Bypassing and EHRs

Arvind
Computer Science and Artificial Intelligence Laboratory
M.I.T.
Bypassing is a technique to reduce the number of stalls (that is, the number of cycles) by providing extra data paths between the producer of a value and its consumer.
Bypassing is a technique to reduce the number of stalls (that is, the number of cycles) by providing extra data paths between the producer of a value and its consumer.
Bypassing is a technique to reduce the number of stalls (that is, the number of cycles) by providing extra data paths between the producer of a value and its consumer.
Bypassing is a technique to reduce the number of stalls (that is, the number of cycles) by providing extra data paths between the producer of a value and its consumer.
Bypassing

- Bypassing is a technique to reduce the number of stalls (that is, the number of cycles) by providing extra data paths between the producer of a value and its consumer.
- Bypassing introduces new combinational paths and this can increase combinational delay (and hence the clock period) and area.
Bypassing

- Bypassing is a technique to reduce the number of stalls (that is, the number of cycles) by providing extra data paths between the producer of a value and its consumer.
- Bypassing introduces new combinational paths and this can increase combinational delay (and hence the clock period) and area.
- The effectiveness of a bypass is determined by how often it is used.
Bypassing in Bluespec

- In Bluespec one thinks on bypassing in terms of reducing the number of cycles it takes to execute two conflicting rules or methods.
Bypassing in Bluespec

- In Bluespec one thinks on 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 is performs a deq simultaneously.
Bypassing in Bluespec

- In Bluespec one thinks on 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 is performs a deq simultaneously.
- Another example: Transform the rules on the right so that they execute concurrently, and ra < rb.

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

- In Bluespec one thinks on 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 is performs a deq simultaneously.

- Another example: Transform the rules on the right so that they execute concurrently, and ra < rb.

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

- In Bluespec one thinks on 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 is performs a deq simultaneously.
- Another example: Transform the rules on the right so that they execute concurrently, and ra < rb.
- That is, communicate the value of x from ra to rb in the same cycle.

```plaintext
rule ra;
   x <= y+1;
endrule
rule rb;
   y <= x+2;
endrule
```
In Bluespec one thinks on 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 is performs a deq simultaneously.

Another example: Transform the rules on the right so that they execute concurrently, and \( ra < rb \).

That is, communicate the value of \( x \) from \( ra \) to \( rb \) in the same cycle.

Not possible in the subset of Bluespec you have seen so far!
Limitations of registers
Limitations of registers

- Using the register primitive no communication can take place in the same atomic action (i.e. clock cycle)
Limitations of registers

- Using the register primitive no *communication* can take place in the same atomic action (i.e. clock cycle)
  - between two methods or
Limitations of registers

- Using the register primitive no *communication* can take place in the same atomic action (i.e. clock cycle)
  - between two methods or
  - between two rules or
Limitations of registers

- Using the register primitive no *communication* can take place in the same atomic action (i.e. clock cycle)
  - between two methods or
  - between two rules or
  - between a rule and a method
Limitations of registers

- Using the register primitive no *communication* can take place in the same atomic action (i.e. clock cycle)
  - between two methods or
  - between two rules or
  - between a rule and a method
Limitations of registers

- Using the register primitive no *communication* can take place in the same atomic action (i.e. clock cycle)
  - between two methods or
  - between two rules or
  - between a rule and a method

EHRs to rescue ...
Limitations of registers

- Using the register primitive no *communication* can take place in the same atomic action (i.e. clock cycle)
  - between two methods or
  - between two rules or
  - between a rule and a method

Ephemeral History Register (EHR): A primitive element to design modules with concurrent methods

EHRs to rescue ...
Ephemeral History Register (EHR)
Dan Rosenband [MEMOCODE’04]
Ephemeral History Register (EHR)
Dan Rosenband [MEMOCODE’04]
Ephemeral History Register (EHR)
Dan Rosenband [MEMOCODE’04]
Ephemeral History Register (EHR)
Dan Rosenband [MEMOCODE’04]
Ephemeral History Register (EHR)
Dan Rosenband [MEMOCODE’04]

- \( r[1] \) returns:
  - the current state if \( w[0] \) is \textit{not enabled}
  - the value being written if \( w[0] \) is \textit{enabled}

\( r[0] < w[0] \)
\( r[1] > w[0] \)
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]** takes precedence over **w[0]**
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

- \( w[1] \) takes precedence over \( w[0] \)

\( r[0] < w[0] \)

\( r[1] > w[0] \)
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]** takes precedence over \( w[0] \)
## 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>
## 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></td>
<td></td>
</tr>
<tr>
<td>EHR.w0</td>
<td>&gt;</td>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EHR.r1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EHR.w1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Conflict Matrix of Primitive modules: Registers and EHRs

**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>

**EHR**

<table>
<thead>
<tr>
<th></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></td>
</tr>
<tr>
<td>EHR.w0</td>
<td>&gt;</td>
<td>C</td>
<td>&lt;</td>
<td></td>
</tr>
<tr>
<td>EHR.r1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EHR.w1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**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></td>
</tr>
<tr>
<td>EHR.w0</td>
<td>&gt;</td>
<td>C</td>
<td>&lt;</td>
<td></td>
</tr>
<tr>
<td>EHR.r1</td>
<td>CF</td>
<td>&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EHR.w1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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></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></td>
<td></td>
</tr>
<tr>
<td>EHR.w1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## 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></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></td>
<td></td>
</tr>
<tr>
<td>EHR.w1</td>
<td>&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# 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></td>
<td></td>
</tr>
<tr>
<td>EHR.w1</td>
<td>&gt;</td>
<td>&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Conflict Matrix of Primitive modules: Registers and EHRs

### Registers

<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>

### EHRs

<table>
<thead>
<tr>
<th></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></td>
</tr>
<tr>
<td>EHR.w1</td>
<td>&gt;</td>
<td>&gt;</td>
<td></td>
<td>C</td>
</tr>
</tbody>
</table>
## 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>

### EHR

<table>
<thead>
<tr>
<th></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>
Designing FIFOs using EHRs

- Pipeline FIFO: An enq into a full FIFO is permitted provided a deq from the FIFO is done simultaneously
Designing FIFOs using EHRs

- **Pipeline FIFO:** An enq into a full FIFO is permitted provided a deq from the FIFO is done simultaneously.
- **Bypass FIFO:** A deq from an empty FIFO is permitted provided an enq into the FIFO is done simultaneously.
Designing FIFOs using EHRs

- **Pipeline FIFO:** An enq into a full FIFO is permitted provided a deq from the FIFO is done simultaneously.

- **Bypass FIFO:** A deq from an empty FIFO is permitted provided an enq into the FIFO is done simultaneously.

- **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 vise versa.
Designing FIFOs using EHRs

- **Pipeline FIFO**: An enq into a full FIFO is permitted provided a deq from the FIFO is done simultaneously.
- **Bypass FIFO**: A deq from an empty FIFO is permitted provided an enq into the FIFO is done simultaneously.
- **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.
One-Element FIFO
from L17

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
One-Element FIFO from L17

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 we make
- deq < enq ?
- enq < deq ?

ME = mutually exclusive
Making One-Element FIFO into a *Pipeline* 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>&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>

November 29, 2018
Making One-Element FIFO into a Pipeline 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
Making One-Element FIFO into a *Pipeline 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>&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>
Making One-Element FIFO into a *Pipeline* FIFO

```verilog
module mkFifo (Fifo#(1, t));
  Reg#(t) d <- mkRegU;
  Ehr#(2, Bool) v <- mkEhr(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>Pipeline FIFO CM</th>
</tr>
</thead>
<tbody>
<tr>
<td>enq</td>
</tr>
<tr>
<td>enq</td>
</tr>
<tr>
<td>deq</td>
</tr>
<tr>
<td>first</td>
</tr>
</tbody>
</table>

November 29, 2018
Making One-Element FIFO into a *Pipeline* FIFO

```verilog
module mkFifo (Fifo#(1, t));
    Reg#(t) d <- mkRegU;
    Ehr#(2, Bool) v <- mkEhr(False);
method Action enq(t x) (!v[1]);
    v <= True; d <= x;
endmethod
method Action deq if (v);
    v[0] <= 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>&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>
Making One-Element FIFO into a *Pipeline* FIFO

```verilog
module mkFifo (Fifo#(1, t));
  Reg#(t)  d  <= mkRegU;
  Ehr#(2, Bool) v  <= mkEhr(False);
method Action enq(t x) (!v[1]);
  v  <= True; d  <= x;
endmethod
method Action deq if (v[0]);
  v[0] <= False;
endmethod
method t first if (v[0]);
  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>&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>

Pipelined FIFO CM
Making One-Element FIFO into a *Pipeline* FIFO

module mkFifo (Fifo#(1, t));
    Reg#(t) d <- mkRegU;
    Ehr#(2, Bool) v <- mkEhr(False);
method Action enq(t x) if (!v[1]);
    v[1] <= True; d <= x;
endmethod
method Action deq if (v[0]);
    v[0] <= False;
endmethod
method t first if (v[0]);
    return d;
endmethod
endmodule
Making One-Element FIFO into a *Pipeline* FIFO

```
module mkFifo (Fifo#{1, t});
   Reg#{t} d <- mkRegU;
   Ehr#{2, Bool} v <- mkEhr(False);
method Action enq(t x) (!v[1]);
   v[1] <= True; d <= x;
endmethod
method Action deq if (v[0]);
   v[0] <= False;
endmethod
method t first if (v[0]);
   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>&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>

- enq ‘sees’ deq
- v has the right value in all cases
- no double write error
Making One-Element FIFO into a Pipeline FIFO

module mkFifo (Fifo#(1, t));
    Reg#(t) d <- mkRegU;
    Ehr#(2, Bool) v <- mkEhr(False);
method Action enq(t x) (!v[1]);
    v[1] <= True; d <= x;
endmethod
method Action deq if (v[0]);
    v[0] <= False;
endmethod
method t first if (v[0]);
    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>&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>

- enq 'sees' deq
- v has the right value in all cases
- no double write error
Making One-Element FIFO into a \textit{Bypassed} FIFO

\begin{verbatim}
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
\end{verbatim}
Making One-Element FIFO into a Bypassed 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
```

### 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>
Making One-Element FIFO into a Bypassed FIFO

module mkFifo (Fifo#(1, t));
    Reg#(t) d <- mkRegU;
    Ehr#(2, Bool) v <- mkEhr(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>&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>
Making One-Element FIFO into a *Bypassed* FIFO

```verilog
module mkFifo (Fifo#(1, t));
    Reg#(t) d <- mkRegU;
    Ehr#(2, Bool) v <- mkEhr(False);
method Action enq(t x) (!v[0]);
    v[0] <= 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>Bypass FIFO CM</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>
Making One-Element FIFO into a Bypassed FIFO

module mkFifo (Fifo#(1, t));
  Reg#(t)  d <- mkRegU;
Endmodule

method Action enq(t x) (!v[0]);
  v[0] <= True; d <= x;
endmethod

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

method t first if (v[1]);
  return d;
endmethod
Making One-Element FIFO into a Bypassed FIFO

module mkFifo (Fifo#(1, t));
    Reg#(t) d <- mkRegU;
    Ehr#(2, Bool) v <- mkEhr(False);
method Action enq(t x) (!v[0]);
    v[0] <= True; d <= x;
endmethod
method Action deq if (v[1]);
    v[1] <= False;
endmethod
method t first if (v[1]);
    return d;
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>
Making One-Element FIFO into a Bypassed FIFO

module mkFifo (Fifo#(1, t));
  Reg#(t) d <- mkRegU;
  Ehr#(2, Bool) v <- mkEhr(False);
method Action enq(t x) (!v[0]);
  v[0] <= True; d <= x;
  endmethod
method Action deq if (v[1]);
  v[1] <= False;
  endmethod
method t first if (v[1]);
  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>&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>
Making One-Element FIFO into a Bypassed FIFO

module mkFifo (Fifo#(1, t));
    Ehr#(2, t) d <- mkEhr(?);
    Ehr#(2, Bool) v <- mkEhr(False);
method Action enq(t x) (!v[0]);
    v[0] <= True; d[0] <= x;
endmethod
method Action deq if (v[1]);
    v[1] <= False;
endmethod
method t first if (v[1]);
    return d[1];
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>&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>
Making One-Element FIFO into a Bypassed FIFO

module mkFifo (Fifo#(1, t));
    Ehr#(2, t) d <- mkEhr(?);
    Ehr#(2, Bool) v <- mkEhr(False);

method Action enq(t x) (!v[0]);
    v[0] <= True; d[0] <= x;
endmethod

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

method t 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
Making One-Element FIFO into a Bypassed FIFO

module mkFifo (Fifo #(1, t));
    Ehr #(2, t) d <- mkEhr(?);
    Ehr #(2, Bool) v <- mkEhr(False);
method Action enq(t x) (!v[0]);
    v[0] <= True; d[0] <= x;
endmethod
method Action deq if (v[1]);
    v[1] <= False;
endmethod
method t 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 from L17

module mkCFFifo (Fifo #(2, t));
    Reg #(t) da <- mkRegU();
    Reg #(Bool) va <- mkReg(False);
    Reg #(t) db <- mkRegU();
    Reg #(Bool) vb <- mkReg(False);
    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>
<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>
Two-Element FIFO from L17

module mkCFFifo (Fifo #(2, t));
    Reg #(t) da <- mkRegU();
    Reg #(Bool) va <- mkReg(False);
    Reg #(t) db <- mkRegU();
    Reg #(Bool) vb <- mkReg(False)
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

Can we eliminate the dead cycle and make canonicalize execute concurrently with enq and deq?
module mkCFFifo (Fifo#(2, t));
    Reg#(t)  da <= mkRegU();
    Reg#(Bool) va <= mkReg(False);
    Reg#(t)  db <= mkRegU();
    Reg#(Bool) vb <= mkReg(False)
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 from L17
Two-Element FIFO
from L17

module mkCFFifo (Fifo#(2, t));
    Reg#(t) da <- mkRegU();
    Reg#(Bool) va <- mkReg(False);
    Reg#(t) db <- mkRegU();
    Reg#(Bool) vb <- mkReg(False)
    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

1. replace all registers by EHRs

<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>
Two-Element FIFO from L17

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

1. replace all registers by EHRs
Two-Element FIFO from L17

module mkCFFifo (Fifo#(2, t));
    Ehr#(2, t) da <- mkEhr(0);
    Ehr#(2, Bool) va <- mkEhr(False);
    Ehr#(2, t) db <- mkEhr(0);
    Ehr#(2, Bool) vb <- mkEhr(False);
    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
<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>

1. replace all registers by EHRs
2. since enq and deq happen first, assign them ports 0
module mkCFFifo (Fifo#(2, t));
    Ehr#(2, t) da <- mkEhr(?);
    Ehr#(2, Bool) va <- mkEhr(False);
    Ehr#(2, t) db <- mkEhr(?);
    Ehr#(2, Bool) vb <- mkEhr(False);
    rule canonicalize if (vb && !va);
        da <= db; va <= True; vb <= False;
    endrule
    method Action enq(t x) if (!vb[0]);
        db[0] <= x; vb[0] <= True;
    endmethod
    method Action deq if (va);
        va <= False;
    endmethod
    method t first if (va);
        return da;
    endmethod
endmodule

1. replace all registers by EHRs
2. since enq and deq happen first, assign them ports 0
module mkCFFifo (Fifo#(2, t));
    Ehr#(2, t) da <- mkEhr(?);
    Ehr#(2, Bool) va <- mkEhr(False);
    Ehr#(2, t) db <- mkEhr(?);
    Ehr#(2, Bool) vb <- mkEhr(False);
rule canonicalize if (vb && !va);
    da <= db; va <= True; vb <= False;
endrule
method Action enq(t x) if (!vb[0]);
    db[0] <= x; vb[0] <= True;
endmethod
method Action deq if (va[0]);
    va[0] <= False;
endmethod
method t first if (va[0]);
    return da[0]; endmethod
endmodule

table

<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>

1. replace all registers by EHRs
2. since enq and deq happen first, assign them ports 0
Two-Element FIFO from L17

module mkCFFifo (Fifo#(2, t));
    Ehr#(2, t) da <- mkEhr(?);
    Ehr#(2, Bool) va <- mkEhr(False);
    Ehr#(2, t) db <- mkEhr(?);
    Ehr#(2, Bool) vb <- mkEhr(False);
rule canonicalize if (vb && !va);
    da <= db; va <= True; vb <= False;
endrule
method Action enq(t x) if (!vb[0]);
    db[0] <= x; vb[0] <= True;
endmethod
method Action deq if (va[0]);
    va[0] <= False;
endmethod
method t first if (va[0]);
    return da[0]; endmethod
endmodule

<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>

1. replace all registers by EHRs
2. since enq and deq happen first, assign them ports 0
3. assign canocalize port 1
Two-Element FIFO
from L17

module mkCFFifo (Fifo#(2, t));

Ehr#(2, t) da <- mkEhr(?);
Ehr#(2, Bool) va <- mkEhr(False);
Ehr#(2, t) db <- mkEhr(?);
Ehr#(2, Bool) vb <- mkEhr(False);

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

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

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

method t 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 canocalize 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>
Two-Element FIFO
from L17

module mkCFFifo (Fifo#(2, t));

Ehr#(2, t) da <- mkEhr(?);
Ehr#(2, Bool) va <- mkEhr(False);
Ehr#(2, t) db <- mkEhr(?);
Ehr#(2, Bool) vb <- mkEhr(False);

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

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

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

method t first if (va[0]);
  return da[0];
endmethod
endmodule

<table>
<thead>
<tr>
<th>Method</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>

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

In any given cycles simultaneous enq and deq are permitted provided the FIFO is neither full nor empty
Two-Element FIFO from L17

module mkCFFifo (Fifo#(2, t));
    Ehr#(2, t) da <- mkEhr(?);
    Ehr#(2, Bool) va <- mkEhr(False);
    Ehr#(2, t) db <- mkEhr(?);
    Ehr#(2, Bool) vb <- mkEhr(False);
    rule canonicalize (vb[1] && !va[1]);
        vb[1] <= False; endrule
    method Action enq(t x) if (!vb[0]);
        db[0] <= x; vb[0] <= True;
    endmethod
    method Action deq if (va[0]);
        va[0] <= False;
    endmethod
    method t first if (va[0]);
        return da[0]; endmethod
endmodule

<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>

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

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

November 29, 2018
module mkRFile(RFile);
    Vector#(32,Reg#(Data)) rfile <- replicateM(mkReg(0));

    method Action wr(RIndx rindx, Data data);
        if(rindx!=0) rfile[rindx] <= data;
    endmethod

    method Data rd1(RIndx rindx) = rfile[rindx];
    method Data rd2(RIndx rindx) = rfile[rindx];
endmodule
module mkRFile(RFile);
    Vector#(32,Reg#(Data)) rfile <- replicateM(mkReg(0));

    method Action wr(RIndx rindx, Data data);
        if(rindx!=0) rfile[rindx] <= data;
    endmethod

    method Data rd1(RIndx rindx) = rfile[rindx];
    method Data rd2(RIndx rindx) = rfile[rindx];
endmodule
module mkRFile(RFile);
    Vector#(32,Reg#(Data)) rfile <- replicateM(mkReg(0));

    method Action wr(RIndx rindx, Data data);
        if(rindx!=0) rfile[rindx] <= data;
    endmethod

    method Data rd1(RIndx rindx) = rfile[rindx];
    method Data rd2(RIndx rindx) = rfile[rindx];
endmodule

Can we design a bypass register file so that:

\[
\text{wr} < \{\text{rd1, rd2}\}
\]
module mkBypassRFile(RFile);
    Vector#(32,Ehr#(2, Data)) rfile <-
        replicateM(mkEhr(0));

method Action wr(RIndx rindx, Data data);
    if(rindex!=0) (rfile[rindx])[0] <= data;
endmethod
method Data rd1(RIndx rindx) = (rfile[rindx])[1];
method Data rd2(RIndx rindx) = (rfile[rindx])[1];
endmodule
module mkBypassRFile(BypassRFile);

rule move;
endrule

method Action wr(RIndx rindx, Data data);
endmethod

method Data rd1(RIndx rindx) =

method Data rd2(RIndx rindx) =

endmodule
module mkBypassRFile(BypassRFile);
    RFile rf <- mkRFile;
    SFifo#(1, Tuple2#(RIndx, Data))
        bypass <- mkBypassSFifo;
rule move;
endrule
method Action wr(RIndx rindx, Data data);
endmethod
method Data rd1(RIndx rindx) =

method Data rd2(RIndx rindx) =
endmodule
Bypass Register File
with external bypassing

```verilog
module mkBypassRFile(BypassRFile);
    RFile rf <- mkRFile;
    SFifo#(1, Tuple2#(RIndx, Data))
        bypass <- mkBypassSFifo;
rule move;
   endrule
method Action wr(RIndx rindx, Data data);
    if (rindex!=0) bypass.enq(tuple2(rindx, data));
endmethod
method Data rd1(RIndx rindx) =

method Data rd2(RIndx rindx) =

endmodule
```
Bypass Register File with external bypassing

module mkBypassRFile(BypassRFile);
    RFile rf <- mkRFile;
    SFifo#(1, Tuple2#(RIndx, Data))
        bypass <- mkBypassSFifo;
rule move;
endrule
method Action wr(RIndx rindx, Data data);
    if (rindex!=0) bypass.enq(tuple2(rindx, data));
endmethod
method Data rd1(RIndx rindx) =
    return (!bypass.search1(rindx)) ? rf.rd1(rindx)
        : bypass.read1(rindx);
method Data rd2(RIndx rindx) =
endmodule
module mkBypassRFile(BypassRFile);
    RFile rf <- mkRFile;
    SFifo#(1, Tuple2#(RIndx, Data))
        bypass <- mkBypassSFifo;
rule move;
endrule
method Action wr(RIndx rindx, Data data);
    if (rindex!=0) bypass.enq(tuple2(rindx, data));
endmethod
method Data rd1(RIndx rindx) =
    return (!bypass.search1(rindx)) ? rf.rd1(rindx)
        : bypass.read1(rindx);
method Data rd2(RIndx rindx) =

endmodule

It is straightforward to provide a search method for a FIFO
module mkBypassRFile(BypassRFile);
    RFile    rf <- mkRFile;
    SFifo#(1, Tuple2#(RIndx, Data))
        bypass <- mkBypassSFifo;

    rule move;
        endrule
    method Action wr(RIndx rindx, Data data);
        if (rindex!=0) bypass.enq(tuple2(rindx, data));
    endmethod
    method Data rd1(RIndx rindx) =
        return (!bypass.search1(rindx)) ? rf.rd1(rindx)
        : bypass.read1(rindx);
    method Data rd2(RIndx rindx) =
        return (!bypass.search2(rindx)) ? rf.rd2(rindx)
        : bypass.read2(rindx);
endmodule

It is straightforward to provide a search method for a FIFO
Bypass Register File
with external bypassing

```
module mkBypassRFile(BypassRFile);
  RFile    rf <- mkRFile;
  SFifo#(1, Tuple2#(RIndx, Data))
    bypass <- mkBypassSFifo;
rule move;
  begin rf.wr(bypass.first); bypass.deq end;
endrule
method Action wr(RIndx rindx, Data data);
  if (rindex!=0) bypass.enq(tuple2(rindx, data));
endmethod
method Data rd1(RIndx rindx) =
  return (!bypass.search1(rindx)) ? rf.rd1(rindx)
  : bypass.read1(rindx);
method Data rd2(RIndx rindx) =
  return (!bypass.search2(rindx)) ? rf.rd2(rindx)
  : bypass.read2(rindx);
endmodule
```

It is straightforward to provide a search method for a FIFO
It is straightforward to provide a search method for a FIFO.
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
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 does not have to learn about EHRs unless he/she also wants to design new modules with different concurrency properties
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 does not have to learn about EHRs unless he/she also wants to design new modules with different concurrency properties
- However, the use of such modules, e.g., bypass FIFO or pipeline FIFO, can affect the delay of combinational paths and thus affect the clock period
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 does not have to learn about EHRs unless he/she also wants to design new modules with different concurrency properties
- However, the use of such modules, e.g., bypass FIFO or pipeline FIFO, can affect the delay of combinational paths and thus affect the clock period

let us illustrate the increase in clock period via an example
Two-Stage Pipeline processor
First attempt: correct code but inefficient code (L18)

Suppose we make pc and epoch EHRs

```verilog
rule doFetch;
  ... pc <= ppc; ... 
endrule

rule doExecute if (state != LoadWait);
  ...
  let mispred = eInst.nextPC != ppc;
  if (mispred) begin pc <= eInst.nextPC; epoch <= !epoch; end
  code to initiate memory ops and go to LoadWait if necessary
endrule
```

doFetch and doExecute conflict! Can't execute concurrently
Two-Stage Pipeline processor
Introducing EHRs

module mkProcTwoStageEHR(Empty);
    Instantiate rf, mem, misc registers, f2d fifo ...
    Ehr#(2, Word) pc <- mkEhr(0);
    Ehr#(2, Bool) epoch <- mkEhr(False);
rule doFetch;
endrule
rule doExecute if (state == Execute);
Two-Stage Pipeline processor
Introducing EHRs

module mkProcTwoStageEHR(Empty);
    Instantiate rf, mem, misc registers, f2d fifo ...
    Ehr#(2, Word) pc <- mkEhr(0);
    Ehr#(2, Bool) epoch <- mkEhr(False);
rule doFetch;
endrule

rule doExecute if (state == Execute);
    let inst <- mem.resp;
    let x = f2d.first; f2d.deq;
    let pcD = x.pc; let ppc = x.ppc; let epochD = x.epoch;
    if (epochD == epoch[0]) begin // right-path instruction
        code to compute eInst from inst
        let mispred = eInst.nextPC != ppc;
        if (mispred) begin pc[0] <= eInst.nextPC; epoch[0] <= !epoch[0]; end
        code to update the state; in case of a memory op, initiate
        memory req and in case of Ld go to LoadWait.
endrule
Two-Stage Pipeline processor
Introducing EHRs

module mkProcTwoStageEHR(Empty);

  Instantiate rf, mem, misc registers, f2d fifo ...
  Ehr#(2, Word) pc <- mkEhr(0);
  Ehr#(2, Bool) epoch <- mkEhr(False);

rule doFetch;
  iMem.req(MemReq{op: Ld, addr: pc, data: ?});
  f2d.enq(F2D {pc: pc[1], ppc: ppc, epoch: epoch[1]});
endrule

rule doExecute if (state == Execute);
  let inst <- mem.resp;
  let x = f2d.first; f2d.deq;
  let pcD = x.pc; let ppc = x.ppc; let epochD = x.epoch;
  if (epochD == epoch[0]) begin // right-path instruction
    code to compute eInst from inst
    let mispred = eInst.nextPC != ppc;
    if (mispred) begin pc[0] <= eInst.nextPC; epoch[0] <= !epoch[0]; end
    code to update the state; in case of a memory op, initiate
    memory req and in case of Ld go to LoadWait.
  endrule
Performance

- Total number of cycles reduced dramatically but clock period got worse by ~15%!
- Must look beyond cycle counts - synthesize to measure the real gain!

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Multi-Cycle (Cycles)</th>
<th>Two-Stage Bad (Cycles)</th>
<th>Two-Stage EHR (Cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcd_sw.S</td>
<td>3022</td>
<td>4403</td>
<td>1530</td>
</tr>
<tr>
<td>no_hazard.S</td>
<td>26</td>
<td>27</td>
<td>14</td>
</tr>
<tr>
<td>all_control_hazard.S</td>
<td>16</td>
<td>31</td>
<td>9</td>
</tr>
<tr>
<td>Clock Period</td>
<td>714.67ps</td>
<td>703.28ps</td>
<td>832.06ps</td>
</tr>
</tbody>
</table>
The project
Three stage pipeline (L18, Lab 7)

pc → epoch → rf → sb

fetch ← iMem → decode → execute

f2d ← d2e

dMem
The project
Three stage pipeline (L18, Lab 7)

Stalls due to data hazards reduce performance
Stalls due to data hazards reduce performance. Suppose we introduce a Bypass Register file, where read happen after write (wr < rd).
The project
Three stage pipeline (L18, Lab 7)

Stalls due to data hazards reduce performance
Suppose we introduce a Bypass Register file, where read happen after write (wr < rd)
Value being written in rf can be made visible to doDecode, avoiding a stall
The project
Three stage pipeline (L18, Lab 7)

Stalls due to data hazards reduce performance
Suppose we introduce a Bypass Register file, where read happen after write (wr < rd)
Value being written in rf can be made visible to doDecode, avoiding a stall
sb has to be changed as well otherwise decode would detect a unnecessary hazard
The effect of remove must be visible to search, i.e., remove < search
Scoreboard implementation using searchable Fifos

- The effect of `remove` must be visible to `search`, i.e., `remove < search`
- We can use BypassSFifo to make such a Bypassing Scoreboard
Scoreboard implementation using searchable Fifos

- The effect of remove must be visible to search, i.e., remove < search
- We can use BypassSFifo to make such a Bypassing Scoreboard

```verilog
module mkBypassScoreboard(Scoreboard#(size));
    SFifo#(size, Maybe#(RIndx), Maybe#(RIndx)) f <- mkBypassSFifo(isFound);
    method insert = f.enq;
    method remove = f.deq;
    method search1 = f.search1;
    method search2 = f.search2;
endmodule
```
Scoreboard implementation using searchable Fifos

- The effect of remove must be visible to search, i.e., remove < search
- We can use BypassSFifo to make such a Bypassing Scoreboard

```verilog
module mkBypassScoreboard(Scoreboard#(size));
    SFifo#(size, Maybe#(RIndx), Maybe#(RIndx))
        f <- mkBypassSFifo(isFound);
    method insert = f.enq;
    method remove = f.deq;
    method search1 = f.search1;
    method search2 = f.search2;
endmodule
```

```verilog
function Bool isFound (Maybe#(RIndx) dst, Maybe#(RIndx) src);
    return isValid(dst) && isValid(src) &&
        (fromMaybe(?,dst)==fromMaybe(?,src));
endfunction
```
Three-stage pipeline

```plaintext
rule doFetch; ... endrule
```
Three-stage pipeline

rule doFetch; ... endrule

rule doDecode;
  let inst <- iMem.resp;
  ... filter wrong-path instructions ...
  ... decode (inst) ...
  ... stall and save inst if data hazards ...
  ... read rf and enter sb ...
  d2e.enq(...);
endrule
Three-stage pipeline

**rule doFetch; ... endrule**

**rule doDecode;**
   **let inst <- iMem.resp;**
   ... filter wrong-path instructions ...
   ... decode (inst) ...
   ... stall and save inst if data hazards ...
   ... read rf and enter sb ...
   d2e.enq(...);
endrule

**rule doExecute (...);**
   **let dInst = d2e.first; d2e.deq;**
   ... filter wrong-path instructions ...
   ... execute (dInst, rval1, rval2, pc) ...
   ... detect and handle misprediction ...
   ... launch memory instructions if needed ...
   ... save info for the LoadWait step ...
   ... update rf for non-memory instructions...
endrule

**rule doLoadWait (...);**...
Need for bypassing can be reduced by code transformation

- The effectiveness of a bypass is determined by how often it is used.

- Compiler can also affect the effectiveness of a bypass by increasing the distance between the instruction that uses a value from the instruction that creates it.
Takeaway
Pipeline speed and throughput

\[ \text{f} \rightarrow \text{g} \rightarrow \text{h} \]
Pipeline speed and throughput

Longest combinational path determines the clock speed
Takeaway

Pipeline speed and throughput

Longest combinational path determines the clock speed
Pipeline speed and throughput

Longest combinational path determines the clock speed

Dividing the critical path into multiple steps improves the clock speed but may decrease the throughput
Longest combinational path determines the clock speed

Dividing the critical path into multiple steps improves the clock speed but may decrease the throughput
Takeaway
Pipeline speed and throughput

Longest combinational path determines the clock speed

Dividing the critical path into multiple steps improves the clock speed but may decrease the throughput

Pipelining any step can improve the throughput
Takeaway

Pipeline speed and throughput

Longest combinational path determines the clock speed

Dividing the critical path into multiple steps improves the clock speed but may decrease the throughput

Pipelining any step can improve the throughput

Bypassing in a pipeline system can further improve the throughput but may increase the clock period