CS250B: Modern Computer Systems
Bluespec Introduction – More Topics

Sang-Woo Jun

Many slides adapted from Arvind’s MIT “6.175: Constructive Computer Architecture” and Hyoukjun Kwon’s Gatech “Designing CNN Accelerators”
More Topics

- Rule Scheduling
- Static Elaboration
- Polymorphism
- Nested Interfaces
Rule Scheduling In Bluespec

- For each rule, the compiler adds two signals
  - CAN_FIRE: Explicit and implicit guards are all met (rule -> scheduler)
  - WILL_FIRE: The rule will fire in the current cycle (rule <- scheduler)
- If there is no conflict between all rules, CAN_FIRE == WILL_FIRE for all
- If there are conflicts, some rules will be scheduled less urgent
  - CAN_FIRE == True, but WILL_FIRE == False for some rules that wait until conflicting rules’ CAN_FIRE becomes false

XXX.sched example

```
Rule: pcieCtrl RelayTLPm
Predicate: pcieCtrl_sendTLPQ.i_notFull &&
>   pcieCtrl_sendTLPm_mb_outQ.i_notEmpty &&
>     (pcieCtrl_dataWordsRemain == 10'd0)
Blocking rules: pcieCtrl_generateHeaderTLP
```
Valid Concurrent Rules

- Simplified story: Rules that write to the same state can’t fire together
  - Except when they can
  - Also, sometimes rules that don’t write to same state can’t fire together
  - Still safe guideline to try to follow

Conflict?  No!

```
Reg#(Bit#(32)) x <- mkReg(0);
rule rule1;
x <= x + 1;
endrule
rule rule2;
x <= 3;
endrule
```

Conflict?  Yes...

```
Reg#(Bit#(32)) x <- mkReg(0);
Reg#(Bit#(32)) y <- mkReg(0);
rule rule1;
x <= y;
endrule
rule rule2;
y <= x;
endrule
```
Valid Concurrent Rules

“A set of rules $r_i$ can fire concurrently if there exists a total order between the rules such that all the method calls within each of the rules can happen in that given order”

- Rules $r_1$, $r_2$, $r_3$ can fire concurrently if there is an order $r_i$, $r_j$, $r_k$ such that $r_i < r_j$, $r_j < r_k$, and $r_i < r_k$ are all valid

What does it mean for rules to be ordered?
Conflict Matrix for an Interface

- Methods can have a “happens” before relationship between them
  - \( f < g : f \) happens before \( g \) (effects of \( g \) do not affect \( f \))
  - \( f \) \( C g : f \) and \( g \) conflict, and cannot be called together
  - \( f \) \( CF g : f \) and \( g \) are conflict-free, and can be called together

- Conflict Matrix (CM) defines which methods of a module can be called concurrently
  - e.g., Conflict matrix of 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>CF*</td>
</tr>
</tbody>
</table>

Conflict-free if called from different rules, conflict if called within same rule
Valid Concurrent Rules

<table>
<thead>
<tr>
<th>Reg#(Bit#(32)) x &lt;- mkReg(0);</th>
<th>Reg#(Bit#(32)) x &lt;- mkReg(0);</th>
<th>Reg#(Bit#(32)) x &lt;- mkReg(0);</th>
</tr>
</thead>
<tbody>
<tr>
<td>rule rule1; x &lt;= x + 1; endrule</td>
<td>rule rule1; x &lt;= x + 1; endrule</td>
<td>rule rule1; x &lt;= mkReg(0);</td>
</tr>
<tr>
<td>rule rule2; x &lt;= x + 3; endrule</td>
<td>rule rule2; x &lt;= 3; endrule</td>
<td>rule rule2; y &lt;= x; endrule</td>
</tr>
</tbody>
</table>

rule1 can’t fire before rule2 (rule1’s x.write > rule2’s x.read)
rule2 can’t fire before rule1 (rule2’s x.write > rule1’s x.read)
Conflict!

rule1 can fire before rule2
rule2 can’t fire before rule1 (rule2’s x.write > rule1’s x.read)
Conflict free! (rule1 < rule2)

rule1 can’t fire before rule2 (rule1’s x.write > rule2’s x.read)
rule2 can’t fire before rule1 (rule2’s y.write > rule1’s y.read)
Conflict!
Example: Up/Down Counter

Problem

```
Reg#(Bit#(32)) counter <- mkReg(0);
rule countUp;
  srcQ.deq;
  counter <= counter + 1;
endrule
rule countDown;
  destQ.deq;
  counter <= counter - 1;
endrule
```

Conflict!
This would be terrible for performance
(Duty cycle is half!)

Simple Solution

```
Reg#(Bit#(32)) counterUp <- mkReg(0);
Reg#(Bit#(32)) counterDn <- mkReg(0);
rule countUp;
  srcQ.deq;
  counterUp <= counterUp + 1;
endrule
rule countDown;
  destQ.deq;
  counterDn <= counterDn + 1;
endrule
rule displayCount;
  $display( "%d", counterUp-counterDn );
endrule
```

Conflict free!
More elegant solutions exist,
for another day
More Topics

- Rule Scheduling
- Static Elaboration
- Polymorphism
- Nested Interfaces
Static Elaboration

- Code that generates other code during compile time
  - Efficient way to write code – “Hardware Generator”
  - Example: Parallelizing high-latency GCD calculation

```plaintext
typedef 2 GCDCount;
Vector#(GCDCount, GCDIfc) gcd1 <- replicateM(mkGCD);

for (Integer idx = 0; idx < valueOf(GCDCount); idx=idx+1) begin
  rule rule1;
    cmdQ.deq; let cmd = cmdQ.first;
    gcd1.start(tpl_1(cmd), tpl_2(cmd));
  endrule
  rule rule2;
    cmdQ.deq; let cmd = cmdQ.first;
    gcd2.start(tpl_1(cmd), tpl_2(cmd));
  endrule
endrule
```

Note: Many rules competing for cmdQ is not good. More elegant solutions later!
Static Elaboration

- Also used within rules
  - Example: Parsing an array of values from a single large blob

```verilog
rule rule1;
    Bit#(128) rawdata = inQ.first; inQ.deq;
    Vector#(8, Int#(8)) parsed;
    for (Integer i = 0; i < 8; i=i+1 ) begin
        parsed[i] = unpack(truncate(rawdata>>(i*8)));
    end
    outQ.enq(parsed);
endrule
```

- Not only for loops, but if statements can be used as well
- As long as they all use variables/literals available at compile time
Static Elaboration

- Powerful way to define complex modules
  - A lot of things can be statically elaborated! (Interface, etc)
  - More to come!

- Very complex modules can be defined recursively
  - Example: `MergeSort#(fanIn, type) NToOneMerge <- mkMergesorter;
  - But in order to give an example of recursive module definition, we must first cover polymorphic modules
More Topics

- Rule Scheduling
- Static Elaboration
- Polymorphism
- Nested Interfaces
Polymorphism – Parameterized Interfaces

- Interface parameterized with types: “#” syntax
  - `Bit#(32) data;
  - `FIFO#(Bit#(32)) exampleQ <- mkFIFO;
  - `Vector#(4, Int#(64)) exampleVector = newVector();

```verilog
interface GCDIfc#(type valType);
  method Action start(valType a, valType b);
  method valType result();
endinterface

module mkGCD (GCDIfc#(valType));
  ...
  method Action start(valType a, valType b);
  ...
endmethod
methods valType result();
  ...
endmethod
endmodule

GCDIfc#(Bit#(32)) gdc32 <- mkGCD;
GCDIfc#(Bit#(48)) gdc48 <- mkGCD;
...etc
```
module mkGCD (GCDIfc#(valType));
    Reg#(valType) x <- mkReg(?);
    Reg#(valType) y <- mkReg(0);
rule step1 (x > y && y != 0);
    x <= y; y <= x;
endrule
rule step2 (x <= y && y != 0);
    y <= x - x;
endrule
method Action start(valType a, valType b) if (y==0);
    ...
endmethod
method valType result() If (y==0);
    ...
endmethod
endmodule
Polymorphism – Provisos

Compiles and works with provisos telling compiler about valType

```verilog
module mkGCD (GCDIfc#(valType))
provisos(Bits#(valType, valTypeSz), Literal#(valType), Arith#(valType), Eq#(valType), Ord#(valType));
Reg#(valType) x <- mkReg(?);
Reg#(valType) y <- mkReg(0);
...
endmodule
```

- valType can be anything, Bit#, Int#, Vector#, typedef struct, as long as the provisos are satisfied.
  - Literal#, Arith#, Ord# not satisfied for Vector#, struct, ...

Now “valTypeSz” is a numeric type available in the module
Polymorphism – numeric type

interface GCDIfc#(type valType);
    method Action start(valType a, valType b);
    method valType result();
endinterface

If we know valType is Bit#, and we only care about width

interface GCDIfc#(numeric type valSz);
    method Action start(Bit#(valSz) a, Bit#(valSz) b);
    method Bit#(valSz) result();
endinterface

GCDIfc#(32) gdc32 <- mkGCD;
...etc

Don’t need provisos because Bit# satisfies everything
Polymorphism – More Provisos

Say, the output has to be 32 bits

interface GCDIfc#(numeric type valSz);
  method Action start(Bit#(valSz) a, Bit#(valSz) b);
  method Bit#(32) result();
endinterface

module mkGCD (GCDIfc#(valSz));
  Reg#(start(Bit#(valSz)) x <- mkReg(?);
  Reg#(start(Bit#(valSz)) y <- mkReg(0);
  ...
  method Bit#(32) result() If (y==0);
  return truncate(x);
endmethod
endmodule

Error: “Top.bsv”, line 23, column 8: (10085)
The provisos for this expression are too general.
Given type:
  _n_#(Top::GCDIfc#(valType))
With the following provisos:
  IsModule#(_n_, _n_)
The following additional proviso is needed:
  Add#(a__, 32, valType)
This proviso was introduced by an extend or truncate operation, which requires that the extended size be larger.
The extend or truncate occurs in or at the following locations:
  “Top.bsv”, line 43, column 24
Makefile:8: recipe for target `all' failed
make: *** [all] Error 1

Requires Add# provisos to tell compiler valType is larger than or equal to 32

Side effect: a__ = valType – 32 available in module
Polymorphism – Arithmetic Provisos

❑ Used to assert relationships between interface parameters
❑ Also used like typedefs inside module context
  o Regular typedef not allowed in module
  o Provisos creates a type name if an argument does not exist in context already

<table>
<thead>
<tr>
<th>Class</th>
<th>Proviso</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>Add#(n1,n2,n3)</td>
<td>Assert $n1 + n2 = n3$</td>
</tr>
<tr>
<td>Mul</td>
<td>Mul#(n1,n2,n3)</td>
<td>Assert $n1 * n2 = n3$</td>
</tr>
<tr>
<td>Div</td>
<td>Div#(n1,n2,n3)</td>
<td>Assert $\text{ceiling } \frac{n1}{n2} = n3$</td>
</tr>
<tr>
<td>Max</td>
<td>Max#(n1,n2,n3)</td>
<td>Assert $\text{max}(n1,n2) = n3$</td>
</tr>
<tr>
<td>Min</td>
<td>Min#(n1,n2,n3)</td>
<td>Assert $\text{min}(n1,n2) = n3$</td>
</tr>
<tr>
<td>Log</td>
<td>Log#(n1,n2)</td>
<td>Assert $\text{ceiling } \log_2(n1) = n2.$</td>
</tr>
</tbody>
</table>

Example module with valType1 and valType2 available as interface parameters

```vhdl
provisos(Log#(valType1, valTypeLog), Max#(valType1, valType2, valTypeMax));
// Types valTypeLog, valTypeMax now available within module context
```
Polymorphism – Type Operations

- Say, we want to correctly handle multiplication overflows
  - Return value needs to be double the width of the input
  - Use “Type Functions”: Types as input, type as output

```
interface SafeMultIfc#(numeric type valSz);
  method Action put(Bit#(valSz) a, Bit#(valSz) b);
  method Bit#(TMul#(2, valSz)) result();
endinterface

Reg#(Bit#(TMul#(2, valSz)) res <- mkReg(0);
```

<table>
<thead>
<tr>
<th>Type Function</th>
<th>Size Relationship</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAdd</td>
<td>TAdd#(n1,n2)</td>
<td>Calculate n1 + n2</td>
</tr>
<tr>
<td>TSub</td>
<td>TSub#(n1,n2)</td>
<td>Calculate n1 – n2</td>
</tr>
<tr>
<td>TMul</td>
<td>TMul#(n1,n2)</td>
<td>Calculate n1 * n2</td>
</tr>
<tr>
<td>TDiv</td>
<td>TDiv#(n1,n2)</td>
<td>Calculate n1/n2</td>
</tr>
<tr>
<td>TLog</td>
<td>TLog#(n1)</td>
<td>Calculate ceiling n1/n2</td>
</tr>
<tr>
<td>TExp</td>
<td>TExp#(n1)</td>
<td>Calculate 2^n1</td>
</tr>
<tr>
<td>TMax</td>
<td>TMax#(n1,n2)</td>
<td>Calculate max(n1,n2)</td>
</tr>
<tr>
<td>TMin</td>
<td>TMin#(n1,n2)</td>
<td>Calculate min(n1,n2)</td>
</tr>
</tbody>
</table>

“TXxx” naming convention
Type Arithmetic Examples

- `Int#(TAdd#(5,n)) a;`
  - `n` must be in scope somewhere

- `typedef TAdd#(vsize, 8) Bigsize#(numeric type vsize);`
  - `typedef` parameterized with numeric type `vsize`
  - Usage example: `Bit#(Bigsize#(32))` is a 40-bit Bit variable

- `typedef Bit#(TLog#(n)) CBToken#(numeric type n);`
  - `typedef` parameterized with numeric type `n`
  - Usage example: `CBToken#(8)` is a 3-bit variable

- `typedef 8 Wordsize; typedef TAdd#(Wordsize, 1) Blocksize;`
Simple Example: Maybe# types

- Maybe# is a polymorphic tagged union type
  - Parameterized with type t
  - isValid, fromMaybe are helper functions

```haskell
typedef union tagged {
  void Invalid; // void is a zero-bit type
  t Valid;
} Maybe#(type t) deriving (Eq, Bits);

case (x) matches
  tagged Valid .a : return a;
  tagged Invalid : return 0;
endcase
```
Polymorphic Functions

- We have seen parameterized interface/modules and types
- Functions can be parameterized as well
  - Requires usual provisos, etc

```plaintext
function Bool equal (valType x, valType y)
  provisos( Eq#(valType) );
  return (x==y);
endfunction
// Now equal(x,y) used with any type satisfying Eq
```
More Topics

- Rule Scheduling
- Static Elaboration
- Polymorphism
- Nested Interfaces
Nested Interfaces

- Interfaces can be hierarchical
  - Interfaces in other interfaces
- Syntax Example:

```plaintext
interface SubIfc;
  method Bit#(64) getResult;
endinterface

interface TestIfc;
  method Action put(Bit#(32) data);
  method Bit#(32) result();
endmethod
interface SubIfc sub;
  method Bit#(64) getResult;
endmethod
endinterface

module mkTest;
...
interface TestIfc;
  method Action put(Bit#(32) data);
  method Bit#(32) result();
endmethod
interface SubIfc sub;
  method Bit#(64) getResult;
endmethod
endinterface
endmodule

TestIfc test <- mkTest;
rule ...
  result = test.sub.getResult;
endrule
```
Nested Vector Interfaces

- Nested interfaces can be a vector of interfaces
- Static elaboration to populate

```verilog
define SubIfc;
  method Bit#(64) getResult;
endinterface

interface TestIfc;
  method Action put(Bit#(32) data);
endinterface

interface Vector#(8, SubIfc) sub;
endinterface

module mkTest;
...
interface TestIfc;
...
Vector#(8, SubIfc) sub_;
for (Integer i = 0; i < 8; i=i+1) begin
  sub_[i] = interface SubIfc;
  method Bit#(64) getResult;
  ...
endinterface
end

interface sub = sub_;
endmodule
```
Tested Interface Use Cases

- N-to-1, N-to-N connections between modules
  - e.g., Switch connected to multiple clients:
    - interface SwitchIfc has a parameterized vector of N ClientConnectionIfc
    - Each module mkClient instance takes as argument one ClientConnectionIfc
  - e.g., Data sink has multiple sources
    - interface SinkIfc has multiple interfaces SinkSubIfc
    - Multiple rules, statically elaborated, call each SinkSubIfc
Connectables

- Concise way to connect interfaces of two modules together
  - Needs “import Connectable::*”
  - Can connect value and action method pairs
    * Cannot deal with ActionValue methods!

```plaintext
interface SourceIfc;
  method Bit #(64) getResult;
endinterface

interface SinkIfc;
  method Action put(Bit #(64) data);
endinterface

rule ...
  sink.put(source.getResult);
endrule

mkConnection(sink.put, source.getResult);
```

Option 1: Cumbersome with many pairs

Option 2: Concise!
Connectables – GetPut

- Get# and Put# interfaces can be used to use ActionValue methods
  - Needs to “import GetPut::*;”
  - Get#(t) has one method, “ActionValue#(t) get”
  - Put#(t) has one method, “Action put(t data)”

```plaintext
interface SourceIfc;
  interface Get#(Bit#(64)) getins;
endinterface

interface SinkIfc;
  interface Put#(Bit#(64)) putins;
endinterface

module mkSource (SourceIfc);
  interface Get getins;
  method ActionValue#(Bit#(64)) get;
  ...
endmodule

module mkSink (SinkIfc);
  interface Put putins;
  method Action get(Bit#(64) d);
  ...
endmodule

mkConnection(sink.putins,source.getins);
```
Putting It All Together: Recursive Module Definition

- Example use case: “MergeN” module
  - Collects data from N sources and funnels it into one stream, tagged with the source index
  - Can be recursively defined using a tree of Merge2’s

```
<srcid, data>
```

```
data
data
data
...```

```
data

```
MergeN Recursive Definition

Parameterized with n and t:
Variable number of sources,
Any type (satisfying Bits),

```haskell
interface MergeEnqIfc#(type t);
  > method Action enq(t d);
endinterface

interface MergeNIfc#(numeric type n, type t);
  > interface Vector#(n, MergeEnqIfc#(t)) enq;
  > method Action deq;
  > method t first;
endinterface

module mkMergeN (MergeNIfc#(n,t))
  > provisos(Bits#(t, a__));
  > if ( valueOf(n) == 1 ) begin
  >   > FIFO#(t) inQ <- mkFIFO;
  >   > Vector#(n,MergeEnqIfc#(t)) enq_;
  >   > enq_[0] = interface MergeEnqIfc;
  >   >   > method Action enq(t d);
  >   >   >   > inQ.enq(d);
  >   >   > endmethod
  >   > endinterface;
  >   > interface enq = enq_;
  >   > method Action deq;
  >   >   > inQ.deq;
  >   > endmethod
  >   > method t first;
  >   >   > return inQ.first;
  >   > endmethod
  > end
```
MergeN Recursive Definition

else if (valueOf(n) == 2) begin
  Merge2Ifc#{(t)} mb <- mkMerge2;
  Vector#{(n, MergeEnqIfc#{(t)})} enq_
  for (Integer i = 0; i < valueOf(n); i = i + 1) begin
    enq_[i] = interface MergeEnqIfc;
    method Action enq(t d);
    > > mb.enq[i].enq(d);
    > endmethod
    endinterface;
  end
  interface enq = enq_
  method Action deq;
  > mb.deq;
  endmethod
  method t first;
  > return mb.first;
  endmethod
  end
```haskell
> else begin
>   Vector#(2, MergeN!fc#(TDiv#(n, 2), t)) ma <- replicateM(mkMergeN);
>   Merge2!fc#(t) mb <- mkMerge2;
>   for ( Integer i = 0; i < 2; i = i + 1 ) begin
>     rule ma[i].deq;
>     mb.enq[i].enq(ma[i].first);
>   endrule
> end
>
> Vector#(n, MergeEnq!fc#(t)) enq_;
> for ( Integer i = 0; i < valueOf(n); i = i + 1 ) begin
>   enq_[i] = interface MergeEnq!fc;
>     method Action enq(t d);
>       if ( i < valueOf(n)/2 ) begin
>         ma[0].enq[i%(valueOf(n)/2)].enq(d);
>       end else begin
>         ma[1].enq[i-(valueOf(n)/2)].enq(d);
>       end
>     endmethod
> endinterface;
> end
>
> interface enq = enq_;
> method Action deq;
>   mb.deq;
> endmethod
> method t first;
>   return mb.first;
> endmethod
> end
```

Continued