Lecture 05: Rules

- Rules: conditions, actions
- Rule Untimed Semantics
  - Non-determinism
- Functional correctness: atomicity, invariants
- Examples
- Performance correctness (not yet!)
- Rule transformations: splitting, merging, adding, deleting
- Brief intro to implicit conditions on methods
Rules

- All behavior in BSV is expressed using Rules
- A rule is a Declarative specification of a state transition
- A rule is an Action guarded by a Boolean condition

Syntax:

```
rule ruleName [( condition )];
    actions
endrule [: ruleName]
```

Over-simplified analogy to a Verilog “always” block:

```
always @(posedge CLK)
    if cond begin
        actions
    end
```

Rules “Fire”

- Rules don’t control clocks, they only generate enable logic and muxing – you define all modules and state
- No clock or reset ??
  - those are wired directly to state elements…
- Firing means it is enabled and selected for this cycle…
- If registers are loaded, then they load at the end of the cycle
- If wires are driven (more later) they drive out of this rule
- Verilog:
  - “always” clocking, but then if-else-if-else-if-else until you finally load
- BSV
  - rules always try to fire unless you or another rule tells it not to
- There are 3 basic considerations for rules to fire (but a more complete description can be found on our wiki under “Rules of Rules”)

Not True in General

{\textcopyright Bluespec Inc. 2005-2008}
Consideration #1
Rule Conditions

• Rule (or explicit) conditions are arbitrary expressions of type Bool
  • Such expressions are purely combinational
  • This is guaranteed by BSV’s type-checking rules
  • If this condition is false, rule doesn’t fire
  • No condition means default True...

Consideration #2
Ready Conditions

• Methods have “ready” signal
  • Classic examples
    • fifo.enq() – ready when !full
    • fifo.first() – ready when !empty
  • Ready can be always True (always_ready)
  • Ready signals are specified for each method in defining module (more on this in interfaces)
• Rule doesn’t fire unless all ready conditions are true
Consideration #3
Conflicting Rules

- A rule may not fire because it conflicts with other rules
  - Compiler will warn you
  - You should look at warnings
- But we need to take a closer look at what a “conflict” is

```
rule rule1;
  x <= x + 1;
endrule

rule rule2;
  x <= x + 2;
endrule
```

When does a rule ever fire?

- Every cycle!
  - Unless you tell it not to (rule condition)
  - Unless a child module tells it not to (ready condition)
  - Unless a “more important” rule needs to fire instead (conflicts)

- The same restrictions you would have to manually code in rtl, are handled more consistently and correctly by bsv.

- Let’s look at some more specifics
Rule Actions

- The simplest rule action is a register assignment:
  - \( r \leftarrow \ldots \) expression \( \ldots \)
  - BSV uses “\( \leftarrow \)” non-blocking assignment, which is a write action to a register

- A rule body can contain multiple actions

- There is no “sequencing” of actions in a rule body: all the actions happen simultaneously, in parallel
  - The following two rules are equivalent

  ```plaintext
  rule rule1 (state == TRANSFER);
  valuea \leftarrow valuea + 1;
  mult \leftarrow valuea * 2;
  endrule : rule1
  ```

  ```plaintext
  rule rule2 (state == TRANSFER);
  mult \leftarrow valuea * 2;
  valuea \leftarrow valuea + 1;
  endrule : rule2
  ```

Parallel Composable Rule Actions

- Because there is no sequencing of actions in a rule body (all the actions happen simultaneously, in parallel), it is meaningless to put conflicting actions in the same rule, as in the examples below

  - The compiler will flag such errors
  - (We’ll be more precise about “conflicting” later. We use the term parallel composable for actions that can be in the same rule body)

  ```plaintext
  rule rule3 (...);
  valuea \leftarrow valuea + 1;
  valuea \leftarrow valuea + 2;
  ...
  endrule : rule3
  ```

  ```plaintext
  rule rule4 (...);
  fifo.enq (23);
  fifo.enq (34);
  ...
  endrule : rule4
  ```
Resource conflicts in Rule Actions

- A rule also cannot contain resource conflicts, even if they are combinational, as in the example below
  
  - (Trying simultaneously to read two different registers in a register file using a single read-port)
  - The compiler will flag such errors
  - (We'll be more precise about "conflicting" later)

```verilog
rule rule5 (...);
    let x = regFile.sel1(5);
    let y = regFile.sel1(7);
    ...
endrule : rule5
```

Rule Untimed Semantics

Given a set of rules and an initial state

while ( some rules are applicable in the current state )*
  
  - choose one applicable rule**
  - apply the rule to the current state to produce the next state of the system

* Notes:
  
  * "Applicable" means: a rule's condition is true in current state
  * ** Yes, the choice of rule to execute is non-deterministic!
    The timed semantics (next lecture) removes this non-determinism
Rule Untimed Semantics

- We call execution of a rule a step

- Steps are not the same as clock cycles!
  - We’ll discuss clocks and scheduling of rule steps into clocks, later
  - In particular, rule steps don’t represent real time

- Rule Untimed Semantics are adequate for reasoning about functional correctness

Reasoning about functional correctness

- Easier to reason about correctness when considered one rule at a time
  - Does the rule condition accurately capture the desired condition for the state transition?
  - If so, do the rule actions perform the desired state change?

- In particular, no problems with concurrency with other rules:
  - No race conditions, incorrect timing, or inconsistent states

- We also say that rules are atomic,
  or that a rule step is an atomic state change

> Major impact on design entry time and on verification time

Note: the concept of atomicity and atomic transactions is well established in Computer Science (Operating Systems, Database Systems, Distributed Systems) as a powerful construct for achieving correctness in concurrent systems
Example: Simple Binary Multiplication

Consider the simple textbook binary multiplication algorithm illustrated below:

\[
\begin{array}{c}
1001 & // d = 4'd9 \\
\times & 0101 & // r = 4'd5 \\
\hline
1001 & // d << 0 (since r[0] == 1) \\
0000 & // 0 << 1 (since r[1] == 0) \\
1001 & // d << 2 (since r[2] == 1) \\
0000 & // 0 << 3 (since r[3] == 0) \\
0101101 & // product (sum of above) = 45 \\
\end{array}
\]

Example: Simple Binary Multiplication

```verilog
module mkMult (Empty);
  int d_input = 9, r_input = 5;
  Reg#(int) r <- mkReg (r_input);
  Reg#(int) d <- mkReg (d_input);
  Reg#(int) product <- mkReg (0);

  rule cycle_add (r[0]==1); // * algorithm
  product <= product + d;
  d <= d << 1;
  r <= r >> 1;
  endrule: cycle_add

  rule cycle_shift (r!=0 && r[0]==0);
  d <= d << 1;
  r <= r >> 1;
  endrule: cycle_shift

  rule done (r == 0); // output msg when done
  $display("Product = %d", product);
  endrule: done

endmodule: mkMult
```

Example: Simple Binary Multiplication

```verilog
module mkMult (Empty);
  int d_input = 9, r_input = 5;
  Reg#(int) r <- mkReg (r_input);
  Reg#(int) d <- mkReg (d_input);
  Reg#(int) product <- mkReg (0);

  rule cycle_add (r[0]==1); // * algorithm
  product <= product + d;
  d <= d << 1;
  r <= r >> 1;
  endrule: cycle_add

  rule cycle_shift (r!=0 && r[0]==0);
  d <= d << 1;
  r <= r >> 1;
  endrule: cycle_shift

  rule done (r == 0); // output msg when done
  $display("Product = %d", product);
  endrule: done

endmodule: mkMult
```
E.g., Simple 2-stage Pipelined Processor

```
module mkCPU (Empty);
    Reg#(int) pc <- mkReg (0);
    RegFile rf <- mkRegFile;
    IMem   imem <- mkIMem;    // instruction mem
    DMem   dmem <- mkDMem;    // data mem
    SFIFO#(DecodedInstr) buf <- mkSFIFO;

    ... rules for behavior ... // (in following slides)

endmodule: mkCPU
```

Sub modules in Processor

- Assume we’re given the sub-modules
  - mkRegFile (register file)
  - mkIMem (instruction memory)
  - mkDMem (data memory)
  - mkSFIFO (intermediate searchable FIFO buffer)

- In this example we’ll use methods on the interfaces pc, rf, buf, imem and dmem, but we won’t delve into the details
  - The focus here is on the rules for module mkCPU
Instructions and Decoded Instructions

- Assume the following types for instructions and decoded instructions
  - Don’t worry about efficient encodings for now; the focus is on the rules, for the moment

```plaintext
typedef enum { Add, Bz, Ld, St } Opcode deriving (Bits, Eq);

typedef Bit#(4) RegName;

typedef struct {
  Opcode op;
  RegName dest;
  RegName src1;
  RegName src2;
} Instr deriving(Bits);

typedef struct {
  Opcode op;
  RegName dest;
  int v1;
  int v2;
} DecodedInstr deriving(Bits);
```

Sub modules in Processor

- Assume the following methods on the buf (SFIFO) submodule

```plaintext
buf.notStall (instr)  Bool method: True if there is room in the FIFO, and there is no decoded instr in the FIFO whose destination register is the same as either instr.src1 or instr.src2
buf.enq (di)          Action method; enqueues decoded instruction di into the FIFO
buf.notEmpty()        Bool method: True if there is a decoded instruction in the FIFO
buf.first()           Returns the decoded instruction at the head of the FIFO
buf.deq()             Action method: dequeues (discards) the decoded instruction at the head of the FIFO
buf.clear()           Discards entire contents of the FIFO
```
Sub modules in Processor

- Assume the following methods on the rf (RegFile) sub-module, imem (IMem) and dmem (DMem)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rf.sel1(r)</td>
<td>One of two “read ports” on the register file</td>
</tr>
<tr>
<td>rf.sel2(r)</td>
<td>Returns the int value in register r</td>
</tr>
<tr>
<td>rf.upd(r,v)</td>
<td>Second of two “read ports” on the register file</td>
</tr>
<tr>
<td>imem.load(addr)</td>
<td>Returns the contents of location addr in the instruction memory (assume combinational)</td>
</tr>
<tr>
<td>dmem.load(addr)</td>
<td>Returns the contents of location addr in the data memory (assume combinational)</td>
</tr>
<tr>
<td>dmem.upd(addr,v)</td>
<td>Action method; updates data mem location addr with value v</td>
</tr>
</tbody>
</table>

The Processor’s Rules

```
Instr instr = imem.load(pc); // the current instruction

rule fetch (buf.notStall (instr));
  let di = DecodedInstr { op: instr.op;
                          dest: instr.dest;
                          v1:  rf.sel1 (instr.src1);
                          v2:  rf.sel2 (instr.src2); };
  buf.enq (di);
  pc <= pc + 1;
endrule: fetch
```

- The rule will execute only if buf.notStall(instr) is True
- “let di =” builds a decoded instruction value from fields in the current instruction and values read from the register file
- If the rule executes, it has two simultaneous actions: enqueue di, and increment pc
The Processor’s Rules

```plaintext
rule execute (buf.notEmpty());
  let di = buf.first();
  case (di.op)
    Add: begin rf.upd (di.dest, di.v1 + di.v2);
      buf.deq(); end
    Ld:  begin rf.upd (di.dest, dmem.load (di.v1));
      buf.deq(); end
    St:  begin dmem.store (di.v1, di.v2);
      buf.deq(); end
    Bz:   if (di.v1 == 0) begin pc <= di.v2;
    St:    begin dmem.load (di.v1);
    Ld:     begin rf.upd (di.dest, dmem.load (di.v1));
      buf.deq(); end
    Add: begin rf.upd (di.dest, di.v1 + di.v2);
      buf.deq(); end
endcase
endrule: execute
```

- Action set depends on opcode:
  - If Bz taken, update pc and clear buf, else deq decoded instr from buf
  - Add and Ld, update the register file rf
  - St stores into the data memory dmem

2 stage Processor Pipeline

- What if both the fetch and the execute rules are enabled
  - We can choose either rule—*it does not matter for functional correctness!*
  - If repeatedly choose the fetch rule, it’ll simply stall, at some point, because of the FIFO (either full, or decoded instr in the FIFO with conflicting register)
    - And then the exec rule will execute
  - If we repeatedly choose the execute rule, it’ll simply stall at some point, because of the FIFO (becomes empty)
    - And then the fetch rule will execute

- It is precisely this kind of equivalence that allows the Bluespec compiler to *schedule* rules to execute concurrently in the same clock (next lecture!) while preserving the functional correctness we established while reasoning one-rule-at-a-time
Atomicity and correctness

- When the exec rule executes a Bz instruction and takes the branch, it updates the pc and clears buf

- But isn’t the fetch rule also trying to update the pc and enqueue something into buf?

- Any chance we might accidentally corrupt the pc, or leave a wrong instruction in the buf?

- Not with one-rule-at-a-time semantics!
  - Either the fetch rule executes before the exec rule, or after. Either way, it is easy to reason that we get correct behavior.

Performance

- Of course, if we produced HW that really executed only one rule per clock, it would not really be pipelined! It would not meet performance requirements.

- The one-rule-at-a-time semantics is the logical behavior, the reference model for establishing functional correctness

- The actual behavior in the hardware will execute multiple rule steps concurrently (in the same clock), details in the next lecture

- But the Bluespec compiler ensures that the actual behavior is always consistent with the logical behavior, thereby preserving functional correctness while achieving performance goals
Rule Transformations (future topic)

- There is extensive Computer Science literature on the impact, in a rule-based system, of
  - Splitting rules
  - Joining (merging) rules
  - Adding rules
  - Deleting rules
  i.e., how does the behavior of the system change?

- This makes Rules an excellent basis for Formal Verification in the future (we will not explore this during this training session)

Implicit Conditions in Methods (later)

- In the processor example, the rules connected to some methods that were, in some sense, legality conditions for other methods. Examples:
  - buf.enq() is only legal if the buffer has available space, which is tested by buf.notStall()
  - buf.first() and buf.deq() are only legal if the buffer has an item available, which is tested by buf.notEmpty()

- What if we had made a mistake, e.g., examining buf.first() when buf.notEmpty() is False?

- In a later lecture we’ll see how buf.notEmpty() can be folded into an implicit condition on buf.first() and buf.deq() so that
  - It does not have to be tested explicitly (compiler will insert it)
  - It becomes therefore impossible to make this mistake
End of Lecture