**同济大学计算机系**

**计算机组成原理实验报告**

****

**学 号 1951042**

**姓 名 王远洋**

**专 业 信息安全**

**授课老师 张冬冬**

目录

[一、 实验介绍 4](#_Toc76127753)

[二、 实验目标 4](#_Toc76127754)

[三、 实验原理 4](#_Toc76127755)

[1. 单周期数据通路设计 4](#_Toc76127756)

[2. 单指令数据通路分析 4](#_Toc76127757)

[2.1 Add 类型指令 4](#_Toc76127758)

[2.2 Slt 类型指令 5](#_Toc76127759)

[2.3 Sll 类型指令 5](#_Toc76127760)

[2.4 Sllv 类型指令 6](#_Toc76127761)

[2.5 Jr 类型指令 6](#_Toc76127762)

[2.6 Addi 类型指令 7](#_Toc76127763)

[2.7 lw 指令 7](#_Toc76127764)

[2.8 sw 指令 8](#_Toc76127765)

[2.9 Beq 和 Bne 指令 8](#_Toc76127766)

[2.10 Slti 和 Sltiu 指令 8](#_Toc76127767)

[2.11 Lui 指令 9](#_Toc76127768)

[2.12 J 指令 9](#_Toc76127769)

[2.13 Jal 指令 10](#_Toc76127770)

[2.14 l 类指令 10](#_Toc76127771)

[2.15 s 类指令 11](#_Toc76127772)

[2.16 读寄存器的指令 11](#_Toc76127773)

[2.17 写寄存器的指令 12](#_Toc76127774)

[2.18 跳转指令 Jarl 13](#_Toc76127775)

[2.20 乘除法指令 14](#_Toc76127776)

[2.21 异常处理指令 14](#_Toc76127777)

[2.22 clz 指令 14](#_Toc76127778)

[2.23 bgez 指令 15](#_Toc76127779)

[3. CPU 整体信号以及通路 15](#_Toc76127780)

[3.1 54 指令控制部件表 15](#_Toc76127781)

[3.2 CPU 结构通路图 16](#_Toc76127782)

[3.3 CPU 指令控制信号 16](#_Toc76127783)

[四、 模块设计 17](#_Toc76127784)

[1. 顶层模块设计 17](#_Toc76127785)

[2. CPU 模块 17](#_Toc76127786)

[3. IMEM 模块 18](#_Toc76127787)

[4. DMEM 模块 18](#_Toc76127788)

[5. Regfile 模块 18](#_Toc76127789)

[6. ALU 模块 18](#_Toc76127790)

[7. PCReg 模块 19](#_Toc76127791)

[8. Controller 模块 19](#_Toc76127792)

[9. 其它模块 20](#_Toc76127793)

[五、 测试模块设计 20](#_Toc76127794)

[六、 实验结果 21](#_Toc76127795)

[1. 前仿真测试 21](#_Toc76127796)

[1.1仿真图像 21](#_Toc76127797)

[2. 后仿真测试 21](#_Toc76127798)

# 一、 实验介绍

在本次实验中，我们将使用 Verilog 语言实现 54 条 MIPS 指令的 CPU 的设计和仿真。

# 二、 实验目标

* 深入了解 CPU 的原理
* 画出实现 54 条指令的 CPU 的通路图
* 学习使用 Verilog 语言设计实现 54 条指令的 CPU

# 三、 实验原理

## 1. 单周期数据通路设计

实现 CPU 有单周期和多周期两种方式，为了在 31 条 CPU 的基础之上进行，所以我们选择实现 54 条指令的单周期 CPU，按照单周期数据通路设计的一般性方法，如下所示：

* 阅读每条指令，对每条指令所需执行的功能与过程都有充分的了解
* 确定每条指令在执行过程中所用到的部件
* 使用表格列出指令所用部件，并在表格中填入每个部件的数据输入来源
* 根据表格所涉及部件和部件的数据输入来源，画出整个数据通路

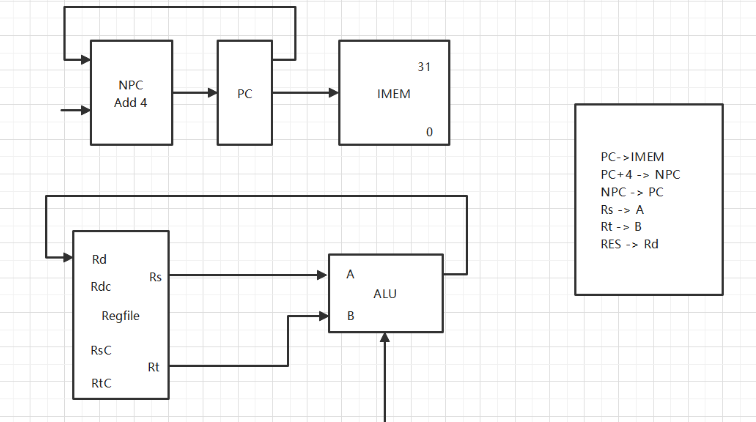
根据方法，下面分析每条指令所使用的部件以及数据通路。

## 2. 单指令数据通路分析

指令命令总共被分为 3 大类，分别是 R，I，J，其中 R 指令和寄存器有很大的关联，两个操作数都是寄存器； I 指令就是其中有立即数； J 指令甚至没有那么多的操作数，跳转使用，除此之外，54 条 CPU 还增加了一些指令，我们在后面进行分析，具体分析如下：

### 2.1 Add 类型指令

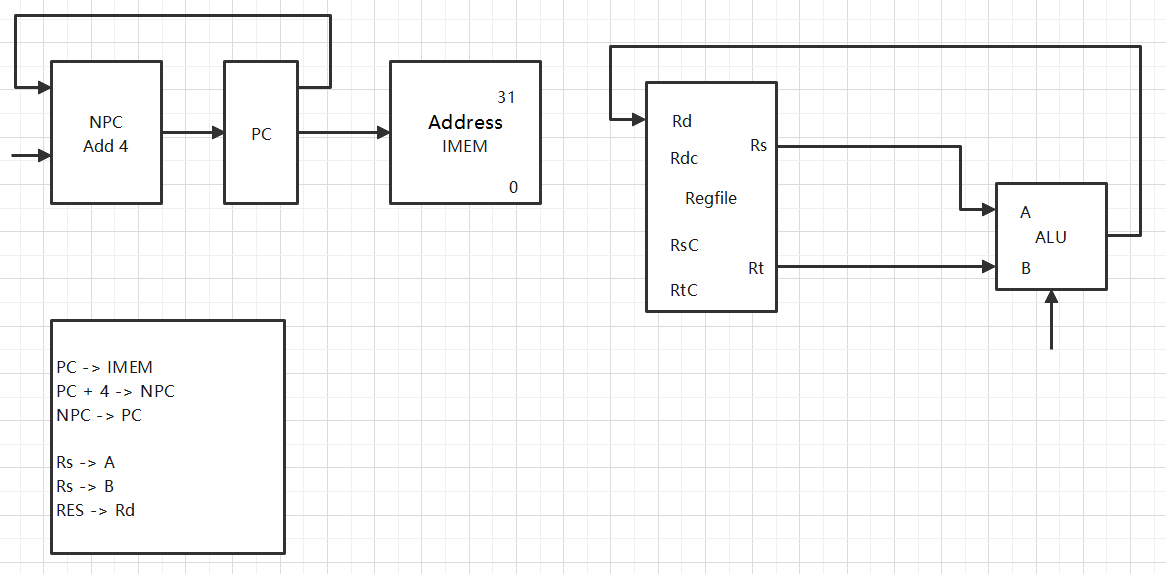
Add 类型的指令包括add/addu/sub/subu/and/or/xor/nor 共 8 条指令，这些指令只有运算器控制信号不同，除此之外完全一致，因此在同一图中表示，如下：



其中使用的器件为：PC 寄存器、指令存储器、NPC、寄存器文件以及 ALU，注意在使用过程中 ALU 传入的信号的不同。

### 2.2 Slt 类型指令

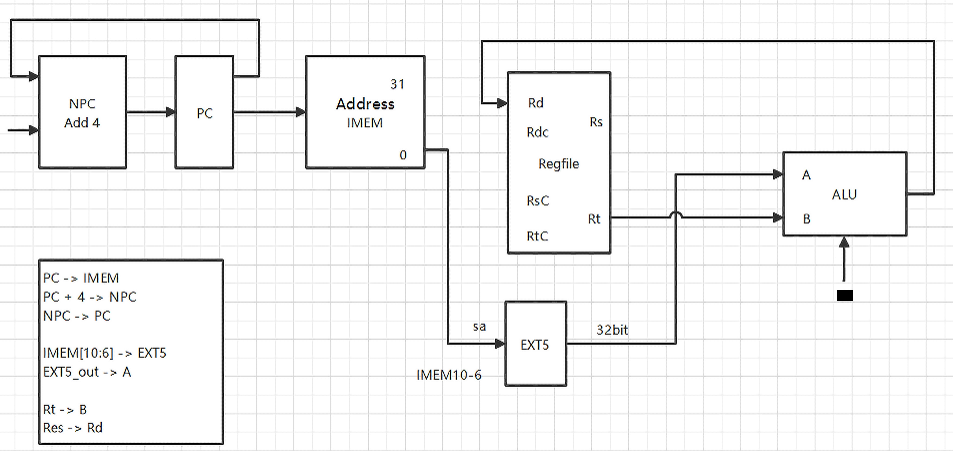
Slt 类型指令包括 slt 和 sltu，其结构通路如图：



使用的器件包括 NPC、PC、指令存储器、寄存器文件以及 ALU，其中，ALU 的信号为 slt 和 sltu.

### 2.3 Sll 类型指令

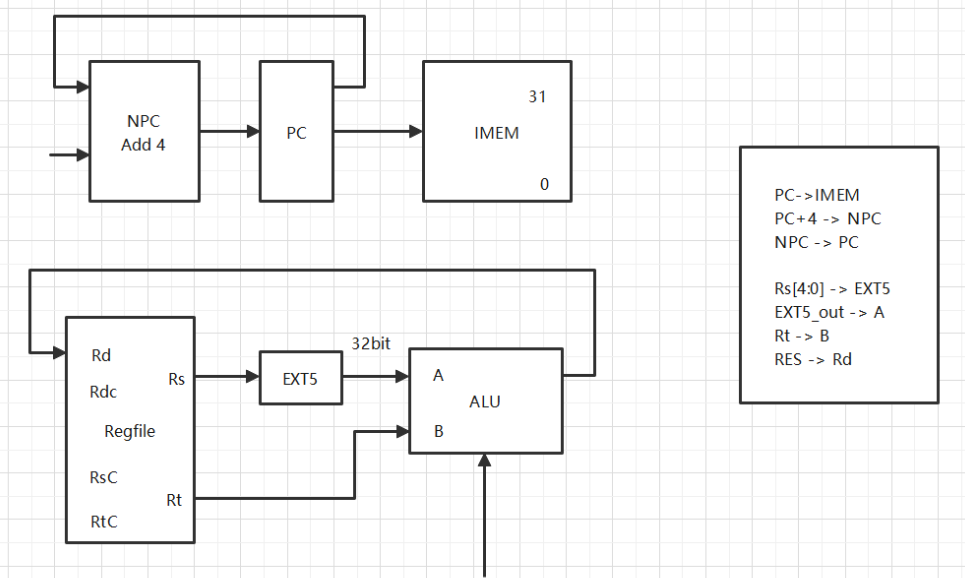
Sll 类型指令包括 sll/srl/sra，其结构通路如图：



使用的器件包括 NPC、PC、指令寄存器、寄存器文件、数据扩展器以及 ALU，其中，ALU 信号为 sll/srl/sra.

### 2.4 Sllv 类型指令

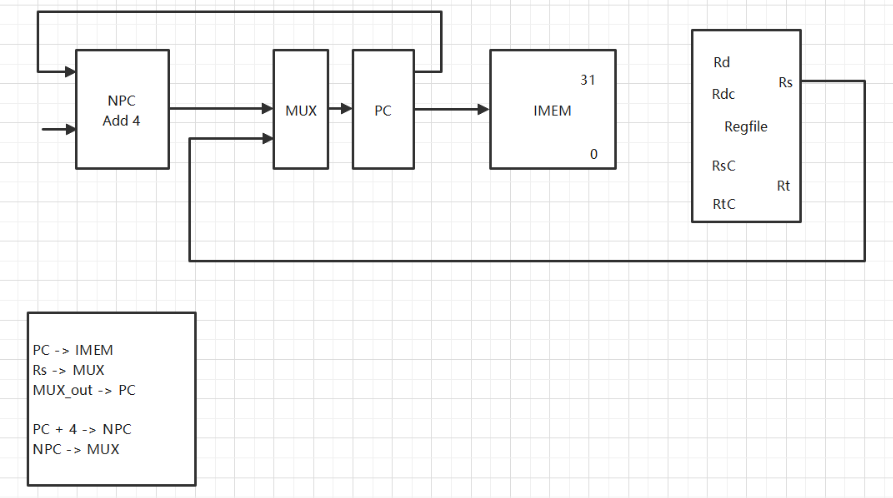
Sllv 类型指令包括 sllv/srlv/srav，其结构通路如图：



使用的器件包括 NPC、PC、指令存储器、寄存器文件、数据扩展器以及 ALU，其中，ALU 信号为 sll/srl/sra.

### 2.5 Jr 类型指令

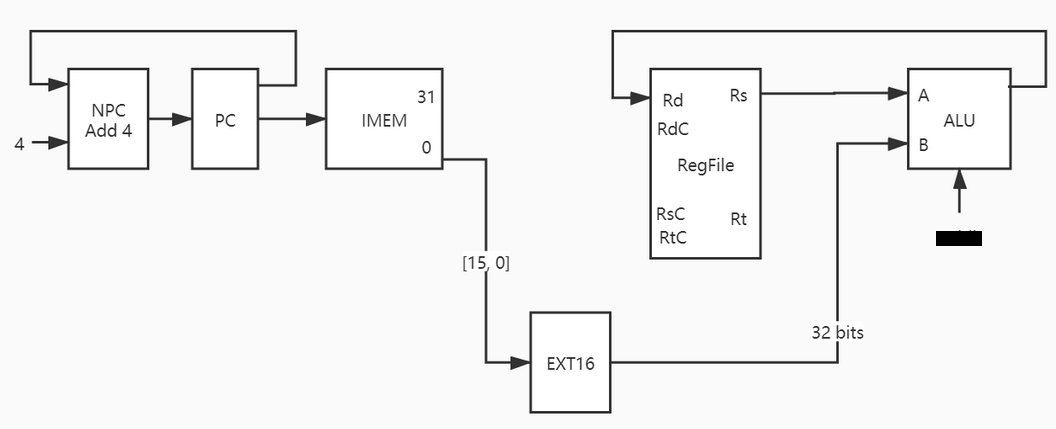
Jr 类型指令即为 jr 指令，其结构通路如图：



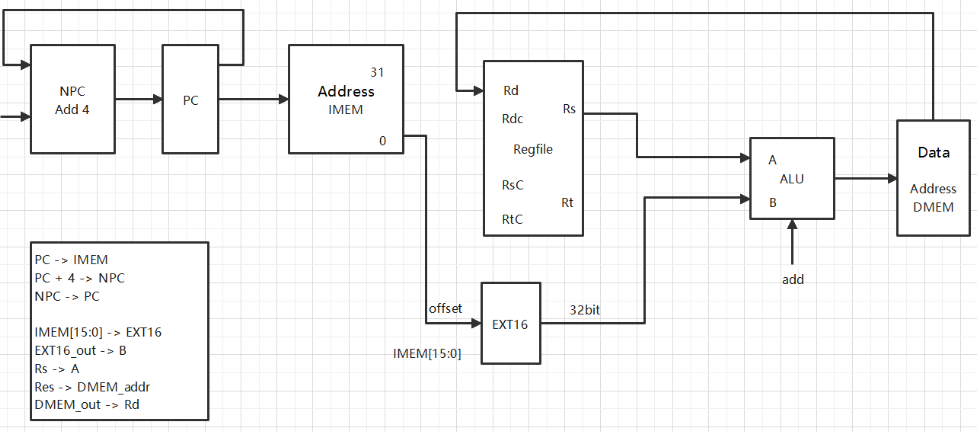
使用的器件包括 NPC、PC、指令存储器、寄存器文件以及二选一 MUX.

### 2.6 Addi 类型指令

Addi 类型指令包括 addi/addiu/andi/ori/xori，除了 ALU 信号之外完全相同，使用的器件包括 NPC、PC、指令存储器、寄存器文件、数据扩展以及 ALU.

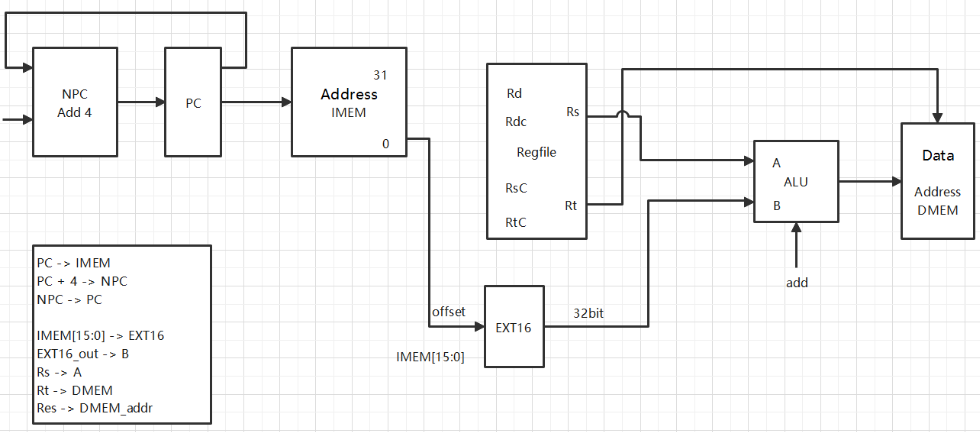


### 2.7 lw 指令



使用的器件包括 NPC、PC、指令存储器、寄存器文件、数据扩展器、数据存储器以及 ALU.

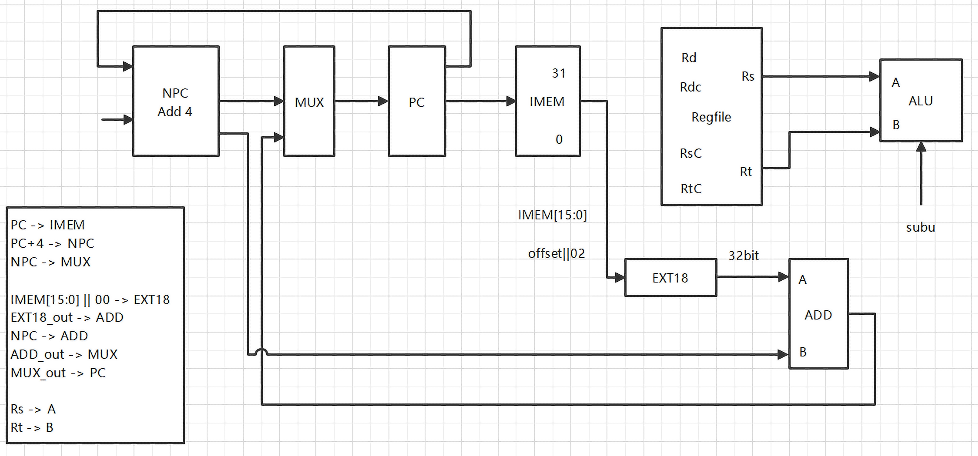
### 2.8 sw 指令



使用的器件同 lw 指令.

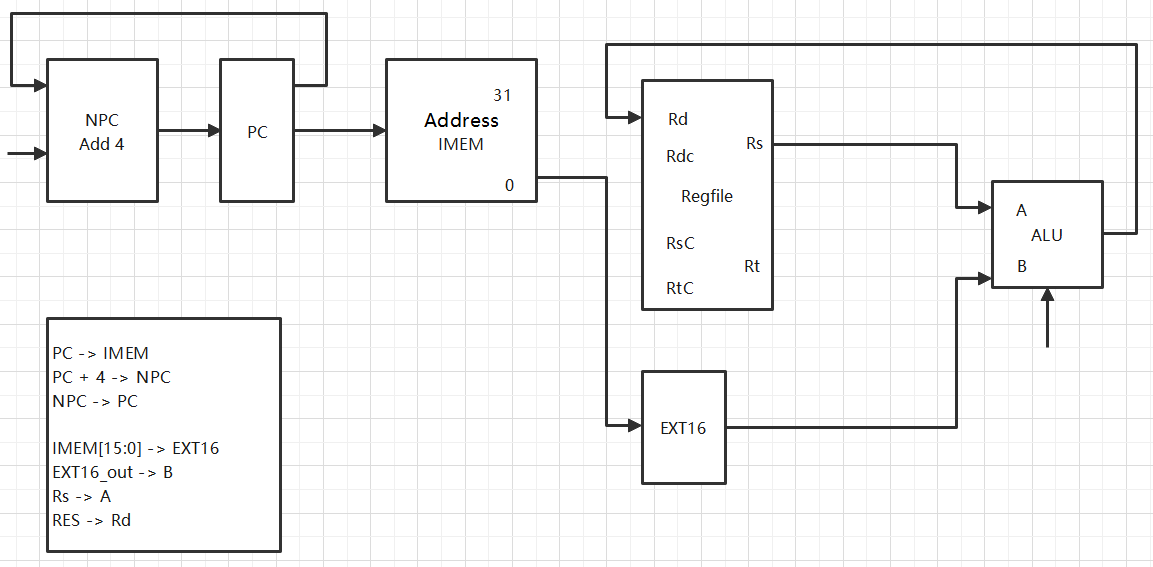
### 2.9 Beq 和 Bne 指令

Beq 和 Bne 是两条比较跳转指令，区别在于 MUX 的信号不同，其余部分完全相同，其结构通路如图：



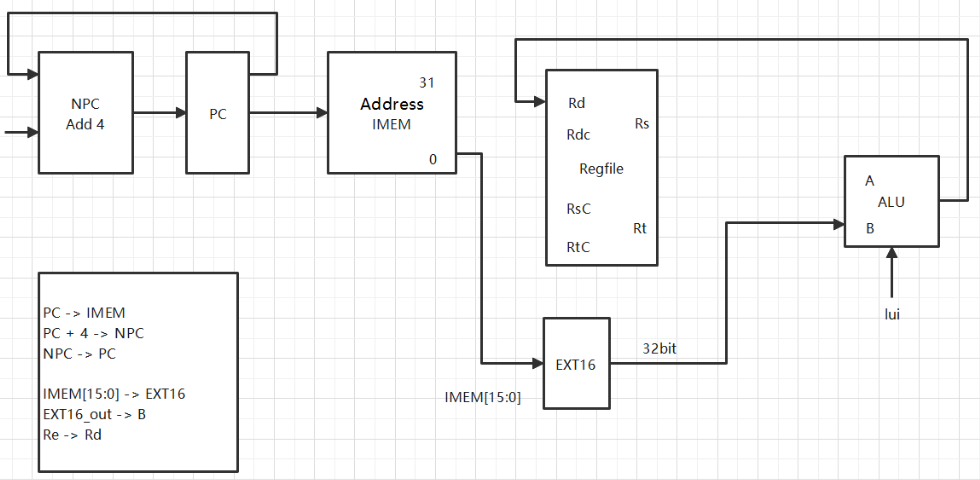
### 2.10 Slti 和 Sltiu 指令

Slti 和 Sltiu 指令的区别在于 ALU 的控制信号不同，其余部分完全相同，结构通路如图：



使用的器件如图所示，ALU 的控制信号为 slt 和 sltu.

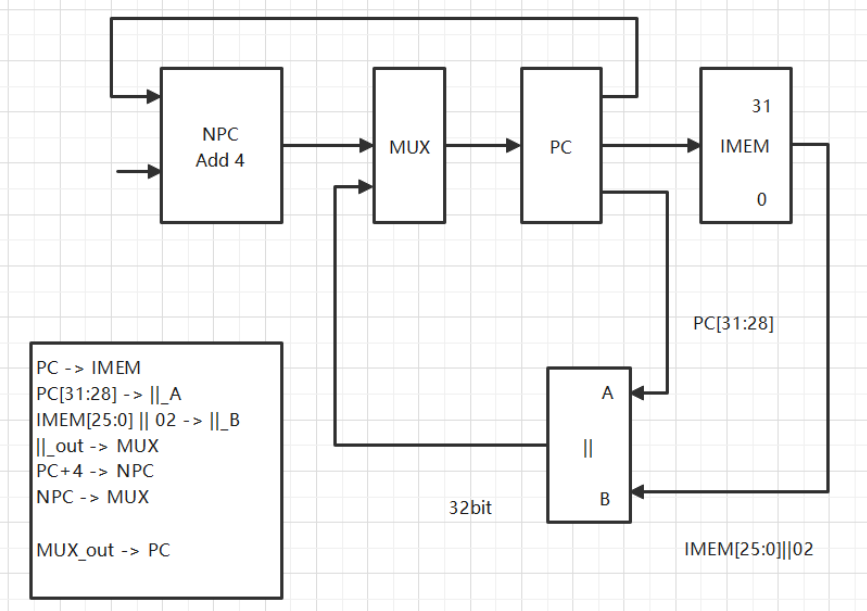
### 2.11 Lui 指令



Lui 指令比较简单，只要通过 ALU 的运算即可完成，结构通路如上图所示。

### 2.12 J 指令

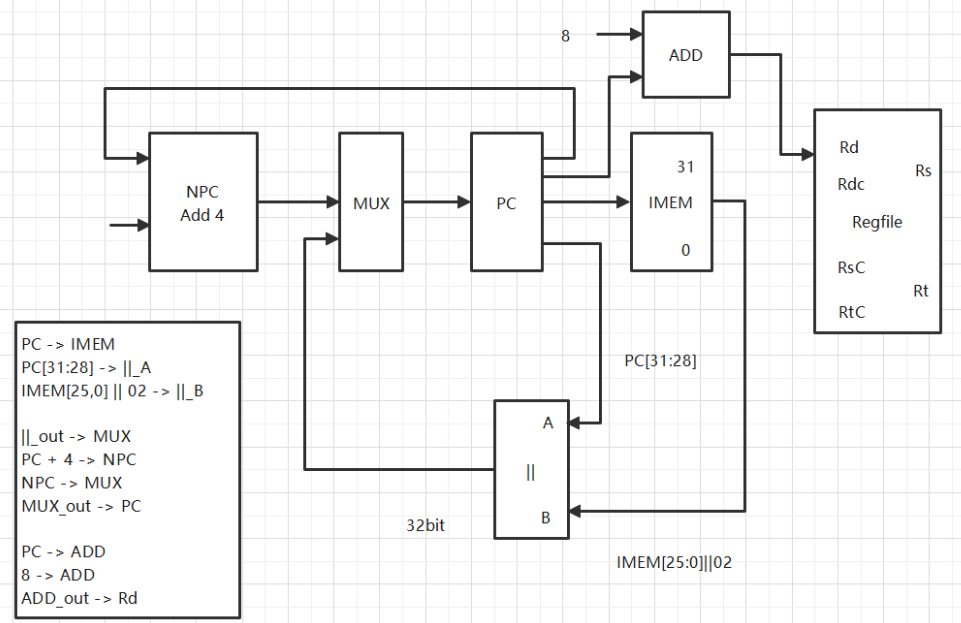
J 指令通路设计如下：



J 指令增加了 II 器件，功能为左移.

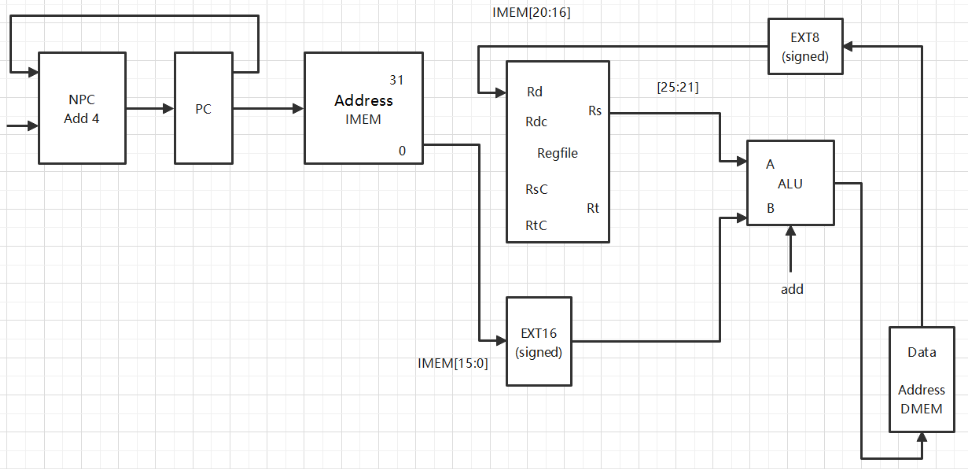
### 2.13 Jal 指令

最后的 Jal 指令如图所示：



### 2.14 l 类指令

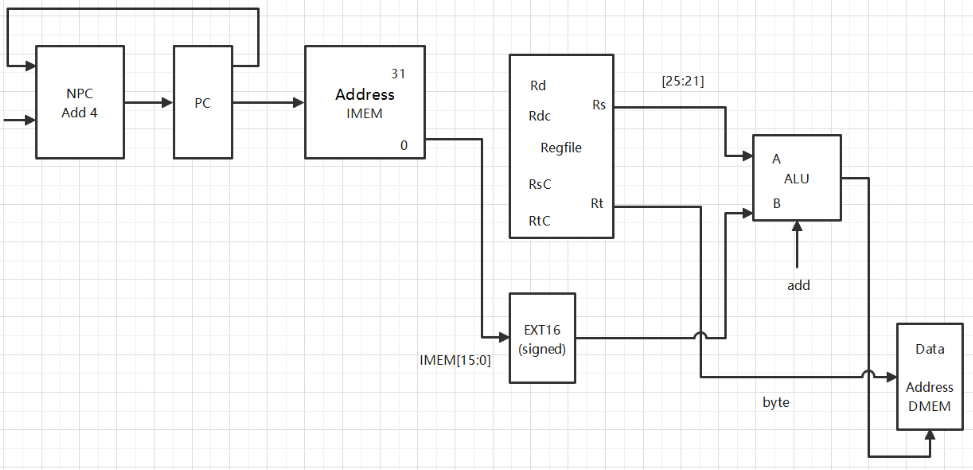
所谓的 l 类指令，即取字节，取半字的指令，lb，lh，lbu，lhu四条指令，这些指令和 lw 指令相似，不过需要符号位的扩展，实现如下：



其中 EXT8 的扩展器可以被 EXT16 扩展器替代，有无符号根据指令具体内容而定。

### 2.15 s 类指令

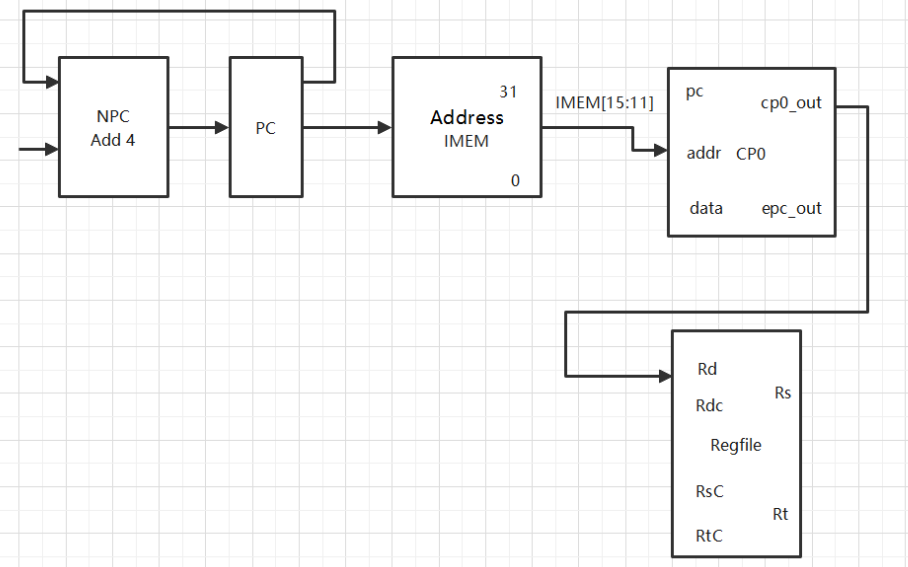
所谓的 s 类指令，便是取字节，取半字的指令，与 l 类指令操作相反，部件图如下：

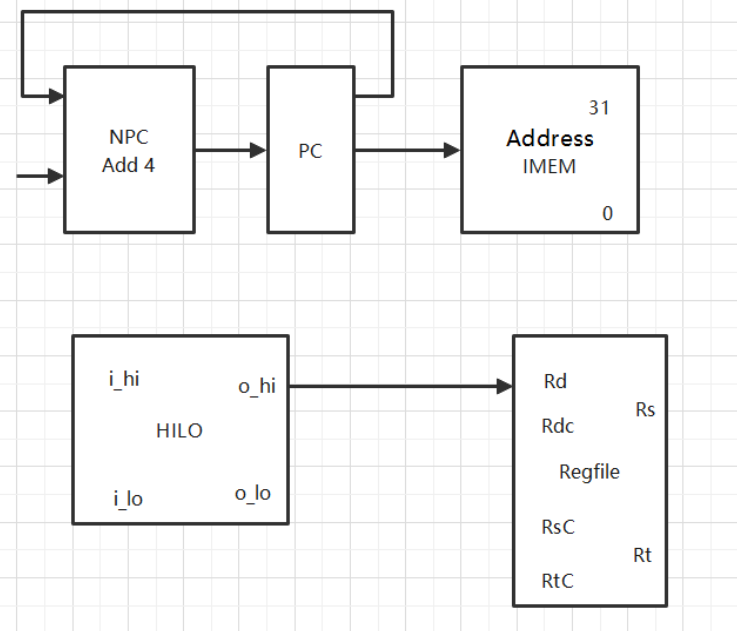


此类指令包括 sb 和 sh 指令.

### 2.16 读寄存器的指令

此类指令包括 mfc0，mfhi，mflo，是对特定寄存器的读取操作，部件图如下：

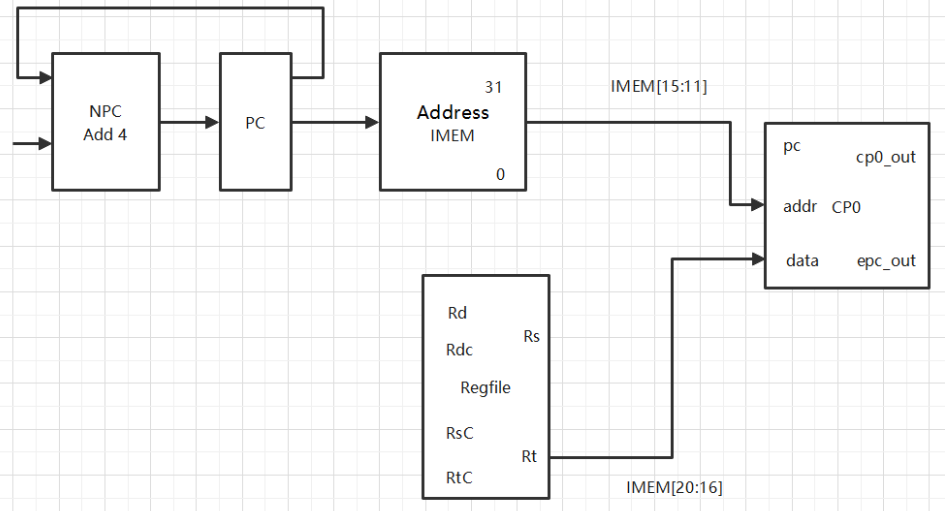


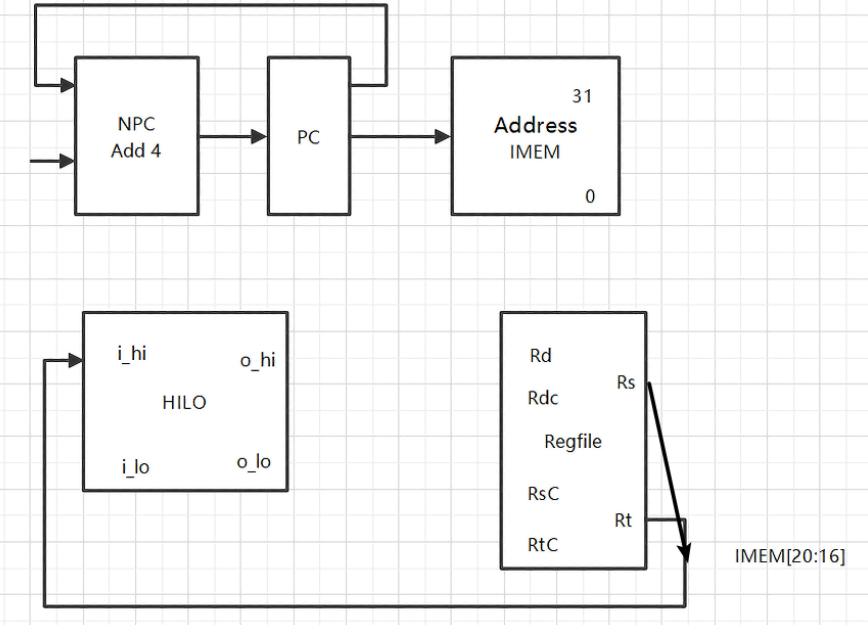


Mflo 指令可以类推，此处省略.

### 2.17 写寄存器的指令

此类指令和读寄存器指令相反，实行对寄存器的写操作，部件图如下：

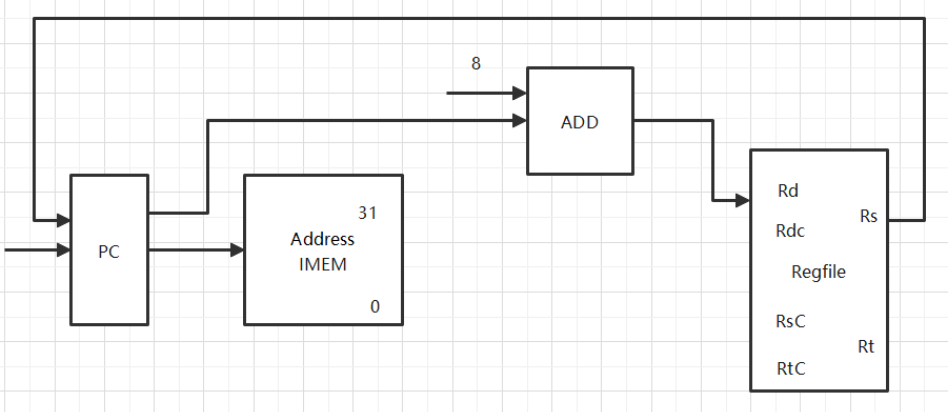




此类指令有 mtc0，mthi，mtlo.

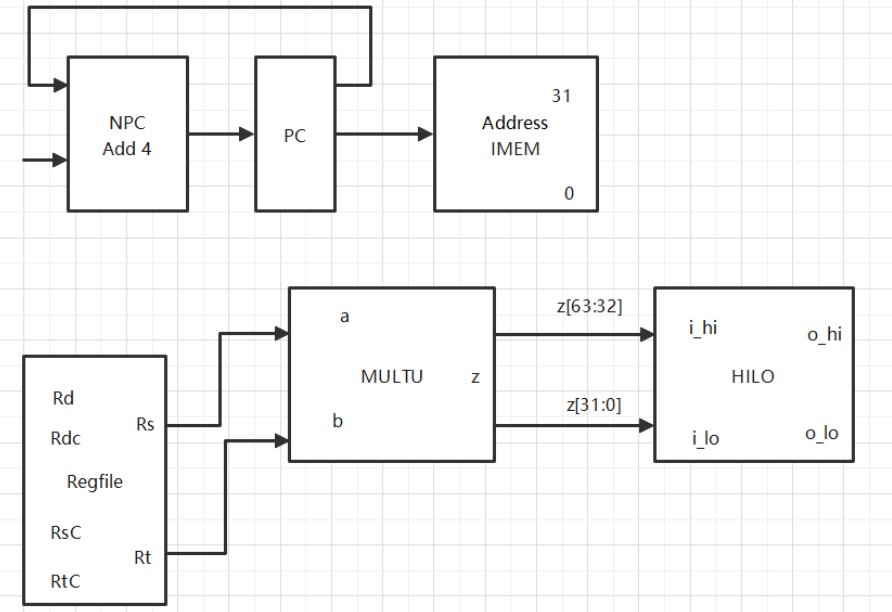
### 2.18 跳转指令 Jarl

和其余跳转指令相似，如下：



### 2.20 乘除法指令

实现乘除法，包括有无符号，注意 mul 与 multu 的区别，如下：



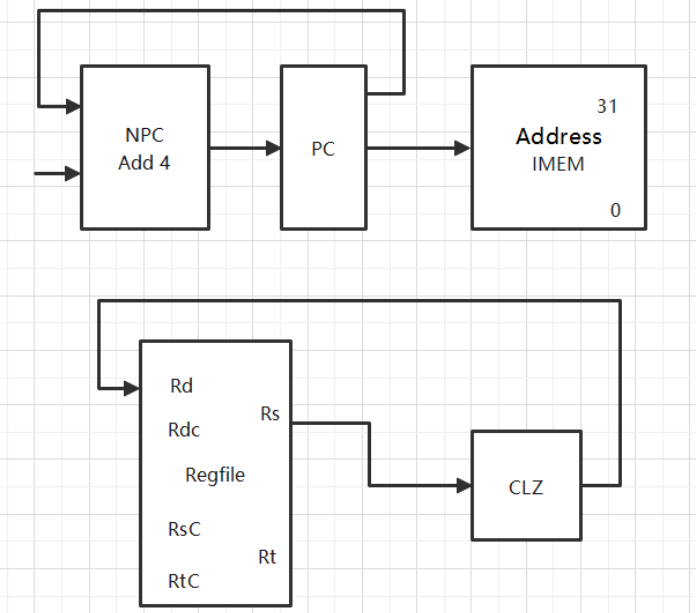
而 mul 指令则是写入寄存器 regfiles，并不写入寄存器 HI 与 LO.

### 2.21 异常处理指令

此类指令包括 eret，syscall，break以及 teq，是 CPU 的一种异常处理，是关于 CP0 寄存器的处理方式.

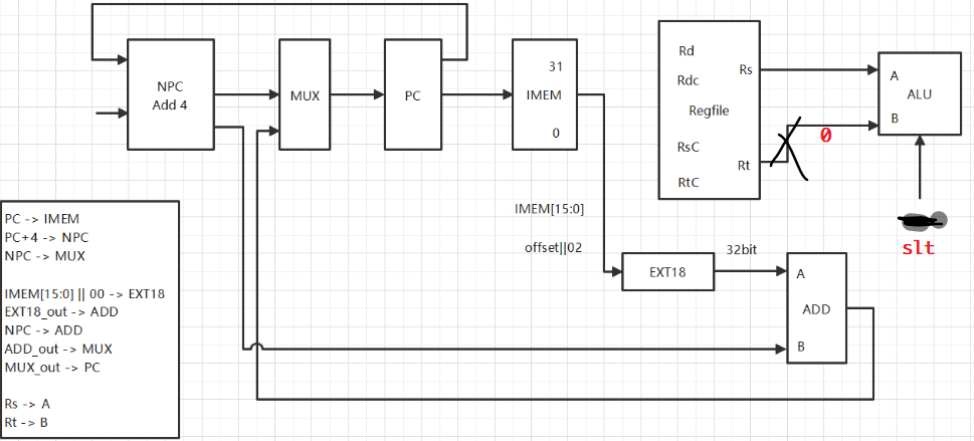
### 2.22 clz 指令

54 增加的指令，前导零计数，可以设计相关的部件实现，如下所示：



### 2.23 bgez 指令

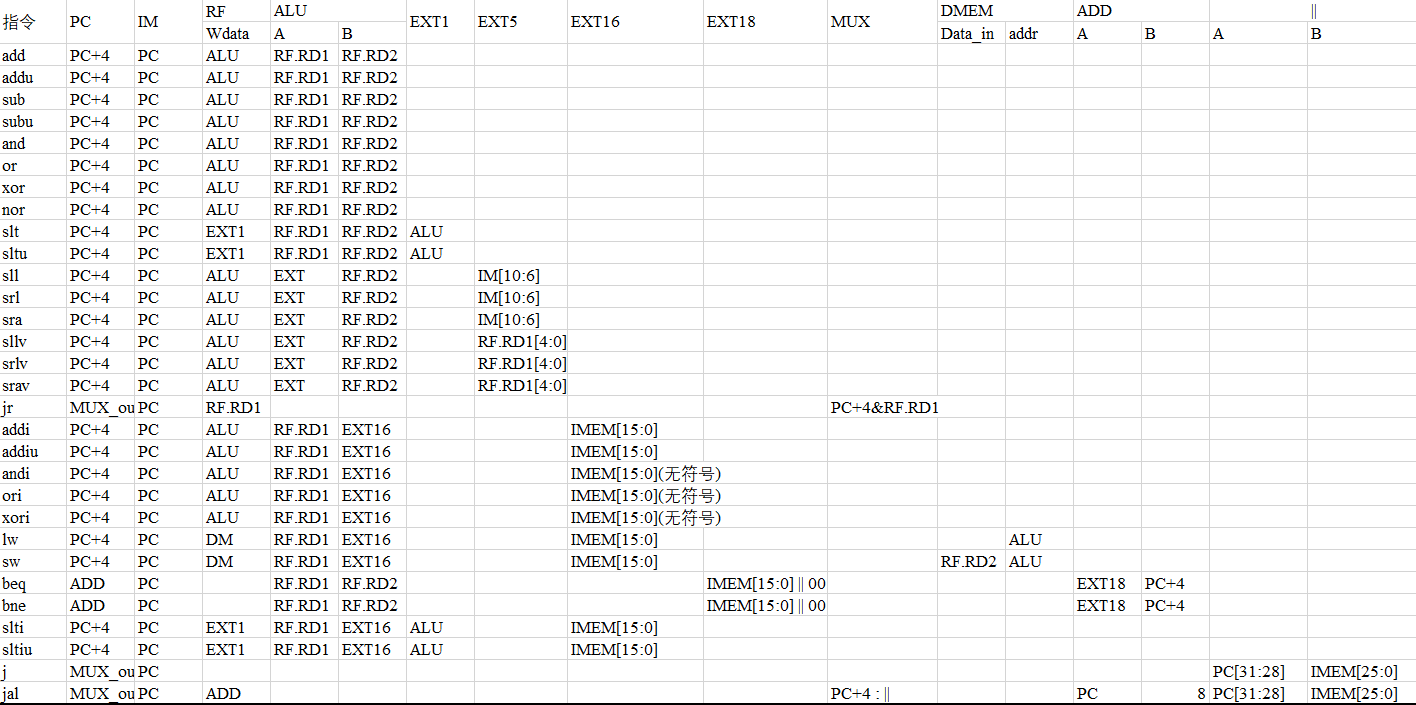
分支跳转指令，部件图如下：



## 3. CPU 整体信号以及通路

### 3.1 54 指令控制部件表

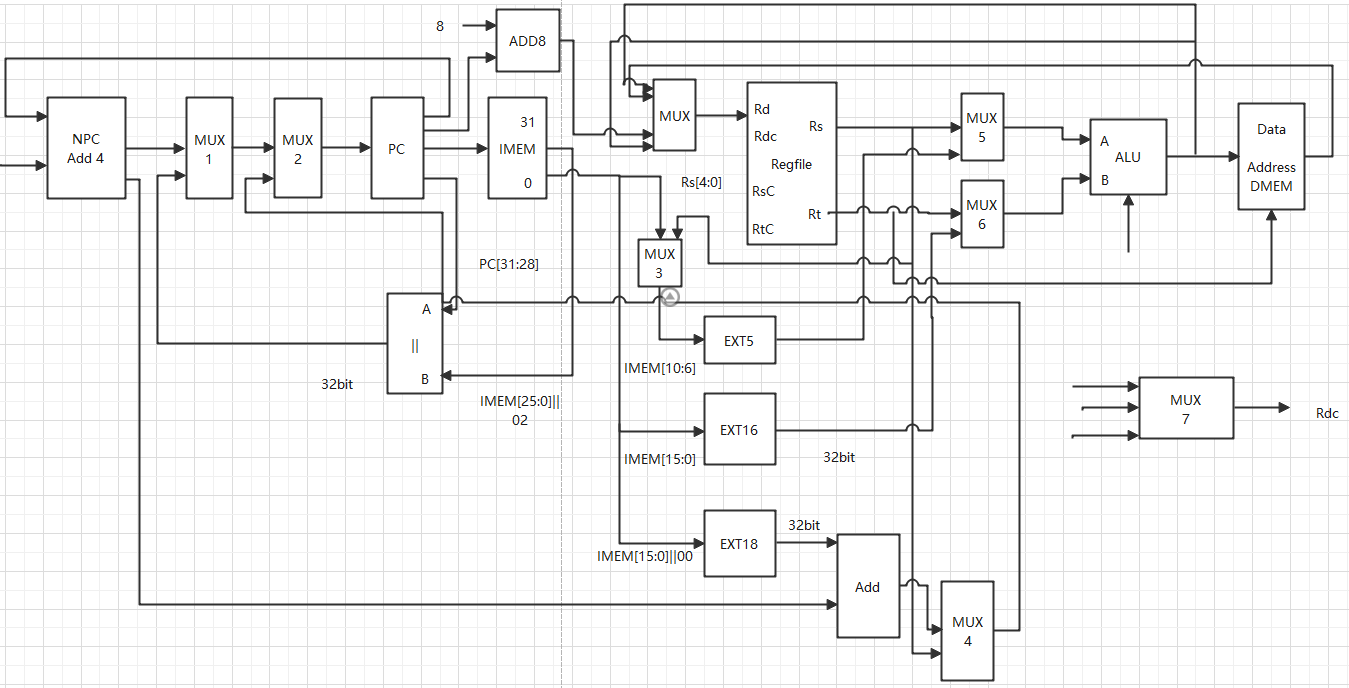
31 条指令的控制部件以及输入的信号如下表所示：



54 条在此基础之上添加相应的部件即可，不做详细绘制.

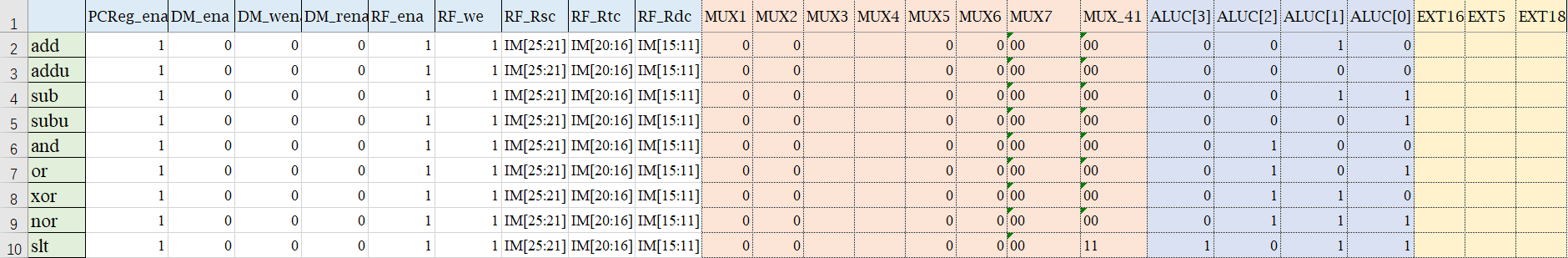
### 3.2 CPU 结构通路图

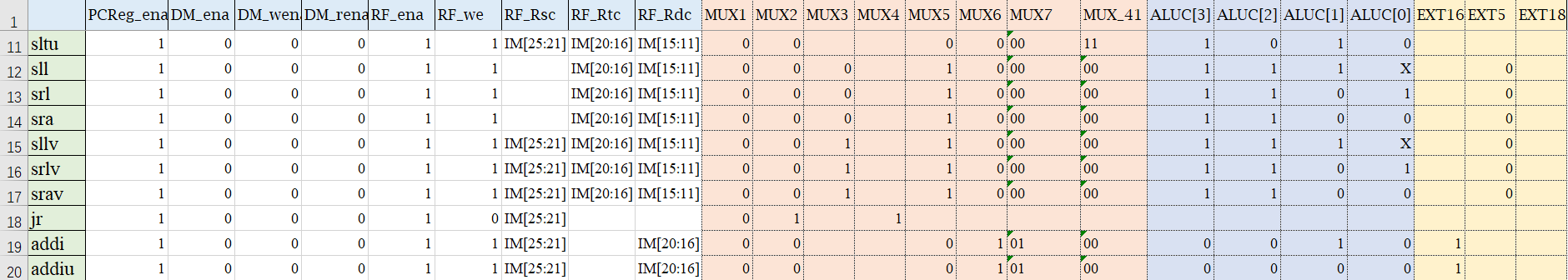
结合所有单条指令的结构图，通过引入多路选择器的方式将所有信号整合，便完成了单周期 54 条指令 CPU 的通路图：

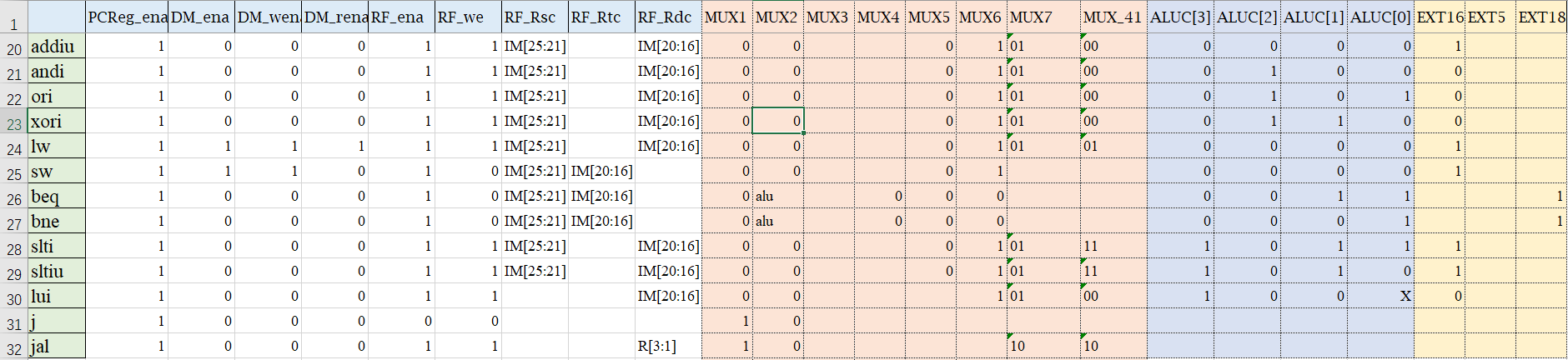


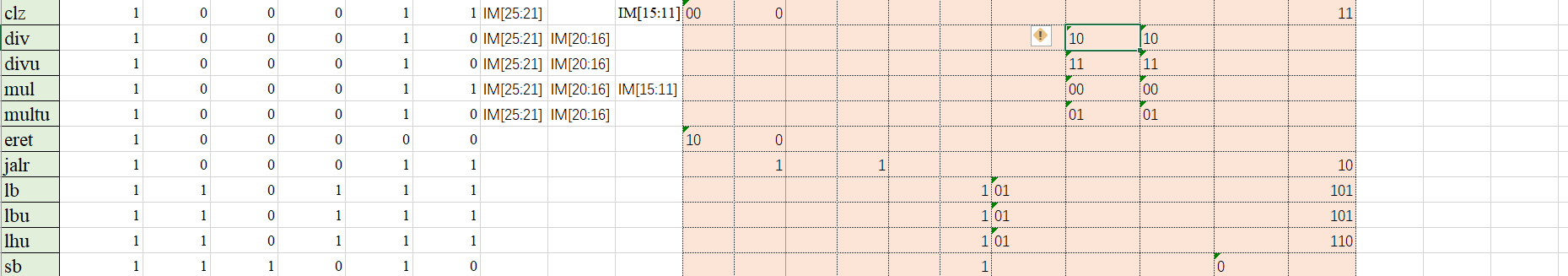
### 3.3 CPU 指令控制信号

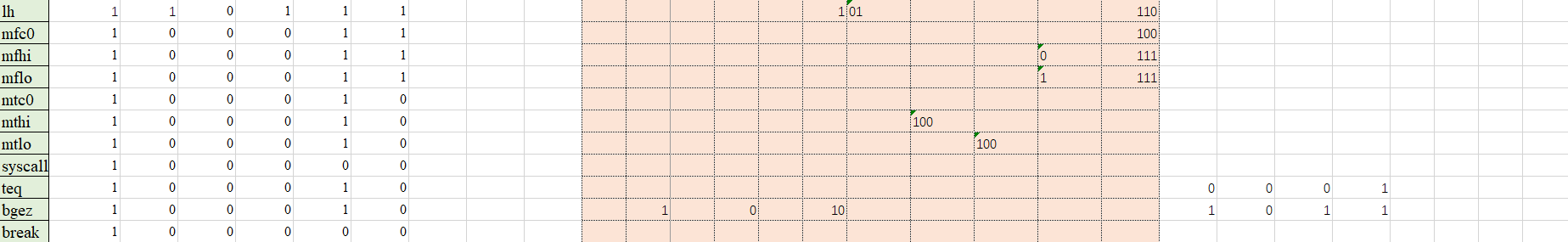
在 54 条指令中， CPU 各部件的控制信号具体如下所示：











通过以上所有的结构图以及信号，便可以设计出 CPU.

# 四、 模块设计

## 1. 顶层模块设计

根据网站提交要求，顶层模块设置如下：

|  |
| --- |
| module sccomp\_dataflow(  input clk\_in,  input reset,  output [31:0] inst,  output [31:0] pc  ); |

## 2. CPU 模块

CPU 模块与顶层模块进行信号的交换控制，形成完整通路，模块设计如下：

|  |
| --- |
| module cpu(  input clk,  input reset,  output [31:0] inst,  output [31:0] pc  ); |

## 3. IMEM 模块

指令存储器的 IP 核由 vivado 提供，需要在使用时调用，设计如下：

|  |
| --- |
| module IMEM(  input [10:0] addr,  output [31:0] instr  ); |

## 4. DMEM 模块

DMEM 模块需要人工书写，相较于 31 条添加了 byte 和 halfword 信号，其模块设计如下：

|  |
| --- |
| module DMEM(  input clk,  input ena,  input wsignal,  input rsignal,  input byte,  input halfword,  input [10:0] addr,  input [31:0] wdata,  output [31:0] rdata  ); |

## 5. Regfile 模块

Regfile 是寄存器文件，模块设计如下：

|  |
| --- |
| module regfile(  input clk,  input ena,  input rst,  input we, // high--write, low--read  input [4:0] Rsc,  input [4:0] Rtc,  input [4:0] Rdc,  input [31:0] Rd,  output [31:0] Rs,  output [31:0] Rt  ); |

## 6. ALU 模块

ALU 模块负责运算部分，模块设计如下：

|  |
| --- |
| module ALU(  input [31:0] a,  input [31:0] b,  input [3:0] aluc,  output reg [31:0] r,  output reg zero,  output reg carry,  output reg negative,  output reg overflow  ); |

## 7. PCReg 模块

PC 寄存器，初始化时为 pc 变量赋值，模块设计如下：

|  |
| --- |
| module PCReg(  input clk,  input rst,  input ena,  input [31:0] data\_in,  output reg [31:0] data\_out  ); |

## 8. Controller 模块

Controller 模块负责对各部件信号的控制，此模块可以包含在 CPU 中，为了保证结构清晰，将其单独提取出，设计如下：

|  |
| --- |
| module Controller(  input z,  input n, // negative  input [53:0] icode,  input DIV\_busy,  input DIVU\_busy,  output [1:0] M1,  output M2,  output M3,  output M4,  output M5,  output [1:0] M6,  output [1:0] M7,  output [2:0] M8,  output [2:0] M9,  output [1:0] M10,  output [2:0] M,  output [3:0] ALUC,  output RF\_we,  output DM\_ena,  output DM\_we,  output DM\_re,  output DM\_byte,  output DM\_half,  output EXT16\_s,  output HI\_ena,  output HI\_we,  output LO\_ena,  output LO\_we,  output EXT8\_s,  output EXT16\_s\_2,  output [4:0] CP0\_cause,  output CP0\_mf,  output CP0\_mt,  output MULT\_CS,  output MULTU\_CS,  output DIV\_start,  output DIVU\_start,  output CP0\_exc,  output CP0\_eret  ); |

## 9. 其它模块

除了主要的部件之外，还有一些小部件，包括多路选择器、左移功能器件、符号扩展器、加法器以及实现特定加法功能的 NPC 和 ADD8 部件，以及乘除法，clz 计算等，模块设计比较简单，不做列举。

# 五、 测试模块设计

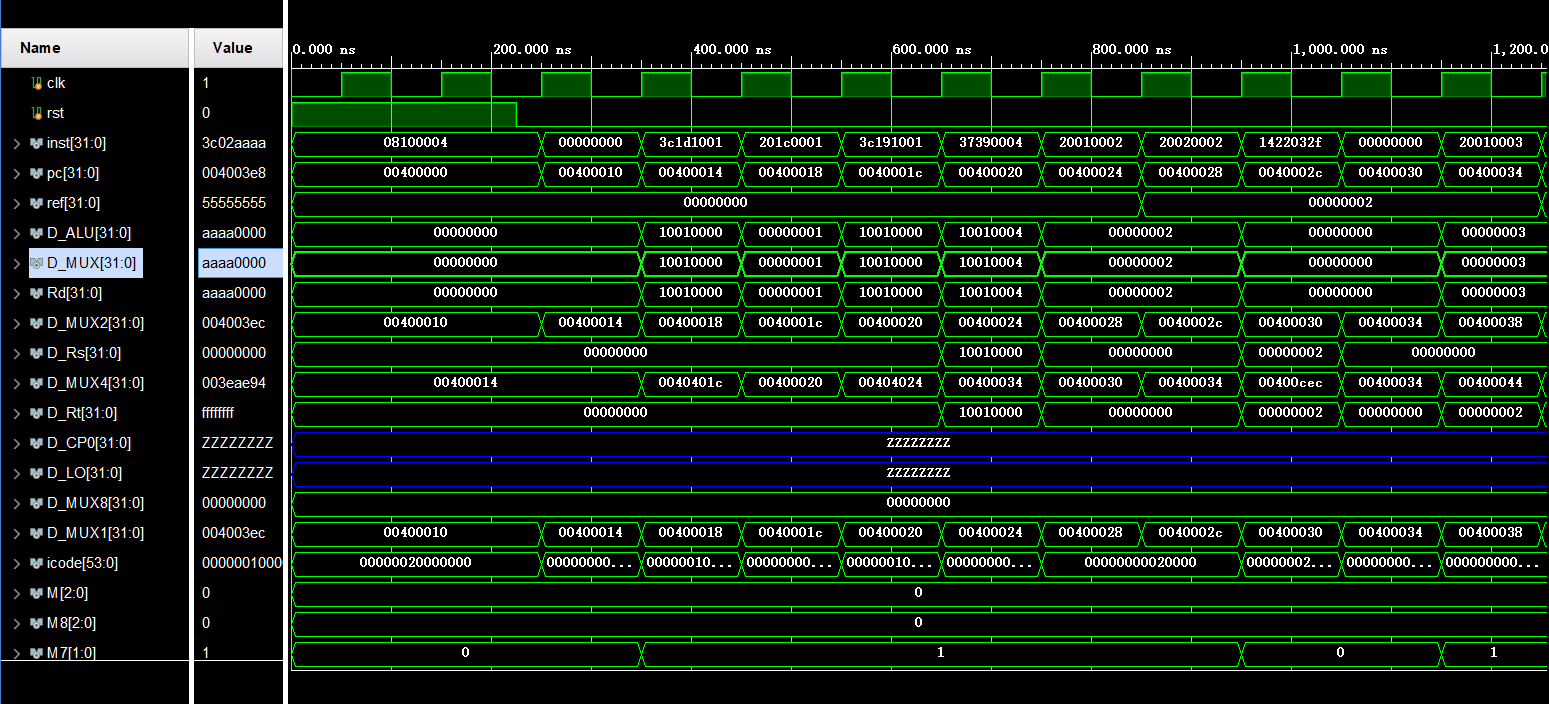
测试模块可以用来检验 CPU 指令设计是否正确，判断标准可以通过寄存器的值进行观察，因此，测试模块实现寄存器值的文件输出，同时也可以设置一些中间变量进行观测，下面是输出到文件的部分代码：

|  |
| --- |
| …………  $fdisplay(file\_open, "regfile0: %h", sc.sccpu.cpu\_ref.array\_reg[0]);  $fdisplay(file\_open, "regfile1: %h", sc.sccpu.cpu\_ref.array\_reg[1]);  …………  $fdisplay(file\_open, "pc: %h", sc.pc);  $fdisplay(file\_open, "instr: %h", sc.inst);  …………  sccomp\_dataflow sc(  .clk\_in(clk), .reset(rst),  .inst(inst), .pc(pc)  );  ………… |

# 六、 实验结果

## 1. 前仿真测试

### 1.1仿真图像



## 2. 后仿真测试

……