

课程实验报告

RISC-V on T-Core

MaTrixV Team

目录

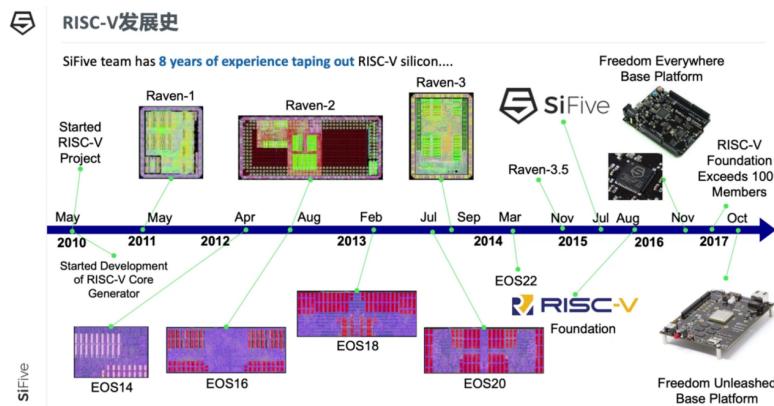
1.	基础篇	2
1.1	RISC-V 简介	2
1.2	蜂鸟 E203 简介	10
1.3	T-core 开发板介绍	11
2.	实践篇	12
2.1	硬件开发	12
2.2	软件开发	12
3.	结果展示	21
4.	未来展望	21

1. 基础篇

1.1 RISC-V 简介

RISC-V 发展过程

1. RISC(精简指令集计算机) 和 CISC(复杂指令集计算机) 是当前 CPU 的两种架构。早些年，市面上只有 CISC 指令集，后来 IBM 的研究员通过统计的方法发现，传统 CISC 处理器中，五分之一的指令承担了五分之四的工作，而剩下五分之四的指令基本没有被使用，或者很少使用，这样，既浪费了 CPU 的核心面积，增大了功耗，还降低了效率。于是，RISC 应运而生。
2. RISC 的指令数目较 CISC 少，CISC 中的一些复杂指令，RISC 需要用多条简单指令来实现。但指令字等长，效率高，功耗低，并发性高。且内部寄存器丰富，更强调对寄存器的合理调用。但高性能 RISC 处理器成本高，性价比低，且不同公司的 RISC 芯片几乎无法通用，生态环境较 X86 的 CISC 而言更闭塞，通用性完全无法和 X86 相比，这就是 RISC 最大的弊端。
3. 20 世纪末和 21 世纪初，市面上绝大多数核心指令集都是不开源的。2010 年，加州大学伯克利分校的 David A. Patterson 教授团队在 3 个月内开发出完全开源指令集 RISC-V，RISC-V 指令集是基于精简指令集计算 (RISC) 原理建立的开放指令集架构 (ISA)，RISC-V 是在指令集不断发展和成熟的基础上建立的全新指令。RISC-V 指令集完全开源，设计简单，易于移植 Unix 系统，模块化设计，完整工具链，同时有大量的开源实现和流片案例，已在社区得到大力支持。
4. 它虽然不是第一个开源的的指令集 (ISA)，但它是第一个被设计成可以根据具体场景可以选择适合的指令集的指令集架构。基于 RISC-V 指令集架构可以设计服务器 CPU、家用电器 CPU、工控 CPU 和传感器中的 CPU 等。



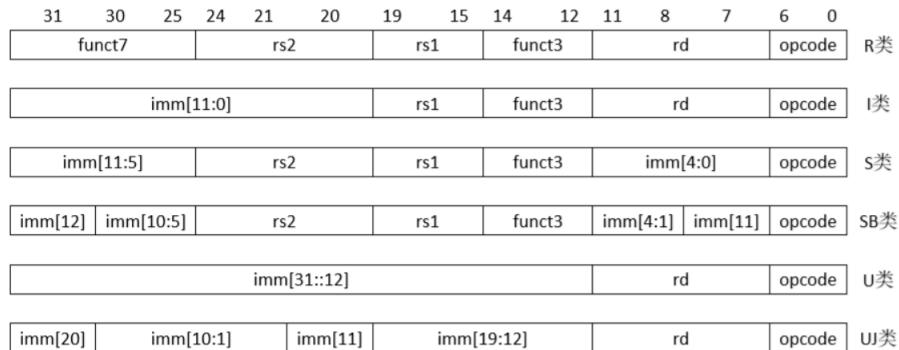
RISC-V 指令简述

1. RSICV 指令集分为基本指令集 I 和扩展指令集 M, A, F, D, C。基本指令集 I 是整数指令集，也是 RISC-V 中，对于任何处理器必须有的指令集，扩展指令集可有可无。

基本指令集	指令数	描述
RV32I	47	32位地址空间与整数指令，支持32个通用整数寄存器
RV32E	47	RV32I的子集，仅支持16个通用整数寄存器
RV64I	59	64位地址空间与整数指令及一部分32位的整数指令
RV128I	71	128位地址空间与整数指令及一部分64位和32位的指令
扩展指令集	指令数	描述
M	8	整数乘法与除法指令
A	11	存储器原子 (Atomic) 操作指令和Load-Reserved/Store-Conditional指令
F	26	单精度 (32比特) 浮点指令
D	26	双精度 (64比特) 浮点指令，必须支持F扩展指令

2. 基本指令集有六种格式：

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

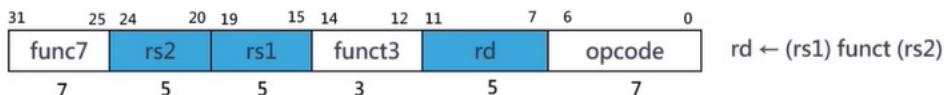


RISC-V 指令分述

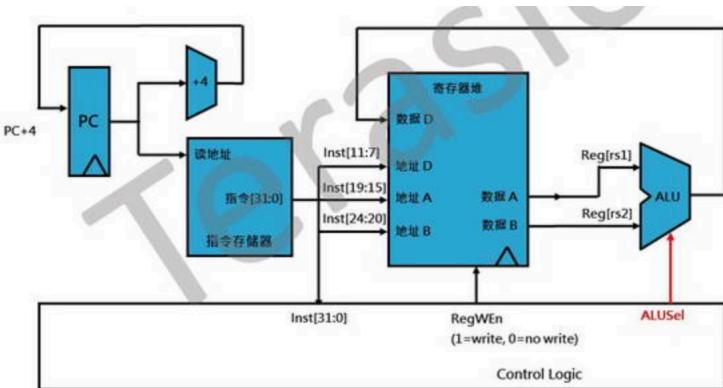
1. R 型指令：

(a) 简介：

R 型指令包含简单的运算指令，由两个 32 位源寄存器 (rs1, rs2) 进行运算，得到的结果存入一个 32 位目的寄存器 (rd)。R 型指令的 6 位操作码 (opcode) 是固定的，由功能码 (func7 和 func3) 确定是何种运算。



(b) 数据通路 & 指令执行流程：



- i. 首先按照 PC 中的地址，从指令存储器中读出 32 位指令，然后 $PC+4$ ；
- ii. 指令被送入控制器，控制器根据功能码和操作码确定这是 R 型指令中的某一条指令，并产生控制信号 ALUSel 和 RegWEn；

- iii. 将指令的 rs1、rs2、rd 字段分别送到寄存器堆的相应地址端，寄存器根据 rs1 和 rs2 字段的地址读出对应的寄存器的值 Reg[rs1] 和 Reg[rs2];
- iv. ALU 根据控制信号 ALUSel，对 Reg[rs1] 和 Reg[rs2] 进行相应的运算，得到运算结果；
- v. 将运算结果送到寄存器堆的数据 D 端口，控制器产生的 RegWEn 信号使寄存器堆允许写入，寄存器堆根据地址 D 端口的地址将数据 D 端口的值写入相应的寄存器。

(c) 指令格式：

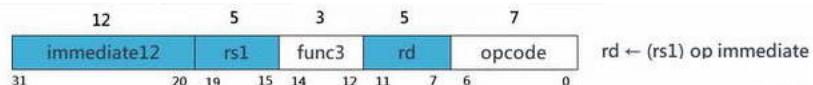
func7	rs2	rs1	func3	rd	opcode	指令
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

- 算数运算：加法 add、减法 sub
- 逻辑运算：与 and、或 or、异或 xor
- 比较运算：有符号小于比较 slt、无符号小于比较 sltu
- 移位运算：逻辑左移 sll、逻辑右移 srl、算数右移 sra

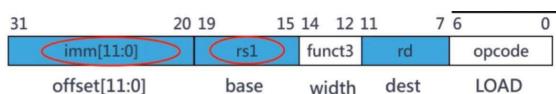
2. I 型指令：

(a) 简介：

I 型指令包含简单的立即数运算指令和 load 指令。立即数运算指令是由一个 32 位源寄存器 (rs1) 和指令中包含的立即数（经扩充后的）进行运算，将得到的结果存入一个 32 位目的寄存器 (rd)。I 型指令中的立即数运算指令的 6 位操作码 (opcode) 固定，由功能码 (func7 和 func3) 确定是何种运算。

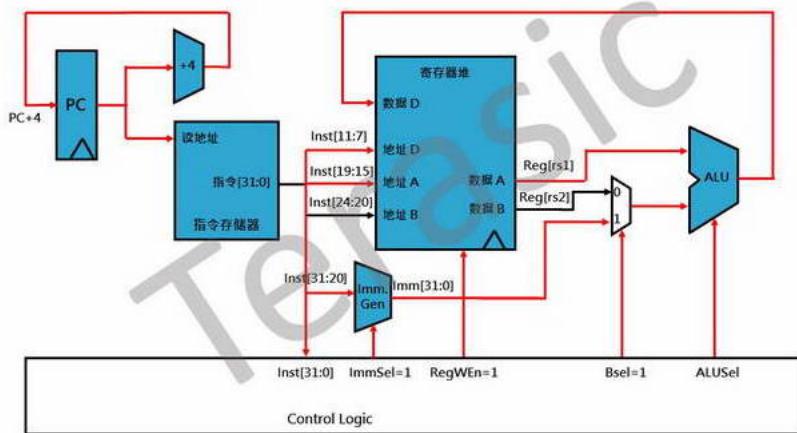


Load 指令属于 I - type 型，其功能是完成存储器的读操作。func3 字段表示读出存储器的数据的字长（32 位、16 位或 8 位）。Load 指令将 rs1 中的数据与 12 位立即数字段符号扩展相加所得到的结果作为访问存储器的地址，即存储器地址 = Reg[rs1] + imm。接着，将从存储器读出的数据存入目的寄存器 rd 中。



(b) 数据通路 & 指令执行流程：

- i. 首先按照 PC 中的地址，从指令存储器中读出 32 位指令，然后 PC+4；



- ii. 指令被送入控制器，控制器根据功能码和操作码确定这是 I 型指令中的某一条指令，并产生控制信号 ALUSel、ImmSel 和 RegWEn；
- iii. 将指令的 rs1、rs2、rd 字段分别送到寄存器堆的相应地址端，寄存器根据 rs1 和 rs2 字段的地址读出对应的寄存器的值 Reg[rs1] 和 Reg[rs2]（但此时 Reg[rs2] 没有用）；
- iv. 将指令的 12 位立即数字段送到立即数扩展电路，通过 ImmSel 信号的指示，将立即数扩展为 32 位，并送到多路选择器的一个端口；
- v. Bsel 信号，控制多路选择器选择立即数的那个端口，将立即数传送到 ALU 的一个输入端；
- vi. ALU 根据控制信号 ALUSel，对 Reg[rs1] 和扩展后的立即数进行相应运算，得到运算结果；
- vii. 将运算结果送到寄存器堆的数据 D 端口，寄存器堆根据地址 D 端口的值将运算结果写入相应的寄存器。

(c) 指令格式：

imm[11:0]	rs1	funct3	rd	opcode	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	111	rd	0010011	ANDI
0000000	shamt	rs1	001	0010011	SLLI
0000000	shamt	rs1	101	0010011	SRLI
0100000	shamt	rs1	101	0010011	SRAI
imm[11:0]	rs1	000	rd	0000011	LB
imm[11:0]	rs1	001	rd	0000011	LH
imm[11:0]	rs1	101	rd	0000011	LHU

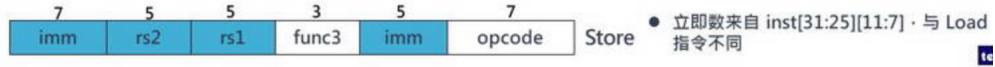
- 算数运算：加法 addi
- 逻辑运算：与 andi、或 ori、异或 xor
- 比较运算：有符号小于比较 slti、无符号小于比较 sltu
- 移位运算：逻辑左移 slli、逻辑右移 srli、算数右移 srai

3. S 型指令：

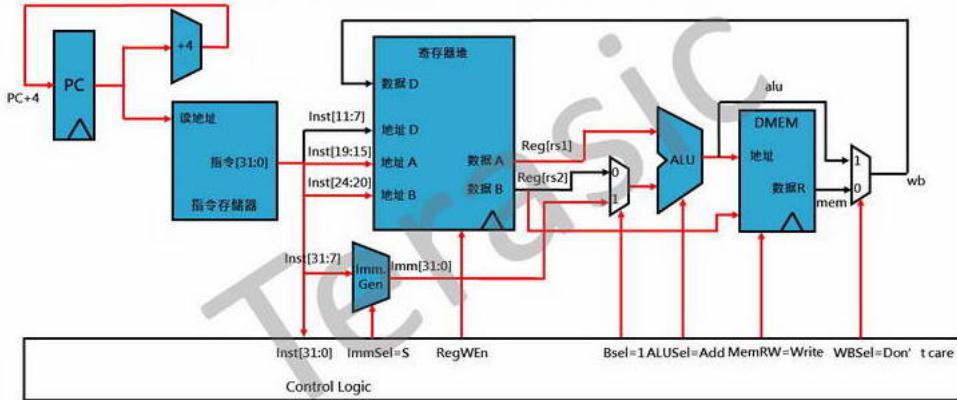
(a) 简介：

S - type 型指令包括 Store 指令，其功能是完成存储器的写操作。Store 使用 funct3 字段选择 Word, HalfWord, Byte。Store 指令将 rs1 中的数据与 12 位立即数字段符号扩展相加所得到的结果

作为访问存储器的地址，即存储器地址 = $\text{Reg}[rs1] + \text{imm}$ 。接着，将 $rs2$ 中的数据写入存储器相应地址中。



(b) 数据通路 & 指令执行流程:



- 首先按照 PC 中的地址，从指令存储器中读出 32 位指令，然后 $PC+4$ ；
- 指令中的控制码和功能码送到控制器中产生相应的控制信号；
- 将 $rs1$ 和 $rs2$ 的地址送入寄存器堆，得到 $rs1$ 寄存器的值 $\text{Reg}[rs1]$ ，作为 ALU 的第一个输入端；
 $rs2$ 寄存器的值 $\text{Reg}[rs2]$ ，作为 DMEM 写入地址的数据；
- 将 12 位的立即数进行符号扩展，得到 32 位的立即数作为 ALU 的第二个输入端。此处控制信号 $Bsel=1$ ，表示指令的立即数部分有效，二选一选择器输出立即数的内容；
- 控制信号 $ALUSel = Add$ ，指示 ALU 做加法运算，得到的地址送入数据存储器 DMEM 的地址接口，并注意此时控制信号 $MemRW = Write$ ，使得数据存储器处于写状态；
- DMEM 接收到地址和数据后，将所选地址空间写入 $\text{Reg}[rs2]$ 。

(c) 指令格式:

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	指令
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

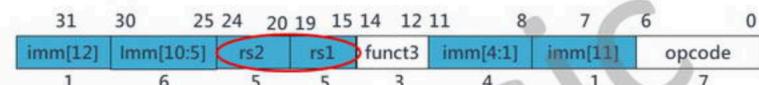
4. B 型指令:

(a) 简介:

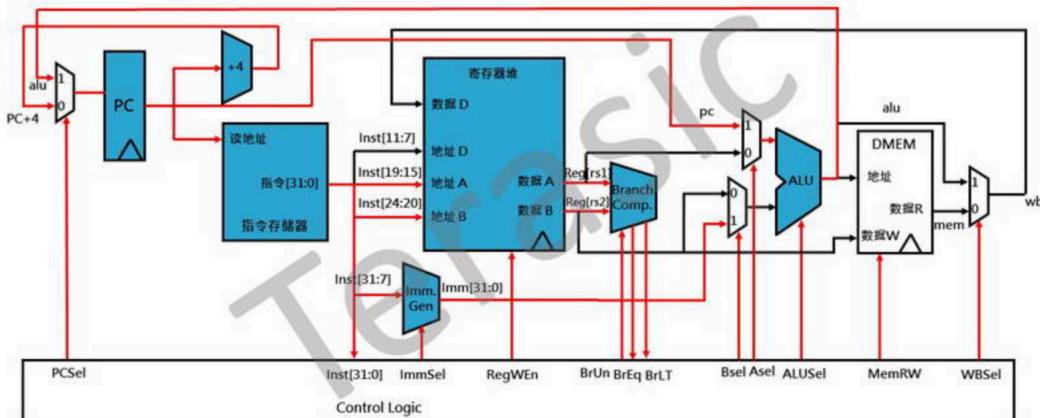
B 型指令的功能是比较寄存器 $rs1$ 、 $rs2$ 中的值，并根据比较结果进行分支跳转。 $funct3$ 字段表示分支跳转的类型。B 型指令中 beq/bne 需进行相等比较的计算， $blt/bltu$ 与 $bge/bgeu$ 需进行量值比较的计算。

(b) 数据通路 & 指令执行流程:

- 首先按照 PC 中的地址，从指令存储器中读出 32 位指令，然后 $PC+4$ ；
- 指令中的控制码和功能码送到控制器中产生相应的控制信号；
- 将指令中的 $rs1$ 和 $rs2$ 的地址送入寄存器堆，得到对应的寄存器的值 $\text{Reg}[rs1]$ 和 $\text{Reg}[rs2]$ ；



- B-type 指令与 S-type 指令类似，需要两个源寄存器 rs1、rs2，和一个 12-bit 的立即数
- 12-bit 的立即数实际上是 13-bit 的有符号数，最低位为 0，所以没有进行存储
- 比较操作数 rs1、rs2
 - ✓ 首先进行符号位的扩展
 - ✓ 然后左移 1 位
 - ✓ 将得到的立即数与 PC 值相加



- Reg[rs1] 和 Reg[rs2] 被送入比较运算单元，比较后的结果送至控制器用于判断是否进行目标地址的跳转；
- 将 PC 的值送至 ALU 的第一个输入端；
- 将指令中的立即数字段送入立即数扩展电路，该电路输出一个 32 位的数，送至 ALU 的第二个输入端；
- 若进行跳转，则由 ALU 计算出目标地址送至 PC 处。

(c) 指令格式：

imm[12 10:5]	rs2	rs1	funct3	imm[4:1 11]	opcode	指令
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

5. U 型指令：

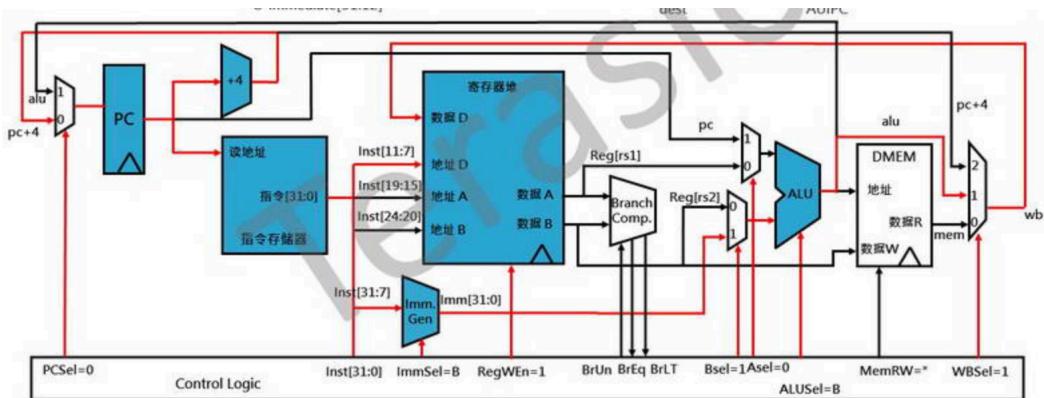
(a) 简介：

U 型指令包含 LUI 和 AUIPC，用于构建 32 位常数，由 20 位立即数，rd 和操作码构成。



(b) 数据通路 & 指令执行流程：

- 首先按照 PC 中的地址，从指令存储器中读出 32 位指令，然后 PC+4；



- ii. 指令中的控制码和功能码送到控制器中产生相应的控制信号；
- iii. Bsel = 1 使得 ALU B 端口输入扩展后的立即数，Asel = 1 使得 ALU A 端口输入 PC；
- iv. 将 20 位的立即数进行扩展（低 12 位填 0），输入进 ALU B 端口；
- v. 若是 LUI 指令，ALUsel = B(输出等于 B 端口输入)；若是 AUIPC 指令，ALUsel = Add(将扩展后的立即数与 PC 相加)；
- vi. WBsel = 1 使得 ALU 运算结果直接存入寄存器堆中 rd 所指寄存器；

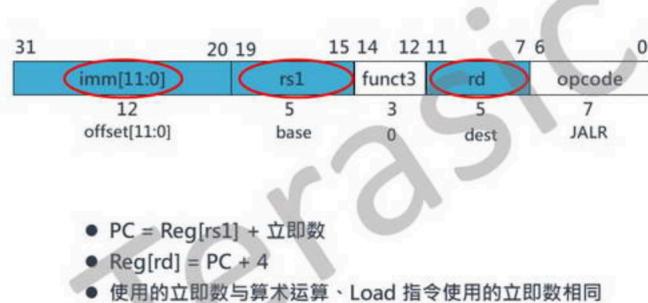
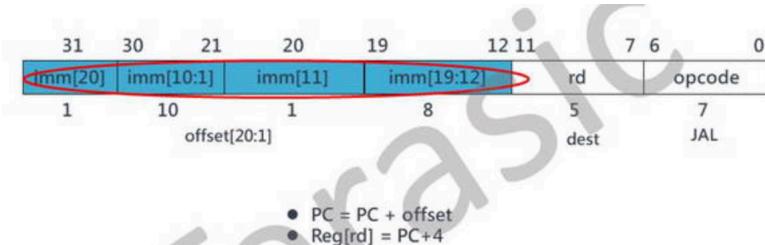
(c) 指令格式：

imm[32: 12]	rd	opcode	指令
imm[32: 12]	rd	0110111	LUI
imm[32: 12]	rd	0010111	AUIPC

6. J 型指令：

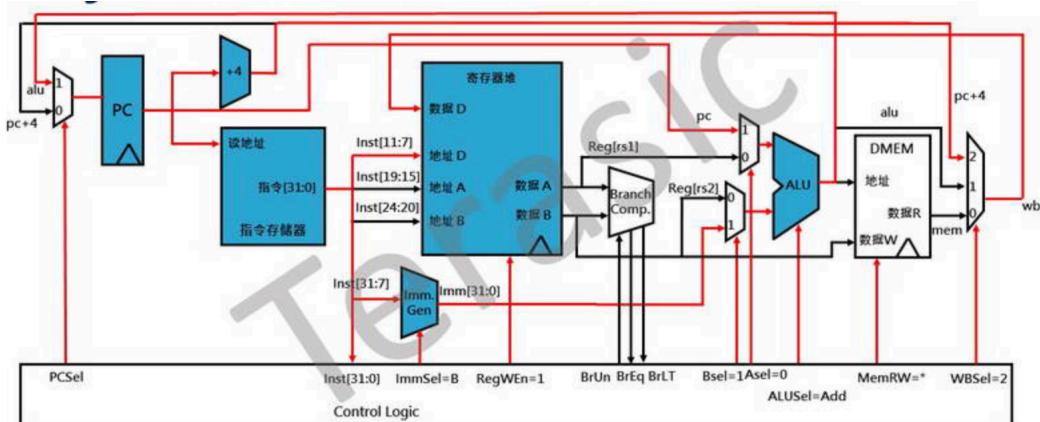
(a) 简介：

JAL/JALR 的功能是无条件跳转到目标地址。其中 JAL 指令中目标地址为 PC 的值加上 20 位的立即数所表示的偏移量，JALR 指令中目标地址为寄存器 rs1 的值加上 12 位的立即数所表示的偏移量。JAL/JALR 所需的计算类型只有加法运算。



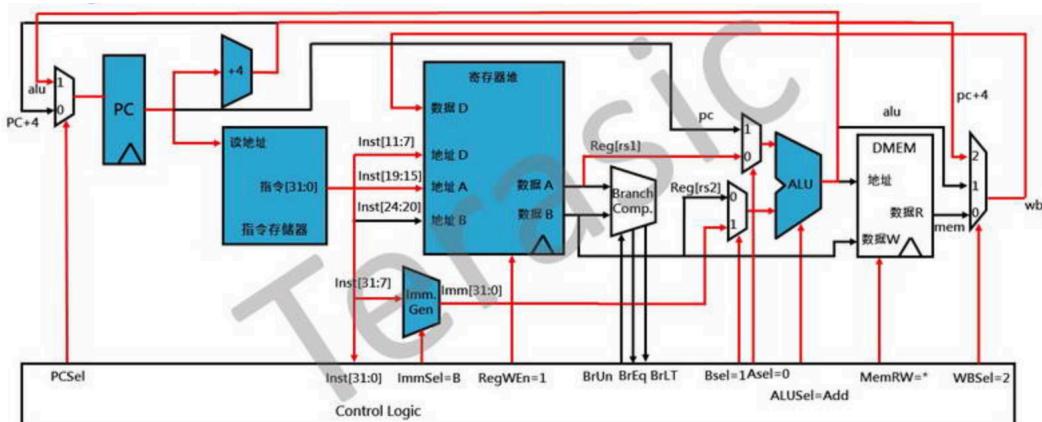
(b) 数据通路 & 指令执行流程:

JAL



- i. 首先按照 PC 中的地址, 从指令存储器中读出 32 位指令, 然后 PC+4;
- ii. 指令中的控制码和功能码送到控制器中产生相应的控制信号;
- iii. 将 PC 的值送至 ALU 的第一个输入端;
- iv. 将指令中的立即数字段送入立即数扩展电路, 该电路输出一个 32 位的数, 送至 ALU 的第二个输入端;
- v. 控制信号 ALUSel = Add, 指示 ALU 做加法运算, 得到的结果即为目标地址, 将其送至 PC;
- vi. 将原来 PC+4 的值写回到寄存器 rd 中;

JALR



- i. 首先按照 PC 中的地址, 从指令存储器中读出 32 位指令, 然后 PC+4;
- ii. 指令中的控制码和功能码送到控制器中产生相应的控制信号;
- iii. 将指令中的 rs1,rd 的地址送入寄存器堆, 得到 rs1 对应的寄存器的值 Reg[rs1], 并送入 ALU 的第一个端口;
- iv. 将指令中的立即数字段送入立即数扩展电路, 该电路输出一个 32 位的数, 送至 ALU 的第二个输入端;
- v. 控制信号 ALUSel = Add, 指示 ALU 做加法运算, 得到的结果即为目标地址, 将其送至 PC;
- vi. 将原来 PC+4 的值写回到寄存器 rd 中;

(c) 指令格式:



1.2 蜂鸟 E203 简介

E203

- 蜂鸟 E203 系列处理器由作者所在的公司开发，是一款开源的 RISC-V 处理器。蜂鸟是世界上最小的鸟类，其体积虽小，却有着极高的速度与敏锐度，可以说是“能效比”最高的鸟类。E203 系列以蜂鸟命名便寓意于此，旨在将其打造成为一款世界上最高能效比的 RISC 处理器。



E203 核心数据通路的模块划分

- IFU 取址单元
- EXU 执行单元
- LSU 访存单元
- BIU 总线

E203 Core 的代码架构

E203 数据通路的两级流程水线

- 第一级是 IFU，包括，取址、分支预测、生成 PC。
- 第二级是译码、派遣、执行、访存、写回。

E203 的特点

- 蜂鸟 E203 处理器研发团队拥有在国际一流公司多年开发处理器的经验，使用稳健的。
- 蜂鸟 E203 的代码为人工编写，添加丰富的注释且可读性强，非常易于理解。
- 蜂鸟 E203 专为 IoT 领域量身定做，其具有 2 级流水线深度，功耗和性能指标均优于目前主流商用的 ARM Cortex-M 系列处理器，且免费开源，能够在 IoT 领域完美替代 ARM Cortex-M 处理器。

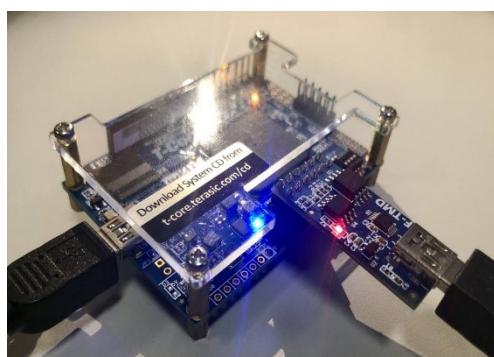
```

e200_opensource
|----rtl           // 存放 RTL 的目录
|----e203          // E203 核和 SoC 的 RTL 目录
|----general       // 存放一些公用的通用 RTL 代码
|----core          // 存放 e203 Core 的 RTL 代码,
                  // 列举主要文件如下, 全部的文件列表请参见 GitHub
|----config.v      // 参数配置文件, 参见第 4.6 节了解具体配置信息
|----e203_biu.v    // BIU 模块
|----e203_reset_ctrl.v // Core 的复位控制 (Reset Control) 模块
|----e203_clk_ctrl.v // Core 的时钟控制 (Clock Control) 模块
|----e203_cpu_top.v // Core 的顶层模块
|----e203_cpu.v    // Core 去除了 SRAM 之后的逻辑顶层模块
|----e203_core.v   // Core 的主体逻辑模块
|----e203_dtcn_ctrl.v // DTCM 的控制模块
|----e203_itcm_ctrl.v // ITCM 的控制模块
|----e203_exu.v    // Core 内部执行单元顶层模块
|----e203_ifu.v    // Core 内部取指令单元顶层模块
|----e203_lsu.v    // Core 内部存储器访问单元顶层模块
|----e203_srams.v  // Core 的所有 SRAM 的顶层模块
|----e203_itcm_ram.v // ITCM 的 SRAM 模块
|----e203_dtcn_ram.v // DTCM 的 SRAM 模块

```

1.3 T-core 开发板介绍

1. T-core 开发板是友晶科技公司的基于 RISC-V 的新款开发板。T-Core 提供了围绕 Intel MAX 10 FPGA 构建的强大的硬件设计平台。它配备完善，可在控制平面或数据路径应用中提供具有成本效益的单芯片解决方案，并提供行业领先的可编程逻辑，以实现最终的设计灵活性。
2. 借助 MAX 10 FPGA，可以获得比上一代更低的功耗/成本和更高的性能。可支持大量应用，包括协议桥接，电机控制驱动，模数转换和手持设备。T-Core 开发板包括硬件，例如板载 USB-Blaster II, QSPI Flash, ADC 接头连接器，WS2812B RGB LED 和 2x6 TMD 扩展接头连接器。通过利用所有这些功能，T-Core 是展示，评估和原型化 Intel MAX 10 FPGA 真正潜力的理想解决方案。T-Core 还通过板载 JTAG 调试支持 RISC-V CPU。它是学习 RISC-V CPU 设计或嵌入式系统设计的理想平台。



2. 实践篇

2.1 硬件开发

2.2 软件开发

我们需要根据我们设计的指令格式修改编译器，使其能够编译 dot 指令，并将其转换为对应的机器码。这样，编译器就可以将包含 dot 指令的测试程序编译成可执行文件，供硬件执行。

1. 生成操作码宏定义

- 文件包 riscv-opcodes 中枚举了全部 RISC-V 指令的操作码信息。因为我们设计的 dot 指令属于 rv32i 指令集，因此需要在 opcodes-rv32i 文件中加入我们编写的 dot 指令的指令格式。

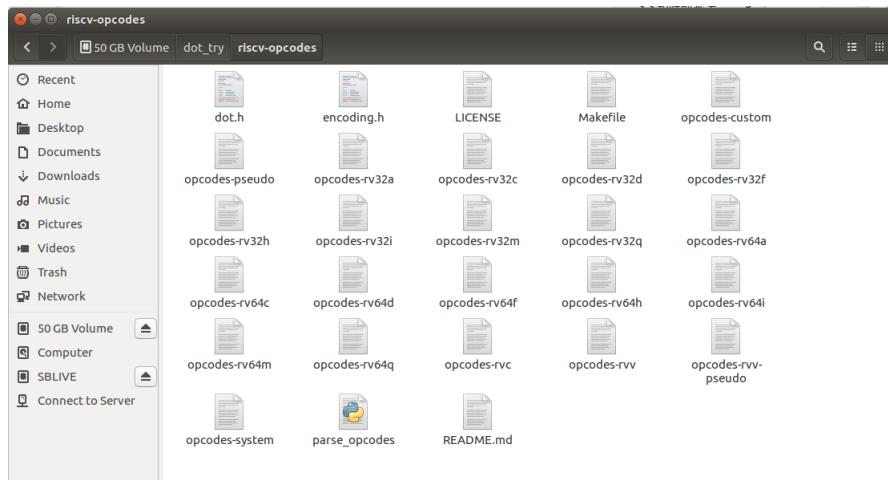


图 1: riscv-opcodes 文件包

```
*opcodes-rv32i (50 GB Volume /media/terasic/2C0827517A4F7
Open ▾ F
sltiu    rd rs1 imm12          14..12=3 6..2=0x04 1..0=3
xori     rd rs1 imm12          14..12=4 6..2=0x04 1..0=3
srli     rd rs1 31..26=0 shamt 14..12=5 6..2=0x04 1..0=3
srai     rd rs1 31..26=16 shamt 14..12=5 6..2=0x04 1..0=3
ori      rd rs1 imm12          14..12=6 6..2=0x04 1..0=3
andi    rd rs1 imm12          14..12=7 6..2=0x04 1..0=3

add     rd rs1 rs2 31..25=0 14..12=0 6..2=0x0C 1..0=3
sub     rd rs1 rs2 31..25=32 14..12=0 6..2=0x0C 1..0=3
sll     rd rs1 rs2 31..25=0 14..12=1 6..2=0x0C 1..0=3
slt     rd rs1 rs2 31..25=0 14..12=2 6..2=0x0C 1..0=3
sltu   rd rs1 rs2 31..25=0 14..12=3 6..2=0x0C 1..0=3
xor     rd rs1 rs2 31..25=0 14..12=4 6..2=0x0C 1..0=3
srl     rd rs1 rs2 31..25=0 14..12=5 6..2=0x0C 1..0=3
sra     rd rs1 rs2 31..25=32 14..12=5 6..2=0x0C 1..0=3
or      rd rs1 rs2 31..25=0 14..12=6 6..2=0x0C 1..0=3
and    rd rs1 rs2 31..25=0 14..12=7 6..2=0x0C 1..0=3
dot4   rd rs1 rs2 31..25=1 | 14..12=0 6..2=0x1A 1..0=3
dot3   rd rs1 rs2 31..25=1 14..12=0 6..2=0x15 1..0=3
dot2   rd rs1 rs2 31..25=1 14..12=0 6..2=0x1D 1..0=3
```

图 2: 修改 opcodes-rv32i 文件

- 运用转换脚本 parse_opcodes 生成编译器所需要的信息，即 C 语言格式的宏定义，存放在 dot.h 中

```
terasic@terasic: /media/terasic/e0a59f2c-acf9-4e5a-9fb5-e7605ff2fbee/dot_try/riscv-opc  
terasic@terasic:/media/terasic/e0a59f2c-acf9-4e5a-9fb5-e7605ff2fbee/dot_try/risc  
v-opcodes$ cat opcodes-rv32i | ./parse_opcodes -c > ./dot.h  
terasic@terasic:/media/terasic/e0a59f2c-acf9-4e5a-9fb5-e7605ff2fbee/dot_try/risc  
v-opcodes$ █
```

图 3: 用脚本转换 opcodes

2. 修改 RISC-V GNU 工具链

RISC-V GNU 工具链是一组用于支持 RISC-V C 和 C++ 的交叉编译工具链，这些工具构成了一个完整的系统。GNU 工具链包括 riscv-gcc、riscv-glibc 等子仓库。

- 从 gitee 上下载完整的工具链后，打开“riscv-gnu-toolchain/riscv-binutils/include/opcode”路径下的 riscv-opc.h 文件，将第 1 步中生成的 dot 指令相关的定义和代码复制到该文件中。

```
riscv-opc.h (50 GB Volume /media/ter  
Open ▾ [+]  
  
#define MATCH_OR 0x6033  
#define MASK_OR 0xfe00707f  
#define MATCH_AND 0x7033  
#define MASK_AND 0xfe00707f  
#define MATCH_DOT4 0x200006b  
#define MASK_DOT4 0xfe00707f  
#define MATCH_DOT3 0x2000057  
#define MASK_DOT3 0xfe00707f  
#define MATCH_DOT2 0x2000077  
#define MASK_DOT2 0xfe00707f  
#define MATCH_ADDIW 0x1b  
#define MASK_ADDIW 0x707f
```

图 4: 更改工具链

```
DECLARE_INSN(xor, MATCH_XOR, MASK_XOR)
DECLARE_INSN(srl, MATCH_SRL, MASK_SRL)
DECLARE_INSN(sra, MATCH_SRA, MASK_SRA)
DECLARE_INSN(or, MATCH_OR, MASK_OR)
DECLARE_INSN(and, MATCH_AND, MASK_AND)
DECLARE_INSN(dot4, MATCH_DOT4, MASK_DOT4)
DECLARE_INSN(dot3, MATCH_DOT3, MASK_DOT3)
DECLARE_INSN(dot2, MATCH_DOT2, MASK_DOT2)
DECLARE_INSN(addiw, MATCH_ADDIW, MASK_ADDIW)
DECLARE_INSN(slliw, MATCH_SLLIW, MASK_SLLIW)
```

图 5: 更改工具链

- 打开“riscv-gnu-toolchain/riscv-binutils/opcodes”路径下的 riscv-opc.c 文件,找到定义的 riscv_opcodes 结构体, 在其中添加 dot 指令

```
riscv-opc.c (50 GB Volume /media/terasic/2C0827517A4F7E4A...y/riscv-gnu-toolchain/riscv-binutils
```

Open  

riscv-opc.h

```
{"dot4", 0, INSN_CLASS_I, "d,s,t", MATCH_DOT4, MASK_DOT4, match_opcode, 0},  
{"dot3", 0, INSN_CLASS_I, "d,s,t", MATCH_DOT3, MASK_DOT3, match_opcode, 0},  
{"dot2", 0, INSN_CLASS_I, "d,s,t", MATCH_DOT2, MASK_DOT2, match_opcode, 0},
```

图 6: 更改工具链

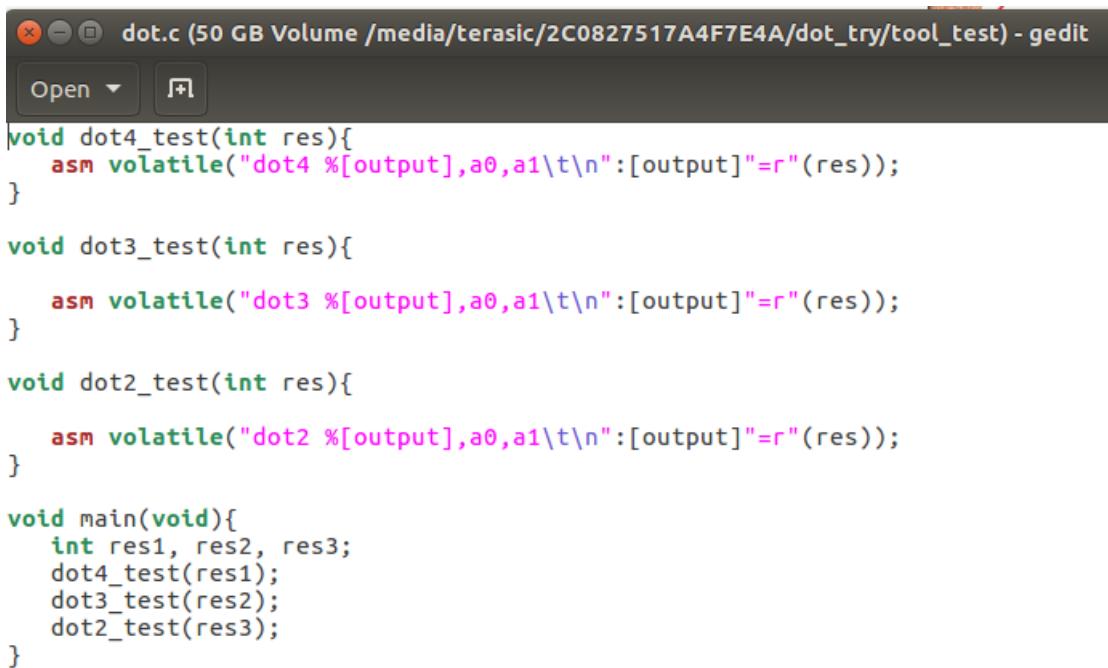
- 修改完成后，我们需要安装一些编译依赖，然后就可以重新编译生成工具链了，此过程较长，需要 20 分钟左右。

```
terasic@terasic: /media/terasic/e0a59f2c-acf9-4e5a-9fb5-e7605ff2fbee/dot_try/riscv-gnu-toolchain$ ./configure --prefix=/media/terasic/e0a59f2c-acf9-4e5a-9fb5-e7605ff2fbee/dot_try/riscv-gnu-toolchain --with-dot --with-arch=rv32imac --with-abi=ilp32
checking for gcc... gcc
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
checking whether we are cross compiling... no
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ISO C89... none needed
checking for grep that handles long lines and -e... /bin/grep
checking for fgrep... /bin/grep -F
checking for grep that handles long lines and -e... (cached) /bin/grep
checking for bash... /bin/bash
checking for __gmpz_init in -lgmp... yes
checking for mpfr_init in -lmpfr... yes
checking for mpc_init2 in -lmpc... yes
checking for curl... /usr/bin/curl
checking for wget... /usr/bin/wget
checking for ftp... /usr/bin/ftp
configure: creating ./config.status
config.status: creating Makefile
config.status: creating scripts(wrapper/awk/awk)
config.status: creating scripts(wrapper/sed/sed)
terasic@terasic: /media/terasic/e0a59f2c-acf9-4e5a-9fb5-e7605ff2fbee/dot_try/riscv-gnu-toolchain$ sudo make -j8
```

图 7: 编译工具链

3. 测试指令

- 首先在 dot_try 文件夹下新建一个 tool_test 文件夹，并在其中新建 dot.c 文件，用于测试 dot 指令是否可以被正常编译。



```

void dot4_test(int res){
    asm volatile("dot4 %[output],a0,a1\t\n":[output] "=r"(res));
}

void dot3_test(int res){
    asm volatile("dot3 %[output],a0,a1\t\n":[output] "=r"(res));
}

void dot2_test(int res){
    asm volatile("dot2 %[output],a0,a1\t\n":[output] "=r"(res));
}

void main(void){
    int res1, res2, res3;
    dot4_test(res1);
    dot3_test(res2);
    dot2_test(res3);
}

```

图 8: 编写测试程序

- 调用 gcc 编译 dot.c 文件，如果成功修改了工具链，就会生成 dot 文件。

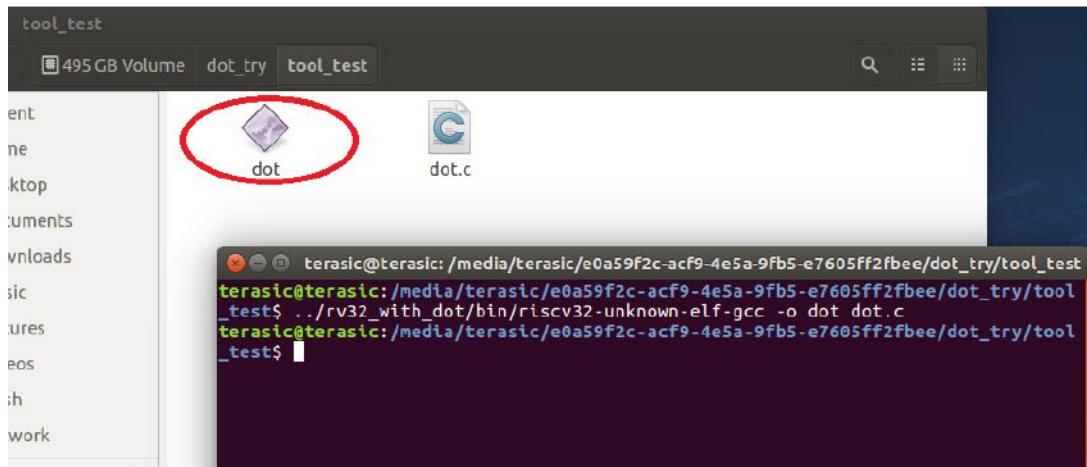


图 9: 调用 gcc 编译

- 进行汇编代码查看。可以在 dot2_test、dot3_test、dot4_test 函数的片段中，找到生成的 dot 汇编指令。

```
@terasic: /media/terasic/2C0827517A4F7E4A/dot_try/tool_test
00010106 <dot4_test>:
10106:    1101                  addi    sp,sp,-32
10108:    ce22                  sw      s0,28(sp)
1010a:    1000                  addi    s0,sp,32
1010c:    fea42623             sw      a0,-20(s0)
10110:    02b507eb             dot4    a5,a0,a1
10114:    fef42623             sw      a5,-20(s0)
10118:    0001                  nop
1011a:    4472                  lw      s0,28(sp)
1011c:    6105                  addi    sp,sp,32
1011e:    8082                  ret

00010120 <dot3_test>:
10120:    1101                  addi    sp,sp,-32
10122:    ce22                  sw      s0,28(sp)
10124:    1000                  addi    s0,sp,32
10126:    fea42623             sw      a0,-20(s0)
1012a:    02b507d7             dot3    a5,a0,a1
1012e:    fef42623             sw      a5,-20(s0)
10132:    0001                  nop
10134:    4472                  lw      s0,28(sp)
10136:    6105                  addi    sp,sp,32
10138:    8082                  ret

0001013a <dot2_test>:
1013a:    1101                  addi    sp,sp,-32
1013c:    ce22                  sw      s0,28(sp)
1013e:    1000                  addi    s0,sp,32
10140:    fea42623             sw      a0,-20(s0)
10144:    02b507f7             dot2    a5,a0,a1
10148:    fef42623             sw      a5,-20(s0)
1014c:    0001                  nop
1014e:    4472                  lw      s0,28(sp)
10150:    6105                  addi    sp,sp,32
10152:    8082                  ret
```

图 10: 查看 dot 指令

4. 创建程序文件

- 在”demo_dot”文件夹下创建一个”demo_dot.c”的文本文档。包含以下头文件。

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdbool.h>
4 #include <stdatomic.h>
5 #include "encoding.h"
6 #include <platform.h>
```

- 宏定义，三个常数 CONST_I、CONST_K、CONST_J 分别为 32、64、32，定义标识符 A_ROW（矩阵 A 的行）为 CONST_I，定义 A_COL（矩阵 A 的列）为 CONST_K；定义标识符 B_ROW（矩阵 B 的行）为 CONST_K，定义 B_COL（矩阵 B 的列）为 CONST_J；定义标识符 RES_ROW（乘法运算得到的新矩阵的行）为 CONST_I，定义 A_COL（乘法运算得到的新矩阵的列）为 CONST_J。

```
// matrix dimensions
```

```

2 // CONST_K must set as multiple of 4 for dot instruction
3 #define CONST_I 32
4 #define CONST_K 64
5 #define CONST_J 32
6 //a[const_i][const_k]
7 #define A_ROW CONST_I
8 #define A_COL CONST_K
9 //b[const_k][const_j]
10 #define B_ROW CONST_K
11 #define B_COL CONST_J
12 //res[const_i][const_j]
13 #define RES_ROW CONST_I
14 #define RES_COL CONST_J

```

- 定义 demo_uart_init 函数，用于初始化，首先配置 GPIO_IOF_EN 对应的比特位为 1，使能 IOF 模式，再将 GPIO_IOF_SEL 对应的比特位清零，选择 IOF0，再配置 UART_DIV 寄存器，设置波特率约为 115200。最后配置 UART_TXCTRL 和 UART_RXCTRL，将 UART0 的 TX 和 RX 使能。

```

void uart_init(){
    // Configure UART to print
    GPIO_REG(GPIO_IOF_EN) |= IOF0_UART0_MASK;
    GPIO_REG(GPIO_IOF_SEL) &= ~IOF0_UART0_MASK;
    // 115200 Baud Rate
    // get_cpu_freq() / baud_rate - 1, and get_cpu_freq() = 16MHz
    UART0_REG(UART_REG_DIV) = 138;
    UART0_REG(UART_REG_TXCTRL) |= UART_TXEN; // enable tx
    UART0_REG(UART_REG_RXCTRL) |= UART_RXEN; // enable rx
}

```

- 主函数中，首先进行 UART 的初始化，并定义 i、j、k 变量分别用于遍历矩阵的行和列，reg 变量作为 dot 指令算法输出矩阵的寄存器，incr 变量作为运算步长。

```

int main(int argc, char *argv[]) {
    uart_init();
    int i = 0, j = 0, k = 0, reg = 0, incr = 4;
}

```

- 打印”Malloc and initial Matrixs!!”字符串，定义四个指针变量，并使用 malloc 函数分配 A 矩阵、B 矩阵、未使用 dot 指令进行矩阵相乘得到的新矩阵、使用 dot 指令进行矩阵相乘得到的新矩阵所需的内存空间，并返回一个指针，指向已分配大小的内存。

```

// malloc and init matrixs
printf("Malloc and initial Matrixs!!\n\n");
// matrix stored as an array of size row * column
int *a = NULL, *b = NULL, *no_dot_res = NULL, *dot_res = NULL;
a = (int*) malloc(A_ROW * A_COL * sizeof(int));
b = (int*) malloc(B_ROW * B_COL * sizeof(int));
no_dot_res = (int*) malloc(RES_ROW * RES_COL * sizeof(int));
dot_res = (int*) malloc(RES_ROW * RES_COL * sizeof(int));

```

- 初始化矩阵 A 和矩阵 B，初始化矩阵 A 和矩阵 B 相乘的两种运算方式的结果矩阵。

```

1 // initialization matrix A
2     for(i = 0; i < A_ROW; i++) {
3         for (j = 0; j < A_COL; j++) {
4             a[i * A_COL + j] = i * A_COL + j;
5         }
6     }
7 // initialization matrix B
8     for(i = 0; i < B_ROW; i++) {
9         for (j = 0; j < B_COL; j++) {
10            b[i * B_COL + j] = i * B_COL + j;
11        }
12    }
13 // initialization matrix no_dot_res,dot_res
14     for(i = 0; i < RES_ROW; i++) {
15         for (j = 0; j < RES_COL; j++) {
16             no_dot_res[i * RES_COL + j] = 0;
17             dot_res [i * RES_COL + j] = 0;
18         }
19     }
20

```

- RISC-V 定义了 3 个 64 位计数器，分别为：instret、cycle、time，这三个寄存器可以用来评估硬件性能。其中，instret 计数器统计自 CPU 复位以来共运行了多少条指令；cycle 计数器统计自 CPU 复位以来共运行了多少个周期；time 计数器统计自 CPU 复位以来共运行了多少时间，驱动 time 计数器是已知的固定频率的时钟，例如 32768Hz 的时钟。采用常规算法进行矩阵的乘法运算。并调用 get_instret_value()、get_cycle_value()、get_timer_value() 三个函数，通过计算得到这种运算方式的指令数、周期数以及运行的时间。

```

1 // no dot instruction(traditional tile matrix multiplication)
2 printf("Matrix multiplication without using custom DOT instruction:
3 \n");
4     unsigned int no_dot_instret_start = get_instret_value();
5     unsigned int no_dot_cycle_start = get_cycle_value();
6     unsigned int no_dot_timer_start = get_timer_value();
7     for (i = 0; i < RES_ROW; i++) {
8         for (j = 0; j < RES_COL; j++) {
9             for (k = 0; k < CONST_K; k++) {
10                 no_dot_res[i * RES_COL + j] += a[i * A_COL + k] * b[k *
11                     B_COL + j];
12             }
13         }
14     }
15     unsigned int no_dot_timer_cost = get_timer_value() -
16     no_dot_timer_start;
17     unsigned int no_dot_cycle_cost = get_cycle_value() -
18     no_dot_cycle_start;
19     unsigned int no_dot_instret_cost = get_instret_value() -
20     no_dot_instret_start;
21     printf("not_dot_time cost: %.02fms\n",
22     (float)no_dot_timer_cost/RTC_FREQ*1000);
23     printf("not_dot_cycle: %u\n", no_dot_cycle_cost);

```

```

25     printf("not_dot_instret: %u\n", no_dot_instret_cost);
26     printf("not_dot CPI: %.2f\n\n",
27           (float)no_dot_cycle_cost/no_dot_instret_cost);

```

- 采用 dot2、dot3、dot4 指令进行矩阵的乘法运算。并调用 get_instret_value()、get_cycle_value()、get_timer_value() 三个函数，通过计算得到这种运算方式的指令数、周期数以及运行的时间。

```

1      unsigned int dot_instret_start = get_instret_value();
2      unsigned int dot_cycle_start = get_cycle_value();
3      unsigned int dot_timer_start = get_timer_value();
4
5      for (i = 0; i < RES_ROW; i++) {
6          for (j = 0; j < RES_COL; j++) {
7              int k = 0;
8
9              asm volatile (
10                  "lw x10, %[a0]\t\n"
11                  "lw x11, %[a1]\t\n"
12                  "lw x12, %[a2]\t\n"
13                  "lw x13, %[a3]\t\n"
14                  "lw x14, %[a4]\t\n"
15                  "lw x15, %[a5]\t\n"
16                  "lw x16, %[a6]\t\n"
17                  "lw x17, %[a7]\t\n"
18                  "dot4 %[output], x12, x13\t\n"
19                  : [output] "=r"(reg)
20                  : [a0] "m"(a[i * A_COL + (k + 0)])
21                  ,[a1] "m"(a[i * A_COL + (k + 1)])
22                  ,[a2] "m"(a[i * A_COL + (k + 2)])
23                  ,[a3] "m"(a[i * A_COL + (k + 3)])
24                  ,[a4] "m"(b[(k + 0) * B_COL + j])
25                  ,[a5] "m"(b[(k + 1) * B_COL + j])
26                  ,[a6] "m"(b[(k + 2) * B_COL + j])
27                  ,[a7] "m"(b[(k + 3) * B_COL + j])
28                  : "x10", "x11", "x12", "x13"
29                  , "x14", "x15", "x16", "x17"
30              );
31
32              dot_res[i * RES_COL + j] += reg;
33              k += 4;
34
35              asm volatile (
36                  "lw x10, %[a0]\t\n"
37                  "lw x11, %[a1]\t\n"
38                  "lw x12, %[a2]\t\n"
39                  "lw x13, %[a3]\t\n"
40                  "lw x14, %[a4]\t\n"
41                  "lw x15, %[a5]\t\n"
42                  "dot3 %[output], x12, x13\t\n"
43                  : [output] "=r"(reg)
44                  : [a0] "m"(a[i * A_COL + (k + 0)])
45                  ,[a1] "m"(a[i * A_COL + (k + 1)])
46                  ,[a2] "m"(a[i * A_COL + (k + 2)])
47                  ,[a3] "m"(b[(k + 0) * B_COL + j])
48                  ,[a4] "m"(b[(k + 1) * B_COL + j])
49                  ,[a5] "m"(b[(k + 2) * B_COL + j])
50              );
51
52          }
53      }

```

```

50      : "x10","x11","x12","x13"
51      , "x14", "x15"
52    );
53
54    dot_res[ i * RES_COL + j ] += reg;
55  }
56}
57
58 unsigned int dot_timer_cost = get_timer_value() - dot_timer_start;
59 unsigned int dot_cycle_cost = get_cycle_value() - dot_cycle_start;
60 unsigned int dot_instret_cost = get_instret_value() - dot_instret_start;
61
62 printf("dot time cost: %.2fms\n", (float)dot_timer_cost/RTC_FREQ*1000);
63 printf("dot_cycle: %u\n", dot_cycle_cost);
64 printf("dot_instret: %u\n", dot_instret_cost);
65 printf("dot CPI: %.2f\n\n", (float)dot_cycle_cost/dot_instret_cost);
66

```

- 对比两种运算方式的结果是否一致，来验证采用 dot 指令进行矩阵的乘法运算的结果是否是正确的。

```

1 // verify the no_dot_res and dot_res array are equal
2 printf("Matrix multiplication result verification: \n");
3 int verifyRes = 1;
4 for(i = 0; i < RES_ROW * RES_COL; i++) {
5     if(dot_res[i] != no_dot_res[i]) {
6         verifyRes = 0;
7         break;
8     }
9 }
10 if(verifyRes)
11     printf("Pass!\n\n");
12 else
13     printf("Fail!\n\n");

```

- 计算两种运算方式的指令数、周期数以及运行的时间的比值，并打印出来。最后释放由 malloc() 函数申请的内存空间。

```

1 printf("Ratio of timer: %.2f\n",
2     (float)no_dot_timer_cost/dot_timer_cost );
3 printf("Ratio of cycle: %.2f\n",
4     (float)no_dot_cycle_cost/dot_cycle_cost );
5 printf("Ratio of instret (retired instruction): %.2f\n",
6     (float)no_dot_instret_cost/dot_instret_cost );
7 // free matrix
8 free(a);
9 free(b);
10 free(no_dot_res);
11 free(dot_res);
12 return 0;
13

```

3. 结果展示

- 常规矩阵运算与自定义矩阵乘法指令的运算耗时对比 (49 x 49 大小的矩阵相乘)

```
Malloc and initial Matrixs!!  
  
Matrix multiplication without using custom DOT instruction:  
not_dot time cost: 185.49ms  
not_dot_cycle: 2968114  
not_dot_instret: 843132  
not_dot CPI: 3.52  
  
Matrix multiplication with custom DOT instruction:  
dot time cost: 31.98ms  
dot_cycle: 511944  
dot_instret: 449372  
dot CPI: 1.14  
  
Matrix multiplication result verification:  
Pass!  
  
Ratio of timer: 5.80  
Ratio of cycle: 5.80  
Ratio of instret (retired instruction): 1.88  
  
Program has exited with code:0x00000000
```

- 结果：耗时减少约 83%

4. 未来展望