System Verilog

For Verification

System Verilog?

- System Verilog는 Verilog에 비해 Functional Coding에 좋음(코드가 줄어듬)
- Package, Interface등을 Generate와 함께 사용하여 설계의 자동화 기여
- Verilog에서 버그로 판별되는 것이 에러로 인식하게 되어, Synthesis/Simulation의 차이를 줄임

- Verilog → 설계
- System Verilog → 검증용

Simulation 에는 bit, Design 에는 logic

SystemVerilog 에 도입된 새로운 변수 중 bit 와 logic 이 있다. bit 는 2-state variable 로 0 과 1만을 갖는다. logic 은 4-state veriable 로 X, Z, 0, 1 을 가질 수 있다. logic 은 기존의 wire 와 reg 의 자리에 모두 사용될 수 있다. 즉, SystemVerilog 에서는 wire 와 reg 어떤 것으로 선언할지 고민하지 않아도 된다. 코드가 복잡해 지고 길어지다 보면 간혹 발생할 수 있는 실수를 예방하는데 큰 도움이 된다. bit 의 경우 0 과 1만을 갖는다는 것은 Simulation 단계에서의 이야기다. Synthesis 를 할 때, Compiler 는 4-state variable 로 인식한다.

enum {WAITE, LOAD, DONE} STATE;

- Defalut : int(2-state, 32-bits)

→합성(Synthesize시, 4-state로 인식)　　※ System Verilog에서는 Integer가 아닌 int형식이 추가됨

enum logic[2:0] {WAITE, LOAD, DONE} State;

→WAITE = 0, LOAD = 1, DONE = 2 가 됨

enum logic[2:0] {WAITE=3'b100, LOAD=3'b010, DONE=3'b001} State;

→WAITE=3'b100, LOAD=3'b010, DONE=3'b001

Ex) enum logic[1:0] {IDLE, RUN, WAIT, DONE} State; 이런식으로 사용

```
localparam WAITE = 0;
localparam LOAD = 1;
localparam DONE = 2;

reg [2:0] State;
```

[verilog 스타일일때]
Verilog 스타일보다 코드가 간결해짐

typedef

- User-defined-type(UDT)를 지원하는 것이 System Verilog의 특징
- UDT는 enum, struct와 함께 사용되어 design을 한결 간편하게 할 수 있도록 도와줌

```
typedef logic [31:0] bus32_t;
typedef enum logic [0:0] {FALSE, TRUE} bool_t;

module mod1 (
    input bus32_t a, b,
    output bool_t aok
);
// mod1 코드
endmodule
```

- 이렇게 typedef enum logic [0:0] {False, True} bool_t; 쓸경우
  → SIM단계에서 aok의 출력은 FALSE, TRUE로 표시되는게 장점(비트단위로 표시 안됨)

struct

- 구조체
  struct → Packet으로 이름 명명

```
localparam N = 8; // localparam 은 define 처럼 module 밖에서 선언 가능!

typedef struct {
    logic [N-1:0] a;
    logic [N-1:0] b;
} packet;

module top (
    input clk,
    input rst,
    output [N-1:0] c
);

packet p1;

always @(posedge clk) begin
    if(rst) p1 <= '{default: 0}; // p1 의 모든 변수를 0 으로 초기화
    else    p1 <= '{0, 1}; // p1 의 a 는 0, p1 의 b 는 1
end

core u_core (.*); // 이름이 같은 모든 port 를 연결

endmodule

module core (
    input packet p1,
    output [N-1:0] c
);

assign c = p1.a ^ p1.b;

endmodule
```

package

- <package를 통해 다음의 것들을 내부에 묶을 수 있음>
  1) .parameter, localparam
  2) .const
  3) .typedef UDT
  4) .automatic task, function ← 얘는 interface에 묶어, package와 interface의 역할을 달리함

package.sv

```
package TABLE;
    parameter N = 8;
    parameter L = 32;

    typedef logic [N-1:0] bus_t;
    typedef enum logic [1:0] {IDLE, ADDR, CAL, DONE} state_t;

    typedef struct {
        logic [L-1:0] signal_0;
        logic [L-1:0] signal_1;
    } signal_s;
endpackage
```

core1.sv

```
module core1 import TABLE::*; // Wildcard import
(
    input bus_t a, b,
    input signal_s signals,
    // port list 선언
);

state_t PS;

always_ff @(posedge clk) begin : STATE_BLOCK
    if(rst) begin : RESET_CODE
        // reset 내용
    end : RESET_CODE
    else begin
        case(PS) : CASE_STATE
        IDLE : begin : IDLE_STATE
        // 내용
        end : IDLE_STATE
        // 나머지 내용
        endcase : CASE_STATE
    end
end : STATE_BLOCK

// 이하 내용
endmodule : core1
```

core2.sv

```
module core2
import TABLE::signal_s; // import item
(
    input TABLE::bus_t a, b, // explicit reference
    input signal_s signals,
    // 이하 내용
);

TABLE::state_t PS;

// 이하 생략
endmodule : core2
```

wildcard import 를 제외한 나머지 방식들이다. 첫번째는 package 내에서 특정 요소만 import 해오는 방법으로 필자는 `import item` 이라고 부른다. 그 다음 방법은 import 없이 참조하는 방법으로 필자는 `explicit reference` 라고 부른다. 이제, 우리는 package 를 이용하여 모든 약속을 기재할 수 있다. 또한, enum 을 사용하여 무분별한 parameter 선언을 줄일 수 있으며, typedef 를 통해 여러가지 varialbe 들의 이름을 정할 수 있다. 또한, struct 를 사용하여 관련된 여러 signal 들을 묶어서 사용할 수 있다!

주의사항

우리가 package 에서 import 를 해오기 위해서는 package 가 먼저 compile 되어 있어야 한다. 여러 모듈에서 import 를 할 때는 계층이 꼬일 수 있으니 package 를 pre-compile 하도록 하자.

Special procedural blocks

- d

많은 곳에서 소개되고 있는 SystemVerilog 의 procedural blocks 이다. 우리는 보통 Verilog 에서 Sequential block 을 만들 때는 `always @(posedge clk)` 등으로 선언하고 Combinational Logic block 을 만들 때는 `always @(*)` 으로 선언하였을 것이다. 이때, 약속처럼 Sequential block 에서는 Non-Blocking 을 사용했고, Combi block 에서는 Blocking 을 사용했을 것이다. 여기서는 mixed-blocking 에 대한 이야기를 할 것은 아니지만 보통 문제는 Combi block 에서 일어난다. 십중 팔구는 Combi block 에서 의도하지 않은 Latch 가 만들어 지는 것이 문제일 가능성이 높고, Prototyping 을 할 때는 cell library 에 Latch 가 없어서 error 를 보고 Latch 가 생긴 것을 파악할 것이다. 이 문제를 다루기 위해 도입된 것이 Special procedural block 이다.

1). always_ff @(posedge clk) 은 sequential block
2). always_comb begin 은 combi block, 더이상 @(*) 같은 것은 쓰지 않아도 된다.
3). always_latch @(a, b) 은 latch block 인데, 필자는 의도적으로 latch 를 만들어 본 적이 없다.

어찌되었든 always_comb 에서 latch 가 만들어 지면 바로 error 메세지가 나온다. 마찬가지로 always_latch 에서 combi block 이 만들어 지면 바로 error 메세지가 나온다. 사실 이것 외에도 SystemVerilog 는 Verilog 에서 bug 로 인식하고 compile 되던 많은 것들을 error 메세지로 출력한다고 하는데 하나하나 따져 본 것이 아니라 그냥 그렇구나 정도로 이해했다.

Case…inside

- d

- d

- d

- d