中国科学技术大学

“高性能处理器体系结构”课题报告

报告题目：基于 RISC-V 架构的 USTCPV 的简单实现

小组组长：谢怡晨、唐凯成

小组成员：谢怡晨、唐凯成

导师姓名：安虹

2019 年 1 月 7 日

一、研究小组成员及其承担的主要工作

学号	姓名	所在学院	在研究和报告撰写中承担的主要工作
PB15051175	谢怡晨	1511	riscv-tools、riscv-linux、课题报告
PB16001695	唐凯成	1600	USTCPV 内核实现、USTCPV 内核测试、PPT 制作与答辩、课题报告

二、进度安排

2018.9~2018.10　　组队、选题
2018.10~2018.11　　学习 Verilog 等相关知识、调研 RISC-V、e200_opensource
2018.11~2018.12　　初步实现USTCPV内核、编译运行riscv-tools、riscv-linux
2018.12~2018.12.29 完善 USTCPV、模拟 helloworld，准备 PPT 及答辩
2018.12~2019.1.7　　撰写课程报告

三、摘要、关键词

关键词：RISCV 指令集、CPU 核、RISCV-Tools
摘要：本报告对最近热门的 RISCV 指令集进行了较为详细的介绍，分析了其设计特点和 RISCV 指令集相对其他指令集的优势，并实现了一个 RISCV 指令集构架下的 CPU 内核，在核外实现了 cache 及其管理。最后使用 RISCV 官方提供的 RISCV-Tools 中的 riscv-gcc 将编写的 hello world 程序转为 riscv 指令，并在 riscv 核上运行后再控制页面输出正确结果。

四、研究报告

（一）、背景和目标

如图 1 所示，指令集体系结构（ISA）是构成处理器底层硬件和运行于其上的软件之间的桥梁与接口，是现代计算机处理器中重要的一个抽象层次。经过几十年的发展，到今天已经相继诞生或消亡过了几十种不同的指令集架构。指令集架构主要可分为复杂指令集（CISC）和精简指令集（RISC）。CISC 不仅包含了处理器常用的指令，还包含了许多不常用的特殊指令，可以用较少的指令完成更多的操作，在早期曾经是主流。但是随着指令集的发展，越来越多的特殊指令被添加到 CISC 指令集中，CISC 的缺点开始显现：典型程序用到的 80%的指令只占所有指令类型的 20%，即 CISC 指令集中只有大约 20%被经常使用，80%很少被用到；并且那些很少被用到的特殊指令让 CPU 设计变得极为复杂，大大增加了硬件设计的时间成本和面积开销。因此，现代指令集架构基本使用 RISC，RISC 只包含处理器常用的指令，而对于不常用的操作，则通过执行多条常用指令的方式来达到同样的效果。

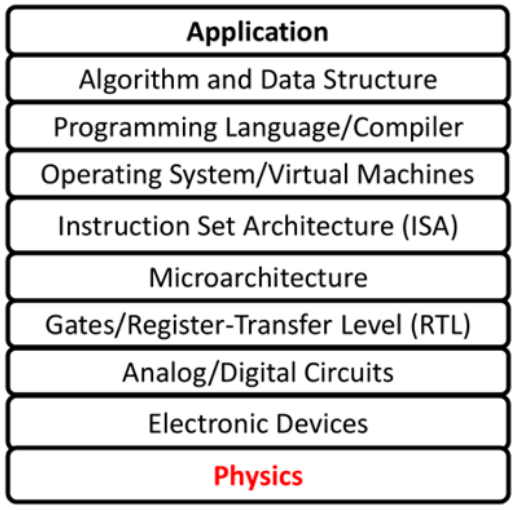


图 1. 计算机系统层次结构

RISC-V 就是基于精简指令集（RISC）原理建立的开放指令集架构。它诞生于最近十年，并且与几乎所有的旧架构不同，它是一个开源的指令集架构，属于一个开放的，非营利性质的基金会，不受任何单一公司的影响。RISC-V 基金会的目标是保持 RISC-V 的稳定性，仅仅出于技术原因缓慢而谨慎地发展它，并力图让它之于硬件如同 Linux 之于操作系统一样受欢迎。

RISC-V 的核心思想就是模块化。它的核心是一个名为 RV32I 的基础 ISA，运行一个完整的软件栈。模块化来源于可选的标准扩展，根据应用程序的需要，硬件可以包含或不包含各种扩展，如 M（整数乘法与除法指令）、F（单精度浮点指令）、D（双精度浮点指令）、C（压缩指令）等（见表 1）。

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	存储器原子操作指令和 Load-Reserved/Store-Conditional 指令
F	26	单精度（32bits）浮点指令
D	26	双精度（64bits）浮点指令，必须支持 F 扩展指令

C	46	压缩指令，指令长度为 16 位

表 1. RISC-V 的模块化指令集

以上模块的一个特定组合“IMAFD”，也被称为“通用”组合,用 G 表示。即 RV32G=RV32IMAFD. 这种模块化特性使得 RISC-V 具有了袖珍化、低能耗的特点，而这对于嵌入式应用可能至关重要。RISC-V 编译器得知当前硬件包含哪些扩展后，便可以生成当前硬件条件下的最佳代码。例如在追求小面积、低功耗的嵌入式场景下，可以选用 RV2EC 架构；在大型的 64 位架构下可以选用 RV64G。除此之外，RISC-V 还在不断增加完善新的扩展，如 B（位操作）、H（特权态架构扩展，支持管理程序）、J（动态翻译语言）、L（十进制浮点）、N（用户态中断）等。

除了模块化之外，RISC-V 还有一个特色——规整的指令编码。这样可以使得在流水线中能够尽快读取通用寄存器组，提高处理器性能、优化时序。反观很多现存的商用 RISC 架构，经过多年反复修改增加新指令后，其指令编码中寄存器索引位置不固定，增加了译码阶段的复杂度。具体以 RV32I 为例。RV32I 的所有指令只有六种格式，并且都是 32 位长（见图 2、表 2）。

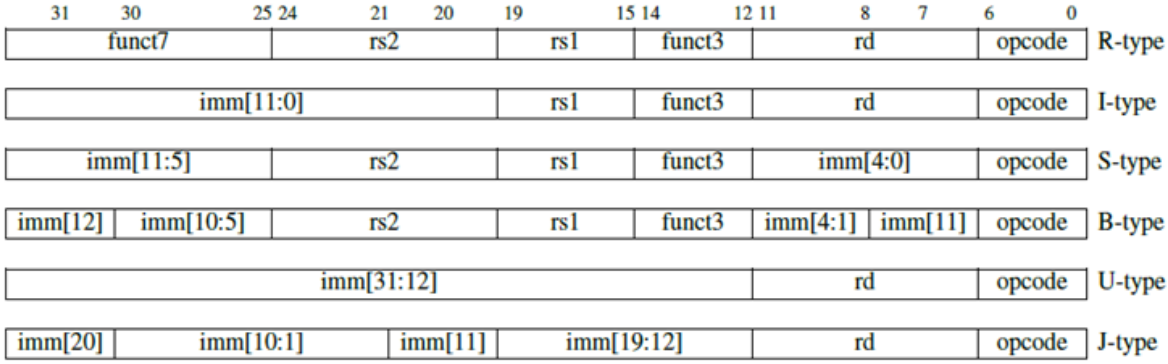


图 2. RV32I 指令编码格式

[31:25]	[24:20]	[19:15]	[14:12]	[11:7]	[6:0]	Instr
imm[31:12]				rd	0110111	U lui
imm[31:12]				rd	0010111	U auipc
imm[20|10:1|11|19:12]				rd	1101111	J jal
imm[11:0]		rs1	000	rd	1100111	I jalr
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	B beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	B bne
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	B blt
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	B bge
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	B bltu
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	B bgeu
imm[11:0]		rs1	000	rd	0000011	I lb
imm[11:0]		rs1	001	rd	0000011	I lh

imm[11:0]		rs1	010	rd	0000011	I lw
imm[11:0]		rs1	100	rd	0000011	I lbu
imm[11:0]		rs1	101	rd	0000011	I lhu
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	S sb
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	S sh
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	S sw
imm[11:0]		rs1	000	rd	0010011	I addi
imm[11:0]		rs1	010	rd	0010011	I slti
imm[11:0]		rs1	011	rd	0010011	I sltiu
imm[11:0]		rs1	100	rd	0010011	I xori
imm[11:0]		rs1	110	rd	0010011	I ori
imm[11:0]		rs1	111	rd	0010011	I andi
0000000	shamt	rs1	001	rd	0010011	I slli
0000000	shamt	rs1	101	rd	0010011	I srli
0100000	shamt	rs1	101	rd	0010011	I srai
0000000	rs2	rs1	000	rd	0110011	R add
0100000	rs2	rs1	000	rd	0110011	R sub
0000000	rs2	rs1	001	rd	0110011	R sll
0000000	rs2	rs1	010	rd	0110011	R slt
0000000	rs2	rs1	011	rd	0110011	R sltu
0000000	rs2	rs1	100	rd	0110011	R xor
0000000	rs2	rs1	101	rd	0110011	R srl
0100000	rs2	rs1	101	rd	0110011	R sra
0000000	rs2	rs1	110	rd	0110011	R or
0000000	rs2	rs1	111	rd	0110011	R and
0000 pred	succ	00000	000	00000	0001111	I fence
0000 0000	0000	00000	001	00000	0001111	Ifence.i
ECALL		00000	PRIV	00000	1110011	I ecall
EBREAK		00000	PRIV	00000	1110011	I ebreak
csr		rs1	001	rd	1110011	I csrrw
csr		rs1	010	rd	1110011	I csrrs
csr		rs1	011	rd	1110011	I csrrc
csr		zimm	101	rd	1110011	I csrrwi
csr		zimm	110	rd	1110011	I cssrrsi
csr		zimm	111	rd	1110011	I csrrci

表 2. RV32I 的 47 条指令

与所有 RISC 处理器架构一样，RISC-V 架构使用专门的 Load/Store 指令访问存储器。不同之处是，RISC-V 的访存指令更为简洁，这也是 RISC-V 的特性之一。RISC-V 没有特殊的堆栈指令，不支持延迟加载，也不支持地址自增自减的模式，仅支持主流的小端格式。这一切都是为了简化基本指令集、简化硬件设计的成本。

高效的分支跳转指令也是 RISC-V 的特点之一。RISC-V 去掉了延迟分支指令，也没有使用条件码，简化了乱序执行时的依赖计算。RISC-V 还将过程调用和无条件跳转合并为一条跳转并链接指令（jal）。若将下一条指令 PC + 4 的地址保存到目标寄存器 rd 中，通常是返回地址寄存器 ra，便可以用它来实现过程调用；如果使用零寄存器 x0 替换 ra 作为目标寄存器 rd，则可以实现无条件跳转，因为 x0 不能被更改。跳转和链接指令的寄存器版本（jalr）同样是多用途的。它可以调用地址是动态计算出来的函数，或者也可以实现调用返回（将 ra 作为源寄存器 rs1，零寄存器 x0 作为目的寄存器 rd）。省去了错综复杂的程序调用指令。

RV32M 向 RV32I 中添加了整数乘法和除法指令（表 3），特别增加了取余数指令（rem/remu），更符合程序员需求；还特别区分了有/无符号数的乘法，都是有符号数-mulh、都是无符号数-mulhu、一个有符号一个无符号-nulhsu.

[31:25]	[24:20]	[19:15]	[14:12]	[11:7]	[6:0]	Instr
0000001	rs2	rs1	000	rd	0110011	R mul
0000001	rs2	rs1	001	rd	0110011	R mulh
0000001	rs2	rs1	010	rd	0110011	R mulhsu
0000001	rs2	rs1	011	rd	0110011	R mulhu
0000001	rs2	rs1	100	rd	0110011	R div
0000001	rs2	rs1	101	rd	0110011	R divu
0000001	rs2	rs1	110	rd	0110011	R rem
0000001	rs2	rs1	111	rd	0110011	R remu

表 3. RV32M 扩展的 8 条指令

对于浮点扩展 RV32F 和 RV32D，RISC-V 有两条加载指令（flw，fld）和两条存储指令（fsw，fsd）。他们和 lw 和 sw 拥有相同的寻址模式和指令格式。添加到标准算术运算中的指令有：fadd.s、fadd.d、fsub.s、fsub.d、fmul.s、fmul.d、fdiv.s、fdiv.d，还包括平方根指令（fsqrt.s、fsqrt.d），最小值和最大值指令（fmin.s、fmin.d、fmax.s、fmax.d）。RV32F 和 RV32D 指令编码上的区别只是某一位上 0 或 1 的区别，节省了编码空间也便于译码。

（二）、设计与实现

1、USTCPV 内核设计与仿真

（1）总述

对于整个 riscv 内核的设计，与设计 MIPS 指令集的 cpu 有相似之处也有不同之处，区别主要在于流水线深度的改变和对于 riscv 指令的解析。对于本次 riscv 流水线的实现，最终实现为三级流水，即类似蜂鸟 e200 里的设计，第一级流水实现取指，第二级流水实现译址、

执行和写会寄存器，第三级流水负责访存和写回存储器。另外在整个 cpu 核外，还实现了一个简单的 cache，即可以实现对 RAM 的一个 cache 作用，后文将依次介绍各级流水及 cache 的简单实现。

（2）第一级流水

第一级流水整体较为简单，只负责实现对指令的读出。

在实现时使用了一个 IDATA 作为指令存储器，事先例化好存储器中的命令并在运行时从对应地址读出指令即可。

```
always@(posedge CLK)
   begin
      XIDATA <= RES ? { ALL0[31:12], 5'd2, ALL0[6:0] } : HLT ? XIDATA : IDATA;
   end
```

新的 PC 值的产生从重置 PC、下周起 PC 和当前 PC 中选择产生。

```
      NXPC <= RES ? RESET_PC : HLT ? NXPC : NXPC2;

      NXPC2 <=  RES ? RESET_PC : HLT ? NXPC2 :   // reset and halt
              JREQ ? JVAL :                      // jmp/bra
                   NXPC2+4;                     // normal flow
      PC   <= RES ? RESET_PC : HLT ? PC : NXPC; // current program counter
```

（3）第二级流水

在第二级流水中，同时完成译址、执行和写会寄存器，分别对这三个阶段进行分析：

A）译址

在译址阶段，由于 RISCV 指令的高度规整性，故只需直接对从 IDATA 中读出的数据按照不同位进行译码即可，其中主要识别的是三段的内容：

a、opcode（0~6 位）

b、funct3（12~14 位）

c、funct7（25~31 位）

由控制单元对三者的判断来确定具体进入的数据通路：

```
   // cut instruction
   wire [6:0] OPCODE = FLUSH ? 0 : XIDATA[6:0];
   wire [4:0] DPTR   = XIDATA[11: 7];
   wire [2:0] FCT3   = XIDATA[14:12];
   wire [4:0] S1PTR  = XIDATA[19:15];
   wire [4:0] S2PTR  = XIDATA[24:20];
   wire [6:0] FCT7   = XIDATA[31:25];
```

不管为何种类型的指令，均可直接从对应字段读出寄存器内的值用于后续操作。

B）执行

对于执行阶段，可根据不同类型的指令分别进行分析：

a）逻辑运算指令

对于逻辑运算型指令，只需对读入的立即数进行拓展（或使用寄存器内的值），在根据

FCT3 字段决定的指令具体操作进行计算即可实现：

```
   wire [31:0] RDATA = FCT3==0 ? (FCT7[5] ? U1REG-U2REG : U1REG+U2REG) :
                       FCT3==1 ? U1REG<<U2REG[4:0] :
                       FCT3==2 ? S1REG<S2REG?1:0 : // signed
                       FCT3==3 ? U1REG<U2REG?1:0 : // unsigned
                       FCT3==5 ? RDATA_FCT3EQ5 :
                       // (FCT7[5] ? U1REG>>>U2REG[4:0] : U1REG>>U2REG[4:0]) :
                       FCT3==4 ? U1REG^U2REG :
                       FCT3==6 ? U1REG|U2REG :
                       FCT3==7 ? U1REG&U2REG :
                                 0;
```

b）转跳类指令

由于转跳类指令较少，可直接根据两个寄存器内数字大小判断是否可以进行转跳，地址直接由当前 PC 地址和位移量相加产生。

```
   wire BMUX      = BCC==1 && (
                      FCT3==4 ? S1REG>=S2REG : // signed
                      FCT3==5 ? S1REG<=S2REG : // signed
                      FCT3==6 ? U1REG>=U2REG : // unsigned
                      FCT3==7 ? U1REG<=U2REG : // unsigned
                      FCT3==0 ? U1REG==U2REG :
                      FCT3==1 ? U1REG!=U2REG :
                                0);

   wire       JREQ = (JAL||JALR||BMUX);
   wire [31:0] JVAL = SIMM + (JALR ? U1REG : PC);
```

c）访存类指令

对于访存类指令，根据 FCT3 灵活德确定 load 或 stored 的位数，地址偏移由逻辑运算单元完成。

```
// L-group of instructions (OPCODE==7'b0000011)
   wire [31:0] LDATA = FCT3==0||FCT3==4 ? ( DADDR[1:0]==3 ? { FCT3==0&&DATAI[31] ? ALL1[31:
8]:ALL0[31: 8] , DATAI[31:24] } :
                                       DADDR[1:0]==2 ? { FCT3==0&&DATAI[23] ? ALL1[31:
8]:ALL0[31: 8] , DATAI[23:16] } :
                                       DADDR[1:0]==1 ? { FCT3==0&&DATAI[15] ? ALL1[31:
8]:ALL0[31: 8] , DATAI[15: 8] } :
                                                       { FCT3==0&&DATAI[ 7] ? ALL1[31:
8]:ALL0[31: 8] , DATAI[ 7: 0] } ):
                       FCT3==1||FCT3==5 ? ( DADDR[1]==1   ? { FCT3==1&&DATAI[31] ?
ALL1[31:16]:ALL0[31:16] , DATAI[31:16] } :
                                                       { FCT3==1&&DATAI[15] ?
ALL1[31:16]:ALL0[31:16] , DATAI[15: 0] } ) :
                                       DATAI;
```

```
// S-group of instructions (OPCODE==7'b0100011)
wire [31:0] SDATA = FCT3==0 ? ( DADDR[1:0]==3 ? { U2REG[ 7: 0], ALL0 [23:0] } :
                          DADDR[1:0]==2 ? { ALL0 [31:24], U2REG[ 7:0], ALL0[15:0] } :
                          DADDR[1:0]==1 ? { ALL0 [31:16], U2REG[ 7:0], ALL0[7:0] } :
                                          { ALL0 [31: 8], U2REG[ 7:0] } ) :
                FCT3==1 ? ( DADDR[1]==1   ? { U2REG[15: 0], ALL0 [15:0] } :
                                          { ALL0 [31:16], U2REG[15:0] } ) :
                          U2REG;
```

C）写回

写回阶段的整体实现较为简单，目标写回的地址已在译址阶段给出，故此处只需考虑不同指令写回的数据的来源不同，因此只需要将对应指令类型的数据存回即可。

```
    REG[DPTR] <=   RES ? RESET_SP  :       // reset sp
                  HLT ? REG[DPTR] :       // halt
                !DPTR ? 0 :               // x0 = 0
                AUIPC ? NXPC+SIMM :
                 JAL||
                 JALR ? NXPC :
                  LUI ? SIMM :
                  LCC ? LDATA :
                  MCC ? MDATA :
                  RCC ? RDATA :
                  CCC ? CDATA :
                       REG[DPTR];
```

（4）第三级流水

第三级流水仍使用之前的结构，由于只有访存需要进入第三级故原数据通路无需改变，只是由于流水层次到达三时，会出现一定的冒险与竞争的问题，因此在转跳时需清空流水线，及加入两个指令寄存器和一个 flush 即可：

```
    FLUSH <= RES ? 2 : HLT ? FLUSH :       // reset and halt
                  FLUSH ? FLUSH-1 :
                  (JAL||JALR||BMUX||RES) ? 2 : 0;  // flush the pipeline
```

至此即实现了 riscv 内核的全部部分。

（5）cache 的实现

在实现了完整的 riscv 内核后，加入对 riscv 内核的 cache 控制单元，以提高整个核的运行速度。对于数据存储器的 cache 结构如下：

```
// data cache

reg  [55:0] DCACHE [0:63]; // data cache
reg  [63:0] DTAG = 0;     // data cache tag

wire [5:0]  DPTR   = DADDR[7:2];
```

```
    wire [55:0] DCACHEO = DCACHE[DPTR];
    wire [31:0] DCACHED = DCACHEO[31: 0]; // data
    wire [31:8] DCACHEA = DCACHEO[55:32]; // address

    wire DHIT = RD ? DTAG[DPTR] && DCACHEA==DADDR[31:8] : 1;
```

其中 DCACHE 为主缓存，缓存每次读取的存储器内的值，并通过后面的 DHIT 来标记本次使用中的饿 cache 是否命中，命中后则从 cache 内直接读取对应数据，未命中则将该数从存储器中读出并加入 cache 中：

```
if(!DHIT)
       begin
          DCACHE[DPTR] <= { DADDR[31:8], RAMFF2 };
          FFX         <= 1;
          DTAG[DPTR]   <= FFX; // cached!
          FFX2        <= FFX;
       end
```

命中后的处理：

```
    assign DATAI = DADDR[31] ? 0 : DCACHED;
    assign IDATA = ICACHED;
```

至此实现了内核外的 cache 功能。

2、riscv-tools 与 riscv-linux 探索

riscv-tools 的源代码在 github 上被维护成一个宏项目，其包含了所有 RISC-V 相关工具链、仿真器和测试套件等子项目：

riscv-fesvr——用于实现上位机和 CPU 之间通信机制的库

riscv-pk——提供 RISC-V 可执行文件运行的程序运行环境，同时也带有简单的 bootloader

riscv-isa-sim——基于 C/C++开发的 RISC-V 指令集模拟器，又名“Spike”

以上三者可以在主机上模拟执行 RISC-V 程序。

riscv-gnu-toolchain 是支持 RISC-V 的 GNU 工具链，包含：

riscv-gcc——GCC 编译器

riscv-binutils-gdb——汇编器、链接器、GDB 调试工具等

riscv-glibc——GNU C 标准库实现

riscv-llvm——基于 LLVM 编译器的框架

riscv-openocd——基于 OpenOCD 的 RISC-V 调试器软件

riscv-opcodes——RISC-V 操作码信息转换脚本

riscv-tests——RISC-V 指令集测试用例

riscv-qemu——支持 RISC-V 的 QEMU 模拟器

下载编译以上工具链：

```
git clone --recursive https://github.com/riscv/riscv-gnu-toolchain
cd riscv-gnu-toolchain
./configure --prefix=/opt/riscv --enable-multilib
make newlib
make linux
export PATH=$PATH:/opt/riscv/bin
export RISCV=/opt/riscv
```

下载编译安装 riscv-qemu 模拟器：

```
git clone https://github.com/riscv/riscv-qemu.git
cd riscv-qemu
./configure --target-list=riscv64-softmmu,riscv32-softmmu
make
sudo make install
```

利用开源 busybear-linux 编译出 riscv-linux：

```
git clone https://github.com/michaeljclark/busybear-linux.git
cd busybear-linux
make

cd..
git clone https://github.com/riscv/riscv-linux.git
cd riscv-linux
git checkout riscv-linux-4.14
cp ../busybear-linux/conf/linux.config .config
make ARCH=riscv olddefconfig
make ARCH=riscv vmlinux
```

下载编译 BBL (Berkeley Boot Loader)：

```
git clone https://github.com/riscv/riscv-pk.git
cd riscv-pk
mkdir build
cd build
../configure --enable-logo --host=riscv64-unknown-elf --with-payload=../../riscv-linux/vmlinux
make
```

编写 ifup、ifdown 两个 QEMU 模拟器网络脚本（暂时没有理解相关作用）：

```
#!/bin/sh

brctl addif br0 $1
ifconfig $1 up
```

```
#!/bin/sh

ifconfig $1 down
brctl delif br0 $1
```

使用 riscv-qemu 模拟运行 riscv-linux：

```
xieyichen@xieyichen-VirtualBox:~/Documents/riscv$ sudo qemu-system-riscv64   -nographic -machine virt -kernel bbl   -append "root=/dev/vda ro console=ttyS0"   -drive file=riscv.bin,format=raw,id=hd0   -device virtio-blk-device,drive=hd0   -netdev type=tap,script=./ifup,downscript=./ifdown,id=net0   -device virtio-net-device,netdev=net0
```

运行结果：

```
       INSTRUCTION SETS WANT TO BE FREE
[    0.000000] OF: fdt: Ignoring memory range 0x80000000 - 0x80200000
[    0.000000] Linux version 4.14.0-00032-gd10799b22913-dirty (mclark@minty) (gcc version 7.1.1 20170509 (GCC)) #79 Sun Dec 17 10:41:31 NZDT 2017
[    0.000000] Zone ranges:
[    0.000000]   DMA      [mem 0x0000000080200000-0x0000000087ffffff]
[    0.000000]   Normal   empty
[    0.000000] Movable zone start for each node
[    0.000000] Early memory node ranges
[    0.000000]   node   0: [mem 0x0000000080200000-0x0000000087ffffff]
[    0.000000] Initmem setup node 0 [mem 0x0000000080200000-0x0000000087ffffff]
[    0.000000] elf_hwcap is 0x112d
[    0.000000] Built 1 zonelists, mobility grouping on.  Total pages: 31815
[    0.000000] Kernel command line: root=/dev/vda ro console=ttyS0
[    0.000000] PID hash table entries: 512 (order: 0, 4096 bytes)
[    0.000000] Dentry cache hash table entries: 16384 (order: 5, 131072 bytes)
[    0.000000] Inode-cache hash table entries: 8192 (order: 4, 65536 bytes)
[    0.000000] Sorting __ex_table...
[    0.000000] Memory: 122556K/129024K available (2658K kernel code, 216K rwdata, 652K rodata, 96K init, 782K bss, 6468K reserved, 0K cma-reserved)
[    0.000000] SLUB: HWalign=64, Order=0-3, MinObjects=0, CPUs=1, Nodes=1
[    0.000000] NR_IRQS: 0, nr_irqs: 0, preallocated irqs: 0
[    0.000000] riscv,cpu_intc,0: 64 local interrupts mapped
[    0.000000] riscv,plic0,c000000: mapped 10 interrupts to 1/2 handlers
[    0.000000] clocksource: riscv_clocksource: mask: 0xffffffffffffffff max_cycles: 0x24e6a1710, max_idle_ns: 440795202120 ns
[    0.000000] Console: colour dummy device 80x25
[    0.000000] Calibrating delay loop (skipped), value calculated using timer frequency.. 20.00 BogoMIPS (lpj=100000)
[    0.000000] pid_max: default: 32768 minimum: 301
[    0.000000] Mount-cache hash table entries: 512 (order: 0, 4096 bytes)
[    0.000000] Mountpoint-cache hash table entries: 512 (order: 0, 4096 bytes)
[    0.100000] devtmpfs: initialized
[    0.110000] clocksource: jiffies: mask: 0xffffffff max_cycles: 0xffffffff, max_idle_ns: 19112604462750000 ns
[    0.110000] futex hash table entries: 256 (order: 0, 6144 bytes)
[    0.110000] random: get_random_u32 called from bucket_table_alloc+0x80/0x1f6 with crng_init=0
[    0.110000] NET: Registered protocol family 16
[    0.140000] vgaarb: loaded
[    0.150000] clocksource: Switched to clocksource riscv_clocksource
[    0.170000] NET: Registered protocol family 2
[    0.180000] TCP established hash table entries: 1024 (order: 1, 8192 bytes)
[    0.180000] TCP bind hash table entries: 1024 (order: 1, 8192 bytes)
```

```
[    0.180000] TCP: Hash tables configured (established 1024 bind 1024)
[    0.180000] UDP hash table entries: 256 (order: 1, 8192 bytes)
[    0.180000] UDP-Lite hash table entries: 256 (order: 1, 8192 bytes)
[    0.180000] NET: Registered protocol family 1
[    0.190000] workingset: timestamp_bits=62 max_order=15 bucket_order=0
[    0.220000] random: fast init done
[    0.250000] Block layer SCSI generic (bsg) driver version 0.4 loaded (major 254)
[    0.250000] io scheduler noop registered
[    0.250000] io scheduler deadline registered
[    0.250000] io scheduler cfq registered (default)
[    0.250000] io scheduler mq-deadline registered
[    0.250000] io scheduler kyber registered
[    0.360000] Serial: 8250/16550 driver, 4 ports, IRQ sharing disabled
[    0.370000] console [ttyS0] disabled
[    0.380000] 10000000.uart: ttyS0 at MMIO 0x10000000 (irq = 1, base_baud = 230400) is a 16550A
[    0.420000] console [ttyS0] enabled
[    0.490000] NET: Registered protocol family 10
[    0.500000] Segment Routing with IPv6
[    0.500000] sit: IPv6, IPv4 and MPLS over IPv4 tunneling driver
[    0.530000] EXT4-fs (vda): couldn't mount as ext3 due to feature incompatibilities
[    0.540000] EXT4-fs (vda): INFO: recovery required on readonly filesystem
[    0.540000] EXT4-fs (vda): write access will be enabled during recovery
[    0.610000] EXT4-fs (vda): recovery complete
[    0.620000] EXT4-fs (vda): mounted filesystem with ordered data mode. Opts: (null)
[    0.620000] VFS: Mounted root (ext4 filesystem) readonly on device 254:0.
[    0.630000] devtmpfs: mounted
[    0.690000] Freeing unused kernel memory: 96K
[    0.690000] This architecture does not have kernel memory protection.
[    1.040000] EXT4-fs (vda): re-mounted. Opts: data=ordered
ucbvax login: root
Password:
root@ucbvax:~#
```

查看 cpuinfo：

```
root@ucbvax:~# cat /proc/cpuinfo
hart    : 0
isa     : rv64imafdcsu
mmu     : sv48
```

```
root@ucbvax:~# uname -a
Linux ucbvax 4.14.0-00032-gd10799b22913-dirty #79 Sun Dec 17 10:41:31 NZDT 2017 riscv64 GNU/Linux
```

模拟启动成功。

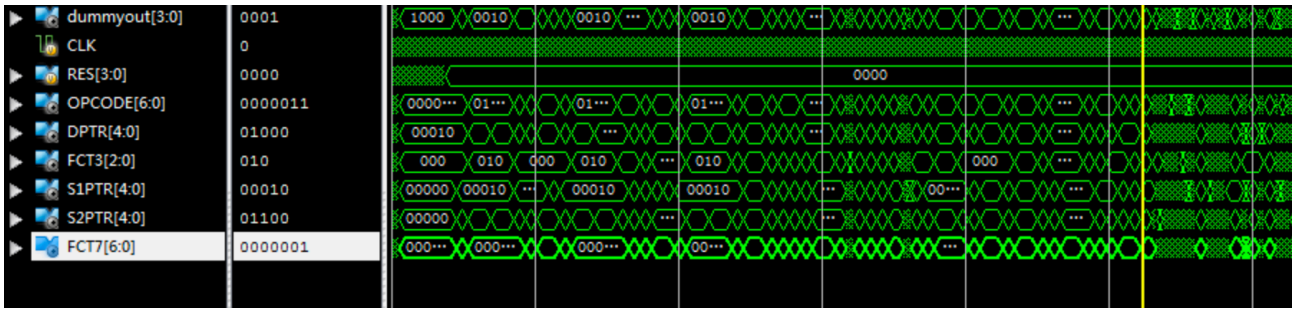
3、riscv-gcc 的使用与 hello world 程序的执行

通过以上步骤安装完成了 riscv 的完整工具链后，即可对手写的 riscv 的 c 语言程序进行编译并生成对应的 riscv 指令集的二进制代码，再将代码例化进 riscv 内核的指令存储器中，即可实现对 c 语言代码的输出。

（三）、结果与分析

USTCPV 最终实现了除 RV32I 指令集中除 CSR 类（状态寄存器类）指令外的所有指令，并经测试可以运行由 RISCV-GCC 编译链编译产生的二进制代码。

由其对指令序列的执行仿真结果如下图：



本次实验中使用最简单的 hello world 作为应用 c 程序，由于未实现 riscv 内核的 IO 指令模块，故无法直接使用函数调用，因此使用将字母写入存储器并将存储器内内容打印到控制台的形式来进行实验。编写的 c 语言程序如下：

```c
void putchar(char c)
{
  volatile int *uart = (int *)0x80000000;

  while(*uart); // uart busy, wait...
  *uart = c;
}

void puts(char *p)
{
  while(*p) putchar(*p++);
  putchar('\r');
  putchar('\n');
}

void main(void)
{
  puts("hello world!");
}
```

注意其中并没有系统预定义的函数，只使用自己的函数即实现了输出 hello world 的结果，对该程序使用 GCC 工具链指令汇编后（实际汇编指令见附录文件中的 src 目录下的 Makefile），即生成二进制代码，将其例化进入存储器后在 ise 中仿真，即得到以下结果：

```
hhhhello world!
```

其中首先输出的 4 个 h 是因为执行时直接输出对应地址的数据，因此由于程序段还未执行完时对应指针不会后移，因此一直指向 h 的位置，故有多余输出的 h。

至此，即完成 ustcpv 的全部测试。

四、实验总结

通过本学期的RISCV实验学习，使得我们更加深入的了解了RISCV指令集的构架及其特点，通过自己实际实现该指令集，才更深地感受到其余之前的risc指令集的区别：简洁。

RISCV指令集具有工整的格式和精确的指令分段，特别是对于opcode和funct3的灵活运用，通过opcode灵活确定指令位数以及通过funct3在不同op下的不同功能，使得同类型的指令不需要通过不同的op译指，因而导致整个CPU的构架都有很大幅度的缩小，在实现同样功能的情况下可能只需要用到原来risc类指令集一半的FPGA资源。

希望在以后的学习工作中可以更多的接触riscv指令集及相关知识，在未来的科研工作中为riscv的开源工程作出自己的一份贡献！

五、参考文献

[1]《Computer Organization and Design RISC-V Edition》
[2]《The RISC-V Instruction Set Manual Volume I: User-Level ISA Document Version 2.1》
[3] riscv/riscv-tools
[4] https://github.com/SI-RISCV/e200_opensource.git