Multiple Clock Domains

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

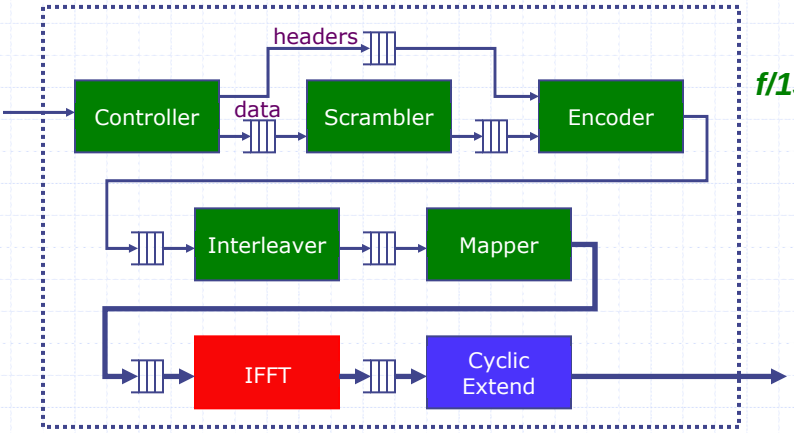
Based on material prepared by Bluespec Inc

802.11 Transmitter Overview

Clock frequencies: *f/52*

f

f/13



The relative clock frequency of each block is based on its internal architecture and the overall performance requirement

Synthesis results for different microachitectures

Design	Area (mm^2)	CLK Period	Throughput (1 symbol)	Latency
Comb.	1.03	15 ns	15ns	15 ns
Pipelined	1.46	7 ns	7 ns	21 ns
Folded	0.83	8 ns	24 ns	24 ns
S Folded 1 Radix	0.23	8 ns	408 ns	408 ns

Nirav Dave
Mike Pellauer
Man C Ng

Will have to run ~20 times faster for the same throughput

TSMC .13 micron; numbers reported are before place and route.

Single radix-4 node design is ¼ the size of combination design but still meets the throughput requirement easily; clock can reduced to 15 to 20 Mhz

BSV point of view

- Automate the simplest things
- Make it easy to do simple things
- Make it safe to do the more complicated things

The simplest case

- Only one clock
- Need never be mentioned in BSV source
  - (Note: hasn't been mentioned in any examples so far!)
- Synthesized modules have an input port called CLK
- This is passed to all interior instantiated modules

Multiple Clock Domains in Bluespec

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

The *Clock* type

- Clock is an ordinary first-class type
- May be passed as parameter, returned as result of function, etc.
- Can make arrays of them, etc.
- Can test whether two clocks are equal

```
Clock c1, c2;

Clock c = (b ? c1 : c2);     // b must be known at
                                compile time
```


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

Instantiating modules with non-default clocks

- Example: instantiating a register with explicit clock

```
Clock c =  …  ;
Reg# (Bool)  b <-  mkReg  (True, clocked_by c);
```

- Modules can also take clocks as ordinary arguments, to be fed to interior module instantiations

The clockOf() function

- May be applied to any BSV expression, and returns a value of type Clock
- If the expression is a constant, the result is the special value noClock
- The result is always well-defined
  - Expressions for which it would not be well-defined are illegal

The clockOf() function

- Example

```
Reg# (UInt# (17)) x <- mkReg (0, clocked_by c);
let y = x + 2;
Clock c1 = clockOf (x);
Clock c2 = clockOf (y);
```

- c, c1 and c2 are all equal
- They may be used interchangeably for all purposes

A special clock

- Each module has a special "default" clock
- The default clock will be passed to any interior module instantiations (unless otherwise specified)
- It can be exposed in any module as follows:

```
Clock c <- exposeCurrentClock;
```

Multiple Clock Domains in Bluespec

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

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
- There is no need for special domain-crossing logic when the clocks involved are from the same family
  - It's all handled by implicit conditions

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

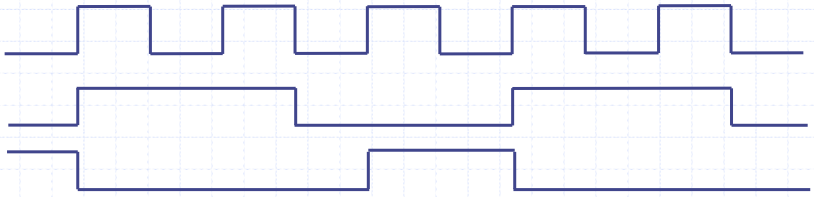
More Clock constructors

- 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

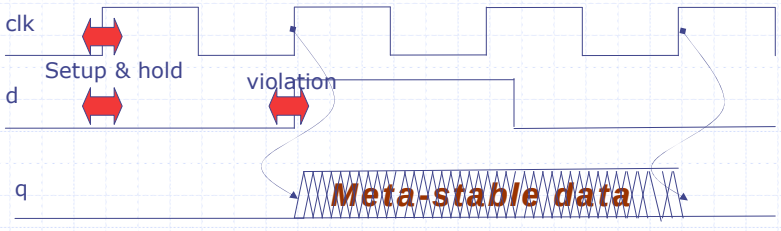
- No need for special synchronizing logic
- The *clockReady* signal can become part of the implicit condition when needed

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*

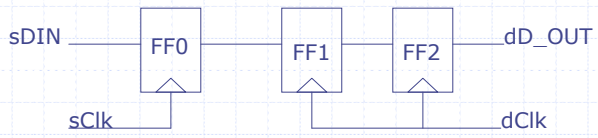


Synchronizers

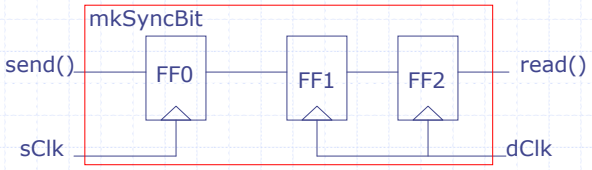
- Good synchronizer design and use reduces the probability of observing meta-stable data
- Bluespec delivers conservative (speed independent) synchronizers
- User can define and use new synchronizers
- Bluespec does not allow unsynchronized crossings (compiler static checking error)

2 - Flop 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 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

Small Example

- Up/down counter, where direction signal comes from separate domain.
- Registers:

```
Reg# (Bit#(1)) up_down_bit <-
                mkReg(0, clocked_by ( readClk ) );

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( readClk,
                               readRst, currentClk ) ;
```

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

clocked by *readClk*

clocked by *currentClk*

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

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


Full Example

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

Other Synchronizers

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

- Documented in Reference Guide

Multiple Clock Domains in Bluespec

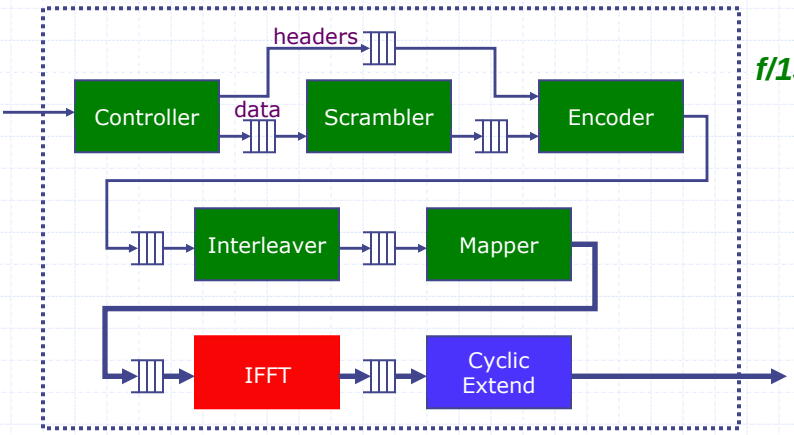
- The *Clock* type, and functions √
- Clock families √
- Making clocks √
- Moving data across clock domains √
- Revisit the 802.11a Transmitter ←

802.11 Transmitter Overview

Clock frequencies: *f/52*

f

f/13



The Transmitter

```
module mkTransmitter(Transmitter#(24,81));
  function Action stitch(ActionValue#(a) x,
                         function Action f(a v));
    action  let v <- x;  f(v);   endaction
  endfunction
  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();
  rule controller2scrambler(True);
      stitch(controller.getData,scrambler.fromControl);
  endrule
  ... more rules ...
```

What is the clock domain ?

The Transmitter

```
module mkTransmitter(Transmitter#(24,81));
  let clockdiv13 <- mkClockDivider(13);
  let clockdiv52 <- mkClockDivider(52);
  let clk13th    = clockdiv13.slowClock;
  let clk52nd    = clockdiv52.slowClock;
  let reset13th <- mkAsyncResetFromCC(0, clk13th);
  let reset52nd <- mkAsyncResetFromCC(0, clk52nd);

  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();
  rule controller2scrambler(True);
      stitch(controller.getData,scrambler.fromControl);
  endrule
```

How should we

1. Generate these clocks?
2. Pass them to modules

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 reset13th <- mkAsyncResetFromCC(0, clk13th);
  let reset52nd <- mkAsyncResetFromCC(0, clk52nd);

  let controller   <- mkController(clocked_by clk13th,
                                   reset_by reset13th);
  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
…more rules…
```

What about rules involving clock domain crossing?

Clock Domain Crossing

```
rule mapper2ifft(True);
     stitch(mapper.toIFFT, ifft.fromMapper);
endrule
```

???

```
 rule mapper2ifft(True);
     let x <- mapper.toIFFT();
     ifft.fromMapper(x)
endrule
```

Different methods in an action are on different clocks – not legal without synchronizers

Clock Domain Crossing

```
rule mapper2ifft(True);
      let x <- mapper.toIFFT();
      ifft.fromMapper(x)
endrule
```

split

```
let m2ifftFF <-
  mkSyncFIFOToFast(2,clockdiv13,reset13th);

rule mapper2fifo(True);
       stitch(mapper.toIFFT, m2ifftFF.enq);
Endrule

rule fifo2ifft(True);
       stitch(pop(m2ifftFF), ifft.fromMapper);
endrule
```

Similarly for IFFT to CyclicExt

```
let ifft2ceFF <-
  mkSyncFIFOToSlow(2,clockdiv52,reset52nd);

rule ifft2ff(True);
     stitch(ifft.toCyclicExtender, ifft2ceFF.enq);
endrule

rule ff2cyclicExtender(True);
     stitch(pop(ifft2ceFF),
  cyc_extender.fromIFFT);
endrule
```


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.

The method's clock is internal!

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
- 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