单周期型 32 位 CPU（核）设计报告

团队成员: 杨沁儒, 何荣燊, 李永浩

学号：320220940961、320220940271、320220940401

数据科学四班

我们的 Git 项目地址：

https://github.com/hasaki321/CPU_design/tree/HRS

仓库包含可以直接运行的程序和完整项目代码

2024 年 5 月 12 日

目录

1 概述

本设计基于 MIPS 或 RISC-V 指令集的要求，设计了一个兼容 32 位指令集的单周期 CPU 核。该 CPU 核采用了冯诺伊曼微结构，使用 Verilog 语言进行设计，利用 FPGA 内部的 block RAM 作为指令缓存（iCache）和数据缓存（dCache），指令和数据通过 coe 文件导入缓存。本文将介绍该 CPU 核的设计细节、性能指标以及加分项的实现。

1.1 CPU 核结构

CPU 核结构主要包括指令存储器、控制单元、运算单元、数据存储器以及输入输出接口等模块。其中，指令存储器和数据存储器采用 FPGA 内部的 block RAM 实现指令缓存，用于存储机器码；控制单元负责解析指令并生成控制信号；运算单元执行指令中的运算操作；数据存储器用于存储数据；输入输出接口用于与外部设备进行数据交互。

1.2 性能指标

通过前仿（功能仿真）验证，本设计的 CPU 核成功实现了指定的指令集要求，并给出了性能指标，包括频率、已实现指令条数和 CPI 等。具体性能指标如下：

- 频率：4 MHz
- 已实现指令条数：21 条
- CPI：1

1.3 加分项实现

为了满足加分项的要求，我们使用 C 语言编写了简单程序，覆盖了 CPU 所支持的分指令。然后使用 RISC-V 交叉编译器将程序编译为汇编源码，并通过 RISC-V 将其翻译为机器指令并执行。最终成功演示了加分项的实现。

1.4 提交材料

为了完成本设计，我们准备了以下提交材料：

- 详细设计文档 (design_doc.zip)
- 软硬件设计工程 (design.zip)

2 基础内容

2.1 性能指标

我们的 cpu 有两种周期，一种是普通取指周期即 clk，另一种是取数据周期 clk_mem。

2.1.1 周期与频率

经过不断的调试，我们能够达到的最短取指周期是 0.25ns，即我们的 cpu 最高频率可以达到 4MHz。

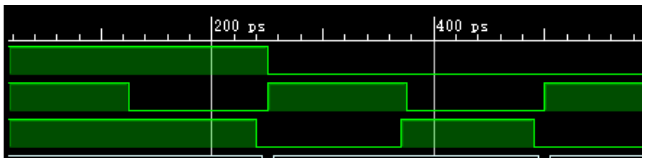

Figure 2.1: clk 周期

2.1.2 CPI

单周期 CPU 的 CPI（每条指令的时钟周期数）通常是 1，因为每条指令都在一个时钟周期内完成。

2.1.3 运作路线

单周期 CPU 指的是一条指令的执行在一个时钟周期内完成，然后开始下一条指令的执行，即一条指令用一个时钟周期完成。
在我们的 cpu 处理指令时，需要经过如下五个阶段：

- 取指令 (IF)：根据程序计数器 PC 中的指令地址，从存储器中取出一条指令，同时，PC 根据指令字长度自动递增产生下一条指令所需要的指令地址，但遇到“地址转移”指令时，则控制器把“转移地址”送入 PC，当然得到的“地址”需要做些变换才送入 PC。
- 指令译码 (ID)：对取指令操作中得到的指令进行分析并译码，确定这条指令需要完成的操作，从而产生相应的操作控制信号，用于驱动执行状态中的各种操作。
- 指令执行 (EXE)：根据指令译码得到的操作控制信号，具体地执行指令动作，然后转移到结果写回状态。

- 存储器访问 (MEM)：所有需要访问存储器的操作都将在这个步骤中执行，该步骤给出存储器的数据地址，把数据写入到存储器中数据地址所指定的存储单元或者从存储器中得到数据地址单元中的数据。
- 结果写回 (WB)：指令执行的结果或者访问存储器中得到的数据写回相应的目的寄存器中。

为了加快 cpu 的工作速度，我们设置了一个 clk_mem 来控制数据的存储。我们设置了在 clk_mem 的上升边沿进行存数据，在下降边沿进行存寄存器，clk_mem 的上升边沿与 clk 的上升边沿有一定的延时。
(周期控制详见`design/design_codes/test_clk.v`)

2.2 实现的指令集

我们采用 RISC-V32I 指令集，并且实现的指令覆盖了 R 型、I 型、S 型、B 型、U 型、J 型指令。其中：

- R 类型指令：用于寄存器 - 寄存器操作；
- I 类型指令：用于短立即数和访存 load 操作；
- S 类型指令：用于访存 store 操作；
- B 类型指令：用于条件跳转操作；
- U 类型指令：用于长立即数操作；
- J 类型指令：用于无条件操作；

为了能够成功适应编译出来的 c 代码，我们一共实现了 21 条指令，具体如表 2.1

2.3 CPU 设计

2.3.1 模块设计

我们采用哈佛结构设计 CPU，包括以下模块：RegFile、InstrMem、DataMem、ID、PC、ALU、ControlUnit。为了页面内容整洁简短我们此处仅详细介绍 alu。
(详细设计内容请见`/design_doc`，详细模块图以及接口说明请参照附录)

- **ControlUnit：**控制单元模块，协调 CPU 中的各个部件，控制指令的执行和数据流动；

type	instr	description
R	add	$rd = rs1 + rs2$
R	sub	$rd = rs1 - rs2$
R	sll	$rd = rs1 \ll rs2$
R	sra	$rd = rs1 \gg rs2$
R	add	$rd = rs1 + \text{imm}$
I	andi	$rd = rs1\&\text{imm}$
I	slti	$rd = (rs1 < \text{imm})$
IL	lw	$rd = M[rs1 + \text{imm}][0:31]$
IL	lb	$rd = M[rs1 + \text{imm}][0:7]$
IL	lbu	$rd = \text{unsigned}(M[rs1 + \text{imm}][0:7])$
S	sw	$M[rs1 + \text{imm}][0:31] = rs2[0:31]$
S	sb	$M[rs1 + \text{imm}][0:7] = rs2[0:7]$
B	beq	if $(rs1 == rs2)$ then $pc = pc + \text{imm}$
B	bne	if $(rs1 \neq rs2)$ then $pc = pc + \text{imm}$
B	blt	if $(rs1 < rs2)$ then $pc = pc + \text{imm}$
B	bltu	if $(\text{unsigned}(rs1) < \text{unsigned}(rs2))$ then $pc = pc + \text{imm}$
B	bgeu	if $(\text{unsigned}(rs1) \geq \text{unsigned}(rs2))$ then $pc = pc + \text{imm}$
U	auipc	$rd = pc + \text{imm}$
U	lui	$rd = \text{imm} \ll 12$
J	jal	$rd = pc + 4; pc = pc + \text{imm}$
JR	jalr	$rd = pc + 4; pc = rs1 + \text{imm}$

Table 2.1: Instruction Set

- **PC 4.1：**程序计数器模块，实现程序计数器的更新逻辑，并从指令存储器中读取指令；
- **InstrMem 4.1：**指令内存模块，它接收指令读取地址作为输入，并输出对应地址处的指令数据。指令存储器用于存储程序的指令序列。
- **ID 4.2：**指令解码器（Instruction Decoder）模块，用于从指令中解析出各种控制信号，操作数地址以及拓展后的立即数；
- **RegFile 4.3：**寄存器模块，用于存储、读写指令执行过程中的数据和中间结果；由寄存器读使能和寄存器写使能信号判断是否读取或者写入数据，输入为 rs1 地址，rs2 地址，rd 地址，输出为 rs1 数据，rs2 数据。
- **DataMem 4.5：**数据内存模块，由写使能和读使能信号判断是否读写数据，输入为地址，输出数据；

- **ALU：**算术逻辑单元

1). 为了让 alu 能够根据不同的指令来进行不同的操作，这里需要根据我们实现的指令集定义 alu 操作码 2.3.1。

2). 大部分的时候我们需要对 input1 和 input2 进行操作，因此这时候只需要一个简单的 aluout 输出就足够了。

3). 根据指令的不同，alu 的第二个输入有可能是来自寄存器 2 的数据或者是立即数 imm，我们需要一个判断信号 addImm 来判断 alu 的第二个输入应该是什么。

4). alu 需要根据输入 branch 判断此次操作是否是分支操作并且提供分支操作码 2.3.1，alu 在完成分支判断后需要判断是否跳转，因此此次需要输出一个跳转指令 jump。

5). 在需要跳转的时候，我们需要获取跳转到的 pc 立即数，这时根据指令的不同例如 auipc 指令，有可能要求输出 rs1 的数据或者 pc。因此我们需要一个 pc 输入和一个 pc 输出。

为了方便理解，我们提供了 ALU 的模块线路图 2.2。

operate	**aluctr**
add	001
sub	010
and	011
sll	100
sra	101
addpci	110
addpc	111

operate	**alub**
beq	001
bne	010
blt	011
bltu	100
bgeu	101

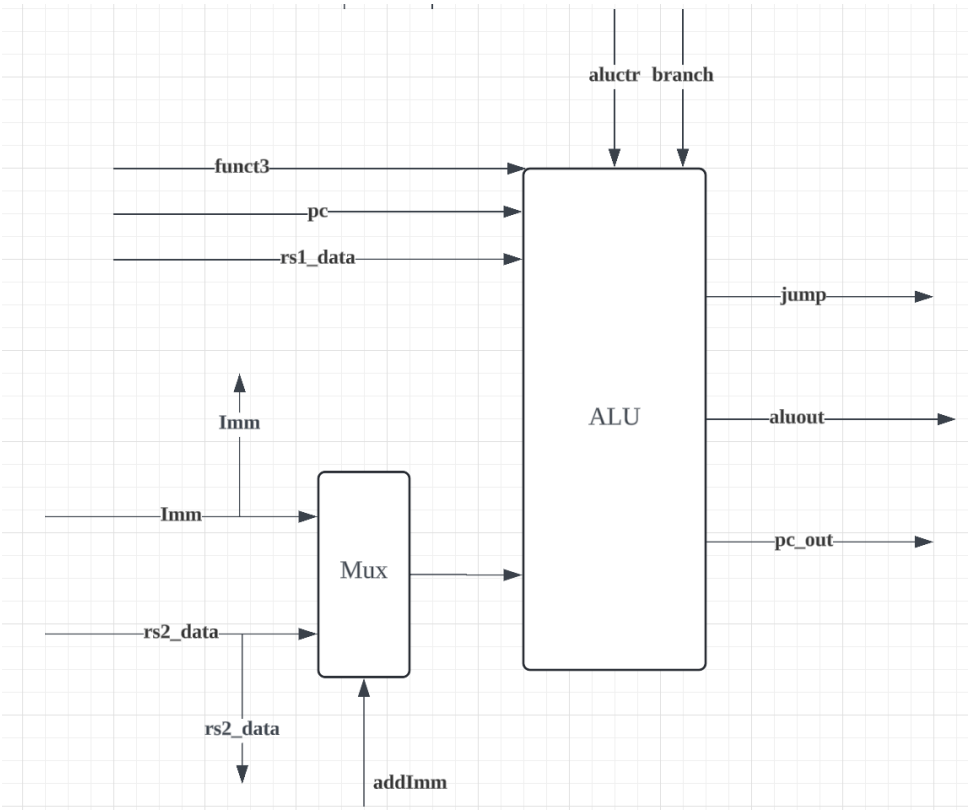

Figure 2.2: ALU 模块

2.3.2 总数据通路

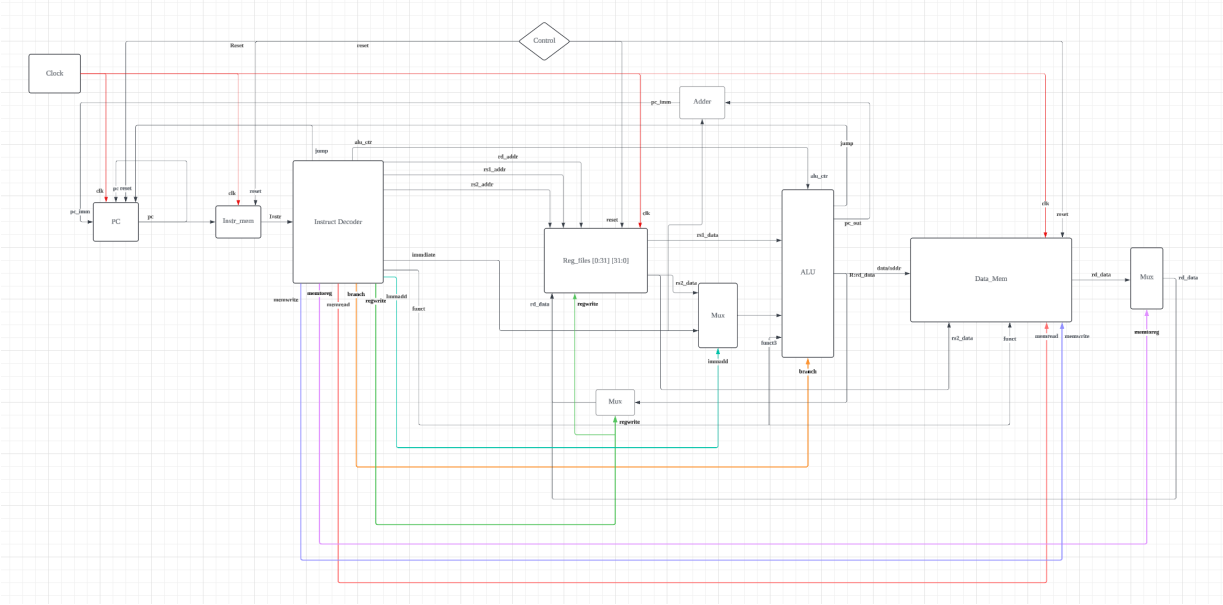

Figure 2.3: 总数据通路

2.4 波形验证

我们的指令存储器和数据存储器采用 FPGA 内部的 block RAM 实现指令缓存。

2.4.1 数据存储器

设置了一些特殊的数据来验证 `lb` 和 `lh` 能否正常运行, 我们使用了双端口 RAM 内存来存储这些数据。(coe 文件详见`/design/design_datas/data_test.coe`)

地址	数据
0x00	00000000000000000000011100000100
0x04	00000000000011100000000000001100
0x08	00000000000000000000000000011100
0x0c	00000000000000000000000000111100

Table 2.2: 数据存储器示例

2.4.2 指令存储器

我们设计了一些指令来验证我们的 cpu 能否正常工作，我们使用了单端口 ROM 来存储这些指令。(coe 文件详见`/design/design_datas/instruce_test.coe`)
指令存储器内容如表 2.3：

2.4.3 波形图

(波形验证的结果详见`/design/design_output/test_clk_behav.wcfg`)

从仿真图 2.4 (可能不是很清晰，大图请查看 design_output)，中我们可以看到，每一条标记对应一个指令，我们一条条讲解。开始时是 reset 信号，pc 为 `0xfffffffffffffffc`，reset 为 0 时 pc+4，此时 pc 为 0 开始执行指令。

- 第一条：
  - pc 值为 0 时读出的指令是 0203，可以确认取指令的过程没有问题。
  - 寄存器 `rs1` 的地址是 00，`rd` 的地址是 04，同时立即数是 0，我们可以确认部分指令译码没有问题。
  - 经过 `alu` 运算后 `alu` 输出地址 00，去 `datamem` 取值，可以看到 `read_data` 值在为 04，符合我们的预期，我们可以确认读字节过程没有问题。

Table 2.3: 指令表

指令	**注释**
IL	
000000000000 00000 000 00100 0000011	从寄存器 0 地址偏移零个字节加载字节到寄存器 4
000000000100 00000 001 00101 0000011	从寄存器 0 地址偏移四个字节加载半字到寄存器 5
000000001000 00000 010 00110 0000011	从寄存器 0 地址偏移八个字节加载半字到寄存器 6
S	
0000000 00110 00000 010 01100 0100011	将寄存器 6 的一个字存进寄存器 0 偏移 12 个字节的地址
IL	
000000001100 00000 000 00111 0000011	从寄存器 0 地址偏移 12 个字节加载字节到寄存器 7
R	
0000000 00110 00101 000 01000 0110011	将寄存器 5 的值加寄存器 6 的值存进寄存器 8 中
0000000 00110 00101 000 01001 0110011	将寄存器 5 的值加寄存器 6 的值存进寄存器 9 中
B	
0000000 01000 01001 000 01000 1100011	判断寄存器 8 和寄存器 9 的值，跳转至下 2 条指令
I	
000000001100 00110 000 01010 0010011	将寄存器 6 的数加 1100 存进寄存器 10 中
000000001000 00110 000 01010 0010011	将寄存器 6 的数加 1000 存进寄存器 10 中

Figure 2.4: 波形图

- 第二条：
  - 从 `rs1` 的数据偏移 4 字节，我们可以看到 `alu` 输出的地址为 04，去 04 地址取值，可以看到在一段时间后 `read_data` 稳定为 1100，符合预期，取半字也没有问题。
- 第三条：
  - 同样的，我们可以发现 `read_data` 为 11100，取字也没有问题。
- 第四条：
  - 我们将寄存器 6 的一个字存进寄存器 0 偏移 12 个字节的地址，`w_data` 的数据为 11100，在写入之前的流程一切正常。
- 第五条：
  - 从寄存器 0 偏移 12 个字节取一个字的数据，我们可以发现读出的数据为 11100，我们可以确认第四条成功存入了数据。
- 第六条：
  - 将寄存器 5 的值加寄存器 6 的值存进寄存器 8 中，即 1100+11100，预期输出为 101000，即 0x28，我们可以看到 `alu` 的输出结果为 0028，可以证明 `alu` 运算部分没有问题。
- 第七条：

  - 与第六条一致，存进寄存器 9 中。

- 第八条：
  - 比较寄存器 8 和 9 的值，若一致则跳转至下两条指令。我们可以发现 `input_1` 和 `input_2` 输入的值都是 28，这也验证了上面两条的寄存器写回步骤是没有问题的，同时 `jump` 变成了 1，这能说明 `alu` 成功做出了比较判断。

- 第十条：
  - 我们可以发现此时 pc 值由上一条的 1c 变成了 24，说明跳转成功了，这也验证了 CPU 的跳转功能是正常的。
  - 这是一条 `addi` 指令，我们可以发现 `imm` 的值是 1000，这说明取的指令是正确的。该操作要求寄存器 6 的值加立即数并写回寄存器 10 中。我们可以观察到 `alu` 的结果是 0024，这说明立即数加法也是正常的。

综上，我们手写的机器码验证全部通过，前仿正常。

3 附加项目

项目的测试代码位于design/test_c

3.1 编译工具链

(https://github.com/riscv-collab/riscv-gnu-toolchain)
我们从 github 克隆了 riscv-gnu-toolchain 交叉工具链, 并编译了 riscv32i 指令集，具体流程如下：

Listing 3.1: 编译工具链

```
#下载依赖
$ sudo apt-get install autoconf automake autotools-dev curl python3 python3-
libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo \
gperf libtool patchutils bc zlib1g-dev libexpat-dev ninja-build git cmake \
libglib2.0-dev libslirp-dev
#克隆仓库
$ git clone https://github.com/riscv/riscv-gnu-toolchain
$ cd riscv-gnu-toolchain
$ git submodule update --init --recursive
#设置变量
$ RISCV=$HOME/riscv-gnu-toolchain
$ PATH=$PATH:$RISCV/bin
#编译
$ ./configure --prefix=$RISCV --with-arch=rv32i --with-abi=ilp32
$ make
```

3.2 C

我们写了一小段 c 代码来测试我们的 cpu：

Listing 3.2: 测试代码

```
int main() {
    int tmp = 0;
    int list[] = {1, 2, 3, 4, 5};
```

```
    for (int i = 0; i < 5; i++) {
        list[i] = list[i] << 2;
        tmp += list[i];
    }
    return 0;
}
```

3.3 编译

编译流程具体如下：

Listing 3.3: 汇编 C 程序

```
$ riscv32-unknown-linux-gnu-gcc -S test_code.c test_code.s
$ riscv32-unknown-linux-gnu-objdump test_code.s > test_code_S.txt
###用python提取出16位机械码
$ python hex.py test_code_S.txt > out.txt
```

编译出来的结果过长，请移步工程文件或者附录查看 4.1。

3.4 仿真测试

我们直接将 out.txt 中的机器码写入.coe 文件中并载入 instrMem 进行仿真测试。
我们的设计的 cpu 支持指令覆盖了该 c 语言代码中的所需的所有指令。如图 3.1 , 我们的 cpu 能够正常的运行编译后的 c 程序，因为机器码过长我们仅挑两行指令分析，波形图大图请查看`design/design_outputs/testc_behav.png`

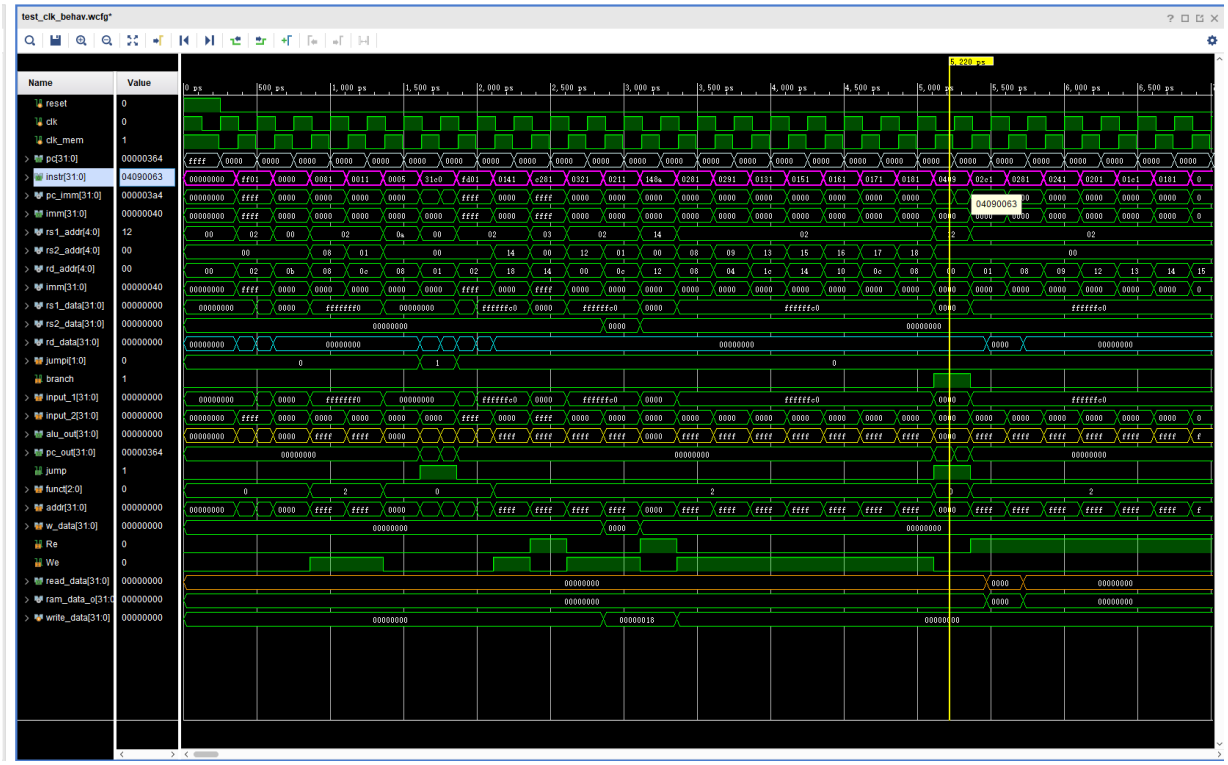

Figure 3.1: 波形图 _C

- ff010113

  10094: ff010113 add sp,sp,-16

  该指令是程序的第一条指令，为操作分配初始地址，我们可以从图中观察到 alu 输出 ffffff0, 该操作执行成功。

- 31c000ef

  100a8: 31c000ef jal 103c4 <___call_exitprocs>

  该指令是一条入口立即跳转指令，我们可以从图中观察到 alu 给出了整个图中第一条 jump 信号,观察到下一条指令为 fd010113,我们可以得知 pc 跳转到了 <___call_exitprocs> 标签中的第一条指令。

- 04090063

  103f8: 04090063 beqz s2,10438 <___call_exitprocs+0x74>

  该指令是一条分支判断指令，也是整个图中第一条分支判断指令，我们可以从图中看到 branch 信号为 1，同时 alu 给出了跳转判断 jump=1，观察到下一条指令为 02c12083，我们可以判断其跳转到了 <___call_exitprocs> 标签中的一串取数据 lw 序列。

4 附录

4.1 模块设计

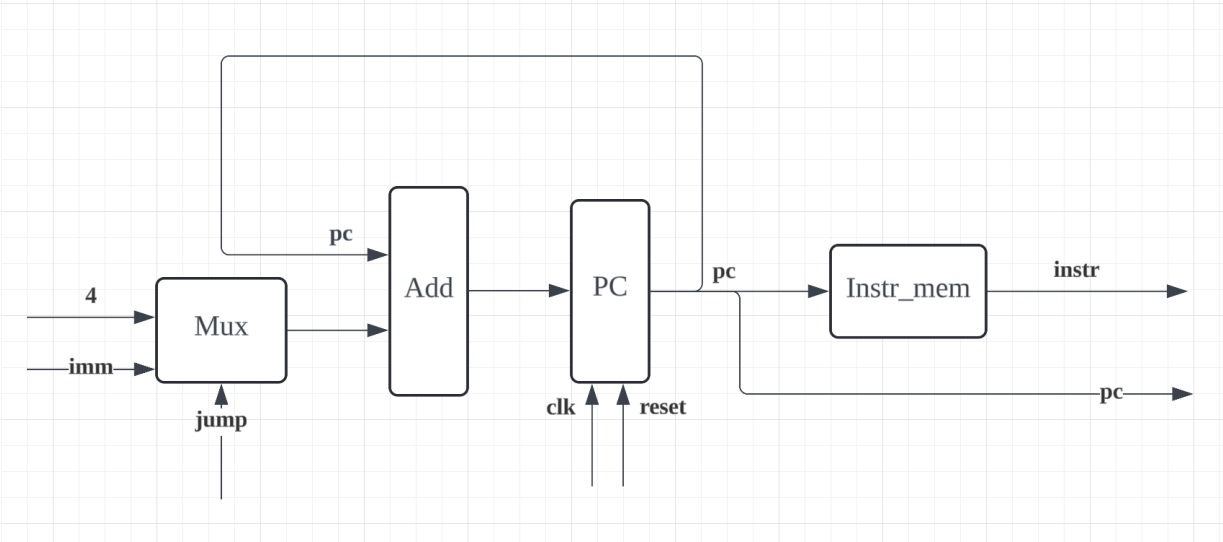

Figure 4.1: PC

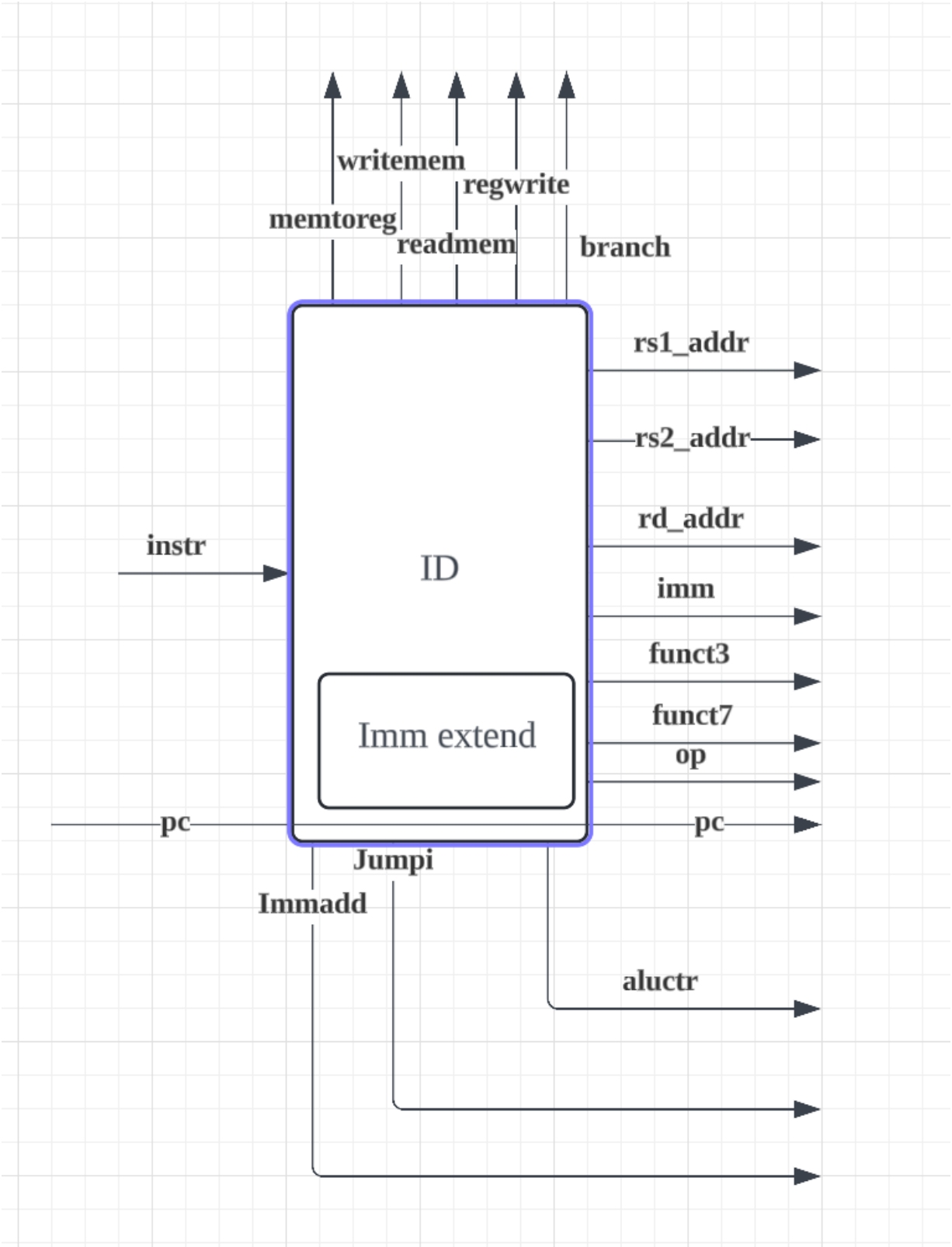
writemem
regwrite
memtoreg
readmem
branch
rs1_addr
rs2_addr
rd_addr
instr
ID
imm
funct3
Imm extend
funct7
op
pc
pc
Jumpi
Immadd
aluctr

Figure 4.2: ID

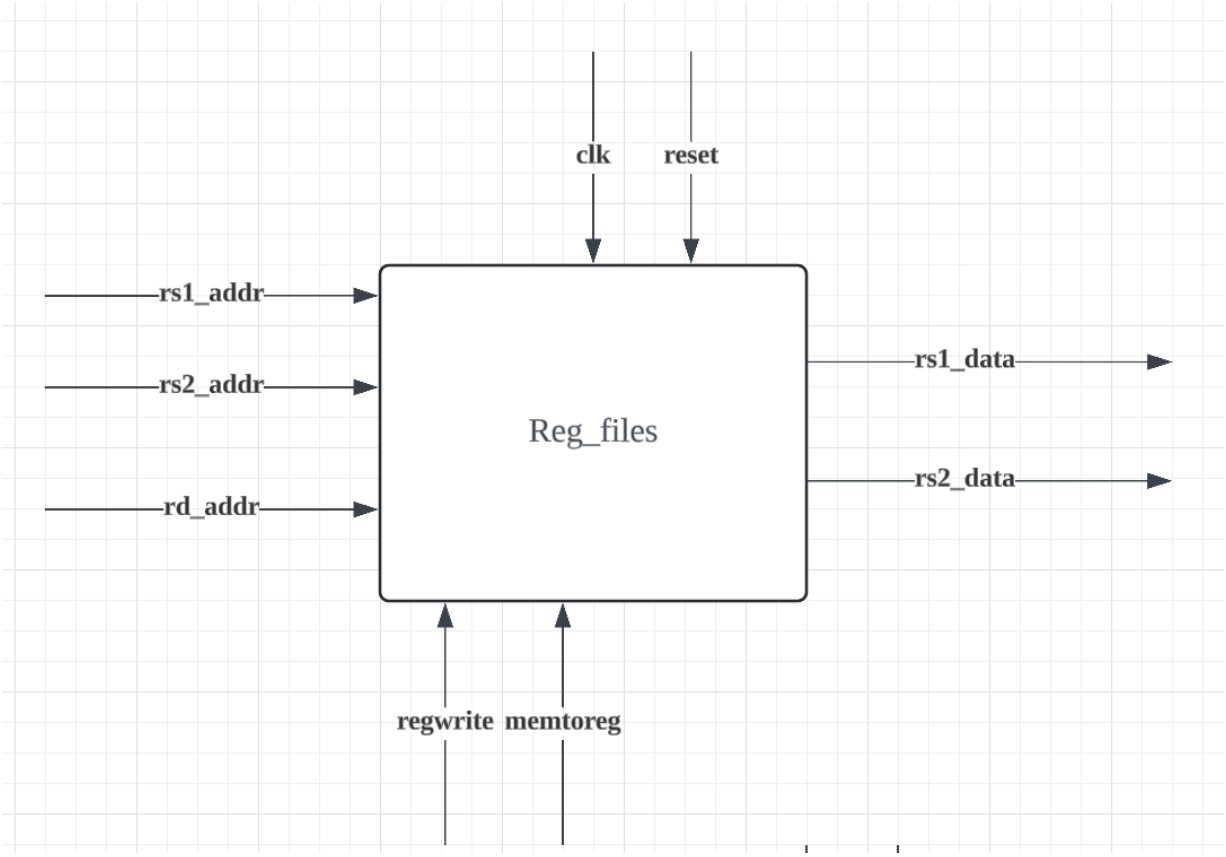

Figure 4.3: RegFiles

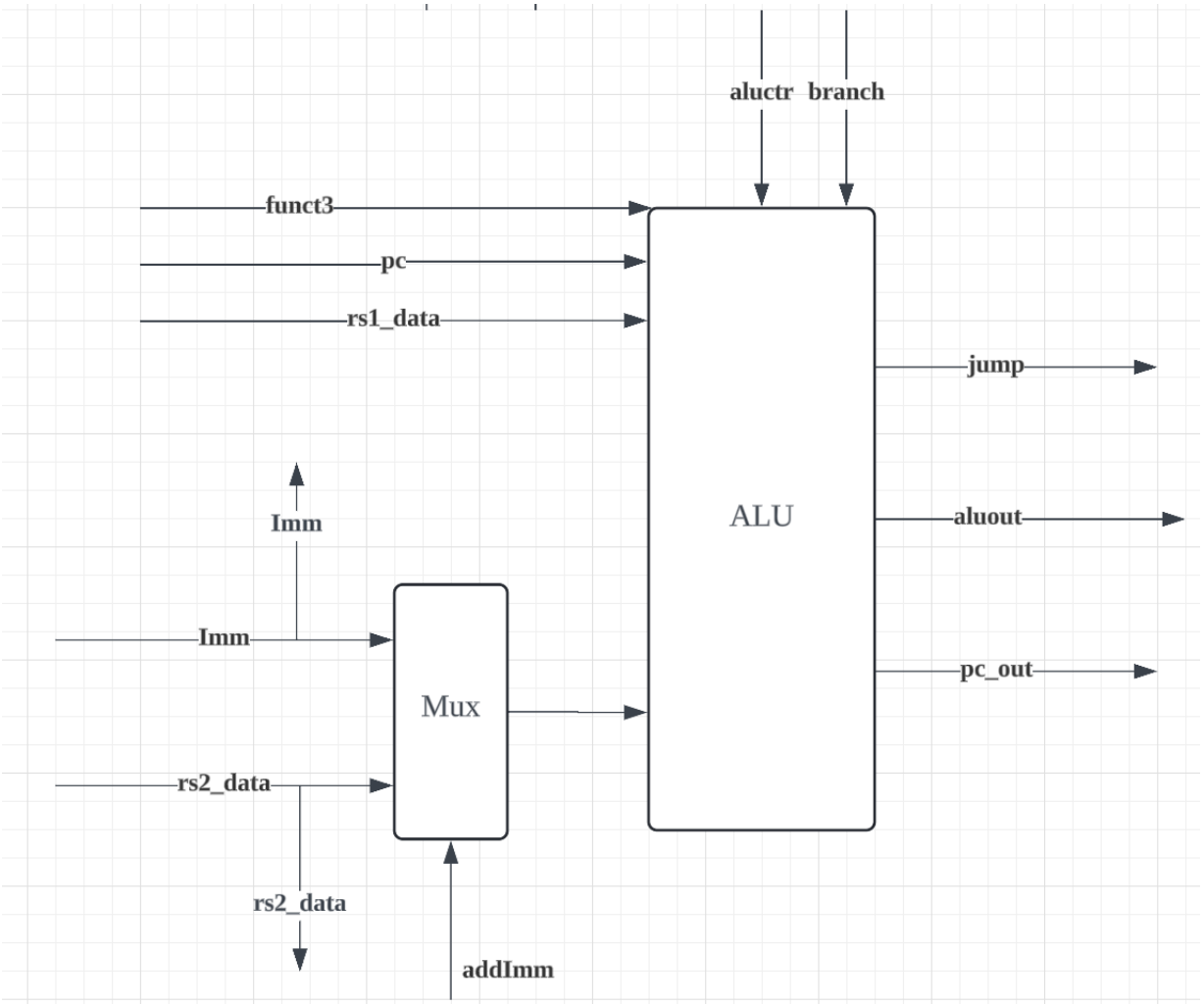

Figure 4.4: ALU

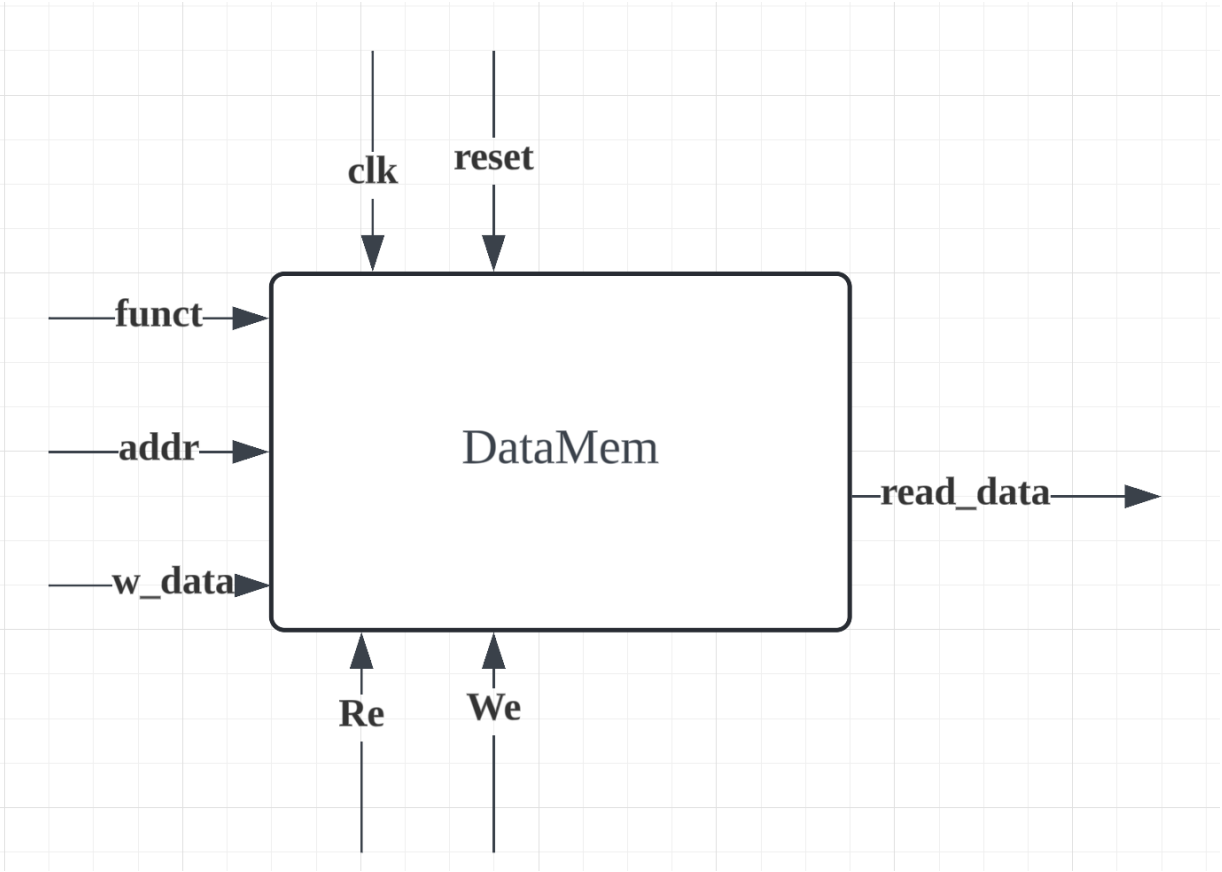

Figure 4.5: DataMem

4.2 汇编结果

Listing 4.1: 汇编代码

```
        .file   "test_code.c"
        .option nopic
        .attribute arch, "rv32i2p1"
        .attribute unaligned_access, 0
        .attribute stack_align, 16
        .text
        .section        .rodata
        .align  2
.LC0:
        .word   1
        .word   2
        .word   3
        .word   4
        .word   5
        .text
        .align  2
        .globl  main
        .type   main, @function
main:
        addi    sp,sp,-48
        sw      s0,44(sp)
        addi    s0,sp,48
        sw      zero,-20(s0)
        lui     a5,%hi(.LC0)
        addi    a5,a5,%lo(.LC0)
        lw      a1,0(a5)
        lw      a2,4(a5)
        lw      a3,8(a5)
        lw      a4,12(a5)
        lw      a5,16(a5)
        sw      a1,-44(s0)
        sw      a2,-40(s0)
```

```
        sw      a3,-36(s0)
        sw      a4,-32(s0)
        sw      a5,-28(s0)
        sw      zero,-24(s0)
        j       .L2
.L3:
        lw      a5,-24(s0)
        slli    a5,a5,2
        addi    a5,a5,-16
        add     a5,a5,s0
        lw      a5,-28(a5)
        slli    a4,a5,2
        lw      a5,-24(s0)
        slli    a5,a5,2
        addi    a5,a5,-16
        add     a5,a5,s0
        sw      a4,-28(a5)
        lw      a5,-24(s0)
        slli    a5,a5,2
        addi    a5,a5,-16
        add     a5,a5,s0
        lw      a5,-28(a5)
        lw      a4,-20(s0)
        add     a5,a4,a5
        sw      a5,-20(s0)
        lw      a5,-24(s0)
        addi    a5,a5,1
        sw      a5,-24(s0)
.L2:
        lw      a4,-24(s0)
        li      a5,4
        ble     a4,a5,.L3
        li      a5,0
        mv      a0,a5
        lw      s0,44(sp)
        addi    sp,sp,48
```

```
jr      ra
.size   main, .-main
.ident  "GCC: (g3dab8f8a542-dirty) 12.2.0"
```