Lecture 11: Compiling, Running and Debugging

- Using bsc, the synthesis tool
- Simulating with a Verilog simulator
- Simulating with Bluesim, the direct simulator
- Understanding type-checking error messages
- Understanding scheduler error messages
**Internal Flow of bsc tool**

- Preprocessing: standard Verilog macros
- Parse: read files to internal representation
- Type check: resolve and check types
- Elaboration: a BSV program describes hardware or describes how to generate hardware
- Scheduler: analyze resources of rules/methods and generate logic for rule execution control
- Verilog/ sim generation: output final Verilog or Bluesim object code

**Error messages**

- Messages are flagged by the compiler phases that generate them
  - P parse error—incorrect syntax
  - T type checking error
  - G Generation which includes
    - Elaboration
    - Scheduler warning
    - Verilog or Simulation generation
Some useful bsc command line options

- List all command-line options
  - -help
- Verbose messages
  - -v
- Verilog or Bluesim
  - -verilog, -sim
- Setting output directories
  - -bdir, -simdir, -vdir, -info-dir
- Package management
  - -u recompiles imported modules if necessary
  - -p sets search path (+ is current path)
- Debugging
  - -keep-fires keeps scheduling signals
  - -show-schedule dumps details of schedule

Details in $BLUESPECDIR/../../doc/user-guide.pdf

Initial and Don’t-Care Values

- Initial values for state elements are the same as in Verilog: 0xAAAAAA...

- Don’t-care values are given a value according to the –unspecified-to flag, during compilation
  - Just as Verilog uses
  - Choices are 0, 1, A, X, Z
  - X and Z are not allowed for Bluesim
Simulating with Verilog

- –verilog flag causes Verilog modules to be created for modules marked by (* synthesis *) attribute, and/or named by –g command-line flag

- Combined with –e flag indicating the top module and the necessary Verilog files, it will invoke a third party Verilog simulator and make the simulation executable.
  - Example: bsc –verilog –e myTop –vsim <simulator> *.v

- Verilog simulators and synthesizers need access to Bluespec’s verilog primitives library— in $BLUESPECDIR/Verilog
  - Example: simulator +libext+.v -y $BLUESPECDIR/Verilog

Bluespec’s main.v

- Bluespec provides a driver module main.v to stimulate your design with a per default clock and reset

- It contains other as options, dump vcd files, print cycle count, etc.

- Verilog simulators need access to Bluespec’s simulation file $BLUESPECDIR/Verilog main.v

- Example:
  - simulator ... $BLUESPECDIR/Verilog/main.v +define+TOP=<top-module>

- You can replace this with your own main.v
Debugging Options

- **-keep-fires** turns off optimization which may remove **CAN_FIRE_rule** and **WILL_FIRE_rule** signals
  - Useful during “performance debugging”

- **-show-schedule** dumps details of schedule phase. i.e., conflicting rule, predicates, etc.

Debugging Simulation

- **Modules**
  - those with the (* synthesize *) attribute are preserved in Verilog as separate modules
  - Bluespec library primitives – e.g., Reg, FIFOs, etc.

- **Interfaces** – methods become Verilog ports (example follows)

- **Rules** – show actions occurring in design

- **Two signals show activity:**
  - **CAN_FIRE_rulename** – True when rule predicate (both implicit and explicit) conditions are all true
  - **WILL_FIRE_rulename** – True when CAN_FIRE is true and scheduler allows rule to fire
Library Module ports

- Primitives use instance$port for signal names
- E.g.,

```verilog
FIFO2 #( .width(16) ) testfifo (.CLK(CLK),
    .RST_N(RST_N),
    .D_IN(testfifo$D_IN),
    .ENQ(testfifo$ENQ),
    .DEQ(testfifo$DEQ),
    .D_OUT(testfifo$D_OUT),
    .CLR(testfifo$CLR),
    .FULL_N(testfifo$FULL_N),
    .EMPTY_N(testfifo$EMPTY_N));
```

// instance testfifo
assign testfifo$D_IN = dout + 16'd148 ;
assign testfifo$ENQ = WILL_FIRE_RL_store || WILL_FIRE_RL_write ;
assign testfifo$DEQ = WILL_FIRE_RL_get || WILL_FIRE_RL_update ;
assign testfifo$CLR = 1'b0 ;

Methods to Ports

```verilog
interface Example_ifc#(type any_t);
    method any_t first();
    method Action clear();
    method Action enqueue( any_t din );
endinterface: Example_ifc
```

- All methods have RDY signal
- Action methods have EN signal
- A value method’s output is its name
- Method arguments become inputs
Running Bluesim

- `bsc` generates `.ba` files (among others!)
  
  ```
  bsc -sim <file>.bsv
  ```

- Link these together with
  
  ```
  bsc -sim -e <top-level-mod> [-o <exefile>] *.ba
  ```

- Creates an executable `<exefile>`

![Diagram of Bluesim process]

Simulating and debugging with Bluesim

Simulator command line options

- `-help` show command line options
- `-m n` run simulator for `n` clock cycles

Powerful interactive capabilities via tcl interface (simulate until clock, single step, examine signals and state values, etc.)

Details in `$BLUESPECDIR/../doc/user-guide.pdf`
Bluesim can give you a significant speedup

<table>
<thead>
<tr>
<th>Designs</th>
<th>Bluesim (secs)</th>
<th>Fastest RTL Simulator (secs)</th>
<th>Speedup (factor)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wide GCD</td>
<td>5.53</td>
<td>39.7</td>
<td>7.18x</td>
</tr>
<tr>
<td>Life Game</td>
<td>8.37</td>
<td>283</td>
<td>33.9x</td>
</tr>
<tr>
<td>FIR Filter</td>
<td>4.23</td>
<td>61.8</td>
<td>14.6x</td>
</tr>
<tr>
<td>Upsize Converter</td>
<td>53.98</td>
<td>98.7</td>
<td>1.83x</td>
</tr>
<tr>
<td>Wallace Multiplier</td>
<td>48.66</td>
<td>417.0</td>
<td>8.57x</td>
</tr>
<tr>
<td>FIFO</td>
<td>11.73</td>
<td>124.2</td>
<td>10.6x</td>
</tr>
<tr>
<td>Mesa</td>
<td>37.45</td>
<td>126.4</td>
<td>3.37x</td>
</tr>
<tr>
<td>DIV3</td>
<td>8.24</td>
<td>116.8</td>
<td>14.2x</td>
</tr>
<tr>
<td>DES Core</td>
<td>28.55</td>
<td>89.36</td>
<td>3.13x</td>
</tr>
<tr>
<td>IDCT</td>
<td>15.69</td>
<td>16.32</td>
<td>1.04x</td>
</tr>
</tbody>
</table>

Bluespec Development Workstation

- GUI-based “project management”
  - Windows for project file listings, browsing packages, modules, types, rule schedules, with hyperlinks to jump between different views
- Hyperlinked to standard text editors (emacs, gvim)
- Single-click compile, link, execute
- Supports standard Verilog simulators (Modelsim, VCS, NCSim, iverilog, cver, ...)
- Hyperlinked to standard waveform viewers (Novas, Modelsim)
- More in Lecture 15.
The Bluespec compiler performs strong type checking to guarantee that values are only used in ways that make sense, according to their type.

Type checking errors can be frustrating… but

- It catches a huge class of design errors and typos statically (without simulation)!
- Strong Type checking replaces simulation-time debugging
- Frequent experience:
  - “once it type-checked, it just worked, first time!”
- Overall design time is shortened
Debugging Type errors

function Int#(3) sum( Int#(2) x, Int#(2) y) ;
  sum = x + y;
endfunction

Sum is of type Int#(3)
Bsc expects to find this.

EUnify.bsv", line 14, column 10: (T0020) Error:
Type error at:
x

Expected type:
Prelude::Int#(3)\}
Inferred type:
Prelude::Int#(2)\}

Copyright © Bluespec Inc. 2005-2008

Debugging Type errors

interface EUnify1 ;
   method Action start() ;
endinterface

... method Action start( ix, iy ) if ( y == 0 ) ;
x <= ix ;
y <= iy ;
endmethod

EUnify1.bsv", line 39, column 21: (T0020)
Error:
Type error at:
start

Expected type:
Action

Inferred type:
function Action f(Prelude::Int#(32) x1,
Prelude::Int#(32) x2)\}

Method start is of type Action
Bsc expects to find this.

Copyright © Bluespec Inc. 2005-2008
Debugging: scheduling

- Some sanity checks needed for scheduling
- if bsc can schedule the rules it will
- `show-schedule` option to bsc displays the schedule
- `sched-dot` option generate graph files for visualizing schedule (See development workstation)

- Guideline: There should be no error or (unexpected) warning about scheduling
  - e.g., dead rules, unexpected urgency
  - Also check rule conflicts

Scheduling example

```verilog
data module mkMult1 (Mult_ifc);
  Reg#(Tout) product <= mkReg(0);
  Reg#(Tout) d    <= mkReg(0);
  Reg#(Tin)    r   <= mkReg(0);

  rule cycle (r != 0);
    if (r[0] == 1) product <= product + d;
    d <= d << 1;
    r <= r >> 1;
  endrule

  rule swap (r > d[15:0] ) ;
    d <= {0,r} ;
    r <= d[15:0] ;
  endrule

  method Actio start (x,y) if (r == 0);
    d <= {0,x} ; r <= y; product <= 0;
  endmethod

  method result () if (r == 0);
    return product;
  endmethod
endmodule: mkMult1
```

- Multiplier example from earlier lecture
- This version includes a new "swap" rule
Scheduling result

"mult1.bsv", line 10, column 8: (G0010) Warning:
Rule "swap" was treated as more urgent than "cycle". Conflicts:
"swap" vs. "cycle":
calls to
d.write vs. d.read
r.write vs. r.read
"cycle" vs. "swap":
calls to
d.write vs. d.read
r.write vs. r.read

rule cycle (r != 0);
  if (r[0] == 1) product <= product+d;
  d <= d << 1;
  r <= r >> 1;
endrule
rule swap (r > d[15:0] ) ;
  d <= {0,r} ;
  r <= d[15:0] ;
endrule

Bsc cannot schedule rule swap and rule cycle together.
(They both update the same registers)
Bsc warns that it made an arbitrary decision

Example of -show-schedule

=== Generated schedule for mkMult1 ===
Method schedule
-----------
Method: start
  Ready signal: r == 0
  Sequenced after: result
  Conflicts: start

Method: result
  Ready signal: r == 0
  Sequenced before: start
  Conflicts: result

Rule schedule
-----------
Rule: swap
  Predicate: ! (r <= d[15:0])
  Blocking rules: (none)

Rule: cycle
  Predicate: ! (r == 0)
  Blocking rules: swap

Logical execution order: start, swap, cycle

Predicate includes implicit conditions
Blocking rules list resource conflict
Module behaves as if rules execute in this logical order.
Hardware execution is concurrent.
Specifying rule urgency

- Urgency annotation
  - (* descending_urgency = “swap, cycle” *)
  - Note position of quotation marks

- SystemVerilog “attribute” syntax

Removing conflicts

- Add rule conditions to disable swap and cycle from occurring together.

```verbatim
let swapem = (r[13:0] > d[15:2]);
rule cycle ((r != 0) && (!swapem));
  if (r[0] == 1)
    product <= product + d;
  d <= d << 1;
  r <= r >> 1;
endrule
rule swap (swapem);
  d <= {0, r};
  r <= d[15:0];
endrule
```

- Explicit condition `swapem` added
- Rules predicates are provably mutually exclusive
- Need to reason that only one swap will occur
More resources

Documents in the Bluespec software distribution:
- $BLUESPECDIR/../doc/reference-guide.pdf
- $BLUESPECDIR/../doc/user-guide.pdf
- $BLUESPECDIR/../doc/style-guide.pdf
- $BLUESPECDIR/../doc/kpns.pdf
- $BLUESPECDIR/../doc/verification-guide.pdf
- $BLUESPECDIR/../doc/quick-reference.pdf

Many examples in the Bluespec software distribution
- $BLUESPECDIR/..training/

Open on-line “forum” for Bluespec community discussions, help, news, etc.
- http://www.bluespec.com/forum/

On-line Tutorials, examples
- http://www.bluespec.com/support/

Or directly...
- support@bluespec.com

End of Lecture