实验三：

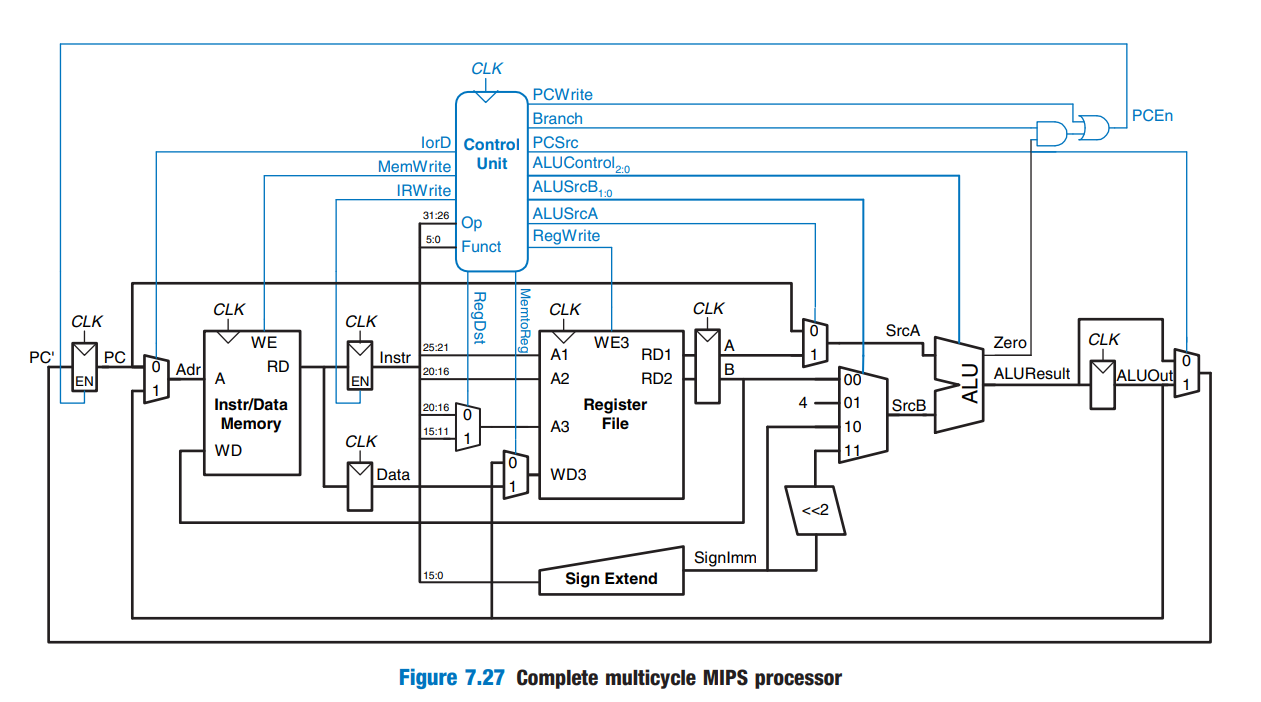
32位MIPS多周期处理器的设计

1. 多周期CPU与仿真：

a.实验方案：

多周期CPU将整个CPU执行过程分成多个阶段，每个阶段用一个时钟完成，然后开始下一条指令的执行，每种指令执行所用时钟数不同，故需暂存每个时钟周期下的中间状态。其中，由于状态改变发生在两个寄存器间，每一步需要增加寄存器进行保存。

根据教材，指令框架如下：



下面分析各条指令：

1. lw:

S0取指令+更新PC：IR指令寄存器受使能端（异步reset）控制；PC通过加法器递增

S1译码：对指令进行译码，同时，对立即数进行符号拓展

S2计算地址：使用alucontrol来控制ALU

S3读存储器：将lord设置为1读取存储器并且存入data寄存器

S4写寄存器：将data写入寄存器文件

1. Sw:

S0->S1->S2

S5写存储器：将寄存器取出的数据存在存储器中

1. R type:

S0->S1

S6执行：传入两个寄存器值，根据译码所得ALUControl值决定ALU的计算操作

S7写回：将ALU计算结果写入目的寄存器（RegDst=0）

1. Beq:

S0->S1

S8分支：通过ALU计算BEQ分支地址（PCSrc=01）

1. Addi:

S0->S1

S9计算：传入寄存器值与立即数，通过ALU进行加法运算

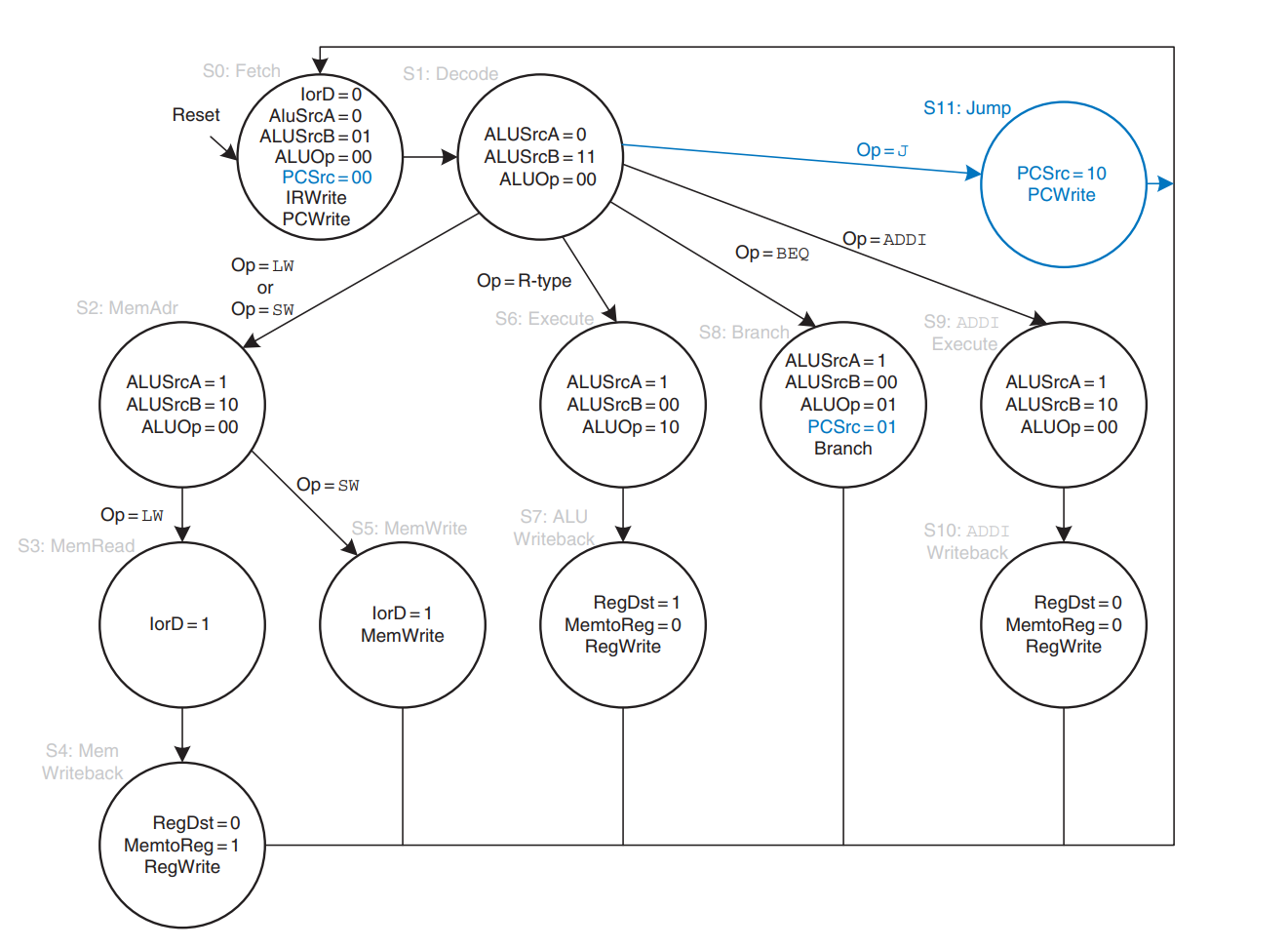
S10写寄存器：将ALU计算结果写入目的寄存器（RegDst=0）

1. Jump:

S0->S1

S11跳转：读取并拼接跳转地址，进行下一PC跳转

综上，状态转换图如下：



需要在单周期CPU基础上改进主译码器和基本数据通路。

其中，控制信号基于指令的opcode字段（Instr31:26）和funct字段（Instr5:0）进行控制译码。主译码器采用FSM方式表达状态转换，以及每个状态涉及的相关部件指令。

由于每条指令在不同时钟周期进行表达，故将imem和dmem合二为一，改写成统一的mem。

同时，不是每个周期每个寄存器都需进行写入，故将所有寄存器分为带使能端和不带使能端。

完成CPU通用核心模块的构造后，同样编写MIPS基准测试程序进行指令检查。将指令对应的机器代码放入十六进制文件memfile.dat中，由mem指令存储器进行读取并传入执行。

b.关键代码：

1.主译码器

module maindec(

input logic clk,

input logic reset,

input logic [5:0] op,

output logic pcwrite, memwrite, irwrite, regwrite,

output logic alusrca, branch, iord, memtoreg, regdst,

output logic [1:0] alusrcb, pcsrc, aluop);

localparam FETCH = 4'b0000; //State 0

localparam DECODE = 4'b0001; //State 1

localparam MEMADR = 4'b0010; //State 2

localparam MEMRD = 4'b0011; //State 3

localparam MEMWB = 4'b0100; //State 4

localparam MEMWR = 4'b0101; //State 5

localparam RTYPEEX = 4'b0110; //State 6

localparam RTYPEWB = 4'b0111; //State 7

localparam BEQEX = 4'b1000; //State 8

localparam ADDIEX = 4'b1001; //State 9

localparam ADDIWB = 4'b1010; //State 10

localparam JEX = 4'b1011; //State 11

localparam LW = 6'b100011; //Opcode for lw

localparam SW = 6'b101011; //Opcode for sw

localparam RTYPE = 6'b000000; //Opcode for R-type

localparam BEQ = 6'b000100; //Opcode for beg

localparam ADDI = 6'b001000; //Opcode for addi

localparam J = 6'b000010; //Opcode for j

logic [3:0] state, nextstate;

logic [14:0] controls;

//state register

always\_ff @(posedge clk or posedge reset)

if(reset) state <= FETCH;

else state <= nextstate;

//next state logic

always\_comb

case(state)

FETCH: nextstate = DECODE;

DECODE: case(op)

LW: nextstate = MEMADR;

SW: nextstate = MEMADR;

RTYPE: nextstate = RTYPEEX;

BEQ: nextstate = BEQEX;

ADDI: nextstate = ADDIEX;

J: nextstate = JEX;

default: nextstate = 4'bx;

endcase

MEMADR: case(op)

LW: nextstate = MEMRD;

SW: nextstate = MEMWR;

default: nextstate = 4'bx;

endcase

MEMRD: nextstate = MEMWB;

MEMWB: nextstate = FETCH;

MEMWR: nextstate = FETCH;

RTYPEEX: nextstate = RTYPEWB;

RTYPEWB: nextstate = FETCH;

BEQEX: nextstate = FETCH;

ADDIEX: nextstate = ADDIWB;

ADDIWB: nextstate = FETCH;

JEX: nextstate = FETCH;

default: nextstate = 4'bx;

endcase

//output logic

assign {pcwrite, memwrite, irwrite, regwrite,

alusrca, branch, iord, memtoreg, regdst,

alusrcb, pcsrc, aluop} = controls;

always\_comb

case(state)

FETCH: controls = 15'h5010;

DECODE: controls = 15'h0030;

MEMADR: controls = 15'h0420;

MEMRD: controls = 15'h0100;

MEMWB: controls = 15'h0880;

MEMWR: controls = 15'h2100;

RTYPEEX: controls = 15'h0402;

RTYPEWB: controls = 15'h0840;

BEQEX: controls = 15'h0605;

ADDIEX: controls = 15'h0420;

ADDIWB: controls = 15'h0800;

JEX: controls = 15'h4008;

default: controls = 15'hxxxx;

endcase

endmodule

2.基本数据通路：

module datapath(

input logic clk, reset,

//from control

input logic iord, irwrite,

input logic memtoreg, regdst,

input logic pcen,

input logic [1:0] pcsrc,

input logic regwrite,

input logic [2:0] alucontrol,

input logic alusrca,

input logic [1:0] alusrcb,

output logic zero,

output logic [5:0] op, funct,

//from imem/dmem

input logic [31:0] rd, //readdata

output logic [31:0] adr,

output logic [31:0] wd);//writedata

//logic

logic [31:0] pcnext, pc;

logic [31:0] instr, data;

logic [31:0] signimm, signimmsh;

logic [31:0] wd3, rd1, rd2;

logic [4:0] a3;

logic [27:0] jumpadr;

logic [31:0] pcjump;

logic [31:0] srca, srcb, a, b;

logic [31:0] aluresult, aluout;

//control

assign op = instr[31:26];

assign funct = instr[5:0];

//mem

assign wd = b;

//next PC logic

muxpc #(32) pcout(aluresult, aluout, pcjump, pcsrc, pcnext);

flopenr #(32) pcreg(clk, reset, pcen, pcnext, pc); //pc更新成pcnext

mux2 #(32) pcadr(pc, aluout, iord, adr);

//instruction or data

flopenr #(32) instrfp(clk, reset, irwrite, rd, instr);

flopr #(32) datafp(clk, reset, rd, data);

//register file logic

regfile rf(clk, regwrite, instr[25:21], instr[20:16],

a3, wd3, rd1, rd2);

flopr2 #(32) rdfp(clk, reset, rd1, rd2, a, b);

mux2 #(5) wrmux(instr[20:16], instr[15:11],

regdst, a3);

mux2 #(32) wdmux(aluout, data, memtoreg, wd3);

//imm

signext se(instr[15:0], signimm);

sl2 immsh(signimm, signimmsh);

//jump

assign jumpadr = {instr[25:0], 2'b00};

assign pcjump = {pc[31:28], jumpadr};

//ALU logic

mux2 #(32) srcamux(pc, a, alusrca, srca); //srca

muxb #(32) srcbmux(b,signimm, signimmsh, alusrcb, srcb); //srcb

alu alu(srca, srcb, alucontrol, aluresult, zero);

flopr #(32) alufp(clk, reset, aluresult, aluout);

endmodule

3.mem：

module mem(

input logic clk,

input logic we,

input logic [31:0] a, wd, //adr, writedata

output logic [31:0] rd);

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

initial

$readmemh("memfile.dat", RAM);

//read组合逻辑

assign rd = RAM[a[31:2]];

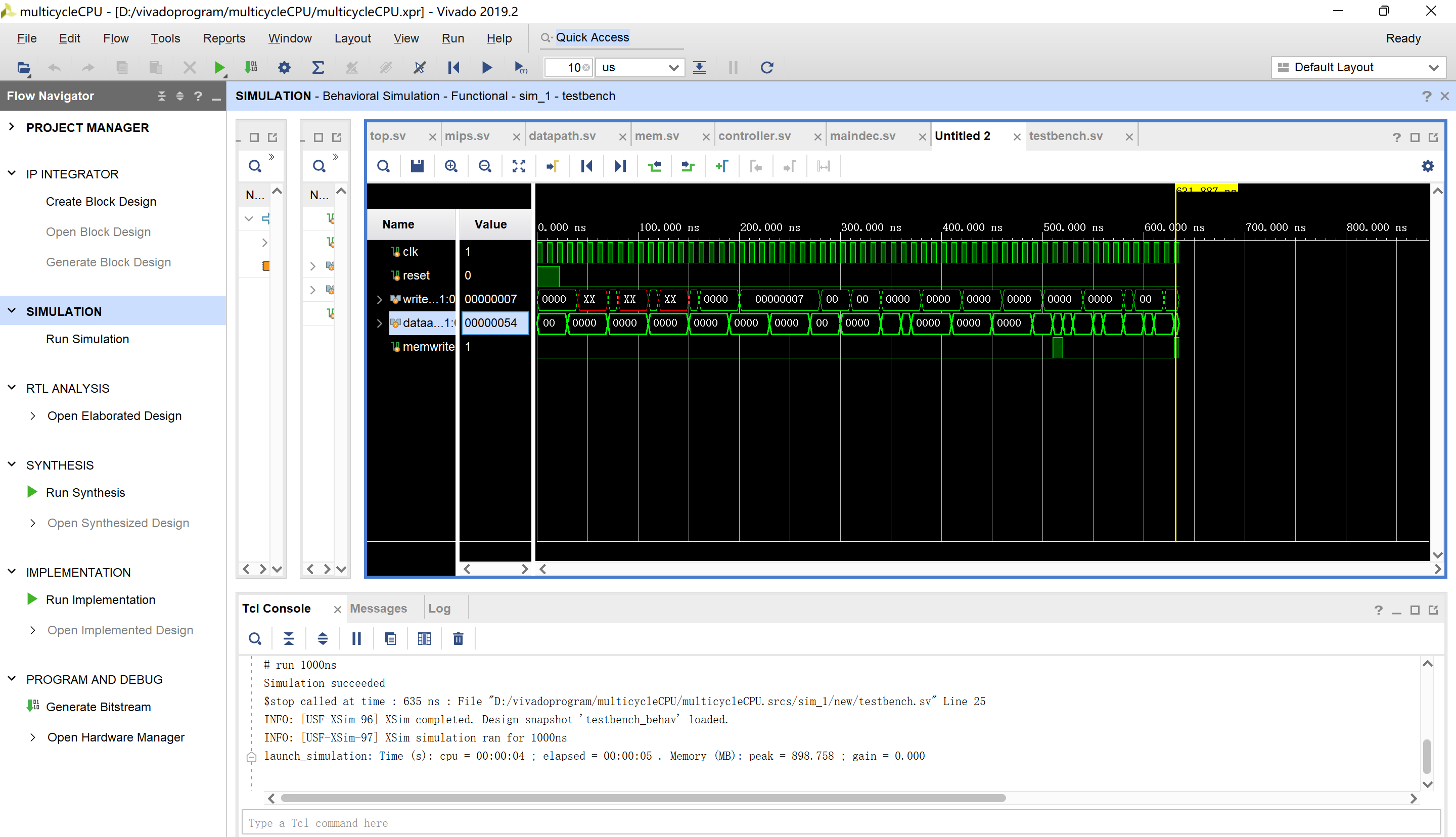
//write时序逻辑

always\_ff @(posedge clk)

if (we) RAM[a[31:2]] <= wd;

endmodule

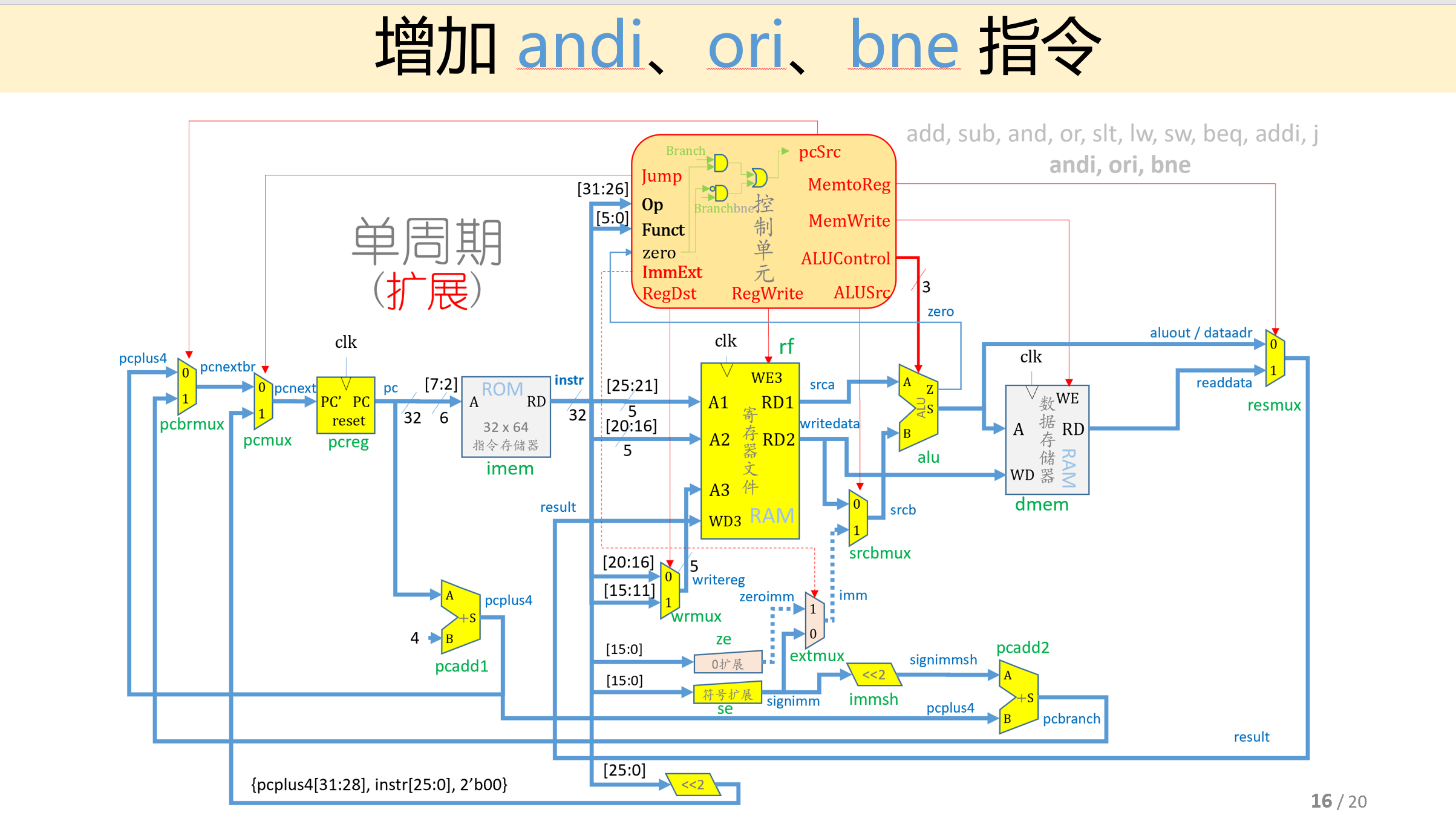
3．仿真：



仿真成功。

1. 多周期CPUext与仿真
2. 实验方案:

在原有指令的基础上添加andi，ori，bne指令，修改参考单周期CPU：



对比增添指令前后的模块框图，发现主要改动在于：1）控制单元信号位数扩展，主译码器增添ImmExt，Branch Bne两位，ALU译码器的控制信号ALUOp位数增加，增加分支指令内部判断逻辑；2）立即数扩展增加0扩展，复用器进行选择，当立即数被用于执行andi、ori指令时，需要进行0扩展；被用于执行addi指令时，需要进行符号扩展。

1. bne指令：

S0->S1

S12分支：通过ALU计算BEQ分支地址（PCSrc=01）同时将branchbne设置为1

1. andi指令：

S0->S1

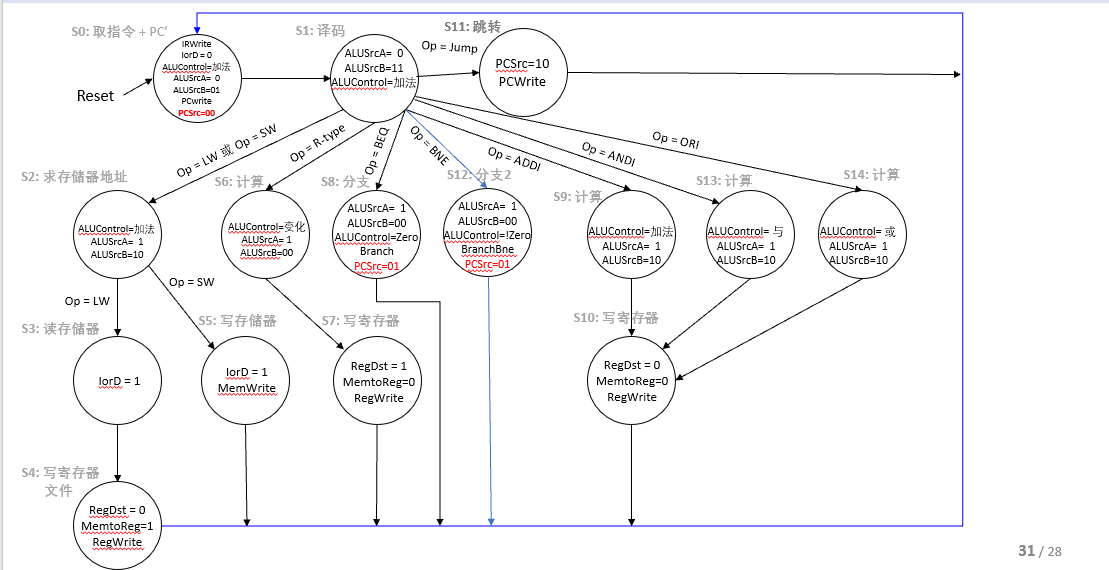
S13执行：传入一个寄存器值和立即数0拓展，执行and操作

S10

1. ori指令：

S14执行：传入一个寄存器值和立即数0拓展，执行or操作

增添指令后的状态转换图如下：



增加3个状态。

在原有代码上将aluop由2位扩展为3位，controls由15位扩展为18位，增加signzero零扩展模块，以及pcsrc复用器模块。

1. 关键代码
2. 主译码器

module maindec(

input logic clk,

input logic reset,

input logic [5:0] op,

output logic pcwrite, memwrite, irwrite, regwrite,

output logic alusrca, branch, iord, memtoreg, regdst,

output logic immext, branchbne,

output logic [1:0] alusrcb, pcsrc,

output logic [2:0] aluop);

localparam FETCH = 4'b0000; //State 0

localparam DECODE = 4'b0001; //State 1

localparam MEMADR = 4'b0010; //State 2

localparam MEMRD = 4'b0011; //State 3

localparam MEMWB = 4'b0100; //State 4

localparam MEMWR = 4'b0101; //State 5

localparam RTYPEEX = 4'b0110; //State 6

localparam RTYPEWB = 4'b0111; //State 7

localparam BEQEX = 4'b1000; //State 8

localparam ADDIEX = 4'b1001; //State 9

localparam ITYPEWB = 4'b1010; //State 10

localparam JEX = 4'b1011; //State 11

localparam BNEEX = 4'b1100; //state 12

localparam ORIEX = 4'b1101; //state 13

localparam ANDIEX = 4'b1110; //state 14

localparam LW = 6'b100011; //Opcode for lw

localparam SW = 6'b101011; //Opcode for sw

localparam RTYPE = 6'b000000; //Opcode for R-type

localparam BEQ = 6'b000100; //Opcode for beg

localparam ADDI = 6'b001000; //Opcode for addi

localparam J = 6'b000010; //Opcode for j

localparam BNE = 6'b000101; //Opcode for bne

localparam ORI = 6'b001101; //Opcode for ori

localparam ANDI = 6'b001100; //Opcode for andi

logic [3:0] state, nextstate;

logic [17:0] controls;

//state register

always\_ff @(posedge clk or posedge reset)

if(reset) state <= FETCH;

else state <= nextstate;

//next state logic

always\_comb

case(state)

FETCH: nextstate = DECODE;

DECODE: case(op)

LW: nextstate = MEMADR;

SW: nextstate = MEMADR;

RTYPE: nextstate = RTYPEEX;

BEQ: nextstate = BEQEX;

ADDI: nextstate = ADDIEX;

J: nextstate = JEX;

BNE: nextstate = BNEEX;

ORI: nextstate = ORIEX;

ANDI: nextstate = ANDIEX;

default: nextstate = 4'bx;

endcase

MEMADR: case(op)

LW: nextstate = MEMRD;

SW: nextstate = MEMWR;

default: nextstate = 4'bx;

endcase

MEMRD: nextstate = MEMWB;

MEMWB: nextstate = FETCH;

MEMWR: nextstate = FETCH;

RTYPEEX: nextstate = RTYPEWB;

RTYPEWB: nextstate = FETCH;

BEQEX: nextstate = FETCH;

ADDIEX: nextstate = ITYPEWB;

ITYPEWB: nextstate = FETCH;

JEX: nextstate = FETCH;

BNEEX: nextstate = FETCH;

ORIEX: nextstate = ITYPEWB;

ANDIEX: nextstate = ITYPEWB;

default: nextstate = 4'bx;

endcase

//output logic

assign {pcwrite, memwrite, irwrite, regwrite,

alusrca, branch, iord, memtoreg, regdst,

alusrcb, pcsrc, aluop, immext, branchbne} = controls;

always\_comb

case(state)

FETCH: controls = 18'h28080; //18'b10 1000 0000 1000 0000

DECODE: controls = 18'h00180; //18'b00 0000 0001 1000 0000

MEMADR: controls = 18'h02100; //18'b00 0010 0001 0000 0000

MEMRD: controls = 18'h00800; //18'b00 0000 1000 0000 0000

MEMWB: controls = 18'h04400; //18'b00 0100 0100 0000 0000

MEMWR: controls = 18'h10800; //18'b01 0000 1000 0000 0000

RTYPEEX: controls = 18'h02008; //18'b00 0010 0000 0000 1000

RTYPEWB: controls = 18'h04200; //18'b00 0100 0010 0000 0000

BEQEX: controls = 18'h03024; //18'b00 0011 0000 0010 0100

ADDIEX: controls = 18'h02100; //18'b00 0010 0001 0000 0000

ITYPEWB: controls = 18'h04000; //18'b00 0100 0000 0000 0000

JEX: controls = 18'h20040; //18'b10 0000 0000 0100 0000

BNEEX: controls = 18'h02025; //18'b00 0010 0000 0010 0101

ORIEX: controls = 18'h0210e; //18'b00 0010 0001 0000 1110

ANDIEX: controls = 18'h02112; //18'b00 0010 0001 0001 0010

default: controls = 18'hxxxxx;

endcase

endmodule

1. 基础数据通路

module datapath(

input logic clk, reset,

//from control

input logic iord, irwrite,

input logic memtoreg, regdst,

input logic pcen,

input logic [1:0] pcsrc,

input logic regwrite,

input logic [2:0] alucontrol,

input logic alusrca,

input logic [1:0] alusrcb,

input logic immext, //add

output logic zero,

output logic [5:0] op, funct,

//from imem/dmem

input logic [31:0] rd, //readdata

output logic [31:0] adr,

output logic [31:0] wd);//writedata

//logic

logic [31:0] pcnext, pc;

logic [31:0] instr, data;

logic [31:0] signimm, zeroimm, signimmsh, imm;

logic [31:0] wd3, rd1, rd2;

logic [4:0] a3;

logic [27:0] jumpadr;

logic [31:0] pcjump;

logic [31:0] srca, srcb, a, b;

logic [31:0] aluresult, aluout;

//control

assign op = instr[31:26];

assign funct = instr[5:0];

//mem

assign wd = b;

//next PC logic

muxpc #(32) pcout(aluresult, aluout, pcjump, pcsrc, pcnext);

flopenr #(32) pcreg(clk, reset, pcen, pcnext, pc); //pc更新成pcnext

mux2 #(32) pcadr(pc, aluout, iord, adr);

//instruction or data

flopenr #(32) instrfp(clk, reset, irwrite, rd, instr);

flopr #(32) datafp(clk, reset, rd, data);

//register file logic

regfile rf(clk, regwrite, instr[25:21], instr[20:16],

a3, wd3, rd1, rd2);

flopr2 #(32) rdfp(clk, reset, rd1, rd2, a, b);

mux2 #(5) wrmux(instr[20:16], instr[15:11],

regdst, a3);

mux2 #(32) wdmux(aluout, data, memtoreg, wd3);

//imm

signext se(instr[15:0], signimm); //sign-extension

signzero ze(instr[15:0], zeroimm); //0-extension

sl2 immsh(signimm, signimmsh);

mux2 #(32) extmux(signimm, zeroimm, immext, imm);

//jump

assign jumpadr = {instr[25:0], 2'b00};

assign pcjump = {pc[31:28], jumpadr};

//ALU logic

mux2 #(32) srcamux(pc, a, alusrca, srca); //srca

muxb #(32) srcbmux(b,imm, signimmsh, alusrcb, srcb); //srcb

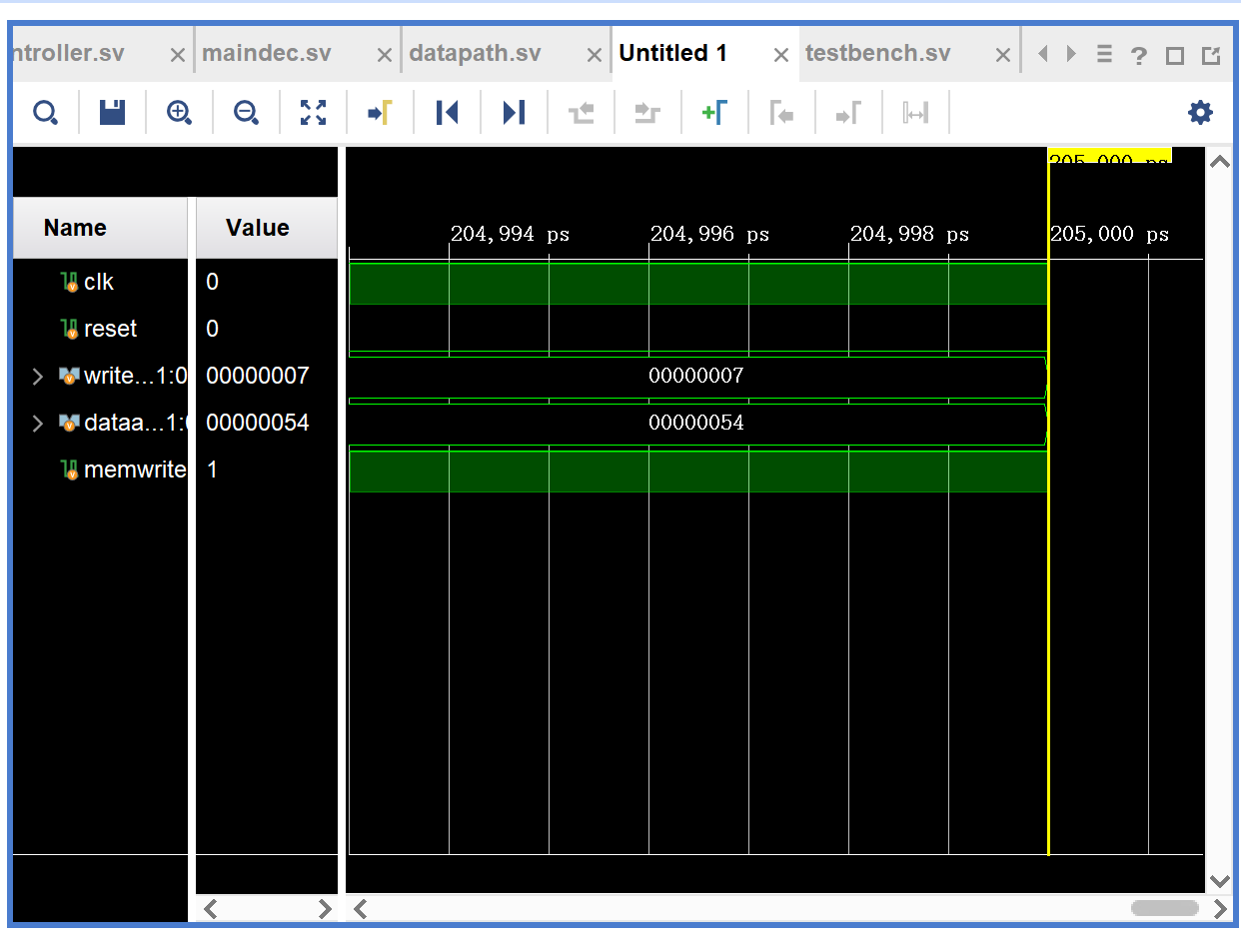
alu alu(srca, srcb, alucontrol, aluresult, zero);

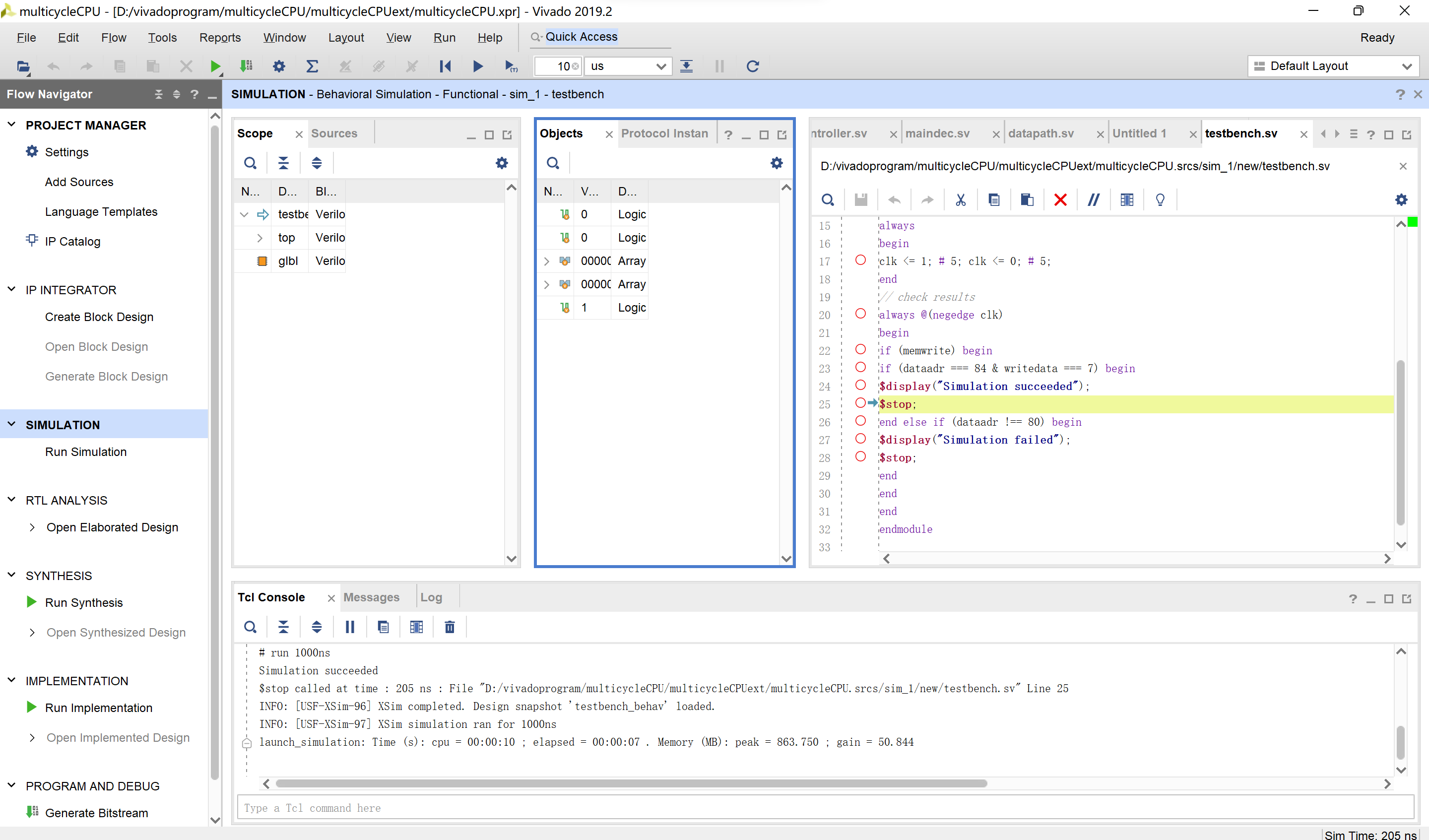
flopr #(32) alufp(clk, reset, aluresult, aluout);

endmodule

1. 仿真

按照单周期的仿真进行：





1. 多周期CPU\_IO：仿真+上板
2. 实验方案：

与单周期IO相同

1. 关键代码：

module dmdecoder(

input logic clk,

input logic writeEN,

input logic [31:0] addr, //32bit

input logic [31:0] writeData,

output logic [31:0] readData,

input logic reset,

input logic btnL, btnR,

input logic [15:0] switch,

output logic [7:0] an,

output logic dp,

output logic [6:0] a2g);

logic [11:0] led;

logic [31:0] digit;

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

logic [31:0] ReadData1, ReadData2;

logic pRead, pWrite, mWrite;

assign pRead = (addr[7] == 1'b1) ? 1:0;

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

assign mWrite = writeEN & (addr[7] == 1'b0);

mux2 #(32) rdmux(.d0(ReadData1),

.d1(ReadData2),

.s(addr[7]),

.y(readData));

mem dMemory(.clk(clk),

.we(mWrite),

.a(addr),

.wd(writeData),

.rd(ReadData1));

io IO(.clk(clk),

.reset(reset),

.pRead(pRead),

.pWrite(pWrite),

.addr(addr[3:2]),

.pWriteData(writeData[11:0]),

.pReadData(ReadData2),

.buttonL(btnL),

.buttonR(btnR),

.switch(switch),

.led(led));

mux7seg m7seg(.x(digit),

.clk(clk),

.a2g(a2g),

.an(an),

.dp(dp));

endmodule

module io(

//与CPU相连

input logic clk,

input logic reset,

input logic pRead,

input logic pWrite,

input logic [1:0] addr,

input logic [11:0] pWriteData,

output logic [31:0] pReadData,

//与外设相连

input logic buttonL,

input logic buttonR,

input logic [15:0] switch, //control

output logic [11:0] led); //digital

logic [1:0] status;

logic [15:0] switchl;

logic [11:0] ledl;

always\_ff @(posedge clk) begin

if(reset) begin

status <= 2'b00;

ledl <= 12'h00;

switchl <= 16'h00;

end

else begin

//开关位置拨号，可以输入

if(buttonR) begin

status[1] <= 1;

switchl <= switch;

end

//led显示准备好，可以输出

if(buttonL) begin

status[0] <= 1;

led <= ledl;

end

//向数据输出端口输出（led）

if(pWrite & (addr==2'b01)) begin

ledl <= pWriteData;

status[0] <= 0;

end

end

end

always\_comb

if(pRead)

case(addr)

2'b11: pReadData = {24'b0, switchl[15:8]};

2'b10: pReadData = {24'b0, switchl[7:0]};

2'b00: pReadData = {24'b0, 6'b0, status};

default: pReadData = 32'b0;

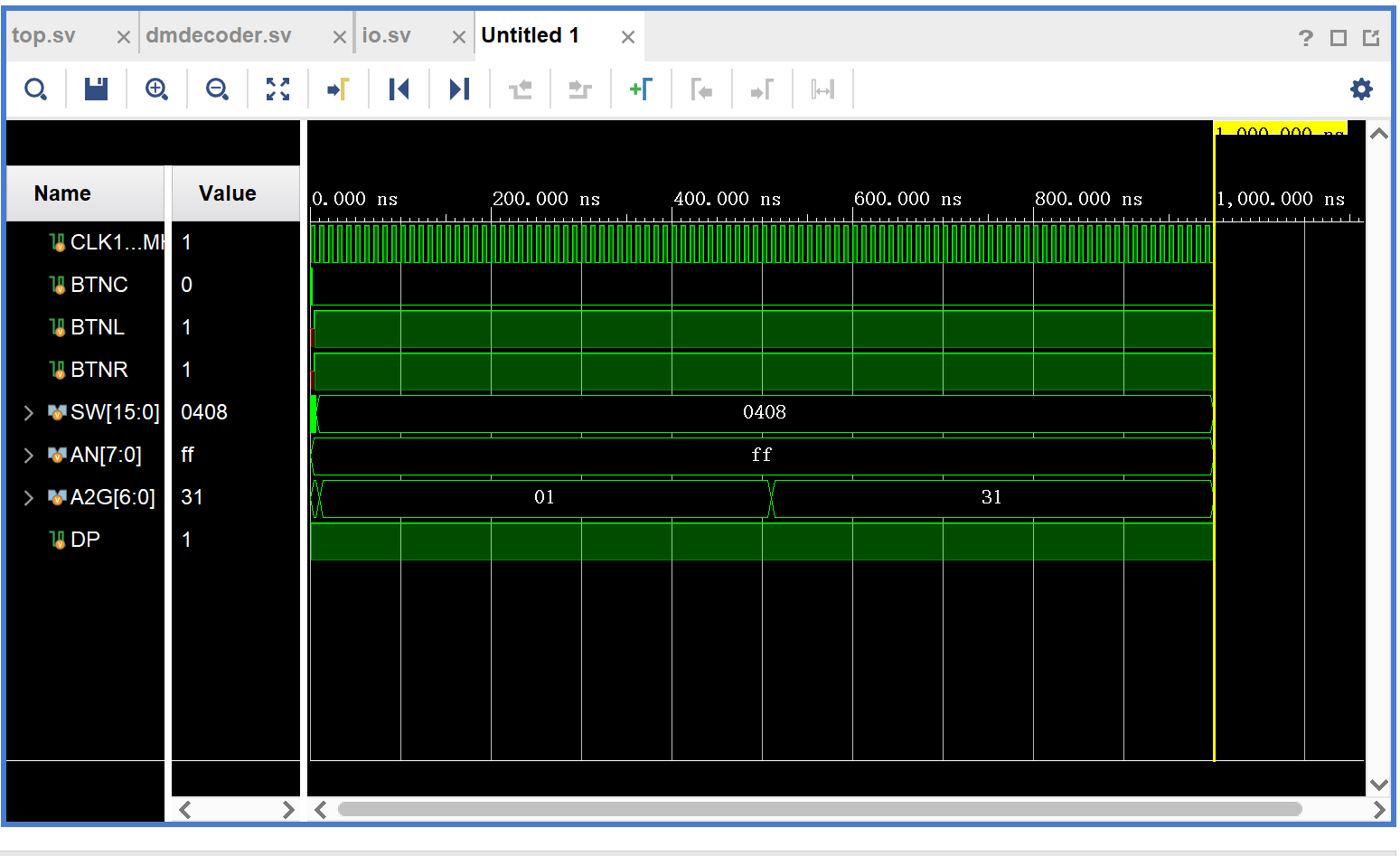
endcase

else

pReadData = 32'b0;

endmodule

1. 仿真结果：



1. 上板结果：



1. 总结反思

在进行第一步时，出现了代码大小写输入错误，但没有报错，导致一直没有出现simulation succeed！还有位数出错，但会在Tcl Console出现warning，改正即可。

本次实验主要考察了在单周期基础上对于多周期CPU原理的进一步认识，其中涉及了对FSM状态转换的理解。由于多周期CPU是同一指令涉及多个时钟周期，其中通用模块被用于不同的指令阶段，如：ALU、mem等。在进行实验前期构思设计时，需遵照状态转换依次厘清每个指令所经过的状态，以及操作、控制模块间的逻辑。

 通过本次实验，巩固了计算机组成的基础硬件知识与基本逻辑。相较单周期CPU而言，多周期CPU因逻辑问题导致的错误较少，调试时间也大大减少。