논문 2009-46TC-11-6

FPGA ORB 활용을 위한 OMG IDL의 변환 방법

(Translation of OMG IDL for Supporting The FPGA ORB)

정 혜 경*, 배 명 남**, 이 인 환**, 이 용 석*

(Heakyung Jeong, Myungnam Bae, Inhwan Lee, and Yongseok Lee)

요 약

HAO는 코바 기반의 로직 컴포넌트를 수용하기 위해 FPGA에 탑재되는 ORB엔진이다. 본 논문은 HAO기반 로직 컴포넌트 개발을 지원하기 위해, IDL로부터 하드웨어 기술 언어인 VHDL로의 변환 규칙과 이에 따른 스켈리톤 로직의 생성에 대해 기술한다. 이를 통해, 범용 프로세서, FPGAs 등의 분산 다중 프로세서 환경에서 컴포넌트간의 상호운용성을 보장할 수 있으며, 아울러, 로직 수준의 컴포넌트 개발을 통해 성능 개선이 가능하다.

Abstract

HAO is a ORB engine to support the logic-based CORBA development environments in FPGA. In this papers, in order to support the logic component developments with HAO, we proposes the translation rule from IDL to VHDL, and the generation of skeleton logic code following the rule. It enables to guarantee the interoperability between the components in distributed multi processor environments includes the general purpose processor and FPGAs, and to improve the performance through the usage of logic-circuit.

Keywords: HAO, VHDL, OMG IDL, SDR, SCA

I. 서 론

SCA(Software Communications Architecture)[1]는 JTRS JPEO에 의해 소프트웨어 수준으로 무선체계(waveform)간의 상호운용과 재구성을 보장하기 위해 제안되었으며, 분산 플랫폼을 제공하기 위해 코바(Common Object Request Broker Architecture) 미들웨어를 기반으로 하고 있다. 그 결과, SCA 환경에서 운용되는 프로토콜, 보안, HCI 등의 모든 컴포넌트들은 상호간의 위치 투명성, 분산 처리, 구현 언어 독립성 보장이 가능하다. 하지만, 최근 SCA 컴포넌트에 대한 엄격한 실시간(hard real-time) 처리와 다수의 병행처리 요구 뿐만 아니라 이들의 재구성에 대한 요구가 지속적으로 확대되고 있다. 이러한 요구는 스케줄링과 이에 따른 문맥 교체가 빈번히 요구되는 범용 프로세서 환경에서는 만족되기 어렵다.

FPGA(Field Programmable Gate Array)는 디지털 로직 수준의 프로그래밍을 허용하며 향후 ASIC으로의 자연스러운 전환이 용이하다는 점에서 중요하게 다루어지고 있다[2]. 특히, 범용 프로세서와 달리, 다양한 타이밍 제약의 적용이 가능하여 성능 측면의 융통성뿐만 아니라 최근 재구성 기술 등을 제공하여 높은 설계 유연성을 보장하고 있다[3-4].

이에 따라, SCA 컴포넌트들을 FPGA에서 로직 수준으로 개발하기 위한 여러 방식이 고려되고 있으며, 특히 FPGA에 사용할 수 있는 코바 ORB IP(Intellectual Property) core를 직접 제공하려는 연구가 진행되고 있다[5]. 즉, FPGA상의 디지털 로직은 일련의 코바 인터페이스를 갖는 코바 컴포넌트로 구현되며, 코바 ORB IP core를 사용하여 구현된 방식과 위치에 대한 종속성 없이 상호 연동될 수 있는 투명성을 보장받을 수 있다.

* 정회원, 전북대학교 전자정보공학부
(Division of Electronics & Information Engineering, Chonbuk Nat'l Univ.)
** 정회원, 한국전자통신연구원 USN응용기술연구팀
(USN-based Application Technology Research Team, ETRI)
※ 본 논문은 국토해양부 지능형국토정보기술혁신사업(06국토정보C01)의 연구비지원에 의해 수행되었음.
접수일자: 2009년6월9일, 수정완료일: 2009년9월23일

이때, FPGA상의 디지털 로직이 코바 컴포넌트가 되기 위해서는 VHDL(VHSIC Hardware Description Language)[7]과 같은 하드웨어 기술 언어 수준에서 1) 현재, 코바 컴포넌트간 연동을 정의하고 있는 IDL(Interface Definition Language)로부터 실제 목적 언어인 VHDL로의 구체화 방안과 2) 코바 ORB와 연결되는 다양한 시그널/버스의 설정과 매핑 체계를 제공하여야 한다. 전형적으로, 코바 환경에서 이러한 역할은 IDL 컴파일러가 제공하는 스켈리톤(skeleton)/스터브(stub)과 그 활용을 통해 달성된다[6].

본 논문은 OMG IDL로부터 VHDL로의 매핑을 위한 규칙과 개발 결과에 대해서 기술한다.

Ⅱ. 관련 연구

이 장은 SCA에서 FPGA 로직을 수용하려는 여러 시도와 FPGA의 ORB엔진인 HAO, 그리고 HAO와 연동하기 위한 IDL 컴파일러 고려사항에 대해 기술한다.

1. SCA에서 로직 구현의 수용

JTRS JPEO는 DSP와 FPGA와 같은 새로운 하드웨어 프로세서 수용에 따른 컴포넌트(하드웨어적으로 구현된)의 융통성을 보다 확대하기 위해, 하드웨어 구현을 수용하기 위한 여러 노력을 진행하고 있다. 초기 어댑터 방식[8]에서는, GPP에 서버 로직에 대한 접근용 소프트웨어 어댑터를 별로도 두고, 클라이언트는 어댑터와 코바 통신을 통해 요청하고 응답을 받는다. 어댑터는 수신한 요청을 FPGA상의 서버 로직에 전달하고 처리 결과를 반환받아 다시 코바 통신 체계를 통해 클라이언트에 제공한다. 이러한 방식의 단점은 해당 단말의 하드웨어 자원을 직접 사용함으로 인해 단말의 하드웨어 구성에 의존성을 갖게 된다는 점이다. 결국, 하드웨어 환경의 변경에 따라 하드웨어 컴포넌트의 재사용이 어렵고 프레임워크내 컴포넌트들 간의 상호운용성 보장이 가능하지 않게 된다. 또한, 부가적으로 소프트웨어 어댑터를 사용함으로 인해 하드웨어 로직의 성능 개선, 서비스 품질 개선 등의 효과를 충분히 발휘할 수 없다. 또 다른 시도[9]로서, GPP상의 어댑터 대신에 MicroBlaze와 같은 FPGA상의 소프트 코어에 운영체제와 미들웨어를 포팅하고 이 환경에서 어댑터를 통해 하드웨어 컴포넌트와 로직 수준에서 연동하는 방식도 고려되고 있지만, 결국, 하드웨어 컴포넌트 연동에 대한 추가의 오버헤드라는 점에서 문제 해결이 아니고, 또한

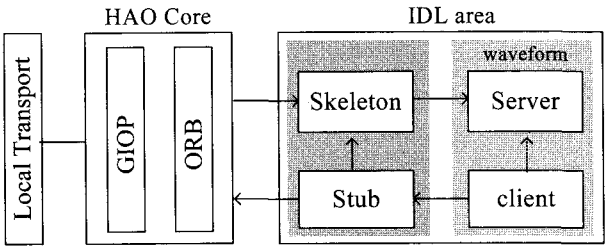

그림 1. HAO의 계층적 구조
Fig. 1. Hierarchical Architecture of HAO.

소프트 코어에 대한 비용과 규모 측면의 부담이 크다.

결국, 보다 근본적으로 시스템의 성능을 향상하고 하드웨어 프로세서를 대상으로 컴포넌트에게 융통성을 폭넓게 제공하기 위한 고성능의 off-the-shelf 기술을 필요로 하게 되었다. 이러한 새로운 기술의 중요한 부분은 FPGA와 같이 특수화된 프로세서에서 운용할 수 있는 표준기반 코바 미들웨어를 제공하는 것이다[5, 10].

2. 하드웨어 ORB(HAO)

HAO[10]는 FPGA에서 빠른 분산 무선체계 개발을 위해 제안되었으며 VHDL을 사용하여 로직 수준으로 개발된 ORB이다(그림 1). HAO는 크게 local transport와 HAO Core로 구성되며, local transport는 적용될 보드 환경에 의존적인 부분을 포함하여, HAO Core와 같은 상위 계층에 보드 독립성을 보장한다. 반면에, HAO Core는 전형적인 ORB 엔진이며, 통상적으로 응용 로직을 포함하는 로직 컴포넌트를 계층화하고 필요한 관리 정책을 바인딩하는 역할을 담당한다. 이를 통해, HAO는 요구 메시지를 인식하고 이의 해석을 통해 대상 로직 컴포넌트를 식별하고 필요한 데이터 변환을 제공할 수 있다. 현재, HAO는 Xilinx FPGA에 IP core 형태로 제공되고 있다[10]. 무선체계는 클라이언트/서버 관점으로 개발자에 의해 직접 작성되는 순수한 로직의 조합으로 구성된다. 이들 클라이언트/서버는 IDL 컴파일러가 제공하는 스켈리톤/스터브를 통해 HAO와 바인드되어 분산 다중 프로세서 환경에서 개발자 관점의 고유 역할을 수행하게 된다.

3. HAO에서 IDL 컴파일러

무선체계 개발자는 스터브/스켈리톤과 바인드됨에 따라 분산 네트워킹과 관련된 부분은 직접 고려하지 않아도 된다. 네트워크를 통한 서비스의 요청(request)과 응답(response)의 전달은 HAO를 통해 처리되기 때문이다. 따라서 HAO를 활용한 로직 개발 환경에서 하드웨

어 컴포넌트를 설계하는 과정은 기존 소프트웨어 컴포넌트를 개발하는 과정과 크게 다르지 않다.

HAO 환경에서 무선체계의 개발 단계는 다음과 같다. 먼저, IDL로 인터페이스(interface)를 정의한다. 이후, IDL 컴파일러는 정의된 인터페이스로부터 스터브와 스켈리톤을 생성하며, 마지막으로 개발자는 스터브/스켈리톤에 목적한 서버/클라이언트 코드를 추가하여 완전한 무선체계를 완성한다. 이때, 빠른 분산기반 무선체계 개발을 위해 개발자의 코드 추가를 최소화할 수 있어야 하며, 분산 환경의 세부적인 사항을 추상화하기 위해 IDL과 목적 언어(여기서는 VHDL)의 변환과 규칙을 매핑해 주는 IDL 컴파일러가 필요하다.

현재, VHDL은 IEEE에 의해 공인되어 하드웨어 개발과 문서화에 사용되고 있으며, 광범위한 기술 능력으로 시스템 레벨에서 게이트 레벨까지 하드웨어 회로 표현이 가능하다[7]. 이러한 VHDL을 지원하는 IDL 컴파일러를 개발하기 위해, IDL로부터 VHDL 스터브와 스켈리톤으로의 변환 규약을 정의하여야 한다.

현재 OMG(Object Management Group)는 VHDL로의 변환 규칙을 제공하지 않기 때문에, 본 논문에서는 관련 언어 변환 규약을 참조하고 HAO와 VHDL의 특성을 고려하여 정의하여야 한다. 본 논문에서는 C 변환 규약[11]을 기반을 두었으며 특히 코바 기반의 SCA 미들웨어 구현에 주로 활용된 코바 기능의 변환 요구사항을 충분히 만족시킬 수 있도록 정의하였다.

III. IDL에서 VHDL로 변환 규약

이 장은 VHDL로의 변환 규약에 대해 [11]에서 언급된 26개 주제를 크게 5개로 분류하여 기술한다.

그림 2는 변환 규약에 따라, IDL 컴파일러가 제공해야하는 스켈리톤 로직 코드의 가장 기본적인 예이다(스터브는 스켈리톤의 특수한 사례이므로 생략). 스켈리톤 로직은 크게 데이터 변환 로직 블록(Data Converter), 로직 선택 로직 블록(Logic Selector), 그리고 메시지 생성 로직 블록(GIOP Message Generator)으로 구성되어야 하며, LS는 버스(data_in)상의 오퍼레이션 이름으로부터 인가되어야 할 하드웨어 컴포넌트를 식별하고, 상태에 따라 전체 로직을 제어하는 기능을 수행한다. DC는 버스(data_in)상의 GIOP 데이터 해석과 변환을 수행하며, 주로 파라미터 데이터의 최적화를 위해 패딩을 제거하고 압축된 형태로 로직 컴포넌트에게 전송한다. MG는 응답이 요구된 경우(혹은 응답 요구가 없지만 예외를 반환해야 하는 경우)에 요청한 코바 컴포넌트에게 응답하기 위한 GIOP 메시지의 구성을 담당한다.

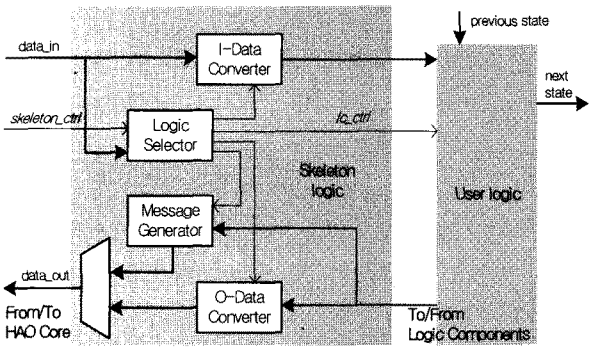

그림 2. 스켈리톤 로직의 블록도
Fig. 2. Block diagram of Skeleton Logic.

다음 각 절은 스켈리톤 로직을 구성하기 위해 필요한 규칙들로 IDL 정의에 따른 의미 변환 및 구체적인 적용 방식을 포함한다.

1. 인터페이스(Interface) 매핑

IDL의 인터페이스는 가장 핵심적인 정의이며, 다른 모든 코바 컴포넌트들 간의 상호 호출에 필요한 오퍼레이션과 관련 정의들(타입, 예외, 상속 등)을 포함한다.

특히 VHDL로의 매핑 과정에서, 상속(inheritance)과 같은 많은 동적인 요소들을 고려하여야 있다. 하지만, FPGA의 활용 목적상, 로직의 단순화와 효율성이 중요하고 또한 다중 패킷 처리와 고정된 반복적인 병행 처리에 주로 활용된다는 점에서 가변성을 포함하는 동적 처리는 제한적으로 허용해야 하며, 허용된 동적 요소들은 가급적 컴파일 시점에 그 특성을 정적으로 고정할 수 있도록 변환하는 매핑 수단을 고려해야 한다.

가. 인터페이스 변환

인터페이스는 내부 상태 저장을 위한 애트리뷰트 요소와 특정 처리를 수행하는 오퍼레이션 요소를 정의하기 위해 interface itf { attr; opr(); }; 형태로 구성한다.

IDL에서 VHDL로의 변환 과정에서, IDL의 인터페이스와 이 요소들은 계층적으로 구성되는 VHDL의 다수 entity들로 매핑된다. 즉, IDL 인터페이스는 스켈리톤 코드내 하드웨어 컴포넌트(skel entity)로 변환되며, 구체적으로 오퍼레이션과 애트리뷰트들은 각각 로직 컴포넌트(opr entity와 attr entity)들로 변환된다. 이외에 하드웨어 컴포넌트를 통해, 로직 컴포넌트를 인가하고 필요한 데이터 변환을 수행할 제어용 entity(ls, dc, mg 등)들도 추가로 생성된다.

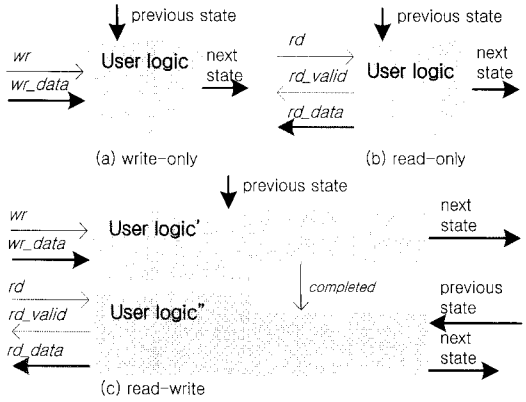

그림 3. 접근 모드에 따른 포트맵 구성
Fig. 3. Port map by the access mode,

```
-- VHDL skeleton code
entity ls is
  port ( ... );
end ls;
architecture arch_ls of ls is
end arch_ls ;

entity dc is
  port ( ... );
end dc;
architecture arch_dc of dc is
end arch_dc ;

entity skel is
  port ( ... );
end skel;
architecture arch_skel of skel is
  component itf_impl is
    port( ... );
  end component;
begin
  impl_inst : itf_impl
  port map ( ... );
end arch_skel ;
```

```
-- VHDL user code

entity itf_impl is
  port ( ... );
end itf_impl;

architecture arch_impl of
             itf_impl is
  component opr
    port ( ... );
  end component;
  component attr
    port ( ... );
  end component;
  signal real_attr ;
begin
end arch_impl;
```

예에서, 인터페이스 itf는 VHDL의 skel entity내 itf_impl을 포함하도록 변환되며, itf의 실제 오퍼레이션과 애트리뷰트는 다시 itf_impl내 sub-entity(opr과 attr entity)들로 변환되었음을 알 수 있다.

나. 오퍼레이션/애트리뷰트 매핑

인터페이스의 오퍼레이션은 기본적으로 요청(request)에 포함된 파라미터를 수신하고, 개발자의 처리 로직의 완료 후 그 결과(반환값, 파라미터, 묵시적 반환값)를 반환하는 독립적인 entity로 변환된다(애트리뷰트는 별도의 저장소를 갖고 결국 이에 대한 접근 오퍼레이션으로 변환(즉, 로직 컴포넌트로)되므로 궁극적으로 오퍼레이션과 동일하게 처리됨).

이때, 오퍼레이션은 설정 모드 혹은 읽기 모드인지에 따라 HAO를 통해 파라미터들을 송수신하고 제어하기 위한 적절한 포트맵(port map) 구성을 고려해야 한다. 오퍼레이션의 포트맵은 필수 제어인 nRST와 CLK을

표 1. 지원하는 IDL 데이터 타입
Table 1. IDL data types allowed in HAO.

IDL data type	
simple type	. char, octet(std_logic_vector(7 downto 0) in VHDL) . unsigned short . unsigned long . unsigned long long
complex type	. string of simple types(*) . sequence of simple types . any of simple types . structure of simple types, string, and any

가지며, 설정 용도의 로직인 경우 컴포넌트를 인가하는 wr 시그널과 설정 데이터의 버스인 wr_data을 포함하며, 읽기 로직인 경우 해당 컴포넌트를 인가하는 시그널인 rd와 읽은 데이터를 전송할 버스인 rd_data, 추가로 반환 가능한 상태(즉, 응답 데이터의 송신 준비 완료)임을 나타내는 rd_valid를 포함하도록 변환될 것이다(시그널과 전송 버스의 구성에 대해서는 2절에서 설명).

다. 데이터 타입 매핑

오퍼레이션을 통해 송수신을 지원하는 데이터 타입은 VHDL에서 사용가능하며 로직 컴포넌트 개발시 준수해야 하는 SCA[1]규격에 포함된 사례로 한정하여 지원한다. 이는 IDL의 동적 요소와 마찬가지로, 로직에서 데이터 크기의 가변성은 단순화를 위해 엄격하게 제한되어야 하기 때문이며, 실제로 FPGA의 응용은 고속 병행 IO 처리 등을 주로 목적하고 있어, 복잡한 데이터 타입을 요구하지 않는다[12]. 현재, HAO가 지원하는 데이터 타입은 표 1과 같다. 표에서 보인 형태 이외의 타입과 중첩 구조, 그리고 Union을 허용하지 않는다. 또한, simple type에서 문자열과 일부 수치 타입(double, float, fixed)은 지원하지 않는다.

2. Invocation of Operation

IDL 오퍼레이션(애트리뷰트의 변형 포함)을 VHDL entity로 변환 후, 이 들에 대한 호출과 파라미터 전달에 대한 추가의 고려가 필요하다. 이 절은 오퍼레이션의 식별, 호출, 반환값, 파라미터의 타입(in, out, inout)을 고려한 VHDL로의 변환 규약에 대해서 다룬다.

가. 오퍼레이션 호출 매핑

IDL 오퍼레이션(애트리뷰트 포함)에 대응하는 로직 컴포넌트는 전원 인가와 동시에 동작한다. 이때, 오퍼레이션의 파라미터와 반환값에 따라 앞의 설명처럼 IO 포트맵은 다르게 구체화된다. 즉, 파라미터의 입출력 형태와 반환값의 존재 유무에 따라, read-only, write-only,

그리고 read-write로 구분된다(그림 3). 아울러, 식별하고 인가하는 제어가 추가로 고려되어야 한다. 예를 들어, c) read-write 오퍼레이션은 두 개 로직 컴포넌트(User logic′, User logic″)로 각각 구분하여 변환되고 각각에 필요한 제어를 수행하여야 한다. 즉, 1) LS는 wr을 통해 User logic′을 인가하고, 이에 따라 User logic′은 처리를 수행하며, 처리를 완료한 후 completed를 활성화한다. 2) 이 시점에, rd에 의해 인가된 User logic″는 자신의 처리를 수행하며, 요청에 대한 응답(response)이 가능한 시점에 rd_valid를 활성화한다. 3) rd_valid의 활성화에 따라, skel은 rd_data를 통해 결과(오퍼레이션의 반환값, 출력용 파라미터, 추가 묵시적 상태)를 획득할 수 있도록 제어되는 과정이 스켈리톤 코드로 생성된다.

나. IOR과 오퍼레이션 식별

wr(혹은 rd)은 하나의 로직 컴포넌트를 유일하게 구분하는 식별자로 사용된다. 코바[6]에서 식별과정은 두 단계를 거쳐 완성된다. 첫째는 ORB와 코바 컴포넌트 수준의 식별 단계, 두 번째는 컴포넌트내 오퍼레이션의 식별 단계이다.

(1) ORB와 코바 컴포넌트 수준의 식별

코바에서 코바 컴포넌트를 식별하기 위해 IOR을 사용한다. 범용 프로세서 환경에서 코바 컴포넌트는 동적으로 생성되므로, 이에 따라 해당 컴포넌트의 IOR 역시 동적으로 생성되어야 한다. 하지만, 로직 컴포넌트는 P&R과정에서 FPGA의 특정 위치에 고정되어야 하므로, 범용 프로세서 환경과 달리 실행시점에 동적으로 생성할 필요가 없다. 이에 따라, 코바 컴포넌트의 IOR은 IDL 컴파일 과정에서 결정될 수 있고, 이 컴포넌트는 고정 IOR을 사용함으로 IOR생성과 해석 관련된 부

표 2. HAO에서 IOR 구성
Table 2. IOR definition in HAO.

IOR:	
01 *000000*	Host endian
0d000000	Str_len(13)
49444c3a4563686f3a312e3000 *000000*	IDL:Echo:1.0
01000000	Seq_len(1)
03000000	Tag(3=IOP_TAG_HAO)
18000000	Mislen(24)
01	Data endian
01000000	Strlen(1)
10 *000000*	Host(=Id0)
0300 *0000*	Port(=3)
04000000	Len(4)
f2000000	Object_key(=f2)

가 로직 구성의 생략이 가능하다는 장점이 있다.

HAO 환경에서, IDL 컴파일러에 의해 작성되는 하드웨어 컴포넌트 IOR의 전형적인 구성은 표 2와 같다. 범용 프로세서 환경에서 이 IOR은 먼저 목적 하드웨어 컴포넌트가 바인드된 HAO를 장치 드라이버를 통해 검색하도록 하고(tag=IOP_TAG_HAO), id0인 FPGA의 3번 IO로 구분한다. 최종적으로 구분된 HAO에 컴포넌트 식별자 f2인 하드웨어 컴포넌트를 지칭하고 있다.

(2) 오퍼레이션의 식별

IOR을 통해 하드웨어 컴포넌트가 식별되면, GIOP 요구 메시지를 통해 처리를 요구(request)할 수 있다.

대표적인 요구 메시지 예는 표 3과 같다. 표에서, IDL 컴파일러는 요구된 GIOP 메시지내 식별자 부분(=0x00000f2)과 오퍼레이션 이름(="echoInt")을 해석하여 실제 로직 컴포넌트를 인가하는 LS 코드를 생성한다. LS는 우선 컴포넌트식별자를 갖는 하드웨어 컴포넌트의 존재여부를 확인하고 존재하지 않는 경우 예외(exception)를 발생한다. 존재할 경우, 오퍼레이션 이름을 해석하여 일치되는 해당 로직 컴포넌트의 로직 인가(wr 혹은 rd) 시그널을 활성화 할 것이다. 즉, LS는 그림 4와 같이 오퍼레이션 이름을 해석하기 위해 현재 상태와 부분 오퍼레이션 이름 비교를 수행하는 순차회로로 구성된다.

표 3. GIOP 메시지 예
Table 3. Examples of GIOP message.

GIOP request:	
47494f50 01000100 28000000	GIOP header
00000000 e0f11200 01000000	
04000000 f2000000	컴포넌트(=0x000000f2)
08000000 6563686f 4e756d00	오퍼레이션(="echoint")
00000000	
64000000	데이터(=100)

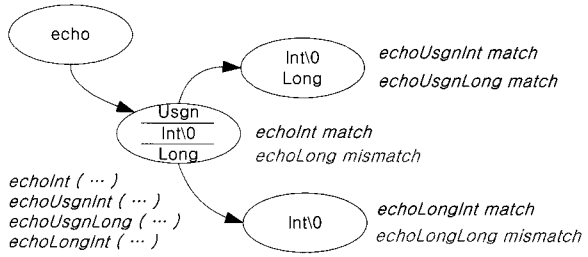

그림 4. IDL 컴파일러에 의한 IOR 생성
Fig. 4. IOR extract from IDL compiler.

다. 데이터 전달

로직 컴포넌트에 대한 인가와 함께 필요한 파라미터

의 전달 과정에 대한 고려가 필요하다. 로직에서 데이터 전달 버스(wr_data, rd_data)는 전원 뿐만 아니라 시그널의 지연으로 인한 타이밍을 고려하므로 버스의 크기나 제어 절차를 최적화하여야 한다. 또한, IDL 데이터 타입에 따른 크기의 다양성(1, 2, 4 byte)을 고려하여야 하며, 가변 길이 데이터로 인한 가변성도 함께 고려하여야 한다.

최적 데이터 전달 버스를 구성하기 위해, 데이터는 기본 타입과 미정의 가변길이를 갖는 복합 타입으로 구분하여 달리 처리한다. 먼저, 오퍼레이션의 파라미터가 기본 타입인 경우, IDL 컴파일 과정에서 길이를 알 수 있기 때문에 파라미터들은 데이터 전달 버스에 직접 포함한다. 이때, GIOP 메시지내 데이터(4byte alignment)는 전달 버스에 패딩을 제거하고 압축(1byte align)하여 가산한다. 예를 들어, short, char 형의 파라미터를 전송하는 경우, 데이터 버스는 24bit 넓이가 된다.

반면에, 가변길이인 복합 타입의 파라미터 전달을 필요로 하는 로직 컴포넌트는 좀 더 복잡하다. 다음 그림은 이를 위한 전달 버스 구성과 부가 제어 로직, 그리고 블록 메모리의 관계를 보인다(그림 5).

복합 타입 데이터의 경우, 파라미터의 크기가 실행 시점이 되어야 결정된다는 가변성을 해결하기 위해 실제 가변 길이의 데이터는 별도의 레지스터(혹은 블록 메모리)에 저장하고 전달 버스에는 고정된 크기로 데이터가 저장된 블록 메모리의 시작 주소와 크기 정보만을 싣는다. 즉, 데이터의 실제 크기와 무관하게, 각각 8bit 넓이(설정 파일을 통해 변경 가능)인 해당 데이터가 존재하는 레지스터(혹은 블록 메모리)의 주소와 데이터 길이만을 데이터 버스에 추가한다(기본 타입 데이터가

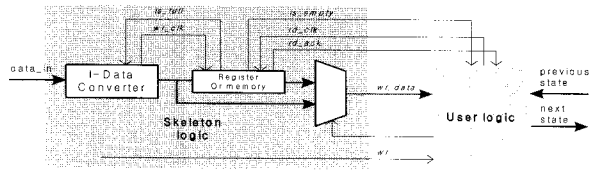

그림 5. 가변길이 데이터에 대한 데이터 버스 변환
Fig. 5. Data-bus mapping for variable-sized data.

표 4. 복합 타입의 데이터 값
Table 4. Data value of Compound data types.

Sequence	길이, { 데이터1, 데이터2 } 02000000 65000000 66000000
any	타입, 데이터 03 00 00 00(long) 65 00 00 00
struct	기본 타입의 열거로 변경

지나치게 많아 일정 한계값(threshold)을 넘어설 경우 일부는 복합 타입 데이터로 간주되어 데이터 버스가 정의될 수 있음).

현재, 블록 메모리를 통해 전달되는 복합 타입에는 struct, any, sequence, string, exception을 포함하고 있으며, 전송되는 비트열의 예는 표 4와 같다(exception은 3.5 절에서 설명). 표에서, sequence 데이터의 경우, 실제 데이터 크기는 12byte이지만, 전달 버스 wr_data에는 8bit 크기의 블록 메모리 상의 시작 위치 주소값과 실제 데이터의 길이인 '0x0c'의 16bit만 전달된다. 이후, 로직 컴포넌트는 블록 메모리의 동기 요청(rd_clk)과 확인(rd_ack)을 통해 블록 메모리로부터 실제 데이터를 획득할 수 있다. 이를 통해, 로직 컴포넌트로의 데이터 버스를 최소화할 수 있고, 복합 데이터의 가변성을 해소할 수 있다.

3. Inheritance

IDL에서는 상속 계층 구조에 따라 상위 인터페이스의 오퍼레이션을 하위 인터페이스에서 사용할 수 있다. 오퍼레이션을 구체화한 로직 컴포넌트는 IDL 컴파일 과정에서 버스와 제어가 고정된다. 하지만, 상속의 오버라이딩에 의해, 해당 오퍼레이션의 재정의가 가능하여야 한다. 이 경우, 클라이언트/서버의 구체화 단계에서 하드웨어 컴포넌트들간에 새로운 제어와 버스가 필요하다. 즉, 오버라이딩된 오퍼레이션을 포함하는 하드웨어 컴포넌트와 상속 구조에서 그 하위로 정의된 하드웨어 컴포넌트간에 제어와 버스 구조가 요구된다.

이러한 부가적인 제어와 버스 구조는 개발자가 직접 반영하기 어려우며, 높은 가변 바인딩에 따른 타이밍 문제의 발생 확률도 매우 높다는 문제점이 있다.

따라서, 본 논문에서는 상속에 의한 오퍼레이션의 동적 바인딩과 애트리뷰트 공유로 인한 제어와 버스 구조의 복잡화 문제를 해결하기 위해, IDL 컴파일러가 상속 계층 구조를 이루는 모든 인터페이스들을 단일의 하드웨어 컴포넌트로 그룹화하는 방식을 사용한다. 이를 통해, 오버라이딩으로 인한 변경은 해당 하드웨어 컴포넌트 내부로 한정할 수 있으며, 개발자의 고려 범위를 매우 크게 줄여줄 수 있어, 결과적으로 잠재적인 타이밍 문제와 하드웨어 컴포넌트들간의 가변적인 연동을 보다 단순화할 수 있다.

즉, 그림 6과 같은 상속 관계를 갖는 인터페이스 A, B, C는 하나의 하드웨어 컴포넌트로 그룹화된다. 추가로, 상속 기능은 개념적으로 다음과 같은 개발자의 코

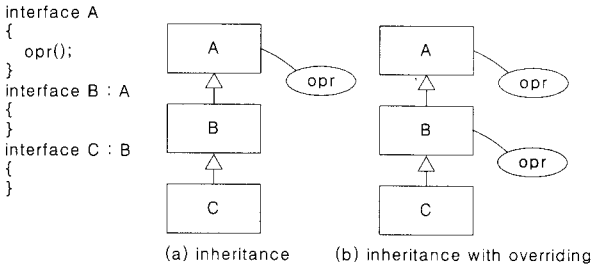

그림 6. 인터페이스 상속의 예
Fig. 6. Examples of inheritance.

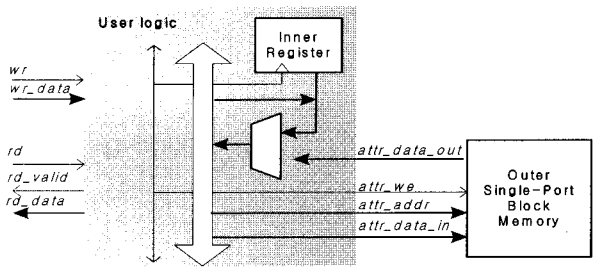

그림 7. 애트리뷰트의 공유 방법
Fig. 7. Representation of shared attribute.

드 반영을 통해 달성하도록 하고 있다. 예로, 다음은 하드웨어 컴포넌트에서 entity A, B, C가 인터페이스 A의 오퍼레이션 opr을 모두 함께 공유하는 경우이며,

```
entity A_op
...
architecture
If ( wr=A::op or wr=B::op or wr=C::op) {
}
end architecture
```

다음 예는 entity B, C가 인터페이스 B의 opr을 공유하는 과정을 보인다.

```
entity A_op
...
architecture
    If ( wr=A::op) {
    }
end architecture

entity B_op
...
architecture
    If (wr=B::op orwr=C::op) {
    }

    B_op_enable <= '1' when wr=B::op or wr=C::op
                   else '0';
end architecture
```

이와 같이, 오버라이딩의 적용 허용 여부는 하드웨어 개발자가 부가적으로 오버라이딩을 직접 명시하는 방법을 통해서 제한적으로 허용하고 있다.

4. 애트리뷰트 구체화

코바 개발 환경에서, 공유 애트리뷰트에 대한 세부적인 구현은 개발자의 선택 사항이다. IDL 컴파일러는 하드웨어 컴포넌트내에 공유 애트리뷰트에 접근하기 위해 앞에서 설명한 제어와 버스 구조를 갖는 기본 로직 컴포넌트들(즉, 오퍼레이션 set, get)만을 제공한다.

한편 그림 7과 같이, 공유 애트리뷰트를 구체화하는 방식은 활용 목적에 따라 SCA 컴포넌트내 레지스터로 구현되거나 혹은 외부의 블록 메모리로 구현될 수 있으며, 이의 선택은 공유 애트리뷰트의 크기, 접근 범위 등을 고려하여 개발자가 직접 정의하여야 한다.

5. Exception

예외(exception)는 오류 상황이나 특수한 내부 상태를 반환하기 위한 복합 데이터이다. 예외는 미리 정의된 시스템 예외(system exception)와 IDL 문맥에서 개발자가 정의하는 사용자 예외(user exception)가 있다.

예외가 발생된 경우, 예외에 대한 처리는 모든 로직 컴포넌트에서 공유가능하다. 따라서 예외를 반환하고자 하는 로직 컴포넌트는 예외의 발생과 예외 번호만을 보고하는 로직이 포함되며, 해당 예외에 대한 처리와 메시지의 구성은 스켈리톤 코드에서 수행하도록 코드가 생성된다.

이를 위해, IDL 컴파일러는 IDL에서 정의한 사용자 예외와 시스템 예외를 파싱하여 유일한 예외 번호와 문자열화 된 예외설명을 ROM 데이터로 자동으로 생성하여 제공한다. MG는 그림 8과 같이 정의된 Exception ROM을 활용하여 발생된 예외에 따라 응답 메시지를 구성할 때 사용한다. 예외의 반환을 위해, 예외가 발생하도록 정의한 write-only 오퍼레이션의 경우, IDL 컴파일러에 의해 예외를 반환할 수 있는 read-write 오퍼레이션으로 자동 변환된다. 이를 위해 개발자는 IDL 정의 과정에 예외를 반환하는 오퍼레이션은 "exception Invalid {}; interface Echo { Long echoNum() raise (ibnvalid); }"와 같이 raise문을 반드시 포함해야 한다.

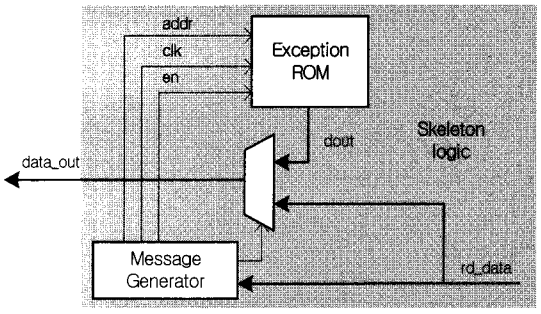

그림 8. 예외 처리
Fig. 8. Exception handling in VHDL.

만일, Echo 오퍼레이션의 수행 과정에 Invalid 예외가 발생하였다면, MG는 예외 타입(=0x01, user exception)을 설정하고, 개발자가 정의한 예외 설명 문자열의 길이(=16)와 내용(=0x49444c3a, 0x496e7661, 0x6c69643a, 0x312e3000, 즉 "IDL:Invalid:1.0")을 포함하는 응답 메시지를 구성할 것이다. 시스템 예외의 경우, MG는 예외 타입(=02, system exception)과 예외 설명, 그리고 minor와 completed 여부 정보를 추가한다.

6. 미지원 기능들

IDL은 언어 독립적으로 컴포넌트들간의 인터페이스를 정의하기 위한 많은 기능들을 제공하고 있다. 특히 다양한 동적인 기능과 역할을 정의하였다. 하지만, FPGA에서 로직 구현은 P&R(Place & Routing)에 따른 물리적인 배치와 타이밍 제약뿐만 아니라 효율성을 위해 매우 제한적인 동적 기능만을 허용해야 한다. 이러한 관점에서, 범용 프로세서상의 다른 소프트웨어 개발 언어와는 지원하는 기능과 범위를 다르게 정의하여야 한다. 이에 따라, [11]를 기준으로 VHDL로의 매핑과정에서 제한하거나 미지원하는 기능은 표 5와 같다.

이들 기능은 다음 이유로 제외되었다. 첫째, FPGA의 활용은 반복적이면서 단순한 고속 I/O 처리에 주로 활용된다는 측면에서 최상위 응용 관점에서 요구되는 가변길이의 복잡한 수치 타입은 제외하였다. 둘째, ORB 초기화와 같은 동적 초기화의 일부 기능은 컴파일단계에서 컴포넌트의 개수, 위치와 자원할당이 이미 고정되므로 동적 초기화의 의미가 없으므로 개발환경 측면에서 초기 구동시에 일괄적으로 처리 할 수 있다. 셋째, 실행시점에 요구되는 고수준의 동적 바인딩은 컴포넌트의 물리적인 위치와 버스가 미리 고정되는 FPGA 환경에서 가능하지 않다. 물론 이러한 요구가 로직 블록에 대한 부분 재구성(Partial Reconfiguration)을 통해 극복될 수 있지만, 소프트웨어 관점에서 동적 바인딩은 개발자에게도 극히 높은 복잡도와 비효율성을 수반한다.

표 5. HAO에서 미지원 기능
Table 5. Unsupported feature in HAO.

basic types	복잡 수치형 미지원, string 미지원
compound types	union/array 불허, nested 구조 불허
inheritance	overriding만 지원
DSI, include file, pseudo object, ORB initialized operations	미지원
scope	모든 interface는 전역 영역으로 정의됨(영역화된 이름은 미허용)

Ⅳ. 개발 결과

본 논문에 적용된 코바는 고속 통신용 코바인 UniORB[13]를 기본 플랫폼으로 사용하였다. IDL 컴파일러는 UniORB의 IDL 컴파일러 전처리기(front end)에 스터브와 스켈리톤을 생성하는 VHDL 후처리기(back end)를 추가하였다. 전처리기는 모든 언어에 공통적으로 사용되며, 기본적으로 IDL로 정의된 인터페이스를 어휘 분석과 구문 분석 과정을 거쳐 언어 독립적인 AST(Abstract Syntax Tree)로 변환한다(그림 9).

개발된 VHDL 후처리기는 AST의 각 노드를 탐색하면서 본 논문에서 정의한 스터브/스켈리톤 변환 규약을 적용하여 VHDL 코드를 생성하며, 이외에 각 코바 컴포넌트에 대한 IOR 저장소와 보드/HAO/로직 컴포넌트 등에 대한 하드웨어적 설정을 별도로 생성한다.

[10]에서는 본 연구의 결과인 IDL 컴파일러를 사용하여, UniORB의 모든 Testbench를 컴파일하고 얻어진 스터브/스켈리톤과 개발자의 클라이언트/서버 로직 코드들을 HAO와 함께 합성/P&R 과정을 거친 후 실제 FPGA에서 성능을 평가한 바 있다. IDL 컴파일러가 생성한 스켈리톤 코드의 상당 부분이 HAO 내부에 내재되므로, IDL 컴파일러의 코드 성능이 명확히 구분되어지기는 어렵다. Testbench의 평균 트랜잭션 성능 관점에서, 개발자 로직을 포함하는 로직 컴포넌트와 하드웨어 컴포넌트, 그리고 HAO를 포함해서 2,900여 로직 셀로 구성되었으며, 커널 레벨에서 UniORB의 성능과 비교하여 평균 30배 이상의 성능 차이가 있었다.

특히, 오퍼레이션의 파라미터에 대한 성능 처리에서 전체적으로 가변길이 데이터가 포함된 경우 전체 트랜잭션의 성능에 미치는 개선 비율이 20%정도 크게 나타났으며, 이는 범용 프로세서 환경에서 가변 데이터의 처리와 관련하여 외부 메모리에 대한 추가 접근 오버헤

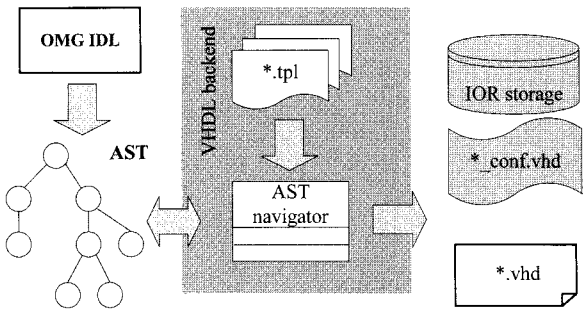

그림 9. IDL 컴파일러 구성
Fig. 9. Architecture of IDL Compiler.

드와 반복 처리 구조로 인한 오버헤드가 동일한 내부 클록에 의해 동기화되는 내부 블록 메모리를 사용하는 FPGA 환경에 비해 보다 큰 부하 요인이기 때문이다.

또한, GIOP 메시지 처리에서 가변길이 파라미터 다음으로 많은 오버헤드가 되고 있는 IOR내 컴포넌트 식별자에 대한 정적 생성 기능도 성능 개선에 일정 부분 기여하고 있다. ORB에 따라 다르지만, 메시지의 크기가 짧을수록 보편적으로 사용되는 24byte 대신, IDL 컴파일러에서 생성한 2(혹은 4)byte 식별자는 메시지 처리 과정을 단축한다. 단위 트랜잭션 처리에서 GIOP 메시지 처리 비율이 높지 않지만(평균 8% 이하), GIOP 메시지 처리 과정이 전체 시스템의 공용자원인 시스템 버스에 대한 제어를 가진 체 동작한다는 관점에서 시스템 버스에 대한 빠른 반환은 전체 시스템 성능에 영향을 미친다는 점을 고려하였다.

V. 결론 및 향후 과제

HAO는 SCA 환경에서 코바 기반의 로직 컴포넌트 수용하기 위해 FPGA에 탑재되는 ORB엔진이다. 본 논문에서는 HAO의 로직 컴포넌트 개발을 지원하기 위해, IDL로부터 디지털 로직 기술 언어인 VHDL로의 변환 규칙과 이에 따른 스켈리톤 로직의 생성 방식, 그리고 그 결과에 대해 기술하였다.

IDL로부터 VHDL로의 효율적인 변환을 위해, 기존 소프트웨어 개발 언어의 매핑 규칙으로 부터 FPGA에서 가능하지 않은 동적 특성들을 배제하였고, 실제 개발자 로직이 HAO와 연동되기 위해 최적화된 스켈리톤 로직 코드의 구조와 역할에 대해 함께 기술하였다. 특히, 기존 소프트웨어 개발 언어용 IDL 컴파일러와는 달리, 하드웨어 의존적인 부분을 미리 정의된 설정 파라미터로 생성하여 ORB엔진인 HAO의 하드웨어 의존성을 최소화하는 한편, 지원되는 동적 특성들에 대한 최적화를 위해 IOR, 초기화 등의 메타 정보들을 추가로 제공하고 있다.

이를 통해, 범용 프로세서, FPGAs 등의 분산 다중 프로세서로 구성되는 SCA 개발 환경에서 컴포넌트의 구현 방식(소프트웨어 혹은 하드웨어적 구현), 컴포넌트의 위치(범용 프로세서나 FPGA)에 무관하게 컴포넌트 간의 상호운용성을 보장할 수 있다. 아울러, 직접 로직 수준의 컴포넌트 개발이 가능하여 무선체계의 부가 오버헤드를 제거할 수 있으므로, 로직 컴포넌트의 활용 목적인 실시간 병행 처리를 통한 성능 개선 효과를 증대시킬 수 있다. 현재, HAO와 IDL 컴파일러는 로직 컴포넌트에 대한 독립된 클록 제공 방안, 다중 POA간 메시지 버퍼 공유 방안, SDR 기본 프리미티브 구체화 기능, 부분 재구성 기능을 사용한 동적 장치 적재/해제 기능과 같은 부가 기능, 그리고 다양한 성능 개선 방안의 적용이 진행되고 있으며, 추가로, Xilinx의 대표적인 칩들에 대한 확장을 추진하고 있다.

참 고 문 헌

[1] Joint Tactical Radio Systems, "Software Communications Architecture Specification V2.2." Nov. 2002.
[2] Xilinx, Virtex-4 FPGA User Guide(UG070) V2.5, June 2008.
[3] Cindy Kao, "Benefits of Partial Reconfiguration," XCell Journal, pp. 65-67, 2005.
[4] Mark Goosman, "Changing Horses in Midstream: Partial Reconfiguration for FPGA Designs," COTS Journal, May 2006.
[5] Joe Jacob, "CORBA for FPGA: The Missing Link For SCA Radios," COTS Journal, Vol. 9, No. 1, pp. 30-33, Jan. 2007.
[6] Object Management Group, "The Common Object Request Broker Architecture: Core Specification Revision 3.0." Dec. 2002.
[7] Zainalabedin Navabi, VHDL: Modular Design and Synthesis of Cores and Systems, 3rd Ed. McGraw-Hill, 2007.
[8] Mark Hermeling, "Component-based support for FPGAs and DSPs in Software Defined Radios," SDR Forum, Nov. 2006.
[9] John Huie, et. al., "Synthesizing FPGA Cores for Software Defined Radio," SDR Forum, Nov. 2003.
[10] 배명남, 이병복, 박애순, "FPGA에서 SCA 컴포넌트 개발을 지원하는 하드웨어 ORB", 한국통신학회논문지, 제34권, 제3호(무선통신), pp. 185-196, 2009.
[11] OMG, "C Language Mapping ver 1.0"(in http://www.omg.org/docs/formal/99-07-35.pdf)
[12] S. Aslam-Mir, "ICO: Integrity Circuit ORB," PrismTech white paper, 2006.
[13] 장종현, 이동길, 한치문, "개방형 통신 시스템을 위한 고성능, 고신뢰성 CORBA 플랫폼에 관한 연구", 대한전자공학회논문지, 제41권 TC편, 제2호, pp. 19-29, 2004.

저 자 소 개

정 혜 경(정회원)
전북대학교 전자정보공학부
컴퓨터전공 박사과정
대한전자공학회 논문지
제46권 TC편 제10호 참조

이 인 환(정회원)
한국전자통신연구원 USN응용기술연구팀
책임연구원
대한전자공학회 논문지
제46권 TC편 제10호 참조

배 명 남(정회원)
한국전자통신연구원 USN응용기술연구팀
책임연구원
대한전자공학회 논문지
제46권 TC편 제10호 참조

이 용 석(정회원)
전북대학교 전자정보공학부 교수
대한전자공학회 논문지
제46권 TC편 제10호 참조