第32卷 第15期 计 算 机 工 程 2006年8月
Vol.32 № 15 Computer Engineering August 2006
·软件技术与数据库· 文章编号：1000—3428(2006)15—0049—03 文献标识码：A 中图分类号：TP229

多进程下子程序综合方法研究

胡燕翔[1,2]，孟 晗[3]
(1. 天津大学电子信息工程学院，天津 300072；2. 天津师范大学计算机学院，天津 300072；3. 北京理工大学ASIC研究所，北京 100081)

摘 要：子程序的综合方法包括内联扩展、单独的控制单元和数据通道部分和子例程，这些方法都是针对在源描述中只包括单独一个进程时的情况。在多进程的情况下会出现一些问题。给出必须采用子程序内联扩展方法的原则，提出并且实现了将子程序转换成临界资源的方法。同时还解决了包含多个return语句的函数的内联扩展实现方法如何在保持语义的同时进行优化的问题。
关键词：高级综合；VHDL；子程序；多进程

Research on Subprogram Synthesis in Mutli-processes

HU Yanxiang[1,2], MENG Han[3]
(1. School of Electronics and Information Engineering, Tianjin University, Tianjin 300072; 2. College of Computer, Tianjin Normal University, Tianjin 300072; 3. ASIC Research Institute, Beijing Institute of Technology, Beijing 100081)

【Abstract】The synthesis methods of subprogram include inline expansion of subprogram, independent control unit and data path and subroutine. These methods are for the case that there is only one process in the source description. In the case of multiple processes some problems will occur. This paper first provides the cases in which the inline expansion method must be taken, then provides and realizes the method of transforming the subprogram to critical resource, and also solves the problem that how to keep the meaning of the function which includes several return statements that are realized in inline expansion method and how to optimize.
【Key words】High-level synthesis; VHDL; Subprogram; Multiple process

子程序是VHDL语言的高级特性，包括函数和过程。目前的设计越来越倾向于在较高的层次上描述数字系统，子程序的使用会提高设计的重用性和可读性。子程序综合有3种方法：内联扩展，单独的控制单元与数据通道和子例程实现方法[1,4,5]。这些方法都是针对源描述中只有一个进程的情况，并没有考虑函数中包括多个return语句和表达式中包括多个函数调用的情况；当函数包括多个return语句和一个表达式中包括多个函数调用的时候，有多种内联扩展的方法，如何在保持语义的同时进行优化，也是一个重要的问题。当VHDL源描述中包含多于一个的进程时，由于子程序不允许多个进程同时进入，多个进程同时对同一个子程序的调用会产生问题。多个进程带来的另一个问题是调用子程序有可能引起死锁。本文提出了多进程情况下使用内联扩展的原则，解决了在保持函数语义的情况下进行内联扩展的优化问题，提出并解决了子程序不能被多个进程同时进入的问题。

1 多进程调用子程序的综合方法

在单进程中调用子程序采用子例程的综合方法不会引起任何问题，因为进程中的语句是顺序执行的，不会发生重复进入子程序的情况。在多进程调用子程序的情况下，由于进程是并发执行的，如果不对子程序加以保护，有可能发生子程序被多次进入的情况。如果子程序被超过一个进程同时进入，子程序会产生错误的运行结果，此时必须对其加以保护，使子程序免于同时被多个进程进入。这时的子程序相当于临界资源，加入的保护电路使子程序在多个进程同时调用的情况下，多次调用会顺序进行，第1个进程调用完毕后，第2个进程才能够进入到子程序中。文献[2]中提出的方法只能处理两个进程，每个进程只调用一次子程序的情况。在多于2个进程并且每个进程多次调用子程序的情况下，必须首先确定每个进程的优先级，也就是在多个进程同时调用子程序的时候，必须决定哪一个进程拥有首先进入子程序的权利，而且在优先权高的进程调用完毕之后，必须保证下一个进程能够进入到子程序中。根据这个原则，本文提出了如下的保护电路。

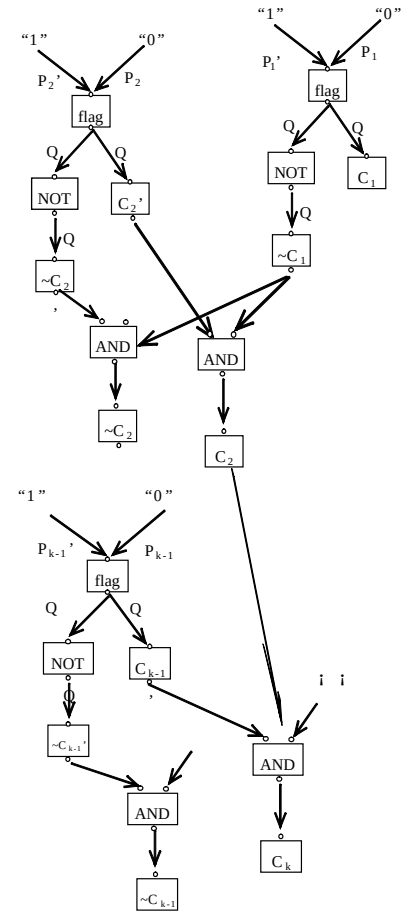
图1中，~C是条件C的非。P_j是进程j(0<j<k)中所有调用子程序的位置的集合，P_j'是进程j中子程序调用前的位置的集合。进程1到进程j调用子程序的保护条件依次是C_1，C_2，...，C_{k-1}，~C_{k-1}(因为最后一个进程的优先级最小，不需要优先级，所以使用前一个进程产生的条件)。C_1，C_2，...，C_{k-1}'的初始值为“1”。2个需要解决的问题是：

(1)当多个进程同时调用子程序的时候，哪一个进程具有较高的优先权；

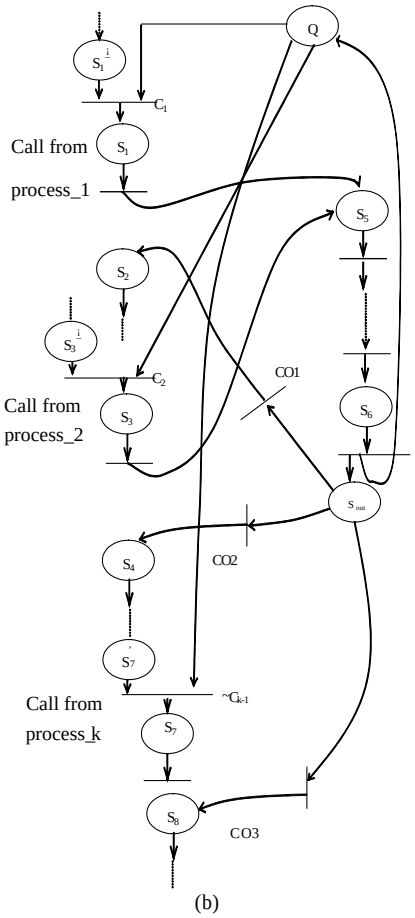
(2)怎样保证优先权高的进程离开子程序时，优先权低的进程能够自动执行。

从图1的保护电路中可以看出，进程优先权按源描述中出现的次序依次降低。从图1可以看出，当多个进程同时调用同一个子程序的时候，由于进程的优先权不同，优先权高的进程首先进入子程序。按照VHDL的语义，同一个进程内的子程序顺序执行，它们之间不会发生同时调用同一个子程序的情况，因此一个进程只具有一个优先权。

基金项目：国防基础科研基金资助项目
作者简介：胡燕翔(1969—)，男，博士后、副教授，主研方向：硬件描述语言及其相关模拟，综合技术，VLSI设计；孟 晗，博士
收稿日期：2005-08-28 **E-mail：**yanxianghu@263.net



(a)



(b)

图 1 保护电路及其控制流程

根据这个保护电路，当优先权高的进程进入到子程序中，优先权低的子程序将会挂起，当优先权高的进程执行完毕后，具有次优先权的进程得到 token，这个进程会执行。当只有一个进程调用子程序的时候，这次调用的保护条件为“1”，进程能够进入子程序。当一个进程正在调用子程序的时候，如果一个优先权更高的进程调用子程序，由于位置 Q 中没有 token，优先权高的进程只能等待当前的调用结束后才能执行。这样子程序在多进程同时调用的时候变成了临界资源，在只有单个进程调用的时候也能够执行，解决了多个进程同时调用子程序带来的问题。

2 使用子程序的内联扩展综合方法的原则

在单进程的情况下，可以使用子程序的 3 种综合方法的任意一种。在多进程的情况下，使用子例程的方法有时会引起死锁问题，如何决定采用哪种方法，是一个很重要的问题。

采用内联扩展方法，每一次进程调用将子程序的层次结构展平。经过内联扩展后，要综合的进程只包含一个单独的控制体和数据通道。内联扩展方法有助于提高综合的优化结果，但是内联扩展方法的结果电路比其它方法实现的结果电路规模要大。在子程序只被调用一次的情况下，使用内联扩展方法是最优的方法，因为其它的实现方法需要一些控制电路来传入参数和传出参数。如果在所有的进程中子程序只被调用过一次，那么这个子程序使用内联扩展的方法综合[4, 5]。

在多进程调用子程序的时候，由于进程是并发执行的，在有些情况下使用子例程的实现方法会引起死锁，这时只能采用子程序内联扩张的方法。

如果通过子例程方式综合，子程序的保护机制通过互斥来解决。这样将子程序变成临界资源。如图 1 所示，如果位置Q拥有一个token，则子程序变成空闲，可以被两个进程之一激活，只要子程序在执行，位置Q就不会拥有token，这样在子程序执行完毕之前，子程序不会有第 2 次激活。但是，如果两个进程同时调用子程序，会发生什么情况呢？根据图中的解决方案，process_1 拥有更高的优先权。这个策略是使用条件C和~C来实现的。条件C和~C基于存储在flag中的值产生。这个节点在位置S_1'的控制下，每次process_1 试图进入子程序时，这个节点得到值 1。

根据VHDL定义[3]，在函数中不会包含wait语句，在过程中可以包含wait语句。wait语句将进程挂起，将过程变成临界资源，有可能引起死锁：在第 1 个进程中调用包含wait语句的过程，如果wait语句中的事件由第 2 个进程引发，而第 2 个进程在引发这个事件之前调用同 1 个过程，由于这个过程是临界资源，第 2 个进程只能等待，这样就引起了死锁。看下面的例子。

```
Signal S: std_logic:='0';
…
procedure P(a: in integer) is
…
begin
…
wait on S;
…
end P;
…
process
begin
```

```
…
  P(1);
…
end process;
process
begin
…
  P(2);
  S≤'1'; …
…
end process;
```

在模拟时，上面的例子不会有任何问题。然而，综合出来的硬件，如果没有采用内联扩展的方式，而是采用硬件保护的措施，当执行到 wait on S 语句的时候，就会有潜在的死锁。如果 P(1)首先调用，这 2 个进程将会发生死锁：第 1 个进程将会在过程里面等待信号 S 的一个事件，而 S 的事件必须作为第 2 个进程的信号赋值的结果，这时第 2 个进程正在从等待过程变成空闲状态。这样，一个被多个进程调用的过程，如果它包含一个 wait 语句或者调用另外一个包含 wait 语句的过程，那么这个过程必须使用内联扩展的方式综合。

综上所述，采用内联扩展方法的原则是：

(1)子程序在所有的进程中只被调用一次；

(2)一个被多个进程调用的过程，如果包含一个 wait 语句或者调用另外一个包含 wait 语句的过程，这个过程必须使用内联扩展的方式综合。

3 内联扩展的语义保持和优化

过程的内联扩展方法可以简单地直接将过程体扩展到进程之中，对于函数的内联扩展方法，使用函数中的 return 语句中的表达式替换包含函数调用的表达式，由于函数可以包括多个 return 语句，表达式中可以包含多个函数调用，如何在保持语义的同时选择一个最佳的方案是一个重要的问题。首先看下面一个例子。

```
…
function abs(integer x) return is
  if (x>0)
return x;
else
return –x;
end if;
end function;
…
function sqr(integer:x) return is
  return x*x;
end function;
…
```

在表达式 a :=abs(b)+sqr(c);中，如果直接按照语义采用内联方法，内联结果如下：

```
…
if (b>0)
  tmp1 := c*c;
  a := b + tmp1;
else
  tmp2 := c*c;
  a := tmp2 – b;
end if;
…
```

在这里，函数 sqr 内联了 2 次，如果函数 sqr 还包括多个 return 语句，那么这种组合数量是 2 个函数中包含的 return 语句数量的乘积。实际上在这里面是可以进行一些优化的。

根据VHDL语义，函数的参数的传递方式是传值[3]，函数的参数只能是变量和常量类型，它们的模式为“ IN ”或者“ INOUT ”，函数不改变它的参数的内容，根据这一点，完全可以通过顺序内联表达式内的函数，将这些函数的结果存放到一个临时变量中，在所有的函数都内联之后，使用临时变量中的值替换表达式中的函数调用。使用这种方法的处理结果如下：

```
…
if (b>0)
  tmp1 := b;
else
  tmp1 := -b;
end if;
tmp2 := c*c;
a := tmp1 + tmp2;
…
```

这样既保持了程序的语义，也优化了程序。

4 结论

多进程下的子程序的综合方法与单进程中的子程序的综合方法有一定的区别：在单进程中，可以使用 3 种综合方法中的任意一种对子程序进行综合；在多进程中，在某些情况下只能使用内联扩展的方法，本文给出了使用内联扩展方法的原则。在多进程中，由于存在多个进程同时进入子程序的情况，因此使用子例程的综合方法需要保护电路，本文提出了一种应用于多于一个进程的子程序的保护电路。在函数内联扩展方法中，函数中存在着多个 return 语句，表达式中也可能包含多个函数调用，本文提出了如何在保持程序语义的同时进行优化的方法。

参考文献

1 张俭峰. 高级综合中 VHDL 子程序综合方法研究[D]. 北京：北京理工大学, 2000: 19-20.

2 Eles P, Kuchcinski K, Peng Z. System Synthesis with VHDL[M]. New York: Kluwer Academic Publishers, 1998: 241.

3 ANSI/IEEE 1076－1993. VHDL Language Reference Manual[S]. 1993.

4 张东晓. 高级综合实用化技术研究[D]. 北京: 北京理工大学, 1999.

5 Elliott J P. Understanding Behavioral Synthesis: A Practical Guide to High-level Design[M]. Netherlands: Kluwer Acdemic Publisher, 1999.