빠른 하드웨어/소프트웨어 통합합성을 위한 데이타플로우 명세로부터의 하드웨어 합성

(Hardware Synthesis From Coarse-Grained Dataflow Specification For Fast HW/SW Cosynthesis)

정 현 욱 † 하 순 회 ††
(Hyunuk Jung) (Soonhoi Ha)

요 약 이 논문에서는 빠른 하드웨어/소프트웨어 통합합성을 위해 데이타플로우 그래프(DFG: Dataflow Graph)로부터 하드웨어를 자동으로 합성하는 내용을 다룬다. 이 데이타플로우 그래프에서 노드는 FIR(Finite Impulse Response) 필터나 DCT(Discrete Cosine Transform) 블록과 같이 크기가 어느 정도 되는 하드웨어 블록을 나타내며, 이 노드의 포트는 한번 수행할 때마다 하나 이상의 데이타 샘플을 주고 받을 수 있다. 즉, 멀티레이트 데이타 샘플(multi-rate data sample)을 교환한다. 이러한 특성들은 기존의 Behavioral Synthesis와 구별되는 점이며, 따라서 Behavioral Synthesis보다 어려운 문제가 된다. 본 논문에서 제안하는 설계 방법을 사용하면 알고리즘을 명세하는 데이타플로우 그래프는 하드웨어 리소스의 할당과 스케줄 정보에 따라 다양한 하드웨어 구조로 매핑될 수 있다. 따라서 하드웨어 설계시에 면적/성능 트레이드오프 관계를 손쉽게 관리할 수 있으며, 하드웨어를 자동으로 합성하는 기존의 방식보다 구현 가능한 하드웨어 설계 공간을 더욱 넓혀주는 효과를 거둘 수 있다.

키워드 : 하드웨어/소프웨어 통합설계, 시스템수준 설계, 데이타플로우 그래프, VHDL, 자동 하드웨어 합성

Abstract This paper concerns automatic hardware synthesis from data flow graph (DFG) specification for fast HW/SW cosynthesis. A node in DFG represents a coarse grain block such as FIR and DCT and a port in a block may consume multiple data samples per invocation, which distinguishes our approach from behavioral synthesis and complicates the problem. In the presented design methodology, a dataflow graph with specified algorithm can be mapped to various hardware structures according to the resource allocation and schedule information. This simplifies the management of the area/performance tradeoff in hardware design and widens the design space of hardware implementation of a dataflow graph compared with the previous approaches. Through experiments with some examples, the usefulness of the proposed technique is demonstrated.

Key words : HW/SW Codesign, System Level Design, Dataflow Graph(DFG), VHDL, Automatic Hardware Synthesis

1. 서 론

시스템온칩(SoC: System-on-Chip) 설계시에 설계복잡도가 증가하고 적기시장출하(time-to-market)에 대한 압력이 점점 늘어남에 따라 시스템 수준의 설계방법론(system level design methodology)이 더욱 주목을 받고 있다. 시스템 수준 설계에서는 시스템 수준의 명세가 체계적인 방법을 통해 최적의 아키텍처로 매핑되고, 빠른 설계 공간 탐색을 위해 그 매핑이 구현되기 전에 평가되어야 한다. 우리는 시스템 수준의 명세를 위해 여러 명세 모델들 중에서 데이타플로우 모델(Dataflow Model)을 사용한다. 이 모델은 읽기 쉽고, 정형적인 특성을 가지기 때문에 많은 상위 수준 설계 환경[1-3]에서 적용하고 있으며 특히 신호처리와 멀티미디어 응용에서 많이 사용되고 있다.

· 본 연구는 국가지정연구실 프로그램(번호 M1-0104-00-0015), 두뇌한국 21 프로젝트, SystemIC 2010 프로젝트에 의해 지원되었으며, 이 연구를 위해 연구장비를 지원하고 공간을 제공한 서울대학교 컴퓨터연구소에 감사 드립니다.

† 비 회 원 : 서울대학교 전기컴퓨터공학부
jung@iris.snu.ac.kr
†† 정 회 원 : 서울대학교 전기컴퓨터공학부 교수
sha@iris.snu.ac.kr

논문접수 : 2005년 1월 24일
심사완료 : 2005년 3월 16일

이 데이타플로우 모델은 Behavioral Synthesis에서 사용하는 CDFG(Control/Data Flow Graph)[4]와는 다르다. CDFG에서는 노드가 덧셈 또는 곱셈과 같은 간단한 연산을 나타내는 데 반해, 그림 1(a)의 데이타플로우 그래프 G(V, E)에서는 노드가 FIR 필터나 DCT 블록과 같이 상대적으로 큰 블록을 나타낸다. 그러므로 데이타플로우 그래프의 노드는 I/O 타이밍, 데이타 타입, 내부 상태(internal state)와 같은 복잡한 특성을 가진다. 지금부터 명확한 설명을 위해 두 용어, 하드웨어 리소스(hardware resource)와 하드웨어 콤포넌트(hardware component)를 다음과 같이 구분하고자 한다. 하드웨어 리소스는 한 노드가 라이브러리 형태로 하드웨어로 구현된 것을 의미하며, 하드웨어 콤포넌트는 FPGA나 ASIC과 같은 단위로서 하드웨어 리소스들이 결합되어 집합적으로 이루어진 것을 의미한다.

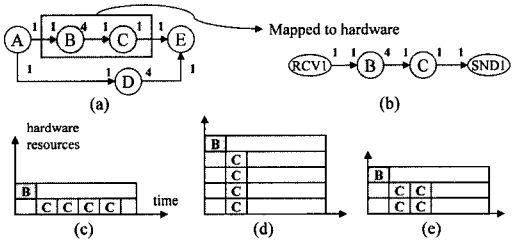


그림 1 : (a) 초기 데이타플로우 명세 (b) 하드웨어로 매핑된 분할된 서브그래프 (c) 멀티레이트 그래프의 순차수행 (d) 멀티레이트 그래프의 병렬수행 (e) 멀리레이트 그래프의 하이브리드 수행

한 노드가 수행이 되면 입력단에서 명시한 개수 만큼의 데이타 샘플을 소비하고 출력단에서 명시한 개수 만큼 데이타 샘플을 내보낸다. 본 논문에서는 SDF (Synchronous Dataflow) 모델[5]과 이 모델을 확장한 FRDF(Fractional Rate Dataflow) 모델[6]을 사용한다. 이 모델들의 특징은 포트에서 입출력되는 데이타 샘플을 개수가 미리 정해져 있어서 수행되는 동안 고정되어 있다는 점이다. 그리고 그 개수가 1보다 큰 경우에는 멀티레이트 그래프(multi-rate graph)라고 한다. 그림1에서 보는 바와 같이 데이타플로우 그래프에서 명시한 데이타 샘플 레이트(data sample rate)에 의하면 노드 B가 한번 수행된 후에 노드 C는 네번 수행되어야 한다. 앞으로 한 노드의 한번 수행을 그 노드의 인스턴스라고 부르기로 하겠다. 이렇게 데이타플로우 모델을 제한함으로서 시스템 성능을 예측하기 위한 중요한 메모리 제한성, 종료여부 등의 속성을 검증할 수 있다.

제안하는 설계 방법론에서는 데이타플로우 그래프의 한 노드에 해당하는 기능 블록 하나가 하드웨어/소프트웨어 분할의 단위가 된다. 본 논문의 설계 환경에서는 각 기능 블록들이 소프트웨어 구현을 위해서는 C 코드로, 하드웨어 구현을 위해서는 VHDL 코드로 기술되어 있다고 가정하고 있으며, 기존의 IP 블록들도 약간의 인터페이스 코드를 붙이면 데이타플로우 블록으로 사용할 수 있다. 하드웨어/소프트웨어 분할이 수행된 후에는 하드웨어 또는 소프트웨어 콤포넌트(Component)로 매핑되는 몇 개의 서브그래프로 나뉘게 된다. 그림 1(b)는 하드웨어 콤포넌트로 매핑되는 서브그래프의 예를 보여주고 있으며 서브그래프의 경계에 Rcv(receive), Snd (send) 등의 인터페이스 블록이 자동으로 붙게 된다.

그림 2는 데이타플로우 명세로부터 시작하는 하드웨어/소프트웨어 통합설계 절차를 단순화하여 보여주고 있다. 그림에서 보듯이 각 Processing Element(PE, 소프트웨어의 경우는 프로세서, 하드웨어의 경우는 하드웨어 IP또는 Component)에서 수행되는 각 노드의 성능은 노드-PE 데이타베이스에서 주어진다고 가정한다. 본 논문에서 제안하는 설계 환경에서는 초기 데이타플로우 명세가 자동으로 분할되고 스케줄되는데, 각 노드들의 성능과 가격에 기반해서 각 데이타플로우가 어느 PE로 매핑될지를 결정한다. 분할 기법에 관한 자세한 내용은 [7]에서 소개하고 있다. 선택한 아키텍처에 대해 분할과 스케줄링이 모두 수행되고 나면 소프트웨어와 하드웨어 코드가 자동으로 생성되고, 시스템 성능을 검증하기 위해 통합시뮬레이션이 수행된다. 만약에 시스템 성능이 원하는 만큼 만족스럽지 않으면, 아키텍처 선택 과정으로 되돌아가서 다른 PE 또는 아키텍처를 선택하여야 한다. 이와 같이 반복되는 설계 과정에 걸리는 시간을 줄이기 위해서는 데이타플로우 명세로부터 하드웨어와 소프트웨어 코드를 자동으로 생성하는 일이 매우 중요해진다. 또한 코드를 자동으로 생성하면 코딩하는 시간 뿐만 아니라 디버깅 시간까지도 절약할 수 있는 효과까지 거둘 수 있다.

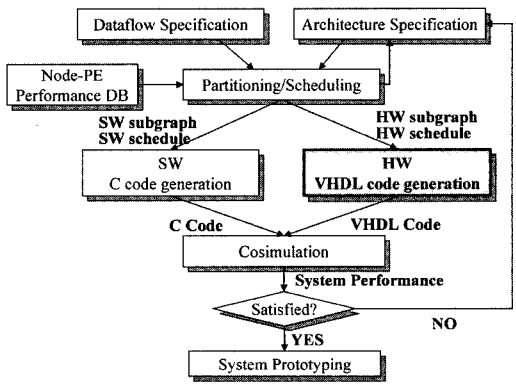


그림 2 제안하는 시스템 설계 과정

본 논문은 그림 2 중에서 특별히 어두운 색으로 표시한 하드웨어의 자동 합성 과정에 초점을 맞추고 있으며, 이는 하드웨어 설계와 검증과정의 시간을 줄이기 위해서 꼭 필요한 일이다. 데이타플로우 노드들이 수행되는 순서를 스케줄이라고 부르는데, 이 스케줄과 구현에 필요한 각 노드의 하드웨어 리소스의 개수가 이전의 분할(Partitioning) 단계에서 주어지므로, 하드웨어를 합성하는 문제는 라이브러리로부터 하드웨어 리소스들을 할당하고 각 블록들간의 인터페이스 코드(interface code)와 전체적으로 필요한 제어 코드(control code)들을 합성하는 일이 된다. 물론 합성된 하드웨어는 데이타플로우 모델의 규칙(dataflow semantic)을 지키도록 설계되어 원래 알고리즘 기술에서 의도한 것과 똑같이 수행되어야 한다.

멀티레이트 데이타플로우 그래프(multi-rate dataflow graph)는 멀티미디어 응용에서 자주 사용된다. 멀티레이트 그래프로부터 하드웨어 콤포넌트를 합성할 때는 알고리즘의 병렬성에 의해 조금 더 자유도가 큰 설계 공간을 고려할 수 있다. 데이타플로우 그래프로부터 자동으로 하드웨어를 합성하는 연구가 몇몇 이루어졌었지만, 이 병렬성을 고려하는 연구는 지금까지 없었다. 기존 연구를 크게 두가지 접근방식으로 나누면, 그림 1(c)와 같이 완전히 순차적으로 수행하는 경우[8,9]와 그림 1(d)와 같이 완전히 병렬적으로 수행하는 경우[10]가 있다. 본 논문에서는 이 두 가지 방식을 포함함과 동시에 그 중간 단계의 구현(그림 1(e))까지 지원함으로서 다양한 하드웨어 성능/면적 트레이드오프 관계를 고려할 수 있는 방법을 제안하려 한다.

본 논문이 기여하는 바를 명확히 열거하면 다음과 같다.

1. 하드웨어 합성의 자동화 : 데이타플로우 모델로 명세한 알고리즘으로부터 하드웨어를 자동으로 합성함으로써 설계 공간을 신속히 탐색할 수 있게 하였으며, 시간이 오래 걸리고 에러가 많이 발생하는, 하드웨어 블록의 연결 및 통합 작업을 자동화하였다.
2. 합성된 하드웨어 동작의 정확성 보장(correct by construction) : 스케줄 정보에 따라 하드웨어의 동작이 데이타플로우 모델의 규칙대로 이루어지도록 부가로직(glue logic)들과 콘트롤러(controller)를 합성하였다. 따라서 자동으로 합성된 하드웨어는 데이타플로우 알고리즘 명세에서 의도한 대로 정확히 동작할 수 있다.
3. 하드웨어 설계 공간의 다양화 : 스케줄링 과정과 하드웨어 합성 과정을 분리함으로써 주어진 하나의 데이타플로우 명세로부터 다양한 하드웨어 아키텍처를 구현할 수 있었다. 사용자는 단순히 스케줄과 리소스 공유 정보만을 변경하면 나머지 합성과정은 자동으로 이루어진다. 이러한 기법을 통해, 데이타플로우 기반으로 하드웨어를 자동생성하는 기존 접근 방식보다 하드웨어 구현의 설계 공간을 훨씬 넓힐 수 있었다.

본 논문의 구성은 다음과 같다. 다음 장에서는 본 연구에 동기를 부여한 예제와 함께 몇몇 관련 연구들을 소개하고, 3장에서 그 예제를 이용하여 제안하는 기법을 설명하고자 한다. 그리고, 4장에서는 하드웨어로 구현되는 각 블록의 타입에 관해 자세히 기술하고, 실험 결과들은 5장에 정리하였다. 마지막으로 6장에서 남은 과제들을 검토하고 결론을 맺고자 한다.

2. 기존 연구들과 동기 부여 예제

빠른 하드웨어/소프트웨어 통합 합성을 하기 위해서, 상위 수준 명세 또는 C/C++로 쓰여진 소프트웨어 명세로부터 하드웨어를 자동으로 합성하기 위한 많은 연구가 있었다. 하드웨어 기술 언어(HDL: Hardware Description Language)들로부터 Behavioral 수준 합성을 하는 연구[4,11,12]는 오랫동안 이루어져왔지만 제한적인 성공을 거두는데 그쳤고, C나 C++ 명세로부터 하드웨어 합성을 하는 연구[13-15]는 최근까지 지속되고 있다. 최근의 연구들은 포인터 분석과 병렬성 검출 및 이용등과 같이 C/C++ 명세의 제약점들을 극복하려는 시도에 초점이 모아지고 있다. 그러나 이러한 연구들은 하드웨어 블록을 자동으로 구현하는 것을 주목적으로 하는 것이고 본 논문에서 언급하는 내용은 주어진 하드웨어 라이브러리 블록을 이용하여 시스템 수준에서 통합하여 하드웨어를 구성하는 방법에 관한 것이라는 점에서 차이가 있다.

한편, 빠른 프로토타이핑을 위해 SDL(Specification and Description Language) 명세로부터 하드웨어를 합성하는 연구[16,17]가 있었다. 이 접근방식은 시스템 수준의 설계를 목적으로 한다는 점에서 우리의 접근 방식과 유사하지만 명세하는 모델과 응용의 범위가 다르다. SDL은 Asynchronously Communicating Process를 모델로 사용하며 주로 통신 시스템의 기술을 목표로 하는 반면에, 본 논문에서는 멀티미디어 응용의 기술을 목표로 하여 잘 정의된 모델인 SDF[5]를 사용한다.

그림 3은 2차원 DCT(discrete cosine transform) 알고리즘의 데이타플로우 그래프 표현이다. 이 그래프는 비교적 큰 64와 8의 데이타 샘플 레이트(data sample rate)로 표현된 멀티레이트 데이타플로우 그래프(multi-rate dataflow graph)이다. 이 알고리즘은 다양한 하드웨어 구조로 구현될 수 있는데, Ptolemy[10] 방식에서는 두개의 Transpose 블록 사이에 8개의 DCT1D 블록 리소스를 사용하여 완전히 병렬적으로 구현한다(그림 4(a)). 그림을 살펴보면 Transpose 블록이 64개의 데이

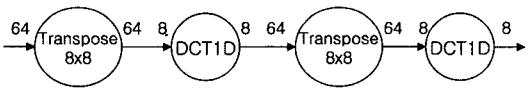


그림 3 2차원 DCT 알고리즘의 데이타플로우 명세

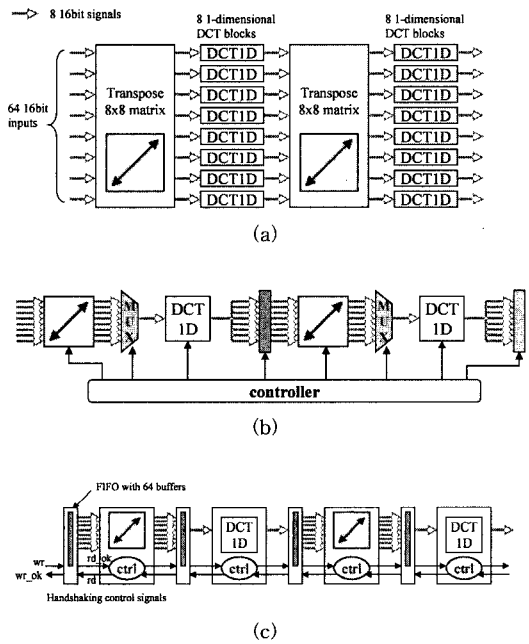


그림 4 (a) Ptolemy[10], (b) Meyr의 연구[9], (c) GRAPE[3]으로부터 생성된 하드웨어 구조

타 샘플을 출력한 후에 8개의 DCT1D 리소스들이 동시에 수행가능한 상태가 됨을 알 수 있다. 그리고 블록들 사이가 버퍼 없이 곧바로 연결되어 있다. 이러한 병렬 구현은 가장 빠른 수행 시간을 갖게 되지만 16개의 DCT1D 리소스를 사용해야 하므로, 우리 실험 결과에 따르면 하드웨어 면적이 약 243,000 게이트(gate)에 이르게 되어 면적면에서 손해가 가장 크다.

Meyr의 접근 방식[8,9,18]에서는 데이타플로우 그래프와 일대일 대응 관계로 하드웨어가 합성된다. 다시 말하면, 그래프의 한 노드가 그대로 하드웨어 블록 리소스 하나로 생성되는 것이다. 그림 4(b)에 나타난 바와 같이 8개의 데이타 샘플을 입력으로 받아 8개의 데이타 샘플을 출력하는 DCT1D 블록[1]이 하나밖에 없기 때문에 64개의 데이타 샘플을 받아 64개 데이타 샘플을 출력하려면 하나의 DCT1D 블록이 순차적으로 8번 수행되어야 한다. 만약에 DCT1D 블록의 내부가 파이프라인 형태로 구현되어 있다면, 8번의 순차적 수행이 파이프라인되어 겹쳐서 수행될 수는 있다. 이러한 순차적 구현에서는 64개의 데이타 샘플을 모아두기 위해 FIFO큐와 MUX 같은 부가 회로가 필요하지만 병렬 구현에서의 하드웨어 크기와 비교해서는 매우 미미한 정도에 불과하다. 이 접근 방식의 어려운 점은 정밀한 그래프 분석[18]을 통해서 많은 제어 신호(control signal)들의 타이밍을 미리 계산해야 한다는 점이다. 또 하나의 제약점은 이러한 타이밍 계산과 콘트롤러 합성을 위해 모든 하드웨어 블록들이 고정된 수행 시간만을 가질 수 있다는 점이다.

Ade가 연구한 GRAPE 시스템[3,19,20] 또한 하드웨어를 순차적인 방식으로 구현한다. Meyr의 방식과 다른 점이 있다면 각각의 하드웨어 블록들이 그 내부에 로컬 콘트롤러(local controller)를 가지고 있어 복잡하게 중앙 콘트롤러(central controller)를 구현할 필요가 없다는 점이다(그림 4(c)). GRAPE 시스템에서는 이웃한 블록들간에 핸드셰이킹 프로토콜(handshaking protocol)을 사용하여 각 블록이 언제 수행할 수 있는지를 판단한다. 이 예제에서는 Transpose 블록이 한번 수행된 후에 DCT1D 블록이 순차적으로 8번 수행되는데, 각 수행마다 핸드셰이킹에 필요한 제어 신호(control signal)를 반복적으로 주고받아야 한다. 따라서 GRAPE 시스템은 노드의 수행시간이 고정되지 않고 가변적인 경우에도 문제없이 잘 수행된다는 장점이 있으나 핸드셰이킹을 사용하는 노드간의 비동기적 통신(asynchronous communication)이 수행시간을 증가시키는 단점을 가지고 있다. 또한 데이타플로우 그래프로 표현한 노드들의 크기가 작은 경우에는 구현에 포함된 FIFO 버퍼들과 로컬 콘트롤러의 크기를 무시할 수 없다. 따라서 최적화된 ASIC 설계에서는 작은 블록들간에 위와 같은 분산 제어(distributed control)와 비동기적 통신을 사용하지 않는 경우가 대부분이다.

그림 4에서 (a)와 (b)는 병렬 구현과 순차 구현의 양극단을 보여주고 있다. 그러나 그 외에 다른 많은 구현 방식이 존재할 수 있다. 그림 5에 그 중 일부를 소개하였다.

그림 5(a)를 보면 그림 3의 두 DCT1D 노드가 하나의 DCT1D 리소스를 공유하고 있다. 이 DCT1D 리소스는 16번 수행되는데, 8번은 column DCT를 위해서, 8번은 row DCT 연산을 위해서 수행된다. 이 구현은 성능은 가장 느리지만 가장 적은 하드웨어 면적을 차지한다. 만약에 더 나은 성능이 필요하다면 column DCT와 row DCT 를 위해 리소스를 각각 따로 할당하고 그림 4(b)와 같이 파이프라인 형태로 동시에 수행할 수 있다. 그래도 시각 제약 조건을 맞추기 어려운 경우에는 그림

1) 원래 Meyr의 연구에서는 노드가 한번 수행될 때마다 출력단에는 한 개의 데이타 샘플만이 생성된다. 따라서 이 예제에서 8개의 데이타 샘플을 출력하기 위해서는 노드가 8번 수행되어야 한다. 그러나 효율적인 구현을 위해서 그림 3의 DCT1D 블록에 8개의 다른 출력단이 있다고 가정하였다.

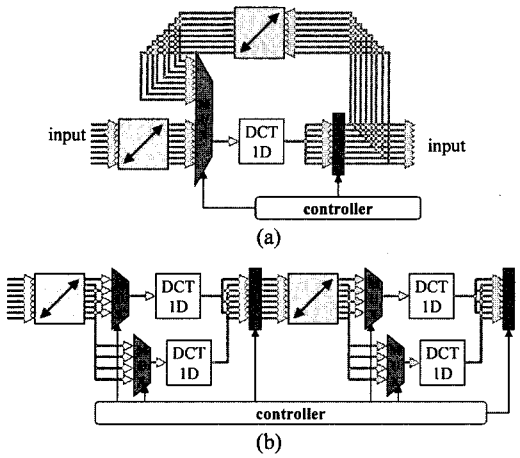


그림 5 일반적으로 사용되는 DCT의 하드웨어 구조 (a) 하나의 DCT1D 리소스 사용 (b) 네개의 DCT1D 리소스 사용

5(b)에서 각 DCT1D 노드에 대해 DCT1D 리소스를 두 개씩 할당한 것처럼 더 많은 리소스를 할당하면 된다.

본 논문에서는 기존 연구에 포함되었던 완전 병렬 구현과 완전 순차 구현을 포함함과 동시에 앞에서 설명한 리소스 공유와, 병렬과 순차의 중간 형태의 구현까지 모두 고려할 수 있는 기법을 제시하고자 한다. 이 기법을 사용함으로써 데이타플로우 명세로부터 자동 생성될 수 있는 하드웨어 설계 공간이 넓어질 수 있다. 제어회로의 설계에서는 블록의 수행시간이 가변적인 경우도 포함하기 위하여 [21]에서 제시한 제어기법을 그대로 적용하였다. [21]에서는 데이타플로우의 명세가 멀티레이트인 경우는 고려하지 않고 데이타플로우 그래프와 일대일 대응 관계로 하드웨어를 생성하였지만, 본 연구에서는 멀티레이트로 명세된 데이타플로우가 여러가지 방식으로 구현될 수 있도록 자동화하는 것을 주 내용으로 하고 있으며, 리소스를 공유하는 코드를 생성하는 내용까지 포함하고 있다. 아래의 표 1에서는 보다 명확한 이해를 돕기 위해서 지금까지 소개한 각 접근방식을 비교하여 정리하였다.

3. 제안하는 하드웨어 합성 기법

앞에서 언급한 내용이지만, 제안하는 방법에서는 분할 단계로부터 스케줄 정보를 얻는다. 그림 6은 스케줄 정보의 한 예를 보여주고 있는데, 그림 4(b)의 하드웨어 아키텍쳐를 얻기 위한 스케줄 정보이다. 이는 크게 두 부분으로 이루어져 있는데 먼저 리소스 할당 테이블(Resource Allocation Table)이 기술되고 그 뒤에 매핑/스케줄 정보(Mapping/Schedule Information)가 기술된다.

리소스 할당 테이블은 단순히 리소스 타입의 이름과 그 리소스의 개수를 열거한 것이다. 그림 6을 보면 Transpose와 DCT1D 리소스가 각각 두개씩 있음을 알 수 있다. 매핑/스케줄 정보는 노드의 각 인스턴스의 리소스 매핑과 수행 시간 및 타이밍 정보를 정의하고 있다. 매핑/스케줄 정보는 또한 루프(Loop)에 의해 계층적으로 묶일 수 있는데, 자세한 정보는 다음의 BNF를 통해 구조적으로 기술하였다.

<schudule information> ::=
<allocation table>
<mapping_schedule information>

<allocation table> ::= set of <allocation item>
<allocation item> ::= <resource type name> <number of resources>

<mapping_schedule information> ::= set of <mapping_schedule item>
<mapping_schedule item> ::= <instance mapping_schedule> | < loop mapping_schedule>
<instance mapping_schedule> ::= <node name> <mapped resource number> <start timing> <execution time>
<loop mapping_schedule> ::=
loop <loop count> <start timing> <loop period>
{ <mapping_schedule information> }

"Transpose_1 1 17 1"을 예로 들어 설명하면 다음과 같다. Transpose_1 노드의 첫번째 인스턴스는 두 Transpose 리소스 중에서 두번째 리소스(Resource

표 1 데이타플로우 명세로부터 하드웨어 합성하는 방식 비교

접근 방식	Ptolemy	Meyr의 방식	GRAPE	제안하는 방식
멀티레이트의 구현	병렬구현	순차구현	순차구현	병렬/순차/하이브리드
리소스 할당	복수할당 (multiple-resource allocation)	단일할당 (single-resource allocation)	단일할당 (single-resource allocation)	복수/단일/공유 할당 (multiple/single/shared allocation)
블록간 통신	동기적 레지스터 버퍼사용	동기적 레지스터 버퍼사용	비동기적 FIFO 버퍼사용	동기적 레지스터 버퍼사용
블록 수행 제어	중앙 제어 방식	중앙 제어 방식	분산 제어 방식	중앙 제어 방식
블록 수행 시간	고정	고정	가변	가변

Number = 1)를 사용하고 시작 타이밍(start timing)은 17 수행시간(execution time)은 1 사이클임을 의미한다.

만약 하드웨어 설계자가 그림 5(a)와 같이 하나의 DCT1D 리소스로 공유하는 구조로 하드웨어를 생성하고자 한다면 그림 6의 내용에서 다음과 같이 DCT1D 리소스의 개수를 1로 바꾸고 DCT1D_1 노드가 매핑된 리소스를 첫번째 리소스(0번 리소스)로 바꾸어 주기만 하면 된다.

DCT1D 1

DCT1D_1 **0** 0 2

이는 단 하나의 DCT1D 리소스가 할당되고, DCT1D_1 노드의 모든 여덟번의 인스턴스들이 0번 리소스로 매핑됨을 의미한다. 부가적으로 설명하면 DCT1D_1 노드는 루프(Loop)에 의해 여덟번 수행되므로 여덟개의 인스턴스를 가진다. 시작 타이밍과 수행시간에 사용되는 시간 단위는 클럭 사이클(clock cycle)이다.

시작 시각은 루프 안에서의 상대적인 시각을 의미하므로 루프 안의 맨 처음에 나오는 매핑/스케줄 정보의 시작 시각은 0으로 시작되어야 한다. Loop 8 1 2 { DCT1D_0 0 0 2 }와 같은 내용이 의미하는 바는 DCT1D 노드의 모든(8개의) 인스턴스들은 첫번째(0번) 리소스를 사용하고 그 리소스는 수행 시간이 2 사이클인데 8번 연속적으로 수행된다는 것이다. 루프(Loop) 명령의 인자들이 의미하는 바는 순서대로 반복 횟수, 시작 시각, 반복의 주기이므로, 이 루프는 1에 시작하여 2 사이클 간격으로 8번 반복 수행된다. 그림 5(b)와 같이 하드웨어를 바꾸어 생성하려면 스케줄 정보를 Loop 4 1 2 { DCT1D_0 0 0 2 , DCT1D_0 1 0 2 }와 같이 넣어 주면 된다. 이 스케줄 정보는 DCT1D_0 노드의 첫번째, 세번째, 다섯번째, 일곱번째 인스턴스는 0번 리소스에, 두번째, 네번째, 여섯번째, 여덟번째 리소스는 1번 리소스에 매핑되도록 만들어준다. 그리고 데이타플로우 그래프는 무한 루프처럼 반복 수행되므로 전체 스케줄 또한 하나의 루프로 간주한다.

```
# resource allocation table
Transpose      2
DCT1D          2
# resource mapping & schedule information
# (instance name, resource number, start, duration)
# loop ( loop count, start, loop period)
Transpose_0      0      0        1
Loop    8    1    2    {
DCT1D_0        0        0        2
}
Transpose_1      1      17       1
Loop    8      18     2{
DCT1D_1        1        0        2
}
```

그림 6 그림 4(b)의 아키텍처를 위한 스케줄 정보

제안하는 하드웨어 합성 기법은 주어진 스케줄 정보를 가지고 그에 상응하는 하드웨어 아키텍처를 합성해 내는 것이다. 그림 6의 스케줄 정보로부터 그림 4(b)의 하드웨어가 자동으로 합성되는데, 이 과정에 필요한 두 가지 핵심 요소는 콘트롤러 합성과 MUX, 레지스터와 같은 부가회로들의 합성이다.

먼저 콘트롤러를 만들기 위해서 두가지 자료구조가 필요한데, 하나는 리소스 제어를 위해 또 하나는 노드의 출력 버퍼를 위해 필요하다. 리소스 제어를 위해서는 리소스 입력단의 MUX제어 신호와 공유 리소스의 start 신호(start signal)가 필요한데 스케줄 정보를 분석하여 이 리소스에 매핑된 인스턴스 정보와 각 인스턴스들의 시작 및 종료 시간을 정리하여 만들어진다. 노드의 출력 버퍼 제어신호는 비교적 간단한데, 어느 리소스의 출력 데이타 샘플을 언제 버퍼에 담을지를 스케줄 정보를 분석하여 결정하면 된다. 이 내용들을 구체적으로 정리하면 다음과 같다.

<resource start information> ::= set of <mapped instance start time>

<resource input mux information> ::= set of <mapped instance input mux information>
<mapped instance input mux information>
::= <input buffer name> <mapped instance start time> <mapped instance end time>

<output buffer control information> ::= set of <output buffer assignment>
<output buffer assignment> ::= <resource output signal name> <buffer latch timing>

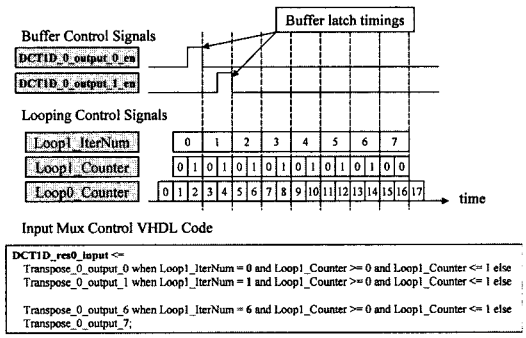
위의 내용에서 리소스 시작 정보(resource start information)는 각 리소스별로 매핑된 인스턴스들의 시작 시간들을 집합으로 가지고 있으며, 리소스 입력단의 MUX 제어 정보(resource input mux information)와 출력 버퍼 제어 정보(output buffer control information)는 각 포트별로 정보가 구성된다. MUX 제어 정보의 경우는 매핑된 각 인스턴스들의 입력 버퍼 이름과 시작 및 종료시간을 하나의 자료구조로 묶어서 이들의 집합으로 구성하며, 출력 버퍼의 제어 정보는 각 리소스 출력의 이름과 그 출력 시각을 하나의 자료구조를 묶어서 이들의 집합으로 구성한다.

자료구조가 다 만들어지고 나면 이 자료구조로부터 VHDL 코드가 생성된다. 맨 먼저 카운터(counter)를 사용해서 루프(Loop)의 제어 신호들이 생성된다. 그림 7(a)는 그림 4(b)의 DCT1D 에 대한 버퍼, 루프, MUX

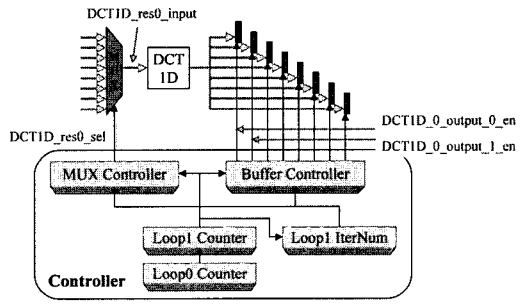
를 제어하는 신호들의 타이밍을 보여준다. Loop0는 가장 상위 수준에서의 제어를 담당하고 나머지 루프들은 순서대로 번호가 매겨진다. Loop_Counter 신호는 매 클럭마다 증가하며 Loop_IterNum 신호는 매 루프 주기의 마지막에 한번씩 증가한다. 그 후에 버퍼/MUX의 제어 신호와 각 리소스의 start 신호(start signal)가 생성된다. 이 신호들의 타이밍은 루프 제어 신호(Looping control signal)에 의해 결정된다. 예를 들면, DCT1D_0의 출력 버퍼는 Loop1_IterNum = 0이고 Loop1_Counter = 1일 때 데이타를 로드한다. MUX와 start 신호들도 같은 방식으로 제어되는데, MUX의 경우는 대상 리소스가 수행되는 동안 계속 입력 데이타의 값을 유지하고 있어야 하므로 카운터 값의 범위에 의해 제어된다. 본 설계 환경에서 생성하는 제어 신호들의 연결 구조는 그림 7(b)에 나타나 있으며, 제어 신호들의 VHDL 코드는 앞에서 만든 자료구조를 바탕으로 다음과 같이 생성된다. 스케줄에 루프가 존재하는 경우는 그림 7(a)처럼 루프의 Iteration조건까지 반영하여 시간 조건문이 보다 복잡하게 나타날 수 있으나 다음의 VHDL 코드에서는 이해를 돕기 위하여 간단하게 카운터(Counter)의 값만을 이용하여 조건문을 나타내었다.

```
--<start signal code>
<start signal name> <=
   '1' when Counter = <instance_1 start time> else
   '1' when Counter = <instance_2 start time> else
   ...
   '1' when Counter = <instance_N start time> else
   '0';

--<input mux code>
<resource input signal name> <=
   <input buffer_1 name> when Counter >= <start
         time_1> and Counter <= <end time_1> else
   <input buffer_2 name> when Counter >= <start
         time_2> and Counter <= <end time_2> else
         ...
   <input buffer_N name>;

--<output buffer code>
process(clk, rst)
begin
   if rst = '1' then
     <output buffer signal name> <= (others => '0');
   elsif rising_edge(clk) then
     if Counter = <latch timing_1> then
       <output buffer signal name> <= <resource
       output signal_1 name>;
     elsif Counter = <latch timing_2> then
       <output buffer signal name> <= <resource
       output signal_2 name>;
     ...
     elsif Counter = <latch timing_N> then
       <output buffer signal name> <= <resource
       output signal_N name>;
     end if;
   end if;
end process;
```

하드웨어 블록의 수행시간이 수행 중에 변하는 경우에는 최소 수행시간을 스케줄 정보에 기술하고 블록 내부로부터 생성되는 done 신호(done signal)를 콘트롤러가 검사하도록 한다. Done 신호가 참이 아닐 때는 콘트롤러가 카운터 값을 계속 같은 값으로 유지하면서 done 신호가 참이 될 때까지 기다리는 방식으로 동작한다. 이 콘트롤러 동작에 관한 자세한 내용은 [21]에 소개되어 있다.



(a)



(b)

그림 7 (a) 타이밍 다이어그램과 VHDL 코드 (b) 그림 4(b)의 DCT1D 블록을 위한 제어 신호의 하드웨어 구조

4. 블록 타입

그림 3에서 사용된 DCT1D와 Transpose 블록은 단순히 조합회로(combinational logic)로 구현되어 있다. 따라서 별다른 제어 신호를 필요로 하지 않는다. 그러나

블록들은 clock과 reset 신호를 필요로 하는 레지스터를 내부에 가지고 있는 순차회로(sequential logic)으로 구성될 수도 있다. 게다가 각 블록의 수행 시간은 외부 메모리 접근이나 조건부 연산과 같은 동적인 특성으로 인해 미리 고정되어 있지 않을 수도 있다. 본 논문에서 제안하는 설계 환경에서는 이러한 블록들의 성질을 반영하여 각 라이브러리 블록들의 타입을 4가지로 분류하였다. 덧셈기(adder)와 같은 경우는 조합회로, 누산기(accumulator)와 같이 한 cycle이면 수행이 완료되는 단사이클 순차회로(single-cycle sequential logic), 수행하는데 여러 사이클이 소요되는 다사이클 순차회로(multi-cycle sequential logic)로 나뉘며, 다사이클 순차회로는 수행시간이 고정되어 있는 경우와 수행시간이 가변적인 경우로 다시 나누었다. 다사이클 순차회로는 콘트롤러가 clock, reset, start 신호를 넣어 주어야 하며, 가변수행시간 다사이클 순차회로는 블록 내부에서 done 신호를 만들어서 수행이 종료되었음을 알려주어야 한다. 그림 8(a)는 네 가지 타입으로 구성된 예제를 보여주고 있다. 이 예제는 라이브러리 블록들과 생성된 콘트롤러를 통합하여 그림 8(b)와 같은 하드웨어로 만들어진다. 그림 8(c)에는 예상되는 제어 신호들의 타이밍을 그려 넣었다. 여기서는 A, B, C 각 블록들의 수행시간을 각각 30, 20, 80이고, D의 수행시간은 가변적이라고 가정하였다. 그러므로 D 블록의 done 신호는 수행중에 변할 수 있다.

모든 경우에 각 블록들은 strict execution을 가정한다. 다시 설명하면, 각 블록은 모든 입력 데이타 샘플들이 유효한 값을 가진 후에 수행 가능하며, 모든 출력 데이타 샘플들의 값이 유효해진 후에 블록의 수행을 끝마친 것으로 본다. 이것은 각 입출력단에서 명세한 데이타 샘플 개수만큼 입출력 버퍼가 필요함을 의미한다. 반면에 Meyr의 연구[9]에서는 멀티레이트 예제에서도 한번에 한 개의 데이타 샘플만을 입출력한다. 그림9의 간단한 예제를 살펴보자. 제안하는 방식에서는 Ramp 블록의 출력 데이타 샘플들을 저장하기 위해 Add4 블록의 입력 버퍼를 4개 할당한다. Meyr의 방식에서는 Add4블록이 4번 수행되므로 한 개의 버퍼만이 필요할 것이다. 사실 두 접근 방식은 Add4블록의 내부 구현에서도 차이를 보인다. Meyr의 방식에서는 Add4 블록이 내부 상태 레지스터를 가지는 순차회로로 만들어져 있다.

멀티레이트 예제에 대해 strict execution을 가정하는 것이 버퍼 면적에 있어 큰 오버헤드를 초래할 때는 Meyr의 방식과 같은 직렬적인 I/O (serial I/O) 구현이 더 좋다고 할 수 있다. 이러한 단점을 극복하기 위해서 FRDF(fractional rate dataflow) 모델[6]을 사용한다. 이 방식을 사용하면 똑같은 알고리즘을 기술하면서도

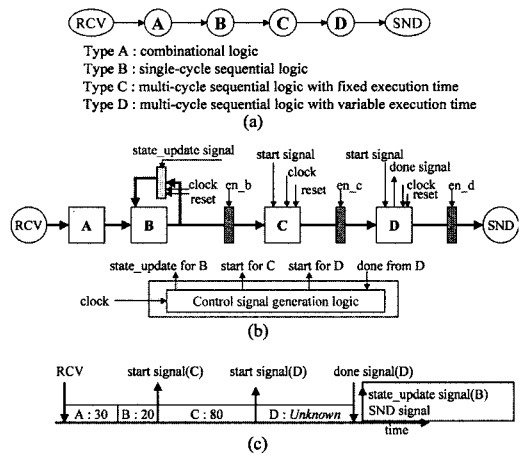


그림 8 (a) 다양한 타입을 블록으로 구성된 DFG의 예 (b) 합성된 하드웨어 구조 (c) 제어 신호들의 예상 타이밍

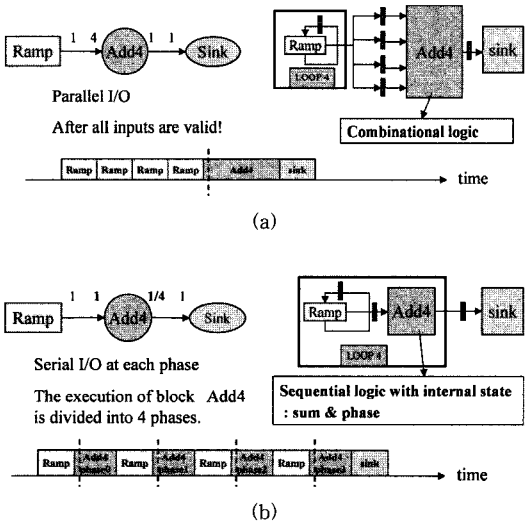


그림 9 보통의 데이타플로우 명세(a)와 FRDF 명세(b)의 하드웨어 구현

Meyr 방식의 Add4블록의 직렬적인 I/O(serial I/O)로 구현할 수 있다. 그림 9(b)에 같은 알고리즘을 기술한 FRDF 그래프와 그 하드웨어 구현을 묘사하였다. Add4 블록의 출력단에 있는 1/4이라는 분수꼴의 샘플 레이트(fractional sample rate)는 Add4블록이 4번 수행되는 동안 하나의 데이타 샘플만을 출력으로 내보낸다는 것을 의미한다. 다시 설명하면, 위 Add4블록은 매 수행마다 한 개의 입력을 받고 네번째 수행마다 한 개의 출력을 내보낸다.

5. 실 험

실험에서도 그림 3의 예제를 그대로 사용하였다. 합성을 위해서는 Synopsys Design Compiler를 사용하였으며 clock 주기는 20ns로 고정하였다. 실험 결과, DCT1D 블록 한 개의 수행시간(propagation delay)은 30.2ns로 2사이클을 차지하였다. 본 논문의 접근 방식의 효율성을 입증하기 위해서 그림 5(a)와 같은 구조를 가지는 하드웨어를 손으로 직접 작성하여 비교하였다. 이 구현은 한 개의 DCT1D리소스와 64×16비트 레지스터를 가지는 Transpose 메모리로 구성되어 있으며, 콘트롤러와 MUX, 카운터 등과 같은 부가회로가 포함되어 있다. 그림 4에서 소개한 다른 접근방식의 합성 결과들도 표 2에 비교하여 놓았다. GRAPE 시스템의 경우는 구할 수가 없어서 논문에 소개되어 있는 방식을 예상하여 하드웨어를 수동으로 구현하였다. 그러므로 실제 GRAPE 시스템의 결과와는 차이가 있을 수 있음을 미리 밝혀둔다. 표 2를 보면 자동으로 하드웨어를 생성했을 때, 버퍼와 콘트롤러가 생성 규칙에 따라 보수적으로 이루어지므로 손으로 생성한 것보다 조금 더 많은 하드웨어 면적을 차지함을 알 수 있다. 그림 10에는 다양한 하드웨어 구현의 성능/면적 트레이드오프 관계를 볼 수 있도록 그래프로 나타내었다.

그림 5(a)의 예제를 수동으로 구현한 것과 자동으로 구현 것을 표 2에서 비교하여 보면, 면적면에서는 21%, 성능면에서는 6% 정도의 오버헤드를 가지는 것을 확인할 수 있다. 이는 버퍼를 제어하는 코드를 생성할 때에, 리소스가 공유되는 경우와 루프를 돌면서 각기 다른 버퍼에 결과를 적는 경우, 블록간의 데이타 레이트가 2대3 등으로 서로 맞지 않는 경우를 일반적으로 해결하기 위하여, 각각의 버퍼마다 따로따로 조건문을 넣어 제어하기 때문이다. 그림 5(a)의 예에서는 DCT1D의 8개 출력

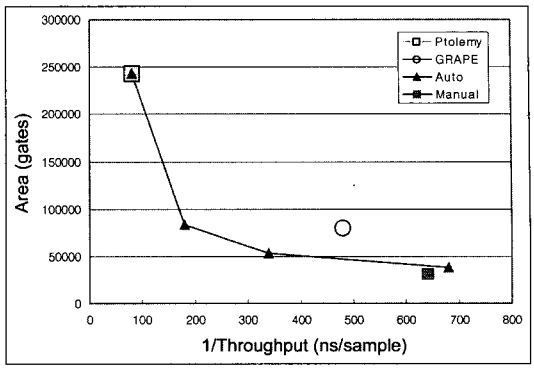


그림 10 표 2의 면적-수행시간 그래프

을 모두 하나의 제어 신호로 제어할 수 있음에도 불구하고 자동생성된 코드에서는 이 신호들이 따로따로 생성되어 표 2에서와 같이 면적상의 오버헤드를 가지게 되었다. 이러한 경우는 대부분의 예제들에서 발생할 수 있기 때문에, 다음 연구에서는 같은 조건들을 검사하여 제어코드를 보다 간결하게 생성할 계획이며 이를 통해 보다 최적화된 결과를 얻을 수 있을 것으로 기대하고 있다. 성능면에서는, 안전한 버퍼링과 다음 블록의 수행을 보장하기 위해서 자동 생성할 때 블록 사이에서 1 사이클의 마진을 두었기 때문에, 그림 5(a)의 구현에서 2 사이클의 차이가 나게 되었다.

또 다른 실험으로는 H.263 디코더 알고리즘의 일부를 사용하였다. 이 예제는 Dequantize, IDCT, MC(motion cmpensation) 블록을 포함하고 있다. 이 명세(그림 11)에서는 배열 타입과 사용자정의 타입을 사용하였으며 자동으로 하드웨어 신호 정의와 포트 매핑을 위한 코드, 콘트롤러 등의 모든 부가회로를 생성하였다. 그리고 MC 블록은 수행 중에 프레임 메모리 접근을 위한 메모리 접근 회로가 필요하기 때문에 블록 정의시에 메모리

표 2 실험 결과 : 2차원 DCT

Design type (number of DCT1D resource)	Area (gates)	Latency			Throughput (sample/ms)
		Clock period (ns)	Cycles	Total (ns)	
Manual (1) 그림 5(a)	**31,431**	20	**32**	640	1562.5
Auto (1) 그림 5(a)	**38,265**	20	**34**	680	1470.6
Auto (2) 그림 4(b)	53,252	20	34	680	2941.2
Auto (4) 그림 5(b)	83,130	20	18	360	5555.6
Ptolemy(16) 그림 4(a)	242,944	20	4	80	12500.0
GRAPE (2) 그림 4(c)	79,832	20	52	1040	2083.3

표 3 실험 결과 (H.263 디코더)

Design Type	Area (gates)	Latency (1 Macro Block)			Throughput (frame/ms) (CIF format)
		Clock period (ns)	Cycles	Total (us)	
Manual	124,762	20	1170	23.40	108
Auto	186,084	20	1188	23.76	106

를 접근하는 데이타 입출력 포트의 속성을 "SRAM"으로 하였다. 이 속성은 메모리 접근에 필요한 포트를 자동으로 생성하고 블록들 간의 프레임 전달은 메모리 주소인 포인터로만 이루어지게 만들어 준다. 또한 프레임 데이타와 mode, dx, dy등의 파라미터들은 FRDF로 명세하여 버퍼 크기를 줄이도록 구현하였다.

이 실험에서는 그림 11의 데이타플로우 명세로부터 자동 생성된 하드웨어와 같은 블록 라이브러리로 같은 기능을 수동으로 구현한 하드웨어를 비교하였다. 표 3을 통해서, 두 하드웨어 설계의 성능은 매우 비슷하고 자동으로 생성된 하드웨어가 면적이 조금 더 큰 것을 알 수 있다.

표 2와 3의 내용에서 본 바와 같이 제안하는 방법을 통해서 다양한 하드웨어 구현이 스케줄 정보 파일만 수정하면 간단히 자동으로 생성될 수 있다. 그리고 자동으로 생성된 하드웨어 구현이 질적으로 수동 설계에 매우 가까워졌다. 성능면에서는 미미한 수준의 오버헤드를 가지고 있으며, 면적면에서는 버퍼 콘트롤러의 오버헤드가 존재하지만 제어 조건 코드를 자동 생성 후에 손으로 손쉽게 최적화할 수 있으며, 이 부분의 자동화는 가까운 미래에 이루어질 것이다.

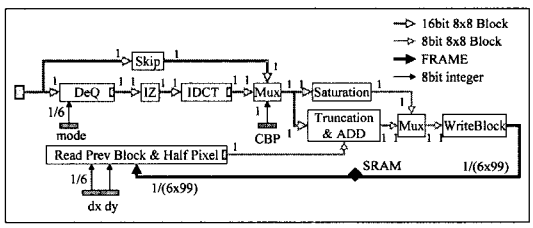


그림 11 H.263 디코더 알고리즘 일부의 데이타플로우 명세

6. 결 론

이 논문에서는 하나의 데이타플로우 명세로부터 다양한 하드웨어 구조를 합성하는 방법을 다루고 있다. 이 과정에서 공유 하드웨어 리소스와 루프(Loop)로 이루어진 스케줄을 고려하였으며, 성능과 면적면에서 실제 설계에 적용할 수 있을 만큼의 하드웨어를 자동으로 생성하였다. 이 논문에는 포함되지 않았지만 시스템 설계 과정에서 하드웨어 리소스의 자동 할당 및 스케줄이 주요한 이슈이며 그 과정은 분할 단계에서 이루어진다. 이 논문에서는 그 스케줄 정보를 바탕으로 실제 하드웨어를 구현하는 내용을 다루었다. 실험 예제로서는 2차원 DCT알고리즘과 H.263 디코더를 사용하여 제안하는 기법이 실제 VHDL코드를 생성함을 보여주었으며, 이 VHDL 코드들은 Synopsys 툴을 통해 검증하였다.

남은 과제로서 생성되는 하드웨어 면적을 좀 더 줄이기 위해 버퍼 콘트롤러를 최적화하고 버퍼들을 공유하는 방법이 필요하며, 실험 결과를 보면 자동 생성 오버헤드의 대부분이 이 버퍼 콘트롤러에 기인하고 있기 때문에 버퍼의 최적화 기법이 적용될 경우 수동 설계에 거의 근접할 수 있을 것으로 예상한다.

참 고 문 헌

[1] Synopsys Inc., 700 E. Middlefield Rd., Mountain View, CA 94043, USA. *COSSAP User's Manual : VHDL Code Generation.*

[2] J. T. Buck, S. Ha, E. A. Lee and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Int'l Journal of Computer simulation, special issues on "Simulation software development," vol.4, pp. 155-182*, April, 1994.

[3] M. Ade, R. Lauwereins, and J. A. Peperstraete. Hardware-software codesign with GRAPE. IEEE *Int'l Workshop on Rapid System Prototyping, pp. 40-47*, June, 1995.

[4] G. De Micheli. Synthesis and Optimization of Digital Circuits. *New York, McGraw-Hill, Inc.*, 1994.

[5] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proc. of the IEEE*, September, 1987.

[6] H. Oh and S. Ha. Fractional rate dataflow model and efficient code synthesis for multimedia applications. *ACM SIGPLAN Notice Vol. 37, pp. 12-17*, July 2002.

[7] H. Oh and S. Ha. Hardware-software cosynthesis of multi-mode multi-task embedded systems with real-time constraints. *International Workshop on Hardware/Software Codesign, pp. 133-138*, May 2002.

[8] P. Zepter and T. Groker and H. Meyr. Digital receiver design using VHDL generation from data

flow graphs. *Proceedings of the Design Automation Conference* 1995.

[9] J. Horstmannshoff and H. Meyr. Optimized system synthesis of complex RT level building blocks from multirate dataflow graphs. *International Symposium on System Synthesis, pp. 38-43*, Nov, 1999.

[10] M. C. Williamson and E. A. Lee. Synthesis of parallel hardware implementations from synchronous dataflow graph specifications. In *30th Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, California, USA, November 1996.

[11] Y.-L. Lin. Recent Devleopments in High-Level Synthesis. *ACM Transactions on Design Automation of Electronic Systems, Vol. 2, No. 1, pp 2-21*, Jan. 1997.

[12] D. W. Knapp. Behavioral Synthesis: Digital System Design Using the Synopsys Behavioral Compiler. *Prentice Hall*, 1996.

[13] L. Semeria, K. Sato, and G. De Micheli. Synthesis of Hardware Models in C With Pointers and Complex Data Structures. *IEEE Transactions on VLSI Systems Vol. 9, pp. 743-756*, Dec. 2001.

[14] L. Semeria and G. De Micheli. Resolution, Optimization, and Encoding of Pointer Variables for the Behavioral Synthesis from C. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems Vol. 20, pp. 213-233*, Feb. 2001.

[15] N. Vanspauwen, E. Barros, S. Cavalcante, and C. Valderrama. On the Importance, Problems and Solutions of Pointer Synthesis. *15th Symposium on Integrated Circuits and Systems Design, pp. 317-322*, Sept. 2002.

[16] F. Slomka, M. Dorfel, and R. Munzenberger. Generating Mixed Hardware/Software systems from SDL Specifications. *9th International Symposium on Hardware/Software Codesign, pp. 116-121*, April 2001.

[17] O. Bringmann, W. Rosenstiel, A. Muth, G. Farber, F. Slomka, and R. Hofmann. Mixed Abstraction Level Hardware Synthesis from SDL for Rapid Prototyping. *IEEE Int'l Workshop on Rapid System Prototyping, pp. 114-119*, June 1999.

[18] J. Horstmannshoff, T. Grotker, and H. Meyr. Mapping multi-rate dataflow to complex RT level hardware models. *In ASAP. IEEE*, 1997.

[19] J. Dalcolmo, R. Lauwereins, M. Ade. Code generation of data dominated DSP applications for FPGA targets. *IEEE Int'l Workshop on Rapid System Prototyping, pp. 162-167*, 1998.

[20] G. Bilsen, M.Engels, R. Lauwereins and J. Peperstraete. Cyclo-static dataflow. *IEEE Transactions on Signal Processing Vol. 44, pp.397-408*, Feb. 1996.

[21] H. Jung, K. Lee, and S. Ha. Efficient hardware controller synthesis for synchronous dataflow graph in system level design. *IEEE Transactions on VLSI Systems Vol. 10, pp. 423-428*, August 2002.

정 현 욱

1998년 2월 서울대학교 컴퓨터공학과 학사. 2000년 2월 서울대학교 컴퓨터공학과 석사. 2000년 3월~현재 서울대학교 전기컴퓨터공학부 박사과정. 관심분야는 하드웨어-소프트웨어 통합설계, 시스템 수준 설계, 하드웨어 자동 합성

하 순 회

1985년 2월 서울대학교 전자공학과 학사 1987년 2월 서울대학교 전자공학과 석사 1992년 미국 UCB 전기컴퓨터공학과 박사. 1993년~1994년 현대전자 근무. 1994년~현재 서울대학교 컴퓨터공학부 교수 관심분야는 하드웨어-소프트웨어 통합설계, 내장형 시스템을 위한 설계 방법론