Multiple Clock Domains (MCD) *Continued …*

Arvind with Nirav Dave
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

Multiple Clock Domains in Bluespec

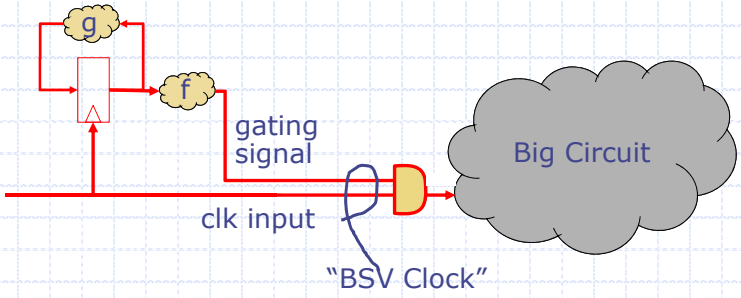
- The *Clock* type, and functions √
- Clock families ←
- Making clocks
- Moving data across clock domain
- Revisit the 802.11a Transmitter

The *Clock* type

- Conceptually, a clock consists of two signals
  - an oscillator
  - a gating signal
- In general, implemented as two wires
- If ungated, oscillator is running
  - Whether the oscillator is running when it is gated off depends on implementation library—tool doesn't care

BSV clock

- Often, big circuits do not do useful work until some boolean condition holds
- We can save power by combining the boolean condition with the clock (i.e. clock gating)



Clock families

- All clocks in a "family" share the same oscillator
  - They differ only in gating
- If c2 is a gated version of c1, we say c1 is an "ancestor" of c2
  - If some clock is running, then so are all its ancestors
- The functions isAncestor(c1,c2) and sameFamily(c1,c2) are provided to test these relationships
  - Can be used to control static elaboration (e.g., to optionally insert or omit a synchronizer)

Clock family discipline

- All the methods invoked by a rule (or by another method) must be clocked by clocks from one family
  - The tool enforces this
  - The rule will fire only when the clocks of all the called methods are ready (their gates are true)
- Two different clock families can interact with each other only though some clock synchronizing state element, e.g., FIFO, register

Clocks and implicit conditions

- Each action is implicitly guarded by its clock's gate; this will be reflected in the guards of rules and methods using that action
  - So, if the clock is off, the method is unready
  - So, a rule can execute only if all the methods it uses have their clocks gated on
- This doesn't happen for value methods
  - So, they stay ready if they were ready when the clock was switched off

Clocks and implicit conditions

- Example:

```
FIFO #(Int #(3))  f  <-  mkFIFO (clocked_by c);
```

- If c is switched off:
  - f.enq, f.deq and f.clear are unready
  - f.first remains ready if the fifo was non-empty when the clock was switched off

The clocks of methods and rules

- Every method, and every rule, has a notional clock
- For methods of primitive modules (Verilog wrapped in BSV):
  - Their clocks are specified in the BSV wrappers which import them
- For methods of modules written in BSV:
  - A method's clock is a clock from the same family as the clocks of all the methods that it, in turn, invokes
  - The clock is gated on if the clocks of all invoked methods are gated on
  - If necessary, this is a new clock
- The notional clock for a rule may be calculated in the same way

Multiple Clock Domains in Bluespec

- The *Clock* type, and functions √
- Clock families √
- Making clocks ←
- Moving data across clock domain
- Revisit the 802.11a Transmitter

Making gated clocks

```
Bool  b  =  …  ;
Clock  c0 <- mkGatedClock (b);
```

- c0 is a version of the current clock, gated by b
  - c0's gate is the gate of the current clock AND'ed with b
- The current clock is an ancestor of c0

Making gated clocks

```
Bool  b  =  …  ;
Clock  c0 <- mkGatedClock (b);

Bool b1 = …;
Clock c1 <- mkGatedClock (b1, clocked_by c0);
```

- c1 is a version of c0, gated by b1
  - and is also a version of the current clock, gated by (b && b1)
- current clock, c0 and c1 all same family
- current clock and c0 both ancestors of c1

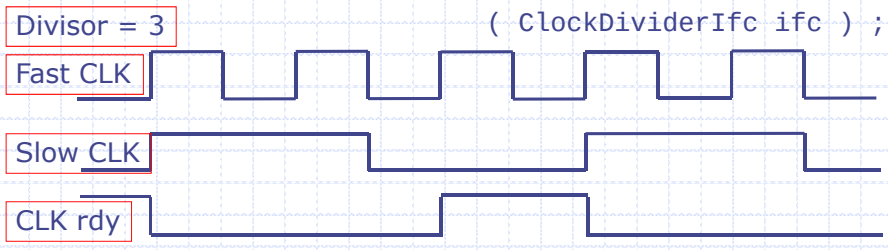
More Clock constructors

- mkGatedClock
  - (Bool newCond)
- mkAbsoluteClock
  - (Integer start, Integer period);
- mkClockDivider
  - #(Integer divider) (ClockDividerIfc clks)

Clock Dividers

```
interface ClockDividerIfc ;
    interface Clock   fastClock ;  // original clock
    interface Clock   slowClock ;  // derived clock
    method    Bool    clockReady ;
endinterface

module mkClockDivider #( Integer divisor )
                       ( ClockDividerIfc ifc ) ;
```



Clock Dividers

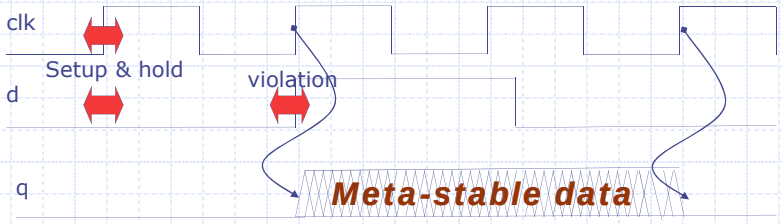
- No need for special synchronizing logic
- The *clockReady* signal can become part of the implicit condition when needed

Multiple Clock Domains in Bluespec

- The *Clock* type, and functions √
- Clock families √
- Making clocks √
- Moving data across clock domains ←
- Revisit the 802.11a Transmitter

Moving Data Across Clock Domains

- Data moved across clock domains appears asynchronous to the receiving (destination) domain
- Asynchronous data will cause meta-stability
- The only safe way: use a *synchronizer*

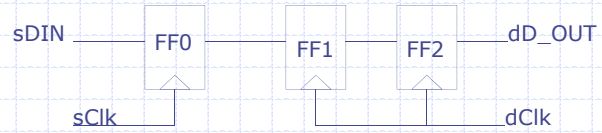


Synchronizers

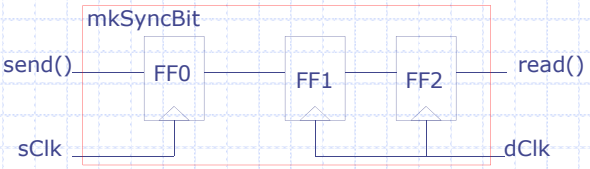
- Good synchronizer design and use reduces the probability of observing meta-stable data
- Bluespec delivers conservative (speed independent) synchronizers
- User can define and use new synchronizers
- Bluespec does not allow unsynchronized crossings (compiler static checking error)

2 - Flop BIT-Synchronizer

- Most common type of (bit) synchronizer
- FF1 will go meta-stable, but FF2 does not look at data until a clock period later, giving FF1 time to stabilize
- Limitations:
  - When moving from fast to slow clocks data may be overrun
  - Cannot synchronize words since bits may not be seen at same time



Bluespec's 2-Flop Bit-Synchronizer




```
interface SyncBitIfc ;
   method Action  send ( Bit#(1) bitData ) ;
   method Bit#(1) read () ;
endinterface
```

- The designer must follow the synchronizer design guidelines:
  - No logic between FF0 and FF1
  - No access to FF1's output

Use example: MCD Counter

- Up/down counter: Increments when up_down_bit is one; the up_down_bit is set from a different clock domain.
- Registers:

```
Reg# (Bit#(1)) up_down_bit <-
                mkReg(0, clocked_by ( writeClk ) );
Reg# (Bit# (32)) cntr <- mkReg(0); // Default Clk
```

- The Rule (attempt 1):

```
rule countup ( up_down_bit == 1 ) ;
     cntr <= cntr + 1;
endrule
```

Illegal Clock Domain Crossing

Adding the Synchronizer

```
SyncBitIfc  sync <- mkSyncBit( writeClk,
                    writeRst, currentClk ) ;
```

Split the rule into two rules where each rule operates in one clock domain

clocked by *writeClk*

```
rule transfer ( True ) ;
   sync.send ( up_down_bit );
endrule
```

clocked by *currentClk*

```
rule countup ( sync.read == 1 ) ;
   cntr <= cntr + 1;
endrule
```

MCD Counter

```
module mkTopLevel#(Clock writeClk, Reset writeRst)
                  (Top ifc);
Reg# (Bit# (1))  up_down_bit <- mkReg(0,
                                clocked_by(writeClk),
                                reset_by(writeRst)) ;
Reg# (Bit# (32)) cntr <- mkReg (0) ;
                                  // Default Clocking
Clock currentClk <- exposeCurrentClock ;
SyncBitIfc  sync <- mkSyncBit ( writeClk, writeRst,
                                currentClk ) ;
rule transfer ( True ) ;
   sync.send( up_down_bit );
endrule
rule countup ( sync.read == 1 ) ;
   cntr <= cntr + 1;
endrule
```

We won't worry about resets for the rest of this lecture

Different Synchronizers

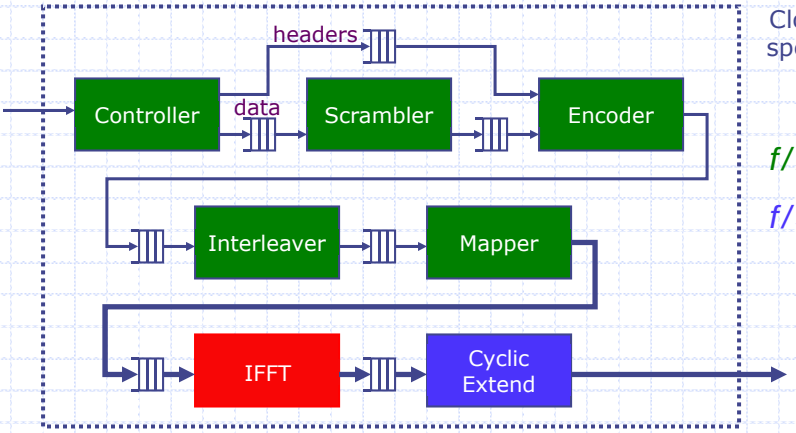
- Bit Synchronizer
- FIFO Synchronizer
- Pulse Synchronizer
- Word Synchronizer
- Asynchronous RAM
- Null Synchronizer
- Reset Synchronizers

Documented in Reference Guide

Multiple Clock Domains in Bluespec

- The *Clock* type, and functions √
- Clock families √
- Making clocks √
- Moving data across clock domains √
- Revisit the 802.11a Transmitter ←

802.11 Transmitter Overview



The Transmitter without MCD

```
module mkTransmitter(Transmitter#(24,81));

  let controller   <- mkController();
  let scrambler    <- mkScrambler_48();
  let conv_encoder <- mkConvEncoder_24_48();
  let interleaver  <- mkInterleaver();
  let mapper       <- mkMapper_48_64();
  let ifft         <- mkIFFT_Pipe();
  let cyc_extender <- mkCyclicExtender();

  // rules to stitch these modules together
```

How should we
1. Generate different clocks?
2. Pass them to modules?
3. Introduce clock synchronizers and fix the rules?

Step 1: Introduce Clocks

```
module mkTransmitter(Transmitter#(24,81));

  let clockdiv13 <- mkClockDivider(13);
  let clockdiv52 <- mkClockDivider(52);
  let clk13      = clockdiv13.slowClock;
  let clk52      = clockdiv52.slowClock;

  let controller   <- mkController();
  let scrambler    <- mkScrambler_48();
  let conv_encoder <- mkConvEncoder_24_48();
  let interleaver  <- mkInterleaver();
  let mapper       <- mkMapper_48_64();
  let ifft         <- mkIFFT_Pipe();
  let cyc_extender <- mkCyclicExtender();

  // rules to stitch these modules together
```

Step 2: Pass the Clocks

```
module mkTransmitter(Transmitter#(24,81));
  let clockdiv13 <- mkClockDivider(13);
  let clockdiv52 <- mkClockDivider(52);
  let clk13      = clockdiv13.slowClock;
  let clk52      = clockdiv52.slowClock;

  let controller   <- mkController(clocked_by clk13);
  let scrambler    <- mkScrambler_48(clocked_by clk13);
  let conv_encoder <- mkConvEncoder_24_48(clocked_by clk13);
  let interleaver  <- mkInterleaver(clocked_by clk13);
  let mapper       <- mkMapper_48_64(clocked_by clk13);
  let ifft         <- mkIFFT_Pipe();
  let cyc_extender <- mkCyclicExtender(clocked_by clk52);
  // rules to stitch these modules together
```

Default Clock

Now some of the stitch rules have become illegal because they call methods from different clock families

Introduce Clock Synchronizers

Step 3: Introduce Clock Synchronizers

```
module mkTransmitter(Transmitter#(24,81));
  let m2ifftFF  <- mkSyncFIFOToFast(2,clockdiv13);
  let ifft2ceSF <- mkSyncFIFOToSlow(2,clockdiv52);
  …
  let mapper       <- mkMapper_48_64(clocked_by clk13);
  let ifft         <- mkIFFT_Pipe();
  let cyc_extender <- mkCyclicExtender(clocked_by clk52);
  rule mapper2fifo(True); //split mapper2ifft rule
       stitch(mapper.toIFFT, m2ifftFF.enq);
  endrule
  rule fifo2ifft(True);
       stitch(pop(m2ifftFF), ifft.fromMapper);
  endrule
  rule ifft2fifo(True); //split ifft2ce rule
       stitch(ifft.toCycExtend, ifft2ceFF.enq);
  endrule
  rule fifo2ce(True);
       stitch(pop(ifft2ceFF), cyc_extender.fromIFFT);
  endrule
```

The Transmitter (after)

```
module mkTransmitter(Transmitter#(24,81));

  let clockdiv13 <- mkClockDivider(13);
  let clockdiv52 <- mkClockDivider(52);
  let clk13th = clockdiv13.slowClock;
  let clk52nd = clockdiv52.slowClock;

  let m2ifftFF  <- mkSyncFIFOToFast(2,clockdiv13);
  let ifft2ceSF <- mkSyncFIFOToSlow(2,clockdiv52);

  let controller   <- mkController(clocked_by clk13th);
  let scrambler    <- mkScrambler_48(… " …);
  let conv_encoder <- mkConvEncoder_24_48 (… " …);
  let interleaver  <- mkInterleaver (… " …);
  let mapper       <- mkMapper_48_64 (… " …);
  let ifft         <- mkIFFT_Pipe();
  let cyc_extender <- mkCyclicExtender(clocked_by clk52nd, …);

  rule controller2scrambler(True);
      stitch(controller.getData, scrambler.fromControl);
  endrule
  rule mapper2fifo(True);
       stitch(mapper.toIFFT, m2ifftFF.enq);
  Endrule
```

Synchronizers for clock domain crossing?

Did not work...

stoy@forte:~/examples/80211$ bsc -u -verilog Transmitter.bsv

Error: "./Interfaces.bi", line 62, column 15: (G0045)
Method getFromMAC is unusable because it is connected to a clock not available at the module boundary.

Need to fix the Transmitter's interface so that the outside world knows about the clocks that the interface methods operate on.

(These clocks were defined inside the module)

The Fix – pass the clocks out

```
interface Transmitter#(type inN, type out);
   method Action getFromMAC(TXMAC2ControllerInfo x);
   method Action getDataFromMAC(Data#(inN) x);

   method ActionValue#(MsgComplexFVec#(out))
                          toAnalogTX();

   interface Clock  clkMAC;
   interface Clock  clkAnalog;
endinterface
```


Summary

- The *Clock* type, and type checking ensures that all circuits are clocked by actual clocks
- BSV provides ways to create, derive and manipulate clocks, safely
- BSV clocks are *gated*, and gating fits into Rule-enabling semantics (clock guards)
- BSV provides a full set of speed-independent data synchronizers, already tested and verified
  - The user can define new synchronizers
- BSV precludes unsynchronized domain crossings