

1. 设计定位

(胡云峰)

随着智能家居、物联网、智能制造等领域的迅猛发展,对于功耗低、性能高、可靠稳定、成本低廉的 MCU 需求也越来越大。因此,我们设计了一款采用 riscv 架构,具有基本整数指令集 I 和扩展的压缩指令集 C 的 32 位 MCU 处理器,能够满足 MCU 领域的大多数需求。由于没有 M 扩展指令集和 F 扩展指令集,其流水线设计相对简洁,电路面积更小,功耗更低,完美符合 MCU 的应用场景。支持压缩指令集 C 使得在相同的存储空间下可以放入更复杂的程序或是相同的程序可以用更小的存储空间,以提高在存储资源受限场合的适应性。

2. 支持的指令集

(胡云峰)

基本整数指令集 I 是 riscv 架构的核心指令集,它包含了 36 条指令(不含 ecall, ebreak),其中包括算术、逻辑、移位、比较、条件分支、无条件跳转、存储、加载等指令,可以满足大部分的程序设计需求。基本整数指令的分类与格式如下表 1 所示:

表 1 基本整数指令集 I

31-25	24-20	19-15	14-12	11-7	6-0	类型
funct7	rs2	rs1	funct3	rd	opcode	R
imm[11:0]		rs1	funct3	rd	opcode	I
imm[11:5]	rs2	rs1	funct3	rd	opcode	S
imm[12 10:5]	rs2	rs1	funct3	imm[4:1 11]	opcode	SB
imm[31:12]				rd	opcode	U
imm[20 10:1 11 19:12]				rd	opcode	UJ

压缩指令集 C 是针对处理器代码密度进行优化设计的指令集,通过统计各条基本整数指令在实际程序中出现的频率,将最常见的指令选出并设计成 16 位的压缩指令集,以实现更紧凑的代码和更高效的内存访问。虽然压缩指令集 C 的指令数比基本整数指令集 I 少,但是它们可以通过解码过程转换成基本整数指令,从而具有与标准指令集相同的功能。由于 RISC-V 的压缩指令集 C 可以在不牺牲性能的情况下节省存储空间,因此它广泛应用于资源受限的嵌入式和移动应用中,并与 RISC-V 的标准指令集 I 一起使用,以提高代码密度和执行效率。压缩指令

的分类与格式如下表 2 所示：

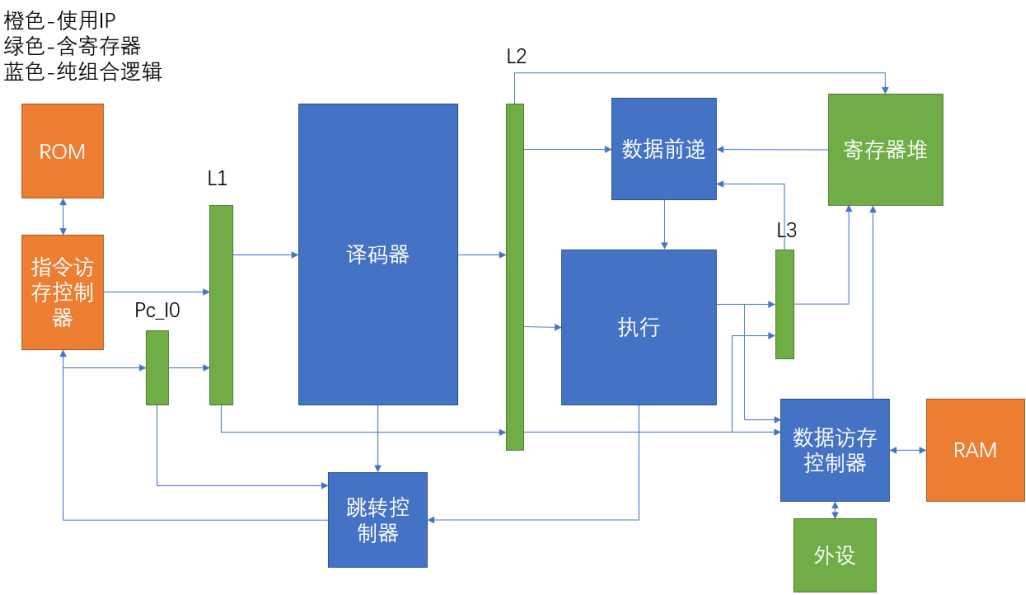
表 2 压缩指令集 C

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	类型
funct4				rd/rs1					rs2					opcode		CR
funct3			imm	rd/rs1					imm					opcode		CI
funct3			imm						rs2					opcode		CSS
funct3			imm									rd'		opcode		CIW
funct3			imm				rs1'		imm		rd'		opcode		CL	
funct3			imm				rd'/rs1'		imm		rs2'		opcode		CS	
funct3			imm				rs1'		imm					opcode		CB
funct3			offset											opcode		CJ

3. 整体架构

(胡云峰)

本设计采用经典的五级流水线设计，若要追求极致的低功耗和小面积，也可以采用单周期或二级流水线的设计整体架构如下图 1 所示。



划分成五级流水线可以提升处理器频率，大大提高处理器性能，同时也会提高设计的复杂性。每级流水线的功能分别是：

- 1) 取指阶段（IF）：从程序存储器取出指令，存储到指令寄存器中。
- 2) 译码阶段（ID）：对指令进行解析，确定操作码和操作数。
- 3) 执行阶段（EX）：根据指令码和操作数从寄存器堆中获取数据进行运算。

- 4) 访存阶段 (MEM): 用于数据存取, 包括从内存读取数据, 将数据写入内存。
- 5) 回写阶段 (WB): 将计算结果写回到寄存器堆中。

pc₁₀ 寄存器存储的是已取出指令所对应的 pc 值, 由于使用的 ROM IP 需要在输入地址后给一个时钟信号才能读取到数据, 所以是将 pc₁₀ 送入跳转控制器, 如果跳转控制器没有收到译码阶段和执行阶段送来的跳转信号, 那么跳转控制器输出的则是 $next_pc = pc_10 + 2$ 或 $+4$ (取决于已取出的指令是否为压缩指令)。时钟信号到来时 $pc_10 \leq next_pc$, 并取出一条指令, 再来一个时钟信号, 取出的指令和对应的 pc 值存入 L1 寄存器并开始译码, 输出立即数, rs1, rs2 以及控制信号。

在译码阶段可以提前做 jal 指令的跳转, jal 指令为 pc+立即数跳转, 因此在译码阶段就可以提前确定跳转地址。

下一个时钟信号到来后进入执行阶段, 译码阶段的各种信号存入 L2, 执行阶段的电路根据 L2 输出的立即数, pc, rs1, rs2 以及控制信号进行运算, 需要注意的是由于流水线的存在, 如果上一条指令写入的寄存器被当前指令用到, 而由于中间间隔着访存步骤导致寄存器堆内的值没有被及时更新, 读取出来的 xrs1 和 xrs2 的值是旧值, 从而产生错误。解决方案可以是将流水线阻塞一个周期, 等寄存器堆中的数值更新后再读取, 或者增加一层判断, 将待写入寄存器堆的值前递到执行阶段。由于 RAM 与 ROM 有同样的特点, 需要在给出地址后再给一个时钟信号才会输出数据, 所以在执行阶段就要算出访存地址并且不经过 L3 寄存器, 直接给 RAM。

如果执行阶段正在执行一条 branch 或 jalr, 译码阶段正在译码一条 jal, 那么跳转控制器会同时收到两个跳转地址信号, 此时应该是跳转到执行阶段给出的地址还是译码阶段给出的地址呢? 显然是跳转到执行阶段给出的地址, 因此跳转控制器需要一个额外的判断, 给予执行阶段的跳转地址更高的优先级。无论是在译码阶段的 jal 跳转还是执行阶段的 branch 和 jalr 跳转, 都需要做清空流水线的操作, 但是在译码阶段跳转可以减少清理一级流水线, 从而提升效率。

4. 各模块的实现细节

4.1. Decode 设计

(沈佳先)

RISC-V 解码模块是 RISC-V 处理器中的一个重要组成部分，其作用是将指令二进制码解码成对应的操作数和操作码，并生成对应的控制信号，以便后续执行模块执行指令。RISC-V 指令集包括基本指令、扩展指令和特权指令等多种类型，每种类型指令的二进制码格式也不同。解码模块需要支持不同类型指令的解码，并将其转换为对应的操作数和操作码，以便后续执行模块正确地执行指令。如果没有解码模块，处理器将无法正确地执行指令，导致系统出错。解码模块还需要生成对应的控制信号，包括 ALU 操作控制信号、访存控制信号、分支控制信号、跳转控制信号等，以便后续执行模块正确地执行指令。这些控制信号是执行指令所必需的，如果没有解码模块生成这些控制信号，处理器将无法正确地执行指令，导致系统出错。

解码模块可以提高处理器的性能。在 RISC-V 处理器中，解码模块是指令流水线的第二个阶段，其输出是下一个阶段的输入。通过解码模块的解码过程，可以将指令的操作数和操作码提前生成，以便后续阶段更快地执行指令。这样可以提高处理器的性能，降低指令执行的延迟。

4.1.1. Decode 模块功能需求

解码模块需要完成以下功能：

- 1) 从指令集存储器读取指令二进制码；
- 2) 生成对应的控制信号，包括 ALU 操作控制信号、访存控制信号、分支控制信号、跳转控制信号等；
- 3) 将解码后的指令信息传递给下一级执行模块。

4.1.2. 接口设计

基于 Decode 模块的功能需求，本设计 Decode 模块架构框图如下图 2 示：

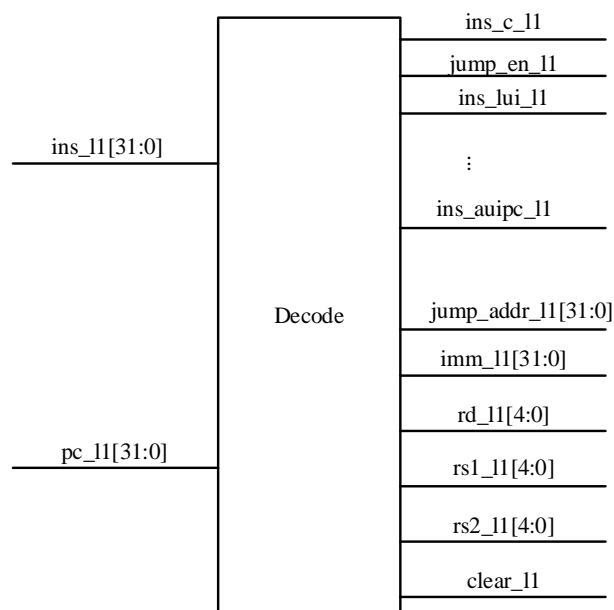


图 2 Decode 模块架构框图

图中 `ins_11` 是输入 32 位指令，`pc_11` 是当前指令地址。输出端口方面，上半部分为所有识别指令的控制信号，当译出指令时，对应指令的控制信号拉高，其余控制信号拉低，`jump_addr_11` 是跳转指令对应的跳转地址，由跳转指令立即数加上当前指令地址得到。`imm_11` 是指令立即数，`rd` 为目的寄存器编号，`rs1_11` 为 `rs1` 寄存器编号，`rs2_12` 是 `rs2` 寄存器编号，`clear_11` 是清除寄存器指令信号，配合跳转指令使用。

总而言之，该模块的输出信号包括指令类型信号（如 `LUI` 指令、`ADDI` 指令等），寄存器编号（`rd`、`rs1`、`rs2`）、立即数信号、跳转地址信号、跳转使能信号，以及一些其他控制信号（如清空信号、压缩指令信号等）。这些信号可以根据不同的指令类型，在后续的执行阶段中控制计算机的运算和数据传输。

4.1.3. Decode 设计

秉持着减少设计面积开销的原则，`Decode` 设计不针对每一个指令进行译码，而是通过代码分类，总结指令规律，进行译码，能够较大程度地减少开销。这也得益于 `RISC-V` 指令集设计的优势。由于本设计除了基本指令集之外还增加了 `C` 扩展指令集，因此以下分两部分讲解。

针对基本指令级，如下图 3 所示为基础指令集的分类，集合具体指令集手册可以看到，各个种类的指令基本可以通过 `opcode` 进行分辨。而每种指令则可以通过 `funct3` 进行分辨，如果还是分辨不了，最后还可以通过 `function7` 进行分辨。

31	25	24	19 15	14 12	11	7	6	0	
funct7	rs2	rs1	funct3	rd	opcode				R类
imm[11:0]		rs1	funct3	rd	opcode				I类
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode				S类
imm[12 10:5]	rs2	rs1	funct3	imm[4:1 11]	opcode				SB类
imm[31:12]				rd	opcode				U类
imm[20 10:1 11 19:12]				rd	opcode				UJ类

图 3 基础指令集类型以及指令字段含义

以 sb 类指令为例子，如下图 4 所示，sb 类指令 opcode 为 1100011，那么其中的指令可以通过 funct3 进行分辨，其余指令类似原理。

```

wire type_sb = opcode == 7'b1100011;
wire ins_beq = type_sb && funct3_000;
wire ins_bne = type_sb && funct3_001;
wire ins_blt = type_sb && funct3_100;
wire ins_bge = type_sb && funct3_101;
wire ins_bltu = type_sb && funct3_110;
wire ins_bgeu = type_sb && funct3_111;

```

图 4 sb 类指令区分

各指令类型的立即数也是通过区分各指令类型来进行译码。其中，跳转指令、加法减法等运算指令的立即数是有符号立即数，应该注意的是对这类指令的立即数扩展需按照有符号数规则进行扩展。如下代码所示，跳转指令的立即数扩展，需要将立即数最高位对寄存器高 11 为赋值。其余 rs1 和 rs2 按照各指令类型译码即可。

```

wire [31:0] imm = (type_uj ? {{11{ins_11[31]}}}, ins_11[31], ins_11[19:12], ins_11[20],
ins_11[30:21], 1'b0}: 32'd0);

```

针对 C 扩展译码，如下图 5 所示，C 扩展指令的规律性不强，虽然采用基础指令集同样的译码方式，但是每条指令基本需要单独的组合逻辑判断。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
funct4				rd/rs1				rs2				op			CR-type	
funct3			imm	rd/rs1				imm				op			CI-type	
funct3			imm					rs2				op			CSS-type	
funct3			imm								rd'		op			CIW-type
funct3			imm		rs1'			imm		rd'		op			CL-type	
funct3			imm		rd'/rs1'			imm		rs2'		op			CS-type	
funct3			imm		rs1'			imm				op			CB-type	
funct3			offset											op		CJ-type

图 5 C 扩展指令类型

最后需要将 C 扩展指令控制信号通过或逻辑门与基本指令集进行合并，以 `ins_lui_11` 控制信号为例，如下代码所示：

```
assign ins_lui_11 = ins_lui || c_ins_lui;
```

对于跳转指令，本设计控制信号有 `jump_en_11` 和 `clear_11`，分别为跳转使能和寄存器清空信号。当跳转指令控制信号拉高时，以上两个信号同时拉高，然后对跳转地址赋值，如下代码所示：

```
assign jump_addr_11 = pc_11 + imm_11;
```

因此，对于基本指令级，由于 RISC-V 的优良特性，使得指令集译码时产生的面积开销并不大，反而是扩展指令集，由于指令没有很好的对 RISC-V 基础指令集适配，因此扩展指令集对于处理器扩展而言会大大增加芯片面积。

4.1.4. 总结

综上所述，RISC-V 解码模块是 RISC-V 处理器中的一个重要组成部分，其作用是将指令二进制码解码成对应的操作数和操作码，并生成对应的控制信号，以便后续执行模块执行指令。RISC-V 解码模块的重要性体现在支持不同类型指令的解码、生成对应的控制信号、提高处理器的性能和适应不同的应用场景等方面。

4.2. 执行模块设计

(王朝栋)

4.2.1. 执行模块定义及功能

执行(ex)模块是纯组合逻辑电路，主要功能如下：

- 1) 根据当前是什么指令执行对应的操作，比如 `add` 指令，则将寄存器 1 的值和寄存器 2 的值相加。
- 2) 如果是内存加载指令，则读取对应地址的内存数据。

3) 如果是跳转指令，则发出跳转信号，并且给出跳转地址。

其中执行部分获取译码阶段的信号后，执行相应的操作.并且将处理后的数据值传给后级。

4.2.2. 执行模块的输入输出

执行模块的输入输出信号如下表所示：

- 1) 数据类 Input: alu_a_l2、alu_b_l2、pc_l2、imm_l2
- 2) 分别是 ALU 运算单元的两个输入端，处理过后的对应每条指令的立即数，pc 地址。
- 3) 指示类 Input: 该指令是否执行 ins_c_l2、ins_lui_l2、ins_auiipc_l2、ins_jal_l2 等等共 38 条线网输入。若其值为 1 表示需要执行该指令，若其值为 0 则表示无需执行。其中需要特别注意一条指令 ins_c_l2，该指令表示是否为压缩指令，其中压缩指令与非压缩指令的最大的区别在于执行完该指令 pc 值加 2or 加 4。
- 4) 输出 Output: alu_q_l2 jump_en_l2 jump_addr_l2 clear_l1 clear_l1

其中 q 为算术单元的输出代表输出的结果 en 代表使能信号,当 en 为 1 时，即代表跳转指令有效，addr 即为跳转的地址，跳转成功时还需要清空流水线，故跳转成功时输出 clear_l1 和 clear_l2 = 1。根据分析指令的执行条件以及判断条件，本设计在处理该信号之前还需要生成额外的信号，并且为了更好的复用加法器以及减少设计的面积，则可以对于共同的信号进行提前处理。

4.2.3. 执行模块的指令分析

ISC-V 指令集架构定义了三种基本指令类型：R 型指令、I 型指令和 S 型指令。此外，还定义了 B 型指令、U 型指令和 J 型指令。下面是这些指令类型的详细介绍：

- 1) R 型指令 (Register Type): 这种指令类型用于执行寄存器操作，包括算术、逻辑和移位等操作。这种指令类型使用三个寄存器作为操作数，并将计算结果存储到一个寄存器中。例如，ADD 指令可以执行两个寄存器相加操作，并将结果存储到一个寄存器中。
- 2) I 型指令 (Immediate Type): 这种指令类型用于执行带立即数的操作，包括加载、存储、算术和逻辑等操作。这种指令类型使用一个寄存器和

一个立即数作为操作数，并将计算结果存储到一个寄存器中。例如，ADDI 指令可以将一个寄存器的值与一个立即数相加，并将结果存储到一个寄存器中。

- 3) S 型指令 (Store Type): 这种指令类型用于执行存储器操作，包括将一个寄存器的值存储到存储器中。这种指令类型使用两个寄存器和一个偏移量作为操作数。其中，一个寄存器包含要存储的数据，另一个寄存器包含存储器的基地址，偏移量指定存储器中的偏移位置。
- 4) B 型指令 (Branch Type): 这种指令类型用于执行分支操作，比如跳转、条件分支和无条件分支等。这种指令类型使用两个寄存器和一个偏移量作为操作数，其中，一个寄存器包含比较的值，另一个寄存器包含要比较的值，偏移量指定跳转的位置。
- 5) U 型指令 (Upper Immediate Type): 这种指令类型用于执行带有扩展立即数的操作，比如将一个立即数扩展为一个 32 位的值。这种指令类型使用一个寄存器和一个立即数作为操作数，并将计算结果存储到一个寄存器中。
- 6) J 型指令 (Jump Type): 这种指令类型用于执行无条件跳转操作。这种指令类型使用一个偏移量作为操作数，并将程序计数器 (PC) 设置为跳转的位置。

对于 R 型 I 型指令，需要两个操作数运算，操作数与立即数运算。对于 B 型指令，需要比较两个操作数的大小进行操作，特别注意的是对于 blt、bge 需要比较有符号数的大小。并且需要提前计算出 pc_l2+imm 对于 S 型指令，需要给出跳转的地址，即为需要提前算出操作数 $rs1+imm$ 。对于 J 指令，则需要给出跳转的地址 $rs1+偏移量 imm$

4.2.4. 执行模块的实现

对于 R 指令中的 add 指令以及 sub 指令，如下图 6 所示当 ins_add_l2 、 ins_sub_l2 或者 ins_sra_l2 为 1 时，代表此时需要执行其中的指令。

```

assign data4 = alu_a_l2; //rs1+(or-)rs2
assign data5 = (ins_sub_l2 == 1'b1)? (~alu_b_l2 + 1'b1) : alu_b_l2; //若为减法指令则对
rs2 取反加一将减法转为加法
//assign data3 = alu_b_l2;
alu_add myadd2 ( //TODO: 输入端口
    .data0    (data4),
    .data1    (data5),
    .ALU_result(sumadd1) //sum = rs1 +(-) rs2
);

```

图 6 add 及 sub 指令部分代码

利用条件判断，当 ins_sub_l2 为 1 时第二个操作数 rs2 需要取反加 1，也就是将减法转换为加法操作。即可以省去一个加法器，减少面积。

对于 ins_sra_l2 指令执行起来较为复杂，如图 7 所示当其为 1 时，首先将 32'hffff_ffff 进行移位操作，因为最多能移动 32 位所以只需要使用移位数的低 5 位表示 ($2^5=32$)。再对操作数 rs1 逻辑移位 rs2 表示的位数，获得移位后的数 alu_a_shift_right_alu_b。

```

alu_q_l2= ((alu_a_shift_right_alu_b) & SRA_mask) | ({32{alu_a_l2[31]}} & (~SRA_mask));

```

图 7 ins_sra_l2 指令部分代码

将移位后的数据与符号位相与即可得到算术移位的数据，同理对于 I 型指令中的 srai 同理，并且再 SRA_mask 中已经实现了相应的操作，当 ins_srai_l2 为 1 时那么表面需要执行立即数的算术右移，此时将把第二个操作数换位 imm 立即数。

对于 B 型指令，以 ins_blt_l2 为例进行说明，如下图 8 所示，当 ins_blt_l2 为 1 时即代表需要执行条件跳转指令。即为当 $rs1 < rs2$ 时跳转到相应的地址且为有符号数的比较。

```

assign op1_ge_op2_signed= $signed(alu_a_l2) >= $signed(alu_b_l2);

```

图 8 blt 指令控制信号赋值

使用上述的语法，对有符号位的数据进行比较，op 代表比较的结果。当标志位有效时，就把跳转使能位置为 1，使能有效，并且给出跳转地址，跳转到对应的位置。并且由于是跳转执行，所以跳转后需要清空流水线，所以 clear 置为 1，否则若标志位比较失败，则不跳转。

对于 S 型存储指令，如下图 9 所示当 `ins_lb_l2` 位 1 时，说明需要执行加载指令，需要将内存中的值读出，而执行部分需要做的操作是，给后续对应读取数据的内存地址。

```
else if (ins_lb_l2) begin //访存操作只需要给出地址即可
    alu_q_l2      = sumadd;
    jump_en_l2    = 1'b0;
    clear_l1      = 1'b0;
    clear_l2      = 1'b0;
    jump_addr_l2 = 32'b0;
end
```

图 9 S 型指令部分代码

注意：每个分支都必须考虑到所有的信号，也就是所有的输出信号都必须给值，否则会出现锁存结构，这是不期望出现的。由于译码阶段给出的信号值为各个指令是否执行的信号，所以不使用 `case` 结构进行判断，而使用大段的 `if else if else` 则会使关键路径变得很长，使延时很大，从而降低时钟频率，所以使用的是 `unique if else if else` 的结构，即为将所有的信号并行判断，综合成并行判断则大大减少了关键路径的长度，降低了延时，增大了最大的时钟频率。

4.2.5. 总结

执行模块是 RISC-V 处理器的核心组件之一，负责执行指令并完成计算任务。它的设计和实现对处理器的性能和功能有着重要的影响。

4.3. 寄存器控制设计

(叶明)

4.3.1. DFF 模块设计

RISC-V DFF 模块是 RISC-V 处理器中的一个重要组成部分，其作用是将单周期处理器的组合逻辑进行插入寄存器，对处理器进行流水线划分，并使用对应的控制信号进行控制，从而实现流水线的控制。

```

1 module dff_set #(
2     parameter DW = 32
3 )
4 (
5     input wire          clk,
6     input wire          rst_n,
7     input wire          clean,
8     input wire          hold_flag,
9     input wire [DW-1:0] set_data,
10    input wire [DW-1:0] data_i,
11    output reg [DW-1:0] data_o
12 );
13
14 always @ (posedge clk) begin
15     if ((rst_n == 1'b0) || (clean == 1'b1))begin
16         data_o <= set_data;
17     end else if (hold_flag)begin
18         data_o <= data_o;
19     end else begin
20         data_o <= data_i;
21     end
22 end
23
24 endmodule

```

图 10 dff 模块部分代码

如图 10 所示，使用统一的 dff 模块进行来对信号进行处理，使用 DW 定义位宽使得在对不同位宽信号进行处理时可以进行统一例化，提高代码编码效率。

```

104 dff_set #(32) dff1(clk, rstn, clear_l2, block_l2, 32'b0, pc_l1, pc_l2);
105
106 dff_set #(32) dff2(clk, rstn, clear_l2, block_l2, 32'b0, imm_l1, imm_l2);
107
108 dff_set #(5) dff3(clk, rstn, clear_l2, block_l2, 5'b0, rd_l1, rd_l2);
109
110 dff_set #(5) dff4(clk, rstn, clear_l2, block_l2, 5'b0, rs1_l1, rs1_l2);
111
112 dff_set #(5) dff5(clk, rstn, clear_l2, block_l2, 5'b0, rs2_l1, rs2_l2);
113
114 dff_set #(1) dff6(clk, rstn, clear_l2, block_l2, 1'b0, ins_c_l1, ins_c_l2);
115
116 dff_set #(1) dff7(clk, rstn, clear_l2, block_l2, 1'b0, ins_lui_l1, ins_lui_l2);
117
118 dff_set #(1) dff8(clk, rstn, clear_l2, block_l2, 1'b0, ins_auipe_l1, ins_auipe_l2);
119
120 dff_set #(1) dff9(clk, rstn, clear_l2, block_l2, 1'b0, ins_jal_l1, ins_jal_l2);
121
122 dff_set #(1) dff10(clk, rstn, clear_l2, block_l2, 1'b0, ins_jalr_l1, ins_jalr_l2);
123
124 dff_set #(1) dff11(clk, rstn, clear_l2, block_l2, 1'b0, ins_beq_l1, ins_beq_l2);
125
126 dff_set #(1) dff12(clk, rstn, clear_l2, block_l2, 1'b0, ins_bne_l1, ins_bne_l2);
127
128 dff_set #(1) dff13(clk, rstn, clear_l2, block_l2, 1'b0, ins_blt_l1, ins_blt_l2);
129
130 dff_set #(1) dff14(clk, rstn, clear_l2, block_l2, 1'b0, ins_bge_l1, ins_bge_l2);
131
132 dff_set #(1) dff15(clk, rstn, clear_l2, block_l2, 1'b0, ins_bltu_l1, ins_bltu_l2);
133
134 dff_set #(1) dff16(clk, rstn, clear_l2, block_l2, 1'b0, ins_bgeu_l1, ins_bgeu_l2);
135
136 dff_set #(1) dff17(clk, rstn, clear_l2, block_l2, 1'b0, ins_lb_l1, ins_lb_l2);
137
138 dff_set #(1) dff18(clk, rstn, clear_l2, block_l2, 1'b0, ins_lh_l1, ins_lh_l2);
139
140 dff_set #(1) dff19(clk, rstn, clear_l2, block_l2, 1'b0, ins_lw_l1, ins_lw_l2);
141
142 dff_set #(1) dff20(clk, rstn, clear_l2, block_l2, 1'b0, ins_lbu_l1, ins_lbu_l2);
143

```

图 11 dff 例化代码

在 dff 模块中具有复位信号(rst_n)、清空信号(clean)、保持信号(hold_flag)三个控制信号，复位或清空信号有效时将输出数据置位至设置数据，保持信号有效时将输出数据保持为前周期输出数据。图 11 中为各信号的寄存器打拍模块，使用统一的例化能够有效减少代码复杂度。

4.3.2. PC 控制模块

PC 控制模块产生下一条指令所在的地址，其中具有从译码阶段产生的跳转使能信号 `jump_en_l1`，与从执行阶段产生的跳转使能信号进行跳转控制，若跳转使能信号拉高则跳转至对应的跳转地址。若无跳转则判断当前指令是否为压缩指令，若为压缩指令则下周期地址为 `PC+2` 否则为 `PC+4`。并且压缩指令集的实现需要进行指令拼接，因此还需要输入下条指令地址+2 的地址(`next_pc_addr2`)。

4.3.3. 流水线前递模块

RAW 的相关性指的是在前序的指令还未把结果写回通用寄存器的时候后序的指令就已经需要读取到该通用寄存器了。这时候译码阶段读取到的通用寄存器的值还未更新，因此读取到的通用寄存器的值为错误的，这种情况就会引发 RAW 相关性的数据冲突，本设计为解决 RAW 冲突在硬件中加入了流水线前递 (Forwarding unit) 模块，原理是将在访存阶段与写回阶段的写回的目的通用寄存器与写回的值同时前递至执行阶段，与译码阶段传输过来的通用寄存器编号进行对比，如果有相同情况则使用后级的写回的值。

```
module ForwardCtrl (
    input wire [4:0] rd_l3,
    input wire [4:0] rs1_l2,
    input wire [4:0] rs2_l2,
    input wire [31:0] xrs1_l2,
    input wire [31:0] xrs2_l2,
    input wire [31:0] wval_l3,
    output wire [31:0] alu_a_l2,
    output wire [31:0] alu_b_l2
);

assign alu_a_l2 = ((rs1_l2 == rd_l3) && rs1_l2 != 5'd0) ? wval_l3: xrs1_l2; //当 rs1_l2 == rd_l3 时, wval前递 alu_a,为x0不前递
assign alu_b_l2 = ((rs2_l2 == rd_l3) && rs2_l2 != 5'd0) ? wval_l3: xrs2_l2; //当 rs2_l2 == rd_l3 时, wval前递 alu_b,为x0不前递

endmodule
```

图 12 数据前递模块

如图 12 所示，在流水线前递模块中，将当前的寄存器读数据地址与写寄存器地址进行比较，当两者相同并且不为 0 时，将寄存器写的值递给读取的寄存器数，否则操作数为读寄存器所读取出的对应的寄存器数。

4.3.4. DataRamCtrl 控制模块

(叶明, 徐义人)

在访存阶段中如果是内存访问指令，L (load) 型或 S (store) 型指令，则将对数据存储器进行访问读写，此设计在访存部分的设计较为简单，本设计中的数据存储器采用了四个双端口的 RAM 来充当 Data memory，并将 Data memory 与处理器直接进行连接，并未经过总线进行请求访问，而根据双端口 RAM 特性，

需要提前一个周期就将地址信息进行读取，因此读写数据存储器的地址需要在执行阶段就传输给数据存储器，而不需要在 MeM/WB 寄存器中寄存一个周期，读取到的数据传向下一个阶段进入写回阶段。四个双端口 RAM 能够加大数据存储器的带宽，能够让处理器在一个周期内对 32 位数据进行读写。

```

wire mem_en; //判断地址，只有在0-2047中才对mem进行读写操作
assign mem_en = ~(alu_q_12[31:1]); //对读写的mem地址进行判断，如果前21位中有1则地址大于2047，不在mem地址中，不对mem进行操作
assign addr_0 = alu_q_12[10:0]; //四字字节对齐，每个ram存入地址相同
assign addr_1 = alu_q_12[10:0];
assign addr_2 = alu_q_12[10:0];
assign addr_3 = alu_q_12[10:0];

always @(*) begin
    unique if (ins_op_12 & mem_en) begin //sh指令存一个字节，用地址低两位判断存在哪个ram块中，其余块不写
        case(alu_q_12[1:0])
            3'b00: begin
                {wea_3, wea_2, wea_1, wea_0} = 4'b0001;
                {din_3, din_2, din_1, din_0} = {8'd0, alu_b_12[7:0]};
            end
            2'b01: begin
                {wea_3, wea_2, wea_1, wea_0} = 4'b0010;
                {din_3, din_2, din_1, din_0} = {16'd0, alu_b_12[7:0], 8'd0};
            end
            2'b10: begin
                {wea_3, wea_2, wea_1, wea_0} = 4'b0100;
                {din_3, din_2, din_1, din_0} = {8'd0, alu_b_12[7:0], 16'd0};
            end
            2'b11: begin
                {wea_3, wea_2, wea_1, wea_0} = 4'b1000;
                {din_3, din_2, din_1, din_0} = {alu_b_12[7:0], 24'd0};
            end
            default: begin
                {wea_3, wea_2, wea_1, wea_0} = 4'd0;
                {din_3, din_2, din_1, din_0} = 32'd0;
            end
        endcase
    end else if (ins_op_12 & mem_en) begin //sh指令存一个半字，用地址低两位判断存在哪个ram块中，其余块不写
        case(alu_q_12[1:0])
            3'b00: begin
                {wea_3, wea_2, wea_1, wea_0} = 4'b0011;
                {din_3, din_2, din_1, din_0} = {16'd0, alu_b_12[15:0]};
            end
            2'b01: begin
                {wea_3, wea_2, wea_1, wea_0} = 4'b0101;
                {din_3, din_2, din_1, din_0} = {alu_b_12[15:0], 16'd0};
            end
            default: begin
                {wea_3, wea_2, wea_1, wea_0} = 4'd0;
                {din_3, din_2, din_1, din_0} = 32'd0;
            end
        endcase
    end else if (ins_op_12 & mem_en) begin //sw指令四个ram块都要写入
        {wea_3, wea_2, wea_1, wea_0} = 4'b1111;
        {din_3, din_2, din_1, din_0} = alu_b_12;
    end else begin
        {wea_3, wea_2, wea_1, wea_0} = 4'd0;
        {din_3, din_2, din_1, din_0} = 32'd0;
    end
end
end

```

图 13 数据控制模块

如图 13 所示，数据控制模块首先对读写 RAM 地址范围进行判断，RAM 的地址在 0-2047 中，如果地址超过此范围则不会对存储器进行读写操作。此使能信号通过对地址前 21 位信号进行或操作后取反得到，而 32 位 RISC-V 处理器中的存储计数器对应为 32 位，因此将地址进行四字字节对齐进而消除数据不对齐问题。

图中对不同 STORE 指令进行分类后，使用地址的低两位来对数据存储进行分类，得到四个存储器的使能信号与存入的数据，而 LOAD 指令也是同理，判断读取的为字、半字还是字节后利用读取地址低两位进行读取操作。其中 unique if 操作将 if else 操作化为并行判断，从而能够对时序起到优化作用。

5. 上板测试

(胡云峰)

由于外设总线地址和外设写数据被映射到内存空间 65536 和 65540，所以使用 sw 指令对这片内存取余进行写入就能改变 8 位数码管的显示内容。其中外设总线地址寄存器的高 31 位是地址，低 1 位是时钟信号，所以总线写入函数如下所示：

// 总线通信

```
void __attribute__((noinline)) busWrite(uint32 addr, uint32 data)
{
    uint32 base = 65536;
    asm volatile("sw %0, 4(%1) \n\t"
                  : "+r"(data), "+r"(base));
    addr <<= 1;
    asm volatile("sw %0, 0(%1) \n\t"
                  : "+r"(addr), "+r"(base));
    addr |= 1;
    asm volatile("sw %0, 0(%1) \n\t"
                  : "+r"(addr), "+r"(base));
}
```

规定数码管外设的地址是 123，所以向 8 位数码管写入数据的函数如下

```
void __attribute__((noinline)) setSeg(uint32 data)
{
    busWrite(123, data);
}
```

FPGA 以 50MHz 运行，如果数码管变化过快，肉眼将无法察觉，所以需要需要一个延时函数。

```
#define FREQ 50
#define MICRO_SECOND ((uint32)(1000 / (1000.0 / FREQ)))
#define MILLI_SECOND ((uint32)(1000000 / (1000.0 / FREQ)))
#define SECOND ((uint32)(1000000000 / (1000.0 / FREQ)))
```

// 延迟函数

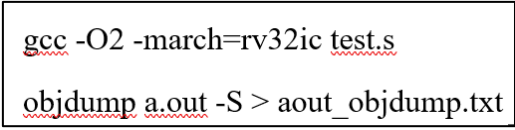
```
void __attribute__((noinline)) delay(uint32 n)
{
    n /= 4;
    if (n == 0)
        return;
    // 每个循环消耗 4 个周期
    asm volatile(
        ".L9889:\n\t"
        "addi %0,%0,-1\n\t"
        "bnez %0,.L9889\n\t"
        : "+r"(n));
}
```

最后我们在数码管上随便显示点什么，就从 0 数到 100，再从 100 数回 0 吧。

```
void __attribute__((noinline)) task1()
{
    // 设置 sp 寄存器
    asm volatile("li sp, 2000\n\t");
}
```

```
while (1)
{
    for (uint32 i = 0; i < 100; i++)
    {
        setSeg(i);
        delay(SECOND);
    }
    for (uint32 i = 100; i > 0; i--)
    {
        setSeg(i);
        delay(SECOND);
    }
}
```

使用支持 riscv 的 gcc 编译器进行编译，生成汇编文件 `gcc -S -O2 -march=rv32ic test.c -o test.s`。在汇编文件中调整 task1 函数的位置，让其位于函数的开头，并对多余的指令进行删改，再进行编译，并使用 `objdump` 对编译生成的 `a.out` 文件进行反汇编，生成测试用例对应的机器码。



```
gcc -O2 -march=rv32ic test.s
objdump a.out -S > aout_objdump.txt
```

图 14 反汇编流程

提取出需要的机器码如附件 1 所示将内容放入 FPGA 的 ram 初始化文件即可开始测试。

6. 总结

本设计以课程所学的 RISC-V 五级流水为基础，扩展了 C 指令。具体在 Decode 模块中增加了压缩指令集的译码，以及对标各标准指令。够满足 MCU 领域的大多数需求，其流水线设计相对简洁，电路面积更小，功耗更低来适配相当存储空间的情况下完成更复杂的功能。

附件一

00010128 <task1>:

10128:	0001	nop
1012a:	0001	nop
1012c:	7d000113	li sp,2000
10130:	02faf4b7	lui s1,0x2faf
10134:	08048493	addi s1,s1,128
10138:	06400913	li s2,100
1013c:	4401	li s0,0
1013e:	8522	mv a0,s0
10140:	2825	jal 10178 <setSeg>
10142:	0405	addi s0,s0,1
10144:	8526	mv a0,s1
10146:	282d	jal 10180 <delay>
10148:	ff241be3	bne s0,s2,1013e <task1+0x16>
1014c:	8522	mv a0,s0
1014e:	202d	jal 10178 <setSeg>
10150:	147d	addi s0,s0,-1
10152:	8526	mv a0,s1
10154:	2035	jal 10180 <delay>
10156:	d07d	beqzs0,1013c <task1+0x14>
10158:	8522	mv a0,s0
1015a:	2839	jal 10178 <setSeg>
1015c:	147d	addi s0,s0,-1
1015e:	8526	mv a0,s1
10160:	2005	jal 10180 <delay>
10162:	f46d	bnezs0,1014c <task1+0x24>
10164:	bfe1	j 1013c <task1+0x14>
10166:	0001	nop

00010168 <busWrite>:

10168:	67c1	lui a5,0x10
1016a:	c3cc	sw a1,4(a5)
1016c:	0506	slli a0,a0,0x1
1016e:	c388	sw a0,0(a5)
10170:	00156513	ori a0,a0,1
10174:	c388	sw a0,0(a5)
10176:	8082	ret

00010178 <setSeg>:

10178:	85aa	mv a1,a0
1017a:	07b00513	li a0,123
1017e:	b7ed	j 10168 <busWrite>

```
00010180 <delay>:
    10180:  478d          li    a5,3
    10182:  00a7f563     bgeua5,a0,1018c <delay+0xc>
    10186:  8109          srli  a0,a0,0x2
    10188:  157d          addi a0,a0,-1
    1018a:  fd7d         bneza0,10188 <delay+0x8>
    1018c:  8082          ret
```