深 圳 大 学 实 验 报 告

课 程 名 称：计算机系统(3)

实验项目名称：RISCV-mini 处理器运行观察

学　　院：计算机与软件学院

专　　业：计算机与软件学院所有专业

指 导 教 师：罗秋明

报告人：林浩晟　学号：2022280310　班级：01

实 验 时 间：2024 年 11 月 15 日星期五

实验报告提交时间：2024 年 11 月 22 日星期五

教务处制

一、 实验目标：

了解 RISC-V mini 处理器架构，了解其 Chisel 设计，观察指令执行。

二、实验内容

1) 编写 risc-v 汇编程序，实现两数减法操作，记录程序编译和仿真过程作为实验说明。
2) 根据给出的 C 程序，完成编译仿真过程，并通过仿真波形图，结合指令执行过程中程序计数器、寄存器文件，相关控制信号等的变化解释 factorial(10)函数调用时的参数 10 传入寄存器的过程。

三、实验环境

硬件：桌面 PC
软件：Chisel 开发环境

四、实验步骤及说明

按下列文档的操作说明，复现实验步骤记录实验过程。

一、实验环境的搭建

RiscV mini 是一款由 chisel 语言编写的三级流水线处理器。chisel 语言的编译可以生成处理器的硬件描述语言代码 Verilog，借助 Verilog 代码与 EDA 软件(Vivado)可以将处理器设计下载到 FPGA 进行原型验证或后续 ASIC 流片。

RiscV mini 处理器运行 RISCV32I 指令集，通过 RiscV 交叉工具链提供的编译器可将汇编程序编译成机器可执行程序，通过 EDA 软件(Vivado)将程序初始化到与处理器相连接的 RAM 中以运行。下面将演示 RiscV mini 软件开发环境的配置，主要涉及 risc-v工具链部分。

首先从 GitHub 上 clone risc-v mini 项目工程并执行make 编译，这里默认已经安装过 sbt 和 make 工具。在这个过程当中会自动下载需要的依赖并生成 risc-v mini 的 Verilog文件。这个过程根据网络情况需要等待一定的时间。

```
1. git clone https://github.com/ucb-bar/riscv-mini.git
2. cd riscv-mini
3. make
```

```
user@ubuntu:~/Desktop/exp6$ git clone https://github.com/ucb-bar/riscv-m
ini.git
Cloning into 'riscv-mini'...
remote: Enumerating objects: 2402, done.
remote: Counting objects: 100% (2402/2402), done.
remote: Compressing objects: 100% (834/834), done.
remote: Total 2402 (delta 1349), reused 2268 (delta 1288), pack-reused 0
 (from 0)
Receiving objects: 100% (2402/2402), 1.40 MiB | 1.54 MiB/s, done.
Resolving deltas: 100% (1349/1349), done.
user@ubuntu:~/Desktop/exp6$ cd riscv-mini
user@ubuntu:~/Desktop/exp6/riscv-mini$ make
sbt -ivy /home/user/Desktop/exp6/riscv-mini/.ivy2 "run --target-dir=/hom
e/user/Desktop/exp6/riscv-mini/generated-src --dump-fir"
[info] welcome to sbt 1.8.2 (Ubuntu Java 11.0.24)
[info] loading settings for project riscv-mini-build from plugins.sbt ..
.
[info] loading project definition from /home/user/Desktop/exp6/riscv-min
i/project
[info] loading settings for project root from build.sbt ...
[info] set current project to riscv-mini (in build file:/home/user/Deskt
op/exp6/riscv-mini/)
[info] compiling 14 Scala sources to /home/user/Desktop/exp6/riscv-mini/
target/scala-2.13/classes ...
[info] running mini.Main --target-dir=/home/user/Desktop/exp6/riscv-mini
/generated-src --dump-fir
[success] Total time: 16 s, completed Nov 21, 2024, 11:29:43 PM
```

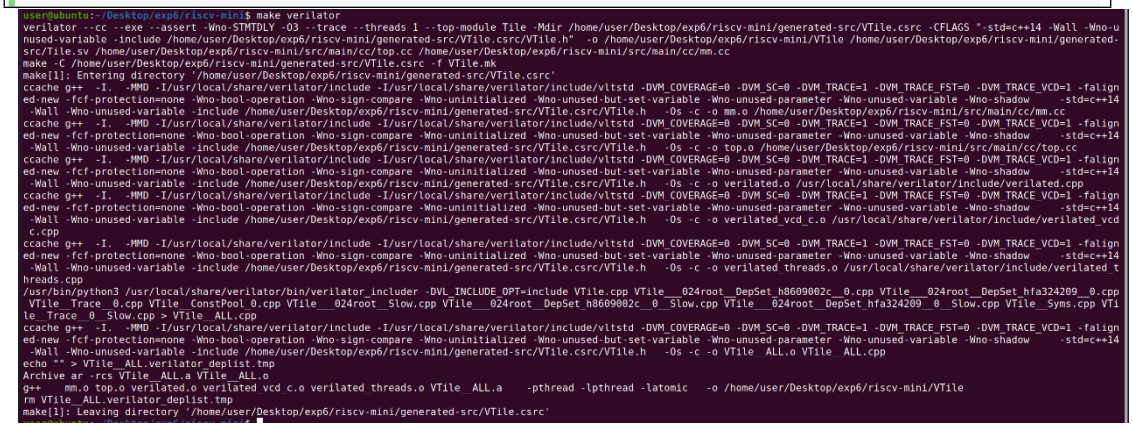
进入 `generated-src` 路径可以看到生成的处理器 verilog 代码文件 Tile.v。

```
1.    cd generated-src/
2.    ls
```

```
user@ubuntu:~/Desktop/exp6/riscv-mini$ cd generated-src
user@ubuntu:~/Desktop/exp6/riscv-mini/generated-src$ ls
Tile.fir  Tile.sv
```

编译完处理器，还要编译处理器的仿真器（用于后续在 riscv mini 上的程序仿真），首先安装 verilator 和 g++，然后在项目根目录执行编译命令：

```
1.    sudo apt install verilator
2.    sudo apt install g++
3.    make verilator
```



编译完成后生成了仿真器文件，即 VTile 文件。

```
user@ubuntu:~/Desktop/exp6/riscv-mini$ ls
build-riscv-tools.sh  build.sbt  custom-bmark  diagram.pdf  diagram.png  generat
ed-src  LICENSE  Makefile  project  README.md  src  target  tests  VTile
```

接下来，若想要在 riscv-mini 上运行自己编写的程序，需要安装特权指令集 1.7 版本的GNU工具链。Riscv32-unkown-elf-gcc 是一款交叉编译器，可以在非 RISCV 平台上，将 C 语言或 RISCV32I 指令集的汇编语言编译成 RISCV32I 机器可以执行的可装入字节文件。这里直接提供编译好的二进制程序以及需要的相关文件，只需要将压缩包解压并且添加环境变量即可使用。使用如下命令将 riscv-tools-priv1.7.tar.gz 解压。

```
1.    tar -zxvf riscv-tools-priv1.7.tar.gz
```

在系统环境变量文件~/.bashrc 中添加刚才解压的文件夹路径：

```
1.    sudo apt install vim
2.    sudo vim ~/.bashrc
3.    export PATH=$PATH:/你的路径
   /riscv-tools-priv1.7/riscv-tools-mini/bin
4.    export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/你的路径
   /riscv-tools-priv1.7/riscv-tools-mini/lib
```

```
119 export PATH=$PATH:/home/user/Desktop/riscv-tools-priv1.7/riscv-tools-mini/bin
120 export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/user/Desktop/riscv-tools-priv1.7/riscv-tools-
    mini/lib
```

然后更新系统变量：

```
1.    source ~/.bashrc
```

软链接 c 语言动态库

```
1.    sudo ln -s /usr/lib/x86_64-linux-gnu/libmpfr.so.6 /usr/lib/x86_64
   -linux-gnu/libmpfr.so.4
```

查看编译器是否安装成功：

```
1.    riscv32-unknown-elf-gcc --version
```

```
user@ubuntu:~/Desktop/exp6/riscv-mini$ riscv32-unknown-elf-gcc --version
riscv32-unknown-elf-gcc (GCC) 5.3.0
Copyright (C) 2015 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

若输出以上结果则说明实验环境搭建完成。后续我们就可以通过这个过程在risc-v mini上编译运行自己写的汇编程序或C程序了。

二、编译运行程序

1. 汇编程序

完成了实验环境的搭建后，我们开始尝试自己编写程序并编译，首先以编写一个汇编程序为例，之后在RISC-V MINI上运行，观察并分析生成的波形文件，深入理解RISC-V MINI处理器的执行过程。

编写一个简单汇编程序如下，其功能部分仅有三条指令，即：将立即数 1 和 2 写入到 x6 与 x7 寄存器中，随后将两个寄存器中的值相加存入到 x5 寄存器当中。此外对于 exit 中的循环，不停将 1 写入到 csr 寄存器 mtohost 中，这段程序的作用是通知仿真器结束仿真（作用类似于 X86 汇编中的 HLT 指令）。

```
1.              .text # Define beginning of text section
2.              .global _start # Define entry _start
3.      _start:
4.              li x6, 1 # x6 = 1
5.              li x7, 2 # x7 = 2
6.              add x5, x6, x7 # x5 = x6 + x7
7.      exit:
8.              csrw mtohost, 1
9.              j exit
10.             .end # End of file
```

汇编程序编写完成后通过以下命令进行编译，注意这里需要添加编译参数-nostdlib 表示不链接标准库，-Ttext=0x200 表示 text 节需要放置到 PC=0x200 的位置上(risc-v mini pc 的起始位置)。

```
1.      riscv32-unknown-elf-gcc -nostdlib -Ttext=0x200 -o test test.s
```

编译完成后我们便可得到 elf 文件，通过 riscv32-unknown-elf-readelf 我们可以查看其具体信息。如下通过查看 elf 头可以看到该 elf 文件的系统架构为 RISC-V，入口地址为 0x200。

```
1.      riscv32-unknown-elf-readelf -h test
```

```
user@ubuntu:~/Desktop/exp6/riscv-mini$ riscv32-unknown-elf-gcc -nostdlib -Ttext=
0x200 -o test test.s
user@ubuntu:~/Desktop/exp6/riscv-mini$ riscv32-unknown-elf-readelf -h test
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           RISC-V
  Version:                           0x1
  Entry point address:               0x200
  Start of program headers:          52 (bytes into file)
  Start of section headers:          860 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         1
  Size of section headers:           40 (bytes)
  Number of section headers:         5
  Section header string table index: 2
```

通过 riscv32-unknown-elf-objdump 命令还可以对 elf 文件进行反汇编来查看程序与指令地址情况。如下可以看到该程序为简单的 1+2 程序，执行完成后进入一个死循环，并且可以看到程序的启始_start 被放置到了 0x200 位置。

```
1.    riscv32-unknown-elf-objdump -S test
```

```
user@ubuntu:~/Desktop/exp6/riscv-mini$ riscv32-unknown-elf-objdump -S test

test:     file format elf32-littleriscv


Disassembly of section .text:

00000200 <_start>:
 200:   00100313                li      t1,1
 204:   00200393                li      t2,2
 208:   007302b3                add     t0,t1,t2

0000020c <exit>:
 20c:   7800d073                csrwi   mtohost,1
 210:   ffdff06f                j       20c <exit>
```

之后若要对编译得到的 elf 文件在 RISC-V MINI 上进行仿真，需要将 elf 文件转化为特定格式的 hex 文件。这里通过 elf2hex 工具来进行，将 elf 文件转化为宽度为 16 字节的 hex 文件。elf2hex 的用法为 elf2hex <width> <depth> <elf_file>，将输出的内容重定向到.hex 文件文件当中。

最后通过前面编译得到的仿真器 VTile 进行仿真，命令如下所示，仿真过程会生成波形图文件，通过 gtkwave 打开波形图文件观察波形便可了解指令在处理器中的执行过程。

```
1.    elf2hex 16 4096 test > test.hex
2.    ./VTile ./test.hex test.vcd
```

```
user@ubuntu:~/Desktop/exp6/riscv-mini$ elf2hex 16 4096 test > test.hex
user@ubuntu:~/Desktop/exp6/riscv-mini$ ./VTile ./test.hex test.vcd
Enabling waves...
Starting simulation!
Simulation completed at time 56 (cycle 5)
Finishing simulation!
user@ubuntu:~/Desktop/exp6/riscv-mini$ ls
build-riscv-tools.sh  diagram.pdf    LICENSE   README.md  test       tests
build.sbt             diagram.png    Makefile  src        test.hex   test.vcd
custom-bmark          generated-src  project   target     test.s     VTile
```

2. C 程序

同样的，我们也可以在 riscv-mini 上运行我们编写的 C 程序。由于使用 RISC-V MINI 提供的 makefile 进行编译会链接部分的库函数，不便于我们接下来的观察，因此在这个过程当中我们自己来完成这个编译的过程。在这里使用如下的 C 代码作为例，该代码的功能是使用递归的方式计算一个数的阶乘。程序的最后调用 exit()函数，循环向 mtohost 写入 1，作用与上面的汇编代码一样，同样是为了在仿真过程当中通知仿真器结束仿真。

```c
int factorial(int num);
void exit();
void main(){
 int ans = factorial(10);
 exit();
}
int factorial(int num){
 if(num <= 1) return 1;
 else return num * factorial(num - 1);
}
void exit(){
 while(1)
        __asm__ __volatile__("csrw mtohost, 1");
}
```

使用 riscv32-unknown-elf-gcc 编译我们编写的 C 程序，在这个过程中需要添加参数让其不链接标准库，并且指定 text 节的地址为 0x200。除此之外由于 riscv-mini 的 RV32I 指令集没有乘法指令，还需要添加-march=RV32I，编译器会自动将乘法指令替换为以移位和加法运算完成的乘法运算，使其能够在 RV32I 下运行。编译完成以后即可得到 elf 文件，在编译过程中会得到一个 warning，这里不必理会。

```
riscv32-unknown-elf-gcc -nostdlib -Ttext=0x200 -march=RV32I -o fa
```

```
ctorial.elf factorial.c
```

```
user@ubuntu:~/Desktop/exp6/riscv-mini$ riscv32-unknown-elf-gcc -nostdlib -Ttext=
0x200 -march=RV32I -o factorial.elf factorial.c
factorial.c: In function 'exit':
factorial.c:13:6: warning: number of arguments doesn't match built-in prototype
 void exit(){
      ^
/home/user/Desktop/riscv-tools-priv1.7/riscv-tools-mini/bin/../lib/gcc/riscv32-u
nknown-elf/5.3.0/../../../../riscv32-unknown-elf/bin/ld: warning: cannot find en
try symbol _start; defaulting to 0000000000000200
user@ubuntu:~/Desktop/exp6/riscv-mini$ ls
 build-riscv-tools.sh   factorial.elf   src          test.vcd
 build.sbt              generated-src   target      'Untitled Document 1'
 custom-bmark           LICENSE         test         VTile
 diagram.pdf            Makefile        test.hex
 diagram.png            project         test.s
 factorial.c            README.md       tests
```

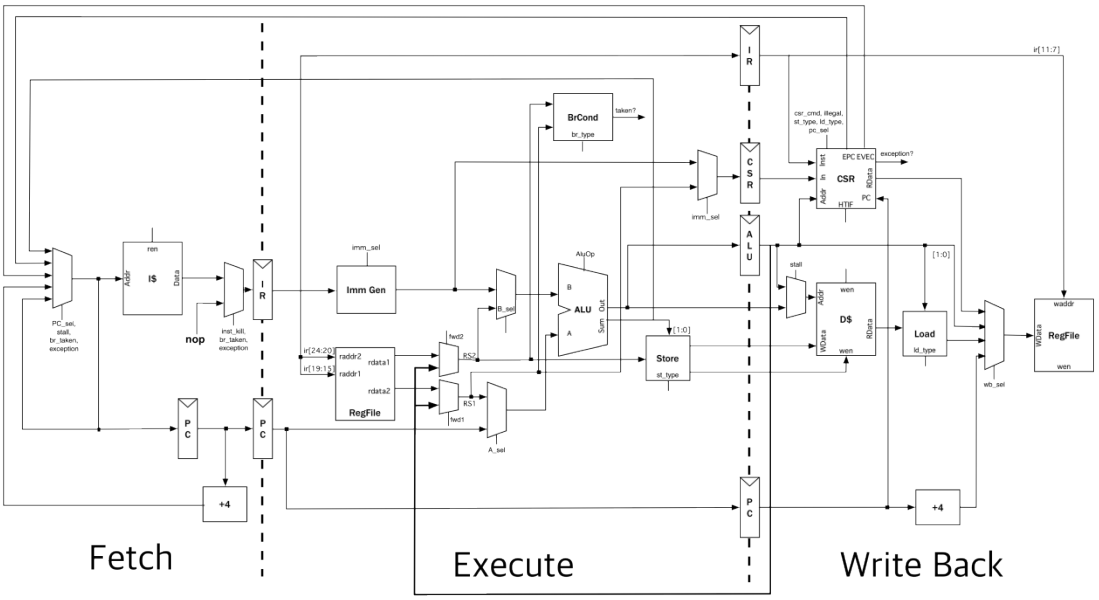
接下来的过程与之前编译运行汇编代码一致，可以通过 riscv32-unknown-elf-objdump 对 elf 文件进行反汇编观察每条指令代码的地址，然后通过 elf2hex 将 elf 文件转化为 hex 文件并使用 VTile 进行仿真就能得到.vcd 波形文件。

```
user@ubuntu:~/Desktop/exp6/riscv-mini$ elf2hex 16 4096 factorial.elf > factorial
.hex
user@ubuntu:~/Desktop/exp6/riscv-mini$ ls
 build-riscv-tools.sh   factorial.elf   README.md   tests
 build.sbt              factorial.hex   src         test.vcd
 custom-bmark           generated-src   target     'Untitled Document 1'
 diagram.pdf            LICENSE         test        VTile
 diagram.png            Makefile        test.hex
 factorial.c            project         test.s
user@ubuntu:~/Desktop/exp6/riscv-mini$ ./VTile ./factorial.hex factorial.vcd
Enabling waves...
Starting simulation!
Simulation completed at time 1376 (cycle 137)
Finishing simulation!
user@ubuntu:~/Desktop/exp6/riscv-mini$ ls
 build-riscv-tools.sh   factorial.elf   project     test.s
 build.sbt              factorial.hex   README.md   tests
 custom-bmark           factorial.vcd   src         test.vcd
 diagram.pdf            generated-src   target     'Untitled Document 1'
 diagram.png            LICENSE         test        VTile
 factorial.c            Makefile        test.hex
```

三、通过波形图观察指令的执行过程

在使用 Verilator 仿真的过程当中可以生成波形图，通过波形图可以观察 CPU 中各个引脚或者寄存器数值的变化情况，从而了解指令在 CPU 中的执行过程。这里以我们上面编写的汇编程序仿真得到的波形文件为例，观察处理器执行过程中各寄存器的具体状况。

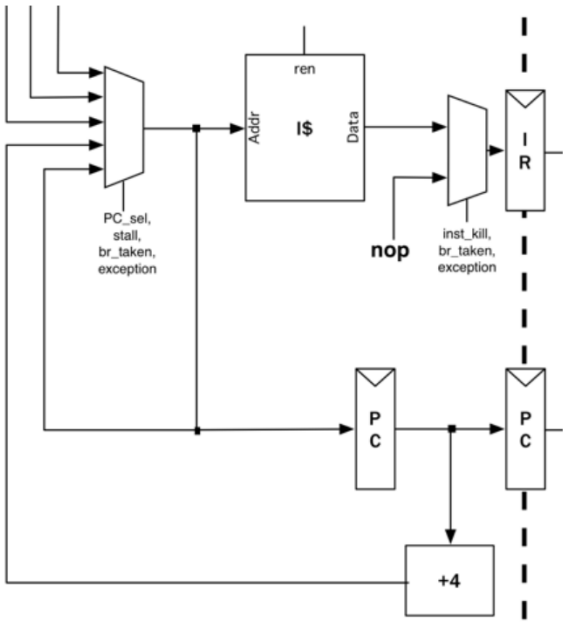
riscv-mini 具有三级流水线，其数据通路如下图所示(注意图中 RegFile 中 Data1 与 Data2 的标注存在错误,位置应该互换)。具体流水线实现代码位于 riscv-mini/src/main/scala/Datapath.scala 中。



将上面汇编程序对应的 elf 文件使用 objdump 进行反汇编得到每条指令对应的地址信息，如下所示。

```
test:     file format elf32-littleriscv

Disassembly of section .text:

00000200 <_start>:
 200: 00100313           li t1,1
 204: 00200393           li t2,2
 208: 007302b3           add t0,t1,t2

0000020c <exit>:
 20c: 7800d073           csrwi mtohost,1
 210: ffdff06f           j 20c <exit>
```

首先观察程序执行过程中取指阶段的内容。在这个阶段当中需要根据 PC 值取出将要执行的指令，该阶段的数据通路如下图所示。首先通过一个多路选择器根据情况选择将要执行的下一条指令地址，随后访问地址 Cache 获取对应地址中的指令，若流水线正常执行未发生分支与异常等情况便可将读取到的指令放入到 Fetch/Execute 流水线寄存器中给下一个阶段使用，否则用空指令冲刷流水线产生一个空泡。

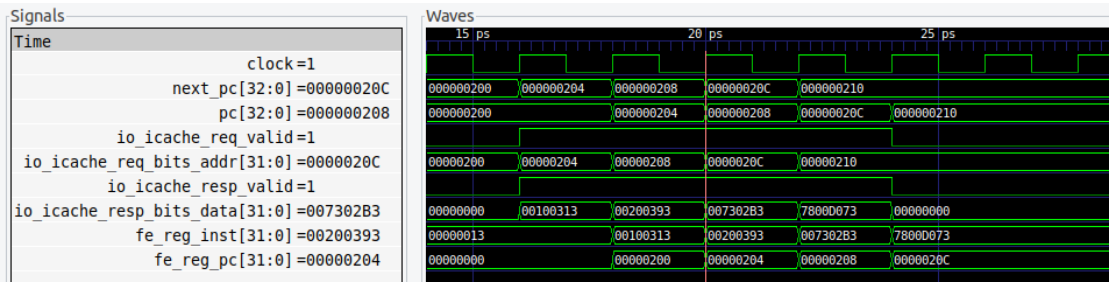


取指阶段的 Chisel 代码如下。其中 next_pc 为一个多路选择器，根据控制信号选择下一条指令的地址。当流水线开始指令时 pc 寄存器获取 next_pc 输出的地址，并从 icache 读取对应地址的指令。当流水线未停顿时便可将 pc 与读取到的指令存入到流水线寄存器 fe_reg.pc 与 fe_reg.inst 中。

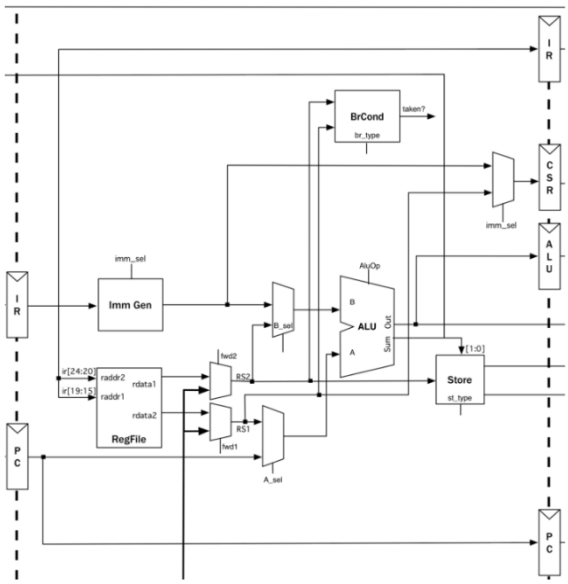
```
  /** **** Fetch *****/
  val started = RegNext(reset.asBool)
  val stall = !io.icache.resp.valid || !io.dcache.resp.valid
  val pc = RegInit(Const.PC_START.U(conf.xlen.W) - 4.U(conf.xlen.W))
  // Next Program Counter
  val next_pc = MuxCase(
    pc + 4.U,
    IndexedSeq(
      stall -> pc,
      csr.io.expt -> csr.io.evec,
      (io.ctrl.pc_sel === PC_EPC) -> csr.io.epc,
      ((io.ctrl.pc_sel === PC_ALU) || (brCond.io.taken)) -> (alu.io.sum
>> 1.U << 1.U),
      (io.ctrl.pc_sel === PC_0) -> pc
    )
  )
  val inst =
    Mux(started || io.ctrl.inst_kill || brCond.io.taken || csr.io.expt,
Instructions.NOP, io.icache.resp.bits.data)
  pc := next_pc
  io.icache.req.bits.addr := next_pc
  io.icache.req.bits.data := 0.U
  io.icache.req.bits.mask := 0.U
  io.icache.req.valid := !stall
```

```
io.icache.abort := false.B

// Pipelining
when(!stall) {
  fe_reg.pc := pc
  fe_reg.inst := inst
}
```

使用 GTKWave 打开仿真得到的.vcd 文件，对应到波形图观察一条特定指令的执行过程，这里以执行的第二条指令 li t2,2 为例，li 指令其实是一条伪指令，用于加载立即数。编译器会根据立即数的大小将其转化为一条或两条等价功能的指令来执行。在这里 li t2,2 与 addi x7，x0，2 等价。这条 li 指令位于 0x204 的内存地址上，从下面的波形图可以观察到 icache 的 req.addr 端口的输入即为 next_pc 中的数值。若 cache 命中则可以直接在下一个时钟周期中取得对应的指令，若未命中则会产生阻塞直至从内存中读取完成。从下图中可以观察到 cache 命中，读取出了指令 00200393，并在下一个周期将其写入到了 fe_reg_inst 寄存器中以供执行阶段使用。



接下来是执行阶段，在该阶段当中需要完成指令译码、准备操作数(从 regfile 中读取或产生立即数)、指令执行以及访存(访存指令)。该阶段的数通路如下图所示。首先根据需要执行的指令通过立即数生成单元以及访问寄存器文件获得执行阶段所需要使用的操作数作为 ALU 的输入。具体的输入由图中的多个多路选择器进行选择，可能为寄存器文件中读取到的数值、立即数、PC 或者由处于下一个阶段的指令前递来的数值。与此同时若为分支指令则由 BrCond 模块进行分支判断，若需要分支则下一条指令的 PC 地址为分支指令的目标地址。



执行阶段的 Chisel 代码如下。自上而下的看，首先将取指阶段得到的指令传给控制单元获得该指令的控制信号，访问寄存器文件读取需要的操作数，通过立即数生成单元获得立即数。接下来对 ALU 的输入数据做出选择，来源可能为立即数、从寄存器文件中读取到的数据或者处于 EXE/WB 流水线寄存器中的数据（前递）。同时判断该指令是否产生分支以及使用计算出的地址访问数据 cache 进行访存操作。若流水线正常执行则将该阶段产生的数据放入到 EXE/WB 流水线寄存器中供给下一个阶段使用，反之冲刷流水线产生气泡或者停顿流水线。

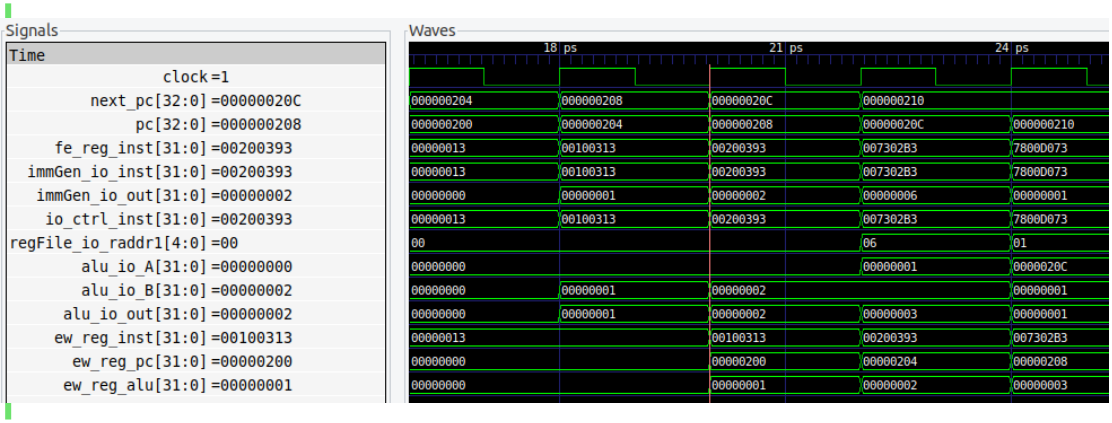
```
/** **** Execute *****/
  io.ctrl.inst := fe_reg.inst

  // regFile read
  val rd_addr = fe_reg.inst(11, 7)
  val rs1_addr = fe_reg.inst(19, 15)
  val rs2_addr = fe_reg.inst(24, 20)
  regFile.io.raddr1 := rs1_addr
  regFile.io.raddr2 := rs2_addr

  // gen immdeates
  immGen.io.inst := fe_reg.inst
  immGen.io.sel := io.ctrl.imm_sel

  // bypass
  val wb_rd_addr = ew_reg.inst(11, 7)
  val rs1hazard = wb_en && rs1_addr.orR && (rs1_addr === wb_rd_addr)
  val rs2hazard = wb_en && rs2_addr.orR && (rs2_addr === wb_rd_addr)
  val rs1 = Mux(wb_sel === WB_ALU && rs1hazard, ew_reg.alu, regFile.io.
rdata1)
  val rs2 = Mux(wb_sel === WB_ALU && rs2hazard, ew_reg.alu, regFile.io.
rdata2)
```

```
  // ALU operations
  alu.io.A := Mux(io.ctrl.A_sel === A_RS1, rs1, fe_reg.pc)
  alu.io.B := Mux(io.ctrl.B_sel === B_RS2, rs2, immGen.io.out)
  alu.io.alu_op := io.ctrl.alu_op

  // Branch condition calc
  brCond.io.rs1 := rs1
  brCond.io.rs2 := rs2
  brCond.io.br_type := io.ctrl.br_type

  // D$ access
  val daddr = Mux(stall, ew_reg.alu, alu.io.sum) >> 2.U << 2.U
  val woffset = (alu.io.sum(1) << 4.U).asUInt | (alu.io.sum(0) << 3.U).
asUInt
  io.dcache.req.valid := !stall && (io.ctrl.st_type.orR || io.ctrl.ld_t
ype.orR)
  io.dcache.req.bits.addr := daddr
  io.dcache.req.bits.data := rs2 << woffset
  io.dcache.req.bits.mask := MuxLookup(
    Mux(stall, st_type, io.ctrl.st_type),
    "b0000".U,
    Seq(ST_SW -> "b1111".U, ST_SH -> ("b11".U << alu.io.sum(1, 0)), ST_
SB -> ("b1".U << alu.io.sum(1, 0)))
  )

  // Pipelining
  when(reset.asBool || !stall && csr.io.expt) {
    st_type := 0.U
    ld_type := 0.U
    wb_en := false.B
    csr_cmd := 0.U
    illegal := false.B
    pc_check := false.B
  }.elsewhen(!stall && !csr.io.expt) {
    ew_reg.pc := fe_reg.pc
    ew_reg.inst := fe_reg.inst
    ew_reg.alu := alu.io.out
    ew_reg.csr_in := Mux(io.ctrl.imm_sel === IMM_Z, immGen.io.out, rs1)
    st_type := io.ctrl.st_type
    ld_type := io.ctrl.ld_type
    wb_sel := io.ctrl.wb_sel
    wb_en := io.ctrl.wb_en
    csr_cmd := io.ctrl.csr_cmd
```

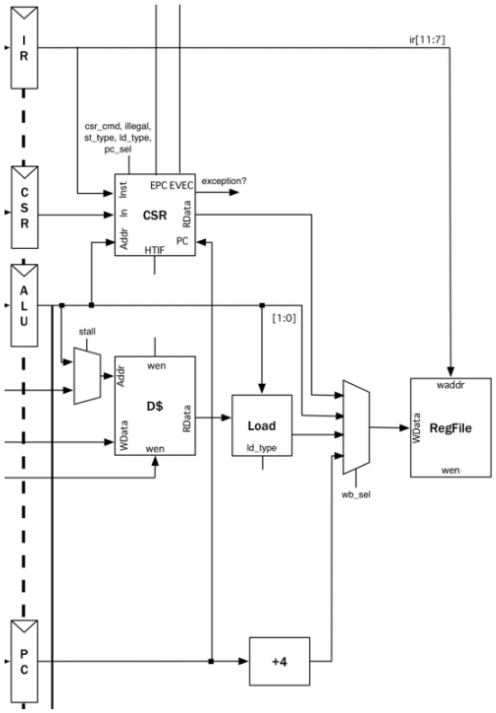
```
    illegal := io.ctrl.illegal
    pc_check := io.ctrl.pc_sel === PC_ALU
  }
```

```
/** **** Execute *****/
io.ctrl.inst := fe_reg.inst

// regFile read
val rd_addr = fe_reg.inst(11, 7)
val rs1_addr = fe_reg.inst(19, 15)
val rs2_addr = fe_reg.inst(24, 20)
regFile.io.raddr1 := rs1_addr
regFile.io.raddr2 := rs2_addr

// gen immdeates
immGen.io.inst := fe_reg.inst
immGen.io.sel := io.ctrl.imm_sel

// bypass
val wb_rd_addr = ew_reg.inst(11, 7)
val rs1hazard = wb_en && rs1_addr.orR && (rs1_addr === wb_rd_addr)
val rs2hazard = wb_en && rs2_addr.orR && (rs2_addr === wb_rd_addr)
val rs1 = Mux(wb_sel === WB_ALU && rs1hazard, ew_reg.alu, regFile.io.rdata1)
val rs2 = Mux(wb_sel === WB_ALU && rs2hazard, ew_reg.alu, regFile.io.rdata2)

// ALU operations
alu.io.A := Mux(io.ctrl.A_sel === A_RS1, rs1, fe_reg.pc)
alu.io.B := Mux(io.ctrl.B_sel === B_RS2, rs2, immGen.io.out)
alu.io.alu_op := io.ctrl.alu_op

// Branch condition calc
brCond.io.rs1 := rs1
brCond.io.rs2 := rs2
brCond.io.br_type := io.ctrl.br_type

// D$ access
val daddr = Mux(stall, ew_reg.alu, alu.io.sum) >> 2.U << 2.U
val woffset = (alu.io.sum(1) << 4.U).asUInt | (alu.io.sum(0) << 3.U).asUInt
io.dcache.req.valid := !stall && (io.ctrl.st_type.orR || io.ctrl.ld_type.orR)
io.dcache.req.bits.addr := daddr
io.dcache.req.bits.data := rs2 << woffset
io.dcache.req.bits.mask := MuxLookup(
Mux(stall, st_type, io.ctrl.st_type),
"b0000".U,
Seq(ST_SW -> "b1111".U, ST_SH -> ("b11".U << alu.io.sum(1, 0)), ST_SB -> ("b1".U << alu.io.sum(1, 0)))
)

// Pipelining
when(reset.asBool || !stall && csr.io.expt) {
        st_type := 0.U
        ld_type := 0.U
        wb_en := false.B
        csr_cmd := 0.U
        illegal := false.B
        pc_check := false.B
        }
.elsewhen(!stall && !csr.io.expt) {
        ew_reg.pc := fe_reg.pc
        ew_reg.inst := fe_reg.instS
        ew_reg.alu := alu.io.out
        ew_reg.csr_in := Mux(io.ctrl.imm_sel === IMM_Z, immGen.io.out, rs1)
        st_type := io.ctrl.st_type
        ld_type := io.ctrl.ld_type
        wb_sel := io.ctrl.wb_sel
        wb_en := io.
        csr_cmd := io.ctrl.csr_cmd
        illegal := io.ctrl.illegal
        pc_check := io.ctrl.pc_sel === PC_ALU
        }
```

接下来同样对应到波形图，对于这一条指令 addi x7,x0,2，需要根据指令产生立即数 2，从寄存器 x0 中读取数值，并使用 ALU 将立即数与读出的数值相加。从下面的波形图中可以观察到，立即数生成单元(immGen)、控制单元(ctrl)的输入均为取指阶段取出的指令 fe_reg_inst。立即数生成单元根据指令译码之后给出对应的立即数 0x02，同时寄存器文件也给出了 x0 寄存器读取的结果(x0 寄存器始终为 0)。alu 中以寄存器读取的结果以及立即数作为输入，根据控制信号将其相加得到结果 0x02，并将结果写入到流水线寄存器 ew_reg_alu 中以供写回阶段使用。



最后是写回阶段，在该阶段当中需要将指令执行的结果写回到寄存器中。根据指令类型将对应的结果写入到寄存器文件当中，若为访存指令则在该阶段获得防存的结果并提取需要的位将其写回。该阶段的数据通路如下图所示。



写回阶段的 Chisel 代码如下。在这个阶段从数据 Cache 获得读取到的数据并根据指令类型提取需要的位。同时 CSR 的访问也在该阶段进行。并且根据控制信号将需要的数据写回到寄存器文件当中。

```
// Load
  val loffset = (ew_reg.alu(1) << 4.U).asUInt | (ew_reg.alu(0) << 3.U).
asUInt
  val lshift = io.dcache.resp.bits.data >> loffset
  val load = MuxLookup(
    ld_type,
    io.dcache.resp.bits.data.zext,
    Seq(
      LD_LH -> lshift(15, 0).asSInt,
      LD_LB -> lshift(7, 0).asSInt,
```

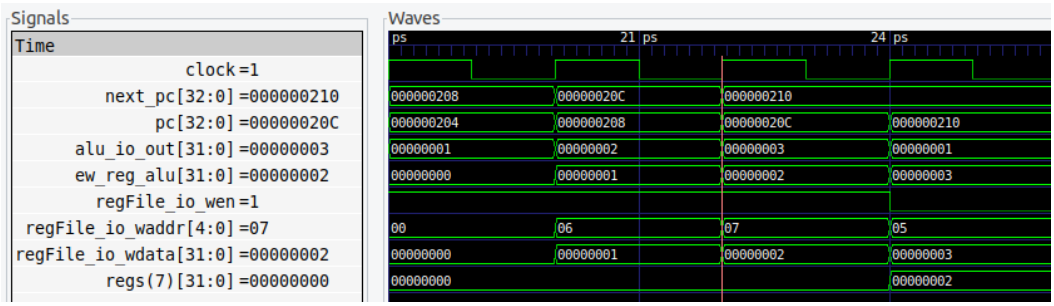
```
      LD_LHU -> lshift(15, 0).zext,
      LD_LBU -> lshift(7, 0).zext
    )
  )

  // CSR access
  csr.io.stall := stall
  csr.io.in := ew_reg.csr_in
  csr.io.cmd := csr_cmd
  csr.io.inst := ew_reg.inst
  csr.io.pc := ew_reg.pc
  csr.io.addr := ew_reg.alu
  csr.io.illegal := illegal
  csr.io.pc_check := pc_check
  csr.io.ld_type := ld_type
  csr.io.st_type := st_type
  io.host <> csr.io.host

  // Regfile Write
  val regWrite =
    MuxLookup(
      wb_sel,
      ew_reg.alu.zext,
      Seq(WB_MEM -> load, WB_PC4 -> (ew_reg.pc + 4.U).zext, WB_CSR -> c
sr.io.out.zext)
    ).asUInt

  regFile.io.wen := wb_en && !stall && !csr.io.expt
  regFile.io.waddr := wb_rd_addr
  regFile.io.wdata := regWrite

  // Abort store when there's an excpetion
  io.dcache.abort := csr.io.expt
```

```
67 // Load
68 val loffset = (ew_reg.alu(1) << 4.U).asUInt | (ew_reg.alu(0) << 3.U).asUInt
69 val lshift = io.dcache.resp.bits.data >> loffset
70 val load = MuxLookup(
71 ld_type,
72 io.dcache.resp.bits.data.zext,
73 Seq(
74 LD_LH -> lshift(15, 0).asSInt,
75 LD_LB -> lshift(7, 0).asSInt,
76 LD_LHU -> lshift(15, 0).zext,
77 LD_LBU -> lshift(7, 0).zext
78 )
79 )
```

```
81 // CSR access
82 csr.io.stall := stall
83 csr.io.in := ew_reg.csr_in
84 csr.io.cmd := csr_cmd
85 csr.io.inst := ew_reg.inst
86 csr.io.pc := ew_reg.pc
87 csr.io.addr := ew_reg.alu
88 csr.io.illegal := illegal
89 csr.io.pc_check := pc_check
90 csr.io.ld_type := ld_type
91 csr.io.st_type := st_type
92 io.host <> csr.io.host
```

```
93 // Regfile Write
94 val regWrite =
95 MuxLookup(
96 wb_sel,
97 ew_reg.alu.zext,
98 Seq(WB_MEM -> load, WB_PC4 -> (ew_reg.pc + 4.U).zext, WB_CSR -> csr.io.out.zext)
99 ).asUInt
100
101 regFile.io.wen := wb_en && !stall && !csr.io.expt
102 regFile.io.waddr := wb_rd_addr
103 regFile.io.wdata := regWrite
104
105 // Abort store when there's an excpetion
106 io.dcache.abort := csr.io.expt
```

对于 addi x7,x0,2 指令，需要将 ALU 计算出的结果 0x02 写入到 x7 寄存器当中。观察如下的波形图，可以看到在上一阶段执行完成得到 0x02 并将结果写入到流水线寄存器 ew_reg_alu 的同时，寄存器文件的写地址信号已经为 07，且写信号为高，所以下一个时钟周期发现 0x02 已经被成功写入到 x7 寄存器当中。



实验内容：解释上述 C 代码中 factorial(10)函数的参数 10 传入寄存器的过程。

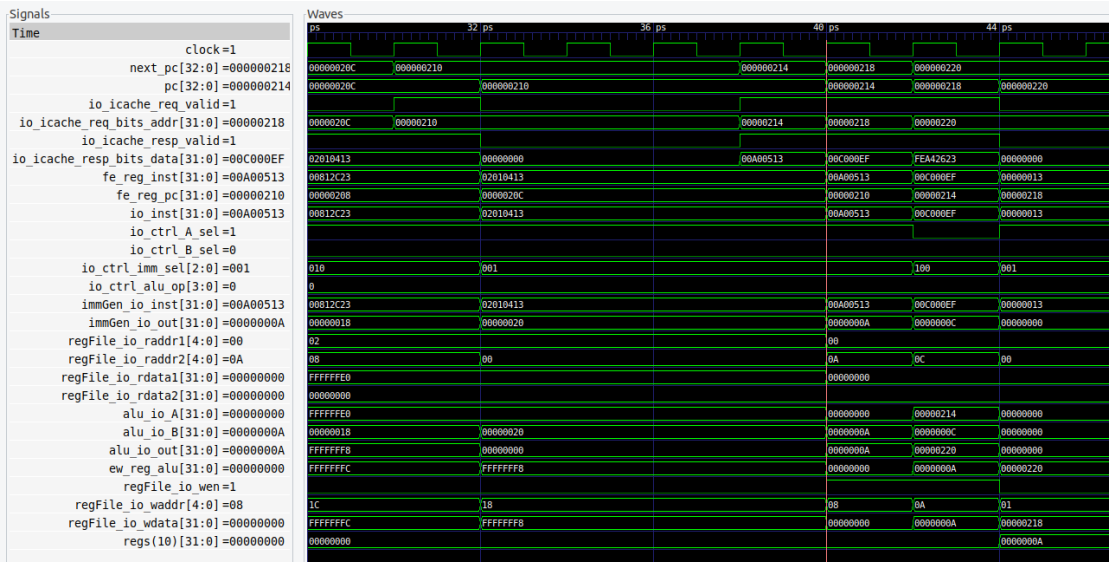
可以先将上述 C 代码编译后得到的 elf 文件反汇编得到汇编代码，查看处理器执行的指令及对应的地址：

```
sktgroup@sktgroup-virtual-machine:~/Desktop/riscv-mini/test_c$ riscv32-unknown-elf-objdump -S factorial.elf

factorial.elf:     file format elf32-littleriscv


Disassembly of section .text:

00000200 <main>:
 200:   fe010113                addi    sp,sp,-32
 204:   00112e23                sw      ra,28(sp)
 208:   00812c23                sw      s0,24(sp)
 20c:   02010413                addi    s0,sp,32
 210:   00a00513                li      a0,10
 214:   00c000ef                jal     220 <factorial>
 218:   fea42623                sw      a0,-20(s0)
 21c:   064000ef                jal     280 <exit>

00000220 <factorial>:
 220:   fe010113                addi    sp,sp,-32
 224:   00112e23                sw      ra,28(sp)
 228:   00812c23                sw      s0,24(sp)
 22c:   02010413                addi    s0,sp,32
 230:   fea42623                sw      a0,-20(s0)
 234:   fec42703                lw      a4,-20(s0)
 238:   00100793                li      a5,1
 23c:   00e7c663                blt     a5,a4,248 <factorial+0x28>
 240:   00100793                li      a5,1
 244:   0280006f                j       26c <factorial+0x4c>
 248:   fec42783                lw      a5,-20(s0)
 24c:   fff78793                addi    a5,a5,-1
 250:   00078513                mv      a0,a5
 254:   fcdff0ef                jal     220 <factorial>
 258:   00050793                mv      a5,a0
 25c:   fec42583                lw      a1,-20(s0)
 260:   00078513                mv      a0,a5
 264:   030000ef                jal     294 <__mulsi3>
 268:   00050793                mv      a5,a0
 26c:   00078513                mv      a0,a5
 270:   01c12083                lw      ra,28(sp)
 274:   01812403                lw      s0,24(sp)
 278:   02010113                addi    sp,sp,32
 27c:   00008067                ret

00000280 <exit>:
 280:   ff010113                addi    sp,sp,-16
 284:   00812623                sw      s0,12(sp)
 288:   01010413                addi    s0,sp,16
 28c:   7800d073                csrwi   mtohost,1
 290:   ffdff06f                j       28c <exit+0xc>
```

然后执行仿真命令生成.vcd 波形文件：

```
sktgroup@sktgroup-virtual-machine:~/Desktop/riscv-mini/test_c$ elf2hex 16 4096 factorial.elf > factorial.hex
sktgroup@sktgroup-virtual-machine:~/Desktop/riscv-mini/test_c$ ../VTile factorial.hex factorial.vcd
Enabling waves...
Starting simulation!
Simulation completed at time 1376 (cycle 137)
Finishing simulation!
```

用 gtkwave 打开 vcd 文件，观察处理器流水线工作中的若干信号。从上面反汇编得到的汇编代码可以看到，程序开始时首先通过 addi 指令偏移了栈指针，申请了 32 字节的栈空间，然后通过两条 sw 指令将寄存器 ra 和 s0 的值保存在栈的指定位置，其中寄存器 ra 是用于保存函数返回地址的，s0 的作用则是存放需要保存的数据。将这两个寄存器值保存到堆栈即是在完成我们常说的在函数调用前保存现场的操作。我们要观察的 factorial（）函数第一次调用时参数 10 的传入是在 li a0,10 这条指令进行的，它在跳转到 factorial 函数前将参数 10 传到 a0（x10）寄存器中，按照 RISC-V 的规范约定，函数的参数传递可以使用寄存器 a0~a7，这里 factoria 函数只有一个参数，所以使用 a0 进行参数传递。重点观察 li a0,10 指令，该指令对应的 PC 地址为 0x210。



如上波形图所示，icache 的 req.addr 端口与 next_pc 相等，若 cache 命中的情况下，在下一个周期 next_pc 的值装入 PC 的同时，就能从 icache 中取出对应的指令。上图中 PC 寄存器为 0x210 时，由于 cache 未命中，PC 的值在后续三个周期保持不变直至成功取出指令 00A00513。

之后指令在下一个时钟周期被传递到执行阶段的流水寄存器 fe_inst，同时传入立即数生成单元和控制单元作为输入，即 immGen_io_inst 和 io_inst。各执行单元对输入的指令进行解析，提取指定位段的数据，执行相应的操作。

其中 Io_ctrl_A_sel，Io_ctrl_B_sel，Io_ctrl_imm_sel 和 Io_ctrl_alu_op 为控制单元经过指令译码之后得到的部分控制信号，Io_ctrl_A_sel 和 Io_ctrl_B_sel 信号表示输入 alu 的操作数来源，Io_ctrl_imm_sel 表示立即数格式，Io_ctrl_alu_op 表示 alu 需要进行的操作。从波形图上可以看到，在取得 00A00513 指令输入后，经过译码，这些控制信号的输出分别为：

```
Io_ctrl_A_sel = 1          # 为 1 表示 alu 的 a 输入端口取源操作寄存器 rs1 的值
Io_ctrl_B_sel = 0          # 为 0 表示 alu 的 b 输入端口取立即数生成器的输出值
Io_ctrl_imm_sel = 001      # 为 001 表示立即数格式为 IMM_I，即 I 型指令的立即数格式
Io_ctrl_alu_op = 0000      # 为 0000 表示 ALU_ADD 操作
```

这些信号取值对应的详细意义可以到 riscv-mini/src/main/scala/mini/control.scala 查看源码。

regFile.io.raddr1 和 regFile.io.raddr2 为从指令中提取出的源操作数 1 和源操作数 2 的寄存器地址，从波形图看到 addr1 为 0，addr2 为 0A，对应 x0 和 x10 寄存器的地址，之后传递给寄存器文件（Register File）的读取端口，读出数据 regFile_io_rdata1 和 regFile_io_rdata2 均为 0。

ImmGen_io_out 为立即数生成单元的输出，即对指令立即数字段提取之后进行扩展之后的结果 0000000A。

根据指令译码的结果，知道 alu 将选取 rs1 寄存器（这里对应 x0）和立即数生成器的输出作为输入，从波形图上看到 alu_io_A 和 alu_io_B 分别为 0 和 0A，即 regFile_io_rdata1 和 ImmGen_io_out 的数值。alu_io_out 则为 alu 执行 ALU_ADD 操作的结果，为 0A。

最后，alu 计算结果在下一个周期又送到了 ew_reg_alu，供写回阶段使用。波形图上看到在写回阶段，regFile_io_wen 写允许位被拉高，写地址 regFile_io_waddr 为 0A，即目的寄存器 x10 的地址，写入数据 regFile_io_wdata 为 0A，来自 ew_reg_alu。观察 reg(10)即 x10 寄存器，可以看到其数值在下一个周期变为 0A，证明写回阶段成功执行。

五、实验结果

实验成功完成。

六、实验总结与体会

通过本次实验，我深入理解了 RISC-V 处理器架构和 Chisel 硬件设计语言，掌握了汇编语言编程、指令集和控制信号的工作原理。通过编写和仿真 RISC-V 汇编程序，我不仅加深了对寄存器、数据传输和运算指令的认识，还通过分析 C 程序中的函数调用，了解了参数传递和栈操作的细节。同时仿真波形图的分析让我直观地观察到了指令执行过程中寄存器和控制信号的变化，从而对计算机体系结构的底层实现有了更深刻的理解。

总的来说，这次实验不仅提升了我的实践技能，也为我未来在计算机体系结构领域的深入学习和研究奠定了坚实的基础。

指导教师批阅意见： **成绩评定：** 指导教师签字： 年　月　日
备注：

注：1、报告内的项目或内容设置，可根据实际情况加以调整和补充。
　　2、教师批改学生实验报告时间应在学生提交实验报告时间后 10 日内。