1 仿真调试说明

在学习并尝试本章节前，你需要具有以下知识：

(1) 仿真工具的使用，比如 Vivado 的 Xsim。

(2) Verilog 的基本语法。

通过本章节的学习，你将获得：

(1) 各类仿真错误排查的方法。

(2) CPU 逻辑出错的调试指导。

(3) Verilog 运算符的优先级。

看完本章节，请询问自己是否知道以下知识点，如果不知道，请找队友讨论：

(1) 何为多驱动的信号？

(2) 阻塞赋值和非阻塞赋值的区别？其使用情况分别是？

(3) 信号为“Z”和“X”的区别。

(4) Verilog 中“+”与“&”的优先级。

(5) Verilog 中，两个 reg 型信号写在一个 always 块里和分开写在两个 always 块里，有何区别？

(6) 请反思自己编写 Verilog 中是 C 语言编程的思想，还是画电路框图的思想。

本文档在介绍的仿真调试手段是基于 Vivado 的 Xsim 仿真器的经验总结。对于其他仿真器，主体内容也是通用的，但在部分细节上有所不同，需要大家具体问题具体分析。

1.1 调试指导思想概述

调试是指在我们设计的一个系统在执行功能出现了错误时，定位出错误的原因。比如我们设计了一个 CPU，在运行一个测试程序时发现结果不对，这时就需要进行调试，以便后续进行纠正。可以看到全局上的调试原理是从结果推原因，难点就是定位错误的源头。

本文档编写时采用的的调试思路是：时间上先定错，空间上再定错。一个设计在执行功能出现错误时，往往是在一个大片的时间段内该设计的电路的执行都不符合预期。“时间上先定错，空间上再定错”具体解释如下：

(1) 时间上先定错：在出错的大片时间段里，定位出源头部分，源头部分是一个较小的时间段。

(2) 空间上再定错：在源头时间段里，查看设计电路的控制部分和数据通路，定位是哪个信号带来的错误，或者是哪几个信号的组合带来的错误，或者是设计上哪里有疏忽带来的错误。

比如一个设计的 CPU 在执行测试程序出错了，这个程序是分很多指令的，这些指令是在时间上顺序执行的，我们首先需要找出第一个错误的的指令（也就是时间上定位错误），随后在 CPU 的数据通路和控制信号里定位该指令错误的原因（也就是空间上定位错误）。

相对于空间上的定错，时间上的定错更加困难。特别是对 CPU 调试而言，更是如此，往往 80%的精力都用于时间上定错了。

时间上定错和空间上定错，是一种针对设计的整体调试的指导思想。但当我们仿真发现一个错误时，往往需要先去辨别错误时什么类型，并安装一定的方法追踪错误原因。

1.2 仿真出错分类

仿真出错情况按照波形直接观察结果可分为两类：

(1) 波形出错：从波形图里直接观察，而不需要分析电路设计的功能，就能判断的错，比如波形中信号为“X”。

(2) 逻辑出错：波形直接观察很正常，但其电路执行结果不符合预期，属于逻辑出错，比如加法器运行结果不对。

“波形出错”为浅层次的出错，都是很容易查找到原因的。“逻辑出错”则是深层次的出错，是真正调试难点，其具体内容也是包罗万象。

1.3 出错分类一：波形出错

波形出错，细分又可归为以下几类：

(1) 发现信号为“Z”。

(2) 发现信号为“X”。

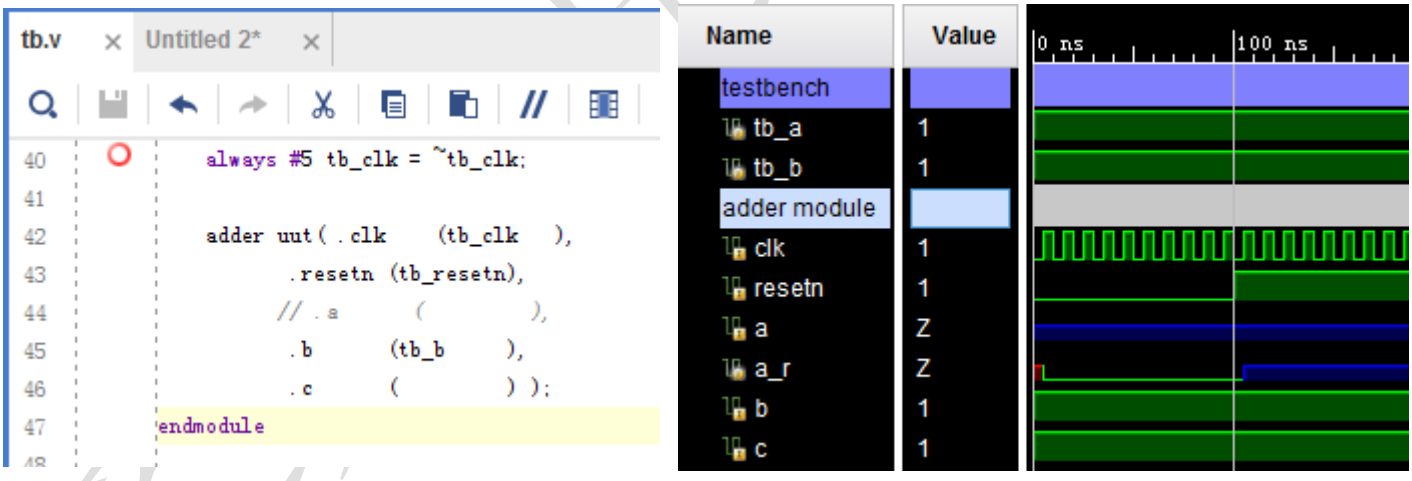
(3) 波形停止。

(4) 越沿采样：上升沿采样到被采样数据在上升沿后的值。

(5) 其他，波形怪异：仿真波形图显示怪异，与设计的电路功能无关的错误。

1.3.1 信号为“Z”

“Z”表示高阻，比如电路断路了就是显示为高阻，往往是模块调用的信号未连接导致的信号悬空，如下图：



(a) 源码　　(b) 波形

图 1-1 信号为“Z”示例

上图示例中有一下几点比较重要：

(1) 信号值为“Z”，为模块调用是信号未连接，未连接包括两种：显式的未连接，如图 1-1(a)中的.c()；隐式的未连接，如图 1-1(a)中模块 adder 调用时，a 端口即未连接。“显式的未连接”一般是人为故意设置的，只针对 output 类接口；“隐式的未连接”则是疏忽，属于代码不规范，往往也是导致信号值为“Z”的主要原因。

(2) adder 模块里，a 端口未连接，导致 a 为“Z”，但 c 端口也未连接，c 却是固定值。这是因而 a 端口是 input，c 端口是 output。output 类接口未连接是母模块里不使用该信号，可能是人为故意设置的。所有的 input 类接口被调用时不允许悬空。

(3) adder 模块里 a 信号从 0 时刻开始就是“Z”，而 a_r 信号确实在 100ns 左右才变成“Z”的。这是因而 a 信号为端口，被调用时就未链接，故从 0 时刻就为“Z”，但 a_r 信号是内部寄存器，从 100ns 时刻才使用 a 信号参与赋值，所以也变成了“Z”。

针对以上以上几点，我们有一下几点建议：

(1) RTL 编写时注意代码规范，特别是模块调用时，按接口顺序一一对应。

(2) 所有 input 类接口被调用时不允许悬空。

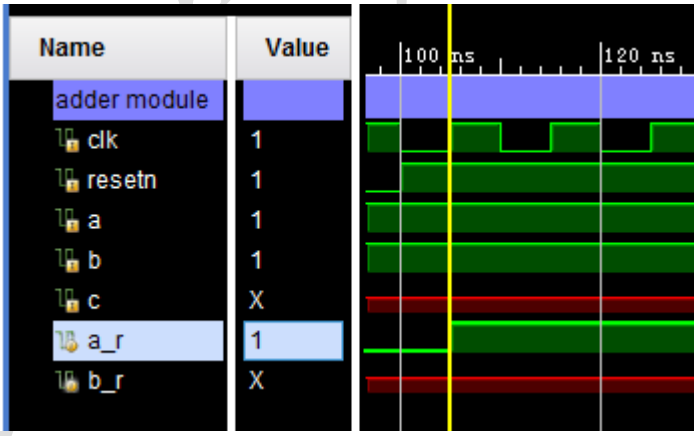
(3) 一旦发现一个信号为“Z”，向前追踪产生该信号的因子信号，看是哪个为“Z”，一直追踪下去直到追踪到该模块里的 input 接口，随后进行修正。

(4) 有可能“Z”只出现在向量信号里的某几位上，也是一样的追踪，有可能调用时某个接口存在宽度不匹配也会带来该接口上某些位为“Z”。

1.3.2 信号为“X”

“X”表示不定值，往往是 RTL 里信号未赋值导致的，如下图：

```
43      /*
44      always @(posedge clk)
45      begin
46          if(!resetn)
47          begin
48              b_r <= 1'b0;
49          end
50          else
51          begin
52              b_r <= b_r;
53          end
54      end*/
55      assign c = a_r + b_r;
```



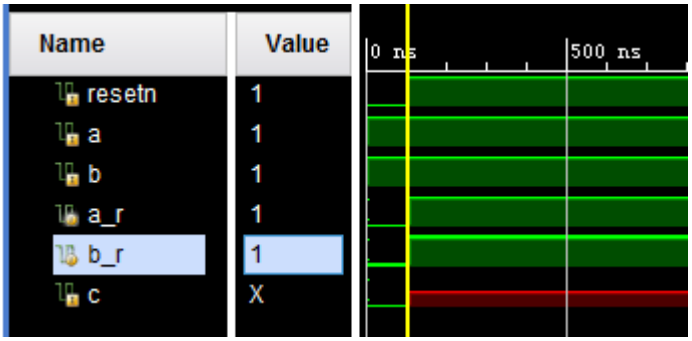
(a) 源码　　(b) 波形

图 1-2 信号为“X”示例

在上图中，由于 b_r 信号声明后始终未赋值，导致其值为“X”，后续 c 信号由于使用了 b_r 信号，导致其值也为“X”。

另外，Vivado2017.1 对于多驱动（2 个及 2 个以上电路单元驱动同一信号），仿真时也会产生“X”，如图 1-3。这种情况下追寻信号为“X”的原因可能不太好追，可以尝试先进行综合，观察下 Critial warning，此时会报出多驱动的警告，如图 1-4。

```
assign c = a_r&b_r;
assign c = a_r + b_r;
```



(c) 源码　　(d) 波形

图 1-3 Vivado2017.1 中多驱动引发“X”

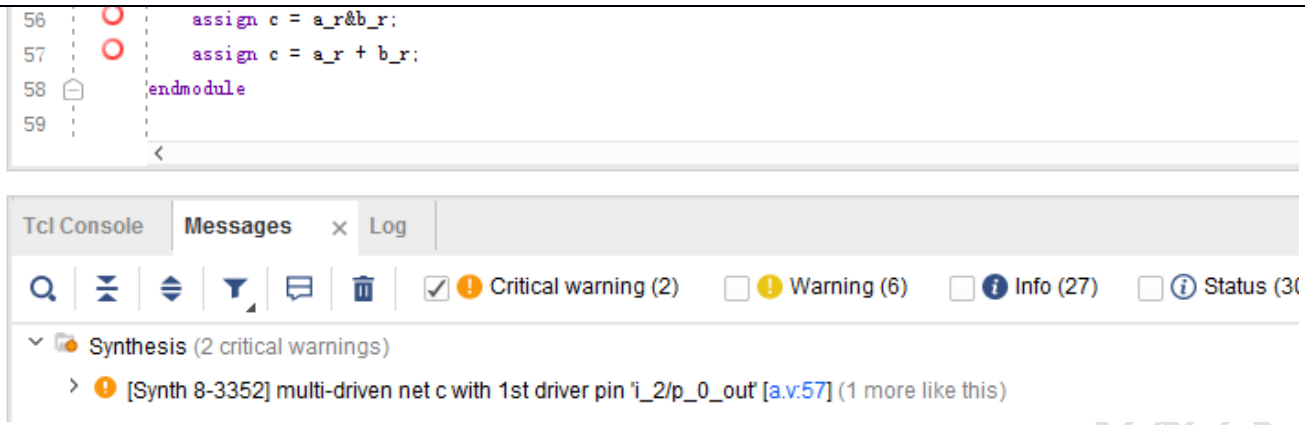


图 1-4 Vivado 中多驱动报出 Critical warning

针对信号为“X”情况，我们有以下几点建议：

(1) 一旦发现仿真错误来自某个信号为“X”，则向前追踪产生该信号的因子信号，看是哪个为“X”，一直追踪下去直到追踪到某个信号未赋值，随后修正。

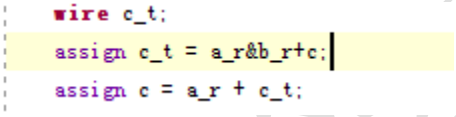
(2) 如果因子信号都没有为 X 的，则很可能是多驱动导致的，则综合排查 Error 和 Critical warning。

(3) 寄存器型信号如果没有复位值，在复位阶段其值可能也为“X”，但可能这并不会带来错误。

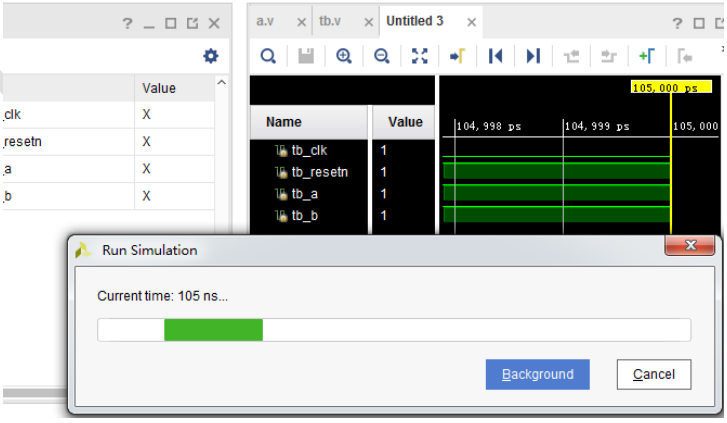
(4) “X”和 1 进行或运算结果为 1，“X”和 0 进行或运行结果为 0。

1.3.3 波形停止

波形停止是指仿真停止某一时刻，再也无法前进分毫，而仿真却显示不停地在运行，如下图：



(e) 源码



(f) 波形

波形停止基本都是由于“组合环路”导致的，所谓组合环路就是信号 A 的组合逻辑表达式中某个产生因子为 B，而 B 的组合逻辑表达式中又用到了信号 A，如上图源码 c_t 用到了 c，而 c 又用到 c_t。仿真器是在每个周期内计算该周期的所有表达式，组合逻辑循环嵌套，带来的是仿真器的循环计算，导致其无法退出该计算，带来了波形停止的现象。

由于波形停止出现时，并不好排查哪里写出了组合环路，我们建议按以下处理：

(1) 一旦发现波形停止，则先对设计进行综合。

(2) 查看综合产生的 Error 和 Critical warning，并尝试修正。比如上图示例中的组合环路，经过 Vivado 的综合后变成了一个多驱动的关键警告，如图 1-5。

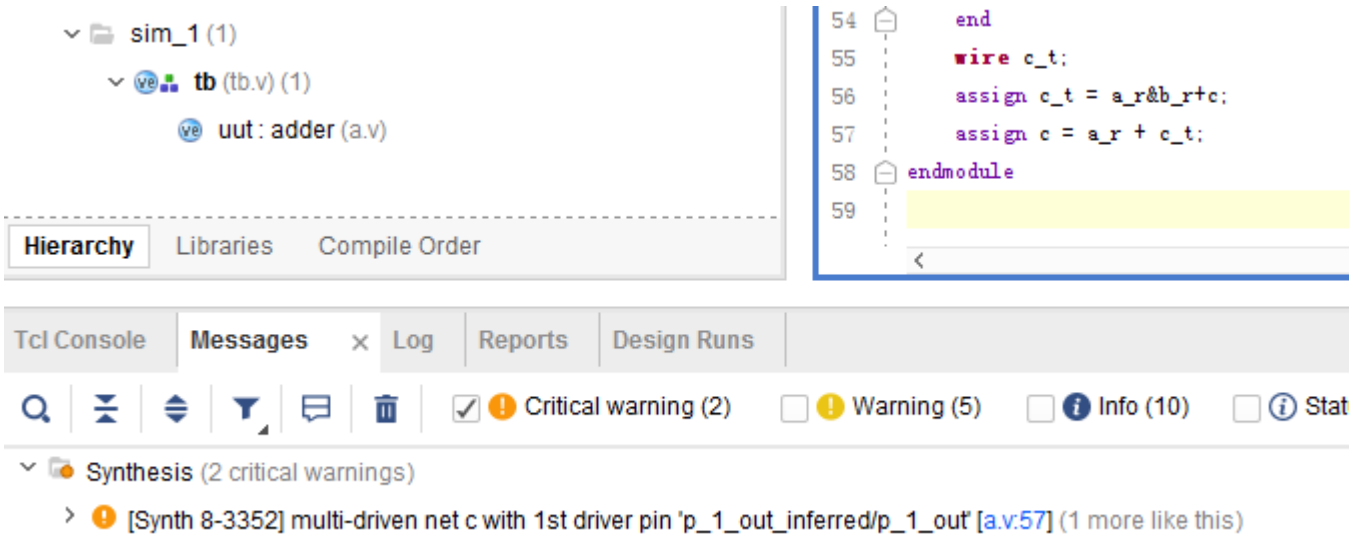


图 1-5 组合逻辑报出多驱动的 critical warning

另外，Vivado 工程中有 TCL 命令 report_timing_summary，会检查组合环路，并报出检查结果。但很遗憾，对于我们上图的示例，该命令并没有检查出组合环路，很有可能和综合时变成了多驱动有关。

1.3.4 越沿采样

越沿采样在波形出错中是一个隐藏较深的出错，往往可能会和逻辑出错混在一起。初看起来，其波形也是很正常的，而且在发生越沿采样后，往往会再执行很长时间才会出错。因而需要大家先按照逻辑出错去调试，最后如果发现数据采样有些异常，就需要甄别下是否是越沿采样的错误了。

越沿采样是指一个被采样的信号在上升沿采样到了其在上升沿后的值，一般情况下，认为这是一个错误，如下图：

```
always @(posedge clk)
begin
    if(!resetn)
        a_r = 1'b0;
    else
        a_r = a;
end
always @(posedge clk)
begin
    a_r_r <= a_r;
end
```

(g) 源码

(h) 波形

图 1-6 越沿采样示例

上图示例中在 105ns 时刻，clk 上升沿到来，a_r 和 a_r_r 同时变为了 1（也就是 a 的值）。a_r 在 105 时刻前是 0，在 105 时刻后是 1。从源码来看，a_r_r 是在上升沿采样 a_r 的值，结果其在 105 时刻采样到 a_r 为 1 的值，也就是采样到了 a_r 在同一上升沿后的值。这就属于越沿采样。

造成这一现象更深层的原因是 Verilog 里阻塞赋值“=”和非阻塞赋值“<=” 混用。上图源码中 a_r 采用阻塞赋值，而 a_r_r 采用非阻塞赋值。

每一次赋值，分为两步：为计算等式左侧的表达式和赋值给右侧的信号，简记为计算和赋值。在一个上升沿到来时，所有由上升沿驱动的信号按以下顺序进行处理：

(1) 先处理阻塞赋值，先完成计算和赋值，同一信号完成计算后立马完成赋值。同一 always 块里的阻塞赋值

从上到下按顺序串行执行，不同 always 块里的阻塞赋值依赖工具实现确定顺序串行执行，一一完成计算和赋值。

(2) 再进行非阻塞赋值的计算。所有非阻塞赋值其等式左侧的值都同时计算好。

(3) 上升沿结束时，所有非阻塞赋值同时完成最终的赋值动作。

从以上描述可以看到，非阻塞赋值是在上升沿的最后一个时间步里完成处理的，晚于阻塞赋值的处理。所以上图示例中，a_r_r 的赋值晚于 a_r 的赋值，造成了越沿采样的情况。

越沿采样，除非特意设计，一般我们认为是一个设计错误，针对越沿采样，我们有一下几点建议：

(1) RTL 编写时注意代码规范，所有 always 写的时序逻辑只允许采用非阻塞赋值。

(2) 一旦发现越沿采样的情况，追踪被采样信号，直到追踪到某一个阻塞赋值的信号，随后进行修正。

1.3.5 波形怪异

本文档将目前未能想到的波形出错的类型都归为波形怪异。

当出现波形怪异类的错时，需要区分其是仿真工具出错还是 RTL 代码出错：

(1) 观察出错的信号，看其生成因子，如果自我判断 RTL 应该没有，且波形显示确实太怪异（比如始终为 32’hxx?x0x?），则很有可能仿真工具出错。重启电脑甚至重建工程试试。

(2) 实在无法从波形里区分出是什么错。可以尝试先运行综合，看出综合后的 Error、Critical warning 和 warning。其中 Error 是必须要修正的，Critical warning 是强烈建议要修正的，warning 是建议能修则修的。

(3) 经常有些不符合规范的代码，Vivado 也不会报出 Warning，需要大家仔细复核自己的代码。常见的隐蔽错误有：对 input 信号进行了赋值，模块调用信号连接错误，reset 信号接成了 clock 信号，等等。

1.4 出错分类二：逻辑出错

逻辑出错则是包罗万象，错误类型是设计的电路功能有错，此时波形界面看起来是很正常的，我们需要利用波形观察各信号的变化，结合预定的电路功能进行定错。

以数据和控制分开来看，逻辑出错可分为两类：数据通路出错和控制信号出错。其中数据通路通常属于较简单的错，比如加法器算两个加数的和，结果不对；而控制信号出错则往往比较难调，往往是设计时的边角问题考虑不周导致的，比如 CPU 的访存系统出错。这些都是逻辑上出错了，但是很不幸的是，在我们未能定位出该错误的源头时，我们往往不能判断出其是数据通路出错，还是控制信号出错。电路设计者在设计之初应当对整个电路有较全面的认识和考虑，尽量减少控制信号出错的情况。

逻辑出错时，不同的电路设计有其特定的调试手段，难以总结出统一的调试手段，但他们的指导思想是一致的：时间上先定错，空间上再定错。

以下我们将针对 CPU 的逻辑出错调试作简单的说明，主要以流水线 CPU 为例进行说明。

1.4.1 CPU 逻辑出错调试

CPU 逻辑出错调试时，首先需要在波形窗口里抓取必要的信息：指令的 PC 值和机器指令编码、指令的写回结果、顶层的访存接口、指令执行的数据通路上的信息。

对于流水线 CPU，需要各流水级信号分组抓出，比如抓出每级流水里的 PC 值、指令编码和执行结果。流水级间的进入和退出的控制信号也尤其重要，必须抓出，CPU 初期调试往往都是流水线控制出错了。还有顶层总线接口也很重要，一旦取指出错了，或者调试访存指令出错了，就需要关注顶层总线接口上的信号了，CPU 后期调试往往都是访存系统出错了。

CPU 逻辑出错调试同样按“时间上先定错，空间上再定错”进行。

(1) 定位出错时间源头

此时需要结合执行的测试程序和 CPU 设计进行定位。定位错误前，需要大家做好以下三项准备：

首先，需要先理解测试程序的大致行为。理解过程中需要阅读测试程序的源码，CPU 设计中会运行的程序有两类：功能测试程序 func 和性能测试程序 coremark、dhrystone 等。功能 func 行为很简单，所有编号的功能点依次测试，每两个功能点测试之间插入 wait_1s 函数。性能测试程序行为则偏复杂，可以不用弄懂，但在调试过程中如有需要应当能很快找到函数间的调用关系。

再者，需要对 mips 汇编和 func 编译环境有一定了解。我们在功能和性能测试程序的编译环境中会生成如下 8 个文件，位于编译目录的 obj/下。其中 test.s 是我们调试过程中重点关注的，它对执行程序的每条机器指令进行了反汇编和注释，对我们的调试很有帮助。

表 1-1 编译生成文件

文件名	解释
data_ram.coe	重新定制 data ram 所需的 coe 文件
data_ram.mif	仿真时 data ram 读取的 mif 文件
inst_ram.coe	重新定制 inst ram 所需的 coe 文件
inst_ram.mif	仿真时 inst ram 读取的 mif 文件
main.bin	编译后的代码段，可以不用关注
main.data	编译后的数据段，可以不用关注
main.elf	编译后的 elf 文件
test.s	对 main.elf 反汇编得到的文件

最后，需要对 CPU 内部逻辑比较熟悉。因为是大家自实现的 CPU，相信大家应该比较熟悉。在定位时间源头时，需要明确 CPU 正在执行的指令的 PC 值、编码和写回结果，将这些信息抓取到波形窗口中，进行对比。

具体调试时，可以采用一下方案：

(1) 在波形最后出错处，确认取会的指令和 PC 值是对应正确的，也就是确认取指正确，这时就需要对照反汇编程序 test.s。

(2) 如果取指不正确，则往前追溯，直到第一个取指正确的地方。追溯的方法也有讲究，有时不能简单一条条指令追溯，因为第一个取指正确的地方可能在很早之前，必要的时候，应跨越一大段指令段，去确认取指是否正确。追溯过程就是程序段不停的压缩，直到找到第一个取指正确的地方，此时往往要用到仿真工具中加标签的方法（）。

(3) 找到第一条指令正确的地方后，可以先确认该指令执行结果是否正确。随后我们的调试目标是确认时间上第一个出错的地方是在该指令前还是在该指令后。确认方法就需要结合测试程序，比如判断该指令位于的函数，确认该函数是否应该执行，其进入条件是否正确。这里有很多种调试方法，需要根据具体情况具体分析，无法很好的总结分类，需要大家在实践中进行总结。总而言之，需要将测试程序代码和 CPU 结合起来联调。

以上方法是由后往前追，如果追溯过程中发现无法再追了，则可以考虑由前往后追。这里的“前”就需要大家好好定位了，甚至可能存在运气的成分，一定要确保这个“前”之前的程序执行时对，这样往后追才能追到正确的第一个错误的店，否则，只会将自己引向错误的方法。

时间上的定错，要求大家对测试程序有一定的了解和掌握，具体调试方法需要大家多多在实践中总结。

值得一提的是，我们的 CPU 实验开发环境 ucas_CDE 仿真时一旦出错，验证平台会很快停止，并打印对比出错的信号，大家最多往前追溯一会就会找到第一条出错的指令。

(2) 定位出错空间源头

在完成时间上的定错后，也就是找到一个执行出错的指令后，需要进行空间定错了。

空间定错时需要大家对 CPU 微结构有更深入的理解。建议大家以空间划分的视角去理解 CPU，特别是流水线 CPU，每一流水级都是有对应的部件的，应当理解清楚各流水级的划分。

空间定位时，有两种方法：

(1) 从 CPU 流水前端向流水后端排查，确认指令在哪个流水级开始出错。首先要排查的就是取指是否正确。

(2) 从 CPU 流水后端想流水前端排查，确认指令是从哪个流水级出错的。首先要排查的就是写回结果是否正确。

1.5 总结

大家在仿真过程中碰到错了可以按需索引本文档，调试过程中记住“时间上先定错，空间上再定错”，不要手忙脚乱，各种信号乱抓，要注意理清调试思路。

设计初期，应当具有全局思路，尽量考虑周到，特别是边角情况，这样可以大大减少后期调试的工作量。当然也不可钻牛角尖，深陷设计考虑中而迟迟不肯动笔，很多设计往往是写着写着就清晰了。

编程过程中，应当注意编码规范，仿真、综合实现中很多怪异现象，以及“仿真通过，上板不过”等很多都是由于代码不规范导致的，强烈建议按照好的规范来编程。

最后，给大家强调下 Verilog 的运算符的优先级，见图 1-7。其中“+”和“—”的优先级是很高的，需要大家留意，比如“assign res[31:0] = a[31:0] + b[0]&&c[0];”，表达式右侧运行结果是一个 1bit 的结果，而不是我们认为的 32bit 的结果，因而上述语句等效于“assign res[31:0] = (a[31:0] + b[0]) && c[0];”。

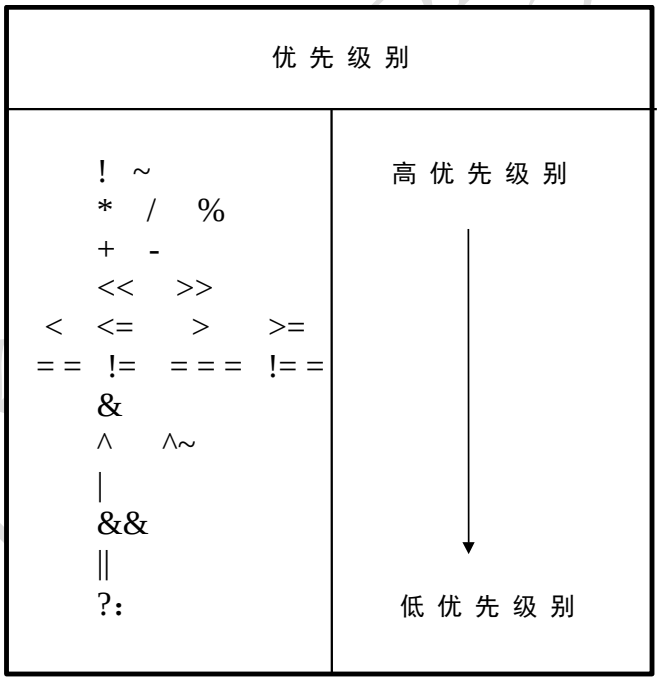


图 1-7 Verilog 运算符优先级