**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。

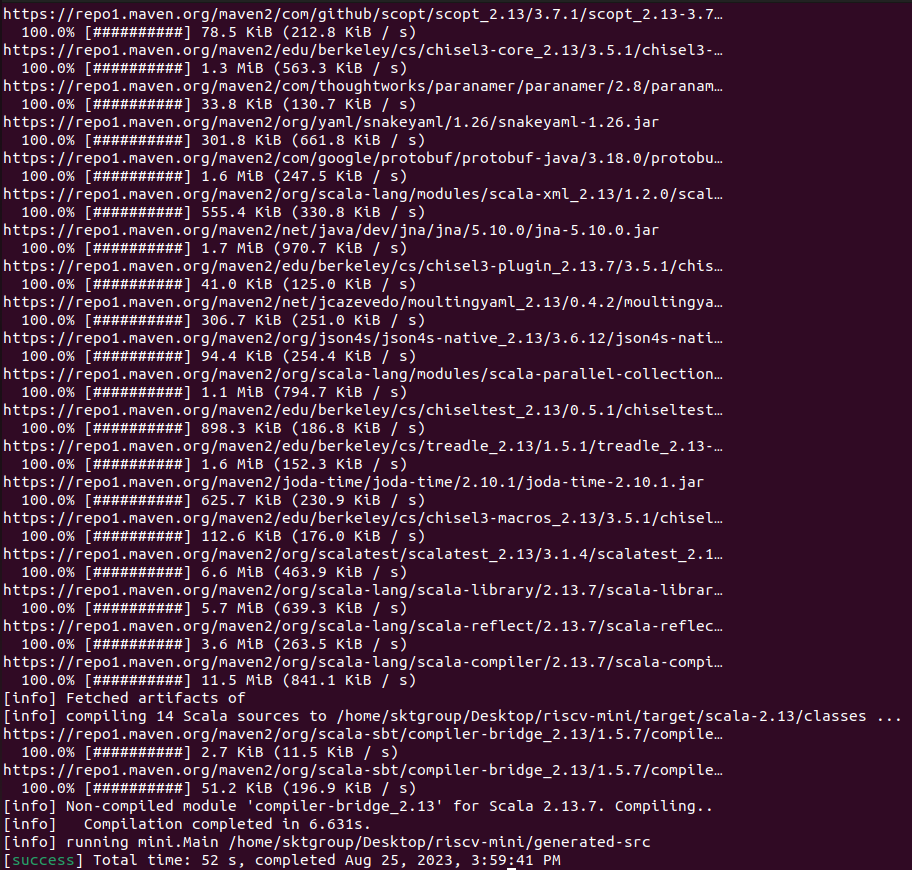
**⼀、实验环境的搭建**

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 |



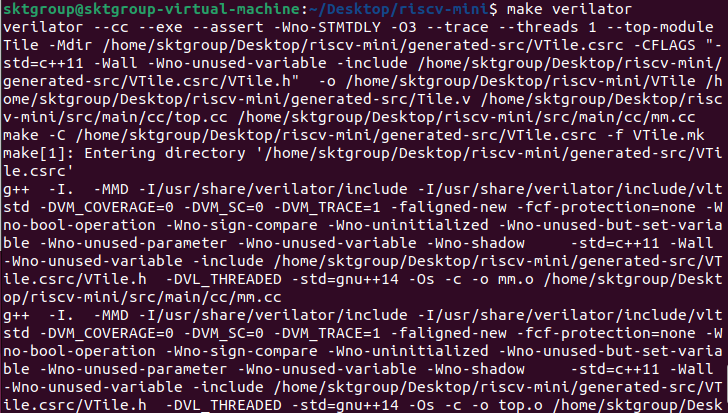
进入generated-src路径可以看到生成的处理器verilog代码文件Tile.v。

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

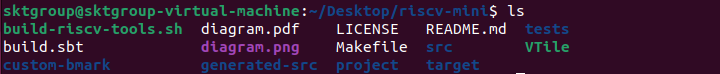


编译完处理器，还要编译处理器的仿真器（用于后续在riscv mini上的程序仿真），首先安装verilator和g++，然后在项目根目录执行编译命令：

|  |
| --- |
| 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。

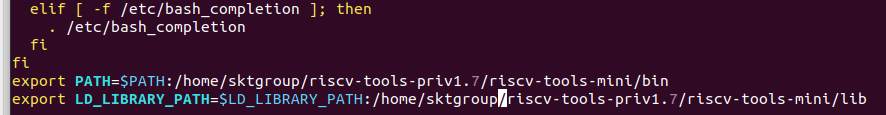


接下来，若想要在riscv-mini上运⾏⾃⼰编写的程序，需要安装特权指令集1.7版本的GNU⼯具链。Riscv32-unkown-elf-gcc是一款交叉编译器，可以在非RISCV平台上，将C语言或RISCV32I指令集的汇编语言编译成RISCV32I机器可以执行的可装入字节文件。这⾥直接提供编译好的⼆进制程序以及需要的相关⽂件，只需要将压缩包解压并且添加环境变量即可使⽤。使⽤如下命令将riscv-tools-priv1.7.tar.gz解压。

|  |
| --- |
| 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 |



然后更新系统变量：

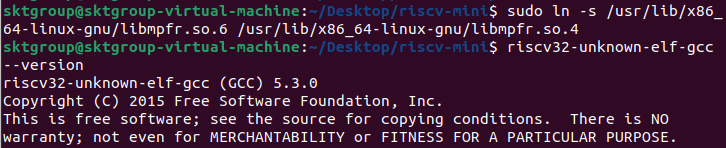
|  |
| --- |
| 1. source ~/.bashrc |

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 |



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）。

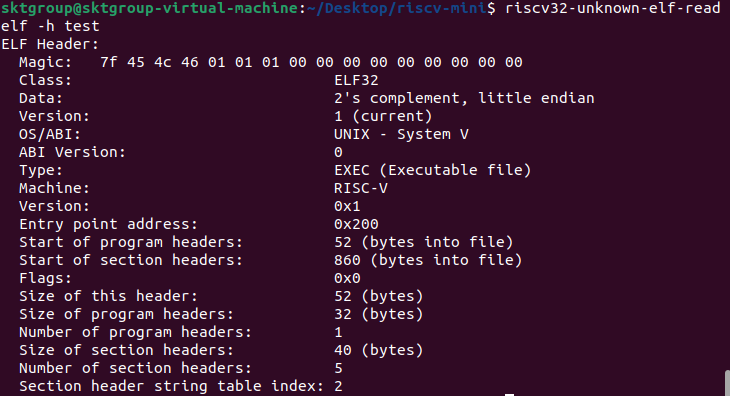
|  |
| --- |
| 1. .text *# Define beginning of text section* 2. .global \_start *# Define entry \_start* 3. \_start: 4. li x6, 1 *# x6 = 1* 5. li x7, 2 *# x7 = 2* 6. add x5, x6, x7 *# x5 = x6 + x7* 7. exit: 8. csrw mtohost, 1 9. j exit 10. .end *# End of file* |

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 |

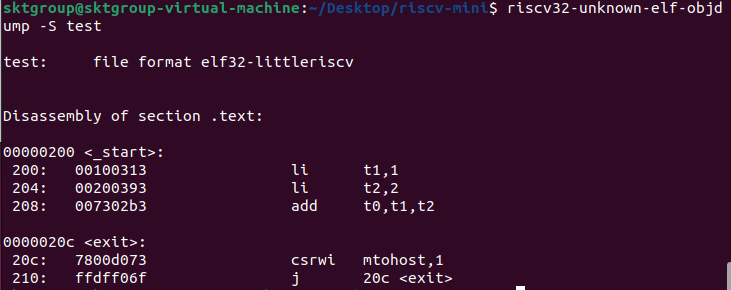
编译完成后我们便可得到elf文件，通过riscv32-unknown-elf-readelf我们可以查看其具体信息。如下通过查看elf头可以看到该elf文件的系统架构为RISC-V，入口地址为0x200。

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



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。

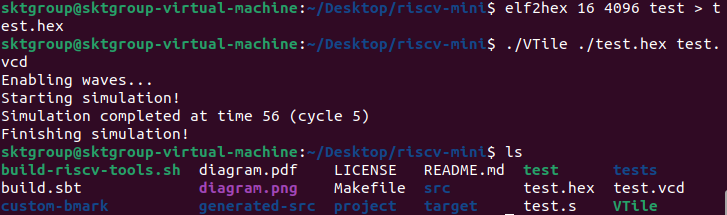
|  |
| --- |
| 1. riscv32-unknown-elf-objdump -S test |



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 |



1. **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。

|  |
| --- |
| 1. int factorial(int num); 2. void exit(); 3. void main(){ 4. int ans = factorial(10); 5. exit(); 6. } 7. int factorial(int num){ 8. if(num <= 1) return 1; 9. else return num \* factorial(num - 1); 10. } 11. void exit(){ 12. while(1) 13. \_\_asm\_\_ \_\_volatile\_\_("csrw mtohost, 1"); 14. } |

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。

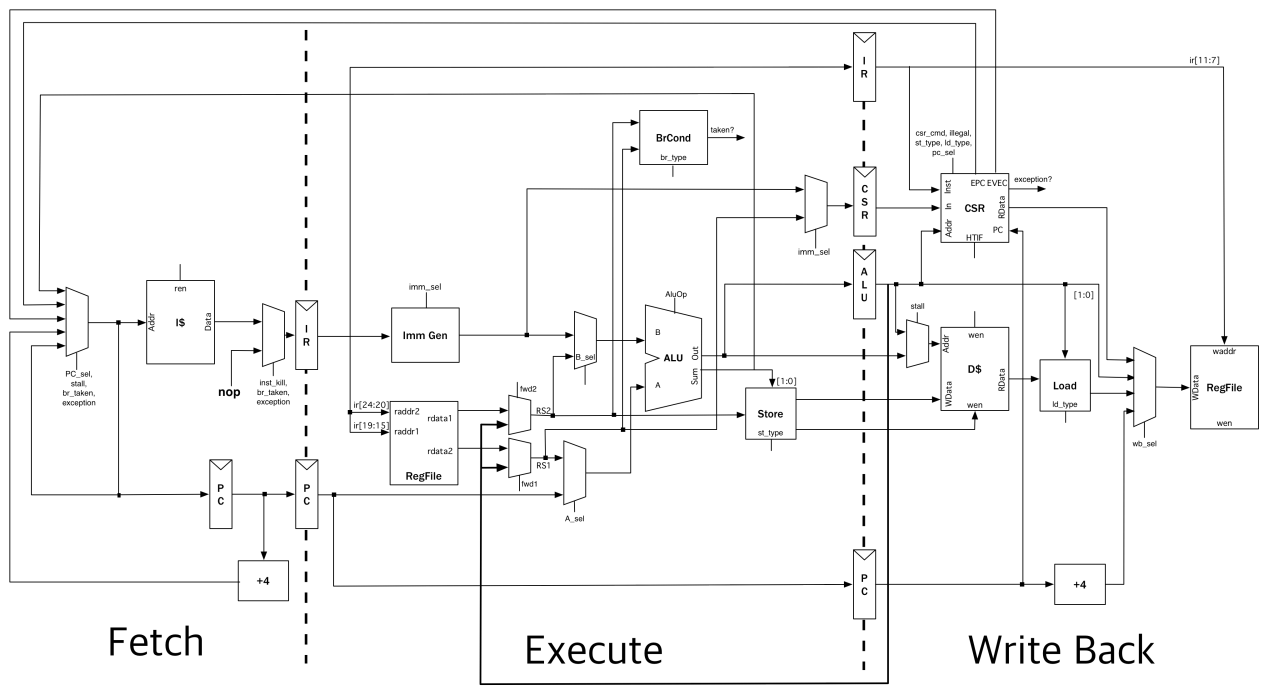
|  |
| --- |
| 1. riscv32-unknown-elf-gcc -nostdlib -Ttext=0x200 -march=RV32I -o factorial.elf factorial.c |

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。

**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。

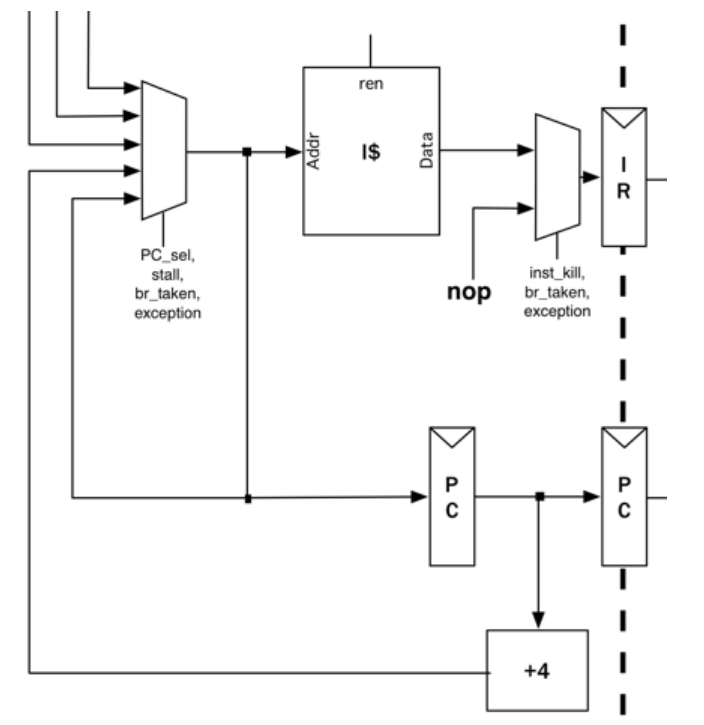
riscv-mini具有三级流水线，其数据通路如下图所示(注意图中RegFile中Data1与Data2的标注存在错误,位置应该互换)。具体流水线实现代码位于riscv-mini/src/main/scala/Datapath.scala中。



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> |

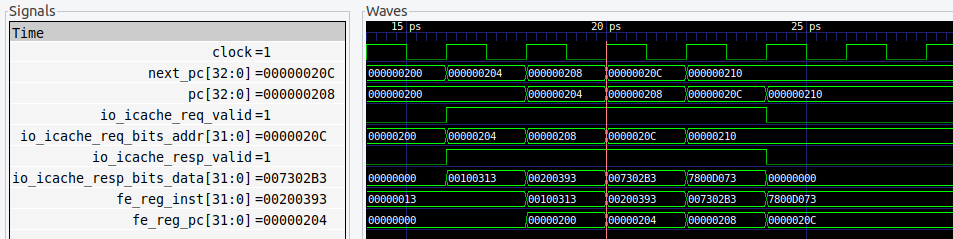
首先观察程序执行过程中取指阶段的内容。在这个阶段当中需要根据PC值取出将要执行的指令，该阶段的数据通路如下图所示。首先通过一个多路选择器根据情况选择将要执行的下一条指令地址，随后访问地址Cache获取对应地址中的指令，若流水线正常执行未发生分支与异常等情况便可将读取到的指令放入到Fetch/Execute流水线寄存器中给下一个阶段使用，否则用空指令冲刷流水线产生一个空泡。



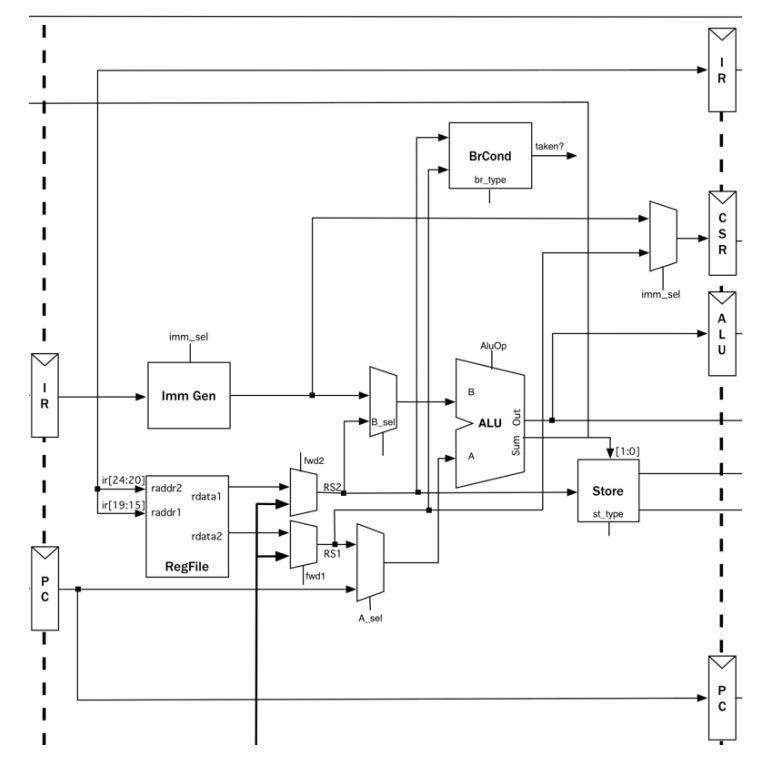
取指阶段的Chisel代码如下。其中next\_pc为一个多路选择器，根据控制信号选择下一条指令的地址。当流水线开始指令时pc寄存器获取next\_pc输出的地址，并从icache读取对应地址的指令。当流水线未停顿时便可将pc与读取到的指令存入到流水线寄存器fe\_reg.pc与fe\_reg.inst中。

|  |
| --- |
| /\*\* \*\*\*\* 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。



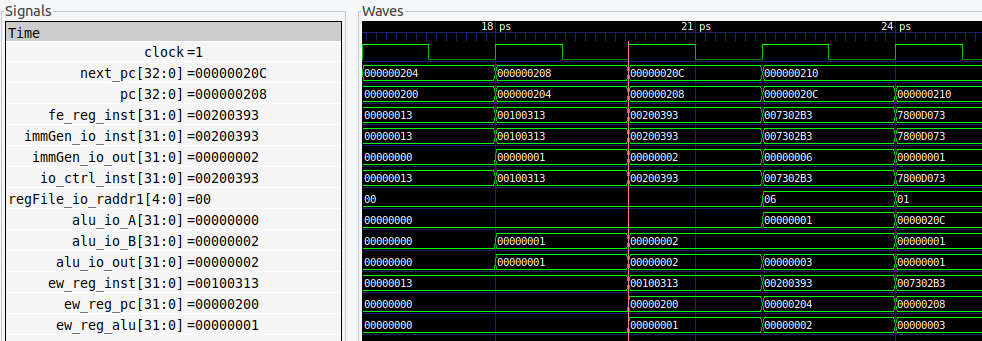
接下来是执行阶段，在该阶段当中需要完成指令译码、准备操作数(从regfile中读取或产生立即数)、指令执行以及访存(访存指令)。该阶段的数通路如下图所示。首先根据需要执行的指令通过立即数生成单元以及访问寄存器文件获得执行阶段所需要使用的操作数作为ALU的输入。具体的输入由图中的多个多路选择器进行选择，可能为寄存器文件中读取到的数值、立即数、PC或者由处于下一个阶段的指令前递来的数值。与此同时若为分支指令则由BrCond模块进行分支判断，若需要分支则下一条指令的PC地址为分支指令的目标地址。



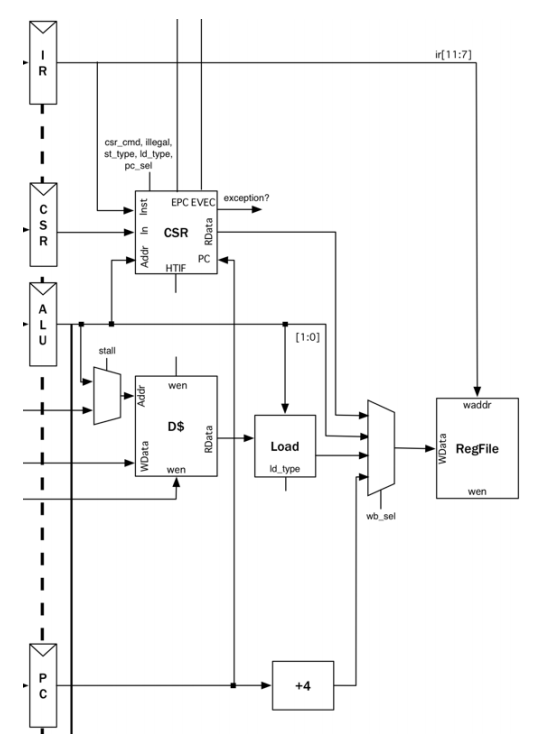
执行阶段的Chisel代码如下。自上而下的看，首先将取指阶段得到的指令传给控制单元获得该指令的控制信号，访问寄存器文件读取需要的操作数，通过立即数生成单元获得立即数。接下来对ALU的输入数据做出选择，来源可能为立即数、从寄存器文件中读取到的数据或者处于EXE/WB流水线寄存器中的数据（前递）。同时判断该指令是否产生分支以及使用计算出的地址访问数据cache进行访存操作。若流水线正常执行则将该阶段产生的数据放入到EXE/WB流水线寄存器中供给下一个阶段使用，反之冲刷流水线产生气泡或者停顿流水线。

|  |
| --- |
| /\*\* \*\*\*\* 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\_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.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    } |

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。



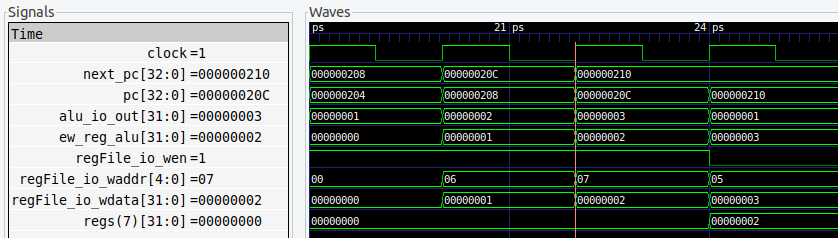
最后是写回阶段，在该阶段当中需要将指令执行的结果写回到寄存器中。根据指令类型将对应的结果写入到寄存器文件当中，若为访存指令则在该阶段获得防存的结果并提取需要的位将其写回。该阶段的数据通路如下图所示。



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。

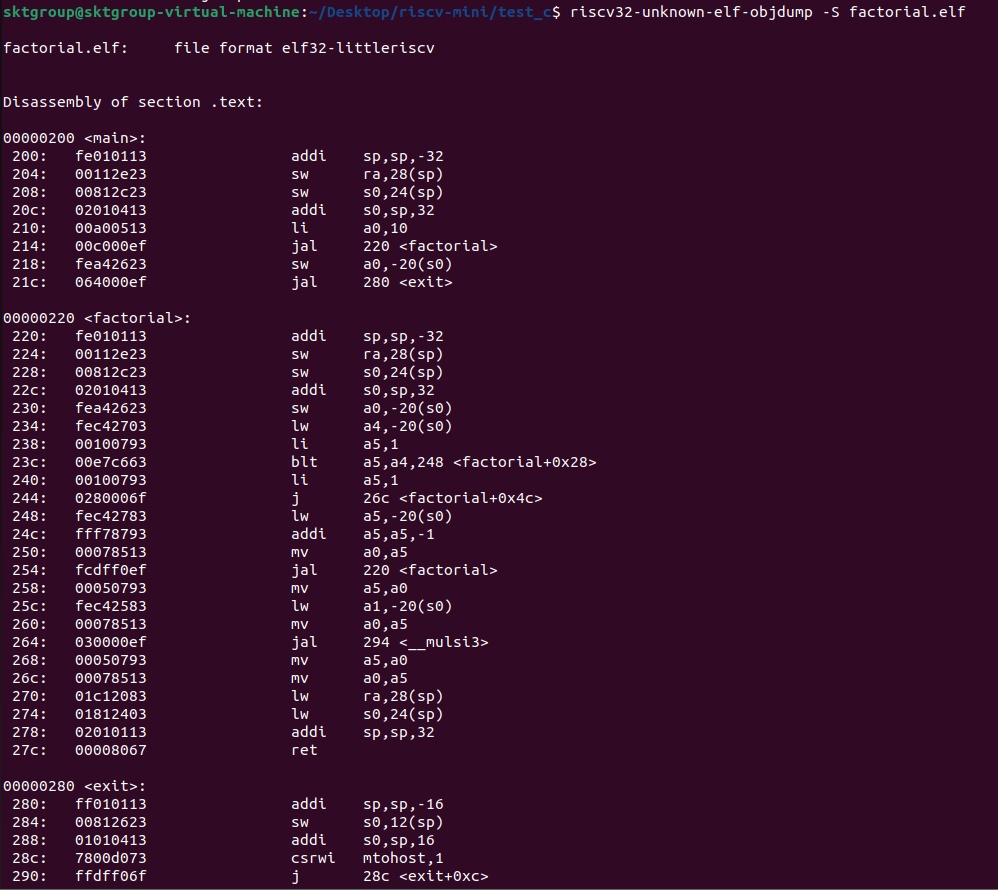
|  |
| --- |
| // Load    val loffset = (ew\_reg.alu(1) << 4.U).asUInt | (ew\_reg.alu(0) << 3.U).asUInt    val lshift = io.dcache.resp.bits.data >> loffset    val load = MuxLookup(      ld\_type,      io.dcache.resp.bits.data.zext,      Seq(        LD\_LH -> lshift(15, 0).asSInt,        LD\_LB -> lshift(7, 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 -> csr.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 |

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。

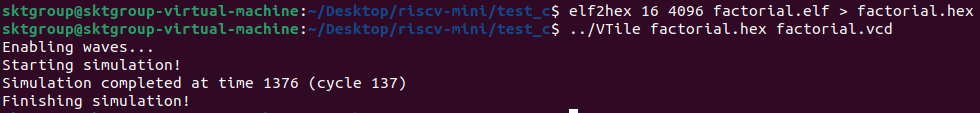


实验内容：解释上述C代码中factorial(10)函数的参数10传入寄存器的过程。

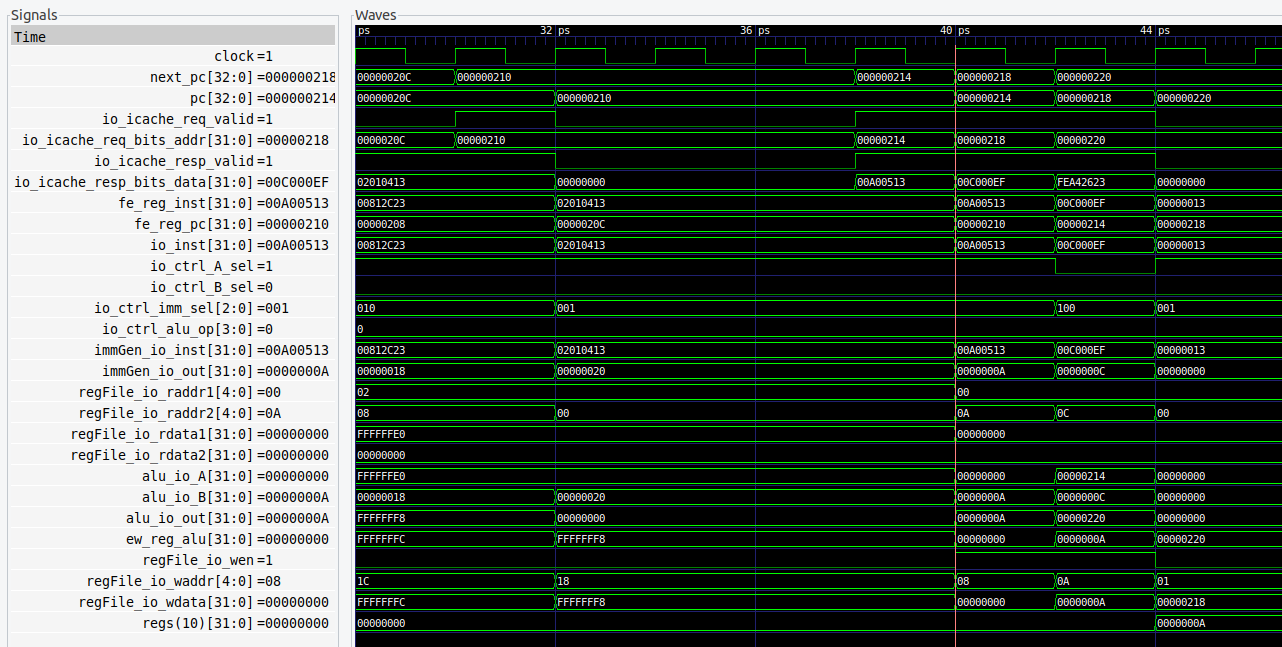
可以先将上述C代码编译后得到的elf文件反汇编得到汇编代码，查看处理器执行的指令及对应的地址：



然后执行仿真命令生成.vcd波形文件：



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。