**数字电路与逻辑设计**

**单周期CPU设计**

学院：数据科学与计算机学院

专业：软件工程

班别：教务二班软工4班

姓名：郑佳豪

学号：16305204

时间：2018年7月8号

目录

[**单周期CPU** 5](#_Toc519019553)

[概念 5](#_Toc519019554)

[工作原理 5](#_Toc519019555)

[**MIPS指令** 6](#_Toc519019556)

[介绍 6](#_Toc519019557)

[指令类型 6](#_Toc519019558)

[常见运算指令 7](#_Toc519019559)

[算术运算指令 7](#_Toc519019560)

[逻辑运算指令 7](#_Toc519019561)

[移位指令 7](#_Toc519019562)

[比较指令 8](#_Toc519019563)

[存储器读/写指令 8](#_Toc519019564)

[分支指令 8](#_Toc519019565)

[跳转指令 8](#_Toc519019566)

[停机指令 8](#_Toc519019567)

[**设计思路** 9](#_Toc519019568)

[数据通路 9](#_Toc519019569)

[模块设计 10](#_Toc519019570)

[指令宏 11](#_Toc519019571)

[Control Unit 11](#_Toc519019572)

[功能介绍 11](#_Toc519019573)

[代码实现 12](#_Toc519019574)

[PC 13](#_Toc519019575)

[功能介绍 13](#_Toc519019576)

[代码实现 13](#_Toc519019577)

[Decoder 14](#_Toc519019578)

[功能说明 14](#_Toc519019579)

[输入说明 14](#_Toc519019580)

[输出说明 14](#_Toc519019581)

[代码实现 15](#_Toc519019582)

[InstructionMemory 16](#_Toc519019583)

[功能说明 16](#_Toc519019584)

[输入说明 16](#_Toc519019585)

[输出说明 16](#_Toc519019586)

[代码实现 16](#_Toc519019587)

[RegFile 17](#_Toc519019588)

[功能说明 17](#_Toc519019589)

[输入说明 17](#_Toc519019590)

[输出说明 17](#_Toc519019591)

[代码实现 17](#_Toc519019592)

[DataMemory 18](#_Toc519019593)

[功能说明 18](#_Toc519019594)

[输入说明 18](#_Toc519019595)

[输出说明 18](#_Toc519019596)

[代码实现 18](#_Toc519019597)

[ALU 19](#_Toc519019598)

[功能说明 19](#_Toc519019599)

[输入说明 19](#_Toc519019600)

[输出说明 19](#_Toc519019601)

[ALU功能 19](#_Toc519019602)

[代码实现 20](#_Toc519019603)

[Extend 21](#_Toc519019604)

[功能说明 21](#_Toc519019605)

[输入说明 21](#_Toc519019606)

[输出说明 21](#_Toc519019607)

[代码实现 21](#_Toc519019608)

[顶层CPU模块 21](#_Toc519019609)

[功能说明 21](#_Toc519019610)

[代码实现 22](#_Toc519019611)

[仿真文件 24](#_Toc519019612)

[功能说明 24](#_Toc519019613)

[代码实现 24](#_Toc519019614)

[**功能测试** 25](#_Toc519019615)

[测试文件 25](#_Toc519019616)

[功能验证 25](#_Toc519019617)

[寄存器内存检查 37](#_Toc519019618)

[**实验心得** 38](#_Toc519019619)

# **单周期CPU**

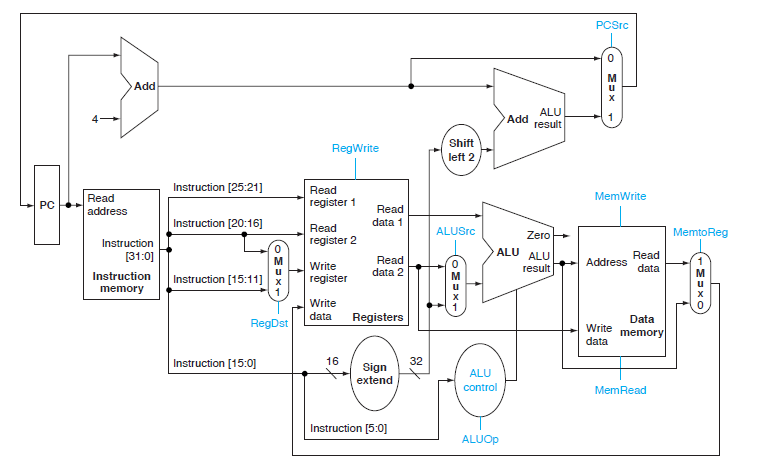
## 概念

单周期CPU指的是一条指令的执行在一个时钟周期内完成，然后开始下一条指令的执

行，即一条指令用一个时钟周期完成。电平从低到高变化的瞬间称为时钟上升沿，两个相

邻时钟上升沿之间的时间间隔称为一个时钟周期。

## 工作原理



根据上图，我们知道单周期CPU在处理指令时，一般需要经过以下几个步骤：

* 取指令(IF)：根据程序计数器(PC)中的指令地址，从存储器中取出一条指令，同时PC根据指令字长度自动递增产生下一条指令所需要的指令地址，但遇到地址转移指令时，则控制器把转移地址送入PC，当然得到的地址需要处理才送入PC。
* 指令译码(ID)：对从PC中取到的指令进行译码，产生相应的操作控制信号。
* 指令执行(EXE)：根据指令译码得到的操作控制信号，执行相应的动作。
* 存储器访问(MEM)：所以需要访问存储器的操作都在此进行，把数据写入到存储器中数据地址所指定的存储单元或者从存储器中得到数据地址单元中的数据。
* 结果写回(WB)：指令执行的结果或者访问存储器中得到的数据写回相应的目的寄存器中。

# **MIPS指令**

## 介绍

MIPS(Microprocessor without Interlocked Pipeline Stages)，是一种采取精简指令集(RISC)的处理器架构，于1981年出现，由MIPS科技公司开发并授权，广泛被使用在许多电子产品、网络设备、个人娱乐设备和商业设备中。

关于MIPS的更详细的信息，请查阅<https://en.wikipedia.org/wiki/MIPS_architecture>。

## 指令类型

在MIPS架构中，指令被分为三种类型：R型、I型和J型。三种类型的指令的最高6位均为6位的opcode码。从25位往下，

* R型指令用连续三个5位二进制码来表示三个寄存器的地址，然后用一个5位二进制码来表示移位的位数（如果未使用移位操作，则全为0），最后为6位的function码（它与opcode码共同决定R型指令的具体操作方式）；
* I型指令则用连续两个5位二进制码来表示两个寄存器的地址，然后是一个16位二进制码来表示的一个立即数二进制码；
* J型指令用26位二进制码来表示跳转目标的指令地址（实际的指令地址应为32位，其中最低两位为00，高四位由PC当前地址决定）

MIPS使用的是大端存储模式，关于大端存储的更详细信息，请查阅<https://en.wikipedia.org/wiki/Endianness#Big-endian>

**R类型**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Type | -31- Format(bits) -0- | | | | | |
| R | opcode(6) | rs(5) | rt(5) | rd(5) | shamt(5) | funct(6) |

**I类型**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Type | -31- Format(bits) -0- | | | |
| I | opcode(6) | rs(5) | rt(5) | immediate(16) |

**J类型**

|  |  |  |
| --- | --- | --- |
| Type | -31- Format(bits) -0- | |
| J | opcode(6) | address(26) |

上述类型指示图中的字母简写含义如下：

* opcode: 6位操作码
* rs: 第1个源操作数寄存器，寄存器地址编号为 00000-11111，即 00-1F
* rt: 第2个源操作数寄存器或目的操作数寄存器，寄存器地址编号为 00-1F
* rd: 目的操作数寄存器，寄存器地址编号为 00-1F
* shamt: 位移量(Shift Amount)，在移位指令中用于指定移动的位数
* funct: 为功能码，在R类型指令中用来指定指令功能
* immediate: 16位立即数，用于无符号的逻辑操作数、有符号的算术操作数、数据加载和数据保存指令的数据地址字节偏移量、分支指令中相对PC的有符号偏移量

## 常见运算指令

### 算术运算指令

* add rd, rs, rt

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 000000 | rs(5) | rt(5) | rd(5) | Reserved |

功能：rd = rs + rt。reserved为预留部分，即未用，一般填“0”

* addi rt, rs, immediate

|  |  |  |  |
| --- | --- | --- | --- |
| 000001 | rs(5) | rt(5) | immediate(16) |

功能：rt = rs + (sign-extend)immediate；对immediate符号扩展，进行加运算

* sub rd, rs, rt

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 000010 | rs(5) | rt(5) | rd(5) | Reserved |

功能：rt = rs + (sign-extend)immediate；对immediate符号扩展，进行加运算

### 逻辑运算指令

* ori rt, rs, immediate

|  |  |  |  |
| --- | --- | --- | --- |
| 010000 | rs(5) | rt(5) | immediate(16) |

功能：rt = rs | (zero-extend)immediate；对immediate零扩展，进行或运算

* and rd, rs, rt

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 010001 | rs(5) | rt(5) | rd(5) | reserved |

功能：rd = rs & rt；逻辑与运算

* or rd, rs, rt

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 010010 | rs(5) | rt(5) | rd(5) | reserved |

功能：rd = rs | rt；逻辑或运算

### 移位指令

* sll rd, rt, shamt

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 011000 | 未用 | rt(5) | rd(5) | shamt | reserved |

功能：rd = rt << (zero-extend)shamt，左移(zero-extend)shamt位

### 比较指令

* slt rd, rs, rt 带符号数

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 011100 | rs(5) | rt(5) | rd(5) | reserved |

功能：rd = rs < rt ? 1 : 0

### 存储器读/写指令

* sw rt, immediate(rs) 写存储器

|  |  |  |  |
| --- | --- | --- | --- |
| 100110 | rs(5) | rt(5) | immediate(16) |

功能：memory[rs+ (sign-extend)immediate] = rt

* lw rt, immediate(rs) 读存储器

|  |  |  |  |
| --- | --- | --- | --- |
| 100111 | rs(5) | rt(5) | immediate(16) |

功能：rt = memory[rs + (sign-extend)immediate]

### 分支指令

* beq rs, rt, immediate

|  |  |  |  |
| --- | --- | --- | --- |
| 110000 | rs(5) | rt(5) | immediate(偏移量，16) |

功能：if(rs = rt) pc = pc + 4 + (sign-extend)immediate <<2；

* bne rs, rt, immediate

|  |  |  |  |
| --- | --- | --- | --- |
| 110001 | rs(5) | rt(5) | immediate(16) |

功能：if(rs != rt) pc = pc + 4 + (sign-extend)immediate <<2 else pc = pc + 4

* bgtz rs, immediate

|  |  |  |  |
| --- | --- | --- | --- |
| 110010 | rs(5) | 00000 | immediate(16) |

功能：if(rs>0) pc = pc + 4 + (sign-extend)immediate <<2 else pc ←pc + 4

### 跳转指令

* j addr

|  |  |
| --- | --- |
| 111000 | addr[27..2] |

功能：pc = {(pc+4)[31..28], addr[27..2], 0, 0}，无条件跳转。

### 停机指令

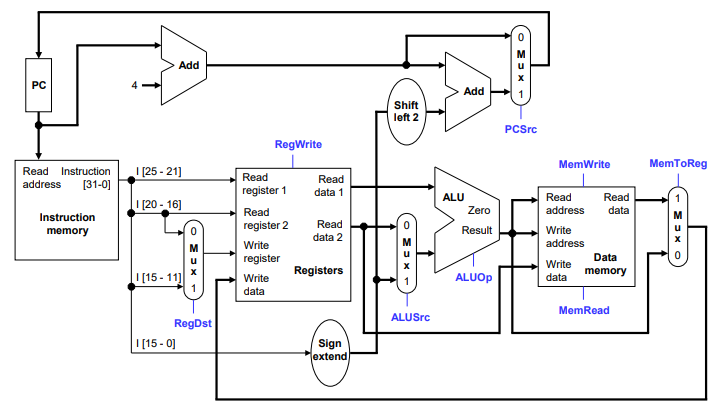
* halt

|  |  |
| --- | --- |
| 111111 | 00000000000000000000000000(26) |

功能：停机，pc保持不变。

# **设计思路**

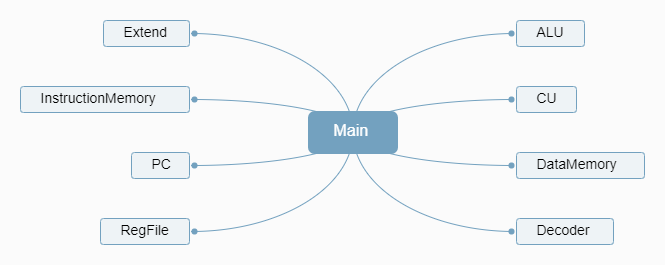
## 数据通路



上图对我们编写Verilog代码，实现单周期CPU是至关重要的。下面是对各控制信号的说明：

|  |  |  |
| --- | --- | --- |
| 控制信号 | 状态0 | 状态1 |
| Reset | 初始化PC为0 | PC接收新地址 |
| ALUSrc | 来自寄存器data2输出，如addu、subu、or、and、bne | 来自sign或zero扩展的立即数，如addiu、ori、sw、lw |
| RegWrite | 不写入寄存器，如beq、sw、halt | 写入寄存器，如add、addi、sub、ori、or、and、sll、lw |
| MEmtoreg | 来自ALU输出，除lw以外 | 来自数据存储器输出，如lw |
| memread | 读数据存储器，如lw | 输出高阻态 |
| memwrite | 写入数据存储器，如sw | 无操作 |
| ExtSel | 立即数零扩展，如ori | 立即数符号扩展，如addi、sw、lw、beq |
| RegDst | 写寄存器组寄存器的地址，如addi、ori、lw | 写寄存器组寄存器的地址，如add、sub、and、or、sll |
| PCSrc | PC=PC+4，如add、addi、sub、ori、or、and、sw、sll、lw、beq(zero=0) | PC=PC+4+符号扩展后的立即数，如beq(zero=1) |
| ALUOp [2:0] | 选择ALU功能 | |

## 模块设计

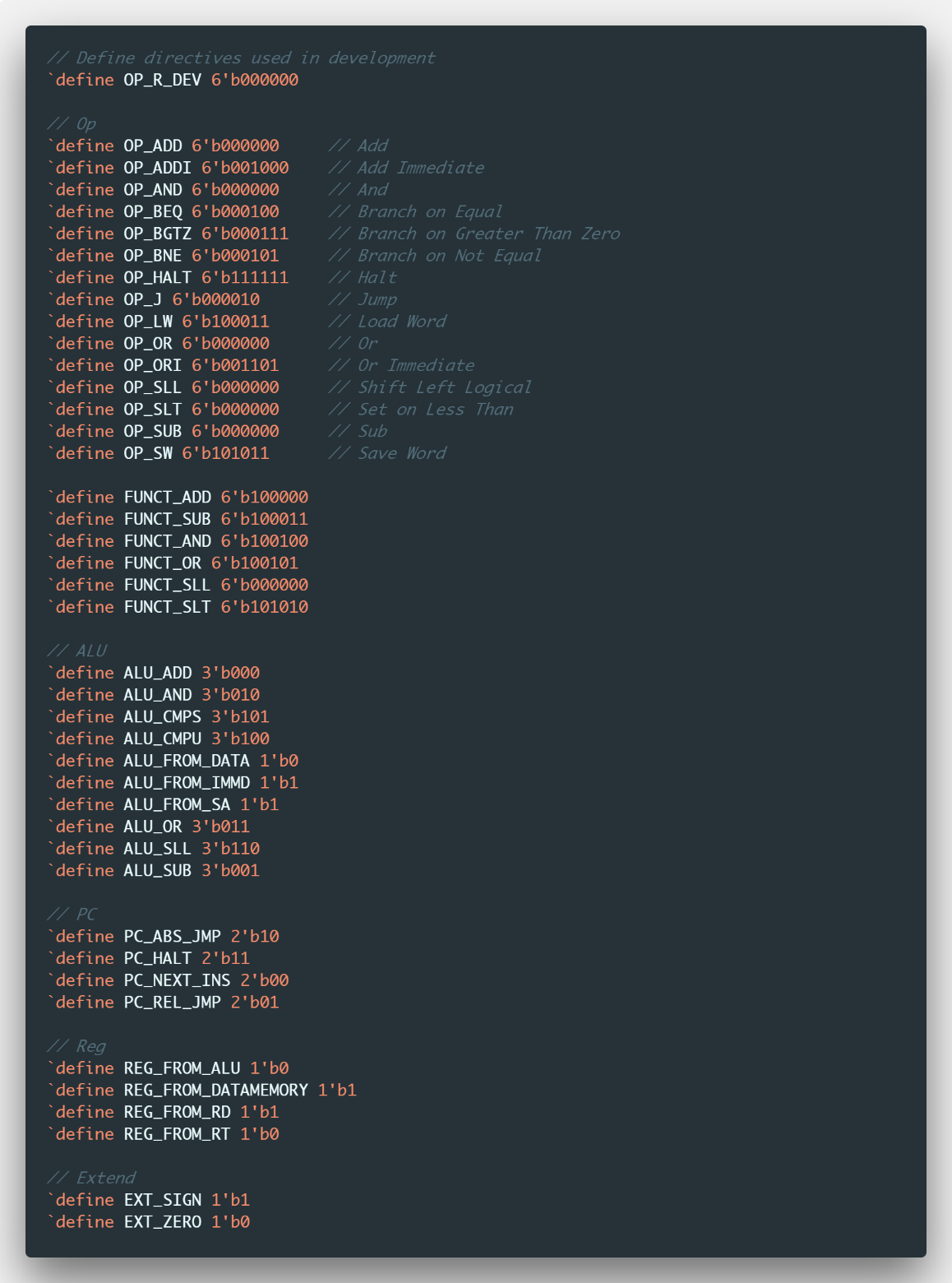


参照上面提及到的数据通路，我将项目拆分为上图所示的几个部分。Main为顶层模块，用来将其余八大模块联系。

* CU：控制数据路线
* ALU：负责算术逻辑运算
* DataMemory：数据存储器
* Decoder：负责按照MIPS指令要求解析指令
* Extend：对立即数进行零扩展和符号扩展
* InstructionMemory：指令存储器
* PC：负责程序要处理指令的提供
* RegFile：寄存器堆

## 指令宏

为了开发的方便，我在项目开发使用了宏，即将易输错的机器码，使用别名代替，从而减少项目中隐藏的错误发生的概率。

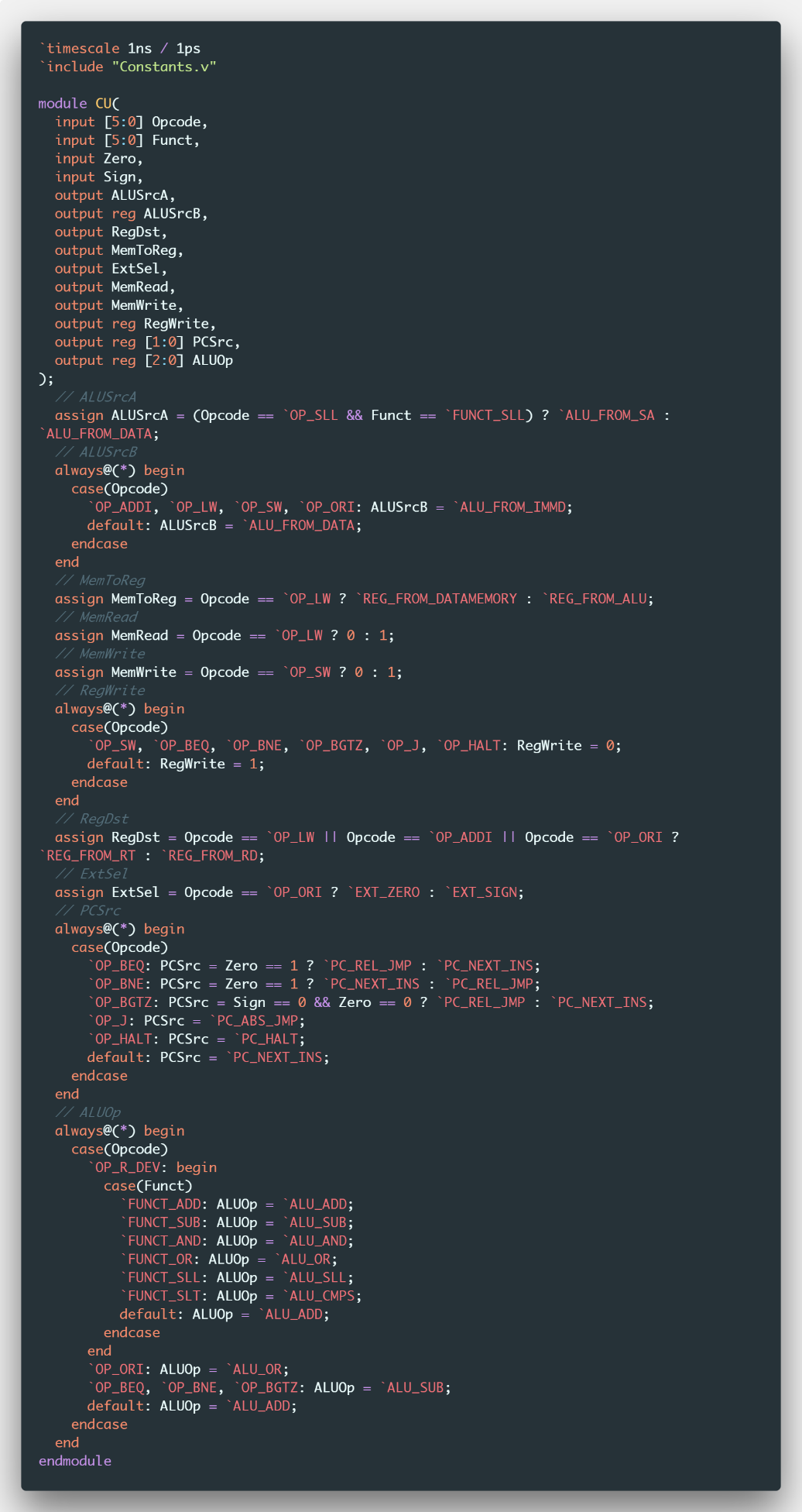


## Control Unit

### 功能介绍

控制数据通路的路线，即控制其余各模块的数据通路。

### 代码实现



## PC

### 功能介绍

PC负责为处理器提供要处理的下一指令的地址，在本项目中我将PC模块拆分为两个模块，一个模块PCHelper根据正在执行的指令，计算下一条指令的地址，另个模块PC 则将 PCHelper计算的地址在时钟上升沿或重置下降沿时作为下一周期执行的指令。

### 代码实现



## Decoder

### 功能说明

将从InstructionMemory得到的指令进行解析，输出opcode、rs、rt、rd、shamt、funct、immediate、address。

### 输入说明

|  |  |
| --- | --- |
| 输入 | 说明 |
| Ins [31:0] | 欲进行解析的指令 |

### 输出说明

|  |  |
| --- | --- |
| 输出 | 说明 |
| Opcode [5:0] | 6位操作码 |
| Funct [5:0] | 功能码 |
| Shamt [4:0] | 位移量Shift Amount |
| rs [4:0] | 第1个源操作数寄存器 |
| rt [4:0] | 第2个源操作数寄存器或目的操作数寄存器 |
| rd [4:0] | 目的操作数寄存器 |
| Immediate [15:0] | 16位立即数 |
| Address [25:0] | 地址 |

### 代码实现



## InstructionMemory

### 功能说明

根据当前的PC地址，得到对应地址的指令。

### 输入说明

|  |  |
| --- | --- |
| 输入 | 说明 |
| Address [31:0] | 执行读取的指令地址 |

### 输出说明

|  |  |
| --- | --- |
| 输出 | 说明 |
| Ins [31:0] | 输出读取出来的指令 |

### 代码实现



## RegFile

### 功能说明

为其余模块提供读取和写入寄存器的功能。

### 输入说明

|  |  |
| --- | --- |
| 输入 | 说明 |
| CLK | 时钟信号 |
| Reset | 复位信号 |
| RegWrite | 使能信号，为0时不写入，为1时写入数据 |
| ReadReg1 | 读数据地址输入端 |
| ReadReg2 | 读数据地址输入端 |
| WriteReg | 写数据地址输入端 |
| WriteData | 写入数据 |

### 输出说明

|  |  |
| --- | --- |
| 输出 | 说明 |
| ReadData1 [31:0] | 读数据输出端 |
| ReadData2 [31:0] | 读数据输出端 |

### 代码实现



## DataMemory

### 功能说明

提供读取和写入数据的功能。

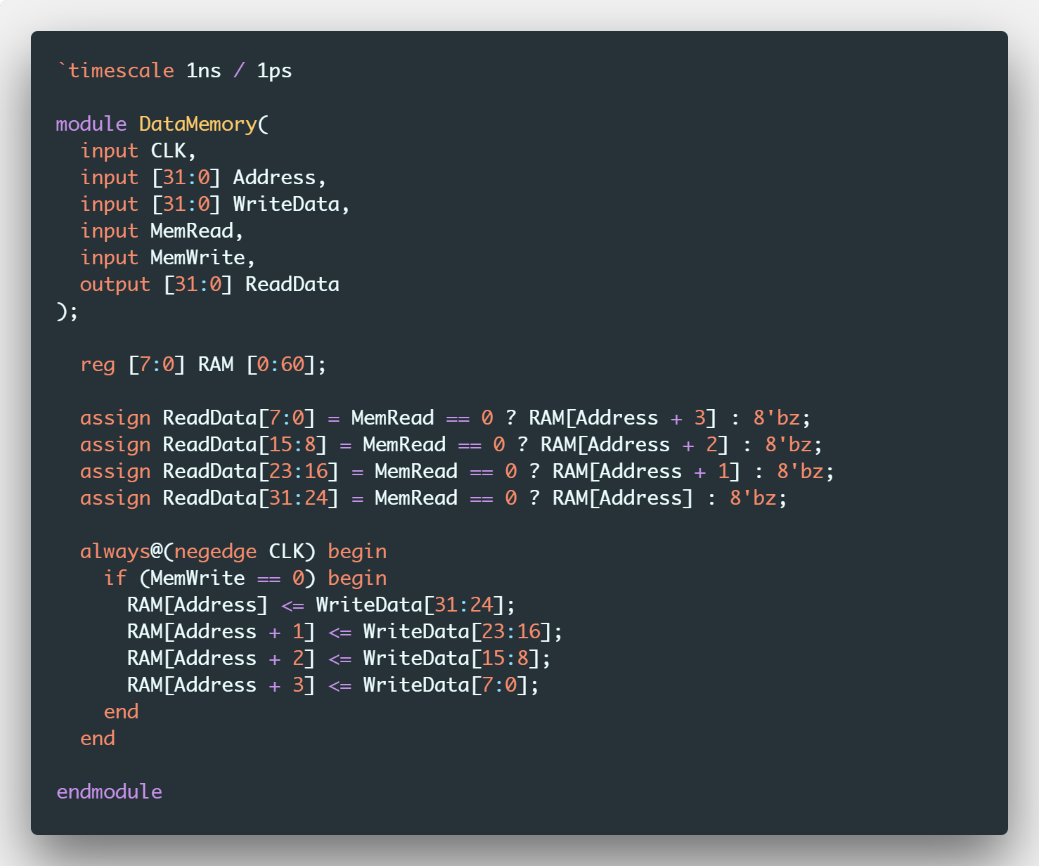
### 输入说明

|  |  |
| --- | --- |
| 输入 | 说明 |
| CLK | 时钟信号 |
| Address [31:0] | 要操作的内存地址 |
| WriteData [31:0] | 来自寄存器要写入的数据 |
| MemRead | 控制信号为0，进行读操作 |
| MemWrite | 控制信号为1，进行写操作 |

### 输出说明

|  |  |
| --- | --- |
| 输出 | 说明 |
| ReadData [31:0] | 输出读操作的结果 |

### 代码实现



## ALU

### 功能说明

ALU模块接收寄存器的数据和控制信号的输入，将运算后的结果输出。

### 输入说明

|  |  |
| --- | --- |
| 输入 | 说明 |
| ALUOp [2:0] | 控制信号，用于运算功能的选择 |
| ReadData1 [31:0] | 来自寄存器rs的数据 |
| ReadData2 [31:0] | 来自寄存器rt的数据 |

### 输出说明

|  |  |
| --- | --- |
| 输出 | 说明 |
| Zero | 当运算结果为0，输出为1，否则输出为0 |
| Sign | 运算结果最高位，0为正数，1为负数 |
| Result [31:0] | 运算结果 |

### ALU功能

|  |  |  |
| --- | --- | --- |
| ALUOp [2:0] | 功能 | 描述 |
| 000 | Y = A + B | 加 |
| 001 | Y = A – B | 减 |
| 010 | Y = A & B | 与 |
| 011 | Y = B | A | 或 |
| 100 | Y = A < B ? 1 : 0 | 无符号判断A与B |
| 101 | Y = (A < B && A[31] == B[31])  || (A[31] == 1 && B[31] == 0)  ? 1 : 0 | 有符号判断A与B |
| 110 | Y = B << A | B左移A位 |

### 代码实现



## Extend

### 功能说明

将一个16位立即数扩展到32位。

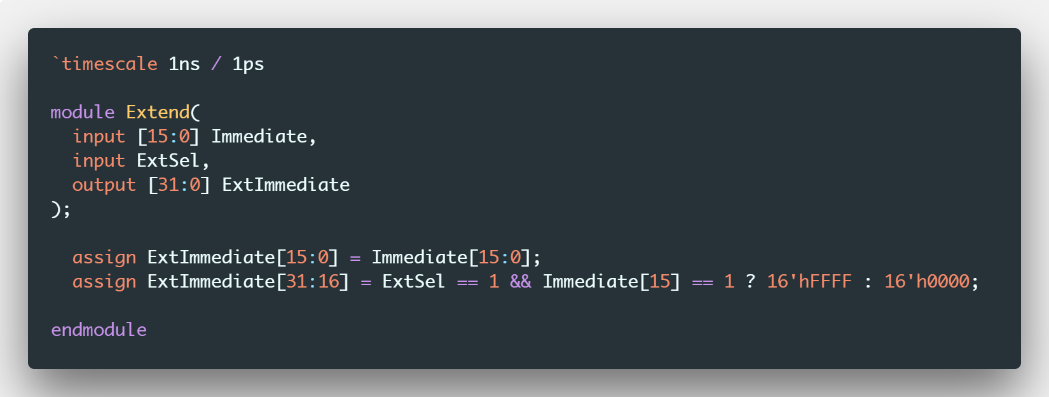
### 输入说明

|  |  |
| --- | --- |
| 输入 | 说明 |
| Immediate [15:0] | 输入的16位立即数 |
| ExtSel | 控制信号为1，进行符号扩展；为0，进行零扩展 |

### 输出说明

|  |  |
| --- | --- |
| 输入 | 说明 |
| Immediate [32:0] | 扩展后的32位立即数 |

### 代码实现

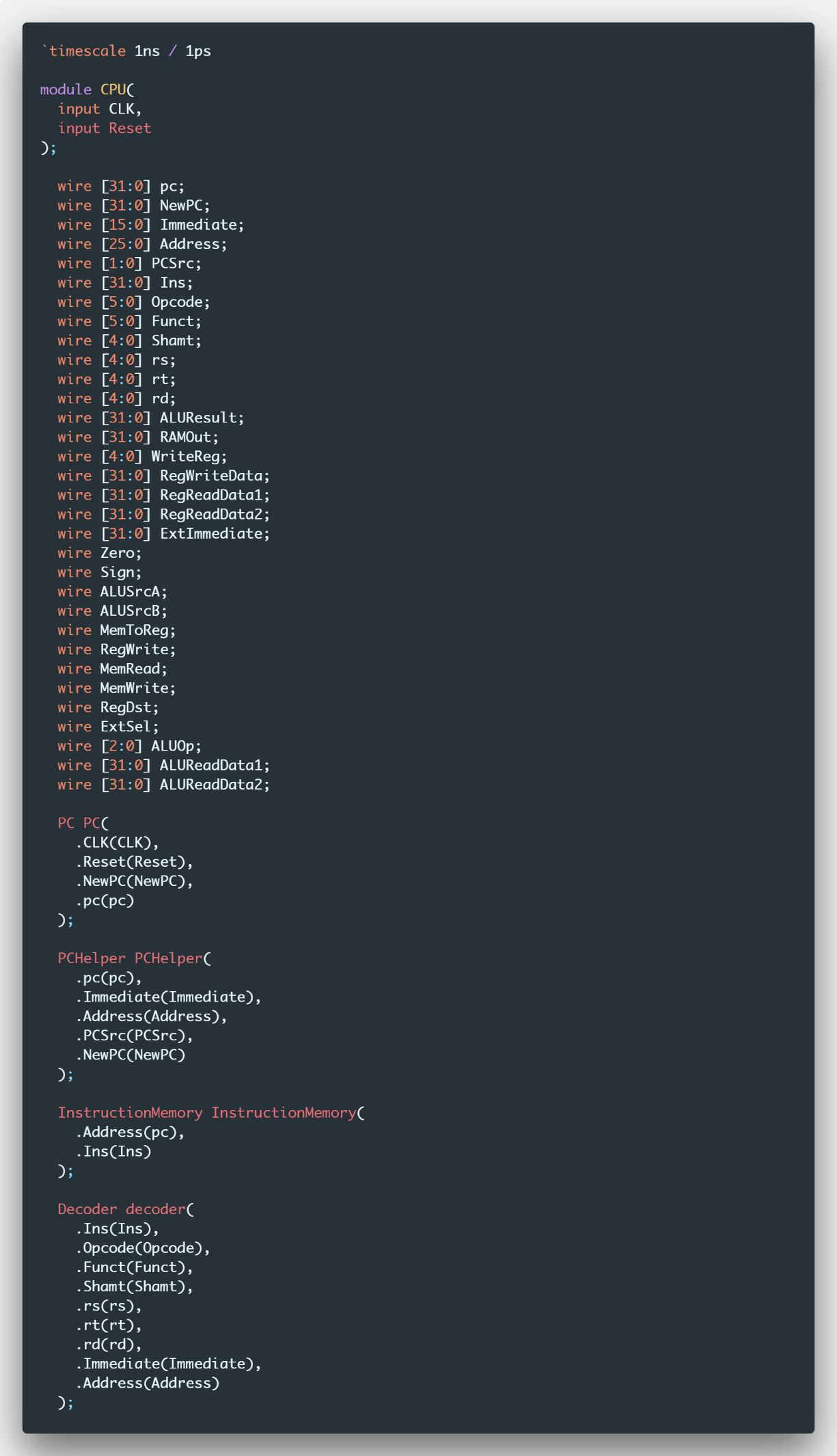


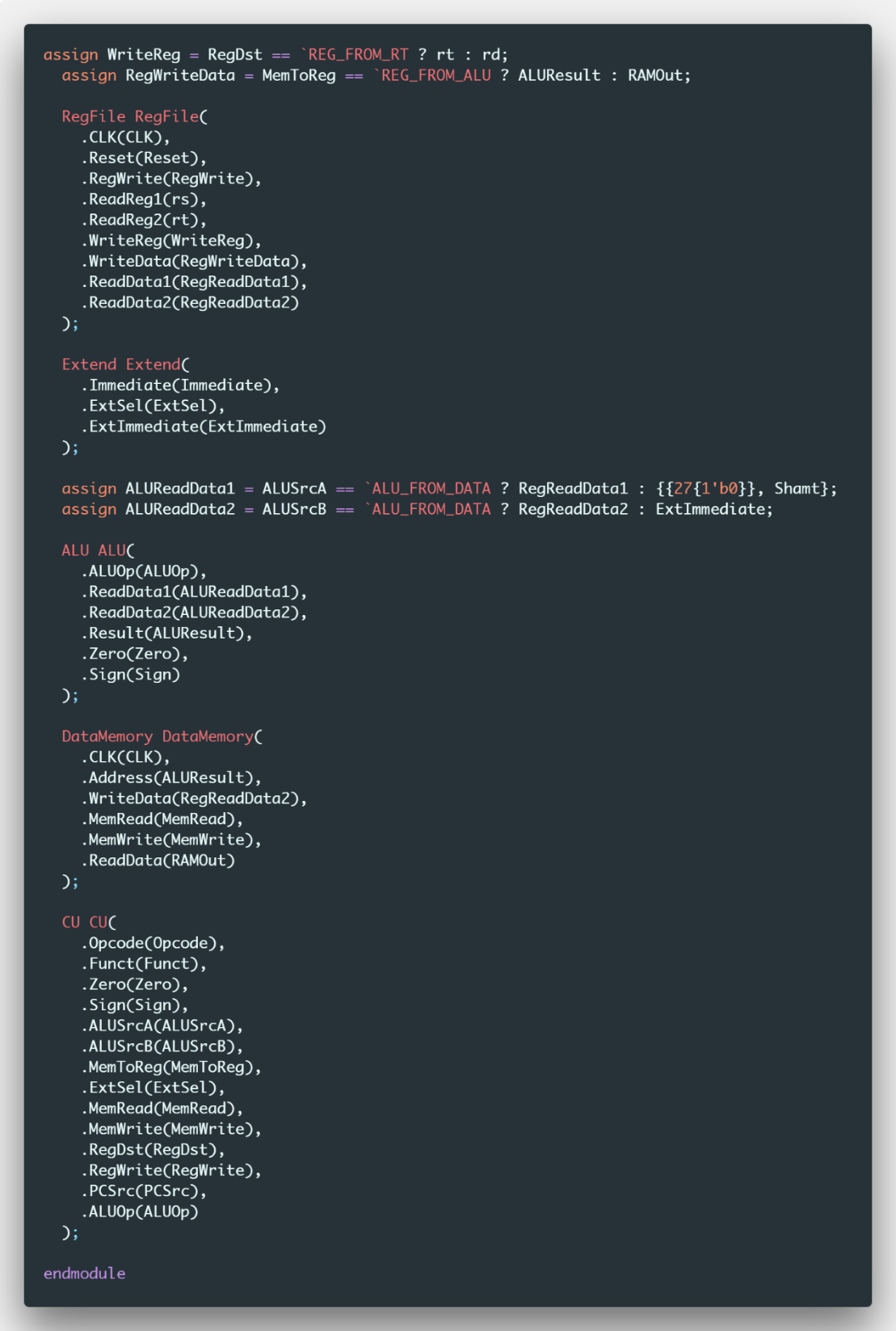
## 顶层CPU模块

### 功能说明

顶层模块是CPU的综合模块，能将各个子模块联系，从而实现CPU功能。

### 代码实现





## 仿真文件

### 功能说明

用于实例化CPU模块，给定初始的CLK和Reset信号。

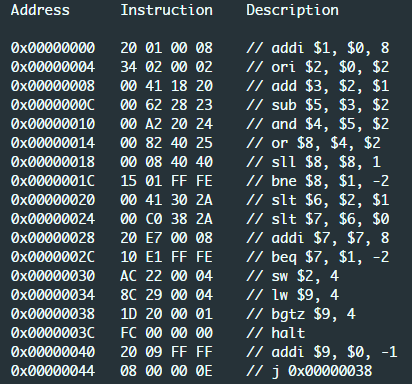
### 代码实现



# **功能测试**

## 测试文件

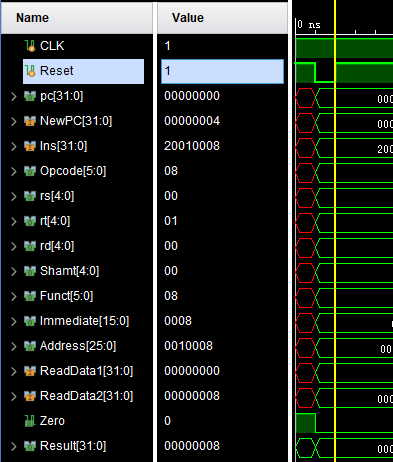
详细测试文件说明，请查看test文件夹的README.docx文件。



## 功能验证

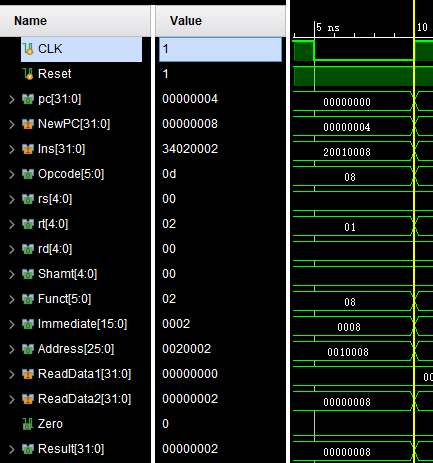
**addi $1, $0, 8**

可以看到：PC为：00000000，Instruction为：20010008，表示与立即数的加法运算，其中寄存器分别是$1，$0，立即数是8，所以ReadData1是0，ReadData2是8，ALUResult是8。



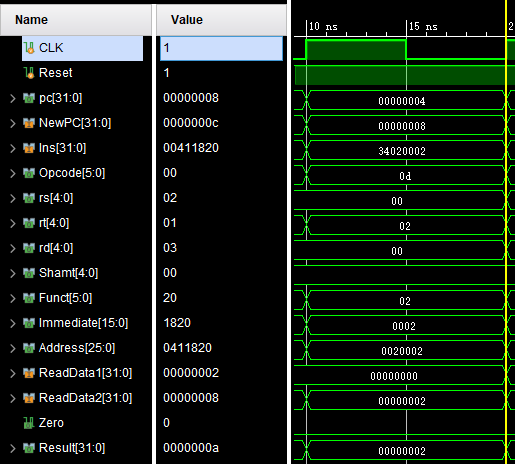
**ori $2, $0, $2**

可以看到：PC为：00000004，Instruction为：34020002，表示与立即数的或运算，其中寄存器分别是$2，$0，$2，立即数是2，所以ReadData1是0，ReadData2是2，ALUResult是2。



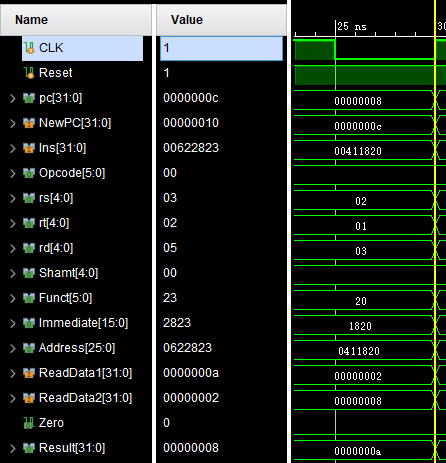
**add $3, $2, $1**

可以看到：PC为：00000008，Instruction为：00411820，表示加法运算，其中寄存器分别是$3，$2，$1，所以ReadData1是2，ReadData2是8，ALUResult是10。



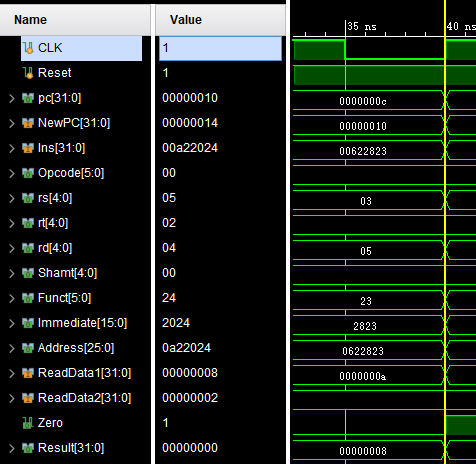
**sub $5, $3, $2**

可以看到：PC为：0000000C，Instruction为：00622823，表示减法运算，其中寄存器分别是$5，$3，$2，所以ReadData1是10，ReadData2是2，ALUResult是8。



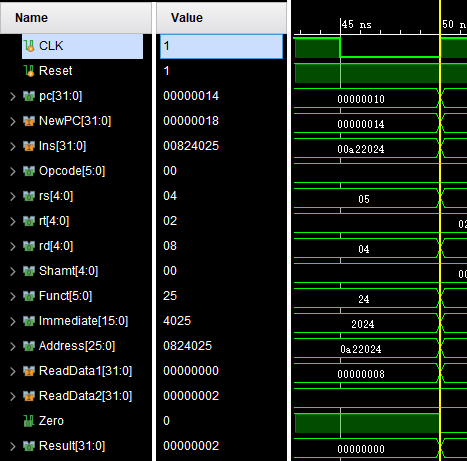
**and $4, $5, $2**

可以看到：PC为：00000010，Instruction为：00a22024，表示与运算，其中寄存器分别是$4，$5，$2，所以ReadData1是8，ReadData2是2，ALUResult是0。



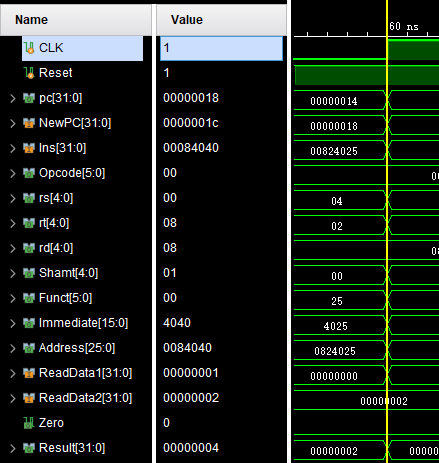
**or $8, $4, $2**

可以看到：PC为：00000014，Instruction为：00824025，表示或运算，其中寄存器分别是$8，$4，$2，所以ReadData1是0，ReadData2是2，ALUResult是2。



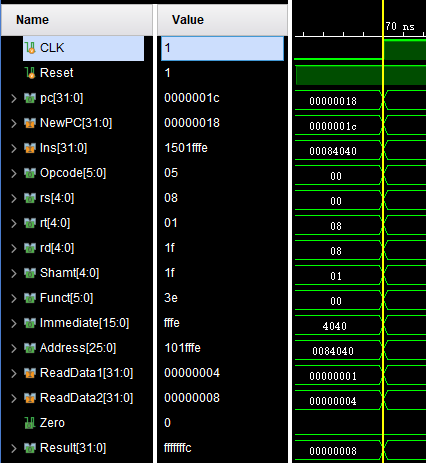
**sll $8, $8, 1**

可以看到：PC为：00000018，Instruction为：00084040，表示左移位运算，其中寄存器是$8，$8，左移位数是1\*2=2，所以ReadData1是1，ReadData2是2，ALUResult是4。



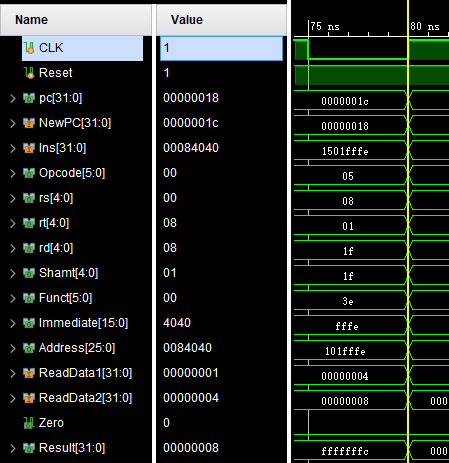
**bne $8, $1, -2**

可以看到：PC为：0000001C，Instruction为：1501FFFE，表示分支指令，由于Zero为0，故将跳转至00000018。



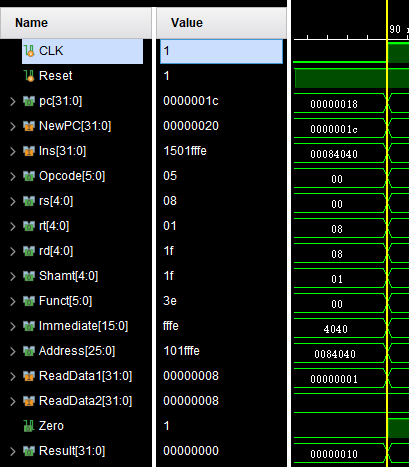
**sll $8, $8, 1**

可以看到：PC为：00000018，Instruction为：00084040，表示左移位运算，其中寄存器是$8，左移位数是1，所以ReadData1是1，ReadData2是4，ALUResult是8。



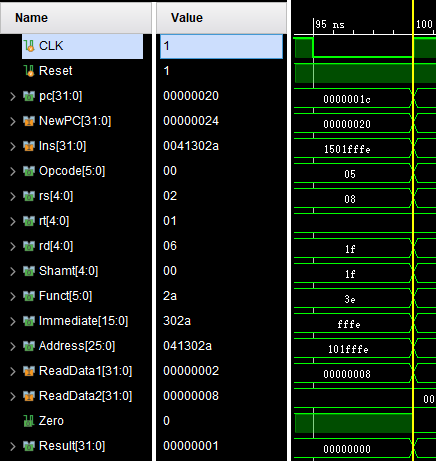
**bne $8, $1, -2**

可以看到：PC为：0000001C，Instruction为：1501FFFE，表示分支指令，由于Zero为1，故将跳转至00000020。



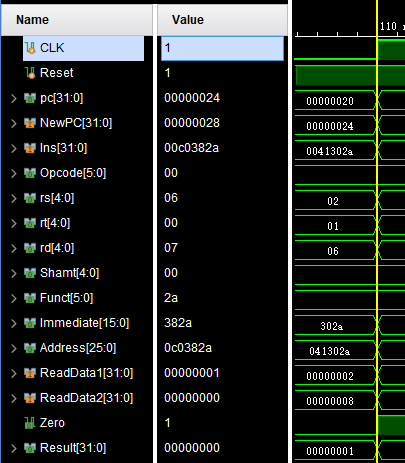
**slt $6, $2, $1**

可以看到：跳转完成后，PC为：00000020，Instruction为：0041302A，表示比较运算，其中寄存器分别是$6，$2，$1，由于rs<rt，故ALUResult为1



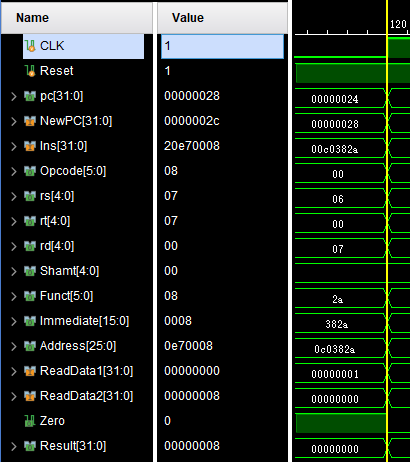
**slt $7, $6, $0**

可以看到：PC为：00000024，Instruction为：00C0382A，表示比较运算，其中寄存器分别是$7，$6，$0，由于rs>rt，故ALUResult为0



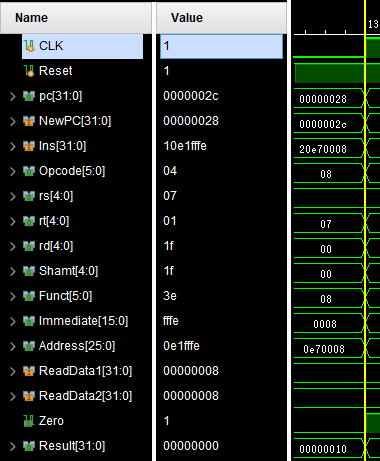
**addi $7, $7, 8**

可以看到：PC为：00000028，Instruction为：20E70008，表示与立即数的加法运算，其中寄存器分别是$7，立即数是8，所以ReadData1是0，ReadData2是8，ALUResult是8。



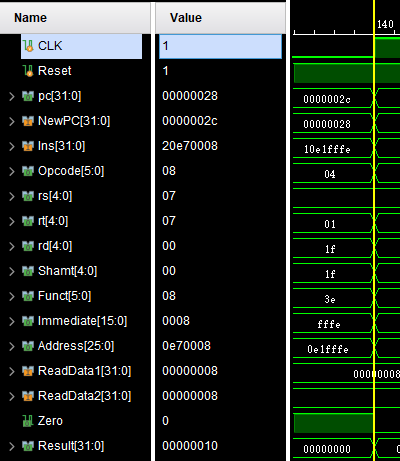
**beq $7, $1, -2**

可以看到：PC为：0000002C，Instruction为：10E1FFFE，表示分支指令，由于Zero为1，故将跳转至00000028。



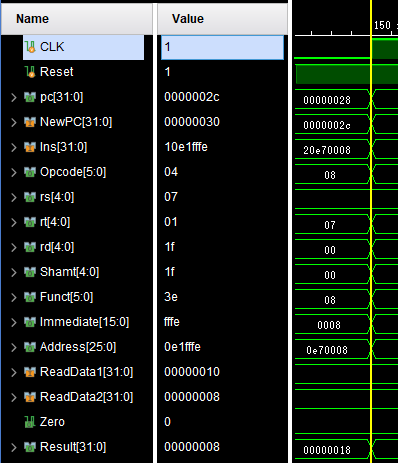
**addi $7, $7, 8**

可以看到：PC为：00000028，Instruction为：20E70008，表示与立即数的加法运算，其中寄存器分别是$7，立即数是8，所以ReadData1是8，ReadData2是8，ALUResult是16。



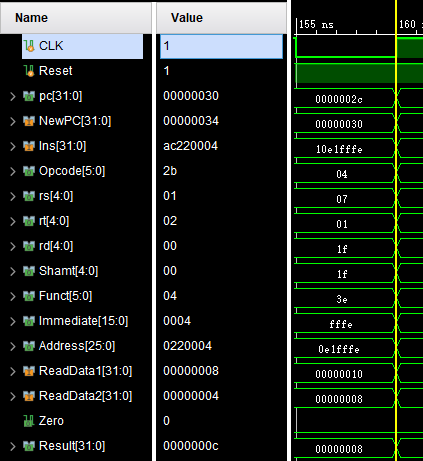
**beq $7, $1, -2**

可以看到：PC为：0000002C，Instruction为：10E1FFFE，表示分支指令，由于Zero为0，故将跳转至00000030。



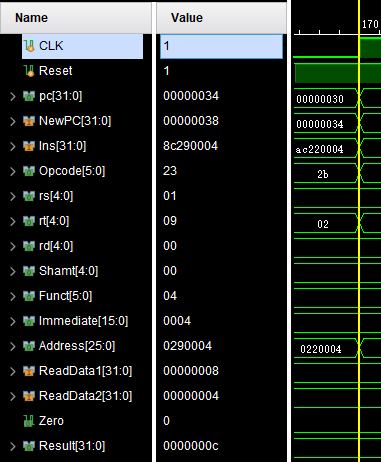
**sw $2, 4($1)**

可以看到：PC为：00000030，Instruction为：AC220004，表示写入寄存器指令，将$2内容写入至$5。



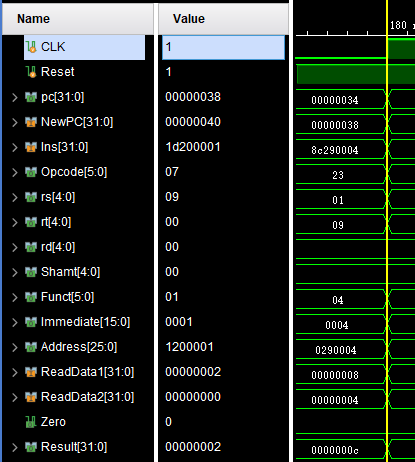
**lw $9, 4($1)**

可以看到：PC为：00000034，Instruction为：8C290004，表示读取寄存器指令，将$5的内容读取至$9里面。



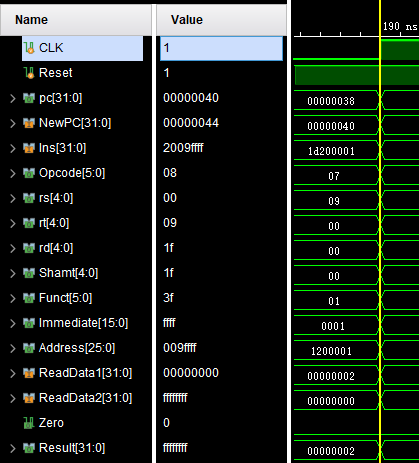
**bgtz $9, 1**

可以看到：PC为：00000038，Instruction为：1D200001，表示分支指令，由于 $9>1，故将跳转至00000040。



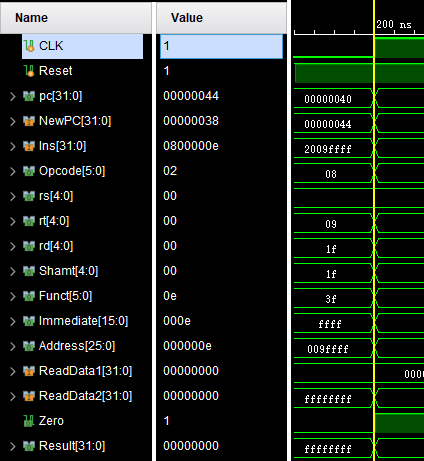
**addi $9, $0, -1**

可以看到：PC为：00000040，Instruction为：2009FFFF，表示与立即数的加法运算，其中寄存器分别是$9，$0，立即数是-1，所以ReadData1是0，ReadData2是-1，ALUResult是-1。



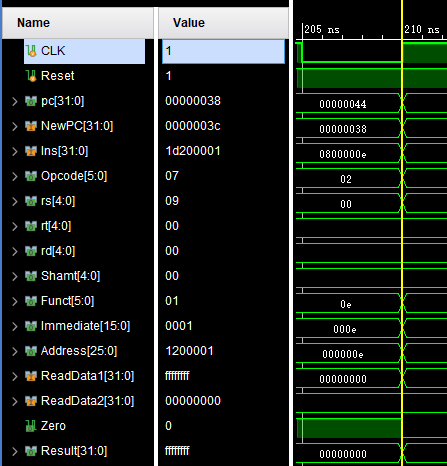
**j 0x00000038**

可以看到：PC为：00000044，Instruction为：0800000E，表示无条件跳转指令，故将跳转至00000038。



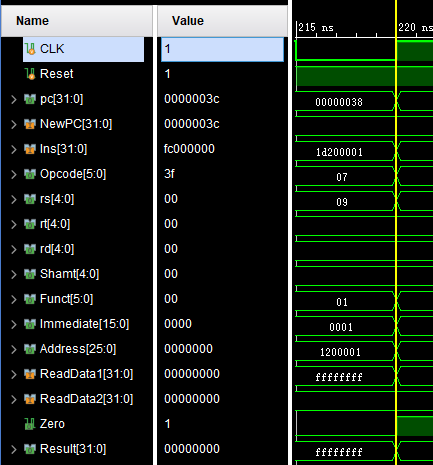
**bgtz $9, 4**

可以看到：PC为：00000038，Instruction为：1D200001，表示分支指令，由于$9<4，故将跳转至0000003C。



**halt**

可以看到：PC为：0000003C，Instruction为：FC000000，表示停机指令，故程序执行结束。



## 寄存器内存检查

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 步骤 | $0 | $1 | $2 | $3 | $4 | $5 | $6 | $7 | $8 | $9 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 8 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 8 | 2 | 10 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 8 | 2 | 10 | 0 | 8 | 0 | 0 | 0 | 0 |
| 6 | 0 | 8 | 2 | 10 | 0 | 8 | 0 | 0 | 0 | 0 |
| 7 | 0 | 8 | 2 | 10 | 0 | 8 | 0 | 0 | 2 | 0 |
| 8 | 0 | 8 | 2 | 10 | 0 | 8 | 0 | 0 | 4 | 0 |
| 9 | 0 | 8 | 2 | 10 | 0 | 8 | 0 | 0 | 4 | 0 |
| 10 | 0 | 8 | 2 | 10 | 0 | 8 | 0 | 0 | 8 | 0 |
| 11 | 0 | 8 | 2 | 10 | 0 | 8 | 0 | 0 | 8 | 0 |
| 12 | 0 | 8 | 2 | 10 | 0 | 8 | 1 | 0 | 8 | 0 |
| 13 | 0 | 8 | 2 | 10 | 0 | 8 | 1 | 0 | 8 | 0 |
| 14 | 0 | 8 | 2 | 10 | 0 | 8 | 1 | 8 | 8 | 0 |
| 15 | 0 | 8 | 2 | 10 | 0 | 8 | 1 | 8 | 8 | 0 |
| 16 | 0 | 8 | 2 | 10 | 0 | 8 | 1 | 16 | 8 | 0 |
| 17 | 0 | 8 | 2 | 10 | 0 | 8 | 1 | 16 | 8 | 0 |
| 18 | 0 | 8 | 2 | 10 | 0 | 8 | 1 | 16 | 8 | 0 |
| 19 | 0 | 8 | 2 | 10 | 0 | 8 | 1 | 16 | 8 | 2 |
| 20 | 0 | 8 | 2 | 10 | 0 | 8 | 1 | 16 | 8 | 2 |
| 21 | 0 | 8 | 2 | 10 | 0 | 8 | 1 | 16 | 8 | -1 |
| 22 | 0 | 8 | 2 | 10 | 0 | 8 | 1 | 16 | 8 | -1 |
| 23 | 0 | 8 | 2 | 10 | 0 | 8 | 1 | 16 | 8 | -1 |

经检验，上述仿真寄存器结果正确。

# **实验心得**

总体来说，本次设计单周期CPU的拓展项目带给我的收获是巨大的，我从中学到了很多东西。

还记得数字电子技术课程刚开课时，老师就在课上介绍了优秀的师兄师姐们做的课外拓展项目，鼓励我们也去尝试做类似的拓展。最初看到师兄们的单周期CPU设计的实验报告时，感觉自己离它非常遥远，觉得单周期CPU设计是一件难以完成的任务。但为了能让自己得到充分的实践锻炼，我选择开发单周期CPU的任务。

虽然在大一上学期《软件工程导论》课上，对计算机组成和CPU相关的知识有所了解，但我知道，课上传授的知识只是皮毛，是不足以为开发CPU提供技术基础的，在学长的推荐下，我开始阅读《深入理解计算机系统》和《计算机组成原理》。

《深入理解计算机系统》从C语言开始讲起，再讲到汇编，再到处理器，还引申了很多底层的知识点，整体来说，讲解的顺序是符合学生的逻辑认知习惯的。《计算机组成原理》是我们大二学期的教材，自我感觉表达比前者更晦涩，但知识更具深度。

大概花了半学期，我把上面提及到的两本书大致看完了，觉得对CPU的工作原理的掌握足够用来设计开发单周期CPU。但有了知识仍不够，我们需要Verilog硬件描述语言，将我们的想法变成程序，甚至是硬件。在课堂上，我们对Verilog的了解并不是很深，因为数字电子技术实验作业大多都是通过 Block Design 完成的，教师对Verilog不作要求。所以在这样的背景下，我选择了自学Verilog，争取快速地将其掌握。在GitHub上，我找到了关于Verilog的入门教程，大概花了两天的时间，我对Verilog有了深入的理解，在教程要求下，通过很多范例的引导，我能够比较熟练地运用Verilog。

在完成项目的过程中，我遇到了较多的困难，印象比较深刻的就是如何规划项目架构。因为看似有了概念、知识和工具，但对一个功能完善的系统，我不知道从哪里开始切入。最后在网上几篇教程的指引下，我有了具体设计思路——面向模块的思想，即只需暴露各个子模块的接口，然后在顶层模块中将各接口连接。

在Vivado进行仿真时，曾出现一直无法仿真的问题，刚开始以为是代码出了问题，就从头开始排查，但始终找不到错误的原因。后来，在舍友（欧子菁）的帮助下，他告诉我是仿真文件的路径问题造成的。按照他的回答，我将我的仿真文件路径从相对路径调整为绝对路径，结果真的能如期仿真，还是很感谢这位舍友的。

由于开发外包项目的原因，这次项目的战线拉得有些长，完成代码和提笔写报告隔了将近一个月的时间。又因为临近期末，复习压力大，所以该实验报告略显仓促，还有很多改进的空间。

最后，感谢老师为我们提供宝贵的机会和鼓励，让我体会到了系统设计的重要性：系统设计对个人知识网络的建立有极大的帮助。虽然我是软件工程的学生，未来可能不做与硬件直接相关的工作，但如果能从硬件层次理解计算机运行的过程和原理，是对软件开发有极大的帮助的。我们学到的数字电子技术知识属于硬件知识，它不能是孤立的，而应该被相互联系的。系统设计，就提供这样的过程，让我们把知识能够联系起来。在日后的学习中，我一定会挤出时间去寻找并完成类似的系统设计，努力成为优秀的软件工程人才。