实验报告——单指令 CPU 电路设计

一、需求分析

第一部分：基本部件的设计

第一部分重在理解各个基本部件的功能，经过查询 MIPS 指令集，总结如下：

指令名称	指令格式	功能描述	汇编指令中的操作码
add	ADD rd,rs,rt	GPR[rd] ← GPR[rs] + GPR[rt]	000000+100000
sub	SUB rd,rs,rt	GPR[rd] ← GPR[rs] - GPR[rt]	000000+100010
and	AND rd,rs,rt	GPR[rd] ← GPR[rs] AND GPR[rt]	000000+100100
or	OR rd,rs,rt	GPR[rd] ← GPR[rs] or GPR[rt]	000000+100101
slt	SLT rd,rs,rt	GPR[rd] ← (GPR[rs] < GPR[rt])	000000+101010
addi	ADDI rt,rs,immediate	GPR[rt]← GPR[rs] + immediate	001000
andi	ANDI rt,rs,immediate	GPR[rt]← GPR[rs] AND immediate	001100
ori	ORI rt,rs,immediate	GPR[rt]← GPR[rs] or immediate	001101
slti	SLTI rt,rs,immediate	GPR[rt]← (GPR[rs] < immediate)	001010
sw	SW rt,offset(base)	memory[GPR[base] + offset] ← GPR[rt]	101011
lw	LW rt,offset(base)	GPR[rt]← memory[GPR[base]+offset]	100011
j	Ins rt,rs,pos,size	GPR[rt]← InsertField(GPR[rt],GPR[rs],msb,lsb])	000010
nop	NOP	No operation	000000+000000

备注：

① GPR[r]表示通用寄存器 r

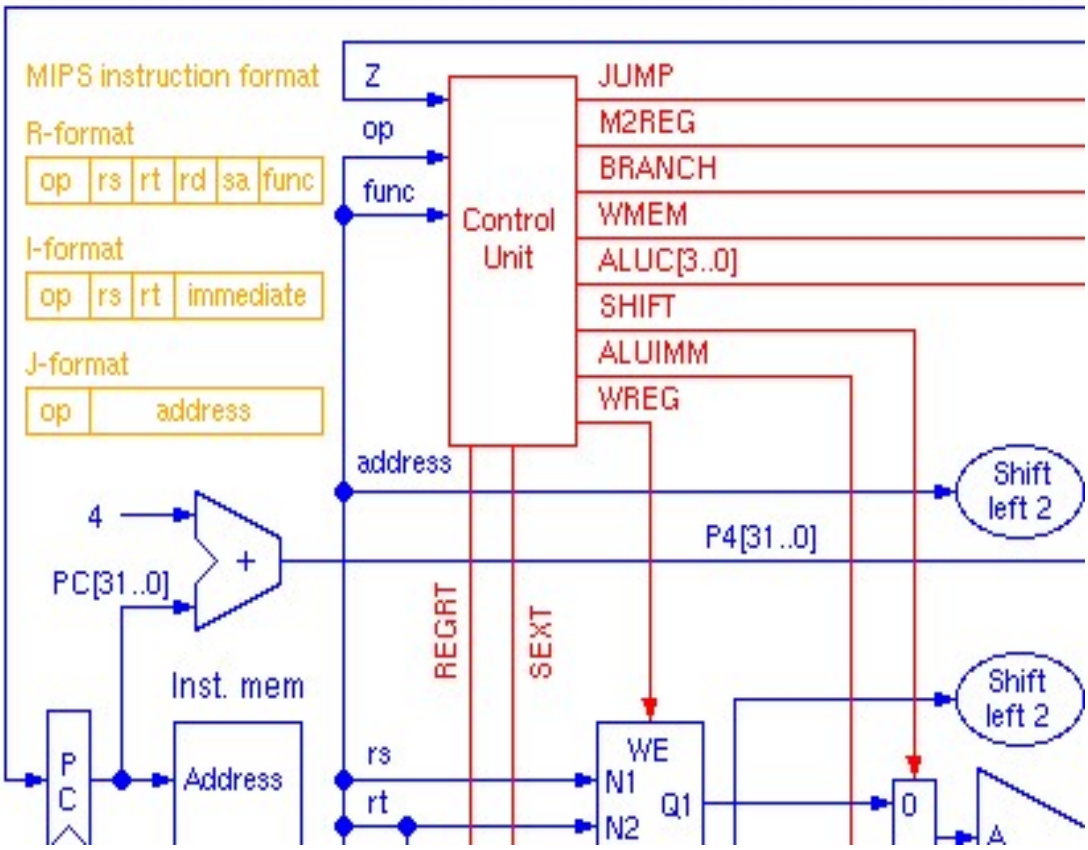
② Immediate 表示立即数

③ memory[x]表示地址为 x 的内存空间

④ 具体的二进制指令具体格式没有在表中列出来

第二部分：控制单元的设计

第二部分主要是控制部分，如何控制各个单元完成具体的 MIPS 指令，时序如何组织。这一部分将会在实验设计中详细展开介绍。



二、实验设计

结合上课所讲的有关数字部件设计的内容，参考文档中给出的两个部分的任务分配，我将实现分为了 datapath 部分和 controller 部分，分别对应了文档中的两个部分的设计。

第一部分:datapath

1.module mux2：

a）输入：

d0：宽度为 WIDTH，默认为 8-bit

d1：宽度为 WIDTH，默认为 8-bit

s：表示选通控制位，为 1 时选择 d1，为 0 时选择 d0

b）输出：

y：(宽度为 WIDTH，默认为 8-bit，选通器的输出结果)

c）功能描述：

参考需求文档中给出的简单的单指令 CPU 的电路图(如上图所示)，我们需要在 regFile、ALU 等多处用到二选一的选通器，所以在此实现作为工具 module

d）代码解析：

代码实现较为简单，只需要一个条件表达式即可，代码如下：

```
`timescale 1ns / 1ps

//mux2 : 二选一复用器
module mux2 # (parameter WIDTH = 8) (d0, d1, s, y);
 input [WIDTH-1:0] d0, d1;
 input s;
 output [WIDTH-1:0] y;

 assign y = s ? d1 : d0;
endmodule
```

2. module alu：

a)输入：

src1：表示第一个操作数，32-bit

src2：表示第二个操作数，32-bit

alucont：用于选择具体的 operation，3-bit；根据文档需求，此处需要实现四个基本指令，add、sub、or、and 以及 beq，具体的 alucont 和 operation 的对应表如下：

alucont	operation
000	src1 & src2
010	src1 + src2
001	src1 | src2
011	src1 - src2
111	src1 < src2 ? 1 : 0

b)输出：

result：表示两个操作数在 alucont 对应操作后结果，32-bit

zero：表示 result 结果是否为 0，1-bit，类似于汇编指令中的 ZF

c)功能：

算术逻辑单元，主要是为了实现需求文档中的加减并或及比较操作

d)关键代码解析：

此处的实现比较简单，在我们的前几个 LAB 中设计过，也用到过，所以此处不再赘述，直接上代码：

```
`timescale 1ns / 1ps
//ALU:用于逻辑运算
module alu(src1, src2, alucont, result, zero);
 input [31:0] src1, src2;
 input [2:0] alucont;
 output reg [31:0] result;
 output zero;

 assign zero = result == 0 ? 1 : 0;
```

```
always@(*) begin
 case(alucont[2:0])
  3'b000 : result <= src1 & src2;
  3'b010 : result <= src1 + src2;
  3'b001 : result <= src1 | src2;
  3'b011 : result <= src1 - src2;
  3'b111 : begin
        if(src1 < src2) result <= 1;
        else result <= 0;
       end
 endcase
end
//endmodule
```

3. module sl2：

a)输入：

a：移位操作的操作数，32-bit

b)输出：

y：移位操作的结果，32-bit

c)功能：

将操作数 a 左移 2 位

d)代码解析：

将 a 的低 30 位赋给 y 的高 30 位，后面两位补 0

```
`timescale 1ns / 1ps

//shift left register: left shift 2 bit
module sl2(a, y);
 input [31:0] a;
 output [31:0] y;

 assign y = {a[29:0], 2'b00};
endmodule
```

4.module extendNum：

a)输入：

imm：要进行符号扩展的操作数，16-bit

b)输出：

y：符号扩展之后的结果，32-bit

c)功能：

将操作数 imm 进行符号扩展，用于对

d)代码：

具体实现中，就是将 y 的最高 16 位用 imm 的符号位填充，也就是 imm 的最高位，低 16 位与 imm 相等，代码如下：

```
`timescale 1ns / 1ps

//signal extend immediate:
module extendNum(imm, y);
 input [15:0] imm;
 output [31:0] y;
 //将imm的高位直接复制到前16位
 assign y = {{16{imm[15]}}, imm};
endmodule
```

5.module regFile：

a)输入：

clk：时钟，1-bit

regwriteEn：写入使能控制位，1-bit

regwriteaddr：当 enable 有效时，在时钟上升沿写入寄存器的值，50bit

regwritedata：当 enable 有效时，在时钟上升沿写入寄存器的“地址”，32-bit

rsaddr：对应与第一部分指令的基本形式中的 rs 寄存器，5-bit

rtaddr：对应于第一部分指令的基本形式中的 rt 寄存器，5-bit

b)输出：

rsdata：rs 寄存器中的值，32-bit

rtdata：rt 寄存器中的值，32-bit

c)功能：

该模块实现了两部分的功能，统一起来都是对寄存器文件的操作，参考了前几次 LAB 中的寄存器文件的设计，写入控制位 regwriteEn 对应的就是前几次 LAB 中的 WE：

①当 enable 有效时，在时钟上升沿将 regwritedata 写入 regwriteaddr 对应的寄存器中

②若 rsaddr(或 rtaddr)不为 0，那么将对应的寄存器中的值读出来

d)代码解析：

因为此处也是前几次 LAB 中做过的内容，并且在此次实现中更为简单，所以此处具体的详细实现不再赘述，直接上代码：

```
`timescale 1ns / 1ps

//register file:向寄存器文件中写入数据;
module regFile(clk, regwriteEn, regwriteaddr,
        regwritedata, rsaddr, rtaddr, rsdata, rtdata);
 input clk, regwriteEn; // write enable
 input [4:0] regwriteaddr;  //reg write addr
 input [31:0] regwritedata;  //reg write data
 input [4:0] rsaddr, rtaddr;  //RsAddr, RtAddr
 output [31:0] rsdata, rtdata;  //RsData, RtData

 reg [31:0] rf[31:0]; //32 * 32 registers

 //we 有效时，写入数据
 always @(posedge clk) begin
  if(regwriteEn) rf[regwriteaddr] <= regwritedata;
 end

 // 读Rs Rt寄存器中的数据
 assign rsdata = (rsaddr != 0) ? rf[rsaddr] : 0;
 assign rtdata = (rtaddr != 0) ? rf[rtaddr] : 0;
endmodule
```

6.module PCPlus4：

a)输入：

a：原始的 PC 寄存器中的值，32-bit

b：PC 寄存器要加上的值，32-bit

b)输出：

result：加法之后，PC 寄存器中的值，32-bit

c)功能：

这个模块主要是为了实现对 PC 寄存器中的值的操作，得到下一条指令

e) 代码解析：

代码实现十分简单，result=a+b，此处不再贴代码截图

7.module PC：

a)输入：

clk：时钟，1-bit

reset：重置控制位，1-bit

next：下一条指令

b)输出：

current：当前指令

c)功能：

输入下一条指令，在时钟周期上沿，根据 reset 控制位，得到当前指令

d)代码解析：

关键的部分在于，在时钟上升沿，如果 reset=1，得到当前指令为 0，否则为 next 指令

```
`timescale 1ns / 1ps

//program counter: output current instruction, input next instu
module PC #(parameter WIDTH = 8) (clk, reset, next, current);
 input clk, reset;
 input [WIDTH-1:0] next;
 output reg [WIDTH-1:0] current;

 always @(posedge clk, posedge reset) begin
  if(reset)  current <= 0;
  else  current <= next;
 end
endmodule
```

8.module datapath：

a)功能：

该 module 利用上面所实现的最基本的功能部件，实现 datapath

b)输入：

clk：表示时钟，1-bit

reset：表示重置 PC 的控制位，1-bit

memtoreg：表示将内存数据写入寄存器的使能控制位，当该值有效时，内存中的值加载到寄存器中，1-bit

alusrc：表示当前指令是 R 类型还是 I 类型，1 表示 I 类型，0 表示 r 类型，1-bit

regdst：表示写入的是 rt 还是 rd，R 类型指令写入的是 rd，对应该位为 1，I 类型指令写入的是 rt，对应该位为 0，1-bit

regwrite：表示是否要写入寄存器，1 表示写入，1-bit

jump：表示当前指令是否为跳转指令，1 表示跳转，1-bit

alucontrol：表示当前的操作码，对应表格已经在上面介绍 module alu 时给出，3-bit

instr：表示当前正在执行的指令，32-bit

readdata：表示从内存中读取的数值，在执行过程中可能会被加载到寄存器中，32-bit

c)输出：

zero：表示 alu 的计算结果是否为 0，1-bit

pc：表示下一条指令的地址

aluout：表示 alu 单元的计算结果

writedata：表示要写入的寄存器的地址

d)整体逻辑：

```
//next PC logic
PC #(32) pcreg(clk, reset, pcnext, pc);
PCPlus4 pcadd1(pc, 32'b100, pcplus4); // pc plus4
sl2 immsh(signimm, signimmsh);
PCPlus4 pcadd2(pcplus4, signimmsh, pcbranch);
mux2 #(32) pcbrmux(pcplus4, pcbranch, pcsrc, pcnextbr);
mux2 #(32) pcmux(pcnextbr, {pcplus4[31:28], instr[25:0], 2'b00}, jump, pcnext);

//register file logic
regFile rf(clk, regwrite, writereg, result, instr[25:21], instr[20:16], srca, writedata);
mux2 #(5) wrmux(instr[20:16], instr[15:11], regdst, writereg);
mux2 #(32) resmux(aluout, readdata, memtoreg, result);
extendNum se(instr[15:0], signimm);

//ALU logic
mux2 #(32) srcbmux(writedata, signimm, alusrc, srcb);
alu alu(srca, srcb, alucontrol, aluout, zero);
```

第二部分：controller

此处是有层次调用关系的，我们按照从下往上的顺序分析。

1.module maindec

a)输入：

op：表示当前指令的 operation 类型，6-bit

b)输出：

memtoreg、memwrite、branch、alusrc、regdst、regwrite、jump、aluop

c)功能：

根据 op 类型得到 datapath 中所需要的控制位

d)代码解析：

关键代码如下，实现很简单，不再过多赘述：

```
reg [10:0] controls;

assign {regwrite, regdst, alusrc, branch, memwrite, memtoreg, jump, aluop} = controls;

always @(*) begin
 case(op)
  6'b000000 : controls <= 10'b1100000010; //R-type
  6'b100011 : controls <= 10'b1010010000; //LW
  6'b101011 : controls <= 10'b0010100000; //SW
  6'b000100 : controls <= 10'b0001000001; //BEQ
  6'b001000 : controls <= 10'b1010000000; //ADDI
  6'b000010 : controls <= 10'b0000001000; //jump
  6'b001100 : controls <= 10'b1010000100; //ANDI
  6'b001101 : controls <= 10'b1010000101; //ORI
  6'b001010 : controls <= 10'b1010000110; //SLTI
  default : controls <= 10'bxxxxxxxxxx; //INVALID
 endcase
end
```

2.module aludec

a)输入：

funct：表示指令的低 6 位，当指令为 R 类型指令时，该 6 位可以标识不

同的具体操作，6-bit

aluop：表示在指令最高 6 位的指令类型，当指令为非 R 类型的操作时，该 6 位可以直接用来区分不同的操作，6-bit

b)输出：

alucontrol：表示操作码，对应表在介绍 module alu 时已经列出来了，此处不再重复

c)功能：

根据指令中的必要信息(高六位加上必要时的低六位)得到操作类型对应的操作码

d)代码解析：

此处的实现较为简单，需要提一下的是对于 R 类型的指令，无论是 ADD 还是 AND 抑或是其他，所有的最高六位都为 0，需要低六位来区分：

```
always @(*) begin
 case(aluop)
  3'b000 : alucontrol <= 3'b010; //add
  3'b001 : alucontrol <= 3'b011; //sub
  3'b100 : alucontrol <= 3'b000; //and
  3'b101 : alucontrol <= 3'b001; //or
  3'b110 : alucontrol <= 3'b111; //slt
  default : case(funct) //R-TYPE
   6'b100000 : alucontrol <= 3'b010; //add
   6'b100010 : alucontrol <= 3'b011; //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
end
```

3.module controller

a)功能：

该模块整合了上面的两个模块，maindec 和 aludec，得到 datapath 中必要的控制位和 ALU 单元的操作类型

d)代码解析：

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

assign pcsrc = branch & zero;
```

此处值得一提的是，只有 branch 为 1 且 zero 也为 1 时，pcsrc 才会为 1

第三部分：top

此部分将指令加载、数据加载与控制单元和 datapath 组成的 mips 组合起来，

实现完整的简易 CPU 实现

1.module mips

整合了 controller 和 datapath，得到一个 CPU

2.module IM

a)功能：

这个模拟的是指令从磁盘加载到内存的过程，完整的 CPU 需要从磁盘加载到内存中的指令

b)代码解析：

从文件中读取指令，放进 RAM 内存中

```
`timescale 1ns / 1ps
//Instruction memory: 6 bit address get instruction
module IM(address, instruction);
 input [5:0] address;
 output [31:0] instruction;

 reg [31:0] RAM [63:0]; //32 * 64 RAM

 initial begin
  //initial memory
  $readmemh("memfile.txt", RAM);
 end

 assign instruction = RAM[address];
endmodule
```

3.module DM

a)功能：

此处模拟的则是从内存加载数据和在时钟上升沿向内存写入数据

b)代码解析：

```
`timescale 1ns / 1ps

//data memory: 当写使能有效时将数据写入数据寄存器
module DM(clk, we, a, wd, rd);
 input clk, we;
 input [31:0] a, wd; //aluresult, writedata
 output [31:0] rd;

 reg [31:0] RAM[63:0];

 assign rd = RAM[a[31:26]]; //readdata

 always @(posedge clk) begin
  if(we) RAM[a[31:26]] <= wd;
 end
endmodule
```

4.module top

a)功能：

整合了 CPU-mips 指令处理部分和数据内存、指令内存三部分

b)代码解析：

```
`timescale 1ns / 1ps

module top(clk, reset, writedata, dataaddr, memwrite);
 input clk, reset;
 output [31:0] writedata, dataaddr;
 output memwrite;

 wire [31:0] pc, instr, readdata;

 mips mips(clk, reset, pc, instr, memwrite, dataaddr, writedata, readdata);
 IM imem(pc[7:2], instr);
 DM dmem(clk, memwrite, dataaddr, writedata, readdata);

endmodule
```

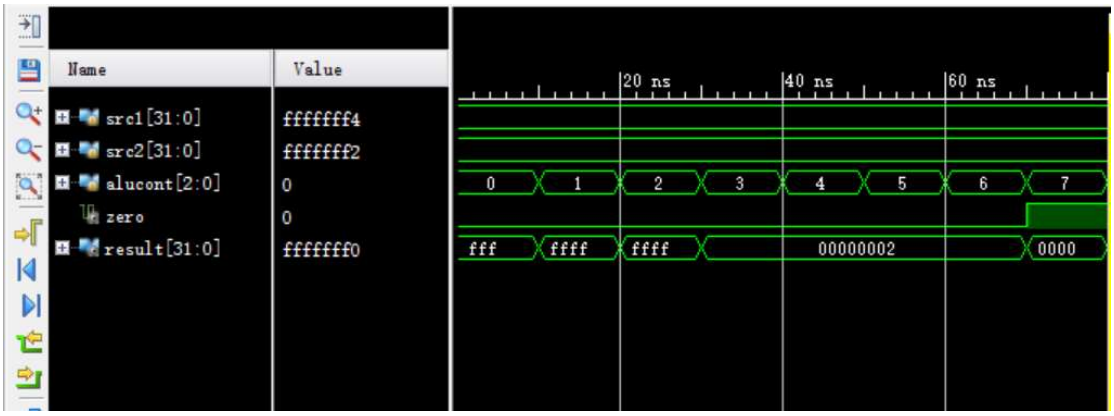
三、仿真波形

1.module alu

两个操作数，0xfffffff4 和 0xfffffff2，初始的 alucont(操作码)为 0，之后每过 10 个时钟周期加一

代码和仿真图：

```
`timescale 1ns / 1ps

module t_alu( );
 reg [31:0] src1, src2;
 reg [2:0] alucont;
 wire zero;
 wire [31:0] result;

 alu test(src1, src2, alucont, result, zero);

 initial begin
  src1 = 32'hFFFFFFF4;
  src2 = 32'hFFFFFFF2;
  alucont = 0;
 end

 always begin
  #10
  alucont = alucont + 1;
 end

endmodule
```

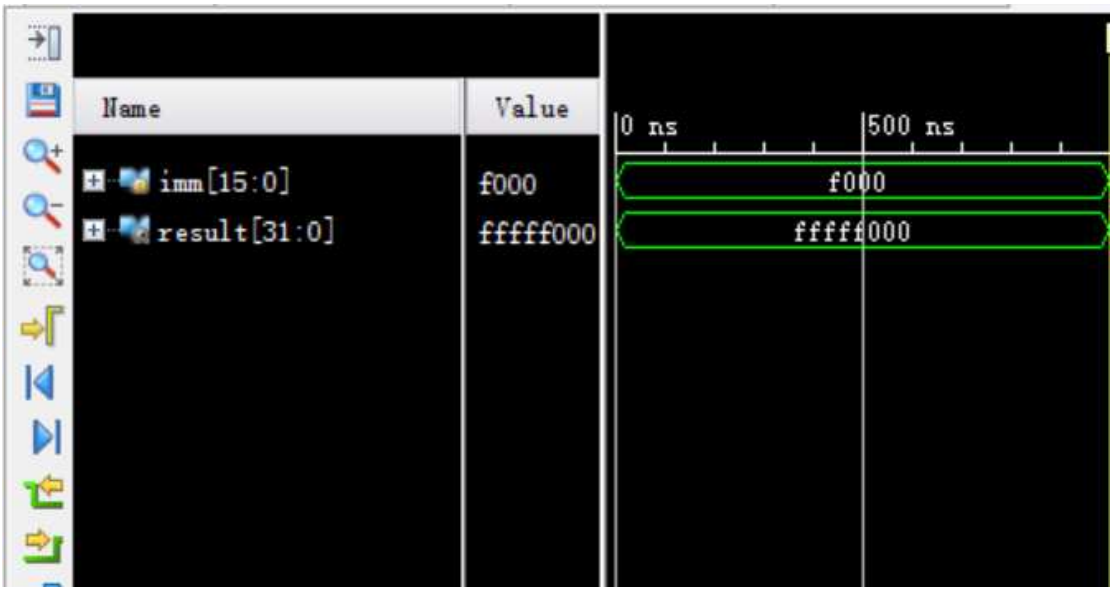


2.module extendNum

对 0xF000 进行扩展：

代码和仿真图：

```
`timescale 1ns / 1ps

module t_extend();
 reg [15:0] imm;
 wire [31:0] result;

 extendNum test(imm, result);

 initial begin
  imm = 'hF000;
 end

endmodule
```

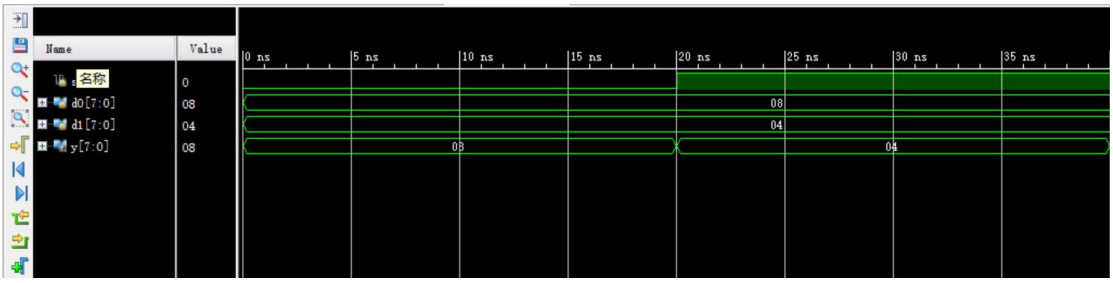


3.module mux2

d0=8，d1=4，s 的初始值为 0，每过 20 个时钟周期，对 s 取反

代码和仿真图：

```
`timescale 1ns / 1ps

module t_mux2( );
 reg s;
 reg [7:0] d0, d1; // parameter is adjustable
 wire [7:0] y;

 mux2 #(8) test(d0, d1, s, y);

 initial begin
  d0 = 8;
  d1 = 4;
  s = 0;
 end

 always begin
  #20
  s = ~s;
 end
endmodule
```



4.module sl2

设置 a 初始值为 0x12344321，每过 10 个时钟周期对其加一，看左移两位生成的 b 随 a 的变化

代码和仿真图：

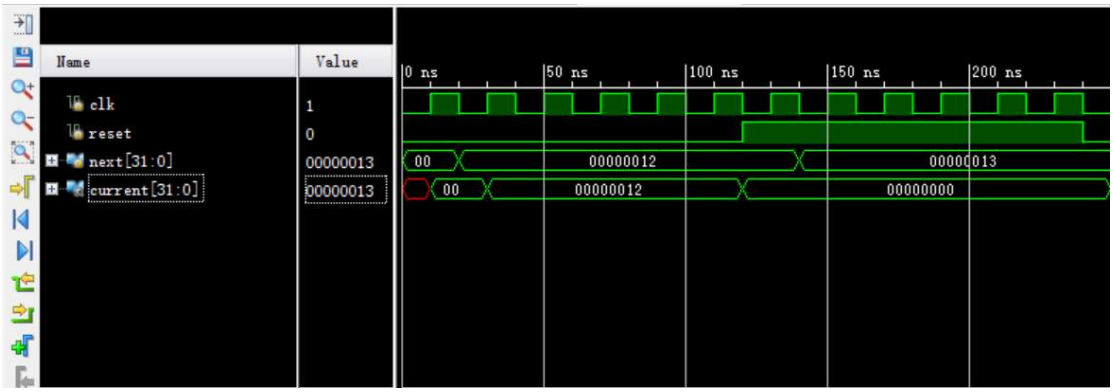
```
`timescale 1ns / 1ps

module t_sl2( );
 reg [31:0] a;
 wire [31:0] b;

 sl2 test(a, b);

 initial begin
  a = 'h12344321;
 end
 always begin
  #10
  a = a + 1;
 end
endmodule
```

5.module PC

初始状态下，clk=0，reset=0，next=0x12343211，每过 10 个时钟周期，对 clk 取反，每过 20 个时钟周期，对 next 加一，100 个时钟周期对 reset 取反

代码和仿真图：

```
PC #(32) a(.clk(clk), .reset(reset), .next(next), .current(current));

initial begin
 clk = 0;
 reset = 0;
 next = 8'h12343211;
end

always begin
 forever #10
 clk = ~clk;
end
always begin
 #20
 next = next + 1;
 #100
 reset = ~reset;
end
```

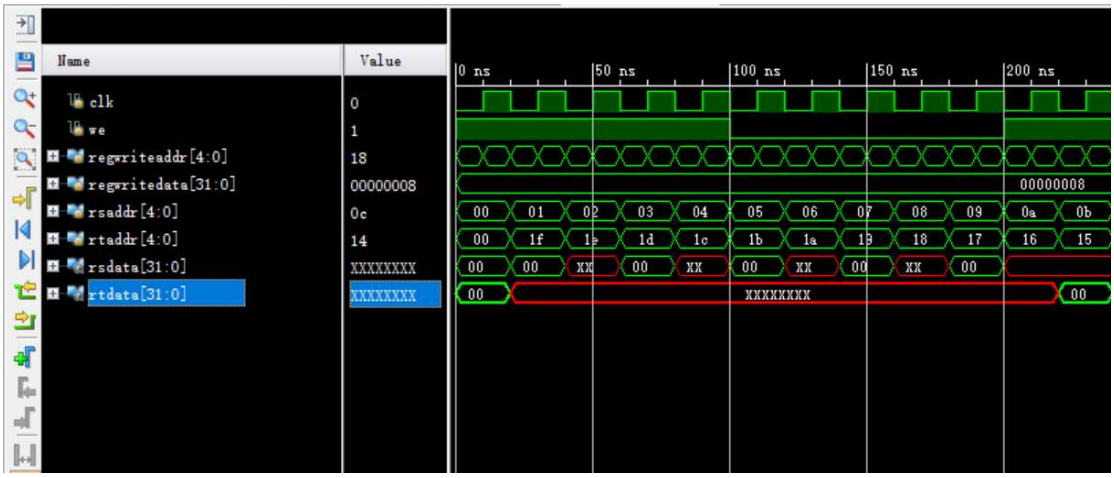


6.module regFile

通过对参数的周期性变化，测试了可写、可读的不同组合情况。

代码和仿真图：

```
`timescale 1ns / 1ps

module t_regfile( );
 reg clk, we;
 reg[4:0] regwriteaddr;
 reg [31:0] regwritedata;
 reg [4:0] rsaddr, rtaddr;
 wire [31:0] rsdata, rtdata;

 regFile test(clk, we, regwriteaddr, regwritedata, rsaddr, rtaddr, rsdata, rtdata);

 initial begin
  clk = 0;
  we = 1;
  regwriteaddr = 'd0;
  regwritedata = 'd8;
  rsaddr = 0;
  rtaddr = 'd32;
 end

 always begin
  #10
  clk = ~clk;
 end
 always begin
  #100
  we = ~we;
 end
 always begin
  #10
  regwriteaddr = regwriteaddr + 1;
 end
 always begin
  #20
  rsaddr = rsaddr + 1;
  rtaddr = rtaddr - 1;
 end

endmodule
```

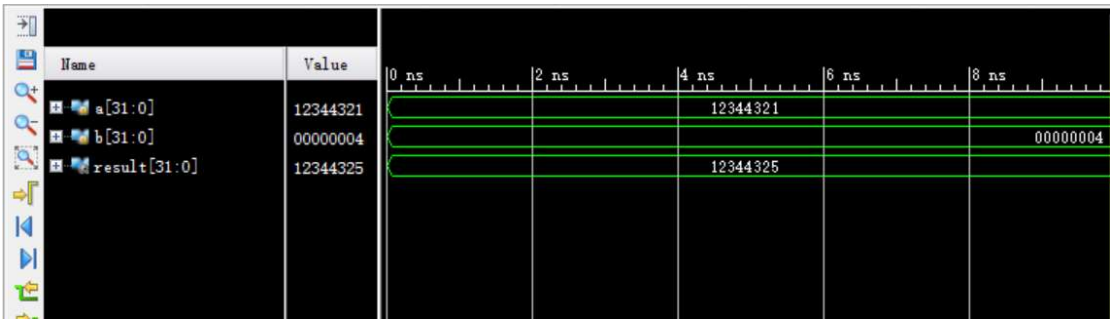


7.module PCPlus4

对 a 和 b 赋初值，每过 10 个时钟周期对 a 加一，查看 result 的变化

代码和仿真图：

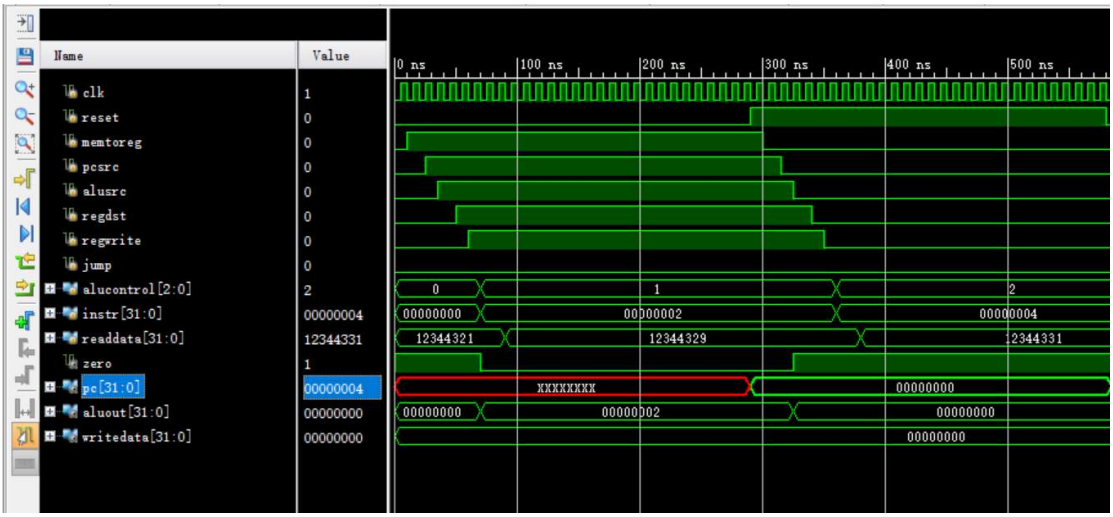
```
`timescale 1ns / 1ps

module t_pcplus4();
 reg [31:0] a, b;
 wire [31:0] result;

 PCPlus4 test(a, b, result);

 initial begin
  a = 'h12344321;
  b = 'd4;
 end
 always begin
  #10
  a = a + 1;
 end
endmodule
```



8.module datapath

datapath 的输入主要是一些使能标志，在测试中，经过特定的时间对这些位取反，具体的测试用例在代码中很清晰，不再赘述：

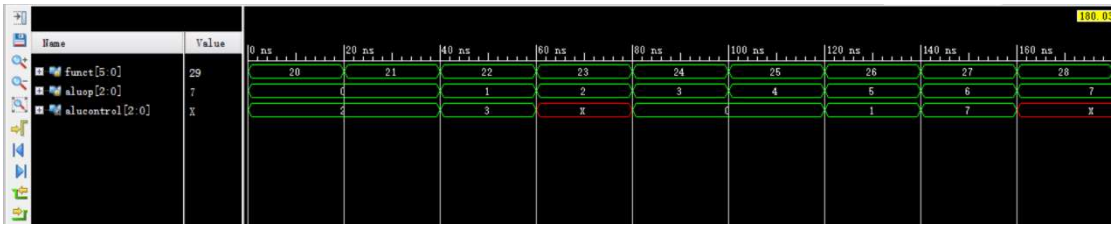
```
module t_datapath( );
 reg clk, reset, memtoreg, pcsrc, alusrc, regdst, regwrite, jump;
 reg [2:0] alucontrol;
 reg [31:0] instr;
 reg [31:0] readdata;
 wire zero;
 wire [31:0] pc, aluout, writedata;

 datapath test(clk, reset, memtoreg, pcsrc, alusrc, regdst, regwrite, jump, alucontrol, zero, pc, instr, aluout, writedata, readdata);

 initial begin
  clk = 0;
  reset = 0;
  memtoreg = 0;
  pcsrc = 0;
  alusrc = 0;
  regdst = 0;
  regwrite = 0;
  jump = 0;
  alucontrol = 0;
  instr = 0;
  readdata = 'h12344321;
 end

 always begin
  #5 clk = ~clk;
 end
 always begin
  #10
  memtoreg = ~memtoreg;
  #15
  pcsrc = ~pcsrc;
  #10 alusrc= ~alusrc;
  #15 regdst = ~regdst;
  #10 regwrite = ~regwrite;
  #10 alucontrol = alucontrol + 1;
     instr = instr + 2;
  #20 readdata = readdata + 8;
  #200 reset = ~reset;
 end
endmodule
```



9.module aludec

测试 aluop 为 0-7 的对应的不同 alucontrol：

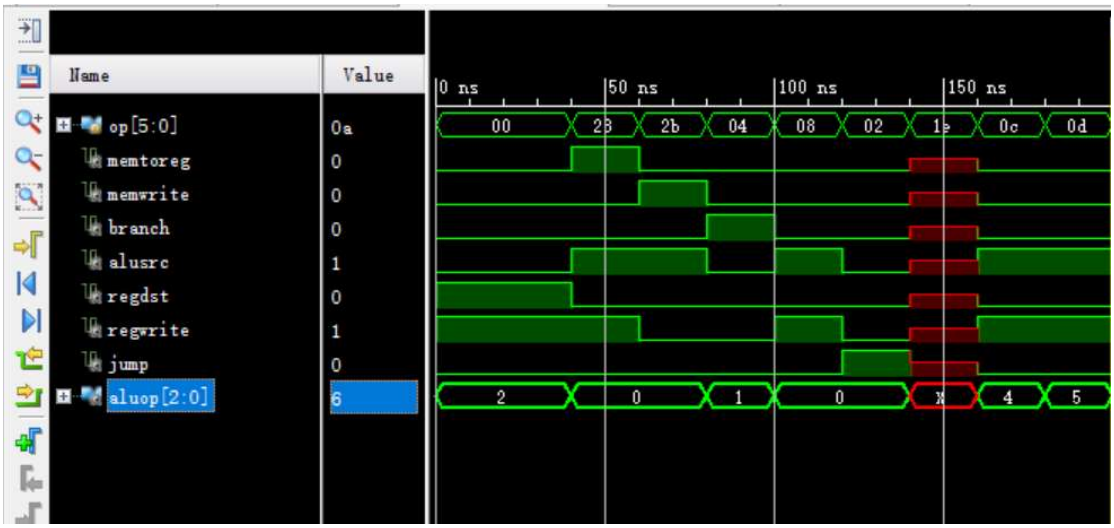
```
module t_aludec();
 reg [5:0] funct;
 reg [2:0] aluop;
 wire [2:0] alucontrol;

 aludec test(funct, aluop, alucontrol);

 initial begin
  funct = 32;
  aluop = 0;
  #20 aluop = 0;
  #20 aluop = 1;
  #20 aluop = 2;
  #20 aluop = 3;
  #20 aluop = 4;
  #20 aluop = 5;
  #20 aluop = 6;
  #20 aluop = 7;
 end
 always begin
  #20
    funct = funct + 1;
 end
endmodule
```



10. module maindec

测试需求文档中不同的 op 对应的控制位输出：

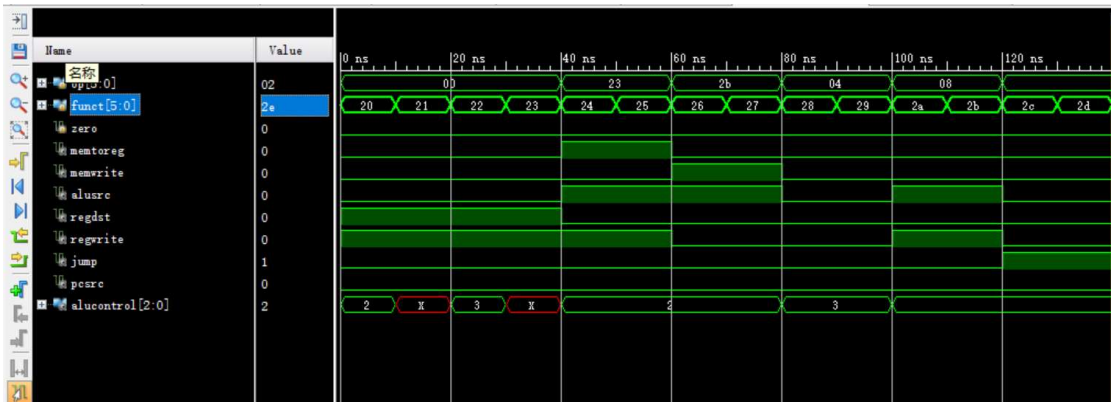
```
`timescale 1ns / 1ps

module t_maindec( );
 reg [5:0] op;
 wire memtoreg, memwrite, branch, alusrc, regdst, regwrite, jump;
 wire [2:0] aluop;

 maindec test(op, memtoreg, memwrite, branch, alusrc, regdst, regwrite, jump, aluop);

 initial begin
  op = 0;
  #20 op = 6'b000000;
  #20 op = 6'b100011;
  #20 op = 6'b101011;
  #20 op = 6'b000100;
  #20 op = 6'b001000;
  #20 op = 6'b000010;
  #20 op = 6'b011110;
  #20 op = 6'b001100;
  #20 op = 6'b001101;
  #20 op = 6'b001010;
 end

endmodule
```



11. module controller

测试 op 不同，funct 不同情况下，不同的控制位输出

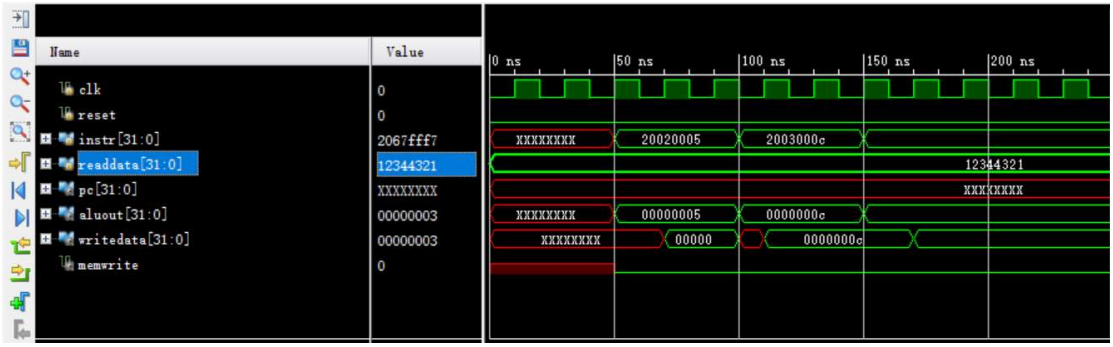
```
module t_controller( );
 reg [5:0] op, funct;
 reg zero;
 wire memtoreg, memwrite, alusrc, regdst, regwrite, jump, pcsrc;
 wire [2:0] alucontrol;

 controller test(op, funct, zero, memtoreg, memwrite, pcsrc, alusrc, regdst, regwrite, jump, alucontrol);
 initial begin
  op = 0;
  zero = 0;
  funct = 32;
  #20 op = 6'b000000;
  #20 op = 6'b100011;
  #20 op = 6'b101011;
  #20 op = 6'b000100;
  #20 op = 6'b001000;
  #20 op = 6'b000010;
 end

 always begin
  #10
  funct = funct + 1;
 end
endmodule
```



12. module mips

取不同的指令，并且每过 10 个时钟周期对 clk 取反

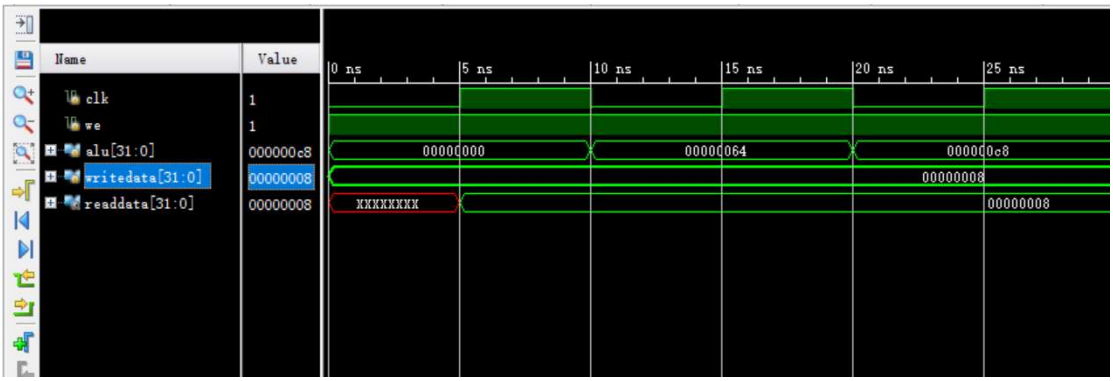
```
`timescale 1ns / 1ps

module t_mips( );
  reg clk, reset;
  reg [31:0] instr, readdata;
  wire [31:0] pc, aluout, writedata;
  wire memwrite;

  mips test(clk, reset, pc, instr, memwrite, aluout, writedata, readdata);

  initial begin
    clk = 0;
    reset = 0;
    readdata = 'h12344321;
    #50 instr = 'h20020005;
    #50 instr = 'h2003000C;
    #50 instr = 'h2067FFF7;
  end

  always begin
    #10 clk = ~clk;
  end
endmodule
```

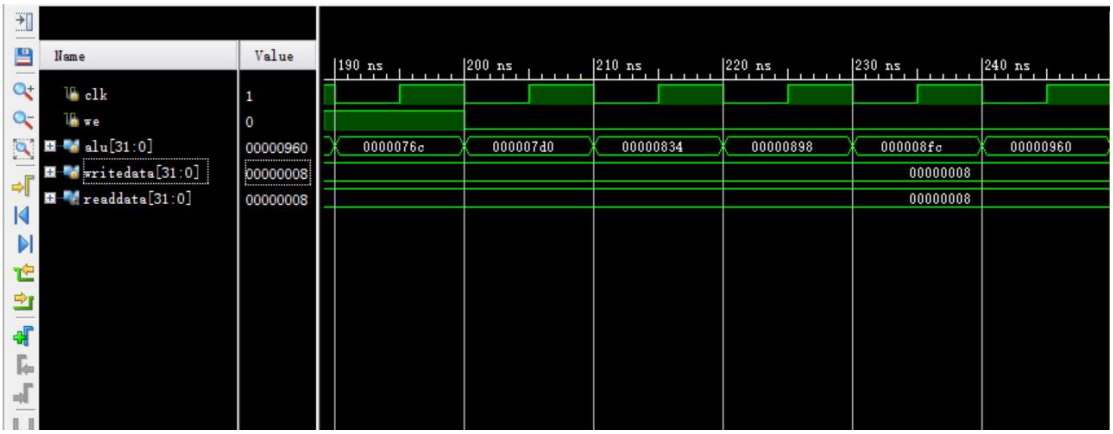


13. module dm

尝试将 8 写入内存并读出来

```
module t_dm();
  reg clk, we;
  reg [31:0] alu, writedata;
  wire [31:0] readdata;

  DM test(clk, we, alu, writedata, readdata);

  initial begin
    clk = 0;
    we = 1;
    alu = 0;
    writedata = 8;
  end
```

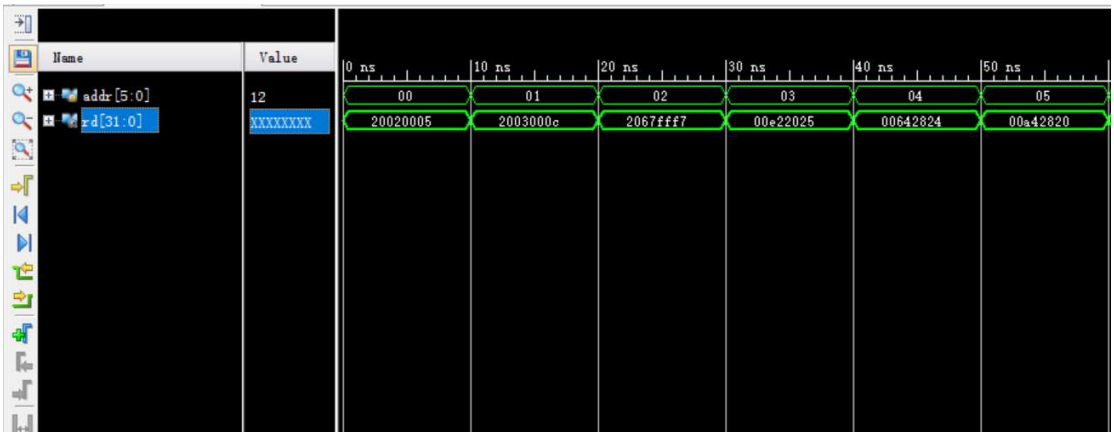




14. module im

测试将 memfile.txt 中的指令加载到内存中

```
`timescale 1ns / 1ps
module t_im();
 reg [5:0] addr;
 wire [31:0] rd;

 IM test(addr, rd);
 initial begin
  addr = 'd0;
 end

 always begin
  #10
  addr = addr + 1;
 end
endmodule
```



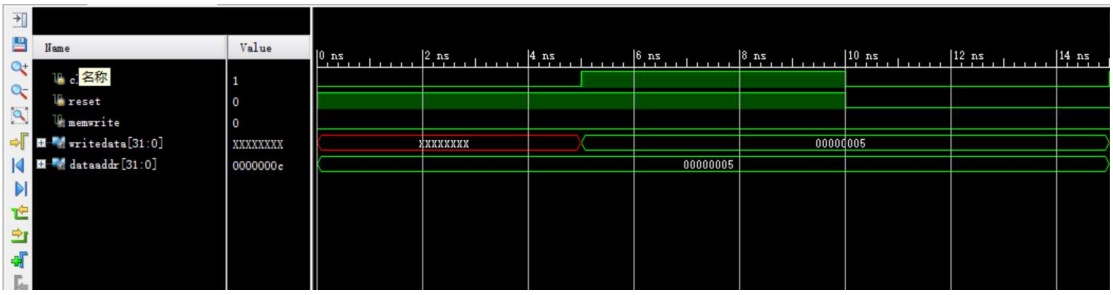
memfile.txt：

```
20020005
2003000C
2067FFF7
00E22025
00642824
00A42820
```

15.module top

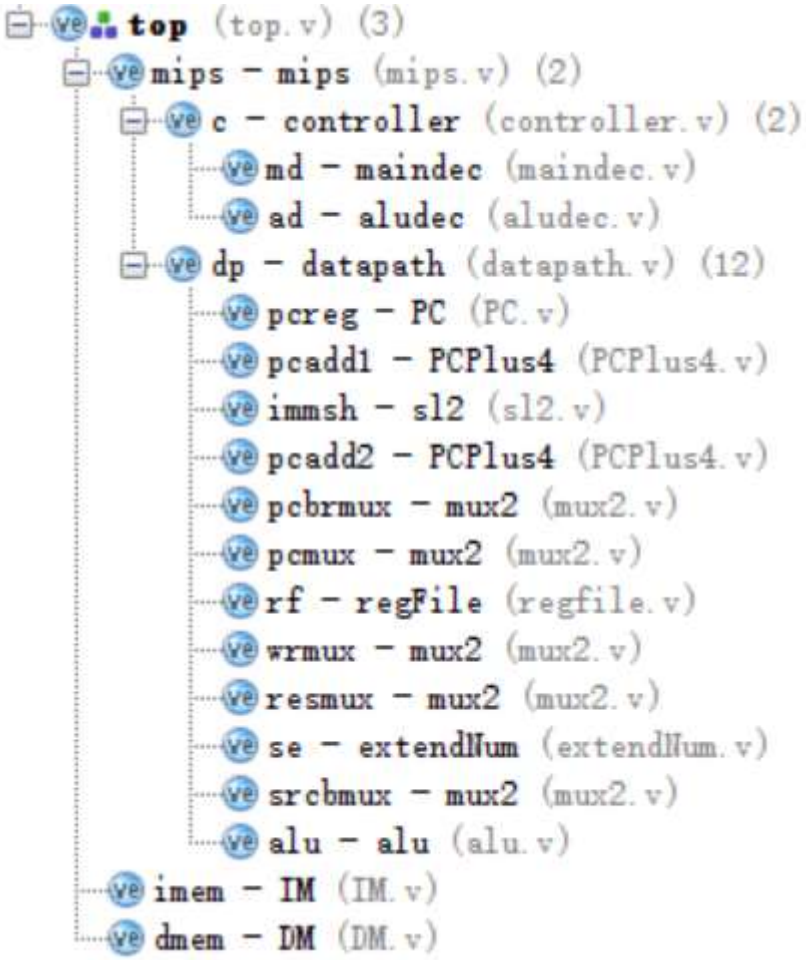
主要测试 reset 和 clk：

```
`timescale 1ns / 1ps

module t_top();
 reg clk, reset;
 wire memwrite;
 wire [31:0] writedata, dataaddr;

 top test(clk, reset, writedata, dataaddr, memwrite);
 initial begin
   clk = 0;
   reset = 1;
   #10
   reset = 0;
 end
 always begin
  #5 clk = ~clk;
  end
endmodule
```



四、实验总结：

1.本次实验结合了上课所讲的设计方法和文档中给出的设计参考，采用了层次化的设计，这点从项目的目录结构也能看出来：



每一部分都是分为了数据层和控制层，数据层包括 datapath、DM 和 IM，功能分别是数据进行处理、从内存加载或读取数据和将磁盘文件中的指令加载到内存中；控制层包括 controller，主要负责读取指令之后的处理，协调各个功能单元完成功能。

2.本次实验中涉及的许多功能单元的设计都是之前 LAB 中做过的，此次 LAB 主要实现的就是如何利用控制单元协调这些单元工作，利用层次化设计，不仅使设计更为简单，也让实现的过程更加的清晰明确

五、实验中的收获与感受：

1.通过此次实验，更加深入的了解了 CPU 是如何工作的

2.对 verilog 语言运用的更加自如，比如在此次实验中用到了新的数据类型 parameter

3.对数字部件设计的思想、流程有了更加深入的体会

六、对实验的反思：

1.我所实现的 CPU 基本上是按照需求文档的标准来实现的，没有做过多的优化，但是根据之前 ICS 课上以及本学期 CSE 课上所讲到的内容，我联想到是否可以利用流水线来优化 CPU 的性能，但是因为时间原因，没有具体的实现

2.此次实验，代码部分很早就完成了，但是文档迟迟没有动工，这就导致写文档有些不流畅，也耽误了老师和 TA 的时间，我对此深刻检讨