CS152: Computer Systems Architecture
A Very Short Introduction to Bluespec

Sang-Woo Jun
Winter 2022

Large amount of material adapted from MIT 6.004, “Computation Structures”, Morgan Kaufmann “Computer Organization and Design: The Hardware/Software Interface: RISC-V Edition”, and CS 152 Slides by Isaac Scherson
Bluespec System Verilog (BSV) High-Level

- Everything organized into “Modules”
  - Modules have an “interface” which other modules use to access state
  - A Bluespec model is a single top-level module consisting of other modules, etc

- Modules consist of state (other modules) and behavior
  - State: Registers, FIFOs, RAM, ...
  - Behavior: Rules, Interface

Module A

Module B

Module C1

Module C2
Peek into a RISC-V processor in Bluespec

**Processor.bsv**

```haskell
interface ProcessorIfc;
  > method ActionValue#(MemReq32) memReq;
  > method Action memResp(Word data);
endinterface

module mkProcessor(ProcessorIfc);
  > Reg#(Word) pc <- mkReg(0);
  > RFile2R1W rf <- mkRFile2R1W;
  > MemorySystemIfc mem <- mkMemorySystem;
  > rule doFetch (stage == Fetch);
  >   let next_pc = pc + 4;
```

**Top.bsv**

```haskell
module mkTop(Empty);
  > ProcessorIfc proc <- mkProcessor;
```

...
Greatest Common Divisor Example

- Euclid’s algorithm for computing the greatest common divisor (GCD)

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>6</td>
<td>subtract</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>subtract</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>swap</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>subtract</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>subtract</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>subtract</td>
</tr>
</tbody>
</table>

answer
module mkGCD (GDCIfc);
    Reg#(Bit#(32)) x <- mkReg(0);
    Reg#(Bit#(32)) y <- mkReg(0);
    FIFOF#(Bit#(32)) outQ <- mkSizedFIFOF(2);
rule step1 ((x > y) && (y != 0));
    x <= y; y <= x;
endrule
rule step2 (( x <= y) && (y != 0));
    y <= y-x;
    if ( y-x == 0 ) begin
        outQ.enq(x);
    end
endrule
method Action start(Bit#(32) a, Bit#(32) b) if (y==0);
    x <= a; y <= b;
endmethod
method ActionValue#(Bit#(32)) result();
    outQ.deq;
    return outQ.first;
endmethod
endmodule
module mkGCD (GDCIfc);
Reg#(Bit#(32)) x <- mkReg(0);
Reg#(Bit#(32)) y <- mkReg(0);
FIFOF#(Bit#(32)) outQ <- mkSizedFIFOF(2);
rule step1 ((x > y) && (y != 0));
  x <= y; y <= x;
endrule
rule step2 (( x <= y) && (y != 0));
  y <= y-x;
  if ( y-x == 0 ) begin
    outQ.enq(x);
  end
endrule
method Action start(Bit#(32) a, Bit#(32) b) if (y==0);
  x <= a; y <= b;
endmethod
method ActionValue#(Bit#(32)) result();
  outQ.deq;
  return outQ.first;
endmethod
endmodule
module mkGCD (GDCIfc);
    Reg#(Bit#(32)) x <- mkReg(0);
    Reg#(Bit#(32)) y <- mkReg(0);
    FIFOF#(Bit#(32)) outQ <- mkSizedFIFOF(2);
rule step1 ((x > y) && (y != 0));
    x <= y; y <= x;
endrule
rule step2 (( x <= y) && (y != 0));
    y <= y-x;
    if ( y-x == 0 ) begin
        outQ.enq(x);
    end
endrule
method Action start(Bit#(32) a, Bit#(32) b) if (y==0);
    x <= a; y <= b;
endmethod
method ActionValue#(Bit#(32)) result();
    outQ.deq;
    return outQ.first;
endmethod
endmodule

Interface methods are also atomic transactions
Can be called only when guard is satisfied
When guard is not satisfied, rules that call it cannot fire
Bluespec Modules – Interface

- Modules encapsulates state and behavior (think C++/Java classes)
- Can be interacted from the outside using its “interface”
  - Interface definition is separate from module definition
  - Many module definitions can share the same interface: Interchangeable implementations
- Interfaces can be parameterized
  - Like C++ templates “FIFO#(Bit#(32))”
  - Not important right now

```
module mkGCD (GDCIfc);
  ...
  method Action start(Bit#(32) a, Bit#(32) b) if (y==0);
    x <= a; y <= b;
  endmethod
  method ActionValue#(Bit#(32)) result();
    outQ.deq;
    return outQ.first;
  endmethod
endmodule
```

```
interface GDCIfc;
  method Action start(Bit#(32) a, Bit#(32) b);
  method ActionValue#(Bit#(32)) result();
endinterface
```
Bluespec Module – Interface Methods

- Three types of methods
  - Action: Takes input, modifies state
  - Value: Returns value, does not modify state
  - ActionValue: Returns value, modifies state

- Methods can have “guards”
  - Does not allow execution unless guard is True

```plaintext
rule ruleA;
  moduleA.actionMethod(a,b);
  Int #(32) ret = moduleA.valueMethod(c,d,e);
  Int #(32) ret2 <- moduleB.actionValueMethod(f,g);
endrule

Guard

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

method ActionValue#(Bit#(32)) result();
  outQ.deq;
  return outQ.first;
endmethod

Note the “<-” notation

Automatically introduces “implicit guard” if outQ is empty
Combinational circuits in Bluespec: Rules

- A Bluespec rule represents a state transfer via combinational circuits
  - Much like Verilog “always” and VHDL “process”
  - Can call methods of other modules
    - e.g., `outQ.enq` – Introduces implicit guard if `outQ` is full

```plaintext
rule step2 ((x <= y) && (y != 0));
    y <= y - x;
    if ( y-x == 0 ) begin
        outQ.enq(x);
    end
endrule
```

“enq” encapsulates more combinational logic
Combinational circuits in Bluespec: Functions

- Functions are combinational – do not allow state changes
  - Can be defined within or outside module scope
  - No state change allowed, only performs computation and returns value

```plaintext
// Function example
function Int#(32) square(Int#(32) val);
    return val * val;
endfunction

rule rule1;
    x <= square(12);
endrule
```

Combinational ALU implemented using a function
Bluespec Rules Are Atomic Transactions

- Only has access to state values from before rule began firing
- State update happens once as the result of rule firing
  - e.g.,
    
    // x == 0, y == 1
    x <= y; y <= x; // x == 1, y == 0

  - e.g.,
    
    // x == 0, y == 1
    x <= 1; x <= y; // write conflict error!

```
rule step2 ((x <= y) && (y != 0));
    y <= y-x;
    if ( y-x == 0 ) begin
        outQ.enq(x);
    end
endrule
```

Intuition: All statements in rule execute in parallel

Fires if:
1.  x <= y && y != 0 && y-x == 0 && outQ.notFull
or
2.  x <= y && y != 0 && y-x != 0
Bluespec State – FIFO

- Fixed size queue
- Parameterized interface with guarded methods
  - e.g., testQ.enq(data); // Action method. Blocks when full
  - testQ.deq; // Action method. Blocks when empty
  - dataType d = testQ.first; // Value method. Blocks when empty
- FIFOF adds two more methods
  - testQ.notEmpty returns bool
  - testQ.notFull returns bool
- Provided as library
  - Needs “import FIFO::*;” at top

```verilog
FIFO#(Bit#(32)) testQ <- mkSizedFIFO(2);
rule enqdata; // whole rule does not fire if testQ is full
  if ( x ) y <= z;
  testQ.enq(32'h0);
endrule
```
Bluespec rules:
State and temporary variables

- State: Defined outside rules, data stored across clock cycles
  - All state updates happen atomically
  - Reg#(...), FIFO#(...)
  - Register state assignment uses “<=“

- Temporary variables: Defined within rules, data local to a rule execution
  - Intuition: Rule-local variables
  - Follows sequential semantics similar to software languages
  - Temporary variable value assignment uses “=“

- Same syntax as Verilog/VHDL
Temporary variables behave as you would expect

```verilog
Reg #(Bit #32) a <- mkReg(1); // State
Reg #(Bit #32) b <- mkReg(4); // State
rule rule_a;
  Bit #32 c = a + 1; // Temporary variable c == 2
  Bit #32 d = (c + b)/2; // Temporary variable d == 3
  a <= d; // State a == 3 after this cycle
  b <= a + d; // State b == 4 after this cycle
endrule
```
Behavior of Bluespec Rules

- At every cycle, all rules that can fire, will fire
  - All guards are satisfied
  - No conflicts between rules

- Conflict between rules?
  - Two rules updating same state (writing to same register, enq’ing to same FIFO)
    - One rule enq’ing, one rule deq’ing is OK!
  - When conflict, only one rule fires
    - Typically the first one in the source file