浙江大学

矩阵乘法加速器设计报告

课程名称：	计算机组成与系统结构
成　　员：	曹逸芃 3180102927
	陶冠辰 3180102156
	张铁沄 3180105190
学　　院：	信息与电子工程学院
指导教师：	黄科杰

2021 年 6 月 26 日

一、设计思路

1. 基于 Sodor 的矩阵乘法器

Sodor 是基于 RISC-V 的教学用处理器，其结构较为简单，可以执行一些基本的 RISC-V 指令。其不仅有单周期的 1stage 处理器，还有包含流水线设计的 3stage 和 5stage 等，同时整体的硬件设计并不复杂，非常适合我们在此基础上进行矩阵乘法器的设计以及修改。因此我们决定以 Sodor 为基础来构造矩阵乘法加速器。

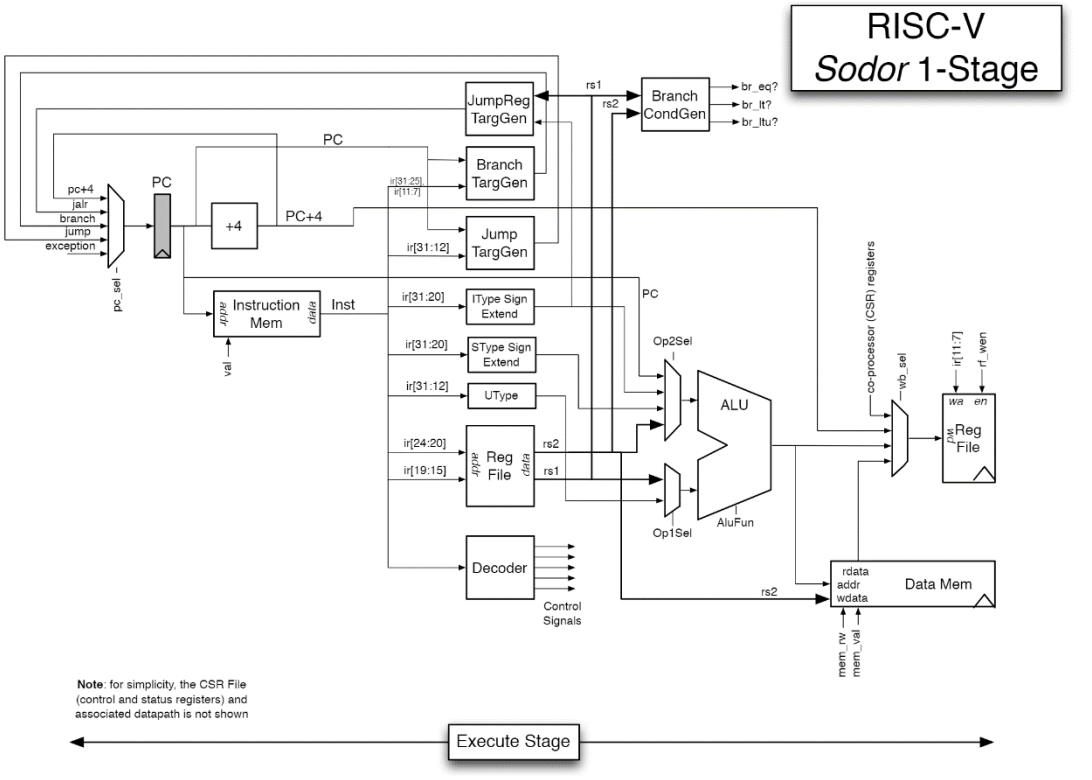

图 1 Sodor_1stage 结构

利用 Sodor 进行矩阵乘法加速器设计的一个较为简单的思路是在利用 RISC-V 指令在 Sodor 处理器上直接实现矩阵乘法。矩阵乘法需要利用的 RISC-V 指令中的乘法以及加法，但是 Sodor 处理器目前支持加法指令，不支持乘法指令。因此我们决定在原有的 Sodor 处理器基础上加入乘法指令 MUL，由 RISC-V 编译器将我们写的矩阵运算的 C 代码编译成 RISC-V 指令，再由修改后的 Sodor 处理器执行，实现矩阵乘法计算。这一思路的关键点在于将乘法指令引入 Sodor 处理器，令其正常执行。

2. 脉动阵列加速器

在 Sodor 中直接引入乘法指令来完成矩阵乘法计算的实现方式较为简单，但是缺点在于效率较低。因此我们打算采取的另一种方法是在原有的 Sodor 基础上引入脉动阵列单元，通过脉动阵列单元来单独对矩阵乘法进行优化加速，并且设计相应的新指令来控制脉动阵列完成矩阵乘法运算。

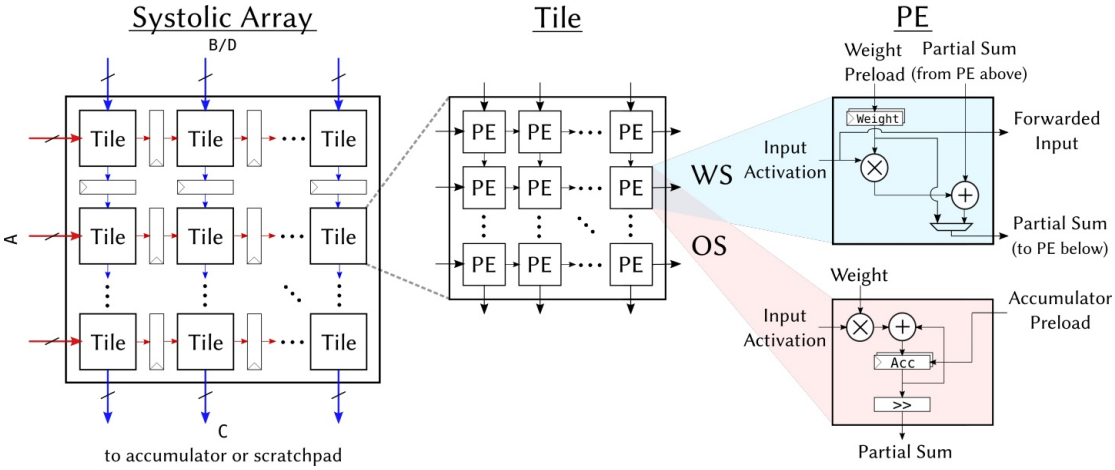

图 2 脉动阵列

脉动阵列是一种复用输入数据的设计，由多个乘加单元构成。在执行矩阵加法运算时，矩阵 B 会首先以广播的形式传播并储存在脉动阵列内部。之后再依次输入矩阵 A 的 N 行，由脉动阵列中的乘加单元按顺序执行计算，并最终得到矩阵乘法的计算结果。脉动阵列可以对矩阵乘法起到很好地加速效果，在同规模下大大缩减矩阵乘法所需要的周期数。

二、基于 Sodor 的矩阵乘法器

1. 添加指令

在设计思路中我们分析了，想要在 Sodor 中实现矩阵乘法，需要在处理器中引入 MUL 乘法指令。由于 RISC-V 指令集中本身就是包含乘法指令 MUL 的，其属于 M 类指令，因此再专门为乘法设计新的 RISC-V 指令并无必要。所以我们打算在编译生成 RISC-V 指令时直接调用包含 M 类指令的 rv32im 指令集，这样利用 RISC-V 工具链我们就可以直接将测试用的 C 代码编译成包含 MUL 的 RISC-V 指令了。为了实现上述想法，我们需要修改编译用的 Makefile 文件：

```
RISCV_PREFIX ?= riscv$(XLEN)-unknown-elf-
RISCV_GCC ?= $(RISCV_PREFIX)gcc
RISCV_GCC_OPTS ?= -DPREALLOCATE=1 -O2 -mcmodel=medany -static -std=gnu99 -ffast-math -fno-common -fno-
builtin-printf -march=rv32im -mabi=ilp32
RISCV_LINK ?= $(RISCV_GCC) -T $(src_dir)/common/test.ld $(incs)
RISCV_LINK_OPTS ?= -static -nostdlib -nostartfiles -lm -lgcc -T $(src_dir)/common/test.ld
RISCV_OBJDUMP ?= $(RISCV_PREFIX)objdump --disassemble-all --disassemble-zeroes --section=.text --secti
on=.text.startup --section=.data
RISCV_SIM ?= spike --isa=rv$(XLEN)gc
```

图 3 修改后的 Makefile

2. 验证指令

添加所需的指令后，需要验证在编译生成 RISC-V 指令时，是否会生成 MUL 等指令。为此我们写了一段简单的 C 代码，其中包含乘法计算，将其编译后，检查生成的 RISC-V 指令：

```
80002984:       00478793                addi    a5,a5,4
80002988:       10068693                addi    a3,a3,256
8000298c:       02b70733                mul     a4,a4,a1
80002990:       00e60633                add     a2,a2,a4
```

图 4 生成的 RISC-V 指令

可以看到 MUL 指令已正常产生了。但是我们进一步检查发现，由于测试代码中还包含一些其他的部分，比如周期数的计算函数，结果的正确性校验函数等等，因此还产生了一些同为 M 类指令的 DIVU 等指令。这些执行同样是目前的 Sodor 所不能执行的，因此我们决定直接在 Sodor 中加入对这些指令的支持。

3. 修改 Sodor

经过检查，需要在 Sodor 处理器中加入如下三条指令的支持和执行逻辑：

mul rd, rs1, rs2 $x[rd] = x[rs1] \times x[rs2]$

乘(Multiply). R-type, RV32M and RV64M.

把寄存器 x[*rs2*]乘到寄存器 x[*rs1*]上，乘积写入 x[*rd*]。忽略算术溢出。

31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	000	rd	0110011

图 5 MUL 指令

divu rd, rs1, rs2 $x[rd] = x[rs1] \div_u x[rs2]$

无符号除法(Divide, Unsigned). R-type, RV32M and RV64M.

用寄存器 x[*rs1*]的值除以寄存器 x[*rs2*]的值，向零舍入，将这些数视为无符号数，把商写入 x[*rd*]。

31 25	24 20	19 15	14 12	11 7	6 0
0000001	rs2	rs1	101	rd	0110011

图 6 DIVU 指令

remu rd, rs1, rs2　　x[rd] = x[rs1] %$_u$ x[rs2]

求无符号数的余数(Remainder, Unsigned). R-type, RV32M and RV64M.

x[*rs1*]除以 x[*rs2*]，向 0 舍入，都视为无符号数，余数写入 x[*rd*]。

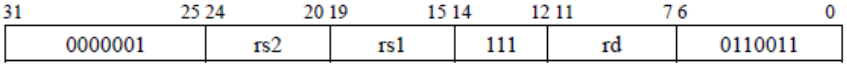

图 7 REMU 指令

确定了需要在 Sodor 中引入的指令，我们需要的 Sodor 的数据通路作出如下修改：

instructions.scala 中指令定义部分：

```
  def MUL                = BitPat("b0000001??????????000?????0110011")
  def DIVU               = BitPat("b0000001??????????101?????0110011")
  def REMU               = BitPat("b0000001??????????111?????0110011")
```

consts.scala 中的常量定义：

```
   val ALU_MUL      = 12.asUInt(5.W)
   val ALU_DIVU     = 13.asUInt(5.W)
   val ALU_REMU     = 14.asUInt(5.W)
```

cpath.scala 中的指令译码与控制信号的产生分配：

```
//译码
MUL->List(Y,BR_N,OP1_RS1,OP2_RS2,ALU_MUL,WB_ALU,REN_1,MEN_0,M_X,MT_X,CS
R.N),
DIVU->List(Y,BR_N,OP1_RS1,OP2_RS2,ALU_DIVU,WB_ALU,REN_1,MEN_0,M_X,MT_X,
CSR.N),
REMU->List(Y,BR_N,OP1_RS1,OP2_RS2,ALU_REMU,WB_ALU,REN_1,MEN_0,M_X,MT_X,
CSR.N),
```

dpath.scala 中的 ALU 设计：

```
val alu_out   = Wire(UInt(conf.xprlen.W))

val alu_shamt = alu_op2(4,0).asUInt

alu_out := MuxCase(0.U, Array(
      (io.ctl.alu_fun === ALU_MUL)  -> (alu_op1 * alu_op2).asUInt,
      (io.ctl.alu_fun === ALU_DIVU) -> (alu_op1 / alu_op2).asUInt,
      (io.ctl.alu_fun === ALU_REMU) -> (alu_op1 % alu_op2).asUInt,
      (io.ctl.alu_fun === ALU_ADD)  -> (alu_op1 + alu_op2).asUInt,
      (io.ctl.alu_fun === ALU_SUB)  -> (alu_op1 - alu_op2).asUInt,
```

```
       (io.ctl.alu_fun === ALU_AND)  -> (alu_op1 & alu_op2).asUInt,
       (io.ctl.alu_fun === ALU_OR)   -> (alu_op1 | alu_op2).asUInt,
       (io.ctl.alu_fun === ALU_XOR)  -> (alu_op1 ^ alu_op2).asUInt,
       (io.ctl.alu_fun === ALU_SLT)  -> (alu_op1.asSInt < alu_op2.asSIn
t).asUInt,
       (io.ctl.alu_fun === ALU_SLTU) -> (alu_op1 < alu_op2).asUInt,
       (io.ctl.alu_fun === ALU_SLL)  -> ((alu_op1 << alu_shamt)(conf.xp
rlen-1, 0)).asUInt,
       (io.ctl.alu_fun === ALU_SRA)  -> (alu_op1.asSInt >> alu_shamt).a
sUInt,
       (io.ctl.alu_fun === ALU_SRL)  -> (alu_op1 >> alu_shamt).asUInt,
       (io.ctl.alu_fun === ALU_COPY1)-> alu_op1
       ))
```

4. 测试代码

我们编写了相应的测试代码来验证我们设计的矩阵乘法器的正确性和性能：

```
#define N 4

int A[N][N], B[N][N], C[N][N];
int D[N][N] =
{
{0, 0, 0, 0},
{0, 4, 8, 12},
{0, 8, 16, 24},
{0, 12, 24, 36}
};

void init_matrix();

void serial_mul();


int main( int argc, char* argv[] )
{

    init_matrix();
    setStats(1);
    serial_mul();
    setStats(0);

    return verify(N*N, C, D);

}
```

```
void init_matrix(){
    for(int i = 0; i < N; i++){
        for(int j = 0; j < N; j++){
            A[i][j] = i;
        }
    }

    for(int i = 0; i < N; i++){
                for(int j = 0; j < N; j++){
                        B[i][j] = j;
                }
    }

    for(int i = 0; i < N; i++){
                for(int j = 0; j < N; j++){
                        C[i][j] = 0;
                }
        }
}

void serial_mul(){
    for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                    int temp = 0;
                    for (int k = 0; k < N; k++) {
                        temp += A[i][k] * B[k][j];
                    }
                    C[i][j] = temp;
            }
        }
}
```

其中 serial_mul()是矩阵乘法的 C 代码，测试相关函数 setStats()用于测试 serial_mul()所用的指令数和周期数。D 矩阵中放置了预先算好的正确的矩阵乘法结果，C 矩阵中则是由我们的矩阵乘法器算出来的结果。其中 return verify(N*N, C, D)将会比较验证 C 矩阵中结果的正确性，并返回相应的值。

5. Sodor_1stage 测试结果

我们将编写的测试代码用 RISC-V 工具链生成相应的 RISC-V 指令，并交由我们设计 Sodor 矩阵乘法器执行。我们共测试了4×4，16×16和64×64三种规模

的矩阵，结果如下：

（1）4×4 矩阵

```
mrrubiks@CYP-PC:~/code/chipyard/riscv-sodor/emulator/rv32_1stage$ make run-bmarks-test
mkdir -p output
./emulator +max-cycles=10000000 +verbose +loadmem=/home/mrrubiks/code/chipyard/riscv-sodor/riscv-tests
/benchmarks/mytest.riscv none 3>&1 1>&2 2>&3 | /home/mrrubiks/code/chipyard/riscv-tools-install/bin/sp
ike-dasm  > output/mytest.riscv.out
mcycle = 603
minstret = 609

  [ PASSED ] output/mytest.riscv.out
```

图 8 Sodor_1stage 4×4矩阵测试结果

$$CYCLE = 603$$

$$\mathrm{INST} = 609$$

$$\mathrm{CPI} = \frac{\mathrm{CYCLE}}{\mathrm{INST}} \approx 1$$

（2）16×16 矩阵

```
mrrubiks@CYP-PC:~/code/chipyard/riscv-sodor/emulator/rv32_1stage$ make run-bmarks-test
mkdir -p output
./emulator +max-cycles=10000000 +verbose +loadmem=/home/mrrubiks/code/chipyard/riscv-sodor/riscv-tests
/benchmarks/mytest.riscv none 3>&1 1>&2 2>&3 | /home/mrrubiks/code/chipyard/riscv-tools-install/bin/sp
ike-dasm  > output/mytest.riscv.out
mcycle = 30591
minstret = 30597

  [ PASSED ] output/mytest.riscv.out
```

图 9 Sodor_1stage 16×16矩阵测试结果

$$CYCLE = 30591$$

$$INST = 30597$$

$$CPI = \frac{CYCLE}{INST} \approx 1$$

（3）64×64 矩阵

```
mrrubiks@CYP-PC:~/code/chipyard/riscv-sodor/emulator/rv32_1stage$ make run-bmarks-test
mkdir -p output
./emulator +max-cycles=10000000 +verbose +loadmem=/home/mrrubiks/code/chipyard/riscv-sodor/riscv-tests
/benchmarks/mytest.riscv none 3>&1 1>&2 2>&3 | /home/mrrubiks/code/chipyard/riscv-tools-install/bin/sp
ike-dasm  > output/mytest.riscv.out
mcycle = 1864150
minstret = 1864156

  [ PASSED ] output/mytest.riscv.out
```

图 10 Sodor_1stage 64×64矩阵测试结果

$$CYCLE = 1864150$$

$$INST = 1864156$$

$$CPI = \frac{CYCLE}{INST} \approx 1$$

从结果中可以看到，我们设计的 Sodor 矩阵乘法器可以得到正确的矩阵乘法计算结果。此外，由于 Sodor_1stage 是简单的单周期处理器，因此矩阵乘法的 CPI 为 1。为了进一步验证基于 Sodor 处理器的矩阵乘法器，我们接下来又修改了 Sodor_5stage 并进行了测试。

6. Sodor_5stage

Sodor_5stage 采用了 5 级流水结构，并设计有旁路和分支预测。我们以类似 Sodor_1stage 的方式修改了 Sodor_5stage 的数据通路，并添加了指令。具体细节可以查看代码，此处不再赘述。之后以相同的测试代码进行了测试。

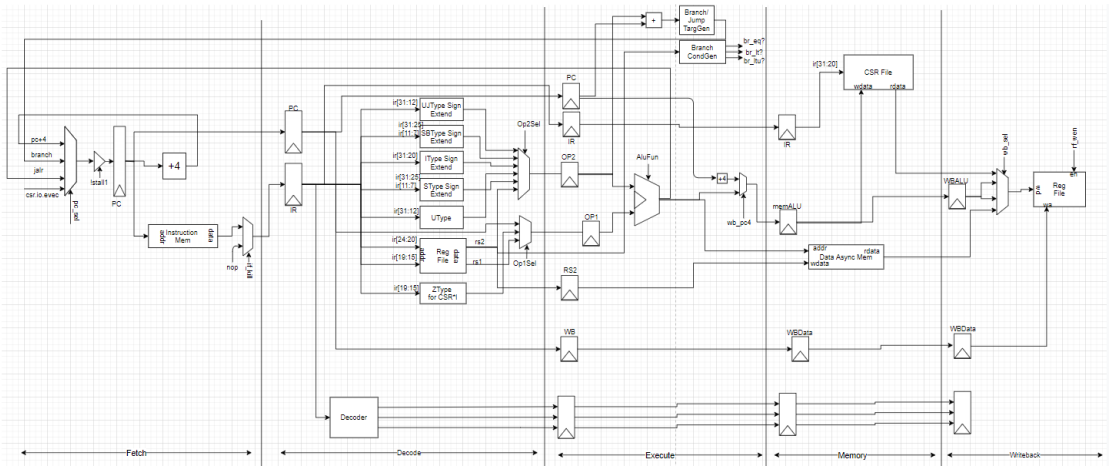

图 11 Sodor_5stage 结构

7. Sodor_5stage 测试结果

以相同的测试代码测试 Sodor_5stage，结果如下：

（1）4×4 矩阵

```
mrrubiks@CYP-PC:~/code/chipyard/riscv-sodor/emulator/rv32_5stage$ make run-bmarks-test
mkdir -p output
./emulator +max-cycles=10000000 +verbose +loadmem=/home/mrrubiks/code/chipyard/riscv-sodor/riscv-tests
/benchmarks/mytest.riscv none 3>&1 1>&2 2>&3 | /home/mrrubiks/code/chipyard/riscv-tools-install/bin/sp
ike-dasm  > output/mytest.riscv.out
mcycle = 739
minstret = 609

  [ PASSED ] output/mytest.riscv.out
```

图 12 Sodor_5stage 4×4矩阵测试结果

$$CYCLE = 739$$

$$\mathrm{INST} = 609$$

$$\mathrm{CPI} = \frac{\mathrm{CYCLE}}{\mathrm{INST}} \approx 1.213$$

（2）16×16 矩阵

```
mrrubiks@CYP-PC:~/code/chipyard/riscv-sodor/emulator/rv32_5stage$ make run-bmarks-test
mkdir -p output
./emulator +max-cycles=10000000 +verbose +loadmem=/home/mrrubiks/code/chipyard/riscv-sodor/riscv-tests
/benchmarks/mytest.riscv none 3>&1 1>&2 2>&3 | /home/mrrubiks/code/chipyard/riscv-tools-install/bin/sp
ike-dasm  > output/mytest.riscv.out
mcycle = 38791
minstret = 30597

  [ PASSED ] output/mytest.riscv.out
```

图 13 Sodor_5stage 16×16矩阵测试结果

$$CYCLE = 38791$$

$$\mathrm{INST} = 30597$$

$$\mathrm{CPI} = \frac{\mathrm{CYCLE}}{\mathrm{INST}} \approx 1.268$$

（3）64×64 矩阵

```
mrrubiks@CYP-PC:~/code/chipyard/riscv-sodor/emulator/rv32_5stage$ make run-bmarks-test
mkdir -p output
./emulator +max-cycles=10000000 +verbose +loadmem=/home/mrrubiks/code/chipyard/riscv-sodor/riscv-tests
/benchmarks/mytest.riscv none 3>&1 1>&2 2>&3 | /home/mrrubiks/code/chipyard/riscv-tools-install/bin/sp
ike-dasm  > output/mytest.riscv.out
mcycle = 2388446
minstret = 1864156

  [ PASSED ] output/mytest.riscv.out
```

图 14 Sodor_5stage 64×64矩阵测试结果

$$CYCLE = 2388446$$

$$\mathrm{INST} = 1864156$$

$$\mathrm{CPI} = \frac{\mathrm{CYCLE}}{\mathrm{INST}} \approx 1.281$$

从结果中可以看到，Sodor_5stage 处理器的矩阵乘法器同样可以得到正确的运算结果。同时，Sodor_5stage 在执行相同的乘法计算时，用的指令数与 Sodor_1stage 一样，但是会花费更多的周期数，导致 Sodor_5stage 的 CPI > 1，大约在 1.2。这是因为 Sodor_5stage 存在五级流水结构，在执行指令时有时会产生数据冒险现象，导致需要插入 bubbles 来解决，因此需要消耗更多的周期数。但是五级流水的优点在于关键路径上的延时更短，可以设计更高的时钟频率。因此整体来说在计算速度上是优于单周期的 Sodor_1stage 的。为了进一步提高矩阵乘法的效率，我们又尝试了脉动阵列。

三、脉动阵列设计

1. 指令设计

为了支持我们设计的脉动阵列，我们需要向 RISC-V 中添加一些新的指令来使用脉动阵列计算矩阵。为此我们设计了 TPUP 和 TPUA 两条指令，其中 TPUP 用于权重的传播，TPUA 用于计算。

```
# cyp_modify_begin
tpup    rd rs1 rs2 31..25=31 14..12=0 6..2=0x0C 1..0=3
tpua    rd rs1 rs2 31..25=31 14..12=1 6..2=0x0C 1..0=3
# cyp_modify_end
```

图 15 指令设计

在 RISC-V 工具链中添加我们设计的两条新指令的描述，由工具链生成相应的掩码等：

```
#define MATCH_TPUP 0x3e000033
#define MASK_TPUP  0xfe00707f
#define MATCH_TPUA 0x3e001033
#define MASK_TPUA  0xfe00707f
```

图 16 指令掩码

之后将指令掩码和描述添加到相应的文件中，并重新编译 RISC-V 工具链。

```
{"tpup",       0, {"M", 0},   "d,s,t",  MATCH_TPUP, MASK_TPUP, match_opcode, 0 },
{"tpua",       0, {"M", 0},   "d,s,t",  MATCH_TPUA, MASK_TPUA, match_opcode, 0 },
```

图 17 添加的指令

这样我们就把新设计的指令成功添加到工具链当中了。

2. 验证指令

新添加的指令使用 C 语言中内嵌汇编的方式调用，经过简单的测试，编译器可以正常地生成我们创建的 RISC-V 指令。

```
8000280c <main>:
8000280c:       00500793                li      a5,5
80002810:       00200713                li      a4,2
80002814:       3ee787b3                tpup    a5,a5,a4
80002818:       00000513                li      a0,0
8000281c:       00008067                ret
```

图 18 生成的新 RISC-V 指令

3. 整体设计

在原有的 Sodor 基础上引入脉动阵列单元，通过脉动阵列单元来单独对矩阵

乘法进行优化加速，并且设计相应的新指令来控制脉动阵列完成矩阵乘法运算。若执行矩阵相乘的指令则使用脉动阵列相关数据通路，如果使用普通指令则使用带有 ALU 的数据通路。相关控制由两个数据通路分别的 controller 或是控制信号实现。

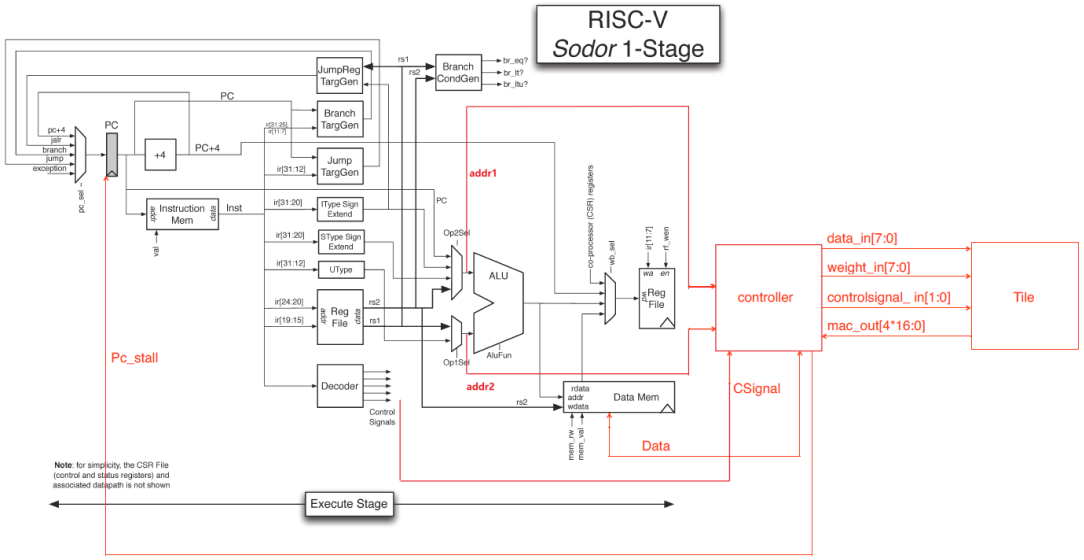

图 19 结合脉动阵列的 Sodor 结构

4. 脉动阵列设计

（1）PE 模块设计

PE 模块作为单个计算单元，将其设计为组合电路。其主要功能用于存储预先加载的 B 矩阵在寄存器中，并且有乘法、加法模块执行矩阵乘法与部分和累加。

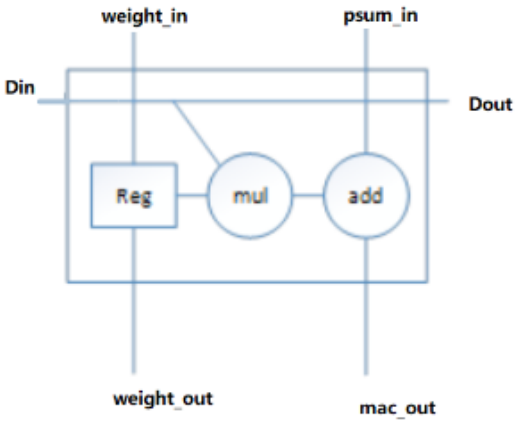

图 20 PE 结构

PE 模块代码：

```
package Sodor{

import chisel3._
import chisel3.util._
import Common._
import Common.Instructions._
import Constants._

class PEControl extends Bundle {
  val propagate =Bool()
  val matmul=Bool()
}

class PE extends Module //TODO:参数设计
{
    val io = IO(new Bundle {
    val weight_in = Input(UInt(8.W))//B 矩阵输入
    val din = Input(UInt(8.W))//A 矩阵输入
    val psum_in = Input(UInt(16.W))//部分和输入
    val mac_out = Output(UInt(16.W))//计算结果输出
    val dout = Output(UInt(8.W))//A 矩阵输出

    val in_control = Input(new PEControl)//控制信号输入
    val out_control = Output(new PEControl)//控制信号输出
```

PE 接口定义，各个输入输出如注释所示。

```
    io:=DontCare
   val c=Reg(UInt(8.W))//寄存器存放 B 矩阵
   val in_control_d=ShiftRegister(io.in_control,1)//将信号延时放在 PE 中方
便同步
   val psum_in_d  = ShiftRegister(io.psum_in, 1)
   val din_d  = ShiftRegister(io.din, 1)

   io.dout:=din_d
   io.out_control:=in_control_d

   when(io.in_control.propagate===true.B)
   {
      c:=io.weight_in//如果控制指令时 propagate，则在传播 B 矩阵
   }
   .elsewhen(io.in_control.matmul===true.B)
   {
```

```
      io.mac_out:=din_d*c+psum_in_d//如果控制指令时 matmul，则进行计算
   }
}
```

PE 中设置寄存器 c，用来存放 B 矩阵。PE 有两个状态，由 io.in_control.propagate，io.in_control.matmul 两个信号进行控制。如果控制指令时 matmul，则进行矩阵乘加计算。如果控制指令为 propagate，则进行 B 矩阵的预加载。

（2）Tile 模块设计

Tile 主要作用例化 PE 模块，并将 PE 模块连接，同时在 PE 模块间加入移位寄存器用以延时，同步计算过程。从数据输入到最后一个结果输出共经过 10 个周期。

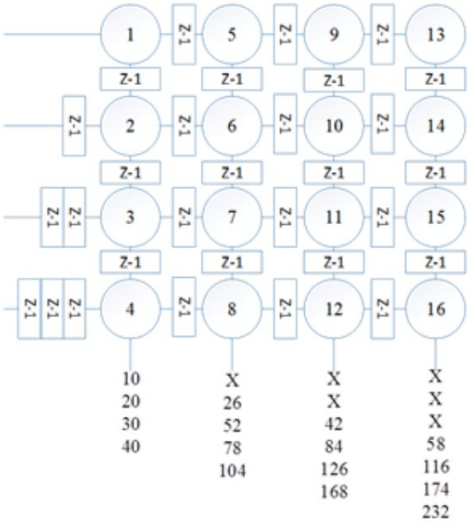

图 21 Tile 结构

Tile 模块关键部分代码：

```
package Sodor
{
import chisel3._
import chisel3.util._
import Common._
import Common.Instructions._
import Constants._

class Tiletoctl extends Bundle{
    val din        = Input(Vec(4, UInt(8.W))) \\Tile 的 A 矩阵输入
    val weight_in  = Input(Vec(16, UInt(8.W))) \\Tile 的 B 矩阵输入
    val in_control  = Input(Vec(4, new PEControl)) \\Tile 的控制信号输入
```

```
    val mac_out       = Output(Vec(4, UInt(16.W))) \\Tile 的计算结果输出
}
```

Tile 和 controller 接口的定义。

```
class Tileio extends Bundle{
    val ctlio = new Tiletoctl \\Tile 与 controller 的接口
    val dout       = Output(Vec(4, UInt(16.W))) \\Tile 中 A 矩阵的输出用于扩展
    val out_control=Output(Vec(4, new PEControl))
}
```

Tile 接口的定义。

```
class Tile extends Module {
  val io = IO(new Tileio)
  io:=DontCare

  val psum=RegInit(VecInit(0.U(8.W), 0.U(8.W),0.U(8.W),0.U(8.W)))

  val din_a=Reg(Vec(4,UInt(8.W)))
  din_a(0):=io.ctlio.din(0)
  din_a(1):=ShiftRegister(io.ctlio.din(1),1)//输入信号的延时
  din_a(2):=ShiftRegister(io.ctlio.din(2),2)
  din_a(3):=ShiftRegister(io.ctlio.din(3),3)
  val tile = Seq.fill(4,4)(Module(new PE))
  val tileT = tile.transpose
```

tile 为例化的二维 PE 阵列，tileT 将 tile 转置方便后面纵向接口的连线。用 ShiftRegister 将 din_a 信号进行延时，输入信号的同步。

```
  for(ii <- 0 until 3)
    for(jj <- 0 until 3)
    {
        tile(ii)(jj).io.weight_in:= io.ctlio.weight_in(ii*4+jj)//将 B 矩阵一起存进各个 PE 的寄存器当中
    }

  // TODO: abstract hori/vert broadcast, all these connections look the same
  // Broadcast 'a' horizontally across the Tile
```

```
  //将各个 PE 的 A 矩阵输入通道级联
  for (r <- 0 until 3) {
    tile(r).foldLeft(din_a(0)) {
      case (din_a, pe) =>
        pe.io.din :=din_a
        pe.io.dout
    }
  }

  // Broadcast 'psum' vertically across the Tile
  //将各个 PE 的部分和输入，计算结果输出级联
  for (c <- 0 until 3) {
    tileT(c).foldLeft(psum(0)) {
      case (psum, pe) =>
        pe.io.psum_in := psum
        pe.io.mac_out
    }
  }

  // Broadcast 'control' vertically across the Tile
  //将各个 PE 的控制信号输入输出级联
  for (c <- 0 until 3) {
    tileT(c).foldLeft(io.ctlio.in_control(c)) {
      case (in_ctrl, pe) =>
        pe.io.in_control := in_ctrl
        pe.io.out_control
    }
  }

  // Drive the Tile's bottom IO
  //驱动 Tile 模块底部的输出
  for (c <- 0 until 3) {
    io.ctlio.mac_out(c) := tile(3)(c).io.mac_out
    io.out_control(c) := tile(3)(c).io.out_control
  }

}
}
```

5. 控制器设计

脉动阵列的执行需要多个周期，并且其流水线的时序、与内存的通信均有较为复杂的规则，因此需要设计控制器在 dpath/memory 与 Tile 之间充当桥梁。

控制器的主要功能是从 dmem 中读取预存的矩阵数据，控制 Tile 中的计算模

块工作，并将计算结果写回 dmem 中相应的区域。控制器需要接收来至 dpath 的译码信号以确定其工作的启停。

下面是与控制器相关的 chisel 代码及其说明。

控制器、脉动阵列的实例化与连线：

```
  val c  = Module(new CtlPath())
  val d  = Module(new DatPath())
  val m  = Module(new TPUcontroler())
  val t  = Module(new Tile)
  d.io.tpu_ctl  <> m.io.dpathio
  t.io.ctlio <> m.io.tpathio
  io.dmem <> m.io.dmem

  io.dmem.req.bits.typ := m.io.dmem.req.bits.typ
  io.dmem.req.bits.fcn := m.io.dmem.req.bits.fcn
```

上面的代码中实例化了一个控制器 m，计算阵列 t。仿照 Sodor 中原来对 dpath 和 cpath 接线方式将控制器与数据通路，控制器与计算阵列对应的端口相连，并将控制器与 dmem 相连。

控制器的接口设计：

```
class MTodpathio(implicit val conf: SodorCoreParams) extends Bundle()
{
   val csignal = Input(UInt(2.W)) //给 TPU 的指令控制
   val addr1 = Input(UInt(32.W)) //A 或 B 矩阵读取地址
   val addr2 = Input(UInt(32.W)) //结果 C 矩阵写回地址
   val stall  = Output(Bool())//stall 控制，在 dpath 中与其他逻辑共同构成流水线启停的控制
}

class MpathIo(implicit val conf: SodorCoreParams) extends Bundle()
{
   val dpathio = new MTodpathio() //与 datapath 相连
   val tpathio= Flipped(new Tiletoctl()) //与 tile 相连
   val dmem = new MemPortIo(conf.xprlen) //与 dmem 相连
}
```

从上面的结构图中也能看出，控制器需要分别与 data path、dmem、Tile 相连，故在 MPathIo 模块中需要将这三部分包括进去。其中与 tile 相连的接口使用了将 tile 里面定义的 Tiletoctl 接口翻转的方式实现。与 dpath 相连的接口

包含 4 个参数，其中 addr1 为矩阵 A 或 B 的读取地址，addr2 为矩阵 C 的写回地址，csignal 用于接收 dpath 对控制器的指令信号。Stall 在 dpath 中与其他逻辑共同构成流水线启停的控制。

Buffer 寄存器与连线设计：

```
class TPUcontroler(implicit val conf: SodorCoreParams) extends Module
{
   val io= IO(new MpathIo())  //实例化接口对象
   io:=DontCare
   val tempBuffer = Reg(UInt(32.W)) //暂存 32 位内存读取结果
   val Bbuffer = Reg(Vec(16,UInt(8.W))) //暂存 B 矩阵从内存中取回的结果并直接与每个 PE 的 b 寄存器相连
   val Abuffer = Reg(Vec(4,UInt(8.W))) //暂存 A 矩阵从内存中取回的结果，按照顺序向 Tile 中逐步输入
   val Cbuffer = Reg(Vec(16,UInt(16.W))) //暂存从 Tile 中取回的计算结果，并将写回 dmem
   io.tpathio.weight_in := Bbuffer //直接与 Tile 中的输入接口相连
   io.tpathio.din := Abuffer //直接与 Tile 中的输入接口相连，按照时序输入
```

上面是控制器 class 的开始部分。定义了用于存放内存取数的缓冲寄存器，其中 Bbuffer 直接与 PE 的 b 寄存器向量，Abuffer 暂存 A 矩阵从内存中取回的结果，按照顺序向 Tile 中逐步输入，Cbuffer 暂存从 Tile 中取回的计算结果，并将写回 dmem。

定时器设计：

```
   val clken1 = Wire(Bool()) //控制指令 1 的计时器启停
   val clken2 = Wire(Bool()) //控制指令 2 的计时器启停
   val(a1,b1)=Counter(clken1,3) //控制矩阵预加载的计数器
   val(a2,b2)=Counter(clken2,16) //控制矩阵乘加运算的计数器
   clken1:=io.dmem.resp.valid&&io.dpathio.csignal===1.U //计数器的使能逻辑：当 dmem 响应数据可用、控制器的使能信号正确
   clken2:=io.dmem.resp.valid&&io.dpathio.csignal===2.U
   //生成辅助 stall 信号：当阵列处于工作模式、时钟使能并且计数未满时给 dpath 传一个 stall 信号用于停止后续指令的执行，等待当前计算完成
   io.dpathio.stall := (io.dpathio.csignal=/=0.U)&&((clken1 && a1<=3.U)||(clken2 && a2<=6.U))
   when (a1===3.U) //确保计数器只计数一轮
   {
      clken1:=false.B
   }
   when (a2===16.U)
```

```
  {
     clken2:=false.B
  }
```

定时器用于确定多个时钟周期中控制器的指令流程。由于控制器需要完成矩阵预存，矩阵乘法计算两个功能，故需要两个寄存器分别完成对应的工作。Clken1 和 clken2 分别用于控制定时器 1 和定时器 2 的启停，并且加入条件使定时器只计一次数。

预存矩阵时序设计：

```
.elsewhen (io.dpathio.csignal === 1.U){ //预加载时序设计
      //按照行的顺序每次读入 32bits 的数据，分成 4 个 8bit 整型分别送至 tile 里面对应的 PE 中
      io.dmem.req.bits.addr := io.dpathio.addr1+a1*4.U //根据时序计算出当前行的地址
      tempBuffer := io.dmem.resp.bits.data //将取出的 32bit 整型存放至缓冲区
      Bbuffer(a1*4.U+0.U) := tempBuffer(7,0) //低 8 位存放矩阵中某一行的第一列
      Bbuffer(a1*4.U+1.U) := tempBuffer(15,8)
      Bbuffer(a1*4.U+2.U) := tempBuffer(23,15)
      Bbuffer(a1*4.U+3.U) := tempBuffer(31,24)
  }
```

上面的代码中按照行的顺序每次读入 32bits 的数据，分成 4 个 8bit 整型分别送至 tile 里面对应的 PE 中。

矩阵乘法时序设计：

```
.otherwise{ //矩阵乘法计算时序设计
      when(a2<=3.U){
        // io.dmem.req.bits.fcn:=M_XRD
        // io.dmem.req.bits.typ := MT_WU 写到指令的 MEM 类型里，在 cpath 里实现内存模式的控制
         io.dmem.req.bits.addr := io.dpathio.addr1+a2*4.U
         tempBuffer := io.dmem.resp.bits.data
         Abuffer(0) := tempBuffer(7,0)
         Abuffer(1) := tempBuffer(15,8)
         Abuffer(2) := tempBuffer(23,15)
         Abuffer(3) := tempBuffer(31,24)
      }
      //3-9 个周期用于将 Tile 中送出的有效结果存入缓存区中，具体时序参见脉动阵列设计部分
      when(a2===2.U) //clk 3
```

```
      {
         Cbuffer(0) := io.tpathio.mac_out(0);
      }
      when(a2 === 3.U) //clk 4
      {
         Cbuffer(0+4*1) := io.tpathio.mac_out(0);
         Cbuffer(1+4*0) := io.tpathio.mac_out(1);
      }
      when(a2 === 4.U) //clk 5
      {
         Cbuffer(0+4*2) := io.tpathio.mac_out(0);
         Cbuffer(1+4*1) := io.tpathio.mac_out(1);
         Cbuffer(2+4*0) := io.tpathio.mac_out(2);
      }
      when(a2 === 5.U) //clk 6
      {
         Cbuffer(0+4*3) := io.tpathio.mac_out(0);
         Cbuffer(1+4*2) := io.tpathio.mac_out(1);
         Cbuffer(2+4*1) := io.tpathio.mac_out(2);
         Cbuffer(3+4*0) := io.tpathio.mac_out(3);
      }
      when(a2 === 6.U) //clk 7
      {
         Cbuffer(1+4*3) := io.tpathio.mac_out(1);
         Cbuffer(2+4*2) := io.tpathio.mac_out(2);
         Cbuffer(3+4*1) := io.tpathio.mac_out(3);
      }
      when(a2 === 7.U) //clk 8
      {
         Cbuffer(2+4*3) := io.tpathio.mac_out(2);
         Cbuffer(3+4*2) := io.tpathio.mac_out(3);
      }
      when(a2 === 8.U) //clk 9
      {
         Cbuffer(3+4*3) := io.tpathio.mac_out(3);
      }
      when(a2 > 8.U) //在剩下的时序中将存在缓冲区中的计算结果存回内存中
      {
         io.dmem.req.bits.fcn := M_XWR
         io.dmem.req.bits.typ := MT_WU
         io.dmem.req.bits.addr := io.dpathio.addr2+(a2-9.U)*4.U
         io.dmem.req.bits.data := Cat(Cbuffer((a2-9.U)*2.U),Cbuffer((a2-
9.U)*2.U+1.U))
      }
```

矩阵乘法的执行周期数较长。Tile 是流水线设计，将 A 矩阵按照顺序输入之后阵列的末端即可按一定顺序流出运算结果，因此只需要按照时序图在对应的周期里面从阵列的末端取回 C 矩阵对应的数字并放回缓存区即可。在剩下的周期中将缓存区里按行对齐的数据写回 dmem。

6. 测试与调试

由于时间原因以及开发环境使用不熟练，矩阵乘法加速器我们没有完成最终的测试，但是在测试中排查出了较多的问题，下面是我们找到的一个关键问题。

Sodor 里面定义的内存只支持两个接口的同时访问，data path 和 control path 已经分别使用了这两个接口，故我们在 core 里面直接将控制器与内存相连是不正确的，会导致 datapath 无法正确执行内存写入，从而导致测试超时。我们通过查看 output 证实了我们的想法：

```
3799591    pc=[00010018] W[r 0=02000004][0] Op1=[r12][02000004] Op2=[r13][00000000] inst=[00d62023] S   sw      a3, 0(a2)
3799592    pc=[00010018] W[r 0=02000004][0] Op1=[r12][02000004] Op2=[r13][00000000] inst=[00d62023] S   sw      a3, 0(a2)
3799593    pc=[00010018] W[r 0=02000004][0] Op1=[r12][02000004] Op2=[r13][00000000] inst=[00d62023] S   sw      a3, 0(a2)
3799594    pc=[00010018] W[r 0=02000004][0] Op1=[r12][02000004] Op2=[r13][00000000] inst=[00d62023] S   sw      a3, 0(a2)
3799595    pc=[00010018] W[r 0=02000004][0] Op1=[r12][02000004] Op2=[r13][00000000] inst=[00d62023] S   sw      a3, 0(a2)
3799596    pc=[00010018] W[r 0=02000004][0] Op1=[r12][02000004] Op2=[r13][00000000] inst=[00d62023] S   sw      a3, 0(a2)
3799597    pc=[00010018] W[r 0=02000004][0] Op1=[r12][02000004] Op2=[r13][00000000] inst=[00d62023] S   sw      a3, 0(a2)
3799598    pc=[00010018] W[r 0=02000004][0] Op1=[r12][02000004] Op2=[r13][00000000] inst=[00d62023] S   sw      a3, 0(a2)
3799599    pc=[00010018] W[r 0=02000004][0] Op1=[r12][02000004] Op2=[r13][00000000] inst=[00d62023] S   sw      a3, 0(a2)
3799600    pc=[00010018] W[r 0=02000004][0] Op1=[r12][02000004] Op2=[r13][00000000] inst=[00d62023] S   sw      a3, 0(a2)
3799601    pc=[00010018] W[r 0=02000004][0] Op1=[r12][02000004] Op2=[r13][00000000] inst=[00d62023] S   sw      a3, 0(a2)
3799602    pc=[00010018] W[r 0=02000004][0] Op1=[r12][02000004] Op2=[r13][00000000] inst=[00d62023] S   sw      a3, 0(a2)
3799603    pc=[00010018] W[r 0=02000004][0] Op1=[r12][02000004] Op2=[r13][00000000] inst=[00d62023] S   sw      a3, 0(a2)
3799604    pc=[00010018] W[r 0=02000004][0] Op1=[r12][02000004] Op2=[r13][00000000] inst=[00d62023] S   sw      a3, 0(a2)
3799605    pc=[00010018] W[r 0=02000004][0] Op1=[r12][02000004] Op2=[r13][00000000] inst=[00d62023] S   sw      a3, 0(a2)
3799606    pc=[00010018] W[r 0=02000004][0] Op1=[r12][02000004] Op2=[r13][00000000] inst=[00d62023] S   sw      a3, 0(a2)
3799607    pc=[00010018] W[r 0=02000004][0] Op1=[r12][02000004] Op2=[r13][00000000] inst=[00d62023] S   sw      a3, 0(a2)
3799608    pc=[00010018] W[r 0=02000004][0] Op1=[r12][02000004] Op2=[r13][00000000] inst=[00d62023] S   sw      a3, 0(a2)
3799609    pc=[00010018] W[r 0=02000004][0] Op1=[r12][02000004] Op2=[r13][00000000] inst=[00d62023] S   sw      a3, 0(a2)
3799610    pc=[00010018] W[r 0=02000004][0] Op1=[r12][02000004] Op2=[r13][00000000] inst=[00d62023] S   sw      a3, 0(a2)
3799611    pc=[00010018] W[r 0=02000004][0] Op1=[r12][02000004] Op2=[r13][00000000] inst=[00d62023] S   sw      a3, 0(a2)
```

图 22 错误信息

上面的 log 中 pc 值没有增加，卡在了一条 sw 指令处，这正是内存接口冲突造成的。我们的解决办法是取消 Memory 到控制器的连接，将控制器内嵌到 dpath 中，借用 dpath 与 memory 的接口进行内存访问。采用这种方法需要将控制器内嵌在 datapath 当中。

四、总结与收获

本次设计中我们认识到前期思路规划的重要性，思路越细致越好。在制定思路之前应当做好资料查找与阅读，熟悉一整套 chipyard 工具链，了解其不同模块对应的功能才能找到最好最便捷达到目的的思路。在制定思路的过程中，分模

块设计，规定模块间的接口方便分工合作。数字电路设计对信号的同步把控要精准，各种不同信号的延时需要在设计的时候就想好，这样设计出的电路才能达到想要的效果。

在设计基本的 Sodor 矩阵乘法器时，由于前期准备比较充分，实现思路也比较清晰，可以较好地按照计划与设想一步一步完成，因此最终达成了比较满意的结果，可以很好地执行矩阵乘法运算。可以说，在这一思路上我们完成了预定的任务。

在设计脉动阵列的时候，由于我们工具选择错误，debug 过程显示不出精确的 error 信息导致我们浪费了很多时间。最后使用 chipyard 进行完整的编译终于发现了问题所在，包括部分输入输出口没有没有初始化，逻辑环等，这些问题在日后数字电路设计的过程中也需要注意。

这次大作业我们切实学习到了如何使用陌生的工具和硬件描述语言进行简单的数字电路设计，收获颇丰。

五、分工情况

曹逸芃（贡献占比 40%）：

1. Sodor_1stage 和 Sodor_5stage 全部的代码修改与编写；
2. MUL 等指令的引入与测试；
3. 测试代码的编写，以及 Sodor 矩阵乘法器的测试；
4. 脉动阵列相关 RISC-V 指令 TPUP 和 TPUA 的设计与引入；
5. 脉动阵列代码的辅助编写与功能调试。

陶冠辰（贡献占比 30%）：

1. 编写脉动阵列的 PE、Tile 等模块；
2. 脉动阵列的后期调试；
3. Chisel 语言的前期验证测试。

张铁沄（贡献占比 30%）：

1. 仿真环境的搭建与测试；
2. 编写脉动阵列的控制器模块；
3. 脉动阵列的后期测试；