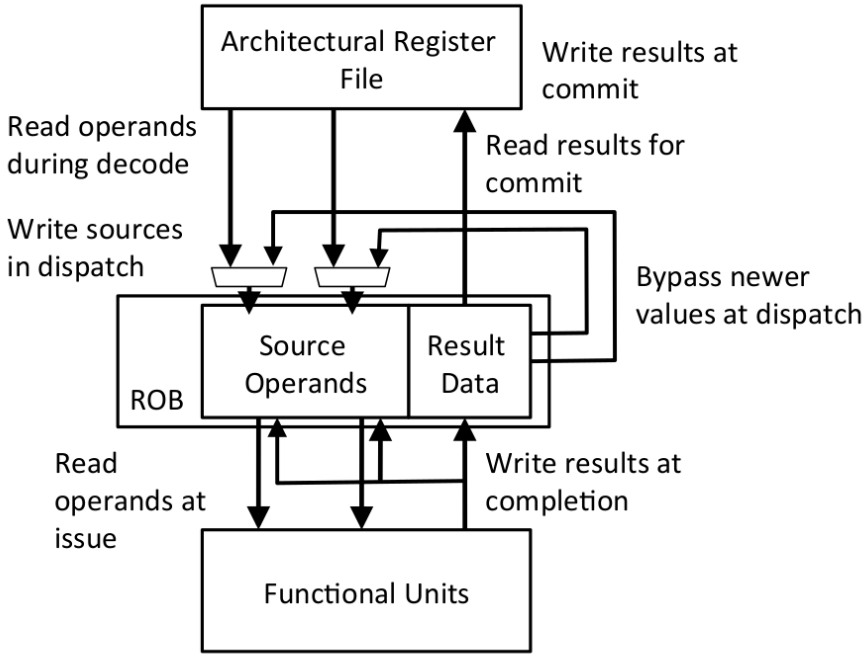
Out-of-Order Scheduling

Q6. This problem deals with an out-of-order single-issue processor that is based on the basic RISC-V pipeline and has a floating-point unit. The FPU has one adder, one multiplier, and one load/store unit. The adder has a two-cycle latency and is fully pipelined. The multiplier has a six-cycle latency and is fully pipelined. Assume that stores take one cycle and loads take two cycles. 此问题涉及基于基本 RISC-V 流水线并具有浮点单元的无序单发出处理器。 FPU 有一个加法器、一个乘法器和一个加载/存储单元。 加法器具有两个周期的延迟，并且是完全流水线化的。 乘法器具有六个周期的延迟，并且是完全流水线化的。 假设存储需要一个周期，加载需要两个周期。

There are 31 integer registers (x1-x32) and 32 floating-point registers (f0-f31). To maximize number of instructions that can be in the pipeline, register renaming is used. The decode stage can add up to one instruction per cycle to the re-order buffer (ROB). The CPU uses a data-in-ROB design, so there is one rename register associated with each ROB entry. Functional units write back to the ROB upon completion. The functional units share a single write port to the ROB. In the case of a write-back conflict, the older instruction writes back first. The instructions are committed in order and only one instruction may be committed per cycle. The earliest time an instruction can be committed is one cycle after write back. 有 31 个整数寄存器 (x1-x32) 和 32 个浮点寄存器 (f0-f31)。 为了最大限度地增加流水线中的指令数量，使用了寄存器重命名。 解码阶段每个周期最多可以将一条指令添加到重新排序缓冲区 (ROB)。 CPU 使用 ROB 数据设计，因此每个 ROB 条目都有一个重命名寄存器。 功能单元在完成后写回 ROB. 功能单元共享一个写入 ROB 的端口。 如果发生回写冲突，则较旧的指令会先回写。 指令按顺序提交，每个周期只能提交一条指令。 一条指令可以提交的最早时间是写回后的一个周期。

Floating-point instructions (including loads writing floating-point registers) must spend one cycle in the write-back stage before their result can be used. Integer results are available for bypass the next cycle after issue and write back two cycles after issue. 浮点指令（包括写入浮点寄存器的负载）必须在回写阶段花费一个周期才能使用其结果。 整数结果可用于绕过发布后的下一个周期并在 issue 后两个周期回写。



For the following questions, we will evaluate the performance of the code segment below.

I_1	`FLD  f1, 0(x1)`
I_2	`FMUL.D f2, f1, f0` 依赖 `f1`
I_3	`FADD.D f3, f2, f0` 依赖前面的 `f2`
I_4	`ADDI  x1, x1, 8`
I_5	`FLD  f1, 0(x1)`
I_6	`FMUL.D f2, f1, f1`
I_7	`FADD.D f2, f2, f3`

A) For this part, consider an ideal case where we have an unlimited number of ROB entries.
In the table below, fill in the cycle number for when each instruction enters the ROB, issues, writes back, and commits. Also, fill in the new register names for each instruction, where applicable. (10 points)
Since we have an infinite supply of register names, you should use a new register name for each register that is written (p0, p1, …). Keep in mind that after a register has been renamed, subsequent instructions that refer to that register must refer to the new register name. 对于这部分，考虑一个理想情况，即我们有无限数量的 ROB 条目。在下表中，填写每条指令进入 ROB、发出、回写和提交时的周期数。 此外，在适用的情况下，为每条指令填写新的寄存器名称。 （10 分）由于我们有无限的寄存器名称，您应该为每个写入的寄存器使用一个新的寄存器名称 (p0, p1, ...)。 请记住，在重命名寄存器后，引用该寄存器的后续指令必须引用新的寄存器名

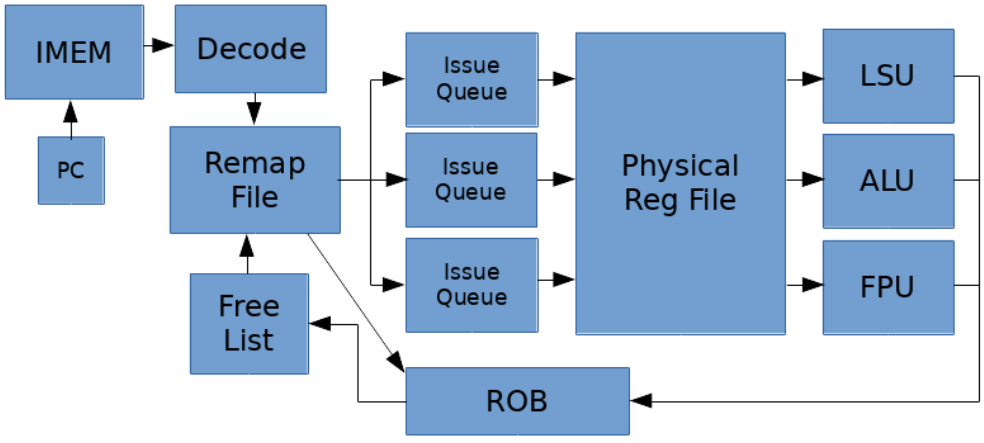
	Time				OP	Dest	Src1	Src2
	Enter ROB	Issue	WB	Commit				
I_1	-1	0	2	3	FLD	p0	x1	-
I_2	0	3	9 乘法器具有六个周期的延迟	10 每个周期只能提交一条指令	FMUL.D	p1	p0	f0
I_3	1	10	12 加法器两个周期的延迟	13	FADD.D	P2	P1	F0
I_4	2	4 和前面无关就可以先发射	6issue 后两个周期	14 提交要等 I3	ADDI	P3 您应该为每个写入的寄存器使用一个新的寄存器名称	X1	-
I_5	3	5	7 加载需要两个周期	15	FLD	F1 重命名为 P4	P3	-
I_6	4	8	14	16	FMUL.D	P5	P4	P4
I_7	5	15	17	18	FADD.D	P6	P5	P2

B) For this part, consider a more realistic system with a four-entry ROB. An ROB entry can be used one cycle after the instruction using it commits. Fill in the table as you did in part A. If the instruction uses a source register that has already been retired, use the architectural name of the register. (10 points) B) 对于这部分，考虑一个带有四入口 ROB 的更现实的系统。 提交 ROB 条目的指令后，可以在一个周期内使用它。 按照您在 A 部分中所做的那样填写表格。如果指令使用已停用的源寄存器，请使用寄存器的架构名称.（10 分）四入口也就是四个名字

	Time				OP	Dest	Src1	Src2
	Enter ROB	Issue	WB	Commit				
I_1	-1	0	2	3	FLD	p0	x1	-
I_2	0	3	9	10	FMUL.D	p1	p0	f0
I_3	1	10	12	13	FADD.D	P2	P1	F0
I_4	2	4	6	14	ADDI	P3	X1	-
I_5	4 不能把 f1 重命名为 P4	5	7	15	FLD	P0	P3	-
I_6	11	12	18	19	FMUL.D	P1	P0	P0
I_7	14	19	21	22	FADD.D	P2	P1	F3(不能叫 p2 了因为重复使用了)

Unified Physical Register Files

Q7. In this problem, we'll consider an out-of-order CPU design using a unified physical register file. All of the data, both retired and inflight, are kept in the same physical register file. The pipeline contains a remap file that is indexed by the architectural register number and stores the physical register number the architectural register maps to. The physical register file contains the register data and a bit indicating whether the data is valid or not. The pipeline also contains a free list, which is a FIFO queue containing the physical register numbers that are not yet mapped to architectural registers. On issue, the current mappings of the destination register and two source registers are read from the remap file and stored in the ROB. The head of the free list is then popped off and written to the entry for the destination architectural register in the remap file. On a mispredict or exception, the remap file can be restored by going backwards through the



ROB and restoring the old physical register mappings. Q7。 在这个问题中，我们将考虑使用统一物理寄存器文件的无序 CPU 设计。 both retired and inflight 的数据都保存在同一个物理寄存器文件中。管道包含一个重映射文件，该重映射文件由体系结构寄存器号索引，并存储体系结构寄存器映射到的物理寄存器号。 物理寄存器文件包含寄存器数据和指示数据是否有效的位。 流水线还包含一个 free list，它是一个 FIFO 队列，其中包含尚未映射到架构寄存器的物理寄存器编号。 Issue 时，目标寄存器和两个源寄存器的当前映射从重映射文件中读取并存储在 ROB 中。 然后将 free list 的头部弹出并写入重映射文件中目标架构寄存器的 entry。 如果发生错误预测或异常，可以通过向后遍历 ROB 并恢复旧的物理寄存器映射来恢复重映射文件。

a. Consider a system with eight architectural registers, sixteen physical registers, and a four-entry ROB. The following table shows the ROB when an exception occurs in the instruction indicated in bold 粗体.

	ROB PC	Arch. Register	Old Phys. Register
	0x80001008	x1	p9
tail ->	0x8000101C	x2	p8
head ->	0x80001010	x6	p5
	0x80001014	x2	p11

The left column of the following table shows the state of the remap file when the exception is detected. Fill out the right column to show the restored state. (8 points) 下表的左列显示了检测到异常时重映射文件的状态。 填写右列以显示恢复状态。

Arch Reg	Current State	Restored State
x0	p1	P1
x1	p6	P9 恢复到之前
x2	p2	P11 恢复到之前
x3	p10	P10
x4	p7	P7
x5	p4	P4
x6	p13	P13
x7	p15	P15

别的都不变

A) When can a physical register be released and put back on the free list? (5 points)

在两种情况下，可以将物理寄存器放回空闲列表。 1）当处理异常，或在错误预测的时候撤销指令时，在重新生成重映射文件时，较新的错误推测指令的 dst 物理寄存器将返回到 free list。2）当一条指令提交时，其 dst 架构寄存器的旧物理寄存器可以放回 free list。

b.How many physical registers must there be so that the pipeline never stalls due to lack of physical registers in the free list? (5 points)

物理寄存器的数量应该是架构寄存器的数量加上 ROB entry 的数量。在这种情况下，如果有空闲的 ROB 条目，总会有一个空闲的物理寄存器，因为每个 ROB 条目只能分配 0 或 1 个物理目标寄存器。 流水线会先用光 ROB slots 再用光物理寄存器,或者同时用光.

c. Here are some of the initial register mappings and the free list for a RISC-V OoO CPU with a unified physical register file (integer and floating point regs in same physical register file). (8 points) C）这是带有统一物理寄存器文件（同一物理寄存器文件中的整数和浮点寄存器）的 RISC-V OoO CPU 的一些初始寄存器映射和空闲列表。

Arch Register	Phys Register		Free List
f0	p5		p6
f1	p12		p19
f2	p11		p10
x2	p8		p21
x3	p24		p27
x4	p17		

For the following instruction sequence, indicate which physical register gets assigned as the destination register and which physical register gets added to the free list on commit.

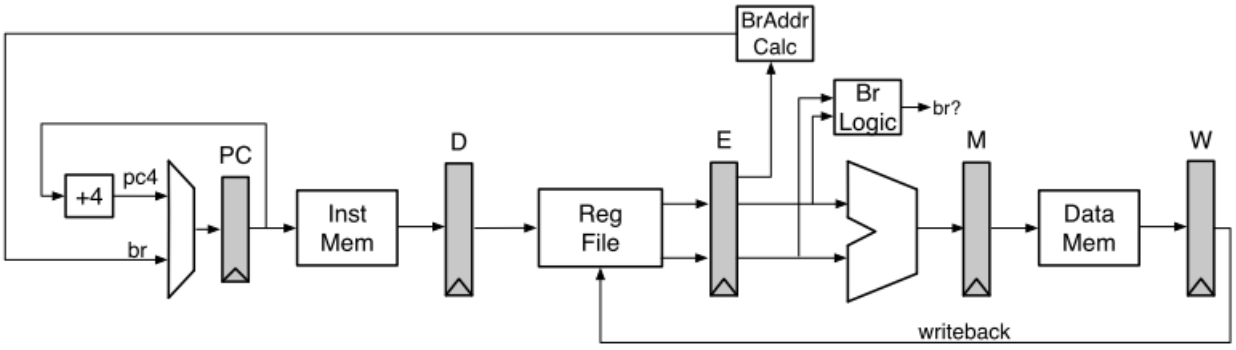
Instruction	Destination Register	Freed Register
`fld f0, 0(x2)`	Pop() 出 P6	释放 P5
`fld f1, 0(x3)`	P19	P12
`fmul.d f0, f0, f2`	P10	(F0 也就是之前用的)P6
`fadd.d f0, f0, f1`	P21	P10 就可以释放了因为后面没用到 f0
`fsd f0, 0(x4)`	-	-
`addi x2, x2, 8`	P27	P8 之前占有的 p8
`addi x3, x3, 8`	P5 就是按队列先进先出	之前占有的 P24
`addi x4, x4, 8`	P12	之前占有的 P17

B) If we wanted to implement register renaming in a superscalar OoO core that can issue two instructions per cycle, what would we have to change? (6 points)

在同一个循环中 issue 的两条指令之间存在危险。 单独重命名它们会导致 assign wrong label。 例如，RAW hazard，第一条指令的目标寄存器需要与第二条指令的读取寄存器相同。我们需要添加一个旁路来实现这一点，因为在第二条指令读取重命名表的同一周期中，第一条指令还没有更新它。 WAW hazard,如果第二条指令写入与第一条指令相同的架构寄存器，则其先前的物理目标寄存器必须是第一条指令使用的物理目标寄存器，而不是重映射表中的当前条目。

Pipelining with Branch Prediction

Q8. For this question, consider a fully bypassed 5-stage RISC-V processor. We have reproduced the pipeline diagram below (bypasses are not shown). Branches are resolved in the Execute Stage, and the Fetch Stage always speculates that the next PC is PC+4. For this problem, we will ignore unconditional jumps, and only concern ourselves with conditional branches.



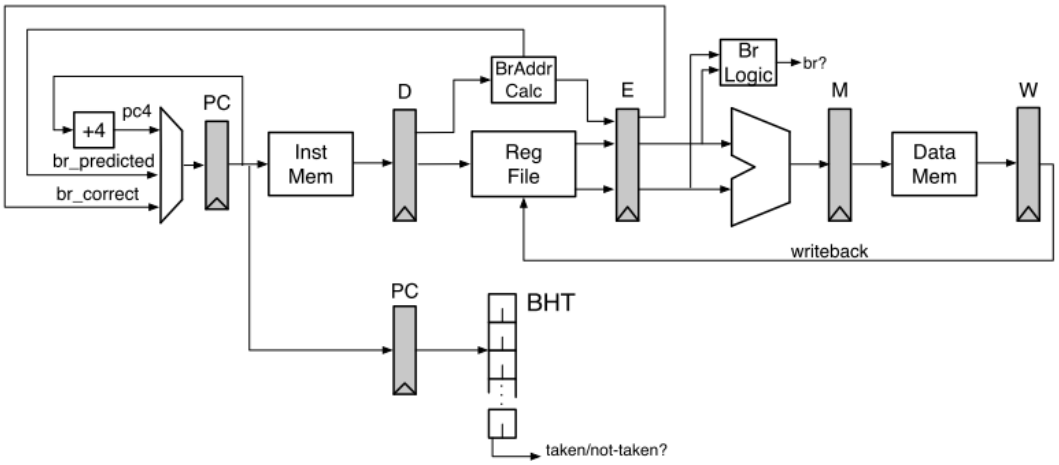
A) Fill in the following pipeline diagram using the code segment below. The first two instructions have been done for you. (5 points)

Throughout this question, make sure you also show instructions that were speculated to be executed and then flushed (it would help to mark them explicitly) in the instruction/time diagrams, as they also consume pipeline resources. 分支比较发生在 t5 的 X。 当发现“错误预测”时，IF_KILL 和 DEC_KILL 信号在 t5 中消失。 bubbles 被插入 pipeline 中，并出现在 t6 上。

```
0x2000: ADDI x4, x0, 1
0x2004: ADDI x5, x0, 1
0x2008: BEQ x4, x5, 0x2004
0x200c: LW   x7, 4(x6)
```

```
0x2010: OR   x5, x7, x5
0x2014: XOR x7, x7, x3
0x2018: AND  x3, x2, x3
```

PC	Instr	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12	t13
0x2000	ADDI	F	D	X	M	W								
0x2004	ADDI		F	D	X	M	W							
0x2008	BEQ			F	D	**X**	**M**	**W**						
0x200c	Lw				F	D	-	-	-					
0x2010	Add					F	-	-	-	-				
0x2004	Addi						F	D	X	M	W			
0x2008	bne							F	D	X	M	W		



b.As you showed in the first parts of this question, branches in RISC-V can be expensive in a 5-stage pipeline. One way to help reduce this branch penalty is to add a Branch History Table (BHT) to the processor. This new proposed data path is shown below:

The BHT has been added in the Decode Stage. The BHT is indexed by the PC register in the Decode Stage. Branch address calculation has been moved to the Decode Stage. This allows the processor to redirect the PC if the BHT predicts "Taken".

On a BHT mis-prediction, (1) the branch comparison logic in the Execute Stage detects mis-predicts, (2) kills the appropriates stages, and (3) starts the Instruction Fetch using the correct branch target (br_correct).

Remember: the Fetch Stage is still predicting PC+4 every cycle, unless corrected by either the BHT in the Decode Stage(br_predicted) or by the branch logic in the Execute Stage(br_correct).

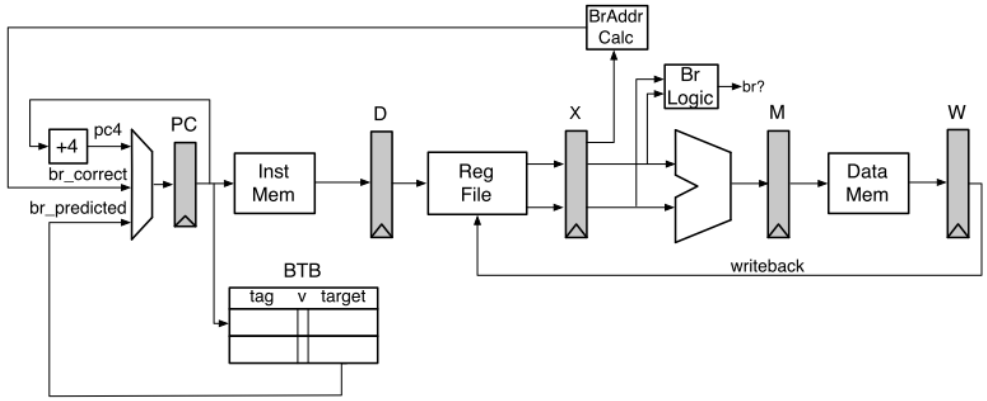
Using the code segment below, fill in the following pipeline diagram. Initially, the BHT counters are all initialized to "strongly-taken". The register x2 is initialized to 0, while the register x3 is initialized to 2. The first instruction has been done for you. It is okay if you do not use the entire table. (6 points) 正如您在本问题的第一部分中所示，RISC-V 中的分支在 5 级流水线中可能代价很高。 帮助减少这种分支惩罚的一种方法是向处理器添加分支历史表 (BHT)。 提出的新数据路径如下所示：BHT 已在解码阶段中添加。 BHT 由解码阶段的 PC 寄存器索引。 分支地址计算已移至解码阶段。 这允许处理器在 BHT 预测"Taken"时重定向 PC。在 BHT 错误预测中，(1) 执行阶段中的分支比较逻辑检测错误预测，(2) 终止适当的阶段，以及 (3) 使用正确的分支目标 (br_correct) 启动指令提取。 请记住：Fetch Stage 仍然在每个周期预测 PC+4，除非被 decode stage 中的 BHT (br_predicted) 或执行阶段中的分支逻辑 (br_correct) 纠正。使用下面的代码段，填写

下面的流水线图。 最初，BHT 计数器都被初始化为“strongly-taken”。 寄存器 x2 初始化为 0，而寄存器 x3 初始化为 2。第一条指令已经为您完成。 如果不使用整个表也没关系。

```
0x2000: LW   x7, 0(x6)
0x2004: ADDI x2, x2, 1
0x2008: BEQ x2, x3, 0x2000
0x200c: SW   x7, 0(x6)
0x2010: OR   x5, x5, 4
0x2014: OR   x7, x7, 5
```

Loop is not taken。但是 BHT strongly-taken，因此当我们看到 BEQ 时，BHT 预测“taken”。 BHT 处于 Decode 阶段，Fetch 阶段总是预测 PC+4,但是 branch 实际上没有 taken（只是分解为 2 周期分支惩罚）。 在 t4，Decode 阶段终止 fetch 阶段以将其重定向到“BEQ taken”路径。 然而，在 t5，Execute 中的分支比较必须纠正 BHT 的错误预测，并终止 Fetch 和 Decode。

PC	Instr	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12	t13
0x2000	LW	F	D	X	M	W								
0x2004	ADDI		F	D	X	M	W							
0x2008	BEQ			F 依然预测+4	**D**	**X**	**M**	**W**						
0x200c	SW				F	-	-	-	-					
0x2000	Lw(其实不跳转)					F	-	-	-	-	-			
0x200c	SW						F	D	X	M	W			
0x2010	Or							F	D	X	M	W		
0x2014	Or								F	D	X	M	W	



C. Unfortunately, while the BHT is an improvement, we still have to wait until we know the branch address to act on the BHT’s prediction. We can solve this by using a two-entry Branch Target Buffer (BTB). The new pipeline is shown below. For this question, we have removed the BHT and will only be using the BTB.

The BTB has been added in the Fetch Stage. The BTB is indexed by the PC register in the Fetch Stage. Branch address calculation has been moved back to the Execute Stage. 不幸的是，虽然 BHT 是一种改进，但我们仍然需要等到知道分支地址才能根据 BHT 的预测采取行动。 我们可以通过

使用 2entry 的分支目标缓冲区（BTB）来解决这个问题。 新 pipeline 如下所示。 对于这个问题，我们已经删除了 BHT，将只使用 BTB。BTB 已添加到 Fetch 阶段。 BTB 由 Fetch Stage 中的 PC 寄存器索引。 分支地址计算已移回到执行阶段。

On a branch mis-prediction, (1) the branch comparison logic in the Execute Stage detects the mis-predict, (2) kills the appropriates stages, and (3)starts the Instruction Fetch using the correct branch target (br_correct).

Remember: The Fetch Stage is still predicting PC+4 every cycle, unless either the BTB makes a prediction (has a matching and valid entry for the current PC) or the branch logic in the Execute Stage corrects for a branch mis-prediction (br_correct).

Using the code segment below (the exact same code from 4.B), fill in the following pipeline diagram. Upon entrance to this code segment, the register x2 is initialized to 0, while the register x3 is initialized to 2. 对于分支错误预测，（1）执行阶段中的分支比较逻辑检测错误预测，（2）终止适当的阶段，以及（3）使用正确的分支目标（br_correct）启动指令提取。 切记：除非 BTB 做出预测（具有与当前 PC 匹配且有效的条目），或者执行阶段中的分支逻辑纠正了分支错误预测，否则提取阶段仍会在每个周期预测 PC + 4。 ）。使用下面的代码段（与 4.B 中的代码完全相同），填写下面的流水线图。 在进入该代码段时，寄存器 x2 被初始化为 0，而寄存器 x3 被初始化为 2。

```
0x2000: LW   x7, 0(x6)
0x2004: ADDI x2, x2, 1
0x2008: BEQ  x2, x3, 0x2000
0x200c: SW   x7, 0(x6)
0x2010: OR   x5, x5, 4
0x2014: OR   x7, x7, 5
```

Initially, the BTB contains:

Tag	V	Target PC
0x2008	1	0x2000
0x201c	0	0x2010

(For simplicity, the Tag is 32-bits, and we match the entire 32-bit PC register in the Decode Stage to verify a match). It is okay if you do not use the entire instruction/time table. (6 points)

PC	Instr	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12	t13
0x2000	LW	F	D	X	M	W								
0x2004	ADDI		F	D	X	M	W							
0x2008	BEQ			F	D	X	M	W						
0x2000	Lw				F	D	-	-	-					
0x2004	Addi 会进入两条错误指令					F	-	-	-	-				
0x200c	Sw						F	D	X	M	W			
0x2010	Or							F	D	X	M	W		
0x2014	or								F	D	X	M	W	

当 BEQ 指令处于取指阶段时，BTB 会找到一个用 PC 标记的有效条目，也就是 v=1 的 target pc 0x2000，因此它错误地预测 br taken 并取 LW 指令。 直到两个周期后，当 BEQ 指令到达执行阶段时，才会检测到错误预测。 所以两条指令将错误地进入 pipeline，必须清除。

Load/Store Speculation

Q9. A.Suppose we want to execute stores out-of-order. Could there be an issue if we allow stores to write to the cache even when there are uncommitted instructions before them in program order? (5 points)

Yes，executing stores before other instructions and write cache before commit may cause some issues.

1.不精确的异常：如果在 store 之前的一条指令有异常，必须 rollback 架构状态。 如果已经修改了缓存内容，这很难做到。 如果 store 是错误预测的分支的结果，情况也是如此。

2. 内存排序：如果在完成 store 之前有 lw 或 sw 指令到相同地址，则在完成时写入缓存将改变程序的行为，和按顺序执行时不同。

B Suppose we bypass load values from a speculative store buffer. If the load address hits in both the store buffer and the cache, which one should we use: the data forwarded from the store buffer or the data from the cache? (5 points)

如果存储缓冲区 hit 对应于 lw 之前的存储指令，我们应该获取 store buffer 数据. 如果存储指令按程序顺序在加载指令之后，我们应该取 cache 数据

c.Suppose that we want loads and stores to execute out-of-order with respect to each other. Under what circumstances in the code below can we execute instruction 5 before executing any others? (5 points)

1. `add x1, x1, x2`
2. `sw  x5, (x2)`
3. `lw  x6, (x8)`
4. `sw  x5, (x6)`
5. `lw  x9, (x3)`
6. `add x9, x9, x9`

instruction 5 not overlap the addresses being stored in instructions 2 and 4. |x2- x3| >= 4, |x6- x3| >= 4.

Under what circumstances can we execute instruction 4 in the code above before executing any others? (5 points)

指令 4 不能先于指令 3,因为存在 RAW 风险，因为它取决于指令 3 正在加载的寄存器 x6 的值。 它不依赖于任何其他指令，因为 store 在提交之前不会更改架构状态，因此它可以在指令 3 完成后立即执行。

Now lets assume that we execute instruction 5 before all other instructions, but instruction 5 causes an exception (e.g., page fault). We want to provide precise exceptions in this processor. What happens with instructions 1, 2, 3, 4, and 6 before execution switches to the OS handler? What should happen if instructions, 1, 2, 3, or 4 also raise an exception? (5 points)

为了提供精确的异常，在切换到操作系统处理程序之前，我们必须在指令 5 之前执行并提交所有指令。 指令 6 必须被终止（因此不会提交），因为它在 5 之后。如果指令 1、2、3 或 4 也引发异常，则应优先考虑最早出现异常的指令，并且之后的指令应该从流水线中 flush 掉。

How can we always be able to execute loads and stores out of order before their addresses are known? What is the downside and how is it handled? Specifically, assume that we executed instruction 5 before instruction 4, but then realized that |x6 – x3| < 4. (5 points)

我们可以推测性地假设所有加载和存储的地址是不重叠的，并在知道它们的地址之前 issue 它们。一旦 realize 地址，如果我们意识到我们不应该对某些加载和存储重新排序，我们必须终止我们不应该执行的那些以及依赖于它们的任何进一步指令。For example, if we realized |x6- x3| <4，we have to terminate instruction 5 and 6. Once instruction 4 completes, we then re-execute instructions 5 and 6.

Branch Predictor Accuracy

Q10. For this problem, we are interested in the following code:

```
int array[N] = {…};
for (int i = 0; i < N; i++)
    if (array[i] != 0)
        array[i]++;
```

Using the disassembler, we get:

```
    li a0, N
    la a1, array
loop:
    lw a2, 0(a1)
    beqz a2, endif
    addi a2, a2, 1
    sw a2, 0(a1)
endif:
    addi a0, a0, -1
    addi a1, a1, 4
    bnez a0, loop
```

1 Full BHT

The processor that this code runs on uses a 512-entry branch history table (BHT), indexed by PC [10:2]. Each entry in the BHT contains a 2-bit counter, initialized to the 00 state.

Each 2-bit counter works as follows: the state of the 2-bit counter decides whether the branch is predicted taken or not taken, as shown in the table below. If the branch is actually taken, the counter is incremented (e.g., state 00 becomes state 01). If the branch is not taken, the counter is decremented. The counter saturates at 00 and 11 (a not-taken branch while in the 00 state keeps the 2-bit counter in the 00 state).

运行此代码的处理器使用 512 个条目的分支历史表 (BHT)，由 PC [10:2] 索引。 BHT 中的每个条目都包含一个 2 位计数器，初始化为 00 状态。每个 2 位计数器的工作原理如下：2

位计数器的状态决定分支是否被预测采用，如下表所示。 如果分支实际被采用，则计数器递增（例如，状态 00 变为状态 01）。 如果没有采用分支，则计数器递减。 计数器在 00 和 11 处饱和（处于 00 状态的未取分支使 2 位计数器保持处于 00 状态）。

State	Prediction
00	Not taken
01	Not taken
10	Taken
11	Taken

If array = {1,0,-3,2,1}, what is the prediction accuracy for the two branches found in the above code for five iterations of the loop, using the 512-entry BHT described above? (10 points)

Loop branch the prediction accuracy is 2/5

State	Prediction	Fact
00	Not taken	Taken
01	Not taken	Taken
10 进阶	Taken	Taken
11 11 饱和	Taken taken	taken Not taken 循环结束

If branch 4/5

State	Prediction	Fact
00	Not taken	Not taken
00	Not taken	Taken 为 0
01	Not taken	Not taken
00 00	Not taken Not taken	Not taken Not taken

Small BHT

Now consider a BHT with only a single entry. That is, both branches will share the same counter. Now what will the prediction accuracy be for each branch? Assume we are using the same array, {1,0,-3,2,1}. (10 points) 预测精确度下降了.

Loop 3/5

If branch 1/5

Branch	State	Prediction	Fact
If0	00	Not taken	Not taken1
Loop0	00	Not taken	Taken1

If1	01	Not taken	Not taken0
Loop1	10	Not taken	Not taken0
If2	11	Not taken	Not taken -3
Loop2	10	taken	Taken -3
If3	11	taken	Not taken2
Loop3	10	taken	Taken2
If4	11	taken	Not taken1
Loop4	10	taken	Not taken1

Static Hints

For this question, assume that the compiler can specify statically which way the processor should predict the branch will go. If the processor sees a "branch-likely" hint from the compiler, it predicts the branch is taken and does NOT update the BHT with this branch (i.e., any branches the compiler can analyze do not pollute the BHT).

Which branches in the program, if any, can the compiler provides hints for? Assume the input array for the compiler's test runs varies widely and the compiler must be fairly confident in the accuracy of a static branch hint. (5 points)

The compiler could statically hint that the loop branch will be taken, since we know it will be taken more often than not. 我们不应该为 if 分支添加静态提示，因为这个分支依赖于数据。