解决数据相关问题并添加逻辑和移位指令

一、 解决数据相关

① 相邻指令间存在数据相关

考虑如下代码。

```
1   ori $1,$0,0x1100        # $1 = $0 | 0x1100 = 0x1100
2   ori $2,$1,0x0020        # $2 = $1 | 0x0020 = 0x1120
```

第 1 条 ori 指令将写寄存器$1，随后的第 2 条 ori 指令需要读出$1 的数据，但是第 1 条 ori 指令在回写阶段才会将其运算结果写入$1，而第 2 条 ori 指令在译码阶段就需要读取$1 的值，此时第 1 条 ori 指令还处于执行阶段，所以得到的必然不是第 1 条 ori 指令计算得出的结果，按这个值运算，必然会出错。如图 5-1 所示，这种情况可以称为相邻指令间存在数据相关，针对 OpenMIPS 的具体情况，也可以称为流水线译码、执行阶段存在数据相关。

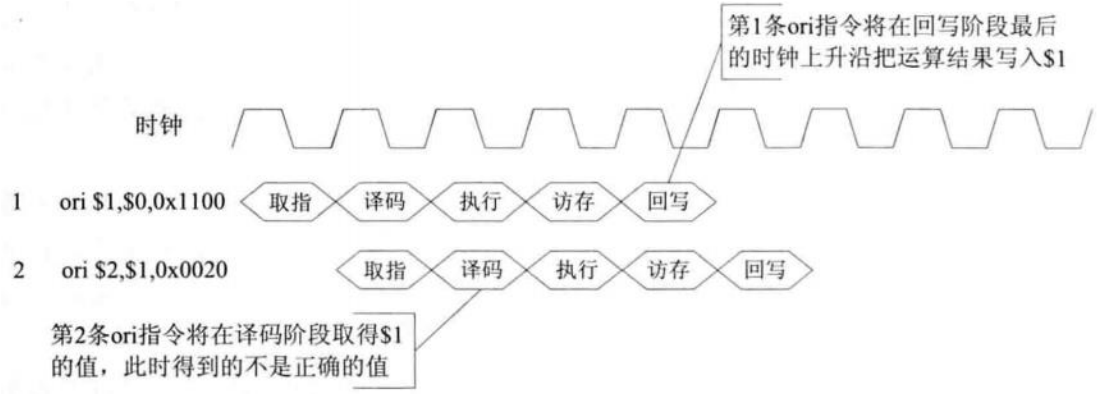


图 5-1　相邻指令间存在数据相关

② 相隔 1 条指令的指令间存在数据相关

考虑如下代码。

```
1   ori $1,$0,0x1100        # $1 = $0 | 0x1100 = 0x1100
2   ori $3,$0,0xffff        # $3 = $0 | 0xffff = 0xffff
3   ori $2,$1,0x0020        # $2 = $1 | 0x0020 = 0x1120
```

第 1 条 ori 指令将写寄存器$1，第 3 条 ori 指令在译码阶段需要读取寄存器$1，此时第 1 条 ori 指令还处于访存阶段，所以得到的必然也不是正确的值。如图 5-2 所示，这种情况可以称为相隔 1 条指令的指令间存在数据相关，针对 OpenMIPS 的具体情况，也可以称为流水线译码、访存阶段存在数据相关。

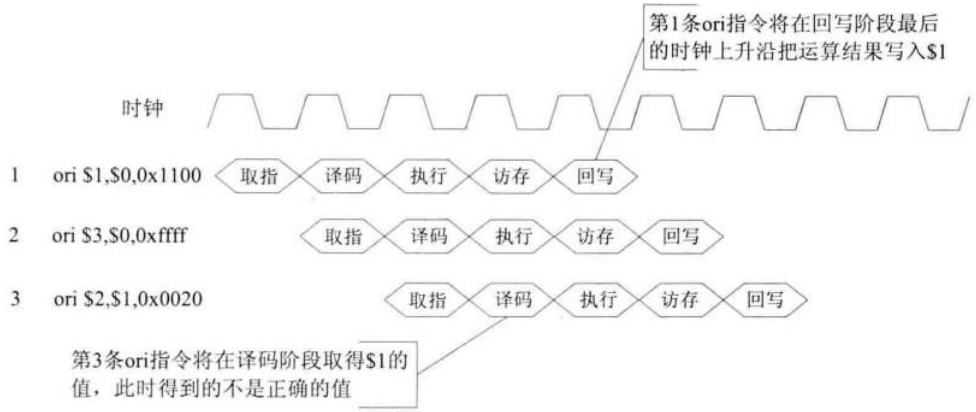


图 5-2　相隔 1 条指令的指令间存在数据相关

③ 相隔 2 条指令的指令间存在数据相关

考虑如下代码。

```
1   ori $1,$0,0x1100        # $1 = $0 | 0x1100 = 0x1100
2   ori $3,$0,0xffff        # $3 = $0 | 0xffff = 0xffff
3   ori $4,$0,0xffff        # $4 = $0 | 0xffff = 0xffff
4   ori $2,$1,0x0020        # $2 = $1 | 0x0020 = 0x1120
```

第 1 条 ori 指令将写寄存器$1，第 4 条 ori 指令在译码阶段需要读取寄存器$1，此时第 1 条指令处于回写阶段，在回写阶段最后的时钟上升沿才会将运算结果写入$1，所以第 4 条 ori 指令得到的不是正确的寄存器$1 的值。如图 5-3 所示，这种情况可以称为相隔 2 条指令的指令间存在数据相关，针对 OpenMIPS 的具体情况，也可以称为流水线译码、回写阶段存在数据相关。

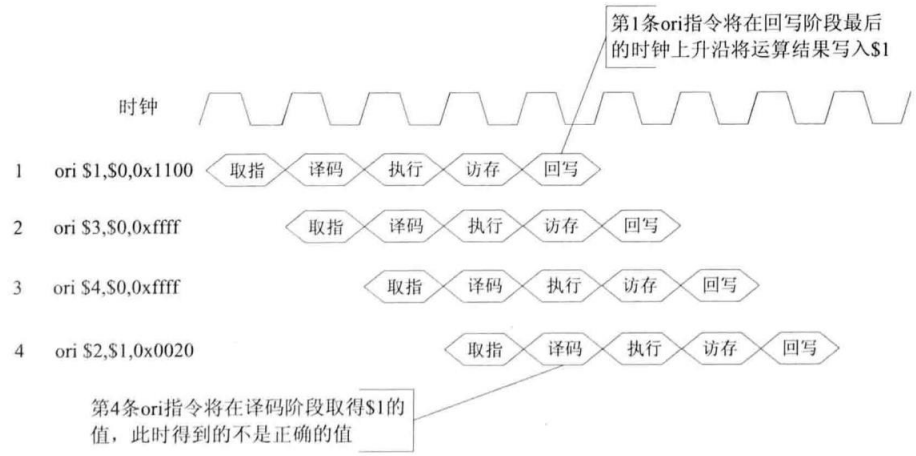


图 5-3　相隔 2 条指令的指令间存在数据相关

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：

① 插入暂停周期：当检测到相关时，在流水线中插入一些暂停周期，如图 5-4 所示。

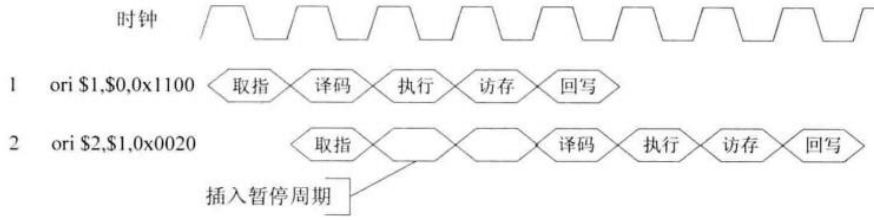


图 5-4　在流水线中插入暂停周期消除数据相关

② 编译器调度：编译器检测到相关后，可以改变部分指令的执行顺序，如图 5-5 所示。

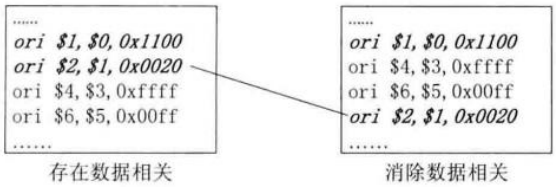


图 5-5　编译器通过改变指令执行顺序消除相关

③ 数据前推：将计算结果从其产生处直接送到其他指令需要处或所有需要的功能单元处，避免流水线暂停。如图 5-6 所示的例子中，新的$1 值实际上在第 1 条 ori 指令的执行阶段已经计算出来了，可以直接将该值从第 1 条 ori 指令的执行阶段送入第 2 条 ori 指令的译码阶段，从而使第 2 条 ori 指令在译码阶段得到$1 的新值。也可以直接将该值从第 1 条 ori 指令的访存阶段送入第 3 条 ori 指令的译码阶段，从而使第 3 条 ori 指令在译码阶段也得到$1 的新值。

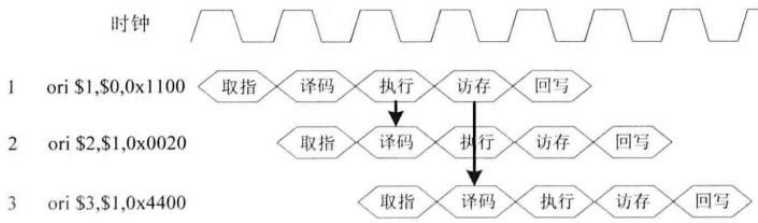


图 5-6　数据前推解决流水线相关

添加了数据前推的数据流图：

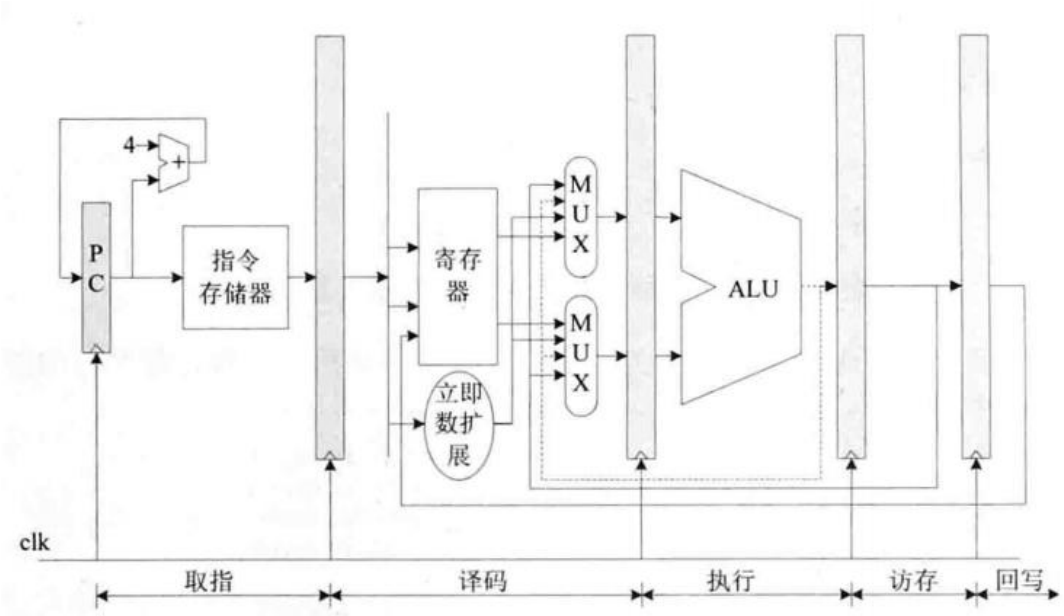


图 5-7　添加了数据前推的 OpenMIPS 数据流图

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：

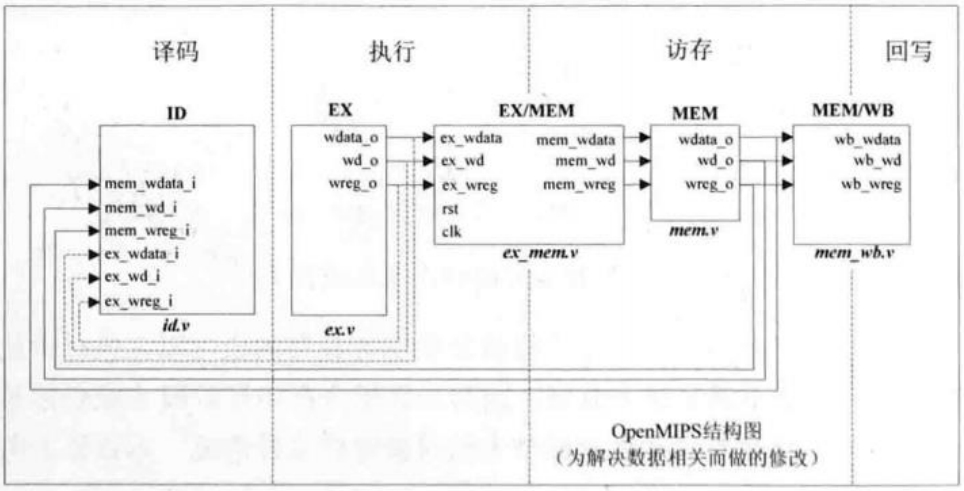


图 5-8　为实现数据前推而对 OpenMIPS 结构所做的修改

表 5-1　ID 模块要增加的接口

序　　号	接 口 名	宽度（bit）	输入/输出	作　　用
1	ex_wreg_i	1	输入	处于执行阶段的指令是否要写目的寄存器
2	ex_wd_i	5	输入	处于执行阶段的指令要写的目的寄存器地址
3	ex_wdata_i	32	输入	处于执行阶段的指令要写入目的寄存器的数据
4	mem_wreg_i	1	输入	处于访存阶段的指令是否要写目的寄存器
5	mem_wd_i	5	输入	处于访存阶段的指令要写的目的寄存器地址
6	mem_wdata_i	32	输入	处于访存阶段的指令要写入目的寄存器的数据

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。

二、 逻辑指令和移位指令的实现

2.1 and、or、xor、nor

指令格式：

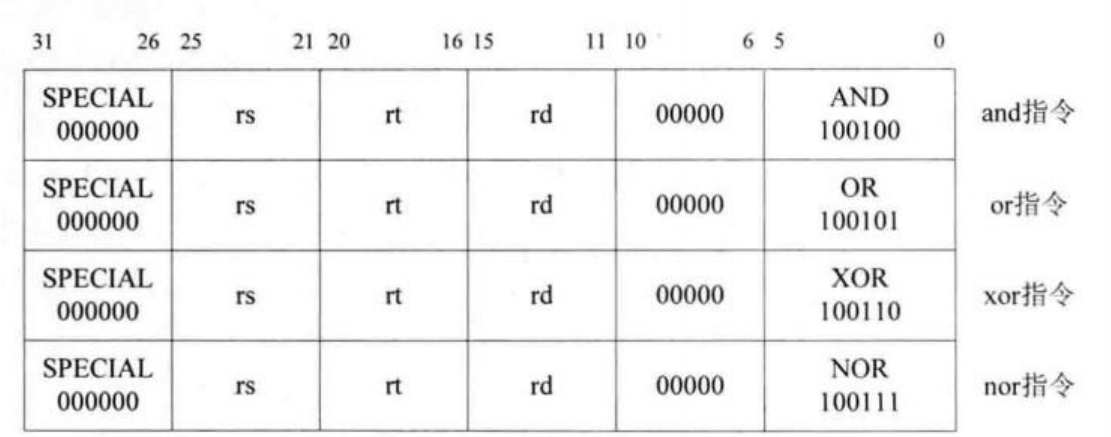


图 5-10　and、or、xor、nor 指令格式

2.2 andi、xori

指令格式：

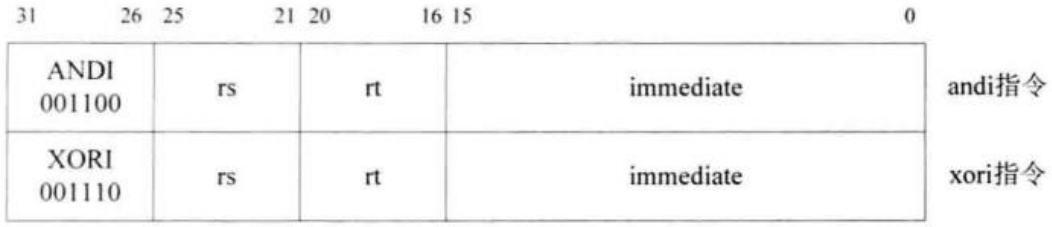


图 5-11　andi、xori 指令格式

2.3 lui 指令

指令格式：

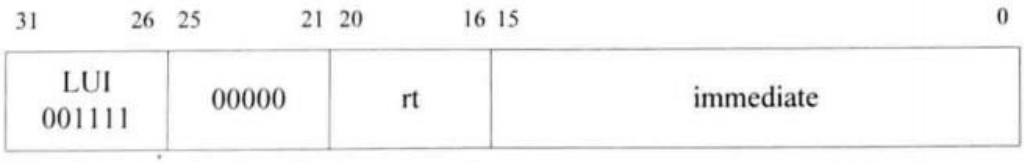


图 5-12　lui 指令格式

2.4 sll、sllv、sra、srav、srl、srlv

指令格式：

31 26	25 21	20 16	15 11	10 6	5 0	
SPECIAL 000000	00000	rt	rd	sa	SLL 000000	sll指令
SPECIAL 000000	00000	rt	rd	sa	SRL 000010	srl指令
SPECIAL 000000	00000	rt	rd	sa	SRA 000011	sra指令
SPECIAL 000000	rs	rt	rd	00000	SLLV 000100	sllv指令
SPECIAL 000000	rs	rt	rd	00000	SRLV 000110	srlv指令
SPECIAL 000000	rs	rt	rd	00000	SRAV 000111	srav指令

图 5-13　sll、srl、sra、sllv、srlv、srav 指令格式

2.5 nop、ssnop、sync、pref

31 26	25 21	20 16	15 11	10 6	5 0	
SPECIAL 000000	00000	00000	00000	00000	SLL 000000	nop指令
SPECIAL 000000	00000	00000	00000	00001	SLL 000000	ssnop指令
SPECIAL 000000	00000	00000	00000	00001	SYNC 001111	sync指令
PREF 110011	base	hint	offset			pref指令

图 5-14　nop、ssnop、sync、pref 指令的格式

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：

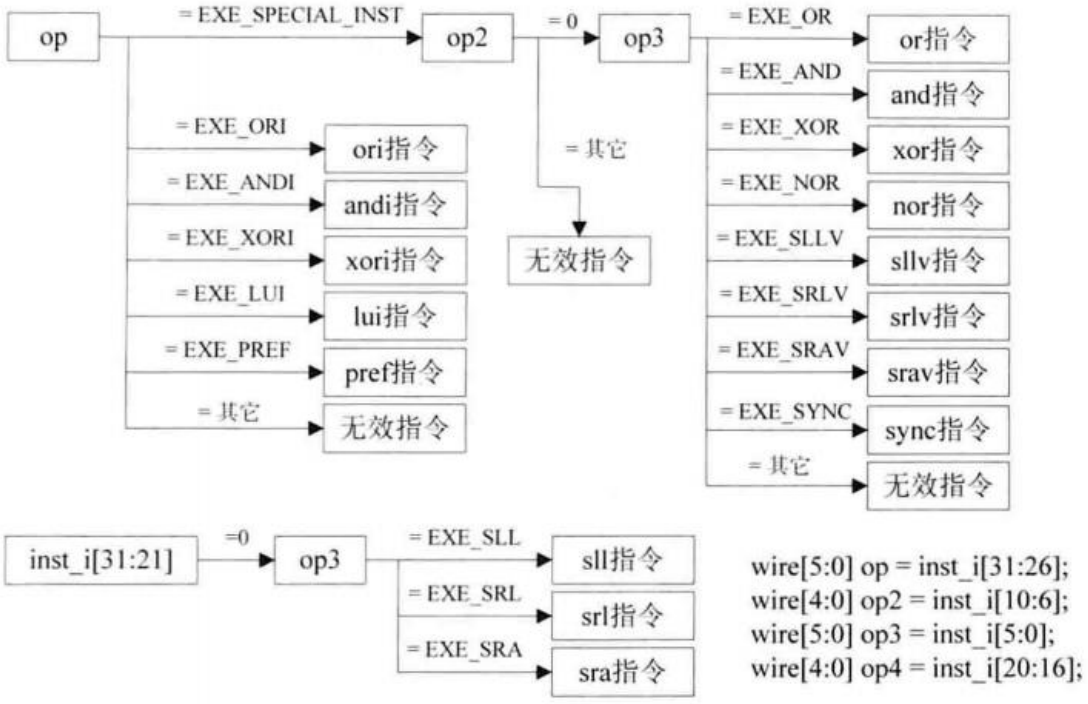


图 5-15　确定指令种类的过程

具体修改的 ID 模块代码见附录 1

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：res <= reg1 << reg2

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

$$shifters <= reg2\_i >> reg1\_i[4:0]) \mid (\{32\{reg2\_i[31]\}) << (6'b32 - \{1'b0, reg1\_i[4:0]\}))$$

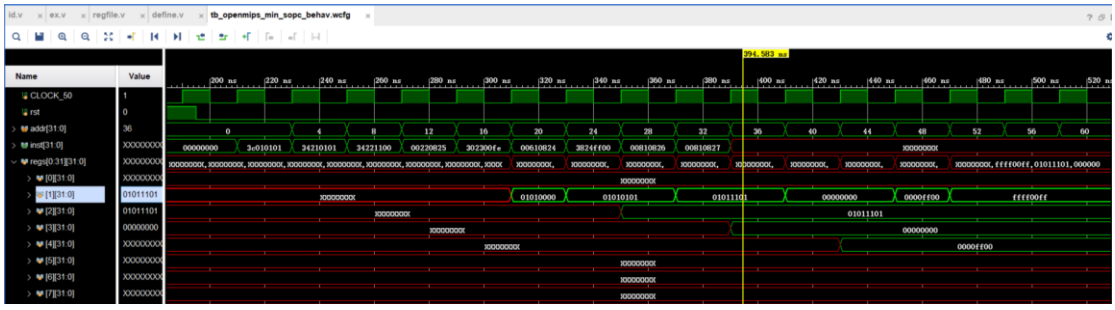
具体修改的 EX 模块代码见附录 2

三、 测试及分析

3.1 逻辑指令测试代码

```
.org 0x0
.global _start
.set noat
_start:
   lui  $1,0x0101
   ori  $1,$1,0x0101
   ori  $2,$1,0x1100          # $2 = $1 | 0x1100 = 0x01011101
   or   $1,$1,$2              # $1 = $1 | $2 = 0x01011101
   andi $3,$1,0x00fe          # $3 = $1 & 0x00fe = 0x00000000
   and  $1,$3,$1              # $1 = $3 & $1 = 0x00000000
   xori $4,$1,0xff00          # $4 = $1 ^ 0xff00 = 0x0000ff00
   xor  $1,$4,$1              # $1 = $4 ^ $1 = 0x0000ff00
   nor  $1,$4,$1              # $1 = $4 ~^ $1 = 0xffff00ff   nor is "not or"
```

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

```
    .org 0x0
    .set noat
    .global _start
_start:
    lui   $2,0x0404
    ori   $2,$2,0x0404
    ori   $7,$0,0x7
    ori   $5,$0,0x5
    ori   $8,$0,0x8
    sync
    sll   $2,$2,8     # $2 = 0x40404040 sll 8  = 0x04040400
    sllv  $2,$2,$7    # $2 = 0x04040400 sll 7  = 0x02020000
    srl   $2,$2,8     # $2 = 0x02020000 srl 8  = 0x00020200
    srlv  $2,$2,$5    # $2 = 0x00020200 srl 5  = 0x00001010
    nop
    sll   $2,$2,19    # $2 = 0x00001010 sll 19 = 0x80800000
    ssnop
    sra   $2,$2,16    # $2 = 0x80800000 sra 16 = 0xffff8080
    srav  $2,$2,$8    # $2 = 0xffff8080 sra 8  = 0xffffff80
```

仿真结果：

最终 2 号寄存器的值为 FFFFFF80，答案正确。

四、 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
……
```