1、cache 一致性协议：考虑一个由 3 个处理器构成的基于总线的共享存储器系统。该共享存储器被分为 x,y,z,w 四块。每个处理器有一个高速缓存，且在任何给定时间只能安装一块。每一块可能处于两个状态之一：有效（V）或无效（I)。假定高速缓存开始时已被清空，且存储器的内容如下：

存储器块	x	y	z	w
内容	20	30	50	20

考虑遵循以下给定顺序的存储器访问事件序列：

1) P1：Read(x) ; 2) P2: Read(x) 3) P3: Read(x) 4) P1: x=x+25 ;
5) P1: Read(z); 6) P2: Read(x) 7) P3: x=15 8) P1: z=z+10

高速缓存的采用的协议为： 写回法和写无效协议。说明在上述的每一次操作后高速缓存和存储器的内容以及高速缓存块的状态。 （15 分）

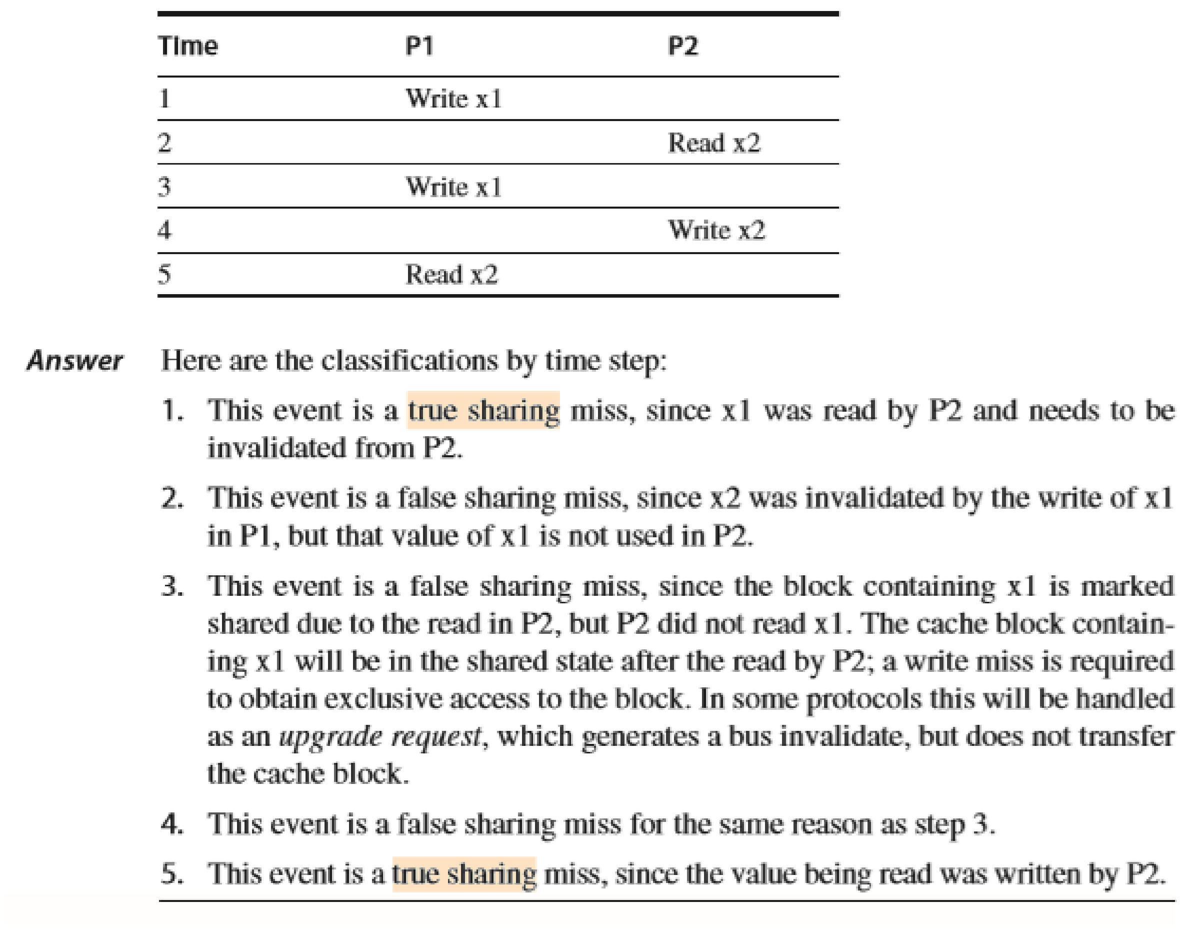
事件	P1 的高速缓存	P2 的高速缓存	P3 的高速缓存	主存			
				x	y	z	w
初始	(I) 清空	(I) 清空	(I) 清空	20	30	50	20
P1:Read(x)	(V) 20	(I) 清空	(I) 清空	20	30	50	20
P2: Read(x)	(V) 20	(V) 20	(I) 清空	20	30	50	20
P3: Read(x)	(V) 20	(V) 20	(V) 20	20	30	50	20
P1: x=x+25	（V) 45	(I)	(I)	20	30	50	20
P1: Read(z)	(v) 50	(I)	(I)	45	30	50	20
P2: Read(x)	(v) 50	(v) 45	(I)	45	30	50	20
P3: x=15	(v) 50	(I)	(v) 15	45	30	50	20
P1: z=z+10	(v) 60	(I)	(v) 15	45	30	50	20

2、 coherence misses, true sharing misses 和 false sharing miss 的区别是什么？
选自： Computer Architecture A Quantitative Approach， Fifth Edition， 5.3 Performance of Symmetric Shared-Memory Multiprocessors，英文版 P367

The misses that arise from inter-processor communication, which are often called coherence misses, can be broken into two separate sources.
The first source is the so-called true sharing misses that arise from the communication of data through the cache coherence mechanism. In an invalidation-based protocol, the first write by a processor to a shared cache block causes an invalidation to establish ownership of that block. Additionally, when another processor attempts to read a modified word in that cache block, a miss occurs and the resultant block is transferred. Both these misses are classified as true sharing misses since they directly arise from the sharing of data among processors.

The second effect, called false sharing, arises from the use of an invalidation based coherence algorithm with a single valid bit per cache block. False sharing occurs when a block is invalidated (and a subsequent reference causes a miss) because some word in the block, other than the one being read, is written into. If the word written into is actually used by the processor that received the invalidate, then the reference was a true sharing reference and would have caused a miss independent of the block size. If, however, the word being written and the word read are different and the invalidation does not cause a new value to be communicated, but only causes an extra cache miss, then it is a false sharing miss. In a false sharing miss, the block is shared, but no word in the cache is actually shared, and the miss would not occur if the block size were a single word. The following example makes the sharing patterns clear.

Example :Assume that words x1 and x2 are in the same cache block, which is in the shared state in the caches of both P1 and P2. Assuming the following sequence of events, identify each miss as a true sharing miss, a false sharing miss, or a hit.



Time	P1	P2
1	Write x1	
2		Read x2
3	Write x1	
4		Write x2
5	Read x2	

Answer Here are the classifications by time step:

1. This event is a true sharing miss, since x1 was read by P2 and needs to be invalidated from P2.
2. This event is a false sharing miss, since x2 was invalidated by the write of x1 in P1, but that value of x1 is not used in P2.
3. This event is a false sharing miss, since the block containing x1 is marked shared due to the read in P2, but P2 did not read x1. The cache block containing x1 will be in the shared state after the read by P2; a write miss is required to obtain exclusive access to the block. In some protocols this will be handled as an *upgrade request*, which generates a bus invalidate, but does not transfer the cache block.
4. This event is a false sharing miss for the same reason as step 3.
5. This event is a true sharing miss, since the value being read was written by P2.

3. Sequential Consistency (SC) model.

SC 其规定：
（1）每个线程内部的指令都是按照程序规定的顺序（program order）执行的（单个线程视角）
（2）线程执行的交错顺序可以是任意的，但是所有线程所看见的整个程序的总体执行顺序都是一样的（整个程序的视角）

These are three small programs, each executed on a different processor, each with its own cache and register set. In the following **R** is a register and **X** is a memory location. Each instruction has been named (e.g., B3) to make it easy to write answers. Assume data in location X is initially 0. Assuming the program is executing under the Sequential Consistency (SC) model.

Processor A	Processor B	Processor C
A1: ST X, 1	B1: R := LD X	C1: ST X, 6
A2: R := LD X	B2: R := ADD R, 1	C2: R := LD X
A3: R := ADD R, R	B3: ST X, R	C3: R := ADD R, R
A4: ST X, R	B4: R:= LD X	C4: ST X, R
	B5: R := ADD R, R	
	B6: ST X, R	

A. Can X hold value of 4 after all three threads have completed? Please explain briefly.
Yes, B1-B3， C1-C4， A1-A4， B4-B6 （订正）

B. Can X hold value of 5 after all three threads have completed?
No. All results must be even!

C. Can X hold value of 6 after all three threads have completed?
Yes， All of C, All of A, All of B， （订正）
Yes, A1-A4, B1-B4, C1-C4, B5, B6 （订正）

D. For this particular program, can a processor that reorders instructions but follows local dependencies produce an answer that cannot be produced under the SC model?

No. All stores/loads must be done in order because they're to the same address, so no new results are possible.

6. This problem evaluates the cache performances for different loop orderings. You are asked to consider the following two loops written in C, which calculate the sum of the entries in a 128 by 64 matrix of 32-bit integers:

Loop A	***Loop B***
`sum = 0;`	`sum = 0;`
`for (i = 0; i < 128; i++)`	`for (j = 0; j < 64; j++)`
`  for (j = 0; j < 64; j++)`	`  for (i = 0; i < 128; i++)`
`    sum += A[i][j];`	`    sum += A[i][j];`

The matrix A is stored contiguously in memory in row-major order. Row major order means that elements in the same row of the matrix are adjacent in memory as shown in the following memory layout: A[i][j] resides in memory location [4* (64 * i + j)]. The memory location is as the following Figure shows.

0	4		252	256		4*(64*127+63)
A[0][0]	A[0][1]	...	A[0][63]	A[1][0]	...	A[127][63]

For Problem A to Problem C, assume that the caches are initially empty. Also, assume that only accesses to matrix A cause memory references and all other necessary variables are stored in registers. Instructions are in a separate instruction cache.

Problem A

- Consider a 4 KB direct-mapped data cache with 8-word (32-byte) cache lines.
- Calculate the number of cache misses that will occur when running Loop A.
- Calculate the number of cache misses that will occur when running Loop B.

We can calculate that there're 4*1024/32 = 128 lines, each of which can contain just 8 integers. Since the matrix has 128 lines, the number of cache misses of Loop A and Loop B are as follows:
Loop A: 128 *64/8 = 1024 misses.
Loop B: 128*64 = 8192 misses.

Problem B
Consider a direct-mapped data cache with 8-word (32-byte) cache lines.

- Calculate the minimum number of cache lines required for the data cache if Loop A is to run without any cache misses other than compulsory misses.
- Calculate the minimum number of cache lines required for the data cache if Loop B is to run without any cache misses other than compulsory misses.

The minimum number of cache lines required for Loop A is 1. （订正）
The minimum number of cache lines required for Loop A is 1024

Problem C

Consider a 4 KB fully-associative data cache with 8-word (32-byte) cache lines. This data cache uses a first-in/ first-out (FIFO) replacement policy.

- Calculate the number of cache misses that will occur when running Loop A.
- Calculate the number of cache misses that will occur when running Loop B.
- Will a larger cache size help to reduce the miss rate? Why or why not?
- Will a larger block size help to reduce the miss rate? Why or why not?

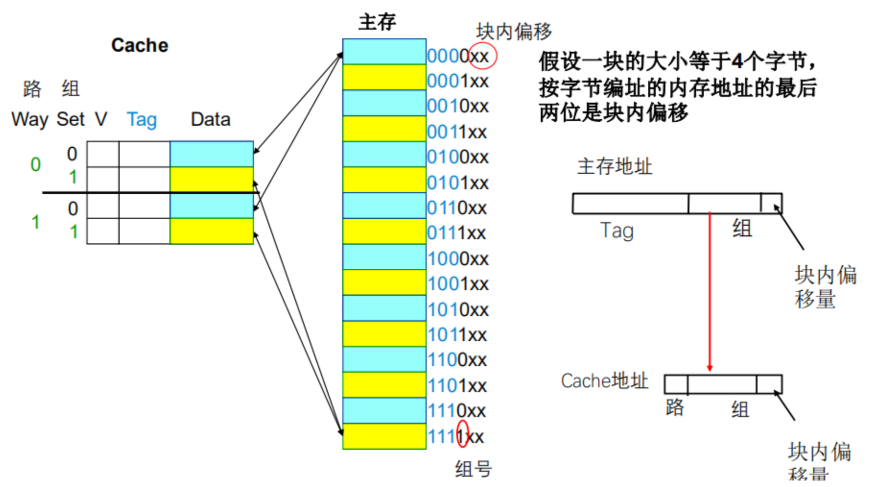
1024
1024
No. We assume that the block size doesn't change. A larger cache size means that there will be more than 128 cache lines, but the miss rate will be still 1/8. Let's consider the special case, which the number of line is 1024, all the cache misses are compulsory miss. There is one miss of every block (8-word), thus the miss rate is still 1/8.
No for Loop B. We only consider the situation that the cache size doesn't change. If the block size gets larger, the number of lines will be less than 128, and for Loop B, this will lead to that every memory access will miss, but for Loop A, the miss rate can be reduced.

Let us assume that the cache contains 2048 words, 16 words per line(block), and 128 lines. If I have a Main-Memory word address(字地址) of [11 0011 0011 0011 0011], what line is the data fetched into with each cache organization? (The answer has to be between 0 and127).

(i) Direct
(ii) Fully associative
(iii) Set associative (with 64 sets)

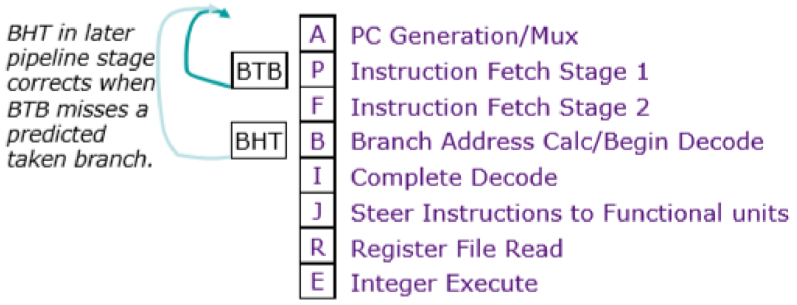
irect: $(011\ 0011)_2=51$th line
Fully associative: Possibly into any line
Set associative: 128/64 = 2 lines per set. $(11\ 0011)_2=51$th set, thus maybe 102th or 103th line.
由于 cache 地址映射，不同的机器可以有不同的实现，如果如下图这样的地址映射，答案 102th or 103th line 是不合理的。所以考试时我们不会出现这样的问题。

组相联映射（分组方式2）



To improve the branch performance further, we decide to add a branch target buffer (BTB) as well. Here is a description for the operation of the BTB.

1. The BTB holds entry_PC, target_PC pairs for jumps and branches predicted to be taken. Assume that the target_PC predicted by the BTB is always correct for this question. (Yet the direction still might be wrong.)
2. The BTB is looked up every cycle. If there is a match with the current PC, PC is redirected to the target_PC predicted by the BTB (unless PC is redirected by an older instruction); if not, it is set to PC+4.



BHT in later pipeline stage corrects when BTB misses a predicted taken branch.
BTB
BHT
A PC Generation/Mux
P Instruction Fetch Stage 1
F Instruction Fetch Stage 2
B Branch Address Calc/Begin Decode
I Complete Decode
J Steer Instructions to Functional units
R Register File Read
E Integer Execute

	BTB Hit?	(BHT) Predicted Taken?	Actually Taken?	Pipeline bubbles
Conditional Branches	Y	Y	Y	
	Y	Y	N	
	Y	N	Y	Cannot occur
	Y	N	N	Cannot occur
	N	Y	Y	
	N	Y	N	
	N	N	Y	
	N	N	N	

提问：　圈出来的地方为什么不能 occur?

回答：BTB hit 就是说明预测转移，所以不可能出现 BHT Predicted Not Taken.

考虑使用 Tomasulo’s 算法来实现动态调度。处理器的数据通路如下图所示。处理器中各个功能单元：　load/store 单元、FP ADD 单元、FP Mult 单元、Integer/Branch 操作单元都只有一个；　各个功能单元（function units）的工作延迟为：　load/store 单元 2 周期，FP ADD 单元 3 周期，　FP Mult 单元 5 周期，Integer/Branch 单元 1 周期。所有功能单元都是流水化的。如果一个操作数在某一个周期写回，依赖于该操作数的指令可以在下一周期开始执行

A. **填充下面的表格，**显示在 Tomasulo’s 算法动态调度下，各条指令在各个阶段的周期数。假设从第 0 周期开始。

在这个小题里，我们假设所有功能单元前端对应的保留站(reservation station)都只有一个表项。因为进入保留站的指令的结果操作数是用保留站号来标记的，所以当保留站中有一条指令，如果该指令已经开始执行，不能将该保留站清除并重新分配给新发射的指令。例如：load/store　指令会因为缺少保留站引发 structural hazard（结构冲突）导致的停顿。引起了下表中第二条 load 指令的发射必须延迟到第 4 周期才能开始。

Solution I	Issue	Execute	WB
Code	Issue	Execute	WB
L.D F2, 0(R1)	0	1-2	3
MUL.D F4, F2, F0	1	4-8	9
L.D F6, 0(R2)	4	5-6	7
ADD.D F6, F4, F6	5	10-12	13
S.D F6, 0(R2)	8	14-15	
DADDUI R1, R1, #8	9	10	11
DADDIU R2, R2, #-8	12	13	14
BGT R1, #800	15	16	

B. 接下来，假设数据通路中有一个 4 个 entry(表项）的 ROB（Reorder Buffer: 重排序缓冲器），如下图所示。当 问题 A 中的 ADD.D 指令已经准备好提交（但未提交）时，填写 ROB 中这 4 个 entry 的状态。表中 Complete? 这一栏表示该指令是否已经执行。

Entry	Instruction	Destination	Complete?
1	S.D F6, 0(R2)	M[R2 + 0]	NO
2	DADDUI R1, R1, #8	R1	YES
3	DADDIU R2, R2, #-8	R2	YES
4	ADD.D F6, F4, F6	F6	YES

提问：为什么它们在 ROB 里这么放置？

回答： ROB 的实现是一个环形缓冲区