論文98-35C-7-1

DSP용 ASIP을 위한 특수 명령어 생성 기법

(Techniques for Special Instruction Generation for DSP ASIP)

金 泓 澈 *, 黃 承 浩 *

(Hong-chul Kim and Seung-Ho Hwang)

요 약

애플리케이션에 맞도록 최적화된 명령어 처리기를 디자인할 때 가장 중요한 점은 하드웨어 특성에 맞는 명령어 셋을 가지는 것이다. 이러한 명령어 셋을 디자인하는 문제는 각각의 명령어를 구현하는 방법을 결정하는 문제와 결부되면 더 복잡해진다. 본 연구에서 프로세서의 모델은 기본 명령어와 특수 명령어라 불리는 두 종류의 명령어를 지원한다. 기본 명령어들은 ALU 같은 일반적인 다기능 하드웨어로 구현된다. 특수 명령어들은 이를 처리하는 별도의 하드웨어를 필요로 하며, 실제로 이러한 하드웨어는 주 프로세서에 대한 보조 프로세서처럼 작동한다. 이 경우, 특수 명령어들과 기본 명령어들은 동시에 수행될 수 있다. 이 논문에서는 주어진 애플리케이션을 위한 특수 명령어를 생성하는 새로운 알고리즘을 제안한다. 생성된 특수 명령어의 성능 예측 시 기본 명령어와 특수 명령어의 병렬성을 충분히 고려하였다.

Abstract

The first thing in designing application-specific instruction set processor is having instruction set closely matching hardware characteristics. This instruction set design problem can be more complicated when combined with implementation method selection problem of each instruction. Our processor model supports two kinds of instructions - primitive or special instructions. Primitive instructions are implemented using common multifunctional hardware such as ALU. Special instructions require a set of dedicated hardware, which actually functions as a coprocessor to the main processor. In this case, special instructions and primitive instructions can be executed independently. In this paper, we present novel algorithm for generating special instructions for given application. Parallelism between special instructions and primitive instructions is also considered during the performance estimation stage of generated special instructions.

Ⅰ. 서 론

VLSI 기술의 발달로 인하여 여러 시스템에서 많은 ASIC(Application Specific Integrated Circuit)들이 사용되고 있다. 일반 프로세서와 비교하여 ASIC은 성능, 면적, 전력 소비 등의 여러 제약을 만족시키는데 유리하다. 그러나 응용 영역의 복잡도가 올라감에 따라 디자인 에러와 알고리즘의 변화를 수용할 수 있는 더 많은 자유도가 요구되어진다. 그러나 ASIC은 하나의 어플리케이션에 최적화 되어있으므로 디자인의 후기에 일어나는 디자인 변화를 수용하기에는 문제가 있다. 이에 반하여 프로그래머블 프로세서들은 이러한 변화를 프로그램을 변화시킴으로 쉽게 수용할 수 있다. 이러한 이유로 최근에 많은 시스템에서 ASIP (Application Specific Instruction set Processor)

* 正會員, 韓國科學技術院 電氣 및 電子工學科
(KAIST Department of electrical engineering)
接受日字:1998年3月31日, 수정완료일:1998年5月27日

이 사용되고 연구되고 있다[1].

일반적으로 하나의 ASIP은 몇 개의 서로 다른 어플리케이션을 위해 최적화된 프로그래머블 아키텍쳐를 가지고 있다. 서로 다른 어플리케이션에서 자주 사용되는 명령어는 일반적으로 서로 다르므로 주어진 제약조건(면적이나 전력 소모등)을 만족시키는 최적의 명령어 집합을 찾아 선택하는 일은 ASIP의 성능을 최대화 하는데 매우 중요하다. 이러한 이유로 주어진 여러 개의 어플리케이션을 분석하고 이들에 가장 적합한 최적의 명령어를 가지는 ASIP을 합성하는 연구가 진행되고 있다[8].

ASIP을 위한 명령어 생성에 관한 지금까지의 연구로는 파이프라인을 가지는 RISC 구조의 프로세서에서 명령어를 합성하는 연구[2]와 특수 명령어를 지원하는 특수 하드웨어 선택에 관한 연구[3] 등이 존재한다. RISC 구조의 프로세서에서 명령어를 합성하는 연구에서는 시뮬레이티드 어닐링[5]이 사용되었기 때문에 어플리케이션이 커지면 명령어를 합성하는 데 걸리는 시간이 너무 커져서 실제 문제에 적용하기 어렵고, 생성되는 명령어 중에서 어플리케이션의 수행 시간 단축을 위하여 한 사이클 내에 처리하는 연산의 개수가 많은 경우 유저가 프로그램하기에는 불편한 명령어들이 생성될 수도 있다. 특수 명령어를 지원하는 특수 하드웨어 선택의 연구에서는 브랜치 앤 바운드(branch and bound) 방법을 사용하여 해를 구하였다. 이 연구의 근본적인 한계는 어플리케이션이 주어질 때 특수 명령어를 구현할 특수 하드웨어를 합성하는 방식이 아니라 주어진 하드웨어의 라이브러리로부터 선택한다는 점이다. 이러한 점은 문제를 단순화시키는 이점이 있지만, 가능한 디자인 영역이 제한되기 때문에 주어진 어플리케이션에 맞도록 최적화된 해를 생성하지는 못한다.

본 연구는 원초적인 작업을 행하는 명령어들이 이미 주어진 상태에서 어플리케이션을 효율적으로 수행할 수 있는 특수 명령어를 합성하는 방법을 기술한다. ASIP이 가지는 프로그래머블한 특성을 최대한 살리기 위해서 DSP 애플리케이션 처리에 적합하고 유저가 사용하기 쉬운 1~2 사이클의 간단한 명령어를 직접 디자인하여 기본적인 명령어로 제공하였다. 프로그램하기는 어렵지만 시간 제약을 만족시키기 위해서 빠르게 수행되면 효과적인 부분은 특수 명령어로 처리하였다. 특수 명령어를 구현할 하드웨어는 기존의 연구처럼 하드웨어 라이브러리로부터 몇 개의 하드웨어를 선택하는 방식이 아니라 알맞은 하드웨어를 직접 합성하는 시도를 하였다. 우리가 아는 바로는 지금까지 이러한 시도를 했던 연구는 발표되지 않았다.

본 논문의 II 장에서는 대상이 되는 ASIP의 아키텍쳐에 대해 기술하고 III 장에서는 제안된 특수 명령어 생성 방법에 대하여 자세히 설명한다. IV 장에서는 실험 결과를 설명하고 V 장에서 결론과 추후의 과제를 논의한다.

II. ASIP의 아키텍쳐

시스템이 생성하는 칩의 목표 구조는 DSP 어플리케이션을 위한 프로그래머블 ASIP이다. 어플리케이션 프로그램은 외부 프로그램 메모리에 있다. ASIP은 명령어를 외부로부터 가져오고 디코딩하기 위하여 페치 유닛과 디코드 유닛을 가지고 있다. 단순한 명령어는 하드와이어드 콘트롤에 의하여 수행되고 복잡한 명령어는 마이크로롬에 의하여 제어된다. 복잡한 명령어의 디코드된 결과는 마이크로롬 내의 마이크로 루틴의 시작 어드레스가 된다.

생성되는 ASIP은 복잡도에 따라 3가지 클래스의 명령어를 지원한다. 첫째, P(primitive) 클래스는 프리미티브한 명령어와 모든 어플리케이션에서 공통으로 사용되는 필수적인 명령어를 포함한다. 그러한 예로는 간단한 산수연산과 branch와 call같은 제어 명령어가 있다. P 클래스는 간단한 수행 커널(execution kernel)과 하드와이어드 콘트롤에 의하여 합성되는 모든 ASIP에서 지원된다.

둘째, B(basic) 클래스는 P 클래스보다는 복잡하지만 타이밍이 크리티컬하지는 않은 명령어들을 포함한다. B 클래스의 명령어들은 P 클래스 명령어들의 조합으로 구현되며 마이크로 롬의 제어를 받는다. B 클래스는 외부 메모리의 크기를 줄이고 외부 메모리의 엑세스에 의한 속도 저하를 줄이기 위하여 제공된다.

마지막으로 S(special) 클래스는 복잡하고 타이밍도 크리티컬한 명령어들을 포함한다. 따라서 S 클래스의 명령어들은 S-HW라고 불리우는 특별한 전용의 하드웨어에 의하여 구현된다.

이러한 클래스 구분의 이유는 다음과 같다. 어플리케이션이 변함에 따라 B와 S 클래스에 속하는 명령어들은 주어진 제약 내에서 최대의 성능을 내기위하여

많은 변화를 일으킨다. 또한 B와 S 클래스 명령어들은 일반적으로 복잡하고 사용자가 이해하기가 쉽지 않다. 따라서 어플리케이션 프로그래머들이 그러한 명령어들을 사용하여 알고리즘의 변화나 디버깅을 위하여 합성되어 생성된 프로그램을 바꾸는 일은 쉽지가 않다. 그러나 다행히도 디자인의 후반기나 칩이 나온 후의 필요한 변화는 일반적으로 데이터 패스보다는 콘트롤 분야가 많다. 데이터 패스의 자주 사용되는 연산은 S나 B 클래스의 명령어로 매핑되고 제어는 P 클래스의 명령어로 매핑되므로 우리는 대부분의 요구되는 변화를 P 클래스의 명령어를 사용하여 수행할 수 있다. P 클래스 명령어의 단순함은 프로그래머들의 명령어에 대한 이해와 사용을 쉽게 만든다.

S와 B 클래스 명령어의 선택은 미리 정해진 제약이 없으므로 기존의 방식에 비하여 좀더 자유롭게 좀더 고성능의 ASIP을 생성할 수 있다.

이 연구에서는 일단 S 명령어라 분류된 특수 명령어를 생성하는 것을 목표로 한다. 대상으로 하는 ASIP의 하드웨어는 P 명령어를 수행하는 커널과 특수 명령어를 처리하는 별도의 하드웨어로 되어있다. P 명령어를 처리하는 커널과 특수 명령어를 처리하는 특수 하드웨어는 하드웨어를 공유하는 부분이 없으므로 동시에 수행될 수 있다. 그림 1은 이러한 ASIP의 구조를 나타낸 그림이다.

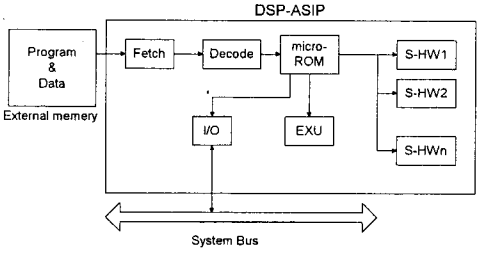


그림 1. ASIP의 아키텍춰
Fig. 1. Architecture of ASIP.

Ⅲ. 특수 명령어 생성

1. 특수 명령어 생성 과정

ASIP을 위한 특수 명령어를 생성하는 전체 흐름도가 그림 2에 나타나 있다. 구현하고자 하는 DSP 알고리즘은 SILAGE[4]로 기술되어 있다. 이것은 파서를 통해서 시그널 흐름 그래프의 형태로 바뀌어진다. 주어진 알고리즘이 유저가 원하는 제약 시간 이내에 수행될 수 있는지를 알아보기 위해서 미리 정해져 있는 P 명령어로 시그널 흐름 그래프를 매핑하여 알고리즘이 P 명령어로만 수행될 때 걸리는 시간을 예측한다. 만약 P 명령어 만으로도 충분하면 특수 명령어를 생성하지 않아도 된다. 이와 반대로 P 명령어 만으로는 주어진 제약 시간을 만족시키지 못 할 경우에는 특수 명령어를 생성하게 된다. 특수 명령어는 제약 면적의 한도 내에서 제약 시간이 만족될 때까지 생성된다. 이런 방식으로 특수 명령어가 생성되면 특수 명령어를 구현할 하드웨어 생성기를 통하여 특수 하드웨어로 만들어져서 ASIP에 통합된다.

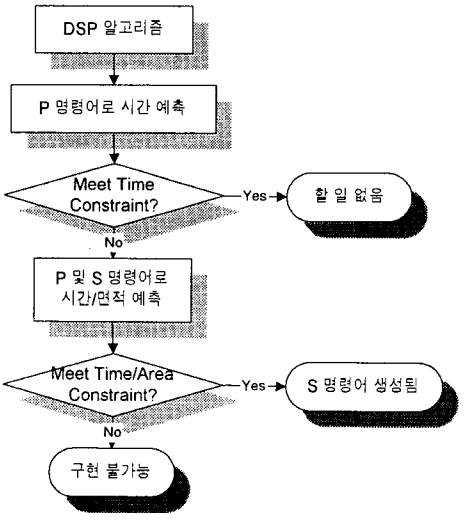


그림 2. 특수 명령어 생성 과정
Fig. 2. Process of S instruction generation.

2. 특수 명령어 생성 알고리즘

알고리즘을 나타내는 시그널 흐름 그래프로부터 특수 명령어를 생성하는 과정이 그림 3에 간략하게 표현되어있다.

시그널 흐름 그래프가 주어지면 우선 P 명령어만으로 매핑시킴으로써 알고리즘의 수행 시간을 예측한다. 특수 명령어가 필요한 경우 시그널 흐름 그래프에서 시그널을 공유하는 임의의 두 노드에 대해서 이를 특수 명령어로 만들었을 때의 시간 절약 및 면적의 증대를 조사하여[9] 면적당 시간 절약의 효과가 가장 뛰어난 것을 특수 명령어로 정한다. 이 과정을 제약 시간이 만족될 때까지 반복하는 것이 이 알고리즘의 기본을 이룬다.

알고리즘을 이루는 세부적인 과정들에는 다음과 같

은 것들이 있다. 우선 두 노드를 병합할 때 전체 시그널 흐름 그래프에서 싸이클이 형성되어 스케줄링이 불가능한 경우가 생기지 않도록 이를 체크해야 한다. 다음으로는 시그널 흐름 그래프에서 특수 명령어로 매핑된 부분을 제외한 나머지 부분을 P 명령어들로 매핑하여 S 명령어와 P 명령어를 포함한 전체 시그널 흐름 그래프를 수행하는데 얼마만큼의 시간이 소요되며, 면적은 얼마나 차지하는가를 효과적으로 예측할 수 있어야 한다.

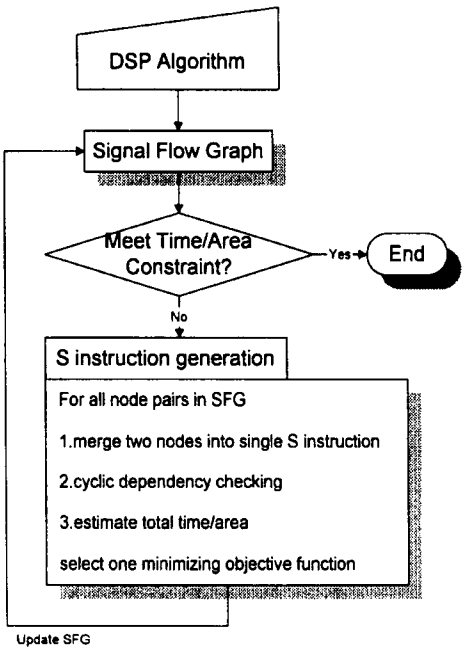


그림 3. 특수 명령어 생성 알고리즘
Fig. 3. S instruction generation algorithm.

이 논문에서는 이러한 특수 명령어 생성 과정을 간결하고 일관된 형태로 표현하기 위해서 시그널 흐름 그래프를 이진 행렬의 형태로 나타내었다. 우선 세부 알고리즘의 기술 과정에서 자주 쓰이는 기호 및 용어들을 소개한다.

1) 기호 및 용어

시그널 흐름 그래프의 각각의 노드는 특정한 자연수에 유일하게 대응된다고 할 때 다음과 같은 용어들을 정의할 수 있다.

시그널 흐름 그래프의 Edge Matrix(약칭 EM) 의 (i, j)번째 원소는 다음과 같이 정의된다.

노드 i에서 노드 j로 향한 에지가 있으면 $EM_i^j = 1$.

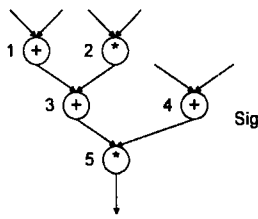
유사하게, 시그널 흐름 그래프의 Dependency Matrix(약칭 DM) 의 (i, j)번째 원소는 다음과 같이 정의된다.

노드 i에서 노드 j를 연결하는 패스가 존재하면 $DM_i^j = 1$.

그림 4는 시그널 흐름 그래프에 대한 EM과 DM을 나타낸 한 예이다.

2) 두 노드의 병합

시그널 흐름 그래프상에서 특수 명령어로 만들어질 두 노드를 단일 노드로 병합하여 이 단일 노드는 한 개의 특수 명령어에 의해서 수행됨을 나타내게 된다. 이렇게 두 노드를 하나로 병합하는 작업이 시그널 흐름 그래프에 대한 EM 및 DM에서 어떠한 형태로 표시되는가에 대해서 알아보기로 한다.



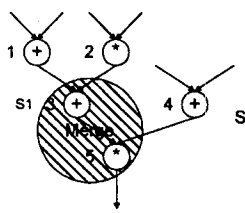
Edge Matrix

	1	2	3	4	5
1	0	0	1	0	0
2	0	0	1	0	0
3	0	0	0	0	1
4	0	0	0	0	1
5	0	0	0	0	0

Dependency Matrix

	1	2	3	4	5
1	0	0	1	0	1
2	0	0	1	0	1
3	0	0	0	0	1
4	0	0	0	0	1
5	0	0	0	0	0

그림 4. 시그널 흐름 그래프의 EM 및 DM 표현
Fig. 4. EM and DM representation of signal flow graph.



Edge Matrix

	1	2	4	S1
1	0	0	0	1
2	0	0	0	1
4	0	0	0	1
S1	0	0	0	0

Dependency Matrix

	1	2	4	S1
1	0	0	0	1
2	0	0	0	1
4	0	0	0	1
S1	0	0	0	0

그림 5. 노드의 병합
Fig. 5. Merging Nodes.

생성되는 특수 명령어를 나타내는 노드는 이 노드가 포함한 모든 노드들의 에지들을 자신의 에지로 가지게 된다. 또한 이렇게 병합된 단일 노드는 여기에 포함된 모든 노드들의 의존 관계(dependency relation)를 상속하게 된다. 즉 병합된 노드 외부의 어떤 노드가 병합된 노드 내부의 또 다른 어떤 노드보다 먼저 수행되는 노드이면 병합된 단일 노드는 이 외부의 노드가 수행된 이후에 수행되어야 한다. 그림 5는 그림 4에서 두 노드를 병합했을 때의 EM과 DM을 나타낸다.

3. 사이클 체킹

시그널 흐름 그래프에 포함된 두 노드를 하나로 병합할 때 그래프 내에 사이클이 형성될 수 있다. 그림 6은 두 노드를 병합할 때 사이클이 생기는 경우를 보여준다.

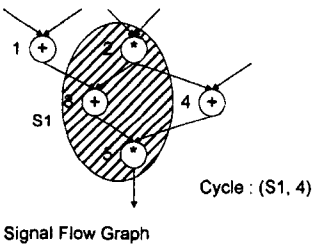


그림 6. 사이클 체킹
Fig. 6. Cycle checking.

시그널 흐름 그래프가 트리 구조가 아닌 일반적인 DAG의 형태일 때 어떤 노드의 출력이 다른 노드의 입력으로 들어가는 패스가 두 개 이상인 경우가 생길 수 있기 때문에 사이클이 형성될 수 있다. 두 노드를 병합할 때 이러한 사이클이 생기는지를 검사하는 일은 주어진 시그널 흐름 그래프의 DM로부터 쉽게 알 수 있다. 특수 명령어를 생성하는 알고리즘은 두 노드를 하나의 특수 명령어로 만드는 작업들의 반복이므로 두 노드가 병합될 때 사이클을 이루는 노드는 위의 그림에서처럼 두 개이다. 세 개 이상의 노드가 사이클에 포함된 경우는 우리가 제안하는 알고리즘에서 발생하지 않는다. 따라서 두 노드를 병합한 후 바뀐 DM에서 모든 i, j에 대하여 (i, j)번째 원소와 (j, i)번째 원소가 동시에 1인 경우가 존재하면 사이클이 발생하는 경우이다.

3. 특수 명령어 사용시 시간/면적 예측

특수 명령어를 생성할 때 생성된 특수 명령어를 사용하면 성능이 얼마나 향상되며 특수 명령어를 지원하기 위해서 필요한 하드웨어의 면적은 얼마나 되는가 하는 점을 알기 위해서 효과적인 시간 및 면적 예측 방법이 필요하다. 특수 명령어를 포함하는 ASIP의 하드웨어 구조의 특성상 특수 명령어를 수행하는 부분은 커널에서 수행되는 P 명령어와 동시에 수행될 수 있기 때문에 특수 명령어를 사용할 때의 전체 수행 시간의 줄어드는 양을 예측하는 작업은 생성된 특수 명령어들과 동시에 수행될 수 있는 P 명령어들이 얼마나 되는가를 파악하고 특수 명령어들과 P 명령어들을 함께 스케줄링하는 작업을 의미한다. 또한 특수 명령어를 지원하는 하드웨어의 면적은 유일하게 결정되는 것이 아니다. 특수 명령어가 나타내는 서브 그래프의 스케줄링 및 자원 할당이 이루어진 후 하드웨어의 면적을 알 수 있다. 따라서 생성된 특수 명령어에 의한 전체 시간 및 하드웨어의 면적을 예측하는 작업은 그림 7과 같은 여러 가지 작업을 통해서 이루어진다.

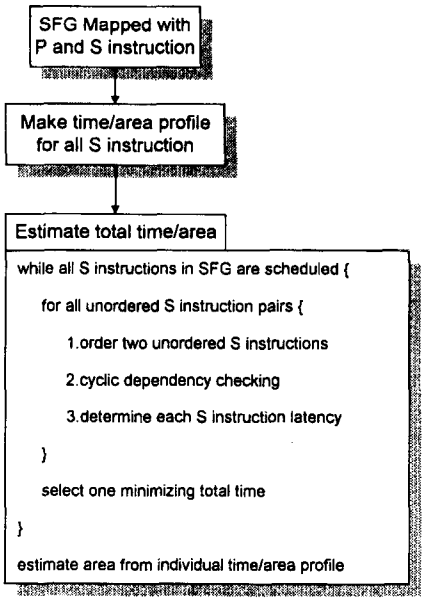


그림 7. 시간 및 면적 예측
Fig. 7. Time and area estimation.

우선 특수 명령어 각각에 대하여 스케줄링과 자원 할당 작업을 하여 수행 시간이 변함에 따라 하드웨어의 면적이 어떻게 바뀌는지를 계산한다. 이런 식으로 계산된 (시간, 면적)의 쌍들을 시간/면적 프로파일이라고 정의한다. 그리고 나서 각각의 특수 명령어에 대하여 동시에 수행 가능한 P 명령어들이 충분히 존재하

여 전체 알고리즘 수행 시간이 최소가 되도록 특수 명령어들을 스케줄링한다.

특수 명령어들의 스케줄링이 끝나면 특수 명령어 각각에 대하여 동시에 수행 가능한 P 명령어들의 개수를 예측 가능하다. 이러한 정보를 가지고 특수 명령어들의 레이턴시를 모두 결정한 다음, 시간/면적 프로파일로부터 특수 명령어들이 차지하는 전체 면적을 예측한다.

1) 시간/면적 프로파일

스케줄링 및 자원 할당이 이루어지지 않은 특수 명령어에 대한 시간/면적 프로파일이란 레이턴시를 변화시키면서 특수 명령어를 하드웨어로 구현할 때 차지하는 면적이 얼마인가를 기록한 것이다. 그림 8은 간단한 특수 명령어에 대하여 시간/면적 프로파일을 계산하는 과정을 보여준다.

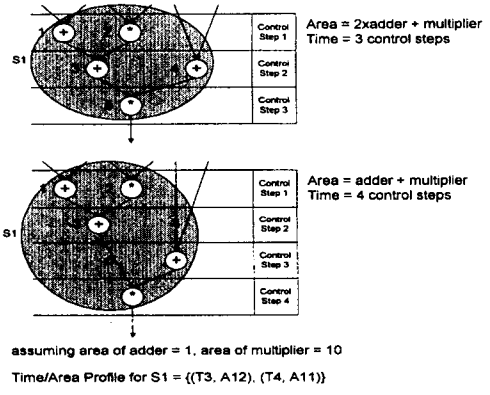


그림 8. 시간/면적 프로파일의 예
Fig. 8. Example of time/area profile.

특수 명령어의 시간/면적 프로파일을 만드는 작업은 1) 특수 명령어 그래프를 스케줄링하는 알고리즘, 2) 매핑시 사용되는 하드웨어 라이브러리, 그리고 3) 사용되는 매핑 알고리즘에 의하여 그것의 특징이 유일하게 결정된다. 이러한 프로파일을 만드는 목적은 주어진 상황에 제일 적합한 특수 명령어의 하드웨어 구현 방식을 능동적으로 결정하기 위함이다.

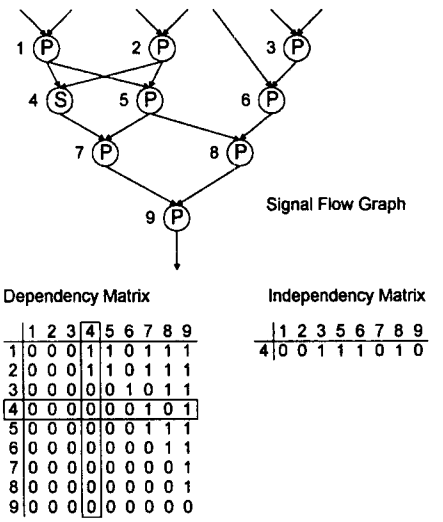
2) 특수 명령어의 레이턴시 결정

특수 명령어를 사용할 때 절약되는 시간은 다음과 같이 계산한다.

시간 절약 = 특수 명령어가 포함한 MOP의 개수
- 특수 명령어의 레이턴시
+ 동시에 수행 가능한 P 명령의 개수

위의 식이 나타내는 의미는 다음과 같다. 특수 명령어를 수행하는 하드웨어는 P 명령어를 수행하는 주 프로세서를 도와주는 코프로세서처럼 동작한다. 따라서 주 프로세서와 특수 하드웨어가 동시에 수행될 경우 특수 명령어에 포함된 작업량 만큼을 주 프로세서에 덜어주게 된다. 그러나 실제로는 주 프로세서의 작업과 특수 하드웨어의 작업이 서로의 결과를 상호 의존하는 경우가 많다. 특수 명령어가 수행되는 동안 주 프로세서에서 수행될 수 있는 P 명령어들이 충분한 경우에는 문제가 없다. 반대로 특수 명령어가 수행되는 동안 동시에 수행될 수 있는 P 명령어의 수가 특수 명령어의 레이턴시보다 적을 때는 '특수 명령어의 레이턴시 - 동시에 수행 가능한 P 명령의 개수' 만큼의 시간을 낭비하게 된다.

한 개의 특수 명령어에 대하여 동시에 수행 가능한 P 명령어를 찾는 일은 매우 쉽다. DM으로부터 주어진 특수 명령어보다 먼저 수행되어야 하는 것들과 나중에 수행되어야 하는 것들을 제외하면 동시에 수행 가능한 것들을 얻을 수 있다. 그림 9는 한 개의 특수 명령어와 동시에 수행 가능한 P 명령어들을 찾는 방법을 보여준다.



	1	2	3	4	5	6	7	8	9
1	0	0	0	1	1	0	1	1	1
2	0	0	0	1	1	0	1	1	1
3	0	0	0	0	0	1	0	1	1
4	0	0	0	0	0	0	1	0	1
5	0	0	0	0	0	0	1	1	1
6	0	0	0	0	0	0	0	1	1
7	0	0	0	0	0	0	0	0	1
8	0	0	0	0	0	0	0	0	1
9	0	0	0	0	0	0	0	0	0

	1	2	3	5	6	7	8	9
4	0	0	1	1	1	0	1	0

그림 9. 단일 특수 명령어
Fig. 9. Single S instruction.

특수 명령어에 대하여 동시에 수행 가능한 P 명령어들을 나타내는 행렬을 Independency Matrix(약칭 IM) 라고 부르며, 그 행렬의 (i, j)번째 원소가 1 이면 노드 i에 해당하는 특수 명령어는 노드 j에 해당하는

P 명령어와 동시에 수행 가능함을 나타낸다.

그림 9에서 4번 노드에 해당하는 특수 명령어의 레이턴시가 5 인 경우에는 특수 명령어를 수행하면서 동시에 3, 5, 6, 8번 노드를 P 명령으로 수행할 수 있다. 이 경우 전체 수행 시간은 9가 된다. 만약 이 특수 명령어의 레이턴시가 10 인 경우에는 3, 5, 6, 8번 노드를 특수 명령어와 동시에 수행시킬 수 있지만 특수 명령어가 끝날 때까지 남은 5사이클을 기다린 다음 7, 9번 노드를 수행해야 하므로 이 때의 전체 수행 시간은 14가 된다. 따라서 주어진 시그널 흐름 그래프의 전체 수행 시간이 최소가 되도록 특수 명령어의 레이턴시를 선택하는 방법을 정리하면 다음과 같다.

특수 명령어의 최소 레이턴시를 l_{min}, 최대 레이턴시를 l_{max} 라 하고 동시에 수행 가능한 P 명령어의 개수를 n, 결정되는 특수 명령어의 레이턴시를 l이라 하면,

1. $n \geq l_{max}$ 이면 $l = l_{max}$
2. $l_{min} \leq n < l_{max}$ 이면 $l = n + 1$
3. $n < l_{min}$ 이면 $l = l_{min}$

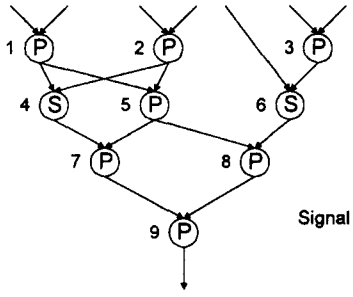
이 된다.

위의 용어 중에서 l_{max} 란 특수 명령어의 스케줄링 및 자원 할당을 했을 때 최소 면적을 가지는 것들 중에서 최소의 레이턴시값이다. l_{min}은 특수 명령어의 크리티컬 패스의 길이와 같다. 위의 방법을 사용하면 특수 명령어의 시간/면적 프로파일로부터 주어진 시그널 흐름 그래프의 전체 수행 시간을 최소화하는 가장 효율적인 구현 방법을 결정할 수 있다. 이를 위한 선행 작업은 물론 특수 명령어와 동시에 수행 가능한 P 명령어들의 개수를 결정하는 일이다.

3) 전체 수행 시간 예측

특수 명령어가 여러 개일 때 동시에 수행 가능한 P 명령어의 개수를 정확히 예측하는 일은 어렵다. 한 개의 P 명령어가 여러 개의 특수 명령어들과 동시에 수행 가능한 경우가 생길 수 있고, 어떤 특수 명령어가 P 명령어들 및 다른 특수 명령어와 동시에 수행이 가능할 때 이 특수 명령어를 P 명령어들과 동시에 수행하는 경우 사이클이 생길 수 있기 때문이다. 그림 10은 이러한 상황을 보여주고 있다.

그림 10에서 노드 5는 특수 명령어 4, 6과 동시에 수행 가능하다. 실제로는 노드 4와 동시에 수행되면 노드 6과 동시에 수행될 수 없고, 반대의 경우에도 그러하다. 따라서 위의 IM으로부터 노드 4와 동시에 수행될 수 있는 P 명령어가 3개이며, 노드 6과 동시에 수행될 수 있는 P 명령어가 4개라고 말하는 것은 틀린 예측이다.

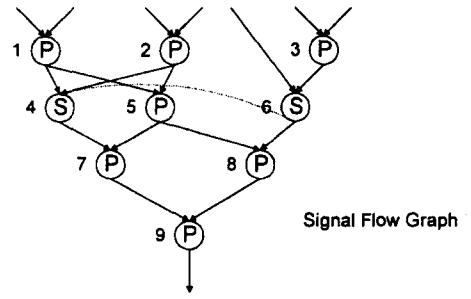


Independency Matrix

	1	2	3	5	7	8	9
4	0	0	1	1	0	1	0
6	1	1	0	1	1	0	0

그림 10. 특수 명령어가 여러 개일 때의 IM
Fig. 10. IM with multiple S instructions.

또한 노드 4와 노드 3, 5, 8을 동시에 수행할 경우 노드 6에 의한 사이클이 형성된다. 그 원인은 노드 4와 노드 6의 수행 순서가 정해져 있지 않고 서로 동시에 수행될 수 있기 때문이다. 따라서 여러 개의 특수 명령어가 존재할 경우 특수 명령어들의 순서를 정해서 이러한 경우가 생기지 않도록 해야 한다.



Independency Matrix

	1	2	3	5	7	8	9
4	0	0	0	1	0	1	0
6	1	1	0	1	0	0	0

Introduced Dependency

그림 11. 특수 명령어의 수행 순서 정하기
Fig. 11. Ordering S instructions execution.

그림 11은 특수 명령어를 나타내는 노드 6이 노드 4보다 먼저 수행되도록 순서를 정했을 때의 IM의 변화를 보여준다. 이 경우에는 노드 4와 노드 5, 8을 동시에 수행할 때 사이클이 생기지 않는다. 마찬가지로 노드 6과 노드 1, 2, 3을 동시에 수행해도 아무런 문제가 없다. 그러나 여전히 노드 5는 노드 4 또는 노드

6과 동시에 수행 가능하므로 이러한 노드를 어느 특수 명령어 노드와 동시에 수행시킬 것인가 하는 문제가 남아있다.

우리가 원하는 것은 각각의 특수 명령어에 대해서 동시에 수행 가능한 P 명령어의 개수이므로 실제로 어떠한 P 명령어들이 특수 명령어와 동시에 수행되는가 하는 문제는 중요하지 않다. 따라서 위의 경우처럼 한 개의 P 명령어가 n 개의 특수 명령어와 동시에 수행이 가능할 때는 한 개의 특수 명령어 당 1/n 개의 동시에 수행 가능한 P 명령어가 있는 것으로 계산하였다.

4) 전체 면적 예측

지금까지 설명한 바에 의하여 모든 특수 명령어들의 수행 순서가 정해지고 동시에 수행될 수 있는 P 명령어의 수가 예측되면 각각의 특수 명령어들에 대한 레이턴시를 결정할 수 있었다. 특수 명령어들의 레이턴시를 알면 시간/면적 프로파일로부터 그것을 구현할 수 있는 하드웨어의 면적을 알 수 있다. 이 때 명령어를 생성하는 과정에서 이미 생성된 명령어와 동일한 패턴을 가지는 명령어가 생성될 경우 이 명령어 패턴은 기존의 명령어와 하드웨어를 완전히 공유할 수 있으므로 동일한 명령어 패턴에 대한 면적은 한 번만 계산한다. 모든 특수 명령어를 지원하는 특수 하드웨어의 면적을 더하면 특수 명령어들을 지원하는 특수 하드웨어의 전체 면적을 예측할 수 있다.

Ⅳ. 특수 명령어 생성 예

시간 제약과 면적 제약이 주어질 때 목적 함수를 아래의 식과 같이 설정하였다.

$$\begin{aligned}\text{Objective} = &(\text{Initial\_Execution\_Time} \\ &- \text{Time\_Constraint}) \times \text{Area} \\ &+ \text{Area\_Constraint} \times \text{Time}\end{aligned}$$

위의 식은 면적과 시간의 교환 관계를 정의한다. 즉 주어진 제약 면적은 주어진 제약 시간을 만족하기 위해 줄여야 할 시간의 양과 동등한 가치임을 의미한다. 만일 면적의 가치를 더 중시할 경우 위의 목적 함수에서 연산 시간의 계수를 줄이거나 면적에 대한 계수를 증가시키면 되고, 시간의 가치를 더 중요시할 경우 반대로 시간에 대한 계수를 증가시키거나 면적에 대한 계수를 줄이면 된다.

특수 명령어 생성에 사용되는 예제로서 HYPER 의 SILAGE 예제 파일 중에 하나인 7th Order IIR 필터 프로그램을 사용하였다.

SILAGE로 기술된 프로그램을 HYPER툴에서 제공되는 파서를 이용하여 변환된 시그널 흐름 그래프를 사용하였다. HYPER툴에서 생성된 그래프는 ASIC 합성을 위하여 만들어지기 때문에 ASIP에서는 필요로 하지 않는 배열 입력 (arrayinput) 노드, 읽기 노드, 쓰기 노드, 딜레이 노드 등의 메모리 관련 유닛 기술을 포함한다. ASIP에서는 모든 데이타들은 데이타 메모리에 저장되어 있음을 가정하기 때문에 위에서 열거한 노드들을 제외한 연산 구조만을 분리하여 사용한다.

표 1. 7^{th} order IIR 필터의 명령어 생성 결과
Table 1. Result for 7^{th} order IIR Filter.

Time	Area	Instructions	Instances
24	0	None	None
23	4	I_1. $R_4 \leftarrow R_1+R_2+R_3$	$I_1(n_{82},n_{83})$
22	4	I_1. $R_4 \leftarrow R_1+R_2+R_3$	$I_1(n_{82},n_{83})$, $I_1(n_{52},n_{53})$
21	4	I_1. $R_4 \leftarrow R_1+R_2+R_3$	$I_1(n_{82},n_{83})$, $I_1(n_{52},n_{53})$, $I_1(n_{22},n_{23})$
20	12	I_1. $R_4 \leftarrow R_1+R_2+R_3$ I_2. $R_5 \leftarrow R_1+R_2+R_3-R_4$	$I_1(n_{82},n_{83})$, $I_1(n_{52},n_{53})$, $I_2(n_{22},n_{23},n_8)$
19	12	I_1. $R_4 \leftarrow R_1+R_2+R_3$ I_2. $R_5 \leftarrow R_1+R_2+R_3-R_4$ $R_6 \leftarrow R_3-R_4$	$I_1(n_{82},n_{83})$, $I_2(n_{52},n_{53},n_{38})$, $I_2(n_{22},n_{23},n_8)$
18	8	I_2. $R_5 \leftarrow R_1+R_2+R_3-R_4$ $R_6 \leftarrow R_3-R_4$	$I_2(n_{82},n_{83},n_{68})$, $I_2(n_{52},n_{53},n_{38})$, $I_2(n_{22},n_{23},n_8)$
17	16	I_2. $R_5 \leftarrow R_1+R_2+R_3-R_4$ $R_6 \leftarrow R_3-R_4$ I_3. $R_6 \leftarrow R_1+R_2+R_3-R_4-R_5$ $R_7 \leftarrow R_3-R_4-R_5$	$I_2(n_{82},n_{83},n_{68})$, $I_2(n_{52},n_{53},n_{38})$, $I_3(n_{22},n_{23},n_8,n_9)$
16	16	I_2. $R_5 \leftarrow R_1+R_2+R_3-R_4$ $R_6 \leftarrow R_3-R_4$ I_3. $R_6 \leftarrow R_1+R_2+R_3-R_4-R_5$ $R_7 \leftarrow R_3-R_4-R_5$	$I_2(n_{82},n_{83},n_{68})$, $I_3(n_{52},n_{53},n_{38},n_{39})$, $I_3(n_{22},n_{23},n_8,n_9)$
15	8	I_3. $R_6 \leftarrow R_1+R_2+R_3-R_4-R_5$ $R_7 \leftarrow R_3-R_4-R_5$	$I_3(n_{82},n_{83},n_{68},n_{69})$, $I_3(n_{52},n_{53},n_{38},n_{39})$, $I_3(n_{22},n_{23},n_8,n_9)$

표 1은 7^{th} Order IIR 필터에 대한 특수 명령어 생

성 결과를 나타낸 표이다. 필터의 시그널 흐름 그래프와 생성된 특수 명령어에 해당하는 부분은 그림 12에 나타나있다. 이 필터의 크리티컬 패스는 10이며, 제안한 알고리즘을 사용한 경우 단 한 개의 특수 명령어를 사용하여 15 클락에 구현될 수 있었다. 특수 명령어가 가지는 오퍼랜드 수의 제약에 의하면 두 개의 출력 오퍼랜드와 여섯 개의 입력 오퍼랜드만을 가질 수 있다. 이러한 제약에 의해 더 이상의 수행 시간을 단축하는 것은 불가능하다. 위의 필터와 같이 비슷한 종류의 연산 패턴이 자주 반복되는 구조를 가진 DSP 애플리케이션의 경우에 이 논문에서 제안한 알고리즘을 쓰면 적은 수의 명령어를 사용하여 최대의 효율을 얻을 수 있음을 알 수 있다.

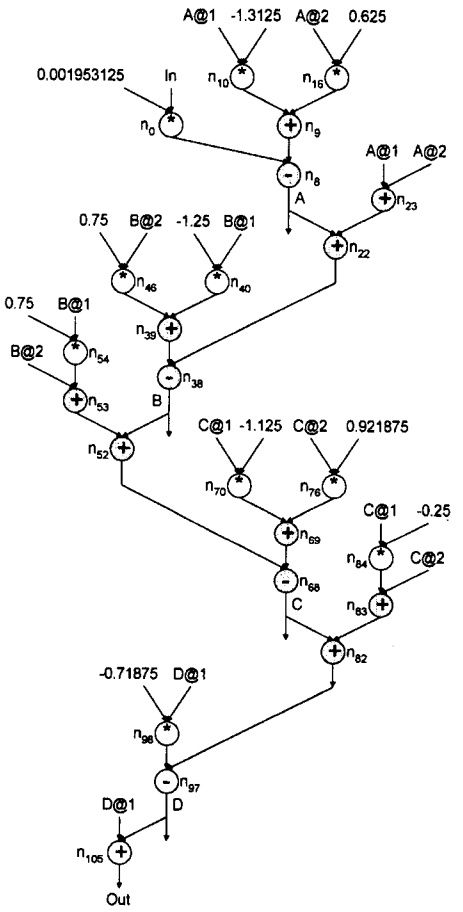


그림 12. 7th order IIR 필터
Fig. 12. 7th order IIR filter.

그림 12의 필터 프로그램은 한 샘플을 얻기 위한 연산의 흐름을 나타낸다. 10개의 샘플 데이타를 얻을 수 있는 타임 루프를 전체 시그널 흐름 그래프에 적용할 경우 그림 13과 같은 시간/면적간의 관계를 나타내는 그래프를 얻을 수 있다.

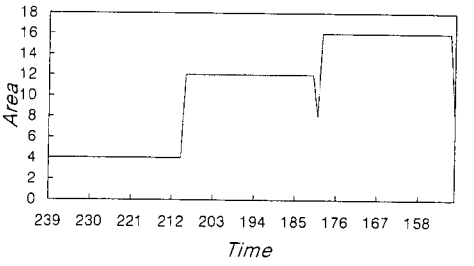


그림 13. 7th order IIR 필터의 시간/면적 관계
Fig. 13. Time/area tradeoff of 7th order IIR filter.

그림 13의 그래프에서 시간이 감소함에도 불구하고 면적이 일정하게 유지되는 부분은 한번 생성된 명령어 패턴이 계속 발견되어 매핑되는 경우이다. 이 경우 이미 생성된 명령어의 하드웨어를 공유할 수 있으므로 면적은 증가하지 않는다.

이렇게 동일한 패턴이 발견되는 경우 추가로 필요한 하드웨어의 면적이 증가되지 않고 전체 수행 시간이 줄어드는 이상적인 경우이므로 이러한 부분들이 우선적으로 선택되어 그래프의 수평인 부분을 만들게 된다. 그리고 그래프에서 수행 시간이 감소하는 동시에 하드웨어의 면적이 줄어드는 지점이 생기는 이유는 겹치는 패턴이 없는 명령어가 자라면서 이미 존재하는 명령어와 같은 패턴을 가질 경우이다. 이 경우에는 이미 존재하는 명령어의 하드웨어를 사용하므로 추가 면적이 필요하지 않고, 또한 전에 이 명령어만 가지는 패턴을 지원하는 하드웨어가 필요 없게 되므로 하드웨어의 전체 면적이 감소하게 된다.

V. 결론 및 추후 과제

이 논문에서는 P 명령어와 S 명령어를 지원하는 ASIP에서 S 명령어를 합성하는 방법을 제시하였다. 효과적인 특수 명령어를 생성하기 위하여 특수 명령어와 P 명령어가 함께 존재할 때 애플리케이션의 전체 수행 시간 및 필요한 하드웨어의 면적을 예측하는 방법을 제안하였으며, 간단한 예제를 통하여 제안된 방법이 어떻게 동작하는가를 알아보았다.

앞으로 연구할 분야는 제안된 알고리즘을 여러 가지 디자인에 적용하여 아직 발견되지 않은 문제점을 발견

하고 해결책을 연구하는 분야와 생성된 특수 명령어 패턴이 전체 시그널 흐름 그래프에서 얼마나 많이 발견되는지를 알아낼 수 있는 효율적인 알고리즘을 개발하는 분야, 그리고 P 명령어와 S 명령어 그리고 B 명령어를 동시에 지원하는 ASIP에서 명령어를 합성하는 연구이다. 또한 하드웨어의 수행 시간 예측시 체이닝 효과를 고려하는 방법[7] 및 하드웨어의 면적 예측시 생성되는 특수 하드웨어간의 유사성을 고려하는 방법[6] 등도 추가로 연구되어야 한다.

참 고 문 헌

[1] Giovanni De Micheli, Computer-Aided Hardware-Software Codesign, in *Magazine of IEEE Micro*, pp. 10-16, August 1994.

[2] Ing-Jer Huang and Alvin M. Despain, Generating Instruction Sets and Microarchitectures from Applications, in *Proc. of DAC*, pp. 391-396, 1994.

[3] Alauddin Alomary, Takeharu Nakata, Yoshimichi Honma, Masaharu Imai and Nobuyuki Hikichi, An ASIP Instruction Set Optimization Algorithm with Functional Module Sharing Constraint, in *Proc. of ICCAD*, pp. 526-532, 1993.

[4] Paul N. Hilfinger, A High-Level Language and Silicon Compiler, in *Proc. of IEEE Custom Integrated Circuits Conference*, pp. 213-216, 1985.

[5] S. Kirkpatrik, et al., Optimization by Simulated Annealing, *Science*, vol. 220, no. 4598, pp. 671-680, 1983.

[6] Ju Hwan Yi, Hoon Choi, In-Cheol Park, Chong-Min Kyung and Seung Ho Hwang, "Multiple Behavior Module Synthesis Based on Selective Groupings," *Design, Automation and Test in Europe* (DATE), 1998.

[7] Stefaan Note, Werner Ceurts, Francky Catthoor, Hugo De Man, Cathedral-III : Architecture-Driven High-level Synthesis for High Throughput DSP Applications, in *Proc. of DAC*, pp. 597-602, 1991.

[8] Jay K. Adams and Donald E. Thomas, The Design of Mixed Hardware/Software Systems, in *Proc. of DAC*, pp. 515-520, 1996.

[9] Frank Vahid and Daniel D. Gajski, Incremental Hardware Estimation During Hardware/Software Functional Partitioning in *Trans. on IEEE Very Large Scale Integrated Systems*, vol. 3. no. 3. September 1995.

저 자 소 개

金 泓 澈(正會員)

1975년 4월 19일생. 1992년 3월 ~ 1996년 2월 한국과학기술원 전자공학(B.S). 1996년 3월 ~ 1998년 2월 한국과학기술원 전자공학(M.S). 1998년 3월 ~ 현재 한국과학기술원 전자공학 Ph.D candidate. 관심분야는 Design Verification

黃 承 浩(正會員)

1956년 7월 6일생. 1975년 3월 ~ 1979년 2월 서울공대 전자공학(B.S). 1979년 3월 ~ 1981년 2월 한국과학기술원 전자공학(M.S). 1984년 8월 ~ 1989년 5월 Univ. of California, Berkeley 전자공학 Ph.D. 1985년 5월 ~ 1989년 5월 Univ. of California, Berkeley Postgraduate Researcher. 1989년 6월 ~ 1990년 9월 Schlumberger Technologies, Inc. Sr. S/W Engineer. 1990년 9월 ~ 현재 한국과학기술원 전기 및 전자공학과 교수. 관심분야는 VLSI CAD