

# 流水线CPU设计

---

👤 主讲人：蔡建

德以明理 学以精工



# 目录 | CONTENTS

- 1 基本原理
- 2 冒险与解决办法



# 1 基本原理



# 从单周期到流水线

在单周期CPU模型下，分析各类指令需要耗费在各个阶段的时间：

指令类型	取指令	读寄存器	ALU计算	访问存储器	写回寄存器	总时间
取字 (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
存字 (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R型 (add、sub、and...)	200 ps	100 ps	200 ps		100 ps	600 ps
分支 (j、beq)	200 ps	100 ps	200 ps			500 ps

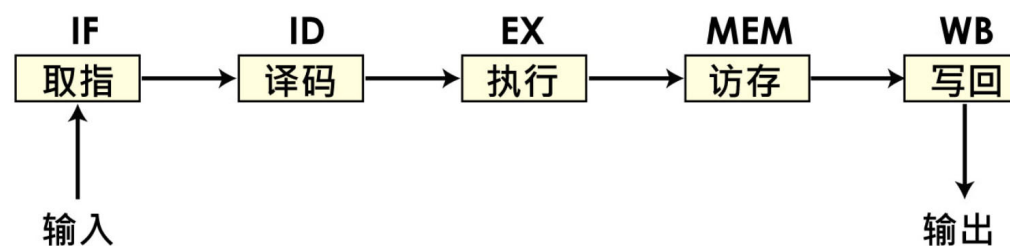
简单却性能低  
单周期

多条指令同时在  
CPU中并行流水  
流水线

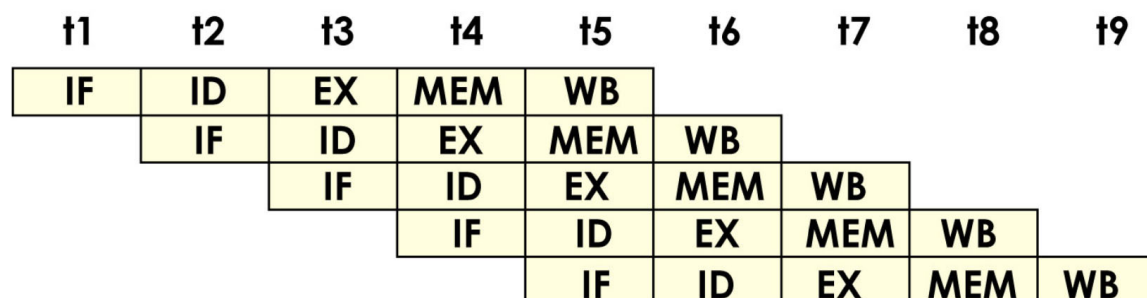


## 实现方式对比

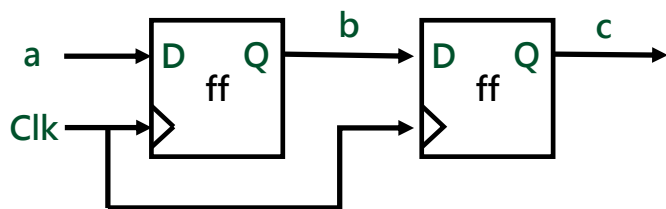
### 单周期



### 流水线

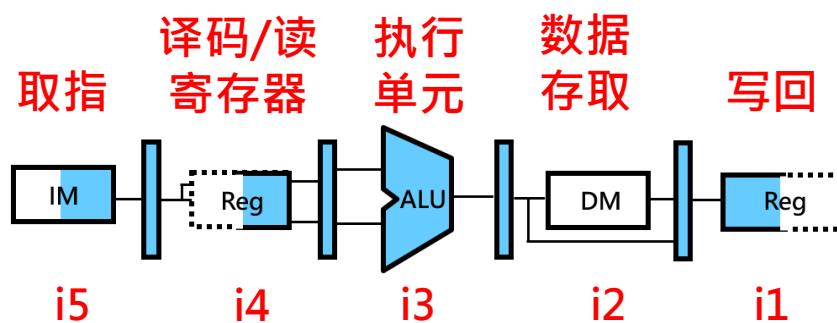
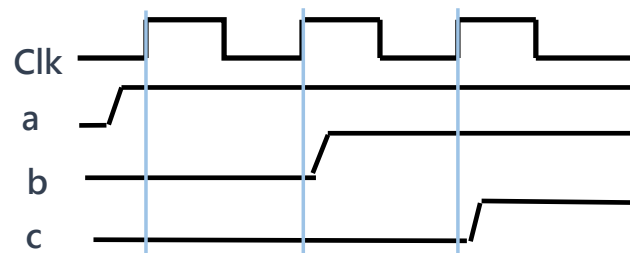


## 怎么实现流水的效果



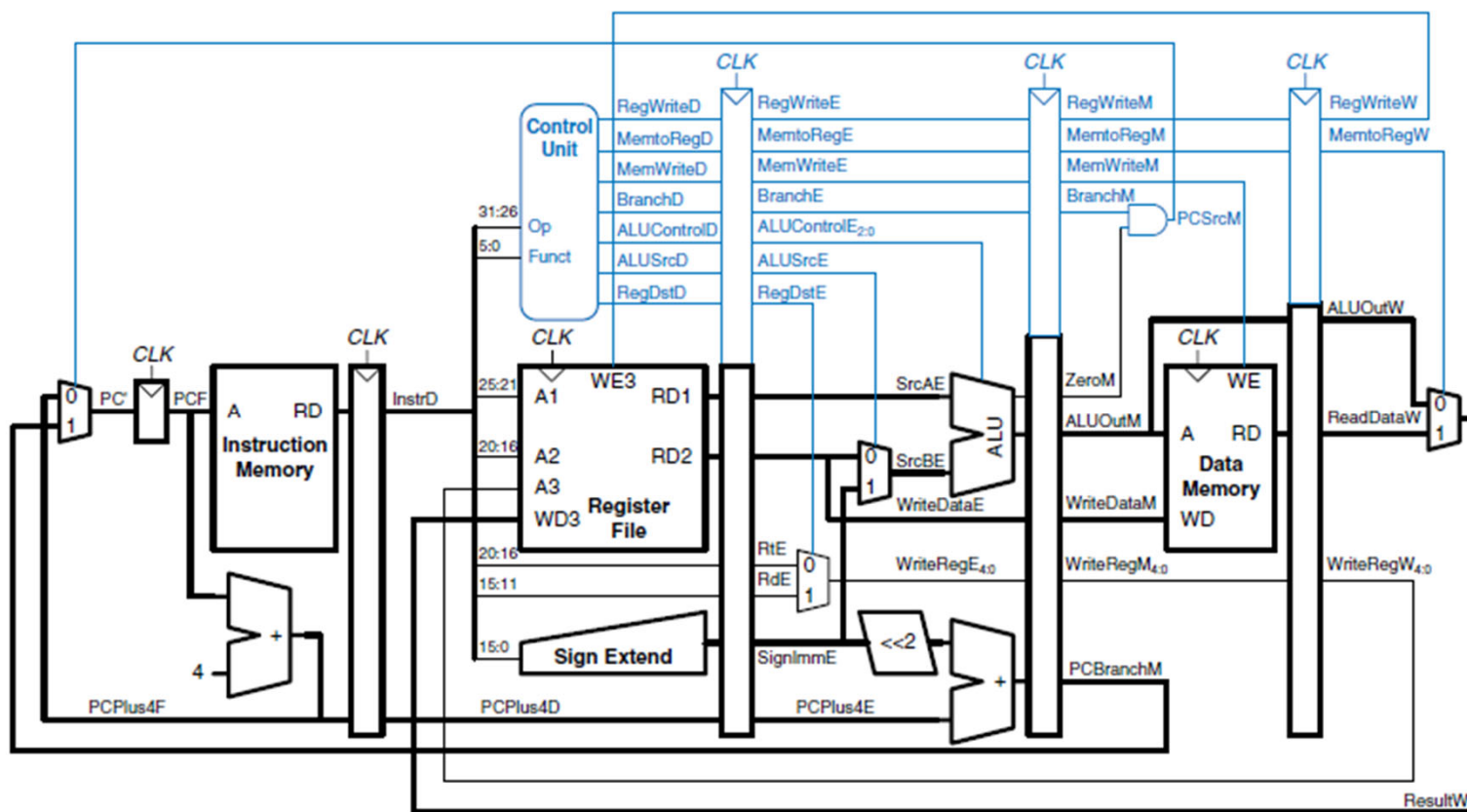
当时钟信号上升沿到来时，实现a处的数据传到b处，b处的数据传到处c。

如果每个模块内用一组并行的触发器缓存当前的指令状态信息，并且各模块之间只用组合逻辑相连。那CPU内部并行的各个指令可以在时钟信号的控制下有节奏地逐级流水。





## 五级流水线CPU模型



- 1、从存储器中读取指令
- 2、指令译码的同时读取寄存器
- 3、执行操作或计算地址
- 4、在数据存储器中读取操作数
- 5、将结果写回寄存器



# EX执行阶段缓存区定义举例

```
1 `timescale 1ns / 1ps
2
3 module EXE(
4     input clk,
5     input rst,
6     input _dmem_we,
7     input _reg_we,
8     input _wbdata_mux2_choose,
9     input [4:0] _alu_ctrl,
10
11     input [4:0] _wb_addr,
12     input [31:0] _alu_num1,
13     input [31:0] _alu_num2,
14     input [31:0] _dmem_wid,
15
16     output dmem_we_,
17     output reg_we_,
18     output wbdata_mux2_choose_,
19
20     output [4:0] wb_addr_,
21     output [31:0] alu_ans,
22     output [31:0] dmem_wid_,
23
24     output done,
25     output error //以上两条输出传给cpu计算状态模块
26 );
```

系统级控制信号:  
时钟与复位信号

从上一级流入  
的控制信号

从上一级流入  
的数据信号

流向下一级的  
控制信号

流向下一级的  
数据信号

```
27 reg dmem_we, reg_we, wbdata_mux2_choose;
28 reg [4:0] alu_ctrl;
29 reg [4:0] wb_addr;
30 reg [31:0] alu_num1;
31 reg [31:0] alu_num2;
32 reg [31:0] dmem_wid;
33
34 always @(posedge clk or negedge rst) begin
35     if(!rst) begin
36         dmem_we<=1'b0;
37         reg_we<=1'b0;
38         wbdata_mux2_choose<=1'b0;
39         alu_ctrl<=31;
40         wb_addr<=0;
41         alu_num1<=0;
42         alu_num2<=0;
43         dmem_wid<=0;
44     end
45     else begin
46         dmem_we<= _dmem_we;
47         reg_we<= _reg_we;
48         wbdata_mux2_choose<= _wbdata_mux2_choose;
49         alu_ctrl<= _alu_ctrl;
50         wb_addr<= _wb_addr;
51         alu_num1<= _alu_num1;
52         alu_num2<= _alu_num2;
53         dmem_wid<= _dmem_wid;
54     end
55 end
56
57 assign dmem_we_=dmem_we;
58 assign reg_we_=reg_we;
59 assign wbdata_mux2_choose_=wbdata_mux2_choose;
60 assign wb_addr_=wb_addr;
61 assign dmem_wid_=dmem_wid;
62
63 alu alu(
64     .ctrl(alu_ctrl), //对应当前的指令
65     .alu_num1(alu_num1),
66     .alu_num2(alu_num2),
67     .ans(alu_ans), //除了16, 17, 18, 20外其他的指令都要经过alu
68     .done(done), //程序完成
69     .error(error) //指令有误-lw/sw地址有误, 指令无法识别
70 );
```

本级缓冲区定义

缓冲区复位

缓冲区暂存上一级传入的  
数据和控制信号, 起各级  
隔断的作用。

赋值传出到下一级的信号

执行阶段ALU模块元件例化



## 流水线的性能优势

$$\text{CPU执行时间} = \text{CPU时钟周期数} \times \text{时钟周期} = \frac{\text{CPU时钟周期数}}{\text{时钟频率}} = \frac{\text{指令数} \times \text{CPI}}{\text{时钟频率}}$$

指令数与程序的结构以及汇编技术有关

时钟频率与两级触发器之间的组合逻辑的实现方式有关

CPI则是跟CPU采用的架构有直接关系  $\text{CPI} = \frac{\text{CPU时钟周期数}}{\text{指令数}}$ , CPI是个平均数

单周期CPU的CPI = 1, 流水线CPU的CPI = ?

t1	t2	t3	t4	t5	t6	t7	t8	t9
IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB

假设流水线级数为5

一条指令流过流水线时:  $\text{CPI} = 5/1 = 5$

两条指令流过流水线时:  $\text{CPI} = 6/2 = 3$

四条指令流过流水线时:  $\text{CPI} = 8/4 = 2$

十条指令流过流水线时:  $\text{CPI} = 14/10 = 1.4$

N条指令流过流水线时:  $\text{CPI} = (n+4)/n \approx 1$  (n较大)

单周期CPU的时钟周期极限为各个阶段所需时长的总和而流水线CPU的时钟周期极限只需要为耗时最长阶段的长度即可

如果流水线各级分配均匀, 理想情况下单个指令执行时流水线相对于单周期CPU所获得的加速比 =  $\frac{\text{单周期的指令周期}}{\text{流水线阶段的时间}} \approx \text{流水线级数}$

## 2 冒险与解决办法





## 结构相关

多条指令进入流水线后在同一机器周期内争用同一个功能部件所发生的冲突。

硬件不支持处于各个不同阶段的指令对其同时进行访问：

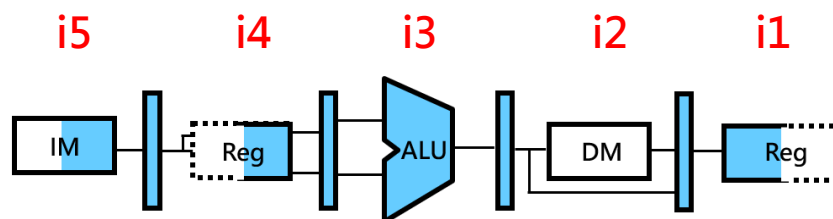
冯·诺伊曼结构让取指阶段和访存阶段对内存有一个竞争的冲突。

硬件自身功能的局限：

寄存器堆只有一个读端口时也无法在译码阶段同时读出两个寄存器数。

解决办法：

- 1、增加硬件数量 - 比如采用哈佛的结构就可以避免各阶段对内存进行访问的冲突。
- 2、改变硬件本身的功能 - 比如可以增加读的独立端口，设计两读一写的寄存器堆。
- 3、阻塞等待 - 发生冲突时阻塞某一阶段的指令，让其等待。



流水线的机制可以让多条指令同时处于CPU的各个阶段，导致后面的指令进入流水线后前面还有未执行完的指令（还未写回寄存器），所以后面的指令能看到的寄存器堆并非最新的，造成读取错误。

## RAW

Read After Write，在一条指令的写操作后跟着另一条指令的读操作，并且读写的是同一寄存器。

## WAR

Write After Read，在一条指令的读操作后跟着另一条指令的写操作，并且读写的是同一寄存器。

## WAW

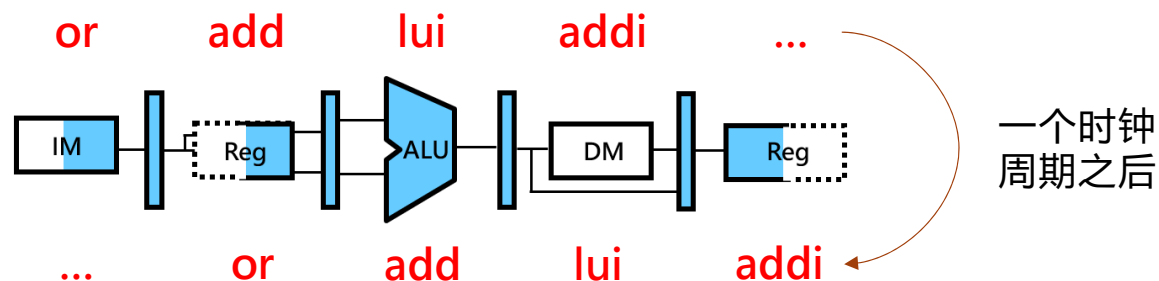
Write After Write，在一条指令的写操作后发生了写，并且两次写的是同一寄存器。



## 解决RAW数据相关

顺序执行的CPU只会产生RAW数据相关

```
...  
addi $t0,$0,100    写t0  
lui $t1,100        写t1  
add $t2,$t1,$t0    读t0、t1,写t2  
or $t3,$0,$t0      读t0,写t3  
...
```



add指令在读t0、t1寄存器的时候，写t0寄存器的指令addi和写寄存器t1的指令lui还未到达写回阶段。

不采取任何措施的话，add指令将读到的寄存器值并非我们期待的。

or指令在读t0寄存器的时候，写寄存器t0的指令addi刚到达写回阶段。

t0寄存器的值需要在下一个周期才会更新，所以此时or指令也读不到正确的值。

可见，对同一寄存器当 read after write 时至少要相隔三条指令才不会发生RAW数据相关。

在写汇编的时候，想这样的数据相关问题是会频繁出现的，一旦出现，程序的运行逻辑将出错。

# 解决RAW数据相关

从机器层面解决——方法一：

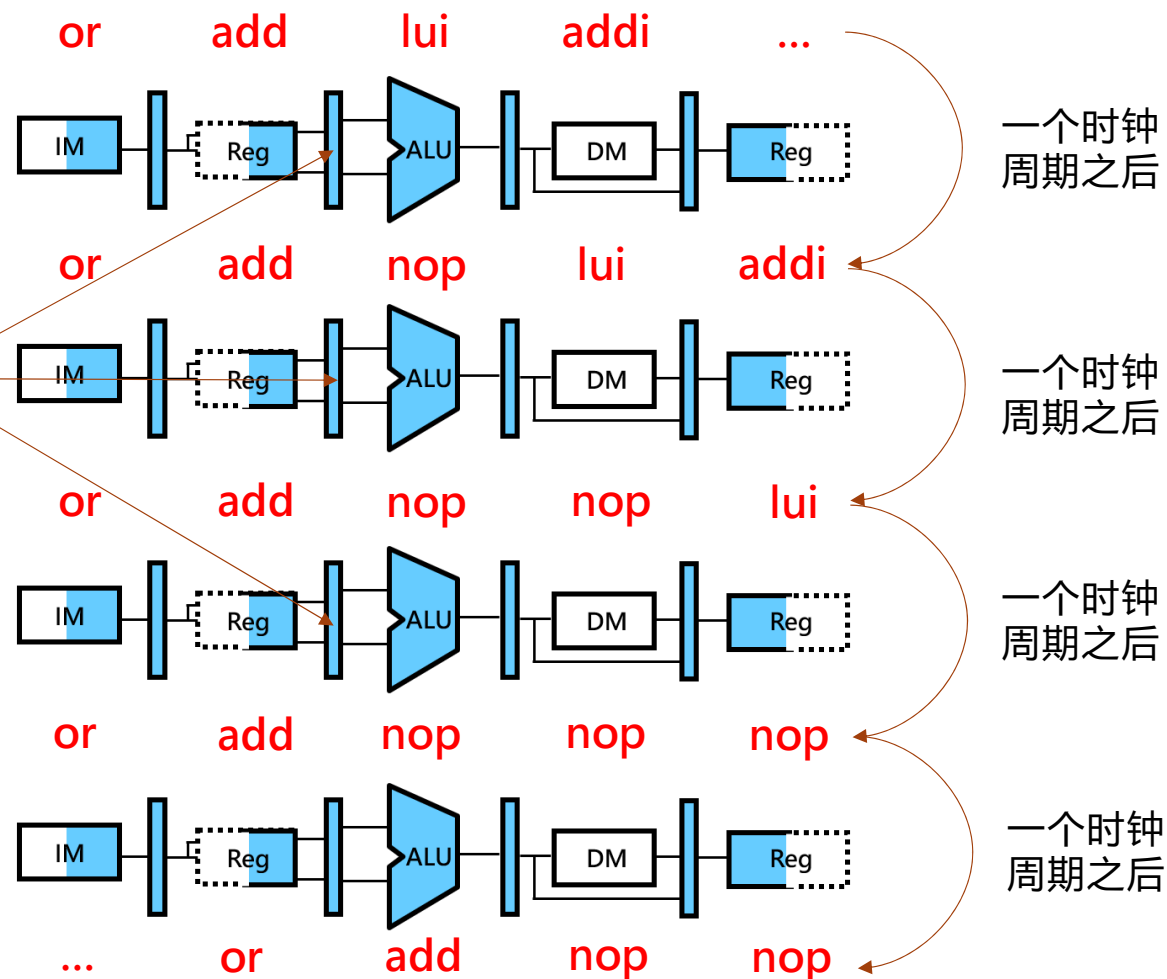
阻塞暂停，插入空的周期

检测到数据相关，流水线从此处暂停，并向后插入空的周期。

等价执行的指令

```
...
addi $t0,$0,100
lui $t1,100
nop
nop
nop
add $t2,$t1,$t0
or $t3,$0,$t0
...
```

保证有RAW的  
两条指令之间至  
少相隔三个周期





# 一种译码阶段的暂停控制模块实现

```
1  `timescale 1ns / 1ps
2
3  module pause_period_insert(
4      input clk,
5      input rst,
6
7      input [4:0] wbaddr,
8      input [4:0] inst,
9      input [4:0] rs_addr,
10     input [4:0] rt_addr, //一条指令要读的寄存器只会是rs或者rt
11     //12/18/14-读rs,0/13/20/19/30/31-都不读, 9/10/11-读rt, 1/2/3/4/5/6/7/8/15/16/17rt、rs都要读
12
13     output [4:0] inst_for_csg,
14     output pause //1-表示下一拍发送一个暂停周期, pc等于自己不变, ir也等于自己不变
15 );
16     reg [4:0] buff1; //存当前指令前一条指令的wbaddr
17     reg [4:0] buff2; //前两条
18     reg [4:0] buff3; //前三条
19     always @(posedge clk or negedge rst) begin
20         if(!rst) begin
21             buff1<=0; //零号寄存器不能写所以没有任何指令会写零号寄存器, 故不会有由于延迟读零号寄存器读错的情况
22             buff2<=0;
23             buff3<=0;
24         end
25         else begin
26             buff3<=buff2;
27             buff2<=buff1;
28             buff1<=wbaddr;
29         end
30     end
31     assign pause=
32     (inst==1||inst==2||inst==3||inst==4||inst==5||inst==6||inst==7||inst==8||inst==15||inst==16||inst==17)?
33     (((buff1==rt_addr||buff2==rt_addr||buff3==rt_addr)&&(rt_addr!=0))? 1:
34     ((buff1==rs_addr||buff2==rs_addr||buff3==rs_addr)&&(rs_addr!=0))? 1:0) :
35     (inst==9||inst==10||inst==11)?
36     (((buff1==rt_addr||buff2==rt_addr||buff3==rt_addr)&&(rt_addr!=0))? 1:0) :
37     (inst==12||inst==18||inst==14)?
38     (((buff1==rs_addr||buff2==rs_addr||buff3==rs_addr)&&(rs_addr!=0))? 1:0) : 0;
39     assign inst_for_csg= pause? 5'b11111: inst;
40
41 endmodule
```

时钟信号和复位信号

当前指令要读和要写的寄存器号输入

输出原指令码/空指令码, 以及暂停控制信号的输出

定义三个相连的缓冲区, 分别流水存储上条/上上条/上上上条指令的写回地址

根据比对缓冲内的写地址和当前指令要读的寄存器地址输入检测数据相关的发生, 若发生则阻塞译码和取指阶段, 同时向下传递空指令。

## 解决RAW数据相关

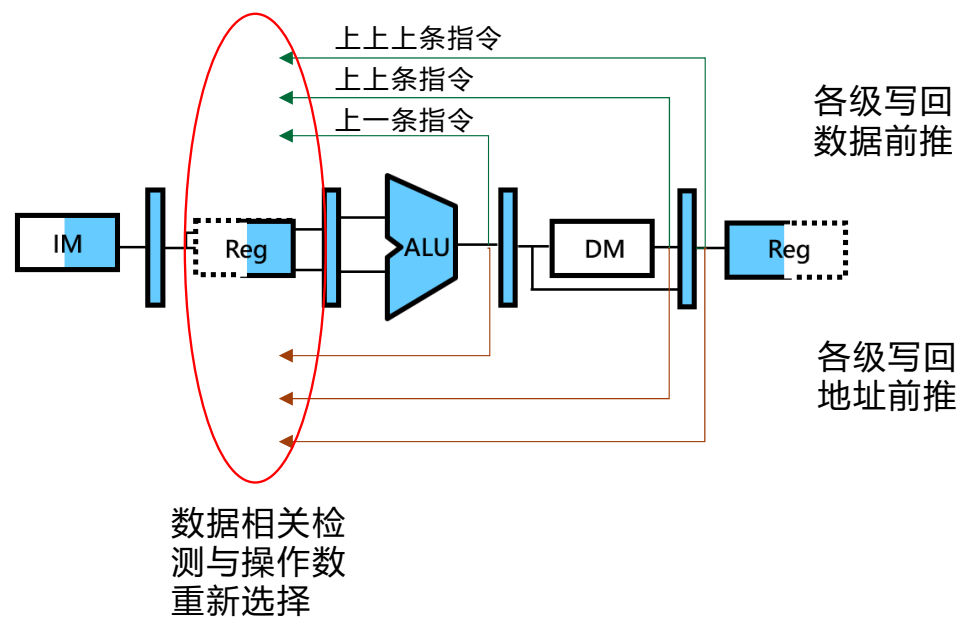
从机器层面解决——方法二：

设计旁路，进行数据前推

当前的读地址与后面处于各级指令的写回地址进行比较，如果相同，则检测到数据相关。

如果检查到数据相关，操作数将选择为后面各级对应前推的写回数据，而放弃从寄存器中读出的数据。

不用插入空周期，但需要额外的复杂逻辑。





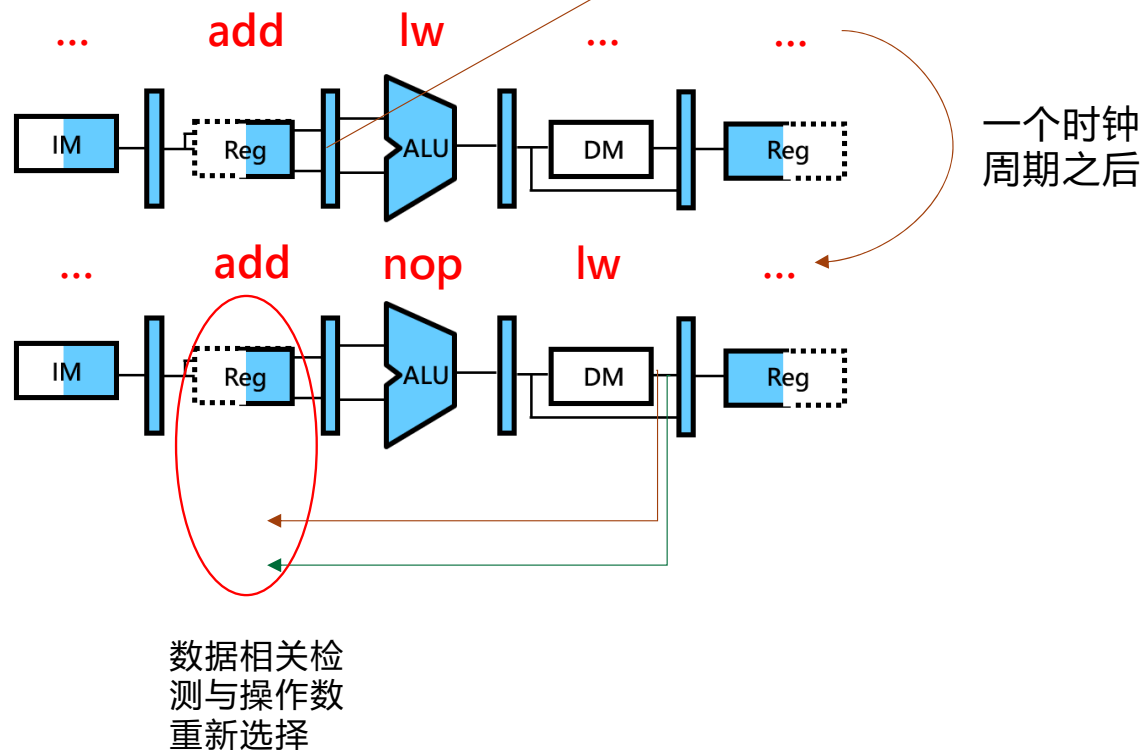
# Load延迟

...  
lw \$t0,0(base)      读数据存储器写t0  
add \$t1,\$t0,\$0      读t0, 发生数据相关  
...

阻塞暂停 + 数据前推

Load类指令要从数据存储器中读出数据才能得到写回的数据，在ALU执行阶段是无法得到写回数据的，故需要等到指令到达了访存阶段才能做数据前推。

检测到load类指令的数据相关，流水线从此处暂停，并向后插入一个空的周期，暂停等待一拍。





## 解决RAW数据相关

从汇编层面解决——方法三：

将问题甩给汇编程序员，要求在写汇编程序时避免数据相关的产生，非要产生就加空指令。

```
...  
addi $t0,$0,100  
lui $t1,100  
add $t2,$t1,$t0  
or $t3,$0,$t0  
...
```



```
...  
addi $t0,$0,100  
lui $t1,100  
nop  
nop  
nop  
add $t2,$t1,$t0  
or $t3,$0,$t0  
...
```

从编译层面解决——方法四：

将问题甩给编译器，要求在汇编转机器码时能检测到数据相关，并通过改变一些无关指令的执行顺序来避免数据相关的产生。

```
...  
addi $t0,$0,100  
lui $t1,100  
add $t2,$t1,$t0  
or $t3,$0,$t0  
[ addi $t4,$0,100  
  addi $t5,$0,100  
  addi $t6,$0,100  
...  
]
```



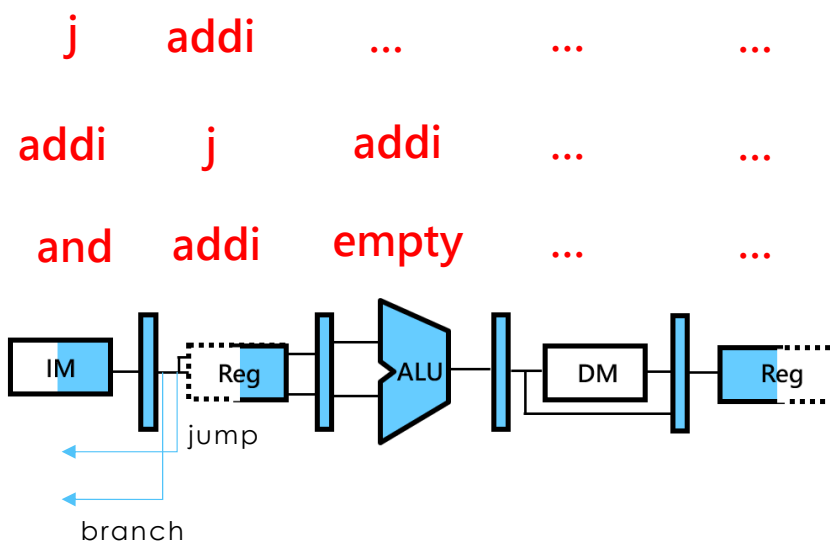
```
...  
addi $t0,$0,100  
lui $t1,100  
addi $t4,$0,100  
addi $t5,$0,100  
addi $t6,$0,100  
add $t2,$t1,$t0  
or $t3,$0,$t0  
...
```

由于流水线的阻隔，导致跳转指令的跳转控制信号不能及时到达PC指导跳转，有一个时钟周期的延时。

```

...
0x1fc00000  addi $t0,$0,100
0x1fc00004  j target
0x1fc00008  addi $t1,$0,100
...
{pc[31:28],target<<2}  and $t2,$t1,$t0
    
```

如果不采取任何措施，j指令控制跳转时跳转时从pc = 0x1fc00008开始，而不是j指令所在的0x1fc00004且后面的指令addi也被读到CPU中执行了。





## 分支预测解决控制相关

取指阶段取出跳转指令后根据情况进行分支跳或者不跳的预测，立即产生控制信号控制跳转。

静态预测 { 总是预测不跳转，若预测错误则冲刷流水线，跳转到对应分支的PC值再开始。  
              { 向前跳转预测为跳，向后跳转预测为不跳。若预测错误则冲刷流水线，再跳转到正确分支的起始PC处开始执行。

动态预测：基于已经执行的指令和CPU的状态进行分支的预测。

最简单的一种就是：1-bit动态预测——根据该指令上次是否跳转来预测此次是否跳转。如果上次跳转，则预测此次也会跳转。

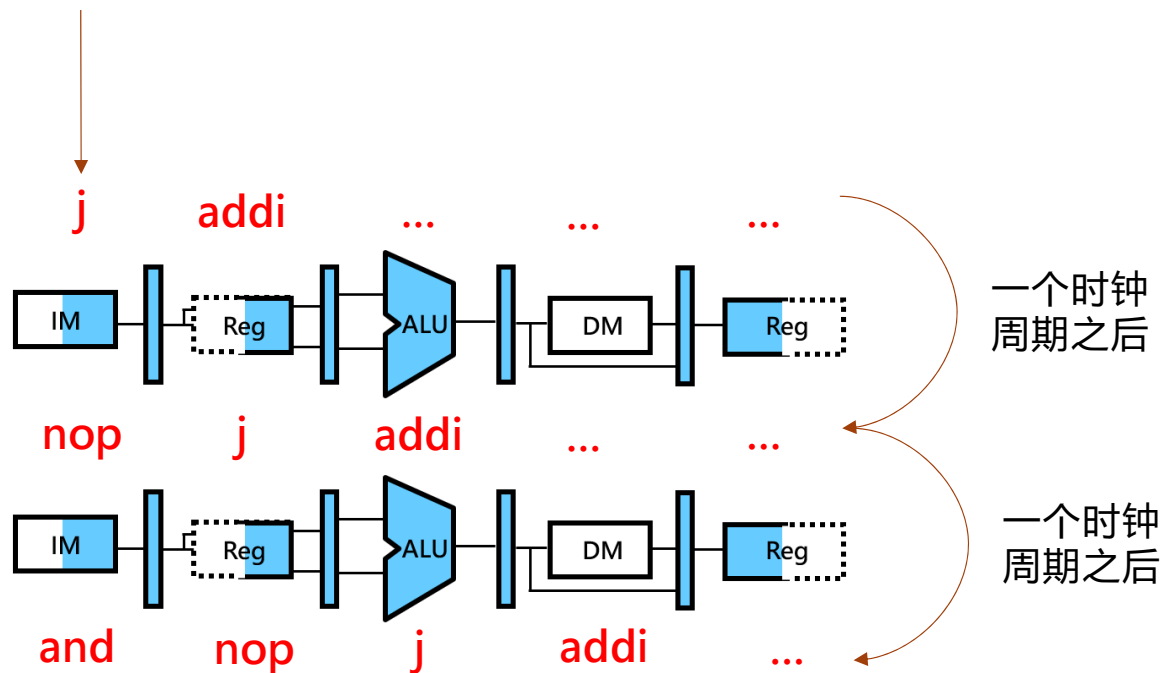
因为循环的程序结构中需要多次进行向前跳转，所以通常情况下如果上一次跳，有很大可能处于循环之中，所以，可以用一位reg记录上次跳转与否，若上次跳了则记录为1，若没跳则重置为0。

冲刷流水线：冲刷流水线要给系统程序员一种错误的指令流从未进入过CPU的感觉，所以冲刷流水线时需要将（从跳转指令以后的）各级流水线清空，也即重置各个阶段的缓冲区。

## 插入空周期解决控制相关

当取指阶段取出到的指令检测为跳转类指令时，维持PC的值，并将下一条指令设置为nop


```
...
addi $t0,$0,100
j target
addi $t1,$0,100
...
and $t2,$t1,$t0
```



## CPU支持延迟槽

CPU在设计时就说明跳转指令后面的一条指令是一定会执行的，跳转发生时也是以延迟槽内指令对应的PC为基础

汇编程序员需要了解这一设定，并在写代码时人为地改变指令的先后顺序。

<pre>... addi \$t0,\$0,100 j target addi \$t1,\$0,100 ... and \$t2,\$t1,\$t0</pre>		<pre>... j target addi \$t0,\$0,100 addi \$t1,\$0,100 ... and \$t2,\$t1,\$t0</pre>	<p>处于延迟槽内的指令在跳转发生前一定会被执行</p>
------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------	------------------------------------------------------------------------------------	------------------------------

MIPS支持延迟槽的机制，但是要注意延迟槽内指令不能为跳转指令，若为跳转指令MIPS规定后续程序的执行将变得不可预测。

# 感谢各位

👤 主讲人：蔡建

🏢 计算机学院

德以明理 学以精工

