Implementing Pipelining: Module Interfaces and Concurrency

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

```
inQ  f0  f1  f2  outQ
s1    s2
```
Inelastic (aka Synchronous) pipeline

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

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

Implicit guard?
Inelastic (aka Synchronous) pipeline

\[
\begin{align*}
\text{rule} & \quad \text{sync\_pipeline}; \\
& \quad \text{inQ}\_\text{deq}; \\
& \quad s1 \leq f0(\text{inQ}\_\text{first}); \\
& \quad s2 \leq f1(\text{s1}); \\
& \quad \text{outQ}\_\text{enq}(f2(s2)) \\
\text{endrule}
\end{align*}
\]

 Implicit guard?

\((\text{inQ}\_\text{notEmpty} \&\& \text{outQ}\_\text{notFull})\)
Pipeline bubbles
Starting and stopping the pipeline

```
rule sync_pipeline;
    inQ.deq;
    s1 <= f0(inQ.first);
    s2 <= f1(s1);
    outQ.enq(f2(s2))
endrule
```
Pipeline bubbles
Starting and stopping the pipeline

Green token must move even if there is nothing in inQ!

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

rule sync_pipeline;
    inQ.deq;
    s1 <= f0(inQ.first);
    s2 <= f1(s1);
    outQ.enq(f2(s2))
endrule
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
```
Pipeline bubbles
Starting and stopping the pipeline

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

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

Valid bits or the Maybe type
Explicit encoding of Valid/Invalid data

A bool to represent Valid/Invalid

initialize v1 and v2 to False
Explicit encoding of Valid/Invalid data

A bool to represent Valid/Invalid

initialize v1 and v2 to False

inQ v1 v2 outQ

<table>
<thead>
<tr>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>NE T T NF</td>
</tr>
<tr>
<td>NE T T !NF</td>
</tr>
<tr>
<td>!NE T F !NF</td>
</tr>
<tr>
<td>!NE T T NF</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

initialize v1 and v2 to False

inQ f0 v1 s1 f1 v2 s2 f2 outQ
Explicit encoding of Valid/Invalid data

A bool to represent Valid/Invalid

initialize v1 and v2 to False

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

A bool to represent Valid/Invalid

inQ  v1  v2  outQ  action
NE  T  T  NF  s1<= f0(inQ.first); s2<= f1(s1); outQ.enq(f2(s2));
NE  T  T  !NF  no action
!NE  T  F  !NF
!NE  T  T  NF
...

initialize v1 and v2 to False
Explicit encoding of Valid/Invalid data

- **inQ**
- **s1**
- **v1**
- **s2**
- **v2**
- **outQ**

A bool to represent Valid/Invalid

**initialize v1 and v2 to False**

<table>
<thead>
<tr>
<th>inQ</th>
<th>v1</th>
<th>v2</th>
<th>outQ</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>NE</td>
<td>T</td>
<td>T</td>
<td>NF</td>
<td>s1&lt;=f0(inQ.first); s2&lt;= f1(s1); outQ.enq(f2(s2));</td>
</tr>
<tr>
<td>NE</td>
<td>T</td>
<td>T</td>
<td>!NF</td>
<td>no action</td>
</tr>
<tr>
<td>!NE</td>
<td>T</td>
<td>F</td>
<td>!NF</td>
<td>s2&lt;=f1(s1); v1&lt;=False; v2&lt;=True</td>
</tr>
<tr>
<td>!NE</td>
<td>T</td>
<td>T</td>
<td>NF</td>
<td>...</td>
</tr>
</tbody>
</table>
Explicit encoding of Valid/Invalid data

A bool to represent Valid/Invalid

initialize v1 and v2 to False

<table>
<thead>
<tr>
<th>inQ</th>
<th>v1</th>
<th>v2</th>
<th>outQ</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>NE</td>
<td>T</td>
<td>T</td>
<td>NF</td>
<td>s1=f0(inQ.first); s2=f1(s1); outQ.enq(f2(s2));</td>
</tr>
<tr>
<td>NE</td>
<td>T</td>
<td>T</td>
<td>!NF</td>
<td>no action</td>
</tr>
<tr>
<td>!NE</td>
<td>T</td>
<td>F</td>
<td>!NF</td>
<td>s2=f1(s1); v1=False; v2=True</td>
</tr>
<tr>
<td>!NE</td>
<td>T</td>
<td>T</td>
<td>NF</td>
<td>s2=f1(s1); outQ.enq(f2(s2)); v1=False;</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Explicit encoding of Valid/Invalid data

A bool to represent Valid/Invalid
inQ  v1  v2  outQ  action
NE  T  T  NF  s1<=f0(inQ.first); s2<= f1(s1); outQ.enq(f2(s2));
NE  T  T  NF  no action
!NE  T  F  !NF  s2<=f1(s1); v1<=False; v2<=True
!NE  T  T  NF  s2<=f1(s1); outQ.enq(f2(s2)); v1<=False;
...

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

```
rule stage1;
  fifo1.enq(f0(inQ.first));
  inQ.deq;  endrule
rule stage2;
  fifo2.enq(f1(fifo1.first));
  fifo1.deq;  endrule
rule stage3;
  outQ.enq(f2(fifo2.first));
  fifo2.deq;  endrule
```
Elastic pipeline
Use FIFOs instead of registers to connect stages

```
rule stage1;
    fifo1.enq(f0(inQ.first));
inQ.deq;  endrule

rule stage2;
    fifo2.enq(f1(fifo1.first));
    fifo1.deq; endrule

rule stage3;
    outQ.enq(f2(fifo2.first));
    fifo2.deq; endrule
```

- When can stage1 rule fire?
Elastic pipeline
Use FIFOs instead of registers to connect stages

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

```vhdl
rule stage1;
    fifo1.enq(f0(inQ.first));
inQ.deq;  endrule
rule stage2;
    fifo2.enq(f1(fifo1.first));
fifo1.deq; endrule
rule stage3;
    outQ.enq(f2(fifo2.first));
fifo2.deq; endrule
```
Elastic pipeline
Use FIFOs instead of registers to connect stages

\begin{center}
\begin{tikzpicture}
  \node (inQ) at (-2,0) {inQ};
  \node (fifo1) at (0,0) {fifo1};
  \node (fifo2) at (2,0) {fifo2};
  \node (outQ) at (4,0) {outQ};

  \draw[->] (inQ) -- (fifo1);
  \draw[->] (fifo1) -- (fifo2);
  \draw[->] (fifo2) -- (outQ);

  \node[draw,shape=circle] at ( fifo1 ) {$f_0$};
  \node[draw,shape=circle] at ( fifo2 ) {$f_1$};
  \node[draw,shape=circle] at ( outQ ) {$f_2$};
\end{tikzpicture}
\end{center}

\begin{itemize}
  \item \textbf{rule stage1;}
    \begin{itemize}
      \item fifo1.enq(f0(inQ.first));
      \item inQ.deq; \textbf{endrule}
    \end{itemize}
  \item \textbf{rule stage2;}
    \begin{itemize}
      \item fifo2.enq(f1(fifo1.first));
      \item fifo1.deq; \textbf{endrule}
    \end{itemize}
  \item \textbf{rule stage3;}
    \begin{itemize}
      \item outQ.enq(f2(fifo2.first));
      \item fifo2.deq; \textbf{endrule}
    \end{itemize}
\end{itemize}

- When can stage1 rule fire?
  - inQ has an element
  - fifo1 has space
- Can these 3 rules execute concurrently?
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
Elastic pipeline
Use FIFOs instead of registers to connect stages

```
rule stage1;
  fifo1.enq(f0(inQ.first));
  inQ.deq;  endrule
rule stage2;
  fifo2.enq(f1(fifo1.first));
  fifo1.deq; endrule
rule stage3;
  outQ.enq(f2(fifo2.first));
  fifo2.deq; endrule
```

- 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?
Elastic pipeline
Use FIFOs instead of registers to connect stages

rule stage1;
    fifo1.enq(f0(inQ.first));
inQ.deq;  endrule
rule stage2;
    fifo2.enq(f1(fifo1.first));
    fifo1.deq;  endrule
rule stage3;
    outQ.enq(f2(fifo2.first));
    fifo2.deq;  endrule

- 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
Elastic pipeline
Use FIFOs instead of registers to connect stages

```
rule stage1;
  fifo1.enq(f0(inQ.first));
  inQ.deq;  endrule
rule stage2;
  fifo2.enq(f1(fifo1.first));
  fifo1.deq;  endrule
rule stage3;
  outQ.enq(f2(fifo2.first));
  fifo2.deq;  endrule
```

- 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

Without concurrent execution it is hardly a pipelined system
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)
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
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?
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?
Repeatedly:
- Select any rule that is ready to execute
- Compute the state updates
- Make the state updates
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.
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*
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

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

Final value of \((x,y)\) given the initial values \((0,0)\)
Can these rules execute concurrently?
(without violating the one-rule-at-a-time-semantics)

Example 1

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

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

Concurrent Execution

\(ra < rb\)

\(rb < ra\)
Can these rules execute concurrently? (without violating the one-rule-at-a-time-semantics)

Example 1

```plaintext
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
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
```

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

Concurrent Execution: \((1,2)\)

- \(ra < rb\): \((1,2)\)
- \(rb < ra\)
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
```

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

Concurrent Execution

- \(ra < rb\): \((1, 2)\)
- \(rb < ra\): \((1, 2)\)
Can these rules execute concurrently?  
(without violating the one-rule-at-a-time-semantics)

Example 1

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

Concurrent Execution

Ra < Rb  =>  (1,2)
Rb < Ra  =>  (1,2)

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

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

Concurrent Execution

\(ra < rb\) \(\rightarrow\) \((1,2)\)

\(rb < ra\) \(\rightarrow\) \((1,2)\)
Can these rules execute concurrently? (without violating the one-rule-at-a-time-semantics)

Example 1

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

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

Concurrent Execution

\(ra < rb\)

\(rb < ra\)

\((1,2)\)

Conflict-Free (CF)
Can these rules execute concurrently? (without violating the one-rule-at-a-time-semantics)

Example 1

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

Concurrent Execution

```
ra < rb
```

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

```
Ex 1
```

```
(1,2)
```

```
Conflict-Free (CF)
```

Example 2

```plaintext
rule ra;
  x <= y+1;
endrule
rule rb;
  y <= x+2;
endrule
```
Can these rules execute concurrently? (without violating the one-rule-at-a-time-semantics)

Example 1

\[
\text{rule ra;}
\begin{align*}
&x \leq x+1; \\
&\text{endrule}
\end{align*}
\text{rule rb;}
\begin{align*}
&y \leq y+2; \\
&\text{endrule}
\end{align*}
\]

Example 2

\[
\text{rule ra;}
\begin{align*}
&x \leq y+1; \\
&\text{endrule}
\end{align*}
\text{rule rb;}
\begin{align*}
&y \leq x+2; \\
&\text{endrule}
\end{align*}
\]

Concurrent Execution

\[
\text{ra < rb}
\]

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

Ex 1

\[
(1,2)
\]

Ex 2

\[
(1,2)
\]

Conflict-Free (CF)
Can these rules execute concurrently? (without violating the one-rule-at-a-time-semantics)

Example 1

\[
\text{rule } \text{ra}; \\
x \leq x+1; \\
\text{endrule} \\
\text{rule } \text{rb}; \\
y \leq y+2; \\
\text{endrule}
\]

Concurrent Execution

ra < rb

rb < ra

Example 2

\[
\text{rule } \text{ra}; \\
x \leq y+1; \\
\text{endrule} \\
\text{rule } \text{rb}; \\
y \leq x+2; \\
\text{endrule}
\]

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

Ex 1

(1,2) = (1,2)

Ex 2

(1,2) = (1,3)

Conflict-Free (CF)
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
```

Example 2

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

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

Ex 1

Concurrent Execution

ra < rb = (1,2) = (1,2)

rb < ra = (1,2)

Ex 2

ra < rb = (1,2)

rb < ra = (3,2)

Conflict-Free (CF)
Can these rules execute concurrently? (without violating the one-rule-at-a-time-semantics)

Example 1

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

Concurrent Execution

- ra < rb: (1,2) ≠ (1,2)
- rb < ra: Conflict-Free (CF)

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

Ex 1

- (1,2) ≠ (1,3)

Ex 2

```plaintext
rule ra;
  x <= y+1;
endrule
rule rb;
  y <= x+2;
endrule
```
Can these rules execute concurrently? (without violating the one-rule-at-a-time-semantics)

Example 1

\begin{verbatim}
rule ra;
  x <= x+1;
endrule
rule rb;
  y <= y+2;
endrule
\end{verbatim}

Example 2

\begin{verbatim}
rule ra;
  x <= y+1;
endrule
rule rb;
  y <= x+2;
endrule
\end{verbatim}

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

Ex 1

Concurrent Execution

\[
\begin{align*}
ra & < rb \\
(1,2) & = (1,2)
\end{align*}
\]

Ex 2

Concurrent Execution

\[
\begin{align*}
rb & < ra \\
(1,2) & = (3,2)
\end{align*}
\]

Conflicting-Free (CF)
Can these rules execute concurrently? (without violating the one-rule-at-a-time-semantics)

Example 1

\begin{verbatim}
rule ra;
  x <= x+1;
endrule
rule rb;
  y <= y+2;
endrule
\end{verbatim}

Concurrent Execution

\begin{align*}
\text{ra} &< \text{rb} \\
(1,2) \quad &\neq (1,2) \\
\text{rb} &< \text{ra} \\
(1,2) \quad &\neq (1,3)
\end{align*}

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

Example 2

\begin{verbatim}
rule ra;
  x <= y+1;
endrule
rule rb;
  y <= x+2;
endrule
\end{verbatim}

\begin{align*}
\text{ra} &< \text{rb} \\
(1,2) \quad &\neq (1,3) \\
\text{rb} &< \text{ra} \\
(3,2) \quad &\neq (3,2)
\end{align*}

\textbf{Conflict-Free (CF)}
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
```

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 \((0,0)\):

- **Concurrent Execution**
  - **ra < rb**
    - Ex 1: \((1,2) = (1,2)\)
    - Ex 2: \((1,2) = (1,2)\)
    - Ex 3: \((1,2) \neq (1,3)\)

- **rb < ra**
  - Concurrency-Free (CF)
    - Ex 1: \((1,2) = (1,2)\)
    - Ex 2: \((1,2) \neq (3,2)\)
    - Ex 3: Conflict
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
(1,2) ≠ (1,3)

rb < ra
(1,2) ≠ (3,2)
Conflict-Free (CF)
Conflict

Example 2

(rule ra;
  x <= y+1;
endrule)

(rule rb;
  y <= x+2;
endrule)

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)

Example 3

(rule ra;
  x <= y+1;
endrule)

(rule rb;
  y <= y+2;
endrule)
Can these rules execute concurrently? (without violating the one-rule-at-a-time-semantics)

Example 1

\[
\text{rule ra; } \\
\quad \begin{array}{c}
\text{x <= x+1; } \\
\text{endrule}
\end{array} \\
\text{rule rb; } \\
\quad \begin{array}{c}
\text{y <= y+2; } \\
\text{endrule}
\end{array}
\]

Concurrent Execution

ra < rb

rb < ra

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

Ex 1

Ex 2

Ex 3

Example 2

\[
\text{rule ra; } \\
\quad \begin{array}{c}
\text{x <= y+1; } \\
\text{endrule}
\end{array} \\
\text{rule rb; } \\
\quad \begin{array}{c}
\text{y <= x+2; } \\
\text{endrule}
\end{array}
\]

Example 3

\[
\text{rule ra; } \\
\quad \begin{array}{c}
\text{x <= y+1; } \\
\text{endrule}
\end{array} \\
\text{rule rb; } \\
\quad \begin{array}{c}
\text{y <= y+2; } \\
\text{endrule}
\end{array}
\]

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

Ex 1

Ex 2

Ex 3

Conflicting-Free (CF)

Conflicting
Can these rules execute concurrently? (without violating the one-rule-at-a-time-semantics)

<table>
<thead>
<tr>
<th>Example 1</th>
<th>Example 2</th>
<th>Example 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{rule ra;}</td>
<td>\textbf{rule ra;}</td>
<td>\textbf{rule ra;}</td>
</tr>
</tbody>
</table>
| \begin{align*} x & \leq x+1; \\
| \textbf{endrule}   | \begin{align*} x & \leq y+1; \\
| \textbf{rule rb;}  | \textbf{endrule}   | \textbf{endrule}   |
| \begin{align*} y & \leq y+2; \\
| \textbf{endrule}   |                     |                     |

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

- **Concurrent Execution**
  - Ex 1: \((1,2) = (1,2)\)
  - Ex 2: \((1,2) \neq (1,3)\)
  - Ex 3: \((1,2) = (1,2)\)

- \(ra < rb\)
  - Ex 1: \((1,2)\)
  - Ex 2: \((1,3)\)
  - Ex 3: \((1,2)\)

- \(rb < ra\)
  - \textbf{Conflict-Free (CF)}
  - Ex 2: \textbf{Conflict}
  - Ex 3: \((3,2)\)

\(\text{November 6, 2018}\)
Can these rules execute concurrently? (without violating the one-rule-at-a-time-semantics)

Example 1

\[
\text{rule ra;} \quad x \leq x+1; \\
\text{endrule} \\
\text{rule rb;} \quad y \leq y+2; \\
\text{endrule}
\]

Example 2

\[
\text{rule ra;} \quad x \leq y+1; \\
\text{endrule} \\
\text{rule rb;} \quad y \leq x+2; \\
\text{endrule}
\]

Example 3

\[
\text{rule ra;} \quad x \leq y+1; \\
\text{endrule} \\
\text{rule rb;} \quad y \leq y+2; \\
\text{endrule}
\]

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

Concurrent Execution

Ex 1: \(\langle 1,2 \rangle = \langle 1,2 \rangle\)

ra < rb

Ex 2: \(\langle 1,2 \rangle \neq \langle 1,3 \rangle\)

rb < ra

Conflict-Free (CF)

Ex 3: \(\langle 1,2 \rangle = \langle 1,2 \rangle\)

ra < rb

rb < ra

Conflict

November 6, 2018
Can these rules execute concurrently? (without violating the one-rule-at-a-time-semantics)

<table>
<thead>
<tr>
<th>Example</th>
<th>Rule 1</th>
<th>Rule 2</th>
<th>Concurrent Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example 1</td>
<td>\texttt{rule ra; x (\leq) x+1; endrule}</td>
<td>\texttt{rule rb; y (\leq) y+2; endrule}</td>
<td>ra &lt; rb: ((1,2)) \text{ Conflict-Free (CF)}</td>
</tr>
<tr>
<td>Example 2</td>
<td>\texttt{rule ra; x (\leq) y+1; endrule}</td>
<td>\texttt{rule rb; y (\leq) x+2; endrule}</td>
<td>rb &lt; ra: ((1,2)) ≠ ((1,3)) Conflict</td>
</tr>
<tr>
<td>Example 3</td>
<td>\texttt{rule ra; x (\leq) y+1; endrule}</td>
<td>\texttt{rule rb; y (\leq) y+2; endrule}</td>
<td>rb &lt; ra: ((1,2)) ≠ ((3,2)) Conflict</td>
</tr>
</tbody>
</table>

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

- \text{Ex 1: } \((1,2)\) = \((1,2)\)
- \text{Ex 2: } \((1,2)\) ≠ \((1,3)\) Conflict
- \text{Ex 3: } \((1,2)\) = \((1,2)\)
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
```

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
```

Concurrent Execution

- Ex 1
  - ra < rb
  - Final value of (x,y) given the initial values (0,0)
    - (1,2)
    - Conflict-Free (CF)

- Ex 2
  - rb < ra
  - Final value of (x,y) given the initial values (0,0)
    - (1,2) ≠ (1,3)
    - Conflict

- Ex 3
  - ra < rb
  - Final value of (x,y) given the initial values (0,0)
    - (1,2) ≠ (3,2)
    - ra < rb
Conflict Matrix (CM)

BSV compiler generates the pairwise conflict information

Example 1

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

Example 2

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

Example 3

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

Example 3

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

- C : rules can’t be executed concurrently

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

<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>

- **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)

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

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

Example 3
```plaintext
rule ra;
  x <= y+1;
endrule
rule rb;
  y <= y+2;
endrule
```
Conflict Matrix (CM)
BSV compiler generates the pairwise conflict information

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>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>

<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>

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

Example 1
\begin{verbatim}
rule ra;
    x <= x+1;
endrule
rule rb;
    y <= y+2;
endrule
\end{verbatim}

<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
\begin{verbatim}
rule ra;
    x <= y+1;
endrule
rule rb;
    y <= x+2;
endrule
\end{verbatim}

<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
\begin{verbatim}
rule ra;
    x <= y+1;
endrule
rule rb;
    y <= y+2;
endrule
\end{verbatim}

<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)
Conflict Matrix (CM)
BSV compiler generates the pairwise conflict information

Example 1

```plaintext
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

```plaintext
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

```plaintext
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
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>
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
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
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
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
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

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
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

One-Element FIFO

<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></td>
<td></td>
</tr>
<tr>
<td>deq</td>
<td></td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>first</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
One-Element FIFO

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
One-Element FIFO

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
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
One-Element FIFO

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></td>
</tr>
<tr>
<td>deq</td>
<td>ME</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>first</td>
<td></td>
<td></td>
<td>CF</td>
</tr>
</tbody>
</table>

ME = mutually exclusive
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></td>
</tr>
<tr>
<td>first</td>
<td></td>
<td></td>
<td>CF</td>
</tr>
</tbody>
</table>

ME = mutually exclusive
One-Element FIFO

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></td>
</tr>
<tr>
<td>first</td>
<td>ME</td>
<td></td>
<td>CF</td>
</tr>
</tbody>
</table>

ME = mutually exclusive
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>ME</td>
<td>CF</td>
</tr>
</tbody>
</table>

ME = mutually exclusive
One-Element FIFO

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
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
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;
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;
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

module mkCFFifo (Fifo#(2, t));
    // instantiate da, va, db, vb
endmodule
module mkCFFifo (Fifo#(2, t));
    //instantiate da, va, db, vb

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

endmodule
module mkCFFifo (Fifo#(2, t));
    // instantiate da, va, db, vb

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

endmodule
module mkCFFifo (Fifo#(2, t));
    //instantiate da, va, db, vb
    
method Action enq(t x) if (!vb);
    begin
        db <= x;
        vb <= True;
    endmethod

method Action deq if (va);
    va <= False;
endmethod

method t first if (va);

endmodule
module mkCFFfifo (Fifo#(2, t));
// instantiate da, va, db, vb

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
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
Two-Element FIFO

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

<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>CF</td>
<td>CF</td>
</tr>
<tr>
<td>deq</td>
<td>CF</td>
<td>C</td>
<td>&gt;</td>
</tr>
<tr>
<td>first</td>
<td>CF</td>
<td>&lt;</td>
<td>CF</td>
</tr>
</tbody>
</table>
Two-Element FIFO

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
Two-Element FIFO

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
Many other FIFO designs are possible

*Pipelined FIFO:* one can enq into a full FIFO if a deq is done simultaneously
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><em>enq</em></th>
<th><em>deq</em></th>
<th>first</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>enq</em></td>
<td>C</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td><em>deq</em></td>
<td>&lt;</td>
<td>C</td>
<td>&gt;</td>
</tr>
<tr>
<td><em>first</em></td>
<td>&lt;</td>
<td>&lt;</td>
<td>CF</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
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>
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>enq</th>
<th>deq</th>
<th>first</th>
</tr>
</thead>
<tbody>
<tr>
<td>enq</td>
<td>C</td>
<td>&gt;</td>
</tr>
<tr>
<td>deq</td>
<td>&lt;</td>
<td>C</td>
</tr>
<tr>
<td>first</td>
<td>&lt;</td>
<td>&lt; CF</td>
</tr>
</tbody>
</table>

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

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

```plaintext
rule ra(p(x));
    m.f(x+1);
endrule
rule rb(q(x));
    m.g(x+2)
endrule
```
Concurrent rule execution

`rule ra(p(x));
  m.f(x+1);
endrule`

`rule rb(q(x));
  m.g(x+2)
endrule`
Concurrent rule execution

```
rule ra(p(x));
  m.f(x+1);
endrule
rule rb(q(x));
  m.g(x+2)
endrule
```
Concurrent rule execution

\begin{align*}
\textbf{rule } & ra(p(x)); \\
& \quad m.f(x+1);
\textbf{endrule} \\
\textbf{rule } & rb(q(x)); \\
& \quad m.g(x+2) \\
\textbf{endrule}
\end{align*}
Concurrent rule execution

rule ra(p(x));
  m.f(x+1);
endrule
rule rb(q(x));
  m.g(x+2)
endrule
Concurrent rule execution

\begin{verbatim}
rule ra(p(x));
  m.f(x+1);
endrule
rule rb(q(x));
  m.g(x+2)
endrule
\end{verbatim}
Concurrent rule execution

```
rule ra(p(x));
  m.f(x+1);
endrule
rule rb(q(x));
  m.g(x+2)
endrule
```
Concurrent rule execution

```plaintext
rule ra(p(x));
  m.f(x+1);
endrule
rule rb(q(x));
  m.g(x+2)
endrule
```

The diagram illustrates the execution flow with nodes and edges representing the rules and their interactions.
Concurrent rule execution

```plaintext
rule ra(p(x));
  m.f(x+1);
endrule
rule rb(q(x));
  m.g(x+2)
endrule
```
Concurrent rule execution

- This circuit will execute rules ra and rb concurrently

```plaintext
rule ra(p(x));
  m.f(x+1);
endrule
rule rb(q(x));
  m.g(x+2)
endrule
```
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 (⇒ methods f and g of m do not conflict)
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 (⇒ methods f and g of m do not conflict).
- Suppose rules ra and rb do conflict!
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 (⇒ methods f and g of m do not conflict).
Suppose rules ra and rb do conflict!
Need for a scheduler

\begin{align*}
\text{rule } ra \ (p(x)) ; \\
\quad m.f(x+1) ; \\
\text{endrule} \\
\text{rule } rb \ (q(x)) ; \\
\quad m.g(x+2) \\
\text{endrule}
\end{align*}

\begin{tikzpicture}
\node [draw] (scheduler) at (5,0) {Scheduler};
\node [draw,cloud] (p) at (1,1) {$p$};
\node [draw,cloud] (q) at (1,-1) {$q$};
\node (x) at (0,0) {$x$};
\node (f) at (5,0) {$f$};
\node (m) at (5,-2) {$m$};
\node (g) at (5,-4) {$g$};
\draw[-latex] (x) -- (p) node [midway, above] {+1};
\draw[-latex] (x) -- (q) node [midway, above] {+2};
\draw[-latex] (p) -- (f);
\draw[-latex] (q) -- (m);
\draw[-latex] (f) -- (m);
\draw[-latex] (p) -- (g);
\draw[-latex] (q) -- (g);
\end{tikzpicture}
Need for a scheduler

- Guards of all rules are fed to a scheduler

```plaintext
rule ra (p(x));
  m.f(x+1);
endrule
rule rb (q(x));
  m.g(x+2)
endrule
```
Need for a scheduler

- Guards of all rules are fed to a scheduler

```plaintext
rule ra (p(x));
  m.f(x+1);
endrule
rule rb (q(x));
  m.g(x+2)
endrule
```
Need for a scheduler

- Guards of all rules are fed to a scheduler

```
rule ra (p(x));
  m.f(x+1);
endrule
rule rb (q(x));
  m.g(x+2)
endrule
```
Need for a scheduler

- Guards of all rules are fed to a scheduler
- Using the CM, the scheduler lets only non-conflicting rules proceed

```plaintext
rule ra (p(x));
m.f(x+1);
endrule
rule rb (q(x));
m.g(x+2)
endrule
```

Scheduler

```
x
ra
+1
\( p \)
ra

rb
+2
\( q \)
```
Need for a scheduler

- Guards of all rules are fed to a scheduler
- Using the CM, the scheduler lets only non-conflicting rules proceed

```
rule ra (p(x));
m.f(x+1);
endrule
rule rb (q(x));
m.g(x+2)
endrule
```
Need for a scheduler

- Guards of all rules are fed to a scheduler
- Using the CM, the scheduler lets only non-conflicting rules proceed

```
rule ra (p(x));
  m.f(x+1);
endrule
rule rb (q(x));
  m.g(x+2)
endrule
```
### 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

```plaintext
rule ra (p(x));
  m.f(x+1);
endrule
rule rb (q(x));
  m.g(x+2);
endrule
```

![Scheduler Diagram](image)
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

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

Diagram:

- Scheduler
- ra → bodyA → gA
- rb → bodyB → gB
- f
A more complete picture
need for muxes

- Multiple rules may invoke the same method, so we need to put a mux in front of the interface
Multiple rules may invoke the same method, so we need to put a mux in front of the interface.
Multiple rules may invoke the same method, so we need to put a mux in front of the interface.
Multiple rules may invoke the same method, so we need to put a mux in front of the interface.
A more complete picture
need for muxes

- 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

![Diagram of rule scheduler]

- gs1
- gsn
- wfn
- wfs1

Scheduler
The rule scheduler

rule guards (aka can_fire signals)
The rule scheduler

rule guards (aka can_fire signals)

Scheduler

will_fire signals

gs1  gs

wfs1  wfn
The rule scheduler

- Guards (gs1 ... gsn) of many rules may be true simultaneously, and some of them may conflict.
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:
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
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.
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.
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-a-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

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