

ASIC 设计中基于 Verilog 语言的 inout(双向) 端口程序设计

王天盛 李斌桥 赵毅强 李树荣 裴志军 姚素英

(天津大学专用集成电路设计中心,天津 300072)

E-mail:echo_dream@163.com

摘要 论文详细介绍了基于 Verilog 硬件描述语言的 inout(双向)端口设计方法,提出了一种与实际情况吻合的仿真方法,并通过 CMOS 图像传感器控制电路设计中一个可综合的设计实例,指出了设计和仿真中应注意的问题。

关键词 ASIC Verilog HDL inout 双向端口 仿真

文章编号 1002-8331-(2003)34-0129-04 **文献标识码** A **中图分类号** TN431.2

Design and Simulation of Inout Ports with Verilog HDL in ASIC Design

Wang Tiansheng Li Binqiao Zhao Yiqiang Li Shurong Pei Zhijun Yao Suying

(ASIC Design Center, Tianjin University, Tianjin 300072)

Abstract: This paper introduces how to design and simulate inout ports with Verilog HDL by a synthesizable module used in the control design of CMOS image sensor, and on the basis of writer's experiences, points out the matters needing attention.

Keywords: ASIC, Verilog HDL, inout, simulation

1 引言

在 ASIC 设计中常常会用到 inout(双向)端口。双向端口顾名思义既可以作为输入端口,也可以作为输出端口。CPU 与存储器所共享的数据总线就是一类双向端口。在很多设计中,如果采用双向端口可以成倍地减少设计的端口数量,提高资源的利用率,缩小芯片面积,降低生产成本。例如,某个设计需要一个 32 位的数据输入端口和一个 32 位的数据输出端口,并且数据输入和数据输出不会同时发生,此时就可以用一个 inout(双向)端口来实现,从而使设计减少 32 个端口。

在许多 Verilog 书籍和参考资料中,有关 inout 端口程序设计和仿真方面的介绍很少,同时一些方法的仿真结果与实际情况不符,论文通过一个在 CMOS 图像传感器控制电路设计中的可综合的设计实例,不仅详细地介绍了基于 Verilog 硬件描述语言的 inout(双向)端口设计方法,并且提出了一种与实际情况吻合的仿真方法。

2 inout(双向)端口的理解与分析

对于含有 inout 端口的模块内部而言,inout 端口可以理解成从“映像寄存器”接收连续赋值的线。作者在定义一个 inout 端口时,同时也要定义一个寄存器作为 inout 端口的“映像寄存器”,并将 inout 端口和这个“映像寄存器”用一个三态门连接起来。当 inout 端口用作输出端口时,将 inout 端口的“映像寄存器”设置成所希望的输出值,并且将三态门选通,这时 inout 端口的值随“映像寄存器”的变化而变化;当 inout 端口用作输入

端口时,三态门设为高阻态,断开“映像寄存器”与 inout 端口的连接,此时就可以像对待普通的输入端口一样对它进行操作。

而对于含有 inout 端口的模块外部而言,需要指定:当它作为输入端口时,其数据的来源,以及当它作为输出端口时,其数据的归属。

3 inout 端口的程序设计

作者以一个在 CMOS 图像传感器控制电路设计中使用的可综合的实例来了解 inout 端口的程序设计和仿真方法。这个实例实现类似 RAM 的功能,主要用来存放在 CMOS 图像传感器工作时需要动态改变的一些参数。为了突出论文的重点 inout 端口的设计且限于文章的篇幅,将这个实例作如下简化: inout 端口的位宽由 8 位变为 2 位;模块中存放动态参数的寄存器个数变为 1 个(从而减少了有关寄存器地址方面的逻辑设计)。图 1 是该实例的模块示意图。实例的模块名为 p_inout。

data: inout(双向)端口; en_n: 片选信号,低电平有效; rd: 从寄存器 state 中读取数据的信号,高电平有效; wr: 向寄存器 state 中写数据的信号,高电平有效。

当 rd 为 1, wr 为 0 时, data 作为输入端口,将 data 上的值存入寄存器 state 中;当 rd 为 0, wr 为 1 时, data 作为输出端口,将 state 中的值从 data 端口读出。而其他情况, data 和 state 都保持各自原有的值不变。

3.1 模块的端口描述

定义输入端口: input en_n, rd, wr;

基金项目:天津市科技攻关重点项目

作者简介:王天盛,硕士研究生,主要研究 CMOS 图像传感器控制电路设计。

计算机工程与应用 2003.34 129

定义双向端口: inout [1:0] data;
 定义双向端口的“映像寄存器”data_reg:
 reg [1:0] data_reg;
 定义一个 2 位的存储器 state, 用来存储输入的值: reg [1:0] state;

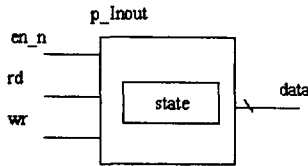


图 1 实例的模块图

3.2 模块的逻辑功能描述

3.2.1 inout 端口与它的“映像寄存器”的连接

用三态门将 inout 端口 data 和它的“映像寄存器”data_reg 连接起来, 可以用 assign 语句来实现:

```
assign data=(~en & rd & ~wr)?data_reg:2'bz;
```

当 rd 为 1, wr 为 0, 即 data 作为输出端口, 且片选信号有效时, data 与它的“映像寄存器”data_reg 的连通并随 data_reg 的变化而变化。当片选信号无效或 data 不作为输出端口时, 赋给 data 高阻态来断开 data 与它的“映像寄存器”间的连接。如果用 synopsys 的 DC 进行综合的话, 这个 assign 语句就实现了用三态门来连接 inout 端口 data 和它的“映像寄存器”data_reg。

3.2.2 分别指定 inout 端口作为输入(出)端口要完成的操作

(1) 当 data 作为输入端口时, 将 data 中的数据存入 state 中:

```
if(~en_n & rd & ~wr)
    state=data;
```

(2) 当 data 作为输出端口时, 将 state 中的数据赋给 data 的“映像寄存器”data_reg:

```
if(~en_n & ~rd & wr)
    data_reg=state;
```

作者希望在信号 en_n, rd, wr, data, state 的值一发生改变就执行所有的操作, 所以用一个 always 语句来实现:

```
always @(en_n or rd or wr or data or state)
    if(~en_n & rd & ~wr)
        data_reg=state;
    else if(~en_n & ~rd & wr)
        state=data;
```

这个模块完整的 Verilog 代码如下:

```
module p_Inout(data, en_n, rd, wr,);
    input      e n_n, rd, wr;
    inout [1:0] data;
    reg [1:0] state;
    reg [1:0] data_reg;
    assign data=(~en & rd & ~wr)?data_reg:2'bz;
    always @(en_n or rd or wr or data or state)
        if(~en_n & rd & ~wr)
            data_reg=state;
        else if(~en_n & ~rd & wr)
            state=data;
endmodule
```

3.3 inout 端口的仿真

130 2003.34 计算机工程与应用

图 2 给出了设计的仿真环境。

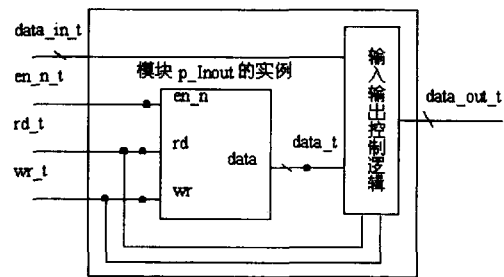


图 2 设计的仿真模块图

3.3.1 构建仿真系统

3.3.1.1 仿真模块的端口定义

(1) 定义片选、写、读端口

```
reg en_n_t, wr_t, rd_t;
```

(2) 定义数据输入端口

```
reg [1:0] data_in_t;
```

(3) 定义数据输出端口

```
wire [1:0] data_out_t;
```

3.3.1.2 模块 p_Inout 的实例化

对于有 inout 端口的模块, 在实例化时, 需要注意的是 inout 端口只能与一个 wire 类型的变量相连。所以对于模块 p_Inout 的 data 端口要定义一个 wire 变量 data_t 来与它连接。对于其他端口则定义相应的 reg 的变量。下面的代码实现模块 p_Inout 的实例化:

```
wire [1:0] data_t;
p_Inout I_p_Inout(.data(data_t), .rd (rd_t), .en_n (en_n_t), .wr (wr_t));
```

3.3.1.3 输入输出控制逻辑的实现

由于 data_t 与一个 inout 端口相连, 所以必须指定它分别作为输入和输出端口时如何与测试模块中的其他端口连接。

当 data_t 作为输入端口时, 由于 data_in_t 是仿真模块的数据输入端口, 是模块 p_Inout 的数据来源, 所以需要将 data_t 与它连通, 而当 data_t 不作为输入端口时, data_t 的值必须赋高阻态来断开二者之间的连接。这样用 assign 语句来实现:

```
assign data_t=(rd_t&~wr_t)?2'bz:data_in_t;
```

同样地, 当 data_t 作为输出端口时, data_out_t 是仿真模块的数据输出端口, 模块 p_Inout 要输出的数据就从这个端口输出, 所以要将二者连通, 而当 data_t 不作为输出端口时, data_out_t 的值也要赋高阻态来断开它们之间的连接。同样地有:

```
assign data_out_t=(rd_t&~wr_t)?data_t:2'bz;
```

至此, 就完成了仿真系统的构建。下面就是向输入端口施加测试信号来检验设计的逻辑功能。

3.3.2 测试信号

测试信号分为三大部分:

(1) 初始化所有输入端口的值

片选信号 en_n_t、写信号 wr_t、读信号 rd_t 都无效 (即 en_n_t 为 1, wr_t 和 rd_t 为 0), 数据输入端口值为 0。

(2) 片选信号 en_n 有效, 设计正常工作

在片选信号有效的前提下, 测试设计的读写功能: 写信号 wr_t 有效后, 从数据输入端口 data_in_t 依次输入 0, 1, 2, 3, 1,

相应地 state 中的值应该随着 data_in_t 的变化而变化, 将写信号变为无效以结束写操作; 使读信号 rd_t 有效, 从 state 中读取数据输出到数据输出端口 data_out_t, 然后将读信号 rd_t 变为无效, 结束读操作; 当读写都无效的时候, 而此时 data_in_t 的数据发生了变化, state 和 data_out_t 的值都不会发生变化。接下来, 向 state 中写入一个数据, 然后马上从 state 中读取出来, 写入的数据有 2'b11, 2'b10, 2'b00, 2'b01。

(3) 片选信号 en_n 无效, 设计停止工作

在片选信号无效的前提下, 执行和前面一样的读写操作, state 和 data_out_t 应该都不会发生变化。

测试文件的部分源代码如下:

```

`include "p_Inout.v"
module p_Inout_testbench;
reg    en_n_t, wr_t, rd_t;
reg    [1:0]data_in_t;
wire   [1:0]data_out_t, data_t;
//模块 p_Inout 的实例化
p_Inout I_p_Inout(.data(data_t), .rd(rd_t),
.en_n(en_n_t), .wr(wr_t));
//输入输出控制逻辑
assign data_t=(rd_t&~wr_t)?2'bz:data_in_t;
assign data_out_t=(rd_t&~wr_t)?data_t:2'bz;
initial begin
//初始化
en_n_t      =1'b1;
wr_t        =1'b0;
rd_t        =1'b0;
data_in_t   =2'b00;

```

//片选信号有效, 设计正常工作。

```

#100 en_n_t   =1'b0;
// data 作为 input, 连续向 state 写数据。
#100 wr_t     =1'b1;
#100 data_in_t =2'b01;
#100 data_in_t =2'b10;
#100 data_in_t =2'b11;
#100 data_in_t =2'b01;
#100 wr_t     =1'b0;
// data 作为 output, 从 state 读数据 2'b01。
#100 rd_t     =1'b1;
#100 rd_t     =1'b0;
// data 作为 input, 向 state 写数据 2'b11。
#100 data_in_t =2'b11;
#100 wr_t     =1'b1;
#100 wr_t     =1'b0;
// data 作为 output, 从 state 读数据 2'b11。
#100 rd_t     =1'b1;
#100 rd_t     =1'b0;
// data 作为 input, 向 state 写数据 2'b10。
#100 data_in_t =2'b10;
#100 wr_t     =1'b1;
#100 wr_t     =1'b0;
// data 作为 output, 从 state 读数据 2'b10。
#100 rd_t     =1'b1;
#100 rd_t     =1'b0;
// data 作为 input, 向 state 写数据 2'b00。
#100 data_in_t =2'b00;
#100 wr_t     =1'b1;

```

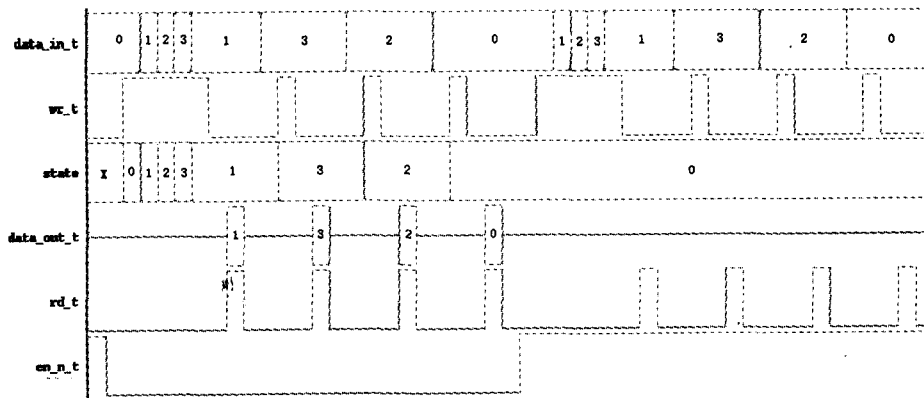


图3 综合前的功能仿真

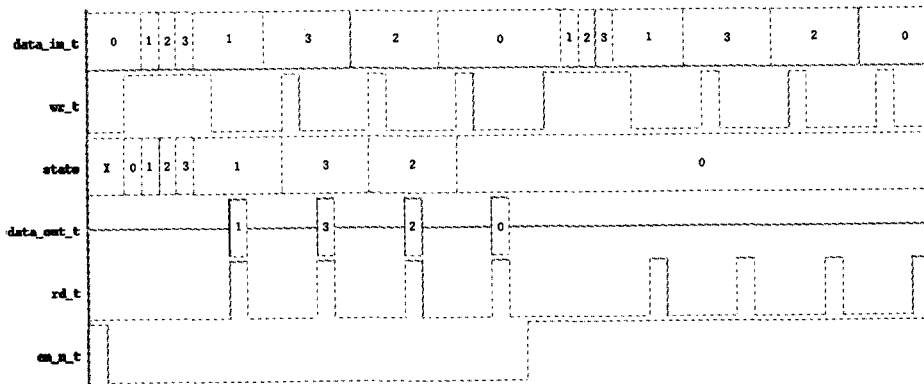


图4 综合后的门级仿真

```

#100 wr_t      =1'b0;
// data 作为 output,从 state 读数据 2'b00。
#100 rd_t      =1'b1;
#100 rd_t      =1'b0;
//片选信号无效,仍然进行和前面相同的
//读写操作,但是设计停止工作。
#100 en_n_t    =1'b1;
//以下对设计读写操作的代码,与片选
//信号有效时的相同,限于篇幅省略之
...
#100 $stop;
end
endmodule

```

3.3.3 仿真结果

图 3 是设计综合前的功能仿真结果。

从仿真的结果可以清楚地看到设计不仅符合要求,而且与实际电路工作时的输入输出结果也是一致的:在片选信号有效时,如果写信号有效,数据输入端口 `data_in_t` 中的数据就被写入 `state` 中,同时数据输出端口 `data_out_t` 由于没有驱动源而呈高阻态。如果读信号有效,就将当前 `state` 中的值输出到 `data_out_t` 上。而在片选信号无效时,读写操作都不被执行,`state` 仍保持原有数据,而 `data_out_t` 一直为高阻态。

3.4 设计综合和综合后的门级仿真

作者用 `synopsys` 的 `DC` 工具对设计进行综合,得到综合后的门级网表和包含时序信息的 `sdf` 文件。这时,就可以进行综合后的门级仿真。

图 4 是综合后的门级仿真结果。

综合后的门级仿真结果与综合前的功能仿真结果相同。

3.5 不同仿真方法仿真结果的比较

在前一种仿真方法中,作者用“输入输出控制逻辑”将 `inout` 端口与数据输入端口 `data_in_t` 和数据输出端口 `data_out_t` 连接起来。而在下面要介绍的这一种方法(以下称为第二种仿真方法)中,用一个“连接模块”来实现这样的连接。

图 5 就是第二种仿真方法的仿真环境。

这个“连接模块”的代码如下:

```

module p_ Join(Join,Join);
input  [1:0] Join;
endmodule

```

实例化这个“连接模块 `p_ Join`”时,`reg` 类型的变量 `data_in_t` 和 `wire` 类型的变量就在该模块内连接起来了。

`p_ Join I_p_ Join(data_in_t,data_out_t);`

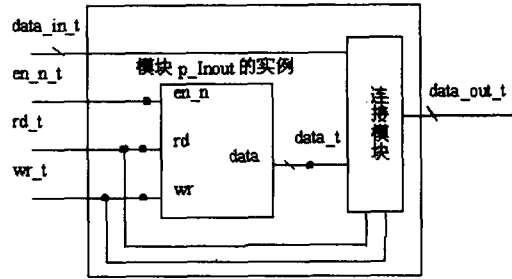


图 5 设计的仿真模块图

然后在模块 `p_Inout` 实例化时,将 `inout` 端口与 `wire` 类型的变量 `data_out_t` 相连。

```

p_Inout I_p_Inout(.data(data_out_t),
.rd(rd_t),.en_n(en_n_t),.wr(wr_t));

```

为了比较这两种测试方法,对设计进行与第一种测试方法完全相同的初始化和读写操作。需要说明的是:在第二种仿真方法中,每次读操作前,需要将数据输入端口 `data_in_t` 赋高阻态来断开它与仿真系统的连接,而将从 `state` 中读取的数据输出到 `data_out_t`。

测试文件的部分源代码如下:

```

#include "p_Inout.v"
#include "p_Join.v"
module p_Inout_testbench;
reg      en_n_t,wr_t,rd_t;
reg  [1:0] data_in_t;
wire  [1:0] data_out_t;
//连接 data_in_t 和 data_out_t。
p_Join I_p_Join(data_in_t,data_out_t);
//模块 p_Inout 实例化,inout 端口与
//wire 类型的变量 data_out_t 连接。
p_Inout I_p_Inout(.data(data_out_t),
.rd(rd_t),.en_n(en_n_t),.wr(wr_t));
initial begin
en_n_t      =1'b1;
wr_t        =1'b0;
rd_t        =1'b0;
data_in_t   =2'b00;
#100 en_n_t  =1'b0;
#100 wr_t    =1'b1;
#100 data_in_t =2'b01;
#100 data_in_t =2'b10;

```

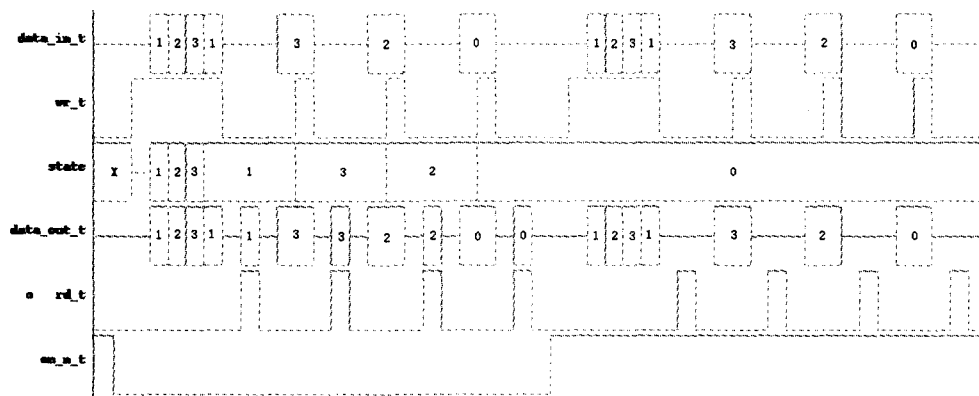


图 6 第二种仿真方法的仿真结果

(下转 183 页)

的当前位置获取相应消息数据,其一次中断过程只处理一个消息。可以认为,BCL/IP 以降低延迟性能为代价增强了其底层通信的可靠性和健壮性(见 3.3.2 节)。

对于机群而言,多结点间的聚集通信(Collective Communication)性能可以更好地反映整个内部通信系统的总体性能。测试采用基于 Socket API 的标准 MPI 性能测试程序分别在 8 结点(每结点 2 进程)BCL/IP 和 GM/IP 平台上进行测试。图 5 所示为其中的全交叉(All-to-All)测试结果。如图可见,BCL/IP 上的 MPI 聚集带宽具有与 GM/IP 类似的性能曲线,前者峰值略高,但随着消息长度的增长,前者的带宽性能稳定在高于后者的水平上。

综上,同 GM 相比,BCL/IP 具有较好的带宽性能(尤其是传输长度较大的消息时)。其小消息延迟性能虽不如 GM,但能提供更好的传输可靠性和系统健壮性,并为其后的 TCP 性能优化与扩展提供了较好基础。

5 结论和下一步研究

BCL/IP 通过在其设备驱动程序中增加一个“TCP 网络驱动”模块,使得 BCL 这一半用户级通信协议能在为用户程序(包括 MPI 和 PVM 库)提供专用 API 的同时直接以二进制兼容的方式支持基于 Socket API 的传统网络应用程序。这不但拓宽了 BCL 乃至曙光 4000L 系统的适用范畴,还在 BCL 底层通信协议的基础上利用高性能机群系统域网(如 Myrinet SAN)大幅度提高了 TCP/IP 的通信性能。论文提供了一种在对系统核心不作修改(或尽可能小的修改)的情况下,利用现有底层通信协议在机群系统域网中实现 TCP/IP 协议的方法。

同曙光 4000L 系统一样,整个 BCL 通信系统目前还处于研制开发过程之中。论文着重介绍了其技术背景和在 Linux 平台上的实现方法,最后给出其在 Linux 实验平台上的性能数据并同国外同类成果 GM 进行了简单对比分析。其点对点带宽性能已经超过 100MB/s,而其单向通信延迟也已低于 40 μ s,远远超过了 TCP/IP 协议通常采用的快速以太网性能(其带宽低于

15MB/s 而延迟高于 70 μ s),和同样基于 Myrinet 的 GM 系统相比其性能基本相当。

由于没有对系统核心中的 TCP/IP 协议栈作任何修改,BCL/IP 具有较好的适用性与移植性,但这也保留了在发送和接收过程中核心内存空间和用户内存空间之间的两次数据拷贝,不利于通信带宽的进一步提高。为拓宽这一瓶颈,可以考虑适当修改核心以去除这两次内存拷贝过程,从而实现全程零拷贝的 TCP/IP 通信。另外,考虑到目前 Myrinet 硬件的网络交换带宽仍旧远低于通信结点中 PCI 总线上的 IO 带宽,作者准备将多网卡并行通信机制引入到 BCL/IP 中,通过在单一结点上同时使用多块 Myrinet 网卡以充分提高网络的点对点通信带宽性能。最后,机群异构网络中的 TCP/IP 通信也是一个值得进一步研究的问题。(收稿日期:2003 年 1 月)

参考文献

1. Ma Jie, He Jin, Meng Dan et al. BCL-3: A High Performance Basic Communication Protocol for Commodity Superserver DAWNING-3000 [J]. Journal of Computer Science and Technology, 2001; 16(6): 522-530
2. Mark Baker. Cluster Computing White Paper. <http://www.dcs.port.ac.uk/~mab/tfcc/WhitePaper/WhitePaper.htm>
3. Boden N J, Cohen D, Felderman R E et al. Myrinet-A gigabit-per-second-local-area network [J]. IEEE Micro, 1995; 15(1): 29-38
4. He Jin, Xu Zhi-wei, Meng Dan et al. Performance Analysis of The Cosmos Cluster File System Based on High-speed Communication Protocol [J]. Journal of Computer Research and Development, 2002; 39(2): 129-135
5. Jeffrey S et al. End System Optimizations for High-Speed TCP [J]. IEEE Commun Mag, 2001; 39(4): 68-74
6. Meng Dan, Ma Jie et al. Semi-User-Level Communication Architecture [C]. In: Proc of IPDPS, 2002
7. Myrinet, Inc. The GM Message Passing System. <http://www.myri.com>, 2001-02

(上接 132 页)

```
#100 data_in_t =2'b11;
#100 data_in_t =2'b01;
#100 wr_t =1'b0;
//读操作前,data_in_t 赋高阻态。
data_in_t =2'bz;
#100 rd_t =1'b1;
#100 rd_t =1'b0;
...
end
endmodule
```

图 6 是第二种仿真方法的仿真结果。

比较两种仿真方法的仿真结果,可以看出第二种方法的仿真结果与实际不符合:在写信号有效时,data_out_t 由于没有驱动源而应该呈高阻态,而仿真结果却是随 data_in_t 的变化而变化。造成这种现象的主要原因是 data_in_t 和 data_out_t 只是通过“连接模块”简单地直接相连,而不是像第一种仿真方法那样通过“输入输出控制逻辑”来分别模拟 inout 端口的数据来源和归属。

4 结束语

对于有 inout(双向)端口的 Verilog 程序设计,要注意以下几点:

(1)对于 inout 端口,要定义一个与之相连的“映像寄存器”。当 inout 端口作为输出端口时,将二者连通;而当 inout 端口不作为输出端口时,要给 inout 端口赋高阻态来断开与“映像寄存器”的连接。

(2)在实例化含 inout(双向)端口的模块时,与 inout 端口相连的只能是一个 wire 类型的变量。

(3)不论是模块设计还是仿真,由于 inout 端口兼有输入端口和输出端口的功能,所以必须分别指定当 inout 端口作为输入端口(输出端口)时,它与其它单元的连接情况和需要完成的操作。(收稿日期:2003 年 10 月)

参考文献

1. 张亮. 数字电路设计与 Verilog HDL [M]. 人民邮电出版社, 2000
2. doulos. The Verilog Golden Reference Guide [M]. 1996