

CPU Report

Author: Caster from the emerging engineering class of Fudan University

Risc CPU实验报告

CPU概述部分

本实验采取MIPS32位指令集，通过建立五级流水线的方式实现CPU，指令集包括：算术、逻辑、移动、分支、跳转、移位、加载、储存、异常指令，五级流水线具体而言为：取指、译码、执行、访存、回写五个模块，在五个模块的基础上为处理数据冲突的问题，在访存和回写阶段设有数据转发连线，能够迅速转发数据到译码或执行阶段；为处理加载指令lw等与分支指令beq等的数据冲突，设有控制模块ctrl进行流水线暂停；为处理异常相关指令，设有协处理器cp0模块；为与FPGA开发板外设通信设有IO模块；为储存指令与数据设有ROM和RAM模块（由于板载限制，RAM至多可设置为512个32位地址）；为处理FPGA按键输入抖动设有debounce消抖模块。

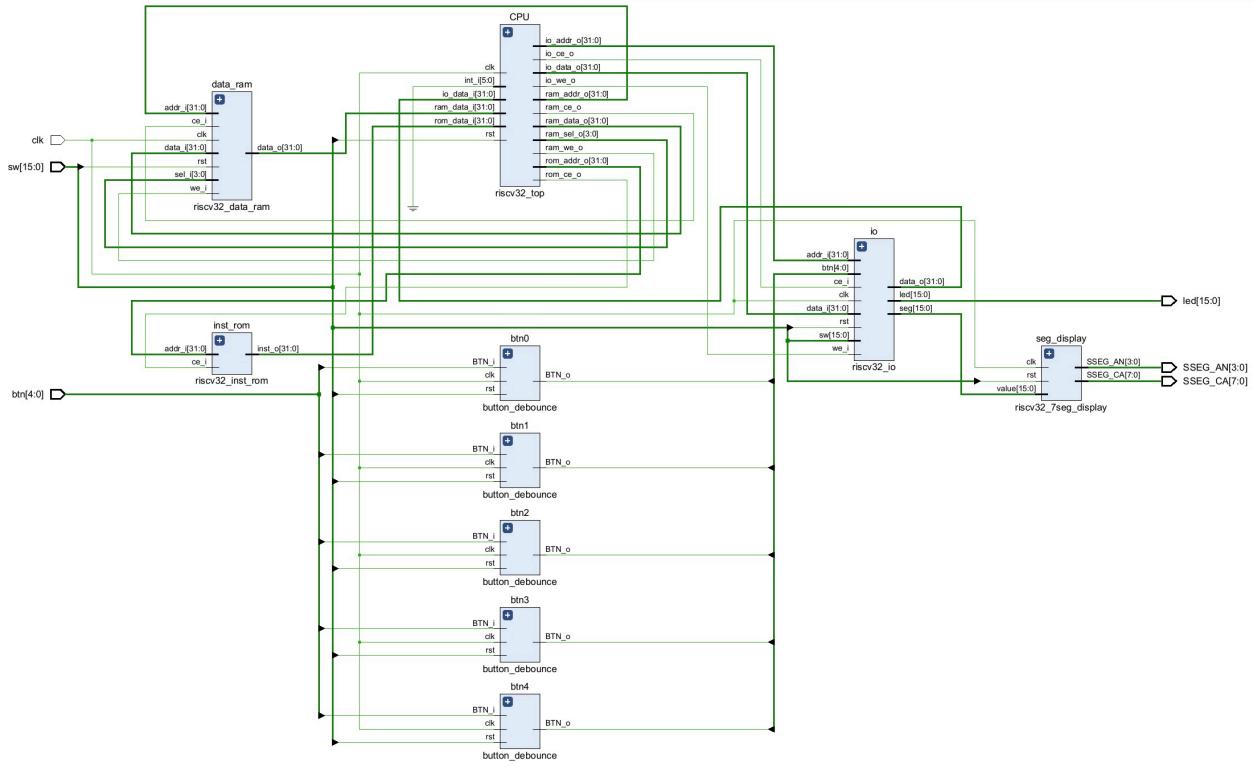
该32位Risc能够在100MHz频率的开发板上顺利完成任务，根据Vivado的时序报告，最坏情况下的时序延迟总和不到9ns，因此最大频率在100MHz之上，为测试CPU，本实验主要通过汇编代码设计输出1-n素数与快速排序的方法进行展示，展示方式为通过开关和按键进行输入，通过LED灯和数码管进行输出。

顶层设计部分

本CPU顶层部分可通过FPGA内置的100MHz时钟作为信号输入，通过FPGA开发板上的开关和按键作为输入，数码管显示和LED灯显示作为输出，具体I/O端口列表说明如下表所示：

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	sw	开关输入数据 Switch	15:0
input	wire	btn	按键输入数据 Button	4:0
output	wire	led	LED 灯显示输出	15:0
output	wire	SSEG_AN	数码管扫描信号	4:0
output	wire	SSEG_CA	数码管显示数据（低电平有效）	7:0

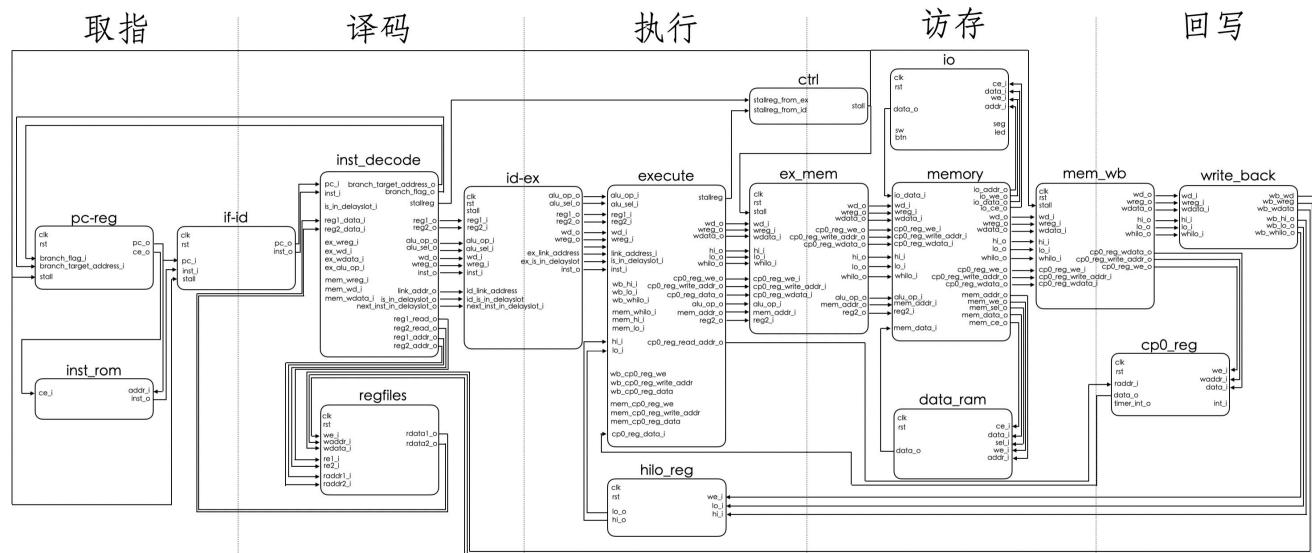
具体而言，顶层模块包括以下部分：ROM模块 `inst_rom` 用于储存输入CPU的指令，RAM模块 `data_ram` 用于储存数据，IO模块 `io` 用于与FPGA开发板上的输入输出进行关联，`debounce`模块用于进行按键消抖，`display`模块 `seg_display` 用于将接收到的数据显示在数码管上，CPU模块用于处理从ROM读入的指令和从RAM（包括io模块）读入的数据，经过CPU处理后将结果写回RAM（包括io模块），具体设计图如下所示：



CPU内部子模块设计部分

在CPU内部采用五级流水线的设计方式，分为：取指、译码、执行、访存、回写五个阶段，在每个阶段的主要模块分别为 pc_reg、inst_decode、execute、memoery、

write_back，在相邻阶段中通过 if_id、id_ex、ex_mem、mem_wb 进行连接，在连接模块主要采取同步时序电路的设计方法，在主要模块主要采取组合电路的设计方法，复位主要发生在连接模块中。在取指阶段，pc_reg 可与ROM进行通信，从ROM中取出指令传递给下一级；在译码阶段，inst_decode 可与通用寄存器堆 regfiles 通信，根据得到的指令从中读出通用寄存器的值；在执行阶段 execute 可以从乘除法特殊寄存器 hilo 中读出高位hi和低位lo 寄存器的值；在访存阶段，memory 可以从RAM和IO模块中读入数据，也可以将数据写入RAM 和IO模块中；在回写模块 write_back 将数据可以写会通用寄存器堆 regfiles 中；在每一个阶段 ctrl 可以通过发出流水线暂停指令 stall 暂停流水线，也可以从 inst_decode 和 execute 中接受流水线暂停信号，具体流程图如下：



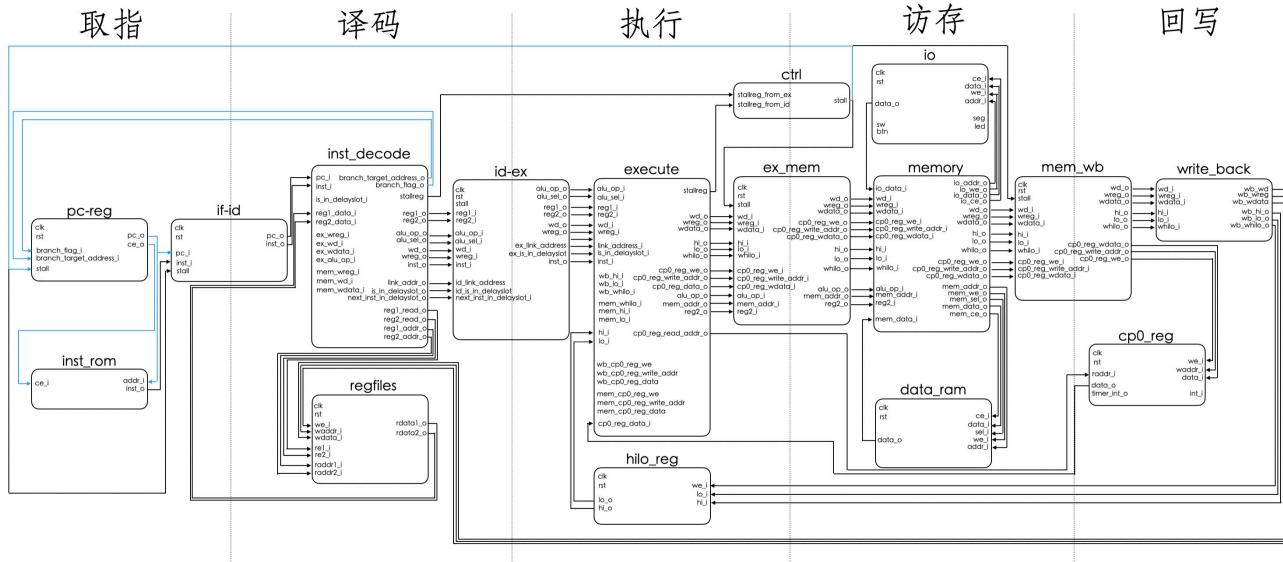
取指部分

(1)程序计数器

在程序计数器 pc_reg 模块，通过接受时钟信号clk、复位信号rst、流水线暂停信号对模块的运行进行控制，接受分支跳转信号和分支跳转指令地址控制程序计数器的值实现跳转，输出程序计数器的值和CPU使能信号，具体I/O端口列表说明如下所示：

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	异步复位信号（高电平有效）	1
input	wire	stall	流水线暂停信号	5:0
input	wire	branch_flag_i	分支跳转信号	1
input	wire	branch_target_address_i	分支跳转目标位置	31:0
output	reg	pc_o	程序计数器计数值	31:0
output	reg	ce_o	CPU 使能信号	1

该模块进行程序计数，当不处于复位状态时，在每个时钟周期下输出芯片使能信号为有效，并且进行程序计数每次加4（这是因为每条指令占32bit长，即 $4\$ \times 8\text{bit}$ ，因此每次加4），再将该使能信号和计数值传递给ROM模块 inst_rom 进行取指，再将从ROM取到的指令传递给间隔模块 if_id，这个过程是组合电路设计，因此上述操作在同一时钟周期内完成。如果接收到流水线暂停信号，则计数器值不发生改变；如果接收到分支跳转信号，则计数器值在下一时钟周期改变为目标跳转地址，程序计数器部分的流程图如下所示：



(2)只读存储器ROM

在只读存储器 inst_rom 模块中主要储存用于CPU测试的指令，这里可以有多种储存方式，比如：Uart发送、调用ROM的IP核、直接用查找表储存，这里由于为了方便调试、综合速度更快、板载压力更小，采用的是查找表的方式储存指令，即通过 assign 对储存总线 inst_mem 进

行初值赋予，该模块主要接受来自 pc_reg 的使能信号和取值地址，并将取出的指令发送给 if_id 模块，具体的I/O端口列表说明如下所示：

I/O 端口	端口类型	端口名称	端口说明	端口位宽
input	wire	ce_i	使能信号	1
input	wire	addr_i	指令提取地址 (程序计数器值)	31:0
output	reg	inst_o	提取出的机器码指令	31:0

译码部分

(1)译码器模块

在译码器模块 inst_decode 中，通过接受机器码指令将机器码译出机器码中储存的信息，将操作指令、源寄存器值、目标寄存器地址、立即数值等信息传给下一模块，具体I/O端口列表如下：

I/O 端口	端口类型	端口名称	端口说明	端口位宽
input	wire	pc_i	程序计数器值	31:0
input	wire	inst_i	提取出的机器码指令	31:0
input	wire	reg1_data_i	寄存器堆传递的源寄存器 rs 数据值	31:0
input	wire	reg2_data_i	寄存器堆传递的源寄存器 rt 数据值	31:0
output	reg	reg1_read_o	传递给寄存器堆的源寄存器 rs 读取信号	1
output	reg	reg2_read_o	传递给寄存器堆的源寄存器 rt 读取信号	1
output	reg	reg1_addr_o	寄存器堆传递的源寄存器 rs 数据地址	4:0
output	reg	reg2_addr_o	寄存器堆传递的源寄存器 rt 数据地址	4:0
output	reg	alu_op_o	执行指令码	7:0
output	reg	alu_sel_o	执行指令选择码	2:0
output	reg	reg1_o	源寄存器 rs 传递数据值	31:0
output	reg	reg2_o	源寄存器 rt 传递数据值	31:0
output	reg	wd_o	目的寄存器 rd 数据地址	4:0
output	reg	wreg_o	目的寄存器 rd 写入信号	1

以最简单的 ori 指令为例，展示用verilog代码实现译码的功能：

```
'EXE_ORI: begin
    wreg_o = `Write_EN;
    alu_op_o = `EXE_OR_OP;
    alu_sel_o = `EXE_RES_LOGIC;
    reg1_read_o = `Read_EN;
    reg2_read_o = `Read_DIS;
    reg1_addr_o = inst_i[25:21];
    imm = {16'h0, inst_i[15:0]};
    wd_o = inst_i[20:16];
    inst_valid = `Inst_Valid;
```

```

branch_flag_o = `NotBranch;
end

```

然而，由于指令集中还存在着分支转移指令 `beq`, `bne`, `j`, `jal` 等指令，因此需要在译码阶段进行指令，将跳转信号和跳转位置传递给程序计数器，因此需要在上面基础上增加相关接口；此外，由于普通的五级流水线数据进行执行计算后，还需要经过访存、回写两级模块后才能写回到寄存器堆在译码阶段读取，因此可能存在数据执行后但是还没有到达寄存器堆但却需要在译码阶段使用的情况，为处理这种数据冲突的情况，本CPU采取了数据转发的方式，增加输入端口接收来自执行、访存的数据信息（写入地址，写入数据，是否写入），当检测到数据冲突的情况时能够及时更新数据值防止出现问题；最后，由于加载指令 `lw` 等从访存中读取数据需要一定流水线时间，如果紧接着为分支指令，可能会出现数据更新不及时的情况，因此在该阶段存在流水线暂停信号输出，当检测到这种数据冲突时发出流水线暂停信号，使得在分支比较时能够正确采用加载指令得到的数据值。

为解决上述的数据冲突问题并实现上述的解决方案，在译码单元中增添部分接口，增添的I/O端口列表如下：

I/O 端口	端口类型	端口名称	端口说明	端口位宽
input	wire	is_in_delayslot_i	当前指令延迟槽信号	1
input	wire	ex_wreg_i	执行阶段回写信号	1
input	wire	ex_wd_i	执行阶段回写地址	4:0
input	wire	ex_wdata_i	执行阶段回写数据	31:0
input	wire	ex_alu_op_i	执行阶段指令代码	7:0
input	wire	mem_wreg_i	访存阶段回写信号	1
input	wire	mem_wd_i	访存阶段回写地址	4:0
input	wire	mem_wdata_i	访存阶段回写数据	31:0
output	reg	branch_flag_o	分支跳转信号	1
output	reg	branch_target_address_o	分支跳转地址	31:0
output	wire	inst_o	机器码指令	31:0
output	wire	stallreg	流水线暂停信号	1
output	reg	link_address_o	转移指令保存的返回地址	31:0
output	reg	next_inst_in_delayslot_o	下一条指令进入延迟槽信号	1

最终，经修改后，寄存器值的传递代码如下所示，为避免重复叙述，下面只展示了源寄存器rs的代码（该单元为方便起见，当源寄存器没有写入使能时，将其作为立即数寄存器，因此从理论上reg1和reg2都可以是立即数寄存器，但本实验主要是reg2承担imm立即数寄存器的作用）：

```

always @ (*) begin
    if (pre_inst_is_load == 1'b1 && ex_wd_i == reg1_addr_o && reg1_read_o ==
1'b1) begin
        reg1_o = reg1_o;
        stallreg_for_reg1_load = `Stop;
    end else if ((reg1_read_o == `Read_EN) && (ex_wreg_i == `Write_EN) &&
(ex_wd_i == reg1_addr_o)) begin

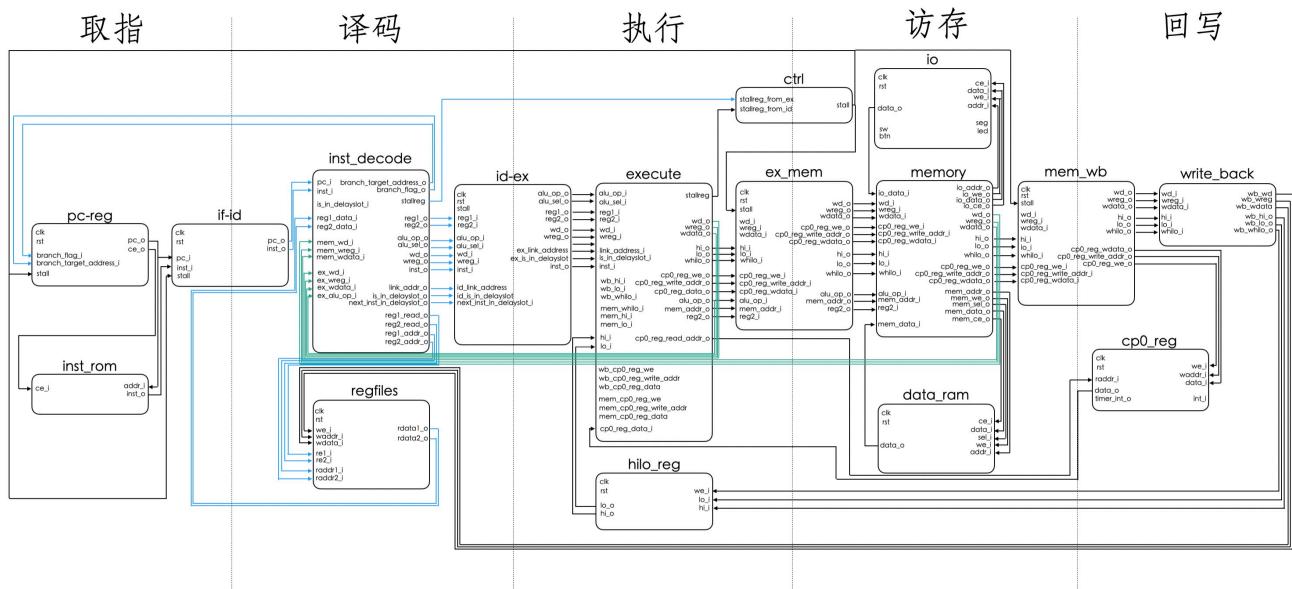
```

```

    reg1_o = ex_wdata_i;
    stallreg_for_reg1_load = 'NoStop';
end else if ((reg1_read_o == 'Read_EN) && (mem_wreg_i == 'Write_EN) &&
(mem_wd_i == reg1_addr_o)) begin
    reg1_o = mem_wdata_i;
    stallreg_for_reg1_load = 'NoStop';
end else if (reg1_read_o == 'Read_EN) begin
    reg1_o = reg1_data_i;
    stallreg_for_reg1_load = 'NoStop';
end else if (reg1_read_o == 'Read_DIS) begin
    reg1_o = imm;
    stallreg_for_reg1_load = 'NoStop';
end else begin
    reg1_o = 'Word_Zero;
    stallreg_for_reg1_load = 'NoStop';
end
end

```

在译码阶段，该模块采取组合电路设计，将根据后面指令集部分的介绍，将机器指令码转换成源寄存器rs与rt、目的寄存器rd、立即数imm、操作指令信息并传递给执行模块用于执行，并在该模块处理分支转移指令和部分数据冲突问题。更具体而言，源寄存器rs和rt的值可能来源于通用寄存器堆存放的数据值，也可能来自后续流水线模块数据转发的值，这将取决于具体情况；而目的寄存器rd的地址、写入使能和立即数imm的数据以及执行指令将直接根据机器指令码给出，具体的流程图如下所示：



(2)通用寄存器堆模块

在通用寄存器堆模块中，存放了32个寄存器。一方面，接受来自译码器的源寄存器信息和读取信号，从通用寄存器堆中将目标信息的数据传递回译码器，这里采用组合电路设计，能够实时读入数据；另一方面，接受回写模块的写入信息和写入信号，将信息存入寄存器堆中，这里采用同步时序电路设计，能够稳定储存数据信息，具体I/O端口列表如下：

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	异步复位信号 (高电平有效)	1
input	wire	we_i	通用寄存器写入使能信号 (高电平有效)	4:0
input	wire	waddr_i	通用寄存器写入地址	4:0
input	wire	wdata_i	通用寄存器输入数据	31:0
input	wire	re1_i	源寄存器 rs 的读取使能 (高电平有效)	1
input	wire	re2_i	源寄存器 rt 的读取使能 (高电平有效)	1
input	wire	raddr1_i	源寄存器 rs 的读取地址	4:0
input	wire	raddr2_i	源寄存器 rt 的读取地址	4:0
output	reg	radata1_o	源寄存器 rs 的数据输出	31:0
output	reg	radata2_o	源寄存器 rt 的数据输出	31:0

执行部分

(1) 算术执行单元ALU

在该阶段，算术执行单元 `execute` 接收来自译码器的源寄存器值和指令操作码或者 `hilo` 寄存器的值并执行指令操作输出结果给下一模块，同时保存译码器传来的写入目的寄存器的相关信息，具体 I/O 端口列表如下：

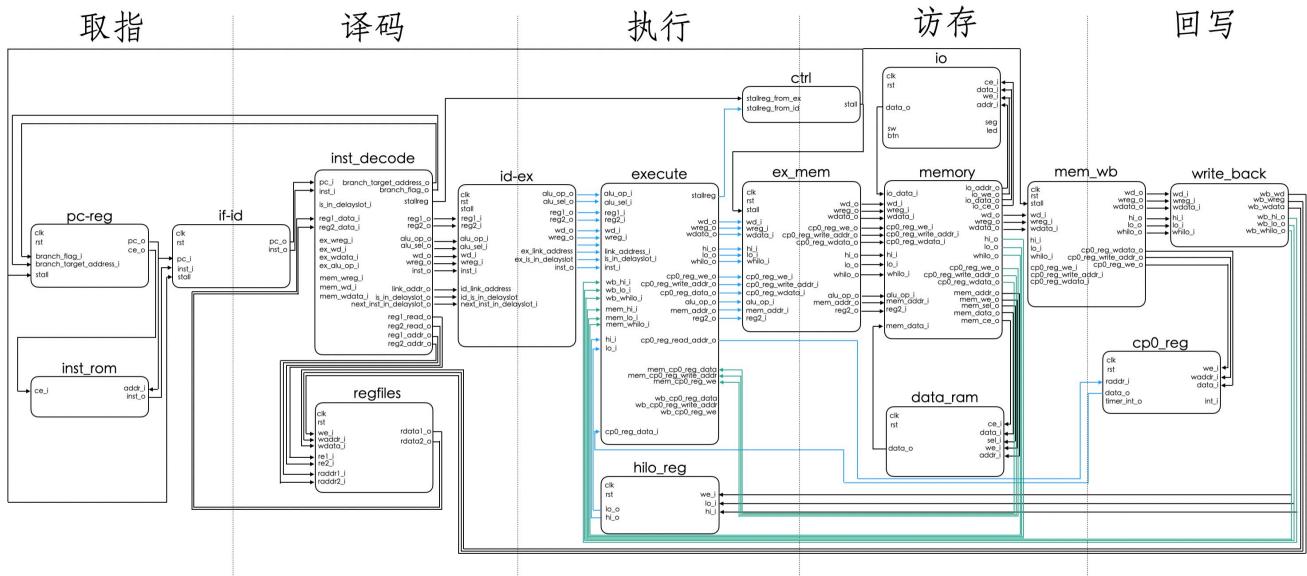
I/O 端口	端口类型	端口名称	端口说明	端口位宽
input	wire	alu_op_i	执行指令码	7:0
input	wire	alu_sel_i	执行指令选择码	3:0
input	wire	reg1_i	源寄存器 rs 传递数据值	31:0
input	wire	reg2_i	源寄存器 rt 传递数据值	31:0
input	wire	wd_i	目的寄存器 rd 写入地址	4:0
input	wire	wreg_i	目的寄存器 rd 写入使能 (高电平有效)	1
input	wire	hi_i	hilo 寄存器高 32 位数据输入	31:0
input	wire	lo_i	hilo 寄存器低 32 位数据输入	31:0
output	reg	wd_o	目的寄存器 rd 数据地址	4:0
output	reg	wreg_o	目的寄存器 rd 写入信号	1
output	reg	wdata_o	目的寄存器 rd 数据值	31:0
output	reg	hi_o	hilo 寄存器高 32 位数据输出	31:0
output	reg	lo_o	hilo 寄存器低 32 位数据输出	31:0
output	reg	whilo_o	hilo 寄存器写入使能 (高电平有效)	1
output	reg	alu_op_o	执行指令码	7:0
output	reg	reg2_o	源寄存器 rt 的输入数据值	31:0
output	reg	mem_addr_o	RAM 访存阶段写入地址	31:0

然而，为了保证 CPU 的运行频率，乘法器与除法器较为复杂的算术运算采用多周期的方式，因此当执行乘除法指令时需要发出流水线暂停信号；此外，类似于前文谈到的译码器阶段寄存

器堆的数据冲突问题，这里也存在特殊寄存器hilo的数据冲突问题，执行阶段提取到的值可能未必是最新的值，因此需要增加访存阶段和回写阶段的数据转发端口；类似上文提及，这里也需要处理指令延迟槽的问题，需要增加相关端口，增加的I/O端口列表如下所示：

I/O 端口	端口类型	端口名称	端口说明	端口位宽
input	wire	is_in_delayslot_i	当前指令延迟槽信号	1
input	wire	link_address_i	转移指令保存的返回地址	31:0
input	wire	mem_hi_i	访存阶段 hilo 的高 32 位数据输入	31:0
input	wire	mem_lo_i	访存阶段 hilo 的低 32 位数据输出	31:0
input	wire	mem_whilo_i	访存阶段 hilo 的写入使能（高电平有效）	1
input	wire	wb_hi_i	回写阶段 hilo 的高 32 位数据输出	31:0
input	wire	wb_lo_i	回写阶段 hilo 的低 32 位数据输出	31:0
input	wire	wb_whilo_i	回写阶段 hilo 的写入使能（高电平有效）	1
output	reg	stallreg	流水线暂停信号	1

算术逻辑单元采取组合电路设计方法，执行指令结果主要分为四大类：逻辑运算结果、移位运算结果、移动结果、算术运算结果，每种结果采用并行计算，在最后通过运算标识符选择结果进行赋值，具体流程图如下所示：



(2)特殊寄存器单元hilo

在该模块采取时序设计，接收hilo寄存器的高输入32位数据和低输入32位数据以及写入使能信号，输出hilo寄存器的高输出32位数据和低输出32位数据，具体I/O端口列表如下所示：

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	异步复位信号（高电平有效）	1
input	wire	hi_i	hilo 寄存器高 32 位数据输入	31:0
input	wire	lo_i	hilo 寄存器低 32 位数据输入	31:0
input	wire	we_i	hilo 寄存器写入使能信号（高电平有效）	1
output	reg	hi_o	hilo 寄存器高 32 位数据输出	31:0
output	reg	lo_o	hilo 寄存器低 32 位数据输出	31:0

该模块主要是存放乘法器得到的64位乘积数据结果，或者存放除法器的结果，用高位寄存器hi储存余数，用低位寄存器lo储存商。当需要读取乘除法结果时，采用 `mfhi`, `mflo` 等指令进行读取，将其高低位分别存入通用寄存器中。

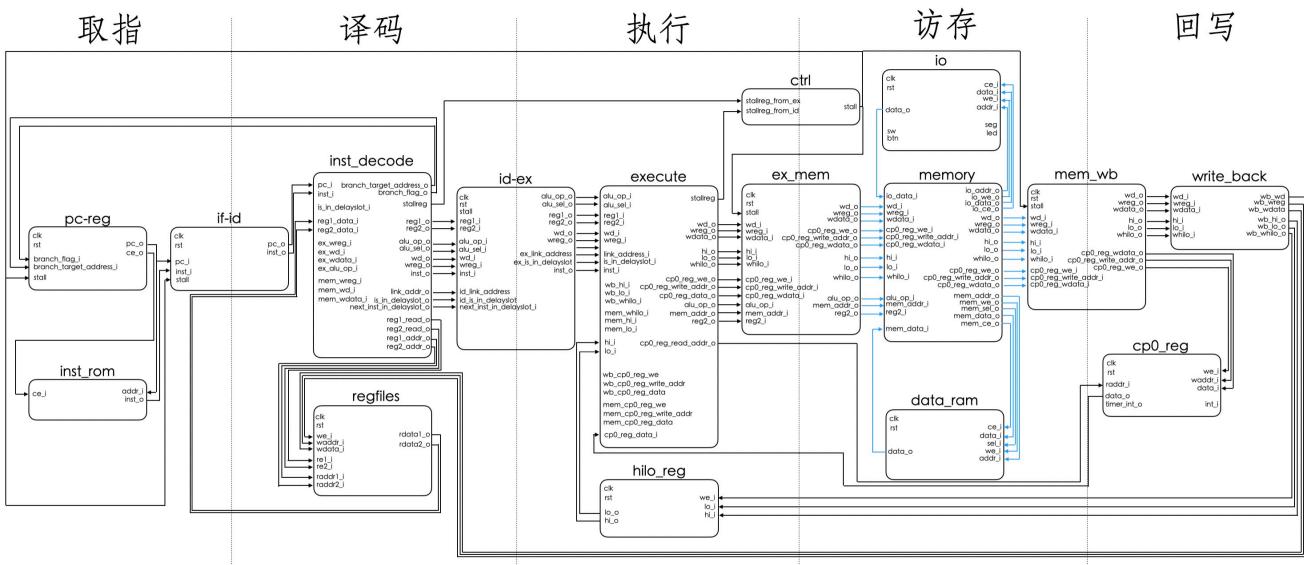
访存部分

(1) 访存单元

在该阶段，访存单元 `memory` 接收来自执行模块的目的寄存器rd和特殊寄存器hilo的相关信息，以及来自执行的指令码信息，并根据接收到的信息将数据传入回写模块或RAM或IO模块中，具体I/O端口列表如下所示：

I/O 端口	端口类型	端口名称	端口说明	端口位宽
input	wire	hi_i	hilo 寄存器高 32 位数据输入	31:0
input	wire	lo_i	hilo 寄存器低 32 位数据输入	31:0
input	wire	whilo_i	hilo 寄存器写入使能信号（高电平有效）	1
input	wire	wd_i	目的寄存器 rd 写入地址	4:0
input	wire	wreg_i	目的寄存器 rd 写入使能（高电平有效）	1
input	wire	wdata_i	目的寄存器 rd 写入数据	31:0
input	wire	alu_op_i	执行指令码	7:0
input	wire	reg2_i	源寄存器 rt 传递数据值	31:0
input	wire	mem_addr_i	RAM 写入地址	31:0
input	wire	mem_data_i	RAM 传入访存单元的数据值	31:0
output	reg	wd_o	目的寄存器 rd 数据地址	4:0
output	reg	wreg_o	目的寄存器 rd 写入信号	1
output	reg	wdata_o	目的寄存器 rd 数据值	31:0
output	reg	hi_o	hilo 寄存器高 32 位数据输出	31:0
output	reg	lo_o	hilo 寄存器低 32 位数据输出	31:0
output	reg	whilo_o	hilo 寄存器写入使能（高电平有效）	1
output	reg	mem_addr_o	RAM 读取或者写入的数据地址	31:0
output	reg	mem_we_o	RAM 的写入使能信号	1
output	reg	mem_data_o	RAM 的写入数据值	31:0
output	reg	mem_sel_o	RAM 写入的字节情况	3:0
output	reg	mem_ce_o	RAM 的使能信号	1

在该模块中采取的是组合电路设计，这里主要是讨论RAM相关的访存，IO模块的访存类似与RAM，具体内容将在后面部分展开讨论。首先对从执行阶段传入的指令进行分支判断，当指令为加载指令 `lw`, `lb`, `lh` 等时，先将要从RAM加载的地址和对应字节位传入RAM中，经过RAM模块取出数据后传回访存单元，将回写数据值赋值为RAM取出后经处理后的数据值（可能存在有无符号数拓展），再将信息传给回写模块用于将数据写回通用寄存器堆中；当判断传入指令为 `sw`, `sb`, `sh` 等储存指令时，对RAM写入使能信号进行激活，将要写入的数据值和传入地址传给RAM进行储存；在指令不为加载、储存指令时，直接将通用寄存器和特殊寄存器的回写信息传给下一模块，具体流程图如下所示：



(2) 随机存取储存器RAM

随机存取储存器RAM接收来自访存单元的写入使能、访存地址、访存数据等信息进行数据的存取，再将读取情况下的数据传回访存单元，具体I/O端口列表如下所示：

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	异步复位信号（高电平有效）	1
input	wire	ce_i	RAM 使能信号（高电平有效）	1
input	wire	we_i	RAM 的写入使能信号（高电平有效）	1
input	wire	address_i	RAM 写入的数据地址	31:0
input	wire	sel_i	RAM 写入的字节情况	3:0
input	wire	data_i	RAM 写入的数据值	31:0
output	reg	data_o	从 RAM 读取到的数据值	31:0

在该模块中读取部分采用组合电路设计，写入部分采用时序电路设计，保证能够在访存时钟周期写能够及时读取数据并且确保写入数据的稳定性。考虑到板载限制大小，这里的RAM容量设为128个32位存储地址（实际上经过测试设为256个32位储存地址也不会出现任何问题，但是综合时间会更长不方便调试）。考虑到加载、储存指令有三大类，按字节、半字、字加载与储存，因此这里为方便存取将每个字的储存分割为4个字节进行存放，为方便后续仿真时能够直观看到每字的储存信息，这里用for循环连接起信息，具体代码展示如下：

```

reg[`Byte_Width] data_mem0[0:127];
reg[`Byte_Width] data_mem1[0:127];
reg[`Byte_Width] data_mem2[0:127];
reg[`Byte_Width] data_mem3[0:127];

wire [31:0] data_mem_combined [0:127];
genvar i;
generate
    for (i = 0; i < 32; i = i + 1) begin: loop_wire
        data_mem_combined[i*4+3: i*4] = data_mem0[i];
        data_mem_combined[i*4+7: i*4+4] = data_mem1[i];
        data_mem_combined[i*4+11: i*4+8] = data_mem2[i];
        data_mem_combined[i*4+15: i*4+12] = data_mem3[i];
    end
endgenerate

```

```

        assign data_mem_combined[i] = {data_mem3[i], data_mem2[i],
data_mem1[i], data_mem0[i]};
      end
endgenerate

```

回写部分

在该部分回写单元 write_back 负责将从访存单元接收到的通用寄存器和特殊寄存器的回写信息传给对应的寄存器堆中，具体I/O端口列表：

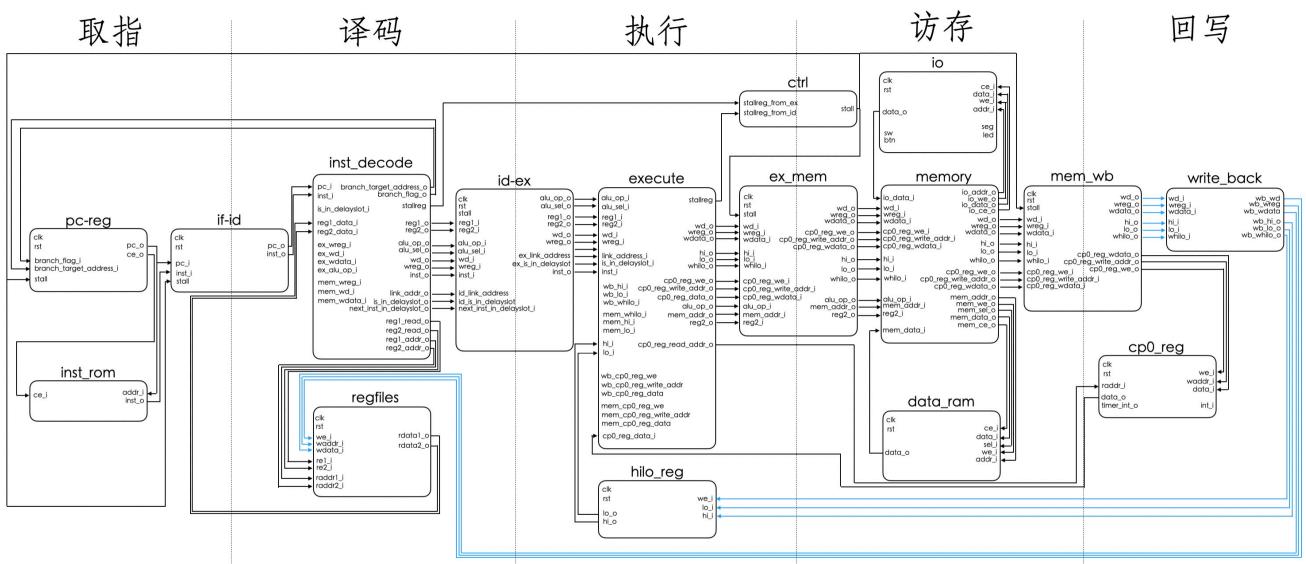
I/O 端口	端口类型	端口名称	端口说明	端口位宽
input	wire	hi_i	hilo 寄存器高 32 位数据输入	31:0
input	wire	lo_i	hilo 寄存器低 32 位数据输入	31:0
input	wire	whilo_i	hilo 寄存器写入使能信号（高电平有效）	1
input	wire	wd_i	目的寄存器 rd 写入地址	4:0
input	wire	wreg_i	目的寄存器 rd 写入使能（高电平有效）	1
input	wire	wdata_i	目的寄存器 rd 写入数据	31:0
output	reg	wb_wd	目的寄存器 rd 数据地址	4:0
output	reg	wb_wreg	目的寄存器 rd 写入信号	1
output	reg	wb_wdata	目的寄存器 rd 数据值	31:0
output	reg	wb_hi_o	hilo 寄存器高 32 位数据输出	31:0
output	reg	wb_lo_o	hilo 寄存器低 32 位数据输出	31:0
output	reg	wb_whilo_o	hilo 寄存器写入使能（高电平有效）	1

回写单元看起来什么事情也没有做，只是单纯花费一个时钟周期对信息进行“连线”（但其实不是assign那样的连线，是对寄存器reg的非阻塞赋值），但这一部分并不能直接去掉化简为四级流水线。

事实上，回写模块主要承担的是调整时序的作用，最大的问题来自于加载指令：在加载指令中，数据是在访存阶段从RAM读取的，但RAM储存器中由于数据量较大，通常存在较大的访问延迟，如果在访存阶段将数据进行写回可能在时钟沿到达时数据并未准备好，无法正确写入数据，导致后续指令出错；此外，如果将回写单元集成在访存单元中，也可能会导致时序路径变长，访存单元需要完成更多的工作，延长了关键路径。比如在本实验中期望达到的频率是100MHz也就是说时序延迟需要控制在10ns内，而经测试删去这一部分后会导致内存访问-寄存器写回的关键路径时序延迟总和来到15ns以上，大大影响CPU的性能。

回写单元同时还解决了寄存器文件的访问冲突（在本实验中不存在这个问题，此问题更多出现于非流水线CPU中），如果在访存阶段直接写会寄存器，那么在同一时钟周期内寄存器文件可能需要同时进行读和写的操作，可能会引起结构冲突问题。

该部分流程图如下所示：



控制模块

该模块接收来自译码阶段或者执行阶段的流水线暂停请求，对每个阶段的流水线暂停与否进行调控，具体I/O端口列表如下：

I/O 端口	端口类型	端口名称	端口说明	端口位宽
input	wire	rst	异步复位信号（高有效）	1
input	wire	stall_from_id	来自译码阶段的流水线暂停请求（高有效）	1
input	wire	stall_from_ex	来自译码阶段的流水线暂停请求（高有效）	1
output	reg	stall	流水线暂停信号	5:0

当接收到来自译码阶段的流水线暂停请求时，暂停取值、译码的流水线，继续执行、访存和回写的流水线，大多发生于加载指令与分支指令的数据冲突上；当接收到执行阶段的流水线暂停请求时，暂停取值、译码和执行的流水线，继续访存和回写的流水线，具体代码如下所示：

```

always @ (*) begin
    if (rst == 'Rst_EN) begin
        stall = 6'b000000;
    end else if (stallreg_from_ex == 'Stop) begin
        stall = 6'b000111; // Stop PC, ir, id, ex, but continue mem and wb
    end else if (stallreg_from_id == 'Stop) begin
        stall = 6'b000011; // Stop PC, ir, id, but continue ex, mem and wb
    end else begin
        stall = 6'b000000;
    end
end

```

衔接模块

上述介绍的主要是五级流水线每个阶段内的模块单元，而在每个阶段间存在衔接单元：if-id（取指-译码）、id-ex（译码-执行）、ex-mem（执行-访存）、mem-wb（访存-回写），在

本实验中衔接单元均采用时序电路设计，与之对应的是阶段内模块单元大多采取组合电路设计。

在衔接单元中，主要是进行异步复位和同步置位，在时钟上升沿将输出信息用输入信息进行非阻塞赋值，保证每个阶段相互独立，占用整数个时钟周期（大多是1个，特殊情况下暂停流水线而导致多个）互不干扰；同时接收异步复位信号，当为高电平时清除CPU内的信息。

此外，衔接单元也负责流水线的暂停功能，接收来自控制单元 `ctrl` 的信息，对流水线的运行进行控制。总的来说，CPU中的时序问题主要都是通过衔接单元进行实现，以 `if-id` 模块为例展示代码如下：

```
always @ (posedge clk, posedge rst) begin
    if (rst == 'Rst_EN) begin
        pc_o <= 'Word_Zero;
        inst_o <= 'Word_Zero;
    end else if (stall[1] == 'Stop && stall[2] == 'NoStop) begin
        pc_o <= 'Word_Zero;
        inst_o <= 'Word_Zero;
    end else if (stall[1] == 'NoStop) begin
        pc_o <= pc_i;
        inst_o <= inst_i;
    end else begin
        pc_o <= pc_o;
        inst_o <= inst_o;
    end
end
end
```

指令集部分

指令集简介

本实验采用MIPS32指令集设计Risc，该指令集主要可以分为R类型指令（基于寄存器操作）、I类型指令（基于立即数操作），J类型指令（跳转指令为主）。

31	26	25	21	20	16	15	11	10	6	5	0	
op	rs	rt	rd	sa	func	R类型						
6位	5位	5位	5位	5位	6位							
31	26	25	21	20	16	15					0	
op	rs	rt				immediate	I类型					
6位	5位	5位				16位						
31	26	25									0	
op					address		J类型					
6位					26位							

(1) R类型指令：具体操作由指令码 `op` 和功能码 `func` 决定，`rs` 和 `rt` 为源寄存器在寄存器堆的编号，`rd` 是目的寄存器在寄存器堆的编号，

MIPS32架构中有32个通用寄存器，因此用五位二进制码表示，sa为移位位数，在移位指令中使用。

(2) **I类型指令**: 具体操作由指令码op决定，指令的低16位是立即数，运算时根据指令码决定有符号拓展为32位或者无符号拓展为32位，然后作为一个源操作数参与运算。

(3) **J类型指令**: 具体操作由指令码op决定，一般为跳转指令，低26位为字地址，用于产生跳转的目标地址。

逻辑运算指令介绍

(1) **ori, andi, xori指令**: 实现逻辑或、逻辑与、逻辑异或运算，以ori指令为例，用法为 ori rt, rs, imm，表示进行运算rs or imm并且把结果保存在通用寄存器rt中。

具体指令码如下：

```
'define EXE_ANDI 6'b001100  
'define EXE_ORI  6'b001101  
'define EXE_XORI 6'b001110
```

(2) **lui指令**: 实现立即数赋值功能，用法为 lui rt, imm，将16bit立即数imm保存为通用寄存器rt的高16位，低16位用0填充。

具体指令码如下：

```
'define EXE_LUI 6'b001111
```

(3) **and, or, xor, nor指令**: 实现逻辑与、逻辑或、逻辑异或、逻辑或非功能，以and指令为例，用法为 and rd, rs, rt，表示进行运算rs and rt并把结果存放在通用寄存器rd中。

具体指令码为 6'b000000，功能码如下：

```
'define EXE_SPECIAL 6'b000000 //指令码  
'define EXE_OR      6'b100101 //功能码  
'define EXE_AND     6'b100100 //功能码  
'define EXE_NOR    6'b100111 //功能码  
'define EXE_XOR    6'b100110 //功能码
```

移位指令介绍

(1) **sll, srl, sra指令**: 实现通用寄存器左移sa (空出来的用0填充)、右移sa (空出来的用0填充)、右移sa (用rt[31]填充空位)，以sll为例，用法为 sll rd, rt, sa 实现对rt << sa并储存在rd中。具体指令码为 6'b000000，功能码如下：

```
'define EXE_SLL 6'b000000  
'define EXE_SRL 6'b000010
```

```
'define EXE_SRA  6'b0000011
```

(2) **sllv, srlv, srav指令**: 实现通用寄存器寄存器左移（空出来的用0填充）、右移（空出来的用0填充）、右移（用rt[31]填充空位），移动距离由rs决定，以sllv为例，用法位 `sll rd, rt, rs` 实现 $rt \ll rs$ 并储存在rd中。具体指令码为 `6'b0000000`，功能码如下：

```
'define EXE_SLLV  6'b000100  
'define EXE_SRLV  6'b000110  
'define EXE_SRAV  6'b000111
```

移动指令介绍

(1) **movn, movz指令**: 前者（Move Conditional on Not Zero）实现判断通用寄存器rt的值是否为0，若不为0，则把rs的值赋给rd，否则保持不变；后者（Move Conditional on Zero）实现判断通用寄存器rt的值是否为0，若为0，则把rs的值赋给rd，否则保持不变。以movz为例，具体用法为 `movz rd, rs, rt`，表示 $if rt = 0 \text{ then } rd = rs$ ，具体指令码为 `6'b0000000`，功能码如下：

```
'define EXE_MOVZ  6'b001010  
'define EXE_MOVN  6'b001011
```

(2) **mfhi, mthi, mflo, mtlo指令**: f表示from，t表示to，hi和lo表示特殊寄存器hi和lo，mfhi实现功能hi的值赋给rd，mthi实现功能rs的值赋给hi，用法为 `mthi rs` 和 `mfhi rd`，具体指令码为 `6'b0000000`，功能码如下：

```
'define EXE_MFHI  6'b0100000  
'define EXE_MTHI  6'b0100001  
'define EXE_MFL0  6'b0100010  
'define EXE_MTLO  6'b0100011
```

算术操作指令介绍

(1) **add, addu, sub, subu指令**: 实现加法功能（检查溢出，若溢出不保存结果），加法功能（不检查溢出），减法功能（检查溢出，若溢出不保存结果），减法功能（不检查溢出），以sub为例用法为 `sub rd, rs, rt` 表示把rs-rt的运算结果保存在rd中，具体指令码为 `6'b0000000`，功能码如下：

```
'define EXE_ADD   6'b1000000  
'define EXE_ADDU  6'b1000001  
'define EXE_SUB   6'b1000010  
'define EXE_SUBU  6'b1000011
```

(2) **slt, sltu指令**: 实现比较运算功能，前者进行有符号数比较，后者进行无符号数比较，若 $rs < rt$ 则把1保存在rd中，否则保存0，实例 `slt rd, rs, rt`，具体指令码为 `6'b0000000`，功能

码如下：

```
'define EXE_SLT    6'b101010
'define EXE_SLTU   6'b101011
```

(3) **addi, addiu指令**: 实现立即数运算，具体与前文提到的add等运算规则一致，将imm进行符号拓展后进行运算，用法为 addi rt, rs, imm 表示把rs+imm的结果保存在rt中具体指令码如下：

```
'define EXE_ADDI   6'b001000
'define EXE_ADDIU  6'b001001
```

(4) **slti, sltiu指令**: 实现立即数比较，具体与前文slt等运算规则一致，将imm进行符号拓展后进行比较，用法为 slti rt, rs, imm 表示把rs < imm的结果存在rt中，具体指令码如下：

```
'define EXE_SLTI   6'b001010
'define EXE_SLTIU  6'b001011
```

(5) **clo, clz指令**: 实现计数运算，对rs寄存器的值进行计数，从最高位到最低位进行检查，直到遇到为0 (1) 的位，并且把前面1 (0) 的个数存给rd，如果全为1 (0) 则存放32，具体指令码为 6'b011100，功能码如下：

```
'define EXE_SPECIAL2_INST 6'b011100
'define EXE_CLZ   6'b100000
'define EXE_CLO   6'b100001
```

转移指令介绍

(1) **j, jal指令**: 实现跳转指令，前者表示跳转到新指令地址，新指令地址的低28位为target左移两位的值，高4位为跳转指令后延迟槽指令的地址高4位；后者在前者的基础上把后面第二条指令的地址作为返回地址保存在寄存器31中。两条指令在转移前都要执行执行槽指令，指令格式为 j target，为方便起见，本实验中通过编写python代码，可以用跳转位置表示target，具体如下：

```
skip_swap:
    addi $4, $4, 1
    j inner_loop
    nop

inner_loop:
    addi $3, $3, 1

outer_loop:
    nop
```

具体指令码如下：

```
'define EXE_J      6'b0000010  
'define EXE_JAL    6'b0000011
```

(2) **jr, jalr**指令：前者实现将通用寄存器rs的值赋值给PC寄存器，作为新的指令地址，后者在前者的基础上把后面第二条指令的地址作为返回地址保存在寄存器31或rd中。两条指令在转移前都要执行执行槽指令，用法为 `jalr rs` 或者 `jalr rd, rs`，指令码均为 `6'b0000000`，具体功能码如下：

```
'define EXE_JALR   6'b0010001  
'define EXE_JR     6'b0010000
```

(3) **beq, bgtz, blez, bne**指令：分别实现判断是否有 $rs = rt$, $rs > 0$, $rs \leq 0$, $rs \neq rt$ 成立，如果成立执行跳转指令，offset左移2位，并符号扩展至32位，然后与延迟槽指令的地址相加，结果为目标跳转指令地址，用法为 `beq rs, rt, offset` 或 `bgtz rs, offset`，具体指令码如下：

```
'define EXE_BEQ    6'b0001000  
'define EXE_BGTZ   6'b0001110  
'define EXE_BLEZ   6'b0001110  
'define EXE_BNE    6'b0001010
```

加载与储存指令介绍

(1) **lb, lbu, lh, lhu, lw**指令：设加载地址为offset进行32位拓展后与base寄存器相加结果，指令作用从前到后依次为：从加载地址处读取1字节（8位）符号拓展至32位存在rt中，从加载地址处读取1字节无符号拓展至32位存在rt中，从加载地址处读取半字（16位）后符号拓展至32位存在rt中，从加载地址处读取半字（16位）后无符号拓展至32位存在rt中，从加载地址处读取1字（32位）后符号拓展至32位存在rt中。后三条指令有地址对齐要求，加载地址最低位为0, 0, 00。用法实例为 `lw rt, offset(base)`，具体指令码如下：

```
'define EXE_LB     6'b1000000  
'define EXE_LBU    6'b1001000  
'define EXE_LH     6'b1000001  
'define EXE_LHU    6'b1001001  
'define EXE_LW     6'b1000110
```

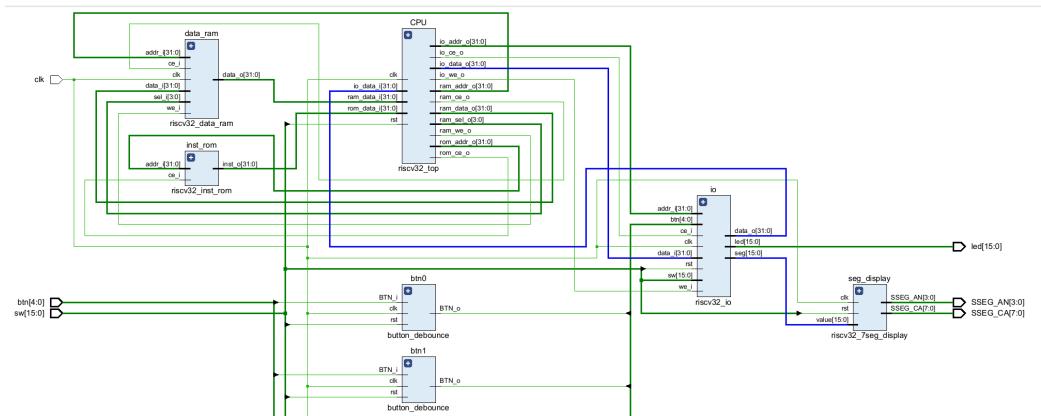
(2) **sb, sh, sw**指令：设加载地址为offset进行32位拓展后与base寄存器相加结果，指令作用从前到后依次为：将rt的最低字节储存在目标地址中，将rt的最低两个字节储存在目标地址中（储存最低位地址位0，有对齐要求），将rt的值存在目标地址中（储存最低位地址位00，有对齐要求），实例为 `sw rt, offset(base)`，具体指令码如下：

```
'define EXE_SB    6'b1010000  
'define EXE_SH    6'b1010001
```

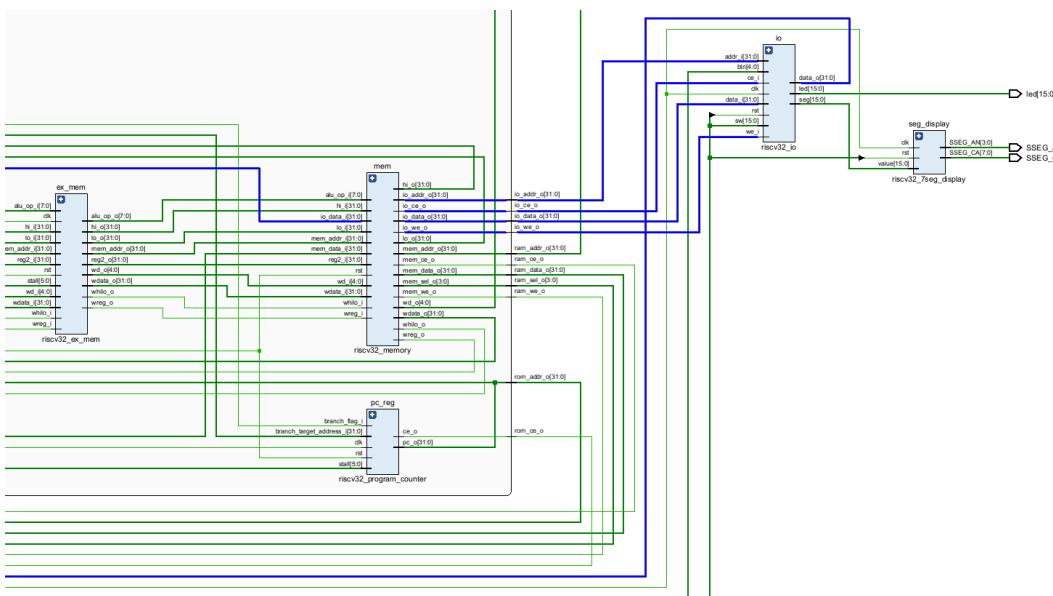
```
'define EXE_SW 6'b101011
```

FPGA开发板外设连接部分

本实验使用Basys3开发板作为FPGA外设，将外设跟内存统一编址，将32位RAM内存地址的后三位划分给外设（实际上由于板子大小有限，RAM目前只有32个地址空间，这里我们假设在RAM中设定的32个寄存器外还有几个空间用于储存外设数据），我们将地址为 $32^4 h0000_f000$ 的RAM划分给LED灯显示；将地址为 $32^4 h0000_f010$ 的RAM划分给SW开关输入；将地址为 $32^4 h0000_f020$ 的RAM划分为数码管的显示数据，用于传递给数码管显示模块；将地址为 $32^4 h0000_f030$ 的RAM划分给按键BTN输入。



如上图所示，新增IO模块与外设相连接，将其接口接入CPU模块中的访存memory模块中，通过传递 `io_addr` 地址来确定操作何种外设或者是否操作外设，通过传递 `io_data_o`, `io_data_i` 实现数据传递，实现对外设的读写操作，通过 `io_ce`, `io_we` 确定是否运行IO模块和是否写入IO模块。



对memory访存模块修改，当执行 `lw` 加载指令时，表示加载数据，在原有代码基础上，对加载数据地址进行判断，如果地址小于 $32^4 h0000_f000$ 则表明是正常的从RAM中读入数据，而非从外设读入数据；否则表明从外设中读入数据，因此对io接口进行操作，传入操作加载地址。执行 `sw` 储存指令类似，表示储存数据，在外设中表示写入输出，与加载指令类似，不再赘述，具体代码如下：

```

`EXE_LW_OP: begin
    if (mem_addr_i < 32'h0000_f000) begin
        mem_addr_o = mem_addr_i;
        mem_we_o = `Write_DIS;
        mem_ce_o = `Chip_EN;
        mem_sel_o = 4'b1111;
        wdata_o = mem_data_i;
    end else begin
        io_addr_o = mem_addr_i;
        io_we_o = `Write_DIS;
        wdata_o = io_data_i;
        io_ce_o = `Chip_EN;
    end

```

设置io模块，这里采用组合电路设计，对从memory访存模块接收到的地址 `addr_i` 和 `we_i` 进行条件判断，如果 `we_i` 确定执行写入操作，则表明是对外设进行输出操作，因此再判断 `addr_i` 地址为LED地址还是SEG地址，并对其进行数据传递；如果不执行写入操作，则执行读操作，因此再判断 `addr_i` 地址为SW地址还是BTN地址，具体代码如下：

```

module risc32_io (
    input wire                      clk,
    input wire                      rst,
    input wire                      ce_i,
    input wire                      we_i,
    input wire[`Data_Addr_Bus]      addr_i,
    input wire[`Data_Bus]            data_i,
    input wire[15:0]                 sw,
    input wire[4:0]                 btn,

    output reg[15:0]                seg,
    output reg[`Data_Bus]           data_o,
    output reg[15:0]                led
);

    reg ce_i_sync, we_i_sync;
    reg [`Data_Addr_Bus] addr_i_sync;
    reg [`Data_Bus] data_i_sync;

    always @(posedge clk or posedge rst) begin
        if (rst == `Rst_EN) begin
            ce_i_sync <= `Chip_DIS;
            we_i_sync <= `Write_DIS;
            addr_i_sync <= `Word_Zero;
            data_i_sync <= `Word_Zero;
        end else begin
            ce_i_sync <= ce_i;
            we_i_sync <= we_i;
            addr_i_sync <= addr_i;
        end
    end

```

```

        data_i_sync <= data_i;
    end
end

always @ (posedge clk or posedge rst) begin
    if (rst == 'Rst_EN) begin
        led <= 16'h0000;
        seg <= 16'h0000;
    end else begin
        led <= led;
        seg <= seg;
        if (ce_i_sync == 'Chip_EN && we_i_sync == 'Write_EN) begin
            if (addr_i_sync == 'LED) begin
                led <= data_i_sync[15:0];
            end else if (addr_i_sync == 'SEG) begin
                seg <= data_i_sync[15:0];
            end
        end
    end
end

always @(*) begin
    if (rst == 'Rst_EN) begin
        data_o = 'Word_Zero;
    end else begin
        if (ce_i == 'Chip_EN) begin
            if (we_i == 'Write_DIS) begin
                if (addr_i == 'SWITCH) begin
                    data_o = {16'b0, sw};
                end else if (addr_i == 'BTN) begin
                    data_o = {27'b0, btn};
                end else begin
                    data_o = 'Word_Zero;
                end
            end else begin
                data_o = 'Word_Zero;
            end
        end else begin
            data_o = 'Word_Zero;
        end
    end
end
endmodule

```

指令操作示例，比如下面我要进行读入开关中的数据，再读入操作按键的指令，进行一番操作后再输出结果到LED灯和数码管中：

```

lui $3, 0x0000
ori $3, $3, 0xf000

```

```

lw $4, 0x10($3) // 从0000_f010读入SW数据，并存放在 $4寄存器堆中

lw $6, 0x30($3) // 从0000_f030读入BTN数据，并存放在 $6寄存器堆中
...
sw $1, 0x0($3) // 将 $1中的数据写入外设0000_f000LED中

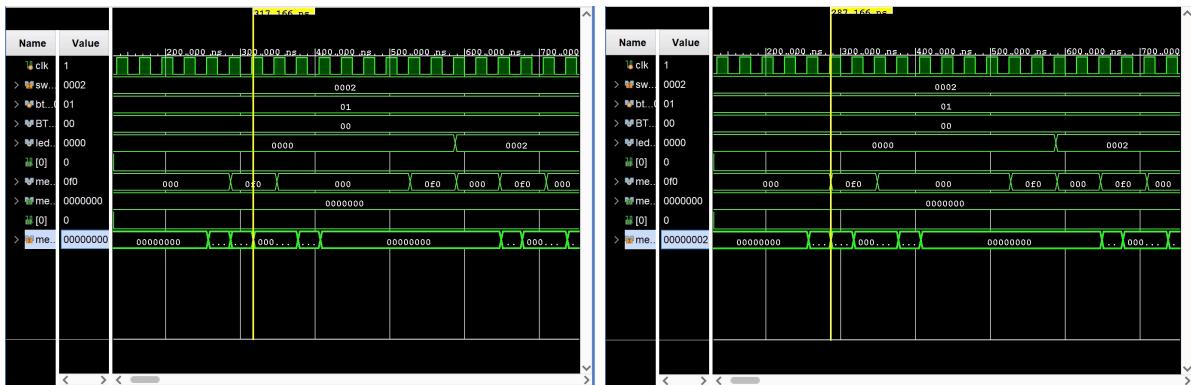
sw $1, 0x20($3) // 将 $1中的数据写入外设0000_f020数码管的输入数据中

```

仿真测试部分

(0)时序仿真测试

本次时序仿真主要测试延迟通常较大的访存模块，以访存模块数据输出为检测对象，进行 Vivado 中的 Run Post-Implementation Timing Simulation，测的结果如下，主要延迟测得为 2.066ns 和 2.166ns，完全符合 FPGA 开发板内置时钟 100MHz 的要求，初步说明了 CPU 设计在时序上的合理性，具体展示如下：



(1)冒泡排序测试

本次仿真测试通过使用 sw 指令对 ram 前 10 个地址随机赋予初值，然后对其进行冒泡排序，并实时更新 ram 中的值以体现冒泡排序过程，最后在 ram 中完成排序。这种情况下，ram 充当了 C 语言中的“数组”的作用，用于储存数据，具体汇编代码如下：

```

# Initialize RAM addresses 0 to 9 with given values
addi $1, $0, 0          # $1 = base address of array (RAM[0])

addi $2, $0, 3          # $2 = 3
sw $2, 0($1)            # RAM[0] = 3
addi $2, $0, 18         # $2 = 18
sw $2, 4($1)            # RAM[1] = 18
addi $2, $0, 4           # $2 = 4
sw $2, 8($1)            # RAM[2] = 4
addi $2, $0, 37          # $2 = 37
sw $2, 12($1)           # RAM[3] = 37
addi $2, $0, 198         # $2 = 198
sw $2, 16($1)           # RAM[4] = 198
addi $2, $0, 210          # $2 = 210

```

```

sw $2, 20($1)      # RAM[5] = 210
addi $2, $0, 102    # $2 = 102
sw $2, 24($1)      # RAM[6] = 102
addi $2, $0, 15     # $2 = 15
sw $2, 28($1)      # RAM[7] = 15
addi $2, $0, 97     # $2 = 97
sw $2, 32($1)      # RAM[8] = 97
addi $2, $0, 0       # $2 = 0 (assuming last value is 0)
sw $2, 36($1)      # RAM[9] = 0

addi $3, $0, 10      # $3 = 10 (array length n)
addi $4, $0, 0       # $4 = 0 (outer loop index i)

outer_loop:
slt $5, $4, $3      # $5 = 1 if $4 < $3
beq $5, $0, end      # if $5 == 0, jump to end

addi $6, $0, 0       # $6 = 0 (inner loop index j)
addi $7, $3, -1      # $7 = $3 - 1

inner_loop:
sub $8, $7, $4      # $8 = $7 - $4
slt $5, $6, $8      # $5 = 1 if $6 < $8
beq $5, $0, outer_increment  # if $5 == 0, increment i

sll $9, $6, 2        # $9 = $6 * 4 (offset for RAM[j])
add $10, $1, $9      # $10 = base address + offset
lw $11, 0($10)      # $11 = RAM[j]

addi $12, $6, 1       # $12 = $6 + 1
sll $12, $12, 2      # $12 = ($6 + 1) * 4
add $13, $1, $12      # $13 = base address + offset
lw $14, 0($13)      # $14 = RAM[j + 1]

# Compare
slt $5, $14, $11      # $5 = 1 if $14 < $11
beq $5, $0, inner_increment  # if $5 == 0, skip swapping

# Swap RAM[j] and RAM[j + 1]
sw $14, 0($10)      # RAM[j] = $14
sw $11, 0($13)      # RAM[j + 1] = $11

inner_increment:
addi $6, $6, 1        # $6 = $6 + 1
j inner_loop
nop

outer_increment:
addi $4, $4, 1        # $4 = $4 + 1
j outer_loop

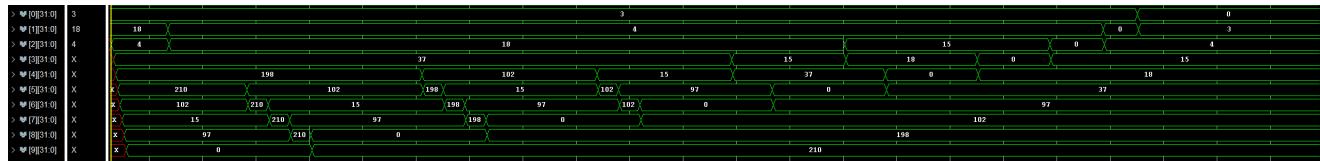
```

```
nop
```

```
end:
```

```
nop
```

如上述汇编代码所展示，对0到9的ram地址赋予初值3, 18, 4, 37, 198, 210, 102, 15, 97, 0然后通过CPU实现冒泡排序，得到仿真波形图如下，如图所示，我们可以看到CPU很好的完成任务并且展示了冒泡排序的过程，从波形图我们也能直观感受到“冒泡”的过程，比如储存在ram[9]的0从下面冒泡到最上面，波形宽度越来越小：



(2)二分查找

本次仿真测试通过使用sw指令对ram前10个地址随机赋予初值，在ram[10]中储存待查找的数据值，最后在ram[11]输出查找到的数据的位置+1，即是第几个数，如果查找失败输出32'hffff_ffff，具体汇编代码如下：

```
# 初始化阶段：将数据存储到 RAM[0] 到 RAM[9]，在 RAM[10] 存储要查找的键值 17
addi $1, $0, 0          # $1 = RAM 的基地址（假设从地址 0 开始）

addi $2, $0, 1          # $2 = 1
sw    $2, 0($1)         # RAM[0] = 1

addi $2, $0, 17         # $2 = 17
sw    $2, 4($1)         # RAM[1] = 17

addi $2, $0, 45         # $2 = 45
sw    $2, 8($1)         # RAM[2] = 45

addi $2, $0, 57         # $2 = 57
sw    $2, 12($1)        # RAM[3] = 57

addi $2, $0, 95         # $2 = 95
sw    $2, 16($1)        # RAM[4] = 95

addi $2, $0, 103        # $2 = 103
sw    $2, 20($1)        # RAM[5] = 103

addi $2, $0, 105        # $2 = 105
sw    $2, 24($1)        # RAM[6] = 105

addi $2, $0, 198        # $2 = 198
sw    $2, 28($1)        # RAM[7] = 198
```

```

addi $2, $0, 240      # $2 = 240
sw   $2, 32($1)       # RAM[8] = 240

addi $2, $0, 253      # $2 = 253
sw   $2, 36($1)       # RAM[9] = 253

# 将要查找的键值 57 存储到 RAM[10]
addi $2, $0, 57        # $2 = 57
sw   $2, 40($1)        # RAM[10] = 57

# 二分查找算法
lw   $5, 40($1)        # $5 = key = RAM[10]

addi $3, $0, 0          # $3 = low = 0
addi $4, $0, 9          # $4 = high = 9

search_loop:
    slt  $8, $4, $3      # $8 = 1 if high < low
    bne  $8, $0, not_found  # 如果 high < low, 跳转到 not_found

    add  $6, $3, $4        # $6 = low + high
    srl $6, $6, 1         # $6 = mid = (low + high) / 2

    sll  $9, $6, 2        # $9 = mid * 4 (字节偏移)
    add  $9, $1, $9        # $9 = RAM[mid] 的地址
    lw   $7, 0($9)        # $7 = RAM[mid]

    beq  $7, $5, found    # 如果 RAM[mid] == key, 跳转到 found

    slt  $8, $7, $5        # $8 = 1 if RAM[mid] < key
    bne  $8, $0, adjust_low  # 如果 RAM[mid] < key, 调整 low

    # 调整 high: high = mid
    addi $4, $6, -1        # high = mid - 1
    j    search_loop       # 返回循环
    nop

adjust_low:
    # 调整 low: low = mid + 1
    addi $3, $6, 1          # low = mid + 1
    j    search_loop       # 返回循环
    nop

found:
    # 找到键值, 计算位置 (从 1 开始计数)
    addi $7, $6, 1          # $7 = position = mid + 1

    # 将结果存储到 RAM[11]
    addi $9, $0, 44          # $9 = RAM[11] 的地址偏移 (11 * 4)
    add  $9, $1, $9          # $9 = 完整地址

```

```

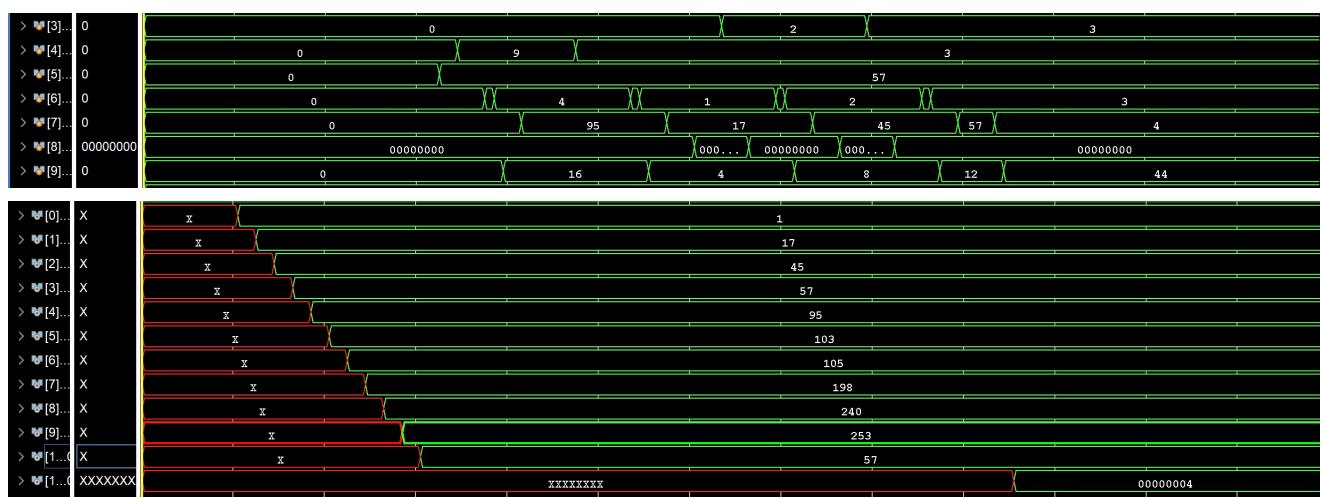
sw    $7, 0($9)      # RAM[11] = position
j     end             # 结束程序
nop

not_found:
# 未找到键值，存储 -1 到 RAM[11]
addi $7, $0, -1       # $7 = -1
addi $9, $0, 44        # $9 = RAM[11] 的地址偏移 (11 * 4)
add  $9, $1, $9         # $9 = 完整地址
sw    $7, 0($9)      # RAM[11] = -1

end:
nop

```

如上述汇编代码所展示，对0到9的ram地址赋予初值1, 17, 47, 57, 95, 103, 105, 198, 240, 253，查找目标为57，查找结果如下所示，通过寄存器堆\$3和寄存器堆\$4储存查找下限和查找上限，用\$6储存中间位置，\$8为比较结果储存器，得到最终结果57位于第四个位置，正确，结果如下：



(3)素数判定

本次仿真测试通过使用sw指令对ram[0]随机赋予初值，在ram[1]中储存结果，当ram[0]的值为素数时保存为1，否则保存为0，具体汇编代码如下：

```

# 初始化 RAM 基地址
addi $1, $0, 0          # $1 = RAM 基地址 (假设基址为 0)

addi $2, $0, 1007        # $2 = 1007
sw    $2, 0($1)          # RAM[0] = 1007

# 从 RAM[0] 加载要判断的数 n
lw    $2, 0($1)          # $2 = n

# 边界检查：如果 n < 2，则不是素数
addi $3, $0, 2            # $3 = 2
slt   $4, $2, $3          # 如果 n < 2, $4 = 1

```

```

bne $4, $0, not_prime # 如果 $4 != 0, 跳转到 not_prime

# 初始化除数 i = 2
addi $5, $0, 2          # $5 = i

check_loop:
    # 计算 i_squared = i * i (使用加法循环)
    add $6, $0, $0          # $6 = i_squared = 0
    add $7, $0, $5          # $7 = counter = i

i_square_loop:
    beq $7, $0, compare_i_n # 如果 counter == 0, 跳转到 compare_i_n

    add $6, $6, $5          # i_squared += i
    addi $7, $7, -1         # counter -= 1
    j i_square_loop

compare_i_n:
    # 如果 i_squared > n, 跳转到 determine_prime
    slt $8, $2, $6          # 如果 n < i_squared, $8 = 1
    bne $8, $0, determine_prime # 如果 n < i_squared, 跳转 determine_prime

    # 不论 n 是否等于 i_squared, 都要检查 n 是否能被 i 整除
    add $9, $0, $2          # $9 = temp = n

mod_loop:
    sub $9, $9, $5          # temp -= i
    slt $10, $9, $0          # 如果 temp < 0, $10 = 1
    bne $10, $0, next_i     # 如果 temp < 0, 跳转到 next_i

    beq $9, $0, not_prime   # 如果 temp == 0, n 能被 i 整除, 非素数

    j mod_loop

next_i:
    # i += 1
    addi $5, $5, 1

    # 重新计算 i_squared
    j check_loop

determine_prime:
    # 没有找到因数, n 是素数
    j is_prime

is_prime:
    # 将结果 1 (素数) 存入 RAM[1]
    addi $11, $0, 4          # 偏移量 = 1 * 4 字节
    add $11, $1, $11          # $11 = RAM[1] 的地址
    addi $12, $0, 1          # $12 = 1

```

```

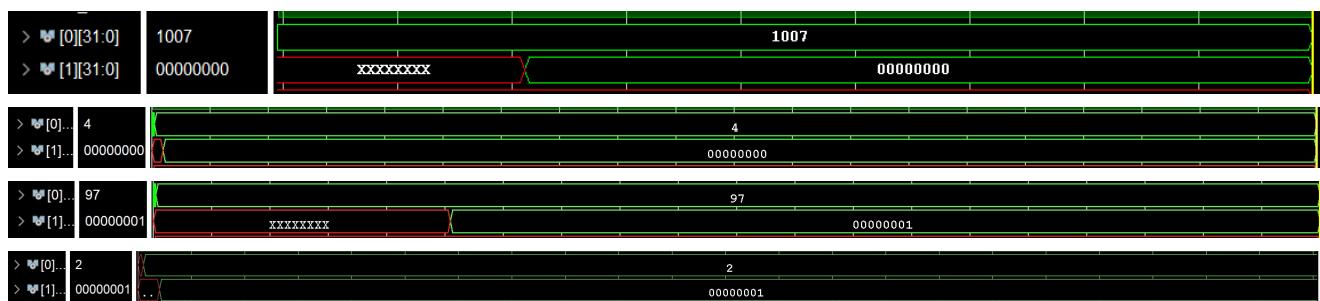
        sw    $12, 0($11)          # RAM[1] = 1
        j     end_program

not_prime:
    # 将结果 0 (非素数) 存入 RAM[1]
    addi $11, $0, 4            # 偏移量 = 1 * 4 字节
    add  $11, $1, $11           # $11 = RAM[1] 的地址
    addi $12, $0, 0              # $12 = 0
    sw   $12, 0($11)            # RAM[1] = 0

end_program:
    # 程序结束
    nop

```

如上述汇编代码所展示，对ram[0]分别赋值1007, 4, 97, 2，得到判定结果为0, 0, 1, 1全部正确，具体仿真结果如下：



(4)快速排序

本次仿真测试通过使用sw指令对ram前10个地址随机赋予初值，然后对其进行快速排序，并实时更新ram中的值以体现快速排序过程，最后在ram中完成排序。这种情况下，ram充当了C语言中的“数组”的作用，用于储存数据，具体汇编代码如下：

```

# 初始化数组
addi $1, $0, 0      # $1 = 数组基地址 (RAM[0])

addi $2, $0, 29      # $2 = 29
sw $2, 0($1)        # RAM[0] = 29
addi $2, $0, 10      # $2 = 10
sw $2, 4($1)        # RAM[1] = 10
addi $2, $0, 14      # $2 = 14
sw $2, 8($1)        # RAM[2] = 14
addi $2, $0, 37      # $2 = 37
sw $2, 12($1)       # RAM[3] = 37
addi $2, $0, 13      # $2 = 13
sw $2, 16($1)       # RAM[4] = 13
addi $2, $0, 20      # $2 = 20
sw $2, 20($1)       # RAM[5] = 20
addi $2, $0, 25      # $2 = 25
sw $2, 24($1)       # RAM[6] = 25

```

```

addi $2, $0, 44      # $2 = 44
sw $2, 28($1)        # RAM[7] = 44
addi $2, $0, 19      # $2 = 19
sw $2, 32($1)        # RAM[8] = 19
addi $2, $0, 17      # $2 = 17
sw $2, 36($1)        # RAM[9] = 17

# 初始化栈指针和参数
addi $29, $0, 200    # $29 = 栈指针 (SP), 初始化为内存地址 200
addi $3, $0, 0         # $3 = left, 数组左边界索引
addi $4, $0, 9         # $4 = right, 数组右边界索引

# 将初始的 left 和 right 压入栈
addi $29, $29, -8     # $29 = $29 - 8, 栈指针向下移动
sw $3, 0($29)          # 栈[$29] = left
sw $4, 4($29)          # 栈[$29 + 4] = right

# 开始主循环
main_loop:
    # 判断栈是否为空 (栈指针是否回到了初始位置 200)
    addi $5, $29, 0        # $5 = $29 (当前栈指针)
    addi $6, $0, 200        # $6 = 200 (初始栈指针位置)
    beq $5, $6, end        # 如果 $29 == 200, 跳转到 end, 排序完成

    # 从栈中弹出 right 和 left
    lw $4, 4($29)          # $4 = 栈[$29 + 4], right
    lw $3, 0($29)          # $3 = 栈[$29], left
    addi $29, $29, 8        # $29 = $29 + 8, 栈指针向上移动

    # 如果 left >= right, 跳过本次循环
    slt $7, $3, $4          # $7 = ($3 < $4) ? 1 : 0
    beq $7, $0, main_loop   # 如果 $3 >= $4, 跳回 main_loop

    # 分区操作
    # 选择 pivot 为数组的 right 位置的元素
    sll $8, $4, 2            # $8 = $4 * 4
    add $8, $1, $8            # $8 = 数组基址 + right * 4
    lw $9, 0($8)              # $9 = pivot = 数组[right]

    # 初始化 i 为 left - 1
    addi $10, $3, -1         # $10 = i = left - 1

    # 初始化 j 为 left
    add $11, $0, $3            # $11 = j = left

partition_loop:
    # 判断 j 是否小于 right
    slt $7, $11, $4          # $7 = ($11 < $4) ? 1 : 0
    beq $7, $0, partition_end # 如果 j >= right, 跳转到 partition_end

```

```

# 获取数组[j]的值
sll $12, $11, 2      # $12 = j * 4
add $12, $1, $12      # $12 = 数组基地址 + j * 4
lw $13, 0($12)        # $13 = 数组[j]

# 比较数组[j]和 pivot
slt $7, $13, $9       # $7 = ($13 < $9) ? 1 : 0
beq $7, $0, partition_loop_increment_j # 如果数组[j] >= pivot, 跳过交换

# i = i + 1
addi $10, $10, 1       # $10 = i = i + 1

# 交换数组[i]和数组[j]
sll $14, $10, 2        # $14 = i * 4
add $14, $1, $14        # $14 = 数组基地址 + i * 4
lw $15, 0($14)         # $15 = 数组[i]

sw $15, 0($12)         # 数组[j] = 数组[i]
sw $13, 0($14)         # 数组[i] = 数组[j]

partition_loop_increment_j:
    # j = j + 1
    addi $11, $11, 1       # $11 = j = j + 1
    j partition_loop      # 跳回 partition_loop

partition_end:
    # 将 pivot 放到正确的位置
    addi $10, $10, 1       # $10 = i = i + 1
    sll $14, $10, 2        # $14 = i * 4
    add $14, $1, $14        # $14 = 数组基地址 + i * 4
    lw $15, 0($14)         # $15 = 数组[i]

    # 读取数组[right]的值
    sll $12, $4, 2          # $12 = right * 4
    add $12, $1, $12        # $12 = 数组基地址 + right * 4
    lw $13, 0($12)          # $13 = 数组[right] (pivot)

    # 交换数组[i]和数组[right]
    sw $15, 0($12)          # 数组[right] = 数组[i]
    sw $13, 0($14)          # 数组[i] = 数组[right]

    # pivot 索引为 i ($10)

    # 将左子数组的边界压入栈
    addi $7, $10, -1        # $7 = pivotIndex - 1
    slt $16, $3, $7          # $16 = ($3 < $7) ? 1 : 0
    beq $16, $0, skip_push_left # 如果 left >= pivotIndex - 1, 跳过

    addi $29, $29, -8        # 栈指针向下移动
    sw $3, 0($29)            # 栈[$29] = left

```

```
sw $7, 4($29)      # 栈[$29 + 4] = pivotIndex - 1
```

skip_push_left:

```
# 将右子数组的边界压入栈
```

```
addi $7, $10, 1      # $7 = pivotIndex + 1
```

```
slt $16, $7, $4       # $16 = ($7 < $4) ? 1 : 0
```

```
beq $16, $0, main_loop # 如果 pivotIndex + 1 >= right, 跳回主循环
```

```
addi $29, $29, -8     # 栈指针向下移动
```

```
sw $7, 0($29)        # 栈[$29] = pivotIndex + 1
```

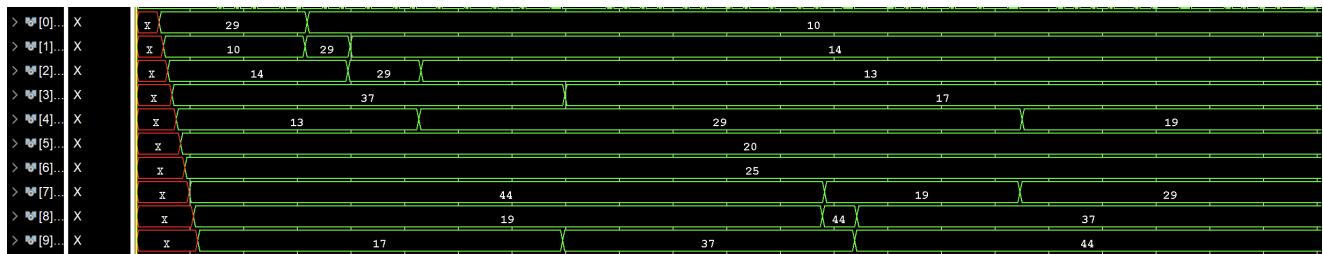
```
sw $4, 4($29)        # 栈[$29 + 4] = right
```

```
j main_loop          # 跳回主循环
```

end:

```
nop # 排序完成
```

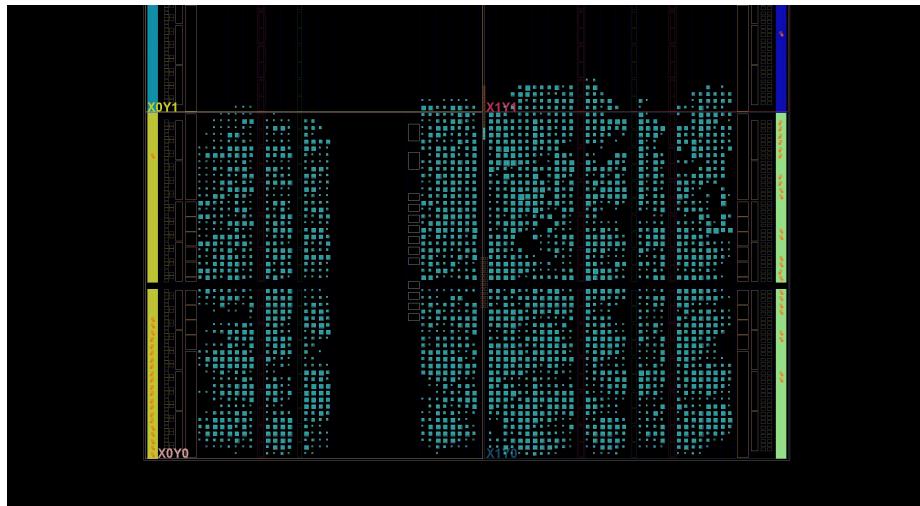
如上述汇编代码所展示，对0到9的ram地址赋予初值29, 10, 14, 37, 13, 20, 25, 44, 19, 17，然后通过CPU实现快速排序，这里采取显式栈代替递归的方式进行快速排序，用一个栈来保存需要排序的子数组的起始和结束索引（left和right），当栈不为空时，从栈中弹出一个子数组的边界索引（left 和 right），如果 left >= right，说明子数组长度为 0 或 1，无需排序，继续下一次循环，否则先选定right位置的元素值作为pivot，将小于pivot的元素放入左子组，将大于pivot的元素放入右子组，再将左右子组的开始结束索引放入栈中继续排序，当栈为空时，则认为排序结束。在这个过程中采用\$ 29即\$ sp 作为栈指针，初始化为内存地址200，通过栈指针的上下移动实现栈的大小改变，得到仿真波形图如下，如图所示，我们可以看到CPU很好的完成任务并且展示了快速排序的过程：



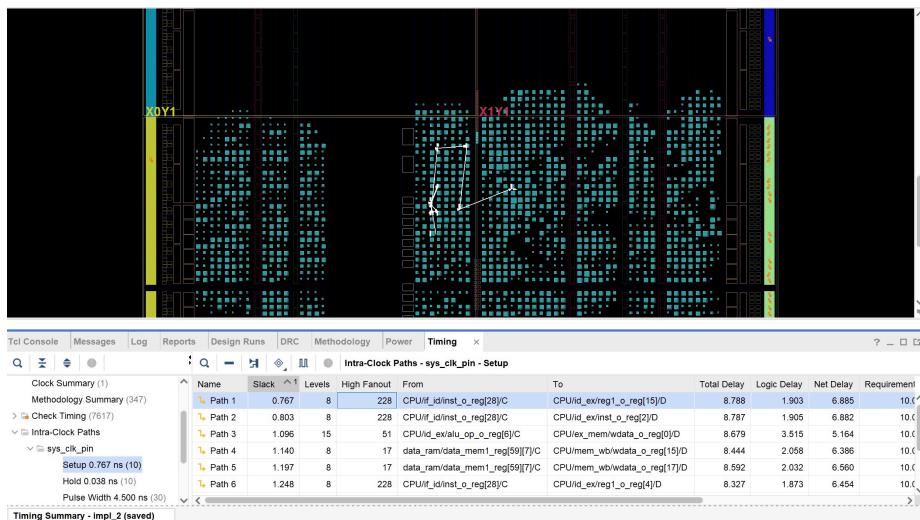
下板实现部分

(0)时序延迟报告

根据Vivado给出的 Implemented DRC Report 中的Device部分，可以看到该CPU设计在FPGA开发板上的空间占用率并不高，大约占据 $\frac{1}{3}$ 左右，意味着在信号传输时的信号相比于占据率更大的CPU会更快，也意味着可以将128个32位储存空间的RAM扩容，经测试最大可扩容至512个32位储存空间，如果为1024个32位储存空间将无法综合成功，具体板载情况如下：



在 Implemented DRC Report 中 Timing 部分可以看到在最坏情况下的最大延迟，在 Intra-Clock Paths 中选择 sys_clk_in 可以看到在使用较为复杂的快速排序作为下板案例时，最大时序延迟总和为 8.788ns，小于 100MHz 开发板的极限时序延迟 10ns，可以认为该 CPU 能够以 100MHz 的频率在开发板上正常运行，经过多次代码验证，最大时序延迟总和始终低于 9ns，因此可以认为该 CPU 的频率能够达到 110MHz，快速排序的最大延迟具体情况如下：



(1) 输出 n 以内的质数

该部分通过开关向 Risc 输入一个数 n，CPU 通过对 1 到 n 的所有数进行质数判断，若为质数，则输出该数，否则不输出，相邻两个质数输出等待约 0.2 秒便于观察，LED 灯每个时刻均显示开关输入方便查看，在判断质数时间外数码管显示为开关值，具体汇编代码如下：

```
# 初始化 FPGA 基地址
lui $1, 0x0000
ori $1, $1, 0xf000      # $1 = FPGA 基地址
main:
lw $15, 0x10($1)        # 从 FPGA 加载输入数 n
lw $13, 0x30($1)        # 加载按键按下情况
ori $14, $0, 0x0001
beq $13, $14, check     # 当按下 UP 按键时开始程序
j display
```

```

check:
    addi $2, $0, 2

loop:
    slt $16, $15, $2
    bne $16, $0, main      # 当检验到n后跳出检查程序

    # 边界检查: 如果 n < 2, 则不是素数
    addi $3, $0, 2          # $3 = 2
    slt $4, $2, $3          # 如果 n < 2, $4 = 1
    bne $4, $0, not_prime  # 如果 $4 != 0, 跳转到 not_prime

    # 初始化除数 i = 2
    addi $5, $0, 2          # $5 = i

check_loop:
    # 计算 i_squared = i * i (使用加法循环)
    add $6, $0, $0          # $6 = i_squared = 0
    add $7, $0, $5          # $7 = counter = i

i_square_loop:
    beq $7, $0, compare_i_n  # 如果 counter == 0, 跳转到 compare_i_n

    add $6, $6, $5          # i_squared += i
    addi $7, $7, -1          # counter -= 1
    j i_square_loop

compare_i_n:
    # 如果 i_squared > n, 跳转到 determine_prime
    slt $8, $2, $6          # 如果 n < i_squared, $8 = 1
    bne $8, $0, determine_prime  # 如果 n < i_squared, 跳转到
determine_prime

    # 不论 n 是否等于 i_squared, 都要检查 n 是否能被 i 整除
    # 检查 n 是否能被 i 整除
    add $9, $0, $2          # $9 = temp = n

mod_loop:
    sub $9, $9, $5          # temp -= i
    slt $10, $9, $0          # 如果 temp < 0, $10 = 1
    bne $10, $0, next_i     # 如果 temp < 0, 跳转到 next_i
    beq $9, $0, not_prime   # 如果 temp == 0, n 能被 i 整除, 非素数
    j mod_loop

next_i:
    # i += 1
    addi $5, $5, 1

    # 重新计算 i_squared
    j check_loop

```

```

determine_prime:
    # 没有找到因数, n 是素数
    j      is_prime

is_prime:
    sw $2, 0x20($1)
    addi $17, $0, 2000
delay_outer_p:
    addi $18, $0, 10000
delay_inner_p:
    addi $18, $18, -1
    bgtz $18, delay_inner_p
    addi $17, $17, -1
    bgtz $17, delay_outer_p

    addi $2, $2, 1
    j loop

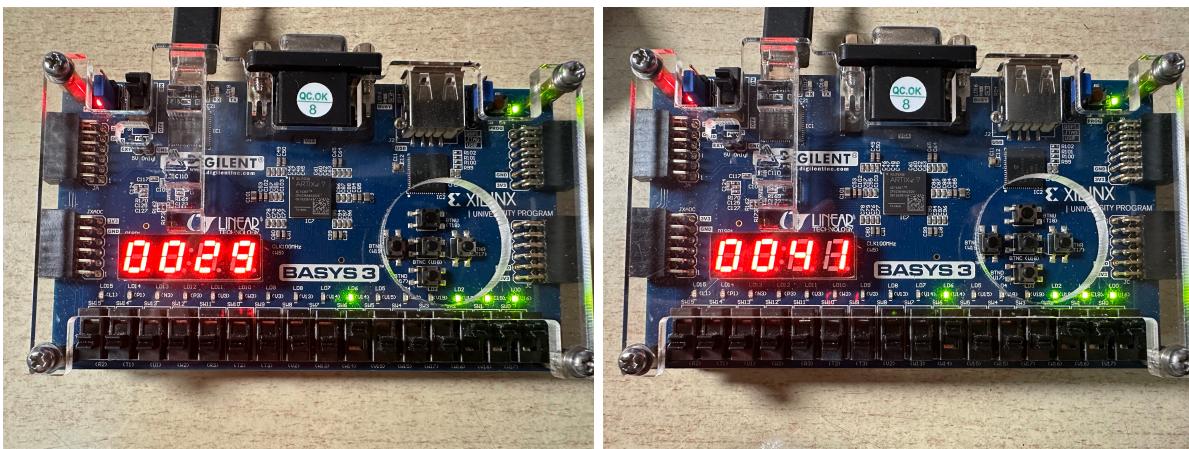
not_prime:
    addi $2, $2, 1
    j loop

display:
    sw $15, 0x0($1)
    sw $15, 0x20($1)
    j main

end_program:
    # 程序结束
    nop

```

下板结果如下，输入71，输出1到71中的全部质数，由于这里没办法直接展示视频，只能从中截取两种图片，可以看到，它们确实都是质数：



(2)对输入的任意个数进行快排序

该部分通过开关输入数，具体而言先用开关拨动到一个数，数码管会显示具体的数，然后按下Up按键进行保存，右边的LED灯闪烁表明保存成功，反复输入若干个数后，按下Left按键，第二个LED灯闪烁进行快速排序，然后从小到大输出在数码管中，相邻两个数间隔为0.2s，具体汇编代码如下：

```
# 初始化数组与FPGA基地址
addi $1, $0, 0          # $1 = 数组基地址 (RAM[0])
lui $17, 0x0000
ori $17, $17, 0xf000 # $17 = FPGA基地址

# 初始化数据计数器
addi $23, $0, 0          # $23 = 数据计数器

main:
    lw $18, 0x10($17)    # 加载输入数据
    lw $19, 0x30($17)    # 加载输入按键情况
    ori $20, $0, 0x0001
    beq $19, $20, input_finish
    nop
    ori $20, $0, 0x0002
    beq $19, $20, sort_begin

    j display

input_finish:
    sll $26, $23, 2
    add $26, $1, $26      # $26 = 数组基地址 + ($23 * 4)
    sw $18, 0($26)        # 储存数据进入RAM
    addi $23, $23, 1       # 数据计数器加1
    sw $20, 0x00($17)     # 闪烁LED灯表明输入完成
    addi $21, $0, 2000

delay_outer:
    addi $22, $0, 10000
delay_inner:
    addi $22, $22, -1
    bgtz $22, delay_inner

    addi $21, $21, -1
    bgtz $21, delay_outer

    sw $0, 0x00($17)      # 关闭LED灯闪烁
    j main

sort_begin:
    sw $20, 0x00($17)
    addi $23, $23, -1      # 调整数据计数器, $23 = 数组长度 - 1
    # 初始化栈指针和参数
```

```

addi $29, $0, 120      # $29 = 栈指针 (SP) , 初始化为内存地址 120
addi $3, $0, 0          # $3 = left, 数组左边界索引
add $4, $0, $23         # $4 = right, 数组右边界索引

# 将初始的 left 和 right 压入栈
addi $29, $29, -8      # $29 = $29 - 8, 栈指针向下移动
sw $3, 0($29)           # 栈[$29] = left
sw $4, 4($29)           # 栈[$29 + 4] = right

# 开始主循环
main_loop:
    # 判断栈是否为空 (栈指针是否回到了初始位置 800)
    addi $5, $29, 0        # $5 = $29 (当前栈指针)
    addi $6, $0, 120        # $6 = 120 (初始栈指针位置)
    beq $5, $6, output     # 如果 $29 == 120, 跳转到 output, 排序完成

    # 从栈中弹出 right 和 left
    lw $4, 4($29)           # $4 = 栈[$29 + 4], right
    lw $3, 0($29)           # $3 = 栈[$29], left
    addi $29, $29, 8         # $29 = $29 + 8, 栈指针向上移动

    # 如果 left >= right, 跳过本次循环
    slt $7, $3, $4           # $7 = ($3 < $4) ? 1 : 0
    beq $7, $0, main_loop    # 如果 $3 >= $4, 跳回 main_loop

    # 分区操作
    # 选择 pivot 为数组的 right 位置的元素
    sll $8, $4, 2             # $8 = $4 * 4
    add $8, $1, $8             # $8 = 数组基址 + right * 4
    lw $9, 0($8)               # $9 = pivot = 数组[right]

    # 初始化 i 为 left - 1
    addi $10, $3, -1          # $10 = i = left - 1

    # 初始化 j 为 left
    add $11, $0, $3            # $11 = j = left

partition_loop:
    # 判断 j 是否小于 right
    slt $7, $11, $4           # $7 = ($11 < $4) ? 1 : 0
    beq $7, $0, partition_end  # 如果 j >= right, 跳转到 partition_end

    # 获取数组[j]的值
    sll $12, $11, 2             # $12 = j * 4
    add $12, $1, $12            # $12 = 数组基址 + j * 4
    lw $13, 0($12)              # $13 = 数组[j]

    # 比较数组[j]和 pivot
    slt $7, $13, $9             # $7 = ($13 < $9) ? 1 : 0
    beq $7, $0, partition_loop_increment_j  # 如果数组[j] >= pivot, 跳过交换

```

```

# i = i + 1
addi $10, $10, 1      # $10 = i = i + 1

# 交换数组[i]和数组[j]
sll $14, $10, 2      # $14 = i * 4
add $14, $1, $14      # $14 = 数组基地址 + i * 4
lw $15, 0($14)        # $15 = 数组[i]

sw $13, 0($14)        # 数组[i] = 数组[j]
sw $15, 0($12)        # 数组[j] = 数组[i]

partition_loop_increment_j:
    # j = j + 1
    addi $11, $11, 1      # $11 = j = j + 1
    j partition_loop      # 跳回 partition_loop

partition_end:
    # 将 pivot 放到正确的位置
    addi $10, $10, 1      # $10 = i = i + 1
    sll $14, $10, 2      # $14 = i * 4
    add $14, $1, $14      # $14 = 数组基地址 + i * 4
    lw $15, 0($14)        # $15 = 数组[i]

    # 读取数组[right]的值
    sll $12, $4, 2        # $12 = right * 4
    add $12, $1, $12      # $12 = 数组基地址 + right * 4
    lw $13, 0($12)        # $13 = 数组[right] (pivot)

    # 交换数组[i]和数组[right]
    sw $13, 0($14)        # 数组[i] = 数组[right]
    sw $15, 0($12)        # 数组[right] = 数组[i]

    # pivot 索引为 i ($10)

    # 将左子数组的边界压入栈
    addi $7, $10, -1      # $7 = pivotIndex - 1
    slt $16, $3, $7        # $16 = ($3 < $7) ? 1 : 0
    beq $16, $0, skip_push_left  # 如果 left >= pivotIndex - 1, 跳过

    addi $29, $29, -8      # 栈指针向下移动
    sw $3, 0($29)          # 栈[$29] = left
    sw $7, 4($29)          # 栈[$29 + 4] = pivotIndex - 1

skip_push_left:
    # 将右子数组的边界压入栈
    addi $7, $10, 1        # $7 = pivotIndex + 1
    slt $16, $7, $4        # $16 = ($7 < $4) ? 1 : 0
    beq $16, $0, main_loop  # 如果 pivotIndex + 1 >= right, 跳回主循环

```

```

    addi $29, $29, -8      # 栈指针向下移动
    sw $7, 0($29)          # 栈[$29] = pivotIndex + 1
    sw $4, 4($29)          # 栈[$29 + 4] = right

    j main_loop             # 跳回主循环

display:
    sw $18, 0x20($17)     # 储存到数码管中
    j main

output:
    addi $24, $0, 0         # $24 = 计数器, 初始化为0
    addi $28, $23, 1         # $28 = 数组长度 ($23 + 1)

out_loop:
    slt $27, $24, $28       # $27 = ($24 < $28) ? 1 : 0
    beq $27, $0, end         # 如果 $24 >= 数组长度, 跳转到 end

    sll $26, $24, 2
    add $26, $1, $26         # $26 = 数组基地址 + ($24 * 4)
    lw $25, 0($26)           # $25 = 数组[$24]
    sw $25, 0x20($17)        # 储存在数码管

    # 延时
    addi $21, $0, 2000

delay_outer_o:
    addi $22, $0, 10000

delay_inner_o:
    addi $22, $22, -1
    bgtz $22, delay_inner_o
    addi $21, $21, -1
    bgtz $21, delay_outer_o

    addi $24, $24, 1         # $24 = $24 + 1
    j out_loop               # 跳回输出循环

end:
    nop
    j end

```

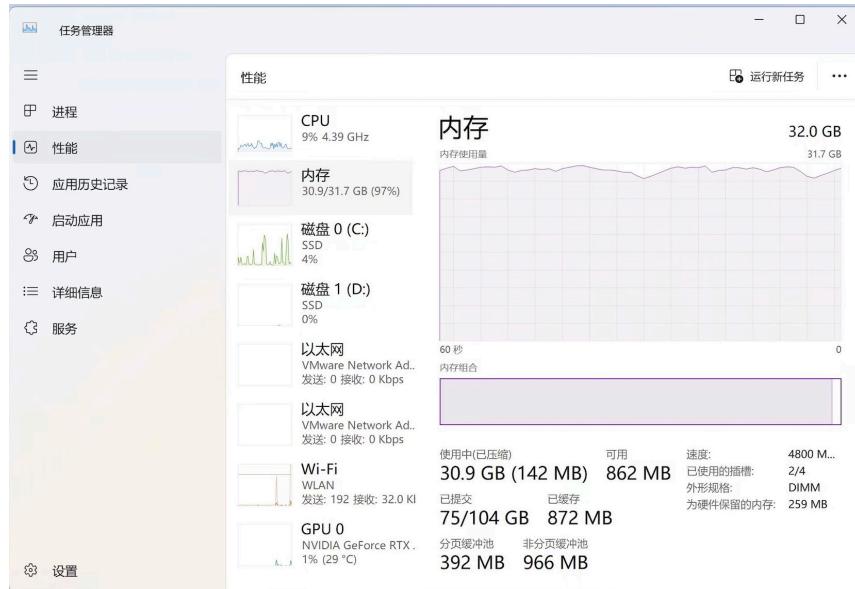
由于设计动态过程，在此不方便展示下板过程，故省略。

实验心得

(1)随机存取储存器RAM相关的各种问题

当添加RAM后，在初设RAM大小时对空间没有概念，导致出现了各种问题。最开始时，将RAM大小设置为1024个32位存储空间，在综合时发现需要耗费大量时间。当时我以为是CPU的代码量太大，属于正常情况，毕竟在模拟仿真的时候没有出现任何问题，于是我在宿舍挂着

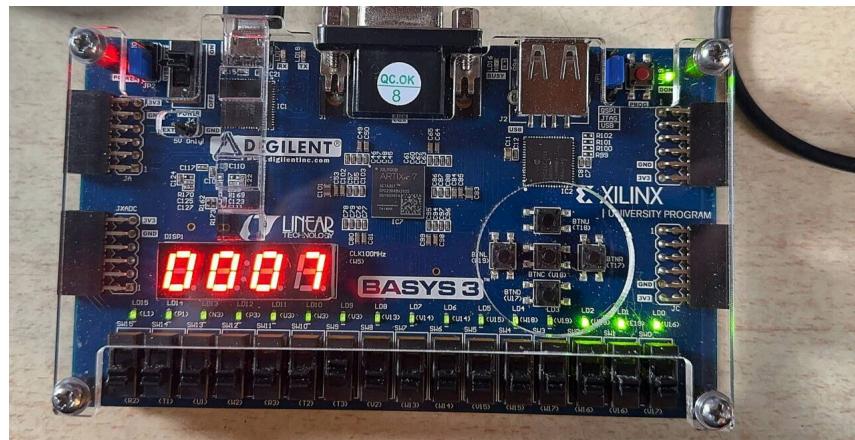
Vivado进行综合后就出去上课了，结果回来发现仍然没有综合结束，于是我调出任务管理器发现如下问题：



如上图所示，发现内存占用率极高并且提交内存接近100G，可以认为是炸内存了，当我在Vivado中的Tcl使用指令：`set_param general.maxThreads 1`使得单核运行，虽然有所改观但提交内存仍然达到了70G以上，仍然是处于炸内存的情况。最终发现问题是由RAM是随机存取，一共存在 1024×32 个触发器能够同时随机存取，这是非常恐怖的，因此我改成了 $\$512 \times 32$ 个触发器，即512个32位储存空间。

然而，这个时候还是出现了问题，能够正常综合、实现、生成比特流，在下板验证素数检验和素数输出功能时没有出现任何问题，但当下板验证快速排序时出现问题，输出的数出现了一些没有输入的数，并且输出只输出了两个数，经不断调试发现问题如下：在实现阶段的布线上，纵向布线已经完全填满，可能存在RAM重叠之类的问题，而素数检验只需要使用1个RAM空间所以没有发现问题，而快速排序需要将排序数放入RAM中，需要用到较多的RAM因此出现上述问题，因此我将RAM设置为256个32位储存空间，下板后没有任何问题，并在布线图上可以看到纵向占用率较低。

此外，在这个阶段还出现过亚稳态的问题，最初RAM为了能够立刻读取，我错误地将读写功能均采取组合逻辑电路设置，尽管在组合逻辑中均使用寄存器reg来存储数据，但依然出现了不少问题，具体表现如下：



如上图所示，LED灯存在弱光和强光两种情况，真实显示应该为弱光部分，但是强光部分一直显示干扰视野（并且强光这部分显示也不是我们想看到的），当我将写入部分改为时序逻辑后解决问题，这也说明了写入时采用时序设计稳定数据，读取时采用组合设计快速读取的正确性。

写入部分代码采用时序设计电路，并且考虑到在快速排序等问题中需要反复使用RAM模块，因此添加了复位清零的信号，防止出现RAM的不定态影响CPU本体，具体代码如下：

```
// Write operation
integer j;
always @ (posedge clk, posedge rst) begin
    if (rst == `Rst_EN) begin
        for (j = 0; j <= 127; j = j + 1) begin
            data_mem0[j] <= 8'h0;
            data_mem1[j] <= 8'h0;
            data_mem2[j] <= 8'h0;
            data_mem3[j] <= 8'h0;
        end
    end else begin
        if (ce_i == `Chip_DIS) begin
            // Chip disabled, no operation
        end else if (we_i == `Write_EN) begin
            if (sel_i[3] == 1'b1) begin
                data_mem3[addr_i[`Data_Mem_Num_Log2 + 1 : 2]] <=
data_i[31:24];
            end
            if (sel_i[2] == 1'b1) begin
                data_mem2[addr_i[`Data_Mem_Num_Log2 + 1 : 2]] <=
data_i[23:16];
            end
            if (sel_i[1] == 1'b1) begin
                data_mem1[addr_i[`Data_Mem_Num_Log2 + 1 : 2]] <=
data_i[15:8];
            end
            if (sel_i[0] == 1'b1) begin
                data_mem0[addr_i[`Data_Mem_Num_Log2 + 1 : 2]] <=
data_i[7:0];
            end
        end
    end
end
end
```

读取部分较为简单，为了能够即时读取数据，因此直接访问目标地址的数据并存入 data_o 中，具体的代码如下：

```
// Read operation
always @ (*) begin
    if (ce_i == `Chip_DIS) begin
```

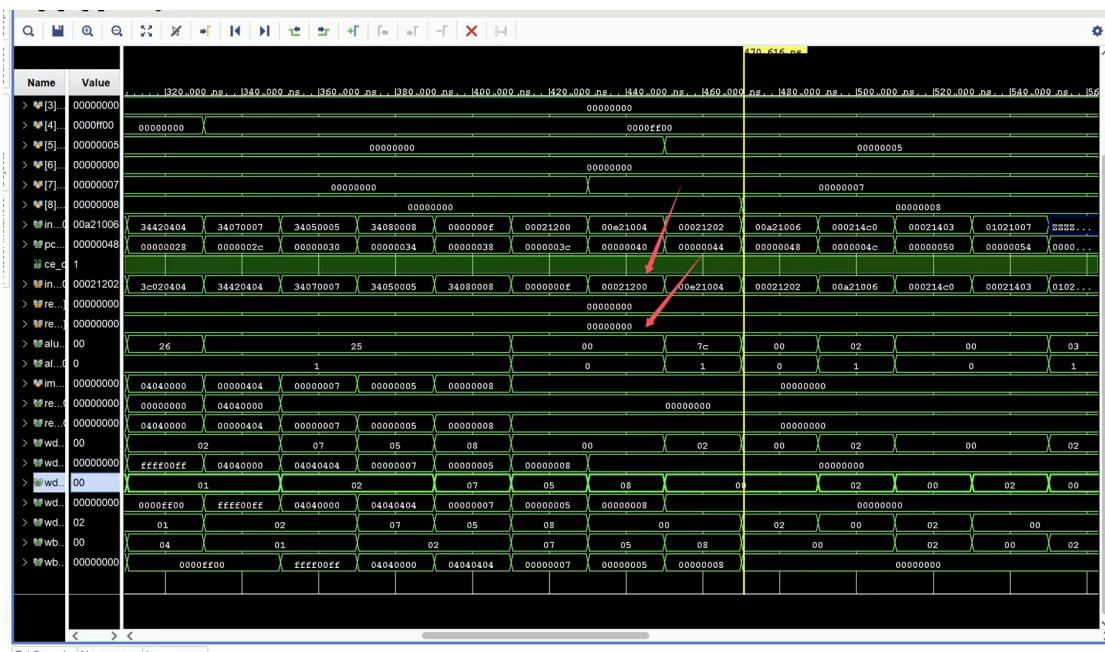
```

        data_o = `Word_Zero;
end else if (we_i == `Write_DIS) begin
    data_o = {data_mem3[addr_i[`Data_Mem_Num_Log2 + 1 : 2]],
              data_mem2[addr_i[`Data_Mem_Num_Log2 + 1 : 2]],
              data_mem1[addr_i[`Data_Mem_Num_Log2 + 1 : 2]],
              data_mem0[addr_i[`Data_Mem_Num_Log2 + 1 : 2]}};
end else begin
    data_o = `Word_Zero;
end
end

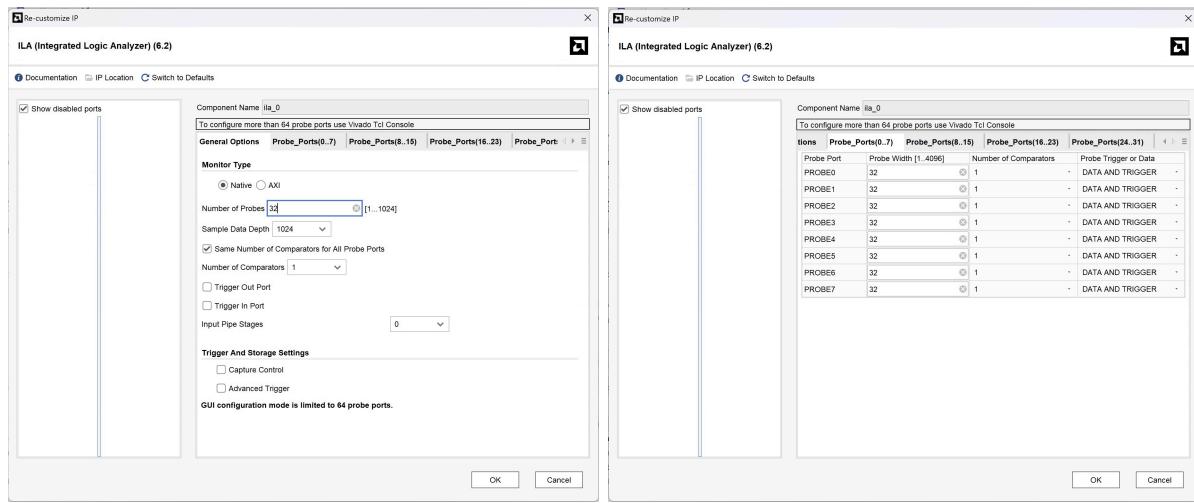
```

(2)CPU调试相关过程

在没有下板时，主要采取行为仿真进行调试，具体而言是通过指令写入功能，而仿真测试文件 `cpu_tb.v` 单纯实现复位后等待的功能。在调试时，打开 `Simulation` 的波形图，通过添加不同模块中的具体变量观察每个变量的变化情况，最主要的观察对象是：源寄存器rs和rt在各个阶段的值，alu指令码，写回数据和使能等，主要是在端口的输入输出接口上进行观察，具体而言如下所示（下面展示的是早期CPU调试截图）：



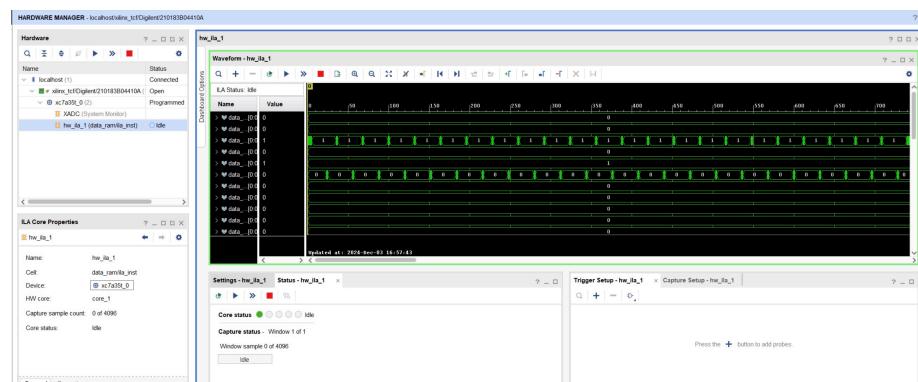
此外，在下板后由于每次生成从综合到比特流需要较长时间，不方便反复调试，因此在后期我采用了 `ila` 这个IP核进行调试，具体使用方法如下：在IP核中搜索 `ILA` 找到 `Integrated Logic Analyzer`，设置为 `Number of Probes` 为调试需要查看的数据数量，深度由于板载限制将 `Sample Data Depth` 设置为 1024（`ILA` 核非常耗费板载资源，在测试的时候需要将 RAM 储存大小压到 32 个 32 位储存空间比较保险），具体的情况如下图展示所示，左图为概览，右图为具体接口调整：



如上图所示，调整好上述数据后在后面的Probe_Port设置具体的数据位宽，这里比较麻烦，需要手动输入每一个的位宽，这里全部写为32（因为RAM的储存是32位宽），使用方法也较为简单，在RAM模块中添加ILA核，具体代码如下(为避免赘述，只展示到前8位的RAM，可以继续拓展到32位)：

```
//Instantiate ILA IP core and connect each data_mem_combined element to a
probe
ila_0 ila_inst (
    .clk(clk),
    .probe0(data_mem_combined[0]),
    .probe1(data_mem_combined[1]),
    .probe2(data_mem_combined[2]),
    .probe3(data_mem_combined[3]),
    .probe4(data_mem_combined[4]),
    .probe5(data_mem_combined[5]),
    .probe6(data_mem_combined[6]),
    .probe7(data_mem_combined[7])
);
```

在生成比特流时，还会生成测试文件，在烧录过程中Vivado会自动添加测试文件（但是Vivado不会自动删除，所以如果不需要ila核后需要在烧录的时候手动删除一次，之后就不会出现了），烧录完毕后会弹出窗口显示从FPGA板采集到的数据波形，点击三角形运行符号开始采集，如果需要继续采集，只需要再点一次三角形运行符号重新采集，具体如下所示（图为早期测试下板是否能够进行的情况）：



(3)分支判断相关问题

在本次实验中，最开始我对分支语句一般都是在 `if-else` 分支中添加`else`语句防止出现问题，在 `case` 分支中添加 `default` 防止出现问题。然而，这也为我带来了很多麻烦，在译码器阶段和储存器单元中，很多时候在`if`和`else if`或者`case`中不存在匹配的情况下应该是保持原来的值不变，达到一个“锁存器”的目的，但我总是在过程中误以为不匹配的情况下赋予复位值，因此造成了许多问题。

这个问题发生最严重的地方在于：流水线暂停时保持数据不变而不是清空，这个代码问题困扰我许久，最后发现问题在分支判断语句上。虽然为了规范角度，添加`else`和`default`是有必要的，但是在实际代码中由于可拓展性和锁存器的必要性，或许在代码写作过程中不添加这个也是可以的（添加当然是更好的，但是需要明确每一步的剩余情况应该对数据进行何种处理）。

可改进部分

(1)数码管显示模块

在此前的VGA设计作业中，分数score的显示我是通过 `Double Dabble` 算法将二进制数转换为bcd码的形式显示在VGA屏上，但是在本次实验中我采用的是直接取模和取商的方式得到每一位的bcd码。由于本次数码管操作频率不高，可能需要十几条甚至更多指令的运行时间后，才修改一次数码管的值，所以用除法和取模并没有造成时序问题，但是为了更稳妥的操作，用 `Double Dabble` 的形式写一个组合设计或许是较好的处理，但由于时间限制，这个并没有完成。

(2)分支预测器

在本实验中最初根据暑假在 `hdlbits` 上的代码写了一份动态分支预测，然而不幸的是由于实验测试的指令码过少（基本都是100条以内），并且排序过程中的分支跳转较为不适合预测，因此预测准确率极低（大约为60%左右）。同时，由于分支预测的复杂性，在不做任何时序修改的情况下添加进CPU后会导致时序延迟增加到13ns以上，无法完成100MHz的目标，由于时间限制最后放弃了动态分支预测的添加，选择使用传统的延迟槽的方式处理分支语句。在时序处理方面，可以考虑添加一级流水线解决问题，但是由于时间限制，我最终放弃了这个想法。

(3)Uart传输

目前的CPU需要在综合前前提通过ROM的IP核或者LUT提前储存指令数据较为不方便调试，通过Uart传输可以在电脑上使用串口调试软件进行指令收发，较为方便，但是由于时间限制，我未能完成这一部分的实现。

附录

(1)汇编语言转32位机器指令码的Python脚本

通过下述代码可以从 `input.asm` 读入汇编代码，并将转换后的机器码输出到 `output.txt` 文件中，输入格式较为自由，允许在分支指令和跳转指令直接跳到某一模块，而不需要手动计算跳跃的位置和距离，更为方便；输入通用寄存器以\$开头，允许数字寄存器的输入方式和命名寄存器的输入方式。

```

import re

instruction_set = {

    # R-type instructions
    'and': {'type': 'R', 'opcode': '0b000000', 'funct': '0b100100'},
    'or': {'type': 'R', 'opcode': '0b000000', 'funct': '0b100101'},
    'xor': {'type': 'R', 'opcode': '0b000000', 'funct': '0b100110'},
    'nor': {'type': 'R', 'opcode': '0b000000', 'funct': '0b100111'},
    'sll': {'type': 'R', 'opcode': '0b000000', 'funct': '0b000000'},
    'sllv': {'type': 'R', 'opcode': '0b000000', 'funct': '0b000100'},
    'srl': {'type': 'R', 'opcode': '0b000000', 'funct': '0b000010'},
    'srlv': {'type': 'R', 'opcode': '0b000000', 'funct': '0b000110'},
    'sra': {'type': 'R', 'opcode': '0b000000', 'funct': '0b000011'},
    'srav': {'type': 'R', 'opcode': '0b000000', 'funct': '0b000111'},
    'sync': {'type': 'R', 'opcode': '0b000000', 'funct': '0b001111'},
    'pref': {'type': 'R', 'opcode': '0b000000', 'funct': '0b110011'},
    'nop': {'type': 'R', 'opcode': '0b000000', 'funct': '0b000000'},
    'movz': {'type': 'R', 'opcode': '0b000000', 'funct': '0b001010'},
    'movn': {'type': 'R', 'opcode': '0b000000', 'funct': '0b001011'},
    'mfhi': {'type': 'R', 'opcode': '0b000000', 'funct': '0b010000'},
    'mthi': {'type': 'R', 'opcode': '0b000000', 'funct': '0b010001'},
    'mflo': {'type': 'R', 'opcode': '0b000000', 'funct': '0b010010'},
    'mtlo': {'type': 'R', 'opcode': '0b000000', 'funct': '0b010011'},
    'slt': {'type': 'R', 'opcode': '0b000000', 'funct': '0b101010'},
    'sltu': {'type': 'R', 'opcode': '0b000000', 'funct': '0b101011'},
    'add': {'type': 'R', 'opcode': '0b000000', 'funct': '0b100000'},
    'addu': {'type': 'R', 'opcode': '0b000000', 'funct': '0b100001'},
    'sub': {'type': 'R', 'opcode': '0b000000', 'funct': '0b100010'},
    'subu': {'type': 'R', 'opcode': '0b000000', 'funct': '0b100011'},
    'clz': {'type': 'R', 'opcode': '0b011100', 'funct': '0b100000'},
    'clo': {'type': 'R', 'opcode': '0b011100', 'funct': '0b100001'},
    'mult': {'type': 'R', 'opcode': '0b000000', 'funct': '0b011000'},
    'multu': {'type': 'R', 'opcode': '0b000000', 'funct': '0b011001'},
    'mul': {'type': 'R', 'opcode': '0b011100', 'funct': '0b000010'},
    'jalr': {'type': 'R', 'opcode': '0b000000', 'funct': '0b001001'},
    'jr': {'type': 'R', 'opcode': '0b000000', 'funct': '0b001000'},

    # I-type instructions
    'andi': {'type': 'I', 'opcode': '0b001100'},
    'ori': {'type': 'I', 'opcode': '0b001101'},
    'xori': {'type': 'I', 'opcode': '0b001110'},
    'lui': {'type': 'I', 'opcode': '0b001111'},
    'slti': {'type': 'I', 'opcode': '0b001010'},
    'sltiu': {'type': 'I', 'opcode': '0b001011'},
    'addi': {'type': 'I', 'opcode': '0b001000'},
    'addiu': {'type': 'I', 'opcode': '0b001001'},
    'beq': {'type': 'I', 'opcode': '0b000100'},
    'bne': {'type': 'I', 'opcode': '0b000101'},
    'bgtz': {'type': 'I', 'opcode': '0b000111'},
}

```

```

'blez': {'type': 'I', 'opcode': 0b000110},
'lb': {'type': 'I', 'opcode': 0b100000},
'lbu': {'type': 'I', 'opcode': 0b100100},
'lh': {'type': 'I', 'opcode': 0b100001},
'lhu': {'type': 'I', 'opcode': 0b100101},
'lw': {'type': 'I', 'opcode': 0b100011},
'sb': {'type': 'I', 'opcode': 0b101000},
'sh': {'type': 'I', 'opcode': 0b101001},
'sw': {'type': 'I', 'opcode': 0b101011},

# J-type instructions
'j': {'type': 'J', 'opcode': 0b000010},
'jal': {'type': 'J', 'opcode': 0b000011},
}

registers = {
    '$zero': 0,
    '$0': 0,
    '$1': 1,
    '$2': 2,
    '$3': 3,
    '$4': 4,
    '$5': 5,
    '$6': 6,
    '$7': 7,
    '$8': 8,
    '$9': 9,
    '$10': 10,
    '$11': 11,
    '$12': 12,
    '$13': 13,
    '$14': 14,
    '$15': 15,
    '$16': 16,
    '$17': 17,
    '$18': 18,
    '$19': 19,
    '$20': 20,
    '$21': 21,
    '$22': 22,
    '$23': 23,
    '$24': 24,
    '$25': 25,
    '$26': 26,
    '$27': 27,
    '$28': 28,
    '$29': 29,
    '$30': 30,
    '$31': 31,
    '$at': 1,
}

```

```
'$v0': 2,
'$v1': 3,
'$a0': 4,
'$a1': 5,
'$a2': 6,
'$a3': 7,
'$t0': 8,
'$t1': 9,
'$t2': 10,
'$t3': 11,
'$t4': 12,
'$t5': 13,
'$t6': 14,
'$t7': 15,
'$s0': 16,
'$s1': 17,
'$s2': 18,
'$s3': 19,
'$s4': 20,
'$s5': 21,
'$s6': 22,
'$s7': 23,
'$t8': 24,
'$t9': 25,
'$k0': 26,
'$k1': 27,
'$gp': 28,
'$sp': 29,
'$fp': 30,
'$ra': 31,
}


```

```
def parse_register(reg):
    reg = reg.strip()
    if reg in registers:
        return registers[reg]
    else:
        raise ValueError(f"未知寄存器 {reg}")


```

```
def assemble_file(input_file, output_file):
    # 第一遍：解析标签，建立标签与地址的映射
    labels = {}
    instructions = []
    with open(input_file, 'r') as f_in:
        addr = 0
        for line in f_in:
            line = line.split('#')[0].strip()
            if not line:
                continue
            if ':' in line:

```

```

        # 存在标签
        label, rest = line.split(':', 1)
        label = label.strip()
        labels[label] = addr
        line = rest.strip()
        if not line:
            continue
        instructions.append(line)
        addr += 1 # 假设每条指令占用 1 单位地址

# 第二遍：解析指令，生成机器码
with open(output_file, 'w') as f_out:
    addr = 0
    for line in instructions:
        try:
            machine_code = assemble_instruction(line, labels, addr)
            if machine_code:
                f_out.write(machine_code + '\n')
            addr += 1
        except ValueError as e:
            print(f"错误: '{line.strip()}' : {e}")

def assemble_instruction(line, labels, addr):
    tokens = line.replace(',', ' ').split()
    if not tokens:
        return None
    op = tokens[0]
    if op in labels:
        tokens = tokens[1:]
        op = tokens[0]
    if op not in instruction_set:
        raise ValueError(f"不支持的指令 {op}")
    instr = instruction_set[op]
    opcode = instr['opcode']
    if instr['type'] == 'R':
        # R 型指令处理
        funct = instr['funct']
        shamt = 0
        if op in ['sll', 'srl', 'sra']:
            rd = parse_register(tokens[1])
            rt = parse_register(tokens[2])
            shamt = int(tokens[3], 0) & 0x1F
            rs = 0
        elif op in ['jr']:
            rs = parse_register(tokens[1])
            rt = 0
            rd = 0
        elif op in ['mfhi', 'mflo']:
            rd = parse_register(tokens[1])
            rs = 0

```

```

        rt = 0
    elif op in ['mthi', 'mtlo']:
        rs = parse_register(tokens[1])
        rt = 0
        rd = 0
    elif op in ['nop']:
        rs = rt = rd = shamt = funct = 0
    else:
        rd = parse_register(tokens[1])
        rs = parse_register(tokens[2])
        rt = parse_register(tokens[3])
    machine_code = (opcode << 26) | (rs << 21) | (rt << 16) | \
                    (rd << 11) | (shamt << 6) | funct
elif instr['type'] == 'I':
    rt = parse_register(tokens[1])
    if op in ['lui']:
        immediate = int(tokens[2], 0) & 0xFFFF
        rs = 0
    elif op in ['beq', 'bne']:
        rs = parse_register(tokens[1])
        rt = parse_register(tokens[2])
        label = tokens[3]
        if label in labels:
            offset = labels[label] - (addr +1)
            immediate = offset & 0xFFFF
        else:
            raise ValueError(f"未知的标签 {label}")
    elif op in ['blez', 'bgtz']:
        rs = parse_register(tokens[1])
        rt = 0
        label = tokens[2]
        if label in labels:
            offset = labels[label] - (addr +1)
            immediate = offset & 0xFFFF
        else:
            raise ValueError(f"未知的标签 {label}")
    elif op in ['addi', 'addiu', 'andi', 'ori', 'xori', 'slti',
'sltiu']:
        rs = parse_register(tokens[2])
        immediate = int(tokens[3], 0) & 0xFFFF
    elif op in ['lb', 'lbu', 'lh', 'lhu', 'lw', 'sb', 'sh', 'sw']:
        # 内存访问指令
        mem = tokens[2]
        match = re.match(r'(-?(?:0x)?[0-9a-fA-F]+)\$([a-zA-Z0-9]+)', mem)
        if match:
            immediate_str = match.group(1)
            immediate = int(immediate_str, 0) & 0xFFFF
            rs = parse_register(match.group(2))
        else:

```

```

        raise ValueError(f"地址格式错误 {tokens[2]}")
    else:
        rs = parse_register(tokens[2])
        immediate = int(tokens[3], 0) & 0xFFFF
        machine_code = (opcode << 26) | (rs << 21) | (rt << 16) | immediate
    elif instr['type'] == 'J':
        label = tokens[1]
        if label in labels:
            address = labels[label] & 0xFFFFFFFF
        else:
            raise ValueError(f"未知的标签 {label}")
        machine_code = (opcode << 26) | address
    else:
        raise ValueError(f"不支持的指令类型 {instr['type']}"))
return f"{machine_code}:08x"

if __name__ == '__main__':
    input_file = 'input.asm'    # 输入文件名
    output_file = 'output.txt' # 输出文件名
    assemble_file(input_file, output_file)

```

(2)将机器码转换为ROM模块LUT储存脚本

从8位16进制机器码输入文件 1.txt，再将 assign 形式的LUT储存代码输出到 output.v 文件中，具体代码如下：

```

def convert_to_verilog(input_file, output_file):
    # 读取输入文件中的十六进制机器码
    with open(input_file, 'r') as infile:
        lines = infile.readlines()

    # 打开输出文件
    with open(output_file, 'w') as outfile:
        # 循环处理每一行
        for i, line in enumerate(lines):
            # 去掉行末的空白符和换行符
            line = line.strip()
            # 写入 Verilog 格式的输出
            outfile.write(f"    assign inst_mem[{i}] = 32'h{line};\n")

    # 调用函数，输入文件名为 '1.txt'，输出文件名为 'output.v'
convert_to_verilog('1.txt', 'output.v')

```

(3)素数输出指令的LUT

```

assign inst_mem[0] = 32'h3c010000;
assign inst_mem[1] = 32'h3421f000;
assign inst_mem[2] = 32'h8c2f0010;

```

```
assign inst_mem[3] = 32'h8c2d0030;
assign inst_mem[4] = 32'h201d0000;
assign inst_mem[5] = 32'h340e0001;
assign inst_mem[6] = 32'h11ae0003;
assign inst_mem[7] = 32'h201d0000;
assign inst_mem[8] = 32'h0800003b;
assign inst_mem[9] = 32'h201d0000;
assign inst_mem[10] = 32'h20020002;
assign inst_mem[11] = 32'h01e2802a;
assign inst_mem[12] = 32'h1600ffff5;
assign inst_mem[13] = 32'h201d0000;
assign inst_mem[14] = 32'h20030002;
assign inst_mem[15] = 32'h0043202a;
assign inst_mem[16] = 32'h14800027;
assign inst_mem[17] = 32'h201d0000;
assign inst_mem[18] = 32'h20050002;
assign inst_mem[19] = 32'h00003020;
assign inst_mem[20] = 32'h00053820;
assign inst_mem[21] = 32'h10e00005;
assign inst_mem[22] = 32'h201d0000;
assign inst_mem[23] = 32'h00c53020;
assign inst_mem[24] = 32'h20e7ffff;
assign inst_mem[25] = 32'h08000015;
assign inst_mem[26] = 32'h201d0000;
assign inst_mem[27] = 32'h0046402a;
assign inst_mem[28] = 32'h1500000d;
assign inst_mem[29] = 32'h201d0000;
assign inst_mem[30] = 32'h00024820;
assign inst_mem[31] = 32'h01254822;
assign inst_mem[32] = 32'h0120502a;
assign inst_mem[33] = 32'h15400005;
assign inst_mem[34] = 32'h201d0000;
assign inst_mem[35] = 32'h11200014;
assign inst_mem[36] = 32'h201d0000;
assign inst_mem[37] = 32'h0800001f;
assign inst_mem[38] = 32'h201d0000;
assign inst_mem[39] = 32'h20a50001;
assign inst_mem[40] = 32'h08000013;
assign inst_mem[41] = 32'h201d0000;
assign inst_mem[42] = 32'h0800002c;
assign inst_mem[43] = 32'h201d0000;
assign inst_mem[44] = 32'hac220020;
assign inst_mem[45] = 32'h201107d0;
assign inst_mem[46] = 32'h20122710;
assign inst_mem[47] = 32'h2252ffff;
assign inst_mem[48] = 32'h1e40ffffe;
assign inst_mem[49] = 32'h201d0000;
assign inst_mem[50] = 32'h2231ffff;
assign inst_mem[51] = 32'h1e20ffffa;
assign inst_mem[52] = 32'h201d0000;
```

```
assign inst_mem[53] = 32'h20420001;
assign inst_mem[54] = 32'h0800000b;
assign inst_mem[55] = 32'h201d0000;
assign inst_mem[56] = 32'h20420001;
assign inst_mem[57] = 32'h0800000b;
assign inst_mem[58] = 32'h201d0000;
assign inst_mem[59] = 32'hac2f0000;
assign inst_mem[60] = 32'hac2f0020;
assign inst_mem[61] = 32'h08000002;
assign inst_mem[62] = 32'h201d0000;
assign inst_mem[63] = 32'h00000000;
```

(4)快速排序指令的LUT

```
assign inst_mem[0] = 32'h20010000;
assign inst_mem[1] = 32'h3c110000;
assign inst_mem[2] = 32'h3631f000;
assign inst_mem[3] = 32'h20170000;
assign inst_mem[4] = 32'h8e320010;
assign inst_mem[5] = 32'h8e330030;
assign inst_mem[6] = 32'h34140001;
assign inst_mem[7] = 32'h12740006;
assign inst_mem[8] = 32'h00000000;
assign inst_mem[9] = 32'h34140002;
assign inst_mem[10] = 32'h12740013;
assign inst_mem[11] = 32'h20000000;
assign inst_mem[12] = 32'h08000060;
assign inst_mem[13] = 32'h20000000;
assign inst_mem[14] = 32'h0017d080;
assign inst_mem[15] = 32'h003ad020;
assign inst_mem[16] = 32'haf520000;
assign inst_mem[17] = 32'h22f70001;
assign inst_mem[18] = 32'hae340000;
assign inst_mem[19] = 32'h201507d0;
assign inst_mem[20] = 32'h20162710;
assign inst_mem[21] = 32'h22d6ffff;
assign inst_mem[22] = 32'h1ec0ffe;
assign inst_mem[23] = 32'h20000000;
assign inst_mem[24] = 32'h22b5ffff;
assign inst_mem[25] = 32'h1ea0fffa;
assign inst_mem[26] = 32'h20000000;
assign inst_mem[27] = 32'hae200000;
assign inst_mem[28] = 32'h08000004;
assign inst_mem[29] = 32'h20000000;
assign inst_mem[30] = 32'hae340000;
assign inst_mem[31] = 32'h22f7ffff;
assign inst_mem[32] = 32'h201d0078;
assign inst_mem[33] = 32'h20030000;
assign inst_mem[34] = 32'h00172020;
```

```
assign inst_mem[35] = 32'h23bdffff8;
assign inst_mem[36] = 32'hafa30000;
assign inst_mem[37] = 32'hafa40004;
assign inst_mem[38] = 32'h23a50000;
assign inst_mem[39] = 32'h20060078;
assign inst_mem[40] = 32'h10a6003a;
assign inst_mem[41] = 32'h200000000;
assign inst_mem[42] = 32'h8fa40004;
assign inst_mem[43] = 32'h8fa30000;
assign inst_mem[44] = 32'h23bd0008;
assign inst_mem[45] = 32'h0064382a;
assign inst_mem[46] = 32'h10e0ffff7;
assign inst_mem[47] = 32'h200000000;
assign inst_mem[48] = 32'h00044080;
assign inst_mem[49] = 32'h00284020;
assign inst_mem[50] = 32'h8d090000;
assign inst_mem[51] = 32'h206affff;
assign inst_mem[52] = 32'h00035820;
assign inst_mem[53] = 32'h0164382a;
assign inst_mem[54] = 32'h10e00010;
assign inst_mem[55] = 32'h200000000;
assign inst_mem[56] = 32'h000b6080;
assign inst_mem[57] = 32'h002c6020;
assign inst_mem[58] = 32'h8d8d0000;
assign inst_mem[59] = 32'h01a9382a;
assign inst_mem[60] = 32'h10e00007;
assign inst_mem[61] = 32'h200000000;
assign inst_mem[62] = 32'h214a0001;
assign inst_mem[63] = 32'h000a7080;
assign inst_mem[64] = 32'h002e7020;
assign inst_mem[65] = 32'h8dcf0000;
assign inst_mem[66] = 32'hadcd0000;
assign inst_mem[67] = 32'had8f0000;
assign inst_mem[68] = 32'h216b0001;
assign inst_mem[69] = 32'h08000035;
assign inst_mem[70] = 32'h200000000;
assign inst_mem[71] = 32'h214a0001;
assign inst_mem[72] = 32'h000a7080;
assign inst_mem[73] = 32'h002e7020;
assign inst_mem[74] = 32'h8dcf0000;
assign inst_mem[75] = 32'h00046080;
assign inst_mem[76] = 32'h002c6020;
assign inst_mem[77] = 32'h8d8d0000;
assign inst_mem[78] = 32'hadcd0000;
assign inst_mem[79] = 32'had8f0000;
assign inst_mem[80] = 32'h2147ffff;
assign inst_mem[81] = 32'h0067802a;
assign inst_mem[82] = 32'h12000004;
assign inst_mem[83] = 32'h200000000;
assign inst_mem[84] = 32'h23bdffff8;
```

```
assign inst_mem[85] = 32'hafa30000;
assign inst_mem[86] = 32'hafa70004;
assign inst_mem[87] = 32'h21470001;
assign inst_mem[88] = 32'h00e4802a;
assign inst_mem[89] = 32'h1200ffcc;
assign inst_mem[90] = 32'h200000000;
assign inst_mem[91] = 32'h23bdffff8;
assign inst_mem[92] = 32'hafa70000;
assign inst_mem[93] = 32'hafa40004;
assign inst_mem[94] = 32'h08000026;
assign inst_mem[95] = 32'h200000000;
assign inst_mem[96] = 32'hae320020;
assign inst_mem[97] = 32'h08000004;
assign inst_mem[98] = 32'h200000000;
assign inst_mem[99] = 32'h20180000;
assign inst_mem[100] = 32'h22fc0001;
assign inst_mem[101] = 32'h031cd82a;
assign inst_mem[102] = 32'h13600010;
assign inst_mem[103] = 32'h200000000;
assign inst_mem[104] = 32'h0018d080;
assign inst_mem[105] = 32'h003ad020;
assign inst_mem[106] = 32'h8f590000;
assign inst_mem[107] = 32'hae390020;
assign inst_mem[108] = 32'h201507d0;
assign inst_mem[109] = 32'h20162710;
assign inst_mem[110] = 32'h22d6ffff;
assign inst_mem[111] = 32'h1ec0ffe;
assign inst_mem[112] = 32'h200000000;
assign inst_mem[113] = 32'h22b5ffff;
assign inst_mem[114] = 32'h1ea0ffa;
assign inst_mem[115] = 32'h200000000;
assign inst_mem[116] = 32'h23180001;
assign inst_mem[117] = 32'h08000065;
assign inst_mem[118] = 32'h200000000;
assign inst_mem[119] = 32'h000000000;
assign inst_mem[120] = 32'h08000077;
```