Journal of The Institute of Electronics and Information Engineers Vol.53, NO.4, April 2016 http://dx.doi.org/10.5573/ieie.2016.53.4.037
ISSN 2287-5026(Print) / ISSN 2288-159X(Online)

논문 2016-53-4-5

멀티플렉서 트리 합성이 통합된 FPGA 매핑

(FPGA Mapping Incorporated with Multiplexer Tree Synthesis)

김 교 선*

(Kyosun Kim©)

요 약

광폭입력함수 전용 멀티플렉서가 슬라이스 구조에 포함되는 상용 FPGA의 현실적 제약 조건을 학계의 대표적 논리 표현 방식인 AIG (And-Inverter Graph)를 근간으로 개발된 FPGA 매핑 알고리즘에 적용하였다. AIG를 LUT (Look-Up Table)으로 매핑할 때 중간 구조로서 컷을 열거하는 데 이들 중에서 멀티플렉서를 인식해 낸 후 이들이 매핑될 때 지연 시간 및 면적을 복잡도 증가 없이 계산하도록 하였다. 이 때 트리 형성 전제 조건인 대칭성과 단수 제약 요건도 검사하도록 하였다. 또한, 멀티플렉서 트리의 루트 위치를 RTL 코드에서 찾아내고 이를 보조 출력 형태로 AIG에 추가하도록 하였다. 이 위치에서 섀넌 확장을 통해 멀티플렉서 트리 구조를 의도적으로 합성한 후 최적 AIG에 겹치도록 하는 접근 방법을 최초로 제안하였다. 이 때 무손실 합성을 가능하게 하는 FRAIG 방식이 응용되었다. 두 가지 프로세서에 대해 제안된 접근 방법과 기법들을 적용하여 약 13~30%의 면적 감소 및 최대 32%까지의 지연 시간 단축을 달성하였다. AIG 트리에 특정 구조를 의도적으로 주입시키는 접근 방법은 향후 캐리 체인 등에 확장 적용하는 연구가 진행될 것이다.

Abstract

The practical constraints on the commercial FPGAs which contain dedicated wide function multiplexers in their slice structure are incorporated with one of the most advanced FPGA mapping algorithms based on the AIG (And-Inverter Graph), one of the best logic representations in academia. As the first step of the mapping process, cuts are enumerated as intermediate structures. And then, the cuts which can be mapped to the multiplexers are recognized. Without any increased complexity, the delay and area of multiplexers as well as LUTs are calculated after checking the requirements for the tree construction such as symmetry and depth limit against dynamically changing mapping of neighboring nodes. Besides, the root positions of multiplexer trees are identified from the RTL code, and annotated to the AIG as AOs (Auxiliary Outputs). A new AIG embedding the multiplexer tree structures which are intentionally synthesized by Shannon expansion at the AOs, is overlapped with the optimized AIG. The lossless synthesis technique which employs FRAIG (Functionally Reduced AIG) is applied to this approach. The proposed approach and techniques are validated by implementing and applying them to two RISC processor examples, which yielded 13~30% area reduction, and up to 32% delay reduction. The research will be extended to take into account the constraints on the dedicated hardware for carry chains.

Keywords : Multiplexor tree synthesis, Field programmable gate array, Mapping, Functionally reduced and-inverter graph

Ⅰ. 서 론

FPGA (Field Programmable Gate Array)에 대한 연구는 디지털시스템을 프로토타이핑 할 때 FPGA 칩을 응용한 사례가 대부분이며 학계의 CAD 툴[1~3] 개발은 단순한 FPGA 가상 구조를 가정한 실험실용 수준에서 벗어나지 못했다. 상용 FPGA 칩 구조의 데이터베이스는 개발사만의 전유물이었고 학계에서 CAD 툴 개발에 필요한 정보 제공이 충분하지 않아 상용 FPGA에 직접 적용할 수 있는 실용적 CAD툴의 개발이 부진해 왔던 것이 사실이다. 특히 단순 FPGA 구조에서 상용 CAD 툴을 능가하는 결과와 성능을 보였던 학계의 AIG (And-Inverter-Graph)[1] 및 MIG (Majority-Inverter Graph)[3] 등의 데이터 구조와 알고리즘들이 실용화되는 데 장애가 되고 있다. 얼마 전 USC 그룹의 Torc[4] 및

* 정회원, 인천대학교 전자공학과
(Department of Electronic Engineering, Incheon National University)
© Corresponding Author(E-mail: kkim@inu.ac.kr)
※ 본 논문은 인천대학교 2014년도 자체연구비 지원에 의하여 연구된 것임.
Received ; February 25, 2016 Revised ; April 1, 2016
Accepted ; April 4, 2016

BYU 그룹의 RapidSmith[5] 툴킷이 발표되었다. 두 툴킷 모두 Xilinx 사의 XDL 파일 형식[6]을 근간으로 데이터베이스를 구축하고 C++/Java 형태의 API (Application Programmable Interface)를 제공함으로써 설계 입력, 논리 합성, 패킹, 배치, 배선, 아키텍처 뷰어 등을 플러그-인 할 수 있도록 하고 있다. 한편, 기본 특허의 만료 시기가 다가오면서 Xilinx 및 Altera의 아성이었던 FPGA 산업에 세계적으로 많은 벤처 기업들이 관심을 보이고 있다. 국내에서도 “Configurable 디바이스 및 회로구현 기술”이라는 지식경제부 기술혁신사업이 2009년부터 5년간 진행되었으며 신규 개발된 K-FPGA 아키텍처 및 툴킷이 소개되었다[7]. 또한, K-FPGA 툴킷을 사용한 FPGA 배치 툴[8]도 발표되었는데 이는 상용 FPGA 구조에서 등장한 현실적 제약조건이 사전 패킹 및 다층 밀도 분석 등으로 해결된 사례라 할 수 있다.

본 논문 역시 상용 FPGA 구조를 정확히 모델링할 수 있는 K-FPGA 툴킷[7] 상에서 구현된 논리 합성 및 FPGA 매핑 최적화를 기술한다. 특히, 기존 학계 연구와는 달리 LUT (Look-Up-Table) 및 플립플롭만 존재하는 단순한 아키텍처에서 벗어나 광폭 입력 멀티플렉서 및 지역 배선을 포함하는 상용 FPGA 아키텍처를 허용하는 실용적 매핑 기술을 제안한다.

본 논문은 먼저 Xilinx Spartan 구조 및 기존 FPGA 매핑 기술을 Ⅱ장 본론 1절에서 소개하고 2절 및 3절에서 멀티플렉서 트리 매핑과 멀티플렉서 트리 조성 방법을 제안하다. Ⅲ장 실험에서 2개의 실용 예제에 대해 적용한 결과를 제시하고 고찰한 후 마지막으로 결론을 맺는다.

Ⅱ. 본 론

1. 배경 이론

가. 상용 FPGA 구조

그림 1에 Xilinx Spartan III[9] 슬라이스 구조를 나타내었다. 먼저 F와 G로 표시된 LUT (Look-Up-Table) 2개, FFX와 FFY로 표시된 플립플롭 2개가 포함되어 있다. 또한, 입력 수가 많은 함수 구현을 위한 광폭 입력 멀티플렉서 (F5MUX 및 F6MUX), 그리고 산술연산 구현을 위한 캐리 멀티플렉서 (CYMUXF, CYMUXG) 및 전용 게이트들 (XORF, XORG, FAND, GAND)이 있다. 이 밖에 연결 선택을 프로그램 할 수 있는 멀티플렉서, 인버터(INV), 스위치 (USED)들이 연결 경로를 재구성하여 구성 요소들을 선택적으로 사용할 수 있도

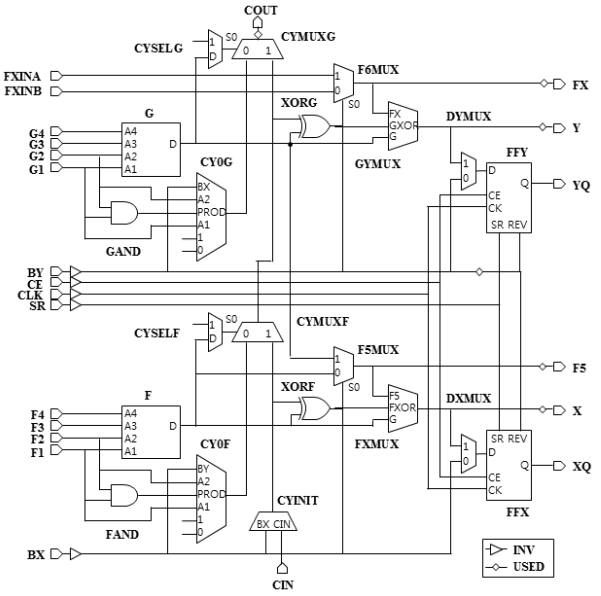


그림 1. Xilinx Spartan III 슬라이스 (SLICEL) 구조
Fig. 1. Slice structure (SLICEL) of the Xilinx Spartan III.

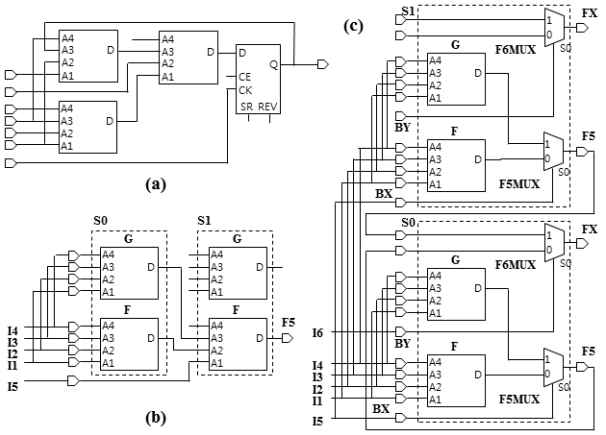


그림 2. 광폭 입력 함수의 FPGA 구현
Fig. 2. FPGA Implementation of a wide fanin function.

록 하고 있다. 이러한 슬라이스들이 2차원 배열 구조 형태로 FPGA 칩 상에 수천~수백만 개 분포되며 그것들 사이사이에 배치된 스위치 박스들이 슬라이스들 간의 연결을 결정한다.

학계의 대표적인 논리합성 툴은 UC Berkeley의 ABC[1]이다. 최근까지 학계에서는 LUT와 플립플롭만 존재하는 매우 단순한 슬라이스 구조를 가정하여 자동화 툴이 연구되어 왔으며 ABC에서 합성되는 회로도 그림 2(a)와 같이 LUT와 플립플롭들만 연결된 형태이다. 사실 4-입력 LUT로 입력이 4개인 모든 조합 논리의 구현이 가능하며 입력 개수가 늘어나도 LUT들을 트리 형태로 연결하면 되므로 어떤 디지털 회로도 구현할 수 있다. 그림 2(b)에 LUT를 트리 형태로 연결하여

입력이 5개인 논리를 구현한 회로를 보이고 있다. 그림 1의 슬라이스 구조를 사용한다면 2개의 슬라이스 (S0와 S1)가 필요함을 알 수 있다. 만약, 5-입력 논리를 변수 하나에 대해 Shannon (섀넌) 확장한 후 F5MUX를 사용한다면 4-입력 LUT 2개가 들어 있는 그림 2(c)의 S0 슬라이스 하나만 가지고도 저렴하게 5-입력 논리를 구현할 수 있다. 더 나아가, S1 슬라이스로 또 하나의 5-입력 논리를 구현하고 이 둘을 S0 슬라이스의 F6MUX로 연결하면 6-입력 논리를 구현할 수 있다.

사실, 광폭 입력 함수가 많지 않은 설계의 경우 이 구조에 의한 슬라이스 사용량 감축은 크지 않을 수 있다. 그러나 신호가 하나의 슬라이스에서 다른 슬라이스로 전달될 경우 긴 연결선과 스위치 박스를 하나 이상 거치게 되는데 이 배선 지연은 LUT의 게이트 지연보다도 더 커질 수 있기 때문에 슬라이스의 단수가 감소된다는 점이 더 중요하다. 그림 2(c)에서 LUT 출력들은 스위치 박스를 거치지 않고 전용 내부 배선 및 지역 배선을 통해 F5MUX 및 F6MUX와 단거리로 연결될 수 있다. 그림 2(b)의 5-입력 회로는 2단 연결이 필요하지면 그림 2(c)와 같이 광폭 입력 멀티플렉서와 전용 배선을 이용하면 6-입력이나 그 이상의 입력에도 슬라이스 단수가 늘어나지 않는다.

나. Functionally Reduced AIG (FRAIG)[10] 구축

AIG는 각 노드가 2-입력 AND 게이트로 구성되며 인버터들이 입출력 에지에 선택적으로 표시되는 네트워크이다. AIG는 정규형이 정의되지 않아 서로 다른 구조를 가진 두 AIG가 등가일 수 있다. 그러나 AIG에 구조적 해시 (STRASH: Structural Hash)를 적용하면 일정한 논리 단수 내에서 정규형이 정의될 있다. 추가적으로 등가 노드들을 합병하면 함수적으로 축약된 AIG (FRAIG: Functionally Reduced AIG)가 된다.

그림 3에 1단 STRASH 후에 함수적 축약을 하는 FRAIG 구축 함수의 의사 코드를 보이고 있다. 처음에 두 입력 노드가 동일한지, 같은 노드가 반전된 것인지 혹은 상수인지 확인한 후 정렬시켜 입력을 맞바꿔도 같은 노드로 인식될 수 있도록 한다. 다음, 두 입력과 이를 받는 AND 게이트를 쌍으로 매핑하는 해시 테이블을 검색하여 1단 STRASH를 수행한다. 기존에 등록된 노드가 있으면 그것을 리턴하고 그렇지 않으면 새로운 노드를 생성하여 해시 테이블에 등록한다. 이어 함수 등가를 확인하기 위해 CheckFunctionEquivalence()를 호출하는데 이는 SAT 풀이 함수로 시간이 가장 많이

```
Node *AND(Manager *p, Node *n1, Node *n2) {
    Node      *r, *n, *t;
    NodeArray *class;
    // trivial cases
    if(n1 == n2) return n1;
    if(n1 == NOT(n2)) return 0;
    if(n1 == const) return 0 or n2;
    if(n2 == const) return 0 or n1;
    if(n1 < n2) { // swap the arguments
        t = n1; n1 = n2; n2 = t;
    }
    // one level structural hashing
    r = HashTableLookUp(
            p->pTableStructure, n1, n2);
    if(r) return r;
    r = CreateNode(p, n1, n2);
    // functional reduction
    class = HashTableLookUp(
                p->pTableSimulation, n1, n2);
    if(class == NULL) {
        class = CreateNewSimulationClass(r);
        HashTableAdd(p->pTableSimulation, class);
        return r;
    }
    for each node n in class
        if(CheckFunctionEquivalence(n, r)) {
            AddNodeToEquivalenceClass(class, r);
            return n;
        }
    AddNodeToSimulationClass(class, r); return r;
}
```

그림 3. FRAIG 구축
Fig. 3. FRAIG construction.

소요된다. 이 함수 호출 횟수를 줄이기 위해 랜덤 시뮬레이션이 먼저 시도된다. 시뮬레이션 벡터를 AIG의 외부 입력 (PI: Primary Input)에 인가하여 비트-병렬 시뮬레이션한 후 각 노드의 벡터를 유도한다. 신규 노드의 시뮬레이션 벡터는 그 입력 노드들의 벡터를 비트별 AND 연산을 하면 계산된다. 물론 에지 상의 인버터 유무에 따라 입력이 반전되기도 한다. 먼저 계산된 시뮬레이션 벡터를 그와 같은 벡터를 가진 AIG 노드들의 클래스에 매핑하는 해시 테이블을 추가로 사용한다.

만약 신규 노드의 벡터와 매핑된 클래스가 비어 있다면 그 노드 함수의 고유성이 시뮬레이션에 의해 이미 확인된 것이므로 그 클래스에 등록한 후 리턴한다. 만약에 비어있지 않다면 그 클래스 내 각 노드 n에 대해 신규 노드와 SAT 방식의 등가성 검사가 실시된다. 만약 검사 결과가 참이라면 신규 노드는 n과 등가이기 때문에 클래스에 등록한 후 기존 노드 n을 리턴한다. 어떤 멤버와도 등가가 아니라면 신규 노드를 클래스에 추가한 후 리턴한다. 만약 새로운 노드가 기존 노드와 등가이면 즉시 제거할 수도 있지만 등가 클래스에 남겨두면 다음번에 이 노드와 같은 형태의 노드가 또 생성될 때 클래스 내에서 단순 비교로 확인할 수 있기 때문

에 등가 검사를 다시 하지 않고 즉시 대표 노드를 리턴할 수 있게 된다. 더욱 흥미로운 점은, 같은 함수의 다른 구조들이 하나의 그래프에 겹쳐져서 기록되는 효과도 있다는 것이다. 이것이 무손실 합성 (Lossless Synthesis)[10~11]의 착안점이다. 합성에서는 네트워크가 최적화의 대상이다. 기존에는 각 스텝이 끝날 때마다 새로운 네트워크로 바뀌게 되며 이전 네트워크는 유실된다. 때로는 중간에 나타났던 네트워크의 일부 또는 전체가 최종 형태보다 더 나은 경우도 있다. 무손실 합성은 중간에 나타났던 모든 논리 표현들을 누적시켜 놓았다가 선택은 나중에 최종적으로 한다는 것이다.

FRAIG는 부울 함수의 구조적 구현의 여러 대안을 쉽게 검출하고 누적시키는 효율적인 방법이라 할 수 있다. 일련의 최적화 명령들을 실행시키면서 유도된 여러 개의 네트워크 버전들을 FRAIG 시키면 함수적으로 등가인 노드들이 같은 클래스에 통합되어 구조적 대안들이 하나의 AIG 내부에 기록된다. 구조적 차이가 어디서 발생했는지 상관없이 FRAIG는 그 차이를 검출하고 저장한다. 차이가 누적된 이 그래프에 테크놀로지 매핑을 적용하면 여러 가지 대안들 중에서 가장 좋은 매핑을 선택할 수 있게 된다.

다. FPGA 매핑[11~12]

FPGA 매핑은 일반적으로 그림 4와 같이 4단계로 진행된다. 먼저 AIG에서 K-입력 LUT으로 구현할 수 있는 컷들(K-feasible cuts)을 모두 찾아 열거하며 노드별로 저장한다. 다음은 최소 지연 매핑이 되도록 각 노드의 대표 컷을 결정하여 저장한다. 마지막으로 면적 최소화를 진행한 후 최종 네트워크를 생성시킨다.

최소 지연 FPGA 매핑은 그림 5와 같이 각 노드들을 위상적 정렬 순서로 처리하며 각 노드에서 경로 지연이 최소가 되는 컷을 찾아 그 지연 시간과 함께 대표 컷(bestCut)으로 저장한다. 지연 시간은 getDelay() 함수에서 컷의 노드들 중 가장 긴 시간에 1을 더한 값으로 계산한다.

면적 최소화는 그림 6과 같이 두 개의 루프로 구성되는데 먼저 면적류 (Area Flow)를 계산하여 각 컷이 면적 전역에 미치는 영향을 파급시켜 면적을 최소화한 후 두 번째는 국소적으로 면적을 정확히 계산하여 다시 최소화 한다. 두 경우 모두 역시 위상학적 정렬 순서로 각 노드들을 처리한다.

면적류는 각 노드 및 그 하위의 면적을 팬-아웃 노드들이 균등 분배하여 계산하는 것이고 국소 면적은 팬-

```
mapFPGA(aig, K) {
  // compute all K-feasible cuts at each node and save them
  emunerateCuts(aig, K);
  // find a min-delay cut and save it as the representative
  // at each node
  mapMinimumDelayNetwork(aig, K);
  // update the representative cut at each node to save area
  recoverArea(aig, K);
  // return the set of nodes used in the final mapping
  deriveFinalMapping(aig, K);
}
```

그림 4. FPGA 매핑
Fig. 4. FPGA mapping.

```
mapMinimumDelayNetwork(aig, K) {
    for each aig node n in topological order {
        n->bestCut = findCutMinimizingDelay(n);
        setDelay(n, getDelay(cut));
    }
}
findCutMinimizingDelay(node) {
    bestCut = NULL;
    for each cut c of node
        if(bestCut == NULL or
           getDelay(bestCut) > getDelay(c))
           bestCut = c;
    return bestCut;
}
getDelay(cut) {
    maxDelay = INFINITE;
    for each node m in cut
        maxDelay = max(maxDelay, getDelay(m));
    return maxDelay + 1;
}
```

그림 5. 최소 지연 FPGA 매핑
Fig. 5. Depth-oriented FPGA mapping.

```
recoverArea(aig, K) {
    computeRequiredTimes(aig);
    for each aig node n in topological order {
        n->bestCut
            = findCutMinimizingAreaFlow(n);
        setDelay(n, getDelay(cut));
    }
    computeRequiredTimes(aig);
    for each aig node n in topological order {
        n->bestCut
            = findCutMinimizingExactLocalArea(n);
        setDelay(n, getDelay(cut));
    }
}
```

그림 6. FPGA 매핑 면적 최소화 단계
Fig. 6. Area recovery in FPGA mapping.

아웃 노드들 중 첫 번째 노드가 계산된 면적을 모두 취하도록 한 것이다. 면적을 정확하게 계산할 때 각 노드별로 참조 계수가 사용되며 처음에 0으로 초기화된다. 각 컷이 고려될 때 재귀적 호출 방법으로 해당 컷 및

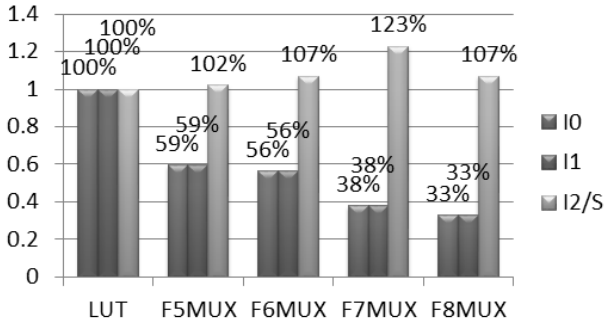


그림 7. 광폭 입력 함수 멀티플렉서의 지연 시간
Fig. 7. Delay of wide function multiplxers.

하위 노드들의 면적을 다시 계산하며 각 하위 노드가 참조될 때마다 계수가 증가되도록 하였다. 계수가 0에서 1이 되는 순간에만 그 하위 노드의 면적을 산입하고 그렇지 않는 경우는 제외한다. 다음 컷을 고려하기 전에 역시 재귀적인 호출 방법으로 하위 노드들의 참조 계수를 감소시켜 그 값을 원상 복귀시킨다.

2. 멀티플렉서 트리 매핑

상용 FPGA에 탑재된 광폭 입력 멀티플렉서 (FiMUX, i = 5, 6, 7, 8)는 LUT과는 달리 함수 기능이 고정되어 있을 뿐만 아니라 배선도 제약적이다. FiMUX의 데이터 입력 노드는 둘 다 다른 LUT이거나 또는 둘 다 다른 FiMUX이어야 연결 가능한데 FiMUX가 연결될 때는 대칭적인 균형 트리 형태로만 가능하다. 즉 두 FiMUX의 선택 신호가 같아야 하며 다단으로 구성될 때는 두 입력에 형성된 트리의 단수가 같아야 한다. 트리의 최하단은 항상 LUT들이 연결된다. 이 트리는 기본적으로 최대 4단까지만 가능하고 더 큰 트리를 구현해야 할 경우 5번째 단 멀티플렉서들을 LUT으로 구현한 후 다시 4단의 트리를 FiMUX로 더 쌓아 올리는 형식으로 계속 확장할 수 있다. FiMUX들은 이미 FPGA 칩 상에 배치되어 있고 광폭 입력 함수 구현을 위해 사용되지 않으면 그대로 낭비된다. 따라서 최대 지연 시간을 악화시키지 않는 한 FiMUX들을 많이 사용하여 LUT의 사용량을 줄일 수 있어야 한다.

또한, 상용 FPGA는 FiMUX 셀 내부 소자 크기를 조정하여 선택 입력 (S)에서 출력까지의 경로 지연이 좀 커지더라도 데이터 입력 (I0, I1)에서 출력까지의 경로 지연은 짧게 조정해 놓았다. 그림 7에 각종 FiMUX의 종류별 경로별 지연을 LUT과 비교하여 요약하였다. FiMUX들은 LUT 사용을 감축시킬 뿐만 아니라 셀 자체의 지연이 작은 경로는 시스템의 성능도 향상시킨다.

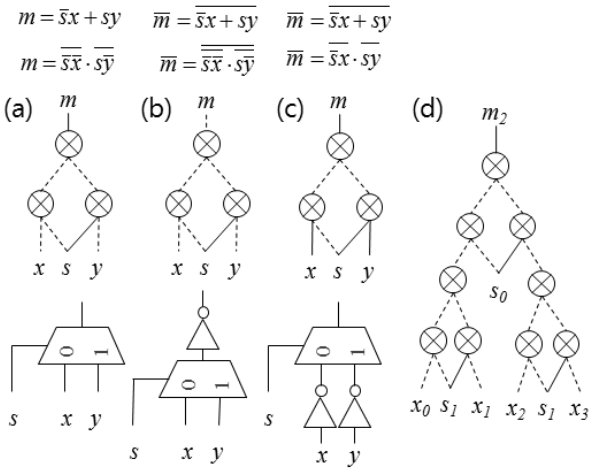


그림 8. AIG에서 멀티플렉서 인식 및 중화
Fig. 8. Multiplexer recognition and neutralization in AIG.

특히 FiMUX들은 전용 지역 배선으로 연결되기 때문에 배치 결과에 따라 지역성를 보장하지 못하는 LUT 연결보다 배선 지연이 작고 고정된 값이 유지된다는 장점을 추가로 가진다.

가. 멀티플렉서 인식

그림 8(a)에 멀티플렉서를 AIG로 나타내었다. 각 노드는 AND 게이트이고 에지는 입출력 관계 및 인버터를 표시한다. 점선 에지에는 인버터가 있고 직선에는 없다. 그림 8(d)와 같이 멀티플렉서를 2단으로 구성하면 하단의 두 멀티플렉서 출력은 반전된 것처럼 보이지만 사실 이것은 상단 멀티플렉서 입력 에지의 인버터이다. 또한 그림 8(b) 및 (c)와 같이 멀티플렉서의 출력 (m), 혹은 두 데이터 입력 (x와 y)이 반전된 그래프도 나중에 인버터를 하위 또는 상위로 이동시키면 그림 8(a)와 같이 변형할 수 있으므로 멀티플렉서로 인식해야 한다.

따라서 멀티플렉서 인식 과정에서 먼저 입력이 3개인 3-컷만 확인하며 이 3-컷이 2단 AIG로 되어 있고 중간 두 에지들이 반전 상태인가 확인한다. 또한 최하단 에지 4개중 2개가 하나는 반전, 하나는 비반전 상태로 같은 노드(s)에 연결되어야 한다. 기존의 매핑 알고리즘을 사용하되 컷 발생 후 3-컷에 대해서만 위 멀티플렉서 조건을 확인하여 인식되면 두 데이터 입력과 선택 입력을 구별하는 정보를 별도의 구조로 저장한다. 각 노드 당 멀티플렉서가 될 수 있는 컷은 한 개밖에 없으므로 각 노드에 포인터 형태로 그 컷(muxCut)과 이 구조(mux)가 추가된다. 따라서 멀티플렉서로 인식되지 않은 노드는 이 포인터가 NULL이 된다.

```
getDelay(node, cut) {
    maxDelay = INFINITE;
    for each node m in cut
        maxDelay = max(maxDelay, getLevel(m));
    if(node->muxCut == cut) {
        mux = node->mux;
        // data input nodes
        in0 = mux->in0; in1 = mux->in1;
        in0mux = in0->mux; in1mux = in1->mux;
        if((in0->bestCut == in0->muxCut) &&
           (in1->bestCut == in1->muxCut) &&
           in0mux->flag && in1mux->flag) {
           // both children are muxes
           if(in0mux->level == in1mux->level) {
               mux->level = in0mux->level + 1;
               mux->flag = (mux->level < 4);
           } else mux->flag = false;
        } else
        if((in0->bestCut != in0->muxCut) &&
           (in1->bestCut != in1->muxCut)) {
            // neither is a mux
            mux->flag = true; mux->level = 0;
            if(isPi(in0) || isPi(in1))
                mux->flag = false;
        } else mux->flag = false;
    } else mux = NULL;
    if(mux && mux->flag)
        return maxDelay + muxDelay[mux->level];
    else return maxDelay + 1;
}
```

그림 9. 멀티플렉서 지연 및 단수 계산
Fig. 9. Calculation of multiplexer delay and level.

나. 멀티플렉서 매핑

LUT만 다루던 매핑 문제에 멀티플렉서 트리가 추가되면 그림 9와 같이 각 컷의 지연 시간을 계산하는 getDelay() 함수가 복잡해진다.

멀티플렉서의 지연 시간은 그림 7에 보인 바와 같이 단수에 따라 다른 값 (muxDelay[mux->level])이 사용되며 주변 노드의 매핑 상황에 따라 단수가 동적으로 변화한다. 먼저 컷을 구성하는 노드들의 지연 시간 중 가장 큰 값을 구한 후 컷이 멀티플렉서로 인식된 것인지 확인되면 (node->muxCut == cut) 인식된 멀티플렉서 (mux)가 트리 구성 요건을 만족하는지 확인하고 그 단수도 계산하며 단수에 따라 지연 시간을 결정하여 더한다. 그렇지 않으면 기존 방식대로 LUT의 지연시간인 1을 더한다.

트리가 구성되려면 멀티플렉서의 데이터 입력 노드 (in0와 in1)는 둘 다 같은 단수의 멀티플렉서이던지 둘 다 LUT이어야 한다. PI일 경우 (isPi(in)) 직접 멀티플렉서에 연결될 수 없다. 노드들의 위상학적 정렬 순서로 매핑이 진행되기 때문에 멀티플렉서 역시 하위 단이 먼저 매핑되고 그 위에 쌓이는 상위 단이 다음에 매핑되므로 상위 멀티플렉서의 단수는 하위 단 멀티플렉서의 단수를 증가시켜 얻을 수 있다. 컷이 멀티플렉서로 인식되더라도 단수 제한 (mux->level < 4)이나 비대칭 등의 이유로 실제로는 LUT으로 매핑되는 경우도 있다. 면적 최소화 과정에서 나중에 주변 조건이 바뀌면 다시 멀티플렉서로 매핑될 수도 있기 때문에 이를 구분하는 플래그 (mux->flag)가 필요하며 단수 (mux->level)와 함께 멀티플렉서 정보 구조에 기록된다.

이 플래그는 컷의 면적을 계산할 때도 사용된다. 면적 계산 전에 항상 getDelay()를 호출하여 지연 시간이 요구 시간보다 작은지 먼저 확인하기 때문에 이때 트리 구성 요건이 확인되어 이미 이 플래그에 기록되므로 이를 다시 확인할 필요 없이 이 플래그가 참이면 멀티플렉서가 매핑되는 것이므로 해당 컷의 면적을 0으로 계산하고 그렇지 않으면 LUT의 면적 1로 계산하면 된다.

다. 멀티플렉서 중화 (Neutralization)

멀티플렉서는 그림 8(a)의 2단 AIG에 매핑되지만 그림 8(b) 및 (c) 같은 AIG도 인버터를 상위 혹은 하위로 이동시키면 그림 8(a)와 같은 형태가 되므로 역시 멀티플렉서가 매핑될 수 있다. 인식된 3-컷을 그림 8(a)의 패턴과 일치시키는 과정에서 부적합한 인버터를 트리 밖으로 밀어 내는 중화가 시작된다. 트리 밖으로 밀려나간 인버터들을 다른 인버터와 만나 소멸되거나 LUT의 함수를 반전시키면서 흡수된다. 잔존하는 인버터는 입력이 1개인 LUT를 추가로 사용하여 구현된다. 중화는 매핑 후에 진행된다.

3. 멀티플렉서 트리 구조 조성

resyn2[13] 같은 스크립트를 수행하여 최적화된 AIG에는 멀티플렉서가 매핑될 수 있는 구조가 많지 않다. ABC 명령 중 AIG를 BDD 형태로 만드는 collapse 같은 명령을 사용하면 멀티플렉서 구조가 많이 나타나지만 트리의 루트 (root)가 항상 외부 출력 (PO: Primary Output)에서 시작하고 BDD 변수 순서에 따라 트리 크기와 균형이 달라지기 때문에 최적화된 AIG를 LUT만으로 구현한 매핑에 비해 지연 시간 및 면적이 큰 경우가 많다. 따라서 FRAIG로 BDD 형태의 AIG와 최적화된 AIG를 겹쳐 놓더라도 특정 작은 예제를 제외하고는 대부분 최적화된 AIG 중심으로 LUT들이 주로 매핑된다.

AIG 상에서 멀티플렉서 트리의 위치를 PI 쪽으로 이동시키면 트리 자체가 여러 개로 복사될 수 있고 PO 쪽으로 이동시키면 그 팬-아웃 쪽에 있던 논리가 그 팬-인 쪽에 여러 개로 복사될 수 있다. 따라서 이 트리의

위치는 대개 상위 수준 연산 최적화 단계에서 결정하며 게이트 수준 합성에서는 이동시키지 않는다. 흩어지면 다른 논리와 섞여 멀티플렉서 트리로 매핑하기 어렵고 최적 위치에서 이탈하면 부적합한 복사가 발생할 수 있기 때문이다. 따라서 collapse 명령처럼 무작정 PO를 루트로 시작하여 멀티플렉서 트리를 조성하지 말고 AIG 중간에 가장 적합한 위치를 루트로 잡아 주어야 한다. 불행히도 최적의 멀티플렉서 트리 루트 노드를 AIG 상에서 자동으로 찾아내는 알려진 방법은 없다.

가. 보조 출력 (AO) 및 보조 선택 (AS) 신호

그러나 멀티플렉서 트리는 주로 주소로 참조되는 메모리에서 나타나기 때문에 RTL 코드에서 메모리를 합성해 내는 구문이나 배열 형식의 수식 표현으로부터 루트 위치를 찾아 AIG 상에 기록할 수 있다. 실제로는 상위 수준 연산 최적화가 수행된 RTL 코드로부터 찾아낸다. 멀티플렉서 트리 구조는 루트 노드의 함수를 새넌 확장하여 구성할 수 있는데 이 때 기준 변수를 지정해야 한다. 이 기준 변수들도 역시 RTL 코드에서 찾아낼 수 있다.

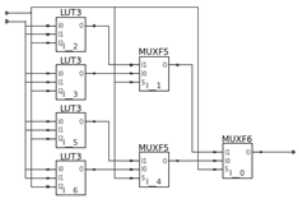
그림 10(a)에 1차원 벡터 입력에서 한 비트를 선택하여 출력하는 Verilog 수식이 멀티플렉서 트리로 구현된 예를 보이고 있다. 첨자를 사용하는 배열 형태의 변수 (d1[a1]) 표현에서 멀티플렉서 구조의 AIG가 합성되는데 그 루트를 보조 출력 (AO: Auxiliary Output) 형태로 외부로 인출한다. AO는 AIG 내에서의 표현이나 최적화 과정에서 PO와 거의 동일하게 취급된다. 따라서 AIG 상에서 유실되지 않고 항상 그 위치를 확인할 수 있다. 그러나 FPGA 매핑을 할 때 PO와 달리 이 노드를 구현하여 외부로 인출하지는 않는다. 배열 형태의 변수에서 첨자를 구성하는 비트들 (d1[a1]에서 a1[2], a1[1], a[0]에 해당)은 보조 선택 (AS: Auxiliary Select) 신호가 되며 이들 역시 AO와 같이 취급되지만 각 AO가 가진 선택 신호 목록에 포함된다. 이 목록의 신호들은 AO를 새넌 확장할 때 기준 변수로 사용된다.

그림 10(b)는 1-입력, 2-출력 레지스터 파일을 Verilog로 기술하고 그것을 FPGA의 분산 메모리로 합성한 회로를 나타내고 있다. 분산 메모리는 16 비트 메모리를 내장한 LUT들을 여러 개 사용하고 디코더 및 멀티플렉서를 확장하여 구현한 메모리이다. 이 때 출력 측에 멀티플렉서 트리가 형성되는데 이에 대해서도 AO와 AS 목록을 추출할 수 있다.

(a) 8-to-1 multiplexer

```
module mt8p1 (a1, d1, y1);
    input   [2:0]   a1;
    input   [7:0]   d1;
    output          y1;

    assign y1 = d1[a1];
endmodule
```



(b) 2-read 1-write 128-entry register file

```
module dpRegfile128(clk, store, result, rz, z, ra, a);
  input         clk, store;
  input  [1:0]  result;
  input  [6:0]  rz, ra;
  output [1:0]  a, z;
  reg    [1:0]  rf[127:0];

  assign a = rf[ra];
  assign z = rf[rz];
  always @(posedge clk)
    if(store) rf[rz] <= result;
endmodule
```

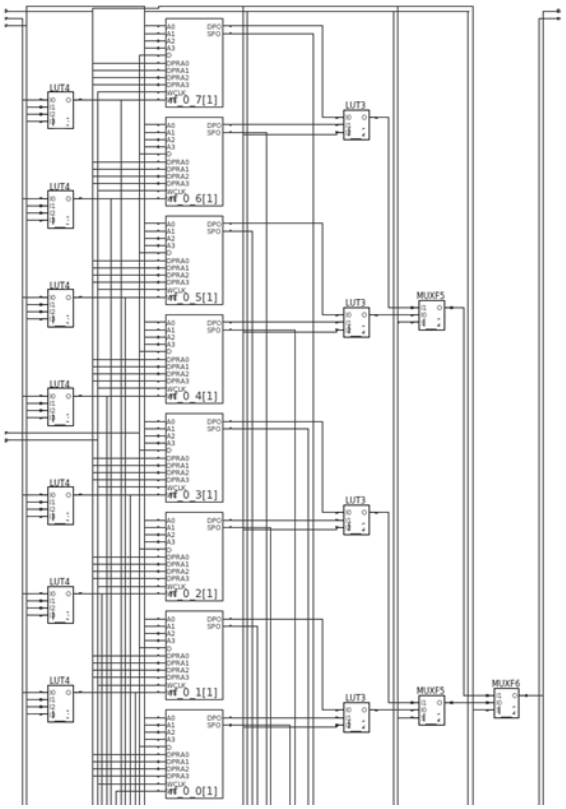


그림 10. 멀티플렉서 트리 구조 조성
Fig. 10. Construction of multiplexer trees.

나. 보조 출력 (AO)의 새넌 확장

그림 11에 보조 출력 (AO)을 새넌 확장하는 알고리즘을 기술하였다. 기존 AIG로부터 PI들을 복사하는 것으로 새로운 AIG 생성을 시작한 후 각 AO에 대해 멀티플렉서 트리를 생성시킨다. 노드가 새로운 AIG에 복사 될 때마다 기존 AIG 노드에도 사본을 포인터 (copy)로 기록해 둔다. 위상학적 정렬 순서로 진행되기 때문에 AND 노드가 복사될 때는 기존 노드의 두 자식 노드 (in0, in1)에 사본이 이미 기록되어 있으므로 이 사

```
shannonExpand(aig) {
  newAig = createAig(getPIs(aig)); p = getManager(newAig);
  AO = getAOsInTopologicalOrder(aig);
  for each o in AO {
    AS = getASs(o);
    ASC = getFaninConeNodesInTopologialOrder(AS);
    for each n in ASC
      if(isAND(n))
        n->copy = AND(p, n->in0->copy, n->in1->copy);
    for(j = 0; j < |AS|; j++) save[j] = AS[j]->copy;
    // calculate cofactors
    AOC = getFaninConeNodesUpToASsInTopologicalOrder(o);
    m = 2^|AS|;
    for(k = 0; k < m; k++) {
      for(j = 0; j < |AS|; j++)
        AS[j]->copy
          = (1 << (|AS| - j - 1)) ? const1 : const0;
      for each n in AOC
        if(isAND(n))
          n->copy = AND(p, n->in0->copy, n->in1->copy);
      cofactor[k] = o->copy;
    }
    for(j = 0; j < |AS|; j++) AS[j]->copy = save[j];
    // construct a multiplexer tree
    for(j = 0; j < |AS|; j++) {
      s = AS[|AS| - j - 1]; m /= 2;
      for(k = 0; k < m; k++)
        cofactor[k] = AND(p,
          NOT(AND(p, NOT(cofactor[2 * k]), NOT(s))),
          NOT(AND(p, NOT(cofactor[2 * k + 1]), s)));
    }
    o->copy = cofactor[0];
  }
  PO = getPOs();
  POC = getFaninConeNodesUpToASsAndAOsInTopologicalOrder(PO);
  for each n in POC
    if(isAND(n))
      n->copy = AND(p, n->in0->copy, n->in1->copy);
  return newAig;
}
```

그림 11. 보조 출력의 섀넌 확장
Fig. 11. Shannon expansion of auxiliary outputs.

본들을 단순히 AND 시키면 된다. 먼저 AO의 선택 신호인 AS들의 팬-인 콘 (Fanin Cone)을 복사한 다음 AS 이하는 포함하지 않는 AO의 팬-인 콘을 사용하여 AS들의 대한 코팩터 (Cofactor)들을 계산한다. AS의 개수가 |AS|일 때 AS[0], AS[1], ...AS[|AS|-1]에 00...0, 10...0, ..., 11...1과 같은 이진 값을 구성하는 상수 노드를 차례로 지정한 후 AO의 팬-인 콘 내의 노드들을 위상학적 정렬 순서로 복사하는 과정을 반복하면 $2^{|AS|}$개의 코팩터가 계산된다. 이들의 인접 쌍들을 멀티플렉서로 묶고 다시 인접 멀티플렉서 출력 쌍들을 멀티플렉서로 묶는 형식으로 멀티플렉서 트리를 만들면 그 루트 노드가 AO가 된다. 이 때 멀티플렉서의 선택 입력은 AS들이 차례로 연결된다. 코팩터 계산을 위해 AS의 사본 포인터에 상수 노드를 덮어 써야 하는데 그 전에 save[j]에 임시 보관했다가 코팩터 계산이 끝나면 멀티플렉서 트리 구축을 위해 복구한다. 마지막으로 각 PO를 AS 및 AO 이하를 포함하지 않는 팬-인 콘 내의 노드들을 위상학적 정렬 순서로 복사하여 구현하면 AO들이 섀넌 확장된 AIG 그래프가 완성된다.

resyn2[13] 같은 스크립트로 최적화된 AIG에 보조 출력이 섀넌 확장된 AIG를 FRAIG한 후 FPGA 매핑을 실시하면 최악의 경우에도 최적화된 AIG만 가지고 매핑했을 때와 같은 지연 시간을 달성할 수 있으며 다행히도 멀티플렉서 트리가 임계 경로에 합성되어 LUT들을 치환한다면 이 지연 시간을 더 단축할 수 있을 것이다. 추가적으로 면적 최적화 과정에서 많은 LUT들이 멀티플렉서로 치환됨으로써 LUT 사용을 줄이고 FPGA 사용 효율을 높일 수 있게 된다. 지금까지 FRAIG를 사용한 무손실 합성은 일반적인 논리 구조를 다양화 시켜서 겹친 것에 불과했다면 본 논문의 시도는 특별한 논리 구조를 의도적으로 합성하여 최적 AIG에 겹침으로써 최적 한계를 더 넓혔다는데 그 의미가 있다. 저자는 기존에 이런 시도가 논문으로 발표된 적은 없다고 파악하고 있다. 이 접근 방법은 멀티플렉서 트리뿐만 아니라 캐리 체인 등 다양한 전용 구조를 매핑하는데도 확장 응용될 수 있을 것이다.

Ⅲ. 실 험

제안된 AIG에서 멀티플렉서 트리 패턴 인식, FPGA 매핑에 멀티플렉서 트리 합성의 통합, 보조 출력의 정의 및 보조 출력의 섀넌 확장 기능을 미 버클리 대학의 ABC 논리 합성 및 검증 패키지를 확장 구현하였으며 ABC는 다시 OpenAcess[14] 데이터베이스 기반으로 하고 Verilog, EDIF, XDL 등 각종 인터페이스가 구축되어 있는 K-FPGA 논리합성 시스템에 플러그-인 되어 있다. 멀티플렉서 트리 루트 및 선택 신호 추출 기능은 RTL Verilog 구문 해석기에 구현되었다.

제안된 기능의 효과를 확인하기 위해 2개의 프로세서를 사용하였다. 먼저, NANO는 256-엔트리, 3-포트, 레지스터 파일을 분산 메모리로 구현하고 있는 간단한 32-비트 RISC 프로세서이며 표 1에 매핑 결과를 보이고 있다. 각 행에 모듈별로 LUT만 매핑한 경우와 (lo), 멀티플렉서 트리 합성을 매핑에 결합한 경우 (mts)를 비교하고 있으며 각 열에는 입력수별 LUT 사용량, 단수별 멀티플렉서 사용량, 플립플롭 사용량, 그리고 분산 메모리 사용량을 나타내고 있다. 마지막 두 열에는 총 면적 (LUT 총 사용량)과 지연 시간 (LUT 단수로 환산) 을 표시하고 있다. FPGA의 사용 면적은 LUT 및 FF의 사용량에 의해 결정된다. FF의 개수는 변화가 없기 때문에 LUT의 총 사용량만 비교했다. 마지막 행에 NANO 프로세서 전체의 계층구조를 평탄화 (Collapse)

표 1. NANO 프로세서의 FPGA 매핑 결과
Table 1. Mapping results of the NANO processor.

Design		LUT				MUXF			FD	RAM 16X1D	Area	Delay
		1	2	3	4	5	6	7				
statem	lo		5	1					2		6	1
	mts		5	1					2		6	1
alu	lo		69	198	748				192		1015	17
	mts	161	52	197	630	161	2	1	192		1040	9
reg_file	lo		116	69	1076					1024	1261	5
	mts		4	844	115	385	192	80		1024	963	5
ren_reg	lo		1	2	2				1		5	2
	mts			1	3				1		4	2
wen_reg	lo		2	1	2				1		5	2
	mts		1		3				1		4	2
pcu	lo		7	4	44				25		55	3
	mts	2	2	8	44	2			25		56	3
pc_ counter	lo		19	23	76				32		118	12
	mts	28	4	37	45	28			32		114	11.59
ir_reg	lo			33					32		33	2
	mts			33					32		33	2
data_in _reg	lo			33					32		33	2
	mts			33					32		33	2
data_out _reg	lo			33	2				32		35	3
	mts			33	2				32		35	3
address _reg	lo			64	34				32		98	3
	mts			32	34				32		66	3
mem_ pause_reg	lo				1				1		1	1
	mts				1				1		1	1
NANO	lo		184	312	2095				382		2591	17
	mts	228	71	1142	826	612	192	80	382		2267	11.59
	mts (mux tree synthesis) / lo (luts only)										0.875	0.682

표 2. DLX 프로세서의 FPGA 매핑 결과
Table 2. Mapping results of the DLX processor.

Design		LUT				MUXF				FD	Area	Delay
		1	2	3	4	5	6	7	8			
IF	lo			34	267					161	301	12
	mts	32	57	47	219	34				161	355	11.6
ID	lo		1211	1773	5179					1252	8163	23
	mts	3	1138	2361	1881	2718	768	384	160	1252	5383	23.6
EX	lo		103	116	436					82	655	23
	mts	1	93	190	379	1				82	663	24
MEM	lo		78		32					110	110	1
	mts		78		32					110	110	1
DLX	lo		1394	1972	5844					1605	9210	23
	mts	33	1373	2525	2517	2758	768	384	160	1605	6448	24
	mts (mux tree synthesis) / lo (luts only)										0.70	1.04

하여 매핑한 결과를 나타내고 있다.

이 예제에서 mts는 lo에 비해 지연 시간 31.8% 단축과 면적 12.5% 감소의 효과가 있었다. 지연 시간 단축은 ABC의 resyn2 스크립트가 찾아내지 못했던 균형 멀티플렉서 트리가 섀넌 확장 과정에서 alu 모듈에 강제로 주입되어 임계 경로의 LUT들을 치환함으로써 발생했다고 판단된다. DLX는 IF, ID, EX, MEM의 명령 처리 스텝을 가지고 있고 레지스터는 개별 플립플롭으로 구현하고 있는 잘 알려진 32-비트 RISC 프로세서이며 표 2에 매핑 결과를 보이고 있다. 표 1과 같은 형식으로 나타내었으며 지연 시간이 근소하게 (4%) 증가했지만 30%의 면적이 감소했다. 이 지연 시간 변화는 사후 처리로 수행되는 멀티플렉서 중화 과정에서 추가되는 LUT에 의한 것으로 판단된다.

Ⅳ. 결론 및 향후 과제

먼저, 광폭입력함수 전용 멀티플렉서가 슬라이스 구조에 포함되는 상용 FPGA의 현실적 제약 조건을 학계의 대표적 논리 표현 방식인 AIG를 근간으로 개발된 FPGA 매핑 알고리즘에 적용하였다. AIG를 LUT으로 매핑할 때 중간 구조로서 컷을 열거하는 데 이들 중에서 멀티플렉서를 인식해 낸 후 이들이 매핑될 때 지연 시간 및 면적을 복잡도 증가 없이 계산하도록 하였다. 이 때 트리를 형성할 때 필요한 대칭성과 단수 제약 요건도 검사하도록 하였다.

또한, 멀티플렉서 트리의 루트 위치를 RTL 코드에서 찾아내고 이를 보조 출력 형태로 AIG에 추가하도록 하였다. 이 위치에서 섀넌 확장을 통해 멀티플렉서 트리 구조를 의도적으로 합성한 후 최적 AIG에 겹치도록 하는 접근 방법을 최초로 제안하였다. 이 때 무손실 합성을 가능하게 하는 FRAIG 방식이 응용되었다.

두 가지 프로세서에 대해 제안된 접근 방법과 기법들을 적용하였으며 약 13~30%의 면적 감소를 달성했다. 지연 시간은 최대 32%까지 단축되었으나 후처리 과정에서 근소하게 커질 수도 있다.

향후, FPGA 패브릭 (Fabric) 상의 전용 게이트들을 활용하기 위해 AIG 트리에 특정 구조를 의도적으로 주입시키는 접근 방법을 캐리 체인 등에 확장 적용하는 연구가 진행될 것이며 후처리 과정의 지연 시간 오버헤드가 매핑 과정에서 고려될 수 있도록 알고리즘을 향상시킬 예정이다.

REFERENCES

[1] *ABC: A System for Sequential Synthesis and Verification*. Berkeley Logic Synthesis and Verification Group, http://www.eecs.berkeley.edu/~alanmi/abc/abc.html, October, 2007.

[2] V. Betz and J. Rose, "VPR: A New Packing, Placement And Routing Tool For FPGA Research," in *Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications*. pp.213 - 222, 1997.

[3] L. Amaru, P-E Gaillardon, and G.D. Micheli, "Majority-Inverter Graph: A New Paradigm for Logic Optimization," IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, 2016.

[4] N. Steiner, A. Wood, H. Shojaei, J. Couch, P. Athanas, M. French, "Torc: Towards Open-Source Tool Flow," in *Procee0dings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp.41-44, February, 2010.

[5] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings, "RapidSmith: Do-It-Yourself CAD Tools for Xilinx FPGAs" in *Proceedings of the 21st International Workshop on Field-Programmable Logic and Applications*, pp.349-355, September, 2011.

[6] *Xilinx Design Language Version 1.6*, Xilinx, Inc., Xilinx ISE 6.1i Documentation in ise6.1i/help/data /xdl, July 2000.

[7] K. Kim, "Evaluation Toolkit for K-FPGA Fabric Architectures," *Journal of the IEEK*, vol. 49-SD, no. 4, pp.157-167, April, 2012.

[8] K. Kim, "Pre-Packing, Early Fixation, and Multi- Layer Density Analysis in Analytic Placement for FPGAs, *Journal of the IEEK*, vol. 51-SD, no. 10, pp.96-106, October, 2014.

[9] *Spartan-3 Generation FPGA User Guide*, UG331, v1.6, Xilinx Inc., December 3, 2009.

[10] A. Mishchenko, S. Chatterjee, R. Jiang, and R. K. Brayton, "FRAIGs: A unifying representation for logic synthesis and verification". ERL Technical Report, EECS Dept., UC Berkeley, March 2005.

[11] A. Mishchenko, S. Chatterjee, and R. Brayton, "Improvements to Technology Mapping for LUT-Based FPGAs," IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, Vol. 26, Issue 2, pp.240-253, February, 2007.

[12] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton, "Combinational and Sequential Mapping with Priority Cuts," Proc. of IEEE/ACM International Conference on Computer-Aided Design, pp.354-361, November, 2007.

[13] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-Aware AIG Rewriting: A Fresh Look at Combinational Logic Synthesis," Proc. of Design Automation Conference, pp.532-535, July, 2006.

[14] M. Guiney, E. Leavitt, "An Introduction to OpenAccess: an Open Source Data Model and API for IC Design," in *Proceedings of Asia and South Pacific Conference on Design Automation*, pp. 434-436, January, 2006.

저 자 소 개

김 교 선(정회원)
1986년 연세대학교 전자공학과 학사 졸업.
1988년 연세대학교 전자공학과 석사 졸업.
1998년 Ph.D. Department of Electrical & Computer Engineering, University of Massachusetts, Amherst, U.S.A.
1988년～2003년 삼성전자 CAE Center 주임, 선임, 책임, 수석연구원.
2003년～현재 인천대학교 전자공학과 교수
<주관심분야: 상위수준합성, 논리합성, FPGA 매핑, FPGA 배치, Reconfigurable Computation, Fault-Tolerance, Embedded Systems, Low-Power Design, Nanoelectronic Architectures>