Hardware Synthesis from BSV

Arvind
Computer Science & Artificial Intelligence Lab. Massachusetts Institute of Technology
Modulo-4 counter

Easy to visualize as a circuit but tedious to write

Easy and abstract description but is it a circuit?
Modulo-4 counter in BSV

interface Counter;
    method Action inc;
    method Bit#(2) read;
endinterface

module moduloCounter(Counter);
    Reg#(Bit#(2)) cnt <- mkReg(0);
    method Action inc;
        cnt <= {cnt[1]^cnt[0],~cnt[0]};
    endmethod
    method Bit#(2) read;
        return cnt;
    endmethod
endmodule

Today's lecture is about understanding how circuits are synthesized from BSV descriptions
Register

A group of flip-flops with a common clock and enable

A register in BSV is declared to hold values of a certain type which determines the width of the register
Synchronous Sequential Machines

Combinational logic
(no cycles, no clock)
BSV to Sequential Circuits

- Each Register and its width is declared explicitly.
- All registers are driven by a common clock which is implicit; your program has no control over it.
- Combinational logic is derived from the rules and methods you write.
- Your program defines the input value and the enable for each register.
- Each rule, action method and action value method generates an enable signal for each register it sets directly or indirectly.
Plan

- Ad hoc but intuitive description of BSV-to-circuits using FIFO as an example
- Show that every module is a sequential circuit
- Systematic way of connecting modules using rules and methods to build larger and larger circuits
- High throughput GCD
One-Element FIFO Implementation with guards

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
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>enq. en</td>
<td>enq. data</td>
<td>deq. en</td>
<td>d^t</td>
</tr>
<tr>
<td>0</td>
<td>x</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>x</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>x</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>x</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**Illegal inputs**

March 6, 2018


L08-11
Constraints on the use of methods of a FIFO

- The BSV compiler makes sure that the enq.en is not set to True unless enq.rdy is True.
- Similarly, for deq.en and deq.rdy.
- Your code is such that enq.rdy and deq.rdy also cannot be True simultaneously. Thus, the input for the v register is always well defined.
Streaming a function: Circuit

```
rule stream;
  outQ.enq(f(inQ.first));
  inQ.deq;
endrule
```

This is a sequential machine too!

Notice that enq.en cannot be True unless enq.rdy is true
Module as a sequential circuit

In general:
- A read method has no enable input wire
- An Action method has no output data wires
- An ActionValue method has both ready and enable wires as well as both input and output data wires

We can determine all the input and output wires of a module from its interface definition
A register is a primitive module in BSV and its implementation is defined outside the language.

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

- Special syntax: we write
  - `x <= e` instead of `x._write(e)`
  - `x` instead on `x._read` in expressions
- The guards of `_write` and `_read` are always true
  - The guard wires are not generated for registers
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
Rules and methods *only* define combinational logic

```verilog
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
```
A new structure for dealing with conditionals

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

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

\[ x_i \] has a meaningful value only if its corresponding \[ v_i \] is true

Only one \[ v_i \] should be true at any given time (\textit{one-hot-encoding})
The circuit for GCD was discussed in your recitation.
Streaming the GCD module

We will discuss the hardware synthesis of this in the recitation section.
We can build a GCD module with the same interface but with twice the throughput by putting two gcd modules in parallel.

We can have a variable “turn” which will be used by the start method to direct the input to the gcd whose turn it is. Then we can flip the turn.

getResult can pull the results from the appropriate gcd by looking at the history in the turnQ.
module mkMultiGCD (GCD);
  GCD gcd1 <- mkGCD();
  GCD gcd2 <- mkGCD();
  Reg#(Bool) turn <- mkReg(False);
  Fifo#(2, Bool) turnQ <- mkFifo;
method Action start(Bit#(32) a, Bit#(32) b);
  if (turn) gcd1.start(a,b) else gcd2.start(a,b);
  turnQ.enq(turn); turn <= !turn;
endmethod
method ActionValue (Bit#(32)) getResult;
  Bit#(32) y;
  if (turnQ.first) y <- gcd1.getResult else y <- gcd2.getResult;
  turnQ.deq
return y;
endmethod
endmodule

More discussion of the hardware synthesis of this in the recitation section

Size 2 FIFO

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

High throughput GCD code