제19회 한국정보처리학회 춘계학술발표대회 논문집 제10권 제1호 (2003.5)

합성 가능한 IPC 생성 방법에 관한 연구

윤창열, 장경선
충남대학교 컴퓨터공학과
e-mail:yun@ce.cnu.ac.kr

A Generation Method of a Synthesizable IPC

Chang-Ryul Yun, Kyoung-Son Jhang
Dept of Computer Engineering, Chung-Nam University

요 약

IP를 재사용 하기 위해 설계자는 IP의 기능에 대한 이해뿐만 아니라, IP의 인터페이스에 대해 알아야 하고, 인터페이스 프로토콜에 따라 테스트 벤치의 작성과 프로토콜 변환 회로를 설계해야 한다. 이런 인터페이스 관련 작업은 오류가 생기기 쉽고, 많은 시간을 필요로 한다. 이러한 어려움을 극복하기 위해 설계의 수준을 트랜잭션 수준으로 높혀야 한다. 이에 본 논문에서는 IP의 인터페이스 프로토콜을 트랜잭션 수준으로 변환시켜주는 인터페이스 프로토콜 컴포넌트를 제안하고, 이를 합성 가능한 VHDL 형태로 생성하는 방법을 제시한다. 실험을 통해 인터페이스 프로토콜 컴포넌트를 이용한 설계가, 그렇지 않은 설계에 비해 많은 면적을 요구하지 않음을 보인다.

1. 서론[1)]

설계 공정 기술의 발달로 집적도가 현저하게 증가됨에 따라, 예전에 가능하지 못했던, SoC(System on a Chip)의 설계가 가능하게 되었다. 이런 SoC의 설계에서 중요한 점 중의 하나는 IP의 재사용이다. 필요한 모든 모듈을 설계하는 것이 아니라, 기존에 이미 설계 검증된 IP(Intellectual Property)를 사용하는 것이다. 이는 설계시간을 현저히 줄여 주기 때문에, 점점 짧아지는 TTM(Time to Market)을 지킬 수 있다. IP의 재사용은 설계의 복잡도를 줄일 수 있을 뿐만 아니라, 이미 검증된 IP를 사용하기 때문에 그에 대한 신뢰성도 높아진다[1].

SoC 설계에서 IP를 사용하기 위해서는 IP의 기능과 동작환경(timing, 전력소모 등)에 대한 이해 뿐만 아니라 시스템에서 IP를 사용하기 위해서는 인터페이스에 대한 이해도 필요하다. 인터페이스 관련 설계는 오류가 발생하기 쉽고, 시간이 많이 걸리는 작업으로 알려져 있다. 설계 시간과 오류를 줄이기 위한 방법으로 설계의 수준을 트랜잭션 수준으로 높여야 한다[2].

대부분 IP의 인터페이스는 트랜잭션 수준으로 추상화 될 수 있고, 설계자는 트랜잭션 수준의 인터페이스 정보만으로 IP간 인터페이스 합성 회로를 설계할 수 있을 것이다. 이렇게 하기 위해서는, 트랜잭션 수준과 사이클 수준의 프로토콜을 변환해 주는 트랜잭터(agent module)가 필요하다. 트랜잭터는 트랜잭션 수준의 입력을 시그널 또는 사이클 수준으로 변화시켜주거나, 시그널 또는 사이클 수준의 요구를 트랜잭션 수준으로 인식한다. 이런 트랜잭터는 IP의 인터페이스에 대한 정보를 갖고 있으므로, 해당 IP를 사용하는 다른 설계에서 재사용 될 수 있다. 본 논문에서는 이런 트랜잭터를 IPC(Interface Protocol Component)라 부른다.

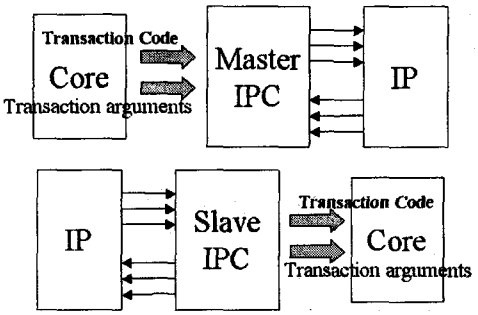
2. IPC

2.1 IPC의 종류

IPC는 마스터 IPC와 슬레이브 IPC의 두 종류가 있다. 마스터 IPC는 트랜잭션 코드와 트랜잭션 전달인자를 입력으로, 해당 신호를 구동하여 IP를 동작시키고, 슬레이브 IPC는 해당 신호를 관찰해서 트랜잭션과 트랜잭션 전달인자를 인식하는 역할을 한

1)이 연구는 BK21충남대학교 정보통신인력양성 지원을 받았음

다.(그림 1)서 보듯이, 마스터 IPC는 Core의 요구에 의해 IP를 구동하는 모듈이고, 슬레이브 IPC는 IP의 동작을 트랜잭션 수준으로 인식해서 코어에 정보를 전달하는 역할을 한다.



(그림 1) Master IPC와 Slave IPC

IPC는 설계자 또는 IP 사용자가 직접 RTL 수준의 VHDL이나 Verilog 모듈로 설계할 수 있다. 그러나 이렇게 설계된 모듈은 트랜잭션 전달인자나 전송 형태가 고전된다. 사용자의 변화하는 요구를 맞추고, 유연성을 갖기 위해, IPC는 좀 더 추상화된 언어로 표현되어야 한다. 그리고 VHDL이나 Verilog 형태로 쉽게 변환할 수 있어야 한다. 본 논문에서 IP의 인터페이스를 기술하기 위해 C 수준의 언어를 제안한다.

2.2 인터페이스 프로토콜 기술 언어

IP의 인터페이스를 기술하기 위해 본 논문에서는 다음의 내용을 가정한다. 첫째, 인터페이스 동작을 사이클 수준에서 기술하고, 둘째, IP는 단일 클럭의 영역에서 동작함을 가정한다. 셋째, 트랜잭션의 전달인자는 (그림 1)에서와 같이 Core 부분에 FIFO나 래지스터에서 입력받음을 가정한다. 인터페이스 기술 언어는 C 언어와 비슷한 형태로 기술한다. 이는 C 언어가 하드웨어 설계자에 친숙하기 때문이다.

(그림 2)는 본 논문에서 제안한 인터페이스 기술 언어로 UTOPIA[3] Transmit 프로토콜을 표현한 예의 일부이다.

```
Interface master  TxMaster {
  out  bit  TxSOC; /* ①interface ports */
  out  bit  TxEnbn;   out  byte TxData;
  in  bit  TxClav;
  reset  rst_n low async; /* ②reset */
  clock TxClk single;  /* ③clock */
transaction Transmit(out FIFO byte Data[53])  {
  int  i, j;   /* ④transaction definition */
    while(TxClav != 1)
     wait_edge(clock, POS);/*⑤cycle boundary*/
    TxEnbn = 0;     TxSOC  = 1;
    Data.delete = '1';  /*⑥ FIFO operation */
    wait_edge(clock, POS);      TxSoC = 0;
    for (i = 1; i <= 52; i++) {
      if(TxFulln == 0)  {
        for (j = i; j < i + 4; j++) {
          Data.delete = '1';/* FIFO operation */
          assert(TxFulln == 0);/*⑦ assertion */
          wait_edge(clock, POS);
        }  wait_edge(clock, POS);
      } TxEnbn = 0;
      Data.delete = '1'; /*  FIFO operation */
      wait_edge(clock, POS);    }
  netlists {        /* ⑧ netlist construction */
    TxData <= Data;  }
}   }
```

(그림 2) UTOPIA transmit 인터페이스 프로토콜 기술

UTOPIA transmit 동작은 8bit의 데이터 53개를 전송하기 위한 프로토콜이다. 첫 줄에 IPC의 종류(마스터/슬레이브)를 기술한다. ①에 IPC의 입출력 포트를 기술하고, ②, ③은 리셋과 클럭에 대해 표현한다. ④는 트랜잭션에 대한 정의이다. 트랜잭션의 이름은 "Transmit"이고, 전달 인자는 FIFO로 정의된 포트로 입력됨을 기술한다. 사용자가 임의의 변수를 정의해서 표현 할 수 있도록 허용했다. ⑤에서 알 수 있듯이, 한 사이클의 경계는 wait_edge() 문으로 표현한다. 임의의 wait_edge() 문에서 다음 wait_edge() 문까지 한 상태로 생성되고, 두 wait_edge()문 사이에 기술된 구문은 전이 동작이 된다. ⑥은 FIFO로 정의된 포트에서 데이터를 하나 읽어옴을 의미한다. 이때 FIFO에서 데이터가 삭제되기 때문에, "delete" 동작으로 기술했다. 트랜잭션 전달인자의 동작 표현은 "." 연산자를 이용해서 이름과 동작을 표현하였다. FIFO의 동작은 {insert, delete, empty, full, count}의 정의하였다. ⑦의 표현은 프로토콜의 오류를 파악할 수 있는 표현식이다. 오류 발생 시에 오류 메시지(신호)를 출력한다. ⑧은 생성된 IPC의 크기를 줄이기 위해 추가되었다. netlists 안에 기술된 할당문은 Concurrent Signal assignment문으로 합성된다.

2.3 IPC의 생성

생성된 IPC는 다음의 네 부분으로 구성된다. 상태 전이 프로세스, 상태 결정 프로세스, 변수처리 프

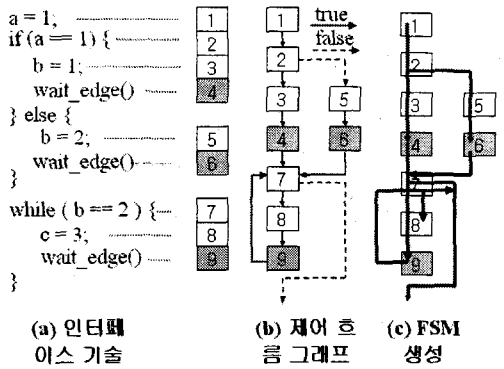
로세스, Concurrent Signal assignment문이다. 상태 전이 프로세스는 순차회로로 구성되며, 각 시그널의 초기값을 설정하고 내부 FSM의 상태를 전이하는 프로세스이다. 상태 결정 프로세스는, 조합회로로 구성되고 입력 신호와 현재 상태에 근거하여 다음 상태를 결정하고, 해당 신호를 구동하는 프로세스이다. 변수처리 프로세스는 사용자 정의 변수를 처리한다. concurrent signal assignment문은 전달인자를 통과(bypass)하는 역할을 한다.

IPC는 2.2절에 기술된 인터페이스 기술 언어를 입력으로 생성된다. 생성 과정은 다음과 같다.

Main()
- Parsing() --- ⓐ
- Make_Control_Flow_Graph() --- ⓑ
  - Avoid_Infinite_Loop_Processing() --- ⓒ
  - Assert_Statement_Processing() --- ⓓ
  - Search_Argument_Variable() --- ⓔ
  - Make_FSM() --- ⓕ
  - Detect_Latch_and_UnexpectedStorage() ⓖ
- Variable_Control_Processing() --- ⓗ
- Generating_IPC() --- ⓘ

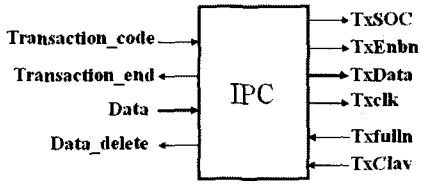
(그림 3) IPC 생성 과정

단계 ⓐ에서는 입력된 인터페이스 기술언어를 정의된 인터페이스의 문법에 맞는지 파악한다. 단계 ⓑ는 파싱된 결과를 근거로 제어 흐름 그래프를 생성한다. 단계 ⓒ는 생성된 그래프를 분석하여 반복문 내에 사이클의 경계를 나타내는 wait_edge()이 기술되어 있는지 조사한다. wait_edge() 문이 없는 경우는, 시간의 지연 없이 계속 반복됨을 의미하기 때문에 무한루프가 발생한다. 이런 경우를 방지하기 위해 그래프를 분석하고 인터페이스 기술 언어가 잘못 기술되었음을 보고한다. 단계 ⓓ에서는 사용자가 기술한 assert문을 VHDL로 변환하기 위해 제어 흐름 그래프를 재구성한다. 단계 ⓔ에서는 인터페이스 기술 언어의 각 문장에 트랜잭션의 전달 인수나 사용자 정의 변수가 정의 되어있는지 찾는다. 이는 변수와 트랜잭션 전달인자를 처리하기 위해, Core 부분의 FIFO나 레지스터를 제어하기 위핸 신호를 생성해야 하기 때문에, 흐름 제어 그래프에 변수 또는 트랜잭션 전달인수가 있음을 표시한다. 단계 ⓕ는 FSM을 생성하는 단계이다. 흐름 제어 그래프의 루트에서 출발하여, wait_edge() 문이 나올 때까지를 하나의 상태로 생성한다. 따라서 모든 흐름 제어 그래프 내의 wait_edge() 노드를 리스트로 연결한 수에, 각 wait_edg() 노드마다 흐름제어 그래프에 따라 상태를 생성한다(그림 4). 단계 ⓖ에서는 생성된 FSM의 각 상태에서 구동하는 신호의 목록을 찾고, 모든 경우에서 신호의 값을 구동하는지 찾아낸다. 만약 모든 조건에서 값을 구동하지 않는 경우가 있다면, 이 신호는 래치가 생성되므로 이를 방지하기 위해, 해당 신호가 Flip-Flop이 되도록 설정한다.



(그림 4) 인터페이스 기술로부터 FSM 생성과정

단계 ⓗ는 변수를 처리하기 위해, 새로운 프로세스를 생성한다. 사용자가 기술한 인터페이스 기술 언어에 근거하여 VHDL 모듈을 생성하면, 사용자가 정의한 변수는 사용자의 의도와 다르게 동작할 수 있다. 즉 임의의 상태에서 시그널에 값을 할당하고 바로 이어 그 할당된 변수를 사용하는 경우, 사용자는 변화된 변수의 값을 가정하고 있는 것이다. 그러나 VHDL 모듈에서 해당 신호를 변수로 처리하게 되면, 래치의 생성을 막을 수 없기에 시그널로 선언하고, 대신 변수의 값 변화에 따른 여러 시그널을 생성하여 사용자의 의도와 같게 동작하도록 구성하였다. 이때 필요한 시그널을 생성하기 위해 또 다른 프로세스가 필요하다. 단계 ⓘ에서는 흐름 제어 그래프를 근거로 VHDL 모듈을 생성한다.



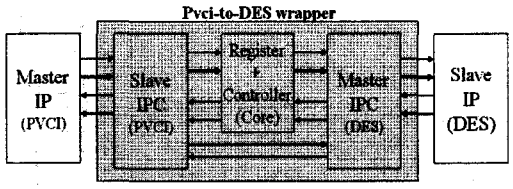
(그림 5) 생성된 마스터 IPC의 입출력 포트

(그림 5)는 생성된 IPC의 입출력 포트이다. 왼편은 Core와 연결되는 포트이고, 오른편은 IP와 연결되는 포트이다. 코어에서 트랜잭션 코드를 받고, 트랜잭션이 끝나면 종료 신호를 보낸다. 트랜잭션 전달인자는 Data 포트를 통해 받고, 그 제어를 위해

"Data_delete"의 포트가 생성되었다. 만약 트랜잭션 전달인자라 레지스터로 정의되었다면, "Data_load"의 신호가 생성되었을 것이다.

3. IPC를 이용한 설계

IPC는 래퍼의 설계에 이용할 수 있다. 래퍼는 IP를 다른 IP와 연결하거나, 시스템 버스에 연결하기 위해 IP의 프로토콜을 변환해 주는 모듈이다.



(그림 6) IPC를 이용한 PVCI-to-DES 래퍼 설계

(그림 6)은 DES 모듈을 PVCI[4] 프로토콜로 변환하는 래퍼이다. PVCI의 요구를 인식하는 슬레이브 IPC와 DES를 구동하는 마스터 IPC가 사용된다. 사용자는 슬레이브 IPC에서 인식한 트랜잭션과, 마스터 IPC를 구동하기 위한 트랜잭션을 연결하고, 전달인자를 저장하기 위한 레지스터와 제어기 부분을 설계한다. 이런 방법을 통해 사용자는 DES의 인터페이스에 대한 이해 없이, DES가 제공하는 트랜잭션의 정보만으로 래퍼를 설계할 수 있다. 따라서 설계 시간을 줄일 수 있다.

3. 실험

(표 1)은 여러 프로토콜의 인터페이스를 인터페이스 기술 언어로 기술한 후에, IPC를 생성한 결과표이다. IPC 생성기는 약 6000라인의 C 프로그램으로 구현되었고, Solaris 2.7 동작 환경에서 테스트하였다.

프로토콜	종류	줄수	FF	크기[1]	크기 2)	크기 2')
DES	Master	44	8	115	62	253
	Slave	55	9	131	64	259
PVCI	Master	52	3	47	24	123
	Slave	57	5	49	31	134
UTOPIA Tx	Master	46	17	292	157	178
	Slave	38	16	204	111	137
UTOPIA Rx	Master	52	11	195	104	112
	Slave	48	11	169	100	108
Wishbone	Master	51	3	45	33	220
	Slave	56	6	46	37	268

Library : 1) HYNIX 0.35 um standard cell 2) Altera Flex 10k

(표 1) IPC 생성 결과

(표 1)에 예제 프로토콜의 이름과 종류, 생성된 크기를 기술하였다. 크기[1]은 HYNIX standard cell을 이용하여 합성한 결과이고, 크기[2]와 크기[2']는 Altera Flex 10K 라이브러리를 사용하였다. 크기[2]와 크기[2']의 차이는 IP의 인터페이스 기술할 때, netlists 구조를 이용해서 기술한 결과와 사용하지 않은 결과의 합성 크기이다. 이 실험의 목적은 생성된 IPC의 크기가 그렇게 크지 않음을 보인다.

래퍼 이름	Non-IPC (전체 면적)	IPC 사용(전체 면적)		
		IPC1(S)	IPC2(M)	Core
PVCI-to-DES	2479	2498		
		PVCI: 49	DES: 115	Core: 2334
60x-to-PVCI	398	463		
		60x : 204	PVCI: 47	Core: 212
PVCI-to-SRAM	154	208		
		PVCI: 49	PVCI: 157	Core: 0

(표 2) IPC를 이용한 래퍼 설계와 Non-IPC의 비교

(표 2)는 IPC를 이용한 설계의 예이다. 이는 IPC를 이용하여 설계한 것과 사용자가 직접 설계한 것과의 크기 비교를 했다. IPC가 전체 면적에서 큰 부분을 차지하지 않는다. PVCI-to-DES의 래퍼의 면적이 큰 이유는 DES가 동작하기 위해 64bit의 key, plaintext 각각이 필요하기 때문에, 그 값을 저장하기 위한 레지스터 때문이다. (표 2)를 통해 알 수 있듯이 프로토콜이 복잡해질수록 IPC의 면적이 그리 크지 않음을 알 수 있다.

4. 결론

본 논문에서는 IP를 재사용 할 때 필요한 인터페이스 관련 작업을 좀더 쉽고 빠르게 하기 위하여, 트랜잭션 수준의 설계를 위한 IPC를 제안하였다. IPC를 생성하기 위한 IP 인터페이스 기술과 IPC의 생성 방법을 설명하고, IPC를 이용한 설계의 예를 통해, 래퍼의 설계에 IPC의 사용이 설계시간을 줄일 수 있도록 해주고, 많은 오버헤드를 차지하지 않음을 보였다. 이렇게 설계된 IPC는 다른 설계에서도 계속 재사용이 가능하다.

참고문헌

[1] W.Wolf and J. Henkel, "Platform-Based Design", Tutorial at DATE 2001, (Munich, March, 2001)

[2] D. S. Brahme, et. al, "Transaction-Based Verification Methodology," Cadence Berkeley Labs, #CDNL-TR2000-0825, Aug., 2000.

[3] TranSwitch Corporation, UTOPIA Interface for the SARA Chipset, Application Node, Document Number TXC-05501-0002-AN, 1.0, Aprial, 1995.