Lecture 10: Multiple Clock Domains

- The Clock type, and functions
- Modules with different clocks
- Clock families
- Making clocks
- Moving data across clock domains
- Synchronizing interfaces
BSV point of view

- Automate the simplest things
- Make it easy to do simple things
- Make it safe to do the more complicated things
- Work with gated clocks, to enable power management

The simplest case

- Only one clock, one reset
- 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 and RST_N
- This is passed to all interior instantiated modules
The Clock and Reset types

- Clock and Reset are ordinary first-class types
- May be passed as parameter, returned as result of function, etc.
- Can make arrays of them, can test whether two clocks are equal, etc.

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

Default Clock and Reset

- Each module has a special “default” clock and reset
- The default clock and reset will be passed to any interior module instantiations (unless otherwise specified)
- They can be exposed by using the following modules (defined in the Standard Prelude)

```
module exposeCurrentClock(Clock);
module exposeCurrentReset(Reset);
```

- Example:

```
Clock clk <- exposeCurrentClock;
Reset rst <- exposeCurrentReset;
```
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 reverse is true depends on implementation library—tool doesn’t care

Some Clock constructors

- mkClock#(t initVal, Bool initGate)(MakeClockIfc#(t));
- mkGatedClock#(Bool initGate)(GatedClockIfc);
- mkAbsoluteClock#(Integer start, Integer period)(Clock);
- mkClockDivider#(Integer divisor) (ClockDividerIfc);
Making clocks with mkClock

```plaintext
Bool gateClk = ...; // gate condition
Reg#(Bit#(1)) osc <- mkReg(0);
MakeClockIfc#(Bit#(1)) mc <- mkClock(False,True);
Clock newClk = mc.new_clk;

rule oscillate;
  let new_osc = invert(osc);
  mc.setClockValue(new_osc);
  mc.setGateCond(gateClk);
  osc <= new_osc;
endrule
```

- `mkClock` is a primitive for generating a clock with an arbitrary waveform controlled from within the design.
- `newClk` is a clock running at half the speed of the current clock, and is gated by `gateClk`.

Making clocks with mkGatedClock

```plaintext
Clock clk <- exposeCurrentClock;
GatedClockIfc gc1 <- mkGatedClock(True);
Clock clk1 = gc1.new_clk;
GatedClockIfc gc2 <- mkGatedClock(True, clocked_by clk1);
Clock clk2 = gc2.new_clk;

rule gate_clocks;
  gc1.setGateCond(gateClk1);
  gc2.setGateCond(gateClk2);
endrule
```

- `clk1` is a version of `clk`, gated by `gateClk1`.
- `clk2` is a version of `clk1`, gated by `gateClk2`.
  - i.e. it is as the current clock gated by `(gateClk1 && gateClk2)`
- `clk`, `clk1` and `clk2` are from the same family
- `clk` and `clk1` are ancestors of `clk2`
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)

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

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

c1 and c2 are equal
Some Reset constructors

- \texttt{mkReset#(Integer stages, Bool startInRst)(Clock dClkIn, MakeResetIfc)};
- \texttt{mkResetSync#(Integer stages, Bool startInRst)(Clock dClkIn, MakeResetIfc)};
- \texttt{mkAsyncReset#(Integer stages)(Reset sRst, Clock dClkIn, Reset)};
- \texttt{mkSyncReset#(Integer stages)(Reset sRst, Clock dClkIn, Reset)};

See details in $\$BLUESPECDIR/..../doc/reference_guide.doc$

Instantiating modules with non-default clocks and resets

- Example: instantiating a register with explicit clock

  \begin{verbatim}
  Clock clk  <- exposeCurrentClock;  // Current Clock
  Reset rst  <- exposeCurrentReset;  // Current Reset
  Clock clk1 <- ... ;                // 2nd clock domain
  Reset rst1 <- ... ;               // 2nd reset domain

  Reg#(Bool) a <- mkRegA(False);
  Reg#(Bool) b <- mkReg(True, clocked_by clk1, reset_by rst1);
  \end{verbatim}

- Modules can also take clocks as ordinary dynamic arguments, to be fed to interior module instantiations
  - (Examples later)
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

- The gating condition of an Action or ActionValue method’s clock becomes one of the method’s implicit conditions
  - 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:

```hs
FIFO#(Int#(3)) myFifo <= mkFIFO(clocked_by clk1);
```

- If clk1 is switched off:
  - myFifo.enq, myFifo.deq and f.clear are unready
  - myFifo.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
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
- Synchronizers needed for all crossings

- Bluespec delivers conservative (speed independent) synchronizers
- User can define and use new synchronizers
- Bluespec does not allow unsynchronized crossings (compiler static checking error)
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

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

Module mkBitSync#(Clock sClk, Reset sRst, Clock dClk) (SyncBitIfc);
```

Synchronizer design guidelines cannot be violated:
- 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:

```verilog
Reg# (Bit# (32)) cntr <- mkReg(0); // Default Clk
Reg# (Bit#(1)) up_down_bit <- mkReg(0, clocked_by clk2, reset_by rst2);
```

- The Rule (attempt 1):

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

---

Adding the Synchronizer

```verilog
module mkTopLevel#(Clock clk2, Reset rst2) (Empty);
  Reg# (Bit# (1)) up_down_bit <- mkReg(0, clocked_by clk2, reset_by rst2);
  Reg# (Bit# (32)) cntr <- mkReg (0 ) ; // Default Clock, default Reset
  Clock currentClk <- exposeCurrentClock ; // Default clock
  SyncBitIfc#(Bit#(1)) sync <- mkSyncBit(clk2, rst2, currentClk); // Bit synchronizer

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

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

- Synchronizes single clock width pulses
- Two "send"s may not be seen if they occur too close together (faster than 2*dstClk)
- Also versions from CurrentClock and to CurrentClock

```
module mkSyncPulse#(Clock sClkIn,
                Reset sRstIn,
                Clock dClkIn)(SyncPulseIfc);
```

interface SyncPulseIfc

```
method Bool   pulse;
method Action send;
```

Handshake Pulse Synchronizer

- A Pulse Synchronizer with a handshake protocol
- The send method is ready after the pulse is received and an acknowledge returned.
- Latency:
  - send to read is (2*dstClk)
  - send to next send (2*dstClks + 2*srcClk)
- Also versions from CurrentClock and to CurrentClock

```
module mkSyncHandshake#(Clock sClkIn,
                        Reset sRstIn,
                        Clock dClkIn)(SyncPulseIfc);
```
Register Synchronizer

- Uses common Reg#(a) interface
- However, write method has (implicit) ready condition, to allow time for data to be received
- No guarantee that destination reads the data, only that it arrives
- Also versions from CurrentClock and to CurrentClock

```verilog
module mkSyncReg#(t initialValue,
    Clock sClkIn,
    Reset sRstIn,
    Clock dClkIn)(Reg#(t));
```

Example

```verilog
interface Reader ;
    method Bit#(32) wordOut () ;
    method Bool pulseOut() ; // Pulses at new data
endinterface
interface Cruncher ;
    method Action crunch(Bit#(32) dataRead) ;
endinterface

module top#(Clock clk2, Reset rst2)(Top);
    Reader reader <- mkWordReader(clocked_by clk2, reset_by rst2) ;
    Cruncher cruncher <- mkWordCrunch;
    Clock clk <- exposeCurrentClock;
    Reg#(Bit# (32)) wordSync <- mkSyncReg(0, clk2, rst2, clk) ;

    rule loadSync( reader.pulseOut ); // it runs in the clk2, rst2 domain
        wordSync <= reader.wordOut;
    endrule
    ...
    cruncher.crunch( sync ); // it runs in the clk domain
```
FIFO Synchronizer

- Good for data buffering across clock domains
- Also versions from CurrentClock, to CurrentClock, etc.

```verilog
module mkSyncFIFO(Integer depth,
Clock sClkIn,
Reset sRstIn,
Clock dClkIn)(SyncFIFOIfc#(t));

interface SyncFIFOIfc#(type t);
method Action enq(t a);
method t first;
endinterface
```

Example

```verilog
interface Reader ;
method Bit#(32) wordOut () ;
method Bool pulseOut() ; // Pulses at new data
endinterface

interface Cruncher ;
method Action crunch(Bit#(32) dataRead) ;
endinterface

Module top#(Clock clk2, Reset rst2)(Top);
Reader reader <- mkWordReader(clocked_by clk2, reset_by rst2) ;
Cruncher cruncher <- mkWordCrunch;

Clock clk <- exposeCurrentClock;
SyncFIFOIfc#(Bit# (32)) fifoSync <- mkSyncFIFO(4, clk2, rst2, clk) ;

rule loadSync( reader.pulseOut );  // it runs in the clk2, rst2 domain
  fifoSync.enq( reader.wordOut);
endrule

... cruncher.crunch(fifoSync.first);  // it runs in the clk domain
```
Null Synchronizer

- Needed when no synchronization is desired
- For example, up/down counter

```verilog
module mkNullCrossingWire#(Clock dClkIn, t dataIn)(SyncFIFOIfc#(t));
    Reg# (Bit# (1))  up_down_bit <- mkReg(0) ;
    Reg# (Bit# (32)) cntr <- mkReg (0);
    endinterface

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

Advantages of Bluespec’s synchronizers

- Bluespec provides a full set of speed-independent data synchronizers, with strong interface semantics preventing their misuse
- But good design practices are still needed
  - Identify different clock domains early in design
  - Partition design such that there is one clock per module
  - Keep the synchronizations to the interfaces
  - Limit number of signals which cross clock domains
Specialized Synchronizers

- (Topic for Advanced Training)
- Above synchronizers work for any clock relations
- If there are known relations between clock edges, then synchronizer can be removed
- Bluespec provides some synchronizers for clocks with coincident edges, e.g., divided clocks.
- Users can create their own synchronizers or import their current ones into Bluespec

Synchronizing Interfaces

- The previous examples show a “hardware approach” to clock-domain crossing
  - Designers instantiate a module having different clocks for different methods, and connect it up explicitly
- BSV also has an alternative “linguistic approach”, to convert an interface from one clock domain to another
  - mkConverter converts an existing interface to another one of the same type, but on different clock
  - In this style, more of the details are implicit
Synchronizing Interfaces

```haskell
typedef Server#(QueryType, RespType) IfcType;
IfcType oldIfc <- mkServer (clocked_by c0);
IfcType newIfc <- mkConverter(4, oldIfc);
```

- **oldIfc** is clocked by **c0**
- **newIfc** is clocked by the current clock
- “4” is a parameter for the conversion (FIFO depth)
- **newIfc** can be used exactly as **oldIfc**, but in the current clock domain

**Synchronizing Interfaces**

- Convertible interfaces form a typeclass `ClockConv`
  - Other instances can be added by the user. Need only to define `mkConverter`, in terms of existing primitives

- Not all interfaces are convertible. The following are defined to be convertible in the BSV library:
  - Get and Put interfaces
  - Client and Server interfaces
  - Tuples of convertible interfaces
Summary

- The *Clock* type, and strong 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
- Roadmap: tool will generate SDC constraints identifying all clocks, clocks associated with signals, false paths at clock-domain crossings, etc.

End of Lecture