1. 在多发射乱序执行的处理器上，编译器的调度还需要吗？请举例论证你的观点。

解：编译器的调度还是需要，通常是**采用编译器的静态调度和硬件的动态调度相结合**来提高流水线的效率。静态指令调度是在还不知道程序某些动态信息和行为的情况下，根据所分析的指令之间依赖关系以及目标机的资源状况，对指令序列进行重排，从而减少流水线停顿。因为**动态调度只是在某个指令窗口中进行调度**，例如是 64 个指令窗口中选择指令进行调度和执行。而**编译器可以在更大的指令窗口进行调度**，例如在程序块或者块之间等进行调度。典型的例子有：1）编译器进行的循环展开，消除控制相关和增加可调度的指令数目。2）延迟槽指令，编译器可以把有效的指令放入到延迟槽进行执行。

2. 请给出一种在多发射动态调度的处理器上解决访存相关的方案。

解：1）使用 store buffer 来作为一个临时性位置存放写操作的值。在 store 指令提交的时候才写入到 cache 中。

2)store 指令按程序的顺序写回到 cache 中。任何访问相同地址的 load 可以从之前的 store 指令中获得值，即 load forwarding 技术。

3）两种方法：一是非推测的方法，后续的 load 指令不能超过前面的 store 指令，通常采用 load forwarding 技术；二是推测的方法，speculative load execution。例如一条 store 指令后面有一条 load 指令，而 store 指令的访存地址还没有计算出来，很可能 load 的地址就是 store 的指令的地址，因此可能存在着 RAW 相关。推测的方法 speculative load-store reordering 是一种推测执行的技术，就像分支预测之后的推测执行一样，可以把阻塞的 store 指令之后的 load 指令提前执行，等 store 指令访存地址计算之后，和其后续的 load 指令的访存地址进行比较，如果访存地址有交叉，则需要取消该 load 指令，以及和它相关的后续指令，重新开始执行，和分支误预测一样处理。

3. 以下有 4 段 MIPS 代码片段，每段包含两条指令：

①	DADDI R2, R2, 2 LD R2, 4(R2)
②	DSUB R3, R1, R2 SD R2, 7(R1)
③	S.D F2, 7(R1) S.D F2, 200(R7)
④	BLE R2, place SD R2, 7(R2)

解：1）

①中存在 RAW 相关和 WAW 相关；

②中没有相关；

③中可能存在访存相关；

④中存在控制相关。（假设没有延迟槽）

通过重命名机制可以消除 WAW 和 WAR 相关。通过延迟槽技术或者可以消除控制相关。

2）①不可以同时发射，②③④可以同时发射。

（此处发射按照课本上的描述，指从保留站发射去执行）

（如果按照 PPT 上的描述，发射为进入保留站之前的步骤，那么 123 可以同时发射，当没有延迟槽和分支预测时，4 不能）

4. 解：

(1) 循环展开 2 次

L:	L.D	F0, 0(R1)	; load X[i]
	L.D	F6, -8(R1)	; load X[i-1]
	MUL.D	F0, F0, F2	; a*X[i]
	MUL.D	F6, F6, F2	; a*X[i-1]
	L.D	F4, 0(R2)	; load Y[i]
	L.D	F8, -8(R2)	; load Y[i-1]
	ADD.D	F0, F0, F4	; a*X[i] + Y[i]
	ADD.D	F6, F6, F8	; a*X[i-1]+ Y[i-1]
	DSUBUI	R2, R2, 16	
	DSUBUI	R1, R1, 16	
	S.D	F0, 16(R2)	; store Y[i]
	S.D	F6, 8(R2)	; store Y[i-1]
	BNEZ	R1, L	

计算一个元素需 14/2=7 拍。

(2) 假设双发射流水线中有彼此独立的一条定点流水线和一个浮点流水线。

	定点指令线		浮点指令线	
L:	L.D	F0, 0(R1)		
	L.D	F6, -8(R1)		
	L.D	F10, -16(R1)	MUL.D	F0, F0, F2
	L.D	F14, -24(R1)	MUL.D	F6, F6, F2

	L.D	F4, 0(R2)	MUL.D	F10, F10, F2
	L.D	F8, -8(R2)	MUL.D	F14, F14, F2
	L.D	F12, -16(R2)	ADD.D	F0, F0, F4
	L.D	F16, -24(R2)	ADD.D	F6, F6, F8
	DSUBUI	R2, R2, 32	ADD.D	F10, F10, F12
	DSUBUI	R1, R1, 32	ADD.D	F14, F14, F16
	S.D	F0, 32(R2)		
	S.D	F6, 24(R2)		
	S.D	F10, 16(R2)		
	S.D	F14, 8(R2)		
	BNEZ	R1, L		

计算每个元素需要 16/4=4 拍

(3)

```
int main()
{
    // initialize the X[i] and Y[i];
    int i;
    for (i=100; i>=0; i--)
        Y[i] = a*X[i] + Y[i];
    return 0;
}
```

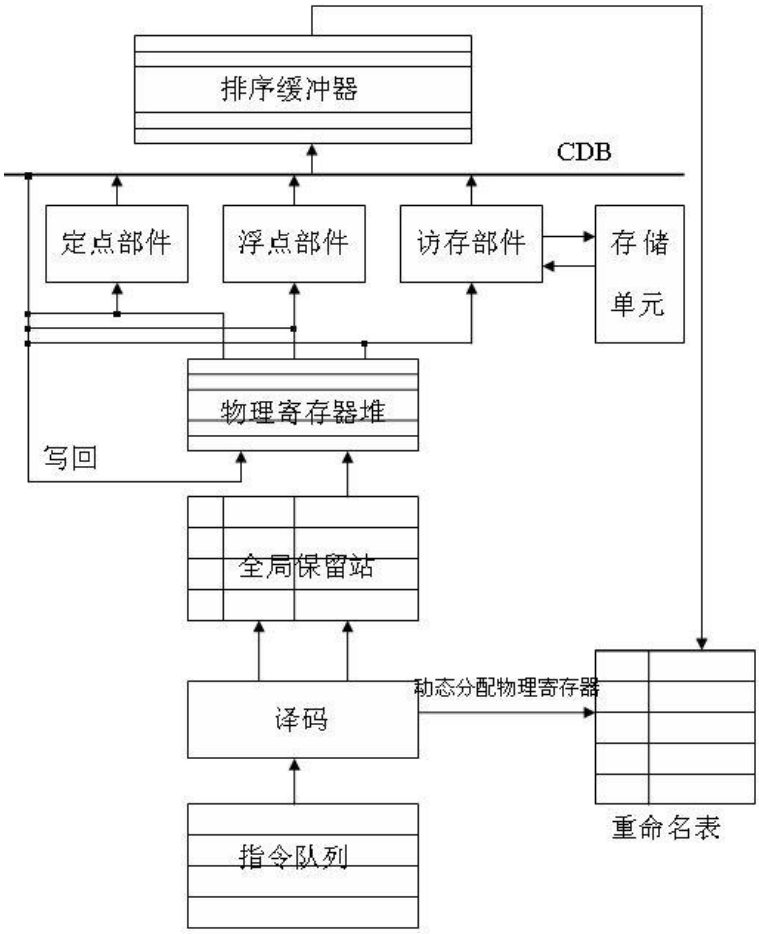
gcc -O 不进行优化， -O1, -O2, 进行部分优化，但是不进行循环展开优化；-O3 进行所有的优化，包括循环展开，对该程序非常有效。

5. 解:

物理寄存器个数应至少为 n×t2+m 才能让流水线满负荷工作。

每条指令在重命名的时候需要占用一个物理寄存器，每拍 n 条指令发射，则需要占用 n 个物理寄存器，被重命名的物理寄存器只有在提交的时候，并且不是当前逻辑寄存器的时候，才被释放。从重命名到提交共有 t2 拍，当流水线满负荷工作，流水线占用了 n*t2 个物理寄存器，另外逻辑寄存器数目为 m 个，因此共需要物理寄存器的个数为 n*t2+m。

6. 解：



用 ROB 实现顺序提交

流水段的相应操作是：

（1） 取指阶段：从指令部件中取指令到指令队列中。

（2） 译码阶段：对取到的指令进行译码，并为逻辑寄存器分配空闲（state 状态为 EMPTY）的物理寄存器，并将这个逻辑寄存器号填入映射表中该物理寄存器所对应的表项，同时置该表项的 state 为 MAPPED，valid 标志位为 1。每个映射表项有三个域：逻辑寄存器号、state、valid。逻辑寄存器号标示了是哪个逻辑寄存器被映射到这个物理寄存器；State 有 EMPTY/WB/COMMIT/MAPPED 四种选择，用于标示这个物理寄存器的当前状态；valid 有 1、0 两种选择，用于在一个逻辑寄存器对应多个物理寄存器的情况下标示最新映射，当一个逻辑寄存器被使用多次（即被多次分配物理寄存器号），则置最后一次分配的那一项的 valid 为 1，前面都为 0。这样，将要发送给保留站的指令中的寄存器号就是重命名后的物理寄存器号。在这个阶段，还要对这两条指令进行相关检测，如果前面指令的目标寄存器号与后面指令的源寄存器号相同，则要按照第 1 题的方法修改后面指令源寄存器所对应的物理寄存器号。

（3） 发射阶段：将指令发射给保留站（每次发射两条），并判断源操作数是否已经准备好，若准备好了，就读物理寄存器堆，否则就等待所需操作数准备好后再读。这个“准备好”有两层含义：一是所需操作数已经写到物理寄存器中了（通过检查映射表中

的 state 项为 WB 或 COMMIT 来判断)，这时就可以读寄存器堆；二是通过 forwarding 判断所需操作数正在写回，这时也可以继续前进执行，在结果总线上拦截所需操作数（体现在流水线结构图中就是：CDB 总线接回到物理寄存器与定点、浮点、访存部件之间的路径上)。

（4） 执行阶段：定点、浮点部件进行 ALU 运算，访存部件对存储单元进行访问。执行结果写回物理寄存器堆，不必写回保留站。映射表中的相应项的 state 置为 WB 状态。

（5） 提交阶段：排序缓冲器顺序提交指令。修改重命名表中逻辑寄存器与物理寄存器的映射关系。映射表中相应项的 state 置为 COMMIT 状态。如果一个逻辑寄存器被重命名过好几次（即先后被映射到好几个物理寄存器)，则只取最近一次映射为有效，其他映射关系取消，也即把其他映射表项的 state 置为 EMPTY。

7.

Intel 的 Nehalem 基于 Core 架构，其结构是 P6 架构和 Netburst 架构的结合。在 P6 的架构上提高了发射的宽度，采用了激进的访存子系统。其每拍能从 cache 中取 16 个 byte 的指令，能同时译码 4 条 X86 指令，能同时发射 6 条微指令 micro-instruction（发射端口包括 3 条访存操作和 3 条算术逻辑操作)，同时有 4 条微码指令能被提交 commit。其流水线的长度为 14 级，分别是取指和预译码，指令队列（Instruction queue)，译码，译码之后进行重命名以及分配资源（分配 ROB，保留站，访存排序队列)，然后指令被送到集中式的保留站（Scheduler)，然后进入执行单元，执行之后结果送回到保留站或者 ROB 中，ROB 按指令顺序进行提交。和 P6 类似，其采用集中式的保留站（Unified Reservation Station)，保留站的项数为36项。Nehalem流水线具有较大的指令窗口，能同时支持128条指令在执行(in-fly)。Nehalem 的重命名机制采用保留站机制，寄存器重命名到保留站中，指令在流水线中往前走，操作数需要随着指令存放，这样也带来了功耗的问题和空间浪费的问题，后续的 Sandy Bridge 采用了基于物理寄存器堆的方法，指令只需要携带指针。Nehalem 的功能部件基本是全流水的设计，大部分指令通常一拍能完成，并且支持操作数的盘路技术（Forwarding)。

流水线中 30%以上的操作是 load 和 store 操作，如果访存不能供应上计算所需要数，则流水线是无效率的。Nehalem 的访存系统能同时进行一条 128 位的 load 操作和一条 128 位的 store 操作。地址计算完之后访存指令被送到访存排序缓存（MOB，Memory Order Buffer）中，MOB 能支持推测的、乱序的 load 和 store 操作，并且能维护访存操作语义的顺序性和正确性。MOB 中用于访存操作的队列包括 load buffer 和 store buffer，Load buffe 具有 48 项，store buffer 具有 36 项，用于跟踪所有的 load 和 store 操作，访存系统还包括 10 项的 fill buffer，用于 cache 的回填。Nehalem 包括三级 cache 层次，一级 cache 各位 32KB，二级 cache 为 256KB，多个处理器核共享的三级 cache 为 8MB，多层的 cache 层次做到了较好的延迟和容量的均衡。为了供应指令和数据，Nehalem 采用了有效的预取机制，采用了多级和多个预取器。

宽发射深度流水线中分支预测器非常重要，Nehalem 采用了两级分支预测器，并增加了

分支目标缓存 BTB 的容量，以及返回地址栈来预测函数调用和返回。分支预测单元 Branch Prediction Unit（BPU），可以预测三类指令：直接调用或跳转，间接调用或者跳转，条件分支。对间接跳转指令的预测相对其他处理器而言具有明显的优势。

Nehalem 中采用了循环流检测器技术 Loop stream detection（LSD），当发现正执行小的循环时，则关闭分支预测器、预取器和译码器，直接从指令译码队列中获取微码指令序列，而不需要重新取指和经过复杂的 X86 译码器。其类似于 Netburst 中的 Trace cache 的概念，其能给后续的乱序执行部分提供连续的指令流。其能减少 X86 处理器复杂的前端对流水线造成的影响。