Implementing Pipelining:
Module Interfaces and Concurrency

Arvind
Computer Science and Artificial Intelligence Laboratory
M.I.T.
Inelastic (aka Synchronous) pipeline

```
rule sync_pipeline;
    inQ.deq;
    s1 <= f0(inQ.first);
    s2 <= f1(s1);
    outQ.enq(f2(s2))
endrule
```

Implicit guard?

\[(\text{inQ.notEmpty} \land \text{outQ.notFull})\]
Pipeline bubbles
Starting and stopping the pipeline

Green token must move even if there is nothing in inQ!
Also nothing should be enqueued in outQ, if there is no token in s2

Modify the rule to deal with these conditions

```
rule sync_pipeline;
  inQ.deq;
  s1 <= f0(inQ.first);
  s2 <= f1(s1);
  outQ.enq(f2(s2))
endrule
```
Explicit encoding of Valid/Invalid data

A bool to represent Valid/Invalid

initialize v1 and v2 to False

Quite tedious to write down all the cases and associated actions
Elastic pipeline
Use FIFOs instead of registers to connect stages

When can stage1 rule fire?
- inQ has an element
- fifo1 has space

Can these 3 rules execute concurrently?
- Yes, but it must be possible to do enq and deq in a fifo simultaneously

Can stage1 and stage3 execute concurrently?
- Yes, even if enq and deq cannot be done simultaneously in a fifo
Multirule Systems

- Most systems we have seen so far had multiple rules but only one rule was ready to execute at any given time (pair-wise mutually exclusive rules)
- Consider a system where multiple rules can be ready to execute at a given time
  - When can two such rules be executed together?
  - What does the synthesized hardware look like for concurrent execution of rules?
One-rule-at-a-time semantics of Bluespec

*Repeatedly:* 
- Select any rule that is ready to execute
- Compute the state updates
- Make the state updates

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

However, for performance we execute multiple rules concurrently whenever possible
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

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

Concurrent Execution

- ra < rb
- rb < ra

Final value of (x,y) given the initial values (0,0)

Ex 1

- (1,2) = (1,2)

Ex 2

- (1,2) ≠ (1,3)

Ex 3

- (1,2) ≠ (3,2)

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
```

Final value of (x,y) given the initial values (0,0)

- ra < rb
- rb < ra

Ex 1

- (1,2) = (1,2)

Ex 2

- (1,2) ≠ (3,2)

Ex 3

- (3,2) ≠ (1,2)

Conflict-Free (CF)
Conflict Matrix (CM)
BSV compiler generates the pairwise conflict information

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

<table>
<thead>
<tr>
<th></th>
<th>ra</th>
<th>rb</th>
</tr>
</thead>
<tbody>
<tr>
<td>ra</td>
<td>C</td>
<td>CF</td>
</tr>
<tr>
<td>rb</td>
<td>CF</td>
<td>C</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>ra</th>
<th>rb</th>
</tr>
</thead>
<tbody>
<tr>
<td>ra</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>rb</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>ra</th>
<th>rb</th>
</tr>
</thead>
<tbody>
<tr>
<td>ra</td>
<td>C</td>
<td>&lt;</td>
</tr>
<tr>
<td>rb</td>
<td>&gt;</td>
<td>C</td>
</tr>
</tbody>
</table>

- C: rules can’t be executed concurrently
- CF: rules can be executed concurrently; the net effect is the same as if ra executed before rb (ra<rb) or (rb<ra)
- ra < rb: rules can be executed concurrently; the net effect is as if ra executed before rb
Conflict Matrix for an Interface

- Conflict Matrix (CM) defines which methods of a module can be called concurrently.
- CM for a register:
<table>
<thead>
<tr>
<th></th>
<th>reg.r</th>
<th>reg.w</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg.r</td>
<td>CF</td>
<td>&lt;</td>
</tr>
<tr>
<td>reg.w</td>
<td>&gt;</td>
<td>C</td>
</tr>
</tbody>
</table>

- Two reads can be performed concurrently.
- Two concurrent writes conflict and are not permitted.
- A read and a write can be performed concurrently and it behaves as if the read happened before the write.
- CM of a register is used systematically to derive the CM for the interface of a module and the CM for rules.

A few examples...
### One-Element FIFO

```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
```

<table>
<thead>
<tr>
<th></th>
<th>enq</th>
<th>deq</th>
<th>first</th>
</tr>
</thead>
<tbody>
<tr>
<td>enq</td>
<td>C</td>
<td>ME</td>
<td>ME</td>
</tr>
<tr>
<td>deq</td>
<td>ME</td>
<td>C</td>
<td>&gt;</td>
</tr>
<tr>
<td>first</td>
<td>ME</td>
<td>&lt;</td>
<td>CF</td>
</tr>
</tbody>
</table>

ME = mutually exclusive
How about a 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

```verilog
module mkCFFifo (Fifo#(2, t));
// instantiate da, va, db, vb
    rule canonicalize if (vb && !va);
        da <= db;
        va <= True;
        vb <= False;
    endrule
    method Action enq(t x) if (!vb);
        begin db <= x; vb <= True; end
    endmethod
    method Action deq if (va);
        va <= False;
    endmethod
    method t first if (va);
        return da;
    endmethod
endmodule
```

Both enq and deq can execute concurrently but both are mutually exclusive with canonicalize

<table>
<thead>
<tr>
<th></th>
<th>enq</th>
<th>deq</th>
<th>first</th>
<th>cano</th>
</tr>
</thead>
<tbody>
<tr>
<td>enq</td>
<td>C</td>
<td>CF</td>
<td>CF</td>
<td>ME</td>
</tr>
<tr>
<td>deq</td>
<td>CF</td>
<td>C</td>
<td>&gt;</td>
<td>ME</td>
</tr>
<tr>
<td>first</td>
<td>CF</td>
<td>&lt;</td>
<td>CF</td>
<td>ME</td>
</tr>
<tr>
<td>cano</td>
<td>ME</td>
<td>ME</td>
<td>ME</td>
<td>C</td>
</tr>
</tbody>
</table>
Many other FIFO designs are possible

*Pipelined FIFO:* one can *enq* into a full FIFO if a *deq* is done simultaneously

<table>
<thead>
<tr>
<th></th>
<th>enq</th>
<th>deq</th>
<th>first</th>
</tr>
</thead>
<tbody>
<tr>
<td>enq</td>
<td>C</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>deq</td>
<td>&lt;</td>
<td>C</td>
<td>&gt;</td>
</tr>
<tr>
<td>first</td>
<td>&lt;</td>
<td>&lt;</td>
<td>CF</td>
</tr>
</tbody>
</table>

*Bypass FIFO:* one can *deq* from an empty FIFO if a *enq* is done simultaneously

<table>
<thead>
<tr>
<th></th>
<th>enq</th>
<th>deq</th>
<th>first</th>
</tr>
</thead>
<tbody>
<tr>
<td>enq</td>
<td>C</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>deq</td>
<td>&gt;</td>
<td>C</td>
<td>&gt;</td>
</tr>
<tr>
<td>first</td>
<td>&gt;</td>
<td>&lt;</td>
<td>CF</td>
</tr>
</tbody>
</table>

Design of such FIFOs requires the use of EHRs, registers with bypasses. We will discuss EHRs in L23
Using *conflict* (CM) information in hardware synthesis
Concurrent rule execution

- This circuit will execute rules \( ra \) and \( rb \) concurrently.
- This circuit is correct only if rules \( ra \) and \( rb \) do not conflict (\( \Rightarrow \) methods \( f \) and \( g \) of \( m \) do not conflict).
- Suppose rules \( ra \) and \( rb \) do conflict!
Need for a scheduler

- Guards of all rules are fed to a scheduler
- Using the CM, the scheduler lets only non-conflicting rules proceed
- Scheduler is a pure combinational circuit with a small number of gates
- A correct but low performance scheduler may schedule only one rule at a time

\[
\text{rule ra} \ (p(x)); \\
\quad m. f(x+1); \\
\text{endrule} \\
\text{rule rb} \ (q(x)); \\
\quad m. g(x+2) \\
\text{endrule}
\]
Multiple rules may invoke the same method, so we need to put a mux in front of the interface.

Again, if the scheduler is implemented correctly, it is guaranteed that only one of the inputs to the mux will be true (one-hot encoding).
The rule scheduler

- Guards (gs1 ... gsn) of many rules may be true simultaneously, and some of them may conflict.
- BSV compiler constructs a combinational scheduler circuit with the following property:

  for all $i$ and $j$, if $wfs_i$ and $wfs_j$ are true then the corresponding $gs_i$ and $gs_j$ must be true and rules $i$ and $j$ must not conflict with each other.
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.
- For high-performance designs we have to worry about the CM characteristics of our modules.
Take-home problem

- Draw the hardware circuit for this design

```verilog
rule stage1;
    fifo.enq(f0(inQ.first));
    inQ.deq;
endrule
rule stage2;
    outQ.enq(f1(fifo.first));
    fifo.deq;
endrule
```