中国科学院大学《计算机组成原理(研讨课)》实验报告

姓名 胡格致 学号 2023K8009929002 专业 计算机科学与技术

实验项目编号 1 实验名称 基本功能部件

1 寄存器堆

1.1 需求分析

根据实验讲义和 reg_file.v 的接口进行分析,我们可以得到每个接口的定义:

```
1  `define DATA_WIDTH 32
2  `define ADDR_WIDTH 5
3
4  module reg_file(
5      input                           clk,
6
7      input                           wen,
8      input    [`ADDR_WIDTH - 1:0] waddr,
9      input    [`DATA_WIDTH - 1:0] wdata,
10
11     input    [`ADDR_WIDTH - 1:0] raddr1,
12     output   [`DATA_WIDTH - 1:0] rdata1,
13
14     input    [`ADDR_WIDTH - 1:0] raddr2,
15     output   [`DATA_WIDTH - 1:0] rdata2
16 );
```

宏名称	宏描述		
`DATA_WIDTH	描述数据接口和寄存器宽度		
`ADDR_WIDTH	描述地址接口宽度		
(1<<`ADDR_WIDTH)	描述寄存器长度		
接口名称	**宽度**	**方向**	**描述**
全局信号			
clk	1	I	时钟
写口			
wen	1	I	写使能
waddr	5	I	写地址
wdata	32	I	写数据
读口一			
raddr1	5	I	读地址一
rdata1	32	O	读数据一
读口二			
raddr2	5	I	读地址二
rdata2	32	O	读数据二

1.2 设计实现

根据 MIPS 对寄存器堆的要求,我们需要特别注意:零号寄存器是锁定在零值上的,不允许写入。
从而,我们可以省去零号寄存器对应的 reg,由此得到寄存器声明:

```
1  reg [`DATA_WIDTH - 1:0] regs [(1 << `ADDR_WIDTH) - 1:1];
```

写口需要在上升沿时注意写使能。当写使能为高,并且地址非零时,将数据写入对应寄存器:

```
1  always @(posedge clk)
2  begin
3      if (wen && waddr != `ADDR_WIDTH'b0)
4      begin
5          regs[waddr] <= wdata;
6      end
7  end
```

而读口只需要特判零号寄存器，然后将数据传递给输出端口即可：

```
    assign rdata1 = (raddr1 == 0 ? 32'b0 : regs[raddr1]);
    assign rdata2 = (raddr2 == 0 ? 32'b0 : regs[raddr2]);
```

1.3 结构分析

在 Vivado 中，我们可以通过 Schematic 视图查看生成的电路结构。如图 1所示，我们可以看到寄存器堆的结构：

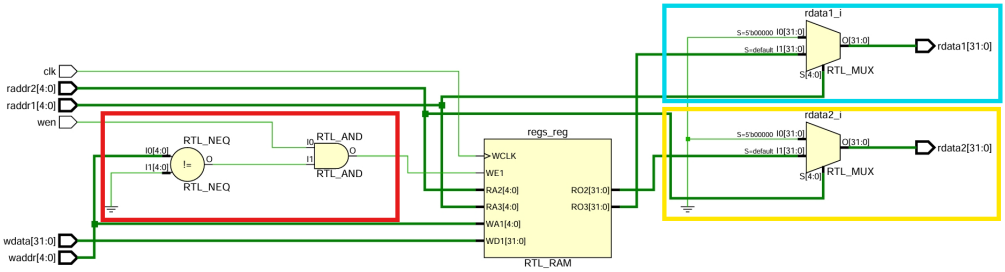

图 1: 寄存器堆电路结构

在图 1种红框是我们设计的写口特判，中间的黄色元件是我们的寄存器堆主体，黄框和蓝框则是两个读口的特判。这样的设计保证了零号寄存器的特殊性。

1.4 分析优化

我们在此给出另一种特判零号寄存器的方法，即在标准的读写之外，额外每周期向零号寄存器写入零值。这样，我们就可以省去特判零号寄存器的步骤。

下给出两种方法的代码的对比：

```
reg [`DATA_WIDTH - 1:0] regs [(1 <<
    `ADDR_WIDTH) - 1:1];

always @(posedge clk)
begin
    if (wen && waddr != `ADDR_WIDTH'b0)
    begin
        regs[waddr] <= wdata;
    end
end

assign rdata1 = (raddr1 == 0 ? 32'b0 :
    regs[raddr1]);
assign rdata2 = (raddr2 == 0 ? 32'b0 :
    regs[raddr2]);
```

```
reg [`DATA_WIDTH - 1:0] regs [(1 <<
    `ADDR_WIDTH) - 1:0];

always @(posedge clk)
begin
    if (wen)
    begin
        regs[waddr] <= wdata;
    end
    regs[0] <= 0;
end

assign rdata1 = regs[raddr1];
assign rdata2 = regs[raddr2];
```

右侧的方法看似可以省去特判，但是实际上相比于左侧的特判法在面积和时序上都是退步的。从 Vivado 的综合结果中可以看出部分原因：

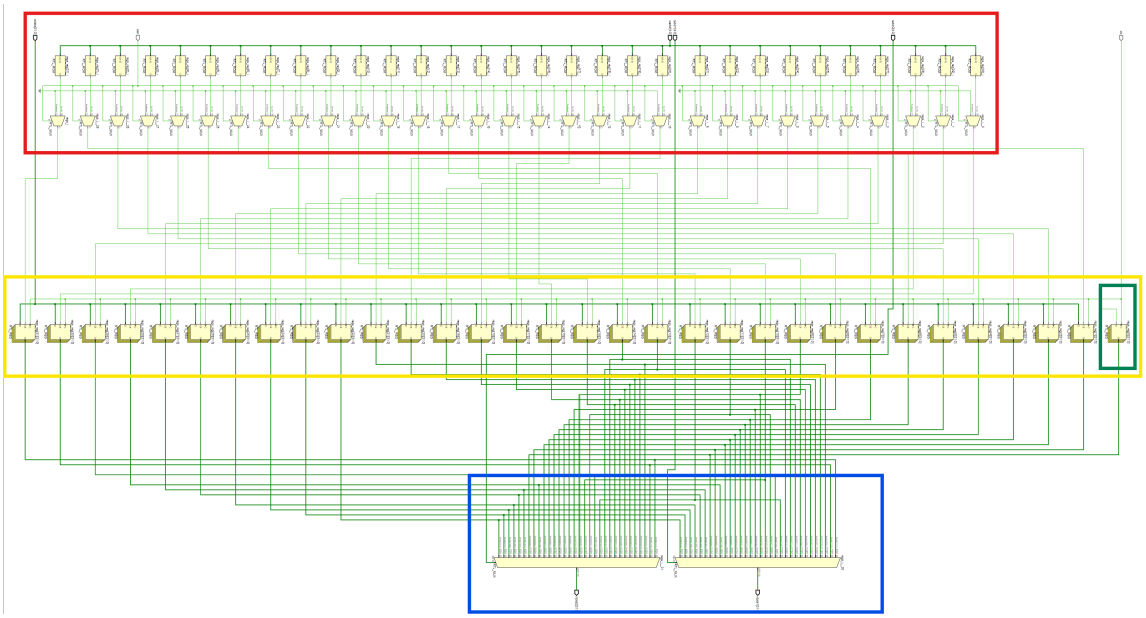

图 2: 寄存器堆电路结构

如图 2所示，其中红框是写筛选，黄框是寄存器堆主题，绿框是专门圈出的零号寄存器，蓝框则是两个读口筛选。零号寄存器需要特别注意，是因为它实际上是一个不可写的只读存储器。

因此，这种方法首先付出了额外一个 32 位 ROM，这对应的面积远大于特判对应的面积。通过 yosys 的综合，从表 1中看到最终的面积和时序差距：

表 1: 综合结果对比

方法	面积/μm^2	频率/MHz
额外零法	13116.194	4134.452
特判零法	12319.258	4363.687

如表 1所示，额外零法的面积和时序都比特判零法要差，因此我们选择特判零法作为最终的设计。

1.5 感想

MIPS 在设计中采用了特殊的零号寄存器，这是为了方便编译器和硬件设计。

具体来讲，通过使用零号寄存器，我们可以用 add a0, x0, a1 来代替 move a0, a1，用 slt a0, x0, a1 来快速做到正负判断，这些优化可以节省指令集，从而优化硬件和编译器的设计。

同时，因为 x0 不可写入，从而可以用于充当 nop 指令。

这种设计思想被 RISC-V 和 ARMv8 等指令集中所引用，是一种非常好的设计思想。

同时，相比起原始的 ARM，MIPS 并未采用 PC 作为通用寄存器，这样的设计配合上延迟槽，让 MIPS 的跳转可以被提前预知，从而让其在流水线上的表现更加优秀。

2 算术逻辑单元

2.1 需求分析

2.1.1 接口分析

根据实验讲义和 alu.v 的接口进行分析，我们可以得到每个接口的定义：

```
1  `define DATA_WIDTH 32
2
3  module alu(
4      input  [`DATA_WIDTH - 1:0]  A,
5      input  [`DATA_WIDTH - 1:0]  B,
6      input  [              2:0]  ALUop,
7      output [`DATA_WIDTH - 1:0]  Result,
8      output                      Overflow,
9      output                      CarryOut,
10     output                      Zero
11 );
```

宏名称	宏描述		
`DATA_WIDTH	描述数据接口宽度		
接口名称	宽度	方向	描述
输入数据			
A	32	I	操作数 A
B	32	I	操作数 B
输入指令			
ALUop	3	I	算数指令
输出数据			
Result	32	O	计算结果
输出信号			
Overflow	1	O	溢出信号
CarryOut	1	O	进借位信号
Zero	1	O	为零信号

2.1.2 信号分析

通过分析讲义对 ALUop 的定义，我们可以得到以下表格：

表 2: ALUop 定义

ALUop[2]	ALUop[1]	ALUop[0]	操作	描述
0	0	0	AND	Res = A & B
0	0	1	OR	Res = A | B
0	1	0	ADD	Res = A + B
1	1	0	SUB	Res = A + (-B)
1	1	1	SLT	Res = (A < B ? 1 : 0)

注意到操作数是-B 当且仅当 ALUop[2] 为 1，结合有符号相反数表示是-B = ~B + 1，因此我们可以定义 D = (ALUop[2] == 0 ? B : ~B) = 32{ALUop[2]} ^ B 和 C = A + D + ALUop[2]，从而简化 ALUop 的定义为：

表 3: ALUop 定义，改版一

ALUop[1:0]	操作	描述
2'b00	AND	Res = A & B
2'b01	OR	Res = A | B
2'b10	ADD & SUB	Res = C
2'b11	SLT	Res = (A < D ? 1 : 0)

从而我们可以分别计算出来这些操作的结果，然后通过 ALUop[1:0] 的定义来选择结果进行输出。

接下来考虑输出时的信号们。首先，我们关注 Overflow 信号的定义：指示有符号数的加减法运算结果是否溢出。

注意到溢出的定义：两个非负数相加得到负数，或者两个负数相加得到非负数，或者非负数减去负数得到负数，或者负数减去非负数得到非负数。

注意到根据补码的定义，一个数的 [31] 位是这个数的符号位，0 表示非负数，1 表示负数。从而我们可以作出如下分析：

表 4: 溢出条件

ALUop[2]	A[31]	B[31]	C[31]	描述
0	0	0	1	非负数加非负数得负数
0	1	1	0	负数加负数得非负数
1	0	1	1	非负数减负数得负数
1	1	0	0	负数减非负数得非负数

结合 D[31] = B[31] ^ ALUop[2]，我们可以得到简化的溢出信号定义：

表 5: 溢出条件，改版一

A[31]	D[31]	C[31]
0	0	1
1	1	0

现在我们考虑 CarryOut 信号的定义：指示无符号数的加减法运算结果是否产生进或借位。

注意到无符号加法的进位的结果 C[32]，无符号减法的借位是 C[32] 的反（这是因为在无符号减法的计算过程中，有-B = B + 1 = (2^32 - 1) - B + 1 = 2^32 - B，从而当且仅当这最高位被借位时才会变为 0）。因此我们可以得到简化的进借位信号定义：

表 6: 进借位条件

C[32]	ALUop[2]
1	0
0	1

最后我们考虑 Zero 信号的定义：指示运算结果是否为零。这个信号的定义是最简单的，只需要判断 Result 是否为零即可。

除了以上三个信号以外，我们还需要专门考虑 SLT 指令的结果。

我们列出所有可能的 A[31], B[31], C[31] 的可能性以及对应的 SLT, Overflow 结果：

表 7: SLT 结果分析

A[31]	B[31]	C[31]	SLT	Overflow	描述
0	0	0	0	0	非负数减非负数得非负数,不满足 SLT,无溢出
0	0	1	1	0	非负数减非负数得负数,满足 SLT,无溢出
0	1	0	0	0	非负数减负数得非负数,不满足 SLT,无溢出
0	1	1	0	1	非负数减负数得负数,不满足 SLT,溢出
1	0	0	1	1	负数减非负数得非负数,满足 SLT,溢出
1	0	1	1	0	负数减非负数得负数,满足 SLT,无溢出
1	1	0	0	0	负数减负数得非负数,不满足 SLT,无溢出
1	1	1	1	0	负数减负数得负数,满足 SLT,无溢出

由上我们可以注意到一个规律:SLT 的结果是 C[31] ^ Overflow

2.2 设计实现

根据上述的分析,我们可以得到 alu.v 的代码实现。接下来我将逐行进行分析:

```
1     assign res_and  = A & B;
2     assign res_or   = A | B;
```

这两行代码分别计算 AND 和 OR 的结果。

```
1     assign signB    = ALUop[2];
2     assign D        = {`DATA_WIDTH{signB}} ^ B;
3     assign {Co, C}  = (A + D + signB);
```

这三行代码分别计算 D 和 C,其中 D 是 ALUop[2] == 0 ? B : B,C 是 A + D + ALUop[2]。

```
1     assign Asign    = A[`DATA_WIDTH-1];
2     assign Bsign    = B[`DATA_WIDTH-1];
3     assign Dsign    = D[`DATA_WIDTH-1];
4     assign Csign    = C[`DATA_WIDTH-1];
```

这四行代码分别计算 A, B, D, C 的符号位。

```
1     assign CarryOut     = Co ^ signB;
2     assign Overflow     = (Asign == Dsign) && (Asign^Csign);
3     assign res_slt = {{`DATA_WIDTH-1{1'b0}},Csign^Overflow};
```

这三行代码分别计算 CarryOut, Overflow 和 SLT 的结果。

```
1     assign Result  = ALUop[1] ?
2                      (ALUop[0] ? res_or  : res_and):
3                      (ALUop[0] ? res_slt : C);
```

这行代码根据 ALUop 的定义，选择输出结果。

```
1     assign Zero    = (Result == 0);
```

这行代码判断 Result 是否为零。

2.3 结构分析

在 Vivado 中，我们可以通过 Schematic 视图查看生成的电路结构。如图 3所示，我们可以看到算术逻辑单元的结构：

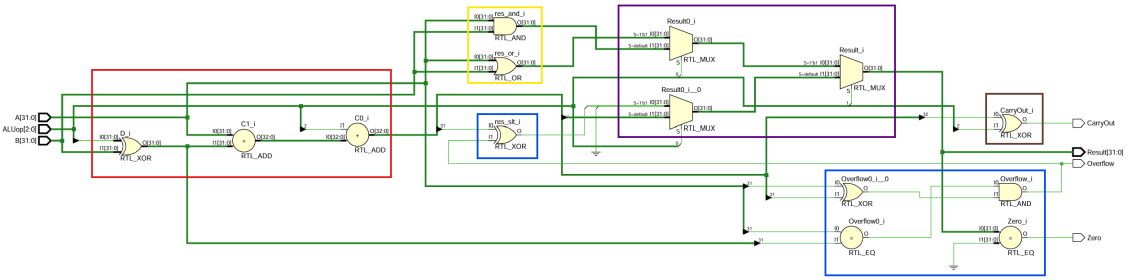

图 3: 算术逻辑单元电路结构

如图所示，左侧的红圈对应我们的D和C的计算，中间的黄色元件对应 AND 和 OR 的计算，右下角的蓝圈的左上，左下，右上三个元件在计算 Overflow 的值，这个值通过信号线传递到中间的蓝框中，计算得到 SLT 的值。最终，四个结果通过紫框的选择器选择，得到最后的 Result。同时，最右侧的棕框计算 CarryOut 的值，而右下角蓝圈的右下角元件计算 Zero 的值。

2.4 分析优化

这一版本的设计实现已经经过了多次的优化，我们在此首先对比最终版和第一版在经过 yosys 综合后的面积进行对比：

表 8: 综合结果对比

版本	面积/μm^2
第一版	913.710
最终版	821.674

（由于本电路为纯组合电路，从而暂时不对频率进行分析）

如表 8所示，最终版的面积比第一版的面积减少了大约 10%。

为什么达到同样效果的组合逻辑电路，不同的设计实现会有不同的面积呢？这是因为不同的设计实现会导致不同的逻辑电路结构，从而会导致不同的面积。如下页的对比所示：

```
assign check_of  = {
b_muxer,
A[`DATA_WIDTH-1],
B[`DATA_WIDTH-1],
res_add[`DATA_WIDTH-1]};
assign Overflow  =
   (check_of == 4'b0001
 || check_of == 4'b0110
 || check_of == 4'b1011
 || check_of == 4'b1100);
assign res_slt  =
{{`DATA_WIDTH-1{1'b0}},
   (check_of == 4'b1100
 || check_of == 4'b1101
 || check_of == 4'b1111
 || check_of == 4'b1001)};
assign Result   =
  {`DATA_WIDTH{ALUop[1:0]==2'b00}} & res_and
| {`DATA_WIDTH{ALUop[1:0]==2'b01}} & res_or
| {`DATA_WIDTH{ALUop[1:0]==2'b10}} & res_add
| {`DATA_WIDTH{ALUop[1:0]==2'b11}} & res_slt;
```

```
assign Overflow = (Asign == Dsign) &&
↪  (Asign^Csign);
assign res_slt  =
↪  {{`DATA_WIDTH-1{1'b0}},Csign^Overflow};
assign Result  = ALUop[1] ?
(ALUop[0] ? res_or  : res_and):
(ALUop[0] ? res_slt : C);
```

（经过一定的排版优化）

如上所示，左侧是第一版的代码，右侧是最终版的代码。可以看到，最终版的代码通过逻辑上的化简，减少了中间运算的数量，从而减少了综合后电路上的元器件数量，从而减少了面积。

2.5 感想

本次实验虽然只是编写了一个简单的算术逻辑单元，但是我们仍然可以通过研究来对其进行大幅度的优化。通过利用设计时的逻辑化简，我们可以减少中间运算的数量，从而减少综合后的面积。

反过来，我们也可以考虑精心设计这样的指令操作对应的值，从而让硬件设计更加简单。

不过与此同时，我们也要注意到 MIPS 中的溢出信号会产生异常，这种设计让其在需要强制处理算术指令的异常，即使软件不需要处理这个异常。

3 实验总结

本次实验中，我们首先实现了一个简单的寄存器堆和一个简单的算数逻辑单元。尽管这两个模块都很简单的，但是我们仍然可以通过对其进行深入的分析和优化，从而让我们的设计更加简洁。同时，我们也可以在设计的背后看到 MIPS 的设计思想，从而更加深入的理解 MIPS

在本次实验中我花费了大约 3 小时完成此次实验。