Verilog에서 SystemC로 변환을 위한 효율적인 방법론 연구

신 윤 수, 고 광 철, 정 재 명
한양대학교 전자통신전파공학과
전화 : 02-2290-0348 / 핸드폰 : 019-9720-7786

A research on an efficient methodology for conversion from Verilog to SystemC

Yun-Soo Shin, Kwang-Cheol Ko, Je-Myeong Jeong
Division of Electrical and Computer Engineering, Hanyang University
E-mail : ys0622@ahpe.hanyang.ac.kr

Abstract

Recently, SystemC is one among the language observed. In Industry, there are many the languages that use Verilog. But, unskillful SystemC users must learn SystemC for conversion that from Verilog to SystemC and need time and effort for this. By these reason, feel necessity of easy and efficient conversion.

This paper argues efficient methodology to change Verilog to SystemC. Abstract concepts of Verilog are proposed fittingly each one by one in SystemC.

I. 서론

C언어를 기반으로 한 새로운 설계언어들 중 가장 주목받고 있는 것이 SystemC이다. 그러나 산업기반에 아직은 HDL, 그중 Verilog를 사용한 언어가 많으며, SystemC에 익숙하지 않은 사용자들은 Verilog를 SystemC로 변환을 위해 SystemC를 익혀야 하고, 이를 위해 시간과 노력을 요한다.

본 논문에서는 Verilog에서 SystemC로 변환하는 효율적인 방법 및 절차를 제안한다. 이러한 변환은 변환 과정에서 흔히 발생하는 에러를 피하고, HDL과 SystemC 사이에 비효율적인 수동변환으로 인해 소비되는 시간을 줄이는 것에 의해서 time-to-market을 짧게 만든다.

먼저 Verilog와 SystemC를 비교하고, 이 비교를 근거로 효율적인 conversion methodology를 제안하고, 시뮬레이션으로 결론을 마치기로 한다.

II. HDL Compared: Verilog vs. SystemC

2.1 Fundamental difference in constructs

① Verilog는 task를 call 할 때 keyword를 module로 사용하고, SystemC는 sc_module()로 call한다.

```
module module_name(port_names);
// ports size. direction
// body
endmodule
```

```
SC_MODULE(midule_name){
// ports, processes, internal data, etc
// body
};
// member function
```

그림1. Component declaration의 예

② Verilog는 하나의 module내에서 하나의 process만 처리하지만, SystemC는 여러 process를 처리 할 수 있다. 만약 쓰고자 한다면 외부에서 call을 해야 한다.

③ Verilog는 process를 modeling할 때 흔히 always구

문을 사용하고, SystemC에서는 module class의 member를 사용한다. 또한 함수를 첨가하기 쉽다.

④ SystemC에서는 methods, threads, Clocked threads의 세 가지 type을 쓴다.
- Methods: 감지리스트에 변화가 일어날 때 실행된다.
- Threads: method와 유사하지만, wait() 같은 특정 이벤트에서 중지 또는 동작한다.
- Clocked threads: threads의 특정한 경우로, 더 나은 합성 결과를 위하여 trigger신호의 특정 에지에서 발생한다.

위의 세 가지는 sc_ctrl() macro(SystemC constructor) 사용에 의해 동작한다.

```
sc_ctrl(module_name){   // module constructor
  sc_method(x);         // x process of method type
  sensitive_pos (clock); // sensitive list
}
```

그림2. Process의 예

⑤ Timing mechanism은 Verilog에서는 module에서 clock으로 정의하고, SystemC에서는 sc_clock()구조를 사용한다.

```
Verilog
module module_name(clock);
  output clock;
  reg clock;
  initial
    #5 clock=1;
  always
    #50 clock=~clock;
endmodule
```

```
SystemC
sc_clock constructor_name("constructor_name",
                          20, 0.5, 5, true);
```

그림3. Clock declaration의 예

⑥ Verilog와 SystemC는 variables와 signal을 support 한다.

HDL_Lang	Data Type
Verilog	register: reg, integer, time, real net: wire or tri, wor or trior, wand or triand, tri0, tri1, supply0, supply1, trireg
SystemC	sc_bit, sc_logic, sc_int, sc_uint, sc_bigint, sc_biguint, sc_bv, sc_lv, sc_fixed, sc_ufixed, sc_fix, sc_ufix

표1. Variable 과 Signal.

⑦ Verilog는 단지 bit-vector만 지원하지만 SystemC는 two-valued and four-valued의 풍부한 signal type을 지원한다. fixed precision types은 빠른 시뮬레이션이 가능하고, arbitrary precision types은 큰 버스나 많은 수를 가진 모델을 계산에 사용이 용이하다.

```
sc_signal<base_type> signal_name;
// base_type: integer, real, time(In C++)
// port: sc_in<>, sc_out<>, sc_inout<>
```

그림4. SystemC의 signal type의 예

Ⅲ.Efficient Conversion Methodology

본 논문의 목적은 Verilog에서 SystemC로 제공된 구조로, building block으로 변환하는 것이다.

3.1 Verilog Constructs vs. SystemC constructs

표1는 Verilog 구조를 target SystemC로 간략히 요약한 것으로 모든 Verilog behavioral문들에 대하여 SytemC에서 matching된다.

Level of Abstraction	Verilog Language Construct	SystemC Target Construct
Behavioral	조건문	C/C++ if-then-else
	Case 문	C/C++ switch문
	CaseX 문	mask를 따르는 C/C++ Switch문
	Loops	C/C++ for-while loops
	Tasks와 Functions	C/C++클래스 함수 선언
	Static events	sensitivity list에서 적당한 signal
	Always	SC_METHOD
	Initial	SC_METHOD의 constructor에서 적당한 코드 발생
	모듈 Instantiation	모듈 클래스의 객체가 instantiation됨
	Disable문	C/C++ goto문
	Expressions	적당한 Expressions
RTL	Continuous 할당	SC_METHOD
	Procedural 할당(Blocking)	적당한 method body에서 할당 entry
	Procedural 할당(Nonblocking)	SC_METHOD
Gate Level	Gate Instantiation	SC_METHOD(macro based)
Switch Level	MOS, tran, tranif switch	SystemC에서 구현 안 됨

표2. Verilog와 SystemC mapping

C언어와 유사하게 설계된 Verilog를 변형하기 위해서 추상 구문의 behavioral 레벨을 만든다. 조건문과 case 문, loop문들은 Verilog와 SystemC 구문에서 매우 유사하다. Verilog에서 사용자가 정의한 function과 task는 SC_MODULE 클래스의 멤버 함수로 변형된다. Verilog 대부분의 병렬문은 SC_METHOD에서 발생한다. "always" 구문의 이벤트에 대한 컨트롤은 SC_METHOD의 sensitivity list에서 만들어 진다.

Verilog 모듈 instantiation은 SystemC에서 object instantiation을 위해 mapping된다.

그림5에서 모듈은 다른 모듈에서 instantiate 되었을 때 발생한다. SUB_M 모듈은 TOP_M 모듈에 body에서 instantiate 되었을 때, SC_MODULE 클래스의 객

체는 instantiate 된다.

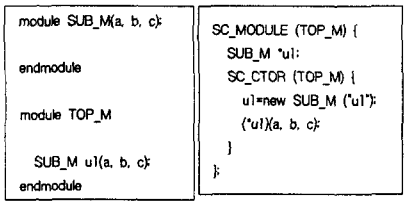

그림5. Module Instatiation

할당문(continuous, blocking, non-blocking)의 세 가지를 포함하는 모든 RTL 구조가 지원된다. 각 continuous문은 SC_MODULE에서 생성된다. Procedural문에 따르는 Blocking문은 단일 SC_METHOD에 생성되는 모든 문은 같은 block문에 있고, 초기에 의도된 것처럼 순차적으로 실행된다. "always" body안에 있는 Non-blocking문은 concurrent문 이다. 그러므로 그들은 각각 단일 SC_MODULE에서 생성된다.

그림6에서 "always" 블록문의 첫 번째는 두 가지 blocking문(c=a; 과 e=a;)과 한 가지 non-blocking문 (d<=b;)이 있다.

non-blocking문은 "non-blocking_d"이란 이름을 갖는 SC_METHOD가 생성될 때, blocking문은 "always_method" body에서 모두 순차적으로 실행된다.

결국, 이러한 문은 전의 두개의 blocking문을 가지고 병렬적으로 실행된다. non-blocking문의 behavioral 방식: 즉, 다른 문을 가진 concurrent는 systemC에서 예약된다.

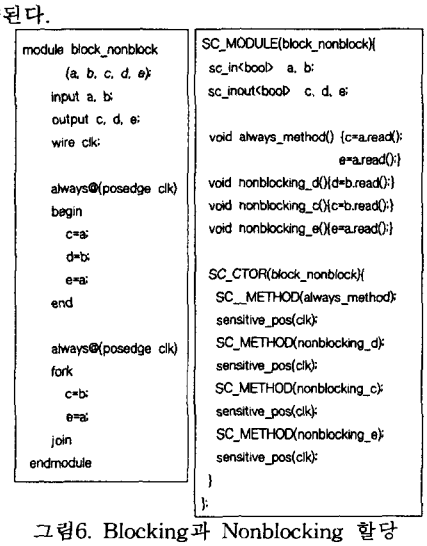

그림6. Blocking과 Nonblocking 할당

그림6에서 다른 "always" block은 Verilog에서 parallel block을 구현하는 fork-join block을 포함한다. fork-join block에서 모든 할당문은 parallel이다.

3.2 Main Features

3.2.1 Verilog에서 Parameters는 SystemC에서 template-based classes로 구현

Verilog에서 parameter에 대한 기본 개념은 광범위하다. 일반적인 개념은 "templates"를 통한 객체지향 언어인 C++에서 구현될 수 있다. SystemC에서 같은 개념으로 구현하기 위하여 template된 SC_MODULE을 사용한다.

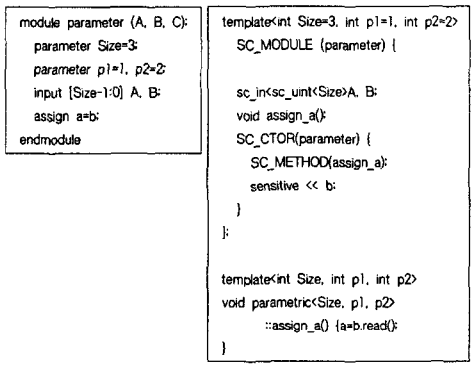

그림7 parameter 구현을 위한 templates

그림7에서 기술된 parameter는 template-base를 기초로 한 SC_MODULE로 만들었다. parameter 값은 template 구조를 통하여 모든 클래스로 연결된다.

3.2.2 Bit와 Range 선택에 대한 Local signals 과 "signal_handler"

Bit select와 range select 연산자들은 "sc_signal" SystemC template 클래스에 대하여 overload되지 않고, 각 signal의 data type은 angle bracket(<>)사이(선언된 시간)에 들어간다.

예를 들면, sc_signal<sc_uint<8>> A;

signal 'A'는 bit select와 range select 연산자들에 대해 sc_uint data type을 갖는다. 그러므로 signal의 range나 bit에 접근하기 위해서 template 클래스안의 data는 끄집어내야 한다. 이것은 "read" 함수를 사용한다. "sc_signal" template 클래스로부터 data type 추출에 의해서 bit([ ])와 range(.range(i,j)) 연산자들이 적용된다. 이것은 모듈 instantiation과 sc_signal" 클래스가 요구되는 *sensitive list method*에서 *signal* passing을 제외한 description body안 어디에서나 동작한다. 따라서 일시적인 local signal들은 index된 부분과 같거나 모듈 instantiation에서 언급된 signal 감지 리스트 요구에 언급된 select range와 같다.

"signal_handler", 모든 index와 range signal을 포함하는 sensitivity list는 sensitivity list와 instantiation에서 사용되는 SC_METHOD, sc_signal template 클래스에 대한 bit select 와 range select 연산자의 공백 때문에 존재한다. single bit 또는 vector 부분은 Verilog 네트리스트에서 선택되고, 적당한 크기를 가진 local 신호는 선언되거나 일반적으로 사용된다. "signal_handler"는 활성화 된 것에 local 신호를 할당한다.

```
module four_bit(A, B, CI, SUM, CO);
    input [3:0] A, B;
    output [3:0] SUM;
    input CI;
    output CO;
    wire C;
    twoB Bit_1_0(A[1:0], B[1:0], CI,
        SUM[1:0], C);
    twoB Bit_3_2(A[3:2], B[3:2], C,
        SUM[3:2], CO);
endmodule
```

```
SC_MODULE(four_bit) {
  sc_signal<sc_uint<2>> A_1_0, A_3_2;
  sc_signal<sc_uint<2>> B_1_0, B_3_2;
  sc_signal<sc_uint<2>> SUM_1_0, SUM_3_2;
  void signal_handler() {
    sc_uint<2> A_tmp;
    for(int i=0; i<2; i++) A_tmp[i]=A.read()[i+1];
    A_1_0= A_tmp;
    sc_uint<4> SUM_tmp=tmp_SUM.read();
    SUM_tmp.range(1,0)=SUM_1_0.read();
    temp_SUM=SUM_tmp;
    for(int i=0; i<2; i++) A_tmp[i]=A.read()[i+2];
    A_3_2=A_tmp;
  }
  SC_CTOR(four_bit) {
    SC_METHOD(signal_handler);
    sensitive<<A<<B<<SUM_1_0<<SUM_3_2;
  }
};
```

그림8 signal_handler method

그림8는 두 2비트 가산기의 instantiation에 의해 구현된 간단한 4비트 가산기이다. range bit 선택은 bit_1_0와 bit_3_2 instantiation에서 사용된다. 활성화 된 신호들의 선택된 범위에 맞추어진 적당한 신호를 가진 loacl 신호들은 선언되고, 대체되어 사용된다. 그리고 "signal_handler"에서 활성 상태로 할당된다.

가령, A[1:0]는 A_1_0의 local signal 선언을 보낸다.

```
sc_signal<sc_uint<2>>A_1_0;
```

signal A_1_0는 모듈 instantiation에 사용되어지고, 코드는 A의 actual range 에 A_1_0 할당에 대하여 "signal_handler"에서 발생한다.

```
sc_uint<2> A_tmp;
for(int i=0; i<2; i++)
A_tmp[i]=A.read()[i+o];
A_1_0=A_tmp;
```

Ⅳ. Simulation Results

본 장에 구현은 DCT Algorithm이며, Verilog 코드는 Altra사의 Quartus2.1 tool을 사용하였고, SystemC 코드는 볼랜드사의 컴파일러를 가진 SystemC_win에서 검증되었다.

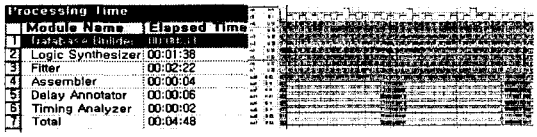

그림9. Verilog DCT Simulation

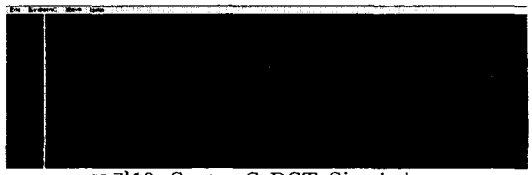

그림10. SystemC DCT Simulation

Line Number	Simulation Time	
	Verilog	SystemC
100	39s	22s
300	4m48s	3m27s

표3. Line수에 따른 Simulation Time

그림9,10은 DCT에 대한 Simulation 결과이고, 표3은 시뮬레이션 시간을 나타냈다. SystemC로 coding할 경우 Line Number가 많아질수록 Verilog에 비해 시뮬레이션이 빠르며, 더 complex coding일 경우 시뮬레이션 차이는 더 벌어질 것으로 보인다.

Ⅴ. Conclusion

최근 들어 대부분의 시스템들은 마이크로프로세서가 내장된 embedded system, co-design 형태가 되었고, 시스템 차원에서 설계를 하게 되었다. 이런 요구에 따라 SystemC의 자연스러운 필요성을 느끼게 되었다.

본 논문은 Verilog로 coding되어진 설계를 SystemC로 변환하는 것에 대한 연구를 하였고. 이와 같이 변환되어진 설계를 시뮬레이션의 속도를 비교함으로서 SystemC의 효율성을 검증하였다.

이와 같은 논문은 SystemC에 익숙하지 않은 Verilog 사용자에게 활용될 수 있으며, 차후 Verilog를 SystemC로 변환하여 설계하는 tool의 기초 자료로 사용될 것으로 본다.

Reference

[1] Thorsten Grotker, Stan Liao , Grant Martin, Stuart Swan, System Design With SystemC, 2002. 3

[2] www.SystemC.org

[3] SOC Design and Verification Using SystemC, IDEC, 2003. 2

[4] Michael D. Ciletti, Advanced Digital Design with the Verilog HDL, 2003