文章编号: 1001- 2486(2003) 06- 0010- 06

基于程序切片的电路提取技术*

朱　丹, 李　暾, 万　海, 郭　阳, 李思昆

(国防科技大学计算机学院, 湖南 长沙　410073)

摘　要: 从 HDL 设计描述中提取电路在 VLSI 设计验证、低功耗分析、测试生成等方面有广泛的应用需求。提出了一种采用程序切片技术实现的新的电路提取方法, 并深入论述了基于程序切片技术从 Verilog 描述中进行电路提取的理论基础。该方法可以为每一个感兴趣的信号获取其"链接切片"。与以前的方法相比, 该方法的优点是细粒度的、不受书写格式的限制, 并且能处理更多 Verilog 的语法元素。该方法已经被集成到现有设计流程中, 实验结果表明其方便、高效, 有良好的通用性。

关键词: 程序切片; 链接切片; 进程依赖图; 电路提取

中图分类号: TP302.7　　文献标识码: A

Automatic Circuit Extraction Using Program Slicing

ZHU Dan, LI Tun, WAN Hai, GUO Yang, LI Si-kun

(College of Computer, National Univ. of Defense Technology, Changsha 410073, China)

Abstract: The design extraction from HDL description has been greatly needed in modern VLSI design process, such as the design verification, low power analysis, test generation and so on. This paper presents a new circuit extraction method using program slicing technique, and develops an elegant theoretical basis, based on program slicing, for circuit extraction from Verilog description. With the technique we can obtain a "chaining slice" for each given signal of interest. Our method has advantage in its fine grain, without writing-style limitation and in dealing with more Verilog components characteristics. The technique has been used in the design process and the results show its convenience, efficiency and good practicability.

Key words: program slicing; chaining slice; process dependence graph; circuit extraction

随着 IC 设计复杂性的增加, 功能验证已成为主要瓶颈。因此, 进行形式化验证和基于模拟的验证时, 须采取不同技术对设计中的数据通路和控制通路部分分别加以验证。然而, 用这些方法进行电路提取时往往依赖于特定标记, 需手工完成。文献[1, 2]提供了一些使用编译技术提取 FSM 的方法, 但基于编译器的方法必须限制用户代码风格, 对来源于不同设计者各式风格的真实设计进行处理相当困难。文献[3]提出的提取控制通路的方法对代码风格限制较少, 然而, 它是基于进程模型的, 粒度较大, 会引入冗余信号。

程序切片技术最早是由 Weiser 提出的[4], 它是一种静态程序分析技术, 能够从程序中提取与应用相关的语句, 这些语句就是所谓的切片。该技术已经得到广泛的研究并在软件工程中得到了大量应用。但是这些研究中的大多数都是为串行程序提出的, 不能直接应用于像 Verilog 或者 VHDL 这样的并发程序。文献[5]提出了一种针对 VHDL 的自动程序切片算法, 然而, 该技术只能对给定的信号和定义该信号的语句进行切片。文献[6]采用程序切片技术从被测单元中提取了外围电路部分来完成 ATPG, 但并没有提出解决 Verilog 并发性的办法。

本文提出一种新的基于程序切片技术提取电路的方法。提取出的部分叫链接切片, 其功能是在两信号集间传输信号。与已有方法相比, 优点如下: ①对程序代码风格无限制, 既能处理行为描述也能处理综合描述; ②在信号级进行处理, 因此是细粒度的, 能减少被提取部分的冗余代码; ③是完全自动化

* 收稿日期: 2003- 07- 20

基金项目: 国家自然科学基金重点项目基金资助(90207019); 863 项目基金资助(2002AA1Z1480)

作者简介: 朱丹(1980—), 女, 硕士生。

的, 无需用户干预。

1　Verilog 进程依赖图

在 Verilog 中, 信号机制与设计物理实现中真实的数据通路紧密(或松散, 这依赖于数据类型) 相关。我们提出了一种新的结构 ——进程依赖图(*PDG*) 来表示这种进程间的通信机制。下面给出 *PDG* 的定义。

定义 1　$SL(p)$表示进程 p 的敏感列表, s 表示p 中的语句。$S(p)$ 表示 p 的语句集合。$DEF(s)$ 表示在语句 s 中定义的变量集合。$REF(s)$ 表示 s 所引用的变量的集合。

定义 2　$INPUT(p)(OUTPUT(p))$表示进程 p 的输入(输出) 信号集合, $REF(p)(DEF(p))$表示进程中引用(定义)的信号集合, 其定义如下:

$$REF(p) = INPUT(p) = SL(p) \cup \bigcup_{s \in S(p)} REF(s)$$

$$DEF(p) = OUTPUT(p) = \bigcup_{s \in S(p)} DEF(s)$$

定义 3　$PDG_M = < N_P, E, \Sigma, D, U, S, I, O>$ 表示模块 M 的进程依赖图。其中: Σ 是M 中变量的集合; N_p 代表M 中的进程结点集合; E 是边的集合; $I \subseteq N_P$, 是 M 的初始输入的集合; $O \subseteq N_P$, 是 M 的初始输出的集合; $D: N_{G_C} \mapsto \Delta(\Sigma)$ 和 $U: N_{G_C} \mapsto Y(\Sigma)$ 分别表示从图 PDG_M 的结点集合N_{G_C}到与这些结点相关的已定义的变量集(Δ) 和被使用的变量集(Ψ) 的函数映射; $S: E \mapsto \Psi(\Sigma)$ 表示图的边集 E 到边所带的变量的函数。

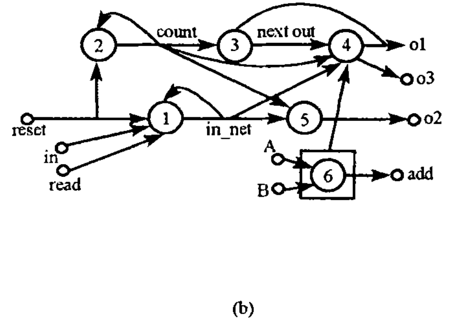
```
Module example (clk, reset, read, in, o1, o2, o3);

input clk, reset, read;
input  [3: 0] in;
output  [3. 0] o1, o2, o3;
reg [3: 0]o1, in_net, count, next_out,

always @ (posedge clk)                          (1)
begin
  if (reset) in_net=0;
  else if (read) in_net=in,
  else in_net=in_net;
end
always @ (posedge clk)                          (2)
begin
  if (reset) count=0;
  else if (count)==15) count=0;
  else count = count+1;
end

always @ (count or o1)                          (3)
begin
  if (count= =15)next_out=o1+1;
  else next_out=o1 ;
end
always @ (posedge clk)                          (4)
begin
1  o1=add4 (next_out, count);
2  o3=in_net;
end
assign o2= in_net+count;                        (5)
function Add4(A, B);                            (6)
  input [3:0]A, B;
  add=A+B;
endfunction
endmodule
```

(a)



(b)

图 1　一个简单的 Verilog 程序及其 *PDG* 示例图
Fig. 1　A simple Verilog example and its *PDG*

进程语句还包括别的并发语句, 例如连续赋值语句和模块实例化语句。我们把这些语句(包括函数和任务) 都当作简单语句, *DEF* 和 *REF* 可以从它们的声明中得到, 分别对应输出和输入集。图 1 给出一个简单的 Verilog 描述及其 *PDG*。在后面的算法中我们都将引用此例进行分析。表 1 列出了图 1 中各个结点的属性。在下面的讨论中, “ 进程语句” 和“ 进程结点” 意思相同, 可以互换。

表 **1**　图 **1** 中结点的 ***DE***　集和 ***RE***　集
Tab. 1　*DEF* and *REF* of the nodes in Fig. 1

Node ID	*DEF* (*OUTPUT*)	*REF* (*INPUT*)
1	{in _ net}	{clk, reset, read, in, in _ net}
2	{count}	{clk, reset, count}
3	{next _ out}	{o1, count}
4	{o1, o3}	{clk, next _ out, in _ net}
5	{o2}	{in _ net, count}
6	{add4}	{A, B}

2　Verilog 程序切片

定义 4　$C= < Vs, V>$ 表示链接切片标准, 其中 V 和 Vs 都是模块 M 的进程依赖图PDG_M 中变

量集的子集。

定义5 根据链接切片标准 $C=<Vs,V>$ 得到一个链接切片，表示为 PrS_C，它是 M 的一个可执行子集，包含了所有从 Vs 出发的对 V 的值有直接或间接影响的进程结点。

算法主要包括三步：①将链接切片标准转换为传统的切片标准；②为切片提取相关的进程结点；③用转换后得到的切片标准在提取出的进程结点中进行切片。

在介绍算法之前，我们对 Verilog 代码作一个合理的假设——在函数和任务中定义或使用的每一个变量都必须是一个已经声明过的输入或输出变量。这个假设就保证了我们可以将函数和任务作为简单的顺序语句来处理，当一个函数或任务被包含在切片中时，简单地将它们加入切片当中即可。

2.1 切片标准转换

本文的切片标准不包含定义关键变量的语句，因此必须将它转换为传统的形式。首先，任选一个变量 $v\in V$，遍历 PDG_M，找出其相关进程集合 $RPS=\{p\mid(p\in N_P)\text{AND}(v\in DEF(p))\}$。然后，将标准转换，在 RPS 中寻找对 $v\in V$ 给出定义的最后一条顺序语句。在传统的标准中，v 是由用户指定的，用户知道在切片标准中应该包含哪些变量。然而，为了转换，我们必须定出应该包含哪些变量。图2给出了一段代码及其切片标准和切片结果。当 C 中只包含了标准语句中所定义的变量时，结果漏掉了语句2。改变 C，使之包含变量 k 以后，结果就正确了。

```
1    i=0;    C=<3, {i}>
2    k=1;
3    i=k;    result={1,3}
```

图2 一段代码

Fig.2 A code fragment

```
always @ (posedge clk)          (4)
  begin
     o1=add(next_out,count);
  end
```

图3 从第一步得到的切片

Fig.3 The slice obtained from step1

可以将链接切片标准 C_v 转换为集合 $C_{T_v}=<\{p\},\{i_p\},v_i\}>$，其中 $p\in RPS$，对于每一个 p，$i_p\in S(p)$，i 是最后一个对任意的 v 给出定义的语句，其中 $v_i\in(\{v\}\cup ref(i))$。我们调用这些标准进程语句、标准语句以及标准变量，并分别用 C_p，C_i，C_v 表示。

然后，先根据切片标准 $C_{T_v}=<i_p,v_i>$ 对 $p\in RPS$ 进行切片，得到的切片中包含直接和间接相关的语句。在第2步中将得到更多的进程依赖语句。每一个进程结点的切片迭代如下。

$SUCC(n)$ 为结点 n 的后继结点，$D(n)$ 为语句 n 中定义的变量集(语句左边)，$U(n)$ 为语句 n 引用的变量集(语句右边)。初值 $R^0_C(n)$ 为与切片标准 C 立即相关的变量集，定义为

$$R^0_C(n)=\{v=V_i\mid n=i\}\cup\{U(n)\mid D(n)\cap R^0_C(SUCC(n))\neq\Phi\}\cup\{R^0_C(SUCC(n))-D(n)\}$$

包含在切片中的语句的集合 $S^0_{C_{T_v}}$ 定义为：

$$S^0_{C_{T_v}}=\{n\in N_{G_{C_{T_v}}}\mid D(n)\cap R^0_{C_{T_v}}(SUCC(n))\neq\Phi\}$$

控制 $S^0_{C_{T_v}}$ 中语句执行的控制语句

$$B^0_{C_{T_v}}=\{b\in N_{G_{C_{T_v}}}\mid INFL(b)\cap S^0_{C_{T_v}}\neq\Phi\}$$

其中，$INFL(b)$ 为结点 b 条件控制的语句集合。

S_C 的建立都是递归地定义在那些对集合 V 有着直接或间接影响的变量和语句集上的，上标初值为0，代表了递归深度。

$$R^{i+1}_C(n)=R^i_C(n)\cup(R^0_{<b,U,(b)>}(n)) \tag{1}$$

$$S^{i+1}_{C_{T_v}}(n)=\{n\in S(p)\mid D(n)\cap R^{i+1}_{C_{T_v}}(SUCC(n))\neq\Phi\}\cup B^i_{C_{T_v}} \tag{2}$$

$$B^{i+1}_{C_{T_v}}=\{b\in S(p)\mid INFL(b)\cap S^{i+1}_{C_{T_v}}\neq\Phi\} \tag{3}$$

上述定义包含了对切片有间接影响的所有控制语句、控制语句中出现的控制变量和那些对控制变量产生影响的语句。程序将无限迭代直到满足 $S_{C_{T_v}}=S^{f+1}_{C_{T_v}}$，其中 f 表示迭代次数，且

$$\forall n \in N: R^{f+1}_{C_{T_v}}(n) = R^{f}_{C_{T_v}}(n) = R_{C_{T_v}}(n) \qquad (4)$$

用 S_1 表示这一步计算所得的切片, 然后必须为切片进程语句构造新的 *REF(INPUT)* 集合, 用 *S-REF(p)* 表示。

对于图 1 中的代码示例及其 *PDG* 图, 当使用 $PrS_C = < I, \{o1\} >$ 时, 则得到相关进程语句的集合为{(4)}, 转换后的切片标准为 $< (4), 1, \{o1, next_out, count\} >$ 。

2.2　*PDG* 切片

在 *PDG* 中, 从标准进程结点 p 起, 向后就能找到其直接依赖结点集 $DPN = \{p \in N_P \mid (e(p, C_p) \in E) \text{AND} (S(e) \in S-REF(p))\}$, 即 *DPN* 包含结点 p, 当且仅当 p 有一条出边指向标准进程的结点, 且该边所带的变量在标准进程结点的切片中被引用。

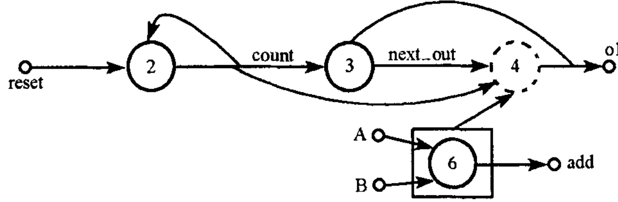


图 4　从第二步得到的 DPN 图
Fig. 4　DPN graph obtained from *PDG* slicing

图 4 是根据图 3 对 *PDG* 切片得到的结果, 粗线圈表示这一步所到达的进程结点, 虚线圈表示最后一次迭代到达的结点。求出 *DPN* 的这种算法是一种宽度优先搜索算法, 第一步标注所有具有到标准进程结点 p 的出边的结点, 这一步所找到的结点集用 *PreP* 表示; 第二步, 对于 *PreP* 中的每一条边, 若与该边相匹配的变量不属于集合 *S-REF(p)*, 就删去该边; 第三步, 对于 *PreP* 中的任一结点 p', 若没有从 p 到 p' 的边, 就删除它, 最后我们就能构造出 DPN 图了。*DPN* 中只包含了与 p 直接相关的结点、变量和边。我们把这种方法叫做前向映射法, 表示为 *Pre_Img(P)*, 其中 P 是我们期望从中找到 DPN 依赖图的进程结点集。

2.3　在 DPN 结点中的切片

经过第二步以后, *DPN* 的结点中就只包含对 *PDG* 中结点有直接或间接影响的输出变量了。它们正是应该加入到切片标准中的变量。

为了从 *PreP* 的进程语句中得到相关代码, 需要为这些进程语句构造切片标准。首先, 为每个 *DPN* 结点的每个输出变量构造切片标准, 采用的方法与 2.1 节中的方法一样。对每一个 *DPN* 结点重复上述的运算就可以为 *DPN* 依赖图找到所有的切片了。对于每个 *DPN* 结点, 该切片的计算和终止条件都与 (1) ~ (4) 式相同。图 5 给出了将上述运算方法运用于图 4 所得到的结果。进程 2 和进程 3 的右边给出了切片标准。

在 *DPN* 中得到了完整的切片后, 再用 2.1 节和 2.2 节的算法迭代, 终止条件是当 *DPN* 的运算达到某个结点 n 时, $DEF(n) \cap V_s \neq \phi$, 且在 Pre-Img 的运算中没有出现新的结点。最后的切片由 2.1 节和 2.2 节所得到的切片组合而成。

```
always@(posedge clk)                    (2)
begin
1   if(reset) count= 0;                 criterion:
2   else if(count= = 15) count= 0;      C= < 3, {count}>
3   else count= count+ 1;
end

always@(count or o1)                    (3)
begin                                   criterion:
1   if(count= = 15) next_out= o1+ 1;    C= < 2, {next_, o1}>
end

function Add4(A, B);                    (6)

  input[3:0] A, B;

  always@(A or B) add= A+ B;

endfunction
```

图 5　对 *DPN* 结点进行切片
Fig. 5　Slicing in *DPN* nodes of the example

当在进程结点中存在循环时(如图 6), 如果从不同的边到达已达的结点时, 将再次对已达的结点重复 2.1 节和 2.2 节中的迭代。例如: 在图 6 中, 假设在进程 2 中找到原始的 PrS_C, 在对 $p2$ 切片后, 我们通过标记为 $v1$ 的边到达 $p1$。在对 $p1$ 切片以后, 我们又从用变量 $w2$ 标记的边到达进程结点 $p2$, 虽然

$p2$ 是一个已达结点, 但用 $v2$ 标记的边与用 $w1$ 标记的边是不同的。因此, 必须将 $p2$ 放入 $PreP$, 对它进行再次切片。当为所有的标准变量计算完切片以后, 就得到了 C 的最终切片 S_F, $S_F = \bigcup_{v \in V} S_v$。

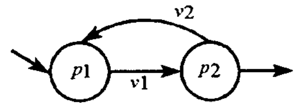


图 6 进程结点循环
Fig.6 Process nodes loop

2.4 Verilog 切片算法的正确性

图7 给出了切片算法, 图 8 是图 1 示例中最后得到的切片。该算法将 Verilog 描述和链接切片标准作为输入, 然后对所有的标准变量进行迭代来得到切片, 最后的切片应为各标准变量之切片的组合。

引理 1 链接切片标准的转换是完全的。

引理 1 表明从链接切片到传统切片的转换为在这些语句上进行切片包含了所有的变量定义语句和变量。

```
/*
  D is the Verilog description file;
  C= < Vs, V>  is the slicing criterion.
/*
Verilog _Slicing(D, C)
{
  for each v in V do
  {
    Transform _Criterion(v);
    Slicing(v, p);
    Found _DPN _Graph(p);
    while((! reach(Vs))||(find _new _DPN()< > NULL))
    {
      for each p in DPN graph
      Sv= Sv+ Slicing _DPN(p);
      /* find new DPN graph for last iteration DPN nodes* /
      Find _DPN _Graph(DPNp);
    }
  }
  Final _Slice();
}
```

图 7 Verilog 切片算法
Fig.7 Circiut extraction algorithm using program slicing

```
Module example(clk, o1);

input olk;
output [3: 0] o1;
reg [3: 0] o1, count, next _out;
always@ (posedge clk)            (2)
begin
  if(reset) count= 0;
  else if(count = = 15) count=
0;
  else count= count+ 1;
end
```

```
always@ (count or o1)            (3)
begin
  if(count= = 15) next _out= o1+
1;
  else next _out= o1;
end

always@ (posedge clk)            (4)
begin
  o1= add(next _out, count);
end

function Add4(A, B);             (6)
  input [3: 0] A, B;
  add= A+ B;
endfunction

endmodule
```

图 8 最后的切片结果
Fig. 8 The chaining slice of the example

引理 2 对进程依赖图进行切片可以找到相关进程结点的最大集合。

引理 2 表明我们能够遍历所有的相关进程语句来为给定的标准变量得到切片。

引理 3 根据切片标准得到的链接切片是分布在标准变量集合之上的。也就是说:

$$S_{<v_1 \cup v_2 \cup \dots \cup v_n>} = S_{<v_1>} \cup S_{<v_2>} \cup \dots \cup S_{<v_n>}$$

$$R_{<v_1 \cup v_2 \cup \dots \cup v_n>}(n) = R_{<v_1>}(n) \cup R_{<v_2>}(n) \cup \dots \cup R_{<v_n>}(n)$$

引理 3 表明对于一个给定信号, 其链接切片可以通过组合每一个标准变量的切片来得到。

定理 1 Q 是根据切片标准 C 从 Verilog 描述 M 得到的链接切片。进程轨迹 $T = < T_0, T_1, T_2, \dots, T_c, \dots >$ 表示 M 中相关信号的序列, 其中 c 表示模拟周期, 且 $T_c \in \{0, 1, X, Z\}$, 则 Q 和 M 的进程轨迹用标准变量加以区别。

3 实验结果

我们将切片方法主要用于设计验证, 尤其是在形式验证和自动化功能验证中(大多数是有限状态机) 分离 Verilog 描述中的控制部分和数据部分。但它并不局限于此, 还能广泛地应用于 VLSI 设计的各种应用之中。我们对 PicoJava Ⅱ[7] 微处理器核的DCU (数据缓存单元), SMU (栈管理单元) 和 IU-Pipe (整数流水线单元) 这几个部件进行了实验。表 2 给出实验的特性, 表 3 给出了在给定的有限状态机上进行切片提取的实验结果。切片算法的输入是 FSM 的状态寄存器。表 3 的第 3 列和第 4 列分别给出了 PDG 的结点和寄存器数目。为了评定算法的正确性, 回顾一下由我们的工具生成的 Verilog 代码的列表。用 VIS 系统将切片综合到 BLIF[8] 文件中, 并遍历 VIS 系统中每一个 FSM 的状态空间之后, 可达的状态和过渡关系都和 PicoJava Ⅱ体系结构手册中相关的描述一同编译。换句话说, 我们的工具所产生的切片与体系结构说明达到了 100% 的匹配, 能极大地减少每一个 FSM 的外围电路, 这有助

于我们为设计的控制部分有效地生成激励。

表 **2**　设计信息

Tab. 2　The design information

Designs	DCU	SMU	IU-Pipe
# Lines in HDL code	3979	1476	1953
# Nodes in PDG	309	74	68
# Edges in PDG	3504	1390	1434
# Registers in designs	385	217	4558

表 **3**　实验结果

Tab. 3　Experimental result

Designs	Extracted FSM Properties		
	FSM Name	# Nodes	# Register
DCU	Fill	4	5
	Miss	14	6
	WB	10	8
	Zero	8	6
SMU	Dribble	39	5
	Six	10	2
	Spill	32	4
IU-Pipe	Smiss	14	4
	Lduse	23	4
	Fold	17	5

这些设计描述中包含了各种一般的 Verilog 元素和书写风格。实验结果显示我们的方法可以有效地处理各种 Verilog 描述的实际设计。

参 考 文 献:

[1] Cheng C T, et al. Compiling Verilog into Timed Finite State Machines[C]. Proc. IEEE Int'l Verilog HDL Conf., IEEE Computer Soc. Press, Los Alamitos, Calif., 1995: 32-39.

[2] Hoskote Y V, et al. Automatic Verification of Implementations of Large Circuits Against HDL Specifications[J]. IEEE Trans. Computer-aided Design, 1997, 16(3): 217-228.

[3] Nan Chien, Liu Jimmy, Jou Jing-Yang. An Automatic Controller Extractor for HDL Descriptions at the RTL[J]. IEEE Computer Design & Test, July-September 2000, 17(3): 72-77.

[4] Weiser M. Program Slicing[J]. IEEE Trans. on Software Engineering, 1984, SE-10(4).

[5] Clarke E M, Fujita M, Rajan P S, Reps T, Shankar S, Teitelbaum T. Program Slicing of Hardware Description Languages[C]. Proc. Conf. on Correct Hardware Design and Verif. Methods, 1999: 298-312.

[6] Vedula V M, Abraham J A, Bhadra J. Program Slicing for Hierarchical Test Generation[C]. Proceedings of the 20th IEEE VLSI Test Symposium (VTS.02), Monterey, California, April 28-May 02, 2002: 237-246.

[7] Sun Microsystems. PicoJava Technology[EB]. http://www.sun.com/microelectronics/communitysource/picojava, 2001.

[8] http://vlsi.colorado.edu/~vis/[EB], 2002.