

一种并行提取指令的方法与可读存储介质

技术领域

本发明涉及处理器技术领域，具体涉及一种并行提取指令的方法与可读存储介质。

背景技术

微处理器经过 50 多年的发展，微处理器的架构伴随着半导体工艺经历了蓬勃的发展。从单核到物理多核及逻辑多核；从顺序执行到乱序执行；从单发射到多发射；尤其在服务器领域，不断的追求处理器的性能。

目前，服务器芯片基本都是超标量乱序执行架构，处理器的处理带宽越来越高，达到每个时钟周期处理 8 条甚至更多的指令。

在取指单元，同时取多条指令时，串行的依次提取每条指令，逻辑链路比较长。目前高性能处理器每个时钟周期需要提取 8 条甚至更高的带宽，并且时钟频率要求比较高。目前的实现方法无法满足要求。

发明内容

针对现有技术的不足，本发明公开了一种并行提取指令的方法与可读存储介质，用于解决在取指单元，同时取多条指令时，串行的依次提取每条指令，逻辑链路比较长。目前高性能处理器每个时钟周期需要提取 8 条甚至更高的带宽，并且时钟频率要求比较高。目前的实现方法无法满足要求的问题。

本发明通过以下技术方案予以实现：

第一方面，本发明公开一种并行提取指令的方法，其特征在于，所述方法根据指令的结束位置向量 `s_mark_end`，产生提取指令有效向量，通过逻辑“与”和逻辑“或”运算，进行每个位置的指令并行译码、计算指令地址和分支指令目标地址运算，最终并行提取多条指令。

更进一步的，所述方法中，首先判断第 1 条指令的低 2bit，如果

低 2bit 为 00、01 或者 10 时，那么第 1 条指令长度为 16bit，如果低 2bit 为 11 时，那么第 1 条指令长度为 32bit，然后从该条指令结束位置的下一个 byte 开始判断第 2 条指令，判断过程与判断第 1 条指令类似，得到第 2 条指令的长度，以此类推，得到 cacheline 中每条指令的长度，得到每条指令的长度后，得到每条指令在指令流中结束位置向量 s_end_mark。

更进一步的，所述方法中，写指令时，计算得到每条指令的结束位置向量 s_end_mark，从写入方返回的指令以 cacheline 为单位，每个 cacheline 为 64byte，指令的高低 32byte 分别计算得到指令的结束位置向量，高 32byte 指令推测计算偏移为 0 和偏移为 2 的指令结束位置向量 s_end_mark_0 和 s_end_mark_1，根据低 32byte 指令结束位置向量选择一个高 32byte 向量，作为高 32byte 指令最终的指令结束向量，指令的结束位置向量和指令同时写入。

更进一步的，所述方法中，当取指单元开始取指时，读取指令时同时读取指令结束位置向量，去进行校验 BPU 的预测信息和提取指令，指令结束位置向量 s_mark_end，表示该位置是否为一条指令的结尾，为 1 时表示是一条指令的结束位置；为 0 表示不是一条指令的结束位置。

更进一步的，所述方法中，取指单元的带宽为 32byte 每个时钟周期，在取指令的同时，进行分支指令的跳转进行预测，根据分支指令的高 2byte 进行预测，如果预测分支指令发生跳转时，那么跳转到目标地址。当从目标地址的指令取回后，需要进行指令别名错误检查，即判定预测跳转的分支指令是否为一条分支指令，并且分支指令的类型也一致。

更进一步的，所述方法中，支持多个线程，并且所有的线程共享 BPU 预测单元，因此线程之间的预测信息会互相干扰，干扰的结果包括：

BPU 可能会把一条指令的中间内容，即不是分支指令的结尾，作为发生跳转的分支指令结束位置；

分支指令的类型不匹配，如果这个 BPU 信息时一条 JA 写入的，但是当一条 JALR 指令可能根据 JAL 的信息进行预测。

更进一步的，所述方法中，所述 BPU 信息包括 BPU 的预测偏移 pred_offset 与指令类型 pred_type，BPU 根据 BPU 预测的目标产生刷新，并且重新取指，在提取指令的时候，检测 s_mark_end[20] 是否为 1，若否，pred_offset 预测的位置不是一条分支指令的结束位置，而是一条指令的中间，则从 pred_offset 最近的一条指令结束的地址加 1 处，产生刷新，并且重新取指令，同时清除 BPU 中该条错误的预测信息。

更进一步的，所述方法中，如果在 pred_offset 是一条分支指令结束位置，但是取指的时候，同时判断 s_mark_end 对应的位置为一条分支指令，如果分支指令的类型和 BPU 预测的类型 pred_type 不同，此时也是别名错误，预测发生跳转的这条指令没有错误，但是预测的目标地址不正确，则从 pred_offset 加 1 的位置重新取指令，并且清除 BPU 中对应该位置的错误信息，只有当 BPU 预测的位置和类型都正确时，BPU 的预测信息才是正确的，否则需要产生刷新，从正确的地址重新取指令。

更进一步的，所述方法中，当每条指令已经从指令流中提取出来时，根据 BPU 的预测信息判断指令中是否存在分支指令，并且是否发生跳转，在指令中，如果存在多条分支指令时，第 1 条指令的优先级最高，其次是第 2 条指令，以此类推，根据分支指令分目标地址产生刷新，取指单元根据这个新的地址重新取指令，如果不存在分支指令时，所有的指令写入指令队列。

第二方面，本发明公开一种可读存储介质，包括存储有执行指令的存储器，当处理器执行所述存储器存储的所述执行指令时，所述处理器硬件执行第一方面所述的并行提取指令的方法。

本发明的有益效果为：

本发明根据指令的结束位置向量 s_mark_end，产生提取指令有效

向量，通过逻辑“与”和逻辑“或”运算，并行提取多条指令的一种方法。可以同时并行提取多条指令，每条指令之间不存在串行依赖关系，时序容易收敛，可以得到更高的主频。特别适合每个时钟周期提取 8 条指令以上的高性能处理器。

附图说明

为了更清楚地说明本发明实施例或现有技术中的技术方案，下面将对实施例或现有技术描述中所需要使用的附图作简单地介绍，显而易见地，下面描述中的附图仅仅是本发明的一些实施例，对于本领域普通技术人员来讲，在不付出创造性劳动的前提下，还可以根据这些附图获得其他的附图。

图 1 是本发明 RISC V 指令模式示意图；

图 2 是本发明实施例取指单元顶层图；

图 3 是本发明实施例指令边界标识图；

图 4 是本发明实施例指令结束位置向量图；

图 5 是本发明实施例跨边界指令发生跳转图；

图 6 是本发明实施例别名错误检查图；

图 7 是本发明实施例并行提取指令图；

图 8 是本发明实施例第 2 条指令产生逻辑图；

图 9 是本发明实施例计算指令地址和分支目标地址图

图 10 是本发明实施例跨边界指令图。

具体实施方式

为使本发明实施例的目的、技术方案和优点更加清楚，下面将结合本发明实施例中的附图，对本发明实施例中的技术方案进行清楚、完整地描述，显然，所描述的实施例是本发明一部分实施例，而不是全部的实施例。基于本发明中的实施例，本领域普通技术人员在没有作出创造性劳动前提下所获得的所有其他实施例，都属于本发明保护的范围。

实施例 1

本实施例是根据指令的结束位置向量 `s_mark_end`，产生提取指令有效向量，通过逻辑“与”和逻辑“或”运算，并行提取多条指令的一种方法。

本实施例不局限于 CPU, GPU, DSP 等芯片；不局限于任何指令集及任何实现工艺等条件。

为了便于说明本方法的原理，主要以 RISC-V 指令集为实例进行说明。

RISC V 指令集支持指令长度 16bit, 32bit, 48bit 和 64bit 等，如图 1 所示。本文主要以长度为 16bit 和 32bit 的指令为例，描述本文的提出的方法。为了便于说明本方法的原理，假设每次取指令带宽为 32byte，每次提取 8 条指令。

16bit 指令的最低 2bit 为 00、01 或者 10。32bit 指令的最低 2bit 为 11。因此在判断当前指令长度时，只需要判断指令的最低 2bit。首先判断第 1 条指令的低 2bit，如果低 2bit 为 00、01 或者 10 时，那么第 1 条指令长度为 16bit。如果低 2bit 为 11 时，那么第 1 条指令长度为 32bit。然后从该条指令结束位置的下一个 byte 开始判断第 2 条指令，判断过程与判断第 1 条指令类似，可以得到第 2 条指令的长度。以此类推，得到 cacheline 中每条指令的长度，如图 2 所示。得到每条指令的长度后，得到每条指令在指令流中结束位置向量 `s_end_mark`。

当指令从 L2 写入 L1 时，计算得到每条指令的结束位置向量 `s_end_mark`。从 L2 返回的指令以 cacheline 为单位，如图 3 所示，每个 cacheline 为 64byte，指令的高低 32byte 分别计算得到指令的结束位置向量。高 32byte 指令推测计算偏移为 0 和偏移为 2 的 2 种指令结束位置向量 `s_end_mark_0` 和 `s_end_mark_1`。根据低 32byte 指令结束位置向量选择一个高 32byte 向量，作为高 32byte 指令最终的指令结束向量，如图 3 所示。指令的结束位置向量和指令同时写入到 L1。

实施例 2

本实施例不局限于 CPU, GPU, DSP 等芯片; 不局限于任何指令集及任何实现工艺等条件。主要以 RISC-V 指令集为实例进行说明。当取指单元开始取指时, 读取 L1 CACHE 中的指令时, 指令结束位置向量同时读出, 去进行校验 BPU 的预测信息和提取指令。

指令结束位置向量 `s_mark_end`, 表示该位置是否为一个指令的结尾。为 1 时表示是一个指令的结束位置; 为 0 表示不是一个指令的结束位置, 即可能是指令的操作码或者是指令内部的立即数等。

在图 4 中第 1 条指令 LUI 长度为 4byte, `s_mark_end[28]` 为 1; 第 2 条指令为 C.ADDI, C.ADDI 是一条 16bit 的压缩指令, `s_mark_end[26]` 为 1; 第 3 条指令 AUIPC 长度为 4byte, `s_mark_end[22]` 为 1; 第 4 条指令 JAL 长度为 4byte, `s_mark_end[18]` 为 1; 第 5 条指令 LB 长度为 4byte, `s_mark_end[14]` 为 1; 第 6 条指令 LH 长度为 4byte, `s_mark_end[10]` 为 1; 第 7 条指令 ADDI 长度为 4byte, `s_mark_end[6]` 为 1; 第 8 条指令 SRAI 长度为 4byte, `s_mark_end[2]` 为 1; 第 9 条指令 BNE 长度为 4byte, 这条指令跨了 32byte, 因此指令 BNE 的指令结束位置不在当前指令块, 如图 4 所示。

取指单元的带宽为 32byte 每个时钟周期。由于支持 16bit/32bit 混合指令, 所以有可能一条分支指令跨 2 个相邻的指令块。分支指令的低 2byte 在一个 32byte 指令块 block0 的末尾, 同时高 2byte 在相邻指令块 block1 的头部, 如图 5 所示。

在取指令的同时, 进行分支指令的跳转进行预测。根据分支指令的高 2byte 进行预测, 如果预测分支指令发生跳转时, 那么跳转到目标地址。当从目标地址的指令取回后, 需要进行指令别名错误检查, 即判定预测跳转的分支指令是否为一个分支指令, 并且分支指令的类型也一致。

由于支持多个线程, 并且所有的线程共享 BPU 预测单元, 因此线程之间的预测信息会互相干扰。

干扰的结果包括：1，BPU 可能会把一条指令的中间内容，即不是分支指令的结尾，作为发生跳转的分支指令结束位置；

2，分支指令的类型不匹配，如果这个 BPU 信息时一条 JA 写入的，但是当一条 JALR 指令可能根据 JAL 的信息进行预测。

BPU 预测信息包括 BPU 的预测偏移 `pred_offset` 与指令类型 `pred_type`，如图 6 所示，`pred_offset` 为 5' d11, 即 BPU 预测图中的位置为一条分支指令的结束位置，并且发生跳转。

BPU 根据 BPU 预测的目标产生刷新，并且重新取指。在提取指令的时候，检测 `s_mark_end[20]` 是否为 1。实际发现 `s_mark_end[20]` 为 0，即 `pred_offset` 预测的位置不是一条分支指令的结束位置，而是一条指令的中间。

此时，需要从 `pred_offset` 最近的一条指令结束的地址加 1 处，产生刷新，并且重新取指令，同时清除 BPU 中该条错误的预测信息。同理，如果在 `pred_offset` 是一条分支指令结束位置，但是取指的时候，同时判断 `s_mark_end` 对应的位置为一条分支指令，如果分支指令的类型和 BPU 预测的类型 `pred_type` 不同，此时也是别名错误。

预测发生跳转的这条指令没有错误，但是预测的目标地址不正确，需要从 `pred_offset` 加 1 的位置重新取指令，并且清除 BPU 中对应该位置的错误信息。只有当 BPU 预测的位置和类型都正确时，BPU 的预测信息才是正确的，否则需要产生刷新，从正确的地址重新取指令。

本实施例根据指令结束向量并行产生提取 8 条指令有效向量，同时，32byte 指令并行推测译码、计算指令地址、计算指令的目标地址等等。然后，8 条指令有效向量和推测译码、计算指令地址、计算指令的目标地址等做“与”和“或”逻辑运算，得到提取的指令及相关属性，如图 7 所示。

实施例 3

本实施例以第 2 条指令的有效向量产生逻辑为例进行说明。`s_prt` 表示第 1 条指令在 32byte 指令流中的偏移。`s_mark_end` 表示 32byte

指令流中的指令结束位置向量，s_mark_end 的每个 bit 为 1 表示是一条指令的结束位置。Inst_2_val 表示第 2 条指令在 32byte 指令流中的有效向量，为 1 的位置表示第 2 条指令开始的 byte，从该位置取 4byte，即为一条完整指令（如果是 16bit 的压缩指令，也已经译码为 32bit 指令）。第 2 条指令的有效向量 inst_2_val 和推测译码得到的 16 条指令先做“与”运算，然后做“或”运算，得到第 2 条指令。

S_ptr 和 s_mark_end 组成一个 35bit 的指令位置标识向量，这个向量映射为另外一个 onehot 形式的向量 inst_2_val。产生第 2 条指令有效向量的逻辑映射关系如下表所示：

表格 1 第 2 条指令有效向量映射

第 2 条指令位置标识向量	第 2 条指令位置有效向量
00000xxxxxxxxxxxxxxxxxxxxxx10001000	00000000000000000000000000000000
00000xxxxxxxxxxxxxxxxxxxxxx101000	00000000000000000000000000000000
00000xxxxxxxxxxxxxxxxxxxxxx100010	00000000000000000000000000000000
00000xxxxxxxxxxxxxxxxxxxxxx1010	00000000000000000000000000000000
00010xxxxxxxxxxxxxxxxxxxxxx10001000xx	00000000000000000000000000000000
00010xxxxxxxxxxxxxxxxxxxxxx101000xx	00000000000000000000000000000000
00010xxxxxxxxxxxxxxxxxxxxxx100010xx	00000000000000000000000000000000
00010xxxxxxxxxxxxxxxxxxxxxx1010xx	00000000000000000000000000000000
00100xxxxxxxxxxxxxxxxxxxxxx10001000xxxx	00000000000000000000000000000000
00100xxxxxxxxxxxxxxxxxxxxxx101000xxxx	00000000000000000000000000000000
00100xxxxxxxxxxxxxxxxxxxxxx100010xxxx	00000000000000000000000000000000
00100xxxxxxxxxxxxxxxxxxxxxx1010xxxx	00000000000000000000000000000000
00110xxxxxxxxxxxxxxxxxxxxxx10001000xxxxxx	00000000000000000000000000000000
00110xxxxxxxxxxxxxxxxxxxxxx101000xxxxxx	00000000000000000000000000000000
00110xxxxxxxxxxxxxxxxxxxxxx100010xxxxxx	00000000000000000000000000000000
00110xxxxxxxxxxxxxxxxxxxxxx1010xxxxxx	00000000000000000000000000000000
01000xxxxxxxxxxxxxxxxxxxxxx10001000xxxxxx	00000000000000000000000000000000
01000xxxxxxxxxxxxxxxxxxxxxx101000xxxxxx	00000000000000000000000000000000
01000xxxxxxxxxxxxxxxxxxxxxx100010xxxxxx	00000000000000000000000000000000
01000xxxxxxxxxxxxxxxxxxxxxx1010xxxxxx	00000000000000000000000000000000
01010xxxxxxxxxxxxxxxxxxxxxx10001000xxxxxxxxxx	00000000000000000000000000000000
01010xxxxxxxxxxxxxxxxxxxxxx101000xxxxxxxxxx	00000000000000000000000000000000
01010xxxxxxxxxxxxxxxxxxxxxx100010xxxxxxxxxx	00000000000000000000000000000000
01010xxxxxxxxxxxxxxxxxxxxxx1010xxxxxxxxxx	00000000000000000000000000000000
01100xxxxxxxxxxxxxxxxxx10001000xxxxxxxxxxxxxx	00000000000000000000000000000000
01100xxxxxxxxxxxxxxxxxx101000xxxxxxxxxxxxxx	00000000000000000000000000000000
01100xxxxxxxxxxxxxxxxxx100010xxxxxxxxxxxxxx	00000000000000000000000000000000

01100xxxxxxxxxxxxxxxx1010xxxxxxxxxxxx	00000000000000001000000000000000
01110xxxxxxxxxx10001000xxxxxxxxxxxx	000000000000100000000000000000
01110xxxxxxxxxxxxxxxx101000xxxxxxxxxxxx	000000000000010000000000000000
01110xxxxxxxxxxxx100010xxxxxxxxxxxx	000000000000000100000000000000
01110xxxxxxxxxxxxxx1010xxxxxxxxxxxx	000000000000000010000000000000
10000xxxxxxxxxx10001000xxxxxxxxxxxx	000000000001000000000000000000
10000xxxxxxxxxx101000xxxxxxxxxxxx	000000000001000000000000000000
10000xxxxxxxxxx100010xxxxxxxxxxxx	000000000000100000000000000000
10000xxxxxxxxxx1010xxxxxxxxxxxx	000000000000100000000000000000
10010xxxxxx10001000xxxxxxxxxxxx	000000000100000000000000000000
10010xxxxxxxx101000xxxxxxxxxxxx	000000000100000000000000000000
10010xxxxxxxx100010xxxxxxxxxxxx	000000000010000000000000000000
10010xxxxxxxx1010xxxxxxxxxxxx	000000000010000000000000000000
10010xxxxxxxx1010xxxxxxxxxxxx	000000000010000000000000000000
10100xxx10001000xxxxxxxxxxxx	000000010000000000000000000000
10100xxxx101000xxxxxxxxxxxx	000000010000000000000000000000
10100xxxx100010xxxxxxxxxxxx	000000001000000000000000000000
10100xxxxxxxx1010xxxxxxxxxxxx	000000000100000000000000000000
10110xx10001000xxxxxxxxxxxx	000001000000000000000000000000
10110xxx101000xxxxxxxxxxxx	000001000000000000000000000000
10110xxx100010xxxxxxxxxxxx	000000010000000000000000000000
10110xxxxxxxx1010xxxxxxxxxxxx	000000010000000000000000000000
1100010001000xxxxxxxxxxxx	000100000000000000000000000000
11000xx101000xxxxxxxxxxxx	000100000000000000000000000000
11000xx100010xxxxxxxxxxxx	000001000000000000000000000000
11000xxx1010xxxxxxxxxxxx	000001000000000000000000000000
11010101000xxxxxxxxxxxx	010000000000000000000000000000
11010100010xxxxxxxxxxxx	000100000000000000000000000000
11010xx1010xxxxxxxxxxxx	000100000000000000000000000000
111001010xxxxxxxxxxxx	010000000000000000000000000000
00000xxxxxxxxxxxxxxxx10001000	000000000000000000000000010000
00000xxxxxxxxxxxxxxxx101000	000000000000000000000000010000
00000xxxxxxxxxxxxxxxx100010	000000000000000000000000001000
00000xxxxxxxxxxxxxxxx1010	000000000000000000000000000100
00010xxxxxxxxxxxxxxxx10001000xx	000000000000000000000000100000
00010xxxxxxxxxxxxxxxx101000xx	000000000000000000000000100000
00010xxxxxxxxxxxxxxxx100010xx	000000000000000000000000010000
00010xxxxxxxxxxxxxxxx1010xx	000000000000000000000000010000
00100xxxxxxxxxxxxxxxx10001000xxx	000000000000000000000001000000
00100xxxxxxxxxxxxxxxx101000xxx	000000000000000000000001000000
00100xxxxxxxxxxxxxxxx100010xxx	000000000000000000000000100000
00100xxxxxxxxxxxxxxxx1010xxx	000000000000000000000000100000
00110xxxxxxxxxxxxxxxx10001000xxx	000000000000000000001000000000
00110xxxxxxxxxxxxxxxx101000xxx	000000000000000000001000000000
00110xxxxxxxxxxxxxxxx100010xxx	000000000000000000000100000000

00110xxxxxxxxxxxxxxxxxxxx1010xxxxxx	000000000000000000000000000000000000
01000xxxxxxxxxxxxxxxxxxxx10001000xxxxxx	000000000000000000000000000000000000
01000xxxxxxxxxxxxxxxxxxxx101000xxxxxx	000000000000000000000000000000000000
01000xxxxxxxxxxxxxxxxxxxx100010xxxxxx	000000000000000000000000000000000000
01000xxxxxxxxxxxxxxxxxxxx1010xxxxxx	000000000000000000000000000000000000
01010xxxxxxxxxxxxxxxxxxxx10001000xxxxxx	000000000000000000000000000000000000
01010xxxxxxxxxxxxxxxxxxxx101000xxxxxx	000000000000000000000000000000000000
01010xxxxxxxxxxxxxxxxxxxx100010xxxxxx	000000000000000000000000000000000000
01010xxxxxxxxxxxxxxxxxxxx1010xxxxxx	000000000000000000000000000000000000
01100xxxxxxxxxxxxxxxxxxxx10001000xxxxxx	000000000000000000000000000000000000
01100xxxxxxxxxxxxxxxxxxxx101000xxxxxx	000000000000000000000000000000000000
01100xxxxxxxxxxxxxxxxxxxx100010xxxxxx	000000000000000000000000000000000000
01100xxxxxxxxxxxxxxxxxxxx1010xxxxxx	000000000000000000000000000000000000
01110xxxxxxxxxxxx10001000xxxxxxxxxxxxxx	000000000000000000000000000000000000
01110xxxxxxxxxxxx101000xxxxxxxxxxxxxx	000000000000000000000000000000000000
01110xxxxxxxxxxxx100010xxxxxxxxxxxxxx	000000000000000000000000000000000000
01110xxxxxxxxxxxx1010xxxxxxxxxxxxxx	000000000000000000000000000000000000
10000xxxxxxxx10001000xxxxxxxxxxxxxx	000000000000000000000000000000000000
10000xxxxxxxx101000xxxxxxxxxxxxxx	000000000000000000000000000000000000
10000xxxxxxxx100010xxxxxxxxxxxxxx	000000000000000000000000000000000000
10000xxxxxxxx1010xxxxxxxxxxxxxx	000000000000000000000000000000000000
10010xxxxxx10001000xxxxxxxxxxxxxx	000000000000000000000000000000000000
10010xxxxxxxx101000xxxxxxxxxxxxxx	000000000000000000000000000000000000
10010xxxxxxxx100010xxxxxxxxxxxxxx	000000000000000000000000000000000000
10010xxxxxxxx1010xxxxxxxxxxxxxx	000000000000000000000000000000000000
10100xxxx10001000xxxxxxxxxxxxxx	000000000000000000000000000000000000
10100xxxx101000xxxxxxxxxxxxxx	000000000000000000000000000000000000
10100xxxx100010xxxxxxxxxxxxxx	000000000000000000000000000000000000
10100xxxx1010xxxxxxxxxxxxxx	000000000000000000000000000000000000
10110xx10001000xxxxxxxxxxxxxx	000000000000000000000000000000000000
10110xxxx101000xxxxxxxxxxxxxx	000000000000000000000000000000000000
10110xxxx100010xxxxxxxxxxxxxx	000000000000000000000000000000000000
10110xxxx1010xxxxxxxxxxxxxx	000000000000000000000000000000000000
1100010001000xxxxxxxxxxxxxx	000100000000000000000000000000000000
11000xx101000xxxxxxxxxxxxxx	000100000000000000000000000000000000
11000xx100010xxxxxxxxxxxxxx	000001000000000000000000000000000000
11000xxxx1010xxxxxxxxxxxxxx	000001000000000000000000000000000000
11010101000xxxxxxxxxxxxxx	010000000000000000000000000000000000
11010100010xxxxxxxxxxxxxx	000100000000000000000000000000000000
11010xx1010xxxxxxxxxxxxxx	000100000000000000000000000000000000
111001010xxxxxxxxxxxxxx	010000000000000000000000000000000000

同理，可以得到其余指令的有效向量。

取指部件，每次译码 32byte，RISC-V 指令长度为 2 或者 4，因此

指令的操作码开始位置为 0,2,4, ……30 的偶数位置, 同理, 指令的结束位置为奇数位置 1,3,5, ……31。

如果指令是从位置 0 开始, 那么该指令的有效向量 `inst_2_val[0]` 为 1; 同时取推测译码得到的指令 `inst0`, 并且长度取 4byte, 当指令为一条 C 扩展指令时, 在推测译码时已经译码为长度为 4byte 的指令。

如果指令是从位置 2 开始, 那么该指令的有效向量 `inst_2_val[2]` 为 1; 同时取推测译码得到的指令 `inst1`。

如果指令是从位置 4 开始, 那么该指令的有效向量 `inst_2_val[4]` 为 1; 同时取推测译码得到的指令 `inst2`。

如果指令是从位置 6 开始, 那么该指令的有效向量 `inst_2_val[6]` 为 1; 同时取推测译码得到的指令 `inst3`。

如果指令是从位置 8 开始, 那么该指令的有效向量 `inst_2_val[8]` 为 1; 同时取推测译码得到的指令 `inst4`。

如果指令是从位置 10 开始, 那么该指令的有效向量 `inst_2_val[10]` 为 1; 同时取推测译码得到的指令 `inst5`。

如果指令是从位置 12 开始, 那么该指令的有效向量 `inst_2_val[12]` 为 1; 同时取推测译码得到的指令 `inst6`。

如果指令是从位置 14 开始, 那么该指令的有效向量 `inst_2_val[14]` 为 1; 同时取推测译码得到的指令 `inst7`。

如果指令是从位置 16 开始, 那么该指令的有效向量 `inst_2_val[16]` 为 1; 同时取推测译码得到的指令 `inst8`。

如果指令是从位置 18 开始, 那么该指令的有效向量 `inst_2_val[18]` 为 1; 同时取推测译码得到的指令 `inst9`。

如果指令是从位置 20 开始, 那么该指令的有效向量 `inst_2_val[20]` 为 1; 同时取推测译码得到的指令 `inst10`。

如果指令是从位置 22 开始, 那么该指令的有效向量 `inst_2_val[22]` 为 1; 同时取推测译码得到的指令 `inst11`。

如果指令是从位置 24 开始, 那么该指令的有效向量

inst_2_val[24] 为 1；同时取推测译码得到的指令 inst12。

如果指令是从位置 26 开始，那么该指令的有效向量 inst_2_val[26] 为 1；同时取推测译码得到的指令 inst13。

如果指令是从位置 28 开始，那么该指令的有效向量 inst_2_val[28] 为 1；同时取推测译码得到的指令 inst14。

果指令是从位置 30 开始，如果当前指令没有跨边界，那么该指令的有效向量 inst_2_val[30] 为 1；同时取推测译码得到的指令 inst15。

如果当前指令跨边界，那么当前指令无效，直到下一个 32byte 指令流有效时，才取这条指令。

如果第 1 条指令的偏移不是 0，而是从一个非 0 的偏移开始。那么第 1 条指令的开始位置是这个偏移的位置。其它指令的位置依次往后做相同的偏移开始。

得到第 2 条指令的逻辑表达式为：

```
Inst_2 = ({32{ inst_2_val[0]}}&inst0)|
          ({32{ inst_2_val[2]}}&inst1)|
          ({32{ inst_2_val[4]}}&inst2)|
          ({32{ inst_2_val[6]}}&inst3)|
          ({32{ inst_2_val[8]}}&inst4)|
          ({32{ inst_2_val[10]}}&inst5)|
          ({32{ inst_2_val[12]}}&inst6)|
          ({32{ inst_2_val[14]}}&inst7)|
          ({32{ inst_2_val[16]}}&inst8)|
          ({32{ inst_2_val[18]}}&inst9)|
          ({32{ inst_2_val[20]}}&inst10)|
          ({32{ inst_2_val[22]}}&inst11)|
          ({32{ inst_2_val[24]}}&inst12)|
          ({32{ inst_2_val[26]}}&inst13)|
          ({32{ inst_2_val[28]}}&inst14)|
          ({32{ inst_2_val[30]}}&inst15));
```

Inst0、inst1、……inst15 为 16 条推测产生的指令。第 2 条指令实现的电路为逻辑“与”和逻辑“或”门实现，如图 8 所示。其它指令，根据相同的原理，可以得到逻辑表达式和逻辑电路图。

实施例 4

本实施例计算指令的地址和目标地址时，也是推测计算。取指单元每次取 32byte，取指的地址为 `fetch_address`，即为计算指令地址的基地址。因为 RISC V 指令的长度为 2 或者 4，所以推测计算 16 个位置的指令地址分别为：`base_address`，`base_address+2`，`base_address+4`，`base_address+8`，`base_address+10`，`base_address+12`，`base_address+14`，`base_address+16`，`base_address+18`，`base_address+20`，`base_address+22`，`base_address+24`，`base_address+28` 和 `base_address+30`。第 2 条指令的地址 `inst_2_addr` 也是利用产生第 2 条指令类似的逻辑得到。如下所示：

```
Inst_2_addr = ({64{ inst_2_val[0]}} & base_address)|
               ({64{ inst_2_val[2]}} & (base_address+2))|
               ({64{ inst_2_val[4]}} & (base_address+4))|
               ({64{ inst_2_val[6]}} & (base_address+6))|
               ({64{ inst_2_val[8]}} & (base_address+8))|
               ({64{ inst_2_val[10]}} & (base_address+10))|
               ({64{ inst_2_val[12]}} & (base_address+12))|
               ({64{ inst_2_val[14]}} & (base_address+14))|
               ({64{ inst_2_val[16]}} & (base_address+16))|
               ({64{ inst_2_val[18]}} & (base_address+18))|
               ({64{ inst_2_val[20]}} & (base_address+20))|
               ({64{ inst_2_val[22]}} & (base_address+22))|
               ({64{ inst_2_val[24]}} & (base_address+24))|
               ({64{ inst_2_val[26]}} & (base_address+26))|
               ({64{ inst_2_val[28]}} & (base_address+28))|
               ({64{ inst_2_val[30]}} & (base_address+30)));
```

在取指单元取出的指令，指令中的分支指令包括 JAL，JALR，BEQ，BNE，BLT，BGE，BLTU，BGEU，C. JAL，C. J，C. BEQZ，C. BNEZ，C. JR，C. JALR。其中指令 JAL，BEQ，BNE，BLT，BGE，BLTU，BGEU，C. JAL，C. J，C. BEQZ，C. BNEZ 的目的地址是指令地址与偏移相加。同理，也假设每偏移 2 个 byte 为一条分支指令，因此也推测并行计算得到每条指令的目标地址。推测计算 16 个位置的指令的目标地址分别为：`base_address+offset`，`base_address+2+offset`，

base_address+4+offset, base_address+8+offset,
 base_address+10+offset, base_address+12+offset,
 base_address+14+offset, base_address+16+offset,
 base_address+18+offset, base_address+20+offset,
 base_address+22+offset, base_address+24+offset,
 base_address+28+offset 和 base_address+30+offset。Offset 为分支指令的偏移。Inst 为指令。

32bit 指令的条件指令立即数 cond_imm 为: cond_imm:
 {inst[31],inst[7],inst[30:25],inst[11:8],1'b0};

32bit 指令的无条件指令立即数为: uncond_imm:
 {inst[31],inst[19:12],inst[20],inst[30:21],1'b0};

16bit 压缩指令的条件指令立即数 cond_imm_c 为: cond_imm_c:
 {inst[12],inst[6:5],inst[2],inst[11:10],inst[4:3],1'b0};

16bit 压缩指令的无条件指令立即数 uncond_imm_c 为:
 uncond_imm_c:
 {inst[12],inst[8],inst[10:9],inst[6],inst[7],inst[2],inst[11],
 inst[5:3],1'b0};

每个位置都可能是这 4 种分支指令,因此每个位置先判断指令类型,然后计算不同类型的偏移 offset。第 2 条指令的目标地址 Inst_2_target_addr 也可以得到 Inst_2_addr 类似的逻辑表达式,如图 9 所示。

实施例 5

本实施例判断得到每个位置的具体分支指令类型 br_type。br_type[0]为 32bit 指令的条件指令;br_type[1]为 32bit 指令的无条件指令;br_type[2]为 16bit 指令的条件指令;br_type[3]为 16bit 指令的无条件指令。因此,分支指令的偏移 offset 根据 br_type 和 cond_imm、uncond_imm、cond_imm_c 和 uncond_imm_c 得到。

由于同时支持 16bit 和 32bit 的指令,指令流中存在 16bit 和

32bit 混合指令。每 32byte 指令包括 8-16 条指令，因此，一条 32bit 的指令可能存在指令跨连续相邻的 32byte 指令流。在指令提取模块中，用一个 2byte 寄存器保存 32byte 指令流的高 2byte，这 2byte 作为跨边界指令的 2byte。

同时判断当前 32byte 指令流是否发生了跨边界指令的这种情况，如果发生，需要产生一个跨边界指令有效的指示信号。当相邻的 32byte 指令块到达取指流水线阶段时，如果跨边界指令有效指示信号为 1 时，表示第 1 条指令存在跨边界的情况。此时，第一条指令由两部分组成，如图 10 所示。

如果跨边界指令有效指示信号为 0 时，表示第 1 条指令不存在跨边界的情况。第 1 条指令即为当前 32byte 指令块的第 1 条指令。其它指令依次从第 1 条指令的后续指令流中取到。当指令是一条分支指令出现跨边界时，指令的 BPU 预测信息也需要保存，直到相邻的指令流有效时，得到第 1 条指令的预测信息，类似于得到第 1 条指令的处理方法。

当每条指令已经从指令流中提取出来时，根据 BPU 的预测信息判断 8 条指令中是否存在分支指令，并且是否发生跳转。8 条指令中，如果存在多条分支指令时，第 1 条指令的优先级最高，其次是第 2 条指令，以此类推。根据分支指令分目标地址产生刷新，取指单元根据这个新的地址重新取指令。如果不存在分支指令时，所有的指令写入指令队列。

实施例 6

本实施例公开一种可读存储介质，包括存储有执行指令的存储器，当处理器执行所述存储器存储的所述执行指令时，所述处理器硬件执行并行提取指令的方法。

综上，本发明根据指令的结束位置向量 s_mark_end，产生提取指令有效向量，通过逻辑“与”和逻辑“或”运算，并行提取多条指令的一种方法。可以同时并行提取多条指令，每条指令之间不存在串行

依赖关系，时序容易收敛，可以得到更高的主频。特别适合每个时钟周期提取 8 条指令以上的高性能处理器。

以上实施例仅用以说明本发明的技术方案，而非对其限制；尽管参照前述实施例对本发明进行了详细的说明，本领域的普通技术人员应当理解：其依然可以对前述各实施例所记载的技术方案进行修改，或者对其中部分技术特征进行等同替换；而这些修改或者替换，并不使相应技术方案的本质脱离本发明各实施例技术方案的精神和范围。

权 利 要 求 书

1、一种并行提取指令的方法，其特征在于，所述方法根据指令的结束位置向量 `s_mark_end`，产生提取指令有效向量，通过逻辑“与”和逻辑“或”运算，进行每个位置的指令并行译码、计算指令地址和分支指令目标地址运算，最终并行提取多条指令。

2、根据权利要求1所述的并行提取指令的方法，其特征在于，所述方法中，首先判断第1条指令的低2bit，如果低2bit为00、01或者10时，那么第1条指令长度为16bit，如果低2bit为11时，那么第1条指令长度为32bit，然后从该条指令结束位置的下一个byte开始判断第2条指令，判断过程与判断第1条指令类似，得到第2条指令的长度，以此类推，得到cacheline中每条指令的长度，得到每条指令的长度后，得到每条指令在指令流中结束位置向量 `s_end_mark`。

3、根据权利要求1所述的并行提取指令的方法，其特征在于，所述方法中，写指令时，计算得到每条指令的结束位置向量 `s_end_mark`，从写入方返回的指令以cacheline为单位，每个cacheline为64byte，指令的高低32byte分别计算得到指令的结束位置向量，高32byte指令推测计算偏移为0和偏移为2的指令结束位置向量 `s_end_mark_0` 和 `s_end_mark_1`，根据低32byte指令结束位置向量选择一个高32byte向量，作为高32byte指令最终的指令结束向量，指令的结束位置向量和指令同时写入。

4、根据权利要求1所述的并行提取指令的方法，其特征在于，所述方法中，当取指单元开始取指时，读取指令时同时读取指令结束位置向量，去进行校验BPU的预测信息和提取指令，指令结束位置向量 `s_mark_end`，表示该位置是否为一个指令的结尾，为1时表示是一个指令的结束位置；为0表示不是一个指令的结束位置。

5、根据权利要求4所述的并行提取指令的方法，其特征在于，所述方法中，取指单元的带宽为32byte每个时钟周期，在取指令的同

时，进行分支指令的跳转进行预测，根据分支指令的高 2byte 进行预测，如果预测分支指令发生跳转时，那么跳转到目标地址。当从目标地址的指令取回后，需要进行指令别名错误检查，即判定预测跳转的分支指令是否为一分支指令，并且分支指令的类型也一致。

6、根据权利要求 1 所述的并行提取指令的方法，其特征在于，所述方法中，支持多个线程，并且所有的线程共享 BPU 预测单元，因此线程之间的预测信息会互相干扰，干扰的结果包括：

BPU 可能会把一条指令的中间内容，即不是分支指令的结尾，作为发生跳转的分支指令结束位置；

分支指令的类型不匹配，如果这个 BPU 信息时一条 JA 写入的，但是当一条 JALR 指令可能根据 JAL 的信息进行预测。

7、根据权利要求 6 所述的并行提取指令的方法，其特征在于，所述方法中，所述 BPU 信息包括 BPU 的预测偏移 `pred_offset` 与指令类型 `pred_type`，BPU 根据 BPU 预测的目标产生刷新，并且重新取指，在提取指令的时候，检测 `s_mark_end[20]` 是否为 1，若否，`pred_offset` 预测的位置不是一条分支指令的结束位置，而是一条指令的中间，则从 `pred_offset` 最近的一条指令结束的地址加 1 处，产生刷新，并且重新取指令，同时清除 BPU 中该条错误的预测信息。

8、根据权利要求 1 所述的并行提取指令的方法，其特征在于，所述方法中，如果在 `pred_offset` 是一条分支指令结束位置，但是取指的时候，同时判断 `s_mark_end` 对应的位置为一条分支指令，如果分支指令的类型和 BPU 预测的类型 `pred_type` 不同，此时也是别名错误，预测发生跳转的这条指令没有错误，但是预测的目标地址不正确，则从 `pred_offset` 加 1 的位置重新取指令，并且清除 BPU 中对应该位置的错误信息，只有当 BPU 预测的位置和类型都正确时，BPU 的预测信息才是正确的，否则需要产生刷新，从正确的地址重新取指令。

9、根据权利要求1所述的并行提取指令的方法，其特征在于，所述方法中，当每条指令已经从指令流中提取出来时，根据BPU的预测信息判断指令中是否存在分支指令，并且是否发生跳转，在指令中，如果存在多条分支指令时，第1条指令的优先级最高，其次是第2条指令，以此类推，根据分支指令分目标地址产生刷新，取指单元根据这个新的地址重新取指令，如果不存在分支指令时，所有的指令写入指令队列。

10、一种可读存储介质，包括存储有执行指令的存储器，当处理器执行所述存储器存储的所述执行指令时，所述处理器硬件执行如权利要求1-9中任一所述的并行提取指令的方法。