中国海洋大学计算机科学与技术系

实验报告

姓名：岳宇轩　　年级：2019　　专业：19 慧与

科目：计算机组成原理　　题目：单周期 cpu

实验时间: 2021 年 5 月 20 日　　实验教师: 张巍

一、实验结果及截图分析：

（※代码挖空的部分必须截图或复制）

1. 代码补全

```
1   timescale 1ns / 1ps
2   //////////////////////////////////////////////////////////////////////////////////
3   //   > 文件名: single_cycle_cpu.v
4   //   > 描述   :单周期CPU模块，共实现16条指令
5   //   >         指令rom和数据ram均采用异步读数据，以便单周期CPU好实现
6   //   > 作者   : LOONGSON
7   //   > 日期   : 2016-04-14
8   //////////////////////////////////////////////////////////////////////////////////
9   define STARTADDR 32'd0  // 程序起始地址
10  module single_cycle_cpu(
11      input clk,     // 时钟
12      input resetn,  // 复位信号，低电平有效
13
14      //display data
15      input  [ 4:0] rf_addr,
16      input  [31:0] mem_addr,
17      output [31:0] rf_data,
18      output [31:0] mem_data,
19      output [31:0] cpu_pc,
20      output [31:0] cpu_inst
21      );
22
23  //------------------------------{取指}begin------------------------------//
24      reg  [31:0] pc;
25      wire [31:0] next_pc;
26      wire [31:0] seq_pc;
27      wire [31:0] jbr_target;
28      wire jbr_taken;
```

```
29
30          // 下一指令地址：seq_pc=pc+4
31          assign seq_pc[31:2]    = pc[31:2] + 1'b1;
32          assign seq_pc[1:0]     = pc[1:0];
33          // 新指令：若指令跳转，为跳转地址；否则为下一指令
34          assign next_pc = jbr_taken ? jbr_target : seq_pc;
35          always @ (posedge clk)    // PC程序计数器
36          begin
37              if (!resetn) begin
38                  pc <= `STARTADDR; // 复位，取程序起始地址
39              end
40              else begin
41                  pc <= next_pc;    // 不复位，取新指令
42              end
43          end
44
45          wire [31:0] inst_addr;
46          wire [31:0] inst;
47          assign inst_addr = pc;  // 指令地址：指令长度32位
48          inst_rom inst_rom_module(          // 指令存储器
49              .addr      (inst_addr[6:2]),   // I, 5,指令地址
50              .inst      (inst          )    // O, 32,指令
51          );
52          assign cpu_pc = pc;       //display pc
53          assign cpu_inst = inst;
54      //————————————————————————————{取指}end————————————————————————————//
```

```
54      //————————————————————————————{取指}end————————————————————————————//
55
56      //————————————————————————————{译码}begin————————————————————————————//
57          wire [5:0] op;
58          wire [4:0] rs;
59          wire [4:0] rt;
60          wire [4:0] rd;
61          wire [4:0] sa;
62          wire [5:0] funct;
63          wire [15:0] imm;
64          wire [15:0] offset;
65          wire [25:0] target;
66
67          assign op     = inst[31:26];  // 操作码
68          assign rs     = inst[25:21];  // 源操作数1
69          assign rt     = inst[20:16];  // 源操作数2
70          assign rd     = inst[15:11];  // 目标操作数
71          assign sa     = inst[10:6];   // 特殊域，可能存放偏移量
72          assign funct  = inst[5:0];    // 功能码
73          assign imm    = inst[15:0];   // 立即数
74          assign offset = inst[15:0];   // 地址偏移量
75          assign target = inst[25:0];   // 目标地址
76
```

```
77      wire op_zero;  // 操作码全0
78      wire sa_zero;  // sa域全0
79      assign op_zero = ~(|op);
80      assign sa_zero = ~(|sa);
81
82      // 实现指令列表
83      wire inst_ADDU, inst_SUBU , inst_SLT, inst_AND;
84      wire inst_NOR , inst_OR   , inst_XOR, inst_SLL;
85      wire inst_SRL , inst_ADDIU, inst_BEQ, inst_BNE;
86      wire inst_LW  , inst_SW   , inst_LUI, inst_J;
87
88
89      assign inst_ADDU  = op_zero & sa_zero    & (funct == 6'b100001);// 无符号加法
90
91      assign inst_SUBU  = op_zero & sa_zero    & (funct == 6'b100011);// 无符号减法
92      assign inst_SLT   = op_zero & sa_zero    & (funct == 6'b101010);// 小于则置位
93      assign inst_AND   = op_zero & sa_zero    & (funct == 6'b100100);// 逻辑与运算
94      assign inst_NOR   = op_zero & sa_zero    & (funct == 6'b100111);// 逻辑或非运算
95
96
97      assign inst_OR    = op_zero & (funct == 6'b100101);// 逻辑或运算
98      assign inst_XOR   = op_zero & (funct == 6'b100110);// 逻辑异或运算
99
99
100     assign inst_SLL   = op_zero & (rs==5'd0) & (funct == 6'b000000);// 逻辑左移
101     assign inst_SRL   = op_zero & (rs==5'd0) & (funct == 6'b000010);// 逻辑右移
102
103
104     assign inst_ADDIU = (op == 6'b001001);                  // 立即数无符号加法
105     assign inst_BEQ   = (op == 6'b000100);                  // 判断相等跳转
106     assign inst_BNE   = (op == 6'b000101);                  // 判断不等跳转
107
108
109     assign inst_LW    = (op == 6'b100011);                  // 从内存装载
110     assign inst_SW    = (op == 6'b101011);                  // 向内存存储
111     assign inst_LUI   = (op == 6'b001111);                  // 立即数装载高半字节
112     assign inst_J     = (op == 6'b000010);                  // 直接跳转
113
114     // 无条件跳转判断
115     wire        j_taken;
116     wire [31:0] j_target;
117     assign j_taken  = inst_J;
118
119
120     // 无条件跳转目标地址：PC={PC[31:28],target<<2}
121     assign j_target = {pc[31:28], target, 2'b00};
```

```
122
123         //分支跳转
124         wire        beq_taken;
125         wire        bne_taken;
126         wire [31:0] br_target;
127         assign beq_taken = (rs_value == rt_value);        // BEQ跳转条件：GPR[rs]=GPR[rt]
128         assign bne_taken = ~beq_taken;                    // BNE跳转条件：GPR[rs]≠GPR[rt]
129         assign br_target[31:2] = pc[31:2] + {{14{offset[15]}}, offset};
130         assign br_target[1:0]  = pc[1:0];    // 分支跳转目标地址：PC=PC+offset<<2
131
132         //跳转指令的跳转信号和跳转目标地址
133         assign jbr_taken = j_taken              // 指令跳转：无条件跳转 或 满足分支跳转条件
134                          | inst_BEQ & beq_taken
135                          | inst_BNE & bne_taken;
136         assign jbr_target = j_taken ? j_target : br_target;
137
138         // 寄存器堆
139         wire rf_wen;
140         wire [4:0] rf_waddr;
141         wire [31:0] rf_wdata;
142         wire [31:0] rs_value, rt_value;
143
144         regfile rf_module(
145             .clk    (clk      ),   // I, 1
146             .wen    (rf_wen   ),   // I, 1
147             .raddr1 (rs       ),   // I, 5
148             .raddr2 (rt       ),   // I, 5
149             .waddr  (rf_waddr ),   // I, 5
150             .wdata  (rf_wdata ),   // I, 32
151             .rdata1 (rs_value ),   // O, 32
152             .rdata2 (rt_value ),    // O, 32
153
154             //display rf
155             .test_addr(rf_addr),
156             .test_data(rf_data)
157         );
158
```

```
160     // 传递到执行模块的ALU源操作数和操作码
161     wire inst_add, inst_sub, inst_slt,inst_sltu;
162     wire inst_and, inst_nor, inst_or, inst_xor;
163     wire inst_sll, inst_srl, inst_sra,inst_lui;
164     assign inst_add = inst_ADDU | inst_ADDIU | inst_LW | inst_SW; // 做加法运算指令
165     assign inst_sub = inst_SUBU; // 减法
166     assign inst_slt = inst_SLT;  // 小于置位
167     assign inst_sltu= 1'b0;      // 暂未实现
168     assign inst_and = inst_AND;  // 逻辑与
169     assign inst_nor = inst_NOR;  // 逻辑或非
170     assign inst_or  = inst_OR;   // 逻辑或
171     assign inst_xor = inst_XOR;  // 逻辑异或
172     assign inst_sll = inst_SLL;  // 逻辑左移
173     assign inst_srl = inst_SRL;  // 逻辑右移
174     assign inst_sra = 1'b0;      // 暂未实现
175     assign inst_lui = inst_LUI;  // 立即数装载高位
176
177     wire [31:0] sext_imm;
178     wire   inst_shf_sa;    //使用sa域作为偏移量的指令
179     wire   inst_imm_sign;  //对立即数作符号扩展的指令
180
181
182     assign sext_imm      = imm[15] ? {16'b1111111111111111,imm} : {16'b0000000000000000,imm};// 立即数符号扩展
183
184     assign inst_shf_sa   = inst_SLL | inst_SRL;
185     assign inst_imm_sign = inst_ADDIU | inst_LUI | inst_LW | inst_SW;
186
```

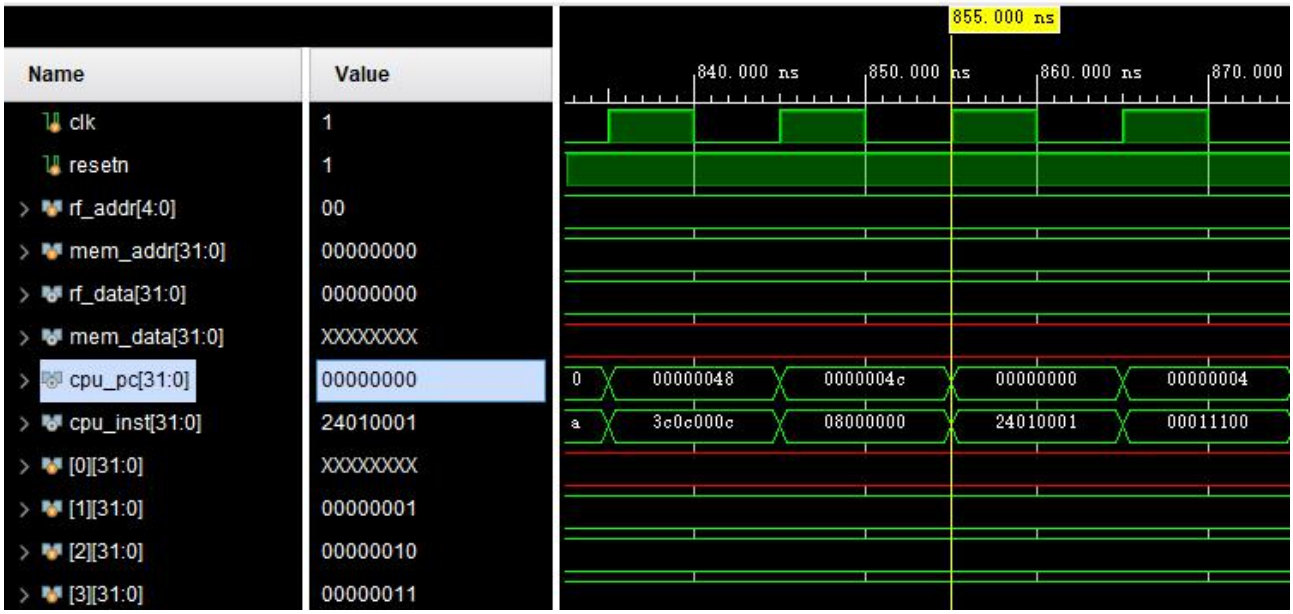
```
187     wire [31:0] alu_operand1;
188     wire [31:0] alu_operand2;
189     wire [11:0] alu_control;
190     assign alu_operand1 = inst_shf_sa ? {27'd0,sa} : rs_value;
191     assign alu_operand2 = inst_imm_sign ? sext_imm : rt_value;
192     assign alu_control = {inst_add,         // ALU操作码，独热编码
193                           inst_sub,
194                           inst_slt,
195                           inst_sltu,
196                           inst_and,
197                           inst_nor,
198                           inst_or,
199                           inst_xor,
200                           inst_sll,
201                           inst_srl,
202                           inst_sra,
203                           inst_lui};
204 //--------------------------------{译码}end--------------------------------//
205
206 //--------------------------------{执行}begin------------------------------//
207     wire [31:0] alu_result;
208
```

```
209         alu alu_module(
210             .alu_control  (alu_control ),  // I, 12, ALU控制信号
211             .alu_src1     (alu_operand1),  // I, 32, ALU操作数1
212             .alu_src2     (alu_operand2),  // I, 32, ALU操作数2
213             .alu_result   (alu_result  )   // O, 32, ALU结果
214         );
215     //------------------------------------{执行}end------------------------------------//
216
217     //------------------------------------{访存}begin----------------------------------//
218         wire [3 :0] dm_wen;
219         wire [31:0] dm_addr;
220         wire [31:0] dm_wdata;
221         wire [31:0] dm_rdata;
222         assign dm_wen   = {4{inst_SW}} & {4{resetn}};    // 内存写使能,非resetn状态下有效
223
224
225         assign dm_addr  = alu_result;                // 内存写地址，为ALU结果 *****
226         assign dm_wdata = rt_value;                  // 内存写数据，为rt寄存器值 *****
227         data_ram data_ram_module(
228             .clk   (clk         ),  // I, 1,  时钟
229             .wen   (dm_wen      ),  // I, 1,  写使能
230             .addr  (dm_addr[6:2]),  // I, 32, 读地址
231             .wdata (dm_wdata    ),  // I, 32, 写数据
232             .rdata (dm_rdata    ),  // O, 32, 读数据
233
234             //display mem
235             .test_addr(mem_addr[6:2]),
236             .test_data(mem_data     )
237         );
238     //------------------------------------{访存}end------------------------------------//
239
240     //------------------------------------{写回}begin----------------------------------//
241         wire inst_wdest_rt;   // 寄存器堆写入地址为rt的指令
242         wire inst_wdest_rd;   // 寄存器堆写入地址为rd的指令
243         assign inst_wdest_rt = inst_ADDIU | inst_LW | inst_LUI;
244         assign inst_wdest_rd = inst_ADDU | inst_SUBU | inst_SLT | inst_AND | inst_NOR
245                              | inst_OR   | inst_XOR  | inst_SLL | inst_SRL;
246         // 寄存器堆写使能信号，非复位状态下有效
247         assign rf_wen   = (inst_wdest_rt | inst_wdest_rd) & resetn;
248         assign rf_waddr = inst_wdest_rd ? rd : rt;        // 寄存器堆写地址rd或rt
249         assign rf_wdata = inst_LW ? dm_rdata : alu_result;// 写回结果，为load结果或ALU结果
250     //------------------------------------{写回}end------------------------------------//
251     endmodule
252
```

2. 仿真图像

为了方便看实验结果，我把所有寄存器也显示了

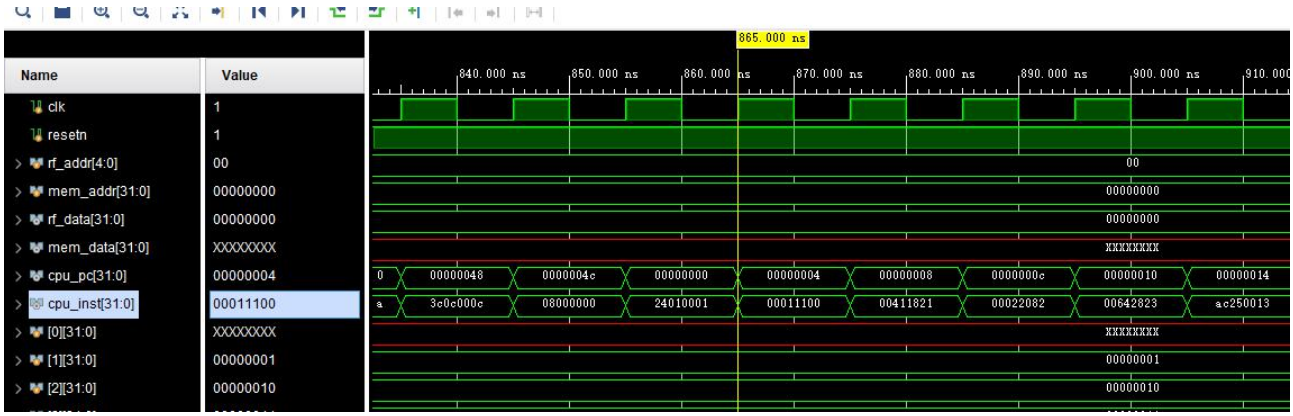
00H




```
assign inst_rom[ 0] = 32'h24010001; // 00H: addiu $1 ,$0,#1    | $1 = 0000_0001H
```

从指令寄存器中取出第一条指令，其指令编码为 24010001h, 指令地址为 00h,对指令进行汇编得到汇编指令 addiu $1 ,$0,#1 ，指令的操作是零号寄存器（0）无符号加 立即数 1，结果送 1 号寄存器。 运行结果为 1 号寄存器值变为 1H，通过仿真图像观察，结果正确

04H

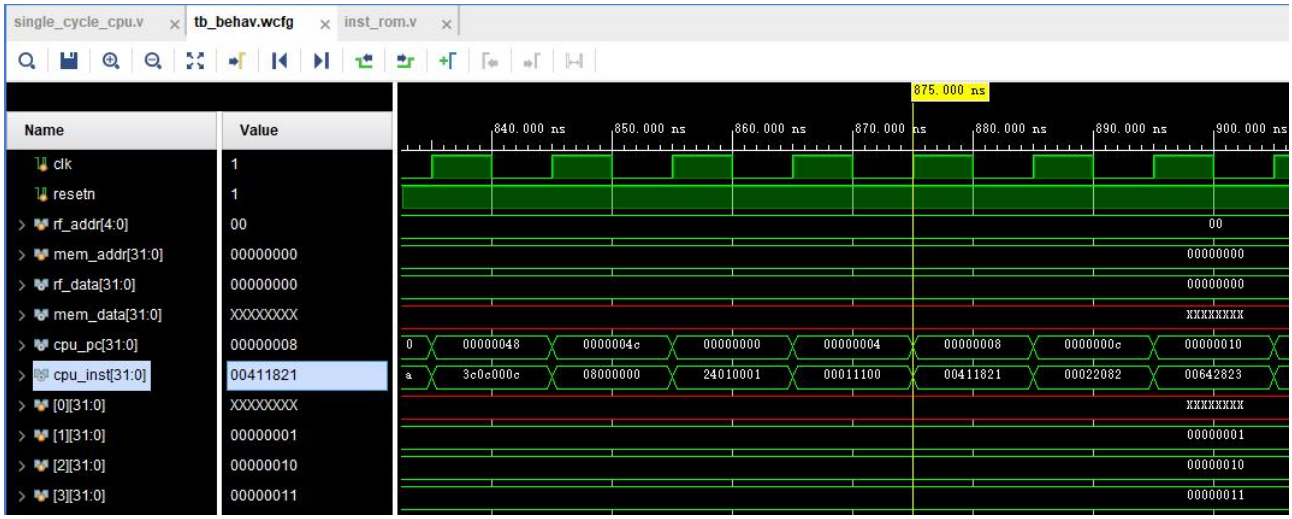



```
assign inst_rom[ 1] = 32'h00011100; // 04H: sll    $2 ,$1,#4    | $2 = 0000_0010H
```

第二条指令地址为 04H，从 IR 中取出的指令编码为 00011100H，对应的汇编指令为 sll $2 ,$1,#4，指令

操作为 1 号寄存器中的值逻辑左移 4 位后送 2 号寄存器。01H 左移 4 位变为 10H，指令结果为 2 号寄存器值变为 10H 。 通过仿真图像观察实验结果正确。

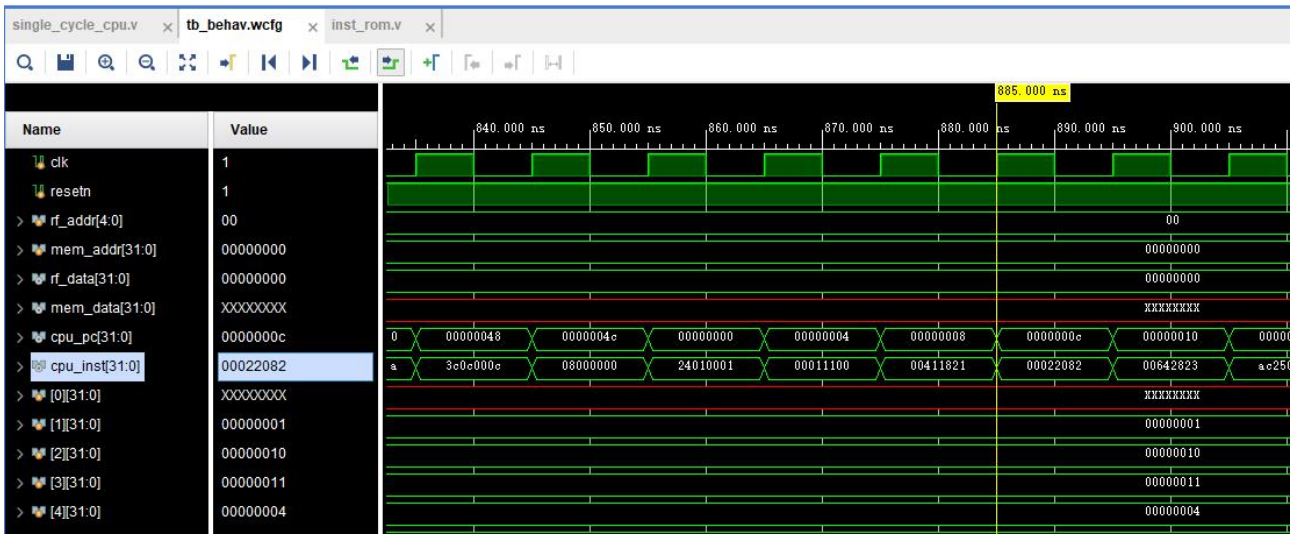
08H




```
assign inst_rom[ 2] = 32'h00411821; // 08H: addu  $3 ,$2,$1    | $3 = 0000_0011H
```

下一条指令的地址为 08H， 取出的指令编码为 00411821H，对应的汇编指令为 addu $3 ,$2,$1，指令操作是将 2 号和 1 号寄存器中的值无符号相加，结果送 3 号寄存器。01H+10H，结果为 11H，通过仿真图像可以看到 3 号寄存器值为 11H。

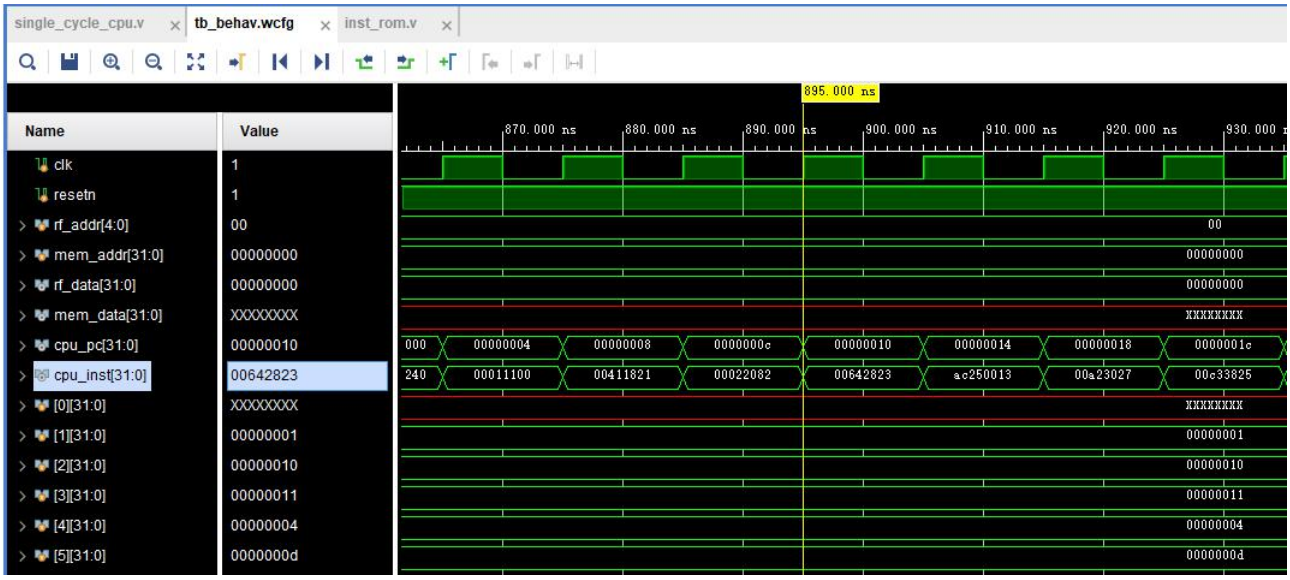
0CH




```
assign inst_rom[ 3] = 32'h00022082; // 0CH: srl   $4 ,$2,#2    | $4 = 0000_0004H
```

指令地址为 0CH，指令编码为 00022082，对应汇编指令为 srl $4 ,$2,#2 ，将 2 号寄存器中的值逻辑右移两位后放入四号寄存器。通过仿真图像可以看到，2 号寄存器中的值为 10H，即 0001 0000，逻辑右移 2 位后为 0000 0100，即 04H，04H 送入 4 号寄存器，实验结果正确。

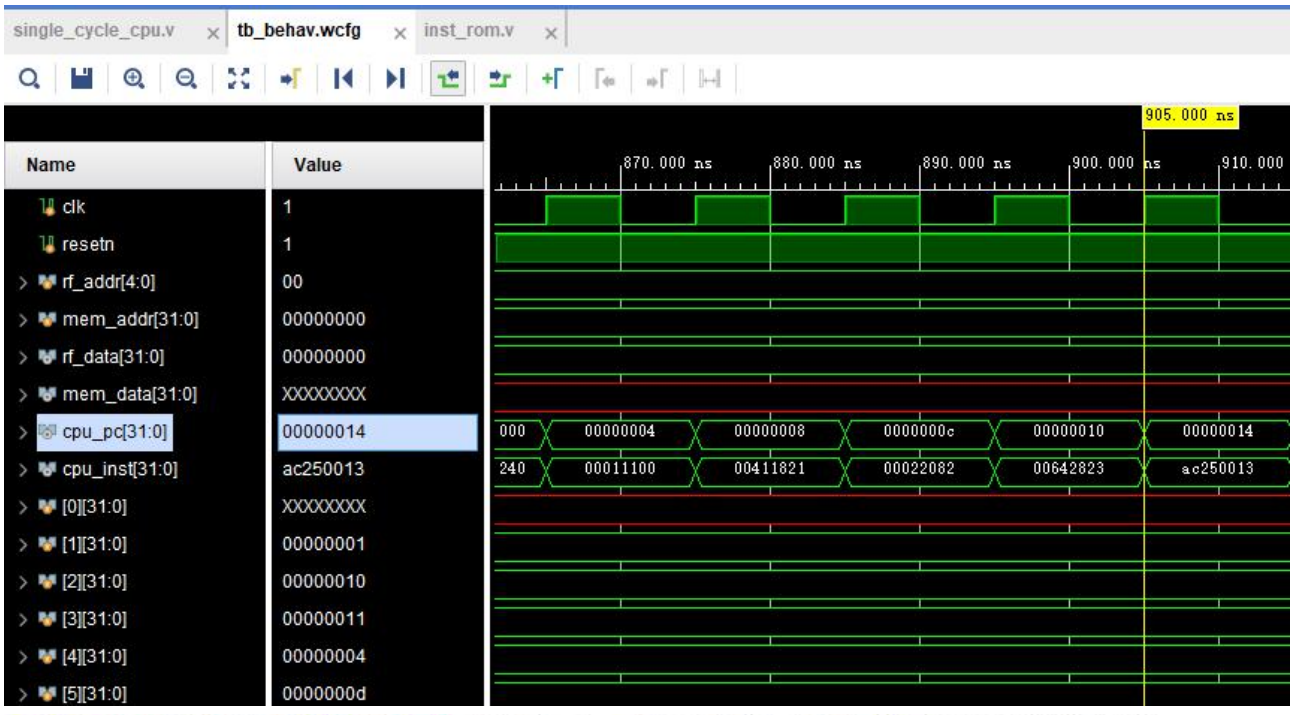
10H




```
assign inst_rom[ 4] = 32'h00642823; // 10H: subu  $5 ,$3,$4    | $5 = 0000_000DH
```

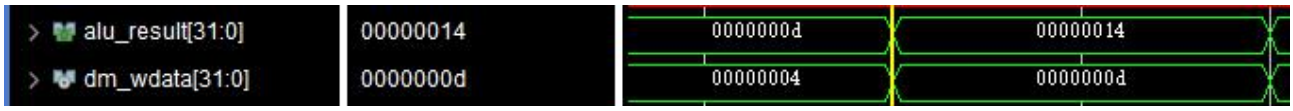
PC+4 来到了 10H，取出指令的编码为 00642823，对应汇编指令为 subu　$5 ,$3,$4 ，指令操作为 3 号寄存器中的数无符号减 4 号寄存器中的数，结果送 5 号寄存器。从仿真图像中可以看出，3 号和 4 号寄存器值分别为 11H 和 04H，结果为 0DH，送 5 号寄存器，5 号寄存器值为 0d，结果正确

14H



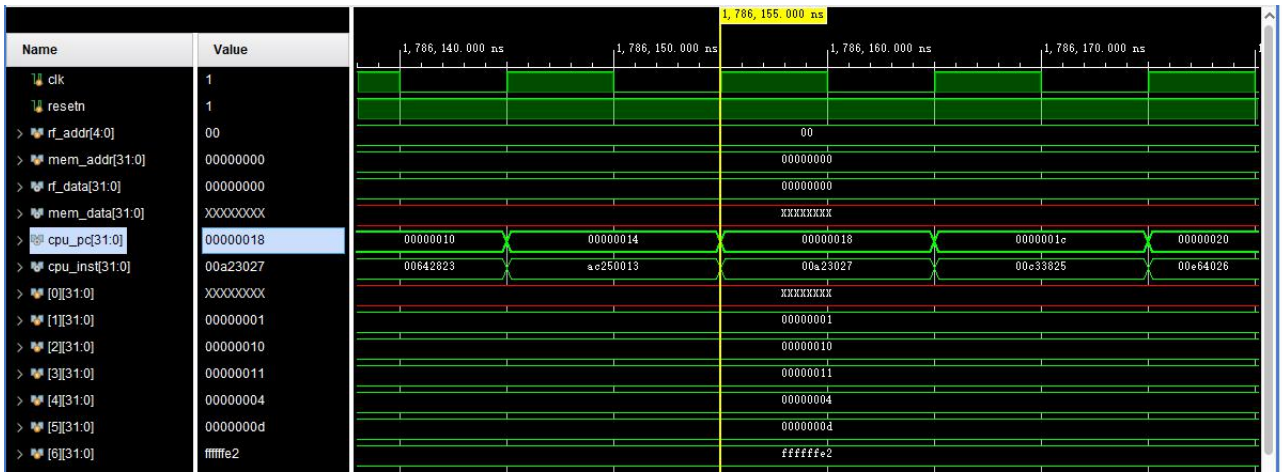

```
assign inst_rom[ 5] = 32'hAC250013; // 14H: sw    $5 ,#19($1)  | Mem[0000_0014H] = 0000_000DH
```

指令编码为 AC250013，对应汇编指令为 sw　　$5 ,#19($1)，进行操作为：寄存器 1 中数+19 作为访存地址，将 5 号寄存器中的数据加载进该地址的内存中



alu 运算结果为访存地址 14H，dm_wdata 是要写入的数据，与 5 号寄存器中数据一致，实验结果正确

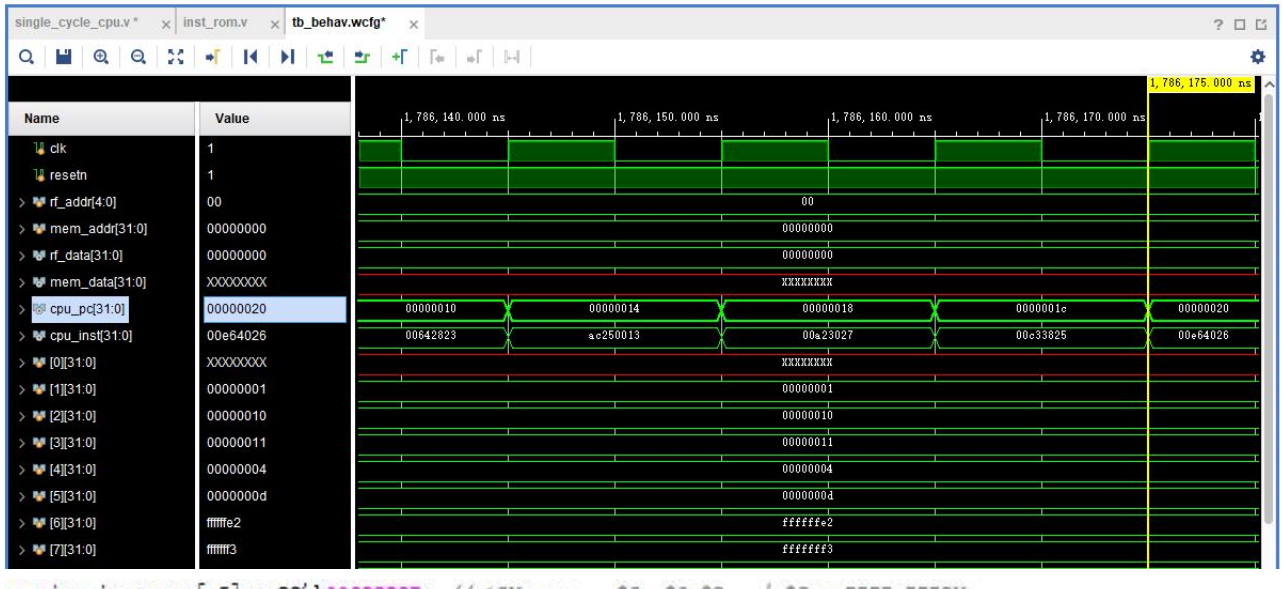
18H




```
assign inst_rom[ 6] = 32'h00A23027; // 18H: nor   $6 ,$5,$2   | $6 = FFFF_FFE2H
```

指令编码 00A23027H，对应汇编指令为 nor $6 ,$5,$2 。寄存器 5 和 2 中的值分别为 0000 000dH 和 0000 0010H，进行或非运算结果为 ffff ffe2H,放入 6 号寄存器中。实验结果正确。

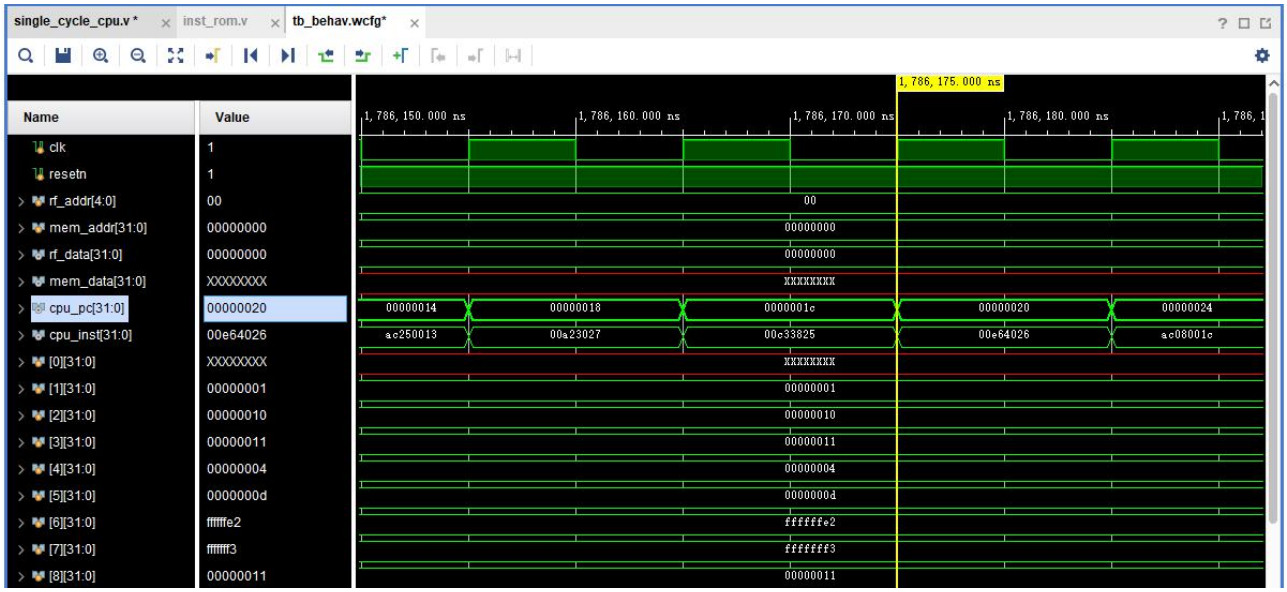
1CH




```
assign inst_rom[ 7] = 32'h00C33825; // 1CH: or    $7 ,$6,$3   | $7 = FFFF_FFF3H
```

指令编码为 00C33825H，对应汇编指令为 or $7 ,$6,$3 ，寄存器 6 和 3 进行或操作，即 ffff ffe2H 与 0000 0011H 进行或操作，结果为 ffff fff3 H,放入 7 号寄存器，7 号寄存器为 ffff fff3H，结果正确。

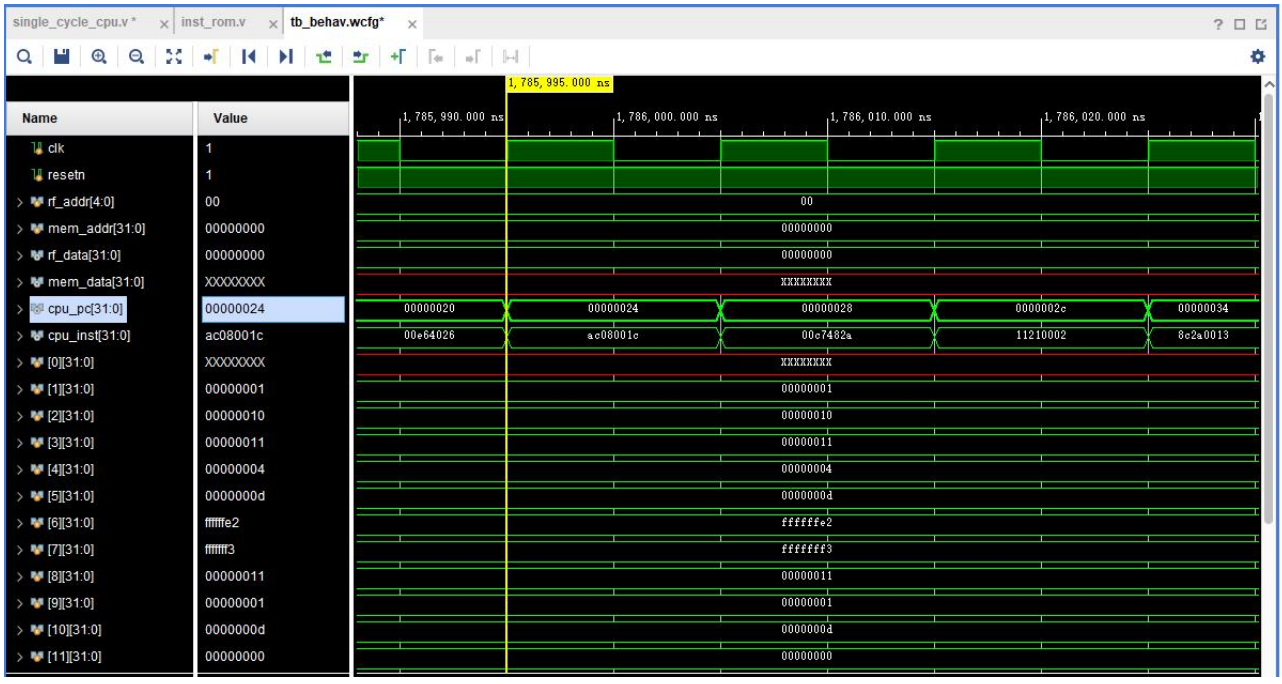
20H




```
assign inst_rom[ 8] = 32'h00E64026; // 20H: xor   $8 ,$7,$6   | $8 = 0000_0011H
```

指令编码 00E64026H,对应汇编 xor $8 ,$7,$6 ，寄存器 7 和 6 中内容（fffffff3 fffffe2）进行异或操作，得到结果为 0000 0011H，放入 8 号寄存器

24H

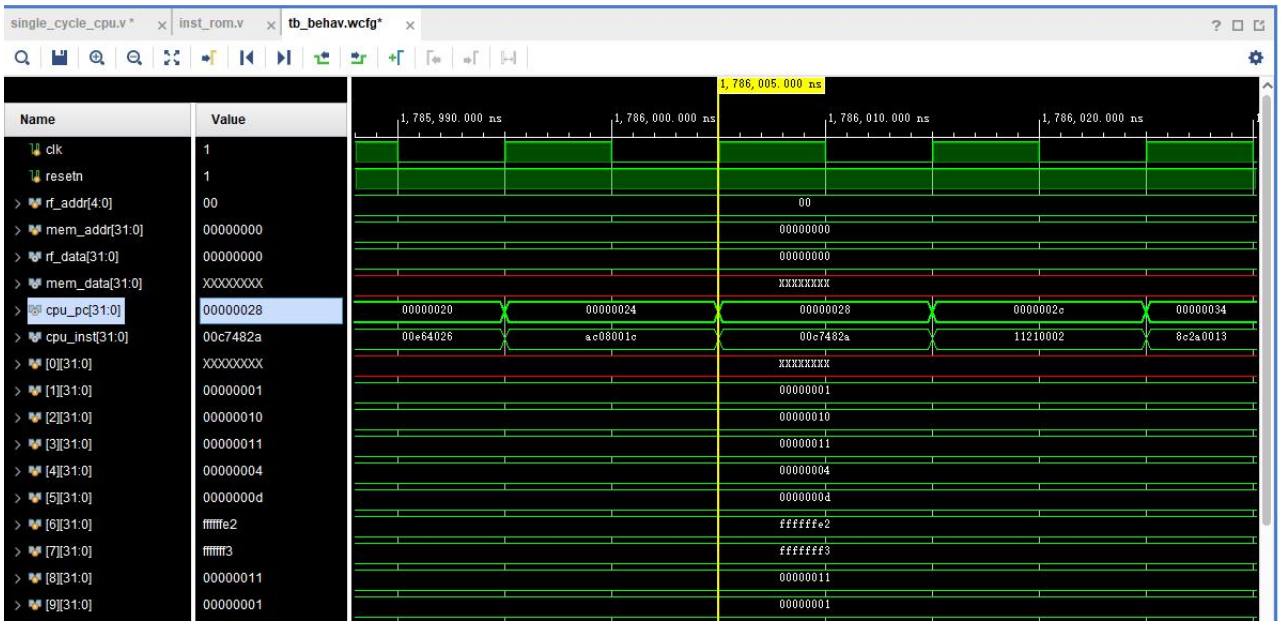



```
assign inst_rom[ 9] = 32'hAC08001C; // 24H: sw    $8 ,#28($0) | Mem[0000_001CH] = 0000_0011H
```

指令编码 AC08001C,对应汇编 sw $8 ,#28($0)，将 8 号寄存器中的内容放入内存中地址为 28 处

alu_result 为内存地址计算，0+28 = 28，即 1cH,要写入的数据是 8 号寄存器的值 0000 0011H

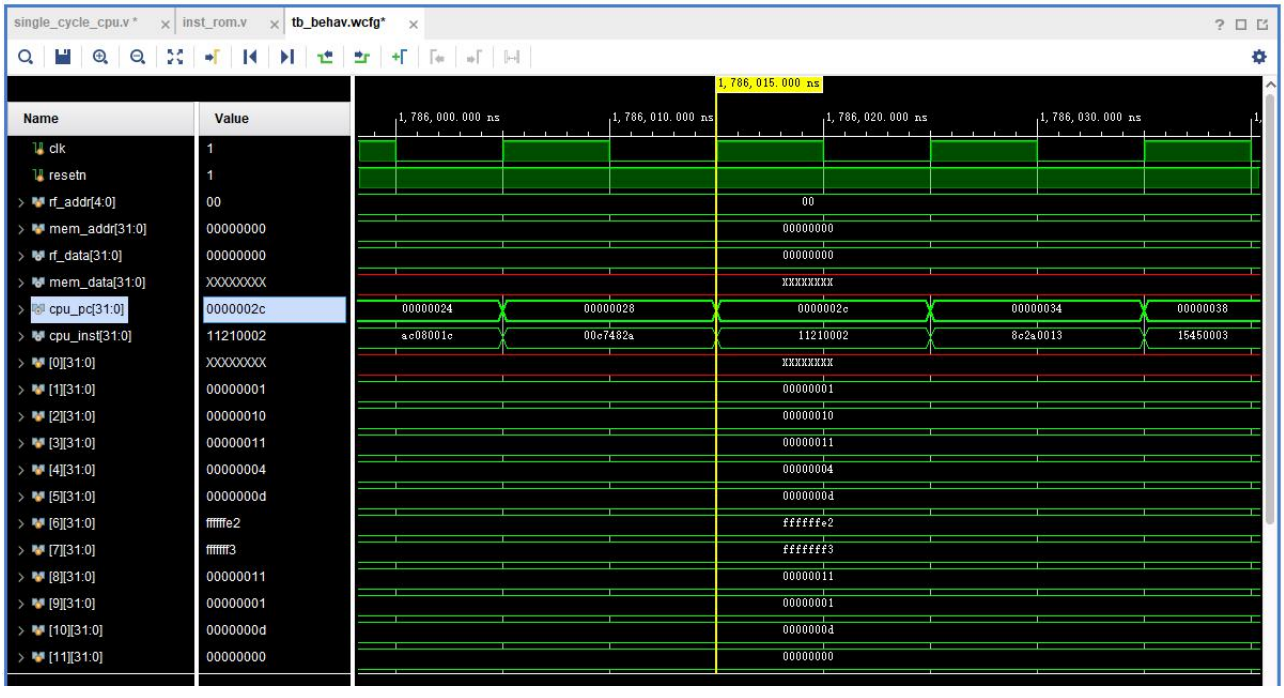
28H




```
assign inst_rom[10] = 32'h00C7482A; // 28H: slt   $9 ,$6,$7   | $9 = 0000_0001H
```

指令编码为 00c7482aH，对应汇编为 slt　$9 ,$6,$7 ，指令操作为小于置位。寄存器 6 中值 ffff ffe2 减寄存器 7 中值 ffff fff3，结果小于 0，将 1 存入 rd 中（9 号寄存器）

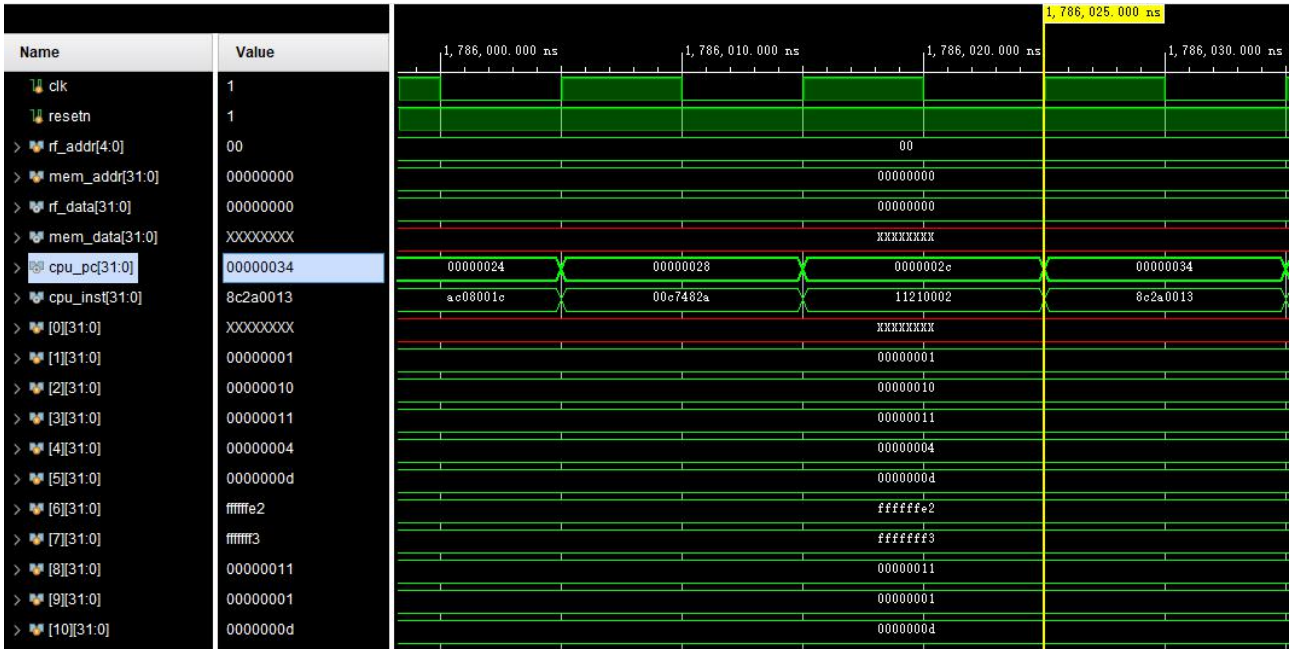
2cH




```
assign inst_rom[11] = 32'h11210002; // 2CH: beq   $9 ,$1,#2   | 跳转到指令34H
```

指令编码 11210002H，对应汇编 beq　$9 ,$1,#2。执行的操作为：9 号寄存器中的值 01H 与 1 号寄存器中的值 01H 比较，相等，跳转到 34H（2cH + 两个 4）

34H



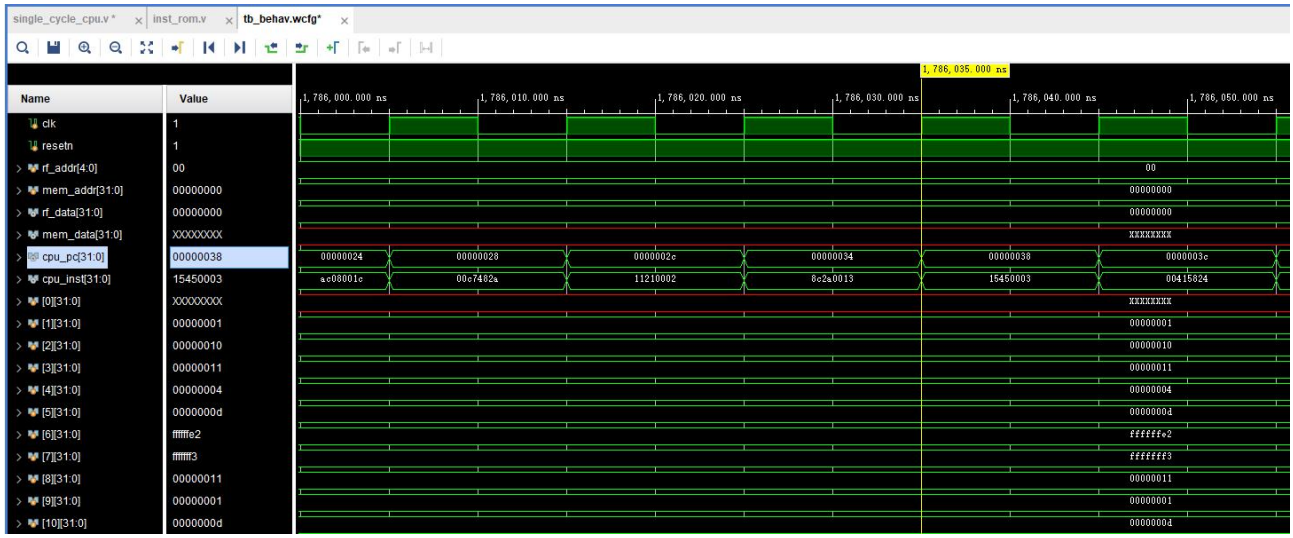

```
assign inst_rom[13] = 32'h8C2A0013; // 34H: lw    $10,#19($1) | $10 = 0000_000DH
```

指令编码为 8C2A0013，对应汇编指令为 lw　$10,#19($1)，1 号寄存器中内容+19 后（14H）作为访存地址，读取内存中数据放入 10 号寄存器

1 号寄存器中内容为 01H，加 19 后是 20，也就是 14H，

通过仿真图像可以看出，alu 计算方寸地址为 14H，读出数据为 0dH，结果正确。

38H

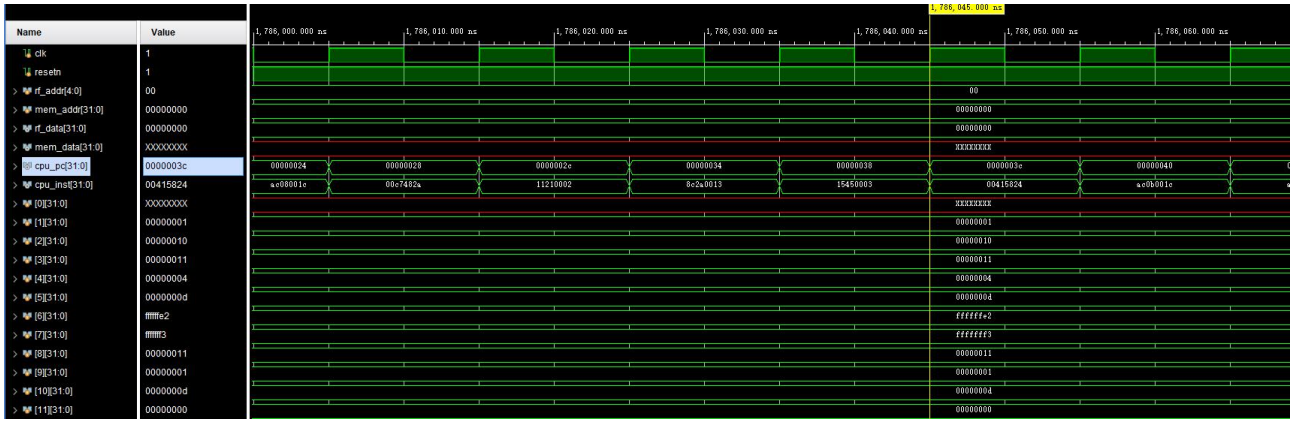


```
assign inst_rom[14] = 32'h15450003; // 38H: bne   $10,$5,#3   | 不跳转
```

PC+4 后指令地址为 38H，取出的指令编码为 15450003H,对应的汇编代码为 bne　$10,$5,#3　，指令执行功能为比较寄存器 10 和 5 中内容，不相等则发生跳转。

通过仿真图像可以看出寄存器 10 和 5 中的内容都是 0dH，故不发生跳转，下一条指令地址应为 PC+4，也就是 3CH

3CH




```
assign inst_rom[15] = 32'h00415824; // 3CH: and   $11,$2,$1   | $11 = 0000_0000H
```

指令编码为 00415824H，对应汇编指令为 and　$11,$2,$1，对 2 号寄存器和 1 号寄存器中的内容进行逻辑与，结果送 11 号寄存器，结果 11 号寄存器中值为 0000 000H，观察仿真图像，结果正确

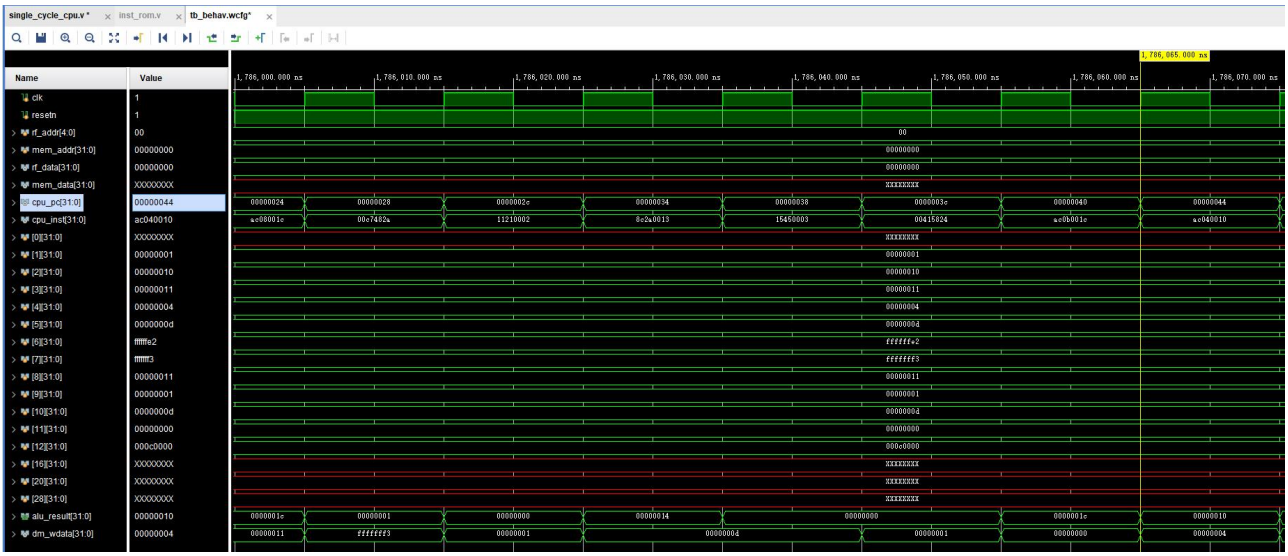
40H


```
assign inst_rom[16] = 32'hAC0B001C; // 40H: sw   $11,#28($0)   | Men[0000_001CH] = 0000_0000H
```

指令编码为 AC0B001C，对应汇编指令为 sw　$11,#28($0)，将 11 号寄存器中的内容加载进 0+28 处地址的内存中

通过上图可以看到 alu 运算地址结果为 1c，dm 值为 11 号寄存器中的内容 0000 0000H，结果正确

44H

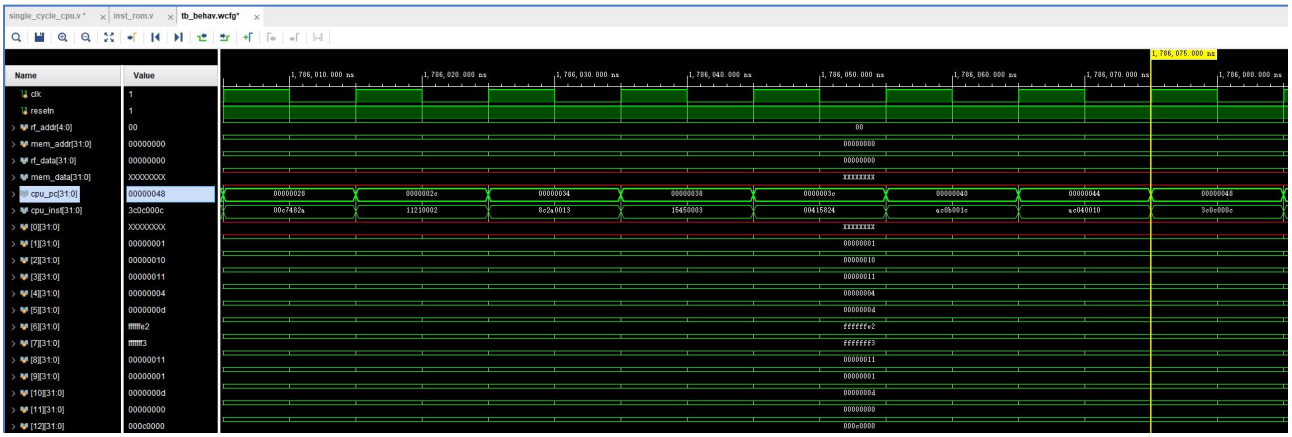


```
assign inst_rom[17] = 32'hAC040010; // 44H: sw    $4 ,#16($0) | Mem[0000_0010H] = 0000_0004H
```

指令编码为 AC040010H,对应汇编指令为 sw $4 ,#16($0)，加载 4 号寄存器中的内容进入内存地址为 16 处。

通过仿真图像可以看到 4 号寄存器中值为 0004H，alu 运算访存地址为 10H，写入数据的值 dm_wdata 为 0004H。结果正确

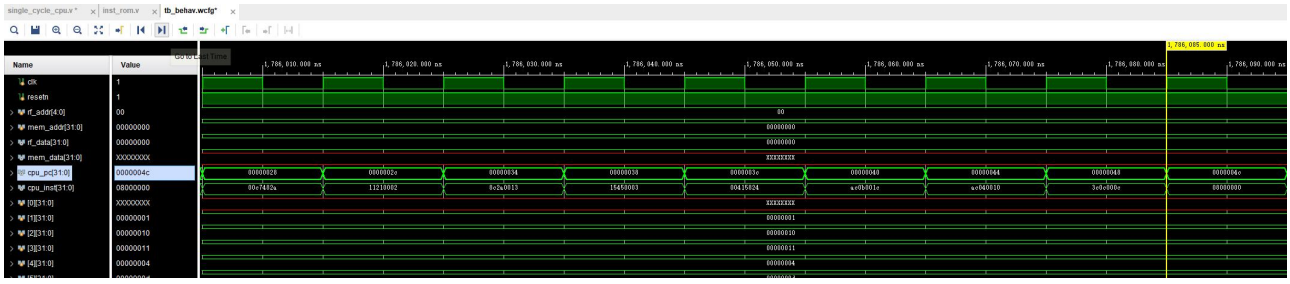
48H



```
assign inst_rom[18] = 32'h3C0C000C; // 48H: lui   $12,#12     | [R12] = 000C_0000H
```

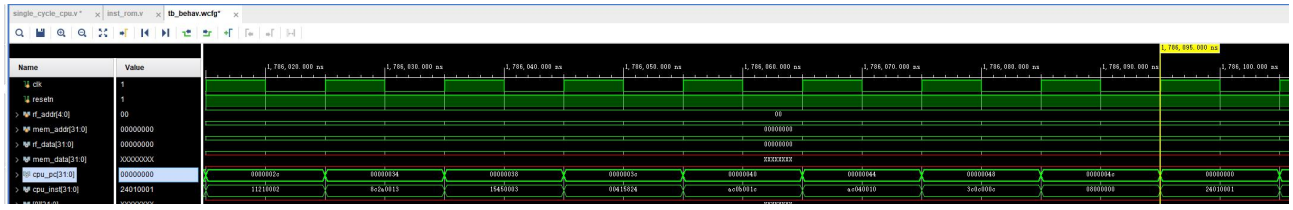
指令编码为 3C0C000CH,对应汇编指令为 lui $12,#12 ,指令功能为高位加载，将立即数 12 加载进 12 号寄存器的高位，地位填充 0，故 12 号寄存器结果应为 000C 0000H，观察仿真图像，结果正确

4CH



```
assign inst_rom[19] = 32'h08000000; // 4CH: j      00H          | 跳转指令00H
```

指令编码为 0800 0000H，对应汇编指令为　j 00H,　指令结果为将 00H 送入 PC，无条件跳转到 00H 处。



点击下一个时钟周期，发现来到了 00H 处，实验结果正确。

实验总结

通过本次实验，我更加清楚的了解了单周期 cpu 运作的原理。

首先对指令的来源，有三种方式，分别是 pc+4,分支和跳转，要予以区分。顺序执行 pc = pc+4，分支要判断是 beq 还是 bne，跳转为 PC=PC+offset<<2，无条件跳转为 PC={PC[31:28],target<<2}

根据指令不同位置取出不同的数据，操作码是 31 到 26，rs 是 25 到 21，rt 是 20 到 16，rd 是 15 到 11，特殊域 10 到 6，功能码 5 到 0，立即数 15 到 0，地址偏移量 15 到 0，j 指令目标地址 25 到 0.

要根据 op 和 funct 等信息确定指令是执行的哪种操作，并根据操作类型生成 alu 控制信号。对立即数进行扩展，进行 alu 运算。

访存阶段要确定内存写使能，非 resetn 状态下有效。内存写的地

址为 alu 运算结果，数据为 rt 寄存器值。写回阶段要判断是写入 rt 还是 rd，是写入 alu 运算结果还是 load 结果。

设计单周期的 cpu 不仅要对指令熟悉掌握，更要清清楚楚的理解好单周期 cpu 的运作流程，以及每种操作对应的控制信号序列。