自己动手写第一条 CPU 指令 ORI

一、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				

1.1 (无)符号拓展

有符号拓展:

0x8000 → 0xFFFF8000　　0x1000 → 0x00001000

无符号拓展

0x8000 → 0x00008000　　0x1000 → 0x00001000

1.2 MIPS 指令的 32 个通用寄存器

REGISTER	NAME	USAGE
$0	$zero	常量0(constant value 0)
$1	$at	保留给汇编器(Reserved for assembler)
$2-$3	$v0-$v1	函数调用返回值(values for results and expression evaluation)
$4-$7	$a0-$a3	函数调用参数(arguments)
$8-$15	$t0-$t7	暂时的(或随便用的)
$16-$23	$s0-$s7	保存的(或如果用，需要SAVE/RESTORE的)(saved)
$24-$25	$t8-$t9	暂时的(或随便用的)
$28	$gp	全局指针(Global Pointer)
$29	$sp	堆栈指针(Stack Pointer)
$30	$fp	帧指针(Frame Pointer)
$31	$ra	返回地址(return address)

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

一张比较官方的图：

0	`zero`	Always returns 0
1	`at`	(assembly temporary) Reserved for use by assembly
2–3	`v0, v1`	Value returned by subroutine
4–7	`a0–a3`	(arguments) First few parameters for a subroutine
8–15	`t0–t7`	(temporaries) Subroutines can use without saving
24, 25	`t8, t9`	
16–23	`s0–s7`	Subroutine register variables; a subroutine that writes one of these must save the old value and restore it before it exits, so the *calling* routine sees the values preserved
26, 27	`k0, k1`	Reserved for use by interrupt/trap handler; may change under your feet
28	`gp`	Global pointer; some runtime systems maintain this to give easy access to (some) `static` or `extern` variables
29	`sp`	Stack pointer
30	`s8/fp`	Ninth register variable; subroutines that need one can use this as a frame pointer
31	`ra`	Return address for subroutine

1.3 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

二、流水线的建立

2.1 流水线的简单模型

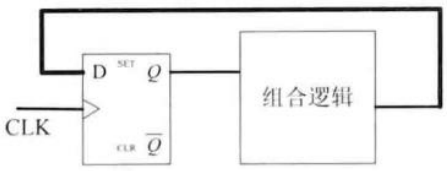


图 4-2　状态机的简单模型

图 4-3　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

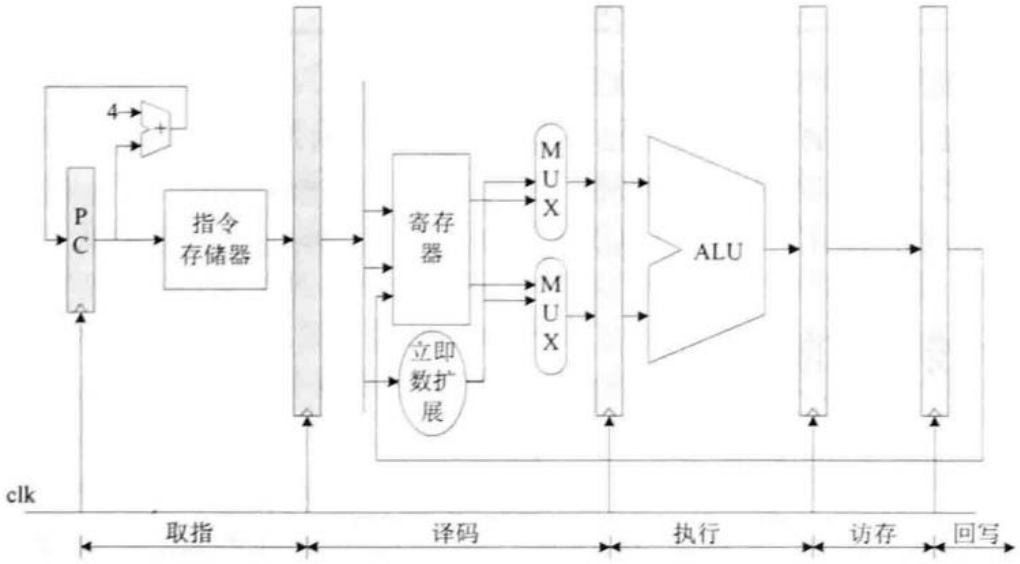


图 4-4　原始的数据流图

各个阶段完成的主要工作如下：

- 取指：取出指令存储器中的指令，PC 值递增，准备取下一条指令。
- 译码：对指令进行译码，依据译码结果，从 32 个通用寄存器中取出源操作数，有的指令要求两个源操作数都是寄存器的值，比如 or 指令，有的指令要求其中一个源操作数是指令中立即数的扩展，比如 ori 指令，所以这里有两个复用器，用于依据指令要求，确定参与运算的操作数，最终确定的两个操作数会送到执行阶段。
- 执行阶段：依据译码阶段送入的源操作数、操作码，进行运算，对于 ori 指令而言，就是进行逻辑“或”运算，运算结果传递到访存阶段。
- 访存阶段：对于 ori 指令，在访存阶段没有任何操作，直接将运算结果向下传递到回写阶段。
- 回写阶段：将运算结果保存到目的寄存器。

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：

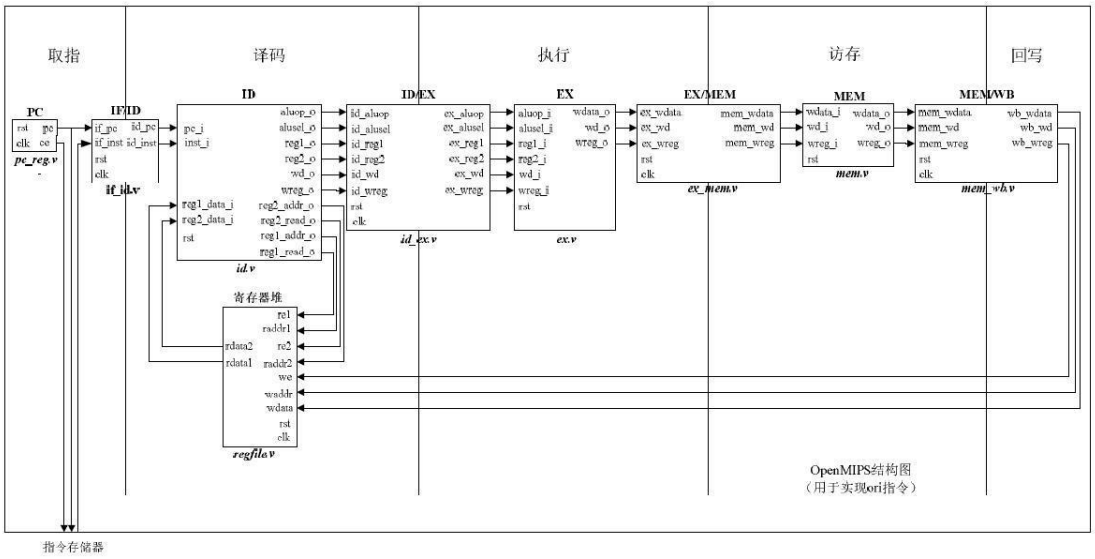


图 4-5 原始的 OpenMIPS 五级流水线结构图

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

表 4-2 PC 模块的接口描述

序　号	接 口 名	宽度（bit）	输入/输出	作　用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	pc	32	输出	要读取的指令地址
4	ce	1	输出	指令存储器使能信号

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：

表 4-3　IF/ID 模块的接口描述

序　号	接 口 名	宽度（bit）	输入/输出	作　用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	if_pc	32	输入	取指阶段取得的指令对应的地址
4	if_inst	32	输入	取指阶段取得的指令
5	id_pc	32	输出	译码阶段的指令对应的地址
6	id_inst	32	输出	译码阶段的指令

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

表 4-4　Regfile 模块接口描述表

序　号	接 口 名	宽度（bit）	输入/输出	作　用
1	rst	1	输入	复位信号，高电平有效
2	clk	1	输入	时钟信号
3	waddr	5	输入	要写入的寄存器地址
4	wdata	32	输入	要写入的数据
5	we	1	输入	写使能信号
6	raddr1	5	输入	第一个读寄存器端口要读取的寄存器的地址
7	re1	1	输入	第一个读寄存器端口读使能信号
8	rdata1	32	输出	第一个读寄存器端口输出的寄存器值
9	raddr2	5	输入	第二个读寄存器端口要读取的寄存器的地址
10	re2	1	输入	第二个读寄存器端口读使能信号
11	rdata2	32	输出	第二个读寄存器端口输出的寄存器值

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
```

（1）第一段：定义了一个二维的向量，元素个数是 RegNum，这是在 defines.v 中的一个宏定义，为 32，每个元素的宽度是 RegBus，这也是在 defines.v 中的一个宏定义，也为 32，所以此处定义的就是 32 个 32 位寄存器。

（2）第二段：实现了写寄存器操作，当复位信号无效时（rst 为 RstDisable），在写使能信号 we 有效（we 为 WriteEnable），且写操作目的寄存器不等于 0 的情况下，可以将写输入数据保存到目的寄存器。之所以要判断目的寄存器不为 0，是因为 MIPS32 架构规定$0 的值只能为 0，所以不要写入。WriteEnable 是 defines.v 中定义的宏，表示写使能信号有效，这些宏定义的含义十分明显，从名称上就可以知道具体含义，所以本书后面对宏定义不再作出说明，除非这个宏定义的含义从名称上不易明白。

（3）第三段：实现了第一个读寄存器端口，分以下几步依次判断：

- 当复位信号有效时，第一个读寄存器端口的输出始终为 0；
- 当复位信号无效时，如果读取的是$0，那么直接给出 0；
- 如果第一个读寄存器端口要读取的目标寄存器与要写入的目的寄存器是同一个寄存器，那么直接将要写入的值作为第一个读寄存器端口的输出；
- 如果上述情况都不满足，那么给出第一个读寄存器端口要读取的目标寄存器地址对应寄存器的值；
- 第一个读寄存器端口不能使用时，直接输出 0。

（4）第四段：实现了第二个读寄存器端口，具体过程与第三段是相似的，不再重复解释。

注意一点：读寄存器操作是组合逻辑电路，也就是一旦输入的要读取的寄存器地址 raddr1 或者 raddr2 发生变化，那么会立即给出新地址对应的寄存器的值，这样可以保证在译码阶段取得要读取的寄存器的值，而写寄存器操作是时序逻辑电路，写操作发生在时钟信号的上升沿。

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：

表 4-5 ID 模块的接口描述

序　号	接 口 名	宽度（bit）	输入/输出	作　用
1	rst	1	输入	复位信号
2	pc_i	32	输入	译码阶段的指令对应的地址
3	inst_i	32	输入	译码阶段的指令
4	reg1_data_i	32	输入	从 Regfile 输入的第一个读寄存器端口的输入
5	reg2_data_i	32	输入	从 Regfile 输入的第二个读寄存器端口的输入
6	reg1_read_o	1	输出	Regfile 模块的第一个读寄存器端口的读使能信号
7	reg2_read_o	1	输出	Regfile 模块的第二个读寄存器端口的读使能信号
8	reg1_addr_o	5	输出	Regfile 模块的第一个读寄存器端口的读地址信号
9	reg2_addr_o	5	输出	Regfile 模块的第二个读寄存器端口的读地址信号
10	aluop_o	8	输出	译码阶段的指令要进行的运算的子类型
11	alusel_o	3	输出	译码阶段的指令要进行的运算的类型
12	reg1_o	32	输出	译码阶段的指令要进行的运算的源操作数 1
13	reg2_o	32	输出	译码阶段的指令要进行的运算的源操作数 2
14	wd_o	5	输出	译码阶段的指令要写入的目的寄存器地址
15	wreg_o	1	输出	译码阶段的指令是否有要写入的目的寄存器

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：

（1）第一段：实现了对指令的译码，依据指令中的特征字段区分指令，对指令 ori 而言，只需通过识别 26-31bit 的指令码是否为 6'b001101，即可判断是否是 ori 指令，其中的宏定义 EXE_ORI 就是 6'b001101，op 就是指令的 26-31bit，所以当 op 等于 EXE_ORI 时，就表示是 ori 指令，此时会有以下译码结果。

- 要读取的寄存器情况：ori 指令只需要读取 rs 寄存器的值，默认通过 Regfile 读端口 1 读取的寄存器地址 reg1_addr_o 的值是指令的 21-25bit，参考图 4-1 可知，正是 ori 指令中的 rs，所以设置 reg1_read_o 为 1，通过图 4-5 可以看出，reg1_read_o 连接 Regfile 的输入 re1，reg1_addr_o 连接 Regfile 的输入 raddr1，结合对 Regfile 模块的介绍可知，译码阶段会读取寄存器 rs 的值。指令 ori 需要的另一个操作数是立即数，所以设置 reg2_read_o 为 0，表示不通过 Regfile 读端口 2 读取寄存器，这里暗含使用立即数作为运算的操作数。imm 就是指令中的立即数进行零扩展后的值。
- 要执行的运算：alusel_o 给出要执行的运算类型，对于 ori 指令而言就是逻辑操作，即 EXE_RES_LOGIC。aluop_o 给出要执行的运算子类型，对于 ori 指令而言就是逻辑“或”运算，即 EXE_OR_OP。这两个值会传递到执行阶段。
- 要写入的目的寄存器：wreg_o 表示是否要写目的寄存器，ori 指令要将计算结果保存到寄存器中，所以 wreg_o 设置为 WriteEnable。wd_o 是要写入的目的寄存器地址，此时就是指令的 16-20bit，参考图 4-1 可知，正是 ori 指令中的 rt。这两个值也会传递到执行阶段。

（2）第二段：给出参与运算的源操作数 1 的值，如果 reg1_read_o 为 1，那么就将从 Regfile 模块读端口 1 读取的寄存器的值作为源操作数 1，如果 reg1_read_o 为 0，那么就将立即数作为源操作数 1，对于 ori 而言，此处选择从 Regfile 模块读端口 1 读取的寄存器的值作为源操作数 1。该值将通过 reg1_o 端口被传递到执行阶段。

（3）第三段：给出参与运算的源操作数 2 的值，如果 reg2_read_o 为 1，那么就将从 Regfile 模块读端口 2 读取的寄存器的值作为源操作数 2，如果 reg2_read_o 为 0，那么就将立即数作为源操作数 2，对于 ori 而言，此处选择立即数 imm 作为源操作数 2。该值将通过 reg2_o 端口被传递到执行阶段。

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

表 4-6　ID/EX 模块的接口描述

序　号	接 口 名	宽度（bit）	输入/输出	作　用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	id_alusel	3	输入	译码阶段的指令要进行的运算的类型
4	id_aluop	8	输入	译码阶段的指令要进行的运算的子类型
5	id_reg1	32	输入	译码阶段的指令要进行的运算的源操作数 1
6	id_reg2	32	输入	译码阶段的指令要进行的运算的源操作数 2
7	id_wd	5	输入	译码阶段的指令要写入的目的寄存器地址
8	id_wreg	1	输入	译码阶段的指令是否有要写入的目的寄存器
9	ex_alusel	3	输出	执行阶段的指令要进行的运算的类型
10	ex_aluop	8	输出	执行阶段的指令要进行的运算的子类型
11	ex_reg1	32	输出	执行阶段的指令要进行的运算的源操作数 1
12	ex_reg2	32	输出	执行阶段的指令要进行的运算的源操作数 2
13	ex_wd	5	输出	执行阶段的指令要写入的目的寄存器地址
14	ex_wreg	1	输出	执行阶段的指令是否有要写入的目的寄存器

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。

1. 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：

表 4-7 EX 模块的接口描述

序　号	接 口 名	宽度（bit）	输入/输出	作　用
1	rst	1	输入	复位信号
2	alusel_i	3	输入	执行阶段要进行的运算的类型
3	aluop_i	8	输入	执行阶段要进行的运算的子类型
4	reg1_i	32	输入	参与运算的源操作数 1
5	reg2_i	32	输入	参与运算的源操作数 2
6	wd_i	5	输入	指令执行要写入的目的寄存器地址
7	wreg_i	1	输入	是否有要写入的目的寄存器
8	wd_o	5	输出	执行阶段的指令最终要写入的目的寄存器地址
9	wreg_o	1	输出	执行阶段的指令最终是否有要写入的目的寄存器
10	wdata_o	32	输出	执行阶段的指令最终要写入目的寄存器的值

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。

2. 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：

表 4-8　EX/MEM 模块的接口描述

序　号	接 口 名	宽度（bit）	输入/输出	作　用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	ex_wd	5	输入	执行阶段的指令执行后要写入的目的寄存器地址
4	ex_wreg	1	输入	执行阶段的指令执行后是否有要写入的目的寄存器
5	ex_wdata	32	输入	执行阶段的指令执行后要写入目的寄存器的值
6	mem_wd	5	输出	访存阶段的指令要写入的目的寄存器地址
7	mem_wreg	1	输出	访存阶段的指令是否有要写入的目的寄存器
8	mem_wdata	32	输出	访存阶段的指令要写入目的寄存器的值

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。

2.7 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。

1. MEM 模块

接口描述如表：

表 4-9 MEM 模块的接口描述

序　号	接 口 名	宽度（bit）	输入/输出	作　用
1	rst	1	输入	复位信号
2	wd_i	5	输入	访存阶段的指令要写入的目的寄存器地址
3	wreg_i	1	输入	访存阶段的指令是否有要写入的目的寄存器
4	wdata_i	32	输入	访存阶段的指令要写入目的寄存器的值
5	wd_o	5	输出	访存阶段的指令最终要写入的目的寄存器地址
6	wreg_o	1	输出	访存阶段的指令最终是否有要写入的目的寄存器
7	wdata_o	32	输出	访存阶段的指令最终要写入目的寄存器的值

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。

2. 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：

表 4-10　MEM/WB 模块的接口描述

序　号	接 口 名	宽度（bit）	输入/输出	作　用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	mem_wd	5	输入	访存阶段的指令最终要写入的目的寄存器地址
4	mem_wreg	1	输入	访存阶段的指令最终是否有要写入的目的寄存器
5	mem_wdata	32	输入	访存阶段的指令最终要写入目的寄存器的值
6	wb_wd	5	输出	回写阶段的指令要写入的目的寄存器地址
7	wb_wreg	1	输出	回写阶段的指令是否有要写入的目的寄存器
8	wb_wdata	32	输出	回写阶段的指令要写入目的寄存器的值

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…

2.8 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。

2.9 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：

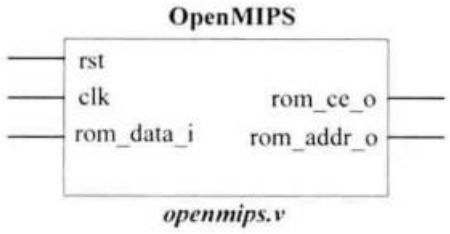


图 4-6　OpenMIPS 模块接口图

其接口描述如表：

表 4-11　OpenMIPS 模块的接口描述

序　号	接 口 名	宽度（bit）	输入/输出	作　用
1	rst	1	输入	复位信号
2	clk	1	输入	时钟信号
3	rom_data_i	32	输入	从指令存储器取得的指令
4	rom_addr_o	32	输出	输出到指令存储器的地址
5	rom_ce_o	1	输出	指令存储器使能信号

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
```

三、验证 OpenMIPS 实现效果

3.1 指令存储器 ROM 的实现

模块图如图：

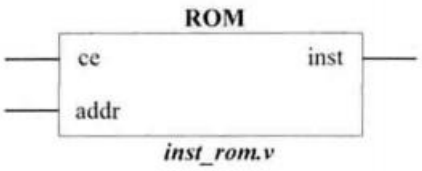


图 4-7　指令存储器 ROM 模块接口图

接口描述如表：

表 4-12　指令存储器 ROM 模块的接口描述

序　号	接 口 名	宽度（bit）	输入/输出	作　用
1	ce	1	输入	使能信号
2	addr	32	输入	要读取的指令地址
3	inst	32	输出	读出的指令

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。

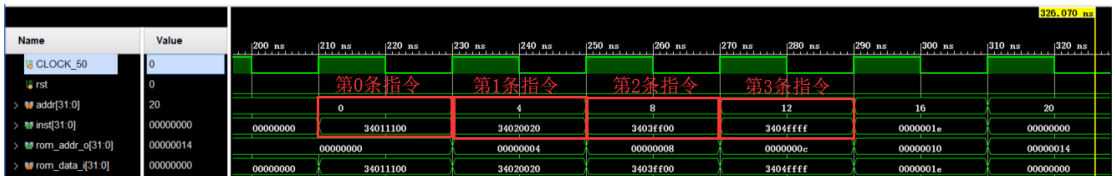
首先使用了 initial 语句，该语句执行过程中只会执行一次，，通常用于仿真模块中对激励向量的描述，或用于给变量赋初值，是面向模拟仿真的过程语句，通常被综合工具支持，如果要将用该方法实现的 OpenMIPS 处理器使用综合工具进行综合，要修改这里的初始化指令存储器的方法。

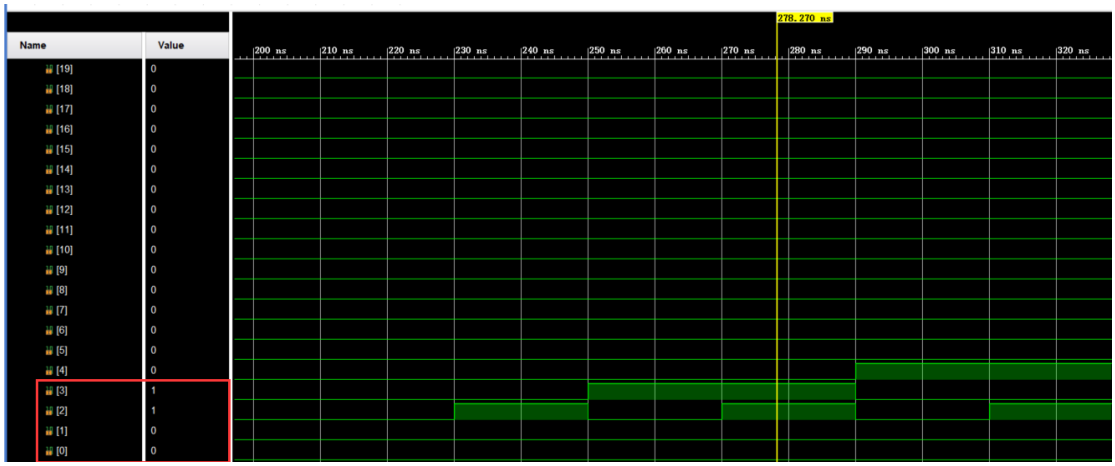
然后对于系统函数$readmemh，表示从 inst_rom.data 问价那种读取数据有以初始化 isnt_mem，inst_mem 是定义的一个大小为 128KB（131072 字节）的数组，每个数组的元素是一个 32 位的数据，用常量 InstAddrBus 表示，因为在 MIPS 指令集中，每一条指令的大小是固定的 4 字节（32bit），相当于每一个数组元素表示一条指令，可以存放 131072 条指令，该大小也就是 ROM 的实际大小。

对于**inst <= inst_mem[addr[`InstMemNumLog2 + 1 : 2]]** 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2^x的那个 x 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：

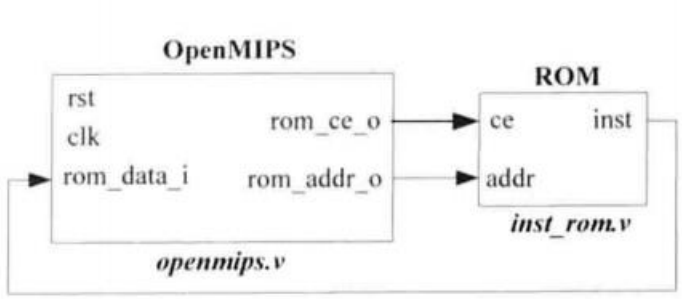


图 4-9　最小 SOPC 的结构

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。

1. 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
```

2. 用 GCC 编译

```
cherry@cherry-localhost: ~/桌面
cherry@cherry-localhost:~$ cd 桌面
cherry@cherry-localhost:~/桌面$ make clean
rm -f *.o *.om *.bin *.data
cherry@cherry-localhost:~/桌面$ make all
mipsel-linux-as -mips32 -EB inst_rom.S -o inst_rom.o
inst_rom.S: Assembler messages:
inst_rom.S:0: Warning: end of file not at end of a line; newline inserted
mipsel-linux-ld -EB -T ram.ld inst_rom.o -o inst_rom.om
mipsel-linux-objcopy -O binary inst_rom.om inst_rom.bin
./Bin2Mem.exe -f inst_rom.bin -o inst_rom.data
mipsel-linux-objdump -D inst_rom.om >inst_rom.asm
cherry@cherry-localhost:~/桌面$
```

查看生成的 asm 汇编文件和机器码：

inst_rom.data × inst_rom.asm ×

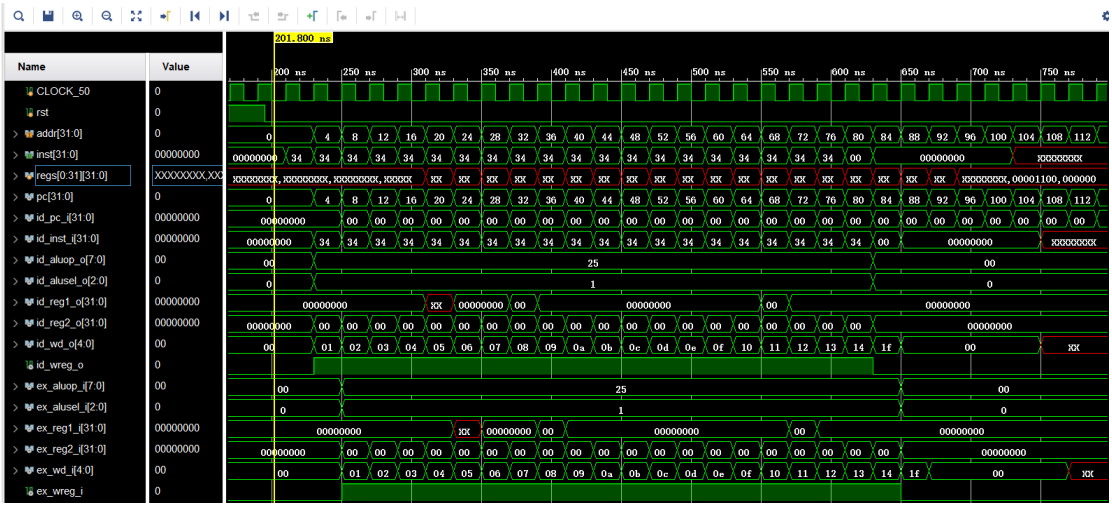
```
Disassembly of section .text:

00000000 <_start>:
   0:   34011100        li      at,0x1100
   4:   34020020        li      v0,0x20
   8:   3403ff00        li      v1,0xff00
   c:   3404ffff        li      a0,0xffff
  10:   34850010        ori     a1,a0,0x10
  14:   3406aaaa        li      a2,0xaaaa
  18:   3407aabb        li      a3,0xaabb
  1c:   34280011        ori     t0,at,0x11
  20:   3409cccc        li      t1,0xcccc
  24:   340adddd        li      t2,0xdddd
  28:   340beeee        li      t3,0xeeee
  2c:   340c2222        li      t4,0x2222
  30:   340d3333        li      t5,0x3333
  34:   340e9876        li      t6,0x9876
  38:   340fabcd        li      t7,0xabcd
  3c:   3410fedc        li      s0,0xfedc
  40:   3431ee00        ori     s1,at,0xee00
  44:   3412ffee        li      s2,0xffee
  48:   3413adc0        li      s3,0xadc0
  4c:   34148e5f        li      s4,0x8e5f

Disassembly of section .reginfo:
```

inst_rom.data ×

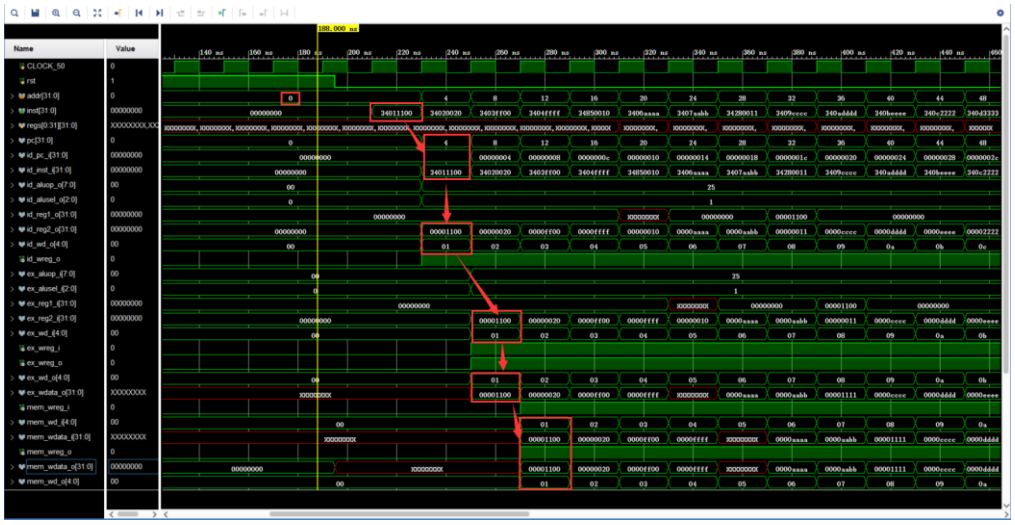
```
34011100
34020020
3403ff00
3404ffff
34850010
3406aaaa
3407aabb
34280011
3409cccc
340adddd
340beeee
340c2222
340d3333
340e9876
340fabcd
3410fedc
3431ee00
3412ffee
3413adc0
34148e5f
001ffffe
00000000
00000000
00000000
00000000
00000000
```

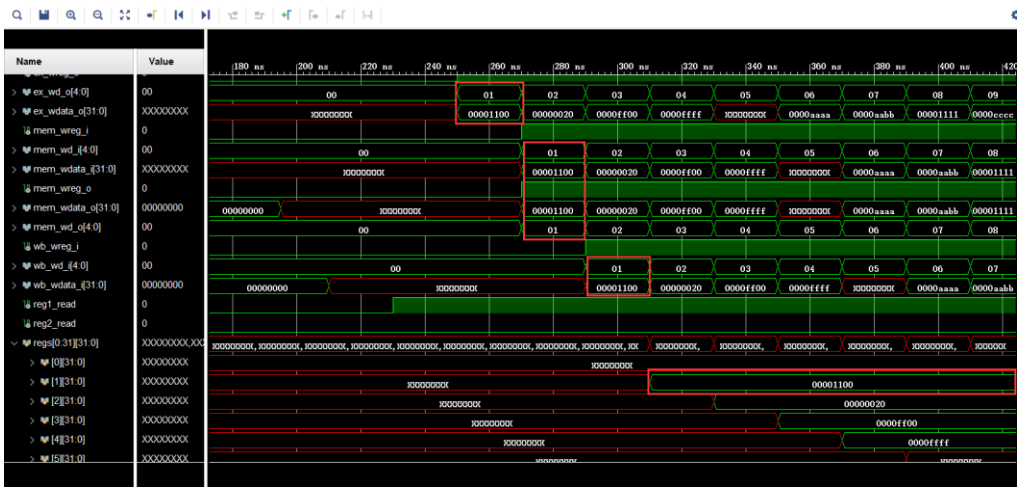
3. 仿真

仿真结果：

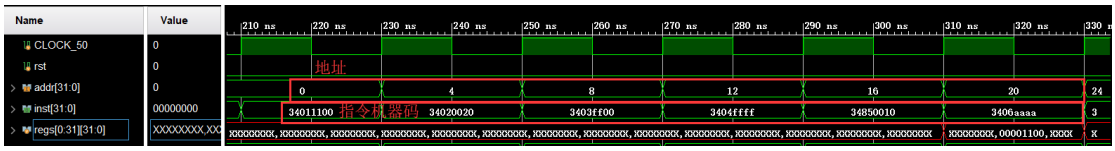


算是成功了。我们来具体分析一下：

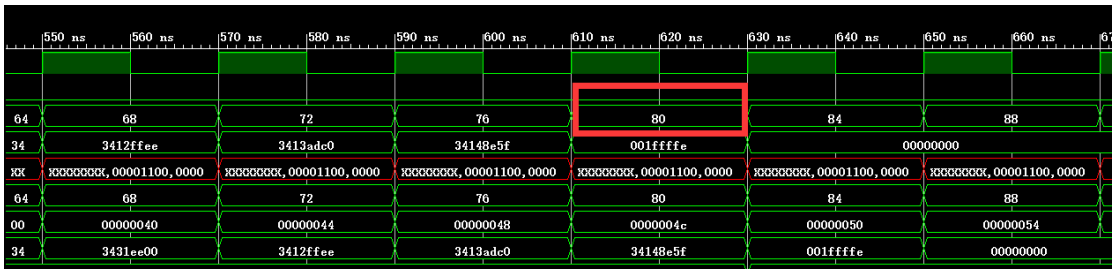




从第一条机器码 34011100 说起，先是通过 IP 模块，在下一个时钟周期从指令存储器通过 IF/ID 模块再通过一个周期读入 ID 模块，接着经过 ID/EX 模块的一个时钟周期进入 EX 模块，经过 EX/MEM 模块的一个时钟周期进入 MEM 模块，再经过一个 MEM/WB 的时钟周期回写到 Regfile。



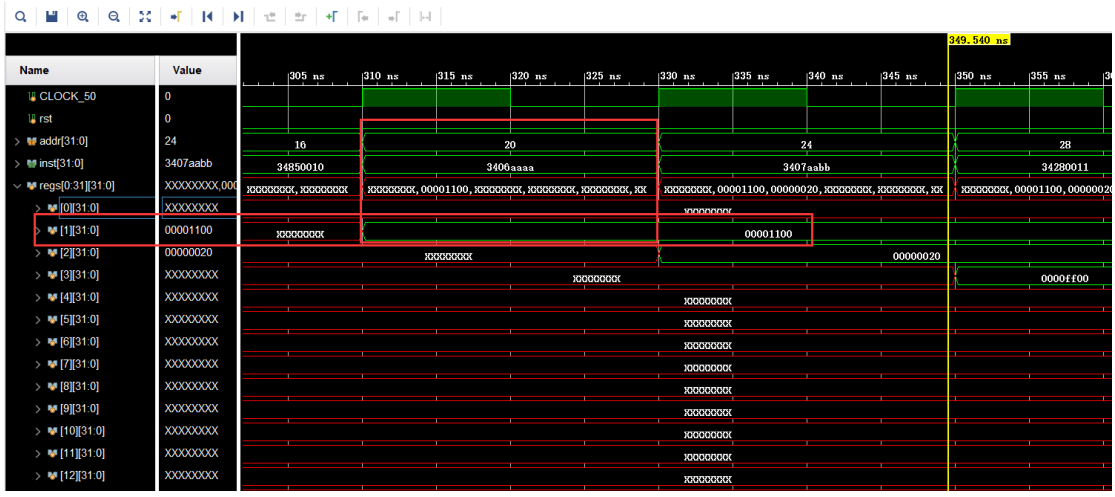
我们写了 20 条指令，地址应该到 80（decimal），确实是这样：

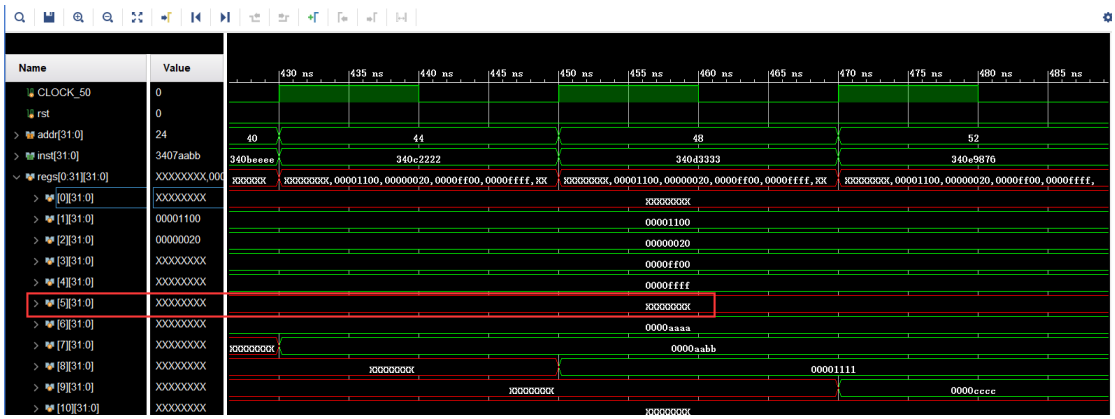


先看前几条指令

```
4   start:
5     ori $1 ,$0,0x1100
6     ori $2 ,$0,0x0020
7     ori $3 ,$0,0xff00
8     ori $4 ,$0,0xffff
9     ori $5 ,$4,0x0010
10    ori $6 ,$0,0xaaaa
11    ori $7 ,$0,0xaabb
12    ori $8 ,$1,0x0011
```

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：





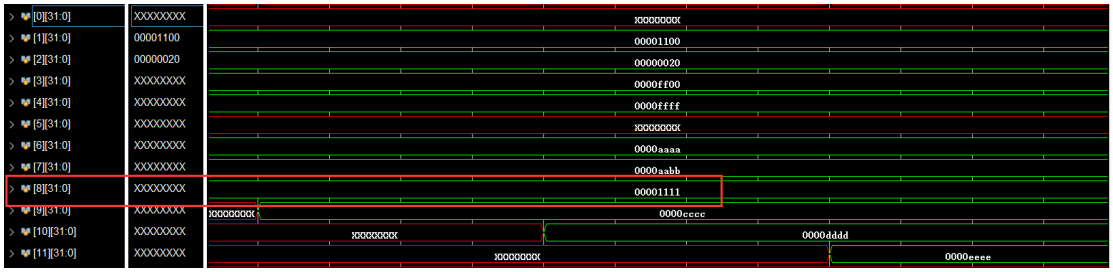
仔细观察仿真波形，有几个问题：为什么 0 号寄存器值为不定态？为什么 5 号寄存器也为不定态？为什么从地址为 20 的地方开始写入寄存器？让我们慢慢来说：

（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；

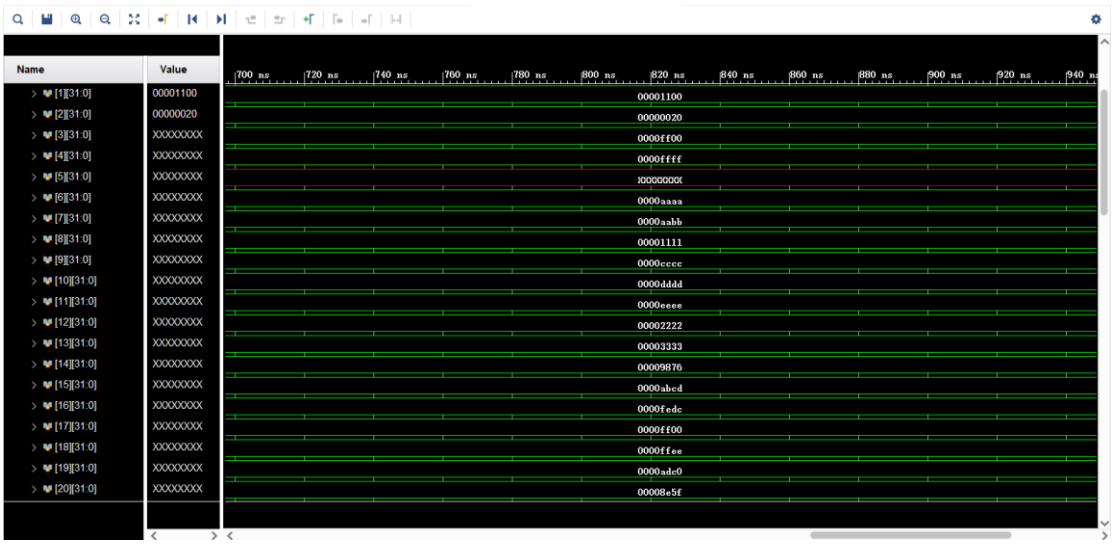
（2） 为什么 5 号寄存器也是不定态？我们之前的流水线中，从读取指令进入 OpenMISP 之后需要经历取指、译码、执行、访存、回写五个模块，也就是五个时钟周期才能写入寄存器，这也就是五级流水名称的含义。我们第五条指令是ori $5 ,$4,0x0010，4 号寄存器显然还没有进行完上述操作，所以值还没有写进去，因此取出来的值也是不定态，进行 ORI 运算过后还是不定态；

（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ori $8 ,$1,0x0011，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。

4. 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。