2021 ICS 课程项目实验报告

杨馥冰 19307130304

August 2021

摘要

本次实验实现了一个 MIPS 架构的多周期多级流水线 CPU，并实现了文档要求的所有指令，提供了优化的缓存，设计了异常处理机制，并对流水线的性能进行了大量的改良。结果而言，我们在开发板上以 30MHz 的时钟频率，得到了 21.159 的流水线得分。

目录

1 流水线

本次实验采用了顺序双发的五级流水线结构。

1.1 取指 Fetch

由于流水线是顺序双发的机制，因此取指阶段需要读取两条指令。原始的 ireq 与 iresp 结构仅能传回一条指令，因此对其接口进行了改造，称为 flex_bus，其 req 保持与 ireq 相同，但 resp 现在可以传回指令 1 data_1 与指令 2 data_2，以及使能 valid_2，表示指令 data_2 是否有效。新的接口可以在一个时间周期中返回两条指令，这提供了顺序双发的实现基础。

当接收到两条指令时，就分别对两条指令进行简单的解码，否则仅对第一条指令 data_1 进行解码，另一条指令输出 NOP。之所以在取指阶段进行简单的解码，主要是为了尽早获得 op 信息，从而可以在取指阶段即判断指令是否处于延迟槽中，以及判断指令是否会引发异常（尤其是 SYSCALL 与 ERET），从而可以避免额外的指令读取延迟。

另一个难点在于对跳转信息的管理，由于取指阶段存在指令读取延迟，最坏情况下，可能会等到后续流水线清空。为了保证指令跳转信息不会丢失，需要用额外的流水线寄存器分别保存译码阶段（对应跳转指令）和写回阶段（对应异常处理）的跳转信息，并且保证写回阶段（指令更早）的跳转信息高于译码阶段（指令更晚）的跳转信息。这样，后续流水线阶段产生一次跳转信息后，即可向下一级流水线正常传递信息。

1.2 译码 Decode

译码阶段会接受两条从取指阶段得到的指令信息，读取寄存器的值，并处理后续阶段的转发信息。每条指令信息单独处理（依然需要考虑加载使用冒险），唯一的交互在于，指令 1 data_1 的写寄存器与指令 2 data_2 的某个读寄存器相同，即存在读写冲突的情况，此时需要分两个时钟周期分别输出两条指令的信息，从而增加一个额外的延时。

一个繁琐之处是在于对通用寄存器，HILO 寄存器，以及 Cp0 寄存器的读取，虽然可以通过将所有寄存器用统一的编号进行管理，但这样会需要所有寄存器进行重编号，而且扩展 CPU 的指令集时可能需要重构编号方案。因此我们选择对这三类寄存器用不同的代码单独进行处理，从而尽可能的保持了 CPU 的原有结构，代价是这部分的代码量变成了三倍。

1.3 执行 Execute

执行阶段同样只需要单独处理两条指令。单周期的指令通常比较简单，按照语义进行实现即可。

朴素实现的单周期乘除法器会造成大量延时，因此使用了文档 2b 提供了实现，其中乘法器为两周期，除法器为 32 周期。为了处理有符号数的乘除法器，额外加入了两个模块用来将有符号数转换成无符号数，以及将无符号数还原成有符号数。

1.4 存储 Memory

存储阶段需要协调两个指令对存储器的读写，因此需要一个额外的仲裁器。当指令 1 需要读写时，选择指令 1 的接口，否则选择指令 2 的接口，此外，由于存储阶段会对存储器产生影响，因此需要额外判断一次，当指令 1 出现异常时，禁止指令 2 的读写。

其中，作为优化，store 指令可以在收到 addr_ok 后即结束该阶段。具体而言，load 指令需要得到具体的数值，因此必须要等到 data_ok 才能结束存储阶段，而 store 指令可以让缓存与流水线独立地进行处理，在缓存位于 store 阶段的多个周期中，流水线可以独立的流过多条与存储器无关的指令且不产生影响，如果在缓存的 store 阶段尚未结束时，流水线中流入一条存储器操作指令，那它也有由于缓存忙碌而进入等待状态，不会产生影响。因此这个改进是无损的。

1.5 写回 Write

写回阶段会产生大量的修改，如果指令 1 出现了异常，则需要处理其异常信息，跳过指令 2 的处理，否则则需要正常处理指令 1 的修改操作，并处理指令 2 的异常信息与修改操作。如此操作即可一周期处理两条指令。

为了方便进行 debug，我们也设计了一个顺序单发射写回阶段代码，除了上述操作外，第一个周期将 debug 信号接到指令 1 上，并阻塞写回阶段，修改计数器，第二个周期将 debug 信号接到指令 2 上，并解除写回阶段的阻塞。这样既不影响前面四个流水线阶段，也几乎不影响写回阶段，同时可以用 test1 4 进行 debug。

1.6 流水线阻塞机制

原始的流水线阻塞机制（如 Y86）通过 STALL 与 BUBBLE 来统一控制每个流水线的状态，且有严格的优先级顺序，当流水线逻辑变复杂时，if-else 语句链会变得极长，从而导致代码冗杂，并产生大量的延时。

为了改善这个问题，我们使用如文档 3b 思考题中提及的握手信号 {ready, valid} 实现阻塞机制。流水线阶段的 valid = 1 表示已经执行完成，输出信号有效；ready = 1 表示已经准备好接受输入信息，下一个时钟周期时将会根据输入信息更新流水线寄存器状态。表1.6详细阐述了两种信号的组合结果。

valid	ready	effect
0	0	流水线不输入也不输出，相当于 STALL
0	1	流水线不输出但输入，相当于 BUBBLE
1	0	流水线不输入但输出，相当于 STALL
1	1	流水线输出也输入，相当于无阻塞

其中，valid = 1 且 ready = 0 的组合，发生在下一阶段尚未 ready，握手失败的情况。此时为了避免当前阶段的信息丢失，当前阶段的 ready 也必须为 0。这会导致一条从 Write 传回 Fetch 阶段的逻辑指令链，但其他的大部分阻塞机制都在各个流水线内部完成，相比原始的设计已经有了足够的改善。

1.7 异常处理 Exception

流水线的异常处理机制在文档的基础上，还参考了 refcpu 的实现。

我们将 refcpu 中异常处理机制的代码进行拆分，其中 interrupt 在 Fetch 阶段判断（越早发现异常就可以越早进行处理），其他异常在各自产生的阶段判断，异常处理代码在 Write 阶段实现。

当靠后的流水线阶段发生异常时，靠前的流水线阶段（对应之后读取的指令）的值应当被舍弃。我们选择在流水线阻塞机制中增加一条额外独立的逻辑指令链，用来从后向前传递异常信息，当某流水线阶段发现更靠后的阶段发生异常时，就将流水线中信息舍弃。这可以避免在一些要舍弃的指令消耗过多时间，如多周期除法上。

2 缓存

我们使用了 4 路组相连的缓存，index 长度为 4，缓存替换逻辑为 FIFO 先进先出。为了方便测试，缓存大小的参数均由 cache.svh 中的参数 cache_set_len 与 cache_line_len 控制。

文档代码中默认使用的 xpm_memory_spram 仅包含一个输入输出端口，与顺序双发要求进行两次读入的需求不符。作为替代，我们使用 xpm_memory_dpdistram 进行改进，该元件具有一个写端口与两个读端口，并相对于直接用代码进行实现，可以节省大量的开发板空间。

此外，我们发现代码中有大量对连续指令的读取操作，常理上也确实如此，代码的读取具有明显的空间局部性，如果缓存没有命中，就需要进行等待。朴素的实现中，需要等待缓存全部加载完毕后才返回指令。一个显而易见的改进是在缓存加载到指令对应地址时就返回，随后继续加载。自然而然的，可以想到另一个改进思路，即是在返回待加载的指令后，下一条待加载的指令通常也是连续的下一条指令，也是缓存加载过程中得到的下一条指令，因此也可以直接返回指令值，而不需要等待缓存加载完毕。这样即可大大提升取指阶段的效率。

3 test5 流水线得分

ID	test program	mycpu	gs132	T_{gs132}/T_{mycpu}
1	bitcount	000b144e	13CF7FA	28.60904786
2	bubble_sort	00477642	7BDD47E	27.7326633
3	coremark	00bf9a38	10CE6772	22.45485287
4	crc32	00881627	AA1AA5C	19.999521
5	dhrystone	0019d88d	1FC00D8	19.65508133
6	quick_sort	0057c6e2	719615A	20.70454265
7	select_sort	0055afc8	6E0009A	20.53998426
8	sha	004f7c6a	74B8B20	23.49525243
9	stream_copy	000b5706	853B00	11.74878561
10	stringsearch	003af46c	50A1BCC	21.8830218
性能分		21.159		

4 项目分工

本次实验项目中,杨馥冰负责顺序双发流水线设计与优化,缓存 xpm_memory_dpdistram 的使用，以及缓存连续取指的优化。