Constructive Computer Architecture: Multirule Systems and Concurrent Execution of Rules

Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology
Multi-rule Systems

Repeatedly:
- Select a rule to execute
- Compute the state updates
- Make the state updates

Non-deterministic choice; User annotations can be used in rule selection

One-rule-at-a-time-semantics: Any legal behavior of a Bluespec program can be explained by observing the state updates obtained by applying only one rule at a time

However, for performance we execute multiple rules concurrently whenever possible
Elastic pipeline

Can these rules fire concurrently?

Yes, but it must be possible to do enq and deq on a fifo simultaneously.
One-Element FIFO Implementation

```verilog
module mkFifo (Fifo#(1, t));
    Reg#(t) d <- mkRegU;
    Reg#(Bool) v <- mkReg(False);
method Action enq(t x) if (!v);
    v <= True; d <= x;
endmethod
method Action deq if (v);
    v <= False;
endmethod
method t first if (v);
    return d;
endmethod
endmodule
```

Can `enq` and `deq` methods be ready at the same time?

No! Therefore they cannot execute concurrently!
Concurrency when the FIFOs do not permit concurrent enq and deq

At best alternate stages in the pipeline will be able to fire concurrently
Two-Element FIFO

- Initially, both va and vb are false
- First enq will store the data in da and mark va true
- An enq can be done as long as vb is false; a deq can be done as long as va is true

Assume, if there is only one element in the FIFO it resides in da
Two-Element FIFO

BSV code

```vhdl
module mkFifo (Fifo#(2, t));
    Reg#(t) da <- mkRegU();
    Reg#(Bool) va <- mkReg(False);
    Reg#(t) db <- mkRegU();
    Reg#(Bool) vb <- mkReg(False);
    method Action enq(t x) if (!vb);
        if (va) begin db <= x; vb <= True; end
        else begin da <= x; va <= True; end
    endmethod
    method Action deq if (va);
        if (vb) begin da <= db; vb <= False; end
        else begin va <= False; end
    endmethod
    method t first if (va); return da;
endmodule
```

Assume, if there is only one element in the FIFO it resides in da

Can both enq and deq be ready at the same time? yes
### Two-Element FIFO

**concurrency analysis**

```action
method Action enq(t x) if (!vb);
  if (va) begin db <= x; vb <= True; end
  else begin da <= x; va <= True; end
endmethod

method Action deq if (va);
  if (vb) begin da <= db; vb <= False; end
  else begin va <= False; end
endmethod
```

- **Will concurrent execution of** `enq` **and** `deq` **cause a double write error?**
  - Initially `vb=false` and `va=true`
  - `enq` will execute: `db <= x; vb <= True;`
  - `deq` will execute: `va <= false;`

- **The final state will be** `va = false` and `vb = true`;
  - with the old data in `da` and new data in `db`

**we can’t get into this state if enq and deq are performed in some order**

- **no double-write error**

- **oops!**
Two-Element FIFO

concurrency analysis - continued

method Action enq(t x) if (!vb);
  if (va) begin db <= x; vb <= True; end
  else begin da <= x; va <= True; end
endmethod

method Action deq if (va);
  if (vb) begin da <= db; vb <= False; end
  else begin va <= False; end
endmethod

In this implementation, enq and deq should not be called concurrently
  later we will present a systematic procedure to decide which methods of a module can be called concurrently

First, we will study when two rules that only use registers can be executed concurrently
Concurrent execution of rules

Two rules can execute concurrently, if concurrent execution would not cause a double-write error, and

The final state can be obtained by executing rules one-at-a-time in some sequential order
Can these rules execute concurrently? (without violating the one-rule-at-a-time-semantics)

Example 1

\[
\begin{array}{rcl}
\text{rule ra;}
\quad & x \leq x+1; \\
\text{endrule}
\end{array}
\]

\[
\begin{array}{rcl}
\text{rule rb;}
\quad & y \leq y+2; \\
\text{endrule}
\end{array}
\]

Example 2

\[
\begin{array}{rcl}
\text{rule ra;}
\quad & x \leq y+1; \\
\text{endrule}
\end{array}
\]

\[
\begin{array}{rcl}
\text{rule rb;}
\quad & y \leq x+2; \\
\text{endrule}
\end{array}
\]

Example 3

\[
\begin{array}{rcl}
\text{rule ra;}
\quad & x \leq y+1; \\
\text{endrule}
\end{array}
\]

\[
\begin{array}{rcl}
\text{rule rb;}
\quad & y \leq y+2; \\
\text{endrule}
\end{array}
\]

Final value of \((x,y)\) (initial values \((0,0)\))

<table>
<thead>
<tr>
<th></th>
<th>Ex. 1</th>
<th>Ex. 2</th>
<th>Ex. 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>ra &lt; rb</td>
<td>(1,2)</td>
<td>(1,3)</td>
<td>(1,2)</td>
</tr>
<tr>
<td>rb &lt; ra</td>
<td>(1,2)</td>
<td>(3,2)</td>
<td>(3,2)</td>
</tr>
<tr>
<td>Concurrent</td>
<td>(1,2)</td>
<td>(1,2)</td>
<td>(1,2)</td>
</tr>
</tbody>
</table>

No Conflict | Conflict | ra < rb
The BSV compiler schedules as many rules as possible for concurrent execution among the rules that are enabled (i.e., whose guards are true), provided it can ensure that the chosen rules don’t conflict with each other.

Conflicts:
- Double write
- If the effect of rule execution does not appear to be as if one rule executed after the other
some insight into

Concurrent rule execution

There are more intermediate states in the rule semantics (a state after each rule step)

In the HW, states change only at clock edges
Parallel execution reorders reads and writes

In the rule semantics, each rule sees (reads) the effects (writes) of previous rules.

In the HW, rules only see the effects from previous clocks, and only affect subsequent clocks.
Correctness

The compiler will schedule rules concurrently only if the net state change is equivalent to a sequential rule execution.
Compiler test for concurrent rule execution

James Hoe, Ph.D., 2000

Let $RS(r)$ be the set of registers rule $r$ may read
Let $WS(r)$ be the set of registers rule $r$ may write

Rules $ra$ and $rb$ are \textit{conflict free} (CF) if
\[
(RS(ra) \cap WS(rb) = \emptyset) \land (RS(rb) \cap WS(ra) = \emptyset) \land (WS(ra) \cap WS(rb) = \emptyset)
\]

Rules $ra$ and $rb$ are \textit{sequentially composable} (SC) $(ra < rb)$ if
\[
(RS(rb) \cap WS(ra) = \emptyset) \land (WS(ra) \cap WS(rb) = \emptyset)
\]

If Rules $ra$ and $rb$ \textit{conflict} if they are not CF or SC
## Compiler analysis

### Example 1
```
rule ra;
  x <= x+1;
endrule
rule rb;
  y <= y+2;
endrule
```

### Example 2
```
rule ra;
  x <= y+1;
endrule
rule rb;
  y <= x+2;
endrule
```

### Example 3
```
rule ra;
  x <= y+1;
endrule
rule rb;
  y <= y+2;
endrule
```

<table>
<thead>
<tr>
<th></th>
<th>Example 1</th>
<th>Example 2</th>
<th>Example 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS(ra)</td>
<td>{x}</td>
<td>{y}</td>
<td>{y}</td>
</tr>
<tr>
<td>WS(ra)</td>
<td>{x}</td>
<td>{x}</td>
<td>{x}</td>
</tr>
<tr>
<td>RS(rb)</td>
<td>{y}</td>
<td>{x}</td>
<td>{y}</td>
</tr>
<tr>
<td>WS(rb)</td>
<td>{y}</td>
<td>{y}</td>
<td>{y}</td>
</tr>
<tr>
<td>RS(ra) ∩ WS(rb)</td>
<td>ø</td>
<td>{y}</td>
<td>{y}</td>
</tr>
<tr>
<td>RS(rb) ∩ WS(ra)</td>
<td>ø</td>
<td>{x}</td>
<td>ø</td>
</tr>
<tr>
<td>WS(ra) ∩ WS(rb)</td>
<td>ø</td>
<td>ø</td>
<td>ø</td>
</tr>
<tr>
<td>Conflict?</td>
<td>CF</td>
<td>C</td>
<td>SC</td>
</tr>
</tbody>
</table>
Concurrent scheduling

The BSV compiler determines which rules among the rules whose guards are ready can be executed concurrently.

It builds a simple list-based scheduler:

- Picks the first enabled rule in the list
- Schedules the next enabled rule if it does not conflict with any of the rules scheduled so far
- Repeats the process until no more rules can be scheduled

Such a scheduler can be built as a pure combinational circuit but it is not fair.

In practice it does fine and one can get around it programmatically.
Scheduling and Control Logic
Compiling a Rule

rule r (f.first() > 0) ;
  x <= x + 1 ;  f.deq ();
endrule

September 18, 2017
Combining State Updates: strawman

π’s from the rules that update R

δ’s from the rules that update R

What if more than one rule has a true guard?
Combining State Updates

\( \pi \)’s from all the rules

\( \delta \)’s from the rules that update \( R \)

Scheduler ensures that at most one \( \phi_i \) is true

One-rule-at-a-time scheduler is conservative

Scheduler: Priority Encoder

\( \phi_1 \)

\( \phi_n \)

\( \pi_1 \)

\( \pi_n \)

\( \delta_{1,R} \)

\( \delta_{n,R} \)
Compiler synthesizes a scheduler such that at any given time $\phi$'s for only non-conflicting rules are true.
Takeaway

- One-rule-at-a-time semantics are very important to understand what behaviors a system can show.
- Efficient hardware for multi-rule system requires that many rules execute in parallel without violating the one-rule-at-time semantics.
- BSV compiler builds a scheduler circuit to execute as many rules as possible concurrently.