Buceros: 一个简单的 RISC-V SOC

张政镒	李少群	王彬
20300750021	20300750018	20300750019

Github 项目链接

日期：2021 年 12 月 22 日

摘　　要

本文介绍了 Buceros 团队制作的简单的 RISC-V SOC, 以五级流水线的操作, 可以实现 70MHz 以上的时钟频率, 较之于网上的 RISC-V 代码有了更多精进改善之处, 从而更好的节省了资源, 提高了运行速度.

关键词： 中央处理器, RISC-V 指令集, SOC

1　概述

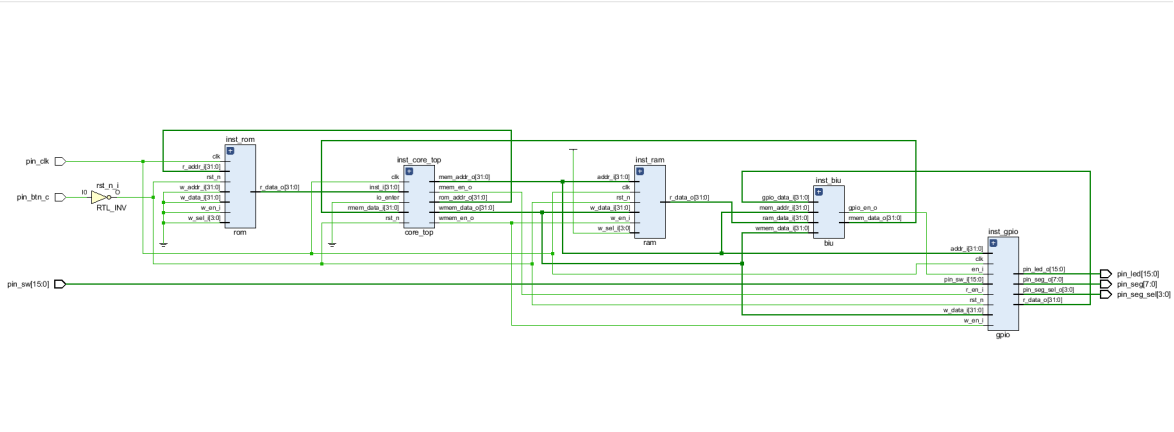
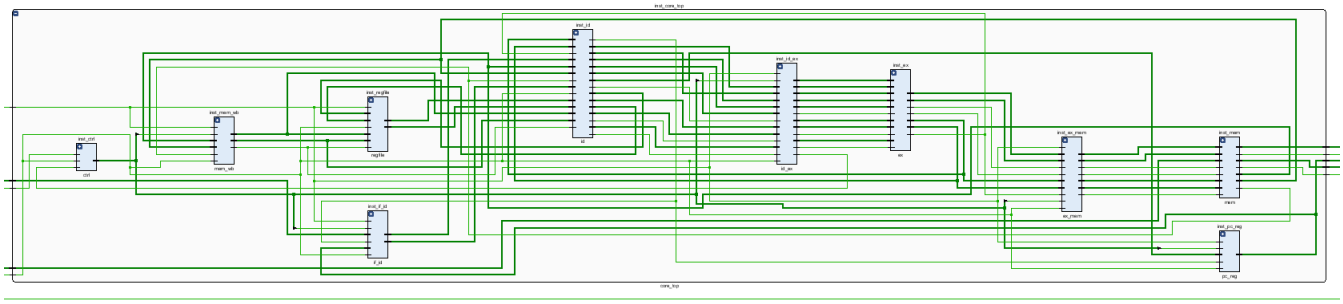


图 1: 总结构图



1.1 core_top 核心模块

本 CPU 按照取指 (IF), 译码 (ID), 执行 (EX), 访存 (MEM), 回写 (WB) 的顺序设计五级流水线, 这五者加上 ctrl 模块及各级寄存器组构成了 core_top 模块, 实现了 CPU 的核心功能, 实现了对 RV32I 指令集的译码、执行 (不包括 ecall 和 ebreak 指令), 实现了本次实验的需求.

1.2 RV32I 指令集

与大多数指令集相比, RISC-V 指令集可以自由地用于任何目的, 允许任何人设计、制造和销售 RISC-V 芯片和软件. 虽然这不是第一个开源指令集, 但它具有重要意义, 因为其设计使其适用于现代计算设备（如仓库规模云计算机、高端移动电话和微小嵌入式系统）. 设计者考虑到了这些用途中的性能与功率效率. 该指令集还具有众多支持的软件, 这解决了新指令集通常的弱点.

1.3 外设模块

1. **bui 模块**作为外设的选择模块, 起到了对 ram 和 gpio 模块的选择作用, 通过接收 mem 模块输入的读/写信号及地址信息, 对不同的外设进行读/写操作, 并将需要的读出来的数据返回给 mem/wb 模块和 id 模块使用
2. **ram 模块**作为读/写数据的主要存储模块, 是整个 CPU 的数据“仓库”
3. **gpio 模块**同 ram 模块一样为一组寄存器组, 包含了对十六个按键开关、七段数码管、四个七段数码管选择开关、十六盏 LED 灯的状态信息, 可以控制上述四类外设是否读取或者写入数据.
4. **rom 模块**功能为存储各类指令的内容, 出于方便更新指令的考虑, 将 rom 模块也写成具有读/写功能的模块, 但在正常运行时其写端口信号始终不使能.

1.4 编译器

本项目采用开源的 GCC 编译器编译 C 语言程序, 并输入到 ROM 中, 从而使得 CPU 的功能设计变得多样化和便利化.

我们使用的 GCC 是由 eclipse 维护的, 使用的版本为 xpack-riscv-none-embed-gcc-10.2.0-1.2.

Core Instruction Formats

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

RV32I Base Integer Instructions

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ^ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0	zero-extends
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm	
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ^ imm	
ori	OR Immediate	I	0010011	0x6		rd = rs1 | imm	
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]	
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]	
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >> imm[0:4]	msb-extends
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0	zero-extends
lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends
sb	Store Byte	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]	
sh	Store Half	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]	
sw	Store Word	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]	
beq	Branch ==	B	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm	
bge	Branch $\geq$	B	1100011	0x5		if(rs1 >= rs2) PC += imm	
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch $\geq$ (U)	B	1100011	0x7		if(rs1 >= rs2) PC += imm	zero-extends
jal	Jump And Link	J	1101111			rd = PC+4; PC += imm	
jalr	Jump And Link Reg	I	1100111	0x0		rd = PC+4; PC = rs1 + imm	
lui	Load Upper Imm	U	0110111			rd = imm << 12	
auipc	Add Upper Imm to PC	U	0010111			rd = PC + (imm << 12)	
ecall	Environment Call	I	1110011	0x0	imm=0x0	Transfer control to OS	
ebreak	Environment Break	I	1110011	0x0	imm=0x1	Transfer control to debugger	

图 2: RV32I 指令集索引

1.5 代码风格

为了保证每个人写的代码可以方便地被所有小组成员阅读，我们详细地制定了一系列的代码风格规范. 包括

1. 输入端口在结尾添加 _i 标识符，输出端口在结尾添加 _o 标识符.
2. 大量使用宏定义，尽量不使用固定的数字，保留对 64 位架构的可拓展性.
3. 宏定义中，固定的数字全部用大写，不同的单词用下划线隔开；位宽定义用大驼峰命名法.
4. 同一组赋值等号、小括号中括号上下对齐，提高代码的美观性和可读性.
5. 封装常用模块，尽量不直接使用时序 always 块，减小了代码密度.
6. 所有的命名和注释都严格规范，不允许使用中文作注释.
7. 同一个模块中，严格按照变量声明、时序逻辑块、组合逻辑块、输出逻辑的顺序编写.
8. 全部使用 Verilog2001 中的端口声明方式.

1.6 项目管理

本次实验全程使用 git 进行代码管理, 并且将改动实时同步到 GitHub 上. 本次 CPU 的代码是完全开源的, 任何人都可以看到我们写的内容, 甚至为我们提交一些更改.

项目地址：https://github.com/0xtaruhi/Buceros

2 CPU 各模块实现方式及亮点特色

2.1 core_top 核心模块

2.1.1 core_top

I/O 端口	端口类型	端口名称	端口说明	端口位宽
input	wire	clk	时钟信号	1
input	wire	rst_n	异步复位信号（低电平使能）	1
input	wire	io_enter	流水线恢复信号	1
input	wire	inst_i	指令寄存器读指令数据端口	‘WordBus
input	wire	rmem_data_i	内存读数据端口	‘WordBus
output	wire	wmem_en_o	内存写使能信号	1
output	wire	rmem_en_o	内存读使能信号	1
output	wire	wmem_data_o	内存或者 gpio 写数据端口	‘WordBus
output	wire	rom_addr_o	指令寄存器读指令地址端口	‘MemAddrBus
output	wire	mem_addr_o	内存访问地址端口	‘MemAddrBus

内部集成了流水线的最核心五级, 为与外设相连提供了便利.

2.1.2 ctrl

I/O 端口	端口类型	端口名称	端口说明	端口位宽
input	wire	rst_n	异步复位信号（低电平使能）	1
input	wire	stallreq_id_ex_i	请求暂停信号（来自 id/ex 模块）	1
input	wire	enter_i	流水线恢复信号	1
output	wire	stall_o	各级流水线暂停端口	[4:0]

若 ex 中正在执行的命令会影响到 id 中正在执行的命令, id/ex 模块会请求暂停其他级流水线使 ex 模块的命令优先完成, 并将需要的数据传输回 id, 等到传输完毕后 id/ex 将拉低请求暂停信号, 使流水线继续运转. enter_i 信号从外部输入使流水线重新运行, 消除未知的暂停错误.

```
    assign stall_o[0] = stallreq_id_ex_i;   // to pc_reg
    assign stall_o[1] = stallreq_id_ex_i;   // to if_id
    assign stall_o[2] = stallreq_id_ex_i;   // to id_ex
    assign stall_o[3] = 1'b0;               // to ex_mem
    assign stall_o[4] = 1'b0;               // to mem_wb
```

2.1.3 pc_reg

I/O 端口	端口类型	端口名称	端口说明	端口位宽
input	wire	clk	时钟信号	1
input	wire	rst_n	异步复位信号（低电平使能）	1
input	wire	imp_en_i	跳转命令使能信号	1
input	wire	jmp_addr_i	跳转命令地址端口	‘InstAddrBus
input	wire	hold_i	暂停命令	1
output	reg	pc_o	命令地址输出端口	‘InstAddrBus

此为命令地址计数器, 也为流水线的**第一级**, 通过对此处的命令地址进行累加或跳转, 从而向 rom 模块请求不同地址的命令.

```
    always @(posedge clk or negedge rst_n) begin
    if(~rst_n) begin
      pc_o <= `PC_RST_ADDR;
        end else if(hold_i) begin
      pc_o <= pc_o;
    end else if(jmp_en_i) begin
      pc_o <= jmp_addr_i;
    end else begin
      pc_o <= pc_o + 32'd4;
    end
  end
```

2.1.4 if_id

I/O 端口	端口类型	端口名称	端口说明	端口位宽
input	wire	clk	时钟信号	1
input	wire	rst_n	异步复位信号（低电平使能）	1
input	wire	hold_i	暂停信号	1
input	wire	jmp_en_i	跳转命令使能信号	1
input	wire	pc_i	命令地址输入端口	‘InstAddrBus
input	wire	inst_i	命令输入端口	‘InstBus
output	reg	pc_o	命令地址输出端口	‘InstAddrBus
output	reg	inst_o	命令输出端口	‘InstBus

此为取指/译码寄存器组, 也为流水线的**第二级**, 将从 ROM 中提取出来的命令预置在其中, 并将命令地址一同传递下去. 值得注意的是, 当复位使能或跳转信号使能时, 将 inst_o 设置为 NOP 指令, 即“addi x0,x0,0”, 通过自增 0 来表示这是一个空指令, 即在跳转指令输入之前先加载一个空指令以实现立即跳转, 不执行其他指令的功能.

2.1.5 id

I/O 端口	端口类型	端口名称	端口说明	端口位宽
input	wire	rst_n	异步复位信号（低电平使能）	1
input	wire	pc_i	命令地址输入端口	‘InstAddrBus
input	wire	inst_i	命令输入端口	‘InstBus
input	wire	ex_wreg_addr_i	执行模块返回数据写地址端口	‘RegAddrBus
input	wire	ex_wreg_en_i	执行模块返回数据写使能信号	1
input	wire	ex_wreg_data_i	执行模块返回数据写数据端口	‘RegDataBus
input	wire	mem_wreg_addr_i	访存模块返回数据写地址端口	‘RegAddrBus
input	wire	mem_wreg_en_i	访存模块返回数据写使能信号	1
input	wire	mem_wreg_data_i	访存模块返回数据写数据端口	‘RegDataBus
input	wire	wb_wreg_addr_i	回写模块返回数据写地址端口	‘RegAddrBus
input	wire	wb_wreg_en_i	回写模块返回数据写使能信号	1
input	wire	wb_wreg_data_i	回写模块返回数据写数据端口	‘RegDataBus
input	wire	rs1_data_i	通用寄存器返回数据一号端口	‘RegDataBus
input	wire	rs2_data_i	通用寄存器返回数据二号端口	‘RegDataBus
output	wire	branch_o	跳转命令使能信号	1
output	wire	pc2pcreg_o	跳转命令地址端口	‘InstAddrBus
output	wire	pc2ex_o	命令地址输出端口	‘InstAddrBus
output	wire	wmem_en_o	访存写使能信号	1
output	wire	rmem_en_o	访存读使能信号	1
output	wire	opcode_o	opcode 输出端口	‘OpcodeBus
output	wire	funct3_o	funct3 输出端口	‘Funct3Bus
output	wire	funct7_o	funct7 输出端口	‘Funct7Bus
output	wire	imm_o	立即数输出端口	‘ImmBus
output	wire	wreg_en_o	通用寄存器组写使能信号	1
output	wire	wreg_addr_o	通用寄存器组写使能地址	‘RegAddrBus
output	wire	rs1_addr_o	通用寄存器组一号地址端口	‘RegAddrBus
output	wire	rs1_data_o	通用寄存器组一号数据端口	‘RegDataBus
output	wire	rs2_addr_o	通用寄存器组二号地址端口	‘RegAddrBus
output	wire	rs2_data_o	通用寄存器组二号数据端口	‘RegDataBus

此模块为五级流水线中**核心**的译码模块, 其将各类命令割分成各个数据或者地址或者使能信号, 并传递给其他不同的模块

亮点一：将从其他模块输入的返回数据写地址端口和信号进行优先级区分, 即 ex>mem>wb>regfile, 从而保证当 id 模块中需要用到的数据正在被 ex、mem、wb 中的一个操作时能够在最短的时间内将当前数据传回 id 模块, 尽管使用三级数据比较器增大了延迟, 但保证了电路工作的可靠性

```
assign rs1_data_o  =
     (ex_wreg_en_i && ex_wreg_addr_i == rs1_addr_o)  ? ex_wreg_data_i  :
     (mem_wreg_en_i && mem_wreg_addr_i == rs1_addr_o) ? mem_wreg_data_i :
     (wb_wreg_en_i && wb_wreg_addr_i == rs1_addr_o)   ? wb_wreg_data_i  :
     rs1_data_i;
```

```
assign rs2_data_o  =
     (ex_wreg_en_i && ex_wreg_addr_i == rs2_addr_o)   ? ex_wreg_data_i  :
     (mem_wreg_en_i && mem_wreg_addr_i == rs2_addr_o) ? mem_wreg_data_i :
     (wb_wreg_en_i && wb_wreg_addr_i == rs2_addr_o)   ? wb_wreg_data_i  :
     rs2_data_i;
```

亮点二：其他所有的数据选择或者数据分割采用与或选择方式, 大幅度减少了因 if、case 等逐步选择带来的高延时和高资源使用量.

```
//例1
assign branch_o = inst_branch & ( ~funct3_o[2] & ~funct3_o[1] & (funct3_o[0] ^
    rs1_eq_rs2) |
funct3_o[2] & ~funct3_o[1] & (funct3_o[0] ^ rs1_lt_rs2) |
funct3_o[2] &  funct3_o[1] & (funct3_o[0] ^ rs1_ltu_rs2)) | inst_jal | inst_jalr;
```

```
//例2
assign imm_o = {`IMM_W{inst_type_I}} & {{20{inst_i[31]}}, inst_i[31:20]} |
  {`IMM_W{inst_type_S}} & {{20{inst_i[31]}}, inst_i[31:25], inst_i[11:7]} |
  {`IMM_W{inst_type_B}} & {{20{inst_i[31]}}, inst_i[7], inst_i[30:25],
  inst_i[11:8], 1'b0} |{`IMM_W{inst_type_U}} & {{12{inst_i[31]}},
  inst_i[31:12]} |{`IMM_W{inst_type_J}} & {{12{inst_i[31]}}, inst_i[19:12],
   inst_i[20], inst_i[30:21], 1'b0};
```

注意：只有我们通用寄存器组一号地址端口的访问地址与 ex、mem、wb 输出地址相匹配时, id 才会提前读取 ex、mem、wb 这三个端口中的数据 **亮点三**：我们尽可能地少利用 LUT 资源，例如我们在译码阶段只使用了一个 31bit 的比较器就实现了无符号数比较和有符号数比较.

```
assign rs_diff_signed = rs1_data_o[`REG_DATA_W-1] ^ rs2_data_o[`REG_DATA_W-1];
assign rs1_ltu_rs2_exclude_msb = rs1_data_o[`REG_DATA_W-2:0] < rs2_data_o[
    `REG_DATA_W-2:0];
assign rs1_ltu_rs2 = rs_diff_signed ? rs2_data_o[`REG_DATA_W-1] :
    rs1_ltu_rs2_exclude_msb;
assign rs1_lt_rs2  = rs_diff_signed ? rs1_data_o[`REG_DATA_W-1] :
    rs1_ltu_rs2_exclude_msb;
```

2.1.6 id_ex

I/O 端口	端口类型	端口名称	端口说明	端口位宽
input	wire	clk	时钟信号	1
input	wire	rst_n	异步复位信号（低电平使能）	1
input	wire	hold_i	暂停信号	1
input	wire	id_pc_i	命令地址输入端口	‘InstAddrBus
input	wire	id_wmem_en_i	访存写使能信号	1
input	wire	id_rmem_en_i	访存读使能信号	1
input	wire	id_opcode_i	opcode 输入端口	‘OpcodeBus
input	wire	id_funct3_i	funct3 输入端口	‘Funct3Bus
input	wire	id_funct7_i	funct7 输入端口	‘Funct7Bus
input	wire	id_imm_i	立即数输入端口	‘ImmBus
input	wire	id_wreg_addr_i	通用寄存器数据写地址端口	‘RegAddrBus
input	wire	id_wreg_en_i	通用寄存器数据写使能信号	1
input	wire	id_rs1_data_i	通用寄存器返回数据一号端口	‘RegDataBus
input	wire	id_rs2_data_i	通用寄存器返回数据二号端口	‘RegDataBus
output	wire	ex_pc_o	命令地址输入端口	‘InstAddrBus
output	wire	ex_wmem_en_o	访存写使能信号	1
output	wire	ex_rmem_en_o	访存读使能信号	1
output	wire	ex_opcode_o	opcode 输出端口	‘OpcodeBus
output	wire	ex_funct3_o	funct3 输出端口	‘Funct3Bus
output	wire	ex_funct7_o	funct7 输出端口	‘Funct7Bus
output	wire	ex_imm_o	立即数输出端口	‘ImmBus
output	wire	ex_wreg_en_o	通用寄存器组写使能信号	1
output	wire	ex_wreg_addr_o	通用寄存器组写使能地址	‘RegAddrBus
output	wire	rs1_data_o	通用寄存器组一号数据端口	‘RegDataBus
output	wire	rs2_data_o	通用寄存器组二号数据端口	‘RegDataBus
output	wire	stallreq_id_ex_o	暂停请求信号	1

本模块为译码/执行模块，也为流水线的**第三级**. 本模块为时序电路，通过统一的时钟控制 id 信号或端口向 ex 传递的时序. 注意：此处 stallreq_id_ex_o 为暂停请求信号，每当 id 模块需要读取内存中数据时，将会从此处发送请求暂停命令，以帮助 id 能够读到最新状态的数据.

```
    //例2
always @(posedge clk or negedge rst_n) begin
  if(~rst_n) begin
    stallreq_id_ex_r <= 1'b0;
  end else if(id_rmem_en_i) begin
    stallreq_id_ex_r <= ~stallreq_id_ex_r;
  end else begin
    stallreq_id_ex_r <= 1'b0;
  end
end
```

2.1.7 ex

I/O 端口	端口类型	端口名称	端口说明	端口位宽
input	wire	pc_i	命令地址输入端口	‘InstAddrBus
input	wire	wmem_en_i	访存写使能信号	1
input	wire	rmem_en_i	访存读使能信号	1
input	wire	opcode_i	opcode 输入端口	‘OpcodeBus
input	wire	funct3_i	funct3 输入端口	‘Funct3Bus
input	wire	funct7_i	funct7 输入端口	‘Funct7Bus
input	wire	imm_i	立即数输入端口	‘ImmBus
input	wire	wreg_addr_i	通用寄存器数据写地址端口	‘RegAddrBus
input	wire	wreg_en_i	通用寄存器数据写使能信号	1
input	wire	rs1_data_i	通用寄存器返回数据一号端口	‘RegDataBus
input	wire	rs2_data_i	通用寄存器返回数据二号端口	‘RegDataBus
output	wire	wmem_en_o	访存写使能信号	1
output	wire	rmem_en_o	访存读使能信号	1
output	wire	mem_addr_o	访存地址端口	‘MemAddrBus
output	wire	funct3_o	funct3 输出端口	‘Funct3Bus
output	wire	wreg_en_o	通用寄存器组写使能信号	1
output	wire	wreg_addr_o	通用寄存器组写地址端口	‘RegAddrBus
output	wire	wreg_data_o	通用寄存器组写数据端口	‘RegDataBus

本模块为五级流水线中核心的执行模块, 通过对不同指令做对应的基础操作, 从而将计算得到的结果输出到对应的内存中或者转存到通用寄存器组中. **注意一：**经过本次操作后, funct7 不再存在利用价值, 而 funct3 还将决定之后存储到内存中的方式, 包括字节加载、半字加载、字加载、字节加载、半字加载等操作. **注意二：**内存的访问地址由 rs1_data_i 和 imm_i 相加得到, 其中 rs1_data_i 可以由 rom 中的指令用立即数左移直接加载得到（LUI 指令）, 而 imm_i 可以提前由 rom 中的指令预置好, 因此, 每一步内存的访问地址都已经由 rom 中指令决定好了.

```
assign mem_addr_o   = rs1_data_i + imm_i;
```

2.1.8 ex_mem

I/O 端口	端口类型	端口名称	端口说明	端口位宽
input	wire	clk	时钟信号	1
input	wire	rst_n	异步复位信号（低电平使能）	1
input	wire	hold_i	暂停信号	1
input	wire	ex_wmem_en_i	访存写使能信号	1
input	wire	ex_rmem_en_i	访存读使能信号	1
input	wire	ex_mem_addr_i	访存地址端口	‘MemAddrBus
input	wire	ex_funct3_i	funct3 输入端口	‘Funct3Bus
input	wire	ex_wreg_addr_i	通用寄存器数据写地址端口	‘RegAddrBus
input	wire	ex_wreg_en_i	通用寄存器数据写使能信号	1
input	wire	ex_wreg_data_i	通用寄存器数据写数据端口	‘RegDataBus
output	wire	mem_wmem_en_o	访存写使能信号	1
output	wire	mem_rmem_en_o	访存读使能信号	1
output	wire	mem_mem_addr_o	访存地址端口	‘MemAddrBus
output	wire	mem_funct3_o	funct3 输出端口	‘Funct3Bus
output	wire	mem_wreg_en_o	通用寄存器组写使能信号	1
output	wire	mem_wreg_addr_o	通用寄存器组写地址端口	‘RegAddrBus
output	wire	mem_wreg_data_o	通用寄存器组写数据端口	‘RegDataBus

本模块为五级流水线中的 ex/mem 模块, 也是流水线中的**第四级**. 起到了缓存数据, 统一时序的功能. **注意：**各级的寄存器组采用 gen_dffr 和 gen_dffsr 来模仿 D 触发器组功能.

```
gen_dffr #(.WIDTH(       1'b1)) dff_wmem_en  (clk, rst_n, hold_i, ex_wmem_en_i,
    mem_wmem_en_o);
gen_dffr #(.WIDTH(       1'b1)) dff_rmem_en  (clk, rst_n, hold_i, ex_rmem_en_i,
    mem_rmem_en_o);
gen_dffr #(.WIDTH(`MEM_ADDR_W)) dff_mem_addr (clk, rst_n, hold_i, ex_mem_addr_i,
    mem_mem_addr_o);
gen_dffr #(.WIDTH(  `FUNCT3_W)) dff_funct3   (clk, rst_n, hold_i, ex_funct3_i,
    mem_funct3_o);
gen_dffr #(.WIDTH(       1'b1)) dff_wreg_en  (clk, rst_n, hold_i, ex_wreg_en_i,
    mem_wreg_en_o);
gen_dffr #(.WIDTH(`REG_ADDR_W)) dff_wreg_addr(clk, rst_n, hold_i, ex_wreg_addr_i,
    mem_wreg_addr_o);
gen_dffr #(.WIDTH(`REG_DATA_W)) dff_wreg_data(clk, rst_n, hold_i, ex_wreg_data_i,
    mem_wreg_data_o);
```

2.1.9 mem

I/O 端口	端口类型	端口名称	端口说明	端口位宽
input	wire	wmem_en_i	访存写使能信号	1
input	wire	rmem_en_i	访存读使能信号	1
input	wire	mem_addr_i	访存地址端口	‘MemAddrBus
input	wire	rmem_data_i	来自内存的读取数据端口	‘MemDataBus
input	wire	funct3_i	funct3 输入端口	‘Funct3Bus
input	wire	wreg_addr_i	通用寄存器数据写地址端口	‘RegAddrBus
input	wire	wreg_en_i	通用寄存器数据写使能信号	1
input	wire	wreg_data_i	通用寄存器/内存数据写数据端口	‘RegDataBus
output	wire	wmem_en_o	访存写使能信号	1
output	wire	rmem_en_o	访存读使能信号	1
output	wire	mem_addr_o	访存地址端口	‘MemAddrBus
output	wire	wmem_data_o	写内存数据端口	‘WordBus
output	wire	wreg_en_o	通用寄存器组写使能信号	1
output	wire	wreg_addr_o	通用寄存器组写地址端口	‘RegAddrBus
output	wire	wreg_data_o	通用寄存器组写数据端口	‘RegDataBus

本模块为五级流水线中的核心模块, 即访存模块. wreg_data_o 既可以输入到通用寄存器组中, 也可以输入到内存当中, 其选择由各个使能信号决定. 而 wreg_data_o 与 wmem_data_o 由 wreg_data_i 根据 funct3_i 与使能信号选择性加载得到, 实现了指令的功能. 注意：此处对写入的数据进行字节加载、半字加载、字加载、字节加载、半字加载等操作.

```
always @ (*) begin
  if (wmem_en_i) begin
    case (funct3_i)
      `INST_BYTE: begin
        wmem_data_r = {{(`ROM_WIDTH - `BYTE_WIDTH){1'b0}},wreg_data_i[`BYTE_WIDTH -
            1:0]};
      end
      `INST_HALF_WORD: begin
        wmem_data_r = {{(`ROM_WIDTH - `HALF_WORD_WIDTH){1'b0}},wreg_data_i[
            `HALF_WORD_WIDTH - 1:0]};
      end
      `INST_WORD: begin
        wmem_data_r = wreg_data_i; // not extended to ARCH_RV64, so the code has
            been simplified
      end
      default: begin
        wmem_data_r = `ZERO_WORD;
      end
    endcase
    wreg_data_r = `ZERO_WORD; // when store, do nothing with wreg_data
  end else if (rmem_en_i) begin
    wmem_data_r = `ZERO_WORD;
```

```
    case (funct3_i)
      `INST_BYTE: begin
        wreg_data_r = {{(`ROM_WIDTH - `BYTE_WIDTH){rmem_data_i[`BYTE_WIDTH - 1]}},
            rmem_data_i[`BYTE_WIDTH - 1:0]};
      end
      `INST_HALF_WORD: begin
        wreg_data_r = {{(`ROM_WIDTH - `HALF_WORD_WIDTH){rmem_data_i[
            `HALF_WORD_WIDTH - 1]}},rmem_data_i[`HALF_WORD_WIDTH - 1]};
      end
      `INST_WORD: begin
        wreg_data_r = rmem_data_i; // not extended to ARCH_RV64, so the code has
            been simplified
      end
      `INST_BYTE_U: begin
        wreg_data_r = {{(`ROM_WIDTH - `BYTE_WIDTH){1'b0}},rmem_data_i[`BYTE_WIDTH -
             1:0]};
      end
      `INST_HALF_WORD_U: begin
        wreg_data_r = {{(`ROM_WIDTH - `HALF_WORD_WIDTH){1'b0}},rmem_data_i[
            `HALF_WORD_WIDTH - 1]};
      end
      default: begin
        wreg_data_r = `ZERO_WORD;
      end
      endcase
    end else begin
      wmem_data_r = `ZERO_WORD;
      wreg_data_r = wreg_data_i;
    end
  end
```

2.1.10 mem_wb

I/O 端口	端口类型	端口名称	端口说明	端口位宽
input	wire	clk	时钟信号	1
input	wire	rst_n	异步复位信号（低电平使能）	1
input	wire	hold_i	暂停信号	1
input	wire	mem_wreg_addr_i	通用寄存器数据写地址端口	‘RegAddrBus
input	wire	mem_wreg_en_i	通用寄存器数据写使能信号	1
input	wire	mem_wreg_data_i	通用寄存器写数据端口	‘RegDataBus
output	wire	wb_wreg_en_o	通用寄存器组写使能信号	1
output	wire	wb_wreg_addr_o	通用寄存器组写地址端口	‘RegAddrBus
output	wire	wb_wreg_data_o	通用寄存器组写数据端口	‘RegDataBus

本模块为五级流水线中的 mem_wb 模块, 也是流水线的**最后一级**. 其作为通用寄存器组, 输出既可以发送到 regfile 并写入数据和地址, 也可以发送到 id 模块直接提供数据和地址. 使得整个流水线成为一个可反

寄存器编号	寄存器别名	描述	保存者
x0	zero	硬编码 0	/
x1	ra	返回地址	调用者
x2	sp	栈指针	被调用者
x3	gp	全局指针	/
x4	tp	线程指针	/
x5-7	t0-2	临时变量	调用者
x8	s0/sp	保存寄存器/栈指针	被调用者
x9	s1	保存寄存器	被调用者
x10-11	a0-1	函数入参/返回值	调用者
x12-17	a2-7	函数入参	调用者
x18-27	s2-11	保存寄存器	被调用者
x28-31	t3-6	临时变量	调用者

表 1: 通用寄存器相关描述

馈的循环网络.

2.1.11 regfile

I/O 端口	端口类型	端口名称	端口说明	端口位宽
input	wire	clk	时钟信号	1
input	wire	rst_n	异步复位信号（低电平使能）	1
input	wire	r_addr1_i	通用寄存器返回地址一号端口	‘RegAddrBus
input	wire	r_addr2_i	通用寄存器返回地址二号端口	‘RegAddrBus
input	wire	w_en_i	通用寄存器写使能信号	1
input	wire	w_addr_i	通用寄存器写地址端口	‘RegAddrBus
input	wire	w_data_i	通用寄存器写数据端口	‘RegDataBus
output	wire	r_data1_o	通用寄存器输出数据一号端口	‘RegDataBus
output	wire	r_data2_o	通用寄存器输出数据二号端口	‘RegDataBus

本模块为五级流水线的数据“数据中转站”__ 通用寄存器组. 在此处加载诸多临时存放的数据, 以供 id 模块使用. 共有 32 个字长为 32 位的通用寄存器（x0 x31）, 其中 x0 寄存器中数据恒为 0. 对于数据的读取部分, 采用组合逻辑描述, 在最短的时间内读取到对应寄存器的数据. 对于数据的写入部分, 采用同步时序电路描述, 类似于 rom、ram、gpio 的操作. 具体的用途可见下页图 3.

```
assign r_data1_o = {`REG_DATA_W{~r1_x0}} & ((w_en_i && (r_addr1_i == w_addr_i))
 ? w_data_i : regfile[r_addr1_i]);
assign r_data2_o = {`REG_DATA_W{~r2_x0}} & ((w_en_i && (r_addr2_i == w_addr_i))
 ? w_data_i : regfile[r_addr2_i]);
```

2.1.12 gen_dffr / gen_dffsr

I/O 端口	端口类型	端口名称	端口说明	端口位宽
input	wire	clk	时钟信号	1
input	wire	rst_n	异步复位信号（低电平使能）	1
input	wire	hold_i	暂停信号	1
input	wire	data_i	输入数据端口	[WIDTH_1:0]
output	wire	data_o	输出数据端口	[WIDTH_1:0]

本模块为 util（工具）类模块, 在此处起到类 D 触发器的功能.

2.2 外设模块

2.2.1 biu

I/O 端口	端口类型	端口名称	端口说明	端口位宽
input	wire	mem_addr_i	内存访问地址端口	‘MemAddrBus
input	wire	wmem_data_i	内存写数据输入端口	‘WordBus
input	wire	ram_data_i	ram 读取数据输入端口	‘MemDataBus
input	wire	gpio_data_i	gpio 读取数据输入端口	‘MemDataBus
output	wire	gpio_en_o	gpio 选择使能信号	1
output	wire	rmem_data_o	内存读数据输出端口	‘MemDataBus

本模块是外设模块中的**核心 biu 模块**, 为组合逻辑电路, 该模块可以对外设（ram, gpio）起到选择作用. 根据输入的内存访问地址端口, 可以使 ram_en_o 或者 gpio_en_o 使能, 选择对应的外设. **写操作：**写数据及写地址都已经由 mem 提供到各个外设模块. **读操作：**对应的选择信号使能后, 将对应的外设输入的读数据通过 biu 输出回 core_top 模块.

2.2.2 gpio

I/O 端口	端口类型	端口名称	端口说明	端口位宽
input	wire	clk	时钟信号	1
input	wire	rst_n	异步复位信号（低电平使能）	1
input	wire	en_i	使能选择信号	1
input	wire	r_en_i	读使能信号	1
input	wire	addr_i	访问地址输入端口	‘MemAddrBus
input	wire	w_en_i	写使能信号	1
input	wire	w_data_i	写数据输入端口	‘MemDataBus
input	wire	pin_sw_i	外部输入开关信号	[15:0]
output	wire	r_data_o	读数据输出端口	‘MemDataBus
output	wire	pin_seg_o	七段数码管显示输出端口	7:0
output	wire	pin_seg_sel_o	七段数码管选通输出端口	3:0
output	wire	pin_led_o	流水灯显示输出端口	15:0

本模块为外设部分中的 **GPIO**（General_purpose input/output）模块, 其中, 按键开关信息由外部直接读

入, 无法写入, 但七段数码管的显示、选通以及流水灯可以实现写入和读取. 选择不同的外设, 从而使得能够将计算的结果输出显示.

2.2.3 ram

I/O 端口	端口类型	端口名称	端口说明	端口位宽
input	wire	clk	时钟信号	1
input	wire	rst_n	异步复位信号（低电平使能）	1
input	wire	addr_i	访问地址输入端口	‘MemAddrBus
input	wire	w_en_i	写使能信号	1
input	wire	w_data_i	写数据输入端口	‘WordBus
input	wire	w_sel_i	字节选择信号	[3:0]
output	wire	r_data_o	读数据输出端口	‘WordBus

此为外设部分中的 **ram** 模块, 每一次读写操作对四个字节（即一个字）去操作. 亮点：可对一个字中的任意几个字节进行写操作, 从而实现对内存中单个或多个字节的修改, 但同时读取时以字为单位进行读取.

2.2.4 rom

I/O 端口	端口类型	端口名称	端口说明	端口位宽
input	wire	clk	时钟信号	1
input	wire	rst_n	异步复位信号（低电平使能）	1
input	wire	r_addr_i	访问地址输入端口	‘MemAddrBus
input	wire	w_en_i	写使能信号	1
input	wire	w_addr_i	访问地址输入端口	‘MemAddrBus
input	wire	w_data_i	写数据输入端口	‘WordBus
input	wire	w_sel_i	字节选择信号	[3:0]
output	wire	r_data_o	读数据输出端口	‘WordBus

此为外设部分中的 **rom** 模块, 每一次读写操作对四个字节（即一个字）去操作. 注意：此处关于 rom 的所有写端口在正常工作时全部置于低电平, 内部存储的为指令信息, 借助

```
initial begin
$readmemh("E:/honorwork/CPU/Buceros/tests/examples/led.mem", _rom);
end
```

代码读入对应的汇编代码.

2.2.5 地址分配

3 编译器

3.1 Makefile 文件

我们利用了 make 这一工具进行编译. 编译完成后会产生.bin .dump .mem .o 四个文件.

可用地址空间	实际占用地址空间	设备	类型	容量
0x0000_0000 --- 0x1fff_ffff	0x0000_0000 --- 0x0000_07ff	指令存储器 (ROM)	只读 *	1KB
0x2000_0000 --- 0x3fff_ffff	0x2000_0000 --- 0x2000_07ff	数据存储器 (RAM)	读/写	1KB
0x4000_0000 --- 0x5fff_ffff	0x4000_0000 --- 0x4000_0001	GPIO(LED)	读/写	2Byte
0x4000_0000 --- 0x5fff_ffff	0x4000_0004 --- 0x4000_0005	GPIO(七段数码管显示)	读/写	2Byte
0x4000_0000 --- 0x5fff_ffff	0x4000_0008 --- 0x4000_0008	GPIO(七段数码管选通)	读/写	1Byte
0x4000_0000 --- 0x5fff_ffff	0x5000_0000 --- 0x5000_0001	GPIO(开关)	只读	2Byte

3.2 链接文件

链接文件中指定了程序各个部分所在的位置. 在链接文件的一开头就有如下两句：

```
MEMORY
{
    rom(wxa!ri) : ORIGIN = 0x00000000, LENGTH = 1K
    ram(wxa!ri) : ORIGIN = 0x20000000, LENGTH = 1K
}
```

这就指明了程序可用的 rom 和 ram 空间. 我们采用的是哈佛结构（在 RISCV/ARM 嵌入式结构中非常常见），随后即是各个 section 位置的说明. 特别要说明的是“栈空间”，在 lds 文件中时这么描述的：

```
.stack ORIGIN(ram) + LENGTH(ram) - __stack_size :
{
    PROVIDE( _heap_end = . );
    . = __stack_size;
    PROVIDE( _sp = . );
    __freertos_irq_stack_top = .;
} >ram AT>ram
```

栈是负增长的，所以需要把初始栈指针 _sp 设置在栈顶. 在入口汇编程序中需要用到这个值.

```
    la sp, _sp
```

3.3 从 bin 转到 mem

由于编译出来的文件是 bin 格式的二进制文件，我们使用了 Python 程序将 bin 转换为 mem.
程序如下

```
import sys
import os

fileloc = os.path.join(os.getcwd(), sys.argv[1])

desfileloc = fileloc.replace('bin', 'mem')

fp = open(fileloc, 'rb')
des_fp = open(desfileloc, 'w')
size = os.path.getsize(fileloc)
```

```
for i in range((int)(size/4)):
    word_info = [0,0,0,0]
    for j in range(4):
        data_temp = fp.read(1)
        word_info[3-j] = ''.join('%02X' %x for x in data_temp)
    des_fp.write(''.join(word_info))
    des_fp.write('\n')

fp.close()
des_fp.close()
```

3.4 编译

编译非常简单，在配置好环境之后，cd 到程序所在的文件夹 (必须有 Makefile 文件)，然后输入

```
make
```

就会自动在文件夹里生成一个 mem 文件.

4 测试实验

5 我们解决的一些 bug

5.1 BUG1——运算符优先级

错误代码

```
assign result_sum   = rs1_data_i + (exe_sub ^ data2 + exe_sub);
```

正确代码

```
assign result_sum   = rs1_data_i + (({`REG_DATA_W{exe_sub}} ^ data2) + exe_sub);
```

5.2 BUG2——流水线吞指令

错误代码

```
assign stall_o[0] = stallreq_id_ex_i;   // to pc_reg
assign stall_o[1] = stallreq_id_ex_i;   // to if_id
assign stall_o[2] = stallreq_id_ex_i;   // to id_ex
assign stall_o[3] = 1'b0;               // to ex_mem
assign stall_o[4] = stallreq_id_ex_i;
```

正确代码

```
assign stall_o[0] = stallreq_id_ex_i;   // to pc_reg
assign stall_o[1] = stallreq_id_ex_i;   // to if_id
assign stall_o[2] = stallreq_id_ex_i;   // to id_ex
assign stall_o[3] = 1'b0;               // to ex_mem
assign stall_o[4] = 1'b0;
```

5.3 BUG3——PC 寄存器的 hold 与 jump

错误代码

```
always @(posedge clk or negedge rst_n) begin
    if(~rst_n) begin
        pc_o <= `PC_RST_ADDR;
    end else if(jmp_en_i) begin
        pc_o <= jmp_addr_i;
    end else if(hold_i) begin
        pc_o <= pc_o;
    end else begin
        pc_o <= pc_o + 32'd4;
    end
end
```

正确代码

```
always @(posedge clk or negedge rst_n) begin
    if(~rst_n) begin
        pc_o <= `PC_RST_ADDR;
    end else if(hold_i) begin
        pc_o <= pc_o;
    end else if(jmp_en_i) begin
        pc_o <= jmp_addr_i;
    end else begin
        pc_o <= pc_o + 32'd4;
    end
end
```

6 一些问题和自己的思考

6.1 为什么我们不做 VGA 显示?

VGA 显示需要使用自己的频率. 一方面我们不希望把 VGA 模块用 Verilog 写死，而是希望通过程序的方式进行实现，否则就违背了做一个“CPU”的初衷，之后在调节主频的时候也会出现问题. 我们希望 CPU 的频率是灵活可调节的.

7 不足及总结

7.1 架构设计失误

我们将所有的读取做成了组合逻辑，后续证明我们的频率之所以上不去，就是因为 mem 的读取延迟过大. 我们当时应该采用时序逻辑的做法，这样可以显著地减少建立时间，大大提高我们 CPU 的主频.

同时将 ram 和 rom 做成时序逻辑还可以调用板上的 bram 资源. 我们当时开始时预想的 ram 和 rom 各有 64K 的大小，但是最后只实现了各 1K 的大小. 一方面，由于不能使用 bram 资源，所有的存储都是用 lut 资源实现的，这使得综合非常缓慢. 另一方面，及时只有 1K 的大小，也占用了 18% 的 lut 资源，相比之下我们的 cpu 只用了 4% 的 lut 资源.