# RISC-V 32流水线CPU的verilog实现

## 1. 实验任务

设计一个32位流水线RISC-V微处理器，具体要求如下

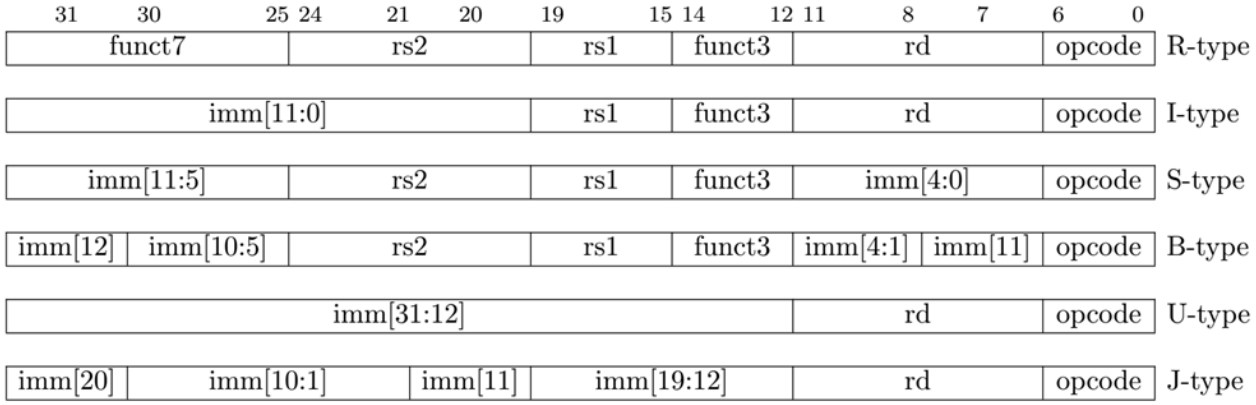
下面详细解析各个指令的含义，便于后续的查询。

运行指令包括：RISC-V 32bit 整型指令集（除去 FENCE,FENCE.I,CSR,ECALL 和 EBREAK 指令）共37条指令

**RV32I Base Instruction Set**

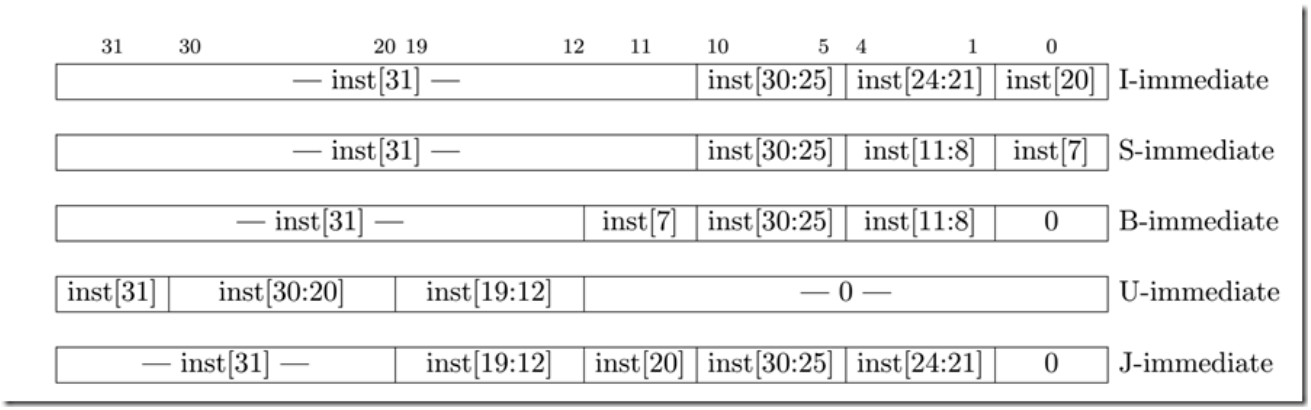
| | | | | | | |
|---|---|---|---|---|---|---|
| imm[31:12] | | | | rd | 0110111 | LUI |
| imm[31:12] | | | | rd | 0010111 | AUIPC |
| imm[20\|10:1\|11\|19:12] | | | | rd | 1101111 | JAL |
| imm[11:0] | | rs1 | 000 | rd | 1100111 | JALR |
| imm[12\|10:5] | rs2 | rs1 | 000 | imm[4:1\|11] | 1100011 | BEQ |
| imm[12\|10:5] | rs2 | rs1 | 001 | imm[4:1\|11] | 1100011 | BNE |
| imm[12\|10:5] | rs2 | rs1 | 100 | imm[4:1\|11] | 1100011 | BLT |
| imm[12\|10:5] | rs2 | rs1 | 101 | imm[4:1\|11] | 1100011 | BGE |
| imm[12\|10:5] | rs2 | rs1 | 110 | imm[4:1\|11] | 1100011 | BLTU |
| imm[12\|10:5] | rs2 | rs1 | 111 | imm[4:1\|11] | 1100011 | BGEU |
| imm[11:0] | | rs1 | 000 | rd | 0000011 | LB |
| imm[11:0] | | rs1 | 001 | rd | 0000011 | LH |
| imm[11:0] | | rs1 | 010 | rd | 0000011 | LW |
| imm[11:0] | | rs1 | 100 | rd | 0000011 | LBU |
| imm[11:0] | | rs1 | 101 | rd | 0000011 | LHU |
| imm[11:5] | rs2 | rs1 | 000 | imm[4:0] | 0100011 | SB |
| imm[11:5] | rs2 | rs1 | 001 | imm[4:0] | 0100011 | SH |
| imm[11:5] | rs2 | rs1 | 010 | imm[4:0] | 0100011 | SW |
| imm[11:0] | | rs1 | 000 | rd | 0010011 | ADDI |
| imm[11:0] | | rs1 | 010 | rd | 0010011 | SLTI |
| imm[11:0] | | rs1 | 011 | rd | 0010011 | SLTIU |
| imm[11:0] | | rs1 | 100 | rd | 0010011 | XORI |
| imm[11:0] | | rs1 | 110 | rd | 0010011 | ORI |
| imm[11:0] | | rs1 | 111 | rd | 0010011 | ANDI |
| 0000000 | shamt | rs1 | 001 | rd | 0010011 | SLLI |
| 0000000 | shamt | rs1 | 101 | rd | 0010011 | SRLI |
| 0100000 | shamt | rs1 | 101 | rd | 0010011 | SRAI |
| 0000000 | rs2 | rs1 | 000 | rd | 0110011 | ADD |
| 0100000 | rs2 | rs1 | 000 | rd | 0110011 | SUB |
| 0000000 | rs2 | rs1 | 001 | rd | 0110011 | SLL |
| 0000000 | rs2 | rs1 | 010 | rd | 0110011 | SLT |
| 0000000 | rs2 | rs1 | 011 | rd | 0110011 | SLTU |
| 0000000 | rs2 | rs1 | 100 | rd | 0110011 | XOR |
| 0000000 | rs2 | rs1 | 101 | rd | 0110011 | SRL |
| 0100000 | rs2 | rs1 | 101 | rd | 0110011 | SRA |
| 0000000 | rs2 | rs1 | 110 | rd | 0110011 | OR |
| 0000000 | rs2 | rs1 | 111 | rd | 0110011 | AND |
| 0000 \| pred \| succ | | 00000 | 000 | 00000 | 0001111 | FENCE |
| 0000 \| 0000 \| 0000 | | 00000 | 001 | 00000 | 0001111 | FENCE.I |
| 000000000000 | | 00000 | 000 | 00000 | 1110011 | ECALL |
| 000000000001 | | 00000 | 000 | 00000 | 1110011 | EBREAK |
| csr | | rs1 | 001 | rd | 1110011 | CSRRW |
| csr | | rs1 | 010 | rd | 1110011 | CSRRS |
| csr | | rs1 | 011 | rd | 1110011 | CSRRC |
| csr | | zimm | 101 | rd | 1110011 | CSRRWI |
| csr | | zimm | 110 | rd | 1110011 | CSRRSI |
| csr | | zimm | 111 | rd | 1110011 | CSRRCI |

首先，RV32I指令格式为




imm表示指令中的立即数，比如imm[11:0]，表示一个12位的立即数，它的高20位会符号位扩展，imm[31:12]表示一个32位的立即数，它的低12位会补0。

下图是各种指令格式扩展后的32位立即数。




与mips相比，格式有如下的变化

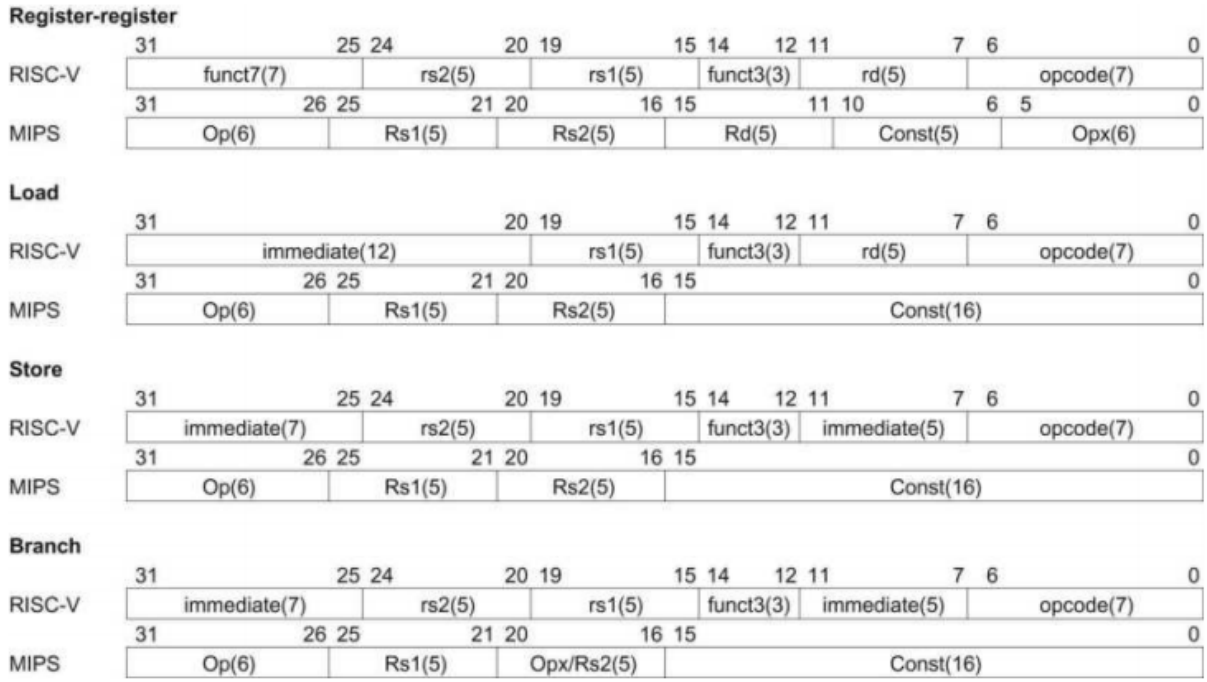
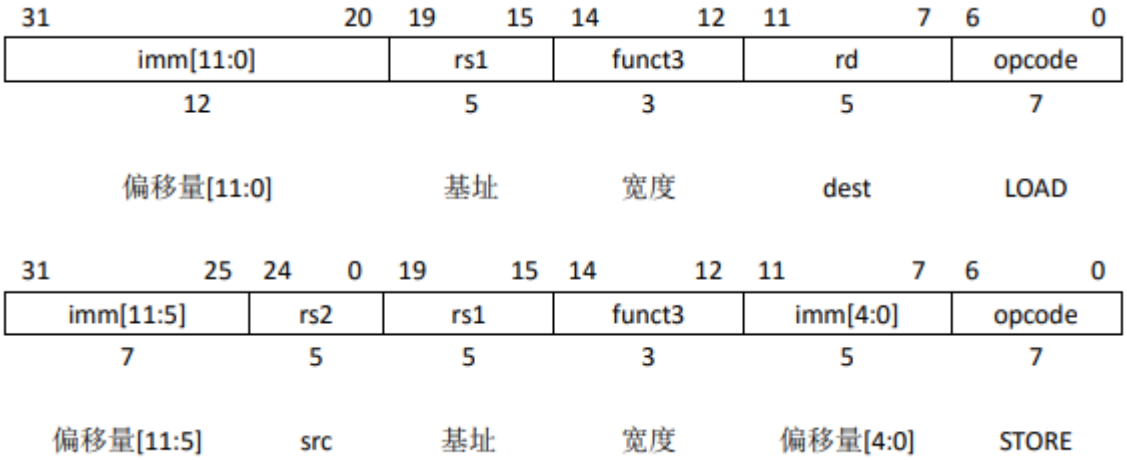



**FIGURE 2.29 Instruction formats of RISC-V and MIPS.**

要实现的指令可以分成如下几个种类，详细的内容请查看`Instr.xls`文件

## 1.1. Load和Store指令




| | | | | | | |
|---|---|---|---|---|---|---|
| imm[11:0] | | rs1 | 000 | rd | 0000011 | LB |
| imm[11:0] | | rs1 | 001 | rd | 0000011 | LH |
| imm[11:0] | | rs1 | 010 | rd | 0000011 | LW |
| imm[11:0] | | rs1 | 100 | rd | 0000011 | LBU |
| imm[11:0] | | rs1 | 101 | rd | 0000011 | LHU |
| imm[11:5] | rs2 | rs1 | 000 | imm[4:0] | 0100011 | SB |
| imm[11:5] | rs2 | rs1 | 001 | imm[4:0] | 0100011 | SH |
| imm[11:5] | rs2 | rs1 | 010 | imm[4:0] | 0100011 | SW |

Load和store指令在寄存器和存储器之间传输数值。Load指令编码为I类格式，而store指 令编码为S类格式。

**有效字节地址**是通过将寄存器`rs1`与**符号扩展**的12位偏移量相加而获得的。

Load指令将存储器中的一个值复制到寄存器rd中。Store指令将寄存器rs2中的值复制到存储 器中。

`LW`指令将一个32位数值从存储器复制到`rd`中。

`LH`指令从存储器中读取一个16位数值， 然后将其进行**符号扩展**到32位，再保存到`rd`中。

`LHU`指令存储器中读取一个16位数值，然后 将其进行**零扩展**到32位，再保存到`rd`中。

对于8位数值，`LB`和`LBU`指令的定义与前面类似。

`SW`、 `SH`、`SB`指令分别将从`rs2`低位开始的32位、16位、8位数值保存到存储器中

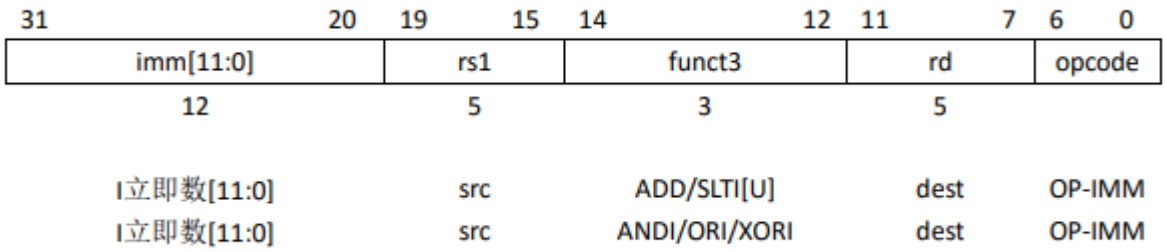
注意，装入目的寄存器如果为x0，将会产生一个异常。

## 1.2. 整数计算指令

(算术，逻辑指令，比较指令以及移位指令)

计算指令在寄存器和寄存器之间，或者在寄存器和立即数之间进行算术或逻辑运算。指令格式为I，R或者U型。整数计算指令不会产生异常。我们能够通过`ADDI x0, x0, 0`来模拟`NOP`指令，该指令除了改变`pc`值外，不会改变其它任何用户状态。

### 1.2.1. 整数寄存器-立即数指令

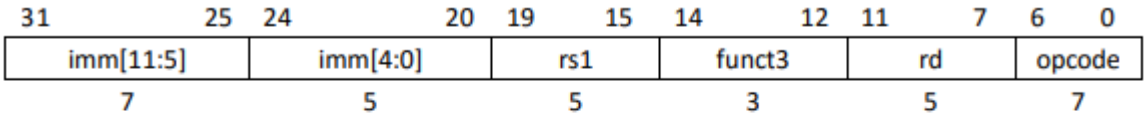



| | | | | | |
|---|---|---|---|---|---|
| imm[11:0] | rs1 | 000 | rd | 0010011 | ADDI |
| imm[11:0] | rs1 | 010 | rd | 0010011 | SLTI |
| imm[11:0] | rs1 | 011 | rd | 0010011 | SLTIU |
| imm[11:0] | rs1 | 100 | rd | 0010011 | XORI |
| imm[11:0] | rs1 | 110 | rd | 0010011 | ORI |
| imm[11:0] | rs1 | 111 | rd | 0010011 | ANDI |

`ADDI`将**符号扩展**的12位立即数加到寄存器`rs1`上。算术溢出被忽略，而结果就是运算结果的低XLEN位。`ADDI rd,rs1,0`用于实现`MV rd,rs1`汇编语言伪指令。

`SLTI`（set less than immediate）将数值`1`放到寄存器`rd`中，如果寄存器`rs1`小于**符号扩展**的立即数（比较时，两者都作为**有符号数**），否则将`0`写入`rd`。`SLTIU`与之相似，但是将两者作 为**无符号数**进行比较（也就是说，立即数被首先**符号扩展**为XLEN位，然后被作为一个**无符号数**）。注意，`SLTIU rd,rs1,1`将设置`rd`为`1`，如果`rs1`等于`0`，否则将`rd`设置为`0`（汇编语言伪指 令`SEQZ rd,rs`）。

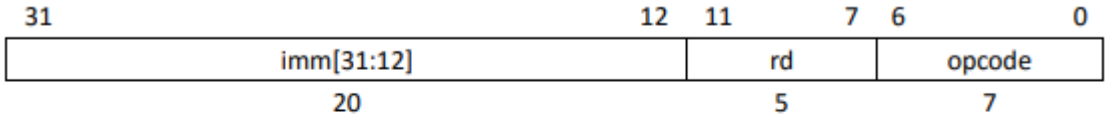
`ANDI`、`ORI`、`XORI`是逻辑操作，在寄存器`rs1`和**符号扩展**的12位立即数上执行**按位**AND、 OR、XOR操作，并把结果写入`rd`。注意，`XORI rd,rs1,-1`在`rs1`上执行一个按位取反操作（汇编 语言伪指令`NOT rd,rs`）。




| | | | | | |
|---|---|---|---|---|---|
| 0000000 | 移位次数[4:0] | src | SLLI | dest | OP-IMM |
| 0000000 | 移位次数[4:0] | src | SRLI | dest | OP-IMM |
| 0100000 | 移位次数[4:0] | src | SRAI | dest | OP-IMM |

| | | | | | | |
|---|---|---|---|---|---|---|
| 0000000 | shamt | rs1 | 001 | rd | 0010011 | SLLI |
| 0000000 | shamt | rs1 | 101 | rd | 0010011 | SRLI |
| 0100000 | shamt | rs1 | 101 | rd | 0010011 | SRAI |

被移位常数次，被编码为I类格式的特例。被移位的操作数放在`rs1`中，移位的次数被编 码到I立即数字段的低5位。右移类型被编码到I立即数的一位高位。`SLLI`是逻辑左移（0被移 入低位）；`SRLI`是逻辑右移（0被移入高位）；`SRAI`是算术右移（原来的符号位被复制到空出的 高位中）。




| | | |
|---|---|---|
| U立即数[31:12] | dest | LUI |
| U立即数[31:12] | dest | AUIPC |

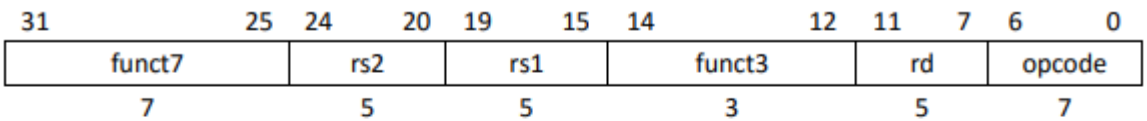
| | | | |
|---|---|---|---|
| imm[31:12] | rd | 0110111 | LUI |
| imm[31:12] | rd | 0010111 | AUIPC |

`LUI`（load upper immediate）用于构建32位常数，并使用U类格式。`LUI`将U立即数放到目标寄存器`rd`的高20位，将`rd`的低12位填0。

`AUIPC`（add upper immediate to pc）用于构建pc相对地址，并使用U类格式。`AUIPC`从20 位U立即数构建一个32位偏移量，将其低12位填0，然后将这个偏移量加到pc上，最后将结 果写入寄存器rd。

### 1.2.2. 整数寄存器-寄存器操作

RV32I定义了几种算术R类操作。所有操作都是读取`rs1`和`rs2`寄存器作为源操作数，并把 结果写入到寄存器`rd`中。`funct7`和`funct3`字段选择了操作的类型

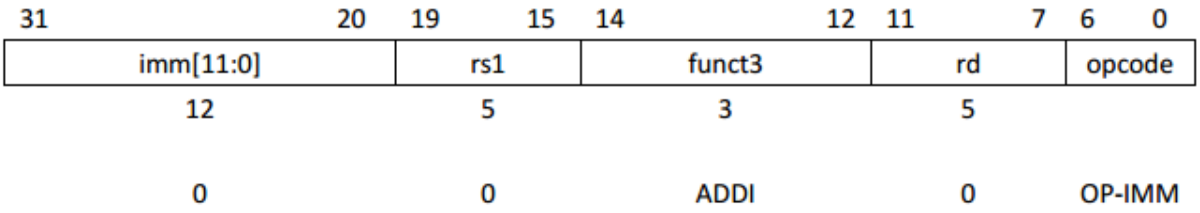



| | | | | | |
|---|---|---|---|---|---|
| 0000000 | src2 | src1 | ADD/SLT/SLTU | dest | OP |
| 0000000 | src2 | src1 | AND/OR/XOR | dest | OP |
| 0000000 | src2 | src1 | SLL/SRL | dest | OP |
| 0100000 | src2 | src1 | SUB/SRA | dest | OP |

| | | | | | | |
|---|---|---|---|---|---|---|
| 0000000 | rs2 | rs1 | 000 | rd | 0110011 | ADD |
| 0100000 | rs2 | rs1 | 000 | rd | 0110011 | SUB |
| 0000000 | rs2 | rs1 | 001 | rd | 0110011 | SLL |
| 0000000 | rs2 | rs1 | 010 | rd | 0110011 | SLT |
| 0000000 | rs2 | rs1 | 011 | rd | 0110011 | SLTU |
| 0000000 | rs2 | rs1 | 100 | rd | 0110011 | XOR |
| 0000000 | rs2 | rs1 | 101 | rd | 0110011 | SRL |
| 0100000 | rs2 | rs1 | 101 | rd | 0110011 | SRA |
| 0000000 | rs2 | rs1 | 110 | rd | 0110011 | OR |
| 0000000 | rs2 | rs1 | 111 | rd | 0110011 | AND |

`ADD`和`SUB`分别执行加法和减法。溢出被忽略，并且结果的低XLEN位被写入目标寄存器`rd`。`SLT`和`SLTU`分别执行**符号数**和**无符号数**的比较，如果`rs1<rs2`，则将`1`写入`rd`，否则写入`0`。 注意，`SLTU rd,x0,rs2`，如果`rs2`不等于0（ 译者注：在RISC-V中，`x0`寄存器永远是0），则把1写入`rd`，否则将0写入`rd`（汇编语言伪指令`SNEZ rd,rs`）。`AND`、`OR`、`XOR`执行按位逻辑操作。`SLL`、`SRL`、`SRA`分别执行逻辑左移、逻辑右移、算术右移，被移位的操作数是寄存器`rs1`，移位次数是寄存器`rs2`的低5位。

### 1.2.3. NOP 指令

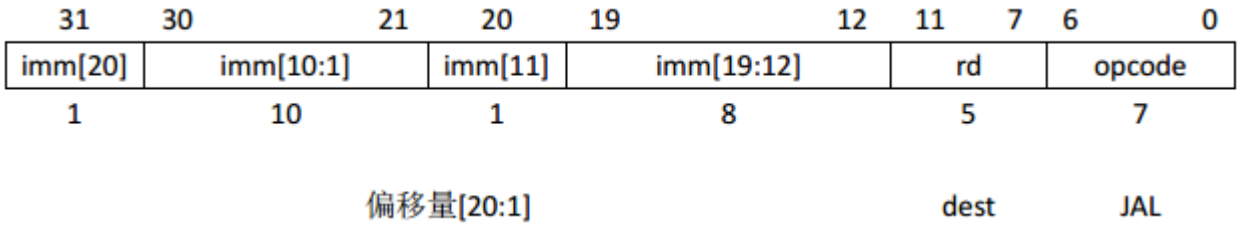



NOP指令并不改变任何用户可见的状态，除了使得pc向前推进。 NOP被编码为`ADDI x0,x0,0`。

## 1.3. 控制指令

包括无条件跳转指令和条件跳转指令

### 1.3.1 无条件跳转




| | | | | | |
|---|---|---|---|---|---|
| imm[20\|10:1\|11\|19:12] | | | rd | 1101111 | JAL |
| imm[11:0] | rs1 | 000 | rd | 1100111 | JALR |

跳转并连接（`JAL`）指令使用了UJ类格式，此处J立即数编码了一个2的倍数的有符号偏移量。这个偏移量被**符号扩展**，加到pc上，形成跳转目标地址，跳转范围因此达到±1MB。`JAL`将**跳转指令后面指令**的地址（`pc+4`）保存到寄存器`rd`中。标准软件调用约定使用`x1`来作为返回地址寄存器。 普通的无条件跳转指令（汇编语言伪指令J）被编码为`rd=x0`的`JAL`指令。（ 译者注： x0是只读寄存器，无法写入）

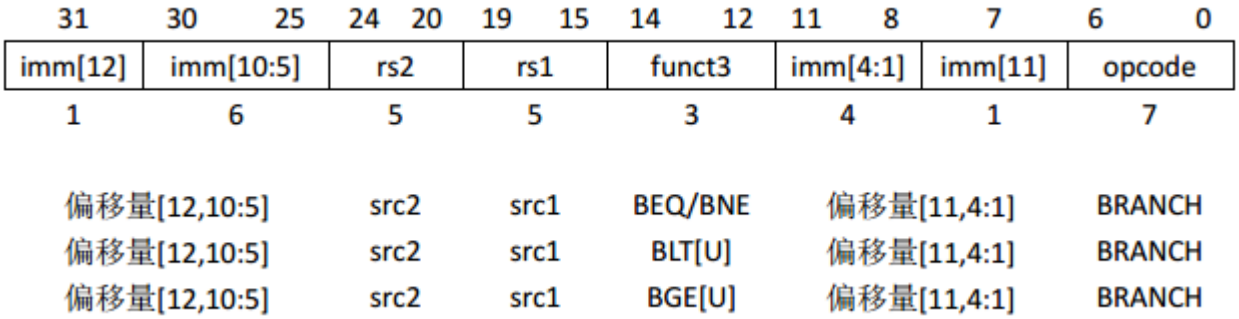
| 31 20 | 19 15 | 14 12 | 11 7 | 6 0 |
|---|---|---|---|---|
| imm[11:0] | rs1 | funct3 | rd | opcode |
| 12 | 5 | 3 | 5 | 7 |
| 偏移量[11:0] | 基址 | 0 | dest | JALR |

间接跳转指令`JALR`（ jump and link register ）使用I类编码。通过将12位**有符号**I类立即数加上`rs1`，然后将结果的最低位设置为0，作为目标地址。**跳转指令后面指令的地址**（ pc+4 ）保存到寄存器`rd`中。如果不需要结果，则可以把x0作为目标寄存器

JAL指令和JALR指令会产生一个非对齐指令取指异常， 如果目标地址没有对齐到4字节边界。

### 1.3.2. 条件分支

所有分支指令使用SB类指令格式。 12位B立即数编码了以2字节倍数的有符号偏移量，并被加到当前pc上，生成目标地址。条件分支范围是±4KB。



| 31 | 30 25 | 24 20 | 19 15 | 14 12 | 11 8 | 7 | 6 0 |
|---|---|---|---|---|---|---|---|
| imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode |
| 1 | 6 | 5 | 5 | 3 | 4 | 1 | 7 |
| 偏移量[12,10:5] | | src2 | src1 | BEQ/BNE | 偏移量[11,4:1] | | BRANCH |
| 偏移量[12,10:5] | | src2 | src1 | BLT[U] | 偏移量[11,4:1] | | BRANCH |
| 偏移量[12,10:5] | | src2 | src1 | BGE[U] | 偏移量[11,4:1] | | BRANCH |

| | | | | | | |
|---|---|---|---|---|---|---|
| imm[12\|10:5] | rs2 | rs1 | 000 | imm[4:1\|11] | 1100011 | BEQ |
| imm[12\|10:5] | rs2 | rs1 | 001 | imm[4:1\|11] | 1100011 | BNE |
| imm[12\|10:5] | rs2 | rs1 | 100 | imm[4:1\|11] | 1100011 | BLT |
| imm[12\|10:5] | rs2 | rs1 | 101 | imm[4:1\|11] | 1100011 | BGE |
| imm[12\|10:5] | rs2 | rs1 | 110 | imm[4:1\|11] | 1100011 | BLTU |
| imm[12\|10:5] | rs2 | rs1 | 111 | imm[4:1\|11] | 1100011 | BGEU |

分支指令比较两个寄存器。 `BEQ`和`BNE`将跳转，如果`rs1`和`rs2`相等或者不相等。 `BLT`和`BLTU`将跳转，如果`rs1`小于`rs2`，分别使用**有符号数**和**无符号数**进行比较。 `BGE`和`BGEU`将跳转，如果`rs1`大于等于`` `rs2 ``，分别使用**有符号数**和**无符号数**进行比较。注意， `BGT`、 `BGTU`、 `BLE`和`BLEU`可以通过将`BLT`、 `BLTU`、`BGE`、 `BGEU`的操作数对调来实现。

# 2. 实验原理

## 2.1. 总体设计

采用五级流水线模式，也就是

```
IF ==> ID ==> EX ==> MEM ==> WB
```

附加对数据相关和控制相关的处理，也就是采用数据转发，Stall和Flush.

## 2.2. 模块详细

### 2.2.1. ALU

输入为`Operand1 Operand2 AluContrl`输出为`AluOut`

基于不同的`AluContrl`做出不同的运算，比较简单的组合逻辑

需要注意的是，对于`SRA ADD SUB SLT`需要特别注明是有符号数`$signed()`

相应的代码如下：

```
always@(*)
begin
    case(AluContrl)
        `SLL: AluOut <= Operand1 << Operand2[4:0];
        `SRL: AluOut <= Operand1 >> Operand2[4:0];
        `SRA: AluOut <= $signed(Operand1) >>> Operand2[4:0];
        `ADD: AluOut <= $signed(Operand1) + $signed(Operand2);
        `SUB: AluOut <= $signed(Operand1) - $signed(Operand2);
        `XOR: AluOut <= Operand1 ^ Operand2;
        `OR:  AluOut <= Operand1 | Operand2;
        `AND: AluOut <= Operand1 & Operand2;
        `SLT: AluOut <= ($signed(Operand1) < $signed(Operand2)) ? 32'b1:32'b0;
        `SLTU: AluOut<= (Operand1 < Operand2) ? 32'b1:32'b0;
        `LUI: AluOut <= Operand2;//LUI的值已在imm上计算了，直接用，而AUIPC使用的是ADD
       default: AluOut <= 32'b0;
    endcase
end
```

### 2.2.2. BranchDecisionMaking

与`ALU`模块很类似，也是根据不同的Control信号来进行计算，然后输出不同的值。

这个模块其实可以和`ALU`合并，分离出来是为了架构更加清晰。

需要注意的是几个需要特别注明是有符号数的，`BLT, BGE`

相应的代码如下

```
always@(*)
    begin
        case(BranchTypeE)
            `NOBRANCH: BranchE <= 0;
            `BEQ: BranchE <= (Operand1 == Operand2);
            `BNE: BranchE <= (Operand1 != Operand2);
            `BLT: BranchE <= ($signed(Operand1) < $signed(Operand2));
            `BLTU: BranchE <= (Operand1 < Operand2);
            `BGE: BranchE <= ($signed(Operand1) >= $signed(Operand2));
            `BGEU:BranchE <= (Operand1 >= Operand2);
           default: BranchE <= 0;
        endcase
    end
```

### 2.2.3. ControlUnit

这是一个比较麻烦的模块，需要根据输入的`Op Fn3 Fn7`来判断很多的输出信号相应的值，但是简化的部分是，所有的输出信号都已经给出了，可以一个一个判断。所以这个模块分成两个阶段，首先根据`Op Fn3 Fn7`来判断是什么指令，然后对每个需要输出的信号依次分析：哪些指令需要这些信号以及相应的值。

首先是第一步，判断指令。可以通过下面的图来分析。

| Format | Instruction | Opcode | Funct3 | Funct6/7 |
|---|---|---|---|---|
| R-type | add | 0110011 | 000 | 0000000 |
| | sub | 0110011 | 000 | 0100000 |
| | sll | 0110011 | 001 | 0000000 |
| | xor | 0110011 | 100 | 0000000 |
| | srl | 0110011 | 101 | 0000000 |
| | sra | 0110011 | 101 | 0000000 |
| | or | 0110011 | 110 | 0000000 |
| | and | 0110011 | 111 | 0000000 |
| | lr.d | 0110011 | 011 | 0001000 |
| | sc.d | 0110011 | 011 | 0001100 |
| I-type | lb | 0000011 | 000 | n.a. |
| | lh | 0000011 | 001 | n.a. |
| | iw | 0000011 | 010 | n.a. |
| | id | 0000011 | 011 | n.a. |
| | ibu | 0000011 | 100 | n.a. |
| | ihu | 0000011 | 101 | n.a. |
| | iwu | 0000011 | 110 | n.a. |
| | addi | 0010011 | 000 | n.a. |
| | slli | 0010011 | 001 | 000000 |
| | xori | 0010011 | 100 | n.a. |
| | srli | 0010011 | 101 | 000000 |
| | srai | 0010011 | 101 | 010000 |
| | ori | 0010011 | 110 | n.a. |
| | andi | 0010011 | 111 | n.a. |
| | jalr | 1100111 | 000 | n.a. |
| S-type | sb | 0100011 | 000 | n.a. |
| | sh | 0100011 | 001 | n.a. |
| | sw | 0100011 | 010 | n.a. |
| | sd | 0100011 | 111 | n.a. |
| SB-type | beq | 1100111 | 000 | n.a. |
| | bne | 1100111 | 001 | n.a. |
| | blt | 1100111 | 100 | n.a. |
| | bge | 1100111 | 101 | n.a. |
| | bltu | 1100111 | 110 | n.a. |
| | bgeu | 1100111 | 111 | n.a. |
| U-type | lui | 0110111 | n.a. | n.a. |
| UJ-type | jal | 1101111 | n.a. | n.a. |

**FIGURE 2.18 RISC-V instruction encoding.**

基本的格式为

```
localparam RType_op=7'b0110011;
...
localparam Fn7_0=7'b0000000;
localparam Fn7_1=7'b0100000;

assign ADD = (Op == RType_op)&&(Fn3 == 3'b000)&&(Fn7 == Fn7_0);
assign SUB = (Op == RType_op)&&(Fn3 == 3'b000)&&(Fn7 == Fn7_1);
...
```

从技术上来说，只用依次判断即可，结构基本上类似。

第二步为输出的依次处理，下面详细说明

**`JalD JalrD`**

都是只依赖一个指令，表示相应的跳转，输出用于`NPC_Generator`对下一个PC的判断上。所以代码为

```
assign JalD=JAL;
assign JalrD=JALR;
```

**`RegWriteD`**

这个输出的信号有两个目的，其一是用在`DataExt`对从`mem`中读出的信号的扩展方式选择上，其二是在`WB`阶段判断是否需要写入寄存器，用作使能。

需要用到这个信号的指令除了Load(`LB LH LW LBU LHU`)还有`LUI AUIPC JAL JALR`以及R-Type和其他的I-Type.

相应的代码为

```
//其他要写入reg的
    wire OtherWriteReg;//除了load的几个
    assign OtherWriteReg=(LUI||AUIPC||JAL||JALR)|||(Op==IType_op)|||(Op==RType_op);

    always@(*)
    begin
        case ({LB,LH,LW,LBU,LHU,OtherWriteReg})
            6'b100000: RegWriteD <= `LB;
            6'b010000: RegWriteD <= `LH;
            6'b001000: RegWriteD <= `LW;
            6'b000100: RegWriteD <= `LBU;
            6'b000010: RegWriteD <= `LHU;
            6'b000001: RegWriteD <= `LW;
            default: RegWriteD <= `NOREGWRITE;
        endcase
    end
```

**`MemToReg`**

表示ID阶段的指令需要将data memory读取的值写入寄存器, 与上面的不同之处在于，此处只能是`mem`得到的数，用于在`WB`阶段对写入数据的判断上，`0`表示从`mem`中读出的数，`1`表示`ALU`计算或者其他地方的数。

那么只用判断是不是`Load`指令即可。

```
assign MemToRegD=(LB || LH || LW || LBU || LHU);
```

**`MemWriteD`**

采用独热码格式，对于data memory的`32bit`字按byte进行写入,`MemWriteD=0001`表示只写入最低1个byte

仅仅有`SB,SH,SW`有存储`mem`功能, 所以仅仅需要处理这几个指令就行了

```
always@(*)
    begin
        case ({SB,SH,SW})
            3'b100: MemWriteD <= 4'b0001;
            3'b010: MemWriteD <= 4'b0011;
            3'b001: MemWriteD <= 4'b1111;
            default: MemWriteD <= 4'b0000;
        endcase
    end
```

这里给出的信息只能判断是`B H W`其他的都判断不了，但是`SB`是按字节对齐，`SH`是按半字对齐，这部分还需要在Mem部分另外进行指定。

**`LoadNpcD`**

表示将NextPC输出到ResultM, 需要nextPC写入到RD的`JAL,JALR`

```
assign LoadNpcD=(JAL||JALR);
```

**`RegReadD`**

`RegReadD[1]==1`表示A1对应的寄存器值被使用到了，`RegReadD[0]==1`表示A2对应的寄存器值被使用到了，用于forward的处理

这个信号的处理需要格外仔细，具体的指令用到的信息在`Instr.xls`

```
assign RegReadD[1]=JALR|(Op==br_op)|(Op==load_op)|(Op==store_op)|(Op==IType_op)|
(Op==RType_op);
assign RegReadD[0]=(Op==br_op)|(Op==store_op)|(Op==RType_op);
```

**`BranchTypeD`**

表示不同的分支类型, 用于`BranchDecisionMaking`

形式比较清晰

```
always@(*)
    begin
        case ({BEQ,BNE,BLT,BLTU,BGE,BGEU})
            6'b100000: BranchTypeD <= `BEQ;
            6'b010000: BranchTypeD <= `BNE;
            6'b001000: BranchTypeD <= `BLT;
            6'b000100: BranchTypeD <= `BLTU;
            6'b000010: BranchTypeD <= `BGE;
            6'b000001: BranchTypeD <= `BGEU;
            default: BranchTypeD <= `NOBRANCH;
        endcase
    end
```

**`AluContrlD`**

表示不同的ALU计算功能

需要仔细分析的是，好几个指令用的是`ALU`的`ADD`形式，分别有`Load, Store`计算mem地址，以及另外的指令`ADD ADDI AUIPC JALR`

弄清除这一点后就可以直接写出来了

```
always@(*)
    begin
        if((Op==load_op)||(Op==store_op)||ADD||ADDI||AUIPC||JALR)begin AluContrlD <= `ADD; end
        else if(SUB)begin AluContrlD <= `SUB; end
        else if(LUI)begin AluContrlD <= `LUI; end
        else if(XOR||XORI)begin AluContrlD <= `XOR; end
        else if(OR||ORI)begin AluContrlD <= `OR;  end
        else if(AND||ANDI)begin AluContrlD <= `AND; end
        else if(SLL||SLLI)begin AluContrlD <= `SLL; end
        else if(SRL||SRLI)begin AluContrlD <= `SRL; end
        else if(SRA||SRAI)begin AluContrlD <= `SRA; end
        else if(SLT||SLTI)begin AluContrlD <= `SLT; end
        else if(SLTU||SLTIU)begin AluContrlD <= `SLTU; end
        else begin AluContrlD <= 4'dx; end
    end
```

**`AluSrc1D AluSrc2D`**

这两个信号的处理最麻烦，

AluSrc2D,表示Alu输入源2的选择; 00:Reg 01:Rs2 5bits 10:Imm

AluSrc1D,表示Alu输入源1的选择; 0:Reg 1:PC

对于源1, 仅仅只有AUIPC用到了PC而且需要ALU处理，所以

```
assign AluSrc1D = (AUIPC);
```

对于源2, 使用了Rs2的是几个立即数移位指令`SLLI SRLI SRAI`

使用了Reg的是branch和Rtype的指令，其他的不同AUL或者使用立即数的，都列为其他

```
assign AluSrc2D = (SLLI||SRLI||SRAI)? 2'b01 : ((Op==br_op||Op==RType_op)? 2'b00 :
2'b10);
```

**`ImmType`**

表示指令的立即数格式

因为指令本身就已经分成了`R I S B U J`几种格式，所以判断即可

```
always@(*)
    begin
        if(Op==RType_op)begin ImmType<=`RTYPE; end
        else if(Op==IType_op || Op==load_op || JALR) begin ImmType<=`ITYPE; end
        else if(Op==store_op)begin ImmType<=`STYPE; end
        else if(Op==br_op)begin ImmType<=`BTYPE; end
        else if(LUI||AUIPC)begin ImmType<=`UTYPE; end
        else if(JAL)begin ImmType<=`JTYPE; end
        else begin ImmType<=3'dx; end
    end
```

### 2.2.4. DataExt

目的是对mem读出的内容进行扩展，需要根据`RegWrite`(也就是Load指令的类型)来做合适的拓展，考虑到不同的对齐方式，还有一个字节的选择，`LoadedBytesSelect`

理解起来比较显然，`LB LH`是符号拓展，`LBU LHU`是零拓展。

基本模式为

```
always@(*)
    begin
        case (RegWriteW)
            `LB: begin
                case (LoadedBytesSelect)
                    2'b00: OUT <= {{25{IN[7]}},IN[6:0]};
                    2'b01: OUT <= {{25{IN[15]}},IN[14:8]};
                    2'b10: OUT <= {{25{IN[23]}},IN[22:16]};
                    2'b11: OUT <= {{25{IN[31]}},IN[30:24]};
                    default: OUT <= 32'bx;
                endcase
            end
            `LH: begin
                casex (LoadedBytesSelect)
                    2'b0x: OUT <= {{17{IN[15]}},IN[14:0]};
                    2'b1x: OUT <= {{17{IN[31]}},IN[30:16]};
                    default: OUT <= 32'bx;
                endcase
            end
            ...
```

```
            default: OUT <= 32'bx;
        endcase
    end
```

### 2.2.5. HazardUnit

这是一个很关键的模块，需要实现三部分，一是数据转发，二是无法转发的数据采用Stall, 三是跳转指令的Flush

首先是数据转发，reg读出的数据，从两个地方转发，`WB`和`Mem`，其中，从`Mem`转发的优先级要高于从`WB`, 需要判断的标准是，相关的阶段准备写入Reg, 但是还没写来得及，以及`src=dst`，`src!=0`，同时注意优先级。

```
//Forward Register Source 1
assign Forward1E[0]=(|RegWriteW)&&(RdW!=0)&&(!((RdM==Rs1E)&&(|RegWriteM)))&&
(RdW==Rs1E)&&RegReadE[1];
assign Forward1E[1]=(|RegWriteM)&&(RdM!=0)&&(RdM==Rs1E)&&RegReadE[1];

//Forward Register Source 2
assign Forward2E[0]=(|RegWriteW)&&(RdW!=0)&&(!((RdM==Rs2E)&&(|RegWriteM)))&&
(RdW==Rs2E)&&RegReadE[0];
assign Forward2E[1]=(|RegWriteM)&&(RdM!=0)&&(RdM==Rs2E)&&RegReadE[1];
```

然后是无法转发然后stall, 比如说`Load`之后马上进行`Store`.

判断的标准为

```
MemToRegE && ((RdE==Rs1D )||(RdE==Rs2D))&& RdE!=0
```

处理的办法是将`StallF StallD`设置为1

最后是控制相关，`Branch`和`Jalr`都是在`Ex`阶段被发现的，所以需要清除`FlushD FlushE`为1

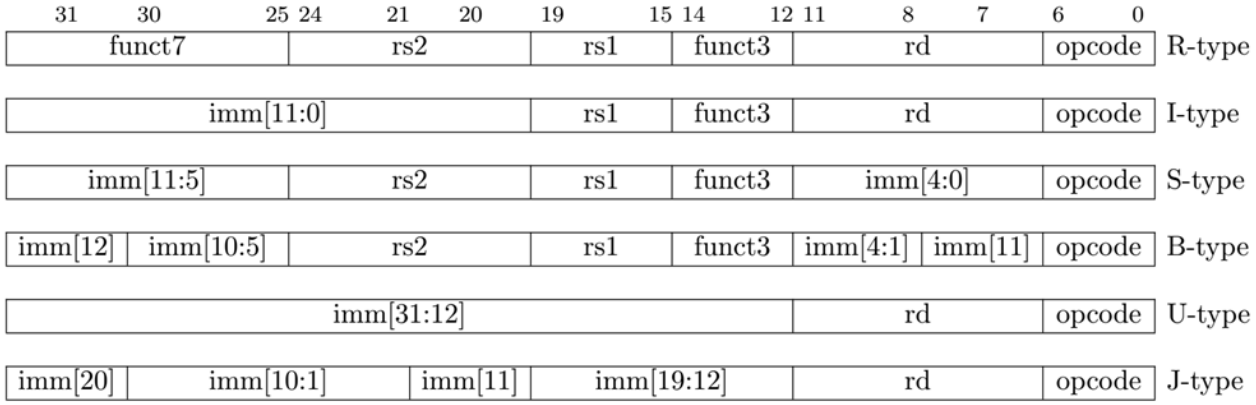
而`Jal`是在`ID`阶段被发现的，需要将`FlushD=1`即可。

需要另外注意的是，`Jal`由于流水线的层次浅，它的优先级小于`Jalr Branch`

### 2.2.6. ImmOperandUnit

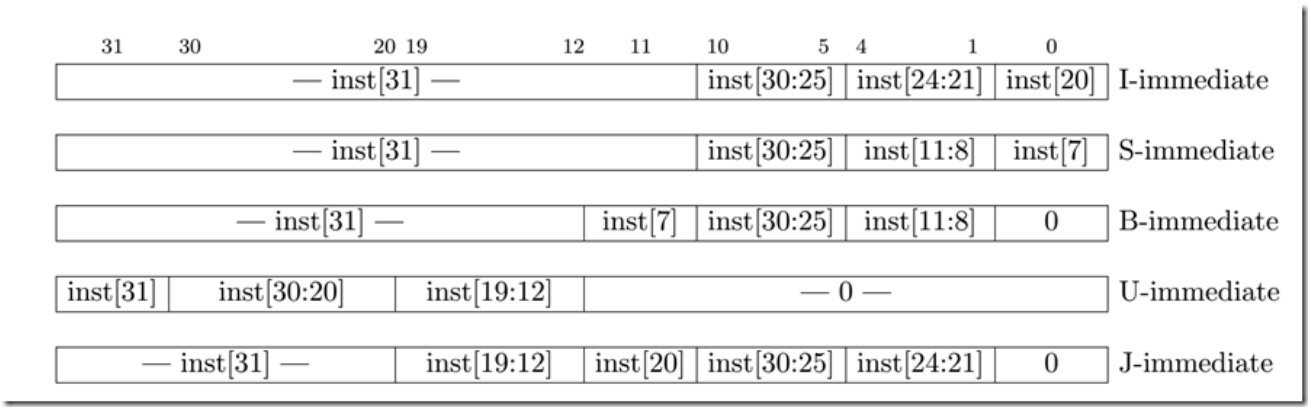
需要根据不同的Type (`R I S B U J`)来进行立即数的拓展，

首先，RV32I指令格式为




imm表示指令中的立即数，比如imm[11:0]，表示一个12位的立即数，它的高20位会符号位扩展，imm[31:12]表示一个32位的立即数，它的低12位会补0。

下图是各种指令格式扩展后的32位立即数。




根据这张图，就很容易写出来立即数的拓展方式了

```
always@(*)
    begin
        case(Type)
            `RTYPE: Out<=32'b0;
            `ITYPE: Out<={ {21{In[31]}}, In[30:20] };
            `STYPE: Out<={ {21{In[31]}}, In[30:25],In[11:7]};
            `BTYPE: Out<={ {20{In[31]}},In[7],In[30:25],In[11:8],1'b0};
            `UTYPE: Out<={ In[31:12],12'b0};
            `JTYPE: Out<={ {12{In[31]}},In[19:12],In[20],In[30:21],1'b0};
                        //请补全!!!
            default:Out<=32'hxxxxxxxx;
        endcase
    end
```

### 2.2.7. WBSegSeg

这部分最关键的是处理非字对齐store. 关键技术在于对 `WE` `A[1:0]` 的判断和分析。

需要另外设置 `WE_test` 重新规划 `WE` 的写入，比如 `SB` 分为 `0001 0010 0100 1000`，`SH` 分为 `0011 1100`，而 `SW` 只有 `1111`.

```
assign WE_test= (|WE)? ((WE==4'b0001)? (WE<<A[1:0]):((WE==4'b0011)? ((A[1]==1'b0)?
4'b0011:4'b1100):4'b1111)):WE;
```

需要另外设置`WD_test`重新规划数据的输入，因为仅仅是低位的数据，所以需要进行复制，如下

```
assign WD_test= (|WE)? ((WE==4'b0001)? ({WD[7:0],WD[7:0],WD[7:0],WD[7:0]}) :
((WE==4'b0011)? ({WD[15:0],WD[15:0]}) : WD)  ):WD;
```

### 2.2.8. NPC_Generator

流水线深的先执行

所以`Jalr=Br > Jal`（`Jalr`与`Br`不可能同时出现）

```
always@(*)
    begin
        if(JalrE)begin PC_In<=JalrTarget; end
        else if(BranchE)begin PC_In<=BranchTarget; end
        else if(JalD)begin PC_In<=JalTarget; end
        else begin PC_In<=PCF+4; end
    end
```

# 3. 问题回答

1. 为什么将DataMemory和InstructionMemory嵌入在段寄存器中？

   这样在时钟沿读出后直接进入下一个流水阶段，不用等下一个时钟周期。

2. DataMemory和InstructionMemory输入地址是字（32bit）地址，如何将访存地址转化为字地址输入进去？

   地址改成`A[31:2]`，而`A[1:0]`在DataExt使用选定特定的位。

3. 如何实现DataMemory的非字对齐的Load？

   将读出的32bits数据按照不同的Load指令格式进行选位与扩展。

4. 如何实现DataMemory的非字对齐的Store？

   WE使能表示使用不同的位写入，比如`0011`表示写入低16bits.

5. 为什么RegFile 的时钟要取反？

   相当于不采用同步读，就是比较方便五段流水，不用在内部转发

6. NPC_Generator中对于不同跳转target 的选择有没有优先级？

   执行越靠后越优先，`Br = Jalr > Jal`

7. ALU模块中，默认wire变量是有符号数还是无符号数？

   无符号数，可以使用`$signed()`

8. AluSrc1E执行哪些指令时等于1'b1？

   `AUIPC`

9. AluSrc2E执行哪些指令时等于2'b01？

`SLLI SRLI SRAI`

10. 哪条指令执行过程中会使得LoadNpcD==1？

    `jalr, jal`

11. DataExt模块中，LoadedBytesSelect的意义是什么？

    从取到的32bits选择相应的位

12. Harzard模块中，有哪几类冲突需要插入气泡？

    Load相关, 比如Load之后紧接着是Store相应的寄存器的值(或其他用到Reg的指令)

13. Harzard 模块中采用默认不跳转的策略，遇到branch 指令时，如何控制flush 和stall信号？

    `FlushD FlushE=1`其他都是0

14. Harzard模块中，RegReadE 信号有什么用？

    判断是否用到了Reg, 从而判断是否需要进行转发。

15. 0号寄存器值始终为 0，是否会对forward的处理产生影响？

    需要判断是否为`x0`,如果是则不需要转发

# Ref


RISC-V指令集手册（卷1-用户级指令集）-中文版

RV32I指令集

《Computer Organization and Design RISC-V edition》

CSE 564 Computer Architecture Summer 2017--Lecture 09: RISC-V Pipeline Implementation