CS250B: Modern Computer Systems

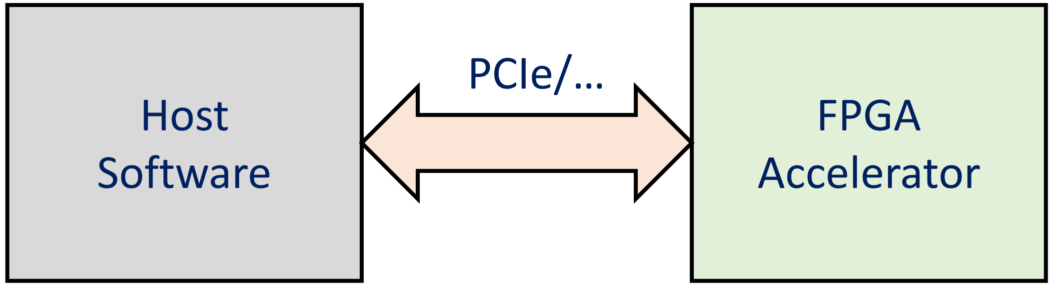
Programming FPGAs With Bluespec

Sang-Woo Jun

Many slides adapted from
Arvind’s MIT “6.175: Constructive Computer Architecture”
and Hyoukjun Kwon‘s Gatech “Designing CNN Accelerators”

FPGA Accelerator Programming Model

- Accelerated application includes both software and hardware portions
  - Accelerator-aware software sends and receives data, controls accelerator
  - Accelerator performs the heavy lifting
  - Typically the two components use different programming languages, toolchain, …
- Similarities with GPU programming
  - GPU executes explicitly implemented kernels, communicating with host software
  - But somewhat unified programming language (CUDA C)
  - Kernel is also software in GPU, FPGA kernel implemented in hardware



Programming FPGAs

- ❑ Languages and tools overlap with ASIC/VLSI design
  - ○ 😳
- ❑ FPGAs for acceleration typically done with either
  - ○ Hardware Description Languages (HDL): Register-Transfer Level (RTL) languages
  - ○ High-Level Synthesis: Compiler translates software programming languages to RTL

- ❑ We are nearing the far end of the performance/programmability spectrum at this point

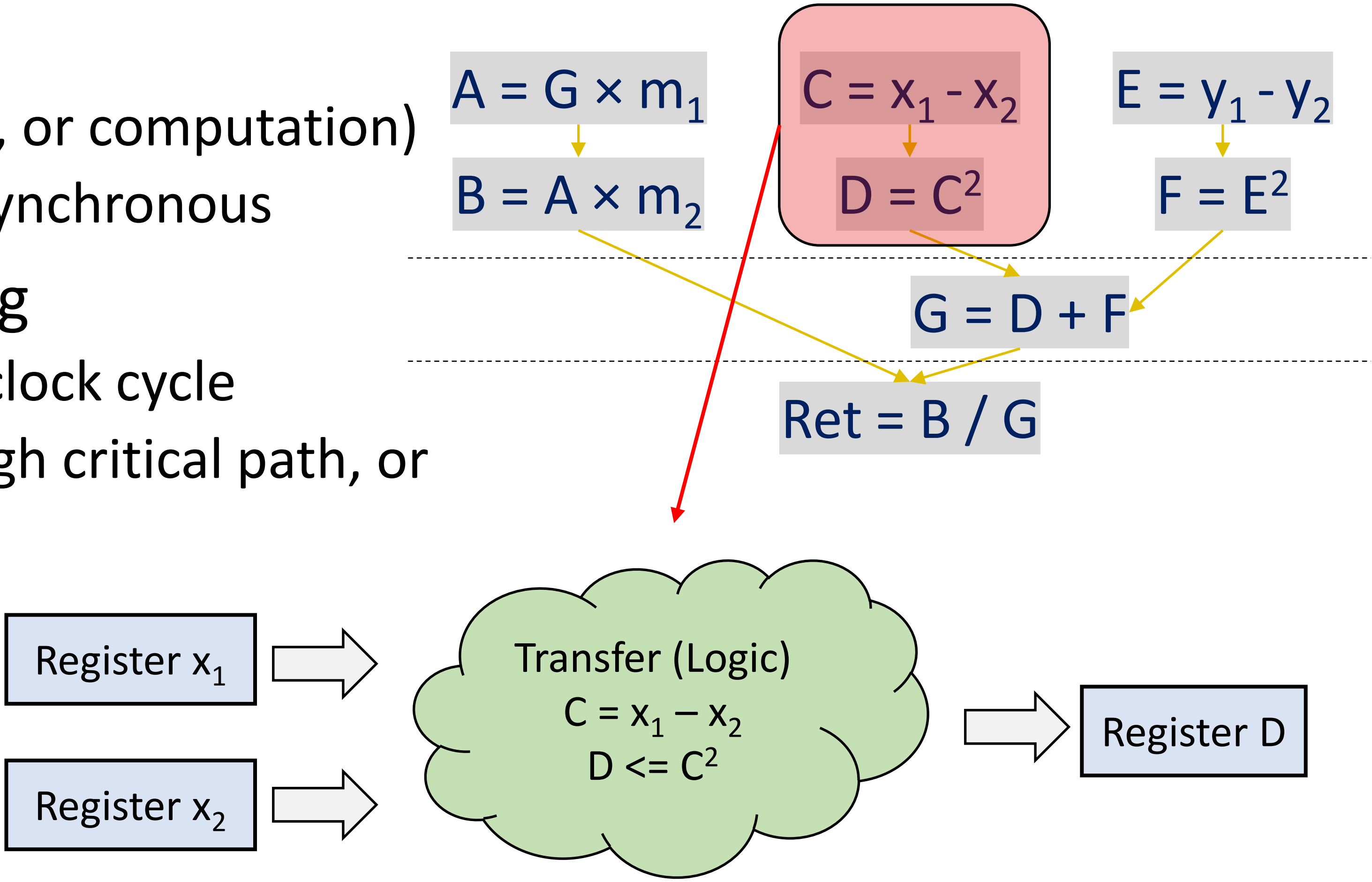
Major Hardware Description Languages

- ❑ Verilog: Most widely used in industry
    - o Relatively low-level language supported by everyone
- ❑ Chisel – Compiles to Verilog
    - o Relatively high-level language from Berkeley
    - o Embedded in the Scala programming language
    - o Prominently used in RISC-V development (Rocket core, etc)
- ❑ Bluespec – Compiles to Verilog
    - o Relatively high-level language from MIT
    - o Supports types, interfaces, etc
    - o Also active RISC-V development (Piccolo, etc)
- ❑ SpinalHDL, MyHDL, …

Register-Transfer Level

$$\frac{G \times m_1 \times m_2}{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

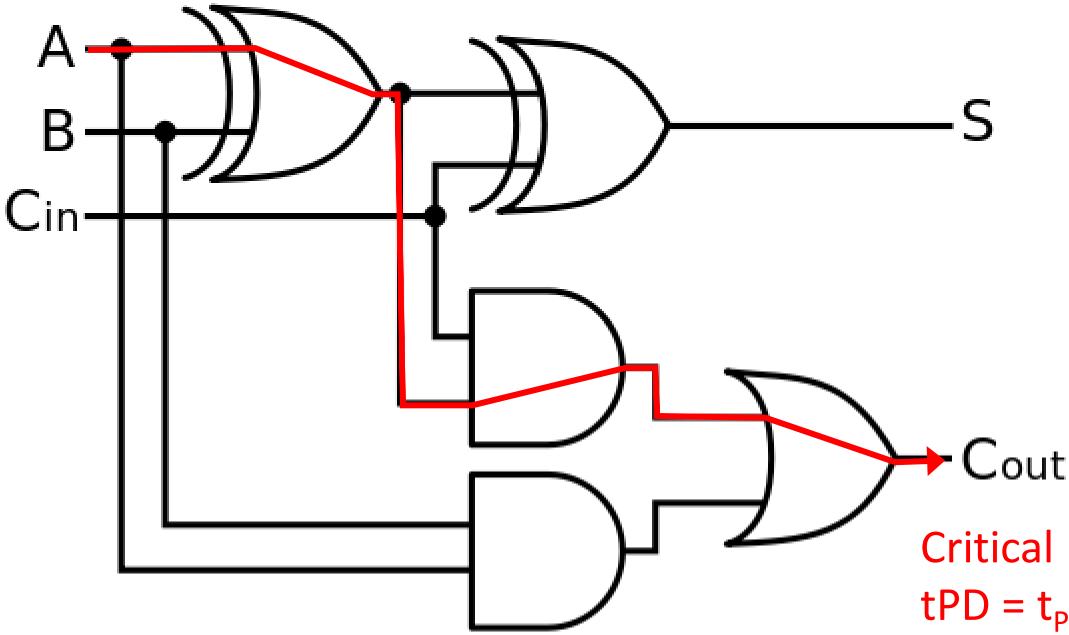
- ❑ RTL models a circuit using:
  - ○ Registers (**State**), and
  - ○ Combinational logic (**Transfer**, or computation)
  - ○ Typically everything is clock-synchronous
- ❑ Unfamiliar constraint: Timing
  - ○ Transfer must finish within a clock cycle
  - ○ Logic must have a short enough critical path, or
  - ○ Clock must be slow enough



x_1, x_2, D is state, C is not!

Reminder: Critical Path

- A chain of logic components has additive delay
  - The "depth" of combinational circuits is important
- The "critical path" defines the overall propagation delay of a circuit



Critical path of three components

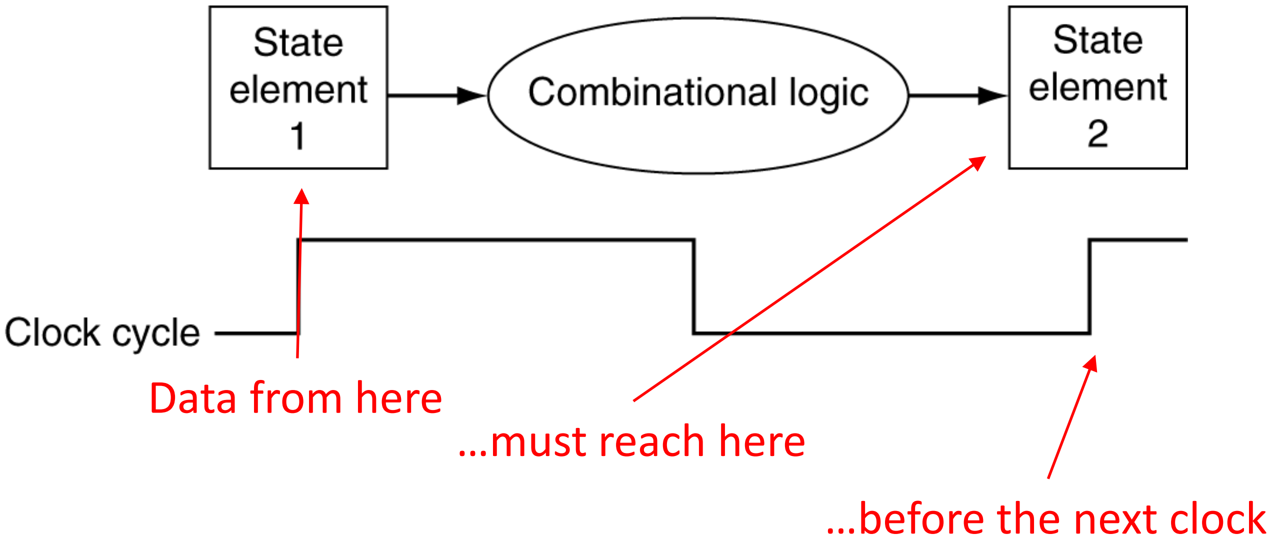
tPD = t_{PD}(xor2)+t_{PD} (and2)+t_{PD} (or2)

Example: A full adder

Source: en:User:Cburnett @ Wikimedia

Timing Behavior of State Elements

- Meeting the setup time constraint
  - "Processing must fit in clock cycle"
  - After rising clock edge,
  - t_{PD}(State element 1) + t_{PD}(Combinational logic) + t_{SETUP}(State element 2)
  - must be **smaller** than the clock period



Otherwise, "timing violation"

Complexities of RTL

- Example RTL logic:
  - Reg#(Bit#(64)) A, B; // Two 64-bit registers
  - A <= (A>>B); // Somewhere, do a variable-width shift
  - This is very inefficient on an FPGA! Very long critical path
    - Long critical path -> Slow clock
    - Aside: Reg#(Bit#(2)) B; then A>>(B*16); Generates much better hardware
- Kind of have to know what kind of circuits are generated by what logic
  - Typically covered by a few rule of thumbs
  - Will be covered later!

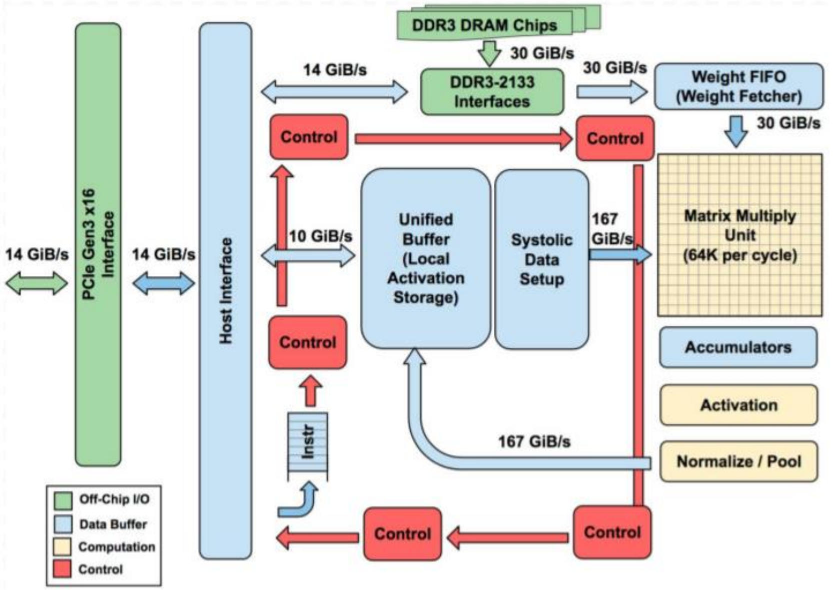
Complexities of RTL

- ❑ Another RTL Example
  - o Reg#(Int#(32)) a, b, c, d, e;
  - o e <= a*b*c*d/e;
  - o Multipliers and divisors are complex, long critical paths!

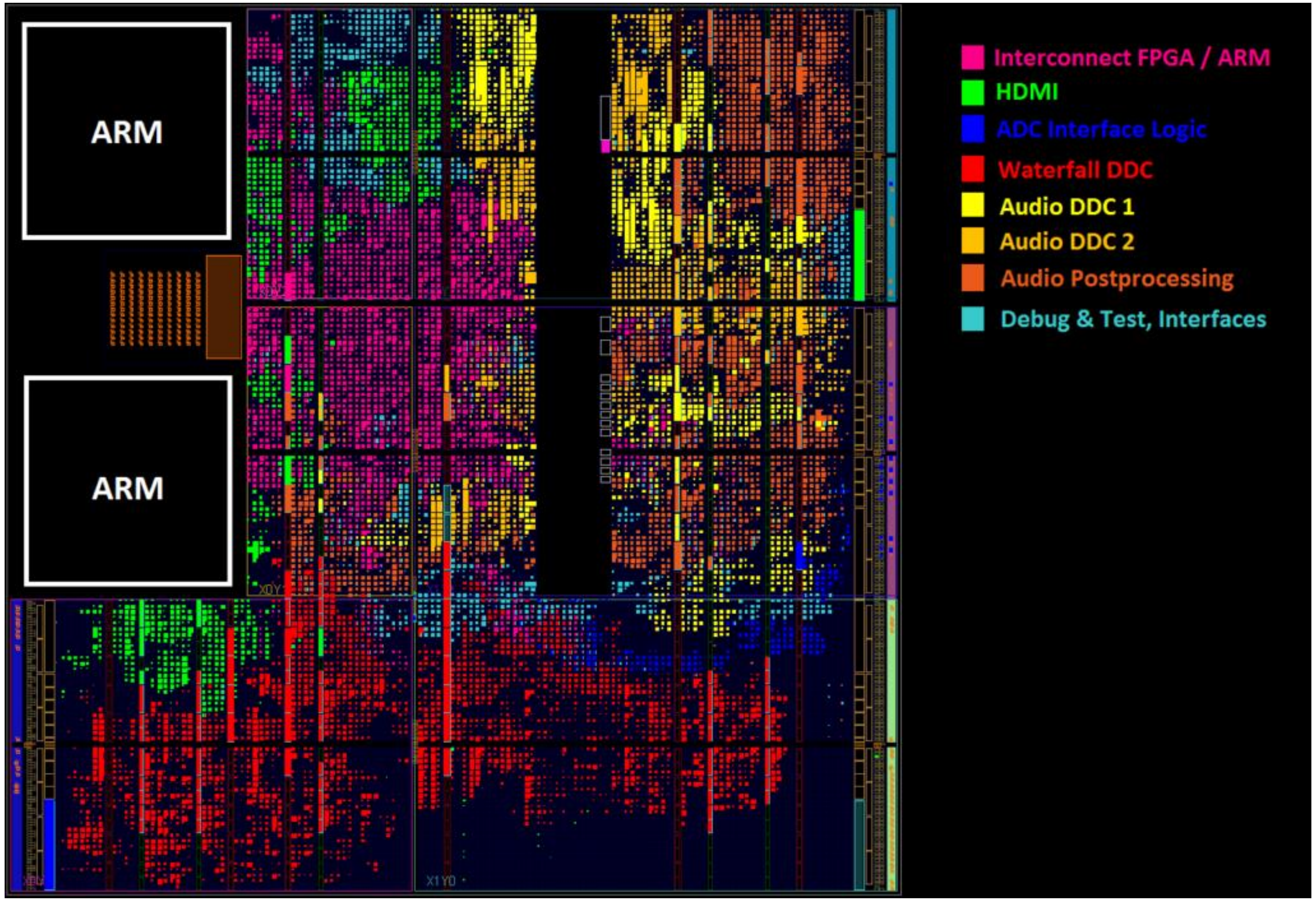
- ❑ Not all arbitrary clock speeds are available
  - o Small number of fixed speed clocks given as input to chip
  - o Multiply/divide clocks to get different frequencies
  - o For practical reasons, target clocks are often fixed, and circuit designed for it

Complexities of RTL

- ❑ Pipelining, datapath, etc must be explicitly handled
- ❑ e.g., ALU with two 32 bit inputs and one 32 bit output
  - Can only process two inputs per cycle
  - Running at 250 MHz, 2 GB/s data sink
  - Even if ALU internally included SIMD unit capable of dozens of GB/s, performance is bottlenecked by the port width



Example FPGA Layout



All functionality occupies chip space/resources

- CLBs/BRAM/DSPs/…

Complex functionality may be difficult to fit

- Run out of resources globally
  (No more resources on chip)

- Runs out of resources locally
  (Due to placement constraints)
  e.g., Too many modules need to be near ARM core, or some IO pad
  Due to timing constraints

Details later!

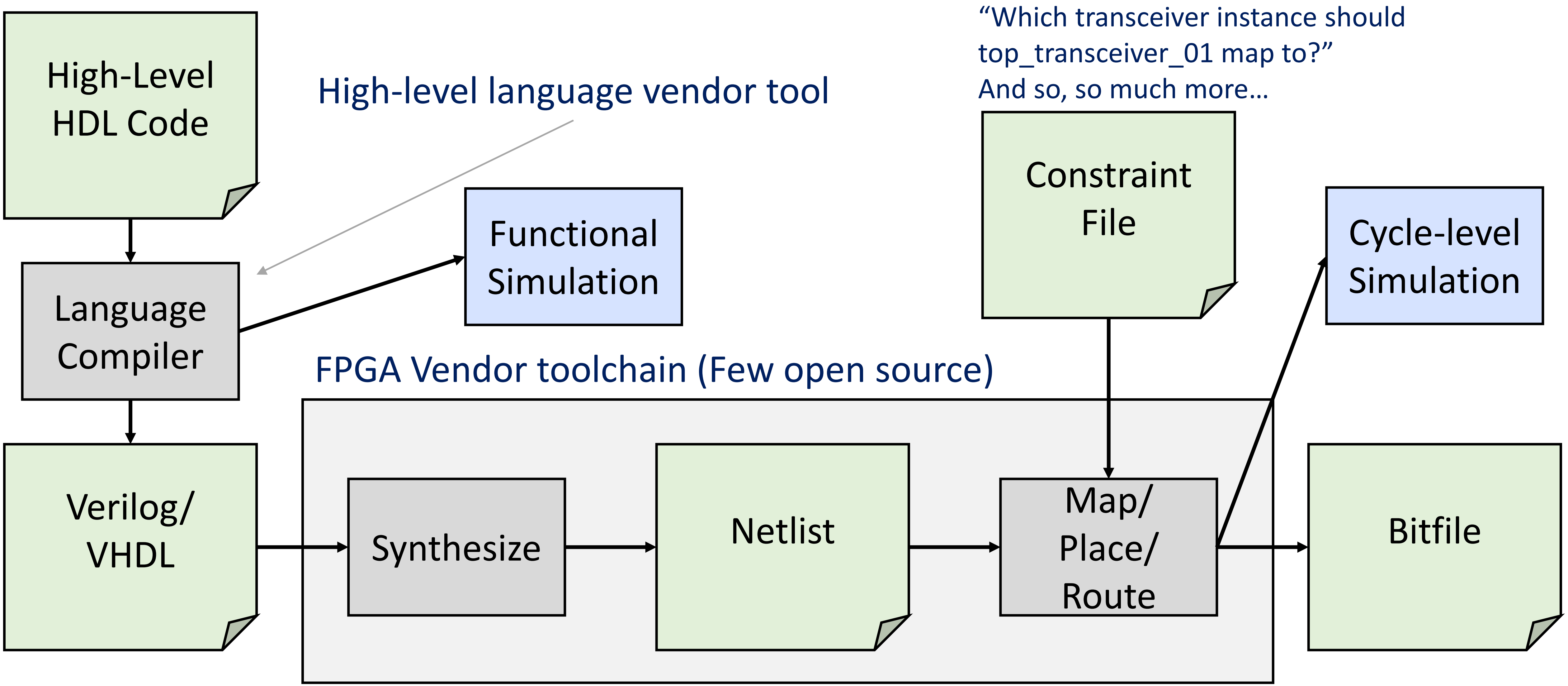
Panoradio SDR, “FPGA Floorplan for high-speed SDR processing,” Accessed 2021 – Using Zynq 7020 chip

High-Level Synthesis

- ❑ Compiler translates software programming languages to RTL
- ❑ High-Level Synthesis compiler from Xilinx, Altera/Intel
  - Compiles C/C++, annotated with ***#pragma***’s into RTL
  - Theory/history behind it is a complex can of worms we won’t go into
  - Personal experience: needs to be HEAVILY annotated to get performance
  - Anecdote: Naïve RISC-V in Vivado HLS achieves IPC of 0.0002 [1], 0.04 after optimizations [2]
- ❑ OpenCL
  - Inherently parallel language more efficiently translated to hardware
  - Stable software interface

[1] http://msyksphinz.hatenablog.com/entry/2019/02/20/040000
[2] http://msyksphinz.hatenablog.com/entry/2019/02/27/040000

FPGA Compilation Toolchain



“Which transceiver instance should top_transceiver_01 map to?”
And so, so much more…

High-level language vendor tool

High-Level HDL Code

Language Compiler

Functional Simulation

Constraint File

Cycle-level Simulation

FPGA Vendor toolchain (Few open source)

Verilog/ VHDL

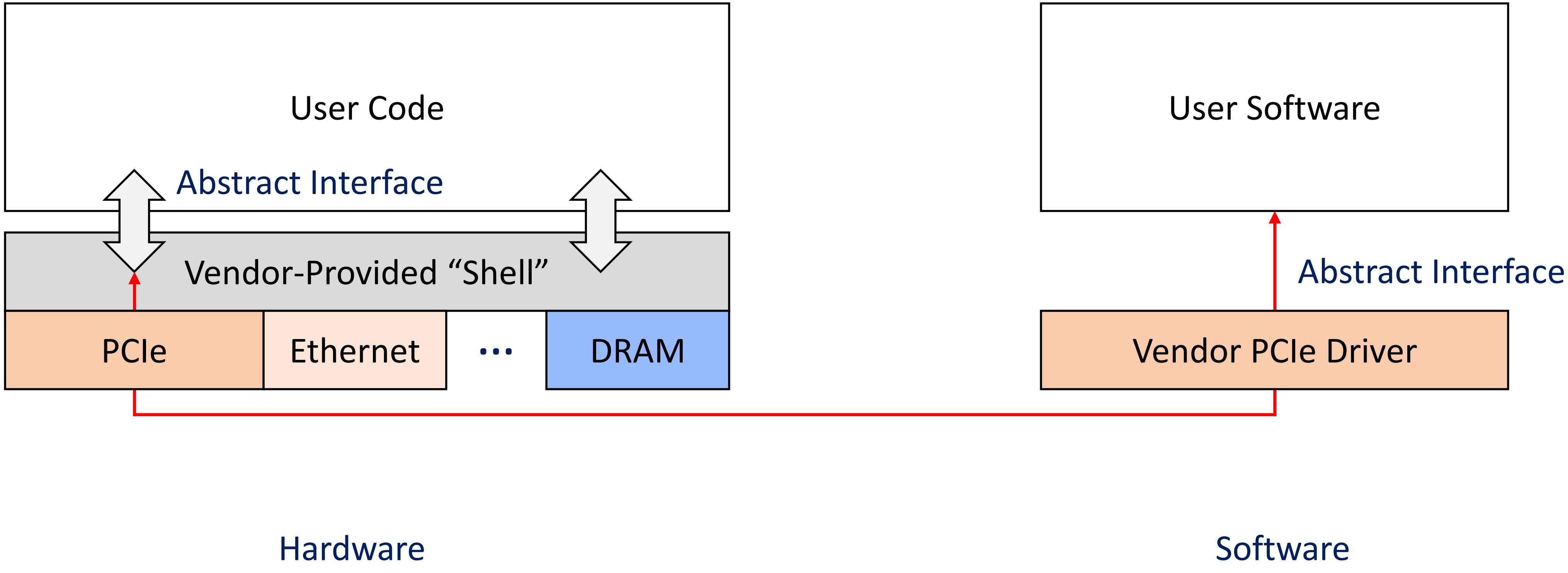
Synthesize

Netlist

Map/ Place/ Route

Bitfile

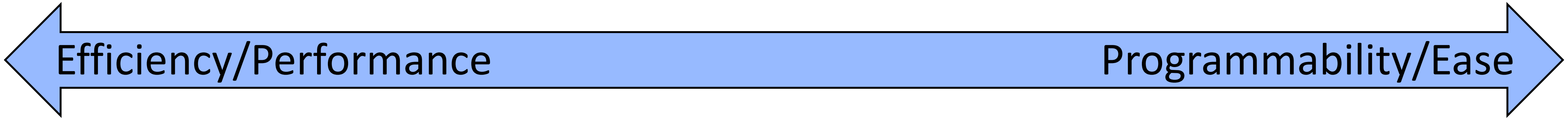
Example System Abstraction For Accelerators



Programming/Using an FPGA Accelerator

- Bitfile is programmed to FPGA over “JTAG” interface
  - Typically used over USB cable
  - Supports FPGA programming, limited debugging access, etc
  - Kind of slow…
  - Bitfile often stored in on-board flash for persistence
- Modern FPGAs provide faster programming methods as well
  - On-chip accelerator to load from local memory
    - e.g., Xilinx ICAP (Internal Configuration Access Port)
  - Milliseconds to program a new design

Various Hardware Description Languages



Efficiency/Performance

Programmability/Ease

Assembly	C/C++	MATLAB Python
Verilog VHDL	Bluespec Chisel	OpenCL High-Level Synthesis

De-facto standard

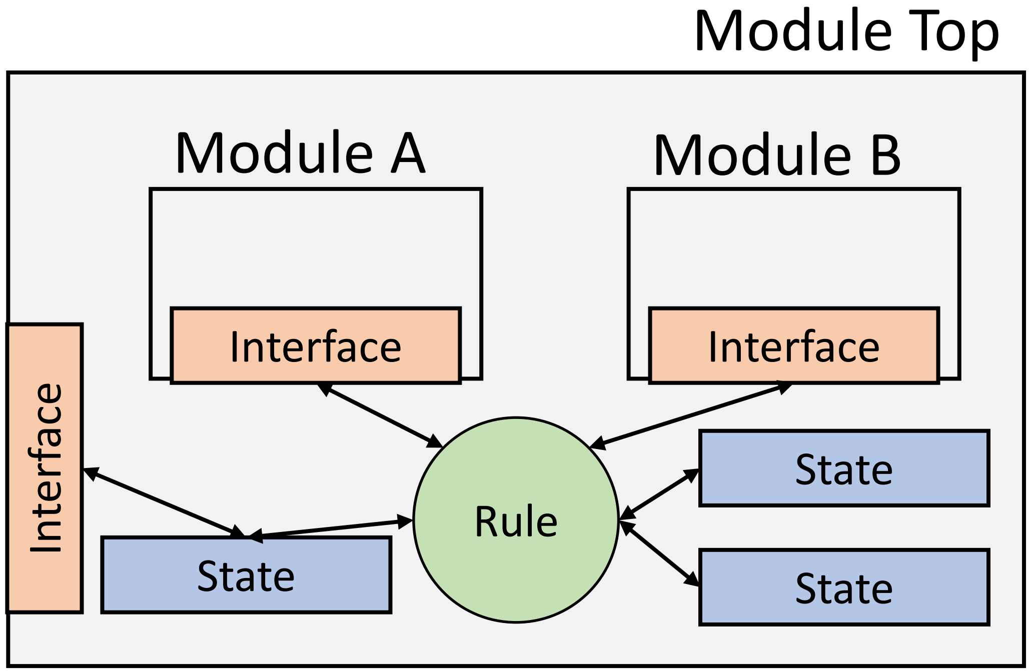
Bluespec System Verilog (BSV)

- ❑ “High-level HDL without performance compromise”
- ❑ Comprehensive type system and type-checking
  - ○ Types, enums, structs
- ❑ Static elaboration, parameterization (Kind of like C++ templates)
  - ○ Efficient code re-use
- ❑ Efficient functional simulator (bluesim)
- ❑ Most expertise transferrable between Verilog/Bluespec

> In a comparison with a 1.5 million gate ASIC coded in Verilog, Bluespec demonstrated a 13x reduction in source code, a 66% reduction in verification bugs, equivalent speed/area performance, and additional design space exploration within time budgets.
>
> -- PineStream consulting group

Bluespec System Verilog (BSV) High-Level

- ❑ Everything organized into “Modules” – Physical entities on chip
  - ○ Modules have an “interface” which other modules use to access state
  - ○ A Bluespec model is a single top-level module consisting of other modules, etc
- ❑ Modules consist of state (other modules) and behavior
  - ○ State: Registers, FIFOs, RAM, …
  - ○ Behavior: Rules



Greatest Common Divisor Example

- ❑ Euclid’s algorithm for computing the greatest common divisor (GCD)

X	Y	
15	6	
9	6	subtract
3	6	subtract
6	3	swap
3	3	subtract
0	3	subtract

answer

State

```
module mkGCD (GDCIfc);
  Reg#(Bit#(32)) x <- mkReg(0);
  Reg#(Bit#(32)) y <- mkReg(0);
  FIFOF#(Bit#(32)) outQ <- mkSizedFIFOF(2);
```

Sub-modules

Module “mkReg” with interface “Reg”, type parameter Bit#(32), module parameter “0”*

*mkReg implementation sets initial value to “0”

outQ has a module parameter “2”*

*mkSizedFIFOF implementation sets FIFO size to 2

Rules (Behavior)

```
  rule step1 ((x > y) && (y != 0));
    x <= y; y <= x;
  endrule
  rule step2 (( x <= y) && (y != 0));
    y <= y-x;
    if ( y-x == 0 ) begin
      outQ.enq(x);
    end
  endrule
```

Interface (Behavior)

```
  method Action start(Bit#(32) a, Bit#(32) b) if (y==0);
    x <= a; y <= b;
  endmethod
  method ActionValue#(Bit#(32)) result();
    outQ.deq;
    return outQ.first;
  endmethod
endmodule
```

```
module mkGCD (GDCIfc);
```

State

```
  Reg#(Bit#(32)) x <- mkReg(0);
  Reg#(Bit#(32)) y <- mkReg(0);
  FIFOF#(Bit#(32)) outQ <- mkSizedFIFOF(2);
```

Rules (Behavior)

```
  rule step1 ((x > y) && (y != 0));
    x <= y; y <= x;
  endrule
  rule step2 (( x <= y) && (y != 0));
    y <= y-x;
    if ( y-x == 0 ) begin
      outQ.enq(x);
    end
  endrule
```

Rules are atomic transactions

“fire” whenever condition (“guard”) is met

Interface (Behavior)

```
  method Action start(Bit#(32) a, Bit#(32) b) if (y==0);
    x <= a; y <= b;
  endmethod
  method ActionValue#(Bit#(32)) result();
    outQ.deq;
    return outQ.first;
  endmethod
endmodule
```

```
module mkGCD (GDCIfc);
```

State

```
 Reg#(Bit#(32)) x <- mkReg(0);
 Reg#(Bit#(32)) y <- mkReg(0);
 FIFOF#(Bit#(32)) outQ <- mkSizedFIFOF(2);
```

Rules (Behavior)

```
 rule step1 ((x > y) && (y != 0));
  x <= y; y <= x;
 endrule
 rule step2 (( x <= y) && (y != 0));
  y <= y-x;
  if ( y-x == 0 ) begin
   outQ.enq(x);
  end
 endrule
```

Interface (Behavior)

```
 method Action start(Bit#(32) a, Bit#(32) b) if (y==0);
  x <= a; y <= b;
 endmethod
 method ActionValue#(Bit#(32)) result();
  outQ.deq;
  return outQ.first;
 endmethod
endmodule
```

Interface methods are also atomic transactions
Can be called only when guard is satisfied
When guard is not satisfied, rules that call it cannot fire

Explicit Pipelining Example

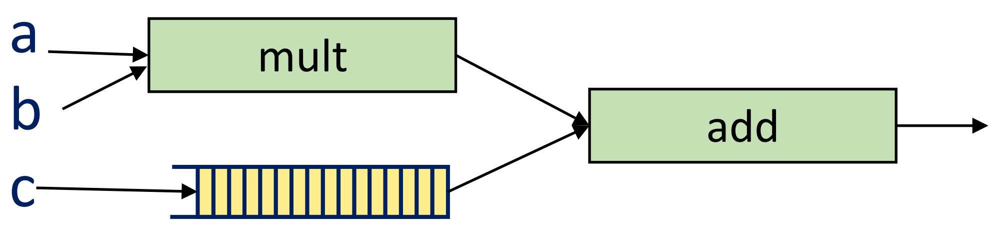
- Floating point operators are complex
  - Typically not combinational implementations
  - Multi-cycle latency, pipelined implementation
    - Input can be inserted every cycle
    - One result available per cycle
    - Answer available N cycles after corresponding input

Fused Multiply-Adder Example

```
module mkFMA (FMAIfc);
 FloatOpIfc mult <- mkFloatMult32;
 FloatOpIfc adder <- mkFloatAdd32;
 FIFOF#(Bit#(32)) latencyMatchQ <- mkSizedFIFOF(7);

 rule fma;
  let mres <- mult.get;
  latencyMatchQ.deq;
  let r = latencyMatchQ.first;
  adder.put(mres,r);
 endrule

 method Action put(Bit#(32) a, Bit#(32) b, Bit#(32) c);
  mult.put(a,b); latencyMatchQ.enq(c);
 endmethod
 method ActionValue#(Bit#(32)) get();
  let ares <- adder.get;
  return ares;
 endmethod
endmodule
```



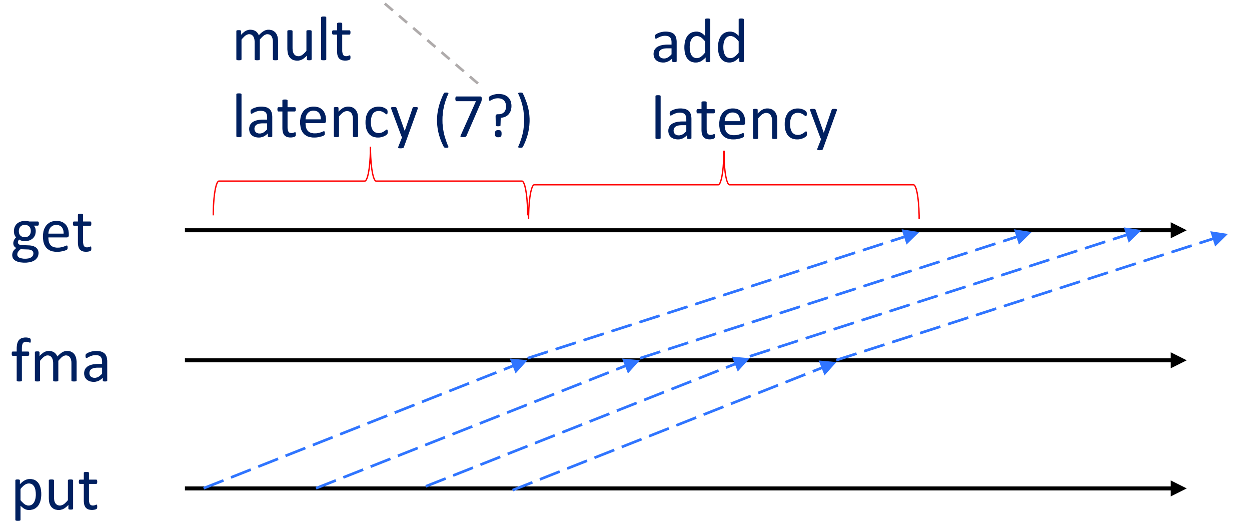
a

b

c

mult

add



mult latency (7?)

add latency

get

fma

put

Let’s Learn Bluespec

❑ Search for “BSV by example”, and “Bluespec(TM) Reference Guide” for more details

❑ Keywords:
  - Modules with interfaces
  - Rules with implicit and explicit guards

❑ Most new hardware-related concepts are shared with Verilog/other HDL

Components To Cover

- ❑ Modules and interfaces
- ❑ Rules and what’s in them
- ❑ State and non-state variables
    - o Registers, FIFOs, Wires
    - o Temporary Variables
- ❑ Functions

Bluespec Modules – Interface

- ❑ Modules encapsulates state and behavior (think C++/Java classes)
- ❑ Can be interacted with from the outside using its “interface”
  - ○ Interface definition is separate from module implementation
  - ○ Many module definitions can share the same interface: Interchangeable implementations
- ❑ Interfaces can be parameterized
  - ○ Like C++ templates “FIFO#(Bit#(32))”
  - ○ Not important right now

```
interface GDCIfc;
  method Action start(Bit#(32) a, Bit#(32) b);
  method ActionValue#(Bit#(32)) result();
endinterface
```

```
module mkGCD (GDCIfc);
    …

  method Action start(Bit#(32) a, Bit#(32) b) if (y==0);
    x <= a; y <= b;
  endmethod
  method ActionValue#(Bit#(32)) result();
    outQ.deq;
    return outQ.first;
  endmethod
endmodule
```

Bluespec Module – Interface Methods

- ❑ Three types of methods
  - ○ Action : Takes input, modifies state
  - ○ Value : Returns value, does not modify state
  - ○ ActionValue : Returns value, modifies state
- ❑ Methods can have “guards”
  - ○ Does not allow execution unless guard is True

Automatically introduces
“implicit guard”
if outQ is empty

Guard

```
method Action start(Bit#(32) a, Bit#(32) b) if (y==0);
 x <= a; y <= b;
endmethod
method ActionValue#(Bit#(32)) result();
   outQ.deq;
   return outQ.first;
 endmethod
```

```
rule ruleA;
   moduleA.actionMethod(a,b);
   Int#(32) ret = moduleA.valueMethod(c,d,e);
   Int#(32) ret2 <- moduleB.actionValueMethod(f,g);
endrule
```

Note the “<-” notation

Bluespec Modules – Polymorphism

- Modules can be parameterized with types
  - GDCIfc#(**Bit**#(32)) gdcModule <- mkGCD;
  - Reg#(**Bit**#(32)) reg1 <- mkReg(0);

  - Set “provisos” to tell compiler facts about types (how wide? comparable? etc…)
  - Will cover in more detail later

```
interface GDCIfc#(type valType);
  method Action start(valType a, valType b);
  method valType result();
endinterface
```

```
module mkGCD (GDCIfc#(valType))
   provisos(Bits#(valType,valTypeSz)
       Add#(1,a__,valTypeSz));
   …
endmodule
```

Bluespec Modules – Module Arguments

- Modules can take other modules and variables as arguments
  - GDCIfc gdcModule <- mkGCD(argumentModule, …);
  - Modules, Integers, variables, …
  - Arguments available inside module context
- However, typically not recommended
  - “argumentReg” is a single register instance. If used in many places, all users must be located nearby (on the chip) to satisfy timing constraints
  - If copies can be made, or updated via latency-insensitive signals etc, likely better

```
module mkGCD#(Reg#(Bit#(32)) argumentReg, Integer cnt) (GDCIfc#(valType));
  …
endmodule
```

Bluespec Rules

- Behavior is expressed via "rules" ("transfer" part of RTL)
  - ***Atomic*** actions on state – only executes when all conditions ("guards") are met
  - Explicit guards can be specified by programmer
  - Implicit guards: All conditions of all called methods must be met
  - If method call is inside a conditional (**if** statement), method conditions only need to be met **if** conditional is met

Explicit guard

```
rule step1 ((x > y) && (y != 0));
  x <= y; y <= x;
  if ( x == 0 ) moduleA.actionMethod(x,y);
endrule
```

Implicit guard: Rule doesn't fire if
x == 0 && actionMethod's guard is not met

Bluespec Rules

- One-rule-at-a-time semantics
  - Two rules can be fired on the same cycle when semantically they are the same as one rule firing after another
  - Compiler analyzes this and programs the scheduler to fire *as many rules at once as possible*
  - Helps with debugging – No need to worry about rule interactions
- Conflicting rules have ordering
  - Can be seen in compiler output (“xxx.sched”)
  - Can be influenced by programmer
    - (* descending_urgency *) attribute
    - Will be covered later

10,000 rules in your code can all fire at once, always
If there are no conflicts!

Bluespec Rules Are Atomic Transactions

- Each statement in rule only has access to state values from before rule began firing
- Each statement executes independently, and state update happens once as the result of rule firing
    - e.g.,
      ```
      // x == 0, y == 1
      x <= y; y <= x; // x == 1, y == 0
      ```
    - e.g.,
      ```
      // x == 0, y == 1
      x <= 1; x <= y; // write conflict error!
      ```

e.g.,

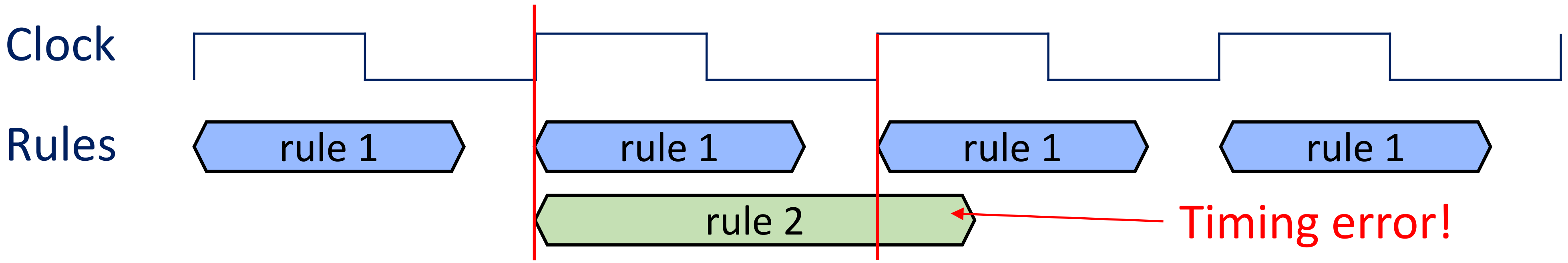
```
rule step2 ((x <= y) && (y != 0));
   y <= y-x;
   if ( y-x == 0 ) begin
     outQ.enq(x);
   end
 endrule
```

Fires if:

1. **x<=y** && **y != 0** && **y-x == 0** && **outQ.notFull**
   or
2. **x<=y** && **y != 0** && **y-x != 0**

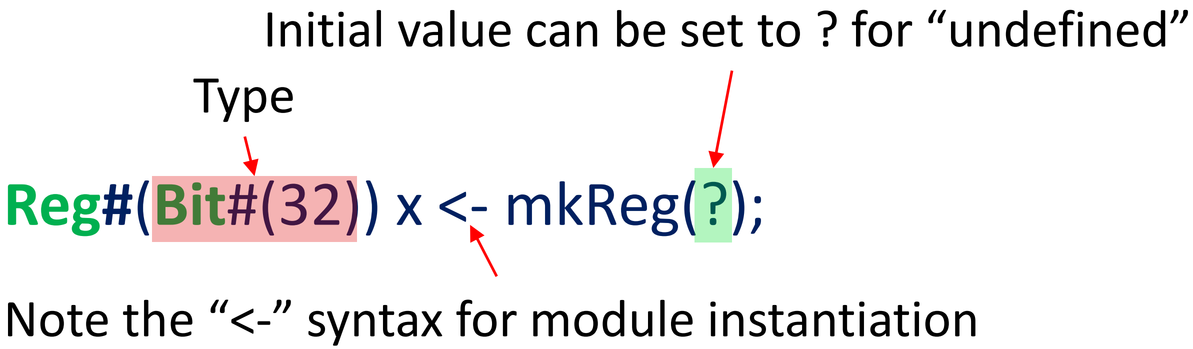
Rule Execution Is Clock-Synchronous

- Simplified explanation: A rule starts execution at a clock edge, and must finish execution before the next clock cycle
- If a rule is too complex, or has complex conditionals, it may not fit in a clock cycle
  - Synthesis tool performs static analysis of timing and emits error
  - Can choose to ignore, but may produce unstable results
- Programmer can break the rule into smaller rules, or set the clock to be fast or slow



Bluespec State

- Registers, FIFOs and other things that store state
- Expressed as modules, with their own interfaces
- Registers: One of the most fundamental modules in Bluespec
  - Registers have special methods _read and _write, which can be used implicitly
    x <= 32’hdeadbeef; **// calls action method x._write(32’hdeadbeef);**
    Bit#(32) d = x; **// calls value method d = x._read();**
  - You can make your own module interfaces have _read/_write as well!



Initial value can be set to ? for “undefined”

Type

```
Reg#(Bit#(32)) x <- mkReg(?);
```

Note the “<-” syntax for module instantiation

Bluespec Non-State

- Temporary variable names can be given to values within a rule

```
Reg#(Bit#(32)) regA <- mkReg;
rule ruleA;
   Bit#(32) dA = regA+regA;
   ….
endrule
```

- “dA” defined only within “ruleA”
  - Disappears after rule execution
  - Not accessible by other rules, or by ruleA at later execution
  - Simply a temporary label given to a value “regA+regA”

Temporary Variables

- Not actual state realized within circuit
  - Only a name/label tied to another name or combination of names
- Can be within ***or outside*** rule boundaries
  - Natural scope ordering rules apply (closest first)
- Target of “=“ assignment

```
// Variables example
FIFO#(Bool) bQ <- mkFIFO;
Reg#(Bit#(32)) x <- mkReg(0);
let bqf = bQ.first;
Bit#(32) xv = x;

rule rule1;
  Bool bqf = bQ.first ^ True;
  bQ.deq;
  let xnv = x * x;

  $display( “%d”, bqf ); // bQ2.first ^ True
endrule
```

Bluespec State – FIFO

- One of the most important modules in Bluespec
- Default implementation has size of two slots
  - Various implementations with various characteristics
  - Will be introduced later
- Parameterized interface with guarded methods
  - e.g., testQ.enq(data); **// Action method. Blocks when full**
    testQ.deq; **// Action method. Blocks when empty**
    dataType d = testQ.first; **// Value method. Blocks when empty**
- Provided as library
  - Needs “`import FIFO::*;`” at top

```
FIFO#(Bit#(32)) testQ <- mkFIFO;
rule enqdata; // rule does not fire if testQ is full
   testQ.enq(32’h0);
endrule
```

More About FIFOs

❑ Various types of FIFOs are provided

- ex) **FIFOF**#(**type**) fifofQ <- mkFIFOF;
  Two additional methods: **Bool** notEmpty, **Bool** notFull
- ex) **FIFO**#(**type**) sizedQ <- mkSizedFIFO(**Integer** slots);
  FIFO of slot size “slots”
- ex) **FIFO**#(**type**) bramQ <- mkSizedBRAMFIFO(**Integer** slots);
  FIFO of slot size “slots”, stored in on-chip BRAM
- And many more! mkSizedFIFOF, mkPipelineFIFO, mkBypassFIFO, …
  - Will be covered later, as some have to do with rule timing issues

Wires In Bluespec

- Used to transfer data between rules within the same clock cycle
- Many flavors
  - **Wire**#(**Bool**) aw <- mkWire;
    Rule reading the wire can only fire if another rule writes to the wire
  - **RWire**#(**Bool**) bw <- mkRWire;
    Reading rule can always fire, reads a “Maybe#(Bool)” value with a valid flag
    - Maybe types will be covered later
  - **DWire**#(**Bool**) cw <- mkDWire(False);
    Reading rule can always fire, reads a provided default value if not written
- Advice I was given: Do not use wires, all synchronous statements should be put in a single rule
  - Also, write small rules, divide and conquer using latency-insensitive design methodology (covered later!)

Statements In Rule -- $write

- $write(“debug message %d %x\n”, a, b);
- Prints to screen, acts like printf
- Only works when compiled for simulation
  - Ignored during synthesis

Statements In Rule

if/then/else/end

```
Bit#(16) valA = 12;
if (valA == 0) begin
  $display(“valA is zero”);
end
else if(valA != 0 && valA != 1) begin
  $display(“valA is neither zero nor one”);
end
else begin
  $display(“valA is %d”, valA);
end
```

arithmetic operations

```
Bit#(16) valA = 12; Bit#(16) valB = 2500;
Bit#(16) valC = 50000;

Bit#(16) valD = valA + valB; //2512
Bit#(16) valE = valC – valB; //47500
Bit#(16) valF = valB * valC; //Overflow! (125000000 > $2^{16}$)
                      //valF = (125000000 mod $2^{16}$)
Bit#(16) valG = valB / valA;
```

Statements In Rule

Logical Operations

```
Bit#(16) valA = 12; Bit#(16) valB = 2500;
Bit#(16) valC = 50000;

Bool valD = valA < valB; //True
Bool valE = valC == valB; //False
Bool valF = !valD; //False
Bool valG = valD &&  !valE;
```

Bit Operations

```
Bit#(4) valA = 4’b1001;  Bit#(4) valB = 4’b1100;
Bit#(8) valC = {valA, valB}; //8’b10011100

Bit#(4) valD = truncate(valC); //4’b1100
Bit#(4) valE = truncateLSB(valC); //4’b1001

Bit#(8) valF = zeroExtend(valA); //4’b00001001
Bit#(8) valG = signExtend(valA);

Bit#(2) valH = valC[1:0]; //2’b00
```

Statements In Rule – Assignment

- “=“ assignment
  - For temporary variables, blocking semantics, no effect on state
  - May be shorthand for _read method on the right hand variable
  - **// initially a == 0, b == 0**
    a = 1; b = a; **// a == 1, b == 1**
- “<=“ assignment
  - shorthand for _write method on the left variable
  - e.g., a <= b is actually a._write(b._read())
  - Non-blocking, atomic transactions on state
  - **// initially a == 0, b == 0**
    a <= 1; b <= a; **// a == 1, b == 0**

```
Reg#(Bit#(32)) x <- mkReg(0);
rule rule1;
   x <= 32’hdeadbeef; // x._write
   Bit#(32) temp = 32’hc001d00d;
   temp = temp + 4; // blocking semantics
   Bit#(32) temp2 = x; // x._read
endrule
rule rule2;
   x = 32’hdeadbeef; // error
   Bit#(32) temp <= 32’hc001d00d; //error
endrule
```

Bluespec Functions

- Functions do not allow state changes
  - Can be defined within or outside module scope
  - No state change allowed, only performs computation and returns value
- Advanced topic: “Action function”
  - Can make state changes, but cannot return value
  - Not important for us right now

```
// Function example
function Int#(32) square(Int#(32) val);
   return val * val;
endfunction
rule rule1;
  $display( “%d”, square(12) );
endrule
```

Bluespec Types Basics

- ❑ Bluespec is a strongly typed language
  - ○ Many basic types: Bit#, Int#, UInt#, …
  - ○ For Bit#(32) a, b, Bit#(16) c, **a <= b+c** fails with type mismatch error
  - ○ a <= b + **zeroExtend**(c);
  - ○ Bit#(16) r = b + **truncate**(c);
- ❑ Supports many compound types
  - ○ Tuple, Vector, Maybe, Union, …

Tuples

- ❑ Types:
  - ○ Tuple2#(type t1, type t2)
  - ○ Tuple3#(type t1, type t2, type t3)
  - ○ up to Tuple8
- ❑ Values:
  - ○ tuple2(x, y),
tuple3(x, y, z), …
- ❑ Accessing an element:
  - ○ tpl_1(tuple2(x, y)) = x
  - ○ tpl_2(tuple3(x, y, z)) = y
  - ○ …

```
module …
  FIFO#(Tuple3#(Bit#(32),Bool,Int#(32))) tQ <- mkFIFO;
  rule rule1;
    tQ.enq(tuple3(32’hc00ld00d, False, 0));
  endrule
  rule rule2;
    tQ.deq;
    Tuple3#(Bit#(32),Bool,Int#(32)) v = tQ.first;
    $display( “%x”, tpl_1(v) );
  endrule
endmodule
```

Vector

- Type: `Vector#(numeric type size, type data_type)`
- Values:
  - newVector()
  - replicate(val)
- Functions:
  - Access an element: []
  - Rotate functions
  - Advanced functions: zip, map, fold, …
- Provided as Bluespec library
  - Must have ‘import Vector::*;’ in BSV file

Vector Example

```
import Vector::*; // required!

module …
  Reg#(Vector#(8,Int#(32))) x <- mkReg(newVector());
  Reg#(Vector#(8,Int#(32))) y <- mkReg(replicate(1));
  Reg#(Vector#(2, Vector#(8, Bit#(32)))) zz <- mkReg(replicate(replicate(0));
  Reg#(Bit#(3)) r <- mkReg(0);

  rule rule1;
    $display( “%d”, x[0] );
    x[r] <= zz[0][r];
    r <= r + 1; // wraps around
  endrule
endmodule
```

Array of Values Using Reg and Vector

- Option 1: Register of Vectors
  - Reg#(Vector#(32, Bit#(32))) rfile;
  - rfile <- mkReg(**replicate**(0)); **// replicate creates a vector from values**
- Option 2: Vector of Registers
  - Vector#(32, Reg#(Bit#(32))) rfile;
  - rfile <- **replicateM**(mkReg(0)); **// replicateM creates vector from modules**
- Each has its own advantages and disadvantages

Partial Writes

- Reg#(Bit#(8)) r;
  - r[0] <= 0 counts as a read and write to the entire register r
  - Bit#(8) r_new = r; r_new[0] = 0; r <= r_new
- Reg#(Vector#(8, Bit#(1))) r
  - Same problem, r[0] <= 0 counts as a read and write to the entire register
  - r[0] <= 0; r[1] <= 1 counts as two writes to register r – **write conflict error**
- Vector#(8,Reg#(Bit#(1))) r
  - r is 8 different registers
  - r[0] <= 0 is only a write to register r[0]
  - r[0] <= 0 ; r[1] <= 1 does not cause a write conflict error

Automatic Type Deduction Using “let”

- ”let” statement enables users to declare a variable without providing an exact type
  - Compiler deduces the type using other information (e.g., assigned value)
  - Like “auto” in C++11, still statically typed

```
module …
  Reg#(Int#(32)) x <- mkReg(0);

  rule rule1;
    let value = x+1;
    Int#(16) value2 = 0;
    if (value+value2 < 0) $write( “yay” ); // error! Int#(32), Int#(16) mismatch
  endrule
endmodule
```

value is Int#(32)

State

```
module mkGCD (GDCIfc);
  Reg#(Bit#(32)) x <- mkReg(0);
  Reg#(Bit#(32)) y <- mkReg(0);
  FIFOF#(Bit#(32)) outQ <- mkSizedFIFOF(2);
```

Rules
(Behavior)

```
  rule step1 ((x > y) && (y != 0));
    x <= y; y <= x;
  endrule
  rule step2 (( x <= y) && (y != 0));
    y <= y-x;
    if ( y-x == 0 ) begin
      outQ.enq(x);
    end
  endrule
```

Interface
(Behavior)

```
  method Action start(Bit#(32) a, Bit#(32) b) if (y==0);
    x <= a; y <= b;
  endmethod
  method ActionValue#(Bit#(32)) result();
    outQ.deq;
    return outQ.first;
  endmethod
endmodule
```

More topics include…

- Types, typeclasses
- Polymorphism
- Rule Scheduling
- Static elaboration
- …