FPGA 原理、设计与应用

赵雅兴　主编

天 津 大 学 出 版 社

内容提要

本书全面介绍了 FPGA 的原理、设计与应用。主要内容有专用集成电路和可编程专用集成电路、ALTERA 可编程逻辑器件、MAX + PLUS Ⅱ 开发工具、ALTERA 硬件描述语言和设计实例与技巧。

本书叙述深入浅出、语言简练，可作为大专学校有关专业教材，也可供有关专业人员参考。

FPGA 原理、设计与应用　　赵雅兴　主编

出版发行：天津大学出版社（电话：022－27403647）

地　　址：天津市卫津路 92 号天津大学内（邮编：300072）

印　　刷：河北省昌黎县印刷厂

经　　销：新华书店天津发行所

开　　本：787mm×1092mm　1/16

印　　张：16 3/8

字　　数：410 千

版　　次：1999 年 4 月第 1 版

印　　次：1999 年 4 月第 1 次

印　　数：1001～4000

书　　号：ISBN 7－5618－1126－8/TN·14

定　　价：19.00 元

前　言

专用集成电路(ASIC)即特定的电子电路和系统(包括模拟、数字与数模混合电路)的设计与制造,在发达国家已经完成了由传统模式向现代化设计模式的转变,即完成了向电子线路与系统功能设计的转变。通过软件开发工具完成硬件电路的设计,近年来在国内也已经逐渐开展起来,并引进了一些国外的先进设计技术在各种新型电子设备和采用电子线路的设备中广泛使用。其中,由于"现场可编程门阵列"(FPGA)设计灵活、速度快,在数字专用集成电路的设计中得到更为广泛的应用。

由于专用集成电路设备价格低、功耗小、可靠性高、体积小、重量轻,已经普遍用于通信、雷达、导航、广播、电视、仪器、自动控制和计算机等领域。可以预计,到21世纪我国的专用集成电路研制工作将有飞速发展。但是,在这方面的设计人才还十分匮乏。摆在我们面前亟待解决的任务是,一方面深入进行高等院校的教育改革,更新教学内容,培养出适应这一发展需要的科技人才;另一方面更新现有电子设备(系统)研制人员的知识,进行继续教育。编写本书的目的就是为了让当代的大学生、研究生和从事电子设备制造的设计人员能由浅入深地迅速掌握数字专用集成电路的基本设计方法,以适应工作的需要。

本书是根据世界领先的著名可编程逻辑器件生产厂家ALTERA公司的器件用户手册和我们在开发、研究过程中积累的经验编写而成的。在编写过程中力求做到浅显易懂、便于应用,以达到学以致用的目的。全书共五章。第一章为绪论,概述可编程ASIC的分类、特点以及当前电子设计自动化(EDA)的发展状况;第二章介绍美国ALTERA公司MAX和FLEX器件的结构性能和特点;第三章介绍ALTERA公司的MAX+PLUSⅡ开发工具;第四章介绍ALTERA公司的硬件描述语言AHDL;第五章介绍设计的一般规则及应用实例。

在本书编写出版过程中,ALTERA公司给予了极大的支持和帮助。全书由天津大学电子信息工程学院电子信息技术系赵雅兴教授和赵松蹼、刘剑锋、李立勋等同志共同编写完成;韩改玲、刘东、张新苗、赵学义等同志也做了大量工作。书中不妥之处,望读者予以批评指正。

作者

1998.10

目　　录

第一章　绪　　论

1.1　专用集成电路(ASIC)概述

专用集成电路的英文写法是 Applications Specific Intergrated Circuit,简写为 ASIC。ASIC 是为专门限定的某一种或某几种特定功能的产品或应用而设计的芯片。所谓专用集成电路是相对于通用集成电路而言的。专用集成电路又分为模拟和数字两大类,而本书只涉及数字专用集成电路。从目前制造的方法看,数字专用集成电路可分为全定制 ASIC(Full Custom ASIC)、半定制 ASIC(Semi-Custom ASIC)和可编程 ASIC(Programmable ASIC)三大类别。全定制 ASIC 芯片没有经过预加工,各层掩膜全部是按特定功能专门制造的;半定制 ASIC 是在硅片上已经预制好晶体管单元电路(这种硅片可以称为母片),只剩金属连线层的掩膜有待按照具体要求进行设计和制造。因此,和全定制 ASIC 相比,当生产量不大时,半定制的成本低而且设计和生产周期都很短。可编程 ASIC 的芯片各层均已由工厂预先制造好,不需要定制任何掩膜,用户可以用开发工具按照自己的设计对可编程器件编程,以实现特定的逻辑功能。

1.2　可编程专用集成电路

可编程逻辑器件(Programmable Logic Device)简称 PLD,是新一代的数字器件。它不仅具有很高的速度和可靠性,而且具有用户可重复定义的逻辑功能即具有可重复编程的特点。因此,可编程逻辑器件使数字电路系统的设计非常灵活,并且大大缩短了系统研制的周期,缩小了数字电路系统的体积和所用芯片的品种。

可编程逻辑器件 PLD 分类框图如图 1-1 所示。

1.2.1　简单 PLD

简单的 PLD 是由"与"阵列及"或"阵列组成,能有效地以"积之和"的形式实现布尔逻辑函数。从技术实现上,输入到 PLD 的信号必须首先通过一个"与"门阵列,在这里形成输入信号的组合。每组相"与"的组合被称为布尔表达式的子项或 PLD 术语中的乘积线。这个乘积线在第二个"或"门阵列中被相加。简单 PLD 在"与"、"或"阵列的基础上有三种基本类型,可根据阵列能否编程来区分:①可编程只读存储器(Programmable Read-Only Memory)即 PROM,它的"与"阵列固定,"或"阵列可编程;②可编程阵列逻辑(Programmable Array Logic)即 PAL,它的"与"阵列可编程,"或"阵列固定;③可编程逻辑阵列(Programmable Logic Array)即 PLA,

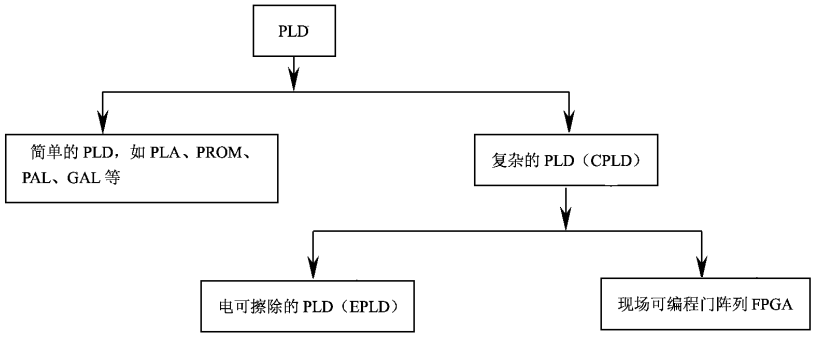


图 1－1　PLD 分类框图

它的‘与’阵列和‘或’阵列都可编程。

可编程只读存储器(PROM)的内部结构如图 1－2 所示。

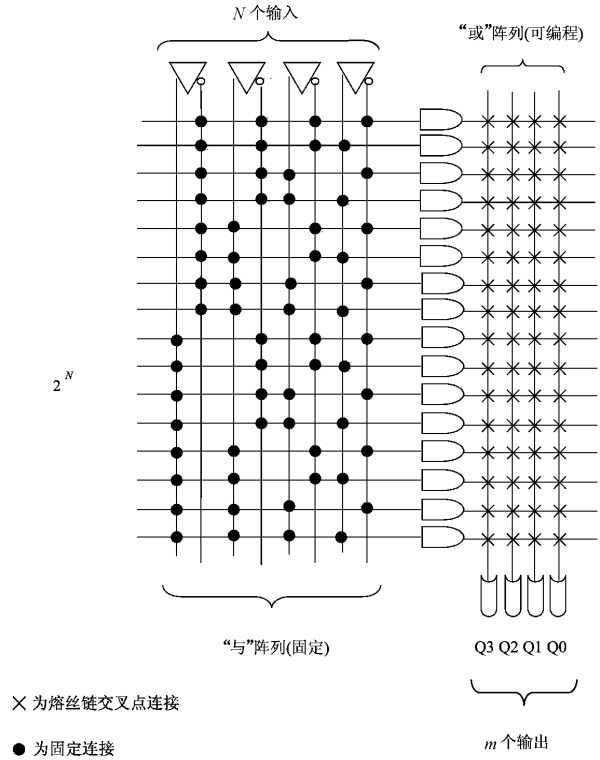


图 1－2　PROM 器件内部结构图

由图 1－2 可以看出，由于‘与’阵列是固定的，输入信号的各种可能组合是由连接线连接好的，不管组合是否会被使用。因此从某种意义上说，PROM 又十分类似于一个查找表，即根据用户要求在‘表’中查找所需要的可能组合。

可编程逻辑阵列(PLA)的内部结构如图 1－3 所示。

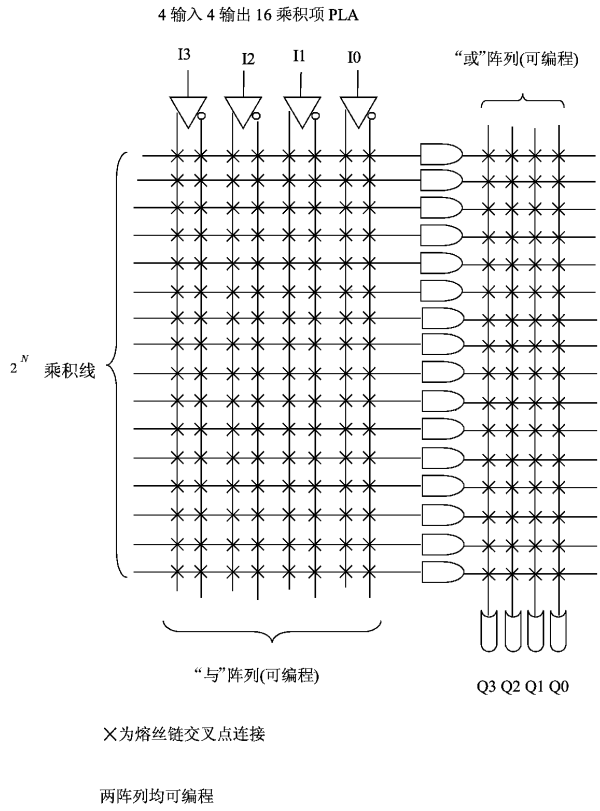


图 1－3　PLA 器件内部结构图

由图 1－3 可以看出，由于‘与’阵列可编程而不需要包含输入信号各种可能的组合，所需包含的组合只是在逻辑功能中实际要求的那些组合。这不仅提供了在可编程器件中的高度灵活性，而且也不会出现在 PROM 器件中由于输入信号数量增加而使器件规模增大的问题。

可编程阵列逻辑(PAL)的内部结构如图 1－4 所示。

通用阵列逻辑(Generic Array Logic —— GAL)器件与 PAL 器件具有相同的内部结构，但又靠各种特性组合而被区别。这类器件综合了 PROM 器件编程的低成本、高速度、容易编程和 PLA 的灵活性，因此成为最早实现可编程 ASIC 的主要器件。尤其是 GAL 的可再编程特性，为开发提供了很大方便。

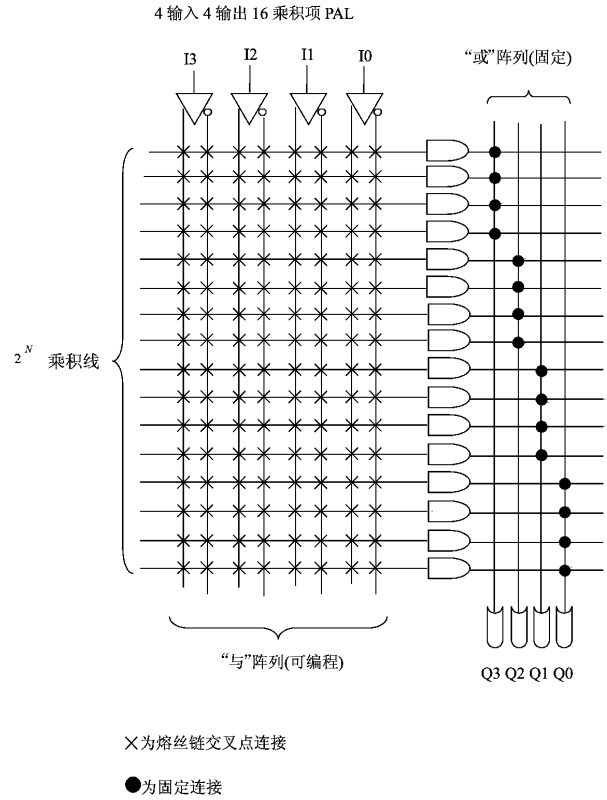


图 1－4　PAL 器件内部结构图

1.2.2　复杂的 CPLD(Complex Programmable Logic Device)

复杂的可编程逻辑器件 CPLD 是由 PAL 或 GAL 发展而来的，基本上是扩充原始的可编程逻辑器件。它通常是由可编程逻辑的功能块围绕一个位于中心和延时固定的可编程互连矩阵构成。

为了增加电路密度而不使性能或功耗受到损失，复杂的可编程逻辑器件 CPLD 在结构上引入了各种特性。如 :引入分页系统，分页的目的在于仅使阵列的一部分在任何给定的时刻被加电 ;按备份模式放置阵列，或者靠变换检测自动地控制加电，或者采用外部指令加以控制。在实现级上，“ 与 ’阵列及‘ 或 ’阵列需要用缓冲器分开，因为这些一般是倒相器。在两个阵列中实际逻辑一般是相同的。某些公司已经引入了折叠 PLA，它仅用了一个实际阵列，但可以将乘积项反馈回阵列。这也允许在单个器件中实现多级逻辑。

从目前发展趋势可以看出 CPLD 又延伸出两大分支，即可擦除可编程的逻辑器件 EPLD

(Erasable Programmable Logic Device)和现场可编程门阵列器件 FPGA(Field Programmable Gate Array)。

EPLD 可擦除可编程逻辑器件分为两类,一类是 UV 可擦 PLD,称为 EPLD;另一类是电可擦 PLD,简称 EEPLD。ALTERA 公司自 80 年代中期推出 EPLD 以来,已经有多种产品推向市场,其中典型代表产品是 MAX7000 系列,它属于电可擦除可编程的逻辑器件。

FPGA 现场可编程门阵列器件通常由布线资源围绕的可编程单元(或宏单元)构成阵列,又由可编程 I/O 单元围绕阵列构成整个芯片,如图 1-5 所示。排成阵列的逻辑单元由布线通道中的可编程连线连接起来实现一定的逻辑功能。一个 FPGA 可能包含有静态存储单元,它们允许内连的模式在器件被制造以后再被加载或修改。

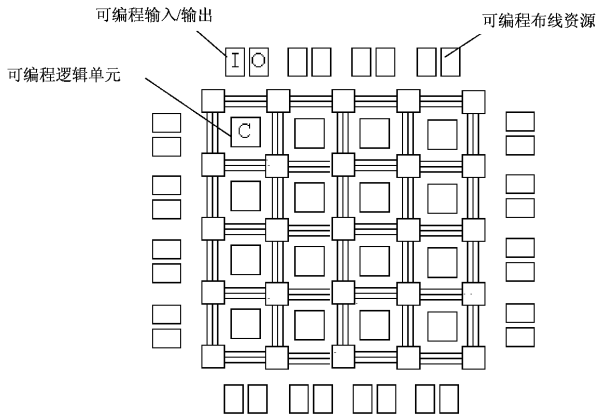


图 1-5 FPGA 的基本结构

FPGA 是由掩膜可编程门阵列和可编程逻辑器件演变而来的,将它们的特性结合在一起,使得 FPGA 既有门阵列的高逻辑密度和通用性,又有可编程逻辑器件的用户可编程特性。目前 FPGA 的逻辑功能块在规模和实现逻辑功能的能力上存在很大差别。有的逻辑功能块规模十分小,仅含有只能实现倒相器的两个晶体管;而有的逻辑功能块则规模比较大,可以实现任何五输入逻辑函数的查找表结构。据此可把 FPGA 分为两大类,即细粒度(fine-grain)和粗粒度(coarce-grain)。细粒度逻辑块是与半定制门阵列的基本单元相同,它由可以用可编程互连来连接的少数晶体管组成,规模都较小,主要优点是可用的功能块可以完全被利用;缺点是采用它通常需要大量的连线和可编程开关,使相对速度变慢。由于近年来工艺不断改进,芯片集成度不断提高,加上引入硬件描述语言(HDL)的设计方法,不少厂家开发出了具有更高级程度的细粒度结构的 FPGA。例如,XILINX 公司的采用 Micro Via 技术的一次编程反熔丝结构的 XC8100 系列,它的逻辑功能块规模较小,而粗粒度功能块规模较大并且功能较强。从构成它的可编程逻辑块和可编程互连资源来看,主要有两类逻辑块的构造。其一是查找表类型;其二是多路开关类型,由此形成两种 FPGA 的结构。

第一种是具有可编程内连线的通道型门阵列。它采用分段互连线,利用不同长度的多种金属线经传输管将各种逻辑单元连接起来。布线延时是累加的、可变的,并且与通道有关。

第二种是具有类似 PLD 可编程逻辑块阵列的固定内连布线,采用连续互连线,利用相同

长度的金属线实现逻辑单元之间的互连，布线延时是固定的，并且可预测。

XILINX 公司和 ACTELl 公司的 FPGA 属于第一种 FPGA 结构。从逻辑块构造看，XILINX 公司的 FPGA 属于查找表类型，ACTEL 公司的 FPGA 属于多路开关类型。而 ALTERA 公司的 FPGA 则是由传统的 PLD 结构演变而来，因此应属于具有类似 PLD 的可编程逻辑块阵列和连续布线这一类，即第二种 FPGA 结构，其逻辑块是基于‘与’或’门电路构成的。

下面针对这三种 FPGA 的结构特点分别介绍。

一、查找表型 FPGA 结构

不同公司产品的查找表型 FPGA 的结构各有特点，但可编程逻辑器件单元基本上都是查找表的静态存储器(SRAM)构成函数发生器，并由它去控制执行 FPGA 应用函数的逻辑。如果有 N 个输入，那么将有 N 个输入的逻辑函数真值表存储在一个 $2^N \times 1$ 的 SRAM 中。SRAM 的地址线起输入作用。SRAM 的输出为逻辑函数的值，由此输出状态去控制传输门或多路开关信号的通断，实现与其他功能块的可编程连接。

查找表结构的优点是功能很多。N 输入的查找表可以实现 N 个任意函数，这样的函数高达 2^{2^N} 个。但是，这也将带来一些问题，如若有多于 5 个输入，则由于 5 个输入查找表的存储单元数是 2^5，它可以实现的函数数目增加得太多，而这些附加的函数在逻辑设计中又经常用不到，并且也很难让逻辑综合工具去开发利用。所以在实际产品中，一般查找表型 FPGA 的查找表输入 $N \leqslant 5$。例如 XILINX 公司的 XC2000 系列的逻辑块是由 4 输入和 1 输出的查找表组成。它可以生成任何四输入变量的逻辑函数，可配置逻辑块 CLB 的方框图如图 1－6 所示。

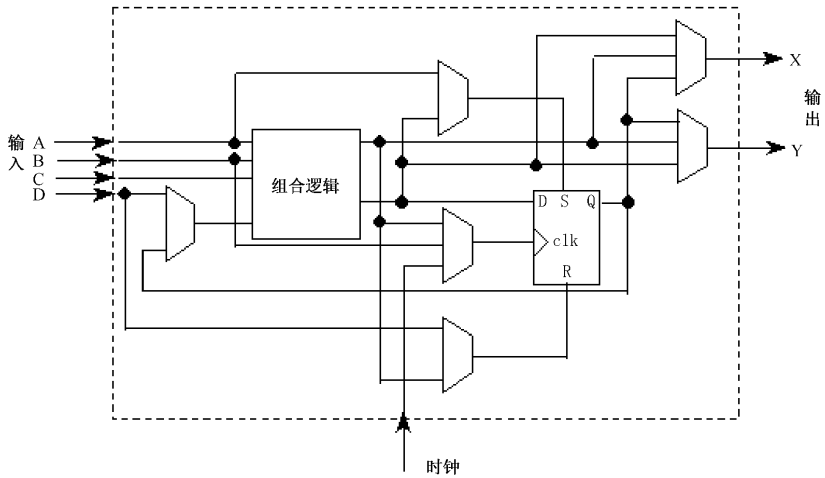


图 1－6　XC2000 系列 CLB 方块图

图 1－7 表示 XILINX 公司的 XC4000 系列的可配置逻辑块(CLB)的方块图。可配置逻辑块包括函数发生器、触发器和编程控制的多路开关。一个函数发生器就是一个 N 输入的 2^N 位存储器，可实现 2^{2^N} 个 N 输入的任何函数。该器件可用变址方法到存储器的真值表中配

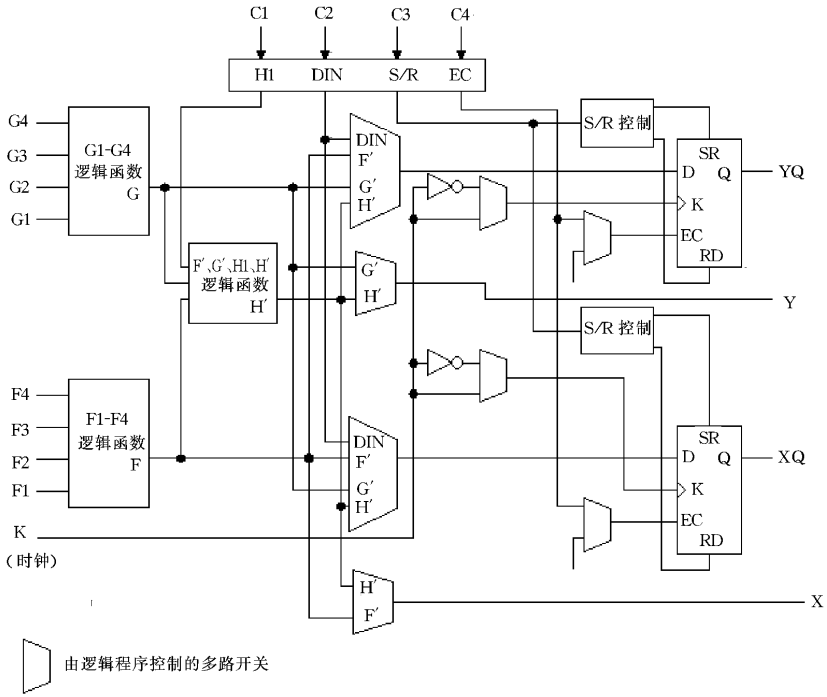


图 1－7　XC4000 系列 CLB 方块图

置逻辑块。它们包含三个函数发生器，分别用 F、G、H 表示。两个第一级的输入函数发生器 F 和 G 为四输入查找表，每个实现四输入的任何函数，其输出可以独立地从这个块输出。或者在 H 函数发生器三输入查找表中组合成五输入的任何函数，或者高达九输入的一些函数。这样就可以允许某些高输入函数，如九输入的‘与’或‘双’异或”，并且在一个逻辑块中译码。四输入查找表的 F 和 G 与三输入查找表 H 之间的连接采用了非可编程的固定连接。尽管这样不具有灵活性，但由于不存在改变查找表形式的可编程连接，所以速度显著加快。

图 1－8 是 XC3000 系列的 CLB 方块图。每个 CLB 由一个组合函数发生器和两个 D 触发器组成。它包括 5 个逻辑输入端（a、b、c、d、e）、一个公共时钟输入 k、一个异步直接复位输入 RD、一个使能时钟 EC 及两个输出 X、Y，还提供一个 Data-in 输入（以便对 CLB 中的触发器直接输入）、两个输出（可以由函数发生器或由触发器来驱动）。两个触发器的输出可以不通过 CLB 的外部直接布线返回到函数发生器的输入。

图 1－9 给出三种可用的组合逻辑。XC3000 系列的函数发生器是由两个四输入的查找表组成的。它可以单独使用，也可以组成单个的函数。由于 CLB 仅有 5 个输入到函数发生器，所以这 5 个输入必须在两个查找表之间共享。对于 F、G 模式，函数发生器提供任何两个 A、B、C 和 D 或 E 四输入函数，其中 D 和 E 之间的选择由每个函数分别进行；F 模式，所有五个输入组合成单个的五输入函数；FGM 模式，两个 A、B、C 和 D 的四输入函数按第五个变量 E

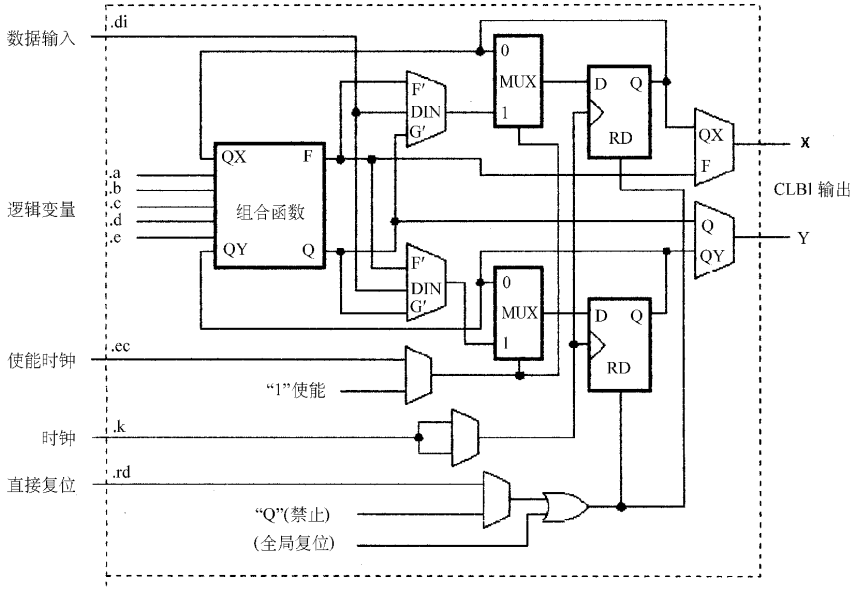


图 1－8　XC3000 系列 CLB 方块图

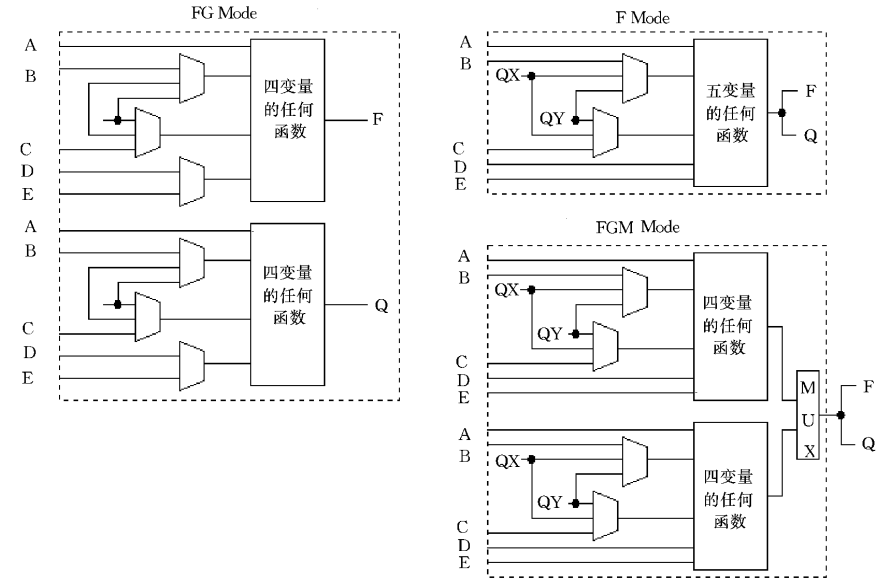


图 1－9　XC3000 系列的三种组合逻辑

复合在一起。对于上述所有模式，B 或 C 输入可选触发器输出 QX 或 QY 代替。在 FG 模式中，这个选择是对两个查找表分别进行的，功能上扩展到由七变量选出的任意两个四变量函数。只要这些变量的两个存储在触发器中，在类似状态机的应用中就特别有用。在 F 模式中，函数发生器由七个变量中选出单个五变量函数。当选择 QX 和 QY 后，两个查找表的内容也就被限定了。FGM 模式与 Y 模式不同，QX 和 QY 可以在两个查找表中分别选择，这一点与 FG 模式相同。这个附加的灵活性允许仿真包括所有七个可能输入的被选函数。

图 1－10 为五输入查找表实现一位全加器的原理图。当输入信号为 A_0 和 B_0 且进位输入位为 C_1 时，全加器输出为 S_0 和 C_0。逻辑方程为：

$$S_0 = A_0 \oplus B_0 \oplus C_1$$

$$C_0 = A_0 C_1 + B_0 C_1 + A_0 B_0$$

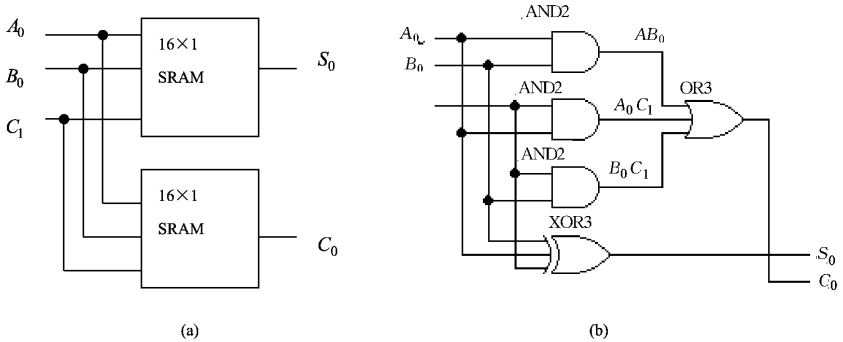


图 1－10　基本的 XILINX 查找表宏单元实现一位全加器
(a) XC3000 系列 FG 模式；(b) 实现一位全加器的逻辑图

表 1－1　全加器输入和输出的真值表

输　入			输　出	
A_0	B_0	C_0	S_0	C_0
0	0	0	0	0
1	0	0	1	0
0	1	0	1	0
1	1	0	0	1
0	0	1	1	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	1

表 1－1 列出一位全加器输入和输出的真值表。在查找表中存储的是全加器真值表的输出数值，而输入变量相应为查找表的地址。由于有两个输出变量，对于 XILINX 的 XC3000 系列，采用它的 FG 模式及两个四输入函数的模式如图 1－9 所示。它将原来的 32×1 的 SRAM 分成两个 16×1 的 SRAM，而且每个存储器仅用一半存入真值表数值即可。若采用 XILINX

的 XC2000 系列，可用图 1－11 所示的两个三变量函数模式，将四输入的 16×1 的 SRAM 分成两个三输入的 8×1 的存储器作为查找表，分别存入 S_0 和 C_0 的真值表数值。

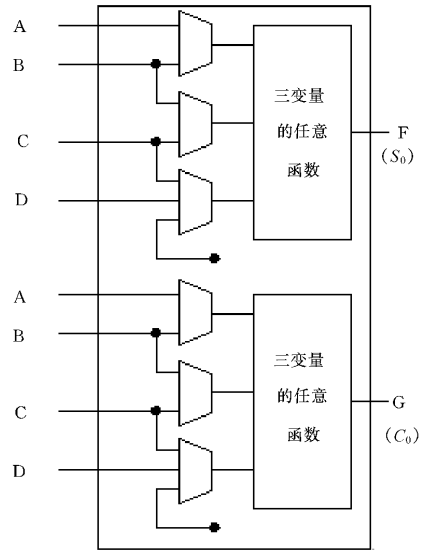


图 1－11　XC2000 系列两个三变量函数

二、多路开关型 FPGA 结构

多路开关型 FPGA 的基本模块是一个多路开关的配置。在多路开关的每一个输入端接上固定电平或输入信号时，可以实现不同的逻辑功能。例如图 1－12 为基本 ACTEL 多路开关型逻辑块的二到一开关。它包含一个具有选择输入 S、两个输入 a 和 b，是一种二到一型多路开关。其输出表达式为：

$$f = Sa + \overline{S}b$$

当 b 输入逻辑零时

$$f = Sa$$

多路开关实现 S“与”a 的功能。当 a 输入置逻辑 1 时，有

$$f = S + b$$

多路开关实现 S 或 b 功能。

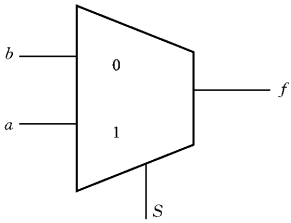


图 1－12　多路开关型逻辑块

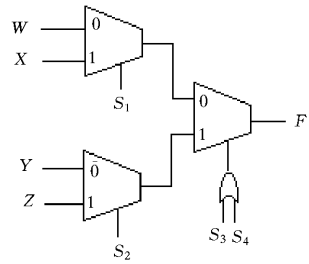


图 1－13　基本的 ACTEL 多路开关型逻辑块

如果把大量的多路开关和逻辑门连接起来，可以构成大量函数的逻辑块。ACTEL 公司的 FPGA、ACT－1 就是由三个两输入多路开关和一个“或”门组成的基本模块，如图 1－13 所示。这个基本模块可称作“宏单元”，具有八个输入和一个输出，输出表达式为：

$$f = (\overline{S_3 + S_4})(\overline{S}_1 W + S_1 X) + (S_3 + S_4)(\overline{S}_2 Y + S_2 Z)$$

当设置每个变量为一个输入信号或一个固定电平时，可以实现 702 种逻辑函数，例如当设置

$$W = A_0, X = \overline{A}_0, S_1 = B_0, Y = \overline{A}_0, Z = A_0, S_2 = B_0, S_3 = C_1, S_4 = 0$$

时，有

$$f = (\overline{C_1 + 0})(\overline{B}_0 A_0 + B_0 \overline{A}_0) + (C_1 + 0)(\overline{B_0 A_0} + B_0 A_0)$$

$$
\begin{aligned}
&= \overline{C_1}(A_0 \oplus B_0) + C_1(\overline{B_0 A_0} + B_0 A_0) \\
&= \overline{C_1}(A_0 \oplus B_0) + C_1(B_0 \overline{B_0} + \overline{B_0}\,\overline{A_0} + B_0 A_0 + A_0 \overline{A_0}) \\
&= \overline{C_1}(A_0 \oplus B_0) + C_1[(B_0 + \overline{A_0})(\overline{B} + A_0)] \\
&- \overline{C_1}(A_0 \oplus B_0) + C_1[(\overline{\overline{B_1}} + \overline{A_0})(\overline{B_1} + \overline{\overline{A_0}})] \\
&- \overline{C_1}(A_0 \oplus B_0) + C_1(\overline{B_0} A_0)(B_0 \overline{A_0}) = \\
&= \overline{C_1}(A_0 \oplus B_0) + C_1(\overline{\overline{B_0} A_0 + B_0 \overline{A} 0}) \\
&= \overline{C_1}(A_0 \oplus + B_0) + C_1(\overline{A_0 \oplus B_0}) \\
&= (A_0 \oplus B_0) \oplus C_1
\end{aligned}
$$

由所得函数 f **的表达式可知,此时所得恰为全加器输出的逻辑函数。再如,当设置**

$$W=0 \quad X=C_1 \quad S_1=B_0 \quad Y=C_1 \quad Z=1 \quad S_2=B_0 \quad S_3=A_0 \quad S_4=0$$

时,有

$$
\begin{aligned}
f &= (\overline{A_0 + 0})(\overline{B_0} + B_0 C_1) + (A_0 + 0)(\overline{B_0} C_1 + B_0 1) \\
&= \overline{A_0} B_0 C_1 + A_0 \overline{B_0} C_1 + A_0 B_0 \\
&= B_0 C_1 + A_0 C_1 + A_0 B_0
\end{aligned}
$$

由所得的函数 f **的表达式可知,此时所得恰为全加器输出的逻辑函数。图** 1－14 **给出** ACT－2 **组合逻辑块的结构图。它是第二类** ACTEL **多路开关型的逻辑块执行四到一线的多路开关作用,可以实现** 766 **种函数。**

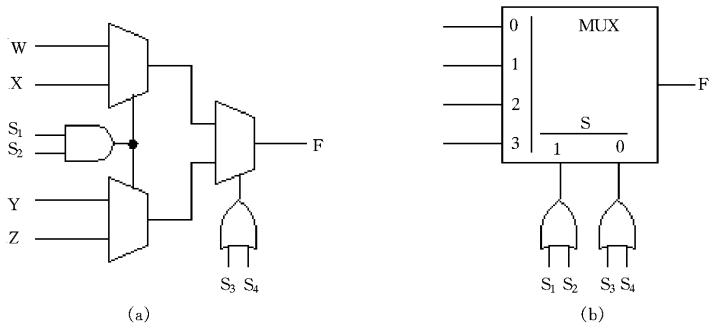


图 1－14　**第二类** ACTEL **多路开关型逻辑块**

(a)多路开关型逻辑块结构 ;(b)多路开关型逻辑块符号

三、多级‘与" 或 ’门 FPGA **结构**

多级‘与" 或 ’门 FPGA **是基于可以实现‘与 ”—“ 或 ’逻辑的‘与 ”—“ 或 ’电路,其输出馈送到一个‘异或 ’门,如图** 1－15 **所示。这个基本电路可以用一个触发器和一个多路开关扩充,如图** 1－16 **所示。此多路开关可选择锁存的输出信号或非锁存的输出信号。这里的‘异或 ’门可以用来获得可编程的‘非 ’逻辑。如果一个‘异或 ’门的输入端是分离的,起作用同‘或 ’门一样,可允许‘或 ’门和‘异或 ’门形成更大的‘或 ’函数,用来实现算术功能。**

ALTERA **公司的** MAX5000、MAX7000 **和** MAX9000 **系列产品就是属于多级‘与或 ’门的** FPGA **结构。图** 1－17 **示出用** MAX5000 **系列器件的逻辑单元实现一位全加器的原理图。**

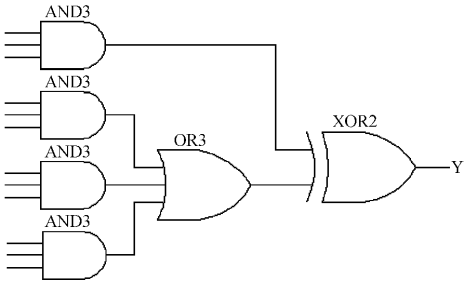


图 1－15 “与”—“或”—“异或”逻辑块

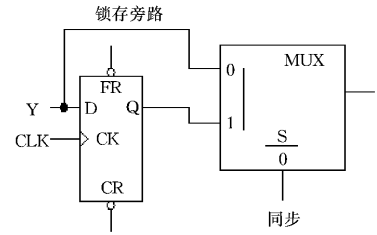


图 1－16 同步/非同步输出电路

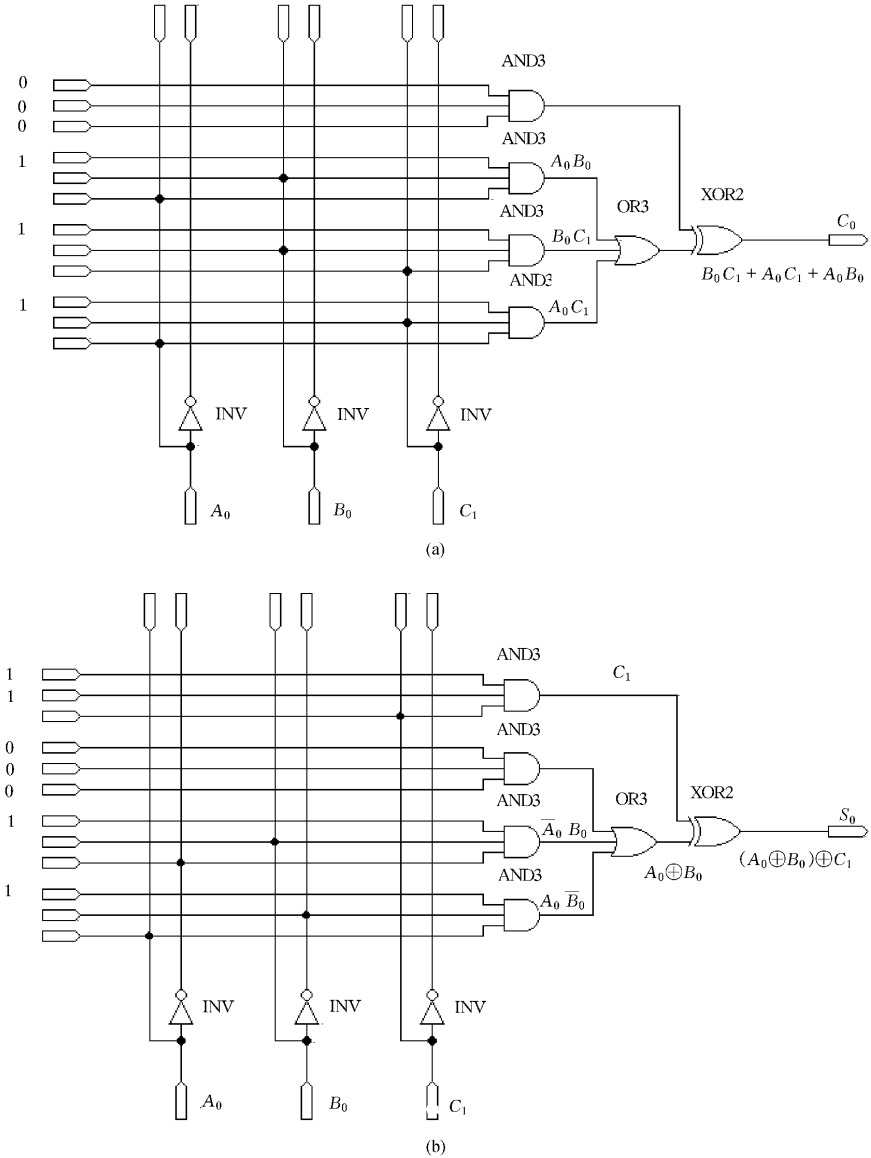


图 1－17　MAX5000 系列器件的逻辑单元实现一位全加器的原理图

(a) $S_0 = (A_0 \oplus B_0) \oplus C_1$ ；(b) $C_0 = B_0C_1 + A_0C_1 + A_0B_0$

1.3 EDA 概述

EDA(Electronic Design Automation,即电子设计自动化)已经逐渐成为电子电路与系统的重要设计手段,目前广泛用于模拟与数字电路系统等许多领域。EDA 研究的对象是电子设计的全过程。从某一角度看,电子设计可分为三个层次,即系统级、电路级和物理实现级。从另一个角度看,EDA 应包括电子线路领域中从低频到高频直至微波、从线性到非线性、从模拟到数字、从分离元件到集成电路的全部设计过程。

EDA 技术从 70 年代兴起,已经历了三个阶段。70 年代是以电子电路 CAD 的 PCB(印刷电路板)布线工具为代表。80 年代是以数字电路分析为代表,主要解决没有完成设计之前的功能检验问题。这时的 PCB 布线工具已经具有电路分析与设计功能。它将特性驱动的概念与分析仿真工具相结合,脱离了只是代替人工布线的概念,使得布出的 PCB 板不单是布通,而且必须符合电路的功能和特性要求。到了 90 年代,随着 TOP-DOWN DESIGN(自顶向下设计方法)的提出和 DSP(数字信号处理)技术的发展,逻辑综合工具和 DSP 设计工具应用的普及,数字、模拟和数模混合电子系统的仿真设计和 PCB 制板前的系统硬件电路仿真分析与试验(FPCB)技术的进展,为缩短电子系统设计周期的竞争又促使 CE(Concurrent Engineering,即并行设计工程)和 DM(Design Management,即设计管理系统)的应用得到迅速发展。从而促进电子设计人员和管理人员要求所有工具(包括系统仿真、PCB 布线、逻辑综合、DSP、FPCB、MEM 等)必须在一个面向用户的统一的数据库及管理框架环境下工作,并已达到最佳效果。因此 EDA 技术发展到 90 年代,它是由 TOP—DOWN DESIGN 新一代设计概念和功能,PCB 布线和设计可制造性分析,数字、模拟混合信号电子系统的分析和设计仿真,DSP 设计等软件包与面向用户的统一的数据库及管理框架共同组合而成。这种 EDA 系统主要以并行设计工程的方式和系统级目标设计方法作为支持。为此,人们又把 90 年代发展起来的 EDA 称作 ESDA(Electronic System Design Automation)。一般的 ESDA 系统级目标设计方法的软件系统框架如图 1－18 所示。ESDA 软件集成系统包含如下软件包:

①具有面向用户的统一的数据库及管理框架环境;

②数字和模拟混合信号电路系统混合级仿真器,包括电路系统拓扑结构、模型库和器件库;

③TOP—DOWN 设计包括行为级描述(支持 VHDL,即 Very High Speed Intergrated Circuit Hardware Description Language,指数字的超高速集成电路硬件描述语言;AHDL,即 Analog Hardware Description Language,指模拟电路的硬件描述语言)、逻辑综合、时序分析、厂家布局布线器、模型库、器件库和工艺规则,主要完成对 ASIC、PLD 和 FPGA 的设计;

④PCB 设计,包括自动布线器、约束条件、热分析、可制造性分析等;

⑤DSP(数字信号处理)软件包;

⑥电子系统机械构造与造型分析;

⑦可靠性分析。

在 ESDA 技术中,系统设计的核心是可编程器件设计。由于可编程器件自身的可重复编

程的特性，使电子设计的灵活性和工作效率大大提高。因此，本书不是对 ESDA 技术展开描述，只是对目前可编程器件的设计的全面介绍。

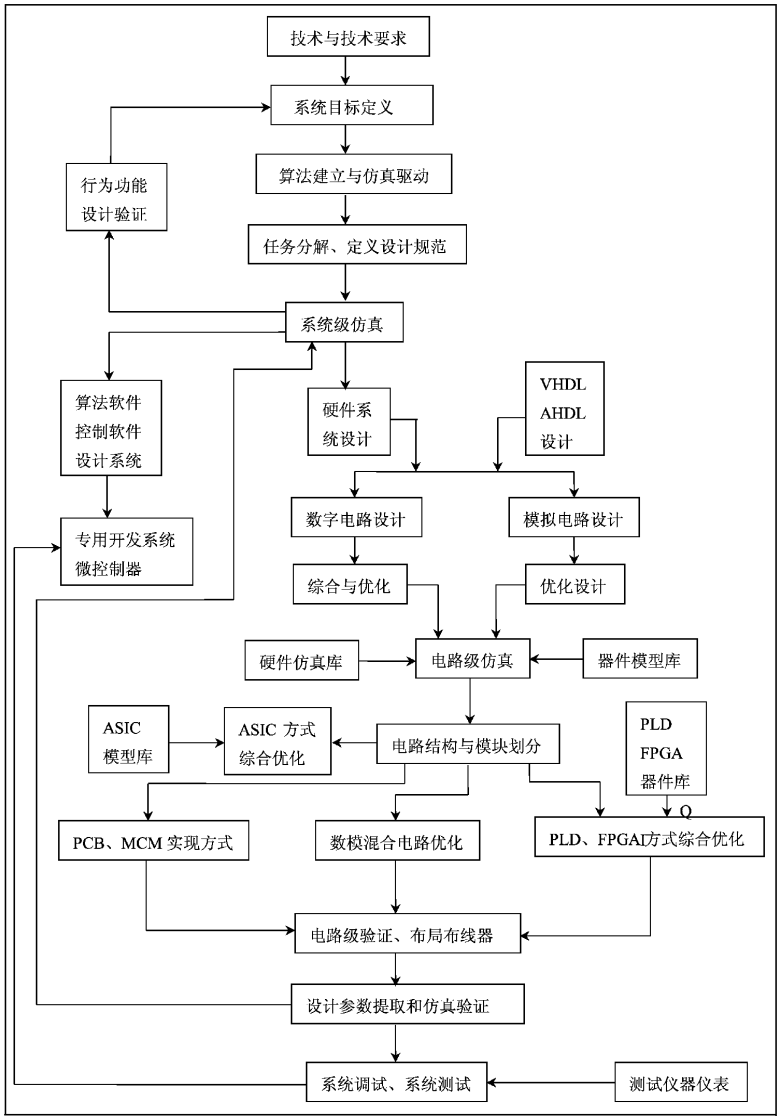


图 1－18　一般的 ESDA 软件系统框图

第二章　ALTERA 可编程逻辑器件

2.1　概　　述

美国 ALTERA 公司以雄厚的技术实力、独特的设计构思和功能齐全的芯片开发系统在激烈的市场竞争中脱颖而出，成为佼佼者。因此本书选择 ALTERA 可编程逻辑器件作为典型产品，说明可编程逻辑器件的构造、性能和设计方法。

美国 ALTERA 公司先后推出了以 Classic 系列（第一代）、MAX5000 系列（第二代）和 MAX7000（第三代）的 EPLD 产品。通过该公司的先进的芯片开发软件 MAX + PLUSⅡ（Multiple Array Matrix + Programmable Logic User SystemⅡ），用户可以任意对芯片进行编程、加密或用软件代替硬件，以满足自己的设计需要。但用户需要更改设计时，又可以方便地将以前的配置擦除，重新进行配置，从而大大提高了设计速度。

为满足更广泛的设计要求，ALTERA 公司又对 FPGA 器件进行了改进，推出了功能超过普通 FPGA 的 FLEX8000 系列。FLEX8000 吸取了 FPGA 容量大、修改快的优点，又通过专用快速通道的设计，弥补了 FPGA 延时不可预测的缺陷，使得 FLEX8000 成为一种优秀的可编程逻辑器件。随后又在 MAX7000 的基础上，结合 FLEX8000 的结构推出了 MAX9000 系列高容量的 EPLD 产品。此后，推出了 FLEX10K 系列产品，采用 0.5μm CMOS SRAM 工艺规程，使器件内门数最高达 10 万。1997 年底又推出了采用 0.25μm CMOS ROM 工艺规程的结构性能优良、密度更高的 FLEX10K 系列的 EPF10K250 器件产品，片内门数已达 25 万。

ALTERA 器件结构的演变过程如图 2－1 所示。

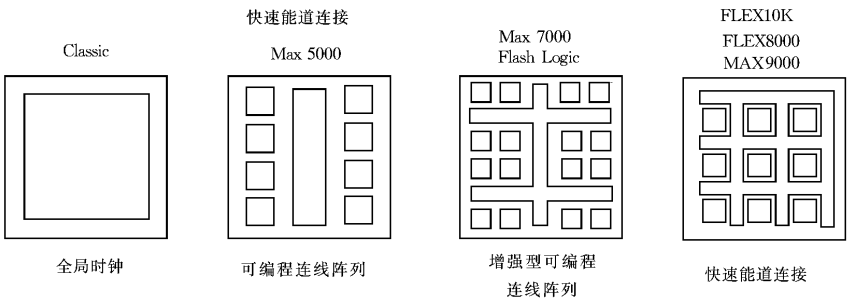


图 2－1　ALTERA 器件结构的演变过程示意图

Classic 类型的集成度较低，采用全局连线，可用门为 900 个，引脚可达 68 个，基于

EPROM 工艺并可用紫外光擦除和多次编程；MAX5000 系列采用可编程连线阵列(Programmable Interconnect Array)，可用门数为 300～3800 个、引脚达 20～100 个。因采用基于 EPROM 工艺，编程信息不会丢失，并可用紫外光擦除和多次编程；MAX7000 系列采用增强型可编程连线阵列(EPIA—Enhanced Programmable Interconnet Array)，可用门为 600～5000 个，还具有 32～256 个宏单元和 36～164 个用户 I/O 引脚，采用 EPROM 工艺，编程信息不会丢失，属于电擦除器件；FLEX8000 系列采用快速通道(Fast Track)连接结构，可用门为 2500～50 000 个，采用 SRAM 工艺，使其维持状态的功耗很低，并可进行在线重新配置；MAX9000 和 FLEX10K 系列也采用了快速通道的连线结构，并且可用门数和引脚数急剧增加。图 2－1 中的小方块是一个逻辑阵列块(LAB—Logic Array Block)，长条块是内部连线通道。表 2－1 示出 ALTERA 器件结构和工艺的特点。

表 2－1　ALTERA 器件的结构和工艺

器件系列	逻辑单元结构	连线结构	工　艺
Classic	积之和	连续式	EPROM
MAX5000	积之和	连续式	EPROM
FLASHLOGIC	积之和	连续式	SRAM&FLASH
MAX7000	积之和	连续式	EPROM
MAX9000	积之和	连续式	EPROM
FLEX8000	查找表	连续式	SRAM
FLEX10K	查找表	连续式	SRAM

图 2－2 示出 ALTERA 公司生产的器件系列产品的引脚数与集成度的发展趋势。

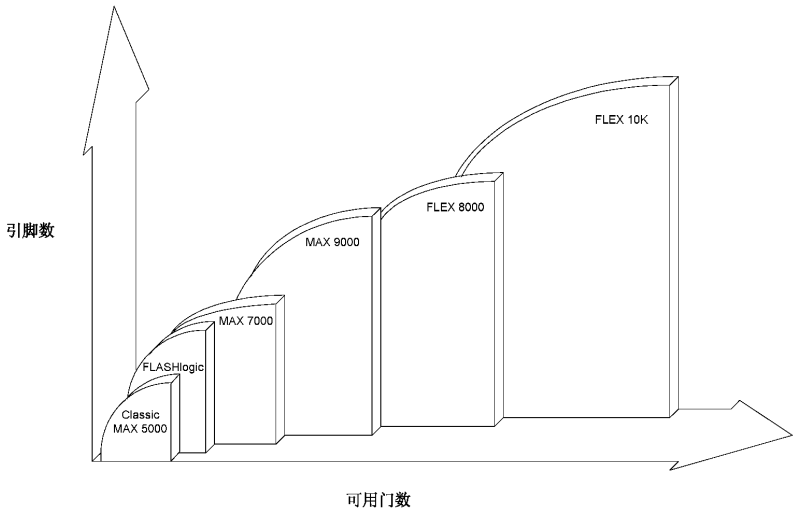
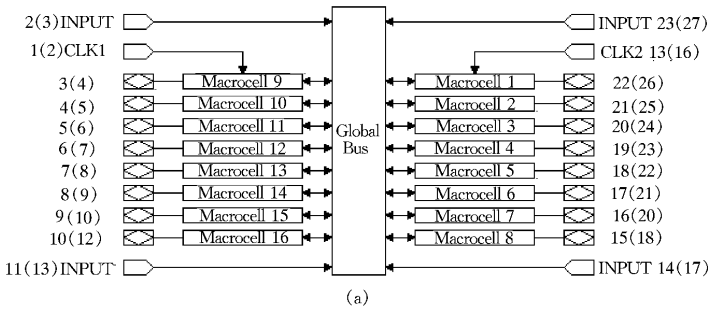


图 2－2　ALTERA 器件系列引脚数与集成度的发展趋势

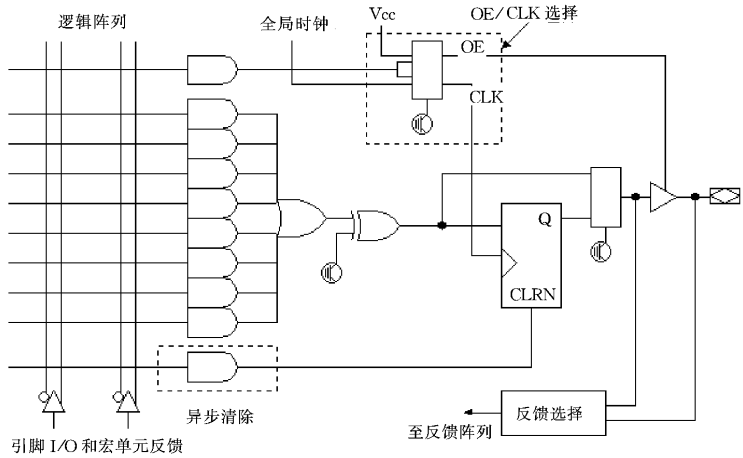
本节用到的 MAX 是多阵列矩阵英文缩写，英文全名为 Multiple Array Matrix；FLEX 是灵活逻辑单元矩阵英文缩写，英文全名为 Flexible Logic Element Matrix。

2.2 各类 ALTERA 器件的基本结构

为了对照了解各类 ALTERA **器件的结构特点，图** 2－3、**图** 2－4、**图** 2－5、**图** 2－6、**图** 2－7、**图** 2－8 **分别示出** Classic、MAX5000、MAX7000、FLEX8000、MAX9000 **和** FLEX10K **的结构方块图。**



(a)



(b)

图 2－3　Classic **类器件宏单元**

(a) Classic **类型的** EP610 **方块图；(b)** Classic **类器件宏单元**

ALTERA **公司的** Classic **系列器件是高速、低功耗的逻辑集成器件。它采用先进的** 0.8μm CMOS **工艺制造，有高速‘**T**’型和高性能‘**A**’型之分。图** 2－3 **示出** Classic **系列器件的基本结**

构,它由宏单元、可编程寄存器、输出使能/时钟选择和反馈选择几部分组成。采用‘积之和’逻辑,提供可编程的‘与’阵和固定的‘或’阵结构,可实现 8 个乘积项逻辑。宏单元是其最基本的模块,能独立地编程为 D 触发器、T 触发器、SR 触发器或 JK 触发器工作方式或组合逻辑工作方式。宏单元的寄存器还可以独立地由全局时钟或‘与’阵的反馈路径上的任何输入进行钟控(除 EP330 以外)。可编程 I/O 结构允许设计者编程输出,并具有反馈路径,使其按高电平有效或低电平有效工作在组合方式或寄存器方式,从而使得器件能够同时实现各种逻辑函数。

许多 Classic EPLD 还包含一个可编程 Turbo 位,以自动控制功率消耗,另外还有一个可编程的保密位。该保密位控制能否读出器件内的编程数据。当保密位被编程时,器件实现的专利不能被复制和取出。

MAX5000 系列器件采用先进的 0.8μm CMOS EPROM 工艺制造,MAX5000A 采用 0.65μm CMOS EPROM 工艺制造。两者具有相同的结构,其中包括逻辑阵列块(LAB)、宏单元、时钟逻辑、扩展乘积项、可编程连线阵列(Programmable Interconnect Array,即 PIA)和 I/O 控制块。它的结构特点是不仅有单个逻辑阵列块的器件,而且还有多个逻辑阵列块的器件。多个逻辑阵列块采用可编程连线阵列(PIA)连接在一起。PIA 起所有 I/O 引脚和宏单元馈送信号全局总线的作用。宏单元是实现逻辑的主要资源,从扩展项可得到附加的逻辑能力。扩展项能用来扩充任何宏单元。扩展乘积项阵列由未分配的反相的乘积项集中在一起组成。这些乘积项可由 LAB 内所有宏单元共享并用以构成组成逻辑和时序逻辑。在 LAB 内的宏单元通过 LAB 内的连线将各个宏单元在 LAB 内相互连接在一起,宏单元的输出也馈到 PIA,为各种多输入设计提供不同的布线通道。宏单元的输出都是全局布线。宏单元的输出还反馈到 I/O 控制块。I/O 控制块由可编程三态缓冲器和 I/O 引脚组成,如图 2-4(a)所示。

由图 2-4(b)可知,MAX5000 的宏单元是由可编程逻辑阵列和可独立配置的寄存器组成。寄存器通过编程可仿真 D 触发器、T 触发器、SR 触发器或 JK 触发器,也可以旁路寄存器实现纯组合逻辑。组合逻辑由可编程逻辑阵列实现。逻辑阵列的三个乘积项‘或’在一起后输出到‘异或’门的一个输入端,另一个输入端由单个乘积项控制,以实现高电平有效或低电平有效的逻辑。“异或”门也用于复杂“异或”运算的逻辑函数(如算术运算)以及狄·摩根反演律。“异或”门的输出送到可编程寄存器或旁路寄存器,实现组合逻辑。对于更为复杂的逻辑函数,MAX5000 不是用附加宏单元,而是采用共享的扩展乘积项。它可以直接送到 LAB 中的任何宏单元,而且保证在进行逻辑综合时,用尽可能少的逻辑资源得到尽可能快的工作速度。

在 MAX5000 中采用了一个可编程连线阵列(PIA)进行各个 LAB 之间信号的布线,且只对 LAB 中实现逻辑所需要的信号进行布线。PIA 由各宏单元和 I/O 引脚反馈给信号。值得一提的是,在掩膜编程或现场编程的门阵列(FPGA)中所使用的布线方式中,布线延时是累积的、可变的,并且和路径有关,而 MAX5000 则具有固定的延时。由此可知,PIA 的使用消除了信号之间的偏移,使延时可以预测。

MAX5000 中的每一个 LAB 都有一个 I/O 控制块。此块将每一个 I/O 引脚独立地配置成输入、输出或双向工作。I/O 控制块由宏单元阵列馈送信号。专用宏单元乘积项用来控制三态缓冲器。三态缓冲器再驱动 I/O 压焊点。

对于时钟选项,每个 LAB 皆支持全局时钟或阵列时钟控制。全局时钟由专用时钟信号(CLK)提供,并且时钟到输出的延时最短。在阵列时钟模式中,每个触发器由乘积项钟控。任何输入引脚或内部逻辑都可以用来作时钟源。阵列钟控允许每个触发器配置成正边沿触发型

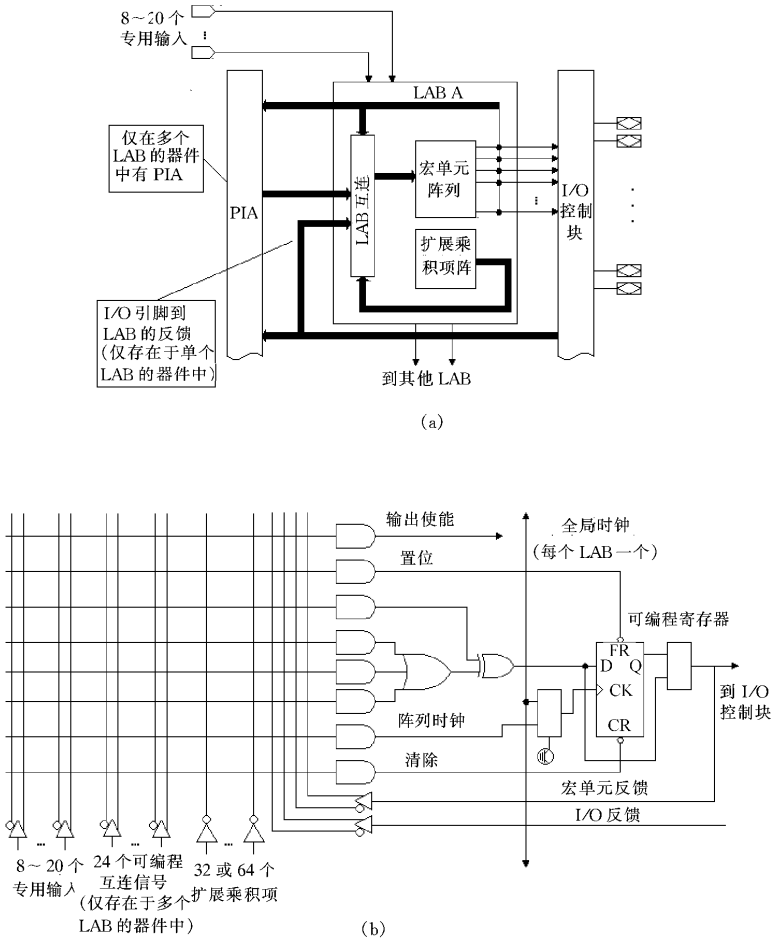


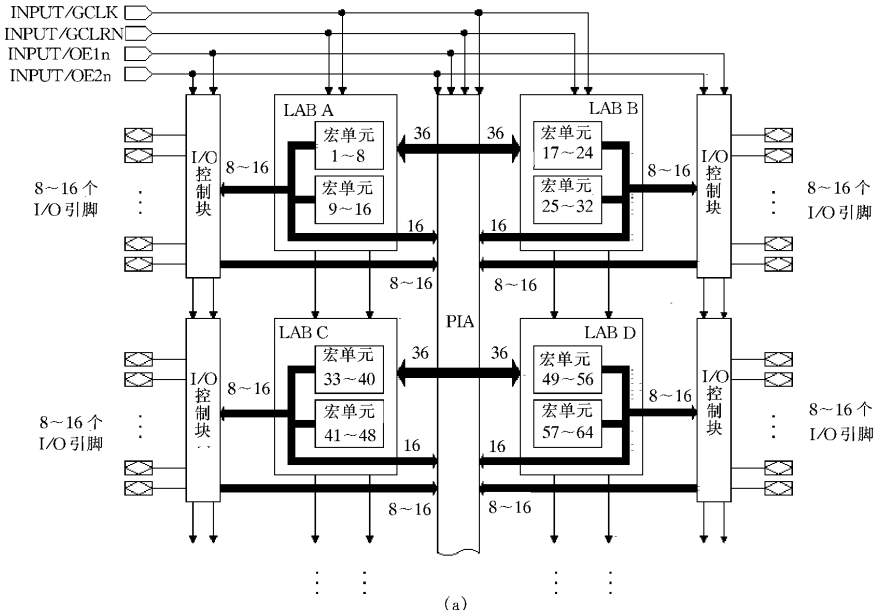
图 2－4　MAX5000 系列器件基本结构

(a)MAX5000 方块图；(b)MAX5000 宏单元

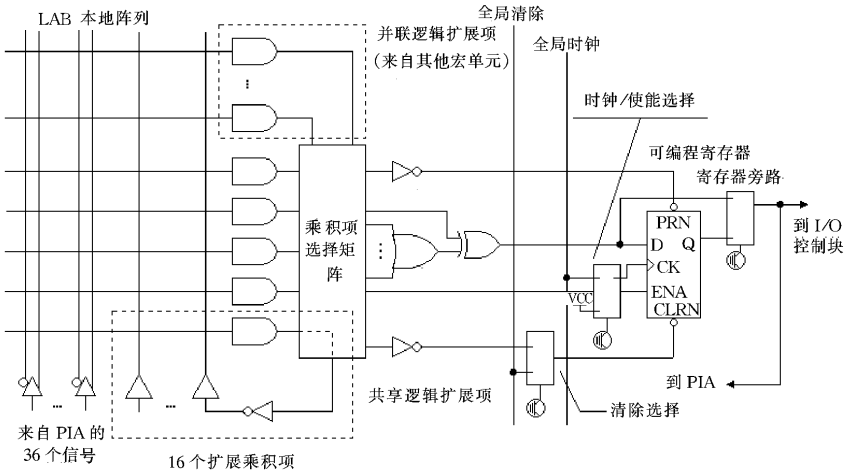
或负边沿触发型。显然，这给宏单元增加了灵活性。但是应该注意的是，尽管在 LAB **中的每个触发器可由阵列产生不同时钟来钟控，然而全局时钟和阵列时钟两种模式不能在一个** LAB **中混合使用。**

另外，MAX5000 EPLD **也包含一个可编程的保密位。**

MAX7000 **系列器件采用先进的** 0.8μm CMOS EPROM **技术制造。它是高密度、高性能的** CMOS EPLD **器件。图** 2－5(a)、(b) **示出** EPM7032、EPM7032V、EPM7064 **和** EPM7096 **的结构方框图和其中的宏单元结构。图** 2.5(c)、(d) **示出了** MAX7000E **和** MAX7000S **器件的结构方框图和其中的宏单元结构。**


INPUT/GCLK
INPUT/GCLRN
INPUT/OE1n
INPUT/OE2n
LAB A
LAB B
LAB C
LAB D
宏单元 1～8
宏单元 9～16
宏单元 17～24
宏单元 25～32
宏单元 33～40
宏单元 41～48
宏单元 49～56
宏单元 57～64
I/O 控制块
8～16 个 I/O 引脚
36
16
8～16
PIA

(a)


LAB 本地阵列
全局清除
并联逻辑扩展项
(来自其他宏单元)
全局时钟
时钟/使能选择
可编程寄存器
寄存器旁路
乘积项选择矩阵
PRN
D Q
CK
ENA
CLRN
VCC
到 I/O 控制块
到 PIA
共享逻辑扩展项
清除选择
来自 PIA 的 36 个信号
16 个扩展乘积项

(b)

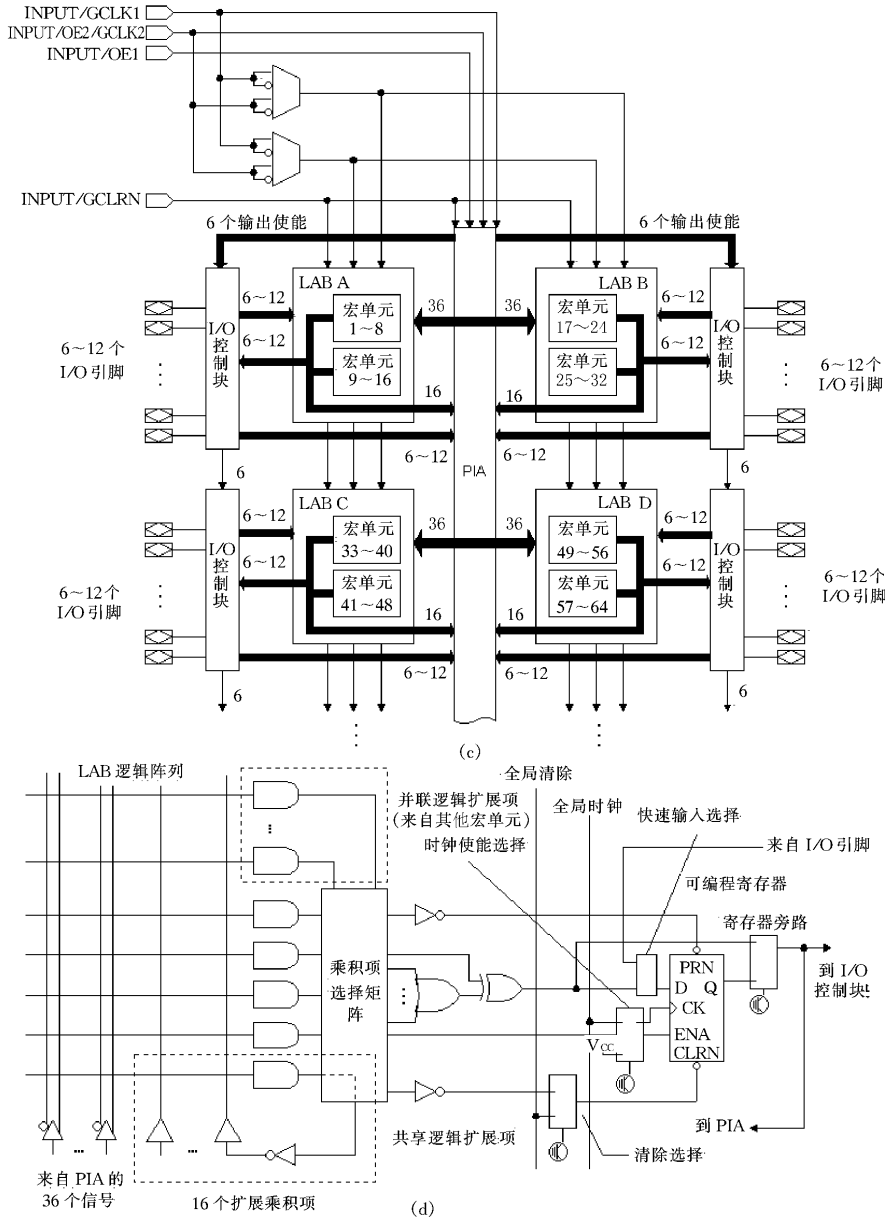


图 2－5　MAX7000 系列器件的基本结构

(a)EPM7032、EPM7032V、EPM7064 和 EPM7069 器件结构方块图；(b)EPM7032、EPM7032V、EPM7064 和 EPM7069 器件的宏单元；(c)MAX7000E & MAX7000S 器件方块图；(d)MAX7000E & MAX7000S 器件的宏单元

MAX7000 系列器件中有四个专用输入。它们可以作为通用输入,也可以作为每个宏单元和 I/O 引脚的高速、全局控制信号,如时钟(Clock)、清除(Clear)和输出使能(Output Enable)等。

MAX7000 的逻辑阵列块(LAB)由 16 个宏单元组成,多个 LAB 通过可编程连线阵列(PIA)和全局总线连在一起。全局总线由所有的专用输入、I/O 引脚和宏单元馈给信号。LAB 的输入信号有:①来自通用逻辑输入的 PIA 的 36 个信号;②用于寄存器辅助功能的全局控制信号;③从 I/O 引脚到寄存器的直接输入通道,用于实现高密度的 MAX7000E 器件的快速建立时间。

MAX7000 的宏单元由逻辑阵列、乘积项选择矩阵和可编程触发器三个功能块组成。EPM7032、EPM7032V、EPM7064 和 EPM7096 器件的宏单元如图 2-5(b)所示。MAX7000E 系列包括 EPM7128E、EPM7160E、EPM7192E 和 EPM7256E,器件的宏单元如图 2-5(d)所示。

在 MAX7000 中逻辑阵列实现组合逻辑,给每个宏单元提供 5 个乘积项。"乘积项选择矩阵"分配这些乘积项作为到"或"门和"异或"门的主要逻辑输入,以实现组合逻辑函数,或者把这些乘积项作为宏单元中触发器的辅助输入,即清除(Clear)、置位(Preset)、时钟(Clock)和时钟使能(Clock Enable)控制。每个宏单元的一个乘积项可以反相后回送到逻辑阵列。这个"可共享"的乘积项能够连接到同一个 LAB 中任何其他乘积项上。根据设计的逻辑需要,MAX+PLUSⅡ开发系统将自动优化乘积项的分配。

作为寄存器功能,每个宏单元的触发器可以单独地编程为具有可编程时钟控制的 D 触发器、T 触发器、SR 触发器或 JK 触发器。另外,只要需要,也可将触发器旁路,实现组合逻辑工作方式。每一个可编程的触发器可以按三种不同的方式实现钟控:①全局时钟信号方式,它可以使时钟到输出最快;②全局时钟信号并由高电平有效的时钟信号所使能,这种方式可提供每个触发器的使能信号,并使全局时钟到输出最快;③用乘积项实现阵列时钟,在这种方式中,触发器是由来自隐埋的宏单元或 I/O 引脚的信号进行钟控。在图 2-5(a)中器件的全局时钟(Clock)信号是由专用时钟引脚 GCLK 提供。在图 2-5(c)中器件可以有两个全局时钟信号,一种是全局时钟引脚 GCLK1 和 GCLK2 信号,另一种是 GCLK1 和 GCLK2 求"反"后的信号。

每个触发器也支持异步清除和异步置位功能,乘积项选择矩阵分配乘积项去控制这些操作。虽然乘积项驱动触发器的置位和复位信号是高电平有效,但是,在逻辑阵列中将信号反相可得到低电平有效的控制。此外,每一个触发器的复位功能可以由低电平有效的、专用的全局复位引脚 GCLRn 信号提供。

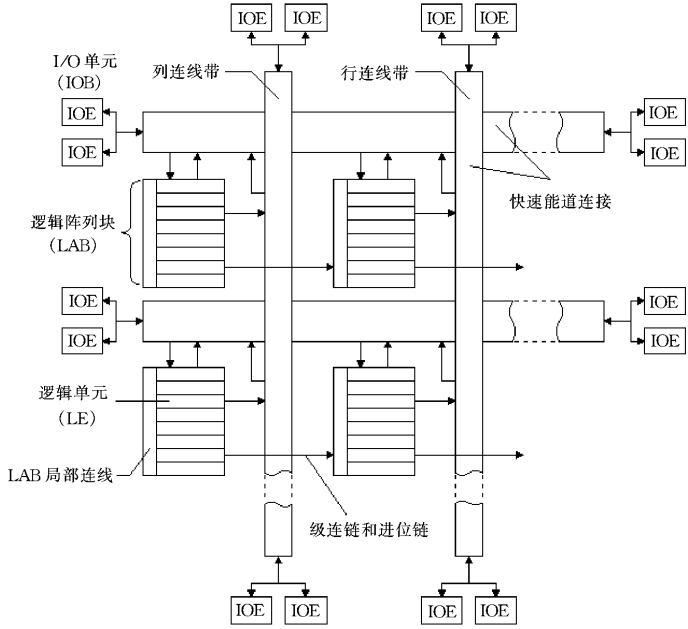
所有与 I/O 引脚相联系的 MAX7000E 的宏单元都具有快速输入功能。这些宏单元的触发器有的直接来自 I/O 引脚的输入通道,它旁路了 PIA 和组合逻辑。这些直接输入通道允许触发器作为具有快速输入建立时间(3ns)的输入寄存器。

与 MAX5000 的结构不同,MAX7000 的结构中还有另一类扩展乘积项。共享逻辑阵列块(LAB)的任意宏单元可以完成某些更为复杂、需要附加乘积项的逻辑函数。在实现逻辑综合时,利用扩展项可以保证用尽可能少的逻辑资源得到尽可能快的工作速度。

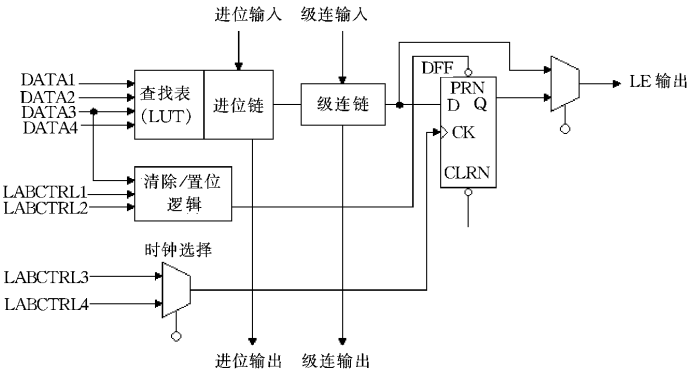
MAX7000 中有可编程连线阵列(PIA)、I/O 控制,并且编程具有保密性,另外还增加了可编程速度/功率控制和电压摆率的控制。

图 2-6 示出 FLEX8000 系列器件的基本结构。与 MAX5000、MAX7000、Classic 系列器

件不同的是，它引入了一种逻辑单元（LE—Logic Element）大矩阵，每 8 个组成一组构成一个逻辑阵列块，即 LAB。每个 LAB 都是独立的，并具有相同的结构。它们具有相同的输入、互连与控制信号。LAB 中每个 LE 都包含一个提供组合逻辑能力的四输入查找表（LUT），一个能



（a）



（b）

图 2－6　FLEX8000 系列器件的基本结构

（a）FLEX 8000 器件方块图；（b）FLEX 8000 逻辑单元（LE）方块图

提供时序逻辑能力的可编程寄存器、进位链、级连链。LE 的结构能有效地实现各种逻辑。LUT 是一个函数发生器,它能快速计算四个变量的任意函数,可编程触发器可设置成 D 触发器、T 触发器、SR 触发器或 JK 触发器。该触发器的时钟、清除和置位控制信号可由专用输入引脚或任何内部逻辑驱动。对于纯组合逻辑,可将触发器旁路,使 LUT 的输出直接连到 LE 的输出。器件内的 LAB 按行与列排列,位于行和列两端的输入/输出单元(IOE)提供 I/O 引脚。每个 IOE 有一个双向 I/O 缓冲器和一个既可作输入寄存器又可作输出寄存器的触发器。

FLEX8000 系列器件结构的另一个特点是,在器件内部信号的互连是由快速通道(Fast Track)连线提供的。它是贯穿器件长、宽的快速连线通道。快速通道由‘行连线带’和‘列连线带’组成。采用这种布线结构,即使对于复杂的设计也可以预测其性能,而 FPGA 中的分段式连线结构需要用一些开关矩阵把数目不同的若干线段连接起来,这就增加了逻辑资源间的延时,使机器性能下降。

MAX9000 系列器件的基本结构如图 2-7 所示,它包括逻辑阵列块、宏单元、扩展乘积项(共享和并联)、快速通道(Fast Track)和 I/O 单元。

逻辑阵列块(LAB)与 MAX7000 系列器件中的 LAB 类似,由 16 个宏单元组成。但各宏单元是由 LAB 局部阵列相互馈送信号。多个 LAB 又通过快速通道 Fast Track 连接在一起。这一点又与 FLEX8000 类似。快速通道 Fast Track 是遍历器件内全部长度和宽度的一系列快速、连续式通路。快速通道 Fast Track 内的每一条‘行连线带’和‘列连线带’的两端都设有 I/O 单元(IOC),并且 IOC 又支持 I/O 引脚。LAB 直接驱动‘行连线带’和‘列连线带’。每个宏单元又能驱动 LAB 的输出到‘行连线带’和‘列连线带’。一旦信号被送到‘行连线带’或‘列连线带’上,就被快速送到其他 LAB 或 I/O 单元。

由结构特点可以看出,MAX9000 系列器件是在 MAX7000 和 FLEX8000 的基础上发展起来的新型 MAX 器件。

FLEX10K 系列器件的片内结构如图 2-8(a)所示,是在 FLEX8000 系列器件基础上发展起来的又一新型器件。它的结构的主要特点除主要的逻辑阵列块(LAB)之外,首次采用了嵌入阵列块(EAB)。

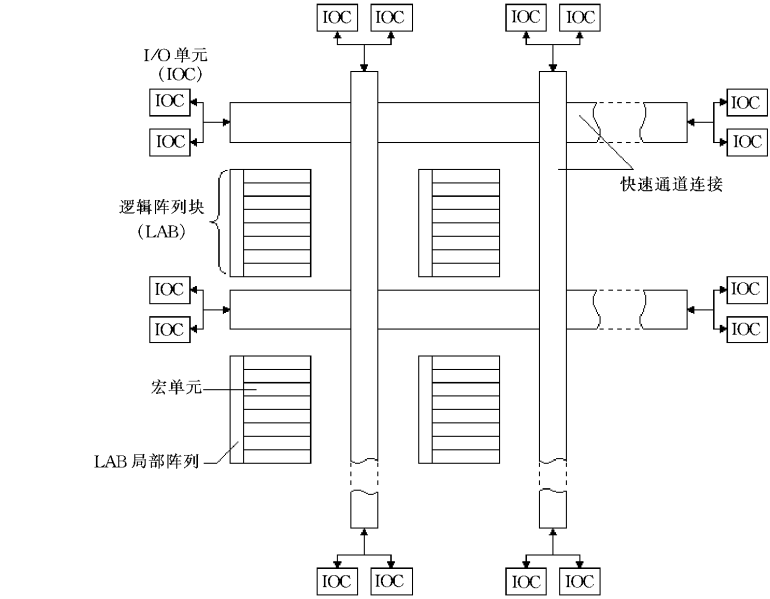
逻辑阵列块是由多个逻辑阵列块组成。每个阵列块包含 8 个逻辑单元(LE)和一个局部互连。一个 LE 又由四输入查找表(LUT)、一个可编程寄存器和专用的载运和级联功能的信号通道所组成。

在 FLEX10K 器件内信号的连接以及进和出器件的管脚,皆由快速通道互连提供。它的行和列通道遍布器件的全部长和宽。每个 I/O 管脚都是由一个处在每行和每列的快速通道末端的 I/O 单元(IOE)提供的。

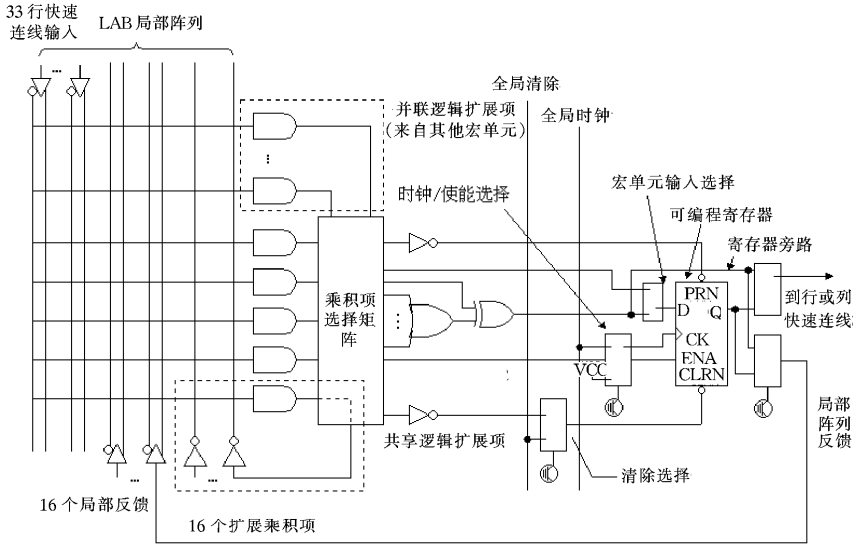
在 FLEX10K 器件中,把每一组逻辑单元(8 个 LE)组成一个逻辑阵列块(LAB),所有的逻辑阵列块(LAB)排成行和列。在一行里还包含一个单一的 EAB。多个 LAB 和多个 EAB 采用快速通道互相连接。

为保证有效地高速分配低摆率控制信号,在 FLEX10K 器件内还提供 6 个专用输入用来驱动触发器的输入控制。这些信号应用专用路线通道提供比快速通道还短的延迟、还低的摆率。四个专用输入驱动四个全局信号。这四个全局信号也可以全部用内部逻辑所驱动。

嵌入阵列块(EAB)是 FLEX10K 系列器件在结构设计上的一个重要部件。它是一个在输



（a）



（b）

图 2－7　MAX9000 系列器件的基本结构图

（a）MAX9000 器件方块图；（b）MAX9000 宏单元和局部阵列图

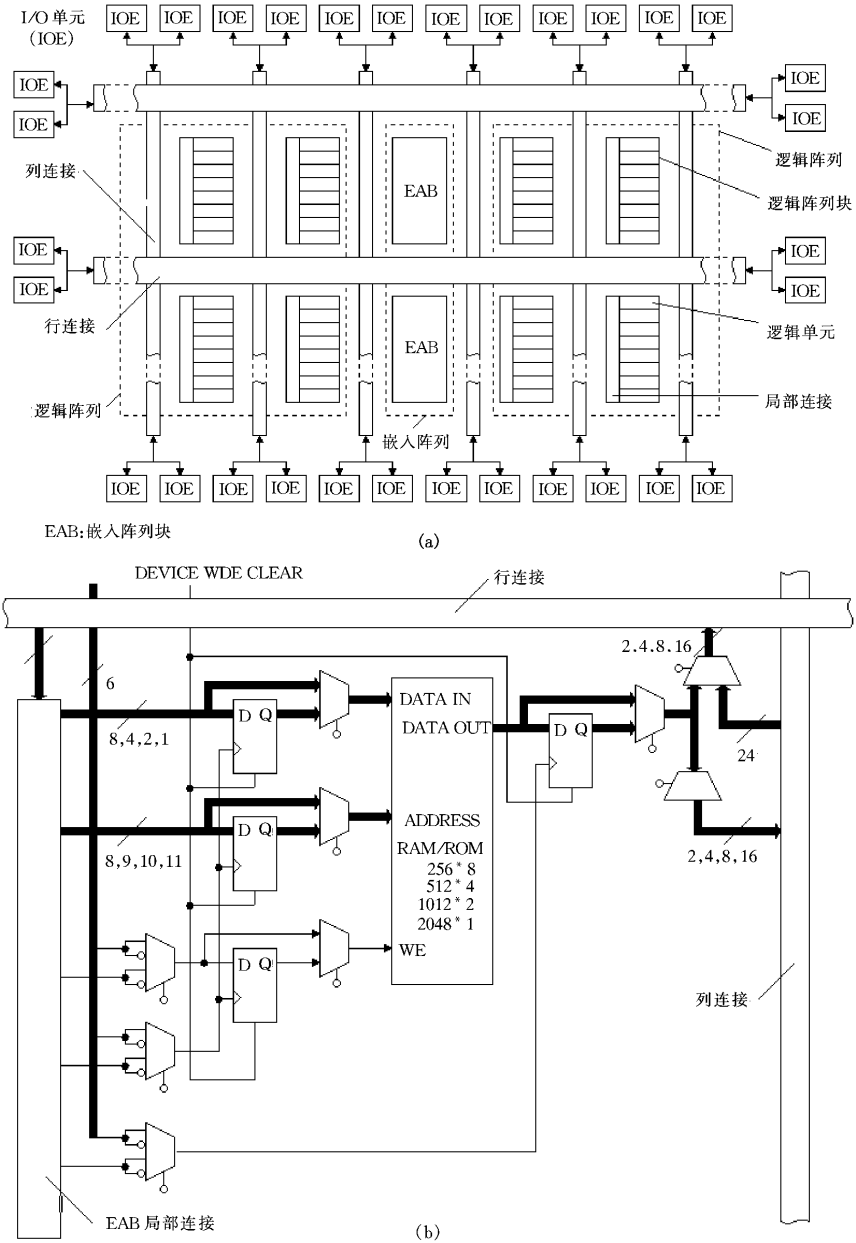


图 2－8　FLEX10K 系列器件的结构

(a)FLEX10K 器件方块图；(b)FLEX10K 嵌入阵列块(EAB)方块图

入端口和输出端口都带有寄存器的一种灵活的 RAM 块，其结构如图 2－8(b)所示。嵌入阵列块(EAB)组成的规模和灵活性只对比较多的内存是适宜的。它的功能包括乘法器、向量的标量和误差矫正电路等。在应用中，这些功能又能联合完成数字滤波器和微控制器的功能。采用可编程的带有只读平台的嵌入阵列块(EAB)，在配置期间可执行逻辑功能并建立一个大的查找表(LUT)，在这个查找表里用查找的结果执行组合逻辑函数，而不用计算它们。显然，用这种组合逻辑函数执行比通常在逻辑里应用算法执行要快，而且嵌入阵列块(EAB)的大容量使得设计者为了实现复杂逻辑，在一个逻辑级上没有布线延迟地连接到联合的逻辑单元(LE)或 FPGA RAM 块上成为可能。例如，一个单一的 EAB 使用 8 个输入和 8 个输出能够实现一个 4×4 的乘法器。

EAB 具有的优点超出 FPGA，它执行片上的 RAM 块。这种 FPGA RAM 块包含延迟比 RAM 增加的规格的预测置还小。另外，FPGA RAM 块有出现布线问题的倾向，因为小的 RAM 必须与组成较大的块连在一起。专用 EAB 容易应用，并且快速提供可能预测的延迟。

可以用 EAB 执行同步 RAM，对应用来说它比异步 RAM 容易。电路采用异步 RAM 必须产生 RAM 的写使能(WE)信号，在此期间保证他们的数据和地址信号满足相对于写使能(WE)信号的建立和保持时间的规定。对比之下，EAB 同步 RAM 产生它自己的使能(WE)信号，并且就全局时钟而言它又是自定时的。电路采用 EAB 的自定时 RAM 只需要满足全局时钟的建立和保持时间。

当采用这样的 RAM 时，可以按下面的任意形式配置(图 2－9)：

256×8，512×4，1024×2，2048×1

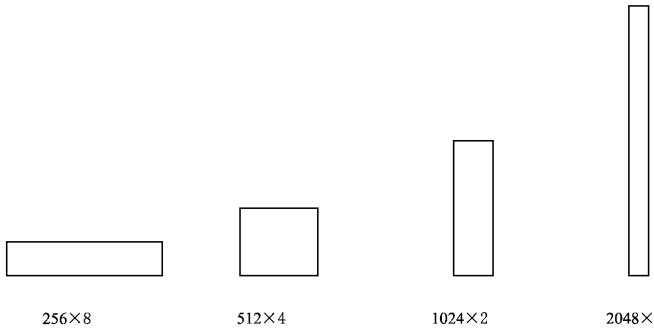


图 2－9　EAB 内存配置

通过连接多个 EAB 可建立较大的 RAM 块。例如，两个 256×8 RAM 可以组成一个 256×16 的 RAM 形式，两个 512×4 的 RAM 可以组成一个 512×8 的 RAM。如果需要，在器件内的所有 EAB 可以级联成一个单一 RAM 的形式。在不影响计时的情况下，可以把 EAB 级联成达到 2 048 字节的 RAM 形式。ALTERA 公司的 MAX＋PLUSⅡ软件自动组合 EAB 去执行设计人员的规定，请参看图 2－10。

为了驱动和控制时钟信号，EAB 提供了灵活的选择。对于 EAB 的各种输入和输出可以采用不同的时钟。在数据输入、EAB 输出或者地址和写使能(WE)信号处可以把寄存器独立地插入。全局信号和 EAB 局部连接可以驱动写使能(WE)信号。全局信号、专用时钟管脚和

EAB 局部互连可以驱动 EAB 时钟信号。由于各个 LE 驱动 EAB 局部互连，所以各个 LE 可以控制写使能(WE)信号或 EAB 时钟信号。

通过行的互连进入每个 EAB，并且每个 EAB 可以驱动到行和列的互联。每个 EAB 的输出可以驱动两个行通道中的任何一个和两个列通道中的任何一个。当采用的行通道时，可以用一个列通道驱动。对于 EAB 的各种输出，这种特点增加了布线资源的可利用性。

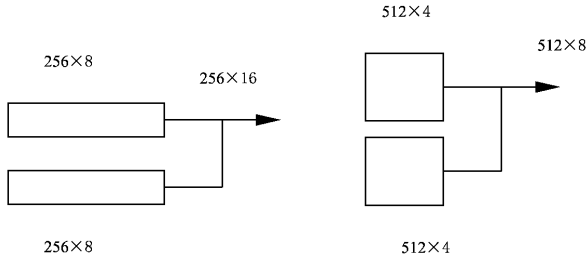


图 2－10　组合 EAB 的例子

2.3　各类 ALTERA 器件的特性指标

各类 ALTERA 器件的特性指标分别列于表 2－1～表 2－8 中，读者可以根据自己的设计需要选择，同时也可以通过下列各表了解各类 ALTERA 器件的基本性能。

表 2－1　Classic 器件特性

特　性	EP610 & EP6101	EP910 & EP9101	EP1810
提供的门数目	600	900	1,800
可用门数目	300	450	900
宏单元数目	16	24	48
最大用户 I/O 数目	22	38	64
t_{PP}(ns)	10	12	20
f_{CNT}(MHz)	100	76.9	50

表 2－2　MAX5000 器件特性

特　性	EPM5032	EPM5064	EPM5128	EPM5130	EPM5192
提供的门数	1,200	2,500	5,000	5,000	7,500
可用门数	600	1,250	2,500	2,500	3,750
宏单元数	32	64	128	128	192
LAB 数目	1	4	8	8	12
扩展项数目	64	128	256	256	384
布线	全局	PIA	PIA	PIA	PIA
最大用户 I/O 数目	24	36	60	68,84	72
t_{PP}(ns)	15	25	25	25	25
t_{ASU}(ns)	4	4	4	4	4
t_{CO}(ns)	10	14	14	14	14
f_{CNT}(MHz)	76.9	50	50	50	50

表 2－3　MAX7000 器件特性

特　　性	EPM7032 EPM7032V EPM7032S	EPM7064 EPM7064S	EPM7096 EPM7096S	EPM7128E EPM7128S EPM7128V	EPM7160E EPM7160S	EPM7192E EPM7192S	EPM7256E EPM7256S
提供的门数	1,200	2,500	3,600	5,000	6,400	7,500	10,000
可用的门数	600	1,250	1,800	2,500	3,200	3,750	5,000
宏单元数	32	64	96	128	160	192	256
LAB 数目	2	4	6	8	10	12	16
最大用户数	36	68	76	100	104	124	164
t_{PP}(ns)	5(12)	5	6	6(10)	6	7.5	7.5
t_{SU}(ns)	4(10)	4	5	5(7)	5	6	6
t_{FSU}(ns)	2.5(－)	2.5	2.5	2.5(3)	2.5	3	3
t_{COI}(ns)	3.5(7)	3.5	4	4(5)	4	4.5	4.5
f_{CNT} (MHz)	178.6 (90.9)	178.6	151.5	151.5 (100)	151.5	125	125

表 2－4　MAX9000 器件的特性

特　　性	EPM9320	EPM9400	EPM9480	EPM9560
提供的门数	12,000	16,000	20,000	24,000
可用门数	6,000	8,000	10,000	12,000
触发器数目	484	580	676	772
宏单元数目	320	400	480	560
LAB 数目	20	25	30	35
最大用户 I/O 数目	168	159	175	216
t_{PPI}(ns)	12	12	12	12
t_{FSU}(ns)	4	4	4	4
t_{FCO}(ns)	5.5	5.5	5.5	5.5
t_{CNT}(MHz)	125	125	118	118

表 2－5　FLEX8000 器件特性

特　　性	EPF82282A EPF8282AV	EPF8452A	EPF8636A	EPF8820A	EPF81188A	EPF1500A
提供的门数	5,000	8,000	12,000	16,000	24,000	32,000
可用门数	2,500	4,000	6,000	8,000	12,000	16,000
触发器数目	282	452	636	820	1,188	1,500
LAB 数目	26	42	63	84	126	162
LE 数目	208	336	504	672	1,008	1,296
最大用户 I/O 引脚数目	78	120	136	152	184	208
JTAG BST 电路	Yes	No	Yes	Yes	No	Yes

表 2－6　FLEX8000 器件性能

应　　用	LES 应用	A－2 速度级	A－3 速度级	A－4 速度级	单位
16 位加载计数器	16	125	95	83	MHz
16 位加/减计数器	16	125	95	83	MHz
16 位预计计数器	24	270	232	185	MHz
24 位累加器	24	87	67	58	MHz
16 位地址译码器	4	4.2	4.9	6.3	ns
16 选 1 的多路选择器	10	6.6	7.9	9.5	ns

表 2－7　FLEX10K 器件特性(一)

器　　件	门　　数	最大 I/O 引脚数	速度级别	触发器数目	LE 数目	总 RAM bits
EPF10K10	10,000	134	－3,－4	720	576	6,144
EPF10K10A	10,000	134	－1,－2,－3	720	576	6,144
EPF10K20	20,000	189	－3,－4	1,344	1,152	12,288
EPF10K30	30,000	246	－3,－4	1,968	1,728	12,288
EPF10K30A	30,000	189	－1,－2,－3	1,968	1,728	12,288
EPF10K30B	30,000	189	－1,－2,－3	1,968	1,728	12,288
EPF10K30	30,000	246	－3,－4	1,968	1,728	12,288
EPF10K30A	30,000	189	－1,－2,－3	1,968	1,728	12,288
EPF10K30B	30,000	189	－1,－2,－3	1,968	1,728	12,288
EPF10K40	40,000	189	－3,－4	2,576	2,304	16,384
EPF10K50	50,000	310	－3,－4	3,184	2,880	20,480
EPF10K50V	50,000	274	－1,－2,－3,－4	3,184	2,880	20,480
EPF10K50B	50,000	274	－1,－2,－3	3,184	2,880	20,480
EPF10K70	70,000	358	－2,－3,－4	4,096	3,744	18,432
EPF10K100	100,000	406	－3,－4	5,392	4,992	24,576
EPF10K100A	100,000	406	－1,－2,－3	5,392	4,992	24,576
EPF10K100B	100,000	406	－1,－2,－3	5,392	4,992	24,576
EPF10K130V	130,000	470	－2,－3,－4	7,120	6,656	32,768
EPF10K130B	130,000	470	－1,－2,－3	7,120	6,656	32,768
EPF10K180B	180,000	470	－1,－2,－3	10,534	9,728	32,768
EPF10K250A	250,000	470	－1,－2,－3	12,624	12,160	40,960
EPF10K250B	250,000	470	－1,－2,－3	12,624	12,160	40,960

表 2－7　FLEX10KE 器件特性(二)

特　　性	EPF10K30E	EPF10K50E	EPF10K100E	EPF10K130E	EPF10K200E	EPF10K250E
典型门数目 (Logic&RAM)	30,000	50,000	100,000	130,000	200,000	250,000
可用门数目	22,000～ 119,000	36,000～ 199,000	62,000～ 257,000	82,000～ 342,000	123,000～ 513,000	149,000～ 474,000
LE 数目	1,728	2,880	4,992	6,656	9,984	12,160

续表

特　性	EPF10K30E	EPF10K50E	EPF10K100E	EPF10K130E	EPF10K200E	EPF10K250E
EAB 数目	6	10	12	16	24	20
总 RAM bits	24,576	40,960	49,152	65,656	98,304	81,920
封装方式	144 脚 TQFP 208 脚 PQFP 240 脚 PQFP 256 脚 BGA	144 脚 TQFP 208 脚 PQFP 240 脚 PQFP 256 脚 BGA 484 脚 BGA	208 脚 PQFP 240 脚 PQFP 484 脚 BGA	240 脚 PQFP 484 脚 BGA 676 脚 BGA	240 脚 RQFP 484 脚 BGA 676 脚 BGA	240 脚 RQFP 484 脚 BGA 676 脚 BGA

表 2－8　FLEX10K 器件特性

应　用	资源利用		性　能			单位
	LEs	EABs	－3 速度级	－4 速度级	－5 速度级，注(1)	MHz
16 位可加载计数器	16	0	104	97	67	MHz
16 位累加器	16	0	104	97	67	MHz
16 选 1 多路选择器②	10	0	9.4	10.6	13.2	ns
4×4 乘法器③	0	1	105	86	66	MHz
8×8 乘法器③	25	4	30	24	18	MHz
256×8 RAM③	0	1	105	86	66	MHz

注：①－5 速度级只可用于 EPF10K50。

②这种应用采用联合的输入和输出。

③这种应用采用寄存器的输入和输出。

第三章　MAX + PLUS Ⅱ开发工具

本章共两部分。第一部分简要介绍 MAX + PLUS Ⅱ的安装及设计过程;第二部分以一个较简单的设计为例,详细介绍 MAX + PLUS Ⅱ的使用方法及设计技巧。

MAX + PLUS Ⅱ是开发 ALTERA 公司 FPGA 产品(包括 MAX 系列和 FLEX 系列)的软件工具。利用 MAX + PLUS Ⅱ提供的设计环境和设计工具,可以灵活高效地完成各种数字电路设计。本文中各种图例均是在 MAX + PLUS Ⅱ 7.0(单机版)下运行的结果,操作系统为 Windows 95。在其他操作系统下,图例会有差异。

3.1　MAX + PLUS Ⅱ简介

3.1.1　MAX + PLUS Ⅱ的安装

本小节主要讲述在 PC 机上安装 MAX + PLUS Ⅱ的几个注意事项。

(1)系统配置要求

①奔腾(推荐)或 486 PC 机;

②DOS 5.0 或更高版本;

③Microsoft Windows NT 3.5 以上版本或 Microsoft Windows 95,对于 Windows 3.1 需加载 Win32s;

④光盘驱动器(自 1996 年 1 月,MAX + PLUS Ⅱ只能通过光盘安装);

⑤双键鼠标及并行口;

⑥MAX + PLUS Ⅱ至少需要 110MB 的硬盘空间,建议 1.2GB 以上硬盘;

⑦编译不同器件的内存要求各不相同,见表 3 - 1。

表 3 - 1　内存配置要求

器　件	最小可用内存	最小物理内存
MAX 7000	32 MB	16 MB
MAX 9000	64 MB	32 MB
FLEX 8000	64 MB	32 MB
FLEX 10K	256 MB	128 MB

利用 Windows 创建永久性交换文件,可以增加虚拟内存的容量,从而增加可用内存的总量,具体方法可参照 Windows 使用手册。建议实际物理内存最好 32MB 以上。

(2)认证码

在第一次运行 MAX + PLUS Ⅱ时,需要键入认证码(Authorization Code)。未键入认证码或键入错误的认证码会使 MAX + PLUS Ⅱ的大部分功能不能使用。

(3)软件狗

有的版本或特殊的认证码不需要软件狗。若有软件狗,在运行 MAX + PLUS Ⅱ之前需将软件狗插接于 25 针并行口上,打印机还可以再接在软件狗上使用。

(4)关掉一些程序

若系统已运行以下防病毒软件,将使 MAX + PLUS Ⅱ的安装和使用出现意料不到的问题,在安装和使用 MAX + PLUS Ⅱ时最好关掉这些程序:

①由 MS-DOS 6.2 提供的 vsafe.com anti-virus software;

②由 Norton Utilities 提供的 Disk Protect anti-virus software。

(5)编程硬件的安装

有几类编程硬件。它们分别适用于各种不同的 FPGA 器件,如表 3 - 2 所示。

表 3 - 2 编程硬件分类表

工作平台	编 程 硬 件	应 用 范 围
PC	LP6 卡 + 主编程单元(MPU 包括适配器)	可对 MAX 系列器件编程
PC	FLEX 下载缆线 + LP6 + MPU + EPROM 适配器	可对 FLEX 系列器件编程
PC 或工作站	位霸器(BitBlaster)	可对 FLEX 系列及 MAX9000 编程

1)安装 LP6 卡 LP6 卡需安装在 PC 机的扩展槽内。卡上的 4 个开关决定了该卡的地址,一般默认为 1111,即开关都处于‘开’的位置。此时板卡的地址是 280H。若此地址与机内其他硬件地址冲突,可根据表 3 - 3 改变板卡的地址。

表 3 - 3 LP6 卡 I/O 地址表

I/O 地址	开关设置(1234)
270	0 0 0 0
260	1 0 0 0
250	0 1 0 0
240	1 1 0 0
230	0 0 1 0
220	1 0 1 0
210	0 1 1 0
200	1 1 1 0
2F0	0 0 0 1
2E0	1 0 0 1
2D0	0 1 0 1
2C0	1 1 0 1
2B0	0 0 1 1
2A0	1 0 1 1
290	0 1 1 1
280	1 1 1 1

2)安装主编程单元 它由基座(base unit)和适配器(adapters)组成。不同的器件有不同的适配器,各种适配器都可压接在基座上,与 LP6 卡一起构成一套完整的编程工具。图 3 - 1 是将适配器压接在基座上的示意图。注意切不可将 MPU 的缆线接在并行口或软件狗上。

3)安装 FLEX 下载缆线 首先依照图 3 - 1 安装 EPROM 编程适配器,然后如图 3 - 2 所示,安装下载缆线。缆线的另一端将连接在用户设计的印刷电路板上。这样可以实现对

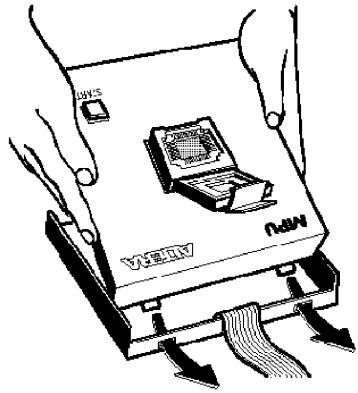


图 3－1　适配器安装示意图

FLEX 器件进行系统内编程。

4)安装位霸器 BitBlaster　BitBlaster 与微机的串行口相接,如图 3－3 所示。BitBlaster 侧面的三个开关可设置串行口的波特率。必须保证 BitBlaster 的波特率设置与微机中的波特率设置相同。表 3－4 表示波特率与开关之间的对应关系。

表 3－4　BitBlaster 的波特率设置表

波特率(bps)	开关设置(123)
9 600	1　1　1
14 400	0　1　1
19 200	1　0　1
38 400	0　0　1
57 600	1　1　0
76 800	0　1　0
115 200	1　0　0
230 400	0　0　0

有关编程硬件的使用将在本章后面详细论述。

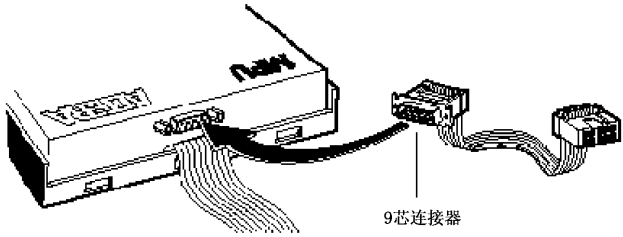


图 3－2　FLEX 下载缆线连接示意图

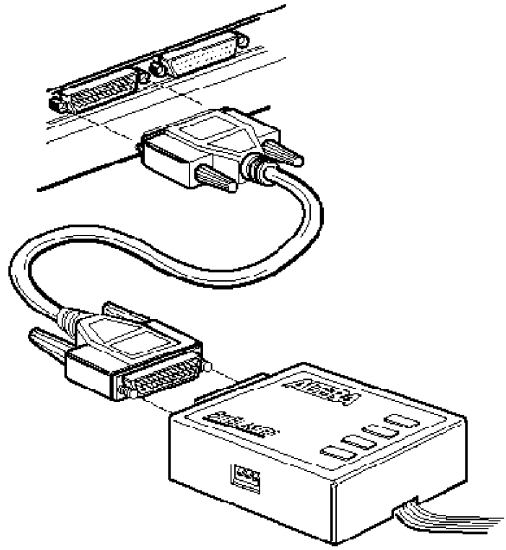


图 3－3　位霸器与微机的串行口相接示意图

3.1.2　MAX＋PLUS Ⅱ设计过程

本节简要介绍 MAX＋PLUS Ⅱ**的设计流程和设计环境，以及如何获得帮助等。**

一、设计流程和设计环境

图 3－4 **表示** FPGA **的整个设计流程，从设计输入到器件编程这四个阶段可在** MAX＋PLUS Ⅱ**提供的环境下完成。与图** 3－4 **对应，图** 3－5 **是** MAX＋PLUS Ⅱ**提供的设计环境。**

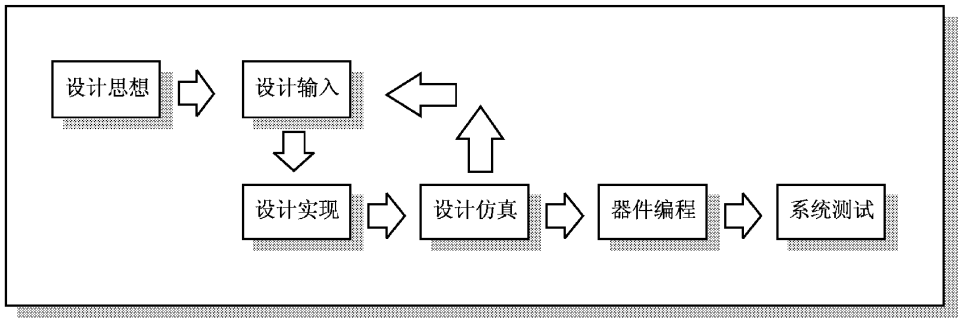


图 3－4　FPGA 设计流程图

1. **设计的输入**

(1)输入方式

MAX＋PLUS Ⅱ**中的输入可以有三种方式，即图形输入、文本输入和波形输入。这三种输入分别利用** MAX＋PLUS Ⅱ**的** Graphic Editor、Text Editor **和** Waveform Editor**。图形输入**

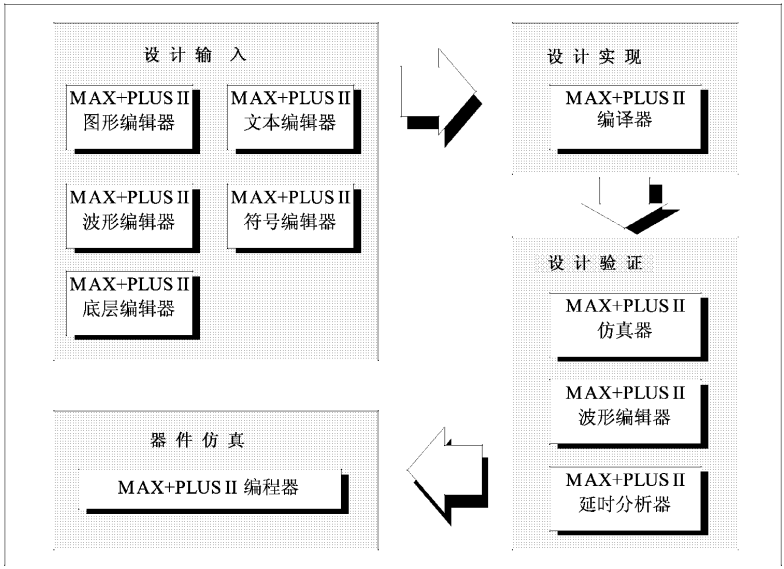


图 3－5　MAX＋PLUS Ⅱ的设计环境

即输入电路原理图，不仅可以使用 MAX＋PLUS Ⅱ中丰富的图形器件库，而且可以使用几乎全部的标准 EDA 设计工具，如可识别标准 EDIF 网表文件、VHDL 网表文件、OrCAD 原理图以及 Xilinx 网表文件等。文本输入方式支持 ALTERA 公司的 AHDL 语言，同时兼容 VHDL 和 Verilog HDL。波形输入最有特点，它允许设计者通过只编辑输入波形，而由系统自动生成该功能模块。

此外，符号编辑器（Symbol Editor）用于编辑用户自己的模块符号。通过底层编辑器（Floorplan Editor）可以观察实际器件的内部结构，并可改变器件管脚分布，或者调整各模块在器件内部宏单元之间的分布，从而优化器件性能。

（2）层次设计

层次设计的思想在设计输入时尤为重要。这种设计有以下优点：

①有助于帮助构思；

②在设计中增加结构性；

③易于设计调试；

④易于对设计的不同部分采用不同的设计输入方式；

⑤易于递增式设计（递增式设计由对单个子块的设计、实现及验证组成，子块用于分阶段建立的设计模块）；

⑥有助于并行式设计，也即可在许多设计人员中分配任务，并行开发整个设计中不同功能的模块。

2. 设计实现

设计实现意味着在所选的 FPGA 器件内物理地实现所需逻辑。这个过程主要由 MAX＋PLUS Ⅱ中的核心部分编译器（Compiler）完成。它主要依据设计输入文件自动生成用于器件

编程、波形仿真及延时分析等所需的数据文件，包括以下几个步骤：

①选择目标器件及设定编译环境参数，这一步由设计者自行设定，以下各步骤将由系统自动执行；

②生成各个模块的二进制网表(.cnf)文件；

③连接所有 CNF 文件，建立数据库，用以描述整个设计；

④进行逻辑综合，计算所有布尔等式，并优化触发器设计等；

⑤将整个设计映射到相应的器件内；

⑥产生波形仿真文件及器件编程文件。

3. 设计仿真

仿真器(Simulator)和时延分析器(Timing Analyzer)利用编译器产生的数据文件自动完成逻辑功能仿真和延时特性仿真。在仿真文件中加载不同的激励，可以观察中间结果以及输出波形。必要时，可以返回设计输入阶段，修改设计输入，最终达到设计要求。

4. 器件编程与测试

在仿真结果正确以后，就可以进行器件编程，也即通过编程器(Programmer)将设计下载到实际芯片中。下载之后，仍需进行动态仿真，因为在上一步骤中的仿真属于静态时序仿真，并未涉及实际器件。动态仿真是将实际信号送入实际芯片中进行的时序验证。最后则是测试芯片在系统中的实际运行性能。

以上所有设计工具都集成在 MAX + PLUS Ⅱ之中，每一个工具以窗口形式打开，并且在设计过程中可以同时打开多个窗口，通过在各个窗口之间切换，分别使用各个设计工具。

二、如何获得帮助

MAX + PLUS Ⅱ是一个易学易用的软件，其中重要的一个原因就是能够提供快速、及时、完全、细致的帮助信息。学会熟练地获取帮助则是掌握乃至精通 MAX + PLUS Ⅱ用法的一个关键。

最直接的帮助来自于 MAX + PLUS Ⅱ的 Help 菜单。图 3 - 6 逐项对菜单选项进行了解释。若需要某个特定项目的帮助信息，可以同时按＜Shift＞+＜F1＞键，或者选用工具栏中的快速帮助按钮  。此时，鼠标变为带问号的箭头，点中‘特定的项目’就可弹出相应的帮助信息。这里的‘特定项目’可以包含某个器件的图形、文本编辑中的单词、菜单选项，甚至可以是一个弹出的窗口。

不管以何种方式进入帮助窗口，其后的操作基本与 Windows 中的帮助使用方法相同，不再赘述，需要时可参看 Windows 使用手册。

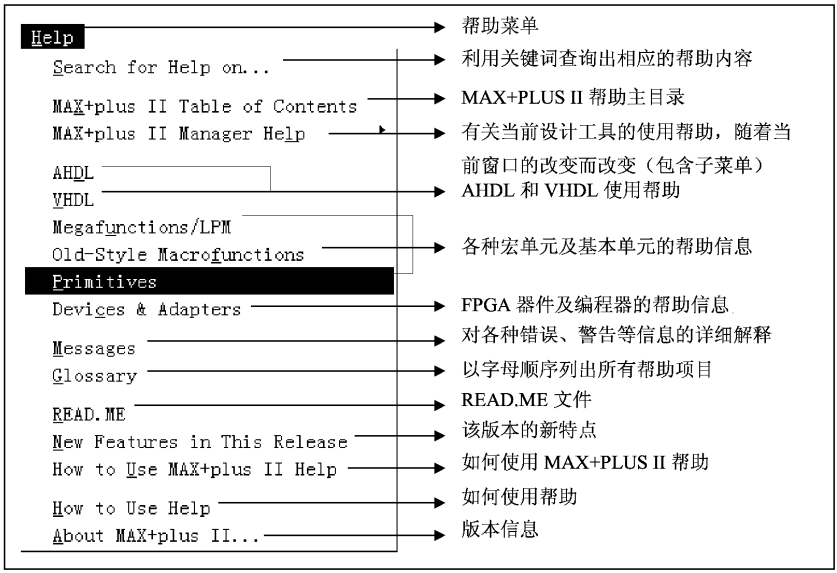


图 3－6　MAX＋PLUS Ⅱ帮助菜单

3.2　MAX＋PLUS Ⅱ系统的使用

此部分将通过一个简单的例子使读者掌握利用 MAX＋PLUS Ⅱ进行 FPGA 设计的过程，尤其是各种设计工具的基本使用方法。在开始工作之前，首先应了解‘工程’的概念。一个工程(Project)是一个设计的总和。它包含所有的子设计文件和设计过程中产生的所有辅助文件。以上一部分提到的层次设计的概念来理解，即所有的子设计文件是底层文件，各子设计文件可以是并列关系，也可以是包含关系，层次的深度没有限制。最顶层文件同子设计文件一样，也可以是图形或文本文件等(波形设计文件只能作为底层文件类型，不能作为顶层文件)。由于工程文件名与最顶层文件名相同，因此，它就代表了自底而上所有设计的总和。MAX＋PLUS Ⅱ的编译器、仿真器等是面向工程文件的，也就是说，编译器编译的是当前的工程文件。因此，在设计过程中，读者编辑一个工程设计文件后，可以通过编译器编译。另外，用斜杠将菜单分层，如 File 菜单下的 Save 子菜单，以 File ＼ Save 表示，并加粗显示。若还有子菜单，继续用反斜杠隔开，表示不同的层次。

图 3－7 是仅用图形编辑器完成的整个工程设计图。为使读者熟悉各种设计工具和层次的设计方法，将其分割成三个底层设计文件。其中 my_decoder. tdf(若操作系统(如 Windows 3.2)不支持长文件名可将 my_decoder 更改为其他名称)为文本文件，完成译码功能；my_dff. gdf 为图形设计文件，是一带使能端的 D 触发器；my_not. wdf 为波形设计文件，实现简单的

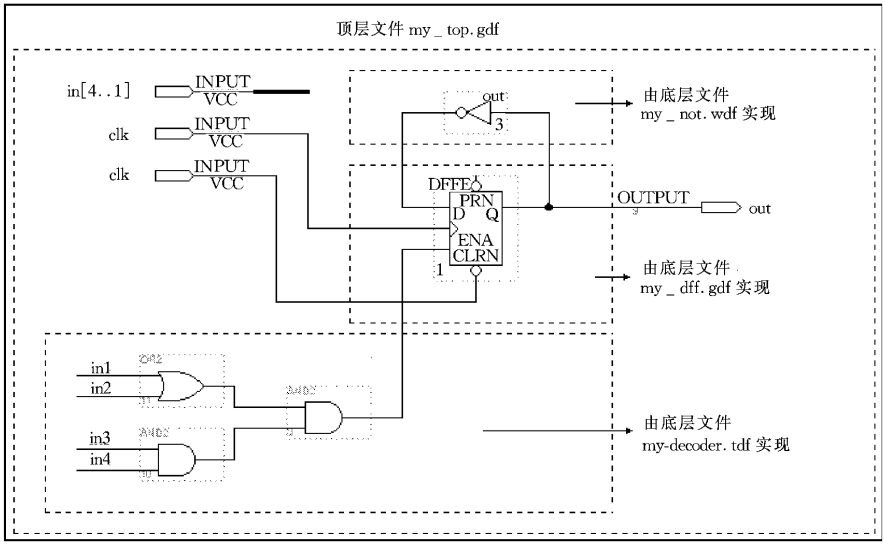


图 3－7　整体设计图

“非”门的功能,顶层文件为 my_top. gdf。整个设计的功能很简单,即输出在输入组合为 1010 时做 01 之间的翻转。

下面,请读者遵循以下设计步骤,一步一步完成整个 FPGA 设计。

一、建立工作目录

在进入 MAX＋PLUS Ⅱ之前,首先在硬盘上建立一个 \ maxplus2 系统目录之外的工作目录,以便把设计过程中的所有设计文件放入同一目录下。这一步并不十分必须,但是一个良好的设计习惯。在本例中,工作目录为 D:\ FPGA。

二、运行 MAX＋PLUS Ⅱ

进入 Windows 95 后,双点 MAX＋PLUS Ⅱ图标,进入 MAX＋PLUS Ⅱ,如图 3－8 所示。

在图 3－8 中,MAX＋plus II 菜单包含了所有设计工具选项。这一菜单始终不变,而其他菜单、工具栏和工具盘会随着打开的窗口而改变。在设计过程中,许多操作都可以通过菜单选项、工具栏、快捷键等几种方法进行,具体可依据个人习惯选用。

在图 3－8 中,窗口标题栏内显示了当前工程文件名 d:\ fpga \ all。这是由于进入 MAX＋plusII 时,系统自动调入上一次编译的工程文件。若是第一次进入 MAX＋plus II,显示的工程文件名为 Untitled1。

三、建立 my_dff. gdf 图形设计文件

1. 建立新文件

可选取菜单项 MAX＋plus II \ Graphic Editor 或菜单项 File \ New...,也可以用鼠标点工具栏中的  ,弹出的对话框如图 3－9 所示。选中 Graphic Editor file,并在下拉列表框中选. gdf(系统默认),选 OK 后开始编辑一个新的图形文件。编辑窗口如图 3－10 所示。

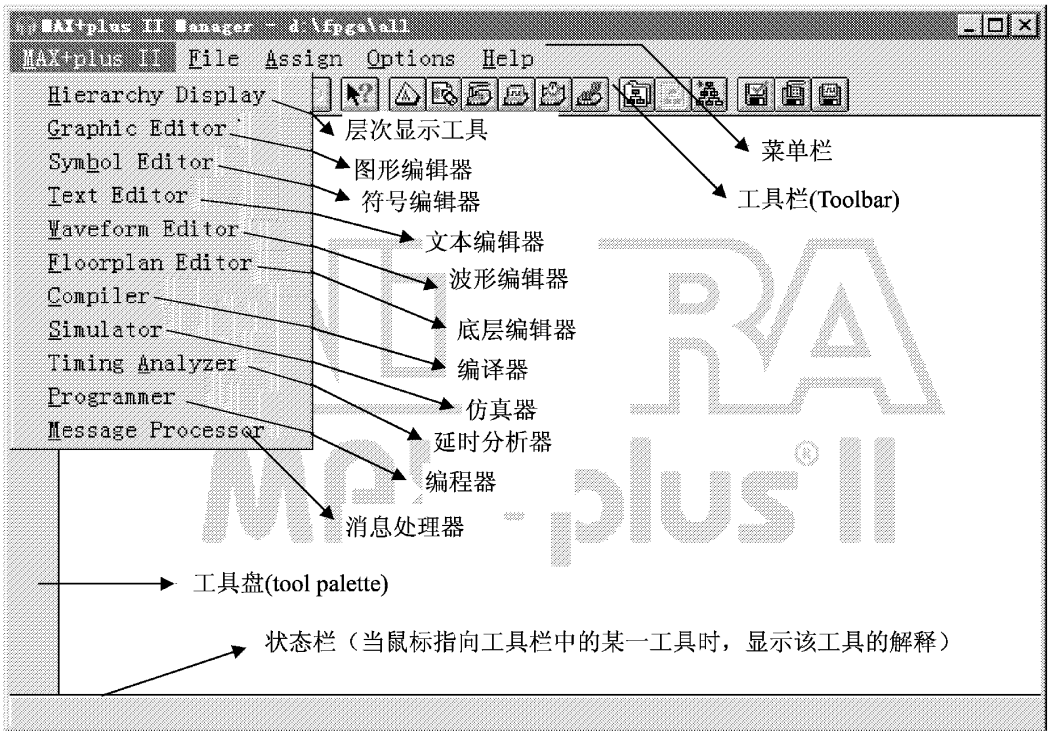


图 3－8　MAX＋plus II 管理器主画面

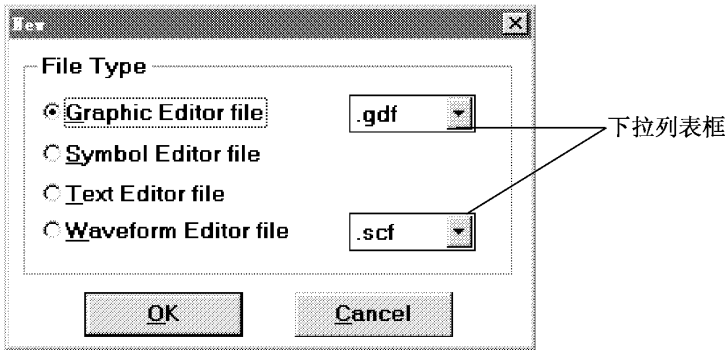


图 3－9　建立新文件对话框

在图 3－10 中看到，当打开一个未命名（Untitled1）图形编辑文件时，主窗口内的菜单、工具栏和工具盘与图 3－8 比较都有了很大的变化。

2. **保存文件**

选菜单 File ＼ Save，**在弹出的** Save As **窗口中，将存放目录更改为** D：＼ FPGA，**以** my_dff.gdf **的文件名保存。这一步也可以放在设计完整个图形后进行。**

3. **确定工程文件名**

选菜单 File ＼ Project ＼ Set Project to Current File，**即设置工程文件名与当前编辑的图形文件名相同，或者选** File ＼ Project ＼ Name. . .，**指定工程文件名。这一步也可以放在图形设计完成之后进行，但必须在上一步‘ 存盘 ’之后。此后，屏幕中间出现装入（Loading）工程文件** my

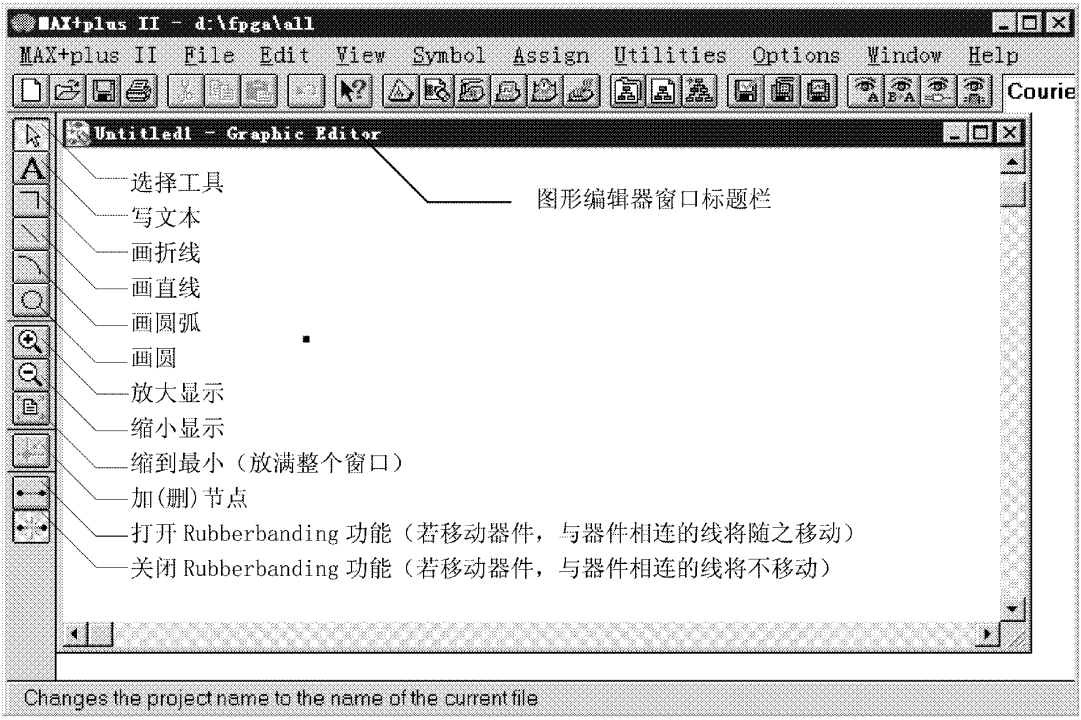


图 3－10　打开图形编辑器

_dff 的提示，持续时间约几秒钟。工程文件越大，所需时间越长。这时可以看到，主窗口的标题栏变为 d：\ fpga \ my_dff。

4. 输入电路原理图

通过前面三步基本完成了编辑图形设计文件的准备工作，下面可以开始输入电路图了。

1）输入逻辑功能符号　可以选择菜单 Symbol \ Enter Symbol...，更快捷的方法是用鼠标双点图形编辑器中的空白处，系统弹出窗口如图 3－11 所示。在 Symbol Name 栏内键入 dffe，选 OK 即可。在不知道器件符号名称时，双点相应符号库目录，在符号文件框内选择也可。本例中，可以双点库目录 d：\ max2work \ max2lib \ prim，再选出符号 dffe。用同样的方法依次键入 input、output 和 vcc。

2）复制、移动功能符号　由于需要 4 个输入端口，可以用上面的方法再绘制 3 个 input。更快捷的方法是，用鼠标点中 input 的符号，此时该符号边缘的虚线变为红色粗实线。然后，左手按住键盘上的＜Ctrl＞键（注意屏幕上的鼠标右上方出现一个小加号），右手操纵鼠标，按住鼠标左键点中 input 的符号并拖动。当出现一个同样大小的红色细矩形时，松开鼠标左键。这样就复制了一个 input 符号。同样，在某符号被选中时，可以用鼠标拖动该符号，也可以用键盘上的 Delete 键删除该符号。若要同时移动多个符号，可以按住鼠标左键画一个将所有要移动的符号包括在内的大矩形，然后用鼠标点中矩形内任意一点就可随意移动。

3）设置网格线　为了使电路图更清晰，可以给图形编辑器设置网格线。选菜单项 Options \ Show Guidelines，还可通过 Options \ Guideline Spacing 设置网格大小。如果想知道 dffe 使用方法的帮助，可以利用第一部分讲到的快捷帮助工具  。用鼠标点中该工具后，再点

dffe 符号，就可阅读 dffe 使用方法的详细帮助。

4）管脚的命名　绘制完所有的符号后，图形编辑器显示如图 3－12 所示。应注意到所有的输入输出管脚名为系统默认名 PIN_NAME，用鼠标左键双点‘ PIN_NAME ”，使其变为黑底白字显示，然后可直接键入管脚名。以同样方法修改所有的输入、输出管脚名。输入分别是 clk、dff_in、enable 和 clr ，输出是 dff_out。

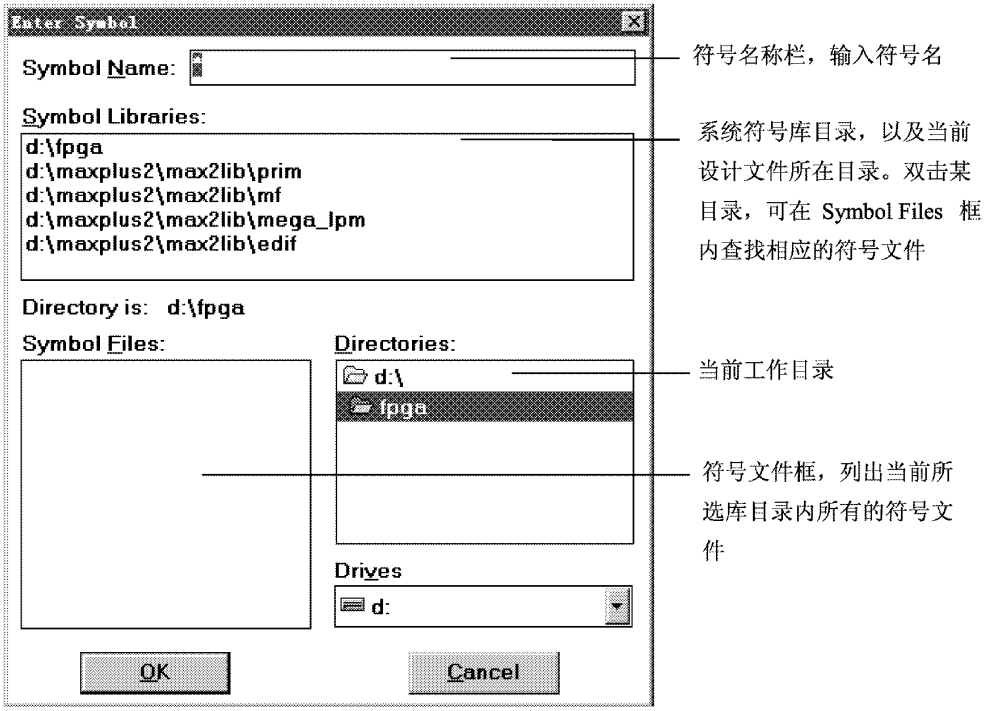


图 3－11　符号输入窗口

5）连接各符号　首先将各符号移动到合适的位置，以易于连线。将鼠标移至输入 clk 符号边缘的引脚处，鼠标箭头会自动变成十字形状。此时可以按住左键拖动，直至 dffe 的 clk 输入引脚处，松开左键。这样将输入时钟与 D 触发器的时钟输入端连在一起。同样方法，连接上所有的输入输出引脚。完成后的图形如图 3－13 所示。

窗口左端工具盘内的绘制直线、弧线等工具是为绘制标题或其他装饰性图形、文字用的。绘制电路图时不用这些工具。

5. 保存文件并检查错误

选菜单项 File \ Project \ Project Save & Check，系统自动弹出编译器窗口（图 3-14），生成网表文件。这一项是编译过程的第一步，主要功能是查找电路中的逻辑连接错误，生成网表文件。

完成后，系统弹出错误和警告信息统计对话框，选 OK 即可。若电路中存在逻辑错误，系统会弹出信息处理窗口。根据信息修改电路后，再重复这一步骤。直至没有错误后，将编译器窗口最小化或关闭该窗口。

6. 创建默认的符号

这是为了在顶层文件中绘制 my_dff 模块图形时，可以用一个符号来代替。正如绘制 my_

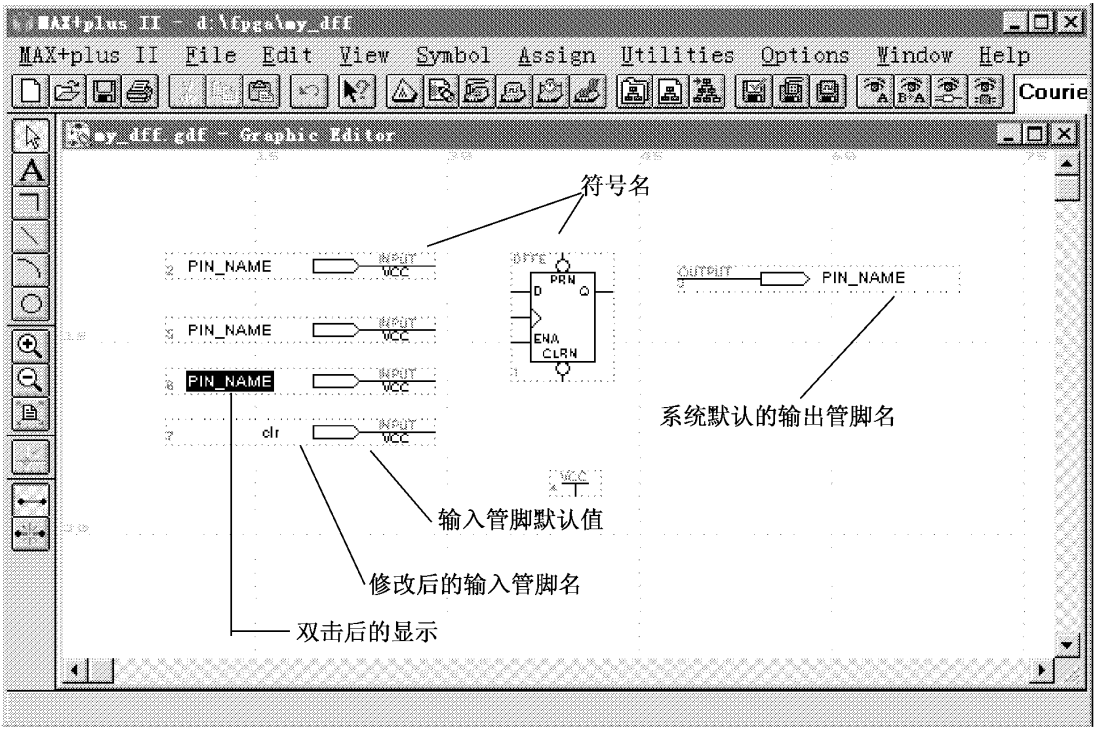


图 3－12　图形编辑器显示图

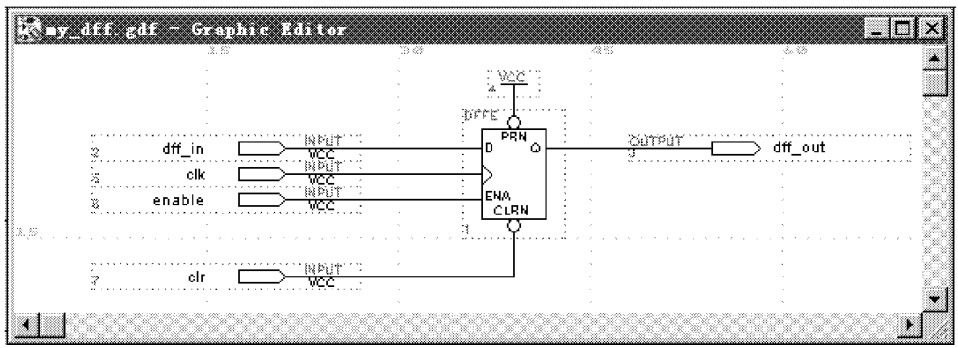


图 3－13　my_dff.gdf 电路原理图

dff.gdf **时用** dffe **符号代替** D **触发器一样。这正是层次设计的关键，即通过创建代表底层设计文件的符号，使得顶层设计文件能够包含所有子设计文件，并且设计结构清晰简明。**

选菜单项 File \ Create Default Symbol**。如果** File **菜单中没有这一选项，说明当前窗口不是** my_dff.gdf **所在的图形编辑器窗口。也就是说，该菜单项只有在编辑器（不仅仅包括图形）窗口下才有效。生成默认符号只需很短的时间，屏幕也没有任何变化，但此时在文件管理器中可以看到目录** d：\ fpga \ **下生成了一个新文件** my_dff.sym**。如果再重复这一步骤，系统出现提示‘符号文件已存在，是否覆盖？’。此时选** OK **即可。**

7. **关闭图形编辑窗口**

通过上面的几个步骤，已经成功地创建了图形设计文件 my_dff.gdf**。最后，关闭图形编辑**

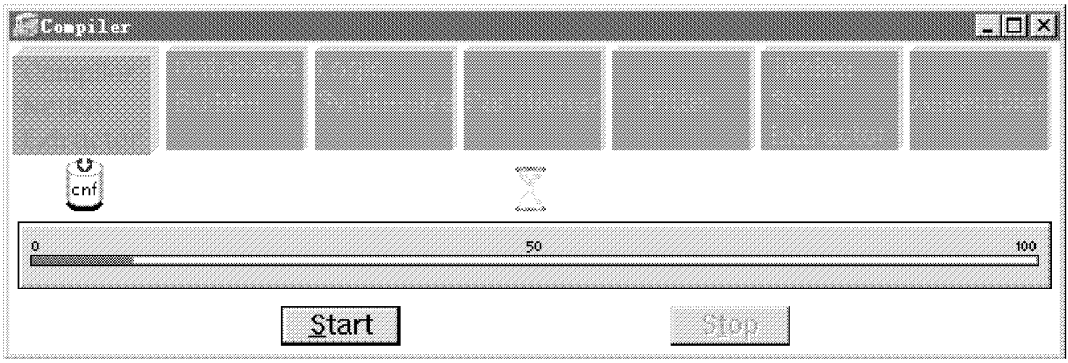


图 3－14　编译器窗口显示

器窗口，继续下面步骤。

四、建立文本设计文件 my_decoder. tdf

1. **建立新文件**

具体做法和步骤与三中的 1 相同。在图 3－9 对话框中选 Text Editor File。以下与步骤三中的 2、3 基本相同，保存文件并确定工程文件名。

2. **输入文本文件**

选中菜单项 Options \ Syntax Coloring。该项前出现一个对号，再选一次则对号消失。若选中该项，系统将自动改变输入文本的颜色，如保留字为蓝色、注释为绿色等。这样可使输入文件一目了然，减少输入错误。

有关 AHDL 语法规则请参看第四章，这里只需按照下面程序输入即可（其中以－－开头的注释部分可以不输入）：

```
－－定义 code 为二进制码 1010
CONSTANT   code = B"1010";
－－ SUBDESIGN 段定义输入输出
SUBDESIGN   my_decoder
(
        in[4..1]                    ;input;
        out                         ;output;
)
－－逻辑功能定义如下，即 out 在 in[]为 1010 时输出 1，否则为 0
BEGIN
        out = (in[] == code);
END;
```

3. **保存工程文件并创建默认符号**

做法与步骤三中的 5、6 相同。假如把上面程序中第四行子设计名 my_decoder 改为 my_decode，然后继续步骤三中的 5，编译器将在消息处理窗口内指出该错误，显示警告信息，如图 3－15 所示。

在图 3－15 中，点中该错误信息后，可以通过 Help on Message 按钮查看错误的详细提示。

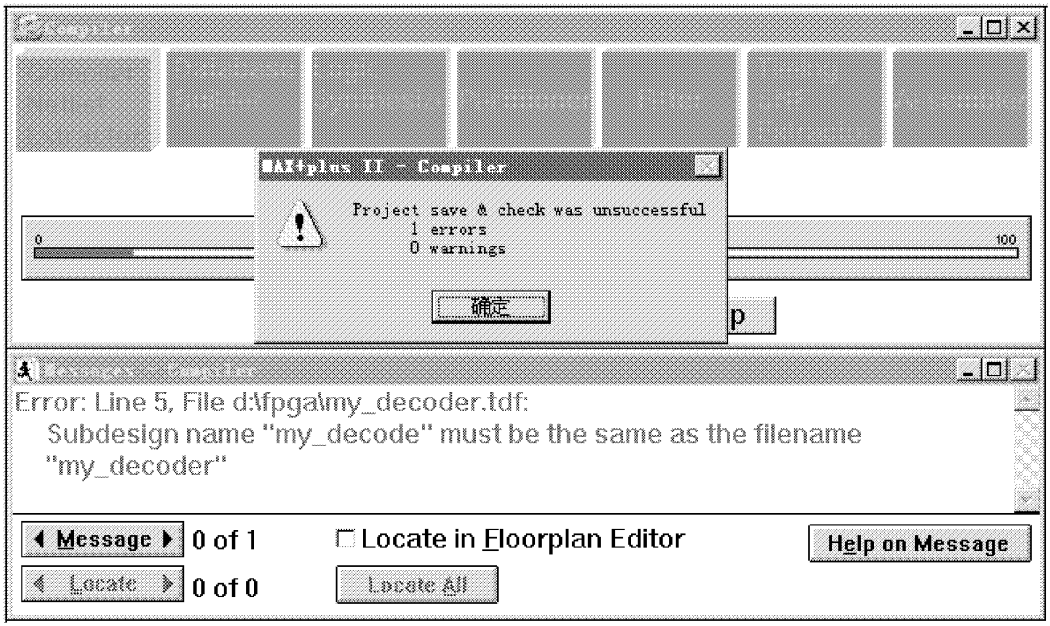


图 3－15　编译警告窗口

通过 Locate 按钮，可以找出错误，系统自动打开文本文件，并将光标定位在 my_decode 处。改正错误后，重复三中的 5、6 步骤。这时可以通过选菜单 File \ Edit Symbol 查看系统生成的默认符号，如图 3－16 所示。

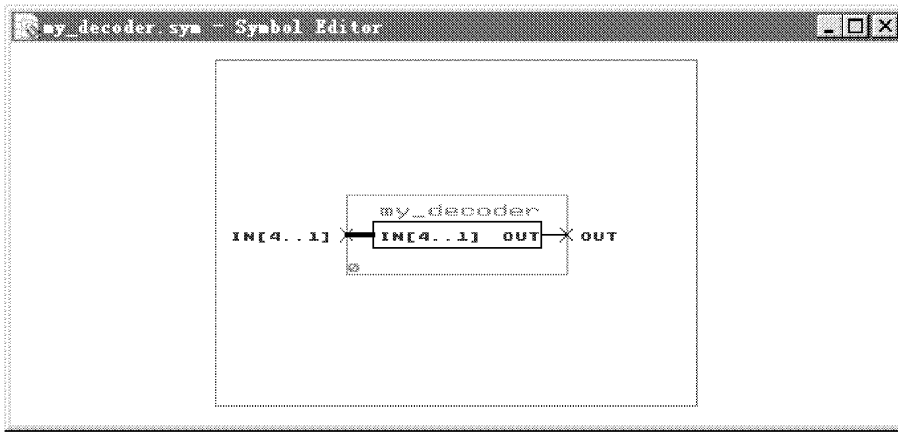


图 3－16　符号编辑器

4. 关闭所有窗口

通过以上几个步骤，已经创建了文本设计文件 my_decoder. tdf。最后，关闭所有窗口，继续下面的步骤。

五、建立波形设计文件 my_not. wdf

1. 建立新文件

做法与步骤三中的 1 相同，在图 3－9 对话框中选 Wave Editor File 并且在右边的下拉列表框中选择. wdf(而非. scf)。以下步骤与三中的 2、3 基本相同，保存文件并确定工程文件名。

选菜单项 Options \ Show Grid 显示网格线，选 Options \ Grid Size. . . 改变网格大小。屏幕显示如图 3－17 所示。

图 3－17　建立新的波形设计文件

2. 编辑波形输入文件

1）创建输入、输出管脚　在图 3－17 所示的管脚（组）名字、类型及赋值区域内双击，或选菜单项 Node \ Insert Node，弹出窗口如图 3－18 所示。

在图 3－18 管脚名栏内键入 in，I/O 类型选择 Input Pin，管脚类型栏自动变成输入型，选 OK 即可。用同样方法，建立输出管脚 out，I/O 类型为 Output Pin，管脚类型为组合型。输入输出建立后的窗口显示如图 3－19 所示。

2）编辑输入输出波形　对输入、输出波形进行编辑即对输入、输出波形赋值。在图 3－19 中，按照图中所示方法给输入输出赋值。

3）保存工程文件并创建默认符号　做法与步骤三中的 5、6 相同。最后，关闭所有窗口，继续下面的步骤。

六、建立顶层图形设计文件

经过以上五大步骤，已基本完成所有底层设计文件的编辑工作，在此基础上可以开始编辑顶层文件 my_top. gdf。

1. 建立图形设计文件

做法与步骤三中的 1 相同。在图 3－11 符号输入窗口中，可以看到符号文件框中有了三个新的符号文件，即在前面建立的三个底层文件。直接点中相应的符号文件，选 OK。依照步

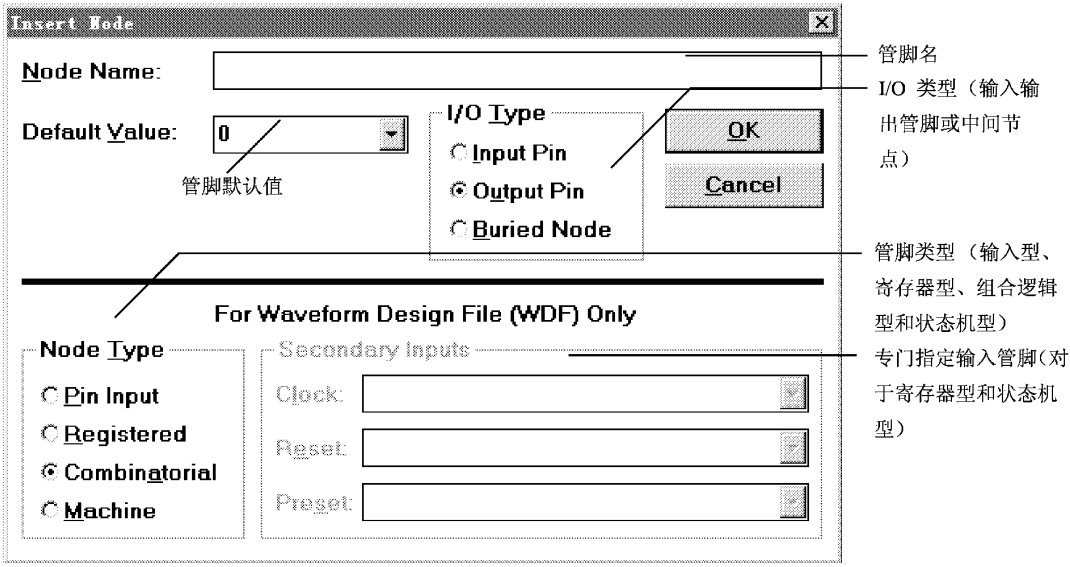


图 3－18　**创建管脚窗口**

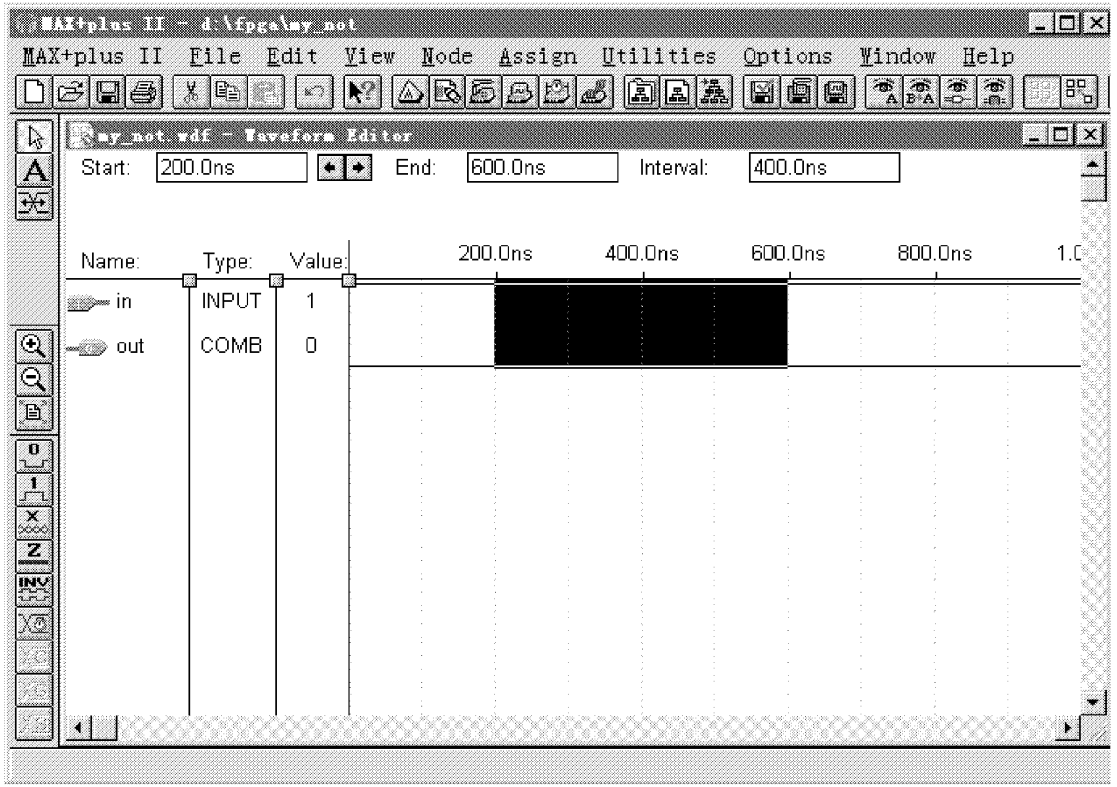


图 3－19　**编辑波形设计文件**

骤三中 4 **的方法，键入** input **和** output。

2. 翻转一个器件符号

如图 3－20 所示，用鼠标右键单击 my_not 的符号，在弹出的菜单中选水平翻转（Flip Horizontal），这样可使符号易于连接。按照步骤三中 4 的方法修改输入、输出管脚名，它们分别为 code[4..1]、clock、clear 和 out。从这里也可以看出，管脚名与其所连的符号引脚名可以不相同。

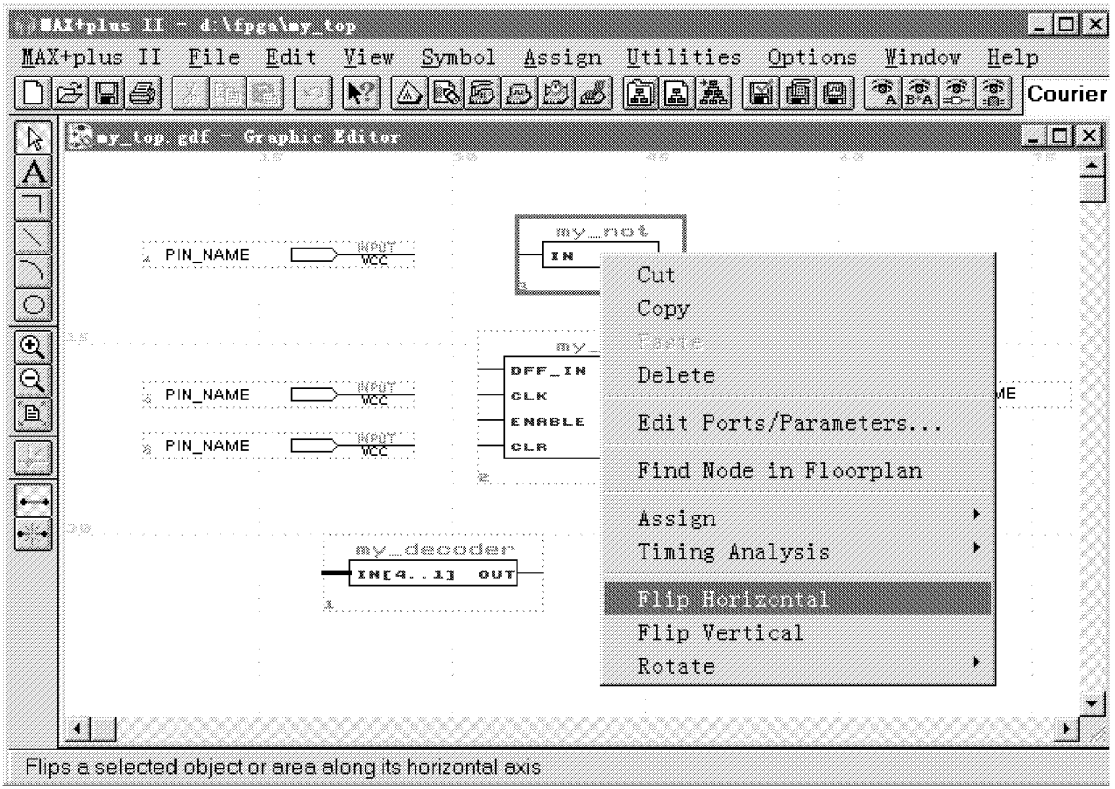


图 3－20　翻转一个符号

3. 连接各符号

按照输入电路原理图中的 3)、4)、5)的方法连接输入、输出，my_dff 的 clk、clr 端分别与输入 clock、clear 相连，my_dff 的 dff_in 端与 my_not 的 in 端、my_dff 的 dff_out 端与 my_not 的 out 端相连，out 与 my_dff 的 dff_out 端相连，my_decoder 的 out 端与 my_dff 的 enable 端相连。

从 my_decoder 的 in[4..1]引脚端绘制出一条直线。由于 in 为管脚组，因此，系统自动将线型定为总线型，即粗实线。用鼠标左键点中该总线，此时总线变为红色，闪烁的小方块位于总线上，然后用键盘输入 code[4..1]。这样该引脚与同名的输入管脚就连接在一起。这称为逻辑连接。还可以在输入 code[4..1]的右端引脚处绘制出一条直线，但此时该线并没有自动变为总线。可以用鼠标右键点中该线，在弹出的右键菜单中选 Line Style 中的总线线型，如图 3－21 所示。

4. 节点的绘制

在连接输出端 out 与 my_dff 的 dff_out 端时，注意系统在两线相交时自动加上了一个节点。这时可利用图 3－10 中指出的加(删)节点工具绘制或删除一个节点。具体方法是先用鼠

标左键击中该节点处，使闪烁的小方块位于节点处，此时会发现窗口左面工具盘内的加(删)节点工具由不可使用(灰色显示)变为可以使用(红色显示)。这时可以用鼠标单击此工具按钮，则能实现绘制一个节点或删除一个节点的功能。

5. 保存文件

完成所有图形绘制后保存文件，电路图如图 3－22 所示。以下操作与前面的有所不同了。

七、编译工程文件

1. 打开编译器窗口

选菜单项 MAX＋plus II ＼ Compiler 或工具栏中的  按钮，弹出窗口如图 3－23 所示。读者的屏幕显示与此相同，请按照以下步骤设置编译环境。

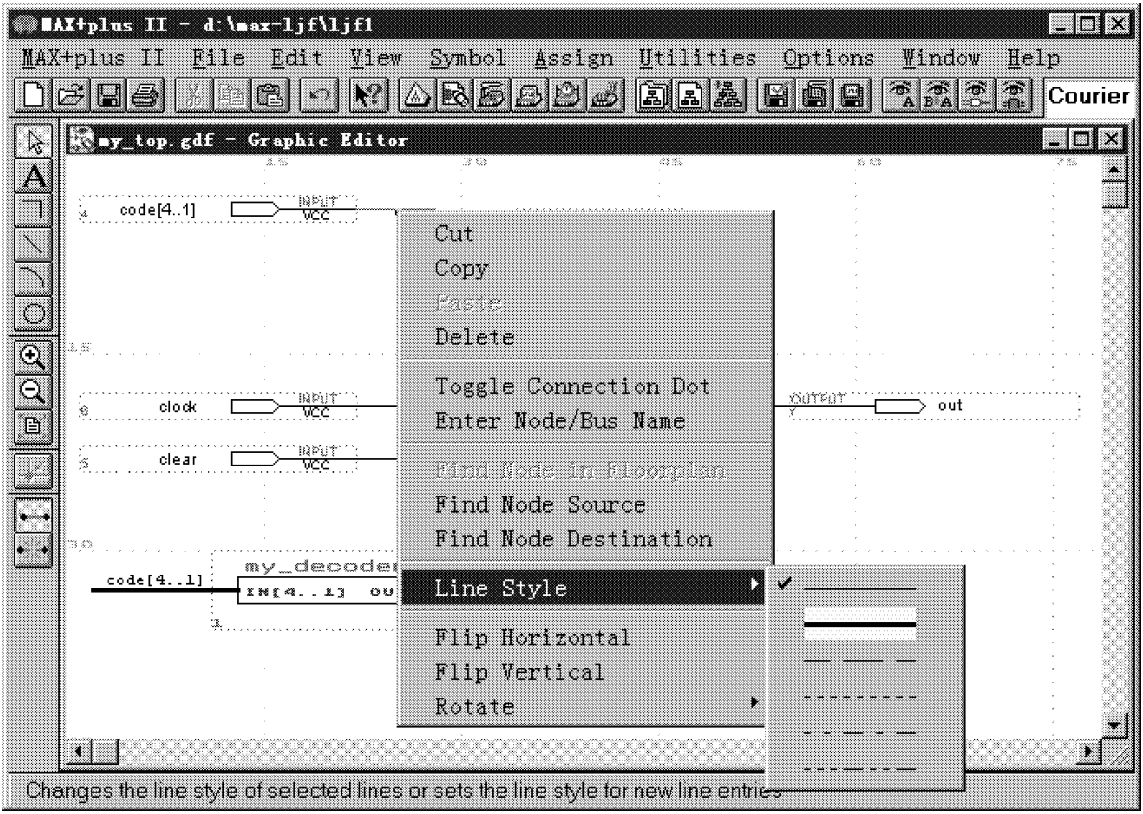


图 3－21　改变线型为总线型

2. 选择器件类型

选菜单项 Assign ＼ Device. . . 弹出对话框如图 3－24 所示。在 Device Family 下拉列表框内选 MAX7000，在 Devices 下拉列表框内选 AUTO，表示由系统自动指定与工程文件相匹配的器件，最后选 OK。

3. 选择灵活编译命令

打开菜单项 Processing ＼ Smart Recompile 的开关，再次编译时此命令可使系统忽略前次编译中不变的部分，而快速重新编译工程文件。

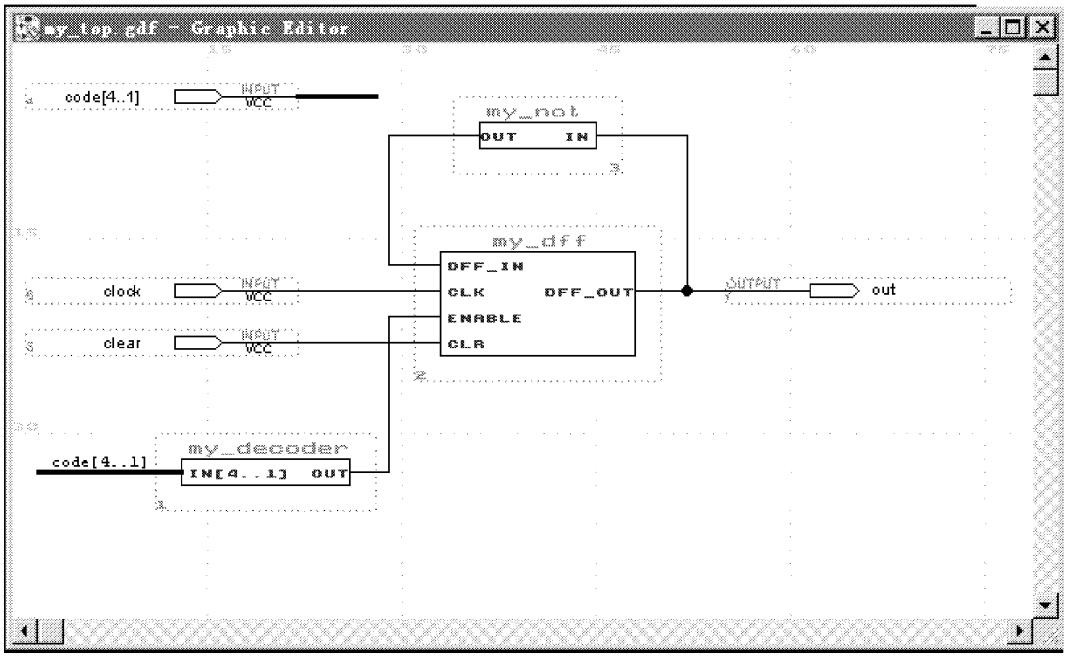


图 3－22　顶层图形设计文件图

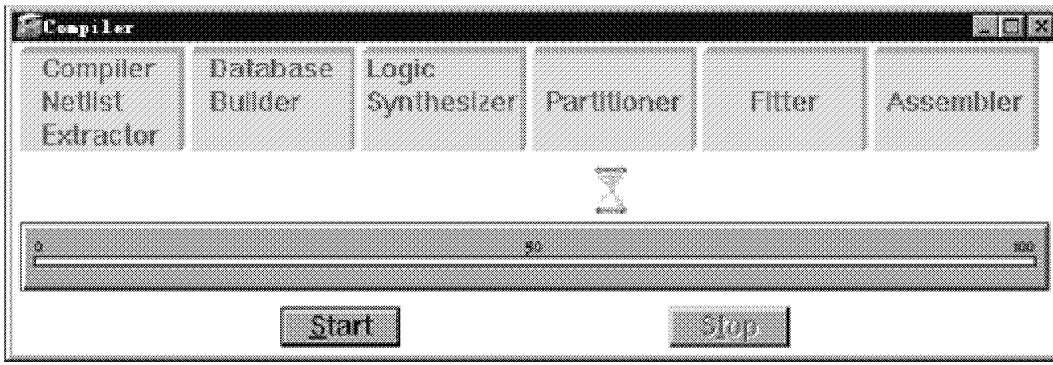


图 3－23　编译器窗口显示

4. 打开设计医生工具

打开菜单项 Processing \ Design Doctor 的开关(用户购买的软件必须包含该项工具),此时编译窗口内增加了一个医生图标。该命令可使系统根据一定规则检验电路设计中的不稳定因素。这些因素并不是设计错误,而是可能导致 FPGA 工作不稳定的原因。选菜单项 Processing \ Design Doctor Settings,在设计规则栏内选择 EPLD Rules(系统默认值),如图 3－25 所示。因为前面选择的 MAX7000 器件属于 EPLD 范围,所以选用该规则。

5. 设定逻辑综合类型

选菜单项 Assign \ Global Project Logic Synthesis,在图 3－26 对话框中选 OK 即可,即选取系统默认值。编译器将根据选择合理配置资源。

6. 打开 SNF 分析器

选菜单项 Processing \ Timing SNF Extractor,此选项将在编译过程中为后面时延分析生

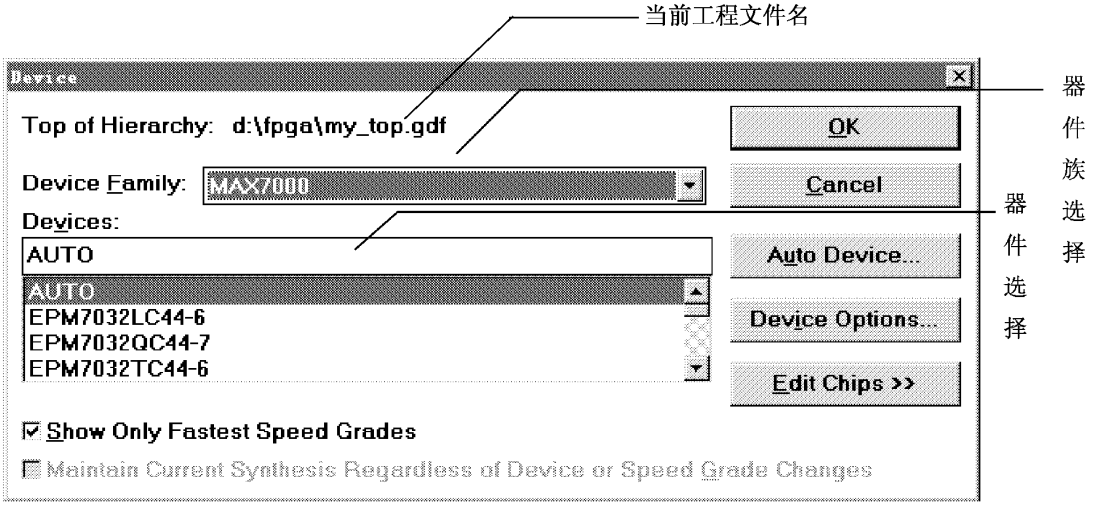


图 3－24　选择器件对话框

成各种数据文件。

经过以上设置之后，编译窗口如图 3－27 所示。在图 3－27 中，从左到右依次分类，编译过程可分为以下 7 个过程：

①网表分析器将所有设计文件转化为二进制网表文件，生成层次文件；

②数据库建立器建立用以描述整个设计的数据库；

③逻辑综合器对整个设计进行逻辑综合，优化触发器设计等；

④分割器选择适合当前工程设计的相应器件；

⑤分配实现器将逻辑设计在特定器件内实现，生成报告文件；

⑥仿真文件生成器生成时延仿真所需的各种文件；

⑦装配器生成用以硬件编程的各种文件。

图 3－25　设计医生设置对话框

由上可见，编译过程是后面仿真和编程的基础，编译器是整个软件的中心。

在图 3－27 中，点 START 按钮，开始编译。状态条和进程光标表示编译过程正在进行。由于编译是在后台进行的，因此编译过程中用户完全可以做其他工作。本例规模很小，编译时间很短。若编译大的工程文件，可以转换到其他工作环境中继续工作。编译成功后屏幕显示如图 3－28 所示。在图 3－28 的消息窗口中显示了三条消息，说明设计医生并未发现任何设计不稳定因素。这样，工程文件 my_top 就可成功地在 EPM7032LC44－6 器件中实现。

7. 查看报告文件

双点图 3－28 中的 rpt 图标，报告文件会出现在文本编辑器窗口内。从 report 文件中可以获得器件资源使用情况、器件管脚分配图、编译时间等信息。在制作目标印刷电路板时，必须依据器件管脚分配图进行。对报告文件中不理解的段落可以通用工具栏中快捷帮助按钮点中相应段落标题（左右分别有两个 * 号），查看详细帮助。

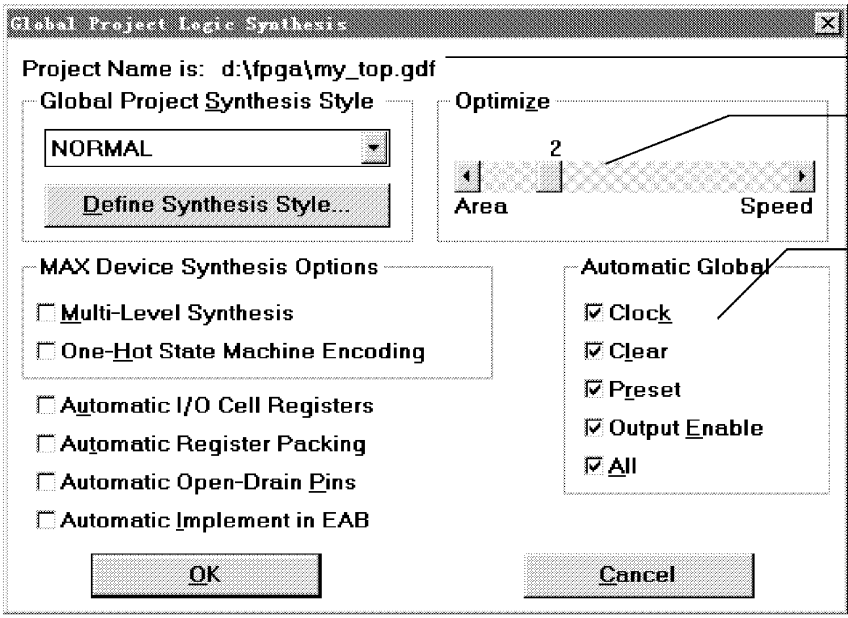


图 3－26　逻辑综合类型设定窗口

图 3－27　编译器窗口

最后关闭所有窗口，继续下面的步骤。

八、查看层次显示图

通过层次显示图可以清晰地看到整个工程文件的结构，并且能够快捷地打开各个设计文件。选菜单项 MAX + plus II \ Hierarchy Display，屏幕显示如图 3－29 所示。

九、改变管脚设置

在步骤七的 7 中，通过报告文件可以看到输入、输出管脚的设置。这些是由系统自动设置的，但在制作印刷电路板时，为了连接方便等，希望某些管脚处在特定的位置。完成此项功能可以有三种方法：

①直接修改 .acf 文件，可在图 3－29 层次显示中双击 .acf 文件的图标，但这样极易导致错误，因此建议不使用该方法；

②通过 Assign 菜单直接指定管脚位置；

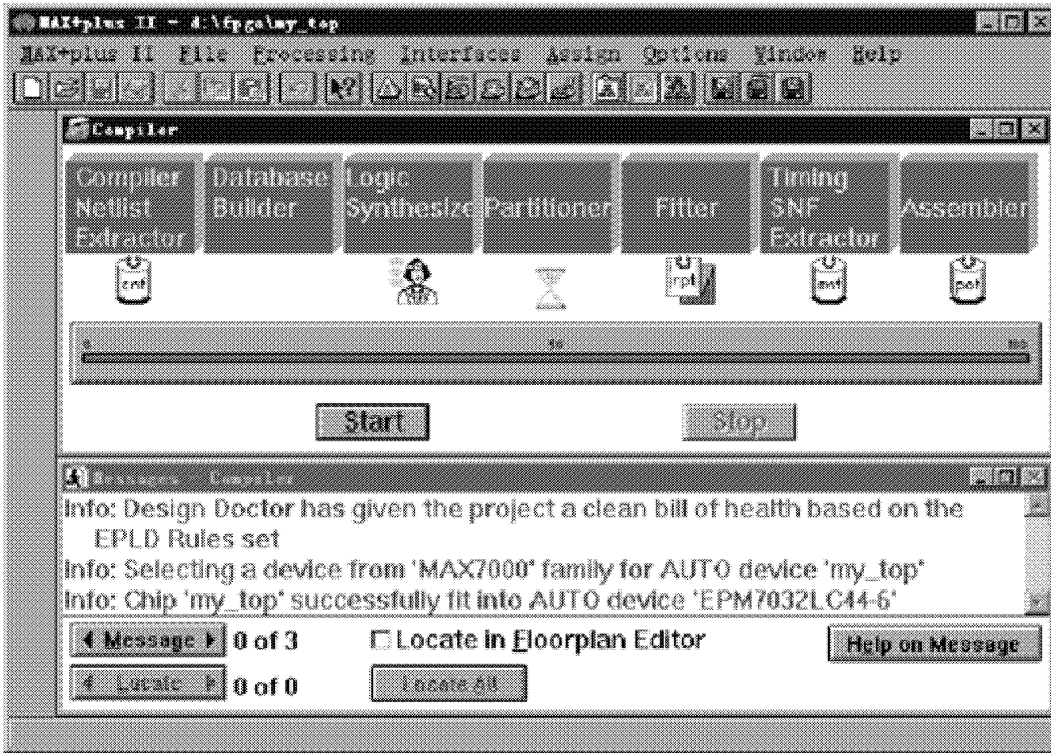


图 3－28　编译结束屏幕显示

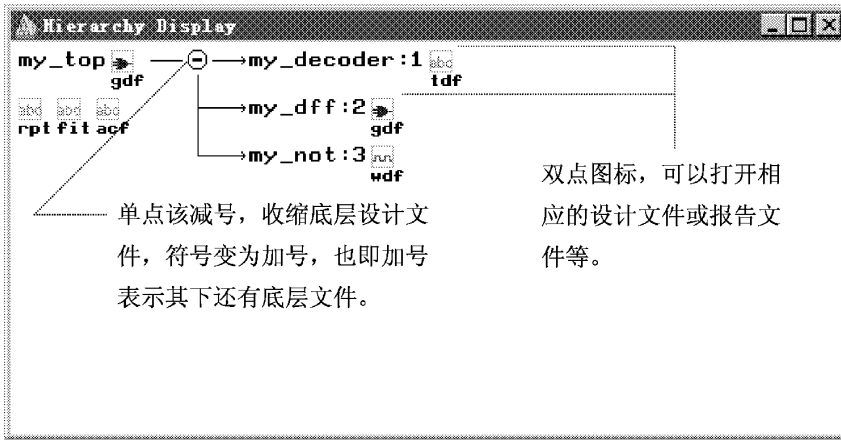


图 3－29　层次显示窗口

③通过底层编辑器直接编辑管脚位置。

以下主要讲述最后一种方法的具体操作步骤。

1. **打开底层编辑器**

选菜单项 MAX＋plus II ＼ Floorplan editor**，编辑器显示如图** 3－30 **所示。若屏幕显示与图中不符，可选菜单项** Layout ＼ Last Compilation Floorplan**。**

2. **重新分配管脚**

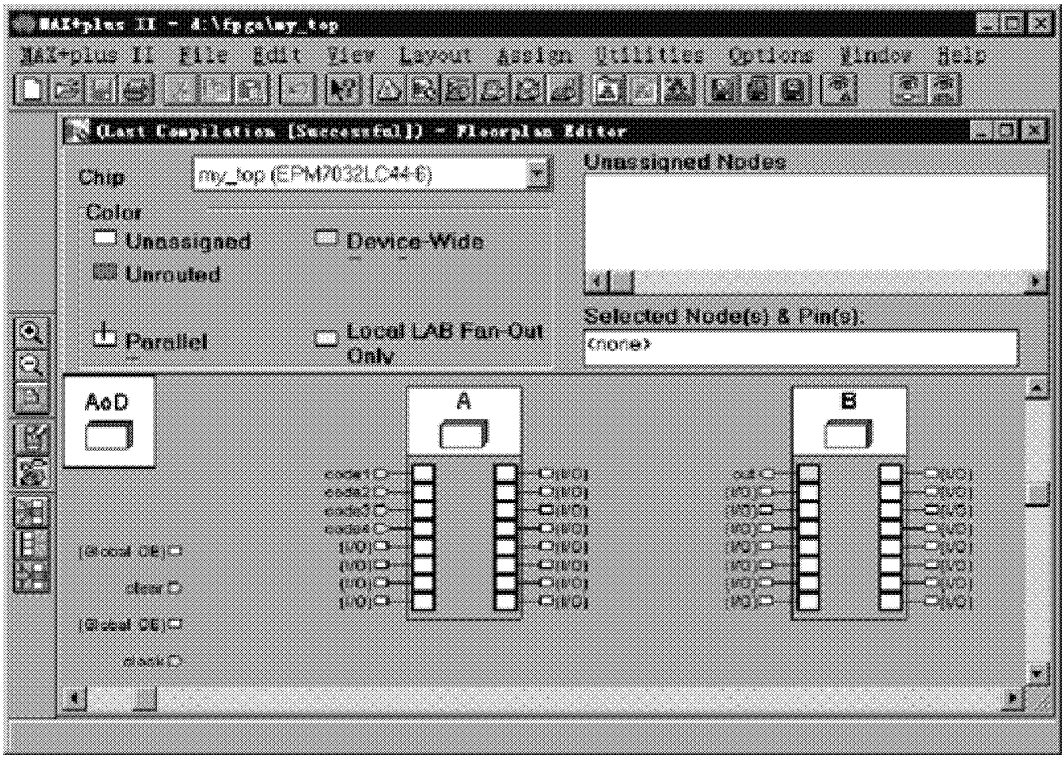


图 3－30　底层编辑器

①选菜单项 Assign \ Back-Annotate Project，在弹出的对话框中将 Chips、Logic Cells、Pins & Devices 选项选中，再选 OK。这一步骤是用工程编译后的管脚分配、资源配置等数据信息（包含在 .fit 文件中）覆盖当前用户指定的分配信息（包含在 .acf 中），这样就可以重新对当前的管脚进行分配了。选菜单项 Layout \ Current Assignments Floorplan。

②选菜单项 Options \ Show Moved Nodes in Gray。

③用鼠标选中代表输出 out 的管脚，如图 3－31 左图所示。鼠标在该处停留片刻，出现提示文字 out @ 41(I/O)，说明 out 位于器件的第 41 管脚。

④按住鼠标左键，将 out 管脚拖动到第 39 管脚，如图 3－31 右图所示。

⑤选菜单项 Layout \ Device View，或双击 LAB 显示图空白处转换两种显示方式，重复以上三个步骤，可重新定义、配置各输入、输出管脚。

3. 重新编译工程文件

打开编译器，选 Start 按钮，编译成功后，可在报告文件 .rpt 中看到，输出 out 管脚已改变为第 39 脚。

注意，有些输入信号（如时钟）有默认管脚，不能随意改变。另外，尽量使用系统所做的管脚和资源配置。

十、仿真器的使用

编译成功的设计并不一定完全正确，只有通过仿真才能验证电路是否真正达到设计要求。Simulation 可以分为三种，即逻辑特性（Functional）仿真、时延特性（Timing）仿真和连接（Linked）仿真。对于时序逻辑来说，一般都用时延特性仿真。仿真器利用编译器产生的数据

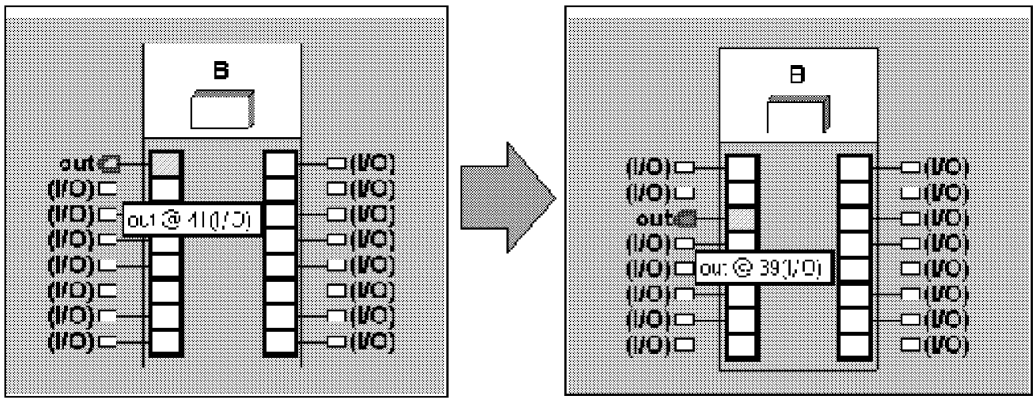


图 3－31　重新配置输出管脚 out

文件工作。因此，确定某种仿真类型需在编译之前进行。具体方法见步骤七中的 6。

1. 创建仿真通道文件

做法与步骤五中的 1 类似。在图 3－9 对话框中选 Wave Editor File，并且在右边的下拉列表框中选择 .scf，然后保存文件。文件名取系统默认值，即与当前工程文件名相同。选菜单项 Options \ Show Grid 显示网格线 ;选 Options \ Grid Size... 改变网格大小。屏幕显示与图 3－17 相同(仅窗口标题栏内文件名不同)。

2. 编辑仿真通道文件

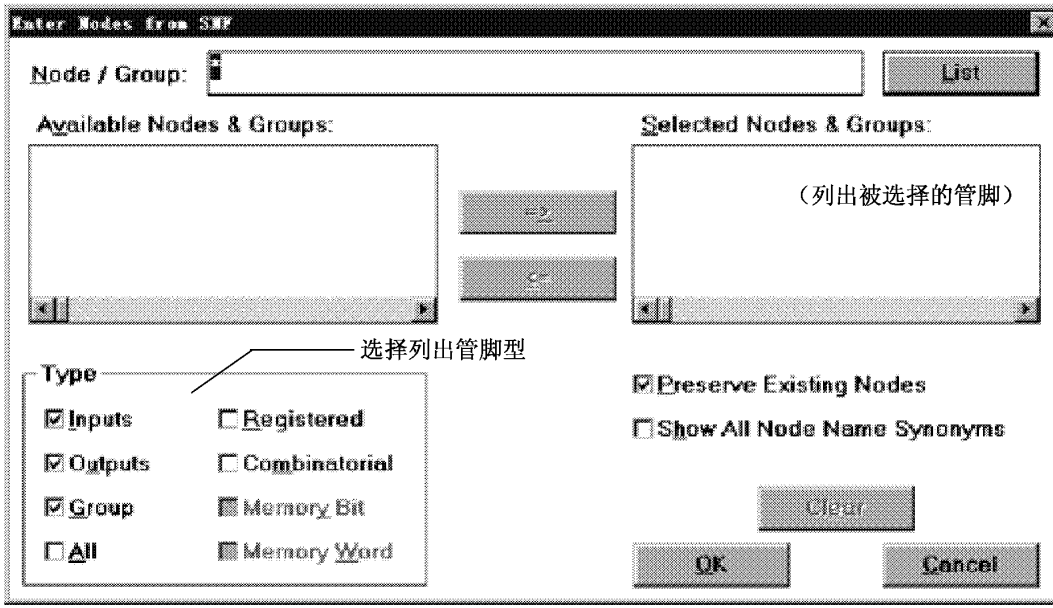


图 3－32　选择节点(包括输入输出及中间节点等)对话框

1)选择输入输出管脚　在波形编辑器编辑区域内击鼠标右键，在弹出的菜单中选 Enter Nodes from SNF...，或选菜单项 Node \ Enter Nodes from SNF...，弹出窗口如图 3－32 所示。可以直接在 Node/Group 内输入管脚名，也可点 List 按钮，在窗口左侧的可用管脚框内列

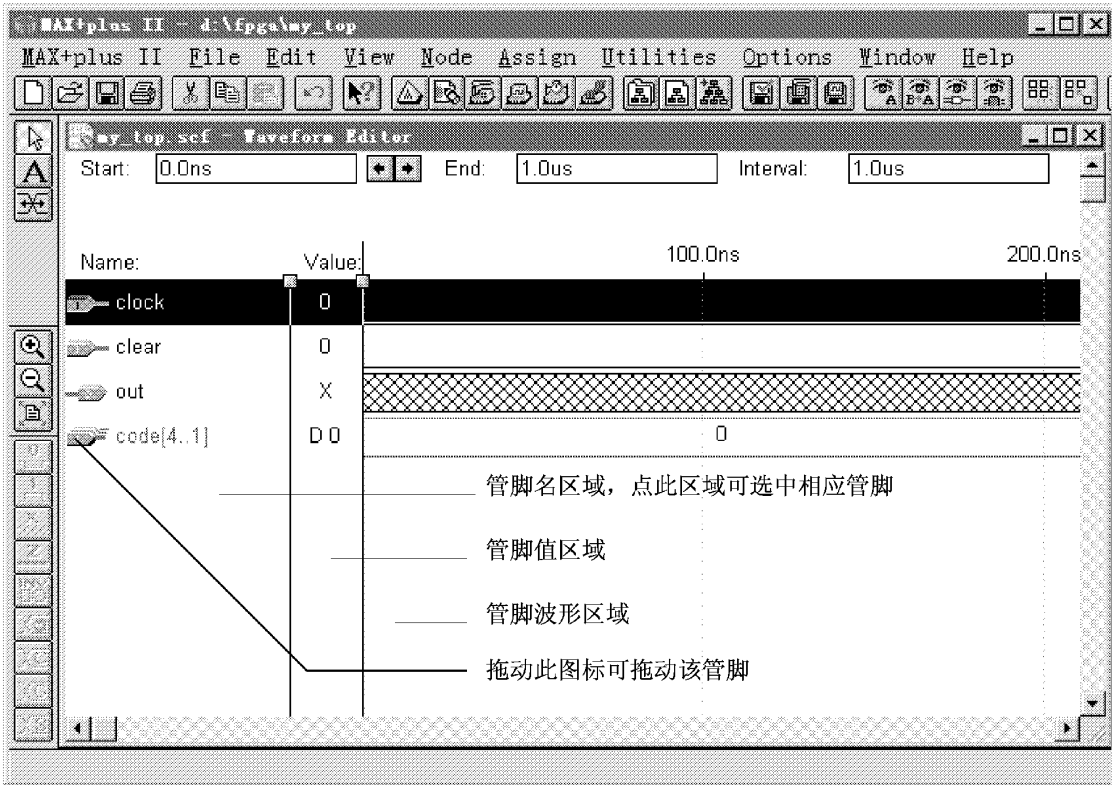


图 3－33　编辑仿真通道文件

出所有已知管脚。选中某个管脚，利用➡按钮，将其选至窗口右侧的被选管脚框内。依次将 clear(I)、clock(I)、out(O)和 code[4..1](I)选中，然后选 OK。图形显示如图 3－33 所示。由图可见，输入管脚默认值为 0，输出管脚为 x 不定状态。

2)移动一个管脚　用鼠标左键按住代表管脚的小图标拖动，可以将该管脚拖至任意位置。

3)拆分管脚组　用鼠标右键击中管脚 code[4..1]的 Name 区域，在弹出的右键菜单中选 Ungroup，此时看到 code[4..1]管脚组分为 code4 至 code1 的四个输入管脚。同样，用左键选中这四个管脚，再按右键，在右键菜单中选 Enter Group...。在弹出对话框中，可以改变管脚组名，也可以改变管脚组值的数制。选 OK，这样又将四个管脚合为一个管脚组。

4)改变仿真总时间　选菜单项 File \ End Time...，在图 3－34 对话框中将系统默认时间 1.0μs 改为 2.0μs，并选 OK。

5)给输入管脚 clock 赋值　选中管脚 clock，击鼠标右键，在弹出的右键菜单中选 Overwrite \ Clock...，在弹出的对话框中将时钟周期改为 100.0ns，如图 3－35 所示，最后选 OK。若时钟周期为灰色显示不可改变，应把菜单项开关 Options \ Snap to Grid 关闭。

6)给输入管脚 clear 赋值　仿照五中的 2 的方法，将管脚 clear 赋 1 值，并且从 0 到 100ns 处赋 0 值。

7)给输入管脚组 code[4..1]赋值　选中管脚组 code[4..1]，击鼠标右键，在弹出的右键菜单中选 Overwrite \ Count Value...，弹出对话框如图 3－36 所示。其中计数周期与时钟周期相同，为 100.0ns。将 Increment By 输入框内的默认值 1 改为 2，表示计数间隔，然后选 OK。

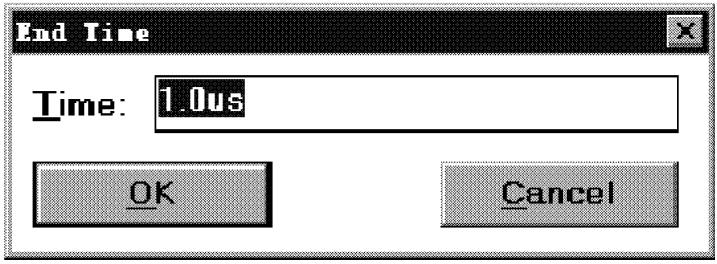


图 3－34　改变仿真总时间对话框

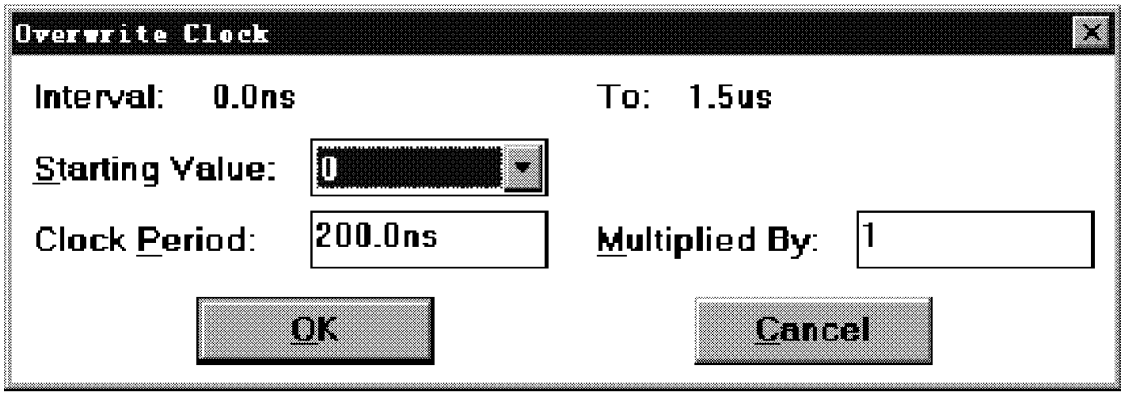


图 3－35 赋时钟值对话框

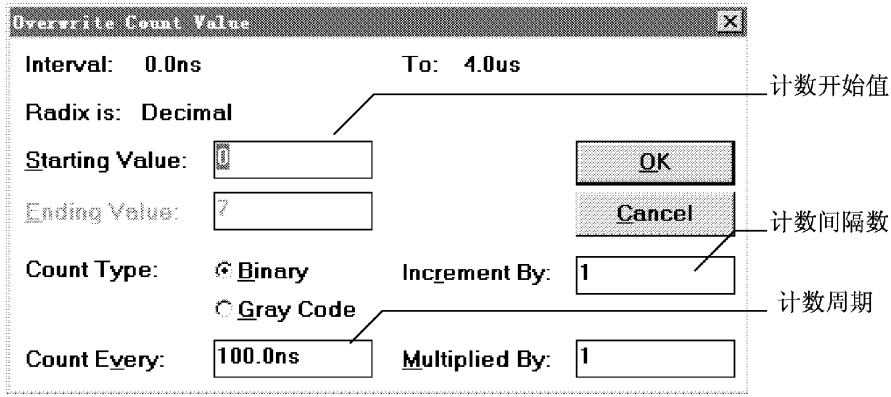


图 3－36　赋计数值对话框

将编辑后的仿真文件存盘，波形如图 3－37 所示。

3. 开始仿真

选菜单项 MAX＋plus II \ Simulator，弹出仿真器窗口如图 3－38 所示。

点 Start 按钮，系统开始仿真。与编译器一样，仿真器也是在后台工作。在仿真时，可以切换到其他环境中工作。仿真完成后，弹出图 3-39 提示窗，击 OK 按钮。

击图 3－38 中的 Open SCF 按钮，可查看仿真文件，如图 3－40 所示。如前所述，在 code 输入编码为 1010（即十进制的 10）时，输出 out 做 0 和 1 之间的翻转。这证明设计达到了要求。

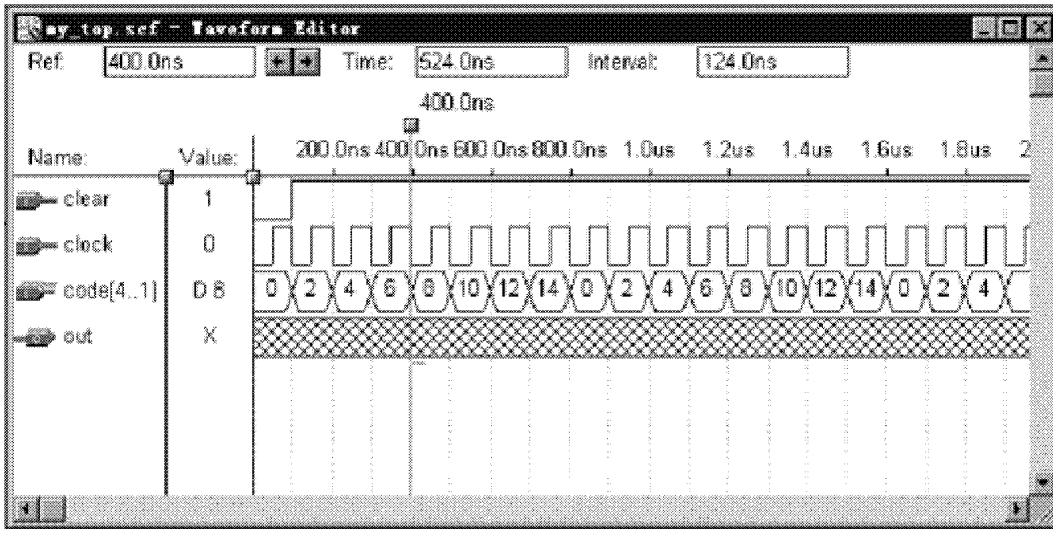


图 3－37　编辑完成仿真通道文件

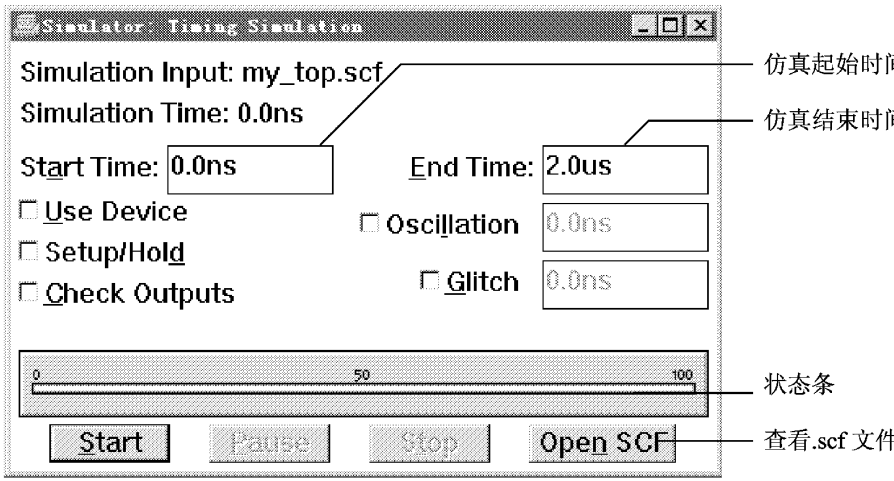


图 3－38　仿真器窗口

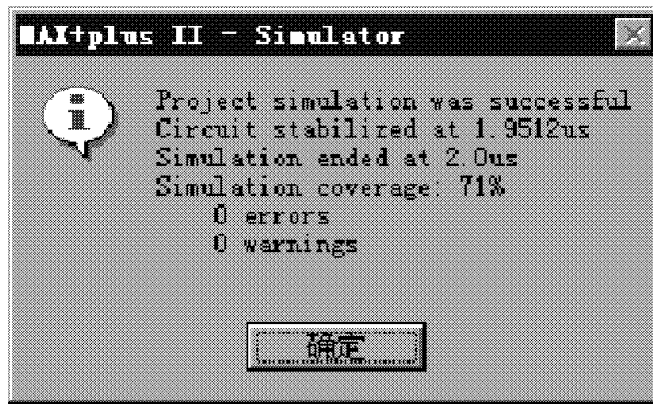


图 3－39　仿真完成提示窗口

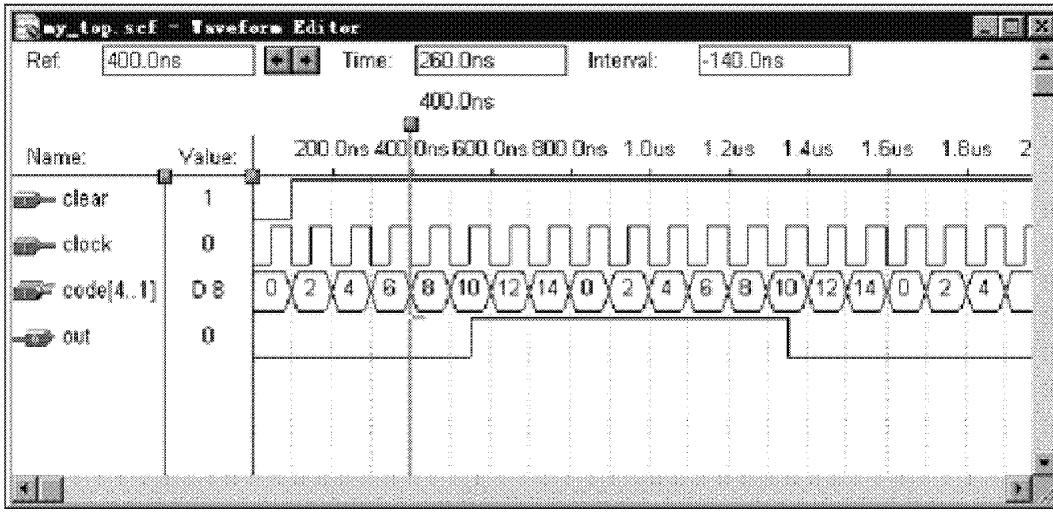


图 3－40　仿真完成后的仿真通道文件

十一、延时分析器

在图 3－40 中，利用窗口左侧的比例显示工具按钮，将波形图放大显示，可以看到在 code 取 10 时，时钟的上升沿与 out 从 0 到 1 的上升沿之间有一段小小的延迟，如图 3－41 所示。利用 MAX＋plus II 中的延时分析器可以详细查看延时情况。

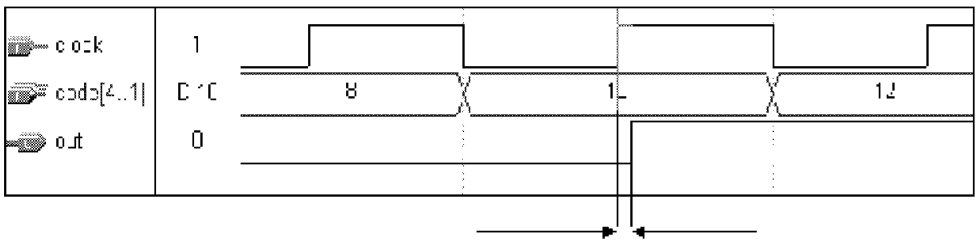


图 3－41　输出延迟的波形显示

选菜单项 MAX＋plus II \ Timing Analyzer，在弹出的延时矩阵窗口中击 Start 按钮，系统分析完毕显示如图 3－42 所示。

由图中可以看出，从输入 clock 到输出 out 间存在 4ns 的延时。在比较复杂的设计中，利用该工具可以观察所有延迟路径，并且依据它对设计进行优化。如对于延时要求严格（延时短）的某部分子设计，可以重新配置资源，使该部分尽量靠近，以降低时间延迟。

十二、器件编程

前面步骤已经完成了 95％的工作。在确认设计正确的基础上，就可以对已选定器件进行编程了。这个过程很快，而且由系统自动完成。本章的例子较小，编程时间约 30s。所以说，一个设计的基础和重点在于电路设计、编译和仿真。

1. 安装硬件设备

在本章第一部分介绍的几种硬件设备里，适合与本章设计实例的是 LP6 卡＋主编程单元（MPU 包括适配器）。在器件编程之前，首先连接好所有硬件设备，并注意适配器类型应与系统编译后选择的器件 MAX7032 相对应。在放置器件时，要注意方向，依照适配器上的示意图

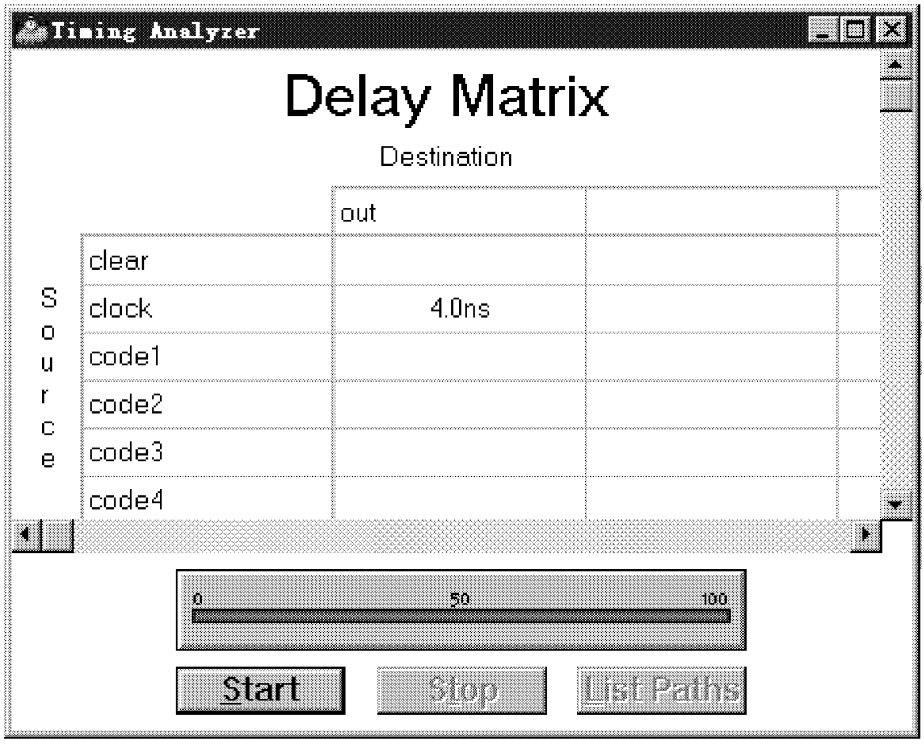


图 3－42　延时分析器

安装。

2. **打开编程器**

做好硬件准备后，进入 MAX＋PLUS II**，选菜单项** MAX＋PLUS II ＼ Programmer**，系统弹出编程器窗口如图 3－43 所示。此时适配器上的显示灯呈黄绿色闪烁。**

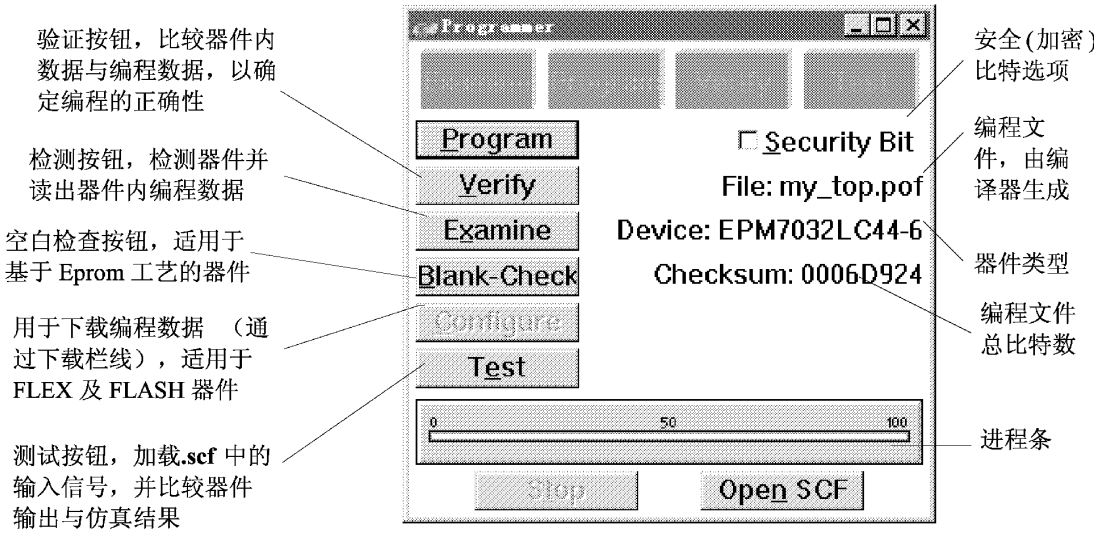


图 3－43　编程器窗口

3. **确认输入文件**

选菜单项 File \ Input /Output. . .，窗口显示如图 3－44 所示，确认输入文件为 my_top.

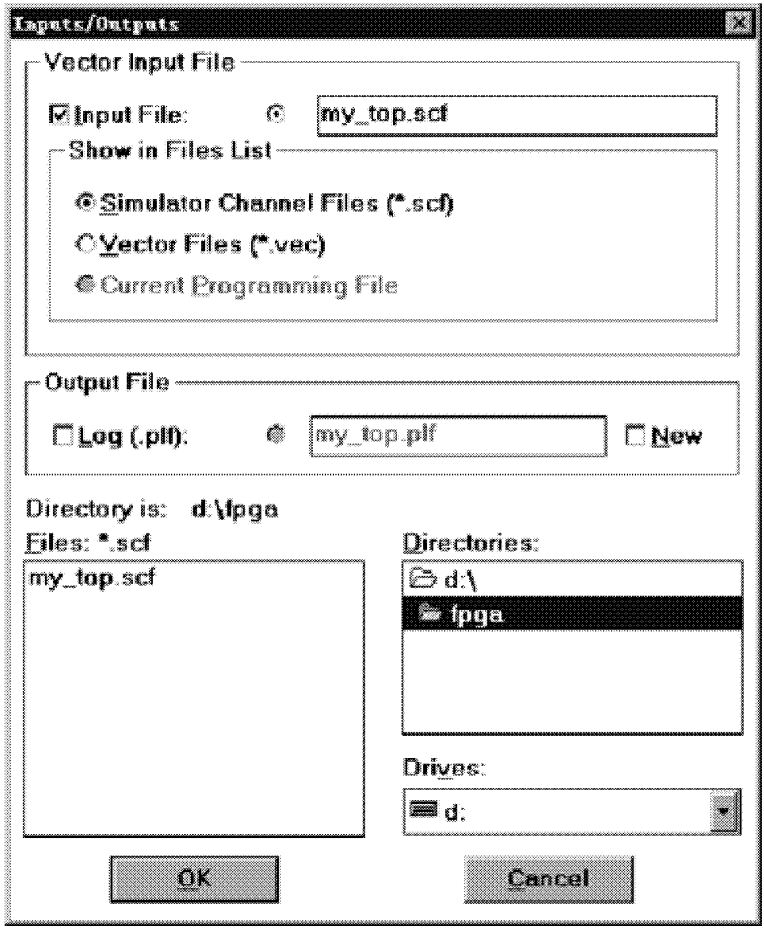


图 3－44　器件编程输入、输出文件

scf，选 OK。

4. 编程

选菜单项 Options \ Programming Options. . .，在图 3－45 选项对话框中选中四个选项后击 OK。在图 3－43 的窗口中击 Program 按钮，这时适配器上的显示灯由黄绿色变为红色，表示编程开始。编程结束后，系统弹出编程成功的提示框，选 OK 即可。

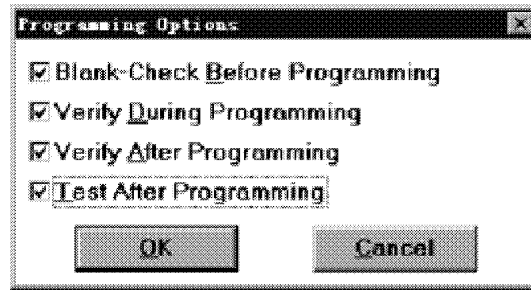


图 3－45　编程器选项对话框

至此，已完成了一个虽然简单但完整的设计。经过以上 12 个步骤，相信读者已对 MAX + PLUS Ⅱ 有了全面的了解，掌握了一些主要的设计工具，熟悉了整体设计流程，下面需要的仅仅是熟练和技巧了。最后还要阐明几个问题。

1. 各种设计文件之间不能相互转化

比如图形设计文件无法转化为文本设计文件，文本设计文件也不能转化为图形设计文件。只能利用符号编辑器或生成包含文件（. inc）分别在图形设计和文本设计中作为底层模块引用。

2. 一个设计并非必须包含各种设计文件

本章中的例子是为介绍各种设计工具而特意将设计分割为三个不同的子设计。对于该例，用图 3 - 6 的图形设计完全可以完成，而且得到同样的结果。同样，也可以全部用文本设计文件来完成，文件清单附在本章末尾。通过全部图形设计和全部文本设计可以进一步熟悉各种设计工具。在实际工作中，具体使用何种工具依个人习惯而定，但应尽量发挥各种设计方式自身的特点。比如图形设计直观易懂，而文本设计在状态机和复杂组合逻辑设计方面具有方便简洁的优点。并且，用硬件描述语言设计电路是当前的发展趋势，尤其是 VHDL 已被定为美国工业标准。MAX + PLUS Ⅱ 兼容 VHDL，因此在开发一些通用模块时，使用 VHDL 可使设计具有更强的通用性和灵活性。另一方面，笔者认为 AHDL 更适合于开发 ALTERA 公司的 FPGA 产品。

附　文本设计文件 sample.tdf **清单：**

```
TITLE "all_txt_with_ahdl";          --标题语句
CONSTANT  code = B"1010";    --定义常量
SUBDESIGN sample
(
        clk,clr,in[4..1]            :INPUT ;
        out                                    :OUTPUT;
)--描述输入输出
VARIABLE
        shifter :dffe;
        enable          :NODE;      --定义变量和中间节点
BEGIN
        DEFAULTS
                enable = GND;
                out = GND;
        END DEFAULTS;       --设定默认值
        shifter.clk = clk;
        shifter.clrn = clr;
        shifter.prn = vcc;          --连接 DFFE 的输入端
        enable = (in[] == code);
        shifter.ena = enable;    --译码电路
        shifter.d = !shifter.q;
        out = shifter.q;            --连接输出
END;
```

第四章　ALTERA 硬件描述语言

4.1 概　　述

ALTERA 硬件描述语言的英文全称是 Altera Hardware Description Language，缩写是 AHDL。AHDL 是 ALTERA 公司根据自己公司生产的 MAX 系列器件和 FLEX 系列器件的特点专门设计的一套完整的硬件描述语言。

AHDL 是一种模块化的高级语言，它完全集成于 ALTERA 公司的 MAX + PLUS Ⅱ（即 Multipul Array Matrix + Programmable Logic User SystemII）的软件开发系统中。AHDL 特别适合于描述复杂的组合逻辑、组（group）运算和状态机、真值表和参数化的逻辑。用户可以通过 MAX + PLUS Ⅱ软件开发系统中的文本编辑器或另外的文本编辑器建立起 AHDL 的文本设计文件，即 Text Design File，简称为 TDF。该文件的扩展名为 . tdf。用户可以通过编译 TDF 文件建立仿真、时域分析和器件编程的输出文件。除此之外，MAX + PLUS Ⅱ编译器还可以产生 AHDL 文本设计的报告文件（后缀为 . tdx）和文本设计输出文件（后缀为 . tdo）。它们可以作为 TDF 文件被存储，而且可以重复使用。正确的设计结果可以对 ALTERA 器件进行下装，从而完成对器件的编程。

AHDL 的语句和元素种类齐全、功能强大而且易于应用。用户可以使用 AHDL 建立完整层次的工程设计项目，或者在一个层次的设计中混合使用 TDF 文件和其他类型的设计文件。AHDL 的 TDF 文件也可以参数化。

在 AHDL 设计文件中，尽管也可以采用任何一种 ASCII 文本编辑器建立 AHDL 设计文件，但是 MAX + PLUS Ⅱ文本编辑器更适合输入、编译和调试一个 AHDL 设计文件取得优良的性能。

一、AHDL 的工作方式

使用 AHDL 所作的设计文件可以非常方便地连入一个设计的某一层次中。在文本编辑器中用户可以自动建立一个符号，用这个符号代表一个 TDF 文件，然后把这个符号加入到一个图形设计文件（后缀为 . gdf）中。同样，用户可以把自己编制的函数和 ALTERA 提供的 300 多个强函数（megafunction）和宏函数（macrofunction），其中包括参数化模型库函数（LPM），通过在文本编辑器中自动建立的一个包含文件（后缀为 . inc）加入到任何一个 TDF 文件中。而且 ALTERA 提供了所有的强函数和宏函数同 MAX + PLUS Ⅱ安装在一起的包含文件。

为分配器件的资源，用户可以使用 Assign 菜单命令或者一个设置文件（Assignment & Configuration File 后缀为 . acf）对 TDF 文件进行资源和器件设置。用户也可以检查 AHDL 句法或者为了调试和处理工程设计项目对所有文件进行编译。在编译过程中出现的任何错误都可以通过消息处理器来定位，并在文本编辑器窗口里以高亮度显示。

图 4－1 示出 TDF 文件是如何集成到 MAX＋PIUSII 系统中的。每个工程设计项目在任何设计层次中都可以包括 TDFs、GDFs、EDIF 输入文件(后缀为. edf),orCAD 图形文件(后缀为. sch),以及 VHDL 设计文件。在通常情况下,波形设计文件(后缀为. wdf)、ALTERA 设计文件(后缀为. adf)、状态机文件(后缀为. smf)和 Xilinx 网表格式文件(后缀为. xnf)只能被用在一个工程设计的低层上。除非整个设计仅有一个单一的波形设计文件、ALTERA 设计文件、状态机文件或 Xilinx 网表格式文件。

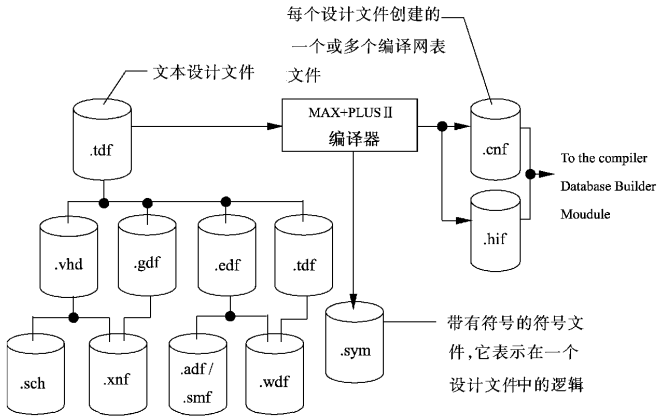


图 4－1　MAX＋PLUS Ⅱ和 AHDL 设计输入

在图 4－1 中,各后缀代表的文件如下：

. tdf——AHDL 文本设计文件(Text Design File)

. vhd——VHDL 设计文件(VHDL Design File),VHDL 是 Very Hign Speed Integrated Circuit (VHSIC)Hardware Description Language 的英文缩写

. gdf——图形设计文件(Graphic Design File)

. edf——EDIF 输入文件(EDIF Input File)。EDIF 是 Electronic Design Interchange Format 的英文缩写.

. sch——orCAD 图形文件(orCAD schematic File)

. xnf——网表格式文件(Xilinx Netlist Format)

. smf——状态机文件(State Machine File)

. wdf——波形设计文件(Waveform Design File)

. sym——符号文件(Symbol File)

. cnf——编译器网表文件(Compiler Netlist File)

. hif——层次互连文件(Hierachy Interconnect File)

在 AHDL 内写的文本设计文件(. tdf)是一个 ASCII 文本文件。它可以用 MAX＋PLUS Ⅱ文本编辑器或任何标准的文本编辑器输入。

下面按照在一个 TDF 文件内出现的顺序列出 AHDL 的段和语句。图 4－2 展示出一个 TDF 文件和它可以包含的 AHDL 段和语句,以及展示如何包含文件和在一个设计层内可以同

AHDL 一起使用的文件。

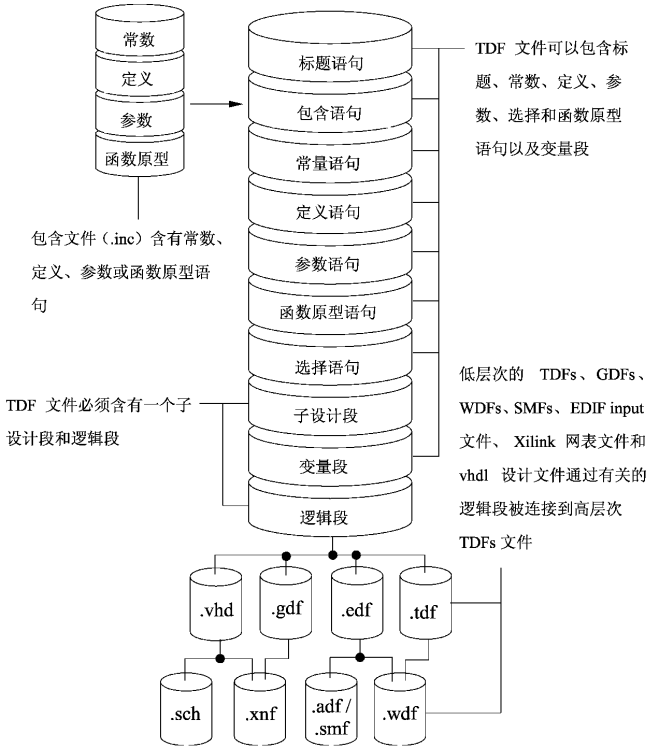


图 4－2　AHDL 文本设计文件结构

1）标题语句（Title Statement）　标题语句为由 MAX＋PLUS Ⅱ编译所产生的报告文件（.rpt）提供解释性文字。该语句可选。

2）包含语句（Include Statement）　在 TDF 文件内包含语句指定一个包含文件。该语句可选。

3）常量语句（Constant Statement）　常量语句定义符号名称。这些符号名称分别代表不同的常数值。该语句可选。

4）定义语句（Define Statement）　定义语句是定义一个运算函数。这个运算函数是一个数学函数。它算出的值是基于选择的自变量。该语句可选。

5）参数语句（Parameter Statement）　参数语句说明一个或多个参数。这些参数控制参数化的强函数和宏函数的执行，也可为每一个参数指定一个默认值。该语句可选。

6）函数原型语句（Function Prototype Statement）　函数原型语句描述一个逻辑函数端口，并且这些端口的顺序在逻辑函数中必须在一个内部直接引用中予以说明。在参数化的函数中，也说明被函数采用的参数。该语句可选。

7）选择语句（Options Statement）　如果该文件是一个顶层 TDF 文件，那么将为该文件或

该设计项目设置默认的位序。该语句可选。

8)断言语句(Assert Statement)　断言语句允许用户测试一个任意表达式的合法性并且报告这个结果。该语句可选。

9)子设计段(Subdesign Section)　子程序段说明一个 AHDL TDF 文件的输入、输出和双向端口。此段必须有。

10)变量段(Variable Section)　变量段用来定义代表和保存内部信息的变量。对于一般的或三态的结点,原型、强函数和宏函数和状态机可以定义为变量。用一个 If Generate 语句也可以有条件地产生变量。该段是可选的。变量段可以包括下列结构的任意一种:

①实例说明(Instance Declaration);

②结点说明(Node Declaration);

③寄存器说明(Register Declaration);

④状态机说明(State Machine Declaration);

⑤状态机别名说明(State Machine Alias Declaration);

⑥如果产生语句(If Generate statement)。

11)逻辑段(Logic Section)　逻辑段用来定义文件中的各种逻辑运算。逻辑段可以用布尔等式、条件逻辑和真值表定义逻辑,它也支持有条件和累接的逻辑,并且可以测试任何表达式的合法性并报告这个结果。此段是必须有的。逻辑段可以包括如下结构的任意种类:

①默认语句(Default Statement);

②断言语句(Assert Statement);

③布尔等式(Boolean Statement);

④情况语句(Case Statement);

⑤For Generate Statement;

⑥If Then Statement;

⑦内部逻辑函数引用(In-Line Logic Function Reference);

⑧真值表语句(Truth Table Statement)。

AHDL 是一种并行语言,在一个 TDF 文件的逻辑段中定义的所有行为并在同一时间内进行运算,而不是相继进行。对于相同的 AHDL 结点或变量,配置多个值的等式时是逻辑连接。(如果结点或变量是高电平有效,那么这些等式之间是‘OR’的关系 如果它们是低电平有效,就是‘AND’的关系。)

一个 TDF 文件必须包含一个子设计段和一个逻辑段。它可以有选择地包括一个单一变量段、选择语句、标题语句和默认语句,以及一个或多个包含语句、常量语句、定义语句和函数原型语句。在一个 TDF 文件里最后记入的是子设计段(必须)、变量段(可选)和逻辑段(必须)。它们合在一起包含了 TDF 文件的行为描述。

在一个设计层次里的文件可以是 TDF 文件、GDF 文件、WDF 文件、ADF 文件、SMF 文件、EDIF 输入文件、ORCAD 图形文件、AHDL 设计文件或 XILINX 网表格式文件。通过每个逻辑函数的输入和输出端口把它们同更高层次上的设计文件相连。

一个包含文件是一个 ASCII 文本文件(后缀为.inc),它可以用一个 AHDL 的包含语句引入一个 TDF 文件。包含文件的内容替代调用这个文件的包含语句。包含文件可以包括函数原型语句、常量语句、定义语句和参量语句。每个 ALTERA 提供的强函数和宏函数具有一个

包含文件，它包含它的函数原型。

关于包含 LPM 函数的强函数，包含文件被定位于在安装期间所建立的 Maxplus2 \ max2lib \ mega-lmp 目录中。宏函数的包含文件被定位于在安装期间所建立的 \ maxplus2 \ max2inc 目录中。

应注意，在 UNIX 工作站上 maxplus2 目录是一个用户目录的子目录。

当用户要在 Graphic、Text 或 Waveform 编辑器窗口打开一个设计文件时，为了自动产生包含关于设计文件的默认函数原型的包含文件，用户可选择 Creat Default Include File (File menu)，也可使用 MAX + PLUS Ⅱ 文本编辑器或其他标准文本编辑器手工创建一个包含文件。

AHDL 文本设计文件(后缀为 . tdf)可以用 MAX + PLUS Ⅱ 文本编辑器或其他文本编辑器遵照标准的 ASCII 字符协议输入。如果用户的文本编辑器有文档和非文档两种模式，那么必须用非文档模式，即只存储文件作为文本。

在用户输入、编译和调试一个 AHDL 文件期间，MAX + PLUS Ⅱ 文本编辑器允许用户取得下面 MAX + PLUS Ⅱ 最好特点：

①AHDL 模板和例子；

②AHDL 范围 – 灵敏帮助；

③句法颜色；

④资源和器件配置；

⑤错误定位。

1. AHDL 模板和例子

为使用户很容易地进行设计输入，MAX + PLUS Ⅱ 提供 AHDL 模板和 AHDL 例子两种方式。用户可以在自己的 TDF 文件中插入 AHDL 模板，而后用自己的标识符和表达式去替换位置模板变量(placeholder variables)。

MAX + PLUS Ⅱ 提供了一些 AHDL 例子。在这个手册里‘ 如何使用 AHDL 章节 ’说明 AHDL 的特点。在 \ max2work \ ahdl 目录内(a subdirectory of the /use directory on a UNIX workstation)和在 MAX + PLUS Ⅱ AHDL 帮助中，这些例子是可以使用的。用户可以按照规格改制这些例子以适应自己的需要。

2. AHDL 范围 – 灵敏帮助

如果当前文件带有后缀 . tdf，那么 MAX + PLUS Ⅱ 文本编辑器提供所有的 AHDL 关键字、运算符、比较符和标识符范围 – 灵敏帮助，以及关于所有 MAX + PLUS Ⅱ 提供的原语、宏函数、强函数。

当用户选择来自工具条或按住 shift + F1 的范围 – 灵敏帮助键  时，指针转向一个寻找标记指针，于是用户可以在 AHDL 内在一个字词和字符上敲击鼠标键 1。如果范围 – 灵敏帮助对于那些项是可用的，那么相互关系信息被显示。另外，帮助显示列出所有的项目。对于这些项目范围 – 灵敏帮助是可以使用的。

3. 句法颜色

MAX + PLUS Ⅱ 文本编辑器允许用户按不同颜色观察一个 TDF 文件的各种元素，句法颜色可以帮助用户改进文法易读性和精确性。例如，在错误旁边给出解释可以帮助用户识别

排错的关键字和文件的段。

为了打开或关闭句法颜色部件,可从选择菜单中选择句法颜色(Syntax Coloring)。用户也可以采用彩色板(Color Palette)命令(选择菜单)给解释、非法字符、强函数和宏函数、保留标识符关键字、行以及文本等设置不同的颜色。

4. 资源和器件配置

用户可以指定资源配置,即引脚、逻辑单元、I/O 单元、嵌套单元、逻辑阵列块(LAB)、嵌入阵列块(EAB)、行、列 、芯片、群集、逻辑选择、连接引脚和定时配置。这些与一个 TDF 文件的器件分配一样去引导逻辑综合和试配用户的设计。用户可以用编译器去选择并自动地试配用户的设计,使其成为目标器件系列中器件的最佳组合,并且在它的内部分配资源。在文本编辑器中也可以选择结点或引脚的名字并且用 Pin / Location / Chip、Cligue、Logic、Option、Timing Reguirrement、Connected Pins 和其他关于分配菜单的命令输入它指定的配置。在所有其他 MAX + PLUS Ⅱ 应用中,分配菜单命令也是可用的。在文本编辑器中用户也可以用平面布置编辑器或用编辑的配置文件输入配置。

例如,用户可以分配逻辑综合方式。它按照用户的需要改编逻辑综合,并且精确地指定如何把一个大的设计分配给多个器件,以及在单一逻辑功能上为达到速度要进行的定时配置。

注意:MAX + PLUS Ⅱ 提供关于 AHDL 运行逻辑选项 Use LPM,允许编译器自动用运算符和比较符代替 lpm-add-sub 和 lpm-compare 功能。这些符号如下:

运算符/比较符	说明
+	加法
−	减法
= =	数值相等
! =	不等于
>	大于
> =	大于或等于
<	小于
< =	小于或等于

(5)错误定位

MAX + PLUS Ⅱ 的报告和确定的各种错误可作为用户修改自己设计的依据。由于用户编译一个设计项目,所以使一个消息处理器窗口打开并且列出关于这个设计的错误信息和警告信息。用户可以通过在消息文本上双击鼠标器左键确定一条消息的来源。于是 MAX + PLUS Ⅱ 自动地打开设计文件错误定位信息内容,它包含消息的来源,而且不关心在设计层次里的位置或创建时应用的类型。如果错误产生在 TDF 文件内,那么 MAX + PLUS Ⅱ 打开文本编辑窗口并且高亮度显示这个引起错误的文本。在 MAX + PLUS Ⅱ 平面布置编辑器中用户也可以确定设计配置平面布置中的错误。

二、编译 AHDL 文本设计文件

MAX + PLUS Ⅱ 自动编译 AHDL TDF 文件。当用户已经完成了一个输入 TDF 文件时,可以用 Project Save&Check 命令(文件菜单)检查它的句法,或者使用 Project Save&Compile 命令(文件菜单)编译设计项目中全部文件。如果用户希望产生一个关于编译设计项目中的 AHDL 文本设计输入文件(. tdo),那么在编译设计之前可以打开编译器的

Generate AHDL TDO File 命令(Processing Menu)。在设计项目编译完成后,用户可以进行仿真和定时分析,而后对一个或多个器件进行编程。

1. 重要规则

下面的重要规则将帮助用户有效地应用 AHDL:

①为帮助用户确定印刷上的错误和 AHDL Code 的各种段,可使用文本编辑器的句法颜色命令(可选项菜单 Option Menu);

②为了改进可读性和避免错误应遵照有关规则指南中的格式和命令说明;

③尽管 AHDL 不是 Case-Sensitive,但为了改进可读性,ALTERA 建议用户遵守在规则指南中指定的用大写字母开头的规则;

④为了改进可读性和避免错误可使用在常量和定义语句中建立的常量和运算函数;

⑤用户不是必须创建原始的 AHDL 函数原型,然而,为了改变输入顺序,用户可以在自己的 TDF 文件中使用函数原型语句重新定义原型;

⑥当可以用一个 Case 语句代替时不要把 If Then 语句套起来使用;

⑦当用户使用 MAX + PLUS Ⅱ 文本编辑器建立一个 TDF 文件时,每行可达到 255 个字符的长度,但理想的行长是用户屏可以容纳的键入到一行末尾的字符数目;

⑧用户能够开始新的一行,无论什么空格(空行、表头和空格)都被允许,在含义上没有任何影响,主要的 AHDL 构造之间的空格都被允许;

⑨关键字、名字和数必须用合适的符号或运算符以及一个或多个空格分开;

⑩解释必须被括在百分号之内(%),解释可以包括除% 之外的任何字符,因此 MAX + PLUS Ⅱ 忽略在百分号之内的任何符号,在百分号之内括起来的解释不能被套入;

⑪当把一个原型连到另一个原型时,用户必须只采用‘合法’的互连,不是全部的原型都可以连到全部另外的原型上;

⑫不要创建用户自己的跨越 - 耦合结构,只采用由 MAX + PLUS Ⅱ 提供的 expdif、explatch、inpltch、nandltch、andnorltch 宏函数(对于 FLEX8000 和 FLEX10k 结构的这些宏函数不能被优化),避免把 expdif、explatch、inpltch、nandltch 和 norltch 宏函数的各种实例连在一起,这种宏函数的各种实例应该用 LCELL 原型分开。

VHDL 规定的解释规则可以被嵌套在% - 解释的规则之中。如果对于文件中的解释类型用户采用 VHDL 的解释规则,那么用户可以采用% - 解释规则去隔离来自编译的码的部分(即码的‘Comment Out’部分)。

为了列出关于原型的合法互连请参看 4.3 节中的‘Primitive/Port interconnection’。

2. 设计输入的重要规则

①在多数情况下用户应该采用 ALTERA 提供的原型和 ALTERA 的逻辑运算符,而不是采用等效的 LPM 函数,因为它们更适合于实例。例如,如果用户在一个指定的全局时钟的上升沿上装载一个寄存器,ALTERA 建议装载寄存器时,用户使用 DFFE、TFFE、JKFFE 或 SRFFE 使能类型触发器的一种时钟使能输入去控制。

②在多数情况下使用 LPM megafunction 而不使用等效的老式 Macrofunction。如果用户改变设计,这种形式更便于利用具体例子说明并且比较容易修改。

③在编译期间使用设计医生去检查用户的设计逻辑的可靠性。要得到可靠的设计信息,在 MAX + PLUS Ⅱ 帮助中找到‘Project Reliability Guidelines’即可。

④不用试图创建用户自己的逻辑函数去实现 RAM 或 ROM，应采用 ALTERA 提供的 Megafunctions 代替 RAM 或 ROM。

⑤当用户开始一个新的设计文件时，应立刻使用 Device (Assign Menu)指定的目标器件系列中的最好器件。如果用户不指定一种器件系列，那么就假定是当前设计的系列。

3. MAX + PLUS Ⅱ 的重要规则

①当用户在一个新的设计文件上开始工作时，使用 Project Set Project to Current File 或 Project Name (File Menu)给当前设计文件命名，以便能够很容易地编译它，以后用户也可以改变这个文件名。

②为了在当前层次树的设计文件之间移动，可使用 MAX + PLUS Ⅱ 中的 built-in hierachy traversal features。为打开底层文件，可打开顶层文件而后使用层次显示窗口或 Hierachy Down (File Menu)。如果用户选择 Open 或 Retrieve (File Menu)去打开这个底层文件，那么，文件被认为是不同层次树、资源、器件和试样分配的顶部，但仅相对于那个层次存储用户的输入，而不对整个设计。

③当用户创建一个对设计来说是可编辑的辅助文件时，如果用户采用和设计相同的名字，那么关于这个文法的插图将出现在层次显示里。

④不要编辑任何 MAX + PLUS Ⅱ 系统文件，包括 HIFs、TOK files、maxplus2. idx files 或 maxplus2. in files。

⑤如果用户希望重新命名一个设计文件或附属文件，应用 Save As 命令(File Menu)。对于来自 MAX + PLUS Ⅱ 系统外边的设计文件不能重新命名。

⑥当用户作完一个设计项目时，为制作所有设计文件的完整拷贝，可应用文件菜单中的 Project Archive 进行。这样，将来对文件进行编辑或删除时不会影响原设计文件。

4.2 基本的 AHDL 设计结构

用 AHDL 描写数字逻辑电路的一个最简单的 TDF 文件必须包括一个子设计段(Subdesign Section)和一个逻辑段(Logic Section)，其他段和语句都是可选的，而不是必需的。下面将按照各种段和语句在 TDF 文法中出现的顺序给出各种段和语句的详细说明。

为了叙述方便，表 4 - 1 给出 BNF(Backus Naur Form)使用的注释符号。BNF 定义了文本文件和信息变量的句法。

表 4 - 1 BNF 的注释符号

字　符	说　明
:=	被定义为
< . . . >	标识符(即变量)
[. . .]	可选项
{ . . . }	重复项(零或另外的项)
. . . | . . .	在各项间指定的一个选择

续表

字　　符	说　　明
:n :n	后缀,指定一个范围(例如〈name char〉:1 :8 含义是“从 1 到 8 名字书写符号”)
斜体字(italics)	句法说明里的变量
(courier font)	句法说明里的文字的文本。为帮助识别来自句法说明里斜体变量文字的文本,前面采用粗体 courier font

4.2.1 标题语句

标题语句允许用户为编译器产生报告文件(.rpt)提供文档注释。标题语句的 BNF 语法规则如下:

〈标题〉::= TITLE〈字符串〉;

例如:

TITLE "Display Controller";

标题语句的特点如下:

①标题语句用关键字 TITLE 开始,后面跟随一个括在双引号(“”)标记内的字符串,最后用一个分号(;)结束;

②如果在一个 TDF 文件内包括一个标题语句,那么标题语句呈现在报告文件的顶部,如上例所示,标题 Display Controller 出现在报告文件的顶部。

在使用标题语句时必须遵守下面规则:

①如果在标题内需要引号标记,就必须使用两个双引号,例如:

TITLE " "EPM5130" " Display Controller";

②标题语句在一个 TDF 文件内只能使用一次;

③标题语句必须放在所有其他 AHDL 段之外。

4.2.2 参数语句

参数语句允许用户说明一个或多个参数,这些参数控制一个参数化的强函数或宏函数的执行。

参数语句的 BNF 语法规则如下:

〈参数语句〉::=

PARAMETERS

(

〈参数〉=〈可选默认值〉——解释内容

.

.

.

);

例如:

PARAMETERS

(

```
    FILENAME = "myfile.mif", _optional default value follwe " = "sign
    WIDTH,
    AD_WIDITH = 8,
    NUMWORDS = 2^AD_WIDTH
);
```

参数语句的特点如下：

①一个参数语句由关键字 PARAMETER 开始，随后列出一个或多个参数以及可选的默认值，并括在圆括号()中；

②参数表中的参数用逗号(,)分开，用一个等号把参数名同默认值分开，如上例所示，其中仅参数 WIDTH 没有设置默认值；

③参数名可以是用户定义的符号名或者再定义的 ALTERA 参数；

④参数值可以由排成一行括在双引号标记(" ")内的文本组成，作为一行计算，当参数值不被括起来时，编辑器试图作为算术表达式看待它们，如果没有参数值，作为一行看待；

⑤语句的末尾用一个分号(;)；

⑥参数每被定义一次，用户可以用它贯通 TDF 文件。

参数语句必须遵守下列规则：

①参数只能用于对它的说明之后；

②每个参数名必须是唯一的；

③参数名不能包含空格，对于分离的字符可使用下画线增加易懂性；

④参数语句可以在 TDF 文件中使用任意次数；

⑤参数语句必须放在所有其他 AHDL 段的外边；

⑥在其他参数的定义里所用的参数必须预先定义；

⑦不允许出现循环。

下面例子示出了一个循环：

```
PARAMETERS
(
    FOO = BAR;
    BAR = FOO;
);
```

当一个设计被编译时，编译器按下列顺序检查参数值：

①作为逻辑函数的实例部分，例如，在一个 TDF 文件内，或在一个实例说明或内部直接引用建立的实例里，用户可以说明被采用的参数并有选择地分配它的值；

②在一个 GDF 文件内，用户可以选择一个符号并且使用 Edit Ports/Parameter 命令(Symbol Menu)去配置关于实例的参数值；

③在下面高层次级上作为逻辑函数的一个实例的端口，如果子设计实例没有给出配置的参数值的话，那么一个逻辑函数的实例参数值就适用于那个逻辑函数的子设计；

④在全局设计内默认参数值用 Global Project Parameter 命令(Assign Menu)，这些值存储在设计的设置文件内(.acf)；

⑤在选择的默认值里列入定义逻辑函数的 TDF 文件的参数语句或者 GDF 文件的 PA-

RAM 原语，这些默认值只用于在文件里列出它们的那个文件，在子设计的文件里不用它们。

4.2.3 包含语句

包含语句允许用户由一个包含文件(.inc)引到当前文件输入文本。包含语句的 BNF 语法规则如下：

〈include 语句〉::= INCLUDE〈文件名〉;

例子：

INCLUDE“const.inc”;

包含语句具有如下特点：

①用关键字 INCLUDE 作为包含语句的开始，随后跟着所要包含的括在双引号(“ ”)中的文件名；

②如果用户未指明文件的扩展名，编辑器将假定这个扩展名为.inc；

③语句用分号结尾(;)；

④当编译器处理设计的时候，来自包含文件的文本代替包含语句，如上例所示，文件 const.inc 代替文本 INCLUDE“const.inc”。

在一个 TDF 文件里，对低层设计文件，为包含函数原型经常使用包含语句。为使用强函数或宏函数，用户必须首先在一个设计文件里定义它的逻辑，而后必须使用函数原型语句指定函数的端口。为包含在包含文件里存储的函数原型，用户可以使用包含语句。于是用户可以用一个实例说明或在内部直接引用代替一个逻辑函数的实例。

用户可以使用 Creat Default Include File(File Menu)自动产生一个包含文件。它包含一个关于设计文件的函数原型。

当用户编译一个文件时，编译器按照下面的顺序检查用户计算机关于包含文件的目录：

①设计目录；

②使用 User Libriaries (Option menu)指定任何用户库；

③在安装期间建立 \maxplus2\max2lib\mega-lpm 和 \maxplu2\max2inc 目录。

如果用户改变包括一个包含文件的 TDF 文件，那么用户可以使用 Project Save&Check (File menu)或者为了修改在层次显示窗口里显示的设计层次树的形式，全部重新编译设计。

包含语句在使用中必须遵守以下规则：

①在包含语句中指定的文件名不能含有通道名；

②在 MAX+PLUS Ⅱ中，文件名可以被列在上层和下层函件内，然而，在一个包含语句内文件名的层必须与包含文件名的层相配；

③一个包含语句必须放在所有其他 AHDL 段的外边；

④一个包含语句可以在一个 TDF 文件中出现任意次。

在工作站环境中，文件名是层敏感的。ALTERA 提供的宏函数、强函数设计文件全部是低层文件名。因此，它们的相应包含文件用低层文件列出函数名。

包含文件在使用中必须遵守下列规则：

①包含文件名必须具有扩展名.inc；

②包含文件只能包含函数原型、定义、参数或常量语句；

③包含文件不能包含子设计段；

④包含文件不能被嵌套。

4.2.4 常量语句

常量语句允许用户用一个有意义的符号名代替一个数或一个算术表达式。这个符号名可以简单地表示那个数。

常量语句的 BNF 语法规则如下：

〈常量语句〉::= CONSTANT〈符号名〉=〈数值〉；

例子：

```
CONSTANT    UPPER_LIMIT = 130；
CONSTANT    BAR = 1 + 2DIV3 + LOG2(256)；
CONSTANT    FOO = 1；
CONSTANT    FOO_PLUS_ONE = POO + 1；
```

常量语句的特点如下：

①常量语句用关键字 CONSTANT 开始，后跟一个符号名、一个等号(=)和一个数(如果需要可以包括一个基数)或者一个算术表达式；

②常量语句用一个分号(;)结尾；

③一个常量被说明一次，用户就可以表示整个 TDF 文件的数，如上例所示，用户可以用在逻辑段里 UPPER_LIMIT 表示十进制数 130；

④常量可以用一个算术表达式说明，这个算术表达式可以包括预先定义的常量。

注意 编译器计算常量语句内的算术表达式，并且用数字表示它们的值。这种表达式不产生逻辑。

常量语句必须遵守下述规则：

①常量只有被说明之后才能被使用；

②每个常量名必须是独立的；

③常量名不能含有空格，应该使用下画线去分隔字符，这样也增加了可读性；

④在一个 TDF 文件里常量语句可以被使用多次；

⑤常量语句必须被放在所有其他 AHDL 段的外边；

⑥用在另外常量的定义的常量必须预先被定义；

⑦循环关系是不允许的，下面的例子示出一个循环关系：

```
CONSTANT    FOO = BAR；
CONSTANT    BAR = FOO；
```

4.2.5 定义语句

定义语句允许用户定义一个运算函数，这个运算函数是根据选择的自变量产生一个值的数学函数。定义语句的 BNF 语法规则如下：

〈定义语句〉::=

DEFINE〈符号名〉(一个或用逗号分开的多个自变量) = (运算表达式)；

例子：

```
DEFINE    MAX(a, b) = (a > b) ? a  : b；
```

```
SUBDESIGN
(
   data a [MAX (WIDTH, 0)..0] : INPUT;
   data b [MAX (WIDTH, 0)..0] : OUTPUT;
  )
  BEGIN
   data b[ ] = data a [ ];
  END;
```

上面的例子定义运算函数 MAX,它保证子设计段在至少一个端口上声明。

定义语句具有下述特点:

①定义语句由关键字 DEFINE 开始,随后是一个符号名和括在圆括号(())内的一个或多个自变量的表;

②在自变量表中的自变量用逗号(,)分开,等号把自变量表和运算表达式分开。

在此应注意以下几点:

①如果没有列出自变量,那么运算函数的行为如同一个常量;

②编译器计算定义语句里的算术表达式并减少他们为表示值所用的数字,对于这个表达式没有逻辑被产生;

③语句末尾用一个分号(;);

④每定义一次运算函数,用户可以用它贯通整个 TDF 文件;

⑤可以把运算函数定义在预先定义的运算函数的项内。

例如,下面的 MIN_ARRAY_BOUND 函数是建立在上面的 MAX 函数定义的基础上的:

```
DEFINE MIN_ARRAY_BOUND(x) = MAX(0, x) + 1;
```

定义语句必须遵守下面规则:

①一个运算函数只能被用在它被定义之后;

②每个运算函数必须是单一的;

③运算函数名不能包含空格,而用下画线分隔‘字符”,从而增加了可读性;

④在一个 TDF 文件内定义语句可以被使用任意次数;

⑤定义语句必须放在所有其他 AHDL 段的外面。

4.2.6 函数原型语句

函数原型语句与在图形设计文件里的符号具有相同的功能,二者皆提供一个逻辑函数的简略描述,并列出它的名字和它的输入、输出以及双向端口。状态机端口可以在函数中作为输入端或输出端状态机。

然而,由于强函数和宏函数输入端口是在 MAX + PLUS Ⅱ图形编辑器文件里,所以它们的默认值不是自动配置。用户必须在一个 TDF 文件的子设计段里明确配置它们。在子设计段里用户也可以给双向端口配置一个默认值。可是,输出端不能被配置一个默认值。

当用户希望执行一个强函数或宏函数的实例时,用户必须保证它的逻辑被定义在它自己的设计文件中,于是用户应用一个函数原型语句去指定这个函数的端口,并且用一个内部直接引用或一个实例说明执行一个函数的实例。

函数原型语句的 BNF 语句规则如下。

1. 参数化的函数

〈函数原型〉::=

FUNCTION〈函数〉(〈输入表〉)

WITH (〈参数表〉)

RETURNS (〈输出表〉);

2. 非参数化的函数

〈函数原型〉::=

FUNCTION 〈函数〉(〈输入表〉)

RETURNS (〈输出表〉);

FUNCTION 〈原语〉(〈输入表〉)

RETURNS (〈输出表〉);

〈函数〉::=〈设计名称〉

〈原语〉::=〈符号名〉

〈输入表〉::=〈〈函数原型组〉{,〈函数原型组〉}

〈输出表〉::=〈〈函数原型组〉{,〈函数原型组〉}

〈参数表〉::=〈参数名〉{,〈参数名〉}

例 1:

```
FUNCTION  lpm_add_sub ( cin,dataa[ LPM_WIDTH - 1..0],
datab [LPM_WIDTH - 1..0 ], add_sub)
WITH  (LPM_WIDTH, LPM_REPRESENTATION, LPM_DIRECTION, ADDER-
TYPE, ONE_INPUT_CONSTANT)
RETURNS (result [LPM_WIDTH - 1. .0 ],cout ,overflow );
```

例 2:

```
FUNCTION  compare (a [3..0 ], b [3..0 ])
RETURNS (less , equal , greater );
```

上面的例子示出函数原型语句。第一个例子是参数化的函数;第二个例子是非参数化的函数。

函数原型语句具有的特点如下:

①函数名字跟在关键字 FUNCTION 之后,如上例所示,函数的名字是 lpm_add_sub 和 Compare;

②函数的输入端口表跟在名字之后,如第一个例子所示,输入端口是 cin、dataa[LPM_WIDTH - 1.. 0]和 datab [LPM_WIDTH - 1.. 0],在第二个例子里,它们是 a3、a2、a1、a0、b3、b2、b1 和 b0;

③在一个参数化的函数里,关键字 WIDTH 和参数名列在输入端口表之后,这个表被括在圆括号()之内,指定的参数名用逗号(,)分开;

④输出表和函数的双向端口跟在关键字 RETURN 之后,如上面第一个例子所示,输出端口是 result [LPM_WIDTH - 1.. 0]、cout 和 overflow,在第二个例子里,它们是 less、equal 和 greater;

⑤输入和输出两种表都被括在圆括号内,指定的端口名用逗号分开;

⑥当输入或输出是一个状态机时,对于文件的函数原型必须使用一个状态机端口(用 MACHINE 关键字识别)标明哪些输入和输出是状态机;

⑦函数原型语句用一个分号结尾;

⑧在一个 TDF 文件中,一个函数原型语句必须被放在子设计段的外面,并且在内部直接引用或实例说明里必须把它放在被用具体例子说明的逻辑函数之前。

例如:

```
FUNCTION ss_def ( clock,  reset, count )
RETURNS  (MACHINE  ss_out);
```

为实现一个原语的实例,用户也可以使用内部直接引用或者实例说明。然而,在对强函数和宏函数的对比中,原语逻辑被预先定义,因此在各种设计文件里用户不需要去定义原语的逻辑。另外,用户不需要去使用函数原型语句,除非用户希望改变原语的输入顺序。

下面的例子示出关于一个 JKFF 原语的默认函数原型:

```
FUNCTION  JKFF(j, k, clk, clrn , prn )
RETURNS (q);
```

下面的例子示出关于一个 JKFF 原语修改的函数原型:

```
FUNCTION  JKFF(k, j, clk, clrn, prn)
RETURN (q);
```

当改变文件内使用的函数原型语句时,用户可以采用一个包含语句去调用含有函数原型语句的包含文件(.inc)。MAX + PLUS Ⅱ 也提供了 Creat Default Include File 命令(File menu)。对任何设计文件,该命令皆自动创建包含一个函数原型的包含文件。

所有 MAX + PLUS Ⅱ 强函数和宏函数的函数原型皆被存储在 \ maxplus2 \ max2lib \ mega-lpm 和 \ maxplus2 \ max2.inc 目录下的包含文件中。所有强函数和宏函数和原语的在线帮助皆显示出每一个 ALTERA 提供的函数的函数原型。

4.2.7 选择语句

选择语句的作用是对 BITO 选项进行设置,以指定组的最低数字位是最高有效位(MSB)、最低有效位(LSB)还是其他。选择语句的 BNF 语法规则如下:

〈OPTIONS 语句〉::= OPTIONS BITO = (ANY | LSB | MSB)

例如:

```
OPTIONS  BITO =  MSB;
```

在这个例子里,一个组的最低位被指定为 MSB,其他可用的设置是 LSB 和 ANY。

选择语句的特点如下:

①选择语句以关键字 OPTION 开始,后跟 BITO 选项及其设置,并以分号结束;

②BITO 即组的最低位可被设置为 MSB(最高有效位)、LSB(最低有效位)和 ANY;

③选择语句位于 TDF 文件的开始,它为整个文件设置了位的默认顺序。

如果整个文件是最顶层的 TDF 文件,那么选择语句将作用于整个文件 如果整个文件在设计的层次结构中处于较低层,那么选择语句所设置的位序只作用于这个文件。

4.2.8 断言语句

断言语句允许用户测试任何任意表达式的有效性。断言语句的 BNF 语法规则如下：

〈Assert 语句〉：：=

ASSERT(〈运算表达式〉)

REPORT(“消息行”)〈消息变量〉)

SEVERITY ＜ERROR，MARNING 或 INFO＞

HELP_ID ＜INIVALUE＞

例子：

```
ASSERT (WIDTH>0)
REPORT  "Width (%) must be a positive integer" WIDTH
  SEVERITY     ERROR
  HELP_ID    INIVALUE ;  -- for internal Altera use only
```

断言语句具有如下特点：

①关键字 ASSERT 后跟一个运算表达式，这个可选运算表达式被括在圆括号内。当这个表达式不对时，断言被激活并且跟随 REPORT 关键字的消息行被显示在消息处理器里。如果用户没指定条件，那么断言总是有效的。

②REPORT 关键字后跟一个消息行和可选的消息变量，消息行被括在双引号标记(“ ”)内，并且可以包括用可选消息变量的值代替％字符。如果没有使用 REPORT 关键字，那么被激活的断言在消息处理器里显示下面消息：

＜severity＞：Line ＜line number＞ ，File ＜filename＞ ：Assertion failed

③可选的消息变量由一个或多个参数、计算函数和运算表达式组成。多个消息变量由逗号分开。括起来的消息行内的％字符按顺序代替消息变量的值。如上例所示，括在消息行内的％代替 WDITH 的值。

④可选的 SEVERITY 关键字后跟 ERROR、WARNING 或 INFO，如果没有 SEVERITY 被指定，那么它默认 ERROR。

⑤HELP_ID 关键字和帮助行被用在一些 ALTERA 提供的逻辑函数内，并且在内部的应用中被保留。

⑥这种语句的结尾用一个分号。

⑦在逻辑段的内部或外边以及任何其他 AHDL 段都可以使用断言语句。

4.2.9 子设计段

子设计段描述 TDF 文件的输入、输出和双向端口。子设计段的 BNF 语法规则如下：

〈子设计段〉::=

SUBDESIGN〈设计名称〉

(

{〈信号表〉;}

〈信号表〉[;]

)

[〈变量段〉]
BEGIN
〈语句组〉
END；
〈信号表〉::=〈端口表〉:〈端口类型〉
〈端口类型〉::=
INPUT[=VCC | =GND]
| OUTPUT
| BIDIR[=VCC | =GND]
| MACHINE INPUT
| MACHINE OUTPUT

下面的例子示出一个子设计段：

```
SUBDESIGN  top
   (
    foo, bar, clk1, clk2  :          INPUT = VCC;
    a0, a1, a2, a3, a4    :          OUTPUT;
    b[7 . . 0]            :          BIDIR;
    )
```

子设计段具有如下特点：

①关键字 SUBDESIGN 后跟子设计名，子设计名必须与 TDF 文件名相同，上例的设计名是 top；

②信号表被括在圆括号里；

③用符号名表示信号名，例如 foo，并且被分配一个端口类型，如 INPUT；

④各信号名用逗号分开，后跟一个冒号和一个端口类型，并且用一个分号结束；

⑤端口类型可以是 INPUT、OUTPUT、BIDIR、MACHINE、INPUT 或 MACHINE OUTPUT；(如上例所示 foo、bar、clk1 和 clk2 信号是输入端口，a0、a1、a2、a3 和 a4 是输出端口，总线 b[7. .0]是双向的端口)

⑥在 TDF 文件和另外的设计文件之间，输入端口和输出端口状态机使用 MACHINE INPUT 和 MACHINE OUTPUT 关键字，然而，MACHINE INPUT 和 MACHINE OUTPUT 端口类型不能被用在一个顶层的 TDF 文件内；

⑦在端口类型后面(另外，没有默认值被分配)用户可以有选择地分配一个默认值 GND 或 VCC(如上例所示，VCC 是输入信号的默认值)。

在顶层的设计文件里 INPUT、OUTPUT 和 BIDIR 端口类型表示实际器件的引脚。在低层次设计文件内所有端口类型都是该文件的输入和输出端口，但不是整个设计的。

4.2.10 变量段

可选的变量段用于说明和产生用在逻辑段的任何变量，AHDL 变量类似于在高级可编程语言里的变量，它们用来定义内部的逻辑。

变量段的 BNF 语法规则如下：

〈变量段〉::=

VARIABLE

〈端口表〉:〈变量类型〉;

{〈端口表〉:〈变量类型〉;}

〈变量类型〉::= NODE |〈函数〉|〈原语〉|〈状态机〉|〈状态机别名〉

〈函数〉::=〈设计名称〉

〈原语〉::=〈符号名〉

下面的例子示出一个变量段：

```
VARIABLE
    a,b,c       :NODE;
    temp        :halfadd;
    tsnode      :TRI_STATE_NODE;
  IF DEVICE_FAMILY= ="FLEX8000" GENERATE
      8Kadder  :flex_adder;
  ELSE GENERATE
      7Kadder:  pterm_adder;
      f,g         :NODE;
 END  GENERATE;
```

变量段可以包括一个或多个下列语句：

①实例说明；

②结点说明；

③寄存器说明；

④状态机说明；

⑤状态机别名说明。

应该注意，变量段也可以包括 If Generate 语句，可以用它产生实例、结点、寄存器，状态机和状态机别名说明。

变量段具有如下特点：

①用关键字 VARIABLE 开始这个变量段；

②用户定义的各变量名之间用逗号分隔，变量与变量类型之间以冒号分隔；

③在变量表内每种输入都用一个分号结束。

变量类型可以是结点(NODE)、三态结点(TRI_STATE_NODE)、Primitive、megafunction、macrofunction 或 state machine declaration。如上例所示，内部变量是结点类型的 a、b 和 c；temp 是 macrofunction 半加器的一个实例；tsnode 是三态结点的一个实例。

最后应注意，编译器产生的包含波动符(~)的名字可以出现在一个设计的 Fit 文件(.fit)内。如果用户在后面注释 Fit 文件配置，这些名字将出现在设计的配置文件内(Assignment&Configuration File--.acf)。波动符只对编译器产生的名字才被保留，用户不能在自己的管脚、结点和组(总线)名字里使用它。

4.2.11 实例说明

在变量段里,当一个变量带有一个实例说明时可以说明一个实际的逻辑函数的每种独立应用或实例。在它被说明之后,正如在逻辑段内的端口那样用户可以应用每个逻辑函数的输入和输出端口。

当用户想实现一个强函数或宏函数实例时,必须保证它的逻辑被定义在自己的设计文件内,而且使用一个函数的原型语句去指定这个函数的各种端口和参数,并用一个内部直接引用或者一个实例说明实现一个函数的实例。

为执行一个原语的实例,用户也使用内部直接引用或实例说明。然而,在同强函数和宏函数对比中,原语逻辑是被预先定义的,因此用户不需要在单独的设计文件内去定义原语逻辑。在大多数情况下,不需要函数原型语句。

为用实例说明,在变量段中用户申述一种关于<Primitive>、<megafunction>或<macrofunction>的变量。对参数化的 megafunction 或 macrofunction,这种说明包括用实例和可选项参数值列出的参数表。用户可以应用下列格式中的实例端口:

<instance name> . <port name>

例如,如果用户希望把 compare 和 adder 函数合并到用户的当前 TDF 文件里,那么在变量段里应做以下实例说明:

```
VARIABLE
comp:  compare;
adder:   lpm_add_sub   WITH(LPM_WIDTH=8)
```

变量 comp 和 adder 是函数 compare 和 lpm_add_sub 的实例。它们有下面的输入和输出:

```
a[3..0],b[3..0]              :INPUT; --inputs to compare
less,equal,greater           :OUTPUT; --outputs of compare

a[8..1],b[8..1]              :INPUT;  --inputs of adder
sum[8..1]                    :OUTPUT; --outputs of adder
```

因此用户在当前逻辑段可以应用下面的 comp 和 adder 的端口:

```
comp.a[], comp.b[], comp.less, comp.equal, comp.greater
adder.data.a[], adder.datab[],adder.result[]
```

这些端口可以和结点一样在任何行为语句中使用。

所有原语只有一个输出端口。如果用户想使用它的输出,可以在等式右边使用没有端口名的原语的名字(例如,没有.q 或 .out)。同样,对于所有具有一个单一原始输入(即除 JKFF、JKFFE、SRFF 和 SRFFE 之外的所有原语)的所有原语,为把原语连接到原始输入上,用户可以在等式左边使用没有端口名的原语的名字。

4.2.12 结点说明

AHDL 支持两种类型的结点:NODE 和 TRI_STATE_NODE。两种类型是全目的变量类型,它们用于存储在子设计段和别处变量段内没有被说明过的信号。因此,每一种变量可以被用在一个等式的左边或右边。

结点和三态结点类似于子设计段的输入、输出和双向端口类型,在此它们也代表一个传输信号的信号线。

应该注意,编译器产生的包含波动(~)符的名字可以出现在所设计的 Fit 文件内(.fit)。如果用户后注释这种 Fit 文件配置,那么这些名字将出现在设计的配置文件里(.acf)。只对编译器产生的名字才保留波动符。在用户自定的引脚、结点和组(总线)的名字里不能用它。

下面的例子示出一个结点说明:

```
SUBDESIGN node_ex
(
  a,oe        :INPUT;
  b           :OUTPUT;
  c           :BIDIR;
)
VARIABLE
  b           :NODE;
  t           :TRI_STATE_NODE;
BEGIN
  b=a;
  out=b   % therefore   out=a %
  t=TRI(a,oe);
  t=c;  % t is bus of c and tri_stated a %
END;
```

结点和三态结点在各种配置方面的差别使它们得出不同的结果:

①对于结点类型(NODE)的各个结点,各种配置通过"线与"或"线或"功能把信号合并在一起。在默认语句内说明的变量的默认值确定这种行为:一个 VCC 默认产生一个"线与"功能;一个 GND 默认产生一个"线或"功能。

②对于三态结点(TRI_STATE_NODE)的多种赋值是把多个信息连接到相同的结点上。

③如果只给三态结点(TRI_STATE_NODE)分配一个变量,那么就用结点(NODE)代替它。

下面的各种原语和信号可以馈给三态结点(TRI_STATE_NODE):

①TRI 原语;

②来自一个较高层次上的设计文件的各种输入(INPUT)端口;

③来自一个较低层次上的设计文件的输出(OUTPUT)和双向(BIDIR)端口;

④当前文件的双向(BIDIR)端口;

⑤在当前文件内作为三态结点类型描述的其他结点。

4.2.13 寄存器说明

寄存器说明包括 D、T、JK、SR 触发器(即 DFF、DFFE、TFF、TFFE、JKFF、JKFFE、SRFF 和 SRFFE)和锁存器(LATCH)的说明。下面的例子示出一种寄存器说明。

VARIABLE:

```
ff      :TFF;
```

这个 T 触发器实例的名字是 ff。在做了这种说明之后，用户可以按照格式〈instance name〉.〈port name〉使用 ff 实例的输入和输出端口：

```
ff.t
ff.clk
ff.clrn
ff.prn
ff.q
```

由于所有原语都只有一个输出，所以如果用户想使用它的输出端，在等式右边可以采用一个不带附加端口名(例如 没有.q 或.out)的原语的实例名。同样，对于所有具有单一原始输入的原语(除了 JKFF、JKFFE、SRFF 和 SRFFE 之外)，为了把它们的原始输入连到原语上，用户可以在等式的左边使用不带端口名的原语实例名(例如 不带.d，.t 或 .in)。

例如，若 DFF 函数原型为

```
FUNCTION  DFF(d, clk, clrn, prn) RETURNS(q);
```

那么在下面的 TDF 文件中有

```
VARIABLE
        a, b    :DFF;
BEGIN
        a = b;
END;
```

这里，在逻辑段中 a = b(不带端口名)与 a.d = b.q(带端口名)的功能是相同的。

4.2.14 状态机说明

用户创建一个状态机时，必须在变量段内说明状态机的名称、状态和可选的状态位。

状态机说明的 BNF 语法规则：

〈符号名〉:〈状态机〉;

〈状态机〉::= MACHINE[〈位组〉]〈状态组〉

〈位组〉::= OF BITS (〈端口表〉)

〈状态组〉::= WITH STATES(〈状态〉{,〈状态〉})

〈状态〉::=〈符号名〉{=〈状态值〉}

〈状态值〉::=〈数值〉|〈符号名〉

下面给出一个状态机说明的例子：

```
VARIABLE
        ss      :MACHINE
                  OF BITS(q1, q2, q3)
                   WITH  STATES(
                        s1 = B "000".
                        s2 = B "010",
                        s3 = B "111");
```

这个状态机的名字是 ss,状态机位 q1、q2 和 q3 是这个状态机的寄存器输出,状态机的状态是 s1、s2 和 s3,每个状态机都给状态位 q1、q2 和 q3 赋予一个状态值。

状态机说明具有如下特点:

①状态机的名字是一个符号名,如上例所示,是 ss;

②用一个冒号(:)和关键字 MACHINE 跟在状态名的后面;

③状态机说明必须包括一个状态表,也可包括状态位名称表;

④用关键字 OF BITS 指定可选的状态位,其后跟着用逗号分隔的一组符号名,这组符号名必须被括在圆括号内,上例示出指定状态位是 q1、q2 和 q3;

⑤用关键字 WITH STATES 指定状态 s1、s2 和 s3;

⑥在 WITH STATES 子句里列出的第一个状态是对状态机的复位状态;

⑦状态名可以有选择地被赋值,方法是用一个等号(=)跟随一个数值,如上例所示,s1 被赋值为 B"000",s2 被赋值为 B"010",s3 被赋值为 B"111";

⑧用户可以采用一个状态机的别名说明,如在下面描述的那样,给一个状态机配置一个变化的名称,它在当前文件里定义或从另外文件引入;

⑨用一个分号(;)结束一个状态机说明。

应该注意的是:用触发器输出信号高和低的一个唯一模式表示一个状态机的每一个状态。状态位是用状态机存储这个状态所需要的一些触发器。状态数同状态机里状态位的数有如下关系:

<number of states> <= 2^<number of state bits>

4.2.15 状态机别名说明

用户可以在变量段中用状态机别名说明语句为状态机起一个临时名字。这样,用户既可以在建立状态机的文件中使用状态机别名,也可以在 MACHINE INPUT 端口引入状态机的文件中使用状态机别名,然后就可以用这个别名代替原来的状态机的名字。

状态机别名说明的 BNF 语法规则如下:

〈符号名〉:MACHINE;

例如:

```
FUNCTION  ss_def (clock, reset, count)
    RETURNS  (MACHINE ss_out);
     ⋮
VARIABLE
     ss   :MACHINE;
BEGIN
     ss = ss_def (sys_clk, reset,   !hold);
IF ss = = s0  THEN
   ⋮
ELSEIF  ss = = s1  THEN
   ⋮
END;
```

一个状态机别名说明语句具有如下特点：

①状态机别名是一个符号名，其后跟一个冒号(：)和一个关键字 MACHINE，如上例 ss 是状态机别名；

②在子设计段内通过设定一个输入或输出端口作为 MACHINE　INPUT 或 MACHINE OUTPUT，在 TDF 文件和其他设计文件之间用户可以输出和输入状态机；

③在用户输出或输入一个状态机时，就必须在代表该文件的函数原型中指明哪些输入端口和输出端口是状态机，上例 ss_out 是状态机名；

④一个分号(；)结束一个状态机别名说明。

应该注意，MACHINE INPUT 和 MACHINE OUTPUT 端口类型不能被用在顶层 TDF 文件中。

4.2.16　逻辑段

逻辑段设定 TDF 文件的逻辑操作并且是一个 TDF 文件的主体。这个段是必需的。下面的语句和结构可以在逻辑段里使用：

①布尔等式 (Boolean Eguations)；

②布尔控制等式(Boolean Control Eguations)；

③情况语句(Case Statement)；

④默认语句 (Defaults Statement)；

⑤If　Then 语句；

⑥If　Generate 语句；

⑦For　Generate 语句；

⑧真值表语句(Truth Table Statement)。

另外，逻辑段也可以包括断言语句(Assert Statement)。

BEGIN 和 END 关键字包围着逻辑段，一个分号(；)跟在 END 关键字之后就结束了这个逻辑段。默认语句必须是在这个逻辑段里的第一条语句。

AHDL 是一种并行语言，编译器在同一时间内同时(而不是顺序地)运算一个 TDF 文件逻辑段内设定的所有行为。把多个值赋给相同的 AHDL NODE 类型结点或变量相当于逻辑“或”。

一、布尔等式

在用户 AHDL TDF 文件逻辑段内的布尔等式用来表示结点连接、输入和输出引脚、原语、强函数、宏函数和状态机的输入信号流和输出信号流。

布尔等式的 BNF 语法规则如下：

〈布尔等式〉::=〈左侧组〉=〈右侧组〉

下面的例子示出了一个复杂的布尔等式：

a[ ] = (c[ ] & - B″001101″) + e[6..1] # (p,g,r,s,t,v)；

布尔等式的左边可以是一个符号、端口或组名。用户可以采用 NOT(即 !)运算符对左边任何项求反。等式右边由布尔表达式组成，按照在基本元素一章里的‘Boolean operator & Comparator Priorities’所说明的那样对它们进行运算。

在布尔等式中用等号(=)说明右边布尔表达式的结果是左边符号结点或组的解。单等号

不同于双等号(==),双等号被用作比较符。

如上例所示,右边的布尔表达式按照布尔等式的优先级规则进行运算,运算过程如下:

①二进制数 B"001101"被求负变成 B"110011",一元减法(-)具有最高优先级;

②B"110011"与组 c[ ]相加(&),构成的表达式具有第二优先级,因为它被括在圆括号内;

③把在第二步里组表达式的结果与组 e[6..1]相加;

④把在第三步里表达式的结果与组(p,q,r,s,m,t,v)相"或"(这个式子具有最低优先级)。

这个最后的结果赋给组 a[ ]。

为了使上面示出的实例的等式合法,等式左边组里的位数必须能被等式右边组里的位数整除,等式左边各位按顺序映射到等式右边各位。

下面的规则适用于布尔等式:

①给一个变量多个赋值是逻辑地"或"(即#),而这个变量的默认值是 VCC 时除外;

②如果布尔等式左边结点的数目等于右边结点的数目,那么存在一一对应关系;

③如果一个等式右边的一个单一结点、地或 VCC 被赋给一个组,那么这个点或常量被复制且直到匹配组的长度,例如可认为"(a,b)=e;"与"a=e,b=e;"是相同的;

④如果等式左边和右边是相同长度的组,那么就把右边的组中每个成员赋给相应位置上的左边组的成员,例如,(a,b)=(c,d)同 a=c、b=d 是一样的;

⑤如果一个等式的左边和右边组的长度不同,那么左边组的位数一定要能被右边组里的位数整除,等式左边的各位按顺序地被映射到等式右边,下边的等式是合法的:

a[4...1]=b[2...1]

在这个等式里,各位按以下形式映射:

a4=b2

a3=b1

a2=b2

a1=b1

⑥一组结点或数值不能赋给一个单独结点;

⑦如果一个等式右边的一个数值赋给一个组,那么这个数被舍位或带符号扩展,直至与组的长度相同 如果有任何一个有意义的位被舍去,编译器就发布一条错误信息,等式右边的每一成员按照对应的位置赋给左边的成员,例如,"(a,b)=1;"同"a=0;b=1"是一样的;

⑧为了保留没有赋值的组成员的位置,在布尔等式里可以使用多个逗号,下面例子示出保留组(a,b,c,d)的二个成员位置的逗号:

(a,,c,)=B"1011";

在这个例子里,a 和 c 被赋值为 1;

⑨每一个布尔等式以分号(;)结束。

应该注意:当用户使用"+"运算符把布尔等式右边的两个组加在一起时,为了用符号扩展组的宽度,用户可以把 0 放在每组的左边。这种方法提供了等式左边组的扩展信息位,可以用它作为进位输出信号。在下面的例子里,组 count[7...0]和 delta[7...0]用 0 提供给 Cout 进位输出信号作为符号扩展:

(cout,answer[7...0])=(0,count[7...0])+(0,delta[7...0])

二、布尔控制等式(Boolean Control Equations)

布尔控制等式是在逻辑段建立状态机时钟(Clock)、复位(Reset)和时钟使能(Clock Enable)信号采用的布尔等式。布尔控制等式的 BNF 语法规则如下：

〈符号名〉·clk = 〈右侧组〉；

[〈符号名〉·reset = 〈右侧组〉；]

[〈符号名〉·ena = 〈右侧组〉；]

下面的例子示出了布尔控制等式：

```
ss.clk = clk 1 ;
ss.reset = a&b ;
ss.ena = clk 1 ena ;
```

布尔控制等式具有如下特点：

①用户可以按照格式〈State machine name〉·〈Port name〉定义每个状态机的时钟、复位、时钟使能输入端口，如上例所示，这些输入端口都是为状态机 ss 定义的；

②在控制等式里用户可以使用在状态机说明语句中作为状态机名字说明过的状态机的名字；

③Clock 信号〈State machine name〉·CLK 一定要始终赋一个值；

④如果状态机的起始状态已经被赋一个非零的值，那么复位(Reset)信号〈State machine name〉·reset 必须被赋值，否则，它是可选的；

⑤给时钟使能信号〈State machine name〉·ena 赋一个值总是可选的；

⑥每个布尔控制等式以分号(;)结束。

三、情况语句(Case Statement)

情况语句列出了几种可能执行的操作，至于执行什么操作，决定于跟在 Case 关键字后面的变量、组或表达式的值。情况语句的 BNF 语法规则如下：

〈Case 语句〉::=

```
    CASE〈右侧组〉IS
        WHEN〈选项〉= >〈语句组〉
        {WHEN〈选项〉= >〈语句组〉}
        [WHEN〈选项〉= >〈语句组〉]
    END CASE ;
```

〈选项〉::=

〈常量组〉{,〈常量组〉}

下面的例子示出了一种情况语句：

```
CASE  f[].q  IS
    WHEN  H"00" = >
        adder[] = 0 ;
        s = a&b ;
    WHEN  H"01" = >
        count[].d = count[].q + 1 ;
    WHEN  H"02",H"03",H"04" = >
```

```
        f[3..0].d = addr[4..1];
    WHEN  OTHERS => 
        f[ ].d = f[ ].q;
END  CASE;
```

情况语句具有如下特点：

①关键字 CASE 和 IS 把一个布尔表达式、组或状态机夹在中间，如上例 f[ ].q；

②情况语句用关键字 END CASE 和一个分号(;)结束；

③一个或多个特定的选择被列在情况语句主体内 WHEN 从句中，每个 WHEN 语句由关键字 WHEN 开始；

④在每一个 WHEN 从句里，在箭头符号(=>)之前都有一个或多个由逗号分开的常量值；(在这个例子里，H'02'、H'03'和 H'04'常量值被列在同一个 WHEN 从句中，而 H'00'和 H'01'常数值被列在分开的 WHEN 语句中)

⑤如果跟在 CASE 关键字后面的布尔表达式运算结果为一个特定值，那么跟在相应的箭头符号后面的所有行为语句都被执行；(如上例所示，如果 f[ ].q 运算结果是 H'01'，那么布尔等式 Count[ ].d = Count[ ].q + 1 被执行)

⑥当没有其他选择符合条件时，将选择由关键字 WHEN OTHERS 定义的默认选择；(如上例所示，如果 f[ ].q 运算结果不等于 H'00'、H'01'、H'02'、H'03'和 H'04'，那么布尔等式就会执行 f[ ].d = f[ ].q)

⑦如果 WHEN OTHERS 从句不被使用，那么默认语句定义默认行为；

⑧如果情况语句被用于定义状态机的转换，那么除非状态机正好包含 2^n 个状态，否则关键字 WHEN OTHERS 不能被用于从一个 n 位状态机的非法状态中重新恢复到合法状态；

⑨每个行为语句用一个分号(;)结束。

四、默认语句(Defaults Statement)

默认语句允许用户用真值表 If Then 和 Case 语句采用的变量指定默认值。由于高电平有效的信号自动地默认为 GND，所以只对低电平有效的信号才需要默认语句。

应该注意的是用户不要把变量的默认值同在子设计段中为端口设定的默认值混淆。

默认语句的 BNF 语句的规则如下：

〈Defaults 语句〉::=

```
    DEFAULTS
        〈默认定义〉;
        {〈默认定义〉;}
    END DEFAULTS;
```

〈默认定义〉::=〈左侧组〉=〈常量组〉

下面的例子示出一个默认语句：

```
BEGIN
    DEFAULTS
        a = VCC;
    END DEFAULTS;
    IF  y&z  THEN
```

```
        a = GND;        % a is active low %
    END IF;
END;
```

默认语句具有如下特点：

①默认语句被关键字 DEFAULTS 和 END DEFAULTS 夹在中间，并用分号(;)结束；

②默认语句体由一个或多个布尔等式组成，它给变量赋一个常量值，上例默认语句给变量 a 赋一个默认值 VCC；

③每一个等式用一个分号(;)结束；

④如果跟在默认语句之后的那个变量在一定条件下没有赋值，那么默认语句被执行，如上例中，当 y 或 z 为逻辑低电平时，变量 a 不能被赋值，因此默认语句中的等式(a = VCC)被执行)。

默认语句应遵照以下规则：

①在逻辑段里只允许有一个默认语句，并且在 BEGIN 关键字之后一定是第一个语句；

②在默认语句中如果一个单独的变量被多次赋值，那么所有赋值只有最后一次有效；

③一个默认语句不能被用于对一个变量设置一个 X(无关位)的默认值；

④在默认语句外边为 NODE 变量类型的结点多次赋值之间的关系是逻辑‘或”，除非这个变量的默认值是 VCC；

⑤被多次赋值的低电平有效的变量应该被赋以默认值 VCC。

在下面的 TDF 文件中，a 具有默认值 GND，bn 具有默认值 VCC。TDF 文件如下：

```
BEGIN
    DEFAULTS
        a = GND;
        bn = VCC;
    END DEFAULTS;
    IF c1 THEN
        a = a1;
        bn = b1n;
    END IF;
    IF c2   THEN
        a = a2;
        bn = b2n;
    END IF;
END
```

这个例子与下面的等式等效：

```
a = c1&a1#c2&a2;
bn = (!c1#b1n)&(!c2#b2n);
```

在下面的例子里 reg[ ].clrn 被赋一个默认值 VCC。

```
SUBDESIGN sbcount
(
```

```
    d[5..1]     :INPUT;
    clk         :INPUT;
    clr         :INPUT;
    sys_reset   :INPUT;
    enable      :INPUT;
    load        :INPUT;
    q[5..1]     :OUTPUT;
)
VARIABLE
    reg[5...1]  :DFF;
BEGIN
    DEFAULTS
        reg[].clrn = vcc;
    END DEFAULTS;
    reg[].clk = clk;
    g[]       = reg[];
    IF sys_reset # clr THEN
        reg[].clrn = GND;
    END IF;
    !reg[].prn = (load&d[]) &    !clr;
    !reg[].clrn = load &    !d[];
    reg[] = reg[] + (0,enable);
END;
```

五、If Then **语句**

在 If Then **语句中可以有一个或多个布尔表达式。如果其中某表达式结果为真,那么该表达式后面的行为语句将被执行。**

If Then **语句的** BNF **语法规则如下:**

〈If Then **语句**〉::=

IF〈**右侧组**〉THEN
〈**语句组**〉
{ELSIF〈**右侧组**〉THEN
〈**语句组**〉}
[ELSE〈**右侧组**〉THEN
〈**语句组**〉]
END IF;

下面的例子示出一个 If Then **语句:**

```
IF a[] = = b[] THEN
    c[8..1] = H"77";
    addr[3..1] = f[3..1].q;
```

```
    f[ ].d = addr[ ] + 1;
ELSIF q3 $ q4 THEN
    f[ ].d = addr[ ];
ELSE
    d = VCC;
END IF;
```

If Then 语句具有如下特点：

①关键字 IF 和 THEN 把要运算的布尔表达式夹在中间并且用一个或多个行为语句跟随其后，每一个行为语句都用一个分号(;)结束；

②关键字 ELSIF 和 THEN 把任何要运算的附加的布尔表达式夹在中间，并且用一个或多个行为语句跟随其后，这些选择语句可以多次重复使用；

③第一个表达式计算为真时，跟在关键字 THEN 之后的行为语句被执行；

④一个或多个行为语句所跟随的关键字 ELSE 类似于情况语句的 WHEN OTHERS 默认选择，如果前面运算的布尔表达式没有一个是真，那么跟随 ELSE 的行为语句被执行(如上例所示，如果任何表达式运算结果都不为真，那么等式 d = VCC 被执行，ELSE 从句也是可选的)；

⑤跟随 IF 和 ELSIF 关键字的表达式(如上例所示，a[ ] = = b[ ]和 g3 $ g4)同时进行运算；

⑥对于 MAX + PLUSⅡ编译器，一个 IF THEN 语句可以产生很复杂的逻辑，如果一个 IF THEN 语句包括复杂表达式，那么每个表达式的‘非’就会更复杂。

在下面的例子里，如果 a 和 b 是复杂的表达式，那么每个表达式的‘非’就更复杂：

If Then 语句	Compiler 的解释
IF a THEN	IF a THEN
c = d;	c = d;
ELSIF b THEN	END IF;
c = e;	IF !a &b THEN
ELSE	c = e;
c = f;	END IF;
END IF;	IF !a & !b THEN
	c = f;
	END IF;

应该注意：If Then 语句只能运算布尔表达式，而 If Generate 语句不同于 If Then 语句，它可以运算算术表达式的 Superset。If Then 语句和 If Generate 语句的基本差别是：在硬件中是运算前面的，而当设计被编译时却运算后面的。

六、If Generate 语句

If Generate 语句列出一系列行为语句，这些语句在确认运算表达式的运算之后起作用。If Generate 语句的 BNF 语法规则与 If Then 语句的相同。

下面的例子示出一个 If Generate 语句：

```
IF DEVICE_FAMILY = = "FLEX8K" GENERATE
```

```
    c[ ] = 8kadder(a[ ],b[ ],cin) ;
  ELSE GENERATE
    c[ ] = otheradder(a[ ],b[ ],cin) ;
  END GENERATE ;
```

If Generate 语句具有如下特点：

①关键字 IF 和 GENERATE 把要计算的运算表达式夹在中间，并用一个或多个行为语句跟在其后，它们中的每一个都用一个分号(;)结束，如果这一表达式为真，那么这些语句就被执行；

②跟在关键字 ELSE GENERATE 后面有一个或多个行为语句，它们中的每一个皆用一个分号结束，如果运算表达式为'假'的则执行这些语句；

③If Generate 语句用关键字 END GENERATE 和一个分号(;)结束；

④If Generate 语句可以用在逻辑段或变量段。

另外，还应注意以下两点：

①同 If Then 语句不同，If Then 语句只能计算布尔表达式，而 If Generate 语句可计算运算表达式的 superset；

②If Generate 语句同 For Generate 语句特别有助于以不同方式处理特殊情况，例如，一个多级乘法器的最小有效位以及被用于测试参数值，如上例所示。

七、For Generate 语句

关于 For Generate 语句，这里先给出一种累接的 For Generate 语句的一个例子：

```
CONSTANT NUM_OF_ADDERS = 8
SUBDESIGN   4 gentst
(
  a[NUM_OF_ADDERS..1], b[NUM_OF_ADDERS..1], cin  : INPUT ;
  c[NUM_OF_ADDERS..1], cout                      : OUTPUT ;
)
VARIABLE
  carryout[(NUM_OF_ADDERS + 1)..1]      : NODE ;
BEGIN
  carryout[1] = Cin ;
  FOR i  IN 1 TO NUM_OF_ADDERS GENERATE
    c[i] = a[i] $ b[i] $ carryout[i]  ;     %Full Adder%
    carryout[i + 1] = a[i] &b[i] # carryout[i] &(a[i] $ b[i]) ;
  END GENERATE ;
  cout = carryout[NUM_OF_ADDERS + 1] ;
END ;
```

由上例可以看出 For Generate 语句具有如下特点：

①关键字 FOR 和 GENERATE 把下面各项夹在中间；

②用一个或多个逻辑语句跟在关键字 GENERATE 之后，每个逻辑语句都以分号(;)结束；

③用关键字 END GENERATE 和一个分号(;)结束 For Generate 语句。

现对关键字 FOR 和 GENERATE 说明如下：

①一个由符号名组成的临时变量名只用在 For Generate 语句的范围内，也就是说，这种变量在编译器处理这种语句之后中止存在，如上例中变量名 i 不能是在设计里其他地方采用的常量、参数或结点名；

②由两个运算表达式定界的范围跟在字 IN 之后，这两个运算表达式由 TO 关键字分开，如上例所示，运算表达式是 1 和 NUM_OF_ADDERS，这个范围终点可由只包含常量和参数的表达式组成，而不需要变量。

八、内部逻辑函数直接引用(In_Line_logic Function Reference)

逻辑函数内部直接引用是一种实现逻辑功能的布尔等式，是一种实现逻辑功能的软件方法。它只能用于逻辑段的一行并且不需要作变量说明。

当用户希望实现 megafunction 或 macrofunction 的一个实例时，必须保证在自己的设计文件内定义它的逻辑，而且采用一个函数的函数原形语句并使用一个内部直接引用或一个实例说明实现函数的实例。

为了实现一种原语的实例，用户也采用一种内部直接引用或者一种实例说明。然而，在用 megufunction 和 macrofunction 的对比中，原语逻辑是被预先定义的，因此用户不需要在一个独立的设计文件里定义原语逻辑。在多数情况下，不需要函数原型语句。

在布尔等式中，逻辑函数内部直接引用的 BNF 语法规则如下：

〈函数〉(〈右侧组表〉)

在布尔等式中，原语内部直接引用的 BNF 语法规则如下：

〈原语〉(〈右侧组表〉)

下面的例子示出关于 Compare 和 Ipm_add_sub 函数的函数原型。Compare 函数具有输入端口 a[3..0]、b[3..0]和输出端口 less、equal 和 greater；Ipm_add_sub 函数具有输入端口 dataa[LPM_WIDTH-1..0]、datab[LPM_WIDTH-1..0]、cin 和 add_sub 以及输出端口 result[LPM_WIDTH-1..0]、cout 和 overflow。

例如：

```
FUNCTION Compare (a[3..0],b[3..0])
    RETURNS (less,equal,greater);
FUNCTION lpm_add_sub (cin,dataa[LPM_WIDTH_1..0],
    datab[LPM_WIDTH_1..0],add_sub)
    WITH(LPM_WIDTH,LPM_REPRESENTATION)
    RETURNS(result[LPM_WIDTH_1..0],Cout,overflow);
```

关于 Compare 和 lpm_add_sub 函数的逻辑函数内部直接引用出现在下面等式的右边：

```
(clockwise,,counterclockwise) = compare(position[ ],target[ ]);
sum[ ] = lpm_add_sub(.datab[ ] = b[ ],.dataa[ ] = a[ ])
    WITH(LPM_WIDTH=8)
    RETURNS (.result[ ]);
```

一个逻辑函数的内部直接引用具有如下特点：

①在一个等号(=)右边的函数名字之后用一个括在圆括号内的信号表跟随，这个信号表

包括用逗号分开的符号名、十进制数或组,这些项对应于函数的输入端口。

②在这个信号表内,可以通过位置端口联合或者命名的端口联合给出各端口名。

③在一个参数化的函数里,关键字 WITH 和参数名表跟着输入端口表。这个表被括在圆括号内,参数名用逗号分开,只有经过实例应用的参数被说明,可选的参数值与参数名用一个等号分隔。如上面的 lpm_add_sub 例子所示,LPM_WIDTH 参数被配置值为 8。如果在内部直接引用中没有配置参数值,那么编译器按照规定的检查顺序对它们在参数值上进行检查。

④在内部直接引用的左边,函数的各种输出被连接到各种变量上。如上面 Compare 例子所示,函数的 Less 和 greater 输出各自地通过位置的端口联合,被连接到变量 clockwise 和 counterclockwise 上。类似地,在 Ipm_add_sub 例子里,通过位置的端口联合连接 sum[ ]各种输出。

⑤在逻辑段内任何位置确定变量的值赋给联合的各个输入端和输出端,如上面 compare 例子所示,position[ ]和 target[ ]的值赋给 compare 的各个输入端,输出端口 less 和 greater 的值分别赋给 clockwise 和 counterclockwise。在逻辑段内各个变量可以被用于其他运算。

如上面的 Compare 例子所示,各自通过位置端口联合把 compare 的 a[3..0]和 b[3..0]输入端口连接到以 position[ ]和 target[ ]命名的各种变量上。当用户使用位置的端口联合时,可以采用多个逗号作为不能连接到变量上的输出端口的保留位置。在 Compare 内,equal 输出端口不被连到任何变量上,因此一个存在的逗号必须在等号左边的组内找到它的位置。

如上面 Ipm_add_sub 例子所示,各自通过命名的端口联合把 Ipm_add_sub 的 dataa[ ]和 datab[ ]输入端口连接到变量 b[ ]和 a[ ] 上,用一等号(=)把端口名连接到变量上。

在这里应注意两点:第一.在应用命名端口联合的内部直接引用的左右两边,各种端口名必须具有〈port name〉格式;第二.仅在一个内部直接引用的右边提供命名的端口联合,一个内部直接引用的左边总是通过位置的端口联合被连接到各个变量上。

九、真值表语句(Truth Table Statement)

真值表语句被用于指定组合逻辑或状态机的行为。在一个 AHDL 真值表内,每个表项都包含输入值的一种组合形式,它产生指定的输出值。这些输出值也可以作为反馈来确定状态机转换和状态机的输出。

真值表语句的 BNF 语法规则如下:

〈真值表〉::=

TABLE〈输入项〉=>〈输出项〉;

〈输入值〉=>〈输出值〉;

{〈输入值〉=>〈输出值〉;}

END TABLE;

〈输入项〉::=〈右侧组〉{,〈右侧组〉}

〈输出项〉::=〈左侧组〉{,〈左侧组〉}

〈输入值〉::=〈常量组〉{,〈常量组〉}

〈输出值〉::=〈常量组〉{,〈常量组〉}

下面的例子示出一个真值表语句:

```
TABLE
    a0        f[4..1].q   =>   f[4..1].d,        control;
```

```
    0,      B"0000"   =>   B"0001",      1;
    0,      B"0100"   =>   B"0010",      0;
    1,      B"0xxx"   =>   B"0100",      0;
    x,      B"1111"   =>   B"0101",      1;
  END TABLE;
```

真值表语句具有的特点如下：

①真值表表头由关键字 TABLE、一组由逗号分开的真值表输入项、一个箭头符号(=>)以及一组由逗号分开的输出项组成，并由一个分号(;)结束；

②真值表各种输入是布尔表达式，真值表输出是变量(如上例所示，输入信号是 a0 和 f[4..1].q，输出信号是 f[4..1].d 和 control)；

③真值表的主体由一项或多项列表值组成，每项占据一行或多行并用一个分号结束；

④每一个真值表项都由一组由逗号分开的输入项和一组由逗号分开的输出项组成，输入项和输出项之间用=>分开；

⑤每个信号在每一个表项里都有一一对应的值，上例中当 a0 具有值 0 和 f[4..1].q 具有值 B"0000"时，f[4..1]将具有值 B"0001"和 control 将具有值 1；

⑥输入和输出值可以是数值、预定义的常量 VCC 或 GND、符号常量(即用作常量的符号名)、数值或常量组，输入值也可以是 x(无关项)；

⑦输入和输出值对应于表头的输入和输出；

⑧真值表以关键字 END TABLE 后跟一个分号(;)结束。

在使用真值表时应遵守下述规则：

①在表头里的名字可以是单个结点，也可以是组；

②所有输入值的组合形式并不是都有必要列出，用户可以在与输出结果无关的输入位上放上 x(无关项)；

③在真值表各行里用逗号分开的项数必须等于真值表表头里用逗号分开的项数；

④在实际的输入值与真值表中所列的输入值不匹配的情况，默认语句将会设定输出值。

下面的例子说明，如果 a0 为高并且 f4 为低，那么其他输入位则无关紧要，因此只要指定输入值组合中的共同部分(此例为 0)，而在其他位上只写字符 x 即可。例子如下：

```
  TABLE
    a0,  f[4..1].q    =>   f[4..1].d,    control;
    0,   B"0000"         B"0001",       1 ;
    0,   B"0100"         B"0010",       0 ;
    1,   B"0xxx"         B"0100",       0 ;
    x,   B"1111"         B"0101",       1;
  END  TABLE;
```

应该注意的是，当用户用 x(无关项)字符指定一个位模式时，必须保证在真值表内这种模式不能假定另一种位模式的值。假定在 AHDL 真值表内只有一个条件在一个时间点上是真，因此重叠的位模式可能引起非预定模式。

4.3 AHDL 的基本元素

在这一节介绍 AHDL 文本设计文件(.tdf)的基本格式和元素。在设计结构一节中所用的基本元素在行为语句中描述。

4.3.1 保留关键字和标识符

在 AHDL 语句的开始、结尾和中间过程都要用保留关键字,并且为了预定义常数值,GND 和 VCC 也要用保留关键字。

保留关键字同保留标识符有区别。当保留关键字被括在单引号(())内时可以被当作符号名使用,而保留标识符则不能。但是保留关键字和保留标识符都能在注释中任意使用。

一、保留关键字(Reserved Keywords)

下面列出全部 AHDL 保留关键字:

AND	FUNCTION	OUTPUT
ASSERT	GENERATE	PARAMETERS
BEGIN	GND	REPORT
BIDIR	HELP_ID	RETURNS
BITS	IF	SEGMENTS
BURIER	INCLUDE	SEVERITY
CASE	INPUT	STATES
CLIQUE	IS	SUBDESIGN
CONNECTED_PINS	LOG2	TABLE
CONSTANT	MACHINE	THEN
DEFAULTS	MOD	TITLE
DEFINE	NAND	TO
DESIGN	NODE	TRI_STATE_NODE
DEVICE	NOR	VARIABLE
DIV	NOT	VCC
ELSE	OF	WHEN
ELSIF	OPTIONS	WITH
END	OR	XNOR
FOR	OTHERS	XOR

二、保留标识符(Reserved Identifiers)

下面列出全部 AHDL 保留标识符:

CARRY	JKFFE	SRFFE
CASCADE	JKFF	SRFF
CELL	LATCH	TFFE

DFFE	LCELL	TFF
DFF	MCELL	TRI
EXP	MEMORY	WIRE
FLOOR	OPNDRN	X
GLOBAL	SOFT	

由 AHDL 保留标识符可以看出,这是为一些专门用途所保留的名称。它们是所有缓冲器、触发器和锁存器原语的名称,以及预定义的逻辑级(Logic Level)X。

4.3.2 符号

表 4-1 列出在 AHDL 中预先定义的符号。这个表包括在布尔表达式中,并作为运算和比较所采用的符号,以及在算术表达式中作为运算所采用的符号。

表 4-1 AHDL 符号(第一部分)

符 号	功 能
_ (下画线) — (破折号) / (正向斜线)	用户自定义的标识符在符号名中作为合法字符应用
— —(两个破折号)	用在 AHDL 格式注释的开始,注释直到这一行的结尾
% (百分号)	包围 AHDL 格式的注释
( ) (左和右圆括号)	①用于括引和定义有顺序(Sequential)组的名称 ②在子设计段和函数原型语句中括引脚的名称 ③在真值表语句中选择性地括引真值表的输入和输出 ④括引状态机说明的状态位和状态 ⑤括引在布尔表达式和算术表达中最高优先级的运算 ⑥括引在参数语句中的参数定义和在函数原形语句中的参数名称、实例说明以及内部直接引用 ⑦选择性地括引判断语句中的条件 ⑧括引定义语句中评估函数的自变量
[ ] (左和右方括号)	括引单一或双域数组的数域

表 4-1 AHDL 符号(第二部分)

符 号	功 能
' ' (单引号)	括引带引号的符号名
" " (双引号)	括引标题语句、参数语句和判断语句中的字符串和在函数语句中括引文件名以及在非十进制数中括引数字
. (句号)	用在逻辑函数变量的符号名和端口名之间以及文件名和延伸部分之间
. . (省略号)	用在一个区域的 MSB 和 LSB 之间
; (分号)	用在 AHDL 语句和段的末尾
, (逗号)	分隔有顺序组和表中的各成员
: (冒号)	在说明语句中分隔符号名和类型名

续表

符　号	功　能
= （等号）	①在子设计段中为输入端口设置默认值 GND 和 VCC ②在选择语句中为选择指定的设置 ③在参数语句或内部直接引用语句中为参数设置的默认值 ④为状态机状态赋值 ⑤在布尔等式中赋值 ⑥把一个信号同内部直接引用中的一个端口联系起来采用给端口联合起名

表 4－1　AHDL 符号(第三部分)

符　号	功　能
=> （箭头）	①在真值表语句中分隔输入和输出 ②在 case 语句中分隔 WHEN 从句和布尔表达式
+ （加号）	加法运算
－ （减号）	减法运算
== （两个等号）	数字或字符串的相等运算
! （感叹号）	“非”运算
!= （感叹等于）	“非”等于运算
> （大于）	大于(比较器)
>= （大于等于）	大于或等于(比较器)
< （小于）	小于(比较器)
<= （小于等于）	小于或等于(比较器)
& （与）	AND 运算
!& （与非）	NAND 运算
$ （美元符）	XOR 运算
!$ （感叹美元符）	XNOR 运算
# （磅符）	OR 运算
!# （感叹磅符）	NOR 运算
? （问号）	三元运算

4.3.3　带引号和不带引号的名称

在 AHDL 中有三种类型的符号名称。

1. 用户定义的标识符

在 AHDL 中符号名称是用户定义的标识符,用它们在 TDF(文本设计文件)中对下面几部分进行命名：

①内部的和外部的节点和组；

②常量；

③状态机变量、状态位和状态名；

④实例；

⑤参量；

⑥存储程序段；

⑦计算函数；

⑧命名的运算。

2. 子设计名

子设计名是用户为下层设计文件定义的名称。子设计名必须与相应的 TDF 文件名相同。

3. 端口名称

端口名称是为逻辑函数的输入和输出指定的名称。

用户应该注意,由编译器生成的引脚名称包含一种代表字符(~)的字符。这些引脚可以出现在设计中的 Fit 文件里。如果用户随后标注这个 Fit 文件指定的任务,那么这些名称将出现分配与配置文件中(.acf)。代字号字符只能用于编译器生成的名称中。用户不能在自己定义的引脚、节点和组(总线)名称中使用它。

子设计、符号和端口名称有两种标准:带引号和不带引号。带引号的名称被括引在单引号内(′);不带引号名称不能括引在单引号内。

当用户为一个包含带引号端口名的 TDF 文件创建一个默认符号时,在引脚接头名称里显示的符号不包括带引号的名称。表 4-2 概括了子设计名称、符号名称和端口名称。

表 4-2 带引号和不带引号名称

合法的名称 字符注释*	不带引号的 子设计名称	带引号的子 设计名称	不带引号的 符号名称	带引号的 符号名称	不带引号的 端口名称	带引号的 端口名称
A—Z	√	√	√	√	√	√
a—z	√	√	√	√	√	√
0—9	√	√	√	√	√	√
下画线 (Underscore) (_)	√	√	√	√	√	√
斜线 (Slash)(/)	No	No	√	√	√	√
破折号(Dash)(—)	No	√	No	√	No	√
数字 (0-9) (Digits only)	√	√	No	√	√	√
关键字 (Keyword)	No	√	No	√	No	√
标识符 (Identifier)	No	√	No	√	No	√
最多字符数 (Max characters)	32	32	32	32	32	32

* 在单值域和双值域组名称的范围的定义符也能包括在算术表达式中所描述的运算。

合法的不带引号和带引号的符号名称举例:

a /a1 ′-bar′ ′table′ ′1221′

不合法的不带引号和带引号的符号名称举例:

foo node 55 ′bowling4 $ ′

′a_name_with_more_then_32_characters′ ′has a space′

4.3.4 组

在布尔表达式和布尔等式中,相同类型的符号名称和端口名称可以当做组来说明和应用。一个组最多可包括 256 个成员(或位)。它可以当做许多结点的集合,并且被当做一个整体来

操作。在 TDF 文件的逻辑段或变量段中组可由许多结点组成。在布尔表达式和布尔等式中，一个结点和常量 GND 和 VCC 可以被复制成一个组。

一、组的标记（Group Notations）

组能用三种记法说明。

1. 单值域组名称

一个单值域组名称由一个符号名称或端口名称后跟一个括在方括号内的单值数域组成，例如 a[4..1]。符号名和端口名加上在区域内最长的数字的总长度不能超过 32 位字符。一个组被定义之后，利用[ ]指定输入范围是一种速记方法。例如，a[4..1] 也可以用 a[ ]表示。也能把一个单个的数放在方括号内，如 a[5]。但是，这种标记符号是一种单个符号名称而不是一个组的名称，并且与 a5 的意义相同。

2. 双值域组名称

双值域组名称由一个符号名或端口名后跟括在方括号中的两个值域组成，即 d[6..0][2..0]。符号名或端口名加上在每一个值域中最长的数值不能超过 32 个字符。

对于总线规定的组和对于具有两个域拓扑的设计来说，双值域组标记是有用的。在一个组已经被定义之后，[ ][ ]指定双值域的速记方法。例如，b[6..0][3..2]也能用 b[ ][ ]标记。

在组内一个单独的结点能当作 name[Y][Z]或 name Y_Z 考虑，那里的 Y 和 Z 是组域内的数值。

3. 一个有序组名称

一个有顺序组的名称是由一组符号名、端口名或者数字组成，它们之间用逗号分隔，并且被括在圆括号中，例如，(a,b,c)。单值域和双值域组的名称也可以被列入圆括号中。例如，(a,b,c[5..1])就是一个合法的组名。这种记法对于指定端口名是非常有用的。例如，一个 DFF 类型的变量 reg 的输入端口能被写作 reg.(d,clk,clrn,prn)。

下面两种设置的例子示出两组不同记法的规定：

```
b[5..0]
(b5,b4,b3,b2,b1,b0)
b[ ]

b[Log2(256)..1+2-1]
b[2^8..3 mod 1]
b[2*8..8 div 2]
```

用户应注意，由编译器生成的引脚名包括一种代字号(~)字符。这些引脚可以出现在设计中的 Fit 文件里(.fit)。如果用户随后标注 Fit 文件指定的任务，那么这些名称将出现在设计的分配与配置文件中(.acf)。代字号字符只能被用于编译器生成的名称中。用户不能在自己定义的引脚、结点和组(总线)名称中使用它。

二、组域和组的子域（Group ranges & subranges）

单值域或双值域的域组名称可由数字或算术表达式组成，在它们中间用二个断续点(..)隔开，并且括在方括号内。例如：

a[4..1]	是一个具有成员 a4、a3、a2 和 a1 的组。
d[B"10"..B"00"]	是一个具有成员 d2、d1 和 d0 的组。

b[2 * 2..2 - 1]　　是一个具有成员 b4、b3、b2 和 b1 的组。这个组域界限由算术表达式确定。

q[MAX..0]　　如果常量 MAX 已经在常数语句中被预先定义，那么它就是合法组名。

c[MIN(a,b)..0]　　如果评估函数 MIN 在定义语句中预先被定义，那么它就是合法组名。

不管域的界限是一个数值还是一个算术表达式，经过编译器后生成的界限均是一个十进制值(整数)。

组的子域是所定义的组中结点的子集，并且能被指定为几种方法的一种。逗点只在组内一个布尔表达式或内部直接引用的左边当做保留位置使用。例如，如果用户定义了组 C[5..1]，那么就能采用下述的这些组的子域：

C[3..1]

C[4..2]

C4

C[5]

(C2,,C4)

在子域(C2,,C4)里，一个逗点被用于保持一个未分配组成员的位置。

域一般按降序排列。为了按升序或是以升序和降序混合排列，就必须用 Options 语句指定BIT0的位置，否则就会在编译时产生警告信息。在二元域组名中BIT0的选择影响两个域。

4.3.5　AHDL 的数字

在 AHDL 中能够按任意组合方式使用十进制、二进制、八进制和十六进制数字。每一种数制(计数系统)的句法如下：

数　制	值
十进制	＜数字 0 到 9 的系列＞
二进制	B＜0、1、X 的系列＞(这里的 X 为“无关项”)
八进制	O“＜数字 0 到 7 的系列＞”或 Q“＜数字 0 到 7 的系列＞”
十六进制	X“＜0 到 9，A 到 F 的系列＞”或 H“＜0 到 9，A 到 F 的系列＞

下面的例子示出有效的 AHDL 数字：

B“0110X1X10”

Q“4671223”

H“123ABCF”

下面规则适用于 AHDL 数字：

①MAX + PLUS Ⅱ 编译器总是把布尔表达式中数字编译为二进制数组，把组域中的数字编译为十进制数值；

②不能把数值赋给布尔表达式中的一个单一结点，必须用 VCC 和 GND 替代这个值。

4.3.6　算术表达式

算术表达式用于定义语句中的计算函数、常量语句中的常量，并且还能作为组域的域界。

例如,用一个算术表达式定义一个域:

```
SUBDESIGN  foo
(
        a[4..2+1-3+8]:        INPUT;
)
```

再如,用一个算术表达式定义一个常量和一个计算函数:

```
CONSTANT  foo = 1+2 DIV 3+LOG2(256);
DEFINE  MIN(a,b) = ((a<b)?a:b);
```

在这些表达式中使用的算术运算符和比较符执行了表达式中关于所用数值的基本算术运算和比较运算。表 4-3 示出 AHDL 算术表达式中使用的各种算术运算符和比较运算符。

表 4-3　在算术表达式中使用的各种算术运算符和比较运算符

运算符/比较符	例　子	说　明	优先权
+(一元)	+1	正	1
-(一元)	-1	负	1
!	!a	非	1
^	a^2	乘方	1
MOD	4 模 2	模数	2
DIV	4DIV2	除	2
*	a*2	乘	2
LOG2	LOG2(4-3)	以 2 为底的对数	2
+	1+1	加	3
-	1-1	减	3
==(数值)	5==5	数值相等	4
==(串)	"a"=="b"	串相等	4
!=	5!=4	不等	4
>	5>4	大于	4
>=	5>=4	大于或等于	4
<	a<b+2	小于	4
<=	a<=b+2	小于或等于	4
& AND	a & b a AND b	"与"	5
!& NAND	1 !& 0 1 NAND 0	"与非"	5
$ XOR	1 $ 1 1 XOR 1	"异或"	6
!$ XNOR	1 !$ 1 1 XNOR 1	"异或非"	6
# OR	a # b a OR b	"或"	7
!# NOR	a !# b a NOR b	"或非"	7
?	(5<4)?3:4		8

一元加号(+)和减号(-)是前缀运算符。+运算符对操作数没有任何影响,但是为了记录的目的可以使用它。例如,明确表示一个正数。

预先规定的计算函数 CELL 和 FLOOR 也能用于算术表达式。一个实数的单元是最小的整数,是这个实数的最小值;最低限(FLOOR)是最大的整数,是这个实数的最大值。虽然这些运算运用于全部算术表达式,但是,它们仅对 LOG2 和 DIV 有意义,并在这些运算中能够得到一个实数结果。这种单元或最低限用括在圆括号中的表达式求得,并在 CELL 或 FLOOR 中得到它。

下例示出一个实数单元和最低限:

CELL(LOG2(255)) = 8

FLOOR(LOG2(255)) = 7

下述规则适用于全部算术表达式:

①算术表达式必须是非负数的结果;

②当 LOG2 的结果不是一个整数时,这个结果自动提升到下一个整数,例如,LOG2(257) = 9。

4.3.7 布尔表达式

由逻辑运算符、算术运算符和比较符分隔的各种运算组成布尔表达式,并且在圆括号内任意组合。表达式用在布尔方程中以及其他语句中,例如 Case 和 If Then 语句。

一个布尔表达式可以是下列之一:

①一个操作数,例如:a,b[5..1],7,VCC;

②一个内部逻辑函数引用,例如:out[15..0] = 16 dmux (q[3..0]);

③在一个布尔表达式前面加一个前缀运算符(!或 -),例如:!C;

④两个布尔表达式间夹一个二元(非前缀)运算符,例如:d1 $ d3;

⑤一个布尔表达式括在括号中,例如:(!foo & bar)。

每一个布尔表达式的结果与它的操作数的宽度是相同的。

4.3.8 逻辑运算符

表 4-4 列出布尔表达式中能够被采用的逻辑运算符。

表 4-4 在布尔表达式中应用的逻辑运算符

运算符	例 子	说 明
! NOT	!tob NOT tob	对 1 求补(前缀非)
& AND	bread & butter bread AND butter	"与"
!& NAND	a[3..1] !& b[5..3] a[3..1] NAND b[5..3]	"与非"
# OR	trick # treat trick OR treat	"或"
!# NOR	c[8..5] !# d[7..4] c[8..5] NOR d[7..4]	"或非"

续表

运算符	例　子	说　明
$ XOR	foo $ bar foo XOR bar	“异或”
! $ XNOR	x2 ! $ x4 x2 XNOR x4	“异或非”

每一个运算符都代表一个二输入逻辑门,但 NOT(!)运算符除外。NOT(!)是一个单一结点前缀反相器,表示一个逻辑运算符既可以用它的名称也可以用它的符号。

根据操作数的不同(一个单独的结点、组、数值),一个布尔表达式对其中运算的解释就不同。另外,对含有 NOT 运算符的表达式的解释与对带有其他逻辑运算符的表达式解释也不同。

4.3.9　应用 NOT 的布尔表达式

NOT 操作符是一个前缀反相器,它的作用依操作数的不同而不同。

与 NOT 操作符一起使用的有三类操作数:

①如果操作数是一个单纯的结点、GND 或 VCC,那么就执行一个单纯的反相运算,例如,!a 表示信号 a 通过一个反相器;

②如果操作数是一组结点,那么该组中的每一个成员都要通过一个反相器,例如,一个组 !a[4..1]被解释为(!a4,!a3,!a2,!a1);

③如果操作数是一个数,就用一个像组一样的许多位的二进制数替代,并且每一位都被求反,例如,!9 在一个四个数字组中被求反如 !B“1001”,即 B“0110”。

4.3.10　应用 AND、NAND、OR、NOR、XOR 和 XNOR 的布尔表达式

二元(非前缀)运算符具有五种操作数的组合方式,每一种组合方式含义都不同。

①如果两个操作数是单独的结点或常量 GND 和 VCC,那么运算符将对这两个操作数进行逻辑运算,例如,(a & b)。

②如果两个操作数都是结点组,那么运算符将对两个结点组中相对应的两个结点进行运算,也就是对两组结点做按位运算。这两组必须具有相同的长度。例如,(a,b,c) # (d,e,f)被解释为(a#d,b#e,c#f)。

③如果一个操作数是一个单独的结点、常量 GND 或 VCC,而另一个操作数是一个结点组,那么这个单独的结点或常量被复制成另一个操作数相同长度的一个结点组。于是这个表达式将用一个组的运算代替。例如,a & b[4..1]解释为(a&b4,a&b3,a&b2,a&b1)。

④如果两个操作数都是数值,那么较短的数值将被带符号扩展至同另一个数值相匹配的长度,于是这个表达式将被一个组的运算代替。例如,在表达式(3#8)中,3 和 8 被转变为二进制数 B“0011”和 B“1000”,于是这个表达式变成 B“1011”。

⑤如果一个操作数是数值,而另一个是结点或结点组,那么该数值将被截至或带符号扩展至另一结点组的长度。在这个过程中,如果某有效值被截掉,那么就会产生一条错误信息。上述处理完成后表达式就用一个组运算代替。例如,在表达式(a,b,c) & 1 中,1 变成 B“001”并且表达式变成(a,b,c) & (0,0,1),于是这个表达式被解释为(a&0,b&0,c&1)。

此外,还应该注意 在表达式中用 VCC 作操作数与用 1 作操作数的含义是不同的。在下面第一个等式中,数值 1 要被带符号扩展至另一个结点组的长度,而在第二个方程中,结点 VCC 要被复制为另一结点组的长度。然后每个表达式再用一个组运算代替。两个等式如下:

(a,b,c)&1 = (0,0,c)

(a,b,c)&VCC = (a,b,c)

4.3.11 在布尔表达式中的算术运算符

在布尔表达式中算术运算符被用来对组和数值进行加减运算。表 4-5 示出可用的运算符。

表 4-5 在布尔表达式中使用的算术运算符

运 算 符	例 子	说 明
+(一元)	+ 1	正号
-(一元)	-a[4..1]	负号
+	Count[7..0] + delta[7..0]	加
-	rightmost_x[] - leftmost_x[]	减

一元加号(+)和减号(-)是前缀运算符。 + 运算符不影响它的操作数,但是为了记录的目的可以使用它(即,表示一个正数)。对于 - 运算符来说,如果它的操作数不是数值,就将该操作数表示为一个二进制数。然后再对其做 2 的求补运算。

对于其他的算术运算符还有如下的规定:

①运算符在两个操作数间执行时,两个操作数必须是结点组或数值;

②如果两个操作数是结点组,那么这两个组的长度必须相同;

③如果两个操作数是数值,那么较短的数值将要带符号扩展至另一个操作数的长度;

④如果一个操作数是一个数值而另一个是结点组,那么这个数值将被截至或带符号扩展至结点组的长度,如果在这个过程中某有效位被截去,MAX + PLUS Ⅱ 编译器发出一个错误信息。

除上述规定外,还应注意,当在布尔等式的右侧有两个组相加时,可以在每组的左边加一个零(0)对各组的长度进行扩展。用这种办法可以给等式在左侧的组提供一个附加位,用来代表进位信息。在下面的例子中,组 Count[7..0]和 delta[7..0] 用零(0)作扩展目的是给进位信号 Count 提供信息:

(Count,answer[7..0]) = (0,count[7..0] + (0,delta[7..0])

4.3.12 比较符

有两类比较符,即逻辑比较符和算术比较符。它们被用于对单独结点或结点组进行比较。表 4-6 给出用于布尔表达式中的比较符。

逻辑比较符能够比较单独结点、结点组以及不带无关项(X)的数值。如果对结点组或数值进行比较,那么结点组必须具有相同的长度。MAX + PLUS Ⅱ 编译器对组进行按位比较。当比较结果为真,则返回 VCC ;当比较结果为假,则返回 GND。

表 4－6　用于布尔表达式中的比较符

比较符	例　子	说　明
= =(Logical) ! =(Logical)	addr[19..4] = = B"B800" b1　! = b3	等于 不等于
<(arithmetic)	fame[ ] < power[ ]	小于
<=(arithmetic)	money[ ] <= power[ ]	小于或等于
>(arithmetic)	love[ ] > money[ ]	大于
>=(arithmetic)	delta[ ] >= 0	大于或等于

算术比较符只能对结点组和数值进行比较，而且结点组的长度必须相同。编译器对结点组执行一个元符号值比较，也就是说，每一个结点组都作为一个正的二进制数与另一个结点组进行比较。

4.3.13　布尔运算符和比较符的优先级

用逻辑运算符、算术运算符和比较运算符隔开的操作数按照表 4－7 列出的优先级（优先级 1 是最高优先级）运算。相同优先级将从左至右运算。圆括号（）可以改变运算顺序。

表 4－7　布尔运算符和比较符优先级

优先级	运算符/比较符	
1	－	（负号）
1	!	（"非"）
2	+	（加号）
2	－	（减号）
3	= =	（等于）
3	! =	（不等于）
3	<	（小于）
3	<=	（小于或等于）
3	>	（大于）
3	>=	（大于或等于）
4	&	（"与"）
4	! &	（"与非"）
5	$	（"异或"）
5	! $	（"异或非"）
6	#	（"或"）
6	! #	（"或非"）

在这里应该注意的是 在布尔表达式中提供的算术运算符是算术表达式中提供的算术运算符的一个子集。

4.3.14　原语

MAX＋PLUSⅡ为 ALTERA 器件提供了一套关于设计电路的原语函数。

图形设计文件（.gdf）中的原语在文本设计文件（.TDF）中被 AHDL 的语句、运算符和关键

字所代替：

①在 AHDL 中所用的 INPUT、OUTPUT 和 BIDIR 端口代替了 GDF 文件使用的 INPUT、OUTPUT 和 BIRID 原语；

②在 GDF 文件中的 AND、NAND、BAND、BNAND、OR、NOR、BOR、BNOR、XOR、XNOR 和 NOT 逻辑原语在 AHDL 语言中被逻辑操作符代替；

③在 GDF 文件中的 VCC 和 GND 原语在 AHDL 中被 VCC 和 GND 关键字代替；

④GDF 标题块原语被 AHDL 的 Title(标题)语句所代替；

⑤GDF 文件中的 PARAM 和 CONSTANT 原语被 AHDL 中的参数和常数语句代替。

本节提供了关于各种原语、在原语和端口之间的互连信息以及每一种原语和它们的 AHDL 函数原型的说明。在 TDF 文件中不需要用函数原型，但必要时可以用函数原型语句重新定义原语输入端口的顺序。

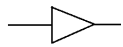
AHDL 提供的缓冲器原语如下：

CARRY = 进位缓冲器
CASCADE = 级联缓冲器
EXP = 扩展缓冲器
GLOBAL = 全局缓冲器(SCLK 同步时钟缓冲器也可用于向后兼容)
LCELL = 逻辑单元缓冲器(MCELL 宏单元缓冲器也可用于向后兼容)
OPNDRN = 漏极开路缓冲器
SOFF = 软缓冲器
TRI = 三态缓冲器

除 TRI 和 OPNDRN 以外，所有的缓冲器原语都能对逻辑综合过程进行控制。在多数情况下，不需要使用这些缓冲器原语。但是，如果编译器提示用户所作的设计太复杂而无法处理，那么用户就可以在设计部分插入上述某些原语。这就产生逻辑表达式，于是引导逻辑综合器产生期望的结果。

一、CARRY 原语

图形符号：CARRY　　函数原型：FUNCTION　　CARRY(in)



RETURNS　　(out)；

CARRY 原语可为一个函数设置进位输出逻辑，还可作为另一函数的进位输入。这个进位函数在加法器和计数器中可用于实现快速进位链逻辑。

应该注意的是，CARRY 原语只支持为 FLEX 8000 和 FLEX 10K 器件系列作的设计，对其它器件是无效的。

在应用 CARRY 原语时，必须注意以下几点：

①一个 CARRY 原语可以输入给一个或两个逻辑体；

②由 CARRY 原语作为一个输入端的逻辑体最多还可以有两个输入端，第三个输入端只能是来自 CARRY 的输入；

③输出给 CARRY 原语的逻辑体最多也只能有两个输入端，第三个输入端只能是来自 CARRY 的输入；

④CARRY 原语的输出不能被送至 OUTPUT 或 OUTPUTC 引脚；

⑤CARRY 原语不能以 INPUT 或 INPUTC 引脚作为输入，也不能以寄存器作为输入；

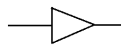
⑥两个 CARRY 原语不能输出至同一个门。

当一个 CARRY 原语输入给两个逻辑体时，其中一个且只能有一个逻辑体要由一个 CARRY 原语作输出缓冲，在这种情况下，两个逻辑体会在同一逻辑单元中实现。对于加法器和计数器的第一级，用户必须遵守这个规则以便把相加和与进位函数束缚在一起。

如果用户错误地使用了 CARRY 原语，它将被忽略，编译器还要发出一条警告信息。用户也可以让编译器在逻辑综合时自动插入或删除 CARRY 原语，具体方法是通过对 Carry Chain 逻辑选项进行不同的设置来控制逻辑综合，或在含 Carry Chain 选项的逻辑综合风格选项中对 Carry Chain 做出选择。

二、CASCADE 原语

图形符号：CASCADE　　函数原型：FUNCTION　　CASCADE (in)



RETURNS　　(out)；

CASCADE 缓冲器表示来自一个“与”门或者“或”门级联输出的函数，也可作为对另一个“与”门或者“或”门的级联输入。这种级联输入功能允许这样一种级联，即安置在每一个组合逻辑单元上的快速输出同在器件内一个邻近的组合逻辑单元的输出进行“或”操作或者“与”操作。对于级联原语而言，“与”门或者“或”门馈入级联原语以及用安放在器件内的级联原语馈给“与”门或者“或”门，用第一级符号逻辑“或”操作或者“与”操作后进入第二级。

在此说明一点：只有 FLEX 8000 器件系列和 FLEX 10K 器件系列支持 CASCADE 原语，其他系列的器件都不能使用该原语。

当用户使用一种 CASCADE 原语时，必须注意以下规则：

①一个 CASCADE 原语只能输出至一个门，或者从一个门得到输入，并且这些门只能是“与”门或者“或”门；

②一个反相“或”门可被看作为一个“与”门，反之亦然，逻辑上和“与”门(AND)相同的门有 BAND、BNAND 和 NOR，逻辑上和“或”门(OR)相同的门有 BOR、BNOR 和 NAND；

③两个 CASCADE 原语不能输出至同一个门；

④CASCADE 原语不能输出至“异或”门(XOR)；

⑤CASCADE 原语不能输出至 OUTPUT 或 OUTPUTC 引脚原语，也不能输出至寄存器；

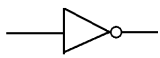
⑥执行级联的“与”门和“或”门时，De Morgan 反相理论要求在级联链路里所有原语都具有相同的类型，级联“与”门不能输出至级联的“或”门，反之亦然。

如果用户使用 CASCADE 原语不正确，它将被忽略并且编译器发出一条警告信息。

在逻辑综合过程中用 Cascade Chain 逻辑选项或逻辑综合风格(它包括 Cascade Chain 逻辑选项)，用户可以让编译器自动插入或删除 CASCADE 原语。

三、EXP 原语

图形符号：EXP　　函数原型：FUNCTION　　EXP (in)



RETURNS　　(out)；

EXP 扩展缓冲器在设计中表示期望的一种扩展乘积项。在器件中扩展乘积是反向的。

在此说明一点：只有 MAX 5000、MAX 7000 和 MAX 9000 器件系列支持 EXP 原语。在其他器件系列中 EXP 被当做反相器(NOT 门)。有关各种器件如何应用逻辑单元和扩展乘积项的具体说明请参阅相应器件的数据信息表。

是否需要应用扩展乘积项要由目标函数的逻辑极性决定。例如，如果一个 EXP 缓冲器输出至两个“与”门(即乘积项)，并且第二个“与”门有一个反相输入端，那么在逻辑综合时，输出至反相输入端的 EXP 就被删除，而产生一个正逻辑。输出至不带反相的输入端的 EXP 不会被删除，而是应用扩展乘积项去执行目标逻辑。一般情况下，由逻辑综合器确定在哪里插入或删除 EXP 缓冲器。ALTERA 建议只供有经验的 MAX + PLUS Ⅱ 设计者在他们的设计应用 EXP 原语，如图 4－3 所示。

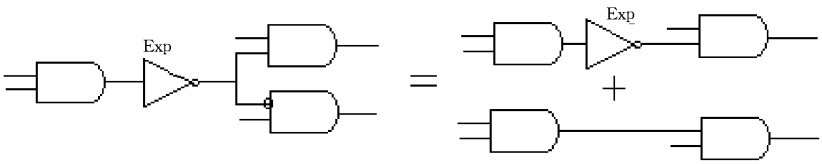


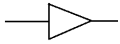
图 4－3　EXP 缓冲器的插入与删除

在包含多个逻辑阵列块(LABs)的器件中，EXP 缓冲器只能输出到同一个 LAB 中的逻辑。对于每一个 LAB 都需要复制 EXP。如果一个设计包含大量的 EXP，逻辑综合器可以把它们转换为 LCELL 缓冲器，目的是对扩展乘积项和逻辑单元的使用进行平衡。

应该注意：不要用 EXP 原语产生一个有意识的延时或异步脉冲。这种方式的延迟随温度、电源供电电压以及器件制造过程的不同而不同，因此会产生竞争条件并且产生一个不可靠的电路。

四、GLOBAL 原语

图形符号：GLOBAL　　函数原形：FUNCTION　GLOBAL　(in)



RETURNS　(out)；

GLOBAL 缓冲器表示某信号必须使用一个全局(同步)时钟(Clock)、清除(Clear)、预置(Preset)或输出使能(Output Enable)信号去替代用内部逻辑产生的或用一般 I/O 引脚驱动的信号。表 4－8 示出在不同器件系列中如何使用全局信号。

表 4－8　全局信号可用性

Device Family	Global Clock	Global Clear	Global Preset	Global Out－put Enable
Classic	√			
MAX 5000	√	√*	√*	√*
MAX 7000	√	√		√
FLEX 8000	√	√	√	√
MAX 9000	√	√		√
FLEX 10K	√	√	√	√

* 只用于 EPS 464 器件

在此应注意以下几点：

①在 Classic 和 MAX 5000 器件中，为了向后兼容，SCLK 缓冲器是可以使用的，并且只为指定全局时钟能被用在 GLOBAL 原语的地方；

②当 CLOCK 用引脚驱动时，对于要求全局时钟的寄存器可以使用 SCLK，这时必须有一个从输入引脚到寄存器的 SCLK 的直接连接；

③对于不支持全局原语的器件忽略全局原语，另外，在 MAX 5000 器件中，全局时钟和阵列局部时钟不能同时在一个逻辑阵列块(LAB)中存在。

如果一个输入引脚直接与 GLOBAL 的输入端相连，那么 GLOBAL 的输出可被作为另一个原语的时钟、清除、预置或输出使能等输入信号。这时 GLOBAL 的输出一定要与寄存器或 TRI 缓冲器的输入直接相连。如果 GLOBAL 的输出端与 TRI 缓冲器的输出使能端相连，有可能需要在输入引脚与 GLOBAL 之间加一个“非”门。

一个输入信号在到达寄存器的时钟、清除或预置端之前，或到达 TRI 缓冲器的输出使能端之前，可能要通过一个 GLOBAL 缓冲器。

全局信号传输要比局部信号(array signals)快得多，并且还会省出器件资源供其他逻辑使用。GLOBAL 应该被用来为整个设计或设计的一部分提供全局时钟。为了检查寄存器所用的时钟是否是全局时钟，用户可以查看设计的报告文件(Report File)。

如果在一个 MAX 5000 器件的设计中既包含全局时钟又包含局部时钟(异步的)，且编译器的装配模块也不能找到合适的调整方法，那么将 GLOBAL 缓冲器删除就有可能解决装配问题。如果在 MAX 7000 的设计中遇到类似问题，就用全局时钟代替局部时钟。

作为应用 GLOBAL 原语的另一种选择，用 Global Project Logic Synthesis 命令指示编译器自动选定在设计中一个已经存在的信号作为全局时钟、清除、预置或输出的使能信号。

图 4－4(a)示出一些正确使用 GLOBAL 缓冲器的例子。

图 4－4(b)示出一种 GLOBAL 缓冲器的不正确应用。

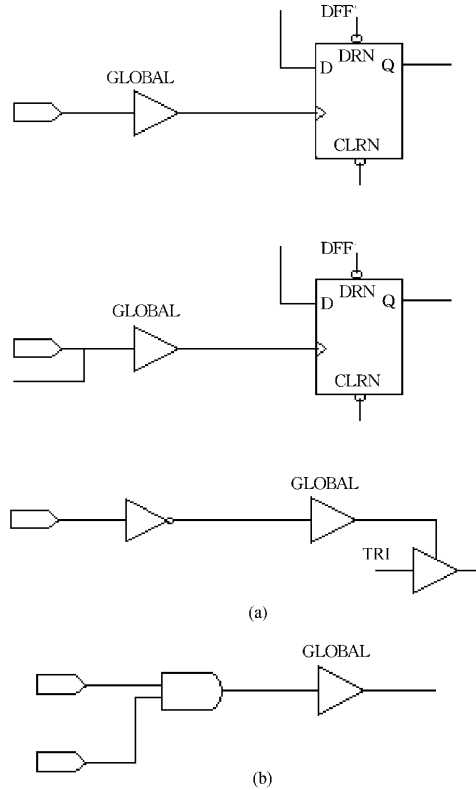
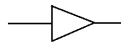


图 4－4　使用 GLOBAL 缓冲器

(a)正确使用 GLOBAL 缓冲器；(b)错误使用的 GLOBAL 缓冲器

五、LCELL 原语

图形符号：LCELL　　函数原型　FUNCTION　　LCELL(in)



RETURNS (out)；

LCELL 缓冲器为设计分配一个逻辑单元，产生一个逻辑功能的正逻辑和反逻辑，并且把它们用于器件内的所有逻辑。为了使用逻辑功能的反逻辑，逻辑缓冲器的输出必须通过一个反向门。

用户应该注意：

①MCELL 缓冲器与 LCELL 缓冲器具有相同的功能，作用是实现与 MAX + PLUS Ⅱ 早期版本的向后兼容，新设计应该完全使用 LCELL；

②一个 LCELL 缓冲器一般要占用一个逻辑单元，并且在逻辑综合过程中不会从一个设计中删除；

③用户不要使用 LCELL 原语去产生需要的延迟或异步脉冲，用这种方法产生的延迟随温度、电源电压以及器件制造过程的不同而变化，因此会产生竞争条件，并且产生一个不可靠的电路。

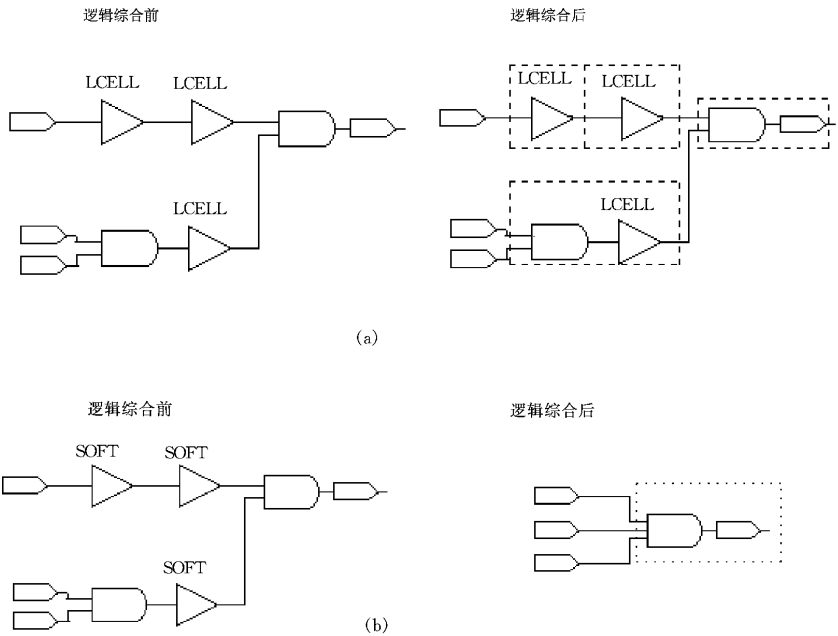


图 4－5　使用 LCELL 或 SOFT 缓冲器对逻辑综合的影响

(a)使用 LCELL 缓冲器；(b)用 SOFT 代替 LCELL

下面讨论设计中使用 LCELL 或 SOFT 缓冲器对逻辑综合的影响。图 4－5(a)中第一个设计在综合之后需要四个逻辑单元(在点线框中)和三个输入引脚。如果用 SOFT 缓冲器取代 LCELL 缓冲器，如图 4－5(b)所示，SOFT 缓冲器通过逻辑综合被删除。这个设计只需要一个逻辑单元和三个输入引脚。

如果在 Design Doctor Setting 对话框(Processing menu)中 Delay Chain 选项被打开，编译器

就会对用来产生延迟或异步脉冲的串联的 LCELL 或 EXP 原语发出一条警告信息。

在标准逻辑综合里，如图 4－6 所示，一个逻辑单元的输出反馈回它本身是非法的，除非使用 LCELL 缓冲器。当用标准综合去编译一个设计时，逻辑综合可以确定错误的反馈连接并且给出一个错误信息。如果采用多级综合，逻辑综合器自动插入一个 LCELL 缓冲器。

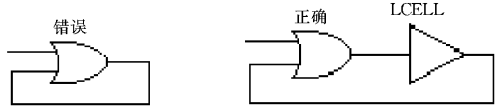
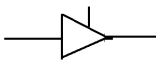


图 4－6　标准逻辑综合

在设计编译过程中，MAX＋PLUSⅡ包括不同的逻辑选项，它自动插入 LCELL 和 SOFT 缓冲器。另外，如果用户在装配时对资源定义或对器件设置使设计遇到困难，那么可以采用编译器的 Fitter Settings 命令(Processing menu)直接编译，自动地把 LCELL 缓冲器插入设计中。

六、OPNDRN 原语

图形符号：OPNDRN



函数原型　FUNCTION　OPNDRN (in)

RETURNS　(out)；

类似于 TRI 原语，OPNDRN 原语有一个单一的输入和一个单一的输出。OPNDRN 原语等效于 TRI 原语。它的输出使得任一信号作为输入成为可能，但是它的原语输入是用一个 GND 原语供给。

如果原语输入 OPNDRN 是用低电平，那么输出也是低电平 如果输入是高电平，那么输出将是一个高阻抗的逻辑电平。

用户应注意 所提供的 OPNDRN 原语仅适合于 FLEX 10K 器件系列，对于其他器件它被转换为一个 TRI 原语。

如果在 Global Project Logic Synthesis 对话框(Assign menu)内打开 Automatic Open-Drain Pins 选项对一个 FLEX 10K 进行设计，编译器把下面的结构转换为 OPNDRN 原语：

①一个 TRI 原语的输出使得用任意信号作为输入成为可能；

②一个 TRI 原语的输出使得用它的最初的输入的组合作为输入成为可能。

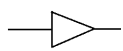
当使用 OPNDRN 缓冲器时，必须遵守下列规则：

①一个 OPNDRN 缓冲器仅能驱动一个 BIDIR 或 BIRDIRC 引脚；

②如果一个 OPNDRN 缓冲器馈入逻辑，那么也必须送给一个 BIDIR 或 BIDIRC 引脚，它可以不给出任何其他输出。

七、SOFT 原语

图形符号：SOFT



函数原型：　FUNCTION　SOFT(in)

RETURNS　(out)；

SOFT 缓冲器表示在设计中可能需要的逻辑单元。在设计处理过程中，逻辑综合器对原语的输入逻辑进行检查，以确定是否需要一个逻辑单元。如果需要，则把 SOFT 缓冲器转变成一个 LCELL 如果不需要，SOFT 缓冲器将被删除。

图 4－7 说明了在设计中采用 SOFT 缓冲器对逻辑综合的作用。在第一个设计中(图 4－

7(a)),逻辑综合将删除 SOFT 缓冲器,这个设计占用一个逻辑单元。在第二个设计(图 4－7(b))中,将删除一个 SOFT 缓冲器并把另一个转换为一个 LCELL 缓冲器。这个 LCELL 缓冲器将减少该设计的复杂度,即乘积项的数目。第二个设计也占用一个逻辑单元。

如果编译器指示某个设计太复杂,那么用户可以在设计中插入 SOFT 缓冲器去禁止做逻辑展开。例如,用户可以在一个逻辑函数的连接输出处加一个 SOFT 缓冲器隔离两个组合电路,如图 4－8 所示。在设计编译过程中 MAX＋PLUS Ⅱ 包含的逻辑选项可自动插入或忽略 SOFT 和 LCELL 缓冲器。

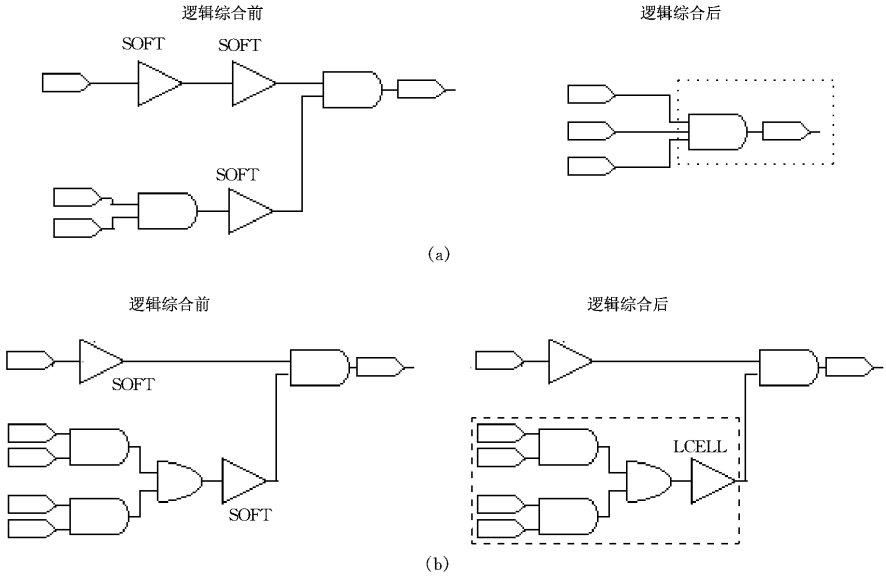
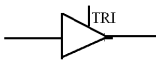


图 4－7　SOFT 缓冲器对逻辑综合的作用
(a)SOFT 缓冲器的删除;(b)对 SOFT 缓冲器的不同处理

八、TRI 原语

图形符号:TRI



函数原型　FUNCTION　TRI (in,oe)
RETURNS　(out);

TRI 原语是一个具有一个输入端、一个输出端和一个输出使能信号端的三态缓冲器。如果输出使能信号为主电平,那么输出端将由输入端驱动。这个输出使能的默认值为 VCC。

如果一个 TRI 缓冲器的输出使能与 VCC 或一个最终结果为真的逻辑函数相连,那么 TRI 缓冲器在逻辑综合过程中可能被转化为 SOFT 缓冲器。

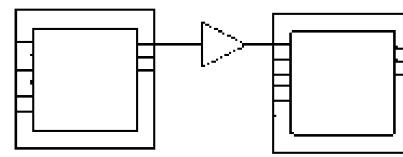


图 4－8　将 SOFT 缓冲器插在两个组合电路之间禁止做逻辑展开(尽管这样做可以引起一个附加的逻辑单元)

在使用 TRI 缓冲器时,用户必须遵守下列规则:

①一个 TRI 缓冲器只能驱动一个 BIDIR 或 BIDIRC 引脚,如果在 TRI 缓冲器之后包括反馈,用户必须采用一个 BIDIR 或 BIDIRC 引脚;

②如果一个 TRI 缓冲器输出到一个逻辑体,它也必须输出到一个 BIDIR 或 BIDIRC 引脚,但是,如果它输出到 BIDIR 或 BIDIRC 引脚,则不必输出到其他任意输出引脚上;

③在输出使能信号没有连接到 VCC 时,TRI 缓冲器的输出一定要连到 OUTPUT、OUTPUTC、BIDIR 或 BIDIRC 引脚上,内部信号不可以为三态。

图 4-9 和图 4-10 是正确使用 TRI 缓冲器的例子。

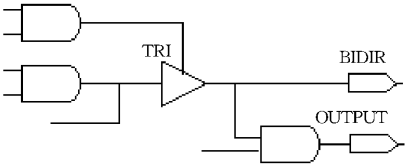


图 4-9 TRI 缓冲器可以驱动一个双向的引脚

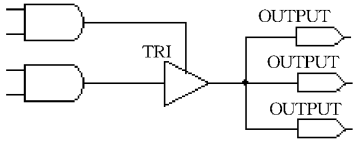


图 4-10 TRI 缓冲器可以驱动多个输出引脚

图 4-11 和图 4-12 是不正确使用 TRI 缓冲器的例子。

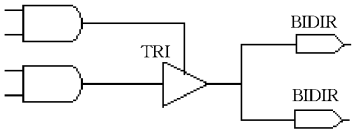


图 4-11 TRI 缓冲器不可以驱动多个双向引脚

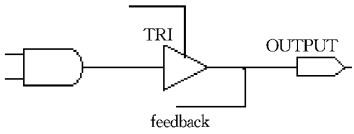


图 4-12 如果在三态缓冲器之后有反馈时,TRI 缓冲器不能驱动一个输出引脚

九、触发器和锁存器原语

表 4-9 列出 MAX+PLUS Ⅱ 的触发器和锁存器原语以及它们的函数原型。所有的触发器都是上升沿触发,而锁存器是电平触发。

用户应该注意:当锁存器使能或时钟使能(ena)输入信号为高电平时,触发器或锁存器会把数据输入端的信号传到 q 端。当这个 ena 输入信号变低后,q 的状态将保持住,而与数据输入无关。

对于没有提供锁存器使能和(或)时钟使能信号的器件,在逻辑综合时产生一些逻辑等式。该等式中包含有时钟使能端的触发器和带锁存器使能端的锁存器的功能。这些逻辑等式会正确实现用户设计中指定的逻辑。

表 4-9 MAX+PLUS Ⅱ 触发器和锁存器

原语(Primitive)	ADHL 函数原型
LATCH	FUNCTION LATCH (d,ena) RETURNS (q);
DFF	FUNCTION DFF (d,clk,clrn,prn) RETURNS (q);

续表

原语(Primitive)	ADHL 函数原型
DFFE	FUNCTION DFFE(d,clk,clrn,prn,ena) RETURNS (q);
JKFF	FUNCTION JKFF(j,k,clk,clrn,prn) RETURNS (q);
JKFFE	FUNCTION JKFFE(j,k,clk,clrn,prn,ena) RETURNS (q);
SRFF	FUNCTION SRFF(s,r,clk,clrn,prn) RETURNS (q);
SRFFE	FUNCTION SRFFE(s,r,clk,clrn,prn,ena) RETURNS (q);
TFF	FUNCTION TFF(t,clk,clrn,prn) RETURNS (q);
TFFE	FUNCTION TFFE(t,clk,clrn,prn,ena) RETURNS (q);

注:clk=寄存器时钟输入端;clrn=清零信号输入端;d、j、k、r、s、t=来自逻辑阵列的数据输入端;ena=锁存器使能输入端或时钟使能输入端;prn=预置信号输入端;q=输出端。

十、原语/端口的互连

在设计文件中并不是所有的原语和端口都可以连接到所有其他的原语和端口上。表 4-10 给出所有 AHDL 原语和端口的正确连接方式。

表 4-10 原语/端口互连(一)

源(Source)	说明(Destination)					
	OUTPUT	BIDIR	TRI(1)	GLOBAL	LCELL	EXP
INPUT	Y	N	Y	Y	na	na
OUTPUT(3)	N	N	N	N	N	N
BIDIR	N	N	Y	N	na	na
TRI	Y	Y	N(4)	na	N(4)	N(4)
GLOBAL(5)	na	N	(6)	Y	na	na
LCELL	Y	N	Y	na	na	na
EXP	na	N	na	N	na	na
SOFT	Y	N	na	N	na	na
VCC	Y	N	Y	N	na	na
GND	Y	N	Y	N	na	na
Logic	Y	N	Y	N	Y	Y
Register Output	Y	N	Y	N	na	na
CARRY	na	na	na	na	na	na
CASCADE	na	na	na	na	na	na
OPNDRN	Y	Y	N(4)	na	N(4)	N(4)

注:Y—互连合法;N—互连不合法;na—互连合法但不合理或执行逻辑可能无效。

表 4－10　原语/端口互连（二）

源 (Source)	说明 (Destination)					
	SOFT	Logic	INPUT(2)	CARRY	CASCADE	OPNDRN
INPUT	na	Y	Y	na	na	Y
OUTPUT(3)	N	N	N	na	na	N
BIDIR	na	Y	Y	na	na	Y
TRI	N(4)	N(4)	N(4)	na	na	N(4)
GLOBAL(5)	na	na	na	Y	na	
LCELL	na	Y	na	na	na	Y
EXP	na	Y	na	na	na	na
SOFT	na	Y	na	na	na	na
VCC	na	Y	Y	na	na	Y
GND	na	Y	Y	na		Y
Logic	Y	Y	Y	Y	Y	Y
Register Output	Y	Y	Y	na	na	Y
CARRY	na	Y	na	na	na	na
CASCADE	na	Y	na	na	na	na
OPNDRN	N(4)	N(4)	N(4)	na	na	N(4)

注：①对 TRI 的输入包括双数据和输出使能；②输入端只能由驱动引脚或本层内的较高电平供给；③OUTPUT（或 OUT）、OUTPUTC、BIDIR（或 INOUT），and BIDIRC 原语/端口只能驱动输出给器件引脚或本层内的较高电平；④只有当 TRI 或 OPNDRN 也被连接到一个 BIDIR（或 INOUT）或 BIDIRC 原语/端口时，这些连接才改变合法性（Y）或不合理性（na）；⑤用 GLOBAL 说明在 MAX＋PLUS（DOS）中的 SCLK 设计，现在各种设计只希望用 GLOBAL；⑥在 FLEX 8000 和 FLEX 10K 器件中把一个 GLOBAL 输出连接到 TRI 输出使能输入是合法的，而在另外器件内连到输出使能是被忽略的；⑦在 3.0 版 MAX＋PLUS Ⅱ 设计文件中，MCELL 被解释为 LCELL，现在的设计只希望采用 LCELL。

表 4－11　原语/端口对寄存器的连接

源(Source)	寄存器 (Register)			
	INPUT(1)	CLK	PRN	CLRN
INPUT	Y	Y	Y	Y
OUTPUT(2)	N	N	N	N
BIDIR(2)	Y	Y	Y	Y
TRI	N(3)	N(3)	N(3)	N(3)
GLOBAL(4)	na	Y	Y	Y
LCELL(5)	na	Y	Y	Y
EXP	na	na	na	na
SOFT	na	Y	Y	Y
VCC	Y	N	Y	Y
GND	Y	N	Y	Y
Logic	Y	Y	Y	Y
Register Output	Y	Y	Y	Y

续表

源(Source)	寄存器 (Register)			
	INPUT(1)	CLK	PRN	CLRN
CARRY	na	na	na	na
CASCADE	na	na	na	na
OPNDRN	N(3)	N(3)	N(3)	N(3)

注:①输入端只能由器件引脚或本层次内的较高电平供给;②OUTPUT(或 OUT)、OUTPUTC、BIDIR(或 INOUT)和 BIDIRC 原语/端口只能驱动输出到器件引脚或本层次内的较高电平;③只有当 TRI 或 OPNDRN 也被连接到一个 BIDIR(或 INOUT)或 BIDIRC 原语/端口时,这些连接将改变合法(Y)或不合理(na);④在 MAX + PLUS(DOS)中 SCLK 设计被解释为 GLOBAL,现在的设计只希望应用 GLOBAL;⑤在早期 3.0 版的 MAX + PLUS Ⅱ 设计文件中,MCELL 被解释为 LCELL,现在的设计只希望应用 LCELL。

Y—互连是合法的　　N—互连是不合法的　　na—互连是合法但不合理或者可能执行的逻辑无效

4.3.15 强函数(Megafunctions)

MAX + PLUS Ⅱ 强函数是一种复杂的逻辑函数的集合,它包括参数设置模式的库函数(LPM),可在逻辑设计中使用它。这些强函数对于 ALTERA 器件在优化逻辑功能方面具有特殊的功效,MAX + PLUS Ⅱ 安装程序在 \ maxplus2 \ max2lib \ mega-lpm 目录中自动地安装这些强函数。这些目录也包括具有每种强函数的函数原型的一种包含文件(. inc)。在一个 UNIX 工作站上,maxplus2 目录是一个关于这个用户目录的子目录。

在所有 MAX + PLUS Ⅱ 逻辑设计中用户可以灵活地应用这些强函数。当 MAX + PLUS Ⅱ 编译器分析一个逻辑电路时,它自动地移去所有不用的门和触发器,使这种设计效率不被降低。

表 4 - 12 说明全部 MAX + PLUS Ⅱ 的强函数。当前包含在 LPM2.0.1/2.1.0 标准中的 LPM 函数的名字用‘lpm_.’起头。关于这些强函数的详细的信息可利用在 MAX + PLUS Ⅱ 的‘帮助’获得。

表 4 - 12　MAX + PLUS Ⅱ　Megafunctions (Part 1 of 2)

类　型	名　称	说　　明
Gates(门)	lpm_and	参数设置的与门
	lpm_bustri	参数设置的三态缓冲器
	lpm_clshift	参数设置的组合移位模块
	lpm_constant	参数设置的恒定振荡器模块
	lpm_decode	参数设置的译码器模块
	lpm_inv	参数设置的反相器模块
	lpm_mux	参数设置的多路选择器模块
	lpm_or	参数设置的‘或’门
	lpm_xor	参数设置的‘异或’门

表 4－12　MAX＋PLUS Ⅱ　Megafunctions（Part 2 of 2）

类　型	名　称	说　　明
运算部件（Arithmetic Components）	lpm_abs	参数设置的绝对值
	lpm_add_sub	参数设置的加法/减法模块
	lpm_decode	参数设置的比较器模块
	lpm_counter	参数设置的计数器模块
	lpm_mult	参数设置的相乘器模块
存储部件（Storage Components）	lpm_dff	参数设置的 D 类触发器和移位寄存器模块
	lpm_latch	参数设置的锁存器组件
	lpm_ram_dg	具有独立输入和输出端口的随机存取存储器
	lpm_ram_io	具有一个单一 I/O 端口的随机存取存储器
	lpm_rom	只读存储器
	lpm_tff	参数设置的 T－触发器组件
	csdpram	循环分配双端口随机存取存储器
	csfifo	循环分配先进先出
另外功能（Other Functions）	a6502	6502 微处理器
	ntsc	NTSC 视频控制信号发生器
	pll	上升沿和下降沿检测器

4.3.16　老式宏函数（old-style Macrofunctions）

MAX＋PLUS Ⅱ的老式宏函数是一种高水平的设计模块，它们可以用在逻辑设计中。在安装过程中 MAX＋PLUS Ⅱ自动将这些宏函数安装在 \ maxplus2 \ max2lib 目录及其子目录中，在另一个被自动安装的目录 \ maxplus2 \ max2inc 中有包含每个宏函数的函数原型的 Include 文件（. inc）。在 UNIX 工作站上，maxplus2 目录是这个或用户目录的一个子目录。

在所有 MAX＋PLUS Ⅱ的逻辑设计中用户可自由地使用这些宏函数。在 MAX＋PLUS Ⅱ编译器对一个逻辑电路进行分析的过程中，它会自动将所有无用的门和触发器删除，因此保证不降低这个设计的有效性。所有的输入端口都有默认值，因此所有不用的输入端允许不作任何连接。

表 4－13 介绍了所有 MAX＋PLUS Ⅱ的老式宏函数。有关这些宏函数的详细信息用户可阅读 MAX＋PLUS Ⅱ的在线帮助。

表 4－13　MAX＋PLUSⅡ宏函数

宏函数类型	宏函数名　称	说　　明
Adder	8fadd	8－bit full adder
	8fadde	8－bit full adder
	7480	Gated full adder
	7482	2－bit binary full adder
	7483	4－bit binary full adder with fast carry
	74183	Dual carry-save full adder

续表

宏函数类型	宏函数名 称	说 明
	74283	4－bit full adder with fast carry
	74285	4－bit adder/subtractor with Clear
Arithmetic Logic Unit	74181	Arithmetic logic unit
	74182	Look-ahead carry generator
	74301	Arithmetic logic unit/function generator
	74382	Arithmetic logic unit/function generator
Application Specific	ntsc	NTSC video control signal generator
	pll	Rising－ and falling-edge detector
Buffer	Btri	Active-low tri-state buffer
	74240	Octal inverting tri-state buffer
	74240b	Octal inverting tri-state buffer with 2 section
	74241	Octal tri-state buffer
	74241b	Octal tri-state buffer with 2 section
	74244	Octal tri-state buffer
	74244b	Octal tri-state buffer with 2 section
	74365	Hex tri-state buffer
	74366	Hex inverting tri-state buffer
	74367	Hex tri-state buffer
	74368	Hex inverting tri-state buffer
	74465	Octal tri-state buffer
	74466	Octal inverting tri-state buffer
	74467	Octal tri-state buffer
	74468	Octal inverting tri-state buffer
	74540	Octal inverting tri-state buffer
	74541	Octal tri-state buffer
Comparator	8mcomp	8－bit magnitude comparator
	8mcompb	8－bit magnitude comparator
	b	4－bit magnitude comparator
	7485	8－bit identity comparator
	74518	8－bit identity comparator
	74518b	8－bit magnitude/identity comparator
	74684	8－bit magnitude/identity comparator
	74686	8－bit identity comparator
	74688	
Converter	74184	BCD-to-binary converter
	74185	Binary-to-BCD converter
Counter	gray4	Gray code counter
	unicnt	Universal 4－bit up/down counter left/right shift register with asynch. set&load，clear&cascade
	16cudslr	16－bit binary up/down counter left/right shift register with asynch. set
	16cudsrb	16－bit binary up/down counter left/right shift register with asynch. clear，&asynch. set
	4count	4－bit binary up/down counter with synchronous load（LDN），asynch. clear，&asynch. load（SETN）
	8count	8－bit binary up/down counter with synchronous load（LDN），asynch. clear，&asynch. load（SETN）
	7468	Dual decade counter
	7469	Dual binary counter

续表

宏函数类型	宏函数名 称	说 明
	7492	Decade or binary counter with clear&set-to－9
	7493	4－bit binary counter
	74143	4－bit counter/latch,7－segment driver
	74160	4－bit decade counter with synch. load & asynch. clear
	74161	4－bit binary up counter with synch. load & asynch. clear
	74162	4－bit decade up counter with synch. load & asynch. clear
	74163	4－bit binary up counter with synch. load & asynch. clear
	74168	Synch. 4－bit decade up/down counter
	74169	Synch. 4－bit binary up/down counter
	74176	Presettable decade counter
	74177	Presettable binary counter
	74190	4－bit decade up/down counter with asych. load
	74191	4－bit binary up/down counter with asych. load
	74192	4－bit decade up/down counter with asych. clear
	74193	4－bit binary up/down counter with asych. clear
	74196	Presettable decade counter
	94197	Presettable binary counter
	94290	Decade counter with Clear
	74292	Programmable frequency divider/digital timer
	74293	Binary counter with Clear
	74294	Programmable frequency divider/digital timer
	74390	Dual decade counter
	74393	Dual 4－bit counter with asych. clear
	94490	Dual 4－bit decade counter
	74568	Decade up/down counter with asych. load & Clear & asych. clear
	74569	Binary up/down counter with asych. load & Clear & asych. clear
	74590	8－bit binary counter with tri-state output registers
	74582	8－bit binary counter with input register
	74668	Synch. decade up/down counter
	74669	Synch. 4－bit binary up/down counter
	74690	Synch. decade counter with output registers,multiplexed tri-state outputs & asych. Clear
	74691	Synch. binary counter with output registers,multiplexed tri-state outputs & asych. Clear
	84693	Synch. binary counter with output registers,multiplexed tri-state outputs & synch. Clear
	74696	Sysch. decade up/down counter with output registers, multiplexedtri-state outputs & asych. Clear
	74697	Sysch. binary up/down counter with output registers, multiplexedtri-state outputs & asych. Clear
	74698	Sysch. decade up/down counter with output registers, multiplexedtri-state outputs & synch. Clear
	74699	Sysch. binary up/down counter with output registers, multiplexedtri-state outputs & synch. Clear
16cudslr	16mux	4－bit binary-to－16－line decoder
	16nmux	4－bit binary-to－16－line decoder
	7442	1－ line-to－10－line BCD-to-decimal decoder
	7443	Excess－3－to-decimal decoder
	7444	Excess－3－Gray-to-decimal decoder
	7445	BCD-to-decimal decoder

续表

宏函数类型	宏函数名 称	说 明
	7446	BCD-to－7－segment decoder
	7447	BCD-to－7－segment decoder
	7448	BCD-to－7－segment decoder
	7449	BCD-to－7－segment decoder
	74137	3－line-to－8－line decoder with address latches
	74138	2－ line-to－8－line decoder
	74139	Dual 2－line-to－4－line decoder
	74145	BCD-to-decimal decoder
	74154	4－line-to－16－line decoder
	74155	Dual 2－line-to－4－line decoder/demultiplexer
	74156	Dual 2－line-to－4－line decoder/demultiplexer
	74246	BCD-to－7－segment decoder
	74247	BCD-to－7－segment decoder
	74248	BCD-to－7－segment decoder
	74445	BCD-to-decimal decoder
Digital Filter	74297	Digital phase-locked loop filter
EDAC	74630	16－bit parallel error detection & correction circuit
	74636	8－bit parallel error detection & correction circuit
Encoder	74147	10－line-to－4－line BCD encoder
	74148	8－line-to－3－line octal encoder
		8－line-to－3－line priority encoder with tri-state outputs
Frequency Divider	freqdiv	Frequency divider
	7456	Frequency divider
	7457	Frequency divider
Latch	explatch	Latch implemented with expanders
	inpltch	Inputs latch
	nandltch	/SR NAND latch with expanders
	lorrltch	SR NOR latch with expanders
	7475	4－bit bistable latch
	7477	4－bit bistable latch
	74116	Dual 4－bit latch with Clear
	74259	8－bit 2－stage pipelined latch
	74279	Quad/SR latch
	74373	Octal transparent D-type latch with tri-state outputs
	74373b	Octal transparent D-type latch with tri-state outputs
	74375	4－bit bistable latch
	74549	8－bit 2－stage pipelined latch
	74604	Octal 2－input multiplexed latch with tri-state outputs
	74841	10－bit D-type latch with tri-state outputs
	74841b	10－bit D-type latch with tri-state outputs
	74842	10－bit D-type latch with tri-state outputs
	74842b	10－bit D-type inverting latch with tri-state outputs
	74843	9－bit bus interface D-type latch with tri-state outputs
	74844	9－bit bus interface D-type inverting latch with tri-state outputs
	74845	8－bit bus interface D-type latch with tri-state outputs
	74846	8－bit bus interface D-type inverting latch with tri-state outputs
	74990	8－bit transparent read-back latch

续表

宏函数类型	宏函数名 称	说 明
Multiplier	mult2	2 - bit sign magnitude multipiler
	mult24	2 - bit-by - 4 - bit parallel binary multiplier
	mult4	4 - bit parallel binary multiplier
	mult4b	4 - bit parallel binary multiplier
	tmult4	4 - bit-by - 4 - bit parallel binary multiplier
	7497	Synch. 6 - bit rate multiplier
	74261	2 - bit parallel binary multiplier
	74284	4 - bit-by - 4 - bit parallel binary multiplier(upper 4 bits of result)
	74285	4 - bit-by - 4 - bit parallel binary multiplier(lower 4 bits of result)
Multiplexer	21mux	2 - line-to - 1 - line multiplexer
	81mux	8 - line-to - 1 - line multiplexer
	161mux	16 - line-to - 1 - line multiplexer
	2×8mux	2 - line-to - 1 - line multiplexer for 8 - bit buses
	74151	8 - line-to - 1 - line multiplexer
	74151b	8 - line-to - 1 - line multiplexer
	74153	Dual 4 - line-to - 1 - line multiplexer
	74157	Quad 2 - line-to - 1 - line multiplexer
	74158	Quad 2 - line-to - 1 - line multiplexer with Inverting outputs
	74251	8 - line-to - 1 - line data selector with tri-state outputs
	74253	Dual 4 - line-to - 1 - line data selector with tri-state outputs
	74257	Quad 2 - line-to - 1 - line multiplexer with tri-state outputs
	74258	Quad 2 - line-to - 1 - line multiplexer with inverting tri-state outputs
	74298	Quad 2 - input multiplexer with storage
	74352	Dual 4 - line-to - 1 - line data selector/multiplexer with inverting outputs
	74353	Dual 4 - line-to - 1 - line data selector/multiplexer with tri-state inverting outputs
	74354	8 - line-to - 1 - line data selector/multiplexer/register with tri-state outputs
	74356	8 - line-to - 1 - line data selector/multiplexer/register with tri-state outputs
	74398	Quad 2 - input multiplexer with storage
	74399	Quad 2 - input multiplexer with storage
Parity Generator/ Checker	74180	9 - bit odd/even parity generate/checker
	74180b	9 - bit odd/even parity generate/checker
	74280	9 - bit odd/even parity generate/checker
	74280b	9 - bit odd/even parity generate/checker
Rate Multiplier	74167	Synch. decade rate multiplier
Register	enadff	Enable D-type flipflop
	expdff	D-type flipflop Implememted with expander(or with DFF primitive for FLEX 8000 projects)
	7470	AND-gated JK flipflop with Preset & Clear
	7471	JK flipflop with Preset
	7472	AND-gated JK flipflop with Preset & Clear
	7473	Dual JK flipflop with Clear
	7474	Dual D-type flipflop with asynch. Preset & asynch. Clear
	7476	Dual JK flipflop with asynch. Preset & asynch. Clear
	7478	Dual JK flipflop with asynch. Preset ,common Clear & common Clock
	74107	Dual JK flipflop with Clear
	74109	Dual JK flipflop with Preset & Clear
	74112	Dual JK negative-edge-triggered flipflop with Preset & Clear

续表

宏函数类型	宏函数名 称	说 明
	74113	Dual JK negative-edge-triggered flipflop with Preset
	74114	Dual JK negative-edge-triggered flipflop with Preset,common Clear & common Clock
	74171	Quad D-type flipflop with Clear
	74172	Multi-port register file with tri-state outputs
	74173	4 - bit D-type register
	74174	Hex D flipflop with common Clear
	74174b	Hex D-type flipflop with common Clear
	74175	Quad D-type flipflop with common Clock & Clear
	74273	Octal D-type flipflop with asynch. Clear
	74273b	Octal D-type flipflop with asynch. Clear
	74276	Quad JK flipflop register with common Preset & Clear
	74374	Octal D-type flipflop with tri-state outputs & Output Enable
	74374b	Octal D-type flipflop with tri-state outputs & Output Enable
	74376	Quad JK flipflop with common Clock & common Clear
	74377	Octal D-type flipflop with Enable
	74377b	Octal D-type flipflop with Enable
	74378	Hex D-type flipflop with Enable
	74379	Quad D-type flipflop with Enable
	74396	Octal storage register
	74548	8 - bit 2 - stage pipelined register with tri-state outputs
	74670	4 - bit by 4 - bit register file with tri-state outputs
	74821	10 - bit bus interface flipflop with tri-state outputs
	74821b	10 - bit D-type flipflop with tri-state outputs
	74822	10 - bit bus interface flipflop with tri-state inverting outputs
	74822b	10 - bit D-type inverting flipflop with tri-state inverting outputs
	74823	9 - bit bus interface flipflop with tri-state outputs
	74823b	9 - bit D-type flipflop with tri-state outputs
	74824	9 - bit bus interface flipflop with tri-state inverting outputs
	74824b	9 - bit D-type inverting flipflop with tri-state inverting outputs
	74825	8 - bit bus interface flipflop with tri-state outputs
	74825b	Octal D-type flipflop with tri-state outputs
	74826	9 - bit bus interface flipflop with tri-state inverting outputs
	74826b	Octal D-type inverting flipflop with tri-state inverting outputs
Shift Register	barrelst	8 - bit barrel shifter
	barrelstb	8 - bit barrel shifter
	7491	Serial-in serial-out shift register
	7494	4 - bit shift register with asynch. Preset & asynch. Clear
	7495	4 - bit parallel-access shift register
	7496	5 - bit shift register
	7499	4 - bit shift register with/JK serial inputs & parallel outputs
	74164	Serial-in parallel-out shift register
	74164b	Serial-in parallel-out shift register
	74165	Parallel load 8 - bit shift register
	74165b	Parallel load 8 - bit shift register
	74166	8 - bit shift register with Clock inhibit
	74178	4 - bit shift register
	74179	4 - bit shift register with Clear

续表

宏函数类型	宏函数名 称	说 明
	74194	4 – bit bidirectional shift register with parallel load
	74195	4 – bit parallel-access shift register
	74198	8 – bit bidirectional shift register
	74199	8 – bit parallel-access shift register
	74295	4 – bit right-shift left-shift register with tri-state outputs
	74299	8 – bit universal shift/storge register
	74350	4 – bit shift register with tri-state outputs
	74395	4 – bit cascadable shift register with tri-state outputs
	74589	8 – bit shift register with input latches & tri-state outputs
	74594	8 – bit shift register with output latches
	74595	8 – bit shift register with output latches & tri-state outputs
	74597	8 – bit shift register with input register
	74671	4 – bit universal shift register/latch with direct-overriding Clear & tri-state outputs
	74672	4 – bit universal shift register/latch with synch. Clear & tri-state outputs
	74673	16 – bit shift register
	74674	16 – bit shift register
Storage Register	7498	4 – bit data selector/storage register
	74298	4 – bit cascadable prioirty register
SSI Fnction	cbuf	Complementtary buffer
	inhb	Inhibit gate
	7400	NAND2 gate
	7402	NOR2 gate
	7404	NOT gate
	7408	AND2 gate
	7410	NAND2 gate
	7411	AND3 gate
	7420	NAND4 gate
	7421	AND4 gate
	7423	Dual 4 – input NOR gate with strobe
	7425	Dual 4 – input NOR gate with strobe
	7427	NOR3 gate
	7428	Quad 2 – input positive NOR buffer
	7430	NAND8 gate
	7432	OR2 gate
	7437	Quad 2 – input positive NAND buffer
	7440	Dual 4 – input positive NAND buffer
	7450	Dual 2 – wide 2 – input AND-OR-INVERT gate
	7451	Dual AND-OR-INVERT gate
	7452	AND-OR gate
	7453	Expandable 4 – wide AND-OR-INVERT gate
	7454	4 – wide AND-OR-INVERT gate
	7455	2 – wide ,4 – input AND-OR-INVERT gate
	7464	4 – 2 – 3 – 2 input AND-OR-INVERT gate
	7486	XOR gate
	74133	13 – input NAND gate
	74134	12 – input NAND gate with tri state output
	74135	Quad XOR/XNOR gates

续表

宏函数类型	宏函数名称	说明
	74260	Dual 5 - input positive NOR gates
	74386	Quadruple XOR gate
True/Complement I/O Element	7487	4 - bit true /complement I/O element
	74265	Quad complementary output elements

4.3.17 端口

端口是一种具有逻辑功能的输入或输出端,它可以出现在子程序段和逻辑段中。在子设计段中它说明当前文件的输入或输出端口 在逻辑段中它采用原语实例或低层设计文件的输入或输出端口。

一、当前文件的端口

当前文件的输入或输出端口在子设计段中按下面格式描述:

<port name> :<port type> [=<default port value]

下面的端口类型是可用的:

INPUT　　MACHINE　INPUT

OUTPUT　　MACHINE　OUTPUT

BIDIR

如果一个 TDF 文件是一个层次设计中的顶层文件,那么端口名称与引脚名称是相同的。还可以为 INPUT 和 BIDIR 端口类型设定默认值 VCC 或 GND。如果一个 TDF 文件的实例要在更高层次设计中使用,只有当这个端口处于不连接状态时,才使用这个默认值。

下面的子设计段中说明了该文件的输入、输出和双向端口:

```
SUBDESIGN          top
(
    foo, bar, clk1, clk2, c[4..0]          :INPUT = VCC;
    % VCC is default   port value %
      a0,a1,a2,a3,a4                        :OUTPUT;
    b[7..0]                                 :BIDIR;
)
```

在子设计段中,通过把输入和输出端口定义为状态机输入(MACHINE INPUT)或状态机输出(MACHIN OUTPUT),用户就可以在 TDFs 文件与其他设计文件之间输入或输出状态机。在代表该文件的函数原型说明中必须指明哪些端口是状态机。状态机输入端口和状态机输出端口只能用在一个设计层次结构的下层文件中。

二、实例的端口

在逻辑段中连接的端口是一种逻辑函数的实例的输入端口或输出端口。为把一个 TDF 文件的一个逻辑函数同其他部分连接起来,用户可以用一个内部直接引用、实例说明或用一个状态机描述来说明一个状态机插入该函数的一个实例,而后才能在逻辑段中应用该函数的端口。

如果用户采用与端口位置相联系的一种内部直接引用建立一个逻辑函数的实例,那么端

口的顺序比端口的名称更重要。端口的顺序定义在函数的函数原型中。

如果用户采用与同命名的端口相联系的一种实例说明或一种内部直接引用来建立一种逻辑函数的实例,那么端口的名称比它的顺序更重要。

在下面的例子里,一个 D 触发器的实例被描述为变量段内的变量 reg,而后在逻辑段中采用:

```
VARLABLE
        reg :DFF;
BEGIN
        reg.clk = clock
        reg.d = data_input
        output = reg.q
END;
```

于是,端口的各种名称可以被连接到逻辑段中的其它结点上,格式如下:

< instance name > . < port name > = <node name >

<instance name>是一个用户定义的一种函数的名称。<port name>等同于端口名,它被描述为一个下层 TDF 文件的子设计段中文件的一种输入或输出,或者等同于其他设计文件中的一种引脚名称。这种<port name>与符号的引脚名称相等,它代表在一个图形设计文件(.gdf)中设计文件的一种实例。

如上例所述,如果用户应用一个实例说明来建立一个逻辑函数的实例,那么在逻辑函数的设计文件中,各端口的名称是重要的。对于内部逻辑函数引用,采用命名的端口组合的右侧同样是正确的(所有内部直接引用的左侧采用位置端口组合)。下面例子表示关于 21mux 宏函数和一个内部直接引用的函数原型。它应用于命名的端口组合:

```
FUNCTION  21mux(s,a,b)
    RETURNS    (y);
.
.
.
BEGIN
    output = 21mux(.s=select,.b=dataB,.a=dataA);
END;
```

结点 output、select、dataA 和 dataB 被连接到 21mux 宏函数的 y、s、a 和 b 端口。然后,在一个内部直接引用里采用命名的端口组合。这些端口在等号的右边按下面格式列出:

.<port name> = <node name>

对比看,如果用户应用一个内部直接引用同位置端口的组合建立一个逻辑函数的实例,那么在这个内部直接引用里按结点顺序排列是重要的,而举例说明的逻辑函数的端口名不重要。这种端口的顺序被定义在关于这个函数的函数原型里。下面的例子示出对于同样的 21mux macrofunction 宏函数的一种内部直接引用。它应用的位置端口联合如下:

```
BEGIN
    output = 21MUX(Select, dataA, dataB);
```

END；

所有 ALTERA 提供的逻辑函数皆已预先定义了端口(引脚)名，它示于函数原型里。通常使用的原语端口名示于表 4－14 中。

表 4－14　一般应用的端口

端口名	说　　明
.q	一个触发器或锁存器的输出端
.d	一个 D 触发器或锁存器的数据输入端
.t	T 触发器的数据输入端
.j	JK 触发器的 J 输入端
.k	JK 触发器的 K 输入端
.s	SR 触发器的设置输入端
.r	SR 触发器的清除输入端
.clk	触发器的时钟输入端
.ena	触发器的时钟使能输入端 锁存器的锁存使能输入端 状态机的使能输入端
.prn	触发器的低有效预置信号输入端
.clrn	触发器的低有效清零信号输入端
.reset	状态机的高有效复位信号输入端
.oe	TRI 原语的输出使能信号输入端
.in	CARRY、CASCADE、EXP、TRI、OPNDRN、SOFT、GLOBAL 和 LCELL 原语的主输入端
.out	CARRY、CASCADE、EXP、TRI、OPNDRN、SOFT、GLOBAL 和 LCELL 原语的输出端

4.3.18　参数

一种强函数(或宏函数)的各种属性确定逻辑或者完成某种功能，也就是说，各类特性确定规格、性能或者执行的功能。例如，经常用参数确定一条总线的宽度。

一个参数化函数是用一个或多个参数去控制该函数性能的一些逻辑函数，例如在参数化组件的库(LPM)中强函数需要配置参数值。为了确定执行形式，对于某些函数也可以有选择地配置参数，但不是被固有参数化，例如老式宏函数。

当用户应用一个存在的参数化函数时，例如一个 LPM 函数，可以设定用过的参数并且在一个实例接实例的基础上配置参数值。在一个 GDF 文件中，用户可以用图形编辑器的 Edit Ports/Parameters 命令(符号菜单)设定一个实例(也就是符号)。在一个 AHDL TDF 文件中，当用户用一个实例说明或者一个内部直接引用建立一个实例时，用户可以申述参数并配置值。

在一个实例接续实例的基础上不一定说明参数值，因为参数值可以从一个层次设计的高层次文件继承下来。编译器检查参数按下面顺序进行：

①作为逻辑函数的实例部分，例如，在一个 TDF 文件中，按照一个实例说明或者一个内部直接引用所建立的实例中，用户可以申述使用的参数并且选择性地配置它们的值。在一个 GDF 文件中，用户可以选择一个符号并且采用 Edit Ports/Parameters 对话框(符号菜单)配置

关于这种实例的参数值。

②如果子设计实例没有完成对参数值的配置,那么逻辑函数的实例的参数值适用于逻辑函数的子设计。

③在全局设计中默认参数值用全局设计参数对话框(分配菜单)具体说明,这些值被存于设计的分配与配置文件(.acf)中。

④在 TDF 或 GDF 的 PARAM 原语的各种参数语句里有选择地列出过默认值,它定义逻辑功能,默认值只适用于在文件里列出过它们的那些文件,它们不能被用于该文件的子设计。

当用户建立一个参数化设计文件时,可以在那个文件内部具体说明用过的参数和选择的默认参数值(如果在别处没有参数值被具体说明它们才被使用)。在一个 GDF 文件里,用户指定在带有 PARAM 原语的当前文件里用过的参数;在一个 TDF 文件的参数语句中,在被详细说明的当前文件内,用户要指定用过的参数。用户一旦建立一个参数化的设计文件,就可以应用建立默认包含文件和建立默认符号命令(文件菜单)来建立默认的 AHDL 函数原型(在包含文件里)和符号(在符号文件里)。它包括在这个文件里用过的各种参数的名(但没有值)。用户可以用符号编辑器的输入参数命令(文件菜单)编辑关于一个符号文件的各种参数和参数值。当这些参数名字和值第一次在 GDF 文件中键入时好像是对这种符号的每个实例的默认。用户一旦在一个 GDF 文件里键入这种符号,那么这些默认参数和值就能在一个实例接着实例的基础上用 Edit Ports/parameters 设定。

MAX+PLUS Ⅱ 允许用户用 Global Project Parameters 命令(分配菜单)配置各种参数全局变量和设计宽度的默认值。改用 Global Project Parameters 时,在 ACF 文件的全局设计参数段用户可以指定设置的默认参数。

下面的准则适用于各种参数:

①对于强函数或宏函数的各自实例都可以分配所有的逻辑值。对一个逻辑函数实例,用一个参数配置一个逻辑选择时,会使全局设计默认综合规则失效。这个规则对于这种实例是用全局设计逻辑综合 Golbal Project Logic Synthesis(分配菜单)做具体说明的。尽管如此,如果对于作为一个参数和作为一个单独逻辑选项两者来说一个实例具有相同的逻辑配置时,逻辑选择设置超出了参数设置的范围。另外,逻辑选择也不能用全局设计参数 Global Project Parameters 配置成全局性的、设计范围默认的参数值。

②用户不能给预先定义的 ALTERA 参数 DEVICE_FAMILY 分配一个值。然而,在对照中用户可以使用这种参数值。这种参数的合法值是 FLEX 10K、FLEX 8000、MAX 9000、MAX 7000E、MAX 7000、MAX 5000、CLASSIC 和 EP330/EP320。

③可以把预先定义的 ALTERA LATENCY 参数分配给一个强函数或宏函数的实例,可是这种参数只适合于那种实例,而不可以用那种实例的子设计去继承。

④如果打开 Show Parameters(选择菜单),那么参数将在图形编辑器或符号编辑器里的一个符号的右上角出现(Show All 也显示或隐藏在当前的 GDF 文件中的所有参数)。

在图形编辑器和符号编辑器的一个参数上双击鼠标 1 键各自打开 Edit Ports/Parameters 或 Enter Parameters 对话框。Show Parameters 或 Show All 显示或隐藏当前 GDF 文件里的全部参数。在打印文件之前通过打开 Show Parameters(或 Show All),用户可以打印一个图形或符号编辑文件。该文件显示各种参数。

4.4 如何使用 AHDL

这部分的主要内容是介绍如何进行 AHDL 设计。所有的样本文件均可在安装 MAX + PLUSⅡ时产生的 \ max2work \ ahdl 目录下找到。如果是在 UNIX 工作站上,该目录为/usr 目录下的一个子目录。

4.4.1 简介

AHDL 是一种用于描述逻辑设计文本的记载语言。它可以使用 MAX + PLUSⅡ文本编辑器或是用户经常使用的文本编辑器来建立 AHDL 文本设计文件(. tdf),可以与其他设计文件合并在一个等级化设计中。

AHDL 包含多种元素,它们被用于逻辑描述语句。本节的内容就是介绍如何使用这些元素及语句。

一、使用数字

在布尔表达式、等式、算术表达式和参数赋值中,数字被用来确定恒定的数值。AHDL 支持十进制、二进制、八进制和十六进制数的任意组合。

下面的程序为一个地址译码器的 tdf 文件。当地址为十六进制数 370 时,产生一个高电压有效的使能信号:

```
 SUBDESIGN decode1
(
        address[15..0]              :INPUT;
        chip_enable                 :OUTPUT;
)
BEGIN
        chip_enable = (address[15..0] == H"0370");
END;
```

上述程序中,十进制数 15 和 0 用来定义地址总线的二进制位数,十六进制数 H0370 用来定义被译码的地址。图 4 - 13 是与 decode1. tdf 功能相同的图形设计文件(. gdf)。

二、使用常量和赋值函数

可以在 AHDL 文件中使用常量描述一个数字或一个字符串,也可以使用赋值函数描述一个算术表达式。这种方法可以给数字、字符串、算术表达式各取一个名字,从而在整个文件中易于识别。例如,数字常量 UPPER_LIMIT 要比数字 130 容易理解。

当相同的数字、字符串、算术表达式在文件中多次出现时,常量和赋值函数非常有用。需要修改时,只修改一个语句。在 AHDL 中,CONSTANT 语句描述常量;DEFINE 语句描述赋值函数。

下面的程序与 decode1. tdf 实现的功能相同,只是前者使用常量 IO_ADDRESS 代替数字 H"0370":

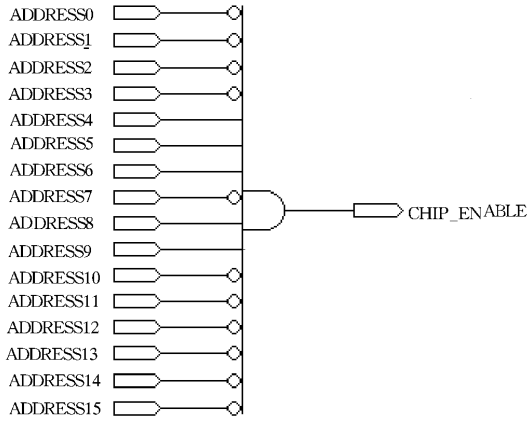


图 4－13　decode1.gdf

```
CONSTANT IO_ADDRESS = H"0370";
SUBDESIGN decode2
(
    a[15..0]  :INPUT;
    ce        :OUTPUT;
)
BEGIN
    ce = (a[15..0] == IO_ADDRESS);
END;
```

可以用算术表达来定义常量和求值函数。常量和求值函数也可以用已定义过的常量、求值函数和参数描述。

在下面的例子中,用一个算术表达式定义常量 foo。**常量** foo_plus_one **是由已经定义过的常量描述的：**

```
CONSTANT foo = 1 + 2DIV3 + LOG2(256);
CONSTANT foo_plus_one = foo + 1;
```

在下例中,求值函数 CEILING_ADD **是在已定义的求值函数** MAX **基础上定义的：**

```
DEFINE MAX(a,b) = (a>b)?a:b;
DEFINE CEILING_ADD(a,b) = MAX(a,b) + 1;
```

编程器计算出算术表达式中的算术运算符,并化简为数值,这些表达式不产生逻辑。下面的程序为 strcmp.tdf **文件,其中定义了常量** FAMILY,**并将它用作为** Assert **语句检查现有器件是否是** FLEX8000 **系列：**

```
PARAMETERS
(
DEVICE_FAMILY        % DEVICE_FAMILY is a predefined Altera parameter %
);
```

```
CONSTANT FAMILY = "FLEX8000";
SUBDESIGN strcmp
(
    a :INPUT;
    b :OUTPUT;
)
BEGIN
    IF (DEVICE_FAMILY == FAMILY) GENERATE
        ASSERT
            REPORT "Detected compilation for FLEX8000 family"
            SEVERITY INFO;
        b = a;
    ELSE GENERATE
        ASSERT
            REPORT "Detected compilation for % family"
                DEVICE_FAMILY
            SEVERITY ERROR;
  b = a;
    END GENERATE;
END;
```

在下面的程序(即 minport.tdf**)中定义了求值函数** MAX**,保证在子设计段为最小端口宽度。**

```
PARAMETERS (WIDTH);
DEFINE MAX(a,b) = (a > b) ?a :b;
SUBDESIGN minport
(
    dataA[MAX(WIDTH,0)..0] :INPUT;
    dataB[MAX(WIDTH,0)..0] :OUTPUT;
)
BEGIN
    dataB[] = dataA[];
END;
```

三、插入一个 AHDL **模板**

设计 AHDL **的最快方法是使用** ALTERA **提供的** AHDL **模板。在** MAX + PLUS Ⅱ **文本编译器中,使用** AHDL **模板命令(模板菜单),可以把** AHDL **模板加入到用户的** TDF **文件中加速设计。**

一个简单的模板在整个 AHDL **文件结构中都是有效的。这个模板被称作"总体结构"。它列出了所有的** AHDL **结构,并且以它们在** TDF **中出现的次序按注释行列出。这些部分的句法和语句不包含在其中,必须用想在文件中使用的正确的** AHDL **语句代替注释行。要将**

AHDL 模板插入到 MAX + PLUS Ⅱ 文本编译器文件的当前插入点，步骤如下：

①将文件以 .tdf 扩展名保存；

②选择 AHDL 模板，显示 AHDL 模板对话框，如图 4 - 14 所示；

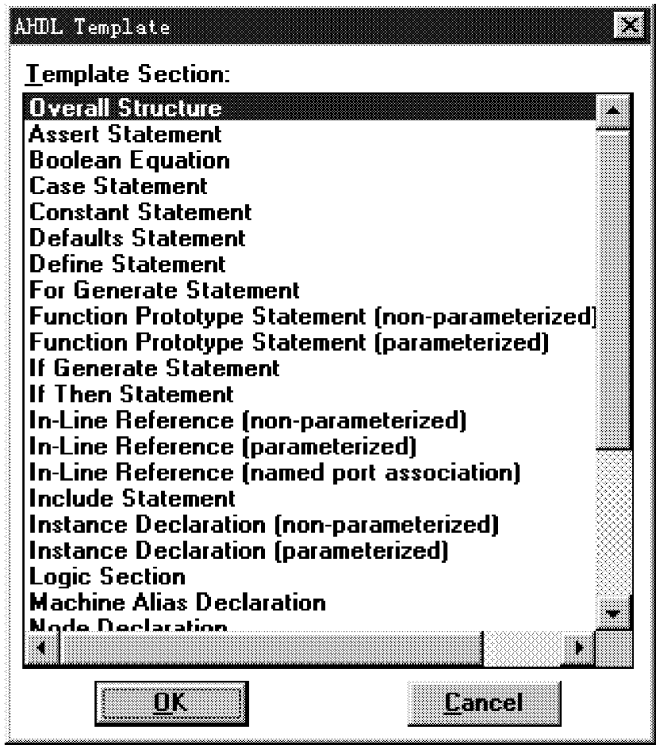


图 4 - 14　AHDL 模板对话框

③在 Template Section 中选中一个模板类型；

④点击 OK 即可。

注意：①快捷方式对这些命令是有效的，在 MAX + PLUS Ⅱ 帮助文件的‘AHDL Template Section’中可以得到详细资料；②所有的 AHDL 模块在 ASCII ahdl.tpl 文件中也是有效的，该文件被自动安装在 \ maxplus2 目录下(或在 UNIX 工作站上的/usr 目录的一个子目录下)。

一旦插入一个模板，必须用用户自己的逻辑取代模块中所有变量。下面的程序表明默认语句模块：

```
DEFAULTS
        _node_name = _constant_value;
END DEFAULTS;
```

每一个 AHDL 关键字都是大写的，每一个变量名以两个下画线(_)开头，帮助用户确认。例如，可以使用句法着色(选项菜单)使关键字和变量易于观察。

MAX + PLUS Ⅱ 为所有 AHDL 结构提供了模板。这些模板按顺序排列，可以被用来代替“Overall Structure”模板中的注释行。

4. AHDL 实例

MAX + PLUS Ⅱ 提供了许多 AHDL 实例，以帮助用户快速进入 AHDL 设计。本节所用的

AHDL 文本设计文件的样本都可以在 \ max2work \ ahdl 目录下找到。UNIX 工作站下,它是在/usr 的一个目录下。用户可以用 MAX + PLUS Ⅱ 文件编译器或任一个标准文件编译器,用另一个文件名来保存它们,并根据用户自己的要求编辑这些文件。

注意:MAX + PLUS Ⅱ AHDL 的帮助还包括许多文件,可以直接复制或粘贴到你的 TDF 文件中。

4.4.2 组合逻辑

当某一时刻的输出只是该时刻输入的函数时,这种逻辑是组合逻辑。在 AHDL 中,组合逻辑可以用布尔表达式、等式、真值表、种类繁多的强函数和宏函数实现。组合逻辑功能的实例包括译码器、乘法器和加法器。

一、布尔表达式及等式的实现

布尔表达式由一系列节点、数字、常量和其他布尔表达式组成,用运算符和比较符分隔,使用多组圆括号将某些项括起来可使表达式更清晰易懂。布尔等式是一个或多个节点的值与一个布尔表达式相等。下面程序的 boole1. tdf 文件用两个简单的布尔表达式代表两个逻辑门:

```
SUBDESIGN boole1
(
    a0, a1, b   :INPUT;
    out1, out2  :OUTPUT;
)
BEGIN
    out1 = a1 & !a0;
    out2 = out1 # b;
END;
```

在程序中,输出 out1 由 a1 和 a0 的逻辑'与'确定,out2 由 out1 和 b 的逻辑'或'确定。由于两等式是同时计算的,书写的先后次序无关紧要。图 4 - 15 是与 boole1. tdf 实现相同功能的 GDF 文件。

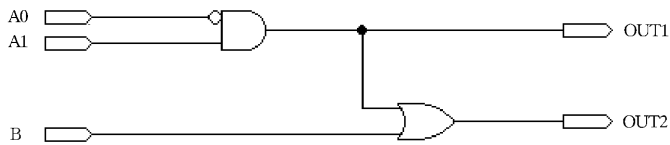


图 4 - 15 boole1. gdf

二、定义节点

在变量段用节点定义语句定义一个节点后,这个节点可以保存一个中介表达式的值。当一个布尔表达式被反复使用时,节点定义非常有用。布尔表达式可以用一个描述性节点名来代替,从而简明易懂。下面程序示出的 boole2. tdf 文件所实现的逻辑和 boole1. tdf 相同,但它只有一个输出。程序如下:

```
SUBDESIGN boole2
```

```
(
    a0, a1, b     :INPUT;
    out           :OUTPUT;
)
VARIABLE
    a_equals_2    :NODE;
BEGIN
    a_equals_2 = a1 & !a0;
    out = a_equals_2 # b;
END;
```

该文件定义了节点 a_equals_2,然后将表达式 a1 & ! a0 的值赋给该节点。当节点在多个表达式中使用时,这种方法可以节省器件资源。

普通节点和三态节点都可以使用,不同之处在于多种任务分别产生不同的结果。结果如下:

①对结点类型的多重定义后,可通过线'与"、线'或'功能将信号连在一起。默认语句中变量的默认值为:一个 VCC 默认产生一个线'与'功能;一个 GND 默认产生一个线'或'功能。

②对于一个 TRI_STATE_NODE 进行多重定义,将使信号连到相同的节点。

③若只有一个变量分配给 TRI_STATE_NODE,它将被当做 NODE 处理。

图 4-16 是与 boole2.tdf 实现相同功能的.gdf 文件。

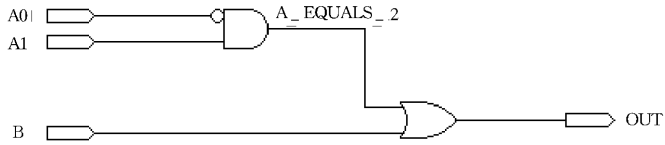


图 4-16 boole2.gdf

3. 组的定义

一个组可以包括多达 256 个成员(或'位"),可以作为一个节点的集合来处理,并且作为一个整体来运行。一个组名可以用单域组名、双域组名或时序组名说明。在布尔等式中,一个组可以被设定与布尔表达式相等,或是与另一个组、一个单一节点、VCC、GND、1、0 相等。每一种情况下,组的值是不同的,Option 语句可以用来定义组的最低位数是 MSB、LSB 或是其他。

注意:当一个组被定义时,[ ]是定义一个完全域的速记法。例如,a[4..1]也可被定义为 a[ ]。同样,b[5..4][3..2]可以用 b[ ][ ]代替。

下面程序描述了对多重组定义的单一布尔表达式:

```
OPTIONS BIT0 = MSB;
CONSTANT MAX_WIDTH = 1+2+3-3-1;  % MAX_WIDTH = 2 %
SUBDESIGN group1
(
    a[1..2], use_exp_in[1+2-2..MAX_WIDTH]      :INPUT;
    d[1..2], use_exp_out[1+2*2-4..MAX_WIDTH]   :OUTPUT;
```

```
    dual_range[5..4][3..2]                    :OUTPUT;
)
BEGIN
    d[] = a[] + B"10";
    use_exp_out[] = use_exp_in[];
    dual_range[][] = VCC;
END;
```

程序中 Option 语句用来定义每组最右边的位是 MSB。a1(十进制)被加到组 a[ ]中。若 a[ ]输入为 00,则样本程序的结果是 d[ ] = =1(十进制)。组 use_exp_in[ ]和组 use_exp_out[ ]说明了常量和算术表达式可以用来界定组的范围。以下实例描述了组的用途:

①当一个组被设定与另一个长度相同的组相等时,右边的每一个数字被分配到左边相应的位置上,如

```
d[2..0] = q[8..6]
```

则位 d2 与位 q8 相连,d1 与 q7、d0 与 q6 相连,又如

```
d[1..0][1..0] = q[10..7]
```

则位 d1_1 与位 q10 相连,d1_0 与 q9、d0_1 与 q8、d0_0 与 q7 相连;

②当一个组与一个单一节点相等时,组中所有位都与该节点相连,例如:

```
d[2..0] = n
```

d2、d1、d0 均与 n 相连;

③当一个组被设定与 VCC 或 GND 相等时,组中每一位都与该值相连,例如:

```
d[2..0] = VCC
```

则 d2、d1、d0 都与 VCC 相关联;

④当一个组被设定与 1 相等时(十进制),只有该组的 LSB 与 VCC 相连,其他位都与 GND 相连,例如:d[2..0] = 1;

⑤只有 d0 与 VCC 相连,十进制数 1 的二进制扩展表达式为 B"001";

⑥当一个组与另一个长度不同的组相等时,等式左边的组中位数必须可以被等式右边组中位数整除,且等式左边的位与等式右边的位按顺序对应,以下等式是合法的:

```
a[4..1] = b[2..1]
```

其中,a4 = b2, a3 = b1, a2 = b2, a1 = b1。

四、实现条件逻辑

不同情况下,条件逻辑的选择要依据逻辑输入而定。If Then 语句和 Case 语句是实现条件逻辑的比较理想的选择。现说明如下:

①If Then 语句计算一个或多个布尔表达式,然后描述表达式不同数值对应的不同情况;

②Case 语句列出表达式每一个值的有效选项并计算表达式,然后根据表达式的值选择一个操作。

注意:不要将使用 If Then 语句和 Case 语句的条件逻辑与 If Generate 语句中的条件产生逻辑相混淆。条件产生逻辑不一定是条件逻辑。

1. If Then 语句逻辑

下面程序示出的 priority.tdf 为优先译码器,可以把有效输入的最高优先权级别转化为一

个数值。它产生一个两位的码，说明 VCC 驱动的输入的最高优先权级别。程序如下：

```
SUBDESIGN priority
(
    low, middle, high      :INPUT;
    highest_level[1..0]  :OUTPUT;
)
BEGIN
    IF high THEN
        highest_level[] = 3;
    ELSIF middle THEN
        highest_level[] = 2;
    ELSIF low THEN
        highest_level[] = 1;
    ELSE
        highest_level[] = 0;
    END IF;
END;
```

本例中，评价 higt、middle 和 Low 三个输入端是为了测定它们是否由 VCC 驱动，然后去执行跟在条件成立的 IF 或 ELSE 从句后的那个等式。也就是说，如果 high 由 VCC 驱动，那么 hignest_level[ ]为 3。

如果 VCC 驱动一个以上的输入，If Then 语句计算输入的优先级，这种优先级是由 IF 和 ELSIF 子句的顺序决定的(第一子句具有最高的优先级)。在 priority. tdf 文件中，优先级从高到低分别为 high、middle、low。当 IF 或 ELSE 子句为真时其后的等式也相应被激活。

若没有输入被 VCC 激活，ELSE 后的等式将成立。图 4－17 为相应的 gdf 文件。

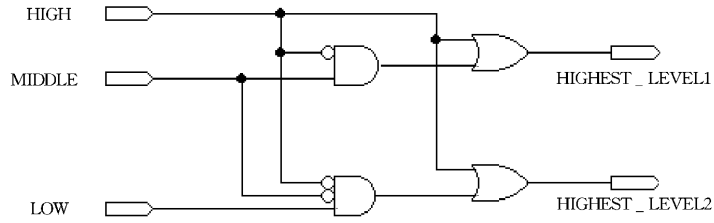


图 4－17　priority. gdf

2. Case 语句逻辑

下面程序示出的 decoder. tdf 文件实现了二－四译码器功能，将两个二进制码输入转换为“one-hot ’码。程序如下：

```
SUBDESIGN decoder
(
        code[1..0]          :INPUT;
```

```
        out[3..0]          :OUTPUT;
)
BEGIN
        CASE code[] IS
                WHEN 0 => out[] = B"0001";
                WHEN 1 => out[] = B"0010";
                WHEN 2 => out[] = B"0100";
                WHEN 3 => out[] = B"1000";
        END CASE;
END;
```

输入组为 code[1..0]**,可取值为** 0、1、2、3。**对于每一个取值,由符号'** => **'表示等式被激活。例如,若** Code[ ]**取** 1,out1 **被设定为二进制数** B'0010"。**因为表达式各不相同,每次只有一个** when **子句被激活。图** 4-18 **示出相应的** gdf **文件。**

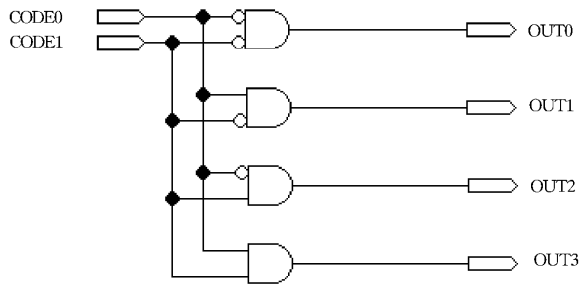


图 4-18　decoder.gdf

3. If Then **语句和** Case **语句的比较**

If Then **语句和** Case **语句是相似的。在默认情况下,可以使用两者中的任一语句,得到的结果是相同的。下例表明用** If Then **语句和** Case **语句可以实现的相同的操作:**

If Then **语句**

```
IF a[] == 0 THEN
        y = c & d;
ELSIF a[] == 1 THEN
        y = e & f;
ELSIF a[] == 2 THEN
        y = g & h;
ELSIF a[] == 3 THEN
        y = i;
ELSE
        y = GND;
```

Case **语句**

```
CASE a[] IS
  WHEN 0 =>
      y = c & d;
  WHEN 1 =>
      y = e & f;
  WHEN 2 =>
      y = g & h;
  WHEN 3 =>
      y = i;
  WHEN OTHERS =>
      y = GND;
```

END IF；　　END CASE；

然而，If Then **语句和** Case **语句之间又存在着许多明显不同：**

①If Then **语句中任何一种布尔表达式均可以被使用。** IF **或** ELSE **子句后的表达式可以互不相关。但是在** Case **语句中，一个简单的布尔表达式都可以与** WHEN **子句中的常量相比较。**

②**使用** ELSIF **子句会导致** MAX + PLUSⅡ **编译器中的逻辑过于繁杂，因为每一个后继的** ELSIF **子句必须检查前面的** IF/ELSIF **子句是否出错。下面的例子表明编译器是如何解释一个** If Then **语句的。如果** a **和** b **是复杂的表达式，那么每一个表达式的逆化将更加复杂。**

If Then **语句：**

```
IF a THEN
        c = d;

ELSIF b THEN
        c = e;

ELSE
        c = f;
END IF;
```

编译器解释语句：

```
IF a THEN
        c = d;
END IF;

IF !a & b THEN
        c = e;
END IF;

IF !a & !b THEN
        c = f;
END IF;
```

五、创建译码器

译码器包含组合逻辑，可以说明输入类型，并把它们转化为输出值。在 AHDL **中，可以使用真值表语句或是** lpm_compare **和** lpm_decode **功能创建译码器。下面程序的** 7segment.tdf **文件可以实现译码器和制订发光二极管逻辑模式。七段显示的发光二极管可以显示十六进制的数字** 0～F**。程序如下：**

```
% -a-                              %
% f|    |b                        %
% -g-                              %
% e|    |c                        %
% -d-                              %
%                                  %
% 0 1 2 3 4 5 6 7 8 9 A b C d E F  %
%                                  %

SUBDESIGN 7segment
(
   i[3..0]              :INPUT;
   a, b, c, d, e, f, g :  OUTPUT;
```

```
)
BEGIN
    TABLE
        i[3..0] => a, b, c, d, e, f, g;

        H"0"    => 1, 1, 1, 1, 1, 1, 0;
        H"1"    => 0, 1, 1, 0, 0, 0, 0;
        H"2"    => 1, 1, 0, 1, 1, 0, 1;
        H"3"    => 1, 1, 1, 1, 0, 0, 1;
        H"4"    => 0, 1, 1, 0, 0, 1, 1;
        H"5"    => 1, 0, 1, 1, 0, 1, 1;
        H"6"    => 1, 0, 1, 1, 1, 1, 1;
        H"7"    => 1, 1, 1, 0, 0, 0, 0;
        H"8"    => 1, 1, 1, 1, 1, 1, 1;
        H"9"    => 1, 1, 1, 1, 0, 1, 1;
        H"A"    => 1, 1, 1, 0, 1, 1, 1;
        H"B"    => 0, 0, 1, 1, 1, 1, 1;
        H"C"    => 1, 0, 0, 1, 1, 1, 0;
        H"D"    => 0, 1, 1, 1, 1, 0, 1;
        H"E"    => 1, 0, 0, 1, 1, 1, 1;
        H"F"    => 1, 0, 0, 0, 1, 1, 1;
    END TABLE;
END;
```

在本例中,输入组 i[3..0] **的** 16 **种可能的输出模式均在真值表中描述。图** 4-19 **为其** GDF **文件。下面的程序中** decode3.tdf **文件是** 16 **位微处理器系统的地址译码器:**

```
SUBDESIGN decode3
(
    addr[15..0], m/io          :INPUT;
    rom, ram, print, sp[2..1]  :OUTPUT;
)
BEGIN
    TABLE
        m/io, addr[15..0]                  => rom, ram,  print, sp[];
        1,    B"00XXXXXXXXXXXXXX"  => 1,   0,    0,     B"00";
        1,    B"100XXXXXXXXXXXXX"  => 0,   1,    0,     B"00";
        0,    B"0000001010101110"  => 0,   0,    1,     B"00";
        0,    B"0000001011011110"  => 0,   0,    0,     B"01";
        0,    B"0000001101110000"  => 0,   0,    0,     B"10";
```

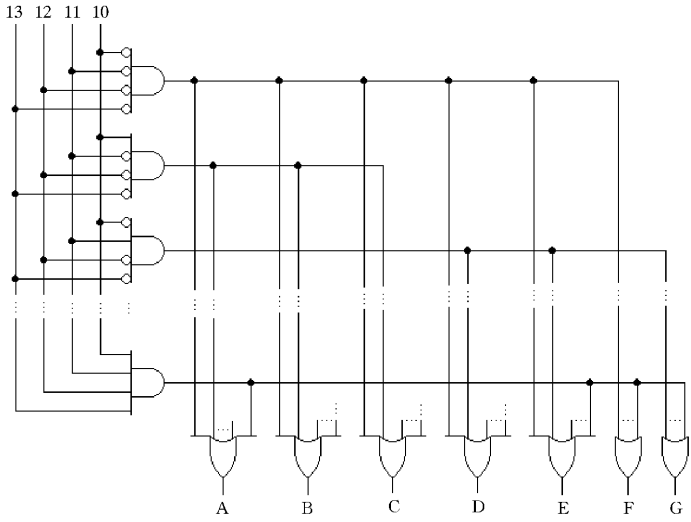


图 4－19　7segment.gdf

```
    END TABLE;
END;
```

在本例中，存在数千个可能的输入模式，不可能将每一种输入都在真值表中定义，所以使用‘X’逻辑说明输出不依赖于对应于 X 位置上的输入。例如，TABLE 语句为第一行，rom 对应于 addr[15..0]以 00 开头的 16,384 个地址输入都是高端有效。因此，用户只需要定义输入模式的公共部分(如 00)，然后用字符‘X’作为输入模式的余下部分。通过这种方式，用户的设计将只占用较少的器件。

注意：当用字符‘X’定义一个位模式的时候，必须保证真值表中该模式没有采用另一个模式的值。假定在某一时刻，AHDL 真值表中只有一个条件为真，那么位模式的重叠会导致不可预料的结果。

下面程序中示出的 decode4.tdf 文件使用 lpm_decode 功能获得了与 decode1.tdf 相同的效果：

```
INCLUDE "lpm_decode.inc";

SUBDESIGN decode4
(
    address[15..0]      :INPUT;
    chip_enable         :OUTPUT;
)
BEGIN
    chip_enable = lpm_decode(.data[] = address[])
```

```
        WITH (LPM_WIDTH = 16, LPM_DECODES = 2^10)
        RETURNS (.eq[H"0370"]);
END;
```

六、使用变量的默认值

当节点或组在文件其他地方没有定义为默认值的时候,可以为节点或组设置默认值。AHDL **允许在一个文件中为节点或组多次赋值。若多重赋值发生矛盾时,默认值可用来解决这类问题。当没有定义默认值时,默认值取** GND。

AHDL **的** Defaults **语句可定义真值表中变量的默认值,也可以定义** If Then **语句和** Case **语句中的默认值。例如,当真值表中没有定义输入条件时,逻辑合成器自动将所有** AHDL **输出与** GND **相连,这时可以使用** defaults **语句的真值表的输出为** VCC。

注意:决不要将变量默认值与端口默认值混淆。端口默认值是在子设计段中定义的。

下面程序所示 default1.tdf **文件用计算输入并选择** 5 **个输入中的一个:**

```
SUBDESIGN default1
(
    i[3..0]              :INPUT;
    ascii_code[7..0]     :OUTPUT;
)
BEGIN
    DEFAULTS
        ascii_code[] = B"00111111"; % ASCII question mark "?" %
    END DEFAULTS;

    TABLE
        i[3..0] => ascii_code[];

        B"1000" => B"01100001"; % "a" %
        B"0100" => B"01100010"; % "b" %
        B"0010" => B"01100011"; % "c" %
        B"0001" => B"01100100"; % "d" %
    END TABLE;
END;
```

当输入模式与真值表左边的模式吻合时,真值表右边为该模式相应的输出。若输入模式与左边每一个模式都不相符,默认值 B"00111111"**为输出,即节点** ascii_code[5..0] **由** VCC **驱动,**ascii_code[7..6] **与** GND **相连。**

下面程序描述当一个节点被多次赋值时产生的冲突以及 AHDL **如何解决这个问题。程序如下:**

```
SUBDESIGN default2
(
    a, b, c                          :INPUT;
```

```
    select_a, select_b, select_c        :INPUT;
    wire_or, wire_and                   :OUTPUT;
)
BEGIN
    DEFAULTS
        wire_or = GND;
        wire_and = VCC;
    END DEFAULTS;

    IF select_a THEN
        wire_or = a;
        wire_and = a;
    END IF;

    IF select_b THEN
        wire_or = b;
        wire_and = b;
    END IF;

    IF select_c THEN
        wire_or = c;
        wire_and = c;
    END IF;
END;
```

本例中，wire_or 依信号 select_a、select_b、select_c 确定 a、b、c 的值。若没有信号为 VCC，则 wire_or 默认值为 GND；若三信号中有两个或三个信号为 VCC，则 wire_or 被设定为对应输入值的逻辑“或”。例如，select_a 和 select_b 为 VCC 时，wire_or 为 a 和 b 的逻辑“或”。

wire_and 信号与 wire_or 相似。不同之处在于，当没有信号输入时默认值为 VCC；当多个信号输入为 VCC 时，为相应的输入值的逻辑“与”。图 4－20 为 default2.tdf 对应的 gdf 文件。

七、实现低端有效逻辑

当低端有效信号为 GND 时，该信号被激活。低有效信号可控制存储器、外围设备及微处理芯片。

如下面程序所示，daisy.tdf 文件实现了一个菊花链判决电路模型。这个模型中，菊花链本级模型向上一级模型提出总线接口请求，并接收本级和下一级模型发出的总线接口请求。发出请求的模型中，优先级最高的模型将获得总线接口。程序如下：

```
SUBDESIGN daisy
(
        /local_request    :INPUT;
        /local_grant      :OUTPUT;
```

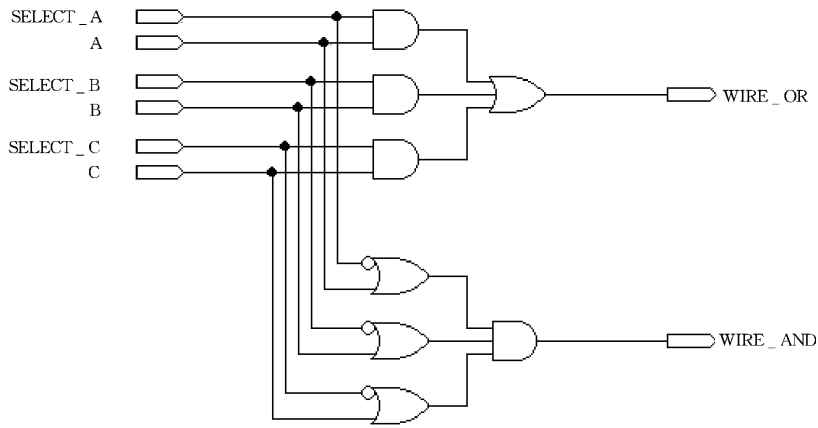


图 4－20　default2. gdf

```
        /request_in     :INPUT;     % from lower priority   %
        /request_out    :OUTPUT;    % to higher priority    %
        /grant_in       :INPUT;     % from higher priority  %
        /grant_out      :OUTPUT;    % to lower priority     %
)
BEGIN
        DEFAULTS
                /local_grant    = VCC;    % active-low output          %
                /request_out    = VCC;    % signals should default     %
                /grant_out      = VCC;    % to VCC                     %
        END DEFAULTS;

IF   /request_in == GND # /local_request == GND THEN
                /request_out = GND;
END IF;

IF   /grant_in == GND THEN
                IF /local_request == GND THEN
                  /local_grant = GND;
                ELSIF /request_in == GND THEN
                  /grant_out = GND;
                END IF;
        END IF;
END;
```

文本中所有的信号都是低有效信号。ALTERA **建议设计者选择一种节点命名方案标明**

低有效信号。例如：以字母‘n’开头或以符号‘/’开头，并保持使用的一致性。符号‘/’不是运算符，而是信号名称的一部分。

If Then 语句用来确定模型是否激活，也就是说模型是否与 GND 相等。若信号激活，If Then 语句后相应的等式也被激活。当信号没有被激活时，Defaults 语句的各个变量被赋值为 VCC。图 4－21 为 daisy.tdf 对应的 gdf 文件。

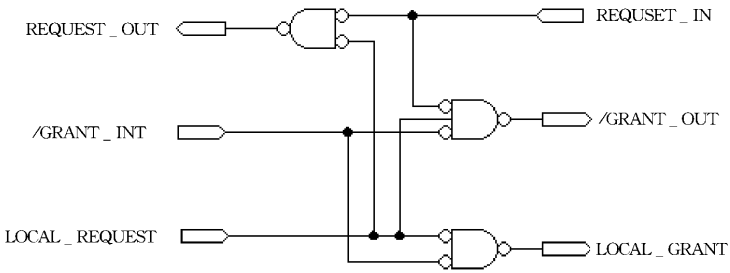


图 4－21　daisy.gdf

八、实现双向管脚设计

MAX＋PLUSⅡ允许将 ALTERA 器件的输入/输出管脚设置为双向管脚。双向管脚可以用 BIDIR 端口定义。该端口与原语 TRI 的输出相连。管脚与‘TRI’原语之间的信号是双向的，可以驱动设计中的其他逻辑。程序如下：

```
SUBDESIGN bus_reg2
(
    clk  ：INPUT；
    oe   ：INPUT；
    io   ：BIDIR；
)
VARIABLE
    dff_out ：NODE；
BEGIN
    dff_out = DFF(io, clk, ,)；
    io = TRI(dff_out, oe)；
END；
```

上面的程序是 bus_reg2.tdf 文件，它实现寄存器功能，可以对三态总线的值进行采样，而且可以将存储的值发回总线。被‘TRI’原语驱动的双向 io 信号作为 D 触发器的输入。符号“,”是用作触发器端口 clrn 和 prn 的占位符。图 4－22 是与 bus_reg2.tdf 文件相对应的图形设计文件。

可以用双向管脚连接低层次 TDF 文件和高层次管脚。子设计端的双向输出端口应与高层次双向管脚相连，低层次 TDF 文件的 FUNCTION 原型应包括 RETURN 子句中的双向管脚。下面程序给出的 bidir1.tdf 文件包含四个与图 4－22 功能相同的实例：

```
FUNCTION bus_reg2 (clk, oe) RETURNS (io)；
```

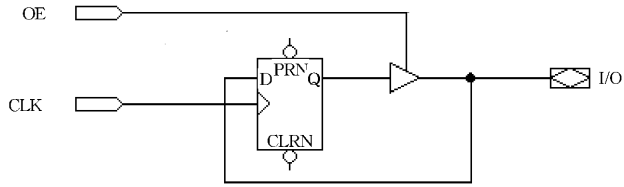


图 4－22　bus_reg2.gdf

```
SUBDESIGN bidir1
(
    clk, oe      :INPUT;
    io[3..0]     :BIDIR;
)
BEGIN
    io0 = bus_reg2(clk, oe);
    io1 = bus_reg2(clk, oe);
    io2 = bus_reg2(clk, oe);
    io3 = bus_reg2(clk, oe);
END;
```

九、实现三态总线

TRI 原语可以驱动 OUTPUT 或 BIDIR 端口，有一个“输出使能”端口使管脚输出为高阻态。在高阻态下该管脚就像没有与电路连接一样。可以将原语 TRI 与 BIDIR 或 OUTPUT 端口相连，成为新的节点类型 TRI_STATE_NODE，来创建三态总线。控制电路必须保证每一时刻至多有一个输出(即不在高阻态)。这种使能输出可以传递低(0)和高(1)逻辑到总线上。

下面程序示出的 tri_bus.tdf 文件中，对 tnode 的多个赋值可将信号连接在一起。必须用 TRI_STATE_NODE 节点类型(不是一般的 NODE 节点类型)实现三态总线。对 NODE 节点赋多个值后，可通过 wire_and 与 wire_or 功能将信号连接在一起。但是，对 TRI_STATE_NODE 的多次赋值把信号连接在相同的节点上。当只有一个变量分配给 TRI_STATE_NODE 节点的时候，它只被当作普通的 NODE 节点类型处理。程序如下：

```
SUBDESIGN tri_bus
(
    in[3..1], oe[3..1]  :INPUT;
    out1                :OUTPUT;
)


VARIABLE
    tnode  :TRI_STATE_NODE;
BEGIN
    tnode = TRI(in1, oe1);
    tnode = TRI(in2, oe2);
```

```
    tnode = TRI(in3, oe3);
    out1 = tnode;
END;
```

4.4.3 时序逻辑

若某一时刻输出为该时刻输入的函数,或者某一时刻输出为该时刻之前某些时刻或所有时刻输入的函数时,称为时序逻辑。所有的时序电路必须包括一个或多个触发器。在 AHDL 中,时序逻辑可以用状态机、寄存器或锁存电路实现。LPM 功能也是有效的。状态机特别适于实现时序逻辑。其他时序逻辑实例还包括计数器和控制器。

一、定义寄存器

寄存器可以存储数据和与时钟信号同步的数据。在变量段通过定义存储器实现其功能。AHDL 提供了许多寄存器原语,AHDL 也支持注册后的 LPM 功能(在逻辑段中,可根据在线提示实现寄存器功能)。

一旦定义了一个寄存器,就可以在 TDF 文件中使用其端口与其它逻辑线连接。一个端口的格式如下:

<实例名>.<端口名>

下面的程序是 bur_reg.tdf、bur_reg.tdf 文件使用寄存器定义语句来创建字节寄存器。输入 load 为高时,d 为输入,时钟上升沿触发的输出 q 被锁存。程序如下:

```
SUBDESIGN bur_reg
(
    clk, load, d[7..0]    :INPUT;
    q[7..0]               :OUTPUT;
)

VARIABLE
    ff[7..0]              :DFFE;

BEGIN

    ff[].clk = clk;
    ff[].ena = load;
    ff[].d = d[];
    q[] = ff[].q;

END;
```

在变量段,寄存器被定义为带使能端的 D 触发器(DFFE)。逻辑段的第一个布尔表达式将文件 bur_reg.tdf 的时钟输入 clk 与 ff[7..0]时钟端口相连;第二个等式将 load 输入与时钟使能端口相连;第三个等式将文件的数据输入 d[7..0]与触发器 ff[7..0]的数据输入端口相连。第四个等式将文件的输出与触发器输出相连接。这四个语句被同时执行。

在变量段中，还可以定义 T 触发器、JK 触发器和 SR 触发器，并在逻辑段使用它们。例如，对 T 触发器(TFF)，将寄存器定义语句 ff[7..0]：DFFE 改为 ff[7..0]：TFF 在第三个等式中将 ff[].d 改为 ff[].t。同理，对于带使能端的 JK 触发器(JKFFE)，可以将寄存器定义语句改为 ff[7..0]：JKFFE 把第三个等式改为两个等式，端口 ff[].j 和端口 ff[].k 与其他信号相连。

注意 如果希望在全局时钟的上升沿装入寄存器，ALTERA 建议使用带使能端输入的触发器 DFFE、TFFE 或 SRFFE 控制寄存器的装载。

如下面程序(即 lpm_reg.tdf)所示，lpm_reg.tdf 文件使用在线提示完成与 bur_reg.tdf 文件功能相同的 lpm_dff 功能。程序如下：

```
INCLUDE "lpm_dff.inc";
SUBDESIGN lpm_reg
(
    clk, load, d[7..0]  : INPUT;
    q[7..0]             : OUTPUT;
)
BEGIN
    q[] = lpm_dff (.clock = clk, .enable = load, .data[] = d[])
        WITH (LPM_WIDTH = 8)
    RETURNS (.q[]);
END;
```

图 4－23 为 GDF 文件，实现的功能与 bur_reg.tdf 文件和 lpm_reg.tdf 文件相同。

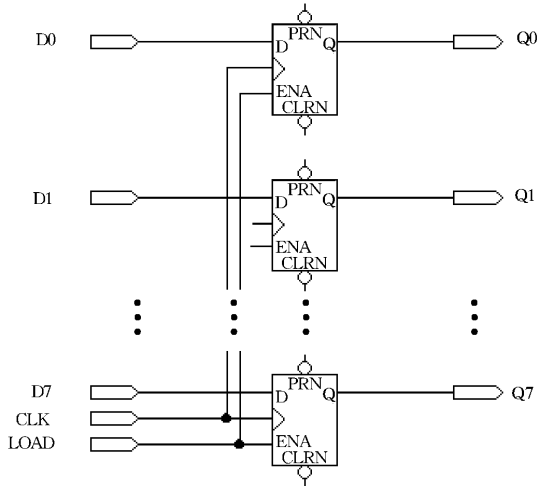


图 4－23　lpm_reg.gdf

二、定义寄存器输出

在变量段寄存器定义部分定义输出端口为触发器，这可以使子设计段的输出为寄存器型。

reg_out.tdf 文件与 bur_reg.tdf 文件实现的功能相同，但是输出为寄存器类型。下面程序是 reg_out.tdf 文件：

```
SUBDESIGN reg_out
(
    clk, load, d[7..0]  :INPUT;
    q[7..0]             :OUTPUT;
)
VARIABLE
    q[7..0]             :DFFE; % outputs also declared as output %
BEGIN
    q[].clk = clk;
    q[].ena = load;
    q[] = d[];
END;
```

给逻辑段中的寄存器输出端赋值后，这个值通过 d 端口输入到寄存器。直到时钟上升沿到来，寄存器的输出才会改变。为了定义时钟输入给寄存器，可使用'＜寄存器名＞.clk'在逻辑段为寄存器定义时钟输入。使用 GLOBAL 原语或使用 Global Project Logic Synthesis 对话框的 Automatic Global Clock 选项均可实现全局时钟。图 reg_out.tdf 的文本文件中，逻辑段中定义的每一个带使能的 D 触发器与其输出端同名。所以，不使用触发器的 q 端口，就可以指定 q 为触发器的输出。

注意 在高层次 TDF 文件中，输出端口与输出管脚是同步的，当定义输出端口与寄存器同名时，任何逻辑选项的分配都是针对管脚的，而不是寄存器。这种同名可以通过制订一个寄存器逻辑选项来避免，如 I/O Cell Register。所以，如果使用特定寄存器逻辑选项，必须为寄存器和端口起不同的名字。可能会有多种方法实现所需的功能，例如，可以用 Global Project Logic Synthesis 对话框的 Automatic I/O Cell Register 选项自动实现 I/O 单元内的寄存器，不论定义的输出端口和寄存器是否同名。

三、创建计数器

计数器使用时序逻辑计算时钟脉冲个数。一些计数器可以前向和后向计数、装载数据及清零。计数器可以用 D 触发器(DFF 和 DFFE)、If Then 语句或 lpm_counter 函数定义。下面程序是 ahdlcnt.tdf 文件，它实现了可清零、可装载的 16 位计数器。程序如下：

```
SUBDESIGN ahdlcnt
(
    clk, load, ena, clr, d[15..0]  :  INPUT;
    q[15..0]                       :OUTPUT;
)
VARIABLE
    count[15..0]                   :DFF;
BEGIN
    count[].clk = clk;
```

```
    count[].clrn = !clr;
    IF load THEN
        count[].d = d[];
    ELSIF ena THEN
        count[].d = count[].q + 1;
    ELSE
        count[].d = count[].q;
    END IF;
    q[] = count[];
END;
```

文件中,16 位 D 触发器在变量段被定义,并分配名字为 count0 到 count15。If Then 语句保证在时钟的上升沿将数据装入触发器(例如:若 load 为 VCC,d[ ]的值赋给触发器)。下面程序是 lpm_cnt.tdf 文件,它使用 lpm_counter 实现与 ahdlcnt.tdf 相同的功能。程序如下:

```
INCLUDE "lpm_counter.inc";
SUBDESIGN lpm_cnt
(
    clk, load, ena, clr, d[15..0] :INPUT;
    q[15..0]                      :OUTPUT;
)
VARIABLE
    my_cntr:lpm_counter WITH (LPM_WIDTH=16);

BEGIN
    my_cntr.clock  = clk;
    my_cntr.aload  = load;
    my_cntr.cnt_en = ena;
    my_cntr.aclr   = clr;
    my_cntr.data[] = d[];
    q[] = my_cntr.q[];
END;
```

图 4-24 所示为 GDF 文件,实现的功能与 lpm_cnt.tdf 和 ahdlcnt.tdf 相同。

4.4.4 状态机

同真值表和布尔表达式一样,在 AHDL 中实现状态机是很容易的。AHDL 可以设置状态位和状态值,余下的工作由 MAX+PLUSⅡ编译器完成。编译器使用高级属性的启发式算法实现自动设置状态,减少实现状态机占用的资源。

只需要画一个状态图表和写一个状态表,编译器就可以自动实现以下功能:

①设置位,选择 T 或 D 触发器(TFF 或 DFF);

②设置状态值;

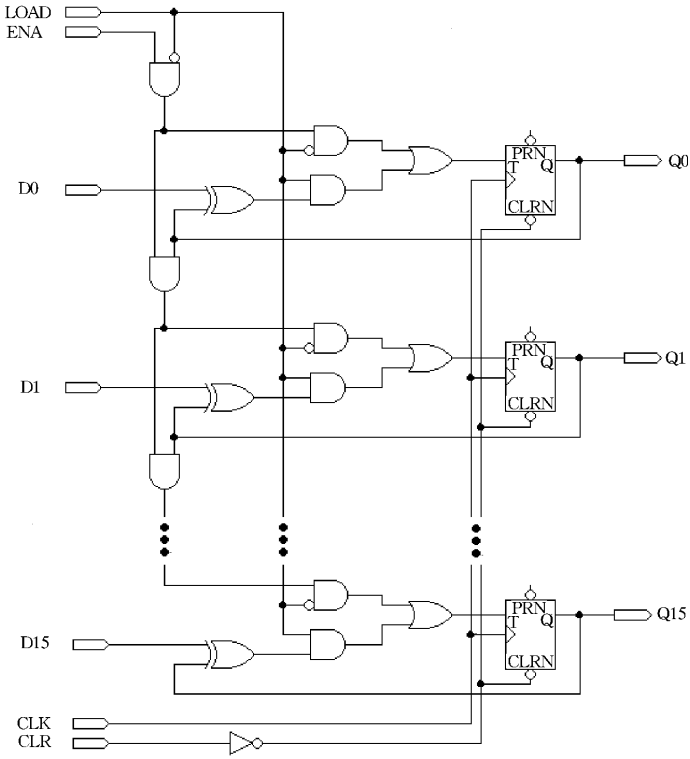


图 4－24　count.gdf

③应用复杂的逻辑综合技术得到激励等式。

在 AHDL 中定义一个状态机，必须在 TDF 文件中包含以下条目：

①状态机定义(变量段)；

②布尔控制等式(逻辑段)；

③真值表语句或 Case 语句中的状态转换(逻辑段)。

通过定义一个输入或输出信号作为子设计段的状态机端口，在 TDF 文件和其它设计文件之间输入或输出 AHDL 状态机。

一、实现状态机

通过定义状态机名及其状态和变量段中状态机定义的状态位，就可以完成对状态机的创建。下面程序是 simple.tdf 文件，它实现的功能与 D 触发器相同。

```
SUBDESIGN simple
(
    clk, reset, d    :INPUT;
    q                :OUTPUT;
)
```

```
VARIABLE
    ss:MACHINE WITH STATES (s0, s1);
BEGIN
    ss.clk = clk;
    ss.reset = reset;
    CASE ss IS
        WHEN s0 =>
            q = GND;
            IF d THEN
                ss = s1;
            END IF;
        WHEN s1 =>
            q = VCC;
            IF  !d THEN
                ss = s0;
            END IF;
    END CASE;
END;
```

simple.tdf 文件变量段的状态机定义语句中,状态机被命名为 ss。两个状态分别是 s0 和 s1,没有定义状态位。

状态机的转移定义了状态机变为新状态的条件。必须在简单行为结构内有条件地分配状态来定义状态机的转移。可以使用 Case 语句或真值表语句实现转移,例如,在 simple.tdf 中,每个状态的转移都在 Case 语句的 When 子句中定义。

用 If Then 或 Case 语句都可以定义一个状态的输出值。Case 语句中,该任务在 WHEN 子句中完成。例如:simple.tdf 中,输出 q 被设置为 GND(状态机为 s0 状态时)。当状态机状态为 s1 时,输出 q 为 VCC。图 4-25 为相应的 GDF 文件。

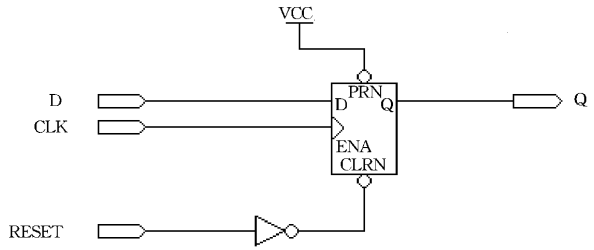


图 4-25 simple.gdf

二、设置时钟、置位和使能信号

时钟信号、置位信号和时钟使能信号在状态机内控制状态寄存器的触发器。在逻辑段中这些信号用布尔控制等式定义。下面程序是 simple1.tdf 文件,其中状态机时钟由输入 clk 驱动。状态机的异步置位信号由 reset 驱动,reset 是高电平有效。设计文件中,子设计段的 ena

输入的定义和布尔等式 ss. ena = ena 与时钟使能信号相连。程序如下：

```
SUBDESIGN simple
(
    clk, reset, ena, d      :INPUT;
    q                       :OUTPUT;
)
VARIABLE
    ss:MACHINE WITH STATES (s0, s1);
BEGIN
    ss.clk = clk;
    ss.reset = reset;
    ss.ena = ena;
    CASE ss IS
        WHEN s0 =>
            q = GND;
            IF d THEN
                ss = s1;
            END IF;
        WHEN s1 =>
            q = VCC;
            IF !d THEN
                ss = s0;
            END IF;
    END CASE;
END;
```

三、设置状态机的位和数值

一个状态位是一个触发器的输出，状态机使用这个触发器存储一位状态机的值。大多数情况下，应该让 MAX + PLUS Ⅱ 编译器分配状态位和状态值使其对逻辑资源的需求最小。在优化器件的利用和性能时，逻辑综合器自动减少所需的状态位数，但是，当一些状态机使用的位数比最小位数多时，可以工作得更快。如果想让状态机的输出状态位更加清晰易辨，可以在状态机定义部分定义状态位和状态值。

注意：Global Project Logic Synthesis 对话框包含一个 One_hot State Machine Encoding 选项，可自动实现 One_hot 编码设计。MAX + PLUS Ⅱ 编译器可用 FLEX8000 和 FLEX10K 器件实现 One_hot 状态机编码，无论 One_hot State Machine Encoding 选项是否被选中。如果在自动 One_hot 编码的同时，还要额外定义一些状态位，那么设计的逻辑实现效率将会降低。下面程序是 stepper. tdf 文件，它实现了步进电机控制器的功能：

```
SUBDESIGN stepper
(
    clk, reset          :INPUT;
```

```
    ccw, cw          :INPUT;
    phase[3..0]      :OUTPUT;
)
VARIABLE
    ss:MACHINE OF BITS (phase[3..0])
        WITH STATES  (
            s0 = B"0001",
            s1 = B"0010",
            s2 = B"0100",
            s3 = B"1000");
BEGIN
    ss.clk   = clk;
    ss.reset = reset;
    TABLE
        ss,   ccw,   cw    =>   ss;
        s0,   1,     x     =>   s3;
        s0,   x,     1     =>   s1;
        s1,   1,     x     =>   s0;
        s1,   x,     1     =>   s2;
        s2,   1,     x     =>   s1;
        s2,   x,     1     =>   s3;
        s3,   1,     x     =>   s2;
        s3,   x,     1     =>   s0;
    END TABLE;
END;
```

本例中,输出 phase[3..0]**在子设计段定义。在状态机定义段,**hase[3..0]**被定义为状态机** ss **的位。需要注意的是,**ccw **和** cw **不能在表中同时为** 1。AHDL **规定,某一时刻只能有一个条件为真,因此位模式的重叠会导致无法预料的结果。**

四、同步输出的状态机

如果状态机的输出只依赖于该状态机的状态,可以用状态机定义中的 WITH STATE **子句来定义状态机输出。分配状态值使机器不易出错。在一些情况下,这个逻辑可以使用较少的逻辑单元。图** 4-26 **为四态摩尔(**Moore**)状态机图。在** Moore **状态机中,状态机的当前状态只依赖于前一状态,当前的输出只依赖于当前状态。下面的程序是实现图** 4-26 **四态摩尔(**Moore**)状态机图的文本文件:**

```
SUBDESIGN moore1
(
    clk     :INPUT;
    reset   :INPUT;
    y       :INPUT;
```

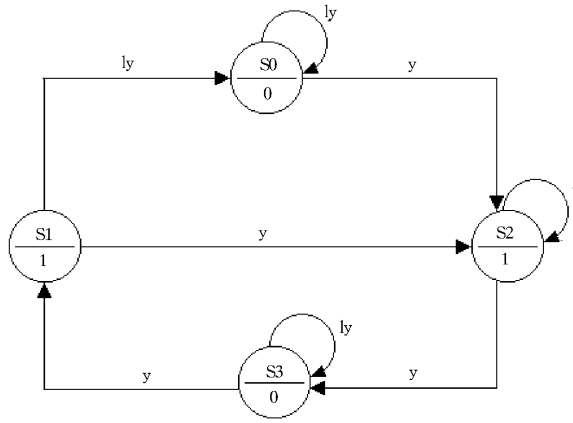


图 4－26　Moore **状态机图**

```
    z       :OUTPUT;
)
  VARIABLE
                        %    current     current  %
                        %    state       output   %
    ss:MACHINE OF BITS (z)
        WITH STATES (s0      =    0,
                     s1      =    1,
                     s2      =    1,
                     s3      =    0);
  BEGIN
    ss.clk    = clk;
    ss.reset  = reset;
    TABLE
    %   current   current       next   %
    %   state     input         state  %
        ss,       y        =>   ss;
        s0,       0        =>   s0;
        s0,       1        =>   s2;
        s1,       0        =>   s0;
        s1,       1        =>   s2;
        s2,       0        =>   s2;
        s2,       1        =>   s3;
        s3,       0        =>   s3;
```

```
        s3,      1       =>    s1;
    END TABLE;
END;
```

本例使用状态机定义段来定义状态机的状态。状态转移定义在下一个状态表中,用真值表语句实现。本例中,状态机 ss **有四个状态,但只有一个状态位(**z**)。**MAX + PLUS Ⅱ **编译器自动添加另一位,产生相应的综合变量,实现四态机。这个状态机需要至少两个位。图** 4 - 27 **为相应的** GDF **文件。**

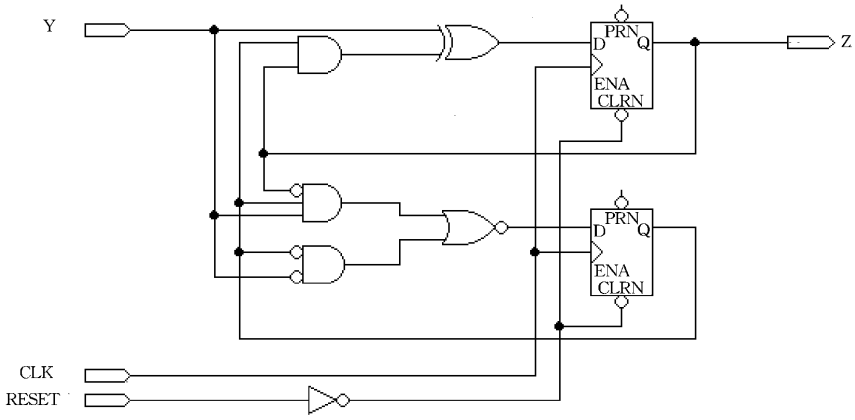


图 4 - 27　moore1.gdf

当状态值被当做输出时,可以使用较少的逻辑单元进行设计,但这些逻辑单元可能需要更多的逻辑驱动触发器输入。编译器的逻辑综合模块也许不能充分简化状态机。设计同步输出状态机的另一种方法是省略状态值的分配,明确定义触发器的输出。下面程序是 moore2.tdf **文件,它描述了这种方法。程序如下:**

```
SUBDESIGN moore2
(
    clk     :INPUT;
    reset   :INPUT;
    y       :INPUT;
    z       :OUTPUT;
)
VARIABLE
    ss:MACHINE WITH STATES (s0, s1, s2, s3);
    zd:NODE;
BEGIN
    ss.clk   = clk;
    ss.reset = reset;
    z = DFF(zd, clk, VCC, VCC);
```

```
TABLE
%   current   current      next      next    %
%   state     input        state     output  %
    ss,       y       =>   ss,       zd;
    s0,       0       =>   s0,       0;
    s0,       1       =>   s2,       1;
    s1,       0       =>   s0,       0;
    s1,       1       =>   s2,       1;
    s2,       0       =>   s2,       1;
    s2,       1       =>   s3,       0;
    s3,       0       =>   s3,       0;
    s3,       1       =>   s1,       1;
END TABLE;
END;
```

在状态机定义段,本例定义了‘下一个输出’栏在‘下一状态’栏之后。这种方法使用一个 D **触发器,使输出与时钟同步。**

五、带异步输出的状态机

AHDL **支持带异步输出的状态机的实现。对这种类型的状态机,输出随输入的改变而改变,与时钟变化无关。图** 4－28 **为四态** Merly **状态机图。在** Merly **状态机中,输出是输入和当前状态的函数。**

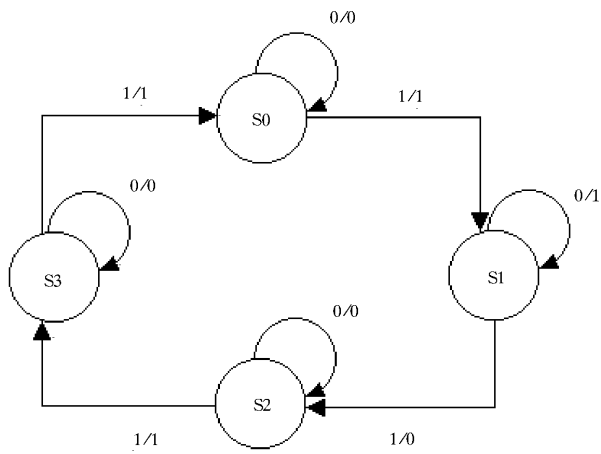


图 4－28　Merly **状态机图**

下面的程序为带异步输出的四态 Merly **状态机的文本设计文件** Merly. tdf**：**

```
SUBDESIGN mealy
(
```

```
    clk      :INPUT;
    reset    :INPUT;
    y        :INPUT;
    z        :OUTPUT;
)
VARIABLE
    ss:MACHINE WITH STATES (s0, s1, s2, s3);
BEGIN
    ss.clk = clk;
    ss.reset = reset;
    TABLE
    %  current   current      current   next   %
    %  state     input        output    state  %
       ss,       y        => z,        ss;
       s0,       0        => 0,        s0;
       s0,       1        => 1,        s1;
       s1,       0        => 1,        s1;
       s1,       1        => 0,        s2;
       s2,       0        => 0,        s2;
       s2,       1        => 1,        s3;
       s3,       0        => 0,        s3;
       s3,       1        => 1,        s0;
    END TABLE;
END;
```

图 4－29 示出一个与 Merly.tdf 等效的 GDF 文件。

六、修复不合法状态

由 MAX＋PLUS Ⅱ编译器使状态机生成的逻辑可以与用户在 TDF 文件中指定的逻辑一样运行。但是，状态机的设计明确定义状态位并不使用 one_hot 编码，经常允许状态位值不被设置成有效状态。这些未分配的无效状态位值被称做非法状态。一个进入非法状态的设计（例如，同建立或保持时间相违背后时），会导致输出错误。尽管 ALTERA 建议状态机输入要符合所有建立和保持时间的要求，设计者可以用 Case 语句的方式强行把一个非法状态转变成一个合法状态的方式，从而完成对非法状态的修复。

注意：Global Project Logic Synthesis 对话框包含一个 One_hot State Machine Encoding 选项，可自动实现 One_hot 编码设计，自动分配所有的状态位给有效状态。MAX＋PLUS Ⅱ编译器可用 FLEX8000 和 FLEX10K 器件来实现 One_hot 状态机编码。无论 One_hot State Machine Encoding 选项是否被选中，如果在自动 One_hot 编码的同时还要额外定义一些状态位，设计的逻辑实现效率将会降低。

为了修复非法状态，必须在状态机中命名所有的非法状态。Case 语句中 WHEN OTHERS 子句可以完成每一个从非法状态到已知状态的转变，但只适用于已经定义过但没有在

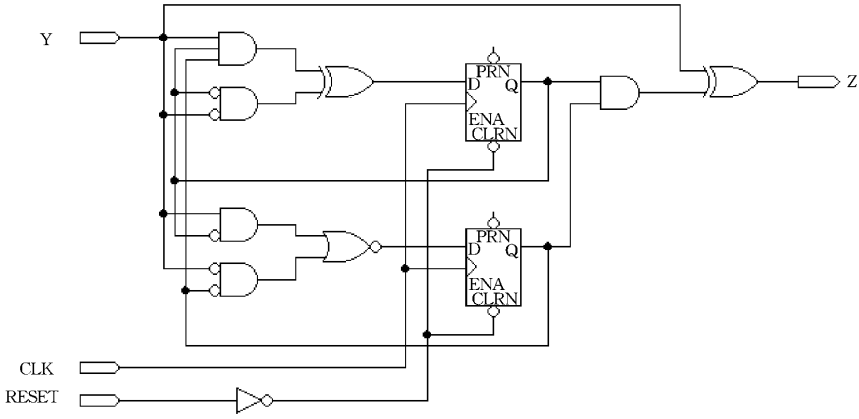


图 4－29　Merly. gdf

WHEN子句中提及的那些状态。如果所有的非法状态均在状态机定义段被说明,那么WHEN OTHERS子句才可以进行所需的转换。

对于一个 N 位状态机,存在着 2^N 个可能的状态。若在状态机内定义 N 位,应该继续添加虚拟状态名直到状态数达到 2 的乘幂。recover. tdf 文件(如下面的程序所示)包含一个可修复非法状态的状态机。程序如下:

```
SUBDESIGN recover
(
    clk    :INPUT;
    go     :INPUT;
    ok     :OUTPUT;
)
VARIABLE
    sequence :MACHINE
             OF BITS (q[2..0])
             WITH STATES (
                 idle,
                 one,
                 two,
                 three,
                 four,
                 illegal1,
                 illegal2,
                 illegal3);

BEGIN
```

```
    sequence.clk = clk;
    CASE sequence IS
        WHEN idle =>
            IF go THEN
                sequence = one;
            END IF;
        WHEN one =>
            sequence = two;
        WHEN two =>
            sequence = three;
        WHEN three =>
            sequence = four;
        WHEN OTHERS =>
            sequence = idle;
    END CASE;
    ok = (sequence == four);
END;
```

上例包含三位,即 q2、q1、q0。**因此共** 2^3 **个状态,其中只有** 5 **个状态被定义,所以加上** 3 **个虚拟状态名,创建总共** 8 **个状态。**

4.4.5 实现层次化设计

AHDL **的** TDF **文件在层次化设计中可与其他设计文件混合,其中的低层次文件可以是** ALTERA **提供的** megafunction、macrofunction **或用户定义的功能。**

一、使用 ALTERA **提供的非参数化函数**

MAX + PLUS Ⅱ **包含原语库和老式宏函数,而且一些** megafunction **也不含固有参数。所有的** MAX + PLUS Ⅱ **逻辑功能都可用来创建层次化逻辑设计。** Megafunction **和** macrofunction **被自动安装在** \ maxplus2 \ max2lib **子目录下,原语逻辑在** AHDL **中建立。如果是在** UNIX **工作站上,该目录为**/usr **目录下的一个子目录。**

AHDL **无参数功能的使用有两种方法:**

①**为功能定义变量,即定义一个实例名(在变量段的实例定义中),使用功能实例的端口(在逻辑段中);**

②**在** TDF **的逻辑段使用内部直接引用逻辑功能。**

Megafunction **和** macrofunction **的输入和输出必须用函数类型语句定义(原语不需要函数类型)。** MAX + PLUS Ⅱ **提供的** Include **文件(**. inc**)包含** MAX + PLUS Ⅱ **所有** megafunction **和** macrofunction **的函数类型,并分别放在** \ maxplus2 \ max2lib \ mega_lpm **和** \ maxplus2 \ max2inc **目录下。使用** Include **语句可以将** Include **文件的内容引入到** TDF **文件中,去定义** MAX + PLUS Ⅱ **的** megafunction **和** macrofunction **函数类型。下面程序(即** macrol. tdf **文件)为四位计数器与一个四位** 16 **线译码器相连。这些** macrofunction **在变量段的实例定义中被调用。程序如下:**

```
INCLUDE "4count";
INCLUDE "16dmux";
SUBDESIGN macro1
(
    clk             :INPUT;
    out[15..0]      :OUTPUT;
)
VARIABLE
    counter         :4count;
    decoder         :16dmux;
BEGIN
    counter.clk = clk;
    counter.dnup = GND;
    decoder.(d,c,b,a) = counter.(qd,qc,qb,qa);
    out[15..0] = decoder.q[15..0];
END;
```

这个文件使用 Include **语句，提供了** macrofunction，**即** 4count **和** 16mux。**在变量段，变量** counter **被定义为** 4count **的一个实例，变量** decoder **被定义为一个** 16mux **功能的实例。两个函数的输入端口形式为＜实例名＞.＜端口名＞，在逻辑段布尔等式的左边定义，输出端口定义在右边。下面程序中的** macro2.tdf **文件与** macro1.tdf **文件功能相同，但使用了内部直接引用和节点** q[3..0] **创建两个功能实例。程序如下：**

```
INCLUDE "4count";
INCLUDE "16dmux";
SUBDESIGN macro2
(
    clk             :INPUT;
    out[15..0]      :OUTPUT;
)
VARIABLE
    q[3..0]         :NODE;
BEGIN
    (q[3..0],) = 4count (clk,,,,,GND,,,,);

% equivalent in-line ref. with named port association          %
%   (q[3..0],) = 4count (.clk=clk,.dnup=GND);                  %
% equivalent in-line ref. with named port association          %
% and RETURNS clause specifying which outputs are used        %
%   q[3..0] = 4count (.clk=clk,.dnup=GND)                      %
%                    RETURNS (qd,qc,qb,qa);                    %
```

```
    out[15..0]   = 16dmux (.(d, c, b, a) = q[3..0]);

% equivalent in-line ref. with positional port association      %
% out[15..0]   = 16dmux (q[3..0]);                              %

END;
```

两个 macrofunction 的函数原型存储在 Include 文件 4count.inc 和 16mux.inc 中如下所示：

```
FUNCTION 4count (clk,clrn,setn,ldn,cin,dnup,d,c,b,a)
        RETURNS (qd,qc,qb,qa,cout);
FUNCTION 16mux (d,c,b,a)
        RETURNS (q[15..0])
```

在逻辑段中，对 4count 和 16mux 的内部直接引用分别出现在布尔等式第一和第二句中。4count 内部直接引用使位置端口关联，而 16mux 的内部直接引用是通过命名端口关联来实现的。两个 macrofunction 的输入端口被定义在内部直接引用的右端，输出端定义在左端。

不同的端口关联可以使用相同的内部直接引用。在内部直接引用中，右边端口可以用位置或命名端口来关联，等号左边的端口总是使用位置端口关联。当使用位置端口关联时，端口的顺序是非常重要的，因为端口的顺序（功能定义中）与逻辑段中端口的顺序是一一对应的。4count 的内部直接引用中‘，’被用做占位符，用来说明没有明确连接的端口。

在函数类型定义的基础上，RETURN 子句可随意选择内部直接引用。RETURN 子句可列出实例中所用的函数输出的子集（subset）。在 macro2.tdf 中，第二句注释描述了另一个 4count 的内部直接引用，省略了 RETURN 子句的 cout 输出。所以，只有输出 q[3..0] 是内部直接引用，而且不必使用‘，’作为 cout 的占位符。

注意 :原语是一种老式的 macrofunction，对未连接的输入取默认值。图 4－30 为 macro1.tdf 和 macro2.tdf 的 GDF 文件。

二、使用 ALTERA 提供的参数化函数

MAX＋PLUS Ⅱ内部有参数化的含有 LPM 功能的 megafunction。例如，用参数定义端口的宽度或者定义是否用 RAM 实现异步或同步的存储器。参数化函数可以包含其它子设计段。这些子设计段可以是参数化的或非参数化的。参数也可以用在一些老式的不带固有参数的 macrofunction（原语是不能参数化的）中。所有的 MAX＋PLUS Ⅱ逻辑功能都可以用来实现层次化设计。megafunction 和 macrofunction 自动安装在 \ maxplus2 \ maxlib 目录下，原语逻辑是在 AHDL 中构建的。

用内部直接引用逻辑函数或实例说明后，按照类似非参数化函数方法使参数化函数具体化。正如在‘使用 ALTERA 提供的非参数化函数’所描述的，只是附加了一些步骤。步骤如下：

①逻辑功能实例必须包括一个 WITH 子句。该子句是建立在函数类型定义的 WITH 子句基础上的。通过使用 WITH 子句，可以为一个实例配置参数。但是，对于函数中所需的所有参数，必须在设计的某个部分为参数赋值。如果实例本身不能包含所有参数所需的值，编译器会在参数表中寻找这些值。

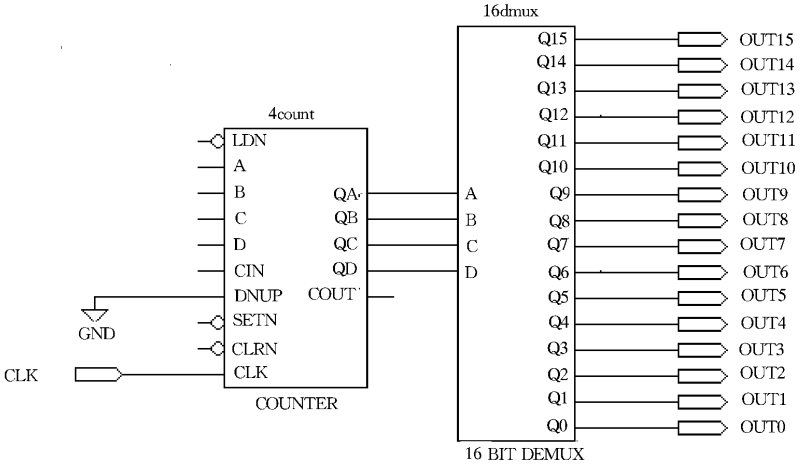


图 4－30　macro. gdf

②**因为参数化的函数不必为没有连接的输入端设定默认值，所以它必须保证所有端口是连接的。相应的，原语和老式的** macrofunction **总是要求对未连接的输入设定默认值。**

可以用一个函数类型语句定义函数的输入、输出和参数。 MAX＋PLUS Ⅱ **提供了** Include **文件。这个文件在包含在** \ maxplus2 \ max2lib \ mega_lpm **和** \ maxplus2 \ max2inc **目录下关于所有** MAX＋PLUS Ⅱ megafunction **和** macrofunction **的函数类型。使用** Include **语句，可以将** Include **文件的内容引入到** TDF **文件中，定义** MAX＋PLUS Ⅱ megafunction **和** macrofunction **的函数类型。下面程序（即** lpm_add1. tdf **文件）对参数化的** lpm_add_sub megafunction **采用内部逻辑函数引用实现了一个** 8 **位加法器：**

```
INCLUDE "lpm_add_sub.inc";
SUBDESIGN lpm_add1
(
    a[8..1], b[8..1]          :INPUT;
    c[8..1]                   :OUTPUT;
    carry_out                 :OUTPUT;
)
BEGIN
% Megafunction instance with positional port association %
--  (c[], carry_out, ) = lpm_add_sub(GND, a[], b[], GND)
--                              WITH (LPM_WIDTH=8,
--                                    LPM_REPRESENTATION="unsigned");
% Equivalent instance with named port association %
--  (c[], carry_out, ) = lpm_add_sub(.dataa[]=a[], .datab[]=b[],
--                                   .cin=GND, .add_sub=GND)
```

```
--                          WITH (LPM_WIDTH=8,
--                                LPM_REPRESENTATION="unsigned");
END;
```

在 Include 文件 lpm_add_sub.inc 中,存储的 lpm_add_sub 函数原型如下:

```
FUNCTION lpm_add_sub(cin, dataa[LPM_WIDTH-1..0], datab[LPM_WIDTH-1..
    0], add_sub)
    WITH (LPM_WIDTH, LPM_REPRESENTATION, LPM_DIRECTION, ADDER-
    TYPE,ONE_INPUT_IS_CONSTANT)
    RETURNS (result[LPM_WIDTH-1..0], cout, overflow);
```

只有 LPM_WIDTH 参数需要赋值。在 lpm_add1.tdf 文件中,lpm_add_sub 函数的实例只定义 LPM_WIDTH 和 LPM_REPRESENTATION 的参数值。下面的程序是 lpm_add1.tdf 文件,它与 lpm_add1.tdf 文件功能相同,但只是用实例定义来完成 8 位加法器。程序如下:

```
INCLUDE "lpm_add_sub.inc";
SUBDESIGN lpm_add2
(
    a[8..1], b[8..1]    :INPUT;
    c[8..1]             :OUTPUT;
    carry_out           :OUTPUT;
)
VARIABLE
    8bitadder :lpm_add_sub WITH (LPM_WIDTH=8,
                    LPM_REPRESENTATION="unsigned");
BEGIN
    8bitadder.cin = GND
    8bitadder.dataa[] = a[]
    8bitadder.datab[] = b[]
    8bitadder.add_sub = GND
    c[] = 8bitadder.result[]
    carry_out = 8bitadder.cout
END;
```

三、使用自定义的 megafunction 和 macrofunction

在 AHDL 的 TDF 文件中,可以很容易地创建和应用自定义的 megafunction 和 macrofunction。在一个设计文件中一旦定义了一个自定义函数的逻辑,在其他 TDF 文件和其他类型设计文件中为应用这种函数需要很少的步骤。

在另外的设计文件里,为配备一个要应用的基于强函数或宏函数的自制 AHDL,要做下面几项工作:

①编译并有选择地进行仿真,保证设计文件函数的正确性;

②若在多种设计中使用这个函数,应该将包含设计文件的目录设计成用户库(使用 Options 菜单中的 User Libraries)或者将文件复制到一个已经存在的用户库目录中,另外,可以将

文件的备份保存在使用自定义函数的设计目录下；

③在打开的文本编译窗口中，创建 Include 文件和代表当前文件的符号。

创建 Include 文件和代表当前文件的符号的方法如下：

①选择 Creat Default Include File(File 菜单中)创建一个 Include 文件，可以被用在高层次 TDF 文件。使用 Include 语句，可以将 Include 文件的内容引入 TDF 文件，自定义 megafunction 或 macrofunction 函数类型。

②选择 Creat Default Symbol(File 菜单中)创建 GDF 文件中可使用的符号。

在其他设计文件中，一旦建立一个使用函数，可以创建新的 TDF 文件，用实例定义或内部直接引用来插入函数实例。自定义的使用与 ALTERA 提供的函数的使用方法相同。

四、输入和输出状态机

通过在子设计段定义输入、输出为 MACHINE INPUT 或 MACHINE OUTPUT，可以在 TDF 文件和其他设计文件之间输入和输出状态机。函数类型表示的包含状态机的文件必须标明哪个输入和输出是用带有关键字 MACHINE 的，而输入和输出又是预先定义了信号名的状态机。

注意 状态机的输入和输出端口类型不能用于顶层设计文件。虽然这种端口类型的顶层文件不能充分编译，但可以用 File 菜单中的 Project Save & Check 检查它们的句法，用 Creat Default Include File 创建代表当前文件的 Include 文件。

用暂态名，即在变量段输入“状态机别名定义”后，可以重新命名一个状态机。可以在状态机内被创建的文件中或使用 MACHINE INPUT 端口来输入状态机文件中使用状态机的别名。也可以用这个名字代替原始状态机的名字。下面程序所示的 ss_def. tdf 文件用 MACHINE OUTPUT 端口 ss_out 定义输出状态机 ss：

```
SUBDESIGN ss_def
(
    clk, reset, count       :INPUT;
    ss_out                  :MACHINE OUTPUT;
)
VARIABLE
    ss:MACHINE WITH STATES (s1, s2, s3, s4, s5);
BEGIN
    ss_out = ss;
    CASE ss IS
        WHEN s1 =>
            IF count THEN ss = s2;ELSE ss = s1;END IF;
        WHEN s2 =>
            IF count THEN ss = s3;ELSE ss = s2;END IF;
        WHEN s3 =>
            IF count THEN ss = s4;ELSE ss = s3;END IF;
        WHEN s4 =>
            IF count THEN ss = s5;ELSE ss = s4;END IF;
```

```
        WHEN s5 = >
            IF count THEN ss = s1 ;ELSE ss = s5 ;END IF ;
    END CASE ;
    ss.(clk, reset) = (clk, reset) ;
END ;
```

下面程序是 ss_use.tdf **文件。该文件使用** MACHINE INPUT **端口** ss_in **输入状态机。程序如下：**

```
SUBDESIGN ss_use
(
    ss_in  :MACHINE INPUT ;
    out    :OUTPUT ;
)
BEGIN
    out = (ss_in == s2) OR (ss_in == s4) ;
END ;
```

下面程序 top1.tdf **使用内部直接引用插入** ss_def **和** ss_use **功能。功能类型** ss_def **和** ss_use **包含关键字** MACHINE**。它指明哪一个输入和输出是状态机。程序如下：**

```
FUNCTION ss_def (clk, reset, count) RETURNS (MACHINE ss_out) ;
FUNCTION ss_use (MACHINE ss_in) RETURNS (out) ;

SUBDESIGN top1
(
        sys_clk, /reset, hold      :INPUT ;
        sync_out                   :OUTPUT ;
)
VARIABLE
        ss_ref :MACHINE ;  % Machine Alias Declaration %
BEGIN
        ss_ref = ss_def(sys_clk, !/reset, !hold) ;
        sync_out = ss_use(ss_ref) ;
END ;
```

图 4－31 **是与** top1.tdf **等效的** GDF **文件。**

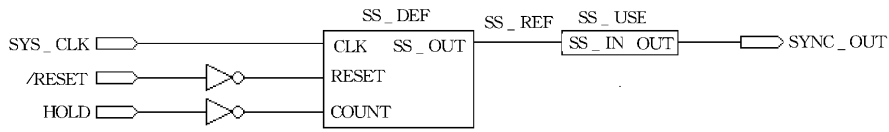


图 4－31　top1.tdf

变量段中，可以用实例定义实现一个外部状态机。下面程序所示的 top2.tdf **文件与** top1.

tdf 功能相同,但使用实例定义代替内部直接引用来实现功能。程序如下:

```
FUNCTION ss_def (clk, reset, count) RETURNS (MACHINE ss_out);
FUNCTION ss_use (MACHINE ss_in) RETURNS (out);

SUBDESIGN top2
(
        sys_clk, /reset, hold      :INPUT;
        sync_out                   :OUTPUT;
)
VARIABLE
        sm_macro                   :ss_def;
        sync                       :ss_use;
BEGIN
    sm_macro.(clk, reset, count) = (sys_clk, !/reset, !hold);
    sync.ss_in = sm_macro.ss_out;
    sync_out = sync.out;
END;
```

4.4.6 实现 LCELL 和 SOFT 原语

将 NODE 变量转换为 LCELL 和 SOFT 原语,可以限制逻辑综合的内容。NODE 变量和 LCELL 原语为逻辑综合提供最大限度的控制。SOFT 原语为逻辑综合提供较少的控制。

NODE 变量是在变量段用结点定义语句来描述的,它对逻辑综合的限制很小。综合过程中,逻辑综合器用变量代表逻辑取代 NODE 变量的每一个实例。然后,去简化逻辑,以适应一个简单的逻辑单元。这种方法通常可以获得最快的速度,但是会使逻辑过于复杂而难于调整。

SOFT 缓冲器控制如何使用资源。它要比 NODE 变量复杂。逻辑综合器可以选择何时用 LCELL 原语取代 SOFT 原语。SOFT 缓冲器可以帮助删除过于复杂的逻辑,因而使适配更加容易。但是也会增加逻辑单元的数量,减缓运行速度。

LCELL 原语可提供最多的控制,逻辑综合器简化所有驱动 LCELL 原语的逻辑,从而使逻辑适用于一个简单的逻辑单元。LCELL 原语总是在一个逻辑单元中实现,即使只有一个简单的输入,LCELL 原语也不能从设计中移走。用 SOFT 原语代替 LCELL 原语。SOFT 原语在逻辑综合的过程中就可以被移走。

MAX+PLUS Ⅱ 提供了逻辑选项,可以自动插入或移走 SOFT 和 LCELL 缓冲器。在帮助文件的‘Assigning a Logic Option’中可以获得更详细的信息。下面两部分程序为一个 TDF 文件的两种写法:一个用 NODE 变量实现;另一个用 SOFT 原语实现。在 nodevar 中,变量 odd_parity 被定义为 NODE,然后指定布尔表达式 d0 $ d1 $ ... $ d8 的值。在 softbuf 中,编译器用 LCELL 原语取代一个 SOFT 原语,从而提高器件的利用率。程序如下:

TDF with NODE Variables:

```
SUBDESIGN nodevar
(
```

TDF with SOFT Primitives:

```
SUBDESIGN softbuf
(
```

```
        ⋮
)
VARIABLE
    odd_parity :NODE;
BEGIN
    odd_parity = d0 $ d1 $ d2
              $ d3 $ d4 $ d5
              $ d6 $ d7 $ d8;
END;
```

```
        ⋮
)
VARIABLE
    odd_parity :NODE;
BEGIN
    odd_parity = SOFT(d0 $ d1 $ d2)
              $ SOFT(d3 $ d4 $ d5)
              $ SOFT(d6 $ d7 $ d8);
END;
```

4.4.7 实现 RAM 和 ROM

MAX + PLUS Ⅱ(和 AHDL)提供了几种 LPM 函数和其他 megafunctions,可以实现 RAM 和 ROM。一般来说,这些函数的可预测性保证可以使用它们实现任一个 MAX + PLUS Ⅱ支持的 RAM 或 ROM 类型。

注意:ALTERA 不支持创建自定义逻辑函数实现存储器。在需要时,可以使用 ALTERA 提供的函数实现 RAM 或 ROM。

以下 megafunctions 可以用来在 MAX + PLUS Ⅱ中实现 RAM 和 ROM。

1)lpm_ram_dq　带独立输入、输出端口的同步或异步存储器。

2)lpm_ram_io　带单独 I/O 端口的同步或异步存储器。

3)lpm_rom　同步或异步的只读存储器。

4)csdpram　循环双端口存储器。

5)csfifo　先入先出循环存储器。

在这些 LPM 函数中,不论数据输入、地址/控制输入和输出是被寄存或是非被寄存,一个 RAM 块将包含一个初始化的存储器存数文件。各种参数皆用来决定输入、输出数据的宽度。数据存放在存储器中。

4.4.8 命名一个布尔运算符或比较符

可以在 AHDL 文件中命名布尔运算符和比较符,使其易于进行资源分配,并解释设计报告文件(.rpt)的等式部分。下面程序是 boole3.tdf 文件。该文件与 boole1.tdf 文件实现的功能相同,但前者使用了命名的运算符。运算符的名字与运算符之间用冒号分开,名字可以包含 32 个字符。程序如下:

```
SUBDESIGN boole3
(
    a0, a1, b   :INPUT;
    out1, out2  :OUTPUT;
)
BEGIN
    out1 = a1 tiger :& !a0;
    out2 = out1 panther :# b;
```

```
END;
```

下面报告文件的程序可以说明 boole3.rpt **和** boole1.rpt **的不同点：**

```
-- boole3.rpt equations:
-- Node name is 'out1' from file "boole3.tdf" line 7, column 2
-- Equation name is 'out1', location is LC3_A1, type is output
out1 = tiger0;
-- Node name is 'tiger0' from file "boole3.tdf" line 7, column 18
-- Equation name is 'tiger0', location is LC2_A1, type is buried
tiger0 = LCELL(_EQ002);
  _EQ002 = !a0 &  a1;
-- boole1.rpt equations:
-- Node name is 'out1' from file "boole1.tdf" line 7, column 2
-- Equation name is 'out1', location is LC3_A1, type is output
out1 = _LC2_A1;
-- Node name is ':33' from file "boole1.tdf" line 7, column 12
-- Equation name is '_LC2_A1', type is buried
LC2_A1 = LCELL(_EQ001);
  _EQ001 = !a0 &  a1;
```

依靠等式逻辑，一个命名的运算符可产生多个节点名，但是所有的名字都是在运算符名字的基础上建立的，在报告文件中易于识别。 boole3.rpt **中，一个简单的节点** tiger0 **是在第一个等式中产生的。在** boole1.tdf **中，编译器将网络** ID :33 **分配给同一节点。**

对一个设计进行编译后，可以使用报告文件中基于运算符的节点名进行下一步编译的资源分配，不论设计逻辑改变与否。当文件中不相关的逻辑(如其他等式)改变时，命名运算符时建立的逻辑单元名字保持不变。例如，可以在节点 tiger0 **执行任务。相对应的，若运算符没有被命名，则只有网络** ID **号是有效的。这些号码被随机地重新分配给每一个编译器。**

4.4.9 使用层次化设计生成逻辑

要使用多个相似的逻辑块时，可以用 FOR GENERATE **语句生成层次化逻辑。程序如下：**

```
CONSTANT NUM_OF_ADDERS = 8;
SUBDESIGN iter_add
(
    a[NUM_OF_ADDERS..1], b[NUM_OF_ADDERS..1], cin      :INPUT;
    c[NUM_OF_ADDERS..1], cout                          :OUTPUT;
)
VARIABLE
    sum[NUM_OF_ADDERS..1], carryout[(NUM_OF_ADDERS+1)..1] :NODE;
BEGIN
    carryout[1] = cin;
```

```
    FOR i IN 1 TO NUM_OF_ADDERS GENERATE
        sum[i] = a[i] $ b[i] $ carryout[i];    % Full Adder %
        carryout[i+1] = a[i] & b[i] # carryout[i] & (a[i] $ b[i]);
    END GENERATE;
    cout = carryout[NUM_OF_ADDERS+1];
    c[] = sum[];
END;
```

上面程序即 iter_add.tdf **文件中,**FOR GENERATE **语句把全加器实例化。每个全加器实现** NUM_OF_ADDERS **位的选加。每一位的** carryout **是与每一位全加器一起产生的。**

注意:当用 For Generate **语句处理特殊情况时,**If Generate **语句特别有用。例如,在一个多级乘法器的第一级和最后一级。**

4.4.10 使用迭代生成逻辑

如果需要在一个算术表达式的值的基础上实现不同功能,使用 If Generate **语句可以有条件地产生逻辑。一个** If Generate **语句列出了一系列的状态语句,在确实求出一个或多个算术表达式的值之后它们才起作用。下面程序(即** condlog1.tdf **文件)使用一个** If Generate **语句执行基于当前器件系列关于** output_b **输出的不同状态:**

```
PARAMETERS (DEVICE_FAMILY);
SUBDESIGN condlog1
(
    input_a :INPUT;
    output_b :OUTPUT;
)
BEGIN
    IF DEVICE_FAMILY == "FLEX8K" GENERATE
        output_b = input_a;
    ELSE GENERATE
        output_b = LCELL(input_a);
    END GENERATE;
END;
```

如上例所示 MAX+PLUS Ⅱ **包括预先定义的参数** DEVICE_FAMILY**,以及预先定义的可用于算术表达式的计算函数** USED。DEVICE_FAMILY **参数可以用来测试当前对这个设计用** Device(Assign **菜单)器件系列。**USED **的计算功能可用来检测一个** Optional **端口是否能在当前实例中使用。**USED **把端口名当作输入。若端口未被使用则返回一个** FALSE **值。**

实现 LPM **功能的** TDF **文件中,有许多带有** If Generate **语句。这些文件放置在** \ max-plus2 \ maxlib \ mega_lpm **目录下(**UNIX **工作站上,为**/usr **的子目录)。**

4.4.11 使用断言(Assert)语句

使用 Assert **语句可检测任一随机表达式的有效性。随机表达式使用参数、数字、计算函**

数或者使用或者不使用一个端口的状态。例如,为测定一个被选择的参数值是否落在用第二个参数值测定的范围内,用户可以使用 Assert 语句。

有条件地使用 Assert 语句时,必须列出满足条件的值。若有一个值不被接受,assertion 将被激活,并发出一条信息。若使用无条件 Assert 语句,assertion 总是处于激活状态。

在编译器网表提取器计算出所有的参数值后,编译器对每一个 assertion 条件只估算一次。一个 assertion 不能依赖于在器件中执行的信号值。例如,如果一个 Assert 语句被放在 If Then 语句‘ If a = VCC Then c = d ’之后,那么 assertion 条件不能依赖 a 的值。

下面程序中 condlog2.tdf 与 condlog1.tdf 有相同的功能,但前者使用 Assert 语句报告逻辑段中 If Generate 语句产生那一个逻辑。程序如下:

```
PARAMETERS (DEVICE_FAMILY);
SUBDESIGN condlog2
(
    input_a :INPUT;
    output_b :OUTPUT;
)
BEGIN
    IF DEVICE_FAMILY == "FLEX8000" GENERATE
        output_b = input_a;
        -- Assertion is always activated if there is no condition
        ASSERT
            REPORT "Compiling for FLEX8000 family"
            SEVERITY INFO;
    ELSE GENERATE
        output_b = LCELL(input_a);
        -- Assertion is activated if current family is not FLEX10K
        -- or FLEX 8000. Severity defaults to ERROR
        ASSERT (DEVICE_FAMILY == "FLEX10K")
            REPORT "Compiling for % family", DEVICE_FAMILY;
    END GENERATE;
END;
```

第五章　设计实例与技巧

在第三章的基础上，本章将以几个具体开发实例深入介绍 MAX + PLUS Ⅱ 的设计技巧和设计经验。书中提到的 FPGA 器件仅指 ALTERA 公司的产品。由于各公司的 FPGA 产品内部结构不同，因此，本章所述设计经验及技巧并不能完全移用到其他 FPGA 产品中。

5.1　设计稳定性

在第三章介绍编译器时，提到打开设计医生可以检查电路设计中的不稳定因素（图 3 - 24）。一个用分立元件构成的电路不变地移用到 FPGA 设计中，很有可能出现性能下降、工作不稳定的现象。这是因为 FPGA 的设计与分立元件设计有很多不同之处。本节要探讨的问题是 FPGA 设计稳定性的一般要求。

5.1.1　FPGA 的设计特点

FPGA 的设计和传统的通用 IC 设计在许多方面有区别，表 5 - 1 对此作了归纳。在采用微处理器的通用 IC 设计中，性能受处理器性能限制。这时，各种不同功能元件之间的互连通常不成问题。然而，在 FPGA 中，连线可能占用了 70 % 的芯片面积，并且有可能影响门的利用率。

表 5 - 1　ASIC 与 CPLD/FPGA、通用 IC 设计区别表

方法 / 项目	通用 IC 设计	CPLD/FPGA 设计
设计方向	从片子到系统	从系统到片子
成本限制	元件数量	工作量及芯片价格
性能限制	功能单元设计	设计及开发工具性能
设计方案选择	主要元件（如处理器）	FPGA 芯片
可测试性要求	可连到 PCB 板上	【注】
验证	制作印刷电路板	模拟
样机制作	通常在实验室内完成	可在实验室内完成
后期更改设计	不方便	方便
设计方法	不灵活	有很大的灵活性
工具	可以不依赖于 CAE	强烈依赖 CAE

【注】对于大规模集成电路，测试分为功能测试和制造后测试。功能测试目的在于验证设计是否能正确地按照技术条件实现其功能；而制造后测试目的在于检查生产的每一片芯片是否合格，也称为结构测试。很显然，对于 FPGA 器件只需要功能测试，因为每一片芯片在出厂前都已作过结构测试。

5.1.2 FPGA 设计的基本单元

在 FPGA 的设计中,可将所有的设计元素抽象成五类基本单元。这些基本单元用于组成分层结构的设计。它们有:

①布尔单元,包括反相器和‘与’门、‘或’门、‘非’门、‘与非’门、‘或非’门、‘异或’门等;

②开关单元,包括传输门、多路选择器和三态缓冲器;

③存储单元,包括边缘敏感器件;

④控制单元,包括译码器、比较器;

⑤数据调整单元,包括加法器、乘法器、筒型移位器、编码器。

在设计中明确定义所用基本单元类别就可以避免所谓‘无结构的逻辑设计’,并花费短的设计时间得到清晰的、结构完善的 FPGA 设计。

5.1.3 信号的分类

FPGA 中的所有信号可以分为时钟、控制信号和数据三种。

简单的时钟信号用于控制所有的边缘敏感触发器,别无它用,也不受任何其他信号的限制。控制信号,如‘允许’和‘复位’,用于使电路元件初始化、使之保持在当前状态、在几个输入信号间做出选择或使信号通到另外的输出端。若干控制信号可以来自同一个允许产生器,但受到状态计数器的控制。数据信号中含有数据,它可以是一些单独的比特,也可以是总线中的并行数据。

5.1.4 FPGA 中的同步设计技术

首先应明确同步的概念,因为它是形成同步设计的基础。一个系统是同步的,需满足:

①每个边缘敏感部件的时钟输入是一次时钟输入的某个函数,并且仍是像一次时钟那样的时钟信号;

②所有存储元件(包括计数器)都是边缘敏感的,在系统中没有电平敏感存储元件,也就是要求存储元件仅在有效时钟边缘上存在状态变化。

在 FPGA 设计中,异步逻辑设计存在很多弊端。这也是由 FPGA 内部结构决定的。利用 FPGA 的逻辑块结构实现异步设计可能会造成逻辑和互连资源的浪费,而且会产生电路中的竞争和冒险。因为 FPGA 中的逻辑块结构一般由馈入到可配置触发器的大规模 AND 阵列和 OR 阵列组成,并且该逻辑块中的所有寄存器必须由同一信号提供时钟。因此若一个逻辑电路中存在多个时钟信号,则必须划分相应个数的逻辑块给具有不同时钟信号的逻辑电路使用。这样就将本来不复杂的电路逻辑分散到多个逻辑块中,并且由于所使用的逻辑块的时钟信号已被占用,具有其他时钟的信号也不能再添加进去,造成逻辑块中大量门阵列的浪费。

图 5-1 是一种典型的异步计数器电路,图 5-2 是等效的利用同一时钟的计数器电路。在 FPGA 设计中,图 5-2 是更有效的一种设计方法。

在传统数字逻辑设计中,由异步电路带来的冒险一般可以采用三种方法消除,即在逻辑表达式中增加冗余项、增加惯性延迟环节以及使用选通法。前两种方法是将经过较少门延时的信号人为地插入延时,使其在时间上与经过较多门延的信号在时间上保持一致。第三种方法是利用选通脉冲在信号作用时间上加以控制,避开产生冒险的尖峰。但 FPGA 逻辑电路的设

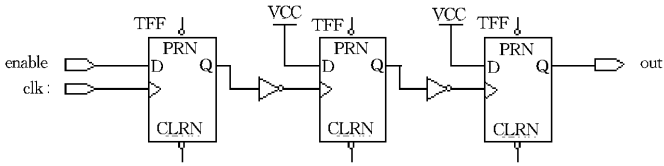


图 5－1　异步计数器(级连时钟)

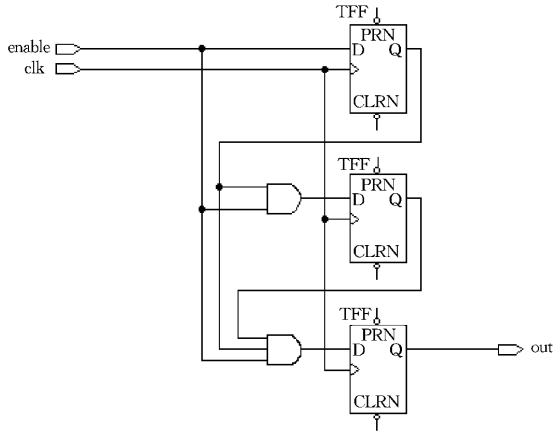


图 5－2　同步计数器(全局时钟)

计有别于分立元件的电路设计。在分立元件的设计中,可以精确计算出每个信号在相互作用前所经过的门延时间,从而可以估计冒险是否发生及如何消除。在 FPGA 设计中,由于系统自动完成布线,为了均衡逻辑块的资源,信号在整个 FPGA 内部经过的门数可能并非是在逻辑表达式中所期望的那样,有时无法正确估计某个信号的延时,而且增加冗余项也会加大逻辑块的开销。因此,第三种方法是最适合于 FPGA 逻辑设计的。

5.1.5　FPGA 设计的稳定性

以下内容是在 FPGA 设计中常遇到的问题,同时,设计医生也依据这些规则来检查电路设计的稳定性。

一、异步输入(Asynchronous Inputs)

许多设计要求各异步系统之间进行同步通信,或者同步系统需要异步输入控制。如果异步输入不满足时钟建立和保持时间的限制,将会导致受控的同步系统出现逻辑混乱。如图 5－3 二进制计数器中,异步输入信号作为使能信号控制计数器的工作就是一个典型的异步输入的例子。

在 FPGA 设计中,图 5－3 情况可以用图 5－4 的方法解决,即在异步输入和计数器之间插入一个 D 触发器(flipflop),从而解决异步输入的不稳定问题。注意,一个异步信号只能驱动一

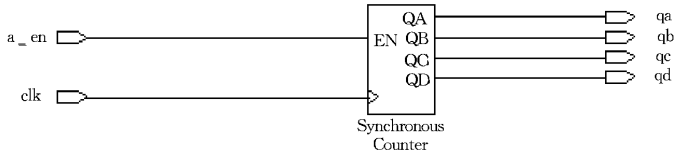


图 5－3　异步输入作为计数器的使能信号

个触发器，否则就会出现不稳定现象。

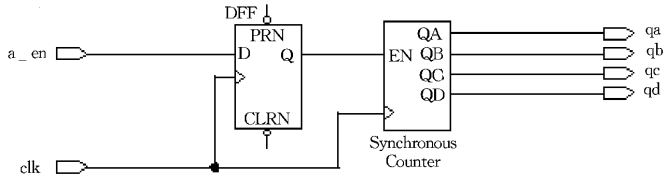


图 5－4　用一个触发器检测异步输入信号

图 5－5 是另一种解决办法。它适用于异步输入使能信号宽度小于时钟周期的情况。

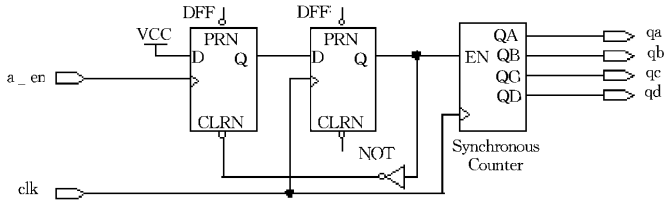


图 5－5　用两个触发器检测异步输入信号

二、时钟电路

在 FPGA 中，稳定的时钟电路是良好设计的基础。若时钟电路设计不当，在环境因素（如电压、温度等）变化时，会严重导致电路逻辑混乱。在设计过程中，建议尽可能使用全局时钟，并且 ALTERA 的 FPGA 器件都有一根全局时钟的引脚。全局时钟可以以最短的路径连接片内所有触发器，从而使输出延迟降到最小。在无法使用全局时钟时，可以使用门控时钟，而多级时钟、级连时钟和多时钟网络将导致电路不稳定。

1. 门控时钟（Gated Clocks）

门控时钟以一个“与”门或一个“或”门的输出作为时钟信号。门控时钟达到全局时钟同样的稳定性时必须满足两个条件：

①门的输入信号中只能有一个信号作为时钟信号；

②门控时钟只能由单个的“与”门或“或”门构成。

图 5－6 是“与”门时钟的例子，图 5－7 是“或”门时钟的例子。这些都是稳定的时钟电路设计。

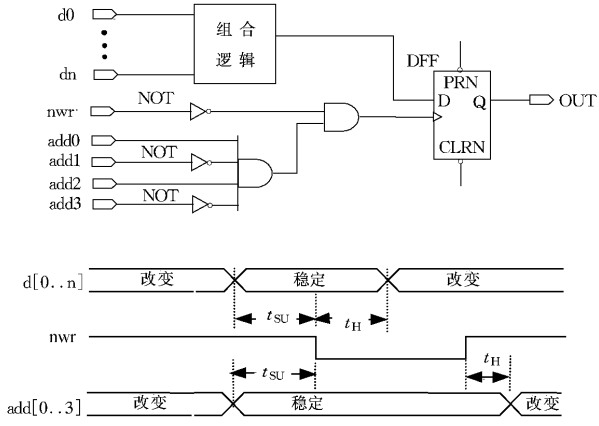


图 5－6 “与”门门控时钟

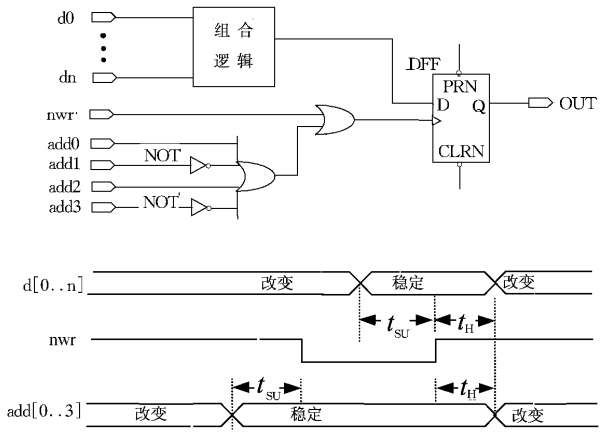


图 5－7 “或”门门控时钟

在以上两图中，nwr **和** nwe **是时钟信号(**nwr **下降沿有效，**nwe **上升沿有效)。该电路常常作为微处理器与外部设备之间的接口。但在** FPGA **设计中，可以将以上两个门控时钟转化为全局时钟电路。“与”门时钟转化为全局时钟如图** 5－8 **所示。“或”门时钟原理与此类似。**

2. **多级时钟(**Multi-Level Clocks**)**

多级时钟即利用多于一个门的组合电路输出作为时钟信号。这种情况下极易出现静态冒险，如图 5－9 **所示。**

多级时钟可以转化为图 5－10 **所示的全局时钟图。**

3. **行波时钟(**Ripple Clocks**)**

在异步计数器的设计中，级连时钟应用较多。级连时钟即以一个触发器的输出(Q**)作为**

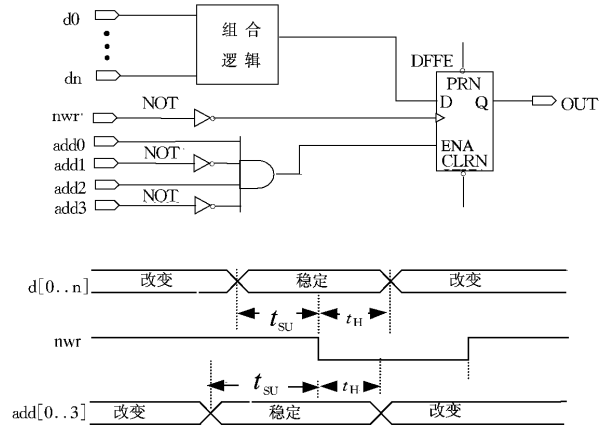


图 5－8　与门时钟转化为全局时钟

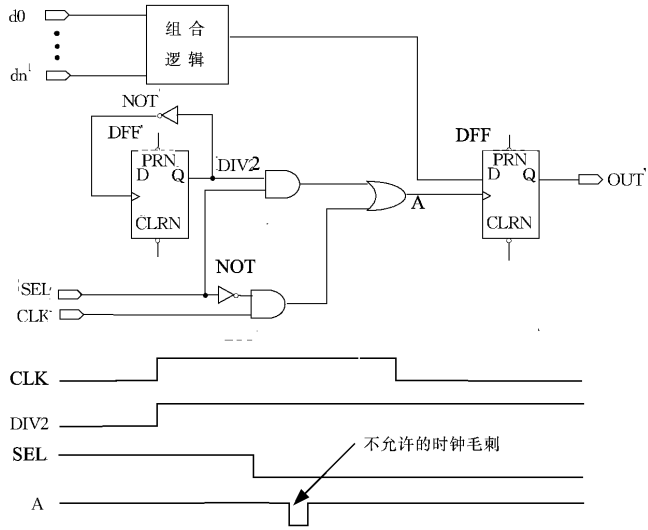


图 5－9　多级时钟(不稳定)

另一个触发器的时钟。在 FPGA **中,级连时钟最易造成时钟歪斜,并导致系统工作极不稳定,如图** 5－1 **所示。改进后的设计如图** 5－2 **所示。**

4. **多时钟网络(**Multi-Clock Network**)**

同样,由于时钟建立、保持时间的限制,FPGA **设计中不宜采用多时钟网络,如图** 5－11 **所示。改进后的电路如图** 5－12 **所示。**

在以上四种时钟设计中,只有门控时钟可以在 FPGA **设计中应用,其他类型均不符合** FP-

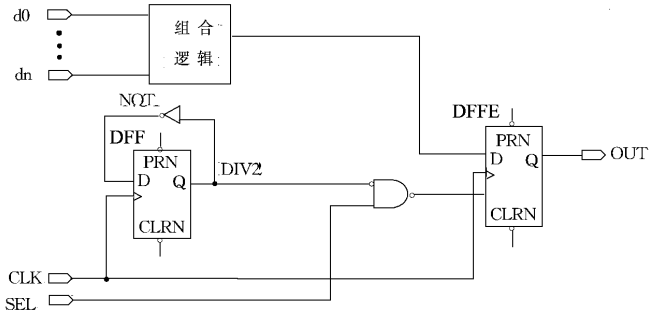


图 5－10　多级时钟转化为全局时钟

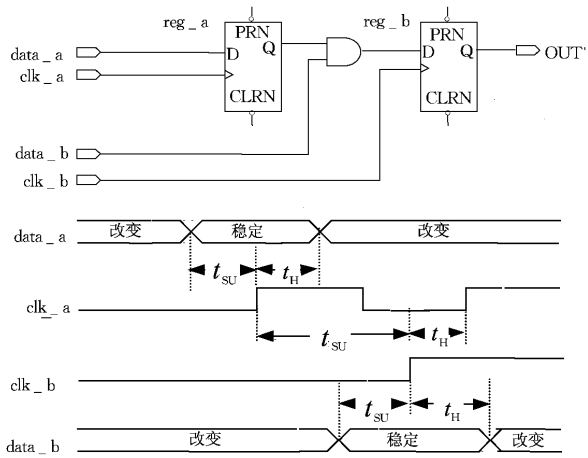


图 5－11　多时钟网络

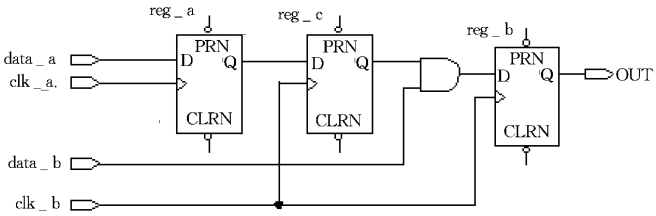


图 5－12　多级时钟网络加同步触发器

GA **稳定性的设计规则，应尽量避免。在上述各图中，**t_{SU}**为建立时间，**t_H**为保持时间。由以上的改进措施可以看出，它们共同的特点是使用全局时钟，并将控制信号转化为触发器的使能信号。这是** FPGA **同步电路设计中常用的设计方法。**

三、延时链(Delay Chains)

利用 MAX + PLUS Ⅱ 中的 LCELL 或 EXP 可以形成内部延时或用来产生脉冲。但这种设计在环境温度、电压等变化时,性能变化很大,且极不稳定,并易造成竞争(race)现象。图 5 - 13 的设计试图通过一系列 EXP 产生一个异步脉冲;图 5 - 14 是转化后的可替代方案的设计图。

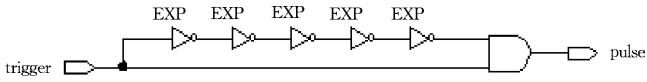


图 5 - 13　用 EXP 产生异步脉冲

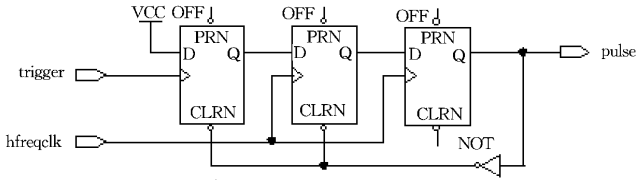


图 5 - 14　用触发器产生异步脉冲

四、全局复位信号(Master Reset)

尽管 ALTERA 公司的 EPLDs 器件和 FLEX8000 器件具有内置的加电复位电路,但在设计中应尽量保证有一全局复位信号,或保证触发器、计数器在使用之前已经正确清零和状态机处于确知的状态。

五、静态冒险(Static Hazards)

在一个电路中,若有多个输入变量同时变化,且变化前后电路输出相同,这时可能出现瞬时错误输出,导致输出端出现毛刺,这种现象称为静态冒险。在 FPGA 设计中,消除静态冒险最有效的办法是加入选通脉冲。具体实例如图 5 - 9 和 5 - 10 所示。

六、竞争(Race Conditions)

竞争现象出现在一个信号经过两条或多条不同的路线后到达并控制同一个单元时出现。信号不可能同时到达该单元,也即每条线路的延时不同。竞争现象会导致输出震荡或使输出无法预测。图 5 - 15 是异步电路设计中常遇到的问题,相应解决办法如图 5 - 16 所示。

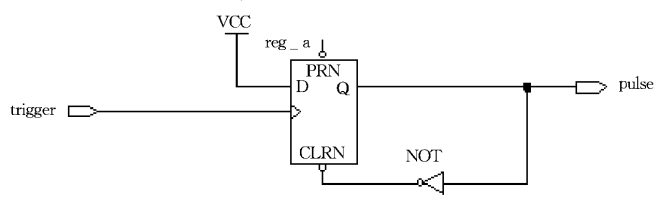


图 5 - 15　用异步电路产生脉冲(存在竞争现象)

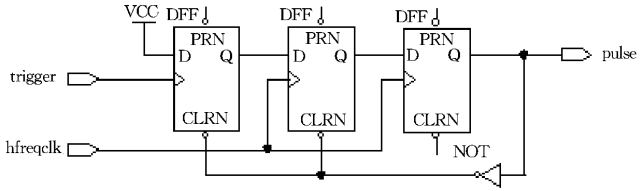


图 5－16　用同步电路产生同步脉冲

七、置位与清零电路(Preset & Clear Configurations)

与时钟一样，置位与清零信号对竞争条件和冒险非常敏感，所以在设计中必须小心对待它们，最好的置位与清零配置是由器件的引脚作为全局信号直接驱动。在图 5-17～**图** 5-22 **中列出了几种置位与清零电路供读者参考。**

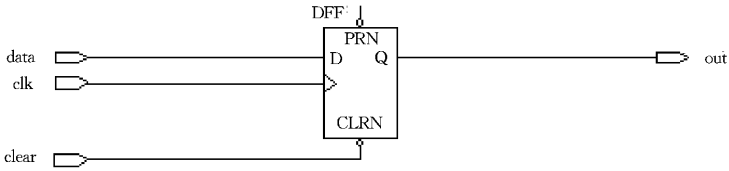


图 5－17　引脚清除电路

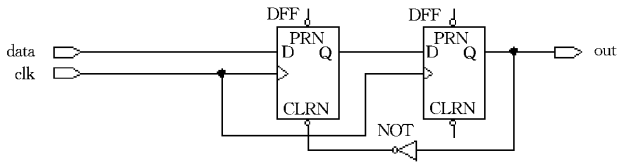


图 5－18　寄存器清除电路

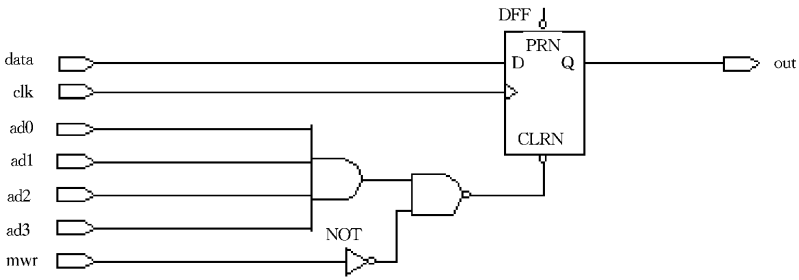


图 5－19　门控清除电路

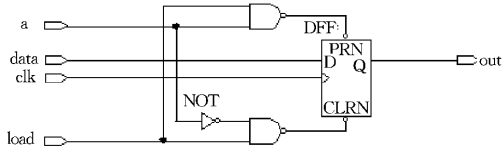


图 5－20　异步装入电路

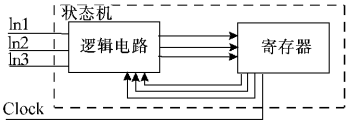


图 5－21　异步清除电路

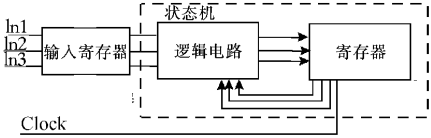


图 5－22　同上清除电路

5.2　频率计实例

由于 ALTERA 公司提供的 MAX 系列芯片与 FLEX 系列芯片分别为‘与非’门结构和查找表结构，所以在选用不同的芯片进行设计时应采用不同的设计方法。下面通过频率计和滤波器设计实例介绍这两种芯片的设计技巧。本节着重介绍采用 MAX7000 系列芯片设计频率计实例，下一节重点介绍采用 FELEX10K 系列芯片设计滤波器的实例。

频率计的基本功能是根据基准时钟对被测时钟进行检测，并且将被测时钟的频率值在数码管上显示出来。这里采用的工作原理是利用基准时钟产生 1s 的时间宽度，从而在这 1s 的时间宽度里对被测时钟记数，然后将最后的记数结果送到数码管显示。因此，根据前面对频率计基本工作原理的分析可以知道它的硬件实现电路不是十分复杂，采用具有‘与非’门结构的 MAX7000 系列芯片完全可以实现它的基本功能。这里将整个频率计划分为以下两个功能模块：基准时间产生模块(fre_base)与被测时钟频率记数模块(fre_count)。基准时间产生模块用于产生 1s 的时间宽度；被测时钟频率记数模块在 1s 的时间宽度里对被测时钟记数，并将最后的结果送给数码管显示。它们与顶层文件(fre_example)的层次关系如 5－23 图所示。

下面按照设计步骤自下而上的顺序介绍各个模块。

1. 基准时间产生模块(fre_base)

基准时间产生模块采用文本编辑方式设计，源程序如下：

```
INCLUDE "lpm_counter";

PARAMETERS
(
   base_num  =10        --若基准时钟为 100 ms，则 1s 的时间信号需分频 10 次
);                      这个参数可以随基准时钟频率的不同而改变
```

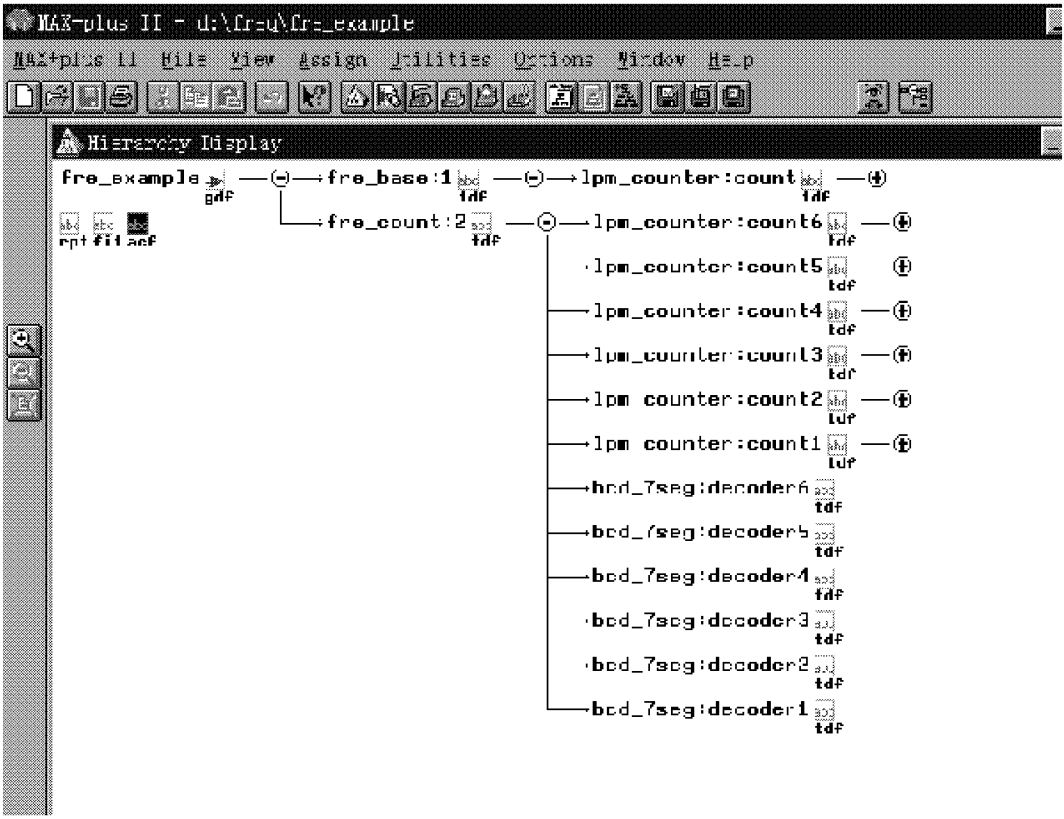


图 5-23　频率计实例层次关系图

```
CONSTANT
width = ceil(log2(base_num));          --设置计数器的记数宽度
                                       --当 base_num = 10 时,width = 4
SUBDESIGN fre_base                     --定义输入、输出端口
(
  base_clk        :INPUT;
  reset           :INPUT = GND;
  base            :OUTPUT;
)

VARIABLE
count             :lpm_counter WITH (lpm_modulus = base_num,
                                     lpm_width = width,
                                     lpm_direction = "up");
start[2..1]       :DFF;
flip              :DFFE;
clr               :NODE;
```

```
trigger          :NODE;

BEGIN
count.clock = GLOBAL(base_clk);
count.sclr = clr;

start[].clk = base_clk;
start1.d = reset;
start2.d = start1.q;

clr = start1.q & !start2.q;                    --开始计数

trigger = (count.q[] == base_num-1);  --设置记数值

flip.ena = trigger;                            --产生 1s 的时间宽度
 flip.clrn = !clr;
 flip.d = !flip.q;
 base= flip.q;

END;
```

基准时间产生模块有两个输入(base_clk 和 reset)与一个输出(base)。其中 base_clk 接收基准时钟信号;reset 可对基准时间产生模块进行复位;输出 base 为一个以 2s 作为周期、占空比为 100% 的时钟信号。该模块利用 lpm_counter 函数作为计数器,在 reset 为低电平时,对基准时钟信号记数。一旦 reset 为高电平,则由 D 触发器组 start[2..1]捕捉 reset 的变化,产生清零信号(clr)对计数器清零,重新开始记数。当计数器记到额定值之后,触发带使能端的 D 触发器 flip 产生基准时间信号 base。

2. 被测时钟频率记数模块(fre_count)

被测时钟频率记数模块(fre_count)包含两个包含文件(lpm_count 和 bcd_7seg)。其中 lpm_count 属于 ATERLA 公司提供的参数化函数,具有很大的使用范围,有关细节可参考 4.3 节;bcd_7seg 是一个将 4 位 BCD 码转换为七段译码的子模块。这里先介绍 bcd_7seg 子模块,然后再介绍被测时钟频率记数模块(fre_count)源程序。

(1)bcd_7seg 子模块

bcd_7seg 子模块采用列真值表的方法实现由 BCD 码到七段译码的转换,源程序如下所示。(其中'0'代表数码管相应的段发光显示;'1'代表数码管相应的段不发光显示。注释值为显示相应数字所需驱动数码管的十六进制代码值。子程序模块如下:

```
SUBDESIGN bcd_7seg
(
        in[4..1]                    :INPUT ;
        g,f,e,d,c,b,a               :OUTPUT;
```

```
)

BEGIN
TABLE
    in[4..1]        =>      g, f, e, d, c, b, a
     0              =>      0, 0, 0, 0, 0, 0, 1;    -----01H
     1              =>      1, 0, 0, 1, 1, 1, 1;    -----4FH
     2              =>      0, 0, 1, 0, 0, 1, 0;    -----12H
     3              =>      0, 0, 0, 0, 1, 1, 0;    -----06H
     4              =>      1, 0, 0, 1, 1, 0, 0;    -----4CH
     5              =>      0, 1, 0, 0, 1, 0, 0;    -----24H
     6              =>      1, 1, 0, 0, 0, 0, 0;    -----60H
     7              =>      0, 0, 0, 1, 1, 1, 1;    -----01H
     8              =>      0, 0, 0, 0, 0, 0, 0;    -----00H
     9              =>      0, 0, 0, 1, 1, 0, 1;    -----0CH
    10              =>      0, 1, 1, 0, 0, 0, 0;    -----30H
    11              =>      0, 1, 1, 0, 0, 0, 0;    -----30H
    12              =>      0, 1, 1, 0, 0, 0, 0;    -----30H
    13              =>      0, 1, 1, 0, 0, 0, 0;    -----30H
    14              =>      0, 1, 1, 0, 0, 0, 0;    -----30H
    15              =>      0, 1, 1, 0, 0, 0, 0;    -----30H
END TABLE;
END;
```

(2)被测时钟频率记数模块(fre_count)

源程序如下：

```
INCLUDE "lpm_counter";
INCLUDE "bcd_7seg.inc";

PARAMETERS
(
  precise = 3                    --测量精度,也即数码管个数
);


SUBDESIGN fre_count
(
  check_clk                      :INPUT;
  base                           :INPUT;
  out[precise..1][7..1]          :OUTPUT;
```

```
  overflow                    :OUTPUT;
  bin[precise..1][4..1]       :OUTPUT;      --作为中间节点的输出
  )

  VARIABLE
  count[precise..1]           :lpm_counter WITH ( lpm_modulus = 10,
                                                  lpm_width = 4,
                                                  lpm_direction = "up");
    decoder[precise..1]       :bcd_7seg;
  delay[precise..2]           :DFF;
  enab[2..1]                  :DFF;
    flip[precise..1][7..1]    :DFFE;
    clr                       :NODE;
  delay_n[precise..1]         :NODE;

  BEGIN
    count[].clock = GLOBAL(check_clk);
    count[].sclr = clr;
  count[1].cnt_en = base;

  delay[].clk = check_clk;
  delay[].clrn = !clr;
  delay_n[1] = count[1].eq9;
  FOR i IN 2 TO precise  GENERATE          --实现十进制记数进位
  delay_n[i] = delay_n[i-1] & count[i-1].eq9;
  delay[i].d = delay_n[i];
  count[i].cnt_en = delay[i].q;
  END GENERATE;

    overflow = count[precise].eq10;       --超出测量范围

    enab[].clk = check_clk;
    enab[1].d = base;
  enab[2].d = enab[1].q;
    clr = !enab[1].q & enab[2].q;         --捕捉基准时间结束
                                          --以产生清零信号

  decoder[].in[4..1] = count[].q[];    --实现 BCD 到七段译码的转换
```

```
    flip[][].ena = clr;
    flip[][].clk = check_clk;

  FOR i IN 1 TO precise GENERATE
    bin[i][] = count[i].q[];                    --输出中间结果
    flip[i][1].d = decoder[i].a;                --锁存最后的译码结果
    flip[i][2].d = decoder[i].b;
    flip[i][3].d = decoder[i].c;
    flip[i][4].d = decoder[i].d;
    flip[i][5].d = decoder[i].e;
    flip[i][6].d = decoder[i].f;
    flip[i][7].d = decoder[i].g;
  END GENERATE;

  out[][] = flip[][].q;                         --输出频率记数值

    END;
```

被测时钟频率记数模块(fre_count)有两个输入端(base 和 check_clk)与三种输出信号(out[precise..1][7..1]、overflow 和 bin[precise..1][4..1])。在输入端中,base 接收基准时间信号;check_clk 接收被测时钟信号。在输出信号中,out[precise..1][7..1]用于驱动数码管,显示被测时钟的频率值。在高电平时,overflow 表明被测时钟超过测量范围;bin[precise..1][4..1]是为了方便读者观察输出结果而设置的中间节点输出。该模块在 base 信号由低电平变为高电平时开始对被测时钟信号记数。记数功能通过一组 lpm_counter(count[precise..1])函数实现。每一个 lpm_counter 分别对应频率值中十进制的相应权值位,从 0 到 9 记数。lpm_counter 之间的十进制进位通过 D 触发器组 delay[]实现。D 触发器组 enab[2..1]捕捉 base 信号由高电平到低电平的变化,一方面将最后的频率记数值译码后锁存,另一方面产生 lpm_counter 的清零信号,使其能够开始记数。

3. 顶层设计文件(fre_example .gdf)

当设计完以上两个模块后,要分别进行编译和仿真,以检查模块设计的正确性,看它们能否正确地实现各自的功能。如果编译和仿真的结果是正确的,就可以开始进行顶层文件的设计了。这里所举的频率计实例的顶层文件采用图形编辑方式设计,文件名为 fre_example .gdf,如 5-24 图所示。

因为在频率计实例中共有两个时钟输入信号 BASE_CLK 和 CHCK_CLK ,所以为了保证设计的稳定性和测试的灵活性,在整个顶层文件中为两个模块在 MAX7000 系列芯片中选择 EPM7032LC44-6 和 EPM7096QC100-7 芯片,分别以 BASE_CLK 和 CHCK_CLK 作为全局时钟。在选用不同的基准频率和测试不同的频率时,只需要改变相应模块的时钟输入与参数,就可以完成频率计的基本测试功能。

在完成频率计的整个顶层文件设计后,按照前文所述步骤对其进行编译和仿真,以检查是否能达到预期的功能。如图 5-25 所示,被测时钟周期为 64μs,仿真结果为 015625,与其频率

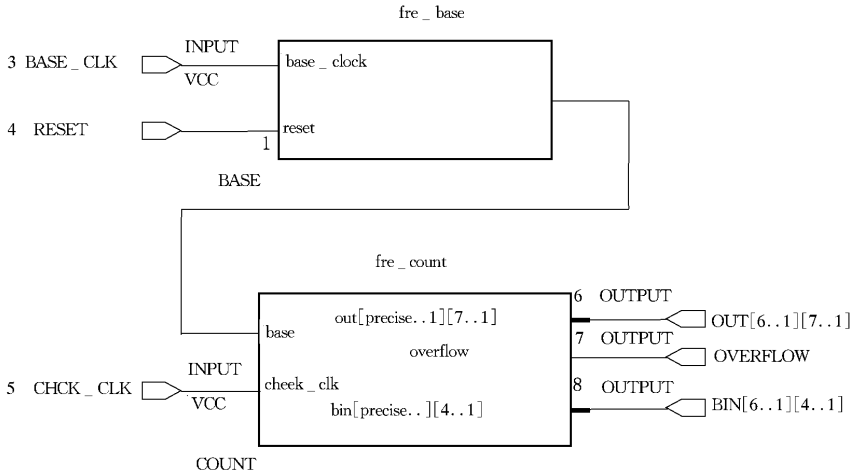


图 5－24　频率计实例顶层文件设计图形 fre_example. gdf

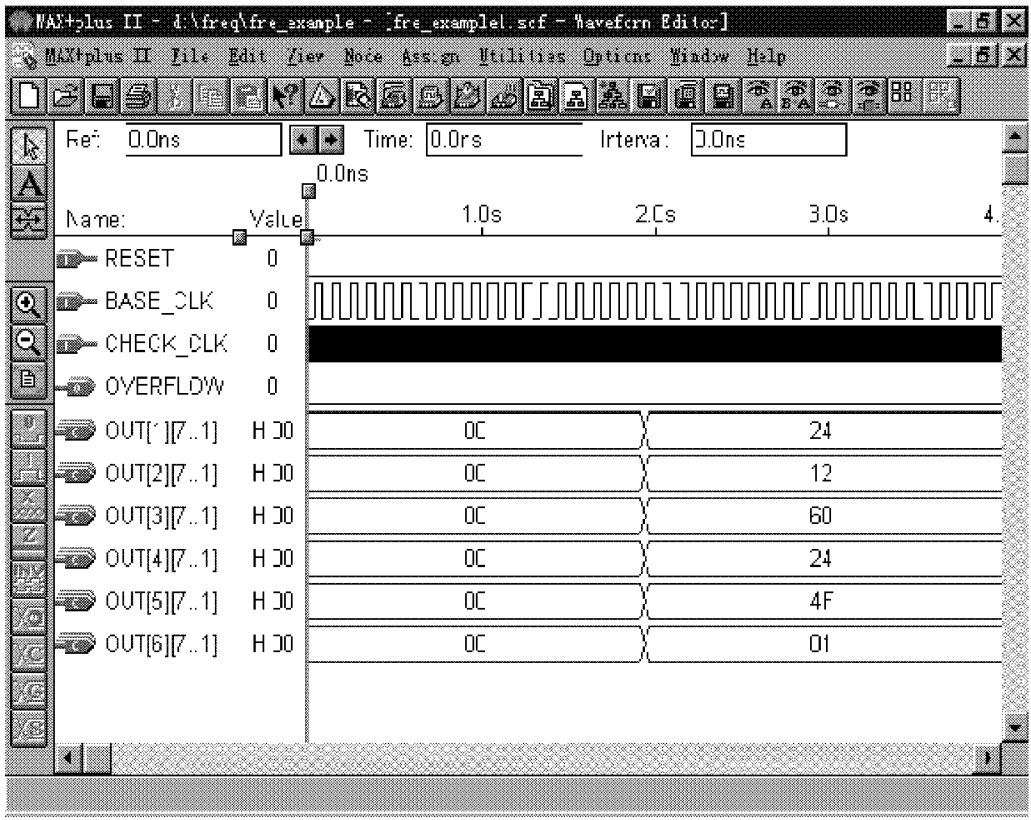


图 5－25　被测时钟周期为 6μs 的仿真结果

值 15625 是一致的。又如图 5－26 所示，被测时钟周期为 1.001ms，复位信号 RESET 在测试期间对其复位。在复位信号有效后，仿真结果从 000999 变为 000599，最后稳定在 000999，与其频率值 999 还是一致的。

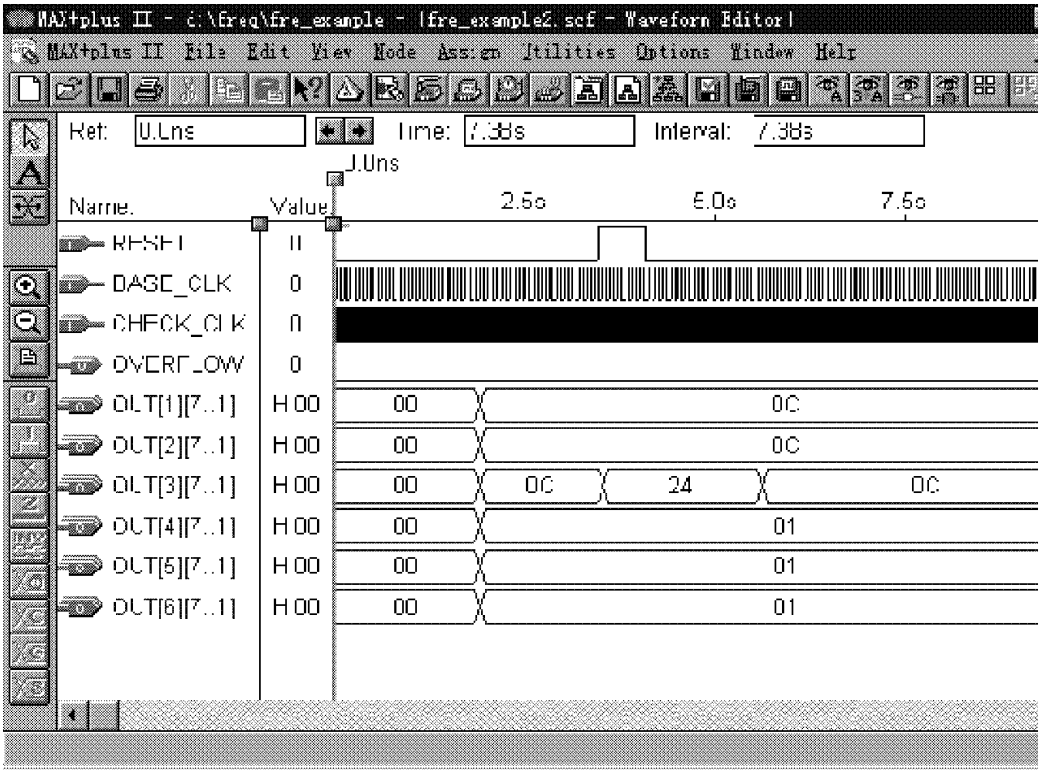


图 5－26　被测时钟周期为 1.001ms 的仿真结果

上面设计的频率计能够测试频率值在 1～1MHz 范围内的时钟信号，并将被测时钟信号频率值的整数部分通过数码管显示出来。介绍这个例子的主要目的是希望读者能够通过它了解采用 MAX 系列芯片进行设计的整个过程，从而迅速掌握设计方法。

5.3　数字滤波器实例

数字滤波器是数字信号处理的一个重要分支，利用它可以在形形色色的信号中提取需要的信号和抑制不需要的信号（干扰、噪声）。数字滤波器实质上是用一有限精度算法实现离散时间线性非时变系统，以完成对信号进行滤波处理的功能。其输入是一组由模拟信号经过取样和量化的数字量，输出是经过处理的的另一组数字量。在实际应用中，多数情况下，利用数字滤波器对模拟信号进行处理的过程一般如图 5－27 所示。

图中输入端接一个低通滤波器 $H_1(s)$ 对输入信号 $x_t(t)$ 的频带进行限制，以避免频谱混叠。输入信号经过 $H_1(s)$ 的预处理后，进行抽样、量化（即所谓的 A/D 转换），然后进入数字滤

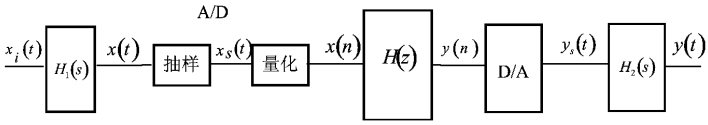


图 5－27　数字滤波器处理模拟信号过程框图

波器 $H(z)$滤波。滤波后的信号 $y(n)$经过 D/A 转换通过另一个低通滤波器 $H_2(s)$，以便将 D/A 输出的模拟量良好地恢复成时间连续信号。

数字滤波器根据单位冲激响应 $h(n)$的时间特性分为无限冲激响应(IIR)数字滤波器和有限冲激响应(FIR)数字滤波器两种。有限冲激响应(FIR)数字滤波器按照基本结构分为直接型、级联型和频率抽样型三种。本节重点介绍采用 FELEX10K 系列芯片进行设计的有限冲激响应直接型数字滤波器实例。

有限冲激响应数字滤波器直接型的结构如图 5－28 所示。它是用一条均匀间隔抽头的延迟线上对抽头信号进行加权求和构成。

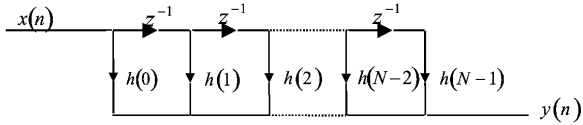


图 5－28　FIR 数字滤波器直接形式

上述结构的 FIR 数字滤波器的输入输出关系所用的时域卷积定理为

$$y(n)\sum_{m=0}^{N-1}h(m)x(n-m) \tag{5-1}$$

根据上式，可以看出数字滤波器涉及到大量的卷积运算，使用硬件实现时会占用大量的资源。为了避免这种情况出现，可充分利用 FELEX10K 系列芯片具有的查找表结构，将卷积运算转化为查表移位求和实现。下面以一个简单的卷积运算为例说明硬件实现结构。对于式

$$y=[x(1)(h(1)]+[x(2)(h(2)]+[x(3)(h(3)]+[x(4)(h(4)] \tag{5-2}$$

假设 x 和 h 都是无符号整型二进制数，宽度两位，取值如下：

$$h(1)=01,\ h(2)=11,\ h(3)=10,\ h(4)=11$$

$$x(1)=11,\ x(2)=00,\ x(3)=10,\ x(4)=01$$

被乘数 $h(n)$ →	01	11	10	11	
乘数 $x(n)$ →	11	00	10	01	
中间数据 $P(n)$ →	01	00	00	11	= 100
中间数据 $P_i(n)$ → +	01	00	10	00	=011
	011	000	100	011	=1010

图 5－29　卷积运算过程

图 5－29 表示式(5－2)运算的实现。在图中可以看到运算纵向和横向分别实现。中间数据 $P1(n)$中的四个数据实际上是乘数 $x(n)$的最低位比特与 $h(n)$相乘的结果，并且该值不是 0 就是 $h(n)$。这是因为二进制的取值只有 0 和 1。进一步考虑，中间数据 $P1$ 和 $P2$ 的值，即“100”和“011”是由不同的 $h(n)$之和构成，而对 $h(n)$的选择是由乘数 $x(n)$的相同位的比特

决定的。如上图中 $x(n)$的最低位为 1001，则 $P1$ 的值为 $h(1)+h(4)$；其高位为 1010，则 $P1$ 的值为 $h(1)+h(3)$。因此利用 ALTERA 公司 FLEX 器件中的查找表(LUT)结构，预先将 $h(n)$的各种组合(在本例中从 0000 到 1111 共 16 种)存入查找表，则上例中的原需 4 次乘法和 3 次加法的卷积运算转化为一次加法。图 5－30 显示了用查找表实现该例的结构。

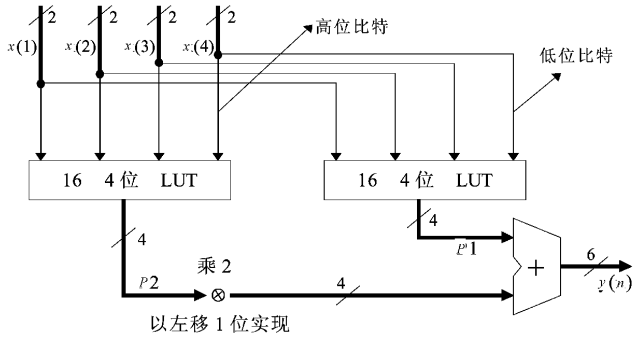


图 5－30　查找表实现卷积运算

用查找表实现卷积运算时，有并行和串行两种结构。图 5－30 显示的就是并行结构，其中的两个 LUT 是完全相同的。在并行结构中，LUT 的数量由 $x(n)$的数据宽度决定。一位对应一个 LUT，这样速度能够达到最大，但占用资源也相当可观。而串行结构中，只需一个 LUT，$x(n)$的每位比特串行查表，并利用累加器累加得出最后结果。显然串行结构比并行结构占用资源要少得多，但代价是处理速度降低。

为了充分说明 FELEX10K 系列芯片查找表结构的使用方法，这里所举的 FIR 数字滤波器实例采取全局并行的方案，即将输入 $x(n)$经过不同的延迟后同时进行处理。数字滤波器实例层次图(Hierarchy Display)如图 5－31 所示。图中只显示了两层结构，更下层的模块没有标出。其中内部各个模块之间的关系如图 5－32 所示。

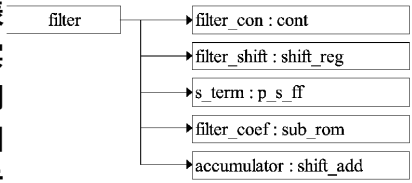


图 5－31　FIR 数字滤波器层次图

下面按照自下而上的顺序对各模块进行介绍。

1. 时延环节模块(filter_shift)

时延环节模块使 A/D 转换后的数据通过不同的触发器，从而产生不同的延迟。该模块采用文本编辑方式进行设计，源程序如下：

```
 PARAMETERS
(
        tap = 3,
  x_wid = 12,
);

SUBDESIGN filter_shift
```

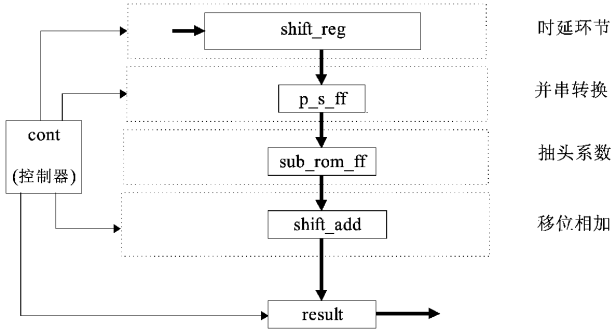


图 5－32　FIR 数字滤波器内部模块示意图

```
(
 clk                                  :INPUT;
 clr                                  :INPUT = VCC;
 xin[x_wid..1]                        :INPUT;
 s_en                                 :INPUT = GND;
 y_ff[tap..1][x_wid..1]               :OUTPUT;
)

VARIABLE
  shifter_ff[tap..1][x_wid..1]        :DFFE;

BEGIN(连接移位寄存器的输入端)
     shifter_ff[][].clrn = clr;
     shifter_ff[][].clk = clk;
     shifter_ff[][].ena = s_en;
     shifter_ff[1][].d = xin[];
     FOR i IN 2 TO tap GENERATE
    shifter_ff[i][].d = shifter_ff[i-1][].q;
     END GENERATE;
     FOR i IN 1 TO tap GENERATE
      y_ff[i][] = shifter_ff[i][].q;
END GENERATE;
END;
```

2. **并/串转换模块**

并/串转换模块将通过时延模块产生的不同延迟分别同时转换为查找表的串行地址，提供给抽头系数模块。并/串转换模块包含一个单通道并/串转换子模块(p_s **模块)。这里先介绍** p_s **子模块，然后再介绍并/串转换模块。**

(1)单通道并/串转换子模块(p_s 模块)

该模块能够将一个通道的并行数据转换为串行数据。程序中,将并行数据宽度参数化,以便能够适应不同宽度数据的需要。模块采用文本编辑方式进行设计,源程序如下:

```
PARAMETERS
(
width = 12                          --并入宽度
);

SUBDESIGN p_s
(
parallel_in[width..1]    :INPUT;          --并入
serial_out               :OUTPUT;         --串出
clk                      :INPUT;
clr                      :INPUT = VCC;
load_n                   :INPUT = GND;
)

VARIABLE
reg[width..1]            :DFF;

BEGIN                                    --给所有的触发器连接时钟和清除信号
reg[].clk = clk;
reg[].clrn = clr;
IF (load_n) THEN                         --装载
  FOR j in 1 TO width GENERATE
  reg[j].d = parallel_in[j];
  END GENERATE;
ELSE
  FOR j in 1 TO width-1 GENERATE         --移位
  reg[j].d = reg[j+1].q;
  END GENERATE;
  reg[width].d = GND;
  END IF;
    serial_out = reg[1].q;
END;
```

(2)并/串转换模块(s_term 模块)

在单通道并/串转换的基础上,并/串转换模块(s_term 模块)实现多通道数据的并/串转换。模块采用文本编辑方式进行设计,源程序如下:

```
INCLUDE "p_s";
```

```
PARAMETERS
  (
  in_width = 12,              --进行转换的并行数据宽度
  term_wid = 3                --进行转换的并行数据通道数
);

  SUBDESIGN s_term
  (
  clk                                    :INPUT;
  clr                                    :INPUT = VCC;
  data_in[term_wid..1][in_width..1]      :INPUT;
  load                                   :INPUT = GND;
  rom_ad[term_wid..1]                    :OUTPUT;
)

  VARIABLE
  regs[term_wid..1]    :p_s WITH ( width = in_width )  ;

  BEGIN
  regs[].clk = clk;
  regs[].clr = clr;
  regs[].load_n = load;

  FOR i IN 1 TO term_wid GENERATE
  regs[i].parallel_in[] = data_in[i][];
  END GENERATE;

  rom_ad[] = regs[].serial_out;

END;
```

3. **抽头系数模块**(filter_coef)

抽头系数模块将抽头系数的各种组合固化在 ROM **中。它的地址输入端接收并/串转换模块的串行输出,然后查表得到卷积的中间数据,并将这些数据输出到移位相加模块。模块采用文本编辑方式进行设计,源程序如下:**

```
  INCLUDE "lpm_rom";

  PARAMETERS
  (
```

```
    addr_wid = 3,
    coef_wid =16,
  ini_file = "coef.mif"
);

  SUBDESIGN filter_coef
  (
    clk                              :INPUT ;
    rom_addr[addr_wid..1]            :INPUT ;
    dataout[coef_wid..1]             :OUTPUT;
  )

  VARIABLE
    rom1                  :lpm_rom
                          WITH (LPM_WIDTH = coef_wid,
                                LPM_WIDTHAD = addr_wid,
                                LPM_FILE = ini_file,
                                LPM_OUTDATA= "REGISTERED");
  BEGIN
    rom1.inclock = clk;
    rom1.outclock = clk;
    rom1.address[] = rom_addr[];
    dataout[] = rom1.q[];
  END;
```

这里需要特别解释的是 LPM_ROM 的初始化文件(coef.mif)以 ASCII 字符表示,后缀为.mif的 MIF(Memory Initialization File)文件在编译和仿真时说明 RAM 或 ROM 的初始内容。对于 RAM 或 ROM 的每一个地址,MIF 文件包含着相应的初始值。在设计中一般将前面提到的查找表写入到相应 ROM 使用的 MIF 文件中。在编写 MIF 文件时需要说明存储数据的长度和宽度。在下面的例子中,假设抽头系数数据宽度为 16 位,精度为 15 位。其中:

h(1) = 0.125(1000H)
h(2) = 0.5(2000H)
h(3) = -0.125(A000H)

则相应的 MIF 文件(coef.mif)的编写如下所示:

```
WIDTH = 16;                    --说明存储数据宽度
DEPTH =8;                      --说明存储数据长度

ADDRESS_RADIX = HEX;           --说明存储数据地址采用十六进制表示
DATA_RADIX = HEX;              --说明存储数据数值采用十六进制表示
```

```
CONTENT BEGIN
    0 :0000;
    1 :1000;                    (h(1))
    2 :2000;                    (h(2))
    3 :3000;                    (h(1)+h(2))
    4 :A000;                    (h(3))
    5 :0000;                    (h(1)+h(3))
    6 :1000;                    (h(2)+h(3))
    7 :2000;                    (h(1)+ h(2)+h(3))
END;
```

在 MIF(coef.mif)文件中,存储数据值和地址时不仅可以使用十六进制(HEX),而且还可以使用二进制(BIN)、八进制(OCT)、十进制 (DEC)。在 CONTENT BEGIN 和 END 之间的每一个程序语句中,冒号(:)左边为存储数据地址,右边为存储数据数值,存储数据的数据宽度与数据个数必须分别与前面为 WIDTH 和 DEPTH 所赋的值相等。当按照上面的格式完成 MIF 文件编写后,在存盘的时候一定要在文件名的后面加上后缀(.mif)。

4. 移位相加模块

移位相加模块通过将中间数据移位相加而实现两个数相乘的功能。为了充分说明该模块的功能,假设被乘数为 3(0011)、乘数为 13(补码为 1101),那么乘积过程为

```
          0011
×         1101
-----------------
          0011
+       0000
+     0011
-   0011
-----------------
    11110111
```

在上式中,乘数的各位分别为 1 或 0。当为 1 时,被乘数作为中间数据参加移位相加运算;当为 0 时,则由 0 作为中间数据参加移位相加运算。如果乘数数据宽度为 n,则前 $n-2$ 次为加法运算,最后一次为减法运算。因为被乘数、乘数都是以补码参加运算,所以乘积也是以补码的形式出现,这样就涉及到中间数据符号位的问题了。为了说明这个问题,再假设被乘数为 -3(补码为 1101)、乘数为 3(0011)。它们的数据宽度都为 4 位,小数点后面的精度为 0;乘积结果数据宽度为 8 位,小数点后面的精度为 0。那么,乘积过程为:

```
        1101
×       0011
-------------
        1101
       11101        --扩展符号位
+      1101         --第一次进行加法运算
-------------
      110111
     1110111        --扩展符号位
+    0000           --第二次进行加法运算
-------------
     1110111
    11110111        --扩展符号位
-   0000            --第三次进行减法运算
-------------
    11110111
```

在上式中,乘积的结果不仅与符号位有关,而且与被乘数、乘数以及所要求乘积的数据宽度、小数点后的数据精度有关,乘积结果的整数部分可以根据对乘积大小的估计设置为一定宽度。小数部分的宽度可根据精度的要求进行取舍。这样就要求模块对被乘数、乘数和乘积的数据宽度、精度均应设置参数,通过对这些参数赋予不同的值以适应不同的需要。

下面是根据上述的设想设计的源程序:

```
INCLUDE "lpm_add_sub";

PARAMETERS
(
  x_width = 16,          --乘数(即要进行累加的数)之宽度:一位整数、十五位小数
  x_pre = 15,            --乘数之精度
  y_width = 12,          --被乘数(即决定累加次数的数)之宽度:二位整数、十位小数
  y_pre = 10,            --被乘数之精度
  out_int = 3,           --乘积的整数部分位数
  out_pre = 15,          --乘积的小数部分位数
  pipeline = "YES"
);

  CONSTANT dff_num = (x_width-x_pre) + (y_width-y_pre) + out_pre;
  CONSTANT out_width = out_int + out_pre ;

  SUBDESIGN accumulator
  (
    clk                    :INPUT;
    clr                    :INPUT = VCC;
    datain[x_width..1]     :INPUT;
```

```
    add_sub                    :INPUT = VCC;     --加减 控制
    f_en                       :INPUT = GND;     --输出使能控制
    a_load                     :INPUT = GND;     --装入数据控制
    dataout[out_width..1]      :OUTPUT;
    )
  VARIABLE
    adder          :lpm_add_sub WITH (lpm_width = x_width+1,
                   lpm_representation = "UNSIGNED");

    accum[dff_num..1]    :DFF;

    IF(pipeline == "YES") GENERATE
    flip[out_width..1]                :DFFE;
    END GENERATE;
    adder_dataa[x_width+1..1]         :NODE;
    adder_datab[x_width+1..1]         :NODE;
    adder_result[x_width+1..1]        :NODE;

  BEGIN
    accum[].clk = clk;
    accum[].clrn = clr;
    CASE a_load IS
     WHEN VCC =>
        accum[dff_num] = datain[x_width];
        accum[dff_num-1..dff_num-x_width] = datain[];
        IF (dff_num-x_width-1 > 0) GENERATE
          accum[dff_num-x_width-1..1] = GND;
          END GENERATE;
       WHEN GND =>
         accum[dff_num..dff_num-x_width].d = adder_result[];
         accum[dff_num-x_width-1..1].d = accum[dff_num-x_width..2];
      END CASE;

        adder.add_sub = add_sub;

        adder_datab[] = (datain[x_width], datain[]);    (扩展符号位)
        adder_dataa[x_width+1] = accum[dff_num];
        adder_dataa[x_width..1] = accum[dff_num..dff_num-x-width+1];
        adder.dataa[] = adder_dataa[];
```

```
        adder.datab[] = adder_datab[];
        adder_result[] = adder.result[];

        IF(pipeline == "YES") GENERATE
          flip[].clk = clk;
          flip[].clrn = clr;
          flip[].ena = f_en;
          flip[].d = accum[out_width..1].q;
          dataout[] = flip[].q;
        ELSE GENERATE
          dataout[] = accum[out_width..1].q;
        END GENERATE;

END;
```

在以上程序的注释中已经对参数的设置进行了解释，而最后需要说明的是乘积结果的数据宽度应该为 28 位。但是，因为整数部分（包括符号位）取 3 位、小数点后面取 15 位已经能满足要求，所以设置乘积结果数据宽度为 18 位。

5. **控制器模块**（filter_con）

上述各种模块虽然各自能够完成一定功能，如延迟、并串转换、移位相加等，但是当它们按一定的形式组合在一起实现滤波器功能时，需要有一系列的控制信号对上述各种模块进行精确的控制。控制器模块在接收到 A/D 转换结束信号（AD_END）后，依次产生移位寄存器使能信号、并行到串行转换的装入信号、移位相加的装入信号、加减控制信号和滤波结果输出信号等各种控制信号，使上述各个模块按照一定的时序进行操作，最终完成滤波功能。该模块采用文本编辑方式进行设计，源程序如下：

```
PARAMETERS
(
x_width = 12,        --输入数据的宽度，将决定 add_sub 和 flip_en 在何时起作用
);

CONSTANT dff_num = x_width + 6;

SUBDESIGN  filter_con
(
  clk          :INPUT ;
  clr          :INPUT = VCC;
  ad_end       :INPUT ;       --A/D 转换结束信号
  shifter_en   :OUTPUT;       --移位寄存器使能信号
  ps_load      :OUTPUT;       --并行到串行转换的装入信号
  add_load     :OUTPUT;       --移位相加的装入信号
```

```
    add_sub         :OUTPUT;      --加减控制信号
    flip_en         :OUTPUT;      --滤波结果输出信号
  )

  VARIABLE
    shift[dff_num..1] :DFFE;
    base              :NODE;

  BEGIN
    shift[].clk = clk;
    shift[].clrn = clr;

    shift[1].d = ad_end;
    shift[2].d = shift[1].q;
    base = !shift[1].q & shift[2].q;
    shifter_en = base;--以上四句为捕捉A/D转换结束信号,并且产生移位寄存器使能信号

    shift[3].d = base;
  FOR i IN 4 TO dff_num GENERATE
    shift[i].d = shift[i-1].q;
  END GENERATE;
    ps_load = shift[3].q;
    add_load = shift[6].q;
    add_sub = !shift[6+x_width-1].q;
    flip_en = shift[6+x_width].q;
  END;--以上产生其它控制信号
```

6. **顶层设计文件**(filter)

下面介绍数字滤波器的顶层设计文件。假设滤波器结构如图 5-33 **所示。在图** 5-33 **中,**$x(n)$**的数据宽度为** 12 **位,其中精度为** 10 **位 抽头系数(**$h(1)$**、**$h(2)$**和** $h(3)$**)的数据宽度为** 16 **位,精度为** 15 **位 输出结果数据宽度为** 18 **位,精度为** 15 **位,则顶层设计文件采用文本编辑方式设计的源程序如下:**

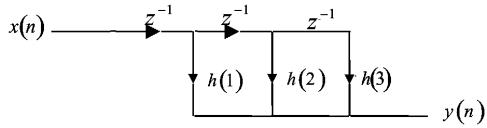


图 5-33 FIR **数字滤波器实例结构**

```
  INCLUDE "filter_con";
  INCLUDE "filter_shift";
```

```
INCLUDE "p_s";
INCLUDE "s_term";
INCLUDE "accumulator";
INCLUDE "filter_coef";

PARAMETERS
(
tap = 3,
x_wid = 12,              --x 的数据宽度为 12 位
x_pre = 10,              --x 的小数点后有 10 位精度
coef_wid = 16,           --滤波器系数数据宽度为 16 位
coef_pre = 15,           --滤波器系数小数点后有 15 位精度
z_wid = 18,              --输出 Z 的数据宽度为 18 位
z_pre = coef_pre,        --输出 Z 精度的小数点后有 15 位精度,2 位整数位,1 位符号
                         位
pipeline = "YES"
);

SUBDESIGN  filter
(
g_clk                    :INPUT ;
clr                      :INPUT = VCC;
ad_end                   :INPUT = VCC;   --A/D 转换完成信号
xin[x_wid..1]            :INPUT;         --输入数据
z[z_wid..1]              :OUTPUT;        --卷积结果
)

VARIABLE
cont          :filter_con WITH ( x_width =x_wid, );

shift_reg     :filter_shift WITH ( tap = tap,
                                   x_wid =x_wid,
                                   );

p_s_c         :s_term WITH (in_width =x_wid,
                            term_wid = tap);
rom_coef      :filter_coef WITH (addr_wid = tap,
                                 coef_wid = coef_wid ,
                                 ini_file = "coef.mif");
```

```
shift_add   :accumulator WITH (x_width = coef_wid ,---------16
                               x_pre = coef_pre,---------15
                               y_width =x_wild ,----------12
                               y_pre = x_pre,-------------10
                               pipeline = "YES",
                               out_int = z_wid - z_pre ,--3
                               out_pre = z_pre ----------15
                               );
clk :node;

BEGIN
clk = global(g_clk);
--连接控制器模块的输入端
  cont.clk = clk;
  cont.clr = clr;
  cont.ad_end = ad_end;

--连接时延环节模块的输入端
  shift_reg.clk = clk;
  shift_reg.clr = clr;
  shift_reg.xin[] = xin[];
  shift_reg.s_en = cont.shifter_en;

--连接并串转换模块的输入端
  p_s_c.clk = clk;
  p_s_c.clr = clr;
  p_s_c.load = cont.ps_load;
  p_s_c.data_in[][] = shift_reg.y_ff[][];

--连接抽头系数模块的输入端
  rom_coef.clk = clk;
  rom_coef.rom_addr[] = p_s_c.rom_ad[];

--连接移位相加模块的输入端
  shift_add.clk = clk;
  shift_add.clr = clr;
  shift_add.a_load = cont.add_load;
  shift_add.add_sub = cont.add_sub;
```

```
    shift_add.f_en = cont.flip_en;
    shift_add.datain[] = rom_coef.dataout[];

    z[] = shift_add.dataout[];          --输出卷积结果
END;
```

为了使读者能直观了解程序，还将顶层设计文件采用图形编辑方式 进行设计，如图 5－34 所示。

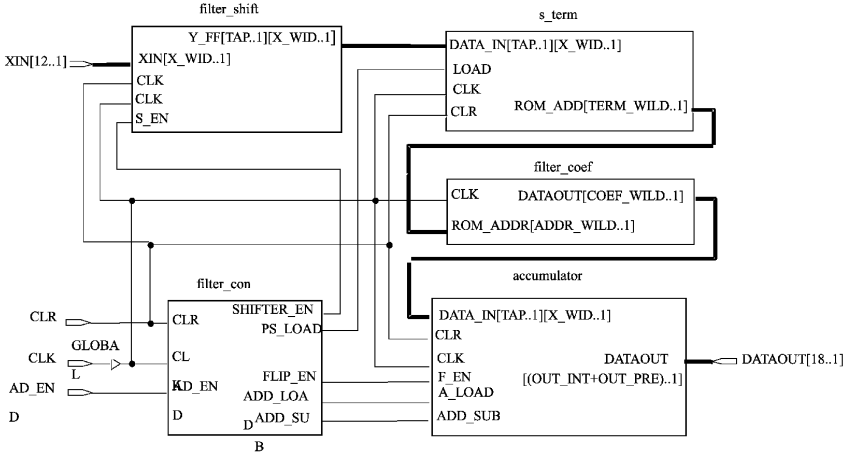


图 5－34　滤波器顶层文件设计图

通过对运用 AHDL 进行数字滤波器设计实例的介绍，着重叙述了在 FLEX10K 芯片中查找表结构的使用方法。因为上例中的抽头系数是固定值，所以存储它们使用 ROM。如果滤波器为自适应，则存储它们要使用 RAM，同时在设计中增加修改抽头系数的模块。但是无论使用 ROM 还是 RAM，都要涉及到查找表。查找表结构不仅能够实现上面提到的卷积运算，而且还可以把许多复杂的数学运算转化为查表。查找表结构使这些数学运算的硬件实现变得十分简单、灵活。另外需要说明的是，在数字滤波器实例中涉及到的子模块如延迟模块、并串转换模块、移位相加模块等都是独立的参数化模块，在其他的应用中根据需要对这些模块的参数进行赋值就可以使用，因而具有很强的灵活性。

5.4　盲均衡器 FLEX10K 器件的实现

信道均衡技术对实现高质量的通信非常重要，它可消除码间窄扰和噪声，并减小误码率。这项技术已由固定均衡、自动均衡发展到自适应均衡。然而自适应均衡必须发训练码，这就使信息传输过程中因训练序列的插入而引起传输时延，并且在有些情况下发送训练序列并非都有可能。而盲均衡技术可以克服这一缺陷，因而引起专家和学者的广泛研究。经过 20 多年的

努力，专家们提出了在不发训练序列情况下仍能使接收机进行自适应均衡的各种算法，从而达到正确接收信息的目的。这就是盲均衡技术。本节给出了一种采用 FPGA 技术实现盲均衡器的设计方法。

一、盲均衡器算法的确定

目前已提出的盲均衡算法主要有常量模板算法、高阶谱算法、神经网络算法以及基于环确定性等算法。其中 CMA 算法(见参考文献 9)的计算量相对较小，因此以 CMA 算法为例在本节中较详细地说明 FLEX10K 器件在盲均衡实现中的设计方法。

这里首先以图 5－35 盲均衡系统为例比较各类 CMA 算法的性能。假设图 5－35 中的均衡器 C 采用横向滤波器，并且均衡器输入信号矢量和均衡器抽头系数 $C_N^T(n)$ 更新矢量分别为：

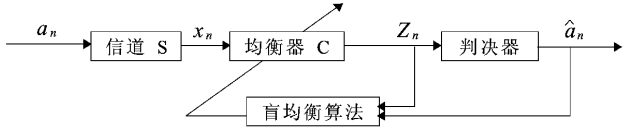


图 5－35　均衡器工作原理

$$X_N^{\mathrm{T}}(n)=[\chi_{n+N},\chi_{x+N-1},\cdots,\chi_{n-N}] \tag{5-3}$$

$$C_N^{\mathrm{T}}(n)=[C_{-N}(n),C_{-N+1}(n),\cdots,C_0(n),\cdots,C_N(n)] \tag{5-4}$$

其中 T 表示转置，则均衡器输出为：

$$Z_n=\sum_{i=-N}^{N}C_i\chi_{k-i} \tag{5-5}$$

各种 CMA 算法的不同之处仅在于误差控制函数不同，在 Y. Sato 提出的 SA 算法中，误差控制函数由下式决定：

$$f(Z_n)=Z_n-r*\mathrm{Sgn}(Z_n) \tag{5-6}$$

其中：$r=E[|a_n^2|]/E[|a_n|]$。 (5－7)

利用经典的最陡梯度算法，可推出均衡器抽头系数更新迭代算法：

$$\begin{aligned}C_i^{m+1}&=C_i^m-\lambda X_{k-i}f(Z_n)\\&=C_i^m-\lambda X_{k-i}(Z_n-r*\mathrm{Sgn}(Z_n))\end{aligned} \tag{5-8}$$

在图 5－35 中，均衡器 C 的目的是消除信道 S 的影响。当滤波器的冲激响应接近 S^{-1} 时(S 为信道的冲激响应)，C 的输出 Z_n 接近 a_n。当 C 等于 S^{-1} 时，C 的值为最佳值。然而，在 SA 算法中，$r*\mathrm{Sgn}(Z_n)$ 只是对真实信号的粗略估计，并且，当 $C=S^{-1}$ 时，误差控制函数 $f(Z_n)$ 虽然均值为零，但本身却不为零，从而导致均衡器在收敛后，仍有较大的均方误差。而通常的误差信号 $e_n=Z_n-a_n$ 却具备在 $C=S^{-1}$ 时均值为零且本身为零的性质。因此，将这两者结合起来，构成了 GSA 算法的误差控制函数：

$$\begin{aligned}f(Z_n)&=k_1\hat{e}_n+k_2|\hat{e}_n|e_n\\&=k_1(Z_n-\hat{a}_n)+k_2|Z_n-\hat{a}_n|(Z_n-r*\mathrm{Sgn}(Z_n))\end{aligned} \tag{5-9}$$

通过理论分析，与其他算法(如 SA、Serra 等)比较，GSA 算法具有良好的性能，而且计算量较小。因此选择该算法实现硬件。

二、盲均衡器结构

如前所述,与传统的自适应均衡器一样,盲均衡器也可采用线性或非线性结构。一般来说,非线性均衡器性能优于线性均衡器。对于线性均衡器,在无限抽头的假设条件下,最佳均衡问题实际上是一个逆滤波问题,它的频域特性是系统频响折叠谱之逆。但当实际信道存在较大畸变时,有可能是系统频响折叠谱形成凹陷点或零点,从而使逆滤波器对噪声十分敏感。判决反馈均衡器本质上是一种特殊形式的非线性算法,它把已经检测到的数据码反馈到输入端,以达到抵消码间串扰的目的。它对信道的幅度畸变和时延畸变均有良好的补偿作用。

以一个有 M 个前馈抽头和 N 个反馈抽头的判决反馈均衡器为例,在 nT 时刻进入 DFE 均衡器的信号矢量分别为:

$\boldsymbol{X}(n)=[\chi_{n+M-1},\chi_{n+M-2},\cdots,\chi_n]$ --前馈滤波器的输入信号矢量

$\hat{\boldsymbol{A}}(n)=[\hat{a}_{n-1},\hat{a}_{n-2},\cdots,\hat{a}_{n-N}]$ --反馈滤波器的输入信号矢量

均衡器的抽头系数矢量分别为:

$\boldsymbol{C}(k)=[C_{-m+1},C_{-M+2},\cdots,C_0]$ --前馈滤波器的抽头系数

$\boldsymbol{B}(k)=[b_1,b_2,\cdots,b_N]$ --反馈滤波器的抽头系数

因此,该时刻均衡器的输出为:

$$Z_n-\sum_{t=-M+1}^{0}C_iX_{n-1}-\sum_{t=1}^{N}b_i\hat{a}_{n-1} \tag{5-10}$$

抽头系数矢量的调整可按下式计算:

$$C(k+1)=C(k)-\lambda f(Z_n)X(n) \tag{5-11}$$

$$B(k+1)=B(k)-\lambda f(Z_n)\hat{A}(n) \tag{5-12}$$

其中,λ 为调整步长。$f(Z_n)$ 由式(5-9)表示。

三、盲均衡器结构的 FLEX10K 器件实现方案

1. 系统的分割

本系统设计是根据功能块自下而上分层次进行的。这样的设计方法可以节省设计时间、减少设计输入的错误、消除重复的电路元件,并能简化校验与进行修改。而在系统介绍时则遵循自上而下的顺序。系统总模块名“Blind Equalizer”,系统入口参数如图 5-36 所示。

上面英文名为入口参数名,等号后为默认值。系统其他底层模块将继承这些参数。因此,修改设计时,如改变滤波器节数、字长,可直接修改系统级参数。各参数具体的含义将在以后各模块中详细解释。

在图 5-35 中,信道 S 以右的部分构成了盲均衡器。从图中看出盲均衡器分为均衡器、判决器和均衡算法三部分。在实际系统中,按照模块的大小和功能分为以下三部分:PART Ⅰ包括接口及 DFE 均衡器;PART Ⅱ包括判决器及均衡算法;PART Ⅲ为抽头系数调整部分。系统层次图(Hierarchy Display)如图 5-37 所示。图中只显示了两层结构,更下层的模块没有标出。

系统硬件框图如图 5-38 所示。

假设均衡器的输入数据已是经过 A/D 转换的数字信号,数据流经过接口模块,进入 DFE 均衡器,完成式(5-10)的卷积运算,得出 Z_n。Z_n 是经过判决器输出均衡后的数据。同时,误差控制函数模块完成式(5-9)即 $f(Z_n)$ 的运算,最后,抽头系数调整模块完成对 DFE 均衡器

```
PARAMETERS
(
tap = 7,----------滤波器节数
term = 3, ----------RAM 组数，解释见后
pulse = 2, ----------输入信号调制方式
x_wid = 12, ----------输入信号宽度
x_pre = 10, ----------输入信号精度，即小数点后位数
main_coef_wid =17,----均衡器主抽头系数宽度
coef_wid = 16, ----------均衡器抽头系数宽度
coef_pre = 15, ----------均衡器抽头系数精度
z_wid = 18,----------均衡器输出数据宽度
z_pre = coef_pre,--------均衡器输出数据精度
err_wid = 17,---------- e_n = Z_n − â_n 数据宽度
err_pre = 15,---------- e_n = Z_n − â_n 数据精度
f_zn_wid = 9----------控制误差函数宽度
f_zn_pre = 5----------控制误差函数精度
);
```

图 5－36　系统入口参数

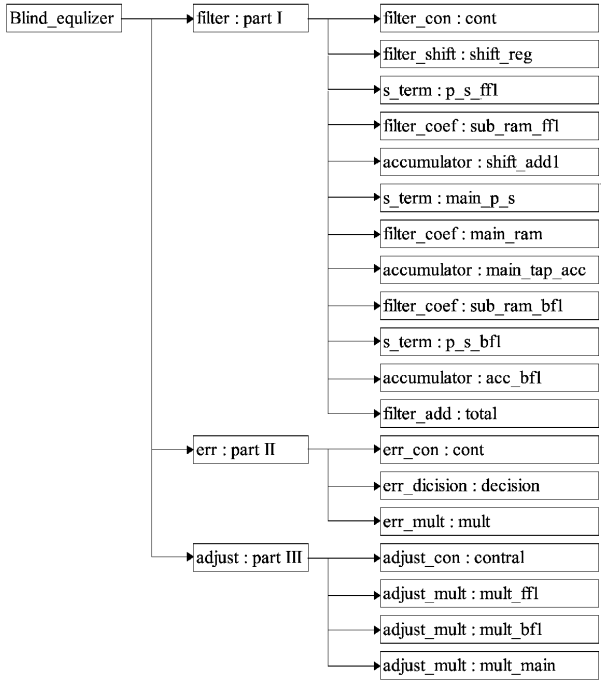


图 5－37　盲均衡系统层次图

抽头系数的调整。各模块之间传送的不仅有数据流，还有控制信号，或称之为同步信号，以保证接收数据的正确性。图 5－39 简要表示模块间通信时的波形示意图。出于灵活性的考虑，

系统采用类似于集散控制而非集中控制的方案。也就是说,各模块内部都有自己的控制器,整个系统并没有一个中心控制器。这样可以保证三大模块彼此之间相对独立。若要改变设计,比如改变算法,只需改变误差控制函数运算模块即可,其他模块可以不变。各模块功能的恰当分割大大增加了系统的灵活性。

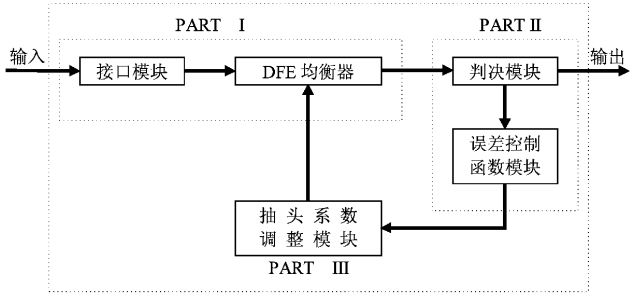


图 5－38　盲均衡器硬件框图

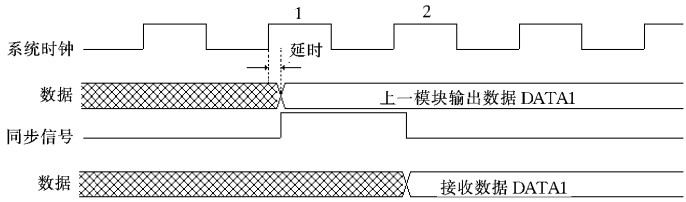


图 5－39　接口模块输出波形图

由于系统内存在反馈环路,因此只能采用串行处理结构,而不适于流水线结构。所谓串行处理结构,是指在时钟频率高于峰值数据通过率时,新的数据只有在上一个数据经过每一部分处理之后,才能进入系统进行处理的一种结构。而流水线结构则是所有数据像流水线上的工件一样,以一定的速率进入系统处理,在某一个时间点,在系统不同逻辑深度的部分正在处理着不同的数据。以下分模块介绍整个系统的结构。

2. 接口模块与误差分析

(1)误差分析

在进行数字信号处理系统的硬件设计之前,应考虑的第一个问题就是数据的精度问题。因为任何一个数字系统的有关参数以及每次运算过程中的结果,总是存储在有限字长的存储单元中,即只能以有限位数表示,从而使它们的精度受到限制。因此,在满足一定要求的范围内,确定适当的字长是十分必要的。

因有限字长的影响,数字系统中通常会产生以下几种误差:

①A/D 转换器将输入信号变为一组离散电平时产生的量化误差;

②用有限位二进制数表示数字信号处理系统的参数的不精确性而产生的量化误差;

③按照所需算法运算时,为限制位数的扩展而进行尾数处理,以及为防止溢出而压缩信号电平所产生的积累误差;

④溢出振荡引起的误差。

上述误差与系统的结构形式、数的表示方法、运算方法以及字长有关。选取足够的字长(位数)就可以减少误差的影响,但却增加了数字信号处理器的成本。

数字信号处理中有限字长影响首先和二进制数的运算方式有关。二进制数有原码、补码、反码三种形式;二进制运算分为定点制、浮点制和成组浮点制。本系统中采用定点补码二进制数的表示方式。这是因为在硬件实现时,补码表示法可以把加法和减法统一起来,从而降低设备的复杂程度。同时,采用定点制运算在速度和系统复杂程度上都优于浮点制运算,缺点是动态范围小,可能出现溢出。但本系统中各类数据采用不同的精度,并非统一的位数,并且根据实际情况中间变量宽度有所扩展,从而有效地弥补了这一缺陷。

系统中需要确定精度的数据主要有均衡器输入 xin(n)、均衡器抽头系数、均衡器输出 z(n)等。精度的确定可依据数字滤波器量化误差模型计算得出。但该法计算复杂,具体可参考文献[4]。这里介绍的方案采用软件模拟法确定。即在大量算法软件模拟的基础上得到数据的动态变化范围,据此确定数据的位数及精度,并通过软件模拟,以数据量化误差不足以影响到系统性能为目的,同时也考虑到系统硬件规模,综合确定数据的位数与精度。

在此基础上,确定均衡器抽头系数精度为 15 位、输入数据宽度 12 位、精度 10 位。选择宽度时也要考虑 12 位的 A/D 器件较为常用、易于选用的因素。

(2)接口模块

本系统中并未涉及具体的 A/D 器件,仅仅根据常用 A/D 器件的工作信号设计系统接口模块。

在转换完成后,一般 A/D 器件输出一个低电平信号作为 A/D 输出的允许信号(如 AD574)。该信号在系统中为输入信号 ad_end,接口模块中系统时钟始终监控 ad_end 的电平变化。当检测到 ad_end 的低电平时,接口模块产生一个“开始”脉冲作为允许信号,允许均衡器的延迟环节(桶形移位寄存器)接收输入数据,并开始移位。输出波形图见图 5-40。这部分比较简单,模块图从略。

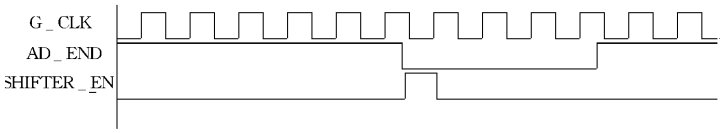


图 5-40 接口模块输出波形图

3. DFE 均衡器结构

DFE 均衡器模块实质是一数字滤波器,主要完成以下运算:

$$Z_n = \sum_{i=-M+1}^{0} C_i X_{n-i} - \sum b_i \hat{a}_{n-1} \tag{5-13}$$

上式中两部分分别为前馈滤波和反馈滤波。两部分结构相似,但节数不同,并且参加运算的数据精度不同。该模块主要由筒形移位寄存器和卷积模块组成,如图 5-41 所示。入口参数继承了系统参数中的 tap、pulse、x_wild、x_pre、main_coef_wild、coef_wid、coef_pre、z_wid、z_pre。

1)桶型移位寄存器 由 M * x_wid + N * an_wid 个 D 型触发器组成。

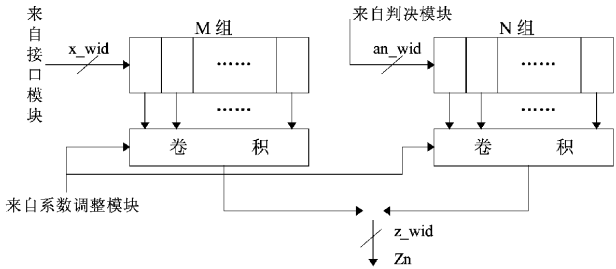


图 5－41　DFE 均衡器模块示意图

2)卷积部分的实现　卷积部分的运算与硬件实现的结构同 FIR 滤波器相同。这里要说明的是 LUT 属于 ROM 结构,而均衡算法中抽头系数需要不断调整。因此,需采用 RAM 结构实现均衡中的卷积算法,但其原理与查找表相同。若本系统完全采用并行结构,需 RAM 的容量为 $=2^7 * 16^* 12=24\text{kB}$,但是 FLEX 系列中最大容量的 EPF10K100 才有 24576 的 RAM,因此,全并行方案是不实际的。

本系统采取的是局部串行、全局并行的方案,即将输入 $x(n)$ 分为几组(如 n 组),每一组采用串行结构运算,而总体结构类似并行处理,每一组都同时处理数据。组的大小由模块入口参数 term 决定。假设 term 取 n,滤波器总节数为 $a * n$,则系统共需 RAM 容量为 $a * 2^n * width$。其中 $width$ 是抽头系数宽度。由于 n 是指数,所以 RAM 容量是随 n 值指数增加的。因此,term 取 3 较为合适。此方案也是在速度与规模之间权衡的结果。模块图见图 5－42。其中控制器利用状态机产生各种控制信号,精确控制各部分的同步运算。输入 $x(n)$ 精度为 10 位,抽头系数精度 15 位。考虑到相加溢出,均衡器输出 Z 的宽度为 18 位,一位符号位,15 位精度。

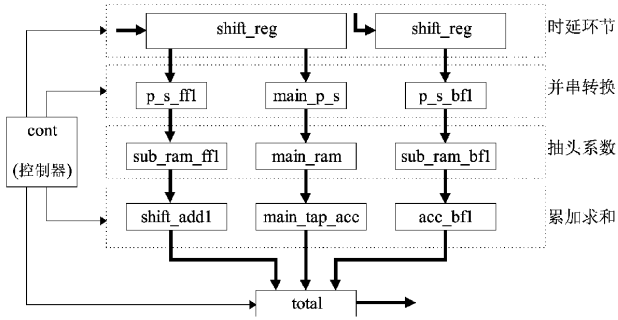


图 5－42　DFE 均衡器模块

4. 误差控制函数运算模块

误差控制函数运算模块主要完成判决输出以及误差控制函数的运算。

1)判决器　判决器判决部分比较简单。由于二进制数采用补码表示,判决以及误差的计算都可以用组合电路实现。

2)误差控制函数　误差控制函数如下式：

$$f(Z_n)=k_1\hat{e}_n+k_2|\hat{e}_n|e_n$$
$$=k_1(Z_n-\hat{a}_n)+k_2|Z_n-\hat{a}_n|(Z_n-r*\mathrm{Sgn}(Z_n)) \quad (5-14)$$

在两电平调制方式时，r 简化为 1，并将上式代入式(5－11)得：

$$C(k+1)=C(k)-\lambda f(Z_n)X(n)$$
$$=C(k)-\lambda X(n)[k_1(Z_n-\hat{a}_n)+k_2|Z_n-\hat{a}_n|(Z_n-r*\mathrm{Sgn}(Z))]$$
$$=C(k)-X(n)[\lambda k_1(Z_n-\hat{a}_n)+\lambda k_2|Z_n-\hat{a}_n|(Z_n-\mathrm{Sgn}(Z_n))]$$
$$=C(k)-X(n)[\lambda k_1(\varepsilon_{n1})+\lambda k_2(\varepsilon_{n2})] \quad (5-15)$$

上式中包含一个类似卷积的乘加运算形式，并且被乘数(即 $\lambda k1$ 和 $\lambda k2$)是定值。因此，可借助 DFE 均衡器模块中的卷积运算结构，用查找表实现式(5－15)的部分运算。模块如图 5－43 所示。

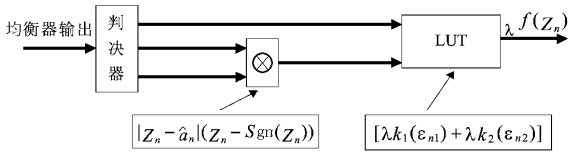


图 5－43　误差控制函数运算模块

5. 抽头系数调整模块(PART III)

本模块是系统中最重要的部分，完成均衡器抽头系数的自适应调整。如前所述，由于均衡器抽头系数在一个码元内要完成自适应调整，因此抽头系数不能以常量形式存放在查找表中，而只能作为中间变量存于可读写 RAM 块中。并且，各系数是以组合的形式存放，即一组系数存于 2^{term} 位 RAM 中。假设抽头系数为 $h1$、$h2$ 和 $h3$，调整量分别为 $\alpha1$、$\alpha2$ 和 $\alpha3$，$term$ 值为 3，表 5－2 显示了抽头系数的调整结果。

表 5－2　RAM 数据调整表

RAM 地址	RAM 内数据	调 整 后 数 据
0 0 0	0	0
0 0 1	$h3$	$h3+\alpha3$
0 1 0	$h2$	$h2+\alpha2$
0 1 1	$h2+h3$	$h2+h3+\alpha2+\alpha3$
1 0 0	$h1$	$h1+\alpha1$
1 0 1	$h1+h3$	$h1+h3+\alpha1+\alpha3$
1 1 0	$h1+h2$	$h1+h2+\alpha1+\alpha2$
1 1 1	$h1+h2+h3$	$h1+h2+h3+\alpha1+\alpha2+\alpha3$

因此，模块分为两大部分，一部分计算调整量 α，另一部分修正 RAM 内抽头系数。

(1)调整量的计算

计算式为

$$\alpha(n)=x(n)*\lambda f(Z_n) \quad (5-16)$$

其中 $\lambda f(Z_n)$ 的数据来自误差控制模块。要进行乘法的次数等于均衡器总节数，包括前馈和反馈节数。

(2)抽头系数的修正

根据分组(term)的结构，各组之间并行地对该组 RAM 进行操作。用每一组内部串行的计算调整量更新 RAM 内数据。

图 5－44 为第一组抽头系数的调整量运算部分的框图。

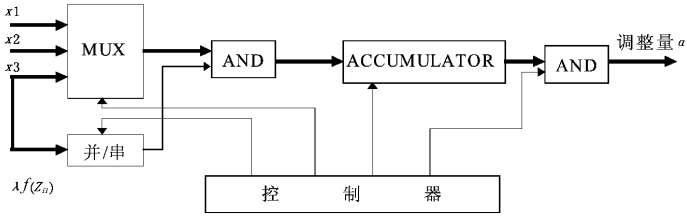


图 5－44　调整量运算模块

在图 5－45 中，一组输入($x1$，$x2$，$x3$)经过多路选择器串行与 $\lambda f(Z_n)$ 相乘，最后结果还要和控制器产生的‘规格化’信号相‘与’，得出一个抽头系数的调整量。抽头系数调整与调整量计算相差一个计算周期，也就是说在计算 $\alpha 2$ 的同时，调整 RAM 中涉及到 $h1$ 的部分。比如，$h1$ 的调整量为 $\alpha 1$。参照表 5－3，对于 $term = 3$ 的 RAM，在连续 8 个读写周期内，顺序写入 RAM 的数据为 0、0、0、0、$\alpha 1$、$\alpha 1$、$\alpha 1$、$\alpha 1$，其余类推。图 5－45 为该模块的部分输出波形图。

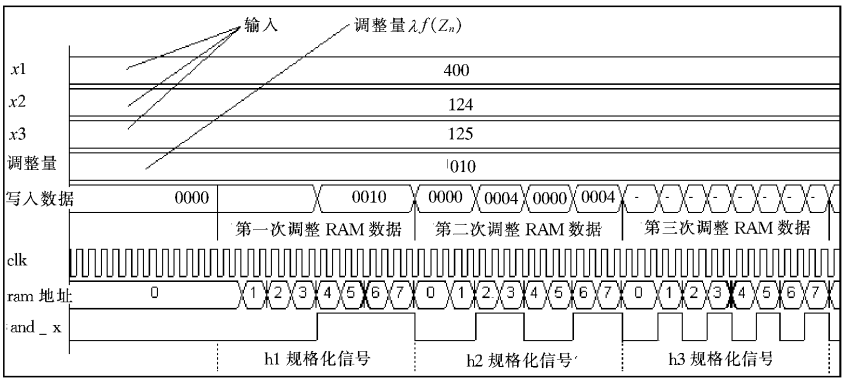


图 5－45　调整量运算模块输出波形图

图 5－45 中的 and_x 为‘规格化’信号。它与图 5－43 中累加器的输出相‘与’，作为写入 RAM 的数据，经过三次循环读写周期(每次循环读写周期含有 8 次读和 8 次写)。这样，抽头系数调整的结果就符合表 5－3 中的数据了。

在调整抽头系数时，必须严格遵守 RAM 的读写时序。图 5－46 为 FLEX10K 器件内置 RAM 的同步读写时序图。

图中具体时钟建立、延迟时间的解释见表 5－3。

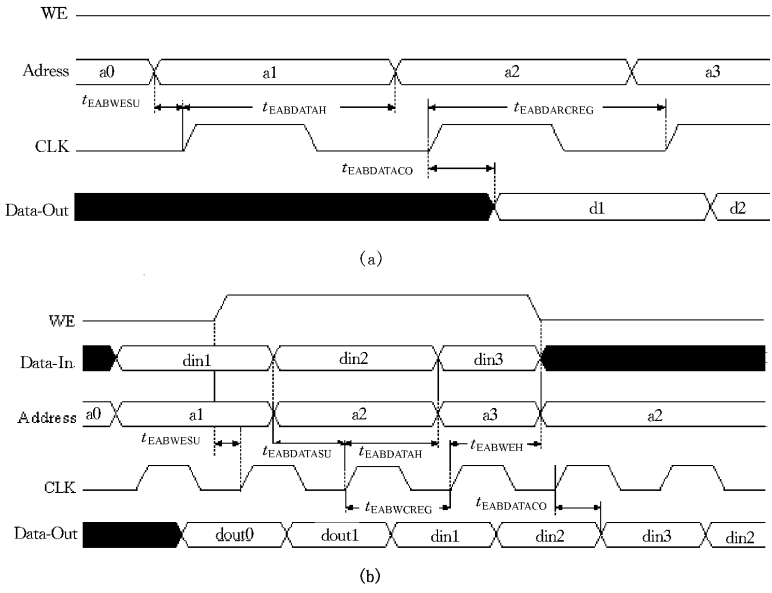


图 5－46　lpm_ram_dq 的同步读写时序图

(a)EAB 同步读时序 ;(b)EAB 同步写时序

表 5－3　EAB 时序宏参数

符号	参　　数	表 达 式
$t_{EABDATA2}$	**寄存器输入时,**EAB **数据或地址延迟时间**	
t_{EABWE2}	**寄存器输入时,**EAB **写使能信号延迟时间**	
t_{EABCLK}	EAB **寄存器时钟延迟时间**	
t_{EABCO}	EAB **寄存器时钟到输出延迟时间**	
t_{EABSU}	EAB **寄存器在时钟信号前的建立时间**	
t_{EABH}	EAB **寄存器在时钟信号后的保持时间**	
t_{AA}	**地址接入延迟**	
t_{DD}	**数据输入到数据输出的有效延迟**	
t_{EABOUT}	**数据输出延迟**	
$t_{EABRCREG}$	EAB **同步读周期时间**	$t_{EABCO}+t_{AA}+t_{EABSU}$
$t_{EABWCREG}$	EAB **同步写周期时间**	$t_{EABCO}+t_{DD}+t_{EABSU}$
$t_{EABDATACO}$	**当使用输出寄存器时,**EAB **时钟到输出的延迟时间**	$t_{EABCLK}+t_{EABCO}+t_{EABOUT}$
$t_{EABDATASU}$	**当使用输入寄存器时,在时钟信号到来之前,**EAB **地址或数据的建立时间**	$t_{EABSU}+t_{EABDATA2}-t_{EABCLK}$
$t_{EABDATAH}$	**当使用输入寄存器时,在时钟信号到来之后,**EAB **地址或数据的保持时间**	$t_{EABH}+t_{EABCLK}-t_{EABDATA2}$
$t_{EABWESU}$	**当使用输入寄存器时,在时钟信号到来之前,**EAB **中** WE **的建立时间**	$t_{EABSU}+t_{EABWE2}-t_{EABCLK}$
t_{EABWEH}	**当使用输入寄存器时,在时钟信号到来之后,** EAB **中** WE **的保持时间**	$t_{EABH}+t_{EABCLK}-t_{EABWE2}$

以 lpm_ram_dq 为底层文件，子模块 Filter_coef 根据前级输出的抽头系数调整量以及控制器产生的地址和读写控制信号完成均衡器抽头系数的自适应调整。模块入口参数为 addr_wid（地址线宽度）、coef_wid（系数宽度）、ini_file（RAM 初始化文件，确定 RAM 初始值），结构如图 5－47 所示。

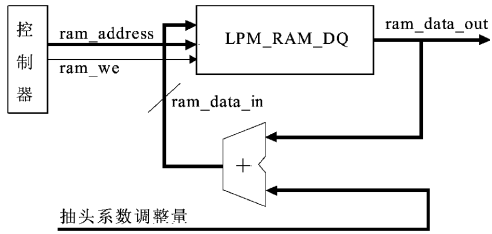


图 5－47　Filter_coef 模块图

6. 系统组成文件清单

我们所设计的盲均衡系统全部采用硬件描述语言 AHDL 完成，共有 17 个源文件（. tdf）。它们分别是：

①盲均衡器（blind_equalizer. tdf）；
②滤波器模块（filter. tdf）；
③滤波器_时延模块（filter_shift. tdf）；
④单通道并串转换模块（p_s. tdf ）；
⑤并串转换模块（s_term. tdf）；
⑥滤波器_系数模块（filter_coef. tdf）；
⑦移位相加模块（accumulator. tdf）；
⑧滤波器_相加模块（filter_add. tdf）；
⑨滤波器_控制模块（filter_con. tdf）；
⑩误差模块（err. tdf）；
⑪误差_判决模块（err_decision. tdf）；
⑫误差_系数模块（err_bk_rom）；
⑬误差_乘积模块（err_mult. tdf）；
⑭误差_控制模块（err_con. tdf）；
⑮调整模块（adjust. tdf）；
⑯调整_乘积模块（adjust_mult. tdf）；
⑰调整_控制模块（adjust_con. tdf）。

关于以上这些源程序的详细情况，读者可参考本书附录。

7. 硬件框图

为了使读者对整个系统硬件规模有直观的了解，特绘制三幅等效硬件框图。由于篇幅所限，仅绘出了一层模块图。

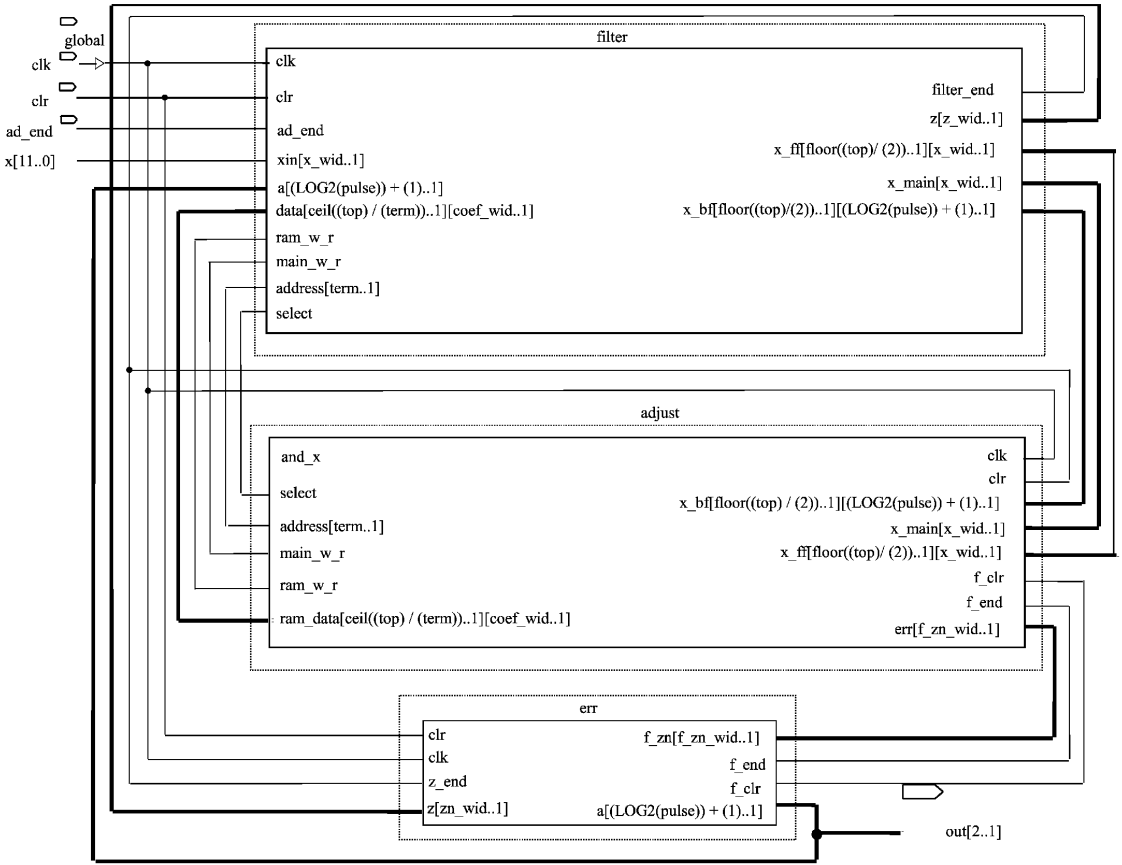


图 5－48　盲均衡器结构框图

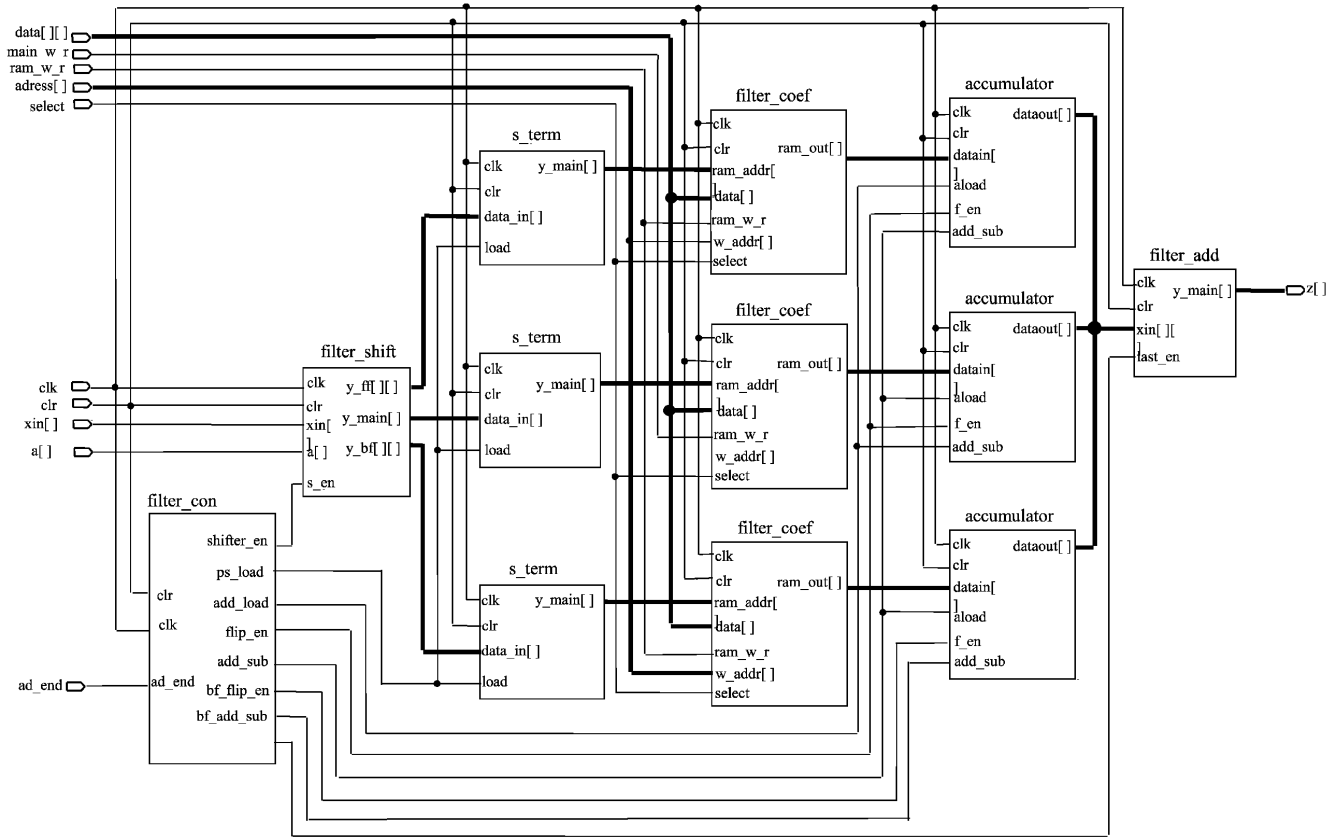


图 5－49　滤波器结构框图

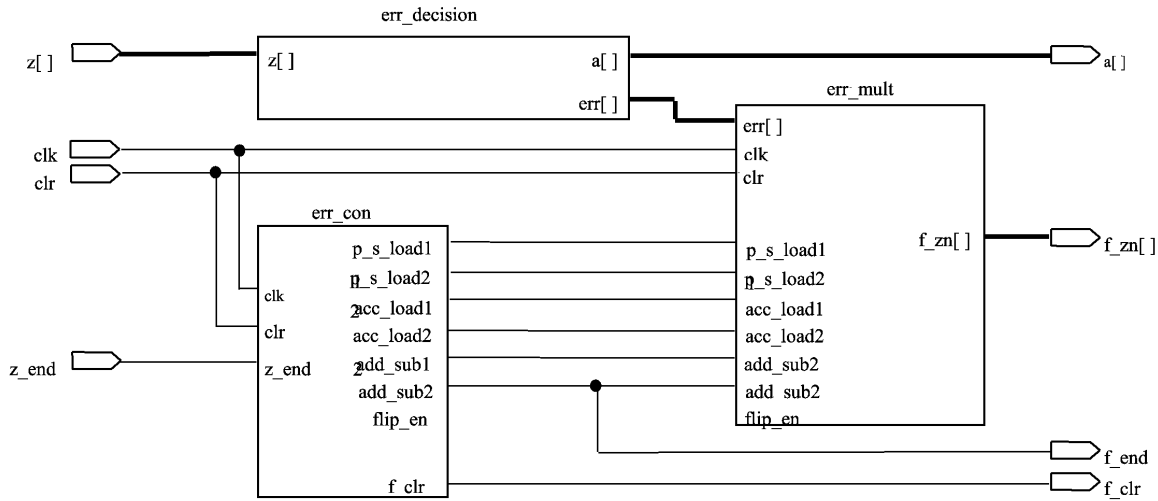


图 5－50　误差模块结构框图

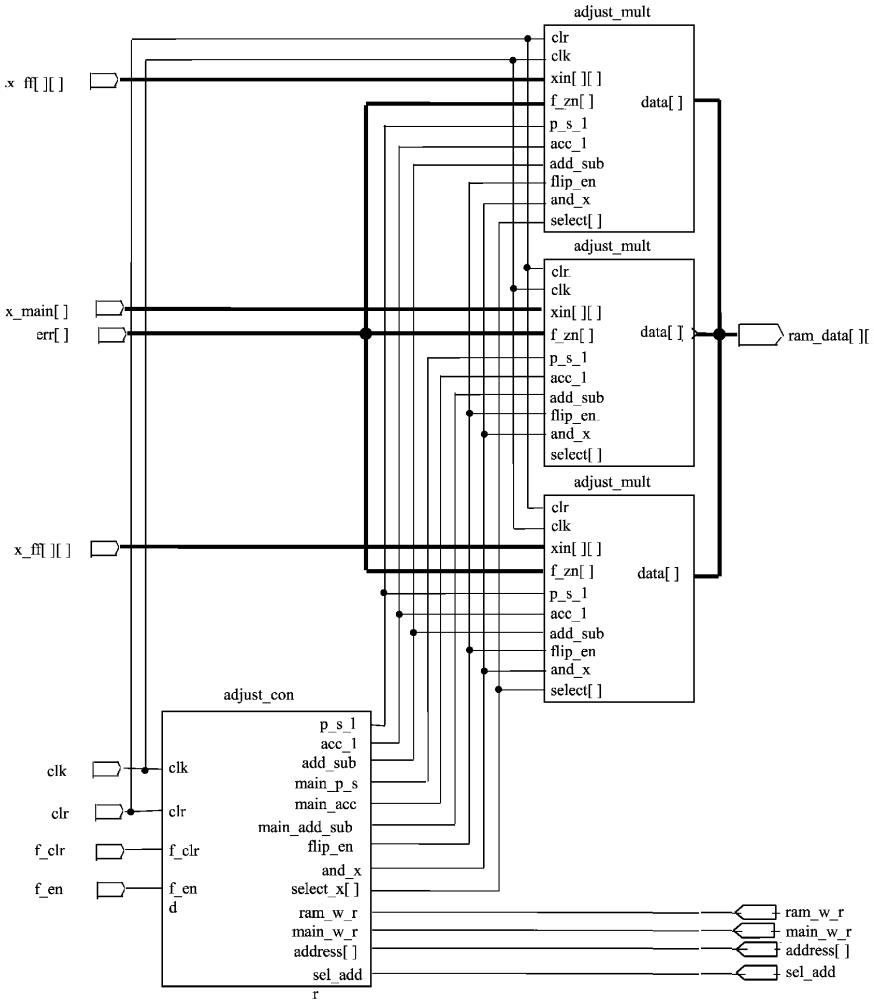


图 5－51　调整模块结构框图

附　录

本附录收录了与盲均衡器有关的 17 个源程序，供读者参考。

(1)盲均衡器(blind_equalizer.tdf**)**

```
INCLUDE "filter.inc";
INCLUDE "err.inc";
INCLUDE "adjust.inc";

PARAMETERS
(
tap = 7,
term = 3,
pulse = 2,              (电平数,实现两电平)
x_wid = 12,
x_pre = 10,             (x 数据宽度为 10 位，精度 10 位)
main_coef_wid =17,      (主抽头数据宽度为 17 位，精度 15 位)
coef_wid = 16,          (滤波器系数数据宽度为 16 位，精度 15 位)
coef_pre = 15,
z_wid = 18,             (输出结果数据宽度为 18 位，精度 15 位)
z_pre = coef_pre,
err_wid = 17,           (误差数据宽度为 17 位，精度 15 位)
err_pre = 15,
f_zn_wid = 9            (控制误差函数数据宽度为 9 位，精度 5 位)
);

CONSTANT an_wid = LOG2(pulse) + 1 ;(判决信号宽度,两电平时为 2 ;一位数
                                    据,一位符号位)
SUBDESIGN blind_equalizer
(
clk                     :INPUT ;
clr                     :INPUT = VCC;
ad_end                  :INPUT = GND;
xin[x_wid..1]           :INPUT;
out[an_wid..1]          :OUTPUT;
)
```

```
VARIABLE
part_i :filter WITH (tap = tap,
                     term = term,
                     pulse = pulse,
                     x_wid = x_wid,
                     x_pre = x_pre,
                     main_coef_wid = main_coef_wid,
                     coef_wid = coef_wid,
                     coef_pre = coef_pre,
                     z_wid = z_wid,
                     z_pre = z_pre);
part_ii :err WITH (  pulse = pulse,
                     zn_wid = z_wid,
                     zn_pre = z_pre,
                     err_wid = err_wid,
                     err_pre = err_pre,
                     f_zn_wid = f_zn_wid);
part_iii :adjust WITH (coef_wid = coef_wid,
                       coef_pre = coef_pre,
                       tap = tap,
                       term = term,
                       x_wid = x_wid,
                       x_pre = x_pre,
                       pulse = pulse,
                       f_zn_wid = f_zn_wid);
  g_clk :NODE;

BEGIN
    g_clk = global(clk);
    (连接滤波器的输入)
    part_i.clk = g_clk;
    part_i.clr = clr;
    part_i.ad_end = ad_end;
    part_i.xin[] = xin[];
    part_i.a[] = part_ii.a[];
    part_i.data[][] = part_iii.ram_data[][];

    part_i.ram_w_r = part_iii.ram_w_r;
    part_i.main_w_r = part_iii.main_w_r;
```

```
part_i.address[] = part_iii.address[];
part_i.select = part_iii.select;

(连接误差模块的输入)
part_ii.clk = g_clk;
part_ii.clr = clr;
part_ii.z_end = part_i.filter_end;
f_z[] = part_i.z[];
part_ii.z[] = part_i.z[];

(连接调整模块的输入)
part_iii.clk = g_clk;
part_iii.clr = clr;
part_iii.f_end = part_ii.f_end;
part_iii.f_clr = part_ii.f_clr;
part_iii.err[] = part_ii.f_zn[];
part_iii.x_ff[][] = part_i.x_ff[][];
part_iii.x_main[] = part_i.x_main[];
part_iii.x_bf[][] = part_i.x_bf[][];
(连接输出)
out[] = part_ii.a[];
END;
```

(2)滤波器模块(filter.tdf)

```
(本模块实现基本滤波器结构)
(完成卷积运算)
INCLUDE "filter_con";
INCLUDE "filter_shift";
INCLUDE "p_s";
INCLUDE "s_term";
INCLUDE "accumulator";
INCLUDE "filter_add";
INCLUDE "filter_coef";

PARAMETERS
(
tap = 7,
term = 3,
pulse = 2,
x_wid = 12,
```

```
x_pre = 10,                 (x 数据宽度为 12 位,精度为 10 位)
main_coef_wid =17,          (主抽头系数数据宽度为 17 位)
coef_wid = 16,              (滤波器系数数据宽度为 17 位,精度为 16 位)
coef_pre = 15,
z_wid = 18,                 (输出 z 数据宽度为 18 位,精度为 15 位)
z_pre = coef_pre
);

CONSTANT ram_num = ceil(tap div term);  (RAM 组数目,此处为 3)
CONSTANT ff_num =(ram_num - 1) div 2;   (前馈 RAM 组数目)
CONSTANT bf_num = ff_num;               (反馈 RAM 组数目)
CONSTANT an_wid = LOG2(pulse) + 1;      (判决信号宽度,两电平时为
                                         2;一位数据,一位符号)

CONSTANT x_num = floor((tap) / (2));    (sub_taps 的节数,应该是 term 的
                                         倍数,送 adjust;7 节时与 term
                                         相等)
SUBDESIGN filter
          (
clk             :INPUT ;
clr             :INPUT = VCC;
ad_end          :INPUT = VCC;       (A/D 转换完成信号)
xin[x_wid..1]   :INPUT;             (输入数据)
a[an_wid..1]    :INPUT = GND;       (判决数据,来自误差判决_模块(err_
                                     decision))
z[z_wid..1]     :OUTPUT;            (卷积结果)
filter_end      :OUTPUT;            (卷积部分完成信号,送下一级误差
                                     模块(err))
(以下输出送至 adjust_mult,以计算系数调整量)
x_ff[x_num..1][x_wid..1]        :OUTPUT;
x_main[x_wid..1]                :OUTPUT;
x_bf[x_num..1][an_wid..1]       :OUTPUT;

data[ram_num..1][coef_wid..1]  :INPUT;(各 ram 系数调整量,来自调整乘积_
                                       模块 adjust_mult,顺序为前馈抽头
                                       ff_taps、反馈抽头 bf_taps、主抽头
                                       main_tap)
(以下输入来自调整_控制模块(adjust_con),并且送到滤波器系数模块)
(filter_coef)调整系数
```

```
ram_w_r                 :INPUT;  (RAM 读写信号)
main_w_r                :INPUT;  (main_ram 的读写信号,由于
                                 main_ram 只需读写一次,因
                                 此与 ram_w_r 不同;且地址
                                 信号只为 1,即无须 address
                                 [ ]信号和 select)
address[term..1]        :INPUT;  (写 RAM 时的地址信号)
select                  :INPUT;  (地址选择信号,在 address[ ]和 s_term.ram
                                 _ad[ ]之间选择)
)
VARIABLE
cont :filter_con WITH (x_width = x_wid,
                       an_wid = an_wid);

shift_reg :filter_shift WITH (tap = tap,
                              x_wid = x_wid,
                              a_wid = an_wid,
                              term = term);

p_s_ff[ff_num..1] :s_term WITH (in_width = x_wid,
                                term_wid = term);

p_s_bf[bf_num..1] :s_term WITH (in_width = an_wid,
                                term_wid = term);

--前馈和反馈抽头 RAM 初始值为 0
sub_ram_ff[ff_num..1] :filter_coef WITH (addr_wid = term,
                                         coef_wid = coef_wid,
                                         ini_file = "zero.mif");
sub_ram_bf[bf_num..1] :filter_coef WITH (addr_wid = term,
                                         coef_wid = coef_wid,
                                         ini_file = "zero.mif");

(主抽头 RAM 初始值为 1 或 1.4 等等)
        main_ram :filter_coef WITH (addr_wid = 1,
                                    coef_wid = main_coef_wid,
                                    ini_file = "VCC.mif");

(主抽头的并串转换只有一个通道)
```

```
main_p_s :s_term WITH ( in_width = x_wid,
                        term_wid = 1 );

(前馈部分副抽头系数 RAM 的移位相加)
shift_add[ff_num..1] :accumulator WITH ( x_width = coef_wid,
                                         x_pre = coef_pre,
                                         y_width = x_wid,
                                         y_pre = x_pre,1
                                         out_int = z_wid
                                         out_pre = z_pre,
                                         pipeline = "YES");

(反馈部分抽头系数 RAM 的移位相加)
acc_bf[bf_num..1] :accumulator WITH (x_width = coef_wid,
                                     x_pre = coef_pre,
                                     y_width = an_wid,
                                     y_pre = 0,
                                     out_int = z_wid - z_pre,
                                     out_pre = z_pre,
                                     pipeline = "YES");

(前馈部分主抽头系数 RAM 的移位相加)
main_tap_acc :accumulator WITH ( x_width = main_coef_wid,
x_pre = coef_pre,
                                 y_width = x_wid,
                                 y_pre = x_pre,
                                 out_int = z_wid - z_pre,
                                 out_pre = z_pre,
                                 pipeline = "YES");

total :filter_add WITH ( tap = tap,
                         term = term,
                         width = z_wid,
                         pipeline = "YES");

filter_z[z_wid..1] :NODE;

BEGIN
(连接时钟和清除信号)
```

```
cont.clk = clk;
shift_reg.clk = clk;
p_s_ff[].clk = clk;
p_s_bf[].clk = clk;
shift_add[].clk = clk;
main_tap_acc.clk = clk;
acc_bf[].clk = clk;
sub_ram_ff[].clk = clk;
sub_ram_bf[].clk = clk;
main_ram.clk = clk;
main_p_s.clk = clk;
total.clk = clk;

cont.clr = clr;
shift_reg.clr = clr;
p_s_ff[].clr = clr;
p_s_bf[].clr = clr;
shift_add[].clr = clr;
main_tap_acc.clr = clr;

acc_bf[].clr = clr;
sub_ram_ff[].clr = clr;
sub_ram_bf[].clr = clr;
main_ram.clr = clr;
main_p_s.clr = clr;
total.clr = clr;

(连接滤波器_控制模块(filter_con)的输入)
cont.ad_end = ad_end;

(连接时延模块(shift_reg)的输入)
shift_reg.xin[] = xin[];
shift_reg.a[] = a[];
shift_reg.s_en = cont.shifter_en;

(连接前馈环节并串转换(s_term)的输入)
p_s_ff[].load = cont.ps_load;
FOR i IN 1 TO ff_num GENERATE
   FOR j IN 1 TO term GENERATE
```

```
      p_s_ff[i].data_in[j][] = shift_reg.y_ff[(i - 1) * term + j][];
    END GENERATE;
END GENERATE;
```

（连接反馈环节并串转换(s_term)的输入）

```
p_s_bf[].load = cont.ps_load;
FOR i IN 1 TO bf_num GENERATE
 FOR j IN 1 TO term GENERATE
   p_s_bf[i].data_in[j][] = shift_reg.y_bf[(i - 1) * term + j][];
 END GENERATE;
END GENERATE;
```

（连接主抽头的输入）

```
main_p_s.data_in[1][] = shift_reg.y_main[];
main_p_s.load = cont.ps_load;
```

（连接前馈环节 RAM **的输入，即前馈环节的系数调整量）**

```
FOR i IN 1 TO ff_num GENERATE
sub_ram_ff[i].data[] = data[i][];
END GENERATE;
sub_ram_ff[].ram_w_r = ram_w_r;
sub_ram_ff[].w_addr[] = address[];
sub_ram_ff[].select = select;
FOR i IN 1 TO ff_num GENERATE
sub_ram_ff[i].ram_addr[] = p_s_ff[i].ram_ad[];
END GENERATE;
```

（连接反馈环节 RAM **的输入，即反馈环节的系数调整量）**

```
FOR i IN 1 TO bf_num GENERATE
sub_ram_bf[i].data[] = data[i + bf_num][];
END GENERATE;
sub_ram_bf[].ram_w_r = ram_w_r;
sub_ram_bf[].w_addr[] = address[];
sub_ram_bf[].select = select;
FOR i IN 1 TO bf_num GENERATE
sub_ram_bf[i].ram_addr[] = p_s_bf[i].ram_ad[];
END GENERATE;
```

（连接主抽头环节 RAM **的输入）**

（对于主抽头环节 RAM **只需要** ram_addr **和** select **，**w_addr **总是** 1**）**

```
main_ram.data[17] = data[ram_num][16];
main_ram.data[16..1] = data[ram_num][];
main_ram.ram_w_r = main_w_r;
main_ram.ram_addr[] = main_p_s.ram_ad[];

main_ram.select = select;
```

(分别连接前馈环节 RAM、反馈环节 RAM 和主抽头 RAM 移位相加所需的输入信号)

(前馈环节 RAM)

```
shift_add[].a_load = cont.add_load;
shift_add[].add_sub = cont.add_sub;
shift_add[].f_en = cont.flip_en;
FOR i IN 1 TO ff_num GENERATE
shift_add[i].datain[] = sub_ram_ff[i].ram_out[];
END GENERATE;
```

(主抽头 RAM)

```
main_tap_acc.add_sub = cont.add_sub;
main_tap_acc.f_en = cont.flip_en;
main_tap_acc.a_load = cont.add_load;
main_tap_acc.datain[] = main_ram.ram_out[];
```

(反馈环节 RAM)

```
acc_bf[].add_sub = cont.bf_add_sub;
acc_bf[].f_en = cont.bf_flip_en;
acc_bf[].a_load = cont.add_load;
FOR i IN 1 TO bf_num GENERATE
acc_bf[i].datain[] = sub_ram_bf[i].ram_out[];
END GENERATE;
```

(连接地址输入)

```
total.last_en = cont.adds_en;
```

(前馈及主抽头值,来自移位相加模块(accumulator))

```
FOR i IN 1 TO ff_num GENERATE
total.xin[i][] = shift_add[i].dataout[];
END GENERATE;
```

```
total.xin[ff_num + 1][] = main_tap_acc.dataout[];

(反馈值,来自移位相加模块(accumulator))
FOR i IN 1 TO bf_num GENERATE
total.xin[i + ff_num + 1][] = acc_bf[i].dataout[];
END GENERATE;

(产生送给调整乘积模块的移位数据)
x_ff[][] = shift_reg.y_ff[][];
x_main[] = shift_reg.y_main[];
x_bf[][] = shift_reg.y_bf[][];

(产生最后的结果(Z[])和结束信号(filter_end))
filter_z[] = total.add_out[];
z[] = filter_z[];
filter_end = cont.adds_en;
END;
```

(3)滤波器时延模块(filter_shift)

```
PARAMETERS
(
tap = 7,
x_wid = 12,
a_wid = 2,
term = 3
);

CONSTANT  ff_tap = ceil( tap div 2 ); (前馈节数为 4)
CONSTANT  bf_tap = floor( tap div 2 );(反馈节数为 3)
CONSTANT  main_tap = ceil( tap div 2 );

SUBDESIGN filter_shift
(
clk                          :INPUT;
clr                          :INPUT = VCC;
xin[x_wid..1]                :INPUT;
a[a_wid..1]                  :INPUT = GND;
s_en                         :INPUT = GND;(来自滤波器_控制模块的
                                          移)
```

```
y_ff[ff_tap - 1..1][x_wid..1] :OUTPUT;    (前馈输出)
y_main[x_wid..1]              :OUTPUT;    (主抽头输出)
y_bf[bf_tap..1][a_wid..1]     :OUTPUT;    (反馈输出)
)

VARIABLE
shifter_ff[ff_tap..1][x_wid..1]     :DFFE;
shifter_bf[bf_tap..1][a_wid..1]     :DFFE;

BEGIN
(连接移位寄存器的输入)
(前馈环节输入)
shifter_ff[][].clrn = clr;
shifter_ff[][].clk = clk;
shifter_ff[][].ena = s_en;
shifter_ff[1][].d = xin[];
FOR i IN 2 TO ff_tap GENERATE
shifter_ff[i][].d = shifter_ff[Ii - 1][].q;
END GENERATE;
(反馈环节输入)
shifter_bf[][].clrn = clr;
shifter_bf[][].clk = clk;
shifter_bf[][].ena = s_en;
shifter_bf[1][].d = a[];
FOR i IN 2 TO bf_tap GENERATE
shifter_bf[i][].d = shifter_bf[i - 1][].q;
END GENERATE;
(连接输出)
(前馈环节输出)
y_main[] = shifter_ff[main_tap][].q;
FOR i IN 1 TO ff_tap - 1 GENERATE
y_ff[i][] = shifter_ff[i][].q;
END GENERATE;
(反馈环节输出)
FOR i IN 1 TO bf_tap GENERATE
y_bf[i][] = shifter_bf[i][].q;
END GENERATE;
END;
```

(4)单通道并串转换模块(p_s. tdf **)**

```
PARAMETERS
(
width = 12                                    (并入宽度)
);

SUBDESIGN p_s
(
parallel_in[width..1]   :INPUT;        (并入)
serial_out              :OUTPUT;       (串出)
clk                     :INPUT;
clr                     :INPUT = VCC;
load_n                  :INPUT = GND(装载)
)

VARIABLE
reg[width..1]           :DFF;

BEGIN
-- 给所有的触发器连接时钟和清除信号
reg[ ].clk = clk;
reg[ ].clrn = clr;

IF (load_n) THEN      (装载)
FOR j in 1 TO width GENERATE
reg[j].d = parallel_in[j];
END GENERATE;    (移位)
ELSE
FOR j in 1 TO width - 1 GENERATE
reg[j].d = reg[j + 1].q;
END GENERATE;
reg[width].d = GND;
END IF;
serial_out = reg[1].q;
END;
```

(5)并串转换模块(s_term. tdf**)**

```
INCLUDE "p_s";
```

```
PARAMETERS
(
in_width = 12,                          (进行转换的并行数据宽度)
term_wid = 3                            (进行转换的并行数据通道数)
);

SUBDESIGN s_term
(
clk                                              :INPUT;
clr                                              :INPUT = VCC;
data_in[term_wid..1][in_width..1]                :INPUT;
load                                             :INPUT = GND;
ram_ad[term_wid..1]                              :OUTPUT;
)

VARIABLE
regs[term_wid..1] :p_s WITH (width = in_width);

BEGIN
regs[].clk = clk;
regs[].clr = clr;
regs[].load_n = load;
FOR i IN 1 TO term_wid GENERATE
regs[i].parallel_in[] = data_in[i][];
END GENERATE;
ram_ad[] = regs[].serial_out;
END;
```

(6)滤波器_系数模块(filter_coef.tdf)

```
INCLUDE "lpm_add_sub";
INCLUDE "lpm_ram_dq";
INCLUDE "busmux";

PARAMETERS
(
addr_wid =3,
coef_wid = 16,
ini_file = "coef.mif"
```

```
);

SUBDESIGN filter_coef
(
clk                              :INPUT;
clr                              :INPUT = VCC;
ram_w_r                          :INPUT = VCC;(来自调整_控制模块(adjust_
                                                con))
ram_addr[addr_wid..1]  :INPUT = GND;(来自并串转换模块(p_s))
w_addr[addr_wid..1]              :INPUT = GND;(来自调整_控制模块(adjust_
                                                con))
data[coef_wid..1]                :INPUT = GND;(来自调整_乘积模块(adjust_
                                                mult))
select                           :INPUT = GND;(来自调整_控制模块(adjust_
                                                con))
ram_out[coef_wid..1]             :OUTPUT;
addre[addr_wid..1]               :OUTPUT;
)

VARIABLE
(若数据宽度为 1,则为 main_ram,无需 busmux,只需一个或门即可)
IF addr_wid > 1 GENERATE
addr_mux :busmux WITH (WIDTH = addr_wid);
END GENERATE;
myram :lpm_ram_dq WITH ( LPM_WIDTH = coef_wid,
                         LPM_WIDTHAD = addr_wid,
                         LPM_NUMWORDS = 2^addr_wid,
                         LPM_FILE = ini_file,
                         LPM_OUTDATA = "UNREGISTERED");

adder :lpm_add_sub WITH (LPM_WIDTH = coef_wid,
                         LPM_DIRECTION = "SUB",
                         LPM_PIPELINE = 1);
res[coef_wid..1]       :NODE;
address[addr_wid..1] :NODE;
BEGIN
(addr_mux 选择地址线)
IF addr_wid > 1 GENERATE
addr_mux.dataa[] = ram_addr[];
```

```
addr_mux.datab[] = w_addr[];
addr_mux.sel = select;
address[] = addr_mux.result[];
ELSE GENERATE
address[] = select # ram_addr[];(因为 main_ram 地址线只有一位,修改系数
                                 时只需写地址 1 即可)
END GENERATE;
addre[] = address[];

myram.we = ram_w_r;
myram.inclock = clk;
myram.address[] = address[];
res[] = myram.q[];

(在 CLK 的下降沿相加)
adder.clock = !clk;
adder.aclr = !clr;
adder.datab[] = data[];
adder.dataa[] = myram.q[];
myram.data[] = adder.result[];
ram_out[] = res[];
END;
```

(7)移位相加模块(accumulator.tdf)

```
INCLUDE "lpm_add_sub";

PARAMETERS
(
x_width =16,        (乘数(即要进行累加的数)之宽度)
x_pre = 15,         (乘数之精度)
y_width = 12,       (被乘数(即决定累加次数的数)之宽度)
y_pre = 10,         (被乘数之精度)
out_int = 3,        (乘积的整数部分位数)
out_pre = 15,       (乘积的小数部分位数)
pipeline = "YES"
);

CONSTANT dff_num = (x_width - x_pre) + (y_width - y_pre) + out_pre;
```

```
CONSTANT out_width = out_int + out_pre ;

SUBDESIGN accumulator
(
clk                    :INPUT;
clr                    :INPUT = VCC;
datain[x_width..1]     :INPUT;
add_sub                :INPUT = VCC;（加减控制）
f_en                   :INPUT = GND;（输出使能信号）
a_load                 :INPUT = GND;（第一个串行数据信号）
dataout[out_width..1]  :OUTPUT;
)
VARIABLE
adder :lpm_add_sub WITH (lpm_width = x_width + 1,
                         lpm_representation = "UNSIGNED");

accum[dff_num..1]   :DFF;

IF(pipeline == "YES") GENERATE
flip[out_width..1] :DFFE;
END GENERATE;

adder_dataa[x_width + 1..1] :NODE;
adder_datab[x_width + 1..1] :NODE;
adder_result[x_width + 1..1] :NODE;

BEGIN
accum[].clk = clk;
accum[].clrn = clr;
CASE a_load IS
  WHEN VCC =>
    accum[dff_num] = datain[x_width];
  accum[dff_num - 1..dff_num - x_width] = datain[];
IF (dff_num - x_width - 1 > 0) GENERATE
  accum[dff_num - x_width - 1..1] = GND;
END GENERATE;
  WHEN GND =>
    accum[dff_num..dff_num - x_width].d = adder_result[];
    accum[dff_num - x_width - 1..1].d = accum[dff_num - x_width..2];
```

```
    END CASE;

    adder.add_sub = add_sub;

    adder_datab[] = (datain[x_width], datain[]);(扩展符号位)
    adder_dataa[x_width + 1] = accum[dff_num];
    adder_dataa[x_width..1] = accum[dff_num..dff_num - x_width + 1];
    adder.dataa[] = adder_dataa[];
    adder.datab[] = adder_datab[];
    adder_result[] = adder_result[];

    IF(pipeline == "YES") GENERATE
    flip[].clk = clk;
    flip[].clrn = clr;
    flip[].ena = f_en;
    flip[].d = accum[out_width..1].q;(需保证 dff_num 大于、等于 out_width)
    dataout[] = flip[].q;
    ELSE GENERATE
    dataout[] = accum[out_width..1].q;
    END GENERATE;
    END;
```

(8)滤波器_相加模块(filter_add.tdf)

```
    SUBDESIGN filter_add
    (
    clk                                 :INPUT;
    clr                                 :INPUT = VCC;
    last_en                             :INPUT = GND;(输出使能信号)
    xin[input_num..1][width..1]         :INPUT;
    add_out[width..1]                   :OUTPUT;
    )

    VARIABLE
    addtree[adder_num..1] :lpm_add_sub WITH (lpm_width = width,
                                             lpm_representation = "signed",
                                             lpm_direction = "add");
    IF (pipeline == "YES") GENERATE
    flip[width..1] :DFFE;
```

```
END GENERATE;

BEGIN
IF (pipeline == "YES") GENERATE
flip[].clk = clk;
flip[].clrn = clr;
flip[].ena = last_en;
END GENERATE;

addtree[1].dataa[] = xin[1][];
addtree[1].datab[] = xin[2][];
FOR i IN 2 TO adder_num  GENERATE
addtree[i].dataa[] = addtree[i - 1].result[];
addtree[i].datab[] = xin[i + 1][];
END GENERATE;
IF (pipeline == "YES") GENERATE
flip[].d = addtree[adder_num].result[];
add_out[] = flip[].q;
ELSE GENERATE
add_out[] = addtree[adder_num].result[];
END GENERATE;
END;
```

(9)滤波器_控制模块(filter_con)

```
PARAMETERS
(
x_width = 12,(输入数据的宽度,将决定 add_sub 和 flip_en 在何时起作用)
an_wid = 2
);

CONSTANT dff_num = x_width + 6;

SUBDESIGN filter_con
(
clk          :INPUT ;
clr          :INPUT = VCC;
ad_end       :INPUT ;(A/D 转换结束信号)
shifter_en   :OUTPUT ;(移位寄存器使能信号)
ps_load      :OUTPUT ;(并行到串行转换的装入信号)
```

```
add_load     :OUTPUT ;(移位相加的装入信号)
add_sub      :OUTPUT ;(加减控制信号)
flip_en      :OUTPUT ;(滤波结果输出信号)
adds_en      :OUTPUT ;(得到最后结果(z[ ])相加模块的使能信号)
bf_add_sub   :OUTPUT ;(反馈环节 RAM 移位相加加减控制信号)
bf_flip_en   :OUTPUT ;(反馈环节 RAMa 移位相加结束输出信号)
)

VARIABLE
shift[dff_num..1] :DFF ;
base              :NODE ;

BEGIN
shift[ ].clk = clk ;
shift[ ].clrn = clr ;

(捕捉 A/D 转换结束信号,并且产生移位寄存器使能信号)
shift[1].d = ad_end ;
shift[2].d = shift[1].q ;
base = !shift[1].q & shift[2].q ;
shifter_en = base ;

(产生其他控制信号)
shift[3].d = base ;

FOR i IN 4 TO dff_num GENERATE -
shift[i].d = shift[i - 1].q ;
END GENERATE ;

ps_load = shift[3].q ;
(第五个时钟读取 RAM)
add_load = shift[5].q ;
add_sub = !shift[5 + x_width - 1].q ;
flip_en = shift[5 + x_width].q ;
adds_en = shift[6 + x_width].q ;

bf_add_sub = !shift[5 + an_wid - 1].q ;
bf_flip_en = shift[5 + an_wid].q ;
END ;
```

(10)误差模块(err.tdf)

```
INCLUDE "p_s";
INCLUDE "s_term";
INCLUDE "accumulator";
INCLUDE "err_bk_rom";
INCLUDE "err_mult";
INCLUDE "err_decision";
INCLUDE "err_con"

PARAMETERS
(
pulse = 2,      (电平数,决定判决值宽度)
zn_wid = 18,
zn_pre = 15,    (卷积值数据宽度为 18 位,精度为 15 位)
err_wid = 17,
err_pre = 15,   (误差数据宽度为 17 位,精度为 15 位)
f_zn_wid = 9    (控制误差函数值数据宽度为 9 位)
);

CONSTANT an_wid = LOG2(pulse) + 1 ;(判决信号宽度,两电平时为 2)

SUBDESIGN err
(
clk                    :INPUT ;
clr                    :INPUT = VCC;
z_end                  :INPUT = GND;
z[zn_wid..1]           :INPUT ;(来自滤波器模块的输出(18 位))
f_zn[f_zn_wid..1]      :OUTPUT ;(最后的结果 (b * k1 * err + b * k2 * err *
                                 err))
a[an_wid..1]           :OUTPUT ;(反馈判决)
f_end                  :OUTPUT ;(计算结束信号)
f_clr                  :OUTPUT ;(每次调整都要先清零!)
)

VARIABLE
cont :err_con WITH (err_wid = err_wid);
decision :err_decision WITH (pulse = pulse,
```

```
                                    z_wid = zn_wid,
                                    z_pre = zn_pre,
                                    err_wid = err_wid,
                                    err_pre = err_pre);
    mult :err_mult WITH (err_wid = err_wid,
                         err_pre = err_pre);
    f_err[f_zn_wid..1] :NODE; (用于调试)

    BEGIN
    (连接误差_判决模块输入)
    decision.z[] = z[];

    (连接误差_控制模块输入)
    cont.clk = clk;
    cont.clr = clr;
    cont.z_end = z_end;

    (连接误差_乘积模块输入)
    mult.clk = clk;
    mult.clr = clr;
    mult.p_s_load1 = cont.p_s_load1;
    mult.p_s_load2 = cont.p_s_load2;
    mult.acc_load1 = cont.acc_load1;
    mult.acc_load2 = cont.acc_load2;
    mult.add_sub1 = cont.add_sub1;
    mult.add_sub2 = cont.add_sub2;
    mult.flip_en = cont.flip_en;
    mult.err[] = decision.err[];

    (连接输出信号)
    f_clr = cont.f_clr;
    f_end = cont.flip_en;
    a[] = decision.a[];
    f_err[] = mult.f_zn[];
    f_zn[] = f_err[];
    END;
```

(11)误差_判决模块(err_decision.tdf)

```
    PARAMETERS
```

```
(
pulse = 2,
z_wid = 18,
z_pre = 15,
err_wid = 17,
err_pre = 15
);

CONSTANT an_wid = LOG2(pulse) + 1 ;(判决信号宽度,两电平时为 2)
CONSTANT z_int = z_wid - z_pre;
CONSTANT err_int = err_wid - err_pre;

SUBDESIGN err_decision
(
z[z_wid..1]                                    :INPUT;
err[err_wid..1]                          :OUTPUT;
a[an_wid..1]                       :OUTPUT;
)

VARIABLE
decision[err_int..1] :NODE;
zn_int[z_int..1]      :NODE;
z_float[z_pre..1]     :NODE;

BEGIN
zn_int[] = z[z_wid..z_pre + 1] ;(zn 的整数部分作为判决表 TABLE 的输入)
z_float[] = z[z_pre..1] ;(zn 的小数部分不动,作为误差的小数部分)
(判决表)
TABLE
zn_int[]          =>          decision[];
0                 =>           3;
1                 =>           0;
2                 =>           1;
3                 =>           2;
4                 =>           1;
5                 =>           2;
6                 =>           3;
7          =>                  0;
END TABLE;
```

```
err[err_wid..err_pre + 1] = decision[2..1];
err[err_pre..1] = z_float[z_pre..z_pre - err_pre + 1];
a[2] = zn_int[z_wid - z_pre];
a[1] = VCC;
END;
```

(12)**误差_系数模块**(err_bk_rom.tdf)

```
PARAMETERS
(
bk1 = 14,--01,0100
bk2 = 4   --00,0100
);

(rom_data of b*k1 and b*k2)
(0.0006 =0.000,0000,0001,0100)
(0.00012=0.000,0000,0000,0100)
(由于 b*k1 和 b*k2 均为已知小数,因此可缩短 ROM 宽度,只取最后 5 位即
 可)
(运算完毕后再移小数点!!!)
(rom 内部图:)
(----------------代表--------实际数)
(    0                 -->0          --000000)
(   b*k2               -->0.00012  --000100)
(   b*k1               -->0.0006    --010100)
( b*(k1+k2)            -->0.00072  --011000)
----------------
CONSTANT bk_wid = 6 ;(取最后 5 位加符号位)

SUBDESIGN err_bk_rom
(
data_in[2..1]              :INPUT;
rom_out[bk_wid..1]         :OUTPUT;
)

BEGIN
rom_out[1] = gnd;
rom_out[2] = gnd;
rom_out[3] = data_in[1] $ data_in[2];
rom_out[4] = data_in[1] & data_in[2];
```

```
rom_out[5] = data_in[2];
rom_out[6] = gnd;
END;
```

(13)误差_乘积模块(err_mult.tdf)

```
INCLUDE "p_s";
INCLUDE "s_term";
INCLUDE "accumulator";
INCLUDE "err_bk_rom";

PARAMETERS
(
err_wid = 17,                    (来自误差判_决模块(err_decision))
err_pre = 15                     (来自误差判_决模块(err_decision))
);

CONSTANT err_pow_wid = 18;(误差平方的宽度)
CONSTANT err_pow_pre = 15;(误差平方的精度)
CONSTANT shift_bits = 10;   (小数点预先右移 10 位)
CONSTANT bk_wid = 6;        (取最后 5 位加符号位)
CONSTANT bk_pre = 5;
CONSTANT zn_wid = bk_wid + err_pow_wid - err_pow_pre;
CONSTANT zn_pre = bk_pre;

SUBDESIGN err_mult
(
clk                              INPUT ;
clr                              :INPUT = VCC;
p_s_load1, p_s_load2             :INPUT = GND;
acc_load1, acc_load2             :INPUT = GND;
add_sub1, add_sub2               :INPUT = VCC;
flip_en                          :INPUT = GND;
err[err_wid..1]                  :INPUT ;(来自判决模块(err_ decision))
f_zn[zn_wid..1]                  :OUTPUT;
)

VARIABLE
p_s1        :s_term WITH (in_width = err_wid,
            term_wid = 1);
```

```
accumul1 :accumulator WITH (x_width = err_wid,
                            x_pre = err_pre,
                            y_width = err_wid,
                            y_pre = err_pre,
                            out_int = err_pow_wid - err_pow_pre,
                            out_pre = err_pow_pre,
                            pipeline = "NO");

accumul2 :accumulator WITH (x_width = bk_wid,
                            x_pre = bk_pre,
                            y_width = err_pow_wid,
                            y_pre = err_pow_pre,
                            out_int = zn_wid - zn_pre,
                            out_pre = bk_pre,
                            pipeline = "YES");

bk_rom :err_bk_rom WITH (bk1 = 14,
                         bk2 = 4);
p_s2 :s_term WITH ( in_width = err_pow_wid,
                    term_wid = 2);

accu1_in[err_wid..1] :NODE;

BEGIN
(连接第一次并串转换的输入)
p_s1.clk = clk;
p_s1.clr = clr;
P_s1.load = p_s_load1;
p_s1.data_in[][] = err[];

(连接第一次移位相加的输入)
accu1_in[] = p_s1.ram_ad[] & err[];
accumul1.datain[] = accu1_in[];
accumul1.clk = clk;
accumul1.clr = clr;
accumul1.a_load = acc_load1;
accumul1.add_sub = add_sub1;

(连接第一次并串转换的输入)
```

```
p_s2.clk = clk;
p_s2.clr = clr;
p_s2.load = p_s_load2;
p_s2.data_in[1][] = accumul1.dataout[];
p_s2.data_in[2][err_pow_wid] = err[err_wid];
p_s2.data_in[2][err_pow_wid - 1..1] = err[];    (扩展误差的符号位)

(连接 bk_rom 的输入)
bk_rom.data_in[] = p_s2.ram_ad[];
(连接第二次移位相加的输入)
accumul2.datain[] = bk_rom.rom_out[];
accumul2.clk = clk;
accumul2.clr = clr;
accumul2.a_load = acc_load2;
accumul2.add_sub = add_sub2;
accumul2.f_en = flip_en;
f_zn[] = accumul2.dataout[];
END;
```

(14)误差_控制(err_con.tdf)

```
PARAMETERS
  (
  err_wid = 17              (误差数据宽度为 17 位,精度为 15 位)
  ))

  CONSTANT  dff_num = 2 * err_wid + 4;

  SUBDESIGN err_con
  (
  clk          :INPUT ;
  clr          :INPUT = VCC;
  z_end        :INPUT = GND;(滤波结束信号)
  p_s_load1    :OUTPUT;      (第一次并串转换的装入信号)
  p_s_load2    :OUTPUT;      (第二次并串转换的装入信号)
  acc_load1    :OUTPUT;      (第一次移位相加的装入信号)
  acc_load2    :OUTPUT;      (第二次移位相加的装入信号)
  add_sub1     :OUTPUT;      (第一次移位相加的加减控制信号)
  add_sub2     :OUTPUT;      (第二次移位相加的加减控制信号)
```

```
flip_en      :OUTPUT;      (装入最后结果的使能能信号)
f_clr        :OUTPUT;      (对于调整模块每次都要先进行清零)
)

VARIABLE
shift[dff_num..1] :DFF;

BEGIN
shift[].clk = clk;
shift[].clrn = clr;
(连接滤波结束信号(z_end))
shift[1].d = z_end;
FOR i IN 2 TO dff_num GENERATE
shift[i].d = shift[i - 1].q;
END GENERATE;
p_s_load1 = shift[1].q;
acc_load1 = shift[2].q;
add_sub1 = !shift[err_wid + 1].q;

p_s_load2 = shift[err_wid + 2].q;
acc_load2 = shift[err_wid + 3].q;
add_sub2 = !shift[2 * (err_wid + 1) + 1].q;
f_clr = !add_sub2;
flip_en = shift[dff_num].q;
END;
```

(15)调整模块(adjust.tdf)

```
INCLUDE "adjust_con";
INCLUDE "adjust_mult";

PARAMETERS
(
coef_wid = 16,
coef_pre = 15,
tap = 7,
term = 3,
x_wid = 12,
x_pre = 10,
```

```
pulse = 2,     (两电平,以确定 an[ ]宽度)
f_zn_wid = 9   (来自误差模块,此参数也许应该在顶层文件中设定,err 模块和
                本模块应用它)
);

CONSTANT coun = ceil(log2(term));
CONSTANT number = 2^coun;
CONSTANT ram_num = ceil(tap div term);  (RAM 组数目,此处为 3)
CONSTANT ff_num = (ram_num - 1) div 2;(前馈 RAM 组数目,此处为 1)
CONSTANT bf_num = ff_num;               (反馈 RAM 组数目,此处为 1)
CONSTANT an_wid = LOG2(pulse) + 1;      (判决信号宽度,两电平时为 2)
CONSTANT x_num = floor((tap) / (2));    (移位寄存器输出的 sub_ram 的 x
                                         [ ] 宽度)

SUBDESIGN adjust
(
f_clr                                  :INPUT ;(来自误差模块(err))
f_end                                  :INPUT ;(来自误差模块(err))
clr                                    :INPUT = VCC;
clk                                    :INPUT ;
x_ff[x_num..1][x_wid..1]               :INPUT ;(来自滤波器模块(filter))
x_main[x_wid..1]                       :INPUT ;(来自滤波器模块(filter))
x_bf[x_num..1][an_wid..1]              :INPUT ;(来自滤波器模块(filter))
err[f_zn_wid..1]                       :INPUT ;(来自误差模块(err))
ram_data[ram_num..1][coef_wid..1]:OUTPUT ;
ram_w_r                                :OUTPUT ;(RAM 读写信号)
main_w_r                               :OUTPUT ;(主抽头 RAM 的读写信号)
address[term..1]                       :OUTPUT ;(RAM 读写时的地址信号)
select                                 :OUTPUT ;(地址选择信号)
)

VARIABLE
control :adjust_con WITH (err_wid = coef_wid,
                          term = term);
mult_ff[ff_num..1] :adjust_mult WITH (x_wid = x_wid,
                                      x_pre = x_pre,
                                      term = term,
                                      coef_wid = coef_wid,
                                      coef_pre = coef_pre,
```

```
                                                f_zn_wid = f_zn_wid);
mult_bf[bf_num..1] :adjust_mult WITH (x_wid = x_wid,
                                      x_pre = x_pre,
                                      term = term,
                                      coef_wid = coef_wid,
                                      coef_pre = coef_pre,
                                      f_zn_wid = f_zn_wid);
mult_main :adjust_mult WITH (x_wid = x_wid,
                             x_pre = x_pre,
                             term = 1,
                             coef_wid = coef_wid,
                             coef_pre = coef_pre,
                             f_zn_wid = f_zn_wid);

xff[x_num..1][x_wid..1]   :NODE;(用于调试)
xmain[x_wid..1]           :NODE;(用于调试)
xbf[x_num..1][an_wid..1]  :NODE;(用于调试)

BEGIN
(以下节点用于程序调试)
xff[][] = x_ff[][];
xmain[] = x_main[];
xbf[][] = x_bf[][];

(连接调节_控制模块(adjust_con)的输入)
control.clk = clk;
control.clr = clr;
control.f_end = f_end;
control.f_clr = f_clr;
(连接前馈副抽头 RAM 调整乘积模块的输入)
mult_ff[].clk = clk;
mult_ff[].clr = clr;
mult_ff[].select[] = control.select_x[];
mult_ff[].p_s_l = control.p_s_l;
mult_ff[].acc_l = control.acc_l;
mult_ff[].add_sub = control.add_sub;
mult_ff[].and_x = control.and_x;
mult_ff[].f_zn[] = err[];
FOR i IN 1 TO ff_num GENERATE
```

```
mult_ff[i].xin[][] = xff[I * term..(i - 1) * term + 1][];
END GENERATE;

(连接反馈副抽头 RAM 调整乘积模块的输入)
mult_bf[].clk = clk;
mult_bf[].clr = clr;
mult_bf[].select[] = control.select_x[];
mult_bf[].p_s_l = control.p_s_l;
mult_bf[].acc_l = control.acc_l;
mult_bf[].add_sub = control.add_sub;
mult_bf[].and_x = control.and_x;
mult_bf[].f_zn[] = err[];
FOR i IN 1 TO bf_num GENERATE
mult_bf[i].xin[][x_wid..x_wid - an_wid + 1] = xbf[i * term..(i - 1)
                                              * [term + 1][];
mult_bf[i].xin[][x_wid - an_wid..1] = GND;(扩展 10 个 0)
END GENERATE;

(连接主抽头 RAM 调整_乘积模块的输入)
mult_main.clk = clk;
mult_main.clr = clr;
mult_main.p_s_l = control.main_p_s;
mult_main.acc_l = control.main_acc;
mult_main.add_sub = control.main_add_sub;
mult_main.flip_en = control.flip_en;
mult_main.f_zn[] = err[];
mult_main.xin[][] = xmain[];
(连接输出)
ram_data[ram_num][] = mult_main.data[];
FOR i IN 1 TO bf_num GENERATE
ram_data[i + bf_num][] = mult_bf[i].data[];
ram_data[i][] = mult_ff[i].data[];
END GENERATE;
ram_w_r = control.ram_w_r;
main_w_r = control.main_w_r;
address[] = control.address[];
select = control.sel_addr;
END;
```

(16)调整_乘积模块(adjut_mult.tdf)

```
INCLUDE "LPM_MUX";
INCLUDE "s_term";
INCLUDE "accumulator";

PARAMETERS
(
x_wid = 12,
x_pre = 10,
term = 3,
coef_wid = 16,
coef_pre = 15,
f_zn_wid = 9                    (取决于 err 模块,现为 9 位,需扩展到 16 位)
);

CONSTANT err_wid = coef_wid;      (f_zn[9]扩展后的宽度)
CONSTANT err_pre = coef_pre;
CONSTANT c = ceil(log2(term));
CONSTANT coun = (c > 1 ?c :1);
CONSTANT number = 2^coun;

SUBDESIGN adjust_mult
(
clk                          :INPUT;
clr                          :INPUT = VCC;
xin[term..1][x_wid..1]       :INPUT;
select[coun..1]              :INPUT = GND;(选择 xin[ ],以顺序计算 h1、
                                           h2、h3)
p_s_l                        :INPUT = GND;(用于副抽头 RAM 并串转换)
acc_l                        :INPUT;      (用于副抽头 RAM 数据装入)
add_sub                      :INPUT;      (用于副抽头 RAM 数据加减控
                                           制)
flip_en                      :INPUT = GND;(用于主抽头 RAM 数据装入使
                                           能)
and_x                        :INPUT = GND;(利用 and_x 形成顺序写入 ram
                                           的数据,即系数调整量)
f_zn[f_zn_wid..1]            :INPUT;      (控制误差函数值,来自 err 模块)
data[coef_wid..1]            :OUTPUT;     (系数调整量,送往滤波器模块
                                           (filter))
```

```
)

VARIABLE
p_to_s :s_term WITH (in_width = err_wid,
                     term_wid = 1);
accumul :accumulator WITH (x_width = x_wid,  (要累加的数)
                           x_pre = x_pre,
                           y_width = err_wid,(要累加的次数)
                           y_pre = err_pre,
                           out_int = coef_wid - coef_pre,
                           out_pre = coef_pre,
                           pipeline = "Yes");
(对于 sub_ram,计算 3 次,即分别计算 h1,h2,h3,而对于 main_ram,只算一次,
无须 lpm_mux)
IF term > 1 GENERATE
mux2 :lpm_mux WITH (lpm_width = x_wid,
                    lpm_size = number,
                    lpm_widths = coun);
END GENERATE;

BEGIN
(连接数据选择器的输入)
IF term > 1 GENERATE
mux2.sel[] = select[];
mux2.data[0][] = GND;
mux2.data[number - 1..1][] = xin[][];
accumul.datain[] = mux2.result[] & p_to_s.ram_ad[];
ELSE GENERATE                (主抽头环节 RAM 不需要数据选择器)
accumul.datain[] = xin[][] & p_to_s.ram_ad[];
END GENERATE;

(连接并串转换模块(s_term)的输入)
p_to_s.clk = clk;
p_to_s.clr = clr;
p_to_s.load = p_s_l;
(扩展 f_zn 的符号位)
p_to_s.data_in[][f_zn_wid..1] = f_zn[];
p_to_s.data_in[][err_wid..f_zn_wid+1] = f_zn[f_zn_wid];(f_zn[]最高位)
```

```
(连接移位相加模块的输入)
accumul.clk = clk;
accumul.clr = clr;
accumul.a_load = acc_l;
accumul.add_sub = add_sub;

IF term > 1 GENERATE
accumul.f_en = p_s_l;
ELSE GENERATE      (主抽头 RAM 只需要一个使能寄存器)
accumul.f_en = flip_en;
END GENERATE;

(输出)
IF term > 1 GENERATE
data[] = accumul.dataout[] & and_x;
ELSE GENERATE(主抽头 RAM 不需要规格化调整)
data[] = accumul.dataout[];
END GENERATE;
END;
```

(17)调整_控制模块

```
INCLUDE "LPM_COUNTER";
INCLUDE "MUX";

PARAMETERS
(
err_wid = 16,
term = 3
);

CONSTANT coun = ceil(log2(term));
CONSTANT number = 2^coun;
CONSTANT dff_num = err_wid + 1;
CONSTANT addr_wid = term;

SUBDESIGN adjust_con
(
f_clr                    :INPUT ;(来自误差模块)
f_end                    :INPUT ;(来自误差模块)
```

```
clr                                  :INPUT = VCC;
clk                                  :INPUT;
(副抽头调节乘积的控制信号)
p_s_l                                :OUTPUT;(并串转换装入信号)
acc_l                                :OUTPUT;(移位相加装入信号)
add_sub                              :OUTPUT;(移位相加加减控制信号)
(主抽头 RAM 调节乘积的控制信号)
main_p_s                             :OUTPUT;
main_acc                             :OUTPUT;
main_add_sub                         :OUTPUT;
flip_en                              :OUTPUT;
(副抽头 RAM 调节乘积的控制信号)
select_x[coun..1]                    :OUTPUT;(规格化控制信号)
and_x                                :OUTPUT;(与相乘数据 AND 后,送 CHANG
                                             _RAM)
(以下信号送往滤波器模块的 RAM)
ram_w_r                              :OUTPUT;(副抽头 RAM 读写控制信号)
main_w_r                             :OUTPUT;(主抽头 RAM 读写控制信号)
address[addr_wid..1]                 :OUTPUT;(地址信号,依次从 0 到 7)
sel_addr                             :OUTPUT;(地址选择,用以控制选择 RAM
                                             地址,即改变)
                                             (系数时为 1,正常读时为 0;也即
                                             改变)
                                             (h1、h2、h3 时为 1,其他任何时间
                                             都为 0)
)

VARIABLE
shifters[dff_num..1]                 :DFFE;(用于副抽头 RAM 数据存贮)
main_shift[dff_num + 2..1]           :DFF;(用于主抽头 RAM 数据存贮)
count1 :lpm_counter WITH (lpm_width = coun + 1,
                          lpm_direction = "up");
count2 :lpm_counter WITH (lpm_width = number,
                          lpm_direction = "up");

mux1 :mux WITH (WIDTH = number,
                WIDTHS = coun);
ff          :DFFE;
count1_en   :NODE;
```

```
write_sub  :NODE;                    (用于调试)
write_main :NODE;                    (用于调试)

BEGIN
(产生 p_s 和累加器所需各信号)
shifters[].clk = clk;
shifters[].clrn = clr & !f_clr;
main_shift[].clk = clk;
main_shift[].clrn = clr & !f_clr;
shifters[].ena = count1_en;

shifters[1].d = f_end # !add_sub;
FOR i IN 2 TO dff_num GENERATE
shifters[i].d = shifters[i - 1].q;
END GENERATE;
p_s_l = shifters[1].q;
acc_l = shifters[2].q;
add_sub = !shifters[dff_num].q;

main_shift[1].d = f_end;
FOR i IN 2 TO dff_num + 2 GENERATE
main_shift[i].d = main_shift[i - 1].q;
END GENERATE;
main_p_s = main_shift[1].q;
main_acc = main_shift[2].q;
main_add_sub = !main_shift[dff_num].q;
write_main = main_shift[dff_num + 2].q;
main_w_r = write_main;
flip_en = main_shift[dff_num + 1].q ;(第一次捕捉后送给主抽头 RAM 作为写
                                      信号)

count1.clock = clk;
count1.aclr = clr & f_clr;
count1.cnt_en = p_s_l;
select_x[] = count1.q[1..0];

count1_en = !(count1.q[] == b"101");
count1.clk_en = count1_en;
```

```
count2.clock = !clk;
count2.aclr = p_s_l;

ff.clk = clk;
ff.clrn = clr & !f_clr;
ff.ena = !add_sub;
ff.d = VCC;
count2.clk_en = ff.q;

sel_addr = ff.q & count1_en;

FOR i IN 1 TO term GENERATE
address[i] = count2.q[i];
END GENERATE;

mux1.sel[] = select_x[];
mux1.data1 = GND;
mux1.data2 = count2.q[3];
mux1.data3 = count2.q[2];
mux1.data0 = count2.q[1];
write_sub = count2.q[0] & count1_en;
ram_w_r = write_sub;
and_x = mux1.result;
END;
```

参 考 文 献

1 MAX + PLUS Ⅱ AHDL ALTERA

2 MAX + PLUS Ⅱ GETTING STARTED

3 Data Book 1996. ALTERA(1998)

4 王世一. 数字信号处理. 北京 :北京工业学院出版社,1988

5 樊昌信. 数字专用集成电路设计. 北京 :人民邮电出版社

6 刘宝琴,张芳兰,田立生. ALTERA 可编程逻辑器件及其应用. 北京 :清华大学出版社,1995

7 陈尚勤,李晓峰. 快速自适应信息处理. 北京 :人民邮电出版社

8 孟宪元. 可编程专用集成电路原理、设计与应用. 北京 :电子工业出版社

9 梁启联,周正,刘泽民." 盲均衡算法综述 ". 电路与系统学报,VOL. 1,NO. 3,PP51—56,Sep,1996