DataLab. Internship Program (2023 Summer)

Week 3

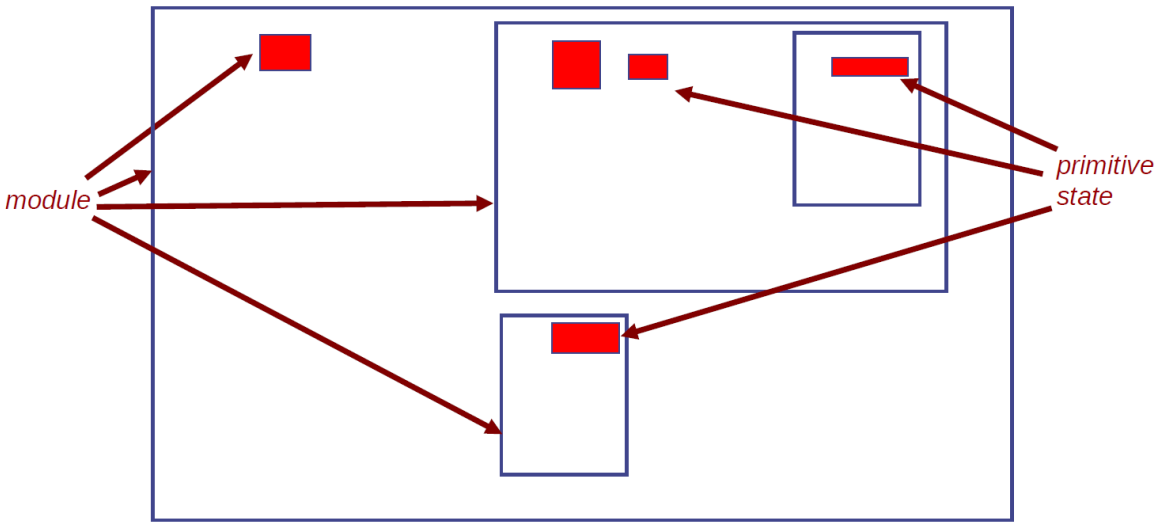
Tutor: 배은태

예제

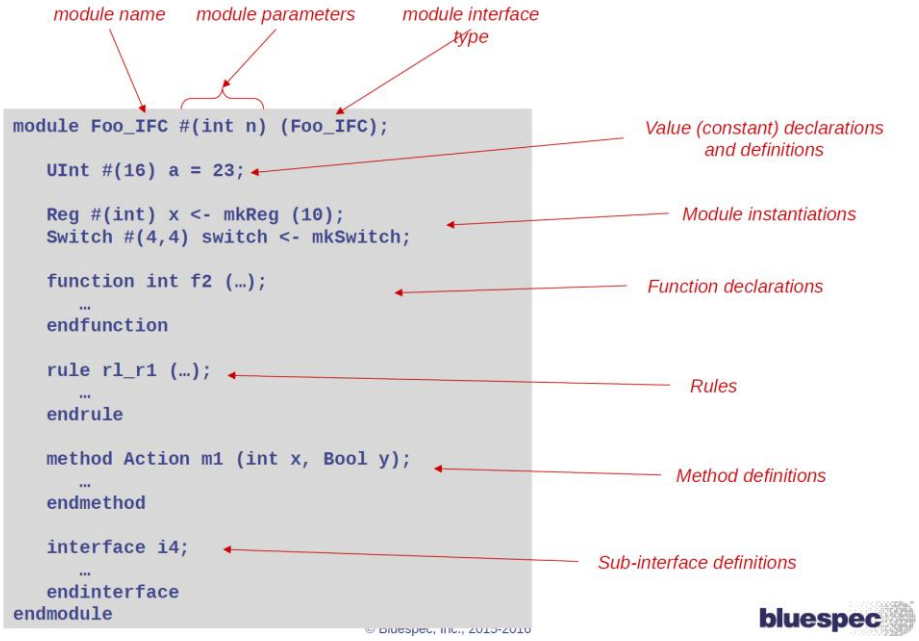
시작하기 앞서

* tutorial-reference/Lec02_Basic_Syntax.pdf와 lecnote-uci/fpga_2 – Bluespec Introduction.pdf를 읽어 보시기를 추천 드립니다.

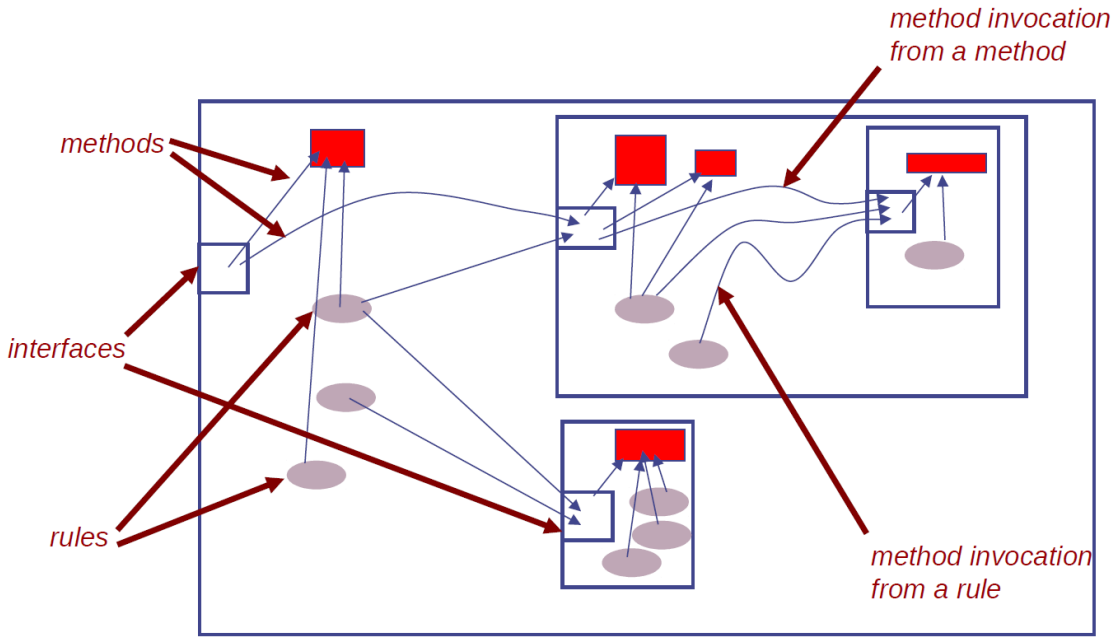
BSV는 Verilog와 하드웨어 개념을 상당 부분 공유하면서도 문법적인 차이도 많이 존재합니다. 때문에 어떤 부분은 친숙하게 다가오고, 어떤 부분은 생소하게 느껴질 것입니다. 예를 들어, BSV는 Verilog와 마찬가지로 module이라는 단위로 하드웨어 모듈을 정의하고, 모듈 안에 모듈, 즉 서브 모듈을 instantiate해서 계층적으로 모듈을 구성할 수 있습니다.



하지만 Verilog에서는 모듈의 몸통(body) 부분에 always나 initial, assign문 등으로 동작을 기술하는 반면에 BSV에서는 method와 rule이라는 단위를 사용합니다.



BSV에는 method들의 선언만을 포함하고 있는 인터페이스(interface)라는 것이 존재한다는 점도 특징입니다. 이는 Java의 interface와 implements와 비슷한 방식입니다. 모듈 이름 다음에는 구현하고자 하는 인터페이스를 명시하고, 인터페이스에 선언된 메서드들을 모듈 내부에 구현합니다.

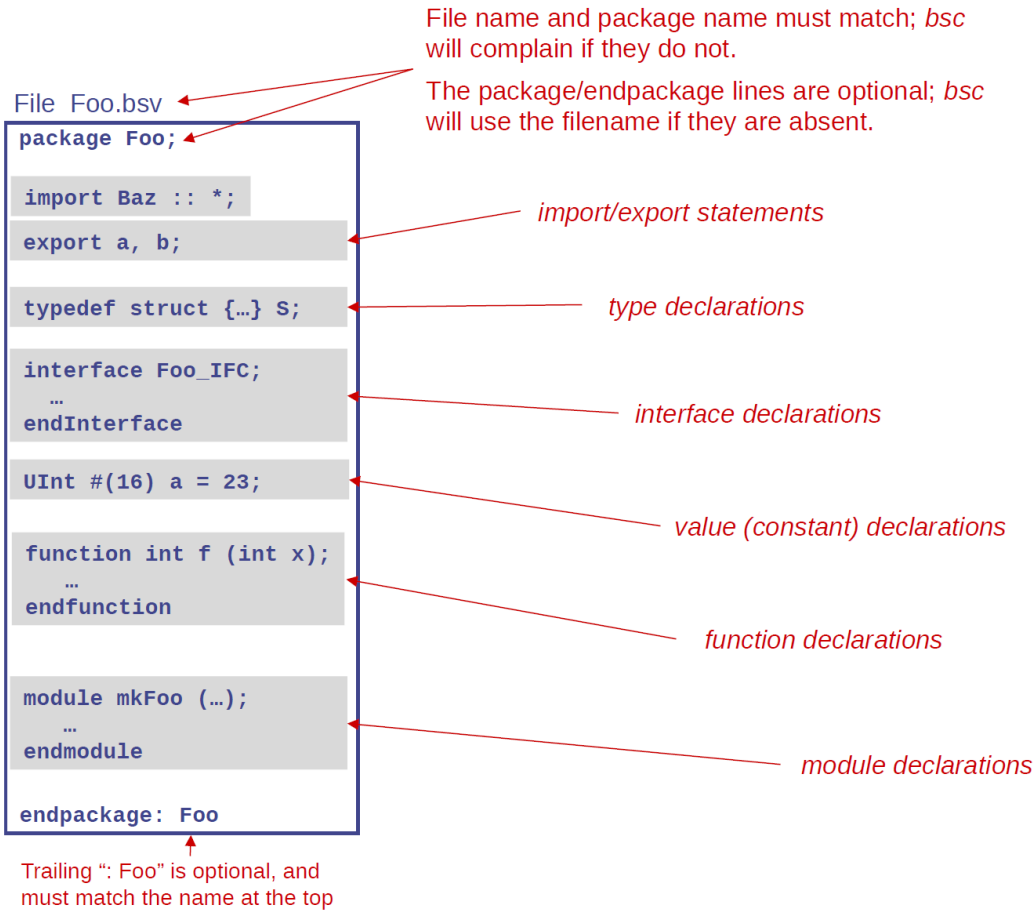


이 메서드들은 모듈 외부에서 호출할 수 있습니다. 예를 들어, 어떤 모듈 Foo가 어떤 모듈 Bar의 서브 모듈이라고 한다면, Bar의 rule에서 또는 Bar의 method에서 Foo의 메서드를 호출할 수 있습니다. 이처럼 BSV는 객체지향과 흡사한 방식의 모듈과 모듈 간의 통신과 상호작용을 문법적으로 제공합니다.

BSV의 또다른 특징은 소스 파일을 '패키지'라는 단위로 관리한다는 점입니다. 패키지 이름은 파

일 이름과 반드시 동일해야 합니다. 예를 들어, Foo.bsv라는 소스파일이 있다면, 코드의 최상단에는 package Foo;라는 패키지 선언과 최하단에 endpackage라는 패키지 종료 선언문이 들어가야 합니다. 이 패키지 시작/종료 선언문은 옵션이기 때문에 별도로 명시하지 않으면 컴파일러가 파일 이름으로 자동 삽입해줍니다. 다음 예시에서는 endpackage 다음에 :Foo라고 종료부에 패키지 이름을 명시한 것을 확인할 수 있는데, 이 부분 역시 옵션입니다. 이런 trailing은 가독성을 위해 패키지 외에도 모듈이나 인터페이스 등에도 명시할 수 있습니다.

패키지를 다른 파일에서 불러오려면 import문을 사용하면 됩니다. 반대로 export문도 제공해주는데, 특정 컴포넌트만 지정해서 export하고 싶을 때 사용합니다. 여기에 명시되어 있지 않은 것들은 패키지 외부에서 볼 수 없습니다. 예를 들어, Foo.bsv의 경우 a와 b를 제외한 나머지 것들은 Foo.bsv 안에서만 접근할 수 있습니다.



Hello World

$display 시스템 태스크를 이용하여 터미널에 "Hello World!"를 띄우는 아주 단순한 예제부터 시작하겠습니다. Tb는 testbench라는 뜻입니다. 테스트벤치는 별도의 인터페이스 메서드가 필요하지 않기 때문에 (Verilog에서 테스트벤치 모듈의 포트가 없는 것처럼) `Empty`라는 인터페이스를 채워줍니다. Bluespec에서는 모듈 이름 앞에 관례상 mk라는 접두어(make라는 의미)를 붙여줍니다. 이는 문법적으로 강제되는 부분은 아니지만, 가독성을 위해 사용합니다.

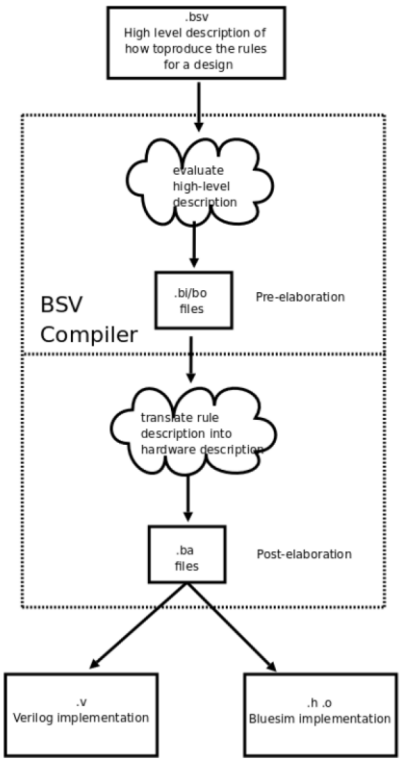
[Tb.bsv]

```
package Tb;

(* synthesize *)
module mkTb (Empty);
    rule greet;
        $display("Hello World!");
        $finish(0);
    endrule
endmodule: mkTb

endpackage: Tb
```

rule 안에 $display와 $finish 시스템 태스크 호출이 있는 것을 볼 수 있습니다. $display를 호출한 뒤에 곧바로 $finish를 호출하기 때문에 "HelloWorld!"는 단 한 번만 출력됩니다.

```
euntae471@EUNTAE-X1:~/project/2023-summer-internship/week3/src/HelloWorld$ ./mkTb
Hello World!
euntae471@EUNTAE-X1:~/project/2023-summer-internship/week3/src/HelloWorld$
```

시뮬레이션 파일을 생성하려면 소스코드를 컴파일 해야 합니다. 이전에 Icarus Verilog를 이용하여 Verilog 소스코드를 컴파일하여 실행파일 형식의 시뮬레이션 파일을 생성한 바 있습니다. 여기에서는 Bluespec 컴파일러인 bsc를 사용합니다. bsc의 컴파일 과정은 다음 그림과 같습니다.



bsc는 BSV 소스코드를 컴파일하여 Verilog 코드를 생성할 수도 있지만, Bluesim이라고 하는

Bluespec에서 사용하는 시뮬레이터 파일을 생성할 수도 있습니다. 물론 Verilog 코드를 생성한 다음에 Icarus Verilog 등으로 컴파일하여 Verilog 시뮬레이터를 생성할 수도 있습니다. 하지만 Bluesim은 cycle accurate할 뿐 아니라 기존의 Verilog 시뮬레이터보다 실행 속도가 10배 빠르다고 합니다. 덕분에 설계한 하드웨어를 더 빠르게 검증할 수 있는 장점이 있다고 합니다. Bluesim을 생성할 때와 Verilog 코드를 생성할 때는 서로 다른 컴파일러 옵션을 사용해야 합니다.

우선 Bluesim을 생성 방법부터 소개하겠습니다. 시뮬레이션 파일을 생성하려면 우선 post-elaboration 파일인 .ba 파일을 생성해야 합니다. .ba 파일은 모듈 단위로 생성됩니다.

```
$ bsc -sim -u Tb.bsv
```

그 다음에는 Bluesim 시뮬레이션 파일을 생성하겠습니다. 여기에서는 실행 파일 이름은 mkTb라고 하겠습니다. -o 옵션을 지정하지 않으면 기본값으로 a.out을 생성합니다.

```
$ bsc -sim -o mkTb -e mkTb mkTb.ba
```

생성된 시뮬레이션 파일은 일반적인 실행파일처럼 실행할 수 있습니다.

```
euntae471@EUNTAE-X1:~/project/2023-summer-internship/week3/src/HelloWorld$ bsc -sim -u Tb.bsv
checking package dependencies
compiling Tb.bsv
code generation for mkTb starts
Elaborated module file created: mkTb.ba
All packages are up to date.
euntae471@EUNTAE-X1:~/project/2023-summer-internship/week3/src/HelloWorld$ bsc -sim -o mkTb -e mkTb mkTb.ba
Bluesim object created: mkTb.{h,o}
Bluesim object created: model_mkTb.{h,o}
Simulation shared library created: mkTb.so
Simulation executable created: mkTb
euntae471@EUNTAE-X1:~/project/2023-summer-internship/week3/src/HelloWorld$ ./mkTb
Hello World!
euntae471@EUNTAE-X1:~/project/2023-summer-internship/week3/src/HelloWorld$
```

이번에는 Verilog 코드 생성 방법입니다. -verilog 옵션을 사용하면 됩니다.

```
euntae471@EUNTAE-X1:~/project/2023-summer-internship/week3/src/HelloWorld$ bsc -verilog Tb.bsv
Verilog file created: mkTb.v
euntae471@EUNTAE-X1:~/project/2023-summer-internship/week3/src/HelloWorld$ ls
Makefile  Tb.bo  Tb.bsv  mkTb.v
euntae471@EUNTAE-X1:~/project/2023-summer-internship/week3/src/HelloWorld$
```

파일을 열어보면 Verilog HDL로 컴파일된 모습을 볼 수 있습니다.

```
//
// Generated by Bluespec Compiler, version 2022.01-29-gc526ff54 (build c526ff54)
//
// On Thu Jul 13 17:05:35 KST 2023
//
//
// Ports:
// Name                         I/O  size props
// CLK                            I     1 unused
// RST_N                          I     1 unused
//
// No combinational paths from inputs to outputs
//
//

`ifdef BSV_ASSIGNMENT_DELAY
`else
  `define BSV_ASSIGNMENT_DELAY
`endif

`ifdef BSV_POSITIVE_RESET
  `define BSV_RESET_VALUE 1'b1
  `define BSV_RESET_EDGE posedge
`else
  `define BSV_RESET_VALUE 1'b0
  `define BSV_RESET_EDGE negedge
`endif

module mkTb(CLK,
            RST_N);
  input  CLK;
  input  RST_N;

  // handling of system tasks

  // synopsys translate_off
  always@(negedge CLK)
  begin
    #0;
    $display("Hello World!");
    $finish(32'd0);
  end
  // synopsys translate_on
endmodule  // mkTb
```

각각의 옵션이 가진 의미는 다음 표에 정리되어 있습니다. 더욱 자세한 내용은 다음 링크에 있는 bsc_user_guide.pdf 파일을 참고하시길 바랍니다.

https://github.com/B-Lang-org/Documentation/tree/master/Language_Spec

```
-g module              generate code for 'module' (requires -sim or -verilog)
-u                     check and recompile packages that are not up to date
-sim                   compile BSV generating Bluesim object
-verilog               compile BSV generating Verilog file
-vsim simulator        specify which Verilog simulator to use
-e module              top-level module for simulation
-o name                name of generated executable
-elab                  generate a .ba file after elaboration and scheduling
```

이에 덧붙여 -g 옵션과 mkTb 모듈 위에 있는 (* synthesize *)라는 코드의 정체에 대해 설명하겠습니다. BSV에서 (*와 *)로 둘러싸인 지시문을 'attribute'라고 합니다. attribute는 컴파일러에게 힌트를 주거나 각종 지시를 내리는 컴파일러 지시자입니다. (* synthesize *) attribute는 **모듈**(패키지가 아니라)에 대한 코드 생성을 컴파일러에게 지시하는 기능을 가지고 있습니다. 이게 없으면 컴파일을 해도 mkTb.v 파일이 생성되지 않습니다. 소스코드에 (* synthesize *)를 넣는 대신 컴파일 옵션으로 -g를 사용하는 방법도 있습니다.

```
$ bsc -verilog -g mkTb Tb.bsv
```

매번 컴파일 명령어를 타이핑하고 빌드 결과물을 지우는 작업은 번거롭기 때문에 Makefile 형식으로 저장해두고 사용하시기를 권장합니다.

```
SRC_NAME=Tb.bsv
all: sim verilog

sim: ba
    bsc -sim -o mkTb -e mkTb mkTb.ba

ba: $(SRC_NAME)
    bsc -sim -u $(SRC_NAME)

verilog: $(SRC_NAME)
    bsc -verilog $(SRC_NAME)

clean:
    rm *.bo
    rm mk*
    rm model_*
```

Value Method

앞서 BSV에서 모듈의 동작은 메서드와 rule을 통해 기술한다고 말씀드린 바 있습니다. 이 메서드라는 것은 모듈의 외부에서 호출될 수 있다고도 말씀드렸죠. 이번 예제에서는 이전에 테스트벤치 단 하나의 모듈만 정의하고 있는 것과 다르게 mkModuleDeepThought이라는 새로운 모듈을 정의하고 있습니다. 우리가 이전에 Verilog로 테스트벤치를 작성할 때 테스트벤치 모듈 안에는 검

증하고자 하는 모듈들을 서브 모듈로 instantiate한 바 있습니다. 이 예제도 마찬가지입니다. mkModuleDeepThought는 mkTb의 서브 모듈입니다. 5행에 ifc라는 이름으로 instantiate되어 있는 것을 확인할 수 있습니다. Verilog와의 차이점에 주목하세요. BSV의 할당 연산자는 세 가지로 나뉜다는 점에 주의해야 합니다. 5행과 같이 모듈의 인스턴스를 instantiate할 때는 <- 연산자를 사용합니다. 이 예제는 모듈의 인터페이스를 정의할 뿐 아니라 메서드의 구현, 해당 인터페이스를 구현하는 모듈의 instantiation, 메서드의 호출까지 다루고 있습니다.

[Tb.bsv]

```
package Tb;

(* synthesize *)
module mkTb (Empty);
    Ifc_type ifc <- mkModuleDeepThought;

    rule theUltimateAnswer;
        $display("Hello World! The answer is: %d", ifc.the_answer(10, 15, 17));
        $finish(0);
    endrule
endmodule: mkTb

interface Ifc_type;
    method int the_answer (int x, int y, int z);
endinterface: Ifc_type

(* synthesize *)
module mkModuleDeepThought (Ifc_type);
    method int the_answer (int x, int y, int z);
        return x + y + z;
    endmethod
endmodule: mkModuleDeepThought

endpackage: Tb
```

일단 실행 결과부터 확인해보겠습니다.

```
euntae471@EUNTAE-X1:~/project/2023-summer-internship/week3/src/ValueMethod$ ./mkTb
Hello World! The answer is:          42
```

8행에서 ifc.the_answer(10, 15, 17)이라고 메서드를 '호출'했고, 10+15+17의 결과가 제대로 출력된 모습을 확인할 수 있습니다. 겉보기에는 클래스의 인스턴스를 생성하고 인스턴스의 메서드를 호출하는 듯한 이 코드만 보면 매우 자연스러운 코드지만, 이 코드의 실제 결과물이 하드웨어라는 점을 늘 염두에 두어야 합니다.

그럼 컴파일된 결과물을 확인해봅시다. 합성 툴이 Verilog로 작성된 RTL 코드를 해석하여 적절한 하드웨어로 매핑하듯 bsc는 BSV로 작성된 코드를 적절한 Verilog 코드로 변환합니다. Tb.bsv 안에

는 두 개의 모듈이 정의되어 있으므로(그리고 각 모듈에 (* synthesize *) attribute가 명시되어 있으므로) 컴파일 결과로 Verilog 소스파일은 mkTb.v와 mkModuleDeepThought.v 두 개가 생성됩니다. mkModuleDeepThought.v부터 보겠습니다.

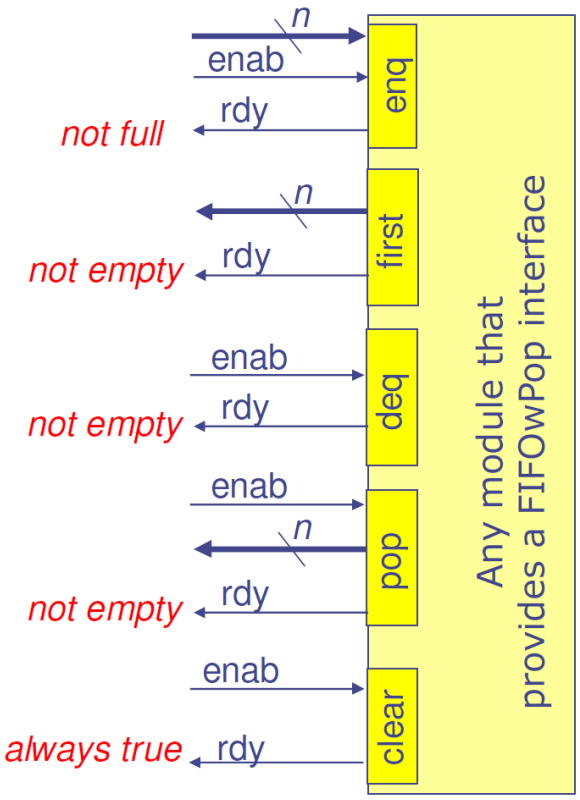
```
module mkModuleDeepThought(CLK,
         RST_N,

         the_answer_x,
         the_answer_y,
         the_answer_z,
         the_answer,
         RDY_the_answer);
  input  CLK;
  input  RST_N;

  // value method the_answer
  input  [31 : 0] the_answer_x;
  input  [31 : 0] the_answer_y;
  input  [31 : 0] the_answer_z;
  output [31 : 0] the_answer;
  output RDY_the_answer;

  // signals for module outputs
  wire [31 : 0] the_answer;
  wire RDY_the_answer;

  // value method the_answer
  assign the_answer = the_answer_x + the_answer_y + the_answer_z ;
  assign RDY_the_answer = 1'd1 ;
endmodule  // mkModuleDeepThought
```

여기서 알 수 있는 사실은 메서드의 매개변수(parameter)와 반환 값(return value)이 입출력 포트로 컴파일 된다는 사실입니다. 매개변수 x, y, z는 각각 method the_answer의 이름을 딴 the_answer_x, the_answer_y, the_answer_z라는 입력 포트로 변환되었습니다. 반면 return value는 the_answer라는 출력 포트로 변환되었습니다. 모두 int형으로 정의되었기 때문에(이 부분에 대해서는 BSV의 자료형을 참고하시기 바랍니다) 32비트 정수인 것을 확인할 수 있습니다.

또 눈여겨볼 지점은 메서드의 구현입니다. BSV에는 세 가지 종류의 메서드가 있습니다: Value method, Action method, ActionValue method. 이번 예제에서 다루는 value method는 레지스터 값 등 하드웨어의 내부 상태(state)를 변화시키는 동작, 즉 side effect가 없다는 것이 특징입니다. 그래서 주어진 신호들을 연결하여 the_answer라는 출력 신호를 내보내는 조합회로로 구현되었습니다. 반대로 하드웨어의 상태를 변화시키는 동작을 Bluespec에서는 Action이라고 합니다. 레지스터 값의 업데이트가 대표적인 Action이라고 할 수 있습니다. Action method는 이런 Action을 수행하는 메서드인 것이고 void 함수와 같이 값을 반환하지 않습니다. ActionValue method는 Action을

수행하면서도 값도 반환하는 메서드입니다. rule은 이런 Action들의 집합으로 이뤄져 있고, 따라서 메서드 호출들이 rule 안에 들어갈 수 있습니다.

그리고 코드를 보면 생소한 신호가 하나 더 있는 것을 볼 수 있습니다. RDY_the_answer라는 출력 신호입니다. 메서드를 '호출'한다고 했는데, 이것이 하드웨어적으로 어떻게 가능한지를 알려주는 열쇠라고 할 수 있습니다. RDY는 ready라는 의미입니다. 준비가 되었다. 즉, method를 호출한 호출자에게 해당 메서드로 구현된 회로가 실행 가능한 상태임을 알려주는 신호입니다. 이렇게 method의 호출을 위해 호출자(caller)와 피호출자(callee) 간에 상호 교환되는 신호를 handshake 신호라고 부릅니다. handshake 신호에는 EN(enable)과 RDY(ready)가 있습니다. 이번 예제에서는 side-effect가 없는 value method를 다루었기 때문에 RDY 신호만 생성됐지만, side-effect가 있는 Action method나 ActionValue method에는 EN 신호가 추가로 생성됩니다.



이번에는 테스트벤치 모듈인 mkTb.v를 보겠습니다. ifc는 역시 모듈 mkModuleDeepThought의 인스턴스임을 확인할 수 있습니다. 서브 모듈 ifc의 메서드를 호출하는 호출자(caller)인 mkTb는 메서드의 매개변수, 즉 입출력 포트에 신호선을 연결하고 있습니다.

```
module mkTb(CLK,
      RST_N);
  input  CLK;
  input  RST_N;

  // ports of submodule ifc
```

```
  wire [31 : 0] ifc$the_answer,
    ifc$the_answer_x,
    ifc$the_answer_y,
    ifc$the_answer_z;

  // submodule ifc
  mkModuleDeepThought ifc(.CLK(CLK),
        .RST_N(RST_N),
        .the_answer_x(ifc$the_answer_x),
        .the_answer_y(ifc$the_answer_y),
        .the_answer_z(ifc$the_answer_z),
        .the_answer(ifc$the_answer),
        .RDY_the_answer());

  // submodule ifc
  assign ifc$the_answer_x = 32'd10 ;
  assign ifc$the_answer_y = 32'd15 ;
  assign ifc$the_answer_z = 32'd17 ;

  // handling of system tasks

  // synopsys translate_off
  always@(negedge CLK)
  begin
    #0;
    $display("Hello World! The answer is: %d", $signed(ifc$the_answer));
    $finish(32'd0);
  end
  // synopsys translate_on
endmodule  // mkTb
```

Conditional Rule

rule과 메서드의 또다른 특징은 실행 조건을 명시할 수 있다는 것입니다. 메서드나 rule의 이름 다음에 붙는 조건식을 Bluespec에서는 guard라고 부릅니다. 메서드를 호출하거나 rule을 실행하기 전 매 사이클마다 guard의 조건을 검사하여 실행 여부를 제어합니다. guard의 조건을 만족하지 않으면 rule이나 메서드는 실행되지 않습니다. 이번 예제는 8비트 레지스터 cnt의 값의 변화에 따라 rule의 실행 여부가 제어되는 것을 확인할 수 있습니다. 이전 예제에서 사용된 Makefile을 이용하여 컴파일하여 시뮬레이션 결과를 확인해봅시다.

```
package Tb;

(* synthesize *)
module mkTb (Empty);
    Reg#(Bit#(8)) cnt <- mkReg(0);
```

```
    rule cntUp if (cnt <= 10);
        $display("cntUp: cnt == %d", cnt);
        cnt <= cnt + 1;
    endrule

    rule cntEnd if (cnt >= 10);
        $display("cntEnd: cnt == %d", cnt);
        $finish(0);
    endrule
endmodule

endpackage
```

Conditional Method

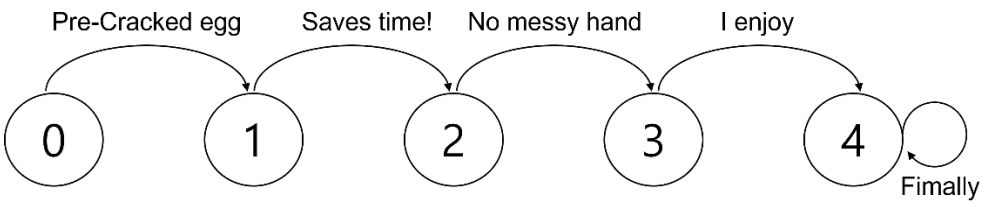
이번에는 guard가 있는 메서드의 실행을 확인해봅시다. 마찬가지로 기존에 제공된 Makefile을 이용하여 컴파일하고 시뮬레이션 결과를 확인해봅시다.

```
package Tb;

interface Egg_IFC;
    method Action pend;
    method Action msgSend;
endinterface

(* synthesize *)
module mkEgg (Egg_IFC);
    Reg#(Bit#(8)) ansCnt <- mkReg(0);
    Reg#(Bool) pending <- mkReg(False);

    rule msgGenerate if (pending);
        //$display("msgGenerate: ansCnt=%d", ansCnt);
        if (ansCnt == 20)
            pending <= False;
        else begin
            $write(".");
            ansCnt <= ansCnt + 1;
        end
    endrule

    method Action pend if (!pending);
        pending <= True;
    endmethod

    method Action msgSend if (!pending && ansCnt >= 20);
```

```
        $display("Fimally!");
        $finish(0);
    endmethod
endmodule

(* synthesize *)
module mkTb (Empty);
    Egg_IFC egg <- mkEgg;
    Reg#(Bool) busy <- mkReg(False);

    rule startup (!busy);
        $display("Pre-Cracked egg");
        $display("Saves time!");
        $display("No messy hand");
        $write("I enjoy ");
        egg.pend();
        busy <= True;
    endrule

    rule msgPend (busy);
        egg.msgSend();
    endrule
endmodule

endpackage
```

Finite State Machine (FSM)

유한상태기계(Finite State Machine, FSM)이란 유한개의 상태가 존재하고, 현재 상태(레지스터 값 등이 되겠죠?)에 따라 입력에 대응하여 상태가 변화하는 기계를 의미합니다. FSM은 계산이론에서도 익숙한 존재지만, 실생활에서도 흔히 접할 수 있습니다. 예를 들어, 자판기에 1000원짜리 지폐를 넣었을 때 1200원짜리 음료 버튼을 누르면 음료가 나오지 않지만, 800원짜리 음료 버튼을 누르면 나오는 것처럼 FSM은 현재의 내부적인 상태 값에 따라 입력에 반응하여 상태를 전이(transition)시킵니다. 상태가 변화하지 않고 현재 상태로 돌아올 수도 있습니다. 이번 예제에서는 상태 레지스터 state에 따라 동작하는 간단한 하드웨어를 준비했습니다. 컴파일해서 시뮬레이션 결과를 확인해본 다음 29행의 $finish 태스크 호출에 주석처리를 하고 다시 결과를 확인해봅시다. 어떤 결과가 나올까요? 힌트는 바로 다음 그림에 있습니다.



```
package Tb;

(* synthesize *)
module mkTb (Empty);
    Reg#(Bit#(3)) state <- mkReg(0);

    rule rlStartup (state == 0);
        $display("Pre-Cracked egg");
        state <= 1;
    endrule

    rule rlSavesTime (state == 1);
        $display("Saves time!");
        state <= 2;
    endrule

    rule rlNoMessyHand (state == 2);
        $display("No messy hand");
        state <= 3;
    endrule

    rule rlIEnjoy (state == 3);
        $display("I enjoy");
        state <= 4;
    endrule

    rule rlFimally (state == 4);
        $display("Fimally!");
        $finish(0);
    endrule
endmodule

endpackage
```

Greatest Common Divisor (GCD)

이번 예제에서는 유클리드 호제법(Euclidian Algorithm)을 이용하여 최대공약수를 계산합니다. 알고리즘의 동작은 크게 두 단계로 나뉩니다: 빼기, 교환하기; 이 두 과정은 두 값이 서로 같아질 때까지 반복됩니다. 입력된 수를 바꿔가며 결과를 확인해봅시다.

```
package Tb;
    interface GCD_IFC;
        method Action start(Bit#(32) a, Bit#(32) b);
        method Bit#(32) result();
    endinterface
```

```
/* Euclidian Algorithm: */
/*
def gcd(a, b):
    if a == 0:
        return b # stop
    elif a >= b:
        return gcd(a - b, b) # subtract
    else:
        return gcd(b, a)
*/

(* synthesize *)
module mkGCD (GCD_IFC);
    Reg#(Bit#(32)) x <- mkReg(0);
    Reg#(Bit#(32)) y <- mkReg(0);
    Reg#(Bit#(32)) res <- mkReg(0);

    rule step1 ((x > y) && (y != 0));
        // swap
        x <= y;
        y <= x;
        $display("step1: %d, %d", x ,y);
    endrule

    rule step2 ((x <= y) && (y != 0));
        y <= y - x;             // subtract
        if (y - x == 0) begin   // stop
            res <= x;
        end
        $display("step2: %d, %d", x ,y);
    endrule

    method Action start(Bit#(32) a, Bit#(32) b) if (y == 0);
        x <= a;
        y <= b;
    endmethod

    method Bit#(32) result();
        return res;
    endmethod
endmodule

(* synthesize *)
module mkTb (Empty);
    GCD_IFC gcd <- mkGCD;
    Reg#(Bit#(32)) gcdResult <- mkReg(0);
    Reg#(Bit#(3)) state <- mkReg(0);
```

```
        rule rlStart (state == 0);
            gcd.start(27, 15);
            state <= 1;
        endrule

        rule rlPend (state == 1);
            $display("wait...");
            gcdResult <= gcd.result();
            if (gcdResult != 0) begin
                state <= 2;
            end
        endrule

        rule rlResult (state == 2);
            $display("GCD Result: %d", gcdResult);
            $finish(0);
        endrule

    endmodule
endpackage
```

연습문제

이번 연습 문제는 Bluespec을 이용하여 개발하기 위해 어느정도 이해하고 넘어가야 하는 내용들을 위주로 정리해봤습니다. 새로운 언어를 학습하는 만큼 생소한 개념들을 익히는 데 충분히 시간이 필요하기 때문에 이번 문제들은 따로 기한을 두지는 않겠습니다. 직접 공부한 내용을 정리한다는 느낌으로 풀어 보시면 좋을 것 같습니다.

1. synthesize attribute의 용도는 무엇일까요?

2-1. method의 종류에는 무엇이 있을까요?

2-2. side-effect가 있다는 것은 어떤 의미일까요?

3-1. Bluespec의 할당(assign) 연산자 세 종류에 대해 알아봅시다.

3-2. 레지스터 할당 연산자는 어떤 메서드로 변환될까요? (BSV에서는 Reg도 일종의 모듈입니다.)

3-3. Bluespec에서 = 연산자는 Verilog와 어떻게 다를까요?

4-1. 컴파일러(bsc)가 rule을 스케줄링 해야 하는 이유는 무엇일까요? (rule이 여러 개 있을 때 어떻게 동작하는지를 생각해봅시다.)

4-2. CAN_FIRE와 WILL_FIRE의 차이점에 대해 알아봅시다.

4-3. 두 attribute execution_order와 descending_urgency의 차이점에 대해 알아봅시다.

5. method와 rule의 'guard'에 대해 알아봅시다.

6. BSV의 자료형인 Int와 Integer의 차이점은 무엇일까요?

예제 소스코드

소스코드는 다음 링크에서 찾아보실 수 있습니다.

https://github.com/euntae-bae/2023-summer-internship/tree/main/week3/src