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。

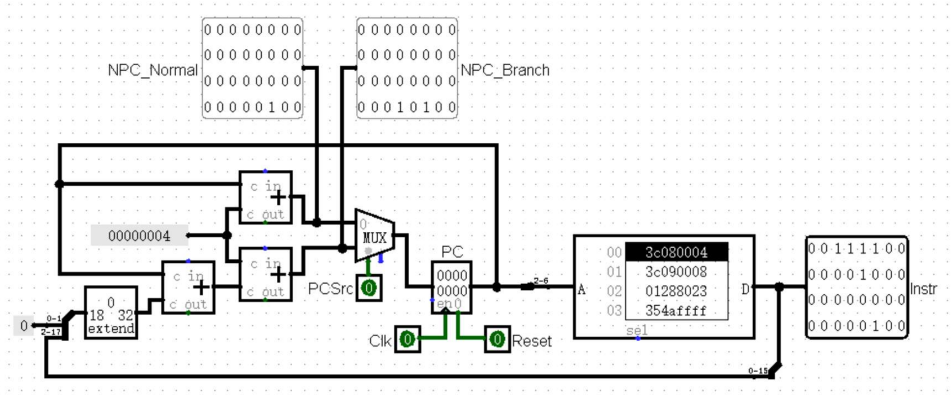

图 1 IFU 设计图

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。

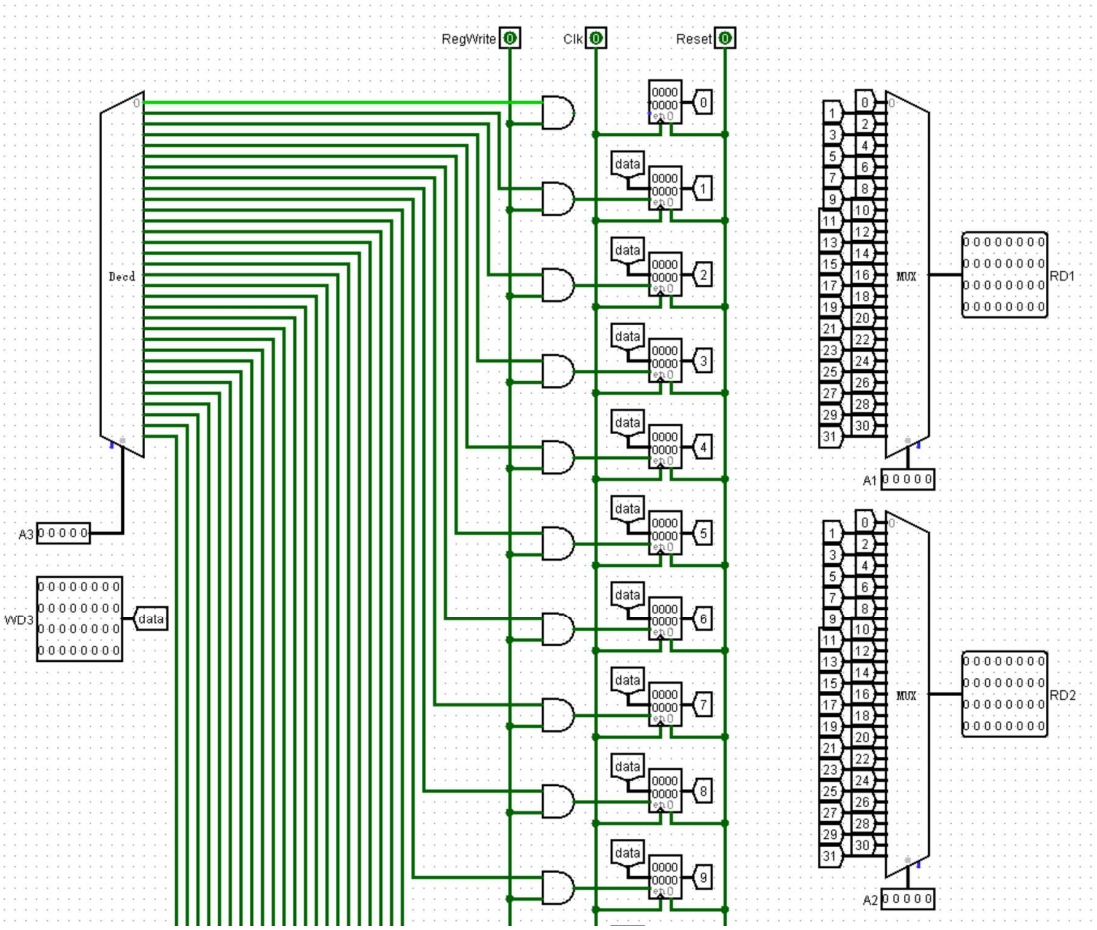

图 2 GRF 设计图（部分）

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）。

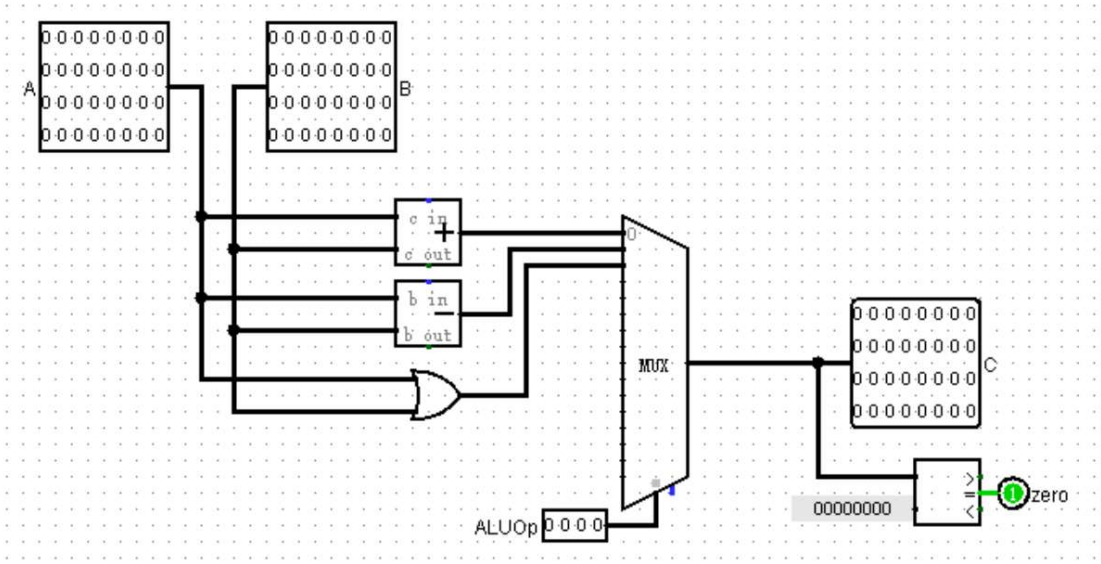

图 3 ALU 设计图

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。

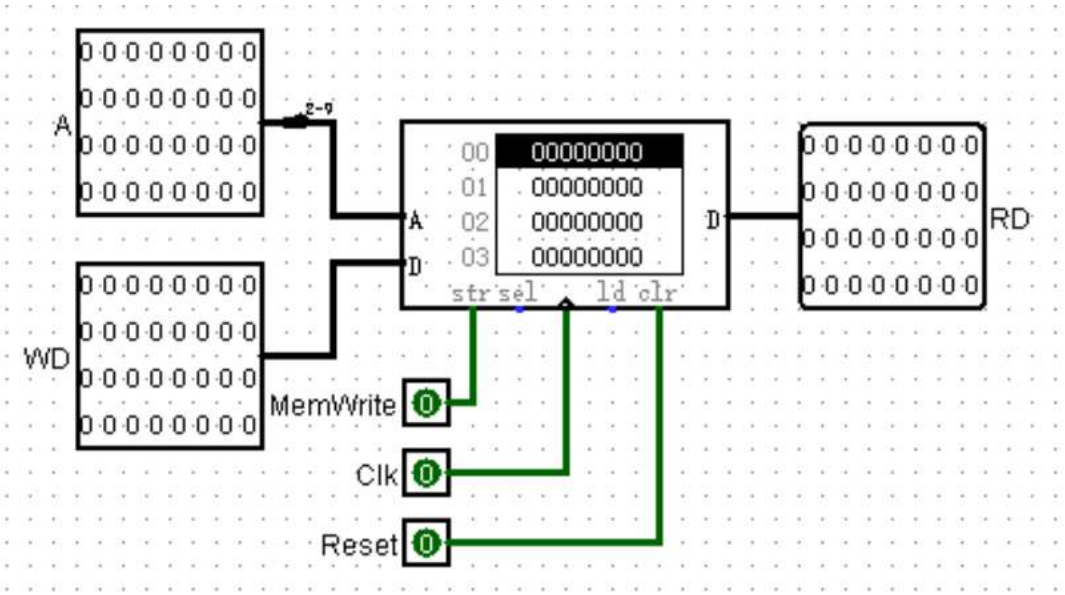

图 4 DM 设计图

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。

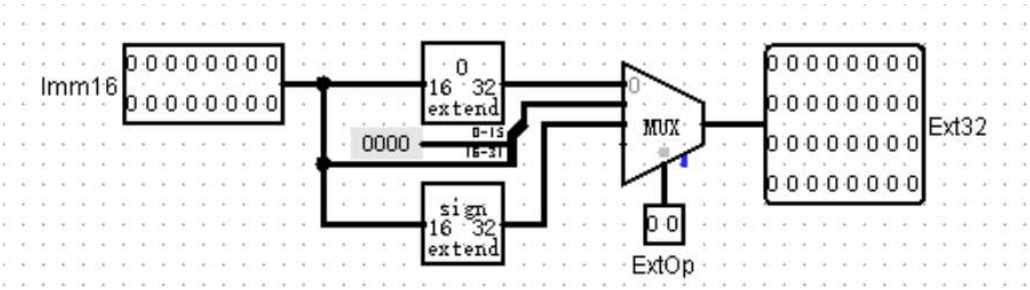

图 5 EXT 设计图

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。

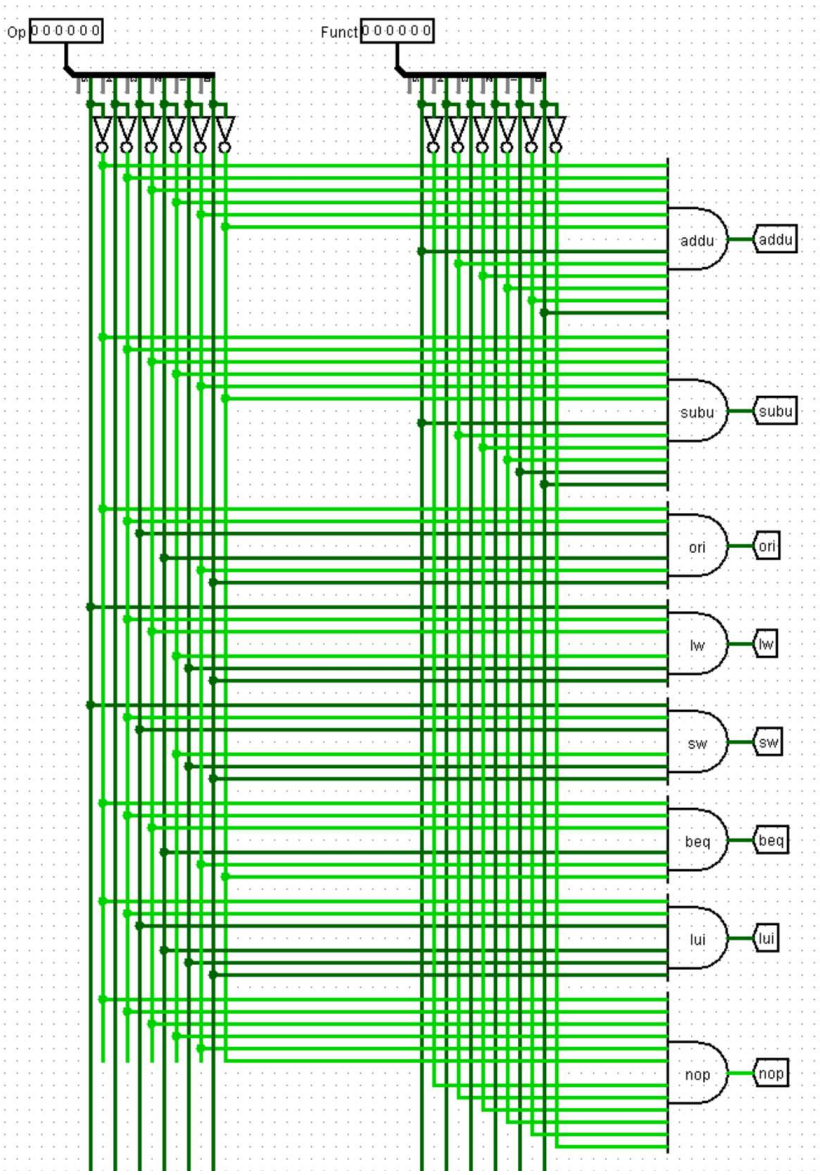

图 6 Controller 和逻辑设计图

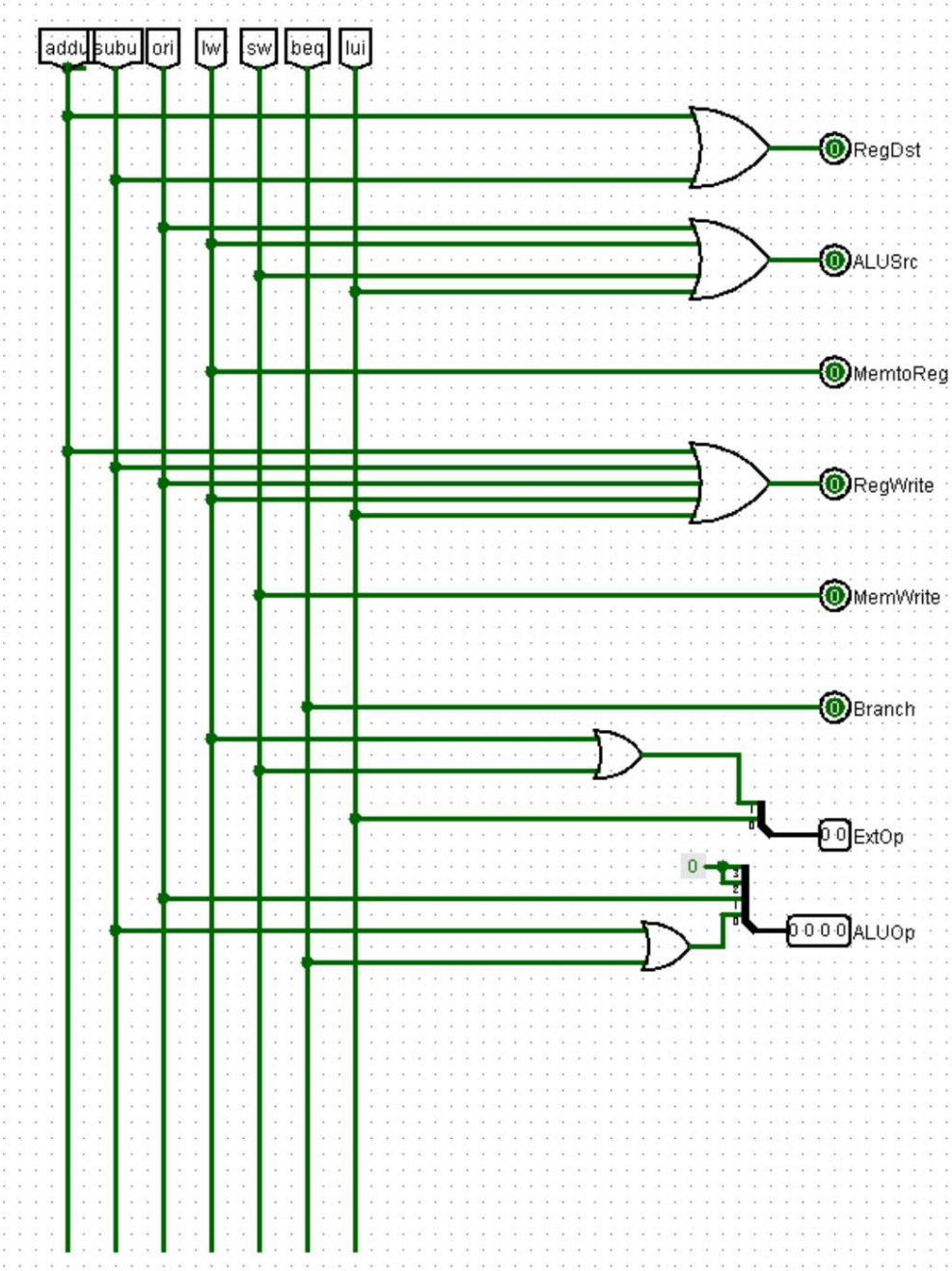

图 7 Controller 或逻辑设计图

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	O	寄存器写地址选择信号
ALUSrc	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	0	寄存器写入控制信号
MemWrite	0	内存写入控制信号
Branch	0	跳转指令信号
ExtOp[1:0]	0	扩展方式选择信号
ALUOp[3:0]	0	运算方式选择信号

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	1	0	0	1	0	0	X	0000(Add)
subu	000000	100011	1	0	0	1	0	0	X	0001(Sub)
ori	001101	无	0	1	0	1	0	0	00	0010(Or)
lw	100011	无	0	1	1	1	0	0	10	0000(Add)
sw	101011	无	X	1	X	0	1	0	10	0000(Add)
beq	000100	无	X	0	X	0	0	1	X	0001(Sub)
lui	001111	无	0	1	X	1	0	0	01	0000(Add)
nop	000000	无	X	X	X	0	0	0	X	X
以上为课下教程要求的指令，以下为另外添加的指令：										

七、顶层设计

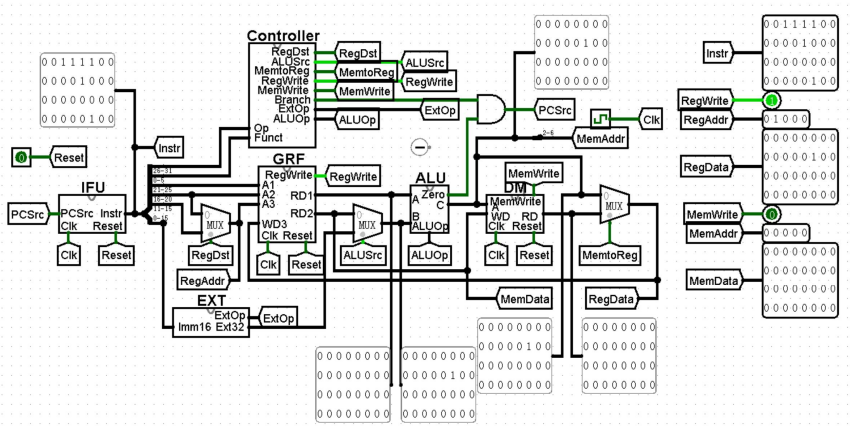

图 8 顶层设计图

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。

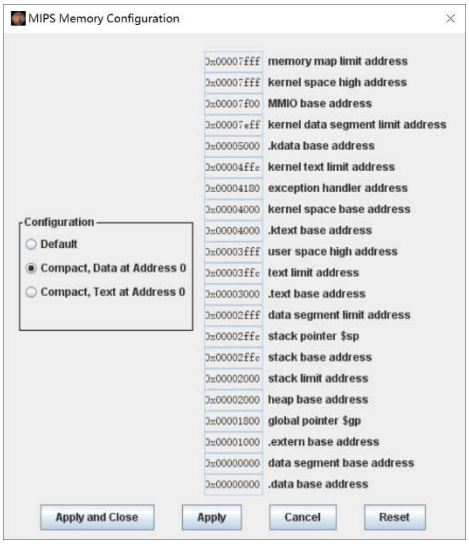

图 9 MARS 设计图

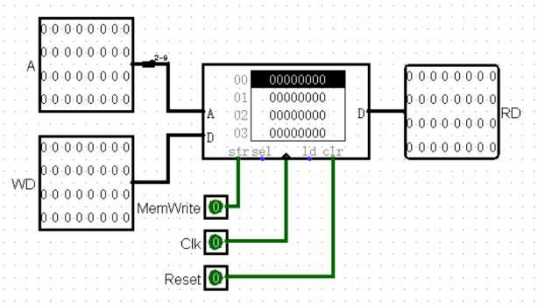

图 10 改造方案

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：

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)
0x00000000	0x0000ffff	0x00080000	0x00040000	0x0000ffff	0x00000000
0x00000020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x000000a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x000000c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x000000e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000100	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

图 11 内存预期结果图

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x000c0000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00140000
$t1	9	0x00080000
$t2	10	0x0000ffff
$t3	11	0x00000004
$t4	12	0x00040000
$t5	13	0x0000ffff
$t6	14	0x00000000
$t7	15	0x000c0000
$s0	16	0x00040000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x0000ffff
$t9	25	0x20000002
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x00001800
$sp	29	0x00002ffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x0000305c
hi		0x00000000
lo		0x00000000

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。