1、请根据 CPU 性能公式，提出几种提高 CPU 速度的方法。

(课本 1.9 节)处理器性能公式

用时钟周期的持续时间(如 1ns)或其频率(如 1GHz)来描述时钟周期的时间。程序的 CPU 时间可以有两种表示方法：

CPU 时间=程序的 CPU 时钟周期数×时钟周期时间 或

CPU 时间=程序的 CPU 时钟周期数/时间频率

所执行的指令数：指令路径长度或指令数(IC)。

每条指令时钟周期数(CPI)=程序的 CPU 时钟周期数/指令数。每时钟周期指令数(IPC)，是 CPI 的倒数。

CPI：每条指令的时钟周期数，表示执行某个程序或者程序片段时每条指令所需的时钟周期平均数。

时钟周期可定义为 IC×CPI

CPU 性能公式：CPU 时间=IC×CPI×时钟周期时间

$$\frac{\text{指令}}{\text{程序}}\times\frac{\text{时钟周期}}{\text{指令}}\times\frac{\text{秒}}{\text{时钟周期}}=\frac{\text{秒}}{\text{程序}}=CPU\text{时间}$$

按测试单位展开

处理器性能取决于三个特性：时钟周期（或时钟频率）、每条指令的时钟周期和指令数。

提高 CPU 速度的方法:

方法一:提高 CPU 的频率，缩短时钟周期时间。

方法二:采用较科学的编程方法，减少指令数。

[例题]假设已经进行以下测量：FP 操作频率=25%；FP 操作的平均 CPI=4.0；其他指令的平均 CPI=1.33；FPSQR 的频率=2%；FPSQR 的 CPI=20。假定有两种设计方案，一种方案将 FPSQR 的 CPI 降至 2，一种是把所有的 FP 操作的 CPI 降至 2.5。请使用处理器性能公式对比这两种设计方案。

解：首先求出没有任何改进时的原 CPI

$$CPI_{原}=\sum_{i=1}^{n}CPI_i\times\left(\frac{IC_i}{\text{指令数}}\right)=(4\times0.25)+(1.33\times0.75)=2.0$$

从原 CPI 中减去节省的周期数就可以求出改进 FPSQR 后的 CPI：

$$CPI_{采用新FPSQR}=CPI_{原}-0.02\times(CPI_{旧FPSQR}-CPI_{新FPSQR})=2.0-0.02\times(20-2)=1.64$$

可以采用相同方式来计算对所有 FP 指令进行改进后的 CPI，也可以将 FP 和非 FP CPI 相加。采用后一种方法，将得到：

$$CPI_{新FP}=0.75\times1.33+0.25\times2.5=1.625$$

由于总 FP 改进的 CPI 稍低一些，所以它的性能也稍好一点。具体来说，总 FP 改进的加速比为：

$$\text{加速比}_{新FP}=\frac{CPU\text{时间}_{原}}{CPU\text{时间}_{新FP}}=\frac{IC\times\text{时钟周期}\times CPI_{原}}{IC\times\text{时钟周期}\times CPI_{新FP}}=\frac{CPI_{原}}{CPI_{新FP}}=\frac{2.00}{1.625}=1.23$$

Ex1.假设 15%的指令是 load，令 20%的指令跟在 load 后面暂停 1 cycle，因为需要 load 的结果，所有的指令和 loads 都命中 cache。0.15×0.2×1=0.03

Ex2.假设 20%的指令是跳转指令，其中 60%实施跳转，跳转代价为 3 cycles，另 40%没有实施跳转，跳转代价为 2 cycle。

20%×60%×3+20%×40%×2=0.52

Ex3.假定 cache 为理想状态，CPI 为 2.0，时钟周期时间为 1.0ns，平均每条指令访问存储器 1.5 次，另外由于增加组相联后，增加 cache 访问的复杂性，因此 2 路组相联的命中时间拓展为原来的 1.25 倍，两个 cache 的容量都是 64KB，块容量是 64 字节，一个 cache 采用直接映射，另一个 cache 采用 2 路组相联映射，命中时间均为 1 个时钟周期，并且假定直接映射和 2 路映射组相联的 cache 缺失率为 1.4%和 1.0%。两者的缺失代价都为 75ns。

平均存储访问时间=命中时间+缺失率×缺失代价

1 路=1.0+(0.014×75)=2.03ns

2 路=1.0×1.25+(0.01×75)=2.00ns

可见，2 路组相联的内存访问性能更好。

那内存访问性能好，是否意味着 CPU 的性能好呢?

CPU 时间=执行指令数×(指令执行周期+确实次数×缺失代价/指令数) ×时钟周期时间

=执行指令数×[(指令执行周期数×时钟周期时间)+(缺失率×(存储器访问次数/指令数) ×缺失代价×时钟周期时间)]

CPU 时间(1 路)= IC×(2×1.0+(1.5×0.014×75))=3.58×指令数

CPU 时间(2 路)= IC×(2×1.0×1.25+(1.5×0.010×75))=3.63×指令数

2 路组相联的处理器性能反而不如直接映射。

结论：性能的考察最终得从 CPU 时间入手。

2、举例说明什么是 Amdal 定理。

Amdal 定理:系统中对某一部件采用更快的执行方式所获得的性能改进程度，取决于这一方式所被使用的频率，或占总时间的比例。简而言之,通过更快的处理器所获得的加速会被慢的系统组件所限制。

$$\text{系统加速比}=\frac{1}{(1-\text{可加速部分比例})+\frac{\text{可加速部分比例}}{\text{理论加速比}}}$$

公式:S=1/(1-a+a/n)

a 表示系统中并行计算所占的比例，n 为并行处理结点个数。当 a=1 时，系统中全为并行计算，没有串行，此时 s=n，系统的加速比最大。a=0 时，系统中全为串行计算，没有并行操作，此时 s=1，加速比最小。当 n→∞时，极限加速比 s→ 1/（1-a），这也就是加速比的上限。

利用 Amdahl 定律，可以计算出通过改进计算机某一部分而获得的性能增益。Amdahl 定律表明，使用某种快速执行模式获得的性能改进受限于可使用此种快速执行方式的时间比例。

Amdahl 定律定义了使用某一特定功能所获得的加速比=整个任务在采用该升级时的性能/整个任务在未采用该升级时的性能或者 加速比=整个任务在未采用该升级时的执行时间/整个任务在采用该升级时的执行时间

Amdahl 定律为我们提供了一种快速方法，用来计算某一升级所得到的加速比，加速比取决于下面两个因素。

(1) 原计算机计算时间中可升级部分所占的比例。例如，一个程序的总执行时间为 60 秒，如果有 20 秒的执行时间可进行升级，那这个比例就是 20/60。我们将这个值称为升级比例，它总是小于或等于 1。

(2) 通过升级执行模式得到的改进，也就是说在为整个程序使用这一执行模式时，任务的运行速度会提高多少倍。这个值等于原模式的执行时间除以升级模式的执行时间。如果为程序的某一部分采用升级模式后需要 2 秒，而在原始模式中需要 5 秒，则提升值为 5/2。我们将这个值称为升级加速比，它总是大于 1。

原计算机采用升级模式后的执行时间等于该计算机未升级部分耗用的时间加上使用升级部分耗用的时间：

$$\text{新执行时间}=\text{原执行时间}\times(1-\text{升级比例})+\frac{\text{升级比例}}{\text{升级加速比}}$$

总加速比是这两个执行时间之比：

$$\text{总加速比}=\frac{\text{原执行时间}}{\text{新执行时间}}=\frac{1}{(1-\text{升级比例})+\frac{\text{升级比例}}{\text{升级加速比}}}$$

3、Web 服务器普遍采用如 Javascript、python、PHP 等动态脚本语言。为加速这些语言中的对象属性访问，采用了硬件机制来缓存这些属性，使得访问时间减小为原来的 1/10。问：

（1）假定对象属性访问指令时间占总时间 40%，则加速后系统性能提高了多少？（2）如果要使加速后系统性能提高 1 倍，则对象属性访问指令时间占总时间为多少？

[同(1)例题：假设我们希望升级一个用于提供 Web 服务器的处理器，新处理器执行 Web 服务应用程序的计算速度是原处理器的 10 倍。假定原处理器有 40%的时间忙于计算，60%的时间等待 I/O，进行这一升级后，所得到的总加速比为多少？]

（1）升级比例=0.4、升级加速比=10

$$\text{总加速比}=\frac{1}{0.6+\frac{0.4}{10}}=\frac{1}{0.64}\approx1.56$$

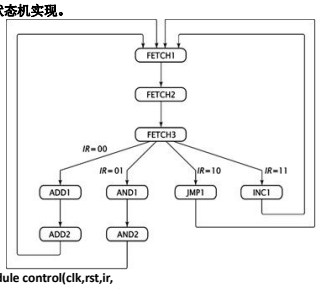
答：系统性能大约提高了 56%

（2）解：设升级比例为 x、升级加速比=10

$$\text{总加速比}=\frac{1}{(1-x)+\frac{x}{10}}=2$$

解得 $x=\frac{5}{9}\approx0.56$ 错误!未指定书签。错误!未指定书签。

4、请用 VerilogHDL 语言给出课内实验 2 中 4 指令处理器控制器的状态机实现。




```
module control(clk,rst,ir,
                membus,
                arload,
                pcload,pcinc,pcbus,
                drload,drbus,
                alusel,
                acload,acinc,
                irload,stateout
);
                parameter FETCH1 = 0;parameter FETCH2 = 1;
                parameter FETCH3 = 2;parameter ADD1 = 3;
                parameter ADD2 = 4;parameter AND1 = 5;
                parameter AND2 = 6;parameter INC1 = 7;
                parameter JMP1 = 8;parameter DECODE = 9;
                input clk,rst;
                input [1:0] ir;
                output     membus,arload,
                           pcload,pcinc,pcbus,
                           drload,drbus,
                           alusel,acload,acinc,irload;
                output [3:0] stateout;
                 reg [3:0] state,nextstate;
                 always @(posedge clk)
                 begin
                           if(rst)
                           state <= FETCH1;
                           else
                           state <= nextstate;
                 end
                 always @(state or ir)
                 begin
                           case(state)
                           FETCH1:nextstate = FETCH2;
                           FETCH2:nextstate = FETCH3;
                           FETCH3:nextstate = DECODE;
                 DECODE:
                           case(ir)
                 2'b00:nextstate = ADD1;
                 2'b01:nextstate = AND1;
                 2'b10:nextstate = INC1;
                 2'b11:nextstate = JMP1;
                 endcase
                 ADD1:nextstate = ADD2;
                 ADD2:nextstate = FETCH1;
                 AND1:nextstate = AND2;
                AND2:nextstate = FETCH1;
                 JMP1:nextstate = FETCH1;
                 INC1:nextstate = FETCH1;
                 default:nextstate = FETCH1;
                 endcase
                  end
   assign stateout = state;
     assign arload = (state == FETCH1 || state== FETCH3)?1:0;
     assign pcload = (state == JMP1)?1:0;
     assign pcinc = (state == FETCH2)?1:0;
     assign drload = (state == FETCH2 || state== ADD1 || state== AND1)?1:0;
     assign acload = (state == ADD2 || state== AND2)?1:0;
     assign acinc = (state == INC1 )?1:0;
     assign irload = (state == FETCH3)?1:0;
     assign alusel = (state == AND2)?1:0;
     assign membus = (state == FETCH2 || state== ADD1 || state== AND1)?1:0;
     assign pcbus = (state == FETCH1)?1:0;
     assign drbus = (state == FETCH3 || state== ADD2 || state== AND2 || state == JMP1)?1:0;
     endmodule
```

5、影响 Cache 性能主要的因素有哪些。

Cache 的容量；Cache 的控制算法；Cache 的结构 cache 相联度；

cache 块大小；不同的替换算法：LRU 和随机法 FIFO 等。

6、（1）什么是局部性原理。；

局部性原理: CPU 访问存储器时，无论是存取指令还是存取数据，所访问的存储单元都趋于聚集在一个较小的连续区域中。

三种不同类型的局部性:

时间局部性（Temporal Locality）：如果一个信息项正在被访问，那么在近期它很可能还会被再次访问。程序循环、堆栈等是产生时间局部性的原因。

空间局部性（Spatial Locality）：在最近的将来将用到的信息很可能与现在正在使用的信息在空间地址上是临近的。

顺序局部性（Order Locality）：在典型程序中，除转移类指令外，大部分指令是顺序进行的。顺序执行和非顺序执行的比例大致是 5:1。此外，对大型数组访问也是顺序的。

指令的顺序执行、数组的连续存放等是产生顺序局部性的原因。

（2）比较 Cache 块的三种放置方法优缺点。

（1）全相联映射方式：

优点：灵活，命中率比较高，Cache 存储空间利用率高

缺点：但映射函数复杂，不易实现，相联存储器庞大，比较电路复杂，访问相关存储器时，每次都要与全部内容比较，速度低，成本高，因而只适合于小容量的 Cache 之用，应用少。

（2）直接映射方式：

优点：比较电路最简单，地址映射方式简单，数据访问时，只需检查区号是否相等即可，因而可以得到比较快的访问速度，快速，硬件设备简单。

缺点：Cache 块冲突率较高，余数相同的主存块无法同时进入 Cache，从而降低了 Cache 的利用率，效率不高，替换操作频繁，命中率比较低，易颠簸。

（3）组相联映射方式：

优点：块的冲突概率比较低，块的利用率大幅度提高，块失效率明显降低。

缺点：实现难度和造价要比直接映射方式高。

7、举例说明流水线中的几种冲突。

流水线冲突有三种类型：结构冲突、数据冲突、控制冲突。

（1）结构冲突：在重叠执行模式下，如果硬件无法同时支持指令的所有可能组合方式，就会出现资源冲突，从而导致结构冒险。例如，处理器可能仅有一个寄存器堆写端口，但在特定情况下，流水线可能希望在一个时钟周期内执行两个写操作，这就会生成结构冒险。为避免这一冒险，在发生数据存储器访问时，使流水线停顿一个时钟周期或停顿流水线直到冲突解决，停顿通常被称为流水线气泡，因为它会漂浮穿过流水线，占据空间却不执行有用工作；或者进行资源复制，使得指令、数据分离。

[例]假定数据引用占总体的 40%，流水化处理器的理想 CPI 为 1(忽略结构冒险)。假定与没有冒险的处理器相比，有结构冒险处理器的时钟频率为其 1.05 倍。不考虑所有其他性能损失，有结构冒险和无结构冒险相比，哪种流水线更快？快多少？

解：计算两种处理器的平均指令时间。由于没有停顿，所以理想处理器的平均指令时间就是时钟周期时间(理想)，有结构冒险处理器的平均指令时间为：

平均指令时间 = CPI × 时钟周期时间=(1+0.4×1)×时钟周期时间(理想)/1.05=1.3×时钟周期时间(理想)

显然，没有结构冒险的处理器更快一些，根据平均指令时间的比值，无冒险处理器的速度快 1.3 倍。

（2）数据冲突：根据流水线中的指令重叠，指令之间存在先后顺序，如果一条指令取决于先前指令的结果，就可能导致数据冒险。

分类：写后读(RAW)[真实相关]、写后写(WAW)、读后写(WAR)。

例子：DADD 之后的所有指令都用到了 DADD 指令的结果，DADD 指令在 WB 流水线写入 R1 的值，但 DSUB 指令在其 ID 级中读取这个值。这一问题成为数据冒险，除非提前防范这种问题，否则 DSUB 指令将会读取错误值并试图使用它。

```
DADD   R1,R2,R3   || DSUB   R4,R1,R5   || AND   R6,R1,R7
OR   R8,R1,R9   || XOR   R10,R1,R11
```

消除涉及 DSUB 和 AND 指令的冒险停顿

①利用转发技术将数据冒险停顿减至最少：转发(forwarding)是一种简单的硬件技术。工作方式如下 1.来自 EX/MEM 和 MEM/WB 流水线寄存器的 ALU 结果总是被反馈回 ALU 的输入端。2.如果转发硬件检测到前一个 ALU 操作已经对当前 ALU 操作的源寄存器进行了写入操作，则控制逻辑选择转发结果作为 ALU 输入，而不是选择从寄存器堆中读取的值。

(DSUB 和 AND 指令的输入是从流水线寄存器转发到第一个 ALU 输入。OR 接收的结果是通过寄存器堆转发而来的，只需要在周期的后半部分读取寄存器、在前半部分写入寄存器即可完成，如寄存器上的虚线所示。)

②需要停顿的数据冒险：(转发不能解决的情形)并非所有签在数据冒险都可以通过旁路方式处理，增加一种称为流水线互锁，以保持正确的执行模式。流水线互锁会检测冒险，并在该冒险被清除之前使流水线停顿。在这种情况下，互锁使流水线停顿，让希望使用某一数据的指令等待，直到源指令生成该数据为止。

```
LD   R1,0(R2)   || DSUB   R4,R1,R5
AND   R6,R1,R7   || OR   R8,R1,R9
```

(因为停顿会导致从 DSUB 开始的指令在时间上向后移动 1 个周期，转发给 AND 指令的数据现在是通过寄存器堆到达的，而对于 OR 指令根本不需要转发。由于插入了气泡，需要增加一个周期才能完成这一序列。4 号时钟周期内没有启动指令(在 6 号周期内没有指令完成)。)

（3）控制冲突：分支指令及其它改变程序计数器的指令实现流水化时可能导致控制冒险。例如，在执行分支时，修改后的程序计数器的值可能等于（也可能不等于）当前值加 4。如果分支计数器改为其目标地址，它就是选中分支；否则就是未选中分支。如果指令 i 为选中分支，通常会等到 ID 末尾，完成地址计算和对比之后才会改变程序计数器。

例子：不能把 IF 语句中的 THEN 部分中的一条指令拿出来，移到这个分支之前，因此会造成控制停顿。

解决：处理分支最简单的方法是：一旦在 ID 期间(此时对指令进行译码)检测到分支，就对该分支之后的指令重新取值。第一个 IF 周期基本上是一次停顿，因为它从来不会执行有用的工作。而如果每个分支产生一个停顿周期，将会使性能损失，应对这一损失的技术：1.降低流水线分支代价(预测)：①冻结或冲刷流水线，保留或删除分支之后的所有指令，直到直到分支目标为止。②假定转移不发生，继续取下一指令。如转移发生，则要取消上述结果。③根据统计信息预测下一指令，如 for 循环中，大多数情况下转移或不转移。④延迟分支，在条件指令后插入延迟时间槽。带有一个分支延迟的执行周期为：分支指令||依序后续指令||选中时的分支目标；局限性源于：①对于可排在延迟时隙中的指令有限制，②编译时预测一个分支是否可能被选中的能力有限。

8、对于以下代码序列，采用循环展开进行静态调度：

```
Loop:    L.D     F0,0(R1)    ;F0=数组元素
         ADD.D   F4,F0,F2    ;加上 F2 中的标量
         S.D     F4,0(R1)    ;存储结果
         DADDUI  R1,R1,#-8   ;使指针递减 8 个字节
         BNE     R1,R2,LOOP  ;R1!=R2 时跳转
```

生成结果的指令	使用结果的指令	延迟（以时钟周期为单位）
FP ALU 运算	另一个FP ALU运算	3
FP ALU 运算	存储双精度值	2
载入双精度值	FP ALU运算	1
载入双精度值	存储双精度值	0

其中浮点单元的延迟如下表（假定转移指令引起的延迟为 1 个时钟周期）：请回答：

（1） 请写出每循环所花费的时钟周期数。

			发射的时钟周期
Loop：	L.D	F0,0(R1)	1
	停顿		2
	MUL.D	F4,F0,F2	3
	停顿		4
	停顿		5
	S.D	F4,0(R1)	6
	DADDIU	R1,R1,#-8	7
	停顿		8
	BNE	R1,R2,LOOP	9

; branches if R1|R2 是注释不执行

所以每个循环花费 9 个时钟周期

（2） 采用 4 份循环展开，并进行调度优化，写出优化后的代码及所需每循环所需时钟周期数。

			发射的时钟周期
LOOP：	L.D	F0,0(R1)	1
	L.D	F6,-8(R1)	2
	L.D	F10,-16(R1)	3
	L.D	F14,-24(R1)	4
	MUL.D	F4,F0,F2	5
	MUL.D	F8,F6,F2	6
	MUL.D	F12,F10,F2	7
	MUL.D	F16,F14,F2	8
	S.D	F4,0(R1)	9
	S.D	F8,-8(R1)	10
	DADDIU	R1,R1,#-32	11
	S.D	F12,16(r1)	12
	S.D	F16,8(R1)	13
	BNE	R1,R2,Loop	14

展开后循环的执行时间缩减到总共 14 个时钟周期，每个元素 3.5 个时钟周期。

9、简述（1）动态指令调度；（2）Tomasulo 算法及主要解决的问题。

1）是由硬件在程序实际运行时实施的，其基本思想是对指令流水线互锁控制进一步改进，能实时的判断出是否有 WR、RW、WW 相关存在，利用硬件绕过或防止这些相关的出错，并允许多条指令在具有多功能部件的执行段中并行操作，从而提高流水线的利用率且减少停顿现象。

2）Tomasulo 算法是将寄存器数据提取到保留站中，如果原来寄存器中的操作数修改了，就将修改后的值提取到另外一个保留站中，以此类推。在命令发射的时候，会将待用操作数的寄存器说明符更名为它对应保留站的名字。

(课本 3.4 节用动态调度克服数据冒险)Tomasulo 算法通过对寄存器进行有效动态重命名来处理反相关(WAR)和输出相关(WAW)。它会跟踪指令的操作数何时可用，将 RAW 冒险降至最低，并在硬件中引入寄存器重命名功能，将 WAW 和 WAR 冒险降至最低。

Tomasulo 方案中的标签引用的是将会生成结果的缓冲区或单元，当一条指令发射到保留站之后，寄存器名称将会丢弃。

动态调度：示例和算法

[例题]对以下代码序列，写出在仅完成了第一条载入指令并已将其结果写到 CDB 总线时的信息表：

```
1.  L.D      F6,32(R2)
2.  L.D      F2,44(R3)
3.  MUL.D    F0,F2,F4
4.  SUB.D    F8,F2,F6
5.  DIV.D    F10,F0,F6
6.  ADD.D    F6,F8,F2
```

解：用 3 个表显示了其结果。当所有指令都已经被发射，但只有第一条载入指令已将完成而且已将其结果写到 CDB 时的保留站与寄存器标签。

指令	指令状态		
	发射	执行	写结果
L.D F6,32(R2)	√	√	√
L.D F2,44(R3)	√	√	
MUL.D F0,F2,F4	√		
SUB.D F8,F2,F6	√		
DIV.D F10,F0,F6	√		
ADD.D F6,F8,F2	√		

名称	保留站			
	繁忙	Op	Vj	Vk
Load1	否			
Load2	是	Load		
Add1	是	SUB		Mem[32+Regs[R2]]
Add2	是	ADD		
Add3	否			

Mult1	是	MUL		Regs[F4]
Mult2	是	DIV		Mem[32+Regs[R2]]

保留站		
Qj	Qk	A
		44+Regs[R3]
Load2		
Add1	Load2	
Load2		
Mult1		

	寄存器状态							
字段	F0	F2	F4	F6	F8	F10	…	F30
Qi	Mult1	Load2		Add2	Add1	Mult2		

Tomasulo 方案有两点优势。1.冒险检测逻辑的分布，2.消除了可能产生 WAW 和 WAR 冒险的停顿。

10、简述 V-MIPS 向量计算机的基本结构。

V-MIPS 的主要组成部分：

向量寄存器（Vector registers）：每个向量寄存器都是一个定长的寄存器组，能够容纳一个向量。VMIPS 有 8 个向量寄存器，每个寄存器能够容纳 64 个向量元素，每个元素宽度为 64 位。每个寄存器至少有两个读端口和一个写端口。这能够保证需要不同向量寄存器的多个向量运算能够互相同步进行[1]（我们并不考虑由于寄存器端口短缺而引起的问题。在实际的机器中，这会导致结构冲突）。总共 16 个读端口和 8 个写端口通过一对 crossbar（交叉交换器） 和功能单元的输入输出相连。

向量功能单元（Vector functional units）：每个单元都是完全流水化的，并且每一个新的时钟周期可以开始对于一个新向量元素的操作。另外还需要一个控制部件来检测 hazard，包括功能单元的结构冲突和寄存器访问的数据冲突。如图所示，VMIPS 有五个功能单元。为了简化讨论，我们将只讨论浮点运算单元。取决于不同的设计，标量运算可能使用向量功能部件，或者有单独的一组功能部件。我们这里假设功能部件是共享的，并且，再一次忽略任何可能的冲突。

向量访存（载入/存储）单元（Vector load-store unit）：这是一个能够 load 和 store 一个向量到主存的单元。VMIPS 中的 load 和 store 操作是完全流水化的，因此在一开始的延迟之后，寄存器和内存之间的带宽可以达到每一个时钟周期一个 word。这个单元通常也处理标量访存工作。

标量寄存器集合（scalar registers）：标量寄存器能为标量功能单元提供输入数据[3]，并且也能为内存访问单元提供地址。这些其实就是 MIPS 中常见的 32 个通用寄存器和 32 个浮点寄存器。标量数据从标量寄存器中读出，然后锁存到向量单元的一个输入之中。

11、并行处理系统的主要挑战有哪些？

第一个障碍与程序中有限的可用并行相关，第二个障碍源于通信的成本较高。

（1）芯片（2）可用性（3）系统管理（4）调试（5）性能预测（6）编程模型（7）功耗

12、机器学习中普遍采用 GPU 来进行计算加速，业界也提出了一些专用的计算架构来处理如深度学习中的大量计算，如 Google 提出针对 TensorFlow 计算框架的张量处理器（TPU）。请比较 GPU 和 TPU 的异同。

相同点：都是可以运行绘图运算工作的微处理器，两者都有总线和外界联系，有自己的缓存体系，以及数字和逻辑运算单元。两者都为了完成计算任务而设计。

差别:1.TPU 与图形处理器（GPU）相比，TPU 采用低精度（8 位）计算，以降低每步操作使用的晶体管数量。降低精度对于深度学习的准确度影响很小，但却可以大幅降低功耗、加快运算速度。

2.TPU 使用了脉动阵列的设计，用来优化矩阵乘法与卷积运算，减少 I/O 操作。此外，TPU 还采用了更大的片上内存，以此减少对 DRAM 的访问，从而更大程度地提升性能。

13、(例)假设浮点(FP)平方根(FPSQR)占用一项关键图形基准测试中 20%的执行时间。升级 FPSQR 硬件，使这一运算速度提高到原来的 10 倍。另一项提议是让图形处理中所有 FP 指令的运行速度提高到原来的 1.6 倍，FP 指令执行速度提高到 1.6 倍所需要的工作量与加快平方根运算的工作量相同，比较这两种设计方案。

解：通过结算加速比来对比两种方案

$$\text{加速比 } FPSQR = \frac{1}{(1-0.2)+\frac{0.2}{10}} = \frac{1}{0.82} = 1.22$$

$$\text{加速比 } FP = \frac{1}{(1-0.5)+\frac{0.5}{1.6}} = \frac{1}{0.81125} = 1.23$$

答：提高整体 FP 运算的性能要稍好一点，原因是它的使用频率较高。

14、(例)通过冗余来提高电源可靠性，将 MTTF 从 200 000 小时提高到 830 000 000 小时，达到 4150 多倍。

磁盘子系统故障的计算为：

$$\text{故障率}_{\text{系统}} = 10 \times \frac{1}{1000000} + \frac{1}{500000} + \frac{1}{200000} + \frac{1}{200000} + \frac{1}{1000000}$$

$$= \frac{10+2+5+5+1}{1000000\text{小时}} = \frac{23}{1000000\text{小时}}$$

因此，可改进的故障率比例是 5 次/百万小时占整个系统 23 次/百万小时的比例，即 0.22。

解：可靠性的改进为：

$$\text{改进}_{\text{电源对}} = \frac{1}{(1-0.22)+\frac{0.22}{4150}} = \frac{1}{0.78} = 1.28$$

尽管一个模块的可靠性提高了 4150 倍之巨，但从系统的角度来看，这一改变所带来的的好处虽然可测，但数值很小。

15、如何实现 Cache 一致性？

a）通过在总线加 LOCK#锁的方式：使得只能有一个 CPU 能使用这个变量的内存。在总线上发出了 LCOK#锁的信号，那么只有等待这段代码完全执行完毕之后，其他 CPU 才能从其内存读取变量，然后进行相应的操作；

b）通过缓存一致性协议：Intel 的 MESI 协议，保证了每个缓存中使用的共享变量的副本是一致的。当 CPU 写数据时，如果发现操作的变量是共享变量，即在其他 CPU 中也存在该变量的副本，会发出信号通知其他 CPU 将该变量的缓存行置为无效状态，因此当其他 CPU 需要读取这个变量时，发现自己缓存中缓存该变量的缓存行是无效的，那么它就会从内存重新读取。

cache 的操作有 2 种：写回和无效。写回操作是将 cache 中数据写回到 DDR 中，无效操作是无效掉 cache 中原有数据，下次读取 cache 中数据时，需要从 DDR 中重新读取。这两种操作其实都是为了保证 cache 数据一致性。

16、简述虚拟地址翻译机制及如何提高其性能。

答：地址翻译指的是 DRAM 缓存命中时，由虚拟地址找到物理地址的过程。该过程是完全由硬件来完成的。

1)CPU 有一个专门的页表基地址寄存器(PTBR)指向当前页表的基地址，快速定位到该进程的页表。

2)根据虚拟页号，找到虚拟地址在页表的值。

3)根据值中的物理页号，找到物理地址。

4)得到实际物理地址后，根据高速缓存的原理，把一个物理地址映射到高速缓存具体的组，行，块中，找到实际存储的数据。

（1)CPU 拿到一个虚拟地址，分为两步，先通过页表机制确定该地址所在虚拟页的内容是否从磁盘加载到物理内存页中，然后通过高速缓存机制从该物理地址中取到数据

2)地址翻译硬件要把这个虚拟地址翻译成一个物理地址，从而可以再根据高速缓存的映射关系，把这个物理地址对应的值找到

3)地址翻译硬件利用页表数据结构，TLB 硬件缓存等技术，目的只是把一个虚拟地址映射到一个物理地址。要记住 DRAM 缓存是全相联的，所以一个虚拟地址和一个物理地址是动态关联的，不能直接根据虚拟地址推导出物理地址，必须根据 DRAM 从磁盘把数据缓存到 DRAM 时存到页表时存的实际物理页才能得到实际的物理地址,用物理页 PPN + VPO 就能算出实际的物理地址(VPO = PPO，所以直接用 VPO 即可)。 PPN 的值是存在页表条目 PTE 中的。地址翻译做了一堆工作，就是为了找到物理页 PPN，然后根据 VPO 页面偏移量，就能定位到实际的物理地址。

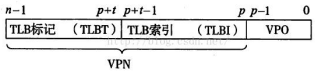
4)得到实际物理地址后，根据高速缓存的原理，把一个物理地址映射到高速缓存具体的组，行，块中，找到实际存储的数据。）

提高其性能

利用高速缓存及 TLB 加速地址翻译

1)在地址翻译的过程中，有一个步骤是通过页表条目地址来查找 PTE，我们可以通过结合高速缓存来提高其查找速度；

2) TLB 是专门用于虚拟寻址的缓存，称为翻译后备缓冲器，由标记（TLBT）和索引（TLBI）组成。



17、相变内存（PCM）

相变内存，通常称为 PCM 技术或相变 RAM 技术。之所以被称为相变内存，是因此它利用特殊材料（包括硫、硒或者碲等，目前主要的研究方向是硫属化物玻璃）在晶态和非晶态之间相互转化时所表现出来的导电性差异来存储数据。它是一种非易失性的内存产品。相变内存是下一代内存（闪存）技术。相变内存结合了 DRAM 内存的高速存取，以及闪存在关闭电源之后保留数据的特性，被业界视为未来闪存和内存的替代品。人们需要一种能够保存数据的 DRAM，而相变内存将满足这一需求。相变内存的另外一个优点是，可以在不删除现有数据的情况下写入数据，这比如今的内存更为快捷。根据统计，相变内存的功耗只有现有闪存的一半，但是读写速度可以达到闪存的 1000 倍。

18、RISC-V 开源处理器

RISC-V 是一个基于精简指令集（RISC）原则的开源指令集架构（ISA）。与大多数指令集相比，RISC-V 指令集可以自由地用于任何目的，允许任何人设计、制造和销售 RISC-V 芯片和软件。虽然这不是第一个开源指令集，但它具有重要意义，因为其设计使其适用于现代计算设备（如仓库规模云计算机、高端移动电话和微小嵌入式系统）。设计者考虑到了这些用途中的性能与功率效率。该指令集还具有众多支持的软件，这解决了新指令集通常的弱点。RISC-V 指令集的设计考虑了小型、快速、低功耗的现实情况来实做，但并没有对特定的微架构做过度的设计。

19、Cache 失效主要原因及解决方法。

Cache 失效主要原因：强制性失效（Compulsory)：第一次访问某一块，只能从下一级 Load，也称为冷启动或首次访问失效

容量失效（Capacity）：如果程序执行时，所需块由于容量不足，不能全部调入 Cache， 则当某些块被替换后，若又重新被访问，就会发生失效。可能会发生“抖动”现象

冲突失效（Conflict (collision)）：组相联和直接相联的副作用，若太多的块映像到同一组（块）中，则会出现该组中某个块被别的块替换（即使别的组或块有空闲位置），然后又被重新访问的情况，这就属于冲突失效

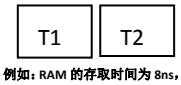
解决方法：①降低失效率：1、增加 Cache 块的大小 2、增大 Cache 容量 3、提高相联度②增大 Cache 容量：对冲突和容量失效的减少有利③增大块：减缓强制性失效、可能会增加冲突失效（因为在容量不变的情况下，块的数目减少了）④通过预取可帮助减少强制性失效：必须小心不要把你需要的东西换出去、需要预测比较准确（对数据较困难，对指令相对容易）

提高相联度，会增加命中时间⑤减少失效开销：4、多级 Cache 5、使读失效优先于写失效；多级 cache 的优点：减少失效开销、缩短平均访存时间（AMAT）⑥由于读操作为大概率事件，需要读失效优先，以提高性能

缩短命中时间： 6、避免在索引缓存期间进行地址转换

20、cache 命中

设 x 为命中率。T=x*T1+(1-x)(T1+T2)=T1+(1-x)T2



例如：RAM 的存取时间为 8ns，CACHE 的存取时间为 1ns，CACHE 的命中率为 90%。则存储器整体访问时间由没有 CACHE 的 8ns 减少为：1ns×90% + 8ns×10% = 1.7ns 速度提高了近 4 倍。

21、评价 cache 性能公式：

平均存储访问时间=命中时间(缓冲命中需要的时间)+缺失率×缺失代价

Ex1.如果缓存的命中时间为 2 个 cycle，缺失率为 0.05，缺失代价为 20 个 cycle，那么平均存储访问时间是多少？

2+0.05×20=3

Ex2.方案一：分立 cache 设计，指令和数据 cache 独立，分别为 16KB

方案二：指令和数据 cache 合并，总共为 32KB，Load 和 Store 操作命中时额外的需要一个时钟周期，因为只有一个 cache 端口满足请求。

假定 cache 命中需要 1 个周期，确实代价为 100 个周期；并且假设 36%的数据存储器访问为数据访问，下表为每一千条指令缺失次数。

容量	指令 cache	数据 cache	一体 cache
16KB	3.82	40.9	51.0
32KB	1.36	38.4	43.3

缺失率=每条指令的缺失次数/每条指令的内存访问次数

=1000 条指令的缺失次数/1000 条指令的内存访问次数

缺失率(16KB 指令 cache)=(3.82/1000)/(1000/1000)=0.004

缺失率(16KB 数据 cache)=(40.9/1000)/0.36=0.114

缺失率(132KB 一体 cache)=(43.3/1000)/(1+0.36)=0.0318

→分立 cache 总的缺失率：74%×0.004+26%×0.114=0.0324

平均存储器访问时间=指令占比×(命中时间+指令缺失率×缺失代价)+数据占比×(命中时间+数据缺失率×缺失代价)

平均存储器访问时间(分立 cache)=74%×(1+0.004×100)+26%×(1+0.114×100)=4.24

平均存储器访问时间(一体 cache)=74% × (1+0.0318 × 100)+26%×(1+0.0318×100)=4.44

CPU 时间=(CPU 执行时间时钟周期数+存储器停顿时间周期数)×时钟周期数

Ex.假设某顺序执行的处理器，其平均缺失率为 2%，平均每条指令要访问存储器 1.5 次，cache 缺失代价为 100 个周期，将 cache 命中时间包含在 CPU 执行时间内，CPU 理想的 CPI 为 1.0，比较 cache 存在与否，对性能的影响。

CPU 时间=指令数×(指令执行时钟周期数+存储器停顿时钟周期/指令数) ×时钟周期时间

其中，存储器停顿时钟周期=代价×次数=1.5×2%×100

CPU 时间=指令数×(1.0+30×100/1000) ×时钟周期时间=指令数×4×时钟周期时间

可知，在 cache 情况下，CPI 为 4.

如果没有 cache，又考虑存储访问时间的话：CPI 增加到 1.0+100×1.5=151，即为带 cache 的 40 倍。

cache 对于低 CPI 和高时钟频率的 CPU 性能影响尤其重要。

22、硬件描述语言写状态集 verilog

1）ALU 模块

```
module alu(op,a,b,c);
        input op;
        input [7:0] a,b;
        output [7:0] c;
        assign c = (op==0)?(a+b):(a&b);
endmodule
```

2）IR 模块

```
module ir(clk,load,din,dout);
        input clk,load;
        input [1:0] din;
        output [1:0] dout;
        reg [1:0] dout;
        always @(posedge clk)
        begin
                if(load)
                        dout = din;
        end
Endmodule
```

（3）PC 模块

```
module pc(clk,rst,load,inc,din,dout);
        input clk,rst,load,inc;
        input [5:0] din;
        output [5:0] dout;
        reg [5:0] dout;
        always @(posedge clk)
        begin
                if(rst)
                        dout = 0;
                else if(inc)
                        dout = dout + 1;
                else if(load)
                        dout = din;
        end
endmodule
```

（5）Control 模块

```
module control(clk,rst,ir,
        read,membus,
        arload,
        pcload,pcinc,pcbus,
        drload,drbus,
        alusel,
        acload,acinc,
        irload
);
        parameter FETCH1 = 0;parameter FETCH2 = 1;
        parameter FETCH3 = 2;parameter ADD1 = 3;
        parameter ADD2 = 4;parameter AND1 = 5;
        parameter AND2 = 6;parameter INC1 = 7;
        parameter JMP1 = 8;
        input clk,rst;input [1:0] ir;
        output  read,membus,
                        arload,
                        pcload,pcinc,pcbus,
                        drload,drbus,
                        alusel,
                        acload,acinc,
                        irload;
        reg [3:0] state,nextstate;
        always @(posedge clk)
        begin
                if(rst)
                        state <= FETCH1;
                else
                        state <= nextstate;
        end
        always @(state or ir)
        begin
                case(state)
                        FETCH1:nextstate <= FETCH2;
                        ETCH2:nextstate <= FETCH3;
                        FETCH3:
                        begin
                                if(ir==0)
                                nextstate <= ADD1;
                                else if(ir == 1)
                                nextstate <= AND1;
                                else if(ir == 2)
                                nextstate <= INC1;
                                else
                                nextstate <= JMP1;
                                end
                        ADD1:nextstate <= ADD2;
                        ADD2:nextstate <= FETCH1;
                        AND1:nextstate <= AND2;
                        AND2:nextstate <= FETCH1;
                        JMP1:nextstate <= FETCH1;
                        INC1:nextstate <= FETCH1;
                        default:nextstate <= FETCH1;
                        endcase
        end
    assign arload = (state == FETCH1 || state== FETCH3)?1:0;
    assign pcload = (state == JMP1)?1:0;
    assign pcinc = (state == FETCH2)?1:0;
    assign drload = (state == FETCH2 || state== ADD1 || state== AND1)?1:0;
    assign acload = (state == ADD2 || state== AND2)?1:0;
    assign acinc = (state == INC1 )?1:0;
    assign irload = (state == FETCH3)?1:0;
    assign alusel = (state == AND2)?1:0;
    assign membus = (state == FETCH2 || state== ADD1 || state== AND1)?1:0;
    assign pcbus = (state == FETCH1)?1:0;
    assign drbus = (state == FETCH3 || state== ADD2 || state== AND2 || state == JMP1)?1:0;
    assign read = (state == FETCH2 || state == ADD1 || state == AND1)?1:0;
endmodule
```

6）CPU 模块的 Verilog 实现

```
module cpu(clk,rst,data,read,addr,acout);
        input clk,rst;input [7:0] data;
        output read;output [5:0] addr;
        output [7:0] acout;wire [7:0] bus;
        wire [5:0] pcout;wire [7:0] drout,acout,aluout;
        wire [1:0] irout;wire alusel,acload,acinc;
        tri8 t0(membus,data,bus[7:0]);
        ar ar1(clk,arload,bus[5:0],addr);
        pc pc1(clk,rst,pcload,pcinc,bus[5:0],pcout);
        tri6 t1(pcbus,pcout,bus[5:0]);
        dr dr1(clk,drload,bus[7:0],drout);
        tri8 t2(drbus,drout,bus[7:0]);
        alu alu1(alusel,acout,bus[7:0],aluout);
        ac ac1(clk,rst,acload,acinc,aluout,acout);
        ir ir1(clk,irload,bus[7:6],irout);
        control c1(clk,rst,bus[7:6],
                        read,membus,
                        arload,
                        pcload,pcinc,pcbus,
                        drload,drbus,
                        alusel,acload,acinc,
                        irload);
endmodule
```

8）4 位计数器

```
module count4(out,reset,clk);
output[3:0] out;
 input reset,clk;
reg[3:0] out;
always @(posedge clk)
begin
if (reset) out<=0;              //同步复位 else
 out<=out+1;   //计数
        End
 endmodule
```

10）用 case 语句描述的 4 选 1 数据选择器

```
module mux4_1(out,in0,in1,in2,in3,sel);
output out;
input in0,in1,in2,in3;
input[1:0] sel;
reg out;
always @(in0 or in1 or in2 or in3 or sel)      //敏感信号列表
 case(sel)
2'b00:    out=in0;
2'b01:    out=in1;
2'b10:    out=in2;
2'b11:    out=in3;
default: out=2'bx;
   Endcase
 Endmodule
```