Stmt FSM

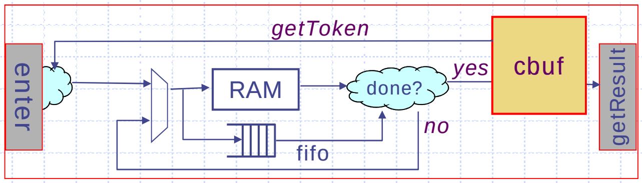
Arvind (with the help of Nirav Dave)
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

Motivation

- Some common design patterns are tedious to express in BSV
  - Testbenchs
  - Sequential machines (FSMs)
    - especially sequential looping structures

These are tedious to express in Verilog as well (but not in C)

Testing the IP Lookup Design



- Input: IP Address
- Output: Route Value

- Need to test many different input/output sequences

Testing IP Lookup

- Call many streams of requests responses from the device under test (DUT)

Case 1 (Check correct with 1 request at a time)	Case 2 (Check correct with 2 concurrent requests)
dut.enter(17.23.12.225)	dut.enter(128.30.90.124)
dut.getResult()	dut.enter(128.30.90.126)
dut.enter(17.23.12.25)	dut.getResult()
dut.getResult()	dut.getResult()

But we usually want more

counters, display, …

```
function Action makeReq(x);
 action
  reqCnt <= reqCnt + 1;
  dut.enter(x);
  $display(“[Req #: ”,fshow(reqCnt),“] = ”,fshow(x));
 endaction
endfunction

function Action getResp();
 action
   resCnt <= resCnt + 1;
   let x <- dut.getResult();
   $display(“[Rsp #:”,fshow(resCnt),“] = ”,fshow(x));
 endaction
endfunction
```


Writing a Testbench (Case 1)

```
rule step0(pos==0);
  makeReq(17.23.12.225);
  pos <= 1;
endrule

rule step1(pos==1);
  getResp();
  pos <= 2;
endrule
```

Wait until response is ready

```
rule step2(pos==2);
  makeReq(17.23.12.25);
  pos <= 3;
endrule

rule step3(pos==3);
  getResp();
  pos <= 4;
endrule

rule finish(pos==4);
  $finish;
endrule
```

A more complicated Case:

Initializing memory

Need an FSM in HW as memory can only do one write per cycle

C

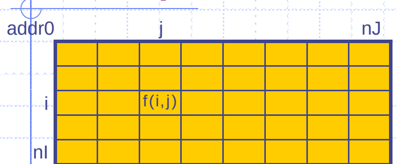
```
int i; Addr addr=addr0;
bool done = False;
for(i=0; i<nI; i++){
 mem.write(addr++,f(i));
}
done = True;
```

BSV

```
Reg#(int)  i    <-mkReg(0);
Reg#(Addr) addr <-mkReg(addr0);
Reg#(Bool) done <-mkReg(False);

rule initialize (i < nI);
  mem.write (addr, f(i));
  addr <= addr + 1;
  i <= i + 1;
  if (i+1 == nI) done<=True;
endrule
```


Initialize a memory with a 2-D pattern



- Bluespec code gets messier as compared to C even with small changes in C, e.g.,
  - initialization based on old memory values
  - initialization has to be done more than once

```
Reg#(int) i     <-mkReg(0);
Reg#(int) j     <-mkReg(0);
Reg#(Addr) addr <-mkReg(addr0);
Reg#(Bool) done <-mkReg(False);

rule loop ((i < nI) && (j < nJ));
  mem.write (addr, f(i,j));
  addr <= addr + 1;
  if (j < nJ-1)
    j <= j + 1;
  else begin
    j <= 0;
    if (i < nI-1) i <= i + 1;
     else       done <= True;
  end
endrule
```

An imperative view

It is easy to write a sequence in C

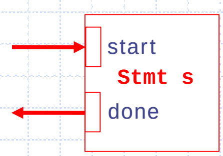
Writing this in rules is tedious:

Can we just write the actions and have the compiler make the rules?

```
void doTest(){
  makeReq(17.23.12.225);
  getResp();
  makeReq(17.23.12.25);
  getResp();
  exit(0);
}
```

```
seq
  makeReq(17.23.12.225);
  getResp();
  makeReq(17.23.12.25);
  getResp();
  $finish();
endseq;
```


From Action Lists to FSMs



- FSM interface

```
interface FSM;
    method Action  start();
    method Bool    done();
endinterface
```

- Creating an FSM

```
module    mkFSM#(Stmt s)(FSM);
```

The Stmt Sublanguage

- Stmt =

```
    <Bluespec Action>
  | seq s1..sN endseq
  | par s1..sN endpar
  | if-then / if-then-else
  | for-, while-, repeat(n)-
          (w/ break and continues)
```


Translation Example:
Seq to FSM

```
Stmt s = seq
  makeReq(17.23.12.225);
  getResp();
  makeReq(17.23.12.25);
  getResp();
  $finish();
endseq;

FSM f <- mkFSM(s);
```

```
module mkFSM_s(FSM)
 Reg#(Bit#(3)) pos <- mkReg(0);
 rule step1(pos==1);
  makeReq(17.23.12.225); pos <= 2;
 endrule
 rule step2(pos==2);
  getResp(); pos <= 3; endrule
 rule step3(pos==3);
  makeReq(17.23.12.25); pos <= 4;
 endrule
 rule step4(pos==4);
  getResp(); pos <= 5; endrule
 rule step5(pos==5);
  $finish; pos <= 0; endrule
 method Action start() if(pos==0);
  pos <= 1;
 endmethod
 method Bool done()
  return (pos == 0);
 endmethod
endmodule
```

Parallel Tasks

```
seq
  refReq(x);
  refRes(rReg);
  dutReq(x);
  dutRes(dReg);
 checkMatch(rReg,dReg);
endseq
```

- We want to check dut and ref have same result
- Do each, then check results

But it doesn't matter that ref finishes before dut starts…

Start ref and dut at the same time

- Seq. for each implementation
- Start together
- Both run at own rate
- Wait until both are done

```
seq
 par
  seq  refReq(x);
       refRes(refv);endseq
  seq  dutReq(x);
       dutRes(dutv); endseq
 endpar
 checkMatch(refv,dutv);
endseq
```

What exactly is the translation?

- The Stmt sublanguage is clearer for the designer; but, what FSM do we get?
- Let's examine each Stmt Construction case and see how it can be implemented

Base Case: Primitive Action: **a**

```
Reg#(Bool) doneR <- mkReg(True);

rule dowork(!doneR);
  a;
  doneR <= True;
endrule

method Action start() if (doneR);
   doneR <= False;
endmethod

method Bool done(); return doneR; endmethod
```

Sequential List - *seq*

- *seq s1…sN endseq*: sequential composition

```
Reg#(int)  s    <-mkReg(0);
FSM s1 <- mkFSM (s1); … ; FSM sN <- mkFSM (sN);
Bool flag = s1.done() && … sN.done();

rule one (s==1); s1.start(); s <= 2; endrule
rule two (s==2&& s1.done());
                 s2.start(); s <= 3; endrule
…
rule n   (s==n && sN-1.done());
                  sN.start(); s <= 0; endrule

method Action start() if (flag); s <= 1; endmethod
method Bool done(); return flag; endmethod
```


Implementation - *par*

- *par s1…sN endpar*: parallel composition

```
FSM s1 <- mkFSM (s1); … ; FSM sN <- mkFSM (sN);

Bool flag = s1.done() && … && sN.done();

method Action start() if (flag);
   s1.start(); s2.start(); …; sN.start();
endmethod

method Bool done(); return flag; endmethod
```

Implementation - *if*

- *if p then sT else sF*: conditional composition

```
FSM sT <- mkFSM (sT); FSM sF <- mkFSM (sF);

Bool flag = sT.done()&& sF.done();

method Action start() if (flag);
  if (p) then sT.start() else sF.start();
endmethod

method Bool done(); return flag; endmethod
```


Implementation - *while*

- *while p do s*: loop composition

```
s <- mkFSM(s);
Reg#(Bool) busy <- mkReg(False);
Bool flag =   !busy;

rule restart_loop(busy && s.done());
  if (p) begin s.start(); busy <= True;
    else busy <= False;
endrule

method Action start() if (flag);
  if (p) begin s.start(); busy <= True;
    else busy <= False;
endmethod
method Bool done(); return flag; endmethod
```

The StmtFSM library

- This **IS** the Library (almost)
  - Some optimizations for seq/base case
  - Stmt syntax added for readability
- Good but not great HW (users can do better by handcoding)
  - state-encoding
    - Use a single wide register (i,j) instead of two
    - Use 1 log(n)-bit register instead of n 1-bit registers
    - See if state can be inferred from other data registers
  - Unnecessary dead cycles can be eliminated

FSM atomicity

- FSM Actions are made into rules
  - rule atomicity governs statement interactions

```
Stmt s1 = seq
  action f1.enq(x); f2.enq(x); endaction
  action f1.deq();   x<=x+1;   endaction
  action f2.deq();   y<=y+1;   endaction
endseq;
```

```
rule s1(…); f1.enq(x);
     f2.enq(x); …;endrule
rule s2(…); f1.deq();
     x<=x+1; … endrule
rule s3(…); f2.deq();
     y<=y+1; … endrule
```

```
Stmt s2 = seq
 action f1.enq(y); f2.enq(y);
endaction
 action f1.deq(); $display("%d", y);
endaction
 action f2.deq(); $display("%d", x);
endaction
endseq;
```

```
rule s1(…); f1.enq(y);
     f2.enq(y); …endrule
rule s2(…); f1.deq();
$display("%d", y);…endrule
rule s3(…); f2.deq();
     y<=y+1; …endrule
```

FSM Atomicity

- We're writing actions, not rules
  - Do they execute atomically?
- Seq. Stmt
  - Only one at a time
  - ⇒
- Par. Stmt
  - all at once
  - ⇒

```
par x <= x + 1;
    x <= 2 * x;
    x <= x ** 2;
endpar
```

What happens here?

FSM summary

- Stmt sublanguage captures certain common and useful FSM idioms:
  - sequencing, parallel, conditional, iteration
- FSM modules automatically implement Stmt specs
- FSM interface permits composition of FSMs
- Most importantly, *same Rule semantics*
  - Actions in FSMs are atomic
  - Actions automatically block on implicit conditions
  - Parallel actions, (in the same FSM or different FSMs) automatically arbitrated safely (based on rule atomicity)