Constructive Computer Architecture:

Scheduling Constraints on Interface methods

Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

September 20, 2017  http://csg.csail.mit.edu/6.175  L07-1
“Happens before” (\(<\)) relation

“happens before” relation between the methods of a module governs how the methods behave when called by a rule, action, method or exp

- \(f < g\) : \(f\) happens before \(g\)
  - \((g\) cannot affect \(f\) within an action\)
- \(f > g\) : \(g\) happens before \(f\)
- \(C\) : \(f\) and \(g\) conflict and cannot be called together
- \(CF\) : \(f\) and \(g\) are conflict free and do not affect each other

This relation is defined as a conflict matrix (CM) for the methods of primitive modules like registers and derived for the methods of all other modules
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 ordering

There is a natural ordering between the values of CM entries

\[ CF = \{<,>\} \]

\[ \{<\} \cap \{>,\} = \{>,\} \]

\[ \{>,\} \cap \{<\} = \{\} \]

This ordering permits us to take intersections of conflict information, e.g.,
Deriving the Conflict Matrix (CM) of a module interface

Let \( g_1 \) and \( g_2 \) be the two methods defined by a module, such that

\[
\text{mcalls}(g_1) = \{g_{11}, g_{12}, \ldots, g_{1n}\}
\]
\[
\text{mcalls}(g_2) = \{g_{21}, g_{22}, \ldots, g_{2m}\}
\]

\[
\text{conflict}(x, y) = \begin{cases} 
\text{CM}[x, y] & \text{if } x \text{ and } y \text{ are methods of the same module} \\
\text{CF} & \text{otherwise}
\end{cases}
\]

Derivation

\[
\text{CM}[g_1, g_2] = \text{conflict}(g_{11}, g_{21}) \cap \text{conflict}(g_{11}, g_{22}) \cap \ldots \\
\cap \text{conflict}(g_{12}, g_{21}) \cap \text{conflict}(g_{12}, g_{22}) \cap \ldots \\
\ldots \\
\cap \text{conflict}(g_{1n}, g_{21}) \cap \text{conflict}(g_{1n}, g_{22}) \cap \ldots
\]

Compiler can derive the CM for a module by starting with the innermost modules in the module instantiation tree.
Two-Element FIFO

Deriving the CM

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

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

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 C \cap C \cap \{>\} \cap C \\
= C
```

http://csg.csail.mit.edu/6.175
Two-Element FIFO
another implementation

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 both enq and deq execute concurrently? yes
But neither enq or deq execute again until the canonicalize rule fires!
...and canonicalize cannot execute concurrently with enq and deq!
⇒ Dead-cycle
Limitations of registers

- Can’t express a FIFO with concurrent enq and deq with no dead cycles!
- It is because in a language with only 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 ...
A new primitive element to design modules with concurrent methods
Ephemeral History Register (EHR)  Dan Rosenband [MEMOCODE’04]

\[ r[1] \text{ returns:} \]
\begin{itemize}
  \item the current state if \( w[0] \) is not enabled
  \item the value being written (\( w[0].\text{data} \)) if \( w[0] \) is enabled
\end{itemize}
\( w[i+1] \) takes precedence over \( w[i] \)

\[ r[0] < w[0] \quad r[1] < w[1] \quad w[0] < w[1] < \ldots \]
Conflict Matrix of Primitive modules: Registers and EHRs

<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>

<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>
Designing FIFOs using EHRs

- **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

- **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

We will derive such FIFOs starting with one and two element FIFO designs
Making One-Element FIFO into a Pipelined FIFO

```verilog
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
```

Pipelined FIFO behavior
- `deq < enq`
- `first < deq`
- `first < enq`

No double write error
module mkPipelineFifo(Fifo#(1, t));
  Reg#(t) d <- mkRegU;
  Ehr#(2, Bool) v <- mkEhr(False);

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

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

  method t first if (v[0]);
    return d;
  endmethod
endmodule

In any given cycle:
- If the FIFO is not empty then simultaneous enq and deq are permitted;
- Otherwise, only enq is permitted
Making One-Element FIFO into a Bypass FIFO

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

    method Action enq(t x) if (!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 behavior

enq < deq
first < deq
enq < first

No double write error
One-Element Bypass FIFO

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

Desired behavior
enq < deq
first < deq
enq < first

In any given cycle:
- If the FIFO is not full then simultaneous enq and deq are permitted;
- Otherwise, only deq is permitted
Making a Two-Element Conflict-Free FIFO

module mkCFFifo (Fifo#(2, t));
    Reg#(t) da <- mkRegU();
    Reg#(Bool) va <- mkReg(False);
    Reg#(t) db <- mkRegU();
    Reg#(Bool) vb <- mkReg(False)
rule canocalize (vb && !va);
    da <= db; va <= True;
    vb <= False; endrule
method Action enq(t x) if (!vb);
    db <= x; vb <= True;
endmethod
method Action deq if (va);
    va <= False;
endmethod
method t first if (va);
    return da; endmethod
endmodule

Desired behavior

1. Turn all registers into EHRs
2. Let enq and deq read and write 0\textsuperscript{th} port
3. Let canocalize read and write the 1\textsuperscript{st} port
Two-Element Conflict-free FIFO

```verilog
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
```

Desired behavior
- `enq` CF deq first < deq first CF enq

In any given cycle:
- Simultaneous `enq` and `deq` are permitted only if the FIFO is not full and not empty
CM for *Pipelined FIFO*

```verbatim
method Action enq(t x) if (!v[1]);
  d <= x; v[1] <= True; endmethod
method Action deq if (v[0]);
  v[0] <= False; endmethod
method t first if (v[0]);
  return d; endmethod

mcalls(enq)={v.r1, d.w, v.w1}
mcalls(deq)={v.r0, v.w0}
mcalls(first)={v.r0, d.r}
```

CM[enq,deq]= conflict[v.r1,v.r0] \(\cap\) conflict[v.r1,v.w0] \(\cap\)
conflict[d.w,v.r0] \(\cap\) conflict[d.w,v.w0] \(\cap\)
conflict[v.w1,v.r0] \(\cap\) conflict[v.w1,v.w0]

\(= \{>\} \cap \{>\} = \{>\}\)

*This is what we expected!*
The Bluespec Compiler compiles modules with (* synthesize *) attributes separately
- The inner-most modules are compiled first
- For each module, the compiler organizes rules into a list scheduler and computes which rules conflict with each other
- The compiler produces a CM for the interface methods which is used when compiling outer modules

Modules that are not compiled separately are effectively inlined wherever they are used

Currently the compiler doesn’t allow separate compilation of a module if it has interface parameters