Hardware Synthesis from Bluespec

Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology
Module as a sequential machine

- Inputs and outputs are defined by the type of the module, i.e., its interface definition
  - A read method has no enable input wire
  - An Action method has no output data wires
  - An ActionValue method has ready, enable, output data wires as well as optional input data wires
module mkGCD (GCD);
    Reg#(Bit#(32)) x <- mkReg(0); Reg#(Bit#(32)) y <- mkReg(0);
    Reg#(Bool) busy <- mkReg(False);

    rule gcd;
        if (x >= y) begin x <= x - y; end //subtract
        else if (x != 0) begin x <= y; y <= x; end //swap
    endrule

    method Action start(Bit#(32) a, Bit#(32) b) if (!busy);
        x <= a; y <= b; busy <= True;
    endmethod
    method ActionValue#(Bit#(32)) getResult if (busy && (x==0));
        busy <= False; return y;
    endmethod
endmodule

interface GCD;
    method Action start (Bit#(32) a, Bit#(32) b);
    method ActionValue#(Bit#(32)) getResult;
endinterface
Another GCD implementation twice the throughput

- We can build a GCD module with the same interface but with twice the throughput by putting two gcd modules in parallel
- turnI is used by the start method to direct the input to the gcd whose turn it is and then turnI is flipped
- Similarly, turn0 is used by getResult to get the output from the appropriate gcd, and then turn0 is flipped
module mkMultiGCD (GCD);
  GCD gcd1 <- mkGCD();
  GCD gcd2 <- mkGCD();
  Reg#(Bool) turnI <- mkReg(False);
  Reg#(Bool) turnO <- mkReg(False);
method Action start(Bit#(32) a, Bit#(32) b);
  if (turnI) gcd1.start(a, b); else gcd2.start(a, b);
  turnI <= !turnI;
endmethod
method ActionValue (Bit#(32)) getResult;
  Bit#(32) y;
  if (turnO) y <- gcd1.getResult
  else y <- gcd2.getResult;
  turnO <= !turnO
return y;
endmethod
endmodule

interface GCD;
  method Action start (Bit#(32) a, Bit#(32) b);
  method ActionValue#(Bit#(32)) getResult;
endinterface
Hardware Synthesis
High-level idea

- Every module represents a sequential machine
- Register is a primitive module – its implementation is outside the language
  - Register bit width is derived from its type
- Each Register and module is instantiated explicitly
  - Input/Output wires of a module are derived from its interface, i.e., type
- Rules and methods define the combinational logic to connect registers and modules

The resulting hardware is a collection of sequential machines, which itself behaves like a sequential machine
Register: The Primitive module

interface Reg#(type t);
  method Action _write(t x);
  method t _read;
endinterface

- Implementation is defined outside the language
- A register is created using mkReg or mkRegU
- The guards of _write and _read are always true and not generated
- Special syntax
  - x <= e instead of x._write(e)
  - x instead of x._read in expressions
Module to Sequential Machine synthesis

Defined by the Module Interface declaration

Declared in the module

Defined by the rules and methods of the module

Combinational logic

Clock

Input

Output

en
rady

een
rady

Module M

reg

reg

reg
Synchronous Sequential Machines

- In this subject we will study only single-clock circuits, where all the registers are connected to the same clock.
- Bluespec design has no control over this clock and consequently it never shows up in Bluespec codes.
  - We will not show the clock in our circuit diagrams
- Bluespec design does specify the enable signal for each register implicitly and the Bluespec compiler generates the enable signal and the associated data for each register.
Rules and methods define the combinational logic

module mkEx1 (...);
    Reg#(t) x <- mkRegU;
    method Action f(t a);
        x <= e;
    endmethod
endmodule

module mkEx2 (...);
    Reg#(t) x <- mkRegU;
    method Action f(t a);
        if (b) x <= e;
    endmethod
endmodule

module mkEx3 (...);
    Reg#(t) x <- mkRegU;
    method Action f(t a);
        if (b) x <= e1;
        else x <= e2;
    endmethod
endmodule
Dealing with multiple sources for register assignments

Both \( f \) and \( g \) can’t assign to \( x \) at the same time.

Compiler has to ensure in such a setting that both \( f \) and \( g \) can’t be enabled together.

\[
\text{module mkEx4 (...);} \\
\quad \text{Reg#(t) } x \leftarrow \text{mkRegU;} \\
\quad \text{method Action } f(t \; a); \\
\quad \quad x \leftarrow e1; \\
\quad \text{endmethod} \\
\quad \text{method Action } g(t \; a); \\
\quad \quad x \leftarrow e2; \\
\quad \text{endmethod} \\
\text{endmodule}
\]
A new mux-like structure to deal with multiple sources

- $x_i$ has a meaningful value only if its corresponding $v_i$ is true
- Compiler has to ensure that at most one $v_i$ is true at any given time; the circuit will behave unpredictably if multiple input signals are valid
Synthesis of multiple sources for register assignments

- Compiler has to ensure that both f and g can’t be enabled together

```verilog
module mkEx4 (...);
    Reg#(t) x <= mkRegU;
    method Action f(t a);
        x <= e1;
    endmethod
    method Action g(t a);
        x <= e2;
    endmethod
endmodule
```

- Diagram showing the relationship between variables f.data, f.en, e1, g.data, g.en, e2, and x.
Example
One-Element FIFO Implementation

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

interface Fifo#(numeric type size, type t);
  method Action enq(t x);
  method Action deq;
  method t first;
endinterface
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

- I/O: Interface
- Instantiate state
- Insert muxes
- Compile enq
- Compile deq
- Compile first
Redrawing the FIFO Circuit

A module is a sequential circuit with input and output wires corresponding to its interface methods.
### Next state transition

Partial Truth Table

<table>
<thead>
<tr>
<th>inputs</th>
<th>state</th>
<th>next state</th>
<th>outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>enq. en</td>
<td>enq. data</td>
<td>deq. en</td>
</tr>
<tr>
<td>0</td>
<td>x</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>0</td>
<td>x</td>
<td>0</td>
<td>d</td>
</tr>
<tr>
<td>1</td>
<td>d</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>0</td>
<td>x</td>
<td>1</td>
<td>d</td>
</tr>
</tbody>
</table>

**Illegal inputs**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Tedious!
Constraints on the use of methods of a FIFO

- Bluespec compiler must ensure:
  - enq.en is not set to True unless enq.rdy is True
  - Similarly, deq.en is not set unless deq.rdy is True

Let us illustrate this by putting the FIFO to use
Streaming a function: Circuit

This is a sequential machine too!

Notice that enq.en cannot be True unless enq.rdy is true; deq.en cannot be True unless deq.rdy is true

MIT 6.004 Fall 2018
Redrawing the sequential circuit
Hierarchical sequential circuits
sequential circuits containing modules

Each module represents a sequential machine

Combinational logic (no cycles, no clock)

Register inputs and outputs are replaced by method inputs and outputs
Take-home problem: Draw the circuit for this GCD

module mkGCD (GCD);
    Reg#(Bit#(32)) x <- mkReg(0);
    Reg#(Bit#(32)) y <- mkReg(0);
    Reg#(Bool) busy <- mkReg(False);
rule gcd;
    if (x >= y) begin x <= x - y; end //subtract
    else if (x != 0) begin x <= y; y <= x; end //swap
endrule
method Action start(Bit#(32) a, Bit#(32) b) if (!busy);
    x <= a; y <= b; busy <= True;
endmethod
method ActionValue (Bit#(32)) getResult if (busy && (x==0));
    busy <= False; return y;
endmethod
endmodule

interface GCD;
    method Action start
        (Bit#(32) a, Bit#(32) b);
    method ActionValue(Bit#(32))
        getResult;
endinterface

Assume b != 0