

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日

目录

[1. 简介 1](#_Toc485801549)

[2. 指令集及其格式 3](#_Toc485801550)

[2.1 简介 3](#_Toc485801551)

[2.2 指令格式 3](#_Toc485801552)

[2.3 用途简介 6](#_Toc485801553)

[2.4 寄存器编号 7](#_Toc485801554)

[3. 指令实现原理 8](#_Toc485801555)

[3.1 基本步骤 8](#_Toc485801556)

[3.2 整数操作指令系列 9](#_Toc485801557)

[3.3 移动指令系列 9](#_Toc485801558)

[3.4 内存操作指令系列 10](#_Toc485801559)

[3.5 栈指令系列 10](#_Toc485801560)

[3.6 跳转指令系列 11](#_Toc485801561)

[4. CPU模块构成 12](#_Toc485801562)

[4.1 CPU总模块组成 12](#_Toc485801563)

[4.2 指令内容：取指模块 13](#_Toc485801564)

[4.3 程序计数器 14](#_Toc485801565)

[4.4 寄存器文件 16](#_Toc485801566)

[4.5 算术逻辑单元 18](#_Toc485801567)

[4.6 内存 20](#_Toc485801568)

[5. 测试样例 22](#_Toc485801569)

[5.1 自然数求和（循环） 22](#_Toc485801570)

[5.2 求斐波那契项（递归） 22](#_Toc485801571)

[6. 心得体会 24](#_Toc485801572)

[6.1 王永锋的心得体会 24](#_Toc485801573)

[6.2 颜彬的心得体会 25](#_Toc485801574)

[7. 参考文献 27](#_Toc485801575)

[8. 附录A（代码） 28](#_Toc485801576)

[9. 附录B（波形图） 48](#_Toc485801583)

# 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。

代码已上传github

<https://github.com/YanB25/MyCPU.git>

# 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

## 整数操作指令系列

|  |  |
| --- | --- |
| 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 |

## 移动指令系列

|  |  |  |  |
| --- | --- | --- | --- |
| 阶段 | rrmovq rA， rB | cmovq rA， rB | Irmovq V, rB |
| 取指 | icod：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 |

## 内存操作指令系列

|  |  |  |
| --- | --- | --- |
| 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 |

## 栈指令系列

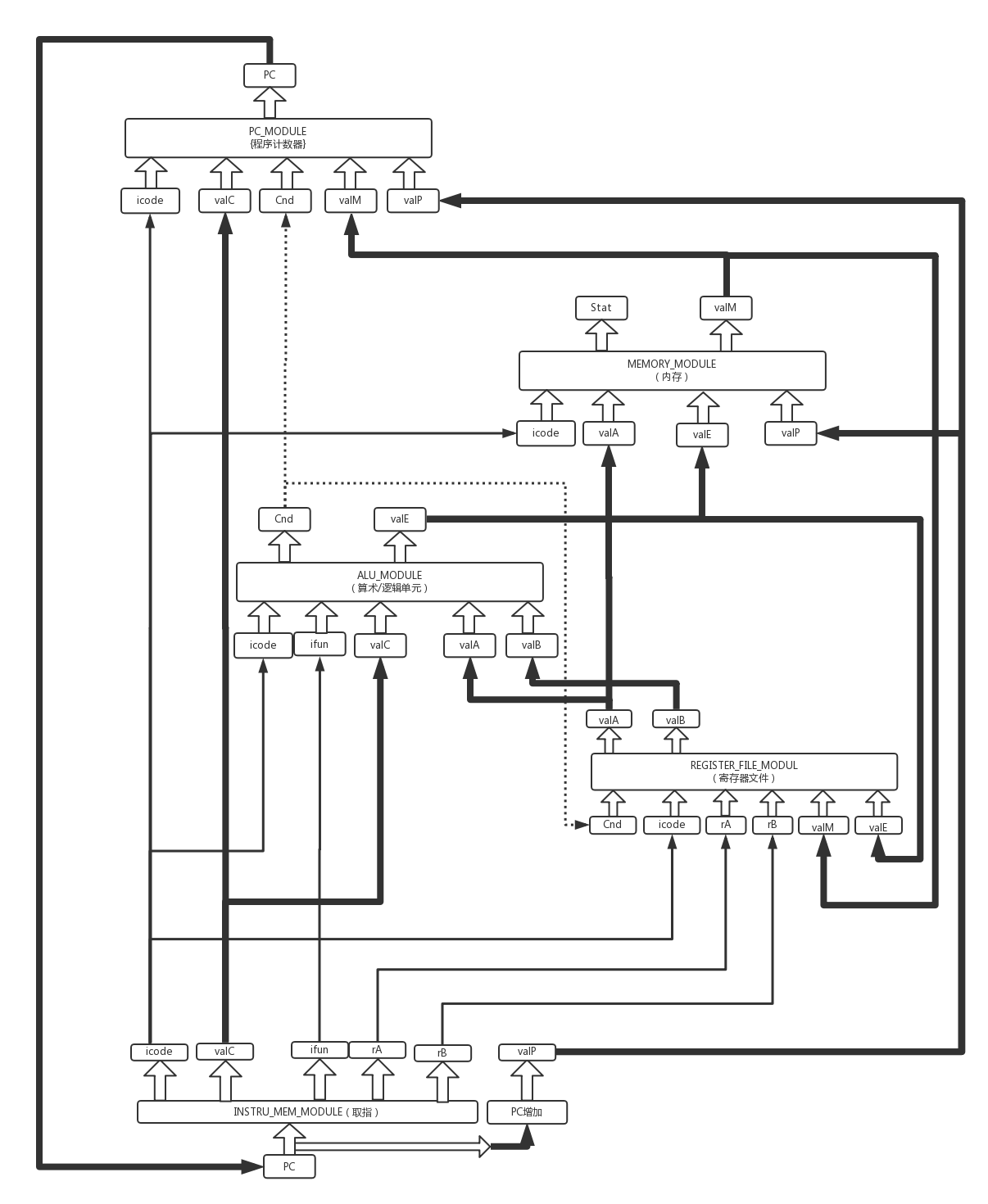
|  |  |  |
| --- | --- | --- |
| 阶段 | pushq rA | popq rA |
| 取指 | icode：fun ← M1[PC]  rA:rB ← M1[PC+1]  valP ← PC+2 | icode：fun ← M1[PC]  rA:rB ← M1[PC+1]  valP ← PC+2 |
| 译码 | valA ← R[rA]  valB ← R[%rsp] | valA ← R[%rsp]  valB ← R[%rsp] |
| 执行 | valE ← valB + (-8) | valE ← valB + 8 |
| 访存 | M8[valE]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

## 跳转指令系列

|  |  |  |  |
| --- | --- | --- | --- |
| 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 |

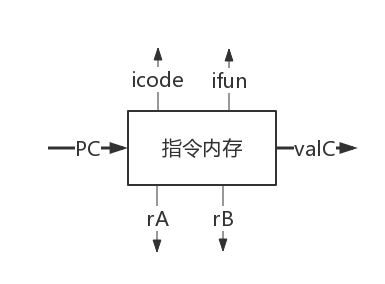
# CPU模块构成

## 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）

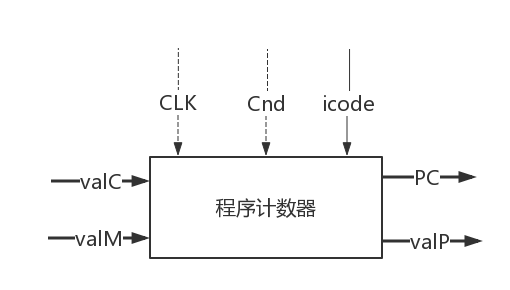
### 模块实现

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。

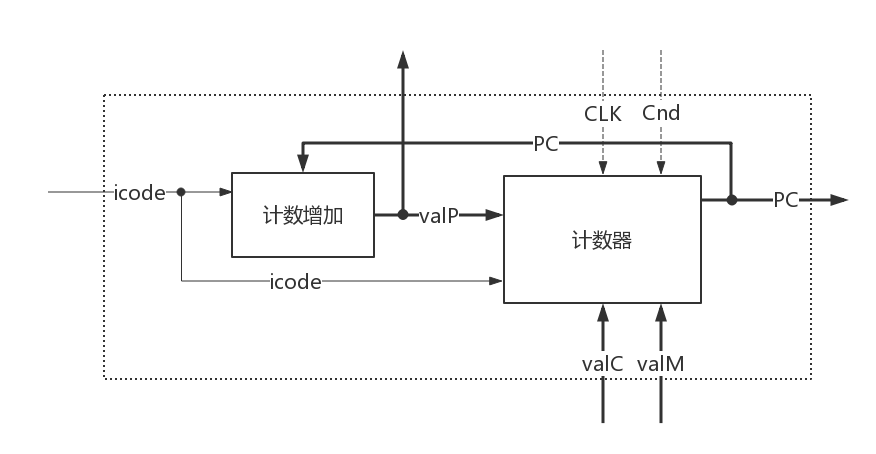
## 程序计数器

### 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

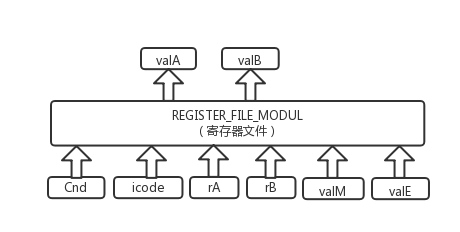


I指令的相邻指令指在指令内存中与I紧挨着的下一条指令的地址。I指令的下一条地址指CPU实际工作时，执行完I指令后执行的地址。

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

1. CLK:当时钟上升沿到达的时候写数据进对应的寄存器，实现同步写入
2. Cnd：如果Cnd为1，则采用
3. icode:从rA，rB确定srcA，srcB，desE，desM的值
4. srcA，valA:将srcA对应的数据读取输出为valA的值
5. srcB，valB:将srcB对应的数据读取输出为valB的值
6. valE，destE:将valE的值写进destE对应的寄存器
7. valM，destM:将valM的值写进destM对应的寄存器

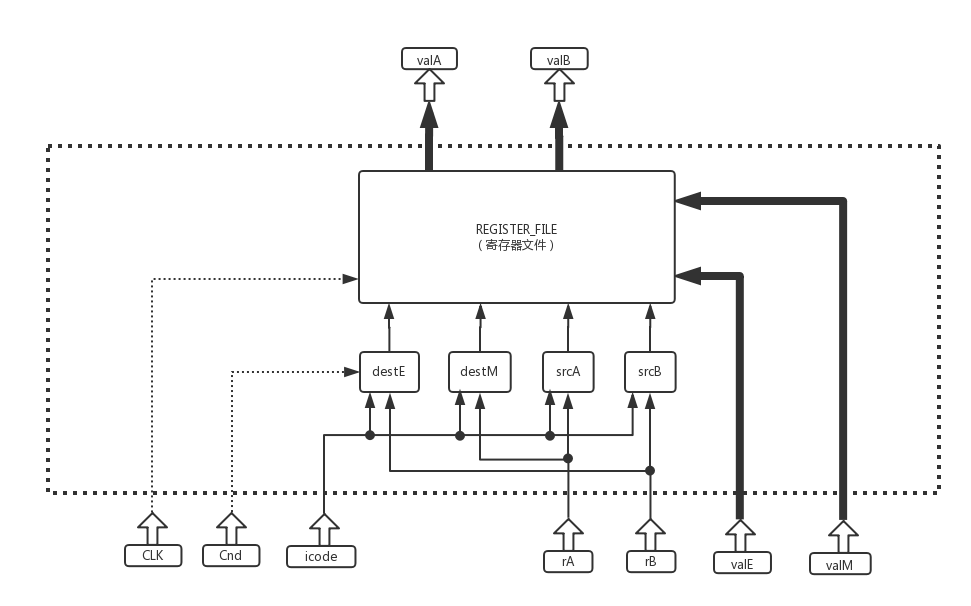
### 模块实现

在该模块内部维护者一个64位的寄存器数组，寄存器数量为16个，访问对应寄存器的方法是使用0-15的索引，因此地址使用四位的数据，输出使用64位的数据。

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，

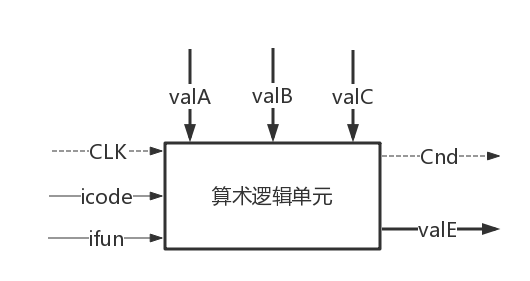
在某些特定的icode下，rA，rB从destM，destE模块输出，作为写寄存器的地址，（rA，rB为四位数据，用以访问索引为0-15的寄存器）。将valM，valE写入对应的寄存器中，该写入操作与时钟同步。

在另一些特定的icode下，rA，rB从srcA，srcB模块输出，作为读寄存器的地址，srcA作为地址读出对应寄存器的值为valA。srcB作为另一个地址读出对应寄存器的值为valB，并作为输出。该读出操作可以异步执行。



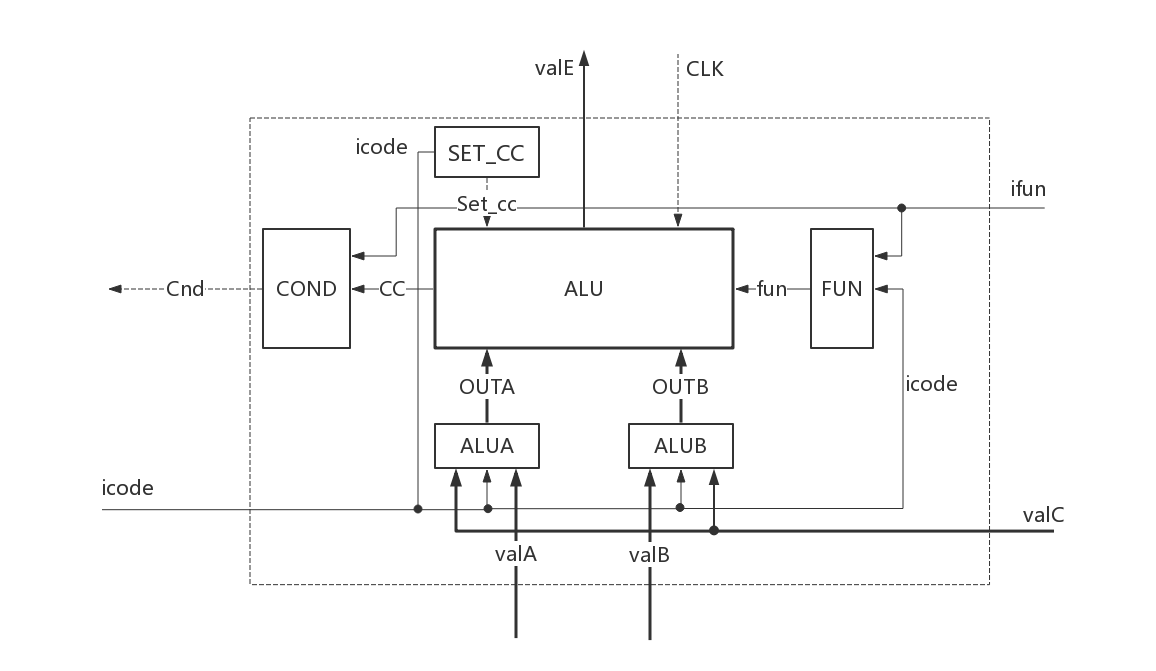
## 算术逻辑单元

### 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。

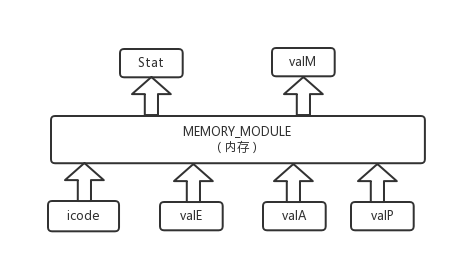
Fun模块根据指令类型（icode）和指令功能（ifun），确定ALU应该使用的功能（function）。比如指令addq应该实现加法（00），指令call应该实现减法（01），将栈指针的值减去8字节。

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

1. icode：确定是否应该进行读写操作，并将结果以flag输出
2. CLK：实现同步写入
3. read\_flag：当该flag为1的时候，进行读取操作
4. write\_flag：当该flag为1的时候，进行写入操作
5. （以上两个flag不能够同时为1）
6. valE，valA：都作为内存读写的地址输入，选择哪一个取决于icode的值
7. valA，valP：都作为内存读写的数据输入，选择哪一个取决于icode的值
8. valM：作为读取结果的输出

### 模块实现

该模块内部维护一个大小为8位的寄存器数组，实现内存以字节为单位存储。

核心模块：MEMORY

输入为write\_flag（1位），read\_flag（1位），

addr（64位），write\_data（64位）

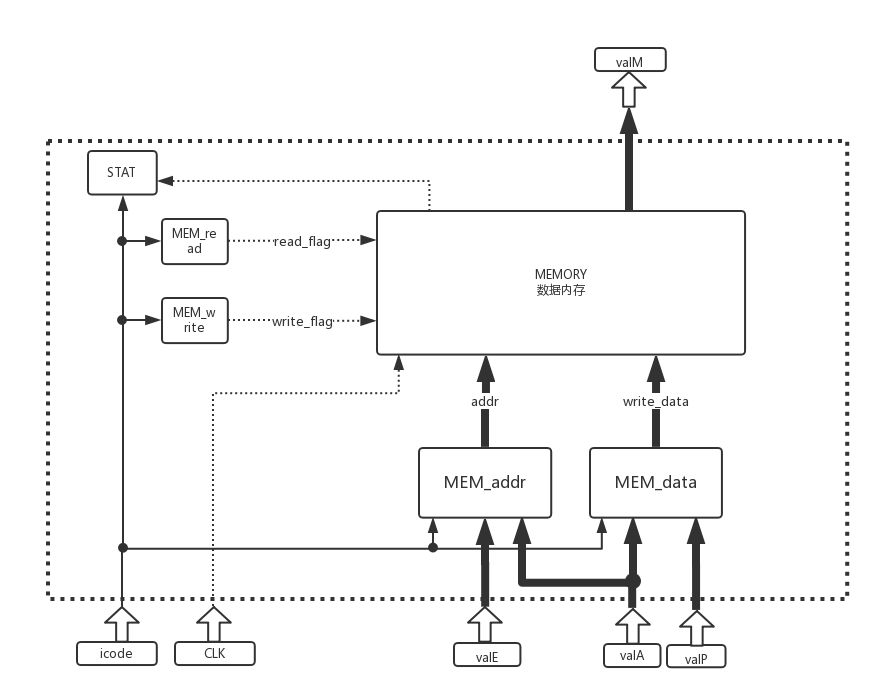
根据flag的不同，实现不同的功能。

当read\_flag为1的时候，将write\_data写入addr对应的内存位置

由于write\_data为64位的数据，占据8个字节，同时为了与其他写入读取的统一，使用小端法写入，小地址在前，往大的地址写入数据。

当read\_flag为1的时候，读取addr对应的内存数据输出到valM。

由于addr只能够确定到一个字节的数据，为了统一，仍然是使用小端法读取，小地址在前，往更大的地址读取，共读取8个字节，输出到valM。



# 测试样例

## 自然数求和（循环）

计算从1累加到10得到的值。

### 汇编及机器码

.main:

1 irmovq $(1), %rcx 30 F1 01 00 00 00 00 00 00 00

2 irmovq $(0), %rax 30 F0 00 00 00 00 00 00 00 00

3 irmovq $(0), %rdx 30 F2 00 00 00 00 00 00 00 00

.Loop:

4 irmovq $(10), %rdi 30 F7 0A 00 00 00 00 00 00 00

5 addq %rcx, %rdx 60 12

6 addq %rdx, %rax 60 20

7 subq %rdx, %rdi 61 27

8 je .End 73 40 00 00 00 00 00 00 00

9 jmp .Loop 70 1E 00 00 00 00 00 00 00

.End:

10 halt 00

### 波形图

篇幅限制，见附录B1。

### 相关分析

由附录B1波形可知，黄线条处icode为6，ifun为0，为最后一个循环时代码的第6行。此时valE输出值为。正确地计算了。随后的指令将检测边界条件，跳转到halt暂停处理器的运行。

## 求斐波那契项（递归）

利用递归的方式求斐波那契数列第4项的值。斐波那契数列为满足

的数列。

### 汇编及机器码

.main:

1 irmovq $(4), %rdi 30 F7 04 00 00 00 00 00 00 00

2 call .fun 80 14 00 00 00 00 00 00 00

3 halt 00

.fun:

4 irmovq $(1), %rcx 30 F1 01 00 00 00 00 00 00 00

5 irmovq $(2), %rdx 30 F2 02 00 00 00 00 00 00 00

6 subq %rdi, %rcx 61 71

7 je .Base 73 75 00 00 00 00 00 00 00

8 subq %rdi, %rdx 61 72

9 je .Base 73 75 00 00 00 00 00 00 00

10 irmovq $(1), %rsi 30 F6 01 00 00 00 00 00 00 00

11 pushq %rdi A0 7F

12 subq %rsi, %rdi 61 67

13 call .fun 80 14 00 00 00 00 00 00 00

14 rrmovq %rax, %rsi 20 06

15 pushq %rsi A0 6F

16 irmovq $(1), %rsi 30 F6 01 00 00 00 00 00 00 00

17 subq %rsi, %rdi 61 67

18 call .fun 80 14 00 00 00 00 00 00 00

19 popq %rsi B0 6F

20 popq %rdi B0 7F

21 addq %rsi, %rax 60 60

22 ret 90

.Base:

23 irmovq $(1), %rax 30 F0 01 00 00 00 00 00 00 00

24 ret 90

### 波形图

由于篇幅限制，见B2

### 相关分析

由附录B1波形可知，黄线条处icode为6，ifun为0，为最后一次递归执行第21行代码的波形。正确地计算出了。随后将会执行ret指令返回main函数，再执行halt停止处理器的运行。

# 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，和王永锋一拍即合。

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，我和王永锋花了很多时间在写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。很感谢王永锋，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。

# 参考文献

1. Randal E. Bryant，David R. O’Hallaron.ComPuer Systems A Programmer’s Perspective［M］.北京：机械工业出版社，2016.7

# 附录A（代码）

## 头文件（Header）

### head.v

// ADDR\_WID means address width

// the length of index for \_REG\_CODE\_

`define DATA\_WID 64

`define ADDR\_WID 4

//below defines some codes for registers

`ifndef \_REG\_CODE\_

`define \_REG\_CODE\_ 0

`define NUM\_OF\_REG 16

`define rax\_ 4'b0000

`define rcx\_ 4'b0001

`define rdx\_ 4'b0010

`define rbx\_ 4'b0011

`define rsp\_ 4'b0100

`define rbp\_ 4'b0101

`define rsi\_ 4'b0110

`define rdi\_ 4'b0111

`define r8\_ 4'b1000

`define r9\_ 4'b1001

`define r10\_ 4'b1010

`define r11\_ 4'b1011

`define r12\_ 4'b1100

`define r13\_ 4'b1101

`define r14\_ 4'b1110

`define NonReg\_ 4'b1111

`endif

// below defines some codes for instruction : icode and ifun

// below defiens some codes for flags

`ifndef \_CONST\_FLAG\_

`define \_CONST\_FLAG\_ 0

`define \_HALT 4'b0000

`define \_NOP 4'b0001

`define \_RRMOV 4'b0010

`define \_IRMOV 4'b0011

`define \_RMMOV 4'b0100

`define \_MRMOV 4'b0101

`define \_OP 4'b0110

`define \_JXX 4'b0111

`define \_CALL 4'b1000

`define \_RET 4'b1001

`define \_PUSH 4'b1010

`define \_POP 4'b1011

`define \_NONE 4'b0000

`define \_RSP 4'b0100

`define \_REGNONE 4'b1111

`define \_OK 1

`define \_ADR 2

`define \_INS 3

`define \_HLT 4

`define \_CMOVXX 4'b0010

`endif

// below define some constants for ALU

// instruction opq’s ifun may use them

`ifndef ALU\_FUN\_CODE

`define ALU\_FUN\_CODE 0

`define \_Add 2'b00

`define \_Sub 2'b01

`define \_And 2'b10

`define \_Or 2'b11

`endif

// below define some constants for relation

// L for less than, G for greater than, e for equal, n for not

`ifndef REL\_CODE

`define REL\_CODE 0

`define NonCond 0

`define REL\_LE 1

`define REL\_L 2

`define REL\_E 3

`define REL\_NE 4

`define REL\_GE 5

`define REL\_G 6

`endif

`ifndef \_CONDITION\_CODE\_

`define \_CONDITION\_CODE\_ 0

// both used in ALU’s [1:0]CC as index

// ZF for zero flag: whether the operation results in a zero

// SF for sign flag: whether get a negetive result

// OF for overflow flag: whether signed numbers cause an overflow

// CF for carry flag: whether unsigned numbers cause an overflow

`define ZF 0

`define SF 1

`define OF 2

`define CF 3

`endif

## 指令内存（Instruction Memory）

### INSTRU\_MEM\_MODULE.v （模块封装）

`timescale 1ns/1ps

`include "./../header/head.v"

module INSTRU\_MEM\_MODULE (

input [`DATA\_WID - 1: 0]PC,

output [3:0] icode,

output [3:0] ifun,

output [3:0] rA,

output [3:0] rB,

output [`DATA\_WID - 1:0]valC

);

INSTRU\_MEN instru\_men (

.PC(PC),

.icode(icode),

.ifun(ifun),

.rA(rA),

.rB(rB),

.valC(valC)

);

Endmodule

### INSTRU\_MEM.v（核心文件）

`timescale 1ns/1ps

`include "./../header/head.v"

`define INS\_LENGTH 2048

module INSTRU\_MEN (

input [`DATA\_WID - 1: 0]PC,

output reg[3:0]icode,

output reg[3:0]ifun,

output reg[3:0]rA,

output reg[3:0]rB,

output reg[`DATA\_WID - 1: 0]valC

);

reg [7:0]INSTRUCTION;

reg [7:0]REGISTER;

reg [7:0]INSTRUCTION\_MEM[0:`INS\_LENGTH - 1];

initial begin

$readmemh("F:/code/MyCPU/Instrument/instrument\_input.mem", INSTRUCTION\_MEM);

$display("%h %h %h %h", INSTRUCTION\_MEM[0], INSTRUCTION\_MEM[1], INSTRUCTION\_MEM[2], INSTRUCTION\_MEM[3]);

end

always@(\*) begin

$display("PC %h", PC[3:0]);

INSTRUCTION = INSTRUCTION\_MEM[PC];

//$display("IN %h %h", INSTRUCTION, PC);

REGISTER = INSTRUCTION\_MEM[PC + 1];

//$display("RE %h", REGISTER);

ifun = INSTRUCTION[3:0];

icode = INSTRUCTION[7:4];

rB = REGISTER[3:0];

rA = REGISTER[7:4];

if (icode == `\_JXX || icode == `\_CALL) begin

valC[7:0] = INSTRUCTION\_MEM[PC + 1];

valC[15:8] = INSTRUCTION\_MEM[PC + 2];

valC[23:16] = INSTRUCTION\_MEM[PC + 3];

valC[31:24] = INSTRUCTION\_MEM[PC + 4];

valC[39:32] = INSTRUCTION\_MEM[PC + 5];

valC[47:40] = INSTRUCTION\_MEM[PC + 6];

valC[55:48] = INSTRUCTION\_MEM[PC + 7];

valC[63:56] = INSTRUCTION\_MEM[PC + 8];

end else begin

valC[7:0] = INSTRUCTION\_MEM[PC + 2];

valC[15:8] = INSTRUCTION\_MEM[PC + 3];

valC[23:16] = INSTRUCTION\_MEM[PC + 4];

valC[31:24] = INSTRUCTION\_MEM[PC + 5];

valC[39:32] = INSTRUCTION\_MEM[PC + 6];

valC[47:40] = INSTRUCTION\_MEM[PC + 7];

valC[55:48] = INSTRUCTION\_MEM[PC + 8];

valC[63:56] = INSTRUCTION\_MEM[PC + 9];

end

end

endmodule

## 程序计数器（Program Counter）

### PC\_MODULE.v（封装模块）

`timescale 1ns/1ps

`include "../header/head.v"

module PC\_MODULE(

input CLK,

output [`DATA\_WID - 1:0]valP,

output [`DATA\_WID - 1:0]PC,

input [3:0]icode,

input Cnd,

input [`DATA\_WID - 1:0] valC,

input [`DATA\_WID - 1:0] valM

);

PC\_INCRE pc\_incre(

.valP(valP),

.icode(icode),

.PC(PC)

);

PC pc(

.CLK(CLK),

.NEW\_PC(PC),

.icode(icode),

.Cnd(Cnd),

.valC(valC),

.valM(valM),

.valP(valP)

);

endmodule

### PC.v（核心器件）

`timescale 1ns/1ps

`include "../header/head.v"

module PC (

input CLK,

output reg [`DATA\_WID - 1 : 0] NEW\_PC,

input [3:0]icode,

input Cnd,

input [`DATA\_WID - 1 : 0] valC,

input [`DATA\_WID - 1 : 0] valM,

input [`DATA\_WID - 1: 0] valP

);

initial begin

NEW\_PC = 0;

end

always @(posedge CLK) begin

if (icode == `\_CALL) begin

NEW\_PC = valC;

end else if ((icode == `\_JXX) && Cnd) begin

NEW\_PC = valC;

end else if (icode == `\_RET) begin

NEW\_PC = valM;

end else begin

NEW\_PC = valP;

end

end

endmodule

### PC\_INCRE.v（周围器件）

`timescale 1ns/1ps

`include "../header/head.v"

module PC\_INCRE (

output reg[`DATA\_WID - 1 : 0] valP,

input [3:0] icode,

input [`DATA\_WID - 1 : 0] PC

);

always@(\*) begin

case (icode)

//halt : stop and increase 0

`\_HALT : valP = PC;

`\_NOP, `\_RET : valP = PC + 1;

`\_RRMOV, `\_OP, `\_CMOVXX, `\_PUSH, `\_POP : valP = PC + 2;

`\_JXX, `\_CALL : valP = PC + 9;

`\_IRMOV, `\_RMMOV, `\_MRMOV : valP = PC + 10;

default:

valP = PC;

endcase

end

endmodule

## 寄存器文件（Register File）

### REGISTER\_FILE\_MODULE.v（封装模块）

`timescale 1ns/1ps

`include "../header/head.v"

module REGISTER\_FILE\_MODULE(

input [`ADDR\_WID - 1:0] rA,

input [`ADDR\_WID - 1:0] rB,

input CLK,

input [3:0] icode,

input Cnd,

output [`DATA\_WID - 1:0] valA,

output [`DATA\_WID - 1:0] valB,

input [`DATA\_WID - 1:0] valE,

input [`DATA\_WID - 1:0] valM

);

wire [3:0]OUTA;

wire [3:0]OUTB;

wire [3:0]OUTE;

wire [3:0]OUTM;

SRC\_A src\_a(

.rA(rA),

.icode(icode),

.OUT(OUTA)

);

SRC\_B src\_b(

.rB(rB),

.icode(icode),

.OUT(OUTB)

);

DEST\_E dest\_e(

.rB(rB),

.icode(icode),

.Cnd(Cnd),

.OUT(OUTE)

);

DEST\_M dest\_m(

.rA(rA),

.icode(icode),

.OUT(OUTM)

);

REGESTER\_FILE regester\_file(

.CLK(CLK),

.valA(valA),

.valB(valB),

.srcA(OUTA),

.srcB(OUTB),

.valE(valE),

.valM(valM),

.destE(OUTE),

.destM(OUTM)

);

endmodule

### REGISTER\_FILE.v（核心文件）

// combinational circuit when read data

// sequential circuit when write data

`timescale 1ns/1ps

`include "../header/head.v"

module REGESTER\_FILE (

input CLK,

// for read data from srcA and srcB

output reg[`DATA\_WID - 1:0]valA,

output reg[`DATA\_WID - 1:0]valB,

input [`ADDR\_WID - 1:0]srcA,

input [`ADDR\_WID - 1:0]srcB,

// for write data to register destE and destM

input [`DATA\_WID - 1:0]valE,

input [`DATA\_WID - 1:0]valM,

input [`ADDR\_WID - 1:0]destE,

input [`ADDR\_WID - 1:0]destM

);

initial begin

data[`rsp\_] = 64;

end

reg [`DATA\_WID - 1:0]data[`NUM\_OF\_REG - 1 : 0];

always@(posedge CLK) begin

data[destE] = valE;

data[destM] = valM;

end

always@(\*) begin

valA = data[srcA];

valB = data[srcB];

end

endmodule

### REGISTER\_FILE\_HELPER.v（周围器件）

`timescale 1ns/1ps

`include "../header/head.v"

module SRC\_A(

input [`ADDR\_WID - 1: 0]rA,

input [3:0] icode,

output reg [3:0] OUT

);

always@(\*) begin

case(icode)

`\_IRMOV, `\_MRMOV, `\_JXX, `\_CALL:

OUT = `NonReg\_;

`\_POP, `\_RET :

OUT = `rsp\_;

default :

OUT = rA;

endcase

end

endmodule

module SRC\_B(

input [`ADDR\_WID - 1: 0] rB,

input [3:0] icode,

output reg [3:0] OUT

);

always@(\*) begin

case (icode)

`\_RRMOV, `\_IRMOV, `\_JXX :

OUT = `NonReg\_;

`\_PUSH, `\_POP, `\_CALL, `\_RET :

OUT = `rsp\_;

default :

OUT = rB;

endcase

end

endmodule

module DEST\_E(

input [`ADDR\_WID - 1:0] rB,

input [3:0] icode,

input Cnd,

output reg [3:0] OUT

);

always@(\*) begin

case(icode)

`\_CMOVXX : begin

if (Cnd) begin

OUT = rB;

end else begin

OUT = `NonReg\_;

end

end

`\_RMMOV, `\_MRMOV, `\_JXX :

OUT = `NonReg\_;

`\_PUSH, `\_POP, `\_CALL, `\_RET :

OUT = `rsp\_;

default :

OUT = rB;

endcase

end

endmodule

module DEST\_M(

input [`ADDR\_WID - 1: 0] rA,

input [3:0] icode,

output reg [3:0] OUT

);

always@(\*) begin

case(icode)

`\_MRMOV, `\_POP :

OUT = rA;

default :

OUT = `NonReg\_;

endcase

end

endmodule

## 算术逻辑单元（Algorithm Logic Unit）

### ALU\_MODULE.v（封装模块）

`timescale 1ns/1ps

`include "../header/head.v"

module ALU\_MODULE(

input CLK,

input [3:0] icode,

input [3:0] ifun,

input [`DATA\_WID - 1:0]valA,

input [`DATA\_WID - 1:0]valB,

input [`DATA\_WID - 1:0]valC,

output Cnd,

output [`DATA\_WID - 1:0]valE

);

wire [`DATA\_WID - 1:0]OUTA;

wire [`DATA\_WID - 1:0]OUTB;

wire [1:0]OUTALUFun;

wire OUTSet\_cc;

wire OUTCond;

wire [3:0]CC;

assign Cnd = OUTCond;

ALUA alua(

.valA(valA),

.valC(valC),

.icode(icode),

.OUT(OUTA)

);

ALUB alub(

.valB(valB),

.valC(valC),

.icode(icode),

.OUT(OUTB)

);

ALUFUN alufun(

.icode(icode),

.ifun(ifun),

.OUT(OUTALUFun)

);

SET\_CC set\_cc(

.icode(icode),

.OUT(OUTSet\_cc)

);

COND cond(

.ifun(ifun),

.CC(CC),

.OUT(OUTCond)

);

ALU alu(

.CLK(CLK),

.valE(valE),

.CC(CC),

.ALUfun(OUTALUFun),

.ALUA(OUTA),

.ALUB(OUTB),

.set\_cond(OUTSet\_cc)

);

endmodule

### ALU.v（核心器件）

//WARNING: valB op valA.

//when apply to sub, it is valB minus valA

//but when apply to greater, like jg %rA, %rB, it means jump if %rB-%rA > 0,

//or if rB > rA

`timescale 1ns/1ps

`include "../header/head.v"

module ALU(

input CLK,

output reg [`DATA\_WID - 1:0]valE, //value after execute

output reg [3:0]CC, //condition code

input [1:0]ALUfun, //determine ALU function

input [`DATA\_WID - 1:0]ALUA, //input A

input [`DATA\_WID - 1:0]ALUB, //input B

input set\_cond

);

always@(\*) begin

case(ALUfun)

`\_Add : begin

valE = ALUB + ALUA;

end

`\_Sub : begin

valE = ALUB - ALUA;

end

`\_And : begin

valE = ALUB & ALUA;

end

`\_Or : begin

valE = ALUB | ALUA;

end

endcase

end

always@(posedge CLK) begin

if (set\_cond) begin

CC[`ZF] = valE == 0;

CC[`SF] = valE[`DATA\_WID - 1] == 1;

// signed overflow iff ALUA and ALUB have the same sign

// but their result has a different sign

CC[`OF] = ((ALUB[`DATA\_WID - 1]^(ALUfun == `\_Sub)) == ALUA[`DATA\_WID - 1]) && (ALUB[`DATA\_WID - 1] != valE[`DATA\_WID - 1]);

// unsigned overflow iff result less than one of operand

CC[`CF] = (valE < ALUA);

end

end

endmodule

### ALU\_HELPER.v（周围器件）

`timescale 1ns/1ps

`include "../header/head.v"

module SRC\_A(

input [`ADDR\_WID - 1: 0]rA,

input [3:0] icode,

output reg [3:0] OUT

);

always@(\*) begin

case(icode)

`\_IRMOV, `\_MRMOV, `\_JXX, `\_CALL:

OUT = `NonReg\_;

`\_POP, `\_RET :

OUT = `rsp\_;

default :

OUT = rA;

endcase

end

endmodule

module SRC\_B(

input [`ADDR\_WID - 1: 0] rB,

input [3:0] icode,

output reg [3:0] OUT

);

always@(\*) begin

case (icode)

`\_RRMOV, `\_IRMOV, `\_JXX :

OUT = `NonReg\_;

`\_PUSH, `\_POP, `\_CALL, `\_RET :

OUT = `rsp\_;

default :

OUT = rB;

endcase

end

endmodule

module DEST\_E(

input [`ADDR\_WID - 1:0] rB,

input [3:0] icode,

input Cnd,

output reg [3:0] OUT

);

always@(\*) begin

case(icode)

`\_CMOVXX : begin

if (Cnd) begin

OUT = rB;

end else begin

OUT = `NonReg\_;

end

end

`\_RMMOV, `\_MRMOV, `\_JXX :

OUT = `NonReg\_;

`\_PUSH, `\_POP, `\_CALL, `\_RET :

OUT = `rsp\_;

default :

OUT = rB;

endcase

end

endmodule

module DEST\_M(

input [`ADDR\_WID - 1: 0] rA,

input [3:0] icode,

output reg [3:0] OUT

);

always@(\*) begin

case(icode)

`\_MRMOV, `\_POP :

OUT = rA;

default :

OUT = `NonReg\_;

endcase

end

endmodule

## 内存（Memory）

### MEMORY\_MODULE.v（封装模块）

`timescale 1ns/1ps

`include "./../header/head.v"

module MEMORY\_MODULE(

input [`DATA\_WID - 1 : 0] valP,

input [`DATA\_WID - 1 : 0] valA,

input [`DATA\_WID - 1 : 0] valE,

input [`ADDR\_WID - 1 : 0] icode,

input instr\_valid,

input imem\_error,

input CLK,

output [`ADDR\_WID - 1 : 0] stat,

output [`DATA\_WID - 1 : 0] valM

);

wire [`DATA\_WID - 1 : 0] output\_data;// wire

wire [`DATA\_WID - 1 : 0] output\_addr;// wire

wire write\_flag;// wire

wire read\_flag; // wire

wire dmem\_error; // wire

MEM\_ADDR mem\_addr1(

.valE(valE),

.valA(valA),

.icode(icode),

.OUT(output\_addr)

);

MEM\_DATA mem\_data1(

.valP(valP),

.valA(valA),

.icode(icode),

.OUT(output\_data)

);

MEM\_READ mem\_read1(

.icode(icode),

.read(read\_flag)

);

MEM\_WRITE mem\_write1(

.icode(icode),

.write(write\_flag)

);

MEMORY memory1(

.write\_data(output\_data),

.addr(output\_addr),

.write\_flag(write\_flag),

.read\_flag(read\_flag),

.CLK(CLK),

.valM(valM),

.dmem\_error(dmem\_error)

);

STAT stat1(

.instr\_valid(instr\_valid),

.imem\_error(imem\_error),

.icode(icode),

.dmem\_error(dmem\_error),

.stat(stat)

);

assign instr\_valid = 1;

assign imem\_error = 0;

endmodule

### MEMORY.v（核心器件）

// edit by wyf

// 2017/6/7 22:10

`timescale 1ns/1ps

`include "./../header/head.v"

`define SIZE\_OF\_MEMORY 96

module MEMORY(

input wire CLK,

input wire [`DATA\_WID - 1 : 0] write\_data,

input wire [`DATA\_WID - 1 : 0] addr,

input wire write\_flag,

input wire read\_flag,

output reg [`DATA\_WID - 1 : 0] valM,

output wire dmem\_error

);

reg [8 - 1 :0] data[`SIZE\_OF\_MEMORY : 0];

assign dmem\_error = (addr > `SIZE\_OF\_MEMORY)?1:0;

always@(posedge CLK) begin

if (write\_flag == 1) begin

data[addr+7] = write\_data[63:56];

data[addr+6] = write\_data[55:48];

data[addr+5] = write\_data[47:40];

data[addr+4] = write\_data[39:32];

data[addr+3] = write\_data[31:24];

data[addr+2] = write\_data[23:16];

data[addr+1] = write\_data[15:8];

data[addr] = write\_data[7:0];

end

else begin

data[addr] = data[addr];

end

end

always@(\*) begin

if (read\_flag == 1) begin

valM = data[addr];//

valM[63:56] = data[addr+7];

valM[55:48] = data[addr+6];

valM[47:40] = data[addr+5];

valM[39:32] = data[addr+4];

valM[31:24] = data[addr+3];

valM[23:16] = data[addr+2];

valM[15:8] = data[addr+1];

valM[7:0] = data[addr ];

end

else begin

valM = valM;

end

end

endmodule // memory

### MEMORY\_HELPER.v（周围器件）

// edit by wyf

// 2017/6/7 22:10

`timescale 1ns/1ps

`include "./../header/head.v"

module MEM\_ADDR(

input wire [`DATA\_WID - 1 : 0] valE,

input wire [`DATA\_WID - 1 : 0] valA,

input wire [`ADDR\_WID - 1 : 0] icode,

output reg [`DATA\_WID - 1 : 0 ] OUT

);

always@(\*) begin

case (icode)

`\_RMMOV, `\_PUSH, `\_CALL, `\_MRMOV :

OUT = valE;

`\_POP, `\_RET:

OUT = valA;

default :

OUT = OUT;

// TODO: if it don't need the address.

endcase

end

endmodule //

module MEM\_DATA(

input wire [`DATA\_WID - 1 : 0] valP,

input wire [`DATA\_WID - 1 : 0] valA,

input wire [`ADDR\_WID - 1 : 0] icode,

output reg [`DATA\_WID - 1 : 0] OUT

);

always@(\*) begin

case(icode)

`\_RMMOV, `\_PUSH:

OUT = valA;

`\_CALL:

OUT = valP;

// `\_MRMOV, `\_RET, `\_POP:

default:

OUT = OUT;

// TODO: if it don't write data.

endcase

end

endmodule // MEM\_DATA

module MEM\_READ(

input wire [`ADDR\_WID - 1 : 0] icode,

output reg read

);

always@(\*) begin

case(icode)

`\_MRMOV, `\_RET, `\_POP:

read = 1;

default:

read = 0;

endcase

end

endmodule // MEM\_READ

module MEM\_WRITE(

input wire [`ADDR\_WID - 1 : 0] icode,

output reg write

);

always@(\*) begin

case(icode)

`\_RMMOV, `\_PUSH, `\_CALL:

write = 1;

default:

write = 0;

endcase

end

endmodule // MEM\_WRITE

module STAT(

input wire instr\_valid,

input wire imem\_error,

input wire [`ADDR\_WID - 1 : 0] icode,

input wire dmem\_error,

output reg [`ADDR\_WID - 1 : 0] stat

);

// `OK `ADR `INS `HLT

initial begin

stat = 4'b0001;

end

always@(\*) begin

case (icode)

`HALT:

stat = `HLT;

default:

stat = `OK;

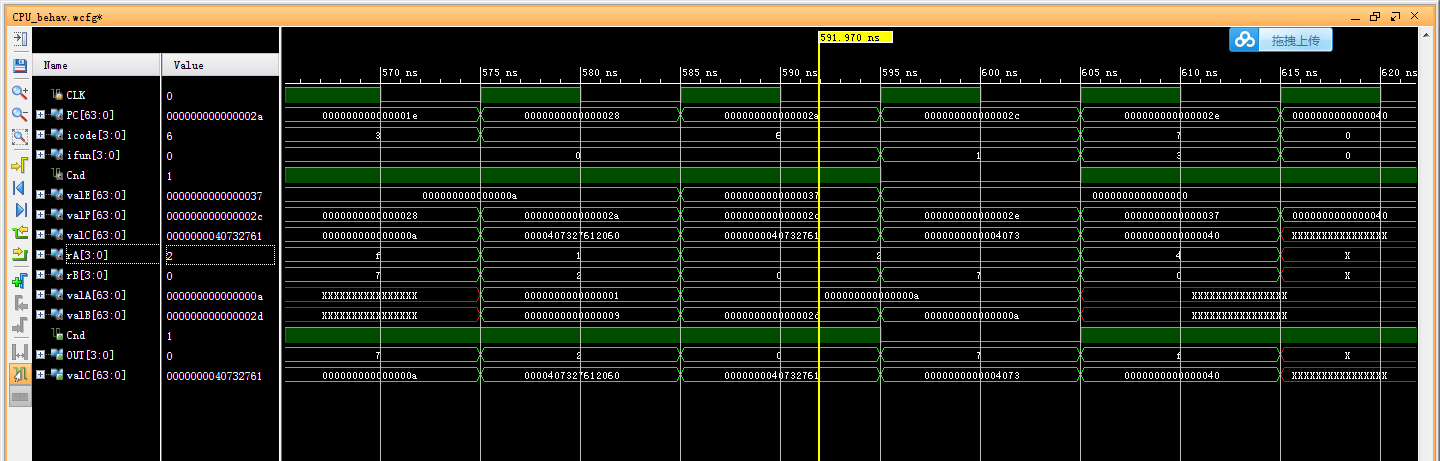
endcase

end

endmodule // STAT

# 附录B（波形图）

## 程序一波形图



## 程序二波形图

