计算机组成课设P4设计文档

1. 模块设计
2. IM

表1 IM接口定义表

|  |  |
| --- | --- |
| 文件 | 模块接口定义 |
| IM.v | module IM(  input [31:0] PC,  output [31:0] Instr  ); |

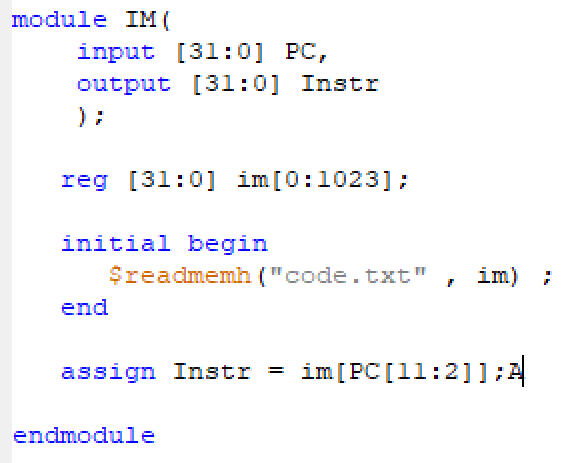


图1 IM代码

首先，IM模块需要初始化读入代码，并将所有代码保存到im中。out输出端口始终输出目前所执行的指令。

1. Controller

表2 控制器接口定义表

|  |  |
| --- | --- |
| 文件 | 模块接口定义 |
| Controller.v | module Controller(  //用来鉴别指令的三个输入，其中ThirdIn与Branch指令有关  input [5:0] Op,  input [5:0] Funct,  input [4:0] ThirdIn,  //关于寄存器和内存读写信号  output RegWrite,  output MemWrite,  output MemtoReg,  //用于多路选择器的信号位  output [1:0] RegDst,  output [1:0] ALUSrc,  output [1:0] ExtOp,  output [3:0] ALUOp,  //Jump和Branch的专用信号  output [2:0] Jump,  //output [2:0] Branch  output beq,  output jal,  output jr  ); |

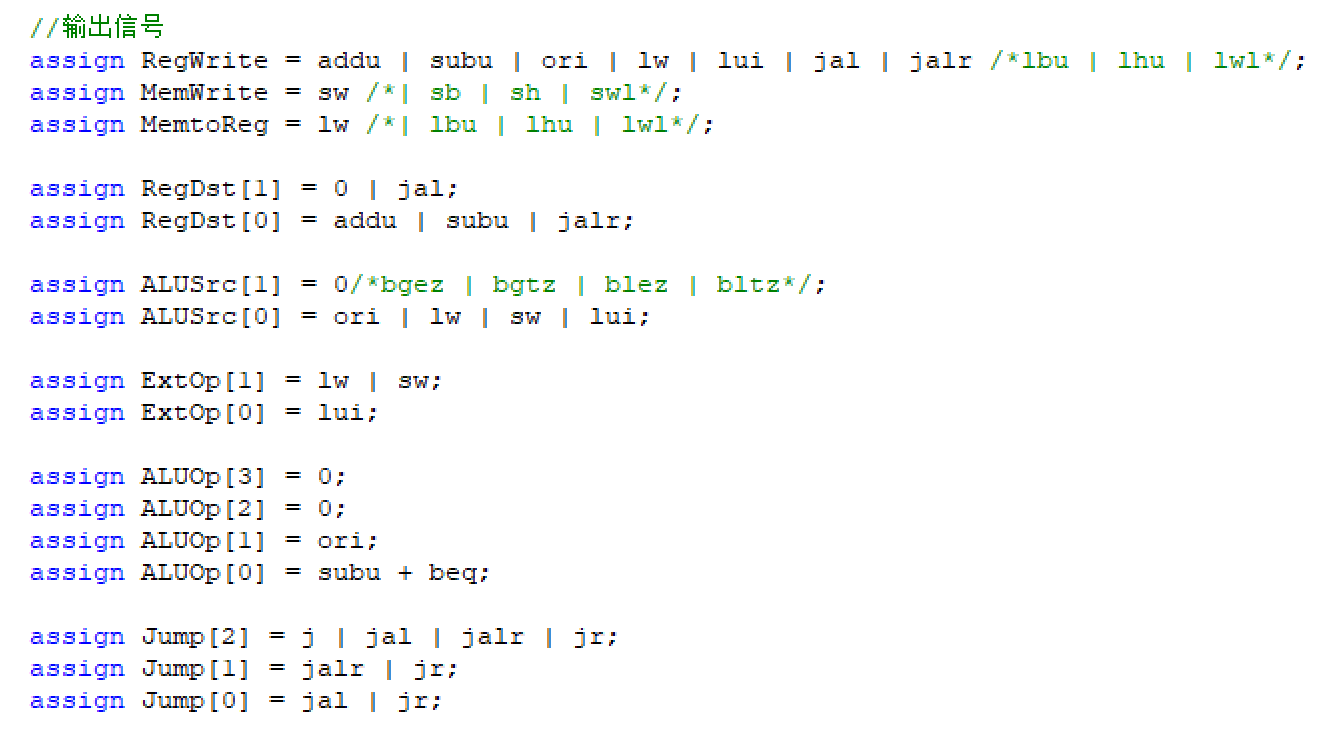


图2 Controller的或逻辑代码

用真值表方式判断指令的种类；用assign方式对各个控制器输出信号进行赋值。由于特殊性，R型指令不区分对待。

1. GRF

表3 GRF接口定义表

|  |  |
| --- | --- |
| 文件 | 模块接口定义 |
| GRF.v | module GRF(  input [31:0] PC,  input [4:0] A1,  input [4:0] A2,  input [4:0] A3,  output [31:0] RD1,  output [31:0] RD2,  input [31:0] WD,  input RegWrite,  input clk,  input reset  ); |

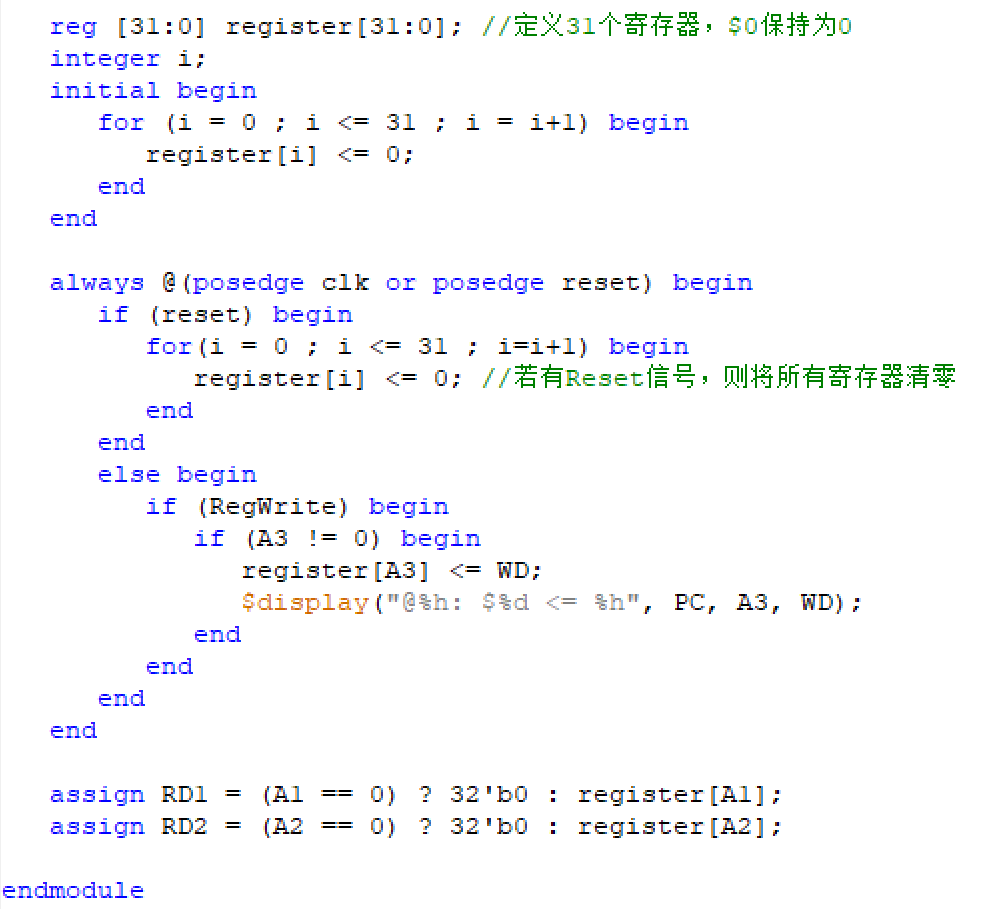


图3 GRF代码

首先需要对32个寄存器进行初始化。用assign语句输出A1与A2号寄存器所储存的值。对于每个时钟上沿，若写入使能信号为1，复位信号为0，并且写入地址不为0，则对A3号寄存器进行写入，并display相应内容。由于CPU要求同步复位，故reset应当在时钟上沿中进行。

1. ALU

表4 ALU接口定义表

|  |  |
| --- | --- |
| 文件 | 模块接口定义 |
| ALU.v | ALU\_ctl (fun, ALUOp, jr, Operation);  input [5:0] fun,  input [2:0] ALUOp,  output jr,  output reg [3:0] Operation |

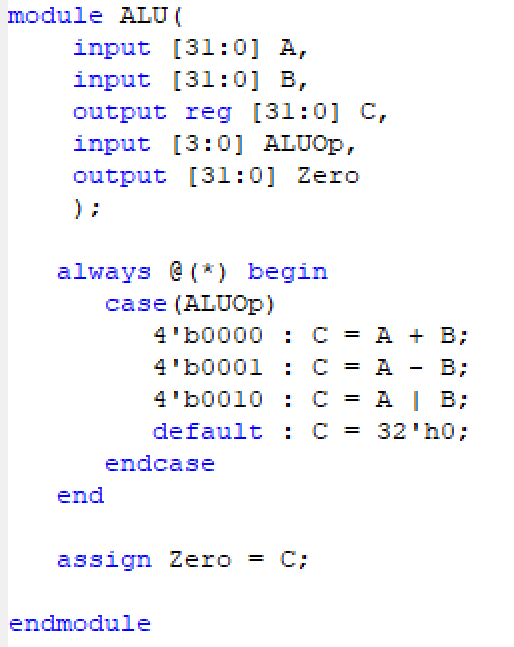


图4 ALU代码

对指令用真值表方式进行ALU运算类型解码。

1. DM

表5 DM接口定义表

|  |  |
| --- | --- |
| 文件 | 模块接口定义 |
| DM.v | module DM(  input [31:0] PC,  input [31:0] A,  input [31:0] WD,  output [31:0] RD,  input clk,  input reset,  input MemWrite  ); |

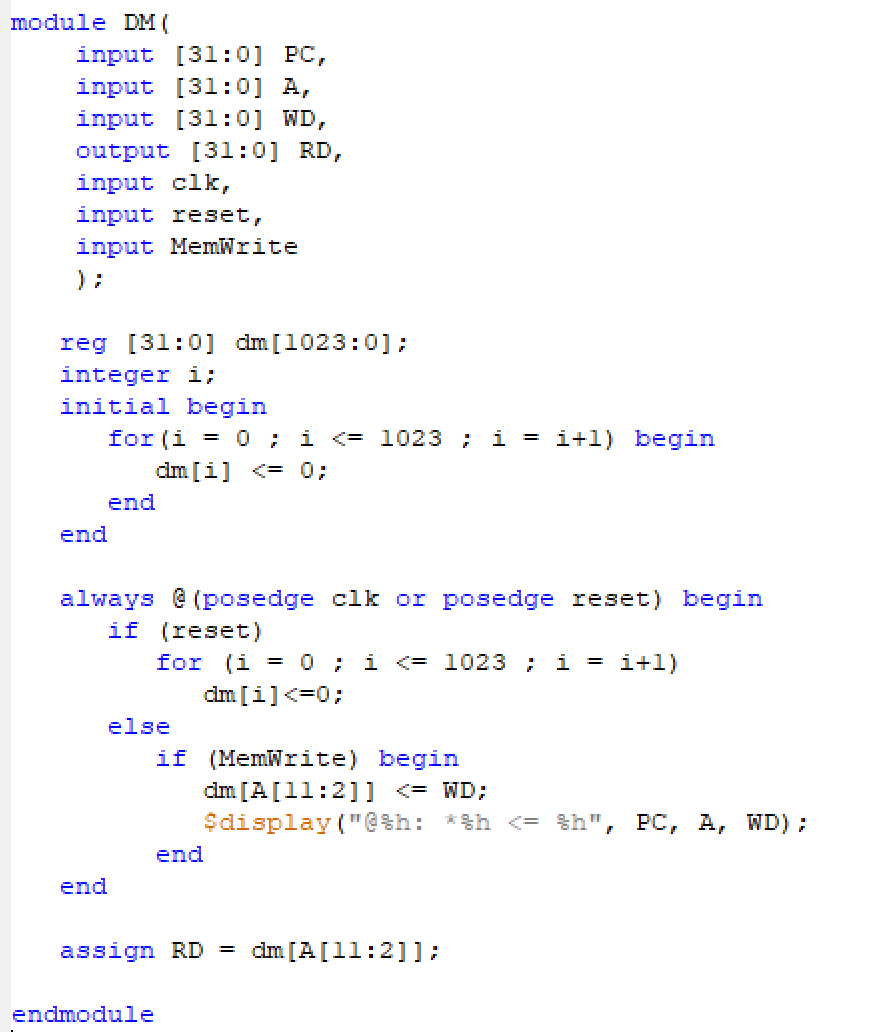


图5 DM代码

1. EXT

表6 EXT接口定义表

|  |  |
| --- | --- |
| 文件 | 模块接口定义 |
| EXT.v | module EXT(  input [15:0] Imm16,  output reg [31:0] Ext32,  input [1:0] ExtOp  ); |

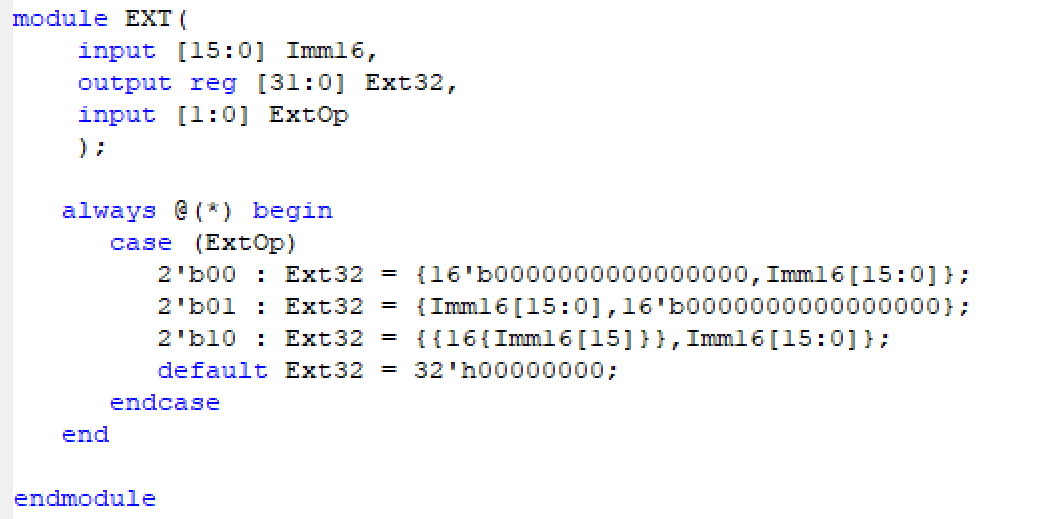
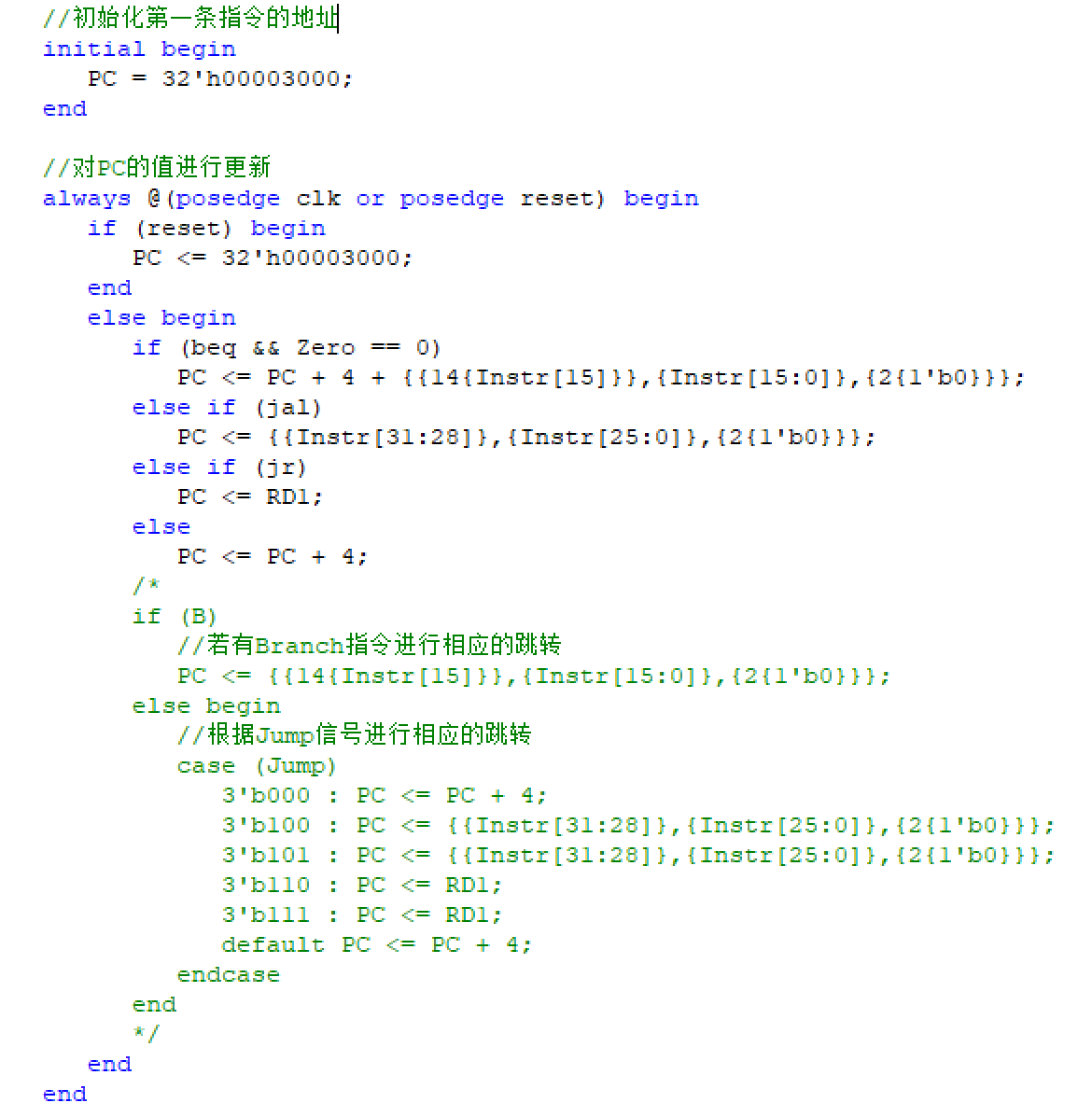
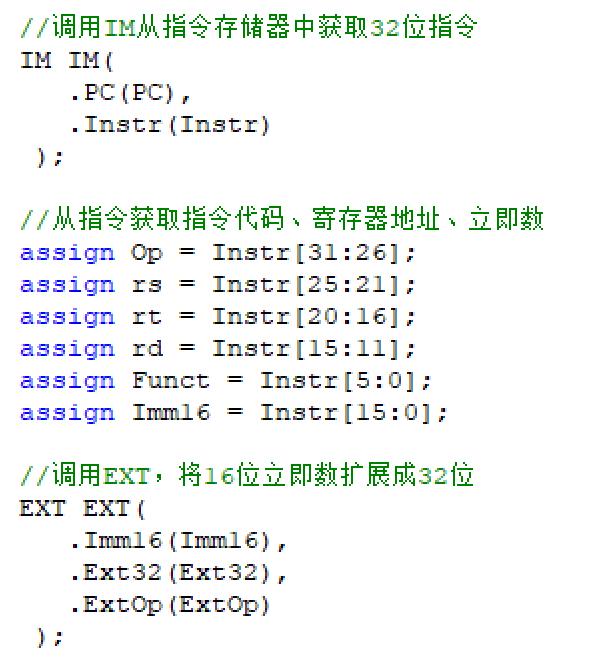


图6 EXT代码

由于ExtOp用2位信号表示，故可以表示4种扩展方式。

1. mips.v





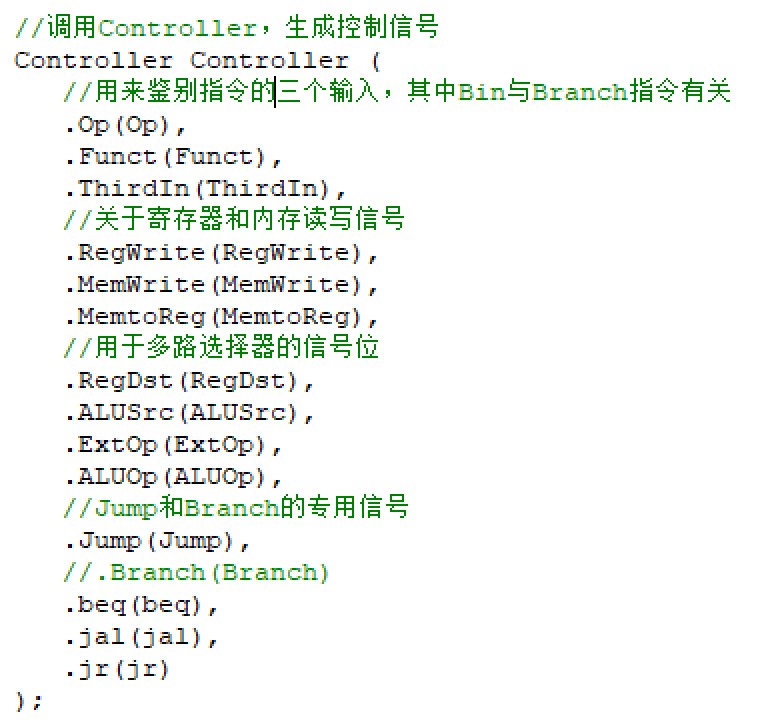




图7 mips.v代码

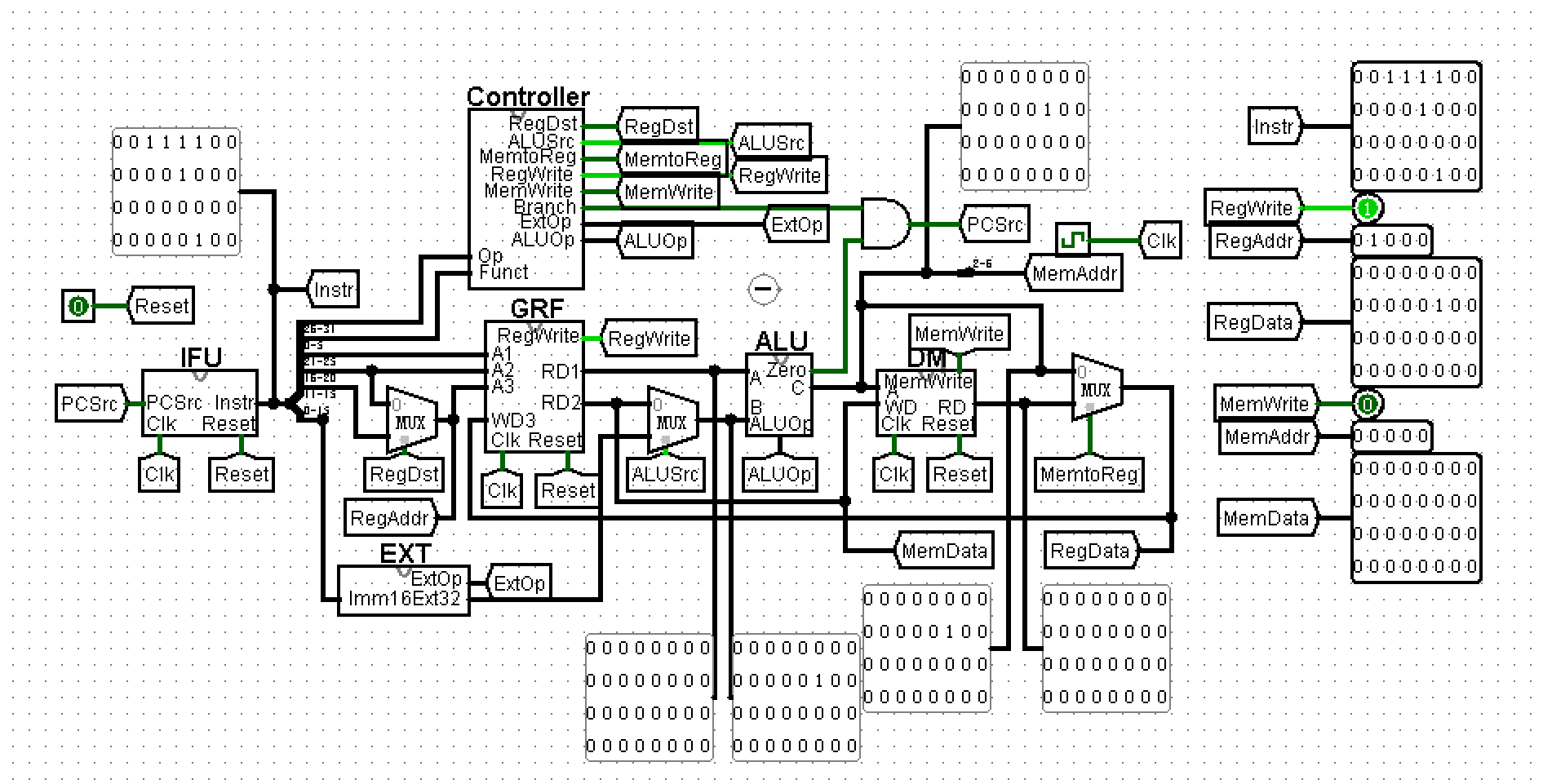


图8 CPU数据通路示意图（IFU部分有修改）

1. 测试程序

由于jal和jr指令较为复杂，需要在测试程序中建立一个循环并多次函数调用，以保证程序正确。

测试代码分为两部分，第一部分只用到了t系列寄存器，意在测试除jal，jr、nop以外的指令；第二部分只使用s系列和sp寄存器，意在测试jal和jr指令在递归过程中的正确性。

# test ori 测试ori指令

ori $t0, 100

ori $s0, 100

ori $s1, 200

ori $t1, 0

ori $t2, 0

# test $0 测试0号寄存器的值是否不变

ori $0, 0xffffffff

# test addu 测试addu指令

addu $a0, $0, $t0

# test subu 测试subu指令

subu $s1, $t0, $0

# test beq 测试beq指令

loop\_beq\_1:

addu $t1, $t1, $t0

beq $t0, $s0, loop\_beq\_2 # 测试向后跳转

ori $t3, 100 # 测试若等于条件不符合是否跳转

loop\_beq\_2:

addu $t2, $t2, $t0

addu $t0, $t0, 100

beq $t0, $s1, loop\_beq\_1 # 测试向前跳转

ori $t4, 0xffffffff

ori $t5, 0xffc

# test sw 测试sw指令，是否能存入所有地址

sw $t4, 0($0)

sw $t4, 4($0)

sw $t4, -4($t5)

sw $t4, 0($t5)

# test lw 测试lw指令，是否能取出所有地址

lw $t6, 0($0)

lw $t7, 4($0)

lw $t8, -4($t5)

lw $t9, 0($t5)

# test lui

lui $s2, 0

lui $s3, 0xffff

lui $s4, 100

ori $s0, 100

ori $s1, 1

# test jal, jr, nop

loop\_jal\_1:

addu $s2, $s2, $s0

nop

jal loop\_jal\_2 # 测试jal函数调用

nop

jal loop\_jal\_4

loop\_jal\_2:

nop

addu $s2, $s2, $s0

nop

sw $ra, 0($s0)

jal loop\_jal\_3 # 测试jal递归过程

lw $ra, 0($s0)

jr $ra

loop\_jal\_3:

addu $s5, $0, $s3

nop

jr $ra

addu $s6, $s5, $0 # 测试jr指令是否正确执行

end:

addu $s4, $s4, $s1

jr $ra

loop\_jal\_4:

addu $s4, $s4, $s1

jal end # 测试jal是否可以向前跳转

nop

addu $s5, $s5, 1

机器代码：

35080064

36100064

363100c8

35290000

354a0000

3c01ffff

3421ffff

00010025

00082021

01008823

01284821

11100001

356b0064

01485021

3c010000

34210064

01014021

1111fff8

3c01ffff

3421ffff

01816025

35ad0ffc

ac0c0000

ac0c0004

adacfffc

adac0000

8c0e0000

8c0f0004

8db8fffc

8db90000

3c120000

3c13ffff

3c140064

36100064

36310001

02509021

00000000

0c000c28

00000000

0c000c35

00000000

02509021

00000000

ae1f0000

0c000c2f

8e1f0000

03e00008

0013a821

00000000

03e00008

02a0b021

0291a021

03e00008

0291a021

0c000c33

00000000

3c010000

34210001

02a1a821

预期结果：

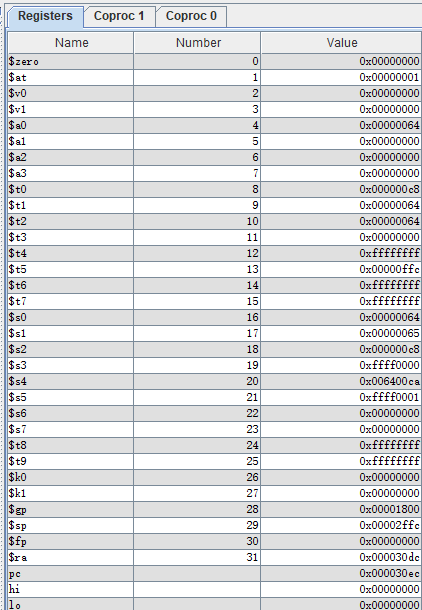


图11 正确结果

1. 思考题
2. 根据你的理解，在下面给出的DM的输入示例中，地址信号addr位数为什么是[11:2]而不是[9:0]？这个addr信号又是从哪里来的？



由于我们设计的MIPS CPU是32位的而且我们的DM是按照字节寻址的，而一个字是32bit，也就是4个字节，所以最后两位一般情况下都为0，所以可以忽略，直接去取[11:2]。

1. 在相应的部件中，reset的优先级比其他控制信号（不包括clk信号）都要高，且相应的设计都是同步复位。清零信号reset是针对哪些部件进行清零复位操作？这些部件为什么需要清零？

需要清零复位的有如下部件：GRF中所有寄存器，DM中的内存，以及对PC值恢复至初始值（0x00003000）。由于reset是要恢复至初始状态，所以要将寄存器和DM都置零。由于GRF、DM以及PC中保存着之前的程序保留下来的数据，所以在reset要执行新的程序之前需要清零来进行下一步。此外之后由于指令存储时相当于用的是ROM，不可以随便修改。

1. 列举出用Verilog语言设计控制器的几种编码方式（至少三种），并给出代码示例。

第一种：用if形式进行编码

代码示例:

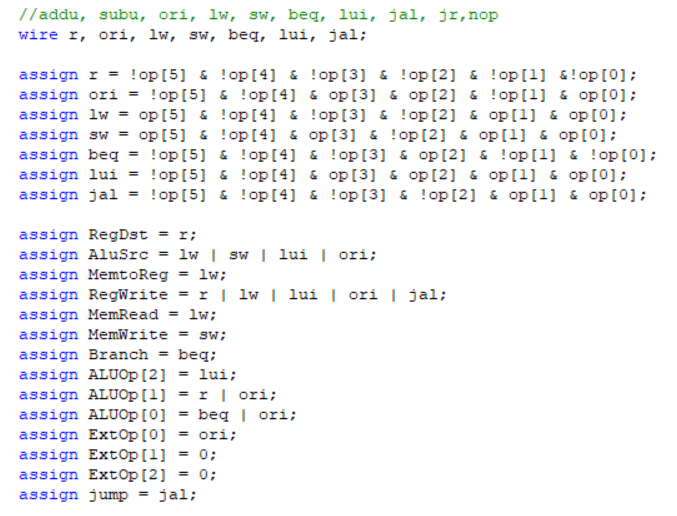
if(op == 6’b000000 && fun == 6’b100001) begin// instruction is addu

RegDst = 1;

…

end

第二种：用真值表方式进行编码



第三种：用宏定义方式进行编码

代码示例：

`define ori 6’001101

…

case(op)

ori:

…

endcase

1. 根据你所列举的编码方式，说明他们的优缺点。

if\_else语句和宏定义编码方式都非常直观，符合思维逻辑的。同时他们所需要的代码量大，因为每一个情况分支都对应着多种不同的信号输出。宏定义编码方式相比于if\_else语句，代码量更少，代码外观更加美观。用真值表assign赋值编码方式代码量少，容易理解。

1. C语言是一种弱类型程序设计语言。C语言中不对计算结果溢出进行处理，这意味着C语言要求程序员必须很清楚计算结果是否会导致溢出。因此，如果仅仅支持C语言，MIPS指令的所有计算指令均可以忽略溢出。 请说明为什么在忽略溢出的前提下，addi与addiu是等价的，add与addu是等价的。提示：阅读《MIPS32® Architecture For Programmers Volume II: The MIPS32® Instruction Set》中相关指令的Operation部分 。

实际上addi和addiu的运行过程很相似，都是将寄存器值与符号拓展后的立即数相加，并将所得结果31-0位赋值给另一个寄存器。二者的区别是addi指令若遇到相加后溢出情况会报错，而addiu则不会。所以对于一个不考虑溢出情况的语言，addi和addiu是一样的。add与addu同理。

1. 根据自己的设计说明单周期处理器的优缺点。

由于所有部件都连接的同一个clk信号，单周期处理器会在一个时钟周期内完成所有的工作，即从指令取出，到得到结果。所有指令时钟周期不用规定，肯定相同。单周期方式控制简单。但是有些指令时钟周期过长（lw），导致所有指令平均周期时间长。

1. 简要说明jal、jr和堆栈的关系。

jal是跳转指令，多用于函数调用。每次使用jal后，都会将此时的PC+4保存进入$ra中。而jr是jal的配套指令，多用于函数返回。由于调用函数时情况复杂，可能使用递归过程，故每次调用jal进入函数之后都需要强保护ra，即用堆栈保存ra的值，退出之前将ra的值恢复。堆栈是配套jal和jr使用的，用于存储每一次的返回地址。

指令真值表

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令 | Opcode[5:0] | Funct[5:0] | ThirdIn[4:0] | ALUSrc | MemtoReg | RegWrite | MemWrite | Branch | ExtOp | ALUOp | RegDst |  |
| addu | 000000 | 100001 | 无 | 00 | 0 | 1 | 0 | 000 | X | 0000(Add) | 01 |  |
| subu | 000000 | 100011 | 无 | 00 | 0 | 1 | 0 | 000 | X | 0001(Sub) | 01 |  |
| ori | 001101 | 无 | 无 | 01 | 0 | 1 | 0 | 000 | 00 | 0010(Or) | 00 |  |
| lw | 100011 | 无 | 无 | 01 | 1 | 1 | 0 | 000 | 10 | 0000(Add) | 00 |  |
| sw | 101011 | 无 | 无 | 01 | X | 0 | 1 | 000 | 10 | 0000(Add) | X |  |
| beq | 000100 | 无 | 无 | 00 | X | 0 | 0 | 001 | X | 0001(Sub) | X |  |
| lui | 001111 | 无 | 无 | 01 | X | 1 | 0 | 000 | 01 | 0000(Add) | 00 |  |
| nop | 000000 | 无 | 无 | X | X | 0 | 0 | 000 | X | X | X |  |
| 以上为课下教程要求的指令，以下为另外添加的指令： | | | | | | | | | | |  |  |
| j | 000010 | 无 | 无 | X | X | 0 | 0 | 000 | X | X |  |  |
| jal | 000011 | 无 | 无 | X | 0 | 1 | 0 | 000 | X | X |  |  |
| jalr | 000000 | 001001 | 无 | X | 0 | 1 | 0 | 000 | X | X |  |  |
| jr | 000000 | 001000 | 无 | X | X | 0 | 0 | 000 | X | X |  |  |
| 以上四条为J型指令，需要在Controller添加Jump信号，若为J型指令则为1，其余指令为0 | | | | | | | | | | |  |  |
| beq | 000100 | 无 | X | 00 | X | 0 | 0 | 001 | X | 0001(Sub) |  |  |
| bne | 000101 | 无 | X | 00 | X | 0 | 0 | 010 | X | 0001(Sub) |  |  |
| bgez | 000001 | 无 | 00001 | 10 | X | 0 | 0 | 011 | X | 0000(Add) |  |  |
| bgtz | 000111 | 无 | 00000 | 10 | X | 0 | 0 | 100 | X | 0000(Add) |  |  |
| blez | 000110 | 无 | 00000 | 10 | X | 0 | 0 | 101 | X | 0000(Add) |  |  |
| bltz | 000001 | 无 | 00000 | 10 | X | 0 | 0 | 110 | X | 0000(Add) |  |  |
| 以上六条为B指令，包括原有的beq和另外添加的bne、bgez、bgtz、blez、bltz(Bin) | | | | | | | | | | |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |