计算机的硬件视角

计算机的数学视角

状态机模型

什么是状态机

状态机是一种数学模型，简单来说，就是一个机器有几种状态，每次给一个输入，这个机器就从一个状态跳转到另一个状态。

举一个简单的例子，一条狗，看到人就会叫，叫的时候如果人给它一根香肠，它会吃香肠不再叫了，那么这个狗就是一个简单的状态机：

- **安静状态** → *看到人* → **吠叫状态**
- **吠叫状态** → *人给香肠* → **进食状态**
- **进食状态** → *吃完香肠* → **安静状态**

（是的这只狗吃完香肠看到同一个人还是会叫。）

计算机的状态机视角

计算机是个状态机

计算机的状态是什么呢？就是各个寄存器的值、内存的值、外部设备等等的集合，每次有新的外部设备输入，转移到下一个状态，也就是设置寄存区、内存等等的值。

程序是个状态机

如果把计算机看成一个状态机， 那么运行在计算机上面的程序又是什么呢?

我们知道程序是由指令构成的，那么我们先看看一条指令在状态机的模型里面是什么。不难理解，计算机正是通过执行指令的方式来改变自身状态的，比如执行一条加法指令，就可以把两个寄存器的值相加，然后把结果更新到第三个寄存器中；如果执行一条跳转指令，就会直接修改PC的值，使得计算机从新PC的位置开始执行新的指令。所以在状态机模型里面，指令可以看成是计算机进行一次状态转移的输入激励。

指令集(Instruction set Architecture, ISA)

CPU (Central Processing Unit，中央处理器)，是现代计算机的核心部件，它就像计算机的大脑一样，处理输入的各种指令。但是，CPU如何将要做什么事情和输入的指令相对应呢？这就需要指令集，指令集是我们和CPU的一种约定，比如我们说"mv a0, a1"，CPU就知道应该把 a1 的值给 a0。

指令集有很多，比如我们的Intel和AMD的CPU的笔记本电脑使用指令集x86_64，我们的手机通常使用Arm指令集等等。

> **Tip**
> ISA 是软件与硬件交互的接口
>
> 我们的 CPU 是基于给定的 ISA 去设计的（比如你手上的电脑大概率是 x64, 手机大概率是 arm），然后在你这个设备上的汇编语言也就是这个 ISA 独有的，你的程序经过在不同的设备上会通过不同的编译器变成对应 ISA 的汇编指令，这样才能准确的与 CPU 交互。

汇编语言和机器语言

我们知道，我们的计算机实际只能阅读二进制的内容，要想让CPU这个仆人做事情，我们只能用二进制的特殊语言使唤他。

让我们看看第二节课最开始的例子，这是CPU可以读懂的机器语言

```
00000040: 4111 06e4 22e0 0008 1705 0000 1305 0500
00000050: 9700 0000 e780 0000 8147 3e85 a260 0264
00000060: 4101 8280
```

全部都是数字，这就是CPU可以读懂的内容，也就是机器语言。但是显然这不是大部分人类可以直接编写的（虽然我们的先驱最开始就是把这些数组打孔在纸袋上面来和计算机交流的），所幸，现在有很多工具，可以帮助我们把我们编写的汇编语言翻译为机器语言。

汇编语言就是一种助记符，我们可以使用**汇编器**将我们编写的汇编语言几乎一对一的转化为机器语言的几个字节，再丢给CPU运行。

```
.LC0:
        .string "Hello World!\n"
main:
    addi    sp,sp,-16
    sd      ra,8(sp)
    sd      s0,0(sp)
    addi    s0,sp,16
    lla     a0,.LC0
    call    puts@plt
    li      a5,0
    mv      a0,a5
    ld      ra,8(sp)
    ld      s0,0(sp)
    addi    sp,sp,16
    jr      ra
```

这就是汇编语言。

当然，我们现在都编写更加高级的语言，比如C语言，由编译器将它们翻译为机器语言。

我们可以来简单地体会这个过程。

- 新建一个文件main.c，编写一个最简单的HelloWorld程序。

```
#include <stdio.h>

int main() {
    puts("Hello World!\n");
    return 0;
}
```

- 使用编译器gcc的`-S`参数告诉gcc，我们需要将它翻译为汇编语言：

```
gcc ./main.c -S ./main.s
```

- 接下来打开文件main.S，就可以看到我们编写的c代码的汇编语言。

```
    .file    "main.c"
    .text
    .section    .rodata
.LC0:
    .string    "Hello World!\n"
    .text
    .globl    main
    .type    main, @function
main:
.LFB0:
    .cfi_startproc
    pushq    %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    leaq    .LC0(%rip), %rax
    movq    %rax, %rdi
    call    puts@PLT
    movl    $0, %eax
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size    main, .-main
    .ident    "GCC: (Debian 12.2.0-14) 12.2.0"
    .section    .note.GNU-stack,"",@progbits
```

- 接下来，我们将它编译为机器语言，让它可以直接运行。

```
gcc ./main.S -o ./main
```

在命令行运行`main`，可以看到输出了`Hello World!`。

> **Tip**
> `main`并不只包含了机器语言，还包含了指示系统如何运行这个文件的信息。

图灵机

有了强大的CPU，计算机该如何让他忠实的运转呢?

答案是图灵机，我们可以用一个很简单的模型让CPU只干它擅长的事情-**计算**。你已经知道汇编语言是什么样子了，为了记录我们的程序跑到哪里了，我们引入一个寄存器 叫做 PC (Program Counter，在x86上，它叫做EIP)，他的值就代表当前命令的位置。然后事情就变得简单了。

从此以后，计算机只需要做一件事情:

```
while (1) {
  从PC指示的存储器位置取出指令;
  执行指令;
  更新PC;
}
```

这就是一个足够简单的计算机模型， 开拓者图灵在1936年就已经提出“类似的核心思想”。 而这个流传至今的核心思想, 就是"存储程序"。为了表达对图灵的敬仰, 我们也把上面这个最简单的计算机称为"图灵机"（Turing Machine, TRM）。

一个简单的图灵机：

- 结构上，有存储器，有PC，有加法器。
- 工作方式上, TRM不断地重复以下过程:
    - 从PC指示的存储器位置取出指令
    - 执行指令
    - 更新PC。

冯诺依曼结构

冯诺依曼体系结构是现代计算机的基本模型，由数学家约翰·冯·诺依曼在20世纪40年代提出。冯诺依曼计算机系统由5个部件组成：

- **运算器** 比如我们前文提到的ALU。
- **控制器** 比如我们前文提到的译码单元。
- **存储器** 比如内存、寄存器等等。
- **输入设备** 比如我们的键盘。
- **输出设备** 比如我们的显示器。

CPU的组成

寄存器（Register）

寄存器是CPU内部的存储器，用于CPU快速地存储和访问临时数据。CPU内部的寄存器数量很有限，能够记录的数据也很有限。通常，CPU的计算结果被存放在寄存器中，在通过其他指令将寄存器内的数据写入内存等空间更大的存储设备。在CPU中，往往有一个寄存器堆。

一般寄存器分为以下两种：

- **通用寄存器(General Purpose Register, GPR)** 用于一般的程序计算、存储。
- **控制状态寄存器(Control Status Register)** 用于控制CPU的状态。

算术逻辑单元（Arithmetic and Logic Unit, ALU）

ALU也是CPU中的重要部件，可以计算两个输入数字作加法、减法、与、或、异或、比较等等，并根据一个选择信号输出一其中的一个结果。

总线(Bus)和缓存(Cache)

CPU和外部的内存、设备等进行交互，需要使用总线。CPU向总线发送例如读取地址、写入地址、写入数据等信息，总线向CPU传输读取内容、写入是否成功等信息。总线与CPU之间有着复杂的传输协议，例如AXI4协议。CPU与外部在物理上是通过引脚相连的。

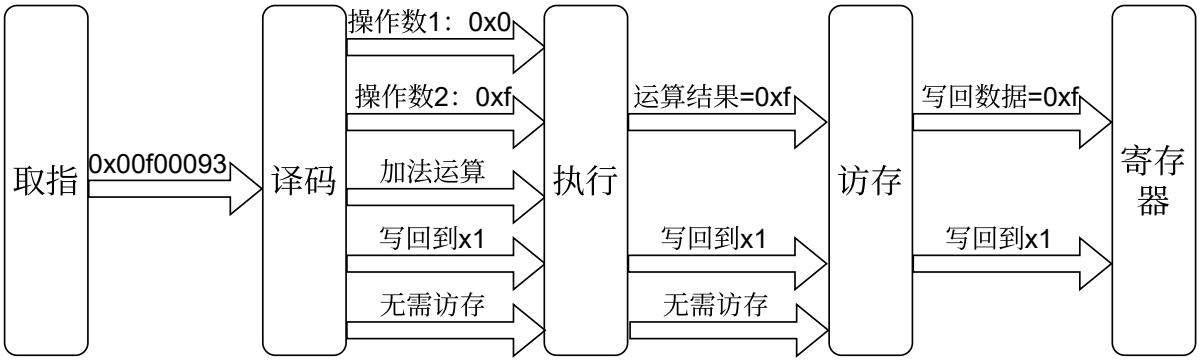
实际上，CPU的速度远快于外部的内存等，我们可以将常用的内存中的信息记载在CPU的内部来加快访问，这就是**缓存**。

CPU的运行过程

下面我们以RISC-V32中的`addi`指令为例，简单描述CPU的运行流程。

`addi x1, x0, 0xf`（`0x00f00093`）

在RISC-V中，`x0`寄存器的值始终为0，这条指令的作用是将寄存器`x1`赋值为`0xf`。RISC-V 中，通常指令的长度是4个字节。



- **取指** IFU (Instruction Fetch Unit) 将PC发送给指令缓存，指令缓存将检查是否命中，如果不命中，则向总线仲裁器发送需要读取内存PC位置数据的请求，并等待总线的回复；如果命中，指令缓存向IFU以更快的速度返回数据。经过数个或者数百个乃至数千个时钟周期，IFU完成了取指，取到了4个字节的数据`0x00f00093`，并将它发送给下游的IDU。
- **译码** IDU (Instruction Decode Unit) 从IFU接受到指令，进行译码工作，解码出如下信息，并将它们继续向下传递。
  - 这条指令需要ALU做求和运算
  - 取得两个操作数：`0`（读取`x0`寄存器得到）和`0xf`（从指令中解码得到）。
  - 这条指令需要写回寄存器，且需要写回到寄存器`x1`。
  - 这条指令不涉及到分支跳转，也不涉及到内存的读写。
- **执行** EXU (Excute Unit) 从IDU获得了操作数和要进行的操作，ALU计算得到了结果`0xf`，并将计算结果和一些有关的译码信息向下传递。
- **访存** LSU(Load Store Unit)发现这不是一条访存指令，将要写回的数据`0xf`和要写回的寄存器继续向下传递。
- **写回** 将数据`0xf`写到寄存器`x1`中，这条指令执行完毕。

> 本篇参考资料：
>
> NJU-ICS-PA[Introduction · GitBook (nju-projectn.github.io)]

计算机的硬件视角

逻辑电路

- 非门：一个输入，一个输出，输入输出相反。
- 与门：两个输入，一个输出。两边全是1，则输出为1，否则为0
- 或门：两个输入，一个输出。两边有一个是1，则输出为1，否则为0
- 异或门：两个输入，一个输出。两边不同，则输出为1，否则为0

加法逻辑

在进行加法`13+8`时

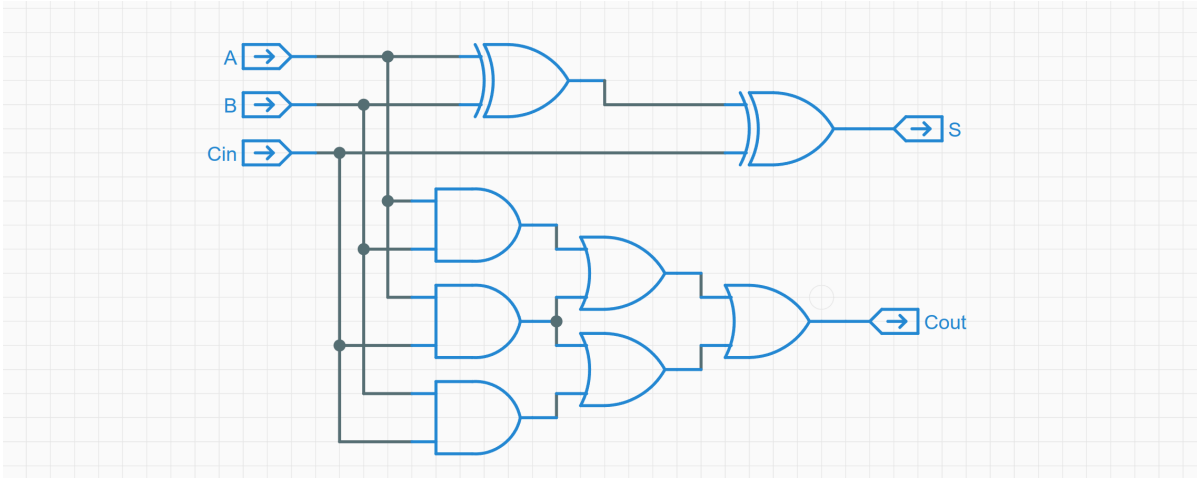
每一位加法需要考虑的有以下几个点:

- 上一级是否存在进位
- 正确完成计算
- 计算完成后需不需要进位

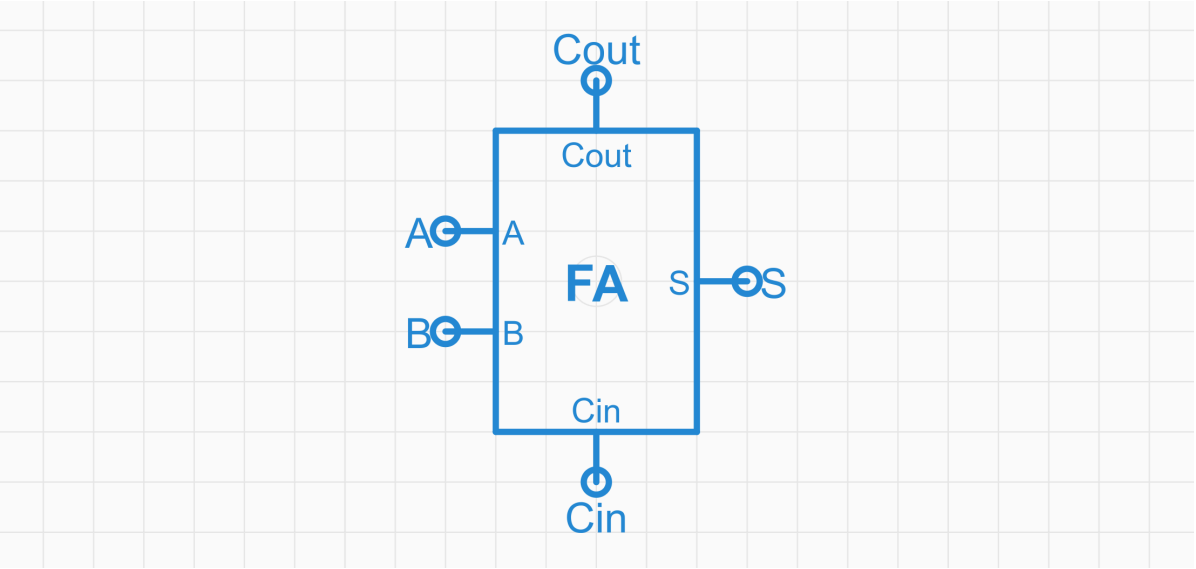
现在我们使用列表的方法,表示出所有的情况

Cin	B	A	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

那么,就可以得到如下规律总结
A,B,Cin有奇数个1则S输出1，否则为0
有两个及以上的1,则Cout输出为1,否则为0

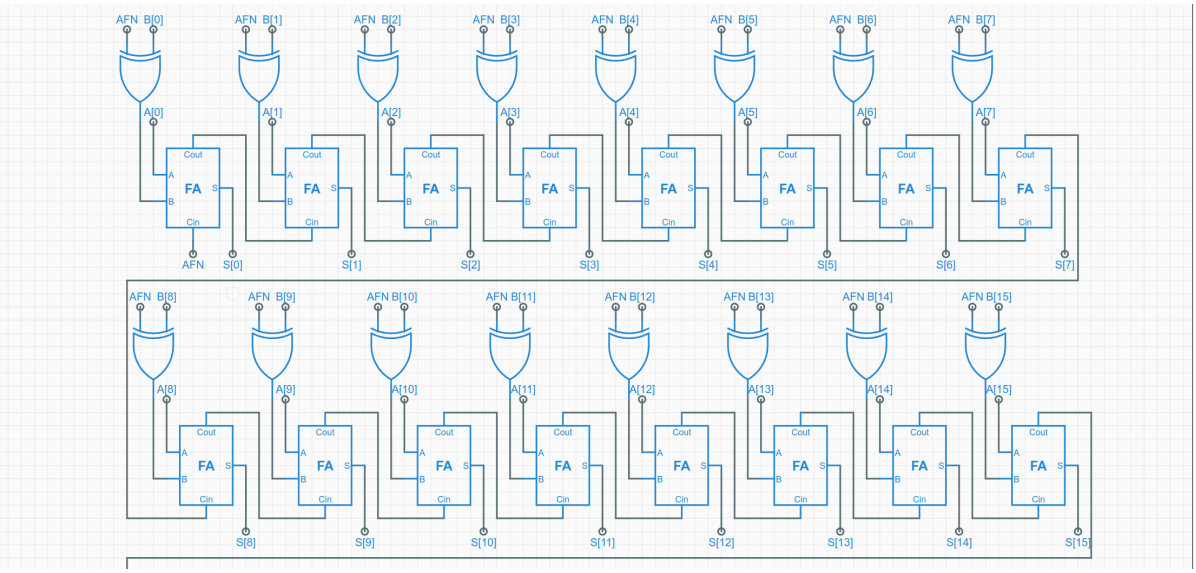


硬件抽象



将之前的电路转化为一个图标,我们不需要了解里面的具体逻辑

多位加法



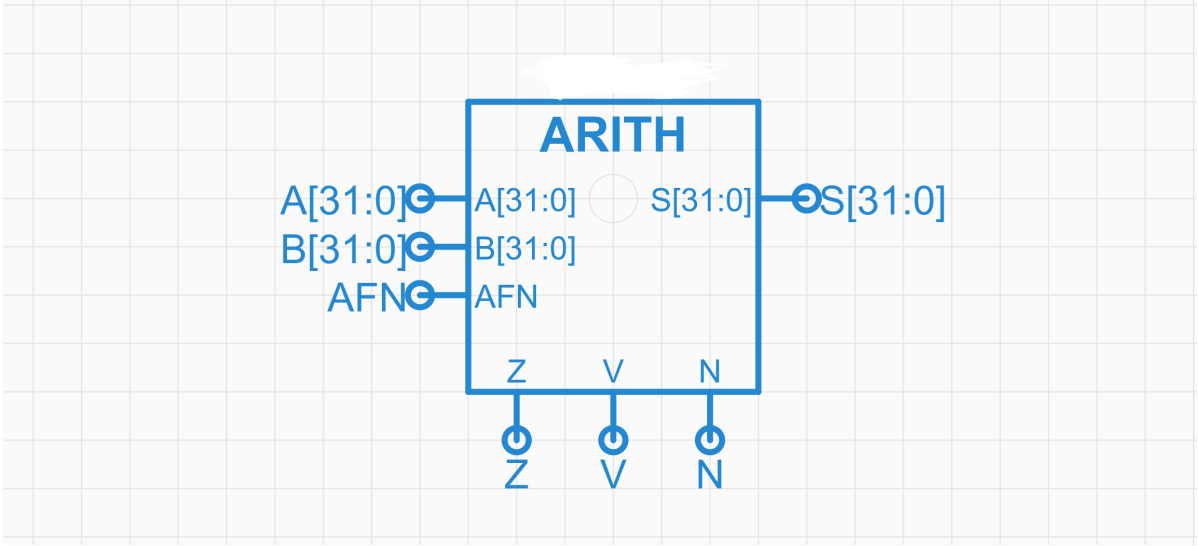
将加法器按照这样进行排列,前一位的进位输出,连接到后一位的进位输入,就能计算多位的结果

> AFN为1则说明是在做减法
> 根据公式`-x=~x-1`将第一位的进位设置为AFN,再添加一个异或门就能将B转化为-B参加运算

再次抽象

A,B各由32根数据线输入到器件中,输出32位结果

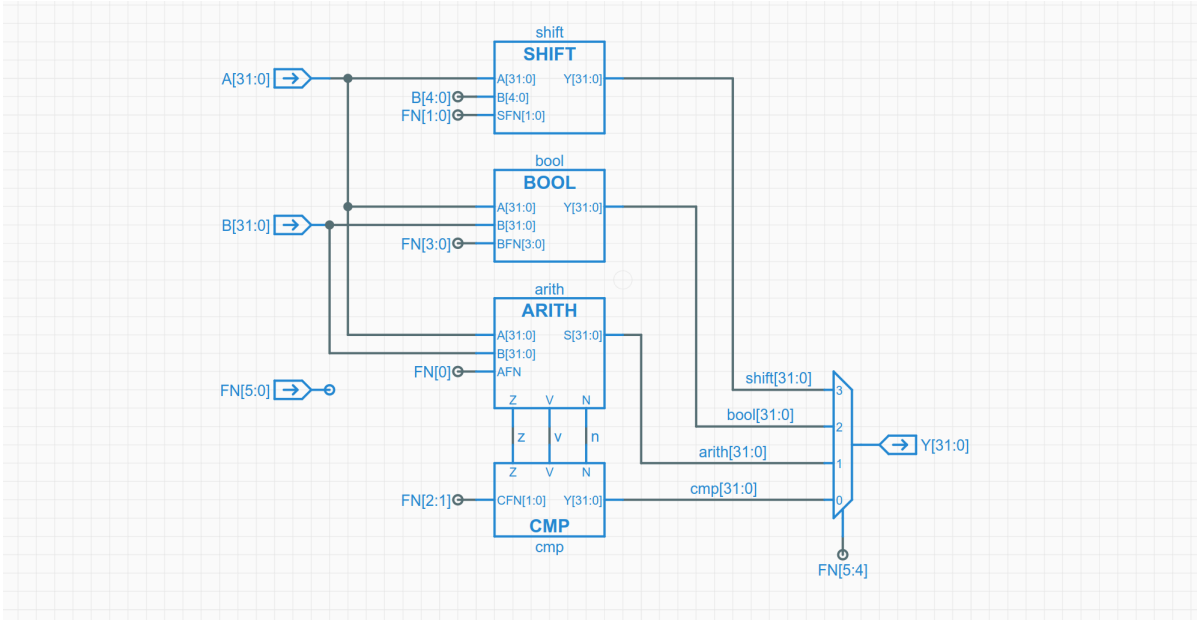
> Z V N是标识位,分别表示输出的值是否为0、是否溢出、是否为负



ALU

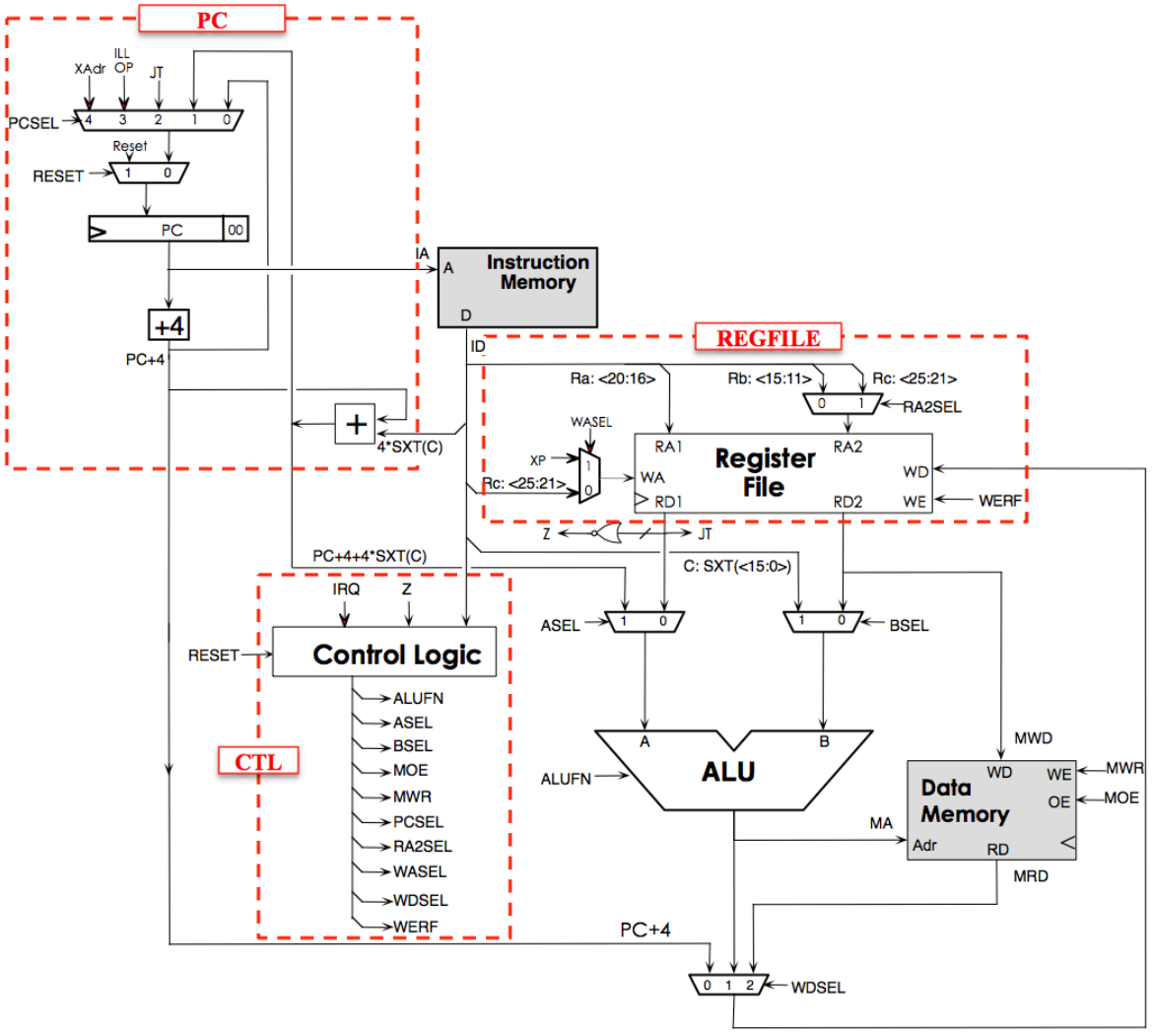
按照这个思路就能搭建出整个ALU(算术逻辑单元)完成基本的运算

- SHIFT：实现数据的移位
- BOOL：进行布尔运算
- AIRTH：进行加法减法运算
- CMP：与AIRTH紧密结合，根据其输出判断A,B大小关系



CPU

同样的,ALU也是一层抽象,它可以成为CPU的一个结构



操作系统

什么是操作系统

操作系统像是一个班主任，而运行在操作系统上的程序就像班里的孩子。操作系统既要保证这些孩子每个人都在合理的范围内做自己的事情，也要服务于这些孩子。

比如一个程序告诉操作系统，我要读一个文件了，那么操作系统就会将文件的内容送还给程序；另一个程序说，我要退出了，操作系统就会回收它的资源。

同时，操作系统负责与硬件交互，管理硬件。

内存模型和规范

程序在内存中是如何分布的？程序的代码存放在什么位置？局部变量又存放在什么位置？这也有规范标准，比如Linux遵循System V ABI标准。通常，运行在Linux的程序的内存分布：

而在Windows等其他操作系统上，内存分布则不同。

内存分布规范由编译器和链接器、操作系统共同遵守，这样，我们编译出来的文件才能被操作系统正确的加载和运行。指令集、编译器、操作系统三者遵循同一规范，互相配合，使得我们编写的代码可以正常运行。

> **实验：输出地址**
>
> 运行下面的程序，观察栈上变量的地址和堆上变量的地址。
>
> ```
> #include <stdio.h>
> #include <stdlib.h>
>
> int main(void) {
>  int x = 1;
>  void *y = malloc(sizeof(int));
>  printf("stack: %p\n", (void *)&x);
>  printf("heap : %p\n", y);
>  free(y);
>  return 0;
> }
> ```
>
> 可能的输出：
>
> ```
> stack: 0x7ffd27bb38dc
> heap : 0x59e598fe42a0
> ```

硬件视角的操作系统

操作系统在硬件视角下来看，更像是一个嵌入式的软件，一个十分普通的程序，只是它直接和硬件交互，并且会以特殊的方式启动其它的程序并且管理他们。

一个运行在操作系统上的程序想要输出`Hello World!`就只需要告诉操作系统，操作系统就会帮它输出，而操作系统就需要直接操作硬件，比如告诉显卡，在显示器上打印这几个字符。

操作系统做的三件事情

操作系统主要做三件简单的事情：

- **虚拟化** 让每个程序都以为自己独享整个内存空间和CPU。
- **并发** 让一个CPU可以运行多个程序，可以不断切换正在运行的程序。
- **持久化** 让信息可以持久存储，比如管理文件。

操作系统仿真（拓展）

> **Tip**
> 获取代码：`wget -r -nH --cut-dirs=5 https://e1psycongroo.github.io/NekoBytes-TheMissing/_site/resources/Lecture3/Codes/os-model`

下面我们来通过一个简单的模型来体验并发。

在这个模型中，我们模拟了一个简单的操作系统，可以运行多个进程。

首先，我们有两个系统调用：`EXIT`用于进程告诉操作系统自己需要退出，`WIRTE`函数用于告诉进程自己需要将一个参数`arg`打印到控制台中。

```
enum {
    SYS_EXIT,  // Process exits
    SYS_WRITE, // Write to console with a character
};

typedef struct {
    unsigned int syscall;
    char arg;
} Syscall;
```

为了简单起见，每一个进程的内部只有两个私有变量，`remainingStep`和`charToOutput`：

```
typedef struct {
    int  remainingStep; // A process will run remainingStep steps
    char charToOutput;  // A process will write this character to console by
doing a syscall
} Process; // Process's Context
```

一个进程每次运行会运行自己的代码，并在某一时刻进行系统调用，请求系统将`charToOutput`打印到控制台中，每个进程在请求了`remainingStep`次后，也就是打印了`remainingStep`个字符后，会通过系统调用`EXIT`请求退出，我们的模型在收到之后，会停止这个进程的运行。

我们定义`process_step`函数来抽象进程的运行，它会模拟一个进程运行，知道进程进行一个系统调用。它会返回这个系统调用。

```
Syscall process_step(Process *proc);
```

接下来，我们看看这个模型的运行：

```
void run() {
    Process *current;
    while (process_count()) {
```

```
        // The Operating System will randomly choose a process to run
        // 操作系统会随机选择一个进程来运行
        current = process_schedule();

        // Switch process context and run it until a syscall
        // 切换进程上下文并运行，直到发起系统调用
        Syscall call = process_step(current);

        if (call.syscall == SYS_EXIT) {
            // Process exits
            // 进程退出
            process_exit(current);
        } else if (call.syscall == SYS_WRITE) {
            // Write the character from syscall arg to the console
            // 将系统调用参数中的字符写入终端
            putchar(call.arg);
        }
    }
    putchar('\n');
}
```

首先，我们定义了一个`Process *current`变量，来表示现在正在运行的进程，接下来我们通过一个循环，只在没有进程的时候退出。在循环内部，我们每次都使用`process_schedule`函数来随机选择一个要运行的进程，并通过`process_step`来实际运行它，并且获取一个进程传出来的系统调用，然后我们在末尾执行这个系统调用。

我们通过不断切换正在运行的进程，实现了三个进程在一个CPU核心上的并发。

```
int main() {
    // Initialize the Operating System
    // 初始化操作系统
    init();

    // spawn processes
    // 创建进程
    spawn_process(5, 'A');
    spawn_process(5, 'B');
    spawn_process(5, 'C');

    // start running
    // 开始运行
    run();

    return 0;
}
```

在主函数中，我们生成了3个进程，它们分别会尝试输出5次`A`、5次`B`和5次`C`。多次运行程序，会发现程序的输出是由5个`A`、5个`B`和5个`C`构成的字符串，但是它们的顺序是随机的。

这只是一个简单的模型，实际上操作系统在进行并发进程的调度的时候，有自己的选择逻辑。

> **对计算机体系结构感兴趣?**
>
> 课程Nand2Tetris[Nand2Tetris]
>
> 书籍《深入入理解计算机系统（Computer System: A Programer's Perspective， CSAPP)》

> 课程CS61C [CS61C]
>
> 实践项目NJU-ICS-PA[Introduction · GitBook]
>
> 实践项目一生一芯[一生一芯 (oscc.cc)]

> **对操作系统感兴趣?**
>
> 课程南京大学《操作系统：设计与实现》操作系统：设计与实现 (2024 春季学期) 蒋炎岩老师
>
> 实践项目: MIT 6.1810: Operating System Enginerring
>
> 书籍《Operating System: Three Easy Pieces》