논문 01-26-7B-18

한국통신학회논문지 '01-7 Vol.26 No.7B

최적의 MUX-based FPGA 설계를 위한 하드웨어 할당 알고리듬

정회원 인 치 호*

A Hardware Allocation Algorithm for Optimal MUX-based FPGA Design

Chi-ho Lin* *Regular Member*

요 약

본 논문에서는 ASIC 벤더의 셀 라이브러리와 MUX-based FPGA에 있는 고정된 입력을 갖는 연결구조의 수를 최소화하는 하드웨어 할당 알고리듬을 제안한다.

제안된 할당 알고리듬은 연산자간을 연결하는 신호선이 반복적으로 이용되어 연결 신호선 수가 최소가 될 수 있도록 연산자를 할당한다. 연결 구조를 고려한 이분할 그래프에 가중치를 설정하고 변수와 레지스터간의 최대 가중치 매칭을 구함으로서 레지스터 할당을 수행한다. 또한 연결구조에 대한 멀티플렉서의 중복 입력을 제거하고 연산자에 연결된 멀티플렉서 간의 입력을 교환하는 입력 정렬 과정으로 연결구조를 최소화한다.

벤치마크 실험을 통하여 제안된 알고리즘의 효용성을 보인다.

ABSTRACT

In this paper, we propose a new hardware allocation algorithm to minimize the number of multiplexers with the predetermined inputs in the ASIC cell libraries or the multiplexer-based FPGAs.

In the proposed algorithm, the operation are allocated to functional units so that the number of interconnection wires between functional units can be minimized. By considering the structure of the interconnection, the bipartite graph is constructing for each edge in order to minimize the interconnection between functional units and registers. Finally, the interconnection is minimized by removing the duplicated inputs of multiplexers and exchanging the inputs across multiplexers.

The efficiency of the proposed allocation algorithm is shown by experiments using benchmark examples.

I. 서 론

상위 레벨 합성은 설계 하고자 하는 시스템의 동작을 알고리즘 레벨의 특정 언어(Hardware Description Language, HDL)로 기술하는 동작 기술, 하드웨어 기술 언어를 분석하는 HDL 분석기(analyzer), 연산(operation)을 특정한 제어 스텝에 할당하는 스케줄링(scheduling), 그리고 상위 레벨 합성의 마지막 단계인 하드웨어 할당(allocation)으로 구성된다[1-5].

하드웨어 할당은 구현되는 하드웨어의 면적이 최소가 되도록 연산 (operation)을 기능 연산자(functional unit)에, 변수(variable)를 메모리에 지정하고 메모리와 연산자 사이의 연결 구조(interconnection)로 버스(bus)나 멀티플렉서(multiplexer)를 할당하는 것이다. 이들 할당 과정과 바인딩 과정은 상호 종속적인 관계에 놓여 있어서 하나의 요소가 결정되면 다른 요소에 영향을 미치기 때문에 다른 요소의 할당에 제약 조건이 된다. 이와 같이 할당

* 세명대학교 컴퓨터과학과 부교수, E-mail : ich4102venus.semyung.ac.kr

논문번호: K01091-0228, 접수일자: 2001년 2월 28일

과정에서 야기되는 문제를 해결하기 위한 기존의 할당 알고리즘은 크게 2가지 형태로 분류 할 수 있다[9-16]. 첫 번째 접근 방법은 제어 스텝을 분리해서 하나의 제어 스텝에서 이들 자원을 동시에 고려하면서 할당하는 것으로서 한 제어 스텝에서는 최적의 해를 구할 수 있으나 전체적으로 제어 스텝이 진행됨에 따라 최적의 해에서 벗어날 수 있다. 여기에 속하는 할당 방법으로는 greedy 방법이 있으며 MABAL[10]에서 사용하고 있다. 두 번째 접근 방법은 기능 연산자 할당, 메모리 할당과 연결구조 바인딩을 분리해서 수행하는 방법으로서 이는 각 자원간의 연결관계를 고려하는 것이 문제가 되지만 할당 문제를 분리해서 수행하므로 시간 복잡도(time complexity)가 줄어든다. 대표적인 방법으로는 clique partitioning 방법[11]이 있다.

그러나 하드웨어 할당 방법은 전체적인 하드웨어 비용에 큰 영향을 미치기 때문에 이들 자원을 최대한 공유하는 할당 방법이 중요한 문제로 부각되고 있다. 1980년대 중반에 발표된 REAL[3], FACET[4] 등에서 제시된 할당 알고리즘은 연산자 사용의 최소화 또는 메모리 사용의 최소화만을 목표로 하여 연결구조에 대한 고려가 없었다. 그러나 연결구조의 비용 증가가 전체 회로의 단가에 미치는 영향이 커짐에 따라 할당 시 연결 구조의 최소화에 대한 반영이 요구되고 있다. 이에 따라 메모리 사용의 최소화보다는 연결구조의 최소화를 고려한 할당 알고리즘들이 제안되었다[8,10].

본 논문의 하드웨어 할당 알고리즘은 하드웨어 할당이 전체적인 하드웨어 비용에 큰 영향을 미치기 때문에 이들 자원을 최대한 공유하는 할당 방법, 즉 메모리와 연결구조의 최소화가 될 수 있는 할당 방법을 제안하였다.

Ⅱ. 최적의 MUX-based FPGA 설계를 위한 하드웨어 할당 알고리듬

2.1 하드웨어 구조 설정

ASIC 벤더의 셀 라이브러리나 MUX-based FPGA에 있는 연결 구조는 입력 수가 고정된 멀티플렉서에 의해 연결 구조를 구성한다. 그림 1은 MUX-based FPGA의 예로 미국 Xilinx사가 개발한 XC3000 시리즈의 CLB(Combinational Logic Block) 내부구조를 보여주고 있다. 한 개의 CLB(이하 "논리 블럭"이라 함)는 sum of product 수에 관계없이 5입력 변수의 조합 논리 회로를 구현할 수 있다. 따라서 1개의 논리 블럭으로 그림 2(a)와 같이 3개의 데이타 입력(IN1, IN2, IN3)과 2개의 선택 신호(S0, S1)로 구성되는 멀티플렉서를 구현할 수 있다. 그림 2는 멀티플렉서 입력에 따른 논리 블럭의 사용 수를 보여주고 있다. 그림 2에서 보듯이 Xilinx FPGA의 특성상 3입력 멀티플렉서와 4입력 멀티플렉서의 차이는 단순히 1개의 입력 수 차이가 아니라 1개의 논리 블럭이 더 사용됨을 나타낸다.

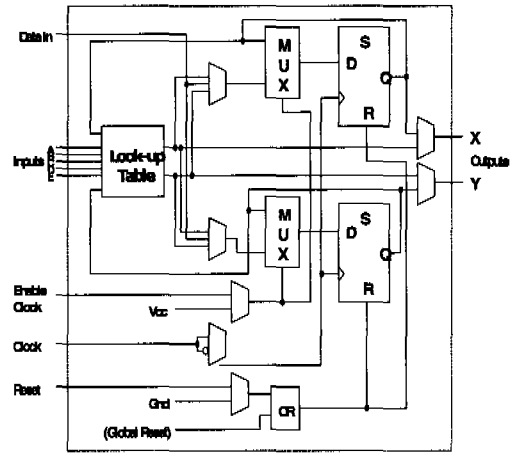
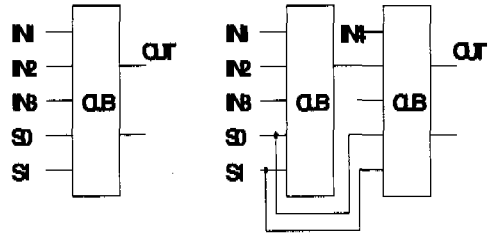


그림 1. XC3000 family의 CLB 논리도



(a) 3 입력 멀티플렉서 (b) 4입력 멀티플렉서

그림 2. 논리 블럭에 의한 멀티플렉서 구현

따라서 MUX-based FPGA를 목표로 하여 연결구조를 최소화하고자 할 때는 연결 구조의 입력 수보다는 연결 구조로 사용되는 논리 블럭의 수를 줄이는 것이 더 효율적이다. 따라서 본 논문에서는 연산자에 연결되는 멀티플렉서의 크기를 최소로 하는 것이 목적이 아니고, 고정된 입력의 멀티플렉서를 최소한으로 사용하는 것을 목적으로 한다. 목표 아키텍춰는 그림 3과 같이 다단으로 멀티플렉서를 이용하여 연결 구조를 구현하였다. 그림 3과 같이 프로세스간에 공유하는 변수(전역 변수)는 전역 레지

스터(global register)에 할당하고 하나의 프로세스에서만 사용하는 국부 변수(local variable)은 그림 3의 REG 1과 같이 각 프로세스 내에 할당되는 레지스터에 할당된다. 각 프로세스는 시스템 클럭 또는 별도의 클럭에 동기 되며 전역 레지스터의 경우는 시스템 클럭에 의해 구동된다.

2.2 최적의 MUX-based FPGA 설계를 위한 하드웨어 할당 알고리듬

본 논문의 할당 방법은 고정된 입력의 연결 구조를 갖는 FPGA나 ASIC 라이브러리를 지원하기 때문에 연결 구조 수의 최소화를 목적 함수로 한다. 예를 들면 그림 4(a)와 같이 레지스터 R1의 입력 수가 2이고 입력 선택을 위해 4:1 멀티플렉서가 할당되어 있다고 가정하자. 변수 V1이 R1에 새로이 할당될 경우 V1의 입력이 되는 V3가 레지스터 R2에 할당되어 있다면 그림 4(b)와 같이 입력 수가 3으로 증가한다. 입력 수가 증가하더라도 입력 연결 구조를 위해 멀티플렉서의 증가는 이루어지지 않는다.

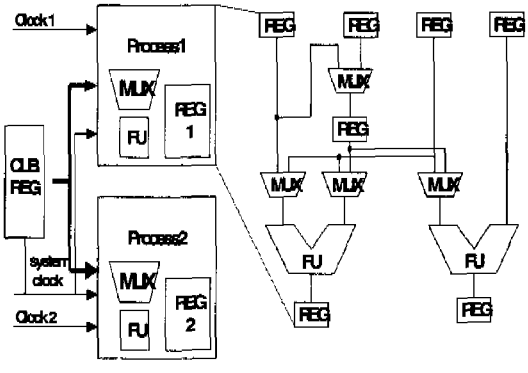


그림 3. 목표 아키텍쳐

따라서 변수 V1이 레지스터 R1에 할당될 경우 연결 신호선 수는 입력 신호와 멀티플렉서 제어 신호의 증가로 기존에 할당된 신호선에 비해 2만큼의 증가분이 발생하지만 멀티플렉서의 수가 증가하지 않으므로 비용(cost)는 0이 된다.

2.2.1 기능 연산자 할당

기능연산자 할당 과정은 연산을 기능연산자에 할당하는 과정으로서 기능연산자의 수는 스케줄링 단계에서 결정된 것으로 한다. 할당에 사용되는 연산자는 다기능 연산자가 아닌 단일 기능을 갖는 연산형 별(예, +, *, 카운터 등)로 할당을 수행한다. 연산을 기능 연산자에 할당하는 방법에 따라 연결 구조가 변하므로 연결 구조를 최소화할 수 있도록 연산을 기능 연산자에 할당하여야 한다. 본 논문에서 제안한 기능 연산자 할당은 그림 5와 같이 두개의 연산 종류에 대해 따로 수행한다. 즉, 1개의 연산만이 존재하는 경우와 2개 이상의 연산이 존재하는 경우이다.

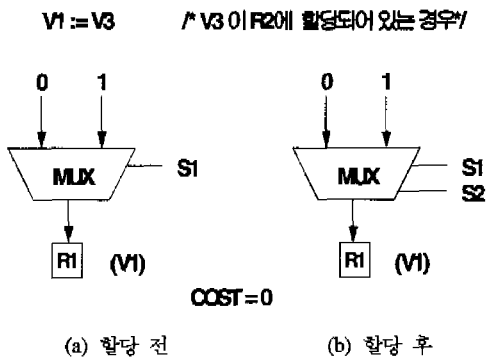


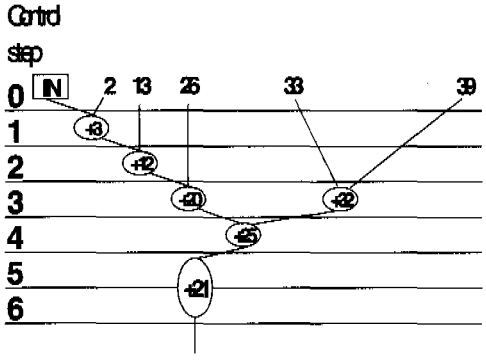
그림 4. 비용 계산 예

```
for(각각의 특정 형의 연산)
    특정 형의 연산을 기능 연산자에 할당;
    for(각각의 제어 스텝)
    for(남은 각 형의 연산에 대하여)
        비용 함수 값 계산;
        최대 비용 값을 갖는 연결 구조 선택;
        연산을 기능 연산자에 할당;
```

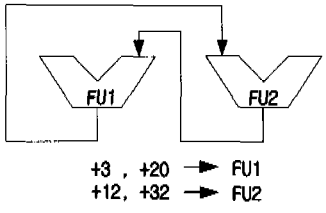
그림 5. 기능 연산자 할당 알고리듬

특정 형(type)의 연산은 그림 6(a)에서 제어 스텝 5에 있는 *21과 같이 하나의 제어 스텝에 올 수 있는 같은 형의 연산의 수가 1개인 연산으로 이 연산을 수행하는 기능연산자를 우선적으로 할당한다.

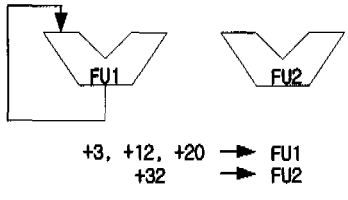
남은 형의 연산은 그림 6(a)에서 제어 스텝 3에 있는 +20과 같이 하나의 제어 스텝에 올 수 있는 같은 형의 연산의 최대수가 2이상인 연산으로 이와 같은 연산이 할당 될 수 있는 기능연산자 수가 N이고 하나의 제어 스텝에 오는 이 연산의 수가 M인 경우에 연산을 기능연산자에 할당하는 방법은 N순열 M의 가지 수가 있고 할당하는 방법에 따라 연결 구조가 변하게 된다. 그림 6(b)와 6(c)는 기능연산자의 할당 형태에 따라 연결구조가 변하는 예를 보여준다.



(a) CDFG(Control Data Flow Graph)의 예



(b) 기능 연산자의 할당



(c) 연결 구조를 고려한 기능 연산자의할당

그림 6. 기능 연산자 할당의 예

그림 6(b)는 연산 +3과 +20을 기능연산자 1에, 연산 +12와 +32를 기능연산자 2에 할당한 경우이고 그림 6(c)는 +3, +12 와 +20을 기능연산자 1에 +32를 기능연산자 2에 할당한 경우이다. 그림 6(c)와 같이 할당할 때 연결 구조가 최소가 될 수 있다. 그림 6(c)의 할당은 그림 6(a)에서 보면 한 연산이 할당된 기능 연산자와 그 연산의 입력과 출력이 되는 연산이 할당된 기능연산자가 같은 경우로서 기능 연산자 할당과정에서 연결 구조를 최소로 하기 위해 비용을 식(1)같이 설정하고 하나의 제어 스텝에서 연산이 기능 연산자에 할당될 수 있는 경우에 대하여 각각의 비용을 계산한 후 비용이 가장 큰 경우로 연산을 할당한다. 스케줄링된 결과의 첫 번째 제어 스텝에서 마지막 제어 스텝까지 식(1)을 이용하여 모든 연산을 기능 연산자에 할당한다.

$$\cos t = \sum_{x=1}^{N}\sum_{y=1}^{M}((\text{같은입력}FU\text{수} + \text{같은출력}FU\text{수}) - (\text{다른입력}FU\text{수} + \text{다른출력}FU\text{수})) \quad (1)$$

(N = 같은 type의 연산이 할당되는 기능연산자의 수)
(M = 각 기능연산자에 할당된 연산의 수)

2.2.2 메모리 할당

메모리 할당 과정은 레지스터의 수를 결정하고 연산의 출력인 변수를 레지스터에 할당하는 과정으로서 변수가 가지는 일반적인 특성과 제약조건에 의해 크게 좌우된다. 변수의 일반적인 특성은 하나의 입력이 되는 연산을 가지고 하나 이상의 출력이 되는 연산을 가지며 life time을 가진다. 기능 연산자 할당과 레지스터 할당간의 종속 관계 때문에 기능연산자 할당 과정 후, 변수는 하나의 입력이 되는 기능 연산자와 하나 이상의 출력이 되는 기능연산자를 가진다는 제약 조건이 부가된다. 이와 같이 변수의 성질 및 제약조건 때문에 생겨나는 연결 구조를 최소화하고 레지스터의 공유를 최대화하기 위한 레지스터 할당 알고리듬은 그림 7과 같다.

```
변수들을 레지스터 입력 별로 분리;
  for(각각의 분리된 변수 집합)
      life time이 전체 제어 스텝인 변수들을
      레지스터에 할당;
      남은 변수들을 life time에 따라 정렬;
      남은 변수들을 life time 중첩에 의해
      clustering;
      변수들에 대표값 부여;
      weighted bipartite graph을 이용하여
      변수들을 레지스터에 할당;
```

그림 7. 메모리 할당 알고리듬

변수들을 레지스터에 할당하는 과정은 먼저 모든 변수들을 변수의 입력이 되는 기능 연산자별로 분리한다. 표 1은 elliptic wave filter의 예를 변수의 입력이 되는 기능 연산자별로 분리한 예이다. 이와 같이 분리된 변수들에 대해 각각의 레지스터 수를 결정하고 분리된 변수들을 각 결정된 수만큼의 레지스터에 할당한다. 이는 기능 연산자의 출력과 레지스터의 입력 사이에 멀티플레서를 지원하지 않고 레지스터의 출력과 기능연산자의 입력 사이에만 멀티플렉서를 지원함으로서 멀티플렉서의 입력이 공유

되는 경우를 증가시키게 되고 따라서 연결 신호선 수가 줄어든다.

표 1. 변수들을 분리한 예

FU1의 출력이 되는 변수	V22 V12 V27 V7 V4 V8 V14 V15 V17 V9 V21 V11 V18 V19 V26
FU2의 출력이 되는 변수	V32 V13 V28 V24 V29 V5 V31 V25 V2 V33
FU3의 출력이 되는 변수	V16 V23 V10 V30 V6 V20 V35 V34

각각의 분리된 변수 집합에 대해 변수가 전체 제어 스텝의 중간에서 발생되고 또한 life time이 전체 제어 스텝인 변수들의 출력이 되는 변수들은 초기에 각각 하나씩의 레지스터를 할당한다. 이는 본 논문의 레지스터 할당 알고리듬이 이러한 변수들을 분리하여 처리하기 때문에 우선적으로 레지스터에 할당하면 레지스터의 공유를 최대화할 뿐 아니라 변수를 life time에 따라 정렬하는 과정에서 변수가 분리되는 것을 방지한다. 분리된 변수 집합에 대해 변수 정렬을 하게 된다. 변수의 정렬 과정에서는 그림 8과 같이 변수들을 생성 시간과 life time에 따라 정렬한다. 즉 생성 시간이 증가하는 순서로 정렬하고 생성 시간이 같은 변수들은 life time이 감소하는 순서로 정렬한다.

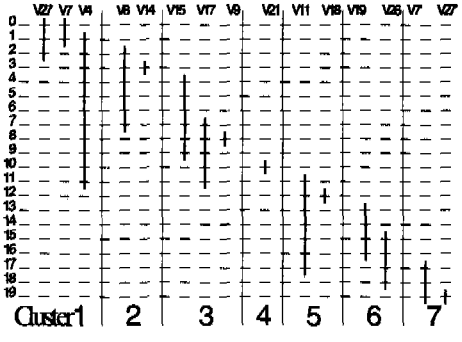


그림 8. 변수들을 정렬한 후 clustering한 예

이와 같은 정렬방법은 시간 복잡도 면에서 우수하나 그림 8의 변수 V27과 변수 V27'와 같이 하나의 변수가 둘로 분리되어 변수 V27과 변수 V27'가 같은 레지스터에 할당되지 않는다고 가정하면 하나의 레지스터에서 다른 레지스터로 데이터 천이가 일어나고 데이터가 전달되는 쪽의 레지스터 입력단에 멀티플렉서가 필요하게 되어 전체 연결 구조는 늘어날 수 있는 단점을 가진다. 본 논문에서는 weighted bipartite graph에 의해 변수를 레지스터로 할당하는 과정에서 이러한 단점을 보완한다.

변수의 정렬 과정이 끝나면 변수들을 clustering한다. Clustering 방법은 그림 8과 같이 cluster 1을 만들고 정렬되어 있는 첫 변수를 cluster 1에 놓고 다음 변수가 cluster 1에 있는 모든 변수들과 life time이 최소한 하나의 제어 스텝 이상 중첩되면 그 변수를 cluster 1에 놓고 overlap되지 않으면 cluster 2를 만들어서 그 변수를 cluster 2에 놓는 방법으로 정렬되어 있는 모든 변수를 clustering한다. 예를 들면 변수 V27을 cluster 1에 할당하고 변수 V7을 고려하면 V7의 life time이 V27의 life time과 중첩되므로 cluster 1에 할당한다. 그러나 V8의 경우 V8의 life time이 cluster 1에 있는 V27과 V4에는 중첩되지만 V7과는 life time의 중첩이 없으므로 새로운 cluser 2에 할당한다. 모든 변수들을 cluster로 분할한 후 각 변수에 대표값을 부여한다.

표 2. 변수들의 대표값

변수	입력 operation (입력FU)	출력 operation (출력 FU)	Life-Time	대표값
V8	+12 (FU1)	+20 (FU1) +19 (FU1) +11 (FU1)	6	1
V15	+25 (FU1)	*21 (FU3) *24 (FU3) +22 (FU3)	6	0

표 2와 같이 변수의 출력이 되는 연산이 할당된 기능연산자가 모두 같은 경우는 기능연산자의 인덱스로 대표값을 부여하고 기능 연산자가 다른 경우는 0으로 대표값을 부여한다. 0이 아닌 특정 값으로 대표값을 갖는 변수가 레지스터에 할당되면 그 레지스터의 출력과 대표값을 인덱스로 가지는 기능연산자의 입력사이에는 연결 신호선이 생기지만 반복적으로 사용될 가능성이 있다. 그러나 0을 대표값으로 가지는 변수는 연결 신호선 수를 증가시키는 요인이 된다.

변수를 레지스터에 할당하는 과정은 먼저 할당되지 않은 변수들의 life time을 조사하여 life time이 가장 많이 중첩되는 제어 스텝에서 변수의 life time이 중첩된 수 만큼 레지스터의 수를 할당한다. 레지스터의 수가 결정되면 그림 8의 cluster 1의 변수

들을 각각의 레지스터에 하나씩 할당하고, 레지스터는 할당된 변수의 대표값을 갖게된다. Cluster 1의 변수가 할당된 후 나머지 cluster에 있는 변수들은 weighted bipartite graph를 이용한 최대 매칭에 의해 레지스터에 할당된다.

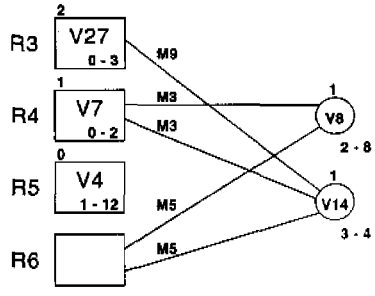


그림 9. 레지스터 할당을 위한 weighted bipartite graph

그림 9는 그림 7의 cluster 2에 있는 변수들을 할당하는 과정의 weighted bipartite graph로서 다음과 같이 변수들을 레지스터에 할당한다. Cluster 1에 있는 변수들이 레지스터에 할당된 후 레지스터들과 cluster 2의 변수들 사이에 bipartite graph를 형성한다. 에지(edge)는 레지스터에 할당된 변수의 life time과 cluster 2에 있는 할당될 변수의 life time이 중첩되지 않으면 레지스터 노드와 변수 노드는 에지를 가진다. 레지스터와 변수의 대표값과 특성에 의해 에지에 가중치를 부여하고 가중치가 큰 에지에서 작은 에지 순으로 양쪽 노드들 사이에 레지스터 할당이 이루어지며 가중치가 같을 경우에는 레지스터 노드와 변수 노드의 life time 차이(gap)가 적은 변수가 할당이 이루어진다. 할당될 변수 중에서 에지를 가지지 않는 변수가 있으면 레지스터를 하나 증가시킨 후 다시 할당 과정을 수행하며 하나의 cluster에 있는 모든 변수들이 할당되면 레지스터의 대표값을 수정한 후 다음 cluster에 있는 변수들과 레지스터들 사이에 bipartite graph를 구성하고 위의 과정을 반복한다.

레지스터 노드와 변수 노드 사이의 에지 가중치는 레지스터 수와 연결 신호선 수의 최소화를 위해 설정되었고 그림 10과 같다.

그림 10에서 보면 가중치가 가장 큰 에지(M1)는 할당 과정에서 레지스터 수의 증가를 최소화하며 다음으로 큰 가중치를 가지는 에지(M2)는 정렬과정에서 하나의 변수가 둘로 분리되어 레지스터 입력쪽 멀티플렉서가 생기는 것을 최소화한다. 나머지 에지 가중치는 연결 구조의 공유가 많은 쪽에서 적

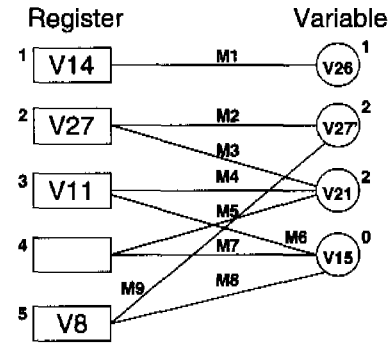


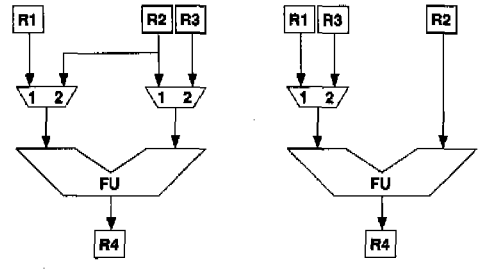
그림 10. 레지스터와 변수사이의 에지 가중치

은 쪽으로, 연결 신호선 수의 증가가 적은 쪽에서 큰 쪽으로 부여된다.

2.2.3 입력 정렬 과정

연결구조의 공유를 최대화하여 연결 구조 할당의 비용 함수를 최소화하는 단계로서 메모리의 할당이 종료되면 각 연산자에 연결되는 레지스터가 결정된다. 따라서 멀티플렉서가 사용되는 수도 결정이 된다. 그림 4에서 레지스터 R1이 변수 V1에 데이타 종속관계를 갖는 후행 노드들이 할당된 레지스터의 입력단도 변하기 때문에 멀티플렉서의 증가분은 데이타 종속관계에 따른 연결 구조 변화를 비용에 반영하기 위한 값이다. 따라서 본 논문에서는 연결구조 바인딩은 고정된 입력의 멀티플렉서를 최소화함을 목적으로 하며, 설정된 목표 구조는 멀티플렉서의 입력을 줄이기 위하여 멀티플렉서의 입력을 정렬 (alignment)을 수행한다.

멀티플렉서의 입력 정렬은 덧셈기나 곱셈기와 같이 입력의 위치에 관계없이 연산을 수행할 수 있는 기능 연산자에 대해 두 입력의 위치를 교환함으로서 멀티플렉서의 입력 수를 최소화시키며 동시에 연결구조 수를 최소화시킨다.



(a) 입력의 정렬 전 (b) 입력의 정렬 후

그림 11. 멀티플렉서 입력 정렬

그림 11(a)는 레지스터 할당후의 멀티플렉서 입력의 예로서 제어 스텝 1에서 한쪽의 멀티플렉서 입력은 레지스터 1의 출력이고 다른 한쪽의 멀티플렉서 입력은 레지스터 2의 출력이다. 교환법칙의 성질을 이용하여 제어 스텝 1의 멀티플렉서 입력을 바꾸어 주면 그림 11(b)와 같이 1개의 2입력 멀티플렉서의 수 가 감소된 회로를 얻을 수 있다. 따라서 입력 정렬 과정은 연결 구조에 대한 하드웨어 비용을 줄이기 위해 필요하다.

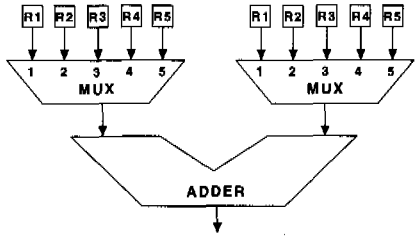


그림 12. 멀티플렉서 입력의 정렬 예

이와 같이 멀티플렉서의 수 및 입력을 최소화하기 위한 입력 정렬에 대한 설명을 하기 위해 그림 12와 같이 덧셈기에 연결된 입력이 있다고 가정하자.

그림 12는 각 레지스터의 연결상태를 나타내어 한 개에 5 입력 멀티플렉서에 의해 연결 구조를 구현할 수 있다고 볼 수 있지만 실제 각 레지스터가 연결되는 시간은 그림 13(a)와 같다. 그림 13(a)와 같이 연결된 입력에 대해 그림 13(b)와 같이 레지스터가 덧셈기의 좌, 우 입력에 나타나는 레지스터의 빈도 수를 계산한 후 레지스터의 수가 1인 레지스터를 제외하고 레지스터의 총빈도 수가 적은 쪽에서 큰 쪽으로 정열하여 총빈도수가 가장 작은 레지스터를 선택한다. 선택된 레지스터는 그림 11에서와 같이 위치를 입력수를 줄이기 위해 위치를 이동하게 된다.

제어스텝	입력	
	좌측	우측
1	R1	R3
2	R1	R5
3	R5	R1
4	R3	R2
5	R3	R4
6	R2	R3
7	R3	R4
8	R4	R1
9	R4	R5
10	R4	R3

(a) 제어스텝별 입력

	총빈도수	좌측	우측
R1	4	2	2
R2	2	1	1
R3	6	3	3
R4	5	3	2
R5	3	1	2

(b) 레지스터별 연결

그림 13. 그림 12. 의 입력 연결 상태

그림 13에서 총빈도수가 가장 적은 레지스터 R2를 선택하여 입력의 위치를 교환한 결과는 그림 14와 같다. 그림 14에서 보면 레지스터 R2가 좌측 입력에서만 나타나므로 우측 입력의 수가 1만큼 감소하였다. 그림 13에서 위치를 이동할 레지스터를 선택하면 총빈도수가 적은 레지스터 R5가 선택된다. 선택된 레지스터가 이동하기 위해서는 같은 제어스텝에서 연결된 반대쪽 입력 레지스터와 위치 교환을 하여야 한다. 따라서 선택된 레지스터의 연결빈도가 많은 쪽 입력에서 적은 쪽 입력으로 이동하게 되면 상대적으로 많은 레지스터들의 교환이 발생하게 된다. 따라서 연결빈도가 적은 쪽의 입력을 이동하게 된다. 그림 14에서 선택된 레지스터 R5는 좌측 입력에서 우측 입력으로 이동한다. 이와 같이 멀티플렉서의 수 및 입력을 최소화 과정을 반복하면 그림 15와 같이 연결구조인 멀티플렉서의 크기가 5입력에서 3입력으로 감소하였음을 알 수 있다.

제어스텝	입력	
	좌측	우측
1	R1	R3
2	R1	R5
3	R5	R1
4	R2	R3
5	R3	R4
6	R2	R3
7	R3	R4
8	R4	R1
9	R4	R5
10	R4	R3

(a) 제어 스텝별 입력 연결

	총빈도수	좌측	우측
R1	4	2	2
R2	2	2	0
R3	6	2	4
R4	5	3	2
R5	3	1	2

(b) 레지스터별 연결

그림 14. 레지스터 R2의 위치 교환

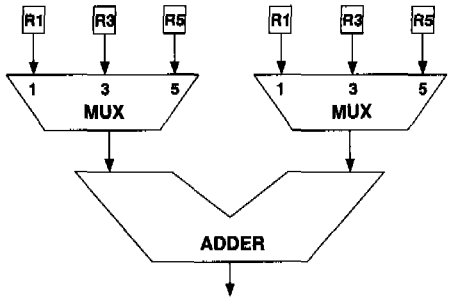


그림 15. 멀티플렉서 입력 정렬후의 결과

이와 같이 멀티플렉서의 수 및 입력을 최소화하기 위한 연결구조 정렬 알고리즘은 그림 16과 같다.

```
{
  for(교환이 되는 할당된 각 functional unit에 대하여)
      각 register가 multiplexer의 입력과 연결되는 수를 계산.
      register가 나타나는 수가 증가하는 순으로 정렬.
      for(정렬된 register에 대하여)
          현재의 multiplexer 입력 수보다 작도록 교환.
}
```

그림 16. 연결 구조 바인딩 알고리즘

Ⅳ. 실험 결과

하드웨어 할당 알고리즘은 Sparc 1+에서 C&C++로 구현하였으며 벤치마크 회로에 대해 다음과 같이 실험을 하였다.

표 3과 표 4는 fifth-order elliptic filter 와 sixth-order bandpass filter에 대한 실험 결과를 보여준다. 하드웨어 할당에서는 제어 스텝의 수 와 연산자의 수는 기존의 다른 시스템과 같으나, 연결구조의 비용을 고려한 멀티플렉서 입력 수의 측면에서 표 3의 HAL [12]과 비교할 때 약 30%만 을 필요로 하며, 표 4의 MAP [14]과 비교 할 때 약 67%만을 필요로 하는 결과를 보여 주고 있다. 그러나 본 하드웨어 할당이 주로 멀티플렉서를 연결 구조로 사용했기 때문에 버스 측면에서는 유리한 결과를 얻지 못했다.

표 5와 표 6은 벤치마크인 AM2910과 AM2901에 대한 [2,16]의 예의 실험 결과를 비교하여 보여준다. [2,16]은 벤치마크 회로가 ISPS (Instruction Set Processor Specifications) 언어로 기술되어 있을 때 나온 결과로서 본 논문에서 적용한 벤치마크 회로와는 기술 형태가 다르지만 실험의 결과에 대한 참고 자료로써 제시하였다. 표 5와 표 6에서 본 논문의 결과와 [16]의 연결 구조 형태인 멀티플렉서 입력의 수를 비교 할 때 표 5에서는 약 54%, 표 6에서는 약 62% 만을 필요로 한다. [2]와 [16]의 경우 여러 비트들의 레지스터를 사용하고 있지만 본 하드웨어 할당의 경우 레지스터 할당 시 비트별 처리를 하면서 멀티플렉서의 입력 수가 작은 것을 사용하도록 하였다. 본 논문의 결과는 표 5에서는 4개의 레지스터를 사용하여 [16]과 총 레지스터의 비트 수를 비교 할 때 약 81%를 사용하였으며 표 6에서는 12개의 레지스터를 사용하여 약 91%의 총 레지스터 비트 수만을 필요로 한다. 그러므로 본 논문의 하드웨어 할당 결과는 작은 수의 논리 블록에 의해 연결 구조가 구현됨을 보여주며, 하드웨어 할당의 경우 칩 면적을 감소시킬 수 있다.

표 3. Fifth-order elliptic filter에 대한 실험 결과

	본 논문	HAL[12]	Map[14]	SPAID[15]
#C_steps	19	19	19	19
#ALUs	2	2	2	2
#Multip-liers	1	1	1	1
#MUX_inputs	9 / (4×MUX21)	26	10	17
#Buses	5	6	5	N/A
#Registers	12	12	14	19

표 4. Sixth-order bandpass filter에 대한 실험 결과

	본 논문	Map[14]	ADPS[16]	PUBSS[17]
#C_steps	11	11	11	11
#ALUs	2	2	2	2
#Multip-liers	1	1	1	1
#MUX_inputs	8 / (4xMUX21)	12	27	10
#Buses	5	5	N/A	5
#Registers	11	11	14	11

표 5. AM2910에 대한 실험 결과

	본 논문 방법	Emerald [2]	CMU [16]
#C_steps	3	-	-
#ALUs	1	1	1
#Multip -liers	N/A	N/A	N/A
#MUX_ inputs	120 / (30×MUX21)	229 / (61×MUX21)	135 / (36×MUX21)
#Registers	5 / (2×6bit-reg., 3×12bit-reg.)	5 / (1×9bit-reg. 2×12bit-reg. 2×13bit-reg.)	5 / (1×9bit-reg. 4×12bit-reg.)
#Total bits	48	59	57

표 6. AM2901에 대한 실험 결과

	본 논문 방법	Emerald [2]	CMU[16]
#C_steps	3	-	-
#ALUs	1	1	1
#Multipliers	N/A	N/A	N/A
#MUX_ inputs	108 / (27×MUX21)	180 / (48×MUX21)	120 / (32×MUX21)
#Registers	11 (9×4bit-reg. 2×5bit-reg.)	13 (2×1bit-reg. 5×4bit-reg. 5×5bit-reg. 1×9bit-reg.)	15 (4×1bit-reg. 10×4bit-reg. 1×9bit-reg.)
#Total bits	46	56	53

MUX21 : 2-input multiplexer,
bit-reg. : bit register,
N/A : Not Available.

V. 결 론

최근 FPGA가 개발되어 많이 사용됨에 따라 논리 합성에서는 FPGA를 지원하기 위한 방법이 개발되어 사용되고 있으나 상위레벨 합성에서는 현재 고려하지 못하고 있다. 따라서 본 논문에서 제안한 하드웨어 할당 알고리듬은 일정한 형태의 아키텍춰를 갖는 FPGA를 목표 칩으로 하여 개발되었기 때문에 ASIC 보다는 FPGA를 이용한 회로 설계에의 응용이 기대된다.

제안된 하드웨어 할당 알고리즘은 연결구조의 최소화와 연결구조의 비용을 고려한 레지스터 및 멀티플렉서 입력 수를 최소화 할 수 있었다. 또한 실제 하드웨어를 목표 칩으로 하여 비용 함수를 설정함으로서 단순히 연결구조의 입력 수보다는 실제 구현되는 하드웨어의 단가를 줄이고자 하였다.

앞으로의 연구 과제로는 계층 설계에 대한 지원과 RAM이나 FIFO 같은 메모리 소자에 대한 합성 방법의 개발이다. 특히 논리 합성 툴 들이 사용하고 있는 하드웨어 모델링 기법을 적용함으로서 사용자가 의도한 회로를 쉽게 설계할 수 있도록 하는 연구가 더 진행되어야 한다.

참 고 문 헌

[1] A. R. Newton, "Computer-Aided Design of VLSI Circuits", Proceedings of the IEEE, Vol. 69, No. 10, pp.1189-1199, Oct. 1981.

[2] A. R. Newton and A. L. Sangiovanni-Vincentelli, "Computer-Aided Design for VLSI Circuits", IEEE Computer, pp.38-63, April 1986.

[3] F. J. Kurdahi and A. C. Parker, "REAL: A Program for Register Allocation," Proc. of the 24th ACM/IEEE Design Automation Conference, June 1987.

[4] C. J. Tseng and D. Siewiorek," Facet: A procedure for the Automated Synthesis of Digital System," Proc. 20th DAC, pp.490-496, 1983..

[5] S. G. Shiva, "Automatic Hardware Synthesis", Proceedings of the IEEE, Vol.71, No.1, pp.76-87, Jan.1983.

[7] Daniel D. Gajski, Silicon Compilation, Addison-Wesley, 1988.

[8] M. C. McFarland, A. C. Parker, and R. Camposano, "The High-Level Synthesis of Digital Systems", Proceedings of the IEEE, Vol.78, No.2, pp.301-318, Feb. 1990.

[9] R. Camposano, "From Behavior to Structure: High-Level Synthesis", IEEE Design & Test of Computers, pp.8-19, Oct. 1990.

[10] K. Küçükçakar and A. C. Parker, "Data Path Tradeoffs Using MABAL," Proc. of the 27th ACM/IEEE Design Automation Conference, pp.511-516, 1990.

[11] C. Y. Hitchcock, D. E. Thomas, "A Method for Automated Data Path Synthesis", Proc. of the 20th ACM/IEEE Design Automation Conference, pp.484-489, 1983.

[12] R. K. Brayton, A. L. Sangiovanni-Vincentelli, and G. D. Hachtel, "Multi-level Logic Synthesis", Proceedings of the IEEE, Vol.78, No.2, pp.256-300, Feb. 1990.

[13] E. S. Kuh and T. Ohtuski, "Recent Advance in VLSI Layout", Proceedings of the IEEE, Vol.78, No.2, pp.237-253, Feb. 1990.

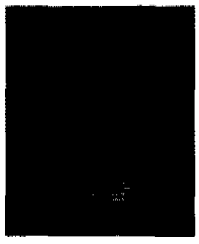
[14] F. Brewer, D. Gajski, "Chippe: A System For Constraint Driven Behavioral Synthesis", IEEE Trans. on CAD Design, Vol.9, No.7, July. 1990.

[15] P. Paulin, J. Knight and E. Girczyc, "HAL: A Multi-Paradigm Approach to Automatic Data Path Synthesis", Proc. of 23rd DAC, pp.263-270, 1986.

[16] S. Devadas and A. R. Newton, "Algorithms for Hardware Allocation in Data Path", IEEE Trans. on CAD, pp.768-781, 1989.

[17] G. J. Casavant, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register Allocation via Coloring," Journal on Computer Languages, vol.6, pp.47-57, 1981.

인 치 호(Chi-ho Lin)



1985년 : 한양대학교 전자공학과 공학사
1987년 : 한양대학교 대학원 공학석사
1996년 : 한양대학교 대학원 공학박사

1992년~현재 : 세명대학교 컴퓨터과학과 부교수
<주관심 분야> VLSI CAD, ASIC 설계, CAD 알고리듬, RTOS 및 내장형 시스템