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 16337269

王永锋 16337237

2017 年 6 月 18 日

目录

1.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

2.指令集及其格式

2.1 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

2.2 指令格式

以下使用十六进制数代表 4bit

2.2.1 类型 A

格式 icode ifun

指令	4	4
halt	0	0
nop	1	0
ret	9	0

2.2.2 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

2.2.3 类型 C

格式 icode ifun valC

指令	4	4	8X8
jXX	7	x	Dest
call	8	0	Dest

2.2.4 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

2.3 用途简介

2.3.1 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。

2.3.2 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

2.3.3 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。

2.3.4 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

2.4 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

3.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。

3.1 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

3.2 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

3.3 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

3.4 内存操作指令系列

阶段	**rmmovq rA, D(rB)**	**mrmovq D(rB), rA**
取指	icode：fun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_8[PC+2] valP ← PC+10	icode：fun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_8[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	M_8[valE] ← valA	valM ← M_8[valE]
写回		R[rA] ← valM
PC	PC ← valP	PC ← valP

3.5 栈指令系列

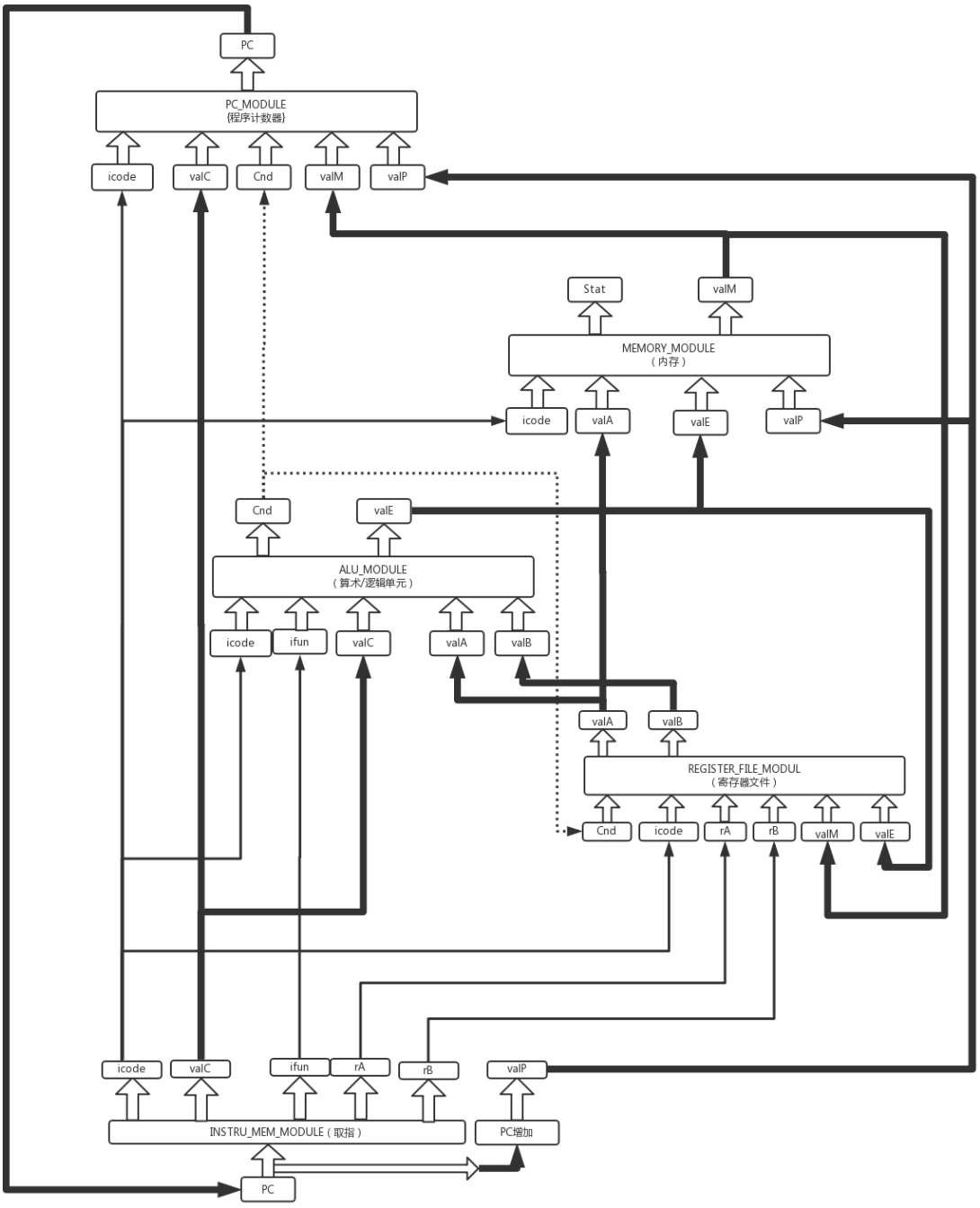
阶段	**pushq rA**	**popq rA**
取指	icode：fun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	icode：fun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2
译码	valA ← R[rA] valB ← R[%rsp]	valA ← R[%rsp] valB ← R[%rsp]
执行	valE ← valB + (-8)	valE ← valB + 8
访存	M_8[valE] ← valA	valM ← M_8[valA]
写回	R[%rsp] ← valE	R[%rsp] ← valE R[rA] ← valM
PC	PC ← valP	PC ← valP

3.6 跳转指令系列

阶段	**jXX Dest**	**call Dest**	**ret**
取指	icode：fun←M_1[PC] valC ← M_8[PC+1] valP ← PC+9	icode：fun ← M_1[PC] valC ← M_8[PC+1] valP ← PC+9	icode：fun ← M_1[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		M_8[valE] ← valP	valM ← M_8[valA]
写回		R[%rsp] ← valE	R[%rsp] ← valE
PC	PC ← Cnd?valC:valP	PC ← valC	PC ← valM

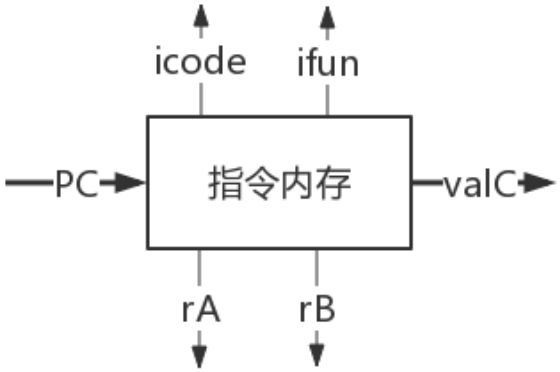
4.CPU 模块构成

4.1 CPU 总模块组成



4.2 指令内容：取指模块

4.2.1 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）

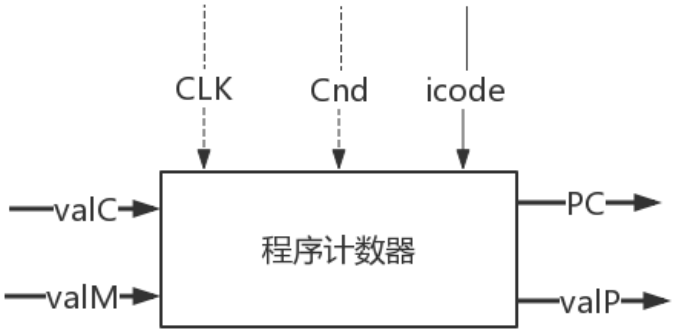
4.2.2 模块实现

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。

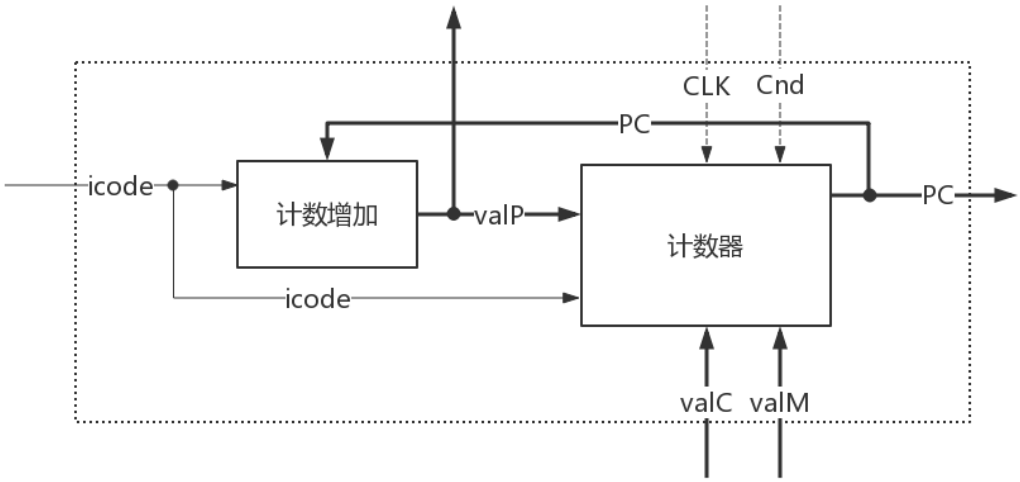
4.3 程序计数器

4.3.1 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

4.3.2 模块实现

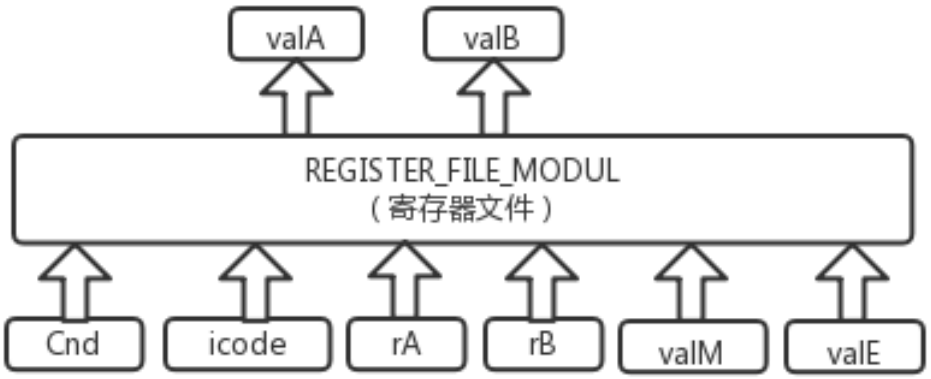


I 指令的相邻指令指在指令内存中与 I 紧挨着的下一条指令的地址。I 指令的下一条地址指 CPU 实际工作时，执行完 I 指令后执行的地址。

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）。

4.4 寄存器文件



4.4.1 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

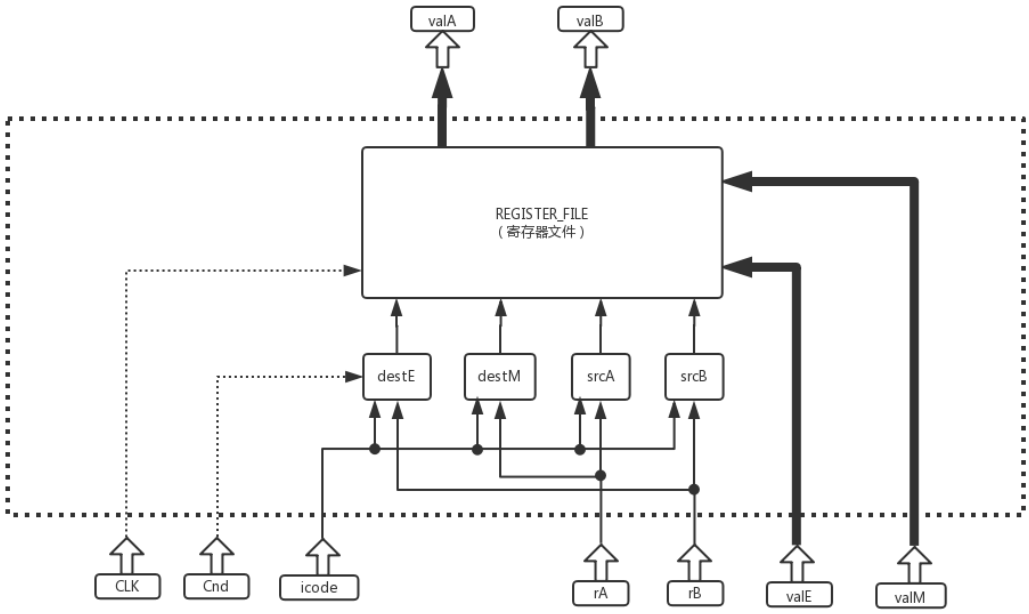
4.4.2 模块实现

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，

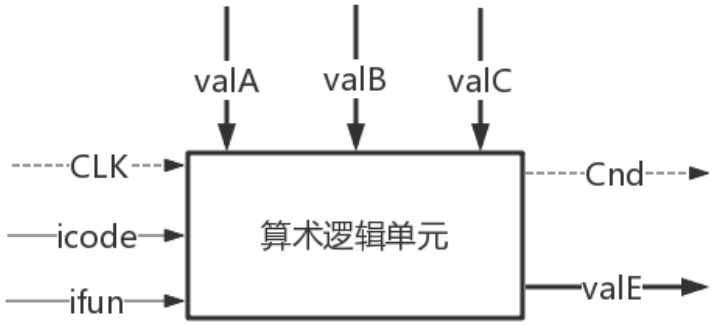
在某些特定的 icode 下，rA，rB 从 destM，destE 模块输出，作为写寄存器的地址，(rA，rB 为四位数据，用以访问索引为 0-15 的寄存器)。将 valM，valE 写入对应的寄存器中，该写入操作与时钟同步。

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。



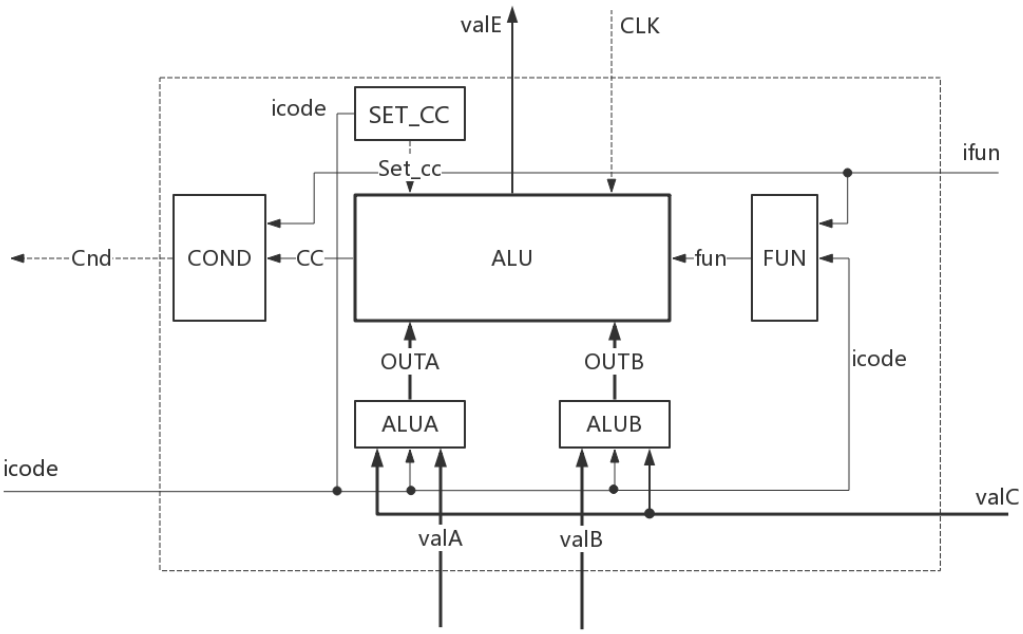
4.5 算术逻辑单元

4.5.1 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

4.5.2 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。

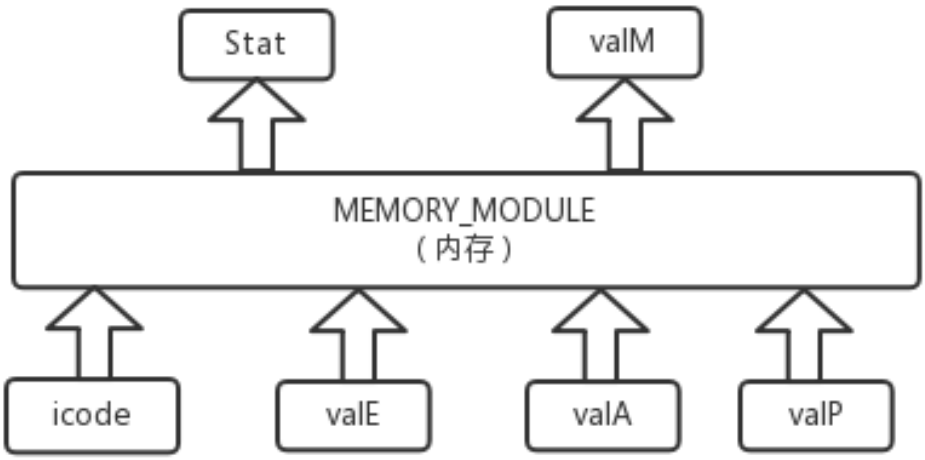
ALU 模块在计算 valE 的同时会设置条件码（condition code，CC），记录 valE 的值是否为 0，是否为负，计算是否发生溢出等。ALU 在输出 valE 的同时还会输出 CC，用于计算 Cond

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。

4.6 内存



4.6.1 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

4.6.2 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

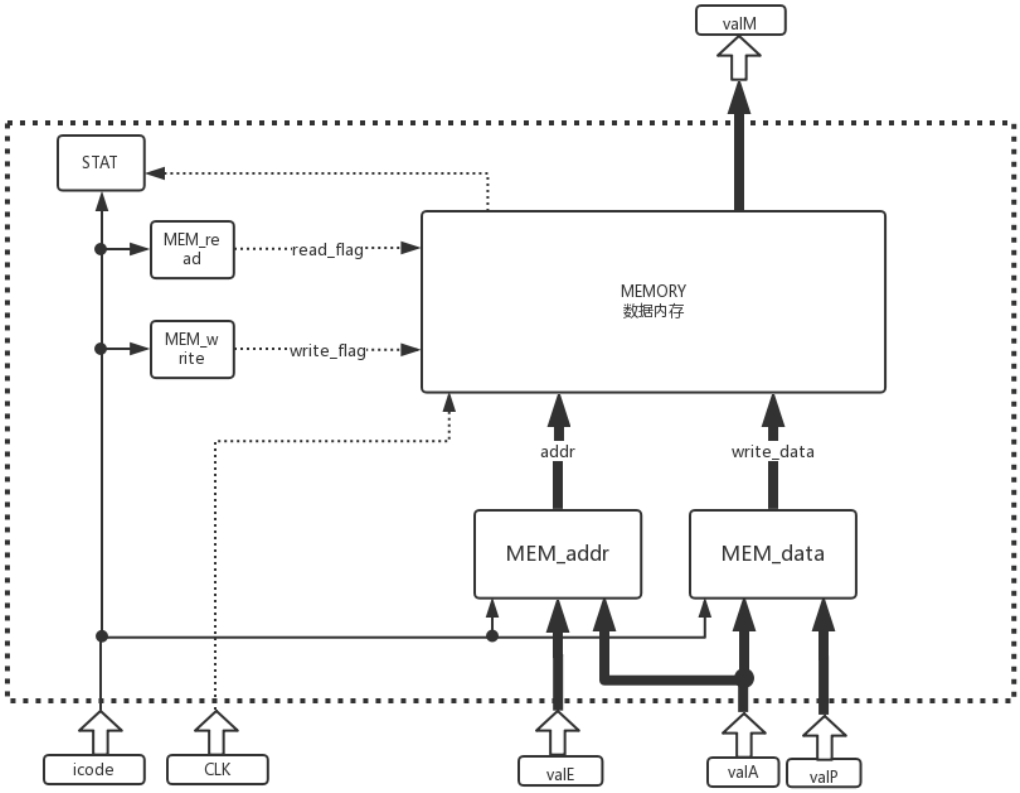
输入为 write_flag（1 位），read_flag（1 位），

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。



5.测试样例

5.1 自然数求和（循环）

计算从 1 累加到 10 得到的值。

5.1.1 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
```

5.1.2 波形图

篇幅限制，见附录 B1。

5.1.3 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37_{16}，即55_{10}。正确地计算了$\Sigma_{i=1}^{10} i$。随后的指令将检测边界条件，跳转到 halt 暂停处理器的运行。

5.2 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

$$a_1 = 1, a_2 = 1 \text{ 且} a_n = a_{n-1} + a_{n-2}$$

的数列。

5.2.1 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
```

5.2.2 波形图

由于篇幅限制，见 B2

5.2.3 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$a_4 = 3$。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。

6.心得体会

6.1 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！

6.2 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。

7.参考文献

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

8.附录 A（代码）

8.1 头文件（Header）

8.1.1 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
```

8.2 指令内存（Instruction Memory）

8.2.1 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
```

8.2.2 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
```

8.3 程序计数器（Program Counter）

8.3.1 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
```

8.3.2 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
```

8.3.3 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
```

8.4 寄存器文件（Register File）

8.4.1 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
```

8.4.2 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
```

8.4.3 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
```

8.5 算术逻辑单元（Algorithm Logic Unit）

8.5.1 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
```

8.5.2 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
```

8.5.3 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
```

8.6 内存（Memory）

8.6.1 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
```

8.6.2 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
```

8.6.3 MEMORY_HELPER.v（周围器件）

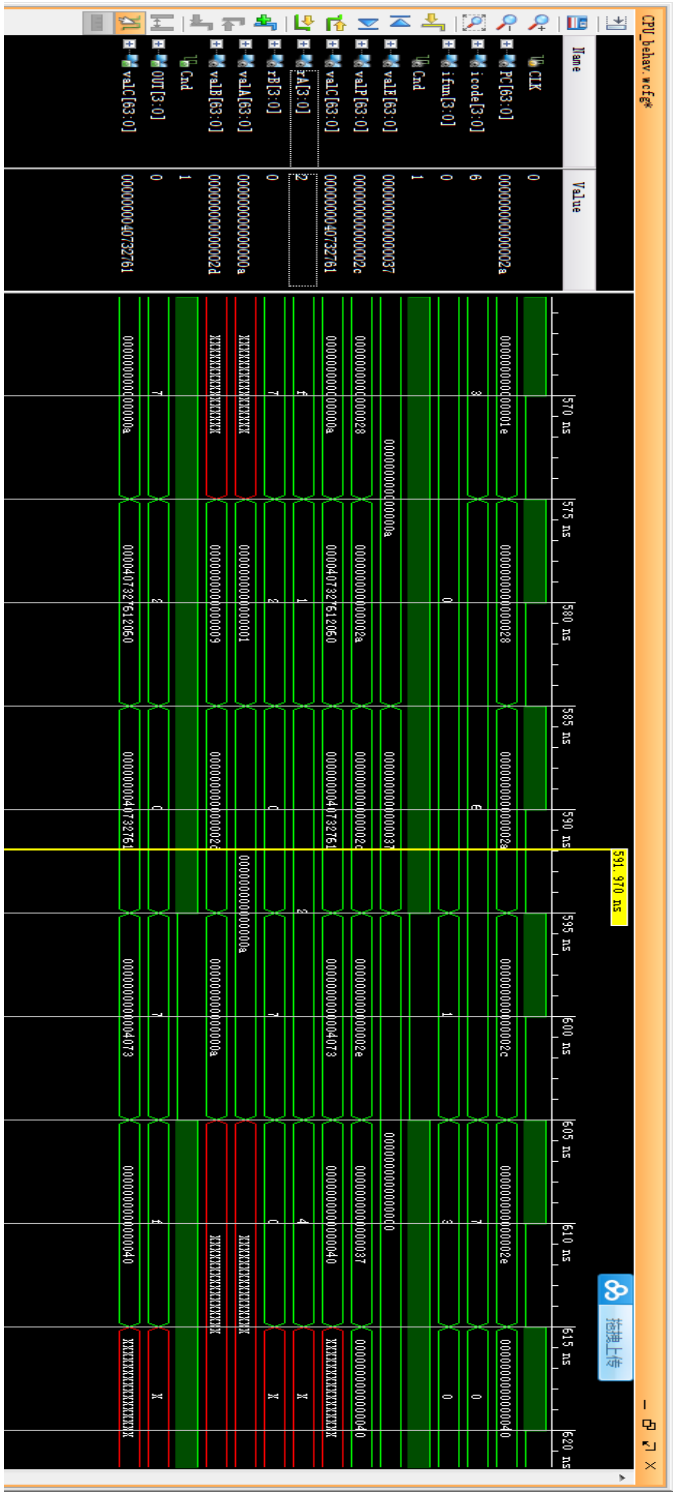
```
// edit by wyf
// 2017/6/7 22:10
`timescale 1ns/1ps
`include "./../header/head.v"
module MEM_ADDR(
    input wire [`DATA_WID - 1 : 0] valE,
    input wire [`DATA_WID - 1 : 0] valA,
    input wire [`ADDR_WID - 1 : 0] icode,
    output reg [`DATA_WID - 1 : 0 ] OUT
);
    always@(*) begin
        case (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
```

9.附录 B（波形图）

9.1 程序一波形图



9.2 程序二波形图