**自己动手写第一条CPU指令ORI**

# ori指令说明

MIPS指令分为三种，分别是R类型，I类型和J类型，如表格所示：

表4-1 MIPS指令的类型

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R 型 | | | | | |
| Op | Rs | Rt | Rd | Sa | Func |
| 31---------26 | 25----------21 | 20----------16 | 15----------11 | 10-----------6 | 5-------------0 |
| I 型 | | | | | |
| Op | Rs | Rt | Imm | | |
| 31---------26 | 25----------21 | 20----------16 | 15------------------------------------0 | | |
| J型 | | | | | |
| Op | Addr | | | | |
| 31---------26 | 25----------------------------------0 | | | | |

## (无)符号拓展

有符号拓展:

0x8000 🡪 0xFFFF8000 0x1000 🡪 0x00001000

无符号拓展

0x8000 🡪 0x00008000 0x1000 🡪 0x00001000

## MIPS指令的32个通用寄存器



**详细说明**：

$0:即$zero,该寄存器总是返回零，为0这个有用常数提供了一个简洁的编码形式。

         move $t0,$t1实际为add $t0,$0,$t1  
       使用伪指令可以简化任务，汇编程序提供了比硬件更丰富的指令集。  
$1:即$at，该寄存器为汇编保留，由于I型指令的立即数字段只有16位，在加载大常数时，编译器或汇编程序需要把大常数拆开，然后重新组合到寄存器里。比如加载一个32位立即数需要 lui（装入高位立即数）和addi两条指令。像MIPS程序拆散和重装大常数由汇编程序来完成，汇编程序必需一个临时寄存器来重组大常数，这也是为汇编 保留$at的原因之一。

$2..$3:($v0-$v1)用于子程序的非浮点结果或返回值，对于子程序如何传递参数及如何返回，MIPS范围有一套约定，堆栈中少数几个位置处的内容装入CPU寄存器，其相应内存位置保留未做定义，当这两个寄存器不够存放返回值时，编译器通过内存来完成。

$4..$7:($a0-$a3)用来传递前四个参数给子程序，不够的用堆栈。a0-a3和v0-v1以及ra一起来支持子程序／过程调用，分别用以传递参数，返回结果和存放返回地址。当需要使用更多的寄存器时，就需要堆栈（stack)了,MIPS编译器总是为参数在堆栈中留有空间以防有参数需要存储。  
$8..$15:($t0-$t7)临时寄存器，子程序可以使用它们而不用保留。

$16..$23:($s0-$s7)保存寄存器，在过程调用过程中需要保留（被调用者保存和恢复，还包括$fp和$ra），MIPS提供了临时寄存器和保存寄存器，这样就减少了寄存器溢出（spilling,即将不常用的变量放到存储器的过程)，编译器在编译一个叶（leaf)过程（不调用其它过程的过程）的时候，总是在临时寄存器分配完了才使用需要保存的寄存器。

$24..$25:($t8-$t9)同($t0-$t7)

$26..$27:($k0,$k1)为操作系统／异常处理保留，至少要预留一个。 异常（或中断）是一种不需要在程序中显示调用的过程。MIPS有个叫异常程序计数器（exception program counter,EPC)的寄存器，属于CP0寄存器，用于保存造成异常的那条指令的地址。查看控制寄存器的唯一方法是把它复制到通用寄存器里，指令mfc0 (move from system control)可以将EPC中的地址复制到某个通用寄存器中，通过跳转语句（jr)，程序可以返回到造成异常的那条指令处继续执行。MIPS程序员都必须保留两个寄存器$k0和$k1，供操作系统使用。

       发生异常时，这两个寄存器的值不会被恢复，编译器也不使用k0和k1,异常处理函数可以将返回地址放到这两个中的任何一个，然后使用jr跳转到造成异常的指令处继续执行。

$28:($gp)为了简化静态数据的访问，MIPS软件保留了一个寄存器：全局指针gp(global pointer,$gp)，全局指针只想静态数据区中的运行时决定的地址，在存取位于gp值上下32KB范围内的数据时，只需要一条以gp为基指针的指令即可。在编译时，数据须在以gp为基指针的64KB范围内

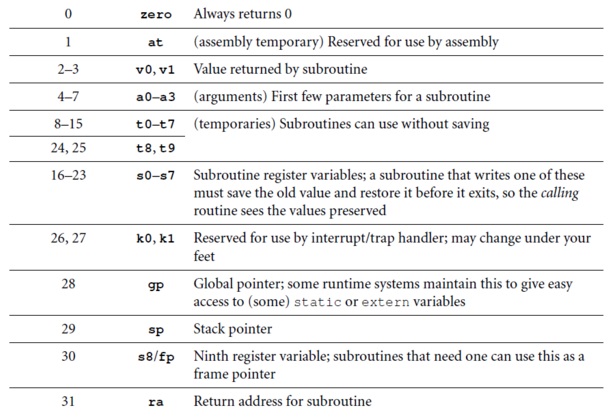
$29:($sp)MIPS硬件并不直接支持堆栈，你可以把它用于别的目的，但为了使用别人的程序或让别人使用你的程序， 还是要遵守这个约定的，但这和硬件没有关系。

$30:($fp)GNU MIPS C编译器使用了帧指针(frame pointer),而SGI的C编译器没有使用，而把这个寄存器当作保存寄存器使用（$s8),这节省了调用和返回开销，但增加了代码生成的复杂性。

$31:($ra)存放返回地址，MIPS有个jal(jump-and-link,跳转并链接)指令，在跳转到某个地址时，把下一条指令的地址放到$ra中。用于支持子程序，例如调用程序把参数放到$a0~$a3,然后jal X跳到X过程，被调过程完成后把结果放到$v0,$v1,然后使用jr $ra返回

**更多MIPS指令请参考：**[**MIPS通用寄存器+指令**](https://blog.csdn.net/gujing001/article/details/8476685)

一张比较官方的图：

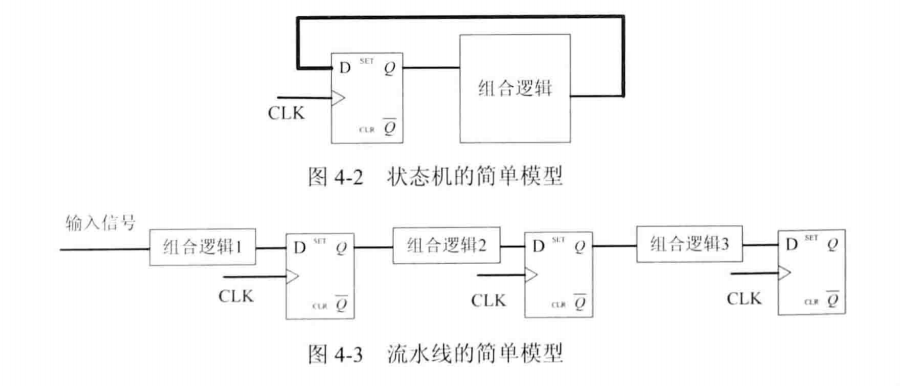


## 其他寄存器

其他包括PC（Programing Counter）程序计数器，类似于80x86汇编中的CS：IP，以及HI（高地址寄存器），LO（低地址寄存器），CPO协处理器

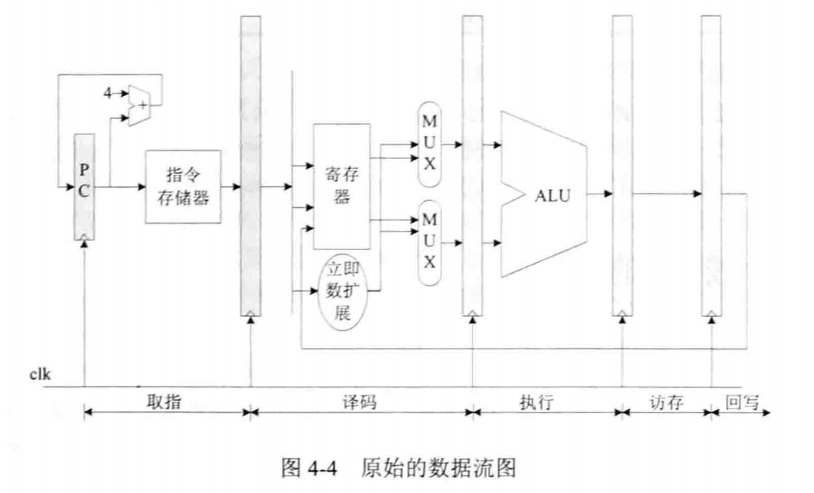
# 流水线的建立

## 2.1 流水线的简单模型

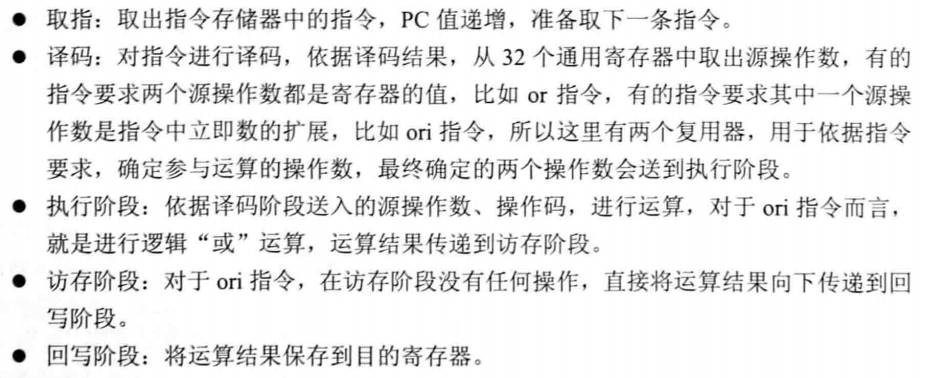


在流水线结构中，信号在寄存器之间传递，每传递到一级都会引起响应的组合逻辑电路变化，对这种模型进行抽象就是寄存器传输级（Register Transfer Level，RTL）

## 2.2原始的OpenMIPS五级流水线结构

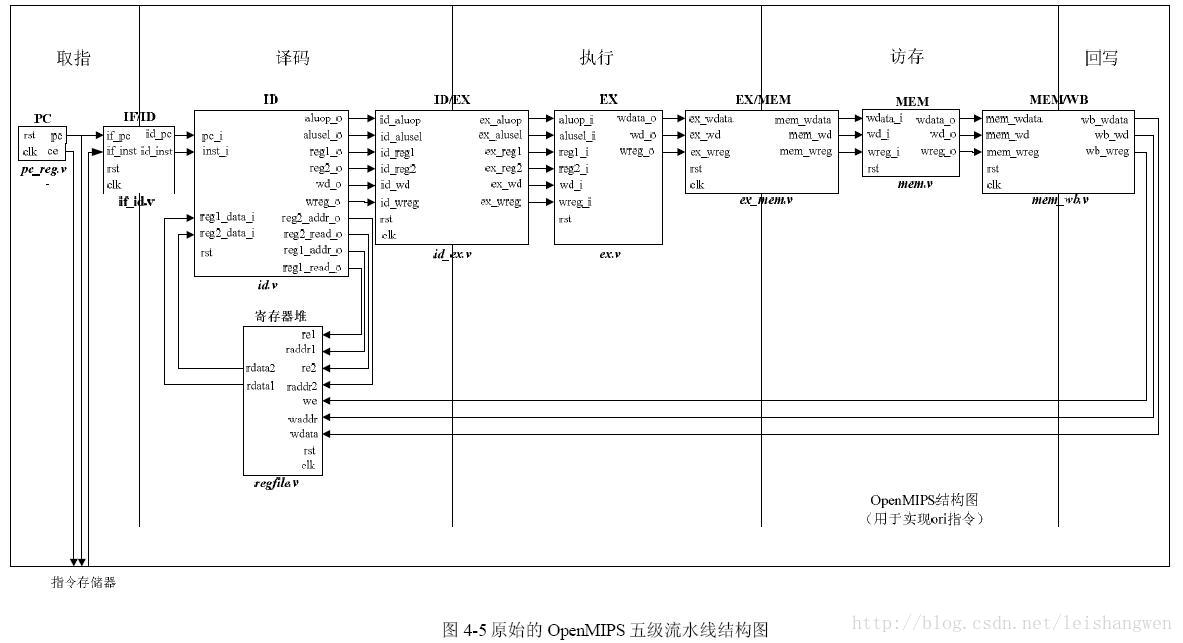


各个阶段完成的主要工作如下：



下面我们使用Verilog具体实现这些模块，分别为PC、IF/ID、ID、ID/EX、EX、EX/MEM、MEM、MEM/WB和WB

原始的OpenMIPS五级流水系统结构图如下所示：



## 2.3 一些宏定义

我们在define.v文件中定义一些常用的宏，方便区分每个变量的含义。语法和规则与C语言类似：

**Define.v：**

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Global macro definition\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

`define RstEnable 1'b1 //Reset signal valid

`define RstDisable 1'b0 //Reset signal invalid

`define ZeroWord 32'h0000\_0000 //Zero with 32bits

`define WriteEnable 1'b1 //WriteEnable

`define WriteDisable 1'b0 //WriteDisable

`define ReadEnable 1'b1 //ReadEnable

`define ReadDisable 1'b0 //ReadDiaable

`define AluOpBus 7:0 //Output the width of aluop\_o in the decoding stage

`define AluSelBus 2:0 //Output the width of alusel\_o in the decoding stage

`define InstValid 1'b0 //Instructure valid

`define InstInValid 1'b1 //Instructure invalid

`define True\_v 1'b1 //Logic True

`define False\_v 1'b0 //Logic False

`define ChipEnable 1'b1 //Chip Enable

`define ChipDisable 1'b0 //Chip Disable

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Macro definitions related to specific instructions\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

`define EXE\_ORI 6'b001101 //Instructure code of 'OR'

`define EXE\_NOP 6'b000000

//AluOp

`define EXE\_OR\_OP 8'b0010\_0101

`define EXE\_NOP\_OP 8'b0000\_0000

//AluSel

`define EXE\_RES\_LOGIC 3'b001

`define EXE\_RES\_NOP 3'b000

//\*\*\*\*\*\*\*\*\*\*\*\*\*Macro definition related to instruction memory ROM\*\*\*\*\*\*\*\*\*\*\*\*\*\*

`define InstAddrBus 31:0 //Addr Bus width of ROM

`define InstBus 31:0 //Data Bus width of ROM

`define InstMemNum 131071 //Real size is 128KB

`define InstMemNumLog2 17 //ROM Addr Bus Width in real

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Macro definition related to GPR RegFile\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

`define RegAddrBus 4:0 //Addr Bus width of RegFile

`define RegBus 31:0 //Data Bus Width of RegFile

`define RegWidth 32 //Width of GPR

`define DoubleRegWidth 64 //Double width of GPR

`define DoubleRegBus 63:0 //Double addr width of GPR

`define RegNum 32 //Amount of GPR

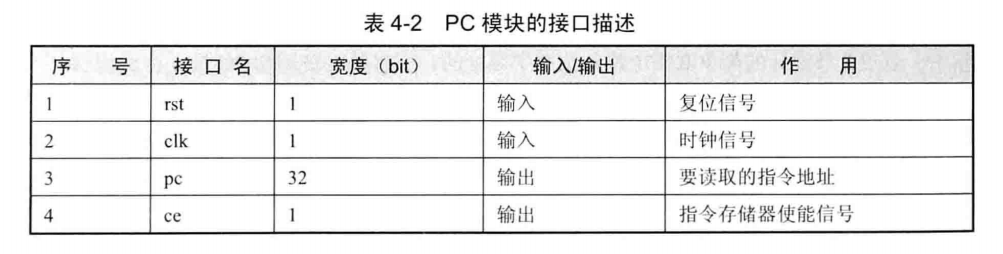
`define RegNumLog2 5 //Number of address bits used to address general purpose registers

`define NOPRegAddr 5'b00000

## 2.4取指阶段的实现

### 1. PC模块

该模块作用是给出指令地址



源文件**pc\_reg.v**

`include "define.v"

module pc\_reg(

input wire clk,

input wire rst,

output reg [`InstAddrBus] pc,

output reg ce

);

always @(posedge clk) begin

if (rst == `RstEnable) ce <= `ChipDisable; //Instruction memory is disabled when reseting

else ce <= `ChipEnable; //Instruction memory is enabled after resetting

end

always @(posedge clk) begin

if (ce == `ChipDisable) pc <= 32'h0000\_0000; //PC equals 0 when instruction memory is prohibited

else pc <= pc + 4'h4; //PC add 4 per clock period when instrution is enabled

end

endmodule

在复位的时候，指令存储器使能信号禁用，为ChipDisable，其余时间为ChipEnable，当指令存储器能使用时，PC的值会每个时钟周期加4，表示下一条指令的地址。因为一条指令是32位，对应4个字节。

### 2. IF/ID模块

该模块作用是暂时保存取指阶段取得的指令，以及对应的指令地址，并在下一个时钟传递到译码阶段，接口描述如表：



源代码**if\_id.v**

`include "define.v"

module if\_id(

input wire clk,

input wire rst,

//Signal from addressing stage. The macro definition of InstBus denote the width of instruction, is 32

input wire [`InstAddrBus] if\_pc,

input wire [`InstBus] if\_inst,

//Signal from decoding

output reg [`InstAddrBus] id\_pc,

output reg [`InstBus] id\_inst

);

always @(posedge clk) begin

if (rst == `RstEnable) begin

id\_pc <= `ZeroWord; //PC equals 0 when resetting

id\_inst <= `ZeroWord; //Instruction is also 0 when resetting, empty instruction actually

end else begin

id\_pc <= if\_pc; //Pass the value of addressing stage down

id\_inst <= if\_inst;

end

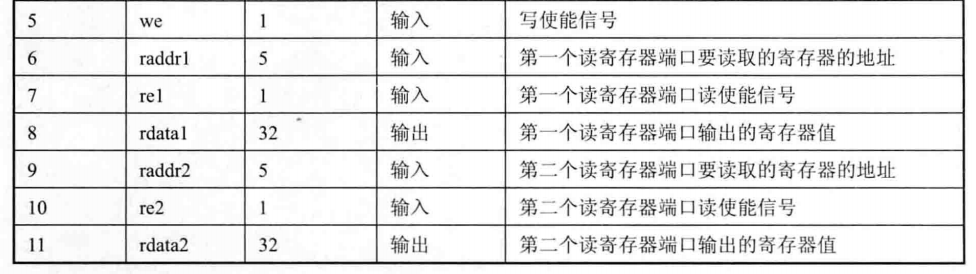
end

endmodule

## 2.5 译码阶段的实现

译码阶段包括Regfile、ID和ID/EX三个模块，将取到的指令进行译码：给猪要进行的运算类型，以及参与运算的操作数。

### 1. Regfile模块



源代码**regfile.v**

`include "define.v"

module regfile(

input wire clk,

input wire rst,

//port to write

input wire we,

input wire [`RegAddrBus] waddr,

input wire [`RegBus] wdata,

//port1 to read

input wire re1,

input wire [`RegAddrBus] raddr1,

output reg [`RegBus] rdata1,

//port2 to read

input wire re2,

input wire[`RegAddrBus] raddr2,

output reg [`RegBus] rdata2

);

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Chapter 1 : Define a Register with 32 bits \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

reg [`RegBus] regs [0:`RegNum - 1];

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Chapter 2 : Write Operation \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

always @(posedge clk) begin

if (rst == `RstDisable) begin

if ((we == `WriteEnable) && (waddr != `RegNumLog2'h0)) begin

regs[waddr] <= wdata;

end

end

end

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Chapter 3 : Read Operation of Reading Port1 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Chapter 4 : Read Operation of Reading Port2 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

always @(\*) begin

if (rst == `RstEnable) begin

rdata2 <= `ZeroWord;

end else if (raddr2 == `RegNumLog2'h0) begin

rdata2 <= `ZeroWord;

end else if ((rdata2 == waddr) && (we == `WriteEnable) && (re2 == `ReadEnable)) begin

rdata2 <= wdata;

end else if (re2 == `ReadEnable) begin

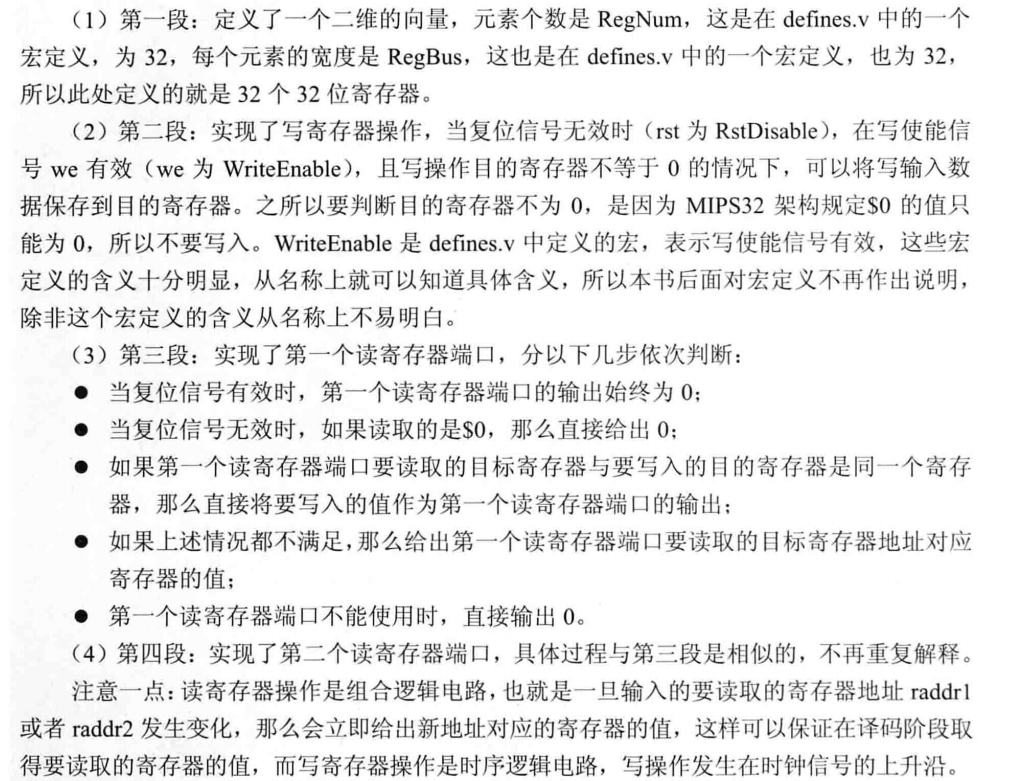
rdata2 <= regs[raddr2];

end else begin

rdata2 <= `ZeroWord;

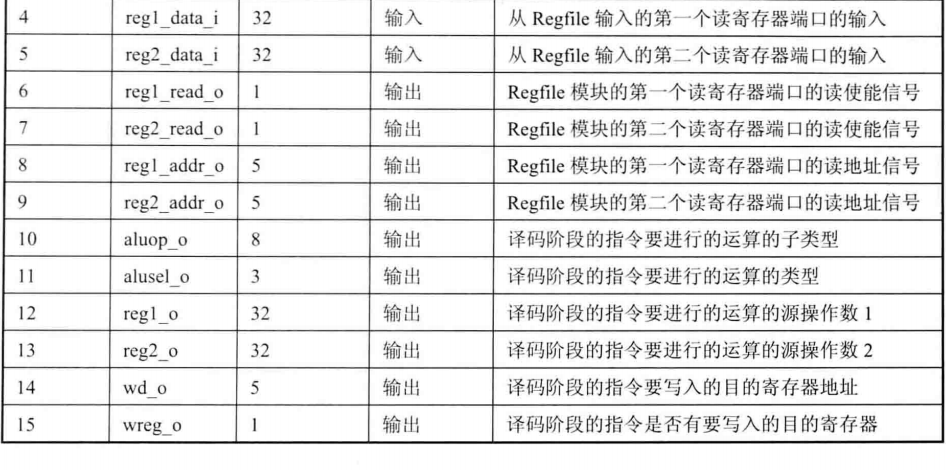
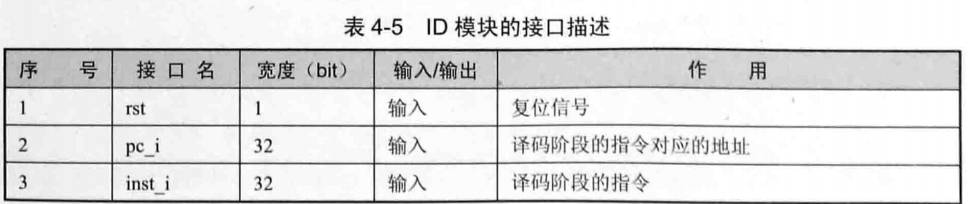
end

end

endmodule

### 2. ID模块

该模块的的作用是对指令进行译码， 得到最终运算的雷子那个、子类型、源操作数1、源操作数2、要写入的目的寄存器地址等信息，其中运算类型指的是**逻辑运算、移位运算、算数运算**等，子类型指的是更加详细的运算类型，ID模块的接口表述如表：



源代码**id.v**

`include "define.v"

module id(

input wire rst,

input wire [`InstAddrBus] pc\_i,

input wire [`InstBus] inst\_i,

//Read the value of Regfile

input wire [`RegBus] reg1\_data\_i,

input wire [`RegBus] reg2\_data\_i,

//Message output to the Regfile

output reg reg1\_read\_o,

output reg reg2\_read\_o,

output reg [`RegAddrBus] reg1\_addr\_o,

output reg [`RegAddrBus] reg2\_addr\_o,

//Information sent to the execution stage

output reg [`AluOpBus] aluop\_o,

output reg [`AluSelBus] alusel\_o,

output reg [`RegBus] reg1\_o,

output reg [`RegBus] reg2\_o,

output reg [`RegAddrBus] wd\_o,

output reg wreg\_o

);

//Fetch the instruction code and function code

//For instruction ORI, you will know wheather it is the ORI instruction only to judge 26th-31st bits

wire [5:0] op = inst\_i[31:26];

wire [4:0] op2 = inst\_i[10:6];

wire [5:0] op3 = inst\_i[5:0];

wire [4:0] op4 = inst\_i[20:16];

//The immediate number to save the instruction

reg [`RegBus] imm;

//Instruct wheather the instruction is valid

reg instvalid;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Chapter 1 : Decoding the Instruction \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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\_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

default: begin

end

endcase //case op

end //if

end //always

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Chapter 2 : Confirm the calculating source Operand \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

always @(\*) begin

if (rst == `RstEnable) begin

reg1\_o <= `ZeroWord;

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

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Chapter 3 : Confirm the calculating source Operand \*\*\*\*\*\*\*\*\*\*\*\*\*\*

always @(\*) begin

if (rst == `RstEnable) begin

reg2\_o <= `ZeroWord;

end else if (reg2\_read\_o == 1'b1) begin

reg2\_o <= reg2\_data\_i; //Regfile read the value of port1

end else if (reg2\_read\_o == 1'b0) begin

reg2\_o <= imm; //immediate number

end else begin

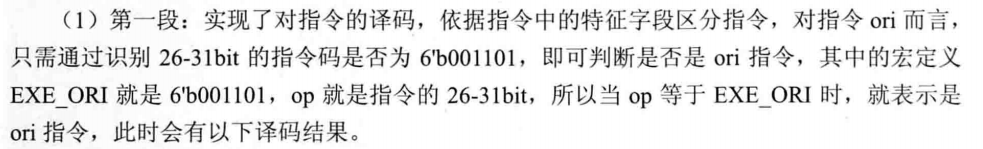
reg2\_o <= `ZeroWord;

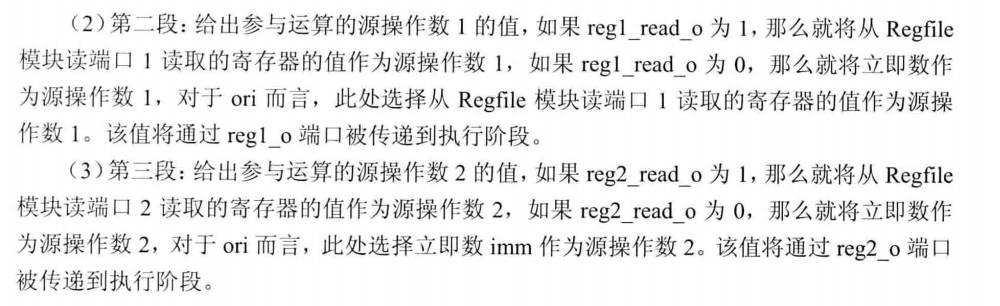
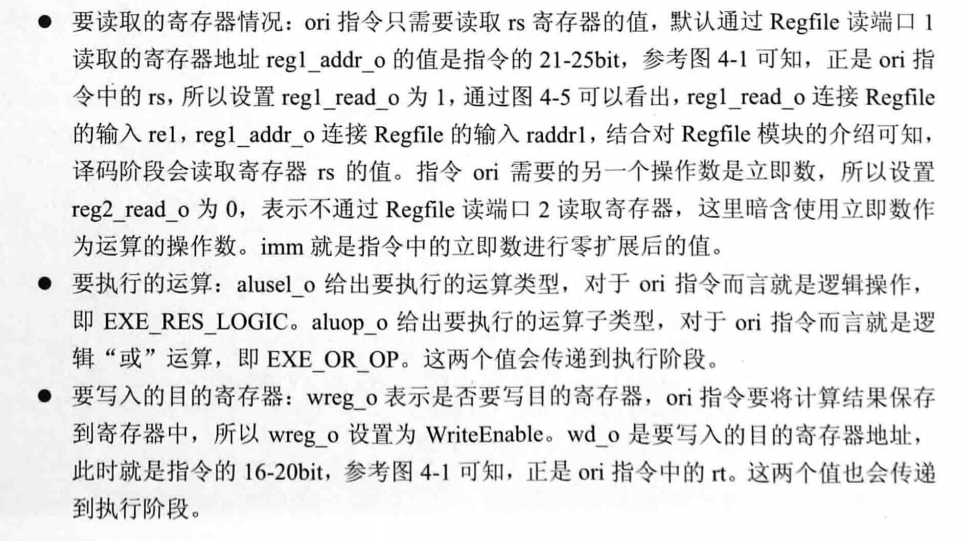
end

end

endmodule

该模块中的电路都是组合逻辑电路，代码可以分为三段理解：





### 3. ID/EX模块

ID模块的输出连接到ID/EX模块，该模块作用是将译码阶段所取得的运算类型、源操作数、要写的目的寄存器地址等结果，在下一个市政传递到流水线执行阶段，接口描述如表



源代码**id\_ex.v**

`include "define.v"

module id\_ex(

input wire clk,

input wire rst,

//Message sent from encoding stage

input wire [`AluOpBus] id\_aluop,

input wire [`AluSelBus] id\_alusel,

input wire [`RegBus] id\_reg1,

input wire [`RegBus] id\_reg2,

input wire [`RegAddrBus] id\_wd,

input wire id\_wreg,

//Message send to execuation stage

output reg [`AluOpBus] ex\_aluop,

output reg [`AluSelBus] ex\_alusel,

output reg [`RegBus] ex\_reg1,

output reg [`RegBus] ex\_reg2,

output reg [`RegAddrBus] ex\_wd,

output reg ex\_wreg

);

always @ (posedge clk) begin

if (rst == `RstEnable) begin

ex\_aluop <= `EXE\_NOP\_OP;

ex\_alusel <= `EXE\_RES\_NOP;

ex\_reg1 <= `ZeroWord;

ex\_reg2 <= `ZeroWord;

ex\_wd <= `NOPRegAddr;

ex\_wreg <= `WriteDisable;

end else begin

ex\_aluop <= id\_aluop;

ex\_alusel <= id\_alusel;

ex\_reg1 <= id\_reg1;

ex\_reg2 <= id\_reg2;

ex\_wd <= id\_wd;

ex\_wreg <= id\_wreg;

end

end

endmodule

该模块仅仅是将译码阶段的结果在时钟上升沿传递到执行阶段，没什么好说的。

## 2.6 执行阶段的实现

现在进入流水线的执行阶段，在此阶段将依据译码阶段的结果，对源操作数1，源操作数2进行制定的运算。执行阶段包括EX、EX/MEM两个模块。

### EX模块

EX模块会从ID/EX模块中获取到运算类型alusel\_i、运算子类型aluop\_i、源操作数reg1\_i、源操作数reg2\_i，要写的目的寄存器地址wd\_i，在EX模块内进行运算，接口表述如表：



源代码**ex.v**

`include"define.v"

module ex(

input wire rst,

input wire [`AluOpBus] aluop\_i,

input wire [`AluSelBus] alusel\_i,

input wire [`RegBus] reg1\_i,

input wire [`RegBus] reg2\_i,

input wire [`RegAddrBus] wd\_i,

input wire wreg\_i,

output reg [`RegBus] wdata\_o,

output reg [`RegAddrBus] wd\_o,

output reg wreg\_o

);

//Save the result of logic calculation

reg [`RegBus] logicout;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Chapter 1 : Calculating by the value of 'aluop\_i' \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

always @(\*) begin

if (rst == `RstEnable) begin

logicout <= `ZeroWord;

end else begin

case (aluop\_i)

`EXE\_OR\_OP: begin

logicout <= reg1\_i | reg2\_i;

end

default: begin

end

endcase

end //if end

end //always 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

default: begin

end

endcase

end

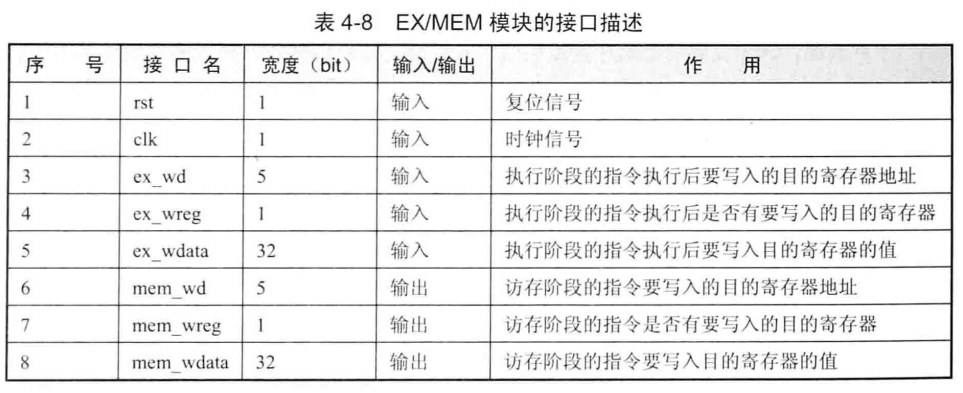
endmodule

该模块都是组合逻辑电路，分为两段理解：

1. 第一段：依据输入的运算子类型进行运算，这里只有一种——逻辑或运算。运算结果保存在logicout中，这是专门保存逻辑操作的结果，以后添加其他指令时还会添加其他变量。
2. 第二段：给出最终的运算结果，包括收要写入目的寄存器wreg\_o、要写的目的寄存器地址wd\_o、要写入的数据wdata\_o，但是这些直接来自译码阶段，不需要改变。wdata\_o的值要依据运算类型进行选择，如果是逻辑运算，那么将logicout的值直接赋给wdata\_o。此处实际是为将来拓展做准备，当添加其他类型的指令时，只需要修改这里的case语句即可。

### EX/MEM模块

该模块的作用是将执行阶段取得的结果，在下一个时钟周期后传递到流水线访存阶段，其接口描述如表：



源代码**ex\_mem.v**

`include "define.v"

module ex\_mem(

input wire clk,

input wire rst,

input wire [`RegBus] ex\_wdata,

input wire [`RegAddrBus] ex\_wd,

input wire ex\_wreg,

output reg [`RegBus] mem\_wdata,

output reg [`RegAddrBus] mem\_wd,

output reg mem\_wreg

);

always @(posedge clk) begin

if (rst == `RstEnable) begin

mem\_wdata <= `ZeroWord;

mem\_wd <= `NOPRegAddr;

mem\_wreg <= `WriteDisable;

end begin

mem\_wdata <= ex\_wdata;

mem\_wd <= ex\_wd;

mem\_wreg <= ex\_wreg;

end

end

endmodule

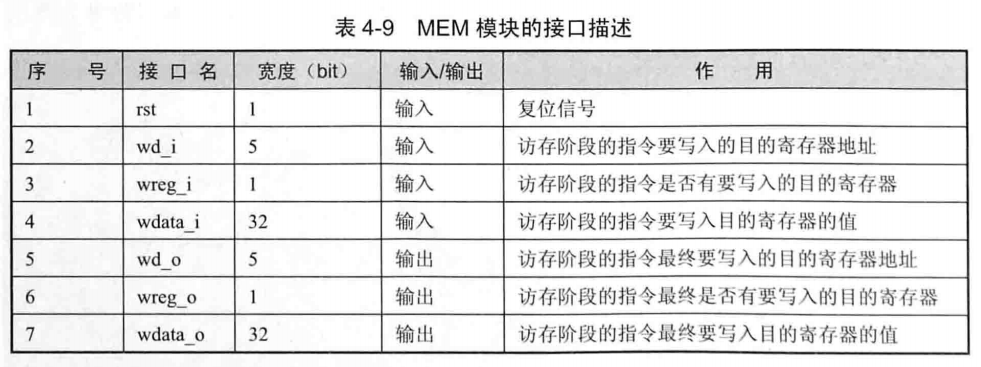
时序逻辑电路，很简单，不说了。

## 访存阶段的实现

现在ori指令进入访存阶段了，但是由于ori指令不需要访问数据存储器，所以在访存阶段，不做任何事，只是简单地将执行阶段的结果向回写阶段传递即可。

### MEM模块

接口描述如表：



源代码**mem.v**

**`include "define.v"**

module mem(

input wire rst,

input wire [`RegAddrBus] wd\_i,

input wire [`RegBus] wdata\_i,

input wire wreg\_i,

output reg [`RegAddrBus] wd\_o,

output reg [`RegBus] wdata\_o,

output reg wreg\_o

);

always @(\*) begin

if (rst == `RstEnable) begin

wd\_o <= `NOPRegAddr;

wreg\_o <= `WriteDisable;

wdata\_o <= `ZeroWord;

end else begin

wd\_o <= wd\_i;

wreg\_o <= wreg\_i;

wdata\_o <= wdata\_i;

end

end

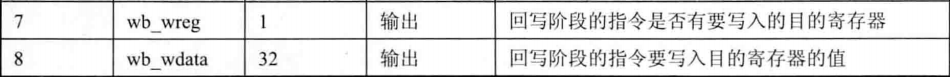
endmodule

组合逻辑电路，很简单，不说了。

### MEM/WB模块

该模块作用是将访存阶段的运算结果传递到回写模块，接口描述如表：





源代码**mem\_wb.v**

module mem\_wb(

input wire clk,

input wire rst,

input wire [`RegAddrBus] mem\_wd,

input wire [`RegBus] mem\_wdata,

input wire mem\_wreg,

output reg [`RegAddrBus] wb\_wd,

output reg [`RegBus] wb\_wdata,

output reg wb\_wreg

);

always @(posedge clk) begin

if (rst == `RstEnable) begin

wb\_wd <= `NOPRegAddr;

wb\_wdata <= `ZeroWord;

wb\_wreg <= `WriteDisable;

end else begin

wb\_wd <= mem\_wd;

wb\_wdata <= mem\_wdata;

wb\_wreg <= mem\_wreg;

end

end

endmodule

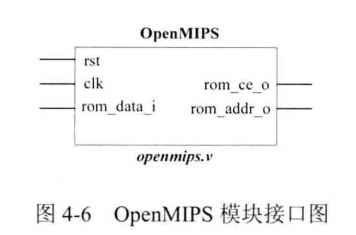
时序逻辑电路…

## 回写阶段的实现

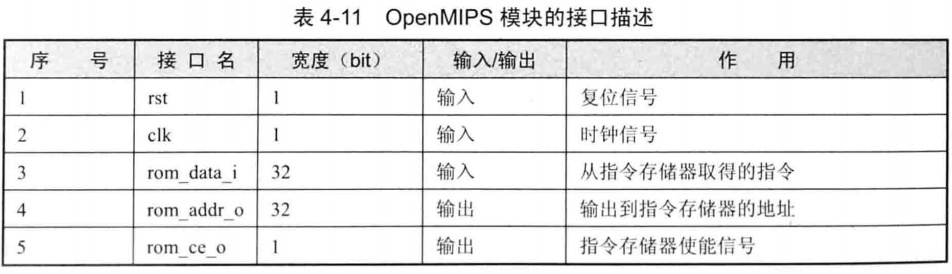
经过上面的传递，ori指令已经进入回写阶段了，这个阶段实际上是在Regfile模块中实现的，具体代码参考Regfile模块。

## 顶层模块OpenMIPS的实现

该模块在openmips.v中实现，主要是对各个阶段的模块进行例化、连接，关系如图：



其接口描述如表：



### 源代码openmips.v

`include "define.v"

module openmips(

input wire clk,

input wire rst,

input wire [`RegBus] rom\_data\_i,

output wire [`RegBus] rom\_addr\_o,

output wire rom\_ce\_o

);

//Variable of connecting IF/ID module and ID module in the encoding stage

wire [`InstAddrBus] pc;

wire [`InstAddrBus] id\_pc\_i;

wire [`InstBus] id\_inst\_i;

//Variable of connecting output of ID mudule and input of ID/EX module

wire [`AluOpBus] id\_aluop\_o;

wire [`AluSelBus] id\_alusel\_o;

wire [`RegBus] id\_reg1\_o;

wire [`RegBus] id\_reg2\_o;

wire [`RegAddrBus] id\_wd\_o;

wire id\_wreg\_o;

//Variable of connecting output of ID/EX module and input of EX module

wire [`AluOpBus] ex\_aluop\_i;

wire [`AluSelBus] ex\_alusel\_i;

wire [`RegBus] ex\_reg1\_i;

wire [`RegBus] ex\_reg2\_i;

wire [`RegAddrBus] ex\_wd\_i;

wire ex\_wreg\_i;

//Variable of connecting output of EX module and input of EX/MEM module

wire ex\_wreg\_o;

wire [`RegAddrBus] ex\_wd\_o;

wire [`RegBus] ex\_wdata\_o;

//Variable of connecting output of EX/MEM module and input of MEM module

wire mem\_wreg\_i;

wire [`RegAddrBus] mem\_wd\_i;

wire [`RegBus] mem\_wdata\_i;

//Variable of connecting output of MEM and input of MEM/WB module

wire mem\_wreg\_o;

wire [`RegBus] mem\_wdata\_o;

wire [`RegAddrBus] mem\_wd\_o;

//Variable of connecting output of MEM/WB and input of WB module

wire wb\_wreg\_i;

wire [`RegAddrBus] wb\_wd\_i;

wire [`RegBus] wb\_wdata\_i;

//Variable of connecting ID module in the encoding stage and Regfile module

wire reg1\_read;

wire reg2\_read;

wire [`RegBus] reg1\_data;

wire [`RegBus] reg2\_data;

wire [`RegAddrBus] reg1\_addr;

wire [`RegAddrBus] reg2\_addr;

//pc\_reg for instance

pc\_reg pc\_reg0(

.clk(clk),

.rst(rst),

.pc(pc),

.ce(rom\_ce\_o)

);

assign rom\_addr\_o = pc; //The input address of instruction register is pc's value

//IF/ID module for instance

if\_id if\_id0(

.clk(clk),

.rst(rst),

.if\_pc(pc),

.if\_inst(rom\_data\_i),

.id\_pc(id\_pc\_i),

.id\_inst(id\_inst\_i)

);

//ID module in the encoding stage for instance

id id0 (

.rst(rst),

.pc\_i(id\_pc\_i),

.inst\_i(id\_inst\_i),

//input from Regfile module

.reg1\_data\_i(reg1\_data),

.reg2\_data\_i(reg2\_data),

//Message sent to RegFile module

.reg1\_read\_o(reg1\_read),

.reg2\_read\_o(reg2\_read),

.reg1\_addr\_o(reg1\_addr),

.reg2\_addr\_o(reg2\_addr),

//Message sent to ID/EX module

.aluop\_o(id\_aluop\_o),

.alusel\_o(id\_alusel\_o),

.reg1\_o(id\_reg1\_o),

.reg2\_o(id\_reg2\_o),

.wd\_o(id\_wd\_o),

.wreg\_o(id\_wreg\_o)

);

//GPR RegFile module for instance

regfile regfile1 (

.clk(clk),

.rst(rst),

.we(wb\_wreg\_i),

.waddr(wb\_wd\_i),

.wdata(wb\_wdata\_i),

.re1(reg1\_read),

.raddr1(reg1\_addr),

.rdata1(reg1\_data),

.re2(reg2\_read),

.raddr2(reg2\_addr),

.rdata2(reg2\_data)

);

//ID/EX module for instance

id\_ex id\_ex0(

.clk(clk),

.rst(rst),

//Message sent from ID module in the encoding stage

.id\_aluop(id\_aluop\_o),

.id\_alusel(id\_alusel\_o),

.id\_reg1(id\_reg1\_o),

.id\_reg2(id\_reg2\_o),

.id\_wd(id\_wd\_o),

.id\_wreg(id\_wreg\_o),

//Message send to EX module in the execution stage

.ex\_aluop(ex\_aluop\_i),

.ex\_alusel(ex\_alusel\_i),

.ex\_reg1(ex\_reg1\_i),

.ex\_reg2(ex\_reg2\_i),

.ex\_wd(ex\_wd\_i),

.ex\_wreg(ex\_wreg\_i)

);

//EX module for instance

ex ex0(

.rst(rst),

//Message sent from ID/EX module

.aluop\_i(ex\_aluop\_i),

.alusel\_i(ex\_alusel\_i),

.reg1\_i(ex\_reg1\_i),

.reg2\_i(ex\_reg2\_i),

.wd\_i(ex\_wd\_i),

.wreg\_i(ex\_wreg\_i),

//Message output to EX module

.wd\_o(ex\_wd\_o),

.wreg\_o(ex\_wreg\_o),

.wdata\_o(ex\_wdata\_o)

);

//EX/MEM module for instance

ex\_mem ex\_mem0(

.clk(clk),

.rst(rst),

//Message sent from EX module

.ex\_wd(ex\_wd\_o),

.ex\_wreg(ex\_wreg\_o),

.ex\_wdata(ex\_wdata\_o),

//Message sent to MEM module

.mem\_wd(mem\_wd\_i),

.mem\_wreg(mem\_wreg\_i),

.mem\_wdata(mem\_wdata\_i)

);

//MEM module for instance

mem mem0(

.rst(rst),

//Message sent from EX/MEM module

.wd\_i(mem\_wd\_i),

.wreg\_i(mem\_wreg\_i),

.wdata\_i(mem\_wdata\_i),

//Message sent to MEM/WB module

.wd\_o(mem\_wd\_o),

.wreg\_o(mem\_wreg\_o),

.wdata\_o(mem\_wdata\_o)

);

//MEM/WB module for instance

mem\_wb mem\_wb0(

.clk(clk),

.rst(rst),

//Message sent from MEM module

.mem\_wd(mem\_wd\_o),

.mem\_wreg(mem\_wreg\_o),

.mem\_wdata(mem\_wdata\_o),

//Message sent to WB(WriteBack) module

.wb\_wd(wb\_wd\_i),

.wb\_wreg(wb\_wreg\_i),

.wb\_wdata(wb\_wdata\_i)

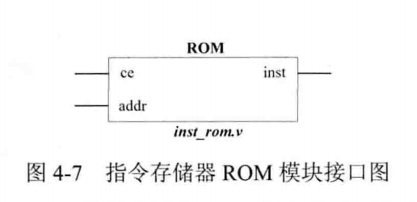
);

endmodule

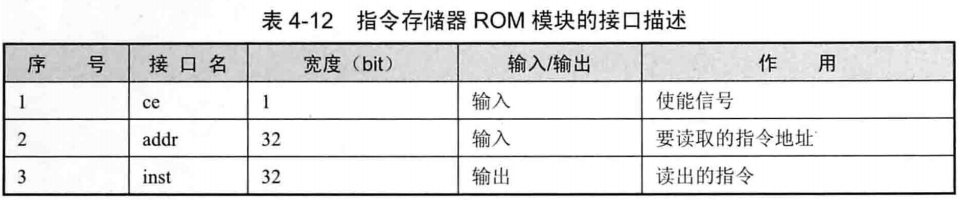
# 验证OpenMIPS实现效果

## 3.1 指令存储器ROM的实现

模块图如图：



接口描述如表：



源代码**inst\_rom.v**

`include "define.v"

module inst\_rom(

input wire ce,

input wire [`InstAddrBus] addr,

output reg [`InstBus] inst

);

//Define an array with size of InstMemNun and with element width of InstBus

reg [`InstBus] inst\_mem [0 : `InstMemNum - 1];

//Use the file inst\_rom.data to initialize the instruction rom

initial $readmemh ("inst\_rom.data", inst\_mem);

//When the resetting signal is invalid, giving the element of ROM by the address typed in

always @(\*) begin

if (ce == `ChipDisable) begin

inst <= `ZeroWord;

end else begin

inst <= inst\_mem[addr[`InstMemNumLog2 + 1 : 2]];

end

end

endmodule

这里要详细解释一下，在我学习的过程中，这里是相对来说比较难理解的，基于仿真结果，我将进行解释。

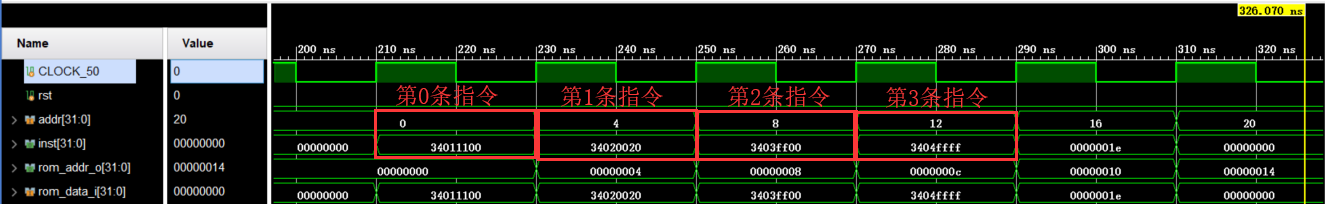
首先使用了initial语句，该语句执行过程中只会执行一次，，通常用于仿真模块中对激励向量的描述，或用于给变量赋初值，是面向模拟仿真的过程语句，通常被综合工具支持，如果要将用该方法实现的OpenMIPS处理器使用综合工具进行综合，要修改这里的初始化指令存储器的方法。

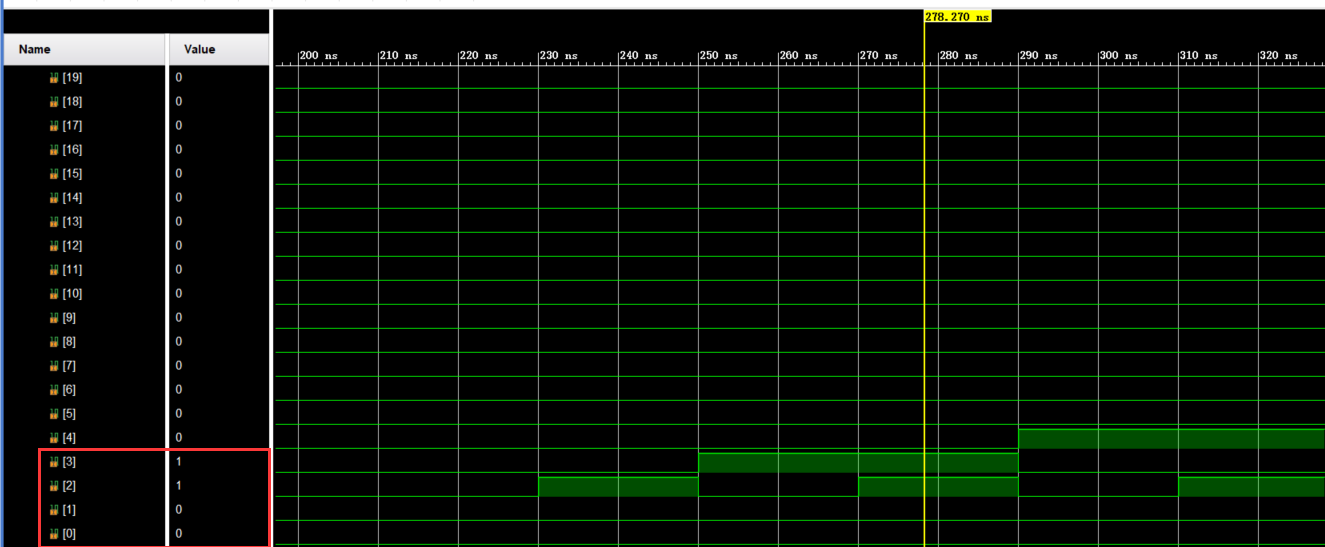
然后对于系统函数$readmemh，表示从inst\_rom.data问价那种读取数据有以初始化isnt\_mem，inst\_mem是定义的一个大小为128KB（131072字节）的数组，每个数组的元素是一个32位的数据，用常量InstAddrBus表示，因为在MIPS指令集中，每一条指令的大小是固定的4字节（32bit），相当于每一个数组元素表示一条指令，可以存放131072条指令，该大小也就是ROM的实际大小。

对于这一行代码，一开始确实有些难理解，让我娓娓道来：

首先我们要知道常量InstMemNumLog2是什么意思，这表示的是地址线的宽度，也就是决定寄存器大小的的那个大小。我们知道OpenMIPS是按照字节寻址的，在此处定义的指令存储器的每个地址是一个32bit的字，我们要将OpenMIPS给出的指令地址除以4使用，也就是说inst\_mem存储的是指令的条数，而addr表示的指令的起始地址，因此从addr的第二位（从第0位开始计数）开始读取，读到地址线宽度+1位停止。

下面用仿真来解释：





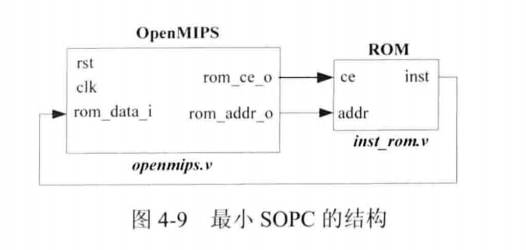
我们看到当addr的值实际上是1100时，十进制为12，是第三条指令，因此从addr的第二位开始读，为11，十进制就是3。

是不是很简单



## 3.2 最小SOPC的实现

SOPC即（System on a Programing Chip，可编程片上系统），其中仅包括OpenMIPS、指令 存储器ROM，所以是一个最小SOPC。OpenMIPS从指令存储器中读取指令，指令进入OpenMIPS开始执行，结构如图：



对应的模块为openmips\_min\_sopc，代码如下：

`include "define.v"

module openmips\_min\_sopc(

input wire clk,

input wire rst

);

//Connecting the instruction register

wire [`InstAddrBus] inst\_addr;

wire [`InstBus] inst;

wire rom\_ce;

//OpenMIPS for instance

openmips openmips0(

.clk(clk),

.rst(rst),

.rom\_addr\_o(inst\_addr),

.rom\_data\_i(inst),

.rom\_ce\_o(rom\_ce)

);

//ROM for instance

inst\_rom inst\_rom0(

.ce(rom\_ce),

.addr(inst\_addr),

.inst(inst)

);

Endmodule

## 3.3 测试程序的编写、编译和仿真

我们现在写一段测试程序，将其存储到指令存储器ROM中，当SOPC开始执行时，机会从ROM中读取我们的程序，送入OpenMIPS处理器中执行。我们现在只有ORI一条指令。

### 测试程序

测试程序inst\_rom.S：

.org 0x0 # 指示程序从地址0x0开始

.global \_start # 定义一个全局符号—\_start

.set noat # 允许自由使用寄存器$1

\_start:

ori $1 ,$0,0x1100 # $1 = $0 | 0x1100 = 0x1100

ori $2 ,$0,0x0020 # $2 = $0 | 0x0020 = 0x0020

ori $3 ,$0,0xff00 # $3 = $0 | 0xff00 = 0xff00

ori $4 ,$0,0xffff # $4 = $0 | 0xffff = 0xffff

ori $5 ,$4,0x0010

ori $6 ,$0,0xaaaa

ori $7 ,$0,0xaabb

ori $8 ,$1,0x0011

ori $9 ,$0,0xcccc

ori $10,$0,0xdddd

ori $11,$0,0xeeee

ori $12,$0,0x2222

ori $13,$0,0x3333

ori $14,$0,0x9876

ori $15,$0,0xabcd

ori $16,$0,0xfedc

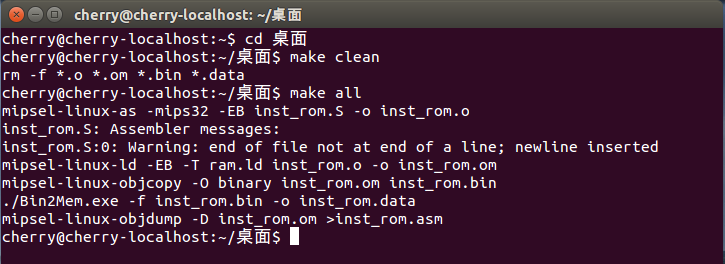
ori $17,$1,0xee00

ori $18,$0,0xffee

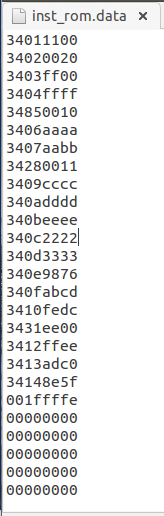
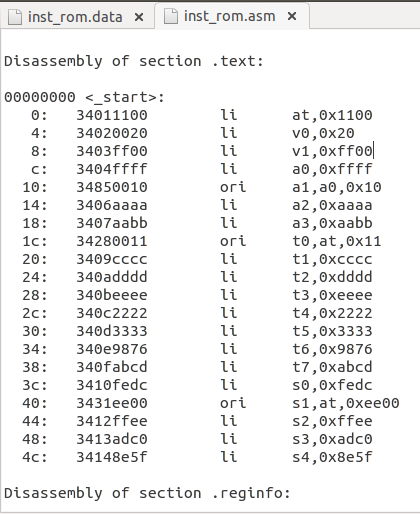
ori $19,$0,0xadc0

ori $20,$0,0x8e5f

### 用GCC编译

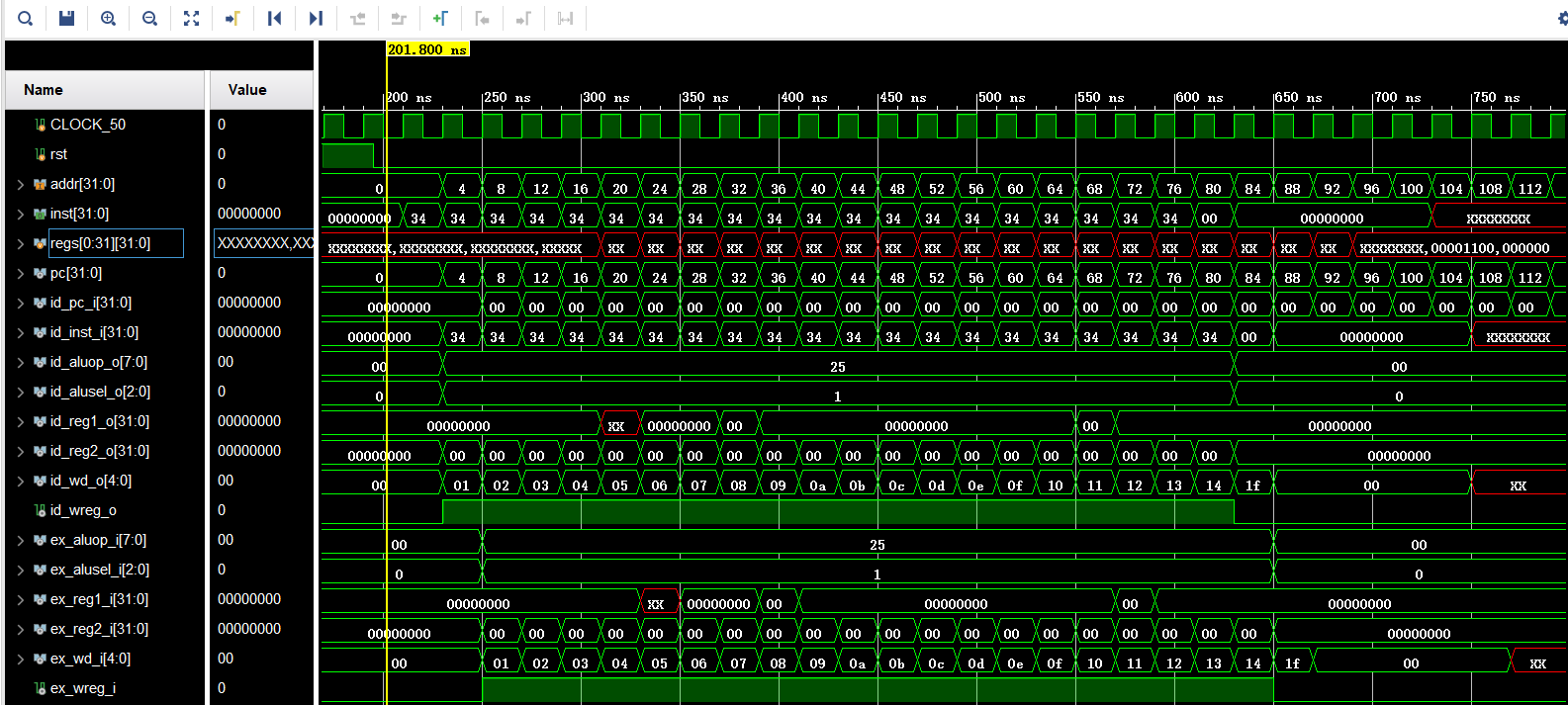


查看生成的asm汇编文件和机器码：

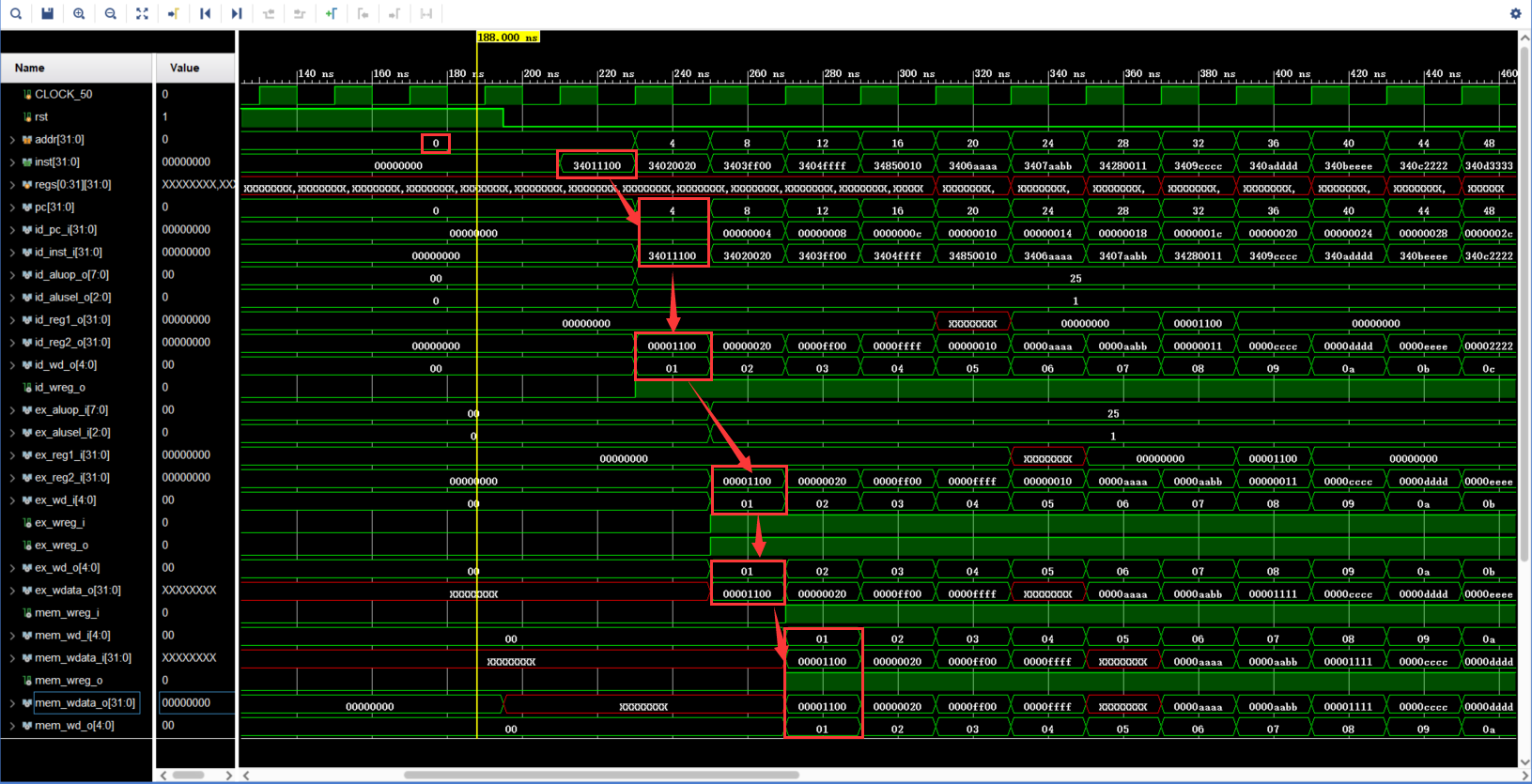


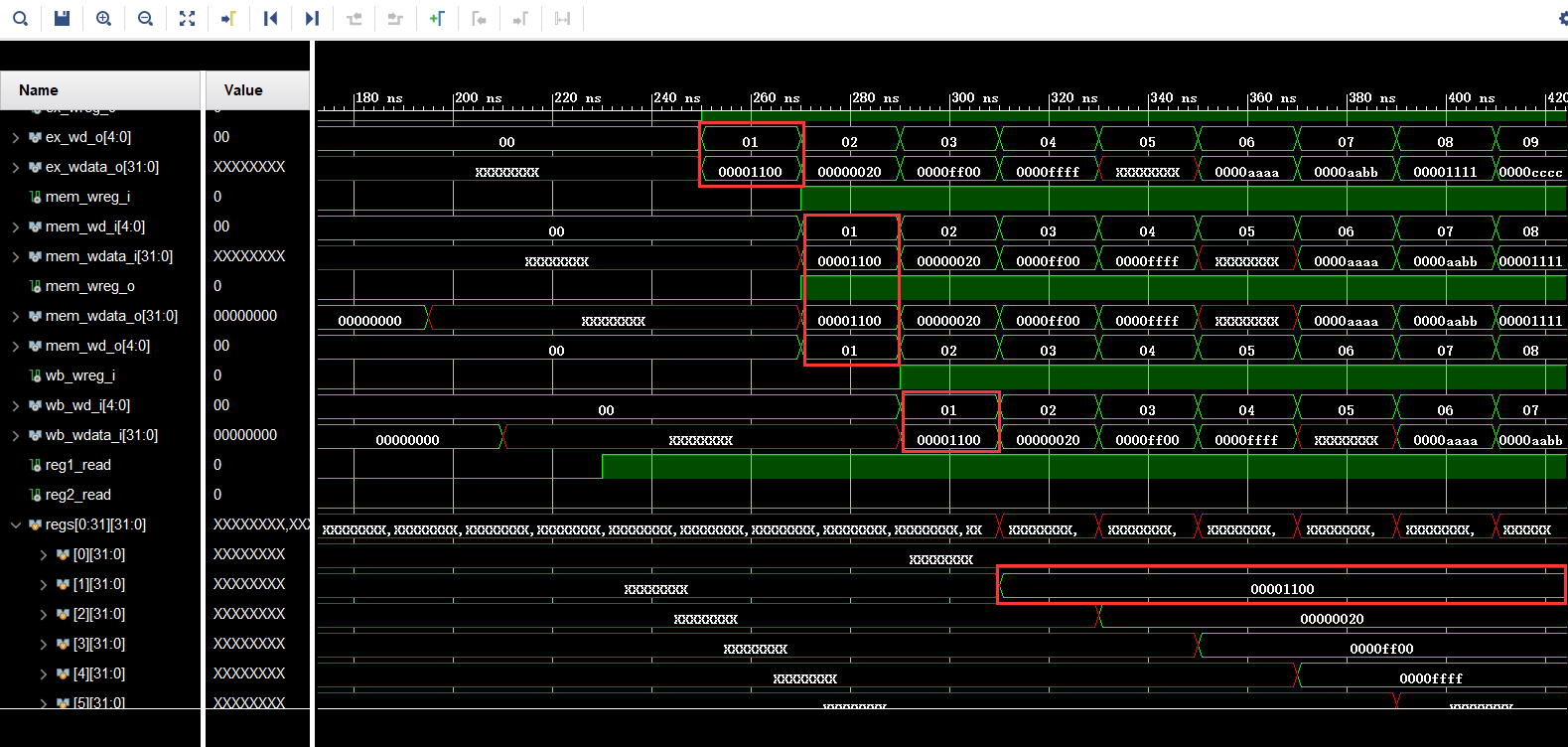
### 仿真

仿真结果：

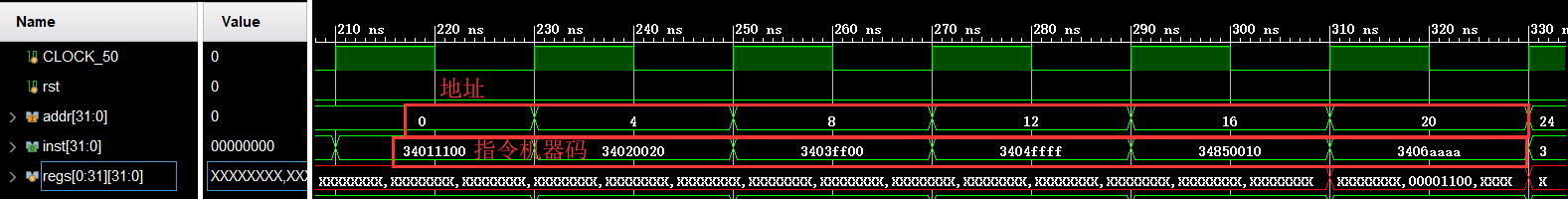


算是成功了。我们来具体分析一下：

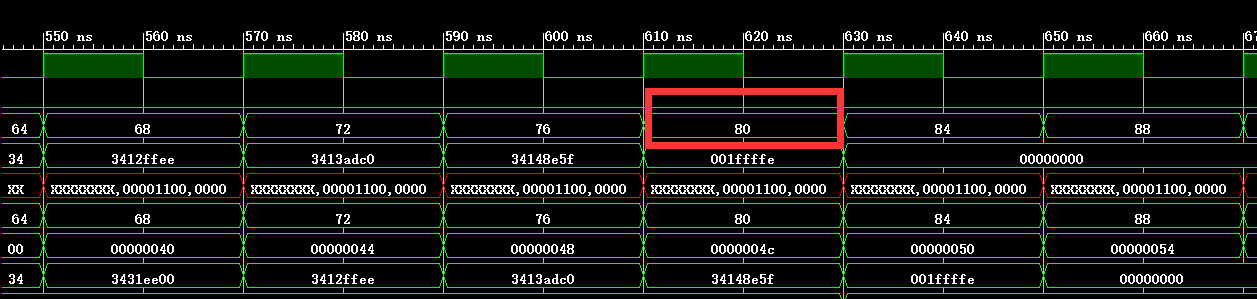




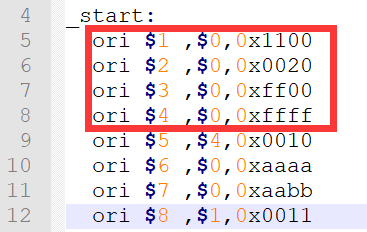
从第一条机器码34011100说起，先是通过IP模块，在下一个时钟周期从指令存储器通过IF/ID模块再通过一个周期读入ID模块，接着经过ID/EX模块的一个时钟周期进入EX模块，经过EX/MEM模块的一个时钟周期进入MEM模块，再经过一个MEM/WB的时钟周期回写到Regfile。



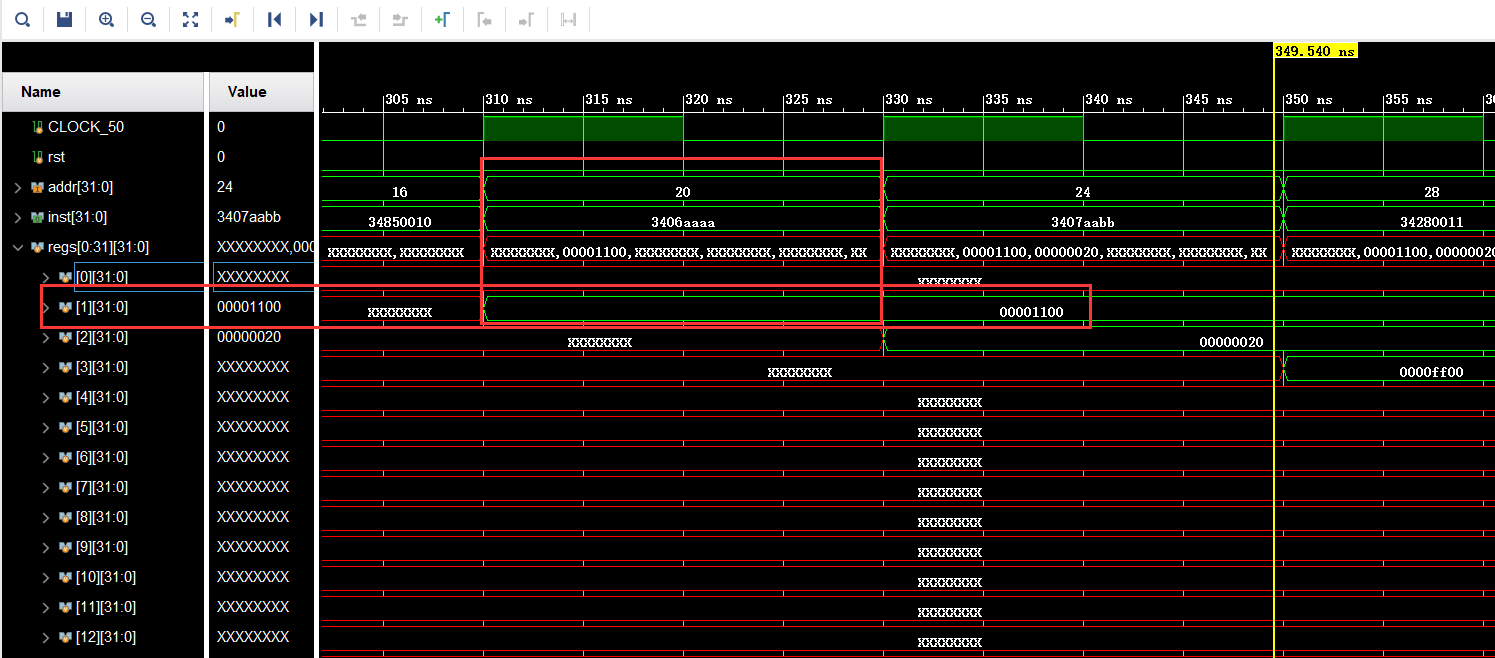
我们写了20条指令，地址应该到80（decimal），确实是这样：

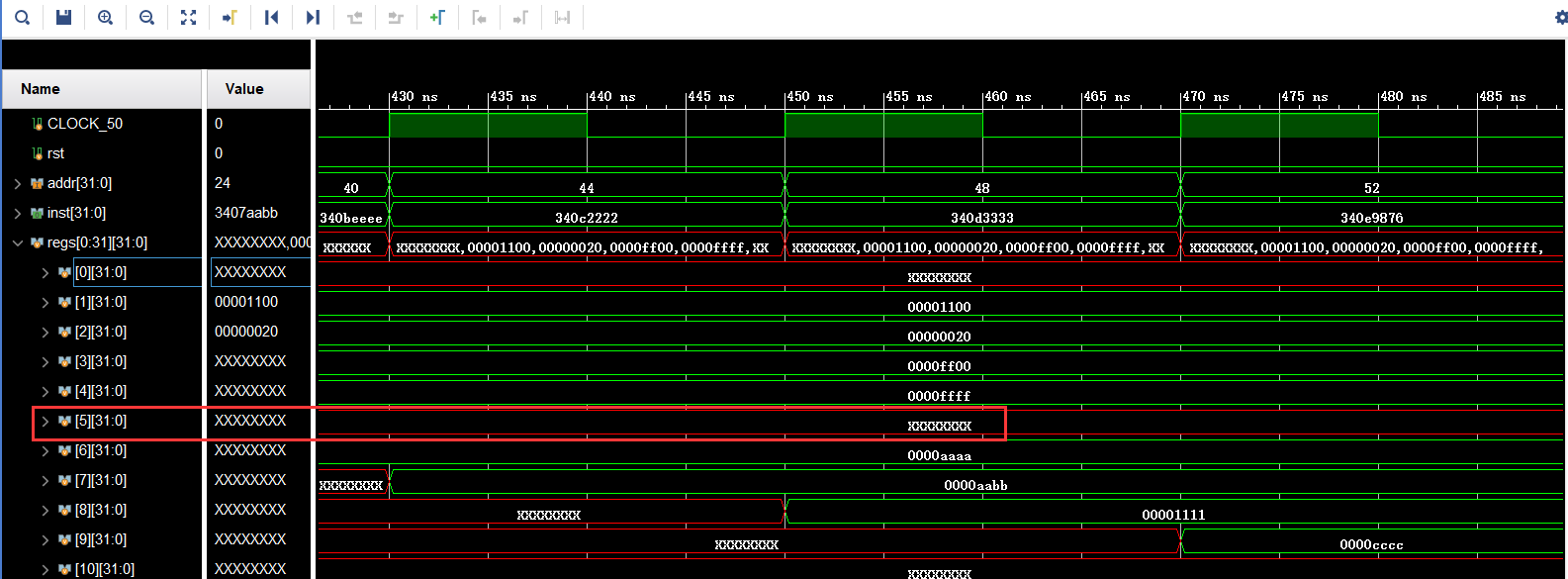


先看前几条指令



都是与0号寄存器做或运算，我们知道0号寄存器存放常量0，因此最终的值就是本身，我们查看寄存器：

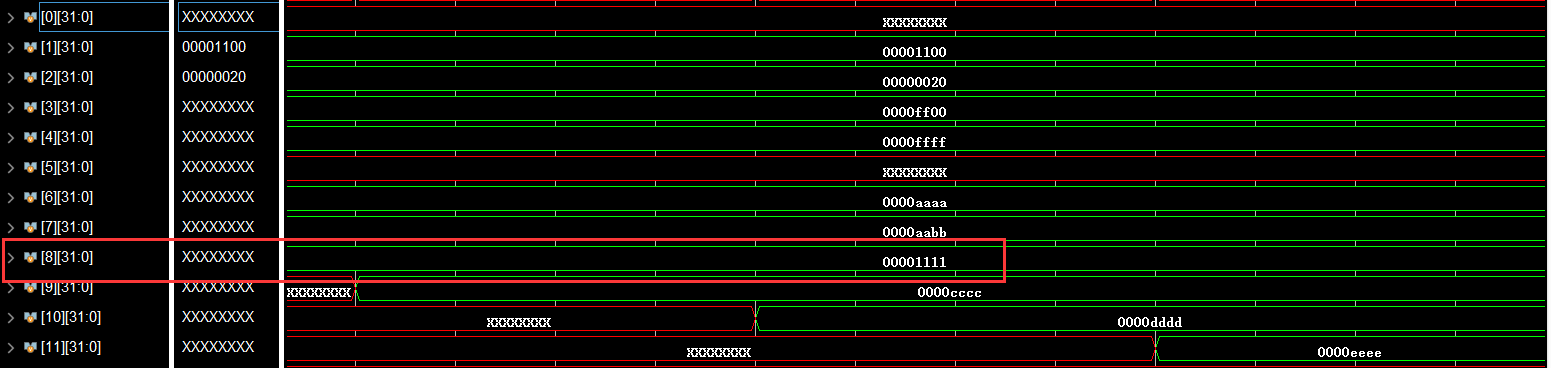




仔细观察仿真波形，有几个问题：为什么0号寄存器值为不定态？为什么5号寄存器也为不定态？为什么从地址为20的地方开始写入寄存器？让我们慢慢来说：

1. 为什么0号寄存器的值为不定态？观察我们的测试程序，发现我们是从$1，也就是1号寄存器开始写入数据，因为MIPS指令集系统的0号寄存器是常量，只存放0，我们没有对它进行初始化，所以便为不定态；
2. 为什么5号寄存器也是不定态？我们之前的流水线中，从读取指令进入OpenMISP之后需要经历取指、译码、执行、访存、回写五个模块，也就是五个时钟周期才能写入寄存器，这也就是五级流水名称的含义。我们第五条指令是，4号寄存器显然还没有进行完上述操作，所以值还没有写进去，因此取出来的值也是不定态，进行ORI运算过后还是不定态；
3. 最后为什么从地址为20的地方开始写入寄存器也就很好理解了，因为指令要经历五个时钟周期才能写入，每个时钟周期读取4字节的指令，五个周期后就是20字节了。

对于5号寄存器的值为不定态的情况，是因为4号寄存器的值还没有写进去，我们看第8条指令，用到了1号寄存器，但是在第八条指令执行时，1号寄存区早已经被写入了值，因此8号寄存器里面是有值的，为1100 | 0011 == 1111，我们看一下：



查看所有寄存器的值：



结果全部正确。

### 流水线数据相关

在之前的操作中，5号寄存器的值是不定态，是因为使用到了4号寄存器，而4号寄存器还没有被赋值，这就产生了流水线的数据相关，就是一个指令所用到的寄存器需要在五个周期后才能使用。我们提出了解决方案：

1. 插入暂停周期（stall），但是这样暂停会降低CPU的效率；
2. 编译器进行自动调度，智能改变指令的执行顺序，也并不是一个好的解决方案；
3. 数据前推（forwarding）：将计算结果从其产生处直接送到其他指令需要或所有需要的功能单元处，避免流水暂停。