Cycle-C를 이용한 제어흐름 중심의 FSM 설계

(FSM Designs with Control Flow Intensive Cycle-C Descriptions)

윤 창 열 † 장 경 선 ††

(Chang-Ryul Yun) (Kyoung-Son Jhang)

요 약 일반적으로 디지털 시스템에서 시스템의 제어부 설계를 위해 FSM이 많이 사용된다. FSM은 제어흐름(Control Flow)으로부터 생성된 상태 다이어그램에 기반하여 구현된다. 설계자는 상태 다이어그램을 이용하여, HDL로 FSM을 설계하고 검증한다. FSM의 상태의 수가 증가할수록, FSM을 검증하거나 변경하는 작업은 매우 복잡해지고 오류가 많이 발생하며 많은 시간을 필요로 한다. 따라서 본 논문에서는 레지스터 전송 수준에서 제어흐름중심으로 하드웨어를 기술하는 언어인 Cycle-C를 제안한다. Cycle-C는 제어 흐름에 시간 정보를 더하여 FSM을 기술한다. Cycle-C로 표현된 FSM은 합성 가능한 VHDL 코드로 자동으로 변환된다. 실험에서는, 인터페이스 회로들에 대한 FSM을 비교 예제로 삼았다. Cycle-C를 이용한 설계와 설계자가 직접 RTL VHDL로 설계한 것은 비슷한 면적을 보였다. Cycle-C를 이용하면 설계자가 직접 기술한 RTL VHDL 행수의 약 10~50%만으로 동일한 동작에 대한 기술을 할 수 있었다.

키워드 : FSM, Cycle-C, 흐름제어, 제어기, 상태다이어그램

Abstract Generally, we employ FSMs for the design of controllers in digital systems. FSMs are implemented with state diagrams generated from control flow. With HDL, we design and verify FSMs based on state diagrams. As the number of states in the system increases, the verification or modification processes become complicated, error prone and time consuming. In this paper, we propose a control flow oriented hardware description language at the register transfer level called Cycle-C. Cycle-C describes FSMs with timing information and control flow intensive algorithms. The Cycle-C description is automatically converted into FSMs in the form of synthesizable RTL VHDL. In experiments, we design FSMs for control intensive interface circuits. There is little area difference between Cycle-C design and manual design. In addition, Cycle-C design needs only 10~50% of the number lines of manual RTL VHDL designs.

Key words : FSM, Cycle-C, control flow intensive, controller, state diagram

1. 소 개

RTL 수준의 VHDL을 이용한 디지털 설계에는 적어도 하나 이상의 FSM을 포함한다. 디지털 시스템에서 FSM은 시스템 동작을 제어하는데 많이 사용된다. 기존의 FSM 설계방법에서는 상태 테이블을 만들고 이 테이블을 근거로 FSM을 생성하였다. 시스템의 복잡도가 증가함에 따라, 캐드툴에서 제공하는 상태 다이어그램을 이용하는 방법을 사용하여 설계를 하거나 사용자가 직접 HDL(Hardware Description Language)로 설계한다. Xilinx ISE[1]에서 간단한 모듈을 설계할 수 있는 ABEL 언어를 제공한다. ABEL은 간단한 부울식(Boolean equation)의 기술에서 CPLD에 다운로드 할 수 있는 회로를 생성한다. 그러나 기술상 제약이 있어서 많이 사용되지 않는 실정이다. 그림 1은 ISE에서 제공하는 StateCAD를 이용한 예이다. 상태 다이어그램을 이용한 방식은 FSM이 간단할 경우에는 설계와 변경이 용이하다. 그러나 FSM의 상태가 많아짐에 따라 설계자의 요구에 따른 상태의 추가/변경되는 과정은 복잡하고, 연관된 모든 상태를 고려해야 하므로 검증하기 어렵고 오류가 많이 발생한다. 이런 단점들을 극복하기 위해서는 FSM의 설계 수준을 알고리즘 수준의 절차적인 표현으로 높이는 것이 요구된다[2].

많은 하드웨어 모듈 설계를 위한 알고리즘이 C 언어로 기술된다. 이런 알고리즘을 하드웨어로 구현하기 위

· 본 논문은 한국과학재단이 지정한 지역협력연구센터(RRC)인 충남대학교 소프트웨어연구센터의 지원으로 수행된 과제의 결과입니다. 본 논문에 사용된 CAD 툴은 IDEC으로부터 지원받았습니다.

† 비 회 원 : 충남대학교 컴퓨터공학과
yun@ce.cnu.ac.kr
†† 종신회원 : 충남대학교 컴퓨터공학과 교수
ksjhang@computer.org

논문접수 : 2004년 4월 6일
심사완료 : 2004년 10월 8일

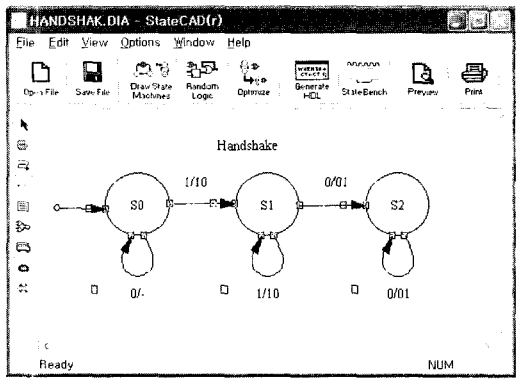


그림 1 핸드쉐이크 프로토콜의 FSM

해서 설계자는 C 언어로 표현된 알고리즘을 정확하게 이해하고 HDL로 변환하게 되나, C 언어를 HDL로 변환하는 과정에서 많은 오류가 발생할 수 있다. 또한 사용자의 새로운 요구에 따라 알고리즘이 변하게 되면, 이는 HDL 기술을 전반적으로 재설계해야 할 수도 있다. 이런 과정은 많은 시간과 노력을 필요로 한다. 이런 어려움들을 극복하기 위해 C 언어에서 HDL로 직접 변환하여 설계 수준을 알고리즘 수준으로 높이려는 노력이 있었다. 특히, 하드웨어-소프트웨어 통합설계를 위한 연구 분야에 C 언어를 VHDL로 변환하는 연구[3-6]들이 있었다. 그러나 앞선 연구들은 C 언어에서 HDL로 자동 변환하려 했지만, 생성되는 HDL의 합성 결과는 좋지 않았다. 또한 behavioral compiler를 이용한 알고리즘 수준의 HDL 기술에서 회로를 합성하는 완전 자동 변환에서는 큰 면적과 동작 속도가 지연되는 합성 결과를 나타낼 수 있다. 알고리즘 수준의 HDL 기술에는 자원 공유를 제어하기 위해 하드웨어 설계자가 제어 단계를 스케줄링 하는 것이 필요하다.

본 논문에서 제안하는 Cycle-C는 기술의 편리성을 제공하고, 합성 결과의 효율을 위해서 생성되는 VHDL의 형태를 정하였으며, 다중 프로세스문의 기술도 허용한다. Cycle-C 내의 프로세스는 VHDL에서의 프로세스문과 같은 의미를 갖는다. 각 프로세스는 C 언어와 비슷한 형태로 동작을 기술하고, 프로세스 내에는 사이클의 경계를 표시하여 시간 제어 개념을 갖는다. 즉 모듈에서 기대하는 동작을 behavioral 수준으로 기술하고 사용자가 스케줄링에 대한 정보를 추가하는 것이다.

본 논문의 구성은 다음과 같다. 2장은 C 언어에서 VHDL 코드를 자동 생성하는 관련 연구에 대한 설명이고, 3장은 본 논문에서 제안하는 Cycle-C를 예시와 함께 설명한다. 4장은 Cycle-C에서 VHDL 모듈로 변환하는 과정을 기술한다. 5장은 Cycle-C를 이용하여 래퍼(Wrapper)를 설계하여 사용자의 설계와 표현의 용이성 및 합성 결과 측면을 비교한다. 또한 Behavioral VHDL 기술과 합성 결과를 비교한다. 6장에서는 요약 및 향후 과제를 논의한다.

2. 관련 연구

C언어에서 합성 가능한 VHDL 코드를 생성하는 연구는 하드웨어-소프트웨어 통합설계 분야에서 많이 연구되었다. 기존의 시스템 설계는 하드웨어와 소프트웨어를 독립적으로 개발하여 통합하는 방식으로 이루어졌으나, 하드웨어와 소프트웨어가 복합된 복잡한 시스템을 체계적이며 효율적으로 설계하기 위하여 하드웨어-소프트웨어 통합설계의 중요성이 부각되고 있다. 이런 하드웨어-소프트웨어의 통합설계의 과정에서 꼭 필요한 알고리즘 수준의 기술에서 하드웨어를 자동으로 생성하는 연구가 진행되었고, 그 중에서 C 언어를 VHDL로 자동으로 변환하는 연구가 있었다. 논문[3]은 알고리즘 수준의 C 언어 기술에서 VHDL 코드를 생성한다. 생성 과정은 먼저 C 언어의 기술에서 변수와 실행 문장을 추출하고, 변수는 포트나 신호가 되고, 실행문장은 VHDL 프로세스 내의 실행문장으로 생성된다. 입출력 포트는 변수에 값이 할당되면 입력포트로 생성되고, 값이 할당되지 않으면 출력포트로 만들어진다. 두 경우를 모두 만족하는 경우에는 포트의 특성을 버퍼로 설정한다. 변수의 크기도 정수형, 부호 없는 정수형에 따라 그 데이타 폭이 간단하게 결정된다. 모듈의 동작은 C언어에서 기술된 순서대로 실행한다. 논문[3]은 하드웨어로 기술하려는 C 코드를 특별한 가공 없이 VHDL로 변환할 수 있으나, 생성되는 하드웨어의 구조를 고려하지 않고 C 언어의 기술대로 VHDL 코드를 생성했기 때문에, 생성된 VHDL의 합성 결과는 그리 좋지 못하다. 또한 변수 선언의 제한과 goto 문을 처리할 수 없는 단점이 있다. 논문[4]은 C 언어로 기술된 알고리즘을 VHDL 프로세스로 변환한다. VHDL 코드의 엔티티와 포트에 대한 고려를 하지 않았고, 동작에 대한 변환만을 포함한다. 엔티티는 사용자가 추후에 직접 작성해야 한다. 논문[4]에서는 알고리즘의 모든 변수 및 상수를 모두 VHDL의 변수로 변환한다. 논문[4]에서는 합성용 코드를 생성하려 하였으나, C 알고리즘에서 기술된 모든 변수는 거의 래치(latch)로 생성될 것이고, 자료형도 C 언어에서 사용되는 자료형만을 허용한다. 정수형 변수의 경우 사용되는 데이타의 크기에 관계없이 항상 32bit의 데이타 크기를 갖게 된다. 따라서 변환된 VHDL 코드의 합성 결과는 그리 좋지 못하였다. 논문[3,4]은 알고리즘의 기술 순서에 따라 V-HDL 코드로 변환 하려 하였기 때문에 VHDL 코드에는 하나의 프로세스가 생성되었다. 그러나 논문[5]에서는 C 알고리즘의 기술에서 병렬 처리가 가능한 부분을 찾고,

병렬 처리가 가능한 부분을 각각의 컴포넌트로 생성하였다. 각각의 컴포넌트가 모여 전체 모듈을 구성하였다. 이는 특정 동작이 반복되는 알고리즘의 경우 최적의 결과를 나타내었다. 앞선 방법보다는 합성 결과가 좋으나, 데이타의 연관성이 많은 경우나 데이타 전송이 많은 경우에는 병렬 처리가 가능한 부분을 찾기 어렵기 때문에, 컴포넌트로 나누기 어렵다는 단점이 있다. 위 세 논문에서 보듯이 C 언어에서 VHDL로의 변환은 그 합성 결과가 좋지 못하거나, 특정 예에서만 좋은 결과를 나타내었다. 따라서 Cycle-C는 생성되는 VHDL코드의 구조를 정하였고, 변수/신호의 크기 최적화 등을 통해 불필요한 자원을 줄여 합성 결과를 좋게 하였고, 사용자가 처리 단계를 설정할 수 있도록 하였으며, 기술의 편의를 위해 C 언어 스타일의 기술을 허용하였고, VHDL의 직접할당문 등의 문법 구조를 허용하여 최적의 합성 결과를 이끌어 내도록 구성하였다.

3. Cycle-C

Cycle-C는 크게 세 부분으로 나뉜다. 코어 부분, 프로세스 부분, 직접 할당문 부분이다. 코어 부분에는 모듈의 입출력 신호, 리셋에 대한 동작(active low/high) 및 이름, 클럭에 대한 이름 및 정보(rising/falling), 프로세스 부분에서 사용되는 신호 선언에 대해 기술된다.

모듈의 입출력 신호에 대한 기술은 프로세스 기술에서 자동으로 추출할 수 있으나, 사용자의 동작에 대한 기술만으로는 내부의 동작 제어를 위한 신호인지, 입출력 포트인지를 알 수 없으므로 불필요한 포트가 생성되지 않도록 하기 위해 사용자가 직접 기술하도록 했다. 프로세스 부분에는 모듈의 동작이 기술된다. 프로세스는 하나 이상이 될 수 있다. 하나의 프로세스는 사용자 변수를 사용 했는지의 여부에 따라, 추후 하나 또는 두 개의 VHDL 프로세스로 생성된다. 이는 해당 변수가 래치(Latch)로 생성되는 것을 방지하기 위함이다. 직접 할당문은 특정 신호를 bypass하거나, 조건에 의해 값을 구동하는 동작을 기술할 수 있다. VHDL기술의 직접 할당문(Concurrent Signal assignment) 부분에서 허용하는 기술을 모두 사용할 수 있다.

프로세스 부분에 기술되는 모듈의 동작은 C 언어와 비슷한 형태로 기술된다. 프로세스 부분에는 모듈에서 기대되는 동작의 기술과 사이클 수준의 시간정보를 포함하고 있다. 사이클 수준의 시간정보는 클럭의 경계를 의미한다. 즉, 모듈의 동작에 대한 기술들 간의 시간적인 순서 관계를 표시하는 것이다. 클럭의 경계는 wait_edge() 문으로 표시하며, 하나의 wait_edge() 문에서 다음에 올 수 있는 wait_edge() 사이의 동작 기술은 한 클럭 동안 수행된다. 물론, 두 wait_edge() 문 사이의 기술에도 문장간의 순서관계는 유지된다. Cycle-C에서 제공하는 연산자는 일반적인 C 언어에서 제공하는 연산자를 허용하되, 함수, 포인터, 구조체 등은 지원되지 않는다. bit 수준의 동작은 VHDL에서 허용하는 문법을 따른다. 즉 and, or, xor 등의 연산자 사용이 허용된다. 또한 연결 연산자(Concatenation)도 허용된다.

표 1 Cycle-C 제공 연산자

연산자	동작	연산자	동작
~	Not	and	And
++	Increase	or	Or
	Decrease	nand	Nand
*, /	Multiplier Divider	nor	Nor
<, >, <, ==, >=	Compare	xor	Xor
==, !=	Equal, not equal	xnor	Xnor
&&, ||	Compare	&	Concatenation
=	Assignment		

3.1 간단한 FSM 설계 예

그림 1은 간단한 핸드쉐이크 프로토콜이다. 입력 값이 '0'이면 S0 상태를 유지하고, '1'이 입력되면 S1 상태로 전이 한다. 다시 '0'이 입력되면 S2 상태로 전이하는 FSM이다. 그림 2는 그림 1의 핸드쉐이크 프로토콜을 Cycle-C와 동일 FSM의 VHDL 기술이다. Cycle-C의 기술은 "Core"(그림 2ⓐ)라는 예약어로 시작한다. 코어 부분은 입출력 포트와 클럭/리셋 신호에 대한 기술이다. 출력포트는 "value"라는 이름을 갖고, 2bit의 크기를 갖는다. 입력 포트는 "cond"의 이름이고, 1bit의 크기이다(그림 2ⓑ). 자료형은 bit이외에도 byte를 사용할 수 있고, byte는 8bit의 크기이다. bit는 추후 VHDL의 'std_logic'이나 'std_logic_vector'로 변환된다. 또한 C 언어에서 제공하는 정수형인 int 형도 허용한다. 클럭의 이름은 "clk"이고 rising_edge에서 FSM의 상태가 전이함을 의미한다(그림 2ⓒ). 프로세스의 부분에는 모듈의 동작을 기술한다. "process" 다음에는 프로세스의 입출력 포트/신호들을 기술한다. ":"을 중심으로 앞에는 프로세스의 입력 포트/신호가 나열되고, 뒤에는 프로세스의 출력 포트/신호가 기술된다(그림 2ⓓ). 일반적인 VHDL 기술에서 "process"문의 감지신호 목록과는 다른 의미를 갖는다. Cycle-C에는 여러 프로세스가 기술 될 수 있으므로, 해당 신호가 어떤 프로세스에서 구동되는지를 사용자가 명확하게 하기 위해서 이러한 구문을 추가했다. 하나의 신호를 여러 프로세스에서 할당하면, 오류가 발생 할 수 있기 때문이다. 생성되는 VHDL의 해당 프로세스의 감지신호에 사용자가 기술한 입력 신호들이 포함된다.

프로세스 기술 내부의 동작은 Cond 신호가 '1'이 될

```
entity handshake is
  Port ( clk, cond, reset : in std_logic;
       value : out std_logic_vector(1 downto 0) );
end handshake;
architecture Behavioral of handshake is
          type StateType is (S0, S1, S2);
          signal CS, NS          : StateType;
begin
process(clk, reset)    begin
          if(reset = '0') then                CS <= S0;
          elsif rising_edge(clock) then       CS <= NS;
          end if;
end process;
process(clk, cond, reset, CS)    begin
 case CS is
   when S0 =>
          if (cond = '1') then
            NS <= S1;              value <= "10";
          else        NS <= S0;
          end if;
   when S1 =>
          if (cond = '1') then
            NS <= S1;              value <= "10";
          else        NS <= S2;    value <= "01";
          end if;
   when S2 =>
          if (cond = '0') then
            NS <= S2;              value <= "01";
          end if;
  end case;
end process;
end Behavioral;
```

(a) VHDL 기술

```
Core handshake {    /* ⓐ 모듈의 이름 */
      out bit[1:0] value;    in bit cond;
      /* ⓑ 입출력 포트 기술 */
      clock clk rising; /* ⓒ 동작 클럭 정보 기술 */
      reset reset low;
 process(cond : value)  /* ⓓ 프로세스 기술 */ {
    while (cond == '0')   wait_edge(); // cond 가 '0'일 때까지
    value = "10";
    wait_edge();
     while (cond == '1')   wait_edge();
     value = "01";
     while (cond == '0')   wait_edge();
  }
}
```

(b) Cycle-C 기술

그림 2 핸드쉐이크 프로토콜의 VHDL 및 Cycle-C 기술 비교

때까지 기다린 후, '1'이 되면 다음 상태로 전이한다. 전이하면서 value에 "10"을 구동한다. 다시 cond의 값이 '1'이 되기를 기다리고, 그 후에는 '0'이 되기를 기다린다. Cycle C의 기술은 모듈의 동작을 사용자가 쉽게 이해할 수 있고, 또한 상태의 추가/변경이 되더라도 알고리즘 수준의 기술이기 때문에 쉽게 대처할 수 있다.

3.2 UTOPIA 전송 프로토콜

그림 4는 UTOPIA 전송 프로토콜[7]을 Cycle-C로 기술한 예이다. 이 프로토콜은 octet level 핸드쉐이크를 사용하여 UTOPIA의 ATM에서 PHY로 53개의 8bit 데이타를 전송한다.

그림 3처럼 UTOPIA_TX 코어는 제어 신호(data_delete)

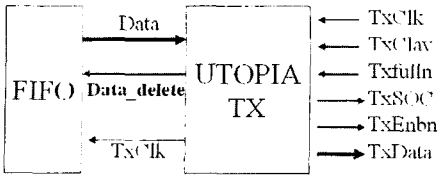


그림 3 UTOPIA전송 모듈의 블록다이어그램

를 통해서 외부의 FIFO에서 데이타를 보내거나 받는 것을 가정했다. Cycle-C의 기술은 입력과 출력 포트를 기술하고, 이어 reset과 clock에 대해 기술한다(그림 4의 ②,③). 사용자는 신호를 선언하여 사용할 수 있다. 자료형은 bit, byte, int형을 허용한다. 정수형의 자료형 선언에 VHDL에서처럼 범위를 지정할 수 있다(예: signal int count range 0 to 9). 만약 범위를 설정하지 않으면, 사용자의 기술에서 해당 변수에 대한 정보를 찾아, 적절한 범위를 설정한다. 그림 4의 ①에서 시그널 또는 출력 포트(TxSoc, Txenbn, data_delete)는 할당 연산자"="를 이용해 기본값을 설정 할 수 있다. 기본값이 설정된 신호나 포트들은 조합회로로 처리된다. 예로 VHDL 코드 생성 과정에서 생성된 임의의 상태에서 해당 신호에 값이 할당되지 않으면 기본값이 할당된다. 예로 TxSoC에 기본값이 0이 할당되어 있고, TxSoC의 값이 할당되지 않은 상태에서는 TxSoC의 값은 0이 된다. 기본값이 할당되지 않은 다른 신호/포트들은 순차 신호로 여겨지고, 따라서 신호나 포트를 구현하기 위해 합성 후 해당 신호는 레지스터로 생성된다. 모듈의 동작은 여러 프로세스 문으로 기술될 수 있다. 각 프로세스는 해당

```
Core Utopia_Tx {
    out bit TxSOC='0'; /* ① ports */
    out bit TxEnbn='1';
    out byte TxData;
    out bit Data_delete = '1';
    in byte Data;
    in bit TxFulln;

    reset rst_n low; /* ②reset */
    clock TxClk rising; /* ③clock */

    Uto: process(TxFulln : TxSOC, TxEnbn, TxData, Data_delete, Data )
    {    /* ④ algorithm description */
        int count, count4;
        bit pTxFulln;        /* ⑤ variable declarations */

        TxEnbn = '1';
        while(TxFulln == '0') {
            wait_edge();          /* ⑥ cycle boundary */
        }
        TxEnbn = '0';
        TxSOC = '1';
        Data_delete = '1';        /* ⑦ FIFO operation */
        count = 52;
        pTxFulln = '1';
        wait_edge();
        TxSoC = '0';
        while (count != 0 ) {
            TxEnbn = '0';
            Data_delete ='1';
            if(TxFulln == '0') {
                if(pTxFulln == '1' ) {
                    count4 = 3;      count--;
                } else if(count4 != 0) {
                    count4--;        count--;
                } else {
                    Data_delete = '0'; /* do not send data */   TxEnbn = '1';
                }
            } else {
                count--;
            }
        pTxFulln = TxFulln;
        wait_edge();
    } /* while */        } /* process */
    netlists {          /* ⑧ netlist construct */
        TxData = Data;
    }
}
```

그림 4 UTOPIA전송 프로토콜

프로세스의 입력/출력 신호의 목록을 갖는다. 입력 신호 목록 다음에는 ":"를 표시하고, 프로세스 내의 문장에서 LHS(left hand side)에 오는 출력 신호의 목록을 기술한다. 이는 해당 프로세스에서 구동하는 신호를 규정한다. 프로세스 내의 동작 기술은 C 언어에서 제공하는 while if for 등의 제어 구조를 동일하게 제공한다. 이런 특징은 하드웨어 설계자가 Cycle-C를 사용하여 C 언어로 기술된 알고리즘을 쉽게 변환할 수 있도록 해준다. 변수 선언은 프로세스 블록 내에서 허용된다(⑤). 사이클 경계는 wait_edge() 문(⑥)으로 표시한다. 하드웨어 설계자는 적절하게 wait_edge() 문을 삽입하여 시간의 흐름을 구분하고, 구분된 wait_edge()문 사이가 추후에 하나의 상태로 생성된다. 따라서 하나의 wait_edge()와 다음 wait_edge() 문 사이의 문장들은 한 사이클에서 실행된다. 그림 4의 처음 while 문은 PHY 레이어가 octet-level 핸드쉐이크 신호인 TxFulln을 구동할 때까지 기다림을 의미한다. 이것은 ATM 레이어에서 데이타를 받을 준비가 되었음을 의미한다. 두번째 while 문장은 count값을 비교하면서 모든 데이타를 보낼 때까지 과정을 반복한다. 전달 인수 값은 FIFO에서 값을 읽어온다(⑦).

⑧의 'netlists' 구조는 신호의 직접연결을 허용한다. 이 구조는 신호들간의 간단한 연결을 기술하기에 적절하지 않은 프로세스 구조상의 제약을 보완한다. 대부분의 경우 'netlists' 구조는 하드웨어 기술을 간단하게 하고, 면적을 줄이는데 기여한다. 'netlists' 안의 할당문은

VHDL에서의 직접할당문(Cuncurrent Signal Assignment)으로 변환한다.

4. Cycle-C로부터 RTL VHDL 코드 생성

C언어에서 VHDL 코드를 생성하는 기존의 방법들은 합성 결과가 좋지 않았다. 생성되는 하드웨어의 형태를 고려하지 않고, 단순히 C 언어가 담고 있는 의미를 VHDL로 변환하려 하였기 때문이다. 또한 정수형 변수의 경우 32bit의 크기를 갖는 signal로 생성되어 불필요한 많은 부분이 생성되었다. 따라서 본 논문은 생성되는 VHDL 코드의 구조를 정하고, 그 구조에 따라 VHDL 코드가 생성되도록 하였다. 그림 5는 생성되는 VHDL 코드의 구조이다. 순차회로 부분에는 신호가 레지스터(Flip-Flop)으로 합성되게 하여 클럭에 의해 동작하도록 구성하고, 조합회로는 입력 조건에 따라 출력과 다음 전이 상태를 결정하도록 구성하였다. 변수처리 부분은 사용자가 선언한 변수를 위해 임시 신호를 생성한다. 이런 구조는 불필요한 래치의 생성을 막고, 조합회로, 순차회로의 특성을 모두 나타낼 수 있는 장점이 있다.

그림 6은 FSM 생성을 위한 각 단계이다. 단계 ⓐ는 사용자의 기술을 토큰으로 분리하고, 분리된 토큰을 입력 받아 파싱한다. 이때 입력된 사용자의 Cycle-C 기술의 문법상 오류를 찾는다. 사용자가 선언한 포트/신호/변수에 대한 테이블을 생성한다. 이 테이블 정보를 이용하여, 프로세스에 선언되지 않은 포트이름/변수/신호의 사용에 대한 오류를 알린다. 파싱한 결과에서 제어흐름그래프를 생성(단계 ⓑ)한다. 사용자가 기술한 문장이 하나의 노드로 생성되고, 노드 내에는 사용자의 표현식이 트리로 구성된다. 이 노드들이 제어흐름그래프를 구성한다. 또한 해당 노드에 포트, 신호, 변수가 노드에서 사용되었는지에 대한 정보를 그래프에 추가 저장한다. 입력된 기술 중 "assert" 문을 비교 구문(if문)으로 변경한다. "assert" 문은 사용자가 프로토콜의 오류를 검증할 수 있는 기능을 한다. 조건에 따라 동작을 수행하지는

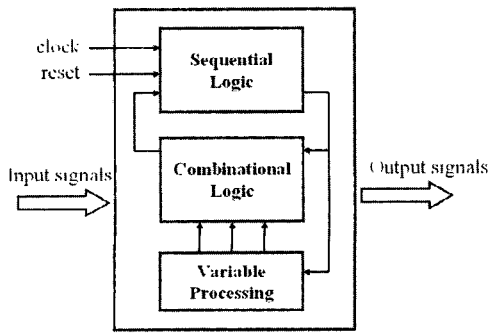


그림 5 생성된 VHDL 코드의 Block Diagram

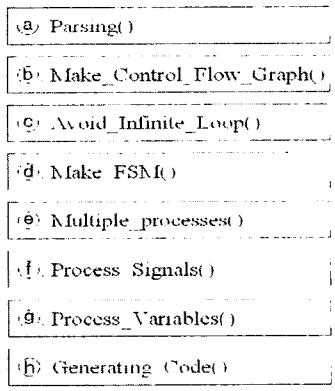


그림 6 RTL VHDL 생성과정

않지만, 프로토콜상 해당 값을 검증해야 할 필요가 있을 경우에 사용한다. 예로, "assert(ack='1');"은 해당 사이클에서 ack 신호의 값이 '1'인지를 조사하고, 만약 조건에 맞지 않는다면, 프로토콜 오류를 보고하는 기능을 추가할 수 있고, FSM은 초기 상태로 전이한다.

그림 7은 제어흐름그래프의 생성 예이다. 그림 7(b)의 제어흐름그래프가 7(a)의 Cycle-C의 기술에서 생성된다. 노드 내의 표현식(예: t = 1;)은 각각의 토큰이 트리로 연결된다. FSM 생성과정의 나머지 단계들도 제어흐름그래프를 사용한다. FSM의 상태는 wait_edge()문 단위로 상태를 생성하기 때문에 반복문(while 문, for 문 등)에 wait_edge()문이 없으면, 생성되는 그래프에 무한 반복의 경로가 생성된다. 따라서 제어흐름그래프에 wait_edge() 문 없는 무한 반복 경로가 있는지를 찾고, 이에 대한 오류를 알린다(단계 ⓒ). 다음 과정은 제어흐름그래프로부터 FSM을 생성한다(단계 ⓓ). 하나의 상태는 wait_edge() 문을 기준으로 생성된다. 상태의 생성 과정이 그림 8에 나타나 있다.

그림 8에서 4,6,9번 노드가 wait_edge() 문을 나타낸다. wait_edge() 다음에 나오는 노드에서부터 그 노드에

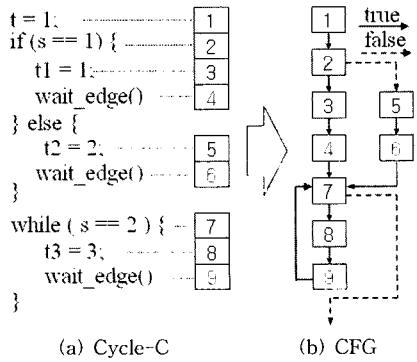
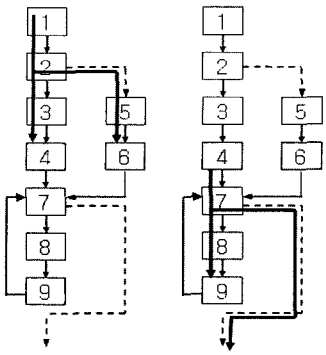


그림 7 제어흐름그래프의 생성



(a) Cycle-C (b) CFG

그림 8 FSM의 상태 생성

서 갈 수 있는 모든 노드가 하나의 상태로 생성된다. 따라서 그림 8에서는 초기 노드에서 오는 상태(State 0)와 4번, 6번, 9번 다음에 생성되는 4 상태가 생성된다. 그러나 4, 6, 9번 노드에 의해 생성되는 상태는 중복된 상태이므로, 4번 노드에 대한 상태(State 1)만 생성하고, 생성되지 않는 상태로 전이되는 링크를 4번 노드에 의해 생성된 상태로 전이하도록 변경한다. 이런 중복되는 상태를 줄임으로써 합성된 면적의 크기를 줄일 수 있다. 그림 9는 중복되는 상태를 제거하는 상태 생성 알고리즘이다.

그림9의 상태 생성 알고리즘에서는 모든 wait_edge() 노드를 검색하여, 동일 상태 인지를 비교하고, 해당 상태로 전이하는 링크를 이전에 생성된 동일 상태로 변경한다. 여러 개의 프로세스가 기술된 경우 제어흐름그래프는 프로세스마다 각각 생성된다. 그에 따라 생성되는 VHDL 코드에도 여러 프로세스가 생성된다. 이 경우에 프로세스간 중첩될 수 있는 요소를 제거하는 단계가 필요하다. 코드 생성시에 이를 고려할 수 있도록 제어흐름그래프에 각종 정보(프로세스 번호, 변수/신호 포함 여부 등)를 추가한다. 각 프로세스에서 생성되는 상태의 이름을 다르게 하고, 각 신호들이 어떤 프로세스에서 구동되는지를 검사하고, 해당 신호에 구동되는 프로세스 정보를 입력한다. 또한 하나의 신호를 여러 프로세스에서 구동하면, 해당 신호를 여러 프로세스에서 구동하는 것에 대한 경고 메시지를 출력한다(단계 ⓔ). 신호 또는 출력 포트는 해당 값들이 기본값을 갖는지에 따라 조합회로 또는 순차회로로 분리 된다(단계 ⓕ). 사용자가 선언한 변수는 VHDL에서의 변수의 의미를 갖는다. 이런 변수들은 합성 후 래치가 생성될 수 있다. 만약 변수의 사용에 저장공간이 필요하다면, 래치가 생성되지 않고 F/F로 생성되도록 몇몇 임시 신호로 변환된다. 이런 처리를 위해 조합프로세스(combinational VHDL process)가 생성된다(단계 ⓖ). 단계 f와 g에서는 사용자가

```
make_state(node) {  // CFG search
if (node != NULL )        {
        if (node.type == WAIT_STATEMENT)   {
            compare_state(node); // 중복 상태 비교
        }
        make_state(node.true); // Next node of true
        make_state(node.false); // Next node of false
    }
}

compare_state(node) {
    for all temp ∈ Q do   // Q : set of state
        if (temp.true == node.true )   //true
        {              // 상태의 중복 → 중복 상태 제거
            for all q ∈ NODE do  // NODE : CFG를 구성하는 모든 노드
                if (q.true == node.true)
                  q.true = temp;   // node.true 의 링크 변경
                if (q.false == node.true)
                  q.false = temp;  // node.false 의 링크 변경
            end for
            return;  // 상태 삭제 후 빠져나감
        }
    end for
    // 중복 상태 없음
    insert_new_state(node); // Q에 새로운 상태 추가
    return;
}

State_generation() {   // 상태 생성 함수
    Q = Ø; // 상태 집합 초기화
    make_state(Root_node); // Root_node : CFG 의 root
}
```

그림 9 상태 생성 알고리즘

선언한 정수형 변수/신호에 대해 만약 사용자가 정수형으로 선언하고 범위를 지정하지 않았다면, 두 과정을 거치면서 생성되는 변수의 범위를 결정하여 합성되는 회로를 최적화 한다. 마지막으로 앞선 단계에서 수집한 정보를 바탕으로 합성 가능한 RTL VHDL 코드를 생성한다. 그림 4의 Cycle-C의 기술에서 생성된 VHDL 코드의 일부가 그림 10이다. 3개의 프로세스 문으로 생성되었다. 하나는 순차프로세스(그림 10ⓐ)이고, 다른 하나는 출력과 다음 상태 결정을 위한 조합프로세스(그림 10 ⓑ), 다른 하나는 변수 처리를 위한 조합프로세스이다(그림 10ⓒ). 'netlist' 구조에서의 할당은 직접할당문으로 생성되었다(그림 10ⓓ).

5. 실험 결과

Cycle-C에서 VHDL 코드를 생성하는 프로그램은 7700 라인의 자바 코드로 구현되었고, 파서는 JavaCC[8] 코드로 만들어졌다. 그림 11은 프로그램을 실행 시킨 화면이다. "source File"에 변환하려 하는 Cycle-C의 기술을 선택하고, "Destination Directory"에는 생성될 VHDL 코드의 저장 위치를 지정한다. "변환" 버튼을 누르면, VHDL 코드가 생성되고, "종료" 버튼을 누르면 프로그램이 종료된다.

표 2는 사용자 설계와 Cycle-C를 이용한 설계의 기술 행 수, 합성 면적, FSM의 상태 수 등을 비교한 표이다. 1개의 FSM으로 구성되는 예인 DES, PVCI[9], UTO PIA, WISHBORN[10]의 설계는 각 모듈로부터

```
entity Utopia_Tx_test is
   port ( TxClk   : in std_logic; /* ports */
      rst_n    : in std_logic;
      Data     : in std_logic_vector(7 downto 0);
      TxFulln  : in std_logic;       . . .      );
end Utopia_Tx_test;

architecture RTL of Utopia_Tx_test is
      type StateType is ( S0, S1, S2, S3);
      signal CS, NS      : StateType;
      signal count : integer range 0 to 63;
      signal count_c : integer range 0 to 63;
      signal count_n: integer;
      . . .
begin
process(TxClk, rst_n)  /* clocked process @*/
begin
      if(rst_n = '1') then
            CS <= S0;
            count_c <= 0;
            . . .
      elsif rising_edge(TxClk) then
            CS <= NS;
            count_c <= count_n;
            . . .
      end if;
end process;
process (TxFulln, TxEnbn_c, count60, pTxFulln88, count4104, CS) /* next state ⓑ*/
begin
    case CS is
      when S0 =>
          TxEnbn_n <= '1' ;
          TxSOC <= '0' ;
         if (TxFulln  = '1'  ) then
           NS <= S1;
         else
           TxEnbn_n <= '0' ;
           TxSOC <= '1' ;
           Data_delete <= '1' ;
           NS <= S2;
         end if;
           :
           :
end process;
process(CS , count, count4, pTxFulln, TxEnbn_c, Data, TxFulln )  /* variable processing ⓒ*/
   variable tmp_count : integer range 0 to 63;
   variable tmp_count4 : integer range 0 to 7;
   variable tmp_pTxFulln : std_logic;begin
   tmp_count := count;
   tmp_count4 := count4;
    case CS is
      when S0 =>
          tmp_count          := 0;
          tmp_count4         := 0;
           if (TxFulln  = '1'  ) then
           else
           end if;
           :
           :
   n_count <= tmp_count;
   n_count4 <= tmp_count4;
end process;
          /* Concurrent Signal assignment @*/
TxData <= Data;
end RTL;
```

그림 10 그림 3의 Cycle-C의 기술에서 자동 생성된 VHDL코드의 일부

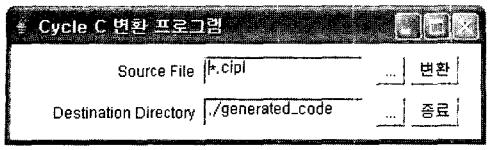


그림 11 프로그램 실행 화면

데이타를 보내거나 받는 인터페이스 FSM이다. Master와 Slave는 마스터 인터페이스와 슬레이브 인터페이스를 의미한다. 마스터 인터페이스는 IP를 구동하는 모듈이고, 슬레이브 인터페이스는 IP의 프로토콜을 인식하여 데이타를 추출하는 역할을 한다. 2개의 FSM으로 구성되어 있는 예는 60x 버스 프로토콜[11]을 PVCI로 변환해 주는 모듈과, AMBA 버스[12]의 AHB 프로토콜을 PVCI로 변환해 주는 모듈이다. 합성 면적은 타겟 라이브러리를 Hynix 0.35 표준셀을 이용한 Synopsys Design Compiler에서의 합성 결과이다.

표 3은 Cycle-C의 기술과 Behavioral VHDL과의 비교표이다. Behavioral VHDL은 알고리즘 수준의 기술을 허용하기 때문에 Cycle-C와 비슷한 기능을 제공한다. Cycle-C는 Behavioral VHDL 기술에 비해 약 50%의 기술로 동일한 동작을 기술 할 수 있었고, 합성 면적에서는, 설계에 따라서 더 좋은 합성 결과를 보였다. 조합회로 영역은 Cycle-C를 이용한 합성 결과가 더 크게 나타났으나, 순차회로 부분은 Behavioral VHDL 기술이 더 크게 나타났다. 이는 Cycle-C에서는 신호나 출력 포트를 위해 디폴트 값을 허용하여, 해당 신호를 조합회로로 만들 수 있으나, Behavioral VHDL 기술에서는 해당 신호가 순차회로로 합성되었기 때문이다.

6. 요약 및 향후 과제

본 논문에서는 제어흐름의 알고리즘 기술에 시간 정보를 포함하여 제어회로를 기술하는 Cycle-C를 소개하였다. Cycle-C를 이용하면 상태의 연관성과 상태 수에 대한 고려 없이 제어부를 설계할 수 있으므로, 추후 설계 변경이나 검증의 노력을 줄일 수 있다. Cycle-C는 C언어와 문법적 동일성의 장점을 갖기 때문에, C 언어로 기술된 하드웨어 알고리즘을 쉽게 Cycle-C로 변환할 수 있는 장점이 있다. 설계의 종류에 따라 다르지만, Cycle-C를 이용하면 사용자 설계의 약 10~50% 미만의 기술로 동일 기능을 수행하는 FSM을 설계할 수 있다. 실험에서는 Cycle-C를 이용한 설계와 사용자 설계가 거의 비슷한 면적으로 합성 되었음을 보였다. 또한

표 2 Cycle-C를 이용한 인터페이스 설계 비교

FSM 수	이 름	Data Bus 크기	사용자 설계		Cycle-C 설계		FSM 상태 수	A'/A (%)	B'/B (%)
			기술 행 수 (A)	합성 면적 (B)	기술 행 수 (A')	합성 면적 (B')			
FSM 1개	Des (Master)	64	212	113	40	118	10	19	104
	Des (Slave)	64	303	125	51	130	12	17	104
	PVCI (Master)	32	151	50	48	50	3	32	100
	PVCI (Slave)	32	234	59	53	59	3	23	100
	UTOPIA-TX (Master)	8	409	170	42	176	12	10	103
	UTOPIA TX (Slave)	8	104	185	34	190	8	9	102
	UTOPIA-RX (Master)	8	238	315	48	325	14	17	103
	UTOPIA RX (Slave)	8	287	198	44	204	20	16	103
	Wishbone (Master)	32	156	241	47	246	3	30	102
	Wishbone (Slave)	32	258	326	42	332	5	17	101
FSM 2개	Xbus To Pvci	32→32	174	770	80	802	9	50	104
	Xbus To Pvci	32→16	194	1047	90	1055	11	46	101
	AHB To Pvci	32→32	286	571	135	584	5	47	102
	AHB To Pvci	32→16	312	941	160	979	6	51	104

표 3 Behavioral VHDL 설계와 Cycle-C 설계 비교표

이름	Behavioral VHDL		Cycle-C 설계		A'/A (%)	B/B' (%)
	기술 줄 수 (A)	합성 면적 (B)	기술 줄 수 (A')	합성 면적 (B')		
Des Master	121	144.8	51	134.6	42	107.5
Des Slave	89	163	49	107.6	55	151
Multiplier (Controller)	60	73	29	28	48	260
XbustoPVCI	135	878	69	798	51	110
Utopia Tx Master	85	189	52	172.2	61	109.7

동일한 기능을 Behavioral VHDL로 기술하고, Synopsys BC로 합성한 결과와 Cycle-C 기술의 합성 결과를 비교하는 실험을 통해 생성되는 레지스터의 수를 줄일 수 있음을 보였다.

향후 과제로는 C 언어에서 제공하는 함수, 구조체 등을 Cycle-C에서도 허용하도록 기능을 추가할 것이다. Cycle-C로부터 생성되는 FSM의 구조가 고정되어 있지만, 이 구조가 모든 설계에 최적의 구조가 아니므로 생성되는 모듈의 구조도 사용자의 의도에 따라 순차회로, 조합회로, 순차회로와 조합회로의 혼합 형태 등을 사용자가 선택할 수 있도록 확장할 계획이다. 생성되는 FSM의 구조를 분석하여, 최적화 방법에 대한 연구도 진행할 예정이다.

참 고 문 헌

[1] http://www.xilinx.com

[2] De Michell, G., Gupta, R.K., "Hardware/software co-design," Proceedings of the IEEE, Volume : 85 Issue : 3, Mar 1997, pp. 349-365.

[3] Sankaran, S., Haggard, R.L., "A convenient methodology for efficient translation of C to VHDL," Southeastern Symposium on System Theory, 2001. Proceedings of the 33rd, Mar 2001, pp. 203-207.

[4] Matthew F. Parkinson, Paul M. Taylor and Sri Parameswaran, C to VHDL Converter in a Codesign Environment, VHDL International Users Forum. Spring Conference, 1994. Proceedings of, 1-4 May 1994, pp. 100-109.

[5] Jie Chen : Haggard, R.L., Extraction of parallel hardware during C to VHDL translation, System Theory, 2002. Proceedings of the Thirty-Fourth Southeastern Symposium on, 18-19 March 2002, pp. 334-338.

[6] Mark Genoe, Paul Vanoostende, Geert van Waewe, "On the use of VHDL-based behavioral synthesis for telecom ASIC design," System Synthesis, 1995., Proceedings of the Eighth International Symposium on, 13-15 Sep 1995, pp. 96-101.

[7] TranSwitch Corporation, UTOPIA Interface for the SARA Chipset, Application Note, Document Number TXC-05501-0002-AN, 1.0, 4/11/95.

[8] http://www.suntest.com, "JavaCC Document".

[9] VSI AllianceTM Virtual Component Interface Standard Version 2(OCB 2 2.0) On-Chip Bus Development Working Group, April 2001.

[10] http://www.opencores.org

[11] PowerPC Microprocessor Family : The Bus Interface for 32-Bit Microprocessors (REV.0).

[12] AMBATM Specification (Rev 2.0).

윤 창 열

2000년 한남대학교 컴퓨터공학과 졸업(학사). 2002년 한남대학교 대학원 컴퓨터공학과 졸업(석사). 2002년~현재 충남대학교 대학원 컴퓨터공학과 박사과정 관심분야는 설계자동화, 인터페이스 자동합성, 시스템온칩 설계

장 경 선

1986년 서울대학교 전자계산기공학과 졸업(학사). 1988년 서울대학교 대학원 컴퓨터공학과 졸업(석사). 1995년 서울대학교 대학원 컴퓨터공학과 졸업(박사). 1996년 3월~2001년 8월 한남대학교 컴퓨터공학과 부교수. 2001년 9월~현재 충남대학교 컴퓨터공학과 부교수 관심분야는 컴퓨터 구조, 설계자동화, 시스템온칩 설계