Implementing Pipelining

Arvind
Computer Science & Artificial Intelligence Lab. Massachusetts Institute of Technology
# Arithmetic versus Instruction pipelining

The data items in an arithmetic pipeline are independent of each other.

In processors, older instructions in the pipeline may affect the younger ones:

- This causes pipeline stalls or requires other fancy tricks to avoid stalls.
- Processor pipelines are significantly more complicated than arithmetic pipelines.

Today's lecture is about the basics of parallel execution; L19, L20 and L21 will be about pipelined processors.

April 24, 2018
Inelastic vs Elastic pipeline

Inelastic: all pipeline stages move synchronously

Elastic: A pipeline stage can process data if its input FIFO is not empty and output FIFO is not Full

Most complex processor pipelines are a combination of the two styles
Elastic pipeline
Use FIFOs instead of pipeline registers

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

rule stage1;
    fifo1.enq(f1(inQ.first));
inQ.deq; endrule

rule stage2;
    fifo2.enq(f2(fifo1.first));
fifo1.deq; endrule

rule stage3;
    outQ.enq(f3(fifo2.first));
fifo2.deq; endrule
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?
Meaning of Multi-rule Systems

Repeatedly:
- Select a rule to execute
- Compute the state updates
- Make the state updates

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

Non-deterministic choice; User annotations can be used in rule selection

However, for performance we execute multiple rules concurrently whenever possible
Concurrent execution of rules

Two rules can execute concurrently, if concurrent execution would not cause a double-write error, and

The final state can be obtained by executing rules one-at-a-time in some sequential order.
Can these rules execute concurrently? (without violating the one-rule-at-a-time-semantics)

<table>
<thead>
<tr>
<th>Example 1</th>
<th>Example 2</th>
<th>Example 3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>rule</strong> ra; x &lt;= x+1; <strong>endrule</strong></td>
<td><strong>rule</strong> ra; x &lt;= y+1; <strong>endrule</strong></td>
<td><strong>rule</strong> ra; x &lt;= y+1; <strong>endrule</strong></td>
</tr>
<tr>
<td><strong>rule</strong> rb; y &lt;= y+2; <strong>endrule</strong></td>
<td><strong>rule</strong> rb; y &lt;= x+2; <strong>endrule</strong></td>
<td><strong>rule</strong> rb; y &lt;= y+2; <strong>endrule</strong></td>
</tr>
</tbody>
</table>

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

- **Ex 1**: Concurrent Execution: \(r_a < r_b\)
  - \((1,2,2)\)
  - No Conflict

- **Ex 2**: Concurrent Execution: \(r_b < r_a\)
  - \((1,2,2)\)
  - Conflict

- **Ex 3**: Concurrent Execution: \(r_a < r_b\)
  - \((1,2,2)\)
  - Conflict
Conflict Matrix (CM)

BSV compiler generates the pairwise conflict information

Example 1

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

Example 2

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

Example 3

```latex
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>
<tr>
<td>ra</td>
<td>C</td>
<td>CF</td>
</tr>
<tr>
<td>rb</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>

ra C rb : rules can’t be executed concurrently
ra < rb : rules can be executed concurrently; the net effect is as if ra executed before rb
CF: rules can be performed concurrently; the net effect is the same with both rule orders
Conflict Matrix for an Interface

Conflict Matrix (CM) defines which methods of a module can be called concurrently.

CM for a register:

<table>
<thead>
<tr>
<th></th>
<th>reg.r</th>
<th>reg.w</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg.r</td>
<td>CF</td>
<td>&lt;</td>
</tr>
<tr>
<td>reg.w</td>
<td>&gt;</td>
<td>C</td>
</tr>
</tbody>
</table>

- Two reads can be performed concurrently.
- Two concurrent writes conflict and are not permitted.
- A read and a write can be performed concurrently and it behaves as if the read happened before the write.

CM of a register is used systematically to derive the CM for the interface of a module and the CM for rules.

A few examples...
One-Element FIFO

```verilog
module mkFifo (Fifo#(1, t));
    Reg#(t) d <- mkRegU;
    Reg#(Bool) v <- mkReg(False);
    method Action enq(t x) if (!v);
        v <= True; d <= x;
    endmethod
    method Action deq if (v);
        v <= False;
    endmethod
    method t first if (v);
        return d;
    endmethod
endmodule
```

enq and deq are mutually exclusive and therefore can never execute concurrently.

This FIFO is not useful for implementing pipelined system.

enq and deq are mutually exclusive and therefore can never execute concurrently.

This FIFO is not useful for implementing pipelined system.
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 $da$. 

April 24, 2018


L18-12
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 neither enq or deq can execute again until
the canonicalize rule fires!

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

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

**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>C</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>&gt;</td>
<td>C</td>
<td>&gt;</td>
</tr>
<tr>
<td>&gt;</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

*Design of such FIFOs requires using BSV features that we won’t have time to teach

However, we will provide primitives to make such FIFOs so that you can use them in your own designs
Hardware generation using *conflict* (CM) information
Preliminaries

Recall from L08, BSV compiler generates a combinational circuit for each rule and method.

If rule or method sets a register $x$ then it must generate both the data and the enable signal for the register, e.g.

```
rule foo(p(x)); x <= f(x); endrule
```

Similarly for each action method and actionValue method.
Preliminaries – need for muxes

We associate a control wire $v_i$ with each value $x_i$; $x_i$ has a meaningful value only if its corresponding $v_i$ is true.

When we merge two or more values, at most one $v_i$ should be true at any given time (*one-hot-encoding*), i.e., $v_i$’s must be pairwise mutually exclusive.

$x$, $x_1$, and $x_2$ are bit vectors and must have the same size.

BSV compiler ensures this.

\[
x = (v_1 \& x_1) \mid (v_2 \& x_2)
\]

\[
v = v_1 \mid v_2
\]
Need for conflict information

```verilog
module mkEx (...);
    Reg#(t) x <- mkReg(0);
    method Action f(t a);
        x <= x+a;
    endmethod
    method Action g(t b);
        x <= b;
    endmethod
endmodule
```

- Note at most one of f.en and g.en should be true; otherwise this circuit is not meaningful
- *How does the compiler ensure that?*
- CM to rescue: CM for mkEx will show that methods f and g conflict and should never be called at the same time
Using CM

module mkEx (...);
    Reg#(t) x <= mkReg(0);
    method Action f(t a);
        x <= x+a;
    endmethod
    method Action g(t b);
        x <= b;
    endmethod
endmodule

The CM for mkEx will show that methods f and g conflict

Suppose m <- mkEx();

rule ra;... m.f(1); m.g(2);... endrule
ra is an illegal rule

rule rb;... m.f(1);... endrule
rule rc;... m.g(2);... endrule
rb and rc should not be scheduled concurrently, and executed one by one

how?
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 rule scheduler

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

BSV compiler constructs a combinational scheduler circuit with the following property:

for all \( i \) and \( j \), if \( \text{wfs}_i \) and \( \text{wfs}_j \) are true then the corresponding \( \text{gs}_i \) and \( \text{gs}_j \) must be true and rules \( i \) and \( j \) must not conflict with each other.
Circuit with a scheduler

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

- The scheduler is generated based on the CM of ra and rb, which in turn depends upon the CM of m.
- Generally a scheduler has small number of gates.
- A correct but low performance scheduler may schedule only one rule at a time.
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).
Takeaway

- One-rule-at-a-time semantics are very important to understand what behaviors a system can show.
- Efficient hardware for multi-rule system requires that many rules execute in parallel without violating the one-rule-at-time semantics.
- BSV compiler builds a scheduler circuit to execute as many rules as possible concurrently.
- For high-performance designs we have to worry about the CM characteristics of our modules.