

# 译 者 序

数字集成电路在过去近 30 年的时间里得到迅速发展,EDA 技术在设计过程中起着至关重要的作用。硬件描述语言(HDL)采用形式化的方法,可以直观准确地描述数字电路,应用于模拟验证、设计综合等设计过程中。在众多的硬件描述语言中,Verilog 逐渐发展成为标准的硬件描述语言,是当前 HDL 设计方法学的基础。

硬件描述语言 Verilog 是本书的作者之一 Philip R. Moorby 于 1983 年在英格兰阿克顿市的 Gateway Design Automation 硬件描述语言公司设计出来的,用于从开关级到算法级的多个抽象设计层次的数字设计的建模。该语言提供了一整套功能强大的基元集,包括逻辑门和用户定义的基元;并提供了丰富的结构,这些结构不仅用于硬件的并发性行为的建模,而且用于硬件的时序特性和结构的建模。也可以通过编程语言接口(PLI)对该语言进行扩展。Verilog 语言从诞生起就与生产实际紧密结合在一起,具有结构清晰、文法简明、功能强大、高速模拟和多库支持等优点,并获得许多工具的支持,深受用户的喜爱。据报道,全世界近 90% 的半导体公司都使用硬件描述语言 Verilog。Verilog 实际上是 IC 行业标准,特别是在 1995 年 12 月被 IEEE 接纳为正式标准后,它成为一种很有竞争力的硬件描述语言。

在国内,国家技术监督局于 1998 年正式将硬件描述语言 Verilog 列入国家标准制定项目。本书是 Verilog 的经典著作,由 Verilog 的发明人撰写。本书的翻译工作是配合制订国家标准《集成电路/计算机硬件描述语言 Verilog》(国家标准编号为 GB/T18349-2001,2001 年 10 月 1 日实施)来进行的。在国家标准制订和本书的翻译过程中,得到国内众多专家的帮助和指导,译者在此向北京华大集成电路设计中心王正华研究员,清华大学薛宏熙教授,中科院计算所林宗楷研究员,北京华虹集成电路设计中心李云岗研究员,中科院半导体所薄建国研究员,北京微电子技术研究所王隆望研究员,林守勋研究员,北京理工大学韩月秋教授及陈禾博士表示衷心的感谢。

参加本书翻译的有:蒋敬旗(译前言、第 1、3、4、5、9、10 章),刁岚松(译第 2 章及附录),李杰(合译第 4 章),李春(合译第 5 章),王娟(译第 6 章),吕秀锋(译第 7、8 章)。

全书由刘明业、蒋敬旗和刁岚松统校。

我们在翻译过程中力求翻译准确,但限于译者的水平,一定存在错误和不足之处,恳请读者批评指正。

译 者

2001 年 5 月

· I ·

# 前 言

Verilog 语言是一种在广泛的抽象层次设定说明数字系统的硬件描述语言。这种语言支持早期的行为级抽象设计概念,以及后期结构级抽象设计的实现。它包括层次式结构,从而允许设计人员控制描述的复杂度。

Verilog 最初于 1983 年至 1984 年之间的冬季被设计为一种专用的验证/模拟工具。后来,基于这种语言又开发了其他几种专用分析工具,包括故障模拟器和时序分析器。最近 Verilog 也为逻辑综合工具和行为综合工具提供了输入规范。Verilog 语言有助于提供这些工具的一致性。现在,这种语言已标准化为 IEEE # 1364-1995 标准,并且广泛用于多种工具。本书介绍 Verilog 语言,为该语言的初学者和高级用户提供素材。

有时很难将该语言和模拟工具分离开,原因是该语言的动态特性是由模拟器的工作方式确定的。而且,将它和综合工具分离开也是困难的,因为它的语义是由综合工具所允许的输入规范以及所产生的实现来限定的。我们尽可能避开专用模拟器和专用综合器的细节而集中考虑设计的规范。但是,书中包含的内容足以写出可以工作的可执行模型。

本书采用辅导的方法来介绍 Verilog 语言。首先我们从辅导入门开始,通过举例介绍 Verilog 语言的主要特点和进行系统描述的一般方式。接着详细介绍该语言结构,给出大量的示例,使读者通过这些示例可以更容易地学习和复习。最后,在附录中给出 Verilog 语言的形式化描述。总之,我们的方法是通过研讨举例和做练习题来提供一种学习方式。

我们还给出一些练习题供阅读时思考,并建议读者尽可能早地借助 Verilog 模拟器来试试这些练习题。如果你有自己的设计,也可以进行彻底检验。书中所举的例题以电子信号的形式存储在密封的 CD 上。另外,还可参阅 <http://www.ccc.cmu.edu/~thomas>。CD 上还包括一个模拟器和一个逻辑综合工具,模拟器所处理的描述规模有限,综合工具给出一个时间不很长的综合演示。

本书的大部分内容假定读者具备初步的逻辑设计和软件编程知识。因此,本书适合于进行集成电路设计的工程师、研究生及电子或计算机工程专业的学生。辅导入门中的内容适于作为逻辑设计的入门课程。它首先介绍结构化组合电路,接着介绍更复杂的可综合的组合电路和时序电路,最后介绍周期准确的系统规范。附录中嵌入辅导入门的内容,给出所讨论的一般错误及辅导入门中要解决的练习题。本书也适合作为高层次逻辑设计、集成电路设计、计算机体系结构及计算机辅助设计(CAD)课程的参考教材。本书为设计课程介绍了覆盖该语言的全部内容,为 CAD 课程介绍了模拟器是怎样工作的。

本书共有十章和七个附录。第 1 章是 Verilog 语言的辅导入门。第 2 章和第 3 章讨论该语言的行为建模方式。第 4 章为逻辑级建模。第 5 章包括定时驱动模拟和事件驱动模拟的高级课题。第 6 章和第 7 章介绍语言在综合中的应用。第 8 章和第 9 章讲述更高级的课题:用户自定义的基元和开关级建模。第 10 章推荐两个可以用作大学教学使用的 Verilog 工程。附录 A 为初学者介绍了辅导入门的内容,附录的其他部分是出现在语言手册中更枯燥的问题,读者可根据自己的需要选择阅读。

祝愿您在设计大型系统时获得乐趣。

您永远的朋友

Donald E. Thomas, Philip R. Moorby

## 致 谢

作者向维护和促进 Verilog 标准的 Open Verilog International (<http://www.ovl.org>) 以及对该语言的不断开发做出贡献的 CAD 工具开发人员和系统设计人员表示感谢。特别是,感谢 Leigh Brady 帮助审阅了初稿,感谢 Elliot Mednick 帮助组织了随书所附 CD 中的内容。

作者也感谢 JoAnn Paul 的帮助及对入门介绍和 CD 内容提出的建议,感谢 John Langworthy 帮助收集附录 A 中的辅导材料,感谢 Tom Martin 帮助整理了第 10 章中的练习题,感谢 H. Fatih Ugurdag 提供了例 7.4。我们也感谢使用本书并提供反馈信息的工程师、教师和学生。最后,感谢 Margaret Hanley 设计了封面和版式。

IEEE 通过了 Verilog 硬件描述语言标准(IEEE #1364-1995),感谢他们认可了附录 G 中语法的形式化规范。标准的副本可以从 <http://standard.ieee.org> 获得。

# 目 录

第 1 章 Verilog 语言入门辅导 .....	1
1.1 开始 .....	1
1.1.1 结构描述 .....	1
1.1.2 模拟 binaryToESeg 驱动源 .....	3
1.1.3 为模块建立端口 .....	5
1.1.4 为模块建立测试台 .....	6
1.2 组合电路的行为建模 .....	9
1.2.1 过程模型 .....	9
1.2.2 综合组合电路的规则 .....	11
1.3 时钟时序电路的行为建模 .....	11
1.3.1 建立有限状态机模型 .....	12
1.3.2 综合时序系统的规则 .....	15
1.3.3 非阻塞赋值(“<=”) .....	15
1.4 模块的层次 .....	17
1.4.1 计数器 .....	17
1.4.2 系统时钟 .....	18
1.4.3 将整个电路结合在一起 .....	19
1.4.4 将行为模块和结构模块连接在一起 .....	22
1.5 有限状态机和数据通道 .....	23
1.5.1 简单计算示例 .....	24
1.5.2 系统的数据通道 .....	24
1.5.3 数据通道功能模块的细节 .....	25
1.5.4 用连线将数据通道连在一起 .....	27
1.5.5 FSM 说明 .....	28
1.6 周期精确的行为描述 .....	32
1.6.1 规范方法 .....	33
1.6.2 几点注释 .....	35
1.7 赋值语句的总结 .....	35
1.8 小结 .....	37

1.9	练习	37
<b>第2章</b>	<b>行为建模</b>	<b>39</b>
2.1	进程模型	39
2.2	If-Then-Else	40
2.2.1	else 如何与 if 语句配对	44
2.2.2	条件操作符	45
2.3	循环语句	46
2.3.1	四种基本循环语句	46
2.3.2	循环的异常退出	49
2.4	多分支语句	50
2.4.1	If-Else-If	50
2.4.2	Case	50
2.4.3	Case 和 If-Else-If 的比较	53
2.4.4	Casex 和 Casex	53
2.5	函数和任务	54
2.5.1	任务	56
2.5.2	函数	59
2.5.3	结构视域	61
2.6	作用域规则和层次名	63
2.6.1	作用域规则	64
2.6.2	层次名	66
2.7	小结	66
2.8	练习	67
<b>第3章</b>	<b>并发进程</b>	<b>69</b>
3.1	并发进程	69
3.2	事件	70
3.2.1	事件控制语句	71
3.2.2	有名事件	72
3.3	等待语句	75
3.3.1	一个完整的生产者和消费者握手示例	76
3.3.2	Wait 语句和 While 语句的对比	79
3.3.3	Wait 语句和事件控制语句的比较	80

3.4	并发进程示例	80
3.5	简单流水线处理器	86
3.5.1	基本处理器	86
3.5.2	流水线之间的同步	88
3.6	有名块的终止	89
3.7	赋值语句内部控制和定时事件	91
3.8	过程持续赋值	94
3.9	顺序模块和并行模块	95
3.10	练习	98
<b>第4章</b>	<b>逻辑级建模</b>	<b>100</b>
4.1	引言	100
4.2	逻辑门与线网	101
4.2.1	用基元逻辑门建模	101
4.2.2	四级逻辑值	104
4.2.3	线网	105
4.2.4	模块例示与端口规范	108
4.2.5	有关逻辑级的一个示例	109
4.3	示例数组	115
4.4	持续赋值	118
4.4.1	组合电路的行为建模	119
4.4.2	线网与持续赋值语句声明	120
4.5	参数化定义	123
4.6	行为级/结构级的混合示例	127
4.7	逻辑延迟建模	131
4.7.1	门级建模示例	131
4.7.2	门和线网延迟	133
4.7.3	时间单位的规定	135
4.7.4	最小延迟、典型延迟和最大延迟	136
4.8	模块中的延迟路径	137
4.9	小结	139
4.10	练习	139

<b>第 5 章 高级时序</b> .....	142
5.1 Verilog 时序模型.....	142
5.2 模拟器的基本模型 .....	145
5.2.1 门级模拟.....	145
5.2.2 更通用的模型.....	146
5.2.3 行为级模型的调度.....	148
5.3 模拟算法的不确定行为 .....	150
5.3.1 临近不确定区.....	150
5.3.2 Verilog 是一种并发语言 .....	152
5.4 非阻塞过程赋值语句 .....	155
5.4.1 阻塞和非阻塞赋值的比较.....	155
5.4.2 非阻塞赋值的一般用法.....	156
5.4.3 事件驱动调度算法的扩展.....	157
5.4.4 非阻塞赋值的举例分析.....	159
5.5 小结 .....	162
5.6 练习题 .....	162
 <b>第 6 章 逻辑综合</b> .....	 167
6.1 综合概述 .....	167
6.1.1 寄存器传输级系统.....	167
6.1.2 限制声明.....	168
6.2 使用门和持续赋值的组合逻辑 .....	168
6.3 用来说明组合逻辑的过程语句 .....	170
6.3.1 基础.....	171
6.3.2 复杂形式——推断出的锁存器.....	172
6.3.3 说明无关项.....	174
6.3.4 过程循环结构.....	175
6.4 时序元件的推断 .....	176
6.4.1 锁存器的推断.....	176
6.4.2 触发器的推断.....	178
6.4.3 小结.....	180
6.5 三态器件的推断 .....	180
6.6 有限状态机的描述 .....	181
6.6.1 有限状态机的示例.....	181



6.6.2	FSM 说明的另一种方式 .....	184
6.7	逻辑综合的总结 .....	185
6.8	习题 .....	186
<b>第 7 章</b>	<b>行为综合</b> .....	<b>188</b>
7.1	行为综合的介绍 .....	188
7.2	周期精确的说明 .....	189
7.2.1	always 块的输入和输出 .....	189
7.2.2	always 块的输入和输出关系 .....	190
7.2.3	复位功能说明 .....	193
7.3	米利/摩尔机的说明 .....	194
7.3.1	复杂控制的说明 .....	195
7.3.2	数据与控制路径的折中 .....	196
7.4	小结 .....	199
<b>第 8 章</b>	<b>用户定义的基元</b> .....	<b>200</b>
8.1	组合基元 .....	200
8.1.1	用户定义基元的基本特征 .....	200
8.1.2	组合逻辑电路的描述 .....	202
8.2	时序基元 .....	203
8.2.1	电平敏感的基元 .....	204
8.2.2	边沿敏感的基元 .....	205
8.3	速记表示法 .....	207
8.4	电平敏感与边沿敏感混合的基元 .....	208
8.5	小结 .....	210
8.6	练习 .....	210
<b>第 9 章</b>	<b>开关级建模</b> .....	<b>212</b>
9.1	动态 MOS 移位寄存器示例 .....	212
9.2	开关级建模 .....	216
9.2.1	强度建模 .....	216
9.2.2	强度的定义 .....	218
9.2.3	使用强度的示例 .....	220
9.2.4	电阻型 MOS 门 .....	221

9.3	二义性强度 .....	223
9.3.1	二义性强度的说明 .....	223
9.3.2	基本计算 .....	224
9.4	miniSim 示例 .....	228
9.4.1	概述 .....	228
9.4.2	miniSim 的源码 .....	229
9.4.3	模拟结果 .....	239
9.5	小结 .....	241
9.6	练习 .....	241
<b>第 10 章</b>	<b>工程项目 .....</b>	<b>243</b>
10.1	建立功耗模型 .....	243
10.1.1	对功耗进行建模 .....	243
10.1.2	需要做什么 .....	243
10.1.3	步骤 .....	244
10.2	软盘控制器 .....	245
10.2.1	介绍 .....	245
10.2.2	磁盘格式 .....	245
10.2.3	功能描述 .....	247
10.2.4	真实设备 .....	248
10.2.5	你一直想知道的有关 CRC 的各种情况 .....	249
10.2.6	起支持作用的 Verilog 模块 .....	249
<b>附录 A</b>	<b>学习指南 .....</b>	<b>251</b>
A.1	结构描述 .....	251
A.2	测试台模块 .....	259
A.3	使用 always 的组合电路 .....	259
A.4	时序电路 .....	262
A.5	层次化描述 .....	264
A.6	有限状态机和数据通道 .....	264
A.7	周期精确描述 .....	265
<b>附录 B</b>	<b>词法 .....</b>	<b>266</b>
B.1	空白符和注释 .....	266

B. 2	操作符 .....	266
B. 3	数字 .....	266
B. 4	字符串 .....	267
B. 5	标识符、系统名和关键字 .....	268
<b>附录 C Verilog 操作符 .....</b>		<b>270</b>
C. 1	操作符表 .....	270
C. 2	操作符优先级 .....	272
C. 3	操作符真值表 .....	273
C. 4	表达式的位数 .....	273
<b>附录 D Verilog 门类型 .....</b>		<b>275</b>
D. 1	逻辑门 .....	275
D. 2	BUF 和 NOT 门 .....	276
D. 3	BUFIF 和 NOTIF 门 .....	276
D. 4	MOS 门 .....	277
D. 5	双向门 .....	278
D. 6	CMOS 门 .....	278
D. 7	Pullup 和 Pulldown 门 .....	278
<b>附录 E 寄存器、存储器、整数和时间 .....</b>		<b>279</b>
E. 1	寄存器 .....	279
E. 2	存储器 .....	280
E. 3	整数和时间 .....	280
<b>附录 F 系统任务和函数 .....</b>		<b>282</b>
F. 1	Display 和 Write 任务 .....	282
F. 2	持续监视 .....	283
F. 3	选通监视 .....	283
F. 4	文件输出 .....	283
F. 5	模拟时间 .....	284
F. 6	停止和完成 .....	284
F. 7	随机函数 .....	285
F. 8	从磁盘文件读数据 .....	285

<b>附录 G 形式化语法定义 .....</b>	<b>286</b>
G.1 形式化语法规范指南 .....	286
G.2 源文本 .....	290
G.3 声明 .....	291
G.4 基元示例 .....	293
G.5 模块例示 .....	294
G.6 UDP 的声明和例示 .....	295
G.7 行为语句 .....	296
G.8 Specify 部分 .....	298
G.9 表达式 .....	301
G.10 通用说明 .....	304

# 第 1 章 Verilog 语言入门辅导

数字系统是非常复杂的。从最基本的层次来看,如果我们把一个系统看作逻辑门或传输晶体管的集合,它们可能由数以百万计的元件组成。从更抽象的层次来看,这些元件可以组成一些功能部件,如高速缓存、浮点部件、信号处理器或实时控制器等。硬件描述语言已经发展起来,用来辅助设计具有大量元件、从电路级到逻辑抽象级诸多层次的系统。

数字系统的设计过程是先建立逻辑系统的概念、最终实现必须满足的一组约束以及建立系统的一组基本元件。设计是一个先用手工做或者先用自动综合、然后再根据给出的约束进行测试的迭代过程。一个设计一般可划分为许多更小的部分(根据众所周知的分治工程方法),而各部分可以再划分,直到整个设计用已知的基本元件说明为止。

Verilog 语言为数字系统设计人员提供了一种在广泛的抽象层次上描述数字系统的方式,同时,在这些层次上为计算机辅助设计工具在工程设计中进行辅助设计提供了方法。该语言支持早期的行为结构设计的概念,以及其后层次化结构设计的实现。在设计过程中,进行逻辑结构设计部分时可以将行为结构和层次化结构混合起来。为确认正确性可以将描述进行模拟,也有一些用于自动设计的综合工具。Verilog 语言为设计者进行大型复杂的数字系统设计提供了途径。本章概括介绍了 Verilog 语言的基本特点。

## 1.1 开始

Verilog 语言将一个数字系统描述为一组**模块**。每个模块与其他模块及其本身的描述内容都有一个接口。一个模块代表一个逻辑单元,可以通过规定其内部逻辑结构来进行描述——例如描述实际的逻辑门,或者通过用像程序一样的方式来描述它的行为。在这种情况下主要考虑模块所完成的功能而不是其逻辑实现。然后将这些模块互连起来,使它们能够互相通信。

### 1.1.1 结构描述

首先介绍初级逻辑设计过程中的一个基本逻辑电路:一个二进制七段显示驱动源,如例 1.1 所示。显示驱动源具有一个 4 位二进制输入,驱动七个段显示数字 0 至 9 及十六进制数 A 到 F。本例中所示的只是驱动 E 段的逻辑。

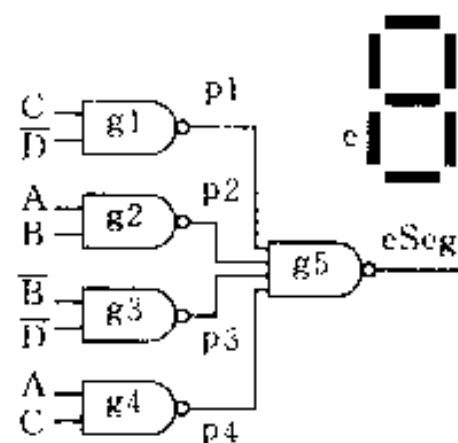
**例 1.1** 二进制七段显示驱动源(仅仅 E 段)

```

module binaryToESeg;
    wire      eSeg, p1, p2, p3, p4;
    reg       A, B, C, D;

    nand #1
        g1 (p1, C, ~D),
        g2 (p2, A, B),
        g3 (p3, ~B, ~D),
        g4 (p4, A, C),
        g5 (eSeg, p1, p2, p3, p4);
endmodule

```



例 1.1 中也给出了这个电路的 Verilog 描述。描述表示的是一个模块的基本定义，模块名为 **binaryToESeg**。每个模块定义包括关键字 **module**，紧随其后的模块名，以及最后的 **endmodule** 语句。描述中的第二行规定在这个模块的子模块间传送逻辑值的 **wire** 的名称。第三行说明保持逻辑值的存储单元名称。这些寄存器是触发器电路元件的一个抽象。

第 5 行及随后的第 6 行到第 10 行示例化 (instantiate) 了 5 个与非门，每个与非门有一个单位时间的延迟。与非门是语言中的一个预定义逻辑门类型，其他类型包括 AND、OR 及 XOR，后边会有详细介绍。这个语句规定电路中的 5 个门，称作 g1 到 g5。“#1”表示每个门都有一个单位时间的延迟。最后，括号内的符号表示连接这些门的连线和寄存器。括号内的第一个符号是门的输出，其他符号是输入。NOT 的运算符 (“~”) 用来规定连接到输入端逻辑值的补码。为进一步说明逻辑图和与之等效的 Verilog 描述之间的一致性，原理图中包括连线、寄存器及示例名称。

尽管这个示例简单，但它列举出 Verilog 语言的几个要点。首先是模块定义与模块例示的概念。使用模块语句，如上例所示，一旦规定了所有的内部细节就定义了一个模块。而后这个模块可以在设计中多次使用 (例示)。每个例示称为模块的一个示例，可以分别命名和连接。基元门，如与非门，是语言提供的预定义逻辑基元。第 4 章内有更详细的介绍。

各种逻辑门是由**线网**(net)连接起来的。线网是语言中两个基本数据类型中的一个 (寄存器是另外一种数据类型)，用来模拟像门这样的结构化实体间的连接。**连线**(wire)是 net 的一种类型，其他类型包括 wired-AND 连接、wired-OR 连接及 trireg 连接。不同的 net 类型分别在第 4 章和第 9 章中有更详细的描述。

我们知道多个逻辑门以层次化方式可以构成更大的模块。在本例中，用与非门来构造 binaryToESeg 模块。如果这个 binaryToESeg 模块有输入输出端口，通过将它例示成另一个模块就可以作为更大模块中的一部分，依此类推。通过将设计划分为更小更有意

义的部分(即子模块),利用层次化描述可以控制设计的复杂度。在例示子模块时,必须知道各子模块的接口,而子模块复杂的实现细节在其他地方描述,因此不用出现在当前模块的描述中。

最后,应当指出 A、B、C、D 命名为寄存器显得不规则。读者可能会认为它们应当是模块 `binaryToESeg` 的输入端,而 `eSeg` 值应当是一个输出端,最终它们会变成输入端和输出端。但是现在我们仍保持这些寄存器定义,并且它们会在下一节的模拟中发挥作用。

参考: 门基元 4.2.1, net 规范 4.2.3。

### 1.1.2 模拟 `binaryToESeg` 驱动源

例 1.2 给出了一个更完整的 `binaryToESeg` 模块定义,称作 `binaryToESegSim`。这个示例包括为 NAND 门示例提供激励的语句,以及监视其输出端变化的语句。尽管没有给出所有可能的输入组合,但给出的这些输入组合将解释怎样提供输入激励。

例 1.2 用于模拟的 `binaryToESeg` 驱动源

```
module binaryToESegSim;
    wire      eSeg, p1, p2, p3, p4;
    reg       A, B, C, D;

    nand #1
        g1 (p1, C, ~D),
        g2 (p2, A, B),
        g3 (p3, ~B, ~D),
        g4 (p4, A, C),
        g5 (eSeg, p1, p2, p3, p4);

    initial          // two slashes introduce a single line comment
        begin
            $monitor ($time, ,
                "A = %b B = %b C = %b D = %b, eSeg = %b",
                A, B, C, D, eSeg);
            // waveform for simulating the binaryToESeg driver
            #10 A = 0; B = 0; C = 0; D = 0;
            #10 D = 1;
            #10 C = 1; D = 0;
            #10 $finish;
        end
endmodule
```

数字系统的模拟器是一个程序,它执行例 1.2 中的 initial 语句(以及我们随后将要看到的 always 语句),并从门和寄存器的输出端将变化的值传播到其他的门和模块的输入端。模拟器的特点还表现在它能保持时间轨迹,使变化的值在将来某一特定时间显示出来而不是立即显示。这些未来的改变值一般存储在按时间排序的事件表中。当模拟器在没有语句可执行或没有值可传播时,就从事件表中找出按时间排序的下一个事件,将时间更新为下一个事件的时间,并执行这个事件。该事件在未来时间可能产生事件也可能不产生事件。模拟循环一直进行,直至不再有模拟的事件或者用户通过其他手段中止模拟。

例 1.2 与例 1.1 不同,它包含驱动模拟的 initial 语句。模拟器通过执行 initial 语句进行模拟的初始化。关键词 begin 和 end 将一些单个语句括起来,它们是初始化语句的一部分。本例的模拟结果如图 1.1 所示。

```

0 A = x B = x C = x D = x, eSeg = x
10 A = 0 B = 0 C = 0 D = 0, eSeg = x
12 A = 0 B = 0 C = 0 D = 0, eSeg = 1
20 A = 0 B = 0 C = 0 D = 1, eSeg = 1
22 A = 0 B = 0 C = 0 D = 1, eSeg = 0
30 A = 0 B = 0 C = 1 D = 0, eSeg = 0
32 A = 0 B = 0 C = 1 D = 0, eSeg = 1

```

图 1.1 例 1.2 的模拟结果

initial 中的第一个语句是一个模拟命令,当任何一个值改变时,监视(和打印)一组值。先打印时间(\$time 请求打印当前时间),然后打印引证串(quoted string),用 A、B、C、D 的值代替 %b(b 代表二进制)串中的打印控制。在 \$time 和引证串之间有几个附加的逗号。需要用一个附加逗号来把 \$time 和引证串分开;其他每个附加逗号引进一个附加空间。当发出监视命令时,监视命令就打印设计中的当前值,随后只要有一个值发生变化将会自动打印出来(但是,当只有 \$time 改变时就不打印)。如图 1.1 所示,开始打印为 X,意思是未知,行中的第一个值是时间。初始语句通过调度将来要发生的 4 个事件来连续工作。语句

```
# 10 A = 0; B = 0; C = 0; D = 0;
```

规定寄存器 A、B、C、D 从当前时间起分别加载 10 个单位时间的 0。本行的执行结果是模拟器将初始语句的执行挂起 10 个单位时间。模拟器检查到在当前(0)时间没有其他动作时,转移到按时间排序的事件表中的下一个事件,重新激活初始语句。在时间为 10 时,初始语句从它挂起的地方重新激活并执行下一个语句。模拟器检查到下一个 # 10 时继续执行下一行。此时初始语句挂起,等待下一个单位时间。

但是在当前时间(时间 10),传播(A、B、C、D 的)变化值。通过传播,将每个基元门连



接到任何一个改变了的值上。而后这些门可以在将来变化时调度它们的输出。由于本例中门的时间延迟规定为 1, 它们的输出变化在未来(在时刻 11)传播一个单位时间。因此, 模拟器调度这些值在将来显示一个单位时间。

如上所述, 初始语句继续执行直到在下一行中找到延迟; 下一行规定了在时刻 20, D 将加载为 1。初始块被挂起并且在时刻 20 被激活。模拟器及时查找下一个事件, 发现有 4 个与非门(g1 到 g4)在时刻 11 需要改变它们的输出值, 并将它们的值传播到最后的与非门 g5。

有意思的是, 门 g1 到 g4 在同一时刻更新各自的输出值。在相同的“模拟时间”时刻 11 都更新各自的值。然而, 模拟器仅能更新 1 个, 而更新的顺序是任意的——我们不能假定哪一个先更新。

在线 p1, p2, p3, p4 上传播这 4 个新值的结果是门 g5 将在时刻 12 改变它的输出值。由于在当前时刻(11)没有其他事件, 下一个事件在下一个时刻(即时刻 12)从事件表中取出来。在时刻 12, eSeg 的变化不会引起其他门的求值, 因为它没有连接到其他门上。

模拟继续进行, 直到初始语句执行了完成命令为止。需要特别说明的是, 在时刻 20, D 被设置为 1。这在两个单位时间之后会引起 cSeg 的改变。在时刻 30, D 被设置为 0, C 被设置为 1, eSeg 将其输出值改变 2 个单位时间。在时刻 40, \$ finish command 停止模拟。

图 1.1 中的模拟器输出值例举了模拟器中一个二进制位具有的 4 个值中的 3 个: 1(真), 0(假)和 x(未知)。第 4 个值 z 用来模拟三态门的高阻输出。注意: 模拟开始时, net 和 register 的值都是 x。

现在我们看看本节示例中 A、B、C、D 为什么被定义为寄存器。由于只有“外部的”与非门输入端, 在模拟期间我们需要一种方式来设置和保持各输入端的值。因为连线不能保持值——它们仅能从输出端将值传送到输入端——寄存器机制就用来保持输入端的值。

我们已经注意到 Verilog 语言中使用了两种主要的数据类型: 线网和寄存器。基元门用来将数值驱动到线网上; 初始语句(及我们在后面将要看到的 always 语句)用来给寄存器赋值。

最后, 我们应该注意到模拟时间是由“单位时间”来描述的。Verilog 描述是用如上所示特定的时间延迟写出的。编译指令 timescale 用来将单位和精确度(四舍五入)连接到这些数据上。本书中的示例不规定实际的时间单位。

参考: 逻辑值 4.2.2; 编译程序指令 4.7.3

辅导: 见附录 A.1 中的辅导问题

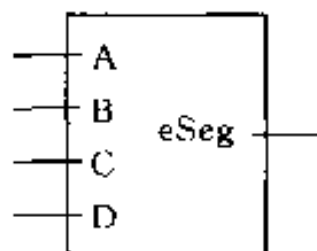
### 1.1.3 为模块建立端口

前面的例 binaryToESeg 既没有输入也没有输出——一种相当有局限性的情况, 在

开发模块层次中是没有用的。本例将扩展定义模块的概念,使模块具有端口。

### 例 1.3 给模块加上端口

```
module binaryToESeg (eSeg, A, B, C, D);  
    output      eSeg;  
    input       A, B, C, D;  
  
    nand #1  
        g1 (p1, C, ~D),  
        g2 (p2, A, B),  
        g3 (p3, ~B, ~D),  
        g4 (p4, A, C),  
        g5 (eSeg, p1, p2, p3, p4);  
endmodule
```



第一行是模块定义语句,模块端口显示在括号内。模块内这些端口必须说明为 inputs, outputs 或双向的 inouts。注意输出端口不一定要出现在第一位,通常基本的与非门就是这种情况。在第二行,说明 eSeg 是一个输出端口。在第三行,说明 4 个输入端口名称为 A、B、C、D。与例 1.2 相比, A、B、C、D 现在是连接输入端口和门的连线。binaryToESeg 模块的逻辑图如示例右边所示。注意逻辑图中所示模块内部的端口名称,它们仅在模块内部已知。

模块可以例示成其他模块。模块定义中的端口表在模块的内部工作和外部应用之间建立起联系,即有一个输出和 4 个输入。模块内其他连接(即连线 p1)不可以连到端口的外面。外部不知道模块的内部结构——模块的内部结构可以用或非门来实现。因此,模块一旦被定义后就是一个黑盒子,可以被多次例示和连接到设计中。但是由于每次例示时不用干涉模块的内部细节,所以我们可以控制设计的描述复杂度。

最后我们注意到例 1.3 中不再说明 eSeg, p1, p2, p3, p4 是连线(在前面例 1.2 中,我们任意选定将它们说明为连线)。由于门只能驱动线网,在默认情况下,它们的名称被隐式地说明为连线。

### 1.1.4 为模块建立测试台

本书中一般都给出了说明 Verilog 语言专有特性的单个模块。但是,在写出 Verilog 描述时,通常采用测试台的方法来组织描述。这种思想来源于工程师的工作台,把要设计的系统连接到一个测试生成器上,测试生成器在被控制的时间间隔内提供输入并监视输出。在 Verilog 中,定义模块时尽可能给出 testBench 的名称。在这个模块内部还有两个模块,一个表示要设计的系统,另一个表示测试生成器和监视器,如图 1.2 所示。

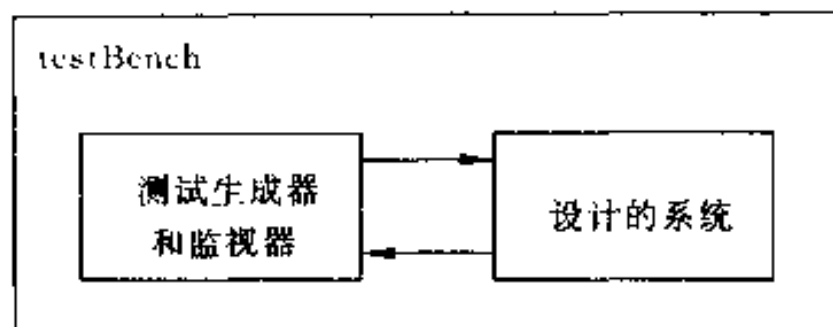


图 1.2 测试台模块的概貌

这种方法清晰地把设计的描述和测试设计描述的方法区分开来。待设计的系统如模块右边所示,可以通过其端口来进行模拟和监视;然后设计可采用其他 CAD 工具来综合。需要指出的是被模拟和被综合的描述是相同的。此外,所有测试功能都密封在模块的右边。如果在设计的模块内包含测试设计的行为,那么在综合时需要删除这个行为——这是一个容易产生错误的过程。例 1.2 的 `binaryToESegSim` 模块显示的是一个将设计的描述(与非门例示)和测试行为(初始语句)结合在一起的模块,例 1.4 显示的是用测试台的方法重新写出的描述。

模块 `testBench` 例示了两个模块:设计模块 `binaryToESeg` 和测试模块 `test_bToESeg`。在例示模块时,如第 4 行和第 5 行所示,给出了名称。第 4 行说明所例示的模块是 `binaryToESeg`,名称为 `d`。第 5 行用名称 `t` 例示 `test_bToESeg`。系统有什么功能(它是一个二进制七段译码器)以及它有哪些端口现在是很清楚的。此外,怎样来测试系统(在测试模块中给出)也是清楚的。测试台方法将这些信息分开,使描述变得清晰,更易于使用像逻辑综合这样的设计工具。这两个模块的连接如图 1.3 所示。

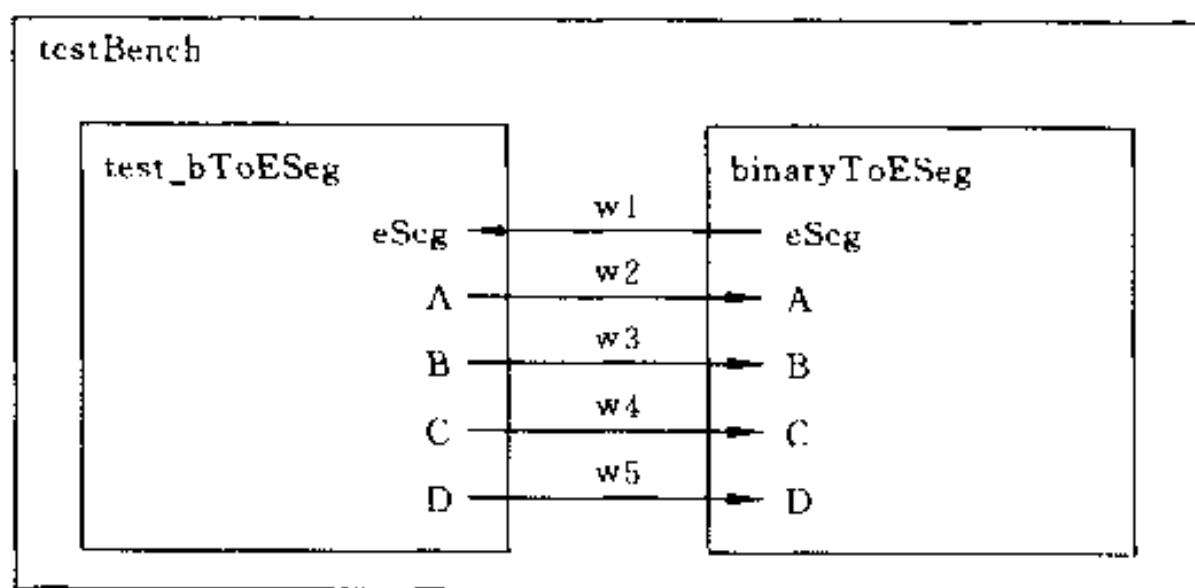


图 1.3 设计模块和测试模块的互连

在将模块连接在一起的时间为 0 的时候需要说明连线。本例中在第 2 行说明的连线 `w1` 至 `w5` 规定了互连。我们可以看到模块 `test_bToESeg` 中的寄存器 `A` 连接到模块的一个输出上。在模块 `testBench` 中,这个端口连接到线 `w2` 上,也连到模块 `binaryToESeg` 的端口 `A` 上。在模块 `binaryToESeg` 中,该端口连到门 `g2` 和 `g4` 上。因此寄存器 `A` 驱动

g2 和 g4 的输入。模拟 testBench 模块与模拟例 1.2 中 binaryToESegSim 模块会产生相同的结果。

#### 例 1.4 采用测试台方式进行描述

```
module testBench;
    wire    w1, w2, w3, w4, w5;

    binaryToESeg    d    (w1, w2, w3, w4, w5);
    test bToESeg    t    (w1, w2, w3, w4, w5);
endmodule

module binaryToESeg (eSeg, A, B, C, D);
    input    A, B, C, D;
    output    eSeg;

    nand #1
        g1 (p1, C, ~D),
        g2 (p2, A, B),
        g3 (p3, ~B, ~D),
        g4 (p4, A, C),
        g5 (eSeg, p1, p2, p3, p4);
endmodule

module test_bToESeg (eSeg, A, B, C, D);
    input    eSeg;
    output    A, B, C, D;
    reg    A, B, C, D;

    initial    // two slashes introduce a single line comment
        begin
            $monitor ($time, ,
                "A = %b B = %b C = %b D = %b, eSeg = %b",
                A, B, C, D, eSeg);
            // waveform for simulating the nand flip flop
            #10 A = 0; B = 0; C = 0; D = 0;
            #10 D = 1;
            #10 C = 1; D = 0;
            #10 $finish;
        end
endmodule
```

```
end  
endmodule
```

在模块 test\_bToESeg 中,说明 A,B,C,D 是寄存器。必须这样说明,原因是在初始块中它们要被赋值。在初始块和 always 块中的赋值必须是寄存器。

你也许会认为模块 test\_bToESeg 中的寄存器 A 和模块 binaryToESeg 中的输入线网 A 是相同的,原因是名称相同。然而在 Verilog 中,每个模块有它自己的名称作用域;在这个示例中的每个模块仅在它所说明的模块内是可知的。因此,两个 A 是不同的实体名,尽管在本例中,连线 w2 将它们连接在一起使它们在电学上相同。

参考:综合 6,名称空间 2.6

辅导:见附录 A.2 中的辅导问题

## 1.2 组合电路的行为建模

我们对 Verilog 语言的观察迄今主要集中在它描述结构的能力上——模块定义、模块示例及其互连。对它在行为上描述模块功能的语言能力观察不够。

一个模块的行为模型是这个模块怎样工作的抽象。模块的输出在描述时与它的输入相关,但是不用结构化的逻辑门来费力地描述模块是怎样实现的。

行为模型在设计过程的初期很有用。在设计初期,设计人员更关心的是模拟这个系统的预期行为以便了解系统总的性能特性,而不用考虑最终实现。随后,采用具有最终实现的准确细节的结构化模型,再重新模拟以便说明功能和时序的正确性。设计过程中的要点在于选定模块的具体实现结构之前,采用行为描述方法来描述和模拟一个模块通常是很有用的。采用这种方法,设计人员可以在工作之前集中精力开发出正确的设计(即能够与系统的其他部分一起正确地工作并具有预期的行为)。然后将这个行为模型作为综合几个结构化的实施方案的起点。

行为模型以类似于编程语言的方式来进行描述。正如我们后面将要看到的那样,我们可以在很多抽象层次模拟系统的行为。对于大型系统,我们可以描述要实现的算法。事实上,算法可以是一个对于 C 这样的编程语言的直接转换。在较低的抽象层次,我们可以描述电路的寄存器传输级行为,说明各个系统的寄存器传输的时钟沿和先决条件。在更低的层次,我们可以描述逻辑门或触发器的行为。在各种情况下,我们都使用 Verilog 语言的行为建模结构来说明模块的功能而不用直接说明它的实现。

### 1.2.1 过程模型

例 1.5 介绍了组合逻辑建模的行为方法。模块的功能是根据过程语句来描述的,而没有用门例示。这里介绍的 always 语句是行为建模的基础。always 语句本质上是一个

“while (TRUE)”语句,包含一个或多个反复执行的过程语句。这些过程语句的执行很像执行一个软件程序:用“=”赋值改变寄存器的值,并执行循环和条件表达式。注意:在 always 语句中,所有使用“=”的赋值都用于说明为寄存器的实体。前面出现过的初始语句也是如此。

### 例 1.5 binaryToESeg 行为模型

```
module binaryToESeg_Behavioral(eSeg,A,B,C,D);
    output      eSeg;
    input       A,B,C,D;
    reg         eSeg;

    always @(A or B or C or D) begin
        eSeg=1;
        if (~A&D)
            eSeg=0;
        if (~A&B&~C)
            eSeg=0;
        if (~B&~C&D)
            eSeg=0;
    end
endmodule
```

CD \ AB				
	00	01	11	10
00	1	0	1	1
01	0	0	1	0
11	0	0	1	1
10	1	1	1	1

这个示例给出二进制七段显示驱动源的行为模型。端口说明与前面相同。我们也说明了一个寄存器 eSeg。这是我们将在 always 语句中进行赋值的寄存器,它也将作为纯组合电路的输出。这个 always 语句由事件控制“@”语句开始。语句:

```
@(A or B or C or D) begin ... end
```

说明模拟器应暂停这个 always 块的执行,直到其中的一个命名实体发生变化。因此,提取出 A,B,C 和 D 中各个数值。当任何一个(或多个)命名实体发生变化时,那么将执行下一条语句——这些语句包含在 begin...end 块内。

当发生一个变化并继续执行时,赋值语句和条件语句很像编程语言那样执行。这种情况下,各语句描述了输出(eSeg)是如何根据输入计算出来的。将这些语句与卡诺图比较,可以看出输出被设置为 1,如果函数的一个零值在输入端上,那么输出设回为 0。当 begin...end 块结束时,always 块重新开始,等待 A,B,C 或 D 的变化。此时,模拟器将 eSeg 的最终值传送给与之相连的设计的其他各个部分。

这个示例中有两个特征要注意。第一,它所描述的行为功能与前面示例相同,但它没有提到具体的门级实现;这个模型是行为级的。

第二, eSeg 被声明为一个寄存器可能会使你觉得它不是一个组合电路。但是, 当仅仅从外部来看它的端口时, 考虑一下这个模块的行为, 会很快得出结论: 任何一个输入上有任何变化, 输出将会只根据模块输入重新计算并驱动输出。这是组合电路的基本特征。从模块外部来看, 显然, 这具有组合电路的行为。

参考: always 2.1; if 2.2

### 1.2.2 综合组合电路的规则

综合工具可以读取电路的行为描述并能够自动设计出该电路的门级结构。因此, 例 1.5 给出输入规范, 综合工具可以产生例 1.3 中规定的设计; 也可以是其他实现方法。

并不是任何行为语句序列都适用于组合电路的综合。为了使用综合工具, 必须十分仔细地写出描述。这里仅概括总结出综合组合电路的规则, 更详细的介绍见第 6 章。确定你要综合的电路是组合电路:

- 检查所有组合函数的所有输入是否都包含在控制事件的敏感表(用“or”分开的名称表)内, 如果其中的一个输入改变了, 那么重新计算输出。

这种要求来源于纯组合电路的定义。组合电路的输出是当前输入的一个函数; 如果一个输入改变了, 输出应当重新计算。

- 检查是否不执行 begin...end 循环就不对组合输出(本例中的 eSeg)赋一个数值。也就是, 在执行每个 begin...end 循环中输出至少被赋值一次。在例 1.5 中, 第 7 行 (eSeg=1;) 对 eSeg 赋一个数值, 满足要求。

为理解这个要求, 考虑一下如果执行 begin...end 循环而不对输出赋值这种情况。在此情况下, 电路需要记住以前的值。因此, 输出就成为当前输入和以前输出的函数相与。这是时序电路的基本特征。这种电路的综合需要有锁存器来实现描述的时序特性。在设计组合电路时, 这是不正确的。

遵循这两条规则有助于你写出既可用于模拟也可用于综合的组合电路的行为描述。

参考: 综合 6

辅导: 见附录 A.3 中的辅导问题

## 1.3 时钟时序电路的行为建模

行为模型也可以用于描述有限状态机。图 1.4 展示了一个具有三种状态的机器的状态转换图, 它有一个输入和一个输出。用两个名称为 Q1 和 Q0 的触发器为这些状态进行编码。复位状态的编码是两个触发器均为 0。图中也给出了使用 D 触发器和逻辑门的一种实现方法。

图 1.5 显示出一个传统的有限状态机。该图实际上说明几个状态寄存器。它们的输

出是机器的当前状态。它们的输入是寄存器在时钟沿过后将要加载的下一个状态。下一个状态是当前状态和输入的组合函数。输出是当前状态和(在某些系统中的)输入的组合函数。这个传统的结构出现在 Verilog 描述中。下一个状态和输出的组合函数将在行为上被描述在一个单独的 always 块中,并遵循前一节中的规则。状态寄存器将被描述在分开的 always 块中并遵循不同的规则。

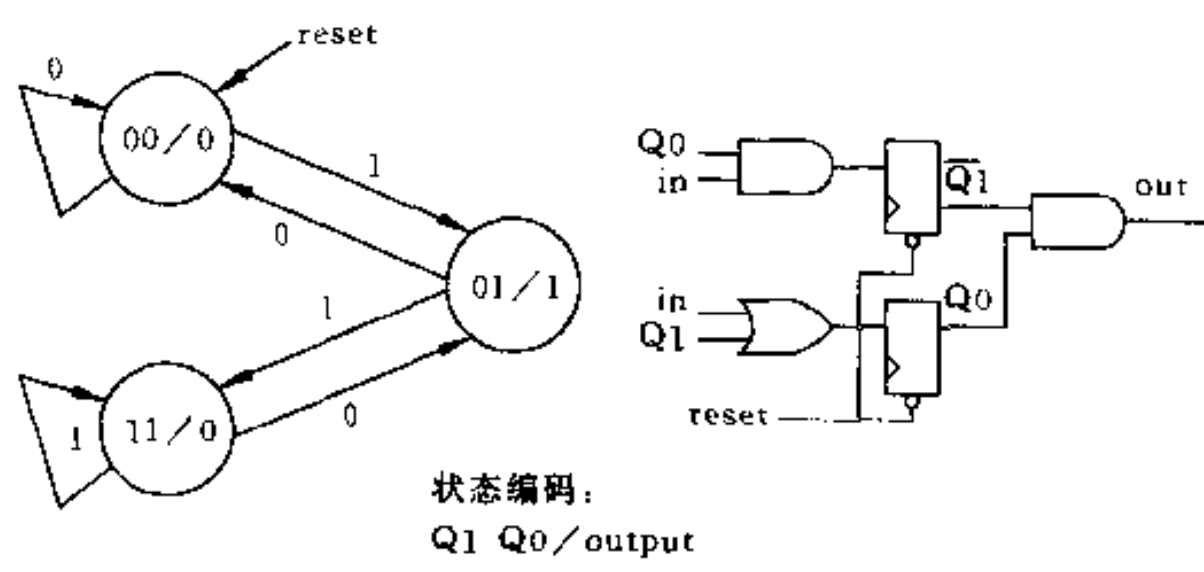


图 1.4 状态转换图

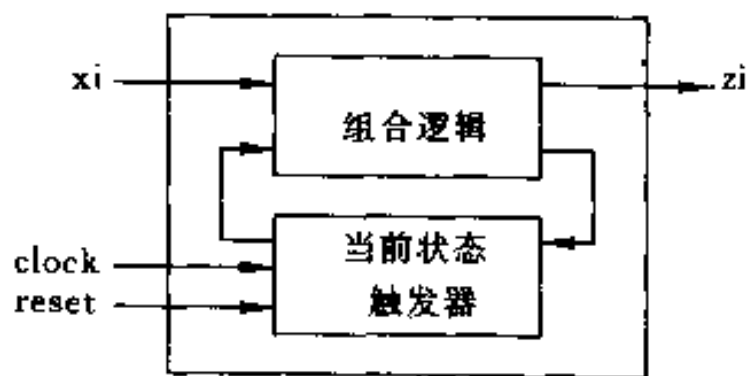


图 1.5 有限状态机标准模型

### 1.3.1 建立有限状态机模型

图 1.4 中机器的行为描述在例 1.6 中。我们把输出命名为 out,输入命名为 in,并且为时钟和复位提供端口。机器的当前状态命名为 currentState。定义

```
reg [1:0] currentState,nextState;
```

表示 currentState 和 nextState 是二位向量。方括号("[ ]")的结构说明了每个寄存器的位数范围,第一个数字是最高位,第二个数字是最低位。out 也被说明为寄存器。它和 nextState 在执行下一个状态和输出函数的组合 always 块内被赋值。本例中我们已经在描述寄存器 nextState 和 currentState 时介绍了术语向量(vector)。寄存器(如 out)和一位的线网被认为是标量的(scalar)。

#### 例 1.6 可综合的有限状态机



```

module fsm(out,in,clock,reset);
    output      out;
    input        in,clock,reset;
    reg          out;
    reg  [1:0]   currentState,nextState;

    always @(in or currentState) begin           // the combinational portion
        out=~currentState[1] & currentState[0];
        nextState=0;
        if (currentState==0)
            if (in) nextState=1;
        if (currentState==1)
            if (in) nextState=3;
        if (currentState==3) begin
            if (in) nextState=3;
            else nextState=1;
        end
    end

    always @(posedge clock or negedge reset) begin // the sequential portion
        if (~reset)
            currentState<=0;
        else
            currentState<=nextState;
    end
endmodule

```

第一个 always 块描述输出和下一个状态逻辑的组合行为。敏感表表示当 in 或 currentState 上出现一个变化时,就执行 begin ... end 语句。begin ... end 中的各个语句规定了组合输出 nextState 和 out 的新值。out 被规定为

```
out=~currentState[1] & currentState[0];
```

它表示 currentState 的第一位的补码(“~”)同 currentState 的第零位相与。结构“currentState[1]”称为向量的一个位选(bit-select)——整个向量中仅有一位在运算中使用。NextState 在随后的 if 语句中计算。考虑第三种情况。

```

If (currentState ==3) begin
    If (in) nextState = 3;

```

```
else nextState = 1;  
end
```

这表明如果 `currentState` 等于 3 (即二位二进制的 11, 相应于状态转换图 1.4 中左下边的状态), 那么 `nextState` 的新值取决于 `in` 的值。如果 `in` 为 True, 则 `nextState` 为 3 (即二位二进制的 11)。否则 `nextState` 为二位二进制的 01。其他的 `always` 语句规定其他状态的 `nextState` 是如何进行计算的。

第一个 `always` 块规定了一个组合电路, 重新检查 1.2.2 节中给出的组合规则是很有用的。首先, 这个组合函数只有输入 `in` 和 `currentState`。这可以通过 `always` 块中赋值语句和条件表达式的右边来进行检查。没有其他命名的实体出现, 所有这些一定是输入。作为组合电路, 以 `or` 分开的事件表一定包含 `in` 和 `currentState`。其次, 组合输出 `out` 和 `nextState` 一定要说明为寄存器并在 `always` 块的每一次执行中都被赋值。它们在 `always` 块中前两行语句中被赋值, 不管它们随后是否被重写。

第二个 `always` 块规定有限状态机的时序部分。我们已经看到过程 1 赋值“=”用在 `initial` 语句和 `always` 语句中。这个 `always` 块引入非阻塞赋值“<=”——一种可描述为并发赋值的赋值语句——使用在具有边沿规范 (即 `posedge` 或 `negedge`) 的 `initial` 语句和 `always` 语句中。到目前为止, 只认为“=”和“<=”是相同的; 以后我们将解释区别。

`always` 块中的敏感表等待两个事件中有一个事件要出现: 或者是一个时钟上升沿, 或者是一个 `reset` 的下降沿。一个信号的上升沿就是它从 0 变为 1, 下降沿就是从 1 变为 0。当其中一个事件或两个事件都出现时, 就执行 `begin ... end` 块。假设 `reset` 上出现一个下降沿。当 `begin ... end` 块开始执行时, `reset` 将为 0, 因此 `currentState` 也将被设置为 0。只要 `reset` 保持 0, 即使 `clock` 出现上升沿, `currentState` 的结果也只设置为 0。这是一个异步复位信号的动作, 它忽略了触发器上的时钟。

现在考虑 `reset` 为 1 并且 `clock` 上出现上升沿的情况; `begin ... end` 循环被执行, 但此时执行 `else` 子句。赋值

```
currentState<=nextState;
```

将 `nextState` 的值送给 `currentState`。这些语句建立了一个两位寄存器的边沿触发的行为模型, 两位寄存器由 D 型触发器组成。

现在, 我们可以理解整个有限状态机模型是如何工作的。假设我们处于状态 0 (`currentState` 为 0), `reset` 为 1 (无效), `clock` 为 0, `in` 为 1。给定了这些值, 那么两个组合输出是: `out` 是 0, `nextState` 是 1。当 `clock` 的上升沿出现时, 第二个 `always` 块将执行并且把 `nextState` 的值赋给 `currentState`, 再等待 `clock` 的下一个上升沿或 `reset` 的下一个下降沿。由于 `currentState` 刚变为 1, 第一个 `always` 块将执行并且计算 `out` 和 `nextState` 的新值。`out` 将变为 1, 如果 `in` 仍为 1, `nextState` 将变为 3。如果 `in` 变为 0, 第一个 `always` 块

将再一次执行,重新计算 out 和 nextState,而与时钟沿无关;nextState 将变为 0,out 将变为 0。

参考: @3.2; if 2.2; bit-select E.1,2.2

### 1.3.2 综合时序系统的规则

除了 1.2.2 节中列出的组合电路规则外,还有时序系统的规则。例 1.6 中的时序部分是第二个 always 块。这些规则是:

- always 块的敏感表仅包括时钟沿、复位和预置条件。

只有这些输入可以引起一个状态改变。例如,如果我们正在描述一个 D 触发器,D 上的一个变化不会改变触发器 D 的状态,所以 D 输入不包括在敏感表内。

- 在 always 块内,首先规定复位和预置条件。如果规定一个复位的下降沿,那么 if 语句应当是“if (~reset)...”。如果等待的是复位的上升沿,则 if 语句应当是“if (reset)...”。

- 在 begin...end 块内没有规定时钟的条件。最后面的 else 中的赋值语句是由综合工具来假定为下一个状态。

- 在顺序 always 块中进行赋值的任何寄存器将采用在综合结果电路中的触发器来实现。因此,你不可能在描述时序逻辑的同一个 always 块中描述纯组合逻辑。你可以写出一个组合表达式,但是表达式的结果将在时钟沿处计算并存入寄存器中,这样的—个示例见例 1.7。

- 当规定边沿敏感的电路行为时,前面的非阻塞赋值(“<=”)是特别的赋值操作符。“<=”表示规定在敏感表中的边沿上出现的整个系统的所有转换将并发地出现。尽管采用常规的“=”描述能正确地综合,但都不能够正确地模拟。因为模拟和综合通常是很重要的,所以使用“<=”。

尽管这些规则看起来好像是很“随意的(picky)”,但对于综合工具推导出被综合的电路需要触发器,进而推导出它是如何连接的,这是很必要的。

最后,注意一下 fsm 模块。名称 clock 和 reset 的使用对于综合工具没有特别的含义。我们在示例中使用这些名称是为了清晰,它们在模型中可以被任意命名。通过使用例 1.6 中所示的规范形式,综合工具可以推导出需要一个触发器,并知道有哪条线连接到它的 D、clock 和 reset 输入上。

### 1.3.3 非阻塞赋值(“<=”)

非阻塞赋值用来使赋值语句同步,所以它们仿佛的同时——并发地出现的。如模块 fsm 中所示的边沿使用非阻塞赋值。当规定的边沿出现时,新值就并发地存储在正在等待信号边沿的所有的赋值中。与通常的赋值(“=”)相比,等待信号边沿的所有赋值的右

边被首先计算,然后左边再被赋值(更新)。将这看作为所有的赋值都是并发出现的——在同一时刻——不依赖于描述中任何阻塞语句。实际上,在大型数字系统中,所有触发器是由相同的时钟边沿控制的,非阻塞赋值模拟了这个行为。

考虑例 1.6 中 fsm 模块的另一种表示方法,如例 1.7 所示。此时 Verilog 几乎是直接根据图 1.4 中的逻辑图进行描述的。我们将当前状态触发器模型化为分开命名的寄存器 cS0 和 cS1,并且我们在第二个 always 块——时序块中包含了下一个状态等式。模块 fsm 和 fsmNB 应综合出相同的硬件。

再考虑第二个 always 块是怎样进行工作的。该模块等待一个 clock 的上升沿或 reset 的一个下降沿,如果 reset 上出现一个下降沿,那么 cS0 和 cS1 被设置为 0。如果时钟上出现一个上升沿,则计算两个“ $\leq$ ”赋值的右边,然后对左边寄存器进行赋值。从而计算出“ $Q0 \& in$ ”(Q0 和 in 的与)和“ $Q1 | in$ ”(Q1 和 in 的或),然后将结果分别赋给 cS1 和 cS0。

检查描述时,应当考虑下述两条语句

```
cS1 <= in & cS0;  
cS0 <= in | cS1;
```

是同时(即并发)出现的。将右边当作是两个触发器的输入,并且 cS0 和 cS1 中的变化在时钟边沿出现时才会出现。应当认识到它们是并发出现的。第一行中左边的 cS1 不是用在第二行右边的 cS1 的值,右边的 cS0 和 cS1 是时钟边沿出现之前的值。左边的 cS0 和 cS1 是时钟边沿出现之后的值。这些语句可以用时钟边沿出现之后 cS0 和 cS1 的相同结果数值的任意顺序写出。

### 例 1.7 说明非阻塞赋值

```
module fsmNB (out, in, clock, reset);  
    output          out;  
    input           in, clock, reset;  
    reg             out, cS1, cS0;  
  
    always @(cS1 or cS0)    // the combinational portion  
        out = ~cS1 & cS0;  
  
    always @(posedge clock or negedge reset) begin    // the sequential portion  
        if (~reset) begin  
            cS1 <= 0;  
            cS0 <= 0;  
        end  
    end
```

```

else begin
    cS1 <= in & cS0;
    cS0 <= in | cS1;
end
end
endmodule

```

这个示例说明用非阻塞赋值规定的功能。在一个完整的设计中,许多不同的模块会有很多 always 语句等待相同信号的相同边沿。非阻塞赋值功能强大的特点在于所有的右边表达式是在左边寄存器更新之前来被计算出来的。因此,你不必担心用 cS1 的哪个值来计算 cS0。使用“<=”使你明白使用的是在时钟边沿出现之前的那个值。

辅导:见附录 A.4 中的辅导问题

## 1.4 模块的层次

现在建立一个更大的设计示例:包括更多个不同的部件,如图 1.6 所示。本节将详述每一个模块以及它们之间的互连。该例由一个 board 模块组成,它包括一个 clock 模块(m555),一个 4 位计数器(m16)及 1.2 节中的 binaryToESeg。

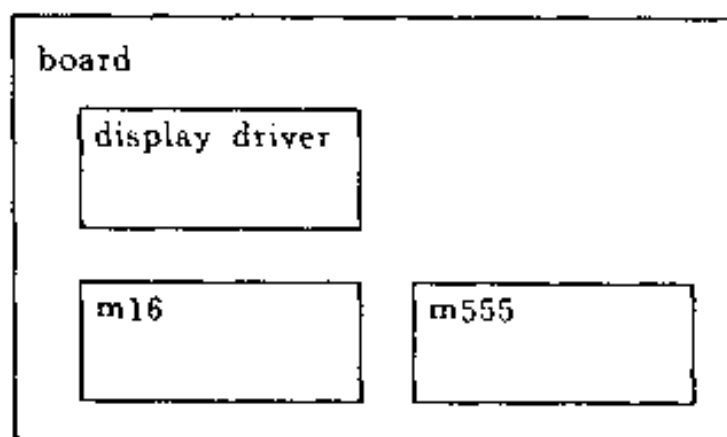


图 1.6 计数器

### 1.4.1 计数器

我们首先观察例 1.8 中所示的计数器模块定义。计数器有 2 个端口:一个加 1 计数器的时钟和 4 位计数器数值 ctr。该例声明内部寄存器 ctr 和它的输出端口是 4 位向量。计数器采用一个 always 块来进行行为建模。该模块等待时钟上升沿。当上升沿出现时,ctr 加 1,模块等待下一个时钟上升沿。由于新的计数器数值产生在信号的边沿上,因此使用了非阻塞赋值操作符(“<=”)。

**例 1.8 四位计数器**

```

module m16 (ctr, clock);
    output [3:0] ctr;
    reg [3:0] ctr;
    input clock;

    always @(posedge clock)
        ctr <= ctr + 1;
endmodule

```

## 1.4.2 系统时钟

计数器需要一个时钟来驱动它。例 1.9 抽象地定义一个称作 m555 的“555”定时器芯片,并给出了模拟描述所产生的波形。

### 例 1.9 计数器的时钟

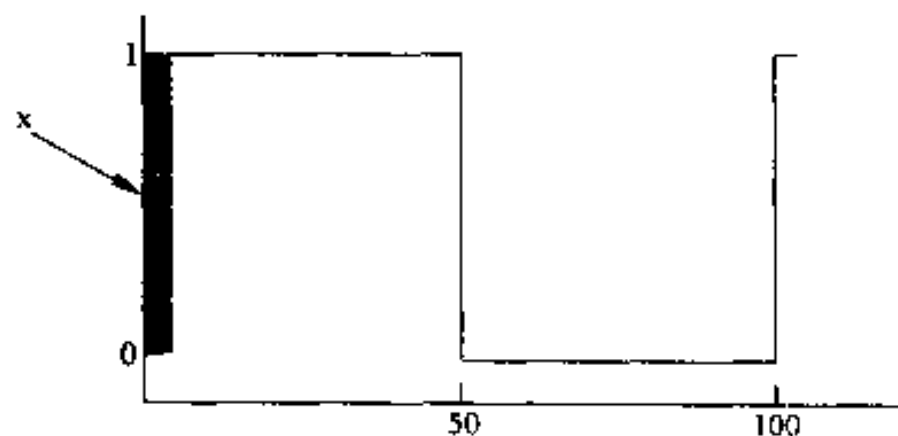
```

module m555 (clock);
    output clock;
    reg clock;

    initial
        #5 clock = 1;

    always
        #50 clock = ~ clock;
endmodule

```



m555 模块有一个内部寄存器(clock),它也是模块的输出。在模拟开始时,输出值是 x,如该例时序图中灰色区域所示。clock 在 5 个单位时间过后被初始化为 1。m555 用一个 always 语句来进行行为建模,该 always 语句说明 clock 在 50 个单位时间后取其补值。由于一个 always 语句本质上是一个“while (TRUE)”循环,第一个 50 个单位时间过后,在另一个 50 个单位时间内调度执行 always 语句并改变它的值;即,这时时间为 100。由于 clock 每隔 50 个单位时间将改变它的数值,因此我们建立一个周期为 100 个单位时间的时钟。

我们可以用实际时间单位来规定时钟周期。时标编译指令用来规定延迟操作符( # )的时间单位,并且时间计算的精度是四舍五入的。如果编译指令

```
`timescale 1ns/100ps
```

在模块定义之前被设置,那么该模块中所有延迟操作符和随后的任何模块将以纳秒为单

位,任何时间的计算在内部将对最接近 100ps 进行四舍五入。

参考: 时标 4.7.3

### 1.4.3 将整个电路结合在一起

现在我们已经定义在本系统中要用到的基本模块,剩下要做的是将它们结合在一起完成图 1.6 所示的设计。例 1.10 通过定义另一个模块(称作 board)将例 1.3、例 1.8 和例 1.9 中的模块定义结合在一起,而模块 board 例示这些模块并将它们互连在一起,如图 1.7 所示。

#### 例 1.10 计数器的顶层模块

```
module board;
    wire    [3:0]    count;
    wire                clock, eSeg;

    m16                counter        (count, clock);
    m555                clockGen       (clock);
    binaryToESeg       disp           (eSeg, count[3], count[2], count[1], count[0]);

    initial
        $monitor ($time, , , "count = %d, eSeg = %d", count, eSeg);
endmodule
```

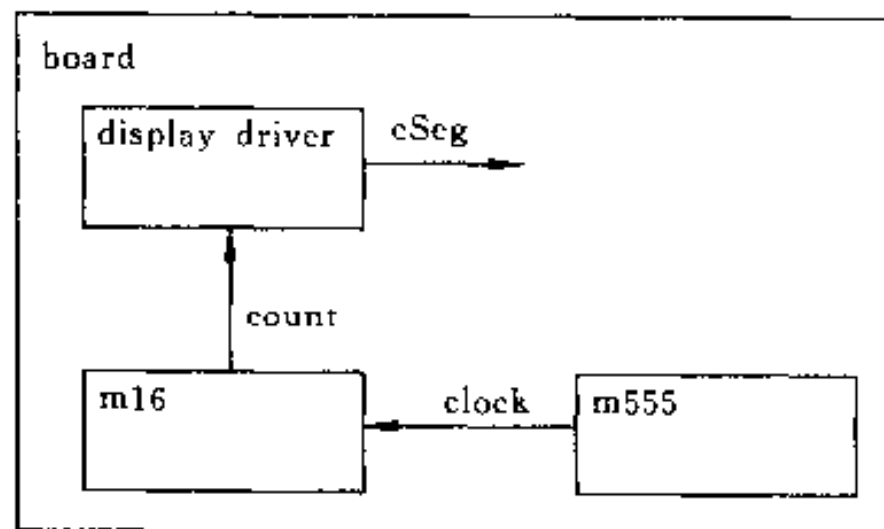


图 1.7 带有互连的计数器示例

在 board 模块定义中的大多数语句在前面已经描述过,但还有几个细节需要指出来,计数器模块说明给出两个端口 ctr 和 clock。

```
module m16(ctr, clock);
```

ctr 是一个 4 位输出,而 clock 是一个 1 位输入。例 1.10 中计数器的输出与

binaryToESeg 模块相连,但是,该模块定义具有 5 个 1 位端口。

```
module binaryToESeg(eSeg,A,B,C,D);
```

在 board 模块中,我们定义一个四位线网 count 连接到 m16 上,当我们将它连到 binaryToESeg 上时,我们需要分别连接每一位(A 到 D)。我们用连线类型的 count 的位选(bit-select)来做这个工作,为每个连接规定合适的位,位选允许在被声明为向量的寄存器或连线内指定一位。因此,在模块 board 中 binaryToESeg 的连接变成

```
binaryToESeg disp (eSeg, count[3], count[2], count[1], count[0]);
```

这将 count[3]连到 A, count[2]连到 B, count[1]连到 C, count[0]连到 D。

作为一种选择方案,如例 1.11 所示,当与 m16 模块连接时,我们可以声明四个标量线,然后将它们连在一起。这里,我们定义标量线 w3, w2, w1, w0,并将它们连到 binaryToESeg 的端口 A,B,C,D 上。然而,模块 m16 期待一个 4 位向量连接到它的 ctr 端口。并置操作符“{…}”允许将几个线合并在一起形成一个多位线。这里 w3, w2, w1, w0 结合(并置)在一起,在与 m16 连接时作为一个 4 位实体。

#### 例 1.11 顶层模块的一种方案

```
module boardWithConcatenation;
    wire          clock, eSeg, w3, w2, w1, w0;

    m16            counter    ({w3, w2, w1, w0}, clock);
    m555           clockGen    (clock);
    binaryToESeg    disp       (eSeg, w3, w2, w1, w0);

    initial
        $monitor ($time, , "count=%d, eSeg=%d", {w3,w2,w1,w0}, cSeg);
endmodule
```

如果将例 1.3、例 1.8、例 1.9 和例 1.10 中的模块定义一起编译,可以形成一个可被模拟的完整描述。尽管这样做很有意义,但是什么也不会发生! 因为 ctr 没有被初始化,它是未知的(x)。当它在 clock 上升沿被加 1 时,对于一个未知数,加 1 的结果仍然是未知的。为使这个模拟模型能够工作,ctr 需要初始化。将语句

```
initial ctr=1;
```

插入到模块 m16 中将会解决这个问题。将这些组合在一起的示例(包括 ctr 的初始化)模拟 802 个单位时间,模拟轨迹如图 1.8 所示。

起初,在时刻 0,系统中所有的值都是未知的,然后使所有初始块和 always 块能够运



行;它们以任意顺序开始运行。m555 中的初始语句开始时分别延迟 #5 和 #50。m16 中的 always 开始时等待 clock 的上升沿。binaryToESeg 中的门基元等待改变它们的输入。board 和 m16 中的初始语句也可以运行。我们可以看到 m16 首先执行将 ctr 设置为 1。然后,board 中的初始语句运行,执行 \$ monitor statement 并打印图中的第一行(如果 ctr 初始化执行 \$ monitor statement,那么 count 打印 x)。

假定 ctr(count)在时刻 0 设置为 1,2 个单位时间后 eSeg 将值改变为 0(当显示 ctr 为 1 时,eSeg 为 0)。在时刻 5,clock 从 x 变为 1。在 Verilog 中,这翻译为一个上升沿,它将 ctr(count)变为 2。2 个单位时间后,在时刻 7,eSeg 改变为 1,因为在显示 ctr 为 2 时 eSeg 为 1。在时刻 50,clock 改变为 0。但在我们的模拟中没有显示出来,因为我们没有监视 clock 的变化。在时刻 100,clock 变为 1,建立一个 clock 上升沿并使 ctr(count)加 1。ctr 变为 3 并且 eSeg 在 2 个单位时间后相应改变。模拟继续执行,如图 1.8 所示。

```

0 count=1,eSeg=x
2 count=1,eSeg=0
5 count=2,eSeg=0
7 count=2,eSeg=1
100 count=3,eSeg=1
102 count=3,eSeg=0
200 count=4,eSeg=0
300 count=5,eSeg=0
400 count=6,eSeg=0
402 count=6,eSeg=1
500 count=7,eSeg=1
502 count=7,eSeg=0
600 count=8,eSeg=0
602 count=8,eSeg=1
700 count=9,eSeg=1
702 count=9,eSeg=0
800 count=10,eSeg=0
802 count=10,eSeg=1

```

图 1.8 例 1.3、例 1.8、例 1.9 和例 1.10 的模拟轨迹

我们注意到初始语句对于模拟例 1.8 是必要的,但是在综合时都是不必要的。实际上,逻辑综合忽略了初始块。

参考:模块例示 4.2.4; always 2.1; \$ display F.1

#### 1.4.4 将行为模块和结构模块连接在一起

现在把分别在几个示例中定义的模块连接在一起。其中一些模块在结构上只采用门级基元来定义。一些模块在行为上是采用 `always` 块来定义的。这是语言功能强大的一个方面,因为它允许我们在一个详细的层次(即结构化模型)上建立系统的一部分模型,而系统的其他部分建立在不很详细的层次(行为模型)上。在设计工程的开始,系统中大部分是建立在行为级上,然后一部分被细化为结构模型。而最终的模拟结果与进行准确的时序模拟及功能模拟的门级定义的模块相同。因此,Verilog 有助于完成整个设计过程,允许设计从行为级到结构级进行转换。

例 1.10 及其各子模块局部地显示行为元件和结构元件是怎样连接在一起的。在本例中,例 1.3 的结构模块 `binaryToESeg` 与例 1.8 的行为模块 `m16` 连接在一起。`m16` 中的寄存器 `ctr` 声明为一个输出。`ctr` 的任何变化从模块端口进行传播,最终传到门的输入端。因此,我们看到行为模型中规定的寄存器可以驱动门基元的输入端。在分开的模块中不需要这样做。

我们将如例 1.12 所示的两个模块的功能组合在一起。这样,在一个模块内我们既有结构元件也有行为元件。无论在什么时候 `ctr` 被更新后,门 `g1` 至 `g4` 将重新计算它们的输出,因为它们的输入连接在 `ctr` 上。因此,一个 `always` 块的“输出”——由 `always` 块赋值的寄存器的值——可以用作门级基元的输入。

同样,门级基元的输出可以作为 `always` 块的“输入”,如例 1.13 所示。这里我们将原来的结构模块 `binaryToESeg` 改变成产生 `mixedUpESegDriver`。变化是,与其他 NAND 门的输出 NAND 在一起的最终的 NAND 门在行为上是采用一个 `always` 块来描述的。这个 `always` 块等待 `p1`, `p2`, `p3` 或 `p4` 上的任何一个变化。当有一个变化出现时,行为语句计算它们的 NAND 并将它存储在寄存器 `eSeg` 中,这个值是模块的组合输出。因此,门基元的输出可以驱动行为模块的输入端——行为表达式右边的值。

##### 例 1.12 驱动结构的行为

```
module counterToESeg (eSeg, clock);  
    output          eSeg;  
    reg [3:0]       ctr;  
    input           clock;  
  
    initial  
        ctr = 0;  
  
    always @(posedge clock)
```

```

ctr <= ctr + 1;

nand #1
    g1 (p1, ctr[1], ~ctr[0]),
    g2 (p2, ctr[3], ctr[2]),
    g3 (p3, ~ctr[2], ~ctr[0]),
    g4 (p4, ctr[3], ctr[1]),
    g5 (eSeg, p1, p2, p3, p4);
endmodule

```

### 例 1.13 驱动行为的结构

```

module mixedUpESegDriver (eSeg, A, B, C, D);
    output          eSeg;
    reg             eSeg;
    input           A, B, C, D;

    nand #1
        g1 (p1, C, D),
        g2 (p2, A, ~B),
        g3 (p3, ~B, ~D),
        g4 (p4, A, C);

    always @(p1 or p2 or p3 or p4)
        eSeg = ~(p1 & p2 & p3 & p4);
endmodule

```

这些示例用来说明 Verilog 语言中两个主要数据类型：寄存器和线网，并说明它们是如何在一起工作的。门基元将它们的输出驱动到线网（即上述示例中的连线）上。门输入既可以是其他的线网，也可以是寄存器。行为模型，即 always 块，改变寄存器的值作为它们的执行结果。它们的输入既可以是其他寄存器，也可以是由门基元驱动的线网。

参考：过程赋值 2.1；连续赋值 4.4；时序模型 5.1

辅导：见附录 A.5 中的辅导问题

## 1.5 有限状态机和数据通道

上述已经使用 Verilog 语言来规定组合逻辑和有限状态机。现在我们开始说明寄存器传输级系统。我们将使用一种规范方法，称为有限状态机和数据通道或 FSM-D。我们

的系统将由两部分构成：一个可以做计算并将结果存入寄存器的数据通道，一个控制数据通道的有限状态机。

### 1.5.1 简单计算示例

我们先做一个简单计算并说明怎样用 Verilog 语言规定逻辑硬件。以类 C 的语法计算如下所示：

```
...
for (x=0,i=0;i<=10;i=i+1)
    x=x+y;
if (x<0)
    y=0;
else x=0;
...
```

计算一开始将  $x$  和  $i$  清为 0，然后，当  $i$  小于或等于 10 时， $x$  被赋值为  $x$  和  $y$  的和，并将  $i$  加 1。当退出循环时，如果  $x$  小于 0， $y$  被赋值为 0，否则， $x$  被赋值为 0。用这个示例来说明建立更大系统的方法。

我们假定这些是 8 位计算，因此系统中所有寄存器都是 8 位的。

### 1.5.2 系统的数据通道

用硬件实现这个计算有很多方法，我们只集中考虑一种方法。这个系统的数据通道必须用寄存器表示  $x$ ， $i$  和  $y$ 。需要能够将  $i$  加 1，将  $x$  和  $y$  相加，并能清除  $i$ ， $x$ ， $y$ 。也需要能够将  $i$  和 10 比较，将  $x$  和 0 比较。图 1.9 显示出能够执行这些寄存器传输的一个数据通道。

图中每个方框的名称建议用它的功能来命名。名称上有横线的是控制信号，它们是低电平有效。在标为 register  $i$  的块上，我们看到它的输出（从底部产生）连接回到加法器的输入端，而加法器其他输入端连接到 1 上。加法器的输出（从底部产生）连接到 register  $i$  的输入端。假定寄存器存储一个值，而加法器是个组合电路，register  $i$  的输入将总是比 register  $i$  的当前值大 1。寄存器也有两个控制输入端： $iLoad$  和  $iClear$ 。当其中一个输入端有效时，规定的功能将出现在下一个时钟沿上。如果我们发出  $iLoad$  信号，那么下一个时钟沿之后 register  $i$  将加载并存储它的输入值，同时将  $i$  加 1。另一方面， $iClear$  将 0 存入 register  $i$ 。比较模块也是组合电路，并产生指定的布尔结果。

计算中给出的寄存器传输是  $x=0, i=0, y=0, i=i+1, x=x+y$ 。从上面数据通道是如何工作的描述中，我们可以看出，计算中所有的寄存器传输可以在这个数据通道上执行。此外，所有在 FSM 中需要分叉的条件值也在数据通道中产生。

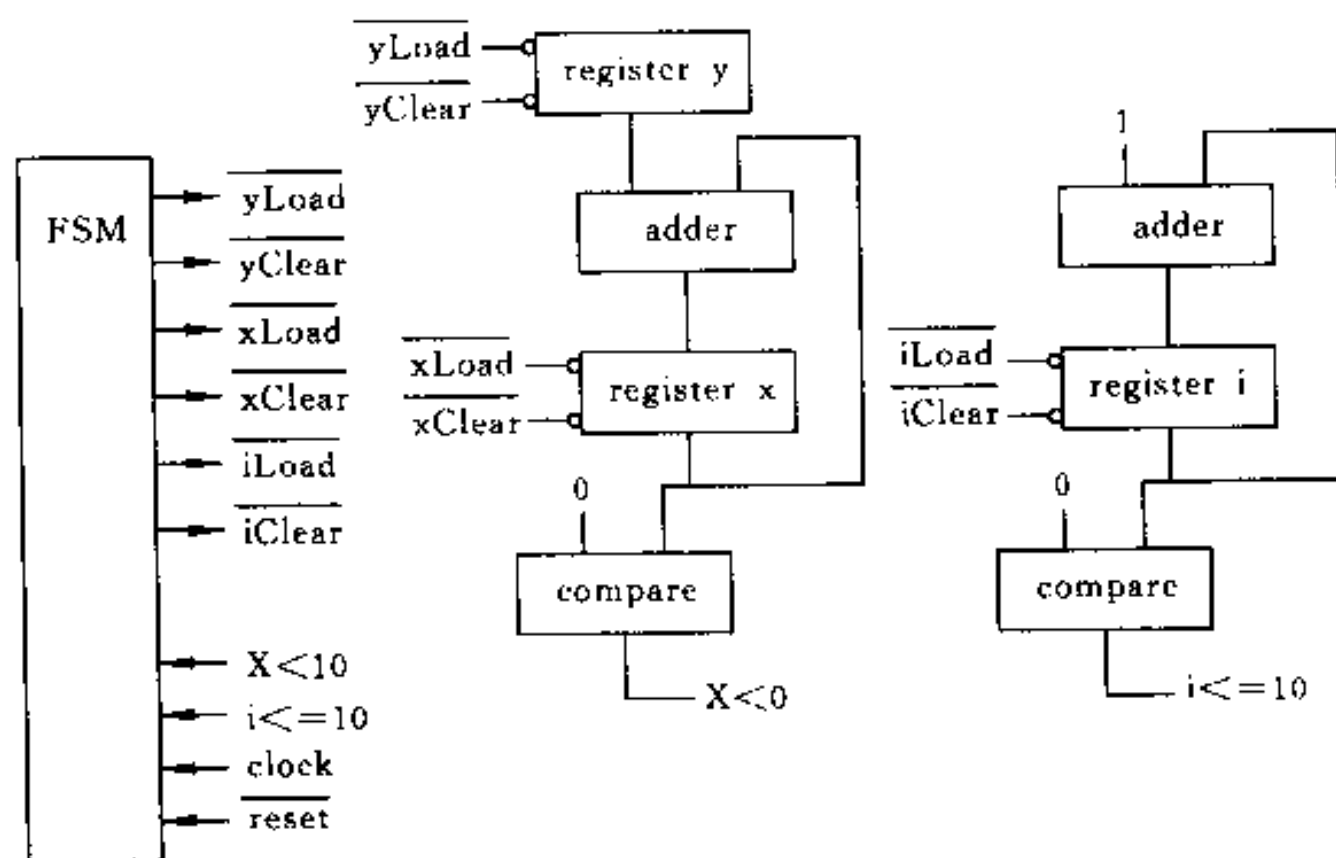


图 1.9 有限状态机和数据通道

左边所示的 FSM 通过一系列导致计算发生的状态来进行排序。FSM 的输出是  $yLoad$ 、 $yClear$ 、 $xLoad$ 、 $xClear$ 、 $iLoad$  和  $iClear$ 。它的输入是  $x<0$  和  $i\leq 10$ 。一个主时钟驱动 FSM 中的状态寄存器及数据通道寄存器。一个 reset 信号也连在上面。

### 1.5.3 数据通道功能模块的细节

数据通道由三个基本模块组成：寄存器、加法器和比较器。寄存器模块定义如例 1.14 所示。首先在 always 块中，我们注意到它与我们在时序电路看到的描述十分相似。寄存器是上升沿触发的，但没有异步复位。为了与数据通道定义的寄存器模块保持一致，它有两个控制点：clear 和 load。当这两个控制点有效时，将导致寄存器执行必要的功能。如果 clear 输入有效，在 clock 边沿它会装入 0。如果 load 有效，它会在 clock 边沿将输入 in 装入到寄存器 out 中。如果这两个信号都有效，那么寄存器将执行清除功能。

#### 例 1.14 寄存器模块

```
module register (out, in, clear, load, clock);
    parameter Width = 8;
    output [Width-1: 0] out;
    reg [Width-1: 0] out;
    input [Width-1: 0] in;
    input clear, load, clock;

    always @(posedge clock)
```

```

        if (~clear)
            out <= 0;
        else if (~load)
            out <= in;
    endmodule

```

这个示例介绍一个新语句：参数语句。参数定义一个名称具有一个数值；在这种情况下，width 的值是 8。这个名称在模块内是已知的，并可以用在任何语句中。这里我们看到它被用来为输出端寄存器 out 和输入端 in 定义向量的最左位编号。假定 width 定义为 8，最左边位数是 7（即  $8-1$ ），out 和 in 的位宽都是 8（即位 7 至 0）。参数的意义在于默认值可以在例示时被覆盖（override）。也就是说，这个模块定义可用来例示不同位宽的寄存器，随后我们会看到。

加法器模块显示在例 1.15 中。它被参数化为 8 位的默认位宽。但是，它也使用了新的行为结构：持续赋值语句。该赋值语句允许我们描述组合逻辑功能而不管它实际的结构化实现——也就是，这里没有带连线和端口连接的示例门。在电路模拟过程中，无论何时，只要逻辑表达式的一个值发生变化，就得计算等号右边逻辑表达式的结果，这个结果驱动输出 sum。持续赋值语句的另外一个形式是，它允许我们写出一个布尔表达式，但是需采用 Verilog 语言中更大的行为运算符集合。本例的赋值语句给出生成“加法器”功能的一种方式。输出 sum 被输入 a 和 b 的和赋值，a 和 b 用运算符“+”连接。在第 4 章会进一步讨论赋值语句。

### 例 1.15 加法器模块

```

module adder (sum, a, b);
    parameter          Width = 8;
    input      [Width-1:0]  a, b;
    output     [Width-1:0]  sum;

    assign sum = a + b;
endmodule

```

例 1.16 给出了 compareLT 和 compareLEQ 模块，又一次用到持续赋值语句。在 compareLT 模块中，a 和 b 比较，如果 a 比 b 小，那么 out 设置为真，否则设置为假。在计算中用来将 i 和 10 进行比较的模块 compareLEQ，与这个模块类似，只是用“<=”运算符代替“<”运算符。这两个模块的宽度都被参数化了。不要被第二个赋值语句，即

```
assign out = a <= b;
```

搞混。

这不是用非阻塞块赋值语句将  $b$  赋给  $a$ , 然后再用阻塞块赋值语句将  $a$  赋给  $out$ 。在一个语句中仅允许一个赋值。因此, 通过它们在语句中的位置, 我们可以知道第一个是一个赋值, 而第二个是一个小于或等于的比较。

#### 例 1.16 CompareLT 和 CompareLEQ 模块

```
module compareLT (out, a, b);           // compares  $a < b$ 
    parameter          Width = 8;
    input               [Width-1:0]  a, b;
    output              out;

    assign out = a < b;
endmodule

module compareLEQ(out, a, b);          // compares  $a \leq b$ 
    parameter          Width=8;
    input               [width-1:0]  a,b;
    output              out;

    assign out = a <= b;
endmodule
```

模块 adder、compareLEQ 和 compareLT 可以用前面 1.2 节中讨论过的 always 块的组合电路形式描述出来。这几个示例中用过的这两种形式是相等的。当用简单语句描述组合函数时, 一般采用持续赋值的方法。而更复杂的组合函数, 包括具有无关项规范的组合函数, 一般采用组合的 always 语句更易于描述。

参考: 持续赋值 4.4

#### 1.5.4 用连线将数据通道连在一起

现在建立一个模块来例示所有必需的 FSM 和数据通道模块, 并用连线将这些模块连在一起。这个模块, 如例 1.17 所示, 首先说明了把数据通道模块连在一起的几个 8 位连线, 后面是把控制线连接到 FSM 的几个 1 位连线。在连线定义的后面, 模块例示规定图 1.9 所示的互连。

#### 例 1.17 将 FSM 和数据通道结合在一起

```
module sillyComputation (yIn, y, x, ck, reset);
    parameter          Width = 8;
    input               ck, reset;
    input               [Width-1:0]  yIn;
```

```

output      [Width-1:0]  y, x;
wire        [Width-1:0]  i, addiOut, addxOut;
wire                                               yLoad, yClear, xLoad, xClear, iLoad, iClear;

register     #(Width)     I      (i, addiOut, iClear, iLoad, ck),
                                     Y      (y, yIn, yClear, yLoad, ck),
                                     X      (x, addxOut, xClear, xLoad, ck);

adder        #(Width)     addI      (addiOut, 1, i),
                                     addX      (addxOut, y, x);

compareLT    #(Width)     cmpX      (xLT0, x, 0);
compareLEQ   #(Width)     cmpl      (iLEQ10, i, 10);

fsm          ctl
(xLT0, iLEQ10, yLoad, yClear, xLoad, xClear, iLoad, iClear, ck, reset);
endmodule

```

注意这个模块也定义了一个 Width 参数,在连线定义中用到它,模块例示中也用到它。考虑一下例 1.17 中寄存器 I 的模块例示。

```
register  #(Width)  I  (i, addiOut, iClear, iLoad, ck),
```

这一行中第二项,“#(Width)”,是新出现的。这个值在模块例示中用它的参数来代替。因此,通过将模块 sillyComputation 中的参数 Width 改为,比如说 23,那么数据通道的所有模块例示将是 23 位宽。参数化模块允许我们在更多的地方可以重复使用类属模块定义,这使描述变得更容易。如果在模块例示语句中没有规定 #(Width),那么采用在模块 register 中规定的默认值 8。

### 1.5.5 FSM 说明

既然已经规定了数据通道,那么就需要有限状态机来激活(evoke)寄存器的顺序传输,必须满足原计算规定的条件。首先介绍这个系统的状态转换图,然后描述出实现它的 fsm 模块。

图 1.10 给出了状态转换图和规定的计算。用“...”标出的状态表示在我们所关心部分之前和之后的计算。每个状态表示在该状态期间将是有效值的 FSM 输出;所有其他信号都将是无效值。箭头指向后继状态;箭头旁边的条件表达式表示状态转换的条件。这个图表示的是一个莫尔(Moove)机,其中输出仅是当前状态的函数。最后,为了讨论方便,状态用 A 至 F 表示。



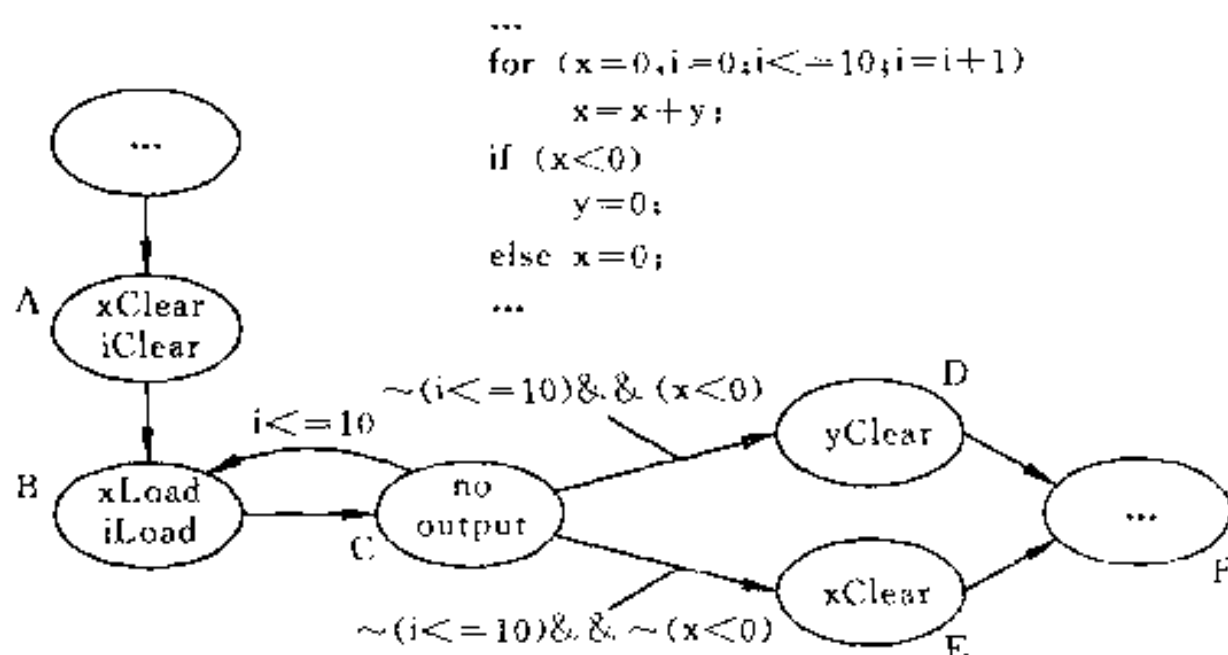


图 1.10 状态转换图

沿着计算和状态转换图,我们看到第一个动作是清除状态 A 中的两个状态寄存器  $x$  和  $i$ 。这意味着当机器在状态 A 时,  $xClear$  和  $iClear$  信号为有效值(低电平)。注意寄存器  $i$  和  $x$  直到时钟上升沿过后才变为 0,并且状态为 B。然后状态 B 发出  $x$  和  $i$  的装入信号。图 1.9 中的数据通道显示出实际上要装入的值分别是  $x+y$  和  $i+1$ 。因此,状态 B 执行循环体和循环更新。系统从状态 B 进入没有 FSM 输出信号的状态 C。但是,从状态 C 起,有 3 个可能的后继状态,取决于我们或者正处在循环中(回到状态 B),或者退出循环并进入条件语句的 then 部分(状态 D),或者退出循环进入条件语句的 else 部分(状态 E)。D 或 E 的下一个状态是状态 F,进行其他计算。

理解为什么系统的实现中需要状态 C 是很重要的。到底状态 C 的条件转换能不能从装入  $x$  和  $i$  的状态 B 中产生出来? 回答是否定的。图 1.11 中的时序图说明状态 A、B 和 C 之间的转换。在系统处在状态 B 期间,有限状态机的有效输出是  $xLoad$  和  $iLoad$ ,意思是寄存器  $x$  和  $i$  能够从它们的输入端加载。但是它们直到下一个时钟沿——将有限状态机转换到状态 C 的同一个时钟沿——才能加载。因此,  $i$  (执行循环的判断条件) 和  $x$  (执行 if-then-else 的判断条件) 的值直到系统处在状态 C 时对于比较判断才是有用的。在时序图中,可以看到由于  $i$  小于等于 10, C 的下一个状态是 B。

应注意到在系统的实现中,在进入循环之前不检查 for 循环的退出条件。但是,假定进入循环之前我们刚清除  $i$ ,检查它小于或等于 10 是不必要的。而且,对于不同的数据通道,状态 C 可能是不必要的。例如,将  $i$ 、 $x$  和各自的未来值的比较可以根据寄存器的输入值来进行。否则,进行比较的常数可能会改变。当然,这些任凭设计者自行处理。

现在考虑例 1.18 所示系统中有限状态机的 Verilog 模型。机器的输入是两个条件,  $x < 0$  和  $i \leq 10$ 。在 fsm 模块内部,分别称作 LT 和 LEQ。模块 fsm 也有一个 reset 输入和一个时钟(ck)输入。模块的输出是寄存器( $yLoad$ ,  $yClear$ ,  $xLoad$ ,  $xClear$ ,  $iLoad$ ,  $iClear$ )上的控制点。就像前面的 fsm 示例一样,有两个 always 模块:一个用于时序状态

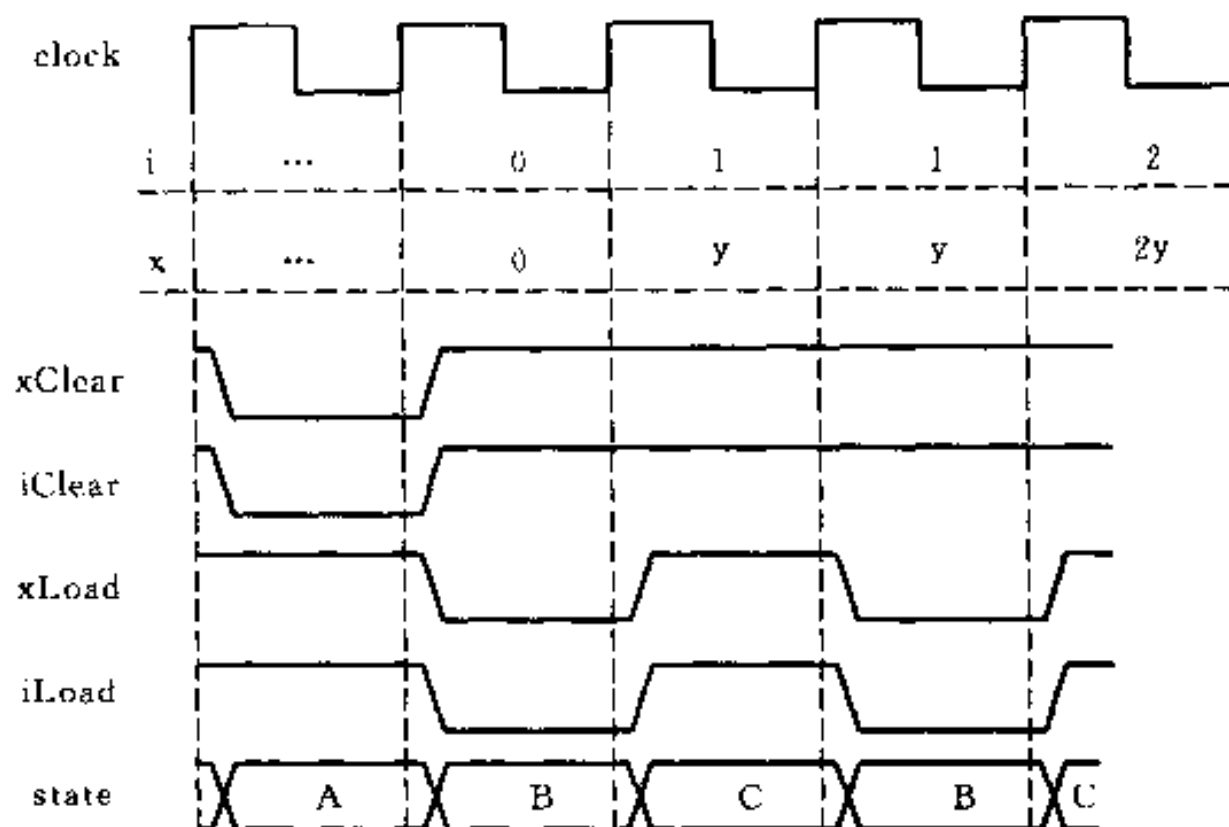


图 1.11 状态 A、B、C 的时序图

改变,而另一个用来实现下一个状态和输出组合逻辑。所有组合逻辑的输出都说明为寄存器。

状态机将只执行状态转换图中所示的状态,尽管在其他计算中还有许多状态。因此,状态寄存器(cState)的宽度选定为 3 位。此外,复位状态显示为状态 0,尽管在整个系统中它可能是其他某个状态。选定一个很简单的状态赋值,状态 A 编码为 0,B 编码为 1,等等。

第一个 always 块与前面的状态机示例很相似。如果 reset 是有效值,那么就进入复位状态;否则,在时钟上升沿,组合值 nState 在时钟上升沿装入 cState。

第二个 always 块执行下一个状态和输出组合逻辑。该组合逻辑的输入是当前状态(cState)和 fsm 的输入(LT 和 LEQ)。Always 块的主体根据 cState 的值来安排。一个 case 语句,实际上是一个多路分支,用来规定当给定 cState 的每个可能值时所要发生的动作。括号内表达式的值,与列在每行上的值进行比较。执行具有匹配值的那一行。

在 case 关键字后面的行规定一个数值,并跟着一个冒号。数字 3'b000 是 Verilog 语言对一个 3 位数的表示,此处以二进制规定 000,b 表示二进制——其他选择包括:h 表示十六进制,d 表示十进制。

如果当前状态变为状态 A,我们的编码 cState 的值是 0。结果,当这个改变发生时,always 块将会执行,并且执行 3'b000:右边的语句。这个语句说明除了 iClear 和 xClear 外所有的输出都是无效的(1),而下一个状态是 3'b001(即状态 B)。如果当前状态是 B,那么执行第二个 case 项(3'b001),将 iLoad 和 xLoad 设置为有效值,而其他信号设置为无效值。状态 B 的下一个状态是 C,编码为 3'b010。状态 C 给出了一个更复杂的后继状态

计算;3 个 if 语句规定状态 C 可能的后继状态以及每个状态选择的条件。

最后一个 case 项规定默认情况,这个语句在 cState 的值不能与其他项目匹配时才执行。为了能够模拟,如果到达了一个非法状态,可能需要一个 \$display 语句来打印出一个错误警告。\$display 在模拟期间在屏幕上打印一条信息,动作非常像编程语言中的打印语句。这会显示信息“Oops,unknown state:%b”,用 cState 的二进制表示代替 %b。

为了使这个 always 块具有可综合的组合功能,默认情况是需要的。考虑一下,如果我们没有默认语句,一旦 cState 的值是个规定值以外的其他值,那会发生什么情况。在此情况下,case 语句会执行,但是没有规定的动作执行。因此,不会赋值给输出。这就违反了组合逻辑综合规则,该规则声明 always 块的每个可能路径必须对每个组合输出进行赋值。因此,尽管模拟中调试一个描述时具有默认项是任选的,但是对于综合组合电路的 always 块而言,默认项是必需的。当然,如果已经规定了所有已知值情况或者在 case 语句之前 cState 已经赋值,对于综合来说就不需要一个默认项。

现在考虑整个 FSM-数据通道系统是怎样一起工作的。假定当前状态是状态 C,i 和 x 的值分别是 1 和 y,如图 1.11 的时序图所示。再假定使系统进入状态 C 的时钟沿刚发生,cState 已装入值 3'b010(状态 C 的编码),不仅 cState 已经改变,而且寄存器 i 和 x 也被加载,作为状态 B 的结果。

在我们的描述中,几个 always 块正在等待 cState、x 和 i 的改变。这些块包括 fsm 的组合 always 块、加法器模块及比较模块。由于 cState、x 和 i 的改变,这些 always 块现在可以执行。模拟器将以任意顺序执行它们。实际上,模拟器可以对其中一些块执行多次。(考虑这样一种情况:fsm 的组合 always 块首先执行。然后执行比较模块以后,它必须再执行一次)。最后,将会为比较器的输出产生新值。fsm 模块中 LT 和 LEQ 的改变将引起组合 always 块的执行,为 nState 产生一个新值。在下一个时钟上升沿,这个值将装入 cState 并进入另一个状态。

#### 例 1.18 用于数据通道的 FSM

```
module fsm (LT, LEQ, yLoad, yClear, xLoad, xClear, iLoad, iClear, ck, reset);
input      LT, LEQ, ck, reset;
output     yLoad, yClear, xLoad, xClear, iLoad, iClear;
reg        yLoad, yClear, xLoad, xClear, iLoad, iClear;
reg        [2:0]    cState, nState;

always @(posedge ck or negedge reset)
    if (~reset)
        cState <= 0;
    else
        cState <= nState;
```

```

always @(cState or LT or LEQ)
  case (cState)
    3'b000 : begin                                // state A
      yLoad = 1; yClear = 1; xLoad = 1; xClear = 0;
      iLoad = 1; iClear = 0; nState = 3'b001;
    end
    3'b001 : begin                                // state B
      yLoad = 1; yClear = 1; xLoad = 0; xClear = 1;
      iLoad = 0; iClear = 1; nState = 3'b010;
    end
    3'b010 : begin                                // state C
      yLoad = 1; yClear = 1; xLoad = 1; xClear = 1;
      iLoad = 1; iClear = 1;
      if (LEQ) nState = 3'b001;
      if (~LEQ & LT) nState = 3'b011;
      if (~LEQ & ~LT) nState = 3'b100;
    end
    3'b011 : begin                                // state D
      yLoad = 1; yClear = 0; xLoad = 1; xClear = 1;
      iLoad = 1; iClear = 1; nState = 3'b101;
    end
    3'b100 : begin                                // state E
      yLoad = 1; yClear = 1; xLoad = 1; xClear = 0;
      iLoad = 1; iClear = 1; nState = 3'b101;
    end
    default : begin                                // required to satisfy combinational synthesis rules
      yLoad = 1; yClear = 1; xLoad = 1; xClear = 1;
      iLoad = 1; iClear = 1; nState = 3'b000;
      $display ("Oops, unknown state: %b", cState);
    end
  endcase
endmodule

```

参考：case 2.4；数字表示 B.3

辅导：见附录 A.6 中的辅导问题

## 1.6 周期精确的行为描述

由于我们已经学习了这个辅导，Verilog 描述抽象层次的概念增加了。我们再上升到

一个更高的层次：**周期精确** (cycle accurate)，有时称为**调度行为** (scheduled behavior)。在这个层次，我们用一个时钟周期一个时钟周期的方式来描述系统，规定每个状态中将要发生的行为。使用术语 cycle accurate 的原因是系统中的值只有在系统状态改变的时候——在一个时钟沿处——才规定是有效的。可以用模拟结果描述，或者通过用行为综合工具可以综合结果描述，产生 1.5 节中所述的 FSM-D 描述风格。

### 1.6.1 规范方法

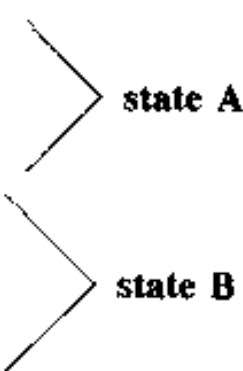
调度行为用 always 块来规定，用“@(posedge clock);”语句将规范分成时钟周期或状态。例 1.19 例示一个 1.5 节中简单计算的预定行为描述。模块的端口是寄存器 x、y 和 clock。寄存器 i 是一个循环计数器，仅在模块内部使用。

在讨论周期精确的特性之前，先解释 always 块和块内用到的新语句——while 语句。while 语句与编程语言中用到的 while 语句类似。这个语句计算 while 条件 (“i ≤ 10”)。如果条件为真，则进入循环执行循环体，此时我们将 y 和 x 相加并将 i 加 1。然后我们继续计算 while 条件并执行循环体，只要条件仍然为真。如果条件为假，则不执行循环体。在本例中，假定第一次计算 while 条件时 i 小于或等于 10，则进入循环（但是，当第一次计算时如果 while 条件为假，则不执行循环体并且控制传送到 while 后面的下一条语句）。因此，只有当条件为真时 while 语句的循环体才会执行。

采用这种描述风格，一个“@(posedge clock);”语句后面跟随着行为语句，然后是另一个“@(posedge clock);”语句。我们将这个“@(posedge clock);”语句称作**时钟事件**。在两个时钟事件之间的语句构成一个状态。时钟事件语句不需要成对出现；如果在一个循环体内仅有一个时钟事件语句，那么循环在一个状态内执行，下一个时钟事件实际上是它自身。

#### 例 1.19 采用预定行为方法的描述

```
module simpleTutorial (clock, y, x);  
    input                clock;  
    output [7:0]         x, y;  
    reg [7:0]            x, y, i;  
  
    always begin  
        @(posedge clock) x <= 0;  
        i = 0;  
        while (i <= 10) begin  
            @(posedge clock);  
            x <= x + y;  
            i = i + 1;  
        end  
    end
```



The diagram shows two clock events, represented by the text “@(posedge clock);”, each with a right-pointing arrow. From the first arrow, a line branches to the right, labeled “state A”. From the second arrow, a line branches to the right, labeled “state B”.

```

end
@(posedge clock);
if (x < 0)
    y <= 0;
else x <= 0;
end
endmodule

```

state C

在例 1.19 中,考虑状态 C、最后一个时钟事件和跟随其后的语句,如图 1.12 所示。跟随在时钟事件后面的语句是一个 if-else 语句。假定 always 连续循环,下一个时钟事件在 always 块的顶部,开始执行状态 A。因此,这些语句给出一个状态的规范。在这个状态, y 或 x 赋值为 0,取决于 x 是否小于 0。与描述相对应的状态显示在图的右边。采用米利机(Mealy)概念(输出是输入和当前状态的函数),表明如果 x 小于 0,那么我们将沿着上边的弧到状态 A 并将寄存器装入 0(使用非阻塞赋值),而下边的弧给出相反的条件(当 x 设置为 0 时)。在模拟中,当等待在 if 语句之前的时钟事件时,则执行状态 C。

```

@(posedge clock);
if (x < 0)
    y <= 0;
else x <= 0;

```

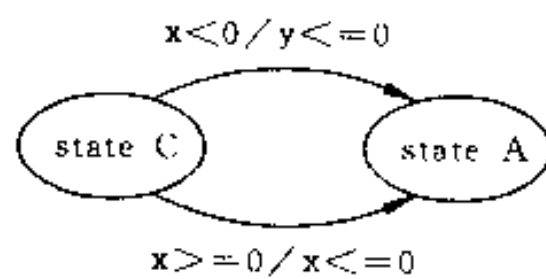


图 1.12 状态 C

整个状态转换图如图 1.13 所示。状态 A 将 x 和 i 初始化为 0,并进入循环。状态 B 是循环体,状态 C 是 if-else,如上面描述所示。状态 B 是具有特殊意义的,因为它给出两种可能的后继状态。状态的开始是循环体中的时钟事件语句。但是,下一个时钟事件可以是执行循环体并处在循环体内的语句(即同一个语句),也可以是执行循环体并退到 while 语句后面的那一个语句。这解释了状态 B 的两个可能的后继状态。

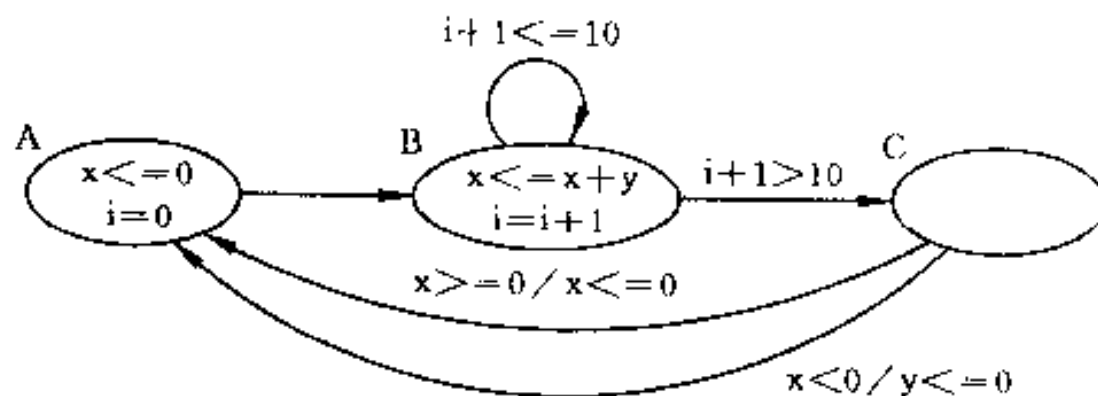


图 1.13 周期精确行为的状态转换图

## 1.6.2 几点注释

关于这个示例和调度行为或周期精确的描述风格,有几点注释。

当需要一个周期一个周期地规定系统的行为,但又不太关心设计的实际数据通道时,使用这种描述风格。我们已经规定简单计算以及产生新值的那些状态。但是没有任意规定数据通道;这留给后面的设计阶段来作。

在这个规范中混合使用阻塞赋值(“=”)和非阻塞赋值(“<=”)。寄存器 x 和 y 使用非阻塞赋值,而寄存器 x 和 y 在 always 块的外面说明。这有效地将寄存器的加载同步到规定的时钟沿上。对于只使用在一个 always 块内的寄存器,例如寄存器 i,这是不必要的。请记住当你使用非阻塞赋值语句进行赋值时,这个数值直到时钟沿过后才可以被寄存器的名称所引用,也就是,在描述的下一行是不可用的!而且,对于一个状态内的任何一个寄存器只进行一次无条件的非阻塞赋值。但是,可以使用阻塞赋值来计算中间值和仅使用在 always 块内的数值。当然,这些数值在描述的下一行是立即可用的。在本例中,使用在循环结束比较判定中的 i 是在循环中计算的那个 i,因为我们使用了一个阻塞赋值。

最后,实现简单计算的两种方法的状态转换图(如图 1.10 和图 1.13 所示)是不同的。这是有意这样做的,并不是哪种规范风格有什么局限性。

参考:周期精确 7.2

辅导:见附录 A.7 中的辅导问题

## 1.7 赋值语句的总结

当开发数字系统的 Verilog 模型时,一个重要特点是抓住新值(输出)是在什么时间如何产生的。这个辅导给出产生新值的四种不同方法:门基元,持续赋值,过程赋值(“=”)和非阻塞赋值(“<=”)。在 Verilog 语言中,这四种方法分成两大类。这两大类在时间上产生输出的方式不同。因此,我们将它们称为时序模型。这些模型是门级时序模型和过程时序模型。

门基元(例如例 1.3)和持续赋值(例如例 1.15)例示了门级时序模型。当写出一个简单的 AND 表达式时,我们既可以写成

```
and (a,b,c);
```

又可以写成

```
assign a=b&c;
```

这两个语句是等价的;两者都执行 b 和 c 的按位与运算,并将结果赋给 a。认识这些语句



的方式是无论什么时候只要任何一个输入(b 或 c)发生变化,都要重新计算输出 a 的值。此外,在这两个语句中,a 是一个线网。

过程时序模型使用 initial 块和 always 块中的过程语句来产生新值。正规的过程赋值(“=”)如例 1.5 中所示,非阻塞过程赋值(“<=”)如例 1.7 中所示。always 块:

```
always @(posedge clock)
```

```
    Q<=D;
```

有两个输入(clock 和 D)和一个输出 Q。与门级时序模型相对应,过程赋值对它的所有输入不敏感;只在特定的时刻对特定的输入敏感。这里,always 块只对时钟上升沿的变化敏感。当时钟上升沿出现时,Q 被 D 的值更新。但是,如果 always 块的输入端另一个输入 D 改变了,Q 的值不更新。过程模型只对它们显式等待的输入敏感。此外,所有过程赋值语句的左侧都是寄存器。

只有当控制被传送到过程赋值语句的时候才将值装入寄存器或存储器。控制以顺序方式传送给过程赋值语句,从一条语句到另一条语句。在这种方式下,过程赋值功能与一般的软件编程语言的赋值语句相似。但是,控制流可能被事件(@)语句(及我们在后面将要看到的 wait 语句和 # 延迟语句)所中断,并且只有当事件发生时才被重新激活。

过程赋值“=”和“<=”当左边更新时,还能进一步分类。正如 1.3.3 节所讨论的那样,“=”立即更新它左边的值,所以新值可以使用在下一条过程语句中。相反,“<=”只有当整个设计中等待同一个边沿的所有“<=”语句的右边计算完毕后才更新它左边的值。因此,左边的新值在下一条过程语句中是不能使用的。这就导致了下面的异常描述:

```
@(posedge clock)           // somewhere in an evil always block
```

```
m=3;
```

```
n=75;
```

```
n<=m;
```

```
r=n;
```

问题是什么值赋给了 r? 答案是 75。虽然第三条语句把 n 的值改为 3,但是这个左侧的值并不立即更新。实际上 always 块不停下来(即阻塞)更新 n;而是继续执行(非阻塞),使用时钟变化之前的 n 值。最终,n 会被更新为 3,但只有在所有其他非阻塞赋值的右侧计算完毕之后才更新。最好不要写出像上面这样一个不好的模型;它们很难阅读。而且,它们也不能被综合工具接受,所以它们的使用受到限制。只有描述边沿敏感系统中的并发传输时,才使用非阻塞赋值。

本质上讲,这两种时序模型与 Verilog 语言的两种基本数据类型(线网和寄存器)有紧密联系。持续赋值和基元门只能驱动线网,而过程赋值只能用于寄存器(和存储器)。

参考:过程赋值 2.1;连续赋值 4.4;时序模型 5.1



## 1.8 小结

这个简介例示了 Verilog 语言的基本性能,这些重要的性能是:

- 能够将设计划分为模块,而模块可进一步划分直到设计可以用基本的逻辑单元来规定。结构的模块化使设计人员可以通过有名的大型工程设计的分治方法来控制设计的复杂度。
- 能够根据设计的抽象行为或者根据实际逻辑结构来描述设计。行为描述使得早期的设计活动能够集中在功能上。当行为确定后,它就变成了设计几种可以选择的结构化实现的规范,这可以通过使用综合工具来完成。
- 能够使并发系统同步。系统的并发部分共享数据,必须互相同步,以便在并行部件间正确地传递信息。我们例示了系统是如何对信号边沿(即一个时钟)进行同步的。

本章辅导的主要目的是对 Verilog 语言进行一个快速介绍。因此,许多细节都省略了。目的是使读者对 Verilog 语言有一个感性认识,方法是给出并描述一些示例,例示语言的主要特点和应用。后面几章的目的是深度覆盖语言,仍然采用面向示例的方法来介绍该语言。我们在本章和全书其他章节中的目标并不是仅仅像一个正规文法规范那样来介绍 Verilog 语言。但是,由于意识到我们给出的示例并不能说明所有的语言文法,所以将要开始介绍语言的一些形式化文法规范。这个文法规范对第一次接触 Verilog 的读者来说可能没用。但是,这对于将本书当作参考书的读者和进行设计描述的读者来说是极其珍贵;附录 G 中有完整的形式化文法规范。

书中的其他部分深度例示了 Verilog 语言的文法和语义。

## 1.9 练习

更多的练习见附录 A。

- 1.1 用持续赋值语句重新描述例 1.4 中的 eSeg 模块。
- 1.2 写出一个两位全加器(包括 carry-in 和 carry-out 端口)的三种不同的描述,其中一个描述使用门级模型,另一个使用持续赋值语句,第三个使用组合的 always。
- 1.3 改变例 1.9 中的时钟发生器 m555,使得时钟周期保持不变,但低脉冲宽度是 40,高脉冲宽度是 60。
- 1.4 写出一个二相位时钟发生器,相位二被相位一偏移 1/4 周期。
- 1.5 保持相同的输出时序,用门基元代替例 1.9 中时钟发生器 m555 中的初始语句和 always 语句。

1.6 写出一个串行加法器的描述,模块定义是:

```
module serialAdder(clock,a,b,sum,start);
    input  clock,a,b,start;
    output sum;
    ...
endmodule
```

A. 各个二进制位以低阶优先的顺序进入模块。进入低阶二进制位的进位假定为 0。在时钟下降沿之前所有输入都是有效的。a 和 b 是要进行相加的两个二进制位,sum 是结果。如果 start 是 1,那么这两个二进制位是第一个要相加的字的二进制位。就在时钟下降沿之后,sum 正好在时钟沿之前基于 a 和 b(和前面二进制位的进位)的新值来更新。

B. 建立一个测试模块验证串行加法器的正确性。

1.7 设有一个 Verilog 模块,除了寄存器、输入和输出的说明外,模块是完整的。假定 a, b, c 是 8 位的“元件”,而其余的是 1 位的,请注意必须加到输入输出表中。不要加任何其他赋值——只有输入、输出和寄存器的说明。

```
module sillyMe(a,b,c,q,...);
// oops, forgot the declarations!
    initial
        q = 1'b0;

    always
        begin
            @(posedge y)
                #10 a=b+c;
                q = ~q;
            end
        nand #10 (y,q,r);
endmodule
```

## 第2章 行为建模

现在开始更深入地讨论数字系统行为建模的结构。它们分为两组。第一组是主要部分,它们类似于编程语言中的语句,如 if-then-else, loops 等。第二组是对数字硬件并行特征进行建模的语句,下一章将讨论它们。

### 2.1 进程模型

行为模型的本质是进程。一个进程可以被看作一个独立的运行单元,它可能十分简单,只涉及一个循环动作,也可能非常复杂,像一个软件程序。它可以作为一个时序状态机,一个微代码控制器,一个寄存器异步清零器,或者一个组合电路。关键在于我们将数字系统的行为看作单独的,但相互通信的进程的集合。它们真正的实现与描述的上下文(正处理的层次)以及实现的时空约束有关。

描述进程的基本 Verilog 语句是 always 结构:

```
always_construct  
::= always statement
```

always 结构反复执行其中语句,永远不退出或者终止执行。一个行为模型可以包含一个或多个 always 语句。(一个不包含 always 语句的模块纯粹是一个层次结构的描述,即子模块示例和它们的互连。)

initial 结构类似于 always 语句,不过它只执行一次。

```
initial_construct  
::= initial statement
```

initial 结构提供一种在实际描述开始模拟之前初始化输入波形和模拟变量的方式。一旦 initial 结构中的语句执行完毕,它不会重复再执行;并且它不再活跃。

这种语言中有很多种类型的语句。它们中的一些在前面的示例中已经出现过,比如“if”,“while”和过程赋值语句(“out = ~currentState[1] & currentState[0];”)。这些语句和大多数其他类型的语句将会在下两章见到。

当在一个 always 或 initial 块中使用这些语句为系统建模时,必须清楚这些语句的执行模型。这些语句按照描述中的顺序执行。使用阻塞赋值符号(“=”)的赋值语句立即生效,并且下一条语句就可以使用写到符号“=”左侧的值。当执行一个事件语句(“@”)、延

迟语句(“#”)或者其表达式的值为 FALSE 的等待语句(后面将会见到)时, always 或 initial 语句的执行将被挂起,直到出现下面的某种情况:发生一个事件,经过延迟中的时间单元数,或者等待语句的表达式的值变为真。那时,将继续执行 always 或 initial 中的语句。

即使一个 always 或 initial 块中的语句顺序执行,其他 always 或 initial 块中的语句也可能与当前进程中的语句交叉执行。当一个 always 或 initial 块在等待(用@, # 或 wait 表示)继续执行时,其他 always 或 initial 块、门基元和持续赋值语句可以执行。结果,建立了并发和交叉行为的模型。

与门基元和持续赋值语句不同,行为模型不会因为某个输入发生变化而执行。确切地说,仅当正在等待的上述三个条件之一发生时,它们才会执行。行为模型与 5.1 节中讨论的过程时序模型一致。

模拟开始时,执行所有的 initial 和 always 语句,直到它们因为一个事件、延迟或等待而被挂起。在这里,initial 或 always 中设置的寄存器值可能激活一个门的输入,或者模拟时间前进到下一个事件,这些都可能唤醒一个或多个挂起的进程。当某一特定时刻有多个进程要执行时,它们执行的顺序是不确定的。一定要注意确保按合适的顺序为寄存器和连线赋值。

总之,initial 和 always 语句是描述并发性的基本结构。运用这些语句时,我们应该注意交叉执行的并发活动进程。虽然可以把行为描述混合写入 always 和 initial 语句,但是,更合适的做法是在 always 语句中描述硬件的行为,在 initial 语句中描述模拟的初始化工作。

参考:对照持续赋值 4.4 节;对照门级建模 4.1 节;交叉 5.3 节;过程时序模型 5.1 节

## 2.2 If-Then-Else

顺序行为描述中的条件语句用于改变控制流程。if 语句和它的变化形式是条件语句的常见形式。例 2.1 是一个除法模块的行为模型,它表现出几个新特征,包括两种 if 语句,有 else 子句的和没有 else 子句的。

该 divide 模块根据两个输入(dvInput 和 ddInput)决定输出 quotient,它使用重复移位减的算法。首先,定义四个文本宏。定义一个宏的名字并且为它赋一个常量文本值,编译器就能够接受它。这个名字就可以用在描述中;编译时,这个文本值将被替换。通常的形式是:

```
`define A alpha
```

不管何时编译这个描述,所有出现“A”的地方都将被 alpha 所替代。注意所有的宏左侧都要求有左单引号(“’”)。例 2.1 给出了一种通过宏将常数的值送入描述的方式。

### 例 2.1 除法模块

```
`define DvLen 15
`define DdLen 31
`define QLen 15
`define HiDdMin 16

module divide (ddInput,dvInput,quotient,go,done);
    input    [`DdLen:0]          ddInput,dvInput;
    output   [`QLen:0]           quotient;
    input                               go;
    output                               done;

    reg      [`DdLen:0]          dividend;
    reg      [`QLen:0]           quotient;
    reg      [`DvLen:0]          divisor;
    reg                               done,negDivisor,negDividend;

    always begin
        done=0;
        wait (go);
        divisor=dvInput;
        dividend=ddInput;
        quotient=0;
        if (divisor) begin
            negDivisor=divisor[`DvLen];
            if (negDivisor)          divisor=-divisor;
            negDividend=dividend[`DdLen];
            if (negDividend)          dividend=-dividend;
            repeat (DvLen+1) begin
                quotient=quotient<<1;
                dividend=dividend<<1;
                dividend[`DdLen:HiDdMin]=
                    dividend[`DdLen:HiDdMin]-divisor;
                if (! dividend[`DdLen]) quotient=quotient+1;
            else
```

```

        dividend[DdLen:HiDdMin]=
            dividend[DdLen:HiDdMin]+divisor;
    end
    if (negDivisor!=negDividend)    quotient=-quotient;
end
done=1;
wait (~go);
end
endmodule

```

除法从对输出 done 清零并等待 go 变为 TRUE 开始。这两个信号是握手信号,它们允许 divide 模块与其他模块通信、同步。done 表示除法模块已经完成了除法运算,并且将结果保存在 quotient 中。既然开始时没有计算出商值,done 被设置为 FALSE(或 0)。然后,等待输入 go 变为 1(或 TRUE),它意味着输入 dvInput 和 ddInput 有效。当 go 变为 TRUE 时,dvInput 和 ddInput 被分别赋值给 divisor 和 dividend。

等待语句等待外部条件变为 TRUE。当它执行(即等待)时,如果圆括号中的条件为 TRUE,就继续执行。然而,如果条件为 FALSE,always 块停止执行并等待条件变为 TRUE。执行将从 wait 语句之后的语句继续开始。wait 语句将在 3.3 节继续讨论。

第一个 if 语句的示例是测试这个语句中的除数是否为零:

```

if (divisor)
    begin
        // ... statements
    end

```

该例展示 if 语句的基本形式。if 后跟圆括号括起来的表达式;零值表达式记为 FALSE,任何其他非零值记为 TRUE。与不确定值(x)或高阻抗值(z)相比较可能产生一个不确定的或高阻抗的结果。这样一个结果被解释为 FALSE。在这个示例中,要测试 divisor。如果它不为零,就继续下面正常的除法运算。if 语句后的 begin-end 块表示所有被包含其中的语句被看作与这个 if 语句相对应的 then 语句部分。

更形式化的说明是:

```

statement
    ::= conditional_ statement
    | ...

conditional_ statement
    ::= if (expression) statement_ or_ null [else statement_ or_ null]

```

继续分析这个除法算法。它继续计算每个输入的绝对值并且保存它们的符号。需要特别说明的是,语句

```
negDivisor=divisor[DvLen];  
if (negDivisor)  
    divisor=-divisor;
```

首先把除数的第 DvLen 位(即第 15 位)的值赋值给 negDivisor。如果这位是 1,意味着该二进制补码表达式的值为负,所以执行 then 语句部分,divisor 取反,使它的值变为正值。注意,既然这个 if 语句没有 begin-end 块,then 语句就是 if 语句后面的第一条语句(直到分号为止)。

这个语句也说明一个位选操作。位选操作指定向量中的某一位参与运算。也可以说明位的范围,通过冒号分隔的位号表达式来说明被选择的位的范围。这叫作域选操作。位选或域选操作以如下所示的表达式形式表示。

```
primary  
    :: = identifier [expression]  
       | identifier [msb_constant_expression;lsb_constant_expression]  
       | ...
```

在这个形式化语法中,primary 是表达式的一种定义。primary 的第一个定义是位选择,第二个定义是域选择。位选择和域选择的索引号可以是正数,也可以是负数。

初始化中判定了最终的算术符号之后,repeat 语句将把 begin-end 块中的语句执行 16 次。每次,quotient 和 dividend 都要左移一位,用操作符<<表示。接着,dividend 的高位部分减去 divisor。如果结果为正,quotient 加 1;如果结果为负(最高位为 1),执行 if 条件语句的 else 部分,把 divisor 加回到 dividend 的高位部分。

更准确地说,如果符号位为 1,那么结果为负。这个非零值(即符号位)将被 if 语句解释为 TRUE。然而,操作符!对值取补,结果 if 表达式的值为 FALSE。这样,如果 dividend 为负,将执行 else 部分。

最后,如果操作数的符号不同,则将 quotient 取反。计算出 quotient 之后,done 被赋值为 1,通知别的模块已经可以读取这个输出值了。

本示例展示了将要讨论的这种语言的一些其他方面。

向量线网和寄存器都遵循算术的模  $2^n$  规则,这里  $n$  是向量的位数。实际上,这种语言把数字看作无符号量。如果用 \$display 或 \$monitor 语句打印这种值,它们将被解释为无符号值。然而,这并不能妨碍我们书写那些采用二进制补码表示法的硬件描述,算术的模  $2^n$  规则仍然有效。事实上,该语言中提供的取反能正确地实现操作。

常用于条件表达式中的关系操作符被列在附录 B 中。它们包括>(大于),>=(大

于等于), == (等于) 和 != (不等于)。在出现了不确定和高阻抗值的情况下, 比较操作的结果被模拟器看作 FALSE。然而, 可以使用 case 等于操作符 (===) 和 不等于操作符 (!==) 来说明每个不确定和高阻抗位将参与比较。也就是说, 四值逻辑比较操作要比较每一位, 包括不确定位和高阻抗位。

因此, 如果执行下面这条语句,

```
if (4'b110z===4'b110z)
    then_statement;
```

将执行 if 语句的 then 部分。然而, 如果执行语句

```
if (4'b110z==4'b110z)
    then_statement;
```

将不会执行 if 语句的 then 部分。

条件表达式比上面出现的这些独立表达式更复杂。逻辑表达式可以用下例中所示的 && (与), || (或) 和 ! (非) 等逻辑操作符来连接:

```
if ((a>b)&&((c>=d)|| (e==f)))
    then_statement
```

本例中, then 语句仅当 a 大于 b, 并且或者 c 大于等于 d, 或者 e 等于 f 时才会执行。

参考: 位选择, 域选择 E. 1; \$display F. 1; \$monitor F. 2; Verilog 操作符 C

### 2.2.1 else 如何与 if 语句配对

例 2.1 也例示出与 if 语句配对的 else 子句的用法。else 子句是可选项, 如果它存在, 它与最近的不完整的 if 语句配对。形式化说明如下:

```
conditional_statement
    ::= if (expression) statement_or_null [else statement_or_null]
    | ...
```

从这个示例中可以找到下面的语句:

```
if (! dividend['DdLen'])
    quotient=quotient+1;
else
    dividend['DdLen':'HiDdMin']=
        dividend['DdLen':'HiDdMin']+divisor;
```

这里, 如果减去 divisor 以后, dividend 是一个正数, quotient 的最低位就设为 1。否



则,把 divisor 加回到 dividend 的高位部分。

正像大多数的过程语言一样,在涉及多个 if 语句的地方,要注意说明 else 子句。看下面的示例:

```
if (expressionA)
    if (expressionB)
        a=a+b;
    else
        q=r+s;
```

该示例中有两个嵌套的 if 语句和一个 else 语句。通常,语言把 else 与最近的 if 语句配对。上例中,如果 expressionA 和 expressionB 都为 TRUE,就为 a 赋新值。如果 expressionA 为 TRUE 并且 expressionB 为 FALSE,则为 q 赋新值。即 else 与第二个 if 语句配对。

下面是另一种描述,它将产生不同的结果:

```
if (expressionA)
    begin
        if (expressionB)
            a=a+b;
    end
else
    q=r+s;
```

这个示例中,第一个 if 语句中的 begin-end 块使 else 与第一个 if 语句配对,而不是第二个。当对 else 与谁配对感到迷惑时,可以使用 begin-end,使结构更清晰。

## 2.2.2 条件操作符

如果要从两个值中选一个来赋值,可以用条件操作符(?:)。例如,例 2.1 中判断 quotient 最终符号的语句可以写成如下的等价形式:

```
quotient = (negDivisor != negDividend) ? -quotient : quotient;
```

该操作符这样工作:首先,计算圆括号中的条件表达式。如果它的值为 TRUE(或非零),则语句右侧的值就是紧跟在问号后面的值。如果它的值为 FALSE,则是紧跟在冒号后面的值。其结果是两个值中的一个赋给 quotient。本例中,如果符号不等,则 quotient 取反。否则,quotient 保持原值不变。例 2.1 中描述了使用二进制补码系统的硬件,它具有这个事实,即 Verilog 的取反操作实现了二进制补码取反操作。

条件操作符的一般形式是:

```

expression
    ::= expression ? expression : expression
    | ...

```

如果第一个 expression 为真,则条件操作的值是第二个 expression;否则,它的值是第三个 expression。这个操作符是右结合的。

if-then-else 与条件操作符之间有个重要的区别。作为一个操作符,条件操作符可以出现在一个表达式中,这个表达式或者是过程赋值语句的一部分,或者是持续赋值语句的一部分。if-then-else 结构是语句,它只能出现在 initial 或 always 语句体中,或一个任务或函数中。因此,if-then-else 只能用在行为建模中,然而,条件操作符既能用在行为级建模中,又能用在门级结构建模中。

参考: if-then-else 2.2; 与多分支语句比较 2.4

## 2.3 循环语句

重复的顺序行为用循环语句来描述。有四种循环语句,包括 repeat, for, while 和 forever 循环。

### 2.3.1 四种基本循环语句

例 2.2 是从例 2.1 摘录的,它例示了怎样使用 repeat 循环。这种形式的循环只在关键字 repeat 后面的圆括号中给出一个循环数。

**例 2.2** 例 2.1 的摘录

```

repeat (DdLen+1)
begin
    quotient=quotient<<1;
    dividend=dividend<<1;
    dividend['DdLen':'HiDdMin']=
        dividend['DdLen':'HiDdMin']-divisor;
    if(! dividend['DdLen'])
        quotient=quotient+1;
    else
        dividend['DdLen':'HiDdMin']=
            dividend['DdLen':'HiDdMin']+divisor;
end

```

在循环开始执行前,已经确定了循环数表达式的值。然后,循环执行给定的次数。循环数表达式只在循环开始处计算一次值,所以,不可能通过改变循环数变量的值来达到退

出循环执行的目的。disable 语句可以用来提前退出循环,后面将讨论它。

repeat 语句的一般形式是:

```
statement
    ::= loop_statement
    | ...
loop_statement
    ::= repeat(expression) statement
    | ...
```

例 2.2 中的循环可以写成 for 循环的形式:

```
for(i=16;i;i=i-1)
    begin
        ... // 移位减语句
    end
```

这里,必须指定一个寄存器来保存循环数。for 循环在函数中与 C 语言的 for 循环很相似。从本质上说,这个 for 循环把 i 初始化为 16,并且,当 i 不为 0 时,都要先执行语句,然后 i 减 1。

for 循环语句的一般形式是:

```
loop_statement
    ::= for (reg_assignment; expression; reg_assignment) statement
    | ...
```

需要特别说明的是,第一条赋值语句在循环开始时执行一次。在循环体执行之前要计算 expression 的值,判断是否停留在循环中。当 expression 为真时,执行循环。循环体执行之后,下一次循环结束条件检查之前,再执行第二条赋值语句。statement 是循环体。for 循环和 repeat 循环之间的区别是:repeat 只是一种指定固定循环次数的方式。for 循环要灵活得多,它赋予了访问和更新循环变量的能力,从而控制循环结束条件。

正像 C 语言中一样,上面的 for 语句可以写成如下的 while 语句的形式:

```
i=16;
while (i)
    begin
        ... // 移位减语句
        i=i-1;
    end
```

while 语句的一般形式:

```

loop_statement
    ::= while (expression) statement
    | ...

```

计算 expression 的值,如果为真,执行语句。再次计算 while 表达式。结果,当 expression 为真时,我们进入循环,并在循环中重复执行。

while 表达式必须在循环内部更新,本例中是“ $i=i-1$ ”。while 语句不能用于等待它所属的 always 语句外部所产生的值的变化,正如下例所示。

```

module sureDeath(inputA);    // 不能工作!!
input inputA;

    always
    begin
        while(inputA)
            ;                // 等待外部变量
            // 其他语句
    end
endmodule

```

这里,while 语句表达式依赖于 inputA 的值,while 语句是空语句。上面的 while 语句在 inputA 的值变为 TRUE 之前不执行任何操作,接着执行 while 语句之后的其他语句。然而,既然在等待一个外部值的变化,正确的方式是使用 wait 语句。进一步的讨论请看 3.3 节的 wait 语句。

### 例 2.3 抽象的微处理器

```

module microprocessor;

    always
    begin
        powerOnInitializations;
        forever
        begin
            fetchAndExecuteInstructions;
        end
    end
endmodule

```

最后,forever 循环语句将一直循环执行。使用它的一个示例是微处理器的抽象描

述。这里我们可以发现特定的初始化工作仅发生在 power-on 时刻。其后，一直在 forever 循环中读取和执行指令。可以使用 disable 语句退出 forever 循环，这将在下一节讨论。如果退出了 forever 循环，那么 always 语句将开始执行 power-on 的初始化工作，并且再次开始 forever 循环。

forever 循环语句的一般形式：

```
loop statement
    ::= forever statement
    | ...
```

参考：disable 2.3, 3.6；等待 3.3；与等待比较 3.3.2；内部赋值循环 3.7

### 2.3.2 循环的异常退出

通常，循环语句会有一个“正常”的出口，比如，循环次数达到了循环计数器所指定的次数或者 while 表达式不再为 TRUE。然而，使用 disable 语句可以退出任何循环。disable 语句能终止任何有名 begin-end 块的执行；从紧接这个块的下一条语句继续执行。begin-end 块可以将块名放在 begin 关键字后面的冒号之后。C 编程语句有 break 和 continue 语句，例 2.4 展示了类似的用法。

#### 例 2.4 终止和继续循环

```
begin:break
  for (i=0; i<n; i=i+1)
    begin:continue
      if (a==0)
        disable continue;      // proceed with i=i+1
      ... // other statements
      if (a==b)
        disable break;         // exit for loop
      ... // other statements
    end
  end
```

例 2.4 列出了两个命名块：break 和 continue。C 语言中的 continue 语句跳过循环体的剩余部分，更新循环条件并再次执行循环。break 语句彻底退出循环，不进行循环条件的更新，也不判断循环结束条件是否满足。本例中的 disable 语句具有相似的功能。具体说来，disable continue 语句终止名为 continue 的 begin-end 块的执行，并且进行 for 循环条件的更新。disable break 语句终止包含 for 循环的块的执行。执行将从下一条语句继续。

disable 语句的一般形式如下:

```
statement
    ::= disable.. statement
    | ...

disable.. statement
    ::= disable task.. identifier;
    | disable block.. identifier;
```

参考: 终止有名块 3.6; 任务 2.5

## 2.4 多分支语句

多分支语句接受一个或多个动作的说明,每个动作基于特定的条件。Verilog 提供两种分支语句:if-else-if 和 case。

### 2.4.1 If-Else-If

if-else-if 只使用 if-else-if 语句说明多分支动作。它是多路选择的最一般的方式,它可以在 if 条件表达式中检查不同的表达式。试看例 2.5 中简单计算机的描述。本例让我们想起早期的 Mark-1 计算机(某些细节不同),这里是它的简单模型。其中采用了一个精确周期风格的描述,它将读取指令和执行指令分别在两个不同的时钟周期完成。

本例使用 if-else-if 语句说明计算机的指令译码。指令寄存器的第 15 到第 13 位 (ir[15:13])与 8 种可能组合中的 7 种相比较。由匹配项决定执行哪一条指令。

参考: if-then-else 2.2; 条件操作符 2.2.2

### 2.4.2 Case

当 if 条件可以用一个共同的基本表达式来表示时,可以使用 case 语句表达多分支结构。例 2.6 中,通过用 case 语句进行指令译码重写了 Mark-1 的描述。

case 表达式按照描述中的顺序进行计算。本例中,指令寄存器的第 15 到第 13 三位 (控制表达式)与 7 个 case 表达式中的每一个进行比较,位数必须一致。如果第一个表达式与控制表达式相匹配,则执行它后面的冒号后的语句。接着执行 case 语句后面的语句。这里进行的是四值逻辑比较。因此,一个两位的 case 条件可以出现十六种不同的值。

例 2.5 用 if-else-if 实现的 Mark-1 处理器

```
module mark1;
```

```

reg [15:0]      m[0:8191];          // 8192 x 16 位存储器
reg [12:0]      pc;                 // 13 位程序计数器
reg [12:0]      acc;                // 13 位累加器
reg [15:0]      ir;                 // 16 位指令寄存器
reg             ck;                 // 时钟信号

always
begin
    @(posedge ck)
        ir = m[pc];                 // 取指令
    @(posedge ck)
        if (ir[15:13] == 3'b000)    // 开始译码
            pc = m[ir[12:0]];        // 执行
        else if (ir[15:13] == 3'b001)
            pc = pc + m[ir[12:0]];
        else if (ir[15:13] == 3'b010)
            acc = -m[ir[12:0]];
        else if (ir[15:13] == 3'b011)
            m[ir[12:0]] = acc;
        else if ((ir[15:13] == 3'b101) || (ir[15:13] == 3'b100))
            acc = acc - m[ir[12:0]];
        else if (ir[15:13] == 3'b110)
            if (acc < 0) pc = pc + 1;
        pc = pc + 1;                 // 累加程序计数器
    end
endmodule

```

case 语句的一般形式:

```

statement
    ::= case.. statement
    |    ...

case.. statement
    ::= case ( expression ) case.. item { case item } endcase
    |    ...

case.. item

```

```

::= expression {,expression} ; statement_or_null
|      default [:] statement_or_null

```

## 例 2.6 用 case 语句实现的 Mark-1 处理器

```

module mark1Case;
reg [15:0]      m[0:8191];      // 8192 x 16 位存储器
reg [12:0]      pc;             // 13 位程序计数器
reg [12:0]      acc;           // 13 位累加器
reg [15:0]      ir;            // 16 位指令寄存器
reg             ck;            // 时钟信号

always
begin
    @(posedge ck)
        ir = m[pc];
    @(posedge ck)
        case (ir[15:13])
            3'b000 :      pc = m[ir[12:0]];
            3'b001 :      pc = pc + m[ir[12:0]];
            3'b010 :      acc = -m[ir[12:0]];
            3'b011 :      m[ir[12:0]] = acc;
            3'b100,
            3'b101 :      acc = acc - m[ir[12:0]];
            3'b110 :      if (acc < 0) pc = pc + 1;
        endcase
        pc = pc + 1;
    end
endmodule

```

可以在 case 表达式的位置使用 default 关键字来说明默认情况。当有默认语句时,如果没有其他 case 表达式匹配控制表达式,将执行默认语句。默认语句可以列在 case 语句的任意位置。

本例也例示了怎样说明同一个动作属于几个 case 项的情形。case 表达式之间的逗号说明任何一个比较如果为真,都将执行后面的语句。在 Mark-1 的示例中,如果选定的三位值为 4 或 5,累加器将被减去一个值。

最后,注意控制表达式和 case 表达式不必是常数。

参考: casez, casex 2.4.4; 与 if-else-if 比较 2.4.3; 条件操作符 2.2.2; 寄存器说明



### 2.4.3 Case 和 If-Else-If 的比较

在上述的 Mark-1 例中, case 和 if-else-if 都可能使用。该例中使用 case 会更加简洁, 更加易读。进一步说, 既然所有的表达式都要和相同的控制表达式相比较, case 语句更加合适。然而, 这些结构之间有两个主要区别。

- if-else-if 结构中的条件表达式更具有一般性。任意一种形式的表达式都可以用在 if-else-if 中, 然而在 case 语句中, 所有的 case 表达式都与控制表达式具有相同的形式。
- case 表达式可以包括未知(x)和高阻抗(z)位。它使用四值逻辑比较运算, 并且仅当每一位的 0, 1, x 和 z 完全匹配时, 比较才成功。因此, 保证 case 表达式和控制表达式的宽度一致是非常重要的。与此相比较, 涉及未知或高阻抗值的 if 语句表达式可能产生一个未知或高阻抗值, 这将被解释为 FALSE(如果没有使用 case 等于)。一个用于调试的具有未知和高阻抗值的 case 语句示例如下所示。

```
reg ready;           // 一位寄存器
// 其他语句
case(ready)
    1'bz:             $display("ready is high impedance");
    1'bx:             $display("ready is unknown");
    default:          $display("ready is %b", ready);
endcase
```

本例中, 一位的 ready 与高阻抗值(z)和未知值(x)相比较; 模拟期间打印输出适当的显示信息。如果 ready 既不是高阻抗值, 也不是未知值, 就显示它的值。

参考: 四值逻辑 4.2.2; casez, casex 2.4.4; case 等于 2.2

### 2.4.4 Casez 和 Casex

casez 和 casex 是两种接受无关值的 case 语句。casez 将 z 值看作无关值, casex 将 z 和 x 值都看作无关值。还有另外一种标识 z 或 x 的方式, 它们可以用表示无关值的问号(“?”)来说明。除了用 casez 或 casex 关键字来替换 case 以外, casez 和 casex 的语法与 case 语句的语法相同。

```
case_statement
::= case ( expression ) case_item { case_item } endcase
| casez ( expression ) case_item { case_item } endcase
| casex ( expression ) case_item { case_item } endcase
```

#### 例 2.7 Casex 示例

```

module decode;
    reg[7:0]      r;

    always
        begin
            // 其他语句
            r = 8'b1x0x1x0;
            casex(r)
                8'b001100xx: statement1;
                8'b1100xx00: statement2;
                8'b00xx0011: statement3;
                8'bxx001100: statement4;
            endcase
        end
endmodule

```

注意例 2.7 中所示的 casex 语句。这个示例中我们使用了具有八位值 x1x0x1x0 的寄存器 r, 它表示每隔一位就有一个未知值。既然未知值 x 被看作无关值, 那么只有语句 2 会执行。虽然语句 4 也匹配, 但是因为语句 2 的条件先被找到, 所以语句 4 不会执行。

x1x0x1x0	寄存器 r 中的值
1100xx00	相匹配的 case 表达式

这两种类型的 case 语句的区别在于只将 z 看作无关值 (casez) 还是将 z 和 x 都看作无关值 (casex)。

参考: Verilog 操作符 C; case 2.4.2

## 2.5 函数和任务

在软件编程中, 用函数和过程把大的程序分成更容易管理的几部分。Verilog 中模块把设计分成更容易管理的各部分, 然而, 使用模块意味着要描述结构化的边界。这些边界可以为逻辑结构或物理包装的边界建模。考虑到把模块的行为描述分成更容易管理的各部分, Verilog 提供了函数和任务, 它类似软件函数和过程。

正像软件编程, 函数和任务因为几个原因而很有用。它们允许将常用的行为描述写在一起, 需要时再调用。它们也采用更清晰的书写风格; 代替冗长的行为描述序列, 这样的序列可以分成一块块更易于阅读的几部分, 不论它们被调用一次还是多次。最后, 它们使局部数据对设计的其他部分不可见。事实上, 函数和过程在设计更易读、易维护的行为

描述中扮演了关键的角色。

### 例 2.8 具有乘法指令的 Mark-1

```
module mark1Mult,  
    reg [15:0]      m[0:8191];      // 8191 x 16 位存储器  
    reg [12:0]      pc;              // 13 位程序计数器  
    reg [12:0]      acc;             // 13 位累加器  
    reg [15:0]      ir;              // 16 位指令寄存器  
    reg             ck;              // 时钟信号  
  
    always  
        begin  
            @(posedge ck)  
                ir = m[pc];  
            @(posedge ck)  
                case (ir [15:13])  
                    3'b000 : pc = m [ir [12:0]];  
                    3'b001 : pc = pc + m [ir [12:0]];  
                    3'b010 : acc = -m [ir [12:0]];  
                    3'b011 : m [ir [12:0]] = acc;  
                    3'b100,  
                    3'b101 : acc = acc - m [ir [12:0]];  
                    3'b110 : if (acc < 0) pc = pc + 1;  
                    3'b111 : acc = acc * m [ir [12:0]];      // 乘法  
                endcase  
                pc = pc + 1;  
            end  
        end  
endmodule
```

这里把前面章节中的 Mark-1 描述的操作码 7 定义为乘法指令。在行为建模进程的初期,可以像例 2.8 那样使用乘法操作符。对设计进程的初期来说,这是一个很合理的行为模型,功能性在其中充分地得到了描述。然而,可能需要指定这个设计中用到的乘法算法的具体过程。我们的第一种方法是使用函数和任务来描述乘法。以后,可把这种方法与将乘法描述成单独的并发操作模块作比较。表 2.1 比较了任务和函数的不同。

表 2.1 任务和函数的比较

项 目	任 务	函 数
使能(调用) enabling(calling)	任务调用是一个单独的过程语句。它不能在持续赋值语句中调用。	函数调用是表达式中的一个操作数。它在表达式中调用,返回表达式中要用的值。函数可以在过程和持续赋值语句中调用。
输入和输出	任务可以有零个或多个各种类型的参数。	函数至少有一个输入。它不能将 inout 类型作为输出。
定时和事件控制( #, @和 wait)	任务可以包含定时和事件控制语句。	函数不能包含这些语句。
调用其他任务和函数	任务可以调用其他任务和函数。	函数可以调用其他函数,但不可以调用其他任务。
返回值	任务不向表达式返回值。然而,被任务写进它的输入输出(inout)端口和输出端口的值在任务执行结束时可以复制回来。	函数为调用它的表达式返回一个值,这个返回值在函数内部被返回给函数标识符。

### 2.5.1 任务

Verilog 的任务与软件中的过程类似。它由调用语句调用,并且执行之后返回到下一条语句。它不能用在表达式中。可以给它送入参数并且返回结果。它内部可以声明局部变量,这些变量的作用域就是这个任务。例 2.9 说明了怎样使用任务描述乘法算法来重写模块 Mark-1。

#### 例 2.9 任务的描述

```

module mark1Task;
    reg[15:0]      m[0:8191];          // 8191 x 16 位存储器
    reg[12:0]      pc;                  // 13 位程序计数器
    reg[12:0]      acc;                 // 13 位累加器
    reg            ck;                  // 时钟信号

    always
        begin; executeInstructions
            reg[15:0]          ir;      // 16 位指令寄存器

            @(posedge ck)
                ir = m[pc];
        end
endmodule

```

```

    @(posedge ck)
        case (ir[15:13])
            // 跟上例一样的其他 case 表达式
            3'b111 : multiply (acc, m[ir[12:0]]);
        endcase
        pc = pc + 1;
    end

task multiply;
    inout      [12:0]      a;
    input       [15:0]      b;

    begin; serialMult
        reg      [5:0]      mcnd, mpy; // 被乘数和乘数
        reg      [12:0]      prod;      // 乘积

        mpy = b[5:0];
        mcnd = a[5:0];
        prod = 0;
        repeat (6)
            begin
                if (mpy[0])
                    prod = prod + {mcnd, 6'b0000000};
                prod = prod >> 1;
                mpy = mpy >> 1;
            end
        a = prod;
    end
endtask
endmodule

```

任务在模块中用 **task** 和 **endtask** 关键字定义。这个任务被命名为 multiply, 它被定义了一个 inout 端口(a)和一个 input 端口(b)。这个任务在 always 语句中调用。在调用点, 参数的顺序必须与任务定义中的顺序相对应。当调用 multiply 时, acc 被复制给任务的变量 a, 从存储器读取的值复制给 b, 然后执行任务。当任务要返回时, prod 的值被复制给 a。一旦返回, a 被复制给 acc, 并且继续执行任务调用点的下一条语句。虽然这里没有说明, 任务可以包含定时和事件控制语句。例 2.9 中 begin-end 块使用了两次, 说明在

这些块中,可以定义新的寄存器标识符。这些名称(ir,mcnd,mpy 和 prod)的作用域是 begin-end 块。任务声明的一般形式:

```
task_declaration
    ::= task task_identifier ;
        {task_item_declaration}
        statement_or_null
    endtask
```

```
task_argument_declaration
    ::= block_item_declaration
    | output_declaration
    | inout_declaration
```

```
block_item_declaration
    ::= parameter_declaration
    | reg_declaration
    | integer_declaration
    | real_declaration
    | time_declaration
    | realtime_declaration
    | event_declaration
```

乘法算法使用移位加方法。操作数的低 16 位相乘产生一个 32 位结果。语句

```
mpy = b[5:0];
```

在 b 上进行域选择并且把低 6 位送入 mpy。乘数(mpy)的最低位被检查六次,如果为 1,被乘数(mcnd)的右侧连接(用“{,}”操作符)六个 0,再加入到结果(prod)中。结果和乘数都右移一位,再循环执行。

连接的一般形式如下所示:

```
concatenation
    ::= {expression{,expression}}
multiple_concatenation
    ::= {expression{expression{,expression}}}
```

第一种形式出现在本例中。第二种形式允许内部连接重复 n 次,n 由第一个表达式所给定。

任务中声明的 input、output 和 inout 名称是与调用点命名的变量相分离的。它们的

范围是 task-endtask 块。当调用任务时,在调用点将值拷贝给声明为 input 或 inout 的内部变量。任务再继续执行。当执行完成后,把所有声明为 inout 或 output 的内部变量的值拷贝给调用点的变量。当值从调用点复制到任务内或从任务内复制到调用点时,所有调用点的变量按从左到右的顺序与任务定义中的 input、output 和 inout 声明的顺序一致。

任务可以自我调用或被它调用的任务所调用。然而,在硬件实现中,只有一组寄存器保存任务的变量。结果,第二次调用任务后用到的寄存器是以前调用中用到的同样的物理实体。模拟器控制了任务的执行,以便一个被调用多次的任务能正确地返回。进一步说,任务可能被不同的进程(也就是 always 和 initial 语句)调用,当这个任务被一个进程调用时,它甚至可能正被另一个进程挂起在一个事件控制处。两次任务只使用一组变量。然而,模拟器保存了符号各时刻的值,所以当任务退出时,返回给调用进程的适当的返回值已经产生。

任务调用的一般形式:

```
task enable
    ::= task_identifier[(expression{,expression})];
```

有必要说明一下本例中的连接操作符。符号“{,}”用于表示值的连接。本例中,我们将两个 6 位的值连接在一起,然后加到 13 位的 prod 上去。6 位二进制 0(这里用二进制格式表示)被连接到 mcnd 的右侧。注意本例中,必须为常数精确地说明位宽,以便 mcnd 与 prod 对准位号,然后相加。

参考:函数 2.5.2; 标识符 B.5,G.10

## 2.5.2 函数

Verilog 函数与软件函数类似。它可以在表达式中调用并且向表达式返回值。函数有一个输出(函数名)和至少一个输入。其他标识符可以在函数内部声明,它们的作用域是函数内部。不像任务,函数不可以包含延迟( # )或事件控制( @ , wait )语句。这里虽然没有说明,函数可以在持续赋值语句中调用。函数也可调用其他函数,但不能调用其他任务。函数执行期间,必须有一个值被赋给函数名;这就是函数的返回值。

例 2.10 列出了用乘法函数说明的模块 mark1Fun。模块中用 function 和 endfunction 关键字来定义函数。函数声明包括函数名和位宽。调用时,参数复制给函数的输入;像任务一样,必须注意声明的顺序。函数执行后,要给函数名赋值。一旦返回,函数名(multiply)的最终值将送回到调用点并复制给寄存器 acc。注意,这里使用了有名 begin-end 块,其中可以定义新的寄存器。函数声明的一般形式:

```
function declaration
    ::= function [range or type] function_identifier ;
```

```

function_ item_ declaration { function_ item_ declaration }
statement
endfunction

```

range\_ or\_ type

```

::= range | integer | real | realtime | time

```

range

```

::= [ msb_ constant_ expression ; lsb_ constant_ expression ]

```

function\_ item\_ declaration

```

::= block item declaration
| input_ declaration

```

函数从过程表达式或持续赋值语句内部调用,调用采用如下形式:

function\_ call

```

::= function_ identifier ( expression { , expression } )

```

参考:持续赋值语句中的函数 4.4.1; 任务 2.5.1; 标识符 B.5,G.10

## 例 2.10 函数说明

module mark1Fun;

```

    reg [15:0]    m [0:8191];      // 8191x16 位存储器
    reg [12:0]    pc;              // 13 位程序计数器
    reg [12:0]    acc;             // 13 位累加器
    reg          ck;              // 时钟信号

```

always

```

    begin: executeInstructions

```

```

        reg [15:0]    ir;        // 16 位指令寄存器

```

```

        @(posedge ck)

```

```

            ir = m [pc];

```

```

        @(posedge ck)

```

```

            case (ir [15:13])

```

```

                // 跟上例一样的其他 case 表达式

```

```

                3'b111,    acc = multiply(acc, m [ir [12:0]]);

```



```

        endcase
        pc = pc + 1;
    end

function [12:0] multiply;
input      [12:0]      a;
input      [15:0]      b;

    begin: serialMult
        reg      [5:0]      mcmd, mpy;

        mpy = b[5:0];
        mcmd = a[5:0];
        multiply = 0;
        repeat (6)
            begin
                if (mpy[0])
                    multiply = multiply + {mcmd, 6'b0000000};
                    multiply = multiply >> 1;
                    mpy = mpy >> 1;
                end
            end
        end
    endfunction
endmodule

```

### 2.5.3 结构视域

前面章节的任务和函数的示例已经阐明了行为模型的不同组织方式。就是说，我们可以选择不同的方式建立行为模型，产生同样的结果。当我们用操作符 \* 时，可能只对模型的模拟感兴趣。硬件中有许多种实现乘法操作的方式，但在设计的早期，我们情愿用模拟器来替代它。

设计时，我们想指定硬件实现的乘法算法。通过使用上例中的任务和函数语句来达到目的。如果描述中使用了任务或函数，表示乘法算法将是最终的数据通道和状态机的一部分，将它们综合，用以实现 Mark-1 处理器。就是说，我们通过说明乘法算法的细节扩大了行为描述，结果实现这个行为的状态机有更多的状态。同样，数据通道可能需要更多的部件来保存这些值，实现这些操作。

另一个设计抉择可能用到一个预定义的与 Mark-1 模块有关的乘法模块。这种情况

在例 2.11 中有说明,在 mark1Mod 模块中用一个模块示例实现乘法。如果要用以前设计的乘法模块,或设计者想在设计中进行模块的功能划分,就会用到这种描述方法。multiply 模块端口连接到 mark1Mod,mark1Mod 模块从 go 行开始执行 multiply 模块。一旦完成,multiply 模块用信号通知 Mark-1,Mark-1 正在 done 行等待。这个结构描述产生了不同的设计。现在我们有二个状态机,一个是 mark1Mod 的状态机,另一个是 multiply 的状态机。为了保持二个模块同步,我们定义了一个握手协议,它使用等待语句和变量 go 与 done。在设计抉择中,不可能指出哪种方案最好。我们只能建议,正如上面所做的那样,为什么不用行为建模结构(\*,函数,任务)独立地描述系统或进行行为的结构划分呢?

参考:持续赋值中的函数 4.4.1

### 例 2.11 乘法作为独立的模块

```
module mark1Mod;
    reg [15:0]    m [0:8191];        // 8191x16 位存储器
    reg [12:0]    pc;                // 13 位程序计数器
    reg [12:0]    acc;               // 13 位累加器
    reg [15:0]    ir;                // 16 位指令寄存器
    reg           ck;                // 时钟信号

    reg          [12:0] mcnd;
    reg           go;
    wire          [12:0] prod;
    wire          done;
    multiply       mul (prod, acc, mcnd, go, done);

    always
        begin
            @(posedge ck)
                go = 0;
                ir = m [pc];
            @(posedge ck)
                case (ir [15:13])
                    // 其他 case 表达式
                    3'b111:  begin
                                wait (~done) mcnd = m [ir [12:0]];
                                go = 1;
                                wait (done);
                            end
                endcase
        end
endmodule
```

```

                                acc = prod;
                                end
                                endcase
                                pc = pc + 1;
                                end
endmodule

module multiply (prod, mpy, mcnd, go, done);
    output      [12:0]      prod;
    input       [12:0]      mpy, mcnd;
    input
    output      done;

    reg         [12:0]      prod;
    reg         [5:0]      myMpy;
    reg         done;

    always
    begin
        done = 0;
        wait (go);
        myMpy = mpy[5:0];
        prod = 0;
        repeat (6)
            begin
                if (myMpy[0])
                    prod = prod + {mcnd, 6'b0000000};
                prod = prod >> 1;
                myMpy = myMpy >> 1;
            end
        done = 1;
        wait (~go);
    end
endmodule

```

## 2.6 作用域规则和层次名

一个标识符的作用域是 Verilog 描述中该标识符可以被识别的范围。作用域规则定

义了这个范围。Verilog 也有层次命名的特征,可以在设计的任何地方访问设计中的任意标识符。

### 2.6.1 作用域规则

模块名称是全局可见的。Verilog 可以在四种实体中定义标识符,这四种实体是:模块、任务、函数和有名块。每一个实体定义了标识符的局部作用域,即描述中标识符可以被识别的范围。这个局部作用域包括 module-endmodule、task-endtask、function-endfunction 和 begin:name-end 对。在一个局部作用域内,一个名称只可以有一个标识符与之对应。

标识符在局部作用域之外也可见。为了理解作用域规则,我们需要区分允许和不允许超前引用的位置。

- 超前引用。模块、任务、函数和有名 begin-end 块中的标识符可以超前引用,即标识符可以在定义之前就使用。就是说,在这些实体定义之前,你可以示例化一个模块,调用一个任务,调用一个函数或终止一个有名块。

- 非超前引用。寄存器和线网不能超前引用。就是说,在使用它们之前,必须先定义。通常的做法是,在使用它们的局部作用域(即模块、任务、函数或有名 begin-end)的开始处定义它们。一个例外情况是逻辑门的输出可以隐式声明(参见 4.2.3 节)。

可以超前引用的实体(模块、任务、函数或有名 begin-end 块)中,有一个模块示例层次定义的向上作用域。从最低层看,可以识别每一个比它层次高的局部作用域中的超前引用标识符。这个模块示例层次的向上路径就是向上作用域。

考虑例 2.12。top 模块的局部作用域中的标识符是 top、instance1、y、r、w 和 t。当在 top 模块中示例化模块 b 时,b 中的过程语句能够调用 top 模块的局部作用域中定义的任务和函数,也能终止这个局部范围中的有名块。模块 top 中的任务 t 在它的作用域内有一个有名块(c)。不能在模块 b 中终止 c,因为 c 不在 top 的局部作用域之内,因此它不在 b 的向上作用域(实际上,它比模块 b 低一级,在任务 t 的局部作用域内)之内,进一步说,可以在模块 b 中终止 top 局部作用域中的有名块 y,模块 top 中定义的任务或函数,或者它们内部的有名块也可以终止它。

#### 例 2.12 作用域和层次名称

```
module top;
    reg      r;      // 层次名称是 top. r
    wire     w;      // 层次名称是 top. w
    b instance1();

    always
```

```

begin: y
    reg          q;    // 层次名称是 top. y. q
end

task t;
    begin: c          // 层次名称是 top. t. c
        reg          q;    // 层次名称是 top. t. c. q
        disable y;    // OK
    end
endtask
endmodule

module b;
    reg          s;    // 层次名称是 top. instancel. s

    always
    begin
        t;            // OK
        disable y;    // OK
        disable c;    // 错误, c 不可见
        disable t. c; // OK
        s = 1;        // OK
        r = 1;        // 错误, r 不可见
        top. r = 1;    // OK
        t. c. q = 1;   // OK
        y. q = 1;      // OK, 不同于 t. c. q 的 q
    end
endmodule

```

然而要注意,寄存器  $r$  和连线  $w$ ,虽然在模块  $b$  的向上作用域中,却不能从这里访问;寄存器和线网不能超前引用,只能在局部作用域内访问。规则是,只有通过模块示例树查找到某示例之后,该示例中的超前引用标识符(即模块、任务、函数和有名块标识符)才可见。当到达顶层时(一个没有在别处示例化的模块),对标识符的查询就结束了。局部范围中,非超前引用的寄存器和线网是直接可见的。

重要的是要注意某模块中定义的任务和函数可以被该模块中任何模块示例(任意级别)调用。因此,在设计的很多部分都要用到的函数和任务应该在顶层模块定义。

可以认为向上标识符树的组成成分有两种:模块层次和过程语句层次。模块层次树

如上所述。过程语句层次由 always 语句、initial 语句、任务以及函数中嵌套的有名块组成。过程语句层次起源于模块(本质上讲,它们是 always 和 initial 语句)并且独立于模块层次。当访问非超前引用标识符(寄存器和线)时,嵌套的最内层的语句在过程语句层次中查找这个标识符。当找到过程语句层次的根时,查询结束。因为可以访问的非超前引用标识符只能是当前局部作用域和它的过程语句层次中的非超前引用标识符。

结果,代表寄存器和线网的标识符在过程语句层次中查找,不跨越模块示例的边界。表示任务、函数和有名块的标识符在过程和模块示例层中查找。

## 2.6.2 层次名

前一节讨论了一个描述中标识符的向上作用域。在可能的情况下,应该使用这样表示的标识符,它们更容易读和写。另一方面,层次名可以唯一标识整个描述中的任意一个任务、函数、有名块、寄存器或者连线。

层次中的各名称是超前引用的,在所有的模块示例化之前,它们是不能识别的。层次名是由点(".")相隔的一系列标识符组成的路径名称。第一个标识符是一个在过程和模块层次名称树中查找到的超前引用标识符。从找到第一个标识符的地方开始,每一个后继标识符指定继续向下查找的有名范围。最后的标识符说明了要查找的实体。

考虑例 2.12。在模块 b 中,寄存器 r 不可见,因为寄存器和线的标识符不能跨模块示例查找;它们只在局部范围内可见。然而,在模块 b 中的层次引用 top.r 将访问 top 中的 r。与此类似,在模块 b 中,t.c.q 访问任务 t 中的寄存器 q(顺便说一句,它不同于有名块 y 中的寄存器 q)。进一步说,可以在 b 中通过层次名 t.c 来终止块 c。注意,后面两种情况不是从 top 开始访问(虽然可以)。当从 b 查找模块层次时,下一个上查范围包括超前引用名 top,y 和 t。它们中的任意一个(事实上任意一个超前引用标识符——模块、任务、函数或者有名块)都可以用作层次名的根。实际上,当使用层次名来指明一个寄存器或连线时,名称中的第一个标识符必须是超前引用标识符。最后一个标识符是寄存器或连线。指定名称不必从 top 开始。从模块 b 看来,top.t.c.q 和 t.c.q 没有区别。

虽然我们可以通过这种机制访问描述中的任何有名项目,但是,保持局部和向上作用域规则是更合适的,这种规则保持了更好的风格、易读性和可维护性。少用层次名称,因为它们违反了局部化访问设计元素的风格,在访问中不要太随意。

确实,通过层次命名,任何元素都可以访问任何别的元素。然而,这在风格上可能不好。应该尽可能坚持局部和向上作用域规则。

## 2.7 小结

到目前为止,我们所见到的行为建模语句很像软件编程语言中的语句。就目前而言,

主要的区别在于 Verilog 语言采用分离的机制处理结构层次和行为分解。为了使模块的行为描述适合软件工程方法,提供了函数和任务。那就是说,我们可以将很长的、可能重复的描述分割成行为子块。同时,我们使用模块定义来描述设计的结构层次,并且将并行操作行为分成不同的模块。2.5 节中的示例列出了两种建模方法。下一章讨论并发行为。

## 2.8 练习

- 2.1 改写例 2.9 中包含右移操作符的表达式,只使用位选、域选和连接操作。
- 2.2 用如下所示的寄存器声明和 for 循环替换例 2.2 中的 repeat 循环能实现同样的功能吗?

```
reg [3:0] i;

for (i=0; i<=DvLen; i=i+1)
    begin
        // shift and subtract statements
    end
```

- 2.3 下面是一个含两个 case 项目的 case 语句。每个项目调用不同的任务。如果要枚举所有可能的 case 项目,应该有多少项呢?

```
reg [3:0] f;
case (f)
    4'b0110: taskR;
    4'b1010: taskS;
endcase
```

- 2.4 在下面的函数中写一个等价于 casez 语句的 for 循环语句,不要增加任何变量。

```
function [7:0] getMask;
input [7:0] halfVal;
casez (halfVal)
    8'b?????? 1: getMask = 8'b11111111;
    8'b?????? 10: getMask = 8'b11111110;
    8'b????? 100: getMask = 8'b11111100;
    8'b???? 1000: getMask = 8'b11111000;
    8'b??? 10000: getMask = 8'b11110000;
    8'b?? 100000: getMask = 8'b11100000;
```

```

        8'b? 1000000; getMask = 8'b11000000;
        8'b10000000; getMask = 8'b10000000;
        8'b00000000; getMask = 8'b11111111;
    endcase
endfunction

```

- 2.5 模拟 multiply 任务并显示 mpy, mcmd 和 prod 的初始值以及每次循环(共 6 次循环)结束时的值。添加一个 \$display 语句来显示这些值。
- 2.6 写出例 2.9 和例 2.10 中每个任务、函数和寄存器的层次名称。
- 2.7 例 2.12 的模块 b 中,任务 t 中的寄存器 q 可以用 top. t. c. q 或 t. c. q 的方式来引用。为什么在模块 b 中只有一种方式来引用寄存器 r 呢?
- 2.8 观察后面的例 4.18。
  - A. 模块 slave 中的行为语句怎样调用任务 wiggleBusLines?
  - B. 如果模块 master 和 slave 都需要调用任务 wiggleBusLines,在哪里定义这个任务合适呢?



## 第3章 并发进程

迄今为止,我们所讨论的大多数建模语句是用单进程示例来说明的。这些语句作为 always 语句的一部分被重复地顺序执行。它们的操作对象是一些数值,这些数值来自模块的输入或输出,或者来自模块的内部寄存器。在本章中,我们将给出行为建模语句。从定义上讲,它们与处于封闭的 always 语句之外的活动进程交互作用。例如,wait 语句总是等待其他进程改变某个值而使得其表达式变为真(true)。在这里,以及在我们将要看到的其他情况中,等待语句的操作依赖于系统中并发进程的行为。

### 3.1 并发进程

前面已经将进程定义为一个控制器的抽象,它是一个能够使存储在系统中寄存器的数值发生改变的控制线程的抽象。对于数字系统,可将其想像为一个相互之间通信的并发进程的集合,或者认为是具有独立性的并且在相互之间传递信息的控制行为的集合。需要注意的是,每一个进程中都包括状态信息,并且状态的变化是该进程的当前输入以及当前状态的函数。

例 3.1 是一个计算机系统的抽象描述。always 语句描述的进程对应于硬件控制器,它由一个带有输出及后继状态的时序状态机来实现。该状态机控制一个数据通道,此数据通道包括寄存器、算术逻辑单元以及定向逻辑,如总线和多路器等。

例 3.1 一个抽象的计算机描述

```
module computer;
    always
        begin
            powerOnInitializations;
            forever
                begin
                    fetchAndExecuteInstructions;
                end
            end
        end
endmodule
```

我们假设这个进程和系统中另一个进程交互作用,这个进程可能是一个输入接口并

且它能够从一个调制解调器(modem)中接受位串行(bit-serial)信息。在这个示例中,将进程抽象化是必要的,因为其中有两个相互独立但是却又互相联系的控制线程:计算机和输入接口。输入接口等待调制解调器的新的输入位,并且在收到一个字节的的数据之后向计算机发出信号。另一个进程,即例 3.1 中描述的计算机进程,当接收到全字节(full byte)的信息时只与输入接口进程发生作用。

这两个进程实际上是可以作为一个整体描述的,但是这将会杂乱无章,难以阅读。实际上,计算机进程中的每一条语句都应包括一条检查指令,检查此时是否有新的输入数据出现,并且包括一个对新输入进行处理的描述。在最坏的情形下,假设有两个进程,它们分别有  $N$  和  $M$  种状态,那么,合为一体后,与其功能相同的组合进程将有  $N \times M$  个状态,这个描述将是极其复杂的。事实上,将一个系统内部的多个进程分别加以考虑和描述是很有必要的。

当几个进程同时存在于一个系统中时,信息在其中流动,我们则要使它们同步以确保所传输信息的正确性。使其同步的原因是因为其中的一个进程并不知道另一个进程此时正处于何种状态,只有靠另一进程给出一个明确的信号来指明状态信息。也就是说,进程之间相对来说是异步的。例如,它们可以按自己的时钟速度进行操作,或者它们产生数据的时间间隔与另一个进程使用这些数据的时间间隔不同步。在本例中,我们就必须使进程同步,并且必须明确给出它们之间的同步信号,指出一些有关内部状态的信息以及这些状态之间所共享的数据的信息。

在硬件中,实现进程同步的一种办法是采用“数据准备好(data-ready)”的握手信号——只有当有其他进程用“data-ready”指示有新的数据出现时,进程才会读取共享数据。当其他进程指示数据已被读取时,第一个进程则会将“data-ready”置为无效,并一直维持到有新的信息出现。另一种方法是用时钟信号来同步多个进程。每个值都要确保有效,并且规定相应的动作出现于一个或者两个时钟边沿上。诸如握手信号、时钟信号之类的同步信号在保证多个进程之间信息的正确传输过程中是必要的。

在这一章中,我们将给出一些语句,这些语句是和一些用来描述并发进程之间的交互作用的行为描述相互关联的。

参考: always, initial 2.1; 过程时序模型 5.1; 不确定性 5.3

## 3.2 事件

**事件控制语句**提供一种监视数值变化的方法。包含事件控制语句的进程的执行将被挂起,直到发生变化。因此,数值必须由另一个进程来改变。

这一节中讨论的结构会触发一个值的改变,注意到这一点很重要。也就是说,这些结构对触发沿是敏感的。当控制权传递到其中的一条语句中时,检测被触发的输入的初始

值。随后,当该数值发生改变时(例如,该值会出现一个上升沿),则完成该事件控制语句,并且控制权继续传递到下一条指令中。

这一节中包含事件控制的两种基本形式:一种形式监视数值的改变,另一种形式称作**有名事件**,用来监视称作事件的抽象信号。

### 3.2.1 事件控制语句

应用例 3.2 开始讨论事件控制语句。本例中语句

```
@(negedge clock) q<=data;
```

用来模拟 D 型触发器的下降沿触发。这个具有过程性质的事件控制语句监视 clock 的负跳变,然后将 data 的值赋给 q。赋给 q 的值恰好是在时钟边沿之前的 data 的值。

#### 例 3.2 D 型边沿触发器

```
module dEdgeFF (q, clock, data);
    output      q;
    reg         q;
    input       clock, data;

    always
        @(negedge clock) q <= data;
endmodule
```

除了指定要触发的下降沿以外,也可以规定一个上升沿(“posedge”)或不做任何规定。请考虑

```
@(ricky) lucy = crazy;
```

这里如果 ricky 发生改变,crazy 的值将装入 lucy 中。

事件控制语句的一般形式是:

```
event_control
    ::= @event_identifier
    |   @(event_expression)

event_expression
    ::= expression
    |   event_identifier
    |   posedge_expression
    |   negedge_expression
```

| event.. expression or event.. expression

限定词可以是“posedge”、“negedge”或者空白。表达式是一个门的输出、连线(wire)或者寄存器,它们的值由另一个进程产生。事件控制语句从接受控制权的开始就观察所指定的变化。而发生在接受控制权以前数值的变化则忽略不计。一旦有事件发生,即执行该语句。如果在等待事件发生时,表达式产生一个新值,并且该值和旧值是相同的,那么就认为没有事件发生。

有时事件控制表达式可能会带有一些未知的值。在这种情况下,下降沿就被定义为从1到0、从1到x或者从x到0的跳变。一个上升沿则被定义为从0到1、从0到x或者从x到1的跳变。

事件控制语句中可以描述任意数目的事件,所以其中任意一个事件的发生都会导致该语句的执行。例3.3显示一个超时范例。

**例 3.3** 在一个事件控制语句中的两个事件的或

```
always
begin
    // start the timer that will produce the timeOut signal;

    @(posedge inputA or posedge timeOut)
        if (timeOut)
            // ... error recovery
        else regA = regB;      // normal operation
    // ... other statements
end
```

本例中我们来观察两个事件中的任意一个, inputA 的上升沿或者 timeOut 的上升沿。用关键字 or 来将这两个事件分开。在这种情况下,我们可以触发预期的事件——inputA 的变化。但是,如果在一定时间内 inputA 事件没有发生,那么进程可以摆脱死锁状态并进行某种形式的错误修正。

or 结构在并发进程的应用中是很重要的。如果一个进程需要等待几个事件中的任何一个事件发生时,在等待另一个事件之前,它不必过早地等待一个特定的事件。的确,由于事件不可能以确定的顺序发生——这个顺序是数据相关的——所以用某一特定顺序等待触发事件可能会导致系统的死锁。也就是过程可能会等待一个永远不会出现的事件。而 or 结构允许我们等待几个事件中的任何一个事件。

参考:电平敏感的 wait 3.3; 比较事件语句和等待语句 3.3.3; 内赋值延迟 3.7

### 3.2.2 有名事件

以上描述的事件控制语句要求明确说明一个变化。事件控制语句的一种更为抽象的

形式,有名事件 (named event),可以将一个触发事件送到设计的另一部分。触发事件不是用寄存器或连线来实现的,因而实际上更为抽象。此外,即使它超出了本模块的有效范围,也不需要端口规范。在继续执行之前,设计的其他部分应观察有没有有名事件的发生。

例 3.3 给出一个 Fibonacci 数字发生器的示例。它运用有名事件在两个模块之间进行通信。topFib 模块只例示了两个模块(fnc 和 ng)。

numberGen 模块中的 always 语句说明事件 ready 的触发:

```
#50 -> ready;
```

事件必须在模块描述的第四行进行声明。always 语句会连续延迟 50 个时间单位,并且使 number 的值递增,再延迟 50 个时间单位,然后触发事件 ready。

模块 fibNumCalc 观察 always 语句中第一行的事件:

```
@ng. ready  
count = startingValue;
```

名称“ng. ready”是事件 ready 的层次化名称,对于它的解释,我们将在讲解完有名事件的工作机理之后再来给出。模块 fibNumCalc 为了接收触发,首先必须开始执行@事件语句,然后执行在模块 numberGen 中的触发语句。这时,模块 fibNumCalc 会继续执行语句“count = startingValue;”。

请注意使一个事件触发的行为本身就是一条语句,它无须与延时运算符结合起来,如例中所示。激活有名事件的一般形式是:

```
statement  
    ::= event. trigger  
  
event. trigger  
    ::= -> event. identifier;
```

如果模块 fibNumCalc 执行 always 循环的时间超过 100 个时间单位,这里描述的 Fibonacci 数字发生器的确会发生竞争状态(race condition)。模块 numberGen 每隔 100 个时间单位就产生一个值,并发出一个触发事件。如果 fibNumCalc 模块在 100 个时间单位内未能完成 always 循环,就将错过 NumberGen 的触发事件。其结果是模块 NumberGen 产生的 Fibonacci 数字将会每隔一个数字才会被计算一次。

例 3.4 使用有名事件的 Fibonacci 数字发生器

```
module topFib;  
    wire [15:0]      number, numberOut;
```

```

        numberGen          ng      (number);
        fibNumCalc          fnc     (number, numberOut);
endmodule

module numberGen(number);
    output      [15:0]      number;
    reg         [15:0]      number;
    event       ready;      // declare the event

    initial
        number = 0;

    always
        begin
            #50 number = number + 1;
            #50 -> ready;      // generate event signal
        end
endmodule

module fibNumCalc(startingValue, fibNum);
    input      [15:0]      startingValue;
    output     [15:0]      fibNum;

    reg        [15:0]      count, fibNum, oldNum, temp;

    always
        begin
            @ng. ready      // wait for event signal
                count = startingValue;
            oldNum = 1;
            for (fibNum = 0; count != 0; count = count - 1)
                begin
                    temp = fibNum;
                    fibNum = fibNum + oldNum;
                    oldNum = temp;
                end
            $display ("%d, fibNum=%d", $time, fibNum);
        end
endmodule

```

```
        end  
    endmodule
```

请注意并没有用寄存器来保持触发事件,也没有用任何线来传送触发事件。相反,它只是一个概念上的事件,当它在一个模块中出现时,能够触发其他先前在@事件处停止的模块。此外,因为有名事件没有与之对应的硬件实现,所以它比事件控制语句更为抽象。相比之下,上升沿事件控制语句意味着使用边沿触发逻辑的某种形式来检测发生的跳变。有名事件一般使用在模拟中。

参考: 层次化名称 2.6

### 3.3 等待语句

等待语句是一条并发进程语句,它等待语句中的条件表达式变为真。从概念上讲,进程的执行一直停止,直到条件表达式变为真。从定义上讲,条件表达式中应该至少包含一个值,该值由一个单独的并发进程产生——否则,条件表达式将永远不会改变。因为等待语句必须使用来自其他进程的输入,所以它是使两个并发进程同步的主要方法。

等待语句的条件是电平敏感的。也就是,它并不等待数值的改变。相反,它仅仅检查条件表达式是否为真。如果为真,进程继续运行;如果为假,进程将等待下去。

等待语句通常用于同步两个进程的握手情况下(设计握手信号)。例 3.5 描述了这样一种情形:如果 ready 的输入为真,进程将只读 dataIn 的输入。通过确保当生产者进程(producer process)产生了 dataIn 并且将 ready 信号设置为真时,消费者进程(consumer process)才通过等待语句并且接收数据,等待语句使得这两个进程同步。ready 信号是一个同步信号,它通知消费者进程:生产者进程已经经过了产生 dataIn 的状态。这样,两个进程通过 ready 信号实现同步。

#### 例 3.5 消费者模块

```
module consumer(dataIn,ready);  
    input  [7:0]  dataIn;  
    input                ready;  
    reg  [7:0]  in;  
  
    always  
        begin  
            wait (ready)  
            in = dataIn;  
            // ... consume dataIn
```

```
        end
    endmodule
```

等待语句的一般形式是：

```
statement
    ::= wait statement
    | ...

wait statement
    ::= wait (expression) statement or null

statement or null
    ::= statement
    | ;
```

对表达式进行求知(evaluate),如果表达式为真,则进程继续执行语句。如果表达式为假,则进程停止执行,直至条件表达式变为真。一旦条件表达式变为真,进程会继续执行语句。请注意等待语句的尾部不含有分号;而 statement\_or\_null 后面有分号。此外,表达式的改变必须由另外的并发进程来产生。

我们要注意这样一个有趣的现象,如果在 always 语句中没有其他的事件控制或者延时操作,那么在模拟例 3.5 的时候就会出现这个问题。如果真是这样,那么一旦等待语句的条件变为真时,循环将永远执行下去,因为 wait 的条件将永远不会变为假。在某种意义上,这种问题产生的原因是:模拟器是采用顺序方式(sequential manner)模拟并发进程的,并且仅仅当遇到以下几种情形时才在所模拟的并发进程之间进行转换,即,等待条件变为假、延时到达或者遇到事件控制语句。由于上述情况在循环中不会遇到,所以模拟将会在该 always 语句中永远循环下去。

实际上,在描述并发系统中,这个问题是司空见惯的。通常,我们并不能确定进程之间彼此的相对速度,因此,我们需要引入更多的同步信号以确保能够正确执行。如果例 3.5 有另外一个同步点,例如 wait (~ready),那么本例中的生产者和消费者的工作速率将更为同步。此外,模拟也会正确运行。在下一节中,我们将用更多的示例来说明这一点。

参考:与 while 进行比较 3.3.2

### 3.3.1 一个完整的生产者和消费者握手示例

例 3.5 可能会出现一些同步错误。特别是,消费者从不会对生产者发出 dataIn 已被接收的信号。由于不完整的握手信号机制,可能会出现两种错误情况,所以有两个可能的



情形出现：

- 生产者的操作太快，并且 dataIn 的值发生变化，而消费者未来得及读取先前的 dataIn 值。
- 消费者的操作太快，以至于再次执行 always 语句时，发现 ready 信号仍为真，因此会将相同的数据读取两次。

我们需要将进程进行同步，所以不管它们的实现速度如何，它们之间仍能正常工作。一种使两个进程同步的方法是采用如图 3.1 所示的全互锁(fully-interlocked)握手信号机制。

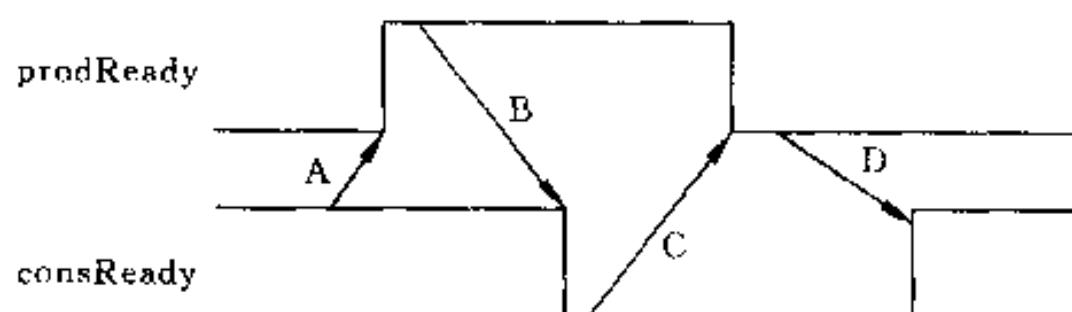


图 3.1 一个全互锁握手信号

例 3.6 和下面的段落描述了上面所示的握手信号。该描述包括两个 always 块，第一个模块建立消费者模型，第二个模块建立生产者模型。开始时，这两个模块都可以运行。当然，其中的一个模块将首先运行，这是由模拟器任意选择的。最初，所有寄存器和连线的值都是未知数。因此，当生产者模块到达“wait (consReady)”或者消费者模块到达“wait (prodReady)”时，always 块会因为 consReady 和 prodReady 未知而停止并且处于等待状态。如果消费者模块首先运行，它会在 consReady 设置为 1 后等待 prodReady 的改变。然后生产者模块运行，将 prodReady 设置为 0，然后继续等待 consReady 变为 1。如果生产者模块首先运行，它会将 prodReady 置为 0，产生一些数据，并且等待 consReady 的改变。然后消费者模块将会运行，置 consReady 为 1，并且等待 prodReady 变化。因为 consReady 刚刚被置为 1，所以生产者模块又会继续运行。

假定我们现在正处在这么一个点上：生产者将 producer-ready (prodReady) 置为假（或 0），表明它还未准备好发送信息，而消费者将 consumer-ready (consReady) 置为真（或 1），表明它已准备好接收数据。当生产者已经产生了一个值，并且它看到 consReady 为 1（图 3.1 中的箭头 A）时，它把值装入输出寄存器 dataOut 中，并且将 prodReady 置为 1。然后，它等待消费者模块接收数据并且将 consReady 置为 0。消费者模块看到 prodReady 为 1 后，将输入拷贝下来并且将 consReady 置为 0（图 3.1 中的箭头 B）。

### 例 3.6 带有全互锁握手信号的消费者的消费者

```
module ProducerConsumer;  
    reg                consReady, prodReady;
```

```

reg          [7:0]      dataInCopy, dataOut;

always          // The consumer process
begin
    consReady = 1;          // indicate consumer ready
    forever
        begin
            wait (prodReady)
            dataInCopy = dataOut;
            consReady = 0;          // indicate value consumed
            // ... munch on data
            wait (! prodReady)      // complete handshake
            consReady = 1;
        end
    end

always          // The producer process
begin
    prodReady = 0;          // indicate nothing to transfer
    forever
        begin
            // ... produce data and put into "dataOut"
            wait (consReady)      // wait for consumer ready
            dataOut = $ random;
            prodReady = 1;          // indicate ready to transfer
            wait (!consReady)      // finish handshake
            prodReady = 0;
        end
    end
endmodule

```

现在生产者得知消费者已经接收了数据,因此它将 prodReady 置为 0,以标志它们之间传输的一半工作已告结束(图 3.1 中箭头 C)。生产者继续进行它的工作,而消费者处理数据。而后,当消费者看到生产者已经完成了数据传输,通过将 consReady 置为 1(图 3.1 中箭头 D),以表明它现在已经准备好接收另一次数据传输。消费者于是等待下次传输。此时,数据传输结束。

请注意我们在 producer 中引入系统函数 random。这个函数会在每次被调用时返回

一个新的随机数。

这种方法在两个并发进程间传输数据的过程中将能正确地工作,既不管进程间的时间延时,也不管它们的相对运行速度。也就是,由于每个进程需要等待另一个进程的同步信号的相应电平(即,生产者要等待 `consReady` 和 `!consReady`),保证进程仍保持在同步锁定(lockstep)。因此,消费者不会很快地完成 `always` 循环并且重读先前的传输数据。生产者也不会因为工作速度过快而导致消费者漏过一些数据。相反,生产者要等待消费者表明它已接收到了数据。用这种方式进行同步的系统称为自定时系统,因为两个交互作用的系统总是保持相互同步,不需要像时钟这样的外部同步信号。

参考: \$ random F; wait 和 event 的比较 3.3.3

### 3.3.2 Wait 语句和 While 语句的对比

使用 `while` 语句监视外部生成的条件是不正确的。尽管状态机的最终实现可能是一个“while”(即,当 `ready` 为假时停留在状态 Q),该状态机等待由另一个并发进程产生的条件,但是在概念上我们是在使相互独立的进程进行同步,并且我们应当使用比较合适的 `wait` 语句结构。

对 `wait` 和 `while` 的不同之处的进一步探讨就涉及到模拟器的使用了。假定一台单处理器正在运行模拟器,对每一个 `always` 语句和 `initial` 语句都用一个单独的进程模拟,每次模拟一个进程。模拟器一旦开始运行一个进程,就不再停止,直到遇到一个延时控制(`#`),或者是遇到一个条件为假的 `wait` 语句,或者是遇到一个事件(`@`)语句时为止。在有延时控制、事件或者是带有 `FALSE` 条件的等待语句情况下,模拟器停止执行该进程,并在时间队列中寻找下一个进程来模拟。在 `wait` 语句的条件为真这种情况下,模拟继续执行同一个进程。而一个 `while` 语句永远不会终止模拟器执行该进程。

因此,由于例 3.7 中所示的 `while` 语句需要等待一个外部变量的改变,所以会导致模拟器进入一个无限循环。本质上讲,控制 `inputA` 的进程没有机会改变这个变量。此外,如果我们用 `wait` 语句代替 `while` 语句来修正这个循环,仍将会出现一个无限循环。因为 `wait` 语句是电平敏感型的,一旦它的条件为真,它将继续运行,除非被循环内带有 `FALSE` 条件的 `wait` 语句、事件控制语句或者延时语句终止。

要使例 3.7 中的 `wait` 语句替换正确,除非在循环体中包含一个延时语句,或者一个 `wait (FALSE)` 语句,或者一个事件控制语句。这些语句都会终止该进程的模拟而使控制 `inputA` 的进程有机会改变它的值。

#### 例 3.7 一个无限循环

```
module endlessLoop (inputA);  
    input          inputA;  
    reg [15:0]     count;
```

```

always
begin
    count = 0;
    while (inputA)
        count = count + 1; // wait for inputA to change to FALSE
    $display ("This will never print if inputA is TRUE!");
end
endmodule

```

参考：while 2.3；延时控制 2.1；事件控制 3.2

### 3.3.3 Wait 语句和事件控制语句的比较

从本质上讲，事件控制语句和 wait 语句都监视一个由外部进程产生的环境。二者之间的不同之处在于事件控制语句是边沿触发的语句，而 wait 语句则是电平敏感的语句。

因此事件控制语句适用于描述包含边沿触发逻辑的模块的模拟，例如触发器。当事件控制语句被激活时，在语句执行以前事件控制语句必须有一个变化发生。我们可以这样写：

@ (posedge clock) statement;

当控制传递到该语句时，如果时钟值为 1，将停止执行，直到再次出现 1 跳变。也就是说，事件控制语句并不认为既然时钟为 1，就一定出现一个上升沿。相反，它要等到上升沿的实际出现才开始工作。

wait 语句是对电平敏感的，当表达式为假时，它才处于等待状态。

参考：while 2.3

## 3.4 并发进程示例

3.3 节中给出的生产者和消费者的示例说明了两个进程如何通过进程间的握手信号线上等待合适的电平来进行通信和传递信息。在这一节中，我们将规定一个简单的同步总线协议，并且建立一个运用事件控制(@)结构的模拟模型。这里将使用到周期准确的描述风格。

图 3.2 给出了本例中使用的同步总线协议。clock 信号在总线上传送，并用它来同步主从总线(bus master and bus slave)的动作。一个写总线周期占用一个完整的时钟周期，而一个读周期占用两个时钟周期。我们用 rwlLine 总线指定所执行的总线周期类型，0 表示读，1 表示写。

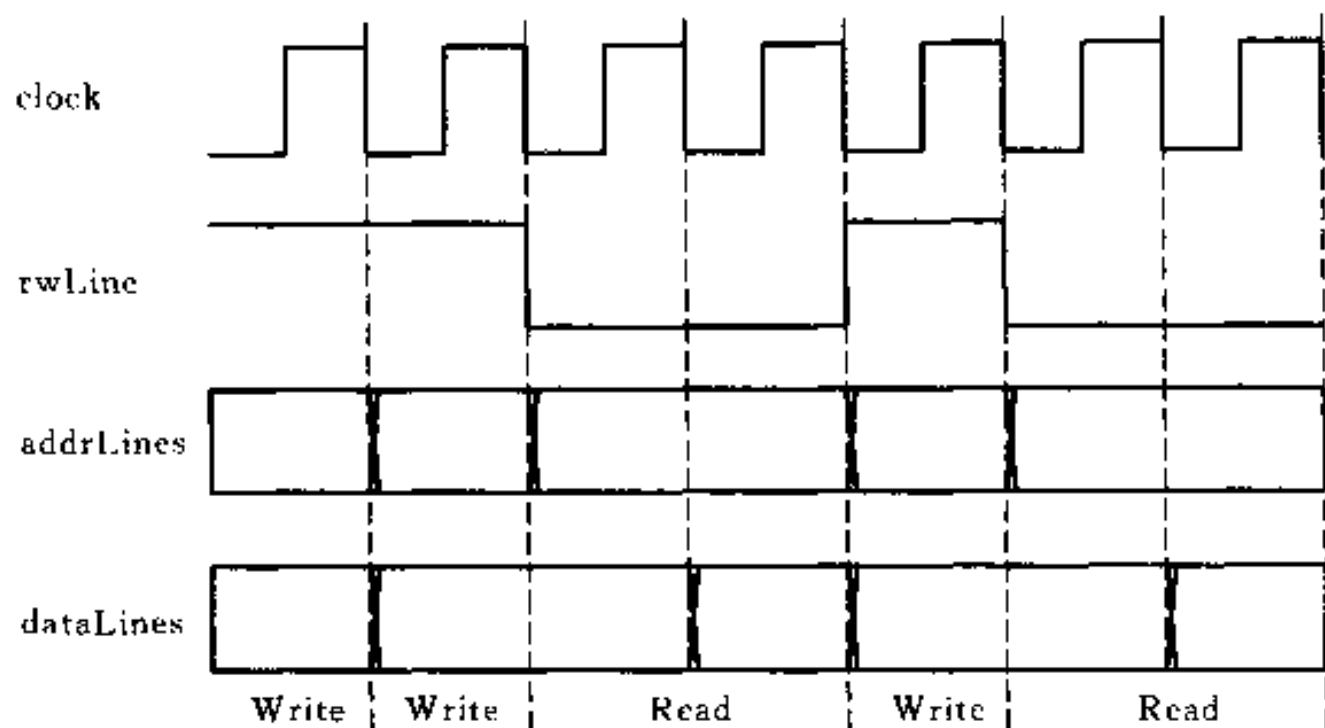


图 3.2 同步总线协议

在写周期的开始,主总线驱动 `rwLine`、`addrLines` 和 `dataLines`,并等待时钟周期的结束。在时钟周期的末尾处,所有的值都将传输到从总线上。在 `clock` 的下降沿处,从总线将 `dataLines` 上的数据装入到一个由 `addrLines` 总线指定的存储单元(memory location)中。

一个读周期占用两个时钟周期。在第一个时钟周期,主总线驱动 `rwLine` 和 `addrLines`。在第二个时钟周期,从总线驱动 `data`,`data` 总线上带有从地址总线 `addrLines` 所指示的存储单元中读取的数据。在第二个时钟的下降沿,主总线将 `dataLines` 上的数据装入内部寄存器。

在 `addrLines` 和 `dataLines` 中,标有 X 的区域显示了时钟沿处发生的数值变化。在我们的描述中,它们在 0 时间内发生改变。图中标有 X 的区域在水平轴上一定要占有物理空间。

尽管这个总线协议简单,但它却描述了一个时钟同步系统的周期精确的规范,并把一些更小的示例中描述的建模结构组合在一起。

例 3.8 是一个模块,包含由 `always` 语句和 `initial` 语句描述的 4 个进程。有三个进程对时钟、主总线和从总线进行建模。第四个进程将系统初始化,并建立起一个监视器以显示数值的改变。主总线利用 `wiggleBusLines` 任务来封装(encapsulate)总线协议的各项行为,并隐藏主总线其他行为的细节。进程通过全局变量 `clock`,`rwLine`,`addressLines` 和 `dataLines` 来进行通信,而不是通过线网(net)来完成。(例 4.18 扩展了这个示例,使得主从之间通过线网通信)。

描述中首先定义了两个系统常量 `READ` 和 `WRITE`。这种定义使得 `wiggleBusLines` 的任务调用更具有可读性。在模块 `sbus` 中定义了一个参数。参数 `tClock` 是时钟周期的一半,并设定为 20。在这里,我们可以认为它是 `tClock` 的默认值。当使用到参数的名称

时,这个值将被替代。(以后,在第4章中,会更深入地讨论参数,将看到这个类属值在初始化时将被重载)。最后,对寄存器进行定义。因为我们仅仅要在存储器 *m* 中实现 32 个 16 位的字,所以我们只定义 *addressLine* 为 5 位宽。

当两个 *always* 语句和两个 *initial* 语句的模拟开始时,它们将以任意顺序开始执行。必须写出描述以便使模拟过程在任何顺序下都能正确运行。

示例中的第一个 *initial* 语句完成三个重要功能。首先,它使用 *\$readmemb* 系统任务将外部文件“*memory.data*”装入存储器 *m* 中。该任务的操作将在以后描述。第二,将 *clock* 初始化为 0;将进程同步时使用到的数值进行初始化是很重要的。最后,*\$monitor* 语句显示 *rwLine*,*dataLines*,*addressLines* 的值,而 *\$time* 则显示这三个之中的任何一个发生改变时的实时时间。

描述中的第一个 *always* 语句仅仅每隔 *tClock* 个时间单位(即 20 个时间单位)就将时钟翻转。在运行以前,它先等待 *tClock* 个时间单位。即使这个 *always* 语句是首先运行的,它也要等到上面描述的 *initial* 语句将时钟置 0 后才去存取它的值。时间延迟将这两个语句进行排序,并确保该 *always* 语句不能对一个未知数进行求反。

### 例 3.8 同步总线的行为描述

```
`define READ 0
`define WRITE 1

module sbus;
    parameter          tClock = 20;

    reg                clock;
    reg  [15:0]        m[0:31]; // 32 16-bit words
    reg  [15:0]        data;
    // registers names xLine model the bus lines using global registers
    reg                rwLine;      // write = 1, read = 0
    reg  [4:0]         addressLines;
    reg  [15:0]        dataLines;

    initial
        begin
            $readmemb("memory.data", m);
            clock = 0;
            $monitor("rw=%d, data=%d, addr=%d at time %d",
```

```

                                rwLine, dataLines, addressLines, $time);

    end

always
    #tClock clock ~ !clock;

initial // bus master end
    begin
        #1
        wiggleBusLines ( 'READ, 2, data);
        wiggleBusLines ( 'READ, 3, data);
        data = 5;
        wiggleBusLines ( 'WRITE, 2, data);
        data = 7;
        wiggleBusLines ( 'WRITE, 3, data);
        wiggleBusLines ( 'READ, 2, data);
        wiggleBusLines ( 'READ, 3, data);
        $finish;
    end

task wiggleBusLines;
    input                readWrite;
    input    [5:0]        addr;
    inout    [15:0]       data;

    begin
        rwLine <= readWrite;
        if (readWrite) begin // write value
            addressLines <= addr;
            dataLines <= data;
        end
        else begin // read value
            addressLines <= addr;
            @(negedge clock);
        end
        @(negedge clock);
        if (~readWrite)

```

```

        data <= dataLines; // value returned during read cycle
    end
endtask

always // bus slave end
begin
    @(negedge clock);
    if (~rwLine) begin // read
        dataLines <= m[addressLines];
        @(negedge clock);
    end
    else // write
        m[addressLines] <= dataLines;
    end
end
endmodule

```

主总线进程在调用 wiggleBusLines 任务时用到三个参数,它们分别指出周期类型、存储器地址类型和数据类型。第三个参数在任务中定义为输入输出类型,用来表示在写周期中要被写入的数据,或者是在读周期中要读出的数据。这个任务被主进程调用 6 次,每次都会传递不同的值。第一次任务的调用将使 wiggleBusLines 从地址 2 中读取数据并将读取的值返回到 data 中。第三次任务调用会将地址 5 中的值写入存储器地址 2 中。

假定 clock 刚刚下降并且一个新的时钟周期正在开始,那么加载新值给主总线。如果这次总线周期是 WRITE,则执行 if 语句中的 then 部分,将传递给任务的值装入 addressLines 和 dataLines 中。然后任务等待 clock 的一个下降沿(即写周期的结束)的到来,再从任务中返回。当时钟下降沿出现时,我们知道 WRITE 周期的末尾到来,并且正如我们将要看到的,从总线将 dataLines 中的值装入到由 addressLines 指向的地址 m 中。  
#1 假定所有其他的 always 块和 initial 块首先执行。

我们来跟踪一下在写周期期间从进程的行为。开始时,从总线进程等待时钟的下降沿的出现。请记住假定时钟下降沿刚刚出现并且总线周期刚刚开始才对这些模型进行写操作。因此,语句 @(negedge clock)一直等到本周期结束,此时执行 if 语句。因为我们正在跟踪的是一个写周期,所以执行 if 语句的 else 部分,并且 dataLines 上的数据被拷贝至由 addressLines 寻址的 m 中。然后从进程等待下一个时钟周期的结束。

我们假定将对存储器进行两个背对背(back-to-back)写操作。检查一下两条 @(negedge clock)语句在写周期结束时是如何工作的对我们是有用的。一个时钟事件发生在 wiggleBusLines 任务的末尾附近,另一个则发生在从进程的开始处。两个进程都等待时钟沿的出现。一旦出现时钟沿,其中一个进程将首先执行;我们无法预知哪一个先执



行。我们现在讨论一下 dataLines。如果 wiggleBusLines 首先执行并开始第二个写周期，它会在第一个 then 部分中将 data 中的新值赋给 dataLines。如果从进程首先执行，它会将 dataLines 中的值写入存储器。那么，dataLines 中的哪个值将被写入存储器？假定这两个传输都是无阻塞的，并且传输是同步的而且顺序无关。的确，必须注意要确保数据传输的顺序无关性。在周期准确的描述中，无阻塞赋值保证了这一点。

在主模型和从模型中，读周期都要求一个额外的周期。任务 wiggleBusLines 用读取的地址装载 addressLines，并且在继续执行前等待第二个时钟周期的结束。在第二个周期结束时，dataLines 中的值被装入 data 中，并且这个值从任务中送回到主总线。

从总线等待第一个时钟周期的结束，然后从 m 的地址 addressLines 中读出数值，并放入 dataLines 中。因此，所读取的值在第二个时钟周期开始时出现。然后从总线在循环进入下一个总线周期之前等待下一个时钟下降沿事件（即读周期的结束）。

图 3.3 显示了例 3.8 的模拟结果。主总线过程驱动模拟的运行，并且调用任务 wiggleBusLines。实际上，进程从地址 2 和 3 中读取，分别对它们写入值 5 和 7，然后再从它们中读出以确保数值被正确地写入。调用系统任务 \$finish 来结束模拟。

打印功能由 initial 语句中的 \$monitor 语句控制。因为数值仅仅在时钟边沿改变，所以模拟结果的每一行显示每一个时钟周期结束时系统中的值。第一行显示的是当首先执行 \$monitor 时系统中的值。第二行显示的是当首先执行任务 wiggleBusLines 时的值（它显示系统从地址 2 中读取）。dataLines 还未被写入，因此显示为 x。下一行显示的是在读周期中两个时钟周期结束处的值。读出的值是 29（这个值对应于文件 memory.data 中的值）。沿着模拟轨迹，我们可以看到地址 2 中的 29 被第一个写操作重写为 5。值被存放在存储器中，这个事实可以从第二到最后一个操作看出来，此时地址 2 被重读。

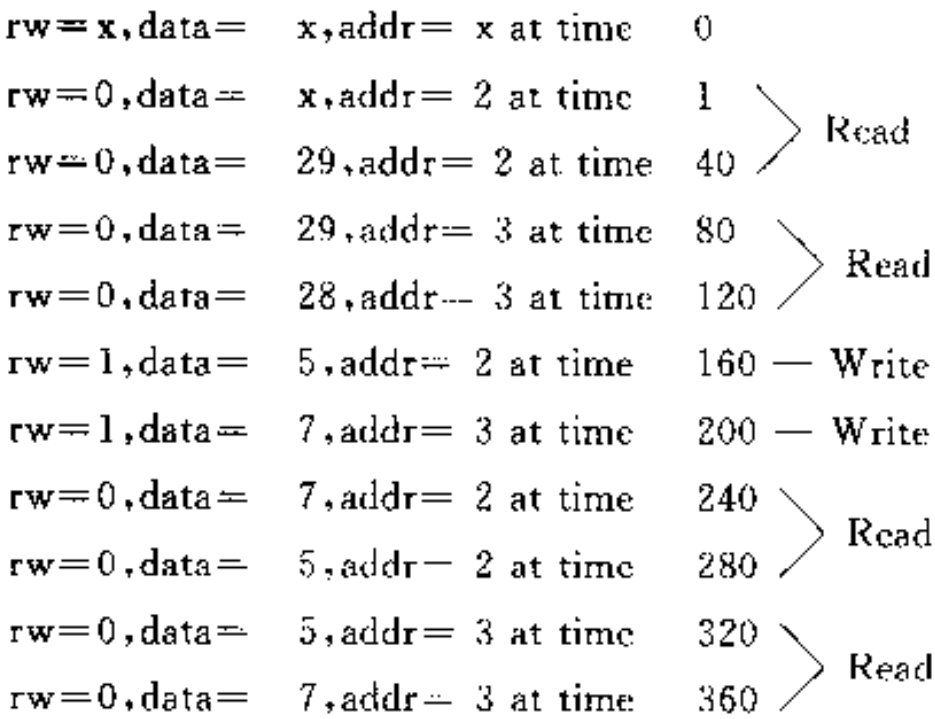


图 3.3 同步总线行为模型的模拟轨迹

这里要强调描述中的几个特征：

- 主总线进程和从总线进程被同步到时钟信号上。在时钟周期的末尾出现下降沿，主从进程开始执行。请注意这些进程不会立刻改变用来在进程之间传递信息的寄存器的值，这一点是很重要的。如果一个进程改变了任何一个寄存器的值，那么模拟的结果将取决于模拟器执行这些事件的顺序——这是不好的。无阻塞赋值保证了正确的操作。

- 存储器阵列 `m` 的初始化由外部文本使用系统任务 `$readmemh` 来完成。这种方法对于将机器指令装入处理器的模拟模型中和初始化存储器中的数据以及装入测试向量是很有用的，这些测试向量将被应用到该系统其他部分。在这里，任务 `$readmemh` 从文件 `memory.data` 中读取以空格分隔的十六进制数，并将其连续装入从地址 0 开始的存储单元中。更详细的细节和选项请参阅附录 F.8。

- 请注意 `READ` 和 `WRITE` 被定义为常量，而 `tClock` 被定义为参数。参数为模块的使用提供了一种指定默认值的方法。但是，当例示模块时，参数值可以被覆盖 (override)。4.5 节将对参数作进一步的讨论。

- 在一条语句中，我们使用操作符“`!`”来规定 `clock` 的补码，而在另一条语句中，我们使用运算符“`~`”来规定 `rwLine` 的补码。在这种上下文环境中，二者都是正确的，因为被取反的值是一位。“`~`”规定对其操作数逐位取反 (即，`~4'b0101` 是 `4'b1010`)。“`!`”则对其操作数的值取反。假定操作数是多位的，那么，如果一个值是 0 (假)，“`!`”取反为真。如果多位操作数非 0 (真)，“`!`”取反为假。因此 `! 4'b0101` 为假。

参考：参数 4.5；`$readmemh` F.8；Verilog 运算符 C；任务 2.5.1；寄存器规范 E.1；存储器规范 E.2

## 3.5 简单流水线处理器

本节给出另一个并发处理系统的示例。这里，我们根据第 2 章中描述的 Mark-1 设计一个非常简单的流水线处理器。尽管这并不代表当前处理器的复杂度，但这种基本方法对于建立这样的处理器模型会有所启示。

### 3.5.1 基本处理器

在例 3.9 中，我们用周期精确的表示方法给出了处理器的模型。这种设计处理器的模型在抽象层次上允许设计者掌握在每个时钟周期中会出现什么功能，该功能在处理器流水线的其他级 (stage) 怎样受并发行为的影响，并且至少可以根据时钟周期来判断机器性能。

这个示例由两个 `always` 块构成，每个 `always` 块构成这个简单处理器的一级。第一个 `always` 块构成处理器的第一级，用来取指令。第二个 `always` 块构成处理器的第二级，用来执行指令。因为每一级处理器由一个 `always` 块描述，所以我们在处理器的流水线之

间建立起并发关系。

设计中使用无阻塞赋值语句将状态的更新同步到时钟边沿 *ck* 上。使用无阻塞赋值语句时,请记住整个设计中(此处指两个 *always* 块)所有赋值语句的右边要在左边被更新以前求值,这一点是很重要的。在本例中,请注意指令寄存器(*ir*)在第一个 *always* 块中加载,而在第二个 *always* 块中存取。因为所有的存取都是由无阻塞赋值语句来实现的,所以我们知道用来加载指令寄存器的指令的存取不会和第二个 *always* 块中指令的执行相互干扰。第二个 *always* 块的右边在 *ir* 被第一个 *always* 块更新以前要被求值。

### 例 3.9 流水线处理器

```
module mark1Pipe;
    reg[15:0]  m[0:8191]; // 8192 x 16 位存储器
    reg[12:0]  pc;        // 13 位程序计数器
    reg[15:0]  acc;       // 15 位累加器
    reg[15:0]  ir;        // 16 位指令寄存器
    reg        ck;        // 时钟信号

    always @(posedge ck) begin
        ir <= m[pc];
        pc <= pc + 1;
    end

    always @(posedge ck)
        case (ir[15:13])
            3'b000: pc <= m[ir[12:0]];
            3'b001: pc <= pc + m[ir[12:0]];
            3'b010: acc <= -m[ir[12:0]];
            3'b011: m[ir[12:0]] <= acc;
            3'b100:
            3'b101: acc <= acc - m[ir[12:0]];
            3'b110: if (acc < 0) pc <= pc + 1;
        endcase
endmodule
```

但是,无阻塞赋值语句并不能解决同步中的所有问题。请注意运行某些指令时,*pc* 被装入到两个 *always* 块中——指令 0 就是这种情况。需要考虑的问题是,*pc* 中的哪个更新会首先出现:是第一个 *always* 块中的更新,还是第二个?当然,更新的顺序是不明确的,所以我们需要更改这个描述来获得正确的操作。

### 3.5.2 流水线之间的同步

在一般的情况下,由两个分开的 always 块写入同一寄存器中会导致不确定的值装入寄存器中。如果一个模型可以保证两个 always 块不会在同一时刻(即在相同的状态或时钟时间)写入同一寄存器中,那么从两个 always 块写入同一寄存器是一个特别有效的方法。尽管如此,在这里 pc 在每个状态由取进程写入,而在某些状态由一些执行进程写入。

#### 例 3.10 各阶段的同步

```
module mark1PipeStage;
    reg [15:0]    m [0:8191];    // 8192 x 16 bit memory
    reg [12:0]    pc, ptemp;     // 13 bit program counter and temporary
    reg [12:0]    acc;           // 13 bit accumulator
    reg [15:0]    ir;            // 16 bit instruction register
    reg          ck, skip;

    always @(posedge ck) begin    // fetch process
        if (skip)
            pc = ptemp;
        ir <= m [pc];
        pc <= pc + 1;
    end

    always @(posedge ck) begin    // execute process
        if (skip)
            skip <= 0;
        else
            case (ir [15:13])
                3'b000 :    begin
                                ptemp <= m[ir[12:0]];
                                skip <= 1;
                            end
                3'b001 :    begin
                                ptemp <= pc + m[ir[12:0]];
                                skip <= 1;
                            end
                3'b010 :    acc <= -m[ir[12:0]];
                3'b011 :    m[ir[12:0]] <= acc;
            endcase
    end
endmodule
```

```

        3'b100,
        3'b101 :    acc <= acc - m[ir[12:0]];
        3'b110 :    if (acc < 0) begin
                        ptemp <= pc + 1;
                        skip <= 1;
                    end
            endcase
        end
    endmodule

```

例 3.10 通过加入寄存器 ptemp 来修正这个问题。当 pc 仅由取指令阶段(fetch stage)写入时,这个寄存器仅由执行阶段(exccute stage)写入。在有分支指令执行的情况下,分支目标写入 ptemp。同时,取出下一条顺序指令,并且 pc 由取指令阶段加 1。但是,因为一条分支指令正在被执行,该指令和 pc 的加 1 是没有必要的。执行阶段设置一个独立的指示寄存器 skip 来指出一条分支指令已经出现,并且下一条指令应该从 m[ptemp]取出而不是从 m[pc]中取出。此外,因为分支之后的指令已经取出,skip 也控制执行阶段以防止它被执行。

在这个示例中,除了一个赋值语句之外其他语句都是无阻塞赋值语句。在取指令过程中,pc 由一个无阻塞赋值语句赋值,以便将它可以使用在随后的进程语句中。

还有其他方法来修正这个问题,包括在取指令阶段复制 case(ir)语句,以便当分支出现时,pc 有条件地装入一个分支目标。但是,执行阶段仍将需要跳过多余的指令。

## 3.6 有名块的终止

在例 2.4 中,我们给出了怎样使用 disable 语句来跳出循环,或者继续执行循环的下次迭代。使用同样的方法,disable 语句也适用于并发进程的情形中。本质上,disable 语句用来终止(或停止)任何有名的 begin-end 块的执行——然后继续执行紧跟在 begin-end 块后的下一条语句。这个 begin-end 块可以处在包含 disable 语句的进程之中,也可以不在其中。如果被终止的块不在局部作用域(local scope)或向上作用域(upward scope)之内,那么就需要有层次化名称。

为了说明如何终止一个并发进程,我们返回到 1.6 节中的预定的行为示例中,它已经被例 3.11 扩展了。加入一个 reset 输入和一个初始化语句,在 always 块中加入了一条 wait 语句。这些加入的语句为周期精确的规范提供一个异步低电平有效的复位信号(reset)。

现在讨论模块是如何工作的。在开始时,initial 块和 always 块都可以执行。其中的

一个停下来等待 reset 的下降沿出现,而另一个等待 reset 变为 TRUE。如果我们假定 reset 信号是无效值(即它的值为 1),那么 always 块开始执行它的周期准确的描述。在某一时刻,使 reset 有效(即它变为 0),并且它的下降沿激活 initial 块。initial 块终止 main。main 是 always 块中的 begin-end 块的名称。不管 main 块中的哪一部分正在被执行,它都要退出,并且 always 块重新开始执行。开始时,第一条语句等待 reset 变为 TRUE……无效。因此,当 reset 是有效值时,main 块被停止,它并不马上重新开始,而要等到 reset 变为无效值。

### 例 3.11 使用预定行为方法的描述

```
module simpleTutorialWithReset (clock, reset, y, x);
    input          clock, reset;
    output [7:0]    x, y;
    reg [7:0]       x, y, i;

    initial
        forever begin
            @(negedge reset)
                disable main;
        end

    always begin; main
        wait (reset);
        @(posedge clock) x <= 0;
        i = 0;
        while (i <= 10) begin
            @(posedge clock);
            x <= x + y;
            i = i + 1;
        end
        @(posedge clock);
        if (x < 0)
            y <= 0;
        else x <= 0;
    end
endmodule
```

一个块的命名如示例中所示。块的声明是一条语句,一般形式为:

```

statement
    ::= seq_block
    | ...

seq_block
    ::= begin [block_identifier {block_item_declaration}]
           {statement}
           end

block_item_declaration
    ::= parameter_declaration
    | reg_declaration
    | integer_declaration
    | real_declaration
    | time_declaration
    | realtime_declaration
    | event_declaration

```

请注意一个有名块的引入也是允许有可选择的块声明。在这一点上,其他参数和寄存器可以在块的范围内定义。

`disable` 语句不仅停止有名块,而且也停止块中调用的函数和任务。同样,任何被调用的函数和任务也会停止。接下来,继续执行该模块的下一条语句。如果你终止了正在执行中的任务(或函数),那么就从该任务(或函数)中返回。

指出什么不被 `disable` 语句终止也是有意义的。如果一个被终止的有名块通过更改一个值或者通过触发一个有名事件,触发一个事件控制,那么监视这些事件的进程将被触发。它们不会被 `disable` 语句终止。

当我们定义术语进程时,强调的是是一个独立的控制线程。控制的实现是不相关的;它可以是一个微代码控制器、一个简单的状态机,等等。在例 3.11 的情形中,如果我们假定实现 `always` 语句的控制器的第一个状态编码为 0,那么 `initial` 块可以实现为 `always` 控制器中的状态寄存器的一个异步复位。即 `initial` 语句不像一个状态机,而有些像一个简单的复位逻辑。关键是,不管两个进程如何实现,系统中有两个独立的行为能够改变状态。每个行为都是活动的(active),并且彼此独立工作。

参考: `always2.1`; 循环中的 `disable` 语句 2.3.2; 并行块 3.9; 层次化名称 2.6; 标识符的作用域 2.6

### 3.7 赋值语句内部控制和定时事件

首先规定在本例中使用的多数控制和定时事件应出现在行为或赋值以前。我们描述

过这样的语句：

```
# 25 a=b;
```

或

```
@(posedge w) q=r;
```

它们分别执行的动作是延迟 25 个时间单位后将 b 的值赋给 a；一直延迟到 w 上出现一个上升沿，然后将 r 的值赋给 q。这些行为语句的共同之处是“延迟”（或者是 #，或者是 @）都在赋值语句执行以前出现。的确，语句的右边是在“延迟”周期之后才求值的。**赋值语句内部**（intra\_assignment）定时控制允许“延迟”出现在赋值语句内——在右边被求值和左边被赋值期间。从概念上讲，这些赋值有一个主从特征：输入被采样，出现延时，然后对输出进行赋值。

赋值语句用延迟或事件控制规范来书写，赋值语句的中间紧紧跟随“=”。这使得阅读赋值语句时感觉很直观。假定等号右边先被求值后再赋值给左边，那么在赋值语句中间的延时或事件控制提醒读者在从右向左赋值时必须先延时。尽管这里所有的示例都是用无阻塞赋值，但是赋值语句内部控制和定时事件也可以用无阻塞赋值。以上语句的赋值语句内部定时控制的格式如表 3.1 所示。

表 3.1 内部赋值控制和定时事件

使用内部赋值结构的语句	不含内部赋值的等价语句
a = # 25 b;	begin bTemp = b; # 25 a = bTemp; end
q = @(posedge w) r;	begin rTemp = r; @(posedge w) q = rTemp; end
w = repeat (2) @(posedge clock) t;	begin tTemp = t; repeat (2) @(posedge clock); w = tTemp; end

前两个赋值语句执行的行为分别是对 b 进行求值并将值存入一个临时地址，延时 25



个时间单位,然后将所存储的值赋给 a;对 r 进行求值并将值存入一个临时地址,等待 w 的下一个上升沿,然后将临时值存入 q 中。它们对应于上面的说明。第三个赋值语句表示赋值语句内部,重复上面未说明的部分。右边先被计算并且将值赋给一个临时地址。延时完成后(在这里等待 clock 的两个上升沿),将值赋给 w。因此,每一条语句都将赋值语句右边值临时拷贝,以后赋给左边。

右边的拷贝实际上被存入模拟器的事件队列中,并且不能用于其他目的。

对于阻塞赋值语句和无阻塞赋值语句,内赋值语句有三种形式,描述如下:

statement

```
 ::= blocking_assignment
 | non_blocking_assignment;
 | ...
```

blocking\_assignment

```
 ::= reg_lvalue = [ delay_or_event_control ] expression
```

non\_blocking\_assignment

```
 ::= reg_lvalue <= [ delay_or_event_control ] expression
```

delay\_or\_event\_control

```
 ::= delay_control
 | event_control
 | repeat ( expression ) event_control
```

delay\_control

```
 ::= # delay_value
 | # ( mintypmax_expression )
```

event\_control

```
 ::= @ event_identifier
 | @ ( event_expression )
```

event\_expression

```
 ::= expression
 | event_identifier
 | posedge_expression
 | negedge_expression
 | event_expression or event_expression
```

内赋值定时控制使用在 D 触发器的规范中。这种方法使用语句:

```
@(posedge clock) q = #10 d;
```

这个语句给触发器的行为提供了一个主从特征。该模型在时钟边沿对 d 的值进行采样,10 个单位后对它进行赋值。但是,下面的语句不能精确地建立一个触发器的模型。

```
q = @(posedge clock) d;
```

无论何时执行这个语句,该语句都对 d 的值进行采样。然后当 clock 的上升沿出现时,该值赋给 q。假定 d 的初始值在 clock 的边沿出现以前已被完全采样,那么触发器的正常行为不会被捕捉到。

参考: 不确定机制 5.3

### 3.8 过程持续赋值

前面章节给出的持续赋值语句允许描述组合逻辑,无论何时只要任何一个输入改变,就会计算组合逻辑的输出。持续赋值语句有一种过程化形式(procedural version),允许在某一特定时间内对寄存器进行持续赋值。因为赋值不会永远有效,持续赋值也是如此,我们称之为**过程持续赋值**(procedural continuous assignment)语句(在手册的早期版本中称为“准持续(quasi-continuous)”)。当过程持续赋值语句有效时,语句的动作就会像连续赋值一样。

考虑例 3.12 所示的对寄存器进行预置位清零的示例。首先请注意持续赋值和过程持续赋值的不同之处在上下文中是显而易见的,过程持续赋值是一个过程语句,只有当控制权传递给它时才执行(持续赋值语句一直是有效的(active),无论何时输入一改变,输出就随着改变)。本例中,第一个 always 语句描述了一个进程,该进程对于信号 clear 或者信号 preset 的变化起反应。如果 clear 变为 0,那么我们将寄存器 q 赋值为 0。如果 preset 变为 0,那么我们将寄存器 q 赋值为 1。当发生一个变化并且二者都不为 0 时,我们对 q 进行重新赋值(deassign)(本质上取消先前的过程持续赋值语句的动作),然后用第二个 always 语句描述的标准时钟法将一个值装入 q。

#### 例 3.12 带有过程持续赋值语句的触发器

```
module dFlop (preset, clear, q, clock, d);  
    input      preset, clear, clock, d;  
    output     q;  
    reg        q;  
  
    always
```

```

    @(clear or preset)
    begin
        if (!clear)
            #10      assign q = 0;
        else if (!preset)
            #10      assign q = 1;
        else
            #10      deassign q;
        end

    always
        @(negedge clock)
            q = #10 d;
endmodule

```

赋值语句的一般形式是：

```

statement
    ::= procedural..continuous..assignments

procedural..continuous..assignment
    ::= assign reg..assignment;
    |   deassign reg..assignment;
    |   ...

reg..assignment
    ::= reg lvalue = expression

```

请注意过程持续赋值语句过载一个正常的过程语句到一个寄存器中,这一点是很重要的。当过程持续赋值语句有效时,reg..assignment 像一个持续赋值语句一样工作。因此,即使在第二个 always 语句中有下降沿出现,d 对 q 的过程赋值也不会生效。此外,如果一个过程持续赋值语句的右边值发生改变(在上面的示例中它不是一个常量),那么左边也会随着改变。重新赋值后,过程持续赋的值保留在寄存器中。

参考:持续赋值 4.4; 过程赋值 2.1; 事件控制语句 3.2.1

### 3.9 顺序模块和并行模块

到目前为止我们所看到的 begin-end 块都是顺序块的示例。尽管它们的主要用途是将多个过程语句组合成为一个复合句,当 begin-end 块被命名时,它们也允许对参数、寄

存器和事件说明进行新的定义。因此,可以在命名的 begin-end 块内对新的局部变量进行规定和存取。

顺序的 begin-end 块的其他形式是下面要讨论的并行块或 fork-join 块。fork-join 块中的每一条语句都是一个独立的进程,该进程在控制权传递到 fork 时开始运行。join 要等到所有的进程都完成之后才继续执行 fork-join 块之后的下一条语句。

下面的示例描述了一个重新启动进程的异步复位。resetSequence 对寄存器进行初始化,然后开始运行命名为 mainWork 的 fork-join 块。fork 的第一条语句是一个 forever 循环,它描述了微处理器的主要行为。第二条语句是一个进程,它监视 reset 信号的上升沿。当 reset 的上升沿出现时,mainWork 块将被终止。如前面所描述,当一个块被终止时,有名块内的所有语句都被终止,而继续执行下一条语句,在这种情况下,always 语句的下一个迭代开始。因此,不管在系统的 fetch 和 execute 行为发生什么情况,reset 都能够异步地重新启动整个系统。

### 例 3.13 fork-join 块的说明

```
module microprocessor;
    always
        begin
            resetSequence;
            fork;   mainWork
                forever
                    fetchAndExecuteInstructions;
                @(posedge reset)
                    disable mainWork;
            join
        end
endmodule
```

下面给出并行块的一般形式。正如先前所描述的有名(顺序)块一样,对块进行命名允许具有选择项的块声明,可以为块的作用域引入新的名称。

```
statement
    ::= par block
    | ...

par block
    ::= fork [ :block-identifier { block-item-declaration } ]
        { statement }
    join
```

```

block item_declaration
    ::= parameter_declaration
    |   reg_declaration
    |   integer_declaration
    |   real_declaration
    |   time_declaration
    |   realtime_declaration
    |   event_declaration

```

例 3.14 显示了一个 fork-join 块的不太抽象的使用方法。例 3.11 在这里改写,采用包括 fork-join 块的单独的一个 always 语句。

此外,请注意我们将 fork-join 块中的每一条语句看作是一个独立的进程,这一点很重要。从本质上,此例用一个含有 fork-join 的 always 语句取代了两个 always 语句。通过和例 3.11 的对比,可以加强概念,会进一步认识到在 fork-join 中的每一条语句至少在概念上应当被看作一个独立的进程。

#### 例 3.14 一个简单的 fork-join 教学示例

```

module simpleTutorialWithReset (clock, reset, y, x);
    input          clock, reset;
    output [7:0]    x, y;
    reg [7:0]       x, y, i;

    always fork; main
        @(negedge reset)
            disable main;
    begin
        wait (reset);
        @(posedge clock) x <= 0;
        i = 0;
        while (i <= 10) begin
            @(posedge clock);
            x <= x + y;
            i = i + 1;
        end
        @(posedge clock);
        if (x < 0)
            y <= 0;
    end
end

```

```

        else    x <= 0;
    end
join
endmodule

```

参考：有名块的终止 3.6

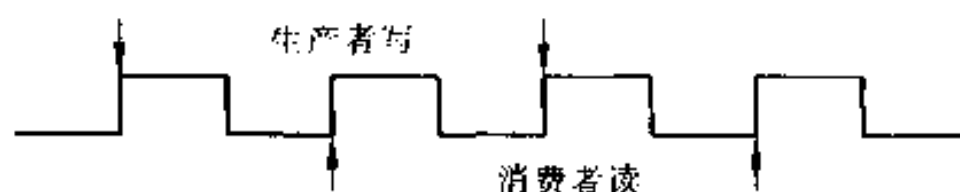
## 3.10 练习

3.1 下面两组 Verilog 代码会产生相同的行为吗？为什么？

<pre> ... @(posedge exp)     #1 statement1; ... </pre>	<pre> ... wait(exp)     #1 statement1; ... </pre>
--	---

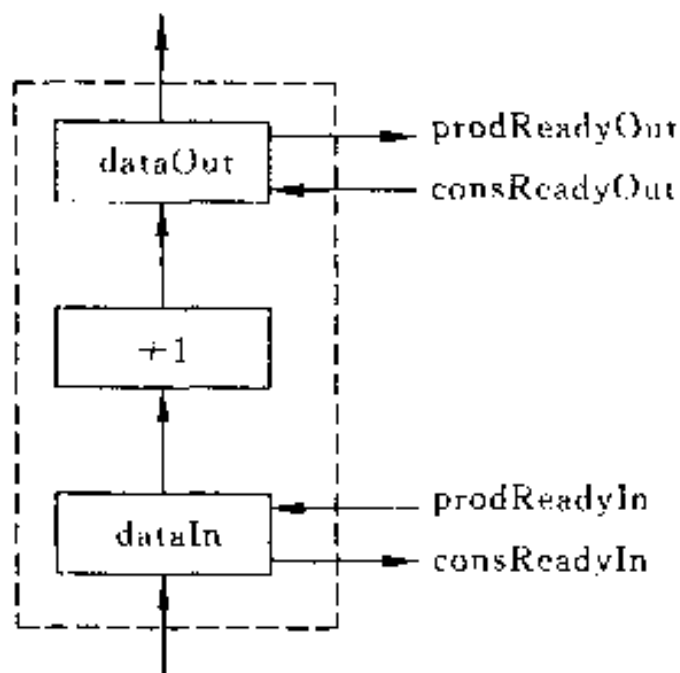
3.2 在行为级上改写例 3.6 中的生产者和消费者模型，让公共时钟信号控制模块之间数据传输的时序，在连续的时钟上升沿，将要发生以下情况：

1) 生产者在它的输出上建立数据，2) 消费者读取数据，3) 生产者建立它的下一个数据值，等等。



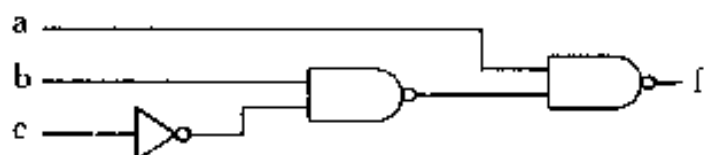
为使设计正确无误，应该加上一个合适的加电(power-on)初始化机制。请找出一个方案并包含在模型描述中。

3.3 扩展例 3.6，使得在生产者和消费者之间包含内部处理，这是一个带有 10 个时间单位延迟的简单加 1 操作。将这个模型的一个示例连接到一个循环中以便数据在这



个循环中永不停止地流动,每次循环时数据都加 1。加上附加代码是为了执行时对模型进行初始化。

- 3.4 考虑四个有名事件:  $e_1, e_2, e_3$  和  $c$ , 写出一个描述, 使得当  $c_1, c_2, c_3$  以严格顺序出现后触发事件  $c$ 。即, 如果任一事件的执行顺序颠倒, 则次序重新排列。然后, 写出一个描述, 使得当  $e_1, e_2, e_3$  不管以什么顺序分别出现三次后触发事件  $e$ 。
- 3.5 下面的组合逻辑块具有三个输入和一个输出。该电路使用某种螺旋式技术来实现并加以分析。现在我们希望向电路中插入正确的输入输出时序信息(无需纠正中间结点的时序)。



下面是必须体现在电路中的时序关系:

- $a$  或  $b$  与输出端  $f$  之间上升沿或下降沿的延迟: 15 个时间单位
- $c$  与输出端  $f$  之间上升沿或下降沿的延迟: 10 个时间单位

以上的时序关系出现在逻辑图中是非常奇特的, 但是这是一种螺旋式的技术, 是一种晶体管实现, 用于某种奇特但实用的时序延迟。

假定  $a, b, c$  是同步定时触发器的输出。用 Verilog 写出功能和时序都正确的行为描述。

- 3.6 针对例 3.10 中的流水线处理器, 在取指令级加入指令译码逻辑部分(就像执行级中的 case 语句)。使用该逻辑取代变量 `skip` 来确定如何装载 `pc`。
- 3.7 针对例 3.10 中的 `mark1PipeStage` 模块, 编写一段初始化语句, 用来装载处理器的存储器, 并执行几条指令。如果需要的话, 加入监视语句与其他的初始化操作和时钟控制。用指定的机器指令编写一段程序, 执行下面的伪代码:

```
if ((m[4] - m[5]) < 0)
    m[4] = 17;
else
    m[4] = -10;
```

- 3.8 针对例 3.10 中的流水线, 加入第三级, 在第二级仅仅只取操作数; 从存储器中读出的值放到存储器数据寄存器(`mdr`)中, 第三级执行下面的指令, 装载 `acc`, `pctemp` 或 `m` 中的结果值(`mdr`), 假定该存储器具有多个读端口和写端口。注意处理可能会引起级与级之间的冲突。

## 第4章 逻辑级建模

关于逻辑级建模,我们主要关心的是数字系统的行为模型。行为模型更多地考虑到一个模块的抽象功能描述,而不考虑其具体实现。逻辑级建模用来为模块建立逻辑结构,指明其端口、子模块、逻辑功能,以及与其实现直接对应的互连方式。Verilog 允许我们描述系统的逻辑功能和结构。本章主要介绍 Verilog 的这一特点。

### 4.1 引言

数字系统的逻辑建模有多种途径。每一种途径都代表逻辑建模的一个子层次,强调一个模块不同方面的特征。

电路的门级模型主要是从诸如与、或、异或之类的逻辑基元互连的角度描述电路。在这一层次建模,设计人员可以通过元件来描述设计的逻辑实现,而这些元件是可以在工艺库和数据手册中查找到的,设计人员从而可以分析设计的时序和功能正确性等特征。正是由于门级建模的作用如此深远,Verilog 语言为标准逻辑函数提供了门级基元。

数字系统的结构基模型使用 Verilog 的模块定义来描述元件,这些元件由其他模块和门基元组成,它们可以任意复杂。正如我们在本书前面的示例中已经看到的,一个结构模型可能包含行为建模语句(一条 always 语句),持续赋值语句(赋值语句),还可能引用其他模块或是门级基元的模块例示语句,以及所有这些语句的组合。通过使用模块定义来描述复杂模块,设计人员能够更好地控制设计的复杂性。例如,通过把互连的门级单元集合封装在一个单独的模块中来实现一个算术逻辑单元(ALU),在很大程度上简化了设计描述,提高了可读性与可理解性。

还有一个更为抽象的方法来描述设计的组合逻辑,即通过持续赋值语句进行描述。这种方法使得逻辑函数可以用一种类似于布尔代数的形式来表示。持续赋值语句有代表性地描述了组合逻辑模块的行为,而非其实现。

最后,Verilog 语言还允许从晶体管开关级描述电路。在这一级,Verilog 语言提供了基本的 MOS 和 CMOS 晶体管,使得设计人员可以用这些器件从逻辑上实现一些电气特征。

为了帮助设计人员编写和阅读逻辑级的模型,首先必须了解这些模型是如何在模拟器上运行的。这种建模方式所用的基本数据类型是线网(net),线网是由门和连续赋值语句驱动的。这些线网还可以作为其他门和连续赋值语句的输入,就像一个连续赋值语句



的右项一样。任何时候当门和连续赋值语句的输入发生变化时,其输出会被求值,输出端的变化再通过输出网传送到其他的输入,在传输的过程中可能会有一定的延迟。我们将这种输入改变时更新输出的方式称为 Verilog 门级时序模型,有关这一点在第 5 章中将进一步讨论。

与此相反,在行为建模中的过程赋值语句只有当控制被传递给它时才执行。因此在过程赋值语句右项的线网发生变化时,并不意味着该语句一定会执行。输入必须有一个事件,或者有一个被触发的等待语句,才会执行行为模型中过程赋值语句。

Verilog 语言为设计人员提供了不同的系统描述方式,允许在设计人员所要求的层次上描述系统。有关描述系统逻辑级功能和结构的不同方式,在本章和后两章中将有介绍。

参见:与过程赋值语句的比较 2.1;门级时序模型 5.1

## 4.2 逻辑门与线网

我们首先在逻辑门级为系统建模。Verilog 提供了一个由该语言定义的 26 条门级基元组成的集合。根据这些基元,可以通过利用线网连接这些门,并将它们封装进模块,从而建立更大的功能模块。当在门级描述一个电路时,我们尽量保持与实际门级实现紧密的一致性。

### 4.2.1 用基元逻辑门建模

例 4.1 是一个全加器的结构模型,使用了一些 Verilog 的门级基元。该例是从 CMOS 一位全加器的数据手册描述推导出来的。三个一位输入和两个一位输出提供了与外部的接口。为了进入模块内部进行描述,我们在这里列出了 11 条基元逻辑模块示例来组成该全加器。图 4.1 是该全加器的内部连接。为了方便比较,我们加上一些标号。需要指出的是,其中有两个与非门,一个输出是 x2(注意门级基元的第一个参数是其输出),输入是 aIn 和 bIn。而另一个输出是 cOut,输入是 x2 和 x8。

#### 例 4.1 一位全加器

```
module fullAdder(cOut,sum,aIn,bIn,cIn);  
    output    cOut,sum;  
    input     aIn,bIn,cIn;  
  
    wire      x2;  
  
    nand      (x2,aIn,bIn),  
              (cOut,x2,x8);
```

```

    xnor      (x9,x5,x6);
    nor       (x5,x1,x3),
              (x1,aIn,bIn);
    or        (x8,x1,x7);
    not       (sum,x9),
              (x3,x2),
              (x6,x4),
              (x4,cIn),
              (x7,x6);

endmodule

```

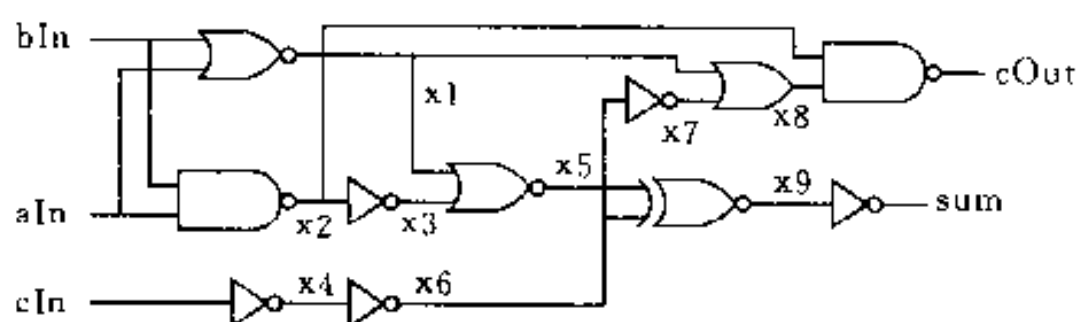


图 4.1 一位全加器

门例示的一般语法如下：

gate instantiation

```

::=      n input_ gatetype [drive_ strength] [delay2] n input_ gate instance {,
        n input_ gate instance};
|
...

```

n input\_ gatetype

```

::=      and | nand | or | nor | xor | xnor

```

n input\_ gate\_instance

```

::=      [name_ of_ gate_instance] (output_ terminal,input terminal[,input terminal])

```

name\_ of\_ gate\_instance

```

::=      gate Instance_identifier[range]

```

input\_ terminal

```

::=      scalar expression

```

output terminal

```

::=      terminal identifier
|        terminal identifier[constant-expression]

```

其中 `n input gatetype` 指明了上面所列出的门级基元之一,可选项驱动强度(`drive strength`)指明了门输出的电气特性,可选项 `delay` 指明该示例所用的模拟门延迟,门示例列表是一个由逗号隔开的每一个门示例的终端(`terminals`),在这些终端前面还有可选的示例名。在默认情况下强度为 `strong0` 和 `strong1`。默认延迟为 0。在第 9 章将进一步讨论有关强度的问题,在 4.7 节和 4.8 节中将进一步讨论有关延迟建模的问题。

应该注意的是,上面说明的公式并没有包括例 4.1 中的非门。非门与缓冲门可以有任意多个输出(首先列出),但只有一个输入,通常用下面的形式来描述:

```

gate instantiation
::=      n output gatetype[drive strength][delay2]n output gate instance{,
        n output gate instance};
|        ...

n output gate instance
::=      [name of gate instance](output terminal,{,output terminal},input terminal)

n output gatetype
::=      buf | not

```

在例 4.1 中并没有为任何的门示例命名。可以将语句改用下面的形式,来命名例 4.1 中的两个与非门:

```

nand      John (x2,aIn,bIn),
          Holland (cOut,x2,x8);

```

或者当已经为 John 和 Holland 指定了强度为 `strong0` 和 `strong1` 的驱动,以及一个三个单元时间的门延迟:

```

nand      (strong0,strong1) #3
          John (x2,aIn,bIn),
          Holland (cOut,x2,x8);

```

驱动强度和延迟说明了门例示的合理性。当给定了这两个合理性描述词中的一个或是全部时,它们作用于表中定义的所有例示,这些例示是由逗号隔开的。为了改变这些合理性描述词,门例示列表必须先以一个“;”结束,然后重新开始。

表 4.1 给出了预定义门级基元的完整列表。在本章中着重考虑前三列的基元,这些基元是构成它们的晶体管的逻辑抽象,后三列的基元允许我们在晶体管开关级建模,在第

9 章中有晶体管开关级元素的详细介绍。

表 4.1 门和开关级基元

n_ input gates	n_ output gates	tristate gates	pull gates	MOS switches	bidirectional switches
and	buf	bufif0	pullup	nmos	tran
nand	not	bufif1	pulldown	pmos	tranif0
nor		notif0		cmos	tranif1
or		notif1		rnmos	rtran
xor				rpmos	rtranif0
xnor				rcmos	rtranif1

在表 4.1 中第一列的门基元实现了所列出的标准逻辑功能。在第二列中, buf 门是一个不可反转的缓冲器,而 bufif 门 notif 门提供了允许三态输入的缓冲(buf)和非(not)功能。bufif0 门当其使能端为 0 时驱动输出,当其使能端为 1 时呈现高阻态。在附录 D 中有关于 Verilog 中四级逻辑真值表(用 0,1,x,z)的说明。

对于第一列中的门级基元,门例示中的第一个标识符为单独的输出端口,或是双向端口,其他的所有标识符都是有关输入的。可以有任意多个输入项。在第二列中, buf 门和 not 门可以有任意多个输出,但只有一个输入,这个唯一的输入放在最后。

尽管在第 9 章中会详细讨论有关驱动强度的概念,在此也有必要指出一点,即只需对前三列中的门指明驱动强度。

参见: Verilog 单元门 D; 四值逻辑 4.2.2; 强度 9.2; 延迟说明 4.7; 开关级门单元 9; 用户自定义基元 8

4.2.2 四级逻辑值

门的输出驱动与其连接的其他门和模块的线网。门对线网的驱动值可以取下列值:

- 0 代表逻辑 0,或者是条件 FALSE
- 1 代表逻辑 1,或者是条件 TRUE
- x 代表未知逻辑值(0,1,z 中的任意一项,或是某种变化状态)
- z 代表高阻状态

值 0 和值 1 是逻辑互补的,值 x 解释为“或者为 0,或者为 1,或者为 z,或者是处于某种变化状态”,而值 z 则是高阻状态。当在门的输入或者是在表达式中有 z 值时,它的作用与 x 值是一样的。值得强调的是,即使是在行为模型中的寄存器都是基于位来存放这四个逻辑值。

辑值。

每一个基元门(primitive gate)都是以这四个逻辑值的形式来定义的。表 4.2 是与门的定义。应该注意的是,当与门的某个输入为 0 时,与门的输出会强制为 0,而不管其他的输入是什么形式(即使它是 x 或者 z)。

表 4.2 AND 基元的四值逻辑定义

AND	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

参考：四值门的定义 D

4.2.3 线网

线网是 Verilog 语言的基本数据类型,用来建立电连接的模型。除了 trireg 线网(trireg 线网将连线建模为存储电荷的电容)之外,其他的线网并不存储值。恰恰相反,它们仅仅传输值,这些值由行为模型中诸如门输出、赋值语句、寄存器等结构化元素驱动到线网。

在例 4.1 中定义了一个连线类型的线网 x2,可以用下面的语句来定义一个具有延迟的线网表项:

```
wire #3 x2;
```

该语句意味着第一个与非门示例输出所驱动的值发生改变后,经过三个单位延迟才在输出连线的末端(末端是另外的与非门和非门)可见。更进一步,延迟也可以包括上升沿和下降沿时间说明:

```
wire #(3,5) x2;
```

该语句表示上升为 1 需要 3 个单位的延迟,而下降为 0 则需要 5 个单位的延迟。

然而,我们在例 4.1 中会发现更多连线的隐含定义。例如,线网表项 x9 作为异或非门的输出,在全加器模块 fullAdder 中并没有被显式地定义。在模块定义中,如果某个标识符在没有预先定义的情况下使用,则该标识符被隐式定义为连线类型的标量线网(在默认情况下,隐含定义的类型为连线,然而我们可以使用 default\_nettype typeOfNet 编译指令来重载这一默认类型,其中 typeOfNet 可以是表 4.4 中除了 supply0 和 supply1 类型的任何类型)。

连线 x2 并不需要单独声明,此处对它的声明只是为了更清楚地说明问题。

在例 4.2 中说明了不同类型线网表项的用法——线与(wired-AND 或 wand)。线与在线网中实现与操作。与门和线与操作的唯一区别在于线与会使输入 z 通过它而与门却会将输入端的 z 看成是 x。

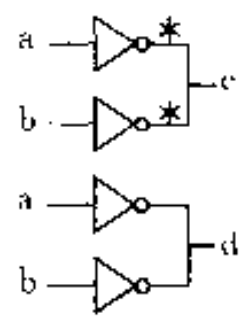
例 4.2 线与

```
module andOfComplements (a, b, c, d);
    input  a, b;
    output c, d;

    wand   c;
    wire   d;

    not    (c, a);
    not    (c, b);

    not    (d, a);
    not    (d, b);
endmodule
```



在此说明了一般的连线类型和线与线网类型的差异。d 被声明为一个连线型线网，c 被声明为线与类型的线网，c 和 d 都被两个不同的非门驱动。声明为线与类型的线网实现线与功能。当两个输入 a 和 b 均为 0 时输出 c 为 1。

另一方面，d 是一个被两个门驱动的连线型线网。除非两个门都输出相同的值，否则其值是未知的(x)。从根本上看，线与类型的允许线网上有多个驱动源，并会在这些驱动源之间实现线与功能。当连线型线网由不同的值驱动时，它会显示未知值(x)。表 4.3 说明了在例 4.2 中各种可能的输入情况下的输出(注：真值表的第十六行被压缩成两行)。

表 4.3 例 4.2 的 wand 和 wire 结果

a	b	c	d	a	b	c	d
0	0	1	1	x	0	x	x
0	1	0	x	x	1	0	x
0	x	x	x	x	x	x	x
0	z	x	x	x	z	x	x
1	0	0	x	z	0	x	x
1	1	0	0	z	1	0	x
1	x	0	x	z	x	x	x
1	z	0	x	z	z	x	x

线网声明的一般形式如下：

```

net_declaration
::= net_type [ vectored | scalared ] [ range ] [ delay3 ] list_of_net_identifiers;
| trireg [ vectored | scalared ] [ charge_strength ] [ range ] [ delay3 ]
    list_of_net_identifiers;
| net_type [ vectored | scalared ] [ drive_strength ] [ range ] [ delay3 ]
    list_of_net_decl_assignments;

net_type
::= wire | tri | tri1 | supply0 | wand | triand | tri0 | supply1 | wor | trior

list_of_net_identifiers
::= net_identifier { , net_identifier }

range
::= [ msb_constant_expression : lsb_constant_expression ]

```

在本章中我们首先考虑第一个线网声明。net\_type 是表 4.4 中所列的类型之一(例如连线 and 线 or); range 是说明位宽度(默认情况下为 1 比特)的可选项; delay 使得线网表项有自己的延迟(默认情况下为 0); list\_of\_net\_identifiers 是一组由逗号隔开的线网表, 这些线网具有给定的范围和延迟特性。在说明了一个线网的延迟后, 来自于任何实体的驱动该线网的值在指定的延迟时间后传播到与该线网表项连接的所有实体中。

表 4.4 线网类型和它们的建模用法

线网类型	建模用法
wire 和 tri	用于建立不带逻辑功能的连接的模型。它们只是名称不同。可以根据需要使用不同的名称, 以增强可读性。
wand, wor, triand, trior	用于建立连线逻辑函数的模型。相同逻辑函数的 wire 和 tri 类型只是名称不同。
tri0, tri1	用于建立对给定电源具有电阻性拉动的连接。
supply0, supply1	用于建立电源连接的模型。
trireg	用于建立线网上电荷存储的模型。参考第 9 章。

线网的范围可以定义为向量或是标量类型, 默认情况下为标量, 说明一个向量的单个位可以使用位选择(bit-select)或部分选择(part-select)来进行访问, 这样就允许一个线网的独立的位或是一部分位可以被门、基元、模块的输出驱动, 或是成为持续赋值语句的左项。实际上, 当指明为标量型时, 一个 n 维向量可以被看成是 n 个标量的集合。向量说明

则禁止这种访问方式,而仅允许向量线网作为一个单独的实体访问。

参考: trireg 9.1; 电荷存储属性 9.2.2; 延迟 4.7; 对线网的持续赋值 4.4.2; 基元 8; 位选和域选 E.1; 标识符作用域 2.6

#### 4.2.4 模块例示与端口规范

模块的端口可以看作是一条纽带,它的一端深入模块例示之内,另一端则在模块例示之外。我们已经见过了很多模块端口说明的示例。

一个输入型端口说明一个向量或是标量的内部名称,该向量或标量是被一个外部实体所驱动的;一个输出型端口则说明了被内部实体驱动而在模块外又可见的向量或是标量的内部名称;一个输入输出型端口说明了一个被内部或外部实体驱动的向量或是标量的内部名称。

在此我们重申说明中的注意事项。首先,一个输入型或者输入输出型端口不可以定义为寄存器类型,这两种类型端口可以通过过程赋值语句读入到寄存器中,作为持续赋值语句的右项,或者是作为模块和门例示的输入。一个输入输出型端口只能够被一个具有高阻能力的门驱动,例如 bufif0 门。

其次,每一个端口连接都是源对沟道的持续赋值,被连接的端口一个是信号源,另一个是信号沟道。一个模块的输出型端口隐含地连接到信源实体,比如线网、寄存器、门输出、例示模块输出,以及模块内部的持续赋值语句的左项,输入型端口连接到门输入、例示模块输入,以及模块内部的持续赋值语句和过程赋值语句的右项,输入输出型端口从内部连接到门输入或者输出。从外部来讲,只有线网才有可能和一个模块的输出相连。

最后,一个模块的端口在例示时通常以它们被定义的顺序联系在一起,从而可以通过命名端口和给定其连接来连接一个模块的端口。在例 4.2 中给出了 andOfComplements 的定义,我们可以将其例示成另外一个模块,并通过如下所示的命名关联其端口。

```
module ace;
    wire r,t;
    reg q,s;
    // 其他声明

    andOfComplements m1(.b(s),.a(q),.c(r),.d(t));
endmodule
```

在这个示例中,我们指明模块 andOfComplements 的例示 m1 中端口 b 与寄存器 s 的输出相关联,端口 a 与寄存器 q 的输出相关联,端口 c 与连线 r 相关联,端口 d 与连线 t 相关联。请注意其中的句点(“.”)引入在模块中定义的端口名称,而且这种关联是以任意的顺序给出的。如果某个端口未指定关联,也就是说,在圆括号中没有给定值,那么,d( )就



意味着没有为模块 `ace` 中的模块 `andOfComplements` 的例示 `m1` 的端口 `d` 建立关联。

关于这一点,我们可以严格地指明例示模块和关联它们的端口的语法。请注意以下的语法说明中包含两种列出模块关联的方法:位置关联和命名关联。

module instantiation

```
::= module_identifier [ parameter_value_assignment ] module_instance { ,  
    module_instance } ;
```

parameter\_value\_assignment

```
::= # ( expression { , expression } )
```

module\_instance

```
::= name_of_instance ( [ list_of_module_connections ] )
```

name\_of\_instance

```
::= module_instance_identifier [ range ]
```

list\_of\_module\_connections

```
::= ordered_port_connection { , ordered_port_connection }  
    | named_port_connection { , named_port_connection }
```

ordered\_port\_connection

```
::= [ expression ]
```

named\_port\_connection

```
::= . port_identifier ( [ expression ] )
```

参见: 参数说明 4.5

#### 4.2.5 有关逻辑级的一个示例

在本节中,我们将给出一个逻辑级建模的示例,这就是一个编码解码系统,实现汉明码(Hamming)的编码解码功能。当系统中可能有噪声进入,从而导致数据被破坏时,就需要使用汉明编码。比如内存中的数据就有可能是以某种编码形式存放的。在下面的示例中,先对一个八位数据进行编码,再将此编码数据通过一个有噪信道,然后再由信道的输出产生出原始数据,如果需要的话,更正一位错误。在大部分介绍逻辑设计的教材中都有关于此项技术的详细推导和描述。

查错与纠错是通过在待编码的消息中加入附加位来实现的。图 4.2 中给出了基本的

编码过程。在图中我们可以看到左边的 8 位原始数据位(Dx)经过编码变成了右边的 12 位数据。从中间一列我们可以看出,原始数据位中插入了四个汉明位(Hamming bit)。这四个汉明位是通过对原始数据位的某些位进行异或操作得到的。这些位的插入次序是非常关键的,因为对编码数解码后,就是通过指明位置来说明哪一位数据(包括汉明位)是不正确的。编码数据各位的次序如图 4.2 右边一列所示。

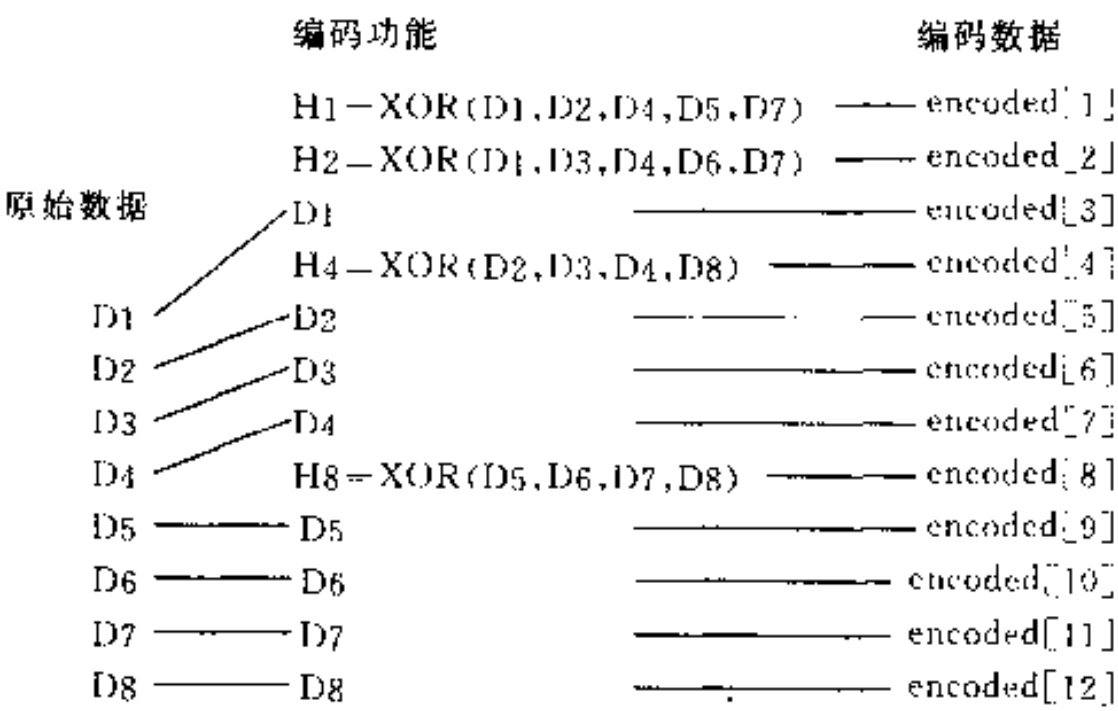


图 4.2 基本的汉明编码过程

本例的完全过程——包括上面所讲的编码功能,如图 4.3 所示。其中的模块 testHam 例示出其他的所有模块,并为它们提供测试向量。testHam 的子模块中包括汉明编码(hamEncode),它实现图 4.2 所示的功能;testHam 的子模块中还包括汉明解码(hamDecode),hamDecode 中还包含它自己的子模块。在该图中间的虚线框中是一个赋值语句,该语句的目的是将单一位错误加入到已经编码的数据中。模块 hamDecode 通过纠正此单一位错误,重新产生原始的八位数据信息。整个系统的跟踪模拟如图 4.4 所示,在其中指明了 testHam 主要的子实体之间值的传递情况(从名称上来讲,指的是 original, encoded, messedUP, regenerated)。

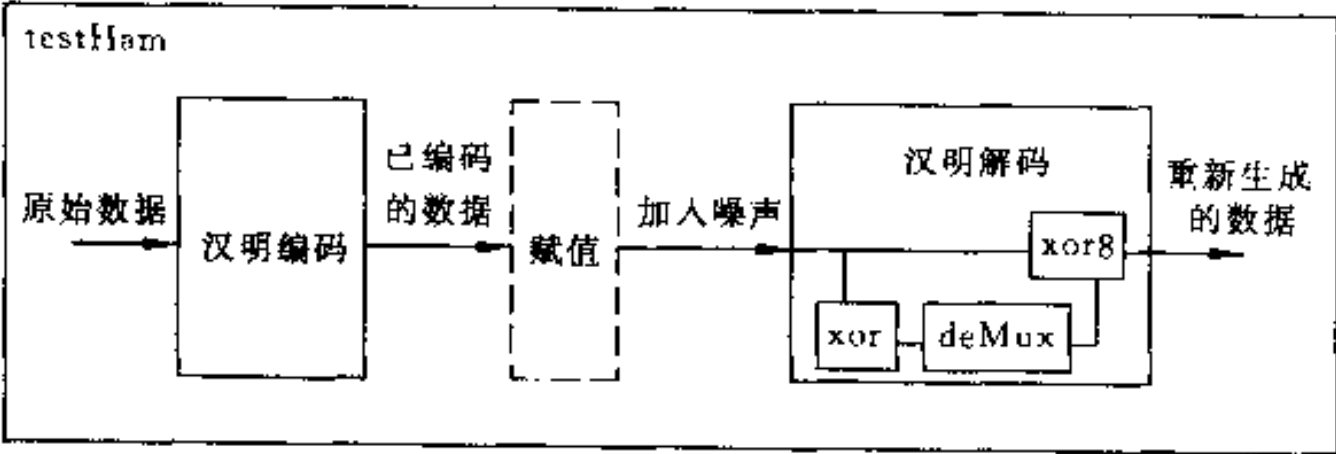


图 4.3 带噪声沟道的汉明编码和译码器

例 4.3 是一个汉明码的示例。模块 hamEncode 产生 vIn 的 12 位编码 (valueOut)。编码是通过四个例示的异或门实现如图 4.2 所示的编码功能的。这些门的输出连接到线 (h1, h2, h4 和 h8) 上, 并且这些线在赋值语句中与数据位连接在一起, 连接关系是通过 “{}” 构造指出的。在输出 valueOut 中是由一些逗号 “,” 隔开的连线和连线的域选 (part select) 的列表。请注意值连接的顺序与在图 4.2 中所给出的顺序是相匹配的。还有一点值得注意的是, 该模块从本质上来讲完全是结构化的, 其中没有过程语句或寄存器。

模块 hamDecode 的输入是 12 位的 vIn, 它产生出 8 位的 valueOut。从内部来看, 模块 hamDecode 需要决定哪些位是错误的 (如果存在位错误的话), 并且更正之。四个异或门产生出在 c1, c2, c4 和 c8 上的输出, 并且通过它们指明某一位是错的。将上述四个输出作为向量的分隔位: c8 有位置值 8, c4 有位置值 4, 依次类推。如果所有这四个输出的值均为 0, 则无须任何更正。若均不为 0, 该值就指明了发生错误位的位序号。此处一位需要反转。这些输出 (c) 作为模块 deMux 的输入, 该模块将其解码为 8 位可能的数位之一。这 8 位数被赋值给 8 位线向量 bitFlippers, 这 8 位数对应了需要被反转的位的位置 (1 表示需要反转, 0 表示不需要反转)。bitFlippers 的单个的位在 xor8 中作为 8 个异或门的输入。8 个异或门的其他输入就是需要被更正的数据的各位。bitFlippers 中某位若为 1, 与它对应的一位是错误的, 需要反转 (更正) 为正确的数据位。xor8 的输出是已经更正过的数据, 并且它也是模块 hamDecode 的输出。

现在仔细地观察模块 deMux, 我们可以看到四个选择输入 (a--d) 和一个使能端 enable。输入 d 与 8 的位置值相对应, 而 a 则与 1 的位置值相对应。该模块的目的是产生一个 8 位的向量。该向量至多有一个位集, 一位与编码位对应的那一位是需要更正的。因为我们只需要更正数据位, 因此仅仅产生位 a--d 的必要的最小项形式。它们是: 3, 5, 6, 7, 9, 10, 11 和 12, 与图 4.2 中所示的已编码数据位相符合。与门产生这些最小项, 赋值语句将它们连接到 outVector 上去。

模块 deMux 的输出 BitFlippers 是 xor8 的输入。这些输入位 (输入 xin1) 和编码数据的第 3, 5, 6, 7, 9, 10, 11 和 12 (输入 xin2) 位一起作为 8 个异或门的输入, 从而可以使某个不正确的位被 xin1 中的某一位的 “1” 指出来, 使得这个不正确的位被异或门反转。xor8 的输出也是模块 hamDecode 的输出。

现在回到模块 hamTest, 我们可以看到 original 是 hamEncode 的输入而 encoded 是其输出, 接下来 encoded 作为赋值语句的输入, 赋值语句产生 messedUP, 在此赋值语句的目的是为了模拟一个有噪信道, 在该有噪信道中某位数据被反转, 在本例中位 7 被反转。赋值语句的输出 (messedUP) 作为模块 hamDecode 的输入, 模块 hamDecode 更正被反转的那一位, 并且在 regenerated 端产生正确数据的原型。

整个示例中的过程语句仅仅出现在模块的初始化语句组中, 其目的是通过系统运行测试向量。为了达到此目的, 我们使用 \$random 系统任务来产生一组随机数。在此, 我

们将 \$ random 系统任务的种子值设定为 1, 然后进入一个死循环。\$ random 系统任务的结果 original 被载入系统。original 的输出驱动模块 hamEncode。因为对于任何门基元或连线都不存在延迟, 因此花费 0 模拟周期即可产生 regenerated。死循环需要延迟一个时间单位来确保产生 regenerated, 然后显示如图 4.4 中所有的中间模块的值。注意该图的第一行, 原始数据 00 被编码为 000, 赋值语句将第 7 位反转, 产生了错误的的数据 020 (在本例中, 位是从左往右数的, 下标从 1 开始, 而且 \$ display 系统任务以十六进制方式显示数据)。接下来错误位被更正, 产生正确的原始数据。

#### 例 4.3 汉明编码/解码示例

```
module testHam();
    reg      [1:8]      original;
    wire     [1:8]      regenerated;
    wire     [1:12]     encoded,
                                messedUp;
    integer          seed;

    initial begin
        seed = 1;
        forever begin
            original = $ random (seed);
            #1
            $ display ("original=%h, encoded=%h, messed=%h, regen=%h",
                        original, encoded, messedUp, regenerated);
        end
    end

    hamEncode      hIn (original, encoded);
    hamDecode      hOut (messedUp, regenerated);

    assign messedUp = encoded ^ 12'b 0000_0010_0000;
endmodule

module hamEncode (vIn, valueOut);
    input  [1:8]      vIn;
    output [1:12]     valueOut;
```

```

wire    h1, h2, h4, h8;

xor      (h1, vIn[1], vIn[2], vIn[4], vIn[5], vIn[7]),
          (h2, vIn[1], vIn[3], vIn[4], vIn[6], vIn[7]),
          (h4, vIn[2], vIn[3], vIn[4], vIn[8]),
          (h8, vIn[5], vIn[6], vIn[7], vIn[8]);

assign valueOut = {h1, h2, vIn[1], h4, vIn[2:4], h8, vIn[5:8]};

endmodule

module hamDecode (vIn, valueOut);
  input  [1:12]    vIn;
  output [1:8]     valueOut;
  wire          c1, c2, c4, c8;
  wire  [1:8]     bitFlippers;
  xor      (c1, vIn[1], vIn[3], vIn[5], vIn[7], vIn[9], vIn[11]),
          (c2, vIn[2], vIn[3], vIn[6], vIn[7], vIn[10], vIn[11]),
          (c4, vIn[4], vIn[5], vIn[6], vIn[7], vIn[12]),
          (c8, vIn[8], vIn[9], vIn[10], vIn[11], vIn[12]);

  deMux mux1 (bitFlippers, c1, c2, c4, c8, 1'b1);
  xor8 x1 (valueOut, bitFlippers, {vIn[3], vIn[5], vIn[6], vIn[7], vIn[9], vIn[10], vIn[11],
                                   vIn[12]});
endmodule

module deMux (outVector, A, B, C, D, enable);
  output [1:8]    outVector;
  input          A, B, C, D, enable;
  and      v (m12, D, C, ~B, ~A, enable),
          h (m11, D, ~C, B, A, enable),
          d (m10, D, ~C, B, ~A, enable),
          l (m9, D, ~C, ~B, A, enable),
          s (m7, ~D, C, B, A, enable),
          u (m6, ~D, C, B, ~A, enable),
          c (m5, ~D, C, ~B, A, enable),

```

```

ks (m3, ~D, ~C, B, A, enable);

assign outVector = {m3, m5, m6, m7, m9, m10, m11, m12};
endmodule

module xor8 (xout, xin1, xin2);
    output [1:8] xout;
    input [1:8] xin1, xin2;
    xor (xout[8], xin1[8], xin2[8]),
        (xout[7], xin1[7], xin2[7]),
        (xout[6], xin1[6], xin2[6]),
        (xout[5], xin1[5], xin2[5]),
        (xout[4], xin1[4], xin2[4]),
        (xout[3], xin1[3], xin2[3]),
        (xout[2], xin1[2], xin2[2]),
        (xout[1], xin1[1], xin2[1]);
endmodule

original=00,encoded=000,messed=020,regen=00
original=38,encoded=078,messed=058,regen=38
original=86,encoded=606,messed=626,regen=86
original=5c,encoded=8ac,messed=88c,regen=5c
original=ce,encoded=79e,messed=7bc,regen=ce
original=c7,encoded=e97,messed=eb7,regen=c7
original=c6,encoded=f86,messed=fa6,regen=c6
original=f3,encoded=2e3,messed=2c3,regen=f3
original=c3,encoded=a83,messed=aa3,regen=c3

```

图 4.4 例 4.3 的模拟结果

对于本例来说,有以下几点值得强调:

- 种子(seed)值声明为一个整型值。整数在模拟模型中通常作为辅助计算的工  
具,在实际的设计中,该值是不存在的,它在这里仅仅用于测试目的。当我们为实际硬件的某  
一部分建模时,我们应该使用寄存器。在附录 E 中有关于整数的详细讨论。
- 在此 \$display 系统任务要求以十六进制方式打印数据,我们是通过“%h”打印控  
制符来指明的。请参看附录 F 中有关 \$display 系统任务的详细介绍。
- 模块 testHam 中的赋值语句展示一种书写大数字的有效方法。下划线“\_”字符  
可以被任意地加入到某个数字中以加强程序的可读性。

• 与其他的示例相比较,本例的位计数方式是不同的。在本例中,位是从左往右数的,这是算法描述中常用的典型表示方法。位也有可能从任何一个方向开始计数,甚至可以包括负索引。

参见: \$display F.1 ; \$random F.7

### 4.3 示例数组

我们在前面例 4.3 的 xor8 模块中已经看到其定义太繁杂,因为我们不得不为每一个异或的示例来选择合适的位进行标号。在 Verilog 中有一种简便的方法来指明一组示例,使得连续示例中各自的位编号能以一种可以控制的方式相异。在本节的例 4.4 中,给出模块 xor8 的等价定义,在左边的定义中就使用了示例数组,右边是其展开形式,它与例 4.3 中原始的模块 original 是等价的。

示例数组的说明使用可选的范围说明符来为示例名称提供序号。在例 4.4 的模块中有 8 个异或门的示例。范围说明符是如下定义的:

```
range
::=[msb_constant_expression:lsb_constant_expression]
```

对于 msb 和 lsb 的绝对值和相互之间的关系并没有任何其他要求——除了它们都必须是整数之外,它们的大小关系也没有任何要求。实际上,它们甚至可以是相等的,在这种情况下,只产生一个示例。在给定 msb 和 lsb 之后,会产生  $1 + \text{abs}(\text{msb} - \text{lsb})$  个示例。

**例 4.4** 使用示例数组的 xor8 的等价形式

```
module xor8 (xout, xin1, xin2);
output      [1:8]      xout;
input       [1:8]      xin1, xin2;
```

```
    xor a[1:8] (xout, xin1, xin2);
endmodule
```

```
module xor8 (xout, xin1, xin2);
output      [1:8]      xout;
input       [1:8]      xin1, xin2;
    xor      (xout[1], xin1[1], xin2[1]),
              (xout[2], xin1[2], xin2[2]),
              (xout[3], xin1[3], xin2[3]),
              (xout[4], xin1[4], xin2[4]),
```

```

        (xout[5], xin1[5], xin2[5]),
        (xout[6], xin1[6], xin2[6]),
        (xout[7], xin1[7], xin2[7]),
        (xout[8], xin1[8], xin2[8]);
endmodule

```

上面的例子是当每一个示例都与输入或输出的一个位选择联系在一起时的情况,当产生了示例和建立了两者之间的关系之后,一定会存在一个位的等价数,该数由终端(端口、线、寄存器)提供,并且对于示例来说是必需的。在上面的情况中,8个示例在每一个输入和输出端口都需要8位(如果这些数不相等的话,就意味着出错)。然而,示例并不仅仅局限在位选择的连接上。当示例数目已知时,可能会有这样的情况,即每个示例会从每个终端收到两(或者更多)位。在这种情况下,一般会为每个连接创建域选。而且,没有剩下任何位。如果一个终端只有一位但有n个示例的话,可以看成是每一个示例分别与一个一位终端相连接。

#### 例 4.5 用示例数组构造寄存器

```

module reggae (Q, D, clock, clear);
    output      [7:0]    Q;
    input       [7:0]    D;
    input                               clock, clear;

    dff         r[7:0]   (Q, D, clear, clock);
endmodule

```

例 4.5 说明 D 触发器连接起来构成一个寄存器。与示例等价的模块扩展的方法在例 4.6 中也可以看到。请注意 clock 和 clear 作为一位终端和各个示例连接在一起。现在我们考虑,当 dff 不是一个一位触发器,而是一个二位触发器时的情况,此时示例数组的说明应该采用如下的形式:

```
twoBitDff  r[3:0] (Q,D,clear,clock);
```

其扩展形式的第一个示例为:

```
twoBitDff  r[3] (Q[7:6],D[7:6],clear,clock)
```

#### 例 4.6 例 4.5 的等价扩展

```

module regExpanded (Q, D, clock, clear);
    output      [7:0]    Q;
    input       [7:0]    D;
    input                               clock, clear;

```



```

dff      r7      (Q[7], D[7], clear, clock),
          r6      (Q[6], D[6], clear, clock),
          r5      (Q[5], D[5], clear, clock),
          r4      (Q[4], D[4], clear, clock),
          r3      (Q[3], D[3], clear, clock),
          r2      (Q[2], D[2], clear, clock),
          r1      (Q[1], D[1], clear, clock),
          r0      (Q[0], D[0], clear, clock);
endmodule

```

#### 例 4.7 带示例数组的端口连接

```

module regFromTwoBusses (Q, busHigh, busLow, clock, clear);
    output  [7:0]  Q;
    input   [3:0]  busHigh, busLow;
    input                                clock, clear;

    dff      r[7:0]  (Q, {busHigh, busLow}, clear, clock);
endmodule

```

一个示例的所有连接并不需要来自于同一终端。例 4.7 说明了当寄存器模块的输入来自于两处时,输入是如何被正确地联系在一起。在本例中, busHigh 和 busLow 通过示例数组说明的形式被连接在一起,而且 8 位数与 8 个示例通过如下方式联系在一起:

```

diff  r7  (Q[7], busHigh[3], clear, clock),
       r6  (Q[6], busHigh[2], clear, clock),
       r5  (Q[5], busHigh[1], clear, clock),
       r4  (Q[4], busHigh[0], clear, clock),
       r3  (Q[3], busLow[3], clear, clock),
       r2  (Q[2], busLow[2], clear, clock),
       r1  (Q[1], busLow[1], clear, clock),
       r0  (Q[0], busLow[0], clear, clock);

```

上面的示例已经说明使用示例数组时一维结构——通过触发器构造寄存器。多维结构的说明可以通过在一个示例数组中来例示另一个一维结构来实现。因此,通过使用如例 4.8 所示的示例数组来例示例 4.5 中的寄存器 reggae,我们就可以建造一个移位寄存器。

#### 例 4.8 移位寄存器

```

module shiftRegister (in, out, clock, clear);

```

```

input      [7:0]      in;
output     [7:0]      out;
input                               clock, clear;
wire       [15:0]     w;

    reggae                stage[2:0]  ({out, w}, {w, in}, clock, clear);
endmodule

```

其展开例示为:

```

reggae  stage2  (out[7:0],w[15:8],clock,clear),
          stage1  (w[15:8],w[7:0],clock,clear),
          stage0  (w[7:0],in[7:0],clock,clear);

```

## 4.4 持续赋值

持续赋值为组合硬件在线网上的驱动值提供一种抽象的建模方法。在前一节中所讲的一位全加器的另外一种使用持续赋值的表现形式见例 4.9。其中我们可以看出两个输出 sum 和 cOut 是使用一个赋值语句来描述的。前者(sum)是三个输入的异或,而后者(cOut)是三个输入的 majority 函数。

### 例 4.9 持续赋值语句示例

```

module oneBitFullAdder(cOut, sum, aIn, bIn, cIn);
    output      cOut, sum;
    input       aIn, bIn, cIn;

    assign      sum = aIn ^ bIn ^ cIn,
              cOut = (aIn & bIn) | (bIn & cIn) | (aIn & cIn);

endmodule

```

在有关行为建模的章节中我们曾经讲述过程赋值语句,持续赋值语句与过程赋值语句是不同的。因为持续赋值语句总是活跃的(驱动一个 0,1,x 或是 z),而不管电路中的任何状态顺序。如果任意时刻赋值语句的任意输入发生变化,赋值语句都会重新求值,并且将输出传送出去。这是组合逻辑的特点之一,也是 Verilog 门级时序模型的特点之一。

赋值语句的一般形式为:

```

continuous assign
    ::= assign [drive strength][delay3]list_of_net_assignments;

```

```
list_of_net_assignments
::= net_assignment { , net_assignment }
```

```
net_assignment
::= net_lvalue = expression
```

在此, assign 是一个关键字, 而说明符 drive\_strength 和 delay3 都是可选部分, list\_of\_net\_assignments 则是一个由逗号隔开的表, 正如例 4.9 中所示的那样。持续赋值语句的驱动强度(drive\_strength)默认为 strong0 和 strong1, 也可以明确地指定为除 supply0 和 supply1 之外任何类型的标量线网。默认延迟为 0。例如, 上面的赋值语句也可以写成下面的形式:

```
assign      (strong0, strong1)
            sum = aIn + bIn + cIn
            cOut = (aIn & bIn) | (bIn & cIn) | (aIn & cIn);
```

在此我们指明这两个持续赋值语句都有相同的默认驱动强度。

参考: 延迟建模 4.7, 4.8; 强度建模 9; 时序建模 5.1

#### 4.4.1 组合电路的行为建模

持续赋值语句提供一种由电路门级模型进行抽象的方法。从这个意义上讲, 持续赋值语句是组合电路行为建模的一种形式。也就是说, 我们只需要说明逻辑功能的布尔代数形式, 而非其实际的门级实现。最终的门级实现留待逻辑综合程序和进一步的设计工作来实现。

赋值语句的右项表达式可能包括对有关 Verilog 函数的函数调用。回忆一下, 在一个函数中, 我们可能用到诸如分支和循环之类的过程语句, 而不是 wait, @event, #delay 等。我们可以用过程语句来描述一个复杂的组合逻辑功能。例如, 在例 4.10 中, 有关多路器的描述就说明了在赋值中的函数调用。

##### 例 4.10 持续赋值语句的函数调用

```
module multiplexor(a, b, c, d, select, e);
    input      a, b, c, d;
    input      [1:0] select;
    output     e;

    assign     e = mux(a, b, c, d, select);

function     mux;
```

```

input      a, b, c, d;
input      [1:0] select;

case (select)
    2'b00:    mux = a;
    2'b01:    mux = b;
    2'b10:    mux = c;
    2'b11:    mux = d;
    default:  mux = 'bx;
endcase
endfunction
endmodule

```

在本例中,模块 multiplexor 有一个持续赋值语句,该语句调用了函数 mux。该函数使用过程化的分支语句来描述组合多路功能的行为。如果某个条件表达式与控制表达式的值相匹配的话,则 mux 被赋予相应的值。如果前四项都无法匹配的话(例如输入端 select 为 x 或 z 的话),则根据默认情况, mux 被赋予未知值 x。

虽然赋值语句提供了对从行为上描述组合硬件的过程语句的一个分类的访问方法,我们还必须认识到行为建模中的不同抽象层次。在一个较高的抽象层次中,我们使用 process 来为时序行为(sequential activity)建模,在第 2 章和第 3 章中有对时序行为的描述。在这一层次,我们描述一种包含独立控制线程的情况,并且实现一般由其自身内部的状态机来跟踪其输入的状态变化情况。为了给其建模,我们用一条 always 语句定义了一个模块,并通过模块端口、进程间等待以及事件语句来进行通信。说得更加明白一点,这并不是例 4.10 中的建模情况,在那里我们仅仅描述了一个组合多路器,在某个输入和输出之间装了一扇门,并无内部状态机来控制它。

而且,在这个比较低的抽象层次上,我们是在为没有包括自身内部状态的组合行为建模,例 4.10 并不是采用布尔代数的形式来描述一个多路器,而是使用了过程语句,从赋值中调用的函数里使用过程语句,仅仅给了我们另一种描述组合行为的方法,用这种方法建模,并不意味着一定要使用时序状态机来实现,也不意味着当为顺序行为建模时顺序状态机制不能够被使用。

参见: 函数 2.5.2

#### 4.4.2 线网与持续赋值语句声明

持续赋值语句说明了一个用于驱动线网的值。描述此种情况的一种简单方法是将线网和持续赋值语句组合到一起,如例 4.11 所示。

**例 4.11** 组合线网与持续赋值

```

module modXor (AXorB, a, b);
    output      [7:0]    AXorB;
    input       [7:0]    a, b;

    wire        [7:0]    #5 AXorB = a ^ b;
endmodule

```

在此我们定义一个 8 位的向量线(vector wire), 以及一个输入 a 和 b 的 8 位异或来驱动这 8 位的向量线。延迟说明指明了延迟包含在异或中, 而不是在线驱动中。

如果我们如下独立地定义连线和异或:

```

wire    [7:0]    AXorB ;
assign  #5       AXorB = a ^ b ;

```

我们也可以使用下面的语句将延迟 10 单独地赋给线驱动:

```

wire    [7:0]    #10 AXorB ;

```

当我们在一个线网声明中给出一个如上所示的延迟时, 延迟被加到驱动该线网的所有驱动源上去。例如例 4.12 中的模块。我们已经定义了一个线与型线网, 它具有延迟 10, 有两个赋值语句驱动该线网, 一个赋值语句具有延迟 5, 而另一个具有延迟 3。当输入 a 发生变化时, 则在这种变化影响到与 c 连接的输入之前, 有一个长度为 15 的延迟; 当输入 b 发生变化时, 在这种变化影响到与 c 连接的输入之前, 有一个长度为 13 的延迟。

#### 例 4.12 线网与持续赋值的延迟

```

module wandOfAssigns (a, b, c);
    input      a, b;
    output     c;

    wand       #10      c;

    assign     #5       c = ~a;
    assign     #3       c = ~b;
endmodule

```

线网说明与持续赋值语句的混合使用在形式上使用 net\_declaration 的第三个入口 (回忆一下, 第一项是用于声明线网的, 而第二项将在第 9 章中介绍)。

```

net_declaration
    ::= net_type [vectorred | scalarred] [range] [delay3] list_of_net_identifiers ;
    |   trireg [vectorred | scalarred] [charge_strength][range] [delay3]

```

```

        list_of_net_identifiers ;
    |
    net_type [ vectored | scalared ] [drive_strength] [range] [delay3]
    list_of_net_decl_assignments ;

net_type
    ::= wire | tri | tri1 | supply0 | wand ; triand | tri0 | supply1 | wor | trior

list_of_net_decl_assignments
    ::= net_decl_assignment { , net_decl_assignment }

net_decl_assignment
    ::= net_identifier = expression

```

net\_declaration 的第三个入口与第一个入口相比的不同之处在于它可以说明强度, 以及有一个与强度-范围-延迟组合相关的赋值链表。

持续赋值语句也可用于驱动一个输入输出型(inout)端口。例 4.13 是一个 buffer-driver 的示例。

#### 例 4.13 对 inout 端口的持续赋值

```

module bufferDriver (busLine, bufferedVal, bufInput, busEnable);
    inout      busLine;
    input      bufInput, busEnable;
    output     bufferedVal;

    assign     bufferedVal = busLine,
             busLine = (busEnable) ? bufInput : 1'bz;
endmodule

```

在此我们使用 busEnable 来选择是使用 busInput 来驱动 busLine 还是用高阻态来驱动该线。然而, 无论 busEnable 处于何种状态, bufferedVal 的值总是随着 busLine 值的变化而变化, 从而当 busEnable 为 0 时, busLine 可以在一个外部模块中被驱动, 而且在 bufferedVal 中会显示 busLine 的值。

三态驱动源的一个典型应用是在存储器模块的设计中, 该存储器是连接在处理器总线上的。例 4.14 说明一个 64K 字节存储器。其中 dataBus 端口定义为 inout 类型, 因此它可以被该模块内的赋值语句所驱动, 也可以作为写内存时的源。写内存是一种由时钟上升沿控制的同步动作。当读允许(re)首先变为允许状态时(例如负边沿), 或者当地址线(addrBus)上发生了某些变化时, 从内存中读取一个新的值。读取的值放在暂存器 out 中, 当读允许(re)允许时, out 驱动 dataBus, 如果读允许(re)未处于允许状态, 则 dataBus

处于三态。

#### 例 4.14 带三态驱动源的存储器模块

```
module Memory_64Kx8 (dataBus, addrBus, we, re, clock);
    input  [15:0]  addrBus;
    inout  [7:0]   dataBus;
    input                      we, re, clock;
    reg    [7:0]   out;
    reg    [7:0]   Mem [65535:0];

    assign dataBus == (~re)? out; 16'bz;          /* drive the tristate output */

    always @(negedge re or addrBus)
        out = Mem[addrBus];

    always @(posedge clock)
        if (we == 0)
            Mem[addrBus] <= dataBus;
endmodule
```

参见:线网,向量/标量 4.2.3

## 4.5 参数化定义

参数允许我们定义一个类属模块,该模块可以根据不同情况的需要分别被参数化,不仅允许我们在多种情况下重复使用一个模块的定义,而且允许我们定义有关模块的类属信息。当模块示例化时,我们可以重载(override)这些类属信息。

在例 4.11 中,我们使用一个持续赋值语句定义一个 8 位的异或模块。在这一节中,我们将推导出其参数化形式。例 4.15 就是从例 4.3 中摘录出来的,它说明了 xor8 模块的定义。首先,我们用单个赋值语句的形式取代 8 个异或门示例:

```
assign xout = xin1 ^ xin2
```

在此 xout, xin1 和 xin2 都是在例 4.3 和例 4.15 中定义过的。

#### 例 4.15 摘自例 4.3 的 xor8 模块

```
module xor8 (xout, xin1, xin2);
    output  [1:8]  xout;
    input   [1:8]  xin1, xin2;
```

```

xor      (xout[8], xin1[8], xin2[8]),
         (xout[7], xin1[7], xin2[7]),
         (xout[6], xin1[6], xin2[6]),
         (xout[5], xin1[5], xin2[5]),
         (xout[4], xin1[4], xin2[4]),
         (xout[3], xin1[3], xin2[3]),
         (xout[2], xin1[2], xin2[2]),
         (xout[1], xin1[1], xin2[1]);
endmodule

```

随后,我们使用例 4.16 中的定义使得该模块更加具有通用性。我们在这里是通过说明两个参数的形式来说明 xorx 模块的一般形式,模块宽度为 4,延迟为 10。参数说明也是模块定义的一部分,其语法形式如下:

```

module declaration
    ::= module_keyword module_identifier [ list_of_ports ] ;
       { module_item }
endmodule

```

```

module_item
    ::= module_item_declaration
       | ...

```

```

module_item_declaration
    ::= parameter_declaration
       | ...

```

```

parameter_declaration
    ::= parameter list_of_param_assignments ;

```

```

list_of_param_assignments
    ::= param_assignment { , param_assignment }

```

```

param_assignment
    ::= parameter_identifier = constant_expression

```

参数定义中指明的值用于模块的 generic 示例。该模块可以用下列语句例示为例 4.3:



```
xorx    #(8,0) x1 (valueOut,bitFlippers,
                {vIn[3],vIn[5],vIn[6],vIn[7],vIn[9],vIn[10],vIn[11],vIn[12]});
```

与模块 xor8 相比,例示该模块的主要变化在于上面给定参数的说明。其中“#(8,0)”说明对于该例示而言,第一个参数(宽度:width)的值为 8,而第二个参数(延迟:delay)的值为 0,这些当然都符合例 4.3 的需要。如果“#(8,0)”被省略的话,则会使用模块定义中指明的值,也就是说我们可以在各个模块例示的基础上重载某些参数的值。

#### 例 4.16 带参数的模块说明

```
module xorx (xout, xin1, xin2);
parameter    width = 4,
              delay = 10;
output       [1:width]    xout;
input        [1:width]    xin1, xin2;

    assign #(delay) xout = xin1 ^ xin2;
endmodule
```

在例示时指明参数值的一般形式如下:

```
module instantiation
    ::= module_ identifier [parameter_ value assignment]
        module_ instance{, module_ instance} ;

parameter_ value_ assignment
    #(expression{, expression})
```

重载值的顺序应该与模块定义中参数说明的顺序是一致的。我们不可能在一个模块定义中跳过一些参数而去说明其他的。要么被跳过的参数应该被重新说明为默认值,要么参数表应该重新排序,以使将被跳过的参数位于参数表的最后。例如,我们已经用如下的语句指明了 xorx 的例示 x1 宽度为 8(此处是重载值),延迟为 10(此处为默认值):

```
xorx    #(8) x1 (valueOut,bitFlippers,{vIn[3],vIn[5],vIn[6],vIn[7],
                vIn[9],vIn[10],vIn[11],vIn[12]});
```

但是,为了指明默认宽度(4)和指定延迟(20),我们还需要重新指明形式如下:

```
xorx    #(4,20) x1 (valueOut,bitFlippers,{vIn[3],vIn[5],vIn[6],vIn[7],
                vIn[9],vIn[10],vIn[11],vIn[12]});
```

另外一种在模块定义中重载参数的方法是使用 defparam 语句和 Verilog 的层次命名规定。该方法如例 4.17 所示。

#### 例 4.17 使用 defparam 进行参数重载

```
module xorsAreUs (a1, a2);
    output      [3:0]      a1, a2;
    reg [3:0]    b1, c1, b2, c2;

    xorx        a(a1, b1, c1),
                b(a2, b2, c2);
endmodule

module xorx (xout, xin1, xin2);
    parameter    width = 4,
                  delay = 10;
    output      [1:width]  xout;
    input       [1:width]  xin1, xin2;

    assign #delay xout = xin1 ^ xin2;
endmodule

module annotate;
    defparam
        xorsAreUs.b.delay = 5;
endmodule
```

使用 defparam 语句,参数的所有重新说明可以被组装到描述中的某一个地方。在本例中,在模块 xorsAreUs 内模块 xorx 的例示 b 的延迟参数(delay)被定义为 5,参数从而可以根据个别情况进行重新定义。

defparam 语句的一般形式如下:

```
parameter_override
    ::= defparam list_of_param_assignments ;

list_of_param_assignments
    ::= param_assignment { , param_assignment }
```

是使用 defparam 语句还是使用模块示例的方法来修改参数,完全取决于个人编程风格以及建模的需要。使用模块示例的方法使得在例示处用新值重载默认值时清晰明了,而使用 defparam 语句则允许我们将参数的重新说明集中起来放到特定的位置。实际上,defparam 语句可以集中存放在一个单独的文件中,最后再和模拟模型的其他部分一起编

译。我们可以通过与另外一个不同的 defparam 文件联合编译来改变一个系统,而不是重新编译整个描述。而且,一个独立的程序可以为延迟的后标注产生 defparam 文件。

参见:模型例示与端口说明 4.2.4; 层次命名 2.6

## 4.6 行为级/结构级的混合示例

例 3.8 是一个同步总线的示例。在这一节中,我们将通过使用线而非寄存器来为总线建模,从而使得模型更加实用化,更加一般化。新的模型如例 4.18 所示。Verilog 描述的组织 and 总线协议与前面的示例是一致的。读者可以参考本书 3.4 节中的内容作为这一节的预备知识。

假定有一个总线主进程与一个总线从进程进行通信。与前面的示例相反,在例 4.18 中的通信是通过在 sbus 模块中定义的连线来进行的。在此我们可以看到连线 rw, addr 和 data, 它们都是例示的 master 模块和 slave 模块之间进行通信的唯一方式。rw 和 addr 线仅被总线 master 驱动,然而, data 线必须在写周期中被 master 驱动,在读周期中被 slave 驱动,我们从而需要找到一种使得数据线同步的方法。当然,由 master 产生的 rw 线是指示总线是处于读状态还是写状态的全局指示器。master 模块和 slave 模块都包括一个寄存器 enable, 用于在合适的时刻从内部允许总线驱动源。

模块 busDriver 是用一种类似于例 4.13 中总线驱动源的方式来定义的,主要的区别在于该模块并不是作为总线接收器。该模块的总线形态可以被参数化,并且当 driveEnable 为 TRUE 时用 valueToGo 来驱动总线,否则的话,它驱动一个 z 值。该模块可以被例示为 master 模块和 slave 模块。

在 slave 模块中,加入 enable 寄存器来控制总线驱动源。初始化时 enable 被设置为 0,使得总线上呈现状态 z。在读周期的第二个时钟周期时 enable 被设置为 1。此时也是所读出的值由 slave 驱动到总线上去的时刻。在 master 模块中加入一个独立的 enable 来控制总线驱动源。初始化时该 enable 被设置为 0,在写周期中, master 将 enable 设置为 1,因为此时 master 驱动数据总线。

sbus 模块经过设置,使其可以被时钟周期参数、地址参数、数据总线布局参数,以及内存布局参数例示,因而它可以用于若干种情况下的建模。

### 例 4.18 使用行为和结构构造方法的同步总线

```
`define READ 0
`define WRITE 1

module sbus;
    parameter
```

```

    Tclock = 20,
    Asize = 4,
    Dsize = 15,
    Msize = 31;

reg clock;

wire          rw;
wire  [Asize:0]  addr;
wire  [Dsize:0]  data;

master  #(Asize, Dsize)    m1  (rw, addr, data, clock);
slave   #(Asize, Dsize, Msize) sl  (rw, addr, data, clock);

initial
    begin
        clock = 0;
        $monitor ("rw=%0d, data=%0d, addr=%0d at time %0d",
                    rw, data, addr, $time);
    end

always
    # Tclock clock = !clock;
endmodule

module busDriver(busLine, valueToGo, driveEnable);
    parameter      Bsize = 15;
    inout  [Bsize:0]  busLine;
    input  [Bsize:0]  valueToGo;
    input                                driveEnable;

    assign busLine = (driveEnable) ? valueToGo: 'bz;
endmodule

module slave (rw, addressLines, dataLines, clock);
    parameter
        Asize = 4,

```

```

        Dsize = 15,
        Msize = 31;
input    rw, clock;
input    [Asize:0]    addressLines;
inout    [Dsize:0]    dataLines;

reg       [Dsize:0]    m[0:Msize];
reg       [Dsize:0]    internalData;
reg       enable;

busDriver #(Dsize)    bSlave (dataLines, internalData, enable);

initial
    begin
        $readmemb ("memory.data", m);
        enable = 0;
    end

always    // bus slave end
    begin
        @(negedge clock);
        if (~rw) begin    // read
            internalData <= m[addressLines];
            enable <= 1;
            @(negedge clock);
            enable <= 0;
        end
        else // write
            m[addressLines] <= dataLines;
    end
endmodule

module master (rw, addressLines, dataLines, clock);
    parameter
        Asize = 4,
        Dsize = 15;
    input                clock;

```

```

output          rw;
output [Asize:0] addressLines;
inout  [Dsize:0] dataLines;

reg          rw, enable;
reg  [Dsize:0] internalData;
reg  [Asize:0] addressLines;

busDriver # (Dsize) bMaster (dataLines, internalData, enable);

initial      enable = 0;

always      // bus master end
    begin
        # 1
        wiggleBusLines (READ, 2, 0);
        wiggleBusLines (READ, 3, 0);
        wiggleBusLines (WRITE, 2, 5);
        wiggleBusLines (WRITE, 3, 7);
        wiggleBusLines (READ, 2, 0);
        wiggleBusLines (READ, 3, 0);
        $ finish;
    end

task wiggleBusLines;
    input          readWrite;
    input  [Asize:0] addr;
    input  [Dsize:0] data;

    begin
        rw <= readWrite;
        if (readWrite) begin // write value
            addressLines <= addr;
            internalData <= data;
            enable <= 1;
        end
        else begin // read value

```

```

        addressLines <= addr;
        @(negedge clock);
    end
    @(negedge clock);
    enable <= 0;
end
endtask
endmodule

```

例 4.18 的模拟结果如图 4.5 所示,它与前面的模拟运行(图 3.3)的不同之处仅仅在于读总线周期的第一个时钟周期数据线上是 z,除此之外,两个模型产生的结果完全相同。

rw=x,data=	z,addr=x at time	0
rw=0,data=	z,addr=2 at time	1
rw=0,data=	29,addr=2 at time	40
rw=0,data=	z,addr=3 at time	80
rw=0,data=	28,addr=3 at time	120
rw=1,data=	5,addr=2 at time	160
rw=1,data=	7,addr=3 at time	200
rw=0,data=	z,addr=2 at time	240
rw=0,data=	5,addr=2 at time	280
rw=0,data=	z,addr=3 at time	320
rw=0,data=	7,addr=3 at time	360

图 4.5 例 4.18 的模拟结果

## 4.7 逻辑延迟建模

当在设计过程中对实际门级实现的时序和功能特别强调时,一般使用门级建模。对门和线网延迟进行建模,可以反映门和线网实际的布局 and 布线。在本节中,我们着重讨论逻辑门基元,并说明它们的模拟时序特征。

### 4.7.1 门级建模示例

在例 4.19 中,三态与非锁存器说明 bufif1 门的应用及其详细的时序信息。电路示意图如图 4.6 所示。当 enable 输入为 1 时,该锁存器驱动 qOut 和 nQOut 端口,这两个端口均被定义成三态线网。该 bufif1 门是一个三态功能的模型。如表 4.5 所示当控制输入

为 1 时,输出被驱动为跟随输入变化而变化。请注意 data 输入上的 z 在数据输出端变为未知值。当控制输入为 0 时,输出为高阻态(z)。

### 例 4.19 三态锁存器

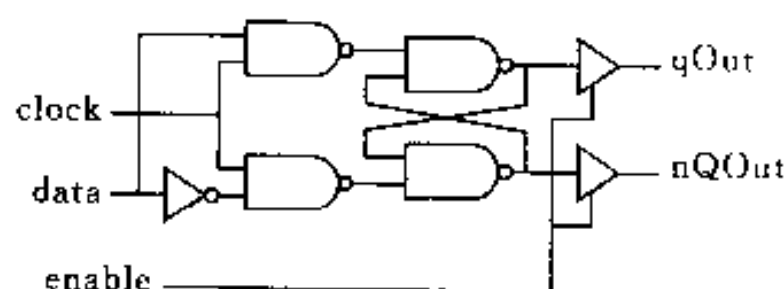
[illegible]

图 4.6 三态锁存器的示意图

**表 4.5 BUFI1 门函数**

		控制输入			
bufif1		0	1	x	z
D	0	z	0	L	L
A	1	z	1	H	H
T	x	z	x	x	x
A	z	z	x	x	x

当控制输入为  $x$  或者  $z$  时,我们用  $L$  和  $H$  为数据输出端建模。其中  $L$  表示输出为 0 或  $z$ ,  $H$  则表示输出是 1 或  $z$ 。

其他的三态建模基元包括 bufif0, notif1 和 notif0。其中 bufif0 和 bufif1 的控制输入刚好相反。对 notif1 而言, 当控制输入为 1 时反转数据输入并驱动数据输出; 对 notif0 而言, 当控制输入为 0 时反转数据输入并驱动数据输出, 在附录 D 中有这些三态建模基元



的真值表。

现在我们可以来描述例 4.19 的功能。基本的锁存器功能是通过交错连接的与非门 qQ 和 nQ 来实现的。当时钟处于低电平时, d 和 nd 的输出为高电平, 而且锁存器保存它们的值。当时钟处于高电平时, d 和 nd 把值传送出去并且改变锁存器的值。只要时钟处于高电平, 与非门 qQ 和 nQ 的值就跟随 data 输入的变化而变化。两个 bufif1 门分别被与非锁存器门的输出和 enable 输入的信号所驱动。对于 bufif1 门的每一个定义来说, 当 enable 为高电平时, 输出被驱动; 当 enable 为低电平时, 输出为 z。

参见: Verilog 门 D; 线网 4.2; 泛模型延迟(delays across a module) 4.8

### 4.7.2 门和线网延迟

门、持续赋值以及线网延迟提供一种精确的描述电路延迟的方法。门延迟描述了从其输入发生变化到其输出发生变化的延迟。持续赋值延迟描述了从右项的某个值发生变化到左项的值发生相应的变化, 并且这种变化被传送出去为止的延迟。线网延迟描述了从任何一个线网的驱动门或是赋值语句发生变化到该值被传送出去为止的延迟。默认的门延迟、线网延迟和赋值语句延迟均为 0。如果说明了一个延迟参数, 则该值使用于所有与门、线网和赋值语句有关的传输延迟。

下面的门例示语句是从例 4.19 中抽取出来的, 用来说明不同的传送情况。

```
not      #5          (ndata, data);
nand     #(12, 15)    qQ(q, nq, wa),
                        nQ(nq, q, wb);
bufif1   #(3, 7, 13)  qDrive(qOut, q, enable),
                        nQDrive(nQOut, nq, enable);
```

传输延迟由跳变到 1、跳变到 0、跳变到 z(关断延迟)来指定。非门被指定延迟为 5, 由于只给定了一个延迟值, 该延迟同样适用于跳变到 1 和跳变到 0 的情况。与非门示例中上升延迟为 12 和下降延迟为 15。bufif1 门的上升延迟为 3, 下降延迟为 7, 升高阻值的延迟为 13。请注意如果门处于高阻状态下, 则当 enable 变为 1 时, 需要延迟 3 个时间单位(例如上升延迟)来使输出变为 1。

一般地说, 用如下的形式进行延迟说明:

#(d1, d2)

或者

#(d1, d2, d3)

其中 d1 是上升延迟, d2 是下降延迟, 而 d3 则是高阻延迟。之所以存在这两种说明方式

的原因在于某些门仅允许指明两个延迟时间而有些门则允许指明三个延迟时间。d3 在特殊的情况下还有特殊的意义,当 d3 用于 trireg 线网时,它是连线的值变为 x 的衰减时间。延迟说明的语法总结如下:

delay2

```
::= #delay_value
|    #(delay_value[,delay_value])
```

delay3

```
::= #delay_value
|    #(delay_value[,delay_value[,delay_value]])
```

delay-value

```
::= unsigned_number
|    parameter_identifier
|    constant_mintypmax_expression
```

constant\_mintypmax\_expression

```
::= constant_expression
|    constant_expression;constant_expression;constant_expression
```

(请注意 mintypmax\_expression 的第二种形式,在 4.7.1 节中有详细的讨论。)

表 4.6 总结了具有两个延迟和三个延迟说明的模拟器所使用的传输延迟。需要指出的是,如果没有指明延迟说明,默认延迟为 0,当且仅当给定一个值时,所有的传送才被当成是取该延迟时间。

记住这些延迟的一个简单的方法是:上升延迟(d1)是从 0 到 x,从 x 到 1 或是从 z 到 1,与此类似,下降延迟是从 x 到 0,从 1 到 x 或是从 z 到 0。

例 4.19 所定义了三态线网并不包括其自身的延迟参数,然而,我们可以用如下的形式定义:

```
tri    #(2,3,5)  qOut, nQOut;
```

表 4.6 使用在模拟中的延迟值

源值	目标值	二延迟声明	三延迟声明
0	1	d1	d1
0	x	min(d1,d2)	min(d1,d2,d3)
0	z	min(d1,d2)	d3

续表

源值	目标值	二延迟声明	三延迟声明
1	0	d2	d2
1	x	min(d1,d2)	min(d1,d2,d3)
1	z	min(d1,d2)	d3
x	0	d2	d2
x	1	d1	d1
x	z	min(d1,d2)	d3
z	0	d2	d2
z	1	d1	d1
z	x	min(d1,d2)	min(d1,d2,d3)

在这种情况下,驱动任何一个线网的任何一个驱动源都会接受一个 2 单位的上升延迟,一个 3 单位的下降延迟,以及一个输出被传送之前保持为 *z* 的 5 单位的延迟。从而在例 4.19 中由于 bufif1 型的 qDrive 门例示驱动 qOut 线网,从门 qDrive 的输入发生变化到结果在 qOut 线网上被传送出去的上升延迟为 5(2+3),下降延迟为 10,*z* 延迟为 18。

在持续赋值情况下,当左边为一个向量时,通过测试右边的值来处理多个延迟。如果该值由非 0 变为 1,则使用下降延迟;如果值变为 *z*,则使用关断延迟,否则使用上升延迟。

参见:泛模块延迟(delays across a module) 4.8

### 4.7.3 时间单位的规定

在我们的示例中使用了“#”操作符来为硬件部分的模拟模型说明时间,然而我们还没有为延迟说明说明时间单位。我们一般使用时间尺度编译指令来说明。

该编译指令的形式为:

```
`timescale <time_unit>/<time_precision>
```

该指令为紧随其后的模块设定时间单位和精度。在一个描述中可以包括多条这样的指令。

表 4.7 timescale 编译指令的单位

度量单位	缩写
seconds(秒)	s
milliseconds(毫秒)	ms

续表

度量单位	缩写
microseconds(微秒)	us
nanoseconds(纳秒)	ns
picoseconds(皮秒)	ps
femtoseconds(飞秒)	fs

$\langle \text{time\_unit} \rangle$  入口是一个整数,  $\langle \text{time\_precision} \rangle$  是时间度量单位, 整数可能取值为 1, 10 或 100。时间单位缩写见表 4.7。一个模块可以跟在如下所示的时间刻度指令后面:

```
timescale 10 ns/1 ns
```

维持时间的精度为 1ns。在延迟说明中的值应该为 10ns 的整数倍, 也就是说, #27 表示延迟为 270ns。表 4.8 说明了延迟说明的几个示例。根据给定的时间刻度指令得出实际的延迟时间, 将小数点调整到合适的位置, 然后乘上时间单位, 就决定了模拟时间。

表 4.8 时间延迟和精度说明

单位/精度	延迟说明	时间延迟	说 明
10 ns/1 ns	#7	70 ns	延迟是 7 乘以时间单位, 即 70 ns
10 ns/1 ns	#7.748	77 ns	7.748 四舍五入到小数点后一位, 然后乘以时间单位
10 ns/100 ps	#7.748	77.5 ns	7.748 四舍五入到小数点后两位, 然后乘以时间单位
10 ns/1 ns	#7.5	75	7.5 四舍五入到小数点后一位, 然后乘以 10
10 ns/10 ns	#7.5	80	7.5 四舍五入成整数(没有小数位), 然后乘以 10

#### 4.7.4 最小延迟、典型延迟和最大延迟

Verilog 允许为每一个上升延迟、下降延迟和关断延迟说明三个值。这些值分别为最小延迟、典型延迟和最大延迟。

##### 例 4.20 最小延迟、典型延迟和最大延迟的说明

```
module IOBuffer (bus, in, out, dir);
    inout      bus;
    input      in, dir;
    output     out;

    parameter
```

```

R_Min = 3, R_Typ = 4, R_Max = 5,
F_Min = 3, F_Typ = 5, F_Max = 7,
Z_Min = 12, Z_Typ = 15, Z_Max = 17;

```

```

bufif1 # (R_Min: R_Typ: R_Max,
        F_Min: F_Typ: F_Max,
        Z_Min: Z_Typ: Z_Max)
      (bus, out, dir);

```

```

buf # (R_Min: R_Typ: R_Max,
       F_Min: F_Typ: F_Max)
    (in, bus);

```

```
endmodule
```

例 4.20 表示了这些延迟的说明。最小延迟、典型延迟和最大延迟用“:”隔开,而上升延迟、下降延迟和关断延迟用“,”隔开。一般说来,延迟说明形式:

```
# (d1,d2,d3)
```

可以被展开成下面的形式:

```
# (d1_min;d1_typ;d1_max;d2_min;d2_typ;d2_max;d3_min;d3_typ;d3_max)
```

这就是在前一节规范的语法说明形式中给出的 mintypmax\_expression 的第二种形式。最小延迟、典型延迟和最大延迟也可以在门级基元、线网、持续赋值语句和过程赋值语句中使用。

## 4.8 模块中的延迟路径

除了任何门级和其他在模块内部说明的内部延迟之外,指明模块中的延迟路径(从管脚到管脚)通常是非常有用的。specify 块允许在模块的输入和输出之间进行时序说明。例 4.21 说明了 specify 块的作用。

### 例 4.21 延迟路径说明

```

module dEdgeFF(clock,d,clear,preset,q);
    input  clock,d,clear,preset;
    output q;

    specify
        // specify parameters

```

```

specparam    tRiseClkQ=100,
              tFallClkQ=120,
              tRiseCtlQ=50,
              tFallCtlQ=60;

// module path declarations
(clock=>q)=(tRiseClkQ,tFallClkQ);
(clear, preset * >q)=(tRiseCtlQ,tFallCtlQ);
endspecify

// description of module's internals
endmodule

```

specify 块以关键字 specify 开头,并以关键字 endspecify 结束。在块内,声明了 specparams 和模块路径。specparams 命名在模块路径声明中使用的常量。模块路径声明列出了其输入(也叫作路径的起点)和输出(也叫作路径的终点)。时序说明会作用于该模块的所有示例。

在本例中,第一处模块路径声明指明了从 clock 输入到 q 输出所用的上升延迟时间为 100 时间单位而下降延迟为 120 个时间单位。第二处模块路径声明指明从 clear 和 preset 到 q 的延迟。在模块描述中延迟路径通常并不与延迟描述符( # )混在一起。然而,如果发生了这样的情况,两者之中的大者将用于模拟。

描述模块路径有两种方法,一种是使用“=>”,而另一种是使用“’>”。“=>”建立一个源输入位和目标输入位之间的并行连接(parallel connection)。输入和输出必须具有相同数目的位数,源的每一位都连接到目标的相应位。

“’>”建立源输入和目标输出之间的全连接(full connection)。源的每一位都有若干条路径到达目标中的所有位。输入和输出并不要求位数相同。在例 4.21 中,我们说明 clear 和 preset 有一个到 q 的输出路径,也可以说明多个输出。因此,我们可以用如下说明:

```
(a,b’>c,d) = 10;
```

这条语句与下面的四条语句是等价的:

```

(a=>c) = 10;
(a=>d) = 10;
(b=>c) = 10;
(b=>d) = 10;

```

在此我们假设 a,b,c 和 d 均为 1 位实体。我们还可以如下说明:

$(e \Rightarrow f) = 10;$

如果  $e$  和  $f$  均为 2 位实体,则这一条语句可以等价于以下的两条语句;

$(e[1] \Rightarrow f[1]) = 10;$

$(e[0] \Rightarrow f[0]) = 10;$

模块路径可以连接任何标量和矢量的组合,但是也存在一些限制。首先,模块路径的源必须被声明为输出型(out)模块或输入输出型(inout)模块;其次,模块路径的目标必须被声明为输入型(in)模块或输入输出型(inout)模块,并且被门级基元而非双向传输门所驱动。

每一条路径的延迟都可以用前一节中所讲的方法进行描述。我们可以说明最小延迟、典型延迟和最大延迟,也可以给出六个延迟值,它们的说明顺序为 0 到 1,1 到 0,0 到 z,z 到 1,1 到 z,z 到 0。需要补充的是,最小延迟、典型延迟和最大延迟可以被说明为其中的任意一个。

specify 块的正规语法可以在附录 G.8 中找到。在模拟器参考手册中有一些系统任务的描述,并允许在 specify 块中作一些时序检查,包括 setup、hold 以及脉宽检查。

## 4.9 小结

本章涵盖了使用 Verilog 语言进行逻辑建模的基本知识。我们已经知道如何定义门和线网,以及如何将它们连接为更复杂的模块,还了解到延迟和强度的作用,以及如何将模块定义参数化。

## 4.10 练习

4.1 编写一个具有下列结构的模块:

```
module progBidirect(ioA,ioB,selectA,selectB,enable);
    inout  [3:0]  ioA,ioB;
    input  [1:0]  selectA,selectB;
    input                enable;
    ...
endmodule
```

使得 selectA 可以按如下方式控制 ioA 的驱动:

selectA	ioA
0	no drive

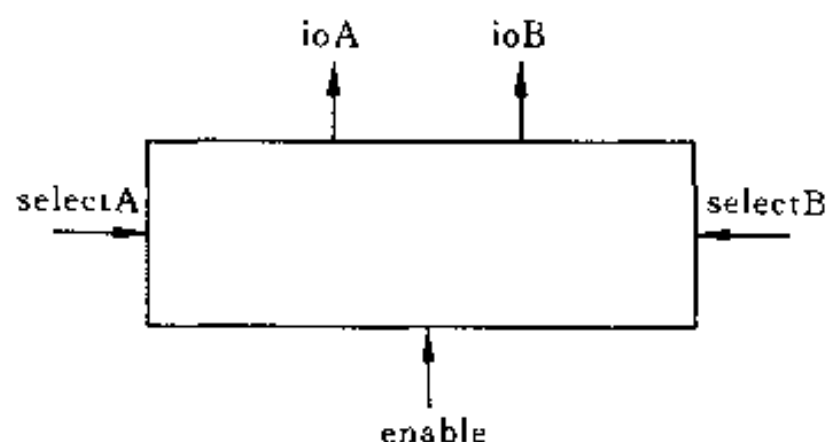
- 1            drive all 0's
- 2            drive all 1's
- 3            drive ioB

并且 selectB 以同样的方式控制 ioB 的驱动。当且仅当 enable 为 1 时驱动有效。如果 enable 为 0 时 ioA 和 ioB 驱动源为高阻态。

A. 仅使用门级基元书写该模块。

B. 仅使用持续赋值语句书写该模块。

- 4.2 改变例 4.3 中的汉明编码/解码器,使得为各个数据项设定的随机独立位可以通过一个有噪信道。
- 4.3 例 4.3 中的汉明编码/解码器查找和纠正单个错误位。通过加入第 13 位(它是另外 12 位的异或),双位错误也可以被发现(但无法纠正)。本例中加入这一点,改变有噪信道使得有时发生两位错。改变 \$display 语句指出该两位错。
- 4.4 使用从前一个问题中得到的双位检测/单位纠错电路改进例 4.18 中的存储器。将系统数据规模改为 8 位。当内存中写入一个字的时候,它以编码形式存放。当该字被读出时,它应该是经过译码和纠正的。加入一条总线,该总线由信号 slave 驱动,由 master 读出,指出出现两个错误的情况。找出一种方法,使得在出现两个错误的时候存储器中将要被破坏的数据可以被显示出来。



- 4.5 使用示例数组来说明一个多位全加器。模块头如下:

```
module fullAdder(cOut,sum,a,b,cIn);
```

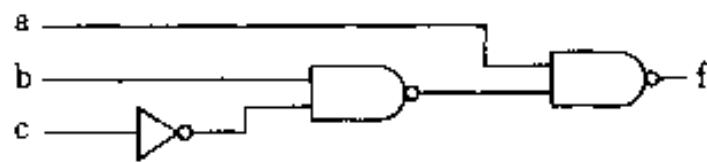
A. 当地址为 8 位, sum, a, b 均为 8 位元素, cOut 和 cIn 为一位元素时,描述该模块。

B. 将模块 fullAdder 中元素 sum, a, b 的位宽参数化。

- 4.6 下面的组合逻辑块具有三个输入和一个输出。该电路使用一种非同寻常的技术实现并加以分析。现在我们希望向电路中插入正确的输入输出时序信息(无需纠正中间结点的时序)。

下面是必须体现在电路中的时序关系:





- 输入端 a 和 b 与输出端 f 之间上升沿或下降沿的延迟:15 个单位时间
- 输入端 c 与输出端 f 之间上升沿或下降沿的延迟:10 个单位时间

以上的时序关系出现在逻辑图中是非常奇怪的,但是对于一些比较奇怪、却非常实用的时序延迟来说,这是一种非同寻常的技术,是一种晶体管实现。

假设 a, b 和 c 是同步触发器的输出。写一个功能和时序都正确的 Verilog 结构描述。

## 第5章 高级时序

前面的章节是基于对 Verilog 模拟器调度和执行事件方式的相对直接的理解来进行阐述的。本章将进一步阐述更具体的模拟器模型,包括语言中大量更细微的时序语义的处理。内容主要包括模拟器的调度算法、语言中不确定性的部分和非阻塞赋值等。

本章内容主要是从概念上解释 Verilog 模拟器是如何工作的。这里给出的描述可能不是任何示例的描述。然而其中有大量能被利用的捷径、技巧和实用描述。但是它们的使用法和适应性都不同,所以没有把它们作为这一章节的主题。

### 5.1 Verilog 时序模型

硬件描述语言用来对数字系统的功能和时序进行建模。这些模型的模拟过程是围绕事件来组织的。**事件**是指特定时刻模拟模型中数值的变化。Verilog 语言的语义规定一个事件导致其他事件及时发生的方式。按这些事件的顺序,模拟得以执行,模拟时间得以向前推进。

**时序模型**是决定模拟时间推进方式的模型——它与硬件描述语言的语义紧密相关。迄今为止,我们已经看到了两种 Verilog 语言使用的时序模型。这些时序模型是用门级和行为级描述形式给出的。

**模拟模型**不应该与**时序模型**相混淆。前者是数字硬件的模型,如 ALU 或寄存文件。后者是决定模拟器时间推进方式的模型。本章我们将讨论这些时序模型和模拟器推进时间的方式。

#### 例 5.1 NAND 锁存器

```
module nandLatch (q, qBar, set, reset);  
    output  q, qBar;  
    input   set, reset;  
  
    nand #2  
        (q, qBar, set),  
        (qBar, q, reset);  
endmodule
```

例 5.1 给出了一个简单的 NAND 锁存器。由 Verilog 语言的语义,我们可以知道当一个门的一个输入发生变化时,这个门便会根据它的输入值进行计算并判断是否改变输

出值。如果需要改变,在给定的门延迟(≠2)之后,便更新并传输输出值。门示例对输入值是敏感的。即任何一个输入值的变化,都将导致门示例模型的执行。

每一个模拟模型都有一个敏感表——模拟模型的输入表。当一个或多个输入发生变化时,将会导致模型的执行。敏感表不同于扇出表。扇出表由那些将产生新值的元素组成——它告诉我们当一个事件发生时需要计算哪些元素。敏感表由接受新值的元素组成——它告诉我们当发生变化时哪些输入的变化会导致模型的执行。

例 5.1 举例说明了 Verilog 门级时序模型。任意时刻任意输入发生变化,门示例将重新计算它们的输出,如果输出有变化,以后很可能会产生一个新的事件。所有的输入对变化总是敏感的,这种变化又将导致模拟模型的执行。门级时序模型应用于所有的门基元、用户自定义基元、持续赋值语句和过程持续赋值语句。持续赋值语句对表达式右侧任何时候的任何改变都敏感。也许在将来的某一时刻,这种改变会导致表达式的计算和对表达式左侧的赋值。

门级时序模型的另一个特征是关于新事件的调度。考虑一个表示门级时序模型的特定元素的一个事件已被调度但仍未被执行的情况。如果这个元素的输出又导致产生一个新事件,那么撤消先前已调度的事件,并调度新的事件。因此,如果门的输入端出现一个短于门的传输时间的脉冲,门的输出不会发生变化。惯性延迟是为了产生输出变化,输入端的值必须保持的最短时间。Verilog 门级模型具有惯性延迟,它刚好比传输延迟要长。这就是说只有当一个门的输入脉冲宽度大于它的传输延迟才会引起输出值的变化。正如我们将会看到的,如果输入脉冲宽度正好等于传输延迟时,输出值是否发生变化是不确定的。这一点对于任何表示门级时序模型的元素来说都是相同的。

现在来看一看例 5.2 所示的 D 触发器的行为模型。Verilog 语言的语义告诉我们,这个 always 语句首先执行,它将等待输入端 clock 的上升沿。当产生了一个上升沿时,模型延迟 5 个单位时间后把输入端 d 的值赋给输出端 q,然后继续等待 clock 的下一个上升沿的到来。本例是一个不同于门级时序模型的时序模型。

### 例 5.2 D 触发器的行为模型

```
module DFF(q, d, clock);  
    output    q;  
    input     d, clock;  
    reg       q;  
  
    always  
        @ (posedge clock)  
            #5 q = d;  
endmodule
```

这个 always 语句可以被看作有两个输入(clock 和 d)和一个输出(q)。这个 always

语句不像门级时序模型,它不是对任何时刻的任何变化都敏感。它的敏感性依赖于控制的上下文。举例而言,在这个 always 语句等待 5 个单位时间延迟期间出现的另一个 clock 上升沿是不起作用的。事实上,模拟模型发现不了第二个上升沿。直到那 5 个单位时间结束,它才会等待下一个 clock 变化沿。它只对 5 个单位时间以后的 clock 上升沿才敏感。因此这个 always 语句在模型执行到 @ 处只对 clock 敏感。更进一步说,这个 always 语句对输入 d 是不敏感的——d 的变化不会导致这个 always 语句产生任何动作。

这个示例说明了在行为块中出现的 Verilog 过程时序模型。这个行为块中包含了 initial 和 always 语句,一般来说,initial 和 always 语句只对输入的一个子集敏感,这种敏感性是随着模型执行时间而改变的。因此这些敏感元素是根据行为模型的当前执行部分来确定的。

过程时序模型的另一个特征是关于事件调度方式的。假设一个寄存器的一个更新事件已经被调度。如果再调度同一个寄存器的另一个更新事件,即使是在同一个时刻,前一个事件也不会被取消。因此,一个实体(比如寄存器)的事件列表中可能有多个事件。如果同时有几个更新事件,那么它们执行的顺序是不确定的。这与门级时序模型不同,在门级时序模型中同一个输出的新事件将撤消先前的未执行的事件。

模拟模型可以有交叉,这可以通过使用两个 Verilog 时序模型来建立。事实上,根据输入敏感元,过程时序模型可以建立门级时序模型所能建立的模型。为了理解这一点,让我们看例 5.3 中的与非门模型的行为描述。这个模型中使用了带控制事件("@")的 or 结构来模拟门级时序模型的输入敏感元。若 in1, in2 或 in3 中有一个发生变化,则重新计算输出。因此,过程时序模型可以用来模拟门级时序模型的输入敏感元。然而如前所述,过程时序模型可以有其他的时序敏感元,使其更灵活多变。

### 例 5.3 时序模型中的交叉

```
module behavioralNand (out, in1, in2, in3);
    output      out;
    input       in1, in2, in3;
    reg         out;
    parameter   delay = 5;

    always
        @ (in1 or in2 or in3)
            #delay out = ~(in1 & in2 & in3);
endmodule
```

例 5.3 和一个三输入 NAND 门示例有一些细微的差别。第一,过程赋值语句使得行为模型在门的传输延迟期间对输入不敏感。第二,如果门级时序模型的输入发生变化,并且已经调度了一个新的输出,那么撤消先前的更新事件,调度新事件。

总的来说,一个 Verilog 描述中的元件包括门级时序模型和过程时序模型。这些时序模型定义了语言中的两大类元件,用来描述它们是如何对其输入敏感的。此外,也提供了两种事件调度方法。

## 5.2 模拟器的基本模型

在本节中,我们将详述事件驱动模拟器的内部工作方式,特别是模拟器如何处理产生事件的模拟模型的执行过程,以及导致其他模拟模型被执行的事件的传输方式。时序模型需要重点理解,因为每一个模型因模拟算法不同而有不同的动作。

### 5.2.1 门级模型

首先考虑一个模拟器的基本操作,它模拟图 5.1 中所示的门级模型。假定每一个门都有  $d$  个单元的延迟。假设开始时,逻辑门有一个如图 5.1(a)所示的稳定值。到时刻  $t$  时  $A$  的输入由逻辑 0 变成逻辑 1,如图 5.1(b)所示,这样就产生一个事件。同时,计算门  $g1$  的值,判断  $B$  输出端是否会有变化。 $B$  将变为 0,而且这个事件将被调度到  $d$  个单位延迟后发生。

在  $t+d$  时刻,门  $g1$  的输出  $B$  被置为 0,如图 5.1(c)中箭头所示,这个新值将被传输到门  $g1$  的扇出。由于  $g1$  的输出与  $g2$ 、 $g3$  相连,便导致这两个门的计算,来判断是否  $B$  上的事件会引起  $g2$ 、 $g3$  的输出变化而产生新的事件。可以看到,只有  $g2$  的输出  $C$  有变化。这个事件( $C=1$ )将调度到下一个  $d$  单位延迟后执行。图 5.1(d)给出了这个时间之后的  $t+2d$  时刻的结果。此时, $C$  上的新值将被传输到与门  $g2$  扇出端相连的所有门。依此类推,这些门将计算并调度新产生的事件。

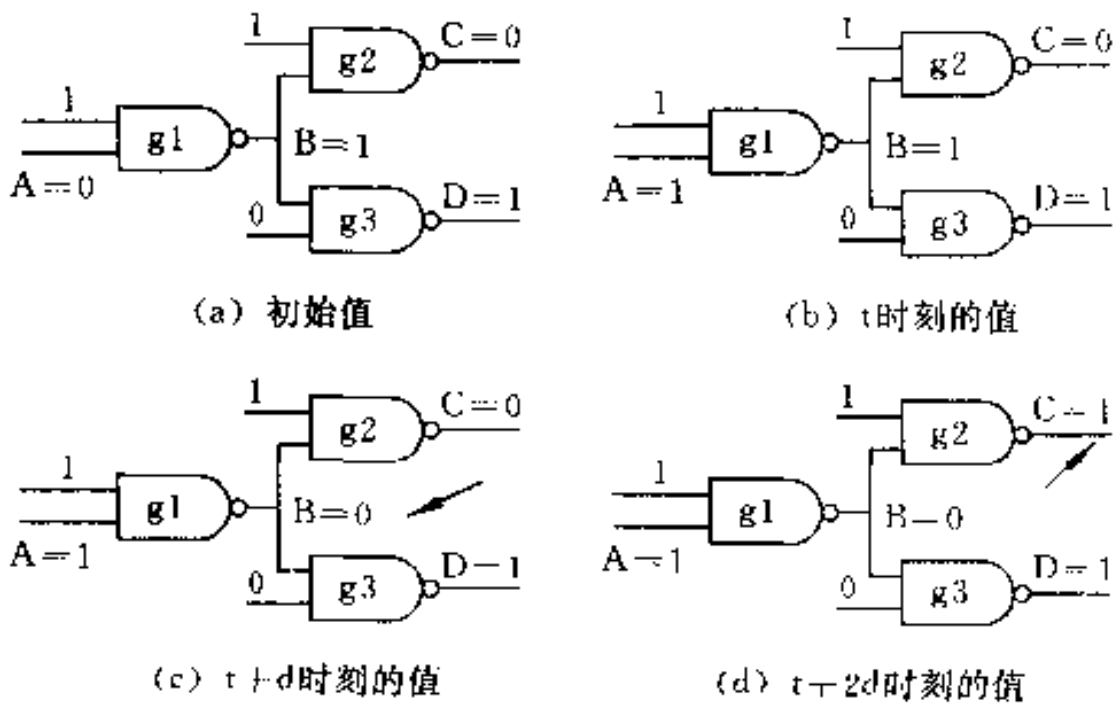


图 5.1 门级电路的模拟

5.2.2 更通用的模型

显然,模拟模型中门级事件驱动模拟器需要跟踪计算所有门示例的输出值。在新事件产生时,还要跟踪每门示例的扇出表。事件被存储在一个**事件列表**的列表中。**事件列表**的列表按时间排序。将来的每一个时刻,都有一个事件列表;特定时刻的所有事件保存在一起,模拟器调度器将跟踪新事件的发生并维护事件列表。调度器可以通过将事件插入到事件列表中来把事件调度到将来的某个时刻,也可以把事件从事件列表中删除从而达到取消调度的目的。

至此,我们已经定义了在一特定的时刻由一个值发生变化而产生的事件。从现在开始,我们来讨论**更新事件**和**计算事件**的差别。在某个时刻,更新事件会引起一个值的更新。计算事件将导致一个门(或我们稍后将看到的行为模型)的计算,可能会产生一个新的输出。事实上,更新事件会导致计算事件,计算事件也可能导致更新事件。

图 5.2 例示了一个事件驱动模拟器的主要元件的互连。模拟调度器被视为系统的主要部分。每一个与其相连的箭头旁都有一个标签写明了调度器所执行的动作。从上一节我们可以知道当前更新事件被从事件列表中删除,同时门的输出被更新。这些更新事件使得调度器查找扇出表,判断哪些门需要重新计算。计算这些门,任何输出值的变化都将产生一个将要调度的更新事件。

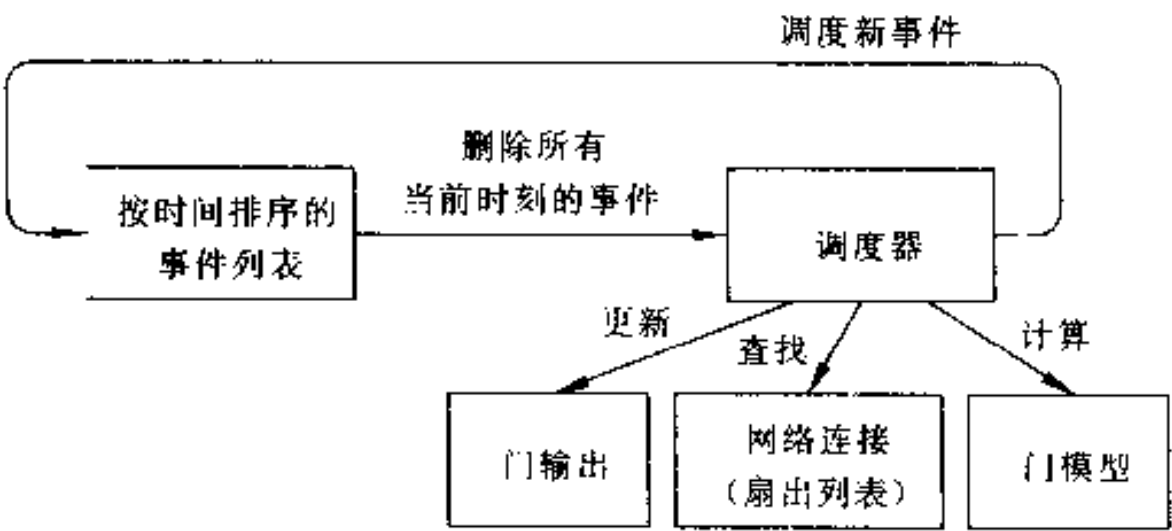


图 5.2 事件驱动模拟器的组织

图 5.3 列出了一个模拟器的调度程序的一个算法。这里我们看到了一个事件驱动模拟器的典型流程。每一次外层循环(while)称为一个**模拟周期**。由于事件列表是按时间顺序组织的,找到下一时刻的事件来执行非常容易;它们肯定位于表首。如果当前时刻再无其他事件了,那么 currentTime 的值将更新为下一顺序事件的时刻值。currentTime 的所有事件都要从事件列表中删除。这些事件按由“**For each**”语句指定任意的顺序被选中执行。

```

while (there are events in the event list) {
    if (there are no events for the current time)
        advance currentTime to the next event time
    Unschedule (remove) all the events scheduled for currentTime
    For each of these events , in arbitrary order{
        if (this is an update event){
            Update the value specified
            Evaluate gates on the fanout of this value and Schedule update
            events for gate outputs that change
            Schedule evaluation events for behaviors waiting for this value
        }
        else {           // it's an evaluation event
            Evaluate the model
            Schedule any update events resulting from the evaluation
        }
    }
}

```

图 5.3 两级事件驱动模拟器的一个模拟周期

如果选择的是一个更新事件,赋值将被执行,然后处理产生扇出表,建立一个需要重新计算的门列表。表中的门都要计算,输出结果的任何变化又将作为一个更新事件被调度。如果扇出上有改变,将调度相应的计算事件。如果选择的是计算事件,门和行为模块将被激活。任何输出的改变将引起一个更新事件的执行。注意,新的更新事件也许是当前时刻的事件(例如,零延迟的情况)。这个事件仍将插入到事件列表中,并在下一个外层循环中被删除。因此,当前时刻会有几个模拟周期。

让我们按本模拟算法来看看事件列表是如何随时间变化的。图 5.4(a)列出了图 5.1 中示例的初始事件列表。列表中未处理的事件用黑体字表示,而已处理的(旧的)事件用普通字体表示,这样可以看到时间的推进(注意:在模拟器中已处理的事件是要被删除的)。当更新事件  $A=1$  被从列表中删除后,门  $g1$  的计算完成。由于门  $g1$  的输出发生了变化,所以  $B=0$  的更新事件被调度到  $t+d$  时刻。如图 5.4(b)所示,本事件被插入到事件列表中,下一个模拟周期又重新开始,并且时间推进到  $t+d$  时刻。在那个时刻,执行更新事件  $B=0$ ,它导致了门  $g2$  和  $g3$  的计算。仅当门  $g2$  发生了变化,才在  $t+2d$  时刻调度更新事件  $C=1$ ,如图 5.4(c)所示。同理,在下一模拟周期更新事件  $C=1$  又将被执行。

目前为止的讨论是围绕模拟门级模拟模型的,它们表示了门级时序模型。那就是说,当一个门的输出有一个事件发生时,所有与这个门的扇出相连的门都将被重新计算,以便确定它们的输出是否发生了变化。下一节将扩展我们对能处理行为模型的模拟器的理解。

(a)	Update A=1 at t		
(b)	Update A=1 at t	Update B=0 at t+d	
(c)	Update A=1 at t	Update B=0 at t+d	Update C=1 at t+2d

图 5.4 事件列表

### 5.2.3 行为级模型的调度

在 Verilog 中的行为模型是过程时序模型。因此这些模拟模型只是对它们的输入子集敏感。而敏感元会随模型的执行而改变。本节中将涉及行为模拟模型的各个方面,包括扇出表的处理和寄存器的更新。

例 5.4 是一个主从锁存器的行为模型。锁存器的操作都依赖于两个时钟相位 phi1 和 phi2。首先,锁存器等待 phi1 的上升沿的到来。当 phi1 的上升沿到来时,d 的值将被保存到锁存器的一个寄存器,然后等待 phi2 上升沿的到来。当 phi2 的上升沿到来时,输出 q 被赋予 qInternal 的值,并且开始下一次 always 循环的重复执行。

#### 例 5.4 主从锁存器

```

module twoPhiLatch (phi1, phi2, q, d);
    input    phi1, phi2, d;
    output   q;
    reg      q, qInternal;

    always begin
        @(posedge phi1)
            qInternal = d;
        @(posedge phi2)
            q = qInternal;
    end
endmodule

```

重要的一点是认识到在本行为模型的执行中,模型交替地对 phi1 和 phi2 的上升沿敏感,而对输入 d 不敏感。每一个行为模型都有一个敏感列表——一个说明哪些输入在当前对 always 和 initial 语句敏感的列表。因此我们可以通过检查所有的 always 和 initial 语句来判断哪些更新事件可以导致时间函数的计算。



为了模拟,使用了在 5.2.2 节主框架中的过程时序模型的元件,我们需要能在模拟模型执行时更改扇出表。这就是说,由于敏感表变了,则扇出表也需要相应的改变。举例来说,在例 5.4 模拟开始时,控制事件("@")被执行,并且包含了控制事件的进程语句(如 always 语句)被放在 phi1 的扇出表中,当 phi1 上出现一个更新事件时,这个进程的一个计算事件便被调度。当这个计算事件执行时,进程同时判断有无上升沿。如果没有,那么 always 块继续等待——扇出表此时也不会改变。如果有上升沿,进程将被从 phi1 的扇出表中删除,并且行为语句恢复它们的执行(在本例中是执行"qInternal=d;")。当下一个控制事件执行时,进程被放入 phi2 的扇出表中。此时,任何在 phi2 上的变化都将导致本进程的一个计算事件被调度。当计算事件执行且出现上升沿时,进程被从 phi2 的扇出表中删除,继续行为语句的执行,并且在下一个控制事件,进程又置入 phi1 的扇出表中。

总之,一个控制事件或不满足条件的 wait 语句的执行将使得当前进程语句(always 或 initial 语句)被挂起并且被放入事件或条件表达式中相应元素的扇出表中。当某个元素的一个事件发生时,该进程将调度一个计算事件来判断是否满足事件或 wait 语句的条件,若满足条件时,该进程语句将从扇出表中删除。

不仅把进程语句放入到扇出表,还需要保存一个指针来指示以后从哪儿开始恢复执行。这与软件程序从输入设备中读入数据的情况相类似。操作系统会挂起程序的执行直到得到了输入数据。一旦获得输入数据,程序将从被挂起的地方继续执行。一个被挂起的 Verilog 进程也以类似的方式从被挂起的地方恢复执行。

现在看例 5.4 的改进版本例 5.5。两者唯一的不同之处是后者的每个过程赋值语句前面都有两个单位时间的延迟。在延迟被执行时,模拟器执行的操作是挂起进程语句,并且把一个计算事件调度到将来的某个适当的时刻。进程的敏感列表没有变化。因此,不是在 phi1 的上升沿到来后执行语句"qInternal = d",而是把一个进程的计算事件被调度到两个单位时间后执行。在那一时刻,进程将恢复 qInternal 的赋值操作的执行。

行为模块的执行会导致寄存器被写入新值。这些值是立刻被写入的,无需去生成一个更新事件。因此,如果一个寄存器在过程表达式的左边被赋值,并且需要立即用于下一个语句的右边时,用到的值是新值。此外,进程输出用的寄存器将会产生更新事件。所以,如果一个寄存器被用作持续赋值语句或连线的源,或者有另一个进程在等待(如@或 wait)该寄存器的变化,那么更新事件将导致计算事件的调度。

行为模块说明了过程时序模型。如果我们允许在模块执行期间改变扇出表,并且如果要更新寄存器的值,以便对下一条行为语句产生影响,那么,它可以用图 5.3 的算法来模拟。

### 例 5.5 行为模型中的延迟

```
module twoPhiLatchWithDelay (phi1, phi2, q, d);  
    input    phi1, phi2, d;
```

```

output      q;
reg         q, qInternal;

always begin
    @(posedge phi1)
        #2 qInternal = d;
    @(posedge phi2)
        #2 q = qInternal;
end
endmodule

```

## 5.3 模拟算法的不确定行为

Verilog 是一种并行语言,允许描述多个同时发生的动作。但执行这些操作时要求将它们序列化。因为所使用的计算机不同于你所建模的硬件,它不是并行的。在 Verilog 语言中有两个不确定行为的来源:在零时刻的任意的执行顺序和进程之间语句的随意交叉。本节对这些方面作一些讨论。虽然每个模拟器对相同的模拟模型和相同的输入给出相同的结果,但是,不同的模拟器版本或不同供应商的模拟器也许选择不同的顺序来执行这些相同的事件。因此,我们说结果是不确定的。

### 5.3.1 临近不确定区

模拟器执行被调度到同一时刻的一组事件也许需要几个模拟周期来完成,因为一些事件可能产生本时刻的其他事件。这里说的执行同一时刻的事件是指在长度为零的时间内执行它们。这并不是说执行这些事件不需要时间,而是说所有事件的发生并不使模拟时间推进。它们都在长度为零的时间内发生。

图 5.3 的调度算法中的“Foreach”语句从当前模拟时刻的事件列表中删除一个或多个事件。进一步而言,模拟器对这些事件的顺序安排是任意的。这种零长度时间内事件的任意执行顺序是模拟语言非确定性的根源。编写模块时,需要对在零长度时间内事件顺序的不确定性特别注意。

例 5.6 所示的是一个能体现非确定性的设计。该例有三个行为进程(一个 initial 语句和两个 always 语句)和一个门模型。假定在某时刻  $q = 0, f = q\text{Bar} = b = 1$  并且  $a = 0$ 。此后,  $a$  从 0 变为 1。这个变化会产生一个上升沿,从而使第一个 always 语句开始执行。这个 always 语句将延迟 10 个时间单位,再把  $b$  的值赋给  $q$ (值为 1),然后等待  $a$  的下一个上升沿的到来。为  $q$  赋新值将为  $q$  的扇出上的两个元件产生两个计算事件——第二个 always 语句和非门。在下一个模拟周期,这些事件都将从事件列表中删除并且按

任意的顺序执行。注意：输出值  $f$  会因执行顺序的不同而不同。如果先执行 `always` 语句， $f$  将保持 1 的值不变。如果先执行非门， $f$  的值将被置为 0。

### 例 5.6 零时间问题

```
module stupidVerilogTricks (f, a, b);
    input    a, b;
    output   f;
    reg      f, q;

    initial
        f = 0;
    always
        @ (posedge a)
            #10 q = b;

    not      (qBar, q);

    always
        @ q
            f = qBar;
endmodule
```

哪种结果才是正确的呢？基于该语言的语义，两个结果都是正确的。因为模拟器允许以任意的顺序从当前时刻的事件列表中取出事件并执行它们。如果想要  $q$  和  $qBar$ （即  $f$ ）总是一对相反的值，那么需要改写一下模拟模型。例如，一种方式是把第二个 `always` 语句和非门示例相合并。如例 5.7 所示把它们合并为一个 `always` 语句。这种解决方法将保持时序。也可以在例 5.6 中的“ $f = qBar$ ”语句前增加一个“ $= 1$ ”，这将确保  $f$  得到“正确”的赋值——然而，模型的时序特性也将有所变化。一个保持时序关系的解决方案是用“ $\#0$ ”代替“ $\#1$ ”放在“ $f = qBar$ ”语句的前面，这种方式将在 5.4 节进一步讨论。

### 例 5.7 例 5.6 的修正

```
always
    begin
        @ q
            qBar = ~q;
            f = qBar;
    end
```

虽然上面举的示例是特意设计的，但可以肯定的是，非确定性也出现在任何其他非特意设计的描述中。现在让我们观察例 5.8 中的波形计数器。在计数器结构上有两个 D 触发器与其相连，低位的触发器（示例 a）以触发器的方式连接。高位（b）把低位作为它的

输入。我们希望在时钟上升沿,计数器会以状态 00,01,10,11,00,... 的方式累加。然而,进一步研究发现,dff 的两个示例中的“q = d”语句被调度到三个单位时间后执行。在那个时刻,调度器将把这两个计算事件同时从事件表中删除,并且以任意的顺序执行它们。当然执行的顺序决定结果。先执行示例 a 会导致不正确的计数顺序(00,11,00,...)。先执行 b 才会产生希望的序列。

这个问题可以通过在 dff 模块中使用内部赋值延迟语句“q = #3 d;”来解决。在对 q 的示例更新前,这个语句将导致所有 dff 示例的 d 输入都被取值并作为更新事件存储到事件列表中。因此,示例可以以任意的顺序执行而且行为是确定的。本问题同样可以通过在 dff 模块中使用非阻塞语句“q <= d;”来解决。这里非阻塞赋值语句与时钟边沿一同把对所有 d 的读操作和 q 的写操作相分开。

零长度时间内的事件可以按任意的顺序执行是该语言基本定义的一部分。设计中的不确定行为反映了建模结构的少量用法或一个实时竞争条件。该语言中允许非确定性有效率上的考虑,也是因为它确实存在现实的世界中存在。应该对无竞争地写入给以注意,以便使得无论零长度时间内的执行顺序如何,结果是确定的。

#### 例 5.8 触发器模型中的非确定性

```
module goesBothWays (Q, clock);
    input    clock;
    output   [2:1]    Q;

    wire     q1, q2;
    assign    Q = {q2, q1};

    dff      a (q1, ~q1, clock),
            b (q2, q1, clock);
endmodule

module dff (q, d, clock);
    input    d, clock;
    output   q;
    reg      q;

    always
        @(posedge clock)
            #3 q = d;
endmodule
```

### 5.3.2 Verilog 是一种并发语言

Verilog 语言中第二个不确定的原因是由于不同行为进程语句的潜在交叉。对行为

进程模型而言,行为语句存在于 always 和 initial 的语句中。更新事件和所有的计算事件要求行为进程模型的执行是原子动作。要保证这些事件在执行完之前不能执行其他事件。initial 和 always 语句中的行为进程模型是按不同的规则存在的。

首先我们考虑一个软件编程环境。在一个普通的编程语言(如 C 语言)中,一个独立的进程是以“main”函数开始和结束的。当它执行时,我们希望按编写的顺序执行语句,并且每一个被计算和更新的值与在下一行中当作源使用的值是相同的。事实上,这是只有单个进程的情况。然而,如果有多个进程并且这些进程共享信息——在同一内存地址存入值,这样可能一个变量的值在下一行程序中用到时因其他进程的更新而改变。

接着以上的分析,图 5.5 是在并行编程环境中运行的两个进程的摘录。每一个进程是自身的控制线程,按自身的速度执行。但两个进程共享一个变量——本例中两进程中的变量 A 引用同一个内存地址。如果这两个进程正在一个处理器执行,那么进程 A 可能运行一段时间,然后进程 B 便开始运行,然后又是 A,以这种方式进行下去。操作系统的调度程序以一种公平的方式分配时间片,中止时间片用尽的进程而开始另一个进程。当然,这样可能产生问题,如果进程 A 正好在“ $a = b + c$ ”语句执行完时停止,进程 B 接着执行。进程 B 将改变 A 的值,该值对进程 A 可见。当进程 A 最后恢复执行时,计算出来的 q 的值将不同于进程 A 在不被中断的情况下执行完这两条语句所得到的值。

另外,可以在两个具有共享内存的并行处理器上执行这两个进程,它们共享变量 A。在这种情况下,进程 B 也可能在进程 A 的两条语句之间执行“ $a = a + 3$ ”而改变 A 的值,仍会引起 q 值的不确定。

这种情况发生的几率是多少? Murphy 规则指出可能性是大于零的。

进程 A	进程 B
...	...
$a = b + c;$	$a = a + 3$
$q = a + 1;$	...
...	

图 5.5 两个并发进程之间的不确定性

在一个软件并行编程环境中,能够保证任何进程的一切语句都按编写的顺序执行。然而,在一个进程的两个语句之间,其他进程的语句可能会插入。假设给出一个这样的并行编程环境,那么可以有很多种不同的交叉方式。我们把任何一种进程与进程的语句交叉方式都称作一个方案。下面两个方案为 q 给出如上所述的两个不同值。

方案 1	方案 2
A: $a = b + c$	A: $a = b + c$
B: $a = a + 3$	A: $q = a + 1$
A: $q = a + 1$	B: $a = a + 3$

那么哪一种方案是正确的呢？根据对并行编程环境的一般了解，两种方式都是正确的！如果程序员希望方案 2 是  $q$  的正确计算方式，那么他将不得不改变描述来保证这是  $q$  的唯一计算方式。在一个并行编程环境中，任何对  $a$  的访问都被视为临界区。当程序代码处于临界区时，操作系统必须确保同一时刻只有一个进程在临界区中执行。假设图 5.5 中进程 A 和 B 的语句处于临界区中，以便保护共享变量  $a$ ，那么方案 1 是不可能的。那就是说，操作系统不会允许进程 B 执行“ $a = a + 3$ ”，因为进程 A 还在临界区中。

上面讨论的并行编程环境正是 Verilog 进程的执行环境。特别地，行为进程的执行规则如下：

- Verilog 进程 begin-end 块中的语句（如 always 和 initial 中的语句）都能确保按编写的顺序执行。
- 一个 Verilog 进程的语句可能会与其他的 Verilog 进程的语句产生交叉。
- 根据范围规则的判断，具有相同实体的寄存器和线是相同的。当一个进程在使用这些共享实体时，它们可以被另一个进程改变。

因此许多方案都是可能的。事实上，进程可以调用没有变量的拷贝的函数和任务——它们是共享的。设计者的任务是保证只有正确的方案才可能。一般来说，这些情况中的肇事者都是共享的寄存器或线。任何能被多个进程写的寄存器或线都会使交叉存取的问题更突出。

有时候我们会希望当前进程挂起足够长的时间以使某些值有时间来传播。考虑一下例 5.9。b 的打印结果是不确定的——值可能是  $x$  或 1。如果初始进程顺利执行完，b 的值将为  $x$ 。然而，正如上述语义所述，有可能当  $a$  被赋值 1 时，行为进程被挂起，b 的值可能接着被更新。当行为进程重新开始执行时，display 语句将显示 b 的值被设成了 1。改变 display 语句，在它的前面加一个“#0”会使得初始化进程被挂起，b 将被更新，并且当 display 语句继续执行时，b 的值将显示为 1。

有趣的是，在并行软件语言中，当多个进程要共享信息时，提供了高级方法来同步它们。P 和 V 信号量就是一种方法；临界区是另一种方法。为了使这些方法有效，有些指令（如：“test and set”）是原子操作——它们执行中不得被任何其他进程中断。这些指令出现在“零长度时间”中时，为更高层的同步基元提供了基础。在硬件上，进程之间的同步由时钟边沿、互锁握手信号和某些情况下的同步限制来维持。

### 例 5.9 并发进程的挂起

```
module suspend;
    reg      a;
    wire     b = a;

    initial begin
        a = 1;
```

```

    $display ("a = %b, b = %b", a, b);
end
endmodule

```

## 5.4 非阻塞过程赋值语句

过程赋值语句有两个作用：第一是为行为语句的左侧赋值，第二是控制赋值语句执行时间的调度。Verilog 的两种类型的过程赋值语句（非阻塞的和阻塞的）都以不同的方式来完成这些功能。

### 5.4.1 阻塞和非阻塞赋值的比较

操作符“ $\leq$ ”用于非阻塞赋值，操作符“ $=$ ”是用于阻塞赋值。过程赋值语句中， $\leq$ 操作符能放在任何可以放置 $=$ 的位置上。非阻塞赋值操作符不能用在持续赋值语句中。虽然 $\leq$ 也用作小于等于操作符，使用的上下文决定它是表达式的一部分还是过程赋值语句的一部分。如果是表达式的一部分则表示小于等于的意思，如果是过程赋值语句的一部分则表示非阻塞赋值的意思。

考虑两条语句：“ $a = b;$ ”和“ $a \leq b;$ ”。这两条语句执行的功能是相同的——把  $b$  的当前值赋予寄存器  $a$ 。事实上，如果每个语句执行时  $b$  的值都是 75，那么  $a$  的值将为 75。对以下成对的语句也是一样的：

- “ $\#3 a = b;$ ”和“ $\#3 a \leq b;$ ”

和

- “ $a = \#4 b;$ ”和“ $a \leq \#4 b;$ ”

两组语句的每一组执行后，保存在  $a$  中的结果都相等。这些语句之间的不同之处在于实际赋值的方式以及对其他赋值语句还有哪些方式。

让我们分析语句“ $a = \#4 b;$ ”和“ $a \leq \#4 b;$ ”之间的差异。前者计算  $b$ （它可能是表达式）的值，把值存入一个内部临时寄存器，并且通过把自身调度为 4 个单位时间后  $a$  的一个更新事件来延迟 4 个单位时间。当这个更新事件执行时，内部临时寄存器的值被写入  $a$  并继续进程的执行。这也可以写成如下形式：

```

bTemp = b;           组合的, 正如 a = #4 b;
#4 a = bTemp;

```

语句（“ $a \leq \#4 b;$ ”）计算  $b$  的值，为  $a$  调度一个更新事件到 4 个单位时间之后。在当前时刻继续执行进程。这表明它没有阻塞或挂起行为进程，因此命名为非阻塞。注意到两种情况中，被赋给  $a$  的值是语句开始执行时  $b$  的值。然而新值将在 4 个单位时间后



被赋给 a。因此如果下一条语句把 a 当作一个源寄存器,将看到的值是 a 的旧值,而不是刚计算出的新值。

以下的两段 Verilog 语句比较了这两种形式的赋值。左边的阻塞赋值导致 b 的值将在 begin-end 块开始执行 4 个单位时间后被赋予 c。右边的非阻塞赋值中,第一条语句将 a 调度到 2 个单位时间后得到 b 的值。因为这条语句是非阻塞的,在当前时刻将继续执行,并且 c 被调度到 2 个单位时间后得到 a 的值。结果,c 在两种情况下结果是不同的。

begin	begin
a = #2 b;	a <= #2 b;
c = #2 a;	c <= #2 a;
end	end

阻塞和非阻塞除了定义之外,还有另一个主要的差异。这就是模拟调度算法处理更新事件的时间不同。在 5.2.3 节中,我们仅讨论了阻塞赋值结果的更新方式。它们是立即更新的,所以下面的行为语句将使用新值。如果它们也是进程的输出,那么它们也将被放入当前时刻的事件列表中。这种情况下它们将在下一个模拟周期被传输。非阻塞赋值语句产生的更新事件被存放在事件列表的一个单独的部分。这些更新事件直到被调度到当前时刻的所有更新和计算事件被执行完以后才会被执行——包括那些由当前事件产生的事件。那就是说,只有当前时刻的所有事件是非阻塞更新命令时,它们才被处理。当然,非阻塞更新可能导致其他被调度到当前时刻的事件列表中的计算事件。

#### 5.4.2 非阻塞赋值的一般用法

如第 1 章中所述,非阻塞赋值主要用于例 5.10(例 1.7 的修订版)中给出的边沿识别符。非阻塞赋值的作用是把时钟边沿到来前存在的值和那些由时钟边沿产生的值区分开。这里,非阻塞赋值语句右侧的值是时钟边沿到来前已存在的值,而左侧的值则是由时钟边沿到来时产生的。

使用非阻塞赋值语句会导致这两条语句并行执行,即在同一时刻发生。always 块中的第一条语句被执行并且 cS1 的非阻塞更新被调度到当前时刻。然而,更新动作不是立即执行的,而是开始执行第二条语句。这里 cS0 的非阻塞更新被调度到当前时刻。更新动作也不是立即执行的,而是继续执行 always 块的语句,等待时钟的下一个上升沿。结果,第一行求得的 cS1 的值与在下一条语句右侧所引用的值是不同的。

cS1 和 cS0 的值是何时被更新的呢?只有在当前时刻的所有阻塞更新都执行完后,它们的值才被更新。这其中包括由它们产生的任何阻塞更新或计算事件。因此所有的右侧都将在任何左侧的值被更新前进行计算。结果,所有的非阻塞赋值在整个设计中都同步地进行下去。为 cS1 和 cS0 赋值的这两条语句的顺序在描述中交换后不影响结果。

非阻塞赋值语句的重要特征不仅仅在于本模块中的两条语句是同步的,而是在整个



设计中,任何 `always` 或者 `initial` 语句中等待同一个变化沿的所有非阻塞赋值语句都是同步的。

### 5.4.3 事件驱动调度算法的扩展

一个模拟调度算法(图 5.3 所示)的扩展版本如图 5.6 所示。这里有部分改动。首先,术语规则事件被用来表示除非阻塞更新事件以外的所有其他事件。因此规则事件包括行为进程和门模型的阻塞赋值更新和计算事件。第二,程序中第二个 `if` 语句的 `then` 子句被修改,当所有的规则事件执行完后,寻找非阻塞更新事件。概念上非阻塞更新事件将转变成规则事件,以便余下的调度算法可以直接处理它们。最后,所有上述事件执行完毕之后,监视事件被处理。

```
while (there are events in the event list){
    if (there are no events for the current time)
        advance currentTime to the next event time
    if (there are no regular events for the current time)
        if (there are non-blocking assignment update events)
            turn these into regular events for the current time
        else
            if (there are any monitor events)
                run these into regular events for the current time
    Unschedule (remove) all the regular events scheduled for currentTime
    For each of these events, in arbitrary order{
        if (this is an update event){
            Update the value specified
            Evaluate gates on the fanout of this value and Schedule update
            events for gate outputs that change
            Schedule evaluation events for behaviors waiting for this value
        }
        else{ // 这是一个计算事件
            Evaluate the model
            Schedule any update events resulting from the evaluation
        }
    }
}
```

图 5.6 包含非阻塞事件的事件驱动调度程序

#### 例 5.10 非阻塞赋值

```

module fsm (cS1, cS0, in, clock);
    output      cS1, cS0;
    input       in, clock;
    reg         cS1, cS0;
    always @(posedge clock) begin
        cS1 <= in & cS0;
        cS0 <= in | cS1;
    end
endmodule

```

事件列表可以被想像成独立的三层结构,如图 5.7 所示。在事件列表中的任意给定时刻  $\tau$  都存在三个独立层次:规则事件、非阻塞事件和监视事件。调度算法把规则事件从列表中删除并执行它们,这可能导致其他一些事件被调度到本时刻或其他时刻。 $\tau$  时刻的规则事件被放入规则事件部分的列表中,这些事件都将在下一个模拟周期被删除。其他事件会被调度到它们各自的部分或者以后某个时刻的事件列表中。

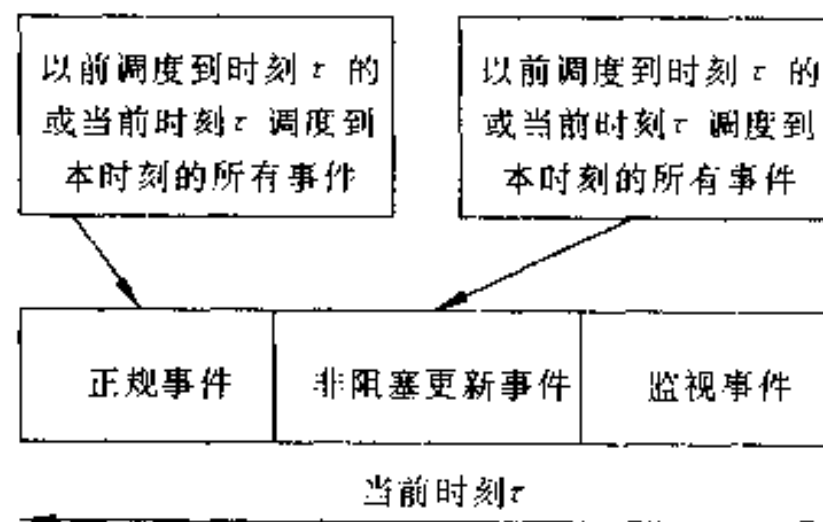


图 5.7 当前时刻的层次化事件列表

当我们到达下一个具有更多规则事件的模拟周期时,它们的处理与前述方式相同。当前时刻不再有规则事件时,非阻塞层的事件被移动到规则事件层来执行。这样反而可能产生其他规则事件和非阻塞事件,它们被调度到它们各自的部分。事件调度算法继续重复执行当前时刻的所有规则事件。然后执行当前时刻的非阻塞事件,直到当前时刻没有事件(规则的或非阻塞的)存在为止。这时候,调度器就处理监视事件。当监视语句的输入有变化时,把这些事件插入监视事件层。这些就是在时间推进前应处理的最后一些事件。它们不会产生其他事件。

综上所述,以下给出一个列表来说明模拟调度器是怎样处理不同的过程赋值语句的:

- “ $a = b;$ ”  $b$  被计算并且立刻用来更新  $a$ 。注意:在行为进程中的下一条语句使用  $a$  作为源时引用的将是新值。如果  $a$  是某个进程的输出,那么  $a$  的扇出表上的元件被作为规则计算事件调度到当前时刻。

- “ $a \Leftarrow b;$ ”  $b$  被计算并且  $a$  的非阻塞更新事件被调度到当前时刻。进程继续执行。非阻塞更新事件被执行之前,  $a$  的新值对其他的元件来说是不可见的(甚至当前正执行的行为进程)。
- “ $a = \#0 b;$ ”  $b$  被计算并且一个  $a$  的更新事件被作为规则事件调度到当前时刻。当前进程被阻塞,直到下一个模拟周期,  $a$  将被更新,进程继续执行。
- “ $a \Leftarrow \#0 b;$ ”  $b$  被计算并且一个非阻塞更新事件被调度到当前时刻。当前进程继续执行。直到当前时刻的所有规则事件执行完之后,更新事件才被执行。这与“ $a \Leftarrow b;$ ”相同。
- “ $a = \#4 b;$ ” 与“ $a = \#0 b;$ ”相同,只是这个更新事件和进程的继续执行被调度到 4 个单位时间之后。
- “ $a \Leftarrow \#4 b;$ ” 与“ $a \Leftarrow \#0 b;$ ”相同,只是  $a$  将在 4 个单位时间之后被更新(用非阻塞更新事件)。
- “ $\#3 a = b;$ ” 在动作“ $a = b;$ ”执行前先等待 3 个单位时间。赋给  $a$  的值将是 3 个单位时间之后  $b$  的值。
- “ $\#3 a \Leftarrow b;$ ” 在动作“ $a \Leftarrow b;$ ”执行前先等待 3 个单位时间。赋给  $a$  的值将是 3 个单位时间之后  $b$  的值。

可以看到上述情况中赋给  $a$  的值都相同。(当然,在后两个示例中,  $b$  的值在 3 个单位时间之后可能会改变。但对这两个示例而言,赋予  $a$  的值是相同的。)差别在于更新被调度到事件列表的哪一个部分,新值在下一条语句中是否有效,当前进程是否因为带有  $\#$  号被阻塞。

#### 5.4.4 非阻塞赋值的举例分析

如前一节所述,非阻塞赋值允许我们把事件调度到一个时间步的最后,无论是当前的时间步还是将来的时间步。而且,它们允许进程继续执行。与阻塞语句一样,事件控制和 repeat 结构可以在赋值语句中说明。非阻塞赋值的一般形式如下:

```

non-blocking assignment
    ::= reg lvalue <= [delay_or_event_control] expression

delay_or_event_control
    ::= delay_control
    | event_control
    | repeat ( expression ) event_control

delay_control
    ::= # delay_value
  
```

```

|      # ( mintypmax_ expression )

event.. control
    ::= @ event.. identifier
    |   @ ( event.. expression )

event expression
    ::= expression
    |   event.. identifier
    |   posedge_ expression
    |   negedge_ expression
    |   event_ expression or event expression

```

我们已经举例说明了作为可选项的延迟控制,这一节将讨论事件控制和 repeat 结构。

非阻塞赋值的一个特征是它在调度一个赋值时并不阻塞当前进程的进一步执行。考虑例 5.11 所示的 NAND 门的行为模型,它把门的惯性延迟改为零。任何 lisa 或者 michael 上的变化将导致一个 doneIt 的更新事件被调度到 pDelay 个单位时间之后。非阻塞赋值在这里是必要的,因为它使行为模块对输入保持敏感。一个单位时间之后的变化会引起 doneIt 上的另一个更新事件。如果使用阻塞赋值的话,行为模型将延迟,而且一个单位时间之后的输入变化在延迟结束以前都将被忽略。

#### 例 5.11 非阻塞赋值

```

module inertialNand (doneIt, lisa, michael);
    output      doneIt;
    input       lisa,
               michael;
    reg         doneIt;
    parameter   pDelay = 5;

    always
        @(lisa or michael)
            doneIt <= #pDelay ~(lisa & michael);
endmodule

```

图 5.8 画出了例 5.11 中 NAND 门的输出波形( $iDelay = 0$ )与一个 NAND 门例示的输出波形的比较。仅当 NAND 门例示应有的输出值已经保持了传输所需的时间长度之后,这个输出值才会传输出来。否则,它们不会对输入作出反应。因此输出中没有看到输入的脉冲并且输出更新事件被取消了两次(如“\*”处所示)。在惯性延迟为零时,输入脉冲在传输时间(pDelay)之后才出现。注意:在右部,两个不同输入的改变可能产生一个输

入“脉冲”。因为零惯性延迟,这个脉冲可以在图右部的输出上看到。

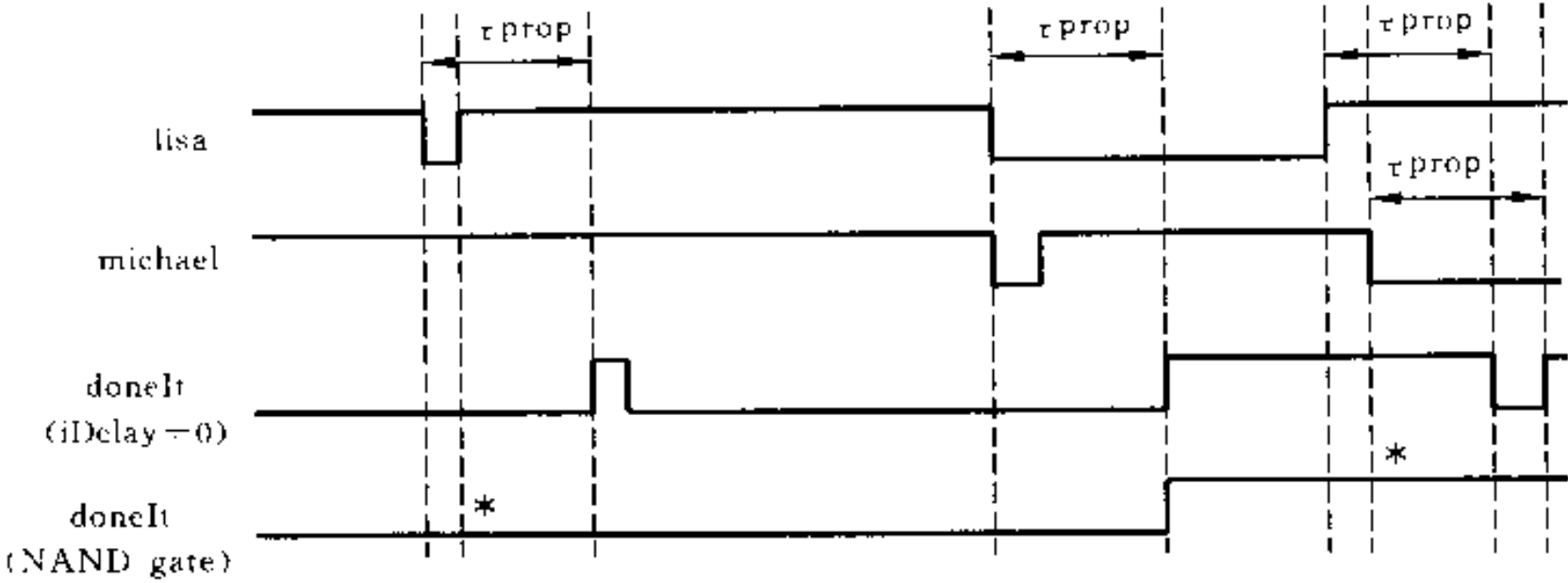


图 5.8 不同惯性延迟的波形

让我们分析例 5.12 所示的流水线乘法器的行为模型。这个乘法器的计算时间为四个时钟周期,而且它在每个时钟周期都可以接收新的输入。go 的上升沿是开始执行乘法运算的信号。此时,输入相乘,而且将 product 的更新调度到时钟的第四个上升沿。由于这是一个非阻塞赋值语句,已计算出来的 product 被存入事件列表,然后 always 语句将等待表示下一个乘法运算开始的信号 go 的到来,两次计算可能叠加在一起执行。在这种情况下,go 信号必须与 clock 信号同步,因为我们不能在一个 clock 的变化沿开始执行多个乘法运算。然而,我们可以在每一个时钟周期都开始执行一个乘法运算。本例进一步说明了事件列表可以用来存储具有相同名字的、由同一个赋值语句产生的多个事件。

例 5.12 流水线乘法器

```
module pipeMult (product, mPlier, mCand, go, clock);
    input          go, clock;
    input  [9:0]    mPlier, mCand;
    output [19:0]   product;
    reg  [19:0]     product;

    always
        @(posedge go)
            product <= repeat (4) @(posedge clock) mPlier * mCand;
endmodule
```

这里给出了一个门级时序模型和过程时序模型的有趣比较。如果一个使用了 Verilog 门级时序模型的元件的输出产生了一个更新事件,那么,已经调度到事件列表的更新事件将被删除,新的更新事件将被调度。这不是使用过程时序模型的情况。正如我们在本例中看到的一样,product 的多个更新事件将在不改变任何已调度的更新事件的情

况下被调度。

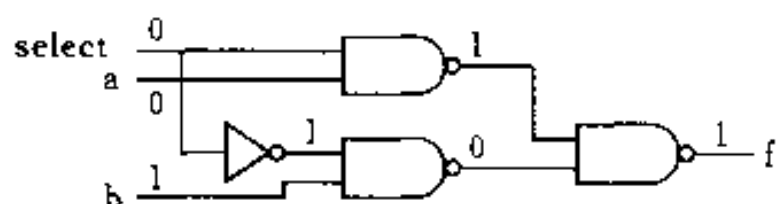
参考:内部赋值循环 3.7

## 5.5 小节

按照模拟时间的推进方式的不同,我们将模拟模型分成两大类进行了介绍,并讲述了处理这些模型的模拟调度算法。覆盖了具体的时序问题,包括语言的非确定性、阻塞与非阻塞过程赋值的比较。

## 5.6 练习题

5.1 下面给出了一个电路和两个 Verilog 模型。一个是结构模型,另一个是行为模型。注意:这两个模型并不是等同的。电路图表示逻辑级。假设事件列表中除了下面给出事件以外没有其他事件。



```
module sMux (f, a, b, select);
    input    a, b, select;
    output   f;

    nand     #8
        (f, aSelect, bSelect),
        (aSelect, select, a),
        (bSelect, notSelect, b);
    not
        (notSelect, select);
endmodule
```

```
module bMux (f, a, b, select),
    input    a, b, select;
    output   f;
    reg      f;

    always
        @select
            #8 f = (select) ? a : b;
endmodule
```

- A. 单独模拟 sMux 模块,写出将最终出现在事件表中的所有事件。事件列表中的初始事件是更新事件:“时刻 35 的  $b = 0$ ”事件。以“时刻 50 的  $select = 1$ ”事件作为初始更新事件来单独模拟它。
- B. 单独模拟 bMux 模块,写出将最终出现在事件表中的所有事件。事件列表中的初始事件是更新事件:“时刻 35 的  $b = 0$ ”事件。以“时刻 50 的  $select = 1$ ”事件作为初始更新事件来单独模拟它。

C. 正如上面提到的那样,这两个模型是不一样的。简述两个模块为什么不同。试改写 bMux 模型的 always 语句,使其在功能和时序上等同于 sMux。考虑这些方面:功能、输入敏感元和惯性延迟的时序。对你的改动加以解释。

D. 作为一个时间的函数,什么是 bMux 中的 always 进程语句的输入敏感表?

5.2 将模型 nbSchedule 模拟 25 个单位时间,根据模拟情况填写下面的表格。表格的每一行代表一个模拟周期。你不必提交事件列表和它们随时间的变化过程。然而,保留这个事件列表可能会帮助你逐行填写表格。

当前时间	模拟周期标号	从事件列表中删除的事件 (标明是阻塞或非阻塞事件)	新事件 (存储和计算事件)
...	...	...	...

```

module nbSchedule (q2);
    wire    q1;
    output  q2;
    reg     c, a;

    xor     (d, a, q1),
            (clk, 1'b1, c);
            // holy doodoo, Batman, a gated clock!
    dff     s1 (q1, d, clk),
            s2 (q2, q1, clk);
    initial begin
        c = 1;
        a = 0;
        #8 a = 1;
    end

    always
        #20 c = ~c;
endmodule

```

```

module dff (q, d, c);
    input  d, c;
    output q;
    reg    q;

    initial q = 0;

    always

```

```

        @(posedge c) q <= d;
    endmodule

```

5.3 从时刻 0 到时刻 40 执行以下的描述:

```

module beenThere;
    reg    [15:0]  q;
    wire                h;
    wire    [15:0]  addit;

    doneThat  dT (q, h, addit);
    initial q = 20;

    always begin
        @(posedge h);
        if (addit == 1)
            q = q + 5;
        else q = q - 3;
    end
endmodule

```

```

module doneThat(que, f, add);
    input  [15:0]  que;
    output                f;
    reg                f;
    output [15:0]  add;
    reg    [15:0]  add;
    always
        #10 f = ~ f;

    initial begin
        f = 0;
        add = 0;
        #14 add = que + 1;
        #14 add = 0;
    end
endmodule

```

按时间顺序给出寄存器发生变化的时间,发生变化的寄存器名称和被赋的新值(根据需要添加新行)。

at time=\_\_\_\_\_ ; \_\_\_\_\_ = \_\_\_\_\_

5.4 试写一个例 5.12 的测试模块并执行它。时钟周期定为 100ns。在 mPlier 和 mCand 中存入几个测试向量(乘法运算的操作数),并且 go 的脉冲宽度要大于 10ns。(什么是测试模块? 参考第 1 章。)

A. 确认与 pipeMult 一起工作的测试模块是否产生正确答案。

B. 跟踪事件列表中的更新事件和计算事件。

5.5 这里给出了两个模块(interleave 和 huh)的框架。它们可能由于与其他行为进程的交叉执行而产生非确定性。描述何处会遇到这个问题。解释如何能改正这个问题。

```

module interleave;
    reg    [7:0]  a;
    huh      h();
    ...
endmodule

```



```

module huh;
    reg      [7:0]    b,c,q,r;
    always begin
        ...
        a = b + c;
        q = a + r;
        ...
    end
endmodule

module ouch (select, muxOut, a, b);
    input    a,b;
    input    select;
    output   muxOut;
    reg      muxOut;
    always begin
        @select
            muxOut = (a & select)|(b & notSelect);

        not
            (notSelect,select);
    endmodule

```

- 5.6 以前曾经讲到,一些非确定性是与模拟器相关的,通过模拟这个电路,解释怎样获得不同的结果,并给出解决本问题的两种不同的方法。
- 5.7 假设所有寄存器都是一位的,且初值为 x,试比较下面的两种情况。在什么时间寄存器的值会改变?

<pre> initial begin     q = #15 1;     r = #25 0;     s = #13 1; end </pre>	<pre> initial begin     q &lt;= #15 1;     r &lt;= #25 0;     s &lt;= #13 1; end </pre>
---	---

- 5.8 改写例 5.12 的 pipeMult 模块,使其每两个时钟周期获取一次新值。也就是说,流水线延迟仍是 4 个时钟周期,但初始化的速率为 2 个时钟周期初始化一次。测试这个新模块。
- 5.9 给出一个不使用临时寄存器的交换两个寄存器值的行为描述。新值要在两个上升沿之后出现。完成下面的模块。

```

module swapIt(dolt);
    reg      [15:0]    george,georgette;
    input                                dolt;

    always
        @(posedge dolt)
            // do it
endmodule

```

- 5.10 使用非阻塞赋值语句重写例 5.4 中的 twoPhiLatch。
- 5.11 写出一个简单元件的模型,它要求在模型中使用“<=”而不是“=”。例 5.12 是一个范例,试给出另一个示例。
- 5.12 一个学生曾经问我,他们是否能够(could)在他们的有限状态机描述中使用阻塞赋值语句而不用非阻塞赋值语句。如果他们这么做了,我会狠狠地批评他们。请记住“may”和“can”的区别。
  - A. 举出一个能够(can)这样用而不出现问题的示例,并解释原因。它需要什么前提?
  - B. 简单解释他们不可以(may)这样做的原因。

## 第6章 逻辑综合

到目前为止,我们对 Verilog 语言的分析一直是针对逻辑硬件进行建模和模拟的。我们已经给出用于描述电路的复杂功能和时序的语言结构。使用这种方法,我们可以根据已经布局 and 布线的电路时序参数来对一个设计进行模拟,并给出令人信服的模拟结果。本章中,我们要进一步了解这种语言的另一方面:综合。当使用该语言作为综合的输入规范时,应该考虑的是在利用 CAD 综合工具设计系统最终门级结构的同时应注重系统运作的正确性。本章内容与前面各章节相辅相成。但是,在编写用于模拟和综合的描述时必须小心谨慎。

### 6.1 综合概述

当今使用的占主导地位的综合技术是逻辑综合。在设计寄存器传输级对系统进行描述,并且通过利用逻辑综合工具获得系统的门级实现。综合工具能优化带有多种约束的设计,这些约束包括时序和(或)面积。综合工具使用工艺库文件来指明设计中用到的各种元件。本章讨论的内容是为逻辑综合工具书写 Verilog 描述。

#### 6.1.1 寄存器传输级系统

寄存器传输级描述包括不同的特征;一部分可以是单纯的组合描述,而其他可能指明时序元件,例如锁存器和触发器。还可以是有限状态机描述,用来说明一个状态转换图。

逻辑综合工具使用两个主要阶段来对寄存器传输级设计进行编译:第一个阶段是一个工艺无关的阶段,设计的读入和处理都不考虑最后的实现工艺。在这个阶段进行组合逻辑的主要化简工作。第二阶段是工艺映射,将设计转换成与元件库中的元件相匹配的形式。如果元件库中只包含两个输入端口的门,则将设计进行转换,使得每个逻辑功能都能用库中的元件来实现。事实上,综合工具能够将一种门级描述转换为另一种门级描述,提供应用一个新的工艺库时对电路进行重新设计的能力。

逻辑综合 CAD 工具吸引人的地方在于它辅助进行很复杂的设计(别忘了,当卡诺图有五或六个以上的变量时,你的逻辑设计教授有没有教过你该怎么做!)。这些工具以大型的组合设计和不同的工艺库为目标,提供时间和面积的折衷实现。并且,它们能够保证初始设计规范 and 实现结果的功能等价性。在设计层次相当复杂的情况下,逻辑综合设计工具能在很多常见设计的情况下提高设计者的效率。

为了提高设计效率,我们必须用一定的方式来规范我们的设计,以便模拟我们的设计,判断它是否正确,然后进行综合。本书的前述章节主要讲述该语言的语义以及怎样对复杂的时序和行为进行建模。本章讨论作为逻辑综合工具输入的寄存器传输级系统的描述方法。

### 6.1.2 限制声明

本章的第一部分定义了用于逻辑综合的可综合描述是什么。有些行为我们能够描述出来,但普通的逻辑综合工具却不能对之进行设计或设计得非常糟糕。因为综合技术还不成熟,将一个任意的行为映射到一组库元件是很复杂的,不允许将任意行为描述作为逻辑综合工具的输入。因此只有该语言的一个子集可以用于逻辑综合,并且对使用该子集书写描述的形式也有约束。本章的第一部分描述了当前在逻辑综合规范中常见的子集和约束。随着逻辑综合技术的成熟,可用结构的集合将可能扩展,描述风格的约束将可能减少,二者在最近的几年内都已经有所发展。

有几家 CAD 工具厂商提供逻辑综合工具。它们的语言子集和约束是不同的。我们对逻辑综合的讨论是根据使用 Synopsys 公司和 Synplicity 公司产品的经验来进行的。如果你使用的是其他公司的产品,你可能有不同的经验。请仔细阅读综合工具使用手册。

## 6.2 使用门和持续赋值的组合逻辑

使用门基元和持续赋值语句来说明用于逻辑综合的逻辑函数相当直观。例 6.1 和 6.2 给出了这种类型的两个可综合描述。两个示例实现的组合逻辑函数是相同的;标准的积之和的说明是:

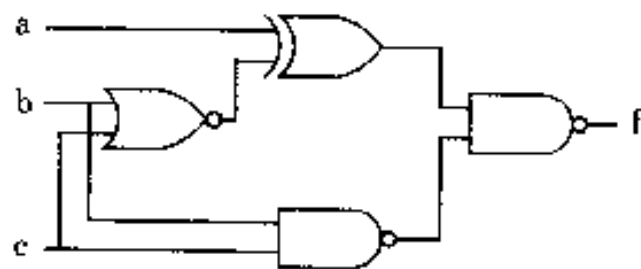
$$f(a,b,c) = \sum m(a,b,c) = \sum m(1,2,3,4,7)$$

本质上,逻辑综合工具读入该说明的逻辑功能,并试着优化与设计约束和库元件有关的最终的门级设计。尽管例 6.1 说明了一个门级设计,但逻辑综合工具可以使用不同的门基元来实现相同的功能,当然也可以带有某些约束。本例给出了一个形式不同但是功能相同的门级设计。这里,工艺库只包含带有两个输入端的门;综合工具把设计转换成如例 6.1 右图所示的实现。工艺库和性能约束的不同也可能出现其他设计形式。

**例 6.1** 一个描述及其综合后的实现

```
module synGate (f, a, b, c);
    output      f;
    input       a, b, c;

    and         A(a1, a, b, c);
```



```

and      B(a2, a, ~b, ~c);
and      C(a3, ~a, 01);
or       D(01, b, c);
or       E(f, a1, a2, a3);
endmodule

```

上例不包含延迟( # )信息, 是否包含延迟信息是 Verilog 模拟描述和综合描述之间的主要区别之一。在模拟描述中, 我们通常向模拟器提供详细的时序信息以便帮助设计人员进行时序验证。逻辑综合工具将忽略这些时序说明, 只用到描述中提供的功能说明。由于逻辑综合工具忽略了时序说明, 因此在描述中加入时序说明有助于区分逻辑综合工具的输入设计的模拟和它的输出结果的模拟。

考虑例 6.1 中的门示例 A, 如果它被说明为:

```
and #5 A(a1,a,b,c);
```

那么该描述的模拟结果显示出门输入到门输出之间的变化有 5 个单位时间的延迟。例 6.1 中所示的实现并没有与 A 相对应的门。因此, 那种实现方式的模拟时序是不同的。逻辑综合并不一定要符合这种时序说明。但是综合工具提供说明时序要求的手段, 如时间周期。这种工具尽力使逻辑设计的所有建立时间在时钟周期内都能得到满足。在本例中, 最重要的是根据触发器的输入值及时地产生布尔函数 f 的值。

## 例 6.2 使用持续赋值的一个可综合描述

```

module synAssign (f, a, b, c);
    output      f;
    input       a, b, c;

    assign      f = (a & b & c) | (a & ~b & ~c) | (~a & (b | c));
endmodule

```

如例 6.2 中所示, 持续赋值语句的使用类似于布尔代数逻辑的表达, 区别是 Verilog 在规范中使用的操作符比布尔代数多。在本例中, 使用与例 6.1 相同的积之和功能, 但是赋值语句将积 1、2 和 3 合并成最后一个乘积项。值得注意的是, 持续赋值语句可以调用一个包含过程赋值语句的函数。描述组合逻辑的过程赋值语句的用法将在 6.3 节讨论; 因此我们这里只讨论不包括函数调用的持续赋值。

持续赋值语句常用于描述数据通道元件。与有限状态机中对后继状态和输出逻辑的逻辑说明相比较而言, 这些模块趋向于只有一行说明。例 6.3 中加法器和多路选择器都用持续赋值语句进行描述。模块 AddWithAssign 用被加字的宽度作为参数, 包括进位输入(Cin)端口和进位输出(carry)端口。请注意赋值语句右侧产生的和生成的结果比输出 sum 更大。连接操作符说明最高位(进位输出)将驱动 carry 的输出, 其他位将驱动 sum

的输出。使用条件操作符描述多路选择器。

### 例 6.3 使用持续赋值描述的数据通道元件

```
module addWithAssign (carry, sum, A, B, Cin);
    parameter WIDTH = 4;
    output      [WIDTH:0]    sum;
    input       [WIDTH:0]    A, B;
    input                               Cin;
    output                               carry;

    assign      {carry, sum} = A + B + Cin;
endmodule

module muxWithAssign (out, A, B, sel);
    output      out;
    input       A, B, sel;

    assign      out = (sel) ? A : B;
endmodule
```

对于可能用到的运算符和使用未知量(x)的方式有一些限制。在可综合的描述中可以使用未知量,但仅使用在某些情况下。下面的程序不可综合,因为它把一个数值和一个未知量进行比较。

```
assign y = (a == 1'bx) ? c : 1;
```

以这种方式使用的未知量在模拟器中是一个数据类型;它用于确定 a 的值是否已经变为未知量。但我们并不建立数字硬件去与未知量进行比较,因此这种结构是不可综合的。然而接下来的程序在非比较方式中使用未知量是允许的。

```
assign y = (a == b) ? 1'bx : c;
```

在这个示例中,我们要说明一种与逻辑综合器无关的情况。也就是说,当 a 等于 b 时,我们并不关心给 y 赋什么值。如果它们不相等,那么 y 被赋值为 c。根据这条赋值语句进行综合的硬件中,y 可能为 0 也可能为 1(毕竟,实际的硬件中未知量并不存在)。在这种情况下使用的无关性说明允许综合器在优化逻辑电路时具有更多的自由度。这个描述的最佳实现就是  $y=c$ 。

参考:赋值 4.4; 门基元 4.2

## 6.3 用来说明组合逻辑的过程语句

除了使用持续赋值语句和基本的门例示来说明组合逻辑外,也可以使用过程语句。

过程语句可以在 always 语句中说明,也可在 always 语句调用的任务中说明,还可以在 always 语句或持续赋值语句调用的函数中说明。尽管使用过程语句的描述看起来是“顺序的”,组合逻辑还是可以用它们来说明。1.2 节介绍了这种说明组合逻辑的方法。本节涵盖了更多的细节内容。

### 6.3.1 基础

组合逻辑过程描述的基本形式如例 6.4 所示。它包括一个带有事件语句的 always 语句,该语句包括组合函数所有的输入变量。本例给出了一个用过程语句描述的多路选择器。在这个示例中,输入 a 控制着将输入 b 还是输入 c 传送给输出 f。尽管 f 被定义为一个寄存器,但是综合工具把该模块看作是一个组合逻辑的说明。

例 6.4 用过程语句描述的组合逻辑

```
module synCombinationalAlways (f, a, b, c);  
    output      f;  
    input       a, b, c;  
    reg         f;  
  
    always @ (a or b or c)  
        if (a == 1)  
            f = b;  
        else  
            f = c;  
  
endmodule
```

下面的几个定义将说明这些描述的读和写方式的一些规则。我们定义 always 块的输入集为所有 always 块中过程语句右侧用到的寄存器、线和输入的集合。在例 6.4 中,输入集合包括 a、b 和 c。进一步地,我们将 always 块的敏感列表定义为出现在事件语句 (“@”) 中的名字的列表。在本例中,敏感表中包含了 a、b 和 c。当使用过程语句来描述组合逻辑时,always 块输入集合中的每一个元素必须出现在 always 语句的敏感列表中。这一规律符合组合逻辑的定义——每一个输入值的变化可以立即影响到输出结果。如果输入集合中的一个元素不在敏感列表中,那么它就不能立即影响输出结果。它必须一直等待其他输入发生改变;实际上组合电路不是这样。

进一步考虑例 6.4,我们注意到组合逻辑的输出 f 在 always 块的每个分支被赋值。在执行 always 循环时控制路径被定义为一个执行操作序列。因为可以使用条件语句(例如 case 语句和 if 语句),在 always 块中可能有许多不同的控制路径。组合函数的输出必须在每个可能的控制路径中被赋值。因此,对每种可预料到的输入变化,组合逻辑的输出将被重新计算;这是组合逻辑的一个特点。

上述示例和讨论实质上概述了使用过程语句对组合硬件进行说明的一些规则：敏感列表必须是输入集合，组合输出必须在每个控制路径中被赋值。在后面我们将见到的另一个规则是敏感列表不可以包含任何边沿敏感说明——这些专门用来说明触发器。

参考：always 2.1；敏感列表 5.1；@3.2；边沿说明 3.2；输入集 7.2.1

### 6.3.2 复杂形式——推断出的锁存器

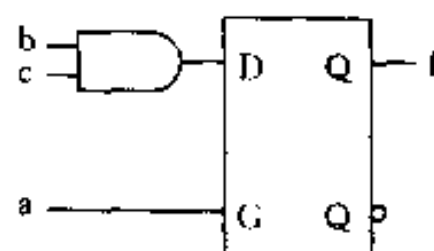
如果存在一条不对输出进行赋值的控制路径，那么前一次输出值需要被存储起来。这不是组合硬件的特点。然而这是时序系统的特征，时序系统中前一个状态被保存在锁存器中，当输入符合这个控制路径时，把前一状态选通输出。逻辑综合工具能识别这种情况，并推断电路需要锁存器。假设我们试图描述组合硬件，我们想要保证设计中不加入推断出的锁存器(inferred latch)。在每个控制路径中对组合逻辑输出进行赋值就可保证这一点。

例 6.5 是一个推断出锁存器的示例。如果我们跟踪本例中的控制路径，可以发现如果 a 等于 1，那么 f 被赋值为 b&c。但是，如果 a 等于 0，那么 f 在 always 块执行中不被赋值。因此，存在一条 f 没有被赋值的控制路径。在这种情况下，推断出一个锁存器，并且本例右侧所示的电路被综合出来。锁存器实际上是一个选通锁存器——一个电平敏感的器件，当锁存器的选通输入(与 a 相连的 G)为 1 时，它能传递输入端 D 的值，并且当输入为 0 时，它保持原值。

例 6.6 例示了使用 case 语句来说明真值表中的组合函数的方法。(本例说明了与例 6.1 和 6.2 相同的逻辑函数。)本例例示并遵循使用过程语句来说明组合逻辑的规则：always 输入集的所有成员都包含在 always 的敏感列表中，组合逻辑的输出在每一个控制路径中都被赋值，并且敏感列表中没有边沿说明。

#### 例 6.5 一个推断出来的锁存器

```
module synInferredLatch (f, a, b, c);  
    output      f;  
    input       a, b, c;  
    reg         f;  
  
    always @(a or b or c)  
        if (a == 1)  
            f = b & c;  
  
endmodule
```



#### 例 6.6 case 语句说明组合逻辑

```
module synCase (f, a, b, c);
```



```

output      f;
input       a, b, c;
reg         f;

always @(a or b or c)
    case ({a, b, c})
        3'b000:      f = 1'b0;
        3'b001:      f = 1'b1;
        3'b010:      f = 1'b1;
        3'b011:      f = 1'b1;
        3'b100:      f = 1'b1;
        3'b101:      f = 1'b0;
        3'b110:      f = 1'b0;
        3'b111:      f = 1'b1;
    endcase
endmodule

```

当然,在使用 case 语句时,有可能列举的情况不完全。如果在 case 的控制表达式中有 n 位,那么就可能有  $2^n$  个控制路径。如果我们没有列出所有的路径,那么就可能存在一条没有给输出端赋值的路径,由此可推断出锁存器。可以使用默认的 case 项来定义余下的未说明的 case 项。因此例 6.6 也可以写成例 6.7 的形式。这里我们使用独立的 case 项显式地列出所有的函数值为 0 的情况。如果输入不与其中的任何一项匹配,那么默认情况下 f 被赋值为 1。

#### 例 6.7 使用默认项来完整地说明 case 语句

```

module synCaseWithDefault (f, a, b, c);
    output      f;
    input       a, b, c;
    reg         f;

    always @(a or b or c)
        case ({a, b, c})
            3'b000:      f = 1'b0;
            3'b101:      f = 1'b0;
            3'b110:      f = 1'b0;
            default:      f = 1'b1;
        endcase
endmodule

```

说明了(显式或默认地)所有控制路径的 case 语句被称为完备的 case 语句。一个完

备的 case 语句通过对每个可能的控制路径的输出进行说明来帮助避免产生推断的锁存器。一些综合工具提供编译指令,允许你在即使所有的 case 项都没有说明的情况下也认为 case 语句是完备的。在这种情况下,不会推断出锁存器,并且出于逻辑优化的考虑,未进行说明的 case 项的输出值被认为是未定义的。

在 case 项的说明中当出现重叠时产生另外一种情况。case 项之间没有重叠的 case 语句是并行 case 语句。也就是,如果控制表达式的给定值使得多于一个以上的 case 项为真,那么该 case 语句不是并行的。如果 case 是并行的(也是完备的),它被认为是积之和的说明,这有助于逻辑优化。如果 case 不是并行的,意味着有重叠,那么逻辑函数将复杂得多。

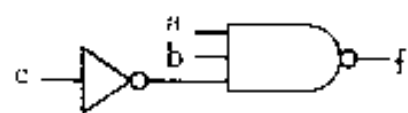
参考: case 2.4

### 6.3.3 说明无关项

逻辑综合工具充分利用无关项来优化逻辑电路。例 6.8 说明了包含一个无关项的逻辑函数。通常在默认的 case 语句中指定这些无关项。如例中所示,给输出赋值 x,在本例中解释为输入 case 3'b000 和 3'b101 是无关项。该函数的优化实现如示例右侧所示;只有函数中唯一的 0(输入 case 3'b110)被实现反向。一般说来,指定一个输入为 x 则允许综合工具把它作为一个逻辑无关项说明。

#### 例 6.8 带有无关项的逻辑函数

```
module synCaseWithDC (f, a, b, c);
    output      f;
    input       a, b, c;
    reg         f;
    always @(a or b or c)
        case ({a, b, c})
            3'b001:      f = 1'b1;
            3'b010:      f = 1'b1;
            3'b011:      f = 1'b1;
            3'b100:      f = 1'b1;
            3'b110:      f = 1'b0;
            3'b111:      f = 1'b1;
            default:      f = 1'bx;
        endcase
endmodule
```



casex 语句可以在描述中使用逻辑无关项。请记住 casex 语句允许在控制表达式或

case 项表达式中使用 x、z 或？。在综合中，x、z 或？只能用在 casex 表达式中，不能用在控制表达式中。请考虑例 6.9 中所示的模块。

### 例 6.9 使用无关项来说明逻辑函数

```
module synUsingDC (f, a, b);
    output      f;
    input       a, b;
    reg         f;
    always @(a or b)
        casex ({a, b})
            2'b0?:      f = 1;
            2'b10:      f = 0;
            2'b11:      f = 1;
        endcase
endmodule
```

		a	0	1
b				
0			1	0
1			1	1

第一个 case 项说明如果 a 为 0，那么输出 f 为 1。语句中？的使用表示在这种情况下 b 的值无关紧要。请注意这个函数不包含无关项。而且，我们使用？这种方式来说明函数显得更加简单和紧凑。因为第一个 case 项包括了四种可能情况中的两种，所以它与其他两个 case 项一起构成了一个完整的 casex。在本例中，用？来指明逻辑无关项。也可以使用 x 或 z。

也可以使用 casez 语句来说明逻辑无关项。在这种情况下，只有 z 或？用于说明 case 项表达式中的无关项。

在描述中使用无关项时应小心谨慎，因为它们引起模拟和综合的差异。在模拟器中，x 是四个已定义的逻辑值中的一个。与 x 进行比较在模拟中是有意义的，而在综合中无意义。为了减少模拟和综合之间的差别，可综合的描述不与 x 或 z 作比较。此外，x 仅赋给组合电路的输出，这已在例 6.8 中说明，在后面的例 6.15 中也将说明。

参考：casex 和 casez 2.4

### 6.3.4 过程循环结构

以上的示例可能暗示，说明逻辑函数的唯一方式是通过 if 语句和 case 语句。循环语句 for、while、repeat 和 forever 是有特殊意义的。在这些语句中，只有 repeat 语句不允许出现在可综合描述中。

for 循环允许重复计算，如例 6.10 所示，该例是例 4.4 的改写。在这个示例中，循环的每次迭代都指明一个由循环变量 i 来指定下标的逻辑单元。这里，有 8 个异或门连接在输入和输出之间。由于这是一个组合逻辑的说明，所以 i 在最终实现时不作为一个寄

存器。

#### 例 6.10 用作异或门数组的说明

```
module synXor8 (xout, xin1, xin2);  
    output      [1:8]  xout;  
    input       [1:8]  xin1, xin2;  
    reg         [1:8]  xout, i;  
  
    always @(xin1 or xin2)  
        for (i = 1; i <= 8; i = i + 1)  
            xout[i] = xin1[i] ^ xin2[i];  
endmodule
```

该例阐明了使用 for 语句描述逻辑的几个要点。for 语句有一个下标 i, 其取值可以从低到高递增, 也可以从高到低递减。循环的结束比较可以是 <、>、<= 或 >=, 而步距不一定为 1。下面是计数递减的一般形式:

```
for (i = highLimit; i >= lowLimit; i = i - step);  
...
```

for 循环结构化程度很高, 清楚地指明了步距变量和范围。while 循环和 forever 循环没有如此清楚的结构, 在使用时需要指明范围。while 循环或 forever 循环中的每一条控制路径必须指定时钟边沿。因此, while 循环和 forever 循环不能用于组合逻辑。

参考: 不允许的结构 6.7

## 6.4 时序元件的推断

时序元件是锁存器和触发器, 它们构成了寄存器传输级系统的存储单元。尽管它们是数字系统的基本组成成分, 很难向综合工具描述这些基本组成成分。主要原因在于它们的行为相当复杂。就像我们将要看到的那样, 这些元件中的一部分(尤其是触发器)的描述形式几乎已经制定好了, 所以综合工具知道用哪个库元件来映射行为。这些预先规定好的描述形式使综合工具不需要把任意的行为说明匹配到每个库元件的行为说明上去。

### 6.4.1 锁存器的推断

锁存器是电平敏感的存储器件。它们的行为一般由系统时钟控制, 系统时钟连接到选通输入 G 上。当选通输入有效时(高电平或低电平), 锁存器的输出 Q 随着输入 D 来变化——这是 D 的组合函数。当选通输入无效时, Q 输出 D 前一次的输入值。有时这些

器件具有异步的置位或复位。如我们在 6.3.2 节中所看到的那样,锁存器没有被显式地说明。相反由描述的书写方式来推断。我们称锁存器被推断。例 6.5 是一个推断锁存器的示例。

可以使用 always 语句作为基本语句来推断锁存器。在 always 语句内,我们把控制路径定义为执行 always 循环时的一个操作序列。因为可以使用条件语句(例如 case 和 if),在 always 块中可能有许多不同的控制路径。为了使用过程语句来生成一个组合电路,组合函数的输出必须在每一个不同的控制路径中被赋值。因此,对于每一个意料中的输入的变化,将重新计算组合输出。

要推断出一个锁存器,在 always 语句中必须存在两种情况:至少存在一条不对输出进行赋值的控制路径,敏感列表不允许包含任何边沿敏感の説明。前一种情况要求保留前一次的输出值。第二种情况要求使用电平敏感的锁存器(与边沿敏感触发器相对立)。这表明时序元件需要一种存储器,在这种元件中前一个状态保存在锁存器中,并且当输入信号选定控制路径时被选通到输出端口。逻辑综合工具能分辨出这种情况,并推断出电路中需要锁存器。假设我们正在描述一个时序元件,输出变量至少在一个路径中未被赋值,这将导致推断出一个锁存器。

例 6.11 描述了带 reset 输入的锁存器。尽管我们已经指定了输出 Q 是一个寄存器,但是并不是这一点导致推断出锁存器。为了说明怎样推断出锁存器,请注意在 always 语句的控制流中,并没有规定 g 和 reset 的所有可能的输入组合。该规范表明如果 g、d 或 reset 中的任一个发生变化,则执行 always 循环。如果 reset 为 0,那么 Q 被置为 0。如果不是这种情况,那么如果 g 为 1,则 Q 被置为输入 d。但是,因为没有指明在 reset 为 1 和 g 为 0 时该如何输出,所以,这种情况下,锁存器需要保存 Q 的前一个值。事实上这就是带复位的电平敏感锁存器的行为。也可以使用 case 语句来推断出锁存器的行为。

#### 例 6.11 带有复位的锁存器

```
module synLatchReset (Q, g, d, reset);  
    output      Q;  
    input       g, d, reset;  
    reg         Q;  
  
    always @(g or d or reset)  
        if (~reset)  
            Q = 0;  
        else if (g)  
            Q = d;  
  
endmodule
```

## 6.4.2 触发器的推断

触发器是边沿触发的存储器件。触发器的行为一般由特定输入上的上升沿或下降沿所控制,这个特定输入被称作时钟。当边沿事件发生时,输入端  $d$  的值被保存,并选通到输出端  $Q$ 。它们通常具有置位和(或)复位输入,可以根据时钟来同步地或异步地改变触发器的状态。输出端  $Q$  值是输入端  $d$  无延迟的组合函数。这些触发器没有被显式地说明。然而它们可由行为推断出。因为它们的一些行为可能相当复杂,因此必须有模板来说明它们。事实上,一些综合工具都提供特殊的编译指令来指明触发器的类型。

例 6.12 给出了一个触发器的可综合模型。其中触发器描述的主要特征是 `always` 语句的事件表达式说明一个边沿。就是这个边沿事件在最终的设计中推断出触发器(与电平敏感的锁存器正相反)。正如我们将要看到的那样,带边沿触发事件表达式的 `always` 块会给所有在这个块的过程赋值语句中赋值的寄存器推断出触发器(因此,带边沿触发事件表达式的 `always` 块不能用来定义一个完整的组合函数)。

### 例 6.12 可综合的 D 型触发器

```
module synDFF (q, d, clock);
    output      q;
    input       clock, d;
    reg         q;

    always @(negedge clock)
        q <= d;
endmodule
```

触发器一般包括在系统开始时对状态进行初始化的一些复位信号。用来指定这些信号的方法具有一定的形式,以便综合工具可以确定要进行综合的器件的行为。例 6.13 给出了一个具有异步置位和复位功能的 D 型触发器。在该例中 `reset` 信号被声明为低电平, `set` 信号被声明为高电平,时钟事件出现在 `clock` 的上升沿。

### 例 6.13 带有置位和复位的可综合的 D 型触发器

```
module synDFFwithSetReset (q, d, reset, set, clock);
    input  d, reset, set, clock;
    output q;
    reg    q;

    always @(posedge clock or negedge reset or posedge set) begin
        if (~reset)
            q <= 0;
        else if (set)
```

```

        q <= 1;
    else q <= d;
end
endmodule

```

例 6.12 和例 6.13 在它们的说明中均用到了非阻塞赋值语句。这种说明在这些模块的多个示例连接在一起时可以产生正确的模拟。

尽管例 6.13 很直观,但格式十分严格,语义可以从语句和语句内的表达式的顺序中推断出。这种描述的形式必须遵循以下规则:

- always 语句必须为每个信号指定边沿。即使异步置位和复位信号不是边沿触发的也应这样说明。(它们不是边沿触发的,因为只要 reset 为 0——并不仅仅当下降沿出现时,q 就被保持为 0)。

- 紧跟在 always 后面的第一条语句必须是 if 语句。

- 在使用了 else-if 结构的 always 语句中,要首先对置位和复位条件进行测试。置位和复位表达式是不能使用下标引用方式;它们必须是 1 位的变量。对它们的值进行的测试必须简单,并且必须以事件表达式中所指明的顺序来完成。

- 如果像上面的 reset 那样指定了下降沿,那么测试应当为:

```
if (!reset)...
```

或

```
if (reset==1'b0)...
```

或

```
if (~reset)...
```

- 如果像上面的 set 那样指定了上升沿,那么测试应当为:

```
if (set)...
```

或

```
if (set==1'b1)...
```

- 在所有的置位和复位均被指定后,最后的语句指明发生在时钟边沿的动作。上例中 q 被输入 d 赋值。因此,“clock”不是保留字。综合工具根据控制路径的赋值位置来推断出特殊的时钟输入;这是置位或复位行为都不出现时发生的行为。

- 在 always 块中所有的过程赋值都必须是阻塞赋值或非阻塞赋值。它们不能共存于一个 always 块中。非阻塞赋值(<=)是说明电路边沿敏感的行为时使用的选择赋值操作符。“<=”表明:在整个系统中的所有指定在敏感列表的边沿发生的传输都应该同时

进行。尽管描述使用正规的“=”将能正确地综合,但不能正确地模拟。由于模拟和综合都相当重要,所以把“<=”用于边沿敏感的电路。

- always 块的敏感列表只包括时钟边沿、复位和预置位条件。只有这些输入才能引起状态改变。例如,如果我们描述一个 D 型触发器,D 的变化将不会改变触发器状态。所以 D 输入端不包括在敏感表中。

- 在顺序 always 块中被赋值的任何寄存器在最终被综合出来的电路中都使用触发器来实现。因此,不能在描述时序逻辑的同一个 always 块中,又描述纯粹组合的逻辑。可以写出一个组合表达式,但是表达式的结果应在时钟边沿被计算并被存入寄存器中。例 1.7 就是这样的一个示例。

参考:无阻塞赋值和阻塞赋值 5.4

### 6.4.3 小结

锁存器和触发器是寄存器传输级系统的基本组成元件。它们复杂的行为要求在其说明中使用严格的格式。我们仅讨论了它们规范的基本内容。大部分综合工具提供编译指令来帮助选择合适的库元件来实现指定的行为。请仔细阅读综合工具的使用手册。

## 6.5 三态器件的推断

三态器件是具有三个输出值的组合逻辑电路:1、0 和高阻态(z)。由于它们具有特定的、非典型的功能,这些器件设备必须从描述中推断出来。例 6.14 说明了一个三态器件的推断。

### 例 6.14 三态器件的推断

```
module synTriState (bus, in, driveEnable);
    input          in, driveEnable;
    output         bus;
    reg            bus;

    always @(in or driveEnable)
        if (driveEnable)
            bus = in;
        else
            bus = 1'bz;
endmodule
```

这个模型中的 always 语句符合描述组合逻辑函数的形式。这里的特殊情况在于条件(这个示例中是 driveEnable)指明了输出为高阻时的情况。综合工具推断出这个条件在最终实现中将是三态使能端。



## 6.6 有限状态机的描述

我们已经知道怎样为综合工具说明组合逻辑元件和时序元件。本节中我们将综合组合逻辑元件和时序元件来形成有限状态机的说明。有限状态机的标准形式如图 6.1 所示。有限状态机有输入  $x_i$ 、输出  $z_i$  和保持当前状态的触发器  $Q_i$ 。输出可以仅是当前状态的函数,这种情况下这是一个摩尔机(Moore machine)。输出也可以是当前状态和输入的函数,这种情况下这是一个米利机(Mealy machine)。触发器的输入是下一个状态;这是当前状态和输入的组合函数。

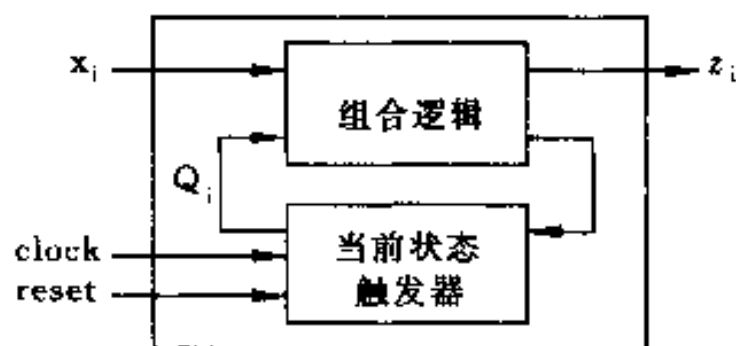


图 6.1 有限状态机的标准模型

有限状态机(FSM)的 Verilog 描述正好符合这个模型。图 6.1 中的外框是 FSM 模块,两个内框是两个独立的 always 语句。一个描述后继状态和输出的组合逻辑函数,另一个描述状态寄存器。

### 6.6.1 有限状态机的示例

本节给出一个采用显式风格描述的 FSM 示例。在这种风格里,使用 case 语句来指定状态机在各状态的动作及状态之间的转换。考虑图 6.2 所示的状态转换图。图中给出了六个状态和它们的状态转换,并有一个输入和三个输出。例 6.15 就是该有限状态机的 Verilog 描述。

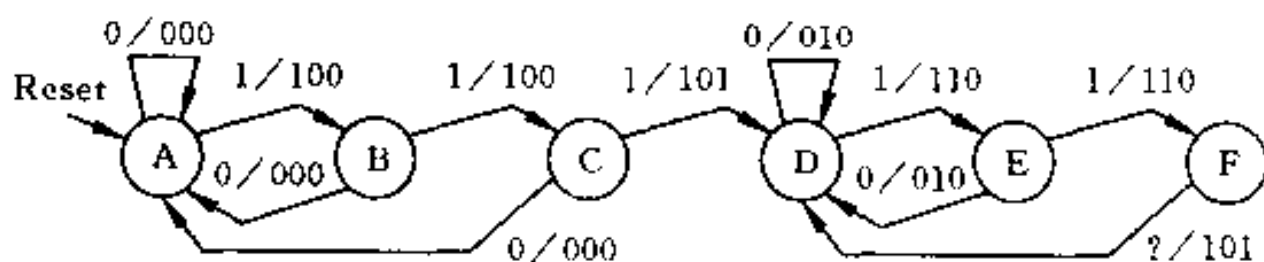


图 6.2 例 6.15 的状态转换图

第一个 always 语句是组合输出(out)和后继状态(nextState)函数的描述。这些函数的输入集合包括输入 i 和寄存器 currentState。这两者中任何一个变化都会引起 always 语句被重新计算。这个 always 语句中唯一的语句是 case 语句,它指明每个状态内所执

行的动作。case 语句的控制表达式是状态变量(currentState)。因此,根据有限状态机所处的状态,只有指定的动作才会出现。请注意:每一个 case 项中,要为两个被计算的组合函数(out 和 nextState)赋值。此外,列出的默认 case 项表示其余的未指定状态。默认值使机器转换成与复位相等同的状态 A。通过任意选择,在未指定的状态中,out 被置为无关项。

这个 always 语句将产生组合逻辑,因为:敏感列表包括所有的输入集,在敏感表中无边沿说明符,并且对于任一条控制路径,两个组合输出均被赋值。这包括了每一个可能的 case 项。因此,将不会有推断的锁存器。

第二个 always 语句根据复位条件推断出状态寄存器。在这种情况下,如果 reset 被设置为低电平,状态机将进入状态 A。如果 reset 没有被设置为低电平(即有效值),那么 always 语句的正常的动作是把 nextState 的值赋给 currentState,在 clock 上升沿时改变 FSM 的状态。

请注意,在 always 的每一个控制路径中 currentState 都被赋值——为什么还会推断出触发器?原因是事件表达式中的边沿说明使在块中被赋值的任何寄存器都用触发器来实现。不能用敏感列表中的边沿触发来说明 always 块中的组合逻辑。这就是为什么我们需要两个 always 块来说明 FSM 的原因:一个用于状态寄存器,另一个用于组合逻辑。

参数语句定义了 FSM 的状态赋值。如果不考虑参数的说明,可以给出另一种状态赋值语句。

两个 always 语句相互协作实现有限状态机的功能。第二个 always 语句的输出是 FSM 的当前状态,并且它在第一个 always 语句的输入集合中。第一个 always 语句是一个产生输出和后继状态函数的组合逻辑描述。

### 例 6.15 简单的有限状态机

```
module fsm (i, clock, reset, out);
    input          i, clock, reset;
    output [2:0]   out;
    reg [2:0]      out;
    reg [2:0]      currentState, nextState;

    parameter [2:0] A = 0,    // The state labels and their assignments
                  B = 1,
                  C = 2,
                  D = 3,
                  E = 4,
                  F = 5;

    always @(i or currentState) // The combinational logic
```

```

case (currentState)
  A: begin
    nextState = (i == 0) ? A : B;
    out = (i == 0) ? 3'b000 : 3'b100;
  end
  B: begin
    nextState = (i == 0) ? A : C;
    out = (i == 0) ? 3'b000 : 3'b100;
  end
  C: begin
    nextState = (i == 0) ? A : D;
    out = (i == 0) ? 3'b000 : 3'b101;
  end
  D: begin
    nextState = (i == 0) ? D : E;
    out = (i == 0) ? 3'b010 : 3'b110;
  end
  E: begin
    nextState = (i == 0) ? D : F;
    out = (i == 0) ? 3'b010 : 3'b110;
  end
  F: begin
    nextState = D;
    out = (i == 0) ? 3'b000 : 3'b101;
  end
  default: begin // oops, undefined states. Go to state A
    nextState = A;
    out = (i == 0) ? 3'bxxx : 3'bxxx;
  end
endcase

always @(posedge clock or negedge reset) // The state register
if (~reset)
  currentState <= A;
else
  currentState <= nextState;
endmodule

```

参考: 参数 4.5; 无阻塞赋值 5.4; 隐式风格 6.6.2

## 6.6.2 FSM 说明的另一种方式

上面说明 FSM 的显式方式很常见,可以说明任意的状态机。如果一个 FSM 是一个不帶有任何条件后继状态的单循环,可以使用隐式风格的说明。

隐式 FSM 规范的基本形式在例 6.16 中得到说明。唯一的 always 语句列出了几个时钟事件,所有的时钟事件都基于相同的边沿(上升沿或下降沿)。因为 always 说明一个顺序循环,有序地执行每一个状态,并且连续地执行循环语句。因此,没有说明后继状态函数。

在这个特殊的示例中,描述了一个数据流。每个状态计算出一个输出值(temp 和 dataOut),它将在随后的状态中使用。终结状态的输出(dataOut)就是 FSM 的输出。因此,每三个时钟周期在 dataOut 中产生一个新的结果。

数据流的另一个示例是流水线,使用一种略有不同的计算方法在例 6.17 中进行说明。这里,每个时钟周期都在 dataOut 中产生一个结果。在这个示例中,说明了三个有限状态机;每个状态机对应流水线的每个阶段。在每个时钟事件中,每个阶段都计算一个新的输出结果(stageOne、stageTwo 和 dataOut)。因为这些变量使用在带有边沿说明符的 always 块内过程语句的左边,所以用寄存器来实现。这里必须使用非阻塞赋值( $\leq$ ),以便确保模拟结果正确。图 6.3 显示了模块 synPipe 的一个简单实现形式。

参考: 显式风格 6.6.1

### 例 6.16 隐式 FSM

```
module synImplicit (dataIn, dataOut, c1, c2, clock);
    input          [7:0]      dataIn, c1, c2;
    input          clock;
    output         [7:0]      dataOut;
    reg            [7:0]      dataOut, temp;

    always begin
        @ (posedge clock)
            temp = dataIn + c1;
        @ (posedge clock)
            temp = temp & c2;
        @ (posedge clock)
            dataOut = temp - c1;
    end
endmodule
```

### 例 6.17 流水线

```
module synPipe (dataIn, dataOut, c1, c2, clock);  
    input      [7:0]      dataIn, c1, c2;  
    input      clock;  
    output     [7:0]      dataOut;  
    reg        [7:0]      dataOut;  
  
    reg        [7:0]      stageOne;  
    reg        [7:0]      stageTwo;  
  
    always      @ (posedge clock)  
        stageOne <= dataIn + c1;  
  
    always      @ (posedge clock)  
        stageTwo <= stageOne & c2;  
  
    always      @ (posedge clock)  
        dataOut <= stageTwo + stageOne;  
endmodule
```

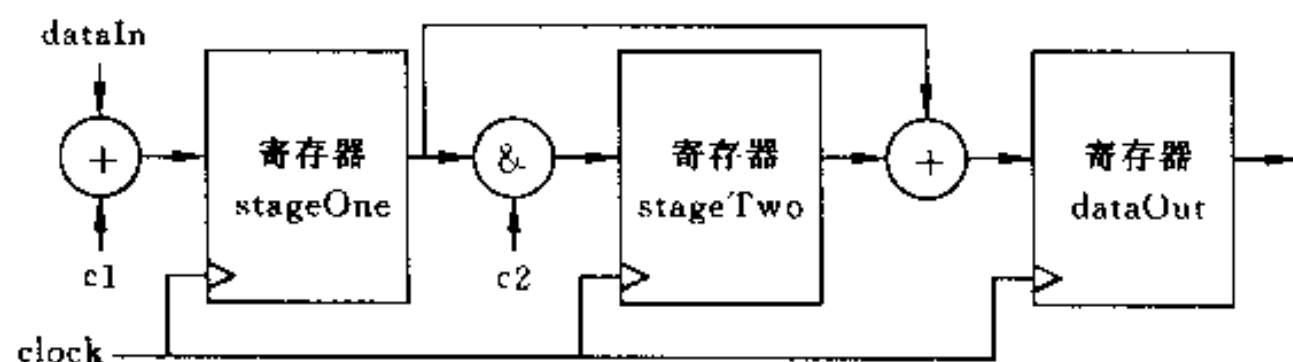


图 6.3 例 6.17 的数据通道

## 6.7 逻辑综合的总结

我们已经看到了用于逻辑综合的描述与风格是非常有关系的,并且为了更好地综合,一些结构被语义重载。此外,有些结构不允许出现在可综合的描述中。由于厂商和工具版本的不同,这些结构也有很大的不同。这里我们不包含这些结构。请参阅你所使用的综合工具的用户手册。

表 6.1 总结出使用过程语句描述组合逻辑的基本规则以及怎样在描述中推断出时序元件。

表 6.1 逻辑综合中使用过程语句的基本规则

逻辑类型	被赋值的输出	敏感列表中的边沿说明符
组合逻辑	在所有控制路径中输出都必须被赋值。	不允许。整个输入集必须在敏感表中。
推断出的锁存器	必须至少有一条未被赋值的控制路径。从这个“省略”中,工具推断出一个锁存器。	不允许。
推断出的触发器	无影响。	要求 --- 从当前的边沿指定中,工具推断出一个触发器。always 块中的所有寄存器都被说明的时钟边沿来定时。

## 6.8 习题

- 6.1 在 6.2 节中,我们说明了综合工具能够用不同的门基元来实现一定的功能,可能有些约束。试解释为什么产生一种实现会有“约束”?
- 6.2 改写例 6.5 中的描述,使得不再有推断出的锁存器。当 a 不为 1 时,b 和 c 相或来产生输出。
- 6.3 改写例 6.11 中的描述,使用 case 语句来推断锁存器。
- 6.4 为什么不能用 while 和 forever 循环来说明组合逻辑硬件?
- 6.5 把例 6.15 改写为摩尔机。必须增加一个附加状态。
- 6.6 使用 one-hot(一位有效)状态编码改写例 6.15,把描述改写为可被完全参数化的形式,使得任何状态编码均可被应用。
- 6.7 为图 6.4 所示的 FSM 写出一个带有输入 Ain、Bin、Cin、clock、reset 及输出 Y 的描述。

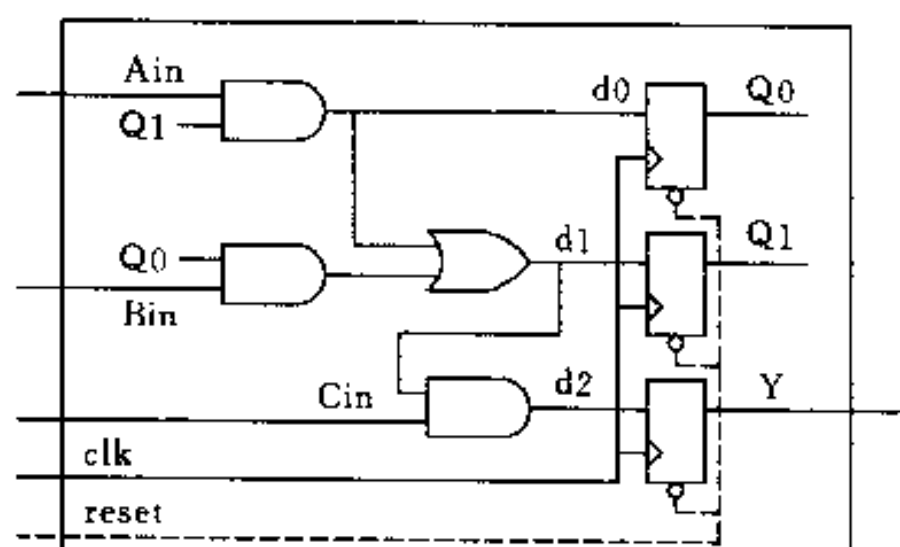


图 6.4 基于综合的控制

A. 一个单独的 always 块。

- B. 两个 always 块：一个用于组合逻辑，另一个用于时序逻辑。
- C. 这个电路太慢。我们不允许在触发器输入和输出之间有三个门延迟，最好只有两个门延迟。改变 B，以便 Y 是一个组合输出，即把产生 d2 的门移到触发器的另一边。
- D. 模拟上述三种情况，看看它们是否在功能上等价。

## 第7章 行为综合

行为综合是一项新兴的计算机辅助设计技术。这项技术工作在较高的抽象层,并且在开发设计的选择性方面比逻辑综合拥有更广阔的研究前景。通过使用一套周期精确的说明,这些工具设计并且优化了数据通道以及控制它的有限状态机。本章将向大家介绍行为综合的基本概念,周期精确的说明的表示方法,并且分析了如何采用行为综合技术来说明一个设计系统。

### 7.1 行为综合的介绍

行为综合技术使用目的在于帮助设计寄存器传输级系统——带数据通道系统的有限状态自动机。如图 7.1 所示,系统的功能是用周期精确的描述来说明的。通过描述的周期精确特性,可以推导出系统输入输出的时序关系。行为综合技术设计出数据通道和有限状态自动机,既实现了系统的功能,又满足了这些时序关系。这种设计依据功能的数据通道模块,例如 ALU、寄存器文件和工艺库文件提供的多路转换器/总线驱动源。此外,还确定了为系统设计的有限状态自动机。顺着图 7.1 的设计方向可知,设计工具包括逻辑综合和模块产生。其中,逻辑综合是为了设计有限状态自动机,而模块产生则是为了设计数据通道。

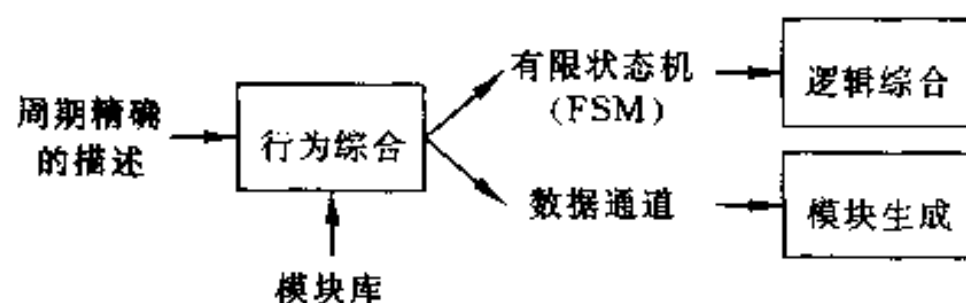


图 7.1 行为综合技术流程图

行为综合具有以下几项基本功能:调度,分配和映射。

- 调度将操作分配给各个控制状态。假设行为综合系统的输入是一个周期精确的说明。人们可能想了解调度扮演的角色。对于行为综合来说,周期精确的说明仅仅约束何时从输入端口读入数据,何时从输出端口写出数据。而在行为综合的内部,只要在适当的时候操作结果可以使用,行为综合可以把这些操作进行灵活调度。

- 分配确定需要用多少对象来实现数据通道。周期精确的说明仅告诉我们如何通



过计算输入数据获得输出结果。而行为综合技术则负责选定设计中所需的操作符数量(即是需要一个还是两个乘法器?)、寄存器数量以及总线数量。分配与调度结合使用可以在实现方面提供更大范围的折中。也就是说,如果存在两个乘法器,那么两个乘法操作可在一个控制态中同时执行。这样做使速度更快,但体积更大。

- **映射**将操作(例如 Verilog 过程语句中的“+”和“-”操作)分配给各功能模块。假设行为综合工具已确定在数据通道中有两个加法器,那么它也将决定描述中的哪个“+”操作符映射到哪个功能模块上。

此处所列的这些是在通常情况下的行为综合工具的功能。在已知的综合工具中,每一种工具的实现都不相同。此外,不同的工具也使输入描述风格各异。事实上,虽然我们采用周期精确的说明作为行为综合工具的输入,但是越来越多的人将会使用一种更高层次的说明。

本小节是基于应用 DASYS 公司的行为综合工具的经验来讨论的。阅读你的综合工具手册来了解工具的特点。

## 7.2 周期精确的说明

周期精确的说明在前边的 1.6 节已经有所介绍,在此我们将对其进行更加详细的描述。`always` 语句是精确周期性说明的基础。我们把这种语句看作是控制线程的一种描述:一个过程。`always` 语句的寄存器传输级的实现包括两个方面:一方面是实现 `always` 语句中说明的处理过程的数据通道;另一方面是有限状态自动机的一种描述,它控制数据通道中寄存器传输级的操作。在一个模块中可能包含多个 `always` 语句。其中每一个 `always` 语句都被综合成一对独立的(虽然它们也相互通信)数据通道和有限状态机。

### 7.2.1 `always` 块的输入和输出

虽然 `always` 块是一种没有标准端口说明的行为结构,但我们可以把其看作是有端口的。如例 7.1 所示,我们考虑的是一种只含有一个 `always` 块的模块,而且它不含有其他持续赋值或模块/门例示。此例向我们清楚的说明了模块的输入端口和输出端口与 `always` 块的输入和输出之间是相互对应的。也就是说,`always` 块需要作为输入的实体来自模块的外部,同时 `always` 块产生的实体在模块外部可以使用。当然,还可能有一些内部寄存器,它们被用来存储 `always` 块执行过程中产生的值,也被用作 `always` 块的输入。但是,因为 `always` 块的外部不能使用这些内部寄存器,所以这些寄存器不能作为输出。同时,由于这些寄存器是在内部产生的,因此也不能把它们作为输入。

**例 7.1** `always` 块的输入、输出和内部设置的说明

```
module inOutExample(r,s,qout);
```

```

input      [7:0]  r,s;
output     [7:0]  qout;
reg        [7:0]  qout,q;

always begin
    @(posedge clock)
        q<=r+s;
    @(posedge clock)
        qout<=q+qout;
end
endmodule

```

通常情况下,一个 always 块的内部寄存器集是一组仅能在 always 块内部使用的有名实体,并且这些有名实体都是位于 always 块内过程赋值语句的左部。这些有名实体包括寄存器和存储器。如在例 7.1 中,寄存器 q 就是内部寄存器集中的一员。但是寄存器 qout 却不属于内部寄存器,原因在于 qout 是在模块外被引用的。

always 块的输入集包括 always 块内过程赋值语句右部的所有有名实体和不属于内部寄存器集的条件表达式中的所有有名实体。也就是说,这类实体是由一种门基元、持续赋值语句或另一个 always 块产生的。在例 7.1 中,r 和 s 就属于输入集。

always 语句的输出集是一组不属于内部寄存器集的过程赋值语句左部的所有有名实体。换句话说,这些实体或者位于持续赋值语句的右部,或者是一种门基元的输入,或者属于另一个 always 块的输入集。如例 7.1 所示,qout 就是输出集中的一员。虽然在此例中 always 块的语句右部也调用了 qout,但因为在 always 块的外部也能使用它,所以把它归入输出集。

一个在周期精确的说明中使用的 always 块经常也会包含时钟和复位输入。例 7.1 就向我们例示了输入时钟的使用。从上述的定义出发,我们不把它们看作 always 块的输入。而把它们看作是特殊的控制输入。这一点类似于在有限状态自动机设计中的做法。在那里也不把时钟和复位归入输入。(为了明确我们的观点,我们有意将时钟排除在输入端口表之外。)

一个模块可能包含多个 always 块、门例示和持续赋值语句。从概念上讲,我们可以把一个 always 块的端口考虑成是由输入集和输出集两部分组成的。虽然没有形式化地列出这些端口,但我们可以认为每个 always 块都在读取输入,产生输出,并且通过输入和输出与系统的其余部分相互作用。

### 7.2.2 always 块的输入和输出关系

一个周期精确的描述是一个说明从输入集读和向输出集写的时序关系的 always 块。

输入输出间的关系确定了与外部的系统界面。通过在过程语句中插入时钟边沿来说明这些关系。时钟边沿说明也称为**时钟事件**。因此,我们也可如此说明:

```
always begin
    @(posedge clock) > state A
        q<= r+s;
    @(posedge clock) > state B
        qout<= q+qout;
end
```

此处的时钟事件是带有上升沿说明符的事件控制语句。当然,也可以把它说明为时钟的下降沿。

现在考虑如何读上面这些语句。标记了 State A 的语句暗示了在一个系统状态中发生的动作。当时钟 clock 的上升沿到来时, q 将被赋予 r 与 s 的和。也就是说,即使表达式  $q \leq r+s$  写在了等待时钟边沿的事件语句的下一行,但是通过语言的模拟语义,我们知道,如果在时钟边沿到来前 r 和 s 的值已经存在,那么就要计算 q 的值。只有当表达式右部都已计算出来后, q 值才能更新。同样的, qout 值的写入将发生在第二个时钟边沿,但是要在此时钟边沿前已经获得 q 和 qout 的值作为前提条件(当然, q 值的计算将发生在前一个状态)。通过这个说明,我们推断出在一个时钟周期内计算 r 和 s 的值,以及 qout 的值。也就是说,在状态 A, r 和 s 被采样输入并且求和得到 q 值。然后,在状态 B 将 q 加入 qout 中求和,产生 qout 的最终值。

图 7.2 再次展示了这段 Verilog 描述,这次包含有一个时序图和一个状态转换图。图例说明了系统从状态 A 转换为状态 B 与 q 在状态 A 获得新值是发生在同一个时钟边沿的。因此, q 的新值在状态 A 产生,直到系统处于状态 B 时,寄存器 q 才有效。

使用周期精确的方法来为系统建模时,我们仅在状态结束的时钟边沿采样 always 块的输入。因此,如图所示,即使 r 和 s 的值可以在较早时间产生(可能在前一个时钟事件),我们却只要求它们能在状态结束的边沿有效即可。毕竟,说明只在时钟周期(时钟事件)上是正确的,所以称为周期精确的说明。既然所有的动作都发生在时钟边沿,那么对输出集成员的赋值必须是非阻塞的。

行为综合中的一个重要概念是,输入集和输出集间的时序关系确定了与系统其余部分之间的完整界面。也就是说,为具有同样输入输出时序关系的行为综合出另一种实现的方式是可能的。这种实现方式在规模或者最大时钟频率方面可能会有所改变。考虑例 7.2 中的 Verilog 片段。

#### 例 7.2 周期精确说明的另一种实现

```
input    [7:0]    i,j,k;
output   [7:0]    f,h;
```

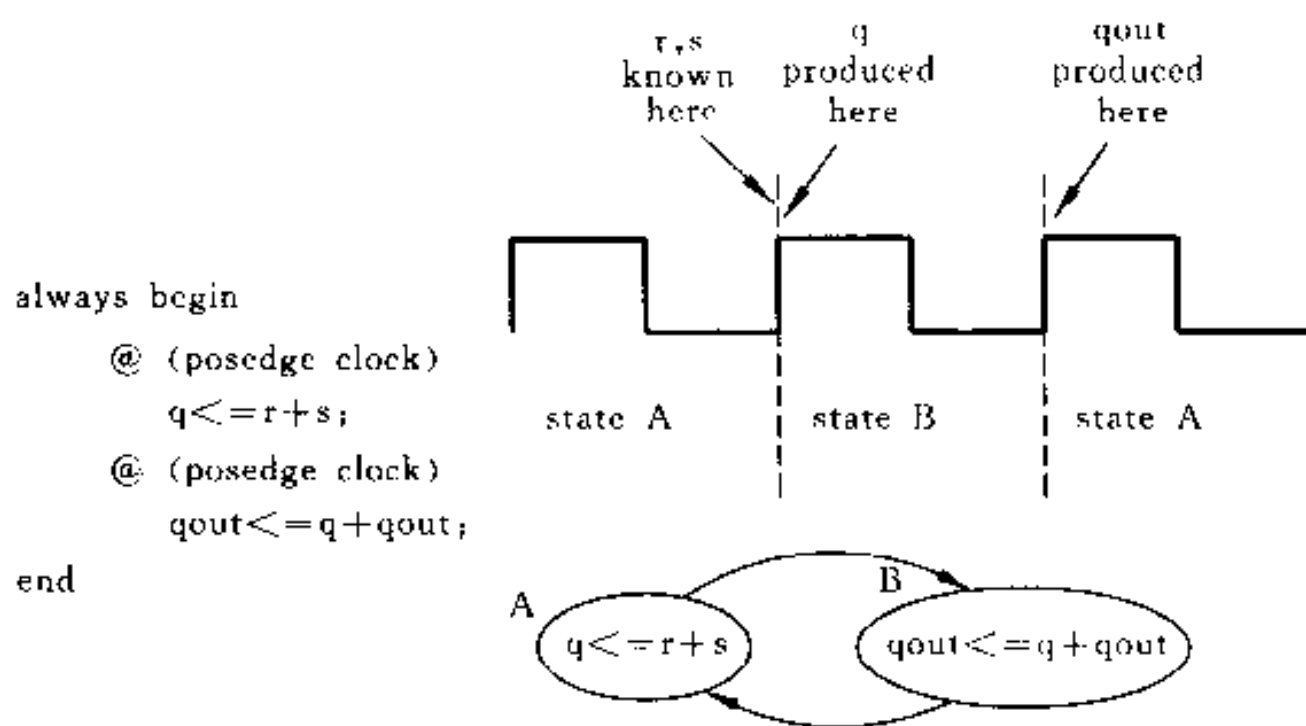


图 7.2 周期精确的行为

```

reg      [7:0]    f,g,h,q,r,s;

always begin
  ...
  @(posedge clock)
    f <= i + j;
    g = j * 23;
  @(posedge clock)
    h <= f + k;
  @(posedge clock)
    f <= f * g;
    q = r * s;
  ...

```

假设  $f$  和  $h$  属于输出集,  $i, j$  和  $k$  属于输入集, 寄存器  $g, q, r$  和  $s$  是内部集的一部分。注意, 状态 C 中的任何一个乘法可能在状态 B 中已经执行了, 因为这两条语句中用作乘法运算的值在状态 B 之前已经被计算出来。为此, 重新调度将是十分有利的, 因为在状态 C 中有两个乘法要调度。这也就意味着, 实现这个 Verilog 片段的数据通道不得不拥有两个独立的乘法单元来执行同一个状态中的两个乘法运算。如果我们把其中的一个乘法运算移到状态 B 中, 那么每个状态都将只需要一个乘法器在数据通道中。这样就节省了空间。行为综合工具能够识别把操作符重新调度到其他状态来节省空间的机会。

如果表达式  $q = r * s$  移入状态 B 中, 那么输入输出的功能将没有任何变化。然而, 若把表达式  $f <= f * g$  移入状态 B, 那么  $f$  的新值在此状态出现得太早了。输入和输出集的

时序关系将被改变。行为综合工具将为此在状态 B 中插入一个临时寄存器来保存这个新值,然后在状态 C 中再将此新值由临时寄存器存入输出 f 中。当然,设计中可能已经存在这种额外的寄存器。例如,假设在完成状态 C 之后,在 always 语句的下一次迭代中给寄存器 g 赋值之前,寄存器 g 不被访问,那么我们可将表达式  $f * g$  的值先存入寄存器 g 中,然后在状态 C 时再移入寄存器 f 中。将状态 B 和状态 C 改写为如下所示的状态 Bnew 和状态 Cnew:

<pre>@ (posedge clock)   h&lt;=f+k;   g=f*g;</pre>	}	State Bnew
<pre>@ (posedge clock)   f&lt;=g;   q=r*s;</pre>	}	State Cnew

人们可能已经发现设计者能够识别优化的时机并且实现它们。事实上,当我们不得不改变操作调度的安排时,设计者可能重新编写描述。然而,给出一种描述后,行为综合工具能迅速提供不同的、折中的实现方案。设计者可以从中进行选择,但并不是所有的行为综合工具都可以提供所有的转换形式。

参考:always 2.1; 控制线程 2.1; 输入集 6.3.1

### 7.2.3 复位功能说明

前边讨论的示例在此扩展为例 7.3。此例包含一个复位的电路行为描述。模块 accumulate 拥有输出端口(qout)、输入端口(r 和 s)以及本系统的特殊输入(clock 和 reset)。

always 块的复位功能由一条 initial 语句来说明。在这里,我们说明了一个异步的 reset,它被声明为低电平,initial 块把等待 reset 的下降沿作为开始。当这个边沿到来时,always 块中的 main 块被终止,同时 qout 置为 0,并且等待下一个 reset 的下降沿。这个动作致使退出有名的 begin-end 块(main)。当其退出时,always 块将重新开始执行,并且等待 reset 变为 TRUE(无效的)。在某种情况下,reset 变为无效的,系统将处于状态 A;在第一次时钟事件时,系统将从状态 A 转变为状态 B。注意,qout 通过复位被初始化为 0,同时系统也将从 0 开始累加。被描述的功能将通过 Verilog 描述得以实现,并且通过例 7.3 中的状态转换图进行说明。

#### 例 7.3 复位功能说明

```
module accumulate (qout, r, s, clock, reset);
  output [11:0] qout;
  input [11:0] r, s;
```

```

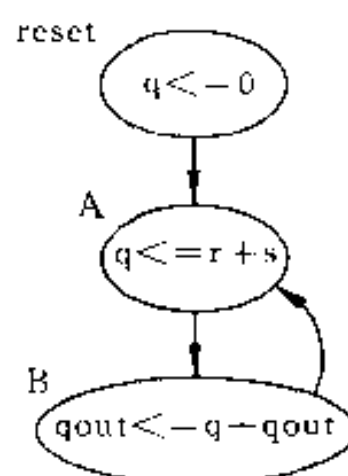
input          clock, reset;
reg    [11:0]  qout, q;

initial
    forever @(negedge reset) begin
        disable main;
        qout <= 0;
    end

always begin : main
    wait (reset);
    @(posedge clock)
        q <= r + s;
    @(posedge clock)
        qout <= q + qout;
end

endmodule

```



这里还有几点需要说明。如果 reset 是无效的,那么 always 块的行为将与前一节示例一样。可以通过在初始块中等待 reset 的上升沿,然后在 always 块中等待  $\sim\text{reset}$  的方式,将复位说明为高电平有效。最后,在“wait(reset);”语句和“@(posedge clock)”语句之间不能说明任何操作。如此一来,这样一个操作将不能归入状态 B 中,因为它必须以 reset 为条件——在有限状态机的设计中通常不允许的一个操作。当 reset 成为无效时,这种动作将在另一个状态中实现,并且需要一个时钟事件,以便它显式地成为状态的一部分。因此,一个时钟事件总是写在等待无效复位值的等待语句之后。

一个完整的、周期精确的系统说明现在包括 always 块和 initial 块两部分。两者共同说明一个控制线程,并且说明线程应如何复位到一个已知状态。当在这样一个设计层次讨论 always 块的输入和输出集的时候,同时考虑 always 块和初始块将会更加精确。因为分析 always 块仅确定了输入输出集,而初始块只说明复位的行为。

### 7.3 米利/摩尔机的说明

前几节的示例已经描述了采用周期精确的说明来为系统建模的一些基本内容。本节将向大家介绍这些说明是如何作为行为综合工具的输入使用的。

本节的示例分析了行为综合说明中的几个特征。首先,时钟事件的安排是任意的。它们可以被安排在条件语句中,且在不同的条件控制路径中时钟事件的数量也会有所不同。也就是说,if-then-else 语句中的任意一条分支并不需要具有相等数目的状态。在条

件语句和循环语句中,时钟事件的任意安排可以说明非常复杂的状态转换。例如,在一个时钟事件到来之前,可以深入多个嵌套的循环。这些安排的唯一限制就是一个循环体必须至少有一个时钟事件说明符,它可以处于循环体内的任意位置。

### 7.3.1 复杂控制的说明

例 7.4 描述了一个插入的三阶 FIR 滤波器,此滤波器每两个或四个时钟周期就会对输入(in)进行采样。基于采样的输入,计算出一个新的输出值 y。这个新值将会在未来的两个或四个时间周期内出现在输出端。用一个或三个中间时钟周期计算出来的中间值是新值(out)与前一次的 y 值(yold)之间的差。计算的结果被存入寄存器 delta 中,再除以 2。如果 switch 等于 0,寄存器 delta 的值将被加入到寄存器 yold(也可称 out)中,以便在下一个状态产生一个输出值。如果 switch 等于 1,寄存器 delta 的值将再一次被除以 2,并且用来为以后的三个状态产生输出值。

在本例中,状态 A 中寄存器 delta 的最终值依赖于 if 语句中的 then 分支是否被执行。也就是说,数据通道的控制信号依赖于状态信息(我们正处于状态 A 中)和系统的输入 switch。这就需要一台米利机来实现。

值得注意的是,当把结果写入输出集中的元素(在此例中指 out)时,将会使用一个非阻塞赋值语句。这排除了与任何其他赋值语句的所有竞争问题。所有其他赋值语句都是阻塞的,使得一条语句中赋予的值可以在下一条语句中使用。

#### 例 7.4 行为综合的说明

```
module synSwitchFilter (Clock, reset, in, switch, out);
    input          Clock, reset, switch;
    input  [7:0]    in;
    output [7:0]    out;
    reg  [7:0]      out, x1, x2, x3, y, yold, delta;

    initial forever @(negedge reset) begin
        disable main;
        out = 0;
        y = 1;
        x2 = 2;
        x3 = 3;
    end

    always begin :main
        wait (reset);
```

```

    @(posedge Clock)
    x1 = in;
    out <= y;
    yold = y;
    y = x1 + x2 + x3;
    delta = y - yold;
    delta = delta >> 1;
    if (switch == 1) begin
        delta = delta >> 1;
    end
    @(posedge Clock) out <= out + delta;
    @(posedge Clock) out <= out + delta;
end
endmodule

```

State A

### 7.3.2 数据与控制路径的折中

在本节中,我们将考虑一台用周期精确方法以行为综合方式说明的 8 点 FIR 滤波器的两种描述。在例 7.5 中,数组 coef\_array 和 x\_array 用来存储系数和各自前一次的输入。滤波器每 8 个时钟周期读入一个采样 x,且每 8 个周期输出一个结果 y。在例 7.5 和例 7.6 中产生了相同的模拟结果,但代表了不同的实现方式。

在例 7.5 中,模块 firFilt 使用了两个状态来说明 FIR 算法。第一状态(A)规定了累加器(acc)的初始值和数组 x\_array 的值。此外,寄存器 index 被初始化成开始位置(start\_pos)。随后,第二个状态(B)执行循环体,并进行循环结束条件检查。因此,它拥有了两个后继状态。循环体将会一直为 acc 和 index 产生新值。如果退出了循环,那么基于 index 更新后的值,y 的新值输出和数组中的下一个开始位置(start\_pos)的值也将产生。一条 disable 语句用来说明一个结束循环的测试语句。当综合以后,将会产生数据通道和一个具有两个状态的控制器。

#### 例 7.5 基本的 FIR

```

module firFilt (clock, reset, x, y);
    input          clock, reset;
    input  [7:0]    x;
    output [7:0]    y;

```



```

reg      [7:0]    coef_array [7:0];
reg      [7:0]    x_array [7:0];
reg      [7:0]    acc, y;
reg      [2:0]    index, start_pos;
                // important; these roll over from 7 to 0

initial
    forever @ (negedge reset) begin
        disable firmain;
        start_pos = 0;
    end

always begin: firmain
    wait (reset);
    @ (posedge clock);                // State A;
    x_array[start_pos] = x;
    acc = x * coef_array[start_pos];
    index = start_pos + 1;
    begin: loop1
        forever begin
            @ (posedge clock);        // State B;
            acc = acc + x_array[index] * coef_array[index];
            index = index + 1;
            if (index == start_pos) disable loop1;
        end
    end // loop1
    y <= acc;
    start_pos = start_pos + 1;
end
endmodule

```

当使用周期精确的说明时,在一个状态中只能有一个非阻塞赋值被用于输出集的某个成员。如果两个这样的赋值对应于一个寄存器,那么该寄存器最终的值将是不确定的。

图 7.3 的左部向我们展示出例 7.5 的状态转换图。图中仅仅把状态中要赋值的寄存器名称与条件标注在一起,并没有把实际的表达式写出来。这种状态转换图是十分简明的。

例 7.6 中的模块 firFiltMealy 是 FIR 算法的一种单态描述。正像典型的米利机那样,所以动作都被编码并标注在有限状态机不同的后继状态弧上。这里, firFilt 的动作就

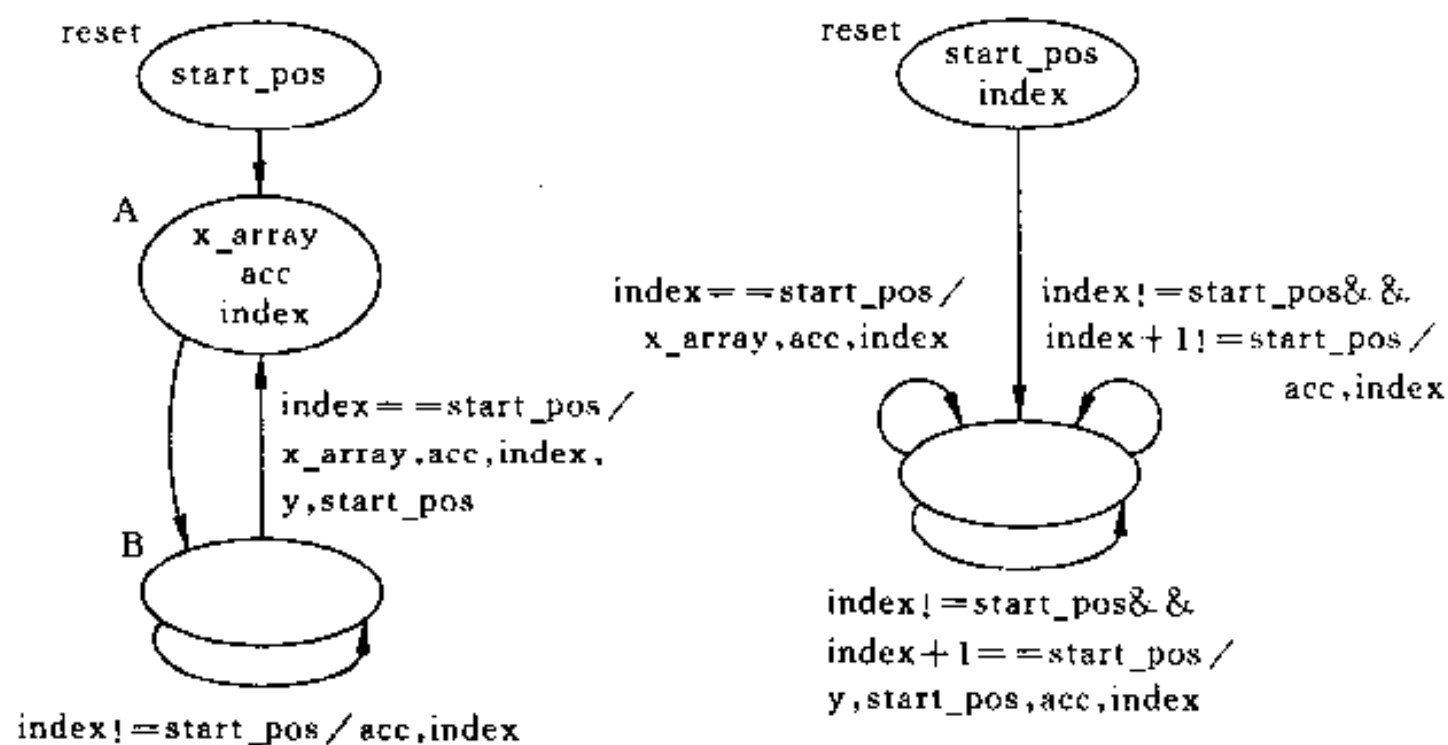


图 7.3 例 7.5(左侧)和例 7.6(右侧)的状态转换图

是如此编码的。firFiltMealy 显示了三种可能发生的动作;这些动作具有相同的后继状态(正好是当前状态)。第一个动作是第一条 if 语句的 then 分支。这相当于 firFilt 中的循环初始化。第二个动作是 else 分支,在这条分支里又存在两个可能的动作。第一个动作(这里不终止 loop1 的执行)更新 acc 和 index,这相当于 firFilt 的循环体。第二个动作既要更新 acc 和 index,又要更新 y 和 start\_pos。这相当于模块 firFilt 中的退出循环部分。有趣的是,当综合 firFiltMealy 时,没有明显的有限状态机。所有这些动作都把寄存器 index 和 start\_pos 的比较结果作为条件分支的依据。只有数据通道寄存器由时钟控制。

#### 例 7.6 米利 FIR

```

module firFiltMealy (clock, reset, x, y);
    input                clock, reset;
    input  [7:0]         x;
    output [7:0]         y;

    reg  [7:0]           coef_array [7:0];
    reg  [7:0]           x_array [7:0];
    reg  [7:0]           acc, y;
    reg  [2:0]           index, start_pos;

    initial
        forever @ (negedge reset) begin
            disable firmain;
            start_pos = 0;
            index = 0;
        end

```

```

always begin: firmain
    wait (reset);
    begin: loop1
        forever begin
            @ (posedge clock); // State 1—the only state
            if (index == start_pos) begin
                x.array[index] = x;
                acc = x * coef.array[index];
                index = index + 1;
            end
            else begin
                acc = acc + x.array[index] * coef.array[index];
                index = index + 1;
                if (index == start_pos) disable loop1;
            end
        end
    end
end
y <= acc;
start_pos = start_pos + 1;
index = start_pos;
end
endmodule

```

图 7.3 的右部向我们展示的是例 7.6 的状态转换图。在这里采用了(与上例 7.5)相同的表示方法,即只标注了被赋值的寄存器的名称。在此示例中,后继状态的转换依赖于 index 的当前值和更新后的值。index 更新后的值被记为 index + 1。

这些示例例示了设计者在说明复杂控制结构时所采用的控制。

## 7.4 小结

本章主要介绍了作为行为综合工具的输入使用的 Verilog 的用法。这些工具的输入使用了周期精确的说明形式。既然综合技术是门新兴的技术,那么今后在语言形式上的限定还将会不断改善。请读者多参考用户手册。

## 第 8 章 用户定义的基元

Verilog 提供了一个由 26 条门级基元组成的集合,用于对数字系统实际的逻辑实现进行建模。使用这些基元,可以像第 4 章所介绍的那样,对一些更大规模的模块进行层次化的描述。本章将介绍一种扩展门级基元的方法,使其包括用户定义的组合电路以及电平敏感和边沿敏感的时序电路。

需要扩展门级基元集合是有几个原因的。首先,用户定义基元是一种描述可能存在的各种逻辑块的非常简洁有效的方法。其次,使用用户定义基元可能会减少由模拟器的三值逻辑中的未知  $x$  值所带来的不现实性,从而为特定的情形建立更现实的模型。最后,通过它们的使用,可以获得模拟效率。注意,不管怎样,这些基元都不能用于描述逻辑综合设计。

### 8.1 组合基元

#### 8.1.1 用户定义基元的基本特征

正如例 8.1 所示,以类似于逻辑函数真值表的枚举方式定义了用户定义基元。基元被定义为与词法同级的模块。即在模块内不定义基元。本例描述了一个产生一位全加法器进位的基元。CarryOut 是输出,carryIn、aIn 和 bIn 是输入。列出的表给出了各种输入组合的输出值。用冒号将其右侧的输出与左侧的输入分隔开。表中各输入的顺序必须与基元定义语句端口表中各输入的顺序相一致。从表中第四行可以看到,如果输入 carryIn 是 0, aIn 和 bIn 都是 1,那么输出 carryOut 就是 1。

例 8.1 用户定义的组合基元

```
primitive carry(carryOut, carryIn, aIn, bIn);
output        carryOut;
input         carryIn,
              aIn,
              bIn;

    table
        0    00    :    0;
        0    01    :    0;
        0    10    :    0;
```