Bluespec SystemVerilog™ Training

Lecture 08: Performance Tuning using Rule Composition

- Review: Composing rules into clock cycles
- Why more aggressive rule composition may be necessary
  - Example: Interlocked pipeline register (1-element FIFO)
- Wires and how they enable aggressive rule composition
  - Implementing an interlocked pipeline register
- Alternative FIFO scheduling: bypassing enq to deq on “empty”
- More examples
- Future: automatic composition from performance specs
- Rule execution assertions, statically checked
Review: practical rule composition

- In principle, any pair of enabled rules can be composed to execute within a clock cycle, by feeding the output of the logic of the body of rule 1 into the condition/body of rule 2.

- However, for pragmatic reasons (discussed in Lecture 06) the compiler only performs restricted compositions, illustrated above, where:
  - there is no intra-cycle communication between rules

More aggressive rule composition is necessary in some situations

- Consider interlocked pipeline registers, e.g., in a CPU pipeline

- “Interlocked” ➔ they are 1-element FIFOs. E.g.,
  - execute stage stalls until an item is available in pipeline register \( r \) from decode stage (“deq”)
  - decode stage cannot write into \( r \) if execute stage has not consumed previous item (“enq”)

- For pipelining, decode and execute stages must be able to concurrently (in same cycle) enq and deq, respectively, into \( r \)
Naïve implementation of pipeline register

```verilog
module mkNaiveFIFO (FIFO#(int));
  Reg#(int) data <- mkRegU;
  Reg#(Bool) full <- mkReg(False);
  method Action enq (int x) if (! full);
    data <= x;
    full <= True;
  endmethod
  method int first if (full);
    return data;
  endmethod
  method Action deq if (full);
    full <= False;
  endmethod
endmodule
```

Problem: In any cycle, the FIFO is either full or empty
- can either use enq, or first/deq, but not both simultaneously

Diagnosing the problem in terms of rule composition

- Let’s look at the rules that use the methods
  - The decode rule incorporates the r.enq() method
  - The execute rule incorporates the r.first()/r.deq() methods

```verilog
module mkCPU (...)
  FIFO#(int) r <- mkNaiveFIFO;
  ...
  rule decode (...);
    ... r.enq (x) ... endrule
  ...
  rule execute (...);
    ... x = r.first (); r.deq(); ... endrule
  ...
endmodule: mkCPU
```

... enq() ... rule decode first() deq() ...
Diagnosing the problem in terms of rule composition

- Suppose \( r \) contains an item (full \( = \) True)
  - The decode rule cannot go (can’t enq)
  - The execute rule can go (can first/deq); this empties \( r \)
  - This state change allows the decode rule to go (can enq)
- Thus, we get schedule \( A \) (below)
- What we want is schedule \( B \), i.e., the same net behavior, except that the state change of deq is visible to enq within the same clock cycle

RWires permit rule composition by enabling intra-clock communication

```
interface RWire #(type element_type);   // Declared in library
    method Action wset (element_type datain);
    method Maybe#(element_type) wget ();
endinterface

module mkRWire (RWire#(element_type));   // Declared in library
    RWire#(Bit#(16)) myWire <- mkRWire;
    // Instantiation of Bit#(16) RWire
```

A \hspace{1cm} B
\begin{align*}
rule steps & \quad \text{execute} \quad \text{decode} & \quad \text{rule steps} & \quad \text{execute} \quad \text{decode} \\
\text{clocks} & \quad \text{execute} \quad \text{decode} & \quad \text{execute} \quad \text{decode} \\
\end{align*}
RWire hardware

- The mkRWire module contains no state and no logic: it’s just wires!
- By testing the valid bit of wget() we know whether some rule containing wset() is executing concurrently (enab is True)
- RWire is available in the Bluespec library
- Recall: a value mx of type Maybe#(t) consists of a value x together with a valid bit

```plaintext
Maybe#(int) m1 = tagged Valid 23;    // valid bit True, value 23
Maybe#(int) m2 = tagged Invalid;     // valid bit False, value unspecified
m2 = m1;                              // This sets m2 Valid and 23
// Some functions
Bool b   = isValid (m2);              // b == valid bit of m2
int d    = fromMaybe (34, m2);        // d = value of m2 if valid, else 34
```

RWires permit rule composition by enabling intra-clock communication

- Suppose a rule Rj uses rw.wset() on an RWire
- Suppose a rule Rk uses rw.wget() on the same RWire
- If rule Rj and Rk execute concurrently (same cycle) then Rj always precedes Rk in the rule-step semantics
- Testing isValid(rw.wget()) allows Rk to test whether Rj is concurrent (executing in the same cycle)
- wset/wget allows Rj to communicate a value to Rk
Implementing the pipeline register

```verilog
declares
module mkPipelineFIFO (FIFO#(int));
Reg#(int) data <- mkRegU;
Reg#(Bool) full <- mkReg(False);
RWire#(int) inEnq <- mkRWire;  // info from enq method
RWire#(Bool) inDeq <- mkRWire;  // info from deq method

rule fifoUpdate (isVal(inEnq.wget)) || isVal(inDeq.wget)); // if inEnq or inDeq
    full <= isVal(inEnq.wget);  // valid bit
    data <= fromMaybe(0, inEnq.wget); // value if inEnq, otherwise keep 0
endrule

method Action enq (x) if (full || isVal(inDeq.wget)); // if empty or inDeq, even full!
inEnq.wset (x);  // set inEnq to Valid and transporting x
endmethod

method Action deq if (full);
inDeq.wset (True);  // set inDeq to Valid and transporting True
endmethod

method int first if (full);
return data;
endmethod
endmodule
```

Note: PulseWire can replace inDeq

```verilog
interface PulseWire;
    method Action send;
    method Bool _read;
endinterface

module mkPulseWire (PulseWire);
```

A HW implication of mkPipelineFIFO

- There is now a combinational path from enab_deq to rdy_enq (a consequence of the RWire)

- This is how a rule using enq() “knows” that it can go even if the FIFO is full, i.e., enab_deq is a signal that a rule using deq() is executing concurrently
Rule composition with the pipeline register

An alternate scheduling of FIFO methods

- The following table summarizes the allowed concurrency in the two versions of FIFO

<table>
<thead>
<tr>
<th></th>
<th>Empty</th>
<th>Full</th>
</tr>
</thead>
<tbody>
<tr>
<td>NaiveFIFO</td>
<td>enq</td>
<td>deq</td>
</tr>
<tr>
<td>PipelineFIFO</td>
<td>enq</td>
<td>deq &lt; enq</td>
</tr>
</tbody>
</table>

- What about the following?

<table>
<thead>
<tr>
<th></th>
<th>enq &lt; deq</th>
<th>deq</th>
</tr>
</thead>
<tbody>
<tr>
<td>BypassFIFO</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The BypassFIFO

- When empty, can concurrently enq and deq, with logical behavior as if:
  - enq, thereby placing an item $x$ in the fifo
  - then deq, thereby yielding $x$
- Since these can happen in the same cycle, $x$ “flies through” the fifo, i.e., it is bypassed through the fifo

<table>
<thead>
<tr>
<th></th>
<th>Empty</th>
<th>Full</th>
</tr>
</thead>
<tbody>
<tr>
<td>BypassFIFO</td>
<td>enq &lt; deq</td>
<td>deq</td>
</tr>
</tbody>
</table>

HW implications of mkBypassFIFO

- There is now a combinational path from enab_enq to rdy_deq and rdy_first
- This is how a rule using deq() “knows” that it can go even if the FIFO is empty, i.e., enab_enq is a signal that a rule using enq() is executing concurrently
- There is also a combinational path from datain to dataout (the “bypass” path)
Which FIFO to use?

- The choice of FIFO depends on performance requirements
- E.g., consider the interconnect below
  - If PipelineFIFOs are used, packets are registered on inputs and outputs, and there is a 2-tick minimum latency through the switch
  - If BypassFIFOs are used on the input side, packets are registered only on outputs, and there is a 1-tick minimum latency through the switch

- In BSV, the code for the switch rules will not change at all
  - You simply replace mkPipelineFIFO by mkBypassFIFO
  - All the required changes in control logic are automatically regenerated by recompilation

The benefits of Rules and Scheduling

- These variations in scheduling are not an artifact of using BSV, Rules and Interface Methods—they are fundamental, reflecting fundamentally different properties of hardware!
  - The choices represent different behaviors
  - The choices need different control logic

- Rules and rule scheduling provide us with a precise and unambiguous way to specify and analyze these issues, and to automatically generate the correct control logic
  - You’d be designing the same control logic manually in RTL, and without the benefit of a precise model in mind
Another useful wires available

<table>
<thead>
<tr>
<th>Interface</th>
<th>Module</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wire(Reg)</td>
<td>mkWire</td>
<td>RWire without Valid/Invalid tag</td>
</tr>
<tr>
<td>PulseWire</td>
<td>mkPulseWire</td>
<td>RWire without any data</td>
</tr>
<tr>
<td>Wire</td>
<td>mkDWire (defaultValue)</td>
<td>Always ready Wire</td>
</tr>
<tr>
<td>Wire</td>
<td>mkBypassWire</td>
<td>Always enabled Wire</td>
</tr>
</tbody>
</table>

More examples in next lecture!

Performance tuning methodology

- Write your design, compile and execute
- If it has inadequate latency/bandwidth, analyze the design to explain it in terms of rule composition:
  - “These rules do not go together; if they did, the goal would be met”
  - Analyze why they do not go together:
    - Cogitate (think!)
    - Examine scheduling outputs from the compiler, which tell you which rules cannot go together, and why
    - Examine simulation waveforms (CAN_FIRE, WILL_FIRE signals)
  - Understand what rule composition is needed
- See if an off-the-shelf component such as PipelineFIFO or BypassFIFO will solve the problem
- Else, use wires to implement any desired rule composition
Implementing the pipeline register (future automation)

- RWires allow you to compose rules *manually*

- In the future, it will be possible to
  - Write the naive FIFO code (mkNaiveFIFO)
  - Declare a simple "performance specification":
    - Just say: "deq < enq"
    - Meaning: "let a rule r1 containing deq go concurrently with (in the same cycle as) a rule r2 containing enq, with behavior logically equivalent to doing r1 before r2 in the rule-step semantics"
  - And have the compiler *automatically* transform it into rule-composed implementation

- We know how to do it (MIT patent); it just hasn’t been implemented in the compiler yet
  - Bluespec has exclusive rights to the MIT patent

Recall CAN_FIRE and WILL_FIRE

- Scheduler incorporates conflict analysis by compiler
  - CAN_FIRE is False  \(\Rightarrow\) WILL_FIRE is False
  - CAN_FIRE is True  \(\Rightarrow\) WILL_FIRE is
    - True if not precluded by conflicts with other rules
    - False otherwise (the rule is blocked for this cycle)
Assertions about rule execution

(* fire_when_enabled, no_implicit_conditions *)
rule ... (cond);
...
endrule

- These use SV attribute syntax, written just above a rule
- They represent assertions that do not change the generated RTL, and are checked statically by the compiler

  - **fire_when_enabled**: Asserts that the rule will always execute when its conditions are True, i.e., it cannot be stalled by more urgent or conflicting rules
  - **no_implicit_conditions**: Asserts that rule actions do not introduce any implicit conditions, i.e., the condition for firing the rule is just \( \text{cond} \), and nothing more
    - Can be combined with **fire_when_enabled** to guarantee that the rule will fire when \( \text{cond} \) is True

Assertions about rule execution

Suppose that when condition (\( \text{cmd} == \text{Swap} \)) is True, the contents of registers \( x \) and \( y \) MUST swap.

(* descending_urgency = "swapEm, addOne" *)
(* fire_when_enabled *)
rule swapEm (cmd == Swap);
  x <= y;
  y <= x;
endrule
rule addOne(incX);
  x <= x + 1;
endrule

... It does not really matter how many rules are added later, the assertion:
if (swapEm_CAN_FIRE == TRUE) then (swapEm_WILL_FIRE == TRUE)
will be always checked during compilation

The compiler will signal an error because rule addOne is more urgent and the assertion fails!
The problem can be solved adding descending urgency attribute
Assertions about rule execution

Suppose that data coming from a sub-hierarchical module MUST be obtained each clock cycle.

```plaintext
... // instantiation of myIP module
Internal_IP myIP <- mkIP;

(* no_implicit_conditions *)
rule swapEm;
  x <= myIP.read;
endrule
...
```

It does not really matter if the internal module is modified by someone else, the compiler will check during compilation time that the rule does not have any implicit conditions inherited by the method call.

Summary

- The kinds of timing and scheduling issues discussed here are the bane of the HW designer
  - Imprecision and misunderstanding can lead to race conditions, errors, nasty surprises

- Rule semantics and rule composition provide a precise and unambiguous vocabulary to specify and analyze these issues

- RWires, PipelineFIFOs, BypassFIFOs, and other components provide ways to implement any desired intra-clock rule composition