ball ball你们不要再集中式写控制器了

前言

去年写计组的时候，上了流水线，采用了集中式译码，单一Controller的同学都苦不堪言。

> “我加一个指令，需要动5，6个模块”

> “连线连得我想吐了”

而今年，我参加龙芯杯的时候，看着我去年的屎山一般的代码，实在不想改了。

后面稍微参考了一下龙芯正式生产的CPU:ls232的源码。然后弄出了一种分布式自控制的代码设计。

看完介绍，相信你会知道为什么 **内部指令总线+自控制** 的设计，才是真正的快乐

很多教程所教的方法：

书本上，还有课程网站给的样例图上，都是说了一种被称为“集中译码式""的架构

这种设计的特征是：有一个超大的Controller，几乎连接了CPU的每一个部分

然后还有一大堆多路选择器。

等下，高老板不是一直说**高内聚，低耦合**吗？这么多连线是闹咋样？

在这种设计下，新增代码所需要改动的地方

（只放主要部分）

1.首先需要在译码器增加这个信号（所有的设计中，这一步都是必须的）

```
assign  addi=    (OpCode==6'h8),
        addiu=   (OpCode==6'h9),
        add=     (OpCode==6'h0 && Funct==6'h20),
        addu=    (OpCode==6'h0 && Funct==6'h21),
        sub=     (OpCode==6'h0 && Funct==6'h22),
        subu=    (OpCode==6'h0 && Funct==6'h23),
        lui=     (OpCode==6'hf);
```

2.设置控制器的参数和信号

```
parameter     ALU_add=0,
              ALU_addu=1,
              ALU_sub=2,
              ALU_subu=3,
              ALU_Or=4,
              ALU_And=5,
              ALU_OutPutB=6,
              ALU_OutPutA=7,
              ALU_sll=8,
              ALU_srl=9,
              ALU_sra=10,
              ALU_Xor=11,
              ALU_Nor=12,
              ALU_slt=13,
              ALU_sltu=14;
```

```
assign AluCtrl= (addiu||addu||lw||sw||lb||lbu||lh||lhu||sb||sh)        ?  ALU_addu:
                (add||addi)                                            ?  ALU_add:
                (subu)                                                 ?  ALU_subu:
                (sub)                                                  ?  ALU_sub:
                (ori||Or)                                              ?  ALU_Or:
                (And||Andi)                                            ?  ALU_And:
                (Xor||Xori)                                            ?  ALU_Xor:
                (Nor)                                                  ?  ALU_Nor:
                (lui||jal||jalr)                                       ?  ALU_OutPutB:
                (sll||sllv)                                            ?  ALU_sll:
                (srl||srlv)                                            ?  ALU_srl:
                (sra||srav)                                            ?  ALU_sra:
                (slt||slti)                                            ?  ALU_slt:
                (sltu||sltiu)                                          ?  ALU_sltu:
                                                                          0;
```

3.在ALU里面添加对应的运算操作

```
assign C=   (ALU_Ctrl==add)      ?   tmp[31:0]:
            (ALU_Ctrl==sub)      ?   tmp[31:0]:
            (ALU_Ctrl==addu)     ?   A+B:
            (ALU_Ctrl==subu)     ?   A-B:
            (ALU_Ctrl==Or)       ?   A|B:
            (ALU_Ctrl==And)      ?   A&B:
            (ALU_Ctrl==Xor)      ?   A^B:
            (ALU_Ctrl==Nor)      ?   ~(A|B):
            (ALU_Ctrl==OutPutA)  ?   A:
            (ALU_Ctrl==OutPutB)  ?   B:
            (ALU_Ctrl==sll)      ?   B<<A[4:0]:
            (ALU_Ctrl==srl)      ?   B>>A[4:0]:
            (ALU_Ctrl==sra)      ?   $signed($signed(B)>>>A[4:0]):
            (ALU_Ctrl==slt)      ?   ($signed(A)<$signed(B)):
            (ALU_Ctrl==sltu)     ?   ($unsigned(A)<$unsigned(B)):
                                     32'h22225678;
```

4.更改各级流水线的位宽

```
output [2:0] Tnew,
output [3:0] XALU_Ctrl,
output XALU_Start,
output [2:0] ALUoutDataSel,
```

（别忘了在顶层模块修改线的位宽）

上面的步骤不算特别多，但有一个很大的问题。**一错就是全错**

任何一个环节出了问题，比如说，

- ALU的控制信号和控制器里面的控制信号没对上
- 控制信号的线位宽忘了弄上去
- 位扩展器配置错误

那么，算出来的结果就炸了（此处不考虑暂停转发设计的问题）

为什么新增一个指令，需要改动那么多地方？这耦合度也太大了吧？

内部指令总线+自控制的设计

在这样的设计中，CPU是如何组成的呢？

首先看一下译码器：

```
module DecodeUnit(
    input   [31:0]  MipsInstr,
    output [4:0] Shamt,
    output [4:0] Rs,
    output [4:0] Rt,
    output [4:0] Rd,
    output [15:0] Imm16,
    output [25:0] Imm26,
    output      RegWriteEnable,
    output  [4:0]   WriteRegId,
    output  [`INSTRBUS_WIDTH-1:0]   InstrBus
    );
```

可以看到，这个译码器就是一个**纯 译码**。除了把这个信号是什么译出来之外，只额外译了寄存器写使能，以及写寄存器的ID这两个信号。非常干净

```
module ALU(
    input   [31:0]  A,
    input   [31:0]  B,
    input   [4:0]   shamt,
    input   [15:0]  Imm16,
    input   [31:0]  PC,
    input   [`INSTRBUS_WIDTH-1:0]   InstrBus,
    output  [31:0]  C,
    output          OverFlow
    );
```

ALU中接受的信号，也非常的干净

有没有发现**译码器的信号，差不多是直接传入了ALU**

那么控制器在哪呢？

内置于ALU内部

位扩展器又在哪呢？

也在ALU的内部

要写出干净，高效，方便维护的代码。一定要模块化书写，并且尽可能减少耦合。

但是，由于verilog并行化的设计，因此很多时候，一个“模块”不一定需要写成一个module，实际上一个模块很可能就是几行代码。

位扩展模块

```
wire    `INSTR_SET;
assign  {`INSTR_SET}    =   InstrBus;
//////////////////Extention////////////////////////////////
wire    zeroExtend  =   ori|Andi|Xori;
wire    [31:0]  Imm32   =   zeroExtend  ?   {{16'b0},Imm16}:
                            lui         ?   {Imm16,{16'b0}}:
                                            {{16{Imm16[15]}},Imm16};
```

当把传入的信号给解开之后，就可以**直接用指令名去控制电路**

比如上面的4行代码，就能完成了位扩展器、位扩展器控制模块的设计。

并行化，模块化的ALU设计

在ALU需要参加的运算中，实际上大致可以分为这几类：

- 加、减法计算
- 位运算
- 移位运算
- 比较运算（比如小于置1）
- jal运算（存储PC的值）

那么，可以把以上的运算，分别写成一个小“模块”

以位运算为例：

```
//////////////////BitCal Family////////////////////////////////
wire    BC_Family   =   ((ori|Or)|
                        (And|Andi))|
                        ((Xor|Xori)|
                        (Nor|lui));
wire    [31:0]  BC_A    =   A,
                BC_B    =   (ori|Andi|Xori) ?   Imm32   :   B;
wire    [31:0]  BC_Ans  =   (Xor|Xori)  ?   BC_A^BC_B:
                            (ori|Or)    ?   BC_A|BC_B:
                            (And|Andi)  ?   BC_A&BC_B:
                            (lui)       ?   Imm32:
                                            ~(BC_A|BC_B);//Nor
```

在这里，运算器和控制器是同时写了出来的。

既判断是否为位运算类型，又决定了操作数的来源（寄存器还是立即数），同时把运算的结果也算出来了

如法炮制其他的运算类型

最后在ALU的输出口处：

```
////////////////OUT PUT//////////////////////////////////////////
assign  C   =   AS_Family   ?   AS_Ans:
                BC_Family   ?   BC_Ans:
                ST_Family   ?   ST_Ans:
                CMP_Family  ?   CMP_Ans:
                JL_Family   ?   JL_Ans:
                                32'h22222222;//DEBUG
```

可以发现**耦合度极低，结构非常清爽，出BUG易于追踪**

如果出现bug了，可以很容易根据中间的信号，判断是否为异常的指令（ALU结果为222222）

如果没有，那么可以通过XX_Family信号，哪个为高进行判断，当前运算的类型是什么。

然后再根据具体亮起的信号，判断当前执行的指令为什么。

因为每个运算类型，均有A,B两个操作数。并且不同运算之间是相互独立的。所以把BUG控制在了极小的一个范围内。

不过，有多少壮士敢现在改架构呢？嘿嘿嘿