

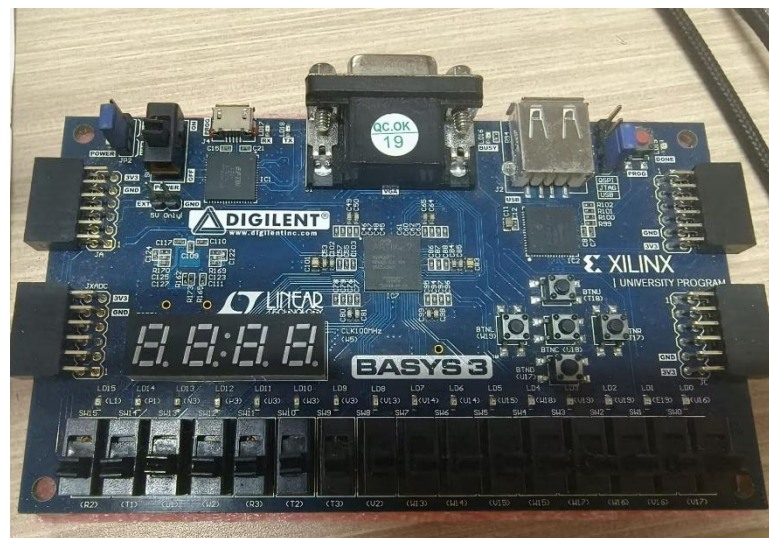
RISC-V CPU Simulator

2024.7.22

陈轶凡 张煜轩

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



硬件

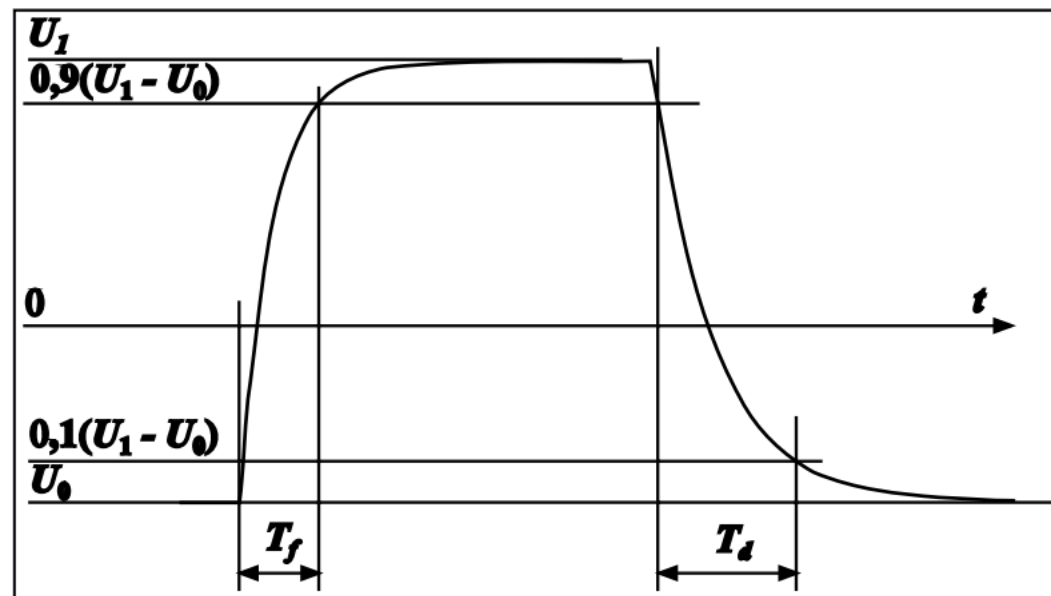
时延

逻辑电路并不能在接受输入时瞬间就能产生相应输出。

从改变输入信号到获得平稳输出信号需要一定时间，这段时间被称为时延。

时延大小和逻辑电路的复杂程度呈正相关

复杂的电路通常包含数个不同的部件，它们电路复杂度有差异，产生的时延也不同，强行将它们连接在一起，会产生各种问题。



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。

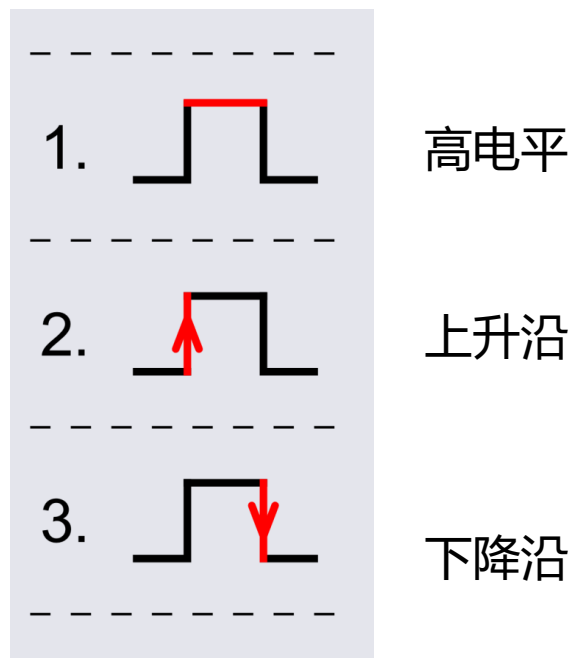
时钟周期即为一次时钟信号的完整周期。

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

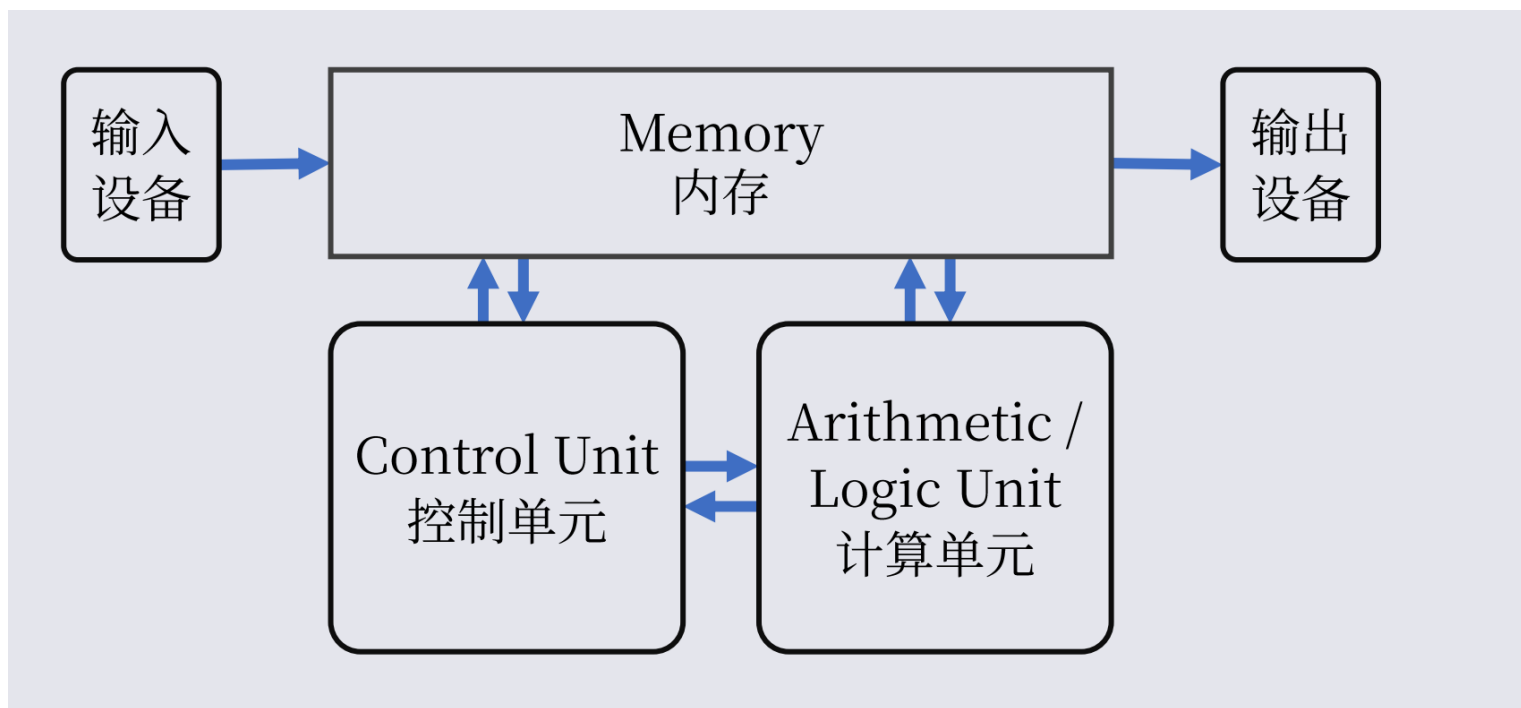
每个时钟周期，根据输入和自己当前状态，进行输出和状态改变

- 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。

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



指令集架构

Instruction Set Architecture

什么是指令集架构？

指令集是一个计算机系统支持的所有机器指令的集合。

- 规定指令的格式和功能
- 规定寄存器的数量、名称和用途

指令集架构大致可分为CISC和RISC两大阵营：

CISC的思想是通过设计一些功能复杂的指令，把原本由软件执行的一些操作交由硬件来实现，以此提高程序的运行速度。（比如x86指令集）

RISC的思想则是尽量简化计算机指令功能，只保留那些功能简单的指令，可以通过硬件优化设计，把时钟频率提得很高，从而实现整个系统的高性能。

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

注意x0寄存器!!!

测试程序中存在以x0为目标的指令，执行这些指令时，应无视其对x0的修改。

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，通过一次访问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

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。

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。

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 9:12]							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 lbu	
imm[11:5]		rs2		rs1		000		imm[4:0]		0100011		S sb	
imm[11:5]		rs2		rs1		001		imm[4:0]		0100011		S sh	
imm[11:5]		rs2		rs1		010		imm[4:0]		0100011		S sw	
imm[11:0]				rs1		000		rd		0010011		I addi	
imm[11:0]				rs1		010		rd		0010011		I slti	
imm[11:0]				rs1		011		rd		0010011		I sltiu	
imm[11:0]				rs1		100		rd		0010011		I xori	
imm[11:0]				rs1		110		rd		0010011		I ori	
imm[11:0]				rs1		111		rd		0010011		I andi	
0000000		shamt		rs1		001		rd		0010011		I slli	
0000000		shamt		rs1		101		rd		0010011		I srli	
0100000		shamt		rs1		101		rd		0010011		I srai	
0000000		rs2		rs1		000		rd		0110011		R add	
0100000		rs2		rs1		000		rd		0110011		R sub	
0000000		rs2		rs1		001		rd		0110011		R sll	
0000000		rs2		rs1		010		rd		0110011		R slt	
0000000		rs2		rs1		011		rd		0110011		R sltu	
0000000		rs2		rs1		100		rd		0110011		R xor	
0000000		rs2		rs1		101		rd		0110011		R srl	
0100000		rs2		rs1		101		rd		0110011		R sra	
0000000		rs2		rs1		110		rd		0110011		R or	
0000000		rs2		rs1		111		rd		0110011		R and	

几个典型指令

```
lui    rd, immediate      x[rd] = sext(immediate[31:12] << 12)
```

高位立即数加载 (*Load Upper Immediate*). U-type, RV32I and RV64I.

将符号位扩展的 20 位立即数 *immediate* 左移 12 位，并将低 12 位置零，写入 *x[rd]* 中。

压缩形式: **c.lui** rd, imm



```
auipc rd, immediate      x[rd] = pc + sext(immediate[31:12] << 12)
```

PC 加立即数 (Add Upper Immediate to PC). U-type, RV32I and RV64I.

把符号位扩展的 20 位（左移 12 位）立即数加到 pc 上，结果写入 $x[rd]$ 。



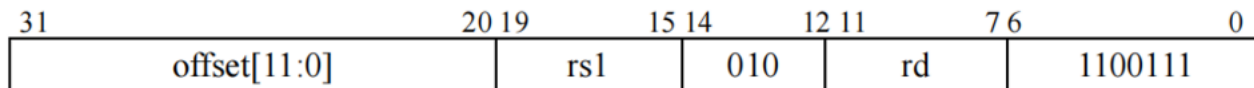
几个典型指令

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

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

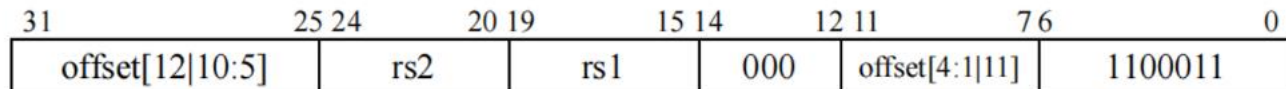


beq rs1, rs2, offset $if (rs1 == rs2) pc += sext(offset)$

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

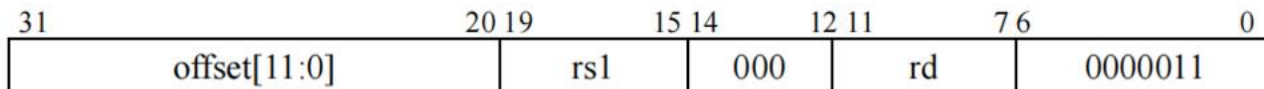


几个典型指令

lb rd, offset(rs1) $x[rd] = \text{sext}(M[x[rs1] + \text{sext}(\text{offset})][7:0])$

字节加载 (*Load Byte*). I-type, RV32I and RV64I.

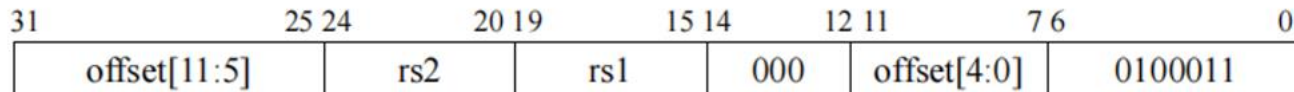
从地址 $x[rs1] + \text{sign-extend}(\text{offset})$ 读取一个字节，经符号位扩展后写入 $x[rd]$ 。



sb rs2, offset(rs1) $M[x[rs1] + \text{sext}(\text{offset})] = x[rs2][7:0]$

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

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



几个典型指令

addi rd, rs1, immediate $x[rd] = x[rs1] + \text{sext}(\text{immediate})$

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

31	20 19	15 14	12 11	7 6	0
immediate[11:0]		rs1	000	rd	0010011

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

31	25 24	20 19	15 14	12 11	7 6	0
0000000		rs2	rs1	000	Rd	0110011



更高效的CPU指令执行

怎么提高CPU执行指令的效率？

最简单的策略：逐个执行

靠一个大逻辑电路一气呵成，直接完成输入指令

运算指令：读取指令（PC+=4）；从寄存器中获取数据；运算；存回目标寄存器

跳转指令：读取指令；从寄存器中获取数据；运算；根据结果修改PC

读取内存指令：读取指令（PC+4）；从内存中获取数据；存回目标寄存器

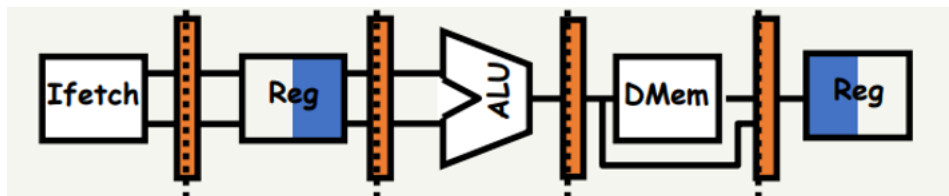
写入内存指令：读取指令（PC+4）；从寄存器中获取数据；存到目标内存地址中

效率低

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。

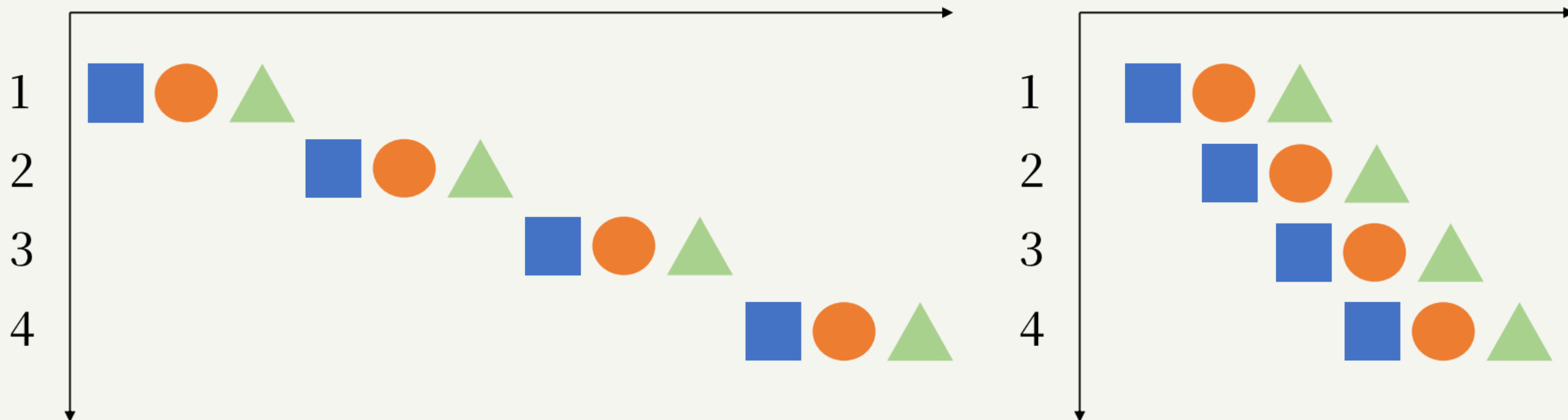


如果每个元件都有2ns的时延，那么执行一条指令的时延就会长达10ns

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

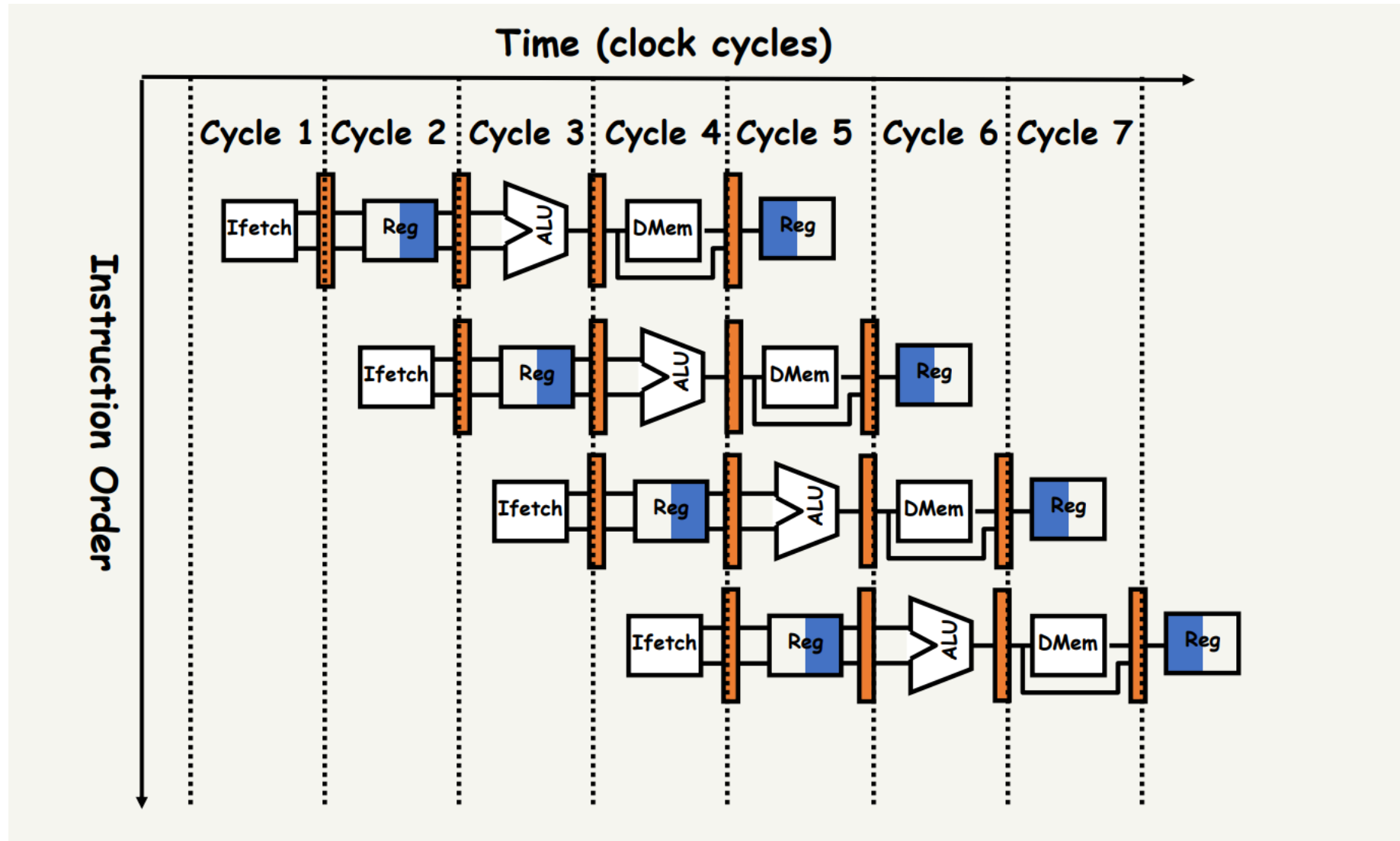
当一条指令执行到后面阶段时，负责前面阶段的元件都会空闲（因为本条指令还没执行完，不能执行下一条指令）

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）

结构冒险：一个元件不能同时处理两条指令的请求。

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。

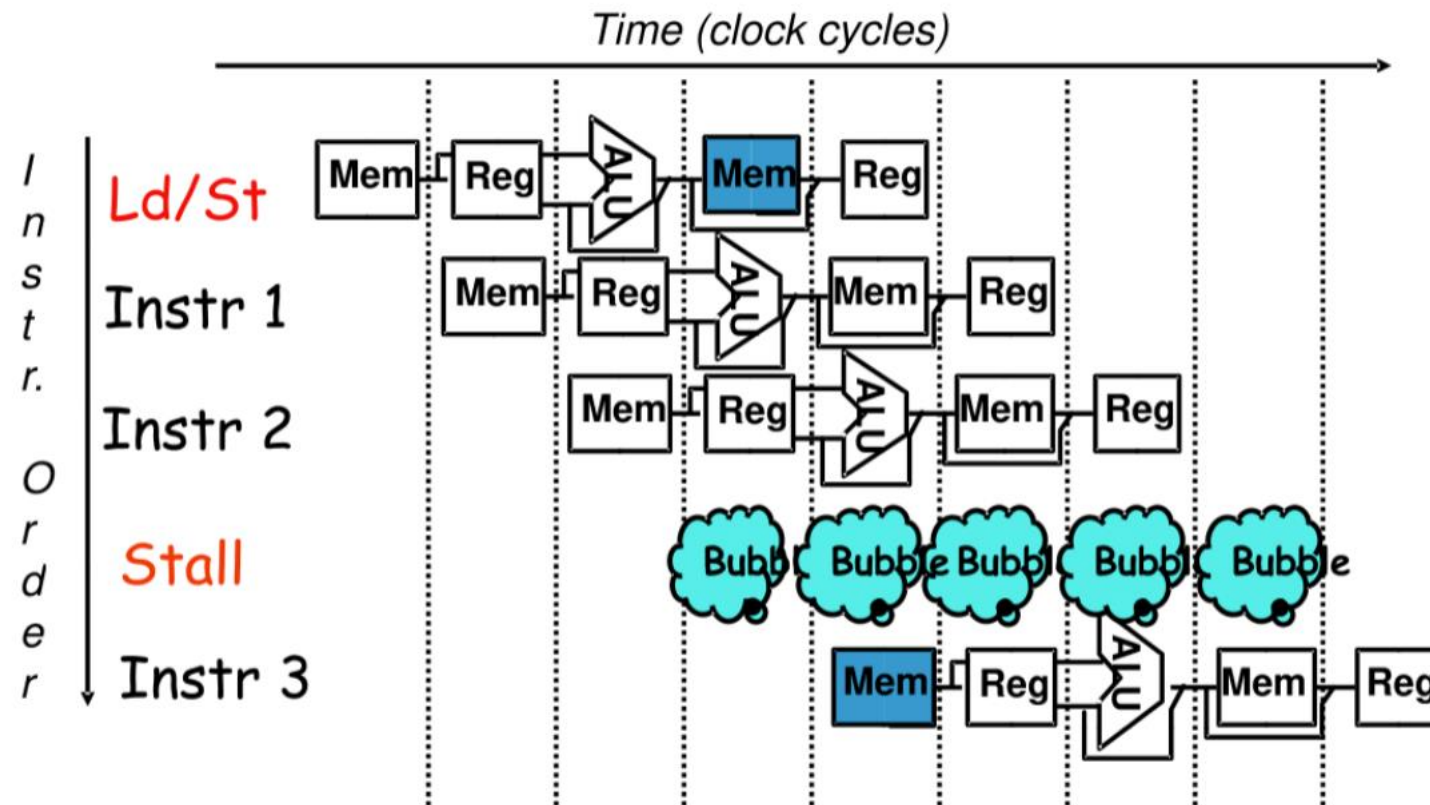
RAW：写后读，前面的指令要修改某寄存器的值，后面的指令要用该寄存器的值

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

控制冒险：由跳转指令引发的各种问题。

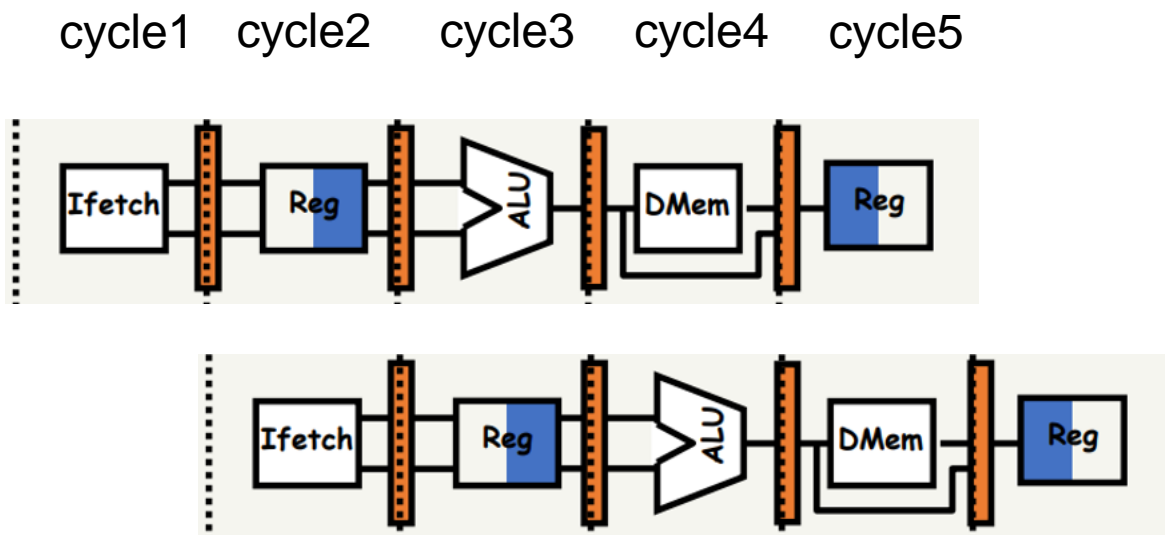
Structural Hazard

Insert Stall—simplest way



RAW

假设初始值: $R1=1$; $R2=2$; $R3=3$



`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

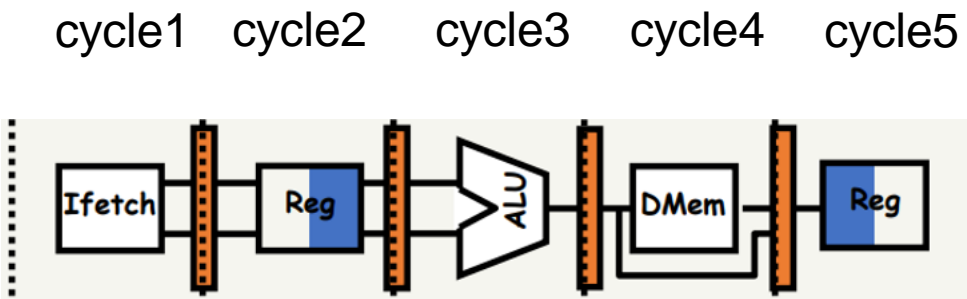
R1的新值要在第5个周期才能被写到寄存器中

但第二条指令在第3个周期就去索要R1的值

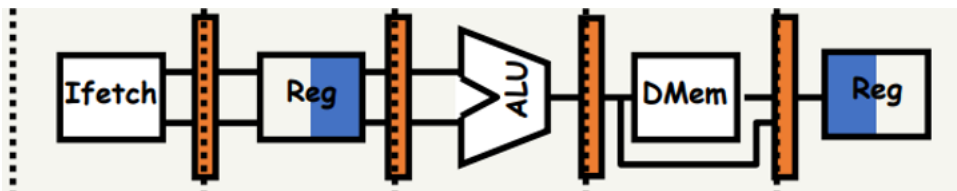
如果不管, R4的值会是4

RAW

解决方案1：暂停流水线

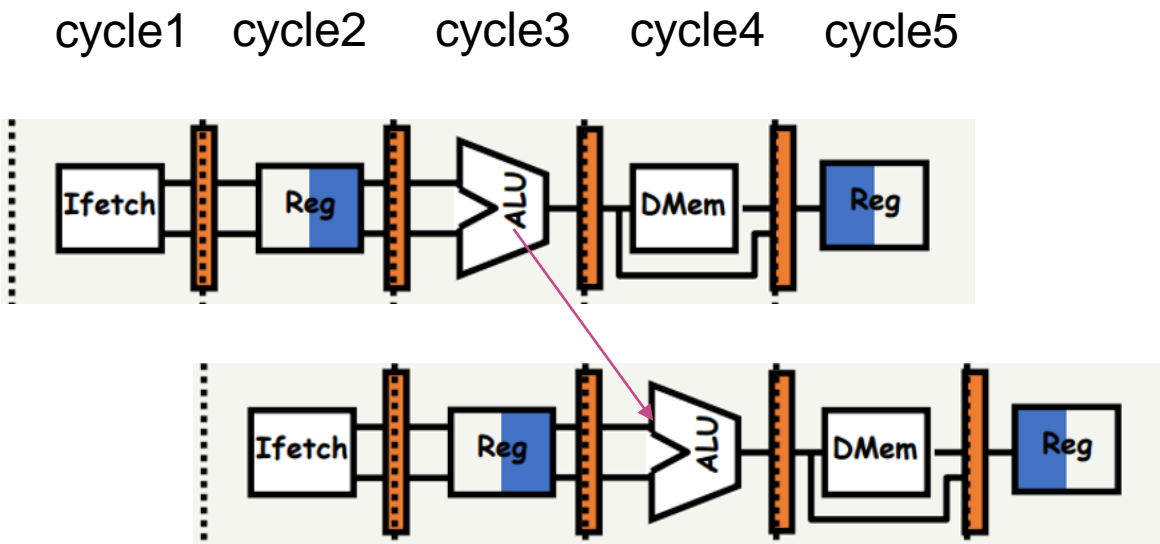


当发现新指令可能会和流水线中的指令出现RAW冒险时，流水线会暂停读入新指令，直到相关指令写入阶段完成后再继续执行新指令



RAW

解决方案2: Forward



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 ($R1=R3+R4$)

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WAW和WAR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？

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后面，第三条只能被卡着。

乱序执行：Tomasulo架构

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
(但指令发射还是顺序的)

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？（记分牌算法）

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。

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

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	f1d	f6,32(x2)	Commit	f6	Mem[32 + Regs[x2]]
2	No	f1d	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

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

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。
- Busy: 该行是否有指令

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。

RS直接连接着ALU，当一条指令变为可执行状态（ Q_j, Q_k 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。

数据广播-CDB

许多元件（RS, Reg File）都需要接收RoB指令的提交信息。因此，RoB会把结果提交到CDB（Common Data Bus, 通用数据总线中），由CDB广播给各个元件。

RS靠持续监视CDB广播的信息来维护依赖、获取新值

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。

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。

Control Hazard

条件跳转指令后面的指令无法确定。

一个简单的策略：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CDB中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

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。

二位饱和分支预测器：

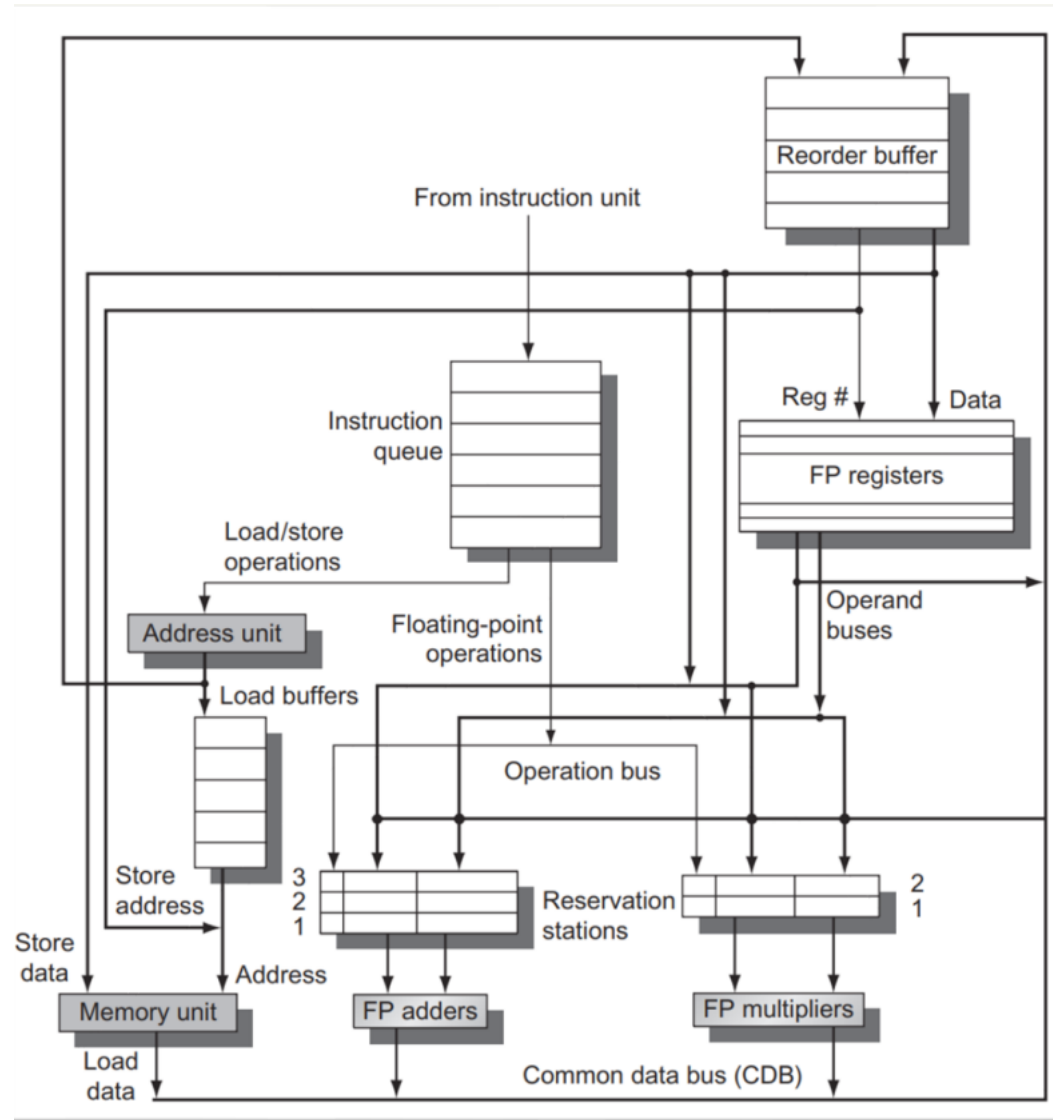
4个状态：00,01,10,11。

00,01不跳；10,11跳。

分支指令提交时：如果应该跳，计数器+1；不应该跳，计数器-1

分支预测器会监视CDB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，同时通过CDB向所有元件发出flush信号，以清空元件中的错误指令。

参考架构 (CAAQA)



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。

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，向标准输出 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

要求

尽量遵循硬件要求

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

对于换序会影响正确性的、用于元件间通讯的变量，建议存储旧值和新值，所有元件统一用旧值，每个时钟周期进行旧值的更新。

建议大家对Verilog中非阻塞赋值有一定了解。

理解时序逻辑和组合逻辑。

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

数据内存访问要花费3个时钟周期

指令内存读写只需一个周期

PC的计算和预测可以是组合逻辑

更详细地说

你应当为每个Tomasulo架构中的元件写一个类

- Rob
- RS
- 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

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

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

Disassembly of section .rom:

00000000 <.rom>:

0: 00020137
4: 040010ef
8: 0ff00513
c: 000306b7
10: 00a68223
14: ff9ff06f

lui
jal
li
lui
sb
j

sp,0x20
ra,1044 <main>
a0,255
a3,0x30
a0,4(a3) # 30004 <__heap_start+0x2e004>
c <printInt-0xff4>

address

machine code

opcode

operand(s)

评分标准

75%测试点得分+15%理解得分

如果未实现分支预测：60%测试点得分+10%理解得分

10%CR

10%Bonus

Bonus

高级分支预测

Cache

多发射

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！

比如：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，你应将其理解为0x00020137

一些提示和注意事项

仔细思考边际情况，会出现很多！

比如：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，怎么处理？

一些提示和注意事项

我们在模拟硬件

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

理论上讲，你不能动态分配内存空间

不能使用绝大多数STL容器、指针、引用

尽量别用全局变量

假装你在写Verilog

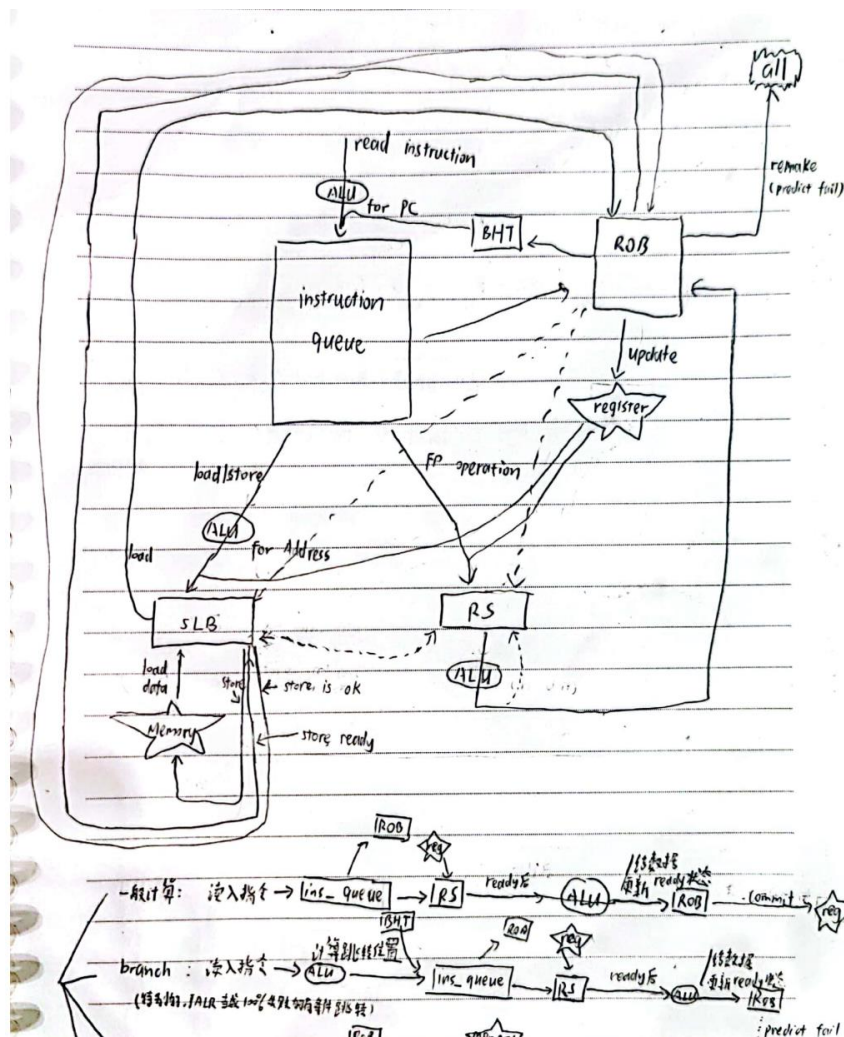
模拟器运行耗时短并不意味着模拟器写得好，所以不必在这方面卷。

你可以尝试在保证Tomasulo架构的情况下，尽可能减少运行的时钟周期数。

时钟周期数并不会影响你的得分。

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，建议画一张设计图，体现各元件的功能、接线



一些提示和注意事项

一般来说，电脑上的计算器会有“程序员”模式，可以方便地进行十六进制到二进制的转换。

Debug时，你可以写一个单级流水（基本上就是解释运行汇编语言）的CPU，比较每提交一条指令后寄存器们的状态。

顺序提交相同指令后，Tomasulo架构和单级流水的寄存器值应当是完全一致的。

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 \(zhihu.com\)](#)

<https://www.bilibili.com/video/BV1EW411u7th> 第五到第九集