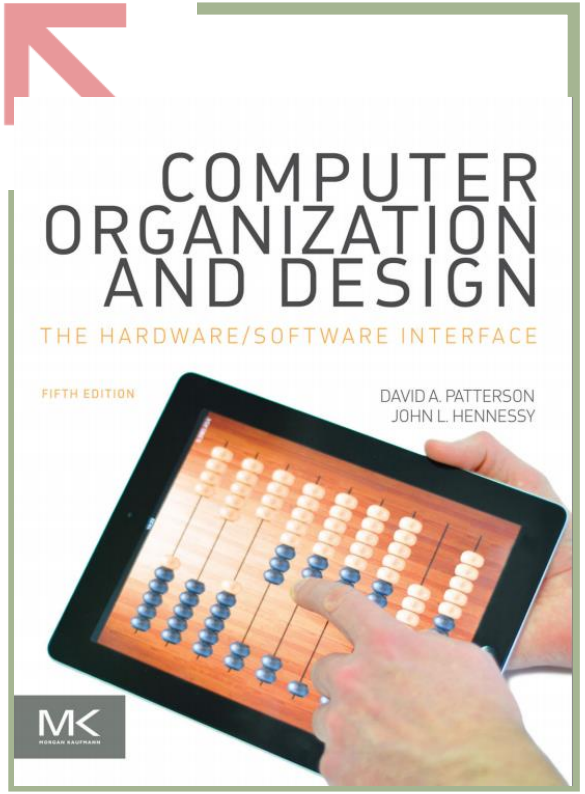
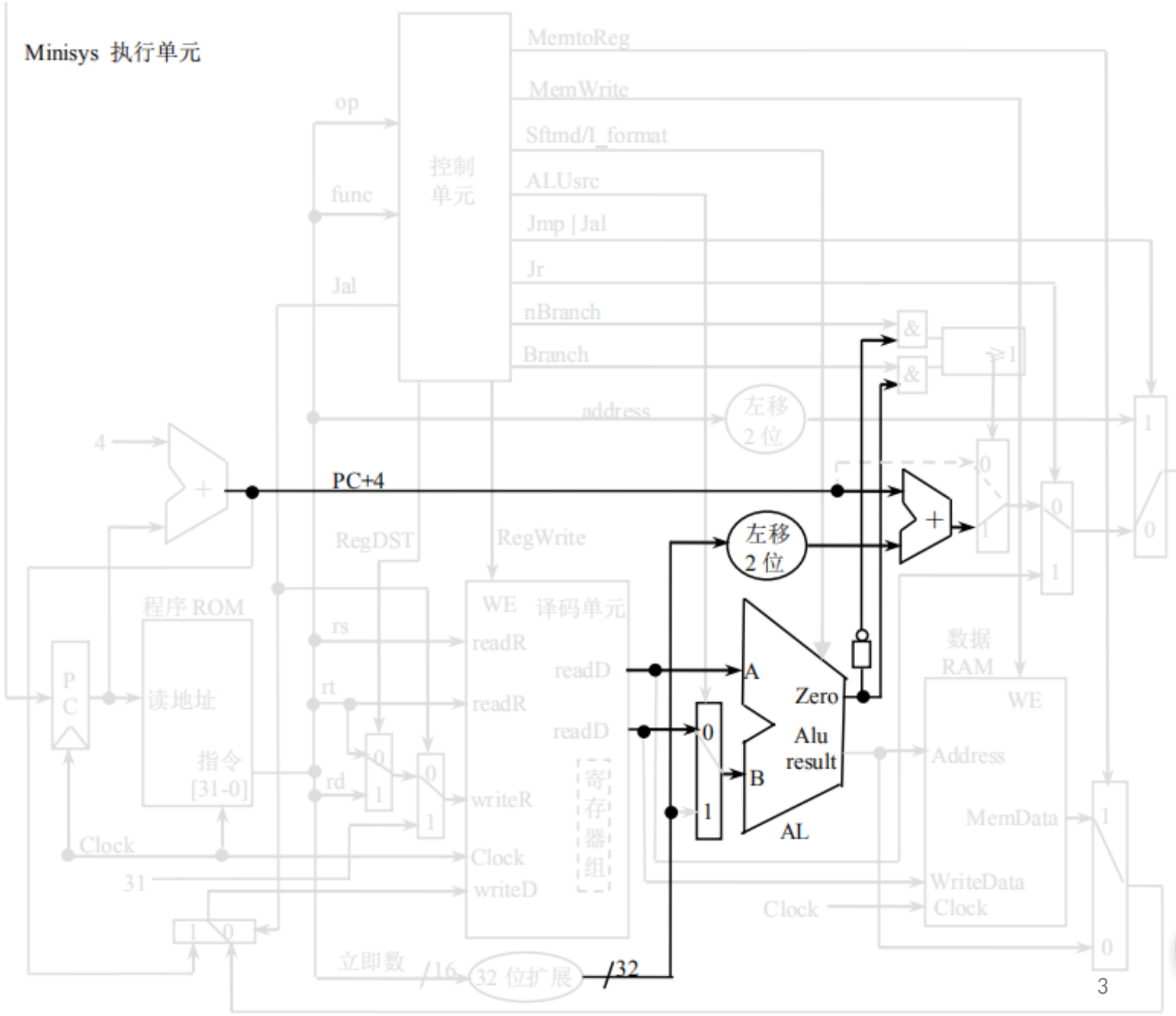
执行单元的设计


COMPUTER
ORGANIZATION
AND DESIGN
THE HARDWARE/SOFTWARE INTERFACE
FIFTH EDITION
DAVID A. PATTERSON
JOHN L. HENNESSY
MK

执行单元设计

- 执行单元的功能及输入、输出端口
- 运算数的选择
- 运算功能的实现
  - 通过组合码实现对运算的控制
  - 运算
    - Arithmetic 算术运算，Logic 逻辑运算
    - Shift 移位运算
    - 特殊运算（slt,lui等）
- 根据操作码执行指定功能，获取最终运算结果作为输出

执行单元设计


Minisys 执行单元
控制单元
op
func
Jal
MemtoReg
MemWrite
Sftmd/I_format
ALUsrc
Jmp | Jal
Jr
nBranch
Branch
address
左移 2 位
4
PC+4
RegDST
RegWrite
程序 ROM
读地址
指令 [31-0]
PC
rs
rt
rd
WE
译码单元
readR
readR
writeR
Clock
writeD
readD
readD
寄存器组
31
立即数
/16
32 位扩展
/32
A
B
Zero
Alu result
AL
数据 RAM
WE
Address
MemData
WriteData
Clock
Clock

执行单元设计

Execute32.v 文件中给出了执行单元所有端口和内部信号的定义：

```
module Executs32(Read_data_1,Read_data_2,Sign_extend, Function_opcode, Exe_opcode, ALUOp,
                 Shamt,ALUSrc,I_format,Zero,Jrn,Sftmd,ALU_Result,Add_Result,PC_plus_4
                 );

    input[31:0]  Read_data_1;                    // 从译码单元的Read_data_1中来
    input[31:0]  Read_data_2;                    // 从译码单元的Read_data_2中来
    input[31:0]  Sign_extend;                    // 从译码单元来的扩展后的立即数
    input[5:0]   Function_opcode;       // 取指单元来的r-类型指令功能码,r-form instructions[5:0]
    input[5:0]   Exe_opcode;            // 取指单元来的操作码，instruction[31:26]
    input[1:0]   ALUOp;             // 运算指令控制编码：{ (R_format || I_format) , (Branch || nBranch) }
    input[4:0]   Shamt;             // 来自取指单元的instruction[10:6]，指定移位次数
    input        Sftmd;           // 来自控制单元的，1表明是移位指令
    input        ALUSrc;          // 来自控制单元，1表明第二个操作数是立即数（beq，bne除外）
    input        I_format;        // 来自控制单元，1表明是除beq, bne, LW, SW之外的I-类型指令
    input        Jrn;             // 来自控制单元，1表明是JR指令
```

执行单元设计

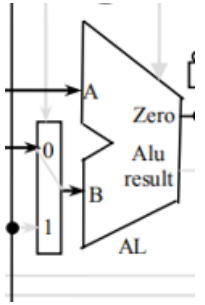
Execute32.v 文件中给出了执行单元所有端口和内部信号的定义：

```
output       Zero;                  // 为1表明计算值为0
output[31:0] ALU_Result;            // 计算的数据结果
output[31:0] Add_Result;            // 计算的地址结果
input[31:0]  PC_plus_4;             // 来自取指单元的PC+4

reg[31:0] ALU_Result;               //执行单元的输出
reg[31:0] ALU_output_mux;           //算术运算与逻辑运算的输出

wire[31:0] Ainput,Binput;           //参与ALU及移位运算的两个运算数
reg[31:0] Sinput;                   //移位运算的结果
wire[32:0] Branch_Add;
wire[2:0] ALU_ctl;
wire[5:0] Exe_code;
wire[2:0] Sftm;
wire Sftmd;
```

运算数据的选择



- 参与运算的两个数据是从运算器的 A 端和 B 端进入，对应的信号为 Ainput 和 Binput。
- B 端情况稍复杂:
  - 除 beq，bne 以外的 I-类型指令，数据都来自扩展的立即数；
  - R-类型及 beq，bne 指令，B 端口的数据来自译码单元的 Read_data_2。下列代码就是分别对 A 端口和 B 端口赋值的相关信号及语句。

```
input[31:0]   Read_data_1;          // 从译码单元的 Read_data_1 中来
input[31:0]   Read_data_2;          // 从译码单元的 Read_data_2 中来
input[31:0]   Sign_extend;        // 从译码单元来的扩展后的立即数
input   ALUSrc; //来自控制单元，表明第二个操作数是立即数（beq，bne 除外）

assign Ainput = Read_data_1 ;      //   为 ALU 的 A 端口赋值
assign Binput = (ALUSrc == 0) ? Read_data_2 : Sign_extend[31:0];
```

执行单元内部的控制信号

- 设计思路：
  - ALU的运算方式多样，控制信号复杂。
  - 为减轻控制单元的负担，由控制模块和实现模块共同产生影响运算的控制信号
- 实现：
  - 一级控制信号：控制单元生成一级控制信号 ALUop（ 描述指令与运算相关的基本特征）
    - bit1 标记是否是R类型或者I类型指令
    - bit0标记是否为beq或bne指令
  - 二级控制信号：实现单元产生二级控制信号Exe_code
    - 根据是否为I类型指令来决定如何生成Exe_code
  - 结合一级与二级控制信号，生成组合码 ALU_ctrl 用于指示运算的执行

执行单元内部的控制信号

- 定义执行码 Exe_code 如下：
  - 当 I_format=1（该指令是除 beq，bne，LW，SW 之外的其他 I-类型指令），执行码等于操作码的低 3 位做 0 扩展到 6 位；
  - 当 I_format=0 （R-类型，beq，bne，LW，SW 指令）执行码等于功能码。

```
input    I_format;  //来自控制单元，表明是除 beq, bne, LW, SW
//之外的 I-类型指令
input[5:0]    Function_opcode; // 取指单元来的 r-类型指令功能码
input[5:0]    Exe_opcode;      // 取指单元来的操作码
wire[5:0] Exe_code;

assign    Exe_code    =    (I_format==0)    ?    Function_opcode    :
{3'b000,Exe_opcode[2:0]};
```

由指令生成的组合码 ALU_ctl[2..0]

```
wire[2:0] ALU_ctl;

//生成一个组合码
assign ALU_ctl[0] = (Exe_code[0] | Exe_code[3]) & ALUOp[1];
assign ALU_ctl[1] = ((!Exe_code[2]) | (!ALUOp[1]));
assign ALU_ctl[2] = (Exe_code[1] & ALUOp[1]) | ALUOp[0];
```

- 逻辑关系

- 对应关系

Exe_code ： // 执行单元中产生的控制码

ALUop ：

//bit1：标记是否是R类型或者I类型指令(1是0否)

//bit0：标记是否为beq或bne指令（1是0否）

Exe_code[3..0]	ALUOp[1..0]	ALU_ctl[2..0]	指令助记符
0100	10	000	and,andi
0101	10	001	or,ori
0000	10	010	add,addi
xxxx	00	010	lw, sw
0001	10	011	addu, addiu
0110	10	100	xor,xori
0111	10	101	nor,lui
0010	10	110	sub, slti
xxxx	01	110	beq，bne
0011	10	111	subu, sltiu
1010	10	111	slt
1011	10	111	sltu

组合码

具有相同运算的指令具有相同的组合码。

重码对于某些指令是没有问题的，因为他们的功能就是运算，而运算也完全一样，只是数据来源有所不同，比如 and 与andi，addu 与 addui 等。

但并不是所有**重码**的指令都是**功能**完全相同，必须加以区别：

1：指令的组合码相同，数据计算功能一样，**无需区别**

add 与 addi，
addu 与 addiu，
and 与 andi，
or 与 ori，
xor 与 xori，
slt 与 sltu 与sltiu

Exe_code[3..0]	ALUOp[1..0]	ALU_ctl[2..0]	指令助记符
0100	10	000	and,andi
0101	10	001	or,ori
0000	10	010	add,addi
xxxx	00	010	lw, sw
0001	10	011	addu, addiu
0110	10	100	xor,xori
0111	10	101	nor,lui
0010	10	110	sub, slti
xxxx	01	110	beq，bne
0011	10	111	subu, sltiu
1010	10	111	slt
1011	10	111	sltu

组合码

- lw，sw 指令的组合码和 add 相同：
  - lw 与 sw 两指令的主要功能在译码单元完成，而执行单元只是负责计算地址，所以在执行部件中，其运算功能和 add 指令等同，都是做加法。

Exe_code[3..0]	ALUOp[1..0]	ALU_ctl[2..0]	指令助记符
0100	10	000	and,andi
0101	10	001	or,ori
0000	10	010	add,addi
xxxx	00	010	lw, sw
0001	10	011	addu, addiu
0110	10	100	xor,xori
0111	10	101	nor,lui
0010	10	110	sub, slti
xxxx	01	110	beq，bne
0011	10	111	subu, sltiu
1010	10	111	slt
1011	10	111	sltu

组合码

- beq，bne 指令的组合码和 sub 相同：
  - 执行结果赋值对象不同，执行过程相同：beq 和 bne 在执行单元做完减法后是赋值给 add_result(地址结果）而不是 alu_result(计算结果)，所以不影响结果
- sub 与 slti，subu 与 sltiu 组合码相同：
  - 执行结果赋值对象不同，执行过程相同：都作减法，但 slti 和 sltiu 指令作完减法后的赋值不同于 sub, subu，可用 **I_format** 是否为 1 **区别**这两类指令。
- subu 和 slt 以及 sltu 组合码相同：
  - 执行结果赋值对象不同，执行过程相同：都需要作减法，但可以利用 slt与 sltu 的 **Function_opcode**(3)=‘1’来**区别**它们。

Exe_code[3..0]	ALUOp[1..0]	ALU_ctl[2..0]	指令助记符
0100	10	000	and,andi
0101	10	001	or,ori
0000	10	010	add,addi
xxxx	00	010	lw, sw
0001	10	011	addu, addiu
0110	10	100	xor,xori
0111	10	101	nor,lui
0010	10	110	sub, slti
xxxx	01	110	beq，bne
0011	10	111	subu, sltiu
1010	10	111	slt
1011	10	111	sltu

组合码

- 还有一些指令也和别的指令重码，但却没有和这些指令同样的运算功能，因此一定要确定它们不会产生错误的结果，这些指令包括：

  - 移位指令。所有的移位指令可以用 sftmd＝’1’来区别
  - lui 指令。nor 与 lui 的组合码相同，可用 I_format 是否为 1 来区别之。
  - jr 指令。jr 指令利用 jrn＝’1’来区别，该指令的执行在取指单元中。
  - j 指令。j 指令利用 jmp＝’1’来区别，该指令的执行在取指单元中。
  - jal 指令。jal 指令利用 jal＝’1’来区别，该指令的执行在取指单元和译码单元中。

练习1：算术及逻辑运算功能的实现

- 请结合组合码的分析，完成Excute32.v设计中基本的算术及逻辑运算功能

```
reg[31:0] ALU_output_mux;
always @(ALU_ctl or Ainput or Binput)  // 处理各类运算
begin
case(ALU_ctl)
    3'b000:ALU_output_mux =? ? ?
    3'b001:ALU_output_mux =? ? ?
    3'b010:ALU_output_mux =? ? ?
    3'b011:ALU_output_mux =? ? ?
    3'b100:ALU_output_mux =? ? ?
    3'b101:ALU_output_mux =? ? ?
    3'b110:ALU_output_mux =? ? ?
    3'b111:ALU_output_mux =? ? ?
    default:ALU_output_mux = 32'h00000000;
endcase
end
```

Exe_code[3..0]	ALUOp[1..0]	ALU_ctl[2..0]	指令助记符
0100	10	000	and,andi
0101	10	001	or,ori
0000	10	010	add,addi
xxxx	00	010	lw, sw
0001	10	011	addu, addiu
0110	10	100	xor,xori
0111	10	101	nor,lui
0010	10	110	sub, slti
xxxx	01	110	beq，bne
0011	10	111	subu, sltiu
1010	10	111	slt
1011	10	111	sltu

完成移位运算

- 共有 6 条移位指令，包括左移指令 2 条，右移指令 4 条。
- wire[31:0] Ainput, Binput 是参与移位运算的操作数

sftm[2:0]	process
3‘b000	sll rd, rt, shamt
3‘b010	srl rd, rt, shamt
3‘b100	sllv rd, rt, rs
3‘b110	srlv rd, rt, rs
3‘b011	sra rd, rt, shamt
3‘b111	srav rd, rt, rs
other	not shift

Execute32.v 文件中与移位运算相关的信号包括：

```
input        Sftmd;      //来自控制单元的，表明是移位指令
input[4:0]    Shamt; //来自取指单元的 instruction[10:6]，指定移位次数
input[5:0]    Function_opcode;      //r-form instructions[5:0]
wire[2:0]    Sftm;                 //移位指令类型
reg[31:0]    Sinput;                //移位指令的最后结果
assign Sftm = Function_opcode[2:0];
```

练习2：完成 6 个移位操作

请参考sftm的编码，以及Ainput,Binput 在指令中的位置补全以下移位操作的实现代码：

```
always @* begin  // 6 种移位指令
   if(Sftmd)
    case(Sftm[2:0])
        3'b000:Sinput = ? ? ?              //Sll rd,rt,shamt
        3'b010:Sinput = ? ? ?              //Srl rd,rt,shamt
        3'b100:Sinput = ? ? ?           //Sllv rd,rt,rs
        3'b110:Sinput = ? ? ?           //Srlv rd,rt,rs
        3'b011:Sinput = ? ? ?             //Sra rd,rt,shamt
        3'b111:Sinput = ? ? ?          //Srav rd,rt,rs
        default:Sinput = Binput;
    endcase
   else Sinput = Binput;
end
```

完成运算结果的输出

执行模块的运算结果有以下几种情况：

- 1） 完成比较后赋值（ slt、sltu、slti 及sltiu ）
  - 得到ALU_output_mux，到最终输出还需要进行转换
- 2） 完成 lui 指令运算
- 3）完成移位运算
  - 得到Sinput
- 4）完成基本算术/逻辑运算
  - 得到ALU_output_mux即为最终输出

可以使用 Exe_code[3..0]、ALUOp[1..0]和 ALU_ctl[2..0]的组合确定所执行的操作，并明确最终结果的选择。

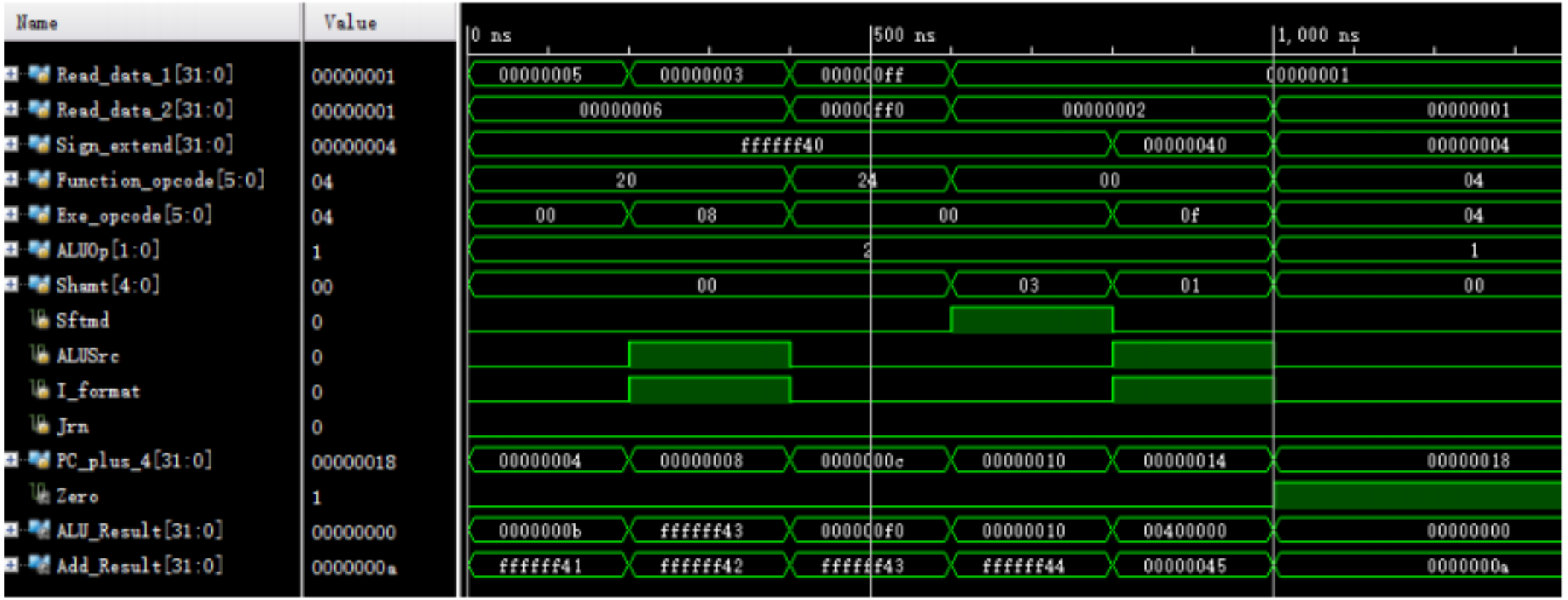
练习3：完成运算结果输出

- execute32.v 文件中给出了下面的运算结果输出相关的信号以及部分代码，请在此基础之上实现执行单元运算结果ALU_Result的赋值操作

```
output[31:0] ALU_Result;      // 计算的数据结果
input  I_format;  // 来自控制单元，表明是除 beq，bne，LW，SW 之外的 I-类型指令;

always @* begin
if(((ALU_ctl==3'b111) && (Exe_code[3]==1))
        ||((ALU_ctl[2:1]==2'b11) && (I_format==1)))
    ALU_Result = ? ? ?              // 处理所有 SLT 类的问题
        else if((ALU_ctl==3'b101) && (I_format==1))
           ALU_Result[31:0] = ? ? ?       //lui data
        else if(Sftmd==1) ALU_Result = ? ? ?          //移位
        else  ALU_Result = ALU_output_mux[31:0];      //otherwise
    end
```

模块仿真：详见exe_sim.v



模块仿真：详见exe_sim.v

时刻 ns	执行指令	结　果（十六进制）
0	add	ALU_Result = 0000000b（5+6），Zero=0
200	addi	ALU_Result =ffffff43（ffffff40+3），Zero=0
400	and	ALU_Result =000000f0（000000ff&00000ff0），Zero=0
600	sll	ALU_Result =00000010（00000002<<3），Zero=0
800	lui	ALU_Result =00400000（00000040<<16），Zero=0
1000	beq	Add_Result =0000000A（00000018>>2+00000004），Zero=1