计算机组成课设P3设计文档

CPU部件设计部分

一、IFU（取指令单元）：内部包括 PC（程序计数器）、IM(指令存储器)及相关逻辑。

教程要求：

* PC 用寄存器实现，应具有复位功能。
* 起始地址：0x00000000。
* IM用 ROM 实现，容量为 32bit \* 32。
* 因 IM 实际地址宽度仅为 5 位，故需要使用恰当的方法将 PC 中储存的地址同 IM 联系起来。

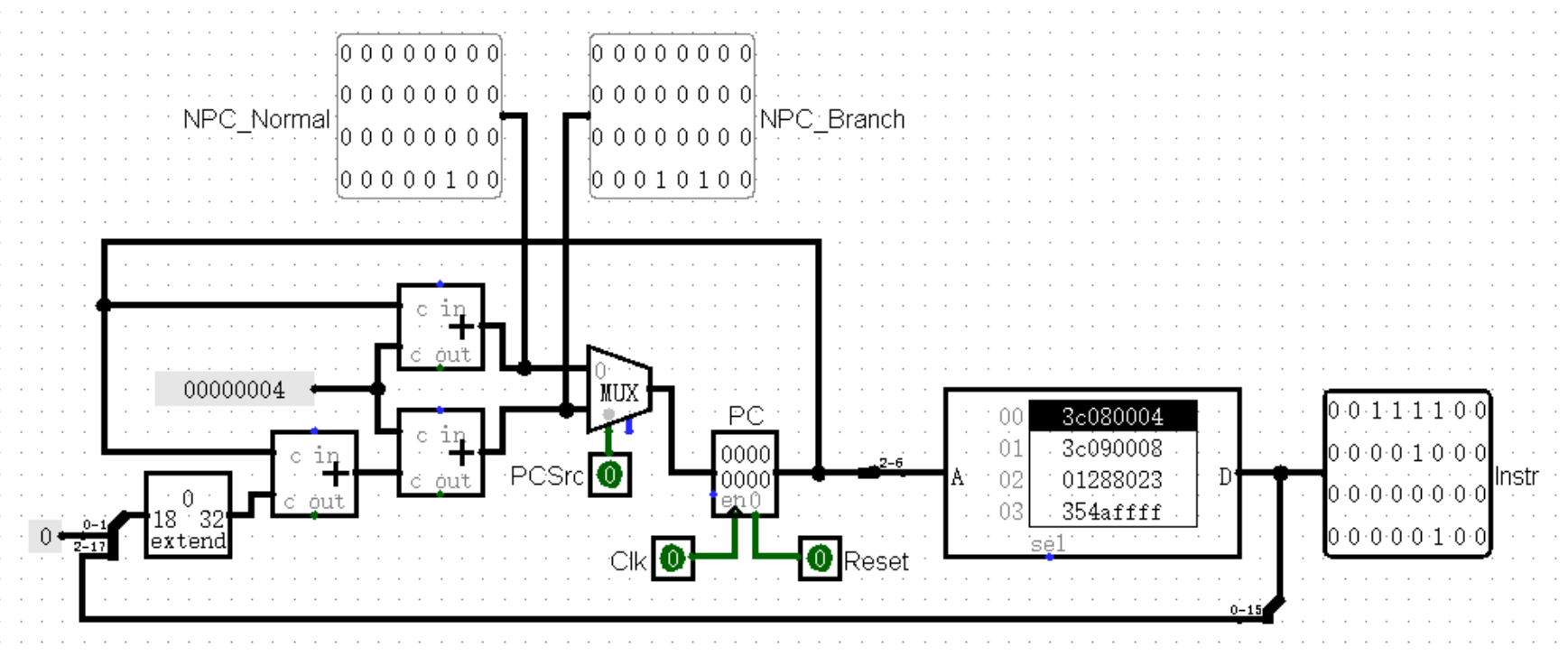


图1 IFU设计图

表1 IFU端口说明表

|  |  |  |
| --- | --- | --- |
| 信号 | 方向 | 功能说明 |
| PCSrc | I | 判断指令地址是否跳转 |
| Clk | I | 时钟信号 |
| Reset | I | 复位信号 |
| Instr[31:0] | O | 输出的32位指令 |

表2 IFU功能说明表

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名称 | 功能描述 |
| 1 | 取指令 | 根据PC从IM中取出指令 |
| 2 | 计算下一条指令 | 若PCSrc=1，则PC = PC + 4 + sign\_extend (offset || 0^2)；反之，则PC = PC + 4 |
| 3 | 复位 | 将PC复位成0x00000000 |

二、GRF（通用寄存器组，也称为寄存器文件、寄存器堆）

* 用具有写使能的寄存器实现，寄存器总数为 32 个。
* 0 号寄存器的值始终保持为 0。其他寄存器初始值均为 0，无需专门设置。

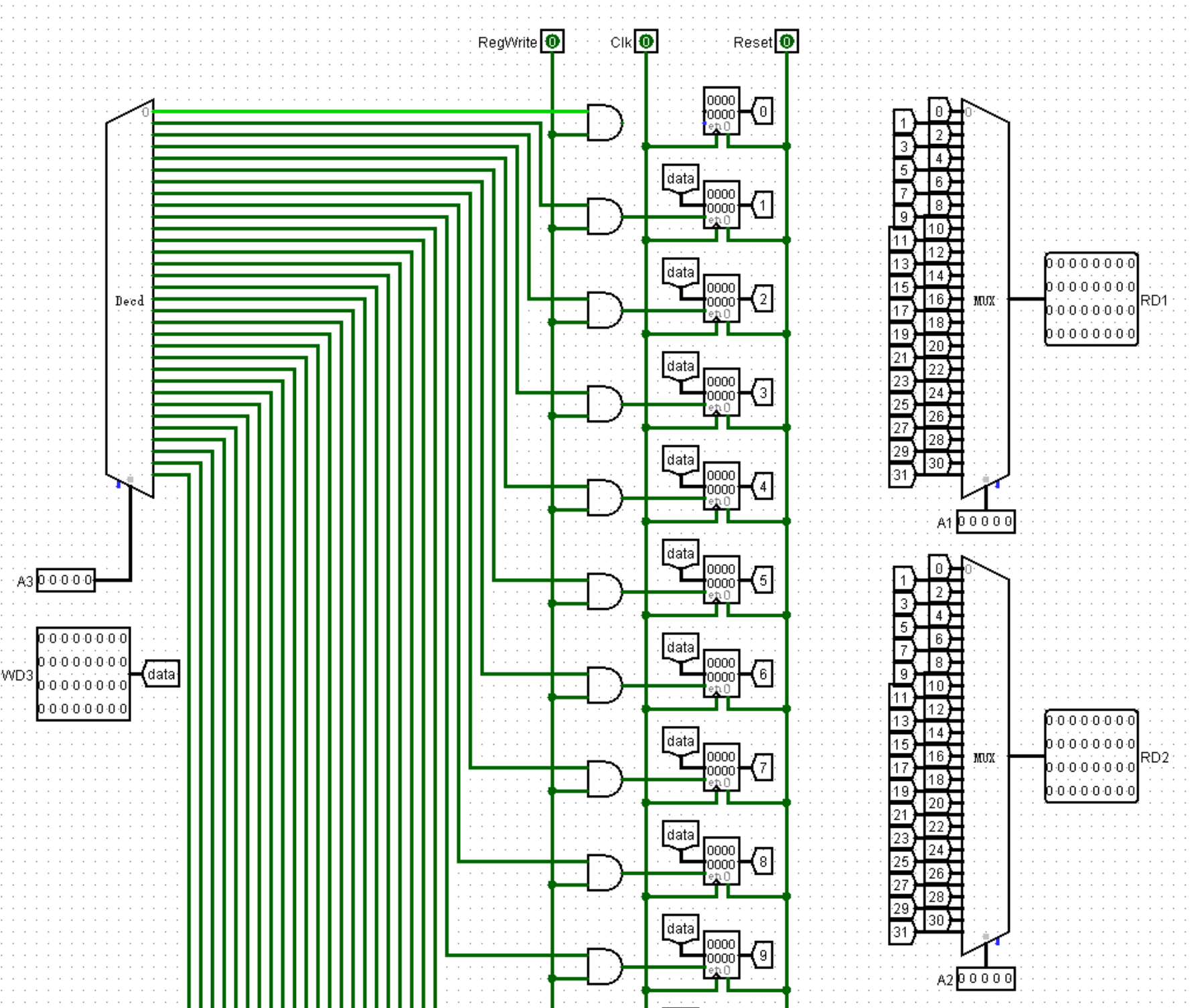


图2 GRF设计图（部分）

表3 GRF端口说明表

|  |  |  |
| --- | --- | --- |
| 信号 | 方向 | 功能说明 |
| RegWrite | I | 寄存器写入信号 |
| Clk | I | 时钟信号 |
| Reset | I | 复位信号 |
| A1[4:0] | I | 读寄存器地址1 |
| A2[4:0] | I | 读寄存器地址2 |
| A3[4:0] | I | 写寄存器地址 |
| WD[31:0] | I | 32位数据输入 |
| RD1[31:0] | O | 32位数据输出1 |
| RD2[31:0] | O | 32位数据输出2 |

表4 GRF功能说明表

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名称 | 功能描述 |
| 1 | 读取寄存器数据 | 根据输入的5位A1、A2信号从相应编号的寄存器中读取32位数据并通过RD1、RD2输出 |
| 2 | 写入寄存器数据 | 根据RegWrite信号决定是否将32位WD数据写入5位的A3信号对应的寄存器 |
| 3 | 复位 | 将32个寄存器全部复位成0x00000000 |

三、ALU（算术逻辑单元）

* 提供 32 位加、减、或运算及大小比较功能。
* 可以不支持溢出（不检测溢出）。

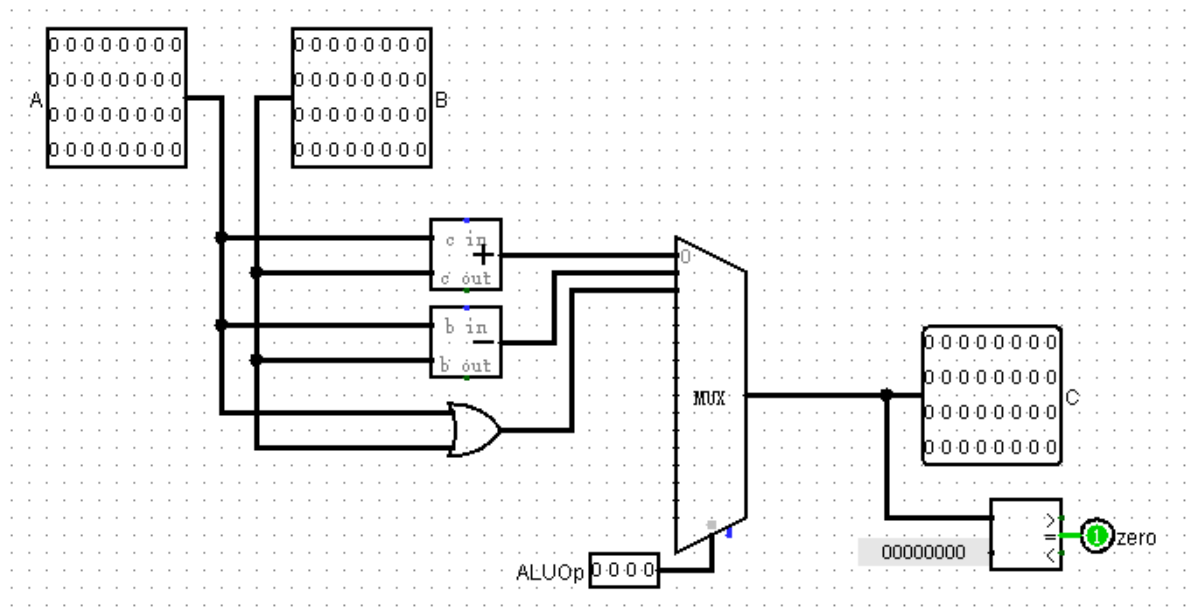


图3 ALU设计图

表5 ALU端口说明表

|  |  |  |
| --- | --- | --- |
| 信号 | 方向 | 功能说明 |
| A[31:0] | I | 32位输入数据1 |
| B[31:0] | I | 32位输入数据2 |
| ALUOp[3:0] | I | 4位运算选择器 |
| C[31:0] | O | 32位输出数据，计算结果 |
| Zero | O | 1位信号输出计算结果是否为0，用于计算PCSrc |

表6 ALU功能说明表

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名称 | 功能描述 |
| 1 | 数据运算 | 根据输入的4位ALUOp信号将输入的32位信号A、B进行相应的计算并从32位C信号输出：0000进行加法，0001进行减法运算，0010进行或运算 |
| 2 | 判断是否运算结果是否为零 | 将A、B数据相减后判断结果是否为0并从1位信号Zero输出判断结果 |

四、DM（数据存储器）

* 使用 RAM 实现，容量为 32bit \* 32。
* 起始地址：0x00000000。
* RAM 应使用双端口模式，即设置 RAM 的 Data Interface 属性为 Separate load and store ports。

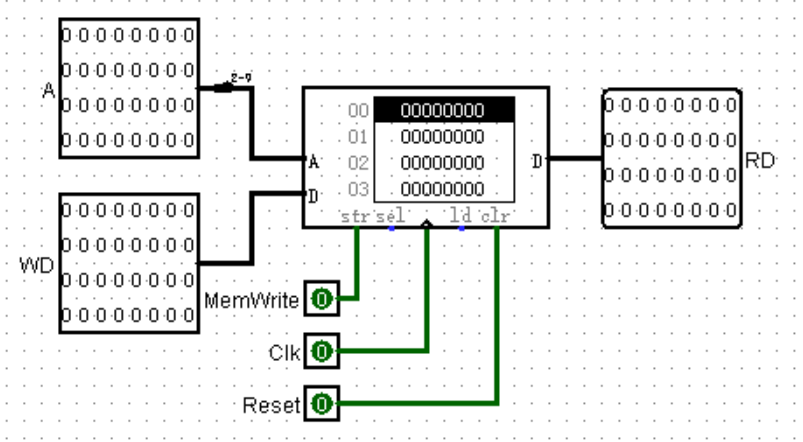


图4 DM设计图

表7 DM端口说明表

|  |  |  |
| --- | --- | --- |
| 信号 | 方向 | 功能说明 |
| MemWrite | I | 内存写入信号 |
| Clk | I | 时钟信号 |
| Reset | I | 复位信号 |
| A[31:0] | I | 32位指令输入信号 |
| WD[31:0] | I | 32位写入数据 |
| RD[31:0] | O | 31位数据输出 |

表8 DM功能说明表

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名称 | 功能描述 |
| 1 | 读取数据 | 根据A输入的地址读取数据并从RD输出 |
| 2 | 写入数据 | 若MemWrite = 1，将WD数据写入A对应的地址 |
| 3 | 复位 | 将内存中的数据全部复位归零 |

五、EXT

* 可以使用 logisim 内置的 Bit Extender。

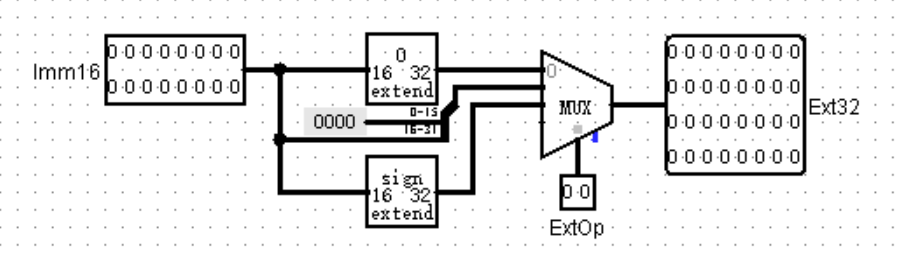


图5 EXT设计图

表9 EXT端口说明表

|  |  |  |
| --- | --- | --- |
| 信号 | 方向 | 功能说明 |
| ExtOp[1:0] | I | 2位扩展方式选择信号 |
| Imm16 | I | 待扩展的16位数据 |
| Ext32 | O | 扩展后的32位数据 |

表10 DM功能说明表

|  |  |  |
| --- | --- | --- |
| 序号 | 功能名称 | 功能描述 |
| 1 | 扩展数据 | 根据2位ExtOp信号选择的扩展方式，对输入的16位Imm16数据扩展成32位并从Ext32输出：00表示高位补0，01表示低位补0，10表示符号扩展 |

六、Controller（控制器）

* 使用与或门阵列构造控制信号，具体方法见后文叙述。

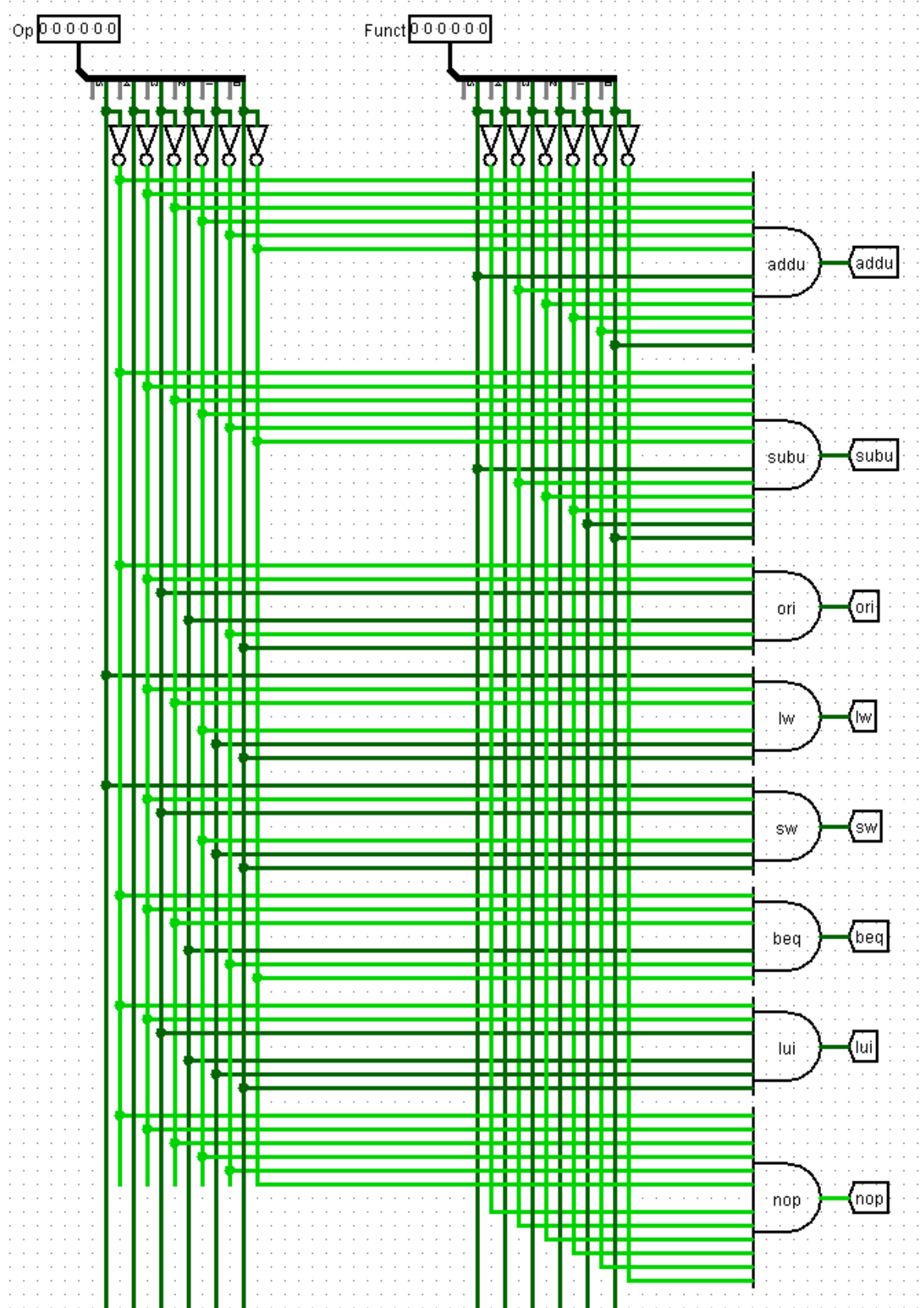


图6 Controller和逻辑设计图

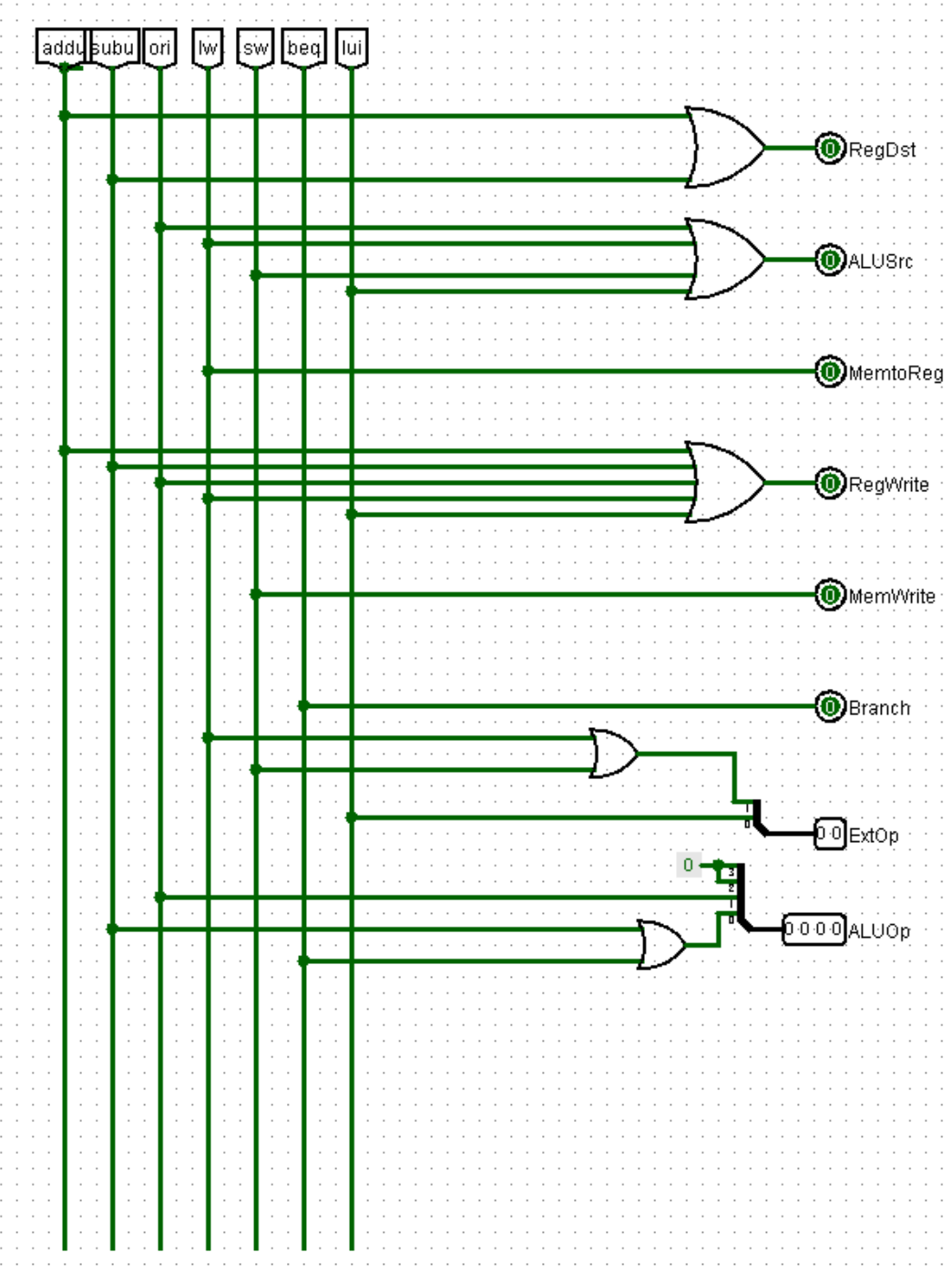


图7 Controller或逻辑设计图

表11 Controller端口说明表

|  |  |  |
| --- | --- | --- |
| 信号 | 方向 | 功能描述 |
| Opcode[5:0] | I | 指令的Opcode字段 |
| Funct[5:0] | I | 指令的Function字段 |
| RegDst[1:0] | O | 寄存器写地址选择信号 |
| ALUSrc[1:0] | O | ALU第二个操作数选择信号 |
| MemtoReg | O | 内存写入寄存器控制信号 |
| RegWrite | O | 寄存器写入控制信号 |
| MemWrite | O | 内存写入控制信号 |
| Branch[2:0] | O | 跳转指令信号 |
| ExtOp[1:0] | O | 扩展方式选择信号 |
| ALUOp[3:0] | O | 运算方式选择信号 |
| 以上为课下教程要求的信号，以下为另外添加的信号： | | |
| J | O | J型指令判断信号(1表示是J型指令，反之为0) |
| Jump[1:0] | O | J型指令信号(j 00,jal 01,jalr 10,jr 11) |
|  | O |  |

表12 Controller功能说明表兼指令信号输出真值表

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令 | Opcode | Funct | RegDst | ALUSrc | MemtoReg | RegWrite | MemWrite | Branch | ExtOp | ALUOp |
| addu | 000000 | 100001 | 01 | 00 | 0 | 1 | 0 | 000 | X | 0000(Add) |
| subu | 000000 | 100011 | 01 | 00 | 0 | 1 | 0 | 000 | X | 0001(Sub) |
| ori | 001101 | 无 | 00 | 01 | 0 | 1 | 0 | 000 | 00 | 0010(Or) |
| lw | 100011 | 无 | 00 | 01 | 1 | 1 | 0 | 000 | 10 | 0000(Add) |
| sw | 101011 | 无 | X | 01 | X | 0 | 1 | 000 | 10 | 0000(Add) |
| beq | 000100 | 无 | X | 00 | X | 0 | 0 | 001 | X | 0001(Sub) |
| lui | 001111 | 无 | 00 | 01 | X | 1 | 0 | 000 | 01 | 0000(Add) |
| nop | 000000 | 无 | X | X | X | 0 | 0 | 000 | 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 | X | X | X | 0 | 0 | 000 | X | X |
| jal | 000011 | 无 | 10 | X | 0 | 1 | 0 | 000 | X | X |
| jalr | 000000 | 001001 | 01 | X | 0 | 1 | 0 | 000 | X | X |
| jr | 000000 | 001000 | X | 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 | 00001 | X | 10 | X | 0 | 0 | 011 | X | 0000(Add) |
| bgtz | 000111 | 00000 | X | 10 | X | 0 | 0 | 100 | X | 0000(Add) |
| blez | 000110 | 00000 | X | 10 | X | 0 | 0 | 101 | X | 0000(Add) |
| bltz | 000001 | 00000 | X | 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) | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |

七、顶层设计

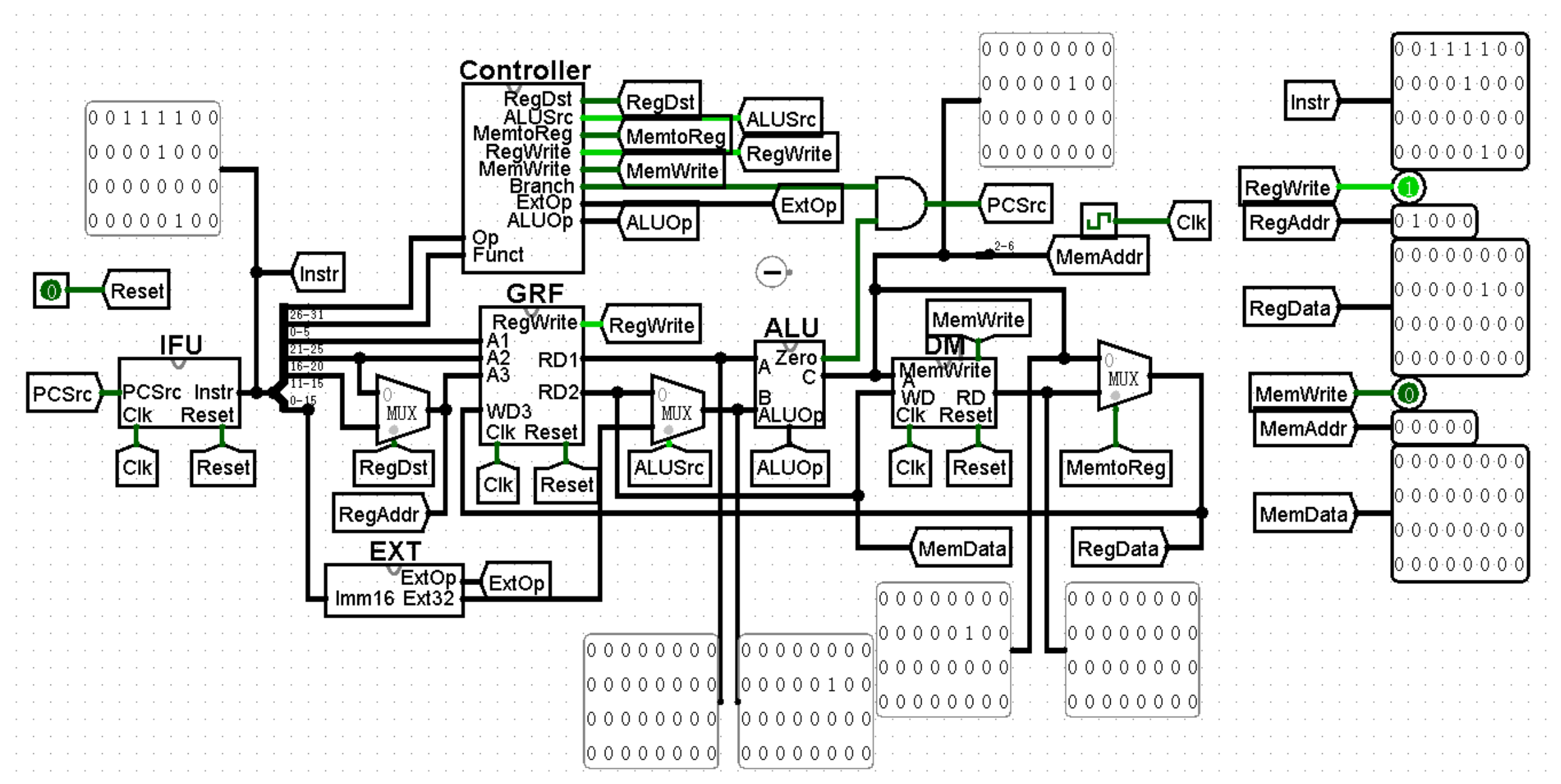


图8 顶层设计图

思考题部分

L0.T2

1. 若PC（程序计数器）位数为30位，试分析其与32位PC的优劣。

优势：由于指令的地址每次是递增4的，所以30位的PC可以将最后两位定为00，每次计算NPC时只需直接加1即可，同时在执行beq、bne等跳转指令时NPC = PC + 4 + sign\_extend (offset || 0^2) 时同样也只需直接加上offset即可，无需左移两位。

劣势：由于CPU的指令均为32位的，所以每次计算出的30位指令需要扩展成32位的，不如32位指令来的灵活。其次，在jr指令至多可以跳转2^32条指令，即整个指令空间，如果PC只有30位则无法满足。

2．现在我们的模块中 IM使用ROM， DM使用RAM， GRF使用寄存器，这种做法合理吗？ 请给出分析，若有改进意见也请一并给出。

合理。寄存器数量固定为32个，同时寄存器有着速度最快但成本较高的特点，而且可以同时操作多个寄存器，比较符合CPU中转站的角色特点；ROM是只读存储器，对于我们写入的指令在CPU运行过程中并不需要修改，很适合用来保存指令；RAM是读写存储器，不仅容量很大，还满足了既可以灵活写入又可以灵活读取的需求，适合用来保存数据。

改进意见：可以像MARS中那样，使用一个RAM存储器，将指令和数据保存在不同的地址，这样可以节省一部分部件。

L0.T3

1．结合上文给出的样例真值表，给出RegDst， ALUSrc， MemtoReg，RegWrite, nPC\_Sel, ExtOp与op和func有关的布尔表达式（表达式中只能使用“与、或、非”3 种基本逻辑运算。）

2．充分利用真值表中的 X 可以将以上控制信号化简为最简单的表达式， 请给出化简后的形式。

第一题和第二题结合如下：

RegDst = (addu | subu) & ~ori & ~lw & (sw | ~sw) & (beq | ~beq)

= (addu | subu)

= (~op[5] & ~op[4] & ~op[3] & ~op[2] & ~op[1] & ~op[0])

AluSrc = ~addu & ~subu & (ori | lw | sw) & ~beq

= ori | lw | sw

= (~op[5] & ~op[4] & op[3] & op[2] & ~op[1] & op[0]) +

(op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0])+

(op[5] & ~op[4] & op[3] & ~op[2] & op[1] & op[0])

MemtoReg = lw & ~addu & ~subu & ~ori & (sw | ~sw) & (beq | ~beq)

= lw

= (op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0])

RegWrite = (addu | subu | ori | lw) & ~sw & ~beq

= (addu | subu | ori | lw)

= (~op[5] & ~op[4] & ~op[3] & ~op[2] & ~op[1] & ~op[0]) +

(~op[5] & ~op[4] & op[3] & op[2] & ~op[1] & op[0]) +

(op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0])

npc\_sel = ~addu & ~subu & ~ori & ~lw & ~sw & beq

= beq

= (~op[5] & ~op[4] & ~op[3] & op[2] & ~op[1] & ~op[0])

ExtOp = (addu | ~addu) & (subu | ~subu) & ~ori & lw & sw & (beq | ~beq)

= lw & sw

=(op[5] & ~op[4] & ~op[3] & ~op[2] & op[1] & op[0])+

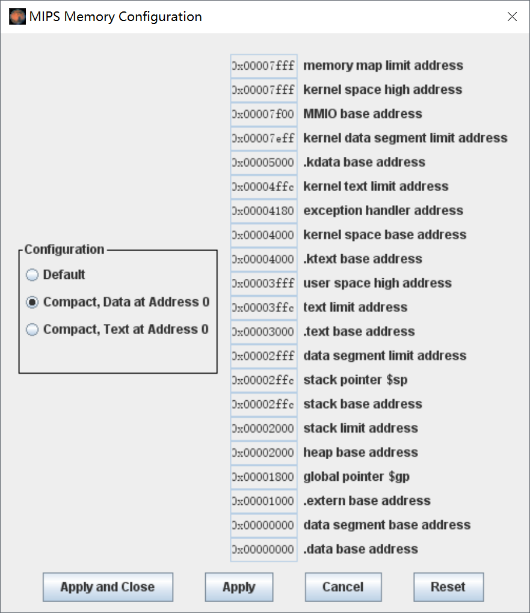
(op[5] & ~op[4] & op[3] & ~op[2] & op[1] & op[0])

3．事实上，实现nop空指令，我们并不需要将它加入控制信号真值表，为什么？请给出你的理由。

因为执行nop空指令时CPU并不需要进行任何操作，对寄存器、内存都不会有任何的影响，而NPC=PC+4跟控制信号没有关系，程序还能照常地运行，所以我们不需要将其加入控制信号真值表。

L0.T4

1．前文提到，“可能需要手工修改指令码中的数据偏移”，但实际上只需再增加一个 DM片选信号,就可以解决这个问题。请阅读相关资料并设计一个 DM 改造方案使得无需手工修改数据偏移。



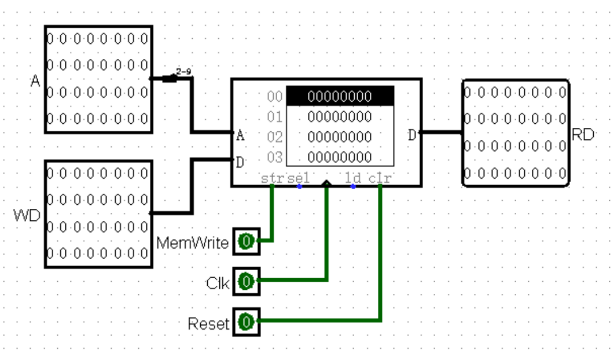


图10 改造方案

图9 MARS设计图

片选是指在地址选择时，首先要选片，只有当片选信号有效时，此片所连的地址线才有效。由图中的MARS设置可知：.data是从0x00000000开始的，.text是从0x00003000开始的。比较粗暴的方法是像右图那样将指令的[9:2]该8位取出作为地址，这样就可严格控制在有效区间内；或者将指令的[13:12]（以图中的设置的为例）取出与2’b11进行比较，比较结果就是对应的片选信号。

2．除了编写程序进行测试外，还有一种验证CPU设计正确性的办法——形式验证。 形式验证的含义是根据某个或某些形式规范或属性，使用数学的方法证明其正确性或非正确性。请搜索“形式验证（Formal Verification)"了解相关内容后，简要阐述相比与测试，形式验证的优劣。

优势：测试验证由于是以仿真为基础的，具有非穷尽的固有特性，因此边际情形无法被检测到，然而形式验证就可以克服测试验证的这个缺点，可以完整覆盖全部测试空间；形式验证使用数学推理来验证设计意图在实现中是否得以贯彻，没有必要表明如何激励设计或创建多种条件来实现较高的可观察性，也就是说测试者不必考虑如何获得测试验证需要输入的参数；形式验证可以进行系统级到门级的验证，而且验证时间短，有利于尽快发现设计中的问题来缩减设计周期。

缺点：形式验证不能发现代码中时序错误，不能对于动态行为进行验证；同时形式验证不能有效地验证电路的性能，比如电路的时延和功耗等。

测试数据部分

汇编代码

lui $t0, 0x0004 #测试lui：向寄存器t0高16位写入0x0004

lui $t1, 8 #测试lui：向寄存器t1高16位写入0x0008

subu $s0, $t1, $t0 #测试subu：向寄存器s0写入t1与t0的差

ori $t2, 0xffff #测试ori：向寄存器t2写入0xffff or $t2(0xffff)

ori $t3, $0, 0x00000004 #测试ori：向寄存器t3写入0 or 0x00000004

ori $t9, $0, 0x0000 #测试ori：向寄存器t9写入0 or 0x0000

sw $t2, 8($t3) #测试sw：将寄存器t2存入存储器地址8+$t3

sw $t0, 4($t3) #测试sw：将寄存器t0存入存储器地址4+$t3

sw $t1, 0($t3) #测试sw：将寄存器t1存入存储器地址0+$t3

sw $t2, 0 #测试sw：将寄存器t2存入存储器地址0

lw $t4, 4($t3) #测试lw：将地址4+$t3的值存入寄存器t4

lw $t5, 0 #测试lw：将地址0的值存入寄存器t5

nop #测试nop：空指令

loop:

addu $t0, $t0, $t1 #测试addu：向寄存器t0写入t1与t0的和

addu $t9, $t9, 0x10000001 #测试addu：向t9写入其与0x10000001的和

nop #测试nop：空指令

beq $t0, 0x000c0000, loop #测试beq：若t0 == 0x000c0000则跳转loop

subu $t7, $t0, $t1 #测试subu：向寄存器t7写入t0与t1的差

lw $t8, 8($t3) #测试lw：将地址8+$t3的值存入寄存器t8

机器代码：

3c080004 3c090008 01288023 354affff

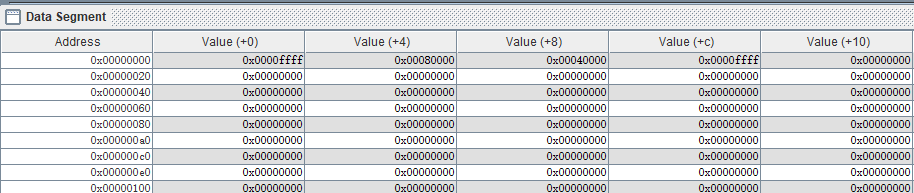
340b0004 34190000 ad6a0008 ad680004

ad690000 ac0a0000 8d6c0004 8c0d0000

00000000 01094021 3c011000 34210001

0321c821 00000000 3c01000c 34210000

1028fff8 01097823 8d780008

预期结果：

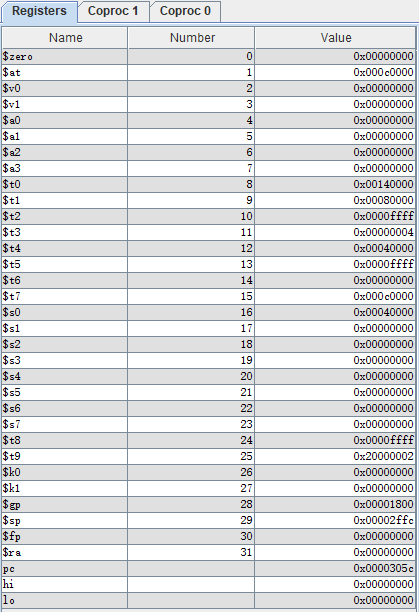
图11 内存预期结果图

图12 寄存器预期结果图

在测试ori指令时，不能采用ori $t0, 32bit imm的形式，因为汇编器会将ori指令转化为or指令与其他指令。