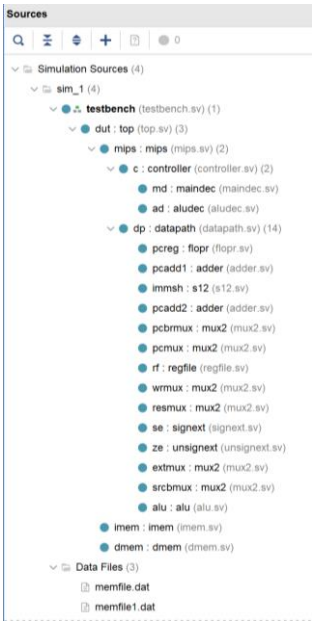
实验一　32 位 MIPS 单周期处理器的设计

一、【设计原理】

指令存储在 imem.sv 中，Top 模块逐条取出指令，并将其数据输入到 mips.sv 中计算，支持的运算有 add, sub, and, or, slt, sw, lw, addi, ori, andi, beq, bne 共 12 种。

1.添加 I/O 接口前，拓展 andi, ori, bne 指令后

项目的仿真源代码结构如下，其中 testbench.sv 是仿真测试代码，其余都是源代码：



其中 top.sv 的输入为 clk（时钟周期信号）和 reset（重置信号），输出为 writedata（写入的数据）, dataadr（写入地址）和 memwrite（写入使能）。

2.添加 I/O 接口后

项目 SingleCycleProcessor 的源代码结构如下：

顶层模块的 Top.sv 文件的输入有 CLK100MHZ（时钟信号），BTNL（加法控制信号），BTNR（操作执行信号），BTNC（系统清零信号）和 SW[15:0]（两个 8 位输入），输出有 AN[7:0]（8 个七段数码管）和 A2G[6:0]（七段数码管）。

二、【实验方案与部分关键代码】

实验源代码与书上和课件上的代码相比，主要有以下增改：

1.实现 mips 中的 alu

alu.sv 源代码如下：

```
`timescale 1ns / 1ps
module alu(input    logic [31:0] SrcA, SrcB,
                input    logic [2:0]    ALUControl,
                output logic [31:0] ALUResult,
                output logic              Zero);
    logic [31:0] S, B;
    assign B = ALUControl[2] ? ~SrcB : SrcB;
    assign S = SrcA + B + ALUControl[2];
    always_comb
        case (ALUControl[1:0])
            2'b00: ALUResult <= SrcA & B;
            2'b01: ALUResult <= SrcA | B;
            2'b10: ALUResult <= S;
            2'b11: ALUResult <= S[31];
        endcase
    assign Zero = (ALUResult == 2'b0);
```

endmodule

ALUControl 控制 ALU 的计算方式，对应关系如下表：

ALUControl	000	001	010	110	111
运算类型	与	或	加	减	小于置位

2.扩展 andi, ori 和 bne 指令

(1)andi 与 ori

为了扩展 andi 和 ori 指令，我将 aluop 由 2 位扩充至 3 位，先前指令的扩充位为 0，后两位不变。
andi 和 ori 都是 i 型指令，[15:0]代表立即数，所以如果不进行扩充，有可能在 aludec.sv（alu 译码器）里将立即数的[5:0]视作 funct 字段处理，这显然是错误的。
扩充 aluop 后，我在 aludec.sv 的 case(aluop)语句里添加了两句内容，请见于以下 aludec.sv 文件源代码的红色字段：

```
`timescale 1ns / 1ps
module aludec(input    logic [5:0] funct,
                       input    logic [2:0] aluop,
                       output logic [2:0] alucontrol);
    always_comb
        case(aluop)
            3'b000: alucontrol <= 3'b010; //add (for lw/sw/addi)
            3'b001: alucontrol <= 3'b110; //sub (for beq/bne)
            3'b011: alucontrol <= 3'b001; //ori
            3'b100: alucontrol <= 3'b000; //andi
            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'bxxx; //???
            endcase
        endcase
endmodule
```

当 aluop == 100 时，检测到 andi 指令，使 alu 作逐位与操作；当 aluop == 011 时，检测到 ori 指令，使 alu 作逐位或操作。

另外在 mips.sv 中引入变量 immext，以便于判断 i 型指令是 andi 与 ori 这样的逻辑运算还是

addi, beq 和 bne（后两者在 alu 中进行减法运算）这样的算数运算。
若 immext 等于 1 则对立即数进行 0 扩展，判断指令为 andi 与 ori；等于 0 则进行符号扩展，判断指令不为这两者。

(2)bne

bne 与 beq 类似，都是分支跳转指令，区别在于 beq 是两者相等时跳转而 bne 是两者不等时跳转，所以可以在原有的 beq 路径基础上添加变量以引入 bne，当满足二者其一时执行跳转指令。
具体方式是在 controller.sv 中增加变量 branchbne 表示是否为 bne 指令，如果指令 instr[31:0]的操作码为 bne（000101），则在主译码器 main.dec 中置 branchbne 为 1，否则为 0。
然后取 pcsrc = (branch & zero) | (branchbne & ~zero)（书上的代码为 pcsrc = branch & zero），作为跳转指令的判断。
controller.sv 中的源代码如下，红色字段为修改过的代码：

```
`timescale 1ns / 1ps
module controller(input    logic [5:0] op, funct,
                          input    logic          zero,
                          output logic          memtoreg, memwrite,
                          output logic          pcsrc, alusrc,
                          output logic          regdst, regwrite,
                          output logic          jump,
                          output logic          immext,
                          output logic [2:0] alucontrol);
    logic [2:0] aluop;
    logic          branch, branchbne;

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

    assign pcsrc = (branch & zero) | (branchbne & ~zero);
endmodule
```

3.实现 0 扩展

为了实现 andi 与 ori 指令，要对 instr[15:0]进行 0 扩展，源代码 unsignext.sv 如下：

```
`timescale 1ns / 1ps
module unsignext(input    logic [15:0] a,
                          output logic [31:0] y);
    assign y[15:0] = a;
```

```
    assign y[31:16] = 0;
endmodule
```

4.实现数据存储译码器 dMemoryDecoder

引入变量 pWrite 和 mWrite 实现解复用器，使用 addr[7]控制写入使能 writeEN 的数据流向，具体方式如下：

```
assign pWrite = (addr[7] == 1'b1) ? writeEN : 0;
assign mWrite = writeEN & (addr[7] == 1'b0);
```

其中 pWrite 是 io.sv 的写入使能，mWrite 是数据存储器（dmem.sv）的写入使能。

另外新建变量 digit[31:0] = switch[15:0] + 4'b0000 + led[11:0]，前 16 位为输入的数据（对应 16 个开关 SW[15:0]），后 12 位为输出的数据（后三个数码管）。将变量 digit 传入七段数码管译码器中（mux7seg.sv）。

dMemoryDecoder.sv 源代码如下：

```
`timescale 1ns / 1ps
module dMemoryDecoder(input  logic        clk,
                      input  logic        writeEN,
                      input  logic [31:0] addr,
                      input  logic [31:0] writedata,
                      output logic [31:0] readdata,
                      input  logic        IOclock,
                      input  logic        reset,
                      input  logic        btnL,
                      input  logic        btnR,
                      input  logic [15:0] switch,
                      output logic [7:0]  an,
                      output logic [6:0]  a2g);

    logic [11:0] led;
    logic [31:0] readdata1, readdata2;
    logic [31:0] digit;
    logic pWrite, mWrite;

    assign pWrite = (addr[7] == 1'b1) ? writeEN : 0;
    assign mWrite = writeEN & (addr[7] == 1'b0);
    dmem dMemory(.clk(clk),
                 .we(mWrite),
                 .a(addr),
                 .wd(writedata),
                 .rd(readdata1));

    IO io(.clk(IOclock),
          .reset(reset),
```

```
            .pRead(addr[7]),
            .pWrite(pWrite),
            .addr(addr[3:2]),
            .pWriteData(writedata[11:0]),
            .pReadData(readdata2),
            .buttonL(btnL),
            .buttonR(btnR),
            .switch(switch),
            .led(led));

    assign digit = {switch, 4'b0000, led};

    mux7seg m7seg(.clk(IOclock),
                  .reset(reset),
                  .x(digit),
                  .an(an),
                  .a2g(a2g));

    mux2 #(32) rdmux(readdata1, readdata2, addr[7], readdata);
endmodule
```

5.实现七段显示管译码器

使用时钟分频信号 clkdiv 减缓数码管的更新频率。七段显示管从数据存储译码器得到 x[31:0]（即 dmdecoder 中的 digit[31:0]），使用变量 s[2:0]随时钟分频信号 clkdiv 的增加而递增，对一个特定的 s 值，digit[3:0]可依次取得某个数码管的输出权（如 x[3:0], x[7:4]……），并根据 digit[3:0]的值改变 a2g[6:0]即 Top 中的 A2G[6:0]，达到输出的目的。

另外我将 case(digit)语句中增加了判断，当 s = 3 即 digit 等于从右向左数第 4 个数字时，x[11:0]即计算结果是否等于 0，如果此时计算结果不为 0，则将从右向左数第 4 个数字变成“=”，否则使它为 x[15:12]所表示的数（在此实验中因为最大结果为 h'1FE，所以 x[15:12]始终为 0）。mux7seg.sv 源代码如下，此处判断已用红色标明：

```
`timescale 1ns / 1ps
module mux7seg(input       logic             clk,
               input       logic             reset,
               input       logic [31:0] x,
               output      logic [7:0]   an,
               output      logic [6:0]   a2g);

    logic [2:0] s;
    logic [3:0] digit;
    logic [19:0] clkdiv;
```

```
  assign s = clkdiv[19:17];

  always_comb
  case(s)
     0: digit = x[3:0];
     1: digit = x[7:4];
     2: digit = x[11:8];
     3: digit = x[15:12];
     4: digit = x[19:16];
     5: digit = x[23:20];
     6: digit = x[27:24];
     7: digit = x[31:28];
     default: digit = x[3:0];
endcase

  always_comb
  case(s)
     0: an = 8'b1111_1110;
     1: an = 8'b1111_1101;
     2: an = 8'b1111_1011;
     3: an = 8'b1111_0111;
     4: an = 8'b1110_1111;
     5: an = 8'b1101_1111;
     6: an = 8'b1011_1111;
     7: an = 8'b0111_1111;
     default: an = 8'b1111_1110;
  endcase

    always @(posedge clk or posedge reset)
       begin
          if(reset == 1)
             clkdiv <= 0;
          else
             clkdiv <= clkdiv+1;
       end

  always_comb
     begin
     if (s == 3 & x[11:0] != 0)
        a2g = 7'b0110111;
     else
        case(digit)
          'h0: a2g = 7'b1000000;
          'h1: a2g = 7'b1111001;
```

```
                'h2: a2g = 7'b0100100;
                'h3: a2g = 7'b0110000;
                'h4: a2g = 7'b0011001;
                'h5: a2g = 7'b0010010;
                'h6: a2g = 7'b0000010;
                'h7: a2g = 7'b1111000;
                'h8: a2g = 7'b0000000;
                'h9: a2g = 7'b0010000;
                'hA: a2g = 7'b0001000;
                'hB: a2g = 7'b0000011;
                'hC: a2g = 7'b1000110;
                'hD: a2g = 7'b0100001;
                'hE: a2g = 7'b0000110;
                'hF: a2g = 7'b0001110;
                default: a2g = 7'b1000000;
            endcase
        end
endmodule
```

三、【仿真截图】

1.增加 andi，ori 和 bne 指令后，执行指令文件 memfile.dat（书上测试代码）

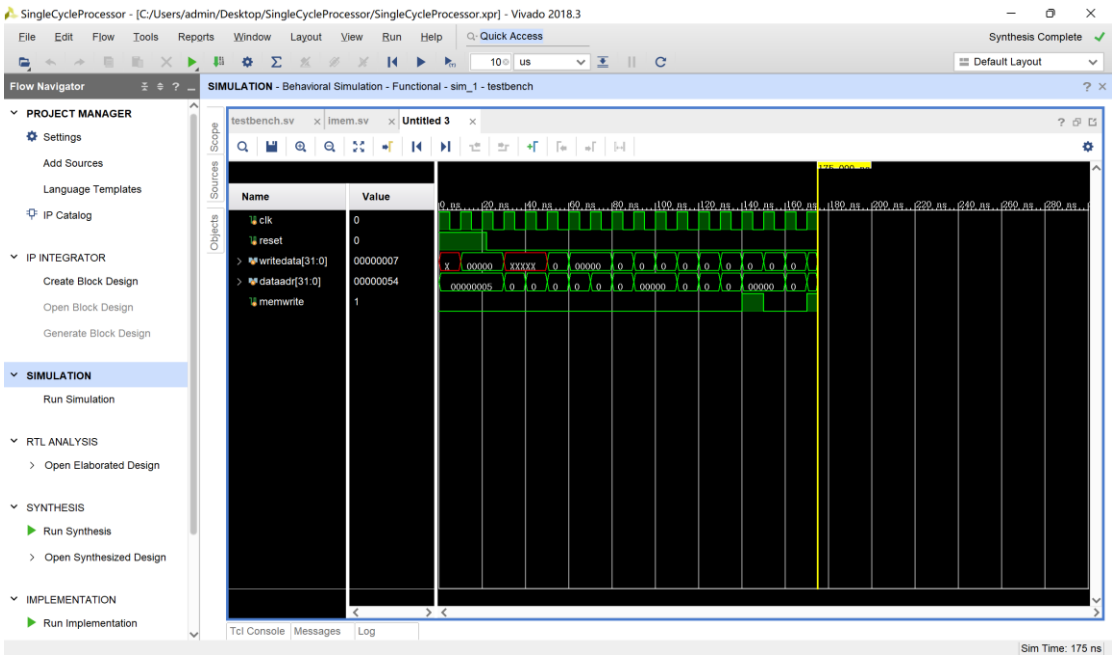
(1)memfile.dat 内容

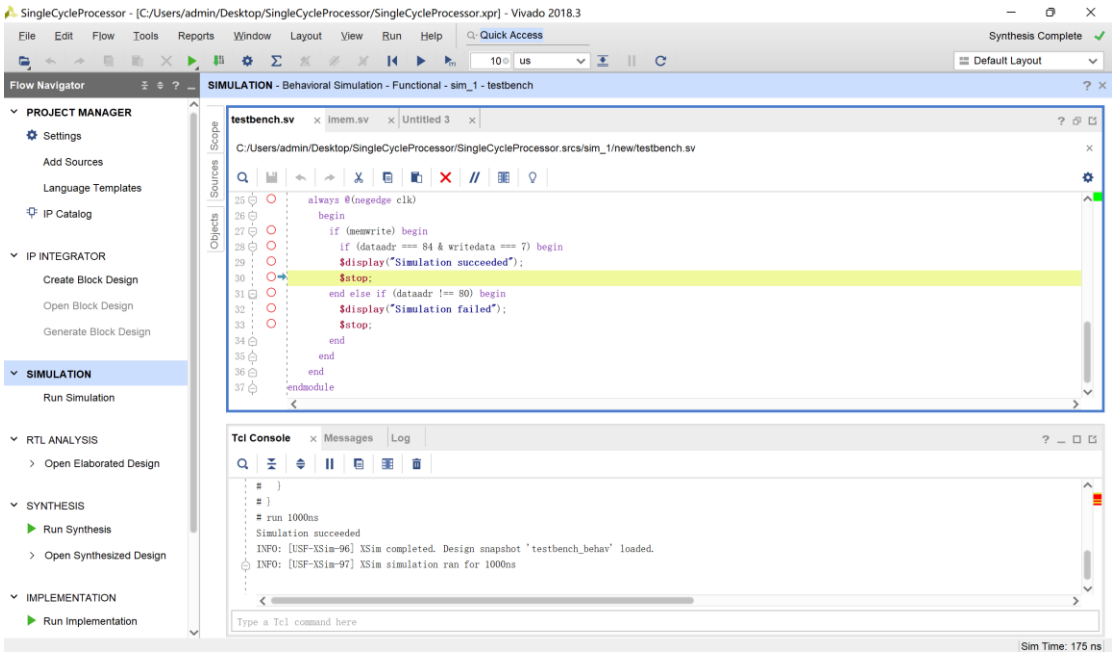
```
20020005
2003000c
2067fff7
00e22025
00642824
00a42820
10a7000a
0064202a
10800001
20050000
00e2202a
00853820
00e23822
ac670044
8c020050
08000011
20020001
```

ac020054

```
#          Assembly              Description              Address    Machine
main:      addi $2, $0, 5        # initialize $2 = 5       0          20020005
           addi $3, $0, 12       # initialize $3 = 12      4          2003000c
           addi $7, $3, -9       # initialize $7 = 3       8          2067fff7
           or   $4, $7, $2       # $4 = (3 OR 5) = 7       c          00e22025
           and  $5, $3, $4       # $5 = (12 AND 7) = 4     10         00642824
           add  $5, $5, $4       # $5 = 4 + 7 = 11         14         00a42820
           beq  $5, $7, end      # shouldn't be taken      18         10a7000a
           slt  $4, $3, $4       # $4 = 12 < 7 = 0         1c         0064202a
           beq  $4, $0, around   # should be taken         20         10800001
           addi $5, $0, 0        # shouldn't happen        24         20050000
around:    slt  $4, $7, $2       # $4 = 3 < 5 = 1          28         00e2202a
           add  $7, $4, $5       # $7 = 1 + 11 = 12        2c         00853820
           sub  $7, $7, $2       # $7 = 12 - 5 = 7         30         00e23822
           sw   $7, 68($3)       # [80] = 7                34         ac670044
           lw   $2, 80($0)       # $2 = [80] = 7           38         8c020050
           j    end              # should be taken         3c         08000011
           addi $2, $0, 1        # shouldn't happen        40         20020001
end:       sw   $2, 84($0)       # write mem[84] = 7       44         ac020054
```

(2)仿真截图





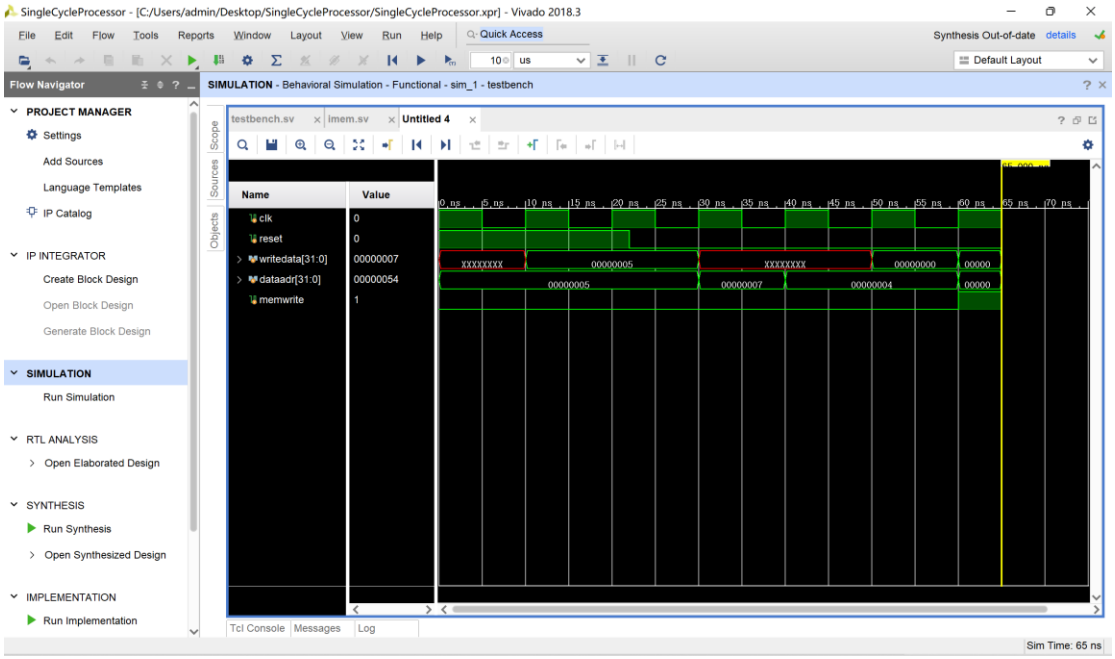
2.增加 andi，ori 和 bne 指令后，执行指令文件 memfile1.dat（课件 I/O 接口的测试代码）

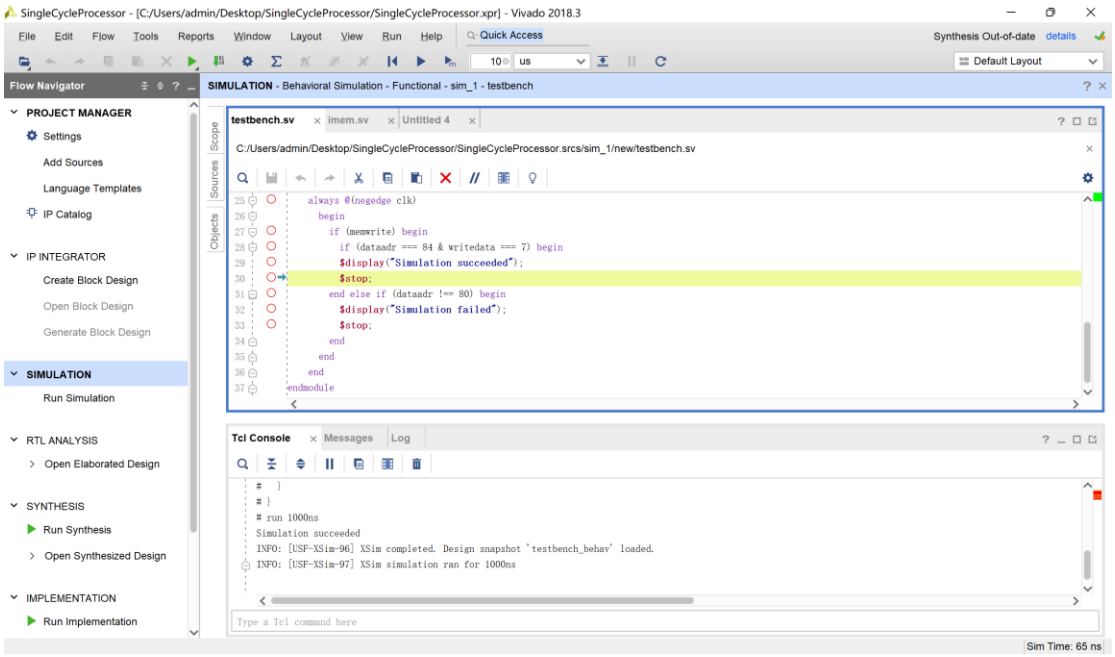
(1)memfile1.dat 内容

20020005
34440007
30450004
14a00001
20040001
ac040054

```
main:   addi $2, $0, 5       # initialize $2 = 5
        ori  $4, $2, 7       # 101 or  111 = 111 ($4)
        andi $5, $2, 4       # 101 and 100 = 100 ($5)
        bne  $5, $0, end     # should be taken
        addi $4, $0, 1       # shouldn't happen $4 = 1
end:    sw $4, 84($0)        # write mem[84] = 7
```

(2)仿真截图





3.增加 I/O 接口后，执行 memfile2.dat 文件（即课件中的 I/O 测试代码）

(1)memfile2.dat 内容

20100000
ac100080
8c110080
32320002

1240fffd
8c130088
8c14008c
0293a820
8c110080
32320001
1240fffd
ac150084
08000002

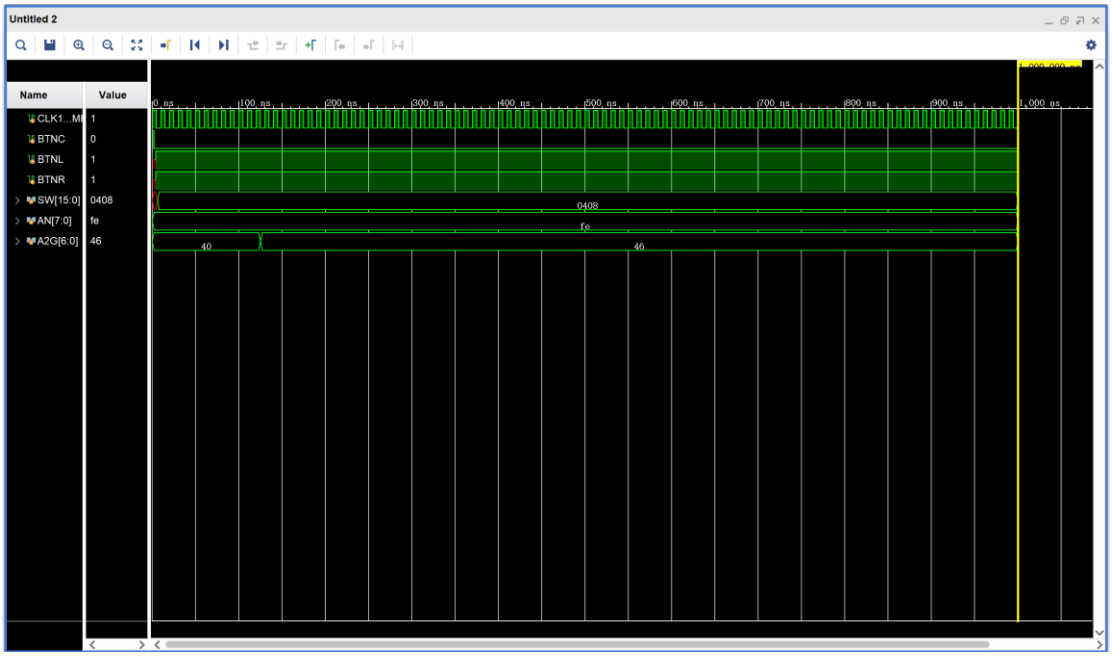
两个16进制数相加的测试汇编代码：

```
main:
    addi $s0, $0, 0
    sw   $s0, 0x80($0)
chkSwitch:
    lw   $s1, 0x80($0)
    andi $s2, $s1, 0x2
    beq  $s2, $0, chkSwitch
    lw   $s3, 0x88($0)
    lw   $s4, 0x8C($0)
    add  $s5, $s4, $s3
chkLED:
    lw   $s1, 0x80($0)
    andi $s2, $s1, 0x1
    beq  $s2, $0, chkLED
    sw   $s5, 0x84($0)
    j    chkSwitch
```

```
20100000
ac100080
8c110080
32320002
1240fffd
8c130088
8c14008c
0293a820
8c110080
32320001
1240fffd
ac150084
08000002
```

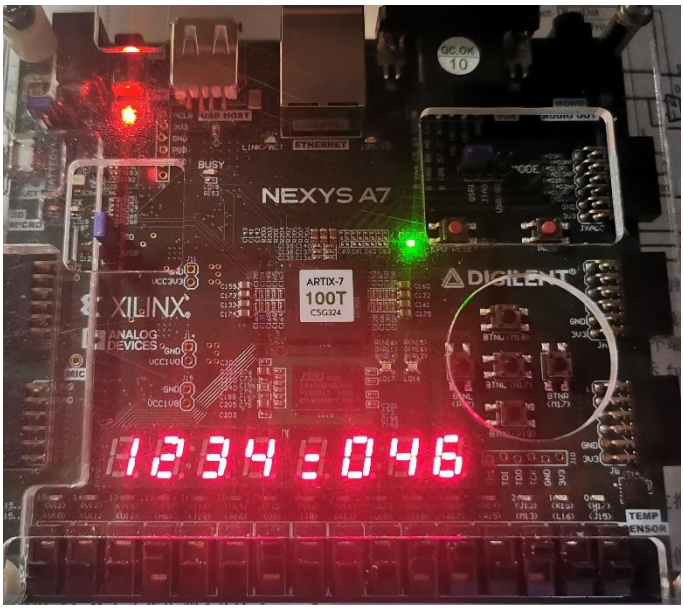
【注】还需要扩展andi指令

(2)仿真截图

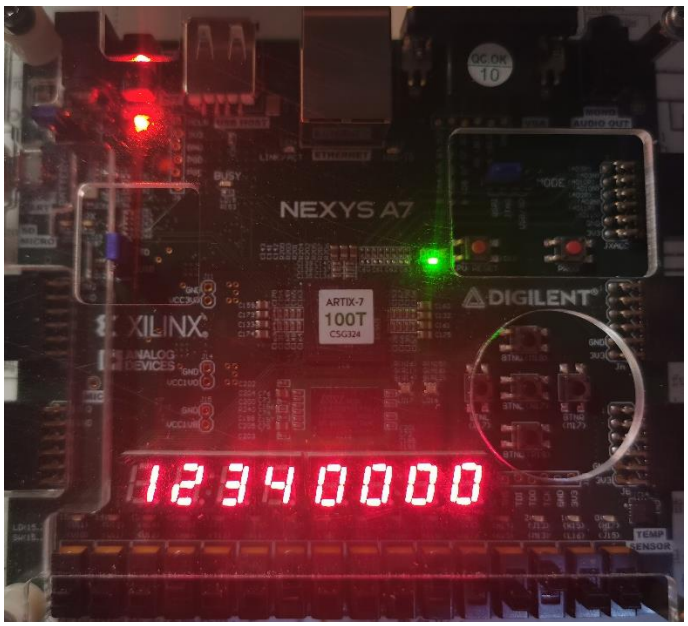


四、【实验开发板照片】

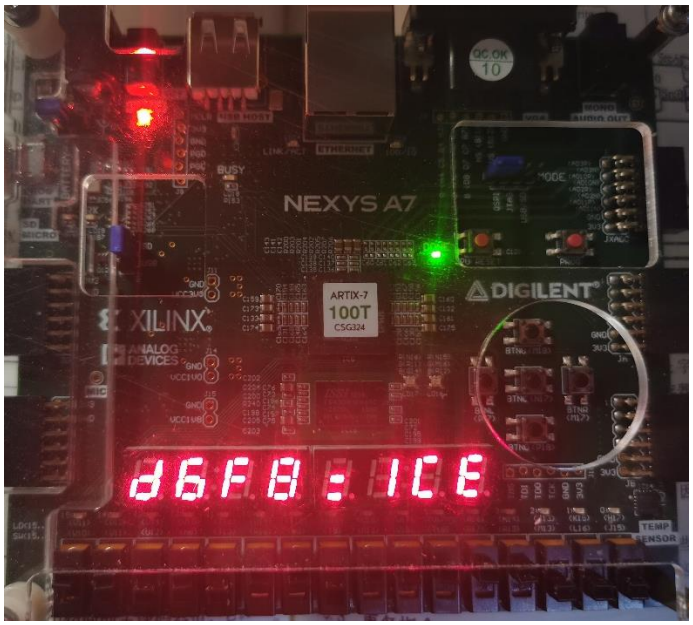
1：12+34=36



此时按下 BTNC 和 BTNL，清零后，等号也变回 0：


NEXYS A7
DIGILENT
XILINX
ARTIX-7
100T
12340000

2：d6+f8=1ce


NEXYS A7
DIGILENT
XILINX
ARTIX-7
100T
d6F8:1CE