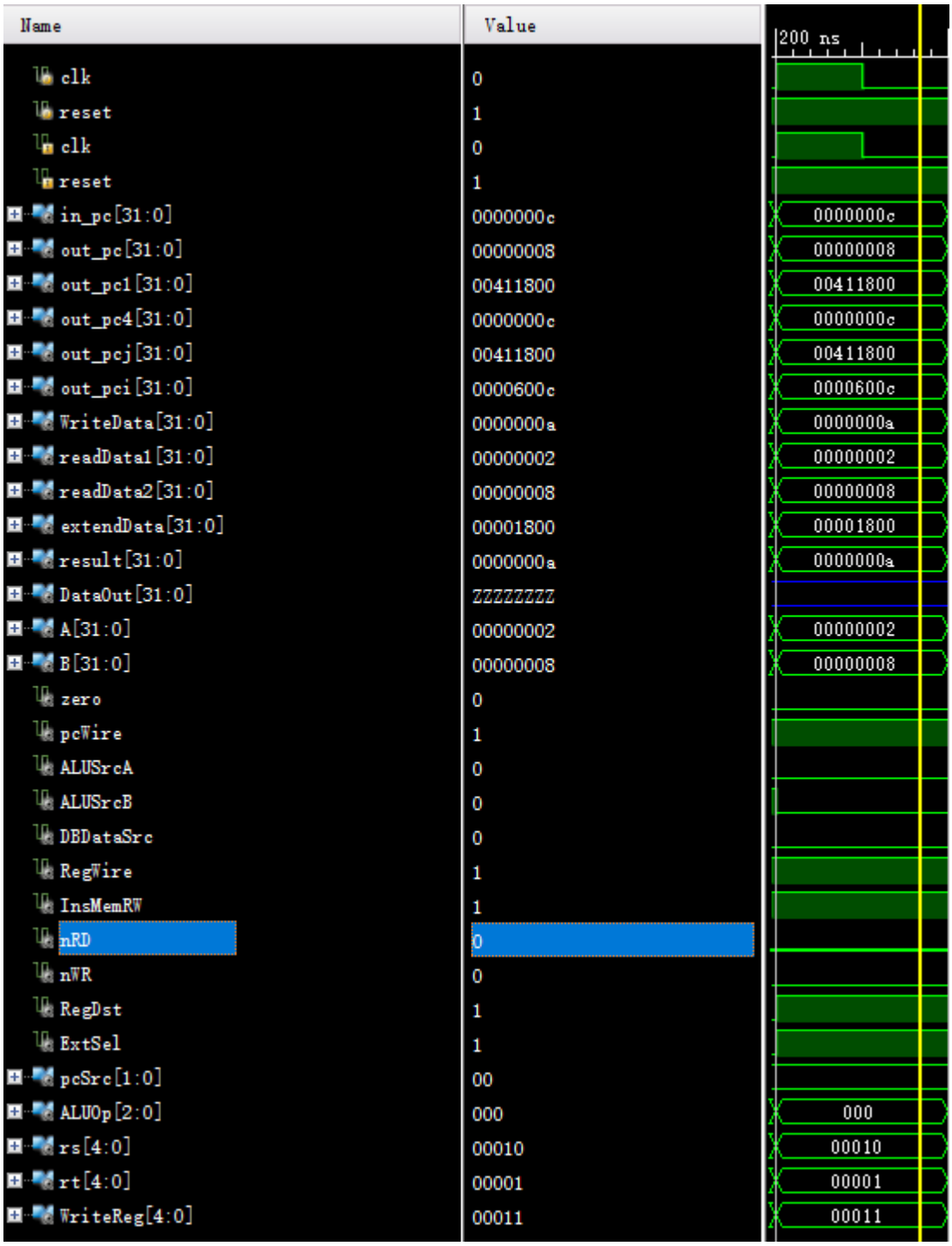
out_pc是当前指令的地址00000000，in_pc是下一条指令的地址00000004，out_pc1是当前指令的内容04010008，out_pc4是当前指令地址加4后的地址,rs为00000，即$0,rt为00001，即$1,extendData为符号拓展后的立即数00000008，A为ALU的输入，即$0中的值0000000,B为ALU的另一个输入，即立即数00000008，操作数ALUSrcA值为0，选择来自寄存器堆输出readData1，操作数ALUSrcB值为1，选择立即数，result为计算结果8，WriteReg为写入寄存器的地址01，即$1.

0x00000004	ori $2,$0,2	010000	00000	00010	0000 0000 0000 0010	=	40020002



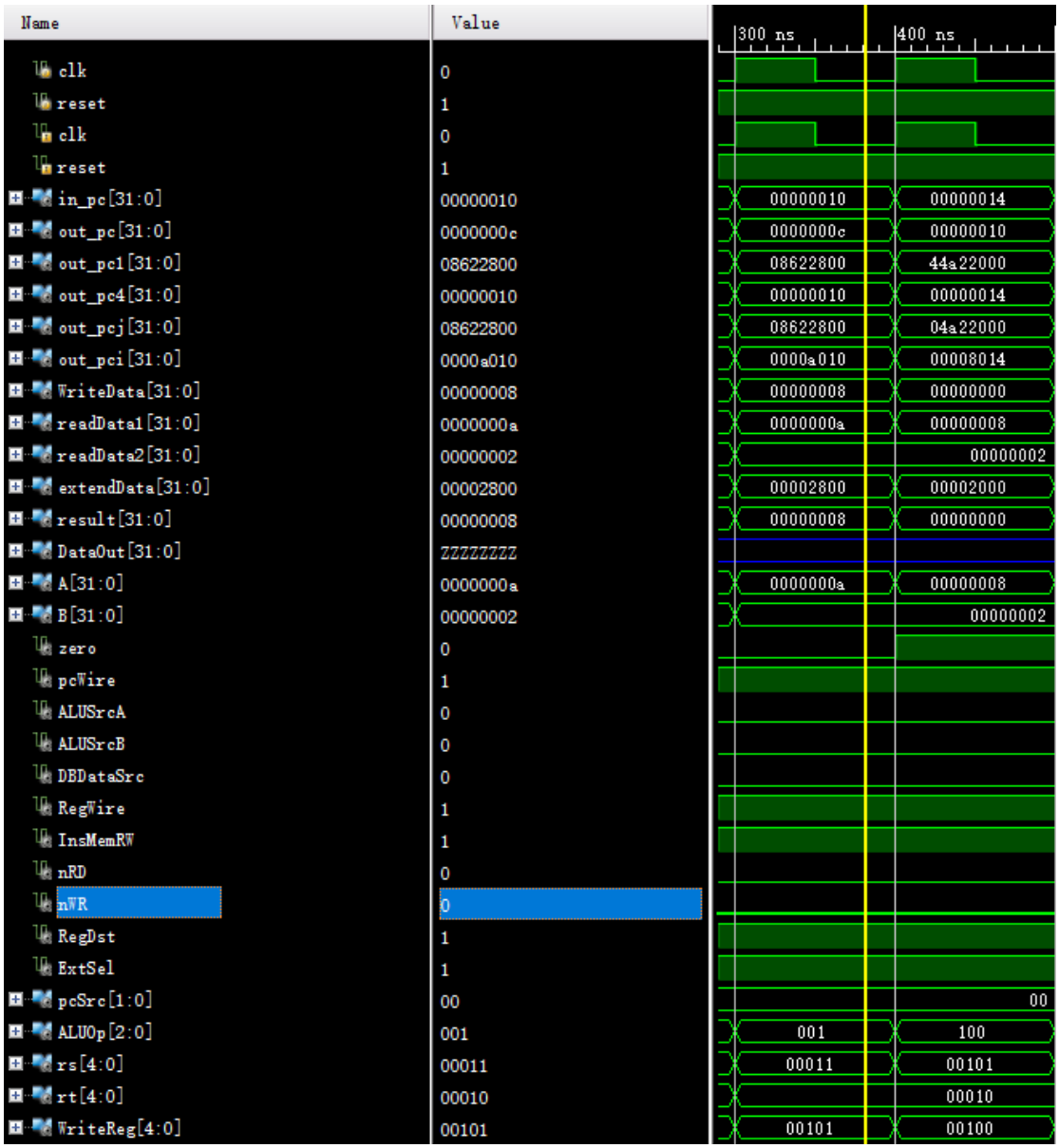
out_pc为当前指令地址00000004，out_pc1为当前指令40020002，rs的值为00000，代表$0的地址，输出到A，extendData为0拓展后的立即数00000002，输出到B，result为计算结果00000002,WriteReg为写入寄存器地址00010,即$2

0x00000008	add $3,$2,$1	000000	00010	00001	00011		00411800



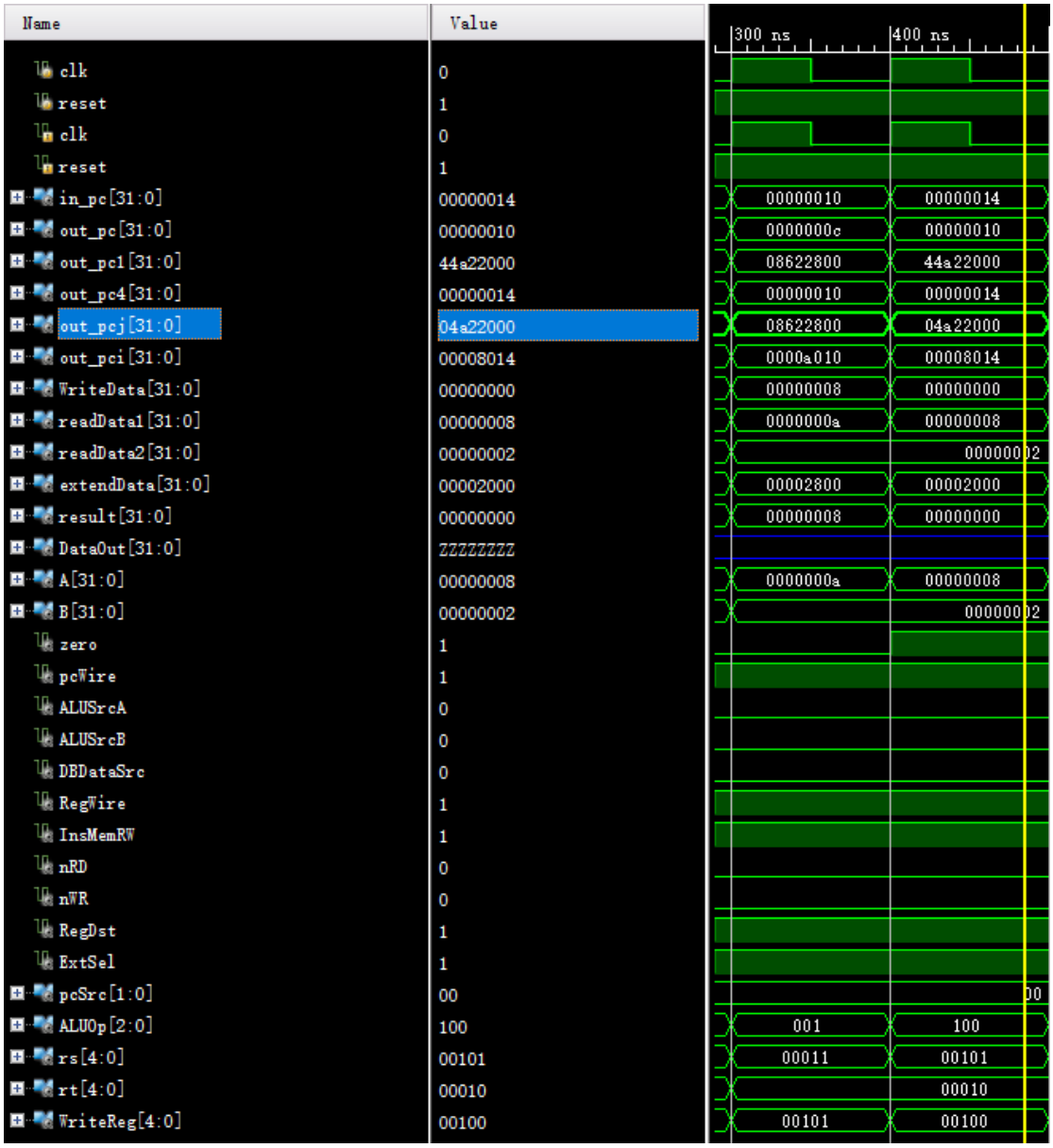
out_pc为当前指令地址00000008，out_pc1为指令00411800，操作数地址rs为00010，即$2,rt为00001，即$1,写入寄存器地址writeReg为00011,即$3,A,B分别为寄存器的值2和8，送入ALU，ALU指令ALUOp为000，执行加操作，计算结果result为10

0x0000000C	sub $5,$3,$2	000010	00011	00010	00101		08622800



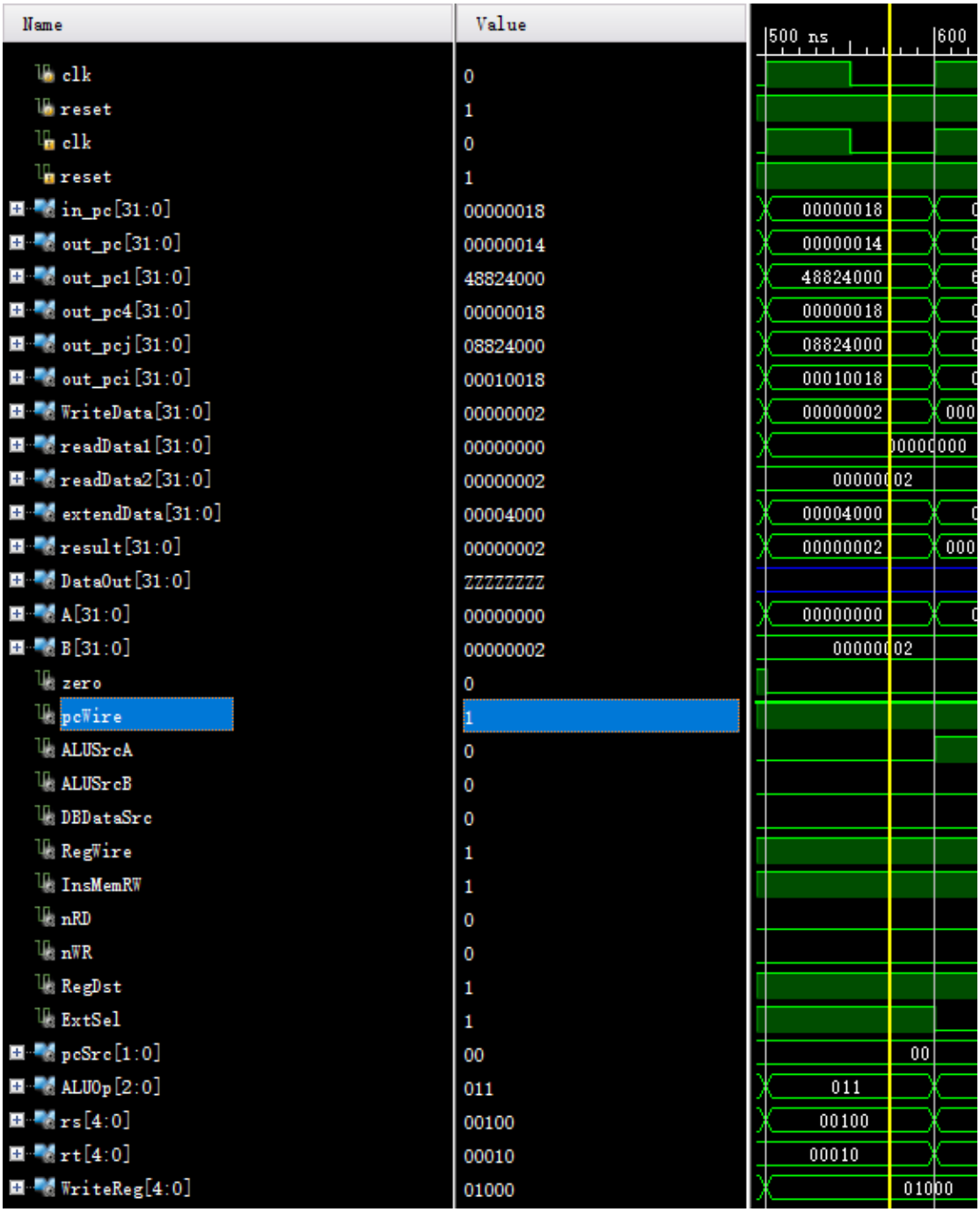
out_pc为当前指令地址0000000C，out_pc1为指令08622800，操作数地址rs为00011，即$3,rt为00010，即$2,写入寄存器地址writeReg为00101,即$5,A,B分别为寄存器的值10和2，送入ALU，ALU指令ALUOp为001，执行减操作，计算结果result为8

0x00000010	and $4,$5,$2	010001	00101	00010	00100		44A22000



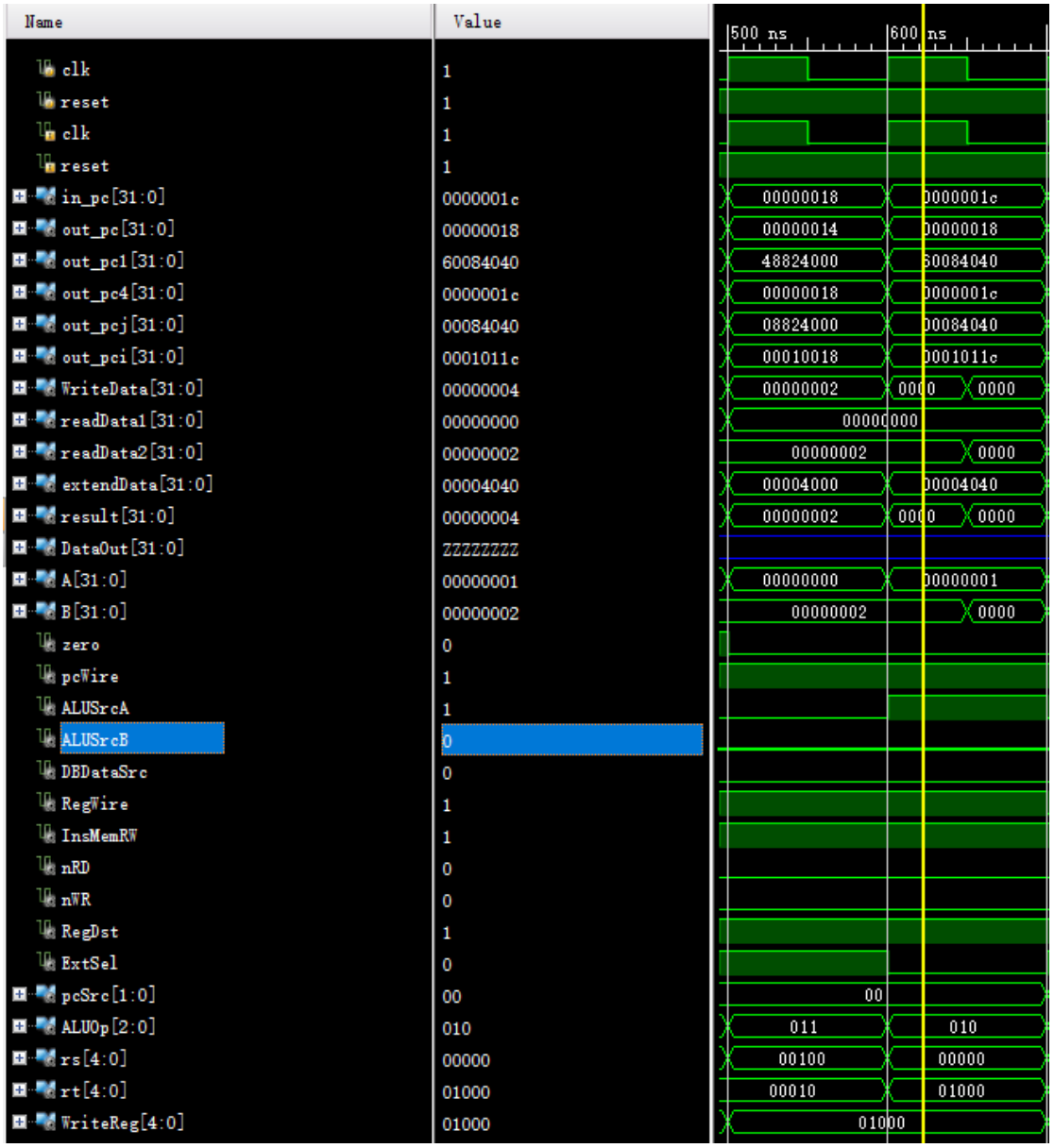
out_pc为当前指令地址00000010，out_pc1为指令44A24000，操作数地址rs为00101，即$5,rt为00010，即$2,写入寄存器地址writeReg为00100,即$4,A,B分别为寄存器的值8和2，送入ALU，ALU指令ALUOp为100，执行与操作，计算结果result为0

0x00000014	or $8,$4,$2	010010	00100	00010	01000		48824000



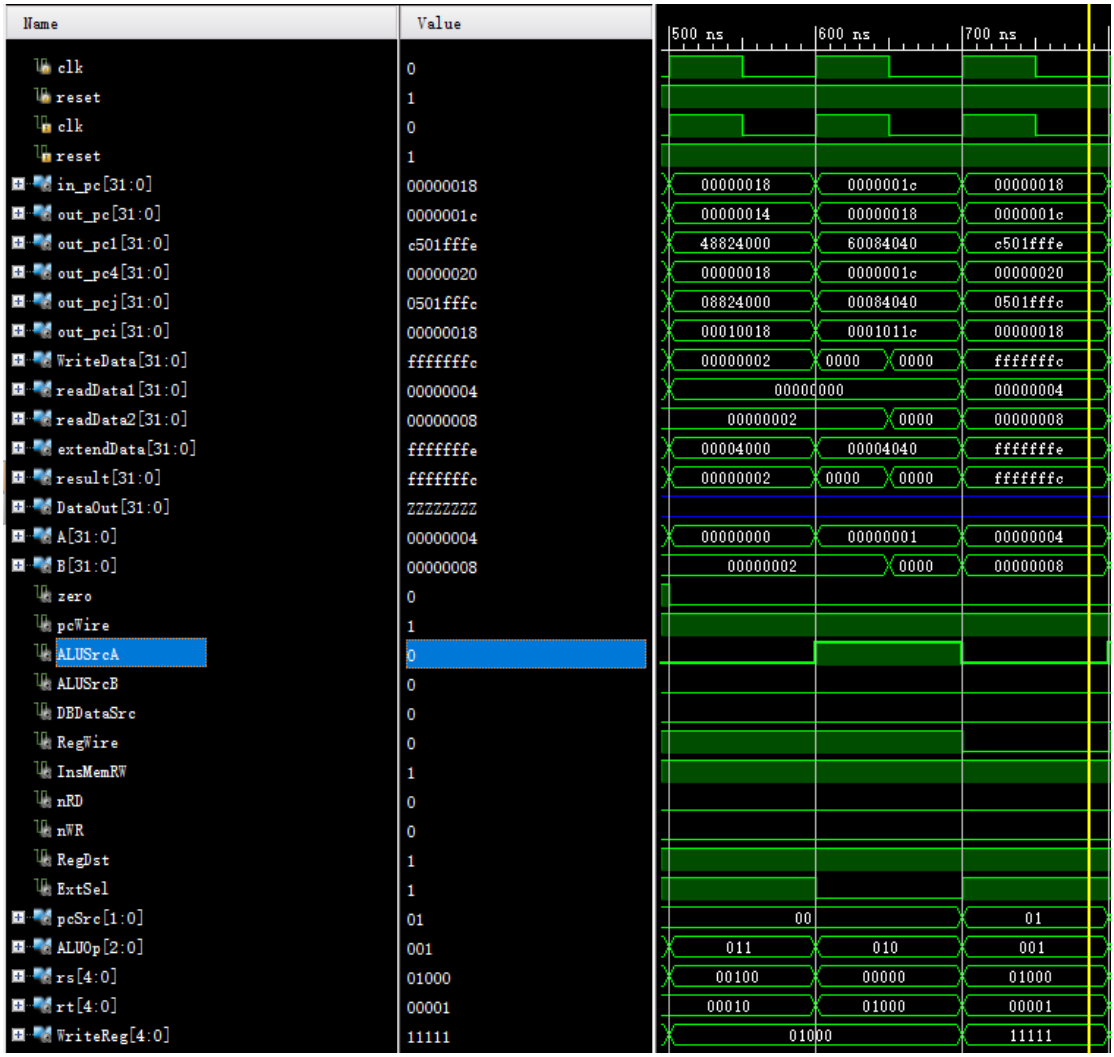
out_pc为当前指令地址00000014，out_pc1为指令48824000，操作数地址rs为00100，即$4,rt为00010，即$2,写入寄存器地址writeReg为01000,即$8,A,B分别为寄存器的值8和0，送入ALU，ALU指令ALUOp为011，执行或操作，计算结果result为2

0x00000018	sll $8,$8,1	011000		01000	01000 00001		60084040



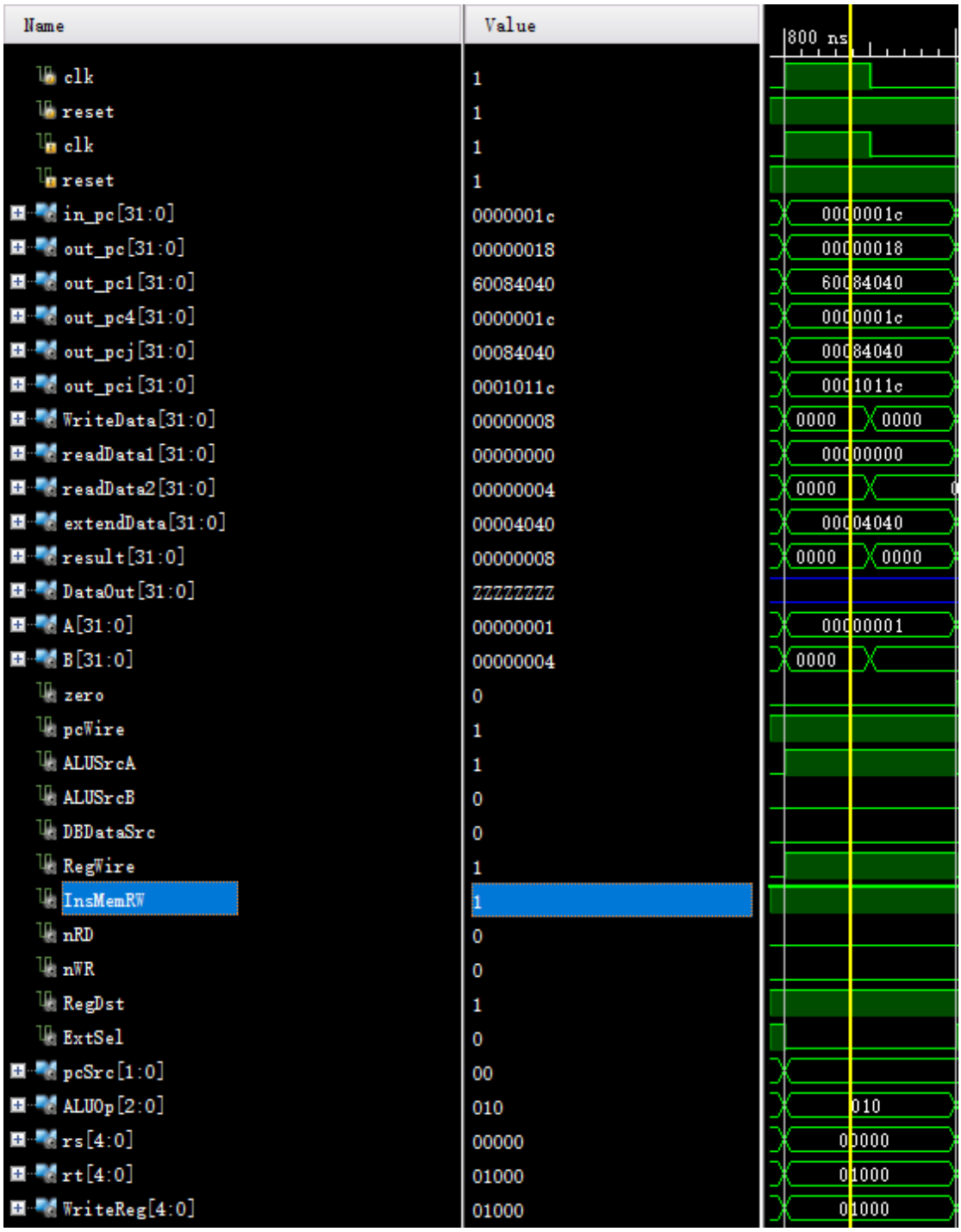
out_pc为当前指令地址00000018，out_pc1为指令60084040，操作数地址rt为01000，即$8，写入寄存器地址writeReg为01000,即$8,A为拓展后的立即数00000001，B为$8中的值00000002，送入ALU，ALU指令ALUOp为010，执行左移操作，计算结果result为00000004

0x0000001C	bne $8,$1,-2 (≠,转 18)	110001	01000	00001	1111 1111 1111 1110		C501FFFE



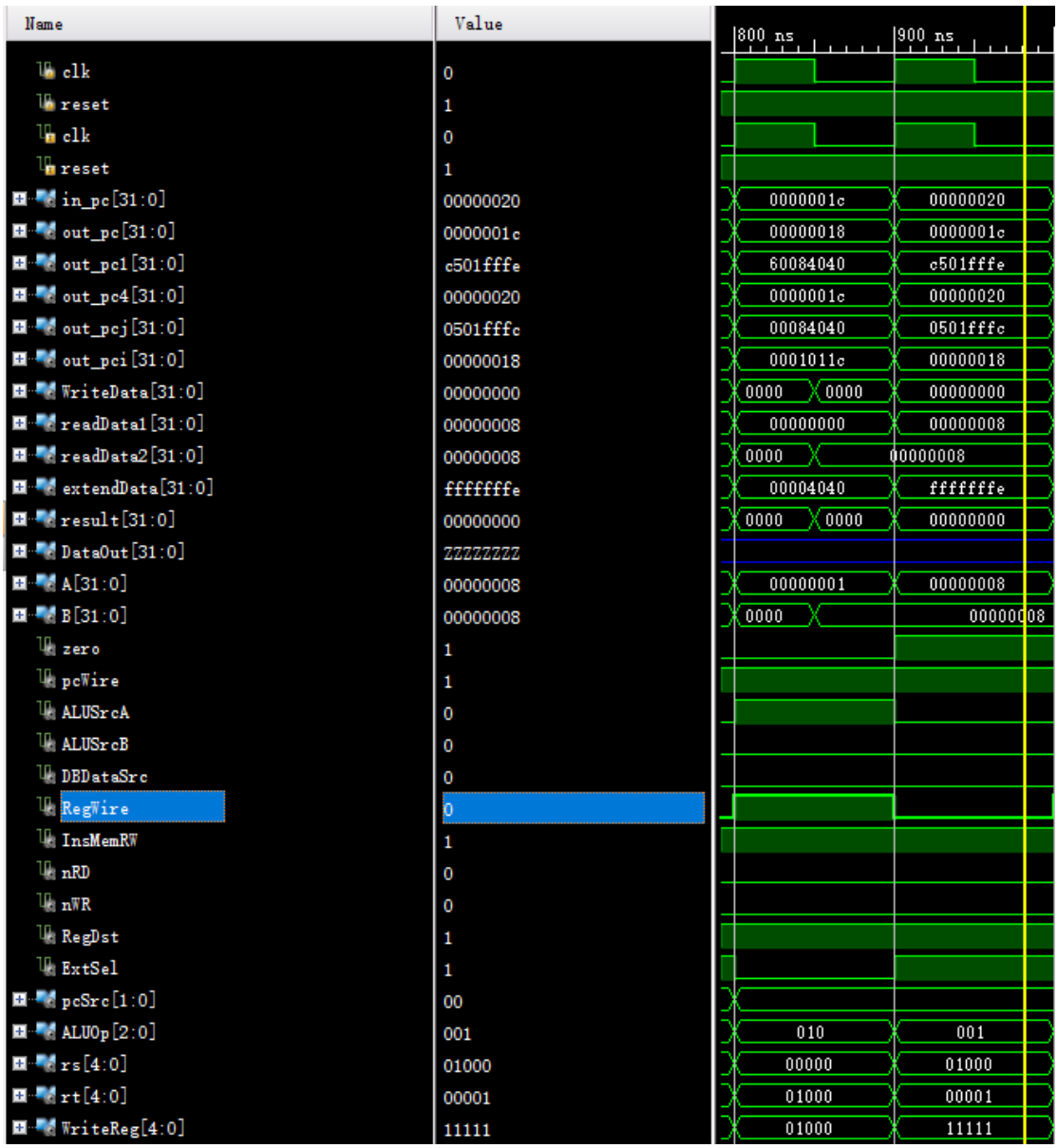
out_pc为当前指令地址0000001c，out_pc1为指令c501fffe，操作数地址rs为01000，即$8，rt为00001，即$1,A为$8的值00000004，B为$1中的值00000008，送入ALU，ALU指令ALUOp为001，执行减操作，计算结果zero为0，表示不相等，extendData为立即数符号拓展后的值ffffffe,跳转到立即数符号拓展并左移两位再加上pc+4的地址，即in_pc的值00000018.

0x00000018	sll $8,$8,1	011000		01000	01000 00001		60084040



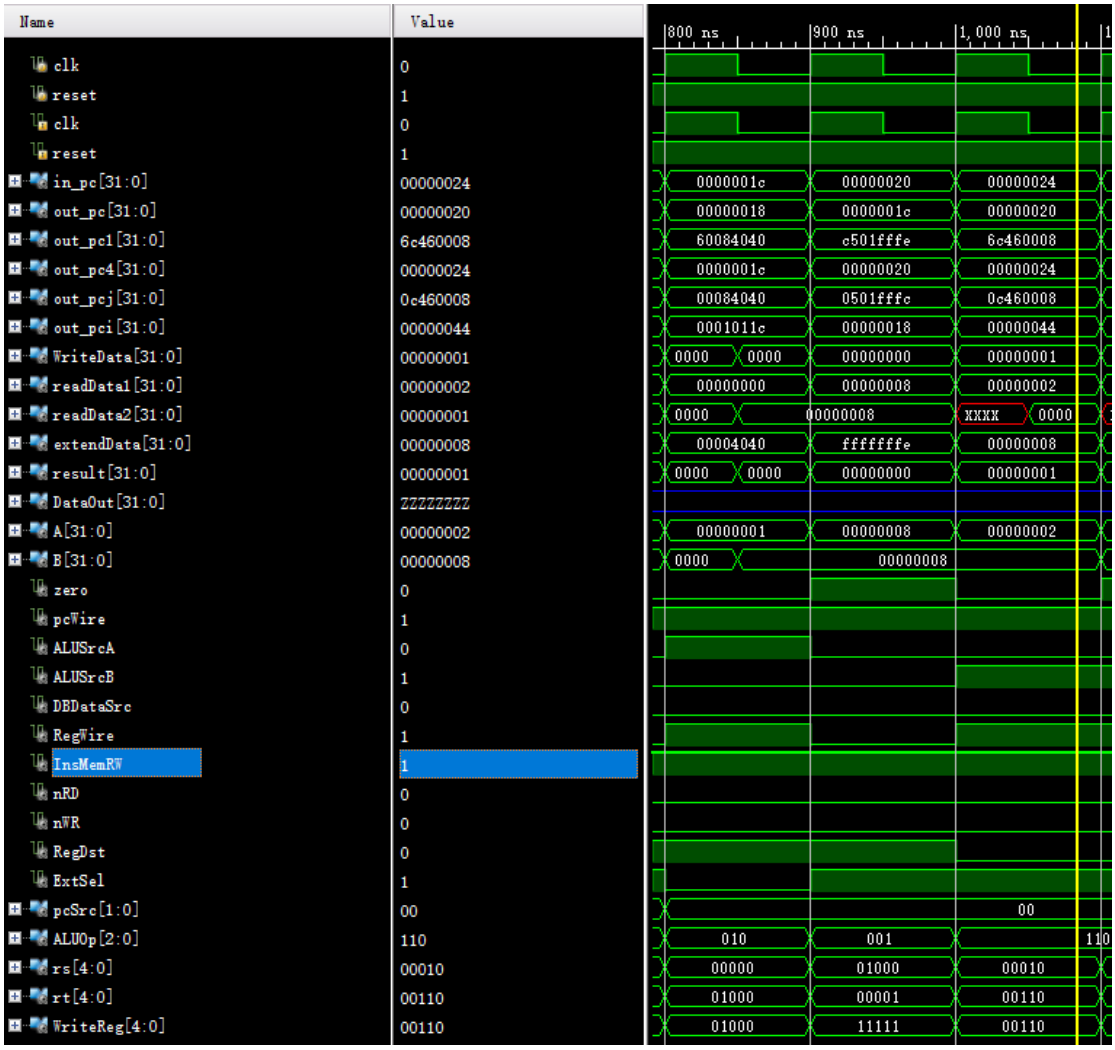
再次执行这条指令，操作数地址rt为01000，即$8，写入寄存器地址writeReg为01000，即$8,A为拓展后的立即数00000001，B为$8中的值00000004，送入ALU，ALU指令ALUOp为010，执行左移操作，计算结果result为00000008

0x0000001C	bne $8,$1,-2 (≠,转 18)	110001	01000	00001	1111 1111 1111 1110		C501FFFE



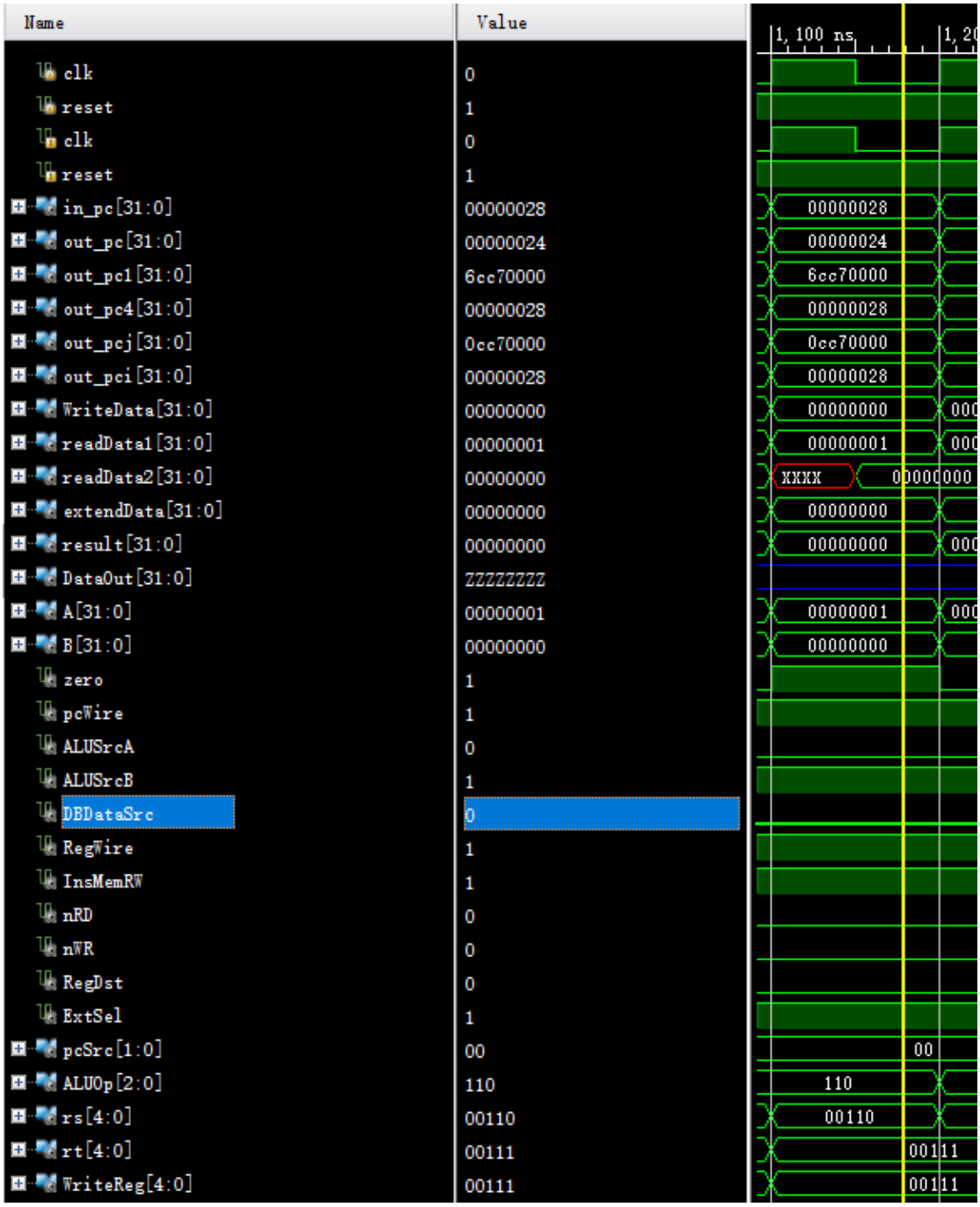
再次执行这条指令，操作数地址rt为01000，即$8，rt为00001，即$1,A为$8的值00000008，B为$1中的值00000008，送入ALU，ALU指令ALUOp为001，执行减操作，计算结果zero为1，表示相等，继续往下执行，即in_pc的值00000020

0x00000020	slti $6,$2,8	011011	00010	00110	0000 0000 0000 1000		6C460008



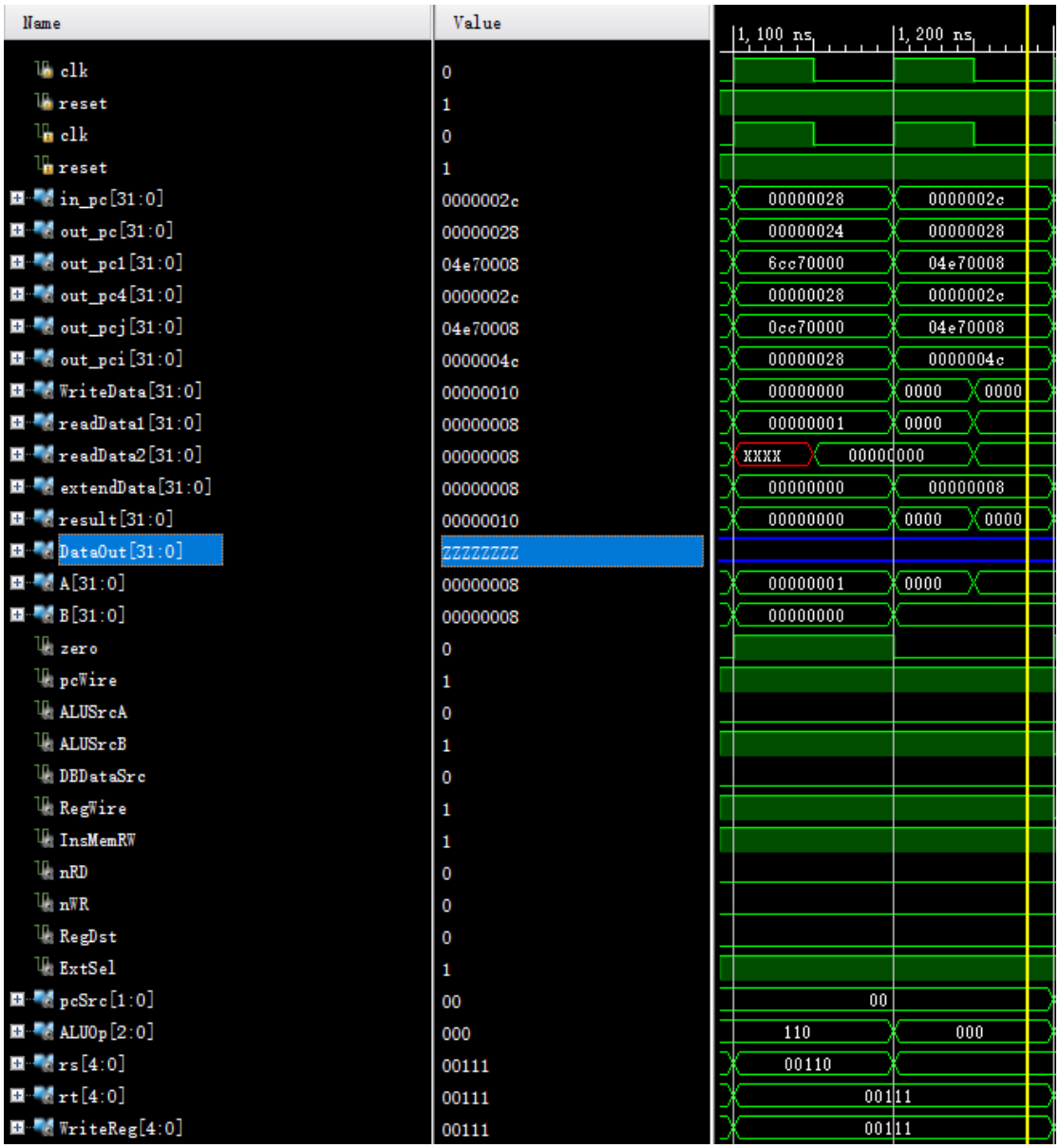
out_pc为当前指令地址00000020，out_pc1为指令6c460008，操作数地址rs为00010，即$2，写入寄存器地址writeReg=rt为00110,即$6,B为拓展后的立即数00000002，A为$2中的值00000002，送入ALU，ALU指令ALUOp为110，执行带符号比较操作，计算结果result为00000001，即$2的值<8.

0x00000024	slti $7,$6,0	011011	00110	00111	0000 0000 0000 0000		6CC70000



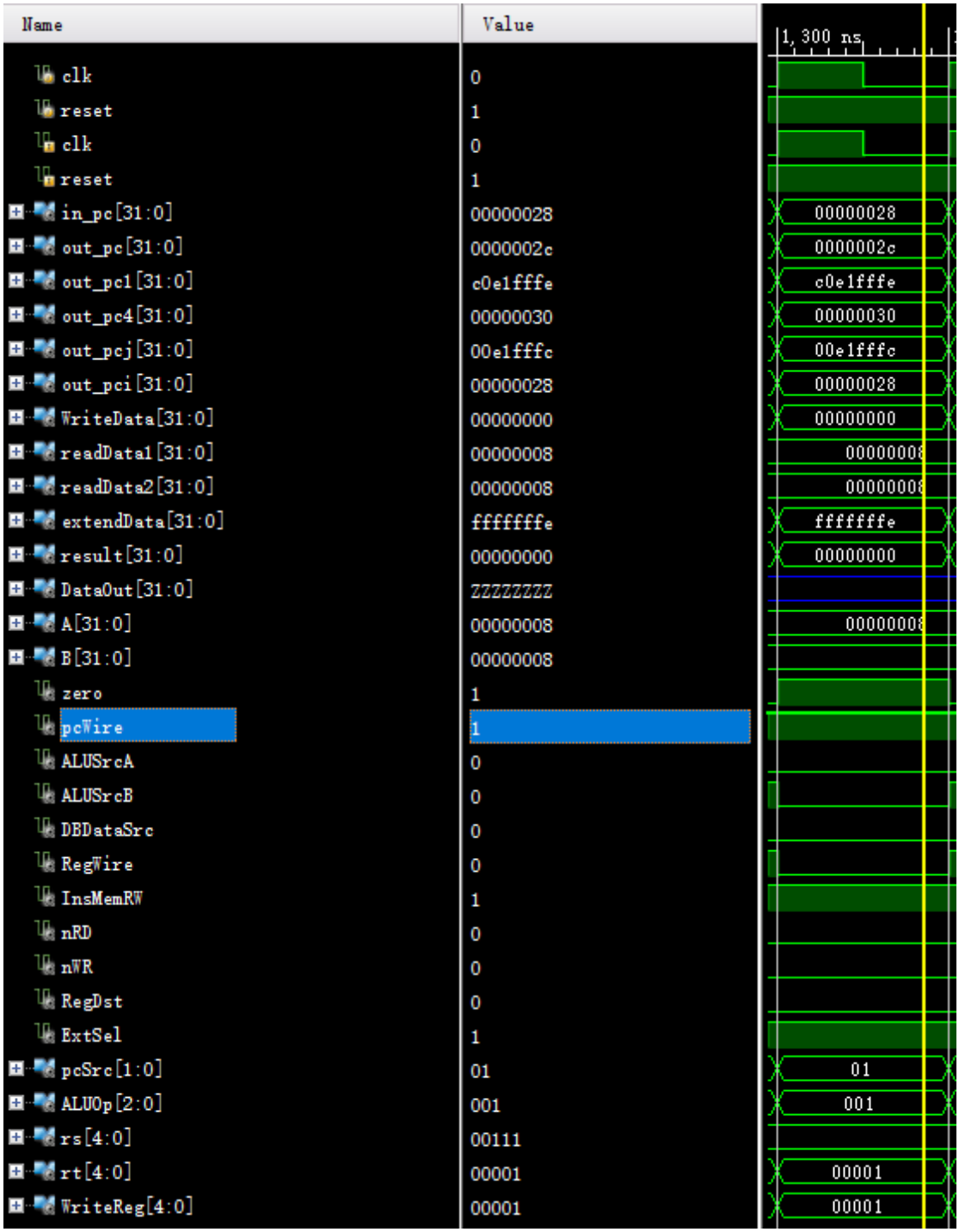
out_pc为当前指令地址00000024，out_pc1为指令6cc70000，操作数地址rs为00110，即$6，写入寄存器地址writeReg=rt为00111,即$7,B为拓展后的立即数00000000，A为$6中的值00000001，送入ALU，ALU指令ALUOp为110，执行带符号比较操作，计算结果result为00000000，即$6的值>0.

0x00000028	addi $7,$7,8	000001	00111	00111	0000 0000 0000 1000		04E70008



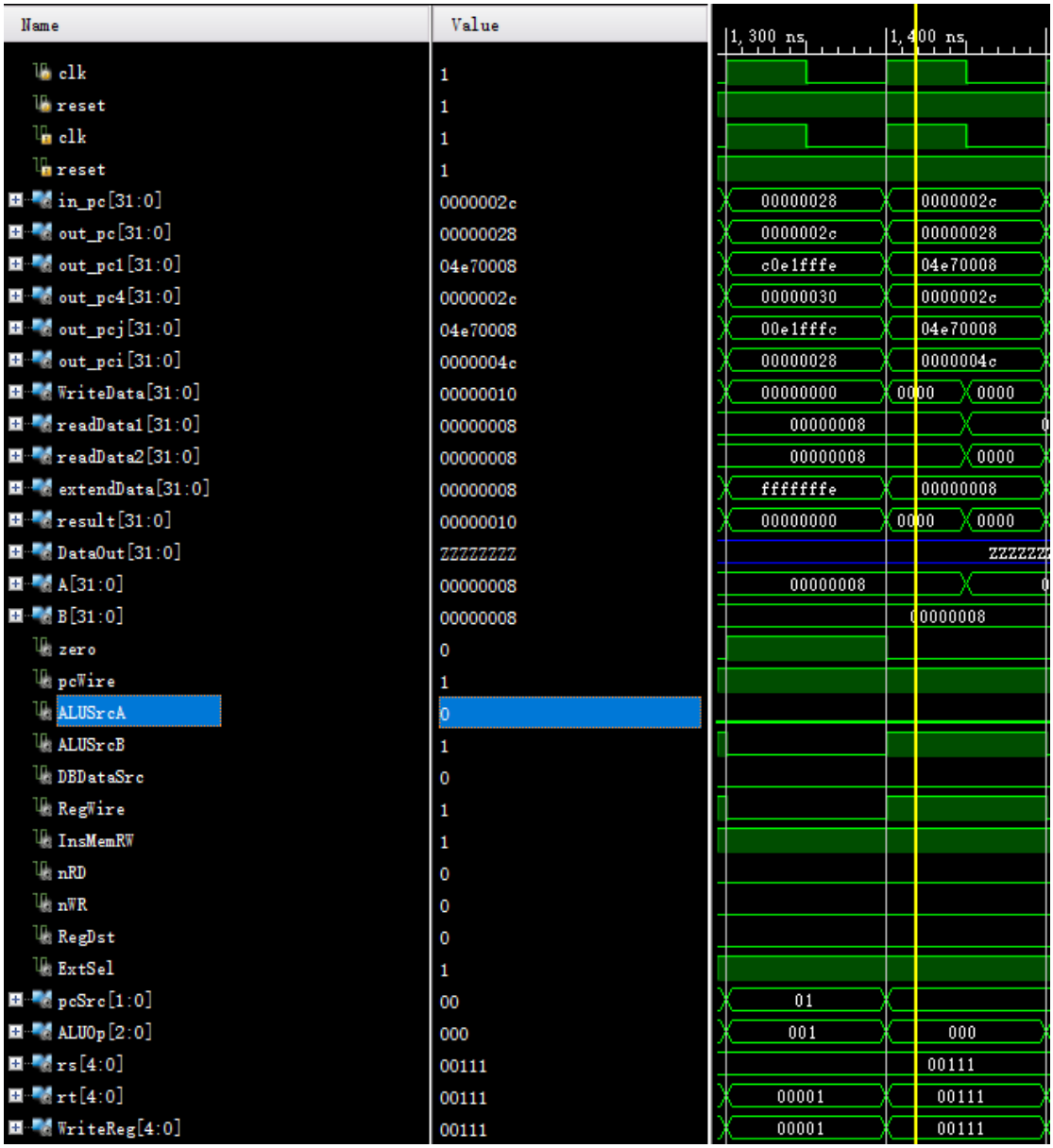
out_pc为当前指令地址00000028，out_pc1为指令04e70008，操作数地址rs为00111，即$7，写入寄存器地址writeReg=rt为00111,即$7,B为拓展后的立即数00000008，A为$7中的值00000000，送入ALU，ALU指令ALUOp为000，执行加操作，计算结果result为00000008，并写入$7.

0x0000002C	beq $7,$1,-2 (=,转 28)	110000	00111	00001	1111 1111 1111 1110		C0E1FFFE



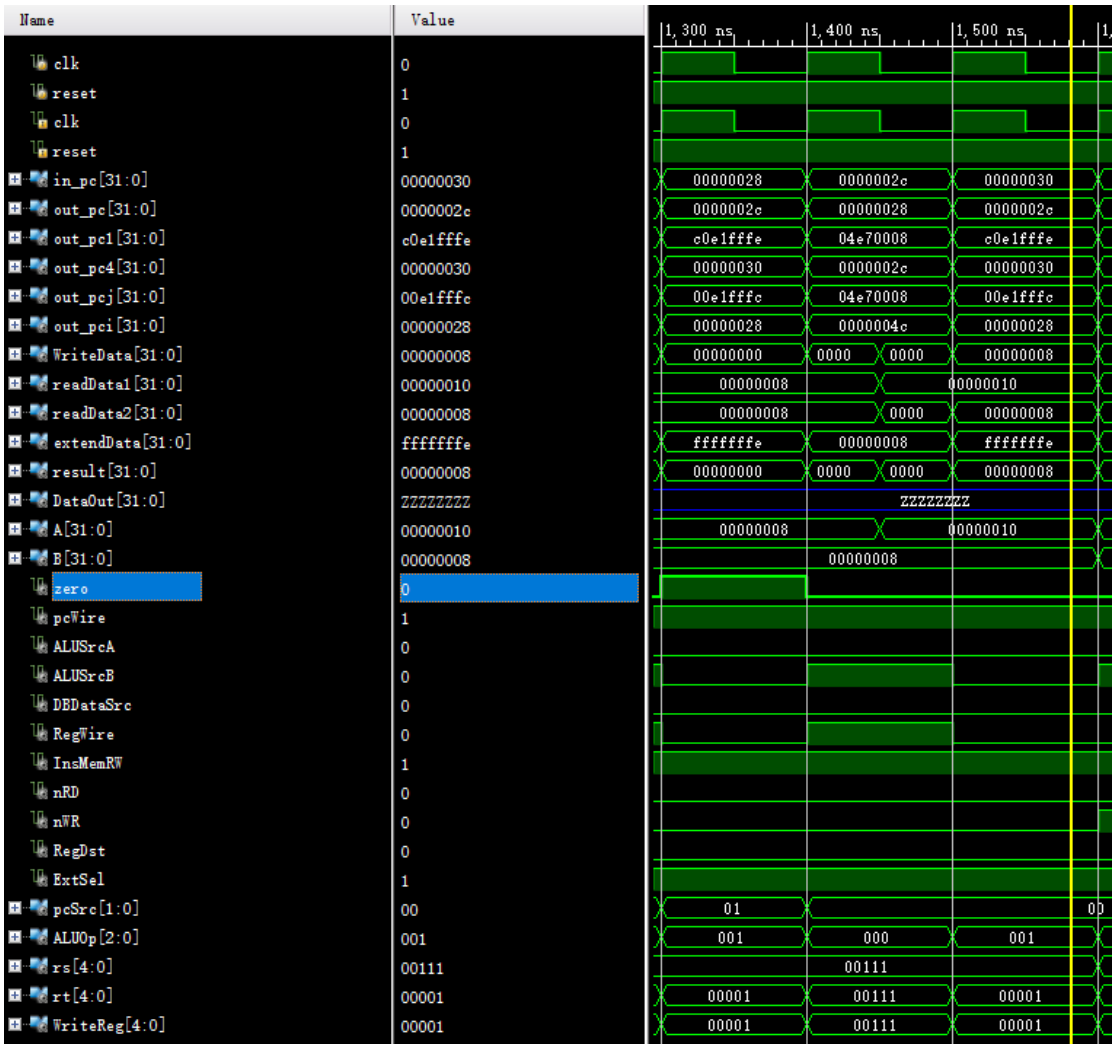
out_pc为当前指令地址0000002c，out_pc1为指令c0e1fffe，操作数地址rt为00001，即$1，rs为00111，即$7,A为$7的值00000008，B为$1中的值00000008，送入ALU，ALU指令ALUOp为001，执行减操作，计算结果zero为1，表示相等,extendData为立即数符号拓展后的值ffffffe，跳转到立即数符号拓展并左移两位再加上pc+4的地址，即in_pc的值00000028.

0x00000028	addi $7,$7,8	000001	00111	00111	0000 0000 0000 1000		04E70008



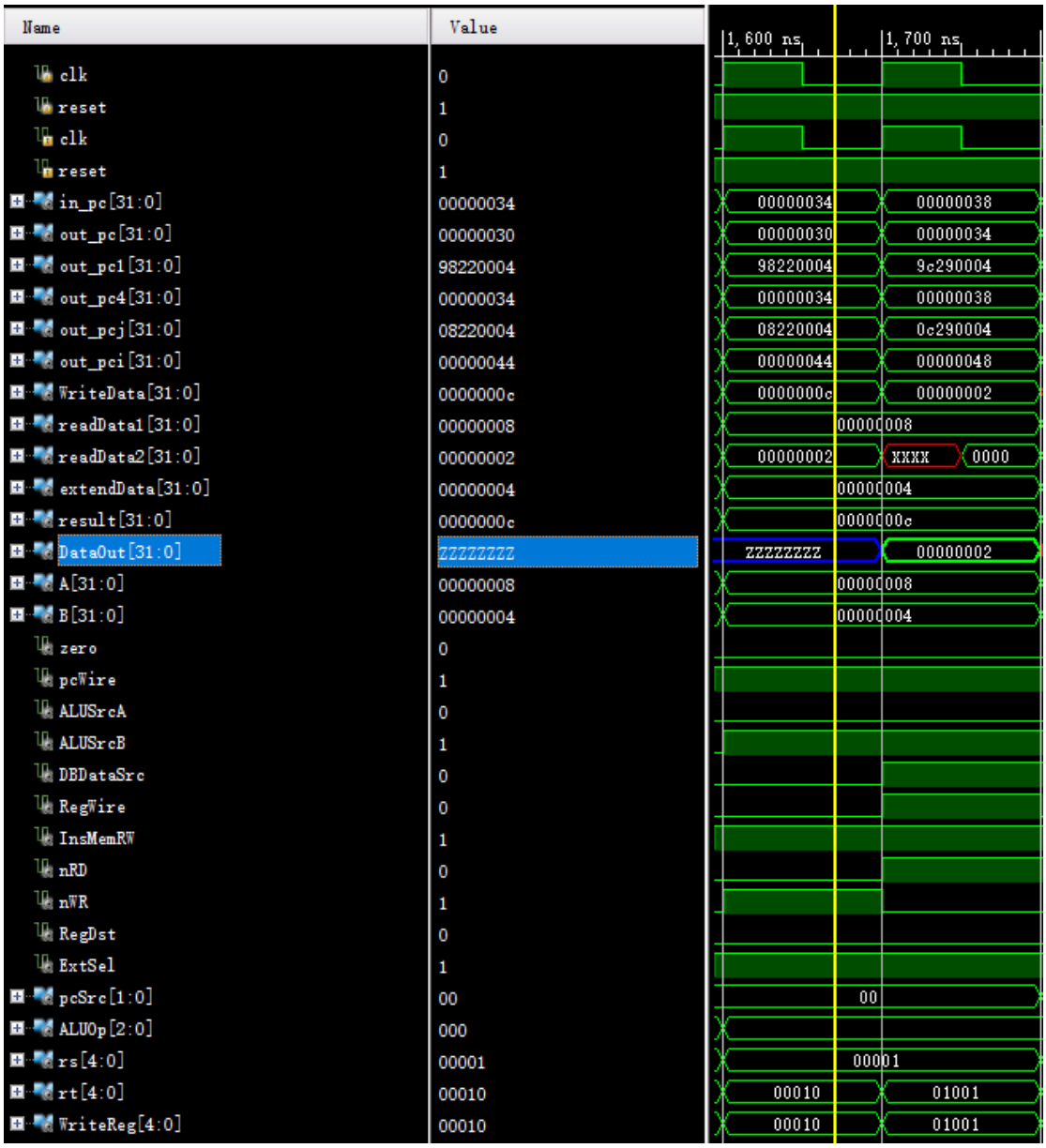
再次执行这条指令，out_pc为当前指令地址00000028，out_pc1为指令04e70008，操作数地址rs为00111，即$7，写入寄存器地址writeReg=rt为00111,即$7,B为拓展后的立即数00000008，A为$7中的值00000008，送入ALU，ALU指令ALUOp为000，执行加操作，计算结果result为00000010，并写入$7.

0x0000002C	beq $7,$1,-2 (=,转 28)	110000	00111	00001	1111 1111 1111 1110		C0E1FFFE



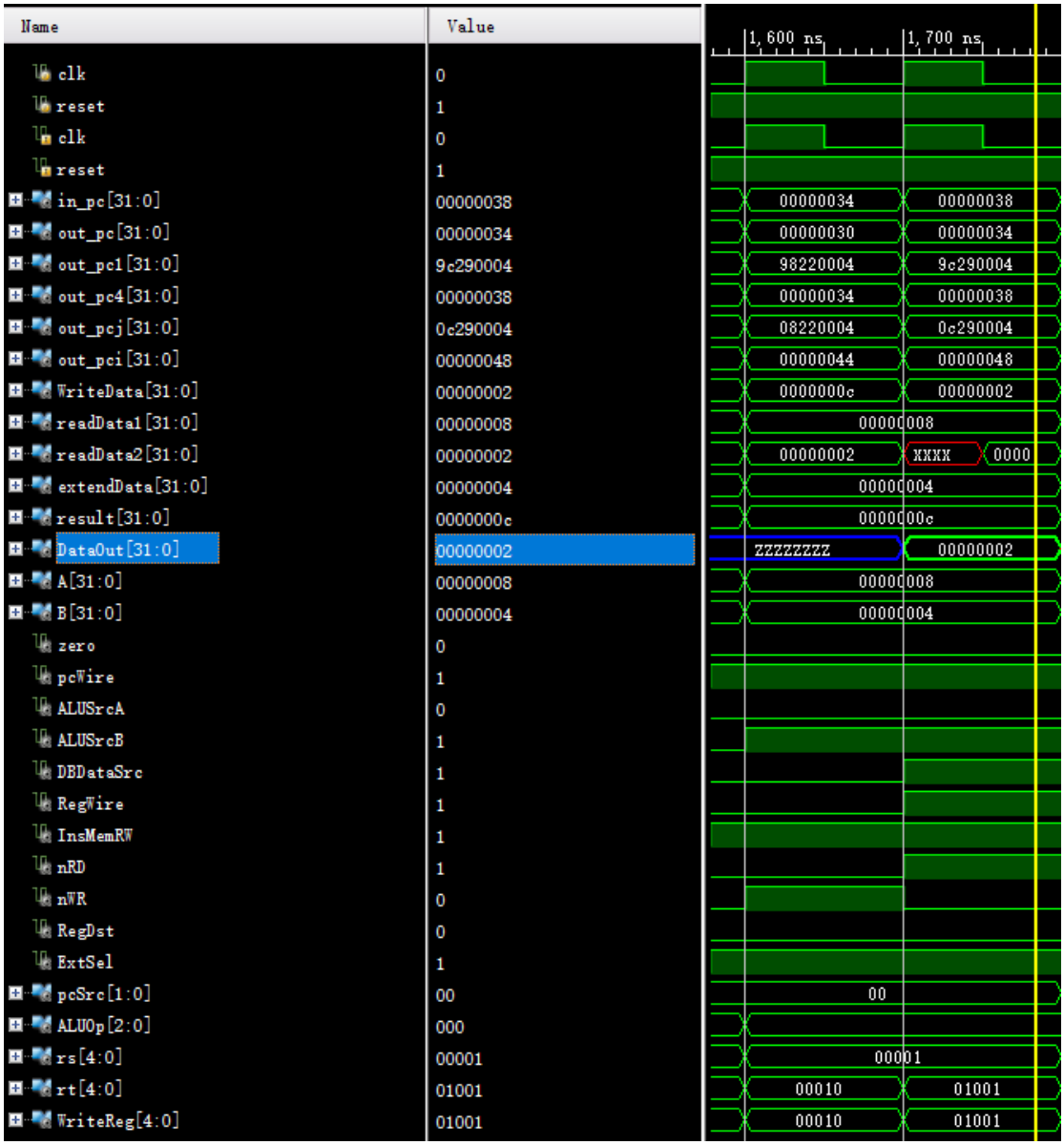
再次执行这条指令，out_pc为当前指令地址0000002c，out_pc1为指令c0e1fffe，操作数地址rt为00001，即$1，rs为00111，即$7,A为$7的值00000008，B为$1中的值00000008，送入ALU，ALU指令ALUOp为001，执行减操作，计算结果zero为0，表示不相等,跳转到pc+4的地址，即in_pc的值00000030.

0x00000030	sw $2,4($1)	100110	00001	00010	0000 0000 0000 0100		98220004



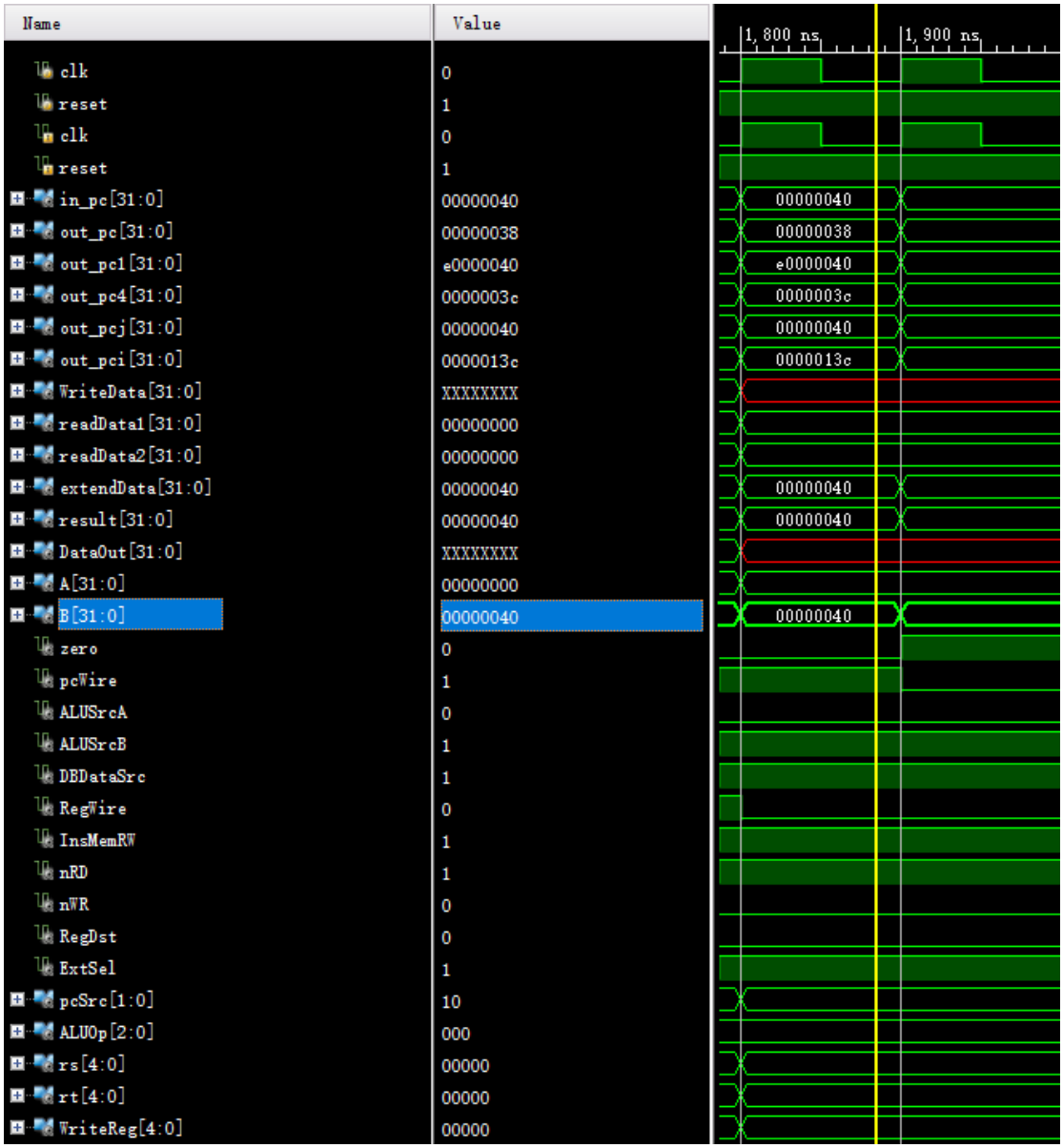
out_pc为当前指令地址00000030，out_pc1为指令98220004,操作数rt为需要存储的寄存器$2的地址00010，extendData为符号拓展后的立即数00000004，输出到B，rs为$1的地址00001，值输出到B，result为A+B，作为存储器的地址0000000c，nWR为1，执行写操作，可以从下一步读操作中确认正确性。

0x00000034	lw $9,4($1)	100111	00001	01001	0000 0000 0000 0100		9C290004



out_pc为当前指令地址00000034，out_pc1为指令9c290004,操作数rt为存储目的寄存器$2的地址01001，extendData为符号拓展后的立即数00000004，输出到B，rs为$1的地址00001，值输出到B，result为A+B，作为存储器的地址0000000c，nRD为1，执行读操作，DataOut即为数据00000002，输出到写入端WriteData为00000002，写入信号RegWire为1.

0x00000038	**j 0x00000040**	111000	00 0000 0000 0000 0000 0100 0000		E0000040

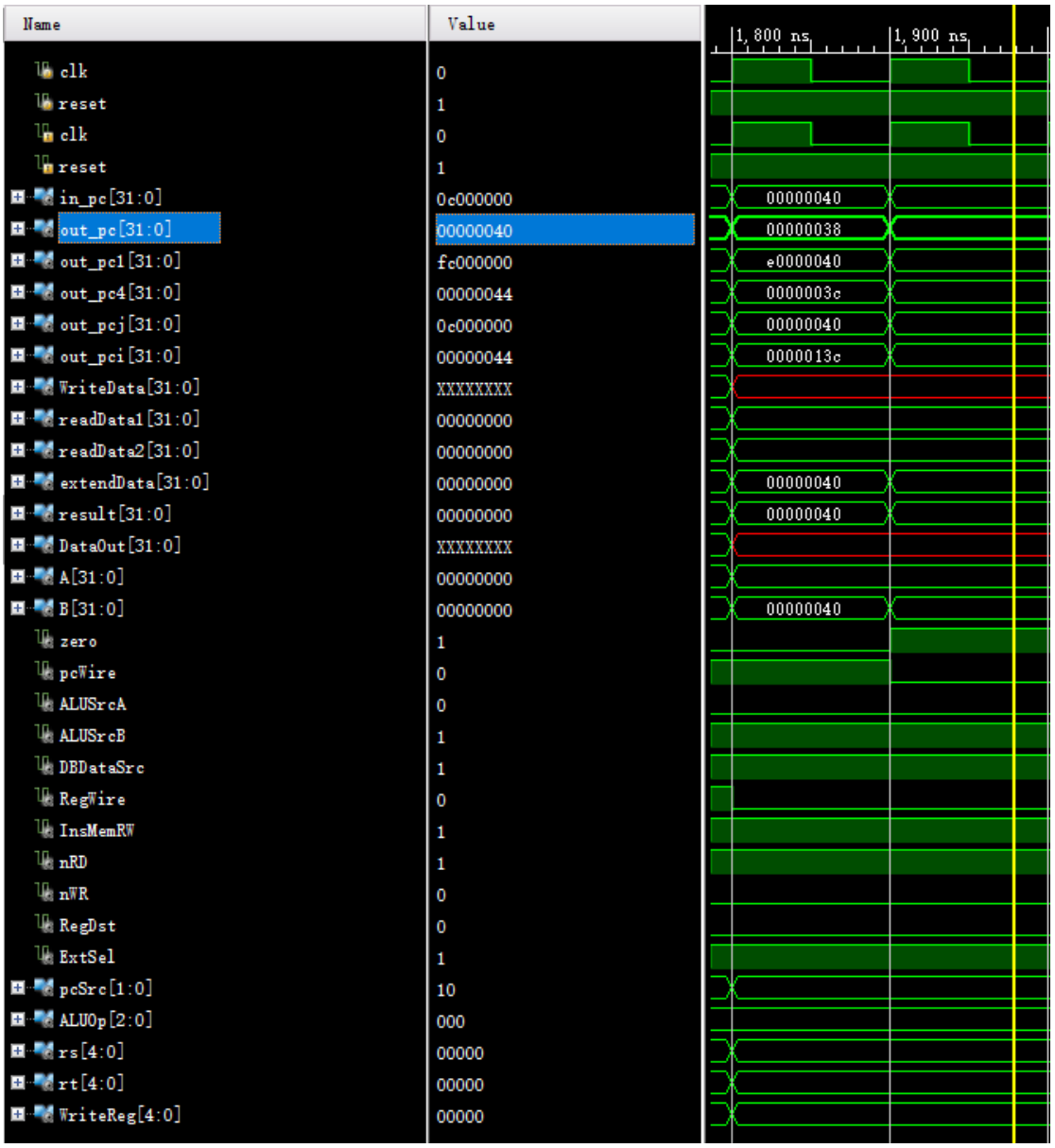


跳转到立即数extendData为00000040所代表的地址，下一条指令in_pc为00000040

0x0000003C	addi $10,$0,10	000001	00000	01010	0000 0000 0000 1010		040A000A

本条指令被跳过没有执行

0x00000040	**Halt**	111111	00000	00000	0000 0000 0000 0000	=	FC000000



停机，pc的值保持不变，in_pc和out_pc都为00000040

3、实现。如何在Basys3板上运行所设计的CPU？运行结果情况说明。

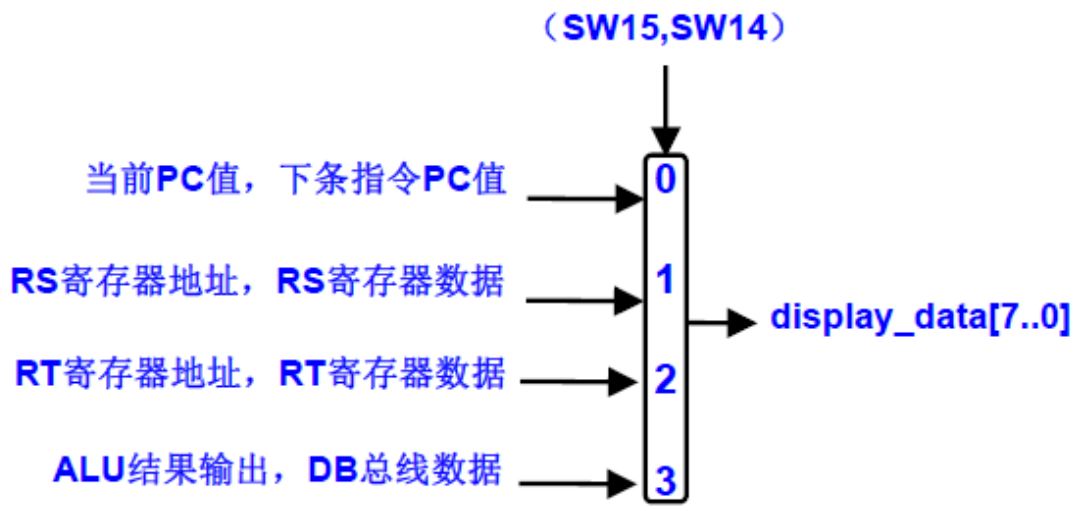
显示模块设计大概分为4个部分：

（1）对Basys3板系统时钟信号进行分频，分频的目的用于计数器；

（2）生成计数器，计数器用于产生4个数。这4数用于控制4个数码管；

（3）根据计数器产生的数生成数码管相应的位控信号（输出）和接收CPU来的相应数据；

（4）将从CPU 接收到的相应数据转换为数码管显示信号，再送往数码管显示（输出），即下页内容。



clock_div模块：时钟分频以及按键消抖，input clk为板上时钟，sw1是按键信号，output clk_sys是分频之后的信号，用作数码管选位显示，clk2为消抖之后的信号，作为cpu时钟。

```
module clock_div(
    input clk, //100MHz□□□□□□□□,
    input sw1,
    output reg clk_sys = 0, //1Hz□□□□□□□□□□□□□□□0,
    output reg clk2
);

    reg [25:0] div_counter = 0;
    always @(posedge clk) begin
    //if(div_counter>=1) begin // □□□□
    if(div_counter>=100000) begin // □□□□□□□□□□□□
        clk_sys <= ~clk_sys; // □□□□
        div_counter <= 0;
    end else begin
        div_counter <= div_counter + 1;
        end
    end
    reg [1:0] counter = 0;
    always @( posedge clk) begin
        if(sw1 == 1) begin
            clk2 = 1;//□□□□CPU□□□□□
            counter = 0;
        end
        else begin
            counter = counter + 1;
            if(counter == 3)
                clk2 = 0; //CPU□□□□□
        end
    end

endmodule
```

四选一选择器：根据按键sw2和sw3来区分显示数据。

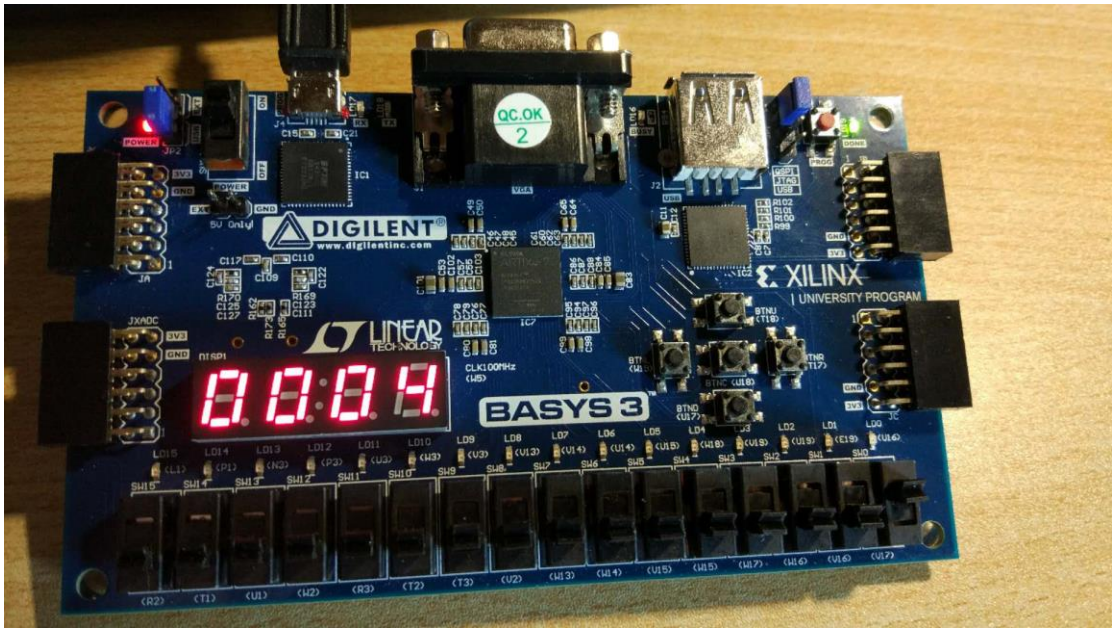
counter模块：根据分频后的信号依次产生控制数码管的选位信号，一次只亮一位。

led模块，根据数据输入产生数码管的显示信号。

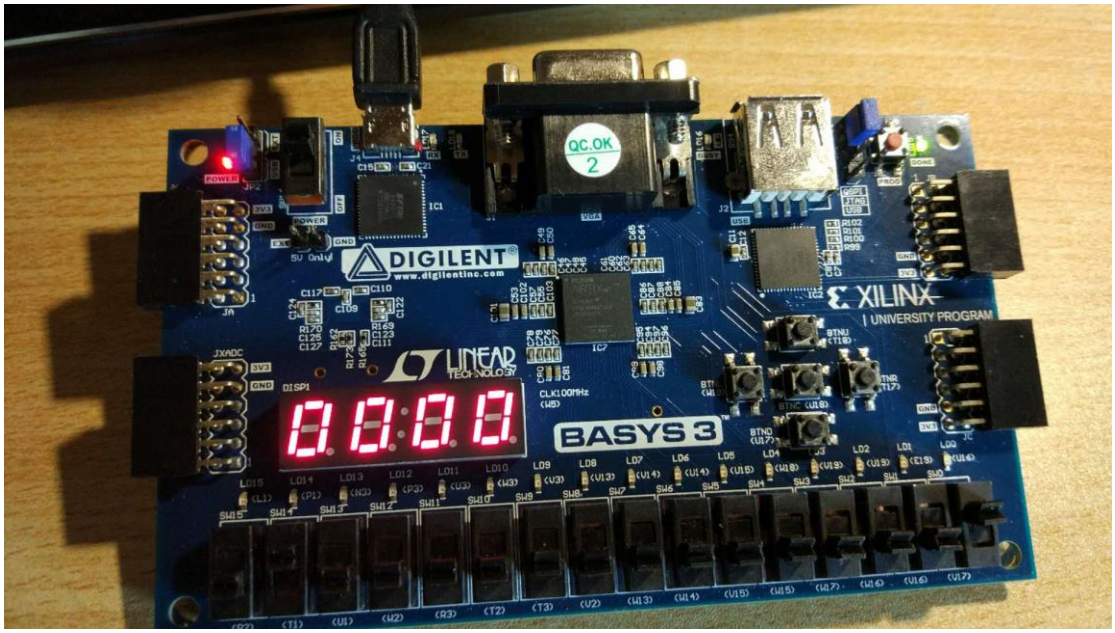
最后在main模块里将要显示的数据送入四选一。

```
module main(
    input b_clk,reset,sw1,sw2,sw3,
    output t,
    output [7:0] led,
    output [3:0] sign
    );

    wire clk,clk1;
    wire [31:0] out_pc1,out_pc4,out_pcj,out_pci,readData1,readData2,extendData,DataOut,out_pc,in_pc,WriteData,result,A,B;
    wire zero,pcWire,ALUSrcA,ALUSrcB,DBDataSrc,RegWire,InsMemRW,nRD,nWR,RegDst,ExtSel;
    wire [1:0] pcSrc;
    wire [2:0] ALUOp;
    wire [4:0] WriteReg,rs,rt;
    wire [3:0] dis0;
    wire [7:0] s1,s0;


    assign rs=out_pc1[25:21];
    assign rt=out_pc1[20:16];
    assign t=clk;

    clock_div clock_div(b_clk,sw1,clk1,clk);
    counter counter(clk1,sign);
    fourSelectOne1 fourSelectOne1(clk1,out_pc,result,rs,rt,sw2,sw3,s1);
    fourSelectOne2 fourSelectOne2(clk1,in_pc,A,B,WriteData,sw2,sw3,s0);
    fourSelectOne fourSelectOne(s1[7:4],s1[3:0],s0[7:4],s0[3:0],sign,dis0);
    SegLED SegLED(in_pc[3:0],led);
```

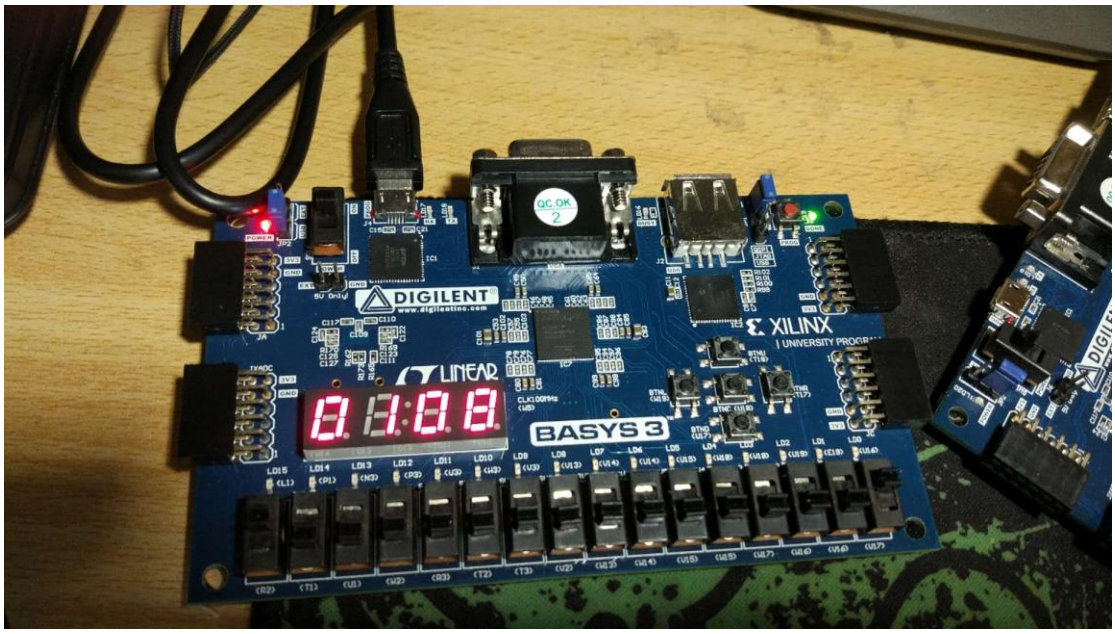
第一条指令地址：下一条指令地址



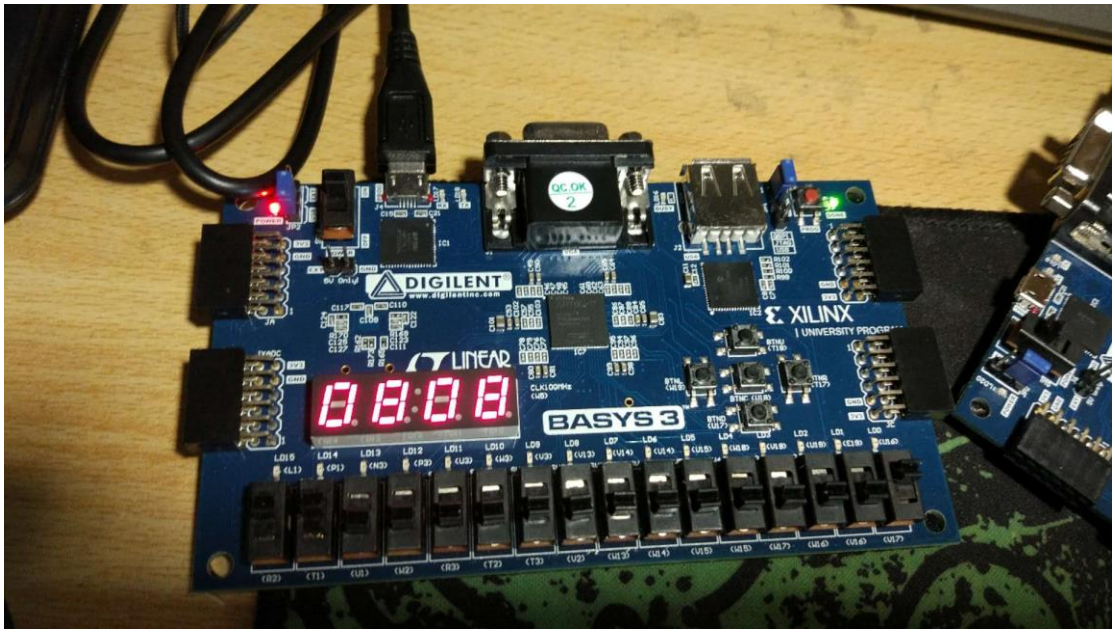
rs寄存器地址：rs寄存器值



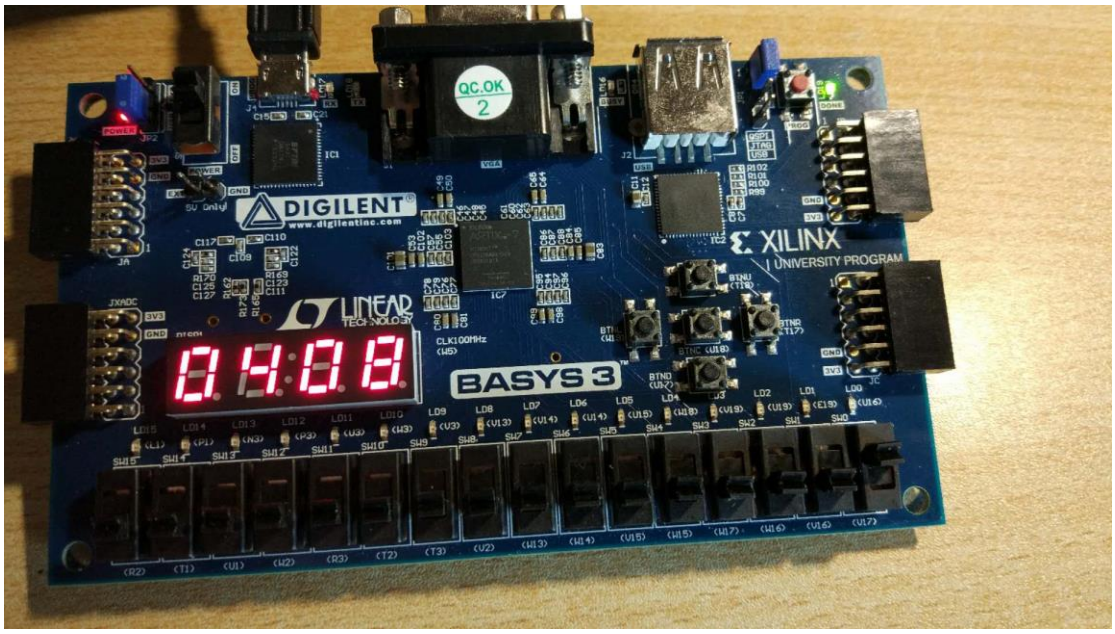
rt寄存器地址：rt寄存器值

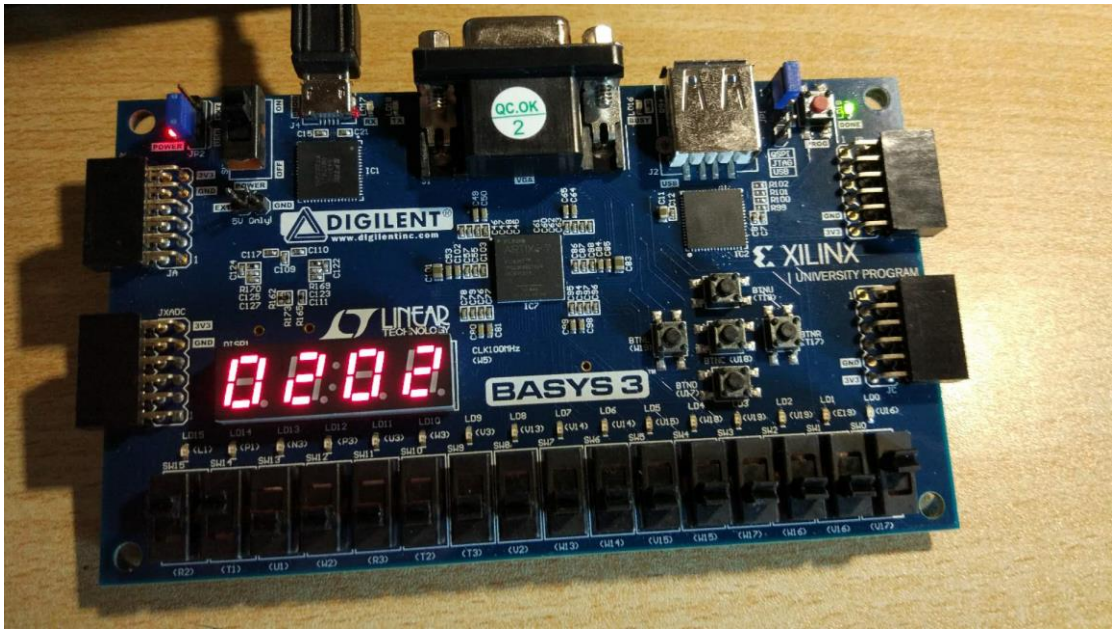


result:writeData



下一条指令





六. 实验心得

做这次实验，首先是要看懂那张数据通路图，尽管老师上课讲过一遍，但一开始还是没有完全理解，于是我对着机组理论给的CPU设计的电子书，仔细看了每个模块的原理，和老师提供的部分模块的代码，才对整个项目有了一定的理解。首先从单个模块开始设计，我是从Control Unit模块开始设计，这个模块的原理比较简单，就是根据输入的不同的操作码生成各种控制信号，控制信号可以根据老师提供的表对应着写。有这个模块入手，后面的思路就很清晰了，紧接着就写ALU，数据选择器和符号拓展等模块，最后照着图连线写出主模块。

指令部分，几个跳转指令需要多一些考虑。beq和bne需要判断条件是否满足，即后面给出的两个操作数的值是否相等，这里要利用ALU的减法功能而不是比较功能，因为比较只是判断<=和>=，并不能一次判断出是否相等，所以使用减法看结果是否为0，如果为0则将标志位置为1，表示相等，这个结果送入CU作为生成PC的数据选择器的控制信号的依据。

在一开始仿真的时候，不知道如何生成时钟信号，以及初始化reset，后来经过询问他人和网上查询，知道 #1 语句，即延迟1个时间单位再执行后面的操作。就可以在测试模块里设置两个不同的时钟。在烧板的时候有一个问题，就是按键时可能会产生波形的抖动，这里就需要进行消抖处理，添加一个计数器，按下后延长处于高电位的时间。

总的来说，这次实验让我深入了解了单周期cpu的工作原理和模块化的设计方法，以及如何测试验证设计的正确性和烧板的实现，为以后的学习打下了坚实的基础。