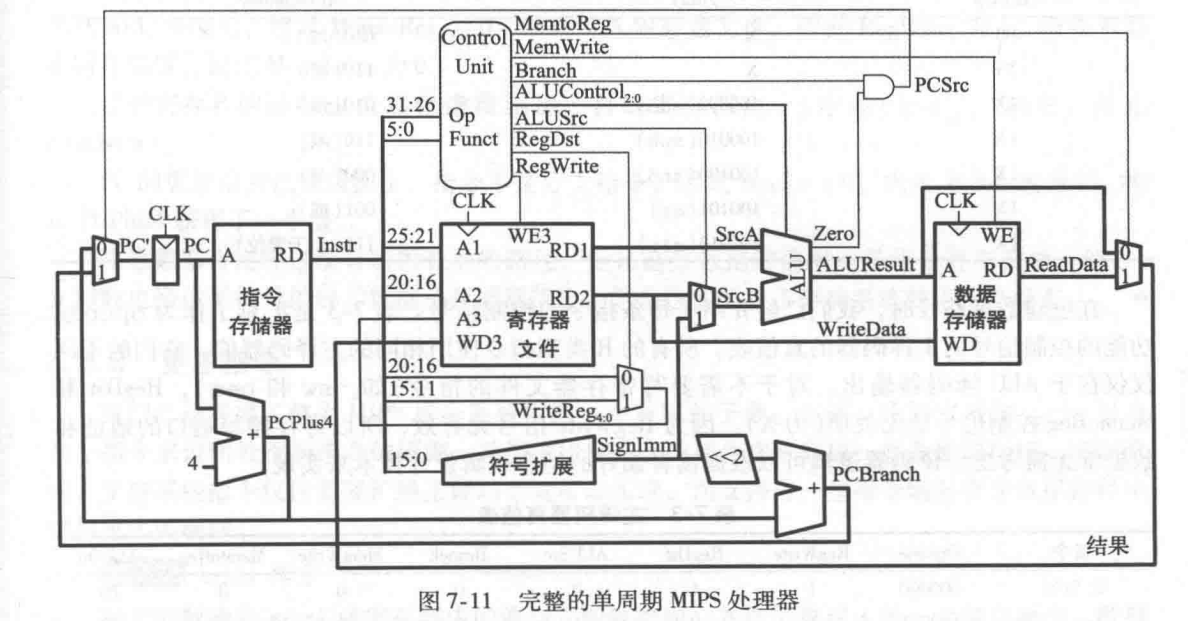
实验2：32位MIPS单周期处理器设计

【一】参考教材7.6节(P270-279)代码，完成MIPS单周期处理器设计

1.用教材图7-60(P276)测试代码测试上述设计

实验方案：



发现：实现单周期CPU的MIPS处理器功能，仿真阶段需要四个大模块：1）中央控制器controller，用于指令译码；2）数据通路datapath，用于串联控制器与其余处理模块；3）指令存储器imem，用于取出译码后的指令进行执行；4）数据存储器dmem，用于存储写入的数据，便于下次读取。

2.关键代码

（仅展示和书上不一样的部分）

module alu(

input logic[31:0] srca,srcb,

input logic[2:0] alucontrol,

output logic[31:0] aluout,

output logic zero

);

always\_comb

case(alucontrol)

3'b000:aluout=srca&srcb;

3'b001:aluout=srca|srcb;

3'b010:aluout=srca+srcb;

3'b011: aluout=0;

3'b100:aluout=srca&(~srcb);

3'b101:aluout=srca|(~srcb);

3'b110:aluout=srca-srcb;

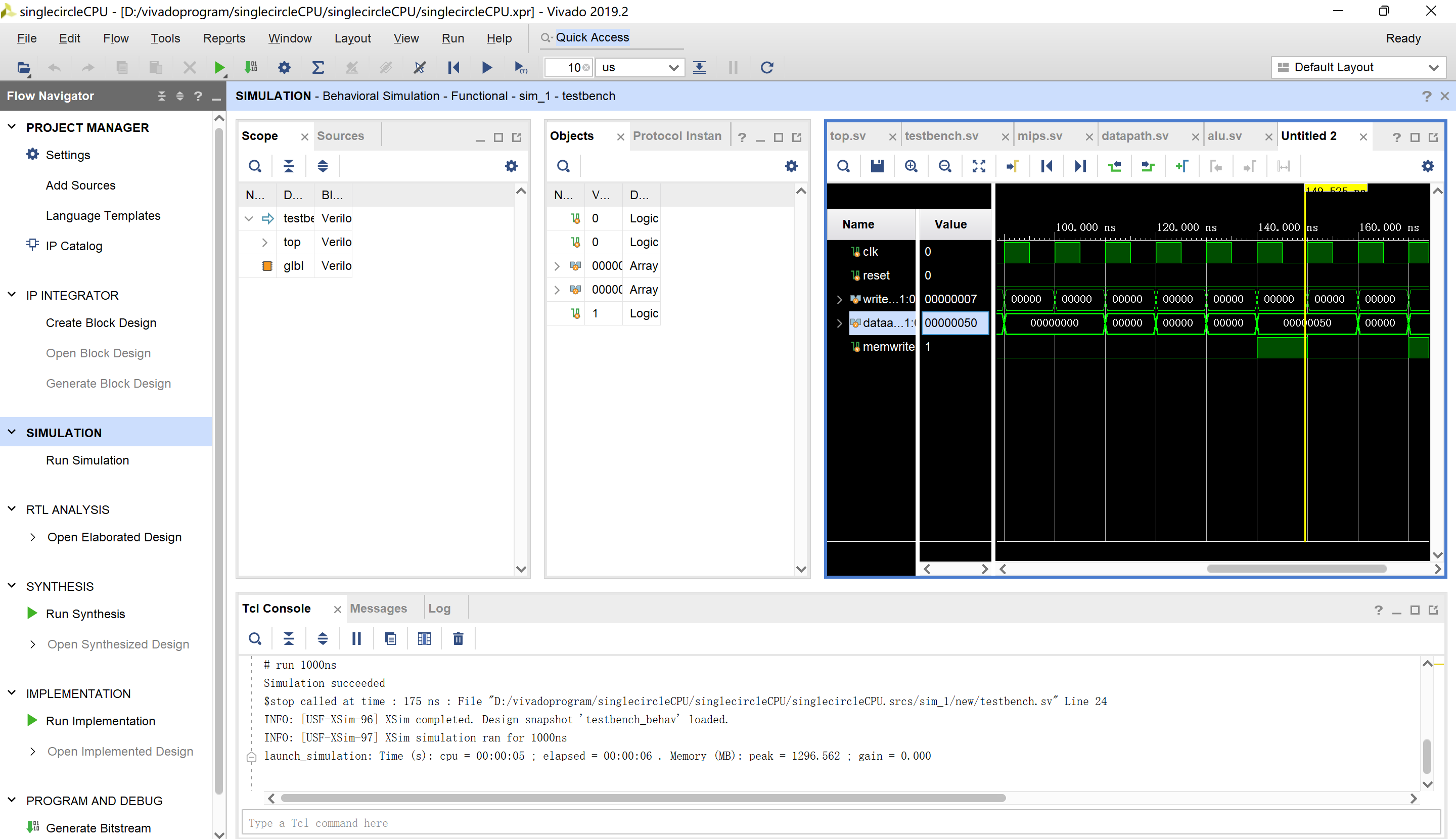
3'b111:aluout= srca<srcb;

endcase

assign zero=(aluout == 32'b0);

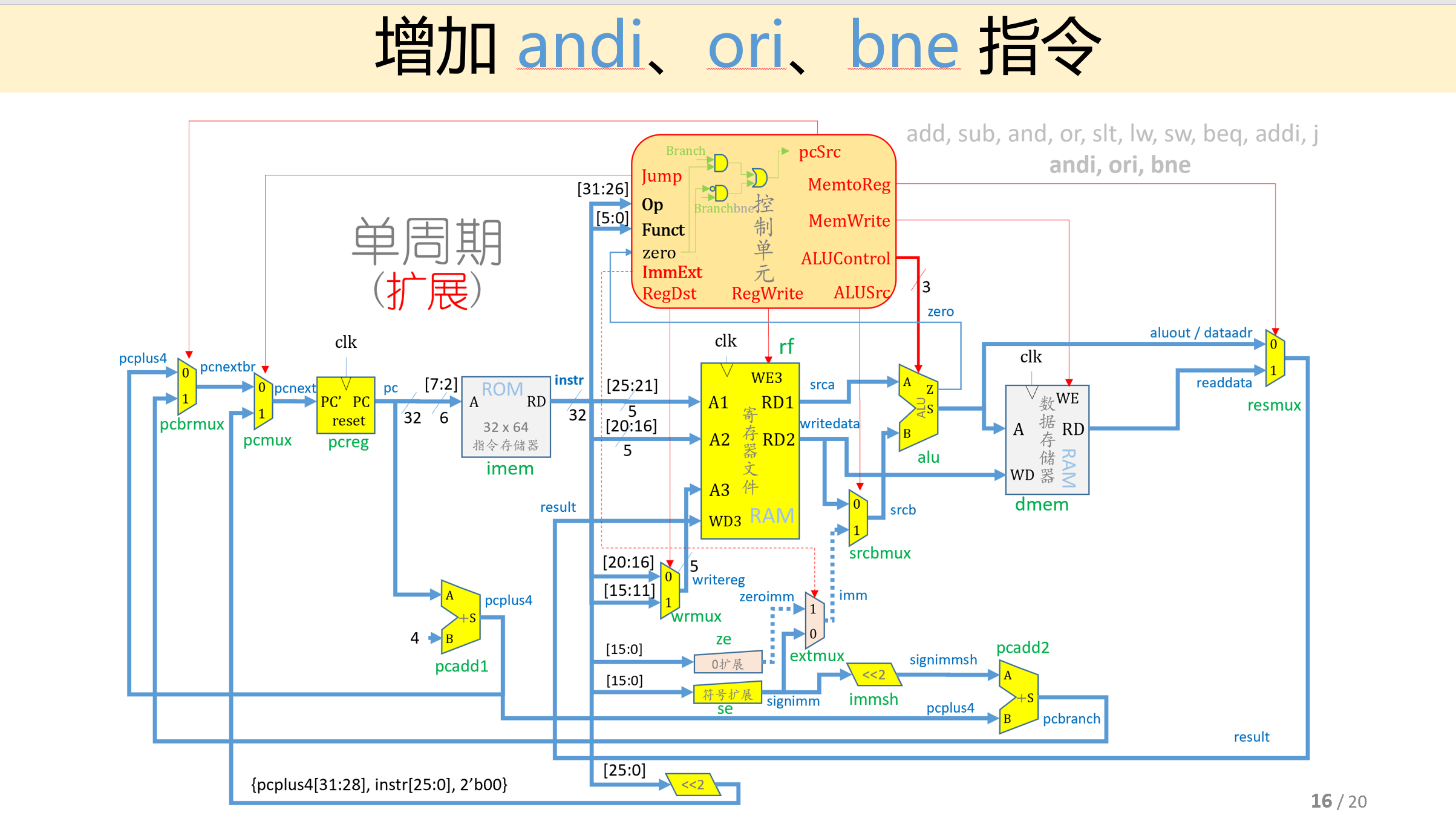
endmodule

测试代码测试上述设计：



【二】

1.在原有指令的基础上进行扩展，添加andi、ori、bne指令，获得修改后的模块框图



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。

为了更为清晰地展示控制单元信号的变化，由真值表进行体现：



2.关键代码：

module mips(

input logic clk, reset,

output logic [31:0] pc,

input logic [31:0] instr,

output logic memwrite,

output logic [31:0] aluout, writedata,

input logic [31:0] readdata);

logic memtoreg, alusrc, regdst,

regwrite, jump, pcsrc, zero,

immext;

logic [2:0] alucontrol;

controller c(instr[31:26], instr[5:0], zero,

memtoreg, memwrite, pcsrc,

alusrc, regdst, regwrite, jump,

immext, alucontrol);

datapath dp(clk, reset, memtoreg, pcsrc,

alusrc, regdst, regwrite, jump,

immext, alucontrol,

zero, pc, instr,

aluout, writedata, readdata);

endmodule

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, immext, branchbne, aluop);

aludec ad(funct, aluop, alucontrol);

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

endmodule

module maindec(

input logic [5:0] op,

output logic memtoreg, memwrite,

output logic branch, alusrc,

output logic regdst, regwrite,

output logic jump,

output logic immext, branchbne,

output logic [2:0] aluop);

logic [11:0] controls;

assign {regwrite, regdst, alusrc, branch, memwrite,

memtoreg, jump, aluop, immext, branchbne} = controls;

always\_comb

case(op)

6'b000000: controls <= 12'b110000001000; //RTYPE

6'b100011: controls <= 12'b101001000000; //LW

6'b101011: controls <= 12'b001010000000; //SW

6'b000100: controls <= 12'b000100000100; //BEQ

6'b001000: controls <= 12'b101000000000; //ADDI

6'b000010: controls <= 12'b000000100000; //J

6'b000101: controls <= 12'b000000000101; //BNE

6'b001101: controls <= 12'b101000001110; //ORI

6'b001100: controls <= 12'b101000010010; //ANDI

default: controls <= 12'bxxxxxxxxxxx; //illegal op

endcase

endmodule

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 beg/bne)

3'b011: alucontrol <= 3'b001; //or(for ori)

3'b100: alucontrol <= 3'b000; //and(for andi)

default: case(funct) //R-type instructions

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

module datapath(

input logic clk, reset,

input logic memtoreg, pcsrc,

input logic alusrc, regdst,

input logic regwrite, jump,

input logic immext,

input logic [2:0] alucontrol,

output logic zero,

output logic [31:0] pc,

input logic [31:0] instr,

output logic [31:0] aluout, writedata,

input logic [31:0] readdata);

logic [4:0] writereg;

logic [31:0] pcnext, pcnextbr, pcplus4, pcbranch;

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

logic [31:0] srca, srcb;

logic [31:0] result;

//next PC logic

flopr #(32) pcreg(clk, reset, pcnext, pc);

adder pcadd1(pc, 32'b100, pcplus4);

sl2 immsh(signimm, signimmsh);

adder 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, instr[25:21], instr[20:16],

writereg, result, srca, writedata);

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

regdst, writereg);

mux2 #(32) resmux(aluout, readdata, memtoreg, result);

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

//ALU logic

mux2 #(32) srcbmux(writedata, imm, alusrc, srcb);

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

//add logic

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

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

endmodule

module signzero(

input logic [15:0] a,

output logic [31:0] y);

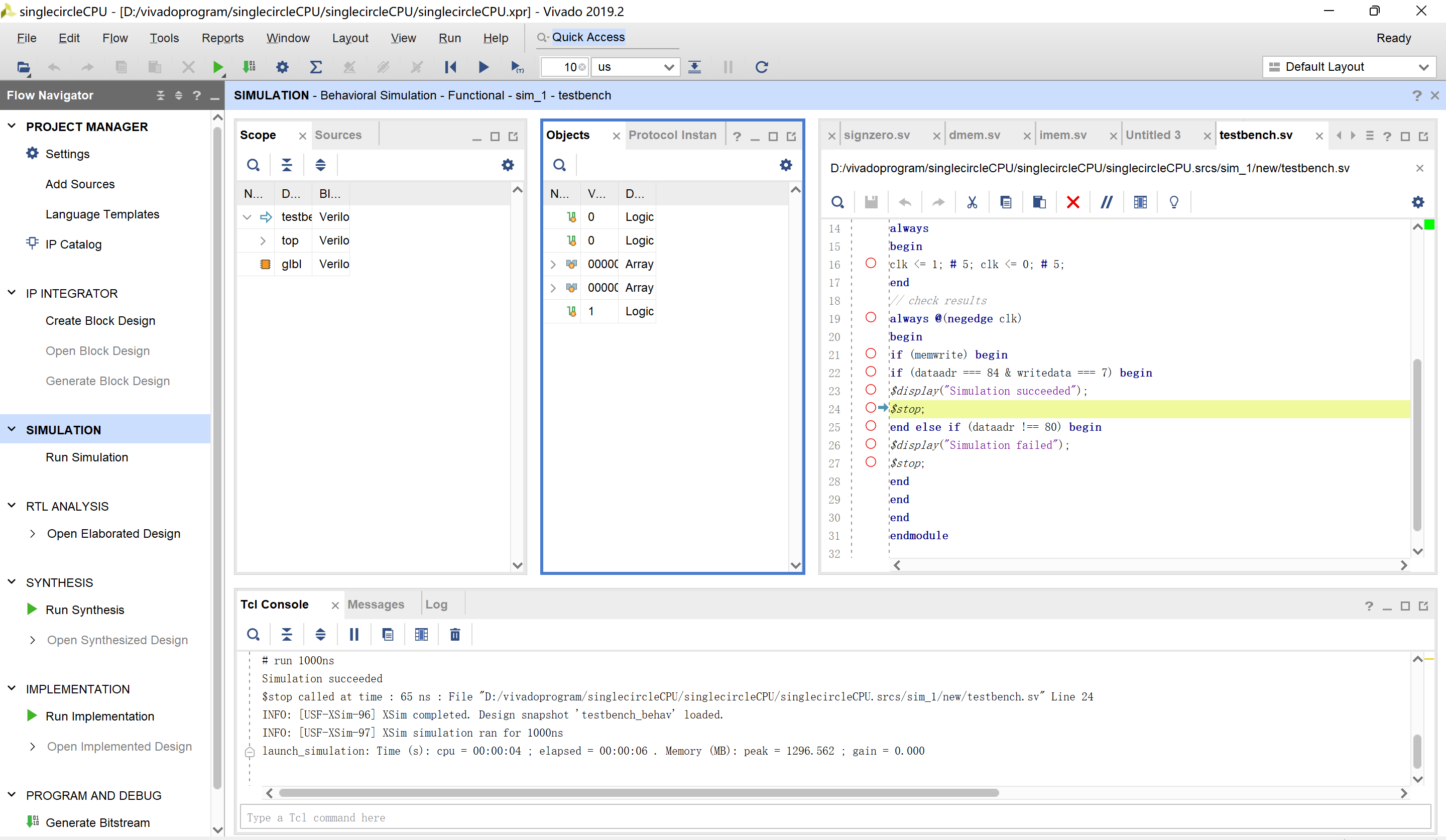
assign y = {{16{1'b0}}, a};

endmodule

3.仿真：

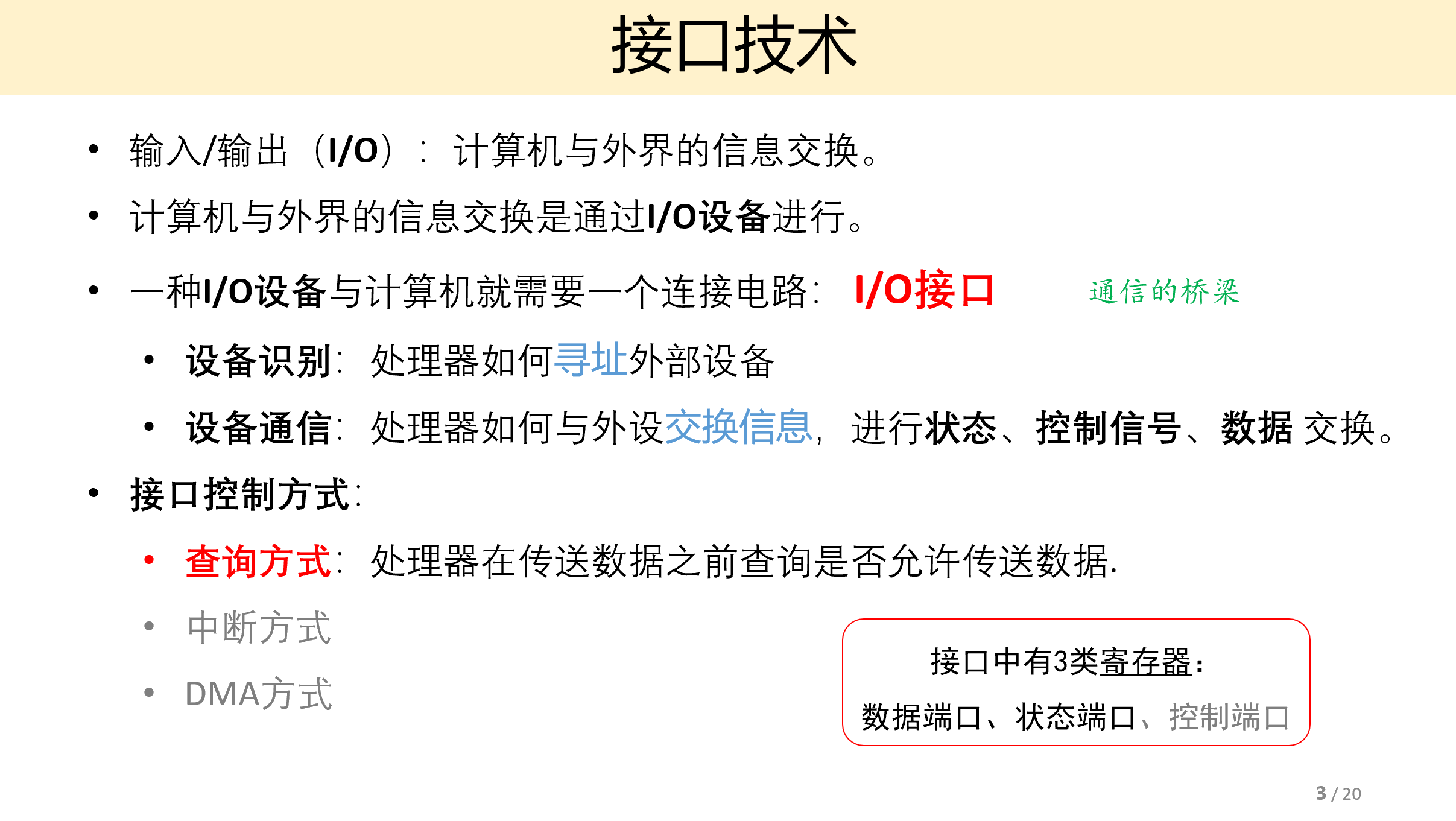


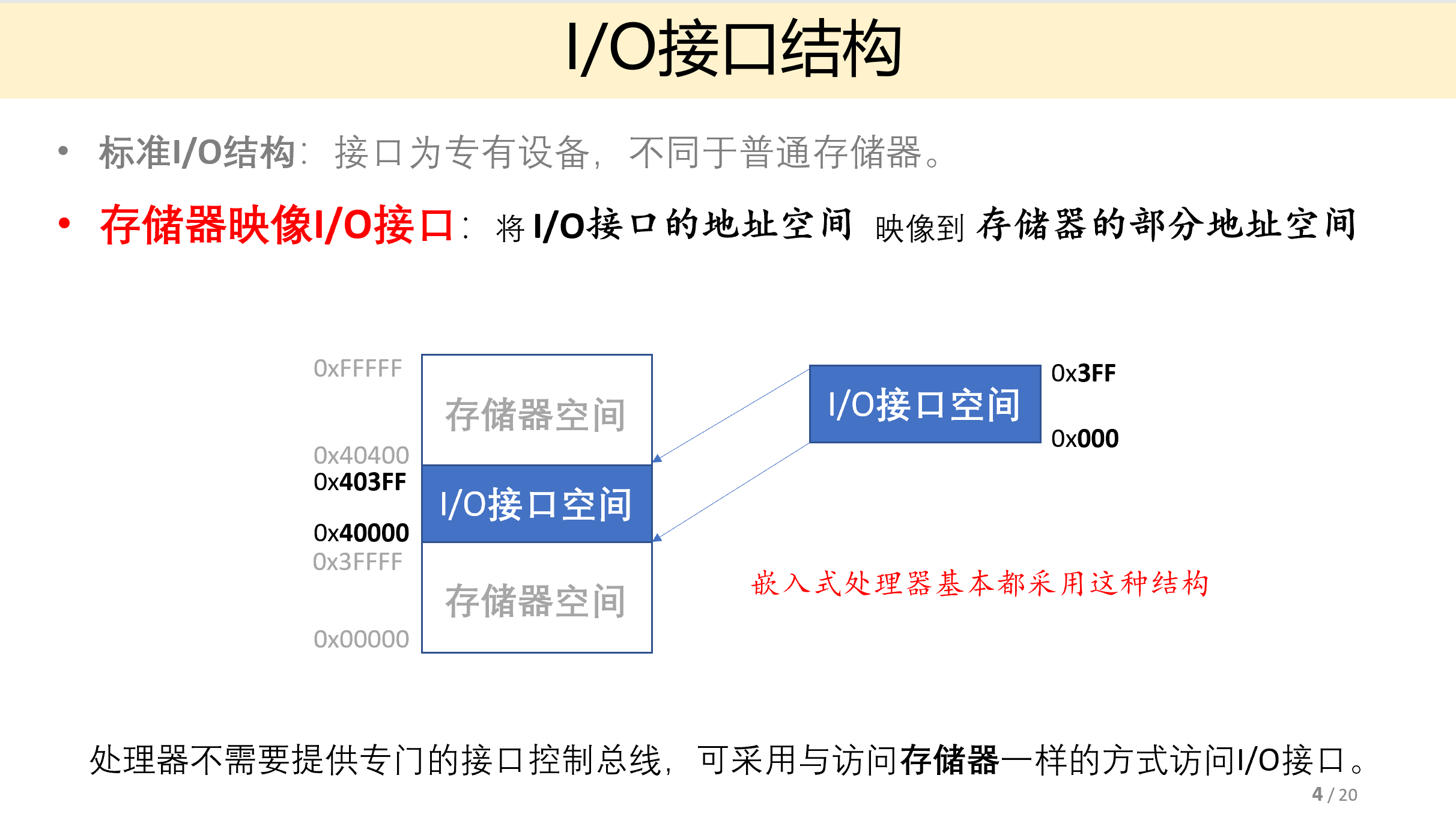
测试结果：

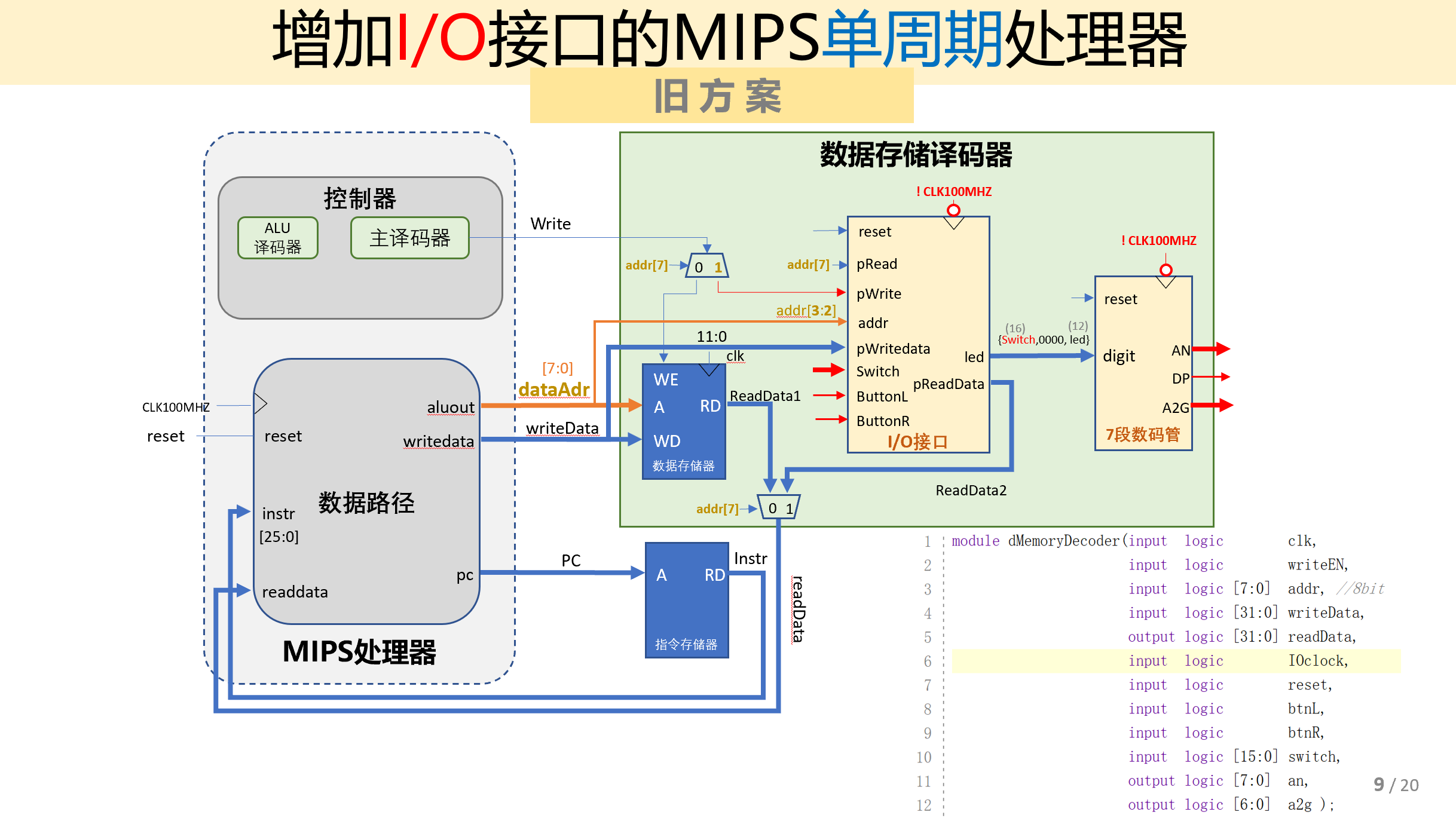


【三】单周期CPU\_IO：仿真+上板

1.方案







这里采用存储器映像IO接口，将IO接口的地址空间映像到存储器部分，使得CPU能够产生存储器读写信号和IO接口读写信号。IO接口一共分为两部分，一部分与CPU相连实现内部逻辑，一部分与外设相连控制输出。与外设相连的端口分为Switch高低端口、结果输出端口与状态端口。状态端口指当status[1]=1时，可输入新数据；当status[0]=1时，可输出新数据。

根据时序逻辑，用button和reset进行状态设置；根据组合逻辑，由加载的指令选择数据读入，并在数据存储译码器中进行复用器选择，由指令地址决定读取数据或者地址。

添加testbench代码进行仿真；添加TestIO.dat文件进行仿真与综合。

2.关键代码：

module top(

input logic CLK100MHZ,

input logic BTNC,//reset

input logic BTNL,//SW input data

input logic BTNR,//七段数码管显示

input logic [15:0] SW,//a:SW[15:8]，b:SW[7:0]

output logic [7:0] AN,

output logic [6:0] A2G,

output logic DP);

logic [31:0] pc;

logic [31:0] instr;

imem imem(pc[7:2],instr); // output

logic Write;//写信号:可能是memWrite，也可能是ioWrite

logic [31:0] dataAdr, writeData, readData;

mips mips(CLK100MHZ,BTNC,pc,instr,Write,dataAdr,writeData,readData);

dMemoryDecoder dmd(.clk(CLK100MHZ),

.writeEN(Write),

.addr(dataAdr[7:0]),

.writeData(writeData),

.readData(readData), //output

.reset(BTNC),

.btnL(BTNL),

.btnR(BTNR),

.switch(SW),

.an(AN), //output

.dp(DP),

.a2g(A2G)); //output

Endmodule

module dMemoryDecoder(

input logic clk,

input logic writeEN,

input logic [7:0] addr, //8bit

input logic [31:0] writeData,

output logic [31:0] readData,

input logic reset,

input logic btnL,

input logic 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;

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

.d1(ReadData2),

.s(addr[7]),

.y(readData));

dmem dMemory(.clk(clk),

.we(writeEN),

.a(addr),

.wd(writeData),

.rd(ReadData1));

io IO(.clk(clk),

.reset(reset),

.pRead(addr[7]),

.pWrite(writeEN),

.addr(addr[3:2]),

.pWriteData(writeData[11:0]),

.pReadData(ReadData2),

.buttonL(btnL),

.buttonR(btnR),

.switch(switch),

.led(led));

m7seg mux7seg(.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

module m7seg(

input logic [31:0] x,

input logic clk,

output logic [6:0] a2g,

output logic [7:0] an,

output logic dp );

logic [2:0] s;

logic [4:0] digit;

logic [19:0] clkdiv;

assign dp =1;

assign s = clkdiv[19:17];

always\_comb

case(s)

0: digit = {{1'b0},x[3:0]};

1: digit = {{1'b0},x[7:4]};

2: digit = {{1'b0},x[11:8]};

3: digit = {{1'b1},x[15:12]};

4: digit = {{1'b0},x[19:16]};

5: digit = {{1'b0},x[23:20]};

6: digit = {{1'b0},x[27:24]};

7: digit = {{1'b0},x[31:28]};

default: digit = {{1'b0},x[4:0]};

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\_1111;

endcase

always @(posedge clk)

begin

clkdiv <= clkdiv + 1;

end

//实例化 7段数码管

Hex7Seg s7(.x(digit), .a2g(a2g));

Endmodule

module Hex7Seg(

input logic [4:0] x,

output logic [6:0] a2g );

always\_comb

case(x)

'h0: a2g = 7'b0000001;

'h1: a2g = 7'b1001111;

'h2: a2g = 7'b0010010;

'h3: a2g = 7'b0000110;

'h4: a2g = 7'b1001100;

'h5: a2g = 7'b0100100;

'h6: a2g = 7'b0100000;

'h7: a2g = 7'b0001111;

'h8: a2g = 7'b0000000;

'h9: a2g = 7'b0000100;

'ha: a2g = 7'b0000010;

'hb: a2g = 7'b1100000;

'hc: a2g = 7'b0110001;

'hd: a2g = 7'b1000001;

'he: a2g = 7'b0010000;

'hf: a2g = 7'b0111000;

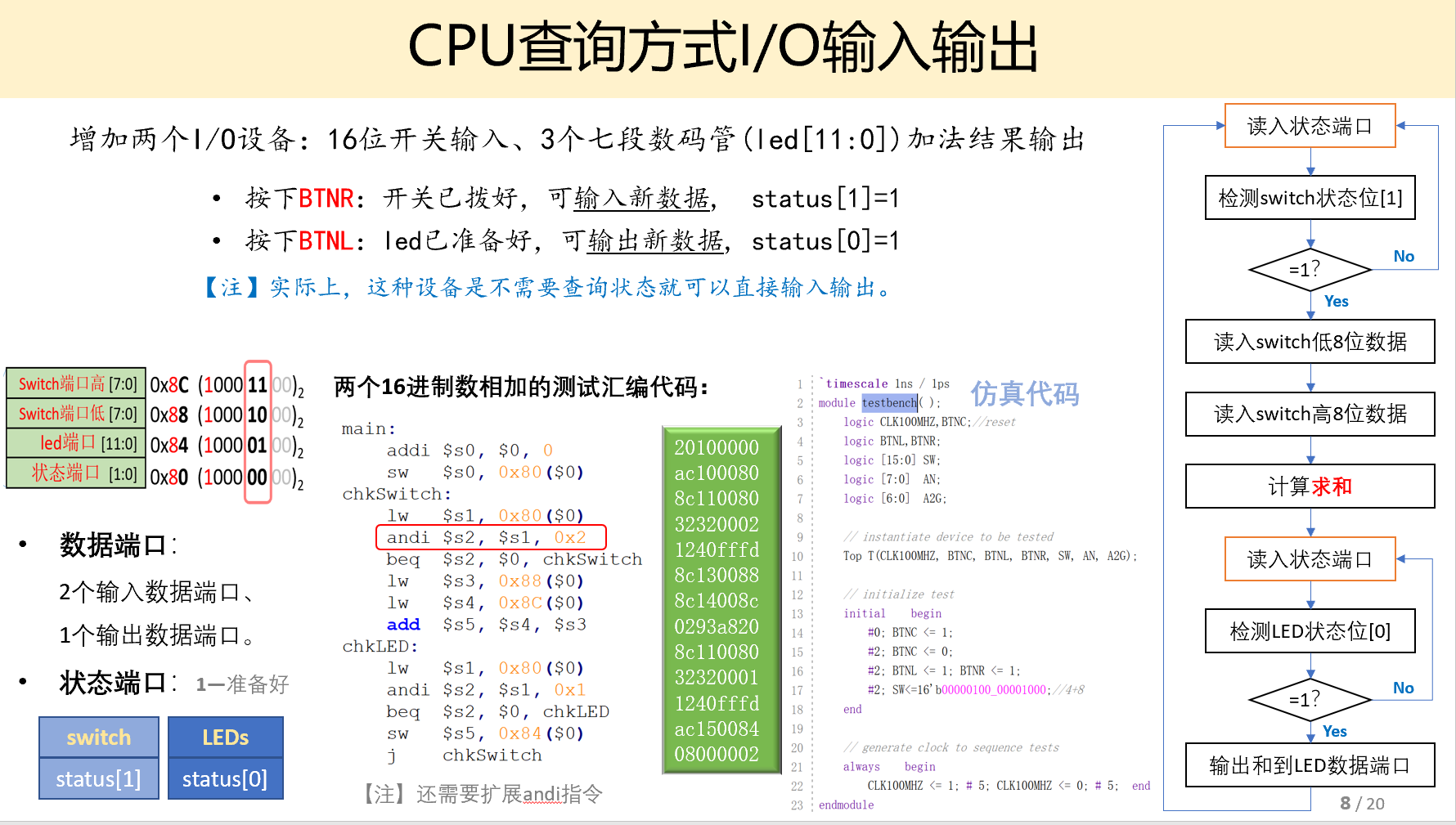
'h10:a2g = 7'b1110110;

default: a2g = 7'b1111111;

endcase

endmodule

3.仿真



实验结果：



上板测试：





【四】问题回顾：总结反思

1在进行第一步时，对照教材编写的时候出现了代码误写，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。

2因为对于IO不熟悉，借用了隔壁班的ppt进行学习设计。

3总结反思：

 本次实验主要考察了对于单周期CPU原理的认识，其中涉及了对MIPS指令体系的理解。对于单周期CPU基本流程熟悉，学习编写各种模板。本次实验复习了上学期数字逻辑中所学的基础硬件知识与基本逻辑。更让我体会到编写代码时需要更加仔细认真，同时要学习读懂控制台的报告信息。同时，发现自己在理解测试机器代码上仍然有所欠缺，需要在之后的实验中进一步加深理解。