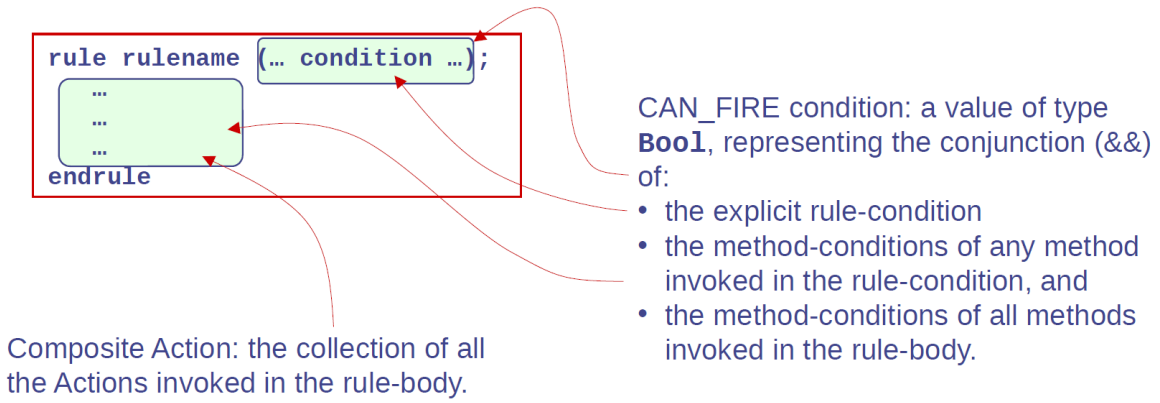
DataLab. Internship Program (2023 Summer)

Week 5

Tutor: 배은태

Actions and Rules

BSV는 Verilog와 SystemVerilog의 문법을 많은 부분 공유하면서도 동작(behavior)을 기술하기 위해 rule과 method를 사용한다는 점에서 큰 차이가 있습니다. 모듈의 동작을 기술하는 rule은 Verilog의 always문과 같은 존재입니다. 하지만 완전히 똑같지는 않습니다. 훨씬 강력하고 직관적입니다. rule은 guard라고 불리는 Boolean 조건식과 일련의 Action들로 구성된 body로 구성됩니다. Bluespec에서 말하는 Action이란 회로의 상태(state)를 변화시키는(즉, side-effect가 있는) 동작을 의미합니다. action은 atomic하다는 특징이 있습니다. 이 rule이라고 하는 것은 매 사이클 guard의 조건식을 검사하여 조건을 만족할 때마다 실행됩니다.



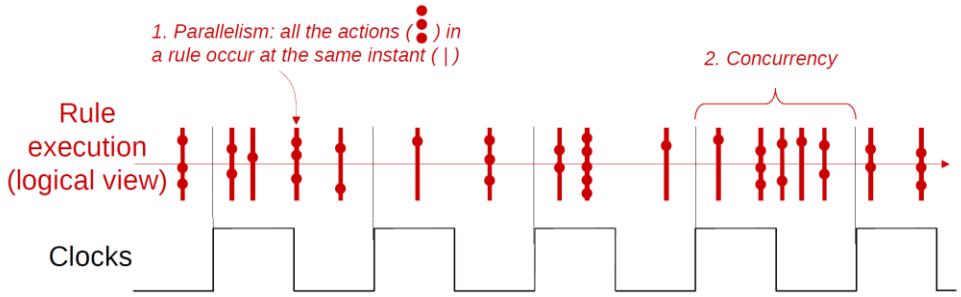
rule은 요컨대 condition+action입니다. 실행조건(condition)이 rule guard에 들어가고, 동작(action)이 rule body에 들어갑니다. rule guard는 rule condition 또는 **explicit condition**이라고도 부릅니다. rule guard에는 반드시 side-effect가 없는 Boolean 조건식만 들어갈 수 있습니다. rule guard를 생략하면 조건이 참으로 간주됩니다.

rule body에는 하나 이상의 action이 들어갑니다. 하나의 rule body 안에 있는 action들은 한 사이클 동안 동시에, 병렬적으로 실행된다는 점을 주의해야 합니다. Bluespec에서는 이를 **parallelism of actions**라고 표현합니다. action들은 병렬적으로 실행되기 때문에 action의 기술 순서를 바꿔도 같은 하드웨어를 생성합니다. 예를 들어, rl1과 rl2의 동작은 서로 동일합니다.

```
rule rl1;
    x <= x + 1;
    fifo.enq(x);
endrule

rule rl2;
    fifo.enq(x);
```

```
    x <= x + 1;
endrule
```

한편 rule은 여러 개가 될 수 있습니다. rule들은 모두 한 사이클 내에 실행되는 동작이지만, 논리적으로는 모종의 순서를 가집니다. 컴파일러는 이를 제어하는 조합회로로 구성된 스케줄러를 생성합니다. 한 사이클 동안 여러 rule들의 실행을 **concurrency of rules**라고 표현합니다. 다음 그림은 이 둘을 시각화한 것입니다. 각각의 막대는 하나의 rule을 나타내고, 막대 안에 있는 점들은 rule body에 있는 action들을 나타냅니다.

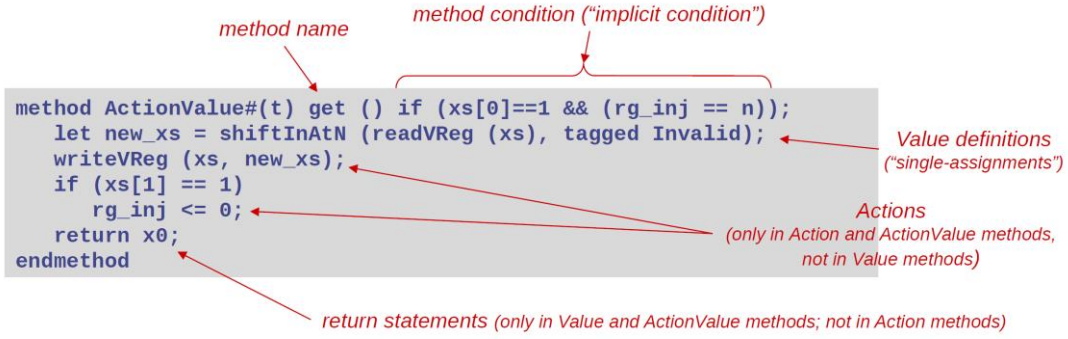


Rule은 언제 실행되나?

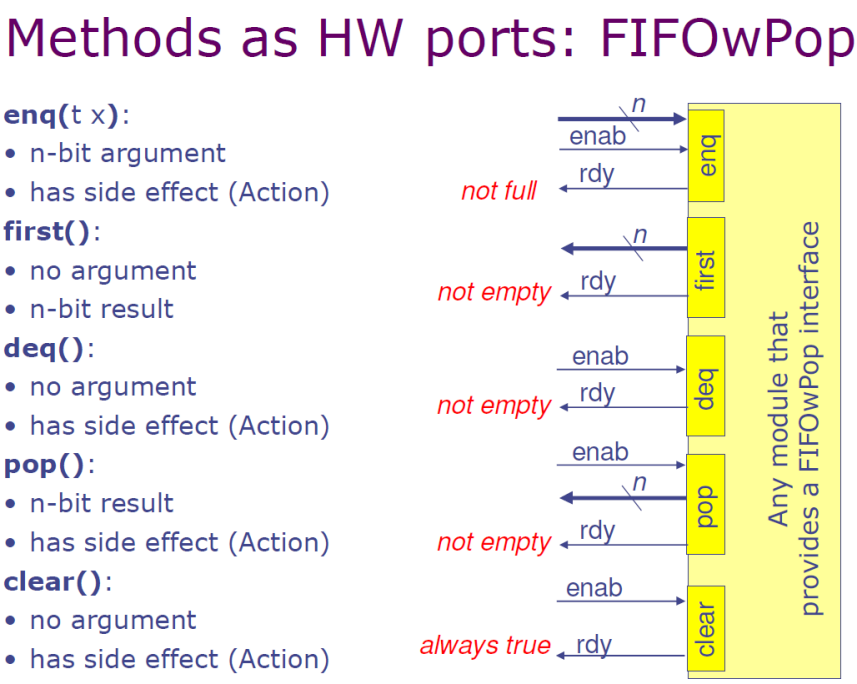
rule은 매 사이클마다 실행 조건을 검사하여 조건을 만족할 때만 실행된다고 설명한 바 있습니다. 그리고 rule guard에 실행 조건을 기술한다고 설명했죠. 이를 **explicit condition**이라고도 합니다. 그 외에도 rule이 실행되려면 크게 두 가지를 더 검사해야 합니다.

(1). rule body에서 호출하는 메서드가 모두 ready 상태인지의 여부

ready 신호는 메서드마다 존재하는 일종의 핸드셰이킹(handshaking) 신호로, 메서드의 실행을 제어하기 위해 컴파일러에 의해 자동으로 생성됩니다. 메서드의 호출 역시 일종의 action이고, action이 실행되려면 메서드도 호출 가능한 상태여야 합니다. 사실 메서드는 일종의 rule의 조각이라고 볼 수도 있습니다. 메서드는 rule과 구성도 semantic도 같습니다. 메서드도 rule과 마찬가지로 guard와 body로 이루어져 있습니다. 들어가는 내용도 body에는 action들이(엄밀하게는 Action 메서드와 ActionValue 메서드에서), guard에는 실행 조건이 들어가는 것으로 동일합니다. 다만 메서드의 guard의 경우에는 **implicit condition**이라고 구분해서 부릅니다. rule에 있는 메서드 호출들 중 실행 조건을 만족하지 못하는 경우가 존재하면 해당 rule은 실행되지 않습니다.



rule에서 서브 모듈 인스턴스의 메서드를 '호출'하는 것은 마치 객체지향 언어의 클래스 객체와 메서드 호출을 보는 것 같습니다. 하지만 Bluespec의 컴파일 결과는 하드웨어라는 사실을 기억해야 합니다. BSV 메서드는 소프트웨어 코드와 달리 호출 시 스택 프레임이 생성되지 않습니다. 다음 그림을 보면 좀 더 명확하게 이해할 수 있을 것입니다. 그림은 FIFO와 그 메서드들의 관계를 나타내고 있습니다. Bluespec 컴파일러는 메서드의 동작을 제어하는 하드웨어를 구현하기 위해 메서드마다 핸드셰이킹 신호인 EN(enable)과 RDY(ready)를 생성합니다. 모두 모듈의 입출력 신호로 구현됩니다. 이들 신호는 메서드의 이름과 연결해서 EN_method_name, RDY_method_name 형식의 이름으로 지어집니다. RDY는 메서드 호출이 가능한 상태를 나타내기 위해 모듈 밖으로 나가는 출력 신호이고, EN은 메서드의 action을 실행하기 위해 모듈로 들어가는 입력 신호입니다. 그 외에 메서드의 매개변수는 모듈의 입력 포트로, 반환 값은 출력 포트로 컴파일됩니다. 이들 신호 역시 메서드의 이름과 연관되어 지어집니다. 이를 통해 메서드는 모듈과 외부 세계의 통신을 위한 '인터페이스'를 제공합니다.

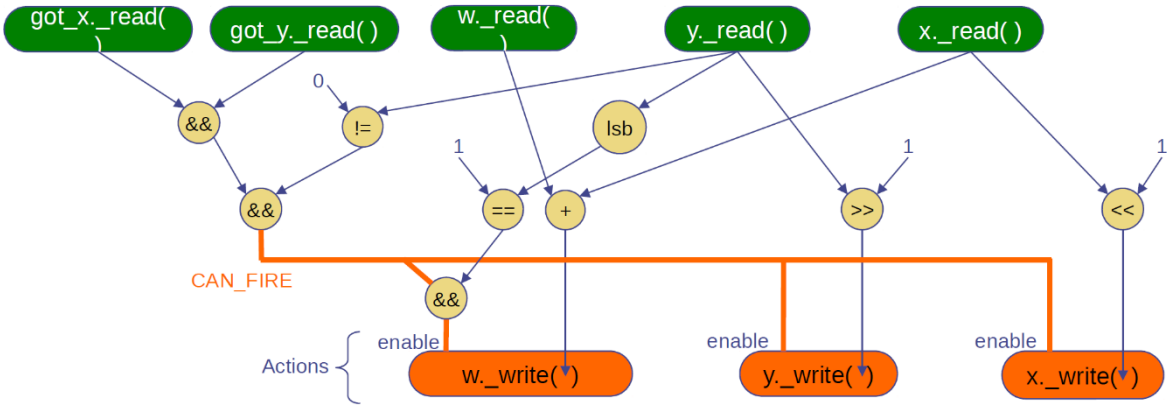


EN과 RDY 두 신호를 동시에 설명하긴 했지만, 사실 EN은 RDY와 달리 Action, 즉 side-effect를 포함하는 Action과 ActionValue 메서드에서만 생성됩니다. 그림을 잘 보시면 Value 메서드인 first에는 EN 신호가 존재하지 않는다는 사실을 확인할 수 있습니다. Action 메서드의 경우에는 side-effect가 있기 때문에 EN 신호로 실행을 제어해야 합니다. 만약 메서드가 매 사이클 실행조건이 참이고(RDY), 매 사이클 실행된다면(EN) 두 신호는 존재할 필요가 있을까요? Bluespec 컴파일러는 불필요한 핸드셰이킹 신호를 제거할 수 있도록 always_enabled와 always_ready라는 attribute를 제공합니다. 물론 이 atttibute를 적용하려면 EN이나 RDY 신호가 항상 참이어야 합니다. 그렇지 않으면 컴파일러는 오류를 발생시킵니다.

지금까지 설명한 rule guard(explicit condition)와 메서드 호출 조건(method guard 또는 implicit

condition)을 모두 만족하면 해당 rule은 현재 사이클에서 실행 가능합니다. 조건식은 모두 side-effect가 없기 때문에 사이클마다 조건을 검사하는 로직은 조합회로로 구현 가능합니다*(참고로 레지스터 읽기는 Value method인 _read, 쓰기는 Action method인 _write입니다. Bluespec 컴파일러는 레지스터에 한해서는 표준적인 메서드 호출 방식 대신 읽기는 생략을, 쓰기는 <= 연산자로 대체하는 것을 허용합니다).* 검사를 통과하면 "이 rule은 현 사이클에서 실행 가능합니다"를 나타내는 **CAN_FIRE**라는 신호가 True가 됩니다. CAN_FIRE가 True이면, rule에서 호출하는 메서드들의 EN 신호도 True가 됩니다. Action 메서드의 EN이 True가 되면 Action이 실행됩니다.

```
rule compute ((y != 0) && got_x && got_y) ;
   if (lsb(y) == 1) w <= w + x;
   x <= x << 1;
   y <= y >> 1;
endrule
```



사실 결론부터 말씀드리자면, CAN_FIRE만 가지고는 rule이 여러 개인 상황을 고려하지 못합니다. CAN_FIRE는 일차적으로 만족해야 하는 것입니다. rule이 여러 개일 때는 상황이 좀 더 복잡합니다. 바로 rule과 rule 사이에 충돌(conflict)이 발생할 수 있기 때문입니다. 이 부분에 대해서는 다음 절에서 좀 더 자세히 다루겠습니다.

(2). 충돌하는 rule의 존재 여부

rule이 충돌(conflict)한다는 것은 무엇을 의미할까요? rule은 여러 개가 될 수 있다고 이미 언급한 바 있습니다. 그리고 rule body에는 일련의 action이 들어가게 되죠. 이 rule들은 한 사이클 내에 실행됩니다. 우선 단일 rule의 경우부터 다시 짚고 넘어갑시다. rule 하나만 봤을 때는 단순명료했습니다. 하나의 rule 안에 있는 action들은 모두 동일한 순간에 동시에, 병렬적으로 실행됩니다. 이를 parallelism of actions라고 소개한 바 있죠. 따라서, rule body에 있는 action들에는 순서가 존재하지 않습니다. Bluespec에서는 rule 하나가 atomic한 실행 단위입니다. DB에서의 atomic transaction을 떠올리면 좋을 것 같습니다. 중간만 실행되다가 마는 건 존재하지 않습니다. all-or-nothing인 것이죠. 그럼 rule이 여러 개일 때는 어떻게 될까요? Bluespec 컴파일러는 여러 rule들이 논리적인 순서를 갖고 실행되도록 **스케줄링** 합니다. rule r1부터 rN까지 있다고 가정했을 때, r1

부터 rN은 모두 한 사이클 안에 실행되지만 모종의 선형적인 순서(우선순위)를 갖습니다. 이를 concurrency of rules라고 부릅니다. rule이 실행되기 위해서는 조건을 만족해야 한다고 했습니다(CAN_FIRE). 만약 현재 사이클에서 여러 rule이 동시에 조건을 만족한다면 어떻게 될까요? 같은 클럭에서 실행 불가능한 rule이 존재하면 이를 충돌(rule)이라고 합니다. 충돌이 발생하면 우선순위가 높은 rule에게 실행이 넘어갑니다. 나머지 우선순위가 낮은 rule은 CAN_FIRE가 True라도 실행되지 않습니다.

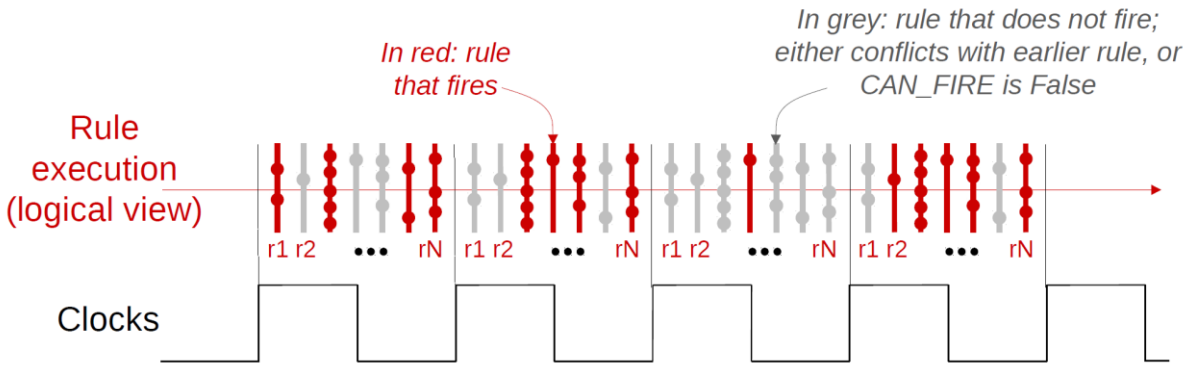
Define a *schedule* as some linear ordering of all rules in a program: r1 r2 ••• rN

Then, the semantics of multiple rules in a clock (logical concurrency) is simple:

For each clock:

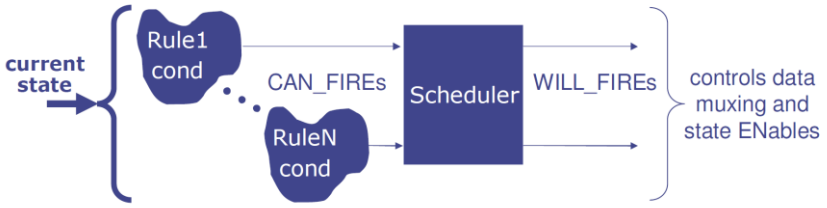
Consider each rule *in order*
If rJ does not *conflict* with earlier rules (r1..rJ-1)
Execute rJ according to the per-rule semantics
(i.e., if its CAN_FIRE is true, do its actions)

(we'll discuss conflicts shortly)

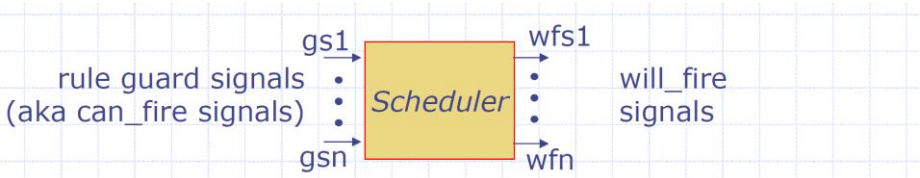


정리하자면, rule은 매 사이클 rule guard의 조건(explicit condition)을 만족하고, rule 내 메서드의 조건(implicit condition)을 만족하면서, 충돌이 없거나 있더라도 우선순위가 높을 때 최종적으로 실행이 결정됩니다. Bluespec 컴파일러는 이런 rule들의 실행을 제어하는 스케줄러 회로를 생성합니다. 스케줄러에 의해 최종적으로 실행이 결정되면 해당 rule의 WILL_FIRE가 True가 됩니다. 다음 그림은 CAN_FIRE와 WILL_FIRE의 관계를 보여줍니다.

CAN_FIRE and WILL_FIRE



- Scheduler incorporates conflict analysis by compiler
  - CAN_FIRE is True ➔(rule_condition && all_ready_conditions)
  - WILL_FIRE is True ➔ CAN_FIRE && ! (WILL_FIRE of any higher priority rules that conflict with this rule)



- Guards (gs1 ... gsn) of many rules may be true simultaneously, and some of them may conflict
- BSV compiler constructs a combinational scheduler circuit with the following property:

> for all *i* and *j*, if wfs*i* and wfs*j* are true then the corresponding gs*i* and gsj must be true and rules *i* and *j* must not conflict with each other

실행 순서(execution order)와 충돌(conflict): Method Ordering Constraints

어떤 모듈 인스턴스 x의 메서드 mA를 호출하는 rule1과 메서드 mB를 호출하는 rule2가 있다고 가정합시다. mA와 mB는 실행 순서와 관계없이 동시에 실행 가능하거나(conflict free), mA와 mB의 실행 순서에 따라 conflict 여부가 결정되거나(즉, 잠재적으로 conflict 존재), conflict가 존재해서 둘 중 하나만 실행되어야 할 수 있습니다. Bluespec 컴파일러는 이런 제약사항을 알고 있습니다. 그리고 충돌이 존재하면 컴파일러는 경고를 발생시킵니다.

Consider two rules rule1 and rule2, in that order.

Consider two method calls x.mA in rule1, and x.mB in rule2, on a common module x. These methods can be either in the condition or body or rule1 and rule2.

For every module (x), the compiler knows certain ordering constraints on its methods:

Constraint	Meaning
mA *conflict_free* mB	Rules invoking mA and mB can fire concurrently (either order)
mA < mB	Rules invoking mA and mB can fire concurrently, provided the rule invoking mA is earlier than the rule invoking mB
mB < mA	Rules invoking mA and mB can fire concurrently, provided the rule invoking mB is earlier than the rule invoking mA
mA *conflict* mB	Rules invoking mA and mB cannot fire concurrently (neither order)

레지스터 모듈 Reg의 메서드 _read와 _write의 경우를 예로 들어봅시다. 이 둘은 실행 순서에 따라 conflict가 발생할 수도 있습니다. 논리적으로 레지스터의 읽기는 쓰기보다 먼저 실행되어야 합니다. 즉, _read가 _write보다 먼저 실행되어야 합니다. 레지스터 x의 읽기(_read)를 호출하는 rl1과 쓰기(_write)를 호출하는 rl2로 구성되어 있을 때 두 rule의 실행 순서에 따라 결과가 어떻게 달라지는지 확인해봅시다. 예제에서는 실행 순서(execution order)를 인위적으로 지정하기 위해 execution_order attribute를 사용했습니다. 쓰기를 포함하는 rl2를 먼저 실행하는 경우를 먼저 보겠습니다. (예제 week5/RuleConflict2-2)

```
package Tb;

(* synthesize *)
module mkTb (Empty);
    Reg#(Bit#(16)) x <- mkReg(0);
    Reg#(Bit#(16)) y <- mkReg(0);

    (* execution_order = "rl2, rl1" *)
    rule rl1;
        $display(x._read()); // $display(x);
    endrule

    rule rl2;
        y <= y + 1;
        x._write(y._read()); // x <= y;
    endrule
endmodule

endpackage
```

이때 컴파일러는 다음과 같이 경고를 출력합니다.

```
$ make
bsc -sim -u Tb.bsv
checking package dependencies
compiling Tb.bsv
code generation for mkTb starts
Warning: "Tb.bsv", line 4, column 8: (G0010)
  Rule "rl1" was treated as more urgent than "rl2". Conflicts:
    "rl1" cannot fire before "rl2":
      earliness attribute at "Tb.bsv", line 9, column 27
    "rl2" cannot fire before "rl1": calls to x.write vs. x.read
Warning: "Tb.bsv", line 14, column 10: (G0021)
  According to the generated schedule, rule `rl2' can never fire.
Elaborated module file created: mkTb.ba
All packages are up to date.
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
bsc -verilog Tb.bsv
Warning: "Tb.bsv", line 4, column 8: (G0010)
  Rule "rl1" was treated as more urgent than "rl2". Conflicts:
    "rl1" cannot fire before "rl2":
      earliness attribute at "Tb.bsv", line 9, column 27
    "rl2" cannot fire before "rl1": calls to x.write vs. x.read
Warning: "Tb.bsv", line 14, column 10: (G0021)
  According to the generated schedule, rule `rl2' can never fire.
Verilog file created: mkTb.v
```

컴파일러는 경고를 출력합니다. rl2가 먼저 실행될 경우 Reg의 write가 read보다 먼저 실행되기 때문에 conflict가 발생하게 됩니다. 따라서 스케줄러는 우선순위가 높다고 판단하는 rl1을 실행하고, rl2는 실행하지 않습니다(이 우선순위에 대해서는 뒤에서 다루겠습니다). 실행 결과는 rl1만 실행되어 계속해서 0만 출력되는 것을 확인할 수 있습니다.

```
$ ./mkTb
    0
    0
    0
    0
    0
    0
    0
    0
    0
    0
    0
```

```
    0
    0
    0
...
```

rl1을 먼저 실행하도록 설정하는 경우에는 rl1과 rl2 모두 같은 사이클에 실행(concurrent)되는 모습을 확인할 수 있습니다. (예제 week5/RuleConflict2-1) execution_order attribute를 "rl2, rl1"에서 "rl1, rl2"로 수정해보고 다시 컴파일해봅시다.

```
$ make
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.
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
bsc -verilog Tb.bsv
Verilog file created: mkTb.v
```

Rule urgency vs. Earliness (Execution order)

앞에서는 rule의 실행 순서(execution order)에 대해 다루었습니다. 그래서 rl2를 r1보다 먼저 실행할 경우 충돌이 발생하여 둘 중 하나의 rule만 실행되고 나머지 하나는 실행되지 않는(suppressed라고 표현합니다) 결과를 확인했습니다. 이때 컴파일러 경고는 한 가지를 더 언급하고 있습니다: **Rule "rl1" was treated as more urgent than "rl2".**

그러면서 결과적으로는 rl2는 실행되지 못한다는(suppressed) 경고와 함께 rl1만 실행되는 것을 확인할 수 있습니다.

Warning: "Tb.bsv", line 14, column 10: (G0021)
  According to the generated schedule, rule `rl2' can never fire.

rl2보다 rl1을 먼저 실행하겠다고 명시했는데, rl2는 실행되지 않고 rl1만 실행된다... 이건 대체 어떤 상황일까요? 이를 알기 위해서는 **urgency**와 **execution order**를 잘 구분해야 합니다.

Bluespec에서 rule의 논리적인 실행 순서(execution order)는 이미 앞에서 다뤘습니다. rule은 여러 개가 존재할 수 있고, 레지스터의 읽기와 쓰기처럼 실행 순서에 따라 하드웨어의 일관적인 동작에 영향을 줄 수 있는 상황 역시 존재할 수 있습니다. 따라서 이전 예제와 같이 실행 순서는 rule 간의 conflict에 영향을 줄 수 있었습니다. 예제에서는 rule의 논리적인 실행 순서를 명시적으

로 지정하기 위해 execution_order라는 attribute를 사용했습니다. 그런데 컴파일러는 rl1에게 우선순위가 있다며(more urgent) rl1과 rl2 사이에 conflict가 있을 때 rl1이 실행되도록 스케줄링 했습니다. 이 '우선순위'라는 것은 과연 무엇일까요? rule에는 execution order 외에도 **urgency**라는 개념이 있습니다. urgency는 **rule 간에 충돌이 발생했을 때 어떤 rule을 실행할지 결정**을 내리기 위해 필요합니다. 만약 충돌이 발생할 때마다 어떤 사이클에서는 rl1이 실행되고, 어떤 사이클에서는 rl2가 실행된다면 일관된 하드웨어 동작을 기대하기 어려울 것입니다. (충돌이 발생했을 때) 스케줄러는 urgency가 높은 rule에게 실행권(WILL_FIRE)을 부여합니다. execution order와 마찬가지로 urgency 역시 순서를 제어할 수 있습니다. 이를 위해 사용하는 attribute가 **descending_urgency** 입니다. 이전 예제에서는 컴파일러 기본값에 의해 rl1이 rl2보다 urgency가 높아 rl1이 실행됐지만, descending_urgency를 이용하여 rl2에게 더 높은 urgency를 주면 conflict가 발생했을 때 rl1 대신 rl2가 실행됩니다. (예제 week5/RuleConflict2-3)

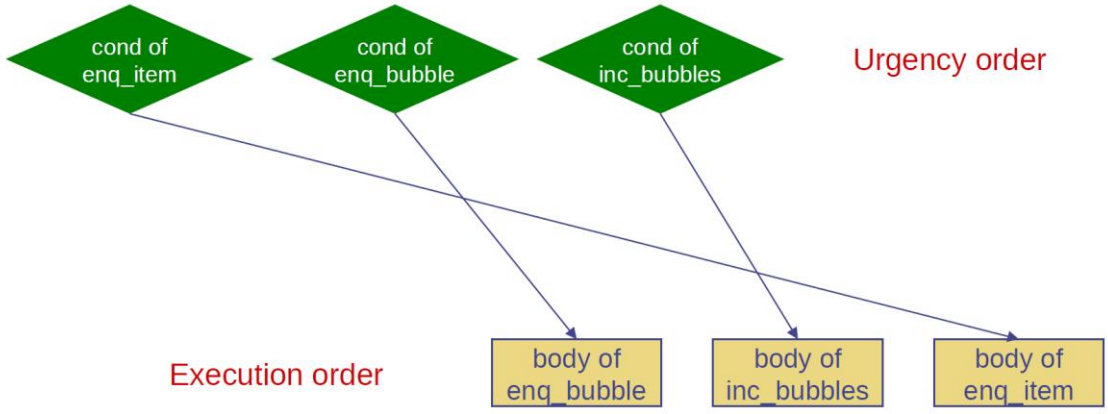
```
package Tb;

(* synthesize *)
module mkTb (Empty);
    Reg#(Bit#(16)) x <- mkReg(0);
    Reg#(Bit#(16)) y <- mkReg(0);

    (* descending_urgency = "rl2, rl1" *)
    (* execution_order = "rl2, rl1" *)
    rule rl1;
        $display("rl1: x=%d", x._read()); // $display(x);
    endrule

    rule rl2;
        $display("rl2");
        y <= y + 1;
        x._write(y._read()); // x <= y;
    endrule
endmodule

endpackage
```

execution_order가 여전히 rl2, rl1 순서로 지정되어 있기 때문에 conflict가 존재하는 건 여전하지만, 이전 예제(week5/RuleConflict2-2)와 달리 rl2의 urgency가 더 높게 부여됐기 때문에 rl1 대신 rl2의 실행이 강제됩니다.

```
$ make
bsc -sim -u Tb.bsv
checking package dependencies
compiling Tb.bsv
code generation for mkTb starts
Warning: "Tb.bsv", line 10, column 10: (G0021)
  According to the generated schedule, rule `rl1' can never fire.
Elaborated module file created: mkTb.ba
```

```
All packages are up to date.
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
bsc -verilog Tb.bsv
Warning: "Tb.bsv", line 10, column 10: (G0021)
  According to the generated schedule, rule `rl1' can never fire.
Verilog file created: mkTb.v

$ ./mkTb
rl2
rl2
rl2
rl2
rl2
rl2
…
```

정리하자면, urgency란 WILL_FIRE가(즉, 현재 사이클에서의 실행 여부) 계산되는 순서를 결정하고, execution order는 rule body를 실행하는 논리적인 순서를 결정합니다. execution order는 earliness라고도 부릅니다.



Parallelism과 Concurrency 다시 보기

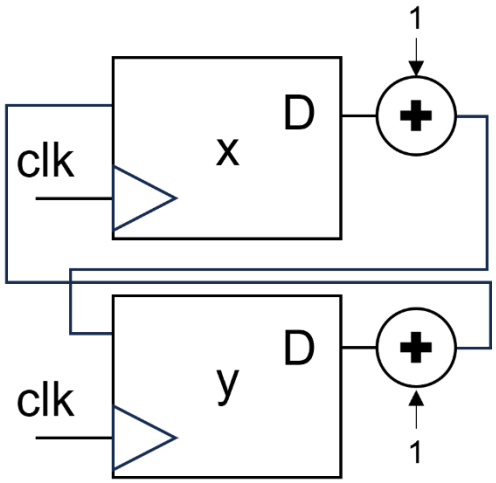
rule 안에 있는 action들은 순서라는 개념 없이 모두 병렬적으로 실행(parallelism)된다 했습니다. rule은 여러 개 존재 가능하며, 여러 rule은 논리적으로 순서를 가지고 실행(concurrency)된다고 했죠. 이번에는 같은 action으로 구성되더라도 여러 rule로 쪼개져서 conflict가 발생하는 경우를 하나 보고 넘어가겠습니다.

[week5/RuleConflict1-1]

```
package Tb;
```

```
(* synthesize *)
module mkTb (Empty);
    Reg#(Bit#(16)) x <- mkReg(0);
    Reg#(Bit#(16)) y <- mkReg(0);

    // runs atomically
    rule rlIncAndSwap;
        x <= y + 1;
        y <= x + 1;
    endrule

    rule rlDisplay;
        $display("x: %d, y: %d", x, y);
    endrule

    rule rlFinish (x > 110);
        $display("finished at ", $time);
        $finish;
    endrule
endmodule

endpackage
```

예제의 rlIncAndSwap은 두 레지스터 x와 y의 값을 1씩 증가시킨 값을 swap하여 저장합니다. 레지스터의 값은 다음 사이클이 시작할 때 업데이트 된다는 사실을 잊지 마시길 바랍니다. rlIncAndSwap은 다음 그림과 같이 x+1의 출력은 y의 입력에, y+1의 출력은 x의 입력에 연결되는 방식으로 구현될 것입니다. 그리고 rule은 action들의 기술 순서와 관계없이 atomic하게 실행될 것입니다.



만약 x <= y +1;과 y <= x + 1;을 각각 별도의 rule에 기술한다면 무슨 일이 벌어질까요?

[week5/RuleConflict1-2]

```
package Tb;

(* synthesize *)
module mkTb (Empty);
    Reg#(Bit#(16)) x <- mkReg(0);
    Reg#(Bit#(16)) y <- mkReg(0);

    //(* descending_urgency = "rlIncAndSwapY, rlIncAndSwapX" *)
    rule rlIncAndSwapX;
        // y.read
        // x.write
        x <= y + 1;
    endrule

    rule rlIncAndSwapY;
        // x.read
        // y.write
        y <= x + 1;
    endrule

    rule rlDisplay;
        $display("x: %d, y: %d", x, y);
    endrule

    rule rlFinish (x > 10);
        $display("finishec at ", $time);
        $finish;
    endrule
endmodule

endpackage
```

각 동작을 rlIncAndSwapX, rlIncAndSwapY로 분할하면 다음과 같은 경고를 출력합니다:

```
Warning: "Tb.bsv", line 4, column 8: (G0010)
  Rule "rlIncAndSwapX" was treated as more urgent than
  "rlIncAndSwapY". Conflicts:
    "rlIncAndSwapX" cannot fire before "rlIncAndSwapY":
      calls to x.write vs. x.read
    "rlIncAndSwapY" cannot fire before "rlIncAndSwapX":
      calls to y.write vs. y.read
Warning: "Tb.bsv", line 15, column 10: (G0021)
  According to the generated schedule, rule `rlIncAndSwapY' can never fire.
```

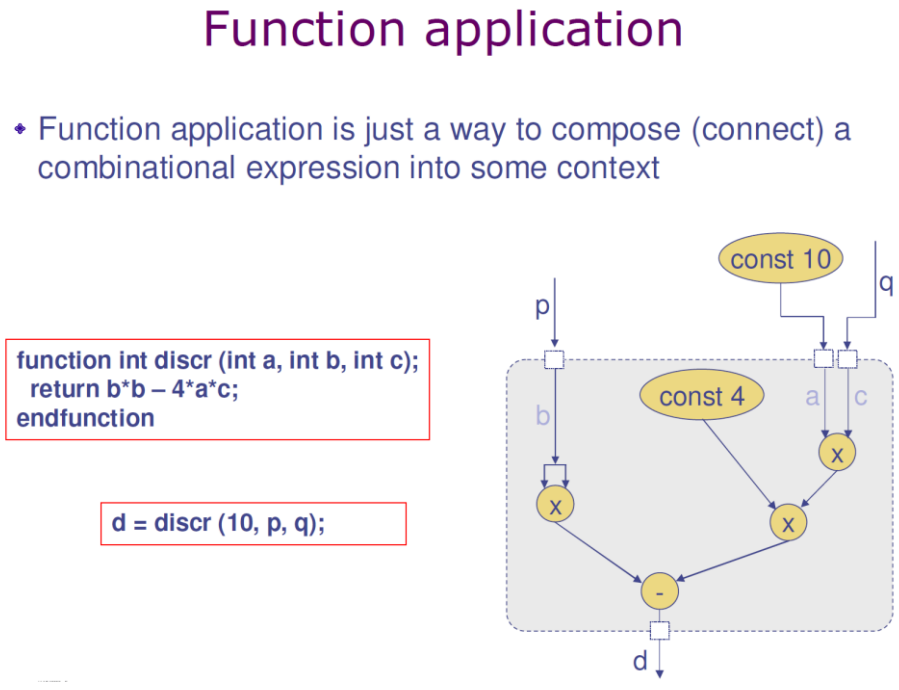
두 rule에는 '순서'가 존재하기 때문에 충돌이 생기게 됩니다. Bluespec에서 레지스터 읽기는 Reg._read() 메서드로, 쓰기는 Reg._write() 메서드로 구현된다고 말씀드린 바 있습니다. 그리고,

_read가 _write보다 먼저 호출되어야 함을 확인했죠. rlIncAndSwapX에서는 y.read와 x.write를 호출합니다. rlIncAndSwapY에서는 x.read와 y.write를 호출합니다. 두 rule은 어떤 순서로 실행되더라도 충돌을 발생시킵니다. rlIncAndSwapX를 먼저 실행할 경우에는 x.write를 x.read보다 먼저 호출하기 때문에, rlIncAndSwapY를 먼저 실행할 경우에는 y.write를 y.read보다 먼저 호출하기 때문에 충돌입니다. 충돌이 발생하기 때문에 두 rule 중 하나만 실행되어야 합니다. 이때 urgency가 더 높은 rule이 실행될 것입니다. 컴파일러는 기본값으로 rlIIncAndSwapX를 실행하고 rlIncAndSwapY는 suppress하게 됩니다. 반대로 rlIncAndSwapY만 실행하고 싶다면 rule 상단에 있는 descending_urgency attribute의 주석을 해제해봅시다.

IncAndSwap은 논리적으로도 순서를 부여하는 순간에 문제가 발생합니다. x <= y +1을 먼저 실행할 경우에는 x=0, y=0에서 시작하므로, x = 0+1, y = 1+1 순으로 실행되어 x=1, y=2가 되고, 반대로 y <= x + 1을 먼저 실행할 경우에는 y = 0 + 1, x = 1 + 1 순으로 실행되어 x=2, y=1이 됩니다.

기타 (1): Action function

function은 메서드와 마찬가지로 '호출'할 수 있습니다. 하지만 function은 인터페이스 정의의 일부가 아닙니다. 즉, 메서드 선언은 인터페이스에 들어가고, 메서드의 구현은 인터페이스를 제공(provide)하는 모듈에 들어가지만, function은 이와 독립적입니다. function은 모듈 밖에서 정의될 수도 있고, 안에서 정의될 수도 있습니다. 그리고 function은 흔히 side-effect가 없는 조합회로로 구현된다는 특징이 있습니다. 그런데 사실 function 역시 Action 메서드와 마찬가지로 Action(즉, side-effect)을 가질 수 있습니다. 이를 action function이라고 합니다.



rule은 일련의 action들로 구성된다고 수차례 언급했습니다. 따라서, function이 어떤 값 대신에 action을 반환하고, rule 안에서 function으로부터 반환된 action을 실행하는 것은 그렇게 이상한 일이 아닙니다. 하지만 메서드를 두고 굳이 action function을 사용해야 할 이유가 있을까요? 이는 function이 메서드와 달리 인터페이스와 모듈로부터 독립적이라는 특징에서 찾을 수 있습니다. 예를 들어, 여러 타입의 모듈에 대한 반복적인 action을 수행할 필요가 있을 때 매번 코드를 작성하는 대신 function을 사용하면 아주 편리하면서도 코드 중복도 제거할 수 있습니다. 우리가 어떤 반복적인 작업을 수행하기 위해 함수 등으로 불리는 서브루틴을 사용하는 것과 같습니다. 여러 타입에 대해서 polymorphic하게 동작할 수 있다는 점도 특징입니다. 이처럼 action function을 사용하면 parameterized된 action을 기술할 수 있습니다.

[week5/ActionFunction]

```
package Tb;

function Action regUpdate(Reg#(t) r) provisos (Bits#(t, sizea), Arith#(t));
    action
        r <= r + 1;
    endaction
endfunction

(* synthesize *)
module mkTb (Empty);
    Reg#(Bit#(16)) x <- mkReg(0);
    Reg#(Bit#(32)) y <- mkReg(0);

    function Action displayRegs();
        action
            $display("x=%d, y=%d", x, y);
        endaction
    endfunction

    (* execution_order = "rlEnd, rlUpdateAndDisplay" *)
    rule rlUpdateAndDisplay;
        regUpdate(x);
        regUpdate(y);
        displayRegs();
    endrule

    rule rlEnd (x > 10);
        $display("rlEnd at", $time);
        $finish;
    endrule
endmodule

endpackage
```

기타 (2): Parameter

Verilog 와 마찬가지로 BSV에서도 parameter를 지원합니다. 지금까지 레지스터를 instantiate할 때 사용했던 mkReg(0) 역시 mkReg 모듈의 parameter입니다. 앞서 다룬 모든 예제에서 직접 작성한 모듈을 instantiate할 때 이런 초기값 등을 지정할 수 없었던 이유는 parameter를 별도로 정의하지 않았기 때문입니다. parameter를 정의하는 방법은 간단합니다. 예제를 통해 확인해봅시다.

[week5/Parameter]

```
package Tb;

interface M1_IFC;
    method int getX();
    method Action incX();
endinterface

(* synthesize *)
module mkM1 #(parameter int init_val) (M1_IFC);
    Reg#(int) x <- mkReg(init_val);
    method int getX = x;
    method Action incX();
        x <= x + 1;
    endmethod
endmodule

(* synthesize *)
module mkTb (Empty);
    M1_IFC m1i1 <- mkM1(2);
    M1_IFC m1i2 <- mkM1(5);

    rule i1Inc (m1i1.getX <= m1i2.getX);
        m1i1.incX();
    endrule

    rule i2Inc (m1i1.getX > m1i2.getX);
        m1i2.incX;
    endrule

    rule m1Display;
        let i1x = m1i1.getX;
        let i2x = m1i2.getX();
        $display("m1i1.x: %d, m1i2.x: %d", i1x, i2x);
        if((i1x == i2x) && (i1x == 20)) begin
            $display("finished at ", $time);
            $finish;
        end
    endrule
```

```
endmodule

endpackage
```

모듈 mkM1에 init_val이라는 parameter를 선언한 것을 확인할 수 있습니다. 인터페이스는 그 다음에 들어갑니다. mkM1은 x라는 레지스터를 가지고 있고, init_val로 지정된 값으로 초기화됩니다. mkM1의 두 인스턴스 m1i1, m1i2는 각각 2와 5에서 출발하여 값이 증가되며 둘 다 20이 되었을 때 시뮬레이션을 종료합니다.

```
$ ./mkTb
x=    0, y=    0
x=    1, y=    1
x=    2, y=    2
x=    3, y=    3
x=    4, y=    4
x=    5, y=    5
x=    6, y=    6
x=    7, y=    7
x=    8, y=    8
x=    9, y=    9
x=   10, y=   10
rlEnd at                120
```

References

Bluespec Compiler Libraries Reference Guide (bsc-doc/bsc_libraries_ref_guide.pdf)

BSV by Example (textbook/bsv_by_example.pdf)

BSV Training

https://github.com/rsnikhil/Bluespec_BSV_Tutorial

- Lec_Basic_Syntax (tutorial-reference/Lec02_Basic_Syntax.pdf)
- Lec_Rule_Semantics (tutorial-reference/Lec03_Rule_semantics.pdf)

UCSB Lecture Notes

https://web.ece.ucsb.edu/its/bluespec/index.html

- Lecture 04: Module Hierarchy (lecnotes-arvind/Lec04_Module_Hierarchy.pdf)
- Lecture 05: Rules (lecnotes-arvind/Lec05_Rules.pdf)
- Lecture 06: Rules (lecnotes-arvind/Lec06_Scheduling.pdf)