计算机系统结构

1. **高速缓存 cache**：基于 SRAM 的高速存储器。
   a) 基本概念
      i. **缓存块 block**：缓存中的一块的大小；
      ii. **缓存命中 hit**：要查询的数据在缓存中；
      iii. **缓存失效 miss**：要查询的数据不在缓存中。
   b) 缓存失效的分类：4 C。
      i. Compulsory miss：当一个数据块第一次被访问时产生的失效；
      ii. Capacity miss：当一个数据块因为缓存容量满而被删除后导致的失效；
      iii. Conflict miss：当一个数据块因为所在的缓存组容量满而被删除后导致的失效；
      iv. Coherence miss：当一个数据块由于缓存数据同步的问题导致的失效。
   c) 高速缓存的基本参数：设 N_c 为缓存命中的次数，N_m 表示缓存失效、访问内存的次数，那么缓存命中率

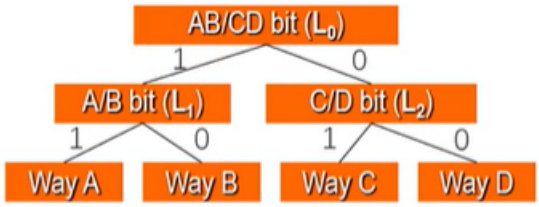
$$h = \frac{N_c}{N_c + N_m}$$

   设 t_c 为访问缓存数据的时间，t_m 为访问内存数据的时间，那么平均访问时间

$$t_a = t_c + (1 - h)t_m$$

   d) **优化方法**：
      i. 当数据块比较大时，降低 t_m 的两种特殊方法
         1. **Early Restart**：可能需要传输若干次才能完整传输某块数据，但是可以让 CPU 收到一部分数据时就先开始继续运行，接着再不断将后续数据传给 CPU；
         2. **Critical Word First**：先传输 CPU 继续运行所缺失的数据给 CPU，让 CPU 得以继续运行，然后再慢慢填充 cache block 中剩余的数据。
      ii. **读权限高于写权限**：向内存写数据时，先将数据写入 write buffer 中；向内存读数据的请求可以直接查询 write buffer 中是否有相应的数据，如果有则可以直接读取；这样就不需要等待内存写操作全部完成再执行其他指令，可以减少 t_m。
      iii. **非阻塞式 cache**：在处理 cache miss 时（向内存读/写数据），能够继续处理其他读写数据的请求。
      iv. **多级 cache**：L1 这类低级别的 cache 强调 t_c，L2、L3 这类高级别的 cache 强调 h。
      v. **Victim cache**：一个小的缓冲区，存储一些最近被从 cache 中删除的数据；查询 cache 时同时查询这里。
      vi. **预取 prefetching**：包括硬件预取 hardware prefetching 和软件预取 software prefetching。硬件预取为硬件自动预取当前访问的块的接下来几块，不占用处理器时间，也不需要额外的指令；软件预取通常为编译器加入预取优化指令预取接下来要访问的内容，需要特殊的 ISA 提供支持。
   e) **地址映射**：提供一个内存地址到缓存行号的映射。所有地址映射的第一步都是把内存地址后若干位去掉（由于缓存中以块为单位，最后若干位其实为块中偏移量 offset）。
      i. **直接映射 direct-mapped**：直接取处理后的内存地址后 x 位作为缓存的行号（则缓存大小为 2^x 行）；产生失效时，若与当前的缓存行中存在数据，则直接替换即可。优点：简单、地址映射速度快、易于实现、替换算法简单；缺点：缓存低利用率、低命中率、乒乓效应。
      ii. **全相联 fully-associated**：直接取处理后的地址在 cache 中的每一行进行查找，逐个比对是否查询成功；若产生失效，当前缓存行不满则直接加入，否则利用替换算法选择一个缓存行进行替换即可。优点：缓存利用率高；缺点：替换算法复杂、搜索过程速度慢。
      iii. **组相联 set-associated**（常用）：直接取处理后的地址后 x 位作为缓存的组号（则缓存大小大于 2^x 行）；每次在缓存组中间的每一行逐一进行查找并比对是否查询成功；若产生失效，当前组内缓存行不满则直接加入，否则利用替换算法选择组内的一个缓存行进行替换即可。组合了全相联和直接映射的优点。在组相联中，每组中的缓存行称为路 way，每组中缓存行数量称为路数。
   f) **替换策略**：当缓存失效并且（组）空间已满时，替换策略决定要将当前数据块替换哪一个缓存行的数据块。
      i. **随机替换 random replacement**：随机选择一个同组内的数据块进行替换；
      ii. **FIFO**：替换最早进入该组的数据块；
      iii. **LFU**：替换近期使用频率最低的数据块；
      iv. **LRU**：替换近期最久没有被使用过的数据块。

v. **伪 LRU**：由于 LRU 一共有 $n!$ 种可能的情况，至少需要 $\log n! = O(n \log n)$ 额外的位进行存储，因此空间消耗过大。提出伪 LRU 算法，仅需要 $O(n)$ 额外的位进行存储。



对于组内的每一行，建立一个叶子节点，并将树组织成完全二叉树的形式，每一个内部节点用一位来表示，那么一共有 $O(n)$ 位。每位存储的值表示的是上一次经过这个子树时走的方式（1-向左走进入左子树，0-向右走进入右子树）。

更新信息 当 cache hit 时，根据最新一次走的路径，更新路径上所有内部节点的信息。

替换策略 根据节点信息，每次均选择最久没有被访问的路径往下走，直到叶子节点即为被替换的行。例：当在 way B 命中时，将 L_0 更新为 1，将 L_1 更新为 0；当 $L_2L_1L_0 = 001$ 且需要替换时，从 L_0 开始，往右子树走到 L_2，再往左子树走到 way C，即为所替换的缓存行。

g) **写策略**：执行写操作时，保持缓存与内存中数据一致性的策略。

i. **写直达 write through**：（缓存命中时）每次向缓存中写数据，同时在内存中对应位置写入数据；

ii. **不按写分配法**：（缓存不命中时）直接向内存中位置写数据，不用将数据移入缓存（与写直达法连用）；

iii. **写回法 write back**：（缓存命中时）每次仅向缓存中写数据，直到该块被从替换出去才将数据写入内存；

iv. **按写分配法**：（缓存不命中时）在缓存中寻找合适的缓存行（可能需要替换）存储该数据，并仅在缓存中进行修改（与写回法连用）；

v. **如何选择？** 按照对同步性要求高/低分别选择两组策略；同步性要求高的使用写直达与不按写分配法；同步性要求低的使用写回和按写分配法。

h) **缓存行**：缓存行包括哪些内容？

i. **Valid-Invalid bit**（1 位）：该缓存行是否有效；

ii. **Dirty bit**（1 位）：仅限写回法，数据是否被修改过；

iii. **LRU bits**（若干位）：仅限 LRU 替换策略，数量根据题目而定；

iv. **Tag bits**（若干位）：用于识别该缓存行在内存中位置的数据；内存地址的总位数扣除掉块内偏移量的位数与可能有的组号的位数；

v. **Data bits**（若干位）：取决于缓存块的大小，根据题目而定。

i) 多级高速缓存

i. **包含性 inclusive**：高级别的高速缓存（L2、L3）包含低级别的高速缓存（L1）的全部信息；可能会浪费部分缓存空间，但是可以加速多处理器时其他处理器的缓存查找速度（查找 L2、L3 即可）。

ii. **非包含性 exclusive**：低级别的高速缓存（L1）的信息可以不包含在高级别的高速缓存（L2、L3）中，并且在相邻级别的缓存中进行替换（如 L1 与 L2 替换）；在**严格非包含性 strictly exclusive** 时，每个数据块只存在在一个缓存中；能够最大化缓存的有效空间，但是一般需要缓存间有一个统一的块大小。

j) **缓存一致性 cache coherence**：多处理器的系统中，每个处理器拥有自己的缓存，如何处理缓存间的数据一致性问题？

i. **基于总线的监听协议 bus-based snooping protocol**：利用总线来进行数据传输，最基本的协议。

1. **写失效 write invalidate**（常用）：当一个数据在某高速缓存中被修改了，利用总线通知其他高速缓存，将包含该数据的缓存行设为无效（invalid）；

2. **写更新 write update**：当一个数据在某高速缓存中被修改了，利用总线将修改后的数据发送到其他高速缓存，并更新（可能存在的）对应数据。

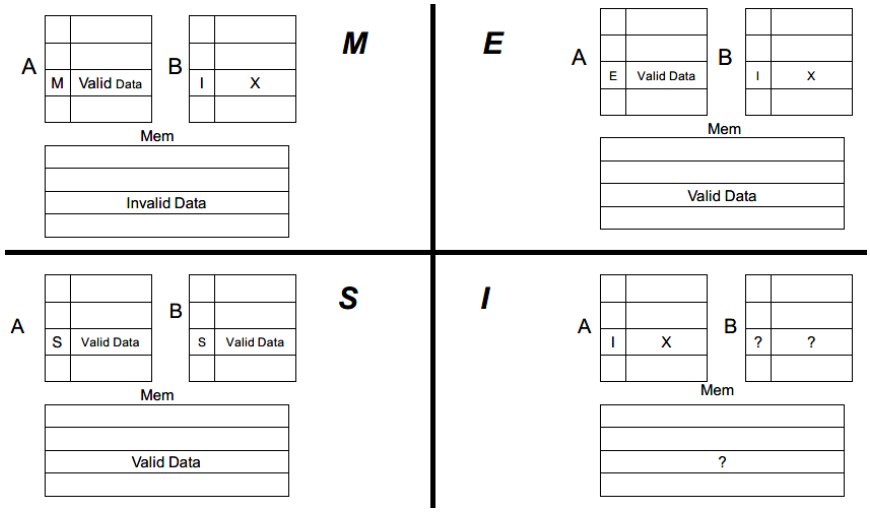
ii. **MSI 协议 modified-shared-invalid protocol**：利用 M/S/I 三种状态来表示缓存行的状态。其中 M 表示该缓存行的数据被修改过，且为最新数据（即 valid=1, dirty=1）；I 表示该缓存行的数据是无效的，即 valid=0；S 表示该缓存行的数据与其他缓存共享（即 valid=1，dirty=0）。

更新规则：

- 对 M 状态，无论是 read/write 都不影响其状态位，仍然保持 M 状态；

- 对 S 状态，如果是 read 则不改变状态，仍然保持 S 状态；
- 对 S 状态，如果是 write，则修改数据后，在总线上发出 invalidate 信号，将其他缓存中的包含相同数据的缓存行的状态设置为 Invalid，这种情况称为 upgrade miss；
- 对 I 状态，则进行读操作时，寻找其他缓存中的合法数据（M/S 状态），利用总线传输更新数据后将所有包含相同数据的缓存行的状态均设为 S；
- 对 I 状态，进行写操作时，寻找其他缓存中的合法数据（M/S 状态），利用总线传输更新数据后将所有包含相同数据的缓存行的状态均设为 I；
- 在以上状态中，某些时候可以顺带更新内存（或下一级高速缓存）的数据，对于写操作可能需要先将下级高速缓存的对应行 blocked，避免过程中读到错误数据。

iii. **MESI 协议 modified-exclusive-shared-invalid protocol**：增加 exclusive 状态表示共享内存（或下一级高速缓存）中的数据是最新的，且只有一个缓存拥有这个数据。更新规则与 MSI 协议类似。



iv. **MESIF 协议 modified-exclusive-shared-invalid-forward protocol**：增加 forward 状态，当其他高速缓存从 modified/forward 状态的高速缓存行中读取数据时，将数据所在的缓存行的状态置为 shared，同时将读取后存放的高速缓存行的状态设为 shared。这样避免了一直向同一个高速缓存中读取数据。

v. **共享失效 sharing miss**：共享失效包含读失效 read miss、写失效 write miss、更新失效 upgrade miss；一个简单的判断失效方法是：本次操作是否有在总线上传输任何数据（包括 invalidation）。

1. **假共享失效 false sharing miss**：由于缓存行的其他数据的影响，导致对当前数据的访问出现了失效的情况。一般判断标准为：（缺一不可）①共享失效；②当前行所需要操作的数据正确；③该数据因为当前行的其他数据的操作，造成了这个共享失效。
2. **真共享失效 true sharing miss**：除了假共享失效的其他共享失效，具体表现为如果缓存行只存储单个数据，仍然存在的共享失效。

2. **虚拟内存**：由于物理内存存在大小限制，我们需要用虚拟内存来进行物理内存的“扩展”。

a) **页 page**：将物理内存地址与虚拟内存地址同时划分的一个数据块，一般为 4 KB，也可以比较大。

i. 物理内存中按页存储，虚拟内存中也按页存储；

ii. 程序在物理内存中加载需要的页然后执行，并在执行过程中不断进行页的替换；

iii. 当程序需要的页不在内存中时，产生缺页中断请求，系统处理请求时会将硬盘上的后备区中的该页与内存中的某个页进行调换，让该页加载进内存，程序得以继续运行。

b) **页表 page table**：将虚拟内存中的地址与物理内存中的页进行映射的表，存储在内存中，每个进程有一个。

i. 线性页表：将所有虚拟内存的地址按序排列，并将映射组织成线性表的形式，规模庞大无法存储；

ii. **多级页表**：由于许多程序的大部分虚拟地址都是空的，因此采取多级的页表架构，一级页表中可能含有许多空项（整个二级页表为空的情况），因此可以节约大部分不必要的空间。如对 32 位地址，4KB 大小的页面，剩下的地址位数位 20，即可以用两级页表存储，每级负责 10 位的映射。

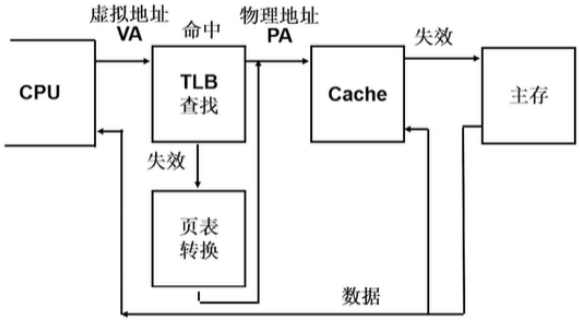
iii. **反向页表 inverted page table**：普通页表浪费空间，因为其时一个虚拟地址（多）到实际地址（少）的映射，我们可以将映射反向（实际地址到虚拟地址的映射）来减少空间消耗，这种页表称为反向页表。

iv. **哈希页表 hash page table**：也是解决普通页表浪费空间的方法，将虚拟地址进行哈希处理减少空间需求，这种页表称为哈希页表。

c) **地址转换策略**：页偏移量不变，页号通过页表（与 TLB）转换成页框号。如果页表中不包含相应的映射，则产生**页面失效 page fault**，进行页面置换。

d) **页面失效 page fault**：页面失效后，从后备区调入需要的页面进行置换，可以使用与 cache 相似的算法进行替换，如 LRU 算法等等。

e) **快表 translation lookaside buffers，TLB**：页表存储在内存中，因此访问页表的时间即为访问主存的时间，为了加快地址转换的速度，我们使用 TLB 作为页表的专用 cache。则数据访问流程见下（图中忽略了 page fault 的情况，该情况需要从后备区调页并更新页表与 TLB；且若 TLB 失效）。

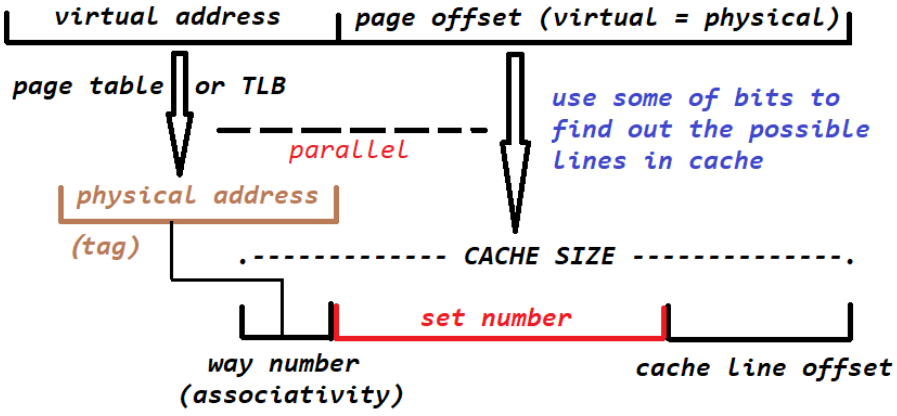


f) **高速缓存与地址转换策略的优化**

i. **Physical-addressed caches PI-PT**：利用物理地址作为索引与缓存行中的 tag，这种实现方式读取数据时，地址转换必须与访问数据串行执行，效率较低；

ii. **Virtual-addressed caches VI-VT**：利用虚拟地址作为索引与缓存行中的 tag，这种实现方式优点在于读取数据时无需进行地址转换，以降低缓存命中的时间，缺点在于高速缓存中可能会出现重复（一个物理地址可能在不同的进程中对应着不同的虚拟地址，导致高速缓存中存在相同的行）。

iii. **Virtual-indexed Physical-tagged VI-PT**：利用虚拟地址作为索引查询缓存行的数据，缓存行中的 tag 利用物理地址来标记。实际上也是用物理地址来进行 cache 查询（因为虚拟地址的页偏移量与物理地址相同），用页偏移量查询出组号后，将若干备选项挑选出来；然后等待地址转换结果并进行比对。这种方法的优点在于可以并行读取数据与地址转换，以降低缓存命中的时间。



实现时，需要满足如下条件：

$$\text{set number} \times \text{cache line size} \le \text{page size}$$

由于

$$\text{set number} \times \text{cache line size} \times \text{associativity} = \text{cache size}$$

因此该条件又可以被写作：

$$\text{page size} \times \text{associativity} \ge \text{cache size}$$

3. **存储器一致性模型 memory consistency**：读操作和写操作在乱序执行中的执行顺序。

a) **顺序一致性模型 sequential consistency，SC**：这个模型是程序员希望机器提供的（MIPS），它能保证：

i. 处理器看到自己读内存和写内存的顺序和程序中的一致；

ii. 处理器看到的别的处理器的读内存和写内存的顺序和程序中的一致；

iii. 所有处理器看到的读内存和写内存的顺序是一致的。

该模型要求对于每个处理器均为按顺序处理，但是允许不同处理器之间的完整顺序中存在处理器之间的访问顺序交叉 interleaving。

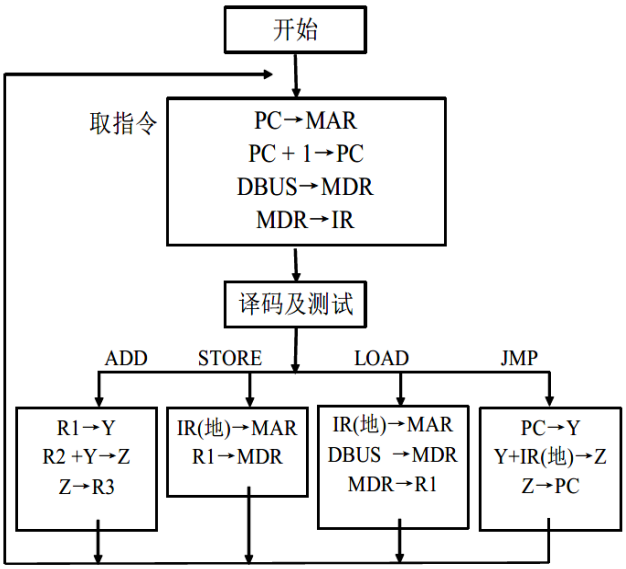
b) **全存储排序模型 total store order，TSO**：这个模型只要求 store 指令按序执行即可（x86）。store 指令可以暂存于写缓冲区 store buffer 中，接下来的 load 指令可以通过旁路 bypassing 从写缓冲区的本线程写入的数据中读取数据（不能从其他线程写入 write buffer 中的内容读取）。同时，写缓冲区还能隐藏写失效 store miss 的延迟。如果需要强行规定存取顺序，可以通过添加内存屏障 fence 来实现来阻塞其他操作。

c) **释放一致性模型 release consistency，RC**：这个模型只要求指向同一地址的操作按顺序排序即可，并且支持 store buffer 内的许多指向同一地址的操作的合并。同样可以通过添加内存屏障 fence 来强制定义顺序。

4. **处理器设计**：设计一个处理器。

a) **信号设计**：Jump 信号（是否跳转，可能需要据情况分成 Jump、Jr 和 Jal 三种）、MemRead（读内存）、MemToReg（是否从内存写入寄存器）、MemWrite（写内存）、Branch（条件转移指令，可能需要据情况分为 Beq 和 Bne 两种），ALUSrc（ALU 输入是寄存器读取出的值还是立即数）、RegDst（目标寄存器地址是 rd 还是 rt）、RegWrite（写寄存器）等等，根据需要实现的指令类型设计。

b) **单总线结构处理器**：仅仅一个总线，每个指令实现一个模块，最后汇总即可。



其中，MAR 表示的是 Memory Address Register，即访存地址的寄存器；MDR 为 Memory Data Register，即读取存储器得到的数据；IR 为 Instruction Register，即存储指令的寄存器。

c) 三种周期

i. **时钟周期**：节拍脉冲 T；

ii. **机器周期**：CPU 周期，从内存中读出一条指令的最短时间（取指令）；

iii. **指令周期**：从内存中取一条指令并执行该指令所用的最短时间，由若干 CPU 周期构成，一个 CPU 周期又包含若干时钟周期。

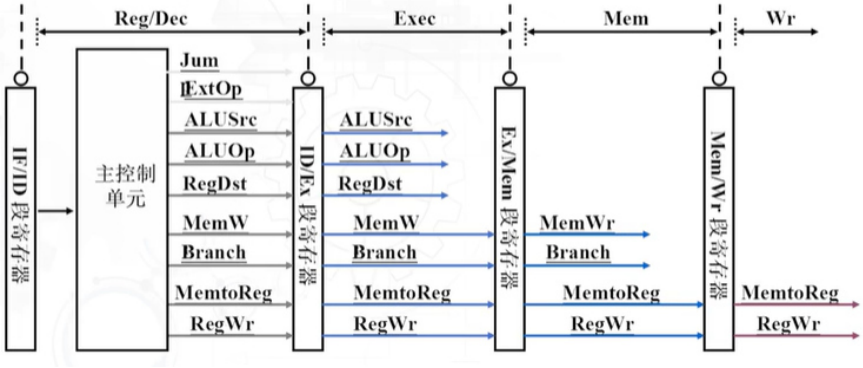
d) **硬连线控制器 hard-wired controller**：直接硬件布线的控制器；其特点为：组成的网络复杂且无规则，设计和调试困难，不可改变指令系统，但是速度快。

e) **微程序控制器 microprogramming controller**：一条指令的处理包含许多微操作序列，将这些操作所需要的控制信号以多条微指令表示，执行一条微指令就给出一组微操作控制信号，而执行一条指令也就是执行一段由多条微指令组成的微程序。指令系统功能的设计以微指令为单位进行存储和布线，微指令存储在 ROM 中。其主要特点是可以用一个简单的处理器实现复杂的 ISA，这也是 CISC 机器逐渐发展的原因；同时，微指令更易于对于指令的调试和扩展，可以在不修改数据通路的情况下增加指令支持。后期，由于编译器技术的进步以及微结构技术（如 pipelining，cache 等等）的进步，微指令逐渐被取代，但是现在仍然有一些复杂指令仍然使用微程序进行控制。

f) **五阶段流水线处理器 five-stage pipelined processor**

i. 五个阶段：取指令 IF、译码与寄存器访问 ID、执行指令 EX、访存 MEM、写回 WB。每条指令都必须经过这五个阶段（虽然有一些可能是空阶段），否则会引起资源冲突 resource conflict。

ii. **段寄存器 segment registers**：储存每两个阶段之间所需要传递的信号与其他数据。段寄存器的位数需要通过分析段寄存器的构成来计算得到，包括临时数据、信号等等。一种可行的段寄存器传递信号的方式如下图。



iii. **冒险 hazard**

1. **结构冒险 structural hazard**：一个功能部件在同一个时间段被两条指令同时使用引发的问题。比如指令内存和数据内存未分离时，IF 与 MEM 阶段都需要访存产生冲突；以及 ID 与 WB 阶段都需要访问寄存器产生冲突。解决方案分别是：将**指令内存与数据内存分离**或**加入空指令**、将周期划分成两个半周期并分别在两个半周期执行寄存器读与寄存器写操作。
2. **数据冒险 data hazard**：前面指令的结果影响后面指令的执行，导致后面指令执行错误引发的问题。比如寄存器的写后读 RAW 数据冒险与读寄存器-使用数据冒险。解决方案有：流水线停顿、（编译器）增加空指令以及**前向通路 forwarding**（将结果直接送入下面指令的当前阶段）。注意：前向通路无法彻底解决读寄存器-使用数据冒险，仍需要流水线停顿或增加空指令。
3. **控制冒险 control hazard**：条件转移、无条件转移指令执行的结果影响后面指令的执行，可能执行了错误指令。解决方案主要是尽早转移（甚至可以对条件转移指令进行**分支提前决策**，在 ID 阶段进行决策），同时**清空**执行的错误指令的进度；另一可行的方法是（编译器）添加**转移延迟槽**。

iv. **综合分析**：理想情况，五阶段流水线处理器 $CPI = 1$。其优点有简单、易于实现；缺点是有 $CPI = 1$ 的限制，并且比较难优化复杂的指令，比如乘法；同时其具有**紧耦合 tightly coupled** 的特点，即一条指令的暂停会导致后续指令的暂停执行，影响效率。

v. **可行的优化方法**：

1. 超流水 superpipelining，即增加时钟频率，减少一个周期的时间，同时增加流水线的长度；但是这种方法会带来成本与复杂性的增加，同时需要更多的段寄存器，可能会带来更多的冒险和停顿，以及功耗比较高。
2. 多发射 multiple issue：每次取多条指令进行发射，利用指令层面的并行性 instruction level parallelism，即 ILP，来减小 CPI 提高效率，使得 $CPI < 1$。将会在后续部分详细阐述。

g) **中断 interrupt 与异常 exception 处理**：

i. **精确中断**：需要满足两个条件：（1）所有在产生中断的指令之前进行的指令均已提交结果；（2）所有在产生中断的指令之后的指令没有对系统产生影响。

ii. **非精确中断**：中断的顺序并不依赖于指令的顺序，可能顺序会有一定的变化。

iii. 精确中断与非精确中断的选择：一般我们需要知道中断的顺序，所以我们偏向于实现精确中断，这样可以直接得到中断的顺序，避免重复处理不必要的中断。

iv. **流水线的精确中断处理**：在开始对结果进行写回（五阶段流水线的 WB）之前，也就是提交点 commit point 之前，加入一个检查流程，检查该指令在执行时是否产生中断。若是，则调用中断处理程序进行处理。需要注意的地方有：

1. 较早阶段的中断和异常会覆盖较晚阶段的中断和异常；
2. 处理中断时，将会关闭外部中断的响应，即此时无法接收外部中断；

h) **转移预测 branch prediction**：

i. 分类：静态转移预测（由编译器完成）、动态转移预测（在执行时完成）；

ii. **转移目标历史表 branch target buffer，BTB**：利用历史数据来预测接下来的转移地址。BTB 由若干行构成（类似 cache），每一次，我们用 PC 的最后若干位作为索引，更新行内的信息（包括 PC 信息以及转移目标 target 信息等）；如果本次为不转移则没有必要记入 BTB，由于默认取指令的地址就是 PC+4。BTB 的预测仅根据上一条指令，如果上一次在这个分支中我们跳转到了 addr 这个地址，且在 BTB 中该行没有被其他转移指令覆盖，我们就预测这一次也会跳转到 addr 这个地址。简单来说，就是

```
Predict_PC = BTB[PC mod xxx].tag == PC ? BTB[PC mod xxx].target : PC + 4;
```

iii. **转移地址栈 return address stack，RAS**：用来解决函数调用以及函数返回时的转移问题。这类的转移问题是有规律的，我们可以用栈来实现。在调用函数时，执行

```
RAS.push(PC + 4);
```

在从函数返回时，执行

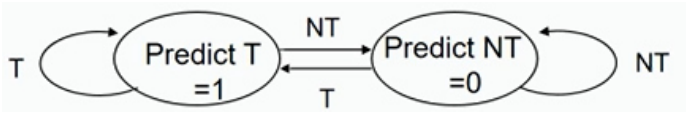
```
PC = RAS.pop()
```

问题 如何在译码阶段前得知这个指令为函数调用或函数返回指令？

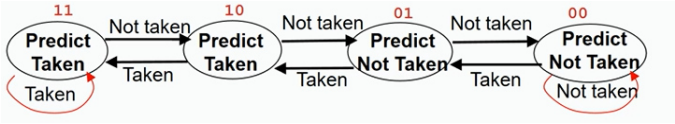
解决方案：利用另一个预测器来记录函数调用与函数返回的地址或直接在 BTB 中加入“return”的标记；也可以通过在指令内存中增加预译码位来解决。

iv. **转移历史表 branch history table，BHT**：通过历史数据来预测是否进行转移。类似 BTB，BHT 也由若干行构成，只不过每一行是一个转移预测器；每一次，我们用 PC 的最后若干位作为索引，利用本次的转移结果更新行内信息（转移预测器的信息）。有如下几种常见的转移预测器：

1. **一位预测器**：只有 Taken 与 Not Taken 两个状态，转移如下：



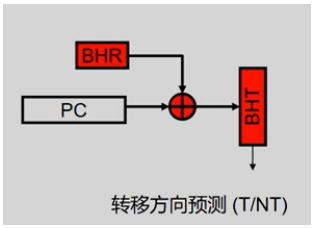
2. **二位预测器**：有 Strong Taken、Weak Taken、Weak Not Taken 与 Strong Not Taken 四种状态，前两种状态时预测 Taken，后两种状态时预测 Not Taken。转移如下（从左至右依次为这四种状态）：



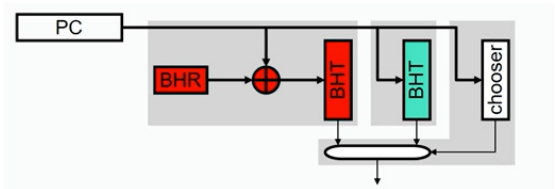
BHT 的缺点：无法通过历史规律来预测，难以利用局部相关性。

v. **转移历史寄存器 branch history register，BHR**：通过记录转移的历史信息（前几次转移的情况）来预测转移的结果。

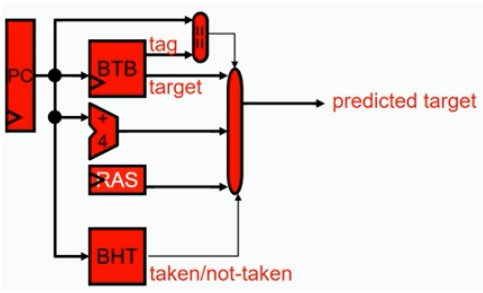
1. **转移历史规律表 pattern history table，PHT**：可以理解为对于每一种规律都有一个预测器，将 BHR 记录的规律送入 PHT 中进行预测得到结果，但是这样功耗大且需要大量空间；
2. **G-share 预测器 G-share branch prediction**：直接利用 $PC \oplus BHR$ 来搜索 BHT 即可，其中 $\oplus$ 表示异或；这种方式功耗小且效果不错，多被应用于嵌入式系统。



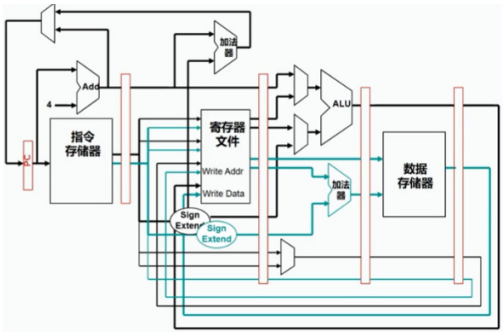
vi. **锦标赛预测器 championship predictor**：结合简单的预测器（如每行为一位/二位预测器的 BHT）与历史相关预测器（如 G-share 预测器），再加入一个选择器。一开始选择简单的预测器，超过阈值后选择历史相关预测器，类似于机器学习中的集成学习思想，可以达到较高的准确率。



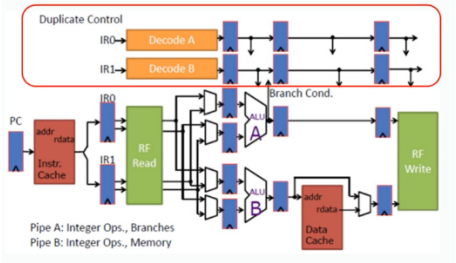
最终的分支预测器结果如下：



5. **进阶处理器设计**：指令级并行 ILP、线程级并行 TLP 与数据级并行 DLP。
   a) **指令级并行性 instruction level parallelism**：
      i. **超长指令字 very long instruction word，VLIW**（静态多发射处理器）：将若干可并行的（不相关的）指令合并成一个超长指令字进行发射执行，进而优化 CPI。VLIW 是由编译器选择并安排可以进行组合成新指令并发射的指令（必要时编译器会通过 nop 空指令填充）。
         1. 一个简单的 VLIW 例子是将 ALU 或 branch 处理指令与数据传送（内存读写）指令组合成新的指令进行发射，于是可能要修改寄存器文件的结构以及段寄存器的个数，如下图。

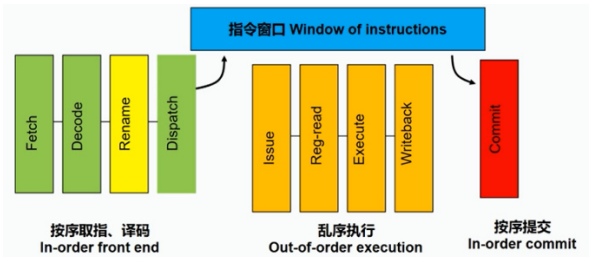


         2. **循环展开 loop unrolling**：通过循环展开来获得更多可以并行的指令，同时减少转移指令的数量；可能需要**对寄存器进行换名 registers renaming** 来消除“假”的由于循环展开引起的相关性。
         3. **缺点**：
            - **编译器复杂**：需要更复杂的编译器进行循环展开、冲突检测和指令预取，可能需要对即将访问的内存地址进行预测并且预取，这会导致很长的编译时间；
            - **代码膨胀**：增加了一些空指令，浪费空间；循环展开增加代码长度，浪费空间；同时需要更宽的指令通路（因为指令字长度增加）；
            - **锁步机制 lock step**：如果一个指令被阻塞了，接下来的所有指令都阻塞了；新的指令仅当不存在相关性时才被允许发射；
            - **兼容性差**：需要支持更复杂的指令，因此需要更加复杂的架构，同时需要特殊编译器，兼容性差，市场接受度不高。
      ii. **超标量 superscalar**（动态多发射处理器，分为按序超标量和乱序超标量）：利用硬件架构，在指令发射时同时发射多条指令，同时解决冲突问题。
      iii. **（静态）按序超标量 in-order superscalar**：在静态阶段，编译器尽力避免结构冒险以及数据相关；在指令发射阶段，多个指令流水线允许同时发射多条指令，动态判断本周期发射的指令条数，以解决结构冒险、数据冒险以及控制冒险。此外，按序超标量处理器发射时还会自动调换同时发射的若干指令的顺序，以适应体系架构（即将 lw 指令调度到处理存储的流水线上，将 add 指令调度到处理 ALU 的流水线上等等），如下图。（按序取指令译码，按序发射执行，按序提交）



      应用：ARM A8 处理器，常用于手机、笔记本电脑、嵌入式系统等。
      iv. **（动态）乱序超标量 out-of-order superscalar，OoO superscalar**：主要思想是执行速度快的指令并不需要等待执行速度慢的指令执行完成，如果当前指令与之前待执行或在执行的其他指令存在相关性，那么我们可以提前执行当前指令后的其他不相关的指令。
         1. **假数据相关**：乱序执行的后果是可能会产生假数据相关，如输出相关与反相关。
            - **输出相关 write-after-write，WAW**：两条指令输出到同一寄存器，如果前一条指令比较迟执行完成，那么最终寄存器的值可能会出错；
            - **反相关 write-after-read，WAR**：前一条指令读取某一寄存器的值，后一条指令输出到该寄存器，如果前一条指令比较晚执行完成，最终读到的寄存器的值可能会出错。

2. **寄存器换名 register renaming**：将存在假数据相关的两条寄存器进行换名（如均为 R0，则更名为 R0a 与 R0b），并且在合适的时候将换名的寄存器合并即可解决假数据相关。
3. **真数据相关 read-after-write，RAW**：前一条指令写某一个寄存器的值，后一条指令读取该寄存器的值，这类相关无法通过寄存器换名来解决，只能添加前向通路或者暂停当前指令的执行。
4. **乱序超标量流水线的一般结构**：按序取指令、译码；然后指令进入指令窗口乱序执行（注意：乱序执行时的数据读写操作均在缓冲区 buffer 中完成，不进行实际修改），最后按序提交（如果没有异常，则根据缓冲区的信息对真实的处理器状态进行更新（如对数据进行修改等）。（按序取指令译码、乱序发射执行、按序提交）。



5. **异常处理**：与流水线的异常检查机制相同，在按序提交前添加检查部分，检查指令的完成情况，如果发现异常则启动异常处理程序，并且清空后续指令的执行情况。

应用：如 IBM Power4 处理器。

v. **Tomasulo 算法**：纯硬件动态调度，达到高性能。

1. **保留站 reservation station，RS**：每一个功能单位 function unit，即 FU，的缓冲区，存储即将进入该功能单位执行的指令。
2. **寄存器换名 register renaming**：对于每一个指令，我们将其视作功能单位，并在每一个功能单位内提供了保留站。指令被暂存在这个保留站中，同时实现寄存器换名，将其结果寄存器用保留站坐标寄存器名字代替（如 Add 指令目标为 F0，且在 Add 功能单位的保留站 Add0 中，那么将后续对 F0 的读写重定向到对 Add0 的读写）。
3. **公共总线广播 common data bus broadcast，CDB broadcast**：每条指令的结果通过总线广播给每一个 FU，使其中对应的寄存器换名操作数得以更新为实际值。
4. **执行流程**：
   - **发射**：每一个周期开始时，只要当前指令的保留站存在空位（即无结构冒险），就发射指令并且进行寄存器的换名（寄存器换名如遇冲突，则执行覆盖，如 F0 已经被换为了 Add0，当前指令再将 F0 换为 Add1 是可行的）；否则，等待指令执行完成。
   - **执行**：每一个指令在缓冲区中，提供两套操作数选择，V_j, V_k 表示已经得到的操作数，Q_j, Q_k 表示通过寄存器名（可能是换名后的）表示的操作数；当一个指令的操作数均已为实际操作数时，则该条指令可以执行。
   - **写结果**：当一条指令执行完成，WB 阶段会通过广播通知其他保留站，将代表该指令的寄存器名替换为实际操作数，并触发所有可执行单元继续执行。
5. **重排序缓冲区 reorder buffer**：由于精确中断要求按序提交，而 Tomasulo 算法中每条指令的完成为乱序的，因此设置重排序缓冲区来延迟先执行完成的部分指令的提交阶段。指令先发射到重排序缓冲区中，重排序缓冲区存储下指令并执行寄存器换名（用重排序缓冲区的编号来进行寄存器换名即可，不需要再对每一个保留站进行编号）。指令执行完成后更新重排序缓冲区的执行状态与结果，并且通过重排序缓冲区将结果广播给其他保留站。对重排序缓冲区来说，仅当最早的指令可以提交时，才会开始按序提交指令。
6. **带重排序缓冲区的 Tomasulo 算法的规则**：
   - **发射 issue**：当 ①重排序缓冲区有空余槽；②对应的保留站有空余槽。操作：①占据保留站的空余槽，并且将操作数更改成寄存器名或确定值（如果可以得到的话）；②按序占据重排序缓冲区的空余槽。
   - **执行 execute**：当操作数都已为确定值，可以直接使用（或已经在 CDB 上广播）。操作：执行。
   - **写回 write back**：当 CDB 可用（没有其他指令在占据其广播）。操作：① 释放占据的保留站空余槽并在 CDB 上广播；② 将结果写回到重排序缓冲区的对应位置并且将重排序缓冲区的

对应本指令的寄存器换成执行结果；③ 将所有保留站中的对应本指令的寄存器换成执行结果。

- **提交 commit**：当重排序缓冲区的第一条指令可以提交时。操作：① 提交该指令，修改相应寄存器或内存数据；② 清零寄存器信息，释放重排序缓冲区的该行。

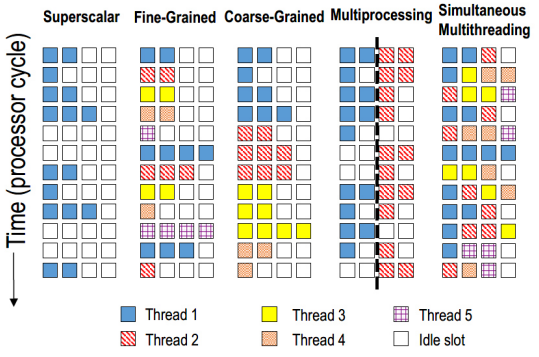
7. **显式寄存器换名 explicit register renaming**：用一个空闲表 freelist 存储全部可用的寄存器（独立于指令集可用的寄存器外的其他寄存器），并根据重排序缓冲区的行和寄存器之间建立映射关系，在重排序缓冲区中加入分配的换名寄存器信息，动态分配寄存器即可。**优点**在于：
   - 可以从单一寄存件文件中读取数据，不需要重排序缓冲区的旁路支持（否则，因为我们暂时不能提交结果，即往目标寄存器写入数据，数据只能暂存在重排序缓冲区中，后续需要从重排序缓冲区中通过旁路得到数据）；
   - 将寄存器换名与调度解耦，可以理解为更类似正常流水线的工作情况。

b) **线程级并行性 thread level parallelism，TLP**

i. **超标量的浪费 superscalar waste**：在于水平浪费 horizontal waste 与垂直浪费 vertical waste；水平浪费指可能无法在单周期内最大限度发射指令造成的浪费，垂直浪费指存在相关性的指令导致的完全空闲的发射周期的浪费。

ii. **多线程 multithreading**：在同一个流水线中，可以执行不同线程的指令来解决相关性带来的等待问题。带来的其他问题有：每个进程需要拥有自己的用户态信息（PC）、系统态信息（页表、异常处理等）与其他开销（如进程竞争导致的 cache/TLB 冲突或需要更大 TLB 容量，以及 OS 调度开销）。

1. **多核多线程 chip multiprocessing，CMP**：分成多个处理器进行处理。从单个核上看，并没有减少水平浪费和垂直浪费，但从宏观上看，由于发射宽度在核间进行了静态分配，总体浪费减少了。
2. **粗粒度多线程 coarse-grain multithreading，CGMT**：当线程运行时需要较长时间延迟时才切换到另一线程进行处理，同一线程指令间的较短延迟可能不进行切换。由于仅当出现了延迟 stall 时才进行上下文切换 context switch，因此至少浪费 1 个周期（因为最早要在发射时检查相关性），同时如果检查相关性过晚，可能会浪费多个周期，同时还需要进行流水线的部分重置。使用 CGMT 可以隐藏较长时间的延迟，而不能隐藏短时间的延迟，因此垂直浪费仍然存在；同时不能减少水平浪费的问题。（一般做题时最优情况默认无缝切换）
3. **细粒度多线程 fine-grain multithreading，FGMT**：多个线程的指令交叉执行（一般采取 round-robin 进行调度）。使用 FGMT 最优情况下可以隐藏所有的延迟，减少所有的垂直浪费，但是仍然不能减少水平浪费的问题。另一问题在于其可能需要非常多的线程，继而需要许多套的寄存器文件（因为每个线程需要一套独立的寄存器文件），因此其在商用上并不成功。
4. **同步多线程 simultaneous multithreading，SMT**：在乱序处理器上运行多线程的思想，线程数量较少，合并利用一个寄存器文件进行存储，利用 round-robin 调度（类似 FGMT），乱序执行。使用 SMT 最优情况下可以隐藏所有的延迟，减少所有的垂直浪费（原因和 FGMT 相同）；同时也能减少水平浪费（因为乱序执行可以减少水平浪费）。存在的问题：
   - Cache 冲突仍然存在（这一点是所有多线程无法避免的）；
   - 需要一个比较大的寄存器文件映射（将线程的寄存器映射到整个寄存器文件中）；
   - 需要对乱序执行的重排序缓冲区 ROB、保留站 RS 进行划分（静态划分 or 动态划分）。
5. **上述方法的比较**：单纯的超标量存在水平浪费与垂直浪费；FGMT 每周期切换一次，可以隐藏几乎所有的垂直浪费，无法解决水平浪费；CGMT 直到需要长延迟才切换，可以隐藏长的垂直浪费，无法解决水平浪费以及短的垂直浪费；CMT 宏观上减少了水平浪费与垂直浪费，但微观上并没有减少；SMT 即可以减少水平浪费（通过乱序），又可以减少垂直浪费（通过 FGMT 思想）。



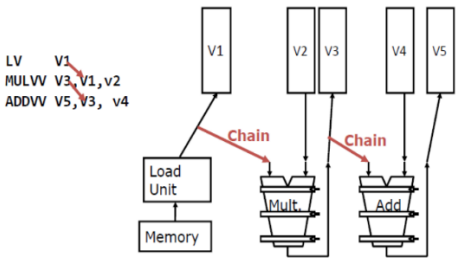
c) **数据级并行性 data level parallelism**

i. 指令流与数据流的关系：

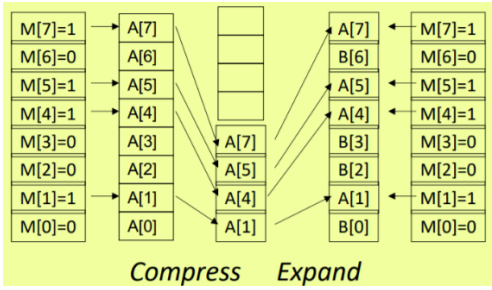
1. **SISD**：单指令流单数据流，如传统 CPU 作用于内存的指令流与单一数据流；
2. **SIMD**：单指令流多数据流，如向量机和 GPU 作用于指令流于多数据流；
3. **MISD**：多指令流单数据流，少见，冗余多用于容错系统；
4. **MIMD**：多指令流多数据流，类似于多个 SISD 系统，如多处理器计算机、企业级服务器等。

ii. **向量机 vector machine**：典型的 SIMD 架构

1. **基本思想**：将标量运算改为向量运算；
2. **基本特征**：一条指令包含多个操作 VSIW；单条指令中包含的操作是独立的（向量的每个单元的独立性）；可以更好利用多体交叉存储系统的传输数据的性质；控制相关少。
3. **基本结构**：向量指令并行执行，向量运算部件流水线执行，向量部件结构为多条运算流水线。
4. **向量链接 vector chaining**：类似于寄存器中的旁路，将上一个运算结果直接链接到下一个输入中。

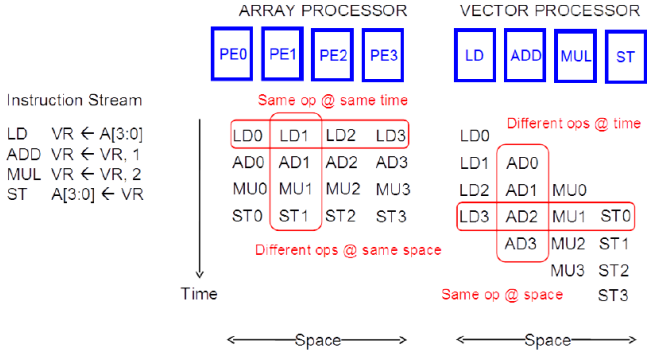


5. **掩码 mask bit**：通过掩码来约束向量机的操作。
   - **简单实现 simple implementation**：执行所有的操作，通过掩码来选择需要写回的操作；
   - **时间密度实现 density-time implementation**：通过掩码选择需要执行的操作，不执行其他操作。
   - **压缩扩展操作 compress-expand operation**：时间密度实现的一种方式，先将选择执行的操作进行压缩成一个连续表，再利用掩码将这个连续表映射回本来的位置。



iii. **数组机 array machine**：又称为并行处理机、SIMD 处理机，核心是多个处理单元构成的阵列，即可以用多个相同的处理单元并行处理多个元素，同时执行相同的指令；

1. **与向量机的区别**：在于向量机指包含单个处理单元，同时执行不同的指令，通过链接用一种类似于流水线的方式执行。



2. **与 VLIW 的区别**：VLIW 是多个独立的操作由编译器封装在一起，而数组机是单个操作作用在不同的数据元素上。

iv. **通用处理器的向量化扩展 vector extension，SIMD extension**

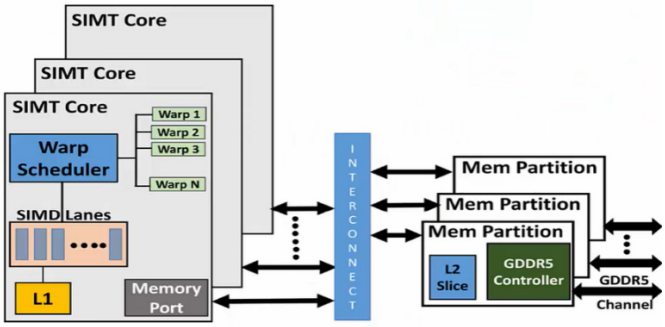
1. 通常为处理多媒体数据流所支持的指令；
2. 加入一些关于向量的寄存器支持，如把 64 位寄存器分割成长度位 8 的 8 位向量等；也可以增加

一些长度更长的寄存器，如 128 位、256 位的寄存器等等。

3. 加入一些向量指令支持。

v. **通用图像处理单元 graphic processing unit，GPU**

1. **SPMD**：单程序/流程多数据，是一种编程模型，即用单个程序在不同的单元上处理不同的数据；即可以应用在 MIMD 架构中，又可以运用在 SIMD 架构中（GPU）。
2. **GPU 体系架构**
   - **核 cores**（SIMT core，流多处理器 streaming multiprocessor）：GPU 由若干个核构成，每一个核是一个多线程的 SIMD 处理器；
   - **车道 lane**（硬件线程 hardware thread，流处理器 streaming processor）：一个核由若干条车道构成，每一条车道是一个单线程的 SIMD 处理器。
3. **CUDA 程序层次结构**
   - **网格 grid**：一个 kernel（程序）启动的所有线程；
   - **线程块 thread block**：网格可细分成若干线程块，一个线程块中包含很多线程；
   - **束 warp**（SIMD thread）：一个线程块可以分为若干束，其中 32 个线程为一组 warp，为执行相同指令的线程；
   - **线程 thread**：标量指令流。
4. **GPU 体系架构与 CUDA 程序层次结构的对应**
   - 一个线程块在一个核上执行；
   - 每个核有一个束队列以及一个束调度器，用来调度一束指令的执行；
   - 一次在一个核上执行一束指令；
   - 每次将一束准备好的指令分给所有车道，每个车道进行按序流水操作（为了隐藏延迟）；
   - 用束调度器 warp scheduler 来细粒度 fine-grain 调度所有的束，隐藏束的延迟，提高吞吐量；
   - 因此，可能需要非常大的寄存器文件（存储每个 warp 的寄存器，因为需要不断切换）。



5. **GPU 的三个核心思想：**
   - 多个处理核，充分利用 TLP；
   - 利用 SIMD 充分开发 DLP，多个 ALU 同步处理不同的数据，大大增强计算能力；
   - 用多线程提高系统的处理能力，并用细粒度多线程隐藏延迟。
6. **CUDA 存储模型**：每一个线程可以
   - 读/写线程的寄存器；
   - 读/写线程块的共享内存；
   - 读/写每一个网格的全局内存；
   - 读每一个网格的的常量内存。

   可以充分利用共享内存来加快访问速度。