DataLab. Internship Program (2023 Summer)

Week 4

Tutor: 배은태

예제

StmtFSM1-1

BSV에서는 StmtFSM이라는 라이브러리 패키지를 제공합니다. BSV에서 이 StmtFSM은 구조적(structured) FSM을 기술하는 아주 강력한 도구입니다. 기존에는 소프트웨어 코드의 실행 흐름처럼 순차적으로 동작하는 FSM을 설계하기 위해서 state를 저장하는 레지스터를 두고 각각의 state에 해당하는 rule을 명시적으로 정의해야 했습니다. 이런 명시적인 FSM의 설계는 지난 시간에 FSM 예제를 다루면서 공부한 바 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
```

```
$ ./mkTb
Pre-Cracked egg
Saves time!
No messy hand
I enjoy
Fimally!
```

보다시피 하드웨어에서는 지극히 단순한 동작일지라도 순차적으로 실행되도록 FSM으로 구성하려면 다소 번거롭게 코드를 작성해야 했습니다. 이건 메모리에 올라가서 CPU에 의해 실행되는 소프트웨어 코드가 아니라 하드웨어 회로로 합성되는 하드웨어 코드이기 때문입니다. 하지만 StmtFSM을 이용하면 순차적으로 동작하는 일련의 action들을 아주 간편하게 작성할 수 있습니다. 이제 기존의 명시적인 FSM을 StmtFSM으로 대체해보겠습니다.

```
package Tb;
import StmtFSM::*;
(* synthesize *)
module mkTb (Empty);
    Reg#(Bool) complete <- mkReg(False);

    Stmt egg = seq
        $display("Pre-Cracked egg");
        $display("Saves time!");
        $display("No messy hand");
        $display("I enjoy");
        $display("Fimally!");
        complete <= True;
    endseq;

    FSM eggFSM <- mkFSM(egg);

    rule startit;
        eggFSM.start();
    endrule

    rule alwaysrun;
        $display("\t\tand a rule fires at", $time);
        if (complete)
            $finish(0);
    endrule
endmodule
endpackage
```

Stmt egg 다음에 있는 seq-endseq 블록으로 묶인 각각의 action들은 매 사이클마다 하나씩 순차적으로 실행됩니다. 실행 확인을 좀 더 명확히 하고, 실행 종료를 제어하기 위해 매 사이클마다 실행되는 또 다른 rule인 alwaysrun을 추가했습니다.

```
$ ./mkTb
               and a rule fires at                10
               and a rule fires at                20
Pre-Cracked egg
               and a rule fires at                30
Saves time!
               and a rule fires at                40
No messy hand
               and a rule fires at                50
I enjoy
               and a rule fires at                60
Fimally!
               and a rule fires at                70
               and a rule fires at                80
```

StmtFSM1-2

이전 예제에서는 seq 블록 안에 있는 문장(action)들이 한 사이클에 하나씩 실행됐습니다. 하지만 여러 레지스터를 한 사이클 동안 동시에 업데이트 하는 등 여러 action을 동시에 수행해야 하는 경우에는 어떻게 해야 할까요? 이번 예제에서는 action-endaction 블록을 활용하여 여러 action을 한 사이클에 동시에 실행하도록 회로를 설계하는 방법을 소개합니다.

```
package Tb;
import StmtFSM::*;

(* synthesize *)
module mkTb (Empty);
    Reg#(Bool) complete <- mkReg(False);

    Stmt egg = seq
        $display("Pre-Cracked egg");
        // an action with several actions still only take one cycle
        action
            $display("Saves time!");
            $display("No messy hand");
            $display("I enjoy");
        endaction
        $display("Fimally!");
        complete <= True;
    endseq;

    FSM eggFSM <- mkFSM(egg);
```

```
    rule startit;
        eggFSM.start();
    endrule

    rule alwaysrun;
        $display("\t\tand a rule fires at", $time);
        if (complete)
            $finish(0);
    endrule
endmodule

endpackage
```

```
$ ./mkTb
                and a rule fires at             10
                and a rule fires at             20
Pre-Cracked egg
                and a rule fires at             30
Saves time!
No messy hand
I enjoy
                and a rule fires at             40
Fimally!
                and a rule fires at             50
                and a rule fires at             60
```

StmtFSM1-1과 1-2의 실행 결과를 서로 비교해봅시다.

StmeFSM2

action들의 시퀀스를 시작할 때 자동으로, 단 한 번만 실행하고 싶은 경우가 있을 수도 있습니다. 특히 테스트벤치를 실행할 때 흔히 발생하는 경우인데요, Bluespec 라이브러리에서는 AutoFSM이라는 것도 제공합니다. AutoFSM에 있는 시퀀스는 자동으로 실행되며, 실행을 마치면 $finish 시스템 태스크를 호출합니다.

```
package Tb;
import StmtFSM::*;

(* synthesize *)
module mkTb (Empty);
```

```
    Stmt egg = seq
        $display("Pre-Cracked egg");
        action
            $display("Saves time!");
            $display("No messy hand");
            $display("I enjoy");
        endaction
        $display("Fimally!");
    endseq;

    mkAutoFSM(egg);

    rule alwaysrun;
        $display("\t\tand a rule fires at", $time);
    endrule
endmodule

endpackage
```

Array

BSV는 Verilog와 마찬가지로 배열을 지원합니다. 다차원 배열도 구성이 가능하며, 차원 개수에는 제한이 없습니다. 이번 예제에서는 1차원 배열만 다루겠습니다. 배열은 C언어와 마찬가지로 대괄호와 인덱스로 접근합니다.

또한 레지스터 배열 원소들은 같은 사이클에 동시 접근이 가능합니다. rule update의 for문은 arr1d[0]부터 arr1d[4]까지의 레지스터를 동시에 업데이트하는 코드임에 주의하세요. Verilog나 BSV에서의 반복문은 소프트웨어 코드처럼 반복적으로 동작하는 게 아니라 단지 여러 줄로 컴파일될 뿐입니다. rule 안에 있는 action들은 한 사이클 동안 동시에 수행된다는 사실도 기억해야 합니다.

```
package Tb;

(* synthesize *)
module mkTb (Empty);
    Reg#(Bit#(16)) arr1d[5];
    /* statically elaborate */
    for (Integer i = 0; i < 5; i = i + 1)
        arr1d[i] <- mkReg(0);

    Reg#(Bit#(8)) step <- mkReg(0);

    rule display;
        $display("[%d] arr1d: %d %d %d %d %d",
            step, arr1d[0], arr1d[1], arr1d[2], arr1d[3], arr1d[4]);
        // $display("arr2d: %d %d %d, %d %d %d",
```

```
        //     arr2d[0][0], arr2d[0][1], arr2d[0][2],
        //     arr2d[1][0], arr2d[1][1], arr2d[1][2]);
        if (step > 10)
            $finish;
    endrule

    rule update;
        for (Integer i = 0; i < 5; i = i + 1) begin
            arr1d[i] <= arr1d[i] + fromInteger((i + 1) * 10);
        end
        step <= step + 1;
    endrule
endmodule

endpackage
```

Vector

벡터는 배열과 비슷하지만 훨씬 강력한 기능을 제공합니다. 벡터는 Bluespec 라이브러리에서 제공하는 패키지입니다. 따라서 벡터를 사용하려면 Vector 패키지를 import해야 합니다. BSV 라이브러리는 벡터를 위한 별도의 초기화 방법을 제공합니다. 이번 예제에서는 헷갈릴 수 있는 주제인 '벡터의 레지스터'와 '레지스터의 벡터'를 비교합니다. 전자는 벡터라는 자료구조를 저장하는 레지스터이고, 후자는 레지스터를 벡터로 둔 형식입니다. 이 둘의 차이를 명확히 알아야 합니다. 초기화 방식부터 다릅니다. 벡터라는 데이터를 저장하는 레지스터는 벡터 데이터의 초기화 값을 생성하는 replicate를 사용하고, 레지스터 벡터를 초기화할 때는 replicateM을 사용합니다. 벡터 레지스터를 업데이트할 때는 한 사이클에 벡터 전체를 업데이트 해야 합니다. 벡터의 레지스터인 vecreg을 업데이트하기 위해 rlUpdate에서 vec이라는 임시변수를 사용했음을 주목하세요. 반면에 regvec은 배열과 마찬가지로 벡터의 각 원소들을 동시에 접근할 수 있습니다. 업데이트도 regvec[0], regvec[1], regvec[2] 이 세 8비트 정수 레지스터에 대해서만 적용했음을 주목해주세요. 그 외의 자세한 내용은 교과서인 BSV by Example과 Bluespec Compiler Libraries Reference Guide를 참고하시기를 바랍니다.

```
package Tb;

import Vector::*;

(* synthesize *)
module mkTb (Empty);
    // Vector#(number of items, type of items)
    Reg#(Vector#(5, Bit#(8))) vecreg <- mkReg(replicate(0));    // register of
vector
    Vector#(5, Reg#(Bit#(8))) regvec <- replicateM(mkReg(0));   // vector of
register
```

```
    Reg#(Bit#(8)) step <- mkReg(0);

    rule rlDisplay;
        $display("vecreg: %10x", vecreg);
        $display("regvec: %x%x%x%x%x\n",
            regvec[0], regvec[1], regvec[2], regvec[3], regvec[4]
        );
        if (step > 10)
            $finish;
    endrule

    rule rlUpdate;
        let vec = vecreg;
        vec[0] = vec[0] + 8'h10;
        vec[3] = vec[3] + 8'h01;
        vec[4] = vec[4] + 8'h05;
        vecreg <= vec;

        regvec[0] <= regvec[0] + 8'h02;
        regvec[1] <= regvec[1] + 8'h03;
        regvec[2] <= regvec[2] + 8'h04;

        step <= step + 1;
    endrule
endmodule

endpackage
```

FIFO

FIFO 역시 Vector와 마찬가지로 Bluespec 라이브러리에서 제공합니다. FIFO는 선입선출(First-In First-Out) 방식의 버퍼입니다. FIFO는 크게 버퍼에 새로운 원소를 집어넣는 enq와 맨 앞의 원소를 버퍼에서 제거하는 deq 연산을 제공합니다. 그 외에도 맨 앞의 원소를 읽을 수 있는 first를 제공합니다. FIFOF라는 것이 별도로 존재하는데, FIFO와는 notFull과 notEmpty 메서드를 명시적으로 제공한다는 점이 다릅니다.

```
import FIFO::*;
    interface FIFO #(type element_type);
    method Action enq(element_type x1);
    method Action deq();
    method element_type first();
    method Action clear();
endinterface: FIFO

import FIFOF::*;
```

```
interface FIFOF #(type element_type);
    method Action enq(element_type x1);
    method Action deq();
    method element_type first();
    method Bool notFull();
    method Bool notEmpty();
method Action clear();
endinterface: FIFOF
```

FIFO에 대한 자세한 내용은 마찬가지로 라이브러리 레퍼런스를 참고하시길 바랍니다.

```
package Tb;

import FIFOF::*;

(* synthesize *)
module mkTb (Empty);
    FIFOF#(Bit#(8)) myFifo <- mkSizedFIFOF(20);
    Reg#(Bit#(8)) cnt <- mkReg(10);
    Reg#(Bool) pushComplete <- mkReg(False);

    //(* mutually_exclusive = "rlPush, rlPop" *)
    rule rlPush (!pushComplete);
        if (cnt == 0) begin
            $display("push complete!");
            pushComplete <= True;
        end
        else begin
            Bit#(8) idx = 10 - cnt;
            Bit#(8) inputValue = (idx + 1) * 10;
            $display("push[%3d]: %d", idx, inputValue);
            myFifo.enq(inputValue);
            cnt <= cnt - 1;
        end
    endrule

    rule rlPop (pushComplete);
        $display("pop[%4d]: %d", cnt, myFifo.first);
        myFifo.deq;
        cnt <= cnt + 1;
    endrule

    rule rlFinish (pushComplete && !myFifo.notEmpty);
        $display("finished at", $time);
        $finish;
    endrule
endmodule
```

```
endpackage
```

GCDEngine

이전에 실습한 GCD 모듈을 활용하여 request 큐에 request가 들어오면 자동으로 GCD 연산을 수행하여 response 큐에 결과를 저장하는 GCD 엔진을 만들어봅시다. 예제에서 mkTb는 producer-consumer 방식으로 GCD 엔진에 request를 넣는 rule rlProduce와 response를 읽어서 출력하는 rlConsume으로 구성되어 있습니다.

Request/Response 큐인 FIFO의 버퍼 크기에 변경해가면서 결과를 비교해봅시다.

```
package Tb;
import FIFOF::*;

typedef struct {
    Bit#(8) id;
    Bit#(32) x;
    Bit#(32) y;
} Req deriving (Bits, Eq);

typedef struct {
    Bit#(8) id;
    Bit#(32) result;
} Resp deriving (Bits, Eq);

interface GCD_IFC;
    method Action put(Req req);
    method ActionValue#(Resp) get();
    method Bool isRespQEmpty();
endinterface

(* synthesize *)
module mkGCD (GCD_IFC);
    FIFOF#(Req) reqQ <- mkSizedFIFOF(20);
    FIFOF#(Resp) respQ <- mkSizedFIFOF(20);

    Reg#(Bit#(8)) reqId <- mkReg(0);
    Reg#(Bit#(32)) x <- mkReg(0);
    Reg#(Bit#(32)) y <- mkReg(0);
    Reg#(Bool) busy <- mkReg(False);

    (* descending_urgency = "fetch, put" *)
    (* mutually_exclusive = "fetch, step1, step2" *)
    rule fetch (reqQ.notEmpty() && !busy);
        Req req = reqQ.first();
        reqQ.deq;
```

```
        reqId <= req.id;
        x <= req.x;
        y <= req.y;
        busy <= True;
        $display("\tfetch: ID=%d, x=%d, y=%d at %t", req.id, req.x, req.y, $time);
    endrule

    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
            Resp resp;
            resp.id = reqId;
            resp.result = x;
            respQ.enq(resp);
            busy <= False;
            //$display("\tcomplete: id=%d, result=%d, at %t", resp.id,
resp.result, $time);
        end
        //$display("step2: %d, %d", x ,y);
    endrule

    method Action put(Req req) if (reqQ.notFull());
        //$display("put: ID=%d, x=%d, y=%d at %t", req.id, req.x, req.y, $time);
        reqQ.enq(req);
    endmethod

    method ActionValue#(Resp) get() if (respQ.notEmpty());
        respQ.deq;
        return respQ.first;
    endmethod

    method Bool isRespQEmpty();
        return !respQ.notEmpty();
    endmethod
endmodule

(* synthesize *)
module mkTb (Empty);
    GCD_IFC gcdEngine <- mkGCD;
    Reg#(Bit#(8)) curId <- mkReg(0);
    Reg#(Bit#(8)) completeCnt <- mkReg(0);
```

```
    rule rlProduce;
        Req req;
        req.id = curId;
        req.x = zeroExtend(curId) * 5 + 72;
        req.y = zeroExtend(curId) * 4 + 21;
        gcdEngine.put(req);
        curId <= curId + 1;
        $display("put: ID=%d, x=%d, y=%d at %t", req.id, req.x, req.y, $time);
    endrule

    rule rlConsume;
        let resp <- gcdEngine.get();
        $display("\tget: ID=%d, result=%d at %t", resp.id, resp.result, $time);
        completeCnt <= completeCnt + 1;
    endrule

    rule rlFinish (completeCnt > 10 && gcdEngine.isRespQEmpty);
        $display("finished at ", $time);
        $finish;
    endrule
endmodule

endpackage
```

References

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

BSV by Example(textbook/bsv_by_example.pdf)