|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 00:00:00,090 --> 00:00:01,643 | 各位老师同学们 |  |
| 2 | 00:00:01,643 --> 00:00:02,820 | 大家下午好 |  |
| 3 | 00:00:02,860 --> 00:00:04,309 | 我是成元虎 |  |
| 4 | 00:00:04,309 --> 00:00:05,840 | 来自国防科技大学 |  |
| 5 | 00:00:05,841 --> 00:00:07,560 | 现在是一名硕士研究生 |  |
| 6 | 00:00:08,106 --> 00:00:09,996 | 然后非常高兴有机会 |  |
| 7 | 00:00:09,996 --> 00:00:11,740 | 在这里分享我们的工作 |  |
| 8 | 00:00:11,880 --> 00:00:13,800 | 我将向大家介绍一个 |  |
| 9 | 00:00:13,800 --> 00:00:16,080 | 我们设计并实现的一个 |  |
| 10 | 00:00:16,260 --> 00:00:19,259 | 低开销嵌入式RISC-V处理器核 |  |
| 11 | 00:00:19,415 --> 00:00:21,721 | 我们把它叫做RV16 |  |
| 12 | 00:00:29,231 --> 00:00:31,731 | 我将从下面四个方面 |  |
| 13 | 00:00:31,731 --> 00:00:32,960 | 来介绍我们的工作 |  |
| 14 | 00:00:32,961 --> 00:00:34,034 | 首先我介绍一下 |  |
| 15 | 00:00:34,034 --> 00:00:35,020 | 我们研究的背景 |  |
| 16 | 00:00:35,420 --> 00:00:37,460 | 刚刚很多老师都提到了 |  |
| 17 | 00:00:37,460 --> 00:00:38,053 | 就是说 |  |
| 18 | 00:00:38,634 --> 00:00:40,121 | 嵌入式和物联网 |  |
| 19 | 00:00:40,121 --> 00:00:41,275 | 可能会是 |  |
| 20 | 00:00:41,575 --> 00:00:44,385 | RISC-V指令集的一个破局点 |  |
| 21 | 00:00:44,565 --> 00:00:46,265 | 但是就目前 |  |
| 22 | 00:00:46,445 --> 00:00:47,734 | 嵌入式和物联网 |  |
| 23 | 00:00:47,734 --> 00:00:49,265 | 它们工作的环境来说 |  |
| 24 | 00:00:49,266 --> 00:00:52,134 | 它们对功耗和面积 |  |
| 25 | 00:00:52,134 --> 00:00:53,881 | 都有非常严格的要求 |  |
| 26 | 00:00:54,043 --> 00:00:54,550 | 所以说 |  |
| 27 | 00:00:54,968 --> 00:01:00,678 | 这就对RISC-V处理器设计的 |  |
| 28 | 00:01:00,678 --> 00:01:03,271 | 功耗和面积提出了一些挑战 |  |
| 29 | 00:01:03,885 --> 00:01:05,075 | 我们都知道 |  |
| 30 | 00:01:05,168 --> 00:01:07,309 | 一个处理器的面积 |  |
| 31 | 00:01:07,309 --> 00:01:08,290 | 和功耗开销 |  |
| 32 | 00:01:08,912 --> 00:01:10,693 | 很大程度上取决于 |  |
| 33 | 00:01:10,834 --> 00:01:12,996 | 它所支持的指令集 |  |
| 34 | 00:01:12,996 --> 00:01:15,718 | 和它所实现的微架构 |  |
| 35 | 00:01:16,737 --> 00:01:18,043 | 在指令集方面 |  |
| 36 | 00:01:18,109 --> 00:01:20,250 | RISC-V它很简洁 |  |
| 37 | 00:01:20,251 --> 00:01:21,640 | 所以说它非常适合 |  |
| 38 | 00:01:21,930 --> 00:01:23,730 | 用来实现一些 |  |
| 39 | 00:01:23,730 --> 00:01:25,237 | 低开销的处理器核 |  |
| 40 | 00:01:25,531 --> 00:01:27,353 | 比如说前面提到的 |  |
| 41 | 00:01:27,353 --> 00:01:28,587 | 蜂鸟的E203 |  |
| 42 | 00:01:28,587 --> 00:01:29,909 | 以及Zero-riscy |  |
| 43 | 00:01:29,909 --> 00:01:32,915 | 他们都是这类低开销处理器 |  |
| 44 | 00:01:33,600 --> 00:01:35,750 | 但是就目前来说 |  |
| 45 | 00:01:35,990 --> 00:01:37,809 | 相较于Cortex-M0 |  |
| 46 | 00:01:37,809 --> 00:01:40,631 | 以及ARM的Cortex-M0 |  |
| 47 | 00:01:40,631 --> 00:01:43,731 | 它们的微架构 |  |
| 48 | 00:01:43,731 --> 00:01:45,365 | 都已经非常的简单 |  |
| 49 | 00:01:45,484 --> 00:01:48,595 | 两到三级的一个顺序流水线 |  |
| 50 | 00:01:48,925 --> 00:01:50,695 | 功能也非常的简洁 |  |
| 51 | 00:01:50,828 --> 00:01:52,975 | 我们已经很难通过 |  |
| 52 | 00:01:53,121 --> 00:01:57,768 | 简单的通过简化它们的结构来 |  |
| 53 | 00:01:58,068 --> 00:02:00,159 | 优化处理器的面积开销 |  |
| 54 | 00:02:00,615 --> 00:02:03,134 | 我们考虑到 |  |
| 55 | 00:02:03,471 --> 00:02:04,828 | 处理器内部 |  |
| 56 | 00:02:04,828 --> 00:02:06,986 | 它的功能单元的面积 |  |
| 57 | 00:02:06,986 --> 00:02:09,228 | 是和它的一个位宽是相关的 |  |
| 58 | 00:02:09,453 --> 00:02:10,606 | 所以说我们就 |  |
| 59 | 00:02:10,606 --> 00:02:12,700 | 提出了一个RV16的架构 |  |
| 60 | 00:02:13,956 --> 00:02:16,960 | 所谓的RV16这个名字来源 |  |
| 61 | 00:02:16,960 --> 00:02:19,596 | 就是因为它是一个基于 |  |
| 62 | 00:02:19,596 --> 00:02:21,520 | 16位数据通路实现的一个 |  |
| 63 | 00:02:21,640 --> 00:02:22,700 | 32位处理器 |  |
| 64 | 00:02:23,056 --> 00:02:26,140 | 它现在我们设计是支持 |  |
| 65 | 00:02:26,141 --> 00:02:27,518 | 16位的地址空间 |  |
| 66 | 00:02:28,271 --> 00:02:29,280 | 在设计过程中 |  |
| 67 | 00:02:29,281 --> 00:02:30,960 | 我们考虑到一个 |  |
| 68 | 00:02:31,062 --> 00:02:34,905 | 在嵌入式它可能应用的场景 |  |
| 69 | 00:02:34,905 --> 00:02:36,580 | 可能非常的广泛 |  |
| 70 | 00:02:36,756 --> 00:02:37,478 | 所以说我们就 |  |
| 71 | 00:02:37,990 --> 00:02:40,380 | 采用了一个可配置的实现方式 |  |
| 72 | 00:02:40,600 --> 00:02:41,943 | 它可配置的支持 |  |
| 73 | 00:02:41,975 --> 00:02:45,553 | RISC-V的 E I M和C扩展 |  |
| 74 | 00:02:46,281 --> 00:02:47,746 | 然后在微架构上面 |  |
| 75 | 00:02:47,965 --> 00:02:50,621 | RV16我们实现的是一个 |  |
| 76 | 00:02:50,621 --> 00:02:52,221 | 非常简单的两级流水线 |  |
| 77 | 00:02:52,593 --> 00:02:55,234 | 第一段流水线就是它的 |  |
| 78 | 00:02:56,390 --> 00:02:58,456 | 指令的取址和对齐 |  |
| 79 | 00:02:58,865 --> 00:03:03,800 | 为了保证每个周期它可以从 |  |
| 80 | 00:03:04,337 --> 00:03:06,340 | 指令存储器中取得一条指令 |  |
| 81 | 00:03:06,340 --> 00:03:09,356 | 因为RISC-V它不能仅仅 |  |
| 82 | 00:03:09,356 --> 00:03:11,690 | 只支持16位的指令 |  |
| 83 | 00:03:11,730 --> 00:03:13,346 | 所以说我们就把它的 |  |
| 84 | 00:03:13,856 --> 00:03:15,140 | 指令的数据通路 |  |
| 85 | 00:03:15,418 --> 00:03:18,055 | 依然保持为32位的 |  |
| 86 | 00:03:18,334 --> 00:03:21,568 | 然后在指令对齐单元就用来 |  |
| 87 | 00:03:21,926 --> 00:03:23,862 | 预译码它的压缩指令 |  |
| 88 | 00:03:23,862 --> 00:03:27,987 | 也就是将16位的压缩指令 |  |
| 89 | 00:03:28,521 --> 00:03:32,453 | 译码成32位的非压缩指令 |  |
| 90 | 00:03:32,668 --> 00:03:34,560 | 在这个指令对齐单元中 |  |
| 91 | 00:03:34,561 --> 00:03:37,353 | 为了尽可能的降低它的开销 |  |
| 92 | 00:03:37,525 --> 00:03:39,520 | 我们只在这里面增加了一个 |  |
| 93 | 00:03:39,625 --> 00:03:40,940 | 16位的寄存器 |  |
| 94 | 00:03:41,287 --> 00:03:44,356 | 用来保存本周期 |  |
| 95 | 00:03:44,356 --> 00:03:46,021 | 暂时没有使用到的数据 |  |
| 96 | 00:03:46,134 --> 00:03:48,546 | 如果它这个周期取到32位数据 |  |
| 97 | 00:03:48,546 --> 00:03:50,580 | 它有可能是一条16位的指令 |  |
| 98 | 00:03:50,581 --> 00:03:53,196 | 就有可能它的16位数据不会用到 |  |
| 99 | 00:03:53,471 --> 00:03:55,943 | 这个数据就需要保存到下一个周期 |  |
| 100 | 00:03:56,140 --> 00:03:59,356 | 和下一个周期取到的数据来拼接 |  |
| 101 | 00:03:59,810 --> 00:04:01,345 | 或者单独的16位 |  |
| 102 | 00:04:01,345 --> 00:04:02,850 | 或者就是一个32位的指令 |  |
| 103 | 00:04:04,425 --> 00:04:06,300 | 然后预译码就是在 |  |
| 104 | 00:04:06,300 --> 00:04:09,693 | 我们图中的一个C2I单元来实现的 |  |
| 105 | 00:04:11,203 --> 00:04:16,560 | 后面过了指令取址和对齐之后 |  |
| 106 | 00:04:16,560 --> 00:04:19,679 | 就是一个指令的译码和执行段 |  |
| 107 | 00:04:20,062 --> 00:04:20,887 | 在这个段 |  |
| 108 | 00:04:21,059 --> 00:04:23,100 | 和传统的32位的 |  |
| 109 | 00:04:23,209 --> 00:04:25,708 | RISC-V处理器就有一定的区别的 |  |
| 110 | 00:04:25,787 --> 00:04:27,390 | 因为它这里面主要的 |  |
| 111 | 00:04:27,390 --> 00:04:28,880 | 数据通路是16位的 |  |
| 112 | 00:04:29,115 --> 00:04:30,721 | 为了通过这个 |  |
| 113 | 00:04:30,721 --> 00:04:32,568 | 16位数据通路来实现 |  |
| 114 | 00:04:32,975 --> 00:04:34,268 | 32位的处理器 |  |
| 115 | 00:04:34,340 --> 00:04:36,170 | 支持32位的指令 |  |
| 116 | 00:04:36,171 --> 00:04:39,278 | 所以说我们就通过 |  |
| 117 | 00:04:39,570 --> 00:04:41,075 | 分多周期复用 |  |
| 118 | 00:04:41,075 --> 00:04:42,700 | 这个功能单元来实现的 |  |
| 119 | 00:04:43,187 --> 00:04:45,960 | 在这个流水段中主要包括 |  |
| 120 | 00:04:45,993 --> 00:04:47,465 | 主要的执行功能 |  |
| 121 | 00:04:47,465 --> 00:04:49,600 | 就是译码执行以及访存 |  |
| 122 | 00:04:49,601 --> 00:04:51,753 | 还有结果的写回 |  |
| 123 | 00:04:53,656 --> 00:04:55,309 | 首先介绍一下 |  |
| 124 | 00:04:55,309 --> 00:04:57,365 | RV16的译码器 |  |
| 125 | 00:04:57,528 --> 00:05:01,031 | 译码器它主要的功能依然是 |  |
| 126 | 00:05:01,165 --> 00:05:03,105 | 译码指令以及取操作数 |  |
| 127 | 00:05:03,243 --> 00:05:04,665 | 但是这里它和 |  |
| 128 | 00:05:04,766 --> 00:05:07,740 | 传统的32位处理器有一定的区别 |  |
| 129 | 00:05:07,921 --> 00:05:11,505 | 它的译码的情况它不仅仅取决于 |  |
| 130 | 00:05:11,593 --> 00:05:13,225 | 本周期输入的指令 |  |
| 131 | 00:05:13,226 --> 00:05:15,685 | 它还和指令的执行状态相关 |  |
| 132 | 00:05:15,686 --> 00:05:18,975 | 也就是图中的high\_16的信号 |  |
| 133 | 00:05:19,309 --> 00:05:21,850 | 这个信号是用来说明 |  |
| 134 | 00:05:22,462 --> 00:05:24,359 | 本周期应该处理的是 |  |
| 135 | 00:05:24,359 --> 00:05:25,731 | 32位数据当中的 |  |
| 136 | 00:05:26,212 --> 00:05:28,646 | 高16位数据还是低16位数据 |  |
| 137 | 00:05:29,446 --> 00:05:33,370 | 对于一条普通的加法来说 |  |
| 138 | 00:05:33,371 --> 00:05:34,490 | 肯定在第一个周期 |  |
| 139 | 00:05:34,490 --> 00:05:35,790 | 我们处理它的低16位 |  |
| 140 | 00:05:35,791 --> 00:05:38,128 | 在第二个周期处于它的高16位 |  |
| 141 | 00:05:38,293 --> 00:05:40,409 | 这样就通过 |  |
| 142 | 00:05:40,975 --> 00:05:43,075 | high\_16这个信号来说明 |  |
| 143 | 00:05:43,075 --> 00:05:44,493 | 数据的选择 |  |
| 144 | 00:05:45,256 --> 00:05:47,890 | 在接下来的所有图中 |  |
| 145 | 00:05:47,891 --> 00:05:50,046 | 我们都有用蓝色的框 |  |
| 146 | 00:05:50,046 --> 00:05:51,528 | 来突出的一个就是 |  |
| 147 | 00:05:51,750 --> 00:05:53,718 | 我们RV16和传统的 |  |
| 148 | 00:05:53,718 --> 00:05:55,187 | 32位处理器的区别 |  |
| 149 | 00:05:56,825 --> 00:05:59,953 | 后面就是寄存器文件 |  |
| 150 | 00:06:00,271 --> 00:06:05,150 | 其实在RV16中可以考虑有两种 |  |
| 151 | 00:06:05,450 --> 00:06:07,003 | 寄存器文件 |  |
| 152 | 00:06:07,120 --> 00:06:09,546 | 第一种它是基于32位的 |  |
| 153 | 00:06:10,000 --> 00:06:13,034 | 这种32位的它保持 |  |
| 154 | 00:06:13,034 --> 00:06:15,841 | 每个寄存器32位宽度不变 |  |
| 155 | 00:06:15,841 --> 00:06:17,780 | 通过一个控制信号来选择 |  |
| 156 | 00:06:17,781 --> 00:06:19,260 | 它的高16位或者低16位 |  |
| 157 | 00:06:19,583 --> 00:06:20,728 | 第二种方式就是 |  |
| 158 | 00:06:20,728 --> 00:06:22,912 | 基于16位的寄存器 |  |
| 159 | 00:06:23,040 --> 00:06:26,205 | 通过增加寄存器的个数 |  |
| 160 | 00:06:26,240 --> 00:06:29,440 | 使用两个16位寄存器 |  |
| 161 | 00:06:29,440 --> 00:06:31,125 | 来表示一个32位的数据 |  |
| 162 | 00:06:31,481 --> 00:06:33,859 | 我们经过简单的实验 |  |
| 163 | 00:06:34,425 --> 00:06:40,425 | 通过基于16位的计算机宽度 |  |
| 164 | 00:06:40,426 --> 00:06:42,865 | 它的开销是更小的 |  |
| 165 | 00:06:42,866 --> 00:06:46,426 | 在我们是实现的计算机文件当中 |  |
| 166 | 00:06:46,426 --> 00:06:48,531 | 都是基于16位的数据通路 |  |
| 167 | 00:06:49,193 --> 00:06:54,393 | 为了支持因为是16位的寄存器 |  |
| 168 | 00:06:54,393 --> 00:06:56,650 | 所以说它的个数就要翻倍 |  |
| 169 | 00:06:56,690 --> 00:06:57,437 | 所以说我们就 |  |
| 170 | 00:06:57,637 --> 00:06:59,703 | 额外增加了一个最低位来表示 |  |
| 171 | 00:07:00,256 --> 00:07:03,175 | 应该叫做 |  |
| 172 | 00:07:03,175 --> 00:07:05,290 | 来寻址这个计算器文件 |  |
| 173 | 00:07:05,631 --> 00:07:08,390 | 就相对于传统的 |  |
| 174 | 00:07:08,390 --> 00:07:10,030 | RISC-V处理器的使用 |  |
| 175 | 00:07:10,030 --> 00:07:12,918 | 它的寄存器号就会多一位 |  |
| 176 | 00:07:14,140 --> 00:07:20,434 | 比如说如果 |  |
| 177 | 00:07:20,434 --> 00:07:22,280 | 加法指令它使用低位的时候 |  |
| 178 | 00:07:22,281 --> 00:07:24,915 | 它其实是根据high\_16来的 |  |
| 179 | 00:07:24,960 --> 00:07:26,621 | 如果high\_16它是0 |  |
| 180 | 00:07:26,696 --> 00:07:27,680 | 那么它低位是0 |  |
| 181 | 00:07:27,809 --> 00:07:29,840 | 当然它也有一些特殊情况 |  |
| 182 | 00:07:29,841 --> 00:07:31,568 | 可能是要先处理高位 |  |
| 183 | 00:07:31,718 --> 00:07:33,412 | 那我们后面会介绍的 |  |
| 184 | 00:07:34,250 --> 00:07:38,035 | 接下来就是它的主要的执行单元 |  |
| 185 | 00:07:38,036 --> 00:07:39,415 | 执行单元主要是ALU |  |
| 186 | 00:07:39,415 --> 00:07:41,735 | 以及它的存储法单元 |  |
| 187 | 00:07:41,736 --> 00:07:43,755 | 其实ALU |  |
| 188 | 00:07:44,071 --> 00:07:47,734 | 在逻辑和比较逻辑 |  |
| 189 | 00:07:47,734 --> 00:07:48,745 | 都是比较简单的 |  |
| 190 | 00:07:48,746 --> 00:07:49,925 | 我这里就不展开介绍了 |  |
| 191 | 00:07:50,106 --> 00:07:52,012 | 在加法器中 |  |
| 192 | 00:07:52,128 --> 00:07:53,785 | 因为分成两个周期 |  |
| 193 | 00:07:53,789 --> 00:07:55,337 | 所以需要考虑它的一个 |  |
| 194 | 00:07:55,337 --> 00:07:56,340 | 低位的进位 |  |
| 195 | 00:07:56,637 --> 00:07:58,465 | 这个也是比较简单的 |  |
| 196 | 00:07:58,466 --> 00:08:00,925 | 我们主要介绍一下它的移位器 |  |
| 197 | 00:08:01,578 --> 00:08:03,571 | 其实移位器因为 |  |
| 198 | 00:08:03,796 --> 00:08:04,705 | 在移位操作中 |  |
| 199 | 00:08:04,706 --> 00:08:07,050 | 它的数据的高低16位数据 |  |
| 200 | 00:08:07,050 --> 00:08:09,325 | 它结果是互相影响的 |  |
| 201 | 00:08:09,453 --> 00:08:11,021 | 因为低位数据可能会 |  |
| 202 | 00:08:11,021 --> 00:08:12,318 | 进入高16位的结果 |  |
| 203 | 00:08:12,585 --> 00:08:14,096 | 所以说在这里我们就 |  |
| 204 | 00:08:14,096 --> 00:08:15,065 | 依然使用了一个 |  |
| 205 | 00:08:15,245 --> 00:08:16,531 | 32位的移位器 |  |
| 206 | 00:08:16,531 --> 00:08:18,837 | 来实现我们的移位操作 |  |
| 207 | 00:08:18,946 --> 00:08:21,743 | 但是由于我们数据通路是16位的 |  |
| 208 | 00:08:21,743 --> 00:08:24,259 | 所以说我们就通过了组合了 |  |
| 209 | 00:08:24,415 --> 00:08:27,500 | 移位器它可能是有三个 |  |
| 210 | 00:08:27,500 --> 00:08:31,037 | 操作输入端口的 |  |
| 211 | 00:08:31,378 --> 00:08:33,025 | 通过每周期组合 |  |
| 212 | 00:08:33,025 --> 00:08:35,143 | 两个16位的操作数 |  |
| 213 | 00:08:35,149 --> 00:08:37,666 | 来形成一个32位的移位数据 |  |
| 214 | 00:08:37,666 --> 00:08:40,628 | 来对32位的移位数据进行移位 |  |
| 215 | 00:08:40,900 --> 00:08:43,446 | 最后取16位的结果来作为 |  |
| 216 | 00:08:43,490 --> 00:08:47,168 | 本周结果写入到目标寄存器当中 |  |
| 217 | 00:08:47,593 --> 00:08:51,325 | 我们以算数右移为例 |  |
| 218 | 00:08:52,046 --> 00:08:54,084 | 比如说在第一个周期 |  |
| 219 | 00:08:54,171 --> 00:08:56,721 | 我们首先是要 |  |
| 220 | 00:08:57,053 --> 00:09:00,378 | 对于右移操作是高位操作数 |  |
| 221 | 00:09:00,378 --> 00:09:02,175 | 可能会影响低位的结果 |  |
| 222 | 00:09:02,221 --> 00:09:04,778 | 我们就先对它的高位进行移位 |  |
| 223 | 00:09:05,050 --> 00:09:05,815 | 这样就是 |  |
| 224 | 00:09:07,162 --> 00:09:09,790 | 就让op1等于它的高16位 |  |
| 225 | 00:09:10,334 --> 00:09:12,615 | 它op3就等于它的符号位 |  |
| 226 | 00:09:12,616 --> 00:09:15,035 | 这样对形成一个32位的数据 |  |
| 227 | 00:09:15,036 --> 00:09:17,987 | 进行移位操作之后取低16位数据 |  |
| 228 | 00:09:17,987 --> 00:09:20,615 | 作为本周期的结果写入到它的 |  |
| 229 | 00:09:21,009 --> 00:09:22,755 | 目标的高16位寄存器当中 |  |
| 230 | 00:09:22,821 --> 00:09:23,675 | 在第二个周期 |  |
| 231 | 00:09:23,676 --> 00:09:25,555 | 我们就对它的低16位进行操作 |  |
| 232 | 00:09:25,775 --> 00:09:27,834 | 然后把op1设置成低位 |  |
| 233 | 00:09:28,209 --> 00:09:31,375 | 就是它的原操作数的低16位 |  |
| 234 | 00:09:31,376 --> 00:09:33,034 | 然后把op3设置为 |  |
| 235 | 00:09:33,034 --> 00:09:34,455 | 原操作数的高16位 |  |
| 236 | 00:09:35,015 --> 00:09:37,535 | 这样一来就可以完成这一次操作 |  |
| 237 | 00:09:38,028 --> 00:09:42,312 | 这样设计的目的是因为可以保证 |  |
| 238 | 00:09:42,312 --> 00:09:45,455 | 寄存器端口依然是每周期两个读操作 |  |
| 239 | 00:09:46,371 --> 00:09:50,084 | 不需要额外增加寄存器的读端口 |  |
| 240 | 00:09:50,265 --> 00:09:52,631 | 这样就可以实现32位的移位操作 |  |
| 241 | 00:09:53,634 --> 00:09:55,535 | 然后在乘除法器的话 |  |
| 242 | 00:09:55,568 --> 00:09:58,000 | 乘法器我们也采取了一种 |  |
| 243 | 00:09:58,000 --> 00:09:59,046 | 可配置的策略 |  |
| 244 | 00:09:59,256 --> 00:10:02,650 | 一种是使用16位的乘法器 |  |
| 245 | 00:10:02,650 --> 00:10:06,320 | 来实现32位的乘法 |  |
| 246 | 00:10:06,320 --> 00:10:09,068 | 通过三个周期的乘法操作 |  |
| 247 | 00:10:09,068 --> 00:10:09,956 | 或者四个周期 |  |
| 248 | 00:10:10,240 --> 00:10:11,620 | 第二种就是慢速惩罚 |  |
| 249 | 00:10:11,620 --> 00:10:13,415 | 采用累加的操作 |  |
| 250 | 00:10:13,603 --> 00:10:15,860 | 移位器依然使用了一个 |  |
| 251 | 00:10:16,060 --> 00:10:18,262 | 32位的多周期的除法器 |  |
| 252 | 00:10:18,262 --> 00:10:22,365 | 通过两个周期送入操作数 |  |
| 253 | 00:10:24,056 --> 00:10:26,440 | 对于指令我们也提出了 |  |
| 254 | 00:10:26,441 --> 00:10:27,900 | 因为RV16的话 |  |
| 255 | 00:10:27,900 --> 00:10:29,881 | 16位数据通路不可避免的会导致 |  |
| 256 | 00:10:29,881 --> 00:10:31,621 | 32位处理器的性能下降 |  |
| 257 | 00:10:31,621 --> 00:10:32,860 | 所以我们对一些指令 |  |
| 258 | 00:10:32,965 --> 00:10:34,360 | 也进行了一些优化操作 |  |
| 259 | 00:10:34,487 --> 00:10:37,287 | 首先对于跳转指令 |  |
| 260 | 00:10:37,287 --> 00:10:40,096 | 因为我们目前支持的地址空间是16位 |  |
| 261 | 00:10:40,096 --> 00:10:42,256 | 所以说依然它只需要 |  |
| 262 | 00:10:42,256 --> 00:10:44,275 | 用一个时钟周期来计算地址 |  |
| 263 | 00:10:44,415 --> 00:10:46,175 | 然后对于分支指令 |  |
| 264 | 00:10:46,176 --> 00:10:47,534 | 我们是可以通过 |  |
| 265 | 00:10:47,534 --> 00:10:49,168 | 先比较它的高16位数据 |  |
| 266 | 00:10:49,435 --> 00:10:53,080 | 如果高16位数据相等的情况下 |  |
| 267 | 00:10:53,081 --> 00:10:54,780 | 我们再比较它的低16位数据 |  |
| 268 | 00:10:54,781 --> 00:10:56,040 | 这样一来就可以 |  |
| 269 | 00:10:56,268 --> 00:10:59,740 | 在一些情况下节省一个时钟周期 |  |
| 270 | 00:10:59,768 --> 00:11:02,490 | 然后对于LB |  |
| 271 | 00:11:03,168 --> 00:11:06,115 | 无符号加载半字或者字节指令 |  |
| 272 | 00:11:06,116 --> 00:11:09,395 | 就可以先向它的第一个时钟周期 |  |
| 273 | 00:11:09,395 --> 00:11:13,315 | 先向高目标寄存器写入0 |  |
| 274 | 00:11:13,578 --> 00:11:17,185 | 它对于一个加载半字或者字节的来说 |  |
| 275 | 00:11:17,221 --> 00:11:18,300 | 它高位始终是0 |  |
| 276 | 00:11:18,300 --> 00:11:19,305 | 所以我们先写0 |  |
| 277 | 00:11:19,509 --> 00:11:21,125 | 然后再在第二个周期 |  |
| 278 | 00:11:21,165 --> 00:11:22,412 | 当漏的数据 |  |
| 279 | 00:11:22,412 --> 00:11:25,518 | 从存储器当中读取到了 |  |
| 280 | 00:11:25,518 --> 00:11:28,293 | 我们再将这个数据写入到 |  |
| 281 | 00:11:28,395 --> 00:11:30,955 | 它的低目标寄存器中 |  |
| 282 | 00:11:31,018 --> 00:11:34,346 | 这样一来就可以隐藏 |  |
| 283 | 00:11:34,346 --> 00:11:36,062 | 一个周期的访存的延迟 |  |
| 284 | 00:11:37,403 --> 00:11:40,637 | 我们对这个设计 |  |
| 285 | 00:11:40,637 --> 00:11:42,170 | 进行了一个RTL的实现 |  |
| 286 | 00:11:42,170 --> 00:11:44,793 | 我们对它的面积进行了一个评估 |  |
| 287 | 00:11:44,865 --> 00:11:46,446 | 首先是在FPGA上面 |  |
| 288 | 00:11:46,446 --> 00:11:47,990 | 它的一个面积表现 |  |
| 289 | 00:11:48,334 --> 00:11:53,037 | 红框框出的就是我们RV16的一个面积 |  |
| 290 | 00:11:53,340 --> 00:11:56,450 | 然后后面是几个参考的一个处理器 |  |
| 291 | 00:11:56,650 --> 00:11:57,970 | 可以看出其实我们的 |  |
| 292 | 00:11:58,034 --> 00:12:01,155 | 在硬件开销上是有明显的优势 |  |
| 293 | 00:12:01,155 --> 00:12:07,028 | 我们在定制的CMOS上面也进行了一个综合 |  |
| 294 | 00:12:08,446 --> 00:12:10,200 | 结果就表明RV16 |  |
| 295 | 00:12:10,200 --> 00:12:11,668 | 占用的CMOS的面积 |  |
| 296 | 00:12:12,043 --> 00:12:14,190 | 比传统的32位处理器 |  |
| 297 | 00:12:14,191 --> 00:12:15,765 | 大约是降低了 |  |
| 298 | 00:12:15,765 --> 00:12:17,170 | 在相似的配置下 |  |
| 299 | 00:12:17,456 --> 00:12:20,531 | 大约降低了28%到37% |  |
| 300 | 00:12:20,843 --> 00:12:25,728 | 然后我们将RV16 |  |
| 301 | 00:12:25,728 --> 00:12:28,356 | 和其他的一些RISC-V处理器 |  |
| 302 | 00:12:28,356 --> 00:12:30,200 | 进行了一个性能的评估 |  |
| 303 | 00:12:30,300 --> 00:12:31,740 | 我们评估分成两个部分 |  |
| 304 | 00:12:31,740 --> 00:12:32,980 | 一个部分是和 |  |
| 305 | 00:12:33,063 --> 00:12:35,703 | 传统的32位RISC-V处理器进行评估 |  |
| 306 | 00:12:35,956 --> 00:12:38,840 | 这个时候我们是使用两个SRAM作为 |  |
| 307 | 00:12:38,878 --> 00:12:42,340 | 指令存储器和数据存储器 |  |
| 308 | 00:12:42,615 --> 00:12:46,046 | 这个结果就显示 |  |
| 309 | 00:12:46,078 --> 00:12:47,576 | 在运行Dhrystone的时候 |  |
| 310 | 00:12:47,712 --> 00:12:49,895 | RV16在相似的配置下 |  |
| 311 | 00:12:49,987 --> 00:12:51,543 | RV16可以达到传统 |  |
| 312 | 00:12:51,543 --> 00:12:53,681 | RISC-V处理器的71% |  |
| 313 | 00:12:53,795 --> 00:12:55,706 | CoreMark可以达到69% |  |
| 314 | 00:12:56,262 --> 00:12:57,784 | 然后我们分析的就是 |  |
| 315 | 00:12:57,784 --> 00:13:00,170 | 从图中就可以看到 |  |
| 316 | 00:13:00,171 --> 00:13:02,930 | 在运行CoreMark的时候 |  |
| 317 | 00:13:03,215 --> 00:13:07,196 | RV16-32E/IC |  |
| 318 | 00:13:07,196 --> 00:13:08,768 | 这两个它的性能其实 |  |
| 319 | 00:13:09,003 --> 00:13:10,890 | 下降的比较明显 |  |
| 320 | 00:13:10,943 --> 00:13:12,690 | 其实这是因为 |  |
| 321 | 00:13:13,415 --> 00:13:17,165 | CoreMark里面包含更多的乘法操作 |  |
| 322 | 00:13:17,245 --> 00:13:20,209 | 就导致了这两个处理器核 |  |
| 323 | 00:13:20,209 --> 00:13:21,885 | 它是不支持硬件乘除法的 |  |
| 324 | 00:13:22,031 --> 00:13:23,059 | 这就导致的就是 |  |
| 325 | 00:13:23,421 --> 00:13:26,025 | 他们运行CoreMark程序的时候 |  |
| 326 | 00:13:26,245 --> 00:13:30,346 | 指令数量就大幅度上升了 |  |
| 327 | 00:13:30,346 --> 00:13:32,685 | 导致的它性能的下降 |  |
| 328 | 00:13:33,159 --> 00:13:37,262 | 我们统计了在运行Dhrystone和CoreMark的IPC |  |
| 329 | 00:13:37,878 --> 00:13:40,709 | RV16在不支持硬件乘除法时 |  |
| 330 | 00:13:40,709 --> 00:13:42,680 | 它的IPC大约是0.46 |  |
| 331 | 00:13:42,920 --> 00:13:45,120 | 支持之后降到了0.43 |  |
| 332 | 00:13:45,884 --> 00:13:47,650 | 在运行Dhrystone的时候 |  |
| 333 | 00:13:47,651 --> 00:13:49,471 | 这其实是因为 |  |
| 334 | 00:13:49,471 --> 00:13:50,830 | 乘法需要更多的周期 |  |
| 335 | 00:13:50,918 --> 00:13:54,074 | 然后CoreMark下降比较少是因为它 |  |
| 336 | 00:13:54,484 --> 00:13:57,162 | CoreMark里面它没有除法操作 |  |
| 337 | 00:13:57,330 --> 00:14:00,250 | 所以说它下降的就没有Dhrystone那么明显 |  |
| 338 | 00:14:01,715 --> 00:14:05,235 | 接下来就是另一个性能的评估方式 |  |
| 339 | 00:14:05,236 --> 00:14:07,535 | 我们是和Cortex-M0进行了对比 |  |
| 340 | 00:14:07,750 --> 00:14:10,535 | 首先是Dhrystone和CoreMark |  |
| 341 | 00:14:10,595 --> 00:14:16,015 | 其实我们重点关注从左到右 |  |
| 342 | 00:14:16,016 --> 00:14:20,034 | 第三个就是RV32 EMCF这个配置 |  |
| 343 | 00:14:20,040 --> 00:14:22,521 | 和Cortex-M0的性能进行对比 |  |
| 344 | 00:14:22,621 --> 00:14:27,723 | 因为它的配置和Cortex-M0是相近的 |  |
| 345 | 00:14:28,068 --> 00:14:29,610 | 在寄存器方面 |  |
| 346 | 00:14:29,611 --> 00:14:31,930 | 因为它是支持基于1扩展的 |  |
| 347 | 00:14:31,931 --> 00:14:34,284 | 1扩展支持16个通用寄存器 |  |
| 348 | 00:14:34,321 --> 00:14:36,178 | 所以说它和Cortex-M0的 |  |
| 349 | 00:14:36,178 --> 00:14:37,762 | 配置相比较接近 |  |
| 350 | 00:14:38,646 --> 00:14:41,709 | 整体上从图中可以看出 |  |
| 351 | 00:14:41,940 --> 00:14:45,180 | 它能够达到在运行最多的时候 |  |
| 352 | 00:14:45,181 --> 00:14:47,360 | 可以达到Cortex-M0的82% |  |
| 353 | 00:14:47,421 --> 00:14:51,365 | 运行CoreMark的时候能够达到86% |  |
| 354 | 00:14:54,118 --> 00:14:57,203 | 除了比较Dhrystone CoreMark |  |
| 355 | 00:14:57,203 --> 00:15:00,618 | 我们还比较了最近刚提出来 |  |
| 356 | 00:15:00,915 --> 00:15:03,634 | 就是19年刚提出的Embench |  |
| 357 | 00:15:03,634 --> 00:15:05,600 | 就是一个基准测试集 |  |
| 358 | 00:15:05,909 --> 00:15:09,096 | 从整体上而言 |  |
| 359 | 00:15:09,315 --> 00:15:12,703 | 就是支持EMC的情况下 |  |
| 360 | 00:15:13,006 --> 00:15:17,446 | RV16的性能能够达到Cortex-M0的76% |  |
| 361 | 00:15:19,671 --> 00:15:22,759 | 最后我们还对它 |  |
| 362 | 00:15:22,946 --> 00:15:26,896 | RV16的功耗和能耗进行了评估 |  |
| 363 | 00:15:28,028 --> 00:15:30,015 | 这幅图是RV16 |  |
| 364 | 00:15:30,015 --> 00:15:36,803 | 和传统处理器的一个功耗的评估 |  |
| 365 | 00:15:37,006 --> 00:15:39,393 | 可以看出主要因为 |  |
| 366 | 00:15:39,393 --> 00:15:41,353 | RV16在面积上的优势 |  |
| 367 | 00:15:41,353 --> 00:15:44,030 | 其实它在功耗上的 |  |
| 368 | 00:15:44,384 --> 00:15:46,103 | 不论是静态功耗 |  |
| 369 | 00:15:46,103 --> 00:15:47,330 | 还是动态功耗都有 |  |
| 370 | 00:15:47,415 --> 00:15:48,621 | 比较明显的优势 |  |
| 371 | 00:15:49,518 --> 00:15:51,803 | 从整体上来说 |  |
| 372 | 00:15:52,068 --> 00:15:57,125 | RV16它的功耗大约在 |  |
| 373 | 00:15:57,287 --> 00:16:00,796 | 传统32位处理器的60%左右 |  |
| 374 | 00:16:01,046 --> 00:16:03,131 | 然后能耗 |  |
| 375 | 00:16:03,390 --> 00:16:08,343 | 这幅图就是加上性能之后的考虑 |  |
| 376 | 00:16:08,612 --> 00:16:11,572 | 相对于Cortex-M0的一个能耗的表现 |  |
| 377 | 00:16:11,825 --> 00:16:13,300 | RV16的一个能耗的表现 |  |
| 378 | 00:16:13,540 --> 00:16:15,100 | 从整体上来说 |  |
| 379 | 00:16:15,140 --> 00:16:18,403 | 其实因为RV16 |  |
| 380 | 00:16:18,403 --> 00:16:19,503 | 在功耗上面表现 |  |
| 381 | 00:16:19,503 --> 00:16:20,960 | 优势还是比较明显的 |  |
| 382 | 00:16:20,961 --> 00:16:24,160 | 最后虽然它的性能相对较低一点 |  |
| 383 | 00:16:24,160 --> 00:16:26,728 | 它在能耗上面的表现 |  |
| 384 | 00:16:26,728 --> 00:16:29,984 | 也是相对于Cortex-M0要更加优秀一点 |  |
| 385 | 00:16:30,890 --> 00:16:34,765 | 我们还是关注EMC这个配置 |  |
| 386 | 00:16:34,765 --> 00:16:39,080 | 他大概是在可运行Dhrystone的时候 |  |
| 387 | 00:16:39,081 --> 00:16:41,520 | 大约是在Cortex-M0的69% |  |
| 388 | 00:16:41,800 --> 00:16:46,865 | 在运行CoreMark的时候大约是在68% |  |
| 389 | 00:16:47,575 --> 00:16:51,240 | 但是如果不支持硬件乘除法的话 |  |
| 390 | 00:16:51,241 --> 00:16:54,634 | 运行CoreMark它的能耗是要 |  |
| 391 | 00:16:54,881 --> 00:16:55,653 | 偏高一点的 |  |
| 392 | 00:16:55,653 --> 00:16:59,296 | 其实就是因为他们性能差距是比较大的 |  |
| 393 | 00:16:59,380 --> 00:17:03,200 | 最终导致了它的能耗相对更高一点 |  |
| 394 | 00:17:04,968 --> 00:17:08,906 | 最后我就做个简单的总结 |  |
| 395 | 00:17:09,112 --> 00:17:14,190 | 本次报告就主要就是提出了一个 |  |
| 396 | 00:17:14,190 --> 00:17:17,505 | 我们设计的一个RV16的微架构 |  |
| 397 | 00:17:18,540 --> 00:17:21,578 | 这个RV16是通过16位数据通路 |  |
| 398 | 00:17:21,578 --> 00:17:24,387 | 实现的一个32位RISC-V处理器 |  |
| 399 | 00:17:24,775 --> 00:17:26,840 | 然后它可配置的支持 |  |
| 400 | 00:17:26,840 --> 00:17:29,468 | RISC-V的E I M和C扩展 |  |
| 401 | 00:17:30,103 --> 00:17:32,540 | 在具体的微架构上面 |  |
| 402 | 00:17:32,541 --> 00:17:34,646 | 它支持两段的流水线 |  |
| 403 | 00:17:34,950 --> 00:17:39,925 | 通过对它的进行RTL实现 |  |
| 404 | 00:17:39,925 --> 00:17:41,728 | 最后评估它的面积大约是在 |  |
| 405 | 00:17:41,728 --> 00:17:45,203 | 传统32位处理器的63%到72% |  |
| 406 | 00:17:45,325 --> 00:17:49,120 | 性能能够达到传统处理器的71% |  |
| 407 | 00:17:49,120 --> 00:17:50,943 | 消耗的能耗和功耗 |  |
| 408 | 00:17:50,943 --> 00:17:53,762 | 大约是在传统 |  |
| 409 | 00:17:53,762 --> 00:17:56,712 | 32位处理器的82%到91% |  |
| 410 | 00:17:57,500 --> 00:17:58,262 | 好 谢谢大家 |  |