

**计算机组成原理实验报告**

姓 名：刘嘉伟

专 业：计算机创新实验18-1班

班 级：1班

学 号：2018213106

指导教师：阙夏

完成日期：2020年6月26日

# 实验一、Verilog与ModelSim基础

## 实验目的：

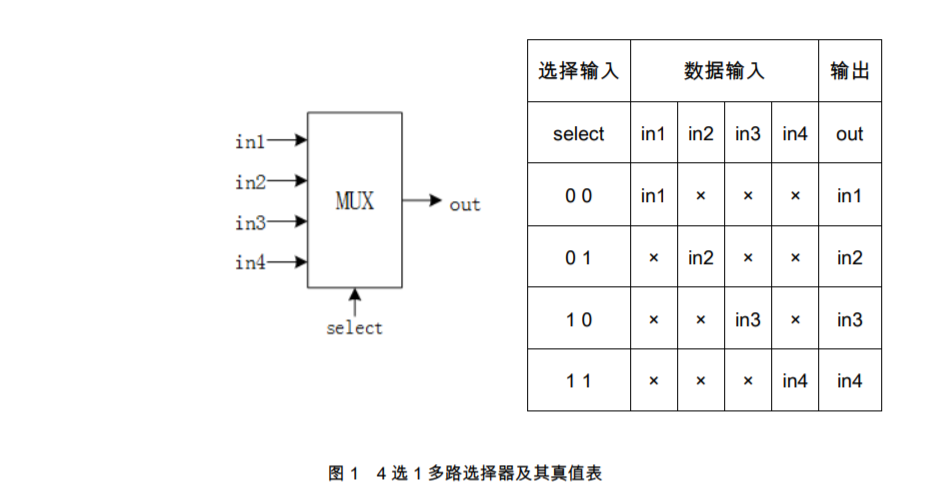
熟悉并掌握 Verilog HDL 与 ModelSim 的使用

## 二、实验内容

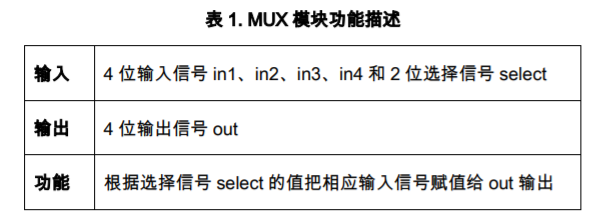
学习使用 Verilog 完成 4 选 1 多路选择器的设计和实现,并使用 ModelSim 工具对设计进 行仿真和分析验证。

## 三、实验原理

多路选择器（MUX）是一种在多路数据传送过程中，能够根据需要将其中任意一路选 出来的电路，其原理图和真值表如下图所示。



## 四、实验步骤



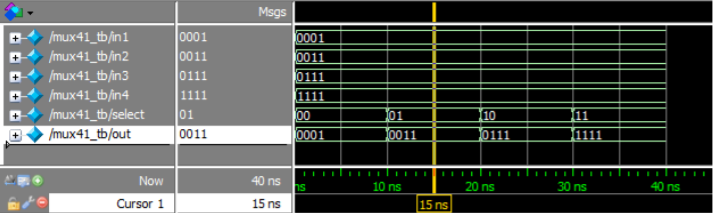
MUX模块的Verilog代码：

|  |
| --- |
| module mux41 (  input wire [3:0] in1, in2, in3, in4,  input wire [1:0] select,  output reg [3:0] out  );  always@\* begin  case (select)  2'b00: out = in1;  2'b01: out = in2;  2'b10: out = in3;  2'b11: out = in4;  default: out = 4'bx;  endcase  end  endmodule |

TestBench代码：

|  |
| --- |
| module mux41\_tb;  reg [3:0] in1, in2, in3, in4;  reg [1:0] select;  wire [3:0] out;  initial begin  in1 = 4'b0001;  in2 = 4'b0011;  in3 = 4'b0111;  in4 = 4'b1111;  select = 2'b00;  #10 select = 2'b01;  #10 select = 2'b10;  #10 select = 2'b11;  #10 $stop;  end  mux41 uut(  .in1 (in1) , .in2 (in2) , .in3 (in3), .in4 (in4), .select(select),  .out (out)  );  endmodule |

实验结果：



## 五、总结

之前已经做过数字逻辑的Verilog实验，所以对Verilog已经比较熟悉，实验一就当是熟悉一下操作流程。

# 实验二、CPU 部件实现之 ALU 和寄存器堆

## 实验目的：

理解和掌握 CPU 中的算术逻辑运算部件（ALU）和寄存器堆（Register File）的工作 原理，并使用 Verilog 和 ModelSim 进行设计和仿真。

## 二、实验内容

1. 使用 Verilog 完成 ALU 的设计,并编写测试仿真文件验证其正确性。要求：

* ALU 支持 16 位的加、减、与、或以及移位运算。

2. 使用 Verilog 完成通用寄存器堆的设计，并编写测试仿真文件验证其正确性。要求

* 寄存器堆包含 8 个 16 位的寄存器；
* 寄存器堆有两个读端口和一个写端口。

## 三、实验原理

ALU：ALU运算需要两个操作数A与B，需要进行多种运算，那么就需要一个判断运算符的输入信号，要求完成五种运算，那么则需要三位信号才能表示五种运算，先设定好每种运算对应的信号，输入运算数和信号就能输出运算结果。

ALU

op

A

B

out

寄存器堆：寄存器堆要求两个读端口和一个写端口，既然需要写入数据，那么就需要时钟周期来控制写入的时间，输入信号包括两个读端口的地址，和写端口的位置与值，输出读端口所读出的数据即可。

regFile

r1

r2

w1

wval

CLK

out1

out2

## 四、实验步骤

ALU：

|  |  |
| --- | --- |
| 输入 | 16位运算数A,B,操作符op |
| 输出 | 运算结果out |
| 功能 | 将A数或A数与B数进行op运算 |

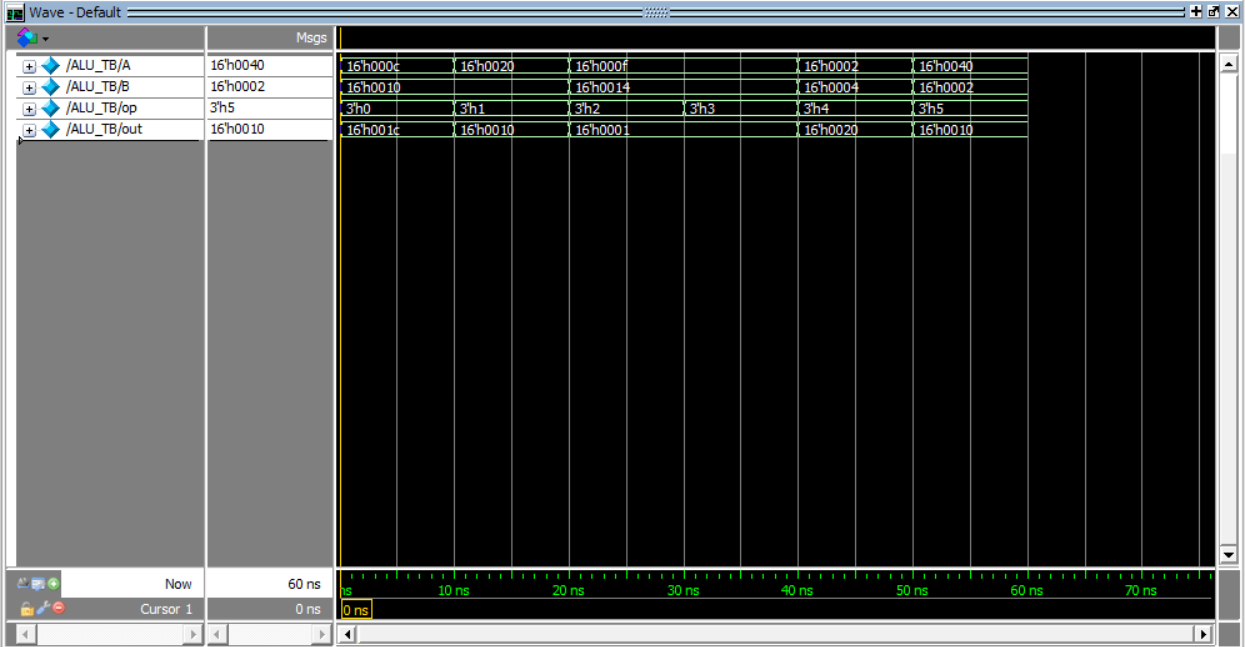
Verilog关键代码：

|  |
| --- |
| module ALU(A,B,op,out);  input[16:1]A,B;  input[3:1]op;  output[16:1]out;  reg[16:1]out;  always@(\*)  begin  case(op)  3'b000:out<=A+B;  3'b001:out<=A-B;  3'b010:out<=A&&B;  3'b011:out<=A||B;  3'b100:out<=A<<B;  3'b101:out<=A>>B;  endcase  end  endmodule |

TestBench代码：

|  |
| --- |
| module ALU\_TB;  reg[16:1]A,B;  reg[3:1]op;  wire[16:1]out;  initial begin  A=12;  B=16;  op=3'b000;  #10;  A=32;  B=16;  op=3'b001;  #10;  A=15;  B=20;  op=3'b010;  #10;  op=3'b011;  #10;  A=2;  B=4;  op=3'b100;  #10;  A=64;  B=2;  op=3'b101;  #10 $stop;  end  ALU uut(  .A(A),.B(B),.op(op),.out(out)  );  endmodule |

moduleSim模拟仿真：



仿真分析：

0~10ns：A与B加法运算，

10~20ns：A与B减法运算，

20~30ns：A与B与运算，

30~40ns：A与B或运算，

40~50ns：A左移B位，

50~60ns：A右移B位。

寄存器堆：

|  |  |
| --- | --- |
| 输入 | 读取地址r1,r2,写信号WE,写地址w1,写数据wval,时钟 CLK |
| 输出 | 地址r1数据out1,地址r2数据out2, |
| 功能 | 读取r1和r2地址的数据,数据wval写入地址w1 |

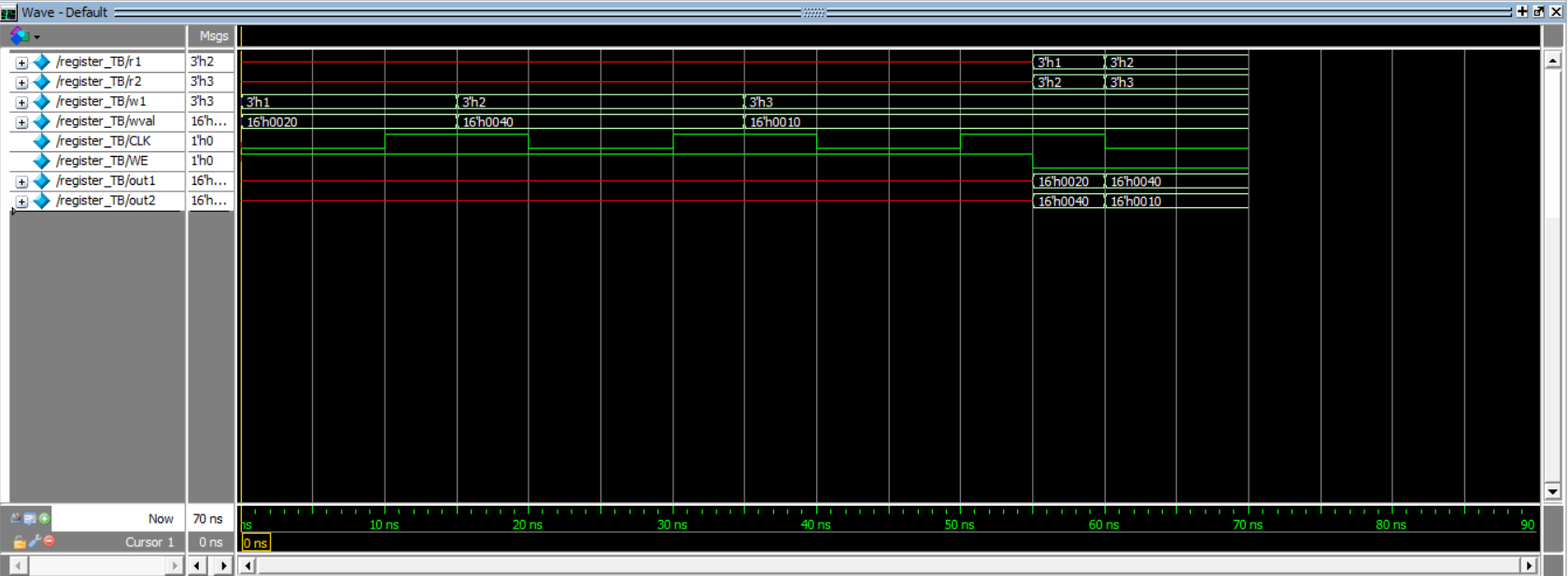
Verilog关键代码：

|  |
| --- |
| module register(r1,r2,WE,w1,wval,out1,out2,CLK);  input wire CLK,WE;  input[3:1]r1,r2,w1;  input[16:1]wval;  output[16:1]out1,out2;  reg[16:1]regHeap[8:1];    assign out1=regHeap[r1];  assign out2=regHeap[r2];    always@(posedge CLK)  begin  if(WE)regHeap[w1]<=wval;  end  endmodule |

TestBench代码：

|  |
| --- |
| module register\_TB;  reg[3:1]r1,r2,w1;  reg[16:1]wval;  reg CLK,WE;  wire[16:1]out1,out2;  register uut(  .r1(r1),.r2(r2),.w1(w1),.wval(wval),.CLK(CLK),.WE(WE),.out1(out1),.out2(out2)  );  initial begin  CLK<=0;  w1<=1;  WE<=1;  wval<=32;  #15;  w1<=2;  wval<=64;  #20;  w1<=3;  wval<=16;  #20;  WE=0;  r1=1;  r2=2;  #5  r1=2;  r2=3;  #10 $stop;  end  always #10 CLK<=~CLK;  endmodule |

ModelSim仿真：



分析：在前三个时钟周期将三组数据存入三个存储单元，最后一个时钟周期将存入的数据进行读取。

## 五、总结

第一次看到这个实验，以为会非常困难，看着题目都觉得可能做不来，但是实际上就是几个基本的数组的存取，比较简单，将几个端口给固定好之后就可以很快的把实验写出来。需要注意的是把各个部件是wire型还是reg型要分清楚，不然编译会失败。

# 实验三、CPU 部件实现之 PC 和半导体存储器 RAM

## 实验目的：

理解和掌握 CPU 中程序计数器 PC 和半导体存储器 RAM 的工作原理，并使用 Verilog 和 ModelSim 进行设计和仿真。

## 二、实验内容

1. 使用 Verilog 完成程序计数器 PC 的设计，要求：

* PC 为 8 位计数器

2. 使用 Verilog 完成数据存储器的设计，并编写测试仿真文件验证其正确性。要求 ：

* 存储字长 16 位，存储容量 1K 字节；
* 一根读写控制信号线控制读写，低电平有效。

## 三、实验原理

PC：给PC输入时钟信号，每到上升沿就把PC的值加1，如果rst信号=1，就把PC值置零。

数据存储器：与实验二的寄存器堆非常相似，将读取端口去掉即可。输入地址既是待取数据的地址又是写入数据的地址。

## 四、实验步骤

PC：

|  |  |
| --- | --- |
| 输入 | 时钟信号clk,清零信号rst |
| 输出 | 当前地址值pc |
| 功能 | 自动计算指令地址 |

Verilog关键代码：

|  |
| --- |
| module PC(  input wire clk, rst,  output reg [7:0] pc  );  always@(posedge clk) begin  if(rst == 1)  pc = 0;  else  pc = pc + 1;  end  endmodule |

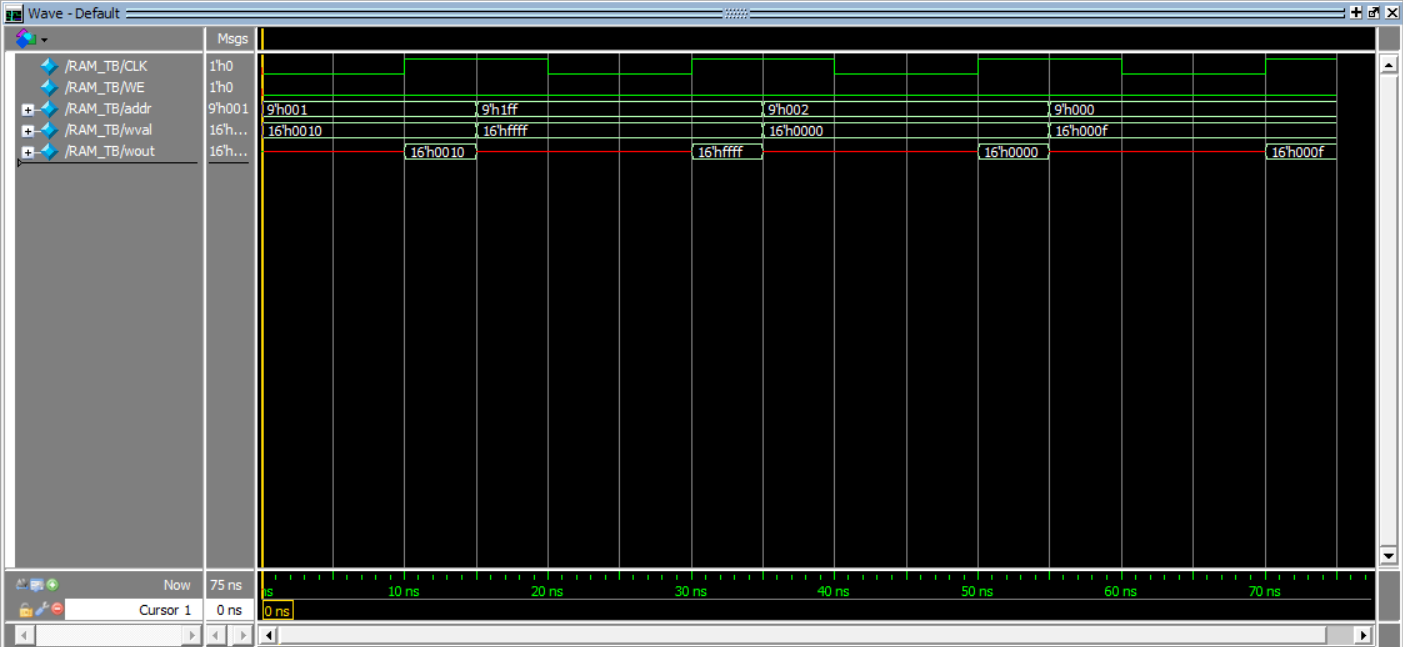
数据寄存器：

|  |
| --- |
| module RAM(CLK,WE,wval,addr,wout);  input CLK,WE;  input[9:1]addr;  input[16:1]wval;  output[16:1]wout;    reg[16:1]regHeap[9'b1000000000:1];    assign wout=regHeap[addr];    always@(posedge CLK)  begin  if(!WE)regHeap[addr]<=wval;  end  endmodule |

TestBench：

|  |
| --- |
| module RAM\_TB;  reg CLK,WE;  reg[9:1]addr;  reg[16:1]wval;  wire[16:1]wout;  RAM uut(  .CLK(CLK),.WE(WE),.addr(addr),.wval(wval),.wout(wout)  );  initial begin  CLK<=0;  WE<=0;  addr<=1;  wval<=16;  #15;  addr<=9'b111111111;  wval<=16'hffff;  #20;  addr<=2'b10;  wval<=17'b10000000000000000;  #20;  addr<=10'b1000000000;  wval<=4'hf;  #20 $stop;  end  always #10 CLK<=~CLK;  endmodule |

ModelSim仿真：



分析：第一个时钟周期在1地址存入数据2，第二个时钟周期存入的数据为17位，自动把最高位给丢弃了，第三个时钟周期输入地址为10位，也自动把最高位给丢弃了。

## 五、总结

这个实验的pc很简单，没有转移等复杂的功能很好写，RAM本来以为是个很复杂的模块，真的做起来的时候实际上也只是在数组中存取数据，很简单。

实验四、单周期 CPU 设计与实现——单指令 CPU

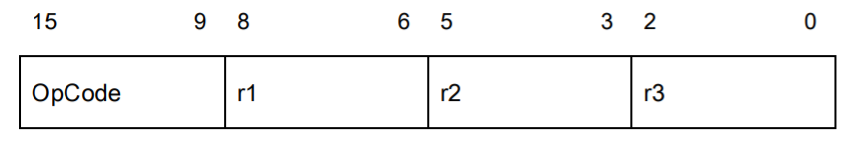
## 实验目的：

通过设计并实现支持一条指令的 CPU，理解和掌握 CPU 设计的基本原理和过程。

## 二、实验内容

设计和实现一个支持加法指令的单周期 CPU。要求该加法指令（表示为 add r1，r2， r3）格式约定如下：

* 采用寄存器寻址，r1，r2，r3 为寄存器编号，r1 和 r2 存放两个源操作数，r3 为目标寄存器，其功能为[r1] + [r2] -> r3；
* 指令字长 16 位，操作码和地址码字段分配如下所示：



## 三、实验原理

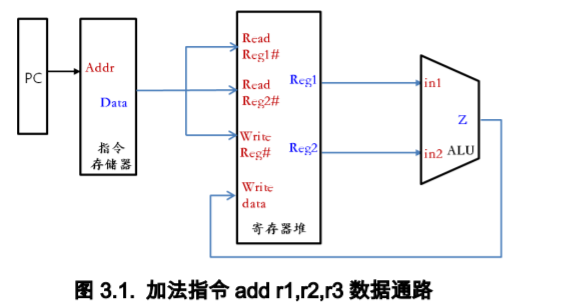
单周期 CPU 是指所有指令均在一个时钟周期内完成的 CPU。CPU 由数据通路及其控 制部件两部分构成，因而要完成一个支持若干条指令 CPU 的设计，需要依次完成以下两件 事：

1）根据指令功能和格式设计 CPU 的数据通路；

2）根据指令功能和数据通路设计控制部件。

3.1 **根据功能和格式完成 CPU 的数据通路设计**

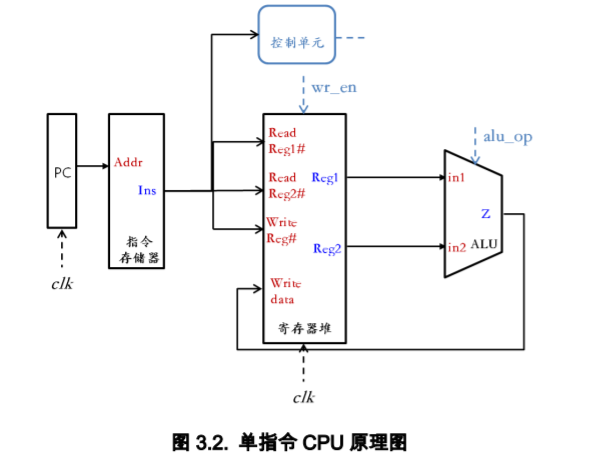
本实验需要设计的CPU只需要支持一条加法指令，而该指令的功能是在一个时钟周期 内从寄存器组中r1和r2中取出两个操作数，然后送到ALU进行加法运算，最后把计算结 果保存到r1寄存器中。下图给出了改加法指令的数据通路图。



此外，还需要确定各个部件的位数，为了简单起见，我们假设目标CPU 的机器字长、 存储字长和指令字长相等均为16位，存储单元个数假设为256，按字寻址，并取 PC 位数为 8。

**3.2 根据指令功能、数据通路完成控制单元的设计**

控制单元的功能是为当前要执行的指令产生微操作命令从而完成该指令的执行。为了能 够完成加法指令的执行，结合图 1，控制单元需要在取出指令后根据指令操作码（本例中是 加法指令），控制 ALU（参考实验二）做加法（通过给 alu\_op 信号线相应赋值），并把结果 写回寄存器组（参考实验三）中（通过给 wr\_en 赋值为 true）。图 2 给出了整合控制单元后 目标 CPU 的原理图，系统时钟信号也已标注。



## 四、实验步骤

PC：

|  |  |
| --- | --- |
| 输入 | 时钟信号clk,清零信号rst |
| 输出 | 当前地址值pc |
| 功能 | 自动计算指令地址 |

Verilog关键代码：

|  |
| --- |
| module PC(  input wire clk, rst,  output reg [7:0] pc  );    always@(posedge clk) begin  if(rst == 1)  pc = 0;  else  pc = pc + 1;  end    endmodule |

指令寄存器IrMemory：

|  |  |
| --- | --- |
| 输入 | 指令地址addr |
| 输出 | 指令内容ins |
| 功能 | 根据PC的指令地址取指令 |

Verilog关键代码：

|  |
| --- |
| module IrMemory(addr,ins);  input[7:0]addr;  output reg[15:0]ins;    reg[15:0]units[8'b11111111:0];  integer i;    initial begin  for(i=0;i<256;i=i+1)  begin  units[i][2:0]=1;  units[i][5:3]=2;  units[i][8:6]=3;  units[i][15:9]=0;  end  end    always@(\*)  begin  ins=units[addr];  end    endmodule |

数据寄存器register：

|  |  |
| --- | --- |
| 输入 | 读取地址r1,r2,写信号WE,写地址w1,写数据wval,时钟 CLK |
| 输出 | 地址r1数据out1,地址r2数据out2, |
| 功能 | 读取r1和r2地址的数据,数据wval写入地址w1 |

Verilog关键代码：

|  |
| --- |
| module register(r1,r2,WE,w1,wval,out1,out2,CLK);  input wire CLK,WE;  input[2:0]r1,r2,w1;  input[15:0]wval;  output reg[15:0]out1,out2;  reg[15:0]regHeap[7:0];  integer i;    initial begin  for(i=0;i<8;i=i+1)  begin  regHeap[i]=i;  end  end      always@(\*)  begin  out1<=regHeap[r1];  out2<=regHeap[r2];  end    always@(negedge CLK)  begin  if(WE)regHeap[w1]<=wval;  end  endmodule |

CU：

|  |  |
| --- | --- |
| 输入 | 操作码ins |
| 输出 | 数据寄存器读写信号WE,操作符op |
| 功能 | 对操作码译码，让运算器做相应操作 |

Verilog关键代码：

|  |
| --- |
| module CU(ins,WE,op);  input[6:0]ins;  output reg WE;  output reg[2:0]op;    always@(\*)  begin  if(ins==0)WE<=1;  op<=3'b000;  end  endmodule |

ALU：

|  |  |
| --- | --- |
| 输入 | 操作数A,B,操作符op |
| 输出 | 运算结果out |
| 功能 | 对两个操作数进行指定运算 |

Verilog关键代码：

|  |
| --- |
| module ALU(A,B,op,out);  input[16:1]A,B;  input[3:1]op;  output[16:1]out;  reg[16:1]out;  always@(\*)  begin  case(op)  3'b000:out<=A+B;  3'b001:out<=A-B;  3'b010:out<=A&&B;  3'b011:out<=A||B;  3'b100:out<=A<<B;  3'b101:out<=A>>B;  endcase  end  endmodule |

CPU：

|  |  |
| --- | --- |
| 输入 | 时钟周期clk,PC清零信号rst |
| 输出 |  |
| 功能 | 各个部件组合自动工作 |

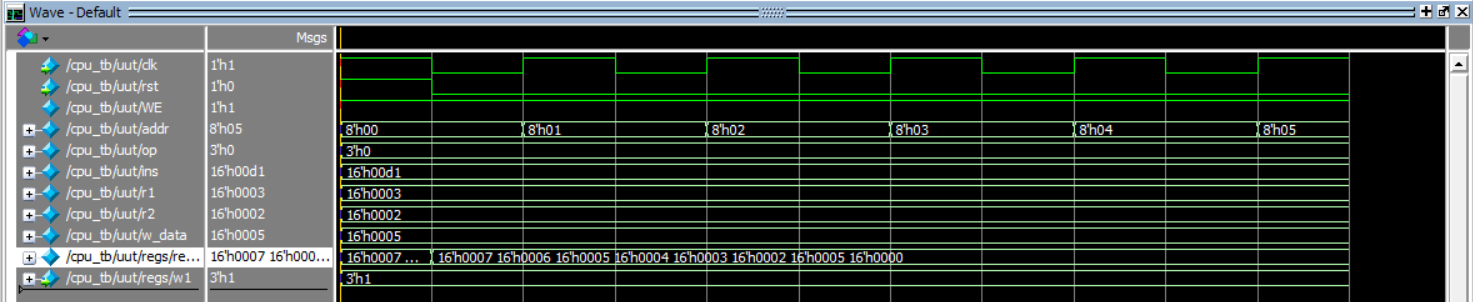
Verilog关键代码：

|  |
| --- |
| module cpu(clk,rst);  input clk;  input rst;    wire WE;  wire[7:0]addr;  wire[2:0]op;  wire[15:0]ins,r1,r2,w\_data;    PC pc(  .clk(clk),.rst(rst),.pc(addr)  );    IrMemory mem(  .addr(addr),.ins(ins)  );    CU cu(  .ins(ins[15:9]),.WE(WE),.op(op)  );    ALU alu(  .A(r1),.B(r2),.op(op),.out(w\_data)  );    register regs(  .r1(ins[8:6]),.r2(ins[5:3]),.WE(WE),.w1(ins[2:0]),.wval(w\_data),.out1(r1),.out2(r2),.CLK(clk)  );      endmodule |

TestBench代码：

|  |
| --- |
| module cpu\_tb;  reg clk, rst;  cpu uut(  .clk(clk), .rst(rst)  );    initial begin  clk = 1;  rst = 1;  #1 rst = 0;  #10 $stop;  end  always #1 clk = ~clk;  endmodule |

ModelSim仿真：



分析：预先在指令寄存器设置的所有指令操作数一是r1=3,r2=2,w1=1，也就是将地址3和地址2中的内容相加存入地址1中，地址n中的数为n，则就是3+2=5，在结果中可以看到w\_data，也就是要储存的结果一直为5，说明运算正确。

## 五、总结

这个实验把之前几个实验的各个部分给重新加强，然后将所有的部件组装起来，就变成了一个只有一个加法指令的简单的CPU，各个部件之间的连接要非常小心，端口也不能写错，任何一个环节出现问题都会导致CPU无法正常运行。自己动手写CPU内部的各个数据通路，让我对CPU内部结构有了更深入的了解，原来只是在书本上看各个部件通路应该怎样连接，需要哪些部件都很清楚，现在到自己写的时候，还是有很多地方细节没有了解，所以花了大量时间用于设计端口通路和调试上，但是还是很有收获。

# 实验五、单周期 CPU 设计与实现——十条指令 CPU

## 实验目的：

通过设计并实现支持 10 条指令的 CPU，进一步理解和掌握 CPU 设计的基本原理和过程。

## 二、实验内容

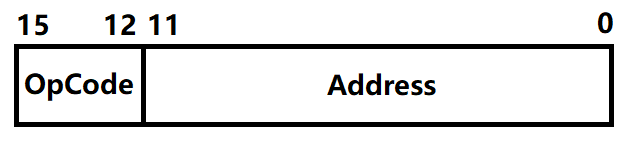
设计和实现一个支持如下十条指令的单周期 CPU。

* 非访存指令
* 清除累加器指令 CLA
* 累加器取反指令 COM
* 算术右移一位指令 SHR：将累加器 ACC 中的数右移一位，结果放回 ACC
* 循环左移一位指令 CSL：对累加器中的数据进行操作
* 停机指令 STP
* 访存指令
* 加法指令 ADD X：[X] + [ACC] –〉ACC，X 为存储器地址，直接寻址
* 存数指令 STA X，采用直接寻址方式
* 取数指令 LDA X，采用直接寻址
* 转移类指令
* 无条件转移指令 JMP imm：signExt(imm) -> PC
* 有条件转移（负则转）指令 BAN X: ACC 最高位为 1 则（PC）+ X -> PC,否则 PC 不变

## 三、实验原理

**3.1 设计指令**

由于需要10条指令，因此需要四位操作码，剩余的12位作地址码，分配如下：



然后分配操作码：

|  |  |  |  |
| --- | --- | --- | --- |
| 0000 | CLA | 0101 | ADD |
| 0001 | COM | 0110 | STA |
| 0010 | SHR | 0111 | LDA |
| 0011 | CSL | 1000 | JMP |
| 0100 | STP | 1001 | BAN |

CPU数据通路：

数据寄存器

ALU

ACC

B

指令寄存器

PC

JMP

BAN

IR

CLK

CLK

RST

## 四、实验步骤

PC

|  |  |
| --- | --- |
| 输入 | 时钟clk,清零信号rst,停机信号stp,跳转信号WR,立即数imm,条件转移信号ban |
| 输出 | 当前地址值addr |
| 功能 | 自动计算指令地址 |

Verilog关键代码：

|  |
| --- |
| module pc(clk,rst,addr,stp,WR,imm,ban);  input clk,rst,stp,WR,ban;  input [11:0]imm;  output reg[11:0]addr;    reg state;  initial begin  state=1;  addr=0;  end    always@(\*)  begin  if(stp)  state=0;  if(rst)  addr=0;  end    always@(posedge clk)  begin  if(state)  addr=addr+1;  end    always@(negedge clk)  begin  if(WR)  addr=imm-1;  if(ban)  addr=addr+imm-1;  end  endmodule |

IrMemory

|  |  |
| --- | --- |
| 输入 | 指令地址addr |
| 输出 | 地址对应的指令ins |
| 功能 | 根据指令地址获取指令 |

Verilog关键代码：

|  |
| --- |
| module IrMemory(addr,ins);  input[11:0]addr;  output reg[15:0]ins;    reg[15:0]units[12'b111111111111:0];    initial begin  units[0]<=16'b0111\_0000\_0000\_0000;//LDA 0  units[1]<=16'b0001\_0000\_0000\_0000;//COM  units[2]<=16'b0010\_0000\_0000\_0000;//SHR  units[3]<=16'b0011\_0000\_0000\_0000;//CSL  units[4]<=16'b0000\_0000\_0000\_0000;//CLA  units[5]<=16'b0101\_0000\_0000\_0001;//ADD  units[6]<=16'b0110\_0000\_0000\_0010;//STA  units[7]<=16'b1000\_0000\_0000\_1001;//JMP 9  units[9]<=16'b1001\_0000\_0000\_0010;//BAN pc+2  units[11]<=16'b0100\_0000\_0000\_0000;//STP  end    always@(\*)  begin  ins=units[addr];  end  endmodule |

cu

|  |  |
| --- | --- |
| 输入 | 操作码ins |
| 输出 | 运算符op,停机信号stp,无条件转移信号pcWR,acc读写信号accWR,数据寄存器读写信号dateWR |
| 功能 | 指令译码，发出控制信号 |

Verilog关键代码：

|  |
| --- |
| module cu(ins,op,stp,pcWR,accWR,dateWR);  input[3:0]ins;  output reg [3:0]op;  output reg stp,pcWR,accWR,dateWR;    initial begin  accWR<=0;  dateWR<=0;  pcWR<=0;  stp<=0;  op<=4'b0100;  end    always@(ins)  begin  case(ins)  4'b0000:{stp,pcWR,accWR,dateWR,op}=8'b00100000;//CLA  4'b0001:{stp,pcWR,accWR,dateWR,op}=8'b00100001;//COM  4'b0010:{stp,pcWR,accWR,dateWR,op}=8'b00100010;//SHR  4'b0011:{stp,pcWR,accWR,dateWR,op}=8'b00100011;//CSL  4'b0100:{stp,pcWR,accWR,dateWR,op}=8'b10000100;//STP  4'b0101:{stp,pcWR,accWR,dateWR,op}=8'b00100101;//ADD  4'b0110:{stp,pcWR,accWR,dateWR,op}=8'b00010110;//STA  4'b0111:{stp,pcWR,accWR,dateWR,op}=8'b00100111;//LDA  4'b1000:{stp,pcWR,accWR,dateWR,op}=8'b01001000;//JMP  4'b1001:{stp,pcWR,accWR,dateWR,op}=8'b00001001;//BAN  endcase  end  endmodule |

register

|  |  |
| --- | --- |
| 输入 | 读写信号WR,时钟clk,地址addr,存入数据in |
| 输出 | 读取数据out |
| 功能 | 读取保存数据 |

Verilog关键代码：

|  |
| --- |
| module register(WR,clk,addr,in,out);//数据寄存器  input WR,clk;  input[11:0]addr;  input[15:0]in;  output[15:0]out;    reg[15:0]units[12'b111111111111:0];    assign out=units[addr];    initial begin  units[0]=16'b0000000000000001;  units[1]=16'b1000000000000000;  end    always@(negedge clk)  begin  if(WR)  units[addr]=in;  end        endmodule |

acc

|  |  |
| --- | --- |
| 输入 | 时钟clk,读写信号WR,输入数据in |
| 输出 | 输出数据out |
| 功能 | 保存运算数据或读写数据 |

Verilog关键代码：

|  |
| --- |
| module acc(clk,WR,in,out);  input clk,WR;  input[15:0]in;  output[15:0]out;    reg[15:0]date;  initial begin  date<=0;  end    assign out=date;    always@(negedge clk)  begin  if(WR)date<=in;  end    endmodule |

alu

|  |  |
| --- | --- |
| 输入 | 运算数A,B,运算符op |
| 输出 | 运算结果out,条件转移信号ban |
| 功能 | 数据计算 |

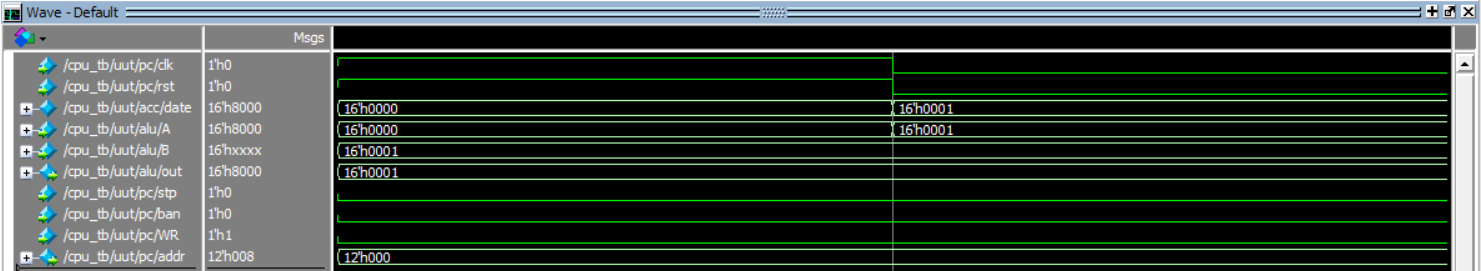
cpu

|  |  |
| --- | --- |
| 输入 | 时钟信号clk,清零信号rst |
| 输出 |  |
| 功能 | 各个部件组合自动工作 |

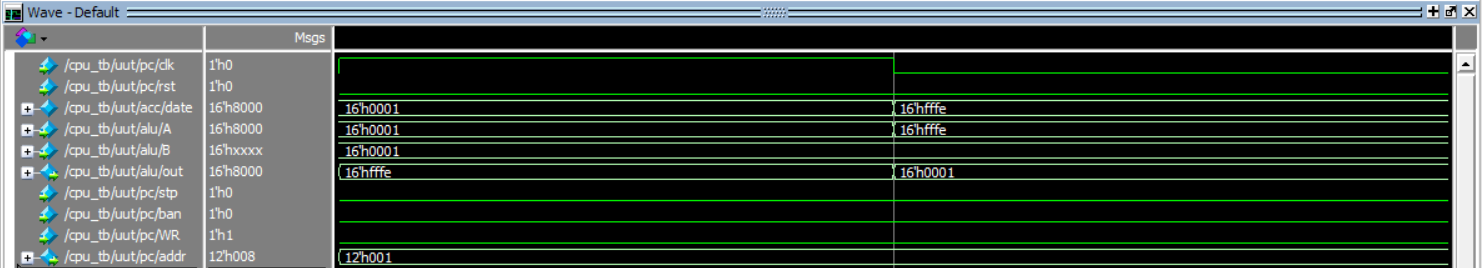
TestBench代码：

|  |
| --- |
| module cpu\_tb;  reg clk,rst;  cpu uut(  .clk(clk),.rst(rst)  );    initial begin  clk=1;  rst=1;    #1 rst=0;    #20 $stop;  end    always #1 clk=~clk;  endmodule |

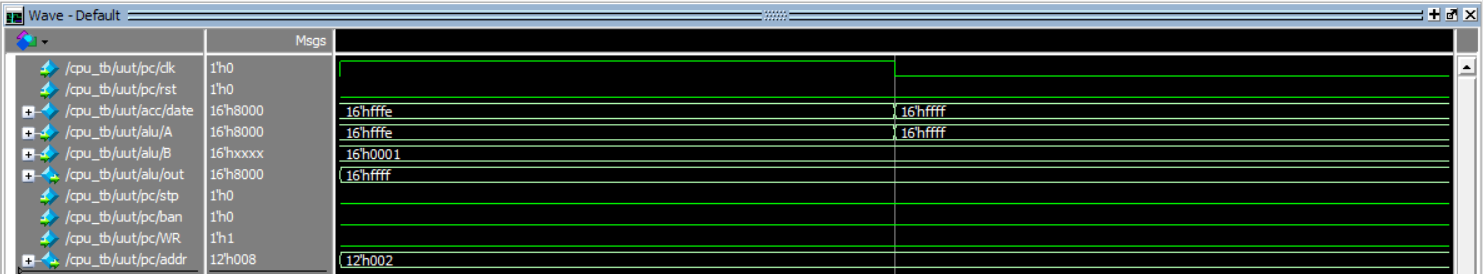
ModelSim仿真：



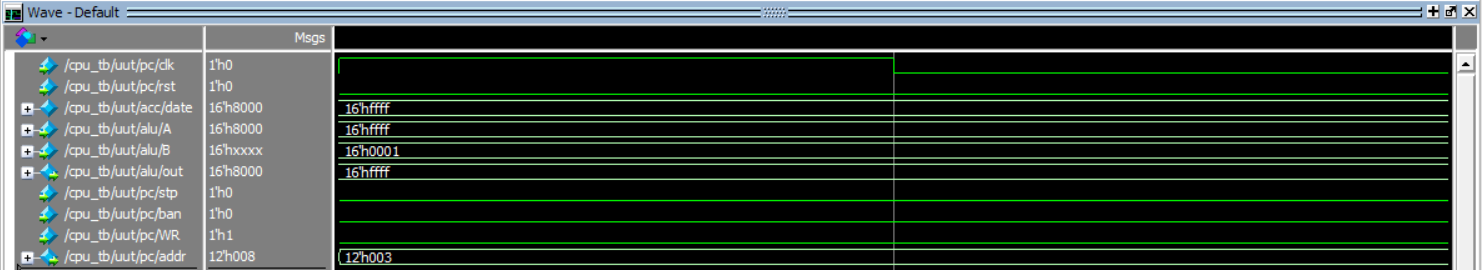
第一周期LDA取数到acc，acc内容为0001h



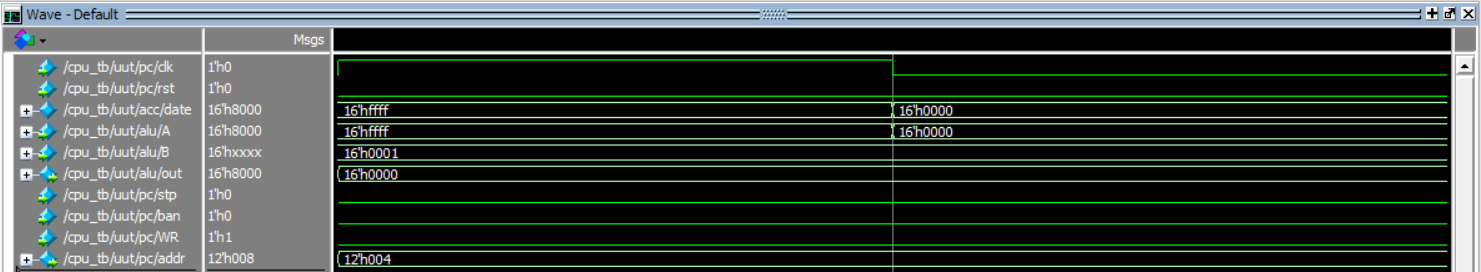
第二周期COM将acc内容取反，变成fffeh



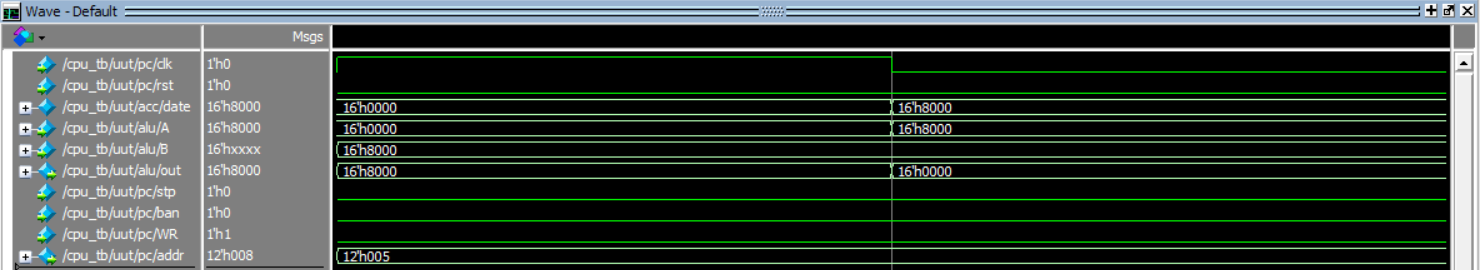
第三周期SHR将acc右移一位，变成ffffh



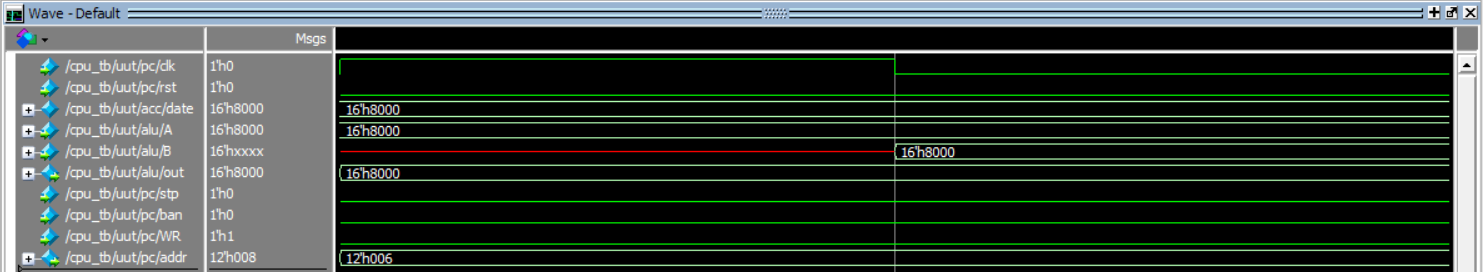
第四周期CSL将acc循环左移，数据不变



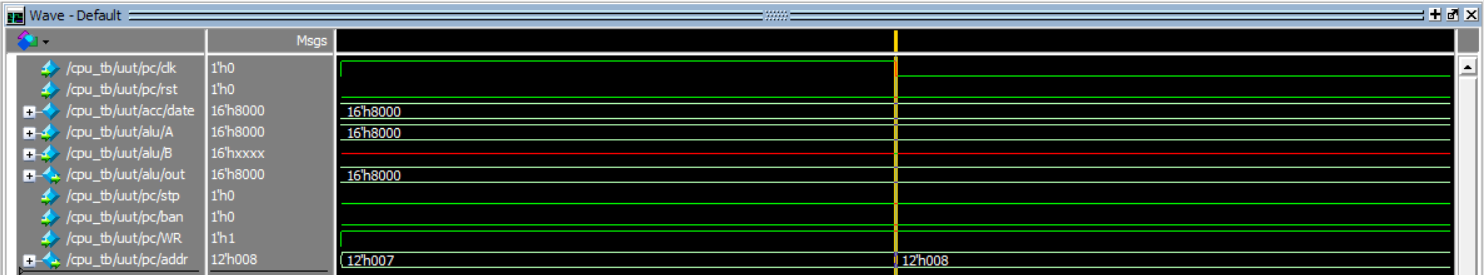
第五周期CLA变为0



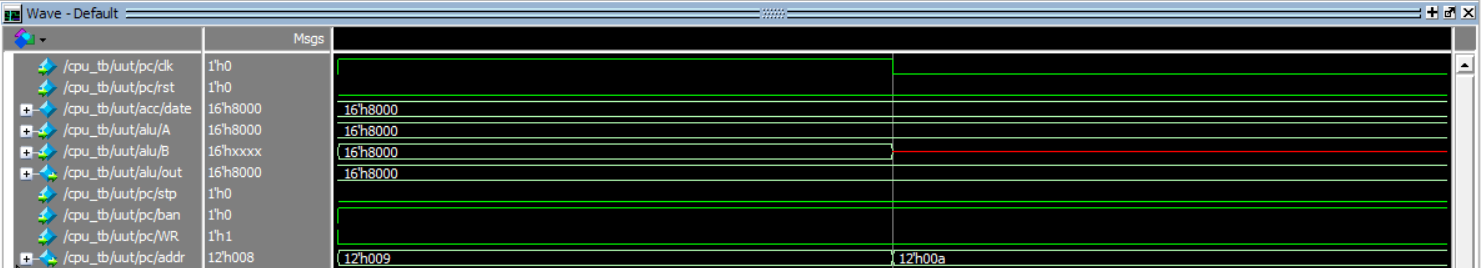
第六周期ADD，加地址为1的数，地址为1存的是8000h，相加结果为8000h



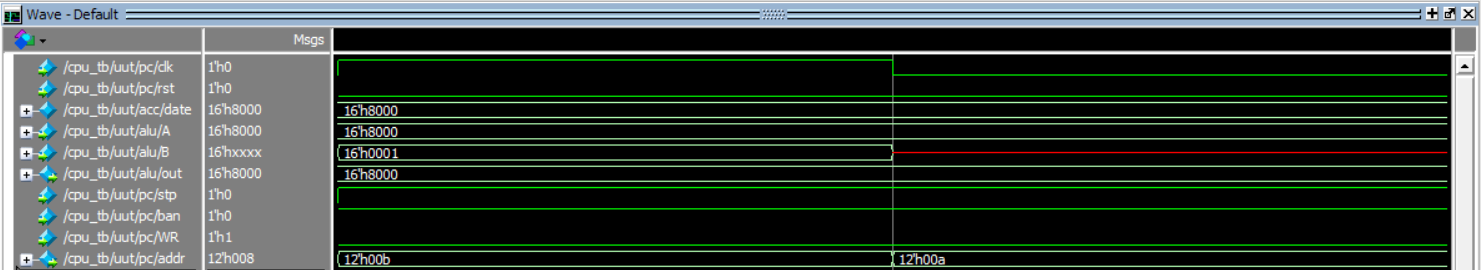
第七周期STA存数到2地址



第八周期JMP无条件转移到地址9，WR=1，BAN=0



第九周期BAN条件转移到地址11，acc内容为1开头，于是BAN=1,WR=0



第十周期STP停机，stp=1，PC停止工作

## 五、总结

多指令CPU的设计比较有难度，需要将实验四的部件再一次提升，然后再将各个部件连接起来，指令的控制信号也更多，连接更复杂，由于ModelSim没有代码的报错位置，所以我修改bug是一段一段代码调试，看哪段代码出现问题，一个非常简单的错误却花费了我大量的时间去改，以后写这种代码需要更加细心，不要把时间浪费在一些简单的错误上。多指令CPU设计也让我更进一步体会了CPU的工作原理和数据通路，对CPU内部的工作有了更加直观的印象。