第二章 参考答案 计算机系统结构基础

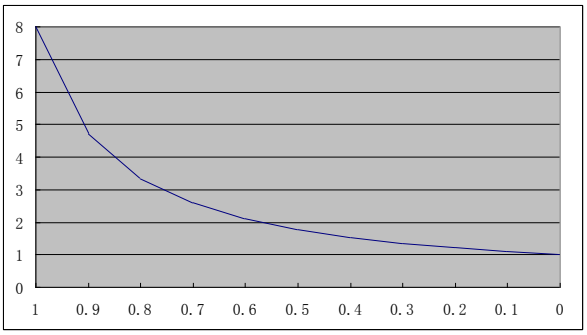
1. 解：A 为 10MIPS，B 为 20MIPS，C 为 40MIPS。

三台机器实际性能相同。

题目问的是运行程序 P 时的性能。

2. 解：加速比 y 与向量化比例 x 之间的关系是：y=1/((1-x)+x/8)=1/(1-7x/8)……(A)

(1) **假设原运行时间为 1，新总运行时间为(1-x) + x/8，得加速比 y**



(2) 在式(A)中令 y=2，可解得 x=4/7≈57.14%。

此时向量模式运行时间占总时间比例是((4/7)/8)/(3/7+((4/7)/8))=1/7=14.29%

(3) 硬件方法，整体加速比为 1/((1-0.7)+0.7/16)=2.91

软件方法，设相同加速比下向量化比例为 x，即 1/((1-x)+x/8)=2.91， x=0.75

所以推荐软件方法。

3. 解：

(1) $MIPS_{EMUL}=(I+F\times Y)/(W\times 10^6)$ ； $MIPS_{FPU}=(I+F)/(B\times 10^6)$ **实际执行总指令除以总时间**

(2) $120=(I+8\times 10^6\times 50)/(4\times 10^6)$ => $I=80\times 10^6$

(3) $80=(80\times 10^6+8\times 10^6)/(B\times 10^6)$ => $B=1.1$

(4) $MFLOPS=F/((B-((W*I)/(I+F*Y)))\times 10^6)\approx 18.46$ **按运行该程序时浮点指令数除以浮点部分所占时间（总时间减去定点部分时间）来计算**

(5) 决策正确，因为执行时间缩短了，这才是关键标准。

这个公式略有一点争议。

4. 解：当取 c=4000 时：

(1) $y=12.29386-0.18295x+0.0015x^2$

(2) $y=342.47443-6.36386x+0.02727x^2$

5. 解：

1.1V 下静态功耗 1.1*1.1/(1.05/0.5)=0.576W **时钟不翻转的静态功耗按电阻算**

1.1V 下 1GHZ 时动态功耗为 1.1*2.5-0.576=2.174W　**动态功耗+静态功耗=总功耗**

1.1V 下 0.5GHZ 动态功耗为 2.174*0.5/1=1.087W　**动态功耗与翻转率成正比**

1.1V 下 0.5GHZ 总功耗为 1.087+0.576=1.663W

6. 解：

a) 先证明 $N=2^k$ 时，正数 $(a_1+a_2+\cdots+a_N)/N \geq \sqrt[N]{a_1a_2\cdots a_N}$。对 k 进行数学归纳法即可。

当 $2^{k-1}<N<2^k$ 时，令 $A=(a_1+a_2+\cdots+a_N)/N$，则 $A=(a_1+a_2+\cdots+a_N+\cdots)/N=(a_1+a_2+\cdots+a_N)/N$。若 $\sqrt[N]{a_1a_2\cdots a_N}>A$，$A\geq\sqrt[2^k]{a_1a_2\cdots a_N A^{(2^k-N)}}>\sqrt[2^k]{A^N A^{(2^k-N)}}=A$，矛盾。因此当当 $2^{k-1}<N<2^k$ 时，$(a_1+a_2+\cdots+a_N)/N \geq \sqrt[N]{a_1a_2\cdots a_N}$。

b) 证：假设参考机的程序分值为 $Z=\{Z_0, Z_1,\cdots,Z_{n-1}\}$，其中 n 为 SPEC CPU2000 中的程序个数；

而 A 机器的程序分值为 $X=\{x_0,x_1,\cdots,x_{n-1}\}$

B 机器的程序分值为 $Y=\{y_0,y_1,\cdots,y_{n-1}\}$

则有：

A 机器的性能为：$\sqrt[n]{\frac{x_0*x_1*\cdots*x_{n-1}}{Z_0*Z_1*\cdots*Z_n}}$，B 机器的性能为：$\sqrt[n]{\frac{y_0*y_1*\cdots*y_{n-1}}{Z_0*Z_1*\cdots*Z_n}}$

从而，A 与 B 机器的性能比为：

$$\frac{\sqrt[n]{\frac{x_0*x_1*\cdots*x_{n-1}}{Z_0*Z_1*\cdots*Z_n}}}{\sqrt[n]{\frac{y_0*y_1*\cdots*y_{n-1}}{Z_0*Z_1*\cdots*Z_n}}}=\sqrt[n]{\frac{x_0*x_1*\cdots*x_{n-1}}{y_0*y_1*\cdots*y_{n-1}}}$$

可见，其结果与参考样机无关。故得证。

7. 解：　**查阅资料题**

AMD 4 核 Barcelona，2.8G，3 发射每个核 1 个 128 位浮点向量功能部件和 1 个 128 位浮点加法向量部件，峰值性能 4*4*2.8=44.8GFlops。 2 路 L1I 64KB；2 路 L1D 64KB 3 latency；16 路 L2 512KB；32 路 2MB 共享 L3，内存带宽 21.34GB/s

Intel 4 核 Nehalem（i7），2.5G-3G，4 发射每个核 1 个 128 位浮点向量功能部件和 1 个 128 位浮点加法向量部件，峰值性能 4*4*3=48GFlops。4 路 L1I 32KB；4 路 L1D 32KB 4 latency；8 路 256KB L2 12 latency；16 路 8MB L3 30-40 latency；内存带宽 31.92GB/s

8. （1）**略。**

第三章 参考答案 二进制与逻辑电路

1. 解：

(1) $[-(2^{63}-1),\ 2^{63}-1]$和$[-2^{63},\ 2^{63}-1]$

(2) -2^{31} **常识题**

2. 解： **常识题**

(1)

0x7ff0000=0,0000 1111,111 1111 0000 0000 0000 0000=$(1.1111111)_2*2^{(15-127)}$=3.836813561083946426009956057493 4e-34

0xbe400000=1,0111 1100,100 0000 0000 0000 0000=$-(1.1)_2*2^{(124-127)}$=-0.1875

0xff800000=1,1111 1111,000 0000 0000 0000 0000=$-\infty$

(2)

0x4035000000000000=0,10000000011,0101000000000000000000000000000000000000 000000000000=$(1.0101)_2*2^{(1027-1023)}$=21

0x8008000000000000 =

1,00000000000,1000000000000000000000000000000000000000000000000000=$-(0.1)_2*2^{-1022}=-2^{-1023}$

(3)

-100.0=$-(1.100100)_2*2^6$=0b1 10000101 10010000000000000000000=0xc2c80000

0.25=$(1.0)*2^{-2}$=0b0 01111101 00000000000000000000000=0x3e800000;

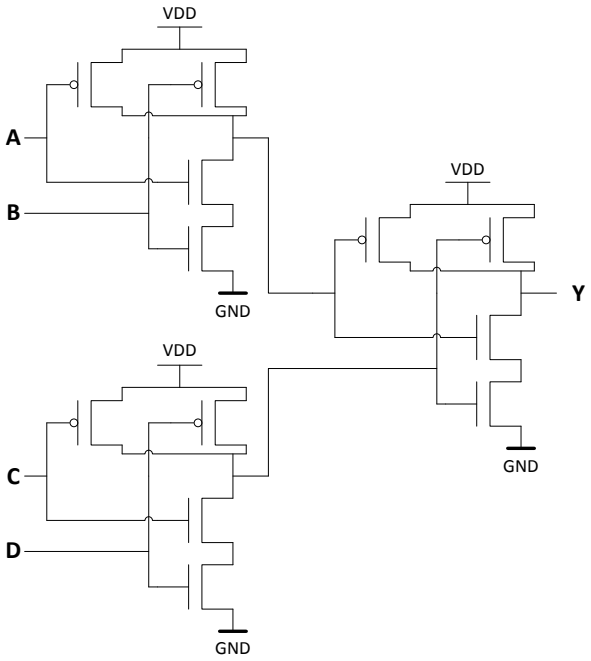
(4)

1024.0=$(1.0)*2^{10}$=0x4090000000000000

0.25=$(1.0)*2^{-2}$=0x3fd0000000000000

3. 解：

A&B|C&D=～(～(A&B)&～(C&D))，两级与非门的逻辑



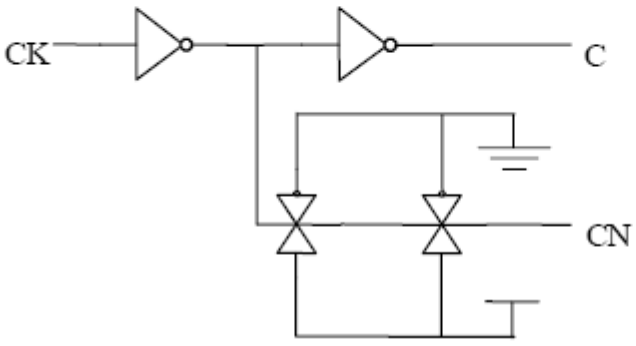
两种解法，第二种，是课本上的~(A&B|C&D)后面再加一个反相器。
N 管要接地，P 管要接电，不能反过来。所以一级逻辑画不出的。注意！

4. 解：

FO4 延迟=本征延迟+负载延迟=0.023+4.5*((0.0036+0.0044)*4)=0.167ns

本课对 FO4 定义为 1 个反相器驱动 4 个相同的反相器（器件延迟和线延迟是 4 份）。鉴于 FO4 定义存疑（参见 wiki），回答 0.023+4.5*(0.0044+0.0036*4*4)=0.302ns 也算对。

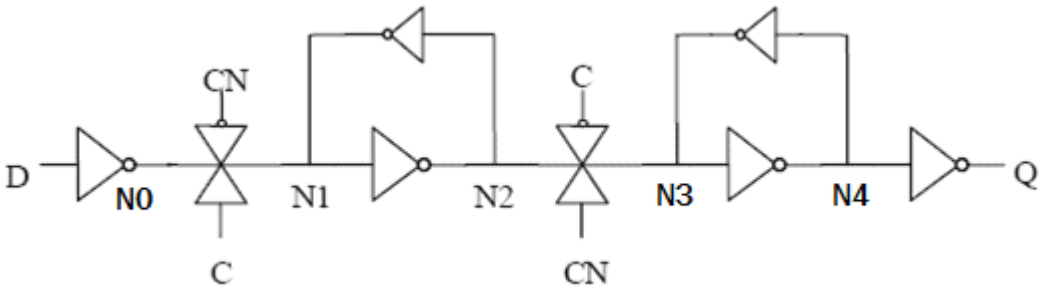
5. 解：



时钟的延迟：

CK→C： 1+1=2ns

CK→CN：1+0.5+0.5=2ns



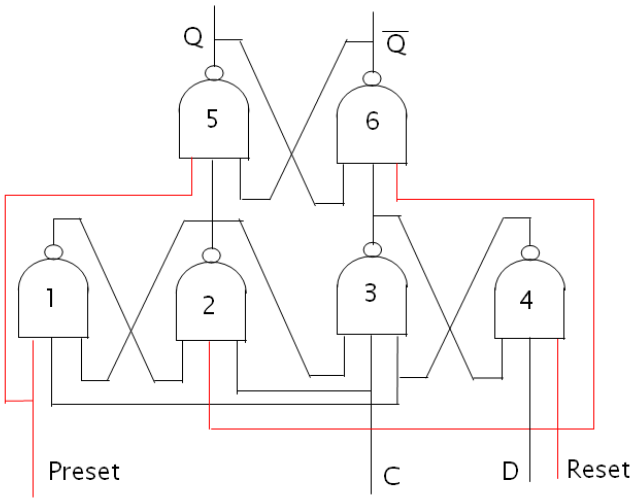
通过分析电路行为可知这是一个下降沿触发的 D 触发器。

建立时间指的是在时钟信号到达 CK 端之前，将触发器内部 N1 及 N2 状态改变并稳定为与 D 端数据相符所需的时间。这样，D 端数据必须通过 D → N0 → N1 → N2 才能真正改变触发器内部状态，但即使如此，由于 N1 和 N2 间反相器环驱动能力不能确定，为保守起见，还需要加上 N2 → N1 时间。此外考虑到接口处 CK 端时钟信号到 C 和 CN 的传播时延，如果 C 和 CN 的传播时延不一，可能导致传输门输出弱 1 或弱 0 情况，仍从保守情况出发取两者的较小值，另外还要算上传输门控制端栅到漏（源）的延迟。这样，该触发器建立时间 $T_{setup}=T_{D-N0-N1-N2-N1}-(\min(T_{CK-C},T_{CK-CN})+T_{tran})=(1+0.5+1+1)-(\min(2,2)+0.75)=0.75$ ns

保持时间指的是在时钟信号到达 CK 端之后，D 端需要等待多长时间，使得即使其数据变化也不影响触发器内部状态。反过来想，那什么情况下 D 端数据变化可能会影响内部状态呢？只有当前级传输门在完全关断之前，D 端数据已经进入到 N1，进而才有可能对内部状态产生影响。所以只需保证在前级传输门关断时变化的 D 端数据不进入 N1 即可。此外也要考虑到时钟信号的传播延迟，仍从保守情况出发取两者较大值，加上传输门控制端栅到漏（源）的延迟。这样，$T_{hold}=(\max(T_{CK-C},T_{CK-CN})+T_{tran})-T_{D-N0-N1}=(\max(2,2)+0.75)-(1+0.5)=1.25$ns。

CK→Q 时间指的是时钟触发沿到来之后 Q 端输出新的触发器状态所需的时间。只有当后级传输门打开后，Q 端才有可能与触发器内部状态相符，也就是 C=1→0（CN=0→1）时钟下降沿时，这时候 N2 处的状态需要通过 N2→N3→N4→Q，此时由于后级传输门出于打开状态，N3-N4 处的反相器环一般不可能再破坏这个新状态。此外仍出于保守考虑时钟信号的传播延迟取较大值，并加上传输门控制端栅到漏（源）的延迟。这样，该触发器 CK→Q 时间 $T_{CK-Q}=(\max(T_{CK-C},T_{CK-CN})+T_{tran})+T_{N2-N3-N4-Q}=(\max(2,2)+0.75)+(0.5+1+1)=5.25$ns

6. 解：



先不考虑 preset 和 reset 信号的影响，即 preset=1 且 reset=1，分析如下：
1. 当 C 信号发生 1→0 的变化时，2 单元和 3 单元强制输出{1,1},5 单元和 6 单元的状态继续保持。
2. 当 C 信号发生 0→1 的变化时，
若 D 输入为 0，4 单元输出为 1,1 单元输出为 0，使得 2 单元和 3 单元输出分别为 1 和 0，进而 5 单元和 6 单元的 Q 和 QN 输出分别为 0 和 1；
若 D 输入为 1，4 单元输出 0，使得 1 单元输出为 1，2 单元和 3 单元输出分别为 0 和 1，进而 5 单元和 6 单元的 Q 和 QN 输出分别为 1 和 0。
当 C 信号继续维持在 1 时，由于 2 单元和 3 单元的状态组合只可能是{0,1}和{1,0}中的一种，若 2 单元输出为 0，无论 D 输入如何影响 4 单元输出，1 单元和 3 单元始终为 1,D 输入信号无法穿透进入下一级；若 3 单元输出为 0，4 单元输出时钟为 1，D 输入信号无法穿透 4 单元，因此数据不再变化。
得到第一步结论：当复位无效时（即 preset 与 reset 均为 1 时），该电路只在 C 信号发生 0→1 变化时接受 D 输入信号，因此是一个 D 触发器。

再考虑 preset 和 reset 信号：
当 preset=0 且 reset=1 时，5 单元的 Q 输出为 1，1 单元输出 1，使得 2 单元输出 0，控制 3 单元输出 1，进而 6 单元受到 5 单元输出 Q 信号影响，输出 QN 为 0；
当 preset=1 且 reset=0 时，6 单元的 QN 输出为 1，2 单元输出 1，4 单元输出 1，使得 3 单元输出 0，进而 5 单元受到 6 单元输出 QN 信号影响，输出 Q 为 0；
当 preset=0 且 reset=0 时，5 单元和 6 单元的 Q 和 QN 输出都是 1，不符合单元逻辑要求，因此应当避免。

第四章 参考答案 指令系统结构

1.

本题常识：A. 从内存取指令，也算内存交换； B. 每个单独的指令，均必须以字节为单位，

不存在半字节。。

有的同学认为累加器型有 sub 指令，这种情况下，两个都是累加器型最优

解：(1)

	堆栈型	累加器型	寄存器-存储器型	寄存器-寄存器型

汇编代码	Push B
	Push C
	Sub
	Pop A
	Push A
	Push C
	Sub
	Pop D
	Push D
	Push A
	Add
	Pop B
		Load C
		Neg
		Add B
		Store A
		Load C
		Neg
		Add A
		Store D
		Add A
		Store B
			Load R1,B
			Sub R1,C
			Store R1,A
			Sub R1,C
			Store R1,D
			Add R1,A
			Store R1,B
				Load R1,B
				Load R2,C
				Sub R3,R1,R2
				Store R3,A
				Sub R4,R3,R2
				Store R4,D
				Add R5,R4,R3
				Store R5,B

指令字节	30	26	28	29
内存交换字节	48	42	42	39
代码量衡量		√		
交换数据量衡量				√

2. 常识题。

解：小尾端：

Address	0xxx000	0xxx001	0xxx010	0xxx011	0xxx100	0xxx101	0xxx110	0xxx111
0x	4E	4F	53	47	4E	4F	4F	4C
ASCII	N	O	S	G	N	O	O	L

大尾端：

Address	0xxx000	0xxx001	0xxx010	0xxx011	0xxx100	0xxx101	0xxx110	0xxx111
0x	4C	4F	4F	4E	47	53	4F	4E
ASCII	L	O	O	N	G	S	O	N

3. 题目问的是性能，可以默认问的是运行该程序（满足 25%是条件转移）的性能。最终以实际运行的时间为标准；想计算时间，需要先计算总指令数量，才能算出所需时间。

解：假设 CPU A 总指令数为 x，根据题意，转移指令有 0.25x，条件码指令 0.25x，其它指令 0.5x，因此 A 执行周期数 2*0.25x+0.75x=1.25x

可得 CPU B 总指令数为 0.75x，其中转移指令 0.25x，其它指令 0.5x。

B 执行周期数 2*0.25x+0.5x=1x

当 CPU A 频率为 1.2 倍时，性能是 CPU B 的 1.2/1.25=0.96 倍

当 CPU A 频率为 1.1 倍时，性能是 CPU B 的 1.1/1.25=0.88 倍

因此 CPU A 两种情况下都差

4.

a） lw $1, 0($n)

add $2, $2, $1

bnez $1, 1f //任何将$1 作为 src 的指令都可以

b） 假设需要减少 x 的 load 指令。减少后，指令数为 1-0.26x。则(1-0.26x)/0.95=1

x=19%

c） 困难在于访存 MEM 在 EXE 之前就要进行，而 add $2, 0($n)需要先访存后 EXE

题意是增加寄存器-内存形式指令，因此有的同学曾经回答“lw r1,0(r3) add r3, r2, r1”，但可以通过增加 add r3, r2, 0(r3)这样的寄存器-内存形式指令来实现消除。虽然与题干不完全一致，但是是可以通过简单思考，就知道这个也属于“寄存器-内存”形式指令。

5.

a） 条件转移指令的跳转范围。

16+2 位 256KB（+-128KB）

b） 直接跳转指令的跳转范围。

26+2 位 256MB （注意并非+-128MB，这是 MIPS 指令集的特点）

6.

解：

按小尾端的解答：

```
dli r2, 1005
lwr r1, 0x0(r2)
lwl r1, 0x3(r2)
dli r2, 2005
swr r1, 0x0(r2)
swl r1, 0x3(r2)
```

7.

解： 常识题。

```
1:  ll r1, 100(r2)
    add r1, r1, 100
    sc r1, 100(r2)
    beqz r1, 1b
    nop
```

如果在 ll 和 sc 之间（含 ll 和 sc），发生了中断或者其他处理器修改 100(r2)，则 sc 之后 r1 会变 0，回到标号 1 重新执行。

8. 解： **资料查阅题**

X86 的减法指令如下：

定点减法：

SUB AL,imm8	Subtract imm8 from AL
SUB AX,imm16	Subtract imm16 from AX
SUB EAX,imm32	Subtract imm32 from EAX
SUB RAX,imm32	Subtract sign-extend imm32 from RAX
SUB r/m8,imm8	Subtract imm8 from 8bit register or 8bit memory location
SUB r/m16,imm16	Subtract imm16 from 16bit register or 16bit memory location
SUB r/m32,imm32	Subtract imm32 from 32bit register or 32bit memory location
SUB r/m64,imm32	Subtract sign-extend imm32 from 64bit register or 64bit memory location
SUB r/m16,imm8	Subtract sign-extend imm8 from 16bit register or 16bit memory location
SUB r/m32,imm8	Subtract sign-extend imm8 from 32bit register or 32bit memory location
SUB r/m64,imm8	Subtract sign-extend imm8 from 64bit register or 64bit memory location
SUB r/m8,r8	Subtract 8bit register from 8bit register or 8bit memory location
SUB r/m16,r16	Subtract 16bit register from 16bit register or 16bit memory location
SUB r/m32,r32	Subtract 32bit register from 32bit register or 32bit memory location
SUB r/m64,r32	Subtract sign-extend 32bit register from 64bit register or 64bit memory location
SUB r8,r/m8	Subtract 8bit register or 8bit memory location from 8bit register
SUB r16,r/m16	Subtract 16bit register or 16bit memory location from 16bit register
SUB r32,r/m32	Subtract 32bit register or 32bit memory location from 32bit register
SUB r64,r/m64	Subtract 64bit register or 64bit memory location from 64bit register

Flag 影响：

OF, SF, ZF, AF, PF, CF 被影响

Protected 模式下例外：

#GP(0)	If the destination is located in a non-writable segment
	If a memory operand effective address is outside the CS DS ES FS or GS segment limit
	If the DS ES FS or GS register contains a NULL segment selector
#SS(0)	If a memory operand effective address is outside the SS segment limit
#PF(fault-code)	If a page fault occurs
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the

current privilege level is 3

Real-address 模式下例外

#GP If a memory operand effective address is outside the CS FS ES FS or GS segment limit

#SS If a memory operand effective address is outside the SS segment limit

Virtual-8086 模式下例外

#GP(0) If a memory operand effective address is outside the CS FS ES FS or GS segment limit

#SS(0) If a memory operand effective address is outside the SS segment limit

#PF(fault-code) If a page fault occurs

#AC(0) If alignment checking is enabled and an unaligned memory reference is made

Compatibility 模式下例外

同 Protected 模式

64bit 模式例外

#SS(0) If a memory address referencing the SS segment is in a non-canonical form

#GP(0) if the memory address is in a non-canonical form

#PF(fault-code) If a page fault occurs

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3

X86 浮点减法指令如下：

FSUB m32fp	Subtract m32fp from ST(0) and store result in ST(0)
FSUB m64fp	Subtract m64fp from ST(0) and store result in ST(0)
FSUB ST(0),ST(i)	Subtract ST(i) from ST(0) and store result in ST(0)
FSUB ST(i),ST(0)	Subtract ST(0) from ST(i) and store result in ST(i)
FSUBP ST(i),ST(0)	Subtract ST(0) from ST(i) and store result in ST(i),and pop register stack
FSUBP	Subtract ST(0) from ST(1) and store result in ST(1),and pop register stack
FISUB m32int	Subtract m32int from ST(0) and store result in ST(0)
FISUB m16int	Subtract m16int from ST(0) and store result in ST(0)

FSUBR m32fp	Subtract ST(0) from m32fp and store result in ST(0)
FSUBR m64fp	Subtract ST(0) from m64fp and store result in ST(0)
FSUBR ST(0),ST(i)	Subtract ST(0) from ST(i) and store result in ST(0)
FSUBR ST(i),ST(0)	Subtract ST(i) from ST(0) and store result in ST(i)
FSUBRP ST(i),ST(0)	Subtract ST(i) from ST(0) and store result in ST(i),and pop register stack
FSUBRP	Subtract ST(1) from ST(0) and store result in ST(1),and pop register stack
FISUBR m32int	Subtract ST(0) from m32int and store result in ST(0)
FISUBR m16int	Subtract ST(0) from m16int and store result in ST(0)

//FISUB 和 FISUBR 将定点转化为 X86 的扩展双精度浮点格式(80bit)

FPU Flags 影响：

C1 set to 0 if stack underflow occurred

Set if result was rounded up; cleared otherwise

C0, C2, C3 undefined

浮点例外：

#IS stack underflow occurred
#IA Operand is an SNaN value or unsupported format
Operands are infinities of like sign
#D Source operand is a denormal value
#U Result is too small for destination format
#O Result is too large for destination format
#P Value cannot be represented exactly in destination format

Protected 模式下例外：
#GP(0) If a memory operand effective address is outside the CS DS ES FS or GS segment limit
If the DS ES FS or GS register contains a NULL segment selector
#SS(0) If a memory operand effective address is outside the SS segment limit
#NMCR0.EM[bit2] or CR0.TS[bit3] = 1
#PF(fault-code) If a page fault occurs
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3
Real-address 模式下例外
#GP If a memory operand effective address is outside the CS FS ES FS or GS segment limit
#SS If a memory operand effective address is outside the SS segment limit
#NMCR0.EM[bit2] or CR0.TS[bit3] = 1
Virtual-8086 模式下例外
#GP(0) If a memory operand effective address is outside the CS FS ES FS or GS segment limit
#SS(0) If a memory operand effective address is outside the SS segment limit
#PF(fault-code) If a page fault occurs
#AC(0) If alignment checking is enabled and an unaligned memory reference is made
#NMCR0.EM[bit2] or CR0.TS[bit3] = 1
Compatibility 模式下例外
同 Protected 模式
64bit 模式例外
#SS(0) If a memory address referencing the SS segment is in a non-canonical form
#GP(0) if the memory address is in a non-canonical form
#NMCR0.EM[bit2] or CR0.TS[bit3] = 1
#MF If there is a pending x87 FPU exception
#PF(fault-code) If a page fault occurs
#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3

MIPS 的减法如下：
定点减法：
DSUB rd, rs, rt 64bit 减法
例外：
Integer Overflow, Reserved Instruction
DSUBU rd, rs, rt 64bit 无符号减法

例外：

Reserved Instruction

SUB rd, rs, rt　　32bit 减法

限制：

On 64-bit processors, if either GPR rt or GPR rs does not contain sign-extended 32-bit values (bits 63..31 equal), then　　the result of the operation is UNPREDICTABLE

例外：

Integer Overflow

SUBU rd, rs, rt 32bit 无符号减法

限制：

On 64-bit processors, if either GPR rt or GPR rs does not contain sign-extended 32-bit values (bits 63..31 equal), then　　the result of the operation is UNPREDICTABLE

例外：

无

MIPS 浮点减法(SUB.fmt)如下：

SUB.S fd, fs, ft　　单精度

SUB.D fd, fs, ft　　双精度

SUB.PS fd, fs, ft　　并行单精度(将 fs 和 ft 的上下两部分分别相减)

限制：

The field fs, ft, fd must specify FPRs valid for operands of type fmt. If they are not valid, the result is UNPREDICTABLE

The operands must be values in format fmt, if they are not, the result is UNPREDICTABLE and the value of the operand FPRs becomes UNPREDICTABLE

The result of SUB.PS is UNPREDICTABLE if the processor is executing in 16 FP registers mode.

例外：

Coprocessor Unusable, Reserved Instruction

浮点例外：

Inexact, Overflow, Underflow, Invalid Op, Unimplemented Op

X86 和 MIPS 减法指令比较：

字长：

X86 的定点减法指令支持 8 位、16 位、32 位、64 位字长；而 MIPS 定点减法支持 32 位和 64 位字长。

浮点：

X86 的浮点减法指令均为 80 位扩展双精度格式；而 MIPS 浮点减法支持单精度、双精度和并行单精度格式。

寻址方式：

MIPS 的减法只支持寄存器寻址。

X86 的定点减法支持寄存器寻址、立即数和内存寻址方式(直接寻址、变址寻址、间接寻址、基址寻址、基址加变址寻址)；

X86 的浮点减法支持寄存器寻址(浮点寄存器栈)和内存寻址方式(直接寻址、变址寻址、间接寻址、基址寻址、基址加变址寻址)。

其他区别：

X86 的定点减法会修改 Flags，浮点减法会修改 FPU flags，而 MIPS 的减法没有 Flags。

X86 的减法和 MIPS 减法产生的例外由于体系结构的不同而有很大不同：

X86 的减法会产生 General-Protection 例外、Stack-Segment Fault 例外；除了在 real-address 模式下之外，还会产生 Page Fault 例外、Alignment Check 例外；所有的浮点减法还会产生 Device Not Available（No Math Coprocessor）例外；在 64bit 模式下进行浮点减法还会产生 x87 FPU Floating Point Error（Math Fault）例外。

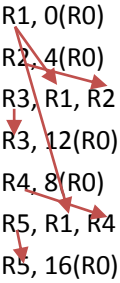
MIPS 的 DSUB、DSUBU 和所有的浮点减法会触发保留指令例外，DSUB 和 SUB 会触发溢出例外，浮点减法会触发协处理器不可用例外和一些浮点例外(Inexact, Overflow, Underflow, Invalid Op, Unimplemented Op)

9. **资料查阅题。略。**

第五章 参考答案 静态流水线

1. 解：

分析相关：



```
LW      R1, 0(R0)
LW      R2, 4(R0)
ADD     R3, R1, R2      ;a=b+e
SW      R3, 12(R0)
LW      R4, 8(R0)
ADD     R5, R1, R4      ;c=b+f
SW      R5, 16(R0)
```

重排序：

如果严格按照教科书中流水线结构求解：该流水线中，没有 WB 和 ID 冲突时的前递路径。存在两种，运算指令 EX 到 ID（或者说下一周期 EX）的前递，以及 MEM 到 ID（或者说下一周期 EX）的前递。

	1	2	3	4	5	6	7	8	9
LW R1, 0(R0)	**IF**	**ID**	**EX**	**MEM**	**WB**				
LW R2, 4(R0)		**IF**	**ID**	**EX**	**MEM**	**WB**			
ADD R3, R1, R2			**IF**	**ID**	**ID**	**ID**	**ID**	**EX**	**MEM**
SW R3, 12(R0)				**IF**	**IF**	**IF**	**IF**	**ID**	**EX**
LW R4, 8(R0)								**IF**	**ID**
ADD R5, R1, R4									**IF**
SW R5, 16(R0)									

	10	11	12	13	14	15
LW R1, 0(R0)						
LW R2, 4(R0)						
ADD R3, R1, R2	**WB**					
SW R3, 12(R0)	**MEM**	**WB**				
LW R4, 8(R0)	**EX**	**MEM**	**WB**			
ADD R5, R1, R4	**ID**	**ID**	**EX**	**MEM**	**WB**	
SW R5, 16(R0)	**IF**	**IF**	**ID**	**EX**	**MEM**	**WB**

排序前共需要 15 拍。重排后的指令序列为：

```
LW      R1, 0(R0)
LW      R2, 4(R0)
LW      R4, 8(R0)
ADD     R5, R1, R4      ;c=b+f（先算后 LW 出来的）
SW      R5, 16(R0)
ADD     R3, R1, R2      ;a=b+e（再算先 LW 出来的）
SW      R3, 12(R0)
```

	1	2	3	4	5	6	7	8	9	10	11	12
LW R1, 0(R0)	**IF**	**ID**	**EX**	**MEM**	**WB**							

LW R2, 4(R0)		IF	ID	EX	MEM	WB						
LW R4, 8(R0)			IF	ID	EX	MEM	WB					
ADD R5, R1, R4				IF	ID	ID	EX	MEM	WB			
SW R5, 16(R0)					IF	IF	ID	EX	MEM	WB		
ADD R3, R1, R2							IF	ID	EX	MEM	WB	
SW R3, 12(R0)								IF	ID	EX	MEM	WB

通过指令重排，上述指令序列只需要 12 拍，**减少了 3 拍**。

利用了存在 MEM 到 EX 的前递，但是没有 WB 到 EX 前递的特点。

或者，按照常识，前递通常是做全的，认为有所有前递。答案 13-11=2，即

重排序前需要 13 拍

	1	2	3	4	5	6	7	8	9	10	11	12	13
LW R1, 0(R0)	IF	ID	EX	MEM	WB								
LW R2, 4(R0)		IF	ID	EX	MEM	WB							
ADD R3, R1, R2			IF	ID	ID	EX	MEM	WB					
SW R3, 12(R0)				IF	IF	ID	EX	MEM	WB				
LW R4, 8(R0)						IF	ID	EX	MEM	WB			
ADD R5, R1, R4							IF	ID	ID	EX	MEM	WB	
SW R5, 16(R0)								IF	IF	ID	EX	MEM	WB

排序后代码：（有多种正确答案）

```
LW      R1, 0(R0)
LW      R2, 4(R0)
LW      R4, 8(R0)
ADD     R3, R1, R2      ;a=b+e
ADD     R5, R1, R4      ;c=b+f
SW      R3, 12(R0)
SW      R5, 16(R0)
```

排序后需要 11 拍，**减少了 2 拍**。

2. 解：　**题目要求无旁路，画出时空图。**

画时空图时，不能出现结构相关，即同一拍，不能有多条指令处于同一个流水线阶段。

MIPS 代码如下：

```
LW      R1, 0(R0)       ; load A
LW      R2, 4(R0)       ; load B
ADD     R3, R1, R2      ; A=A+B
SUB     R4, R3, R2      ; C=A-B
SW      R3, 0(R0)       ; store A
SW      R4, 8(R0)       ; store C
```

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
LW R1, 0(R0)	IF	ID	EX	MEM	WB										
LW R2, 4(R0)		IF	ID	EX	MEM	WB									
ADD R3, R1, R2			IF	ID	ID	ID	ID	EX	MEM	WB					

SUB R4, R3, R2				IF	IF	IF	IF	ID	ID	ID	ID	EX	MEM	WB	
SW R3, 0(R0)								IF	IF	IF	IF	ID	EX	MEM	WB
SW R4, 8(R0)												IF	ID	ID	ID

	16	17	18
LW R1, 0(R0)			
LW R2, 4(R0)			
ADD R3, R1, R2			
SUB R4, R3, R2			
SW R3, 0(R0)			
SW R4, 8(R0)	**EX**	**MEM**	**WB**

3. 解：

假设此处的浮点运算均为单精度浮点运算。数组的长度为 N(N<2^{15}-1)。

	ADDIU R3, R0, N	; N
	SLL R3, R3, 2	; N*4
	ADDU R3, R1, R3	; R3=R1+N*4
Loop:	LWC1 F1, 0(R1)	; load X(i)
	LWC1 F2, 0(R2)	; load Y(i)
	MUL.S F3, F0, F1	; a*X(i)
	ADD.S F4, F2, F3	; a*X(i)+Y(i)
	ADDIU R1, R1, 4	; i++ for X
	ADDIU R2, R2, 4	; i++ for Y
	BNE R1,R3, Loop	
	SWC1 F4, -4(R1)	; store to X(i)

（本题引入的新的概念，即“花费 X 个周期”，根据课本，此处应该很自然地理解为 EX 阶段需要几个周期。）

	1	2	3	4	5	6	7	8	9
LWC1 F1, 0(R1)	**IF**	**ID**	**EX**	**MEM**	**WB**				
LWC1 F2, 0(R2)		**IF**	**ID**	**EX**	**MEM**	**WB**			
MUL.S F3, F0, F1			**IF**	**ID**	**EX1**	**EX2**	**EX3**	**EX4**	**MEM**
ADD.S F4, F2, F3				**IF**	**ID**	**ID**	**ID**	**ID**	**EX1**
ADDIU R1, R1, 4					**IF**	**IF**	**IF**	**IF**	**ID**
ADDIU R2, R2, 4									**IF**
BNE R1,R3, Loop									
SWC1 F4, -4(R1)									

	10	11	12	13	14	15	16	17
LWC1 F1, 0(R1)								
LWC1 F2, 0(R2)								

MUL.S F3, F0, F1	WB							
ADD.S F4, F2, F3	EX2	EX3	MEM	WB				
ADDIU R1, R1, 4	ID	ID	EX	MEM	WB			
ADDIU R2, R2, 4	IF	IF	ID	EX	MEM	WB		
BNE R1,R3, Loop			IF	ID	EX	MEM	WB	
SWC1 F4, -4(R1)				IF	ID	EX	MEM	WB

4. 解： **注意读题，本题两问里，旁路的设计是不同的。**

(1) 无旁路部件，前 99 次循环的流水线时空图如下

	1	2	3	4	5	6	7	8	9	10	11
LD R1, 0(R2)	IF	ID	EX	MEM	WB						
DADDI R1, R1, 4		IF	ID	ID	ID	EX	MEM	WB			
SD R1, 0(R2)			IF	IF	IF	ID	ID	ID	EX	MEM	WB
DADDI R2, R2, 4						IF	IF	IF	ID	EX	MEM
DSUB R4, R3, R2									IF	ID	ID
BNEZ R4, Loop										IF	IF
（延迟槽 NOP）											
LD R1, 0(R2)											

	12	13	14	15	16	17	18	19	20	21
LD R1, 0(R2)										
DADDI R1, R1, 4										
SD R1, 0(R2)										
DADDI R2, R2, 4	WB									
DSUB R4, R3, R2	ID	EX	MEM	WB						
BNEZ R4, Loop	IF	ID	ID	ID	EX	MEM	WB			
（延迟槽 NOP）		IF	IF	IF	ID	EX	MEM	WB		
LD R1, 0(R2)						IF	ID	EX	MEM	WB

前 99 次循环，每次都是预测错，每个循环 16 个时钟周期。最后一个循环，预测对，但整个循环最后一条指令写回需要 19 个时钟周期（有延迟槽）。

共计：16×99+19=1603 时钟周期

(2) 有旁路，预测 taken，前 99 次循环流水线时空图如下：

	1	2	3	4	5	6	7	8	9	10	11	12	13
LD R1, 0(R2)	IF	ID	EX	MEM	WB								
DADDI R1, R1, 4		IF	ID	ID	EX	MEM	WB						
SD R1, 0(R2)			IF	IF	ID	EX	MEM	WB					
DADDI R2, R2, 4					IF	ID	EX	MEM	WB				
DSUB R4, R3, R2						IF	ID	EX	MEM	WB			
BNEZ R4, Loop							IF	ID	EX	MEM	WB		
（延迟槽 NOP）								IF	ID	EX	MEM	WB	
LD R1, 0(R2)									IF	ID	EX	MEM	WB

前 99 次循环，执行一个循环需要 8 拍。最后一次循环完整执行完需要 12 拍。

共计：8×99+12=804 时钟周期

(3) 仍认为分支预测 taken

Loop:

```
LD      R1, 0(R2)
DADDI   R2, R2, #4
DSUB    R4, R3, R2
DADDI   R1, R1, #4
BNEZ    R4, Loop
SD      -4(R2), R1
```

	1	2	3	4	5	6	7	8	9	10	11
LD R1, 0(R2)	IF	ID	EX	MEM	WB						
DADDI R2, R2, 4		IF	ID	EX	MEM	WB					
DSUB R4, R3, R2			IF	ID	EX	MEM	WB				
DADDI R1, R1, 4				IF	ID	EX	MEM	WB			
BNEZ R4, Loop					IF	ID	EX	MEM	WB		
SD -4(R2), R1						IF	ID	EX	MEM	WB	
LD R1, 0(R2)							IF	ID	EX	MEM	WB

前 99 次循环，执行一个循环需要 6 拍。最后一次循环完整执行完需要 10 拍。

整个循环共计：6×99+10=604 时钟周期

5. 解： 概念题

空操作指令(nop 指令)，其不改变程序可见寄存器、状态寄存器以及内存的状态，以及用于等待需要一定周期执行的操作。nop 指令的作用，常见的有：取指的强制访存对齐（memory alignment），防止相关风险（hazard），以及用于填充延迟槽（branch delay slot）。

7. 解： verilog 题。

(1) ALU 模块

```
module alu_module
(
    input  [ 3:0] aluop,
    input  [31:0] in1,
    input  [31:0] in2,
    output [31:0] out
);

wire slt_res;
assign slt_res = (in1[31] && !in2[31])   & 1'b1 |
                 (in1[31] && in2[31] )   & (in1<in2) |
                 (!in1[31] &&  !in2[31]) & (in1<in2) |
                 (!in1[31] && in2[31])   & 1'b0;

assign out =
  {32{aluop==4'b0000}} & (in1+in2) |
  {32{aluop==4'b0001}} & (in1-in2) |
```

```
  {32{aluop==4'b0010}} & {31'b0, slt_res} |
  {32{aluop==4'b0011}} & {31'b0, in1<in2} |
  {32{aluop==4'b0100}} & (in1&in2) |
  {32{aluop==4'b0101}} & (in1|in2) |
  {32{aluop==4'b0110}} & (in1^in2) |
  {32{aluop==4'b0111}} & ~(in1|in2) |
  {32{aluop==4'b1000}} & (in1<<in2) |
  {32{aluop==4'b1010}} & (in1>>in2) |
  {32{aluop==4'b1011}} & ($signed(in1)>>>in2); //verilog2001 中算术右移的新表达方式

endmodule
```

如果信号声明不加 signed 标记，则都为无符号操作(包括小于号“<”、算术右移号“>>>”)。当然也可以用其他表达式，强行写出算术右移的功能。

第六章 参考答案 动态流水线

1. 答： 概念题

保留站：指令有序进入保留站，在保留站内等待相关被解决后乱序发射至功能部件进行执行。

重命名寄存器：通过对同一个逻辑寄存器重命名成多个不同的物理寄存器，可以解决 WAW 相关，并且避免伪 RAW 相关。

Reorder buffer：指令不论何时写回寄存器，它在 reoder buffer 中退出的顺序必须**符合程序序**。这保证了乱序执行下，**中断能有精确现场**。

2. 证明：

如果 A(a*i+b)和 A(c*i+d)之间存在相关，则必然存在某个 i1 和 i2，使得 a*i1+b=c*i2+d

这样 d-b=c*i2-a*i1=GCD(c,a)*((c/GCD(c,a))*i2-(a/GCD(c,a))*i1)。

也就是说(d-b)/GCD(c,a)=((c/GCD(c,a))*i2-(a/GCD(c,a))*i1)。

由于上式右半部分显然是整数，因此 GCD(c,a)能够整除(d-b)

3. 答：概念题

在 Tomasulo 算法中：

通过寄存器重命名技术，消除了 WAR 和 WAW 相关。在 tomasulo 算法中，寄存器重命名的功能是通过保留站来实现的。当指令发射到保留站时，它的目标寄存器对应保留站号，消除了 WAR 相关和 WAW 相关。

如果指令的某操作数没有准备好，在它的保留站中的该操作数的位置上标记的是产生该操作

数的指令的保留站号。只有当操作数都准备好了，指令才能执行，保证了 RAW 相关。

在记分板技术中：
在指令发射阶段，如果存在 WAW 相关，则停止发射该指令和任何后继指令。
如果指令的源操作数都准备好了，则计分板会通知相应功能部件读取该操作数并开始执行该指令。这样保证了 RAW 相关。
在指令写回阶段，如果存在 WAR 相关，则停止写回。

4. 答：**概念题**

所谓精确例外，就是指在处理例外的时候，发生例外指令之前所有的指令都已经执行完了，例外指令后面的所有指令都还没有执行(严格的说是没有产生执行效果，即修改处理机状态)。实现精确例外处理的办法，就是把后面指令对机器状态的修改延迟到前面指令都已经执行完。具体来讲，在流水线中增加一个叫提交（Commit）的阶段，在这个阶段指令才真正修改机器状态。在执行或者写回阶段，把指令的结果先写到被称为重排序缓存（ReOrder Buffer，简称 ROB）的临时缓冲器中；在提交的时候，再把 ROB 的内容写回到寄存器或者存储器。一条指令发生了例外，那么对应的 ROB 项，以及之后的指令的 ROB 项就丢弃掉不写回。而这条指令之前的 ROB 项都在 commit 时真正修改寄存器或者存储器。这样就保证了精确中断。

5. 解： **有的同学喜欢多展开几次（比如 4 次）。但尽量按最少展开次数来做题。**

a） 展开两次循环即可消除相关带来的阻塞影响。

```
bar:
        L.D     F2, 0(R1)       ;
        L.D     F3, 8(R1)       ;
        L.D     F6, 0(R2)       ;
        MUL.D   F4, F2, F0      ;
        MUL.D   F5, F3, F0      ;
        L.D     F7, 8(R2)       ;
        DADDUI  R1, R1, #16     ;
        ADD.D   F6, F4, F6      ;
        ADD.D   F7, F4, F7      ;
        DADDUI  R2, R2, #16     ;
        S.D     -16(R2), F6     ;
        DSGTUI  R3, R1, #800    ;
        BEQZ    R3, bar         ;
```

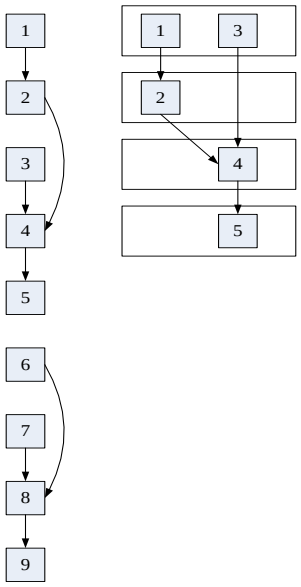
S.D -8(R2), F7 ;

14 拍两个，也就是 7 拍 1 个。

软流水的全代码参见下面表格。

b） 首先分析原有循环中数据流图，将运算部分的流图进行流水切分。

```
1   L.D     F2, 0(R1)
2   MUL.D   F4, F2, F0
3   L.D     F6, 0(R2)
4   ADD.D   F6, F4, F6
5   S.D     0(R2), F6
6   DADDUI  R1, R1, #8
7   DADDUI  R2, R2, #8
8   DSGTUI  R3, R1, #800
9   BEQZ    R3, bar
```



软件流水后主题循环代码如下：

```
S.D     -24(R2), F6     ;第 i-3 次循环的 5
ADD.D   F6, F4, F8      ;第 i-2 次循环的 4
MUL.D   F4, F2, F0      ;第 i-1 次循环的 2
L.D     F2, 0(R1)       ;
L.D     F8, -8(R2)      ;
DADDUI  R1, R1, #8      ;
DSGTUI  R3, R1, #800    ;
BEQZ    R3, bar         ;
DADDUI  R2, R2, #8      ;
```

9 拍处理 1 个元素

注： 假如 SD 的偏移为 A，LD(R1)的偏移为 B，LD(R2)的偏移为 C，循环退出条件为 D，有 A+16 = C ， D+B=800，一个例子是（-24，0，-8， 800）或者（-16，0， 0， 800）

全部代码：

装入		循环		排空	
L.D	F2, 0(R1)	S.D	-Y(R2), F6	S.D	-Y(R2),F6
L.D	F8, 0(R2)	ADD.D	F6, F4, F8	ADD.D	F6, F4, F8
MUL.D	F4, F2, F0	MUL.D	F4, F2, F0	S.D	8-Y(R2),F6

```
ADD.D   F6, F4, F8
L.D     F2, 8(R1)
L.D     F8, 8(R2)
MUL.D   F4, F2, F0
L.D     F2, 16(R1)
DADDUI  R1, R1, X
DADDIU  R2, R2, Y
```

```
L.D     F2, 24-X(R1)
L.D     F8, 16-Y(R2)
DADDUI  R1, R1, #8
DSGTUI  R3, R1, 800-24+X
BEQZ    R3, bar
DADDUI  R2, R2, #8
```

```
MUL.D   F4, F2, F0
L.D     F8, 16-Y(R2)
ADD.D   F6, F4, F8
S.D     16-Y(R2),F6
```

6. 解： **题干里，关于“浮点”的定义不是很精确。**

指令执行的状态表：(数字表示该操作在第几个时钟周期发生)

指令	发射	执行	写回	提交
DIV F0, F1, F2	1	2	4	5
MUL F3, F0, F2	2	4	6	7
ADD F0, F1, F2	3	4	5	8
MUL F3, F0, F2	4	5	7	9

Cycle 1

3		4.0	3				resbus
2		3.0	2				
1		2.0	1				
0	0	1.0	0	div	F0		

Cycle 2

3	1	4.0	3				resbus
2		3.0	2				
1		2.0	1	mul	F3		
0	0	1.0	0	div	F0		

Cycle 3

3	1	4.0	3				resbus
2		3.0	2	add	F0		
1		2.0	1	mul	F3		
0	2	1.0	0	div	F0		

Cycle 4

3	3	4.0	3	mul	F3		resbus
2		3.0	2	add	F0		(0, 0.67)
1		2.0	1	mul	F3		

0	2	1.0		0	div	F0	0.67

Cycle 5

3	3	4.0		3	mul	F3		resbus
2		3.0		2	add	F0	5.0	(2, 5.0)
1		2.0		1	mul	F3		
0	2	0.67		0				

Cycle 6

3	3	4.0		3	mul	F3		resbus
2		3.0		2	add	F0	5.0	(1, 2.0)
1		2.0		1	mul	F3	2.0	
0	2	0.67		0				

7. 解：

a)

mul f0,f0,f0;

add f1,f1,f1;

上述指令序列会同时写回，导致停顿。

b)至少需要 3 条结果总线。

注：如果访存定点部件为全流水且延迟固定为 2 或者 3，则只需要 2 条结果总线。

第七章 参考答案 多发射数据通路

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。**概念题**

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。**概念题**

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 LD	R2, R2, 2 R2, 4(R2)
②	DSUB SD	R3, R1, R2 R2, 7(R1)
③	S.D S.D	F2, 7(R1) F2, 200(R7)
④	BLE SD	R2, place 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： 注意没有延迟槽,注意指令的延迟。（延迟的定义参见 6 章 5 题）

(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。

正常展开即可

(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

参考答案假设是直接按定点一条浮点一条来做。

(3) **实验题。**

```
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;
}
```

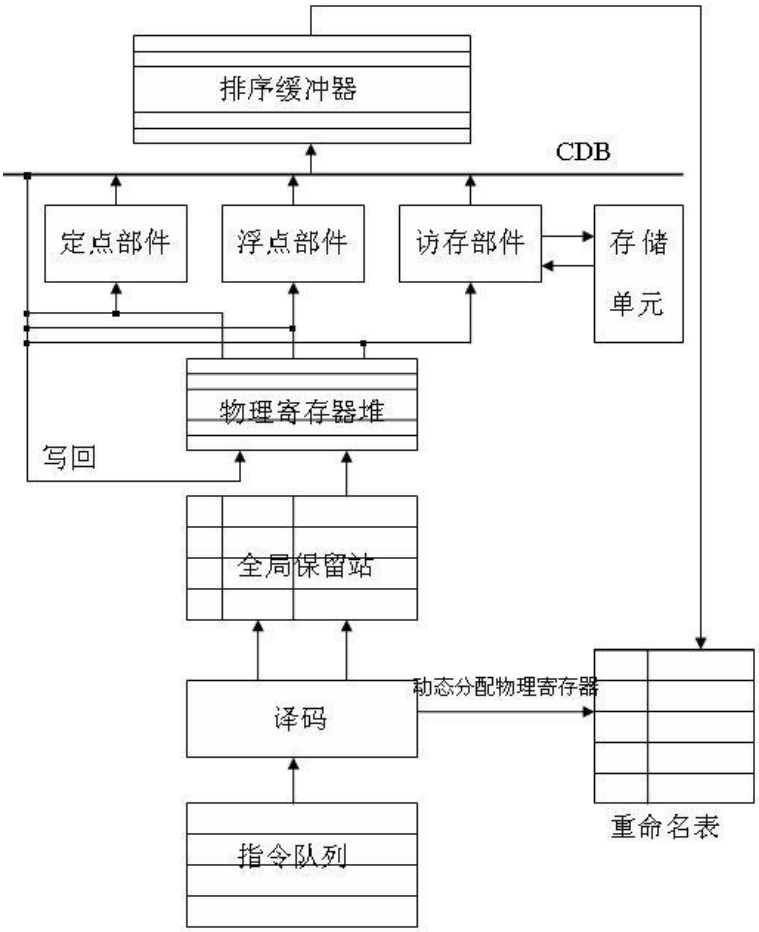
gcc -O 不进行优化， -O1, -O2, 进行部分优化，但是不进行循环展开优化；-O3 进行所有的优化，包括循环展开，对该程序非常有效。

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：**概念题**



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. 资料查阅题

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。

第八章 参考答案 转移预测

1. 解：BHT $2^6*9=576b$

PHT $2^8*2^9*2=262144b$

共 262720b

基本原理：BHT 根据地址低 6 位选出一个 9 位向量，和地址低 8 位一起到 PHT 中选取 2 位饱和计数。图略

2. 解： 这个题的题干歧义较大。第一方面，表的格式有歧义，出题的本意是：(1)2 项的局部历史表 PHT （2）用 1 位全局历史去选择，因此，共有 2 张表，每张表两项，根据历史选择哪张表，根据 b1/b2，决定用表中的哪一项。第二方面，程序本身有个漏洞，出题的本意是这个序列会循环执行，但 L2 没有回跳，此外回跳也可能影响全局历史。

未来出题时，会明确几张表、每张几项、两条分支是否落在不同索引项上

下面的参考答案按上述假设来回答：

注：下表中的 NT/NT，表示两张表中属于某分支的项分别为 NT 和 NT。前者为历史为 NT 的表中的项，后者为历史为 T 的表中的项。加粗的是根据当前历史选中的表项。

a	b1 prediction	b1 action	New b1 prediction	b2 prediction	b2 action	New b2 prediction
0	**NT**/NT	NT	**NT**/NT	**NT**/NT	NT	**NT**/NT
1	**NT**/NT	T(miss)	T/NT	NT/**NT**	T(miss)	NT/T
0	T/**NT**	NT	T/NT	**NT**/T	NT	NT/T
1	**T**/NT	T	T/NT	NT/**T**	T	NT/T

因此，总共有 2 次预测错误出现。

3. 解：未来出题时，会明确“额外开销”

a) 1 + 20%*((90%*10%*5)+(10%*3)) = 1.15

b) 1 + 20%*2=1.4

1.4/1.15=1.22，慢 22%。

4. 解：**概念题**

循环内：

S1 中 a[i]赋值和 a[i]读出反相关

S2 中 a[i]读出和 S1 中 a[i]赋值真相关

循环间：

S1 中 b[i]读出和上一次循环 S4 中 b[i+1]赋值真相关

S3 中 a[i-1]赋值和上一次循环 S1 中 a[i]赋值输出相关

S3 中 a[i-1]赋值和上一次循环 S1 中 a[i]读出反相关

S3 中 a[i-1]赋值和上一次循环 S2 中 a[i]读出反相关

S3 中 b[i]读出和上一次循环 S4 中 b[i+1]赋值真相关

S4 中 b[i]读出和上一次循环 S4 中 b[i+1]赋值真相关

5. 解： **概念题**

(a)

```
1:beqz t1, 1f
  nop
  ……
1:bnez t1, 1b
  nop
```

假设两条转移指令共享一个历史表项，它们的结果完全相反

(b)

```
For(i=0;i<10;i++) j+=i;
For(i=0;i<10;i++) j+=2*i;
```

假如两个 for 循环的转移指令共享共享一个历史表项，第二个循环会得益于第一个循环的预测，第一次跳转能猜测成功（假设是 2 位饱和计数器）。

(c) 如果使用其他地址，例如高位地址，对某些程序会有好处，但离得很近的多条分支指令就会共享同一个表项，对不同的程序，会对分支正确率产生不同影响。

如果对多位地址进行哈希，效果会更好，会减少两条转移指令共享一个历史表项的概率。

6. 解：写出软流水代码，计算拍数即可。留意指令的延迟

（1）

```
//装入代码
        LDC1        F0,0(R1)
        ADD.D       F2,F0,F1
        LDC1        F0,-8(R1)
        ADDIU       R1,R1,-24
//主体循环
L1:     SDC1        F2,24(R1) //Loop i-2
        ADD.D       F2,F0,F1  //Loop i-1
        LDC1        F0,8(R1)  //Loop i
```

```
        BNE         R1,R2,L1
        ADDIU       R1,R1,-8
//排空代码
        SDC1        F2,24(R1)
        ADD.D       F2,F0,F1
        SDC1        F2,16(R1)
```

总的执行周期数：(4+1) + (5*(n-2)) + (3+2) = 5n

（2）软流水无法降低分支频率，循环展开可以；

软流水代码量增加少，循环展开会带来代码膨胀。

7. 答：通过在处理器维护一个指令的有序队列（如 reorder-buffer, branch-queue）。当发现某转移指令猜测失败时，依据该有序队列的顺序信息，将该转移指令后面的指令取消，同时根据正确的跳转目标重新取指。如果采用的是 reorder-buffer，取消的粒度是指令；如果采用的是 branch-queue，取消的粒度是基本块。 **概念题**

第九章 参考答案 功能部件

1. 解：

a). 补码: 分别表示-0x70104000 和 0
b). 无符号数；分别表示 0x8FEFC000 和 0
c). 单精度浮点数；分别表示$-(1.110111111)_b \times 2^{31-127}$ 和+0.0
d). 一条 MIPS 指令：分别表示 LW R15, 0xC000(R31)以及 NOP 指令 **查阅资料题**

2. 解： 第二问先行进位记得算 pg 和全加器的延迟。

a). 串行进位加法器；
每一级进位传递的延迟为 2T，因此生成 c16 需要 32T；
每一级产生结果的延迟为 3T，因此生成 s15 需要（30T+3T）=33T

b). 先行进位加法器
参照课本图 9.6 的方式搭建：组内并行、组间并行，4 位一组。
延迟共 2T（产生 p、g）+2T（产生每组 P、G）+2T（产生组间进位）+2T（产生组内进位）+ 3T（全加器逻辑）= 11T

c). 先行进位加法器快的原因是它能更快地生成第 i 位的 c 而不需要依赖于第 i-1 位的 c。 **概念题**

3. 解： 部分同学对第二问“加法树”的理解有歧义，题目中想说这是华莱士树

a). 使用多个加法器；

采用先行进位加法器两两相加，需要 2*11T=22T 延迟

b). 使用加法树及加法器。

使用加法树把四个数相加变成两个数相加，需要 2 级全加器延迟（6T），然后再使用先行进位加法器（11T）得到最后结果，因此共 6T+11T=17T 延迟。

4. 证明：

假定带符号数 x, y, x+y 都在 n 位数表示范围之内，由于求补的本质是取模运算，则它们的补码可以如下表示：$[x]_{补} = 2^{n+1} + x$, $[y]补 = 2^{n+1} + y$, $[x+y]补 = 2^{n+1} + x + y$ ，其中第 n+1 位舍弃。于是:

$[x]补 + [y]补 = 2^{n+1} + x + 2^{n+1} + y = 2^{n+1} + (2^{n+1} + x + y) = 2^{n+1} + [x+y]补 = [x+y]补$（第 n+1 位舍弃）

■

5. Verilog 题

```
module add16(a, b, cin, out, cout);
    input    [15:0]   a;
    input    [15:0]   b;
    input             cin;
    output   [15:0]   out;
    output            cout;

    wire [15:0] p = a|b;
    wire [15:0] g = a&b;
    wire [3:0]    P, G;
    wire [15:0]   c;

    assign c[0] = cin;

    C4 C0_3(.p(p[3:0]),.g(g[3:0]),.cin(c[0]),.P(P[0]),.G(G[0]),.cout(c[3:1]));
    C4 C4_7(.p(p[7:4]),.g(g[7:4]),.cin(c[4]),.P(P[1]),.G(G[1]),.cout(c[7:5]));
    C4 C8_11(.p(p[11:8]),.g(g[11:8]),.cin(c[8]),.P(P[2]),.G(G[2]),.cout(c[11:9]));
    C4 C12_15(.p(p[15:12]),.g(g[15:12]),.cin(c[12]),.P(P[3]),.G(G[3]),.cout(c[15:13]));
    C4 C_INTER(.p(P),.G(G),.cin(c[0]),.P(),.G(),.cout({c[12],c[8],c[4]}));

    assign cout = (a[15]&b[15]) | (a[15]&c[15]) | (b[15]&c[15]);
    assign out = (~a&~b&c)|(~a&b&~c)|(a&~b&~c)|(a&b&c);
endmodule
```

```
module C4(p,g,cin,P,G,cout)
    input    [3:0]    p, g;
    input             cin;
    output            P,G;
    output   [2:0]    cout;
```

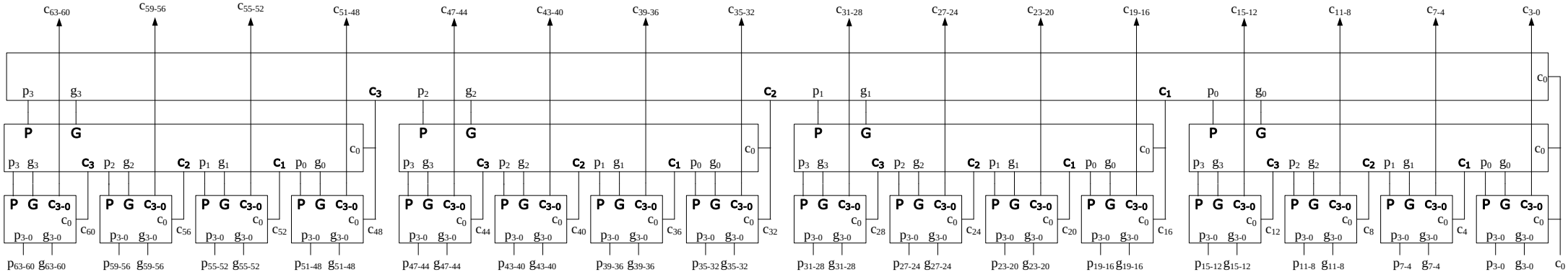
```
    assign P=&p;
    assign G=g[3]|(p[3]&g[2])|(p[3]&p[2]&g[1])|(p[3]&p[2]&p[1]&g[0]);

    assign cout[0]=g[0]|(p[0]&cin);
    assign cout[1]=g[1]|(p[1]&g[0])|(p[1]&p[0]&cin);
    assign cout[2]=g[2]|(p[2]&g[1])|(p[2]&p[1]&g[0])|(p[2]&p[1]&p[0]&cin);
endmodule
```

6. 扩展单精度 扩展双精度，网上有不同的材料，确实说法不一样。这个参考答案取课本中表 3.2 的说法。

参数	格式			
	单精度	扩展单精度	双精度	扩展双精度
尾数位宽 P	23	>=31	52	>=63
指数最大值 Emax	127	>=1023	1023	>=16383
指数最小值 Emin	-126	1-Emax	-1022	1-Emax
指数偏移量 Bias	127	Emax	1023	Emax
指数位数	8	>=11	11	>=15
浮点格式宽度	32	>=43	64	>=79

7.


c63-60 c59-56 c55-52 c51-48 c47-44 c43-40 c39-36 c35-32 c31-28 c27-24 c23-20 c19-16 c15-12 c11-8 c7-4 c3-0
p3 g3 c3 p2 g2 c2 p1 g1 c1 p0 g0 c0
P G c3-0
p3-0 g3-0
p63-60 g63-60 p59-56 g59-56 p55-52 g55-52 p51-48 g51-48
p47-44 g47-44 p43-40 g43-40 p39-36 g39-36 p35-32 g35-32
p31-28 g31-28 p27-24 g27-24 p23-20 g23-20 p19-16 g19-16
p15-12 g15-12 p11-8 g11-8 p7-4 g7-4 p3-0 g3-0

正确性证明略。

8. 答： **概念题**

16 个数相加的华莱士树，按照课本上的画法，共有 14 个进位输入；最后的全加器有一个 cin；最后加法器中，C 和 S 是错位相加的，因此 C 的低位还有一个空位。共计 16 个位置。

第十章 参考答案 高速缓存

1、解： 参考课本内容即可

页大小为 4KB=2^{12}B，页内地址为[11:0]位。

cache 容量 64KB=2^{16}B，地址范围为[15:0]；cache 块大小为 32B=2^{5}B，地址范围为[4:0]。

1）直接相联：

cache 索引位数为地址的[15:5]，需要页着色的是地址[15:12]，共 4 位；

2）二路组相联：

cache 索引位数为地址的[14:5]，需要页着色的是地址[14:12]，共 3 位；

3）四路组相联：

cache 索引位数为地址的[13:5]，需要页着色的是地址[13:12]，共 2 位。

2、解： 代价、损耗等词语，一看就是描述额外开销

$MissPenalty_{L1}=HitTime_{L2} + MissPenalty_{L2} \times MissRate_{L2}$

1）直接相联：

$MissPenalty_{L1}=4 + 60 \times 25\% = 19$ 个时钟周期

2）2 路组相联：

$MissPenalty_{L1}=5 + 60 \times 20\% = 17$ 个时钟周期

3）4 路组相联：

$MissPenalty_{L1}=6 + 60 \times 15\% = 15$ 个时钟周期

3、解： 哪个缺失率高，可以直接按 1.5+38 和 40 进行比较；但是如果算各个 cache 的访问缺失率，则需要按下面的方式算：

1）MissRate = CacheMissOps/MemOps

每 1000 条指令中 load/store 指令的条数是 1000×(26%/74%)=351

32KB 指令 cache MissRate 为 0.0015；

32KB 数据 cache MissRate 为 38/351=0.1083

指令 cache 和数据 cache 各 32KB 的 MissRate = (1.5 + 38)/(1000+351) = 0.0292

64KB 一体 cache 的 MissRate = 40/(1000+351) = 0.0296

所以指令 cache 和数据 cache 各 32KB 组织方式缺失率更低

2）AMAT 的定义存在歧义，参考答案提供其中一种算法：实际执行时间 除以 总访问次数

指令 cache 和数据 cache 各 32KB 的 AMAT = (1000+1.5×100+38×100)/(1000+351)=3.66

64KB 一体 cache 的 AMAT = (1000+351+40×100)/(1000+351)=3.96

4、解： 这个题歧义比较严重。 这个题出的不好， 1.内存访问序列本应是字节为单位，但是参考答案按照“字”为单位。 2.区分容量缺失和冲突缺失的概念不合理：容量缺失和冲突缺失的区分，更适合用来在“缺失率”的意义上做，而非针对某次访问到底是什么类缺失。

地址访问序列（单位：字）	访问类型	
	直接相联	2 路组相联
0	强制缺失	强制缺失
1	命中	命中
2	命中	强制缺失
3	命中	命中
4	强制缺失	强制缺失
15	强制缺失	强制缺失
14	命中	命中
13	命中	强制缺失
12	命中	命中
11	强制缺失	强制缺失
10	命中	命中
9	命中	强制缺失
0	命中	命中
1	命中	命中
2	命中	命中
3	命中	命中
4	命中	命中
56	强制缺失	强制缺失
28	强制缺失	强制缺失
32	强制缺失	强制缺失
15	容量缺失	命中
14	命中	命中
13	命中	冲突缺失
12	命中	命中
0	容量缺失	冲突缺失
1	命中	命中
2	命中	命中
3	命中	命中

5、答： **概念题**

包含式 cache 关系中一级 cache 的内容是二级 cache 内容的真子集；非包含式 cache 关系中一级 cache 的内容与二级 cache 内容无交集。

1）一级 cache 缺失后查找二级 cache：

包含式 cache 关系中，若二级 cache 命中则直接返回，若二级 cache 也缺失，则从更低层次的存储中取回所需数据，填入二级 cache 并返回至一级 cache。在此过程中，二级 cache 无论命中与否均存在一次读访问过程，若不命中，则存在一次替换操作（一次读访问过程读出被替换 cache 块，一次写访问过程写入回填 cache 块）。

非包含式 cache 关系中，若二级 cache 命中则在将命中的 cache 块返回给一级 cache 的同时将该 cache 块从二级 cache 中无效，若二级 cache 也缺失，则从更低层次的存储中取回所需数据，直接返回至一级 cache。在此过程中，二级 cache 无论命中与否均存在一次读访问过程，二级 cache 命中时存在一次写访问过程，不命中时则无其它访问过程。

2）一级 cache 替换出的 cache 块：

包含式 cache 关系中，替换出的 cache 块若不是脏块，则数据不用写回二级 cache，若是脏块，则直接写回二级 cache 的对应位置。在此过程中，若替换出的不是脏块，则一级 cache 只需要将替换出的地址通知一致性管理部件即可，一级 cache 与二级 cache 间无数据通信发生。

非包含式 cache 关系中，替换出的 cache 块无论是否是脏块，都要写入二级 cache。在将此 cache 块写入二级 cache 时，可能先要替换出一个 cache 块。在此过程中，一级 cache 与二级 cache 间有数据通信发生。

3）硬件实现复杂性方面，略。

4）多核之间一致性维护：（此处考虑二级 cache 被多个处理器核共享的情况）

如果采用基于目录的协议：

包含式 cache 关系中，因所有一级 cache 的内容均在二级 cache 中有备份（可能数据是旧值），所以二级 cache 处可以获得足够的信息来维护多核间的 cache 一致性。而在非包含式 cache 关系中，一级和二级 cache 上发生的所有 cache 块操作的事件的消息都要传递给目录进行维护。

如果采用基于侦听的协议：

包含式 cache 关系中，每个核向外广播的无效请求或更新请求也要传递给二级 cache。而在非包含式 cache 关系中，每个核向外广播的无效请求或更新请求则不必传递给二级 cache。

6、答： **概念题**

1）几乎没有空间局部性和时间局部性：数据库查询操作中被查询的数据

2）较好的空间局部性，几乎没有时间局部性：音视频编解码等流式应用被处理的数据

3）较好的时间局部性，很少的空间局部性：以指针链表、树、图等数据结构开发的应用程序。

4）较好的空间和时间局部性的应用：针对 cache 优化后的矩阵乘法、大量的科学计算、压缩解压缩、加解密、文字处理

7、答： **概念题**

降低 cache 缺失代价的技术：1）读优先；2）关键字优先；3）写合并；4）Victim Cache；5）多级 cache。（降低原因参考教材 P191~192，略）

降低 cache 缺失率的技术：1）增加 cache 容量；2）增加 cache 块大小；3）提高 cache 相联度；4）软件优化。（降低原因参考教材 P188~189，略）

8、答： **概念题**

1） 直接相联：每个内存块只能映射到 cache 的唯一位置；

全相联：每个内存块可以映射到 cache 的任一位置；

组相联：每个内存块对应 cache 的唯一的一个组，但可以映射到组内的任一位置。

2） 直接相联：根据访问地址将唯一映射位置上的 cache 块读出，将地址高位与 tag 比较看是否相等；

全相联：将 cache 中所有 cache 块读出，将地址高位与每一项的 tag 比较，看哪一项相

等；

组相联：根据访问地址将唯一映射的组内的所有 cache 块读出，将地址高位与每一项的 tag 比较，看哪一项相等。

3）LRU、Random、FIFO。

4）写穿透策略和写回策略；写分配策略和写不分配策略。

第十一章 参考答案 存储管理

1. 在如下一段 C 语言程序的 for 循环中，

```
void cycle(double* a) {
  int i;
  double b[65536];
  for(i=0; i<3; i++) {
   memcpy(a, b, sizeof(b));
  }
}
```

程序memcpy函数执行过程中可能发生哪些例外，各多少次。

答： 题目中没有给TLB项数和替换算法，参考答案按项数足够计算，但各位同学需要掌握项数不足情况的计算公式。题目中没有考虑modify例外和a数组已在父函数进行过访问的情况，题干不是很严谨。

【说明：此题中试图问的是与页处理相关的例外（忽略实际系统中可能发生的其他内部外部中断），以MIPS体系结构为例，包括三类共四种例外：

(1) TLB refill 例外；

(2) TLB invalid Load例外 / TLB invalid Store例外；

(3) TLB modify 例外。

其中(2)和(3)两类例外仅与程序对于页的访问需求相关，因此回答次数时需要假设操作系统中页的大小；而(1)还与处理器中TLB的容量相关，因此还需要假设处理器中TLB的项数和TLB的替换策略】。

下面给出一种常用假设条件下的回答：

操作系统页大小为4KB，同时系统的物理内存足够大，即页表分配的物理内存在程序执行过程中不会被交换到swap区上。并且在后续的分析中忽略代码和局部变量i所在的页的影响。a、b两数组大小均为65536×8=512KB。再假设a、b两数组的起始地址恰为一页的起始地址。

那么a、b两数组各自均需要512KB/4KB=128个页表项。

所以a、b的访问造成的TLB invalid 例外次数为128+128=256次；

假设TLB表项为128项，每项映射连续的偶数奇数两个页，采用LRU算法。那么第一次次循环中，a、b两数组每访问相邻偶数奇数两页的首地址时，均会触发一次TLB refill例外。后续各次循环TLB中均命中。那么共触发的TLB refill例外次数为128/2 + 128/2 = 128次。

2. 对于指令 Cache 是否有必要考虑 Cache 别名问题？

答： **概念题**

有必要。

1、当程序中有自修改代码时，可能会由于 cache 别名导致取错指令。

2、由于 Last Level Cache 通常是用物理地址索引的，那么当一级指令 cache 存在别名问题时，整个处理器无法维护一级指令 cache 和 Last Level Cache 之间相同物理地址的数据备份之间的关系，导致处理器进入混乱状态，最终导致程序执行错误甚至死机。

3. 假定在某一个 CPU 的 Cache 中需要 64 位虚拟地址，8 位的进程标识，而其支持的物理内存最多有 64GB。请问，使用虚拟地址索引比使用物理地址作为索引的 Tag 大多少？这个值是否随着 Cache 块大小的变化而发生改变？

答：参考课本内容即可得

1、由于虚拟地址有 64 位，进程标识为 8 位；而物理地址是 64GB，即需要 36 位物理地址。这样，使用虚拟地址比使用物理地址总共增加的位数为 8＋（64－36）＝8＋28＝36 位。
2、改变块的大小，亦或改变 cache 的其它参数，cache 占据的地址的低位的长度对于虚地址索引和物理地址索引都是一样的，所以对于 cache 的 tag 而言，两中索引方式相差位数保持不变。

4. 考虑一个包含 TLB 的当代处理器。（1）请阐述 TLB、TLB 失效例外、页表和 Page fault 之间的关系。（2）如果有这样一个机器设计：对于同样的虚拟地址，TLB 命中和 Page fault 同时发生，这样的设计合理么？为什么？（3）现代计算机页表普遍采用层次化的方式，请解释原因。

答：概念题。参考答案给出一个合理解答：（在另一种解答中，TLB 命中指的是硬件上 TLB 查询 VPN 命中。而 TLB refill 填回一个 V=0 的非法页，然后才会触发 page fault，因此此时 TLB 命中（但是 V 为 0）且触发 page fault。第二问如何回答，与“page fault”、“TLB 命中”的概念定义有关。）

1） **TLB**：Translation lookaside buffer，即旁路转换缓冲，或称为页表缓冲；里面存放的是一些页表文件（虚拟地址到物理地址的转换表）的一个子集。

TLB 失效例外：假如某个虚地址 VA，在 TLB 中没有找到对应的页表项，则发生 TLB 失效

例外(TLB miss exception)。

页表：简单的说，页表是内存块的目录文件，作用是实现从逻辑页号到物理块号的地址映射。即用指定大小（称之为页）来划分程序逻辑地址空间和处理机的物理内存空间，并建立起从逻辑页到物理页的**页面映射**关系，这些页面映射就是一张张页表（Page Table）。

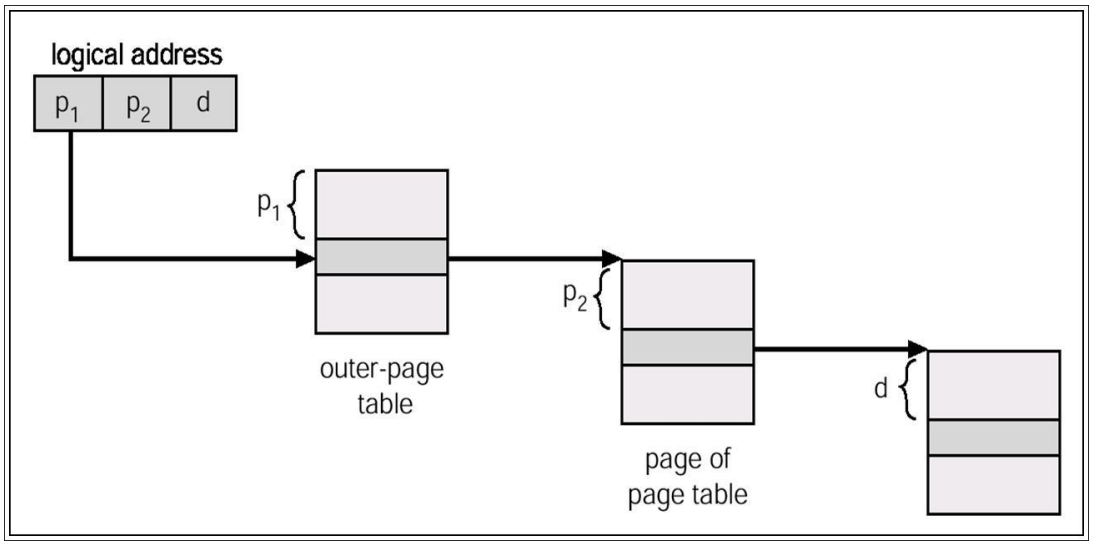
Page Fault：缺页，就是 TLB 中没有虚地址 VA 对应的页表，主存中也没有 VA 对应的物理地址。这时候发生 Page Fault Exception（缺页例外），需要交给操作系统处理，通常需要消耗很多的时间来处理该例外。

2) 不合理，因为 Page Fault 发生时意味着主存中没有虚地址 VA 对应的页表，这时 TLB 中也应该没有，因为 TLB 里的页表是主存中页表的子集，应该发生 TLB miss。倘若没有发生，说明 TLB 不是主存页表的子集，这会导致存储不一致，也违背了层次化设计的基本要求。

3) 多级页表可以减少页表所占用的存储空间。比如：

一个 32 位逻辑地址空间的计算机系统，页大小为 4KB，其虚地址可以被分为：20 位的页号 + 12 位的偏移。那么单级页表的条目有 2^{20} 条。假设每个条目占 4B，则需要 4MB 连续物理地址空间来存储页表本身。

如果采用两级页表，假设将 20 位的页号分解为：10 位的外部页表号 + 10 位的外部页表偏移。如下图所示，逻辑地址中：p1 用来访问外部页表的索引，p2 是外部页表的页偏移。其中第一级页表需要 4KB 连续物理地址空间，第二级页表每张表也需要 4KB 连续物理地址空间，由于通常情况下第一级页表中并非每一项都需要相应的第二级页表，所以总的存储空间要少于单级页表。



5. 已知一台计算机的虚地址为 48 位，物理地址为 40 位，页大小为 16KB，TLB 为 64 项全相联，TLB 的每项包括一个虚页号 vpn，一个物理页号 pfn，以及一个有效位 valid，请根据如

下模块接口写出一个 TLB 的地址查找部分的 Verilog 代码。

module tlb_cam(vpn_in, pfn_out, hit,…);

其中 vpn_in 为输入的虚页号，pfn_out 为输出的物理页号，hit 为表示是否找到的输出信号，“…”表示与该 TLB 输入输出有关的其他信号。重复的代码可以用“…”来简化。

答： Verilog 题

```
module tlb_cam(vpn_in, pfn_out, hit, valid_out);

input [33:0] vpn_in;
output [25:0] pfn_out;
output hit;
output valid_out;

reg[60:0] cam_content [63:0];//[60:60] valid [59:34] pfn [33:0] vpn
wire [63:0] entry_hit;

assign entry_hit[0] = (vpn_in==cam_content[0][33:0]);
assign entry_hit[1] = (vpn_in==cam_content[1][33:0]);
……
assign entry_hit[63] = (vpn_in==cam_content[63][33:0]);

assign hit = |entry_hit;

assign pfn_out =  {26{entry_hit[ 0]}} & cam_content[0][59:34] |
                  {26{entry_hit[ 1]}} & cam_content[1][59:34] |
……
                  {26{entry_hit[63]}} & cam_content[63][59:34];

assign valid_out = entry_hit[ 0] && cam_content[ 0][60] ||
                   entry_hit[ 1] && cam_content[ 1][60] ||
                   …
                   entry_hit[63] && cam_content[63][60];
endmodule
```

第十二章 参考答案 多处理器

1、答： **用 C 写的算法：（不在本课考察范围内）**

（1） Filter Lock 算法：

```
int level[p];
int victim[p]; //use 1..n-1
for(int i = 0; i < p; ++i)
```

```
{
    level[i] = 0;
}

void lock(int tid)
{
    for (int i=1; i<p; i++)
    {
        level[tid] = i;
        victim[i] = tid;
        while (same_or_higher(tid, i) && victim[i]==tid);
    }
}

void unlock(int tid)
{
    level[tid] = 0;
}

bool same_or_higher(int tid, int i)
{
    for (int k=0; k<p; k++)
        if (k!=tid && level[k] >= i) return true;
    return false;
}
```

（2）Lamport's bakery 算法：

```
int number[p];
bool entering[p];
for(int i = 0; i < p; ++i)
{
    number[i] = 0;
    entering[i] = false;
}

void lock(int tid)
{
    entering[tid] = true;
    number[tid] = 1 + array_max(number); //find max in the array
    entering[tid] = false;
    for (int i=0; i<p; i++)
    {
        if (i != tid)
        {
```

```
            // Wait until thread i receives its number
            while (entering[i]);
            // Wait until all threads with smaller numbers
            // or with the same number, but with higher priority,
            // finish their work
            while ((number[i]!=0) && (number[tid]>number[i] ||
                    (number[tid]==number[i] && tid>i)));
        }
    }
}

void unlock(int tid)
{
    number[tid] = 0;
}
```

（3）最少需要访问 p 个内存地址。假设只需要 i 个地址，存在某个时刻，p 个进程都要向这 i 个地址写以表明自己想获得锁。如果 i<p, 必然有某个进程 p_j 希望获得锁的意图会被其它进程的写覆盖掉，从而其它进程不知道 p_j 希望获得锁。这样就会产生不公平。

我用 LLSC 汇编写的（有很多种参考答案，以下只是一种）:（请掌握其原理）

(1) 简单自旋锁，设 a0 为锁的地址

```
Lock:
  1:  LL      t0, 0x0(a0)    //取回锁值，0 表示未被锁，1 表示有人持有锁
      BNEZ    t0, 1b         //如果已经被锁，则自旋等待
      ADDIU t0, t0, 0x1      //将 t0 改为已持有
      SC      t0, 0x0(a0)    //存回
      BEQZ    t0, 1b         //检查是否成功，否则重做
      NOP

Unlock:
      //mem fence
      SW    $0, 0x0(a0)   //解锁时写 0
      //mem fence
```

(2) 简单自旋排队锁　设 a0 为锁的地址。按请求顺序来 service。

```
Lock:  (锁值为高低各 2 字节组成，高位为 tickets, 低位为 serve)
  1: LL      v0, 0x0(a0)     //同时取回高低位
     LI      t0, 0x10000     //将高位 tickets 加一
     ADDU    t0, t0, v0
     SC      t0, 0x0(a0)     //存回去
     BEQZ    t0, 1b          //检查是否 LLSC 成功，失败则重试
     NOP
```

```
    SRL     v0, v0, 16      //取出此时已经确定拿到的 tickets，放到低位
2:  LHU     t0, 0x0(a0)     //检查当前 serve 值
    BNE     t0, v0, 2b      //如果当前 serve 不是自己，则循环等待
    NOP

Unlock：
    //mem fence
    LHU     t0, 0x0(a0)   //取出当前 serve
    ADDIU   t0, t0, 0x1
    SH      t0, 0x0(a0)   //加 1 存回去
    //mem fence
```

(3) 需要两个内存地址，一个记录当前服务者，另一个记录当前排队最晚号码；同时，每个进程需要自己使用寄存器或内存地址，记录自己拿到的号码。

2、答： **概念题+MIPS 汇编代码题**

（1）Compare_And_Swap(CAS)和 Load-Linked and Store-Conditional(LL/SC)是两种比较常见的硬件同步原语。例如，Intel、AMD 的 x86 指令集和 Oracle/SUN 的 Sparc 指令集实现了 CAS 指令；Alpha、PowerPC、MIPS、ARM 均实现了 LL/SC 指令。

（2）CAS 指令硬件实现比 LL/SC 的硬件实现复杂。使用 CAS 指令会碰到 ABA 问题，但是 LL/SC 指令不会碰到该问题。LL/SC 指令中由于 SC 是尝试去写，因此在某些情况下，SC 执行成功率很低，导致用 LL/SC 实现的锁执行开销变得很大。

（3）下面仅给出用 CAS 和 LL/SC 实现普通自旋锁的例子。公平的自旋锁可利用这些普通的自旋锁实现。

CAS：指令定义 cas r1, r2, Mem；r1 存放期望值，r2 存放更新值，

获得锁：//锁初始化为 0

```
    la      t0, sem
TryAgain:
    li      t1, 0
    li      t2, 1
    cas     t1, t2, 0x0(t0)
    bnez    t1, TryAgain
    nop
```

释放锁：

```
    la      t0, sem
    li      t1, 0
    sw      t1, 0x0(t0)
```

LL/SC：

获得锁：//锁初始化为 0

```
    la      t0, sem
TryAgain:
    ll      t1, 0x0(t0)
    bnez    t1, TryAgain
    li      t1, 1
```

```
    sc      t1, 0x0(t0)
    beqz    t1, TryAgain
    nop
```

释放锁：

```
    la      t0, sem
    sw      zero, 0x0(t0)
```

（4）LL/SC 需要 n 个地址才能实现公平。CAS 只需要两个地址。最主要的是保证每个进程希望获得锁的信息不会被覆盖掉。CAS 可以保证，而 LL/SC 可能会失败。

3、答： **概念题**

（1）n 个共享存储处理器，每个有 m 内存。进行 O(nm)个变量排序就可能出现超线性加速比。一些随机算法也可能出现。很多搜索算法也会有超线性加速比，例如通过剪枝使得总的被搜索数据量少于顺序搜索的情况，此时并行执行将完成较少的工作。

（2）Amdahl 定律定义是：使用某种较快执行方式获得的性能提高，受到可受这种较快执行方式的时间所占比例的限制。Speedup = 1 / ((1-p)+k*p)。加速因子 k 可以比 1/n 增长快，即所谓超线性。因此并不矛盾。

4、答：**拓展题**

避免多个处理器同时写的不同变量处于同一 cache 行上。

5、答： **概念题**

（1）可能 P1 已经把 A 替换出去了，随后 P1 又希望访问 A，因此向目录发出了 A 的 miss 请求。如果网络不能保证 P1 发出的替换 A 消息和 miss 访问 A 消息达到目录的顺序，后发出的 A 的 miss 请求越过先发出的 A 的替换请求，先到达目录，就会产生上述现象。

（2）两种可行的处理方式：a)网络保证点到点消息的顺序性；b)目录发现不一致时，暂缓 miss 请求的处理，等待替换消息到达后，目录状态正确后，再返回 miss 请求的响应。

6、答： **请掌握 MSI 一致性协议。**

事件	A 状态	B 状态
初始	I	I
CPU A 读	S	I
CPU A 写	M	I
CPU B 写	I	M
CPU A 读	S	S

7、答： **概念题**

（1）P1 对 A 的写被网络阻塞，一直没有传播到 P2 和 P3。

（2）无须施加限制。

（3）

```
P1                  P2                  P3
                    barrier1            barrier1
                                        barrier2
A=2000              while (B!=1) {;}    while (C!=1) {;}
B=1                 C=1;                D=A
```

barrier1　　barrier2

barrier2

8、答：　**概念题**

（1）顺序一致性：

i.a=1 先于 print a 执行，b=1 先于 print b 执行。最终结果：a=1, b=1

ii.a=1 先于 print a 执行，print b 先于 b=1 执行。最终结果：a=1, b=0

iii.print a 先于 a=1 执行，b=1 先于 print b 执行。最终结果：a=0, b=1

（2）弱一致性：

除顺序一致性中的三种执行序外，还可以有第 iv 中执行序：

print a 先于 a=1 执行，print b 先于 b=1 执行。最终结果：a=0, b=0