第六章 利用流水线提高性能

作业参考答案

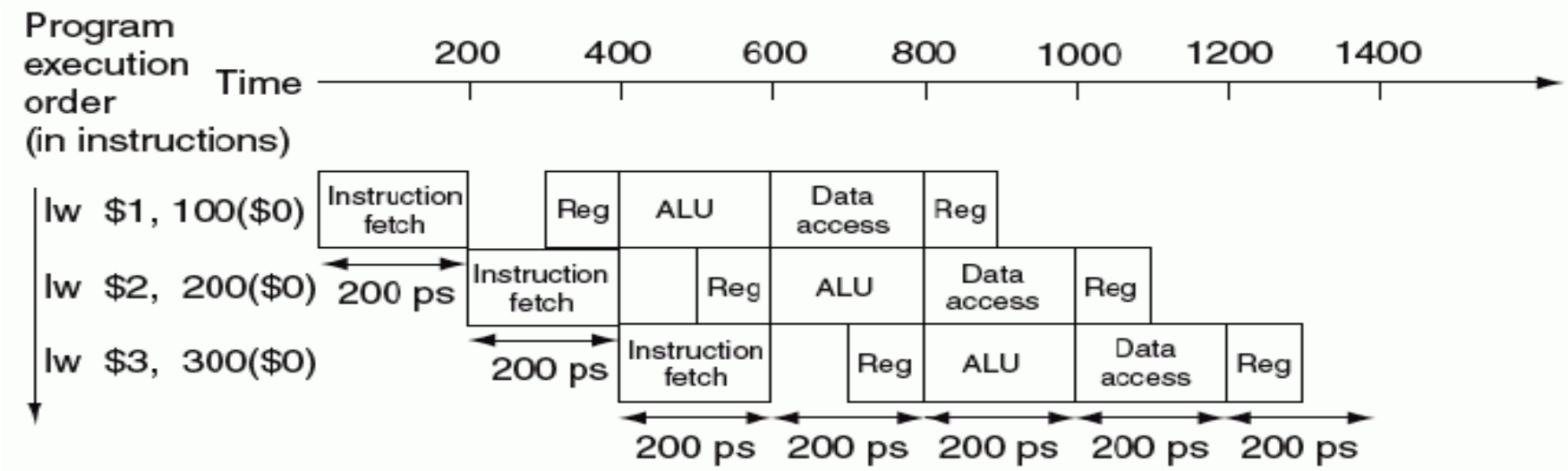
6.1 [5] <§6.1> If the time for an ALU operation can be shortened by 25% (compared to the description in Figure 6.2 on page 373);

a. Will it affect the speedup obtained from pipelining? If yes, by how much? Otherwise, why?

b. What if the ALU operation now takes 25% more time?

参考答案：

P.373中ALU操作时间为200ps。



a. ALU操作时间缩短25%不能加快流水线指令速度。

因为流水线的速度最终由时钟周期的宽度决定，而它不会缩短时钟周期。

b. 如果ALU操作时间延长25%，那么，ALU时间将变为250ps，这样，ALU操作将变成瓶颈，使得流水线的时钟周期为250ps，其效率降低（250-200)/250=20%。

6.2 [10] <§6.1> A computer architect needs to design the pipeline of a new microprocessor. She has an example workload program core with 10^6 instructions. Each instruction takes 100 ps to finish.

a. How long does it take to execute this program core on a nonpipelined processor?

b. The current state-of-the-art microprocessor has about 20 pipeline stages. Assume it is perfectly pipelined. How much speedup will it achieve compared to the nonpipelined processor?

c. Real pipelining isn't perfect, since implementing pipelining introduces some overhead per pipeline stage. Will this overhead affect instruction latency, instruction throughput, or both?

参考答案：

a. 一个程序核心模块共**10^6**条指令，每条指令花**100ps**完成，则在非流水线处理器上执行的时间为：**100 x10^6=100us.**

b. 若在一个**20**级流水线的处理器上执行，理想情况下，每个时钟周期为：**100/20=5ps**，所以，程序执行时间为**5 x10^6=5us.** 快了**100/5=20**倍

c. 流水线并不是理想的，流水线段之间数据的传递会产生额外的开销。

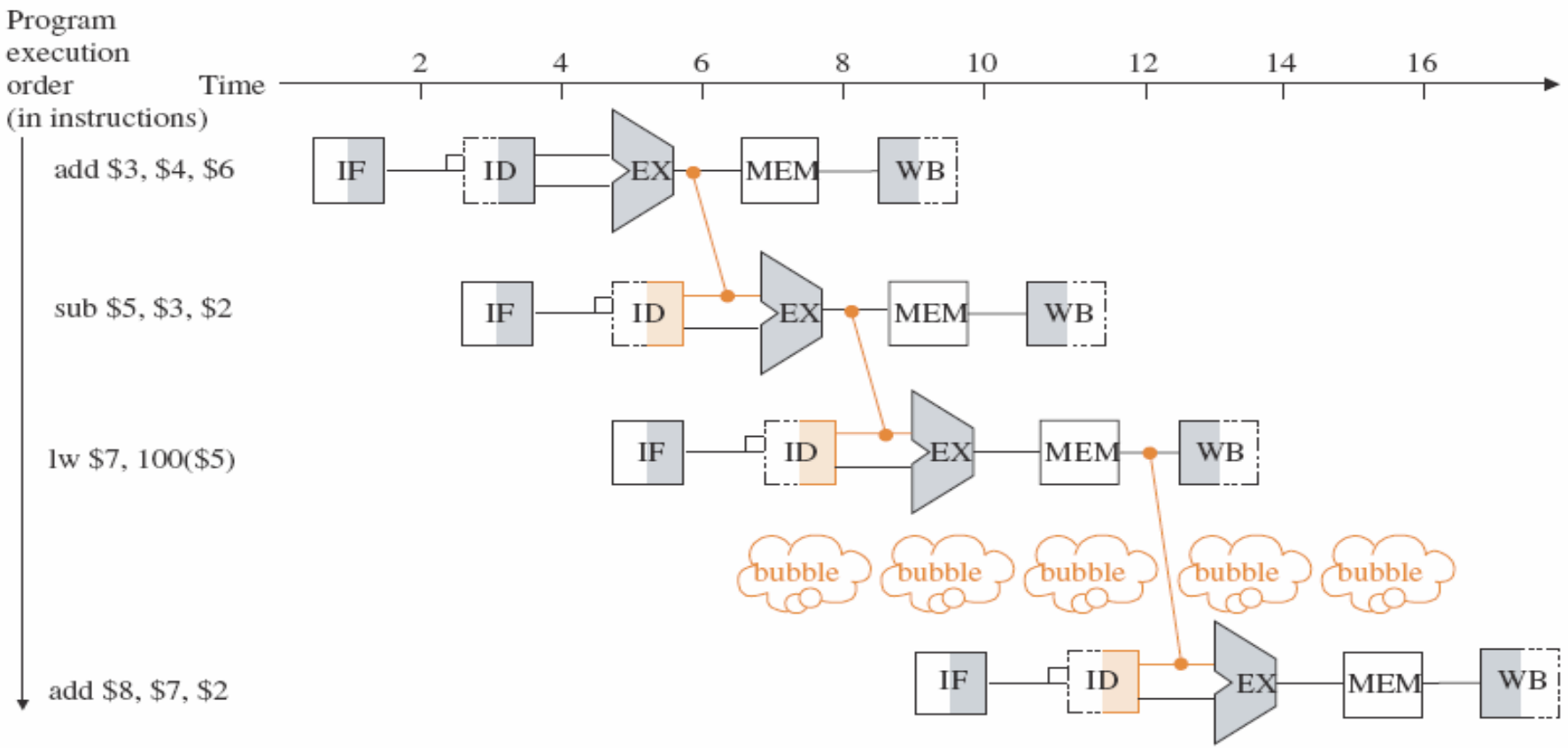
一方面，这种开销使得一条指令的执行时间被延长，即影响 **Instruction latency**

另一方面，这种开销也拉长了每个流水段的执行时间，即影响 **Instruction throughput**

6.3 [5] <§6.1> Using a drawing similar to Figure 6.5 on page 377, show the forwarding paths needed to execute the following four instructions:

```
add $3, $4, $6
sub $5, $3, $2
lw  $7, 100($5)
add $8, $7, $2
```

参考答案：有三个**RAW**数据冒险，其中一个是**load-use**数据冒险，需要“阻塞”一个时钟



6.4 [10] <§6.1> Identify all of the data dependencies in the following code. Which dependencies are data hazards that will be resolved via forwarding? Which dependencies are data hazards that will cause a stall?

```
add $3, $4, $2
sub $5, $3, $1
lw  $6, 200($3)
add $7, $3, $6
```

参考答案：有四个**RAW**数据冒险

（**1**）第一条**add**指令和第二条**sub**指令之间

（**2**）第一条**add**指令和第三条**lw**指令之间

（**3**）第一条**add**指令和第四条**add**指令之间

（**4**）第三条**lw**指令和第四条**add**指令之间

其中，（**1**）、（**2**）和（**3**）是关于寄存器**$3**的数据冒险，可以通过“转发”解决

（**4**）是关于寄存器**$6**的数据冒险，是**load-use**数据冒险，不能通过“转发”解决，将发生一次“阻塞”

6.17 [5] <§§6.4, 6.5> Consider executing the following code on the pipelined datapath of Figure 6.36 on page 416:

```
add    $2,  $3,  $1
sub    $4,  $3,  $5
add    $5,  $3,  $7
add    $7,  $6,  $1
add    $8,  $2,  $6
```

At the end of the fifth cycle of execution, which registers are being read and which register will be written?

参考答案：图6.36是一个带“冒险”检测和“转发”处理的五阶段流水线数据通路。

第五个时钟结束时，各条指令的执行情况如下：

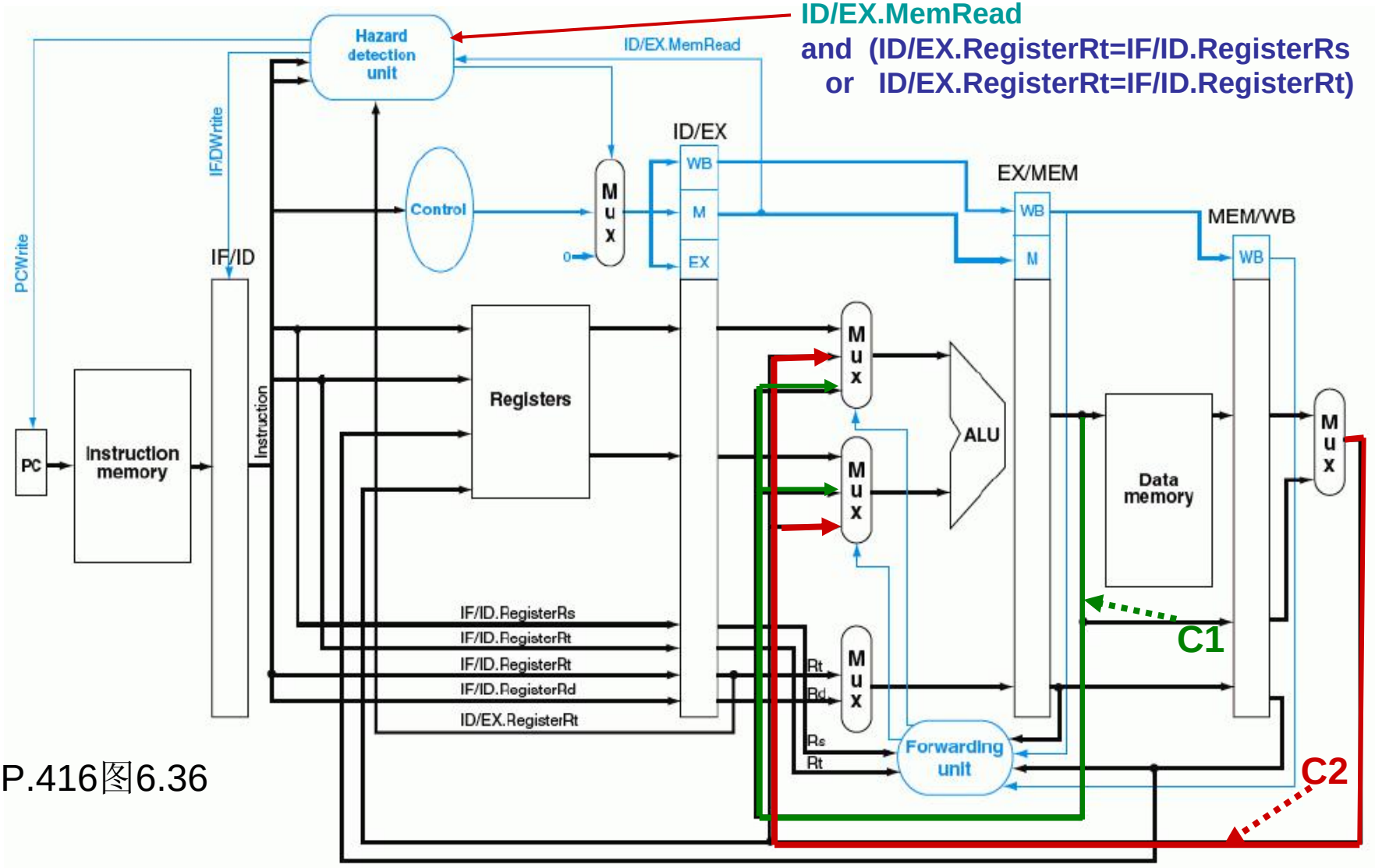
第一条指令在“WB”阶段，寄存器$2正被写入

第二条指令在“MEM”阶段，sub指令是NOOP操作

第三条指令在“EXE”阶段，ALU正在执行“add”操作

第四条指令在“ID/REG”阶段，寄存器$6和$1正被读出

第五条指令在“IF”阶段，指令正被读出



P.416图6.36

C1: EX/MEM.RegWrite and EX/MEM. RegisterRd ≠ 0
and (EX/MEM. RegisterRd=ID/EX. RegisterRs or EX/MEM. RegisterRd=ID/EX. RegisterRt

C2: MEM/WB.RegWrite and MEM/WB. RegisterRd ≠ 0
and (MEM/WB. RegisterRd=ID/EX. RegisterRs or MEM/WB. RegisterRd=ID/EX. RegisterRt

6.18 [5] <§§6.4, 6.5> With regard to the program in Exercise 6.17, explain what the forwarding unit is doing during the fifth cycle of execution. If any comparisons are being made, mention them.

参考答案：第6.17题中的‘指令序列为：

```
add    $2,  $3,  $1
sub    $4,  $3,  $5
add    $5,  $3,  $7
add    $7,  $6,  $1
add    $8,  $2,  $6
```

第五个时钟中，各条指令的执行情况如下：

指令1在“WB”阶段，控制信息等在MEM/WB.Reg中

指令2在“MEM”阶段，控制信息等在EX/MEM.Reg中

指令3在“EXE”阶段，控制信息等在ID/EX.Reg中

指令4在“ID/REG”阶段，指令在IF/ID.Reg中

指令5在“IF”阶段，指令正被读出

“转发”检测条件为：

C1: EX/MEM.RegWrite and EX/MEM. RegisterRd ≠ 0
and (EX/MEM. RegisterRd=ID/EX. RegisterRs or EX/MEM. RegisterRd=ID/EX. RegisterRt

C2: MEM/WB.RegWrite and MEM/WB. RegisterRd ≠ 0
and (MEM/WB. RegisterRd=ID/EX. RegisterRs or MEM/WB. RegisterRd=ID/EX. RegisterRt

根据以上“转发”检测条件，得到比较结果为：

C1: EX/MEM.RegWrite=1(sub指令)、EX/MEM. RegisterRd($4) ≠ 0、EX/MEM. RegisterRd ($4) ≠ ID/EX. RegisterRs($3) 、EX/MEM. RegisterRd($4) ≠ ID/EX. RegisterRt($5)

C2: MEM/WB.RegWrite=1 (add指令) and MEM/WB. RegisterRd ($2)≠ 0 、MEM/WB. RegisterRd($2) ≠ ID/EX. RegisterRs($3) 、MEM/WB. RegisterRd($2) ≠ ID/EX. RegisterRt($7)

由此可知：C1和C2都不满足“转发”条件，所以不需要转发。

6.19 [5] <§§6.4, 6.5> With regard to the program in Exercise 6.17, explain what the hazard detection unit is doing during the fifth cycle of execution. If any comparisons are being made, mention them.

参考答案：第6.17题中的‘指令序列为：

```
add    $2,  $3,  $1
sub    $4,  $3,  $5
add    $5,  $3,  $7
add    $7,  $6,  $1
add    $8,  $2,  $6
```

第五个时钟中，各条指令的执行情况如下：

指令1在“WB”阶段，控制信息等在MEM/WB.Reg中

指令2在“MEM”阶段，控制信息等在EX/MEM.Reg中

指令3在“EXE”阶段，控制信息等在ID/EX.Reg中

指令4在“ID/REG”阶段，指令在IF/ID.Reg中

指令5在“IF”阶段，指令正被读出

“Hazard”(冒险)检测条件为：

ID/EX.MemRead and ID/EX.RegisterRt=IF/ID.RegisterRs or ID/EX.RegisterRt=IF/ID.RegisterRt)

根据以上“冒险阻塞”检测条件，得到比较结果为：

ID/EX.MemRead=0(add指令)、ID/EX. RegisterRt($5) ≠IF/ID. RegisterRs ($6)

ID/EX. RegisterRt($5) ≠ IF/ID. RegisterRt($1)

说明：因为当前在EXE阶段的指令为“add”，所以目的地址应该为Rd而不是Rt。所以， ID/EX. RegisterRt为寄存器$5而不是$3!

由此可知：不满足“冒险”条件，所以不需要阻塞。

6.21 [5] <§6.5> We have a program of 10^3 instructions in the format of "`lw, add, lw, add, ...`" The `add` instruction depends (and only depends) on the `lw` instruction right before it. The `lw` instruction also depends (and only depends) on the `add` instruction right before it. If the program is executed on the pipelined datapath of Figure 6.36 on page 416:

a. What would be the actual CPI?

b. Without forwarding, what would be the actual CPI?

参考答案：

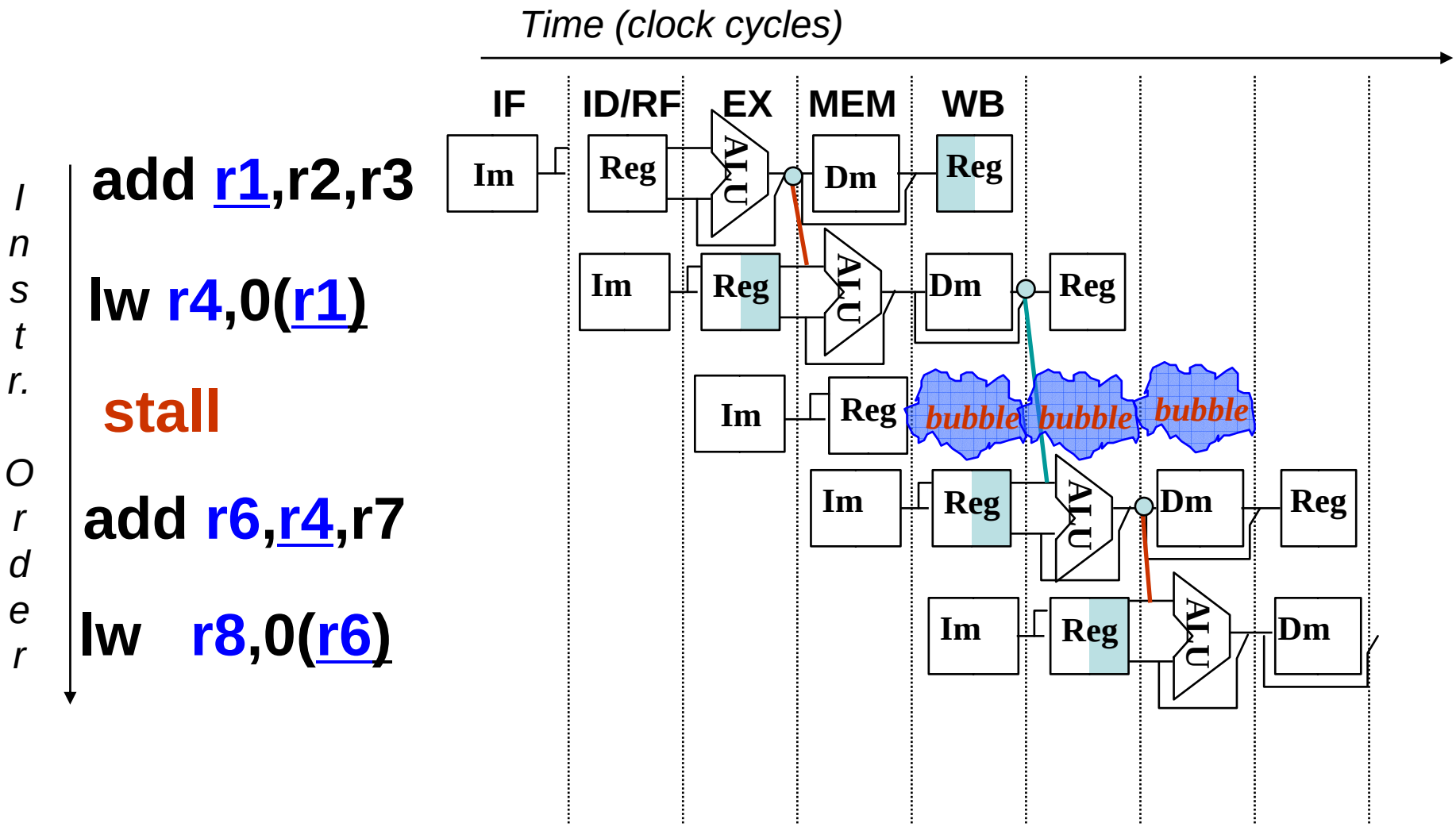
a. 因为**lw**指令和**add**指令之间存在一个**load- use**数据冒险（满足图**6.36**中数据冒险检测条件），所以每个**lw**指令和**add**指令之间要有一次流水线阻塞。而**add**指令和**lw**指令之间的数据冒险满足图**6.36**中的**“转发”**检测条件，故可通过数据转发解决冒险。

即：实际的**CPI**为**1.5**

b. 如果没有转发，则在每条**lw**指令和**add**指令之间将会有两个阻塞，这样每条指令相当于都要有三个时钟才能完成。

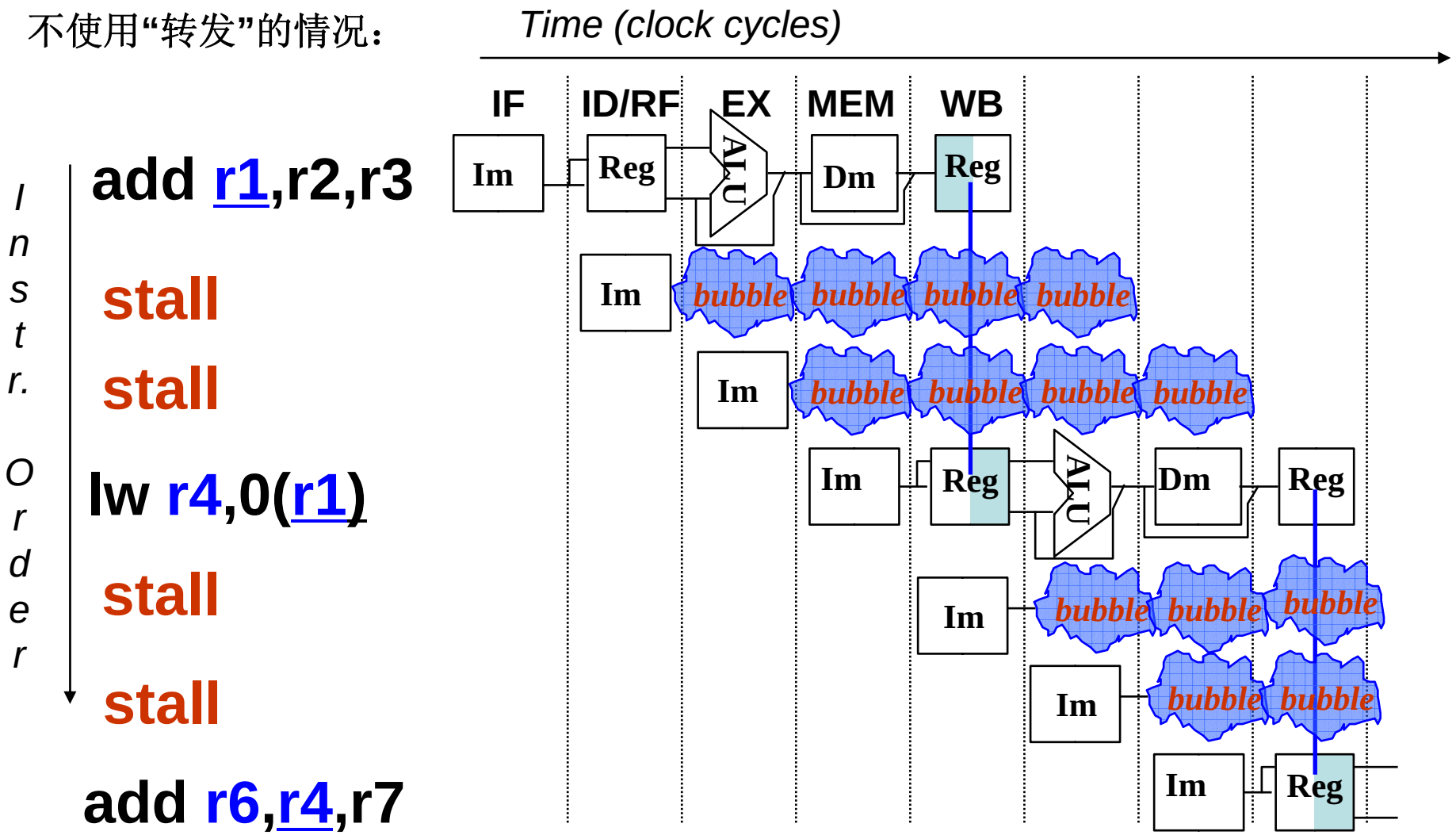
即：**CPI**为**3**

使用“转发”的情况：



使用“转发”时，只有lw指令后需要一次阻塞！

不使用“转发”的情况：



通过寄存器写口/读口分别安排在前半/后半周期，在不使用“转发”时使得每条指令之间只要阻塞两次就可解决！

6.22 [5] <§§6.4, 6.5> Consider executing the following code on the pipelined datapath of Figure 6.36 on page 416:

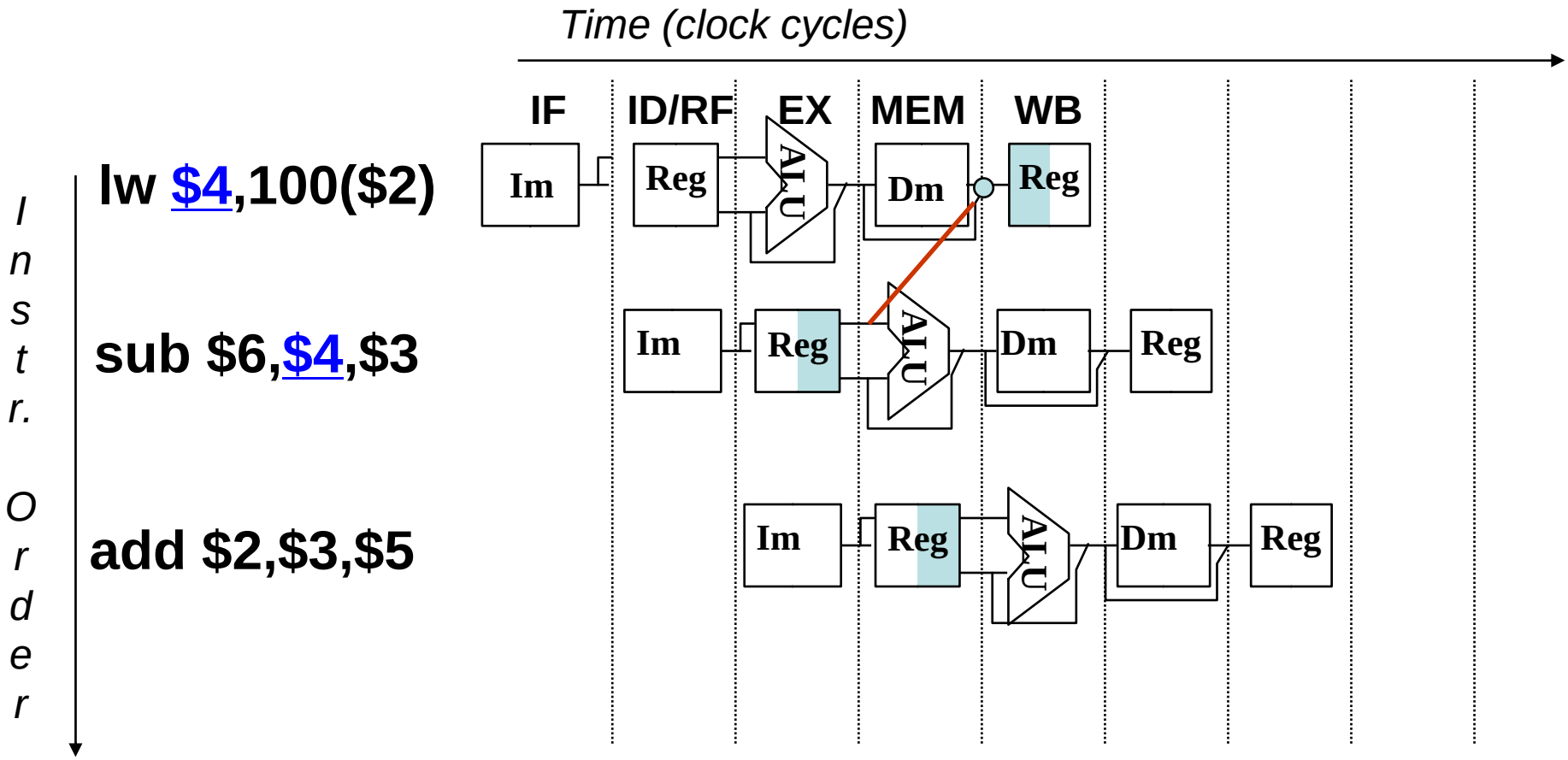
```
lw   $4, 100($2)
sub  $6, $4, $3
add  $2, $3, $5
```

How many cycles will it take to execute this code? Draw a diagram like that of Figure 6.34 on page 414 that illustrates the dependencies that need to be resolved, and provide another diagram like that of Figure 6.35 on page 415 that illustrates how the code will actually be executed (incorporating any stalls or forwarding) so as to resolve the identified problems.

参考答案：从后面的图中可以看出：

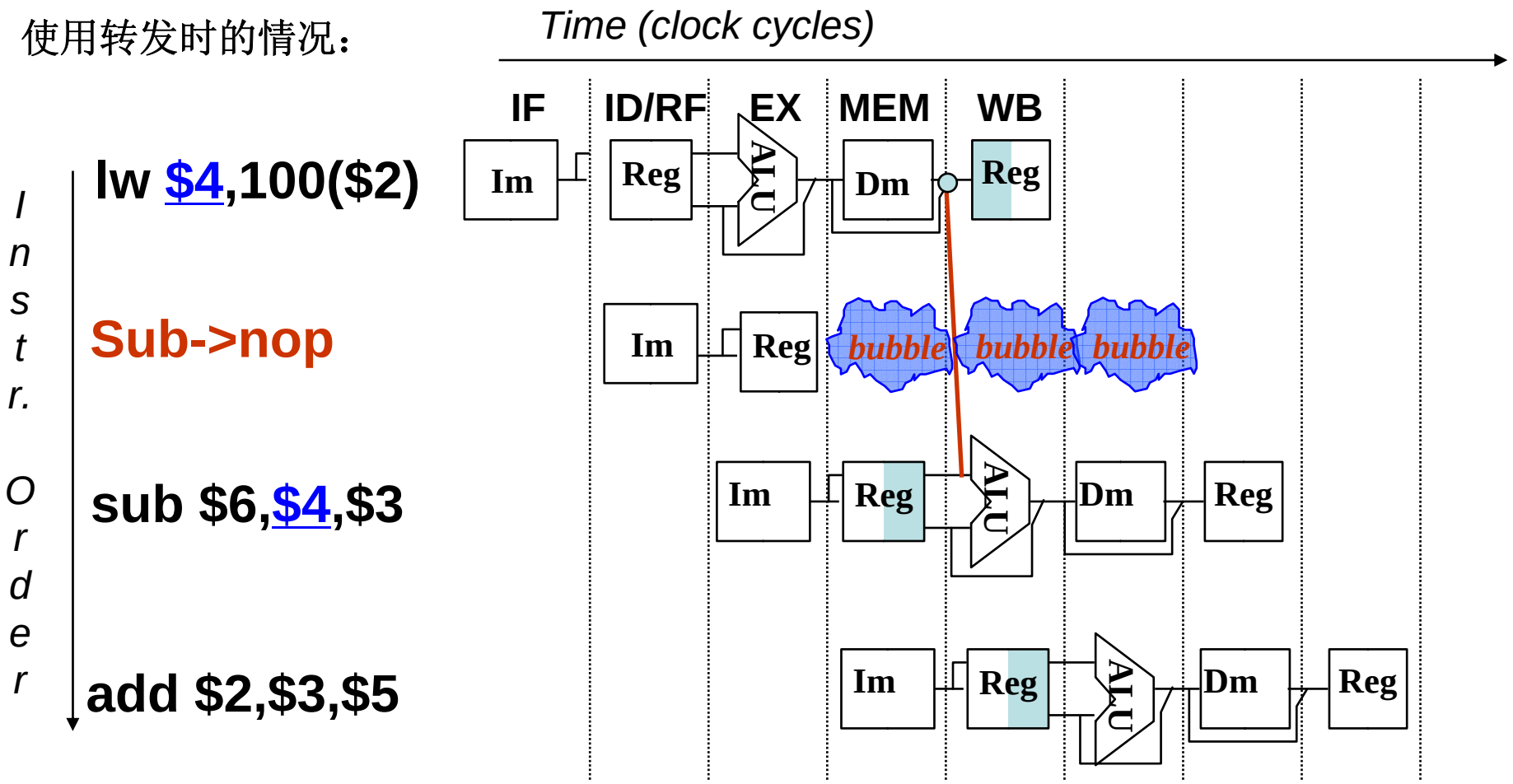
若采用“转发”技术，则执行这段代码需要**8**个时钟周期

若不采用“转发”技术，则执行这段代码需要**11**个时钟周期



寄存器**$4**在第四时钟周期结束时才有值，但**sub**指令在第四周期开始就要用，所以必须使**sub**指令延迟一个周期执行！

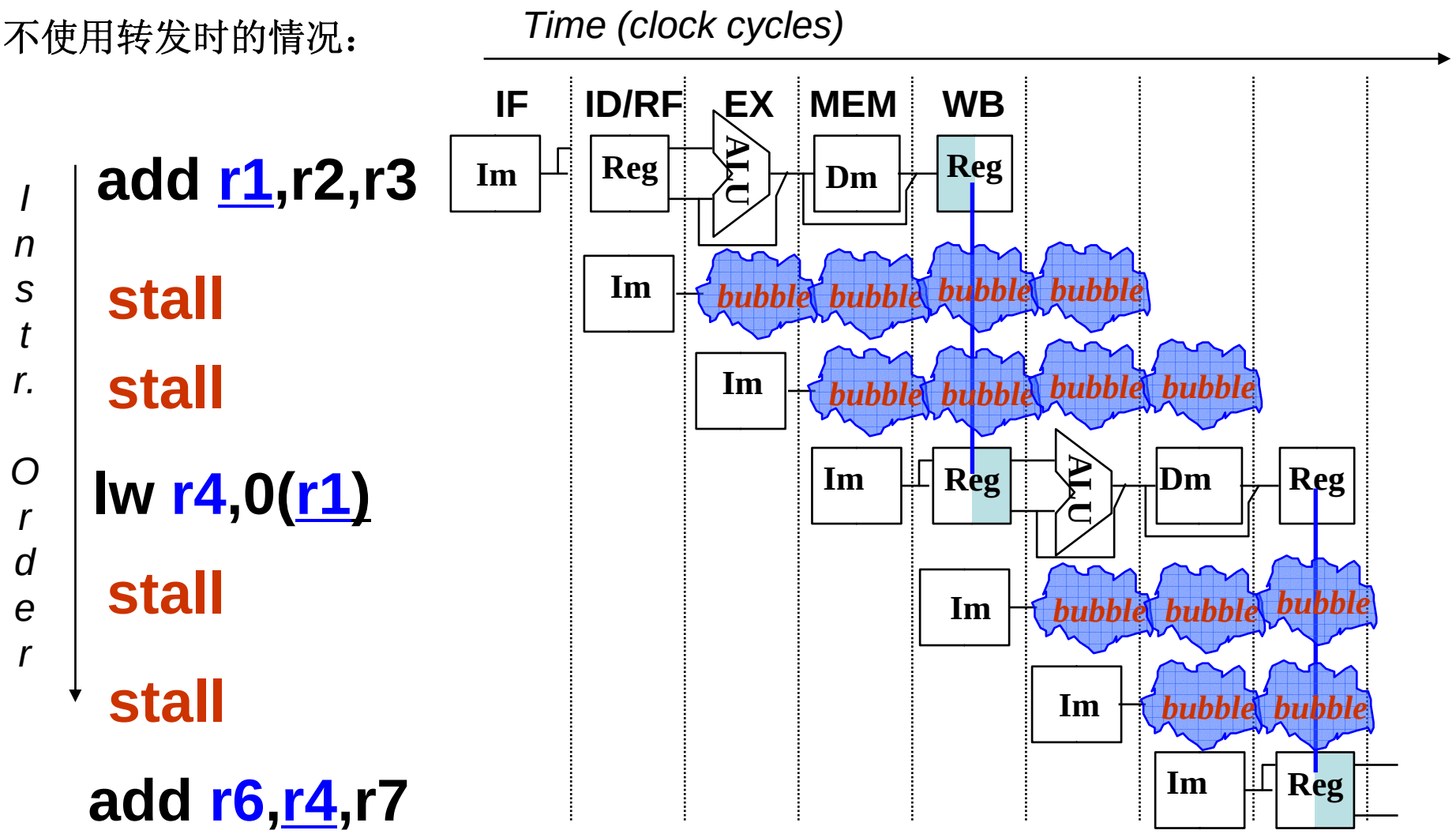
使用转发时的情况：



在EX阶段内，通过“冒险”检测，发现有一个数据冒险存在，在EX阶段结束前进行阻塞，做：

1) 使sub指令控制信号冲刷为0，Sub操作变为nop
2) 使IF/ID流水段中的Sub指令不被add指令冲掉，下个周期继续对sub指令译码
3) 使当前PC不变，下个周期继续取add指令

不使用转发时的情况：



通过寄存器写口/读口分别安排在前半/后半周期，在不使用“转发”时使得每条指令之间只要阻塞两次就可解决！

6.23 [15] <§6.5> List all the inputs and outputs of the forwarding unit in Figure 6.36 on page 416. Give the names, the number of bits, and brief usage for each input and output.

参考答案：图6.36中“转发”检测条件和控制信号为：

C1: EX/MEM.RegWrite and EX/MEM. RegisterRd ≠ 0
and (EX/MEM. RegisterRd=ID/EX. RegisterRs or EX/MEM. RegisterRd=ID/EX. RegisterRt

C2: MEM/WB.RegWrite and MEM/WB. RegisterRd ≠ 0
and (MEM/WB. RegisterRd=ID/EX. RegisterRs or MEM/WB. RegisterRd=ID/EX. RegisterRt

ForwardA (ForwardB) = { 01 当c2=1时
10 当c1=1时

由此可见，图6.36中“转发”单元的输入和输出为：

Input	Number of bits	Usage
ID/EX.RegisterRs	5	operand reg number, compare to see if match
ID/EX.RegisterRt	5	operand reg number, compare to see if match
EX/MEM.RegisterRd	5	destination reg number, compare to see if match
EX/MEM.RegWrite	1	TRUE if writes to the destination reg
MEM/WB.RegisterRd	5	destination reg number, compare to see if match
MEM/WB.RegWrite	1	TRUE if writes to the destination reg
Output	**Number of bits**	**Usage**
ForwardA	2	forwarding signal
ForwardB	2	forwarding signal

6.30 [7] <§§6.4, 6.5> In the example on page 425, we saw that the performance advantage of the multicycle design was limited by the longer time required to access memory versus use the ALU. Suppose the memory access became 2 clock cycles long. Find the relative performance of the single-cycle and multicycle designs. In the next few exercises, we extend this to the pipelined design, which requires lots more work!

参考答案：各类指令所用功能部件的时间为

Instruction class	Instruction memory	Register read	ALU operation	Data memory	Register write	Total
R-type	200	50	100	0	50	400 ps
Load word	200	50	100	200	50	600 ps
Store word	200	50	100	200		550 ps
Branch	200	50	100	0		350 ps
Jump	200					200 ps

存储器操作变为两个时钟周期后，其单周期数据通路的时钟周期不变，为**600ps**

而多周期数据通路中，各类指令的时钟周期变为：

load：7；Store：6；ALU：5；beq：4；Jump：4

以**SPECINT2000**混合指令比例计算，**CPI=0.25x7+0.10x6+0.52x5+0.11x4+0.02x4 = 5.47**

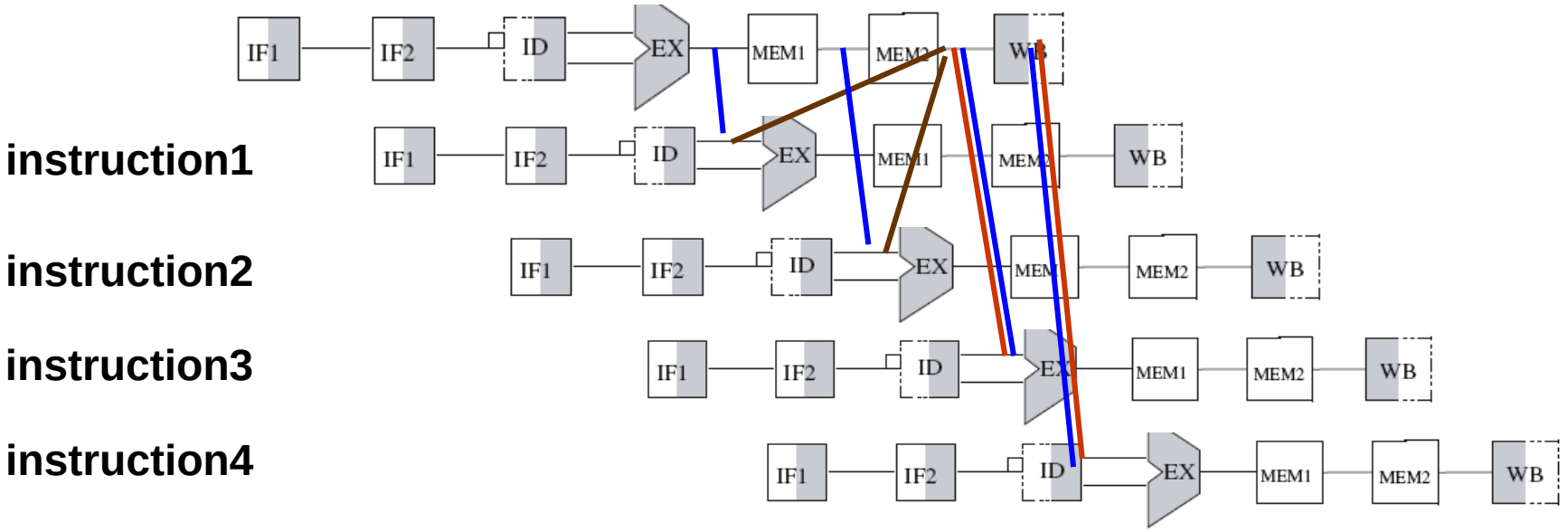
存储器操作变为两个时钟周期后，多周期数据通路的时钟周期为**100ps**，

故一条指令的执行时间为**100x5.47=547ps**

比较结果：多周期比单周期快！

6.33 [20] <§§6.2–6.6> In the example on page 425, we saw that the performance advantage of both the multicycle and the pipelined designs was limited by the longer time required to access memory versus use the ALU. Suppose the memory access became 2 clock cycles long. Draw the modified pipeline. List all the possible new forwarding situations and all possible new hazards and their length.

参考答案：存储器操作变为两个时钟周期后，其流水线包含了以下7个阶段：



当第一条指令为lw指令时，instruction1-4的执行情况由咖啡色（不能转发）和红色（可转发）表示。instruction1需要2个“stall”； instruction2需要1个“stall”；后续指令3可以通过“转发”解决（后续指令4不是数据冒险）。

当第一条指令为ALU指令时，instruction1-4的执行情况由兰色表示。说明后续所有的数据冒险都可以通过“转发”解决（后续指令4不是数据冒险）。

6.34 [20] <§§6.2–6.6> Redo the example on page 425 using the restructured pipeline of Exercise 6.33 to compare the single-cycle and multicycle. For branches, assume the same prediction accuracy, but increase the penalty as appropriate. For loads, assume that the subsequent instructions depend on the load with a probability of 1/2, 1/4, 1/8, 1/16, and so on. That is, the instruction following a load by two has a 25% probability of using the load result as one of its sources. Ignoring any other data hazards, find the relative performance of the pipelined design to the single-cycle design with the restructured pipeline.

参考答案：根据P425中的例子，已知：

各主要功能单元的操作时间为：

- 存储单元：200ps （被分成100ps的两个阶段）
- ALU和加法器：100ps
- 寄存器堆（读 / 写）：50ps

假设MUX、控制单元、PC、扩展器和传输线路都没有延迟，指令组成为：

25%取数、10%存数、52%ALU、11%分支、2%跳转

则下面实现方式中，哪个更快？快多少？

（1）单周期方式：每条指令在一个固定长度的时钟周期内完成

（2）流水线方式：取指1、取指2、取数/译码、执行、存取1、存取2、写回七个阶段

对于单周期方式：

时钟周期将由最长指令来决定，应该是load指令，为600ps

所以，N条指令的执行时间为600N(ps)

对于流水线方式：

存储器操作变为两个时钟周期后，其流水线包含了7个阶段.

对于beq，若预测正确，则为1个周期，若预测错误，则为3个周期（与原五段流水线相比，多一个取指周期，多阻塞了1个周期），故CPI=1/4x3+3/4x1=1.5

对于load，随后第一条则为3个（阻塞2个）周期；随后第二条则为2个（阻塞1个）周期，以后的指令都不需要阻塞，故CPI=1/2x3+1/2x1/4x2+3/8x1=2.125

对于ALU指令，随后的数据相关指令都可通过转发解决，故CPI=1

对于Store指令，不会发生数据冒险，故CPI=1

对于Jump指令，总要等到译码结束才能确定转移地址，故CPI=3

平均CPI为：2.125x25%+1x10%+1x52%+1.5x11%+3x2%=1.38

所以， N条指令的执行时间为1.38x100xN=138N(ps)

流水线比单周期快600/138=4.38倍。

6.35 [10] <§§6.4–6.6> As pointed out on page 418, moving the branch comparison up to the ID stage introduces an opportunity for both forwarding and hazards that cannot be resolved by forwarding. Give a set of code sequences that show the possible forwarding paths required and hazard cases that must be detected, considering only one of the two operands. The number of cases should equal the maximum length of the hazard if no forwarding existed.

参考答案：根据**6.6.2**节中所指出的，将分支比较操作提前到**ID**阶段，会导致来不及通过转发来解决数据冒险（即：若不提前的话，本可以通过转发解决的）

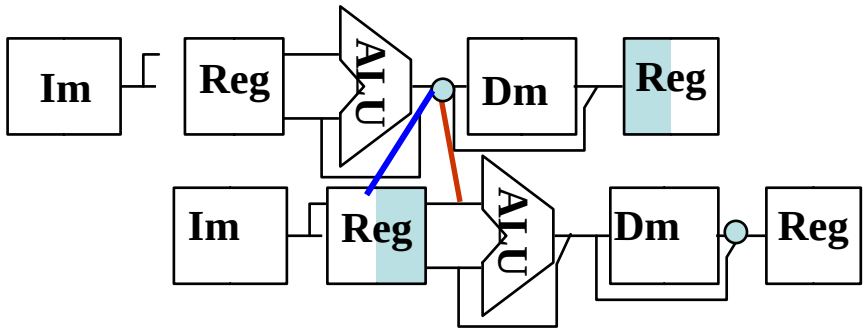
当分支指令依赖于仍在流水段中的结果时，便来不及通过转发来解决。

例如，以下的例子中，如果分支比较不提前，则**add**指令**EXE**阶段执行的结果（在**EX/MEM**流水段寄存器）可以转发给**beq**指令的**EXE**阶段进行比较（红线），但如果**beq**指令在**ID**阶段比较的话，就来不及转发了（兰线）。

lw $1, 100 ($2)

add $1, $1, 8

beq $1, $3, 10



在给出的例子中，第一条指令和第二条指令中还有一个**load-use**数据冒险，也不能通过转发来消除。

所以，该例中**1-2**之间的“阻塞”和**2-3**之间的“转发”都不可通过转发来消除！

6.36 [15] <§6.6> We have a program core consisting of five conditional branches. The program core will be executed thousands of times. Below are the outcomes of each branch for one execution of the program core (T for taken, N for not taken).

Branch 1: T-T-T
Branch 2: N-N-N-N
Branch 3: T-N-T-N-T-N
Branch 4: T-T-T-N-T
Branch 5: T-T-N-T-T-N-T

Assume the behavior of each branch remains the same for each program core execution. For dynamic schemes, assume each branch has its own prediction buffer and each buffer initialized to the same state before each execution. List the predictions for the following branch prediction schemes:

a. Always taken
b. Always not taken
c. 1-bit predictor, initialized to predict taken
d. 2-bit predictor, initialized to weakly predict taken

What are the prediction accuracies?

参考答案：预测准确率=预测正确次数 / 总预测时间 *100%

a. B1: R-3, W-0； B2: R-0, W-4； B3: R-3, W-3； B4: R-4, W-1； B5: R-5, W-2； 60%

b. B1: R-0, W-3； B2: R-4, W-0； B3: R-3, W-3； B4: R-1, W-4； B5: R-2, W-5； 40%

c. B1: R-3, W-0； B2: R-3, W-1； B3: R-1, W-5； B4: R-3, W-2； B5: R-3, W-4； 52%

d. B1: R-3, W-0； B2: R-3, W-1； B3: R-3, W-3； B4: R-4, W-1； B5: R-5, W-2； 72%

6.39 [10] <§§6.4–6.6> The example on page 378 shows how to *maximize* performance on our pipelined datapath with forwarding and stalls on a use following a load. Rewrite the following code to *minimize* performance on this datapath—that is, reorder the instructions so that this sequence takes the *most* clock cycles to execute while still obtaining the same result.

```
lw    $2, 100($6)
lw    $3, 200($7)
add   $4, $2, $3
add   $6, $3, $5
sub   $8, $4, $6
lw    $7, 300($8)
beq   $7, $8, Loop
```

参考答案：要使得上述代码段的性能最差，则只要让代码段中出现**load-use**冒险最多

```
lw    $2, 100($6)
add   $4, $2, $3
lw    $3, 200($7)
add   $6, $3, $7
sub   $8, $4, $6
lw    $7,  300($8)
beq   $7, $8, Loop
```

这样做有什么问题吗？

6.40 [20] <§6.6> Consider the pipelined datapath in Figure 6.54 on page 461. Can an attempt to flush and an attempt to stall occur simultaneously? If so, do they result in conflicting actions and/or cooperating actions? If there are any cooperating actions, how do they work together? If there are any conflicting actions, which should take priority? Is there a simple change you can make to the datapath to ensure the necessary priority? You may want to consider the following code sequence to help you answer this question:

```
        beq $1, $2, TARGET   # assume that the branch is taken
        lw  $3, 40($4)
        add $2, $3, $4
        sw  $2, 40($4)
TARGET: or  $1, $1, $2
```

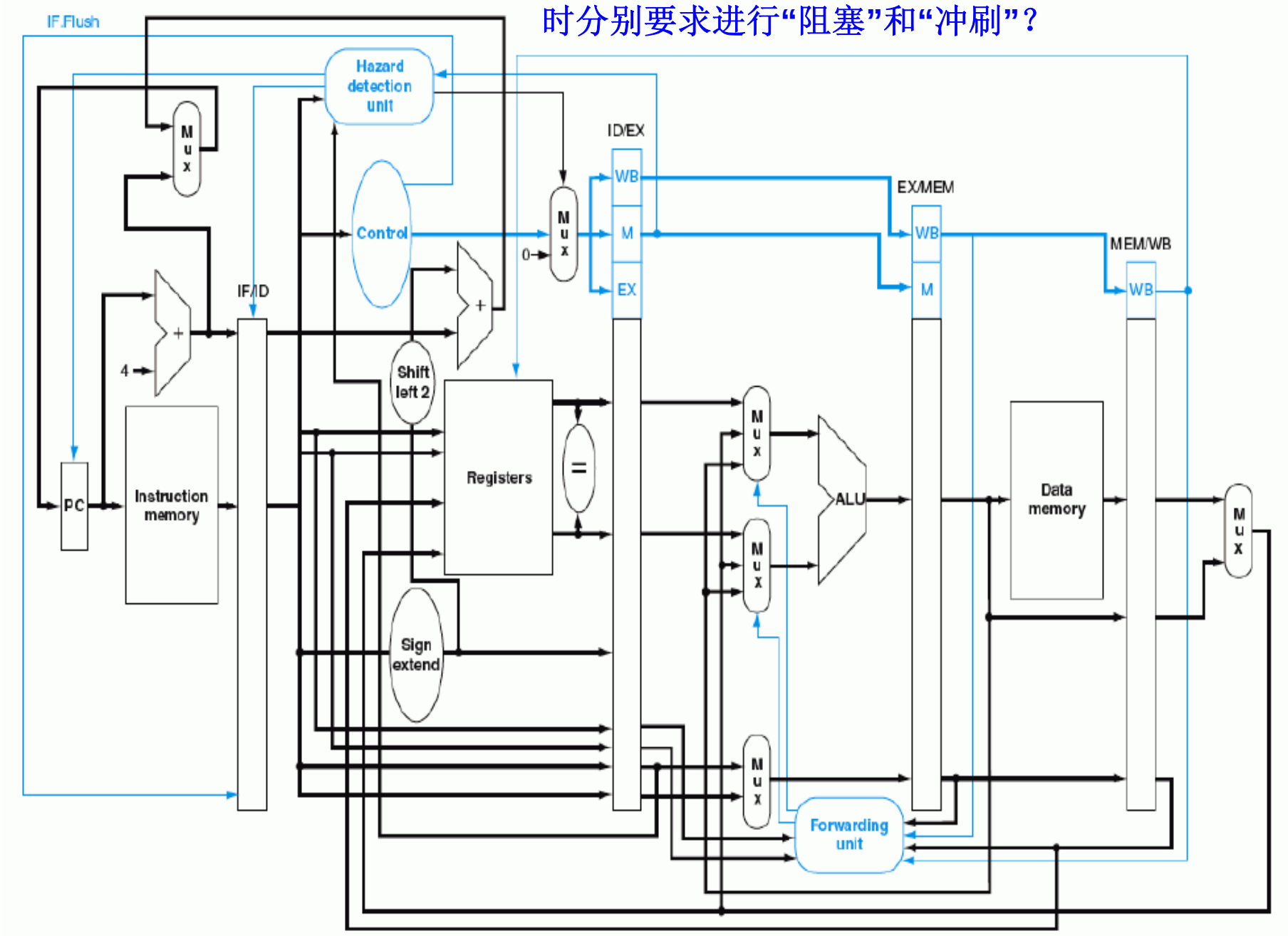
参考答案：

从给定的图可以看出，**beq**指令是在**ID**段确定是否转移，并计算转移地址的。当判断要转移（**taken**）时，**ID**段会产生一个**Flush**信号，使得下一条已被取出的指令**(lw)**被清**0**，并控制将转移地址送**PC**，流水线被阻塞一个时钟后，从转移地址处开始执行。故不会同时发生**“Flush”**和**“Stall”**

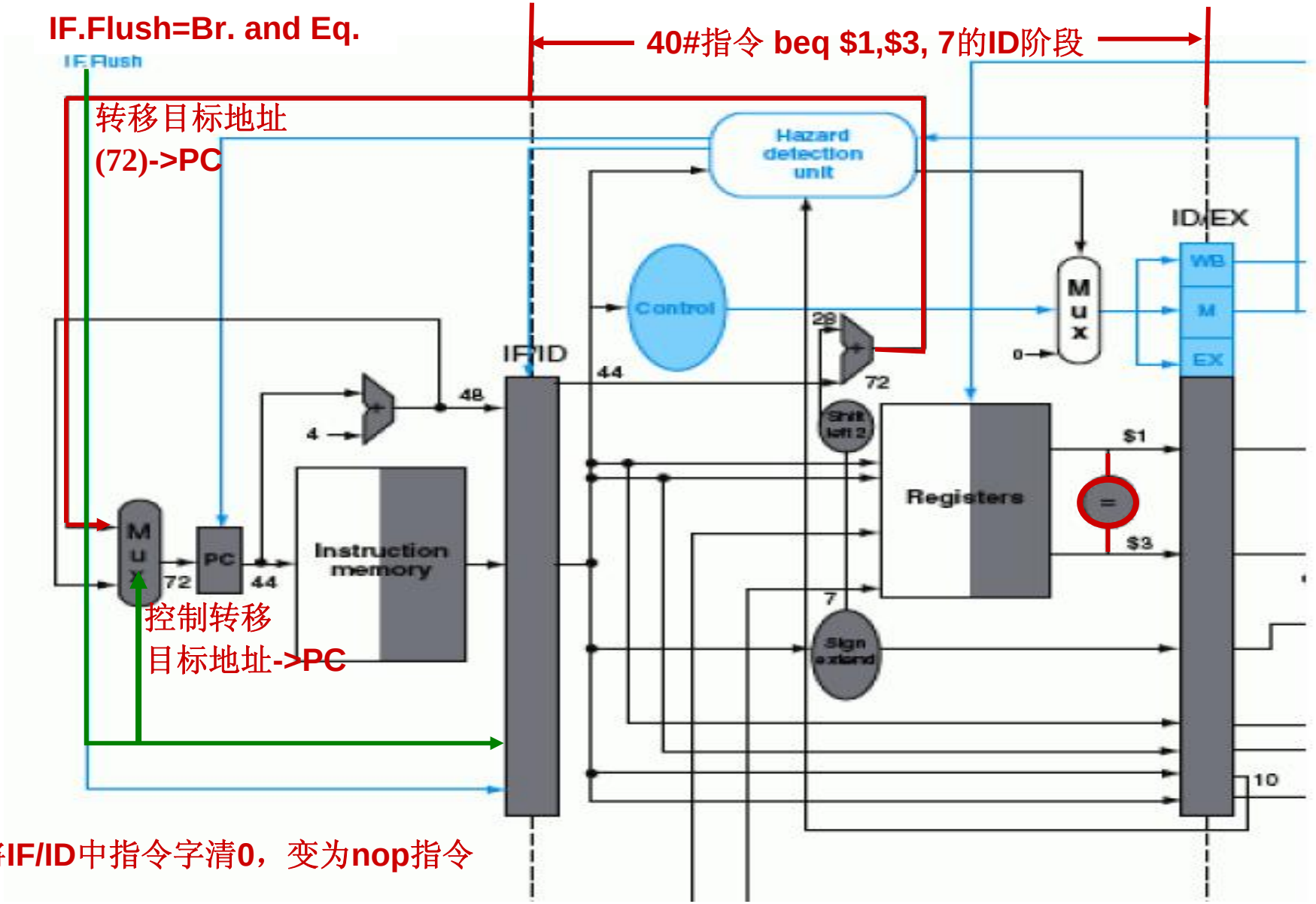
假定**beq**指令的分支判断和转移地址计算没有提前，还是在**MEM**阶段时，则在**MEM**阶段会产生一个**Flush**信号，此时，在**ID**阶段同时检测到**load-use**冲突，并引起一次**“stall”**。因此，这种情况下，**Flush**和**Stall**会同时发生。

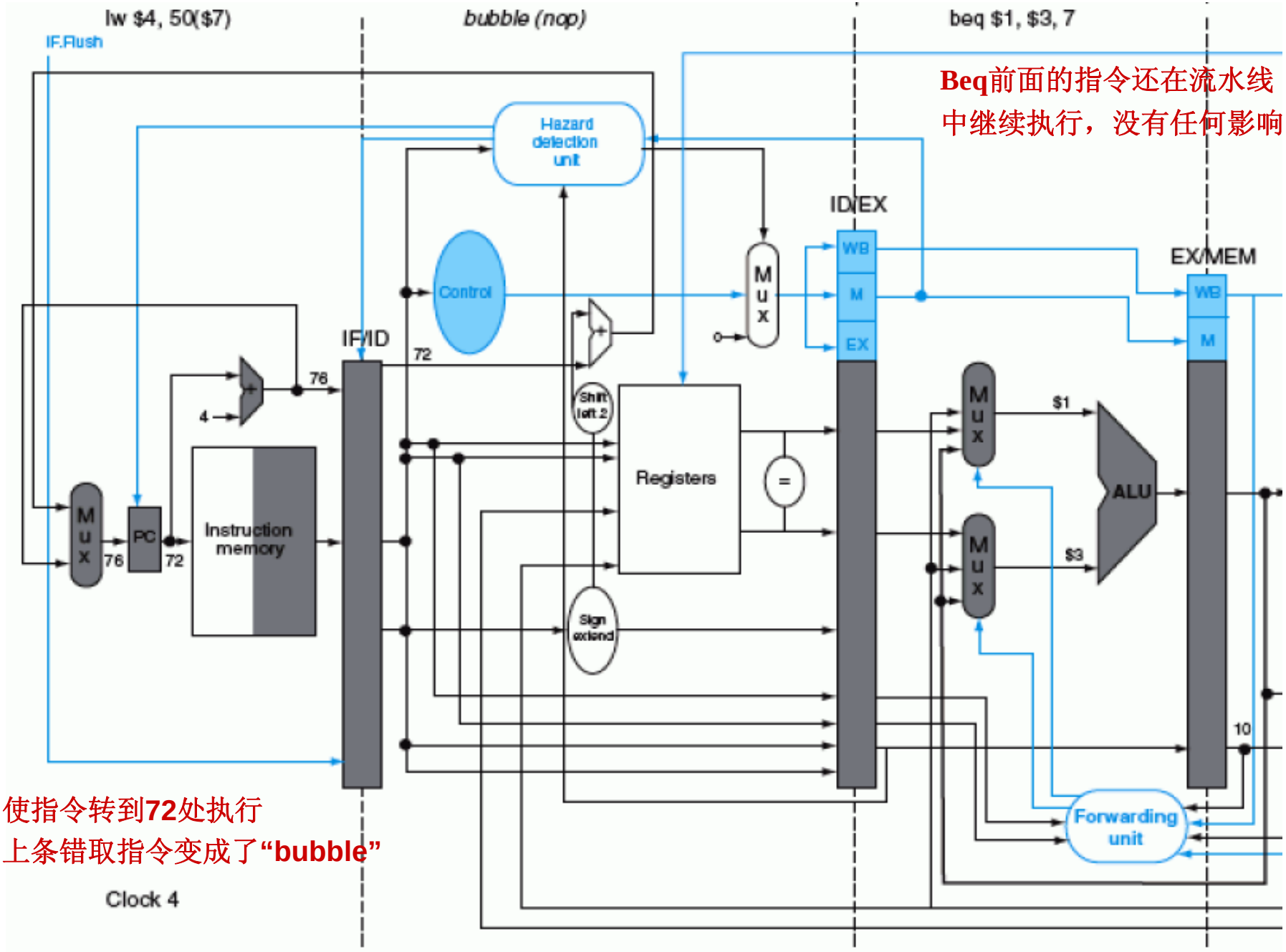
Figure6.45 on P.461

题意：图中“冲突”检测和“分支判断”是否可能同时分别要求进行“阻塞”和“冲刷”？



带静态分支预测处理的数据通路





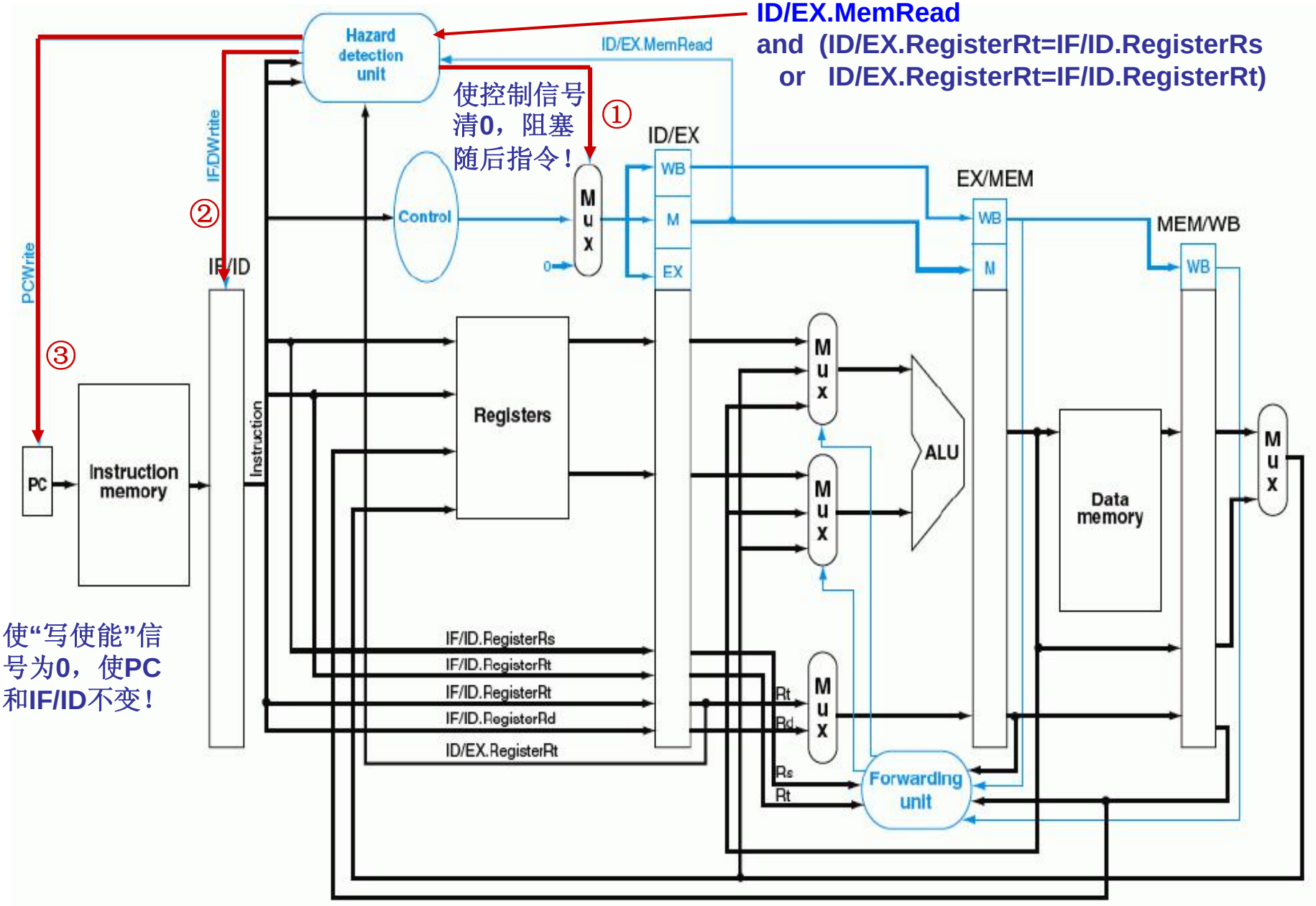
Beq前面的指令还在流水线中继续执行，没有任何影响

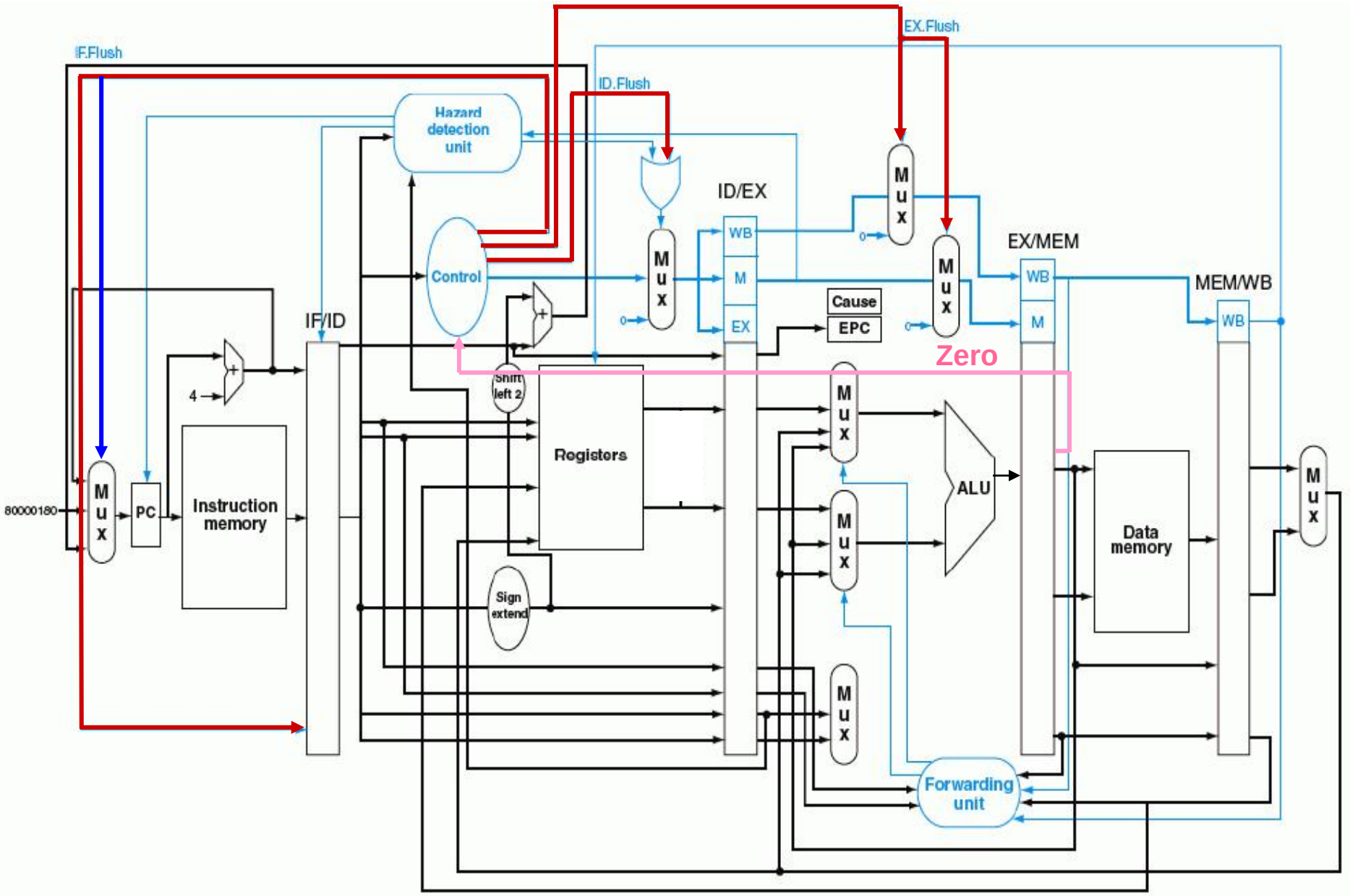
使指令转到72处执行

上条错取指令变成了“bubble”

BACK

带“转发”和“阻塞”检测的流水线数据通路





同时检测到“Flush”和“Stall”时，会产生矛盾，此时，必须保证Flush的优先级更高！

可以将两个检测电路合在一起，并加上并行判优电路

6.47 [10] <§6.9> The following code has been unrolled once but not yet scheduled. Assume the loop index is a multiple of two (i.e., $10 is a multiple of eight):

```
Loop:   lw   $2,  0($10)
        sub  $4,  $2, $3
        sw   $4,  0($10)
        lw   $5,  4($10)
        sub  $6,  $5, $3
        sw   $6,  4($10)
        addi $10, $10, 8
        bne  $10, $30, Loop
```

循环中，有2个load-use冒险和一个控制冒险（3次阻塞），所以共有5次阻塞，因而每次循环共需8+5=13个周期。

这样对吗？ 不采用任何预测时，如是！

采用静态预测(初始预测转移)时，怎样？

最后1次循环需8+5=13个周期，前面各次预测都能成功，故只需8+2=10个周期

Schedule this code for fast execution on the standard MIPS pipeline (assume that it supports addi instruction). Assume initially $10 is 0 and $30 is 400 and that branches are resolved in the MEM stage. How does the scheduled code compare against the original unscheduled code?

参考答案：优化调度后的代码段为：

```
Loop:   lw   $2, 0($10)
        lw   $5, 4($10)
        sub  $4, $2, $3
        sub  $6, $5, $3
        sw   $4, 0($10)
        sw   $6, 4($10)
        addi $10, $10, 8
        bne  $10, $30, Loop
```

不预测时，优化调度指令顺序后，消除了load-use冒险，有一个控制冒险（3次阻塞），因而每次循环共需8+3=11个周期。

的操作次数为400/4=100次，循环次数为100/2=50

，所以优化后的程序用11x50=550个周期

程序为13x50=650个周期

以，性能提高了 650/550=1.18倍

采用简单静态预测(初始预测转移)时，性能如何？

最后1次需8+3=11个周期，前面各次只需8个周期

优化后的程序用11x1+8x49=403个周期

原程序为13x1+10x49=503个周期

性能各提高了 650/503=1.29，550/403=1.36

优化后提高了 503/403=1.25倍

6.48 [20] <§6.9> This exercise is similar to Exercise 6.47, except this time the code should be unrolled twice (creating three copies of the code). However, it is not known that the loop index is a multiple of three, and thus you will need to invent a means of ensuring that the code still executes properly. (Hint: Consider adding some code to the beginning or end of the loop that takes care of the cases not handled by the loop.)

参考答案：

题目给出的一次循环展开代码中循环体有三条指令（lw,sub和sw），已知循环次数是2的倍数，将循环体展开三次后，循环结束条件的判断要作相应的调整，以保证展开后的代码能得到正确的结果。

展开为3次后，操作次数可能的情况为3、6、9、12、15、18、… …，故对于4、6、8、10、12、14、16、18、… … 的操作次数来说，有以下三种可能：

1) 操作次数是4、10、16…，按3的倍数循环的对应次数为3、9、15 …，少了1次
2) 操作次数是6、12、18 …，正好既是3的倍数，又是2的倍数，不多不少
3) 操作次数是8、14、20…，按3的倍数循环的对应次数为6、12、18 …，少了2次

所以，需要在代码中加入操作次数调整部分

循环展开3次的代码段如右所示。

在循环的开始，判断是否剩下的循环次数小于3，是的话，转到leftover进行结束前处理；否则，进入循环

在循环中，判断循环次数是否正好是3的倍数，是的话，则直接跳转到结束（finish处）

在结束前处理中，先补充操作一次，然后判断是否结束，是的话，跳转到结束（finish处）；否则，再补充操作一次

右边代码段的性能分析如下：

1）控制冒险：循环内有两条分支指令bgt、bne；循环外有一条jump指令(1次阻塞)和一条分支指令

2）load-use冒险：1次或2次

总操作次数为400/4=100，则循环次数为100/3=33次，补充1次操作

总的周期数为
(12+3+3)x33+1+4+2+3+4=608

比前面两种代码的性能分别提高了
650/608=1.07倍　550/608=0.90倍

周期数计算是在无预测的情况下进行的

```
Loop:       addi $10, $10, 12
            bgt  $10, $30, Leftover
            lw   $2, -12($10)
            lw   $5, -8($10)
            lw   $7, -4($10)
            sub  $4, $2, $3
            sub  $6, $5, $3
            sub  $8, $7, $3
            sw   $4, -12($10)
            sw   $6, -8($10)
            sw   $8, -4($10)
            bne  $10, $30, Loop
            jump Finish
Leftover:   lw   $2, -12($10)
            sub  $4, $2, $3
            sw   $4, -12($10)
            addi $10, $10, -8
            beq  $10, $30, Finish
            lw   $5, 4($10)
            sub  $6, $5, $3
            sw   $6, 4($10)
Finish: ...
```

假定采用简单静态预测方法，则结果应该不同！

采用静态预测(初始预测转移)时，性能分析如下：

总操作次数为400/4=100，循环次数为100/3=33次，补充1次操作

1）控制冒险：

bgt只在最后1次转移，第一次和最后一次预测错误；

bne指令每次都发生转移，无预测错误；

beq指令发生转移，无预测错误；

jump指令发生一次阻塞

2）load-use冒险：1次或2次

总的周期数为：

(12+3)x1+12x32+1+4+2+3+1=410

采用静态预测后，性能提高了608/410=1.48倍

比前面两种代码的性能分别提高了

503/410=1.22倍

403/410=0.98倍

本题说明了是否优化调度、循环展开次数如何选择、是否采用预测等方面对程序性能的影响。你的结论是什么？

优化调度能消除循环内大量load-use冒险，性能约提高20%；循环展开次数选择不当会影响性能，降低2%-10%；采用预测会大大提高性能，约提高30%-59%

```
Loop:    addi $10, $10, 12
         bgt  $10, $30, Leftover
         lw   $2, -12($10)
         lw   $5, -8($10)
         lw   $7, -4($10)
         sub  $4, $2, $3
         sub  $6, $5, $3
         sub  $8, $7, $3
         sw   $4, -12($10)
         sw   $6, -8($10)
         sw   $8, -4($10)
         bne  $10, $30, Loop
         jump Finish
Leftover: lw  $2, -12($10)
         sub  $4, $2, $3
         sw   $4, -12($10)
         addi $10, $10, -8
         beq  $10, $30, Finish
         lw   $5, 4($10)
         sub  $6, $5, $3
         sw   $6, 4($10)
Finish: ...
```

6.49 [20] <§6.9> Using the code in Exercise 6.47, unroll the code four times and schedule it for the static multiple-issue version of the MIPS processor described on pages 436–439. You may assume that the loop executes for a multiple of four times.

参考答案：

将代码段中的循环展开四次，放在一个**2**发射**MIPS**流水线中执行时，可以按以下方式进行调度。

	ALU 或 Branch	lw 或 sw
Loop:	addi $20, $10, 0	lw $2, 0($10)
		lw $5, 4($10)
	sub $4, $2, $3	lw $7, 8($10)
	sub $6, $5, $3	lw $9, 12($10)
	sub $8, $7, $3	sw $4, 0($10)
	sub $11, $9, $3	sw $6, 4($10)
	addi $10, $10, 16	sw $8, 8($20)
	bne $10, $30, loop	sw $11, 12($20)

用重命名机制避免了寄存器**$10**在最后两行中的名字依赖关系**name Dependence**（或反依赖关系**antidependence**，不是真实依赖）

用**$20**替换了**$10**

这种超标量方式下，其性能又如何呢？