分支预测测试更新以及记分牌算法的实现与测试

童敢 黄家腾 赵琛然

摘要：本次实验报告分为两个部分，第一部分针对分支预测实现部分的新测试程序对处理器进行分析调试，经过仔细排查最终得到正确结果，第二部分依据上次报告中的设计实现了记分牌算法，并编写了简单的测试程序完成了验证，取得了预期效果，但因为结构复杂修改难度大，导致了分支指令不再受支持。

一、分支预测测试部分更新

在上次的实验中，我们组使用了 riscv-gnu-toolchain 生成的二进制可执行文件，进行处理后得到了可以写入处理器内存的 mem 文件，在指令内存中和数据内存中存放了相同的内容，根据工具链提供的程序入口地址，设定为处理器运行的第一个地址，但是并没有得到预期的结果，甚至整个程序的运行都并没能完全按照程序逻辑进行。

由于工具链生成的代码量比较大，手动排查极其困难，因此上次实验未能完成，在有了充分的时间之后，我们组重新分析了整个过程，讨论与测试验证结果如下：

（一）指令的生成

之前的实验使用工具链生成了汇编指令，再使用相关工具将其转化为二进制机器指令，这一过程中删去了汇编指令中一些不影响执行的部分；而在这次实验中，我们直接使用工具链生成可执行的 ELF 文件，再将其处理为 mem 文件，略去了生成汇编指令的过程。但是我们认为，不管是哪种方式，生成的指令都应该是可以按照预期执行的，因为所有涉及到的指令在我们的处理器中都已经“正确”实现。之前有同学汇报说很多指令在 riscv 指令集中并没有，但其实在 riscv 指令集手册中，有一章是汇编程序员手册，其中明确了很多 riscv 的汇编等价指令、寄存器使用的约定等。比如 jr ra 这一条指令，表示跳转到返回地址，ra 寄存器存放的就是返回地址，jr 表示寄存器跳转。但是在 riscv 指令集中只有 jal 和 jalr 指令，寄存器也只有 x0-x31，并没有 ra 寄存器，那是否意味着这条指令就没办法使用呢？并非如此，在约定中，ra 寄存器即为 x1 寄存器（图一），而 jr 可以使用链接到 x0 的 jalr 指令来实现，即等效于没有链接寄存器，最终 jr ra 等效的标准 riscv 指令为 jalr x0, 0(x1)（图二）。

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—

图一

pseudoinstruction	Base Instruction	Meaning
j offset	jal x0, offset	Jump
jal offset	jal x1, offset	Jump and link
jr rs	jalr x0, 0(rs)	Jump register
jalr rs	jalr x1, 0(rs)	Jump and link register
ret	jalr x0, 0(x1)	Return from subroutine

图二

（二）程序执行的入口

既然指令可以认为是没有问题的，那问题确实出在我们的处理器上吗？由于时间比较充裕，所以我们选择了一条条分析指令，不过将其中的 printf 函数、putchar 函数省略了，strcmp 函数和 strlen 函数也使用了自己的简单实现，经过分析排查发现程序入口是有问题的。

之前通过工具链分析 elf 文件得到的程序入口如下图所示：

```
There are 9 section headers, starting at offset 0x123c:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00010074 000074 000510 00  AX  0   0  4
  [ 2] .rodata           PROGBITS        00010584 000584 0009c8 00   A  0   0  4
  [ 3] .bss              NOBITS          00011000 001000 000408 00  WA  0   0  4
  [ 4] .comment          PROGBITS        00000000 000f4c 000011 01  MS  0   0  1
  [ 5] .riscv.attributes RISCV_ATTRIBUTE 00000000 000f5d 000021 00      0   0  1
  [ 6] .symtab           SYMTAB          00000000 000f80 0001b0 10      7  13  4
  [ 7] .strtab           STRTAB          00000000 001130 0000c0 00      0   0  1
  [ 8] .shstrtab         STRTAB          00000000 0011f0 000049 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  p (processor specific)
```

根据.text 部分的 offset 确定了入口地址为 0x00000074，当时对 Addr 一列中的 0x00010074 产生了疑虑，但是并没有深究。首先 0x00000074 并不是正确的程序入口地址，在上一次的报告中已经意识到了这个问题，这个地址只不过是代码段的起始地址而已，要找到真正的入口地址，还是需要对 main 函数的第一条指令位置进行确定，经过和汇编指令的比对，最终确定了“正确”的程序入口地址；但是程序的运行依然是有问题的，在从内存中读取字符串的字节时未能读取到正确的值，对指令进行拆解还原发现，访问字符串的地址其实是 0x00010xxx 这样的地址，而这串地址的首次使用是通过立即数形成的，因此得到结论，正确的.text 段就是应该从 0x00010074 开始，而不是从 0x00000074 开始。

对此我们组的解决方案是在 Addr 为 0x00000000-0x0000ffff 这一地址段填充为全 0，即截图中的 NULL，即后续的程序从 0x00010000 开始，而程序的入口地址为 0x00010xxx，重新进行仿真可以看到能够读取到字符串中的字节（此处没有截图）。

（三）非对齐访问和字符串存储格式

在上一步中虽然可以按照预期开始运行程序并访问字符串，但是访问到的值经常是不正确的，比如字符串 cabbie，用 ASCII 码表示为 16 进制数为 63 61 62 62 69 65 00 00，但是多次访问这一字符串时前四次均为 62，第五次为 00，最终得到字符串长度为 4，而正确结果为 6，这里是因为我们设计的处理器不支持非对齐访问导致的。

```
000003a0: 13 01 01 05 67 80 00 00 61 62 62 00 61 63 63 00    ....g...abb.acc.
000003b0: 63 61 62 62 69 65 00 00 61 63 63 65 6C 65 72 61    cabbie..accelera
000003c0: 74 6F 72 00 61 62 61 61 61 61 63 00 A8 03 01 00    tor.abaaaac.(...
```

在 Load 数据时，data_ram 模块会直接将送进来的地址向右移位 2 位，再访问内存返回数据，这意味着当需要访问后二位为 01、10、11 时，访问的依然是 00 处的数据，重新修改允许非对齐访问后，这一问题解决，但结果依然为 4。原因是虽然在内存中存放的顺序为 63 61 62 62 69 65 00 00，但 62 和 00 为低地

址，63 和 69 为高地址，所以访问到的顺序其实为 62 62 61 63 00 00 65 69，这是因为没有使用小端存储的方式，这里字符串存储方式和代码的存储方式不一致让我有点疑惑，但还是手动将字符串的存储进行了修改，最终可以得到正确的访问数据和执行结果。

（四）静态分配数据段和全局变量

虽然 strlen 和 strcmp 部分是正确了，但字符串匹配的结果依然是有问题的，我们发现在程序的开头，我们定义了几个全局变量：

```
18   static size_t table[UCHAR_MAX + 1];
19   static size_t len;
20   static char *findme;
```

而在得到的汇编指令中，经常通过这样的方式访问它们：

```
lui a5,%hi(len)
lw  a4,%lo(len)(a5)
```

但是在机器码中，经过仔细比对，发现这两条指令整合为了同一条指令 c011a703，即 11000000000100011010011100000011，翻译为汇编指令即为 lw x14,-1023(x3)，其中 x14 即为 a4 寄存器，相当于汇编指令的执行汇总 a5 充当了临时变量，最终将 len 的值 load 到了 a4 中，也就是说-1023(x3)就是 len 的地址。但是在整个程序中，没有对 x3 寄存器进行修改，默认为 0，所以会访问 0xfffffc01 这一地址，结果必然是没有的。查阅 riscv 汇编程序员手册发现，x3 为 gp 寄存器，即全局指针（Global pointer），这一寄存器的值肯定不应该为零，应该像 sp 寄存器一样，在初始化时，或者是程序加载到内存中时初始化为一个特定的值。网上资料显示，gp 寄存器的值与.bss 段有关，.bss 段通常是指用来存放程序中未初始化的或者初始值为 0 的全局变量的一块内存区域。bss 是 Block Started by Symbol 的缩写，属于静态内存分配。但是对 table、len、findme 这三个全局变量进行分析，并没有得到一个确切的 gp 寄存器的值，只能确定大概为 0x00011820 左右，具体调试工作耽误了很长时间没有进展，只能选择放弃。既然如此，不使用全局变量可能就不会使用 gp 寄存器，因此重新实现了该字符串匹配算法。由于我们组还是希望通过工具链完成实验，也算是得到了“官方的认可”,所以并没有直接采用汇编来实现,还是基于 c 语言实现了算法。

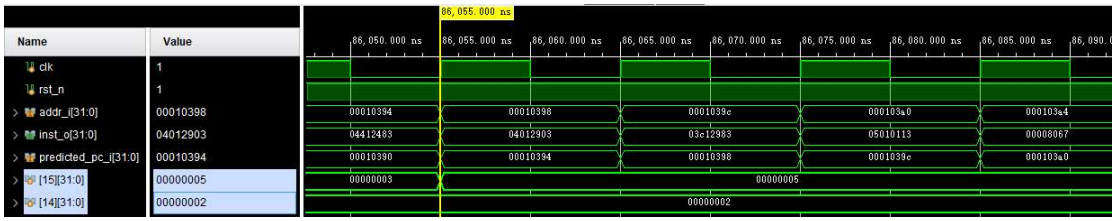
（五）一点致命的小错误和运行结果

在中期调试过程中，发现有一个大小为 255 循环并没有按照预期结束，陷入了死循环，排查发现在判断循环变量是否大于等于 ff 时，bgeu 指令始终判断为假，最终发现是在进行判断的比较时，有一个操作数使用的是输入进来的，另一个是输出出去的，中间在实现中进行了一些处理，仅仅是一个字母的差别而已，但是排查了很久。有意思的是，在之前的测试程序中，该条指令是可以正常执行的,也就是说,某一个操作数在处理前后可能是一样的,会侥幸得到正确的结果，但也有可能因为具体的运行程序而发生变化,因此对我们组的实现抱有完全没问题的幻想是不切实际的，正确运行的处理器还是需要大量测试程序的考验。

经过漫长的调试，微调不计其数，最后终于得到了正确的结果。最终的测试数据如下：

```
char  *find_strings[]  =  {"abb","acc","acc" };
char  *search_strings[]  =  {"cabbie","accelerator","abaaaac"};
```

依然为了便于分析，使用两个变量，一个用来存放字符串的数目，一个用来存放匹配上的字符串数目，容易分析发现两者值分别为 3 和 2，返回二者的和，结果为 5，通过 gcc 在 Ubuntu 下运行验证了程序的正确性。在我们组的处理器上运行结果如下：



可以看到，x5 和 x4 值分别为 3 和 2，返回的二者的和为 5，结果正确！

二、记分牌算法的实现与测试

（一）记分牌算法

引入记分牌算法是为了实现指令在处理器中的动态调度，减少流水线中因为数据相关而导致的 Stall。在处理器中存在三种相关：数据相关、结构相关、控制相关，其中结构相关可以通过增加功能单元的数目来解决，控制相关可以通过分支跳转预测、验证、回退来解决，而数据相关则是通过指令的动态调度来解决。记分牌算法在指令的发射阶段保证了结构相关和数据相关中 WAW 相关的消除，在读操作数阶段保证了 RAW 相关的消除，在最后的写回阶段保证了 WAR 相关的消除，最终在保证程序正确执行的基础上提高了程序的执行效率。

（二）流水线的修改

在上一次实验报告中，我们组基于《高级计算机体系结构》设计了新的流水线，其中我们将 ID 阶段拆分为两个阶段——发射（IS）阶段和操作数（RO）阶段，并对后续的流水线段进行相应的修改，但在实际实现过程中发现，直接在原有实现的处理器基础上修改比较困难，原因如下：

1. ID 阶段的拆分比较困难。在原有的实现上，ID 阶段用于解码，同时读取操作的操作数，如果存在数据转发也会在 ID 阶段进行处理，完成之后发送操作数和操作类型到 EX 阶段进行处理，更为难以控制的是分支相关的处理判断是 IF 和 ID 阶段相互配合实现的，如果不能在 ID 阶段确定操作数，而是在新的 RO 阶段确定操作数，则必然会产生一个时钟周期的指令槽，也就违背了分支预测的初衷。由于没能考虑清楚对策，所以经过小组讨论决定暂时放弃对分支指令的支持，以实现对记分牌算法的支持为主，虽然这么做是不对的，但也是在实现过程中做出的无奈决定。

2. 流水线的控制变复杂。在之前的实现中，我们使用 ctrl 模块控制流水段的 stall，处理由分支、跳转和 Load 指令（其后紧跟使用该数进行运算的指令）带来的流水线暂停。但是引入记分牌之后，其就充当了控制模块的角色。考虑到需要对整个流水线进行控制，应当为每个模块都增加一个控制输入，在记分牌确定需要暂停流水线的时候暂停流水线。需要暂停流水线的情况有这几种：①记分牌的指令状态表已满。可以将指令状态表视为指令队列，取指得到的指令会首先存放到指令状态表中，因此如果指令状态表已满，就不能再取指了，此时需要维持 PC 值不变；②指令队列中所有指令需要的功能单元都被占用。发射阶段需要

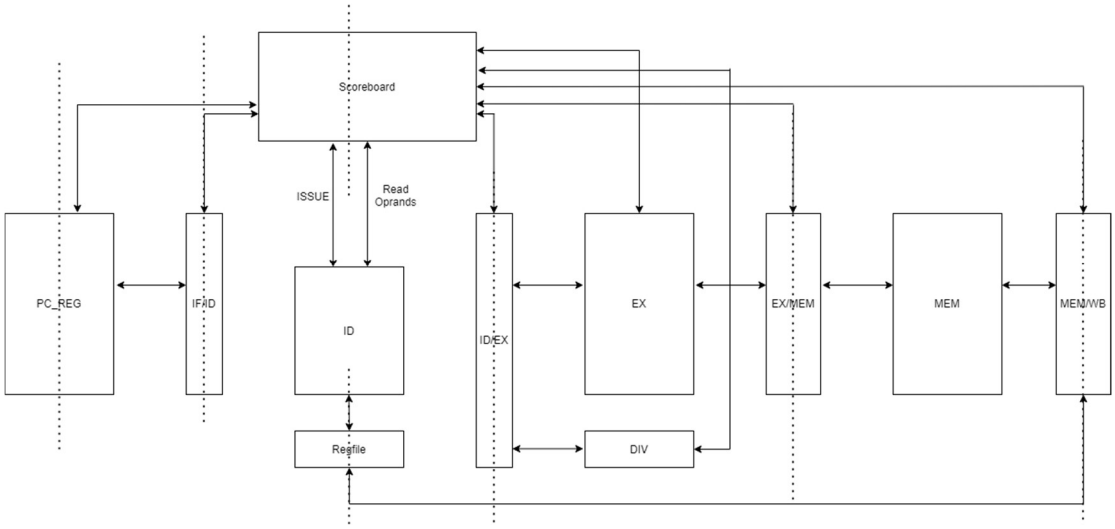
指令对应的功能单元可用，然后将指令发射，同时将功能单元状态进行修改，如果指令功能单元已满，则需要停止发射；③已发射的指令都在运行中或等待操作数。比如连续多条指令都需要一条运行中的除法的运行结果，则都会等待操作数，即使功能单元不被占用，也不能让指令进入执行阶段；④如果指令在写回阶段发现，要写回的寄存器是待执行功能单元的操作数之一并且另一个操作数正在等待其他指令的运行结果，则指令不能继续写回，需要等待相应的寄存器的数据被该待执行指令取到后才能写回寄存器和记分牌。总的来说控制的复杂度提高很多，对于此，我们也做了相应的简化。

（三）修改后的处理器结构

根据上一阶段分析的结果和实现中遇到的问题，我们保留了组合逻辑的 ID 阶段，但将记分牌实现为一个时序逻辑，IS 阶段和 RO 阶段都在记分牌中实现，IS 阶段配合原 ID 阶段的解码部分完成指令的解码和发射，RO 阶段配合原 ID 阶段的读操作数部分完成操作数的读取并将指令发送到执行阶段。

另外为了简化实现，EX 阶段只区分了两个功能单元——一般功能单元和除法单元。考虑如下：课本上将功能单元区分为整数运算、两个浮点乘法单元、一个浮点加法单元、一个浮点除法单元，但是我们实现的处理器中未实现对浮点运算的支持，因此只剩下除法单元。好在我们的除法实现为了试商法，需要 33 个时钟周期才能完成，因此可以作为单独的功能单元；另外在 Tomasolu 算法中有 Load/Store 队列，但我们的 Load/Store 在 MEM 阶段中实现，因此也没必要引入单独的 Load/Store 功能单元，也不需要将 Load/Store 集成到 EX 阶段，虽然这么做会使流水线的控制变复杂一些，但是相比较而言对前期实现的代码修改比较集中，所以最终确定了保持 MEM 阶段，将 EX 阶段区分为除法单元和其他运算的单元。

最后实现的处理器结构如下所示：



为简便起见，结构图省略了时钟信号和复位信号，其中 PC_REG、IF/ID、Scoreboard、ID/EX、EX/MEM、MEM/WB、Regfile 为时序逻辑，其余为组合逻辑，流水线段间的时序逻辑驱动，Scoreboard 本身包含两个流水段。可以看到，即便做了很多简化，控制逻辑复杂了很多，不过在模块方面做了尽可能少的变化，最大化利用了之前的成果。

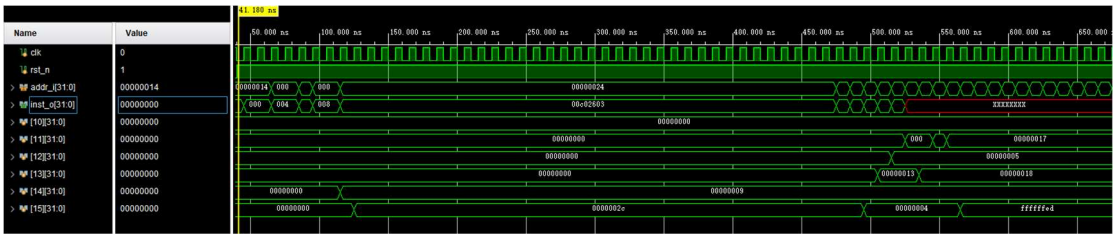
（四）实现与测试

关于记分牌每个阶段的工作主要参照了课本，主要因为未能舍弃的 MEM 段引入了有少许控制上的不同，在此不做赘述。我们组最后实现的记分牌中，指令状态表最多可以存放 4 条指令，为了方便后续发送到 EX 阶段，添加了操作类型和操作这两个变量，之前是由 ID 阶段发送到 EX 阶段的；因为所有除了除法运算之外的运算都为一个周期，Load/Store 指令需要到 MEM 阶段才能获得结果，因此需要额外一个周期，所以功能单元有新增了一项用于周期倒计时，功能单元状态表只有两项，一个是除法，一个是非除法，所以预期除法周期倒计时从 33 开始，Load/Store 从 2 开始，其余指令从 1 开始。此外，为了防止经常需要等待上一条指令写回再发射新的指令，我们保留了后续阶段的数据转发到记分牌，一旦得到运算结果就可以清空功能单元状态表项，这么做可以避免新设计中 RAW 带来的额外等待时间。

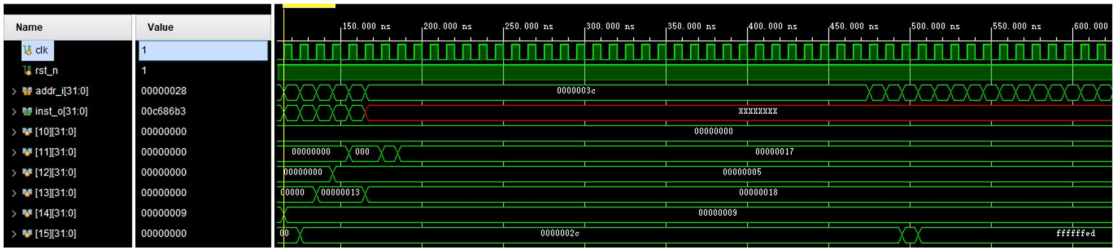
考虑下面一段简单的代码：

```
LW a4,0(zero)
LW a5,4(zero)
DIV a5,a5,a4
LW a3,8(zero)
LW a2,12(zero)
LW a1,16(zero)
ADD a3, a3,a2
SUB a1,a3,a1
ADD a1,a2,a1
SUB a5,a5,a1
```

由于 DIV 指令的耗时较长，后续指令（最后一条指令除外）都与该条指令不相关但却只能一直等到 DIV 指令执行完毕再执行，基于修改前的处理器，仿真结果如下：



直到最后一条指令更新完寄存器，共耗时 565ns。而引入记分牌算法之后，仿真结果如下：



共耗时 505ns，和未引入记分牌算法相比节约了 6 个时钟周期且运算结果相同、正确。虽然针对我们随意设计的测试代码，经过大量的调试测试之后终于跑出了貌似正确的结果，但是经历过之前实验中的问题，个人认为纯属侥幸。另外引入记分牌过程中抛弃了对分支、跳转指令的支持，相当于导致处理器功能变得不完整起来，至少引入记分牌前经过适当修改可以直接运行处理后的 ELF 程序，

总的来说是拣了芝麻丢了西瓜，但单就记分牌而言是基本实现了动态调度的功能。

三、课程总结

（一）实验总结

本次实验从指令的选择开始，经历基本流水线和分支预测，直到最后的记分牌，上学期高体课程中学习的知识得到了运用，虽然最后的记分牌算法引入后我们的作品成为了残次品，但是总体来说收获很多。在最开始的设计中，我们参考了很多前人的经验，即使不知所以然也会照做，但在老师一步步的引导下和同学们的汇报中认识到了很多自己忽略的东西，在我们对涉及到的交叉课程的探索中学到了很多，拓展了认识计算机科学学科的高度，也在不太熟练的团队配合中加深了和另外两位组员的革命友谊。

（二）组员工作介绍

1. 童敢：组长，负责统筹协调任务分工，确定总体框架和思路，负责了主要代码的编写、整合、调试和测试，撰写实验报告；

2. 黄家腾：组员，负责了基本流水线中 EX 阶段和除法单元的编写，编写了用于从汇编生成可直接使用的机器指令的 bash 脚本、用于将机器指令转化为十六进制文件的程序、针对数据 ram 存放格式的数据处理程序，在理解、使用 ELF 文件上提供了大量宝贵经验；

3. 赵琛然：组员，负责了基本流水线中内存模块的编写，测试程序的重构、本地测试和汇编代码、机器指令的标记，完成了处理器结构图的绘制，虽然上学期没有选修高体课程，但利用计组知识在分支预测部分提供了清晰明确的逻辑框架。

（三）课程感想

作为组长，我研究方向就是体系结构相关的，在假期中在某开源 32 位 RV 处理器的基础上完成了其到 64 位的修改工作，遇到了不少很有挑战性的工作。在本课程开始后，本以为有经验可以顺利完成实验，没想到自己从零开始写和站在巨人的肩膀上完全是两个概念（虽然所谓从零开始也是参考了很多前人的工作），实验过程中困难重重，团队配合也不如想象中顺利，很多时候都觉得“明明大概知道咋回事可就是不知道怎么做”，最后磕磕绊绊勉强完成了实验真的很不容易，相信对后续的学习科研会有很大帮助。最后感谢老师的辛勤指导和两位组员的大力支持！