实验4：流水线设计

具体功能：

1）32位MIPS架构的cpu5级流水线，实现13条指令

2）对数据冲突采取插入空操作（nops）、重定向（forward）及阻塞（stall）方法，对控制冲突采取提前预测分支（eqcmp），如果判断将因分支前置产生数据冲突，即同样使用插入空操作、重定向及阻塞方法解决。

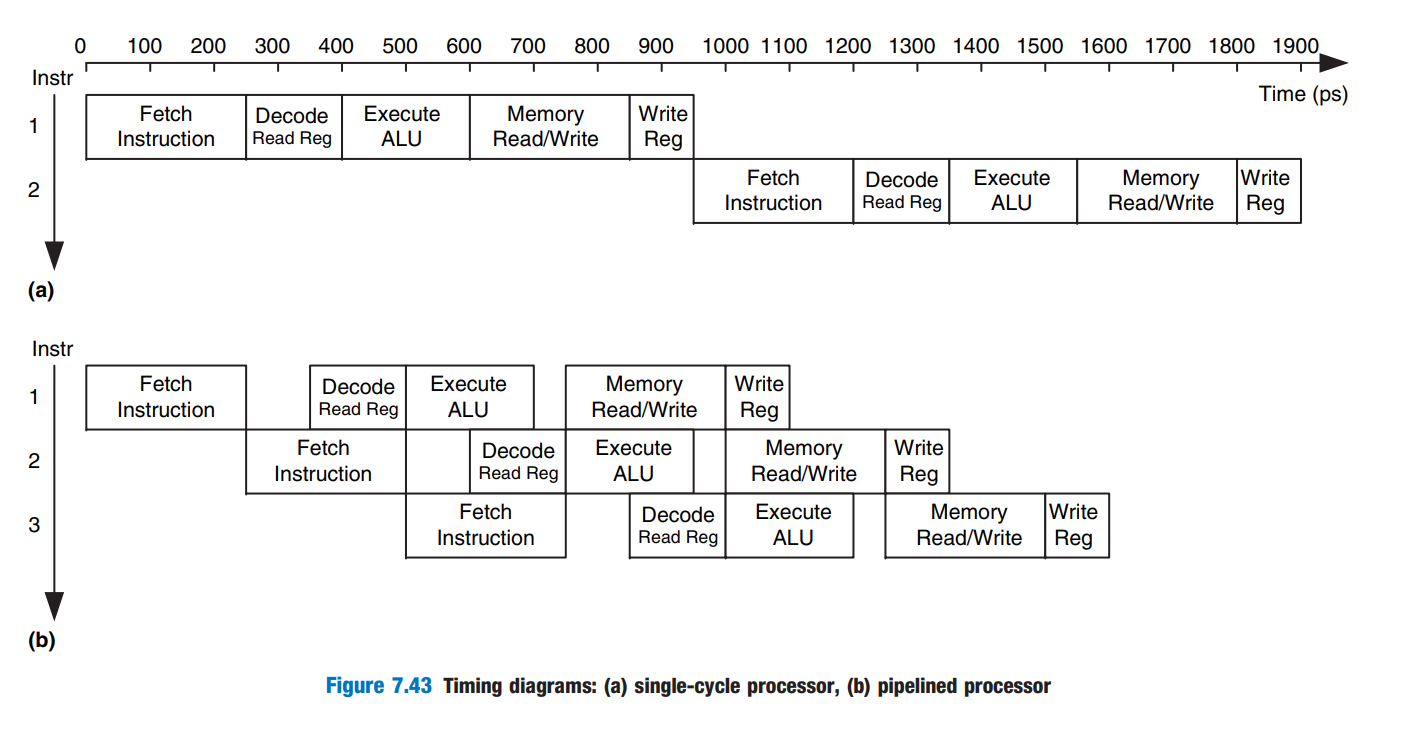
3）实验板上的开关用于模拟输入操作数和其它控制输入。

4）实验板上的8位数码管输出运算结果。

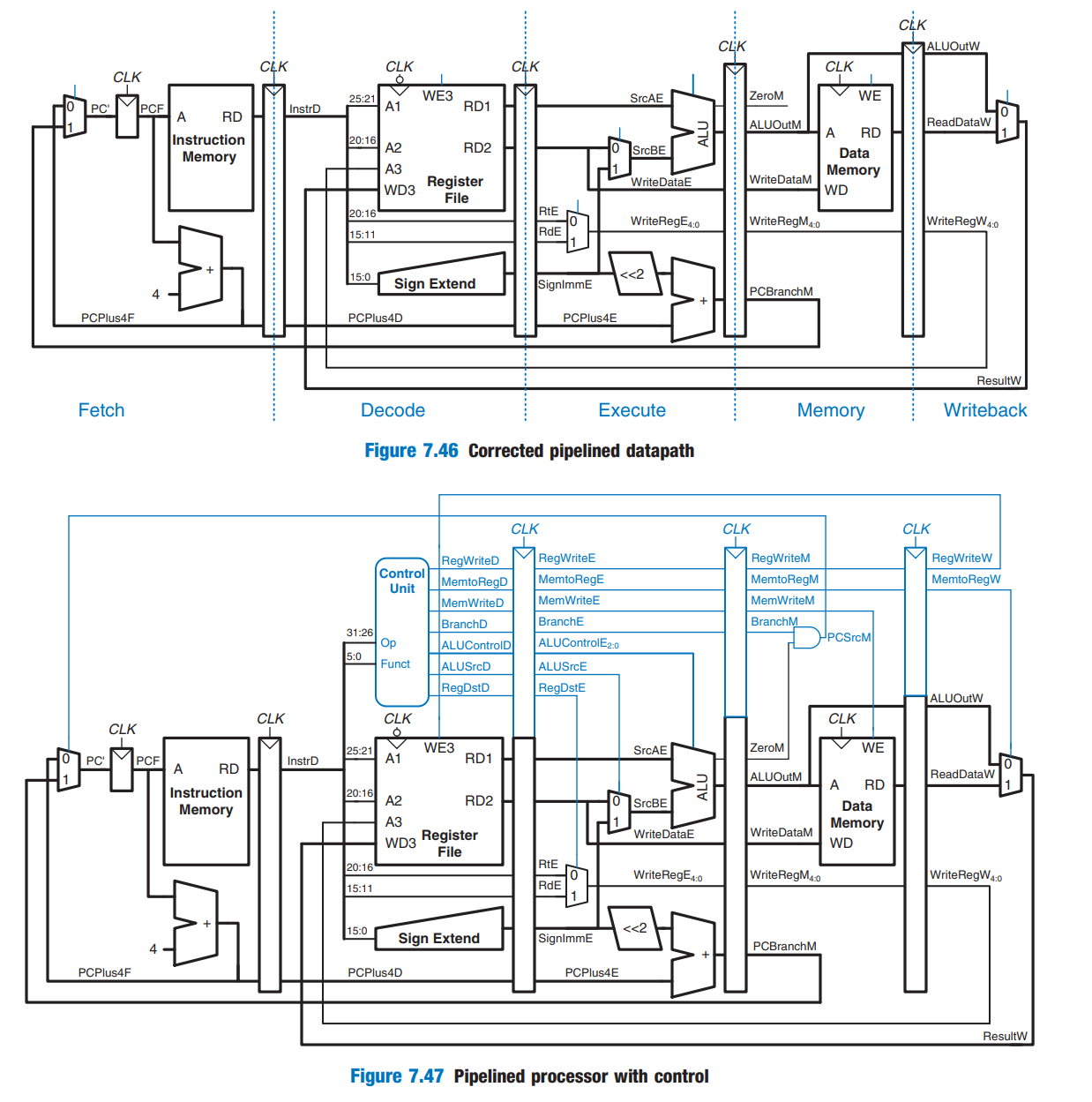
【一】流水线CPU与仿真

1. 实验方案

流水线CPU将每条指令的执行分成5个阶段，采用时间并行的方式，进行取指令——译码——执行——存储器——写回的过程，每个阶段的时间长度以最慢阶段的为准，抽象表示如下：（为了区分寄存器的写/读状态，规定寄存器在下降沿前半个周期写入，在上升沿后半个周期读取）



相较单周期CPU，流水线CPU在每个阶段间增添了暂存寄存器保存各控制信号的中间状态，以此并行执行。指令执行框图如下：



流水线CPU用于阶段间存在复用，存在数据冲突及控制冲突，前者指寄存器值还未写回寄存器文件在下一阶段已经开始使用，后者指在取下一条指令时，分支跳转尚未确定。

为了解决由于指令依赖于另一条还未结束指令的结果，对数据冲突采取插入空操作（nops）、重定向（forward）及阻塞（stall）方法，对控制冲突采取提前预测分支（eqcmp），如果判断将因分支前置产生数据冲突，即同样使用插入空操作、重定向及阻塞方法解决。

分析冲突解决具体的执行方法：

1. 插入空操作（nops）

插入空指令以使提前调用的指令后移。

1. 重定向（forward）

当执行阶段（E）中的指令有一个源寄存器rsE、rtE与存储器阶段（M）或写回阶段（W）中的目的寄存器writeRegM、writeRegW相匹配时，即将M或W阶段应存入寄存器中的值提前定向到下一指令的E阶段。

1. 阻塞（stall）

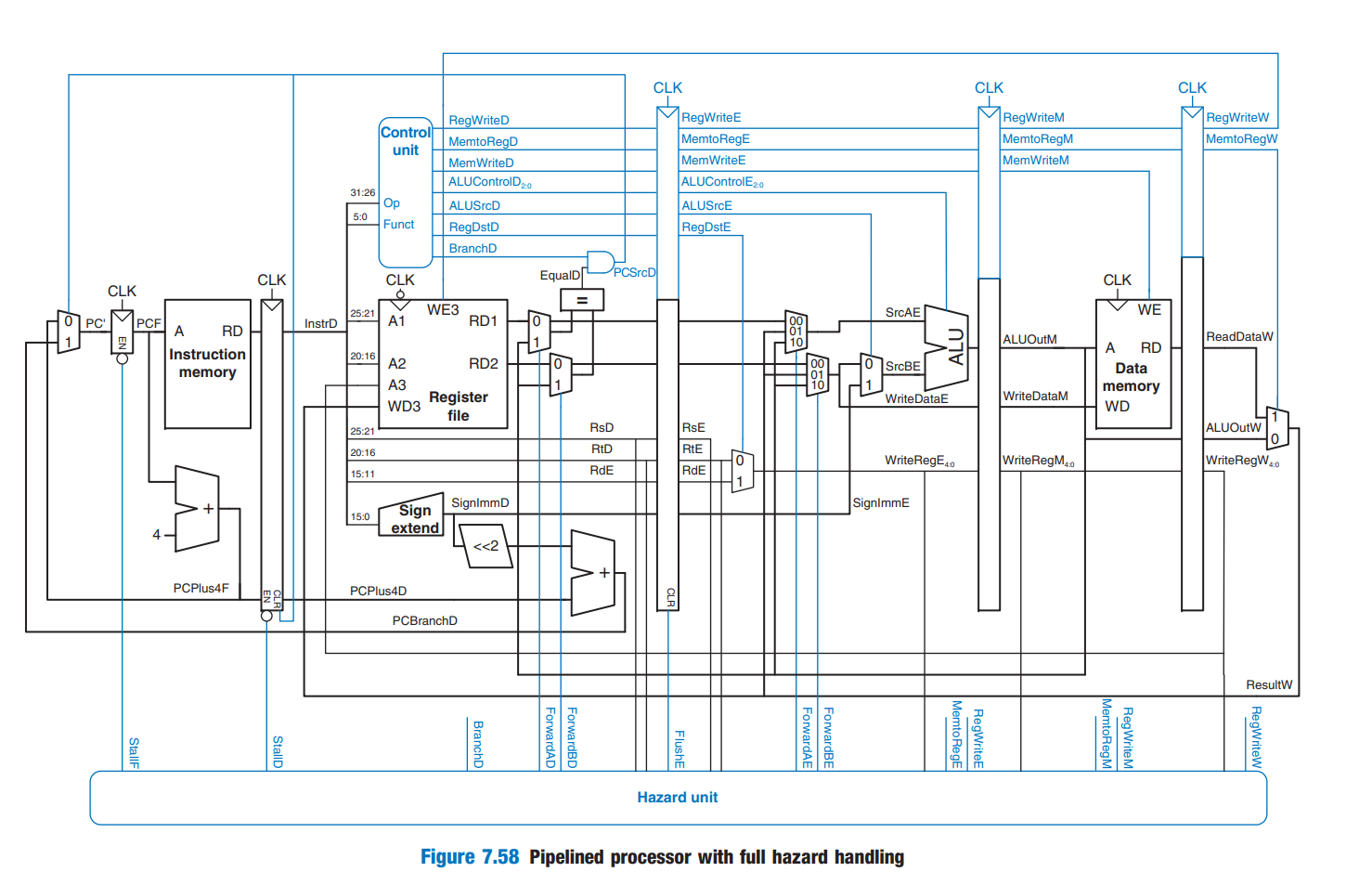
当lw的目的寄存器rtE与译码阶段（D）中的源操作数rsD、rtD相匹配时，由于需要获取的值还未出现，故无法使用重定向进行提前获取，需要进行阻塞。

使用信号StallF、StallD迫使取指阶段（F）、译码阶段（D）流水寄存器保持原来的值，使用信号FlushE清除执行阶段（E）流水线寄存器的内容，并去掉无效控制信号，以防止更新内存和状态。

1. 分支预测（eqcmp）

采用专门的相等比较器，在译码阶段结束时确定下一个PC，进行分支预测。如果分支需要跳转，则仅下一指令完成第一阶段，只需要抛弃下一指令即可。

由此，处理所有冲突的流水线CPU结构框图如下：



2. 关键代码

主要呈现流水线CPU执行流水指令的部分

1. 控制器controller

module controller(

input logic clk, reset,

input logic [5:0] opD, functD,

input logic flushE, equalD,

output logic memtoregE, memtoregM, memtoregW, memwriteM,

output logic pcsrcD, branchD, alusrcE,

output logic regdstE, regwriteE, regwriteM, regwriteW,

output logic jumpD,

output logic [2:0] alucontrolE);

logic [1:0] aluopD;

logic memtoregD, memwriteD, alusrcD, regdstD, regwriteD;

logic [2:0] alucontrolD;

logic memwriteE;

maindec md(opD, memtoregD, memwriteD, branchD,

alusrcD, regdstD, regwriteD, jumpD, aluopD);

aludec ad(functD, aluopD, alucontrolD);

assign pcsrcD = branchD & equalD;

floprc #(8) regE(clk, reset, flushE,

{memtoregD, memwriteD, alusrcD, regdstD, regwriteD, alucontrolD},

{memtoregE, memwriteE, alusrcE, regdstE, regwriteE, alucontrolE});

flopr #(3) regM(clk, reset,

{memtoregE, memwriteE, regwriteE},

{memtoregM, memwriteM, regwriteM});

flopr #(2) regW(clk, reset,

{memtoregM, regwriteM},

{memtoregW, regwriteW});

Endmodule

1. 数据通路datapath

module datapath(

input logic clk, reset,

input logic memtoregE, memtoregM, memtoregW,

input logic pcsrcD, branchD,

input logic alusrcE, regdstE,

input logic regwriteE, regwriteM, regwriteW,

input logic jumpD,

input logic [2:0] alucontrolE,

output logic equalD,

output logic [31:0] pcF,

input logic [31:0] instrF,

output logic [31:0] aluoutM, writedataM,

input logic [31:0] readdataM,

output logic [5:0] opD, functD,

output logic flushE);

logic forwardaD, forwardbD;

logic [1:0] forwardaE, forwardbE;

logic stallF, stallD;

logic [4:0] rsD, rtD, rdD, rsE, rtE, rdE;

logic [4:0] writeregE, writeregM, writeregW;

logic flushD;

logic [31:0] pcnextFD, pcnextbrFD, pcplus4F, pcbranchD;

logic [31:0] signimmD, signimmE, signimmshD;

logic [31:0] srcaD, srca2D, srcaE, srca2E;

logic [31:0] srcbD, srcb2D, srcbE, srcb2E, srcb3E;

logic [31:0] pcplus4D, instrD;

logic [31:0] aluoutE, aluoutW;

logic [31:0] readdataW, resultW;

// hazard detection

hazard h(rsD, rtD, rsE, rtE, writeregE, writeregM, writeregW,

regwriteE, regwriteM, regwriteW,

memtoregE, memtoregM, branchD,

forwardaD, forwardbD, forwardaE, forwardbE,

stallF, stallD, flushE);

// next PC logic (operates in fetch and decode)

mux2 #(32) pcbrmux(pcplus4F, pcbranchD, pcsrcD, pcnextbrFD);

mux2 #(32) pcmux(pcnextbrFD, {pcplus4D[31:28], instrD[25:0], 2'b00},

jumpD, pcnextFD);

// register file (operates in decode and writeback)

regfile rf(clk, regwriteW, rsD, rtD, writeregW,

resultW, srcaD, srcbD);

// Fetch

flopenr #(32) pcreg(clk, reset, ~stallF, pcnextFD, pcF);

adder pcadd1(pcF, 32'b100, pcplus4F);

// Decode

flopenr #(32) r1D(clk, reset, ~stallD, pcplus4F, pcplus4D);

flopenrc #(32) r2D(clk, reset, ~stallD, flushD, instrF, instrD);

signext se(instrD[15:0], signimmD);

sl2 immsh(signimmD, signimmshD);

adder pcadd2(pcplus4D, signimmshD, pcbranchD);

mux2 #(32) forwardadmux(srcaD, aluoutM, forwardaD, srca2D);

mux2 #(32) forwardbdmux(srcbD, aluoutM, forwardbD, srcb2D);

eqcmp comp(srca2D, srcb2D, equalD);

assign opD = instrD[31:26];

assign functD = instrD[5:0];

assign rsD = instrD[25:21];

assign rtD = instrD[20:16];

assign rdD = instrD[15:11];

assign flushD = pcsrcD | jumpD;

// Execute

floprc #(32) r1E(clk, reset, flushE, srcaD, srcaE);

floprc #(32) r2E(clk, reset, flushE, srcbD, srcbE);

floprc #(32) r3E(clk, reset, flushE, signimmD, signimmE);

floprc #(5) r4E(clk, reset, flushE, rsD, rsE);

floprc #(5) r5E(clk, reset, flushE, rtD, rtE);

floprc #(5) r6E(clk, reset, flushE, rdD, rdE);

mux3 #(32) forwardaemux(srcaE, resultW, aluoutM, forwardaE, srca2E);

mux3 #(32) forwardbemux(srcbE, resultW, aluoutM, forwardbE, srcb2E);

mux2 #(32) srcbmux(srcb2E, signimmE, alusrcE, srcb3E);

alu alu(srca2E, srcb3E, alucontrolE, aluoutE);

mux2 #(5) wrmux(rtE, rdE, regdstE, writeregE);

// Memory

flopr #(32) r1M(clk, reset, srcb2E, writedataM);

flopr #(32) r2M(clk, reset, aluoutE, aluoutM);

flopr #(5) r3M(clk, reset, writeregE, writeregM);

// Writeback

flopr #(32) r1W(clk, reset, aluoutM, aluoutW);

flopr #(32) r2W(clk, reset, readdataM, readdataW);

flopr #(5) r3W(clk, reset, writeregM, writeregW);

mux2 #(32) resmux(aluoutW, readdataW, memtoregW, resultW);

endmodule

1. 冲突处理hazard

module hazard(

input logic [4:0] rsD, rtD, rsE, rtE,

input logic [4:0] writeregE, writeregM, writeregW,

input logic regwriteE, regwriteM, regwriteW,

input logic memtoregE, memtoregM, branchD,

output logic forwardaD, forwardbD,

output logic [1:0] forwardaE, forwardbE,

output logic stallF, stallD, flushE);

logic lwstallD, branchstallD;

assign forwardaD = (rsD != 0 & rsD == writeregM & regwriteM);

assign forwardbD = (rtD != 0 & rtD == writeregM & regwriteM);

always\_comb

begin

forwardaE = 2'b00; forwardbE = 2'b00;

if (rsE != 0)

if (rsE == writeregM & regwriteM) forwardaE = 2'b10;

else if (rsE == writeregW & regwriteW) forwardaE = 2'b01;

if (rtE != 0)

if (rtE == writeregM & regwriteM) forwardbE = 2'b10;

else if (rtE == writeregW & regwriteW) forwardbE = 2'b01;

end

assign lwstallD = memtoregE & (rtE == rsD | rtE == rtD);

assign branchstallD = branchD &

(regwriteE & (writeregE == rsD | writeregE == rtD) |

memtoregM & (writeregM == rsD | writeregM == rtD));

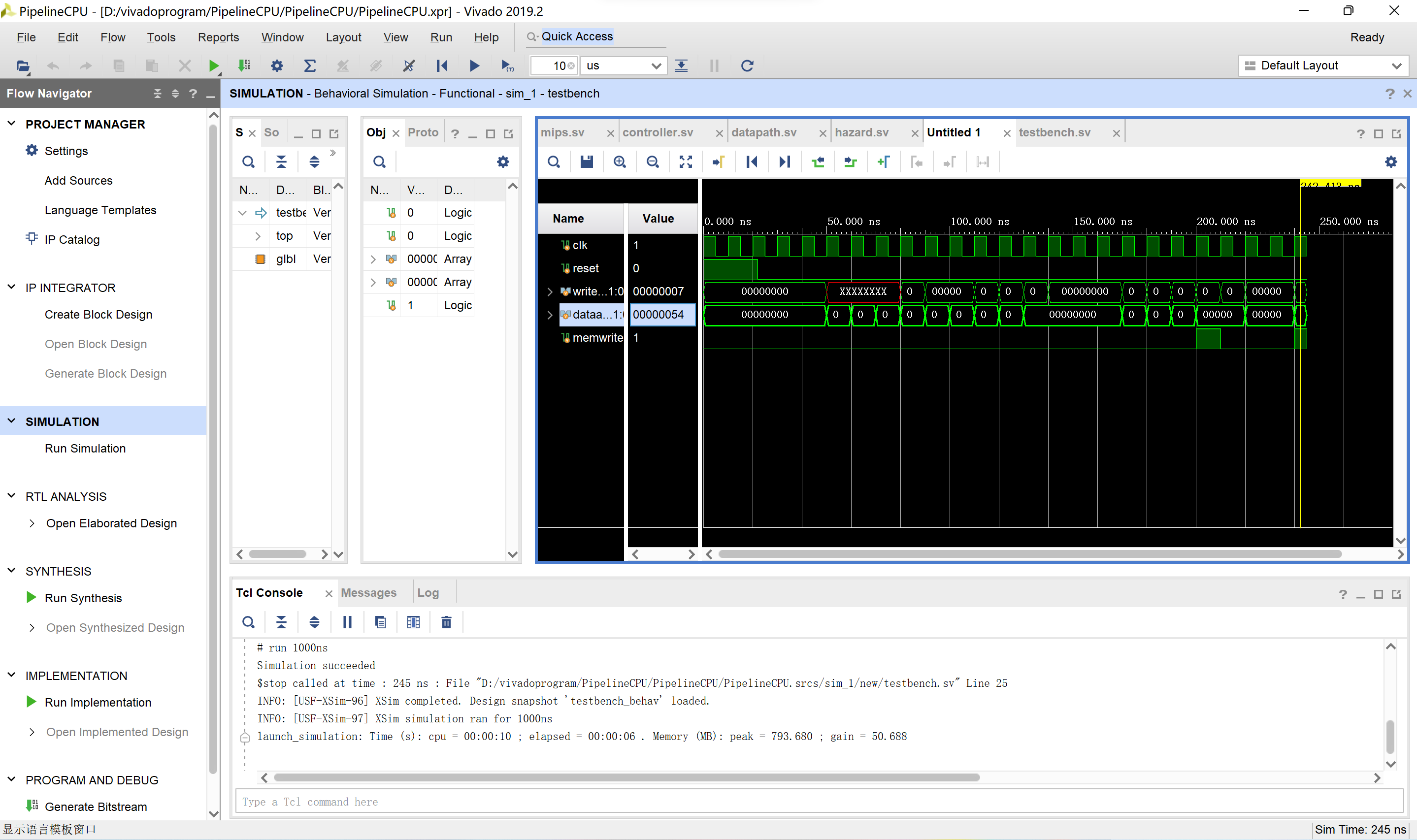
assign stallD = lwstallD | branchstallD;

assign stallF = stallD;

assign flushE = stallD;

endmodule

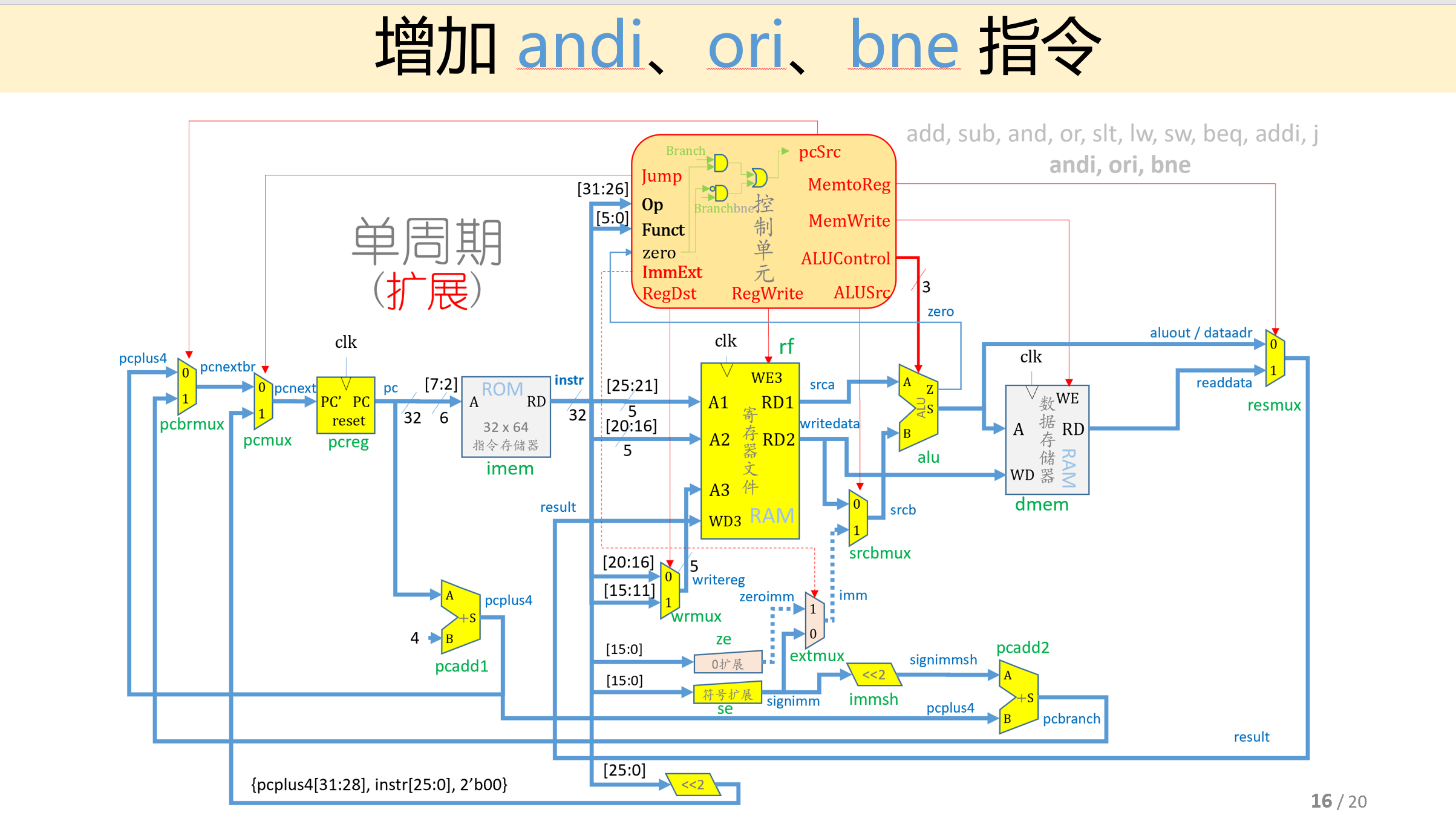
3.仿真结果：



【二】流水线CPU\_add：仿真

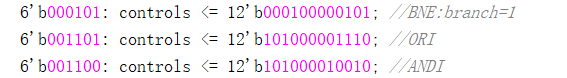
1. 实验方案

在原有指令的基础上添加andi、ori、bne指令，修改后的模块框图与单周期CPU类似：



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，BranchBne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。

对于新增指令无需进行新的冲突处理，andi、ori可能产生的冲突为数据冲突，branchbne产生的分支控制冲突可沿用branch的冲突处理方法进行处理，只需在主译码器进行control译码时，将对应的branch位与branchbne位均置为1：



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

2.关键代码

显示更改部分：

1. 控制器controller/maindec/aludec

module controller(

input logic clk, reset,

input logic [5:0] opD, functD,

input logic flushE, equalD,

output logic memtoregE, memtoregM, memtoregW, memwriteM,

output logic pcsrcD, branchD, branchbneD, alusrcE,

output logic regdstE, regwriteE, regwriteM, regwriteW,

output logic jumpD, immext,

output logic [2:0] alucontrolE);

logic [2:0] aluopD;

logic memtoregD, memwriteD, alusrcD, regdstD, regwriteD;

logic [2:0] alucontrolD;

logic memwriteE;

maindec md(opD, memtoregD, memwriteD, branchD, branchbneD,

alusrcD, regdstD, regwriteD, jumpD, immext, aluopD);

aludec ad(functD, aluopD, alucontrolD);

assign pcsrcD = (branchD & equalD) | (branchbneD & (~equalD));

floprc #(8) regE(clk, reset, flushE,

{memtoregD, memwriteD, alusrcD, regdstD, regwriteD, alucontrolD},

{memtoregE, memwriteE, alusrcE, regdstE, regwriteE, alucontrolE});

flopr #(3) regM(clk, reset,

{memtoregE, memwriteE, regwriteE},

{memtoregM, memwriteM, regwriteM});

flopr #(2) regW(clk, reset,

{memtoregM, regwriteM},

{memtoregW, regwriteW});

Endmodule

module maindec(

input logic [5:0] op,

output logic memtoreg, memwrite,

output logic branch, branchbne, alusrc,

output logic regdst, regwrite,

output logic jump, immext,

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'b000100000101; //BNE:branch=1

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'b100: alucontrol <= 3'b000;

3'b011: alucontrol <= 3'b001;

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

1. 数据通路datapath

module datapath(

input logic clk, reset,

input logic memtoregE, memtoregM, memtoregW,

input logic pcsrcD, branchD,

input logic alusrcE, regdstE,

input logic regwriteE, regwriteM, regwriteW,

input logic jumpD,

input logic [2:0] alucontrolE,

output logic equalD,

output logic [31:0] pcF,

input logic [31:0] instrF,

output logic [31:0] aluoutM, writedataM,

input logic [31:0] readdataM,

output logic [5:0] opD, functD,

output logic flushE,

input logic immext);

logic forwardaD, forwardbD;

logic [1:0] forwardaE, forwardbE;

logic stallF, stallD;

logic [4:0] rsD, rtD, rdD, rsE, rtE, rdE;

logic [4:0] writeregE, writeregM, writeregW;

logic flushD;

logic [31:0] pcnextFD, pcnextbrFD, pcplus4F, pcbranchD;

logic [31:0] signimmD, signimmE, signimmshD;

logic [31:0] zeroimm, imm;

logic [31:0] srcaD, srca2D, srcaE, srca2E;

logic [31:0] srcbD, srcb2D, srcbE, srcb2E, srcb3E;

logic [31:0] pcplus4D, instrD;

logic [31:0] aluoutE, aluoutW;

logic [31:0] readdataW, resultW;

// hazard detection

hazard h(rsD, rtD, rsE, rtE, writeregE, writeregM, writeregW,

regwriteE, regwriteM, regwriteW,

memtoregE, memtoregM, branchD,

forwardaD, forwardbD, forwardaE, forwardbE,

stallF, stallD, flushE);

// next PC logic (operates in fetch and decode)

mux2 #(32) pcbrmux(pcplus4F, pcbranchD, pcsrcD, pcnextbrFD);

mux2 #(32) pcmux(pcnextbrFD, {pcplus4D[31:28], instrD[25:0], 2'b00},

jumpD, pcnextFD);

// register file (operates in decode and writeback)

regfile rf(clk, regwriteW, rsD, rtD, writeregW,

resultW, srcaD, srcbD);

// Fetch

flopenr #(32) pcreg(clk, reset, ~stallF, pcnextFD, pcF);

adder pcadd1(pcF, 32'b100, pcplus4F);

// Decode

flopenr #(32) r1D(clk, reset, ~stallD, pcplus4F, pcplus4D);

flopenrc #(32) r2D(clk, reset, ~stallD, flushD, instrF, instrD);

signext se(instrD[15:0], signimmD);

ze ze(instrD[15:0], zeroimm);

sl2 immsh(signimmD, signimmshD);

adder pcadd2(pcplus4D, signimmshD, pcbranchD);

mux2 #(32) forwardadmux(srcaD, aluoutM, forwardaD, srca2D);

mux2 #(32) forwardbdmux(srcbD, aluoutM, forwardbD, srcb2D);

eqcmp comp(srca2D, srcb2D, equalD);

assign opD = instrD[31:26];

assign functD = instrD[5:0];

assign rsD = instrD[25:21];

assign rtD = instrD[20:16];

assign rdD = instrD[15:11];

assign flushD = pcsrcD | jumpD;

// Execute

floprc #(32) r1E(clk, reset, flushE, srcaD, srcaE);

floprc #(32) r2E(clk, reset, flushE, srcbD, srcbE);

floprc #(32) r3E(clk, reset, flushE, imm, signimmE);

floprc #(5) r4E(clk, reset, flushE, rsD, rsE);

floprc #(5) r5E(clk, reset, flushE, rtD, rtE);

floprc #(5) r6E(clk, reset, flushE, rdD, rdE);

mux3 #(32) forwardaemux(srcaE, resultW, aluoutM, forwardaE, srca2E);

mux3 #(32) forwardbemux(srcbE, resultW, aluoutM, forwardbE, srcb2E);

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

mux2 #(32) srcbmux(srcb2E, signimmE, alusrcE, srcb3E);

alu alu(srca2E, srcb3E, alucontrolE, aluoutE);

mux2 #(5) wrmux(rtE, rdE, regdstE, writeregE);

// Memory

flopr #(32) r1M(clk, reset, srcb2E, writedataM);

flopr #(32) r2M(clk, reset, aluoutE, aluoutM);

flopr #(5) r3M(clk, reset, writeregE, writeregM);

// Writeback

flopr #(32) r1W(clk, reset, aluoutM, aluoutW);

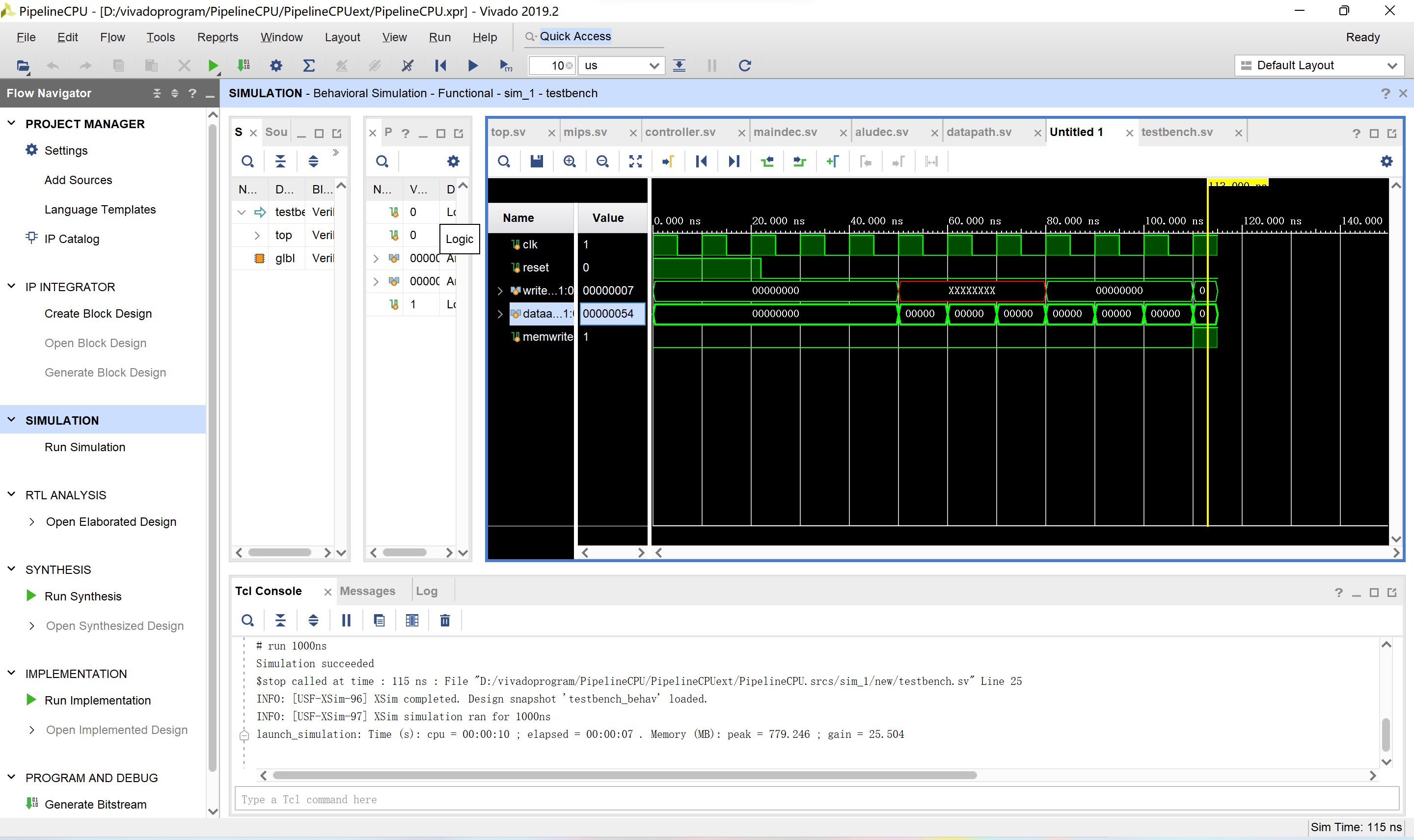
flopr #(32) r2W(clk, reset, readdataM, readdataW);

flopr #(5) r3W(clk, reset, writeregM, writeregW);

mux2 #(32) resmux(aluoutW, readdataW, memtoregW, resultW);

endmodule

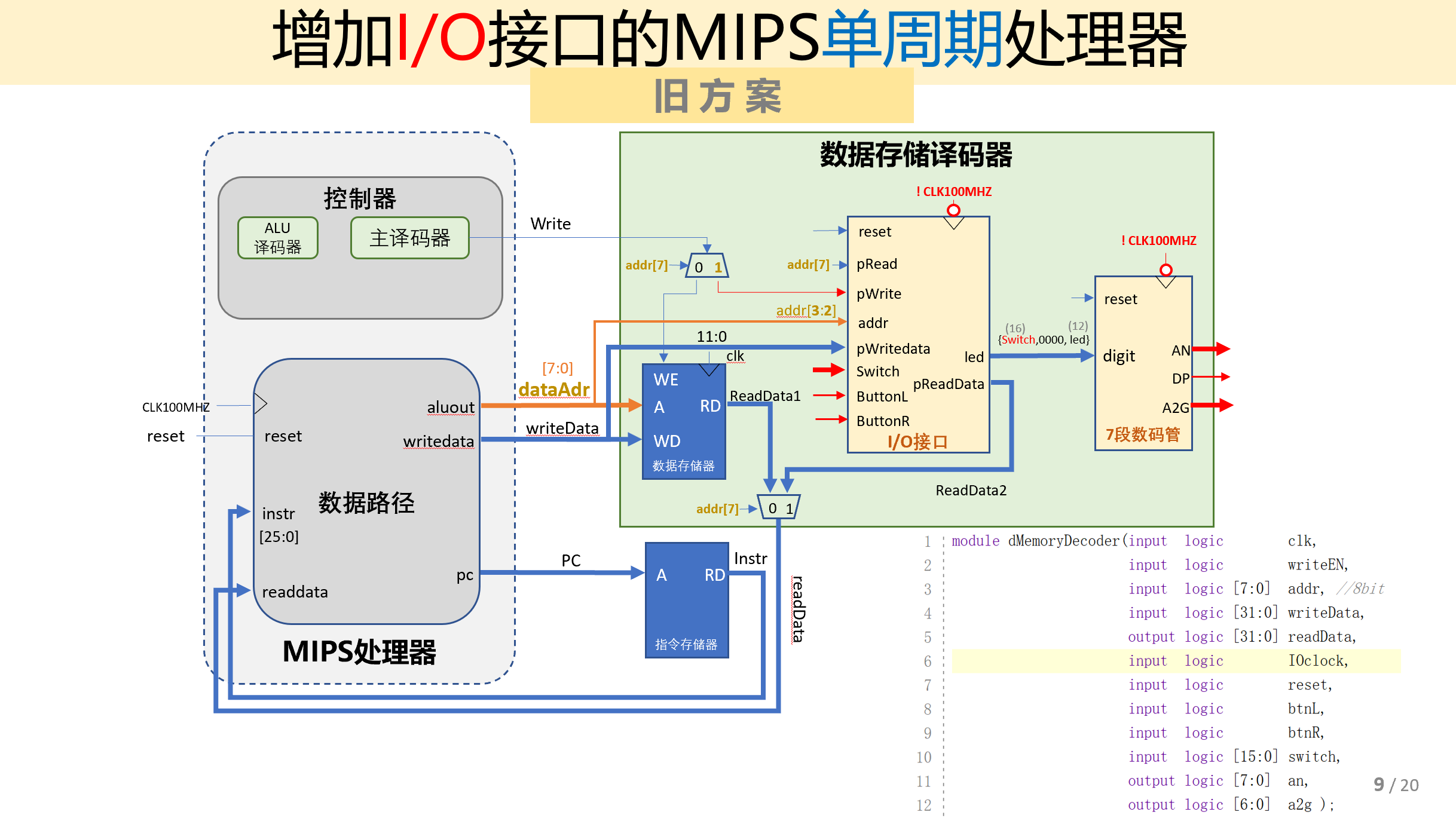
3. 仿真结果：



【三】流水线CPU\_IO：仿真+上板

1. 实验方案

流水线CPU增添IO接口作为CPU与外设的连接桥梁，获得的模块框图与单周期类似：



发现：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'b1000010;

'he: a2g = 7'b0010000;

'hf: a2g = 7'b0111000;

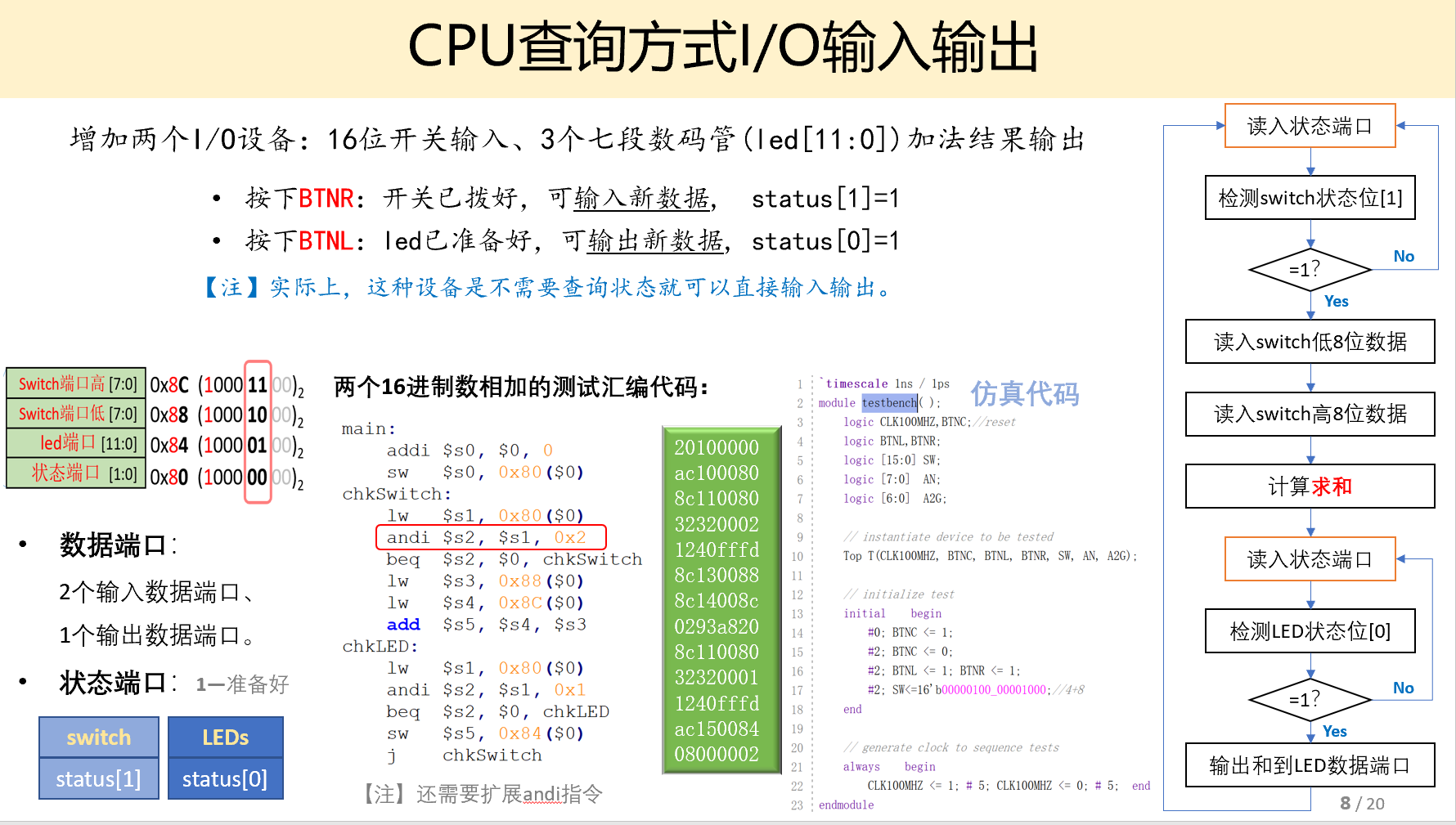
'h10:a2g = 7'b1110110;

default: a2g = 7'b1111111;

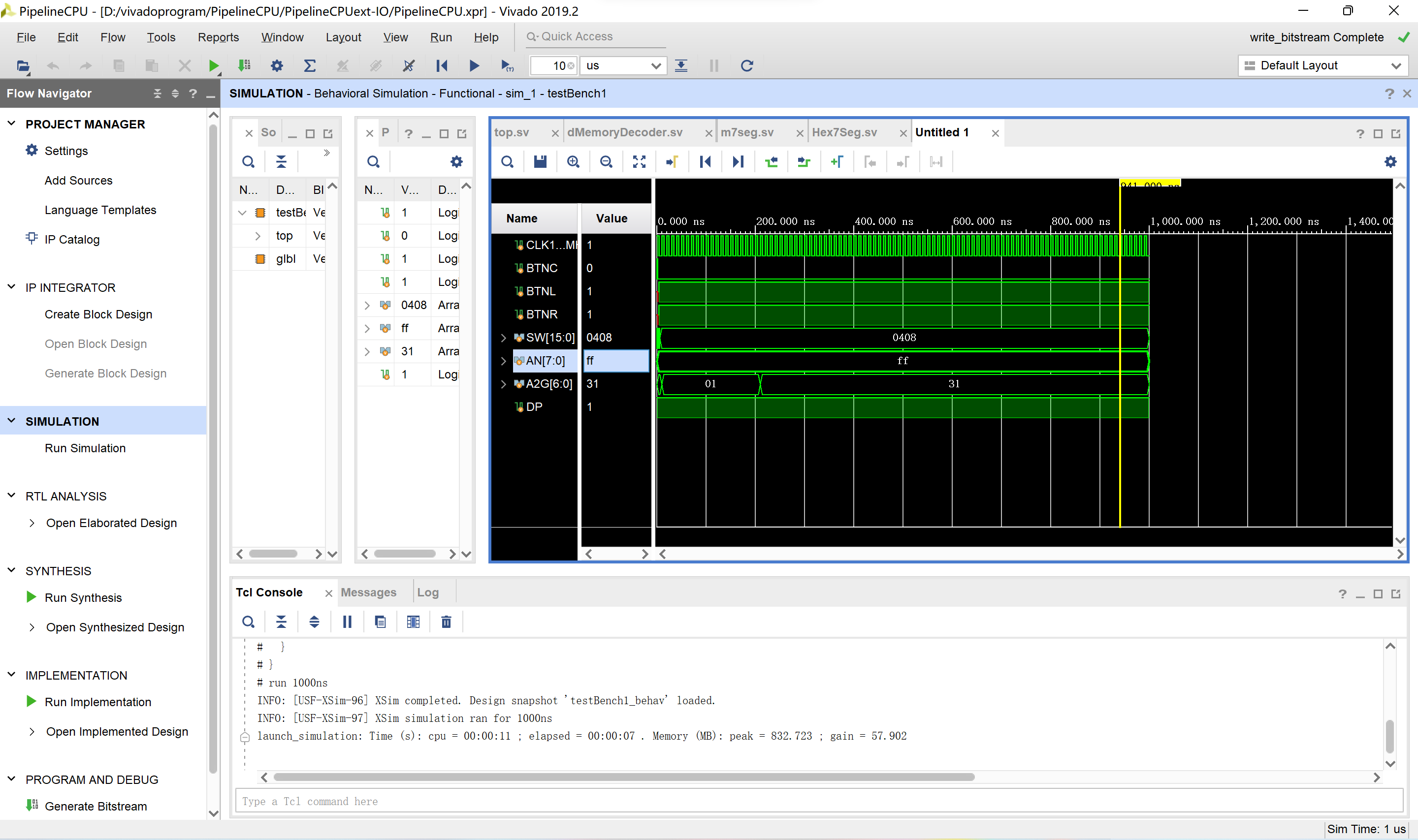
endcase

endmodule

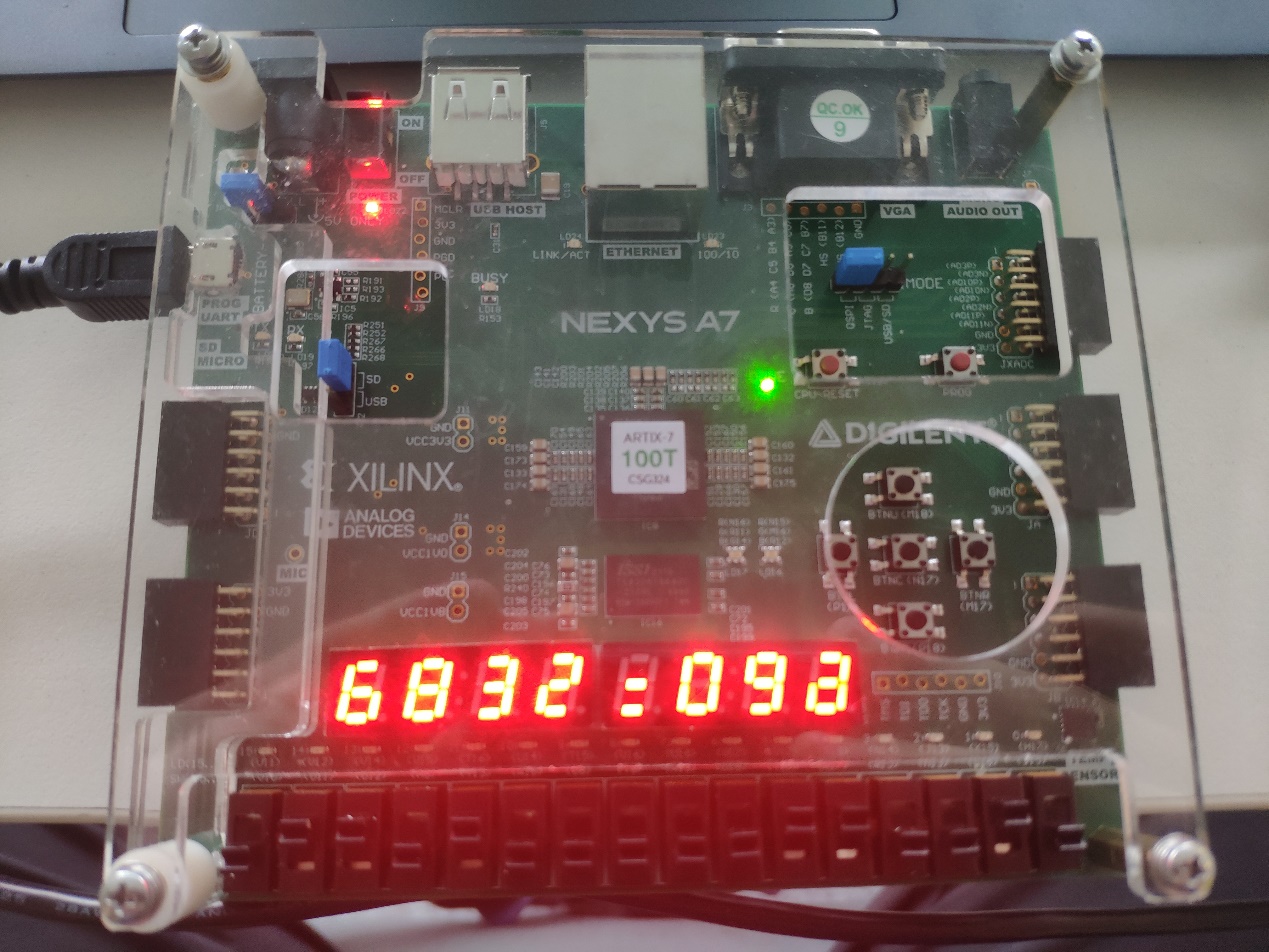
3.仿真



实验结果：



4.上板：（实验视频见文件夹）

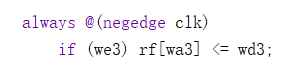


【四】问题回顾与总结反思

问题回顾：

未区分寄存器写/读的下降沿/上升沿

刚开始誊写寄存器写入代码时，未将时钟触发条件设置为下降沿，对寄存器写/读进行区分。



总结反思：

本次实验主要考察了在单周期基础上对于流水线CPU原理的进一步认识，其中重点涉及了对流水寄存器和冲突处理的理解。由于流水线CPU是将指令分为多个阶段依次执行，不同阶段需要使用流水寄存器进行状态保存。对于数据/指令冲突，需要设置单独的hazard模块进行处理。在进行实验前期构思设计时，需遵照每个阶段依次厘清指令所经过的状态，并判断冲突类型。

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、多周期CPU而言，流水线CPU因逻辑问题导致的错误进一步减少，调试时间也大大减少，说明对CPU的体系结构了解更深。由于粗心大意犯的错再次提醒在实验中细心与耐心的重要性。同时，在调试阶段，应该避免惯性思维，如认为寄存器触发条件一定为上升沿，导致未处理下降沿的错误一直没被发现。