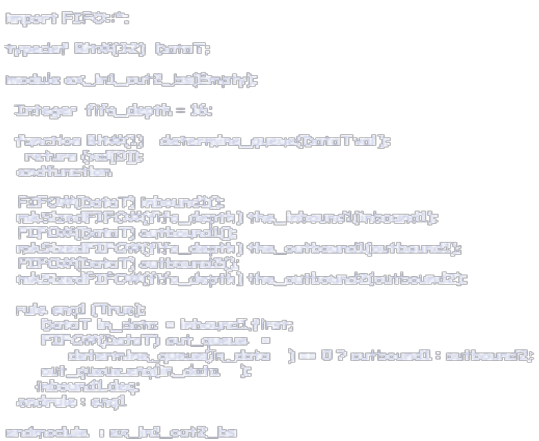
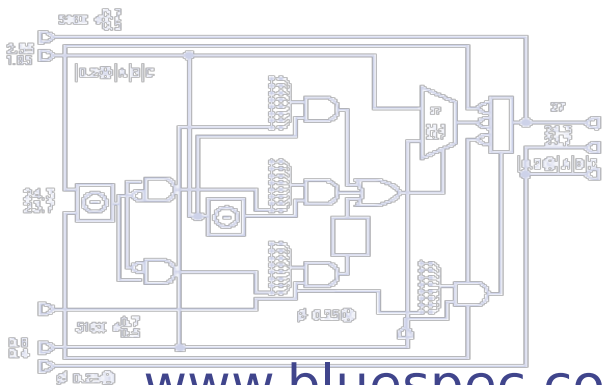
Note: we have not yet created a Bluespec Classic version of these slides

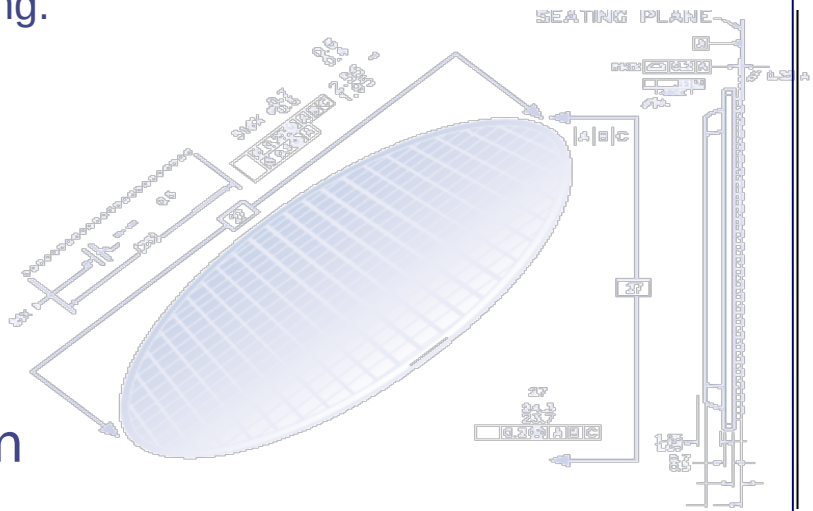
BSV Training

Lec_Multiple_Clock_Domains (MCD)

Clocks, the “Clock” type, clock plumbing (passing clocks down the module hierarchy to modules and primitives), clock domains and clock discipline, synchronizers for clock domain-crossing. Resets, the “Reset” type, Reset plumbing.







www.bluespec.com

Introduction

- Most hardware systems have components driven by multiple clocks, for various engineering reasons (performance, power, standards, silicon technology, legacy, etc.).
- In modern systems, clock signals are special. Clock signals are not used as ordinary signals, and vice versa.
- In modern systems, clocks are often *gated*, so that they can be switched off under certain conditions to avoid power consumption by components.
- The set of components sharing a common clock is known as a *clock domain*. Clock domains are traditionally disjoint.
- The organization of a system into clock domains may not be the same as its organization into modules, i.e., a module may involve multiple clocks.
- Communications between clock domains must usually go through specially engineered circuits called *synchronizers*, to avoid “metastability” problems.

Topics

In this lecture, we describe:

- The “Clock” type (different from all other types), clock gating, and clock “families” (that differ only in gating condition)
- Clock “plumbing”: how clocks are passed down the module hierarchy and to primitive modules (including, for example, registers)
- Clock domains in BSV
- Synchronizers (communicating between clock domains)

(we will also briefly discuss Resets, at the end)

The Clock type, gated clocks, and clock families

The Clock type

- There is a primitive type “Clock” in BSV
- Because of BSV’s strong type checking, entities of type Clock can never be accidentally used as ordinary values, or vice versa
- The full power of BSV’s static elaboration is available to manipulate clocks:
  - Can write expressions with clocks
  - Can write functions on clocks
  - Can created data structures (e.g., arrays) of clocks
  - etc.

  These are all statically resolved (no dynamic clock selection)

```
Clock c1, c2;

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

BSV supports gated clocks

- Gated clocks are common in modern designs. By gating a clock “off”, it stops switching activity in a circuit, and therefore reduces power consumption.

- In BSV, a gated clock consists of two signals
  - an oscillator
  - a gating signal

  implemented (in the generated Verilog) with two wires

- If the gate is True, oscillator is running
  - If the gate is False, oscillator may or may not be running, depends on implementation library—tool doesn’t care

- Clock gate conditions integrate smoothly into rule semantics, i.e., clock gate signals contribute to method and rule conditions (more about this later)

Clock gating is optional

- By default, a Clock input to a module is ungated (and has a single wire)
- By using certain attributes at module synthesis boundaries, one can override this default and request either all input clocks to be gated, or selected input clocks to be gated (gated clocks have two wires)

All these modules have two input clocks

```
(* synthesize *)
module mkMod1#(Clock clk2) (Ifc);
   …
endmodule
```

Default clock: *ungated* clk2: *ungated*

```
(* synthesize, gate_all_clocks *)
module mkMod2#(Clock clk2) (Ifc);
   …
endmodule
```

Default clock: *gated* clk2: *gated*

```
(* synthesize, gate_input_clock = “default_clock” *)
module mkMod3#(Clock clk2) (Ifc);
   …
endmodule
```

Default clock: *gated* clk2: *ungated*

```
(* synthesize, gate_input_clock = “clk2” *)
module mkMod3#(Clock clk2) (Ifc);
   …
endmodule
```

Default clock: *ungated* clk2: *gated*

Clock families

- Clocks that share the same oscillator, and differ only in gating, are termed a "clock family".
  - The *bsc* tool keeps track of clock families, to avoid unnecessary synchronization (clocks in the same family are in the same clock domain, since their edges will never be so close as to cause metastability problems)

- 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, i.e., gating only increases down the ancestor relationship

- 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)

Making gated clocks with mkGatedClock

A gated clock can be created from an existing clock (gated or ungated) using the following module:

```
module mkGatedClock #(Bool v) /* Clock clk_in */ (GatedClockIfc ifc);
```

The boolean parameter is the state on reset (True = gate on, False = gate off). The clk_in parameter is the clock to be gated. It provides the following interface:

```
interface GatedClockIfc;
   method Action setGateCond(Bool gate);
   method Bool getGateCond();
   interface Clock new_clk;
endinterface
```

The new, gated clock is new_clk. It's gating condition can be changed using setGateCond. The current gating condition can be read with getGateCond.

(These facilities are described in the Reference Guide, Sec C.9.1)

Example: Making gated clocks with mkGatedClock

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

rule gate_clocks;
   Bool gateClk1 = …; Bool gateClk2 = …;
   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 the current clock gated by (gateClk1 && gateClk2)
- clk, clk1 and clk2 are from the same family
- clk1 is an ancestor of clk2; clk and clk1 are ancestors of clk2

Clock “plumbing”:
feeding clocks to modules in the module hierarchy, all the way down to primitive modules such as registers, FIFOs and memories

Default clocks and default plumbing

- Every BSV module has an implicit default input clock (being implicit, it is not mentioned in the interface or in the module header)
- When a module m1 instantiates a module m2, by default the m2 instance inherits m1's default clock as its own default clock
  - This structure is followed recursively down the module hierarchy, all the way down to primitive modules (such as registers, FIFOs and memories)

This explains why, in a single-clock-domain BSV design, one never sees any mention of a clock at all—there is a single, default clock that covers everything in the design

Explicit clocks and explicit plumbing

- Extra (non-default) clock inputs to a module can be declared as parameters of type “Clock”
- When such a module is instantiated, the extra clocks are passed as parameters just like any other parameter

- During module instantiation, the default clock for the new instance can be specified explicitly using the optional “clocked_by” attribute (otherwise the instance’s default clock is the same as the parent’s default clock)

- Inside a module instance, its default clock can be named using the primitive module “exposeCurrentClock”

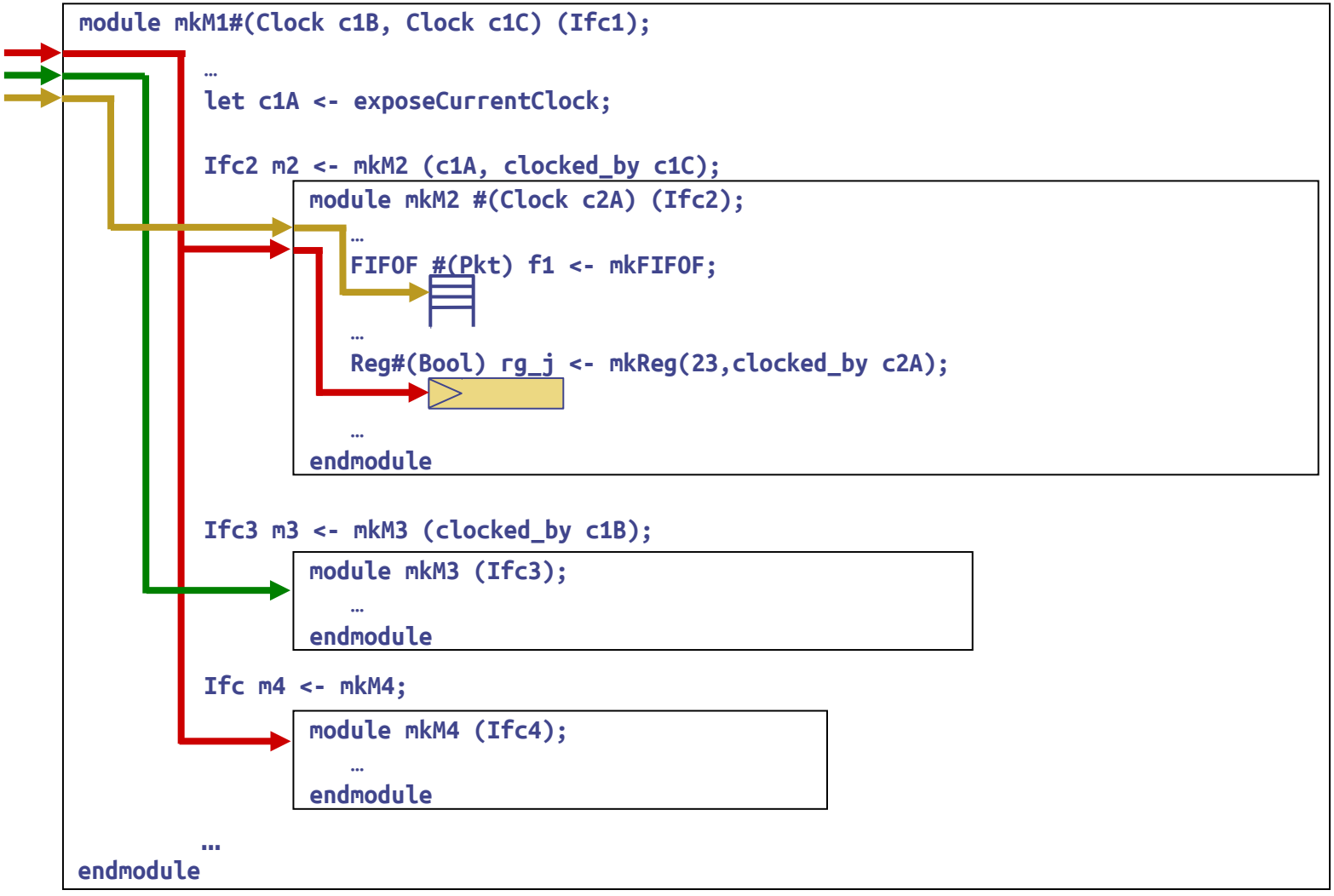
```
module mkMod1#(Clock c1b, Clock c1c, …) (Ifc1);
   …
   Clock c1a <- exposeCurrentClock;
   Ifc2 m2 <- mkMod2 (c1a,True, clocked_by c1b);
   …
endmodule

module mkMod2 #(Clock c2c, Bool b) (Ifc2);
   …
endmodule
```

mkMod1 has 3 input clocks: default, c1b, c1c. c1a is a local name for the default clock.

The m2 instance of mkMod2 uses c1b as its default clock, and also has access to c1a as an explicit parameter.

Inside m2, c1a is known as c2c. The boolean b is just an ordinary (non-clock) parameter.

Example: explicit clocks and explicit plumbing



(instance of) mkM1:
- default clock, c1A
- c1B
- c1C

Instance m2:
- default clock
- c2A

f1 clocked by default clock
rg_j clocked by c2A

Instance m3:
- default clock

Everything inside m3 will have the same clock

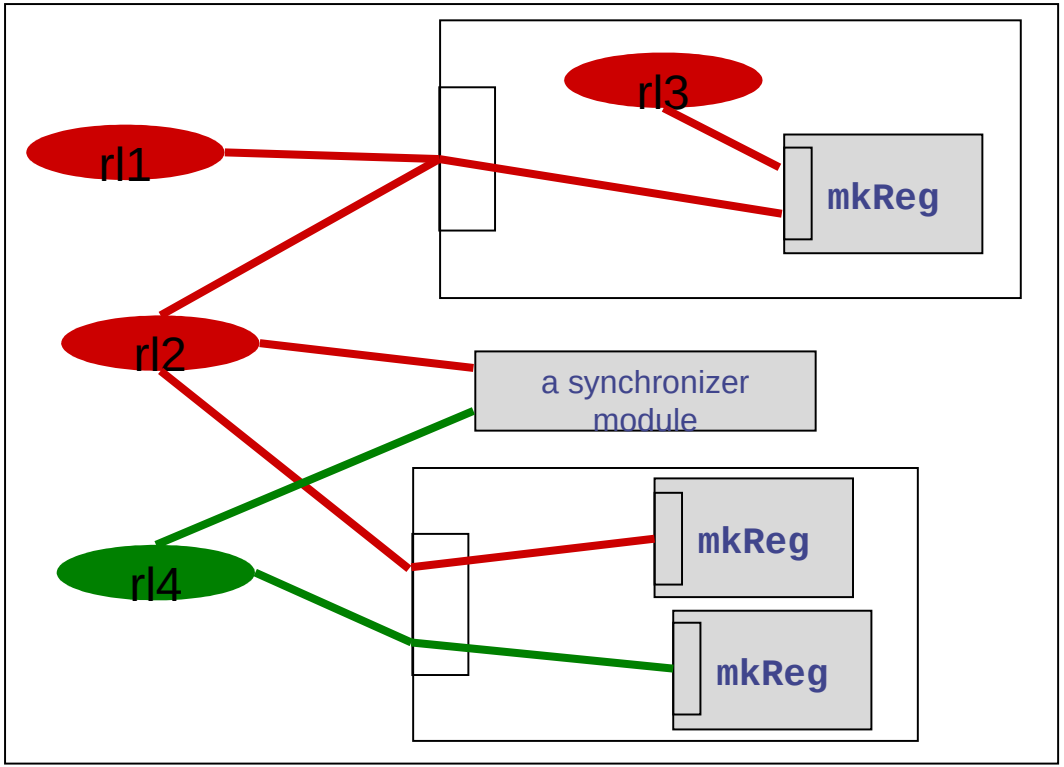
Instance m4:
- default clock

Everything inside m4 will have the same clock

Clock domains in BSV

Clock domains in BSV

- A rule, and every method it invokes, directly or indirectly, must be on the same clock.
- All the rules that are on the same clock constitute a clock domain.
- Clock gating does not play any role in defining clock domains



The diagram illustrates two clock domains, colored red and green, respectively. Note:

- rl1 and rl2 must be on the same clock since they call a common method
- A module can contain rules/methods in different clock domains, i.e., clock domains may not follow the module hierarchy
- Clock domains are completely disjoint, and cover the entire design

The only place where different clock domains “touch” is at a synchronizer module. These are primitive modules with methods on different clocks (many examples to be discussed shortly).

The clockOf() function

- From the previous slide, it should be clear that every combinational circuit (every data expression in BSV) must be in some clock domain, since every data expression is part of some rule or method
- The ‘clockOf()’ function may be applied to any BSV expression and returns a value of type Clock—the clock for that expression
  - If the expression is a constant (does not involve any method calls), the result is the special value noClock

Example:

```
Reg #(Uint #(17)) x <- mkReg (0, clocked_by  c);
let y = x + 2;
Clock c1 = clockOf (x + 3);
Clock c2 = clockOf (y - 10);
```

c1 and c2 are the same clock

Clocks and rule/method scheduling

- Each clock domain is scheduled independently of all other clock domains
- This is because the methods of primitive synchronizer modules that are on different clocks are always defined to be “conflict-free”. Since this is the only place where two clock domains “meet”, this implies that one clock domain never imposes any scheduling constraint on another.
- Concretely, this means that the rules in a clock domain have their own linear schedule, which has no relationship to the schedule of rules in another clock domain.

Gated Clocks and rule/method readiness

- The clock domain of a rule or a method may contain zero or more gated clocks (different methods they invoke may have different gatings).
- These gating conditions are AND'ed together and contribute to rule and method conditions (CAN_FIRE for a rule, READY for a method). This ensures that we do not accidentally try to invoke a method whose clock gate is currently OFF.

- Value methods are not disabled by clock gating conditions
  - They remain ready if they were ready when the clock was gated off

- Example:

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

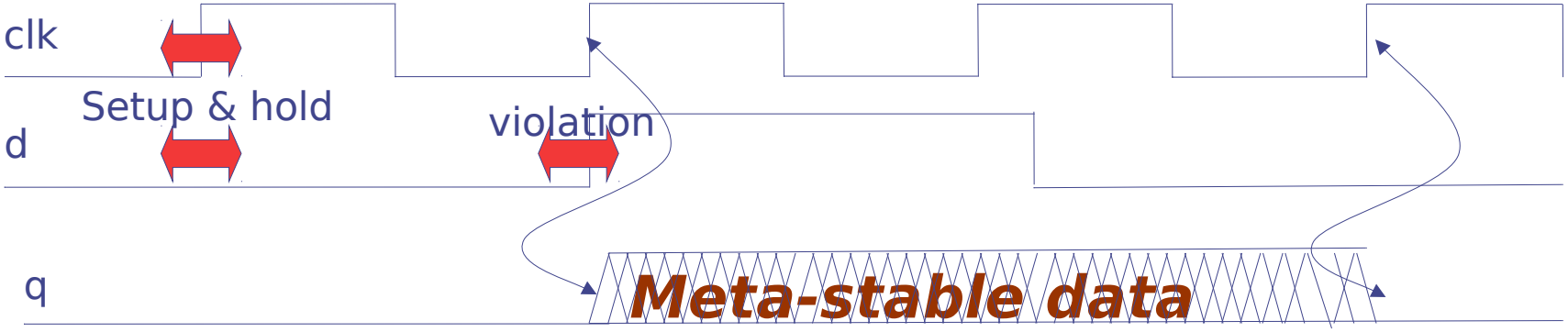
If clk1 is gated off:

- myFifo.enq, myFifo.deq and myFifo.clear are not READY
  - (and any rule that invokes these methods cannot fire)
- myFifo.first remains READY if the FIFO was non-empty when the clock was gated off

Communicating between clock domains using synchronizers

Moving Data Across Clock Domains

- Data moved across clock domains appears asynchronous to the receiving (destination) domain, because clock edges in the source and destination domains have no specific timing relationship
- Asynchronous data can cause meta-stability, i.e., circuits can violate digital discipline by sitting for an arbitrarily long time at an analog voltage that cannot be detected unambiguously either as a ‘0’ or a ‘1’



- The only safe way to move data between clock domains is through certain carefully engineered primitives called *synchronizers*

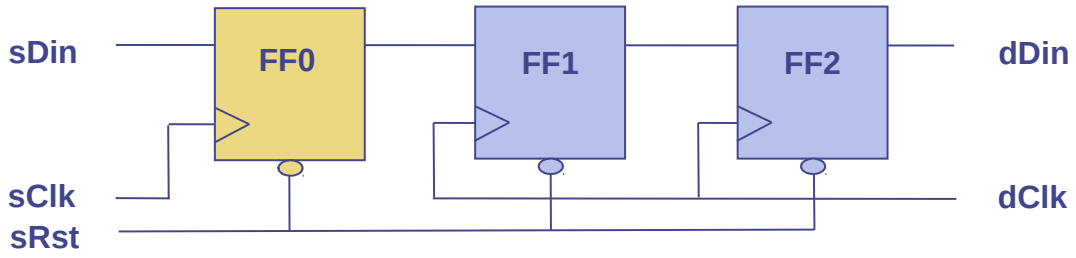
Synchronizers

- Good synchronizer design and usage reduces the probability of observing meta-stable data
- Good design discipline insists on synchronizers for all domain crossings
  - BSV enforces this through static (compile-time) checking
  - (With RTL, clock domain discipline is usually checked with post-facto “lint”-like tools which can be inaccurate, quite fragile, and susceptible to minor variations in design style)

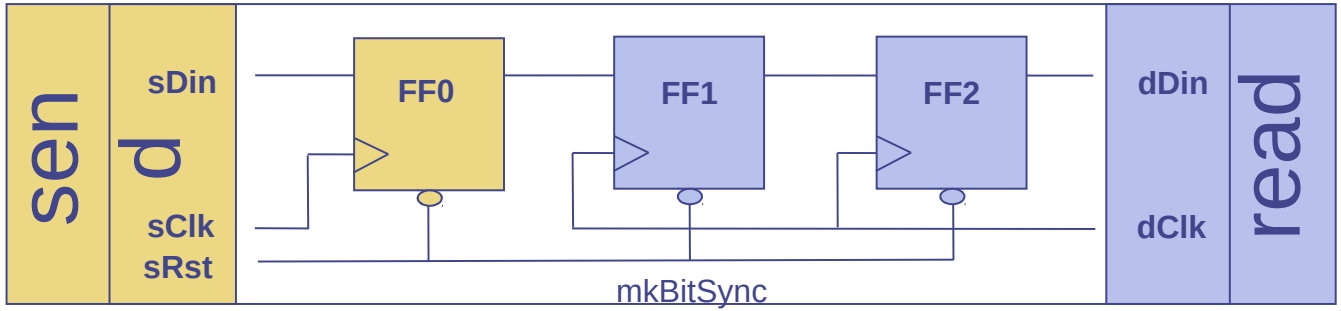
- The BSV library provides a wide range of conservative (speed independent) synchronizers (examples on the following pages)
- However, many organizations have their own libraries of internally-developed synchronizers. These can be used instead of, or in addition to, BSV’s libraries, by utilizing BSV’s mechanism to “import” RTL (see Reference Guide Sec.15)

1-bit synchronizer

- This is the most common type of synchronizer, for carrying a 1-bit signal across a clock domain boundary
- The picture below shows a typical implementation. Even though FF1 can go meta-stable, FF2 does not look at the data until a clock period later, giving FF1 time to stabilize.
- Limitations:
  - When moving from fast to slow clocks data may be overrun
  - Cannot use *n* of these in parallel to synchronize an *n*-bit word, since the bits may not arrive at the destination on the same clock edge



1-bit synchronizer: BSV encapsulation



The classical 1-bit synchronizer circuit is encapsulated into BSV as a module that is clocked by the source and destination clocks, and has “send” and “read” methods on those clocks, respectively:

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

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

Example: 1-bit synchronizer

- A counter that is enabled by a signal from another clock domain
- Registers:

```
Reg# (Bit# (32)) cntr <- mkReg(0);      // Default clock

Reg# (Bit#(1))  enable_count <- mkReg(0, clocked_by clk2);
```

- The Rule (attempt 1):

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

Illegal Clock Domain Crossing

- It's illegal because the method enable_count._read() is on a different clock from the methods cntr._read() and cntr._write()
- This error will be detected and reported by the *bsc* tool

Example: 1-bit synchronizer (contd.)


```
module mkTopLevel #(Clock clk2, Reset rst2) (Empty);
    Reg# (Bit# (32)) cntr <- mkReg (0 ) ;          // Default Clock
    Reg# (Bit# (1))  enable_count <- mkReg(0, clocked_by clk2);

    Clock currentClk <- exposeCurrentClock ;       // Default clock
    SyncBitIfc#(Bit#(1)) sync <- mkSyncBit(clk2, rst2, currentClk);

    rule cross;
        sync.send( enable_count );
    endrule

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

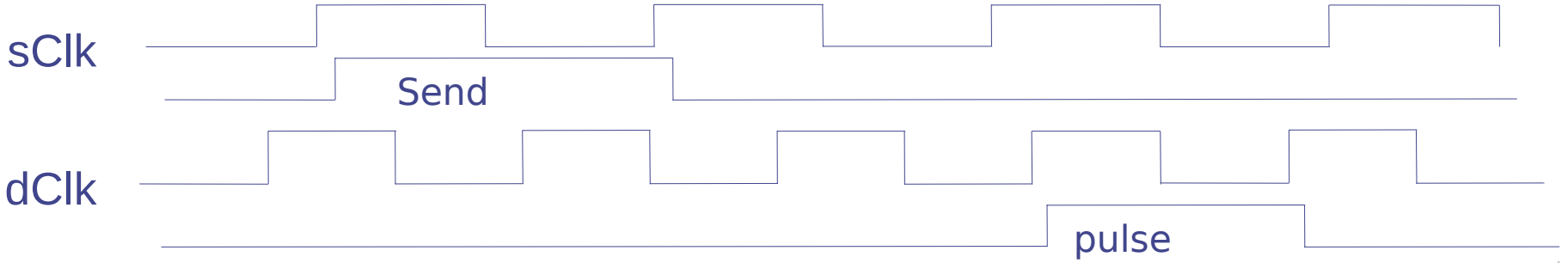
All methods in rule countup are now in the same clock domain

Pulse Synchronizer

- This is also 1-bit wide, but instead of delivering a level (0/1) across clock domains, it sends single clock width pulses
- Caveat: two “send”s may not be seen if they occur too close together (faster than 2*dstClk), i.e., pulses can be dropped

```
interface SyncPulseIfc ;
   method Action send;
   method Bool   pulse;
endinterface
```

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



Handshake Pulse Synchronizer

- A Pulse Synchronizer with a handshake protocol, so that pulses are not dropped
- After the send method is invoked, its method condition goes False (it is disabled) until the the pulse has reached the other domain
- Latency:
  - send to read is (2*dstClk)
  - send to next send (2*dstClks+ 2*srcClk)

Same interface, and similar module header, as previous example:

```
interface SyncPulseIfc ;
   method Action send;
   method Bool   pulse;
endinterface
```

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

Register Synchronizer

- Uses same Reg#(a) interface as ordinary registers, but _read and _write are in different clock domains
- The _write method has an (implicit) ready condition, to allow time for data to be available for _read.
- No guarantee that destination reads the data, only that it arrives

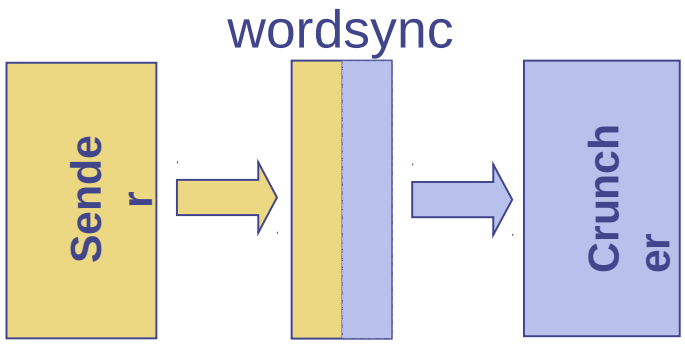
```
interface Reg#(type t);
   method Action  _write(t a);
   method t       _read;
endinterface
```

Standard Reg interface

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

Example: Register Synchronizer

```
interface Sender;
   method Bit#(32) wordOut;
   method Bool     valid;
endinterface

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




```
module top#(Clock clk2, Reset rst2)(Top);

   Sender   sender   <- mkSender(clocked_by clk2, reset_by rst2);
   Cruncher cruncher <- mkCruncher;

   …
   Clock    clk             <- exposeCurrentClock;
   Reg#(Bit# (32)) wordSync <- mkSyncReg(0, clk2, rst2, clk);
   …
   rule r1(sender.valid);  // in domain: clk2, rst2
      wordSync <= sender.wordOut;
   endrule
   …
   rule r2;
      cruncher.crunch( wordsync );    // in domain: clk
   endrule
```

FIFO Synchronizer

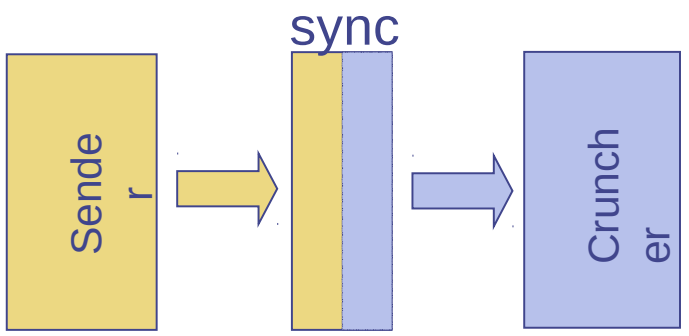
- Good for buffered, flow-controlled transmission across clock domains
- Like a standard FIFO, except that items are enqueued in one clock domain, and read/dequeued in another clock domain

```
interface SyncFIFOIfc#(type t);
   method Bool   notFull;
   method Action enq (t a);

   method Bool   notEmpty;
   method Action deq;
   method t      first;
endinterface
```

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

Example: FIFO Synchronizer

```
interface Sender;
   method Bit#(32) wordOut;
   method Bool     valid;
endinterface

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




```
module mkTop#(Clock clk2, Reset rst2)(Top);

   Sender    sender   <- mkSender(clocked_by clk2, reset_by rst2);
   Cruncher  cruncher <- mkCruncher;
   …
   Clock     clk      <- exposeCurrentClock;
   SyncFIFOIfc#(Bit# (32)) fifoSync <- mkSyncFIFO(4,clk2,rst2,clk);
   …
   rule r1 ( sender.valid );          // in domain: clk2, rst2
      fifoSync.enq( sender.wordOut);
   endrule
   …
   rule r2;
     cruncher.crunch(fifoSync.first);  // in domain: clk
   endrule
```

Null Synchronizer

- There are some situations where the user may have external knowledge that synchronization logic is not necessary (no danger of meta-stability). For example, suppose a design specifies two input clocks, but is used in some contexts where both clocks are fed by the same external clock.
- In such situations, you can use a Null synchronizer, which appeases *bsc*'s strict clock discipline checker, but which contains no synchronization logic

```
interface ReadOnly#(type t);
   method t  _read;
endinterface
```

This is a standard BSV interface (not specifically for synchronizers). It is the _read half of a Reg interface.

This module contains nothing but a wire

```
module mkNullCrossingWire #(Clock  dClkIn, t dataIn)
                           (ReadOnly #(t));
```

(for *n*-bit null crossings, you can use mkNullCrossingReg)

Example: Null Synchronizer

```
module mkTopLevel #(Clock clk2, Reset rst2 ) (Empty);
   Reg# (Bit#(32)) cntr <- mkReg (0 ) ;          // Default Clock
   Reg# (Bit#(1))  enable_count <- mkReg(0, clocked_by clk2);

   ReadOnly#(Bit#(1)) nullSync <- mkNullCrossingWire(clk2, enable_count);

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

Null synchronizers are often used conditionally, i.e., depending on whether the context is safe or not for use of a null synchronizer; see example on next page.

Example: Null Synchronizer (contd.)

```
module mkTopLevel #(Bool safe_context, Clock clk2, Reset rst2 ) (Empty);
    Reg# (Bit#(32)) cntr <- mkReg (0 ) ;          // Default Clock
    Reg# (Bit#(1))  enable_count <- mkReg(0, clocked_by clk2);

  Bit#(1) b;
  if (safe_context) begin
     ReadOnly#(Bit#(1)) nullSync <- mkNullCrossingWire(clk2, enable_count);
     b = nullSync;
  end
  else begin
     Clock currentClk <- exposeCurrentClock ;       // Default clock
      SyncBitIfc#(Bit#(1)) sync <- mkSyncBit(clk2, rst2, currentClk);
     rule cross;
         sync.send( enable_count );
      endrule
     b = sync.read;
  end

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

- Here, safe_context is an external boolean indicating synchronization is unnecessary
  - It must be static, i.e., resolved at compile time by *bsc*
- If True, enable_count is taken through a null synchronizer before use in rule countup
- If False, enable_count is taken through mkSyncBit

Resets

Resets

Much of our description of Clocks can be repeated for Resets

- There is a special type called “Reset”, and strong type-checking ensures that it can never be confused with ordinary signals or clocks
- Reset are “plumbed” through the module hierarchy just like clocks
  - Every module instance has a default Reset, and can take additional resets as parameters
  - By default, a module instance inherits its parents’ default Reset, but this can be overridden using a “reset_by” clause in the instantiation
  - When the whole design uses a single default reset throughout, the source code never mentions resets

- “Reset discipline” is not as strict as clock domain discipline:
  - Every method is associated with a reset (just like it is with a clock)
  - However, a rule or method can invoke methods with different resets. The *bsc* tool will warn about this (it is not an error). A rule that executes while some of its methods are still in reset will have unpredictable behavior (and so it is not good practice to have mixed resets in a rule or method).

Creating clocks and resets for simulation

Creating clocks and reset for simulation

In actual hardware, clocks are resets are usually “external” inputs to a design, and are generated, shaped and conditioned by specially engineered circuits before being fed to the design. As such, one does not use BSV to design clock- and reset-generation circuits.

However, in simulation, for testing, debugging and analysis, one may want to “generate” clocks and resets in the testbench, to be fed to the DUT (Design Under Test). For this purpose, BSV provides a number of modules for creation and derivation of clocks and resets.

Please see the Reference Guide Secs. C.9.1 and C.9.10 for details about these facilities

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

bluespec

End

Questions?

Join online forums at www.bluespec.com, and ask your question,
or send an e-mail to support@bluespec.com