Increasing concurrency using bypasses and EHRs

Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology
Up-Down counter

module mkUpDownCounter (UpDownCounter);
    Reg#(Bit#(8)) ctr <- mkReg (0);
    method ActionValue#(Bit #(8)) up if (ctr < 255);
        ctr <= ctr+1; return ctr;
    endmethod
    method ActionValue#(Bit #(8)) down if (ctr > 0);
        ctr <= ctr-1; return ctr;
    endmethod
endmodule

Using the counter

UpDownCounter Bit#(8) x <- mkUpDownCounter;
rule producer;
    ... x.up ...;
endmethod
rule consumer;
    ... x.down ...;
endmethod

- methods up and down can be ready at the same time but if they are executed concurrently a double write error will occur
- Hence, rules producer and consumer cannot be allowed to execute concurrently either
Up-Down counter
How to avoid the double write error?

When producer’s `rdy` is True, it makes consumer’s `en` False, preventing it from making any state updates, and hence, no double write error.

Can we design an up and down counter where the up and down methods won’t conflict?
Rules for pipeline

```
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
```

- These rules must execute concurrently in a pipelined system
  - They can execute concurrently, only if fifos allow concurrent enq and deq
  - In our one-element fifo design, enq and deq were mutually exclusive!
Can we make a fifo where enq and deq can be done concurrently?
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, Bit #(n)));  // instantiate da, va, db, vb
  rule canonicalize if (vb && !va);
    da <= db;
    va <= True;
    vb <= False;
  endrule
method Action enq(Bit #(n) x) if (!vb);
  begin db <= x; vb <= True; end
endmethod
method Action deq if (va);
  va <= False;
endmethod
method Bit #(n) first if (va);
  return da;
endmethod
endmodule

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

Canonicalize rule introduces a dead cycle after an enq/deq.
Limitations of registers in Bluespec

- Using only registers, no *communication* can take place in the same clock cycle between
  - two methods or
  - two rules or
  - a rule and a method

- At times *bypassing* values between rules and methods is necessary to achieve high performance
Bypassing in Bluespec

- In Bluespec one thinks of bypassing in terms of reducing the number of cycles it takes to execute two conflicting rules or methods.
- For example, design a FIFO, where a rule can perform an `enq` on a full FIFO provided another rule performs a `deq` simultaneously.
  - requires signaling from `deq` to `enq`

- Another example: Transform the rules on the right so that they execute concurrently, and behave functionally as if `ra` happened before `rb` (`ra < rb`).
  - requires communicating the value of `x` from `ra` to `rb` in the same cycle.
  - Not possible in the subset of Bluespec you have seen so far!

```
rule ra;
  x <= y+1;
endrule
rule rb;
  y <= x+2;
endrule
```
New types of registers to enable bypassing

Normal Reg

Priority Reg

Functionally: \( w[0] < w[1] \)

Bypass Reg

Functionally: \( r[0] < w; \ w<r[1] \)

Priority Reg

Functionally: \( w[0] < w[1] \)

EHR

Functionally: \( w[0] < w[1]; \ w[0]<r[1]; \ r[0]<w[0]; \ r[0]<w[1]; \ r[1]<w[1]; \)
Ephemeral History Register (EHR)
Dan Rosenband [MEMOCODE’04]

- $r[1]$ returns:
  - the current state if $w[0]$ is not enabled
  - the value being written if $w[0]$ is enabled
- $w[1]$ has higher priority than $w[0]$

We will use EHRs to enhance concurrency in our designs but EHRs because internal bypass can increase the critical combinational path length.
Do up and down counter using EHRs

```
module mkUpDownCounter (UpDownCounter);
    Reg#(Bit#(8)) ctr <- mkReg (0);
    method ActionValue#(Bit#(8)) up if (ctr < 255);
        ctr <= ctr+1; return ctr;
    endmethod
    method ActionValue#(Bit#(8)) down if (ctr > 0);
        ctr <= ctr-1; return ctr;
    endmethod
endmodule
```

Replace ctr Reg by ctr EHR

```
module mkUpDownCounter (UpDownCounter);
    Ehr#(2, Bit#(8)) ctr <- mkEhr (0);
    method ActionValue#(Bit#(8)) up if (ctr[0] < 255);
        ctr[0] <= ctr[0]+1; return ctr[0];
    endmethod
    method ActionValue#(Bit#(8)) down if (ctr[1] > 0);
        ctr[1] <= ctr[1]-1; return ctr[1];
    endmethod
endmodule
```

Assuming, functionally we want to execute up before down, i.e., up < down

September 18, 2019
http://csg.csail.mit.edu/6.375
L07-12
Do up and down counter using EHRs: Analysis

```verbatim
module mkUpDownCounter (UpDownCounter);
  Ehr#(2, Bit#(8)) ctr <- mkEhr (0);
  method ActionValue#(Bit#(8)) up if (ctr[0] < 255);
      ctr[0] <= ctr[0]+1; return ctr[0];
  endmethod
  method ActionValue#(Bit#(8)) down if (ctr[1] > 0);
      ctr[1] <= ctr[1]-1; return ctr[1];
  endmethod
endmodule
```

- Method `down` will see the `ctr` value being written by method `up`.
- Method `down`’s write of `ctr` value will overwrite the `ctr` write by method `up`.
- The functionality of this counter is the same as the one using registers, except for one edge case.
  - EHR version would allow method `down` to be executed even when `ctr` is 0 provided method `up` is executed at the same time.
Inside the Up-Down counter with EHRs

```
module mkUpDownCounter (UpDownCounter);
    Ehr#(2, Bit#(8)) ctr <- mkEhr (0);
    method ActionValue#(Bit#(8)) up if (ctr[0] < 255);
        ctr[0] <= ctr[0]+1; return ctr[0];
    endmethod
    method ActionValue#(Bit#(8)) down if (ctr[1] > 0);
        ctr[1] <= ctr[1]-1; return ctr[1];
    endmethod
endmodule
```

No double write problem but potentially a longer combinational path
Using the EHR Up-Down counter

- No need to prevent the execution of the consumer!
- For proper circuit generation we need to reflect in the interface definition of the counter whether up and down methods can be called concurrently

```
UpDownCounter Bit#(8) x <- mkUpDownCounter;
rule producer;
  ... x.up ...;
endmethod
rule consumer;
  ... x.down ...;
endmethod
```
We can define a Conflict Matrix (CM), which specifies for a given pair of methods, or a pair of rules, or a method and rule, the effect of concurrent execution:

- **ra < rb**: ra and rb can be executed concurrently; the net effect is as if ra executed before rb.
- **ra CF rb**: ra and rb can be executed concurrently; the net effect is the same as (ra<rb) and (rb<ra).
- **ra C rb**: ra and rb Conflict; either the concurrent execution will cause a *double-write* error or the resulting effect is neither (ra<rb) nor (rb<ra).
- **ra ME rb**: the guards of ra and rb are mutually exclusive and thus, ra and rb can never be rdy together.
## Conflict Matrix of Primitive modules

### Registers and EHRs

<table>
<thead>
<tr>
<th>Register</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>

<table>
<thead>
<tr>
<th>EHR</th>
<th>EHR.r0</th>
<th>EHR.w0</th>
<th>EHR.r1</th>
<th>EHR.w1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ehr.r0</td>
<td>CF</td>
<td>&lt;</td>
<td>CF</td>
<td>&lt;</td>
</tr>
<tr>
<td>Ehr.w0</td>
<td>&gt;</td>
<td>C</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>Ehr.r1</td>
<td>CF</td>
<td>&gt;</td>
<td>CF</td>
<td>&lt;</td>
</tr>
<tr>
<td>Ehr.w1</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>C</td>
</tr>
</tbody>
</table>
CMs for Up-Down counter with and without EHR

```verilog
tmodule mkUpDownCounter (UpDownCounter);
    Reg#(Bit#(8)) ctr <- mkReg (0);
    method ActionValue#(Bit#(8)) up if (ctr < 255);
        ctr <= ctr+1; return ctr;
    endmethod
    method ActionValue#(Bit#(8)) down if (ctr > 0);
        ctr <= ctr-1; return ctr;
    endmethod
endmodule
tmodule mkUpDownCounter (UpDownCounter);
    Ehr#(2, Bit#(8)) ctr <- mkEhr (0);
    method ActionValue#(Bit#(8)) up if (ctr[0] < 255);
        ctr[0] <= ctr[0]+1; return ctr[0];
    endmethod
    method ActionValue#(Bit#(8)) down if (ctr[1] > 0);
        ctr[1] <= ctr[1]-1; return ctr[1];
    endmethod
endmodule
```

Given the CM, we can generate proper hardware for the users of these different modules.

<table>
<thead>
<tr>
<th></th>
<th>up</th>
<th>down</th>
</tr>
</thead>
<tbody>
<tr>
<td>up</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>down</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>up</th>
<th>down</th>
</tr>
</thead>
<tbody>
<tr>
<td>up</td>
<td>C</td>
<td>&lt;</td>
</tr>
<tr>
<td>down</td>
<td>&gt;</td>
<td>C</td>
</tr>
</tbody>
</table>
Designing FIFOs using EHRs

- **Pipeline FIFO:** An enq into a full FIFO is permitted provided a deq from the FIFO is done simultaneously (deq < enq)
- **Bypass FIFO:** A deq from an empty FIFO is permitted provided an enq into the FIFO is done simultaneously (enq < deq)
- **Conflict-Free FIFO:** Both enq and deq are permitted concurrently as long as the FIFO is not-full and not-empty
  - The effect of enq is not visible to deq, and vice versa

We will derive such FIFOs starting with one or two element FIFO implementations
Making One-Element FIFO into a Pipeline FIFO

module mkFifo (Fifo#(1, Bit#(n)));  
  Reg#(Bit#(n)) d <- mkRegU;  
  Ehr#(2, Bool) v <- mkEhr(False);

method Action enq(Bit#(n) x) if (!v[1]);
  v[1] <= True; d <= x;
endmethod

method Action deq if (v[0]);
  v[0] <= False;
endmethod

method Bit#(n) first if (v[0]);
  return d;
endmethod
endmodule
Making One-Element FIFO into a Bypassed FIFO

```verilog
module mkFifo (Fifo#(1, Bit#(n)));
    Ehr#(2, Bit#(n)) d <- mkEhr(?);
    Ehr#(2, Bool) v <- mkEhr(False);

method Action enq(Bit#(n) x) if (!v[0]);
    v[0] <= True;
    d[0] <= x;
endmethod

method Action deq if (v[1]);
    v[1] <= False;
endmethod

method Bit#(n) first if (v[1]);
    return d[1];
endmethod
endmodule
```

---

**Bypass FIFO CM**

<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>

- deq 'sees' enq
- v and d have the right values in all cases
- no double write error
Two-Element FIFO

```verilog
module mkCFFifo (Fifo#(2, Bit#(n)));
    Ehr#(2, Bit#(n)) da <- mkEhr(?);
    Ehr#(2, Bool) va <- mkEhr(False);
    Ehr#(2, Bit#(n)) db <- mkEhr(?);
    Ehr#(2, Bool) vb <- mkEhr(False);

    rule canonicalize (vb[1] && !va[1]);
        vb[1] <= False; endrule

    method Action enq(Bit#(n) x) if (!vb[0]);
        db[0] <= x; vb[0] <= True;
    endmethod

    method Action deq if (va[0]);
        va[0] <= False;
    endmethod

    method Bit#(n) first if (va[0]);
        return da[0];
    endmethod
endmodule
```

1. replace all registers by EHRs
2. since enq and deq happen first, assign them ports 0
3. assign canonicalize port 1

<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>&lt;</td>
</tr>
<tr>
<td>deq</td>
<td>CF</td>
<td>C</td>
<td>&gt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>first</td>
<td>CF</td>
<td>&lt;</td>
<td>CF</td>
<td>&lt;</td>
</tr>
<tr>
<td>cano</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>C</td>
</tr>
</tbody>
</table>

In any given cycle simultaneous enq and deq are permitted provided the FIFO is neither full nor empty.

September 18, 2019

http://csg.csail.mit.edu/6.375
Revisiting the rules for pipeline

---

**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**

These rules will execute concurrently provided we use Pipeline or Conflict-Free fifos

---

**Pipeline 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>&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>

**Conflict-free 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>CF</td>
<td>&gt;</td>
</tr>
<tr>
<td>deq</td>
<td>CF</td>
<td>C</td>
<td>&gt;</td>
</tr>
<tr>
<td>first</td>
<td>&lt;</td>
<td>&lt;</td>
<td>CF</td>
</tr>
</tbody>
</table>

---
Using EHRs

- EHRs can be used to design a variety of modules to reduce the conflict between its methods
  - FIFO, RF, Score Board, memory systems
- This way the user of such modules only has to understand the CM of the module, and not whether or how EHRs were used internally
- However, modules that use EHRs, e.g., bypass FIFO or pipeline FIFO, can increase the length of combinational paths and thus, affect the clock period
Serializability of Concurrent Execution of Rules
Serializability

- We could say that the concurrent execution of rules or methods is allowed as long as no double write error is possible.
- In fact, we impose the additional constraint of *serializability* on the concurrent execution of rules:

  *Serializability* means that a concurrent execution of rules must match some serial execution of rules, aka one-rule-at-a-time execution of rules.

- The serializability constrain is imposed to make it easier to analyze the behavior of concurrent systems; it is a common and well established practice all distributed systems and databases.

  In the hardware domain the idea of serializability is new at the design level but it has been used extensively in proving properties of the design.
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 without violating the one-rule-at-a-time semantics
Concurrent execution of rules

What results will these examples produce if we executed the two rules in each example concurrently

- There is no possibility of a double-write error
- But what does it mean to execute these rules concurrently

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
```
Concurrent Execution

Example 1

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

Example 2

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

Example 3

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

- We are allowed to read and write a register in the same clock cycle and when we do that the result of the read is the old value of the register; the value of the write is not visible until the next clock cycle
  - We show these values by writing a time as a superscript. Thus, $x^t$ is the old value and $x^{t+1}$ is the new value; For any $x$, if there is no $x^{t+1}$ defined, then $x^{t+1} = x^t$
- Assuming initially $x$ and $y$ are both 0, concurrent execution of the two rules in each of the three example will result in value 1 in $x$ and 2 in $y$
Executing \( ra \) before \( rb \) (\( ra < rb \))

### Example 1

```
rule ra;
  \( x_{t+1} \leq x_t + 1; \)
endrule
rule rb;
  \( y_{t+2} \leq y_{t+1} + 2; \)
endrule
```

### Example 2

```
rule ra;
  \( x_{t+1} \leq y_t + 1; \)
endrule
rule rb;
  \( y_{t+2} \leq x_{t+1} + 2; \)
endrule
```

### Example 3

```
rule ra;
  \( x_{t+1} \leq y_t + 1; \)
endrule
rule rb;
  \( y_{t+2} \leq y_{t+1} + 2; \)
endrule
```

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

<table>
<thead>
<tr>
<th>Concurrent Execution</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>( (1,2))</td>
<td>( (1,2))</td>
</tr>
<tr>
<td>( rb &lt; ra )</td>
<td>( (1,2))</td>
<td>( (1,3))</td>
<td>( (1,2))</td>
</tr>
</tbody>
</table>
Executing \( \text{rb} \) before \( \text{ra} \) \((\text{rb} < \text{ra})\)

**Example 1**

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

**Example 2**

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

**Example 3**

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

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

<table>
<thead>
<tr>
<th>Concurrent Execution</th>
<th>Ex 1</th>
<th>Ex 2</th>
<th>Ex 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{ra} &lt; \text{rb})</td>
<td>((1,2))</td>
<td>((1,2))</td>
<td>((1,2))</td>
</tr>
<tr>
<td>(\text{rb} &lt; \text{ra})</td>
<td>((1,2))</td>
<td>((3,2))</td>
<td>((3,2))</td>
</tr>
</tbody>
</table>
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 \((0,0)\)

<table>
<thead>
<tr>
<th>Concurrent Execution</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>Yes, Conflict-Free (CF)</td>
<td>No, Conflict</td>
<td>Yes, ra&lt;rb</td>
</tr>
</tbody>
</table>
Why is serializability important?

- As you have seen it is straightforward to build hardware so that ra and rb will execute concurrently. However, in general, it is difficult to derive the behavior of the resulting circuit.
- Serializability lets us apply one rule at a time in some order to derive the behavior of the composite system.
- Without serializability, the atomicity of each rule has no meaning in a complex system.
- Even though serializability imposes an additional constraint, and will make us reject some RTL implementations for a Bluespec design, in practice its advantages far outweigh its disadvantages in debugging and verification.

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

Initially (x,y) are (0,0)

Concurrent Execution

ra < rb (1,3)
rb < ra (3,2)
The derivation of CM

- There is a natural ordering between the values of CM entries
  \[ CF = \{<,>\} \]

  \[
  \begin{array}{c}
  \{<\} \\
  \{>\}
  \end{array}
  \]

  \[ C = \{\} \]

- This ordering permits us to take intersections of conflict information, e.g.,
  - \( \{>\} \cap \{<,>\} = \{>\} \)
  - \( \{>\} \cap \{<\} = \{\} \)

- We use the CM of primitive modules (register, EHR) to derive the CM for the interface methods of a module
Deriving the Conflict Matrix (CM) of a module interface

- Let g1 and g2 be the two methods defined by a module, such that

\[ \text{mcalls}(g1) = \{g11, g12, \ldots, g1n\} \]
\[ \text{mcalls}(g2) = \{g21, g22, \ldots, g2m\} \]

- \[ \text{CM}[g1, g2] = \text{conflict}(g11, g21) \land \text{conflict}(g11, g22) \land \ldots \land \text{conflict}(g12, g21) \land \text{conflict}(g12, g22) \land \ldots \land \text{conflict}(g1n, g21) \land \text{conflict}(g1n, g22) \land \ldots \]

- \[ \text{conflict}(x, y) = \text{if } x \text{ and } y \text{ are methods of the same module then } \text{CM}[x, y] \text{ else CF} \]

Compiler can derive the CM for a module by starting with the innermost modules in the module instantiation tree.
CM for rules

- The conflict between two rules or a rule and a method can be derived in a similar manner by examining the CM properties of the constituent method calls.

**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 <= y+2;
endrule
```

**Example 3**

```plaintext
rule ra;
  x <= y+1;
endrule
rule rb;
  y <= x+2;
endrule
```
Two-Element FIFO

Deriving the CM

<table>
<thead>
<tr>
<th>method Action enq(t x) if (!vb);</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (va) begin db &lt;= x; vb &lt;= True; end</td>
</tr>
<tr>
<td>else begin da &lt;= x; va &lt;= True; end</td>
</tr>
<tr>
<td>endmethod</td>
</tr>
<tr>
<td>method Action deq if (va);</td>
</tr>
<tr>
<td>if (vb) begin da &lt;= db; vb &lt;= False; end</td>
</tr>
<tr>
<td>else begin va &lt;= False; end</td>
</tr>
<tr>
<td>endmethod</td>
</tr>
</tbody>
</table>

We can derive a conservative CM by ignoring the conditionals

\[
\text{mcalls(enq)} = \{vb.r, va.r, db.w, vb.w, da.w, va.w\}
\]
\[
\text{mcalls(deq)} = \{va.r, vb.r, da.w, db.r, vb.w, va.w\}
\]

\[
\text{CM[enq,deq]} =
\]
\[
\text{CM[vb.r,va.r]} \cap \text{CM[vb.r,vb.r]} \cap \text{CM[vb.r,da.w]} \cap \text{CM[vb.r,db.r]} \cap \text{CM[vb.r,vb.w]} \cap \text{CM[vb.r,va.w]} \cap \text{CM[va.r,va.r]} \cap \text{CM[va.r,vb.r]} \cap \text{CM[va.r,da.w]} \cap \text{CM[va.r,db.r]} \cap \text{CM[va.r,vb.w]} \cap \text{CM[va.r,va.w]} \cap \text{CM[db.w,va.r]} \cap \text{CM[db.w,vb.r]} \cap \text{CM[db.w,da.w]} \cap \text{CM[db.w,db.r]} \cap \text{CM[db.w,vb.w]} \cap \text{CM[db.w,va.w]} \cap \text{CM[vb.w,va.r]} \cap \text{CM[vb.w,vb.r]} \cap \text{CM[vb.w,da.w]} \cap \text{CM[vb.w,db.r]} \cap \text{CM[vb.w,vb.w]} \cap \text{CM[vb.w,va.w]} \cap \text{CM[da.w,va.r]} \cap \text{CM[da.w,vb.r]} \cap \text{CM[da.w,da.w]} \cap \text{CM[da.w,db.r]} \cap \text{CM[da.w,vb.w]} \cap \text{CM[da.w,va.w]} \cap \text{CM[va.w,va.r]} \cap \text{CM[va.w,vb.r]} \cap \text{CM[va.w,da.w]} \cap \text{CM[va.w,db.r]} \cap \text{CM[va.w,vb.w]} \cap \text{CM[va.w,va.w]} \]
\[
= \text{CF} \cap \{<\} \cap \text{CF} \cap \{<\} \cap \{>\} \cap \{>\} \cap \text{C} \cap \text{C} \cap \{>\} \cap \text{C}
\]
\[
= \text{C}
\]