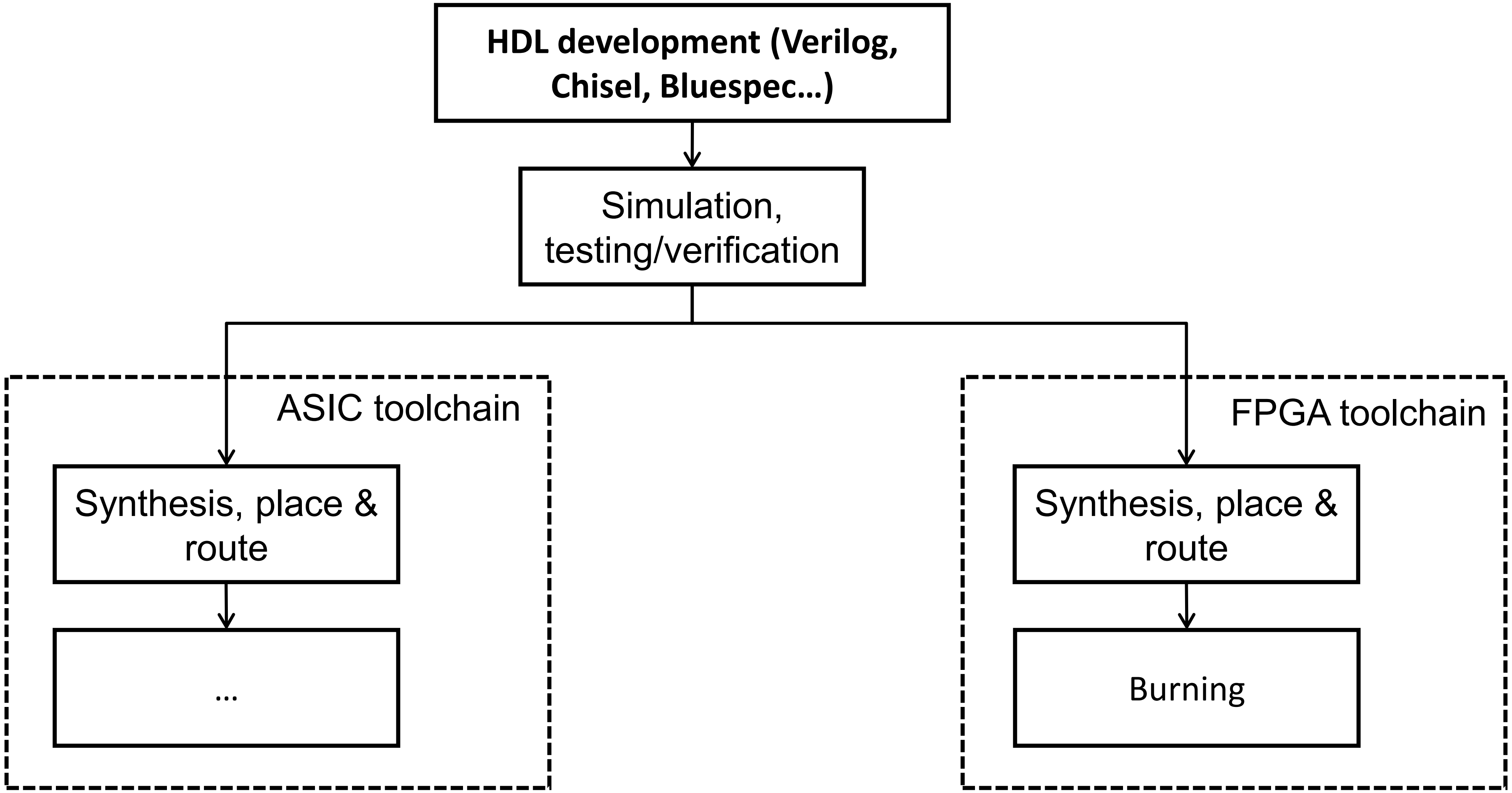
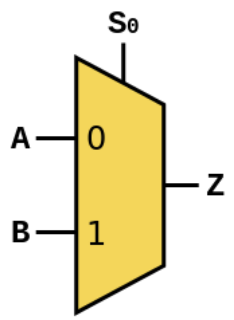
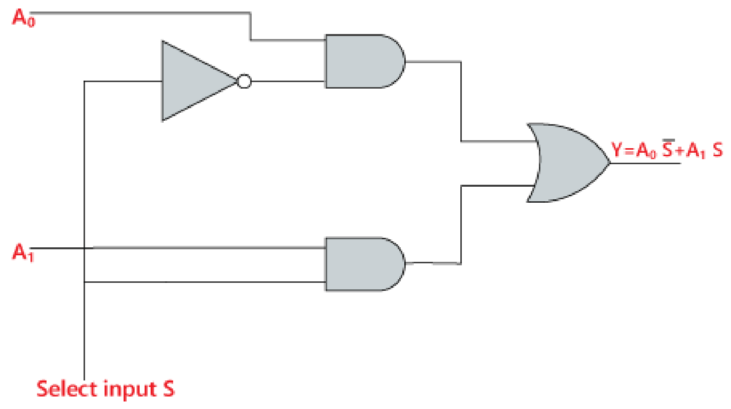
Workflow of Hardware Design & Implementation



What is an HDL and what does it do?

- **Hardware is a combination of basic electronic components**
- **Hardware Description Language (HDL)**
  - HDL provides definitions for basic electronic components (wires, logic gates, registers, etc.)
  - HDL describes
    - Connection of electronic components
  - HDL does not describe：
    - Specific placement and layout
    - How the components are implemented in reality
- **Hardware is not a "sequence of instructions"**





"Mux"

Evaluation Metrics of Hardware

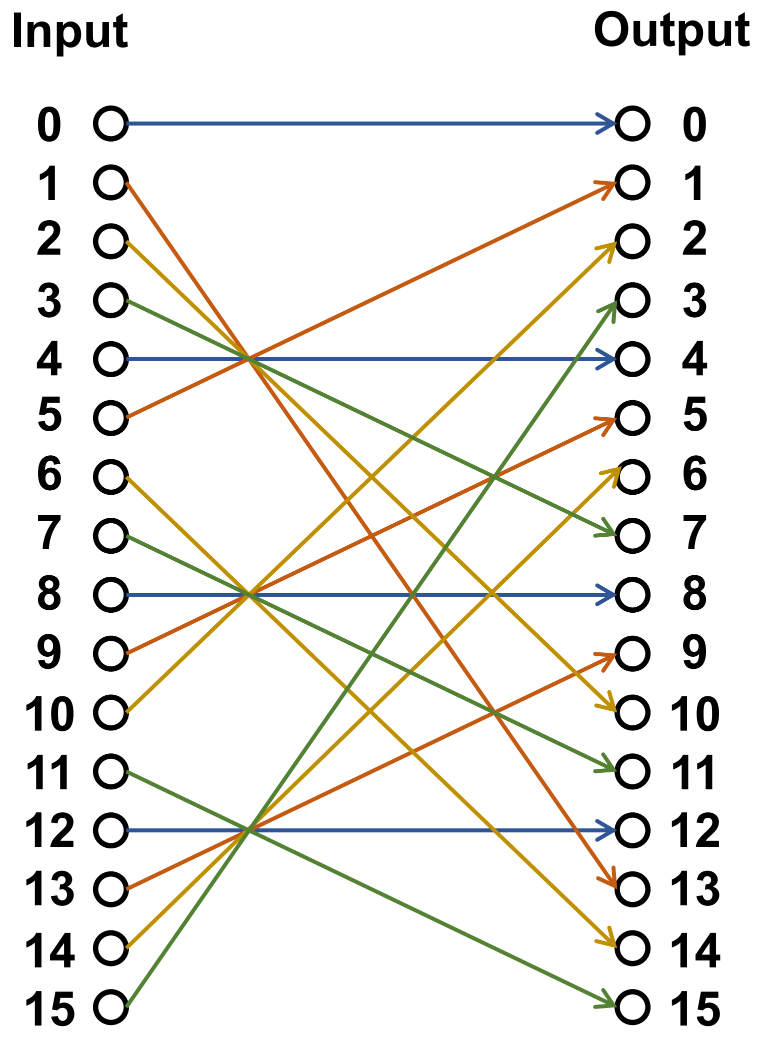
- **PPA evaluation (P-Performance, P-Power, A-Area)**
- **Performance**
  - Comprehensive metrics such as **throughput/latency/operations per second**
  - Usually related to the hardware's operating **frequency**
- **Power**
- **Area**
  - Physical resources occupied by the hardware
  - Manifests as area/gate count in ASIC, and resource utilisation in FPGA

A Bit of “Hardware Thoughts” – AES ShiftRows

0	4	8	C
1	5	9	D
2	6	A	E
3	7	B	F

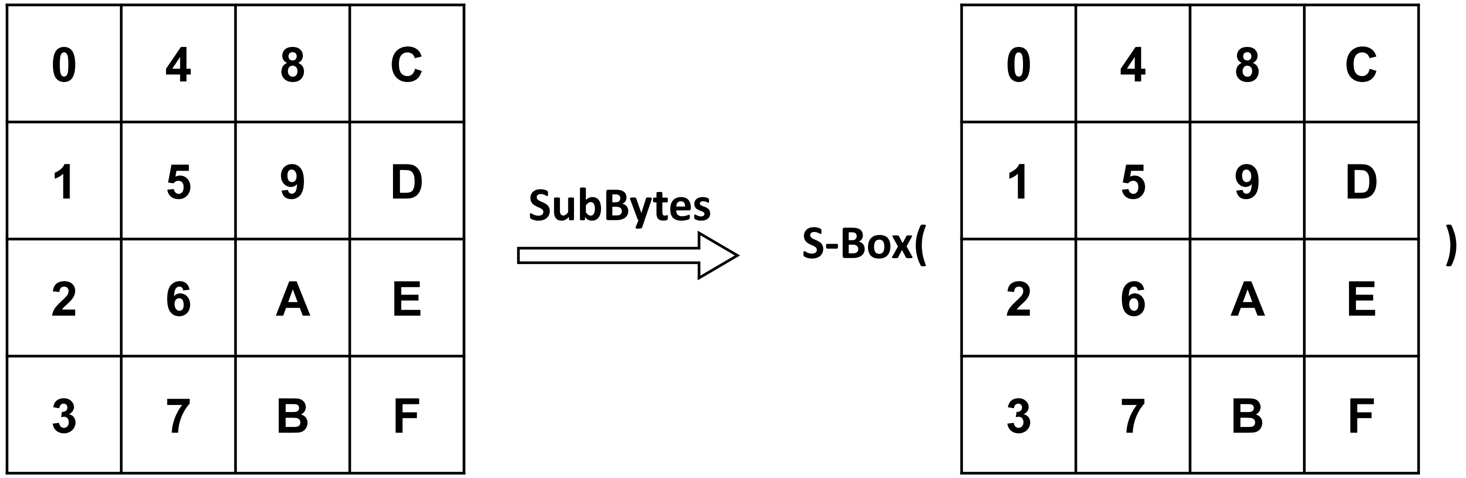
$\xrightarrow{\text{ShiftRows}}$

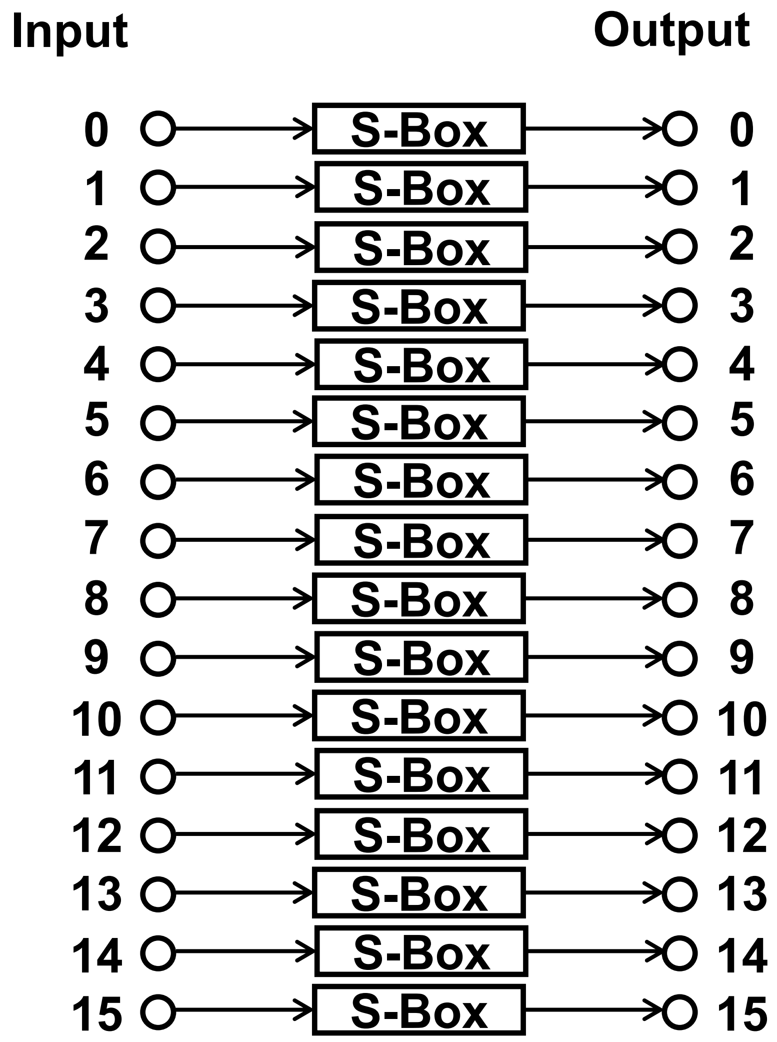
0	4	8	C
5	9	D	1
A	E	2	6
F	3	7	B



```
for(int i = 0; i < 16; i++) {
  if(...) {
    // ...
  } else if(...) {
    // ...
  } // else if...
}
```

A Bit of “Hardware Thoughts”– AES SubBytes





```
char s_box[256] = {...};
for(int i = 0; i < 16; i++) {
  state[i] = s_box[state[i]];
}
```

Chisel Is a Modern HDL

- **Chisel is an EDSL in Scala. It provides**
  - Register transfer layer (RTL) based programming model (same as Verilog/VHDL)
  - Better abstraction and flexibility compared to Verilog/VHDL
  - Library for common circuits (FIFO, Arbiter, Decoder, …)
  - Better testing framework
  - Compilation down to .v/.sv

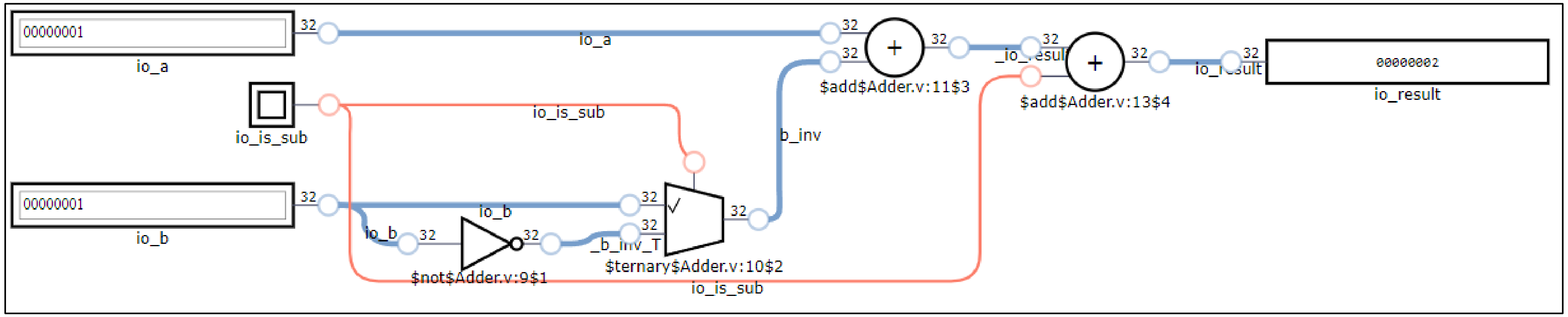
First Chisel Module

```
class Adder(width: Int) extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(width.W))
    val b = Input(UInt(width.W))
    val is_sub = Input(Bool())
    val result = Output(UInt(width.W))
  })
  val b_inv = Mux(io.is_sub, ~io.b, io.b)
  io.result := io.a + b_inv + io.is_sub
}
```

Chisel code

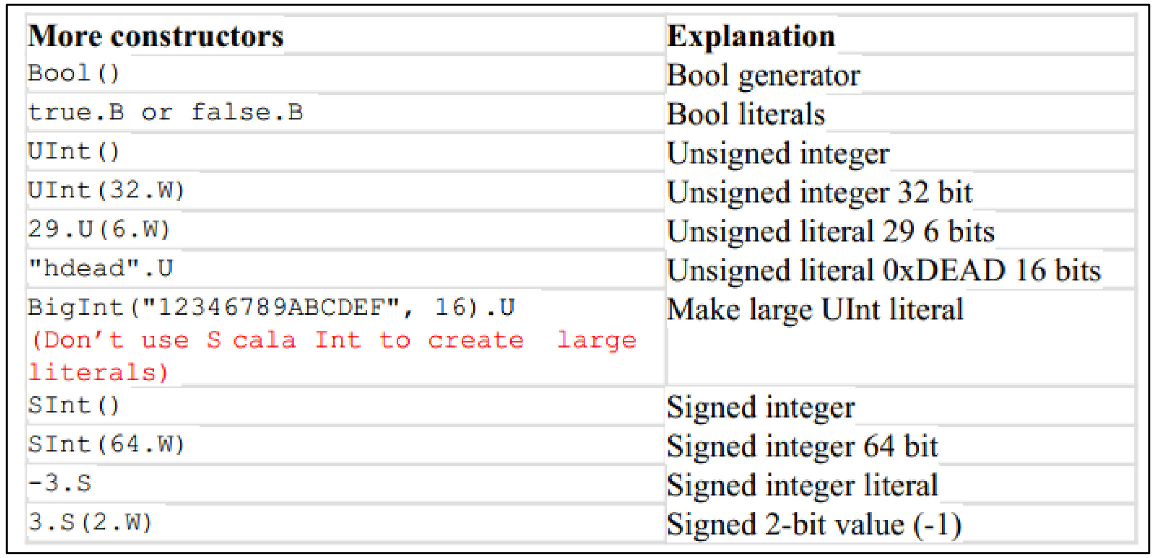
```
module Adder(
  input         clock,
  input         reset,
  input  [31:0] io_a,
  input  [31:0] io_b,
  input         io_is_sub,
  output [31:0] io_result
);
  wire [31:0] _b_inv_T = ~io_b;
  wire [31:0] b_inv = io_is_sub ? _b_inv_T : io_b;
  wire [31:0] _io_result_T_1 = io_a + b_inv;
  wire [31:0] _GEN_0 = {{31'd0}, io_is_sub};
  assign io_result = _io_result_T_1 + _GEN_0;
endmodule
```

Verilog code

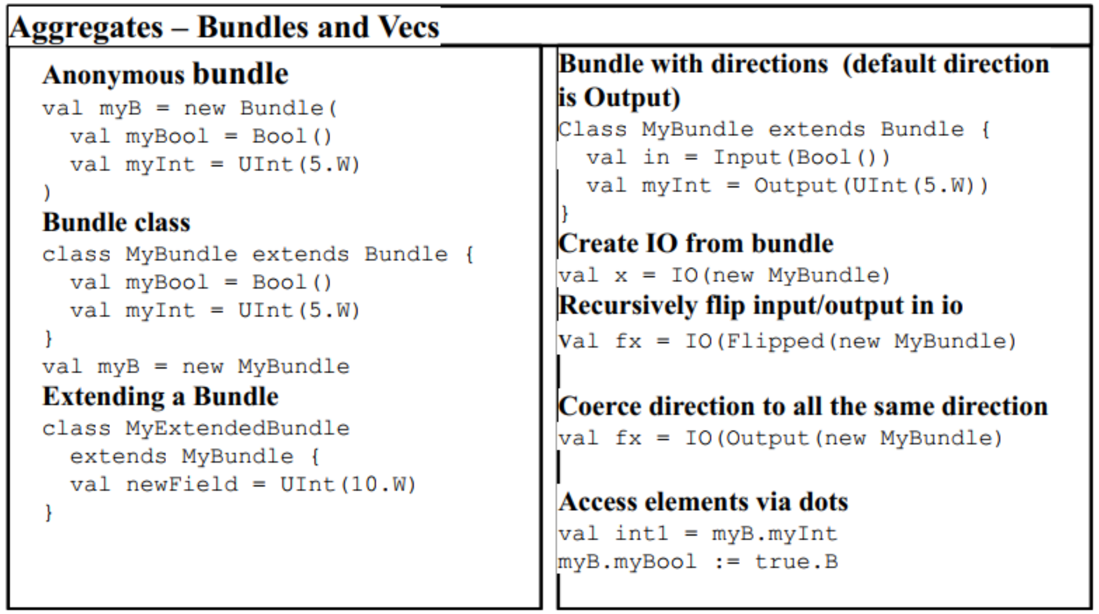


Online circuit visualization: http://digitaljs.tilk.eu

Basic Data Types and Operations in Chisel



More constructors	Explanation
`Bool()`	Bool generator
`true.B or false.B`	Bool literals
`UInt()`	Unsigned integer
`UInt(32.W)`	Unsigned integer 32 bit
`29.U(6.W)`	Unsigned literal 29 6 bits
`"hdead".U`	Unsigned literal 0xDEAD 16 bits
`BigInt("12346789ABCDEF", 16).U` (Don't use S cala Int to create large literals)	Make large UInt literal
`SInt()`	Signed integer
`SInt(64.W)`	Signed integer 64 bit
`-3.S`	Signed integer literal
`3.S(2.W)`	Signed 2-bit value (-1)



Aggregates – Bundles and Vecs

Anonymous bundle

```
val myB = new Bundle(
  val myBool = Bool()
  val myInt = UInt(5.W)
)
```

Bundle class

```
class MyBundle extends Bundle {
  val myBool = Bool()
  val myInt = UInt(5.W)
}
val myB = new MyBundle
```

Extending a Bundle

```
class MyExtendedBundle
  extends MyBundle {
  val newField = UInt(10.W)
}
```

Bundle with directions (default direction is Output)

```
Class MyBundle extends Bundle {
  val in = Input(Bool())
  val myInt = Output(UInt(5.W))
}
```

Create IO from bundle

```
val x = IO(new MyBundle)
```

Recursively flip input/output in io

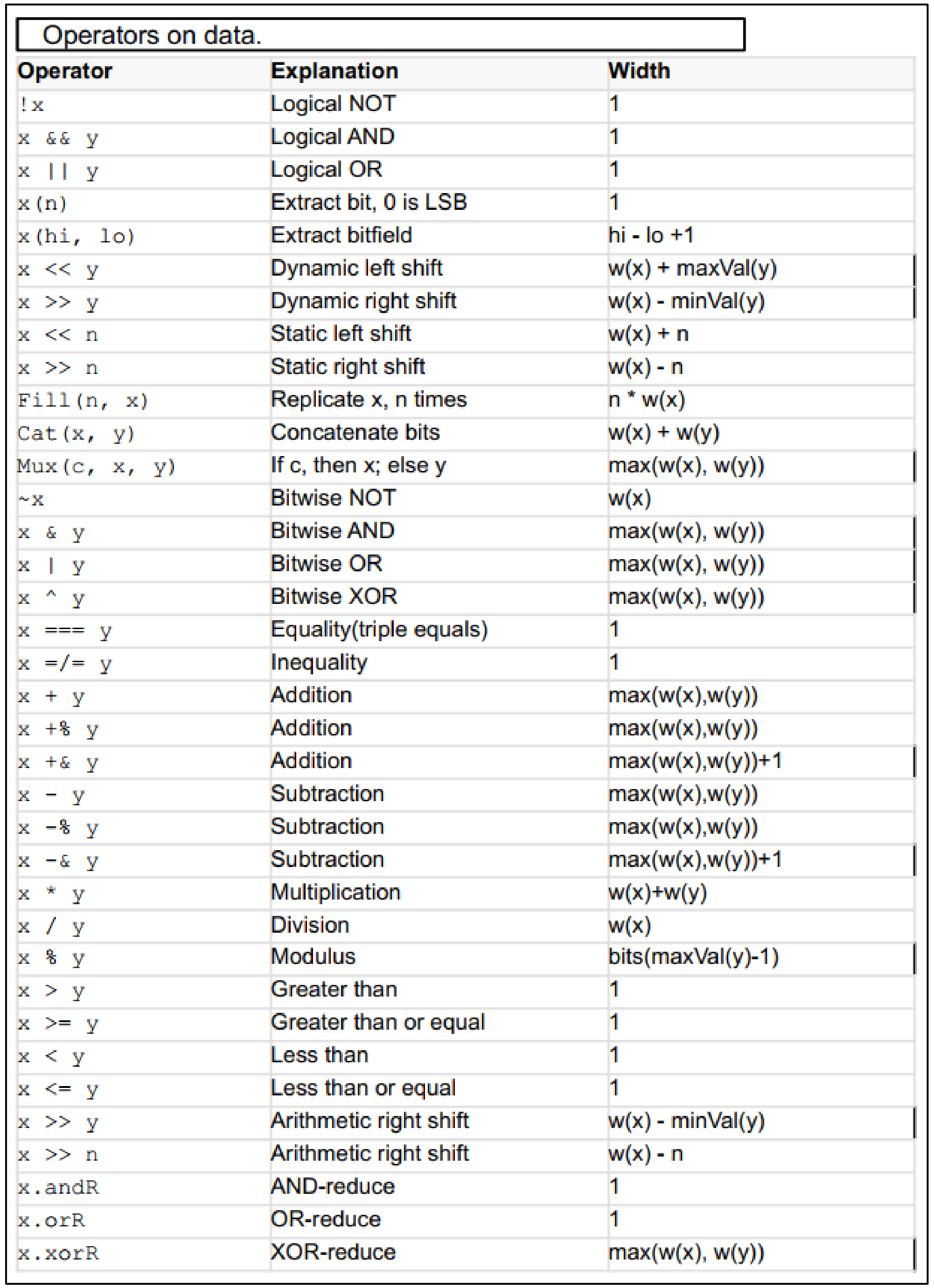
```
Val fx = IO(Flipped(new MyBundle)
```

Coerce direction to all the same direction

```
val fx = IO(Output(new MyBundle)
```

Access elements via dots

```
val int1 = myB.myInt
myB.myBool := true.B
```



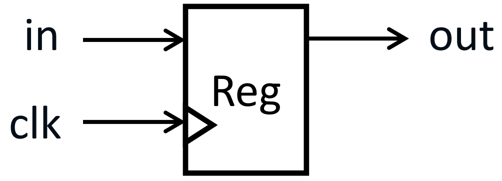
Operators on data.

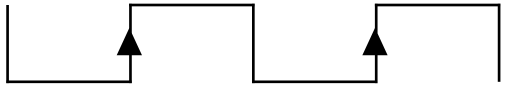
Operator	Explanation	Width
`!x`	Logical NOT	1
`x && y`	Logical AND	1
`x || y`	Logical OR	1
`x(n)`	Extract bit, 0 is LSB	1
`x(hi, lo)`	Extract bitfield	hi - lo +1
`x << y`	Dynamic left shift	w(x) + maxVal(y)
`x >> y`	Dynamic right shift	w(x) - minVal(y)
`x << n`	Static left shift	w(x) + n
`x >> n`	Static right shift	w(x) - n
`Fill(n, x)`	Replicate x, n times	n * w(x)
`Cat(x, y)`	Concatenate bits	w(x) + w(y)
`Mux(c, x, y)`	If c, then x; else y	max(w(x), w(y))
`~x`	Bitwise NOT	w(x)
`x & y`	Bitwise AND	max(w(x), w(y))
`x | y`	Bitwise OR	max(w(x), w(y))
`x ^ y`	Bitwise XOR	max(w(x), w(y))
`x === y`	Equality(triple equals)	1
`x =/= y`	Inequality	1
`x + y`	Addition	max(w(x),w(y))
`x +% y`	Addition	max(w(x),w(y))
`x +& y`	Addition	max(w(x),w(y))+1
`x - y`	Subtraction	max(w(x),w(y))
`x -% y`	Subtraction	max(w(x),w(y))
`x -& y`	Subtraction	max(w(x),w(y))+1
`x * y`	Multiplication	w(x)+w(y)
`x / y`	Division	w(x)
`x % y`	Modulus	bits(maxVal(y)-1)
`x > y`	Greater than	1
`x >= y`	Greater than or equal	1
`x < y`	Less than	1
`x <= y`	Less than or equal	1
`x >> y`	Arithmetic right shift	w(x) - minVal(y)
`x >> n`	Arithmetic right shift	w(x) - n
`x.andR`	AND-reduce	1
`x.orR`	OR-reduce	1
`x.xorR`	XOR-reduce	max(w(x), w(y))

Chisel cheatsheet: https://github.com/freechipsproject/chisel-cheatsheet

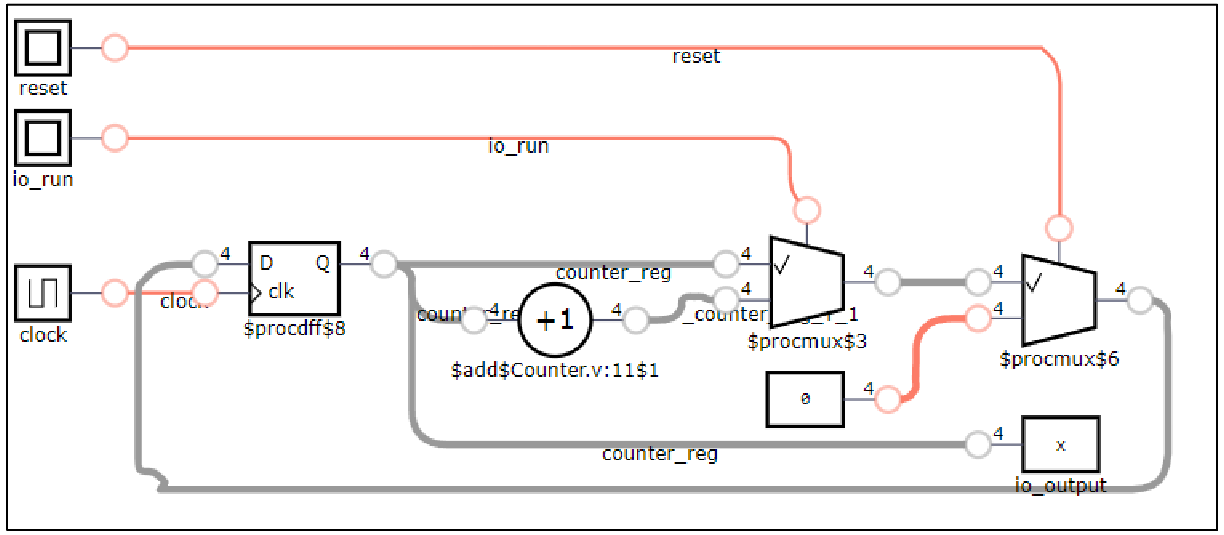
Basic Components in Chisel

- **Wire + Gates**
  - Basic elements in **combinational** circuits
  - “0 delay”
- **Registers**
  - Store **states**
  - Introduce **sequential** behavior
  - Generally update on the rising edge of the clock:



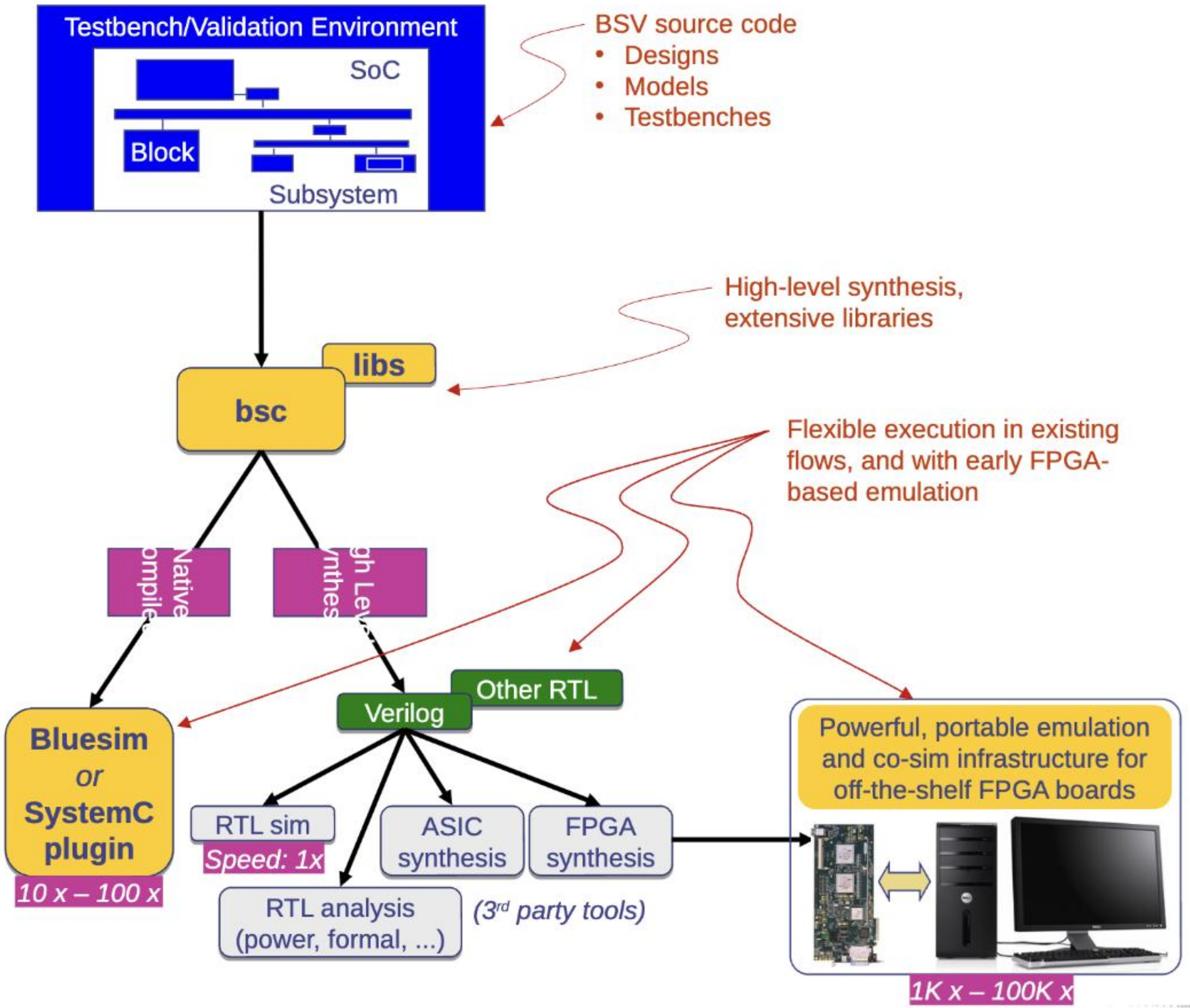


```
class Counter(width: Int) extends Module {
  val io = IO(new Bundle {
    val run = Input(Bool())
    val output = Output(UInt(width.W))
  })
  val counter_reg = RegInit(0.U(width.W))
  when(io.run) {
    counter_reg := counter_reg + 1.U
  }
  io.output := counter_reg
}
```



Online circuit visualization: http://digitaljs.tilk.eu bluespec™

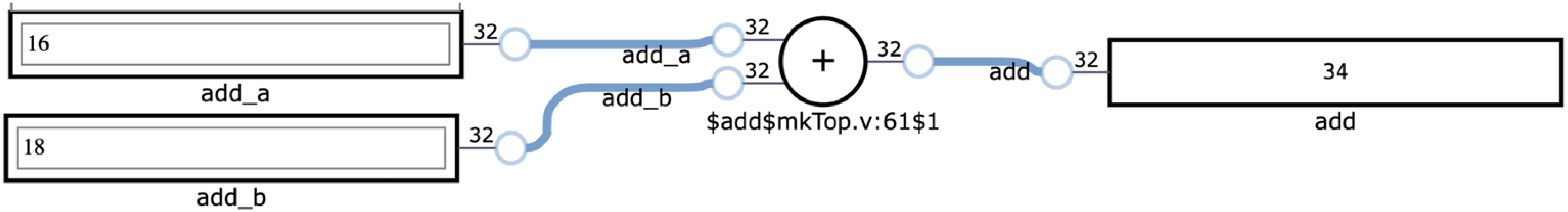
- **Two front end:** Bluespec Verilog (**BSV**) and Bluespec Haskell (**BH**).
- **Rule-Based Design:** Safe, modular behavior through **rules.**
- **Higher Abstraction:** Focus on *what* hardware does, not *how*


Testbench/Validation Environment
SoC
Block
Subsystem
BSV source code
• Designs
• Models
• Testbenches
High-level synthesis,
extensive libraries
libs
bsc
Flexible execution in existing
flows, and with early FPGA-
based emulation
Native
Other RTL
Verilog
Bluesim
or
SystemC
plugin
10 x – 100 x
RTL sim
Speed: 1x
ASIC
synthesis
FPGA
synthesis
RTL analysis
(power, formal, ...)
(3rd party tools)
Powerful, portable emulation
and co-sim infrastructure for
off-the-shelf FPGA boards
1K x – 100K x

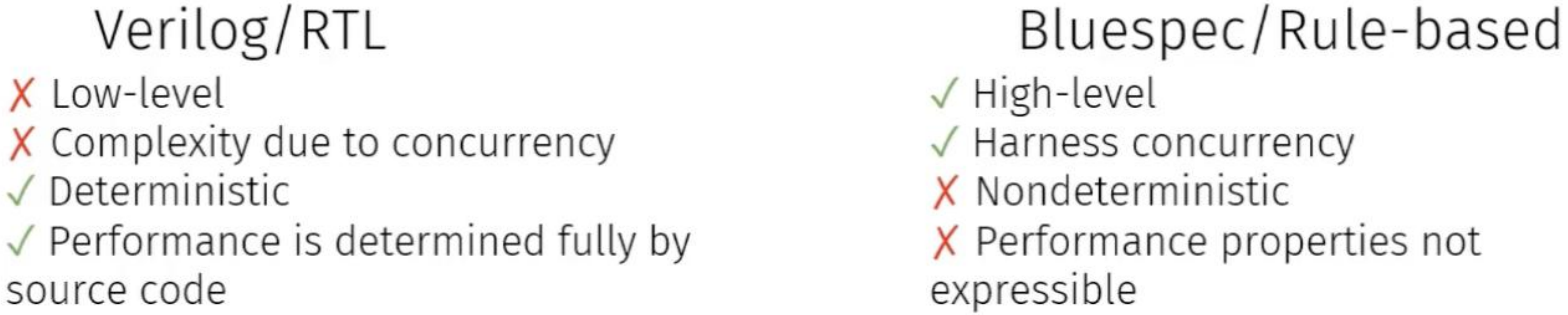
Example: Adder

```
interface Adder_ifc;
 method ActionValue#(Bit#(32))
   add(Bit#(32) a, Bit#(32) b);
endinterface
```

```
(* synthesize *)
module mkTop(Adder_ifc);
 function Bit#(32) do_add(Bit#(32) x,
                          Bit#(32) y);
   return x + y;
 endfunction
 method ActionValue#(Bit#(32))
   add(Bit#(32) a, Bit#(32) b);
     let out = do_add(a, b);
     return out;
 endmethod
```



Semantic difference between Chisel vs Bluespec



Verilog/RTL

✗ Low-level
✗ Complexity due to concurrency
✓ Deterministic
✓ Performance is determined fully by source code

Bluespec/Rule-based

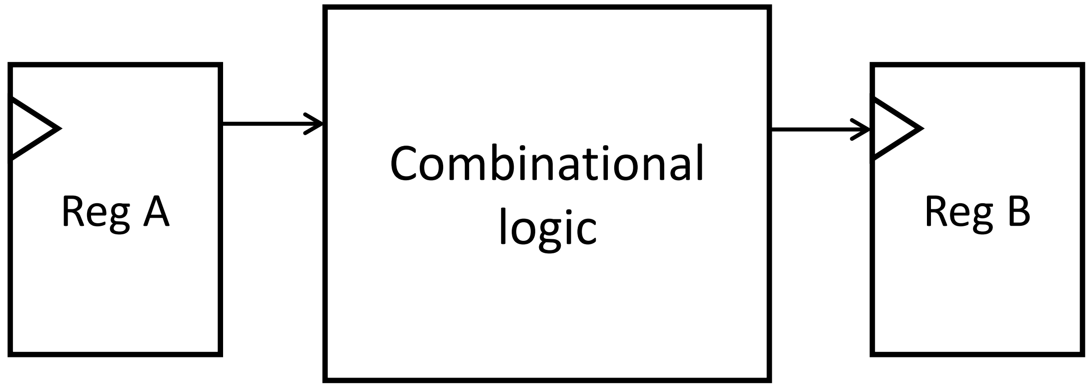
✓ High-level
✓ Harness concurrency
✗ Nondeterministic
✗ Performance properties not expressible

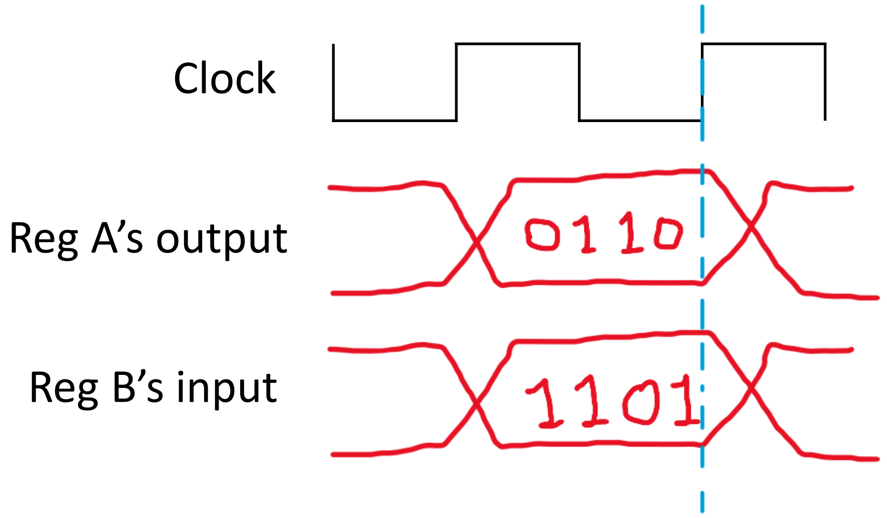
Pipelining!

If combinational circuit has 0 delay, can we use it to realize any complex circuit?

Difficulty in reality:

- Combinational logic has 0 delay in **RTL**, but does have delay in **reality**
- The delay is related to **depth of gates** and **physical distance** between components
- When a signal cannot reach the next register from one register **within one clock cycle**, the circuit's behavior becomes unstable



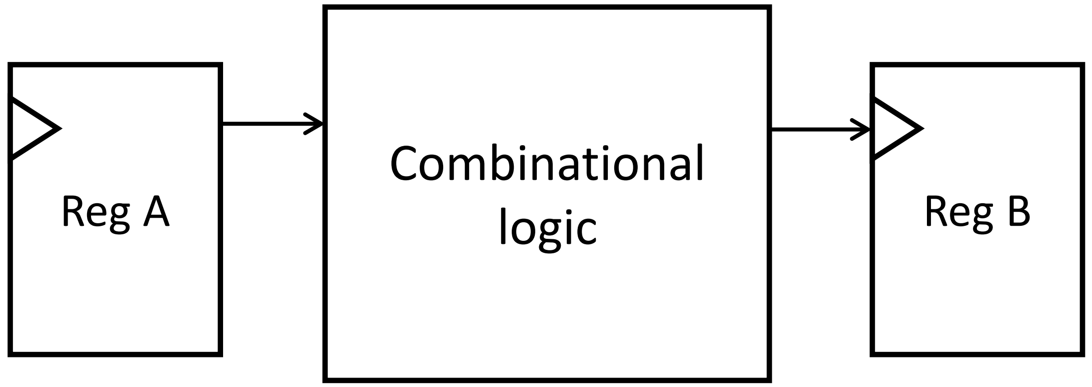


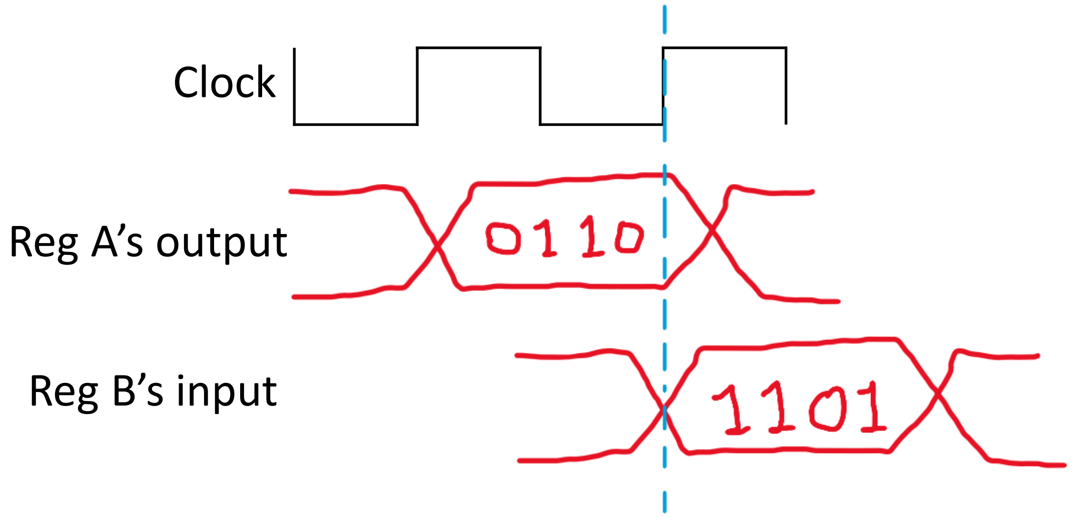
Pipelining!

If combinational circuit has 0 delay, can we use it to realize any complex circuit?

Difficulty in reality:

- Combinational logic has 0 delay in **RTL**, but does have delay in **reality**
- The delay is related to **depth of gates** and **physical distance** between components
- When a signal cannot reach the next register from one register **within one clock cycle**, the circuit's behavior becomes unstable

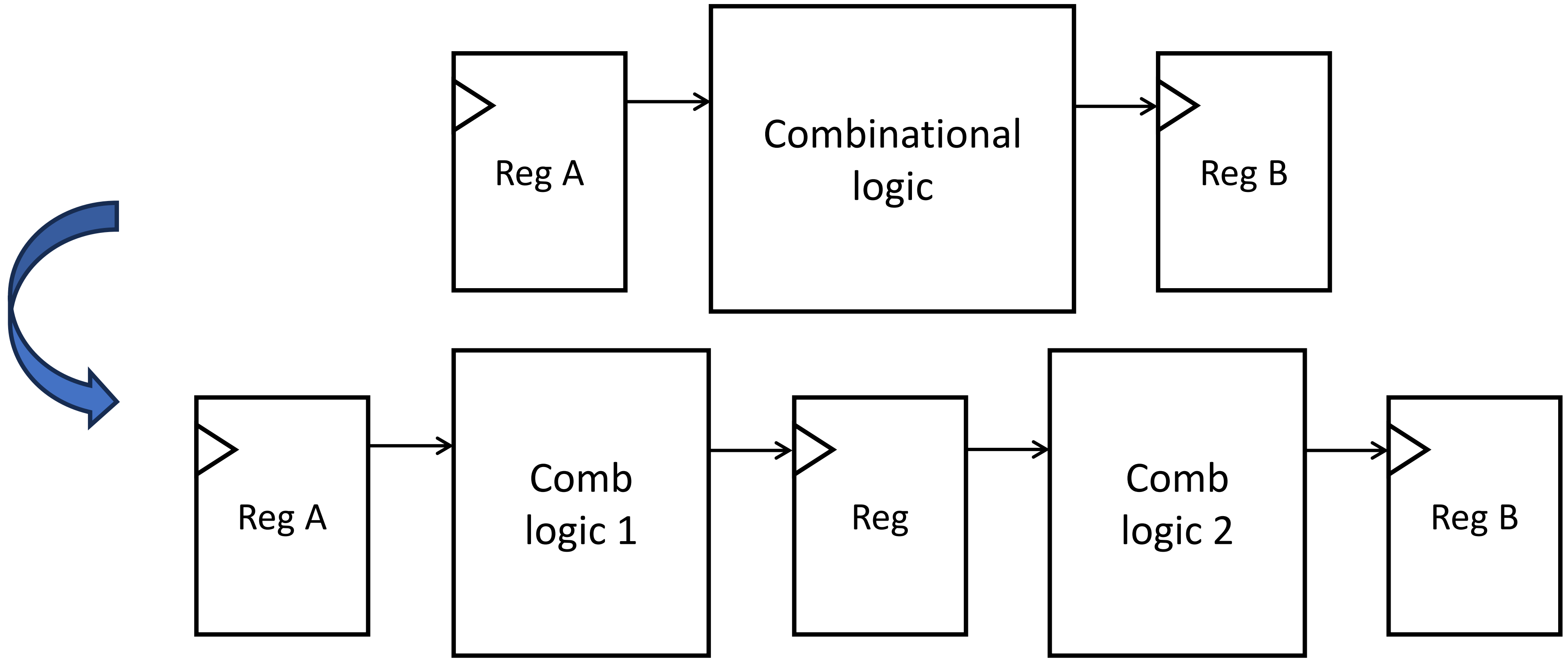




Pipelining!

When a circuit cannot converge at a given clock frequency, solutions are:

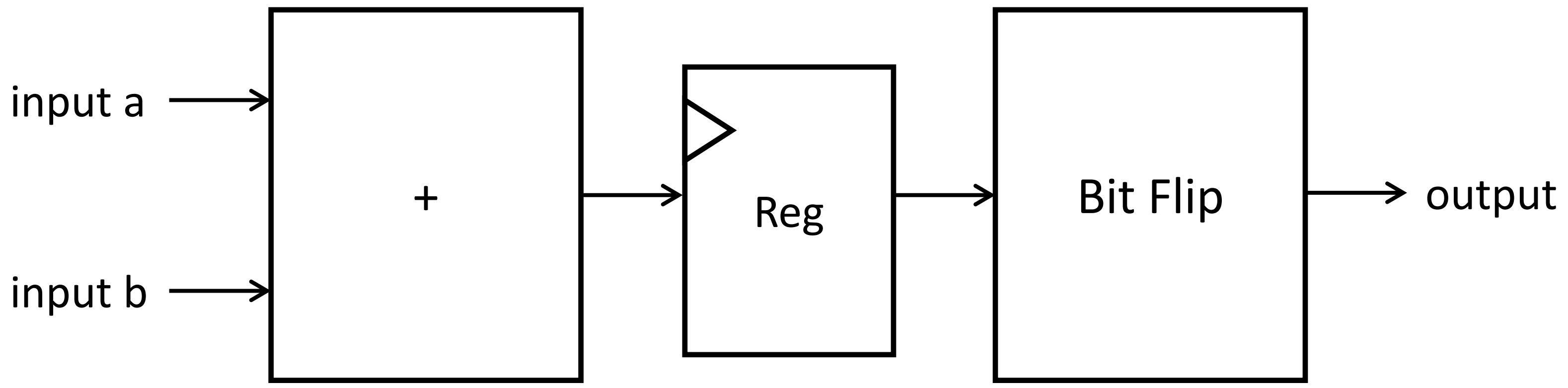
- Lowering the clock frequency
- Inserting registers into the original combinational logic to "pipeline" the circuit



Live Coding!

Live Coding Example of a Pipeline

~(a+b)


input a
input b
+
Reg
Bit Flip
output