論文2003-40SC-1-4

Verilog UDP Library의 등가 게이트수준 모델 생성

(Generation of Gate-level Models Equivalent to Verilog UDP Library)

朴 京 俊 *, 閔 炯 福 **

(Kyoung Jun Park and Hyoung Bok Min)

요 약

Verilog HDL의 UDP(User Defined Primitive) 라이브러리는 디지털 회로 설계 과정에서 시뮬레이션을 위해 사용된다. 그러나 합성이 되지 않는 특성으로 인해 이와 등가의 게이트수준 라이브러리를 따로 만드는 데에 많은 시간과 노력이 소요된다. 등가의 게이트수준 모델이 존재하지 않을 경우 이는 테스트 과정에서 고장 검출율을 낮추는 요인이 되므로 등가 게이트수준 모델 생성은 필수적이며 이의 자동화가 필요하다. 이를 위해 매우 복잡한 알고리즘이 발표되기는 했지만 Verilog UDP library의 특성상 보다 더 간단한 알고리즘으로 구현이 가능하다. 알고리즘이 간략해짐에 따라 이를 구현하는 데에 걸리는 시간과 노력이 절약되고 프로그램 실행시간도 크게 줄일 수 있다.

Abstract

UDP library of Verilog HDL has been used for simulation of digital systems. But it takes a lot of time and efforts to generate a gate-level library equivalent to the UDP library manually due to the characteristic of UDP that does not support synthesis. It is indispensable to generate equivalent gate-level model in testing the digital systems because fault coverage can be reduced without the equivalent gate-level models. So, it is needed to automate the process of generating the equivalent gate-level models. An algorithm to solve this problem has been proposed, but it is unnecessarily complex and time-consuming. This paper suggests a new improved algorithm to implement the conversion to gate-level models, which exploits the characteristic of UDP. Experimental results are demonstrated to show the effectiveness of the new algorithm.

Keyword : VLSI, CAD, Verilog, HDL, HDP

Ⅰ. 서 론

Verilog HDL(Verilog Hardware Description language)는 VHDL(Very high speed integrated circuit Hardware Description Language)와 함께 하드웨어 기술언어의 표준으로 이용되고 있다. 두 언어 모두 진리표를 이용한 모듈의 기술을 허용하고 있는데 VHDL에서는 VITAL이, Verilog HDL에서는 Verilog UDP(Verilog User Defined Primitive)가 이에 해당한다. 이 중 Verilog UDP는 주로 작은 규모의 모듈의 기능을 테이블로 기술하여 이를 시뮬레이션에 이용하는데 빠른 시뮬레이션 속도를 보여, 주로 시뮬레이션을 위한 라이브러리에 사용된다[1]. 하지만 자동 테스트 패턴 생성(Automatic test pattern generation)등에 이용하고자 할 때에는 빠른 시뮬레이션 속도에도 불구하고 합성이

* 正會員, 成均館大學校大學院 電氣電子및컴퓨터工學部
(Graduate school of Sungkyunkwan Univ. School of Electrical and computer engineering)

※ 한국 과학재단 과제번호 : 2000-1-30200-002-3
接受日字:2001年10月22日, 수정완료일:2002年12月26日

불가능하기 때문에 시뮬레이션에 이용한 라이브러리와 이 라이브러리를 이용해 설계한 회로를 같은 기능의 게이트수준 모델로 합성하고 다시 합성이 제대로 되었는지 확인하는 과정을 사람의 손으로 직접 해줘야 하므로 이로 인해 개발 시간이 지연되는 불편함이 있었다. 본 논문에서는 이러한 Verilog UDP로 이루어진 라이브러리로부터 같은 기능을 하는 게이트수준 라이브러리를 자동으로 생성하기 위한 알고리즘을 제시한다.

Ⅱ. 등가 게이트수준 라이브러리의 필요성

반도체 산업의 동향을 보면 설계 자동화에 따라 IC의 집적도와 복잡도가 나날이 증가하고 공정이 미세화되고 있으며 이로 인해 칩의 테스트에 소요되는 시간과 경비가 기하급수적으로 증가하고 있다. 최근에는 테스트과정 역시 자동화되고 있지만 게이트수준 모델을 입력으로 받아 이에 대해 테스트 알고리즘을 적용하므로 테스트 과정 자동화를 위해서는 게이트수준의 모델을 합성해내야 한다. Verilog UDP는 빠른 시뮬레이션을 위해 테이블 형식으로 기술되며 합성은 지원되지 않는다. 또한 그 특성상 Verilog UDP로 대규모의 모델을 직접 기술하기보다는 작은 규모의 primitive들을 Verilog UDP로 기술해놓은 라이브러리를 생성하고 이를 이용하는 방식이 주를 이룬다. 따라서 Verilog UDP는 일종의 시뮬레이션 라이브러리를 위한 기술 방식이라 하겠다[1]. 그런데 이와 같이 Verilog UDP를 이용해 시뮬레이션을 거쳐 설계를 마친 회로에 대해 테스트 과정을 수행하거나 게이트수준로 합성하고자 한다면 Verilog UDP 라이브러리로부터 호출된 부분은 합성되지 않고 흑상자로 처리될 것이고[1] 이로 인해 테스트가 더욱 어려워지는 문제가 발생한다. 이를 방지하기 위해 Verilog UDP를 이용해 시뮬레이션 라이브러리를 만들었다면 이와 등가의 게이트수준 라이브러리, 즉 테스트용 라이브러리 역시 만들어야 한다. 그러나 시뮬레이션 라이브러리와 등가의 게이트수준 라이브러리를 만들고 그 결과가 올바른지 formal verification을 하는 데에는 시뮬레이션 라이브러리를 만드는 것보다 더욱 많은 시간이 소요된다. 따라서 이 과정을 자동화 할 수 있다면 설계로부터 테스트를 거쳐 칩이 생산되기까지의 기간을 크게 줄일 수 있게 될 것이다. 이와 관련하여 1998년 ITC에 Peter Whol과 John Waicukauski의 논문[1]이 발표된 바 있다. 해당 논문에서는 먼저 OTDD(Ordered Ternary Decision Diagram)를 생성하고 이를 최적화함으로써 게이트수준 모델을 얻어냈으나 이는 수많은 OTDD 관련 연산을 수행해야 하므로 구현하기에 복잡하고 프로그램 수행시간이 많이 걸릴 뿐만 아니라 메모리를 많이 차지하는 단점이 있었다.

Ⅲ. Verilog User Defined Primitives(UDP)

Verilog UDP는 크게 조합 UDP와 순차 UDP로 나뉜다. 조합 UDP는 다음과 같은 형식으로 구성되며 예제는 2-입력 multiplexer를 나타내고있다.

```
primitive U_MUX_2_1 (Q, A, B, SL);
    output Q;
    input A, B, SL;
// FUNCTION :  TWO TO ONE MULTIPLEXER
   table
    //  A   B   SL  :   Q
        0   0   ?   :   0 ;
        1   1   ?   :   1 ;

        0   ?   0   :   0 ;
        1   ?   0   :   1 ;

        ?   0   1   :   0 ;
        ?   1   1   :   1 ;
   endtable
endprimitive
```

다음은 순차 UDP의 예제로, 1-입력 D-Flip Flop을 나타낸다.

```
primitive U_FD_N   (Q, D, CP);
    output Q;
    input   D, CP;
    reg     Q;
// FUNCTION :  NEGATIVE EDGE TRIGGERED D
FLIP-FLOP ( Q OUTPUT UDP ).
    table

    // D   CP          :   Qt  :   Qt+1
```

```
        1   (10)     :  ?  :  1;  // clocked data
        0   (10)     :  ?  :  0;
        1   (1x)     :  1  :  1;  // reducing
pessimism
        0   (1x)     :  0  :  0;
        1   (x0)     :  1  :  1;
        0   (x0)     :  0  :  0;
        ?   (0x)     :  ?  :  -;  // no change
on rising edge
        ?   (?1)     :  ?  :  -;
        *    ?       :  ?  :  -;  // ignore
edges on data
    endtable
endprimitive
```

Verilog UDP에서는 다음과 같은 제한사항을 두고 있다.

(1) 출력은 오직 1-bit의 포트 1개만을 가진다.

(2) 조합 UDP의 입, 출력 포트에는 0, 1, x, ?의 한 값만이 올 수 있다. x는 unknown 값이고 ?는 0, 1, x 모두를 가질 수 있음을 의미한다.

(3) 테이블에 기술되지 않은 입력의 조합에 대한 출력값은 x로 간주한다.

(4) 순차 UDP의 입력 값에는 (01), (10), (0x), (1x)와 같은 edge의 표현이 가능하며 모든 edge를 표현하는 데에는 ??와 *를 이용한다. edge는 테이블의 한 line에 오로지 한번만 표현될 수 있다.

(5) 순차 UDP의 출력포트는 reg 포트로 선언되어야하며 조합 UDP의 경우와 마찬가지로 1-bit의 포트 한 개만을 가질 수 있다.

(6) 순차 UDP의 출력에는 Q와 Q+1의 두 가지 상태가 있다. Q는 이전의 출력 상태를 나타내고 Q+1은 다음 상태를 나타낸다. 다음 상태가 현재의 상태와 변함이 없다면 다음 상태에 '-'를 이용한다.

Ⅳ. 알 고 리 즘

Verilog UDP로 이루어진 라이브러리를 분석해보면 조합 primitive의 경우는 일반적인 조합 logic의 논리 최적화와 같은 경우이므로 그리 복잡하지 않음을 알 수 있다. 그러나 순차 primitive의 경우는 입력 포트에 edge가 표현될 수 있으므로 기존의 경우와는 다른 접근을 시도해야한다. Wohl의 알고리즘[1]에서는 모든 입력 포트에 대해 OTDD(Ordered Ternary Decision Diagram)를 생성하고 이를 최소화한 뒤 게이트수준 모델을 생성하는 방법을 이용했지만, 이와 같은 알고리즘은 OTDD를 처리하는 과정과 게이트수준 모델 생성 이후 flip-flop을 찾아내기 위해 매우 복잡한 과정을 거쳐야 하는 문제점을 안고 있다. 특히 Input Reordering이라는 과정은 작은 규모의 OTDD에 대해서는 크게 문제가 되지 않지만 OTDD의 규모가 커질수록 그 복잡도가 심각하게 증가하는 문제점을 안고 있다[1,6]. Verilog UDP 라이브러리의 순차 primitive를 분석한 결과 모두 한 개의 메모리 소자로만 표현되고 있음을 발견하게 되었다. Verilog UDP에 오로지 1개의 상태가 존재함을 고려한다면 당연한 결과라 하겠다. 따라서 본 논문에서는 다음 <그림 1>과 같이 순차 primitive에 대해 메모리 소자 한 개를 갖고있는 기본적인 게이트 수준 모델을 생성하고 입력포트들을 각각 reset 포트, clock 포트, data 포트, optional 포트의 네 가지 종류로 나눈 뒤, 이들을 따로 따로 처리하는 방법을 이용하도록 하였다. reset 포트와 clock 포트, 그리고 data 포트에는 그림 상에서는 여러 개의 포트들이 입력됨을 가정하고 있지만 실제로 대부분의 경우에는 한 개의 포트만이 입력으로 할당된다는 사실도 라이브러리의 관찰을 통해 알 수 있다. 위와 같은 방법을 이용하면 몇 가지 이득을 얻을 수 있는데, 첫 번째로 primitive를 처리하는데 있어 복잡도가 감소한다는 점이다. primitive의 입력 포트의 개수는 대략 10개를 넘지 않는데 이를 다시 세 가지 종류의 포트로 나누어 처리하면 평균적으로 2~3개 정도의 포트들만을 처리한다. 한번에 처리하는 포트의 개수를 줄임으로써 프로그램 수행속도와 복잡도를 줄일 수 있다. 두 번째 잇점은 다음 <그림 2>와 같이 프로그램의 순서도가 간단해진다는 점이다.

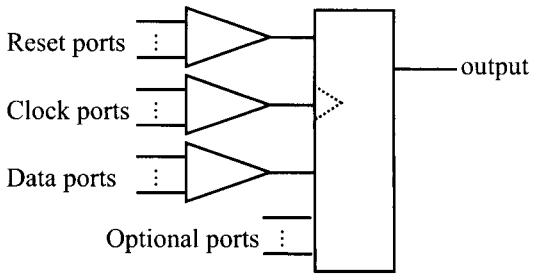


그림 1. 순차 primitive 처리를 위한 기본 모델
Fig. 1. Basic model for processing sequential primitives.

Wohl의 논문[1]에서는 parsing이 끝난 후 조합인 경우와 순차인 경우의 처리 방법이 달라 각각의 처리를 따로 해주는 방법을 취했는데, 본 논문에서는 위의 <그림 1>에서 볼 수 있듯이 순차 모델에서 기억소자와 clock 포트, reset 포트들에 대한 처리를 생략하면 바로 조합 모델이 된다. 따라서 프로그램의 순서도는 다음 <그림 2>와 같다. 다음으로 기존의 게이트수준 라이브러리의 활용이 가능하다는 점을 들 수 있다. 기본 모델을 생성할 때에 기존에 존재하는 라이브러리의 cell들과 연결을 시켜주게 되므로 특정 timing 특성을 필요로 하는 게이트수준 모델 생성 시 복잡한 과정 없이 바로 해당 라이브러리로의 전환이 가능하다.

이와 같은 알고리즘은 O(n)의 시간 복잡도를 가지며 n은 입력 포트의 갯수이다. 전반적으로 모든 연산은 각 포트에 입력되는 값과 이에 따른 출력을 판단의 기본으로 하고있기 때문에 이와 같은 시간 복잡도를 갖는다. 그러나 일반적으로 시간 복잡도는 n->∞인 경우 해당 알고리즘의 우수성을 판단하기 위해 존재하지만 본 논문과 같이 n이 실제적으로 10을 넘지 않는 특수한 경우에 있어서는 그 의미가 크지 않다.

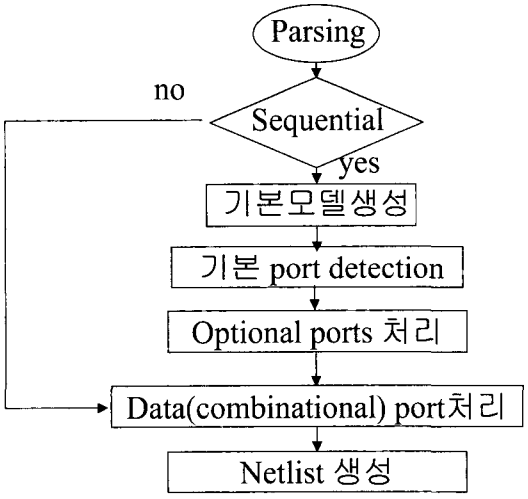


그림 2. 개선된 순서도
Fig. 2. Improved flowchart.

본 논문의 알고리즘을 좀 더 구체적으로 기술하기 위하여 포트별 처리 방법을 기술하도록 하겠다.

1. 기본 포트에 대한 처리

일단 기본 모델이 생성되고 나면 각각의 포트들에 대한 처리를 하게 되는데, 먼저 모든 primitive에 공통적인 기본 포트들에 대한 처리를 한다. 기본 포트에는 clock 포트와 set, reset 포트, 그리고 clock enable 포트가 속한다. 이중에서 우선 clock 포트를 찾아내는 과정을 거친다.

Verilog UDP의 문법적인 특성상 clock이 enable되지 않은 상태일 때에는 출력에 '-'가 출력되게 되는데 이를 clock의 "off 상태"라 하며 이전의 상태와 변함이 없음을 의미한다. 역으로 clock 포트가 enable 되기 위해서는 clock의 off 상태를 만족하지 않는 값들이 clock 포트에 인가되어야 하는데 이를 "on 상태"라 한다. 출력이 on 상태이고 입력이 '0', '1', 'rising edge', 'falling edge'인 포트를 clock 포트 가능성이 있는 포트로 분류한다. 이미 clock 포트로 분류 되어있었다 하더라도 이후 위의 조건을 만족하지 않으면 해당 포트는 clock 포트로서의 자격을 상실한다. 이 과정을 테이블의 모든 부분에 대해 반복하면 최종적으로 clock 포트를 얻을 수 있다.

clock 포트가 정해지면 clock 포트에 대한 입력 값이 on 상태를 만족함에도 출력이 그렇지 않은 경우, clock enable 포트를 찾아낼 수 있다. 이와 같은 과정으로부터 clock 포트와 clock enable 포트들의 부울 대수가 결정되어 clock 포트에 대한 처리가 끝난다.

set, reset 포트는 그 동작의 특성상 clock 포트를 override하게 되고 off 상태에서는 그 입력 값이 active한 값일 수 없다는 점을 이용하면 처리할 수 있다. 처음부터 set, reset 포트를 구분해서 처리하기보다는 우선 clock 포트를 override하고 off 상태에서는 active하지 않은 값만을 갖는 포트를 찾아내 set, reset 포트 중 하나로 간주하고 다시 이를 set, reset 포트로 분류하는 순차적인 방법을 이용하면 구현이 간단해진다.

2. optional 포트에 대한 처리

clock 포트와 reset 포트에 대한 처리가 끝나면 남는 포트들은 data 포트와 optional 포트로 간주된다. optional 포트들은 각 primitive의 특성을 나타내는 포트들로 그 동작 특성에 따라 처리가 가능하다. 예를 들어 clocked scan cell의 경우 원래의 clock 포트 이외에 또 하나의 clock 포트를 찾아내고 각 clock 포트에 대한 data 포트를 찾아내 이를 연결 지어주면 된다. 일단 clock 포트와 set, reset 포트에 대한 처리가 끝나고 나면 optional 포트에 대한 처리는 훨씬 수월해지므로 각각의 종류에 맞는 적합한 알고리즘을 구현하기는 쉬워

진다.

data 포트는 다른 모든 포트들이 처리 된 이후 처리하는데 역시 대부분의 경우 오로지 1개의 포트가 할당된다. 순차 primitive에서 data 포트에 1개 이상의 포트가 할당되는 경우가 1가지 있는데 바로 muxed scan cell인 경우이다. 이 경우 본 논문에서는 heuristic 알고리즘을 적용하여 처리하였다. 다른 포트들이 존재하지 않을 경우 조합 primitive의 처리와 같으므로 이 부분에 해당 알고리즘을 구현하면 muxed scan cell을 비롯한 모든 primitive를 처리할 수 있는 tool의 구현이 가능해지지만 앞에서 밝혔듯 조합 primitive를 위한 특별한 알고리즘의 개발이 목적은 아니므로 본 논문에서는 이를 생략하였다.

이상의 내용을 바탕으로 프로그램의 pseudo-code를 작성하면 다음과 같다.

```
procedure main is
begin
   while not_end_of_file()
      parse_next_primitive();
      separate_ports();
      write_gate_level_network();
   end-while
end main
```

그림 3. 프로그램 시작 및 기본 모델 생성
Fig. 3. Basic model generation.

프로그램의 시작 부분이며 기본 모델을 생성하는 부분이다. 기본 모델 생성은 코드가 실제로 존재하는 것이 아니고 단순히 프로그램 전개상 필요한 부분이다. 순서도에 있어 최 상위부분에 해당한다.

```
procedure separate_ports is
begin
   find_clock_ports();
   find_reset_ports();
   distinguish_set_and_reset_ports();
   if clock_is_a_level_clock then
      process_level_primitive();
   else
      process_edge_primitive();
   end-if
   find_optional_ports();
   record_port_types();
end separate_ports
```

그림 4. 입력포트의 분할
Fig. 4. Input ports separation.

기본 모델 생성 이후 순서도의 각 block 부분에서 처리하는 여러 종류의 포트들을 각각 순서에 따라 분리해내는 부분이다.

```
procedure find_clock_ports is
begin
   for every_state_transition
      if on_state then
         for every_ports
            if port_has_valid_value then
               if    state_transition_is_toggled
               then

candidate_as_toggle_clock_port();
               else
                  candidate_as_clock_port();
               end-if

record_port_value_for_clock_type();
            end-if
         end-for
      end-if
   end-for

   for every_state_transition
      if off_state then
         if     port_is_candidated_as_clock_port
         and port_is_active then
            cancel_candidated_port();
            candidate_as_data_port();
         end-if
      end-if
   end-for
end find_clock_ports
```

그림 5. clock 포트 detectoion
Fig. 5. Clock ports detection.

clock 포트를 찾아내는 부분이다. 순서도의 기본 포트 detection중 clock 포트 detection에 해당한다. 출력값이 on 상태인 경우와 off 상태인 경우에 따라 각기 다른 처리를 한다. 한번에 모든 포트들의 속성을 찾아낼 수는 없으므로 각 종류별로 해당 포트일 가능성이 있는 포트들을 골라내고 해당 포트가 아닐 경우 이를 제거하는 기법을 사용한다.

set 포트 및 reset 포트를 찾아내는 부분이다. 처음부터 set 포트와 reset 포트를 구분하려면 복잡한 과정을 거쳐야하므로 처음에는 단순히 clock 포트를 지배하는 포트를 set 포트, 혹은 reset 포트로 분류하고 이후에 그 속성에 따라 다시 분류하는 방법으로 처리한다. clock 포트와 마찬가지로 일단 가능성 있는 포트를 찾아내고 set 혹은 reset 포트가 아닌 것이 확실한 경우

이를 탈락시키는 기법을 이용한다.

```
procedure find_reset_ports is
begin
   for every_state_transition
      if on_state then
         if  clock_port_is_not_working_or_don't_
         care then
            for every_ports_with_valid_ value
               candidate_as_reset_port();
            end-for
            record_port_value_for_reset_type();
         end-if
      end-if
   end-for

   for every_state_transition
      if off_state
         for every_ports
            if  port_is_candidated_as_reset_port
            and port_is_active then
               cancel_candidated_port();
               candidate_as_data_port();
            end-if
         end-for
      end-if
   end-for
end find_reset_ports
```

그림 6. set(reset) 포트 detection
Fig. 6. Set(reset) ports detection.

```
procedure find_optional_ports is
        determine_primitive_type();
        find_proper_type_of_ports();
end find_optional_ports
```

그림 7. 부가적인 포트 detection
Fig. 7. Optional ports detection.

clock 포트와 reset 포트, 그리고 data 포트 이외의 부가적인 포트를 가진 primitive에 대한 처리를 하는 부분이다. 이미 기본적인 포트에 대한 처리가 끝났기 때문에 남은 포트의 작동 특성을 관찰하여 그 종류를 판단하고 이를 다시 active high, active low로 구분하거나 scan cell의 scan data 포트와 같이 종속적인 포트들을 찾아 연결 시켜주면 처리가 끝난다.

V. 실 험

이상과 같은 알고리즘을 구현하여 다음의 환경 하에서 실험을 수행하였다.

i. Compiler : GNU C compiler
ii. CPU : Intel Pentium MMX 200 processor
iii. Memory : 32MB
iv. OS : LINUX
v. 실험 대상 UDP 라이브러리 : cadence 라이브러리에 포함된 UDP 라이브러리

알고리즘을 구현하여 실행한 결과 302개의 primitive 중 259개의 순차 primitive를 처리하는 데에 0.51초가 소요되었다. 따라서 한 개의 primitive에 대해 소요된 처리시간은 1.96ms가 된다. 기존의 알고리즘이 순차 primitive 한 개당 23ms가 소요됐음을 감안하면 수행시간이 매우 감소했음을 알 수 있다. 실험 환경과 결과에 있어 기존 알고리즘과 비교를 하자면 다음 <표 1>과 같다.

표 1. 실험 환경 및 결과 비교
Table 1. Camparison of experimental environment and result

	기존 알고리즘	새로운 알고리즘
CPU	200MHz Pentium pro	Pentium MMX 200MHz
OS	Windows NT 4.0	LINUX
사용 언어	C++	C
평균 계산 시간	23msec	1.96msec

기존의 알고리즘과 같은 환경 하에서 실험이 수행되었다면 더 완전한 비교가 가능했겠지만 완전히 같은 환경을 만들기는 어려워 위와 같은 환경에서 실험을 수행하였다. 하지만 실험 환경 때문에 결과에 큰 영향을 줄 정도로 새로운 알고리즘을 수행한 환경이 크게 뛰어난 것은 아니며, 유사한 환경에서 1/10 이하로 프로그램 수행 시간이 줄어들었음을 알 수 있다.

처리한 primitive중 90.7%에 해당하는 235개에 대해서는 올바른 게이트수준 모델을 얻을 수 있었다. 24개에 대해서는 올바르지 못한 결과를 얻게 되었는데 이는 level-sensitive primitive에서 clock 포트를 제대로 찾지 못함으로 인해 발생하는 문제였다.

다음은 primitive로 기술된 negative edge triggered d flip-flop을 나타낸다.

```
primitive U_FD_N_CE_RB_NO (Q, D, CP, RB, CE, NOTIFIER_REG);
    output Q;
    input  NOTIFIER_REG,
           D, CP, RB, CE;
    reg    Q;
// FUNCTION : NEGATIVE EDGE TRIGGERED D FLIP-FLOP WITH ACTIVE LOW
//            ASYNCHRONOUS CLEAR and ENABLE ( Q OUTPUT UDP ).
    table
    //  D   CP      RB   CE      NOTIFIER_REG  :   Qt  :   Qt+1
        1   (10)    1    1          ?          :   ?   :   1;  // clocked data
        0   (10)    1    1          ?          :   ?   :   0;
        1   (10)    1    x          ?          :   1   :   1;  // clocked data
        0   (10)    1    x          ?          :   0   :   0;
        0   (10)    x    1          ?          :   ?   :   0;  // pessimism
        0    ?      x    1          ?          :   0   :   0;  // pessimism
        1    1      x    1          ?          :   0   :   0;  // pessimism
        1    x    (?x)   1          ?          :   0   :   0;  // pessimism
        1    0    (?x)   1          ?          :   0   :   0;  // pessimism
        x    1      x    1          ?          :   0   :   0;  // pessimism
        x    x    (?x)   1          ?          :   0   :   0;  // pessimism
        x    0    (?x)   1          ?          :   0   :   0;  // pessimism
        0    ?      x    x          ?          :   0   :   0;  // pessimism
        1    1      x    x          ?          :   0   :   0;  // pessimism
        1    x    (?x)   x          ?          :   0   :   0;  // pessimism
        1    0    (?x)   x          ?          :   0   :   0;  // pessimism
        x    1      x    x          ?          :   0   :   0;  // pessimism
        x    x    (?x)   x          ?          :   0   :   0;  // pessimism
        x    0    (?x)   x          ?          :   0   :   0;  // pessimism
        1   (1x)    1    1          ?          :   1   :   1;  // reducing
pessimism
        0   (1x)    1    1          ?          :   0   :   0;

        1   (x0)    1    1          ?          :   1   :   1;
        0   (x0)    1    1          ?          :   0   :   0;
        1   (1x)    1    x          ?          :   1   :   1;  // reducing
pessimism
        0   (1x)    1    x          ?          :   0   :   0;

        1   (x0)    1    x          ?          :   1   :   1;
        0   (x0)    1    x          ?          :   0   :   0;
        ?    ?      0    ?          ?          :   ?   :   0;  // asynchronous
clear
        ?   (?0)    1    0          ?          :   ?   :   -;  // chip is not
enabled
        ?   (1x)    1    0          ?          :   ?   :   -;  // chip is not
enabled
        ?   (?0)    x    0          ?          :   0   :   0;  // chip is not
enabled pessimism with reset
        ?   (1x)    x    0          ?          :   0   :   0;  // chip is not
enabled pessimism with reset
        ?    ?    (?x)   0          ?          :   0   :   0;  // chip is not
enabled pessimism with reset
        ?   (?1)    ?    ?          ?          :   ?   :   -;  // ignore falling
clock
        ?   (0x)    ?    ?          ?          :   ?   :   -;  // ignore falling
clock
        *    ?      ?    ?          ?          :   ?   :   -;  // ignore the edges
on data
        ?    ?    (?1)   ?          ?          :   ?   :   -;  // ignore the edges
on clear
        ?    ?      ?    *          ?          :   ?   :   -;
        ?    ?      ?    ?          *          :   ?   :   x;
    endtable
endprimitive
```

위의 primitive 모델에 대해 다음과 같은 게이트수준 모델의 출력을 얻었다.

```
module U_FD_N_CE_RB_NO(Q, D, CP, RB, CE, NOTIFIER_REG);
 input    D, CP, RB, CE, NOTIFIER_REG;
 output   Q;
 reg      Q;
 wire     CP_inv, CE_inv, RB_inv;
 inv(CP_inv, CP)
 inv(CE_inv, CE)
 inv(RB_inv, RB)
 DFF_CK_D_CE_RESET(Q, CP_inv, D, CE_inv, RB_inv)
endmodule
```

VI. 결 론

지금까지와 같은 알고리즘을 이용해 프로그램을 작성한 결과를 정리해보면 다음과 같다.

1. 구현하기에 쉽고 간단하다.

기존의 알고리즘은 게이트수준 모델을 먼저 생성하고 다시 이로부터 메모리 소자를 찾아내는 과정을 거쳐야하므로 구현하기가 매우 까다롭고 오랜 시간이 걸리며 많은 노력을 필요로 한다는 단점을 갖고있었다. 본 논문의 알고리즘은 이와 같은 단점을 먼저 기본 모델을 생성하여 처리하는 방법으로 간략화 함으로써 극복하는 해결책을 제시하였다.

2. 프로그램의 실행속도가 크게 줄어든다.

이 수행시간에는 다량의 디버깅 코드 출력시간이 포함되어있고 그 90%이상이 라이브러리 파일로부터 정보를 읽어들이는 parsing 과정임을 감안하여 순수하게 논문의 알고리즘을 위한 수행시간만을 비교한다면 속도면에서 매우 뛰어난 알고리즘임을 알 수 있다. 수행 속도 절감의 원인으로는 무엇보다도 OTDD의 처리에 필요한 많은 부분의 생략을 들 수 있다. OTDD는 주로 동적인 메모리 할당을 통해 구현하게 될 뿐만 아니라 입력 포트 1개가 늘어날 때마다 그 node 수가 기하급수적으로 늘어나게 되며, 다시 이를 최적화하고 관찰하는 데에 부가적으로 매우 복잡한 알고리즘이 필요하므로 프로그램 수행 시간을 크게 늘이는 원인이 된다. 본 논문의 알고리즘은 이와 같은 최적화의 과정과 메모리 소자를 찾아내는 과정이 필요 없고 단지 포트 separation만으로도 프로그램 수행 시간을 크게 줄이는 효과를 낼 수 있다는 점에 착안, 최종적인 결과를 얻어내 프로그램 수행시간을 크게 단축하게 되었다. Wohl의 알고리즘[1]과의 수학적 비교를 할 수 있다면 더 완전하게 비교가 되겠으나 Wohl의 논문[1]에 수학적 알고리즘 성능 검증 내용이 없어 이를 수행할 수 없었다.

3. 메모리를 절약할 수 있다.

이 장점 역시 기본 모델의 생성으로부터 그 원인을 찾을 수 있다. OTDD는 구현하기 복잡하고 시간이 많이 걸릴 뿐만 아니라 많은 메모리를 필요로 한다. 또한 포트가 1개 증가할 때마다 그 복잡도와 필요한 메모리의 용량이 기하급수적으로 늘어난다. 그러나 새로운 알고리즘은 포트들을 그 종류에 따라 따로따로 처리하게 되므로 필요한 메모리의 양을 크게 줄일 수 있다.

새로운 알고리즘의 문제점으로는 다음을 지적할 수 있다.

우선 전체 primitive 중에서 90%에 대해 올바른 게이트수준 모델을 얻을 수 있었다. 올바르지 못한 게이트 수준 모델을 얻게되는 비율이 기대치보다는 높다는 문제점을 안고 있는데 이는 level-sensitive primitive에 대한 처리율이 저조함이 주요 원인이다. 그러나 이는 Whol의 알고리즘이 조합 primitive에 대한 알고리즘만을 공개됐을 뿐 순차 primitive에 대해서는 완전히 공개되지 않아 이를 정확히 재현할 수 없는 데에서 기인한다. 본 알고리즘이 Whol의 알고리즘과 차별화 하고자 하는 부분은 프로그램의 전체적인 처리 방법을 개선함으로써 구현 및 실행 시간을 줄이고자 함이었으며 따라서 level-sensitive primitive의 clock 포트 처리 방법에 관한 부분은 Whol의 알고리즘을 적용하면 프로그램의 실행시간과 성공률을 모두 높일 수 있다.

참 고 문 헌

[1] Peter Whol and John Waicukauski, "Extracting gate-level networks from simulation tables", ITC pp. 622~631, 1998.

[2] Samir Palnitkar, "Verilog HDL A Guide to Digital Design and Synthesis", Prentice Hall 1998.

[3] 조경순, "Verilog HDL 이론 및 실습", 한양대학

교 반도체 설계교육 지역센터, 1999.

[4] Shin-ichi Minato, "Binary Decision Diagrams and Applications for VLSI CAD".

[5] Sheldon B. Akers, "Binary Decision Diagrams", IEEE TRANSACTIONS ON COMPUTERS VOL. c-27, NO. 6, pp. 509~516, JUNE 1978.

[6] Masahiro Fujita, "Variable Ordering Algorithms for Ordered Binary Decision Diagrams and Their Evaluation", IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, VOL. 12, NO. 1, pp. 6~12, JANUARY 1993.

[7] Youpyo Hong, "Safe BDD Minimization Using Don't Cares", Design Automation Conference 1997, pp. 208~213.

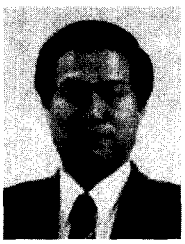
[8] Giovanni De Micheli, "Synthesis and Optimization of Digital Circuits", McGraw Hill 1998.

[9] John R. Levine, Tony Mason and Doug Brown, "Lex & Yacc", O'Reilly & Associates, 1990.

저 자 소 개

朴 京 俊(正會員)

2001~현재. LG전자(주) 디지털 디스플레이 연구소 주임연구원. 1999~2001. 성균관대학교 전기전자 및 컴퓨터공학부(공학석사). 1994~1999. 성균관대학교 공과대학 전기공학과(공학사)



閔 炯 福(正會員)

1991~현재. 성균관대학교 전기전자 및컴퓨터공학부 교수. 1987~1990. The University of Texas at Austin (Ph.D.). 1987~1988. Intelligent Signal Processing, Inc : Software Engineer. 1985~1986. Neuro Institute, Columbia University : Research Staff. 1982~1985. 금성통신(주) 연구소 : 주임연구원. 1980~1982. 한국과학기술원 전기 및 전자공학과 (공학석사). 1976~1980. 서울대학교 공과대학 전자공학과 (공학사)