

计算机组成原理(H) 实验

PB20111689 蓝俊玮

PB20111699 吴骏东

2022.5.25

一、实验题目

CPU 综合设计——基于 RISC-V 的迷宫游戏

二、实验目的

1. 理解计算机硬件系统的组成结构和工作原理；
2. 掌握软硬件综合系统的设计和调试方法。

三、实验平台

- Vivado2019
- Nexys4 DDR 开发板
- VGA 显示器

四、CPU 设计概述

本实验在 Lab5 所设计的流水线 CPU 上进行了进一步的改造。

4.1 CPU 设计概述

本实验设计的流水线 CPU 支持 RISC-V32I 的所有基本指令，同时还进行了 M 扩展和 B 扩展。总指令条数达到了 76 条。这些指令包括：

指令类型	指令名称 (RISC-V)
单周期运算指令	add、addi、and、andi、or、ori、sll、slli、slt、slti、sltu、sltiu、sra、srai、srl、srli、sub、xor、xori、andn、orn、max、maxu、min、minu、sh1add、sh2add、sh3add、xnor
乘除法指令	div、divu、mul、mulh、rem、remu
访存指令	lb、lbu、ld、lh、lhu、lw、sb、sd、sh、sw
跳转指令	beq、bge、bgeu、blt、bltu、bne、jal、jalr
位操作指令	bclr、bclri、bext、bexti、binv、binvi、clz、cpop、ctz、rol、roli、ror、rori
高位立即数操作指令	auipc、lui
CSR 系列指令	csrrc、csrrci、csrrs、csrrsi、csrrw、csrrwi、ebreak、ecall

指令的具体内容与含义请参考附件中的指令集手册与拓展指令说明文档。

基于此，我们的 CPU 可以使用如下的拓展指令（伪指令）：

beqz、bgez、bgt、bgtu、bgtz、ble、bleu、blez、bltz、bnez、csrr、csrrc、csrrci、csrrs、csrrsi、csrrw、csrrwi、j、jr、la、li、lla、mv、neg、nop、not、ret、seqz、sgtz、sltz、snez
这将大大丰富编写汇编程序时的选择，同时简化程序流程，提高运行效率。

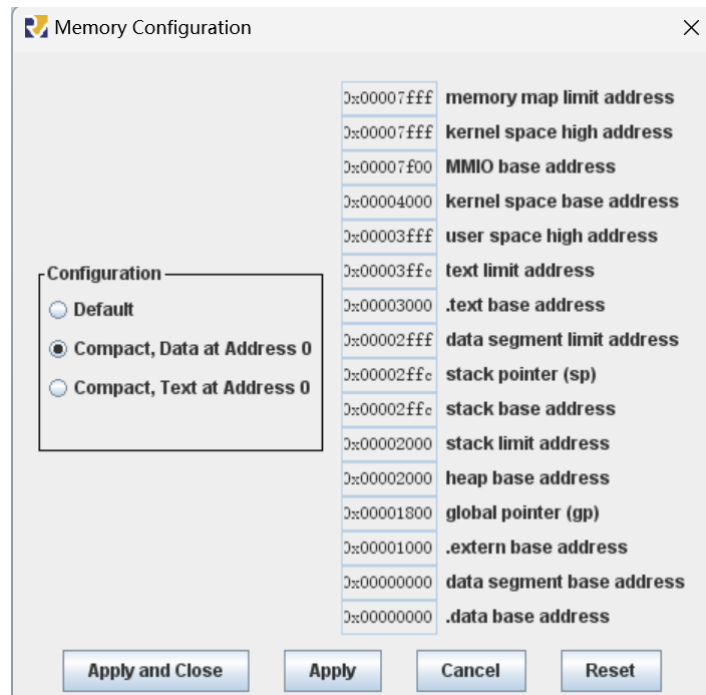
简而言之，本实验所设计的流水线 CPU 可以处理所支持指令的所有可能的数据冒险与控制冒险。与传统五级流水线 CPU 设计不同，本实验中的 CPU 做出了如下改动：

- 1. 将分支跳转模块前移至 ID 流水段，从而压缩分支预测可能导致的流水线停顿时长。分支预测采用全不跳转模式，预测失败后流水线只需要停顿一个时钟周期；
- 2. 调整综合运算单元 CCU，增加了位运算、乘除法运算核心，并针对乘除法指令的延迟设计了全新的解决方案；
- 3. 合并数据冒险控制与流水线停顿控制模块。本实验中 CPU 通过唯一的 Forward-Hazard 模块进行数据前递选择与流水线停顿控制，使得内部逻辑更为统一；
- 4. 增加 CSR 硬件实现，现在可以将 CSR 视作等效无限扩展的寄存器堆阵列，为未来的功能扩展奠定基础；
- 5. 增加流水线控制单元 PCU。CPU 的工作状态将完全由 PCU 控制，PCU 内部集成的状态机可以检测所有的异常信号并进行中断处理；
- 6. 重新设计并规划了存储结构，为各种可能的需要设计了对应的存储空间。

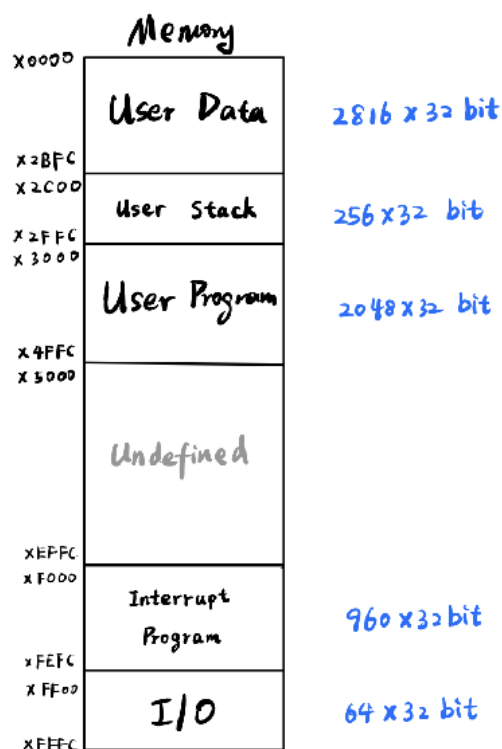
以上为本实验中 CPU 所做出的调整。接下来，我们将对流水线 CPU 内部构造进行逐一介绍与分析。

4.2 内存空间设计 (Memory Unit, MU)

4.2.1 整体规划



如上图所示，RISC-V 编译器与仿真器 RARS 在模拟运行前，会要求用户给出所模拟的 CPU 的内存结构。本实验参考了其中的紧凑型内存模式，设计了如下的内存规划：

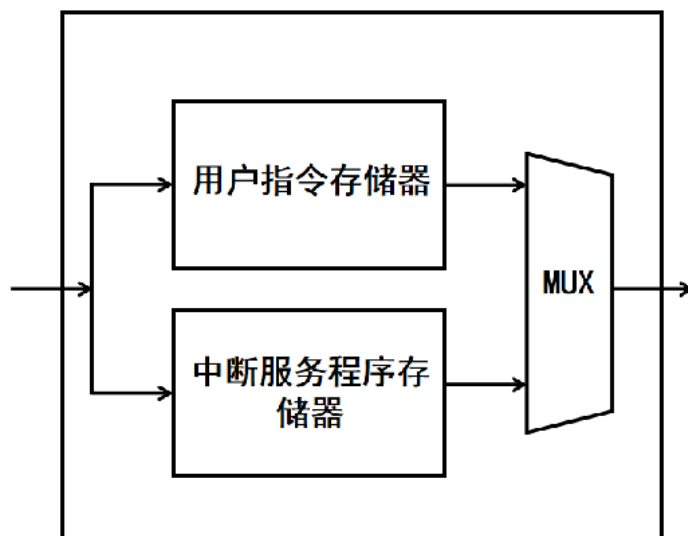


CPU 有效地址空间为 16 bit，共分为用户数据空间、用户栈、用户程序空间、系统中断服务程序空间、外设内存映射五大部分。各部分的实际长度可以根据需要进行更改，使用时只需要修改对应存储器大小与地址判断条件即可。

对于不同空间内的数据，我们进行了严格的访问限制。例如，如果用户试图往用户栈对应的地址内写入数据时，数据存储器单元会检测到访问越界并抛出异常。正常的程序运行过程中，PC 一定不会跳转到中断服务处理程序对应的地址。

4.2.2 指令存储器单元 IMU

CPU 中涉及到的指令包括用户指令与中断服务处理程序指令。后者是在 CPU 设计时就被写入存储器的固定程序。用户实际可以编写并运行的程序只有 0x3000~0x4FFC 之间的部分。为了防止冲突，我们对每一种空间分别设计了一个分布式存储器 IP 核。IMU 内部结构如下图所示：



为了实现不同存储器之间的数据切换，在读数据时，两个存储器均会根据 PC 对应计算出的物理地址获取内容，由 PC 的范围决定最终将哪一个结果输出。核心逻辑为：

```
always @(*) begin
    imu_dout = 32'b0;

    if (imu_addr >= 32'h3000 && imu_addr < 32'h4FFC) begin
        // user program
        imu_dout = im_dout;
    end
    else if (imu_addr >= 32'hF000 && imu_addr < 32'hFF00) begin
        // interrupt program
        imu_dout = interrupt_dout;
    end
end
```

如果发生了访问越界，则 IMU 为抛出访问异常信号。

```
always @(*) begin
    imu_error = 1'b0;
    if (imu_addr < 32'h3000 || imu_addr > 32'h4FFC && imu_addr < 32'hF000 ||
    imu_addr > 32'hFEFC) begin
        imu_error = 1'b1;
    end
end
```

4.2.3 数据存储器单元 DMU

数据存储器的设计与指令存储器类似。但由于其涉及到写操作，所以针对不同存储器，其对应的写使能信号需要额外生成。

```

assign user_data_we_flag = (dmu_addr < 32'h2c00) ? 1'b1 : 1'b0;
assign user_data_we = dmu_we & user_data_we_flag;

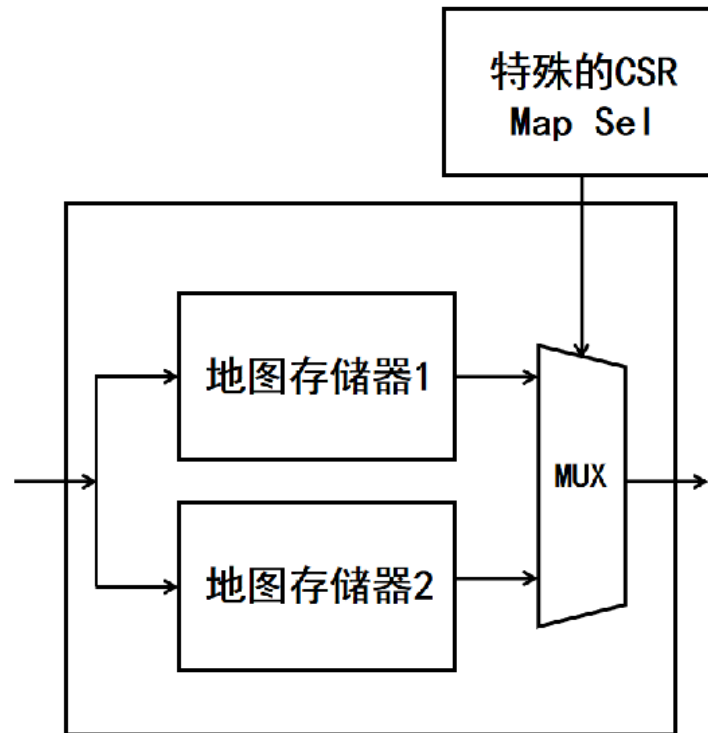
assign user_stack_we_flag = (dmu_addr < 32'h3000 && dmu_addr >= 32'h2c00) ? 1'b1
: 1'b0;
assign user_stack_we = dmu_we & user_stack_we_flag;

```

对应的数据输出采用与 IMU 相同的选择方式即可。

4.2.4 针对迷宫设计的专用数据存储器单元

由于迷宫游戏并没有涉及到栈指针操作，加上图像显示时会占用大量的内存空间。基于此我们对原先设计的数据存储器进行了改造。如下图所示：



地图存储器为双端口块式存储器，分别储存了不同的地图数据（RGB 数值）。实际启用的数据存储器由一个特定的寄存器（后续会介绍）的数值决定。任何时候有且仅有一个数据存储器执行读写操作，其他数据存储器均处于无效状态。

4.2.5 访存指令实现

为了实现 RISC-V 32I 的访存指令，在写入与读出数值之前，我们对其进行了二次处理操作。根据实际操作的数据模式选择相应的分割与重组方式。实现细节如下：

```

always @(*) begin
    // store data
    din = data_1;
    case (mode)
        BY_WORD: begin
            din = data_1;
        end
        BY_HALF: begin

```

```

        if (data_1[15] == 1'b1)
            din = {{16{1'b1}}, {data_1[15:0]}};
        else
            din = {{16{1'b0}}, {data_1[15:0]}};
        end

    BY_BYTE: begin
        if (data_1[7] == 1'b1)
            din = {{24{1'b1}}, {data_1[7:0]}};
        else
            din = {{24{1'b0}}, {data_1[7:0]}};
        end
    endcase
end

```

```

always @(*) begin
    // load data
    out_1 = dout;
    case (mode)
        BY_WORD: begin
            out_1 = dout;
        end

        BY_BYTE: begin
            if (dout[7] == 1'b1)
                out_1 = {{24{1'b1}}, {dout[7:0]}};
            else
                out_1 = {{24{1'b0}}, {dout[7:0]}};
            end

        BY_BYTE_U: begin
            out_1 = {{24'b0}, {dout[7:0]}};
        end

        BY_HALF: begin
            if (dout[15] == 1'b1)
                out_1 = {{16{1'b1}}, {dout[15:0]}};
            else
                out_1 = {{16{1'b0}}, {dout[15:0]}};
            end

        BY_HALF_U: begin
            out_1 = {{16'b0}, {dout[15:0]}};
        end
    endcase
end

```

4.3 寄存器堆 (Register File, RF)

在 Lab4 设计的寄存器堆文件的基础上，本实验进行了如下改动：

1. 读写顺序由读优先变回写优先。由于增加了段间寄存器，原先出现的瞬时循环写入的仿真 bug 已经被清除；

2. 增加新的读端口与输出端口，用作外部调试。

相关模块设计如下：

```
/*
    This is the register file module.
    The register file is the same as RISC-V.
    As a result, Reg[0] is forever 0, Reg[2] is initialized to 32'h2ffc, Reg[3]
    is initialized to 32'h1800.
    The other registers are initialized to 0.

    Register file is write-first.
    Synchronous writing number, asynchronous reading number.

    In order to make it easy to debug, RF specially provides the read-address-2
    and read-data-2.
    Remember: RF doesn't have the reset signal, which means you can not
    initialize them as you like while programming!
*/

module REG_FILE (
    input clk,                // clk
    input [4:0] ra0, ra1, ra2, // read address
    output [31:0] rd0, rd1, rd2, // read data output
    input [4:0] wa,           // write address
    input [31:0] wd,          // write data input
    input we                 // writing enable
);
    integer i;
    reg [31:0] rf [0: 31];    // regfile

    initial begin
        i = 0;
        while (i < 32) begin
            rf[i] = 32'b0;
            i = i + 1;
        end
        rf[2] = 'h2ffc;
        rf[3] = 'h1800;
    end

    assign rd0 = (ra0 == wa && ra0 && we) ? wd : rf[ra0]; // read
    assign rd1 = (ra1 == wa && ra1 && we) ? wd : rf[ra1]; // read
    assign rd2 = rf[ra2];

    always @(posedge clk)
        if (we && wa != 'b0)
            rf[wa] <= wd;    // write

endmodule
```

4.4 立即数符号扩展 (Sign Extend, SEXD)

本实验所支持的指令中，除了运算指令（R）外，其他所有指令均会涉及到立即数运算。根据其立即数扩展前操作的不同，我们将其划分为：12bit 立即数、20bit 立即数、12bit 高位立即数。其对应关系表如下：

```

The instruction used imm contains:

ArithmeticI, like addi, xori, ori, andi
    format: is[31:20] -> 12bit imm                // 12

Data Shift, like slli, srai, srli
    format: is[24:20] -> 5bit imm                <RV32I> // 12

Condition jump, like beq, bne, blt, bltu
    format: is[31] -> imm[12]
           is[30:25] -> imm[10:5]
           is[11:8] -> imm[4:1]
           is[7] -> imm[11]
           imm[0] = 0

           13bit imm                                // 12

Upper imm, like auipc, lui
    format: is[31:12] << 12 -> imm[31:0]          // <<

jal, jal
    format: is[31] -> imm[20]
           is[30:21] -> imm[10:1]
           is[20] -> imm[11]
           is[19:12] -> imm[19:12]
           imm[0] = 0

           21bit imm                                // 20

jalr, jalr
    format: is[31:20] -> 12 bit imm                // 12

sw:
    format: is[31:25] -> imm[11:5]                // 12
           is[11:7] -> imm[4:0]

lw:
    format: is[31:20] -> imm[11:0]

csr:
    format: is[31:20] -> imm[11:0]                // 12

```

根据上表，我们可以对应进行立即数计算。三种立即数在经过汇总后由 imm 总线输出。该模块的声明如下：

```

module Signextend
(
    input [31: 0] instruction,
    output reg [31: 0] imm
);

```


4.5 综合计算单元 (Calculate Unit, CCU)

综合计算单元是一个集成了基本运算、位运算、乘除法运算的计算模块。由于支持的计算模式庞杂，模块采用了 8-bit 位宽作为工作选择信号。以下是选择信号对照表：

```
/*
    Below is the CCU working mode table
*/
// Integer Arithmetic Logic Unit (Except MUL & DIV)
// Code begin with 8'h0 ~ 8'h2

// RISCv 32I
localparam SUB = 8'h00;
localparam ADD = 8'h01;
localparam AND = 8'h02;
localparam OR = 8'h03;
localparam XOR = 8'h04;
localparam RMV = 8'h05;    // Right shift (logic)
localparam LMV = 8'h06;    // Left shift (logic)
localparam ARMV = 8'h07;   // Right shift (arithmetic)
localparam SLTS = 8'h08;   // Sign less then set bit
localparam SLTUS = 8'h09;  // Unsign less then set bit

// RISCv 32B
localparam ANDN = 8'h10;    // Not then and
localparam MAX = 8'h11;
localparam MAXU = 8'h12;
localparam MIN = 8'h13;
localparam MINU = 8'h14;
localparam ORN = 8'h15;    // Not then or
localparam SH1ADD = 8'h16;
localparam SH2ADD = 8'h17;
localparam SH3ADD = 8'h18;
localparam XNOR = 8'h19;

// Integer Bit Calculate Unit
// Code begin with 8'h3

// RISCv 32B BALU
localparam BCLR = 8'h30;    // Clear single bit
localparam BEXT = 8'h31;    // Get single bit
localparam BINV = 8'h32;    // Not single bit
localparam BSET = 8'h33;    // Set single bit
localparam CLZ = 8'h34;     // Leading zeros count
localparam CPOP = 8'h35;    // Set bits count
localparam CTZ = 8'h36;     // Suffix zeros count
localparam ROL = 8'h37;     // High bits reverse
localparam ROR = 8'h38;     // Low bits reverse

// Integer Arithmetic Unit For MUL & DIV
// Code begin with 8'h4

// RISCv 32M MDU
localparam MUL = 8'h40;     // Multiply
localparam MULH = 8'h41;    // High bit multiply
localparam MULHSU = 8'h42;  // High bit sign - unsign multiply
localparam MULHU = 8'h43;   // High bit unsign multiply
```

```

localparam DIV = 8'h44;      // Divide
localparam DIVU = 8'h45;     // Unsigned Divide
localparam REM = 8'h46;      // Remind number
localparam REMU = 8'h47;     // Unsigned remide number

```

采用 8-bit 的好处就是模式编号可以不连续，采用 [7:4]-[3:0] 二级划分结构，更有利于编码与整理。

4.5.1 算术与逻辑单元 ALU

ALU 中会进行大部分都基础运算过程，其内部集成了大小比较单元，可以执行相应的比较操作。

```

assign equal = (num1 == num2 ? 1 : 0);
assign sign_lessthan = ((num1[31] == 1 && num2[31] == 0) ||
                        (num1[31] == num2[31] && num1 < num2)) ? 1 : 0;
assign unsign_lessthan = ((num1[31] == 0 && num2[31] == 1) ||
                          (num1[31] == num2[31] && num1 < num2)) ? 1 : 0;

```

对于一般的运算，我们采用了 Verilog 的运算符进行计算。

```

case(mode_sel)
  SUB: begin
    ans = num1 - num2;
  end

  ADD: begin
    ans = num1 + num2;
  end

  AND: begin
    ans = num1 & num2;
  end

  OR: begin
    ans = num1 | num2;
  end

  XOR: begin
    ans = num1 ^ num2;
  end
  .....
endcase

```

对于移位操作，我们进行了如下的特别处理：

```

case(mode_sel)
  .....
  RMV: begin      // Right Move (Logical)
    if (num2 >= 32) begin
      ans = {32{1'b0}};
    end
    else begin
      ans = num1 >> num2;
    end
  end

```

```

end

LMV: begin
    if (num2 >= 32) begin
        ans = {32{1'b0}};
    end
    else begin
        ans = num1 << num2;
    end
end

ARMV: begin    // Right Move (Arithmetic)
    temp = 32'b0;
    counter = num1[31];
    if (num2 >= 32) begin
        if (counter == 0)
            ans = {32{1'b0}};
        else
            ans = {32{1'b1}};
    end
    else begin
        temp = num1 >> num2;
        if (counter == 1)
            ans = temp | ({32{1'b1}} << num2);
        else begin
            ans = temp | ({32{1'b0}} << num2);
        end
    end
end

SLTS: begin    // Set-bit when signed less than
    if (sign_lessthan)
        ans = 32'b1;
    else
        ans = 32'b0;
end

SLTUS: begin
    if (unsign_lessthan)
        ans = 32'b1;
    else
        ans = 32'b0;
end
.....
endcase

```

对于 B 扩展的基础操作，我们采用了组合的方式进行处理：

```

case (mode_sel)
    .....
    ANDN: begin
        ans = num1 & (~num2);
    end

    MAX: begin
        if (sign_lessthan)
            ans = num2;
        else
            ans = num1;
        end
    end
endcase

```

```

        else
            ans = num1;
        end

MAXU: begin
    if (unsign_lessthan)
        ans = num2;
    else
        ans = num1;
    end

MIN: begin
    if (sign_lessthan)
        ans = num1;
    else
        ans = num2;
    end

MINU: begin
    if (unsign_lessthan)
        ans = num1;
    else
        ans = num2;
    end

ORN: begin
    ans = num1 | (~num2);
end

SH1ADD: begin
    ans = (num1 << 1) + num2;
end

SH2ADD: begin
    ans = (num1 << 2) + num2;
end

SH3ADD: begin
    ans = (num1 << 3) + num2;
end
endcase

```

4.5.2 位操作单元 BALU

位操作单元会对寄存器进行一些特殊的位操作。首先，我们做出如下的初始化：

```

ans = 0;
mask = (32'b1) << num2[4:0];

```

对于单比特操作指令，我们通过位运算 + 逻辑操作的形式进行计算。

```

case(mode_sel)
    .....

```

```

BCLR: begin
    ans = num1 & (~mask);
end

BEXT: begin
    ans = (num1 >> num2[4:0]) & (32'b1);
end

BINV: begin
    ans = num1 ^ mask;
end

BSET: begin
    ans = num1 | mask;
end

ROL: begin
    ans = (num1 << num2[4:0]) | (num1 >> (32'd32 - num2[4:0]));
end

ROR: begin
    ans = (num1 >> num2[4:0] | num1 << (32'd32 - num2[4:0]));
end
.....
endcase

```

对于三条特殊的位计数指令，我们设计了专门的算法进行处理。核心思路是二分查找 + 并行搜索。实现细节如下：

```

case(mode_sel)
CLZ: begin // 前导0计数
    if (num1 >> 16 == 0)
        if (num1 >> 8 == 0)
            if (num1 >> 4 == 0)
                if (num1 >> 2 == 0)
                    if (num1 >> 1 == 0)
                        if (num1 == 0)
                            ans = 32'd32;
                    else
                        ans = 32'd31;
                else
                    ans = 32'd30;
            else
                if (num1 >> 3 == 0)
                    ans = 32'd29;
                else
                    ans = 32'd28;
            else
                if (num1 >> 6 == 0)
                    if (num1 >> 5 == 0)
                        ans = 32'd27;
                    else
                        ans = 32'd26;
                else
                    if (num1 >> 7 == 0)
                        ans = 32'd25;
                    else

```

```
    ans = 32'd24;
else
    if (num1 >> 12 == 0)
        if (num1 >> 10 == 0)
            if (num1 >> 9 == 0)
                ans = 32'd23;
else
    ans = 32'd22;
else
    if (num1 >> 11 == 0)
        ans = 32'd21;
else
    ans = 32'd20;
else
    if (num1 >> 14 == 0)
        if (num1 >> 13 == 0)
            ans = 32'd19;
else
    ans = 32'd18;
else
    if (num1 >> 15 == 0)
        ans = 32'd17;
else
    ans = 32'd16;
else
    if (num1 >> 24 == 0)
        if (num1 >> 20 == 0)
            if (num1 >> 18 == 0)
                if (num1 >> 17 == 0)
                    ans = 32'd15;
else
    ans = 32'd14;
else
    if (num1 >> 19 == 0)
        ans = 32'd13;
else
    ans = 32'd12;
else
    if (num1 >> 22 == 0)
        if (num1 >> 21 == 0)
            ans = 32'd11;
else
    ans = 32'd10;
else
    if (num1 >> 23 == 0)
        ans = 32'd9;
else
    ans = 32'd8;
else
    if (num1 >> 28 == 0)
        if (num1 >> 26 == 0)
            if (num1 > 25 == 0)
                ans = 32'd7;
else
    ans = 32'd6;
else
    if (num1 >> 27 == 0)
        ans = 32'd5;
```

```

else
    ans = 32'd4;
else
    if (num1 >> 30 == 0)
        if (num1 >> 29 == 0)
            ans = 32'd3;
        else
            ans = 32'd2;
        else
            if (num1 >> 31 == 0)
                ans = 32'd1;
            else
                ans = 32'd0;
        end
    end

CPOP: begin        // 1 计数
    ans = add_41 + add_42;
end

CTZ: begin        // 后缀0计数
    temp = num1 & (-num1);
    case(temp)
        32'h0: ans = 32'd32;
        32'h1: ans = 32'd0;
        32'h2: ans = 32'd1;
        32'h4: ans = 32'd2;
        32'h8: ans = 32'd3;
        32'h10: ans = 32'd4;
        32'h20: ans = 32'd5;
        32'h40: ans = 32'd6;
        32'h80: ans = 32'd7;
        32'h100: ans = 32'd8;
        32'h200: ans = 32'd9;
        32'h400: ans = 32'd10;
        32'h800: ans = 32'd11;
        32'h1000: ans = 32'd12;
        32'h2000: ans = 32'd13;
        32'h4000: ans = 32'd14;
        32'h8000: ans = 32'd15;
        32'h10000: ans = 32'd16;
        32'h20000: ans = 32'd17;
        32'h40000: ans = 32'd18;
        32'h80000: ans = 32'd19;
        32'h100000: ans = 32'd20;
        32'h200000: ans = 32'd21;
        32'h400000: ans = 32'd22;
        32'h800000: ans = 32'd23;
        32'h1000000: ans = 32'd24;
        32'h2000000: ans = 32'd25;
        32'h4000000: ans = 32'd26;
        32'h8000000: ans = 32'd27;
        32'h10000000: ans = 32'd28;
        32'h20000000: ans = 32'd29;
        32'h40000000: ans = 32'd30;
        32'h80000000: ans = 32'd31;
    endcase
endcase

```

```

assign add_11 = num1[1] + num1[0];
assign add_12 = num1[3] + num1[2];
assign add_13 = num1[5] + num1[4];
assign add_14 = num1[7] + num1[6];
assign add_15 = num1[9] + num1[8];
assign add_16 = num1[11] + num1[10];
assign add_17 = num1[13] + num1[12];
assign add_18 = num1[15] + num1[14];
assign add_19 = num1[17] + num1[16];
assign add_110 = num1[19] + num1[18];
assign add_111 = num1[21] + num1[20];
assign add_112 = num1[23] + num1[22];
assign add_113 = num1[25] + num1[24];
assign add_114 = num1[27] + num1[26];
assign add_115 = num1[29] + num1[28];
assign add_116 = num1[31] + num1[30];

assign add_21 = add_11 + add_12;
assign add_22 = add_13 + add_14;
assign add_23 = add_15 + add_16;
assign add_24 = add_17 + add_18;
assign add_25 = add_19 + add_110;
assign add_26 = add_111 + add_112;
assign add_27 = add_113 + add_114;
assign add_28 = add_115 + add_116;

assign add_31 = add_21 + add_22;
assign add_32 = add_23 + add_24;
assign add_33 = add_25 + add_26;
assign add_34 = add_27 + add_28;

assign add_41 = add_31 + add_32;
assign add_42 = add_33 + add_34;

```

4.5.3 乘除法单元 MDU

乘除法单元内集成了 Vivado 提供的三种 IP 核。通过这些 IP 核，我们可以将乘除法运算压缩在一个时钟周期延迟完成。对于 IP 核使用，我们遵循了：

- 乘法指令的结果为 64 bit，经拆分后分为高 32 bit 和低 32 bit 进行输出；
- 除法指令包括整除结果与余数，经过符号选择后进行输出。

所有的乘除法运算都是同步执行的，只是根据不同的工作信号选择不同的结果输出。实现的核心如下：

```

always @(*) begin
    sign_mode = SIGNED;
    error = NO_ERROR;
    mdu_mux_sel = 3'h7;    // zero
    case (mode)
        MUL: begin
            mdu_mux_sel = 3'h0;
        end

        MULH: begin
            mdu_mux_sel = 3'h1;
        end
    endcase
end

```



```

end

DIV: begin
    if (num2 == 0)
        error = DIVID_BY_ZERO;
        mdu_mux_sel = 3'h2;
    end

DIVU: begin
    if (num2 == 0)
        error = DIVID_BY_ZERO;
        mdu_mux_sel = 3'h2;
        sign_mode = UNSIGNED;
    end

REM: begin
    if (num2 == 0)
        error = DIVID_BY_ZERO;
        mdu_mux_sel = 3'h3;
    end

REMU: begin
    if (num2 == 0)
        error = DIVID_BY_ZERO;
        mdu_mux_sel = 3'h3;
        sign_mode = UNSIGNED;
    end

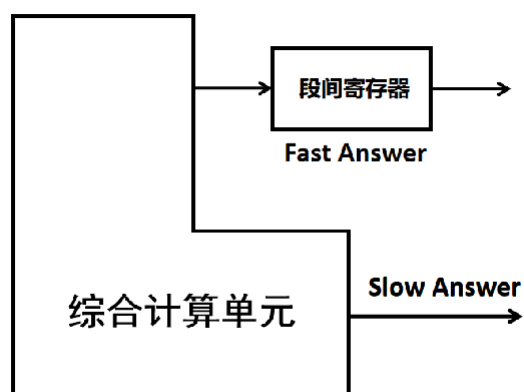
default: begin
    error = MODE_ERROR;          // mul mode error(this is not M
instruction)
end
endcase
end

```

除零检测是 CPU 的中断内容之一，其错误信号会在被除数为 0 时发出。

4.5.4 结果选择与同步

由于乘法指令不是单周期指令，为了统一其运算流程，我们将 MDU 的输出结果放在了 MEM 段。基于此，CCU 的完整结构如下：



非乘除法运算指令以 Fast Answer 输出，经过 EX-MEM 段间寄存器延迟后来到 MEM 段，与直接输出的 Slow Answer（乘除法结果）进行结果选择。这样就保证了任何计算指令在 MEM 段都可以得到正确结果。然而，这样处理会带来额外的相关问题。对于乘除法指令后的数据相关，流水线需要额外的停顿；相邻的乘除法指令也必须分开进行计算。

4.6 段间寄存器（Inter-segment Register, ISR）

流水线数据通路中共有四个段间寄存器：IF/ID、ID/EX、EX/MEM、MEM/WB。本实验中段间寄存器采用了整体声明、内部分散的设计方式，即在段间寄存器内部例化多个独立的寄存器作为数据缓存，并为其分配独立的数据借口。各段间寄存器的内容包括：

```
/*
  This is the inter-segment register between IF and ID
  Include the regs below:
    *PC: current PC
    *IS: current instruction
*/
```

```
/*
  This is the inter-segment register between ID and EX
  Include the regs below:
    *PC: current PC
    *IS: current instruction
    *Imm: the immediate number

    *Ctrl-WB: the control signals for WB
              (0000 + rfmux(3), rffwe(1))
    *Ctrl-MEM: the control signals for MEM
              (00 + dmu_mode(3) ccu_ans_mux(1) + dmwe(1), dmr(1))
    *Ctrl-EX: the control signals for EX
              (0 + ebreak(1) + sr1mux(3), sr2mux(3), alumode(8))

    *SR1: the source number A
    *SR2: the source number B
    *CSR: the CSR from ID
    *DR: the dr

    *CCU-EX: the ccu answer from EX
    *CCU-MEM: the ccu answer from MEM
    *CCU-WB: the ccu answer from WB
    *NPC-MEM: the pc+4 from MEM
    *DM-MEM: the dm-out from MEM
    *MUX-SEL: the mux control signals
              (0000 + csr(3) + sr1(3), sr2(3), bsr1(3), bsr2(3), dsr2(3), npc(2))
*/
```

```
/*
  This is the inter-segment register between EX and MEM
  Include the regs below:
    *PC: current PC
    *IS: current instruction
*/
```

```

    *Ctr-WB: the control signals for WB
        (0000 + rfmux(3), rffwe(1))
    *Ctrl-MEM: the control signals for MEM
        (00 + dmu_mode(3) ccu_ans_mux(1) + dmwe(1), dmrdr(1))

    *CSR: the CSR from EX
    *CCU-ANS: the ccu fast answer
    *DR: the dr
    *DM-ADDR: the address for dm unit
    *DM-DATA: the data for dm unit

*/

```

```

/*
This is the inter-segment register between MEM and WB
Include the regs below:
    *PC: current PC
    *IS: current instruction
    *Ctr-WB: the control signals for WB
        (0000 + rfmux(3), rffwe(1))
    *CCU-ANS: the ccu answer
    *MDR: the data from dmu
    *CSR: the CSR from MEM
    *DR: the dr

*/

```

4.7 指令解码器 (Decode Unit, DU)

指令解码器主要负责根据当前指令状态进行控制信号的发出与更改。本实验中 DU 模块涉及到的信号根据功能分类包括：

```
control_signals - 35 bit

MUX Ctrl signal:
    control_signals[2:0] - alu-sr1mux (3)
    control_signals[5:3] - alu-sr2mux (3)
    control_signals[8:6] - rfmux (3)
    control_signals[9] - rf-sr1mux (1)
    control_signals[10] - rf-sr2mux (1)
    control_signals[11] - ccu-ansmux (1)

CCU mode signal:
    control_signals[21:14] - alumode (8)

Regfile writing enable:
    control_signals[22] - rfi_we (1)

Memory working mode:
    control_signals[26:24] - dmu_mode (3)

Data memory unit reading and writing enable:
    control_signals[27] - dm_we (1)
    control_signals[28] - dmu_rd (1)

CSR write enable
    control_signals[29] = csr_wen (1)

B & J control signal:
    control_signals[33:30] - jump_ctrl (4)

Interrupt signal:
    control_signals[34] - ebreak (1)
```

所有控制信号可归类总结为：选择器选择信号、综合运算单元模式、寄存器堆写使能、数据存储器读写使能、数据访存模式、CSR 寄存器阵列写使能、跳转模式、中断指令检测。

所有的控制信号初始值设置如下：

```
ccu_mode = ADD;
ccu_ans_mux_sel = 1'b0;
dmu_mode = BY_WORD;
ebreak = 1'b0;
csr_we = 1'b0;
error = 1'b0;

sr1_mux_sel = SR1_ZERO;
sr2_mux_sel = SR2_ZERO;
rf_wb_mux_sel = RF_ZERO;

rfi_we = 1'b0;
dm_we = 1'b0;
dm_rd = 1'b0;
```

```
jump_ctrl = NPC;
```

由于指令数目过于庞大，此处不再列出每一条指令对应的控制信号内容。

4.8 跳转判断单元 (Branch Control Unit, BCU)

在解码器的基础上，针对跳转指令，我们设计了一个特别判断模块，用来发出对 `npc_mux` 的控制信号与跳转地址的计算。模块声明如下：

```
/*
===== Branch_CTRL module
=====
Author:      wintermelon
Last Edit:   2022.5.5

This is the branch control unit
Add the function of comparing 2 numbers
Add the function of add 2 numbers
*/

module Branch_CTRL(
    input [3:0] branch_sel,
    input [31:0] sr1, sr2,
    input [31:0] imm,
    input [31:0] pc,
    output reg [1:0] npc_mux_sel,
    output reg [31:0] pc_offset, reg_offset
);
```

与 Lab4 相比，本实验中 BCU 模块增加了数值比较与加法功能。这样就可以直接获得两个源操作数的大小关系并算出跳转地址，进而在 ID 段做出跳转判断。

跳转地址的产生由立即数 + PC 或 SR1 + PC 计算产生，两条数据会被同时送入 PC 选择器中，根据 `npc_mux_sel` 进行后续的判断。

```
always @(*) begin
    pc_offset = pc + imm;
    reg_offset = sr1 + imm;
end
```

跳转逻辑采用了基本判断准则。模块内部会根据 SR1、SR2 的数值生成三条信号：等于、小于、有符号小于。其他所有的条件跳转均根据这三条信号进行处理。

```
assign equal = (sr1 == sr2 ? 1 : 0);
assign sign_less_than = ((sr1[31] == 1 && sr2[31] == 0) ||
    (sr1[31] == sr2[31] && sr1 < sr2)) ? 1 : 0;
assign unsign_less_than = ((sr1[31] == 0 && sr2[31] == 1) ||
    (sr1[31] == sr2[31] && sr1 < sr2)) ? 1 : 0;
```

不同条件跳转的逻辑实现为：

```
case (branch_sel)
```

```

NPC: npc_mux_sel = PLUS4;
OFFPC: npc_mux_sel = PC_OFFSET;

EQ: begin // equal
    if (equal)
        npc_mux_sel = PC_OFFSET;
    else
        npc_mux_sel = PLUS4;
end

NEQ: begin // not equal
    if (~equal)
        npc_mux_sel = PC_OFFSET;
    else
        npc_mux_sel = PLUS4;
end

SLT: begin // sign less than
    if (sign_less_than)
        npc_mux_sel = PC_OFFSET;
    else
        npc_mux_sel = PLUS4;
end

ULT: begin // unsign less than
    if (unsign_less_than)
        npc_mux_sel = PC_OFFSET;
    else
        npc_mux_sel = PLUS4;
end

SGT: begin // sign greater than
    if (~sign_less_than && ~equal)
        npc_mux_sel = PC_OFFSET;
    else
        npc_mux_sel = PLUS4;
end

UGT: begin // unsign greater than
    if (~unsign_less_than && ~equal)
        npc_mux_sel = PC_OFFSET;
    else
        npc_mux_sel = PLUS4;
end

JALR: begin // jump and link R
    npc_mux_sel = REG_OFFSET;
end

default: begin
    npc_mux_sel = PLUS4;
end
endcase

```

4.9 程序状态寄存器阵列（CSR Unit）

4.9.1 物理 CSR 与 虚拟 CSR

为了更好地支持中断与 CSR 指令，我们采用物理结构实现了 CSR。本实验中设计的 CSR 包括：

Mtevc	Machine Trp-Vecor Base-Address Register 存储中断服务程序基址 编号 0x305	Mipd	Machine Interrupt Program Done 标记中断/异常处理完成 编号 0x100
Mcause	Machine CauseRegister 存储中断/异常原因代码 编号 0x342	Bs	Button Status 存储按钮信息 编号 0x000
Mepc	Machine Exception Program Counter 存储中断返回地址 编号 0x341	Map	Map choice 控制当前屏幕输出内容（启用的地图存储器） 编号 0x001
Mtval	Machine Trap Value Resister 存储中断相关数据 编号 0x343		

CSR 模块为每一个 CSR 都设计了读写端口。在迷宫程序中，我们将 CSR 的写使能信号全都置为 1。在模块外部，我们设计了对应的虚拟 CSR（reg 变量）。这些变量的赋值操作为：

```
always @(*) begin
    mtevc = mtevc_dout;
    mcause = mcause_dout;
    mepc = mepc_dout;
    mtval = mtval_dout;
    mipd = mipd_dout;
    bs = bs_dout;
    map = map_dout;
    led = led_dout;

    if (button_sig) begin
        // User press the button
        mtevc = 32'hF010;
        mcause = User_Button;
        mepc = (id_ex_clear && if_id_wen) ? if_pc : id_pc;
        // none:0 up:1 down:2 left:3 right:4 reset(mid): 5
        if (butu)
            bs = 32'h1;
        else if (butd)
            bs = 32'h2;
        else if (butl)
            bs = 32'h3;
        else if (butr)
            bs = 32'h4;
        else if (butc)
            bs = 32'h5;
        mipd = 1'b0;
    end
    else if (ebreak) begin
        // Progrom starts at 0xF000
        mtevc = 32'hF000;
        mcause = Program_Breakpoint;
        mepc = id_pc;
        mtval = 32'h0;           // no working infomation
        mipd = 1'b0;
    end
end
```

```

end
else if (csr_wen) begin
    // Program CSR instruction write
    case (csr_wadd)
        32'h0305: mtevc = csr_din;
        32'h0342: mcause = csr_din;
        32'h0341: mepc = csr_din;
        32'h0343: mtval = csr_din;
        32'h0100: mipd = csr_din;
        32'h0000: bs = csr_din;
        32'h0001: map = csr_din;
        32'h0002: led = csr_din;
    endcase
end
.....
end

```

默认情况下，虚拟 CSR 与物理 CSR 的输出直接相连，相当于每个时钟周期内对 CSR 进行数值更新。当发生中断时，硬件会更改虚拟 CSR 的数值；当外部需要更改 CSR 的数值时，软件也可以修改对应的虚拟 CSR。这样的设计可以在改变虚拟 CSR 数值后的下一个时钟周期更新物理 CSR 的数值，实现了硬件与软件的联合操作。

```

CSR_UNIT csr(
    .csr_we(1'b1),
    .csr_clk(clk),
    .rstn(rstn),

    // CSR 与 reg 变量的交互
    .mtevc_din(mtevc),
    .mtevc_dout(mtevc_dout),

    .mcause_din(mcause),
    .mcause_dout(mcause_dout),

    .mepc_din(mepc),
    .mepc_dout(mepc_dout),

    .mtval_din(mtval),
    .mtval_dout(mtval_dout),

    .mipd_din(mipd),
    .mipd_dout(mipd_dout),

    .bs_din(bs),
    .bs_dout(bs_dout),

    .map_din(map),
    .map_dout(map_dout),

    .led_din(led),
    .led_dout(led_dout),

    .csr_debug_addr(csr_debug_addr),
    .csr_debug_dout(csr_debug_data)
);

```


为了支持汇编程序访问 CSR，我们设计了一读一写双地址端口。在 ID 段，CPU 像读取寄存器堆一样读取 CSR 的数值；在 WB 段，CPU 像写回寄存器堆一样修改 CSR 的数值。读写操作互不干扰，实现了汇编程序对 CSR 阵列的高效访存。

```
// CSR 读写端口
input [31:0] csr_din,
output reg [31:0] csr_dout,
input [31:0] csr_radd,
input [31:0] csr_wadd,
input csr_wen,

// CSR read
always @(*) begin
    csr_dout = 32'h0;
    case (csr_radd)
        32'h0305: csr_dout = mtevc_dout;
        32'h0342: csr_dout = mcause_dout;
        32'h0341: csr_dout = mepc_dout;
        32'h0343: csr_dout = mtval_dout;
        32'h0100: csr_dout = mipd_dout;
        32'h0000: csr_dout = bs_dout;
        32'h0001: csr_dout = map_dout;
        32'h0002: csr_dout = led_dout;
    endcase
end
```

4.9.2 CSR 指令支持

RISC-V 32I 中的 CSR 指令主要包括：

指令	操作
csrrc	$t = \text{CSRs}[\text{csr}]; \text{CSRs}[\text{csr}] = t \& \sim x[\text{rs1}]; x[\text{rd}] = t$
csrrci	$t = \text{CSRs}[\text{csr}]; \text{CSRs}[\text{csr}] = t \& \sim \text{zimm}; x[\text{rd}] = t$
csrrs	$t = \text{CSRs}[\text{csr}]; \text{CSRs}[\text{csr}] = t \mid x[\text{rs1}]; x[\text{rd}] = t$
csrrsi	$t = \text{CSRs}[\text{csr}]; \text{CSRs}[\text{csr}] = t \mid \text{zimm}; x[\text{rd}] = t$
csrrw	$t = \text{CSRs}[\text{csr}]; \text{CSRs}[\text{csr}] = x[\text{rs1}]; x[\text{rd}] = t$
csrrwi	$t = \text{CSRs}[\text{csr}]; \text{CSRs}[\text{csr}] = \text{zimm}; x[\text{rd}] = t$

正如前面提到的，我们将 CSR 的访存模式设置成与寄存器堆完全一致，而 CSR 涉及到的计算操作采用 CCU 完成。基于此，我们将 CSR 接入了 SR1-MUX 与 SR2-MUX，便于后续的选择操作。同时，考虑到寄存器堆为写优先而 CSR 为读优先，在处理 ID 与 WB 段 CSR 指令的数据相关时，我们需要将 WB 段 CCU 计算结果也进行前递。

4.10 冒险与停顿控制单元 (Forwarding Hazard, FH)

4.10.1 可能涉及到的数据冒险

如下表所示。针对本实验所设计的基本指令，程序中可能的数据冒险情况如下：

在后 \在前	alur	alui	lw	sw	beq	jal	jalr	lui	auipc	csr
alur	解决	解决	解决	无相关	无相关	解决	解决	解决	解决	解决
alui	解决	解决	解决	无相关	无相关	解决	解决	解决	解决	解决
lw	解决	解决	解决	无相关	无相关	解决	解决	解决	解决	解决
sw	解决	解决	解决	无相关	无相关	解决	解决	解决	解决	解决
beq	解决	解决	解决	无相关	无相关	解决	解决	解决	解决	解决
jal	无相关	无相关	无相关	无相关	无相关	无相关	无相关	无相关	无相关	无相关
jalr	解决	解决	解决	无相关	无相关	解决	解决	解决	解决	解决
lui	无相关	无相关	无相关	无相关	无相关	无相关	无相关	无相关	无相关	无相关
auipc	无相关	无相关	无相关	无相关	无相关	无相关	无相关	无相关	无相关	无相关
csr	解决	解决	解决	无相关	无相关	解决	解决	解决	解决	解决

其中，sw、beq 指令因为不涉及对寄存器堆的修改，所以不会与其之后的指令产生数据相关；jal、lui、auipc 指令因为不涉及对寄存器堆的读取操作，所以不会与其之前的指令产生数据相关。

对于可能出现的多级相关情况，本实验的解决方式是：**从 EX 到 MEM 由近到远顺次判断。**

由于本实验采用了写优先的寄存器堆设计，在 WB 段不需要进行数据前递。（尽管寄存器还没有被写入，但将要写入的数值已经被正确读出）所以不存在 ID 与 WB 段指令的数据相关情况。

对于不同的数据相关情况，我们按照种类进行一一分析。

针对 sr1 与 dr 相关的情况：

IF	ID	EX	MEM	WB
	alu/lw/sw/jalr	alu/auipc/lui		

- 选择 alu_ex

IF	ID	EX	MEM	WB
	alu/lw/sw/jalr	任何指令	alu/auipc/lui/lw/jal/jalr	

- MEM 为 lw 选择 dm_mem
- MEM 为 jal/jalr 选择 pc+4
- 其他 选择 alu_mem

针对 sr2、dm_sr2 与 dr 相关的情况

(仅有 alur, beq, sw 有 sr2)

IF	ID	EX	MEM	WB
	alur/sw	alu/auipc/lui		

- 选择 alu_ex

IF	ID	EX	MEM	WB
	alur/sw	任何指令	alu/auipc/lui/lw/jal/jalr	

- MEM 为 lw 选择 dm_mem
- MEM 为 jal/jalr 选择 pc+4
- 其他 选择 alu_mem

针对 bsr1、bsr2 与 dr 相关的情况：

(由于分支预测模块提前，我们需要对其进行专门的数据前递判断)

IF	ID	EX	MEM	WB
	beq	alu/auipc/lui		

- 选择 alu_ex

IF	ID	EX	MEM	WB
	beq	任何指令	alu/auipc/lui/lw/jal/jalr	

- MEM 为 lw 选择 dm_mem
- MEM 为 jal/jalr 选择 pc+4
- 其他 选择 alu_mem

4.10.2 可能遇到的流水线停顿

本实验中所涉及的流水线停顿情况包括：

- lw 指令与其后指令数据相关：由于 lw 在 MEM 段执行完成，故将其后续指令暂停一个时钟周期，插入一条空白指令。
- beq 指令与其前指令数据相关：由于 beq 在 ID 段执行完成，故将 beq 指令暂停一个时钟周期，插入一条空白指令。
- 乘除法指令与其前指令数据相关：由于乘除法指令在 MEM 段执行完成，故将其后续指令暂停一个时钟周期，插入一条空白指令。
- 两条相邻的乘除法指令：由于乘除法指令在 MEM 段执行完成，故将其后续指令暂停一个时钟周期，插入一条空白指令。
- 跳转成功：由于分支预测失败，需要将错误进入流水线的 pc+4 指令清除。此时流水线不停顿。

具体的细节如下表所示：

IF	ID	EX	MEM	WB
	相关的任何指令	lw		

停顿 + 清空

IF	ID	EX	MEM	WB
pc+offset	pc+4	beq(成功跳转)/jal/jalr		

仅清空

IF	ID	EX	MEM	WB
pc+offset	pc+8	0	jalr	

仅清空

IF	ID	EX	MEM	WB
	beq	任何指令	lw/jal	

停顿 + 清空

IF	ID	EX	MEM	WB
	beq	alu/lui/auipc		

停顿 + 清空

IF	ID	EX	MEM	WB
	任何相关的指令 & mul/div	mul/div		

停顿 + 清空

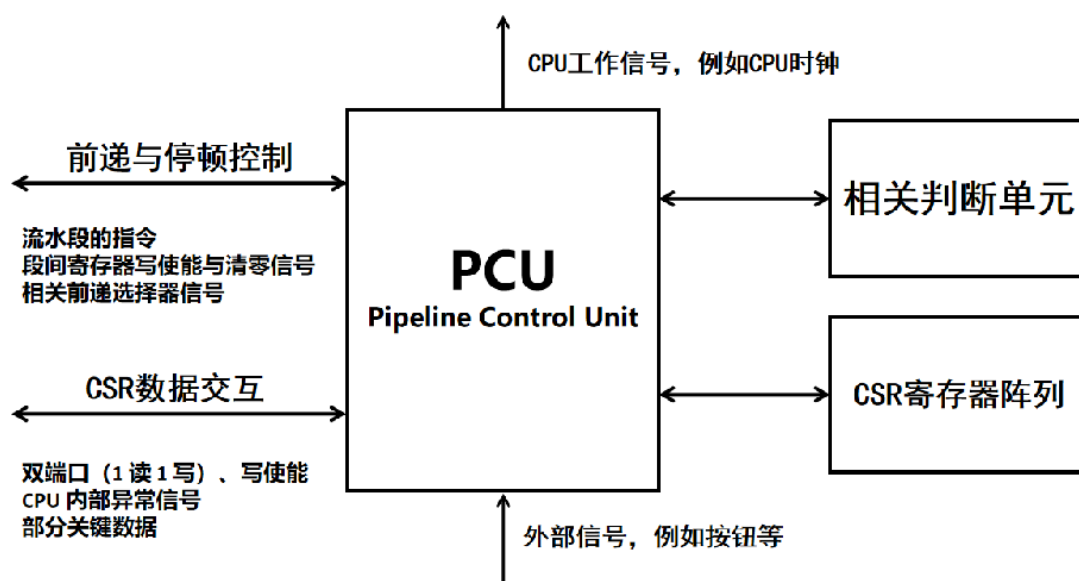
4.10.3 HS 模块设计

最终，由 HS 生成的控制信号进入 EX 段控制各源操作数选择器的数值。对于 b_sr1_mux、b_sr2_mux、dm_sr2_mux，其对应控制信号即为 FH 所生成的控制信号；对于 sr1_mux、sr2_mux，其控制信号需要综合 DU 与 FH 生成的控制信号。具体操作如下：

```
always @(*) begin
    if (sr1_mux_sel_fh[2] == 1'b1)
        sr1_mux_sel = sr1_mux_sel_fh;
    else
        sr1_mux_sel = sr1_mux_sel_du;
end

always @(*) begin
    if (sr2_mux_sel_fh[2] == 1'b1)
        sr2_mux_sel = sr2_mux_sel_fh;
    else
        sr2_mux_sel = sr2_mux_sel_du;
end
```

4.11 流水线控制器 (Pipeline Control Unit, PCU)

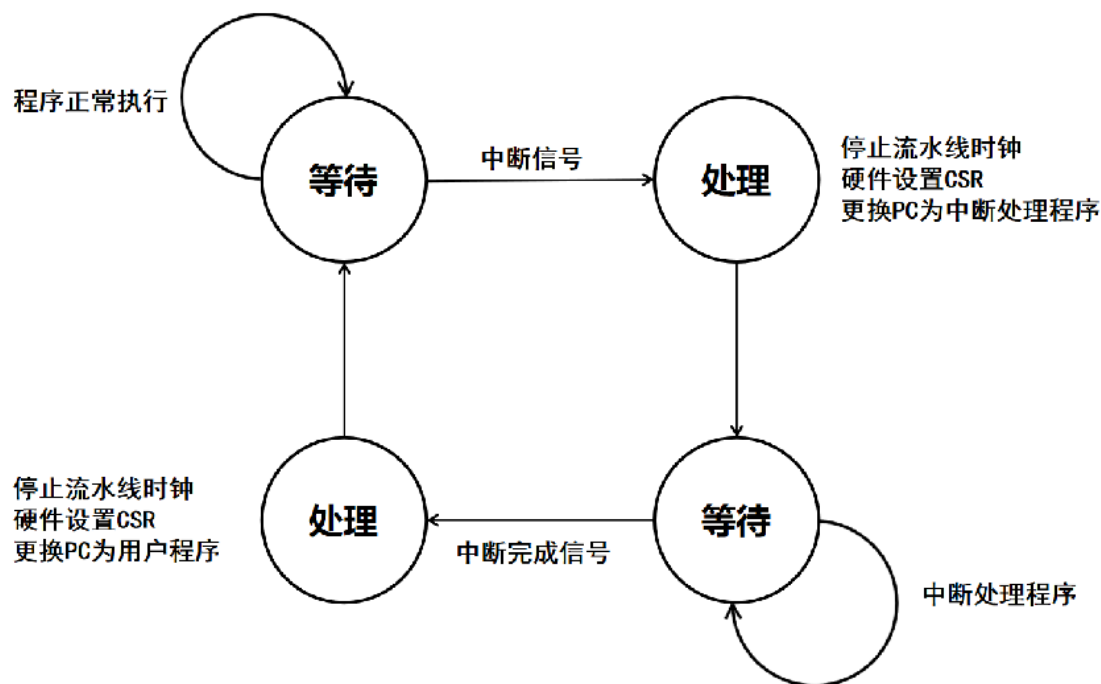


最后，我们来介绍流水线控制器模块。该模块控制着流水线的全部工作时序，并根据各种中断信号进行异常处理。其工作内容包括但不限于：

- 流水线停顿部分：定向清除段间寄存器、定向控制段间寄存器写使能、控制 PC 写使能；
- 数据前递部分：定向控制前递选择器的信号；
- 中断检测与处理部分：检测 CPU 内外的中断信号，根据信号种类不同进入不同的中断响应态，并在中断处理完成后恢复 CPU 的正常运行；
- CSR 数据交互部分：封装 CSR，采用与寄存器堆类似的方式与流水线进行数据交互。

流水线控制器是 CPU 的心脏，其成功与否直接影响着流水线 CPU 的工作性能。接下来我们针对其内部结构进行一一拆解。

4.11.1 中断状态机



上图为中断处理过程的一个简要状态转换示意图。正常情况下，CPU 处于用户等待态，执行者用户程序的具体内容。当外来的中断信号发生时，状态机会跳转到中断处理状态。此状态下的 PCU 需要完成：停止流水线时钟、设置相应的 CSR、设置 PC 为中断服务程序起始地址。完成这些操作后，PCU 恢复流水线时钟，状态机进入中断等待态。等到中断处理完成后，PCU 再次停止流水线时钟，恢复 PC 并设置 CSR。这样就完成了一次中断处理的循环。

CPU 时钟采用另一个小型状态机控制。PCU 内部集成了 Lab5 中提到的可变周期时钟。我们通过将该时钟信号按位与上 flag 标记，即可实现 CPU 工作时钟的启用与关闭。

```
localparam CPU_CLK_N = 11'd1;
// 可变周期时钟
always @(posedge clk or negedge rstn) begin
    if (~rstn) begin
        cpu_clk_conter <= 0;
        slow_clk <= 0;
    end
    else begin
        if (cpu_clk_conter == CPU_CLK_N + CPU_CLK_N) begin
            cpu_clk_conter <= 32'b1;
            slow_clk <= 1'b1;
        end
        else begin
            if (cpu_clk_conter < CPU_CLK_N)
                slow_clk <= 1'b1;
            else
                slow_clk <= 1'b0;
                cpu_clk_conter <= cpu_clk_conter + 'h1;
            end
        end
    end
end
```

```
// 小型状态机
```



```

always @(*) begin
    if (~rstn) begin
        clk_ns = CLOCK_STOP;
    end
    else begin
        case (clk_cs)
            CLOCK_RUN: begin
                if (ebreak || next_state == Reload_Part1 || button_sig) begin
                    // Breakpoint
                    clk_ns = CLOCK_STOP;
                end
                else
                    clk_ns = CLOCK_RUN;
            end

            CLOCK_STOP: begin
                if (pcu_run) begin
                    // Breakpoints: user press [enter], then pdu_run will be 1
                    // Other interrupt: the program will set Mipd to 1
                    clk_ns = CLOCK_RUN;
                end
                else
                    clk_ns = CLOCK_STOP;
            end
            default: clk_ns = CLOCK_STOP;
        endcase
    end
end
end

```

4.11.2 中断信号

本实验中，我们设计了如下的中断与异常信号：

编号与名称	信号类型	产生原因
0 - 程序断点	中断	ID 段为 ebreak 或 ecall 指令
1 - 用户断点	中断	ID 段 PC 值等于用户设置的断点
2 - 按钮信号	中断	用户按下了按钮（迷宫游戏限定）
3 - 被除数为 0	异常	字面意思（除法限定）
4 - 访问越界	异常	IMU DMU 的访问越界
5 - 未定义指令	异常	不属于 RISC-V 32 IMB 的 opcode

4.11.3 中断处理

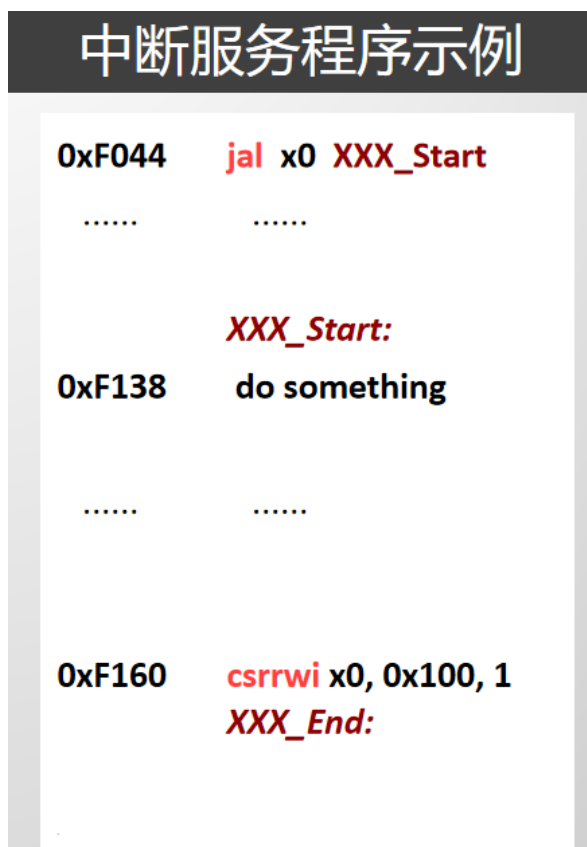
为了简化流程，我们将所有的中断处理程序设置成等待用户操作。程序实际上并不会对流水线做什么，而是以停机的形式告诉用户程序出现了异常。此时用户便可以通过查看 CSR 中对应的数值了解中断原因。

编号与名称	Mtevc	Mcause	Mepc	Mtval
0 - 程序断点	F000	0	IF 段 PC	0
1 - 用户断点	F004	1	IF 段 PC	0
2 - 按钮信号	F010	2	ID 段 PC	0
3 - 被除数为 0	F020	3	ID 段 PC	EX 段 PC
4 - 访问越界	F024	4	IF 段 PC	访问地址
5 - 未定义指令	F028	5	IF 段 PC	ID 段 IS

4.11.4 中断服务程序

中断服务程序是存储于另一个数据存储器中的程序。本实验中，我们将中断程序空间分为：中断服务处理程序向量表、中断处理程序。其中，0xF000~0xF07C 存储了 32 条中断服务程序的跳转指令，从 0xF080 到 0xFEFC 存储了真正的中断服务程序。

一个中断服务程序示例如下图所示：



所有的中断服务处理程序的最后一条指令一定为 `csrrwi x0, 0x100, 1`，其作用是将 CSR-Mipd 的数值设为 1。此时 PCU 会检测到中断处理完成的信号，并执行用户的后续操作。当然，CSR-Mipd 的清零操作是由硬件自动完成的。