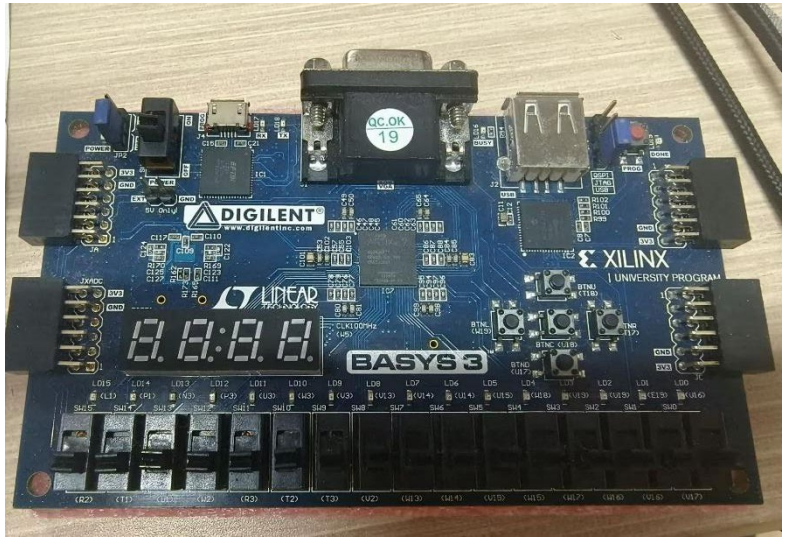
RISC-V CPU Simulator

作业内容：用C++模拟一个采用Tomasulo架构的RISC-V CPU

温馨提示：

在下学期的《计算机系统》课程大作业中，大家会用Verilog写一个Tomasulo架构的CPU，并在 FPGA板上测试（上板）。

如果C++版本写得足够好，可以大大减少编写、调试难度；反之其折磨程度可能会达到编译器的60%。



硬件与 Verilog

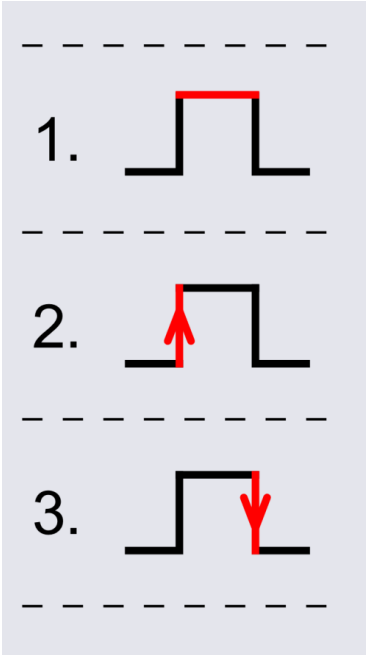
时钟周期

计算机内部会以一定频率发出时钟信号，其决定了各个部件的工作节奏。

时钟周期即为一次时钟信号的完整周期。

在设计一个时序逻辑的部件时，通常会规定它在一个时钟周期的各个阶段都干些什么。

```
always @(posedge clk or negedge rstn) begin
    if (!rstn) begin
        st_cur      <= 'b0 ;
    end
    else begin
        st_cur      <= st_next ;
    end
end
```



一个简单的状态机。在时钟信号上升沿到来时对st_cur进行赋值

clk是时钟信号。（rstn是“不复位”信号，和clk没什么关系。当其从true变为false，也即检测到其下降沿时，执行复位操作）

组合逻辑

直接将逻辑电路组合到一起。

- 输出信号只由输入信号决定
- 一般不能存储状态
- 立即响应，当输入信号改变时，输出立即开始变化
- 一般用于运算器等
- 当电路极其复杂时，会带来很大时延
- 存在竞争冒险

竞争冒险：当一个逻辑门的输入信号通过多条路径到达时，如果这些路径的传播延迟不同，可能会导致信号到达的顺序不一致。

时序逻辑

每个时钟周期，根据输入和自己当前状态，进行输出和状态改变

- 输出信号取决于输入信号和当前状态
- 一般包含存储元件，可以存储状态
- 由时钟信号控制响应（因此各元件可以同步执行，避免产生竞争冒险）。
- 一般用于状态机、存储器等。

wire

wire 类型表示硬件单元之间的物理连线，由其连接的器件输出端连续驱动

1. wire 可简单地理解为电线，连接的过程中可以夹杂逻辑门（对数据进行运算）。
2. wire 只能用 assign 赋值，并且只能被赋值一次（因为电线不可能随意变动）。
3. wire 中数值的变化是实时的

阻塞赋值与非阻塞赋值

阻塞赋值：阻塞赋值属于顺序执行，即下一条语句执行前，当前语句一定会执行完毕。阻塞赋值语句使用等号 = 作为赋值符。

非阻塞赋值：非阻塞赋值属于并行执行语句，即下一条语句的执行和当前语句的执行是同时进行的，它不会阻塞位于同一个语句块中后面语句的执行。非阻塞赋值语句使用小于等于号 <= 作为赋值符。

reg

寄存器用来表示存储单元，它会保持数据原有的值，直到被改写。

在 @always(*) 中只使用阻塞赋值 = 给 reg 赋值，这等价于 wire 用 assign 赋值。

在 @always(posedge clk) 中只使用非阻塞赋值 <= 给 reg 赋值，所有赋值操作是并行的，等价于只会使用上一周期的数据进行赋值。

注：
@always(*)：任何一个时钟信号发生变化后，该代码块均会执行（不建议使用，可以直接用wire）

@always(posedge clk)：当时钟上升沿信号发出后，代码块执行

你也可以在 @always(...)添加任何wire/reg变量，使得这些变量发生变化后代码块执行

以下两种方式的不同？

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

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

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

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

避免了数据竞争！
因为a,b在赋值的时候**使用的是上一个周期的值**

向量

当位宽大于 1 时，wire 或 reg 即可声明为向量的形式。例如：

```
reg  [3:0]      counter ;     //声明4bit位宽的寄存器counter
wire [32-1:0]   gpio_data;    //声明32bit位宽的线型变量gpio_data
```

参考

更为全面的Verilog语法与使用请参考：

https://www.runoob.com/w3cnote/verilog-intro.html

https://hdlbits.01xz.net/wiki/Main_Page

指令集架构

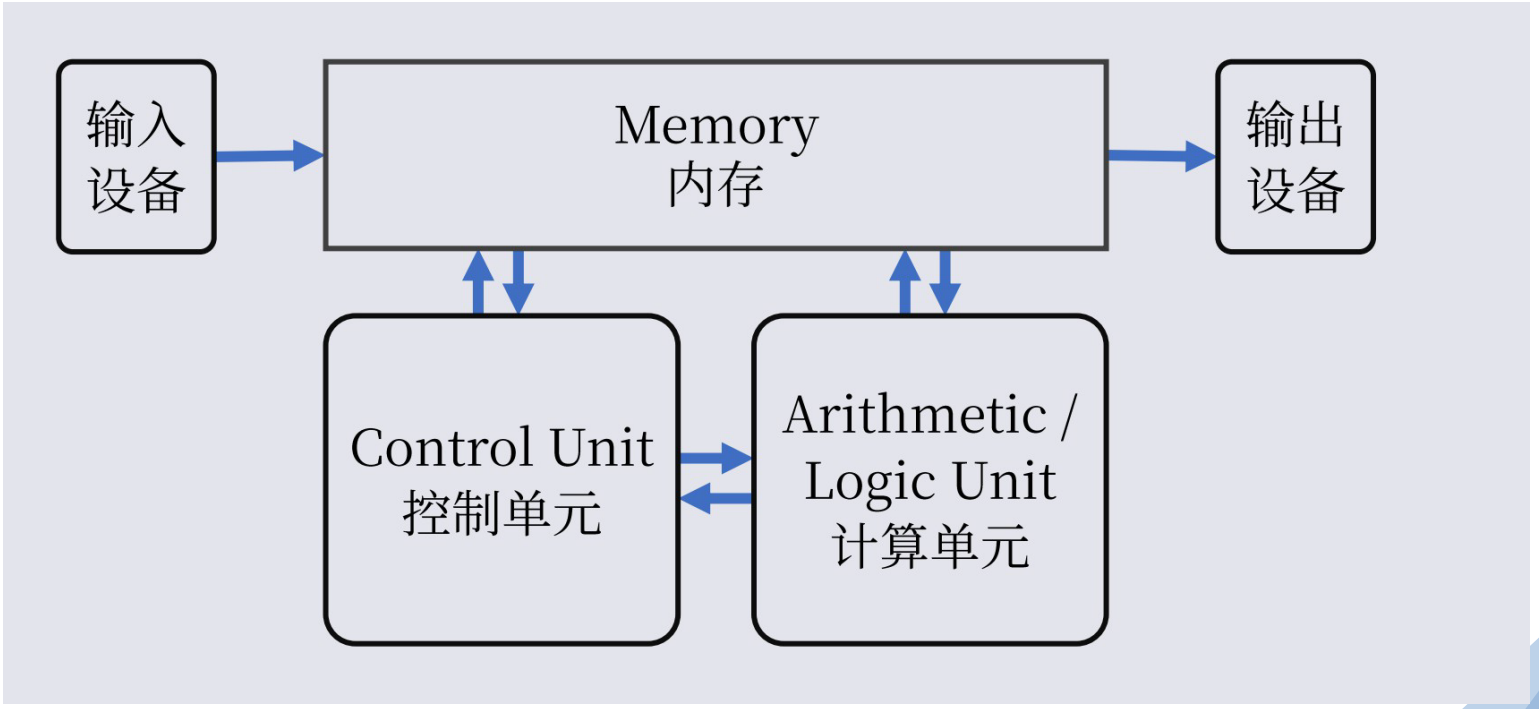
Instruction Set Architecture

什么是指令集架构?

冯诺依曼架构

指令和数据都存储在内存中

CPU：计算单元+控制单元



RISC-V RV32I

RISC-V是一个基于精简指令集（RISC）原则的开源指令集架构。

本次作业中涉及到的指令是RV32I（RISC-V 32-bit基础整型指令集）的一部分。其支持32位寻址空间，支持字节地址访问。其寄存器也都是32位。

- 1个指令地址寄存器PC。
- 32个通用寄存器。

寄存器	ABI 名称	用途	保存者
x0	zero	始终为 0	-
x1	ra	返回地址 (return address)	Caller
x2	sp	栈指针 (Stack pointer)	Callee
x3	gp	全局指针 (Global pointer)	-
x4	tp	线程指针 (Thread pointer)	-
x5-7	t0-2	临时量	Caller
x8	s0/fp	调用者保存数据 (Saved register)/帧指针 (frame pointer)	Callee
x9	s1	调用者保存数据 (Saved register)	Callee
x10-11	a0-1	函数参数/返回值	Caller
x12-17	a2-7	函数参数	Caller
x18-27	s2-11	调用者保存数据 (Saved register)	Callee
x28-31	t3-6	临时量	Caller

指令地址寄存器PC（Program Counter）

计算机要执行的指令在内存中。为了获取指令，需要知道指令在内存中的位置，通过一次访存操作读取到下一条指令。

PC就是一个指针，指向下一条指令的地址

RV32I每条指令长度都是4个byte，因此在读取完一条非跳转指令后，PC应当+4，以指向下一条指令。

当读到跳转类指令时，根据跳转类指令的执行结果修改PC的值。

RISC-V RV32I

指令大概分为如下几类：

- 跳转指令：主要作用是修改PC的值。条件跳转指令会根据运算结果是否为0，将PC设置为立即数或让PC+=4
- 运算指令：用两个寄存器中的数，或者一个寄存器中的数和一个立即数进行一些运算，包括加减、比较、位运算。计算结果存在目标寄存器中。
- 访存指令：将寄存器中的值存到某个内存地址，或将内存地址中的数据读到某个寄存器中。在现实场景，访存操作通常极耗时间，通常要花费几十上百个周期。

RISC-V RV32I

本次作业涉及到的指令为手册第27页表中fence之前（不包含fence）的所有指令，共37条

具体指令请自行阅读手册。

之后会在群里发布Reference Card.

在我们下发的dump文件中会看到一些伪指令，
实际可以认为是一般指令的简写，
例如 j 105c 即为 jal x0 105c

31 25	24 20	19 15	14 12	11 7	6 0	
imm[31:12]				rd	0110111	U lui
imm[31:12]				rd	0010111	U auipc
imm[20|10:1|11|19:12]				rd	1101111	J jal
imm[11:0]		rs1	000	rd	1100111	I jalr
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	B beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	B bne
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	B blt
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	B bge
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	B bltu
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	B bgeu
imm[11:0]		rs1	000	rd	0000011	I lb
imm[11:0]		rs1	001	rd	0000011	I lh
imm[11:0]		rs1	010	rd	0000011	I lw
imm[11:0]		rs1	100	rd	0000011	I lbu
imm[11:0]		rs1	101	rd	0000011	I lhu
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	S sb
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	S sh
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	S sw
imm[11:0]		rs1	000	rd	0010011	I addi
imm[11:0]		rs1	010	rd	0010011	I slti
imm[11:0]		rs1	011	rd	0010011	I sltiu
imm[11:0]		rs1	100	rd	0010011	I xori
imm[11:0]		rs1	110	rd	0010011	I ori
imm[11:0]		rs1	111	rd	0010011	I andi
0000000	shamt	rs1	001	rd	0010011	I slli
0000000	shamt	rs1	101	rd	0010011	I srli
0100000	shamt	rs1	101	rd	0010011	I srai
0000000	rs2	rs1	000	rd	0110011	R add
0100000	rs2	rs1	000	rd	0110011	R sub
0000000	rs2	rs1	001	rd	0110011	R sll
0000000	rs2	rs1	010	rd	0110011	R slt
0000000	rs2	rs1	011	rd	0110011	Rsltu
0000000	rs2	rs1	100	rd	0110011	R xor
0000000	rs2	rs1	101	rd	0110011	R srl
0100000	rs2	rs1	101	rd	0110011	R sra
0000000	rs2	rs1	110	rd	0110011	R or
0000000	rs2	rs1	111	rd	0110011	R and

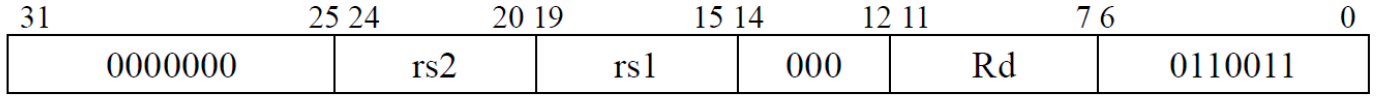
例子

add rd, rs1, rs2 　　x[rd] = x[rs1] + x[rs2]

加 *(Add)*. R-type, RV32I and RV64I.

把寄存器 x[*rs2*]加到寄存器 x[*rs1*]上，结果写入 x[*rd*]。忽略算术溢出。

压缩形式：**c.add** rd, rs2; **c.mv** rd, rs2



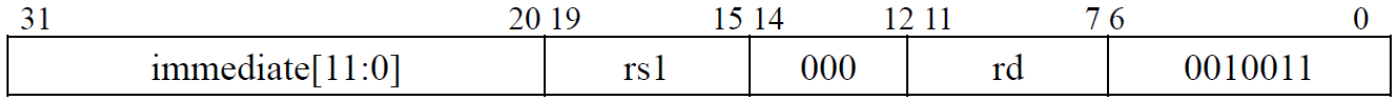
---

addi rd, rs1, immediate 　　x[rd] = x[rs1] + sext(immediate)

加立即数*(Add Immediate)*. I-type, RV32I and RV64I.

把符号位扩展的立即数加到寄存器 x[*rs1*]上，结果写入 x[*rd*]。忽略算术溢出。

压缩形式：**c.li** rd, imm; **c.addi** rd, imm; **c.addi16sp** imm; **c.addi4spn** rd, imm



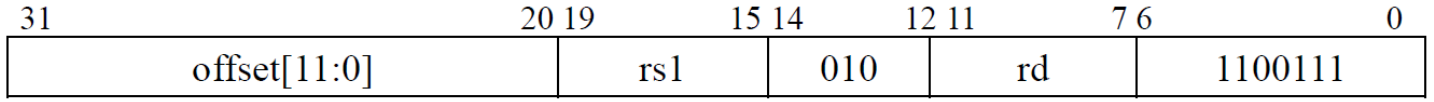
例子

jalr rd, offset(rs1)　　t =pc+4; pc=(x[rs1]+sext(offset))&~1; x[rd]=t

跳转并寄存器链接 *(Jump and Link Register)*. I-type, RV32I and RV64I.

把 *pc* 设置为 x[*rs1*]+*sign-extend*(*offset*)，把计算出的地址的最低有效位设为 0，并将原 *pc+4* 的值写入 f[*rd*]。*rd* 默认为 x1。

压缩形式：**c.jr** rs1；**c.jalr** rs1



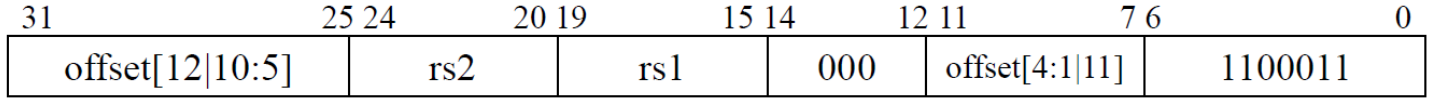
---

beq rs1, rs2, offset　　if (rs1 == rs2) pc += sext(offset)

相等时分支 *(Branch if Equal)*. B-type, RV32I and RV64I.

若寄存器 x[*rs1*]和寄存器 x[*rs2*]的值相等，把 *pc* 的值设为当前值加上符号位扩展的偏移 *offset*。

压缩形式：**c.beqz** rs1, offset



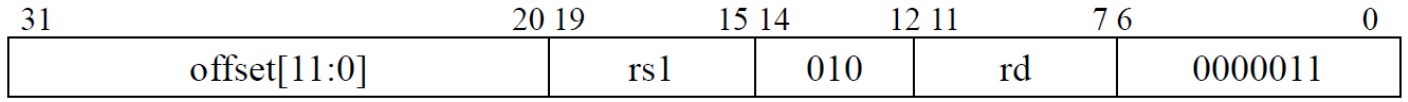
例子

lw rd, offset(rs1)　　x[rd] = sext(M[x[rs1] + sext(offset)][31:0])

字加载 *(Load Word)*. I-type, RV32I and RV64I.

从地址 x[*rs1*] + *sign-extend*(*offset*)读取四个字节，写入 x[*rd*]。对于 RV64I，结果要进行符号位扩展。

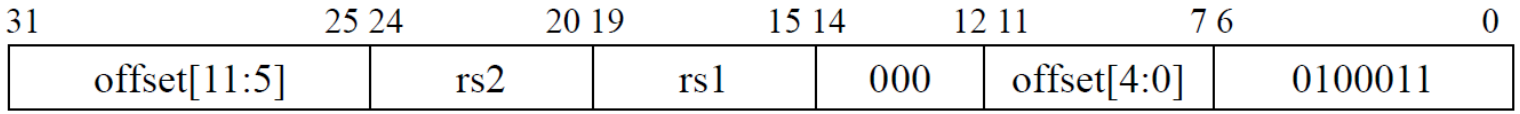
压缩形式：**c.lwsp** rd, offset; **c.lw** rd, offset(rs1)



sb rs2, offset(rs1)　　M[x[rs1] + sext(offset) = x[rs2][7:0]

存字节*(Store Byte)*. S-type, RV32I and RV64I.

将 x[*rs2*]的低位字节存入内存地址 x[*rs1*]+*sign-extend(offset)*。

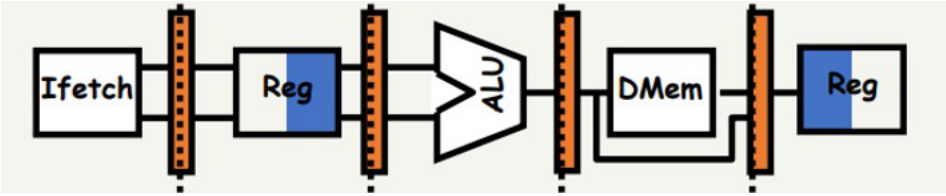


更高效的CPU指令执行

效率低的原因

上述几类指令的执行过程可分为几个阶段：读取指令（IF, Instruction Fetch），解析指令（ID, Instruction Decode，此步骤包括读取指令中涉及到的寄存器的值），运算（EX），访存（MEM），写回寄存器（WB, Write Back)

每个阶段都由CPU中不同的元件执行。

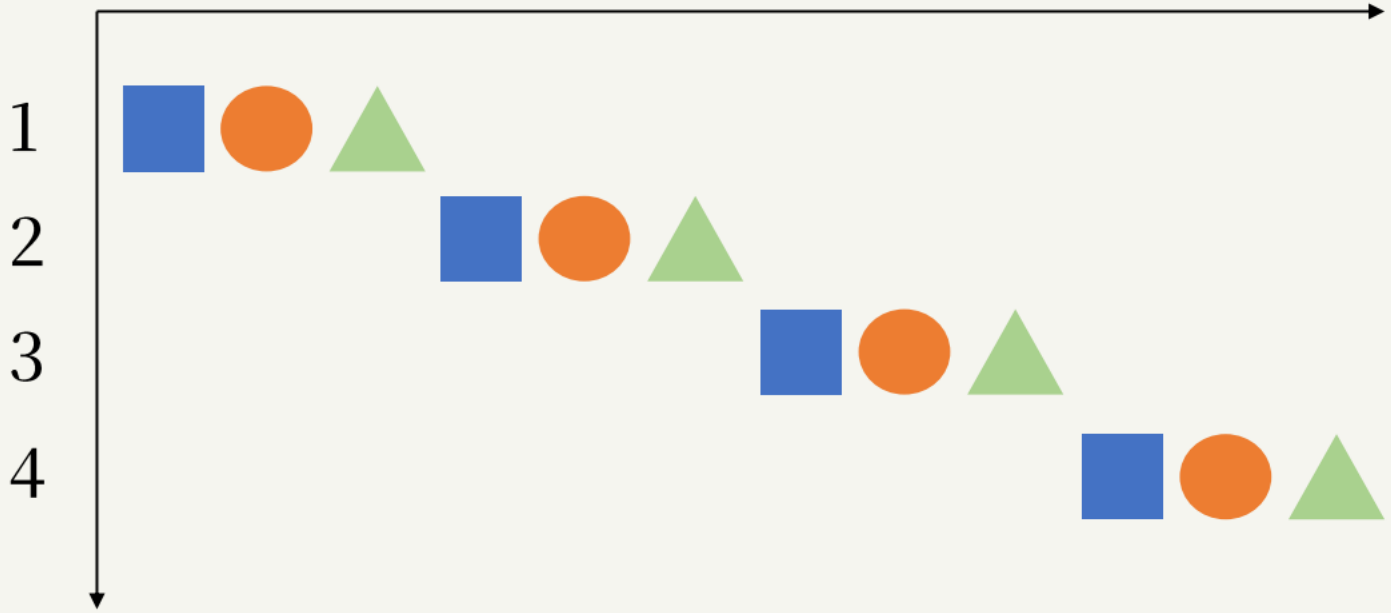


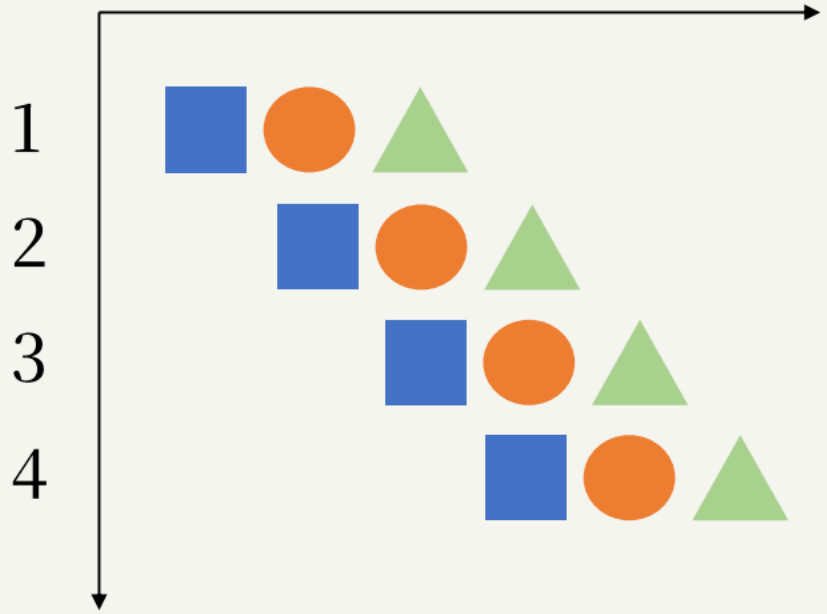
如果每个元件都有2ns的时延，那么执行一条指令的时延就会长达10ns

采用时序逻辑，将一个大逻辑电路划分为几个小逻辑电路（相对独立的几个执行阶段），每个时钟周期（2ns）执行一步

当一条指令执行到后面阶段时，负责前面阶段的元件都会空闲（因为本条指令还没执行完，不能执行下一条指令）

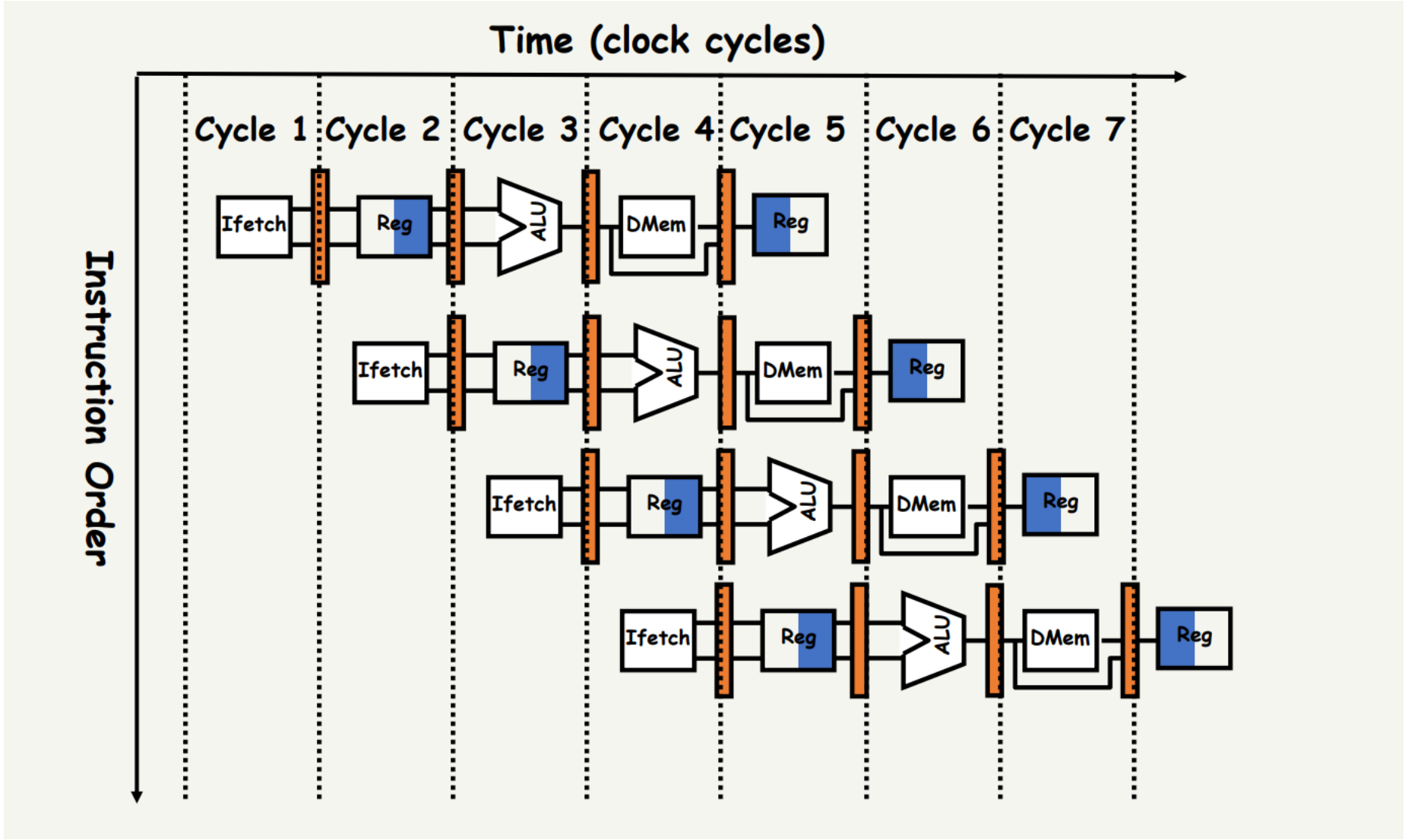
解决方案：流水线





例：当一条指令被送入负责解码的元件后，负责读取指令的元件立刻开始读取下一个指令。

一个简单的五级流水



理想情况下，有几级流水，CPU的吞吐量就能提升几倍

冒险（Hazard）

一旦指令的执行周期存在重叠，就会遇到冒险（或者称之为潜在问题）

数据冒险：由指令间数据的依赖关系引发的各种问题

WAW：写后写，前后两条指令修改同一个寄存器。

RAW：写后读，前面的指令要修改某寄存器的值，后面的指令要用该寄存器的值

WAR：读后写，前面的指令要用某寄存器的值，后面的指令要修改该寄存器的值

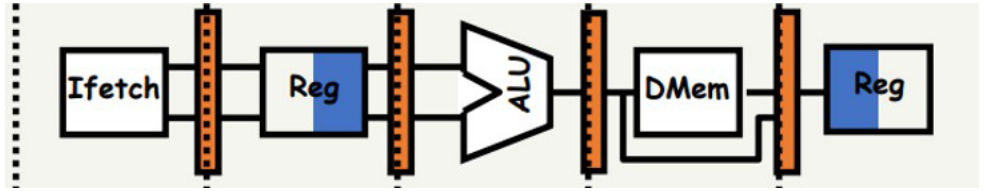
结构冒险：一个元件不能同时处理两条指令的请求。

控制冒险：由跳转指令引发的各种问题。

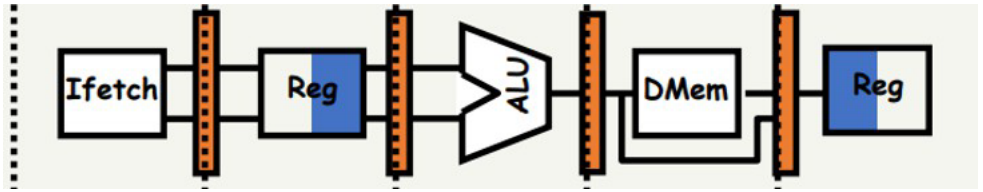
RAW

假设初始值：R1=1；R2=2；R3=3

cycle1 cycle2 cycle3 cycle4 cycle5



ADD R1，R2，R3 （R1=R2+R3）



ADD R4，R1，R3 （R4=R1+R3）

显然，最后R4的值应是8

第二条指令需要R1的新值

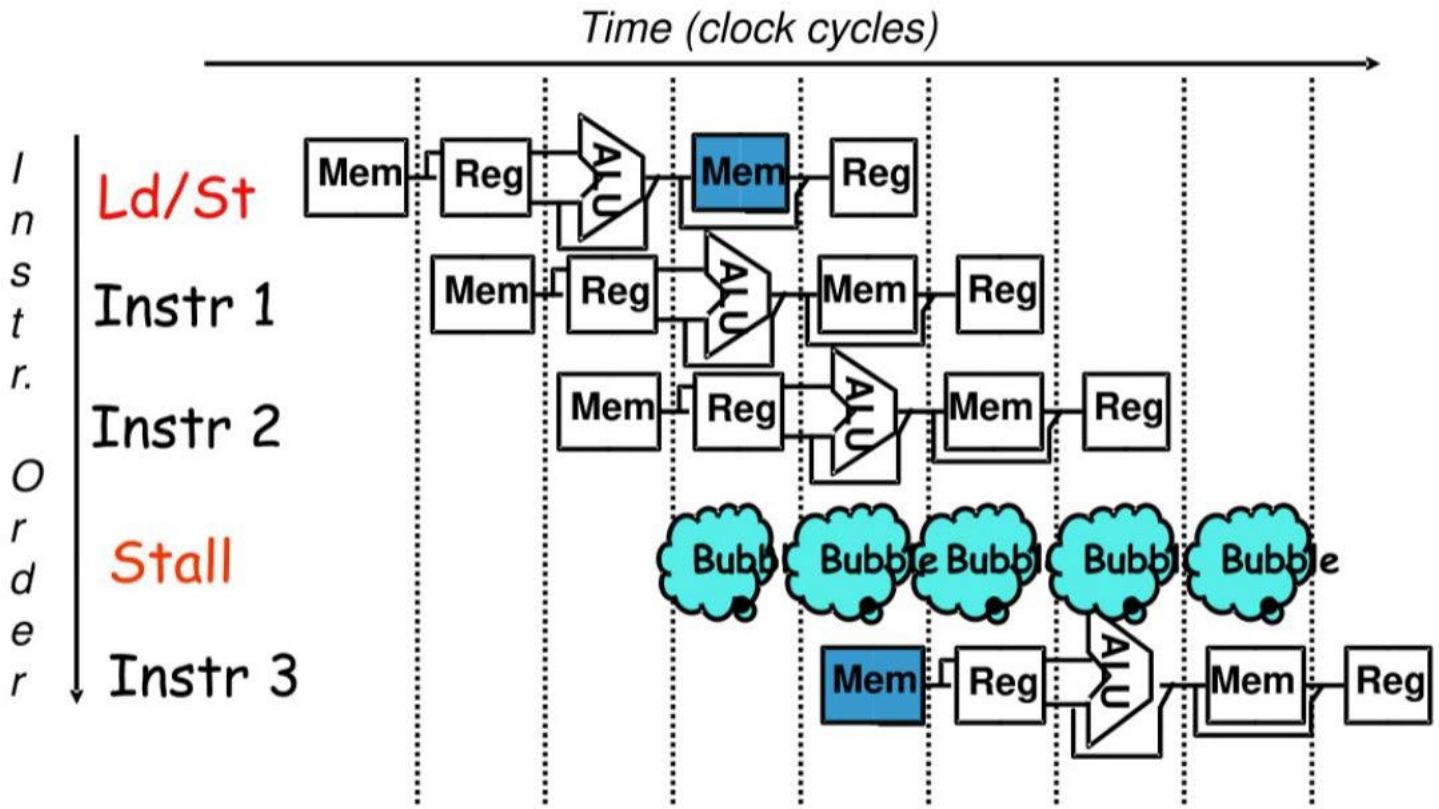
R1的新值要在第5个周期才能被写到寄存器中

但第二条指令在第3个周期就去索要R1的值

如果不管，R4的值会是4

Structural Hazard

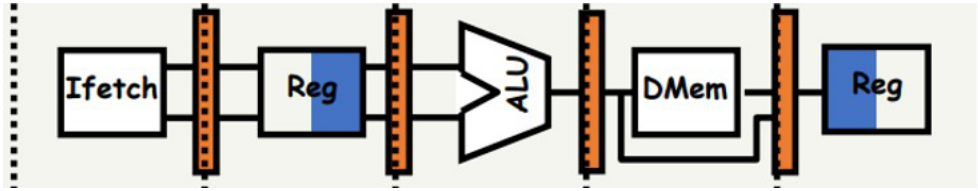
Insert Stall—simplest way



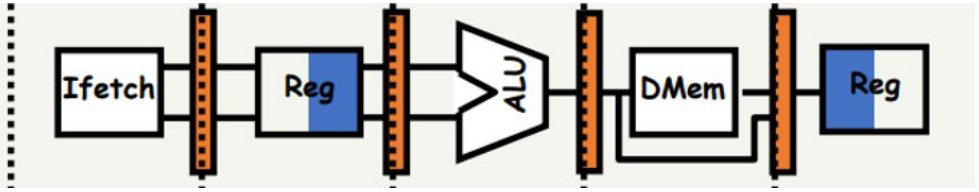
RAW

解决方案1：暂停流水线

cycle1 cycle2 cycle3 cycle4 cycle5



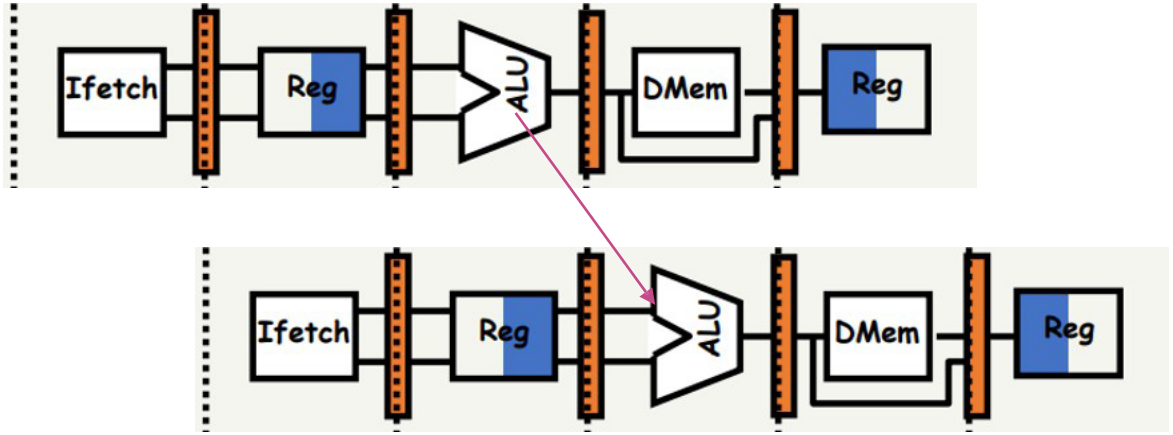
当发现新指令可能会和流水线中的指令出现RAW冒险时，流水线会暂停读入新指令，直到相关指令写入阶段完成后再继续执行新指令



RAW

解决方案2：Forward

cycle1 cycle2 cycle3 cycle4 cycle5



计算出结果后立刻传给需要的部件

WAW

假设初始值：R2=2；R3=3；R4=4

ADD R1，R2，R3　　(R1=R2+R3)

ADD R1，R3，R4　　(R1=R3+R4)

R1的最终值应当为7

如果第二行指令的写入操作先于第一行指令的写入

R1的最终值会变成5

WAR

假设初始值：R1=1；R2=2；R3=3

```
ADD R4, R1, R3        (R4=R1+R3)
ADD R1, R2, R3        (R1=R2+R3)
```

R4的最终值应当为4

如果第二行指令的写入操作先于第一行指令的读取

R4的最终值会变成8

思考：流水线中会出现RAR冒险吗？

回顾：

这三条指令可能引发哪些数据冒险（不一定在流水线中）？

```
ADD   R1, R2, R3        (R1=R2+R3)
ADD   R4, R1, R3        (R4=R1+R3)
ADD   R1, R5, R6        (R1=R5+R6)
```

数据冒险分别存在于哪两条指令之间?

假数据冒险

ADD R1，R2，R3　　(R1=R2+R3)
ADD R4，R1，R3　　(R4=R1+R3)
ADD R1，R5，R6　　(R1=R5+R6)

有WAW和WAR冒险，如果第三行指令先于第一行或第二行指令完成，将会出错。

这三行指令的执行结果：R4=R2+2R3；R1=R2+R3

第一行指令仅仅是为了给第二行的指令提供源数据，随即被第三行指令的结果覆盖

如果我们在这些通用寄存器外新增一个寄存器T1

ADD T1，R2，R3　　(T1=R2+R3)
ADD R4，T1，R3　　(R4=T1+R3)
ADD R1，R5，R6　　(R1=R5+R6)

和上面三行的效果是完全一致的，并且没有数据冒险

继续提升效率？

考虑如下三条相邻的指令：

```
LW    R1，0x4000      (将一个内存地址中的值读到R1寄存器中)
ADD   R5，R1，R3      (算出R1与R3的和，存到R5寄存器中)
ADD   R2，R3，R4      (算出R3与R4的和，存到R2寄存器中)
```

第一条lw执行起来很慢。

第二行的add指令需要R1的新值，需等待第一条指令执行完毕。

但第三行的add指令和前两条没有任何依赖关系，此时EX，WB都闲置，其实它已经能执行了。

然而，因为前两条指令在流水线的前面，第三条只能被卡着。

乱序执行：Tomasulo算法

想法：指令不必等到前面的指令都执行完才能执行；只要所需数据准备好，就可以执行

（但指令发射与提交还是顺序的）

算法的五个阶段：

指令获取(fetch) -> 指令发射(issue) -> 指令执行(exec) -> 指令结果写入(write&broadcast) ->指令提交(commit)

算法的几个部件：

Decoder	ReorderBuffer(ROB)	ReservationStation(RS)
LoadStoreBuffer(LSB)	ALU	
RegisterFile(RF)	Predictor	

寄存器重命名

RAW冒险的本质原因，是后面的指令需要前面指令算出来的数据

而WAW、WAR不包含真正的数据流动，只是被覆盖的值可能还会被用到。

只要保证最后寄存器的状态是正确的，新值旧值暂存在哪里都无所谓。

通过增加一些寄存器，使得不影响新值立刻写入的同时，仍允许获取旧值

更宽泛地说，只需要保证覆盖前的指令能正确获取到旧值就行。

顺序提交-RoB

只需要用一个循环队列暂存一下正在执行的所有指令和对应的目标寄存器、指令执行结果。一旦队头指令执行完毕，就用其结果更新目标寄存器。

指令发射（被读取的暂存于CPU中的指令开始被执行）时入队，提交时出队。

该结构被称为RoB（Reorder Buffer，重排序缓冲）

正在执行的指令和其在RoB中的编号是一一对应的。

	Reorder buffer					
Entry	Busy	Instruction		State	Destination	Value
1	No	fld	f6,32(x2)	Commit	f6	Mem[32 + Regs[x2]]
2	No	fld	f2,44(x3)	Commit	f2	Mem[44 + Regs[x3]]
3	Yes	fmul.d	f0,f2,f4	Write result	f0	#2 × Regs[f4]
4	Yes	fsub.d	f8,f2,f6	Write result	f8	#2 – #1
5	Yes	fdiv.d	f0,f0,f6	Execute	f0	
6	Yes	fadd.d	f6,f8,f2	Write result	f6	#4 + #2

实现并行、维护依赖-Reservation Station

在指令发射后，会进入RS中。RS会维护这条指令的数据依赖关系。

	Reservation stations							
Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
Load1	No							
Load2	No							
Add1	No							
Add2	No							
Add3	No							
Mult1	No	fmul.d	Mem[44 + Regs[x3]]	Regs[f4]			#3	
Mult2	Yes	fdiv.d		Mem[32 + Regs[x2]]	#3		#5	

- Vj，Vk：真正参与该指令运算的值（操作数，指令要操作的数据）
- Qj，Qk：记录该指令的操作数（即寄存器/内存地址）需要靠哪个正在执行的指令算出来（用RoB中的编号来表示）。若为空，则说明对应的值已准备好。
- Dest：该指令在RoB中的编号。
- A：记录立即数或地址。
- Op：指令类型。

RS直接连接着ALU，当一条指令变为可执行状态（Qj,Qk均为空，表明操作数都已准备好）时，将其送入ALU中进行计算，同时清空该行。ALU算完后直接把结果交到RoB。

实现并行、维护依赖-Reservation Station

在一条指令的发射过程中，会向Register File询问其源寄存器（i.e.向哪个寄存器获取所需值）是否有依赖。若有，则将对应位置的Q更新为该寄存器的依赖；否则直接获取寄存器的值，填入对应的V中。

询问过程可以在Decode时就进行，也可以在送入RS后再进行，请自行设计

在一条指令（不妨设其RoB编号为#1）被RoB提交时，RS会检查所有栏位中是否有依赖该指令的操作数（i.e. Q==1），若有，则将它们的Q置空，对应的V更新为提交的值。

RS的核心思想是在一个操作数的值可用时马上获取它，这样就不用再从寄存器里获取了。这种方法事实上是一种寄存器重命名，用RS相关行的V值暂存旧值。

延迟指令的执行，直到其操作数可用，消除了RAW冒险。

记录依赖关系-Register File

CPU的通用寄存器们在Register File中。Register File还可以存储并维护一些有关通用寄存器的信息。

在Tomasulo架构中，为每个通用寄存器记录并维护其依赖（i.e.其最新值将由哪条正在执行的指令算出）

可以用RoB中的编号来代表指令

当一条指令被发射时，将其目标寄存器的依赖修改为其在RoB中的编号

当一条指令被提交时，如果目标寄存器的依赖仍是该指令，则将其设置为无依赖，并更新目标寄存器的值

访存指令特殊处理-LSB

LSB（LoadStoreBuffer），顾名思义，用来专门处理**访存**指令。(Load/Store)

其结构功能和RS差不多，只不过连的不是ALU，是RAM。

访存指令也同样存在Data Hazard。所以处理时要小心，不能像RS一样操作数准备好了就执行。

有完全顺序、部分乱序、全乱序三种策略，请自行设计。

本次作业中的访存指令均需要耗费三个周期才能成功执行！这与跳转/运算指令不同。

Control Hazard

条件跳转指令后面的指令无法确定。

一个naïve的策略：在读取指令时，判断是否是条件跳转。如果是，则暂停指令读取，直到传来该跳转指令的提交信息，再根据其结果更改PC。

分支预测-Predictor

希望即使遇到分支程序仍能继续运行！**先猜一个**

如果预测正确，正常进行；如果预测错误?

分支预测出错如何处理？

清空所有未commit内容

分支预测-Predictor

希望即使遇到分支程序仍能继续运行！**先猜一个**

CPU执行的大多数指令都是循环中的指令。

多次运行同一个跳转指令，其跳转与否一般存在规律.

分支预测器会监视中指令的提交，如果预测正确，正常进行；如果预测错误，立刻向指令读取元件发送正确的PC，同时向所有元件发出flush信号，以清空元件中的错误指令。

之后由HRY来为我们带来更多相关内容。

乱序执行：Tomasulo算法

如果你还是不太明白这个算法是在干什么的话，我们简单总结一下每个模块和算法的阶段是如何对应的.....

【回顾】

算法的五个阶段：

指令获取(fetch) -> 指令发射(issue) -> 指令执行(exec) -> 指令结果写入(write&broadcast) ->指令提交(commit)

算法的几个部件：

Decoder ReorderBuffer(ROB)	ReservationStation(RS)
LoadStoreBuffer(LSB)	ALU
RegisterFile(RF)	Predictor

乱序执行：Tomasulo算法

如果你还是不太明白这个算法是在干什么的话，我们简单总结一下每个模块和算法的阶段是如何对应的.....

算法的几个阶段：

1.fetch：顺序读取指令，你可以通过指令的预处理/缓存来提高效率

Decoder -> 对指令进行解码

2.issue：指令发射

ROB/LSB -> 指令入队，进行指令的编号

RS -> 会维护这条指令的数据依赖关系，Vi/Vj/Qi/Qj的对应

Reg -> 将目标寄存器重命名为ROB中的编号

3.exec：ALU -> 执行指令目标的计算/访存指令开始访问内存（三个周期）

乱序执行：Tomasulo算法

4.write&broadcast:

ROB -> 更改指令完成情况，存储指令结果

向其他元件广播结果，依赖于该条指令的其他指令可以减除依赖

LSB -> 内存的写入/读取

5.commit：提交完成的指令，这里的指令提交是按照rob里的顺序（因此保证了指令提交的正确性），如果rob.front()没有准备好则没有指令提交。

RF -> 删除寄存器对这条指令的依赖

ROB -> 指令出队

Predictor -> 对于跳转指令，如果指令的结果和预测不相同，则清空整条流水线， 从跳转地址处开始重新执行

乱序执行：Tomasulo架构

问题1：乱序执行的时候，顺序执行不需要考虑的WAW、WAR冒险会出现。

方案1：像流水线处理冒险一样，在检查到WAW、WAR时进行停顿？

Tomasulo架构有更聪明的算法：寄存器重命名

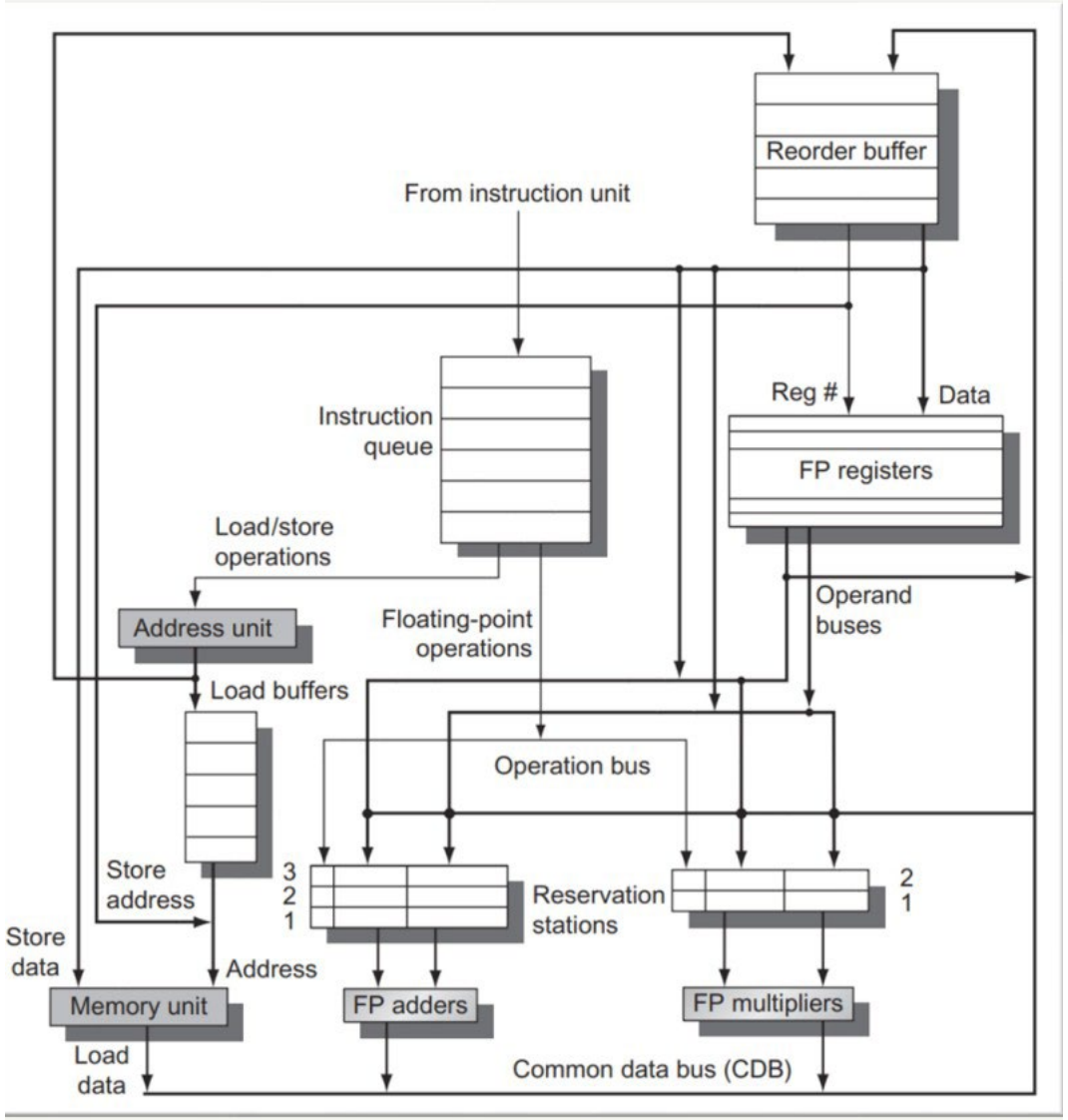
问题2：冯诺依曼架构承诺指令会被顺序执行。

重排序缓冲

问题3：怎么确认“数据准备好”（即维护依赖关系）？

Reservation Station、Register File和广播。

参考架构（CAAQA）


Reorder buffer
From instruction unit
Instruction queue
Reg #
Data
FP registers
Load/store operations
Address unit
Floating-point operations
Operand buses
Load buffers
Operation bus
Store address
3
2
1
Reservation stations
2
1
Store data
Address
Memory unit
FP adders
FP multipliers
Load data
Common data bus (CDB)

举个例子

RS

- `ADD R1, R2, R3`
- `ADD R7, R1, R6`

Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
RS1	No							
RS2	No							

Field	R1	R2	R3	...	R6	R7
Value	0	4000	396		1	0
Reorder	0	0	0		0	0

RS

- `ADD R1, R2, R3`
- `ADD R7, R1, R6`

Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
RS1	Yes	ADD	4000	396			R1	
RS2	No							

Field	R1	R2	R3	…	R6	R7
Value	0	4000	396		1	0
Reorder	RS1	0	0		0	0

RS

- `ADD R1, R2, R3`
- `ADD R7, R1, R6`

Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
RS1	Yes	ADD	4000	396			R1	
RS2	No							

Field	R1	R2	R3	...	R6	R7
Value	0	4000	396		1	0
Reorder	RS1	0	0		0	0

RS

- `ADD R1, R2, R3`
- `ADD R7, R1, R6`

Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
RS1	No	ADD	4000	396			R1	
RS2	No							

→ ALU

Field	R1	R2	R3	...	R6	R7
Value	0	4000	396		1	0
Reorder	RS1	0	0		0	0

RS

- `ADD R1, R2, R3`
- `ADD R7, R1, R6`

Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
RS1	No	ADD	4000	396			R1	
RS2	Yes	ADD		R6	RS1		R7	

→ ALU

Field	R1	R2	R3	...	R6	R7
Value	0	4000	396		1	0
Reorder	RS1	0	0		0	RS2

RS

- `ADD R1, R2, R3`
- `ADD R7, R1, R6`

Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
RS1	No	ADD	4000	396			R1	
RS2	Yes	ADD		R6	RS1		R7	

→ DONE

Field	R1	R2	R3	...	R6	R7
Value	0	4000	396		1	0
Reorder	RS1	0	0		0	RS2

RS

- `ADD R1, R2, R3`
- `ADD R7, R1, R6`

Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
RS1	No	ADD	4000	396			R1	
RS2	No	ADD	4396	1	0		R7	

→ ALU

Field	R1	R2	R3	...	R6	R7
Value	4396	4000	396		1	0
Reorder	0	0	0		0	RS2

RS

- `ADD R1, R2, R3`
- `ADD R7, R1, R6`

Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
RS1	No	ADD	4000	396			R1	
RS2	No	ADD	4396	1	0		R7	

Field	R1	R2	R3	...	R6	R7
Value	4396	4000	396		1	4397
Reorder	0	0	0		0	0

再看看ROB
是如何变化的

ROB

```
L1: DIV R1, R2, R3
L2: BEQ R1, R4, 0
L3: ADD R7, R5, R6
```

Name	Busy	Op	Vj	Vk	Qj	Qk	Reorder	A
RS1	No							
RS2	No							
RS3	No							

Op	rs1	rs2	Reorder

Field	R1	...	R7
Value	666		233
Reorder			

Entry	Busy	Instruction	Ready	Dest	Value
1					
2					
3					

ROB

```
L1: DIV R1, R2, R3
L2: BEQ R1, R4, 0
L3: ADD R7, R5, R6
```

Name	Busy	Op	Vj	Vk	Qj	Qk	Reorder	A
RS1	Yes	DIV	R2	R3			1	
RS2	No							
RS3	No							

Op	rs1	rs2	Reorder

Field	R1	...	R7
Value	666		233
Reorder	1		

Entry	Busy	Instruction	Ready	Dest	Value
1	Yes	L1	No	R1	
2					
3					

ROB

```
L1: DIV R1, R2, R3
L2: BEQ R1, R4, 0
L3: ADD R7, R5, R6
```

Name	Busy	Op	Vj	Vk	Qj	Qk	Reorder	A
RS1	Yes	DIV	R2	R3			1	
RS2	Yes	BEQ		R4	1		2	0
RS3	No							

Op	rs1	rs2	Reorder
DIV	R2	R3	1

Field	R1	...	R7
Value	666		233
Reorder	1		

Entry	Busy	Instruction	Ready	Dest	Value
1	Yes	L1	No	R1	
2	Yes	L2	No		
3					

ROB

```
L1: DIV R1, R2, R3
L2: BEQ R1, R4, 0
L3: ADD R7, R5, R6
```

Name	Busy	Op	Vj	Vk	Qj	Qk	Reorder	A
RS1	Yes	DIV	R2	R3			1	
RS2	Yes	BEQ		R4	1		2	0
RS3	Yes	ADD	R5	R6			3	

Op	rs1	rs2	Reorder
DIV	R2	R3	1

Field	R1	...	R7
Value	666		233
Reorder	1		3

Entry	Busy	Instruction	Ready	Dest	Value
1	Yes	L1	No	R1	
2	Yes	L2	No		
3	Yes	L3	No	R7	

ROB

```
L1: DIV R1, R2, R3
L2: BEQ R1, R4, 0
L3: ADD R7, R5, R6
```

Name	Busy	Op	Vj	Vk	Qj	Qk	Reorder	A
RS1	Yes	DIV	R2	R3			1	
RS2	Yes	BEQ		R4	1		2	0
RS3	Yes	ADD	R5	R6			3	

Op	rs1	rs2	Reorder
DIV	R2	R3	1
ADD	R5	R6	3

Field	R1	...	R7
Value	666		233
Reorder	1		3

Entry	Busy	Instruction	Ready	Dest	Value
1	Yes	L1	No	R1	
2	Yes	L2	No		
3	Yes	L3	No	R7	

ROB

```
L1: DIV R1, R2, R3
L2: BEQ R1, R4, 0
L3: ADD R7, R5, R6
```

Name	Busy	Op	Vj	Vk	Qj	Qk	Reorder	A
RS1	Yes	DIV	R2	R3			1	
RS2	Yes	BEQ		R4	1		2	0
RS3	No	ADD	R5	R6			3	

Op	rs1	rs2	Reorder
DIV	R2	R3	1

Field	R1	...	R7
Value	666		233
Reorder	1		3

Entry	Busy	Instruction	Ready	Dest	Value
1	Yes	L1	No	R1	
2	Yes	L2	No		
3	Yes	L3	Yes	R7	2333

ROB

```
L1: DIV R1, R2, R3
L2: BEQ R1, R4, 0
L3: ADD R7, R5, R6
```

Name	Busy	Op	Vj	Vk	Qj	Qk	Reorder	A
RS1	No	DIV	R2	R3			1	
RS2	Yes	BEQ	6666	R4	0		2	0
RS3	No	ADD	R5	R6			3	

Op	rs1	rs2	Reorder

Field	R1	...	R7
Value	666		233
Reorder	1		3

Entry	Busy	Instruction	Ready	Dest	Value
1	Yes	L1	Yes	R1	6666
2	Yes	L2	No		
3	Yes	L3	Yes	R7	2333

ROB

```
L1: DIV R1, R2, R3
L2: BEQ R1, R4, 0
L3: ADD R7, R5, R6
```

Name	Busy	Op	Vj	Vk	Qj	Qk	Reorder	A
RS1	No	DIV	R2	R3			1	
RS2	Yes	BEQ	6666	R4	0		2	0
RS3	No	ADD	R5	R6			3	

Op	rs1	rs2	Reorder

Field	R1	...	R7
Value	6666		233
Reorder	0		3

Entry	Busy	Instruction	Ready	Dest	Value
1	No	L1	Yes	R1	6666
2	Yes	L2	No		
3	Yes	L3	Yes	R7	2333

ROB

```
L1: DIV R1, R2, R3
L2: BEQ R1, R4, 0
L3: ADD R7, R5, R6
```

Name	Busy	Op	Vj	Vk	Qj	Qk	Reorder	A
RS1	No	DIV	R2	R3			1	
RS2	Yes	BEQ	6666	R4	0		2	0
RS3	No	ADD	R5	R6			3	

Op	rs1	rs2	Reorder
BEQ	6666	R4	2

Field	R1	...	R7
Value	6666		233
Reorder	0		3

Entry	Busy	Instruction	Ready	Dest	Value
1	No	L1	Yes	R1	6666
2	Yes	L2	No		
3	Yes	L3	Yes	R7	2333

ROB

```
L1: DIV R1, R2, R3
L2: BEQ R1, R4, 0
L3: ADD R7, R5, R6
```

- 如果需要跳转

Name	Busy	Op	Vj	Vk	Qj	Qk	Reorder	A
RS1	No	DIV	R2	R3			1	
RS2	No	BEQ	6666	R4	0		2	0
RS3	No	ADD	R5	R6			3	

Op	rs1	rs2	Reorder

Field	R1	...	R7
Value	6666		233
Reorder	0		3

Entry	Busy	Instruction	Ready	Dest	Value
1	No	L1	Yes	R1	6666
2	Yes	L2	Yes		1
3	Yes	L3	Yes	R7	2333

ROB

```
L1: DIV R1, R2, R3
L2: BEQ R1, R4, 0
L3: ADD R7, R5, R6
```

- 如果需要跳转

Name	Busy	Op	Vj	Vk	Qj	Qk	Reorder	A
RS1	No	DIV	R2	R3			1	
RS2	No	BEQ	6666	R4	0		2	0
RS3	No	ADD	R5	R6			3	

Op	rs1	rs2	Reorder

Field	R1	...	R7
Value	6666		233
Reorder	0		0

Entry	Busy	Instruction	Ready	Dest	Value
1	No	L1	Yes	R1	6666
2	No	L2	Yes		1
3	No	L3	Yes	R7	2333

ROB

```
L1: DIV R1, R2, R3
L2: BEQ R1, R4, 0
L3: ADD R7, R5, R6
```

- 如果不需要跳转

Name	Busy	Op	Vj	Vk	Qj	Qk	Reorder	A
RS1	No	DIV	R2	R3			1	
RS2	No	BEQ	6666	R4	0		2	0
RS3	No	ADD	R5	R6			3	

Op	rs1	rs2	Reorder

Field	R1	...	R7
Value	6666		233
Reorder	0		3

Entry	Busy	Instruction	Ready	Dest	Value
1	No	L1	Yes	R1	6666
2	Yes	L2	Yes		0
3	Yes	L3	Yes	R7	2333

ROB

```
L1: DIV R1, R2, R3
L2: BEQ R1, R4, 0
L3: ADD R7, R5, R6
```

- 如果不需要跳转

Name	Busy	Op	Vj	Vk	Qj	Qk	Reorder	A
RS1	No	DIV	R2	R3			1	
RS2	No	BEQ	6666	R4	0		2	0
RS3	No	ADD	R5	R6			3	

Op	rs1	rs2	Reorder

Field	R1	...	R7
Value	6666		233
Reorder	0		3

Entry	Busy	Instruction	Ready	Dest	Value
1	No	L1	Yes	R1	6666
2	No	L2	Yes		0
3	Yes	L3	Yes	R7	2333

ROB

```
L1: DIV R1, R2, R3
L2: BEQ R1, R4, 0
L3: ADD R7, R5, R6
```

- 如果不需要跳转

Name	Busy	Op	Vj	Vk	Qj	Qk	Reorder	A
RS1	No	DIV	R2	R3			1	
RS2	No	BEQ	6666	R4	0		2	0
RS3	No	ADD	R5	R6			3	

Op	rs1	rs2	Reorder

Field	R1	...	R7
Value	6666		2333
Reorder	0		0

Entry	Busy	Instruction	Ready	Dest	Value
1	No	L1	Yes	R1	6666
2	No	L2	Yes		0
3	No	L3	Yes	R7	2333

作业内容

作业内容

用C++模拟一个Tomasulo架构的RISC-V CPU。

通过所有下发数据。

评分标准

75%测试点得分+15%理解得分

如果未实现分支预测，或有主要功能缺陷：60%测试点得分+10%理解得分

10%CR

10%Bonus

执行流程

从标准输入读入机器指令

从内存0x0000处开始取指令执行，每次连取4个byte，拼成一条指令

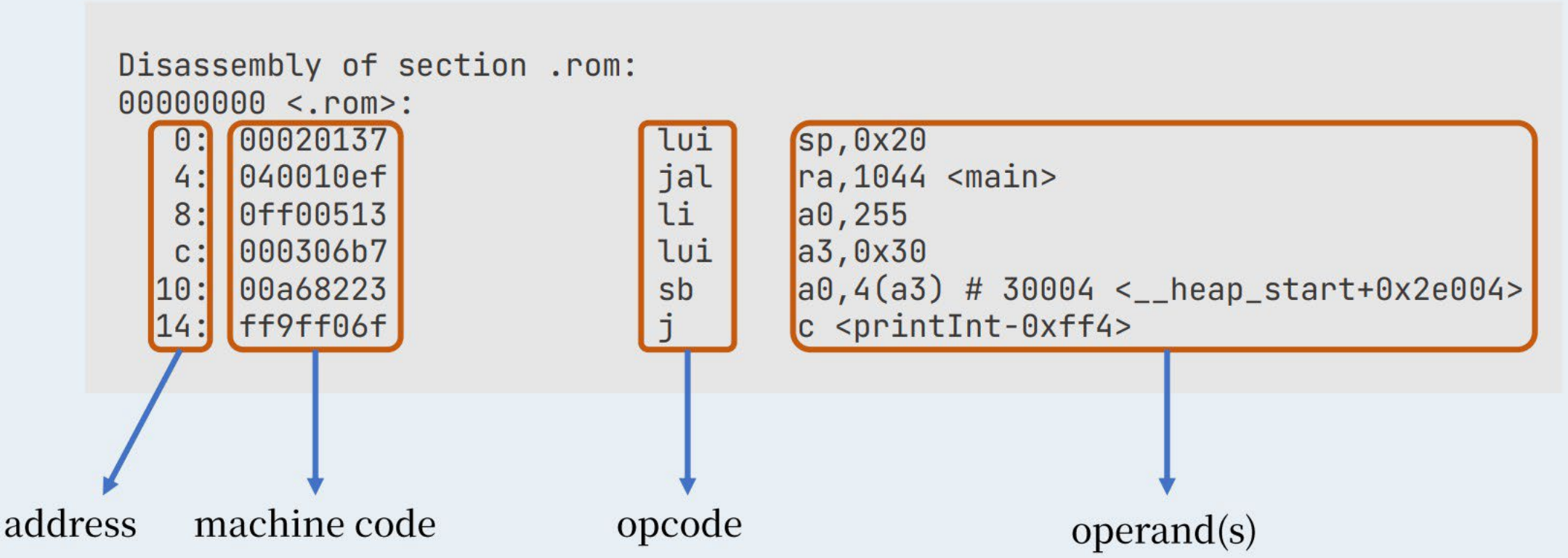
执行到指令0x0ff00513（li a0 255）时，向stdout输出程序的返回值。
返回值应当是一个0-255的非负整数。

程序的返回值是a0寄存器的后8位

下发文件

.data :内存中的指令，是你simulator的实际输入

为了方便理解指令在干什么，每个.data文件还会伴有同名的.c文件和.dump文件
.c :源代码
.dump :数据对应的汇编指令



一些提示和注意事项

请务必仔细阅读RISC-V手册中的相关内容，理解其特性和规范！

RISC-V采用小端字节序，如果你在一个地址（4byte对齐）中存储 0x01020304，那么其在内存中的样子是：

04 03 02 01

同样地，.data文件中前四个byte为 37 01 02 00，对应的指令码为 0x00020137

寄存器 x0 每周期重置回 0

更详细地说

你应当为每个Tomasulo架构中的元件写一个类

- RoB
- RS
- LSB
- Memory
- ALUs

…

这些类中，应至少有一个成员函数，包含这个元件在每个周期中的行为

各部件的存储结构？

要求

尽可能遵循硬件要求

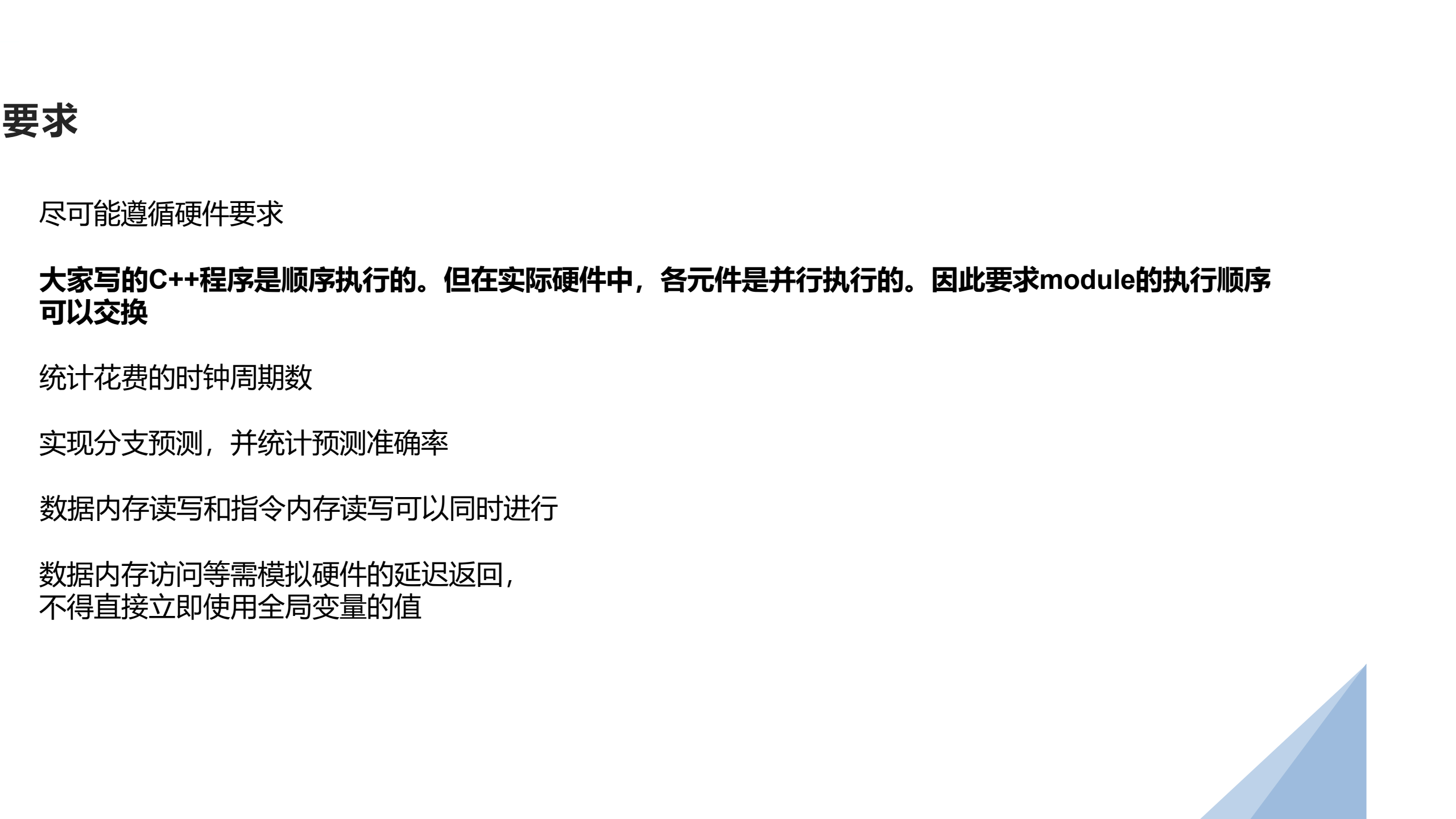
大家写的C++程序是顺序执行的。但在实际硬件中，各元件是并行执行的。因此要求module的执行顺序可以交换

统计花费的时钟周期数

实现分支预测，并统计预测准确率

数据内存读写和指令内存读写可以同时进行

数据内存访问等需模拟硬件的延迟返回，
不得直接立即使用全局变量的值



仅作参考

```
int main() {
    // 进行一些初始化操作
    // 把输入文件的指令读进“内存”中
    while (true){
        //CPU运行一步
    }
}
```

你的main函数可能长这个样子

```
void function_name(){
    ++clk;
    //一些操作，比如用新值更新旧值

    //元件1运行
    //元件2运行
    //...
    //元件n运行

    //一些操作，更新旧值也可以在这里进行
}
```

“CPU运行一步”大概是这个样子

Code Review时，会将元件的执行顺序随机交换

一些提示和注意事项

模拟硬件思想！

一个元件只能通过自己当前状态和外部输入来改变状态、进行输出

请勿使用绝大多数STL容器、指针、引用，或动态分配内存空间

避免全局变量

想清楚Verilog中的对应（Reg/Wire）

模拟器运行耗时短，并不意味着模拟器写得好。

可以尝试在保证Tomasulo架构的情况下，尽可能减少运行的时钟周期数。

（并不会影响得分）

建议：各部件存储新、旧两个状态，执行时用自己和其他部件的旧状态运算出新状态，更新时用新状态覆盖（模拟reg随时钟周期更新的时序逻辑行为）

一些提示和注意事项

想清楚每种指令都怎么处理之后再写，可以画一张设计图，体现各元件的功能、接线

不同元件之间传输哪些信息

建议先写一个单级流水的 naïve interpreter!

①熟悉 RISC-V 指令功能

②方便 debug 对拍：比较每 commit 一条指令后寄存器的状态。

严禁AI生成代码或自动补全!

请勤于 git commit

一些提示和注意事项

边际情况处理：

Sanity examples:

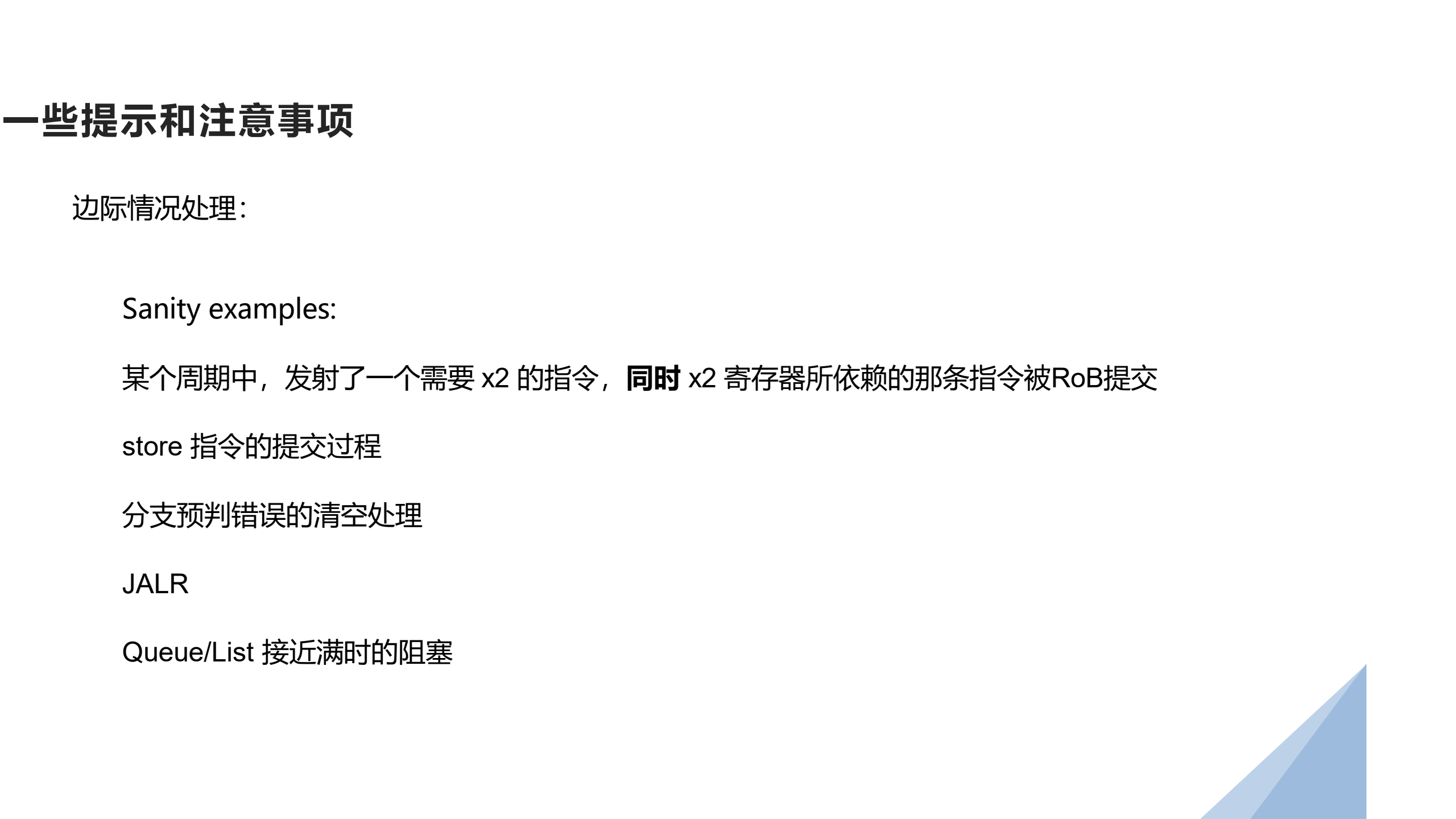
某个周期中，发射了一个需要 x2 的指令，**同时** x2 寄存器所依赖的那条指令被RoB提交

store 指令的提交过程

分支预判错误的清空处理

JALR

Queue/List 接近满时的阻塞



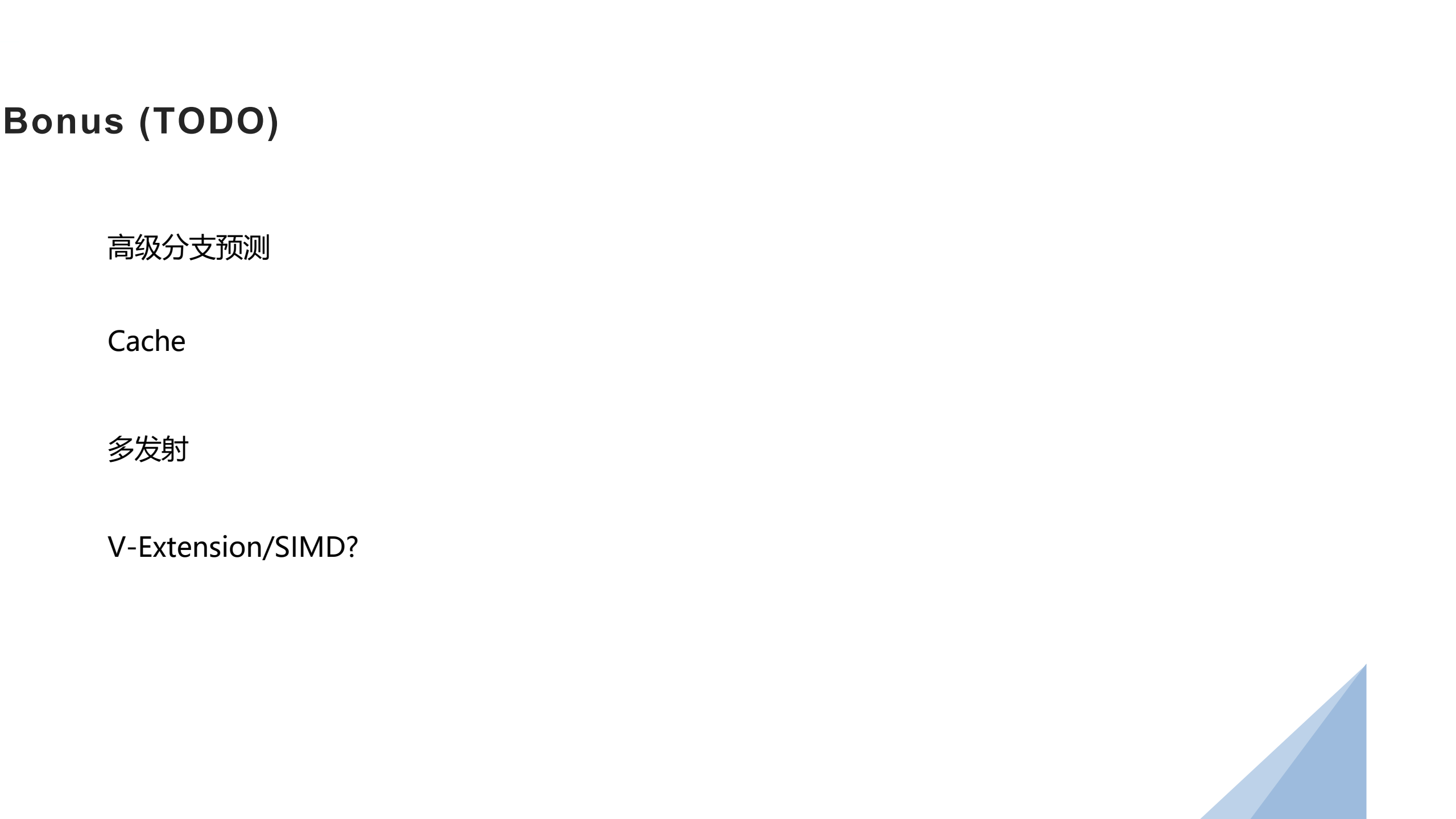
Bonus (TODO)

高级分支预测

Cache

多发射

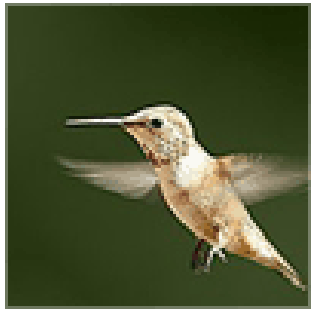
V-Extension/SIMD?



负责助教

马翊加

李芷妍



胡瑞岩

(Bonus)

推荐阅读：

CAAQA 第三章：指令级并行

计算机体系结构-存储指令的加速 - 知乎

同作者系列文章

https://www.bilibili.com/video/av21376839/ 第五到第九集