

数字系统课程设计

****单周期CPU设计****

院系：数据科学与计算机学院

专业：计算机类

班级：16级 计科7班

学生姓名：颜彬，王永锋

2017年6月18

目录

[1 简介 1](#_Toc485678934)

[2 指令集及其格式 2](#_Toc485678935)

[2.1 格式简介 2](#_Toc485678936)

[2.2 用途简介 5](#_Toc485678937)

[2.3 寄存器编号 6](#_Toc485678938)

[3 指令实现原理 7](#_Toc485678939)

[3.1 基本步骤 7](#_Toc485678940)

[3.2 OPq指令：addq，subq等 7](#_Toc485678941)

[3.3 movq操作寄存器的指令系列 8](#_Toc485678942)

[3.4 操作内存指令系列 8](#_Toc485678943)

[3.5 操作栈的指令 9](#_Toc485678944)

[3.6 跳转指令 9](#_Toc485678945)

[4 CPU模块构成 10](#_Toc485678946)

[4.1 CPU总模块组成 10](#_Toc485678947)

[4.2 指令内容：取指模块 11](#_Toc485678948)

[4.1 程序计数器 12](#_Toc485678949)

[4.2 寄存器文件 14](#_Toc485678950)

[4.3 算术逻辑单元 15](#_Toc485678951)

[4.4 内存 17](#_Toc485678952)

[5 测试样例 18](#_Toc485678953)

[6 心得体会 18](#_Toc485678954)

[6.1 王永锋的心得体会 18](#_Toc485678955)

# 简介

CPU（即中央处理器）在计算机中担当计算和处理的重任。就CPU本身的功能而言，CPU本身并不关心数据的存储以及结果的显示，而是需要根据指令对数据集进行运算或指令控制，更新内部的核心寄存器，并提供输出。从功能实现角度来讲，CPU只是一个用于将用01串描述的机器指令转变为实现一定功能的数字逻辑。这里的机器指令，就是CPU对应的指令集。

每一个CPU都有一个对应的指令集。目前因特尔处理器中，64位的处理器大多数采用的是x64-64指令集，受这个指令集的启发，我们定义一个更为简单的指令集，称之为“Y86-64”指令集。与x86-64指令集相比，Y86-64指令集的数据类型，指令和寻址方式都要少一些，字节级编码也比较简单。即使指令集简单，它仍足够完整，能够让我们写一些简单的程序，包括从1-100整数的求和，或者求解斐波那契数列的第n个数。

确定好了指令集，就开始确定CPU的内部实现。CPU的实现有多种，其中包括单周期CPU，多周期CPU，还有流水线化的CPU，其中流水线化的CPU能够充分提高CPU的计算效率，增加了系统的吞吐量，但实现的难度也更高。在本次试验中，为了简单起见，我们选择实现单周期CPU。

参考《深入理解计算机系统（第三版）》，我们把CPU的顶层结构实现分为五个模块：取指模块，寄存器文件，算术/逻辑单元（ALU），内存文件，程序计数器。我们使用了verilog语言用于实现CPU的内部硬件结构，并且使用vivado仿真测试CPU的功能。其中取指模块还集成了读取机器码的功能，用于机器码程序的输入。为了方便将汇编代码翻译成机器码，我们还用python实现了一个汇编器，该汇编器支持将满足一定语法的汇编程序转成适用于使用“Y86-64”指令集的CPU的机器码程序。

总之，在本次实验中，我们使用verilog语言，结合vivado实现了一个能够运行我们的汇编程序的CPU。

# 指令集及其格式

## 简介

1. type：每条指令的第一个字节（8bits）唯一地编码了指令，称为type
   1. icode：type的前四位（4bits）为代码部分，编码了指令的种类，称为icode
   2. ifun：type的后四位（4bits）为功能部分，编码了一类指令的具体功能，称为ifun
2. register：指令可能包含一个字节(8bits)编码其所使用的寄存器
   1. rA：register的前四位编码了第一个寄存器的ID，称为rA
   2. rB：register的后四位编码了第二个寄存器的ID，称为rB
3. valC：指令可能包含8个字节（64位）用于编码一个数字
   1. Val：当valC看做一个立即数时，也记为Val
   2. Dest：当valC看做一个地址时，也记为Dest

## 指令格式

以下使用十六进制数代表4bit

### 类型A

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 格式 | icode | ifun |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令 | 4 | 4 |  |  |  |  |  |  |  |
| halt | 0 | 0 |  |  |  |  |  |  |  |
| nop | 1 | 0 |  |  |  |  |  |  |  |
| ret | 9 | 0 |  |  |  |  |  |  |  |

### 类型B

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 格式 | icode | ifun | registerA | registerB |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令 | 4 | 4 | 4 | 4 |  | |  | |  | |  | |  | |
| rrmovq | 2 | 0 | rA | rB |  | |  | |  | |  | |  | | |
| opq | 6 | x | rA | rB | |  | |  | |  | |  | |  | | |
| cmovXX | 2 | x | rA | rB |  | |  | |  | |  | |  | |
| pushq | A | 0 | rA | F |  | |  | |  | |  | |  | |
| popq | B | 0 | rA | F |  | |  | |  | |  | |  | |

### 类型C

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 格式 | icode | ifun | valC |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 指令 | 4 | 4 | 8x8 |  |  |  |  |
| jXX | 7 | x | Dest |  |  |  |  | |
| call | 8 | 0 | Dest |  |  |  |  |

### 类型D

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 格式 | icode | ifun | registerA | registerB | valC |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令 | 4 | 4 | 4 | 4 | 8x8 |  |  |  |  |
| irmovq | 3 | 0 | F | rB | Val |  |  |  |  | |
| rmmovq | 4 | 0 | rA | rB | Dest |  |  |  |  |
| mrmovq | 5 | 0 | rA | rB | Dest |  |  |  |  |

opq、jXX，cmoXX指令，具体ifun对应的功能如下：

其中e表示equal，g表示greater，l表示less，n表示not

比如jle表示jump if less or equal to

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| opq | icode | ifun |  | jXX | icode | ifun |  | cmovXX | icode | ifun |
| addq | 6 | 0 | jmp | 7 | 0 | rrmovq | 2 | 0 |
| subq | 6 | 1 | jle | 7 | 1 | cmovle | 2 | 1 |
| andq | 6 | 2 | jl | 7 | 2 | cmovl | 2 | 2 |
| xorq | 6 | 3 | je | 7 | 3 | cmove | 2 | 3 |
|  | | | jne | 7 | 4 | cmovne | 2 | 4 |
| jge | 7 | 5 | cmovge | 2 | 5 |
| jg | 7 | 6 | cmovg | 2 | 6 |

## 用途简介

### 类型A

1. halt：停机。暂停执行一切指令
2. nop：空指令。不执行操作直到下一个指令到达。
3. ret：函数返回。取出栈指针指向的数据，执行指令内存中该数据位置的指令。栈pop8个字节。

### 类型B

1. rrmovq：寄存器-寄存器移动。将寄存器rA中的值复制到到rB中。
2. opq：整数操作。根据ifun的不同分别对rA与rB执行rB+rA，rB-rA，rB&rA，rB|rA的操作
3. cmovXX：条件传送。根据ifun编码的条件决定是否执行寄存器-寄存器移动。
4. pushq：入栈。将rA的值写入栈中
5. popq：出栈。将栈中读出的值赋给rA

### 类型C

1. jXX：条件跳转。根据ifun的条件决定是否跳转到Dest指明的位置
2. call：函数调用。直接跳转到Dest指明的位置。将当前执行的指令位置存入栈中。

### 类型D

1. irmovq：立即数-寄存器移动。将Val的值赋给rB
2. rmmovq：寄存器-内存移动。将rA的值赋值到内存中Dest+rB的位置
3. mrmovq：内存-寄存器移动。将内存中rB+Dest的值赋给rA

## 寄存器编号

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 数字 | 寄存器 |  | 数字 | 寄存器 |  | 数字 | 寄存器 |  | 数字 | 寄存器 |
| 0 | %rax | 4 | %rsp | 8 | %r8 | C | %r12 |
| 1 | %rcx | 5 | %rbp | 9 | %r9 | D | %r13 |
| 2 | %rdx | 6 | %rsi | A | %r10 | E | %r14 |
| 3 | %rbx | 7 | %rdi | B | %r11 | F | none |

# 指令实现原理

一般而言，每一条指令的实现，都需要经过六个步骤的处理:取指，译码，执行，访存，写回，更新PC。

## 基本步骤

1. 取指：读取程序计数器（PC）的值，并且以此值为地址读取当前字节及之后的字节作为当前处理的指令。读取后抽取指令的内容并传递到其他模块
2. 译码：从寄存器文件中读取最多两个操作数，得到值valA，valB
3. 执行：根据ifun，进行计算，对某一些命令需要更新条件码。
4. 访存：从内存中读出数据，或者将数据写回内存
5. 写回：最多将两个结果写回到寄存器文件
6. 更新PC：讲PC设置成下一条指令的地址

## OPq指令：addq，subq等

|  |  |
| --- | --- |
| 阶段 | OPq rA， rB |
| 取指 | icode：fun ← M1[PC]  rA:rB ← M1[PC+1]  valP ← PC+2 |
| 译码 | valA ← R[rA]  valB ← R[rB] |
| 执行 | valE ← valB OP valA  Set CC |
| 访存 |  |
| 写回 | R[rB] ← valE |
| 更新PC | PC ← valP |

## movq操作寄存器的指令系列

|  |  |  |  |
| --- | --- | --- | --- |
| 阶段 | rrmovq rA， rB | cmovq rA， rB | Irmovq V, rB |
| 取指 | icode：fun←M1[PC]  rA:rB ← M1[PC+1]  valP ← PC+2 | icode：fun←M1[PC]  rA:rB ← M1[PC+1]  valP ← PC+2 | icode：fun ← M1[PC]  rA:rB ← M1[PC+1]  valC ← M8[PC+2]  valP ← PC+10 |
| 译码 | valA ← R[rA] | valA ← R[rA] |  |
| 执行 | valE ← 0 + valA | valE ← 0 + valA  Cnd ← Cond(CC, ifun) | valE ← 0 + valC |
| 访存 |  |  |  |
| 写回 | R[rB] ← valE | if(Cnd)  R[rB] ← valE | R[rB] ← valE |
| PC | PC ← valP | PC ← valP | PC ← valP |

## 操作内存指令系列

|  |  |  |
| --- | --- | --- |
| 阶段 | rmmovq rA，D(rB) | mrmovq D(rB), rA |
| 取指 | icode：fun ← M1[PC]  rA:rB ← M1[PC+1]  valC ← M8[PC+2]  valP ← PC+10 | icode：fun ← M1[PC]  rA:rB ← M1[PC+1]  valC ← M8[PC+2]  valP ← PC+10 |
| 译码 | valA ← R[rA]  valB ← R[rB] | valB ← R[rB] |
| 执行 | valE ← valB + valC | valE ← valB + valC |
| 访存 | M8[valE] ← valA | valM ← M8[valE] |
| 写回 |  | R[rA] ← valM |
| PC | PC ← valP | PC ← valP |

## 操作栈的指令

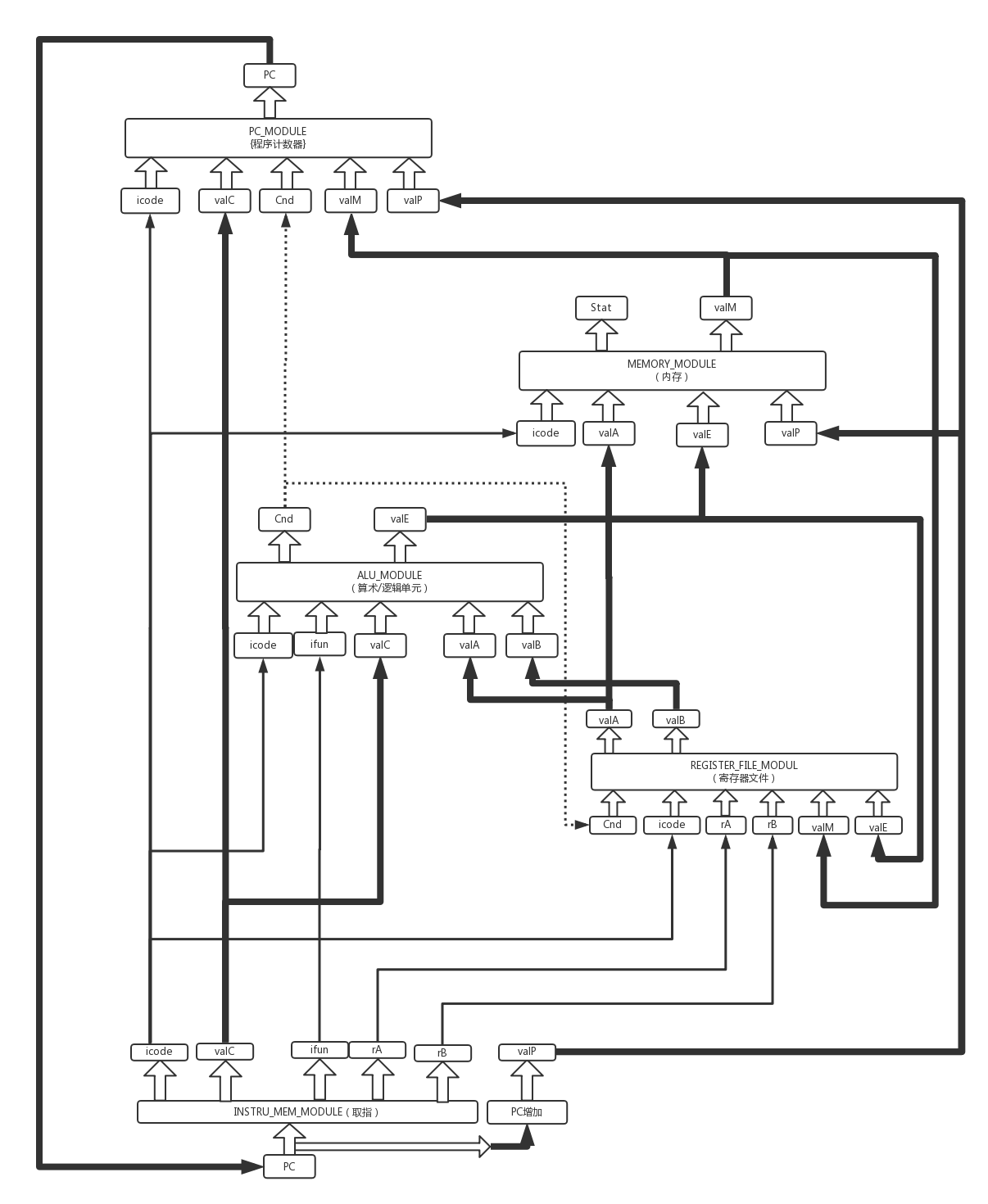
|  |  |  |
| --- | --- | --- |
| 阶段 | pushq rA | popq rA |
| 取指 | icode：fun ← M1[PC]  rA:rB ← M1[PC+1]  valP ← PC+2 | icode：fun ← M1[PC]  rA:rB ← M1[PC+1]  valP ← PC+2 |
| 译码 | valA ← R[rA]  valB ← R[%rsp] | valA ← R[%rsp]  valB ← R[%rsp] |
| 执行 | valE ← valB + (-8) | valE ← valB + 8 |
| 访存 | M8[valE] ← valA | valM ← M8[valA] |
| 写回 | R[%rsp] ← valE | R[%rsp] ← valE  R[rA] ← valM |
| PC | PC ← valP | PC ← valP |

## 跳转指令

|  |  |  |  |
| --- | --- | --- | --- |
| 阶段 | jXX Dest | call Dest | ret |
| 取指 | icode：fun←M1[PC]  valC ← M8[PC+1]  valP ← PC+9 | icode：fun ← M1[PC]  valC ← M8[PC+1]  valP ← PC+9 | icode：fun ← M1[PC]  valP ← PC+1 |
| 译码 |  | valB ← R[%rsp] | valA ← R[%rsp]  valB ← R[%rsp] |
| 执行 | Cnd ← Cond(CC, ifun) | valE ← valB + (-8) | valE ← valB + 8 |
| 访存 |  | M8[valE] ← valP | valM ← M8[valA] |
| 写回 |  | R[%rsp] ← valE | R[%rsp] ← valE |
| PC | PC← Cnd?valC:valP | PC ← valC | PC ← valM |

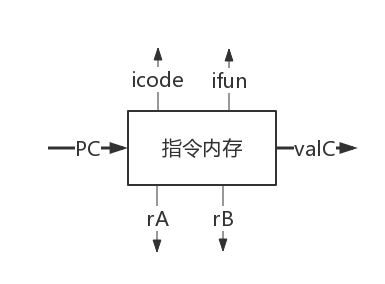
# CPU模块构成

## CPU总模块组成



## 指令内容：取指模块

### 使用方式



1. PC：程序计数器。当前指令的地址
2. icode，ifun：指令的代码和功能
3. rA，rB：指令使用的寄存器（不使用则输出F）
4. valC：指令的立即数（没有则输出X）

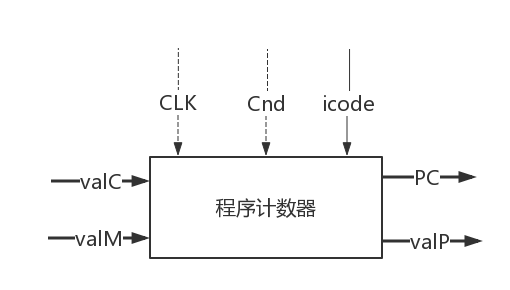
### 模块实现

指令内存中有一组以8位为单位的长度足够的数组，用于存储指令。初始化时，代码采用$memread指令将写好的字节码读入数组中。

随着PC的改变，指令内存会异步解析并输出以PC为首地址的指令。对上述四种类型的指令，指令内存会输出icode，ifun，rA，rB，valC。若当前指令没有出现前者的一个或几个（例如没有出现立即数valC），该模块依旧会输出某些数值（或不定态X）。其他器件保证不会使用这些数值。

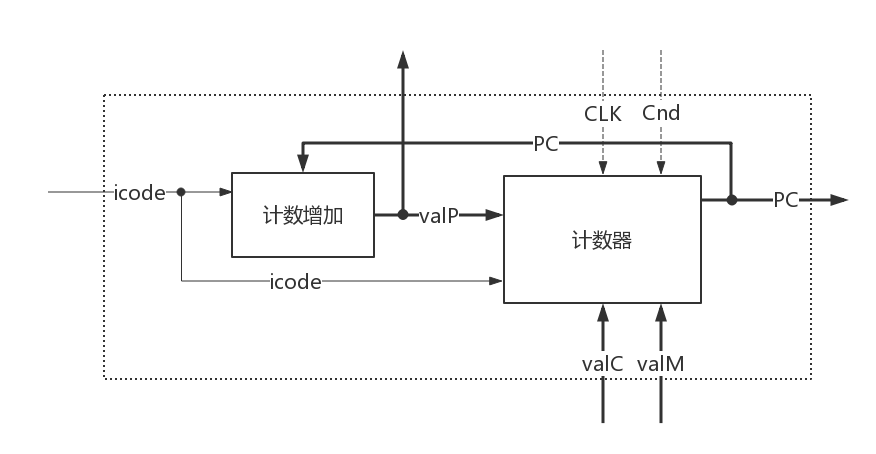
## 程序计数器

### 模块使用方式



1. CLK：时钟信号。每个上升沿到达后，PC会改变为新指令的首地址
2. Cnd：条件信号。决定是否发生跳转。当icode为jXX、call时，决定是否跳转到valC对应的地址
3. icode：旧指令的指令代码。旧指令是否为跳转指令将决定新指令的地址
4. PC：当前指令的首地址
5. valP：无跳转时的下一条指令的地址。即当前指令的首地址加上当前指令的长度
6. valC：如果存在，则为旧指令的valC
7. valM：如果存在，则为旧指令访存读取的数值

### 模块实现



I指令的相邻指令指在指令内存中与I紧挨着的下一条指令的地址。I指令的下一条地址指CPU实际工作时，执行完I指令后执行的地址。

计数增加模块根据当前指令的地址（PC）和指令类型（icode），计算相邻指令的地址（valP）。

每个时钟上升沿到达后，计数器输出下一条指令的地址。若当前指令为跳转指令（jXX），当Cond为1时，输出的地址为跳转目的地（valC），否则则为相邻指令的首地址（valP）。若当前指令为调用（call）指令，则输出的地址为调用目的地（valC）。若当前指令为返回（ret）指令，则输出地址为栈指针指向的地址（valM）。否则，输出的地址为相邻指令的首地址（valP）。

## 寄存器文件

### 使用方式

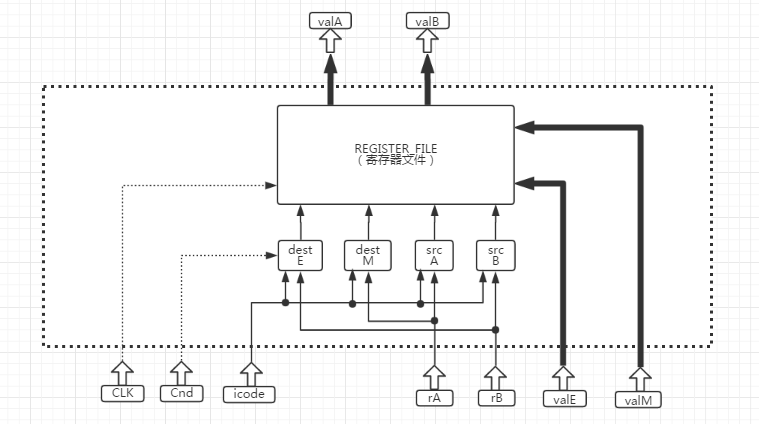
寄存器文件，内部有16个寄存器。根据时钟上升沿同步写入寄存器 。异步读取寄存器内容。

### 模块实现

在取指阶段过后，得到的两个rA，rB作为寄存器文件地址的输入端，

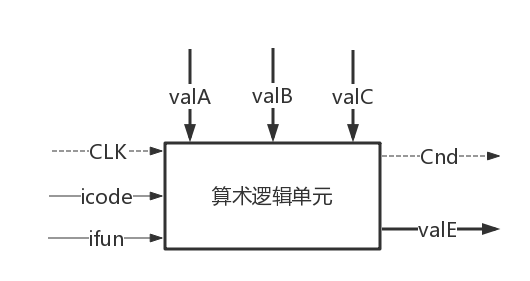
在某些特定的icode下，rA，rB作为写寄存器的地址，往rA，rB对应的寄存器写入valM，valE，此时的输出无效。

在另一些特定的icode下，rA，rB作为读寄存器的地址，rA作为地址读出对应寄存器的值为valA。rB作为另一个地址读出对应寄存器的值为valB，并作为输出。



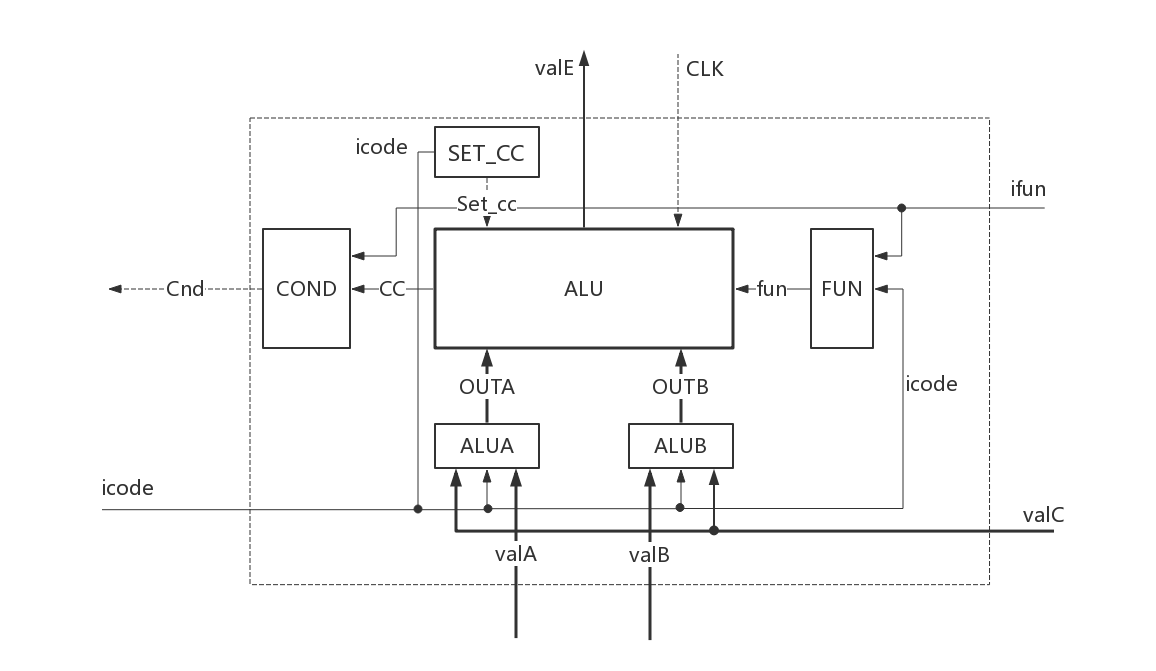
## 算术逻辑单元

### 模块使用方式



1. CLK：时钟上升沿到达后更新Cnd。Cnd输出ifun对应的条件是否为真。比如上一次运算的结果是否为0等
2. icode，ifun：共同确定ALU执行的操作（加、减、与、异或）
3. valA，valB，valC：valA和valB为寄存器rA与rB中的值（如果该条指令有出现寄存器）。valC为指令中的立即数（如果该条指令由出现立即数）。它们参与决定ALU的输出（valE）
4. Cnd：输出指令需要的条件是否成立。比如对je（jump if equal），Cnd输出“相等”条件是否成立。Cnd由valE决定。
5. valE：输出运算后的结果

### 模块实现



ALUA和ALUB根据指令的类型（icode），从valA，valB，valC中选择合适的值传送到ALU。

Fun模块根据指令类型（icode）和指令功能（ifun），确定ALU应该使用的功能（function）。比如指令addq应该实现加法（00），指令call应该实现减法（01），将栈指针的值减去8字节。

ALU模块在计算valE的同时会设置条件码（condition code，CC），记录valE的值是否为0，是否为负，计算是否发生溢出等。ALU在输出valE的同时还会输出CC，用于计算Cond

COND模块将利用来自ALU的CC，计算Cond。比如对于jg指令（jump if greater），Cond为1等价于valE不是负数且不为零。利用CC可以同理计算出其他情况Cond的值。

SET\_CC模块将利用指令类型计算出ALU是否需要修改或设置条件码。只有当类型为运算（operation，opq）时，才会设置条件码。

## 内存

### 使用方式

内存数据，实现的功能是

内存数据内有以字节为单位的大容量存储器

根据时钟同步的对对应地址的内存写入数据

异步读取对应地址的内存的值

### 模块实现

最核心的模块：MEMORY

当read\_flag为1，读取addr对应的内存的值，并且作为valM的值输出

当write\_flag为1，将write\_data写进addr对应的内存，此时valM输出无效。

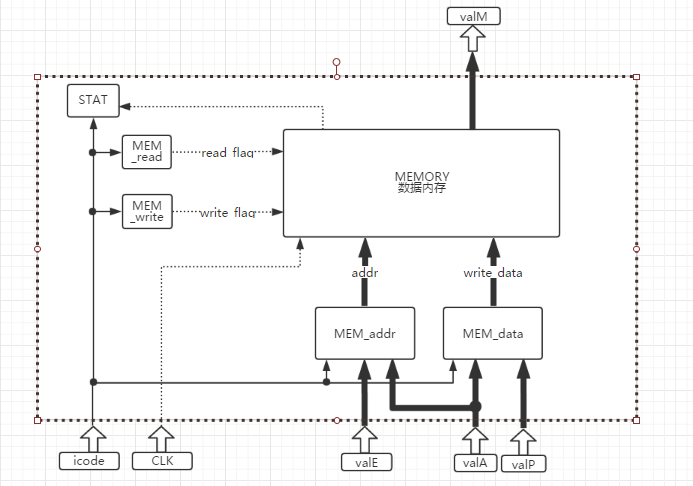
flag的确定以及addr，write\_data的确定

由icode确定

在某些icode下，write\_data为valP，在另一些icode下，write\_data为valA

这个选择的功能有MEM\_data子模块实现

其他模块同理



# 测试样例

## 自然数求和（循环）

### 汇编及机器码

### 波形图

### 相关分析

## 求斐波那契项（递归）

### 汇编及机器码

### 波形图

### 相关分析

# 心得体会

## 王永锋的心得体会

从一开始对CPU一无所知，到现在完成这样一个CPU的设计，中间也经历了不少吧。

一开始，“CPU这么厉害，应该好难吧”，于是先拿了一本《数字设计与计算机结构体系》来看，看着看着，好像懂了很多，CPU应该也就这样了吧，不就知道一个很大很大很巨大的有线状态机吗？这个时候颜彬也对这个颇有兴趣，正巧看到了老师上课放的数电项目论文，大受鼓舞，于是马上决定要做CPU。

想来简单，做起来难。为了完成CPU，还需要更多的知识储备。原来那本书介绍的太过简略，介绍的指令集也和颜彬所看的书不一样。为了统一，先是买了《深入理解计算机系统》，梦想着“CPU一夜通”，然而事情并没有这么简单。打开书，看了一段时间，才发现之前那本参考书实在介绍的过于简略，好吧，那就继续看书，看了一堆汇编语言相关的材料，终于迫不及待翻开有CPU结构体系的一章。此后的事情，便是不断的翻书，翻前面查阅之前忘记的内容，然后又翻回来看。

没有一夜通这样的事情，追求速度最终导致的就是编写代码的缓慢还有无限的debug。在这里要大大感谢颜彬大神呀，他对CPU了如指掌，大大提高了CPU的设计进程，我在旁边除了膜，还有膜，他还用Python写出了汇编器，加速了之后测试样例的速度。在整个设计过程中，他给我的帮助真是数也数不清，清晰的思路，睿智的分析，也让我开始反思自己学习习惯的缺点。

这个项目还没有完成，我希望，以后还能继续完成多周期CPU，流水线化的CPU，更希望能够完成一个基于自己CPU的操作系统，这里就当做是给自己立了一个flag吧。当然希望自己能够克服自己在这个项目的完成过程中暴露的问题，期待以后的进步！

## 颜彬的心得体会

算起来这个学期写过不少项目了，但让我收获最大的还是这个项目，单周期CPU。

这个学期初我就开始看了《深入理解计算机系统》（Computer System A Programmer’s Perspective）。前一阵子一直被人推荐这本书。这本书从C语言讲起，讲到汇编，再到处理器，还设计许多底层的知识点。当我在数电课上听说要写项目时，几乎没有思索就打算实现一个简易的处理器，和王永峰一拍即合。

这次项目让我的代码能力进步很大。对我而言Verilog是一门很新的语言。学期初刚接触它的时候，我对着编辑器一个单词都写不下来。短短的几行程序都一直在报错。就连很简单的ALU和学号循环显示等作业，都花了我十几个小时。但这些不断的debug-coding循环，不仅让我对该语言的语法有了深入了解，还让我明白了这个语言的代码思维：“把Verilog语言看成硬件描述语言“。这短短的一句话真的需要一定的代码量才能体会得到。坑爬得多了，写起代码才会得心应手。很感谢这个项目和这个学期的作业，给了我这种“历练”。

这次项目，我和王永峰花了很多时间在写test bench文件上。我和他都觉得，处理器的结构如此复杂，如果不花大量精力验证底层器件的正确性，将来debug时会极其痛苦。很感谢王永峰，他思维发散，心思缜密，能定位到许多隐秘的bug。和他一起打代码实现处理器会特别心安。

只有自己实现一遍处理器，才能真正理解参考书中的一些精巧的思想。比如书中给出的参考汇编里，寄存器-内存移动的格式为rmmovq rA, D(rB), 内存-寄存器移动的格式为mrmovq D(rB), rA.为什么后者rB在前rA在后，与前者排布相反呢。因为这样能让指令的格式统一，与立即数相加的永远是rB。实现ALU时可以少连一根线。参考书提供的实现含有许多类似的小细节，只有在实现时才能理解其精妙。同时参考书还为我们留下了许多空白，许多器件的具体实现都没有给出，需要我们自行设计。整个设计有许多细节都值得思考。

设计时遇到了许多意外的困难。对Verilog语法细节不熟悉，google相关的资料找了很久。更新代码时有些代码漏了修改，旧代码与新代码冲突，导致意外的bug。触碰到了Verilog的语法坑点，踩到了坑。甚至还因为递归的汇编比较难写，汇编连续写错两次，怀疑是器件有bug找了很久，最终发现是汇编写错。还有些很简单的bug，比如单纯的单词拼错，给wire线赋初值等等。这些bug都让我们对verilog以及相关设计更加熟练。

总之，这次项目是一个很难得的机会，让我体会到了团队配合的重要性，了解了计算机较为底层的实现，熟悉并使用了一个新语言verilog，还有把课堂上抽象的知识运用到实践上。