

课程实验报告

RISC-V on T-Core

MaTrixV Team

目录

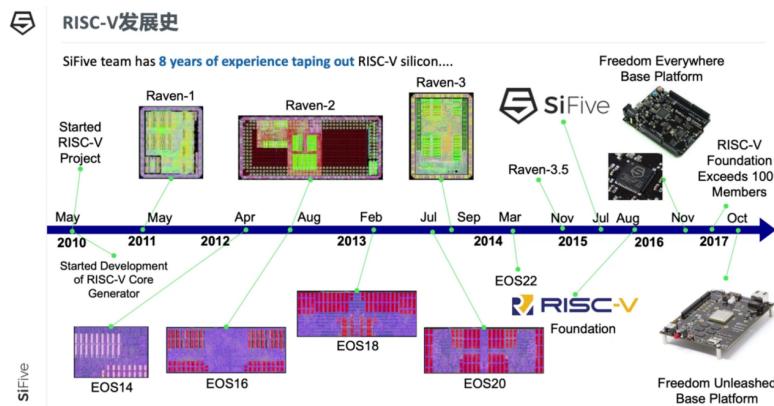
1.	基础篇	2
1.1	RISC-V 简介	2
1.2	蜂鸟 E203 简介	10
1.3	T-core 开发板介绍	11
1.4	矩阵乘法运算指令及数据通路	12
2.	实践篇	12
2.1	硬件开发	12
2.2	软件开发	18
3.	结果展示	27
3.1	编译并上传 C 语言程序	27
3.2	运行结果	27
4.	未来展望	29
4.1	可以改进的地方	29
4.2	对 FPGA 的理解	29

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。



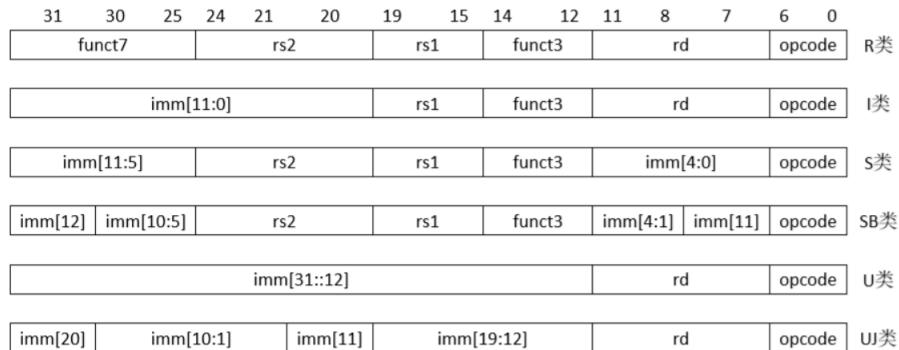
RISC-V 指令简述

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；

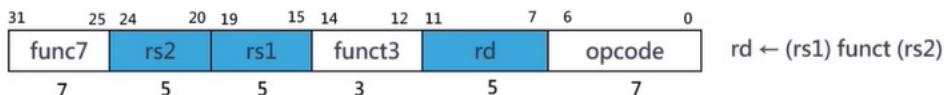


RISC-V 指令分述

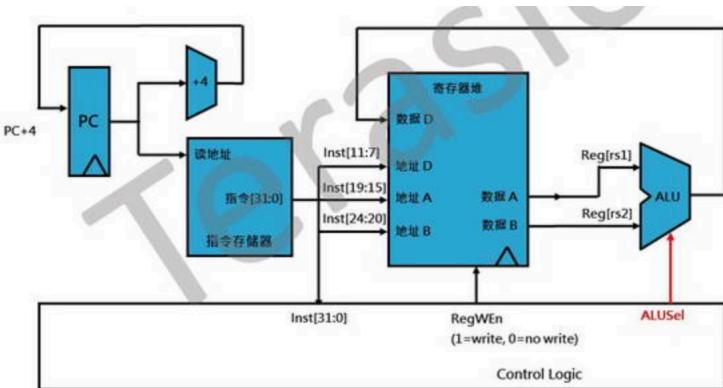
1. R 型指令：

(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：

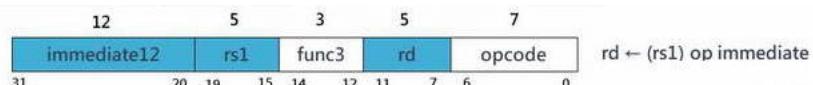
func7	rs2	rs1	func3	rd	opcode	指令
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd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

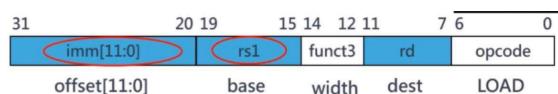
2. I 型指令：

(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。

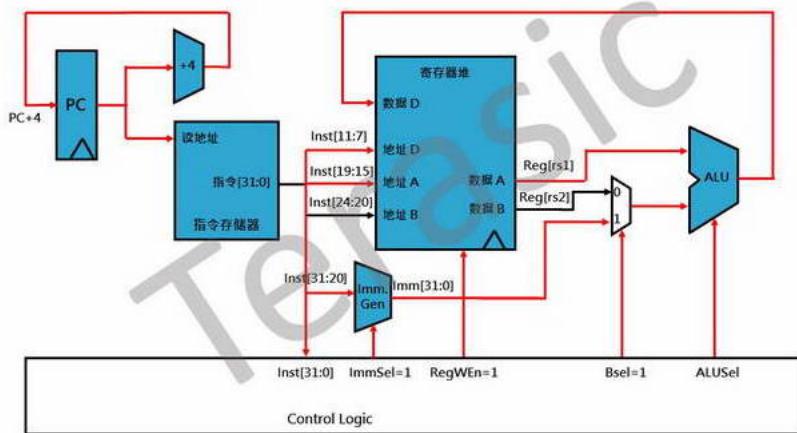


Load 指令属于 I - type 型，其功能是完成存储器的读操作。func3 字段表示读出存储器的数据的字长（32 位、16 位或 8 位）。Load 指令将 rs1 中的数据与 12 位立即数字段符号扩展相加所得到的结果作为访问存储器的地址，即存储器地址 = Reg[rs1] + imm。接着，将从存储器读出的数据存入目的寄存器 rd 中。



(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

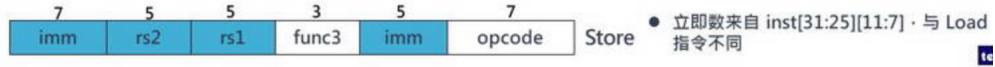
- 算数运算：加法 addi
- 逻辑运算：与 andi、或 ori、异或 xor
- 比较运算：有符号小于比较 slti、无符号小于比较 sltu
- 移位运算：逻辑左移 slli、逻辑右移 srli、算数右移 srai

3. S 型指令：

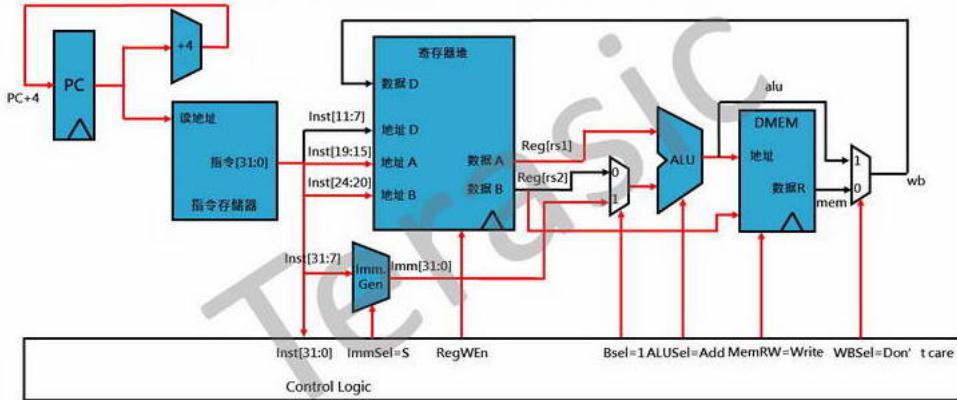
(a) 简介：

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

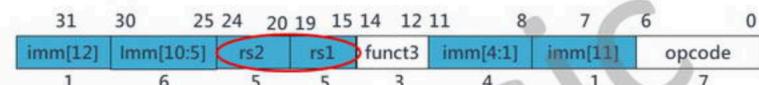
4. B 型指令:

(a) 简介:

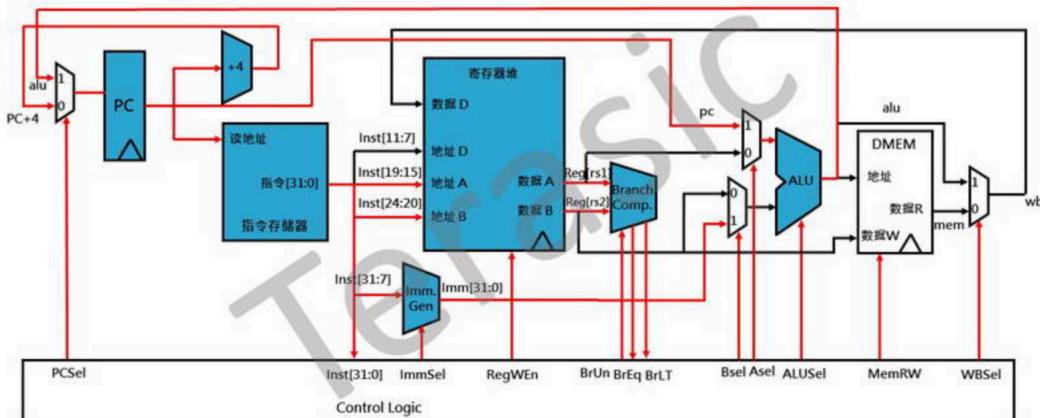
B 型指令的功能是比较寄存器 $rs1$ 、 $rs2$ 中的值，并根据比较结果进行分支跳转。 $funct3$ 字段表示分支跳转的类型。B 型指令中 beq/bne 需进行相等比较的计算， $blt/bltu$ 与 $bge/bgeu$ 需进行量值比较的计算。

(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：

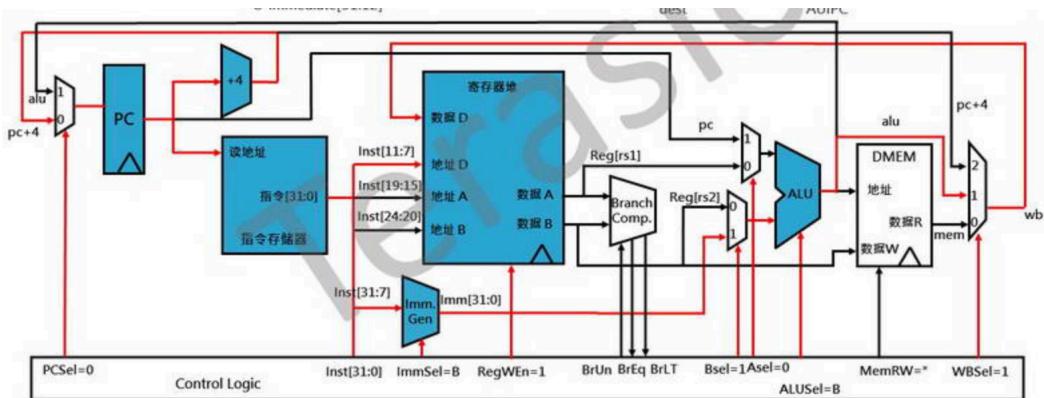
(a) 简介：

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；

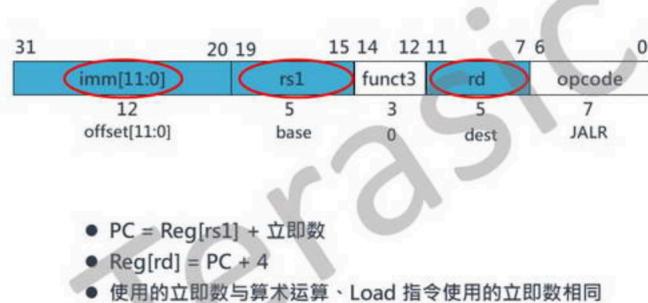
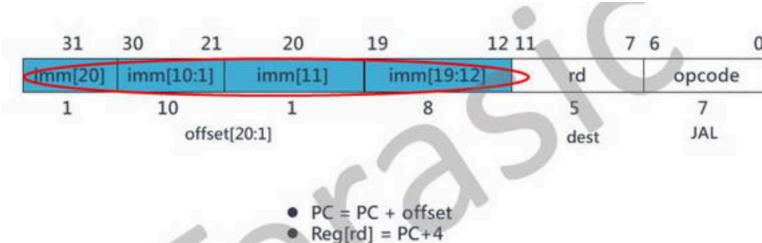
(c) 指令格式：

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

6. J 型指令：

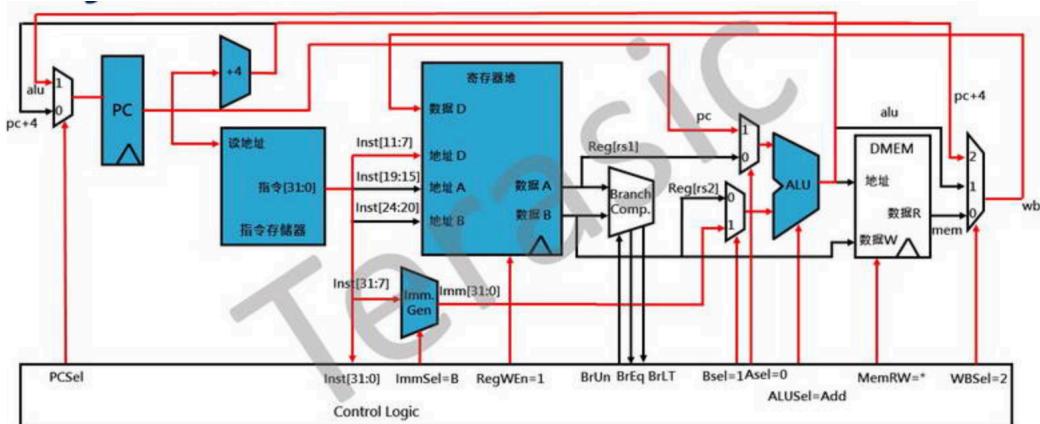
(a) 简介：

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。



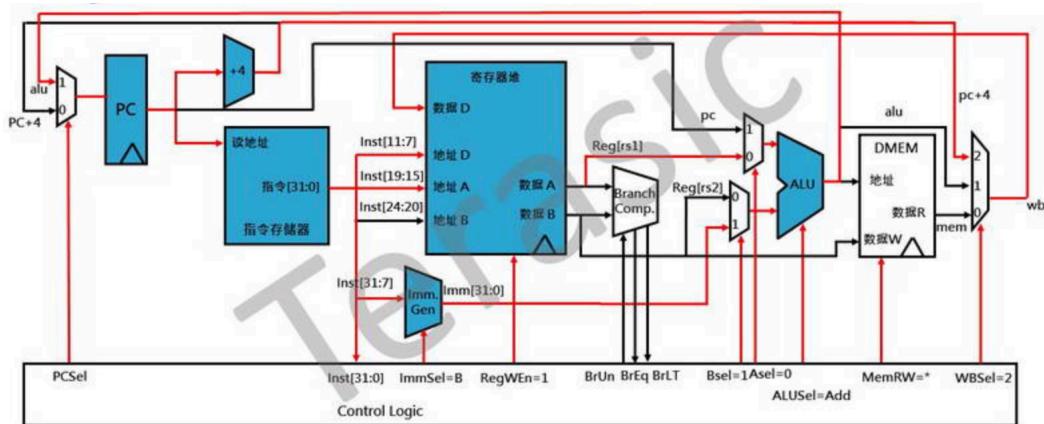
(b) 数据通路 & 指令执行流程:

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。

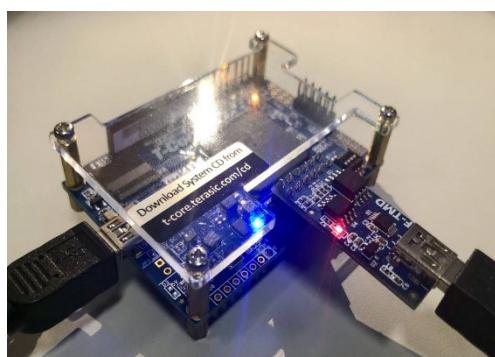
```

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。



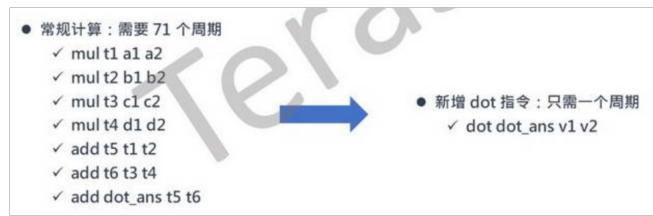
1.4 矩阵乘法运算指令及数据通路

1. 常规矩阵运算与自定义矩阵乘法指令的对比

- 以 1×4 与 4×1 大小的矩阵相乘为例



- 常规计算需要的总周期数: 71; DOT 指令需要的总周期数: 1
(每条乘法指令: 17 个周期; 每条加法指令: 1 个周期)



2. DOT 指令格式

操作码字段 (opcode) 选用 1101011; 将功能字段 (funct3) 定义为 000, 将 funct7 字段定义为 0000001; rs1、rs2 字段分别表示两个源操作数寄存器, rd 字段表示目标寄存器。(注: 在实际运算的时候, rs1、rs2 是没有被使用到的)。

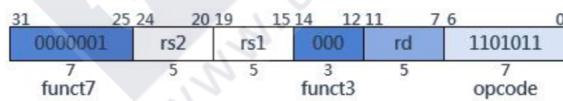


图3.3 dot 指令格式

3. DOT 指令数据通路分析

通过在 ALU 数据通路中增加几条选择线, 可以实现在 ALU 中添加自定义逻辑单元, 并通过译码信号使 ALU 输出该单元的计算结果。

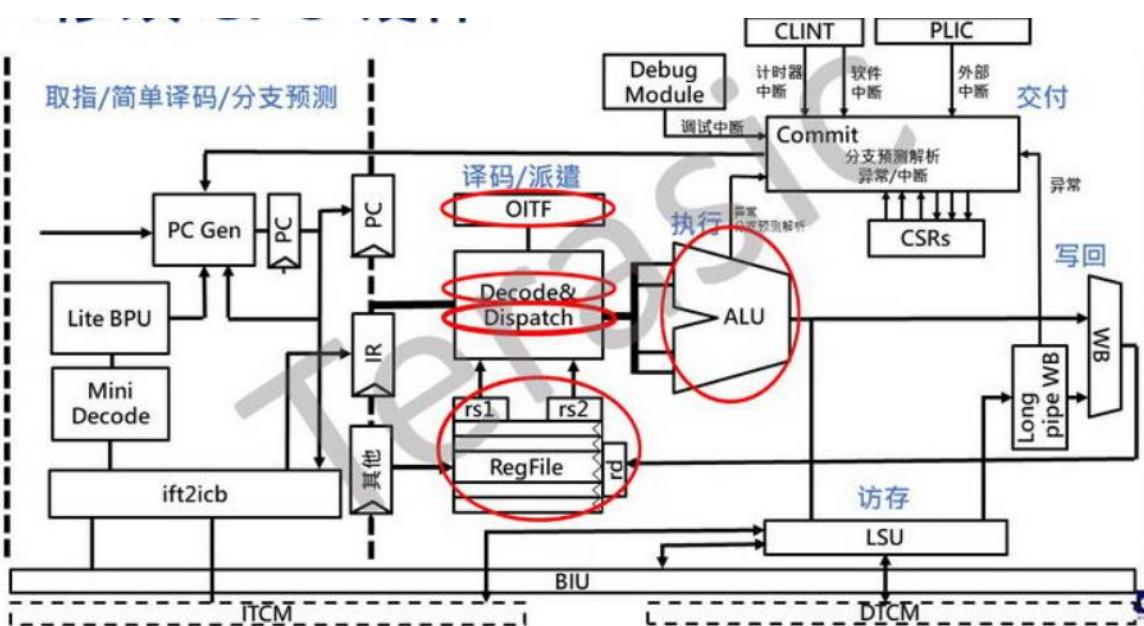
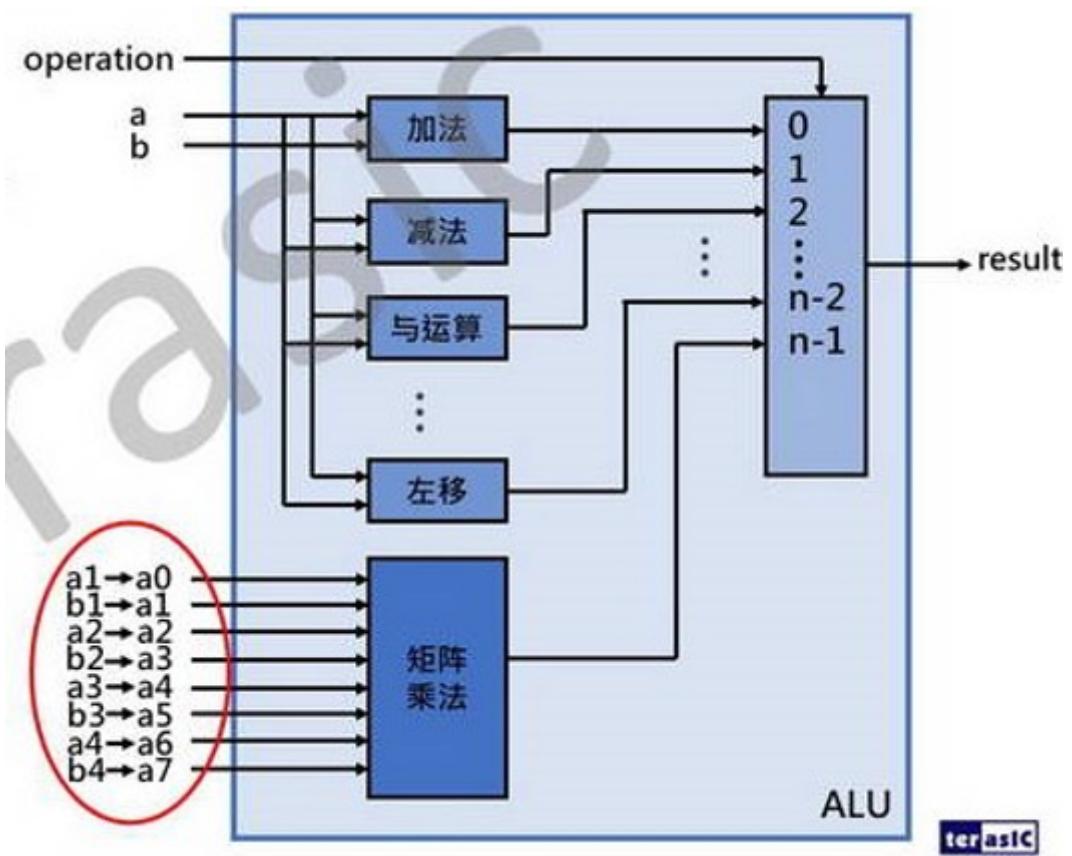
- RegFile: 8 个寄存器完成 $1, 4 \times 4, 1$ 的矩阵乘法空间, 并与 Dispatch 模块建立读取通道
- Decode: 对指令进行译码
- Dispatch: 增加 dot 指令相关的信号, 将取得的数值传入 ALU
- ALU: 实现 dot 指令的实现
- OITF: 解决寄存器与长指令的存取目的寄存器之间的冲突; 同时 Dispatch 模块负责处理冲突后的操作

2. 实践篇

2.1 硬件开发

我们在 Quartus 上通过 Verilog 语言对 CPU 中的 EXU 执行单元模块进行修改, 修改的步骤大致分为以下几个方面:

- 增加 DOT4、DOT3、DOT2 指令的宏定义;



- 给出 DOT4、DOT3、DOT2 的译码条件，加入 ALU 运算集，连接 ALU INFO 总线；
- 声明 a0-a7 的端口，把这 8 个端口对应到 x10-x17 寄存器上；
- 如果 OITF 中有长指令正在维护并且要写回目的寄存器 x10-x17 中的话，那就有数据冒险，不可以派遣；
- 把传来八个寄存器转到 ALU；
- 编写 DOT4、DOT3、DOT2 指令的 ALU 逻辑实现模块，并将得到的结果写回。

硬件修改的数据通路图如下图所示，我们将在红圈的地方进行增添和修改。

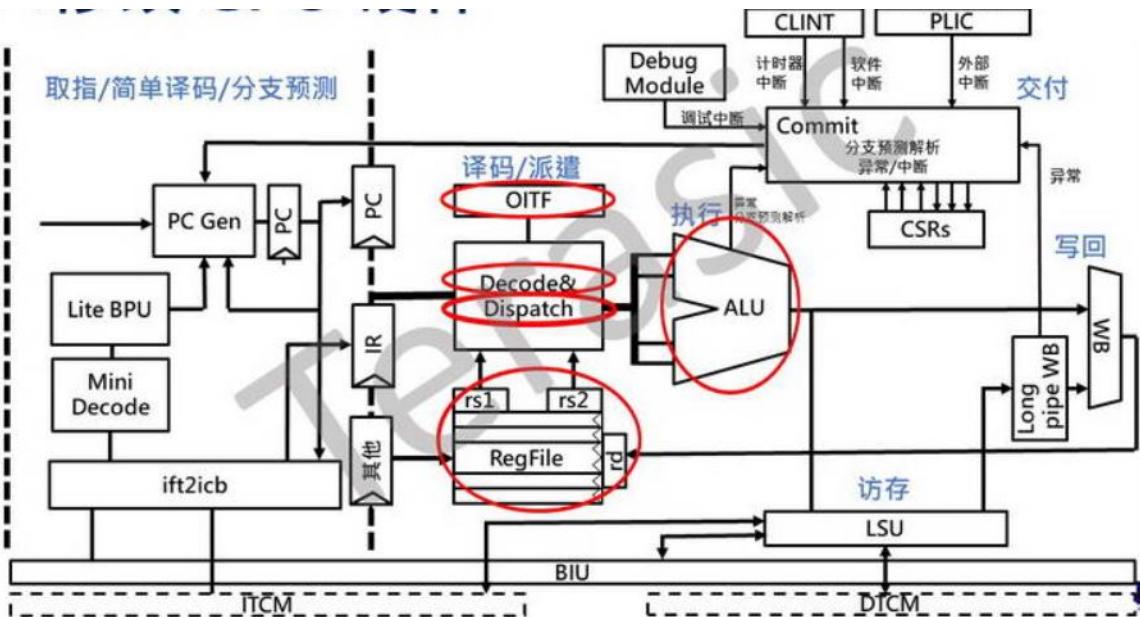


图 1: 硬件数据通路图及修改部分

宏定义修改

为了增加 ALU 的多路选择数，我们在宏定义中增加三条选择线路，分别连接 DOT4、DOT3、DOT2 的自定义逻辑功能模块，对应的修改为将 ALU 的宏定义宽度加 3：

```

381 `define E203_DECINFO_ALU_NOP    E203_DECINFO_ALU_NOP_MSB : E203_DECINFO_ALU_NOP_LSB
382 `define E203_DECINFO_ALU_ECAL_LSB ('E203_DECINFO_ALU_NOP_MSB+1)
383 `define E203_DECINFO_ALU_ECAL_MSB ('E203_DECINFO_ALU_ECAL_LSB+1-1)
384 `define E203_DECINFO_ALU_ECAL    E203_DECINFO_ALU_ECAL_MSB : E203_DECINFO_ALU_ECAL_LSB
385 `define E203_DECINFO_ALU_EBRK_LSB ('E203_DECINFO_ALU_ECAL_MSB+1)
386 `define E203_DECINFO_ALU_EBRK_MSB ('E203_DECINFO_ALU_EBRK_LSB+1-1)
387 `define E203_DECINFO_ALU_EBRK    E203_DECINFO_ALU_EBRK_MSB : E203_DECINFO_ALU_EBRK_LSB
388 `define E203_DECINFO_ALU_WFI_LSB ('E203_DECINFO_ALU_EBRK_MSB+1)
389 `define E203_DECINFO_ALU_WFI_MSB ('E203_DECINFO_ALU_WFI_LSB+1-1)
390 `define E203_DECINFO_ALU_WFI    E203_DECINFO_ALU_WFI_MSB : E203_DECINFO_ALU_WFI_LSB
391
392 // define three macro for decoder use.
393 `define E203_DECINFO_ALU_DOT_LSB ('E203_DECINFO_ALU_WFI_MSB+1)
394 `define E203_DECINFO_ALU_DOT_MSB ('E203_DECINFO_ALU_DOT_LSB+1-1)
395 `define E203_DECINFO_ALU_DOT    E203_DECINFO_ALU_DOT_MSB : E203_DECINFO_ALU_DOT_LSB
396 `define E203_DECINFO_ALU_WIDTH ('E203_DECINFO_ALU_DOT_MSB+1)
397

```

图 2: 宏定义修改说明

译码模块修改

将 DOT4、DOT3、DOT2 的指令字段确定之后，给出其译码的条件，并将其放入 ALU 运算集合之中。之后还需将这三个指令的译码信息放入总线用于数据交换。

```

398 // add the dot signal
399 // dot instruction format:
400 // 0000001(fun7) rd(5 bits) 000(fun3) rs1(5 bits) rs2(5bits) 1101011(opcode)
401 wire rv32_dot = rv32_resved0 & rv32_func3_000 & rv32_func7_0000001;

407 wire alu_op = (~rv32_sxxi_shamt_ilg1) & (~rv16_sxxi_shamt_ilg1)
408 & (~rv16_li_lui_ilg1) & (~rv16_addi4spn_ilg1) & (~rv16_addi16sp_ilg1) &
409 | rv32_op_imm
410 | rv32_op & (~rv32_func7_0000001) // Exclude the MULDIV
411 | rv32_auipc
412 | rv32_lui
413 | rv16_addi4spn
414 | rv16_addi
415 | rv16_lui_addi16sp
416 | rv16_li | rv16_mv
417 | rv16_slli
418 | rv16_miscalu
419 | rv16_add
420 | rv16_nop | rv32_nop
421 | rv32_wfi // We just put WFI into ALU and do nothing in ALU
422 ecall_ebreak
423
424 // connect alu_op singal with rv32_dot singal.
425 | rv32_dot)
426 ;
427

454
455 // add dot_to_alu_info_bus.
456 assign alu_info_bus[E203_DECINFO_ALU_DOT] = rv32_dot;
457

```

图 3: 译码模块修改说明

寄存器堆模块修改

该处的修改较为简单，主要工作是声明 a0-a7 这 8 个端口，并分别对应到寄存器堆中 x10-x17 这 8 个寄存器。这里需要强调一下，使用这 8 个寄存器是由于他们均为函数调用的传参寄存器，在运行过程中使用几率较小，所以产生冲突的可能性更小，从而保证更高的处理效率。

```

28 module e203_exu_regfile(
29   input [`E203_RFIDX_WIDTH-1:0] read_src1_idx,
30   input [`E203_RFIDX_WIDTH-1:0] read_src2_idx,
31   output [`E203_XLEN-1:0] read_src1_dat,
32   output [`E203_XLEN-1:0] read_src2_dat,
33
34   input wbck_dest_wen,
35   input [`E203_RFIDX_WIDTH-1:0] wbck_dest_idx,
36   input [`E203_XLEN-1:0] wbck_dest_dat,
37
38   output [`E203_XLEN-1:0] x1_r,
39
40   input test_mode,
41   input clk,
42   input rst_n,
43
44   // output the value of the registers a0 ~ a7.
45   output [`E203_XLEN-1:0] read_a0_dat,
46   output [`E203_XLEN-1:0] read_a1_dat,
47   output [`E203_XLEN-1:0] read_a2_dat,
48   output [`E203_XLEN-1:0] read_a3_dat,
49   output [`E203_XLEN-1:0] read_a4_dat,
50   output [`E203_XLEN-1:0] read_a5_dat,
51   output [`E203_XLEN-1:0] read_a6_dat,
52   output [`E203_XLEN-1:0] read_a7_dat,
53 );
54

99 assign read_src1_dat = rf_r[read_src1_idx];
100 assign read_src2_dat = rf_r[read_src2_idx];
101
102 // get the value from rf_r which are the real registers.
103 assign read_a0_dat = rf_r[10];
104 assign read_a1_dat = rf_r[11];
105 assign read_a2_dat = rf_r[12];
106 assign read_a3_dat = rf_r[13];
107 assign read_a4_dat = rf_r[14];
108 assign read_a5_dat = rf_r[15];
109 assign read_a6_dat = rf_r[16];
110 assign read_a7_dat = rf_r[17];
111
112 assign x1_r = rf_r[1];

```

图 4: 寄存器堆修改说明

OITF 模块修改

e203 解决结构冒险是用到了握手机制。用一个 op_ready 信号 & op 信号进行判断，若为 1 就可以派遣，若不为 1 就不能派遣。若正在执行乘法，下一条还是乘法，此时 op_ready 就是 0，op 就是 1，与后等于 0，无法派遣。如果正在执行乘法，下一条是跳转指令 op_ready 就是 1，op 为 1，与后等于 1，可以派遣。

运用这个机制，我们对我们所使用的 8 个寄存器进行冲突判断。在 e203 中，有一种特殊定义的长指令，在派遣后还需执行几个周期，访存相关的指令。e203 在执行长指令的时候，OITF 指令会把相关信息压入 OITF

中，执行完释放，这叫做维护或数据依赖性，维护期间如果来了一个非长指令，就很有可能冲突。所以 OITF 就是用来判断是否会发生长指令和非长指令的冲突。

```

62 // output the hazard signal of the registers a0 ~ a7.
63 output otifrd_match_dispa0,
64 output otifrd_match_dispa1,
65 output otifrd_match_dispa2,
66 output otifrd_match_dispa3,
67 output otifrd_match_dispa4,
68 output otifrd_match_dispa5,
69 output otifrd_match_dispa6,
70 output otifrd_match_dispa7;

181 // the instructions in pitf match the registers a0 ~ a7
182 assign rd_match_a0[i] = vld_r[i] & rdwen_r[i] & (rdidx_r[i] == E203_RFIDX_WIDTH'b01010);
183 assign rd_match_a1[i] = vld_r[i] & rdwen_r[i] & (rdidx_r[i] == E203_RFIDX_WIDTH'b01011);
184 assign rd_match_a2[i] = vld_r[i] & rdwen_r[i] & (rdidx_r[i] == E203_RFIDX_WIDTH'b01100);
185 assign rd_match_a3[i] = vld_r[i] & rdwen_r[i] & (rdidx_r[i] == E203_RFIDX_WIDTH'b01101);
186 assign rd_match_a4[i] = vld_r[i] & rdwen_r[i] & (rdidx_r[i] == E203_RFIDX_WIDTH'b01110);
187 assign rd_match_a5[i] = vld_r[i] & rdwen_r[i] & (rdidx_r[i] == E203_RFIDX_WIDTH'b01111);
188 assign rd_match_a6[i] = vld_r[i] & rdwen_r[i] & (rdidx_r[i] == E203_RFIDX_WIDTH'b10000);
189 assign rd_match_a7[i] = vld_r[i] & rdwen_r[i] & (rdidx_r[i] == E203_RFIDX_WIDTH'b10001);

199 // if any value in rd match a* is one, then otifrd_match_dispa* is one.
200 assign otifrd_match_dispa0 = |rd_match_a0;
201 assign otifrd_match_dispa1 = |rd_match_a1;
202 assign otifrd_match_dispa2 = |rd_match_a2;
203 assign otifrd_match_dispa3 = |rd_match_a3;
204 assign otifrd_match_dispa4 = |rd_match_a4;
205 assign otifrd_match_dispa5 = |rd_match_a5;
206 assign otifrd_match_dispa6 = |rd_match_a6;
207 assign otifrd_match_dispa7 = |rd_match_a7;

```



图 5: OITF 模块修改说明

派遣模块修改

派遣模块需要修改两个部分，一个是针对 OITF 反馈的派遣信息的响应，一个是针对寄存器值派遣给 ALU 模块的赋值过程。

```

102 // get the hazard singal of the registers a0 ~ a7 from oitf.
103 input otifrd_match_dispa0,
104 input otifrd_match_dispa1,
105 input otifrd_match_dispa2,
106 input otifrd_match_dispa3,
107 input otifrd_match_dispa4,
108 input otifrd_match_dispa5,
109 input otifrd_match_dispa6,
110 input otifrd_match_dispa7;

209 // add the new signal dot_inst to judge whether the current instruction is dot.
210 wire dot_inst = (disp_i_info[E203_DECINFO_GRP] == E203_DECINFO_GRP_ALU) ? disp_i_info[E203_DECINFO_ALU_DOT] : 1'b0;
211
212 wire raw_dep = ((otifrd_match_disprs1) |
213 (otifrd_match_disprs2) |
214 (otifrd_match_disprs3) |
215
216 // if any registers(a0 ~ a7) match otifrd, then the raw_dep signal is true.
217 (dot_inst &
218 ((otifrd_match_dispa0) |
219 (otifrd_match_dispa1) |
220 (otifrd_match_dispa2) |
221 (otifrd_match_dispa3) |
222 (otifrd_match_dispa4) |
223 (otifrd_match_dispa5) |
224 (otifrd_match_dispa6) |
225 (otifrd_match_dispa7)))
226 );
227
228

```

图 6: OITF 反馈的派遣信息的响应

```

58 // get the value of the registers a0 ~ a7 from regfile.
59 input [E203_XLEN-1:0] disp_i_a0,
60 input [E203_XLEN-1:0] disp_i_a1,
61 input [E203_XLEN-1:0] disp_i_a2,
62 input [E203_XLEN-1:0] disp_i_a3,
63 input [E203_XLEN-1:0] disp_i_a4,
64 input [E203_XLEN-1:0] disp_i_a5,
65 input [E203_XLEN-1:0] disp_i_a6,
66 input [E203_XLEN-1:0] disp_i_a7,
67
68 // pass the value of the registers a0 ~ a7 to the alu.
69 output [E203_XLEN-1:0] disp_o_alu_a0,
70 output [E203_XLEN-1:0] disp_o_alu_a1,
71 output [E203_XLEN-1:0] disp_o_alu_a2,
72 output [E203_XLEN-1:0] disp_o_alu_a3,
73 output [E203_XLEN-1:0] disp_o_alu_a4,
74 output [E203_XLEN-1:0] disp_o_alu_a5,
75 output [E203_XLEN-1:0] disp_o_alu_a6,
76 output [E203_XLEN-1:0] disp_o_alu_a7,
77
78 // connect disp_o_alu_a* with disp_i_a*.
79 assign disp_o_alu_a0 = disp_i_a0;
80 assign disp_o_alu_a1 = disp_i_a1;
81 assign disp_o_alu_a2 = disp_i_a2;
82 assign disp_o_alu_a3 = disp_i_a3;
83 assign disp_o_alu_a4 = disp_i_a4;
84 assign disp_o_alu_a5 = disp_i_a5;
85 assign disp_o_alu_a6 = disp_i_a6;
86 assign disp_o_alu_a7 = disp_i_a7;

```

图 7: 寄存器值派遣给 ALU 模块的赋值过程

ALU 模块修改

首先需要接收到来自总线上关于 DOT 指令的控制信息，来指示 ALU 当前需要执行 DOT 指令的操作。接着我们在 ALU 的数据通路模块中增加自定义逻辑模块，将我们想实现的三种子块矩阵乘法写入相应位置中。在最后我们需要将 ALU 输出的结果存入相应的寄存器中，来保证结果被缓存而不被覆盖。

```

47      input
48      input
49      input
50      input
51      input
52      input
53      input
54      input
55      input
56      [E203_XLEN-1:0] alu_req_alu_dot,
57      [E203_XLEN-1:0] alu_req_alu_a0,
58      [E203_XLEN-1:0] alu_req_alu_a1,
59      [E203_XLEN-1:0] alu_req_alu_a2,
60      [E203_XLEN-1:0] alu_req_alu_a3,
61      [E203_XLEN-1:0] alu_req_alu_a4,
62      [E203_XLEN-1:0] alu_req_alu_a5,
63      [E203_XLEN-1:0] alu_req_alu_a6,
64      [E203_XLEN-1:0] alu_req_alu_a7,
65
66
67 // the implementation of the dot computation.
68 // the implement is very violent and simple.
69 // put four multipliers and three adders in dpath.
70 wire [E203_XLEN-1:0] a0_a4_mul; // a0 * a4
71 wire [E203_XLEN-1:0] a1_a5_mul; // a1 * a5
72 wire [E203_XLEN-1:0] a2_a6_mul; // a2 * a6
73 wire [E203_XLEN-1:0] a3_a7_mul; // a3 * a7
74 wire [E203_XLEN-1:0] add_1_2; // a0 * a4 + a1 * a5
75 wire [E203_XLEN-1:0] add_3_4; // a2 * a6 + a3 * a7
76
77 assign a0_a4_mul = alu_req_alu_a0 * alu_req_alu_a4;
78 assign a1_a5_mul = alu_req_alu_a1 * alu_req_alu_a5;
79 assign a2_a6_mul = alu_req_alu_a2 * alu_req_alu_a6;
80 assign a3_a7_mul = alu_req_alu_a3 * alu_req_alu_a7;
81 assign add_1_2 = a0_a4_mul + a1_a5_mul;
82 assign add_3_4 = a2_a6_mul + a3_a7_mul;
83 wire [E203_XLEN-1:0] dot_res = add_1_2 + add_3_4; // the dot result
84
85
86 168 // the op_dot is same with alu_req_alu_dot.
87 169 wire op_dot = alu_req_alu_dot;
88
89
90
91
92
93
94 // Generate the final result
95 wire [E203_XLEN-1:0] alu_dpath_res =
96   ({ E203_XLEN{op_or} } & orer_res )
97   ({ E203_XLEN{op_and} } & ander_res )
98   ({ E203_XLEN{op_xor} } & xorer_res)
99   ({ E203_XLEN{op_addsub} } & adder_res[E203_XLEN-1:0])
100  ({ E203_XLEN{op_srl} } & srl_res)
101  ({ E203_XLEN{op_sll} } & sll_res)
102  ({ E203_XLEN{op_sra} } & sra_res)
103  ({ E203_XLEN{op_mvop2} } & mvop2_res)
104  ({ E203_XLEN{op_slttu} } & slttu_res)
105  | ({ E203_XLEN{op_max} | op_maxu | op_min | op_minu } & maxmin_res)
106
107 // add new option dot to return dot_res value.
108 | ({ E203_XLEN{op_dot} } & dot_res)
109
110 ;

```

图 8: DOT4 的自定义逻辑实现

以上便是所有硬件修改的流程，在完成修改之后，我们需要对 Quartus 工程项目进行编译，若编译成功，证明修改完全正确。编译之后会在 outfile 文件夹中得到 pof 文件，这个文件正是需要我们烧写进 T-Core 的文件。

烧写 POF 文件到 T-Core 开发板

- 设置 T-Core 开发板的 SW2 开关: SW2.1=0, SW2.2=1。

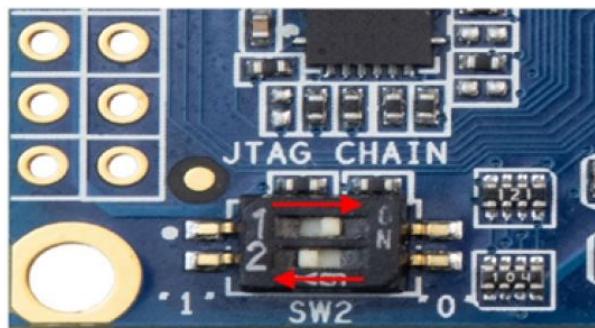


图 9: JTAG 切换模式

- 用 Mini USB 线连接 T-Core 开发板的 J2 接口与主机。
- 打开 Quartus 的 Programmer 工具，点击 Hardware Setup...，选择 T-Core，然后点击 Auto Detect。选择当前 T-Core 开发板的 MAX 10 FPGA 器件，点击 change file 按钮并选择刚刚生成好的 POF 文件，勾号选项之后点击 start 开始烧写。

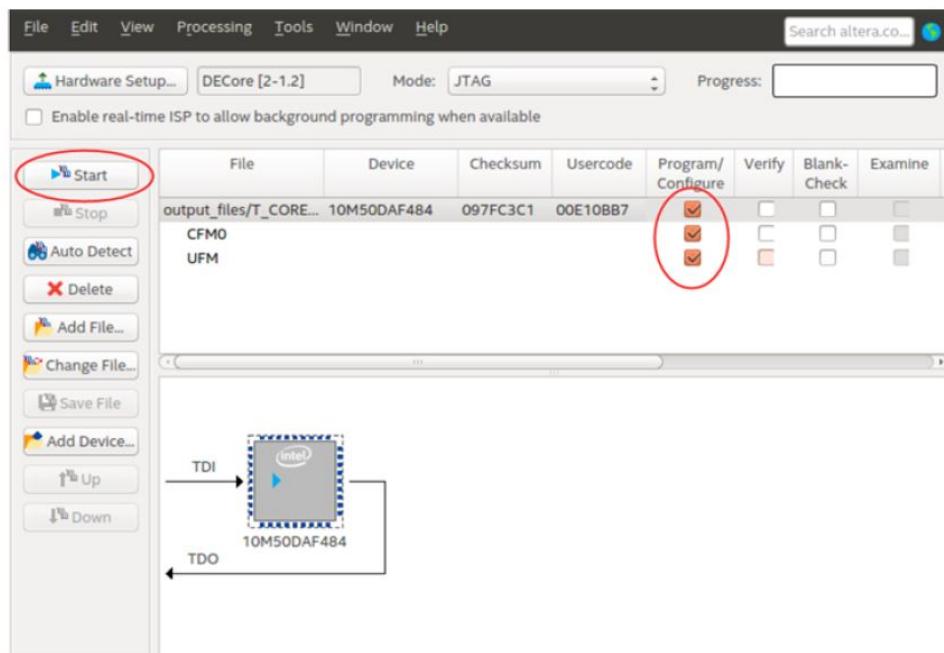


图 10: PROGRAMMER 界面操作一览

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。

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。

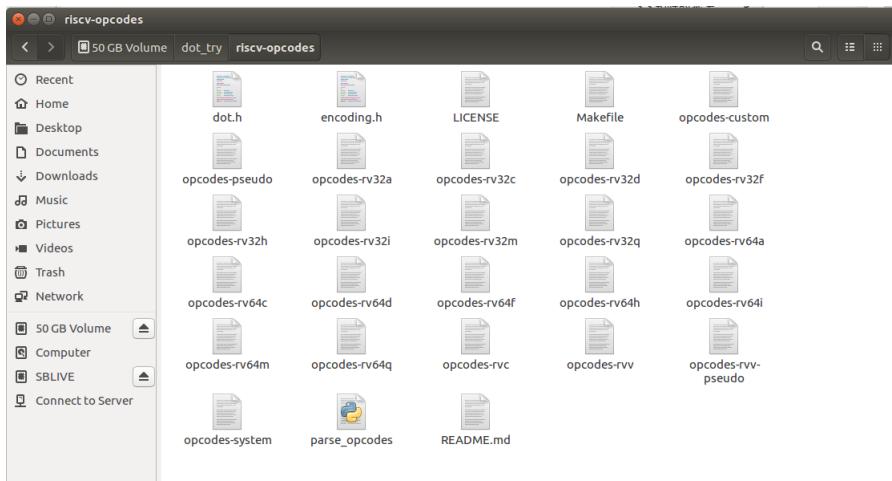


图 11: riscv-opcodes 文件包

```

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

```

图 13: 用脚本转换 opcodes

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。

```
#define MATCH_OR 0x0033
#define MASK_OR 0xfe00707f
#define MATCH_AND 0x7033
#define MASK_AND 0xfe00707f
#define MATCH_DOT4 0x2000006b
#define MASK_DOT4 0xfe00707f
#define MATCH_DOT3 0x20000057
#define MASK_DOT3 0xfe00707f
#define MATCH_DOT2 0x20000077
#define MASK_DOT2 0xfe00707f
#define MATCH_ADDIW 0x1b
#define MASK_ADDIW 0x707f
.
```

图 14: 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 15: 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.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 16: 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
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/rv32_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 17: 编译工具链

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 18: 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。

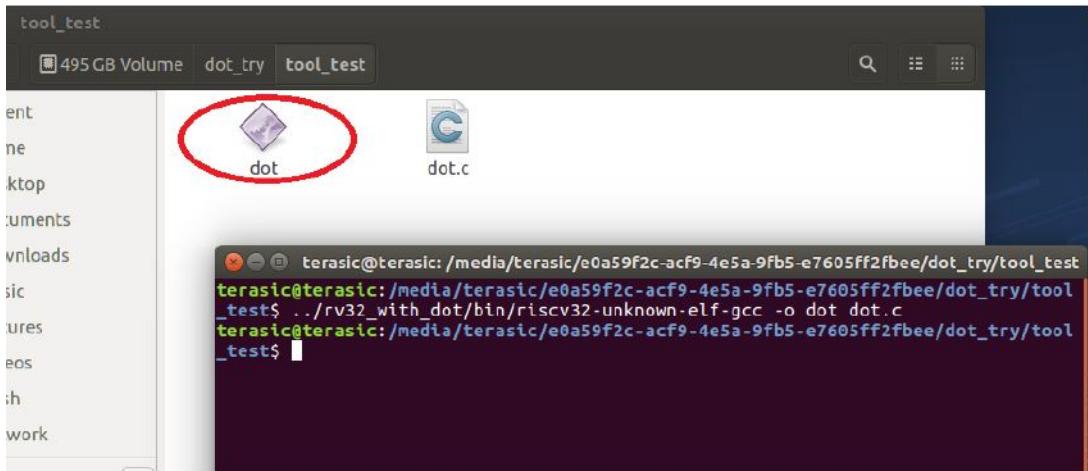


图 19: 调用 gcc 编译

- 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 20: 查看 dot 指令

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>
7

```

- 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
// CONST_K must set as multiple of 4 for dot instruction
#define CONST_I 32
#define CONST_K 64
#define CONST_J 32
//a[const_i][const_k]
#define A_ROW CONST_I
#define A_COL CONST_K
//b[const_k][const_j]
#define B_ROW CONST_K
#define B_COL CONST_J
//res[const_i][const_j]
#define RES_ROW CONST_I
#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。

```

2     int main(int argc, char *argv[]) {
3         uart_init();
4         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。

```

1 // malloc and init matrixs
2 printf("Malloc and initial Matrixs!!\n\n");
3 // matrix stored as an array of size row * column
4 int *a = NULL, *b = NULL, *no_dot_res = NULL, *dot_res = NULL;
5 a = (int*) malloc(A_ROW * A_COL * sizeof(int));
6 b = (int*) malloc(B_ROW * B_COL * sizeof(int));
7 no_dot_res = (int*) malloc(RES_ROW * RES_COL * sizeof(int));
8 dot_res = (int*) malloc(RES_ROW * RES_COL * sizeof(int));
9

```

- 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: %.2fms\n",
22          (float)no_dot_timer_cost/RTC_FREQ*1000);
23      printf("not_dot_cycle: %u\n", no_dot_cycle_cost);
24      printf("not_dot_instret: %u\n", no_dot_instret_cost);
25      printf("not_dot CPI: %.2f\n\n",
26          (float)no_dot_cycle_cost/no_dot_instret_cost);
27

```

- 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。

```

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

```

```

30         , "x14", "x15", "x16", "x17"
31     );
32
33     dot_res[ i * RES_COL + j ] += reg;
34     k += 4;
35
36     asm volatile (
37         "lw x10, %[a0]\n"
38         "lw x11, %[a1]\n"
39         "lw x12, %[a2]\n"
40         "lw x13, %[a3]\n"
41         "lw x14, %[a4]\n"
42         "lw x15, %[a5]\n"
43         "dot3 %[output], x12, x13\n"
44         : [output] "=r"(reg)
45         : [a0] "m"(a[i * A_COL + (k + 0)])
46         ,[a1] "m"(a[i * A_COL + (k + 1)])
47         ,[a2] "m"(a[i * A_COL + (k + 2)])
48         ,[a3] "m"(b[(k + 0) * B_COL + j])
49         ,[a4] "m"(b[(k + 1) * B_COL + j])
50         ,[a5] "m"(b[(k + 2) * B_COL + j])
51         : "x10", "x11", "x12", "x13"
52         , "x14", "x15"
53     );
54
55     dot_res[ i * RES_COL + j ] += reg;
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

3.1 编译并上传 C 语言程序

- 编译

进入 dot_try/TCORE-RISCV-E203/TRRV-E-SDK 路径下的 software 文件夹，右键打开 terminal 终端，输入 make software PROGRAM=demo_dot 命令编译应用程序
- 连接 T-Core 及 SIF 子卡
 - 关闭 T-Core 开发板电源后，将开发板上的 SW2 设置为 SW2.1=1, SW2.2=0，选择 RISC-V JTAG 链路。
 - 将 SIF 子卡连接到 T-Core 的 TMD 2×6 header (JP6)，并使用 USB miniB 线缆将 SIF 子卡与 PC 连接。
 - 将 USB Blaster II 线缆的一端连接到开发板的 USB Blaster 接口 (J2)，另一端连接至 PC 主机的 USB 接口。



- 打开串口调试工具

使用 sudo minicom 命令打开 linux 系统的串口调试工具。
- 上传程序并执行

使用 make upload PROGRAM=demo_dot 将可执行文件 demo_dot 下载到 T-Core 开发板的 QSPI Flash 中。

3.2 运行结果

- 常规矩阵运算与自定义矩阵乘法指令的运算耗时对比 (49×49 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

4.1 可以改进的地方

对于 1×4 乘以 4×1 的矩阵乘法而言，会需要用到 8 个寄存器。我们可以在此基础上增加更多的寄存器以在一个周期内实现更大的矩阵乘法操作，例如 1×5 乘以 5×1 等，甚至可以到 1×8 乘以 8×1 ，用到 16 个寄存器（这种情况应该是行不通的）。这种思想便是占用更多的空间开销来换取时间上的效率提升。

4.2 对 FPGA 的理解

FPGA 是可编程逻辑器件，我们通过 Risc-V 指令集和 FPGA 开发中领悟到，如今的硬件设计越来越自主化和多样化，甚至可以进行自定义逻辑指令和相应的硬件数据通路设计。这种极高的自由度给计算机行业带来新的机遇，对于遇到诸多瓶颈的软件算法优化而言，利用 FPGA 硬件设计加速优化软件算法，会成为突破现有瓶颈的关键技术。如今可以在诸多论文中发现硬件设计的比重越来越大，人们逐渐将理论算法优化中无法解决的问题尝试带到硬件设计进行优化。