

論文95-32A-1-26

모듈 합성을 위한 비아 겹침이 없는 미로 배선

(Non-Stacked-Via Maze Routing for Module Generation)

權成勳 *, 吳命燮 *, 申鉉哲 **

(Sung Hoon Kwon, Myoung Sub Oh, and Hyun Chul Shin)

요약

효과적인 모듈 합성을 위해서는 여러 층의 복잡한 배선 허용 영역에서 다양한 제약조건을 만족시키며 배선 할 수 있는 세부 배선기가 필요하다. 본 논문에서는 모듈 합성에서의 세부 배선 단계에서 비아의 겹침을 허용하지 않고 최단 경로를 찾아서 배선하는 새로운 미로 배선 기법을 기술하였다. 이 방법에서는 비용을 이용하여 네트의 경로를 찾을 때, 각 그리드 점마다 두 개의 비용 값을 저장하는 데이터 구조를 사용하였다. 각 점에서의 비용 값은 배선하고자 하는 네트의 단자에서 그 점까지의 배선 비용을 나타내는데, 하나의 비용은 현재의 점에 비아를 만들지 않는다는 조건 하에서의 최단거리 비용을, 또 하나의 비용은 현재의 점에 비아를 만들며 도달했을 때의 최단 거리 비용을 간직한다. 제안한 배선 기법은 비아 겹침 없이 최소 비용의 미로 배선을 할 수 있는 최초의 체계적인 방법이다.

Abstract

For effective module generation, a detailed router which can handle complicated routing regions on multiple layers of interconnection under various constraints is necessary. In this paper, a new improved maze routing technique is described, which can find the shortest path for each net without allowing stacked vias. In this method, two cost values are stored at each grid point. The cost values represent the routing costs from the terminal of the net being routed to the grid point. One cost value shows the cost of the shortest path without making a via at the grid point and the other cost value shows that with making a via at the grid point. This is the first systematic maze routing technique which can find the shortest path without via-stacking.

I. 서 론

집적회로 공정 기술의 발달로 인하여 대규모의 복잡한 회로가 하나의 집적회로에 구현 가능해졌으며, 근래

*準會員, **正會員, 漢陽大學校 電子工學科

(Dept. of Elec. Eng., Hanyang Univ.)

※ 본 논문은 1993-1994년도 한국전자통신연구소의 연구 지원으로 연구되었음.

接受日字 : 1994年 2月 3日

에도 하나의 집적회로에 포함될 수 있는 트랜지스터의 개수는 3년에 2배 정도의 속도로 증가하고 있다. 따라서 효율적으로 원하는 기능의 모듈(module) 또는 셀(cell)을 자동 설계하는 모듈 합성기가 혼히 사용된다^{[1][2]}. 집적회로의 설계 · 제조에서의 또 하나의 변화는 신호의 연결을 위하여 점차 많은 층(layer)의 메탈(metal)을 사용한다는 것이다. 집적도가 증가하여 한 집적회로 내의 소자 수가 많아지면 이들을 적절히 연결하는 배선에 대한 요구도 많아지기 때문에, 주

어진 수의 메탈 층을 효과적으로 사용하여 최소의 면적에 원하는 사양을 만족시키도록 하는 배선 과정이 집적회로 설계에서 중요한 부분이 되었다. 모듈의 합성은 트랜지스터 수준의 네트 리스트(net list)를 입력으로 하여 주어진 회로의 레이아웃을 자동 생성하는 과정으로, 트랜지스터의 클러스터링(clustering : pairing 및 chaining 포함), 배치, 배선 등의 과정으로 이루어진다. 흔히 모듈 합성은 심볼릭(symbolic)하게 그리드(grid) 상에서 레이아웃을 생성한 후에, 이를 컴팩터(compact) 등을 이용하여 지오메트릭(geometric)하게 변환시키는 과정을 거친다.

본 논문에서는 모듈 합성을 위한 여러 층에서의 세부 배선 특히 비아(via)의 겹침을 허용하지 않는 새로운 미로(maze) 배선 기법을 기술한다. 현재까지 비아 겹침이 없는 미로 배선에 대한 연구는 전혀 없는 것으로 알고 있으며, 본 논문에 기술한 배선 기법은 복잡한 배선 영역에서 비아의 겹침을 허용하지 않는 체계적인 방법으로, 모듈 합성에 매우 유용하다. 먼저 모듈 합성에서의 세부 배선의 특성을 살펴보면 다음과 같다.

1. 배선 가능한 영역이 복잡하다.

예를 들어서 폴리(polysilicon)층에서의 배선은 트랜지스터 또는 기타 장애물이 없는 영역에 한하여 신호 선의 배선이 가능하다. 메탈(metal)층에는 많은 단자(terminal)가 있으며, 또한 불규칙한 트랜지스터의 폭(width)도 배선 영역을 복잡하게 만든다.

2. 트랜지스터 위로 다른 메탈 신호 선이 지나가는 것을 허용하는 경우도 있고 허용하지 않는 경우도 있다. 후자인 경우에는 트랜지스터 위에 장애물이 있는 것으로 처리해야 한다.

3. 비아(via)를 만들 수 있는 위치에도 제약이 있다.

예를 들어, 폴리 신호선 상에서는 메탈 1 층으로의 비아를 만들 수 있지만, 트랜지스터의 게이트(gate) 폴리 위치에서는 수율(yield)을 높이기 위하여 메탈 1 층으로의 비아를 허용하지 않는다.

4. 집적회로 칩의 전체 배선에 비하여 모듈 합성에서는 배선 영역이 작다.

따라서 어느 정도 복잡도가 높은 알고리듬의 사용이 가능하다. 위와 같은 특징으로 인하여 모듈 합성에서의 배선을 위해서는 미로(maze) 배선이 적합하다고 생각

된다^[3]. 미로 배선은 신호를 연결할 수 있는 경로(path)가 있지만 반드시 가장 짧은 경로를 찾을 수 있다는 장점으로 인하여 널리 사용되고 있다. 미로 배선의 단점으로는 배선 영역의 그리드(grid)들을 저장하기 위하여 많은 기억 용량을 필요로 하고, 배선 결과가 배선 순서에 의존적이라는 점이다. 배선 순서에의 의존을 적게하기 위해서는 배선의 변환(routing modification) 또는 재배선(riput and rerouting)을 주로 사용하며^[4]. 모듈 또는 셀 합성에서는 한 번에 하나의 모듈 또는 셀만을 대상으로 하므로 기억용량에 대한 소요도 대부분 지나치게 과다하지 않다. 생성하고자 하는 모듈의 크기도 점차 커지지만, 컴퓨터의 기억용량도 증가하고 있으므로, 모듈 또는 셀 생성에서의 미로 배선의 사용은 앞으로도 합리적일 것으로 보인다.

II 장에서는 비아 겹침 문제를 고찰하고, III 장에서는 비아 겹침을 허용하지 않는 새로운 배선 기법을 설명한다. IV 장에서는 간단한 예제 회로의 배선 과정과 실험결과를 보여 준다. 끝으로 V 장에서는 결론을 기술하였다.

II. 세부 배선에서 비아의 겹침 문제

비아의 겹침은 다음과 같이 설명될 수 있다. 평면에서의 좌표가 (i, j)이고 k번째 층에 위치하는 점을 (i, j, k)로 표시하기로 하자. 점 (i, j, k - 1)과 점 (i, j, k)가 같은 네트로 비아를 이용하여 연결되고, 또 점 (i, j, k)와 점 (i, j, k + 1)이 같은 네트로 비아를 통하여 연결되면 (i, j, k) 위치에서는 두 개의 비아가 연속하여 존재하게 되므로 비아의 겹침이 발생되었다고 한다.

비아의 겹침(via stacking)은 수율(yield)을 크게 떨어뜨린다. 예를 들어 k-1층과 k층을 연결하는 비아를 만들면, k층 면의 비아 위치에서 굴곡이 생기게 되어, 그 위에 k층과 k+1층을 연결하는 비아를 만들면 수율이 감소된다. 이러한 이유로 k층 면을 평면화 한 후에 k와 k+1층간의 비아를 만드는 방법도 있으나, 이 경우에는 공정(processsing)의 비용이 증가한다. 따라서 비아의 위치가 겹치지 않도록 배선하는 새로운 배선 방법이 필요하다.

일부의 다층 배선 방법^[5]에서는 비아의 겹침을 허용하거나 편을 원하는 층에 위치시킬 수 있다고 가정하여 다층 배선 문제를 간단화하였다. 그러나 이러한 가정은 모듈 합성에는 적합하지 않다. 일반적인 비아의 겹침을 허용하지 않는 미로 배선 기법은 현재까지

발표되지 않은 것으로 알고 있으며, 대부분 휴리스틱 알고리듬으로 처리한다.

여러 층의 미로 배선에서 비아의 겹침을 허용하지 않는 간단한 방법으로 경로 탐색 과정에서 비아의 위치 에서는 또 다른 비아를 허용하지 않는 방법을 생각할 수 있다. 즉 미로 배선의 탐색 과정 중에, 한 점에서 비아가 발생하였다면 그 점에서의 이웃하는 층으로의 또 다른 비아는 허용하지 않는 방법이다. 이 경우에는 한 점에서 비아가 형성되면 그 점에서는 또 다른 비아 형성이 이루어질 수 없으므로, 비아의 위치에 따라서 배선 되는 경로가 달라질 수 있다. 이 방법의 단점은, 실제로 네트의 배선 경로가 존재하는 경우에도 비아의 위치를 임의로 정하기 때문에 배선 경로를 찾지 못하거나 최단 거리가 아닌 경로를 찾을 수 있다는 점이다. 일반적으로 한 층에서 이웃하는 층으로의 비아를 만들 수 있는 위치는 여러 점이기 때문에 탐색 과정에서 어느 점을 선택하는 것이 최소 비용으로 비아 겹침이 없는 경로를 얻을 수 있는지는 쉽게 결정할 수 없다.

III. 비아 겹침을 허용하지 않는 새로운 미로 배선 방법

본 장에서는 비아 겹침을 허용하지 않고 최소 비용으로 배선을 하기 위한 새로운 개선된 미로 배선 기법을 설명한다. 1 절에서는 전체 배선 알고리듬을 설명하고, 2 절에서는 비아 겹침을 허용하지 않는 미로 배선을 설명한다. 3 절에서는 본 논문에서 제시한 배선 기법이 비아의 겹침이 없는 최소 비용의 경로를 찾는다는 것을 증명한다.

1. 전체 배선 알고리듬

전체의 미로 배선 알고리듬은 다음의 그림 1과 같다. 이 과정의 수행이 끝나면 모든 네트의 배선은 완료된다.

배선 알고리듬은 초기에 각 네트를 스케줄(schedule)하는 부분과 스케줄된 각 네트들에 대하여 반복적으로 경로를 형성하는 두 부분으로 구성된다. 초기에 네트 n을 스케줄 하는 과정을 설명한다. 먼저 네트 n에 대하여 Find_Groups에서 네트 n의 연결된 소자 수(connected component 수)를 구한다.

초기에는 각 단자가 전혀 연결되지 않았으므로 Find_Groups는 단자의 수를 리턴 한다. 연결된 소자의 수가 2 이상이면 네트 n의 배선이 완료되지 않은 것이므로 Find_Path에서 최소 비용의 경로를 찾는다.

Read data & prepare routing environment:

```
/* Initially, schedule all the nets */
for ( each net n ) {
    num_groups = Find_Groups ( n );
    if ( num_groups >= 2 ) {
        if ( Find_Path ( n ) == YES )
            Schedule ( n );
        else {
            Report_Error;
            exit;
        }
    }
}

/* main routing loop */
while ( ( n = Pop_Schedule () ) != EMPTY ) {
    Try_Tmppath ( n );
    num_groups = Find_Groups ( n );
    if ( num_groups >= 2 ) {
        if ( Find_Path ( n ) == YES )
            Schedule ( n );
        else if( stopping_criterion ) {
            Report_Error;
            exit;
        }
    }
    else
        Ripup and rerouting;
}
}
```

그림 1. 전체 배선 알고리듬

Fig. 1. Global routing algorithm.

경로를 찾을 때는 네트 n의 연결된 소자 중에 임의의 한 소자에서 시작하여 다른 소자를 찾을 때까지 탐색한다. 네트 n의 두 소자를 연결하는 경로를 찾았으면 그 경로의 비용 순서로 스케줄하고, 경로를 찾지 못하였으면 초기에 입력으로 준 배선 영역 내에서 배선이 불가능한 경우이므로 에러를 출력한다. 이 과정에서 배선되어야 할 각 네트들은 모두 스케줄 된다.

다음은 스케줄된 네트들에 대하여 반복적으로 실제의 경로를 형성하는 과정을 수행한다. 먼저 Pop_Schedule에서 스케줄된 네트들 중에 비용이 가장 작은 네트 n을 꺼낸다. Try_Tmppath에서는 스케줄할 때의 경로가 현재까지 이용 가능하면 이 경로로 배선을 수행한다. 네트 n이 스케줄될 때 경로가 있었다고 하더라도 다른 네트가 먼저 그 경로를 사용하여

배선되었을 때에는 네트 n은 다시 스케줄 된다. 한 네트의 경로를 찾아 실제로 배선을 하면 그 네트의 소자 수는 하나 감소하게 된다. Try_Tmppath에서의 배선의 성공 여부에 관계없이, 네트 n의 소자 수를 구하여 소자 수가 2 이상이면 Find_Path에서 다시 네트 n의 경로를 찾는다. 경로가 있으면 네트 n을 다시 스케줄하고, 경로가 없으면 일정 회수 ripup과 재배선을 수행하며, 그 후에도 경로가 없으면 어려를 출력한다. Ripup과 재배선은 [4] 등의 방법을 사용한다. Ripup과 재배선은 다른 어느 방법을 사용해도 무방하다. 이 과정은 스케줄된 네트가 없을 때까지, 즉 소자 수가 2 이상인 네트가 존재하지 않을 때까지 반복적으로 수행된다.

2. 비아 겹침을 허용하지 않는 미로 배선

미로 배선 기법은 경로가 존재하면 반드시 최소 비용의 경로를 찾는다.

```
Find_Path ( n )
/* a path is to be found for net n */
{
    Initialize cost;
    for (positions (i,j,k) of all the pins or existing
    paths of net n) {
        if (there is a via at (i, j, k)) flag = B;
        else flag = A;
        Search_Next (flag, n, i, j, k, cost);
    }
    while (cost < Limit_Cost) {
        /* Points to be searched are pushed in
        * Search_Next, and popped
        * by Pop_Searchq. Pop_Searchq returns a
        pushed net of cost.
        * if there is one.
        */
        if ((queue = Pop_Searchq (cost)) == EMPTY)
        {
            cost++;
            continue;
        }
        /* search at queue position */
        if (Search_Next (queue.flag, queue.n, queue.i,
        queue.j, queue.k, cost)
            == YES) {
            Back_Trace(queue.flag, queue.n, queue.i,
            queue.j, queue.k, cost);
            return (YES);
        }
    }
    return (NO);
}
```

그림 2. 경로 찾는 알고리듬

Fig. 2. Algorithm for finding a path

비아의 겹침이 없는 최소 비용의 경로를 찾기 위하여 각 그리드 점에 두 개의 비용을 저장할 수 있도록 하였다. 점 (i, j, k)에서의 두 비용을 $A(i, j, k)$ 와 $B(i, j, k)$ 로 나타낸다. 따라서 본 방법은 비아의 겹침을 허용하는 기준의 미로 배선에 비하여 2 배의 기억 소자(memory element)를 사용한다. 각 그리드

점에서의 비용은 배선하고자 하는 네트의 소자에서 (i, j, k)점까지의 최소 비용을 나타내는데, $A(i, j, k)$ 은 (i, j, k)점에 비아를 만들지 않고 (i, j, k)까지 배선 할 때의 최소 비용을 나타내며, $B(i, j, k)$ 는 점 (i, j, k ± 1)에서 점 (i, j, k)로 비아를 만들며 배선할 때의 최소 비용을 나타낸다.

전체 배선 과정 중에서 비아의 겹침을 허용하지 않고 네트 n의 경로를 찾는 알고리듬은 다음의 그림 2와 같다. Find_Path 는 네트 n의 경로를 찾으면 YES, 그렇지 않으면 NO를 되돌려 준다.

```
Search_Next (flag, n, i, j, k, cost)
/* search neighbor points from (i, j, k) for net n */
{
    if (flag == A) {
        if ( $A(i, j, k) \neq \infty$ ) {
            /* (i, j, k) has been searched at
            smaller cost */
            if (target of net n is found at (i, j, k))
                return (YES);
            else
                return (NO);
        }
        else if ( $B(i, j, k) \neq \infty$ ) {
            if (Selfloop is found)
                return (NO);
        }
         $A(i, j, k) = cost$ ;
        if ( $A(i \pm 1, j, k) == \infty$ )
            Push(A, n, i ± 1, j, k, cost + k. $\alpha$ );
        if ( $A(i, j \pm 1, k) == \infty$ )
            Push(A, n, i, j ± 1, k, cost + k. $\beta$ );
        if ( $B(i, j, k + 1) == \infty$ )
            Push(B, n, i, j, k + 1, cost + k. $\gamma$ );
        if ( $B(i, j, k - 1) == \infty$ )
            Push(B, n, i, j, k - 1, cost + (k-1). $\gamma$ );
    }
    else { /* flag == B */
        if ( $B(i, j, k) \neq \infty$ ) {
            /* (i, j, k) has been searched at
            smaller cost */
            if (target of net n is found at (i, j, k))
                return (YES);
            else
                return (NO);
        }
        else if ( $A(i, j, k) \neq \infty$ )
            return (NO);
         $B(i, j, k) = cost$ ;
        if ( $A(i \pm 1, j, k) == \infty$ )
            Push(A, n, i ± 1, j, k, cost + k. $\alpha$ );
        if ( $A(i, j \pm 1, k) == \infty$ )
            Push(A, n, i, j ± 1, k, cost + k. $\beta$ );
    }
}
return (NO);
```

그림 3. 탐색 알고리듬

Fig. 3. Searching algorithm.

초기애 각 그리드 점에서의 비용을 초기화 한다. 점 (i, j, k)를 n이외의 다른 네트 m이 점유하고 있으면 $A(i, j, k) = B(i, j, k) = -m$ 으로 초기화 하고, (i, j, k) 점이 비어 있어서 사용가능하면 $A(i, j, k) = B(i, j, k) = \infty$ 로 초기화 한다. 맨 처음의 탐색

은 네트 n의 단자의 위치 또는 이미 형성된 네트 n의 경로에서부터 시작된다. 탐색은 네트의 한 소자에서만 수행하며, 이미 형성된 경로에서 주변으로 탐색할 때, 또는 주변으로부터 형성된 경로로 탐색할 때는 비아 겹침을 허용하지 않았다. 즉, 이미 형성된 경로 상의 비아 위치 (i, j, k) 에서는 $(i, j, k \pm 1)$ 로 탐색을 못하도록 하고 $(i, j, k \pm 1)$ 위치에서도 경로 상의 비아 위치 (i, j, k) 로는 탐색을 못하도록 하였다. 이와 같은 방법으로 탐색할 경우, 형성된 경로 상의 비아 위치에서는 또 다른 비아가 만들어지지 않는다.

Search_Next에서는 점 (i, j, k) 에서 주변의 4 점, 즉 $(i \pm 1, j, k)$ 및 $(i, j \pm 1, k)$ 과 이웃하는 총의 2 점 $(i, j, k \pm 1)$ 을 탐색할 수 있다. 탐색된 점들은 비용이 적은 순으로 heap 또는 priority queue의 막디에 저장되며^[16], min-heap을 사용하여 최소 비용이 heap의 루트(root)에 저장되도록 하였다. 현재 점 (i, j, k) 에서 이웃하는 점 (i', j', k') 으로 탐색할 때, $A(i', j', k')$, $B(i', j', k')$ 값은 (i, j, k) 점에서의 값에 설계자가 지정한 단위 비용 $(k.\alpha, k.\beta, k.\gamma)$ 의 값은 표 2, 3 참조)을 더한 값을 사용한다.

탐색 과정에서 두개의 비용을 사용함으로써 발생할 수 있는 셀프 루프(self loop)를 설명하기로 한다. 셀프 루프란 점 (i, j, k) 에서 이웃하는 점 (i', j', k') 으로 탐색 범위를 확장하여 탐색을 계속하는 중에, 다시 점 (i', j', k') 에서 원래의 점 (i, j, k) 를 탐색하여 생기는 루프를 말한다.

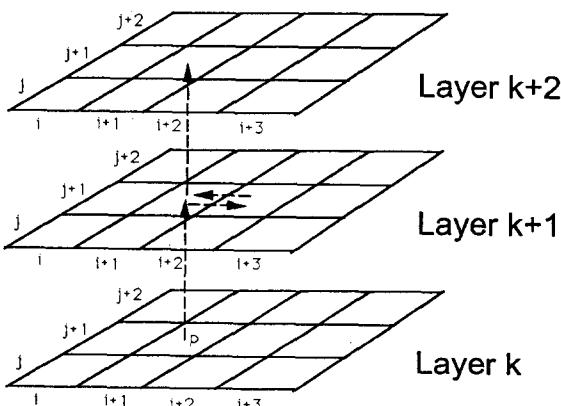


그림 4. 셀프 루프가 발생되는 예

Fig. 4. An example where a self loop is generated.

그림 4는 편 P의 위치 $(i + 1, j + 1, k)$ 에서 화살표 방향으로 탐색을 수행하는 경우, 아래와 같은 경

로로 루프가 발생하여 결과적으로 점 $(i + 1, j + 1, k + 1)$ 에서 비아 겹침이 발생하는 것을 보여준다.

점 $(i + 1, j + 1, k) \Rightarrow$ 점 $(i + 1, j + 1, k + 1) \Rightarrow$ 점 $(i + 2, j + 1, k + 1) \Rightarrow$ 점 $(i + 1, j + 1, k + 1) \Rightarrow$ 점 $(i + 1, j + 1, k + 2)$.

따라서 $B(i + 1, j + 1, k + 1) < \infty$ 일 때에는 셀프 루프가 없을 때에 한하여 인접층을 탐색해야 한다. Search_Next에서 flag가 A인가 B인가에 따라서 탐색 방법이 다르다. 점 (i, j, k) 에서 이웃하는 점을 탐색할 때 flag가 A일 경우에는 $(i \pm 1, j, k)$ 및 $(i, j \pm 1, k)$ 또는 $(i, j, k \pm 1)$ 로 탐색하고, 점 (i, j, k) 의 flag가 B일 경우에는 $(i \pm 1, j, k)$ 및 $(i, j \pm 1, k)$ 로만 탐색하여 비아의 겹침을 허용하지 않는다. 그럼 3의 알고리듬에서 $k.\alpha, k.\beta, k.\gamma, (k-1).\gamma$ 는 점 (i, j, k) 에서 점 $(i \pm 1, j, k)$ 및 $(i, j \pm 1, k)$ 또는 점 $(i, j, k \pm 1)$ 및 점 $(i, j, k - 1)$ 을 탐색하는 경우 표 2, 3에서 지정한 단위 비용을 나타내는 상수이다.

점 (i, j, k) 에서 이웃점을 탐색할 때 발생 가능한 모든 경우를 나열하면 다음과 같다. 표 1은 이를 정리한 것이다.

< flag 가 A일 경우 >

탐색 경우 1 : $A(i, j, k) = \infty$ 이고 $B(i, j, k) = \infty$ 이면, Search_Next에서 인자로 받아들인 비용(cost)을 $A(i, j, k)$ 에 저장하고 탐색 가능한 점 ($A(i \pm 1, j, k) = \infty$ 일 경우 점 $(i \pm 1, j, k)$, $A(i, j \pm 1, k) = \infty$ 일 경우 점 $(i, j \pm 1, k)$, $B(i, j, k \pm 1) = \infty$ 일 경우 점 $(i, j, k \pm 1)$)을 탐색하여 heap에 저장한다. 탐색을 수행한 후 NO를 리턴 한다.

탐색 경우 2 : $A(i, j, k) = \infty$ 이고 $B(i, j, k) \neq \infty$ 이면, 점 (i, j, k) 는 $(i, j, k \pm 1)$ 에서 이미 탐색된 점이므로 이 점을 포함하는 루프가 존재하는지를 검사한다. 셀프 루프가 존재하면 NO를 리턴 한다. 셀프 루프가 존재하지 않는 경우는 인자로 받아들인 비용(cost)을 $A(i, j, k)$ 에 저장하고 탐색 가능한 점 ($A(i \pm 1, j, k) = \infty$ 일 경우 점 $(i \pm 1, j, k)$, $A(i, j \pm 1, k) = \infty$ 일 경우 점 $(i, j \pm 1, k)$, $B(i, j, k \pm 1) = \infty$ 일 경우 점 $(i, j, k \pm 1)$)을 탐색하여 heap에 저장한다. 탐색을 수행한 후 NO를 리턴 한다.

탐색 경우 3, 탐색 경우 4 : $A(i, j, k) \neq \infty$ 이면, 경로를 발견하였는지를 검사한다. 경로를 찾았다면 YES를 리턴하고 아니면 탐색 과정중 지나왔던 경로를 의미하므로 NO를 리턴 한다.

< flag 가 B일 경우 >

탐색 경우 5 : $A(i, j, k) = \infty$ 이고 $B(i, j, k) = \infty$ 이면, 인자로 받아들인 비용 (cost)을 $B(i, j, k)$ 에 저장하고 탐색 가능한 점 ($A(i \pm 1, j, k) = \infty$ 일 경우 점 $(i \pm 1, j, k)$, $A(i, j \pm 1, k) = \infty$ 일 경우 점 $(i, j \pm 1, k)$)을 탐색한 후 heap에 저장한다. 탐색을 수행한 후 NO를 리턴 한다.

탐색 경우 6, 탐색 경우 8 : $B(i, j, k) \neq \infty$ 이면, 경로를 발견하였는지를 검사한다. 경로를 찾았다면 YES를 리턴하고 아니면 탐색 과정 중 지나왔던 경로를 의미하므로 NO를 리턴 한다.

탐색 경우 7 : $A(i, j, k) \neq \infty$ 이고 $B(i, j, k) = \infty$ 이면, 더 이상의 탐색을 하지 않고 NO를 리턴 한다.

표 1. 발생 가능한 탐색의 모든 경우

Table 1. All the cases of possible searching
(T : True, F : False)

조건 발생 하는 경우	비용이나 네트의 존재		탐색 방법
	flag	A(i,j,k) B(i,j,k)	
탐색 경우 1 A	F	F	$(i \pm 1, j, k)$, $(i, j \pm 1, k)$, $(i, j, k \pm 1)$
탐색 경우 2 A	F	T	센프 루프가 없는 경우, $(i \pm 1, j, k)$, $(i, j \pm 1, k)$, $(i, j, k \pm 1)$
탐색 경우 3	A	T	F
탐색 경우 4	A	T	T
탐색 경우 5	B	F	F
탐색 경우 6	B	F	T
탐색 경우 7	B	T	F
탐색 경우 8	B	T	T

Find_Path에서 초기의 탐색 과정이 끝나면 while 루프를 수행한다. 이 과정에서는 저장된 탐색 정보를 heap의 루트에서부터 하나씩 거내어 경로를 찾을 때까지 반복적으로 탐색 범위를 확장한다. 탐색 범위는 제한을 두어 탐색해야 할 점의 비용이 주어진 비용 제한 값 (Limit_Cost) 보다 작은 범위 내에서만 경로를 찾도록 하였다.

이는 지나치게 우회하여 네트의 경로를 찾지 않도록 하기 위해서이며, 비용이 제한 값보다 크게 되면 네트의 탐색을 중단하고 NO 값을 리턴 한다.

네트 n의 경로를 발견하였으면 그 경로를 발견한 점을 시작점으로 경로를 역추적하여 임시 경로를 만든다. 네트 n의 경로를 발견한 점 (i, j, k) 에서 그 경로의 시작점으로 역추적 하는 Back_Trace 과정은 탐색 과정을 역으로 수행하는 것과 같다. 현재점 (i, j, k) 에서 이웃하는 점 (i', j', k') 으로 역추적할 때, $A(i', j', k')$, $B(i', j', k')$ 값은 (i, j, k) 점에서의 비용 값에 설계자가 지정한 단위 비용 (표 2, 3 참조)을 감산한 값을 사용한다. 현재점 (i, j, k) 는 $(i \pm 1, j, k)$, $(i, j \pm 1, k)$ 또는 $(i, j, k \pm 1)$ 의 어느 한 위치에서 탐색되었던 점이므로 역으로 추적하면 경로를 찾아낼 수 있다.

일반적으로 미로 배선에서 경로를 역으로 추적하는 경우 실현 가능한 경로는 여러개 존재할 수 있으며, 이들 중에서 비아의 겹침이 없는 경로를 선택해야 하는 문제가 있다. 세안한 미로 배선 기법에서 비아의 겹침이 없는 경로를 선택하기 위한 방법을 설명하기 위하여 그림 5를 예로 든다.

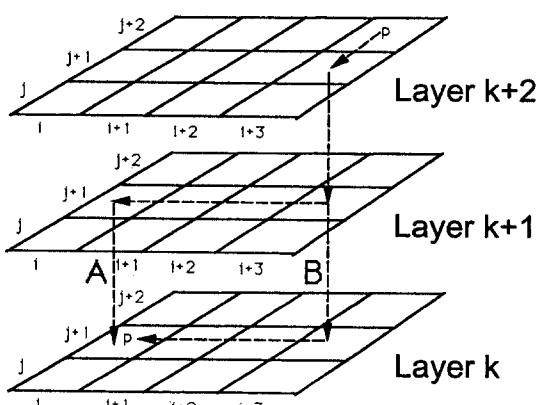


그림 5. 여러개의 역추적 경로

Fig. 5. Several backtracking paths.

핀 P의 위치 $(i + 3, j + 2, k + 2)$ 에서 핀 $(i, j + 1, k)$ 까지 역으로 경로를 추적할 때 여러개의 가능한 경로가 존재할 수 있다. 그림 5에서는 두 개의 가능한 역추적 경로 A 와 B를 보였다. 그러므로, 여러 개의 가능한 경로 중에서 비아의 겹침이 없는 경로를 선택해야 한다. 이러한 문제를 쉽게 해결할 수 있는 방법의 하나는 존재하는 모든 경로를 찾은 후, 비아 겹침이 없는 경로를 선택하면 된다. 그러나 본 논문에서는 복잡도를 줄이기 위하여 다음과 같은 간단한 역추적 방법을 사용하였다.

점 (i, j, k) 에서 점 $(i \pm 1, j, k)$, $(i, j \pm 1, k)$

또는 $(i, j, k \pm 1)$ 로 네트의 경로를 역으로 추적할 때, 점 (i, j, k) 가 점 $(i \pm 1, j, k)$, $(i, j \pm 1, k)$ 에서 역추적된 점이면 $A(i \pm 1, j, k)$ 또는 $A(i, j \pm 1, k)$ 값을 이용하여 역추적 한다. 그러나, 점 (i, j, k) 가 $(i, j, k \pm 1)$ 에서 역추적된 점이면 $B(i, j, k \pm 1)$ 의 값을 이용하여 역추적 한다. A 비용과 B 비용 모두를 이용하여 역추적이 가능한 경우에는 우선적으로 A 비용을 이용하여 역추적하도록 하였다. 이같은 방법으로 역추적 하는 경우, 그림 5에서는 A 경로를 선택하게 되는데,

비아를 형성하며 탐색하였던 (i, j, k) 위치에서는 $(i \pm 1, j, k)$, $(i, j \pm 1, k)$ 으로 역추적 되므로 (i, j, k) 의 위치에서 비아 겹침이 발생할 수 없다. 또한 이 때의 비용은 가설 1에서 증명한 바와 같이 최소 비용이다. ■

IV. 실험

1. 간단한 회로의 배선 예

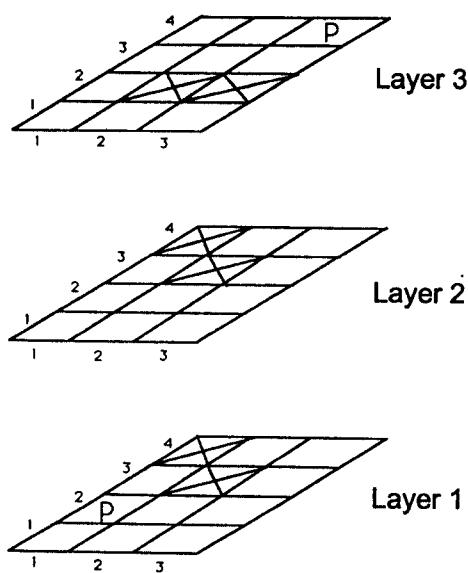


그림 6. 배선 입력 데이터

Fig. 6. Routing input data.

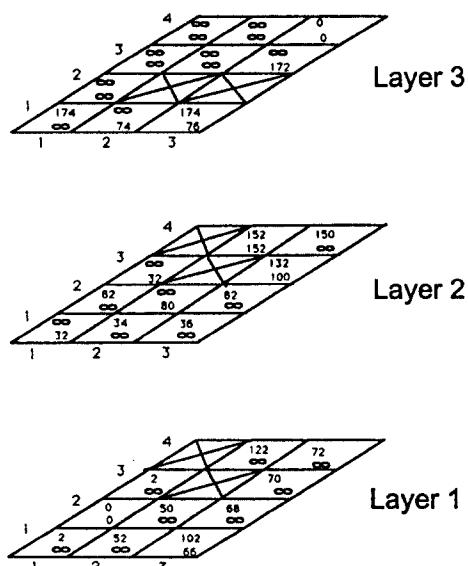


그림 7. 배선 결과 데이터

Fig. 7. Routing result data.

현재 위치에서 이웃하는 위치로 탐색할 때, 표 2에 지정되어 있는 배선 비용을 사용한다. 핀 (1, 2, 1) 위치에서 수직 방향의 이웃하는 점인 (1, 1, 1) 또는

(1, 3, 1)들을 탐색할 때의 비용은 2이며, 수평 방향의 이웃하는 점 (2, 2, 1)을 탐색할 때의 비용은 50이다. 그럼 그림 7은 이와 같은 방법으로 탐색하여 경로를 발견할 때까지의 각 층에 표시된 배선 비용을 보여주고 있다. 셀프 루프는 (1, 1, 2) 위치에서 찾아 볼 수 있다. 실제 탐색 과정에서는 (2, 1, 2) 위치에서 (1, 1, 2) 위치로 탐색을 하여 비용 36으로 A(1, 1, 2)를 heap에 저장해 놓았다. 탐색된 점 (1, 1, 2)를 heap에서 꺼내어 이 위치에서 다시 탐색하고자 할 때 (1, 1, 2), (2, 1, 2), (1, 1, 3)의 경로로 루프가 존재하므로 점 (1, 1, 2)에서는 (1, 1, 3)으로 탐색을 하지 않는다. 계속적으로 탐색을 해 나가는 중에, (3, 3, 3) 위치에서 (3, 4, 3)을 탐색하면 배선되어야 할 다른 핀이 발견된다.

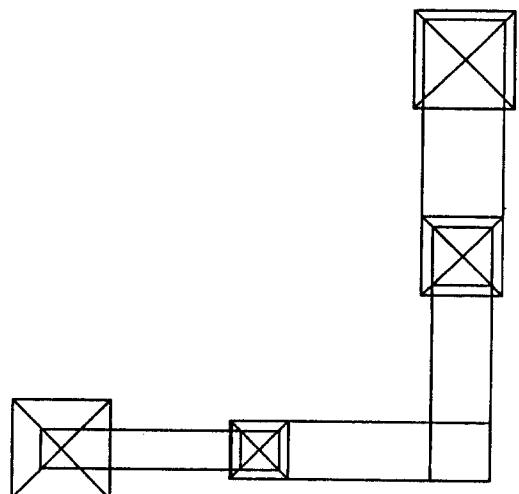


그림 8. 비아 겹침이 발생하지 않은 예

Fig. 8. An example without stacked vias.

이 핀을 찾을 때의 비용은 $172 + 4$ 가 되며, 더 이상의 탐색은 하지 않는다. 다음은 비용 176을 가진 핀으로부터 역으로 경로를 추적하기 시작한다. B(3, 3, 3)의 비용 172는 같은 층의 주변의 점에서 탐색된 비용이 아니므로, (3, 3, 2) 위치로 비아를 형성하여 역으로 추적된다. 점 (3, 3, 2)는 같은 층의 주변의 (3, 2, 2) 위치에서 탐색된 비용이므로 이 위치로 역추적 된다. 점 (3, 2, 2)에서는 점 (2, 2, 2)로 역추적되며, 다시 1번 층의 (2, 2, 1) 위치로 비아를 통하여 연결된다. 점 (2, 2, 1)은 점 (1, 2, 1)로 역추적되어 네트의 경로를 찾게 된다.

그림 8은 본 논문에서 제안한 알고리듬을 이용하여 배선이 완료된 상태를 보여 주며, 비아 겹침이 발생하

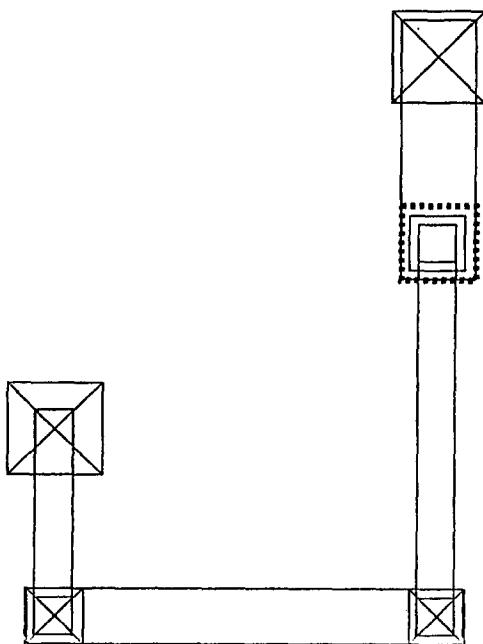


그림 9. 비아 겹침이 발생한 예

Fig. 9. An example with stacked vias.

지 않았음을 보여준다. 그림 9는 같은 배선 조건으로 비아의 겹침을 고려하지 않았을 때의 미로 배선 결과를

보여주며, 실제로 비아 겹침이 발생하였음을 알 수 있다. 그럼 9의 배선 경로가 그림 8보다 우회한 이유는, 이 경로가 현재의 지정된 단위 길이당의 배선 비용과 비아의 배선 비용을 사용하여 배선 할 경우 최소 비용에 해당하기 때문이다.

2. 미로 배선의 실험 결과

본 논문에서 제안한 알고리듬으로 여러 개의 예제 회로를 배선하여 실험하였다.

표 4. 예제 회로

Table 4. An example circuit.

회로 이름	트랜지스터 수	네트 수	열 수
dff	40	26	2
alu	302	164	4

표 4는 잘 알려진 두 예제 회로의 초기 배치 상태 및 회로 특성을 나타내었다. dff 회로는 2개의 열상에 배치하고, alu는 4개의 열상에 배치한 후 배선을 수행하였다. 그림 10은 비아의 겹침을 허용하지 않고 dff 회로를 배선한 결과이며, 그림 11은 비아 겹침을 허용하여 dff 회로를 배선한 결과이다. 그림 12는 배선이 완료된 alu 회로를 출력한 결과이다.

표 5는 SUN SPARC2 workstation에서 비아의 겹침을 허용할 때와 허용하지 않을 때의 수행시간을 비교하여 나타낸 것이다. 비아의 겹침이 없는 배선 기법

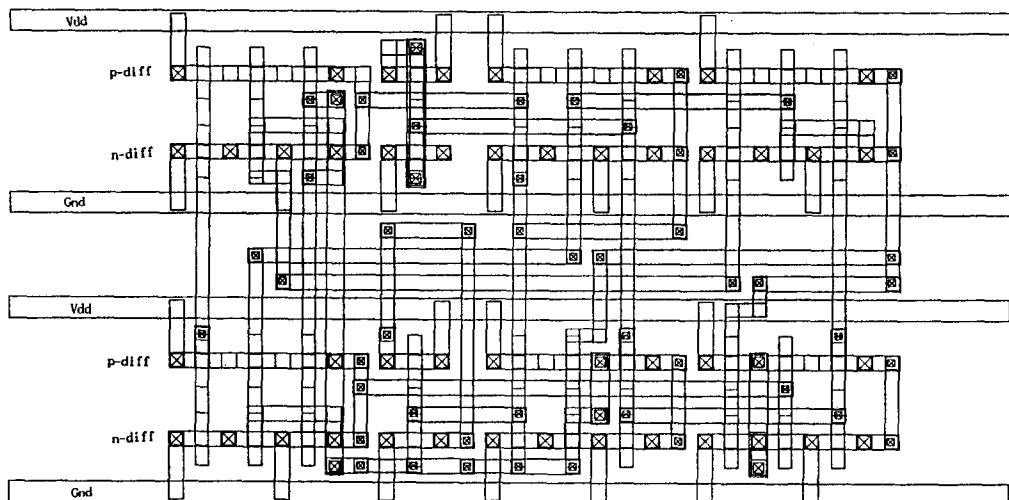


그림 10. dff 회로: 비아 겹침을 허용하지 않은 배선 결과

Fig. 10. dff circuit: A routing result when stacked vias are not allowed.

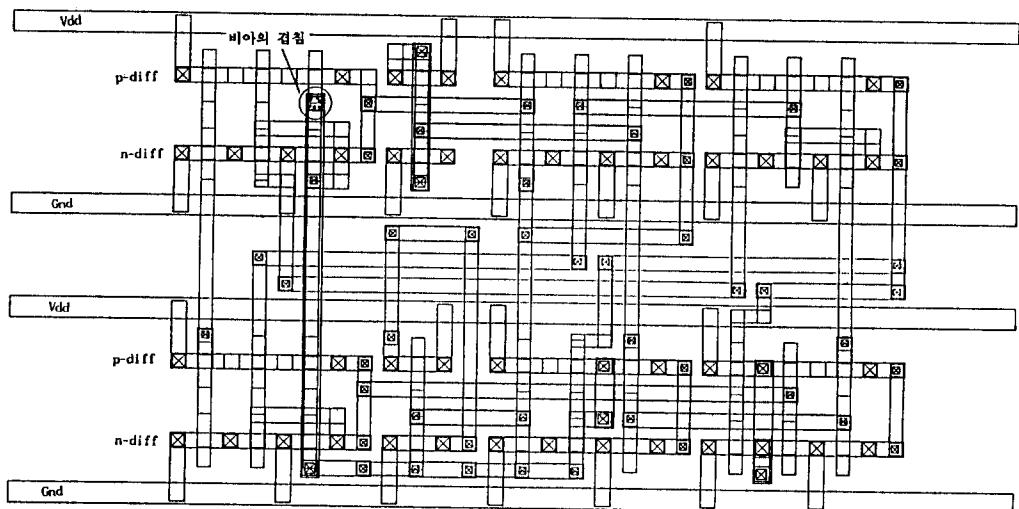


그림 11. DFF 회로: 비아 겹침을 허용한 배선 결과

Fig. 11. DFF circuit: A routing result when stacked vias are allowed

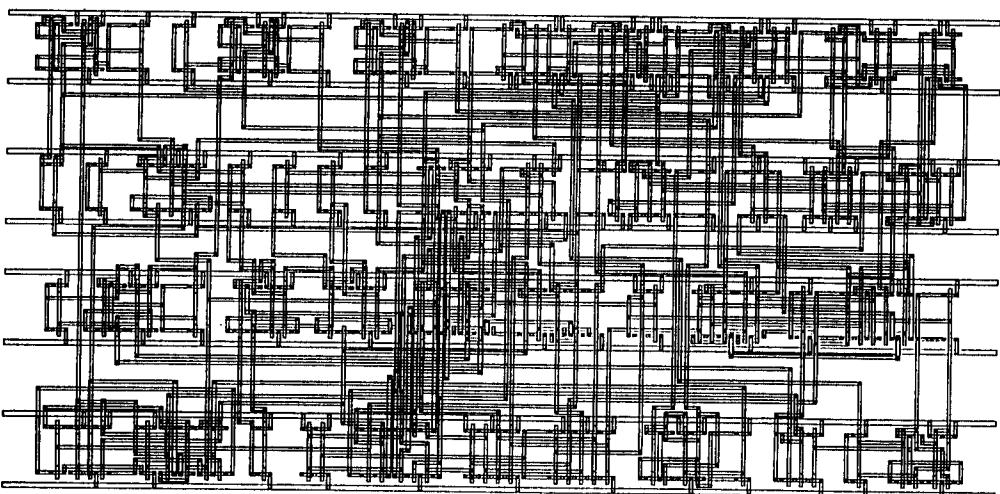


그림 12. ALU 회로: 비아 겹침을 허용하지 않은 배선 결과

Fig. 12. ALU circuit: A routing result when stacked vias are not allowed.

에서 수행시간이 늘어난 주된 이유는, 경로를 찾는 과정에서 점 (i, j, k) 에서 $A(i, j, k)$, $B(i, j, k)$ 두 개의 비용을 동시에 고려하므로, 하나의 비용으로 탐색할 때 보다 많은 탐색을 수행하기 때문이다. 표 5의 비아 겹침이 없는 미로 배선에서 팔호 안의 수행 시간은 표 1의 탐색 경우 중에서 “탐색 경우 2”의 탐색을

생략했을 때의 수행 시간이다. “탐색 경우 2”를 생략하면 셀포 루프를 검사할 필요가 없어서 수행 시간이 감소한다. 물론 이때에 최소 비용의 경로를 찾지 못할 수도 있다. 그러나, 비아 겹침이 없는 경로를 찾는 것은 가능하다.

표 5. 수행 시간 (초)
Table 5. Run time (sec.).

회로 이름	기준의 미로 배선	비아 겹침없는 미로 배선
dff	4	4 (4)
alu	107	760 (590)

V. 결 론

본 논문에서는 모듈 합성에서의 세부 배선 단계에서 비아 겹침을 허용하지 않는 새로운 미로 배선 기법을 기술하였다. 이 배선 기법에서는 비아 겹침을 허용하지 않는 제약하에서 최소 비용의 경로를 찾기 위하여, 각 층의 각 그리드 점에 A, B 두 개의 비용 값을 이용하였다. 연결되어야 할 네트의 경로를 찾는 탐색 과정에서는 비아가 발생하지 않도록 탐색을 수행하였다. 또한 경로를 찾은 후에 역추적하는 과정에서도 두 개의 비용을 조사하여 비아의 겹침이 발생하지 않는 경로를 선택하였다. 본 논문에서 제안한 비아 겹침을 허용하지 않는 미로 배선 알고리듬을 이용하여 몇 개의 예제 회로에 대한 배선을 실현한 결과, 배선 시간은 더 길었지만 같은 수의 배선 영역 그리드 내에서 비아의 겹침이 없는 배선을 완료할 수 있었다. 일반적으로 비아의 겹침이 없는 배선은 비아의 겹침을 허용한 배선에 비하여 어려운 문제이므로, 배선에 필요한 면적이 증가할 수도 있다. 본 연구의 의미는 비아의 겹침이 없는 최소 비용의 미로 배선 방법을 개발하였는데 있다.

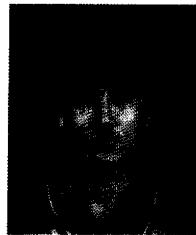
참 고 문 헌

- [1] Uehara, T., W.M. van Cleemput, "Optimal Layout of CMOS Functional Array," IEEE Trans. on Computers, pp. 305-312, May, 1981.
- [2] J.A. Feldman, I.A. Wagner, and S. Wimer, "An Efficient Algorithm for some Multirow Layout Problems," IEEE Trans. on CAD of IC&S, Vol. 12, No.8, pp. 1178-1185, Aug. 1993.
- [3] Lee, C.Y., "An Algorithm for Path Connections and Its Applications," IRE Trans. on Electronic Computers, EC-10, pp. 346-365, Sep. 1961.
- [4] H. Shin, A. Sangiovanni-Vincentelli, "A detailed router based on incremental routing modifications : Mighty," IEEE Trans. on CAD, vol. CAD-6, pp. 942-955, Nov. 1987.
- [5] D. Braun, J. Burns, S. Devadas, H.K. Ma, K. Mayaram, F. Romeo, and A. Sangiovanni-Vincentelli, "Chameleon: A New MultiLayer Channel Router," DAC, pp. 495-502, Jun. 1986.
- [6] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, "Introduction to Algorithms," The MIT Press, pp. 140-152, 1992.

저자소개

**權成勳(準會員)**

1968年 1月 17日生. 1993年 2月 한양대학교 전자공학과 석사 과정 졸업. 1993年 2月 ~ 현재 한양대학교 대학원 전자공학 박사과정 재학중. 주관심분야는 VLSI CAD, 디지털 신호 처리 등임.

**吳命燮(準會員)**

1969年 4月 4日生. 1993年 2月 한양대학교 전자공학과 졸업. 1993年 2月 ~ 현재 한양대학교 대학원 전자공학 석사과정 재학중. 주관심분야는 VLSI CAD, 영상 처리 등임.

**申鉉哲(正會員)**

1955年 9月 12日生. 1978年 2月 서울대학교 전자공학과 졸업. 1980年 2月 한국과학기술원 전기 및 전자공학과 공학석사. 1987年 미국 U.C. Berkeley 전기 및 컴퓨터공학과 공학박사. 1980年 ~ 1983年 금오공과대학 전자공학과 전임강사 조교수. 1987年 ~ 1989年 미국 AT&T Bell Laboratories, Murray Hill 연구원. 1989年 9月 ~ 현재 한양대학교 전자공학과 부교수. 주관심분야는 집적회로 설계자동화, 디지털 신호처리 시스템 설계 등임.