4指令集和ALU设计

英语

1. *issue* 问题、发表

4.1 指令集

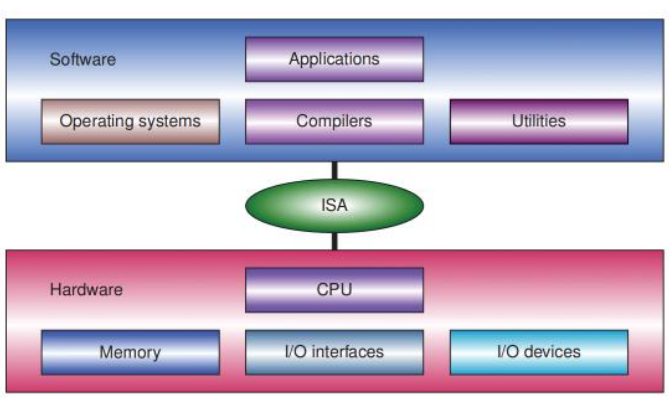


Figure 4.1 ISA as an interface between software and hardware

指令级是硬件以及软件设计中的一个重要环节，指令集的存在可以告诉编译开发者“CPU可以做什么”，告诉CPU设计者“CPU应该做什么”。[注释1]——**指令集是软硬件一个重要的交界面**

指令集定义了指令的格式、指令的操作、操作数的类型以及指令能访问的存储器和寄存器地址、字节顺序(大小端)和寻址方式

> *ISA is an important issue in hardware/software codesign. An ISA tells compiler developers “what a CPU can do,” and tells CPU designers “what a CPU should do.” That is, an ISA is an interface between software and hardware, as shown in Figure 4.1. An ISA defines the formats of instructions, the operations of instructions, the types of operands, the memory and registers the instructions can access, the byte ordering, and the addressing modes.*

4.1.1 操作数类型

操作数可以是位、字节、半字、字以及双字。**CPU中字节的位数始终是8位，但是字的位数取决于该CPU的位处理能力**

> *A byte is always 8 bits but the length of a word depends on the bit-processing ability of the CPU: for example, it has 16 bits in the 8086 CPU. Here we use 32 bits as the length of a word*

Table 4.1 Operand types

Operand type	Bits	Value range	Corresponding to C
Byte	8	−128 to +127	signed char
Unsigned byte	8	0 to 255	unsigned char
Half word	16	−32,768 to +32,767	short int
Unsigned half word	16	0 to 65,535	unsigned short int
Word	32	−2,147,483,648 to +2,147,483,647	int
Unsigned word	32	0 to 4,294,967,295	unsigned int
Single-precision FP	32		float
Double-precision FP	64		double

对于一个32位的CPU，如果操作数不是32位的，那么需要进行位扩展扩展为32位

如果该操作数是无符号数，那么则进行0扩展

如果该操作数是有符号数，那么则按符号位进行扩展

4.1.2大端和小端

大端和小端定义了一个字的字节序列在内存中的存放顺序

大端：数据的高位MSB放在内存的低位

小端：数据的低位LSB放在内存的低位

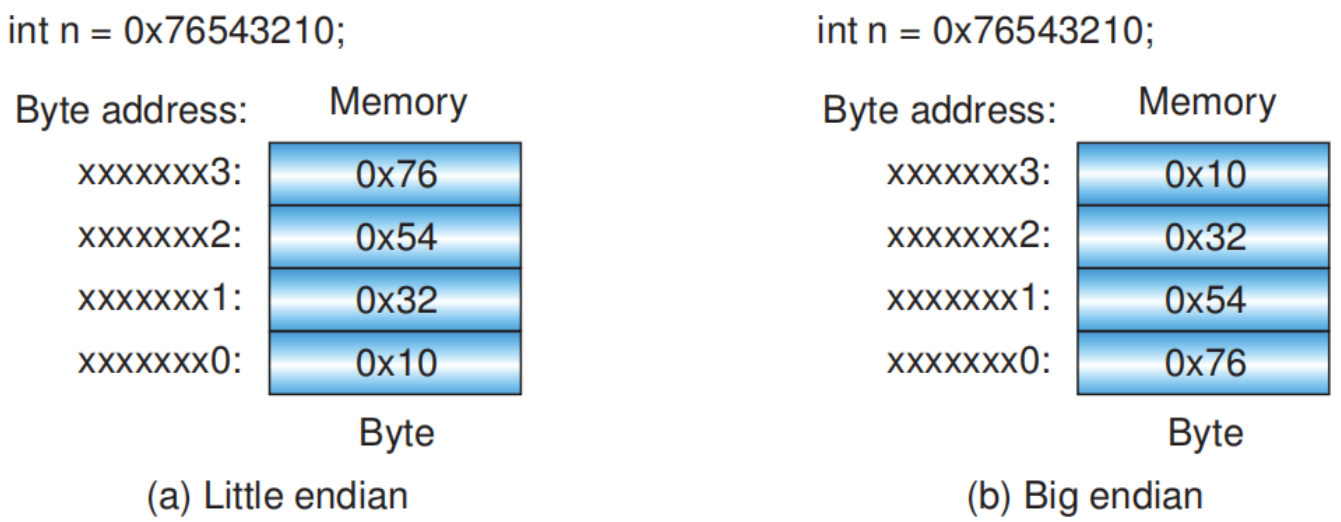


Figure 4.2 Little endian and big endian

检测机器大小端的方式

1. C语言指针point

```c
int main () { // endian_pointer.c
  int n = 0x76543210; // a word
  if (*(unsigned char *)&n == 0x10) // char starting address
      printf("little endian\n"); // stored 0x10
  else
      printf("big endian\n"); // stored 0x76
}
```

2. C语言组合Union

```c
int main() { // endian_union.c
  union { // union:
      int intword; // to assess a word
      unsigned char characters[sizeof (int)]; // with different names
  } u; // union u
  u.intword = 0x76543210; // access as an integer
  if (u.characters[sizeof (int) - 1] == 0x76) // access as array of byte
```

```
        printf("little endian\n"); // high byte in high address
    else
        printf("big endian\n"); // high byte in low address
}
```

c

内存对齐

字节是内存访问和存储的最小单位，它可以存储在内存的任何地方➡都可以一次读出

但是对于半字、字、双字这些数据，如果连续存储（不采用对齐），那么会有部分数据可能，会占用内存的不同字（**CPU可一次读取一个存储字长的存储单元，不在一个存储单元(跨单元存储)，则需要多次读取**）。为了**加快读取，则需要采用内存对齐技术，将数据放在2的幂的起始地址上开始存放，当前存储单元剩余位数不够存储时，则直接空缺，放在下一个存储单元**

如果CPU不支持没对齐的内存访问，但访问了没有对齐的地址，会出现**对齐异常**

> *A byte is the smallest unit of memory access. It can be stored at any location of the memory. For other data types, there is a problem of data alignment. The data alignment refers to where the data is located in the memory. Referring to Figure 4.3, a memory address a is said to be n-byte aligned when n is a power of 2 and a is a multiple of n bytes. Aligned data can be accessed faster. For example, the 4 bytes of an aligned word can be written to or read from memory with one access by using the word address (all the 4 bytes have a same word address). On the other hand, we must access to memory two times if the word is unaligned. Some ISAs allow the unaligned data placement. If it is not allowed and there is an unaligned access, an alignment fault exception will occur.*

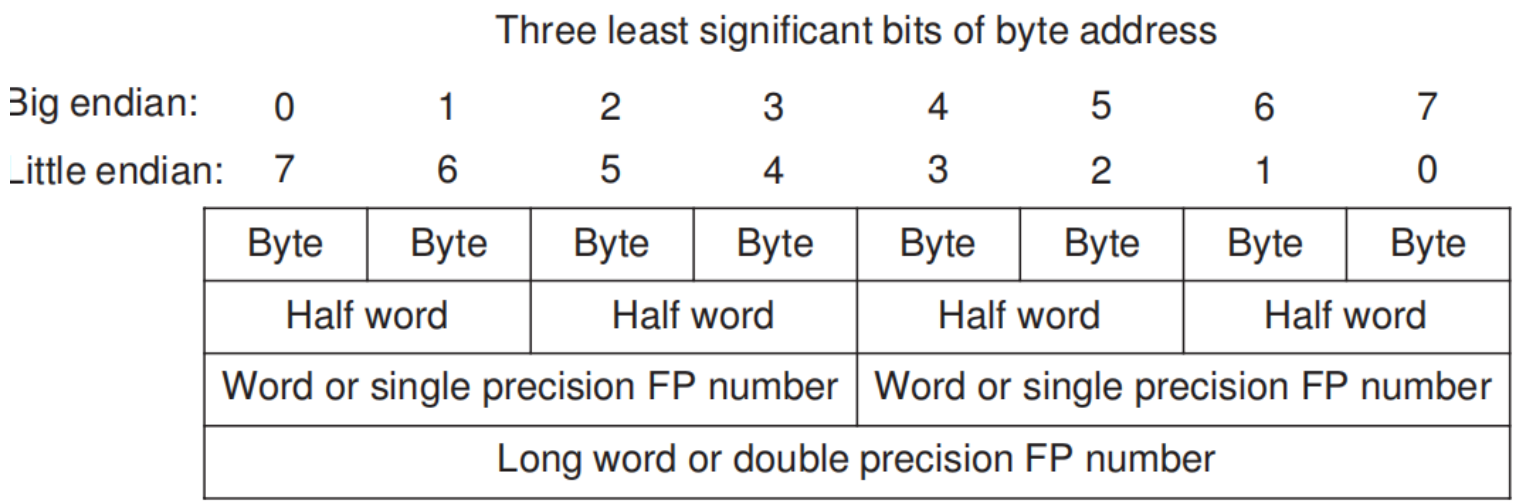


Figure 4.3 Data alignment in memory locations

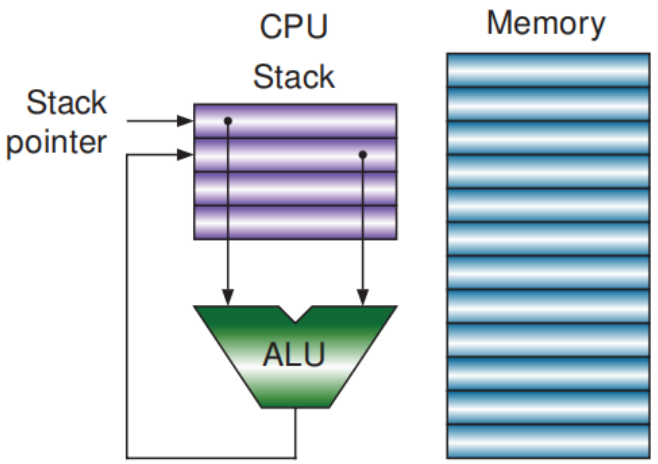
4.1.3指令的操作类型

之前在第一章已经介绍过指令的操作类型→指令能进行什么操作：

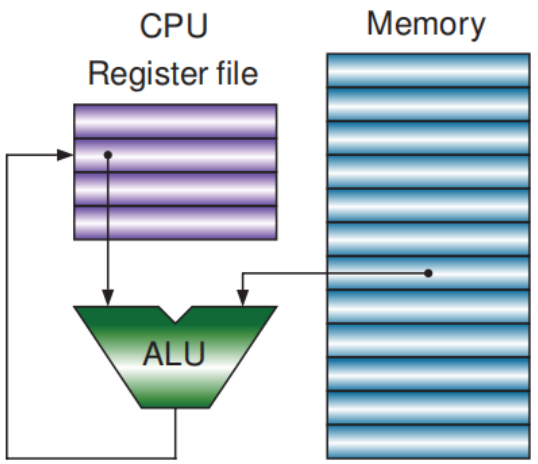
1. 整数的算术运算：+、-、*、/、根号
2. 逻辑运算：and、or、not、xor、nor

   大多数指令集提供了xor或者nor来代替not→not x==xor x,1、nor x,0
3. 移位运算：<<、>>、>>>
4. 内存访问：RISC只允许load-store指令，CISC则可以使用计算指令直接访存
5. 控制转移：分支跳转beq、bne，无条件跳转j、jr、jal，中断返回eret，子程序调用call
6. 输入输出访问：端口统一编址load-store指令，独立编址input、output
7. 系统控制
8. 浮点数运算

4.1.4指令的体系结构



(a) Stack architecture



(c) Register memory architecture

Figure 4.4 Instruction arcl

之前在第一章已经介绍过指令的类型：堆栈型、累加器型、寄存器-存储器型、寄存器-寄存器型

Table 4.2 Code examples of instruction architectures

Stack		Accumulator		Register–mem	
push	x	load	x	load	r1,
push	y	**add**	**y**	**add**	**r1,**
add		store	z	store	z,
pop	z				

4.1.5寻址方式

寄存器直接寻址→操作数位于寄存器中，指令给出寄存器的编号

```
add r3,r1,r2;r1+r2→r3
```

立即数寻址→操作数直接位于指令的某些位中

```
addi r3,r1,-1;r1-1→r3
```

存储器直接寻址→操作数位于存储器中，指令给出存储器的地址

```
add r3,r1,[0x1234]→r1+mem[0x1234]→r3
```

寄存器间接寻址→操作数位于存储器中，存储器的地址位于寄存器中，指令给出这个寄存器的编号

add r3,r1,(r2)→r1+mem[reg[r2]]→r3

寄存器偏移寻址→操作数位于存储器中，存储器的地址需要使用偏移地址和给定寄存器内容相加得到

add r3,r1,0x1234(r2)→r1+mem[reg[r2]+0x1234]→r3

4.2 MIPS指令格式和寄存器

📌MIPS指令有两种版本：**32位操作数的MIPS32和64位操作数的MIPS64**

之后的学习都是基于MIPS32

4.2.1 MIPS指令的格式

MIPS指令集存在3种指令格式：**R型，I型，J型。三种类型均为32位**，其具体格式如下图：

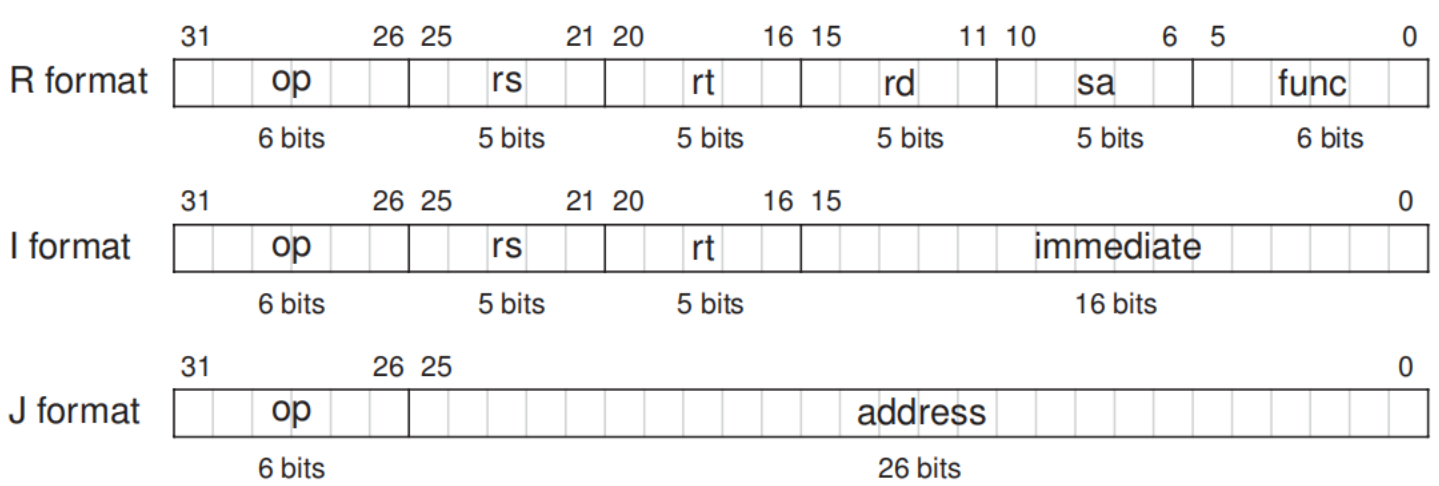


Figure 4.5 MIPS instruction formats

R型指令包括**不含立即数的算术逻辑运算以及移位运算**

op字段为000000，指令的具体操作由func[5:0]字段决定。rs、rt、rd为寄存器的地址编号(32个寄存器用5位)。sa为移位指令的移位数，移rt置于rd中。

> *The op of the R (register) format instructions is 0, and the operation of an instruction is specified by func. Each of rs, rt, and rd is a 5-bit register number. Shift instructions use sa to specify the shift amount (a constant). The content in register rt will be shifted, and the result of the shift will be written to register rd. Other R format instructions read two source operands from registers rs and rt, respectively, operate on the two operands, and write the result to the register rd.*

I型指令包括**所有的立即数运算、分支指令和lw、sw指令**

立即数运算是将**rt与符号扩展后的imm运算写结果到rs**中

分支指令是**比较rs与rt，若满足分支条件，则跳转到立即数与pc所组成的分支目标地址处**

lw、sw指令则是**将rs与扩展后的立即数的运算结果作为访存地址，写rt到对应单元或者从对应单元读数据到rt**

imm的符号扩展是采用无符号还是有符号取决于所做的运算，**如果是算术运算或者lw、sw、分支指令则是有符号扩展；如果是逻辑运算则是无符号扩展**

> *The computational instructions of the I (immediate) format use immediate as the second source operand. The 16-bit immediate must be sign- or zero-extended into 32 bits. The first source operand is in register rs, and the result is written to register rt.*
> *The conditional branch instructions of the I format compare the two operands located in registers rs and rt, respectively, and determine whether to branch to the target address. immediate is a signed word-offset and is used to calculate the branch target address.*
> *The load instructions of the I format load memory data to the register rt.*
> *The store instructions store the register rt data to the memory. For both load and store instructions, the memory address is the sum of the sign-extended immediate and the operand in the register rs.*

J型指令包括所有的无条件跳转指令：J、Jr、Jal

J指令的跳转地址是{pc_add4[31:28],imm[25:0]<<2}

jr指令的跳转地址存在于给定编号的寄存器中（r型指令）

Jal则先需要保存pc_add4到31号寄存器，然后跳转到{pc_add4[31:28],imm[25:0]<<2}

4.2.2 MIPS通用寄存器组

Table 4.3 MIPS general-purpose registers

Register name	Register number	Use
$zero	0	Constant 0
$at	1	Assembler temporary
$v0 to $v1	2 to 3	Function return value
$a0 to $a3	4 to 7	Function parameters
$t0 to $t7	8 to 15	Temporaries
$s0 to $s7	16 to 23	Saved temporaries
$t8 to $t9	24 to 25	Temporaries
$k0 to $k1	26 to 27	Reserved for OS kernel
$gp	28	Global pointer
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Return address

左图即为MIPS32的32个寄存器，注意0号并不是一个寄存器，它的内容总是0，可以看成直接接地的导线

4.3 MIPS 指令

后面的学习主要是表4.4中的20条整数指令、以及表4.7中的中断/异常指令，TLB指令和浮点运算指令

Table 4.4 Twenty MIPS integer instructions

Inst.	[31:26]	[25:21]	[20:16]	[15:11]	[10:6]	[5:0]	Meaning
add	000000	rs	rt	rd	00000	100000	Register add
sub	000000	rs	rt	rd	00000	100010	Register subtract
and	000000	rs	rt	rd	00000	100100	Register AND
or	000000	rs	rt	rd	00000	100101	Register OR
xor	000000	rs	rt	rd	00000	100110	Register XOR
sll	000000	00000	rt	rd	sa	000000	Shift left
srl	000000	00000	rt	rd	sa	000010	Logical shift right
sra	000000	00000	rt	rd	sa	000011	Arithmetic shift right
jr	000000	rs	00000	00000	00000	001000	Register jump
addi	001000	rs	rt		Immediate		Immediate add
andi	001100	rs	rt		Immediate		Immediate AND
ori	001101	rs	rt		Immediate		Immediate OR
xori	001110	rs	rt		Immediate		Immediate XOR
lw	100011	rs	rt		offset		Load memory word
sw	101011	rs	rt		offset		Store memory word
beq	000100	rs	rt		offset		Branch on equal
bne	000101	rs	rt		offset		Branch on not equal
lui	001111	00000	rt		immediate		Load upper immediate
j	000010			address			Jump
jal	000011			address			Call

Table 4.7 MIPS instructions related to interrupt/exception, TLB, and FP

Inst.	[31:26]	[25:21]	[20:16]	[15:11]	[10:6]	[5:0]	Meaning
syscall	000000	00000	00000	00000	00000	001100	System call
eret	010000	10000	00000	00000	00000	011000	Return from exception
tlbwi	010000	10000	00000	00000	00000	000010	Write indexed TLB entry
tlbwr	010000	10000	00000	00000	00000	000110	Write random TLB entry
mfc0	010000	00000	rt	rd	00000	000000	Load control word
mtc0	010000	00100	rt	rd	00000	000000	Store control word
lwc1	110001	rs	ft		offset		Load FP word
swc1	111001	rs	ft		offset		Store FP word
add.s	010001	10000	ft	fs	fd	000000	FP add
sub.s	010001	10000	ft	fs	fd	000001	FP subtract
mul.s	010001	10000	ft	fs	fd	000010	FP multiply
div.s	010001	10000	ft	fs	fd	000011	FP division
sqrt.s	010001	10000	00000	fs	fd	000100	FP square root

其它主要的MIPS汇编命令见MIPS汇编以及AsmSim

4.4 ALU设计

这一部分实现一个实现行内引用的算术逻辑运算指令的ALU，因此ALU必须要支持的运算功能有：

1. *ADD (addition) for instructions of add, addi, lw, and sw;*
2. *SUB (subtraction) for instructions of sub, beq, and bne;*
3. *AND (bitwise logical and) for instructions of and and andi;*
4. *OR (bitwise logical or) for instructions of or and ori;*
5. *XOR (bitwise logical exclusive or) for instructions of xor and xori;*
6. *LUI (load upper immediate) for lui instruction;*
7. *SLL (shift left logical) for sll instruction;*
8. *SRL (shift right logical) for srl instruction;*
9. *SRA (shift right arithmetic) for sra instruction;*

理想的ALU是基于C2、C3所实现的组件，使用指令译码后的控制信号来选择结果输出，其电路结构图如下：

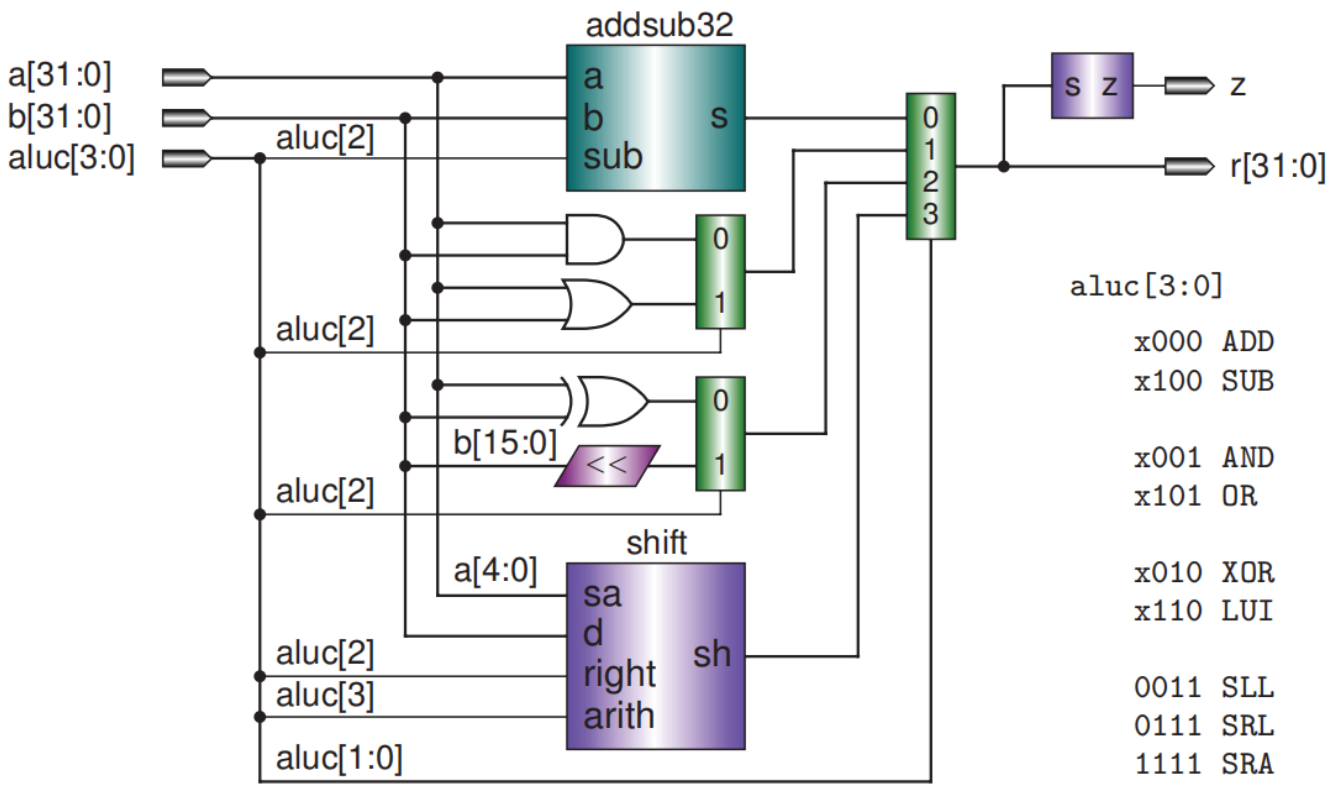


Figure 4.13 Schematic diagram of ALU

```
`timescale 1ns / 1ps

module alu(
    input [31:0]a,b,//a端多路选择pc(pc的adder可以放在IF),sa,reg_a;b端多路选择
reg_b,imm,4(pc_adder),imm<<2;
    input [3:0]aluc,
    //x000 ADD,x100 SUB,x001 AND,x101 OR,x010 XOR,x110 LUI,0011 SLL,0111 SRL,1111 SRA
    output wire[31:0]res,
    output wire z
);
    reg [31:0]s0,s1,s2,s3;

    always @(*) begin
        casex (aluc)//带有x的匹配
            4'bx000:s0=a+b;
```

```
            4'bx100:s0=a-b;
            4'bx001:s1=a&b;
            4'bx101:s1=a|b;
            4'bx010:s2=a^b;
            4'bx110:s2={b[15:0],16'b0};
            4'b0011:s3=b<<a[4:0];
            4'b0111:s3=b>>a[4:0];
            4'b1111:s3=$signed(b)>>>a[4:0];
        endcase
    end
    assign z=(a==b)?1:0;
    mux4to1 mux4to1_inst(s0,s1,s2,s3,aluc[1:0],res);
endmodule

module mux4to1(
    input [31:0]s0,s1,s2,s3,
    input [1:0]sel,

    output reg[31:0]res
);
    always @(*) begin
        case (sel)
            2'b00:res=s0;
            2'b01:res=s1;
            2'b10:res=s2;
            2'b11:res=s3;
        endcase
    end
endmodule
```

Verilog

```
`timescale 1ns / 1ps

module alu_test();
    reg [31:0]a,b;//a绔   璺  鍙戝嫙pc(pc鑶劮dder鍙  互鏀惧湪IF),sa,reg_a;b绔   璺  鍙戝嫙
reg_b,imm,4(pc_adder),imm<<2;
    reg [3:0]aluc;
    //x000 ADD,x100 SUB,x001 AND,x101 OR,x010 XOR,x110 LUI,0011 SLL,0111 SRL,1111 SRA
    wire[31:0]res;
    wire z;

    initial begin
        //ADD
        aluc=4'b0000;a=32'd4;b=32'd7;
        #5;aluc=4'b0000;a=-32'd5;b=-32'd10;
        #5;aluc=4'b0000;a=-32'd5;b=32'd6;
        //sub
        #5;aluc=4'b0100;a=32'd5;b=32'd7;
        #5;aluc=4'b0100;a=-32'd5;b=-32'd10;
        #5;aluc=4'b0100;a=-32'd5;b=32'd6;
        //AND
        #5;aluc=4'b0001;a=32'd5;b=32'd7;
        //OR
        #5;aluc=4'b0101;a=32'd5;b=32'd7;
        //XOR
        #5;aluc=4'b0010;a=32'd5;b=32'd7;
        //LUI
        #5;aluc=4'b0110;a=32'd7;b=32'd25;
        //SLL
        #5;aluc=4'b0011;a=32'd3;b=32'd13;
        //SRL
        #5;aluc=4'b0111;a=32'd5;b=32'd37;
        //SRA
```

```
        #5;aluc=4'b1111;a=32'd5;b=32'd37;
        #5;aluc=4'b1111;a=32'd7;b=-32'd26;
        #5;$finish;

    end
    alu  alu_inst (
        .a(a),
        .b(b),
        .aluc(aluc),
        .res(res),
        .z(z)
    );
endmodule
```

Verilog

[注释1] 这里我的理解是，CPU设计者需要实现CPU，需要给出一些必须由CPU所完成的东西，因此是CPU一个根做什么。而告诉编译开发者是“CPU可以做什么”即编译开发者可以利用已实现的CPU的功能来进行进一步的开发