目录

[一、 指令格式设计 1](#_Toc169907265)

[二、 微操作的定义及节拍的划分 2](#_Toc169907266)

[三、 处理器结构设计 4](#_Toc169907267)

[（一） 程序计数器PC 4](#_Toc169907268)

[（二） 指令存储器IM 4](#_Toc169907269)

[（三） 数据存储器DM 6](#_Toc169907270)

[（四） 寄存器堆RegHeap 6](#_Toc169907271)

[（五） 算术逻辑单元ALU 7](#_Toc169907272)

[（六） 杂项 8](#_Toc169907273)

[四、 硬布线控制逻辑设计 8](#_Toc169907274)

[（一） 分离式设计 9](#_Toc169907275)

[（二） 一体式设计 12](#_Toc169907276)

[1． RegBranch 14](#_Toc169907277)

[2． NearBranch 14](#_Toc169907278)

[3． AbsBranch 14](#_Toc169907279)

[4． AluOp 14](#_Toc169907280)

[5． RegWrite 15](#_Toc169907281)

[6． MemRead 16](#_Toc169907282)

[7． MemWrite 16](#_Toc169907283)

[8． Shift 16](#_Toc169907284)

[9． Imm 17](#_Toc169907285)

# 指令格式设计

MIPS32指令集的指令字长均为32位，分为R型指令、I型指令与J型指令三种，具体指令格式见下表1.1。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 指令格式 | 二进制位范围 | | | | | |
| 31~26 | 25~21 | 20~16 | 15~11 | 10~6 | 5~0 |
| R型 | op(6) | rs(5) | rt(5) | rd(5) | shamt(5) | func(6) |
| I型 | op(6) | rs(5) | rt(5) | imm(16) | | |
| J型 | op(6) | addr(26) | | | | |

表1.1 MIPS32的指令格式

三种类型的指令均包含了op字段，但R型指令采用了扩展操作码技术，其6位op字段均为0，具体指令功能由低6位的func字段决定。

对于R型指令，rs、rt为源寄存器，rd为目的寄存器，func字段表明的执行的操作。shamt字段为移位位数（shift amount），仅用于移位指令。

对于I型指令，rs为源寄存器，rt为目的寄存器，imm为16位的立即数。I型指令是立即数指令，主要用于立即数的操作，典型指令有立即数运算，Load/Store访存，条件分支等。

对于J型指令，addr为地址字段。J型指令主要可实现大范围的无条件转移j，其目标地址由PC+4的高4位与addr左移2位后的值拼接得到。

本次设计采用的是简化版的MIPS32指令集，挑选部分典型R型、I型与J型指令进行实现，但也添加了少部分指令便于程序员编程。具体采用的指令集分别见表1.2，表1.3与表1.4。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 助记符 | 解释 | 含义 | op | rs | rt | rd | shamt | func |
| add | 加法 | rd ← rs + rt | 000000 | rs | rt | rd | - | 000001 |
| sub | 减法 | rd ← rs - rt | 000010 |
| and | 按位与 | rd ← rs & rt | 000011 |
| or | 按位或 | rd ← rs | rt | 000100 |
| xor | 按位异或 | rd ← rs ^ rt | 000101 |
| slt | 有符号小于时置位 | rd ← (rs < rt) ? 1 : 0 | 001001 |
| seq | 等于时置位 | rd ← (rs = rt) ? 1 : 0 | 001010 |
| sgt | 有符号大于时置位 | rd ← (rs > rt) ? 1 : 0 | 001011 |
| sll | 逻辑(算术)左移 | rd ← rt << shamt | - | shamt | 000110 |
| srl | 逻辑右移 | rd ← rt >>> shamt | 000111 |
| sra | 算术右移 | rd ← rt >> shamt | 001000 |
| jr | 跳转至寄存器 | PC ← rs | rs | - | - | - | 010000 |

表1.2 采用的R型指令（其中“-”意味着可为任意值）

|  |  |  |  |
| --- | --- | --- | --- |
| 助记符 | 解释 | 含义 | op |
| addi | 立即数加法 | rt ← rs + (sign-extend) imm | 000001 |
| subi | 立即数减法 | rt ← rs - (sign-extend) imm | 000010 |
| andi | 立即数按位与 | rt ← rs & (sign-extend) imm | 000011 |
| ori | 立即数按位或 | rt ← rs | (sign-extend) imm | 000100 |
| xori | 立即数按位异或 | rt ← rs ^ (sign-extend) imm | 000101 |
| slti | 有符号小于立即数时置位 | rt ← (rs < (sign-extend) imm) ? 1 : 0 | 001001 |
| seqi | 等于立即数时置位 | rt ← (rs = (sign-extend) imm) ? 1 : 0 | 001010 |
| sgti | 有符号大于立即数时置位 | rt ← (rs > (sign-extend) imm) ? 1 : 0 | 001011 |
| slli\* | 逻辑(算术)左移\* | rt ← rs << (sign-extend) imm | 000110 |
| srli\* | 逻辑右移\* | rt ← rs >>> (sign-extend) imm | 000111 |
| srai\* | 算术右移\* | rt ← rs >> (sign-extend) imm | 001000 |
| lw | 读内存单元 | rt ← mem[rs + (sign-extend) imm] | 010000 |
| sw | 写内存单元 | mem[rs + (sign-extend) imm] ← rt | 010001 |
| beq | 相等时跳转 | if (rs == rt) PC ← PC + 4 +  (sign-extend) imm << 2 | 100000 |

表1.3 采用的I型指令（其中“\*”意味着虽然CPU支持但功能重复的指令）

|  |  |  |  |
| --- | --- | --- | --- |
| 助记符 | 解释 | 含义 | op |
| j | 无条件跳转 | PC ← (PC + 4)[31:28] || addr || 00 | 100001 |

表1.4 采用的J型指令

# 微操作的定义及节拍的划分

本次设计的CPU为单周期CPU，除了PC ← PC + 4外，几乎没有公共的微操作。各指令的微操作延续表1.2，表1.3与表1.4中的内容，具体如下表2.1所示。

|  |  |  |
| --- | --- | --- |
| 指令 | 解释 | 包含的微指令序列 |
| add rd, rs, rt | 加法 | rd ← rs + rt  PC ← PC + 4 |
| sub rd, rs, rt | 减法 | rd ← rs - rt  PC ← PC + 4 |
| and rd, rs, rt | 按位与 | rd ← rs & rt  PC ← PC + 4 |
| or rd, rs, rt | 按位或 | rd ← rs | rt  PC ← PC + 4 |
| xor rd, rs, rt | 按位异或 | rd ← rs ^ rt  PC ← PC + 4 |
| slt rd, rs, rt | 有符号小于置位 | rd ← (rs < rt) ? 1 : 0  PC ← PC + 4 |
| seq rd, rs, rt | 等于置位 | rd ← (rs = rt) ? 1 : 0  PC ← PC + 4 |
| sgt rd, rs, rt | 有符号大于置位 | rd ← (rs > rt) ? 1 : 0  PC ← PC + 4 |
| sll rd, rt, shamt | 逻辑(算术)左移 | rd ← rt << shamt  PC ← PC + 4 |
| srl rd, rt, shamt | 逻辑右移 | rd ← rt >>> shamt  PC ← PC + 4 |
| sra rd, rt, shamt | 算术右移 | rd ← rt >> shamt  PC ← PC + 4 |
| jr rs | 跳转至寄存器 | PC ← rs |
| addi rt, rs, imm | 立即数加法 | rt ← rs + (sign-extend) imm  PC ← PC + 4 |
| subi rt, rs, imm | 立即数减法 | rt ← rs - (sign-extend) imm  PC ← PC + 4 |
| andi rt, rs, imm | 立即数按位与 | rt ← rs & (sign-extend) imm  PC ← PC + 4 |
| ori rt, rs, imm | 立即数按位或 | rt ← rs | (sign-extend) imm  PC ← PC + 4 |
| xori rt, rs, imm | 立即数按位异或 | rt ← rs ^ (sign-extend) imm  PC ← PC + 4 |
| slti rt, rs, imm | 有符号小于立即数时置位 | rt ← (rs < (sign-extend) imm) ? 1 : 0  PC ← PC + 4 |
| seqi rt, rs, imm | 等于立即数时置位 | rt ← (rs = (sign-extend) imm) ? 1 : 0  PC ← PC + 4 |
| sgti rt, rs, imm | 有符号大于立即数时置位 | rt ← (rs > (sign-extend) imm) ? 1 : 0  PC ← PC + 4 |
| slli rt, rs, imm | 逻辑(算术)左移 | rt ← rs << (sign-extend) imm  PC ← PC + 4 |
| srli rt, rs, imm | 逻辑右移 | rt ← rs >>> (sign-extend) imm  PC ← PC + 4 |
| srai rt, rs, imm | 算术右移 | rt ← rs >> (sign-extend) imm  PC ← PC + 4 |
| lw rt, imm(rs) | 读内存单元 | rt ← mem[rs + (sign-extend) imm]  PC ← PC + 4 |
| sw rt, imm(rs) | 写内存单元 | mem[rs + (sign-extend) imm] ← rt  PC ← PC + 4 |
| beq rs, rt, imm | 相等时跳转 | if (rs == rt)  PC ← PC + 4 + (sign-extend) imm << 2  else  PC ← PC + 4 |
| j | 无条件跳转 | PC ← (PC + 4)[31:28] || addr || 00 |

表2.1 各指令及其微操作序列

同时，由于采用单周期的设计，每个指令都使用1个时钟周期来执行。这意味着，时钟周期的长度取决于耗时最长的指令（如lw与sw）。因此，在这里我们没有特殊的节拍划分，仅有唯一的节拍T1。进一步说：时钟周期 = 机器周期 = 指令周期。

# 处理器结构设计

本次CPU设计采用单周期设计，故无设置A、B、MAR、MDR、IR等寄存器。同时，程序和数据必须分开存放，使得指令的读取和数据的读写可同时进行。

CPU整体结构及数据通路如图3.1所示。所有的使能信号均为高有效。

设计过程已发布至互联网，可参见<https://www.bilibili.com/video/BV1xgTRegEiM>。

接下来，我们将详细阐述各关键部件。

## 程序计数器PC

程序计数器采用32位寄存器、加法器与数据选择器实现

* 输入：跳转地址BranchAddr，跳转使能Branch
* 输出：当前指令的字节地址PC，以及下一条指令的字节地址PC+4

当跳转使能Branch为0时，执行PC ← PC + 4；当跳转使能Branch为1时，执行PC ← BranchAddr

图示, 示意图

描述已自动生成图片包含 图示

描述已自动生成

图3.2 程序计数器设计图与元件图

## 指令存储器IM

指令存储器用于单独存放程序指令

* 输入：32位的字节地址Addr
* 输出：32位的指令字Inst

本文假定指令存储器为只读存储器，无读写控制信号

文本

中度可信度描述已自动生成

图3.3 指令存储器元件图

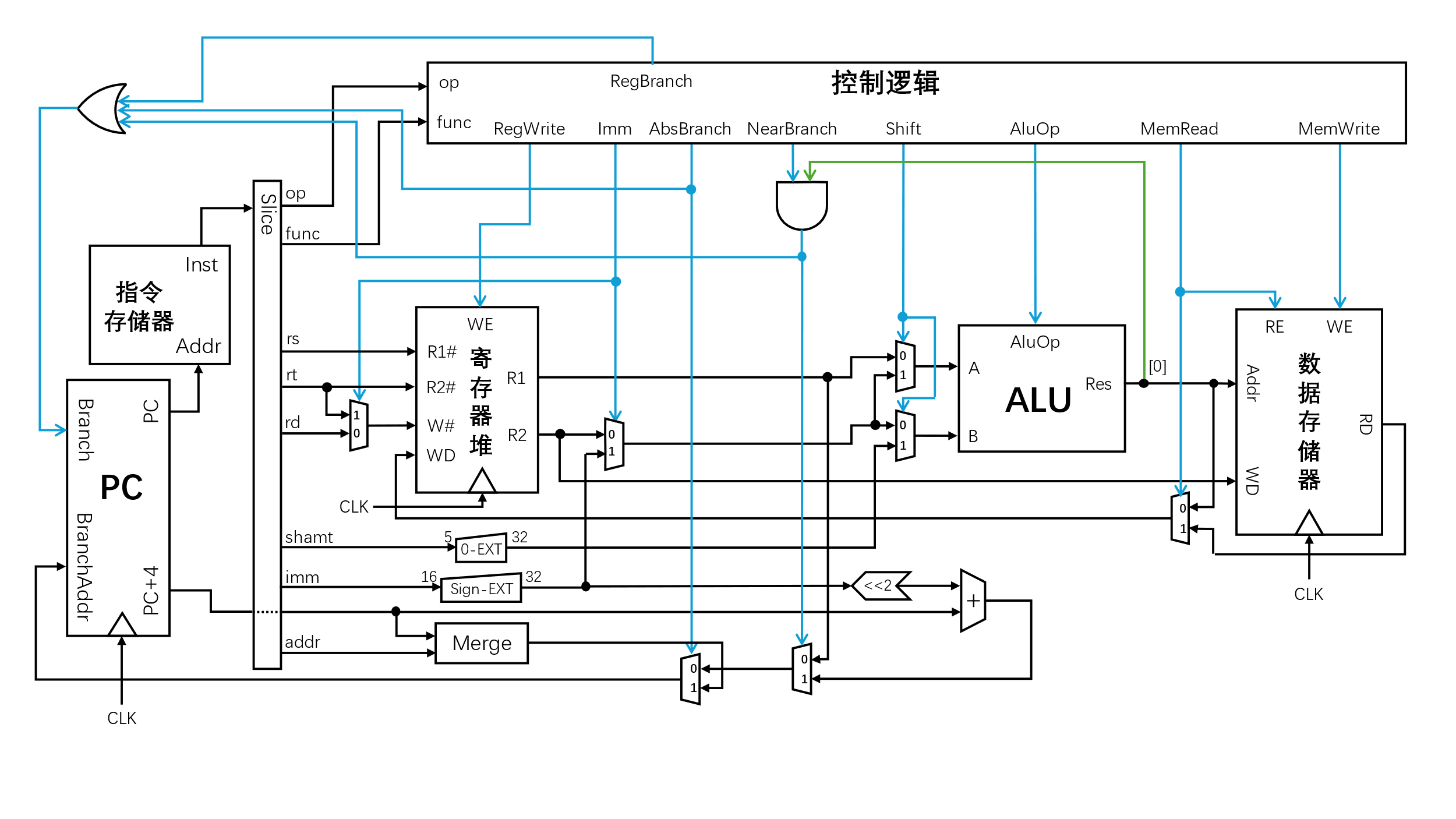


图3.1 CPU整体结构及其数据通路（箭头仅表示数据流向，无物理上的意义）

## 数据存储器DM

数据存储器可读可写，Addr端为32位的地址输入，RE及WE端为读、写使能信号

* RE为1、WE为0时，数据存储器进行读操作，将Addr地址对应存储单元的32位数据输出到RD端
* RE为0、WE为1时，数据存储器进行写操作，在时钟上升沿将WD端的32位数据写入Addr地址对应的存储单元

图示, 示意图

描述已自动生成

图3.4 指令存储器元件图

## 寄存器堆RegHeap

MIPS32中包含32个32位的通用寄存器，按照编号分为$0~$31，本设计中将这32个寄存器结合为寄存器堆以进行集中管理

* 输入：读寄存器编号R1#、R2#，写寄存器编号W#，写数据WD，写使能WE
* 输出：编号为R1#、R2#的寄存器的值R1、R2

图片包含 表格

描述已自动生成

图3.5 寄存器堆元件图

在实际的MIPS32中，这32个寄存器都被赋予了特定的用途，详见表3.6

|  |  |  |
| --- | --- | --- |
| 寄存器号 | 名称 | 用途 |
| 0 | $zero | 常量0 |
| 1 | $at | 为汇编程序保留 |
| 2~3 | $v0~$v1 | 函数调用返回值 |
| 4~7 | $a0~$a3 | 函数调用参数 |
| 8~15 | $t0~$t7 | 临时变量 |
| 16~23 | $s0~$s7 | 通用寄存器 |
| 24~25 | $t8~$t9 | 更多临时变量 |
| 26~27 | $k0~$k1 | 操作系统保留 |
| 28 | $gp | 全局指针 |
| 29 | $sp | 栈指针，指向栈顶 |
| 30 | $fp | 帧指针，用于过程调用 |
| 31 | $ra | 子程序返回地址 |

表3.6 MIPS寄存器功能说明

## 算术逻辑单元ALU

核心的算术逻辑运算单元

* 输入：左操作数A，右操作数B，操作码AluOp（4位）。不同AluOp对应的运算类型见表3.8
* 输出：对于表3.8中允许的AluOp，输出对应的运算结果Res；否则输出0

图片包含 直方图

描述已自动生成

图3.7 ALU元件图

|  |  |  |  |
| --- | --- | --- | --- |
| 操作符 | 解释 | 执行 | 操作码AluOp |
| + | 加法 | Res = A + B | 0001 |
| - | 减法 | Res = A - B | 0010 |
| & | 按位与 | Res = A & B | 0011 |
| | | 按位或 | Res = A | B | 0100 |
| ^ | 按位异或 | Res = A ^ B | 0101 |
| << | 逻辑（算术）左移 | Res = A << B | 0110 |
| >>> | 逻辑右移 | Res = A >>> B | 0111 |
| >> | 算术右移 | Res = A >> B | 1000 |
| < | 有符号小于比较  （输出1位结果，零拓展至32位） | Res = A < B | 1001 |
| = | 等于比较  （输出1位结果，零拓展至32位） | Res = A == B | 1010 |
| > | 有符号大于比较  （输出1位结果，零拓展至32位） | Res = A > B | 1011 |

表3.8 ALU可执行的运算类型及其操作码AluOp

## 杂项

* 数据选择器：在图中用带0、1标记的等腰梯形表示，根据控制信号选择0路或1路的输入进行输出
* 加法器：用带“+”符号的等腰梯形表示，将输入的两个数进行相加并输出，不考虑溢出
* 移位器：用带“<<2”字样的粗左箭头符号表示，将输入左移2位后输出
* 零拓展单元：用带“0-Ext”字样的图形表示，将5位的输入零拓展至32位后输出
* 有符号拓展单元：用带“Sign-Ext”字样的图形表示，将16位的输入有符号拓展至32位后输出
* 合并单元：用带“Merge”字样的矩形表示，为指令j的专用运算器，将输入的PC+4的高4位，与addr输入的26位，与00，合并成32位的地址输出
* 切片单元：用带“Slice”字样的矩形表示，提取出指令输入Inst的各个字段op、func、rs、rt、rd、shamt、imm、addr后并输出，对应关系见下图3.9

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 字段 | op | func | rs | rt | rd | shamt | imm | addr |
| 位范围 | [31,26] | [5,0] | [25,21] | [20,16] | [15,11] | [10,6] | [15,0] | [25,0] |

图3.9 各字段与输入指令位的对应关系

# 硬布线控制逻辑设计

为了完成对数据通路的控制，我们设计了如下的控制信号，如表4.1所示。

|  |  |
| --- | --- |
| 控制信号 | 信号描述 |
| RegBranch | jr等无条件寄存器跳转信号 |
| NearBranch | beq等有条件分支跳转信号 |
| AbsBranch | j等大范围无条件分支跳转信号 |
| AluOp | 控制ALU进行不同的运算 |
| RegWrite | 控制寄存器堆的写操作，为1时写入寄存器 |
| MemRead | 控制数据存储器的读取，为1时为读操作 |
| MemWrite | 控制数据存储器的写入，为1时为写操作 |
| Shift | sll、srl、sra这些移位运算的信号 |
| Imm | 立即数参与运算时的信号 |

表4.1 控制信号描述

接下来，我们使用组合逻辑进行控制单元的实现。

## 分离式设计

这里的“分离式设计”指，先从op、func指令字段中获取add、sub、beq等代表某条指令执行的中间信号，再从这些中间信号获取需要的操作信号。

获取中间操作信号的电路如下图4.1.1：

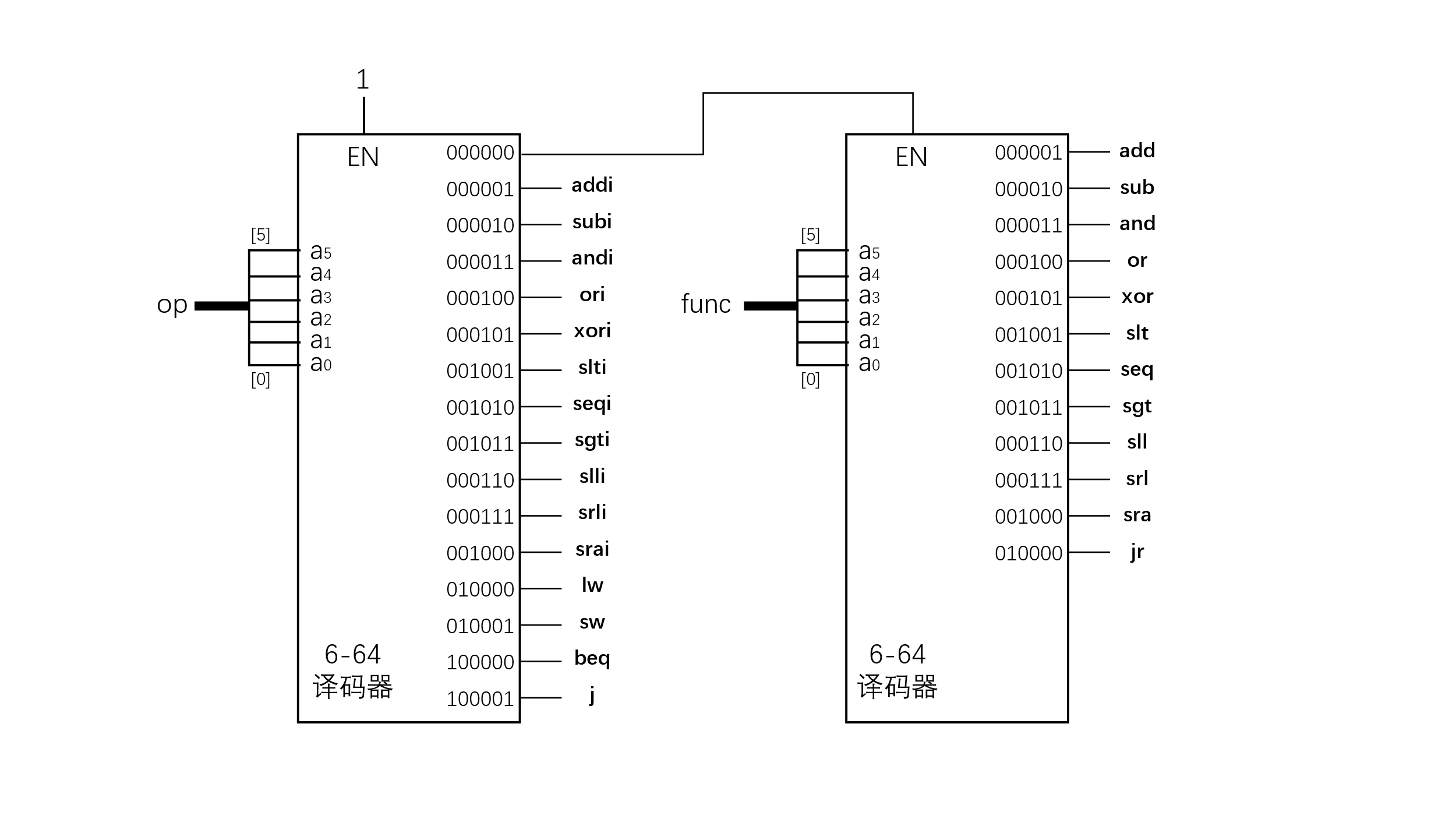


图4.1.1 从op、func中获取中间信号

表4.1.2列出了中间信号与最终控制信号的操作时间表：

表4.1.2 控制信号的操作时间表（前表）

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| slt |  |  |  | func  [3:0] | 1 |  |  |  |  |
| xor |  |  |  | func  [3:0] | 1 |  |  |  |  |
| or |  |  |  | func  [3:0] | 1 |  |  |  |  |
| and |  |  |  | func  [3:0] | 1 |  |  |  |  |
| sub |  |  |  | func  [3:0] | 1 |  |  |  |  |
| add |  |  |  | func  [3:0] | 1 |  |  |  |  |
| 控制信号 | RegBranch | NearBranch | AbsBranch | Aluop | RegWrite | MemRead | MemWrite | Shift | Imm |

表4.1.2 控制信号的操作时间表（中表）

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| srai |  |  |  | op  [3:0] | 1 |  |  |  | 1 |
| srli |  |  |  | op  [3:0] | 1 |  |  |  | 1 |
| slli |  |  |  | op  [3:0] | 1 |  |  |  | 1 |
| sgti |  |  |  | op  [3:0] | 1 |  |  |  | 1 |
| seqi |  |  |  | op  [3:0] | 1 |  |  |  | 1 |
| slti |  |  |  | op  [3:0] | 1 |  |  |  | 1 |
| xori |  |  |  | op  [3:0] | 1 |  |  |  | 1 |
| ori |  |  |  | op  [3:0] | 1 |  |  |  | 1 |
| andi |  |  |  | op  [3:0] | 1 |  |  |  | 1 |
| subi |  |  |  | op  [3:0] | 1 |  |  |  | 1 |
| addi |  |  |  | op  [3:0] | 1 |  |  |  | 1 |
| jr | 1 |  |  |  |  |  |  |  |  |
| sra |  |  |  | func  [3:0] | 1 |  |  | 1 |  |
| srl |  |  |  | func  [3:0] | 1 |  |  | 1 |  |
| sll |  |  |  | func  [3:0] | 1 |  |  | 1 |  |
| sgt |  |  |  | func  [3:0] | 1 |  |  |  |  |
| seq |  |  |  | func  [3:0] | 1 |  |  |  |  |
| 控制信号 | RegBranch | NearBranch | AbsBranch | Aluop | RegWrite | MemRead | MemWrite | Shift | Imm |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 控制信号 | lw | sw | beq | j |
| RegBranch |  |  |  |  |
| NearBranch |  |  | 1 |  |
| AbsBranch |  |  |  | 1 |
| AluOp | 0001（加） | 0001（加） | 1010（相等） |  |
| RegWrite | 1 |  |  |  |
| MemRead | 1 |  |  |  |
| MemWrite |  | 1 |  |  |
| Shift |  |  |  |  |
| Imm | 1 | 1 |  |  |

表4.1.2 控制信号的操作时间表（后表）

根据上述的操作时间表，我们就能给出每个控制信号的最简表达式，结果见下表4.1.3：

|  |  |
| --- | --- |
| 控制信号 | 最简表达式 |
| RegBranch | jr |
| NearBranch | beq |
| AbsBranch | j |
| AluOp | (add+sub+and+or+xor+slt+seq+sgt+sll+srl+sra)\*func[3:0]+  (addi+subi+andi+ori+xori+slti+seqi+sgti+slli+srli+srai)\*op[3:0]+  (lw+sw)\*0001+beq\*1010 |
| RegWrite | add+sub+and+or+xor+slt+seq+sgt+sll+srl+sra+  addi+subi+andi+ori+xori+slti+seqi+sgti+slli+srli+srai+  lw |
| MemRead | lw |
| MemWrite | sw |
| Shift | sll+srl+sra |
| Imm | addi+subi+andi+ori+xori+slti+seqi+sgti+slli+srli+srai+  lw+sw |

表4.1.3 各个控制信号的最简表达式

进而，根据我们表格中的表达式，就可绘制出控制逻辑的电路图，见图4.1.4。

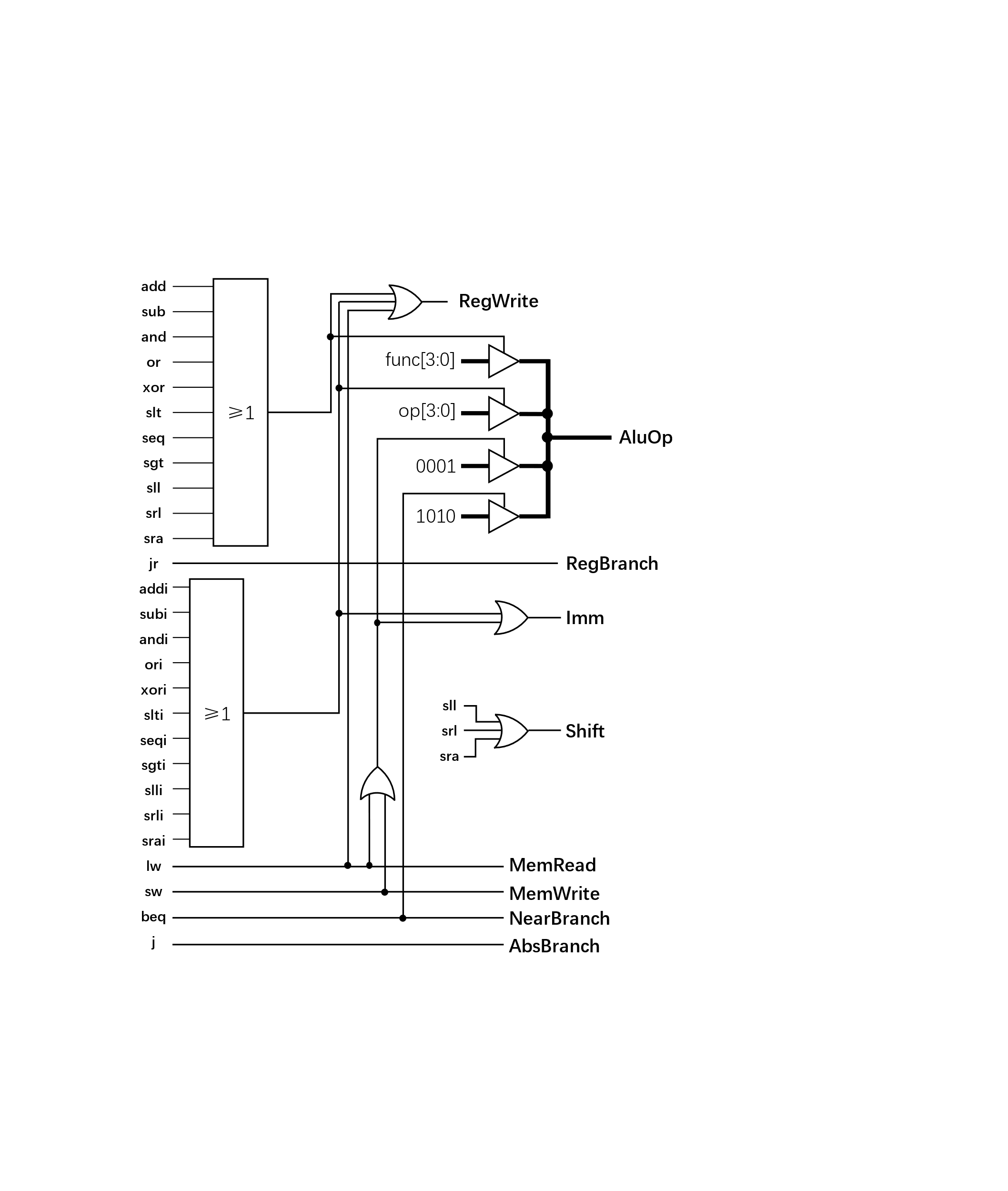


图4.1.4 分离式控制逻辑电路

## 一体式设计

这里的“一体式设计”指，直接从op、func指令字段中获取需要的操作信号。

一体式控制单元整体电路见图4.2.1。

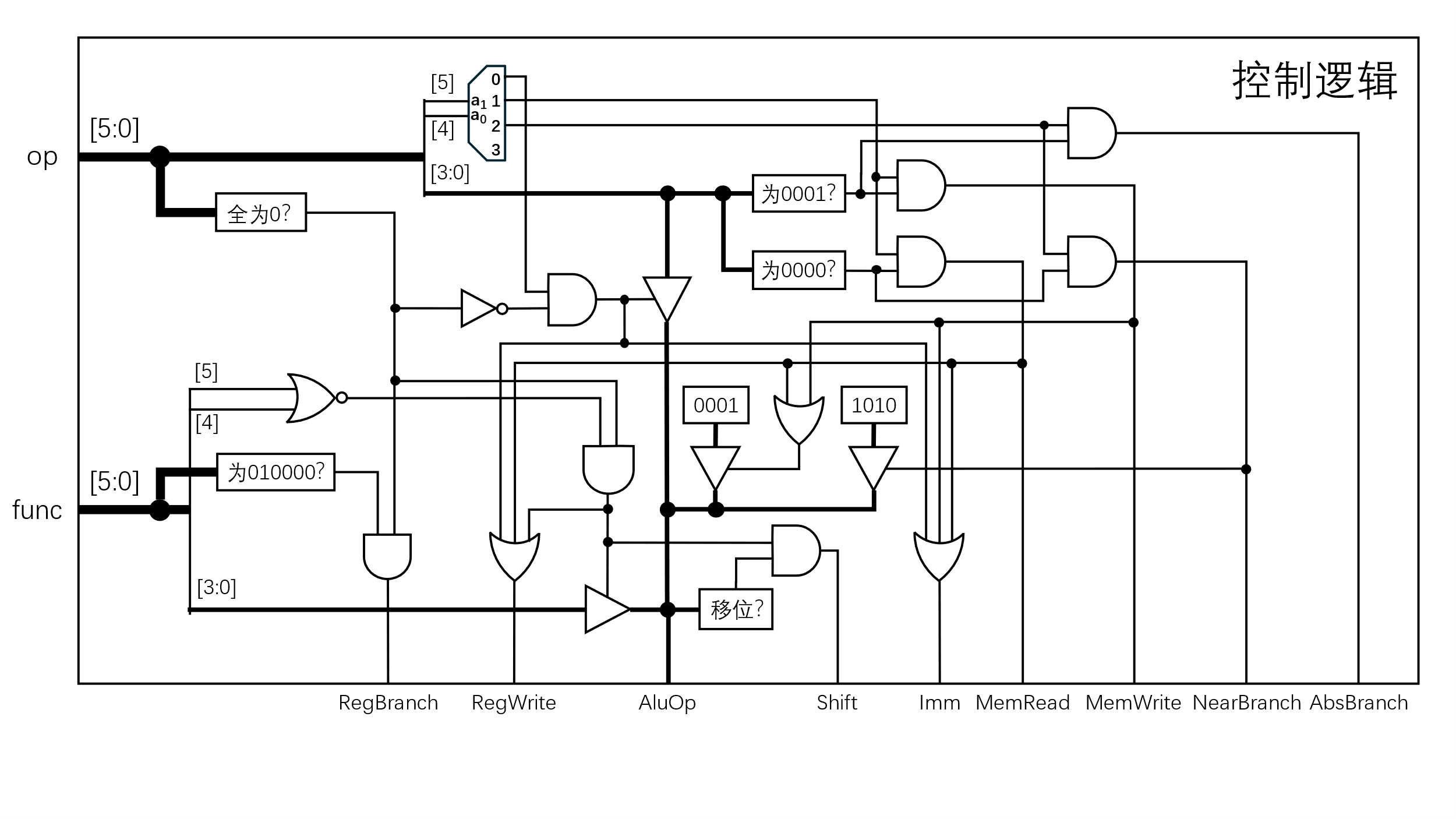


图4.2.1 一体式CPU控制单元电路示意图（图中包含a0、a1的部件为2-4译码器，输出端高有效）

接下来我们将具体阐述每个信号的产生及其表达式。

### RegBranch

其示意图见下。表达式为：~(op[5] + op[4] + op[3] + op[2] + op[1] + op[0]) \* (~func[5] \* func[4] \* ~func[3] \* ~func[2] \* ~func[1] \* ~func[0])

图示

描述已自动生成

图4.2.2 RegBranch信号产生的示意图

### NearBranch

其示意图见下。表达式为：op[5] \* ~op[4] \* ~op[3] \* ~op[2] \* ~op[1] \* ~op[0]

图示

描述已自动生成

图4.2.3 NearBranch信号产生的示意图

### AbsBranch

其示意图见下。表达式为：op[5] \* ~op[4] \* ~op[3] \* ~op[2] \* ~op[1] \* op[0]

图示

中度可信度描述已自动生成

图4.2.4 AbsBranch信号产生的示意图

### AluOp

其示意图见下。表达式为：

func[3:0] \* ~(func[5] + func[4]) \* ~(op[5] + op[4] + op[3] + op[2] + op[1] + op[0]) +

op[3:0] \* (~op[5] \* ~op[4]) \* (op[5] + op[4] + op[3] + op[2] + op[1] + op[0]) +

0001 \* (~op[5] \* op[4] \* ~op[3] \* ~op[2] \* ~op[1]) +

1010 \* (op[5] \* ~op[4] \* ~op[3] \* ~op[2] \* ~op[1] \* ~op[0])

其中4位数据与1位数据的位运算为类似分配律的按位运算。

e.g. 0101 \* 1 = 1 \* 0101 = 0101，0101 \* 0 = 0 \* 0101 = 0000。

其中4位数据与4位数据的位运算为正常的按位运算

e.g. 0101 | 0000 = 0101，0101 | 1100 = 1101（实际上不可能出现这种情况，实际实现时使用的是“三态门+类总线”的方式来实现AluOp的生成，控制逻辑保证同时至多只有一个三态门打开）

化简该表达式后的可能结果如下：

~(op[5] + op[4]) \* [

func[3:0] \* ~(func[5] + func[4] + op[3] + op[2] + op[1] + op[0]) +

op[3:0] \* (op[3] + op[2] + op[1] + op[0])

] + ~(op[3] + op[2] + op[1]) \* [

0001 \* (~op[5] \* op[4]) +

1010 \* (op[5] \* ~op[4] \* ~op[0])

]

图示

描述已自动生成

图4.2.5 AluOp信号产生的示意图

### RegWrite

其示意图见下。表达式为：

~(op[5] + op[4] + op[3] + op[2] + op[1] + op[0]) \* ~(func[5] + func[4]) +

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

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

（这里我们默认与运算的优先级比或运算高）

其可能的化简结果为：

~(op[5] + op[4]) \* [

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

(op[3] + op[2] + op[1] + op[0])

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

另一个可能的化简结果为：

~(op[3] + op[2] + op[1] + op[0]) \* [

~(op[5] + op[4] + func[5] + func[4]) +

(~op[5] \* op[4])

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

图示, 示意图

描述已自动生成

图4.2.6 RegWrite信号产生的示意图

### MemRead

其示意图见下。表达式为：~op[5] \* op[4] \* ~op[3] \* ~op[2] \* ~op[1] \* ~op[0]

图示

描述已自动生成

图4.2.7 MemRead信号产生的示意图

### MemWrite

其示意图见下。表达式为：~op[5] \* op[4] \* ~op[3] \* ~op[2] \* ~op[1] \* op[0]

图示

描述已自动生成

图4.2.8 MemWrite信号产生的示意图

### Shift

其示意图见下。其中“移位？”元件的行为是“判断输入的4位数据是否为0110、0111、1000三者之一”。表达式为：

~(op[5] + op[4] + op[3] + op[2] + op[1] + op[0]) \* ~(func[5] + func[4]) \* [

(~func[3] \* func[2] \* func[1] \* ~func[0]) +

(~func[3] \* func[2] \* func[1] \* func[0]) +

(func[3] \* ~func[2] \* ~func[1] \* ~func[0])

]

其可能的化简结果为：

~(op[5] + op[4] + op[3] + op[2] + op[1] + op[0]) \* ~(func[5] + func[4]) \* [

(~func[3] \* func[2] \* func[1]) +

(func[3] \* ~func[2] \* ~func[1] \* ~func[0])

]

图示

描述已自动生成

图4.2.9 Shift信号产生的示意图（图中未简化）

### Imm

其示意图见下。表达式为：

~(op[5] + op[4]) \* (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])

其可能的化简结果为：

~op[5] \* [

~op[4] \* (op[3] + op[2] + op[1] + op[0]) +

op[4] \* ~(op[3] + op[2] + op[1])

]

可能的进一步化简结果为：

~op[5] \* [

op[4] ^ (op[3] + op[2] + op[1]) +

~op[4] \* op[0]

]

（其中“^”表示异或运算）

图示

描述已自动生成

图4.2.10 Imm信号产生的示意图