

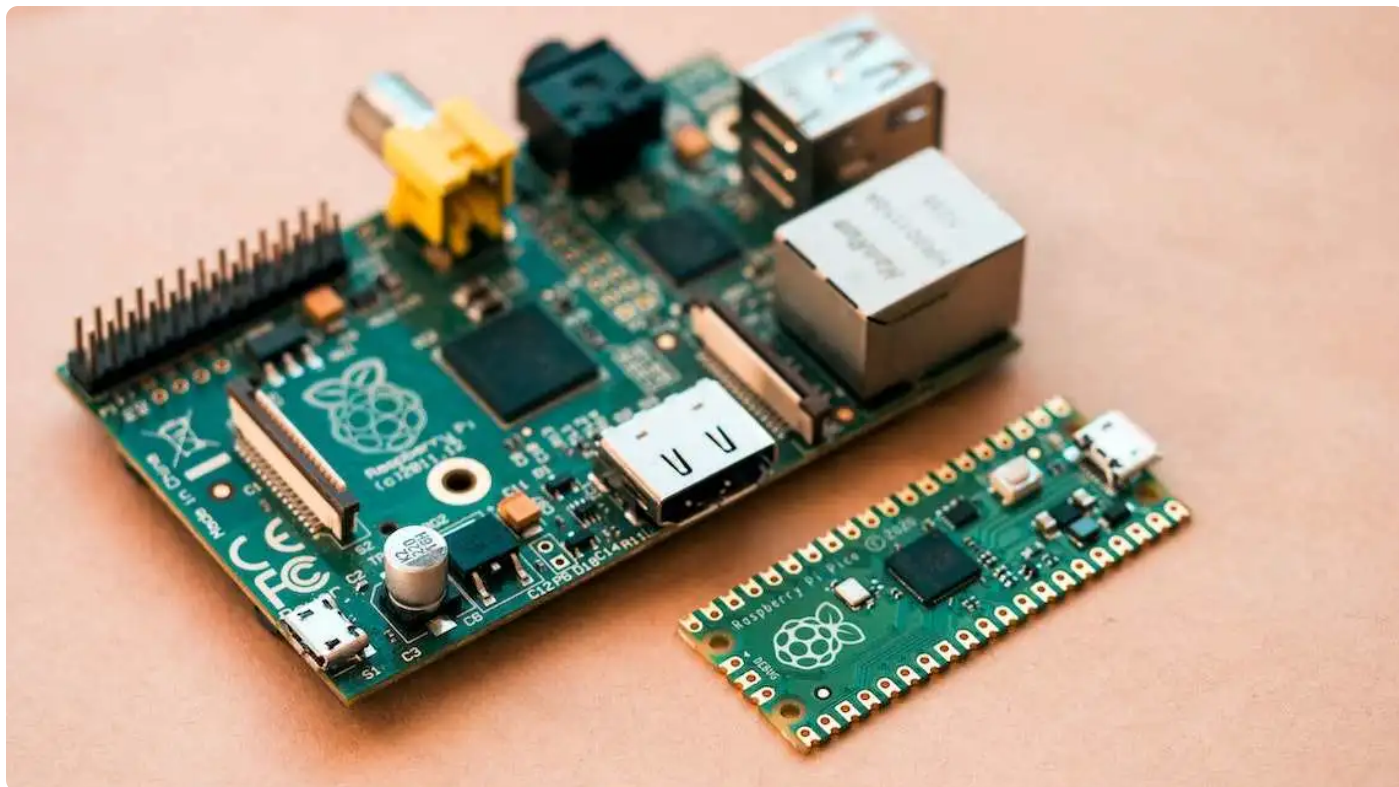
09 | 手写CPU（四）：如何实现CPU流水线的访存阶段？

2022-08-15 LMOS 来自北京

天下无鱼
<https://shikey.com/>

《计算机基础实战课》

[课程介绍 >](#)



讲述：陈晨

时长 11:14 大小 10.26M



你好，我是 LMOS。

先简单回顾一下上一节课，我们设计了 MiniCPU 流水线的执行相关模块。其中包括执行控制模块、通用寄存器模块，以及可以进行加减法运算、大小比较、移位操作的 ALU 模块。

指令执行之后就到了流水线的下一级——访存。这节课我们就重点聊聊怎么设计实现访存的相关模块。在你的设想里，访存模块必要的组成部分有哪些呢？

如果你的第一反应是访存控制模块，我只能说你只答对了一部分。访存控制模块虽然是流水线的主线，但你可能忽略了流水线中的数据相关性问题。因此，今天我们先想办法解决流水线的冒险问题，然后再完成流水线访存阶段相关模块的设计。

这节课的代码你可以从 [这里](#) 获取。

流水线数据冒险

在开始设计访存模块之前，我们得先解决一个问题，即流水线的数据冒险。

在 CPU 流水线里，执行不同的指令时会发生这样的情况：一条指令 B，它依赖于前面还在流水线中的指令 A 的执行结果。当指令 B 到达执行阶段时，因为指令 A 还在访存阶段，所以这时候就无法提供指令 B 执行所需要的数据。这就导致指令 B 无法在预期的时钟周期内执行。

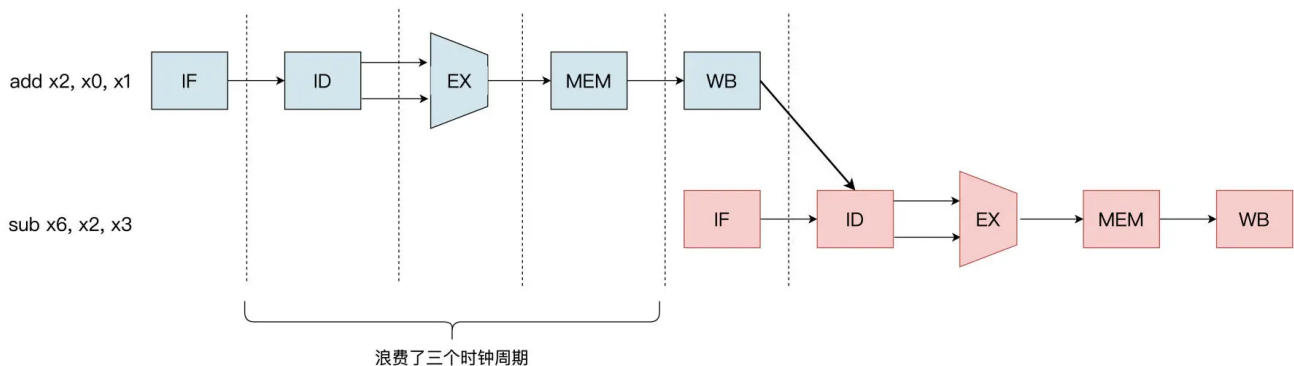
当指令在流水线中重叠执行时，后面的指令需要用到前面的指令的执行结果，而前面的指令结果尚未写回，由此导致的冲突就叫数据冒险。

我再举个更具体些的例子。比如，有一条减法指令，它需要用到前面一条加法指令的运算结果作为被减数：

 复制代码

```
1 add x2,x0,x1
2 sub x6,x2,x3
```

结合下面的示意图，我们可以看到，在不做任何干预的情况下，sub 依赖于 add 的执行结果，这导致 sub 指令要等到 add 指令走到流水线的第五个阶段，把结果写回之后才能执行，这就浪费了三个时钟周期。

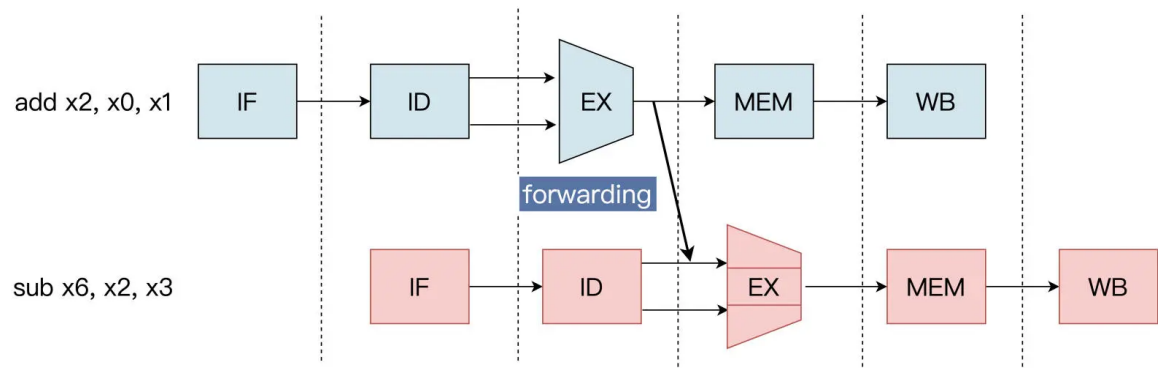


 极客时间

这种数据冒险将会严重地阻碍 CPU 的流水线，设置流水线目的就是为了提升效率，让某个时间点上有多条指令可以同时执行，这种让指令“干等”的状态显然不是最佳选项。那我们怎么解决这样的问题呢？方法其实不止一种，让我带你分析分析。

结合前的例子，我们最直接的处理办法就是通过编译器调整一些指令顺序。不过指令存在依赖关系的情况经常发生，用编译器调整的方式会导致延迟太长，处理的结果无法让我们满意。

别灰心，我们再另寻出路。把前面的加法指令 **add** 放到流水线中，通过观察整个运算过程，我们不难发现，其实 **add** 加法运算的结果，经过流水线的执行阶段（也就是第三个模块 **EX**）之后就已经出来了，只是还没把结果写回到 **x2** 寄存器而已。



所以，另一种解决办法也就有了头绪，能不能通过向内部资源添加额外的硬件，来尽快找到缺少的运算项呢？

这当然可以。对于上述的指令序列，一旦 **ALU** 计算出加法指令的结果，就可以将其作为减法指令执行的数据输入，不需要等待指令完成，就可以解决数据冒险的问题。

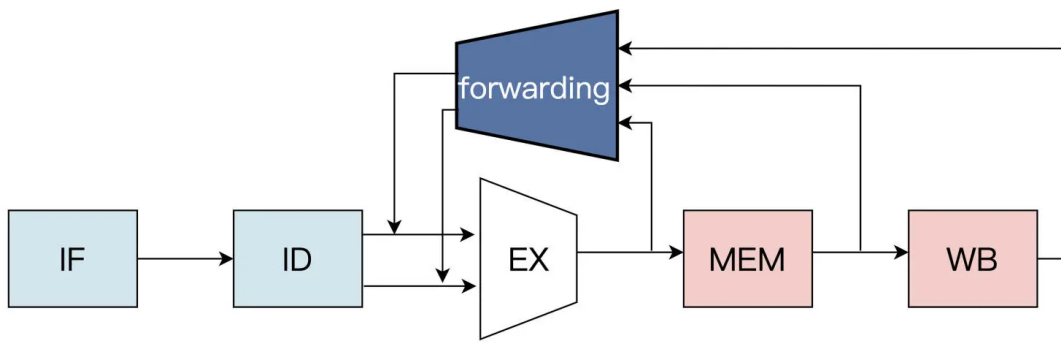
如上图所示，将 **add** 指令执行阶段运算的结果 **x2** 中的值，直接传递给 **sub** 指令作为执行阶段的输入，替换 **sub** 指令在译码阶段读出的寄存器 **x2** 的值。这种硬件上解决数据冒险的方法称为前递（**forwarding**）。

好，思路既然理清楚了，我们就把它落地到实际设计里。

数据前递模块的设计

通过上节课执行模块的设计，我们知道了 **ALU** 的逻辑运算需要两个操作数，一个来自于通用寄存器，另一个来自于通用寄存器或者指令自带的立即数。

但是当需要读取的通用寄存器的值，依赖于前面第一条或者第二条指令的运算结果时，就出现了刚才我们提到的数据冒险问题。为了解决这个问题，就需要我们专门设计一个数据前递模块（**forwarding**），它在流水线中的位置如下图所示：



所谓前递，顾名思义，就是把流水线中后面阶段产生的数据向前传递的过程。

正如上图中的 forwarding 模块，可以看到它的数据来自于流水线中的执行模块 EX、访存模块 MEM、写回模块 WB 的输出，经过 forwarding 模块处理后，把数据传递到执行模块的输入。

然后，流水线根据当前指令的译码信号，选择读取通用寄存器的数据作为执行模块的操作数，或者选择来自前递模块的数据作为执行模块的操作数。

那么具体是如何选择前递数据作为执行模块的操作数呢？让我们结合下面 forwarding 模块的代码来寻找答案：

复制代码

```

1 module forwarding (
2     input [4:0] rs1,
3     input [4:0] rs2,
4     input [4:0] exMemRd,
5     input      exMemRw,
6     input [4:0] memWBRd,
7     input      memWBRw,
8     input      mem_wb_ctrl_data_toReg,
9     input [31:0] mem_wb_readData,
10    input [31:0] mem_wb_data_result,
11    input [31:0] id_ex_data_regRData1,
12    input [31:0] id_ex_data_regRData2,
13    input [31:0] ex_mem_data_result,
14
15    output [31:0] forward_rs1_data,
16    output [31:0] forward_rs2_data
17 );
18 //检查是否发生数据冒险
19 wire [1:0] forward_rs1_sel = (exMemRw & (rs1 == exMemRd) & (exMemRd != 5'b0))
20                             : (memWBRw & (rs1 == memWBRd) & (memWBRd != 5'b0))
21                             : 2'b00;
```

```

22
23 wire [1:0] forward_rs2_sel = (exMemRw & (rs2 == exMemRd) & (exMemRd != 5'b0))
24                               : (memWBRw & (rs2 == memWBRd) & (memWBRd != 5'b0))
25                               : 2'b00;
26
27 wire [31:0] regWData = mem_wb_ctrl_data_toReg ? mem_wb_readData : mem_wb_data
28
29 //根据数据冒险的类型选择前递的数据
30 assign forward_rs1_data = (forward_rs1_sel == 2'b00) ? id_ex_data_regRData1 :
31                               (forward_rs1_sel == 2'b01) ? ex_mem_data_result :
32                               (forward_rs1_sel == 2'b10) ? regWData : 32'h0;
33
34 assign forward_rs2_data = (forward_rs2_sel == 2'b00) ? id_ex_data_regRData2 :
35                               (forward_rs2_sel == 2'b01) ? ex_mem_data_result :
36                               (forward_rs2_sel == 2'b10) ? regWData : 32'h0;
37 endmodule

```

我们分别看看代码中的各种信号。前递模块输入的端口信号 **rs1** 和 **rs2**，来自于指令译码后得到的两个通用寄存器索引。**exMemRd** 信号是来自访存模块的对通用寄存器的访问地址。**exMemRw** 是流水线访存阶段对通用寄存器的写使能控制信号。**memWBRd** 和 **memWBRw** 分别是写回模块对通用寄存器的地址和写使能控制信号。

利用这些信号就可以判断是否发生数据冒险，我们结合下面这段代码继续分析分析：

 复制代码

```

1 //检查是否发生数据冒险
2 wire [1:0] forward_rs1_sel = (exMemRw & (rs1 == exMemRd) & (exMemRd != 5'b0))
3                               : (memWBRw & (rs1 == memWBRd) & (memWBRd != 5'b0))
4                               : 2'b00;
5
6 wire [1:0] forward_rs2_sel = (exMemRw & (rs2 == exMemRd) & (exMemRd != 5'b0))
7                               : (memWBRw & (rs2 == memWBRd) & (memWBRd != 5'b0))
8                               : 2'b00;

```

当需要读取的通用寄存器的地址等于访存，或者写回阶段要访问通用寄存器地址时（也就是 **rs1 == exMemRd** 和 **rs1 == memWBRd**），就判断为将要发生数据冒险。

当然，由于通用寄存器中的零寄存器的值永远为“0”，所以不会发生数据冒险，需要排除掉这种特殊情况（也就是 **exMemRd != 5'b0** 和 **memWBRd != 5'b0**）。根据这样的判断结果，就会产生前递数据的两个选择信号 **forward_rs1_sel** 和 **forward_rs2_sel**。

发生数据冒险的情况就是这样，那不发生数据冒险又是什么情况呢？下面是选择前递的数据对应的代码段，我们结合这段代码继续分析。

 复制代码

```
1 //根据数据冒险的类型选择前递的数据
2 assign forward_rs1_data = (forward_rs1_sel == 2'b00) ? id_ex_data_regRData1 :
3                           (forward_rs1_sel == 2'b01) ? ex_mem_data_result :
4                           (forward_rs1_sel == 2'b10) ? regWData : 32'h0;
5
6 assign forward_rs2_data = (forward_rs2_sel == 2'b00) ? id_ex_data_regRData2 :
7                           (forward_rs2_sel == 2'b01) ? ex_mem_data_result :
8                           (forward_rs2_sel == 2'b10) ? regWData : 32'h0;
```

我们先把目光聚焦到 `id_ex_data_regRData1` 和 `id_ex_data_regRData2` 这两个信号上。它们来自于指令译码之后读出通用寄存器的两个操作数，这是流水线不发生数据冒险时，流水线正常选择的数据通路。

而 `ex_mem_data_result` 信号是访存阶段需要写到通用寄存器的数据，`regWData` 是回写阶段需要更新到通用寄存器的数据。这样，通过判断将要发生数据冒险的位置，前递模块选择性地 把处于流水线中的数据前递，就可以巧妙地解决流水线中的数据冒险问题了。

访存控制模块设计

好了，解决了流水线的数据冒险问题，让我们回到流水线设计的主线来，继续完成流水线的第四级——访存相关模块的设计。

在 [🔗 第六节课](#) 讲 CPU 流水线的时候我们提到过，流水线中一条指令的生命周期分为五个阶段。流水线的访存阶段就是指，将数据从存储器中读出或写入存储器的过程。这个阶段会出现由 `LOAD / STORE` 指令产生的内存访问。

因为访存阶段的功能就是对存储器读写，所以访存控制信号中，最重要的两个信号就是**存储器读控制信号 `memRead`** 和**写控制信号 `memWrite`**。当然，访存的控制信号通路也会受流水线冲刷等流水线管理信号的控制，具体的代码如下：

 复制代码

```
1 module ex_mem_ctrl(
2     input      clk,
3     input      reset,
```



```

4   input      in_mem_ctrl_memRead,      //memory读控制信号
5   input      in_mem_ctrl_memWrite,     //memory写控制信号
6   input [1:0] in_mem_ctrl_maskMode,    //mask模式选择
7   input      in_mem_ctrl_sext,         //符合扩展
8   input      in_wb_ctrl_toReg,         //写回寄存器的数据选择，“1”时为mem读取的数据
9   input      in_wb_ctrl_regWrite,      //寄存器写控制信号
10  input      flush,                    //流水线数据冲刷信号
11  output     out_mem_ctrl_memRead,
12  output     out_mem_ctrl_memWrite,
13  output [1:0] out_mem_ctrl_maskMode,
14  output     out_mem_ctrl_sext,
15  output     out_wb_ctrl_toReg,
16  output     out_wb_ctrl_regWrite
17 );
18
19  reg  reg_mem_ctrl_memRead;
20  reg  reg_mem_ctrl_memWrite;
21  reg [1:0] reg_mem_ctrl_maskMode;
22  reg  reg_mem_ctrl_sext;
23  reg  reg_wb_ctrl_toReg;
24  reg  reg_wb_ctrl_regWrite;
25
26  assign out_mem_ctrl_memRead = reg_mem_ctrl_memRead;
27  assign out_mem_ctrl_memWrite = reg_mem_ctrl_memWrite;
28  assign out_mem_ctrl_maskMode = reg_mem_ctrl_maskMode;
29  assign out_mem_ctrl_sext = reg_mem_ctrl_sext;
30  assign out_wb_ctrl_toReg = reg_wb_ctrl_toReg;
31  assign out_wb_ctrl_regWrite = reg_wb_ctrl_regWrite;
32
33  always @(posedge clk or posedge reset) begin
34      if (reset) begin
35          reg_mem_ctrl_memRead <= 1'h0;
36      end else if (flush) begin
37          reg_mem_ctrl_memRead <= 1'h0;
38      end else begin
39          reg_mem_ctrl_memRead <= in_mem_ctrl_memRead;
40      end
41  end
42
43  always @(posedge clk or posedge reset) begin
44      if (reset) begin
45          reg_mem_ctrl_memWrite <= 1'h0;
46      end else if (flush) begin
47          reg_mem_ctrl_memWrite <= 1'h0;
48      end else begin
49          reg_mem_ctrl_memWrite <= in_mem_ctrl_memWrite;
50      end
51  end
52
53  ..... //由于代码较长，结构相似，这里省略了一部分，完整代码你可以从Gitee上获取
54
55  endmodule

```

虽然代码有几十行，但过程还是很容易理解的。首先要根据流水线的冲刷控制信号 `flush`，判断访存阶段的控制信号是否需要清零。

如果 `flush` 等于“0”，就把上一阶段送过来的控制信号（比如存储器读控制信号 `memRead`、存储器写控制信号 `memWrite`.....等），通过寄存器保存下来，然后发送给存储器读写控制模块（`dmem_rw.v`）或者流水线的下一级使用。

访存数据通路模块设计

接下来，我们继续完成访存数据通路模块的设计。访存数据通路就是把访存阶段读取到的存储器数据，或者是指令执行产生的结果发送流水线的下一级处理。

由于下一级也就是流水线的最后一级——写回，所以访存的数据通路主要包括要写回的通用寄存器地址 `regWAddr`、访问存储器读取的数据 `regRData2`、指令运算的结果 `result` 等。

访存的数据通路也会受流水线冲刷等流水线管理信号的控制，具体代码如下：

 复制代码

```
1 module ex_mem(  
2     input        clk,  
3     input        reset,  
4     input  [4:0]  in_regWAddr,    //写回寄存器的地址  
5     input  [31:0] in_regRData2,   //读存储器的数据  
6     input  [1:0]  ex_result_sel,  //执行结果选择  
7     input  [31:0] id_ex_data_imm, //指令立即数  
8     input  [31:0] alu_result,     //ALU运算结果  
9     input  [31:0] in_pc,          //当前PC值  
10    input        flush,           //流水线数据冲刷控制信号  
11    output [4:0]  data_regWAddr,  
12    output [31:0] data_regRData2,  
13    output [31:0] data_result,  
14    output [31:0] data_pc  
15 );  
16 reg [4:0] reg_regWAddr;  
17 reg [31:0] reg_regRData2;  
18 reg [31:0] reg_result;  
19 reg [31:0] reg_pc;  
20  
21 wire [31:0] resulet_w = (ex_result_sel == 2'h0) ? alu_result :  
22                        (ex_result_sel == 2'h1) ? id_ex_data_imm :  
23                        (ex_result_sel == 2'h2) ? (in_pc +32'h4) : 32'h0;  
24 assign data_regWAddr = reg_regWAddr;  
25 assign data_regRData2 = reg_regRData2;  
26 assign data_result = reg_result;
```



```

27 assign data_pc = reg_pc;
28
29 always @(posedge clk or posedge reset) begin
30     if (reset) begin
31         reg_regWAddr <= 5'h0;
32     end else if (flush) begin
33         reg_regWAddr <= 5'h0;
34     end else begin
35         reg_regWAddr <= in_regWAddr;
36     end
37 end
38
39 always @(posedge clk or posedge reset) begin
40     if (reset) begin
41         reg_regRData2 <= 32'h0;
42     end else if (flush) begin
43         reg_regRData2 <= 32'h0;
44     end else begin
45         reg_regRData2 <= in_regRData2;
46     end
47 end
48
49 always @(posedge clk or posedge reset) begin
50     if (reset) begin
51         reg_result <= 32'h0;
52     end else if (flush) begin
53         reg_result <= 32'h0;
54     end else begin
55         reg_result <= resulet_w;
56     end
57 end
58
59 always @(posedge clk or posedge reset) begin
60     if (reset) begin
61         reg_pc <= 32'h0;
62     end else if (flush) begin
63         reg_pc <= 32'h0;
64     end else begin
65         reg_pc <= in_pc;
66     end
67 end
68 endmodule

```

和上面的访存控制模块类似，访存数据通路模块也是根据流水线的冲刷控制信号 **flush**，判断访存阶段的数据是否需要清零。如果不需要清零，就把上一阶段送过来的数据通过寄存器保存下来。

对于代码的第 21 到第 23 行代码，我为你单独解释一下。

`ex_result_sel` 就是对流水线执行阶段的结果进行选择。当 (`ex_result_sel == 2'h0`) 时，就选择 ALU 的运算结果；当 (`ex_result_sel == 2'h1`) 时，就会选择指令解码得到的立即数（其实就是对应 LUI 指令）；当 (`ex_result_sel == 2'h1`) 时，选择 PC 加 4 的值，也就是下一个 PC 的值。

 复制代码

```
1  wire [31:0] resulet_w = (ex_result_sel == 2'h0) ? alu_result :  
2                          (ex_result_sel == 2'h1) ? id_ex_data_imm :  
3                          (ex_result_sel == 2'h1) ? (in_pc + 32'h4) : 32'h0;
```

重点回顾

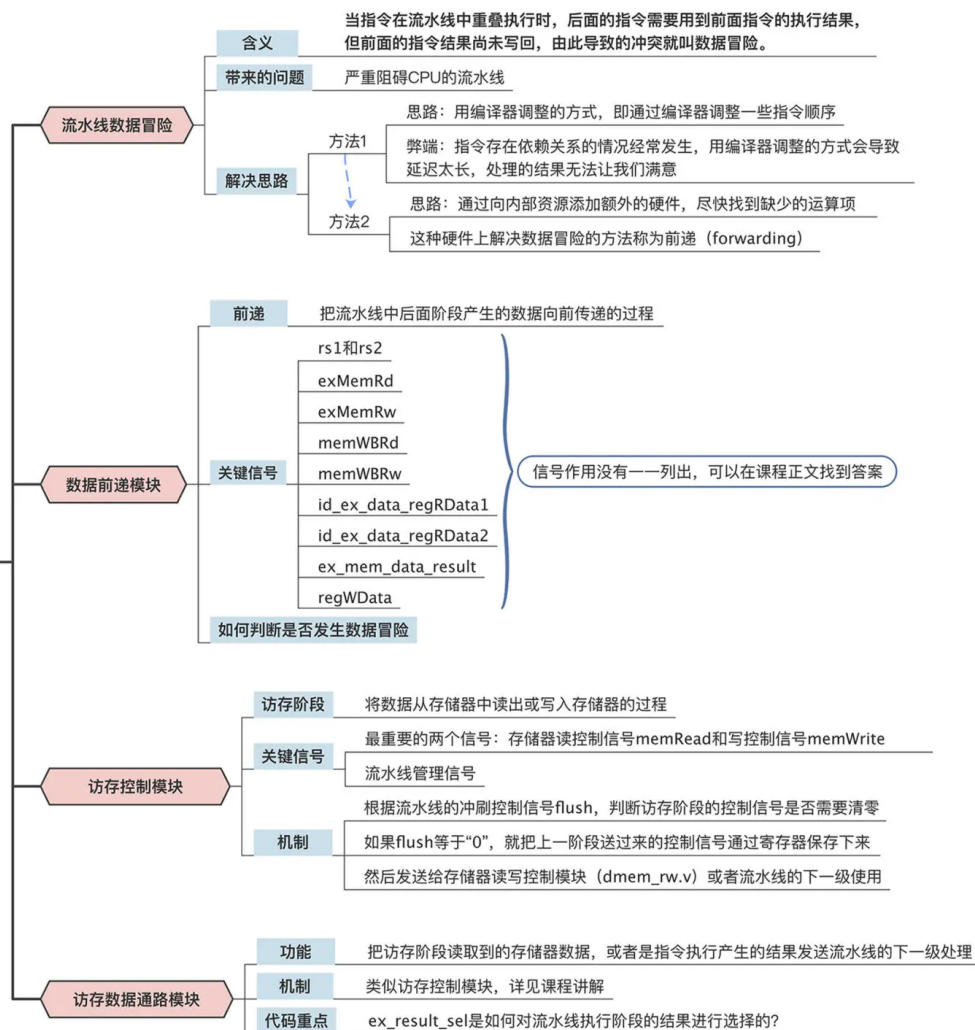
这节课的内容到这里就告一段落了，我给你做个总结吧。

今天我们在设计访存模块之前，先探讨了流水线中的数据冒险问题。在执行指令时，如果发生了数据冒险，就可能使流水线停顿，等待前面的指令执行完成后才能继续执行后续的指令，严重影响了指令在 CPU 流水线中并行执行。因此，我们设计了**数据前递模块**来解决数据冒险的问题。

但是，添加前递模块并不能避免所有的流水线停顿。比如，当一条读存储器指令（LOAD）之后紧跟一条需要使用其结果的 R 型指令时，就算使用前递也需要流水线停顿。因为读存储器的数据必须要在访存之后才能用，但 `load` 指令正在访存时，后一条指令已经在执行。所以，在这种情况下，流水线必须停顿，通常的说法是**在两条指令之间插入气泡**。

最后，我们根据流水线的控制信号，完成了访存控制信号通路和访存数据通路的模块设计。这节课的要点你可以参考下面的导图。

CPU流水线 访存阶段



通过课程的讲解，CPU 流水线中访存阶段的设计实现的思路，相信你已经心中有数了，别忘了课后结合配套代码再找找“手感”。下节课，我们将会介绍流水线的最后一级——写回模块的设计，敬请期待。

思考题

除了数据冒险，我们的 CPU 流水线是否还存在其它的冲突问题，你想到解决方法了么？

欢迎你在留言区和我交流，踊跃提问或者记录笔记对我们加深理解有很大的帮助。如果你觉得这节课还不错，别忘了分享给更多朋友，和他一起手写 CPU。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 08 | 手写CPU（三）：如何实现指令执行模块？

下一篇 10 | 手写CPU（五）：CPU流水线的写回模块如何实现？

精选留言 (4)

写留言



苏流郁宓

2022-08-15 来自湖北

在硬件层面，为提高流水线效率。按照同时钟周期原则，假如ac为一组指令，c的运算必须依赖a的结果，那可以把同样周期的bd插入ac中组为abcd指令，a指令下去运算，紧接着b指令也下去运算，这样表面上c需要a的等待时间，但等待时间途中b指令也忙着，相当于cpu核没有空闲，利用指令b指令的时间间隙，a指令完成执行，结果返回给c，c执行差不多，b指令完成执行结果就可以立马返回给d指令执行了

cpu核执行指令没有空闲时间，乱序执行，这是通过增加晶体管来完成的！

这可以和今天的流水线执行方式进行互补的啊

作者回复: 是的



1



苏流郁宓

2022-08-15 来自湖北

cpu流水线下，软硬件中断导致须停下手头工作来处理中断问题，频繁的软硬件中断会降低cpu工作效率（比如正在运行的数据需要来回存取读写），在明面上如用户一下打开多个app容易卡住，或者浏览器打开多个网页.....

解决方法，如浏览器给定同时允许打开网页的最大数，或者app中也设软件cache，减少因网络即时数据存储问题导致的中断量加大，比如，优化软件结构。能就近找到的数据不要让cpu遍历去找！

在cpu层面，尽量将宽度一致的数据/指令列为一组，减少内存不该有的缺页异常（该有的还得有）！

作者回复: 是的



👍 1



+1

2022-08-21 来自上海

不太理解在什么情况下需要在访存这个阶段从存储器中读取数据，不是能在执行阶段通过寄存器地址完成么？另外存储器和寄存器有什么区别呢？然后为何还有给pc地址加4的情况，不是在预读取的时候就完成了么

共 1 条评论 >



青玉白露

2022-08-17 来自湖北

精彩！锁、内存屏障等概念其实说到底就是基于访存阶段进行的特性化操作，还有很多的东西需要妈那美女消化。

作者回复: 哈哈

