前言

第一次听到RISC-V这个词大概是两年前，当时觉得它也就是和MIPS这些CPU架构没什么区别，因此也就不以为然了。直到去年，RISC-V这个词开始频繁地出现在微信和其他网站上，此时我再也不能无动于衷了，于是开始在网上搜索有关它的资料，开始知道有SiFive这个网站，知道SiFive出了好几款RISC-V的开发板。可是最便宜的那一块开发板都要700多RMB，最后还是忍痛入手了一块。由于平时上班比较忙，所以玩这块板子的时间并不多，也就是晚上下班后和周末玩玩，自己照着芯片手册写了几个例程在板子上跑跑而已。

再后来发现网上已经有如何设计RISC-V处理器的书籍卖了，并且这个处理器是开源的，于是果断买了一本来阅读并浏览了它的开源代码，最后表示看不懂。从那之后一个“从零开始写RISC-V处理器”的想法开始不断地出现在我的脑海里。我心里是很想学习、深入研究RISC-V的，但是一直以来都没有verilog和FPGA的基础，可以说是CPU设计领域里的门外汉，再加上很少业余时间，为此一度犹豫不决。但是直觉告诉我已近不能再等了，我决定开始自学verilog和FPGA，用简单易懂的方式写一个RISC-V处理器并且把它开源出来，在提高自身的同时希望能帮助到那些想入门RISC-V的同学，于是tinyriscv终于在2019年12月诞生了。

tinyriscv是一个采用三级流水线设计，顺序、单发射、单核的32位RISC-V处理器，全部代码都是采用verilog HDL语言编写，核心设计思想是简单、易懂。

绪论

RISC-V是什么

RISC，即精简指令集处理器，是相对于X86这种CISC（复杂指令集处理器）来说的。RISC-V中的V是罗马数字，也即阿拉伯数字中的5，就是指第5代RISC。

RISC-V是一种指令集架构，和ARM、MIPS这些是属于同一类东西。RISC-V诞生于2010年，最大的特点是开源，任何人都可以设计RISC-V架构的处理器并且不会有任何版权问题。

既生ARM，何生RISC-V

ARM是一种很优秀的处理器，这一点是无可否认的，在RISC处理器中是处于绝对老大的地位。但是ARM是闭源的，要设计基于ARM的处理器是要交版权费的，或者说要购买ARM的授权，而且这授权费用是昂贵的。

RISC-V的诞生并不是偶然的，而是必然的，为什么？且由我从以下两大领域进行说明。

先看开源软件领域（或者说是操作系统领域），Windows是闭源的，Linux是开源的，Linux有多成功、对开源软件有多重要的意义，这个不用多说了吧。再看手机操作系统领域，iOS是闭源的，Android是开源的，Android有多成功，这个也不用多说了吧。对于RISC处理器领域，由于有了ARM的闭源，必然就会有另外一种开源的RISC处理器。RISC-V之于CPU的意义，就好比Linux之于开源软件的意义。

或者你会说现在也有好多开源的处理器架构啊，比如MIPS等等，为什么偏偏是RISC-V？这个在这里我就不细说了，我只想说一句：大部分人能看到的机遇不会是一个好的机遇，你懂的。

可以说未来十年乃至更长时间内不会有比RISC-V更优秀的开源处理器架构出现。错过RISC-V，你注定要错过一个时代。

浅谈Verilog

verilog，确切来说应该是verilog HDL(Hardware Description Language)，从它的名字就可以知道这是一种硬件描述语言。首先它是一种语言，和C语言、C++语言一样是一种编程语言，那么verilog描述的是什么硬件呢？描述电阻？描述电容？描述运算放大器？都不是，它描述的是数字电路里的硬件，比如与、非门、触发器、锁存器等等。

既然是编程语言，那一定会有它的语法，学过C语言的同学再来看verilog得代码，会发现有很多地方是相似的。

verilog的语法并不难，难的是什么时候该用wire类型，什么时候该用reg类型，什么时候该用assign来描述电路，什么时候该用always来描述电路。assign能描述组合逻辑电路，always也能描述组合逻辑电路，两者有什么区别呢？

用always描述组合逻辑电路

我们知道数字电路里有两大类型的电路，一种是组合逻辑电路，另外一种是时序逻辑电路。组合逻辑电路不需要时钟作为触发条件，因此输入会立即(不考虑延时)反映到输出。时序逻辑电路以时钟作为触发条件，时钟的上升沿到来时输入才会反映到输出。

在verilog中，assign能描述组合逻辑电路，always也能描述组合逻辑电路。对于简单的组合逻辑电路的话两者描述起来都比较好懂、容易理解，但是一旦到了复杂的组合逻辑电路，如果用assign描述的话要么是一大串要么是要用好多个assign，不容易弄明白。但是用always描述起来却是非常容易理解的。

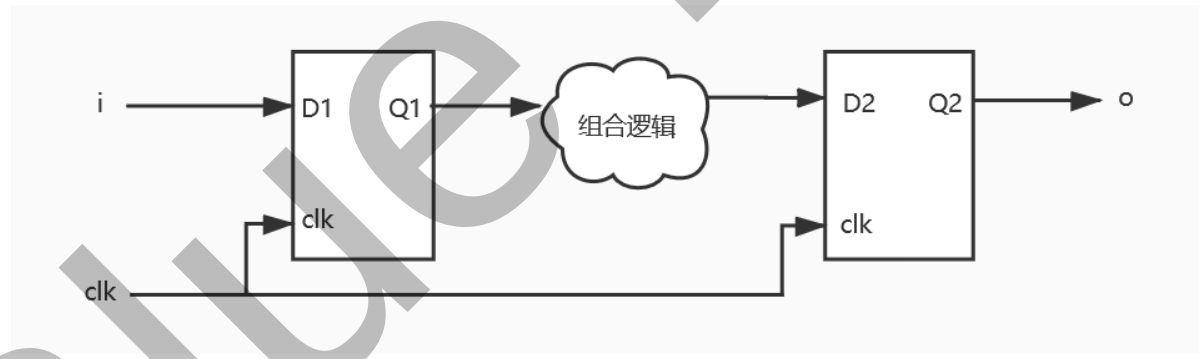
既然这样，那全部组合逻辑电路都用always来描述好了，呵呵，既然assign存在就有它的合理性。

用always描述组合逻辑电路时要注意避免产生锁存器，if和case的分支情况要写全。

在tinyriscv中用了大量的always来描述组合逻辑电路，特别是在译码和执行阶段。

数字电路设计中的时序问题

要分析数字电路中的时序问题，就一定要提到以下这个模型。



其中对时序影响最大的是上图中的组合逻辑电路。所以要避免时序问题，最简单的方法减小组合逻辑电路的延时。组合逻辑电路里的串联级数越多延时就越大，实在没办法减小串联级数时，可以采用流水线的方式将这些级数用触发器隔开。

流水线设计

要设计处理器的话，流水线是绕不开的。当然你也可以抬杠说:"用状态机也可以实现处理器啊，不一定要用流水线。"

采用流水线设计方式，不但可以提高处理器的工作频率，还可以提高处理器的效率。但是流水线并不是越长越好，流水线越长要使用的资源就越多、面积就越大。

在设计一款处理器之前，首先要确定好所设计的处理器要达到什么样的性能(或者说主频最高是多少)，所使用的资源的上限是多少，功耗范围是多少。如果一味地追求性能而不考虑资源和功耗的话，那么所设计出来的处理器估计就只能用来玩玩，或者做做学术研究。

tinyriscv采用的是三级流水线，即取指、译码和执行，设计的目标就是要对标ARM的Cortex-M3系列处理器。

代码风格

代码风格其实并没有一种标准，但是并不代表代码风格不重要。好的代码风格可以让别人看你的代码时有一种赏心悦目的感觉。哪怕代码只是写给自己看，也一定要养成好的代码风格的习惯。tinyriscv的代码风格在很大程度上沿用了写C语言代码所采用的风格。

下面介绍tinyriscv的一些主要的代码风格。

缩进

统一使用4个空格。

if语句

不管if语句下面有多少行语句，if下面的语句都由begin...end包起来，并且begin在if的最后，如下所示：

```
if (a == 1'b1) begin
    c <= b;
end else begin
    c <= d;
end
```

case语句

对于每一个分支情况，不管有多少行语句，都由begin...end包起来，如下所示：

```
case (a)
    c: begin
        e = g;
    end
    default: begin
        b = t;
    end
endcase
```

always语句

always语句后跟begin，如下所示：

```
always @ (posedge clk) begin
    a <= b;
end
```

其他

=、==、<=、>=、+、-、*、/、@等符号左右各有一个空格。

,和:符号后面有一个空格。

对于模块的输入信号，不省略wire关键字。

每个文件的最后留一行空行。

if、case、always后面都有一个空格。

硬件篇

硬件篇主要介绍tinyriscv的verilog代码设计。

tinyriscv整体框架如图2_1所示。

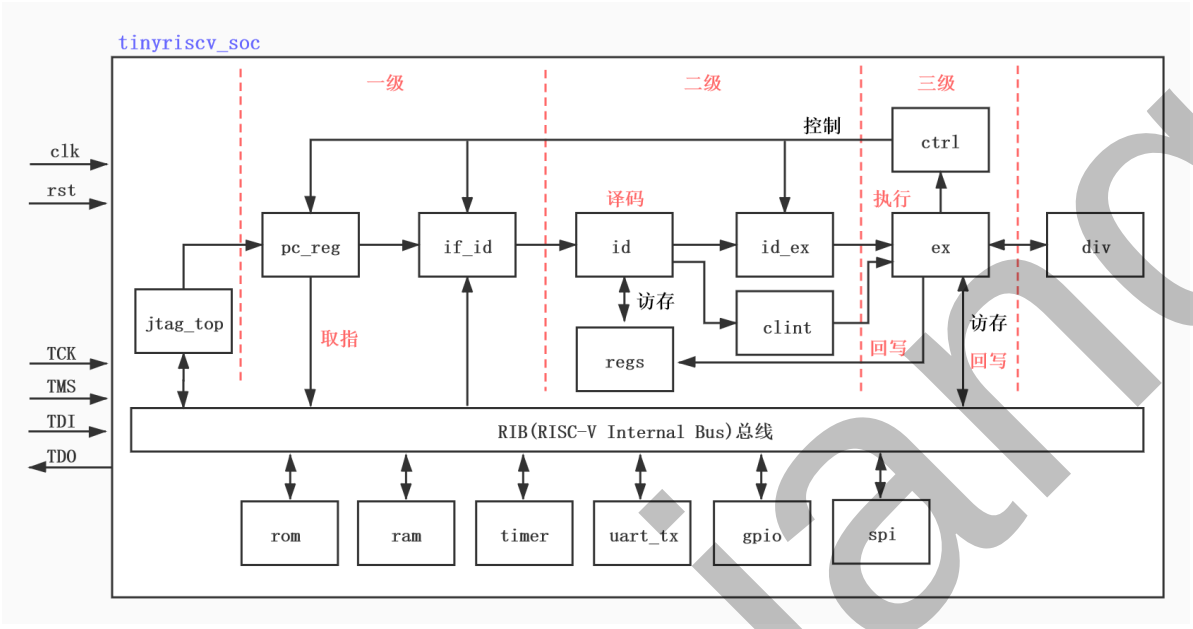


图2_1 tinyriscv整体框架

可见目前tinyriscv已经不仅仅是一个内核了，而是一个小型的SOC，包含一些简单的外设，如timer、uart_tx等。

tinyriscv SOC输入输出信号有两部分，一部分是系统时钟clk和复位信号rst，另一部分是JTAG调试信号，TCK、TMS、TDI和TDO。

上图中的小方框表示一个个模块，方框里面的文字表示模块的名字，箭头则表示模块与模块之间的的输入输出关系。

下面简单介绍每个模块的主要作用。

jtag_top：调试模块的顶层模块，主要有三大类型的信号，第一种是读写内存的信号，第二种是读写寄存器的信号，第三种是控制信号，比如复位MCU，暂停MCU等。

pc_reg：PC寄存器模块，用于产生PC寄存器的值，该值会被用作指令存储器的地址信号。

if_id：取指到译码之间的模块，用于将指令存储器输出的指令打一拍后送到译码模块。

id：译码模块，纯组合逻辑电路，根据if_id模块送进来的指令进行译码。当译码出具体的指令(比如add指令)后，产生是否写寄存器信号，读寄存器信号等。由于寄存器采用的是异步读方式，因此只要送出读寄存器信号后，会马上得到对应的寄存器数据，这个数据会和写寄存器信号一起送到id_ex模块。

id_ex：译码到执行之间的模块，用于将是否写寄存器的信号和寄存器数据打一拍后送到执行模块。

ex：执行模块，纯组合逻辑电路，根据具体的指令进行相应的操作，比如add指令就执行加法操作等。此外，如果是lw等访存指令的话，则会进行读内存操作，读内存也是采用异步读方式。最后将是否需要写寄存器、写寄存器地址，写寄存器数据信号送给regs模块，将是否需要写内存、写内存地址、写内存数据信号送给rib总线，由总线来分配访问的模块。

div：除法模块，采用试商法实现，因此至少需要32个时钟才能完成一次除法操作。

ctrl：控制模块，产生暂停流水线、跳转等控制信号。

clint：核心本地中断模块，对输入的中断请求信号进行总裁，产生最终的中断信号。

rom：程序存储器模块，用于存储程序(bin)文件。

ram：数据存储器模块，用于存储程序中的数据。

timer：定时器模块，用于计时和产生定时中断信号。目前支持RTOS时需要用到该定时器。

uart_tx：串口发送模块，主要用于调试打印。

gpio：简单的IO口模块，主要用于点灯调试。

spi：目前只有master角色，用于访问spi从机，比如spi norflash。

PC寄存器

PC寄存器模块所在的源文件：rtl/core/pc_reg.v

PC寄存器模块的输入输出信号如下表所示：

序号	信号名	输入/输出	位宽(bits)	说明
1	clk	输入	1	时钟输入信号
2	rst	输入	1	复位输入信号
3	jump_flag_i	输入	1	跳转标志
4	jump_addr_i	输入	32	跳转地址，即跳转到该地址
5	hold_flag_i	输入	3	暂停标志，即PC寄存器的值保持不变
6	jtag_reset_flag_i	输入	1	复位标志，即设置为复位后的值
7	pc_o	输出	32	PC寄存器值，即从该值处取指

PC寄存器模块代码比较简单，直接贴出来：

```
1         always @ (posedge clk) begin
2             // 复位
3             if (rst == `RstEnable || jtag_reset_flag_i == 1'b1) begin
4                 pc_o <= `CpuResetAddr;
5             // 跳转
6             end else if (jump_flag_i == `JumpEnable) begin
7                 pc_o <= jump_addr_i;
8             // 暂停
9             end else if (hold_flag_i >= `Hold_Pc) begin
10                pc_o <= pc_o;
11            // 地址加4
12            end else begin
13                pc_o <= pc_o + 4'h4;
14            end
15        end
```

第3行，PC寄存器的值恢复到原始值(复位后的值)有两种方式，第一种不用说了，就是复位信号有效。第二种是收到jtag模块发过来的复位信号。PC寄存器复位后的值为CpuResetAddr，即32'h0，可以通过改变CpuResetAddr的值来改变PC寄存器的复位值。

第6行，判断跳转标志是否有效，如果有效则直接将PC寄存器的值设置为jump_addr_i的值。因此可以知道，所谓的跳转就是改变PC寄存器的值，从而使CPU从该跳转地址开始取指。

第9行，判断暂停标志是否大于等于Hold_Pc，该值为3'b001。如果是，则保持PC寄存器的值不变。这里可能会有疑问，为什么Hold_Pc的值不是一个1bit的信号。因为这个暂停标志还会被if_id和id_ex模块使用，如果仅仅需要暂停PC寄存器的话，那么if_id模块和id_ex模块是不需要暂停的。当需要暂停if_id模块时，PC寄存器也会同时被暂停。当需要暂停id_ex模块时，那么整条流水线都会被暂停。

第13行，将PC寄存器的值加4。在这里可以知道，tinyriscv的取指地址是4字节对齐的，每条指令都是32位的。

通用寄存器

通用寄存器模块所在的源文件：rtl/core/regs.v

一共有32个通用寄存器x0~x31，其中寄存器x0是只读寄存器并且其值固定为0。

通用寄存器的输入输出信号如下表所示：

序号	信号名	输入/输出	位宽(bits)	说明
1	clk	输入	1	时钟输入
2	rst	输入	1	复位输入
3	we_i	输入	1	来自执行模块的写使能
4	waddr_i	输入	5	来自执行模块的写地址
5	wdata_i	输入	32	来自执行模块的写数据
6	jtag_we_i	输入	1	来自jtag模块的写使能
7	jtag_addr_i	输入	5	来自jtag模块的写地址
8	jtag_data_i	输入	32	来自jtag模块的写数据
9	raddr1_i	输入	5	来自译码模块的寄存器1读地址
10	rdata1_o	输出	32	寄存器1读数据
11	raddr2_i	输入	5	来自译码模块的寄存器2读地址
12	rdata2_o	输出	32	寄存器2读数据
13	jtag_data_o	输出	32	jtag读数据

注意，这里的寄存器1不是指x1寄存器，寄存器2也不是指x2寄存器。而是指一条指令里涉及到的两个寄存器(源寄存器1和源寄存器2)。一条指令可能会同时读取两个寄存器的值，所以有两个读端口。又因为jtag模块也会进行寄存器的读操作，所以一共有三个读端口。

读寄存器操作来自译码模块，并且读出来的寄存器数据也会返回给译码模块。写寄存器操作来自执行模块。

先看读操作的代码，如下：

```
1    // 读寄存器1
2    always @ (*) begin
3        if (rst == `RstEnable) begin
```

```
4               rdata1_o = `ZeroWord;
5           end else if (raddr1_i == `RegNumLog2'h0) begin
6               rdata1_o = `ZeroWord;
7           // 如果读地址等于写地址，并且正在写操作，则直接返回写数据
8           end else if (raddr1_i == waddr_i && we_i == `WriteEnable) begin
9               rdata1_o = wdata_i;
10          end else begin
11              rdata1_o = regs[raddr1_i];
12          end
13      end
14
15      // 读寄存器2
16      always @ (*) begin
17          if (rst == `RstEnable) begin
18              rdata2_o = `ZeroWord;
19          end else if (raddr2_i == `RegNumLog2'h0) begin
20              rdata2_o = `ZeroWord;
21          // 如果读地址等于写地址，并且正在写操作，则直接返回写数据
22          end else if (raddr2_i == waddr_i && we_i == `WriteEnable) begin
23              rdata2_o = wdata_i;
24          end else begin
25              rdata2_o = regs[raddr2_i];
26          end
27      end
```

可以看到两个寄存器的读操作几乎是一样的。因此在这里只解析读寄存器1那部分代码。

第5行，如果是读寄存器0(x0)，那么直接返回0就可以了。

第8行，这涉及到数据相关问题。由于流水线的原因，当前指令处于执行阶段的时候，下一条指令则处于译码阶段。由于执行阶段不会写寄存器，而是在下一个时钟到来时才会进行寄存器写操作，如果译码阶段的指令需要上一条指令的结果，那么此时读到的寄存器的值是错误的。比如下面这两条指令：

```
1  add x1, x2, x3
2  add x4, x1, x5
```

第二条指令依赖于第一条指令的结果。为了解决这个数据相关的问题就有了第8~9行的操作，即如果读寄存器等于写寄存器，则直接将要写的值返回给读操作。

第11行，如果没有数据相关，则返回要读的寄存器的值。

下面看写寄存器操作，代码如下：

```
1       // 写寄存器
2       always @ (posedge clk) begin
3           if (rst == `RstDisable) begin
4               // 优先ex模块写操作
5               if ((we_i == `WriteEnable) && (waddr_i != `RegNumLog2'h0)) begin
6                   regs[waddr_i] <= wdata_i;
7               end else if ((jtag_we_i == `WriteEnable) && (jtag_addr_i !=
    `RegNumLog2'h0)) begin
8                   regs[jtag_addr_i] <= jtag_data_i;
9               end
10          end
11      end
```

第5~6行，如果执行模块写使能并且要写的寄存器不是x0寄存器，则将要写的值写到对应的寄存器。

第7~8行，jtag模块的写操作。

CSR寄存器模块(csr_reg.v)和通用寄存器模块的读、写操作是类似的，这里就不重复了。

取指

目前tinyriscv所有外设(包括rom和ram)、寄存器的读取都是与时钟无关的，或者说所有外设、寄存器的读取采用的是组合逻辑的方式。这一点非常重要!

tinyriscv并没有具体的取指模块和代码。PC寄存器模块的输出pc_o会连接到外设rom模块的地址输入，又由于rom的读取是组合逻辑，因此每一个时钟上升沿到来之前(时序是满足要求的)，从rom输出的指令已经稳定在if_id模块的输入，当时钟上升沿到来时指令就会输出到id模块。

取到的指令和指令地址会输入到if_id模块(if_id.v)，if_id模块是一个时序电路，作用是将输入的信号打一拍后再输出到译码(id.v)模块。

译码

译码模块所在的源文件：rtl/core/id.v

译码(id)模块是一个纯组合逻辑电路，主要作用有以下几点：

1.根据指令内容，解析出当前具体是哪一条指令(比如add指令)。

2.根据具体的指令，确定当前指令涉及的寄存器。比如读寄存器是一个还是两个，是否需要写寄存器以及写哪一个寄存器。

3.访问通用寄存器，得到要读的寄存器的值。

译码模块的输入输出信号如下表所示：

序号	信号名	输入/输出	位宽(bits)	说明
1	rst	输入	1	复位信号
2	inst_i	输入	32	指令内容
3	inst_addr_i	输入	32	指令地址
4	reg1_rdata_i	输入	32	寄存器1输入数据
5	reg2_rdata_i	输入	32	寄存器2输入数据
6	csr_rdata_i	输入	32	CSR寄存器输入数据
7	ex_jump_flag_i	输入	1	跳转信号
8	reg1_raddr_o	输出	5	读寄存器1地址，即读哪一个通用寄存器
9	reg2_raddr_o	输出	5	读寄存器2地址，即读哪一个通用寄存器
10	csr_raddr_o	输出	32	读csr寄存器地址，即读哪一个CSR寄存器
11	mem_req_o	输出	1	向总线请求访问内存信号
12	inst_o	输出	32	指令内容
13	inst_addr_o	输出	32	指令地址
14	reg1_rdata_o	输出	32	通用寄存器1数据
15	reg2_rdata_o	输出	32	通用寄存器2数据
16	reg_we_o	输出	1	通用寄存器写使能
17	reg_waddr_o	输出	5	通用寄存器写地址，即写哪一个通用寄存器
18	csr_we_o	输出	1	CSR寄存器写使能
19	csr_rdata_o	输出	32	CSR寄存器读数据
20	csr_waddr_o	输出	32	CSR寄存器写地址，即写哪一个CSR寄存器

以add指令为例来说明如何译码。下图是add指令的编码格式：

1	2	3	4	5	6	
0000000	rs2	rs1	000	rd	0110011	ADD

可知，add指令被编码成6部分内容。通过第1、4、6这三部分可以唯一确定当前指令是否是add指令。知道是add指令之后，就可以知道add指令需要读两个通用寄存器(rs1和rs2)和写一个通用寄存器(rd)。下面看具体的代码：

```
1  case (opcode)
2  ...
3      `INST_TYPE_R_M: begin
4          if ((funct7 == 7'b0000000) || (funct7 == 7'b0100000)) begin
5              case (funct3)
6                  `INST_ADD_SUB, `INST_SLL, `INST_SLT, `INST_SLTU, `INST_XOR,
   `INST_SR, `INST_OR, `INST_AND: begin
7                      reg_we_o = `WriteEnable;
8                      reg_waddr_o = rd;
9                      reg1_raddr_o = rs1;
10                     reg2_raddr_o = rs2;
11                 end
12 ...
```

第1行，opcode就是指令编码中的第6部分内容。

第3行，`INST_TYPE_R_M的值为7'b0110011。

第4行，funct7是指指令编码中的第1部分内容。

第5行，funct3是指指令编码中的第4部分内容。

第6行，到了这里，第1、4、6这三部分已经译码完毕，已经可以确定当前指令是add指令了。

第7行，设置写寄存器标志为1，表示执行模块结束后的下一个时钟需要写寄存器。

第8行，设置写寄存器地址为rd，rd的值为指令编码里的第5部分内容。

第9行，设置读寄存器1的地址为rs1，rs1的值为指令编码里的第3部分内容。

第10行，设置读寄存器2的地址为rs2，rs2的值为指令编码里的第2部分内容。

其他指令的译码过程是类似的，这里就不重复了。译码模块看起来代码很多，但是大部分代码都是类似的。

译码模块还有个作用是当指令为加载内存指令(比如lw等)时，向总线发出请求访问内存的信号。这部分内容将在总线一节再分析。

译码模块的输出会送到id_ex模块(id_ex.v)的输入，id_ex模块是一个时序电路，作用是将输入的信号打一拍后再输出到执行模块(ex.v)。

执行

执行模块所在的源文件：rtl/core/ex.v

执行(ex)模块是一个纯组合逻辑电路，主要作用有以下几点：

1.根据当前是什么指令执行对应的操作，比如add指令，则将寄存器1的值和寄存器2的值相加。

2.如果是内存加载指令，则读取对应地址的内存数据。

3.如果是跳转指令，则发出跳转信号。

执行模块的输入输出信号如下表所示：

序号	信号名	输入/输出	位宽(bits)	说明
1	rst	输入	1	复位信号
2	inst_i	输入	32	指令内容
3	inst_addr_i	输入	32	指令地址
4	reg_we_i	输入	1	寄存器写使能
5	reg_waddr_i	输入	5	通用寄存器写地址，即写哪一个通用寄存器
6	reg1_rdata_i	输入	32	通用寄存器1读数据
7	reg2_rdata_i	输入	32	通用寄存器2读数据
8	csr_we_i	输入	1	CSR寄存器写使能
9	csr_waddr_i	输入	32	CSR寄存器写地址，即写哪一个CSR寄存器
10	csr_rdata_i	输入	32	CSR寄存器读数据
11	int_assert_i	输入	1	中断信号
12	int_addr_i	输入	32	中断跳转地址，即中断发生后跳转到哪个地址
13	mem_rdata_i	输入	32	内存读数据
14	div_ready_i	输入	1	除法模块是否准备好信号，即是否可以进行除法运算
15	div_result_i	输入	64	除法结果
16	div_busy_i	输入	1	除法模块忙信号，即正在进行除法运算
17	div_op_i	输入	3	具体的除法运算，即DIV、DIVU、REM和REMU中的哪一种
18	div_reg_waddr_i	输入	5	除法运算完成后要写的通用寄存器地址
19	mem_wdata_o	输出	32	内存写数据
20	mem_raddr_o	输出	32	内存读地址
21	mem_waddr_o	输出	32	内存写地址
22	mem_we_o	输出	1	内存写使能
23	mem_req_o	输出	1	请求访问内存信号
24	reg_wdata_o	输出	32	通用寄存器写数据
25	reg_we_o	输出	1	通用寄存器写使能
26	reg_waddr_o	输出	5	通用寄存器写地址
27	csr_wdata_o	输出	32	CSR寄存器写数据
28	csr_we_o	输出	1	CSR寄存器写使能

序号	信号名	输入/输出	位宽(bits)	说明
29	csr_waddr_o	输出	32	CSR寄存器写地址，即写哪一个CSR寄存器
30	div_start_o	输出	1	开始除法运算
31	div_dividend_o	输出	32	除法运算中的被除数
32	div_divisor_o	输出	32	除法运算中的除数
33	div_op_o	输出	3	具体的除法运算，即DIV、DIVU、REM和REMU中的哪一种
34	div_reg_waddr_o	输出	5	除法运算完成后要写的通用寄存器地址
35	hold_flag_o	输出	1	暂停流水线信号
36	jump_flag_o	输出	1	跳转信号
37	jump_addr_o	输出	32	跳转地址

下面以add指令为例说明，add指令的作用就是将寄存器1的值和寄存器2的值相加，最后将结果写入目的寄存器。代码如下：

```
1  ...
2  `INST_TYPE_R_M: begin
3      if ((funct7 == 7'b0000000) || (funct7 == 7'b0100000)) begin
4          case (funct3)
5              `INST_ADD_SUB: begin
6                  jump_flag = `JumpDisable;
7                  hold_flag = `HoldDisable;
8                  jump_addr = `ZeroWord;
9                  mem_wdata_o = `ZeroWord;
10                 mem_raddr_o = `ZeroWord;
11                 mem_waddr_o = `ZeroWord;
12                 mem_we = `WriteDisable;
13                 if (inst_i[30] == 1'b0) begin
14                     reg_wdata = reg1_rdata_i + reg2_rdata_i;
15                 end else begin
16                     reg_wdata = reg1_rdata_i - reg2_rdata_i;
17                 end
18         ...
19     end
20 ...
```

第2~4行，译码操作。

第5行，对add或sub指令进行处理。

第6~12行，当前指令不涉及到的操作(比如跳转、写内存等)需要将其置回默认值。

第13行，指令编码中的第30位区分是add指令还是sub指令。0表示add指令，1表示sub指令。

第14行，执行加法操作。

第16行，执行减法操作。

其他指令的执行是类似的，需要注意的是没有涉及的信号要将其置为默认值，if和case情况要写全，避免产生锁存器。

下面以beq指令说明跳转指令的执行。beq指令的编码如下：

imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ

beq指令的作用就是当寄存器1的值和寄存器2的值相等时发生跳转，跳转的目的地址为当前指令的地址加上符号扩展的imm的值。具体代码如下：

```
1  ...
2  `INST_TYPE_B: begin
3      case (funct3)
4          `INST_BEQ: begin
5              hold_flag = `HoldDisable;
6              mem_wdata_o = `ZeroWord;
7              mem_raddr_o = `ZeroWord;
8              mem_waddr_o = `ZeroWord;
9              mem_we = `WriteDisable;
10             reg_wdata = `ZeroWord;
11             if (reg1_rdata_i == reg2_rdata_i) begin
12                 jump_flag = `JumpEnable;
13                 jump_addr = inst_addr_i + {{20{inst_i[31]}}, inst_i[7],
   inst_i[30:25], inst_i[11:8], 1'b0};
14             end else begin
15                 jump_flag = `JumpDisable;
16                 jump_addr = `ZeroWord;
17             end
18     ...
19 end
20 ...
```

第2~4行，译码出beq指令。

第5~10行，没有涉及的信号置为默认值。

第11行，判断寄存器1的值是否等于寄存器2的值。

第12行，跳转使能，即发生跳转。

第13行，计算出跳转的目的地址。

第15、16行，不发生跳转。

其他跳转指令的执行是类似的，这里就不再重复了。

访存

由于tinyriscv只有三级流水线，因此没有访存这个阶段，访存的操作放在了执行模块中。具体是这样的，在译码阶段如果识别出是内存访问指令(lb、lh、lw、lbu、lhu、sb、sh、sw)，则向总线发出内存访问请求，具体代码(位于id.v)如下：

```
1  ...
2  `INST_TYPE_L: begin
3      case (funct3)
4          `INST_LB, `INST_LH, `INST_LW, `INST_LBU, `INST_LHU: begin
5              reg1_raddr_o = rs1;
6              reg2_raddr_o = `ZeroReg;
7              reg_we_o = `WriteEnable;
```

```
8             reg_waddr_o = rd;
9             mem_req = `RIB_REQ;
10         end
11         default: begin
12            reg1_raddr_o = `ZeroReg;
13            reg2_raddr_o = `ZeroReg;
14            reg_we_o = `WriteDisable;
15            reg_waddr_o = `ZeroReg;
16         end
17    endcase
18end
19`INST_TYPE_S: begin
20    case (funct3)
21        `INST_SB, `INST_SW, `INST_SH: begin
22            reg1_raddr_o = rs1;
23            reg2_raddr_o = rs2;
24            reg_we_o = `WriteDisable;
25            reg_waddr_o = `ZeroReg;
26            mem_req = `RIB_REQ;
27    end
28...
```

第2~4行，译码出内存加载指令，lb、lh、lw、lbu、lhu。

第5行，需要读寄存器1。

第6行，不需要读寄存器2。

第7行，写目的寄存器使能。

第8行，写目的寄存器的地址，即写哪一个通用寄存器。

第9行，发出访问内存请求。

第19~21行，译码出内存存储指令，sb、sw、sh。

第22行，需要读寄存器1。

第23行，需要读寄存器2。

第24行，不需要写目的寄存器。

第26行，发出访问内存请求。

问题来了，为什么在取指阶段发出内存访问请求？这跟总线的设计是相关的，这里先不具体介绍总线的设计，只需要知道如果需要访问内存，则需要提前一个时钟向总线发出请求。

在译码阶段向总线发出内存访问请求后，在执行阶段就会得到对应的内存数据。

下面看执行阶段的内存加载操作，以lb指令为例，lb指令的作用是访问内存中的某一个字节，代码(位于ex.v)如下：

```
1 ...
2 `INST_TYPE_L: begin
3     case (funct3)
4         `INST_LB: begin
5             jump_flag = `JumpDisable;
6             hold_flag = `HoldDisable;
7             jump_addr = `ZeroWord;
8             mem_wdata_o = `ZeroWord;
9             mem_waddr_o = `ZeroWord;
```

```
            mem_we = `WriteDisable;
            mem_raddr_o = reg1_rdata_i + {{20{inst_i[31]}}, inst_i[31:20]};
            case (mem_raddr_index)
                2'b00: begin
                    reg_wdata = {{24{mem_rdata_i[7]}}, mem_rdata_i[7:0]};
                end
                2'b01: begin
                    reg_wdata = {{24{mem_rdata_i[15]}}, mem_rdata_i[15:8]};
                end
                2'b10: begin
                    reg_wdata = {{24{mem_rdata_i[23]}}, mem_rdata_i[23:16]};
                end
                default: begin
                    reg_wdata = {{24{mem_rdata_i[31]}}, mem_rdata_i[31:24]};
                end
            endcase
        end
...
```

第2~4行，译码出lb指令。

第5~10行，将没有涉及的信号置为默认值。

第11行，得到访存的地址。

第12行，由于访问内存的地址必须是4字节对齐的，因此这里的mem_raddr_index的含义就是32位内存数据(4个字节)中的哪一个字节，2'b00表示第0个字节，即最低字节，2'b01表示第1个字节，2'b10表示第2个字节，2'b11表示第3个字节，即最高字节。

第14、17、20、23行，写寄存器数据。

回写

由于tinyriscv只有三级流水线，因此也没有回写(write back，或者说写回)这个阶段，在执行阶段结束后的下一个时钟上升沿就会把数据写回寄存器或者内存。

需要注意的是，在执行阶段，判断如果是内存存储指令(sb、sh、sw)，则向总线发出访问内存请求。而对于内存加载(lb、lh、lw、lbu、lhu)指令是不需要的。因为内存存储指令既需要加载内存数据又需要往内存存储数据。

以sb指令为例，代码(位于ex.v)如下：

```
...
`INST_TYPE_S: begin
    case (funct3)
        `INST_SB: begin
            jump_flag = `JumpDisable;
            hold_flag = `HoldDisable;
            jump_addr = `ZeroWord;
            reg_wdata = `ZeroWord;
            mem_we = `WriteEnable;
            mem_req = `RIB_REQ;
            mem_waddr_o = reg1_rdata_i + {{20{inst_i[31]}}, inst_i[31:25],
inst_i[11:7]};
            mem_raddr_o = reg1_rdata_i + {{20{inst_i[31]}}, inst_i[31:25],
inst_i[11:7]};
            case (mem_waddr_index)
                2'b00: begin
```

```
15                      mem_wdata_o = {mem_rdata_i[31:8], reg2_rdata_i[7:0]};
16                  end
17                  2'b01: begin
18                     mem_wdata_o = {mem_rdata_i[31:16], reg2_rdata_i[7:0],
    mem_rdata_i[7:0]};
19                  end
20                  2'b10: begin
21                      mem_wdata_o = {mem_rdata_i[31:24], reg2_rdata_i[7:0],
    mem_rdata_i[15:0]};
22                  end
23                  default: begin
24                      mem_wdata_o = {reg2_rdata_i[7:0], mem_rdata_i[23:0]};
25                  end
26              endcase
27          end
28  ...
```

第2~4行，译码出sb指令。

第5~8行，将没有涉及的信号置为默认值。

第9行，写内存使能。

第10行，发出访问内存请求。

第11行，内存写地址。

第12行，内存读地址，读地址和写地址是一样的。

第13行，mem_waddr_index的含义就是写32位内存数据中的哪一个字节。

第15、18、21、24行，写内存数据。

sb指令只改变读出来的32位内存数据中对应的字节，其他3个字节的数据保持不变，然后写回到内存中。

跳转和流水线暂停

总线

中断

JTAG

RTL仿真验证

软件篇

RISC-V汇编语言

Makefile与链接脚本

启动代码

异常和中断处理