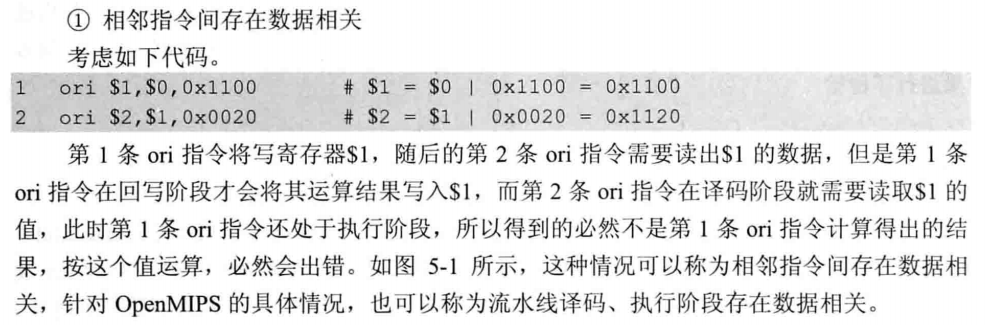
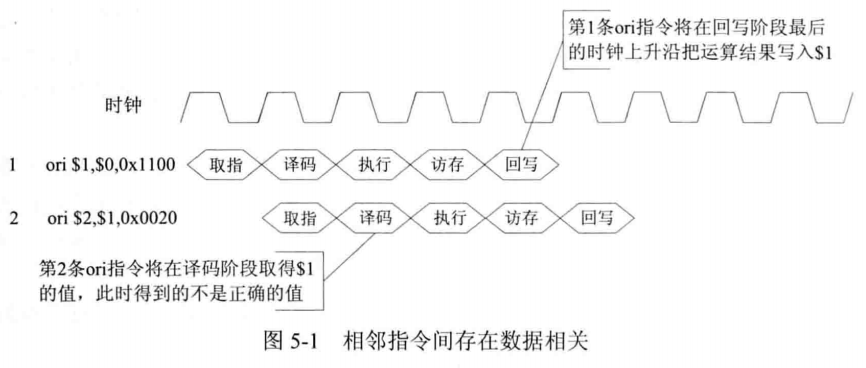
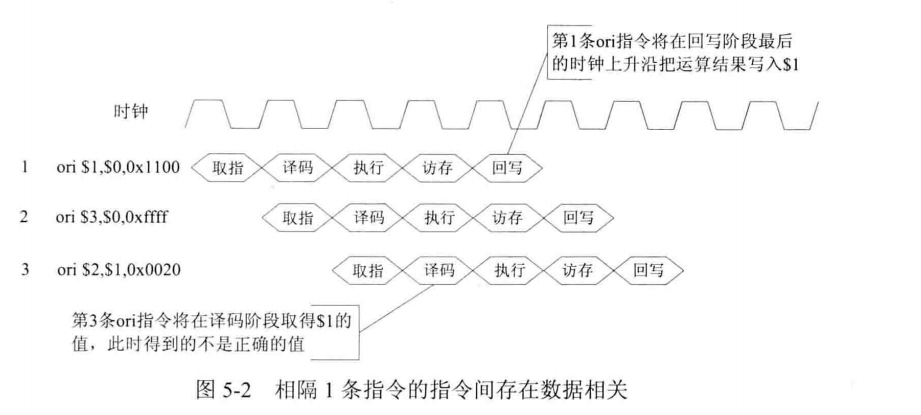
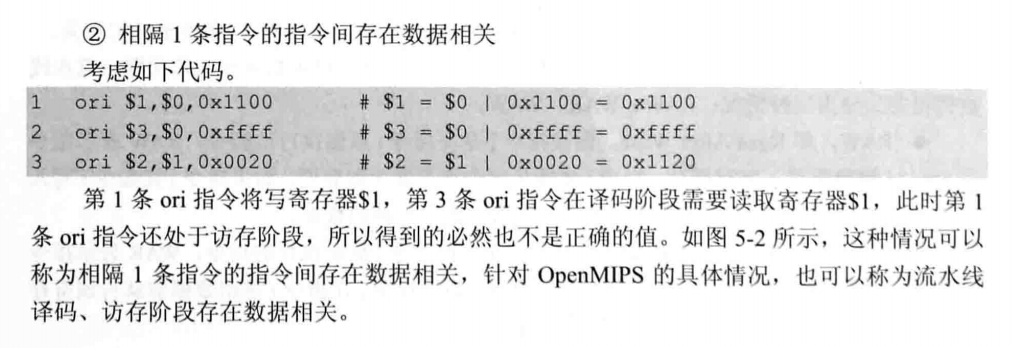
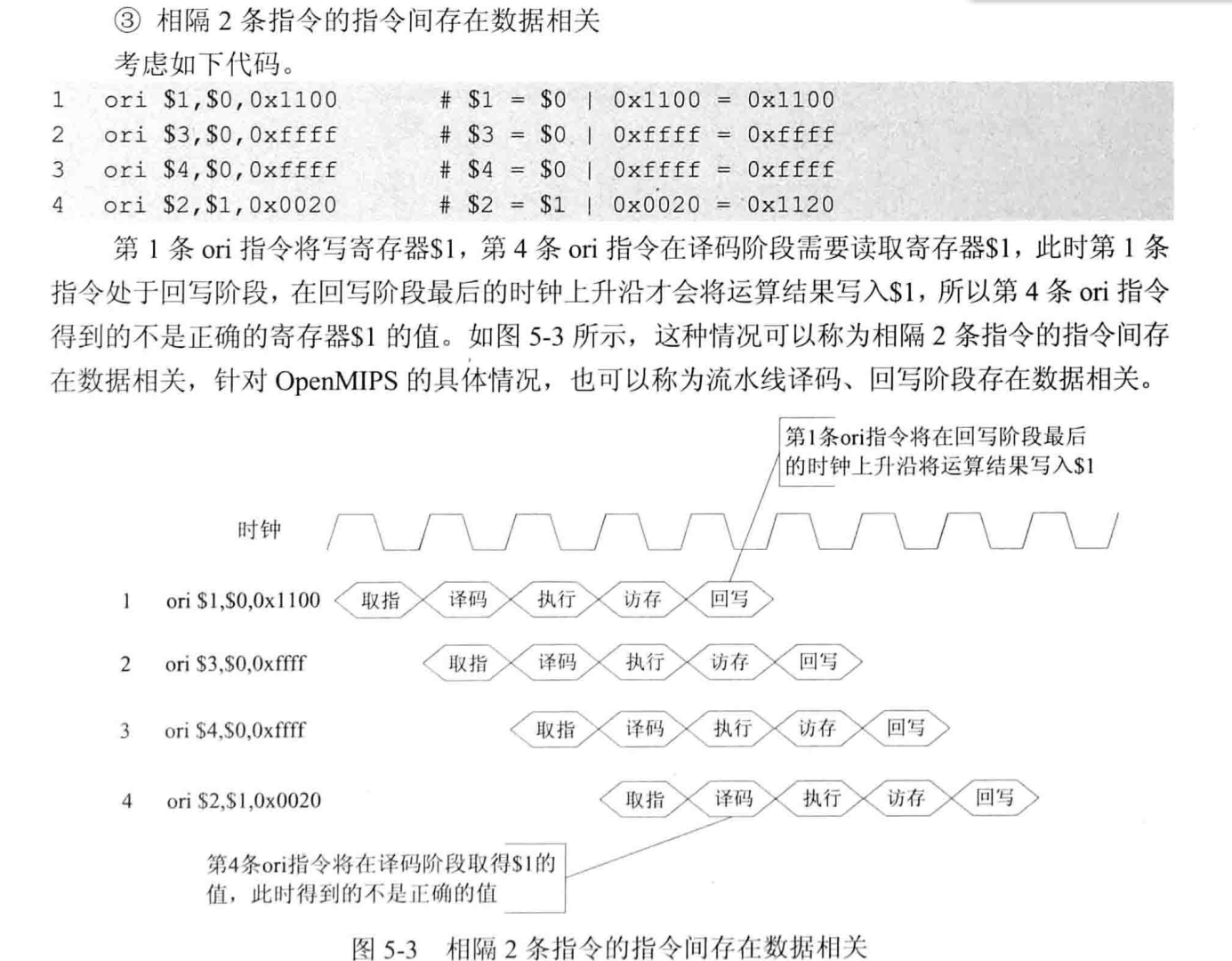
**解决数据相关问题并添加逻辑和移位指令**

# 解决数据相关









我们修改代码：

……

always @(\*) begin

if (rst == `RstEnable) begin

rdata1 <= `ZeroWord;

end else if (raddr1 == `RegNumLog2'h0) begin

rdata1 <= `ZeroWord;

end **else if ((raddr1 == waddr) && (we == `WriteEnable) && (re1 == `ReadEnable)) begin**

**rdata1 <= wdata;**

**end** else if (re1 == `ReadEnable) begin

rdata1 <= regs[raddr1];

end else begin

rdata1 <= `ZeroWord;

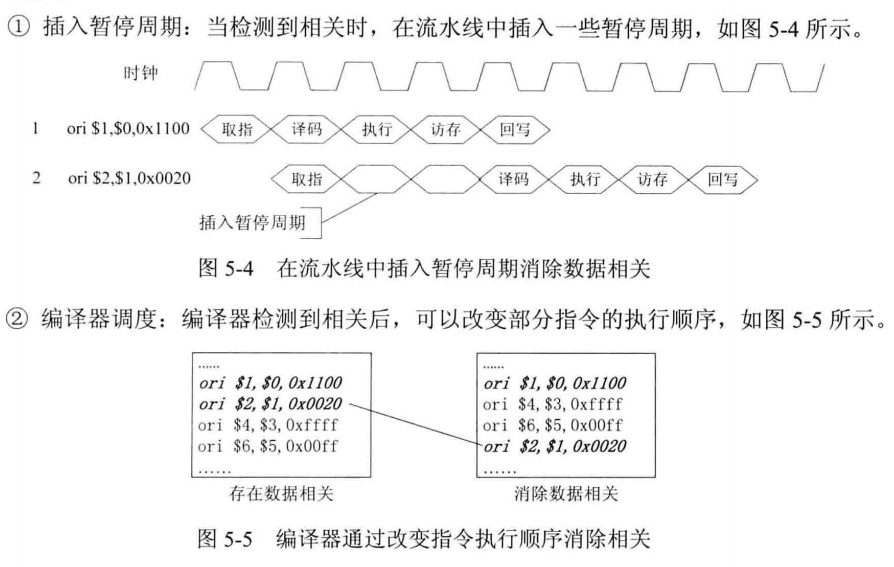
end

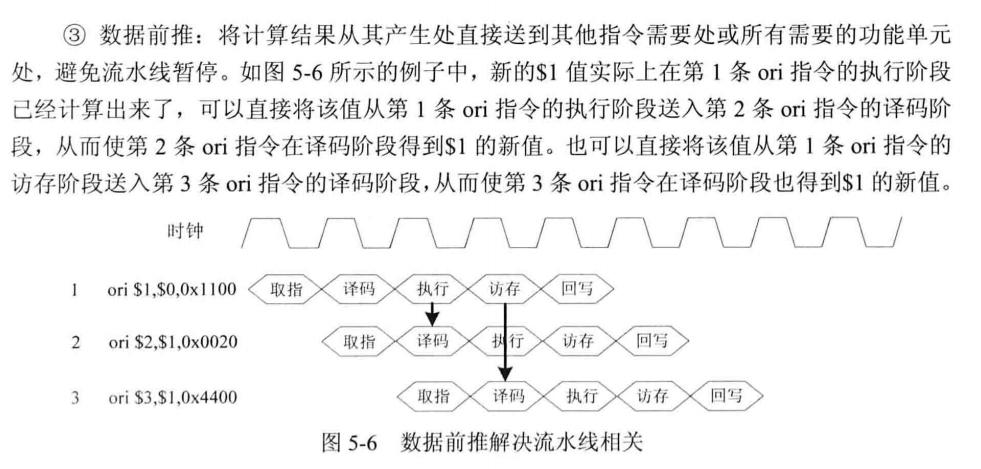
end

……

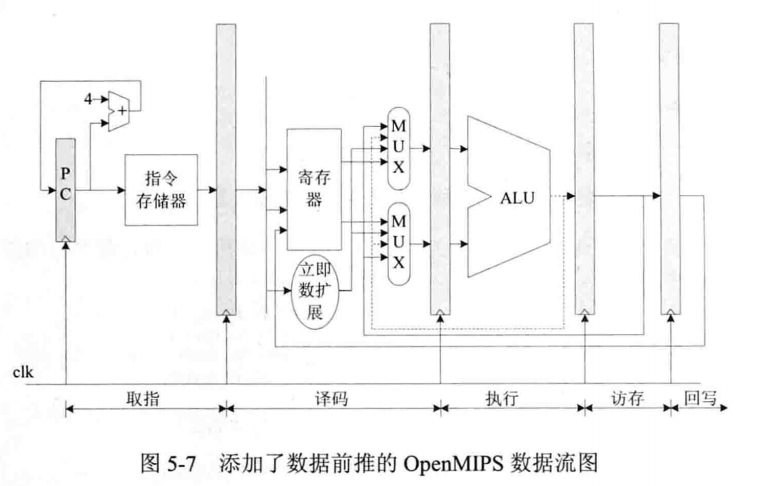
端口2同理

这样便解决了相隔两条指令的数据相关，对于相邻和相隔一条指令的这两种情况，有三种解决办法：

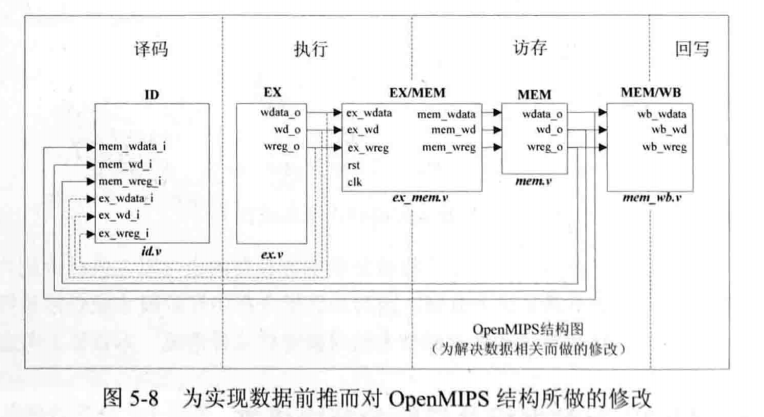




添加了数据前推的数据流图：



我们希望从执行（或访存阶段）直接获取上一阶段（或上上阶段）的地址和值，因此需要在id模块添加几个接口来接收来自执行或访存阶段的数据。增加的接口如图和如表：



修改id.v的代码如下：

……

//Fetch the result from the execution stage

input wire ex\_wreg\_i,

input wire [`RegBus] ex\_wdata\_i,

input wire [`RegAddrBus] ex\_wd\_i,

//Fetch the result from the mem stage

input wire mem\_wreg\_i,

input wire [`RegBus] mem\_wdata\_i,

input wire [`RegAddrBus] mem\_wd\_i,

……

always @(\*) begin

if (rst == `RstEnable) begin

reg1\_o <= `ZeroWord;

end **else if ((reg1\_read\_o == 1'b1) && (ex\_wreg\_i == 1'b1) && (ex\_wd\_i == reg1\_addr\_o)) begin**

**reg1\_o <= ex\_wdata\_i;**

**end else if ((reg1\_read\_o == 1'b1) && (mem\_wreg\_i == 1'b1) && (mem\_wd\_i == reg1\_addr\_o)) begin**

**reg1\_o <= mem\_wdata\_i;**

**end** else if (reg1\_read\_o == 1'b1) begin

reg1\_o <= reg1\_data\_i; //Regfile read the value of port1

end else if (reg1\_read\_o == 1'b0) begin

reg1\_o <= imm; //immediate number

end else begin

reg1\_o <= `ZeroWord;

end

end

……

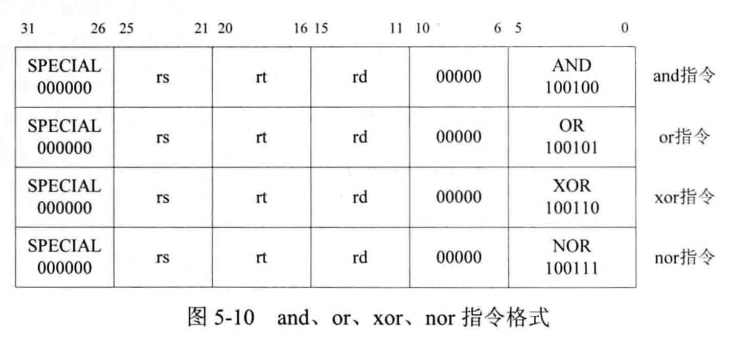
端口2同理

至此，我们就解决了数据相关问题，在下一节我们将进行简单的测试。

# 逻辑指令和移位指令的实现

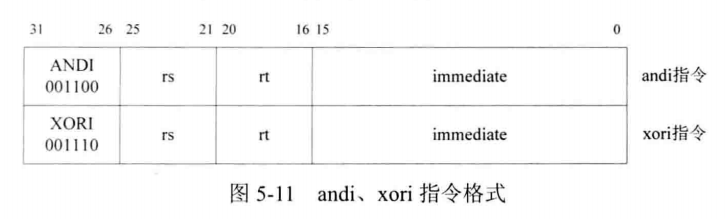
## 2.1 and、or、xor、nor

指令格式：



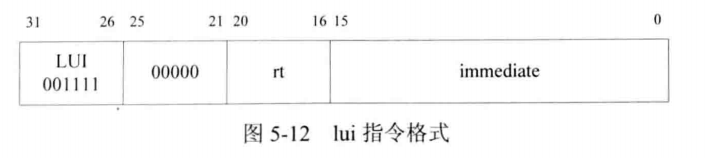
## 2.2 andi、xori

指令格式：



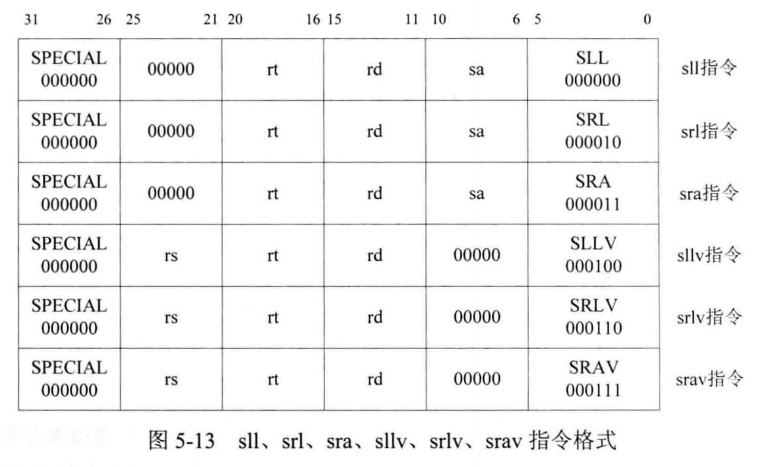
## 2.3 lui指令

指令格式：

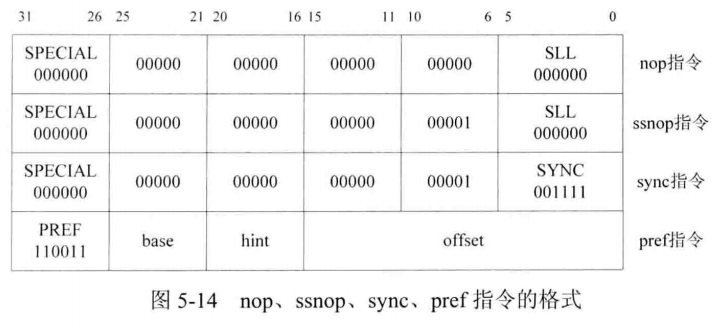


## 2.4 sll、sllv、sra、srav、srl、srlv

指令格式：



## 2.5 nop、ssnop、sync、pref



仔细观察我们发现，nop指令和ssnop指令和sll指令一致，是因为我们为了简化指令，使用逻辑左移指令代表空指令，实际上逻辑左移0位就是没有执行任何操作。

## 2.6 修改译码阶段的ID模块

首先添加宏定义：

//AluOp

`define EXE\_AND\_OP 8'b00100100

`define EXE\_OR\_OP 8'b00100101

`define EXE\_XOR\_OP 8'b00100110

`define EXE\_NOR\_OP 8'b00100111

`define EXE\_ANDI\_OP 8'b01011001

`define EXE\_ORI\_OP 8'b01011010

`define EXE\_XORI\_OP 8'b01011011

`define EXE\_LUI\_OP 8'b01011100

`define EXE\_SLL\_OP 8'b01111100

`define EXE\_SLLV\_OP 8'b00000100

`define EXE\_SRL\_OP 8'b00000010

`define EXE\_SRLV\_OP 8'b00000110

`define EXE\_SRA\_OP 8'b00000011

`define EXE\_SRAV\_OP 8'b00000111

`define EXE\_NOP\_OP 8'b00000000

`define EXE\_RES\_SHIFT 3'b010

`define EXE\_AND 6'b100100 //The function code of 'AND' instruction

`define EXE\_OR 6'b100101 //The function code of 'OR' instruction

`define EXE\_XOR 6'b100110 //The function code of 'XOR' instruction

`define EXE\_NOR 6'b100111 //The function code of 'NOR' instruction

`define EXE\_ANDI 6'b001100 //The function code of 'ANDI' instruction

`define EXE\_ORI 6'b001101 //The function code of 'ORI' instruction

`define EXE\_XORI 6'b001110 //The function code of 'XORI' instruction

`define EXE\_LUI 6'b001111 //The function code of 'LUI' instruction

`define EXE\_SLL 6'b000000 //The function code of 'SLL' instruction

`define EXE\_SLLV 6'b000100 //The function code of 'SLLV' instruction

`define EXE\_SRL 6'b000010 //The function code of 'SRL' instruction

`define EXE\_SRLV 6'b000110 //The function code of 'SRLV' instruction

`define EXE\_SRA 6'b000011 //The function code of 'SRA' instruction

`define EXE\_SRAV 6'b000111 //The function code of 'SRAV' instruction

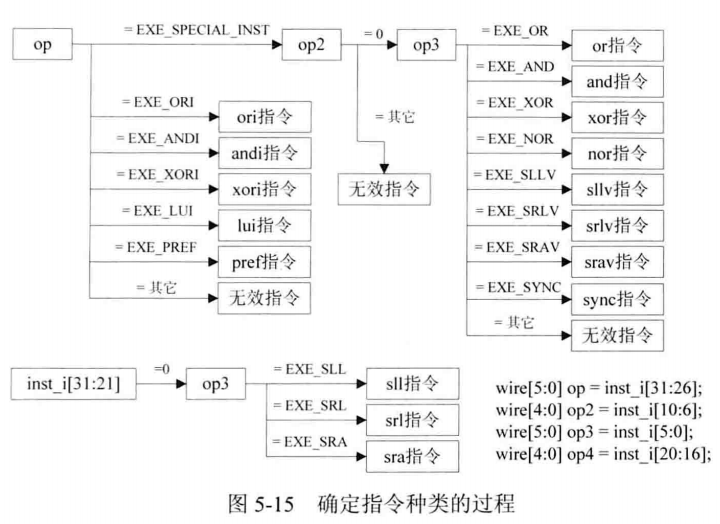
`define EXE\_SYNC 6'b001111 //The function code of 'SYNC' instruction

`define EXE\_PREF 6'b110011 //The instruction code of 'PREF' instruction

`define EXE\_SPECIAL\_INST 6'b000000 //The instruction code of type'SPECIAL'

`define EXE\_NOP 6'b000000

`define SSNOP 32'b00000000\_00000000\_00000000\_01000000

值得注意的是，运算子类型aluop是8位，而每一个运算的功能码是6位，我们知道，判定一个运算类型的方式是判断一条指令的前六位和后六位（[31:26], [5:0]），我们用一个8位的运算子类型和一个3位的运算类型来唯一标识一种运算。具体确定运算种类的过程如下：

具体修改的ID模块代码见[**附录1**](#_4.1_ID模块)

## 2.7 修改执行阶段的EX模块

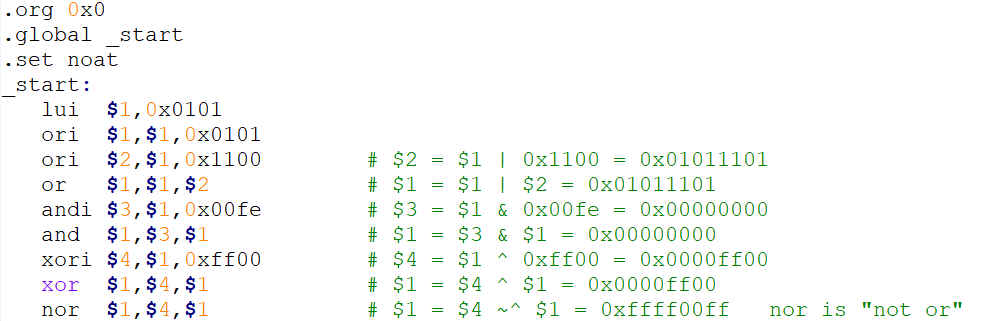
对于执行模块，相对来说会简单一点，只需简单的几步就可以实现逻辑运算，例如：逻辑左移：

值得关注的是算术右移，因为我们知道，算术右移需要在左边补上符号位，因此我们想到把符号位单独拿出来，并扩展成32位。试想，若将一个32位数算数右移n位，那么左边将会空出n位，用逻辑右移后，左边n位将补0，而我们又知道逻辑或运算可以实现将结果变成自身，因此我们考虑将逻辑右移后的左边n位与符号位做或运算，那么左边n位将变成符号位，具体实现如下

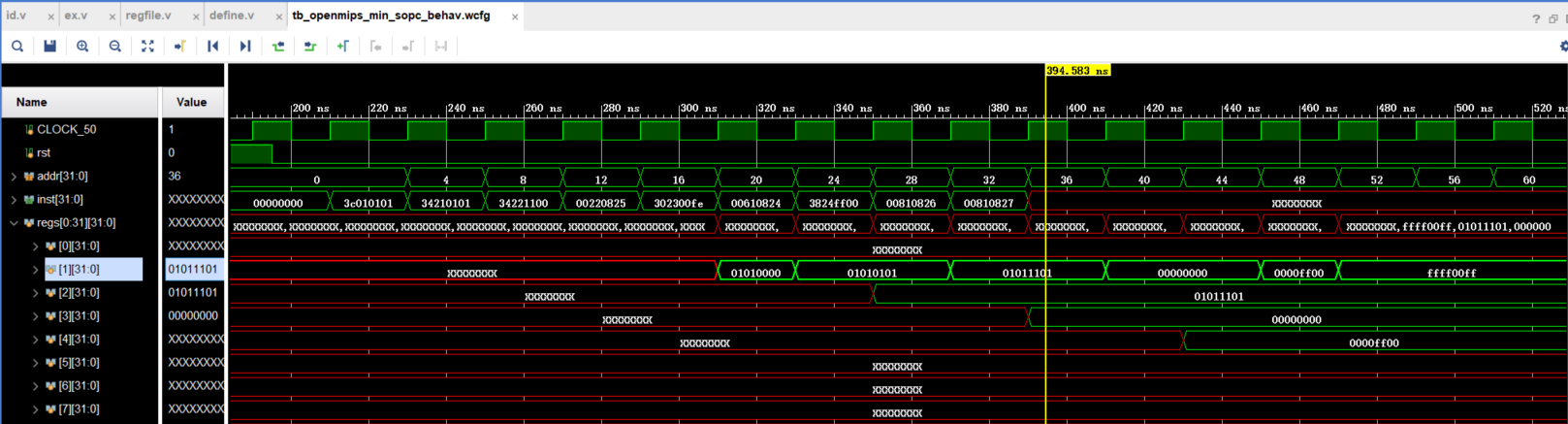
具体修改的EX模块代码见[**附录2**](#_4.2_EX模块)

# 测试及分析

## 3.1 逻辑指令测试代码

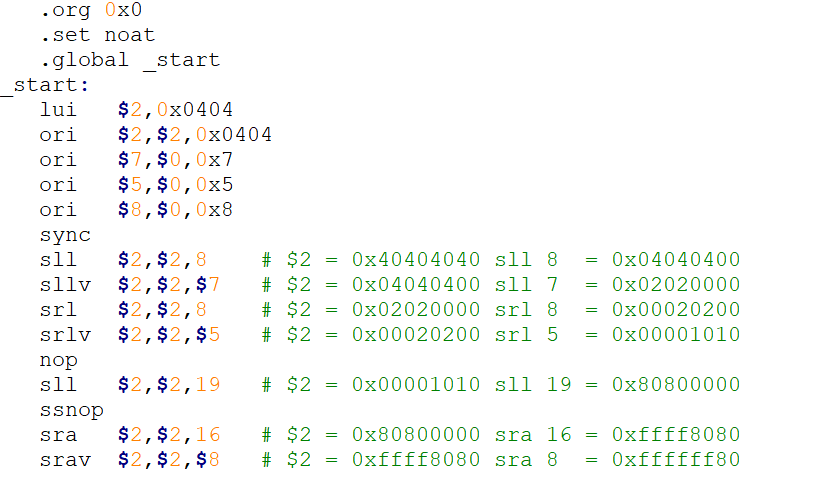


仿真结果：

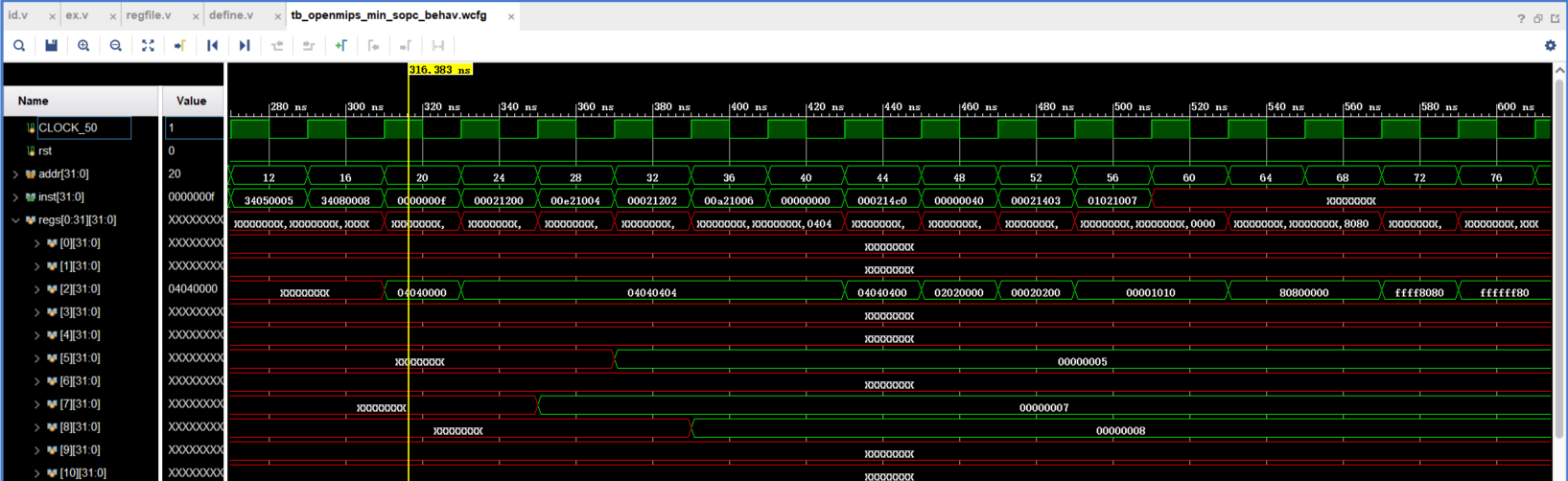


我们发现一次执行了指令，并且结果正确

## 3.2 移位指令测试代码



仿真结果：



最终2号寄存器的值为FFFFFF80，答案正确。

# 附录

## 4.1 ID模块

……

always @(\*) begin

if (rst == `RstEnable) begin

aluop\_o <= `EXE\_NOP\_OP;

alusel\_o <= `EXE\_RES\_NOP;

wd\_o <= `NOPRegAddr;

wreg\_o <= `WriteDisable;

instvalid <= `InstValid;

reg1\_read\_o <= 1'b0;

reg2\_read\_o <= 1'b0;

reg1\_addr\_o <= `NOPRegAddr;

reg2\_addr\_o <= `NOPRegAddr;

imm <= 32'h0;

end else begin

aluop\_o <= `EXE\_NOP\_OP;

alusel\_o <= `EXE\_RES\_NOP;

wd\_o <= inst\_i[15:11];

wreg\_o <= `WriteDisable;

instvalid <= `InstInValid;

reg1\_read\_o <= 1'b0;

reg2\_read\_o <= 1'b0;

reg1\_addr\_o <= inst\_i[25:21]; //Read the register address of port1 from Regfile

reg2\_addr\_o <= inst\_i[20:16]; //Read the register address of port2 from Regfile

imm <= `ZeroWord;

case (op)

`EXE\_SPECIAL\_INST: begin

case (op2)

5'b00000: begin

case (op3)

`EXE\_OR: begin

wreg\_o <= `WriteEnable;

aluop\_o <= `EXE\_OR\_OP;

alusel\_o <= `EXE\_RES\_LOGIC;

reg1\_read\_o <= 1'b1;

reg2\_read\_o <= 1'b1;

instvalid <= `InstValid;

end

`EXE\_AND: begin

wreg\_o <= `WriteEnable;

aluop\_o <= `EXE\_AND\_OP;

alusel\_o <= `EXE\_RES\_LOGIC;

reg1\_read\_o <= 1'b1;

reg2\_read\_o <= 1'b1;

instvalid <= `InstValid;

end

`EXE\_XOR: begin

wreg\_o <= `WriteEnable;

aluop\_o <= `EXE\_XOR\_OP;

alusel\_o <= `EXE\_RES\_LOGIC;

reg1\_read\_o <= 1'b1;

reg2\_read\_o <= 1'b1;

instvalid <= `InstValid;

end

`EXE\_NOR: begin

wreg\_o <= `WriteEnable;

aluop\_o <= `EXE\_NOR\_OP;

alusel\_o <= `EXE\_RES\_LOGIC;

reg1\_read\_o <= 1'b1;

reg2\_read\_o <= 1'b1;

instvalid <= `InstValid;

end

`EXE\_SLLV: begin

wreg\_o <= `WriteEnable;

aluop\_o <= `EXE\_SLL\_OP;

alusel\_o <= `EXE\_RES\_SHIFT;

reg1\_read\_o <= 1'b1;

reg2\_read\_o <= 1'b1;

instvalid <= `InstValid;

end

`EXE\_SRLV: begin

wreg\_o <= `WriteEnable;

aluop\_o <= `EXE\_SRL\_OP;

alusel\_o <= `EXE\_RES\_SHIFT;

reg1\_read\_o <= 1'b1;

reg2\_read\_o <= 1'b1;

instvalid <= `InstValid;

end

`EXE\_SRAV: begin

wreg\_o <= `WriteEnable;

aluop\_o <= `EXE\_SRA\_OP;

alusel\_o <= `EXE\_RES\_SHIFT;

reg1\_read\_o <= 1'b1;

reg2\_read\_o <= 1'b1;

instvalid <= `InstValid;

end

`EXE\_SYNC: begin

wreg\_o <= `WriteEnable;

aluop\_o <= `EXE\_NOP\_OP;

alusel\_o <= `EXE\_RES\_NOP;

reg1\_read\_o <= 1'b0;

reg2\_read\_o <= 1'b1;

instvalid <= `InstValid;

end

default: begin

end

endcase

end

default: begin

end

endcase

end

`EXE\_ORI: begin //Judge whether it is the ORI instruction by the value of OP

//The instruction of ORI need to put the result to the destination register

wreg\_o <= `WriteEnable;

//The sub-type of calculation is 'OR'

aluop\_o <= `EXE\_OR\_OP;

//The type of calculation is Logic

alusel\_o <= `EXE\_RES\_LOGIC;

//Need the Regfile read port1 to read the register

reg1\_read\_o <= 1'b1;

//Not need the Regfile read2 to read the register

reg2\_read\_o <= 1'b0;

//The immediate number

imm <= {16'h0, inst\_i[15:0]};

//The register address which the instruction will execuate

wd\_o <= inst\_i[20:16];

//The instruction of ORI is valid

instvalid <= `InstValid;

end

`EXE\_ANDI: begin

wreg\_o <= `WriteEnable;

aluop\_o <= `EXE\_AND\_OP;

alusel\_o <= `EXE\_RES\_LOGIC;

reg1\_read\_o <= 1'b1;

reg2\_read\_o <= 1'b0;

imm <= {16'h0, inst\_i[15:0]};

wd\_o <= inst\_i[20:16];

instvalid <= `InstValid;

end

`EXE\_XORI: begin

wreg\_o <= `WriteEnable;

aluop\_o <= `EXE\_XOR\_OP;

alusel\_o <= `EXE\_RES\_LOGIC;

reg1\_read\_o <= 1'b1;

reg2\_read\_o <= 1'b0;

imm <= {16'h0, inst\_i[15:0]};

wd\_o <= inst\_i[20:16];

instvalid <= `InstValid;

end

`EXE\_LUI: begin

wreg\_o <= `WriteEnable;

aluop\_o <= `EXE\_OR\_OP;

alusel\_o <= `EXE\_RES\_LOGIC;

reg1\_read\_o <= 1'b1;

reg2\_read\_o <= 1'b0;

imm <= {inst\_i[15:0], 16'h0};

wd\_o <= inst\_i[20:16];

instvalid <= `InstValid;

end

`EXE\_PREF: begin

wreg\_o <= `WriteEnable;

aluop\_o <= `EXE\_NOP\_OP;

alusel\_o <= `EXE\_RES\_NOP;

reg1\_read\_o <= 1'b0;

reg2\_read\_o <= 1'b0;

instvalid <= `InstValid;

end

default: begin

end

endcase //case op

if (inst\_i[31:21] == 11'b00000000000) begin

$monitor("SHIFT DONE");

if (op3 == `EXE\_SLL) begin

wreg\_o <= `WriteEnable;

aluop\_o <= `EXE\_SLL\_OP;

alusel\_o <= `EXE\_RES\_SHIFT;

reg1\_read\_o <= 1'b0;

reg2\_read\_o <= 1'b1;

imm[4:0] <= inst\_i[10:6];

wd\_o <= inst\_i[15:11];

instvalid <= `InstValid;

end else if (op3 == `EXE\_SRL) begin

wreg\_o <= `WriteEnable;

aluop\_o <= `EXE\_SRL\_OP;

alusel\_o <= `EXE\_RES\_SHIFT;

reg1\_read\_o <= 1'b0;

reg2\_read\_o <= 1'b1;

imm[4:0] <= inst\_i[10:6];

wd\_o <= inst\_i[15:11];

instvalid <= `InstValid;

end else if (op3 == `EXE\_SRA) begin

wreg\_o <= `WriteEnable;

aluop\_o <= `EXE\_SRA\_OP;

alusel\_o <= `EXE\_RES\_SHIFT;

reg1\_read\_o <= 1'b0;

reg2\_read\_o <= 1'b1;

imm[4:0] <= inst\_i[10:6];

wd\_o <= inst\_i[15:11];

instvalid <= `InstValid;

end

end

end //if done

end //always done

……

## 4.2 EX模块

……

always @(\*) begin

if (rst == `RstEnable) begin

logicout <= `ZeroWord;

end else begin

case (aluop\_i)

`EXE\_OR\_OP: begin

logicout <= reg1\_i | reg2\_i;

end

`EXE\_AND\_OP: begin

logicout <= reg1\_i & reg2\_i;

end

`EXE\_NOR\_OP: begin

logicout <= ~(reg1\_i | reg2\_i);

end

`EXE\_XOR\_OP: begin

logicout <= reg1\_i ^ reg2\_i;

end

default: begin

logicout <= `ZeroWord;

end

endcase

end //if end

end //always end

//SHIFT

always @(\*) begin

if (rst == `RstEnable) begin

shiftres <= `ZeroWord;

end else begin

case (aluop\_i)

`EXE\_SLL\_OP: begin

shiftres <= reg2\_i << reg1\_i[4:0];

end

`EXE\_SRL\_OP: begin

shiftres <= reg2\_i >> reg1\_i[4:0];

end

`EXE\_SRA\_OP: begin

shiftres <= ( {32 {reg2\_i[31] } } << (6'd32 - {1'b0, reg1\_i[4:0] } ) ) | (reg2\_i >> reg1\_i[4:0]);

end

default: begin

shiftres <= `ZeroWord;

end

endcase

end

end

//\*\*\*\*\*\*\*\*\* Chapter 2 : Choose one result by type of the instruction of 'alusel\_i' \*\*\*\*\*\*\*\*

always @(\*) begin

wd\_o <= wd\_i;

wreg\_o <= wreg\_i;

case (alusel\_i)

`EXE\_RES\_LOGIC: begin

wdata\_o <= logicout;

end

`EXE\_RES\_SHIFT: begin

wdata\_o <= shiftres;

end

default: begin

wdata\_o <= `ZeroWord;

end

endcase

end

……