**CPU 设计文档**

- CPU模块设计
  - NPC
    - 输入与输出
      - 输入
      - 输出
    - IM(指令存储器)
      - 输入与输出
        - 输入
        - 输出
      - 实现
      - 注意事项
    - DM(数据存储器)
      - 输入与输出
        - 输入
        - 输出
      - 实现
    - ALU(算术逻辑单元)
      - 输入与输出
        - 输入
        - 输出
      - ALU控制码
      - 实现
    - Controller(控制器)
      - 输入与输出
        - 输入
        - 输出
      - “与”逻辑部分
      - “或”逻辑部分
      - 各控制信号功能
    - Ex(位扩展器))
      - 输入与输出
        - 输入
        - 输出
      - 位扩展方式
      - 实现
    - GRF(通用寄存器组)
      - 输入与输出
        - 输入
        - 输出
      - 实现
      - 注意事项
- 思考题
  - 1
  - 2
  - 3
  - 4
  - 5
    - ori指令
    - lui指令
    - add 指令
    - sub指令
    - sw指令

# CPU 设计文档

## CPU模块设计

### NPC

#### 输入与输出

##### 输入

- `PC[31:0]`
- `imm26[25:0]`
- `EXT_IMM16[31:0]`
- `GPR[rs][31:0]`
- `nPC_sel[1:0]`
- `isZero`

##### 输出

- `NPC[31:0]`

### IM(指令存储器)

#### 输入与输出

##### 输入

- `PC[31:0]`

##### 输出

- 指令`Instruction[31:0]`

#### 实现

- 这里使用到的存储器是ROM存储器，ROM是只读存储器，具有非易失性

#### 注意事项

- 实际地址宽度为5位，所以ROM的Address Bit Width应设置为5
- 向ROM中读指令时，文件头应为：

```
v2.0 raw
```

### DM(数据存储器)

#### 输入与输出

##### 输入

- 写地址`Addr[31:0]`
- 写入数据`WD[31:0]`
- 写使能信号`MemWrite`
- `MemtoReg`

- 异步复位信号`reset`

#### 输出

- 数据输出`RD[31:0]`

### 实现

- 这里使用的存储器是RAM

## ALU(算术逻辑单元)

### 输入与输出

#### 输入

- 运算数`A[31:0]`
- 运算数`B[31:0]`
- ALU控制码`ALUop[3:0]`

#### 输出

- 运算结果`Result[31:0]`
- 零判断`isZero`

### ALU控制码

根据ALU控制码`ALUop`来确定进行何种运算：

| ALUop | Function |
| --- | --- |
| 0000 | ADD |
| 0001 | SUB |
| 0010 | OR |
| 0011 | AND |
| 0100 | XOR |

### 实现

- 这里`ALUop`用了4位，以便于后续添加各种类型的运算
- 添加零判断输出`iszero`可以在`beq`等指令中快速判断结果

## Controller(控制器)

### 输入与输出

#### 输入

- `opcode[5:0]`
- `funct[5:0]`

**输出**

- `RegDst`
- `ALUSrc`
- `MemtoReg`
- `RegWrite`
- `nPC_sel[1:0]`
- `ALUctr`

## “与”逻辑部分

根据 `opcode` 与 `funct` 将输入的机器码识别为相应的指令：

| Instruction | opcode | funct |
|---|---|---|
| add | 000000 | 100000 |
| sub | 000000 | 100010 |
| ori | 001101 | n/a |
| lw | 100011 | n/a |
| sw | 101011 | n/a |
| beq | 000100 | n/a |
| lui | 001111 | n/a |

## “或”逻辑部分

根据指令生成相应的控制信号：

| | add | sub | xor | ori | lw | sw | beq | lui | j | jal | jr |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **RegDst** | 1 | 1 | 1 | 0 | 0 | X | X | 0 | X | X | X |
| **ALUSrc** | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | X | X | X |
| **MemtoReg** | 0 | 0 | 0 | 0 | 1 | X | X | 0 | X | X | X |
| **RegWrite** | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 |
| **MemWrite** | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| **nPC_sel** | 00 | 00 | 00 | 00 | 00 | 00 | 01 | 00 | 10 | 10 | 11 |
| **ALUctr** | ADD | SUB | XOR | OR | ADD | ADD | SUB | ADD | X | X | X |
| **ExtOp** | None | None | None | Zero | Sign | Sign | Sign | High | None | None | None |

## 各控制信号功能

- RegDst：由于R型指令和I型指令所要写入的寄存器对应指令中的位数不同，需要该信号来控制写入哪个寄存器。R型指令置1（写入rd），I型指令置0（写入rt）。
- ALUSrc：选择ALU的操作数B是从寄存器中读取的还是立即数，R型指令（从寄存器中读取数）置0，I型指令（立即数）置1。
- MemtoReg：选择写入寄存器堆的值为运算器运算结果（置0）还是存储器中取出的数（置1）的选择信号。
- RegWrite：寄存器(GRF)写使能。

- MemWrite：数据存储器(DM)写使能。
- nPC_sel：跳转信号，beq指令置01，j、jal指令置10，jr指令置11。
- ALUctr：选择需要何种运算操作（见ALU部分）。
- ExtOp：选择需要何种位扩展操作（见Ext部分）。

# Ex(位扩展器))

## 输入与输出

### 输入

- 16位立即数`Imm[15:0]`
- 位扩展方式`ExtOp[1:0]`

### 输出

- 位扩展结果`Ext[31:0]`

## 位扩展方式

| ExtOp | Function |
| --- | --- |
| 00 | Zero |
| 01 | Sign |
| 10 | High |
| 11 | ? |

## 实现

- 这里ExtOp用两位表示，并留出一种了情况留待扩展

# GRF(通用寄存器组)

## 输入与输出

### 输入

- 读地址1`A1[4:0]`
- 读地址2`A2[4:0]`
- 写地址`A3[4:0]`
- 数据输入`WD[31:0]`
- 异步复位信号`reset`
- 写使能信号`we`

### 输出

- 输出A1指定的寄存器数据`RD1[31:0]`
- 输出A2指定的寄存器数据`RD2[31:0]`

### 实现

- 零寄存器的值始终为0
- 这里直接使用了在 *P0_L0_GRF* 中搭建的电路

### 注意事项

- 将RAM的"Data Interface"设为"Seperate load and store ports"

# 思考题

## 1

**上面我们介绍了通过 FSM 理解单周期 CPU 的基本方法。请大家指出单周期 CPU 所用到的模块中，哪些发挥状态存储功能，哪些发挥状态转移功能。**

答：状态存储功能： PC 、 GRF ； 状态转移功能： NPC 、 DM 。

## 2

**现在我们的模块中 IM 使用 ROM， DM 使用 RAM， GRF 使用 Register，这种做法合理吗？ 请给出分析，若有改进意见也请一并给出。**

答：ROM是只读存储器，速度慢于RAM，但它的内容一经写入不易被更改，具有非易失性，使用于IM中可以很好地保存指令；RAM是随机存取存储器，它的速度很快，可以随时对DM中相应的地址进行读或写，用于DM中方便又快捷；Register的使用效率更高，高于RAM，将它用于GRF可以更加快捷地对32个寄存器进行读或写。

## 3

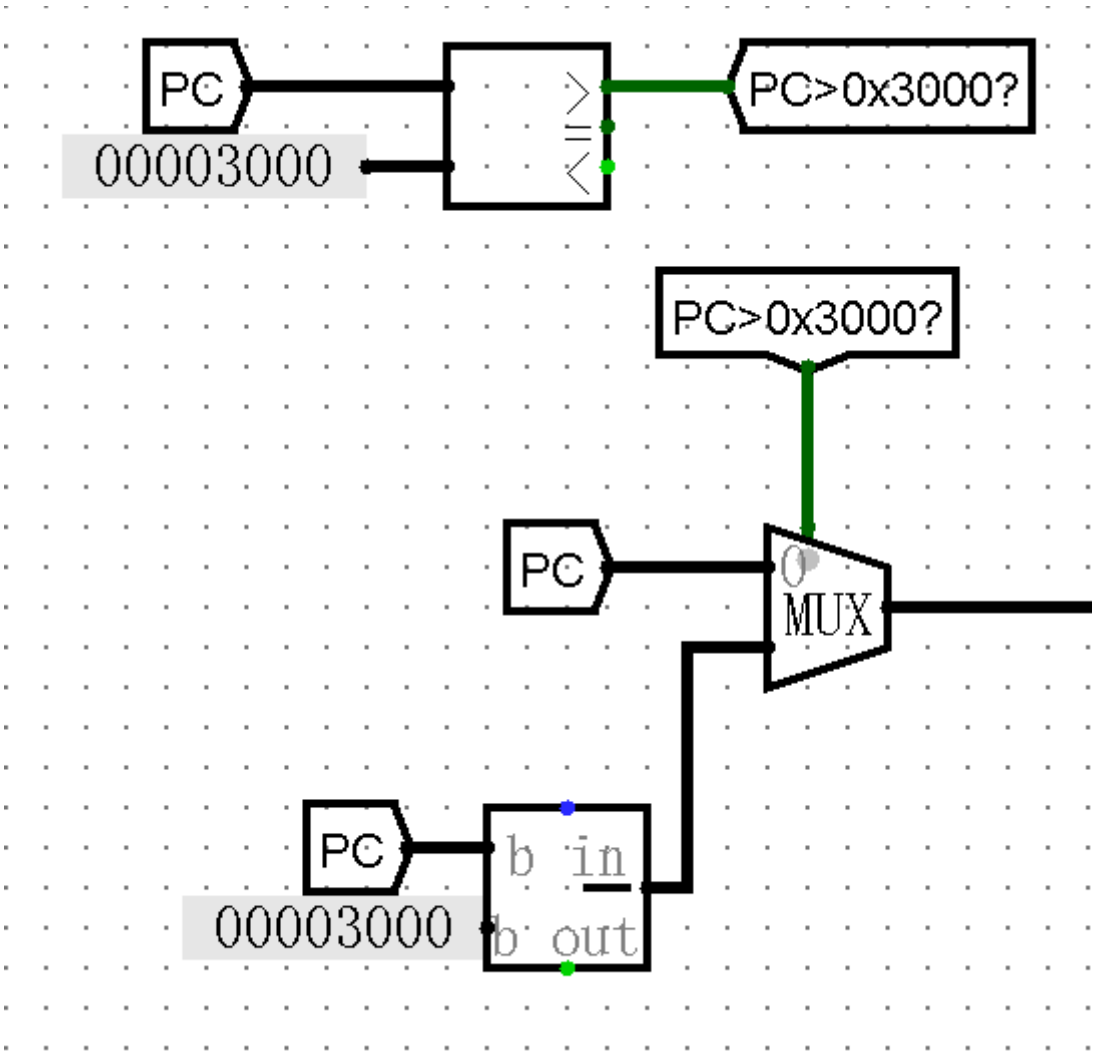
**事实上，实现 `nop` 空指令，我们并不需要将它加入控制信号真值表，为什么？**

答：对于 `nop` 空指令，其对应的控制型号RegWrite、MemWrite、nPC_sel均为0，且不关心RegDst、ALUSrc、MemtoReg、ALUctr、ExtOp这些控制信号，所以无需加入控制信号真值表。

## 4

**上文提到，MARS 不能导出 PC 与 DM 起始地址均为 0 的机器码。实际上，可以避免手工修改的麻烦。请查阅相关资料进行了解，并阐释为了解决这个问题，你最终采用的方法。**

在 MARS 中，默认 PC 的地址从 0x00003000 开始，所以可以在 Logisim 中的 IM 模块中判断输入是否大于 0x00003000 ，若大于，则减去 0x00003000 。




## 5

**阅读 Pre 的 “MIPS 指令集及汇编语言” 一节中给出的测试样例，评价其强度（可从各个指令的覆盖情况，单一指令各种行为的覆盖情况等方面分析），并指出具体的不足之处。**

答：从不同指令来分析：

### ori指令

```
ori $a0, $0, 123
ori $a1, $a0, 456
# ori $a2, $0, -456
```

包含了非零数与 0 进行或运算，非零数之间进行或运算的情况，覆盖情况很好。~~（应该没有必要测试 0 与 0 进行或运算的情况）~~

### lui指令

```
lui $a2, 123            # 符号位为 0
lui $a3, 0xffff         # 符号位为 1
ori $a3, $a3, 0xffff    # $a3 = -1
```

未包含高位非 0 的情况。

### add 指令

```
add $s0, $a0, $a2      # 正正
add $s1, $a0, $a3      # 正负
add $s2, $a3, $a3      # 负负
```

包含了正数加正数，正数加负数，负数加负数三种情况。可以在此基础上测试一下数据范围边界的一些数之间的 add 。

### sub指令

给出的测试样例未对 sub 指令进行测试。

### sw指令

```
ori $t0, $0, 0x0000
sw $a0, 0($t0)
sw $a1, 4($t0)
sw $a2, 8($t0)
sw $a3, 12($t0)
sw $s0, 16($t0)
sw $s1, 20($t0)
sw $s2, 24($t0)
```

### lw指令

```
lw $a0, 0($t0)
lw $a1, 12($t0)
sw $a0, 28($t0)
sw $a1, 32($t0)
```

应该测试一下向 $0 写入数据的情况。

### beq指令

```
ori $a0, $0, 1
ori $a1, $0, 2
ori $a2, $0, 1

beq $a0, $a1, loop1     # 不相等
beq $a0, $a2, loop2     # 相等

loop1:sw $a0, 36($t0)
loop2:sw $a1, 40($t0)
```

## 测试方案

## ori指令

```
ori $a0, 123
ori $a1, 456
```

## lui指令

```
lui $a2, 234
lui $a2, 123            # 符号位为 0
```

## add指令

```
add $t0, $a0, $a1       # 测试正数+正数
lui $a3, 0xffff         # 符号位为 1
ori $a3, $a3, 0xffff    # $a3 = -1(先构造一个负数)
add $t1, $a0, $a3       # 测试正数+负数
add $t2, $a3, $a3       # 测试负数+负数
```

## sub指令

```
sub $t0, $a0, $a1       # 测试正数-正数(结果为负数)
sub $t1, $a3, $a0       # 测试负数-正数
sub $t2, $a3, $t0       # 测试负数-负数
sub $t3, $a3, $a1       # 测试负数-正数
```

## xor指令

```
xor $t4, $a0, $a1
xor $t5, $a0, $a2
```

## sw指令

```
ori $t0, $0, 0x0000
sw $a0, 0($t0)
sw $a1, 4($t0)
sw $a2, 8($t0)
sw $a3, 12($t0)
sw $t1, 20($t0)
sw $t2, 28($t0)
sw $t3, 32($t0)
```

## lw指令

```
lw $0, 0($t0)
lw $a1, 0($t0)
lw $a0, 8($t0)
lw $a0, 12($t0)
sw $a0, 24($t0)
sw $a1, 28($t0)
```

## beq指令

```
ori $a0, $0, 1
ori $a1, $0, 2
ori $a2, $0, 1

beq $a0, $a1, beq1
beq $a0, $a2, beq2

beq1:
sw $a1, 36($t0)
beq2:
sw $a1, 40($t0)
```

## 完整测试样例

```
ori $a0, 123
ori $a1, 456

add $t0, $a0, $a1
lui $a3, 0xffff
ori $a3, $a3, 0xffff
add $t1, $a0, $a3
add $t2, $a3, $a3

sub $t0, $a0, $a1
sub $t1, $a3, $a0
sub $t2, $a3, $t0
sub $t3, $a3, $a1

xor $t4, $a0, $a1
xor $t5, $a0, $a2

ori $t0, $0, 0x0000
sw $a0, 0($t0)
sw $a1, 4($t0)
sw $a2, 8($t0)
sw $a3, 12($t0)
sw $t1, 20($t0)
sw $t2, 28($t0)
sw $t3, 32($t0)

lw $0, 0($t0)
lw $a1, 0($t0)
lw $a0, 8($t0)
lw $a0, 12($t0)
sw $a0, 24($t0)
sw $a1, 28($t0)

ori $a0, $0, 1
ori $a1, $0, 2
ori $a2, $0, 1

beq $a0, $a1, beq1
```

```
beq $a0, $a2, beq2

beq1:
sw $a1, 36($t0)
beq2:
sw $a1, 40($t0)
```