디지털회로설계 HW2

VHDL에 관한 조사: Verilog와의 비교와 함께

20161453 전자공학과

김규래

디지털회로업계에서는 주로 VHDL, Verilog 등의 HDL을 사용하고 있다. 현재 시점에서는 전자의 업계 점유율이 더 높다[[1]](#footnote-1). VHDL과 Verilog는 같은 목적을 위해 등장한 언어이지만 언어의 설계 자체는 상이하며 사용되는 영역에도 어느 정도 차이가 있다. 하지만 최근 들어 Verilog를 통해 디지털회로설계를 하는 학교들이 증가하고 있다. VHDL을 Verilog와 비교해가며 HDL로서 어떠한 특징들이 있는지를 조사하였다.

1. Basic Syntax

VHDL과 Verilog는 기본 문법부터 상당한 차이를 보고 있다. VHDL은 Ada의 문법에서 영향을 주로 받았고, Verilog는 C계열 언어들의 문법 형태에서 영향을 받았기 때문이다. 현대의 개발자들의 대부분은 C계열 언어들의 문법에 더 익숙한 경향이 있기 때문에 Ada 스타일의 문법은 VHDL을 많은 사람들이 기피하게 되는 원인 중에 하나라고 할 수 있다. 또한 Ada 스타일의 문법과 강타입 성향은 코드가 Verilog에 비해 길어지고 더 많은 단어와 키워드들을 사용하게 되는 현상을 가져온다.

<Figure 1>과 <Figure 2> 는 동일한 코드를 각각 VHDL과 Verilog로 구현한 것이다. 둘을 비교하면 알겠지만 Verilog의 코드가 조금 더 단순하며, 신호 선언이 C계열 언어들의 변수선언과 매우 유사한 모습이다. 그리고 모듈을 선언하는 방법도 C계열 언어들에서 함수를 선언하는 것과 비슷하다.

1. Simulators

현재 시장에는 Cadence 와 Xilinx사의 다양한 시뮬레이션들이 제일 흔하게 쓰이고 있다. Cadence와 Xilinx는 개발, 시뮬레이션, Synthesis를 통합된 환경으로 제공하기 때문에 이들이 제일 흔히 사용되고 있는 것을 보인다.

Figure

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.std\_logic\_arith.all;

use ieee.std\_logic\_unsigned.all;

use work.classic\_multiplier\_parameters.all;

entity poly\_multiplier is

port (

a, b: in std\_logic\_vector(M-1 downto 0);

d: out std\_logic\_vector(2\*M-2 downto 0)

);

end poly\_multiplier;

architecture simple of poly\_multiplier is

type matrix\_ands is array (0 to 2\*M-2) of STD\_LOGIC\_VECTOR(2\*M-2 downto 0);

signal a\_by\_b: matrix\_ands;

signal c: std\_logic\_vector(2\*M-2 downto 0);

begin

gen\_ands: for k in 0 to M-1 generate

l1: for i in 0 to k generate

a\_by\_b(k)(i) <= A(i) and B(k-i);

end generate;

end generate;

gen\_ands2: for k in M to 2\*M-2 generate

l2: for i in k to 2\*M-2 generate

a\_by\_b(k)(i) <= A(k-i+(M-1)) and B(i-(M-1));

end generate;

end generate;

d(0) <= a\_by\_b(0)(0);

gen\_xors: for k in 1 to 2\*M-2 generate

l3: process(a\_by\_b(k),c(k))

variable aux: std\_logic;

begin

if (k < M) then

aux := a\_by\_b(k)(0);

for i in 1 to k loop aux := a\_by\_b(k)(i) xor aux; end loop;

else

aux := a\_by\_b(k)(k);

for i in k+1 to 2\*M-2 loop aux := a\_by\_b(k)(i) xor aux; end loop;

end if;

d(k) <= aux;

end process;

end generate;

end simple;

나의 경우 Synthesis 등이 필요하지 않기 때문에 사용이 편리하고 현재 사용하고 있는 툴(Emacs)들과의 연계할 수 있는 GHDL을 사용하기로 했다. GHDL은 GNU 위원회에서 운영하고 있는 VHDL 컴파일러이다. 오픈소스라서 무료로 사용이 가능하며 누구든 개발에 언제든지 참여할 수 있는 만큼 프로그램의 지원이 끝날 걱정이 없다. 한가지 단점이 있다면 Ada로 프로그래밍돼 있어서 다른 오픈소스 컴파일러에 비해서는 개발에 참여하는데에 단점이 있다는 것.

module poly\_multiplier(a, b, d);

parameter M = 0;

input [M - 1:0] a;

input [M - 1:0] b;

output [2 \* M - 2:0] d;

wire [M - 1:0] a;

wire [M - 1:0] b;

reg [2 \* M - 2:0] d;

wire [2 \* M - 2:0] a\_by\_b[0:2 \* M - 2];

wire [2 \* M - 2:0] c;

generate

genvar k0;

for (k0 = 0; k0 <= M - 1; k0=k0+1) begin : gen\_ands

genvar i0;

for (i0 = 0; i0 <= k0; i0=i0+1) begin : l1

assign a\_by\_b[k0][i0] = a[i0] & b[k0 - i0];

end

end

endgenerate

generate

genvar k2;

for (k2 = 1; k2 <= 2 \* M - 2; k2=k2+1) begin : gen\_xors

reg aux;

integer i;

always @(\*) begin : P2

if ((k2 < M)) begin

aux = a\_by\_b[k2][0];

for(i = 1; i <= k2; i = i + 1) begin

aux = a\_by\_b[k2][i] ^ aux;

end

end

else begin

aux = a\_by\_b[k2][k2];

for(i = k2 + 1; i <= 2 \* M - 2; i = i + 1) begin

aux = a\_by\_b[k2][i] ^ aux;

end

end

d[k2] = aux;

end

end

endgenerate

endmodule

Figure

1. Type System

프로그래밍 언어에서 타입간의 암묵적인 상호작용들이 허용된다면 그것을 약하게 타입화됐다는 의미에서 약타입언어라고 부른다. 반대로 서로 다른 타입들간의 암묵적인 상호작용들을 허용하지 않으면 강타입언어라고 부른다. 일반적인 프로그래밍 언어들 중에서는 C가 약타입, 그리고 C++(C와의 호환성으로 인해 완전한 강타이라고 부를 수는 없으나 언어의 최초 설계자와 현재 국제 운영 위원회는 강타입을 표명하고 있음)이 대표적인 강타입 언어이다.

**signal** count : **std\_logic\_vector**(11 **to** 0)

element <= to\_integer( unsigned( count ), 12 );

VHDL Code

integer element;

always @( count )

element = count;

Verilog Code

Figure

타입시스템의 강약으로 인해 생기는 가장 큰 차이는 프로그래밍의 난이도와 완성되는 결과물의 안정성이다. 약타입 언어에서는 타입간의 상호작용을 위해서 추가적으로 필요한 설계들이 필요 없을 수는 있으나, 원하지 않는 데이터들간의 상호작용이 발생할 수 있다. 반대로 강타입 언어에서는 원치 않는 현상들의 발생들을 예방할 수 있으나 프로그래밍 할 때에 타입들간의 상호작용을 위해 명시적인 변환들을 해줘야 한다(대표적으로 Type Casting). 이러한 특성들은 HDL인 Verilog와 VHDL에도 똑같이 해당되는 사항들이다. Verilog는 타입과 관련해서 명시적인 표현들을 크게 필요로 하지 않는다.

<Figure 3>을 참고하면 위가 VHDL 코드이고 아래가 Verilog 코드이다. VHDL 코드에서는 std\_logic\_vector 타입의 신호를 integer로 만들려면 unsigned 함수를 통해 먼저 signed 타입으로 신호를 바꿔야 한다. 그 다음에 signed를 integer로 만들려면 to\_integer라는 함수를 거쳐야 한다. 이러한 명시적인 type casting들을 해줘야 하기 때문에 VHDL은 강타입 언어이다. 반대로 verilog에서는 비트 데이터를 integer 타입의 데이터에 대입이 특별한 문제 없이 진행된다. 암묵적으로 byte 에서 integer로 변환되기 때문이다.

VHDL은 데이터의 구분을 명확하게 하는 것을 추구하는 만큼 사용자가 직접 타입을 선언하는 것을 허용하고 있다. <Figure 4> 는 새로운 타입을 정의하는 모습이다. Verilog의 경우에는 설계자가 직접 타입을 선언하는 것을 허용하지 않고 있으며 기본적으로 제공되는 타입들만 사용할 수 있다.

Figure

**type** node **is** – binary tree

**record**

key : string(1 to 3);

data : integer;

left : node\_ptr;

right : node\_ptr;

color : color\_type;

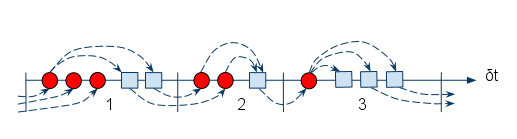
**end record**;

1. Entity and Architecture

모든 VHDL 모듈은 2 종류의 요소들로 이루어져있다. Entity는 각 모듈의 인터페이스를 정의하고, Architecture는 Entity의 행동양식(behavior)을 정의한다.

각 Entity는 여러 개의 Architecture를 가질 수 있다. 이 덕분에 같은 인터페이스를 여러 Architecture에 재사용할 수 있다는 장점이 있으며 이는 일반적인 프로그래밍 언어들의 상속기능과 비슷하게 사용될 수 있다. 또한 테스트벤치를 작성해서 여러 Architecture들을 서로 비교할 때도 사용될 수 있다.

1. Simulating Concurrency in VHDL and Non-Determinism of Verilog

VHDL 코드를 시뮬레이션 프로그램을 통해 컴파일하고 나면 시뮬레이션 실행파일이 출력된다. 이 시뮬레이션은 하드웨어의 동시성을 구현하기 위해 일반적인 프로그램들과는 다른 방식으로 컴파일 된다.

Figure

VHDL 시뮬레이션은 delta-cycle이라고 부르는 단위로 쪼개지며 각 사이클은 몇 개의 스텝으로 이루어져 있다. <Figure 5>는 여러 개의 델타 사이클로 프로그램의 런타임이 나눠진 모습을 표현한 것이다. 각 delta-cycle은 여러 개의 신호 업데이트 프로세스로 이루어져 있는데, 한 사이클 내에 있는 프로세스들은 시뮬레이션하고 있는 회로의 물리적 상태에서 동시에 실행되는 것을 표현한 것이다.

VHDL에서는 일종의 변수할당인 Signal Assignment Statement를 포함하여 모든 연산들이 이 delta-cycle 시스템에서 작동하도록 돼있다.

Verilog의 가장 큰 비판 중에 하나는 Verilog가 기본적으로 동시적이지 못하다는 것이다. Verilog에는 Non-blocking Signal Assignment와 Blocking Signal Assignment가 구분돼 있다. 따라서 프로그래밍을 할 때 동시성이 보장되지 않는다. 설계자는 설계가 동시적이라는 것을 신경 써야 한다는 단점이 있다. 설계자가 이러한 사항에 관해서 충분히 숙달돼 있지 않다면 모든 동시성 프로그램들이 만나는 문제인 Race Condition이 발생한다. 설계에 따라서는 시뮬레이션을 실행할 때마다 시뮬레이션의 과정이나 결과가 달라지는 치명적인 문제가 발생한다.

VHDL은 이러한 Verilog의 문제와는 전혀 무관하다. 하드웨어를 소프트웨어상으로 시뮬레이션 하는 과정에서 발생하는 문제들이 언어상에서 추상화된 상태로 제공되고 있다. 따라서 VHDL의 본 목적인 HD에 집중할 수 있게 해준다.

1. “TIOBE Programming Lanugage Index”; TIOBE; 2017-03-24 13:50. [↑](#footnote-ref-1)