LC-3 流水线调试报告

在 LC-3 流水线的整个设计实现过程中，调试占用了主要时间。由于我们对 CPU 设计和时序设计没有经验，在很多小问题上花费了大量时间。

一、流水段的调试

在最初的设计中，流水线分为取指与译码、执行、访存、写回四个流水段。由于寄存器使用了消除相关的“多版本”设计，不再需要读寄存器的流水段。由于取指是从指令内存中取出，译码时间相对从内存中取指令的时间要小很多，因此译码没有单独占用一个流水段，而是与取指过程合并，这样流水段寄存器存放的就是译码后的各部件动作信号了。

也正是因为指令不随流水线流水，考虑到暂停流水线，在调试过程中很难推断当前部件执行的是哪条指令，需要根据其他信号（如暂停流水信号）来推断当前的执行进度，并不直观；在调试后期才将指令的 opcode（四位操作码）从流水段寄存器中取出并在仿真波形中查看，使得每个部件在执行什么操作一目了然，便于发现问题所在。经验是：对于流水线这种并行的电路，调试时要输出每个并行部分的状态。

调试中发现的流水段错误主要有：

1. 接口错误（本来该从第 2 流水段引出针脚，结果接到了第 1 流水段）；
2. 同步异步错误（本来应该异步赋值，结果同步赋值使得读到的值总是落后一个周期）；
3. 信号电平错误（使能信号的高低电平弄反了）；

这些错误的主要原因是设计中模块不清晰，大量的组合工作放在了顶层文件 cpu.v，不同流水段的信息混杂在一起，容易出错。事实上，各流水段应当“各司其职”，从上一流水段的状态寄存器中读取信息，向当前流水段的状态寄存器中存储信息，并发出全局的暂停流水线、排空流水线的信号。然而，目前的顶层文件“为了方便起见”，“越俎代庖”地将流水段寄存器解包后送给流水段，再将流水段的计算结果打包到下一流水段寄存器，并利用流水段的结果生成一些全局控制信号。从逻辑上来说，这里并没有错误；但从结构上来说，这些打包解包、控制信号生成的工作应该在流水段内部完成。感到遗憾的是我们没有充分发挥模块化的力量。

二、消除相关（hazard）

根据我们的流水线和寄存器设计，在两种情况下会发生“写后读”相关：

1. 从内存取数到寄存器后，读寄存器：LD R0, PCoffset; ADD R1, R1, R0. 由于 R0 的结果要在访存阶段才能确定，而此时下一条指令已经要使用 R0 的值了，这从时序上就是无法用 forwarding 解决的相关。必须在两条指令间插入一个“气泡”，即空指令，使第二条指令错后一个周期执行。
2. 从内存取数到寄存器后，再把这个寄存器写入另一内存单元：LD R0, PCoffset; ST R0, PCoffset. 这从逻辑上本来没有相关（都是在访存阶段），但由于我们的流水线中从寄存器取数是在 ALU（执行）阶段就完成的，因此需要按照上面一种情况暂停流水线（bubble）。

由于存在需要访存两次的 LDI 和 STI 指令，LDI 和 STI 指令也需要暂停流水线（indirect），与 bubble 不同的是 indirect 的内存阶段流水寄存器保留原值，而 bubble 需要清空此流水段寄存器（不然就把从内存读出的错误值写入寄存器了）。注意到这里的暂停（indirect）后可能正好又碰上了“写后读”相关，使得问题比较复杂，稍有不慎时序就会出错。

现在反思，不论是“写后读”相关还是间接访存，都可以在译码阶段完全确定是否需要 bubble 或 indirect。如果做新的设计，将把相关逻辑放到 decoder，在 decoder 中增加一个计

数器判断 bubble、indirect 的当前状态，在需要 bubble 的周期重新发出执行阶段（ALU）控制信号，在间接内存访问（indirect）的第二周期发出特殊信号（将读数据缓冲中的值作为内存地址）。这样不再需要在流水过程中临时暂停，整个流水过程会显得规整得多。

三、时钟边沿的毛刺

1. 数据源在时钟边沿不稳定

我们在调试过程中，发现寄存器的地址（addr）、写使能（we）、数据（data）在仿真波形中都是正确的，然而在时钟边沿触发的寄存器却写入了“风马牛不相及”的数据。经过放大波形图，我们发现时钟边沿并非绝对的“边沿”，当 always 语句内部有较复杂的判断（if、else 嵌套）时，内部赋值语句的执行时机并非整齐划一，而是加上了组合逻辑判断的（不固定）时间，always 仅仅保证了组合逻辑的触发时间是时钟边沿。如果赋值的数据源（如 data）在时钟上升沿由另一组语句控制也发生了改变，则不能保证写入的数据是正确的。

为了避免时钟边沿的数据不稳定，我们约定：

存储部件（包括内存、寄存器）在时钟下降沿（同步）写入数据；

部件内部计数器（如暂停流水线部件内部的“上周期是否暂停”状态寄存器）在时钟下降沿触发；

其他需要边沿触发的部件在时钟上升沿触发。

这样，根据我们的设计，被写入的数据存储单元与数据源不会在同一时钟边沿被触发，保证了所保存数据的稳定性。

2. 控制信号的毛刺

在暂停流水线的设计中，输出信号是当前周期是否要暂停流水。简化的判断准则是：在时钟边沿判断，如果上一周期没有暂停流水，且当前周期符合暂停流水的条件，则暂停流水。但这个信号在时钟边沿到来之前必须产生且稳定，使得相关部件的使能信号在时钟上升沿附近稳定。为此，模仿 latch 的原理，我们采用了一个输出 wire 和一个状态 reg，wire 的值由 reg 和外部条件经组合逻辑生成；时钟下降沿 reg 更新为当前 wire 的值，随之 wire 的值就可能发生改变。这就保证了时钟上升沿前后 wire 的值是稳定的。

四、与 Verilog 有关的调试技巧

1. 防止变量被优化

在 Verilog 中有些中间变量被优化掉了，因此无法在波形文件中查看，这使我们很头疼。后来发现对有扇出的 wire 型变量加入指示注释：

```
wire flush /* synthesis keep */;
```

或者对不重复的 reg 型变量加入指示注释：

```
reg buffer /* synthesis noprune */;
```

即可使 Verilog 分析器保留这些变量，便于调试查看。上述做法仅对有扇出（fan-out）的变量有效。

事实上，将中间变量逐层传递到顶层文件作为 output，就一定不会被优化掉了，但这样做就破坏了模块的封装性，因此我们没有采用。

2. 选择合适的内存类型

开始设计时，我们选用的是 lpm_ram_dq 作为 RAM，lpm_rom 作为 ROM，每次编译都要两分钟左右，而且有不少 Warning。Quartus 帮助文件中说，上述 Megafunction 仅仅是提供“向后兼容”的，在 Cyclone 系列中应使用 altsyncram 组件。这个组件不仅可设置的选项更多，更重要的是它是 Cyclone FPGA 内置支持的，不需要用 latch 来模拟，这就提高了编译

速度（半分钟左右就编译完成），可能在烧写到 FPGA 后还能提高运行速度。

五、实验平台上的调试

这个过程最让人无语了，浪费了一下午的时间，第二天却发现 reset 信号的电平与实验平台反了。给我们的教训是，调试时要控制变量。最初盲目修改时错误程序与正确程序有多处不同：对 ep1c12.v 进行了修改；没有使用 bdf 图形文件；没有打开外部内存；CPU 顶层文件的接口不同。由于仿真通过了，当程序计数器（pc）始终不动时，都怀疑问题是实验平台没有给时钟（CLK），而没有想到 reset 信号的问题。因此在 ep1c12.v 里做了很多“无用功”。

第二天采用控制变量法，利用实验平台上的 LED 灯，发现 CPU 外面（ep1c12.v）有 CLK，里面（cpu.v）没有 CLK，再检查 CLK 的接线完全正常。于是在老师的指导下，看看 cpu.v 的其他管脚信号是否有变化。当测试到 reset 时立刻发现了问题。前后不超过半个小时。

六、单元测试

由于本次 LC-3 实验时间紧，刚测试完几条互不相关的简单指令就使用了老师提供的测试程序，这个程序有一百多条指令，每次调试要费好几步：

1. 找到出错的位置（我们的程序与正确程序相比，第一处不同出现在哪里）
2. 定位到出错的指令（开始时没有输出每个流水段正在执行哪条指令，只好凭其他信号和对程序的印象来推测是前面载入的几条指令中的哪一条）；
3. 根据程序计数器（pc）定位到错误指令所在汇编语言文件中的汇编语句；
4. 如果是首次调试此段汇编，还要到 asm 文件中阅读此段汇编语言的用意；
5. 比对正确程序的仿真波形图与当前程序的仿真波形图，对错误的写入内存值进行“追本溯源”，看看是哪个寄存器的读写错误导致的此问题，这个寄存器的错误值又是从哪里来的，亦即第一处发生错误的是哪条指令；
6. 在汇编文件中找到引起最初错误的指令含义，并与前面的指令进行比较，推断是哪一方面的问题：时序、译码、接线、相关（hazard）；
7. 确定问题来源后，对相关模块的内部变量添加/* synthesis keep */，重新编译，在波形文件中添加要查看的针脚，仿真；
8. 查看哪个针脚的输出有异常，并查看相关模块的代码，看与“我们认为的”代码的动作方式是否相同；
9. 对于多数错误，已经能定位到具体的错误代码，采取适当措施对错误进行修正；
10. 再次编译仿真，看原错误是否消除。

下次再做相关实验时，我认为应首先编写代码进行“单元测试”，即对每个模块的行为进行单独测试，再整合起来做整体测试，可以大大提高问题的定位速度。

以 decoder（译码器）模块为例，这是 LC-3 CPU 中最大的一个模块，就是一个组合逻辑电路（大 case 语句），根据指令生成各种部件的控制信号。在调试过程中发现了多处错误。事实上应该把 decoder 作为顶层文件，设计一个单独的波形文件，每种类型的指令各有一条，输出所有控制信号。Decoder 不涉及时序问题，因而通过查看控制信号的输出很容易发现译码中的错误。这比在整合调试时发现错误，定位到这条指令，再定位到 decoder 模块要省时省力得多。

在相关（hazard）的测试中，由于测试程序中指令数量多、相同 opcode 的指令多次重复（例如 ST、LD、STI、LDI 反复出现），不仔细思考就难以定位出问题的阶段执行到哪条指令了，它的数据是从哪里来的，理论值应该是多少。事实上调试的开始阶段应该专门挑几组有代表性的指令进行测试，每组指令代表一种引发相关的情况，组与组之间用若干空指令（如

ADD R0, R0, R0）填充，使得组与组之间分界清晰，集中精力于处理相关的机制而排除周围无关指令波形的干扰。如果当时这样做了，可能就不会白白浪费很多时间在定位指令和分析各寄存器的理论值上。

七、缺陷或尚未实现的功能

1. 没有实现外部中断；
2. 写回阶段在调试时消失了，四级流水事实上变成了三级流水，导致寄存器原来是 d1、d2、df，现在只剩下 d1、d2 了；
3. 没有实现 VFn 和 VFp 的判断；