C의 재귀 호출로부터 동적 구조를 활용한 VHDL로의 변환

홍 승 완† · 이 정 아††

요 약

하드웨어와 소프트웨어의 통합 설계 방법을 사용하면 다양한 신호처리 시스템을 설계 시간 및 비용에 있어서 효율적으로 구축 할 수 있다. 기존에 연구된 C로 구현된 다양한 신호 처리 시스템을 통합 설계 환경에서 효과적으로 활용하기 위하여 C로 구현된 알고리즘을 하드웨어 설계 언어(VHDL)로 변환할 필요성이 있다. C를 VHDL로 변환하는 경우 특히 동적 할당, 포인터, 재귀 호출 구문의 변환이 용이하지 않다 본 논문에서는, 현재까지 소프트웨어로 구현되어 왔던 재귀 호출문을 동적 구조를 활용하여 VHDL 구문으로 변환하는 방법론을 제시하고자 한다. 이를 통해 통합 설계의 하드웨어 소프트웨어 분할 시 유연성을 부여할 수 있고, 통합 설계의 궁극적인 목표인 시스템의 전체적인 성능 향상과 설계 시간 단축으로 우수한 복적 시스템을 구축할 수 있을 것으로 기대된다.

Translation utilizing Dynamic Structure from Recursive Procedure & Function in C to VHDL

Seung-Wan Hong† · Jeong-A Lee††

ABSTRACT

In recent years, as the complexity of signal processing systems increases, the needs for designers to mix up hardware-part and software-part grow more and more considering both performance and cost. There exist many algorithms in C for various Signal processing applications. We have to translate the algorithm in C to hardware description language(HDL), if portion of the algorithm needs to be implemented in hardware part of the system. For this translation, it's difficult to handle dynamic memory allocation, function calls, pointer manipulation. This research shows a design method for a hardware model about recursive calls which was classified into software part of the system previously for the translation from C to VHDL. The benefits of having recursive calls in hardware structure can be quite high since it provides flexibility in hardware/software partitioning in codesign system.

1. 서 론

점점 더 복잡화 다양화되어지는 시스템을 구성하는 각 component를 설계함에 있어 시스템의 성능과 비용 및 시간을 고려한 하드웨어와 소프트웨어를 혼합한 통합 설계(codesign) 환경에 대한 연구가 활발히 진행되고 있다.

최근에는 합성 가능한 상위 수준 합성 툴(high level synthesizer)에 관련된 기술의 발달은 개발자의 설계 추상화 수준을 높여 구조적 기술(structural description, Schematic이나 FPGA(Field Programmable Gate Array)에 의한 설계)뿐만 아니라, 행위적 기술(behavioral description, VHDL이나 Verilog같은 하드웨어 기술언어의

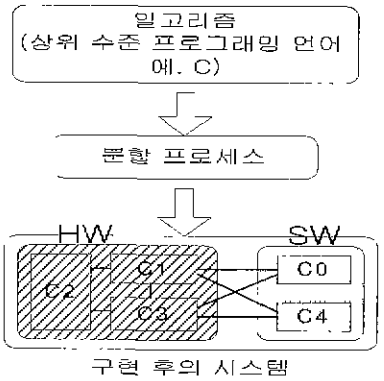
※ 이 논문은 1998년도 조선대학교 연구비의 지원을 받아 연구되었음
† 준 회 원 : 조선대학교 대학원 전자계산학과
†† 정 회 원 : 조선대학교 건지계산학과 교수
논문접수 2000년 5월 20일, 심사완료 : 2000년 10월 2일

행위적 기술에 의한 설계)을 통한 하드웨어의 설계도 가능하게 하였다. 이러한 발달은 하드웨어로만 이루어진 시스템의 설계를 용이하게 하였을 뿐만 아니라 하드웨어로 만들 대상의 범위에 유연성을 요구하는 통합 설계 환경이 실제적인 설계환경으로서 실득력을 가질 수 있는 요인이 되었다.

하드웨어 시스템 설계에서는 하드웨어 기술 언어(hardware description language, HDL)를 쓰고, 소프트웨어 시스템의 설계에서는 C와 같은 고급 언어(high level language, HLL)를 쓴다. 서로 다른 이질적인 부분들로 구성된 소프트웨어 부분과 하드웨어 부분이 혼합된 시스템(Codesign) 환경에서는 어떤 기술 방식을 써야 할지가 분명하지 않다. 시스템의 각 component에 어떤 프로그래밍 언어를 사용하여 구현하는가에 따라 합성에 영향을 미치게 된다. 왜냐면 실제로 쓰이는 합성 툴들이 모든 기술 언어를 지원하지는 않기 때문이다. 이를테면 Synopsys Design Compiler의 경우는 VHDL로 된 입력은 받아들이지만 C로 된 입력은 받아들이지 않는다.

보통 통합 설계 시스템에서 입력 알고리즘은 C언어로 구현된다. C 언어는 소프트웨어 설계뿐만 아니라, 하드웨어 설계 시, 칩 설계자들의 생각을 모두 형태화 할 수 있는 장점이 있다. 또한 C 언어는 빠르고 편리하며, 쉽게 에러를 수정할 수 있어, 하드웨어의 프로토타입을 설계하는데 매우 용이하다. C 언어로 구현된 알고리즘의 각 부분의 하드웨어 구현과, 소프트웨어 구현시의 성능-비용을 평가하여, 각기 다른 알고리즘으로 분할한다. 구현 과정에서 분할된 부분들에 대하여 각각 알맞은 합성 툴을 이용하여 합성하게 된다. 각각 합성 후 각 부분들 간의 인터페이스까지 합성하게 되면 통합 설계 시스템은 완성되게 된다. 이를 그림으로 나타내면 (그림 1)과 같다[1].

통합 설계 환경의 분할 단계에서 하드웨어 시스템 설계로 분할되는 경우에는 C 언어로 기술된 알고리즘을 하드웨어 기술 언어(hardware description language, HDL)로 변환해주어야 한다. 즉 C 언어를 VHDL이나 Verilog로 변환해주어야 한다. 이 과정에서 C 언어의 VHDL 언어로의 변환은 수작업에 의해 이루어진다. 수작업에 의한 변환은 시간과 비용이 많이 들고, 변환 시 에러가 발생할 수 있는 문제점이 있다. 따라서 C 언어를 합성 툴에 맞게 자동적으로 변환해 주는 기능이 필요하다.



(그림 1) 통합 설계 환경에서의 분할

VHDL에서 C 언어로의 변환은 compiled-code simulation 과 관련하여 많은 연구가 되어지 왔으나, 역으로 C를 VHDL로 변환하는 연구는 최근 들어 활발히 진행중이다. 특히 자동 변환 과정에서 C 언어의 동적 메모리 할당, 함수 호출, 포인터, 재귀 호출, 형 변환(type casting)에 대한 변환이 어렵다. 포인터에 관한 연구는 많이 진행되고 있지만, 재귀 호출에 관한 연구는 전무한 상태이다.

본 논문은 이러한 기술 언어의 자동 변환의 한 예인 C로부터 VHDL로 변환하는 방법 중 소프트웨어 설계 부분(C언어에 의한 구현)으로 분류되어지는 재귀 호출에 대한 하드웨어 설계 방법 알고리즘과 모델을 제시한다.

본 논문의 구성은 다음과 같다. 먼저 2장에서는 기존의 연구된 C로부터의 VHDL로의 변환에 대해 알아보고, 3장에서는 함수 호출부분의 재귀 호출에 대한 상세한 내용을 알아보고, 4장에서는 재귀 호출에 대한 하드웨어 설계 방법과 각 재귀 호출 형태에 대한 컨트롤 데이터 플로우에 대해 알아본다. 5장에서는 실험 환경에 대해 알아보고 6장에서는 결론과 앞으로의 연구 방향을 제시한다.

2. 국내외 연구 동향

최근 들어 통합설계를 연구하는 그룹들에 의하여 C로부터 VHDL로의 변환을 자동화한 시스템들이 등장하기 시작했다. 간단하게 살펴보면,

• C2VHDL Queensland 대학에서 통합설계 시스템 구축의 일환으로 개발된 툴로써, 대부분의 표준 C 구

문을 변환할 수 있으나, 포인터 구문은 변환하지 못하며, 계층적인 구조를 가지지 않는 단순한 형태의 입력만을 수행한다[2].

• Cosyma : Braunschweig 대학에서 개발한 통합설계 시스템으로, 소프트웨어로부터 시간이 오래 걸리는 부분을 하드웨어로 분할해 나가는 방식이다. 하드웨어 기술 언어로 VHDL 대신 HardwareC를 사용하였으며, 시스템의 성능에 초점을 맞추고 개발되었다[3]

• 서울대 연구 : C로부터 합성 가능한 VHDL로의 자동변환을 수행하며, 포인터 구문에 중점을 두고 개발되었다. 동적할당 및 재귀호출, 이중 포인터에 대한 것은 수행하지 못한다. 본 논문은 이 연구의 계속으로 재귀호출에 관하여 중점적으로 연구한다[4].

• SpC : Stanford 대학에서 개발한 툴로써, 그래프 이론을 바탕으로, 포인터에 중점을 두고 합성가능하고, 최적화된 VHDL 코드를 개발하는데 중점을 두었다. 그러나 이 툴도 동적 할당과 재귀 호출을 수행하지 못한다[5].

3. C구문에서의 재귀 호출

3.1 재귀 호출

일반적으로 C와 같은 대부분의 고급 언어는 재귀 호출을 수행한다. 재귀(recursion)라는 것은 함수가 직접 또는 간접적으로 자기 자신을 다시 호출하는 것을 말한다. 함수가 재귀적으로 자기 자신을 다시 호출할 때는 이전에 이미 갖고 있던 형식매개변수와 자동변수(==지역변수)들과는 전혀 무관하게 새로이 마련된 형식매개변수와 자동변수들을 갖게 된다. 즉, 함수가 자기 자신을 다시 호출하더라도 모든 형식 매개 변수들의 값은 값에 의한 호출(call by value) 원칙에 의거하여 스택 상에 호출 순서에 따라 층층이 유지되며 그 함수 내의 모든 자동 변수들도 역시 스택 상에 유지된다. 재귀 호출은 스택 상에서 실 매개변수를 새로운 형식매개변수에 복사하고 새로운 자동변수를 반복적으로 생성하면서 진행하기 때문에, 메모리를 상당히 많이 소모하며, 처리 속도 또한 상대적으로 무척 느리다.

프로그램을 설계시 재귀 호출을 사용하는 이유는 재귀적인 함수 정의가 인간의 자연적인 사고를 표현하기에 적합한 경우가 많기 때문이다. 2^n에 대한 재귀 호출을 사용하지 않는 PowerOF2()의 C 코드는 다음과 같다

```
int PowerOf2(int n) {
    int counter, result = 1 ;
    for (counter = n ; counter ; --counter)
            result= result * 2 ;
    return result ; }
```

재귀 함수를 갖지 않는 프로그램으로 코딩시, 프로그램 구문을 이해하기가 어려울 뿐만 아니라, 복잡한 재귀 호출 구문의 경우, 구문 표현을 위한 코딩 작업도 늘어나게 된다.

재귀 구문을 표현할 때 C 구문은 3개의 조건을 가진다[14].

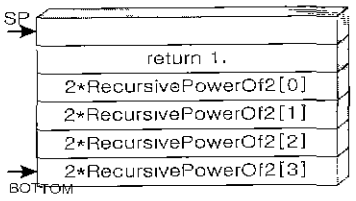
(1) 기본 조항(Base Part) : 재귀 조건을 종료하게 만드는 문장이 명시된다.
(2) 재귀 조항(Recursive Part) : 재귀 조건을 명시한다
(3) 한계 조항(Limitation Part) : (1), (2) 이외에는 정의에 포함되지 않음을 명시한다.

재귀 호출을 사용하여 구현한 2^n에 대한 RecursivePowerOf2()의 함수는 다음과 같다.

```
int RecursivePowerOf2(int n) {
  if (n==0)
          return 1 ;
  else
          return 2*RecursivePowerOf2(n-1) ; }
```

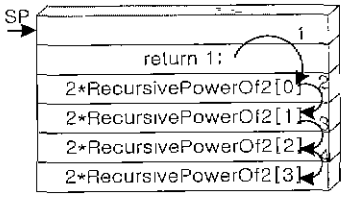
재귀 호출이 컴파일러 내부에서 수행되는 과정은 다음과 같다. 위의 함수를 가지고 RecursivePowerOf2(4)를 호출했을 때, 컴파일러는 스택을 사용하여, 일련의 코드를 생성한다. 첫 번째 호출에서 컴파일러는 임시의 결괴를 스택에 푸시한다. 스택에 푸시되는 정보는 임시 결과이므로, 불완전한 결과가 나중에 완성되어야 한다는 정보도 함께 들어간다.

이러한 과정은 계속되다가 첫 번째 if 문을 만족하는 경우에 마침내 종료한다. (그림 2)는 임시 결과가 스택에 푸시된 상태를 보여준다.



(그림 2) 푸시된 상태의 스택

결과를 산출하기 위해, 컴파일러가 자동으로 생성한 코드는 스택의 결과를 거슬러 올라가는 역추적(Back-tracking)이 수행된다. 역추적은 첫 번째 푸시된 데이터를 만나기 전까지 계속된다. (그림 3)에서 화살표에 부여된 번호는 역추적의 순서를 의미한다.

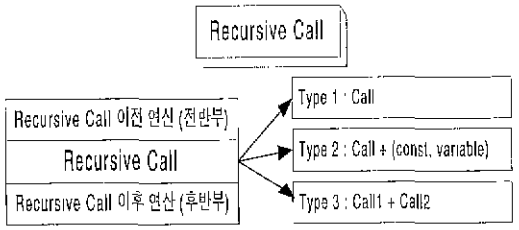


(그림 3) 스택의 데이터 역추적

3.2 재귀 호출 구문의 구조

C 언어에서 재귀 호출을 어떻게 구현하는가에 따라서, 재귀 호출 구문에 전반부, 재귀 호출부, 후반부가 나타날 수 있다. 즉, function 타입의 재귀 호출의 경우, 전반부와 재귀 호출부가 나타날 수 있고, Proccedure 타입의 재귀 호출의 경우에는 전반부, 재귀 호출부, 후반부가 나타날 수 있다. 프로그램에 따라서 재귀 호출 구문에서 전반부, 후반부는 나타나지 않을 수 있다

또한 재귀 호출부는, 재귀 함수만 나타나는 경우, 재귀 함수와 상수나 변수가 연산을 수행하는 형태. 재귀 함수가 여러개 나열된 상태로 나눌 수 있다 이를 그림으로 나타내면 (그림 4)와 같다



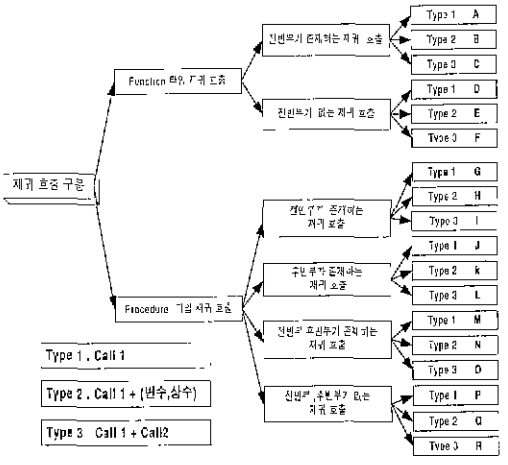
(그림 4) C에서의 재귀 호출 구문 구조

재귀 호출 구문 구조를 BNF 형태로 나타내면 <표 1>과 같다. 재귀 호출 구문의 전반부, 후반부에서는 변수 정의, 할당문, 그리고 연산문이 나타날 수 있다. 재귀 호출에서는 (그림 4)에서 설명한 3가지 형태의 재귀 호출 이 일어난다 재귀 호출부는 종결 조건과 재귀 호출로 이루어진다 재귀 호출에서는 재귀 함수만 단독으로 쓰이는 경우, 어떤 변수나 상수와 계산이 이루어지는 경우, 재귀 함수가 여러 개 나열되어 연산이 이루어지는 경우로 나눌 수 있다[13]

〈표 1〉 재귀 호출에 대한 BNF

• 재귀 호출 구문	::= <재귀 호출부>| <전반부> <재귀 호출부>| <재귀 호출부> <후반부>| <전반부><재귀 호출부><후반부>
• 전반부	::= <변수 정의>|<expression>|전반부
• 재귀 호출부	::= 종결 조건 <재귀 호출>
• 후반부	::= <변수 정의>|<expression>|후반부
• 변수 정의	::= <type>변수|변수 정의
• type	::= *int*|*real*
• expression	::= 할당문|연산문|expression
• 재귀 호출	::= 재귀 함수|<exp>operation 재귀 함수| 재귀 함수 operation 재귀 함수 | 재귀 호출
• operation	::= + | - | * | /
• exp	::= 상수 | 변수

(그림 4)의 재귀 호출 구문 구조와 <표 1>의 재귀 호출에 대한 BNF 구조에 의해 C 언어에서 나타날 수 있는 재귀 호출의 형태는 (그림 5)와 같다.



(그림 5) 재귀 호출의 여러 가지 형태

4. 재귀 호출문의 VHDL로의 변환 방법과 컨트롤 데이터 플로우

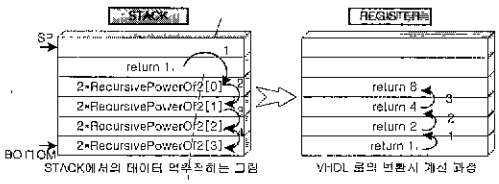
재귀 호출을 통합 설계 환경에서 소프트웨어(C)로 설계하면, 수행 속도가 느리기 때문에 통합 설계 내에서의 병목 현상을 일으킬 수 있으며, 설계된 제품의 전체적인 성능 저하를 일으킬 수 있다. 재귀 호출을 하드웨어로 설계하게 되면, 재귀 호출을 제어하기 위한 별도의 컨트롤러와 메모리 시스템(레지스터)이 통합 설계 시스템에 필요하게 된다. 즉, 설계시 비용과 시스템의 크기가 증가하지만, 통합 설계 시스템의 속도를 빠르게 하여 시스템 내에서의 병목 현상을 없앨 수 있다[6-8]

본 장에서는 3장에서 언급된 재귀 호출 구문을 하드웨어(VHDL)로 구현하기 위한 변환 방법과 그에 대한 데이터 플로우, 컨트롤 플로우를 제시하고, 또한 하드웨어(VHDL)로 구현 시 사용되는 component에 대해 알아보고, 각 재귀 호출에 대한 변환 예제를 다룬다.

4.1 변환 모티브(motive)

C로 구현된 재귀 호출의 경우, 재귀 호출이 몇 번 일어나는지 또 재귀 호출 수행 과정에서 중간 값을 정적으로 알 수 없다 그러나, 하드웨어로 구현하기 위해서는 재귀 호출 수행 과정에서 중간 값을 정적으로 알 수 있어야 한다 재귀 호출 구문의 중간 값은 고급 언어에서 스택에 보관된다. 스택은 초기 값이 나타날 때까지 반복 횟수를 알지 못하고, 계속 푸시를 수행한다. 본 논문에서는 이를 극복하기 위해 반대로 초기 값부터 반복해 나가는 방식을 택한다. 재귀 호출 구문의 경우, 어떠한 조건에 따라 일정하게 함수를 반복해서 부르게 된다. 따라서, 초기 값부터 조건에 만족할 때까지 Tail recursion을 수행하게 되면, C언어로 구현된 재귀 호출을 VHDL로 구현할 수가 있다. 즉, 본 논문에서는 값이 정해지지 않는 재귀 호출을 값이 정해진 재귀 호출로부터 시작하는 것이다.

3장에서 설명한 2^n의 관계를 예를 들면 다음의 (그림 6)과 같다. (그림 6)은 재귀 호출을 수행할 때 C에서의 데이터 흐름 상태와 VHDL로의 변환시 데이터의 흐름을 보여준다. C에서는 스택에 푸시할 때 정해지지 않는 크기 및 값들이 역추적에 의하여 중간 결과 값들을 생성해 낸다. 본 논문에서는 값이 정해지지 않는 중간 단계를 없애기 위해 제어기에서 조건에 만족할 때까지 반복을 수행하게 해주기 때문에 C에서와 같은 역추적이 필요 없게 되며, 수행 시간도 동시에 단축시킬 수 있다. VHDL로 구현할 때, 목적 시스템(Target System)의 크기를 줄이고, 동적인 값을 수행 할 수 있는 저장 장소로 레지스터를 사용한다. 레지스터를 사용하면, 재귀 호출이 일어나면서 중간 결과 값이 계속 레지스터에서 update 되기 때문에, 하드웨어의 공간을 줄일 수 있을 뿐 아니라, 재귀 호출을 동적 구조로 구현 할 수 있다.



(그림 6) 2^3에 대한 스택 수행 과정 및 변환 시 수행 과정

4.2 여러 재귀 호출에 대한 컨트롤 데이터 플로우

C로 구현된 재귀 호출 구문이 VHDL로 구현될 때, 각각의 C 구문에 맞는 하드웨어 component로 구문 매칭된다. 또한 하드웨어에서 메모리의 불필요한 소모를 줄이기 위해, 레지스터를 사용한다. C의 초기 값(종결 조건)은 VHDL에서 레지스터로, 조건 값(입력 조건, Parameter)은 컨트롤 플로우의 입력 조건 값(입력 신호)으로 매칭된다. 재귀 호출 구문 상에서 쓰이는 변수들도 레지스터에 1 : 1 매칭된다. 연산이 일어나는 부분 (+, −, ×, ÷)의 경우, 연산을 수행할 수 있는 연산 component로 매칭된다. 재귀 호출 구문 내의 전반부 연산의 경우, 이를 수행할 수 있는 특별한 component에 의해 구현되며, 후반부 연산의 경우, 재귀 호출 구문이 수행이 끝난 후의 값을 받아서 수행 할 수 있는 특별한 component를 사용한다. 전반부나 후반부에 들어갈 수 있는 component는 어떤 연산을 수행하는 연산 component(덧셈기, 곱셈기, 나눗셈기 등), 레지스터, 카운터 등이 들어갈 수 있다. 마지막으로 C구문에서의 조건 값은 VHDL 구문의 각 component를 제어할 수 있는 제어 component 내에서 조건에 맞게 카운트되는 특정 카운터와의 값 비교를 수행하게 된다. <표 2>는 각각의 C 구문이 하드웨어 component와의 매칭을 보여준다.

〈표 2〉 구문 매칭

C 구문 (software 설계)	VHDL 구문 (hardware 설계)
Variable	register, multiplexer
return value	register
연산 (+, −, ×, −)	연산 component (÷, −, ×, ÷)
초 기 값	register
전반부 연산	external component (전반부 연산 수행)
후반부 연산	external component (후반부 연산 수행)
Parameter(변 수)	중간 register
Paramenter(조건값)	제어 입력 신호, (counter, 조건 비교 component)

재귀 호출의 컨트롤-데이터 플로우의 형태에 따라 (그림 5)에서 분류하였던 재귀 호출의 형태를 다음과 같이 6가지로 재분류할 수 있다. 재귀 호출 구문 내에서 쓰이는 재귀 함수부가 어떤 형태인지, 또 재귀 호출 구문에 전반부나 후반부가 존재하는지에 따라서 구분하게 된다 본 논문에서 제시하는, 비슷한 컨트롤 데이터 플로우를 갖는 재귀 호출 구문을 분류하면 다음과 같다. 다음의 분류에서 나타나는 Type과 알파벳은 (그림 5)의 재귀 호출 형태에 따른 분류에서 표현한 알파벳이다

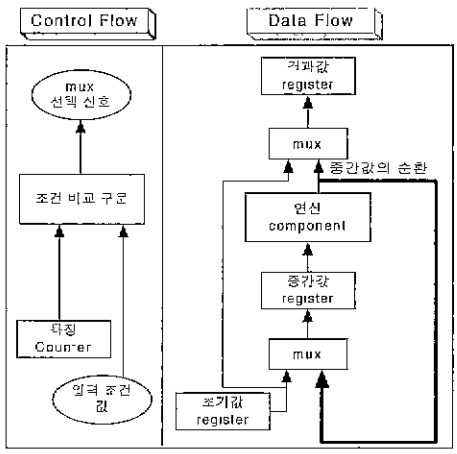
(1) 분류 Ⅰ : Type Ⅰ만 나타나는 경우 - D, P
(2) 분류 Ⅱ : Type Ⅱ만 나타나는 경우 - E, Q
(3) 분류 Ⅲ : Type Ⅲ만 나타나는 경우 - F, R
(4) 분류 Ⅳ : 전반부만 있는 Type ⅠⅡⅢ - A, B, C, G, H, I
(5) 분류 Ⅴ : 후반부만 있는 Type ⅠⅡⅢ - J, K, L
(6) 분류 Ⅵ : 전반부 후반부가 동시에 존재하는 Type ⅠⅡⅢ - M, N, O

위의 (1), (2), (3)은 재귀 호출부의 형태에 따른 분류이고, (4), (5), (6)은 재귀 호출 구문의 형태에 따른 분류이다.

위의 분류에 따른 각각의 재귀 호출 구문의 컨트롤 데이터 플로우를 알아본다[13].

4.2.1 분류 Ⅰ : Type Ⅰ만 나타나는 경우

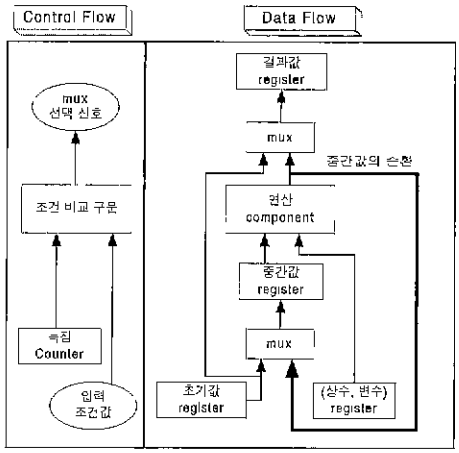
재귀 호출 구문에서 가장 기본이 되는 컨트롤 데이터 플로우이다. 이 기본 구조에서 파생되어 다른 분류의 재귀 호출 구문들은 추가되는 레지스터나 어떤 연산 component를 불여서 컨트롤 데이터 플로우를 만들게 된다. (그림 7)은 Type Ⅰ의 컨트롤 데이터 플로우를 보여준다. 재귀 호출이 하드웨어로 구현 될 때는 재귀 호출이 function 형태로 구현되었는지, Procedure 형태로 구현되었는지 상관하지 않는다 즉, 재귀 호출이 어떤 흐름을 갖는지에 따라서 하드웨이로 구현되는 형태가 달라지게 된다. Type Ⅰ의 데이터 플로우는 초기 값이 들어와서 어떠한 연산을 수행한 후, 다시 중간 값이 되돌아가서 동일한 연산을 수행하는 구조이다. 중간 결과 값은 순환하면서 레지스터에서 update 되고, 출력 레지스터에 의해 값들이 출력된다. 마지막 결과 값이 나오게 되면 컨트롤 플로우에 의해 결과값 레지스터는 초기 값을 출력하게 된다. 컨트롤 플로우는 입력 조건과 재귀 함수의 반복 크기(ex, rec(q-2)의 경우에는 2씩 증가하는 카운터)에 맞게 설계된 카운터와의 조건 비교에 의해 반복 횟수를 결정한다. 반복 횟수는 mux의 선택 신호를 출력함으로써 데이터 플로우의 값을 제어한다



(그림 7) 기본 Control-Data Flow

4.2.2 분류 Ⅱ : Type Ⅱ만 나타나는 경우

분류 Ⅰ의 기본구조에 어떠한 상수나 변수가 덧붙어져서 연산이 수행된다. (그림 8)은 Type Ⅱ에 대한 컨트롤 데이터 플로우이다 여기에서 쓰이는 어떤 변수나 상수는 레지스터를 기본 구조에 추가함으로써 수행할 수 있다 컨트롤 플로우는 재귀 호출의 반복 횟수만을 결정해주기 때문에 분류 Ⅰ과 동일하다. 변수나 상수가 추가 될 때마다 레지스터의 개수가 증가하게 된다 재귀 호출 시 변수가 계속 update 되는 경우에는, 그에 맞게 데이터 흐름을 맞춰 줘야 한다.



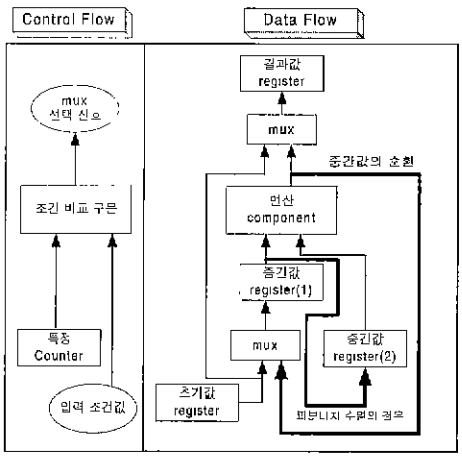
(그림 8) Type Ⅱ만 나타나는 경우

4.2.3 분류 Ⅲ : Type Ⅲ만 나타나는 경우

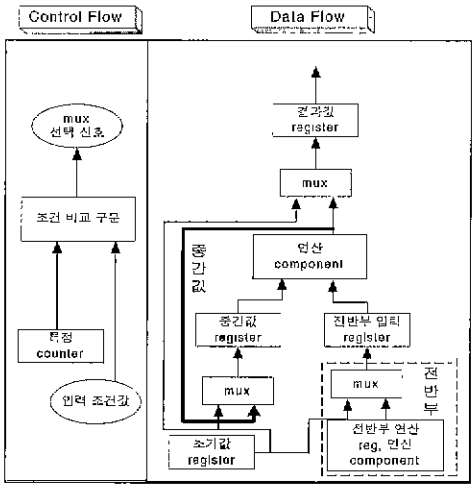
몇 개의 재귀 호출 구문이 나열되는 경우, 재귀 호출의 데이터 플로우는 재귀 함수의 개수에 따라서 중간 값이 순환하는 데이터 플로우의 개수가 정해진다. 즉, 2개의 재귀 함수가 나열된 경우에는 두 개의 중간 값 순환 데이터 플로우가 그려진다. (그림 9)는 재귀 호출 구문이 나열되는 경우의 컨트롤 데이터 플로우를 보여준다. 피보나치 수열의 경우, return fibo(n-1) + fibo(n-2)와 같이 프로그램 할 수 있다. 이 경우, 재귀 호출 구문이 두 개가 나타나므로 중간 값의 순환 데이터 플로우는 2개가 나타나게 된다. fibo(n-1)의 값이 쓰인 후에 fibo(n-2)의 값이 되기 때문에 (그림 9)와 같은 데이터 흐름이 일어나는 것이다. 즉, 재귀 호출 구문이 어떻게 쓰이느냐에 따라, 중간 값의 순환 화살표를 조건에 맞게 update되는 레지스터에 맞춰야 한다. 재귀 함수의 반복 크기에 따라서 데이터의 순환 흐름도 달라지게 된다.

4.2.4 분류 Ⅳ : 전반부만 있는 Type Ⅰ, Ⅱ, Ⅲ

기본 컨트롤 데이터 플로우에 전반부가 나타나는 형태이다. (그림 10)은 재귀 호출 Type Ⅰ, Ⅱ, Ⅲ에 전반부가 삽입된 형태의 재귀 호출 구문의 컨트롤 데이터 플로우이다. 전반부이기 때문에 기본 재귀 호출 구조의 앞부분에 붙게 된다 전반부는 변수 정의나 변수에 대한 할당문, 연산문이 들어갈 수 있다. 따라서, 컨트롤 데이터 플로우에는 전반부에 맞는 하드웨어 component를 삽입해 준다. 변수 정의의 경우, 데이터 플로우에 레지스터가 추가되고, 변수 할당문의 경우에는 값을 할당할 수 있는 레지스터와 값이 계속 변하게 되는 경우에는 mux나 어떠한 순환이 이루어 질 수 있도록 데이터 흐름을 연결해 준다. 연산문이 전반부에 존재하는 경우, 연산문에 맞는 레지스터와 연산 component, mux등이 필요하다 만약 전반부에서도 어떠한 중간 값이 되돌아오게 되면 데이터의 흐름이 재귀 호출 구문에서 순환하는 흐름과, 전반부에서 순환하는 흐름 두 가지가 나타날 수 있다. 컨트롤 플로우의 경우, 전반부와 재귀 호출부의 조건 값에 의한 mux 선택 신호를 내보낸다.



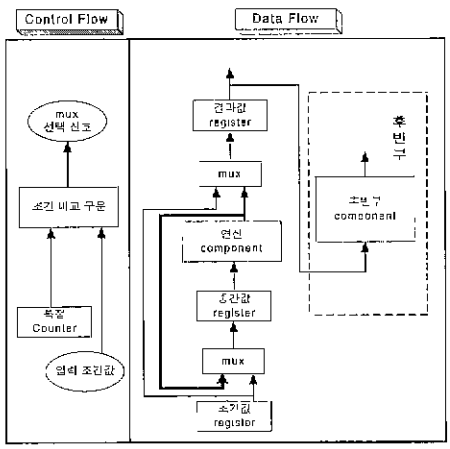
(그림 9) Type Ⅲ만 나타나는 경우



(그림 10) 전반부만 나타나는 Type Ⅰ,Ⅱ,Ⅲ

4.2.5 분류 Ⅴ : 후반부만 있는 Type Ⅰ, Ⅱ, Ⅲ

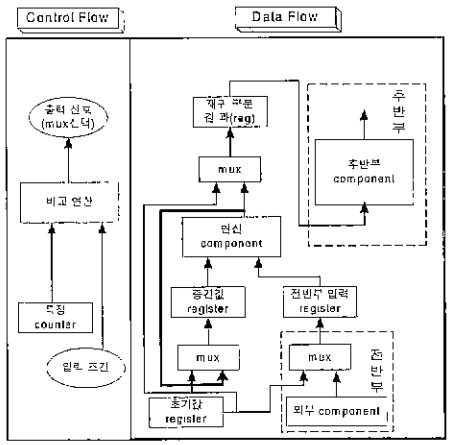
기본 컨트롤 데이터 플로우에 후반부가 삽입된 형태이다. (그림 11)은 재귀 호출 Type Ⅰ, Ⅱ, Ⅲ에 후반부가 들어간 형태의 재귀 호출 구문의 컨트롤 데이터 플로우이다. 후반부가 나타날 수 있는 형태는 Procedure 타입의 재귀 호출 밖에 없다. Function 타입의 재귀 호출의 경우, 재귀 호출부(return 문) 다음에 나오는 부분은 전부 무시되어 진다. Procedure 타입의 재귀 호출에서 후반부에 나올 수 있는 형태는 전반부에서 나타날 수 있는 형태와 동일하다. 후반부가 나타나는 재귀 호출에서 기본 데이터 플로우의 결과 값과 어떤 연산을 수행하는 경우, 기본 재귀 호출의 데이터 플로우의 다음 부분에 위치한다. 컨트롤 플로우의 경우, 전반부가 있는 재귀 호출과 마찬가지로, 제어 구조에는 별다른 영향이 없다. 다만, 만약 어떤 연산이 이루어지는 연산문이 후반부에 존재하는 경우, 연산 횟수를 제어할 수 있는 특별한 카운터가 추가된다. 컨트롤 플로우는 재귀 호출부와 후반부를 제어할 수 있는 mux의 선택 신호를 출력하게 된다.



(그림 11) 후반부가 있는 Type Ⅰ, Ⅱ, Ⅲ

4.2.6 분류 Ⅵ : 전반부, 후반부가 동시에 존재하는 Type Ⅰ, Ⅱ, Ⅲ

(그림 12)는 기본 컨트롤 데이터 플로우에 분류 Ⅳ와 분류 Ⅴ에서 설명한 전반부, 후반부의 형태가 같이 존재하는 컨트롤 데이터 플로우이다. 이 형태도 분류 Ⅴ와 마찬가지로 후반부가 존재하므로 Function 타입의 재귀 호출은 이에 해당되지 않는다. 분류 Ⅵ의 형태에서 중간 값의 순환은 전반부, 재귀 호출부, 후반부에서 모두 나타날 수 있다. 컨트롤 플로우는 전반부, 재귀 호출부, 후반부를 모두 제어할 수 있는 특정 카운터를 사용하여, 데이터 흐름을 제어 할 수 있도록 mux 선택 신호를 출력한다.



(그림 12) 전반부, 후반부가 존재하는 Type Ⅰ, Ⅱ, Ⅲ

4.3 각 재귀 호출에 대한 변환 예제

위에서 분류한 여러 재귀 호출 구문 형태에 대한 C 구문 예제와, 그에 따른 컨트롤 데이터 플로우에 대해 알아본다. 또한 재귀 호출 구문의 다른 VHDL 구현 방법인 재귀 호출 구문을 재귀 호출 횟수만큼 flat하게 펼쳐서 구현했을 때의 컨트롤 데이터 플로우에 대해서도 알아본다.

4.3.1 예제 Ⅰ : Type Ⅰ만 나타나는 경우

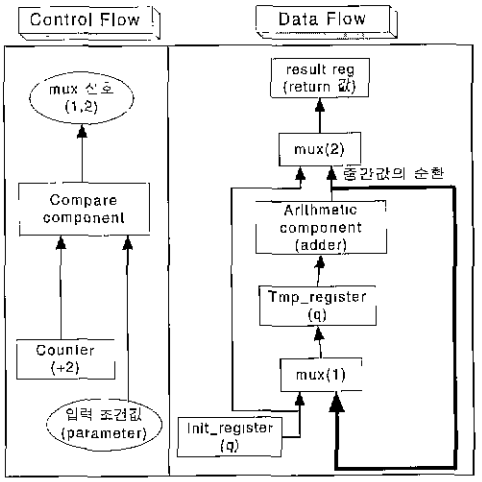
<표 3>의 C 구문은 단순히 재귀 함수만 갖는 C 구문이다. 이 구문은 모든 재귀 호출의 기본이 되는 구문이다. 이 구문이 응용되어 분류된 재귀 호출이 파생되어 나간다.

〈표 3〉 기본 재귀 호출

```
#include<stdio.h>
int rec(int q)
void main()
{
        int i,      int x,      i = 6,
        x = rec(i);             // 재귀 함수 호출
        printf("recursive call value is %d\n",x);
}
```

```
rec(int q)
{
  if ( q==0) return 1,              // 재귀 호출 부분 시작
  else       return rec(q-2)        // 재귀 호출 종료
}
```

(그림 13)는 <표 3>의 컨트롤 데이터 플로우를 보여준다. <표 3>의 재귀 호출 파라미터 q는 (그림 13)에서 초기 레지스터와 중간 레지스터에 매칭된다. 리턴 값은 마지막 결과 레지스터에 매칭된다. 연산 component는 <표 3>의 재귀 함수에서 (q-2)에 매칭되는 adder가 된다. (q-2)에서 빼기 연산이 수행되지만 본 논문에서는 Tail recursion을 수행하므로 반대되는 연산인 덧셈 연산 component가 들어가게 된다. 재귀 함수 내에서 연산은 반대되는 연산이 수행되고, 재귀 함수 외부에서 일어나는 연산의 경우는 올바른 연산 component가 들어가게 된다 컨트롤 플로우에서는 초기 값과, 중간 값을 선택할 수 있는 mux 선택 신호 (1)과, C 구문의 파라미터 값에 맞는 결과 값을 선택할 수 있도록 하는 mux 선택 신호 (2)를 출력해 준다.



(그림 13) 기본 예제의 컨트롤 데이터 플로우

<표 4>는 기본 재귀 호출에 대한 VHDL Netlist와 제어 구조에 대한 VHDL 구문이다 (그림 13)의 데이터 플로우와 컨트롤 플로우가 <표 4>과 같이 구문으로 만들어진다. 이후의 재귀 호출 예제도 표 4와 같은 형태로 VHDL 구문이 만들어진다[10-11].

<표 4> 기본 재귀 호출에 대한 VHDL 구문 Netlist, 제어 구조

```
- 기본 재귀 호출에 대한 component 구조

reg1  reg port map(input => init_value, output =>
tmp_init_value, clock => clock, rst => rst);
sel_mux1 : mux port map(a => tmp_init_value, b =>
tmp_last_out, mux_out => tmp_mux_out, sig =>
mux_sel1);
reg2 : reg port map(input => tmp_mux_out, output =>
tmp_reg_out, clock => clock, rst => rst);
inc : adder port map(a => tmp_reg_out, b =>
tmp_add_out),
sel_mux2 : mux port map(a => tmp_init_value, b =>
tmp_add_out, mux_out => tmp_last_out, sig =>
mux_sel2);
reg3  reg port map(input => tmp_last_out, output =>
tmp_last_result, clock => clock, rst => rst);

- 기본 재귀 호출에 대한 제어 구조

range_compare : counter port map( rst => rst, clock =>
clock, q => tmp_q);
process(clock, rst, condition_value)
begin
        if rst = '1' then
            tmp_sel1 <= '0';  tmp_sel2 <= '0';
        else
            if clock'event and clock = '0' then
                if condition_value = "00000000"
                then
                        tmp_sel1 <= '0';
                        tmp_sel2 <= '0';
                elsif condition_value < tmp_q then
                        tmp_sel1 <= '0';
                        tmp_sel2 <= '0';
                elsif condition_value > tmp_q then
                        tmp_sel1 <= '1';
                        tmp_sel2 <= '1';
                end if;          end if;       end if;
end process;
top  top_body port map(clock => clock, rst => rst,
init_value => init_value, mux_sel1 => tmp_sel1, mux_sel2
=> tmp_sel2, out_value => tmp_out_value),
```

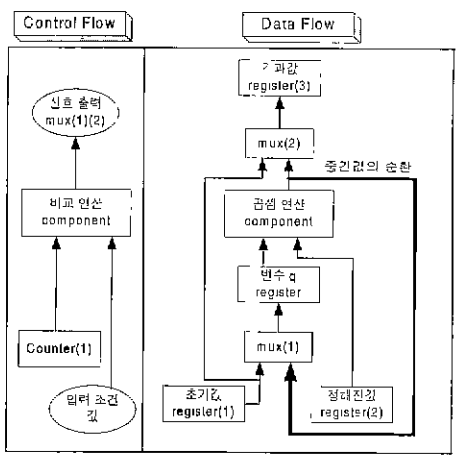
4.3.2 예제 II : Tape II만 나타나는 경우

하나의 재귀 함수와 상수나 변수가 연산을 수행하는 경우의 예제이다. <표 5>의 예제는 2의 멱수승에 대한 재귀 호출 C 구문을 보여준다 정해진 상수 2와 재귀 함수(rec(q-1))이 연산을 수행하는 구문이다

〈표 5〉 2^n에 대한 C구문

```
#include <stdio.h>
int rec(int q);

void main()
{
        int i;      int x, i = 5;
        x = rec(i);                  // 재귀 함수 호출
        printf("recursive call value is %d\n" x),
}

rec(int q)
{
  if ( q==0)
        return 1;                    //초기값
  else
        return 2*rec(q-1),           // const - 재귀 함수
 }
```

2^n에 대한 컨트롤 데이터 플로우는 (그림 14)와 같다. <표 4>의 C 구문에 대한 데이터 플로우의 component는 초기 값을 위한 레지스터 (1)과 변수 q와 중간 값을 동시에 사용하기 위한 레지스터 (2), 그리고, 상수 {2}를 저장하는 레지스터 (3), 출력 값을 표시하기 위한 레지스터 (4)가 쓰인다. 2^n을 계산하기 위한 곱하기를 수행하는 component, 초기 값과, 중간 값을 선택하는 mux(1), 출력값을 선택하는 mux(2)가 필요하다. 컨트롤 플로우에서는 몇 번을 수행하는지를 계산하는 카운터와, 그 카운터를 비교하는 조건 비교 구분이 필요하다 컨트롤 플로우에서는 mux(1),(2)의 선택신호를 보낸다



(그림 14) 2^n에 대한 컨트롤 데이터 플로우

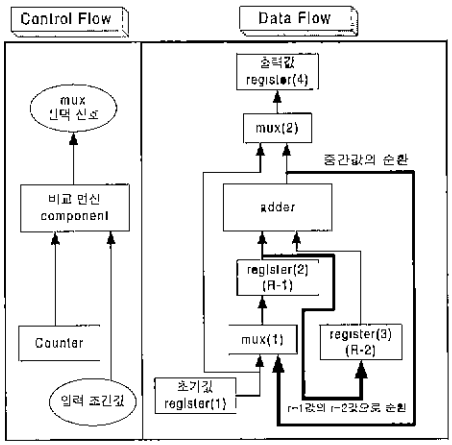
4.3.3. 예제 Ⅲ : Type Ⅲ만 나타나는 경우

여러 재귀 함수가 나열되어 연산을 수행하는 예제이다. <표 6>는 fibonacci 수열을 C로 구현한 것이다. C 구문을 살펴보면, fibonacci(q-1)과 fibonacci(q-2)라는 재귀 함수가 동시에 등장한다. 아래의 구문 피보나치 수열에서는 2개의 재귀 함수와 2개의 초기 값이 존재한다.

〈표 6〉 fibonacci 수열에 대한 C 구문

```
#include <stdio.h>
#include <iostream.h>

int fibonacci(int q);

void main()
{
        int a,
        int x,

    cout << "fibonacci number \n";
    cin >> (int)a ,
    x = fibonacci(a).            // 재귀 함수 호출

    printf("fibonacci value is %d\n",x);

}

int fibonacci(int q)
{
  if(q==1 || q==0)                    //초기값
  return 1;
  else                                // 2개의 재귀 함수
  return fibonacci(q-1) + fibonacci(q-2);
}
```

fibonacci 구문의 경우도 Type Ⅱ의 경우와 비슷한 구조를 갖는다. 위의 예제에서는 재귀 함수가 2개 있기 때문에 중간 값에 대한 순환 흐름 그래프가 두 군데에서 일어난다. (그림 15)는 fibonacci 수열에 대한 컨트롤 데이터 플로우를 보여준다. 종결 조건이 두 개이지만, 값이 등일하므로, 초기 값 저장 레지스터는 하나만 사용된다. 초기 값을 저장하는 레지스터 (1)과, 재귀 함수 2개와 중간 값을 저장하는 레지스터 (2), (3) 그리고, 결과 값을 출력하는 레지스터 (4)를 갖는다. fibonacci 수열은 덧셈 연산을 수행하므로, 덧셈 연산 component(adder)를 갖고, 초기 값과 중간 값을 선택하는 mux(1)과, 출력 값을 선택하는 mux(2), 그리고, 반복 횟수를 정하는 카운터(counter)가 VHDL 구문의 component로 들어간다.



(그림 19) fibonacci 수열에 대한 컨트롤 데이터 플로우

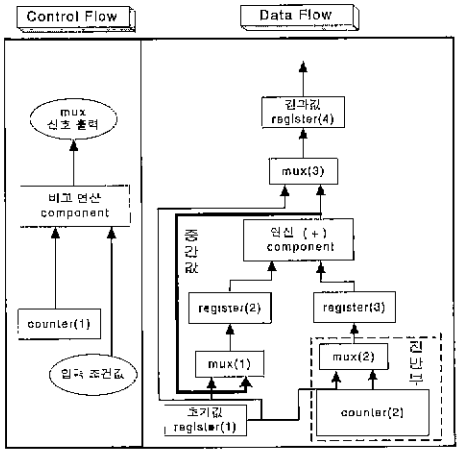
4.3.4 예제 Ⅳ 전반부만 있는 Type Ⅰ, Ⅱ, Ⅲ

<표 7>의 예제는 전반부가 있는 재귀 호출 구문에 대한 예제이다. 전반부는 변수 정의와 변수 할당문이 수행되고, 재귀 호출부에서는 Type Ⅱ의 형태의 재귀 호출이 수행된다.

〈표 7〉 전반부가 있는 function 타입의 재귀 호출 예

```
#include <stdio.h>
int rec(int q);
void main()
{
        int i;      int x;      i = 5;
        x = rec(i);
        printf("recursive call value is %d\n",x);
}
rec(int q)
{
  int y;    y = q;   // 전반부 (변수 정의, 변수 할당문)
  if ( q==0)         return 1;     // 재귀 호출부 시작
  else               y = y + rec(q-1);
        return y;                  // 재귀 호출부 종료
}
```

(그림 16)은 <표 6>의 C 코드에 대한 VHDL 구문의 컨트롤 데이터 플로우를 보여준다. <표 7>의 C 코드에서 재귀 호출 부분의 초기 값 레지스터(1)와, 변수(y,q) 2개가 있기 때문에 입력으로 쓰이는 레지스터는 (2)와 (3)이 필요하며, 결과 값을 출력하기 위한 레지스터(4)가 필요하다. 또한 + 연산을 수행하는 adder와, mux는 초기 값과 중간 결과 값을 동시에 제어할 수 있도록 하는 mux(1)과 전반부의 초기 값과, 중간 값을 제어하기 위한 mux(2), 그리고 결과 값을 제어하기 위한 mux(3)이 필요하다 C 구문의 조건 값과, VHDL 구문의 데이터 플로우를 제어하기 위한 counter(1)와, 전반부에 y = q에 대응하는 counter(2)가 같이 필요하다. (그림 16)의 컨트롤 플로우는 mux의 선택신호 3개를 데이터 플로우에 보내준다. 중간 결과 값은 mux(1)의 입력과 결과 레지스터의 입력으로 들어가게 된다.



(그림 16) 전반부가 있는 재귀 호출의 컨트롤 데이터 플로우

4.3.5. 분류 Ⅴ : 후반부만 있는 Type Ⅰ, Ⅱ, Ⅲ

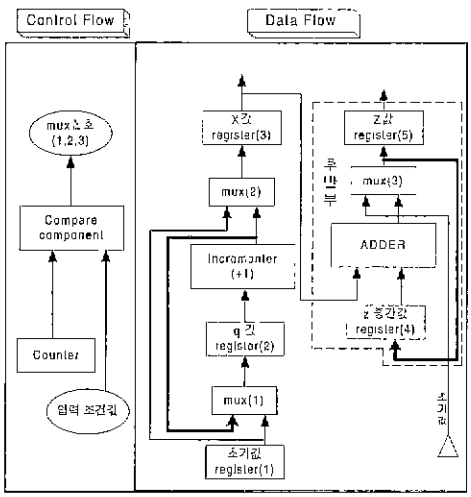
후반부가 있는 재귀 호출에 대한 예제이다. <표 8>의 C 구문에서는 후반부로 Z라는 변수가 들어간다. Z의 값에 재귀 호출부에서 나온 값을 더하는 프로그램이다. 우선 변수 Z를 저장할 수 있는 레지스터와, 연산을 수행하는 Adder, 그리고, 값을 선택하는 mux가 필요하다 이 구문에서는 x와 z라는 결과 값을 출력하게 되어 있다. 따라서, x와 z의 결과 값 레지스터가 필요하다

〈표 8〉 후반부가 있는 재귀 호출

```
#include <stdio.h>
rec(int q,int *x, int *z)
{
        if ( q==0)
                *x = 1;
        else
        {
                rec(q-1,x, z);
                *x = q;
        }
        *z += *x;
        printf("recursive call value is %d\n",*z);
}
```

```
void main()
{
        int i;          int x, z = 0;
         i = 5;
        rec(i, &x, &z);
        printf("recursive call value is %d\n",x);
}
```

(그림 17)은 <표 8>의 C 코드에 대한 VHDL 구문의 컨트롤 데이터 플로우이다. 재귀 호출부에서는 분류 I 형태의 재귀 호출이 이루어지므로, 초기값 레지스터 (1)과 중간값 레지스터(2), 그리고 결과값 레지스터(3)이 쓰인다. 재귀 호출부에서 사용되는 연산 component는 +1을 수행하는 adder가 쓰인다. 또한 재귀 호출부의 값 선택을 위한 mux(1), (2)가 쓰인다. 후반부는 변수 Z를 표현하기 위한 레지스터 (4)와 x와 z값을 더해주는 adder, 그리고 결과값 레지스터 (5)가 사용된다. mux (3)은 z값을 선택하기 위한 component이다. (그림 17)에서는 재귀 호출부와 후반부에서 중간 값의 순환이 이루어진다.



(그림 17) 후반부가 있는 재귀 호출의 Control-Data Flow

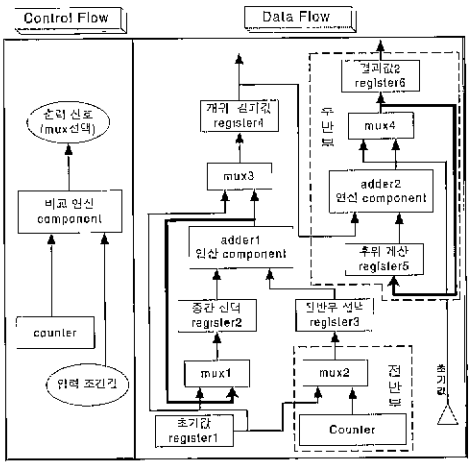
4.3.6 예제 VI : 전반부, 후반부가 동시에 존재하는 Type Ⅰ, Ⅱ, Ⅲ

Function 타입의 재귀 호출에서는 나타날 수 없는 구조이다. Procedure 타입의 재귀 호출은 전반부, 재귀 호출부, 후반부가 동시에 존재할 수 있다. <표 9>는 Procedure 타입의 재귀 호출의 예를 보여준다. 재귀 호출부는 분류 Ⅱ의 형태이고, 이 재귀 호출부에 전반부, 후반부를 추가한 구문이다.

<표 9> 전반부, 후반부가 존재하는 재귀 호출

```
#include <stdio.h>
rec(int q, int *x, int *z)
{ int y;  y = q;              // 전반부
  if ( q==0)                  // 재귀 호출부 시작
          *x = 1;
  else
{
  rec(q-1, &y, z);
   x = q - y;                 // 분류 Ⅱ 형태의 재귀 호출부
}                             // 재귀 호출부 종료
  *z += *x;                   // 후반부
  printf("recursive call value is %d\n",*z); }

void main()
{
  int i; int x, z = 0; i = 5;
  rec(i, &x, &z);
  printf("recursive call value is %d\n",x);
}
```

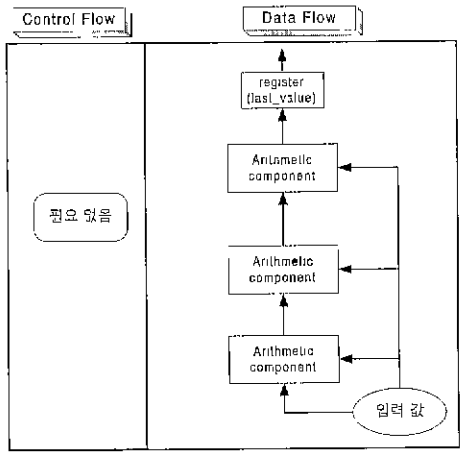
(그림 18)은 전반부, 후반부가 존재하는 procedure 타입의 재귀 호출 예제에 대한 컨트롤 데이터 플로우이다. 여기에서 재귀 호출 부분과 후반부가 나타나므로, 순환 흐름은 두 개가 된다. (그림 18)의 데이터 플로우의 경우, C 구문에서 초기 값을 나타내는 레지스터1, 변수 y, z에 대한 레지스터 (3), (5)와 중간 값을 저장하는 레지스터 (2), 그리고 출력 레지스터 (4), (6)의 총 6개의 레지스터가 사용된다. 재귀 호출 부분에서의 adder(1)과 후위 연산 부분의 adder(2)가 쓰이며, 각각의 선택 신호를 제어하기 위한 mux(1),(2),(3),(4)가 쓰인다. 전반부에서는 y=q 이므로 이는 반복 횟수와 동일한 값이 전반부에서 들어간다. 따라서 카운터를 같이 사용할 수 있다. 컨트롤 플로우는 입력 조건 값과 카운터에 의해 나오는 값을 비교하는 비교 연산 component(구문)이 들어가게 된다. 컨트롤 플로우는 재귀 호출의 중간값을 제어하는 mux(1)과 전반부 제어 신호 mux(2) 그리고, 재귀 호출부의 결과값 제어 신호 mux(3)과 후반부 제어 신호 mux(4)의 선택 신호 값을 출력한다.



(그림 18) 전반부, 후반부가 존재하는 재귀 호출의 컨트롤 데이터 플로우

4.4 Flat하게 펼친 재귀 구문 비교

재귀 호출 구문의 VHDL 구현의 다른 방법인 재귀 구문에 대한 하드웨어 component를 재귀 호출이 수행되는 횟수만큼 전부 삽입하여, 재귀 호출을 Flat하게 펼치는 방법에 대해 알아본다. (그림 19)는 Type II의 2^3에 대해서 재귀 호출 구문을 Flat하게 펼쳐서 VHDL로 표현했을 때의 컨트롤 데이터 플로우이다. 2^3에 대한 재귀 호출 구문을 펼친 경우 출력을 나타내는 레지스터 하나와, 2^3을 연산하는 곱셈기 3개만 있으면 값을 출력할 수 있다. 재귀 호출의 반복 횟수를 제어하기 위한 컨트롤 플로우도 필요하지 않게 된다. 재귀 호출을 Flat하게 펼친 VHDL 구문으로 구현했을 때, 본 논문에서 제시한 방법보다 빠르게 결괴값을 얻을 수 있다. 그러나 이 경우, 재귀 호출이 일어나는 횟수에 따라 component의 개수가 늘어나는 단점이 있다. 만일 component의 개수가 제한되어 있는 경우에는 부분적으로 펼쳐서 앞에서 제시한 컨트롤 플로우를 포함한 하이브리드(Hybrid) 형태로 구성하는 것도 가능하다. 예를 들어, 2^{100}의 경우, 100개의 곱셈기를 사용하는 것이 아니라, 몇 개의 곱셈기를 Flat하게 펼쳐서 구현한후, 이를 재사용하는 방법을 병행해서 구현할 수 있다. 이런 경우에는, 시스템을 제어할 수 있는 제어 구조가 들어가게 된다. 범용 시스템을 구현하기에는 적합하지 않은 구문이지만 하드웨어의 크기보다는 성능에 중점을 둔 특별한 시스템의 경우에는 이 방법을 사용할 수 있다.



(그림 19) Flat한 재귀호출(2^3)

5. 실험 환경

본 논문에서 쓰인 실험 환경은 다음과 같다.

(1) C 코드 생성
- UltraSPARC1, Solaris 2.5.1, WORKSHOP™ SPARCompiler C++4.1

(2) VHDL 코드 생성 및 simulation
- Pentium ce 433, MS-window 98(한글), Aldec Active VHDL 3.3

(3) 합성
- UltraSPARC1, Solaris 2.5.1 Synopsys 98.2 [12]

6. 결 론

지금까지 통합 설계 환경에서 하드웨어-소프트웨어 간의 변환의 일환으로 소프트웨어 설계 분야로 인식되었던 재귀 호출에 대한 하드웨어 설계 방법에 대하여 논의하였다. VHDL로의 변환 중 걸림돌이 되는 문제점들을 정리하고 해결 방법을 제시하였으며, 재귀 호출을 변환하는데 가장 큰 문제점인 할당(allocation)과 스케쥴링(scheduling) & 바인딩(binding)을 하기 위해 그 함수가 몇 번 수행하는지에 대한 문제를 해결하는데

많은 비중을 할애하였다. 본 논문의 알고리즘을 적용하여 C 구문에서 실제로 쓰이는 많은 프로그램을 적용해 본 결과, 대부분의 프로그램들을 합성 가능한 하드웨어로 구현할 수 있었다. 반복 횟수가 정해지지 않는 경우는 재귀 호출은 제어 구조에 특정 카운터가 아닌 비교 구문(ex, q > p)이 들어가게 된다. 모든 재귀 호출은 종결 조건이 있기 때문에 반복 횟수가 정해지지 않는 재귀 호출의 경우에도 문제가 없다. 본 논문의 알고리즘을 적용하게 됨으로써, 좀더 완벽한 소프트웨어와 하드웨어간의 통합 설계의 유연성을 조금 더 획득할 수 있게 되었다. 또한 이를 통해 통합 설계의 궁극적인 목표인 시스템의 전체적인 성능과 설계 시간상으로 우수한 목표 시스템을 구축할 수 있을 것으로 기대된다.

본 논문에서 많은 재귀 호출에 관한 예제에 대해서 성공적인 결과를 보이고 있으나 아직 완전한 것은 아니다. 더 많은 프로그램들에 대하여 성공적인 변환을 하기 위하여 다음과 같은 부분이 보완과 개선이 필요하다.

- 재귀 호출 구문 내에서 또 다른 재귀 호출 구분을 부르는 경우에 대한 논의.
- 재귀 호출 구문의 하드웨어 변환시의 다양한 방식 (순차적인 구조, Flat한 구조, Hybrid 구조 등)에 대한 설계 Trade-off에 관한 연구.

이러한 부분은 앞으로 연구 과제가 될 것이다.

참 고 문 헌

[1] R Ernst, J. Henkel, "Hardware-Software Codesign of Embedded Controllers Based on Hardware Extraction," Handout from first Int'l Workshop on Hardware-Software Codesign, Estes Park, Colo., 1992.

[2] M. F. Parkinson, P. M. Taylor, and Sri Parameswaran, "C to VHDL Converter in a Codesign Environment," Proceedings of VHDL International User's Forum, 1994.

[3] R Ernst, J. Henkel, and T. Benner, "Hardware-Software Cosynthesis for Microcontrollers," IEEE Design & Test of Computers, PP. 64-75, December 1993

[4] Hwayong Kim, Kiyoung Choi, "Transformation from C to Synthesizable VHDL," in Proceedings of Asia Pacific Conference on HDL APCHDL'98, July 1998.

[5] Luc semeria, Giovanni De Micheli, "spc : Synthesis of Pointers in C. Application of Pointer Analysis to the Behavioral Synthesis from C," Proceeding of the 1998 ICCAD, pp.340-346, November 1998.

[6] G D. Jong, B Lin, C. Verdonck, S. Wuytack, and F Catthor, "Background Memory Management for Dynamic Data Structure Intensive Processing Systems," proceedings of International Conference on Computer-Aided Design, pp.515-520, 1995.

[7] V. J. Mooney, C N. Coelho Jr., T. Sakamoto, and G. D. Micheli, "Synthesis from Mixed Specifications," Proceedings of European Conference on Design Automation September 1996

[8] Matthew F Parkinson, Paul M. Taylor, and Sri Parameswaran, "An Automated Hardware/Software Codesign(HSC) using VHDL," proceedings of the first Asia Pacific Conference on Hardware Description Languages and their Applications(APCHDLSA '93), December 1993.

[9] Douglas L. Perry, "VHDL," Mcgraw-Hill, Inc., 2nd edition, 1994

[10] Giovanni De Micheli "Synthesis and Optimization of Digital Circuits," McGraw Hill, Highstown, Nj, 1994.

[11] Peter J. Asbenden, "The Designer's Guide to VHDL," Morgan Kaufmann Publishers, Inc., 1996

[12] "Synopsys tool 교육", 반도체 설계 교육 센터(IDEC), 1998

[13] A. V. Aho, R SEthi, and J. D. Ullman, "*Compilers-Principles, Techniques, and Tools*," Addison-Wesley Publishing Company, 1987

[14] "*Outside C*" Reference Manual - Internet

이 정 아

e-mail : jeong@rain.chosun.ac.kr

1982년 서울대학교 컴퓨터공학과 (학사)

1985년 미국 인디애나주립대학 컴퓨터학과(석사)

1990년 미국UCLA 컴퓨터공학과 (박사)

1990년~1995년 미국 휴스턴주립대학 전기전산공학과 (조교수)

1993년~1994년 미국 국립초전도가속기연구소 (객원연구원)

1995년~현재 조선대학교 컴퓨터공학부(교수)

관심분야 : Computer Arithmetic, Application-Specific Processor Design, (Re)Configurable Computing

홍 승 완

e-mail : swhong@rain.chosun.ac.kr

1998년 조선대학교 전자계산학과 졸업(학사)

2000년 조선대학교 대학원 전자계산학과(석사)

2000년~한국 통신 연구소 (위촉연구원)

관심분야 : VHDL, Hardware-Software Codesign, DSP