Chipper Bootcamp

Jonathan Bachrach

EECS UC Berkeley

July 9, 2015

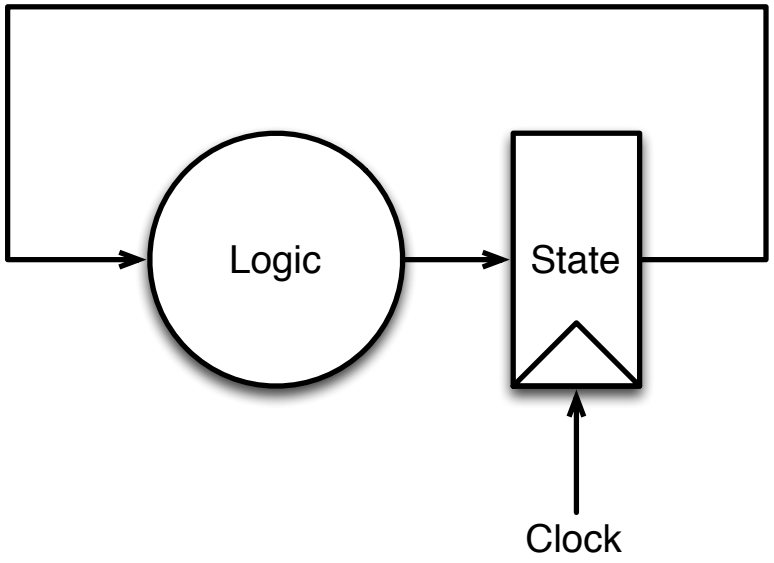
Goals for Bootcamp

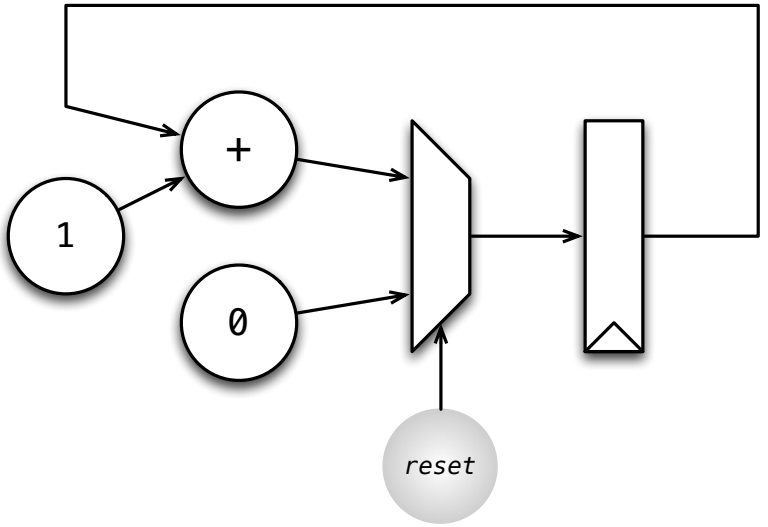
- introduction to reconfigurable computing
- get you started with Chipper
- get a basic working knowledge of Chipper
- learn how to think in Chipper
- know where to get more information

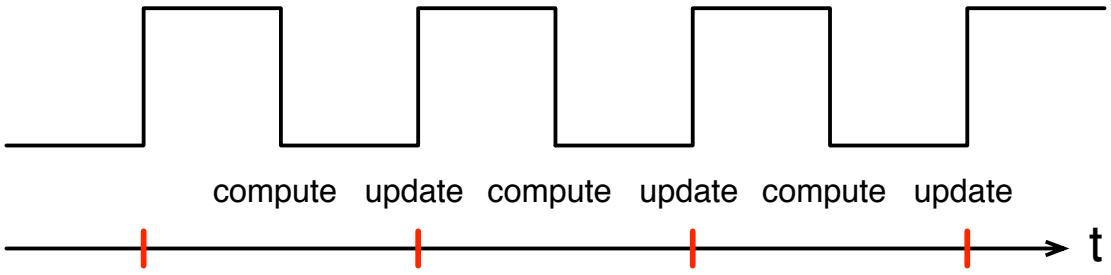
Register Transfer Level = RTL

- logic and state
- state machines

- e.g., adders and registers
- e.g., counter example:

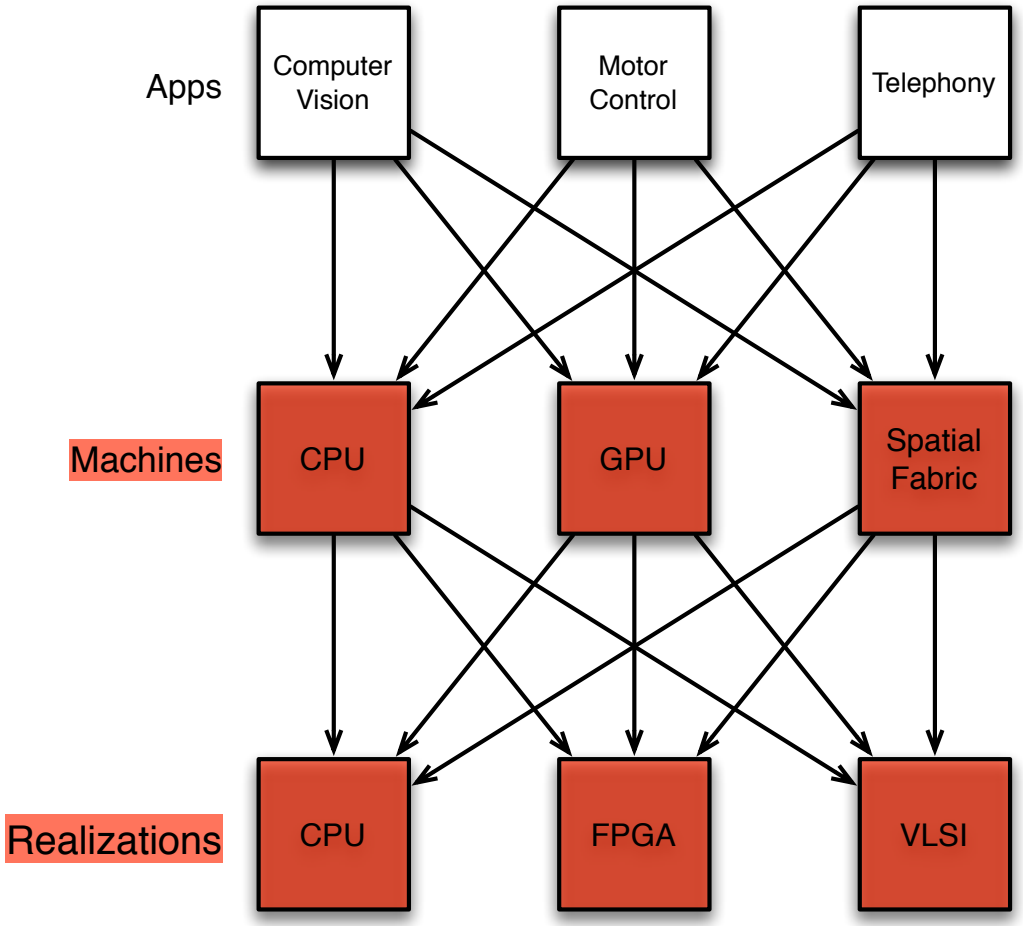






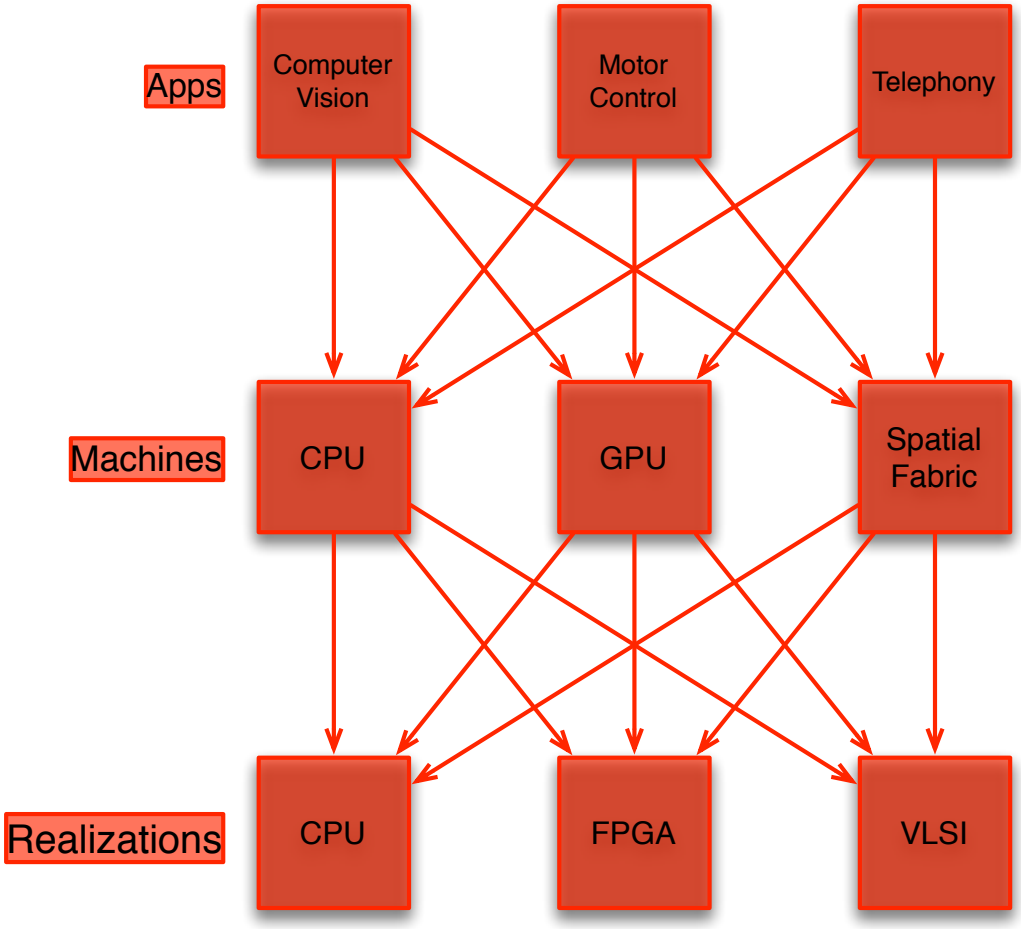
synchronous clock

HW Design Context



- what's best machine?
- how to most efficiently implement machine?

Three Hard HLS* Tasks



would love to go app -> gates but ... it's really hard

* HLS = High Level Synthesis

Opportunity Is Huge

- use 10% of energy
- yearly power use of phone == refridgerator
- cost of computing infrastructure is often < yearly energy bill
- only starting to bring world online

- energy is starting to dominate everything!!!
- digital designs yield 100-1000x win in efficiency

Hardware Designers

- mostly EE backgrounds
- very little PL experience
- efficiency is everything!

Dire Hardware Design Situation

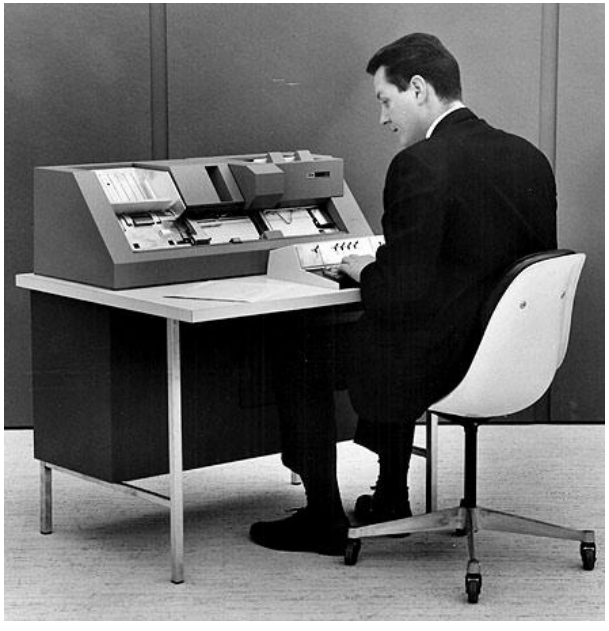
- slow hardware design
  - 1980's style languages and baroque tool chains with ad hoc scripts
  - manual optimization obscuring designs
  - minimal compile-time and run-time errors
  - army of people in both CAD tools and design – costs $10Ms
- slow and expensive to synthesize
  - takes order days
  - not robust and so largely manual process
  - proprietary tools – cost > $1M / year / seat
- slow testing and evaluation
  - runs 200M x slower than code runs in production
  - army of verification people – costs $10Ms
- slow and expensive fabrication
  - very labor intensive – costs $1Ms

- design costs dominate
- very few ASIC designs and chip startups

Imagine If In Software Design

- had to program in assembly language (or fortran)
- compilation took hours or days
- got minimal compile or run time warnings or errors
- finding bugs took hours or weeks
- burning “CD”s cost $1Ms

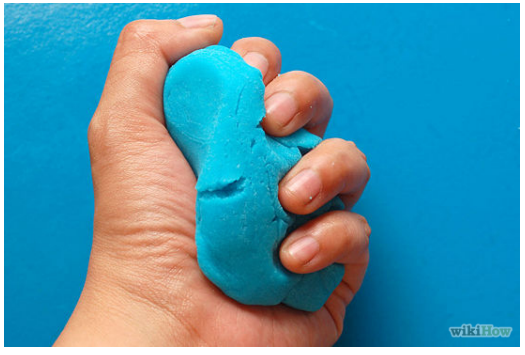
- Software startups would be rare
- CS enrollment would go way down!



Imagine If In Hardware Design

- could build a “chip” in an hour (or a day) *
- build and fab tools were low cost (or free) *
- could create reusable and abstract modules
- had large library of standard components to choose from
- could create chips with small teams

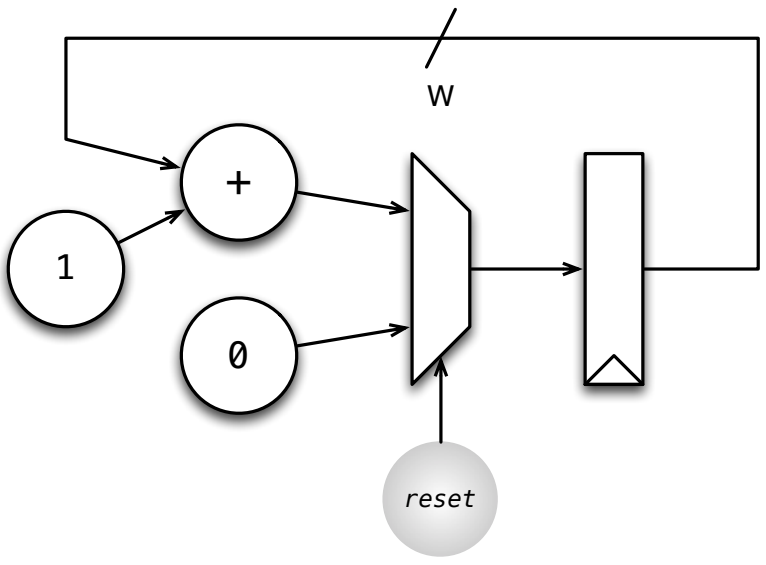
- Hardware startups would be common
- EE enrollment might go way up!



** FPGAs almost fit the bill but tools are painful to use and painfully slow*

1. write verilog design structurally – literal
2. verilog generate command – limited
3. write perl script that writes verilog – awkward

```
module counter (clk, reset);
  input clk;
  input reset;
  parameter W = 8;
  reg [W-1:0] cnt;
  always @ (posedge clk)
  begin
    if (reset)
      cnt = 0
    else
      cnt = cnt + 1
  end
endmodule
```

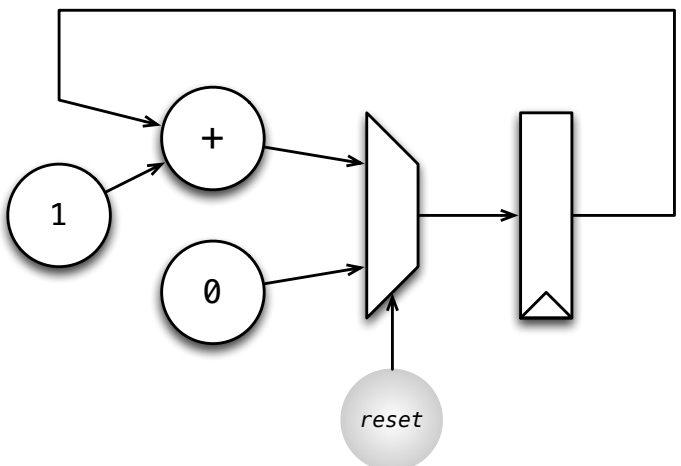


- Classes are provided for circuit components:
  - `Register(name)`
  - `Adder()`
  - `Multiplexor()`
  - `Wire(name)`
  - `Constant(value)`

  and `new` used to construct components and `connect` used to wire them together:
  - `new Register(name)`
  - ...
  - `connect(input, output)`

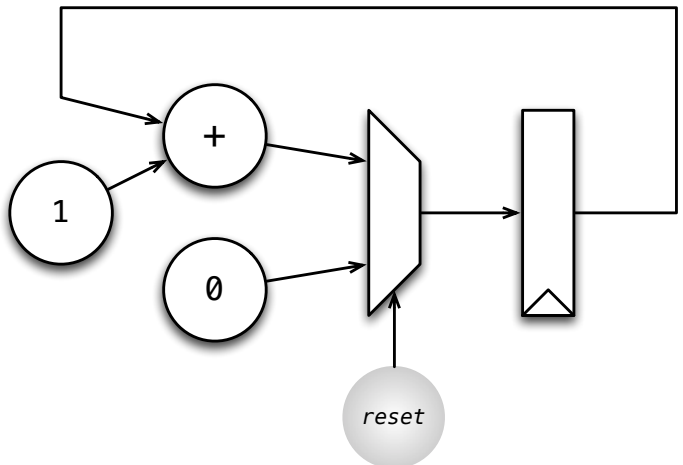
What if Chisel was a Stanza Library?


```
defn main () :
  ;; Create Components
  val reset       = Wire("reset")
  val counter     = Register("counter")
  val adder       = Adder()
  val multiplexor = Multiplexor()
  val one         = UInt(1)
  val zero        = UInt(0)

  ;; Connect Components
  connect(multiplexor.choice, reset)
  connect(multiplexor.in_a, zero.out)
  connect(multiplexor.in_b, adder.out)
  connect(counter.in, multiplexor.out)
  connect(adder.in_a, counter.out)
  connect(adder.in_b, one.out)

  ;; Produce Verilog
  generate_verilog(counter)
```



What if Chisel was a Stanza Library?

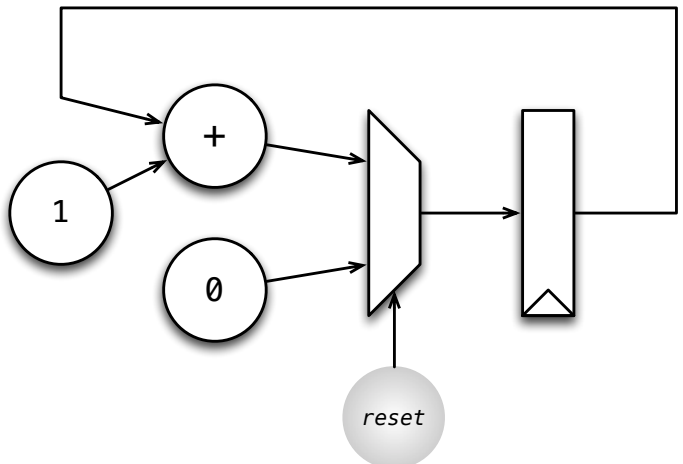

```
defn main () :
  ;; Create Components
  val reset       = Wire("reset")
  val counter     = Register("counter")
  val adder       = Adder()
  val multiplexor = Multiplexor()
  val one         = UInt(1)
  val zero        = UInt(0)

  ;; Connect Components
  connect(multiplexor.choice, reset)
  connect(multiplexor.in_a, zero.out)
  connect(multiplexor.in_b, adder.out)
  connect(counter.in, multiplexor.out)
  connect(adder.in_a, counter.out)
  connect(adder.in_b, one.out)

  ;; Produce Verilog
  generate_verilog(counter)
```



- using Stanza to programmatically generate hardware
- can use full power of Stanza (loops, arrays, conditionals, ...)
- zero cost abstraction

What if Chisel was a Stanza Library?


```
defn main () :
  ;; Create Components
  val reset       = Wire("reset")
  val counter     = Register("counter")
  val adder       = Adder()
  val multiplexor = Multiplexor()
  val one         = UInt(1)
  val zero        = UInt(0)

  ;; Connect Components
  connect(multiplexor.choice, reset)
  connect(multiplexor.in_a, zero.out)
  connect(multiplexor.in_b, adder.out)
  connect(counter.in, multiplexor.out)
  connect(adder.in_a, counter.out)
  connect(adder.in_b, one.out)

  ;; Produce Verilog
  generate_verilog(counter)
```



- but Stanza is pretty Verbose, how can we do better?

Functional Composition of Adder


```
defn main () :
  ;; Create Components
  val reset       = Wire("reset")
  val counter     = Register("counter")
  val adder = Adder()
  val multiplexor = Multiplexor()
  val one         = UInt(1)
  val zero        = UInt(0)

  ;; Connect Components
  connect(multiplexor.choice, reset)
  connect(multiplexor.in_a, zero.out)
  connect(multiplexor.in_b, adder.out)
  connect(counter.in, multiplexor.out)
  connect(adder.in_a, counter.out)
  connect(adder.in_b, one.out)

  ;; Produce Verilog
  generate_verilog(counter)
```

```
defn main () :
  ;; Create Components
  val reset       = Wire("reset")
  val counter     = Register("counter")
  val multiplexor = Multiplexor()
  val one         = UInt(1)
  val zero        = UInt(0)

  ;; Connect Components
  connect(multiplexor.choice, reset)
  connect(multiplexor.in_a, zero.out)
  connect(multiplexor.in_b,
          make_adder(one.out, counter.out))
  connect(counter.in, multiplexor.out)

  ;; Produce Verilog
  generate_verilog(counter)
```

functional programming

Functional Composition of Multiplexor


```
defn main () :
  ;; Create Components
  val reset       = Wire("reset")
  val counter     = Register("counter")
  val multiplexor = Multiplexor()
  val one         = UInt(1)
  val zero        = UInt(0)

  ;; Connect Components
  connect(multiplexor.choice, reset)
  connect(multiplexor.in_a, zero.out)
  connect(multiplexor.in_b,
          make_adder(one.out, counter.out))
  connect(counter.in, multiplexor.out)

  ;; Produce Verilog
  generate_verilog(counter)
```

```
defn main () :
  ;; Create Components
  val reset   = Wire("reset")
  val counter = Register("counter")
  val one     = UInt(1)
  val zero    = UInt(0)

  ;; Connect Components
  connect(counter.in,
    make_multiplexor(reset,
      zero.out
      make_adder(one.out, counter.out)))

  ;; Produce Verilog
  generate_verilog(counter)
```

functional programming

Functional Composition of UInt Creation


```
defn main () :
  ;; Create Components
  val reset   = Wire("reset")
  val counter = Register("counter")
  val one = UInt(1)
  val zero = UInt(0)

  ; Connect Components
  connect(counter.in,
    make_multiplexor(reset,
      zero.out ,
      make_adder( one.out , counter.out)))

  ;; Produce Verilog
  generate_verilog(counter)
```

```
defn main () :
  ;; Create Components
  val reset   = Wire("reset")
  val counter = Register("counter")

  ;; Connect Components
  connect(counter.in,
    make_multiplexor(reset,
      UInt(0) ,
      make_adder( UInt(1) , counter.out)))

  ;; Produce Verilog
  generate_verilog(counter)
```

functional programming

Overload Addition Operator


```
defn main () :
  ;; Create Components
  val reset   = Wire("reset")
  val counter = Register("counter")

  ;; Connect Components
  connect(counter.in,
    make_multiplexor(reset,
      UInt(0),
      make_adder(UInt(1), counter.out) ))

  ;; Produce Verilog
  generate_verilog(counter)
```

```
defn main () :
  ;; Create Components
  val reset   = Wire("reset")
  val counter = Register("counter")

  ;; Connect Components
  connect(counter.in,
    make_multiplexor(reset,
      UInt(0),
      UInt(1) + counter.out ))

  ;; Produce Verilog
  generate_verilog(counter)
```

operator overloading

Introduce Connect Infix Operator


```
defn main () :
  ;; Create Components
  val reset   = Wire("reset")
  val counter = Register("counter")

  ;; Connect Components
  connect( counter.in,
    make_multiplexor(reset,
      UInt(0),
      UInt(1) + counter.out) )

  ;; Produce Verilog
  generate_verilog(counter)
```

```
defn main () :
  ;; Create Components
  val reset   = Wire("reset")
  val counter = Register("counter")

  ;; Connect Components
  counter.in :=
    make_multiplexor(reset,
      UInt(0),
      UInt(1) + counter.out)

  ;; Produce Verilog
  generate_verilog(counter)
```

operator overloading

Automatically Create Multiplexors


```
defn main () :
  ;; Create Components
  val reset   = Wire("reset")
  val counter = Register("counter")

  ;; Connect Components
  counter.in :=
    make_multiplexor(reset,
      UInt(0),
      UInt(1) + counter.out )

  ;; Produce Verilog
  generate_verilog(counter)
```

```
defn main () :
  ;; Create Components
  val reset   = Wire("reset")
  val counter = Register("counter")

  ;; Connect Components
  when reset :
    counter.in := UInt(0)
  else :
    counter.in := UInt(1) + counter.out

  ;; Produce Verilog
  generate_verilog(counter)
```

dynamic scoping, closures, object-orientation

Grab Names of Wires Directly


```
defn main () :
  ;; Create Components
  val reset = Wire("reset") \colorbox{red}{val
        counter = Register("counter")}

  ;; Connect Components
  when reset :
    counter.in := UInt(0)
  else :
    counter.in := UInt(1) + counter.out

  ;; Produce Verilog
  generate_verilog(counter)
```

```
defn main () :
  ;; Create Components
  wire reset  : UInt
  reg counter : UInt

  ;; Connect Components
  when reset :
    counter.in := UInt(0)
  else :
    counter.in := UInt(1) + counter.out

  ;; Produce Verilog
  generate_verilog(counter)
```

introspection

Abstract Counter


```
defn main () :
  ;; Create Components
  wire reset  : UInt
  reg counter : UInt

  ;; Connect Components
  when reset :
    counter.in := UInt(0)
  else :
    counter.in := UInt(1) + counter.out

  ;; Produce Verilog
  generate_verilog(counter)
```

```
defn make_counter(reset:  UInt) :
  reg counter : UInt
  when reset :
    counter.in := UInt(0)
  else :
    counter.in := UInt(1) + counter.out
  counter
}

defn main () :
  ;; Create Components
  wire reset  : UInt
  wire counter = make_counter(reset)

  ;; Produce Verilog
  generate_verilog(counter)
```

functional programming

Make Reset Implicit

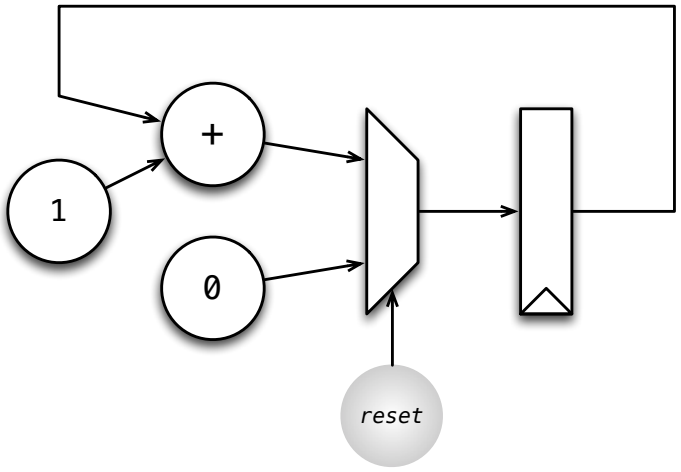

```
defn make_counter( reset: UInt ) :
  reg counter : UInt
  when reset :
    counter.in := UInt(0)
  else :
    counter.in := UInt(1) + counter.out
  counter

defn main () :
  ;; Create Components
  wire reset  : UInt
  wire counter = make_counter( reset )

  ;; Produce Verilog
  generate_verilog(counter)
```

```
defn make_counter() :
  reg counter : UInt
  when reset :
    counter.in := UInt(0)
  else :
    counter.in := UInt(1) + counter.out
  counter

defn main () :
  ;; Create Components
  val reset   = Wire()
  val counter =
    withReset(reset):
      make_counter()
  ;; Produce Verilog
  generate_verilog(counter)
```

dynamic scoping

Looks “Behavioral” but ...


```
def make_counter() = {
  reg counter : UInt
  when reset :
    counter.in := UInt(0)
  else :
    counter.in := UInt(1) + counter.out
  counter

defn main () :
  ;; Create Components
  wire reset  : UInt
  wire counter =
    withReset(reset) :
      make_counter(reset)
  ;; Produce Verilog
  generate_verilog(counter)
```



- every construct actually creates a concrete circuit
- know cost of everything
- layered and can choose level of abstraction
- not “stanza to hardware” compiler

Crucial

- Type Inference
- Infix Operator Overloading
- Lightweight Closures
- Dynamic Scoping
- Introspection (or Simple Macros)
- Functional Programming

Even Better with

- Object Orientation
- Powerful Macros

Bootcamp Chipper Installation


```
-Install VirtualBox
-File->Import appliance, chipper-vm.ova
-Start
-Login (username: chipper-bootcamp, password: chipper)
-GTKWave, Emacs, etc. all installed
```

Updating the Chipper Tutorial


```
cd chipper-tutorial
git pull
git submodule update
```

Chipper Tutorial Contents


```
chipper-tutorial/
  Makefile
  generated/
    examples/
    problems/
    solutions/
  examples/    # Contains chipper examples
    Makefile
    FullAdder.stanza ...
  problems/    # Contains skeletal files for tutorial problems
    Makefile
    Accumulator.stanza ...
  solutions/   # Contains solutions to problems
    Makefile
    Counter.stanza ...
```

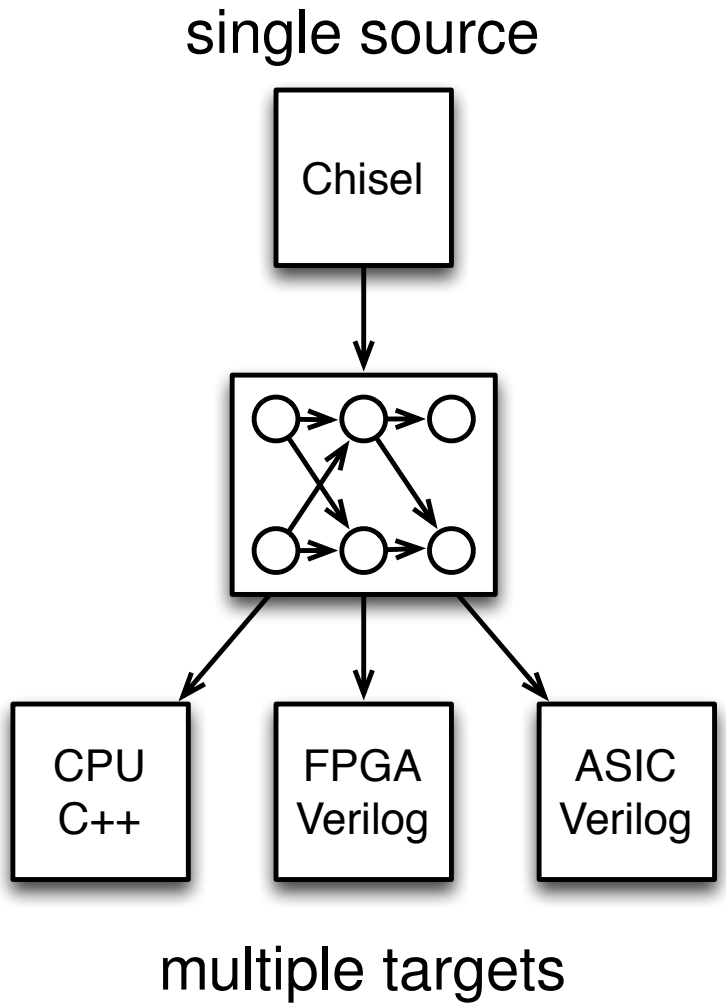
```
cd $TUT_DIR
make
```

If your system is set up correctly, you should see a messsage `[success]` followed by the total time of the run, and date and time of completion.

Get This


```
open chipper-tutorial/chipper/doc/bootcamp/bootcamp.pdf
```

- A hardware construction language
  - “synthesizable by construction”
  - creates graph representing hardware
- Embedded within Stanza language to leverage mindshare and language design
- Best of hardware and software design ideas
- Multiple targets
  - Simulation and synthesis
  - Memory IP is target-specific
- **Not** Stanza app -> Verilog arch

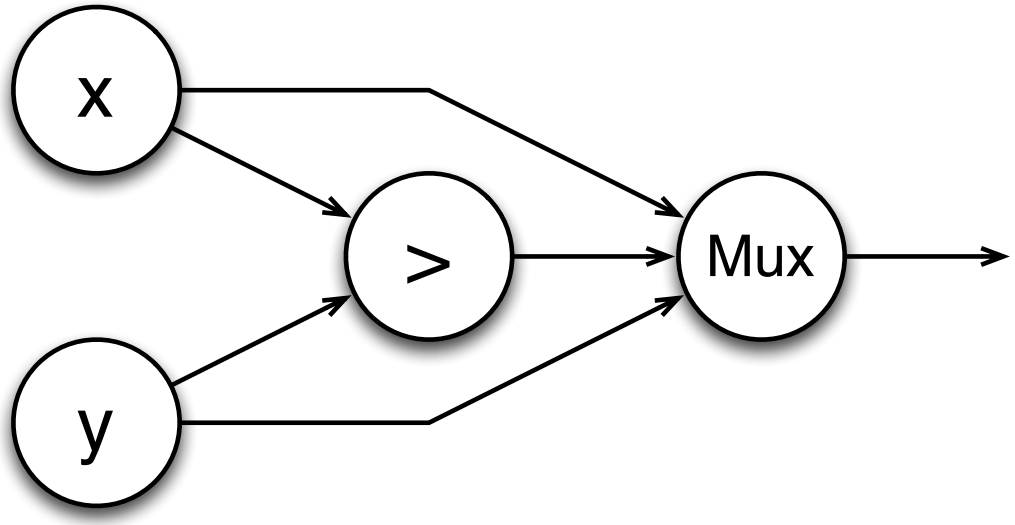


- Object Oriented
  - Factory Objects, Classes
  - Traits, overloading etc
  - Strongly typed with type inference
- Functional
  - Higher order functions
  - Anonymous functions
  - Currying etc
- Extensible
  - Macro System
  - Domain Specific Languages (DSLs)
- Fast
  - Cooperative Coroutine System
  - Native Optimizing Compiler

** Invented by Patrick Li @ EECS Berkeley

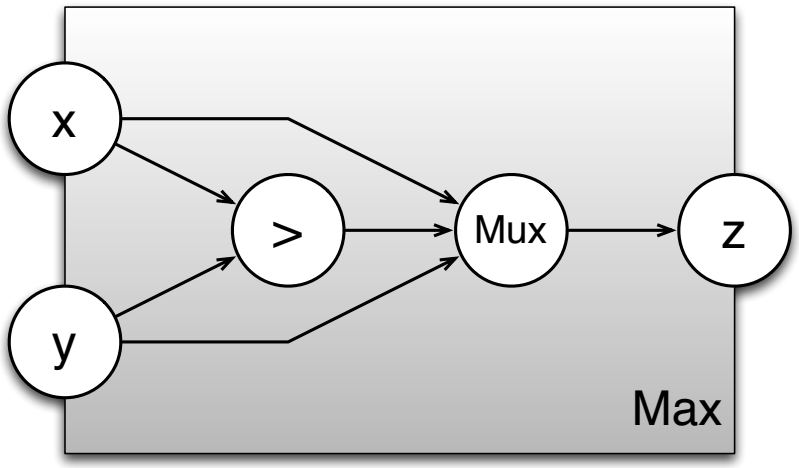
`http://www.lbstanza.org`

```
mux(x > y, x, y)
```



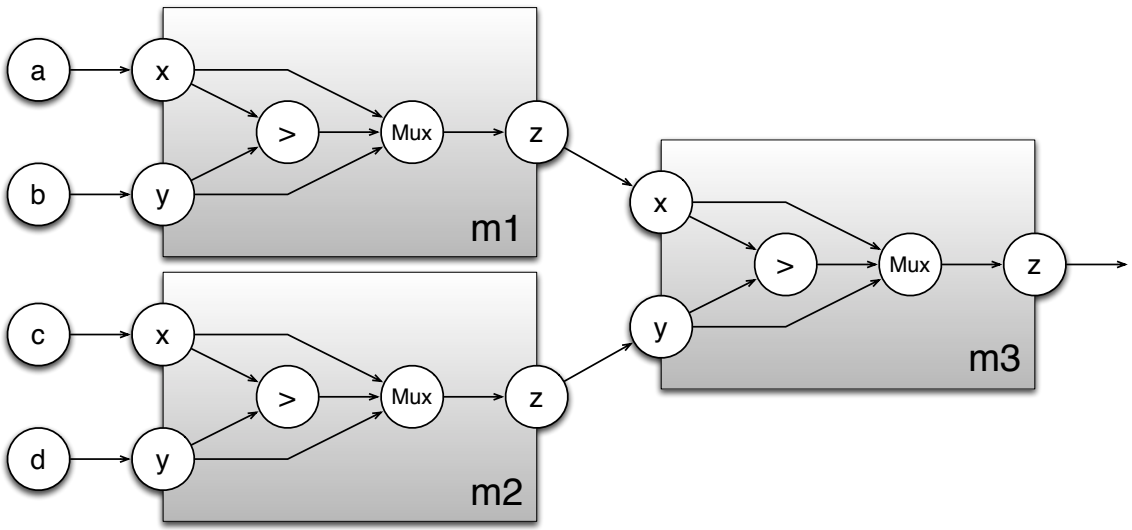
Creating Module


```
defmodule Max2 :
  input x : UInt<8>
  input y : UInt<8>
  output z : UInt<8>
  io.z := mux(io.x > io.y, io.x, io.y)
```



Connecting Modules

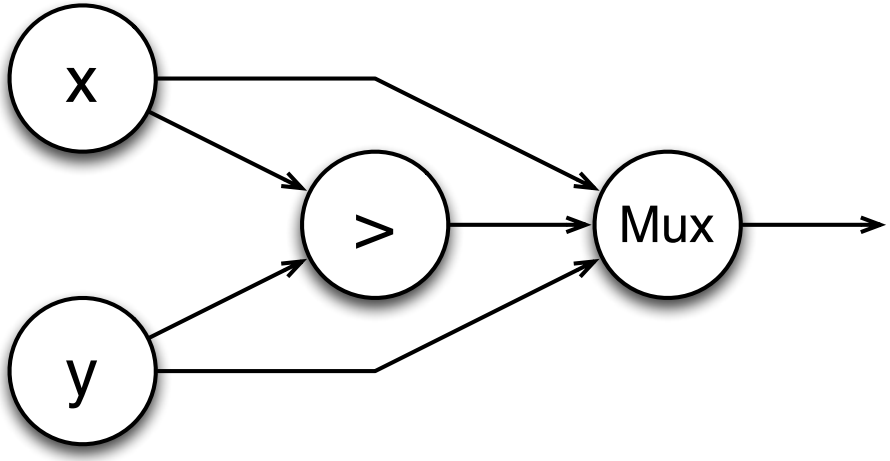

```
inst m1 : Max2()
m1.x := a
m1.y := b
inst m2 : Max2()
m2.x := c
m2.y := d
inst m3 : Max2()
m3.x := m1.z
m3.y := m2.z
```



Defining Construction Functions

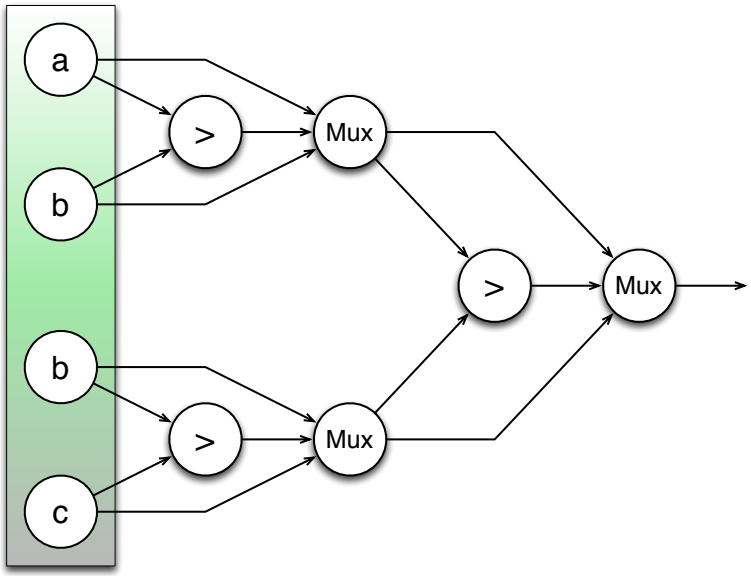

```
defn max2 (x, y): mux(x > y, x, y)
```

```
max2(x, y)
```



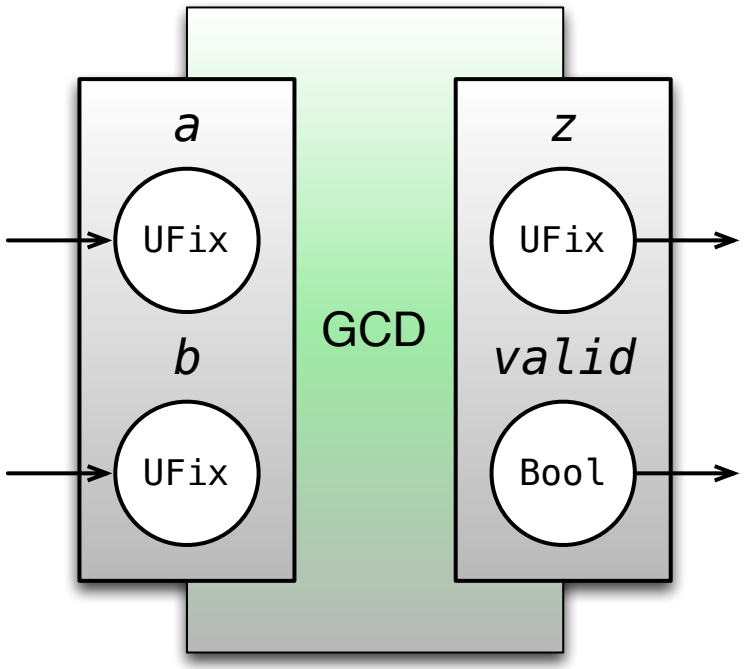
Functional Construction


```
defmodule MaxN (n: Int, w: Int) :
  input in : UInt<w>[4]
  output out : UInt<w>
  out := reduce(max2, in)
```



Example


```
defmodule GCD :
  input a     : UInt<16>
  input b     : UInt<16>
  output z    : UInt<16>
  output valid : UInt<1>
  reg x = a
  reg y = b
  when x > y :
    x := x - y
  else :
    y := y - x
  z     := x
  valid := y === UInt(0)
```



Running the Chipper Simulation


```
cd ~/chipper-tutorial/examples
make GCD.out
```

```
...
PASSED
[success] Total time: 2 s, completed Feb 28, 2013 8:14:37 PM
```

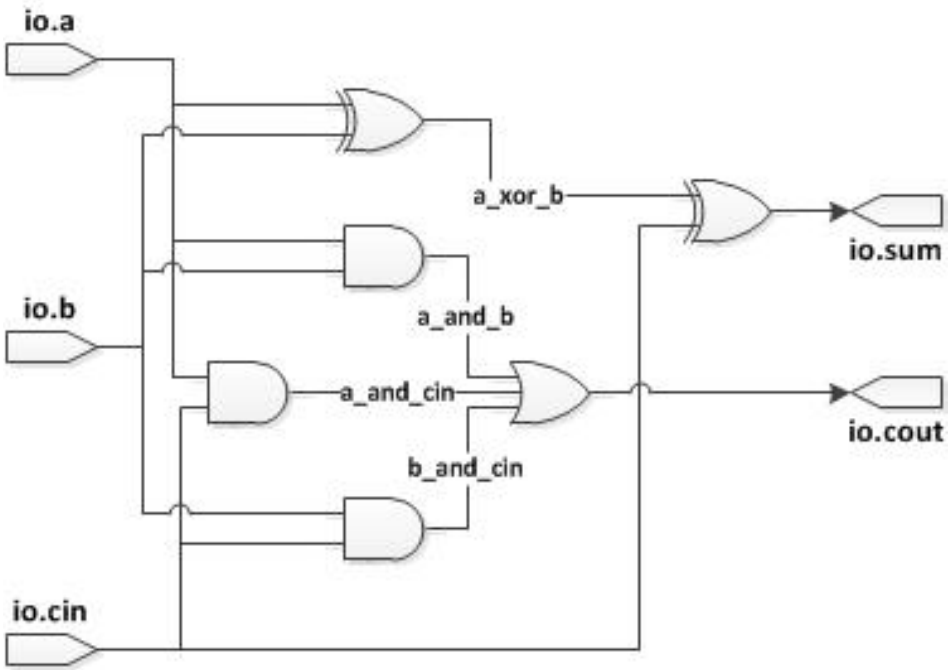
```
cd ~/chipper-tutorial/examples
make GCD.v
```

The Verilog source is roughly divided into three parts:

1. Module declaration with input and outputs
2. Temporary wire and register declaration used for holding intermediate values
3. Register assignments in `always @ (posedge clk)`

FullAdder – Type Inference


```
defmodule FullAdder :
  input a: UInt<1>
  input b: UInt<1>
  input cin: UInt<1>
  output sum: UInt<1>
  output cout: UInt<1>
  ;; Generate the sum
  wire a_xor_b = a ^ b
  sum := a_xor_b ^ io.cin
  ;; Generate the carry
  wire a_and_b   = a & b
  wire b_and_cin = b & cin
  wire a_and_cin = a & cin
  cout := a_and_b |
    b_and_cin | a_and_cin
```



FullAdder Verilog – Width Inference 1


```
defmodule FullAdder :
  input a: UInt<1>
  input b: UInt<1>
  input cin: UInt<1>
  output sum: UInt<1>
  output cout: UInt<1>
  ;; Generate the sum
  wire a_xor_b = a ^ b
  sum := a_xor_b ^ io.cin
  ;; Generate the carry
  wire a_and_b   = a & b
  wire b_and_cin = b & cin
  wire a_and_cin = a & cin
  cout := a_and_b |
    b_and_cin | a_and_cin
```

```
module FullAdder(
    input  io_a,
    input  io_b,
    input  io_cin,
    output io_sum,
    output io_cout);
  wire T0;
  wire a_and_cin;
  wire T1;
  wire b_and_cin;
  wire a_and_b;
  wire T2;
  wire a_xor_b;

  assign io_cout = T0;
  assign T0 = T1 | a_and_cin;
  assign a_and_cin = io_a & io_cin;
  assign T1 = a_and_b | b_and_cin;
  assign b_and_cin = io_b & io_cin;
  assign a_and_b = io_a & io_b;
  assign io_sum = T2;
  assign T2 = a_xor_b ^ io_cin;
  assign a_xor_b = io_a ^ io_b;
endmodule
```

FullAdder2 Verilog – Width Inference 2


```
defmodule FullAdder :
  input a: UInt<2>
  input b: UInt<2>
  input cin: UInt<2>
  output sum: UInt<2>
  output cout: UInt<1>
  ;; Generate the sum
  wire a_xor_b = a ^ b
  sum := a_xor_b ^ io.cin
  ;; Generate the carry
  wire a_and_b   = a & b
  wire b_and_cin = b & cin
  wire a_and_cin = a & cin
  cout := a_and_b | b_and_cin |
       a_and_cin
```

```
module FullAdder(
    input [1:0] io_a,
    input [1:0] io_b,
    input [1:0] io_cin,
    output[1:0] io_sum,
    output[1:0] io_cout);
  wire[1:0] T0;
  wire[1:0] a_and_cin;
  wire[1:0] T1;
  wire[1:0] b_and_cin;
  wire[1:0] a_and_b;
  wire[1:0] T2;
  wire[1:0] a_xor_b;

  assign io_cout = T0;
  assign T0 = T1 | a_and_cin;
  assign a_and_cin = io_a & io_cin;
  assign T1 = a_and_b | b_and_cin;
  assign b_and_cin = io_b & io_cin;
  assign a_and_b = io_a & io_b;
  assign io_sum = T2;
  assign T2 = a_xor_b ^ io_cin;
  assign a_xor_b = io_a ^ io_b;
endmodule
```

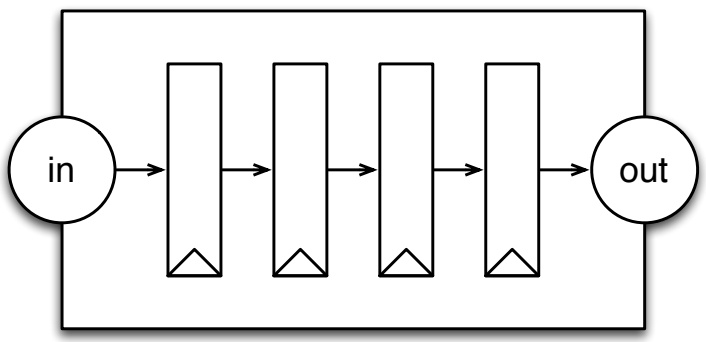
Using Registers


```
// clock the new reg value on every cycle
wire y = x
reg z := y
```

```
// clock the new reg value when the condition a > b
reg x : UInt
when a > b :
  x := y
else when b > a :
  x := z
else :
  x := w
```

Unconditional Register Update

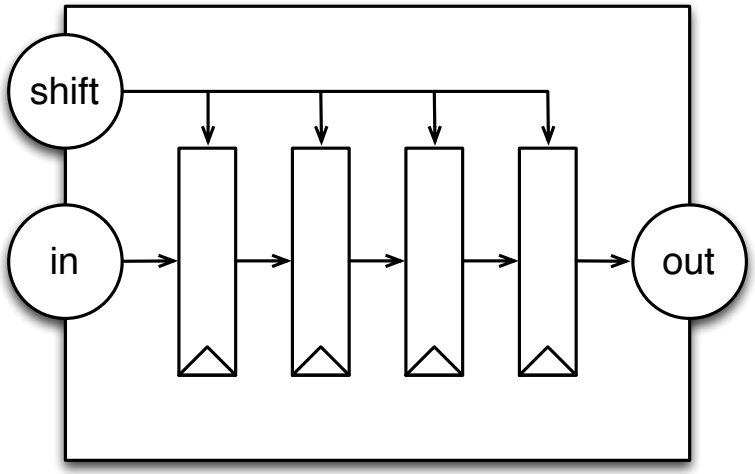

```
defmodule ShiftRegister :
  input in : UInt<1>
  output out : UInt<1>
  reg r0 := in
  reg r1 := r0
  reg r2 := r1
  reg r3 := r2
  out := r3
```




```
module ShiftRegister(input clk, input reset,
    input  io_in,
    output io_out);

  reg[0:0] r3;
  reg[0:0] r2;
  reg[0:0] r1;
  reg[0:0] r0;

  assign io_out = r3;
  always @(posedge clk) begin
    r3 <= r2;
    r2 <= r1;
    r1 <= r0;
    r0 <= io_in;
  end
endmodule
```

Conditional Register Update


```
defmodule ShiftRegister :
  input in : UInt<1>
  input shift : UInt<1>
  output out : UInt<1>

  reg r0 : UInt<1>
  reg r1 : UInt<1>
  reg r2 : UInt<1>
  reg r3 : UInt<1>

  when shift :
    r0 := in
    r1 := r0
    r2 := r1
    r3 := r2
  out := r3
```



Conditional Register Update with Reset


```
defmodule EnableShiftRegister :
  input in : UInt<1>
  input shift : UInt<1>
  output out : UInt<1>

  ;; Register reset to zero
  reg r0 = UInt<1>(0)
  reg r1 = UInt<1>(0)
  reg r2 = UInt<1>(0)
  reg r3 = UInt<1>(0)

  when shift :
    r0 := in
    r1 := r0
    r2 := r1
    r3 := r2
  out := r3
```

UInt Literals

inferred width

```
UInt(1)       ;; decimal 1-bit literal from Stanza Int.
UInt("ha")    ;; hexadecimal 4-bit literal from string.
UInt("o12")   ;; octal 4-bit literal from string.
UInt("b1010") ;; binary 4-bit literal from string.
```

specified widths

```
UInt("h_dead_beef") ;; 32-bit literal of type UInt.
UInt(1)             ;; decimal 1-bit literal from Stanza Int.
UInt<8>("ha")       ;; hexadecimal 8-bit literal of type UInt.
UInt<6>("o12")      ;; octal 6-bit literal of type UInt.
UInt<12>("b1010")   ;; binary 12-bit literal of type UInt.
UInt<8>(5)          ;; unsigned decimal 8-bit literal of type UInt.
```

Sequential Circuit Problem – `Accumulator.stanza` 49

- write sequential circuit that sums `in` values
- in `chipper-tutorial/problems/Accumulator.stanza`
- run `make Accumulator.out` until passing

```
defmodule Accumulator :
  input in : UInt<1>
  output out : UInt<8>

  ;; flush this out ...

  out := UInt(0)
```

UInt Operations and Conditional Assignment


```
defmodule BasicALU :
  input a      : UInt<4>
  input b      : UInt<4>
  input opcode : UInt<4>
  output out = UInt<4>
  out := UInt(0)
  when opcode === UInt(0) :
    out := a                    ;; pass A
  else when opcode === UInt(1) :
    out := b                    ;; pass B
  else when: opcode === UInt(2) :
    out := a + UInt(1)          ;; inc A by 1
  else when: opcode === UInt(3) :
    out := a - UInt(1)          ;; dec B by 1
  else when: opcode === UInt(4) :
    out := a + UInt(4)          ;; inc A by 4
  else when: opcode === UInt(5) :
    out := a - UInt(4)          ;; dec A by 4
  else when: opcode === UInt(6) :
    out := a + b                ;; add A and B
  else when: opcode === UInt(7) :
    out := a - b                ;; sub B from A
  else when: opcode === UInt(8) :
    out := (a < b)              ;; set on A < B
  else :
    out := (a === b)            ;; set on A == B
```

- wire `io.output` defaulted to 0 and then
- conditionally reassigned to based on opcode
- unlike registers, wires are required to be defaulted
- wires also allow forward declarations

UInt Operations

Symbol	Operation
+	Add
-	Subtract
*	Multiply
/	UInt Divide
%	Modulo
~	Bitwise Negation
^	Bitwise XOR
& Bitwise AND	UInt
|	Bitwise OR
===	Equal
!==	Not Equal
>	Greater
<	Less
>=	Greater or Equal
<=	Less or Equal

Bit Extraction


```
;; extracts the x through y bits of value
wire x_to_y = value[x, y]
```

```
;; extract the x-th bit from value
wire x_of_value = value[x]
```

```
defmodule ByteSelector :
  input in: UInt<32>
  input offset: UInt<2>
  output out: UInt<8>

  when offset === UInt(0) :
    out := in[7,0]
  else when offset === UInt(1) :
    out := in[15, 8]
  else when offset === UInt(2) :
    out := in[23,16]
  else :
    out := in[31,24]
```

Bit Concatenation and Filling

You concatenating bits using `Cat`:

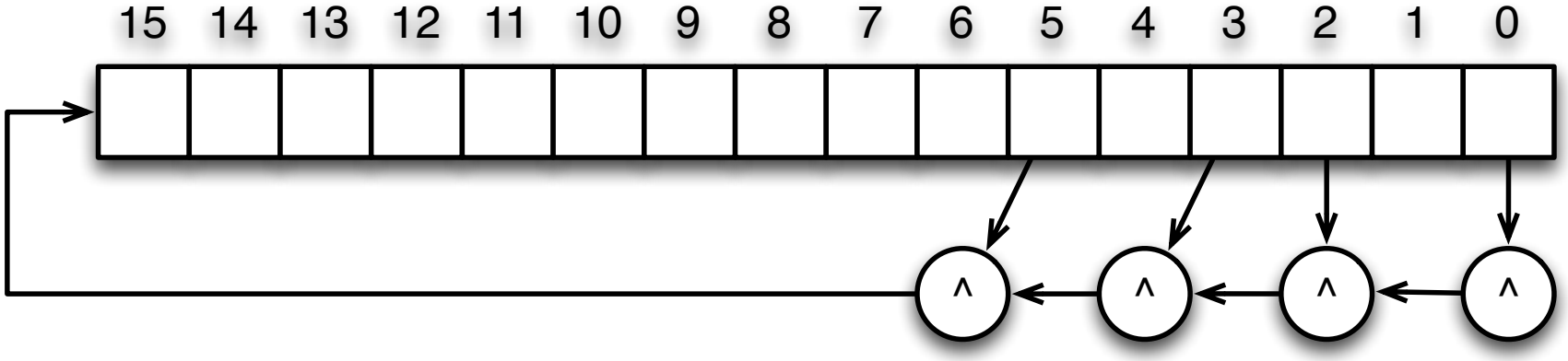
```
wire A   : UInt<32>
wire B   : UInt<32>
wire bus = cat(A, B) ;; concatenate A and B
```

and replicate bits using `Fill`:

```
// Replicate a bit string multiple times.
val usDebt = Fill(3, UInt("hA"))
```

```
defmodule LFSR16 :
  input inc : UInt<1>
  output out : UInt<16>
  ;; ...
  out := UInt(0)
```

- reg, cat, extract, ^
- init reg to 1
- updates when inc asserted



UInt Bit Inference


```
defmodule HiLoMultiplier :
  input A  : UInt<16>
  input B  : UInt<16>
  output Hi : UInt<16>
  output Lo : UInt<16>
  wire mult = A * B
  Lo := mult[15, 0]
  Hi := mult[31, 16]
```

```
module HiLoMultiplier(
    input [15:0] io_A,
    input [15:0] io_B,
    output[15:0] io_Hi,
    output[15:0] io_Lo);

  wire[15:0] T0;
  wire[31:0] mult; // inferred as 32 bits
  wire[15:0] T1;

  assign io_Lo = T0;
  assign T0 = mult[4'hf:1'h0];
  assign mult = io_A * io_B;
  assign io_Hi = T1;
  assign T1 = mult[5'h1f:5'h10];
endmodule
```

Bit Inference Rules

Operation	Result Bit Width
`Z = X + Y`	max(Width(X), Width(Y))
`Z = X - Y`	max(Width(X), Width(Y))
`Z = X & Y`	min(Width(X), Width(Y))
`Z = X | Y`	max(Width(X), Width(Y))
`Z = X ^ Y`	max(Width(X), Width(Y))
`Z = ~X`	Width(X)
`Z = mux(C, X, Y)`	max(Width(X), Width (Y))
`Z = X * Y`	Width(X) + Width(Y)
`Z = X << n`	Width(X) + n
`Z = X >> n`	Width(X) - n
`Z = cat(X, Y)`	Width(X) + Width(Y)
`Z = fill(n, x)`	Width(X) + n

Bool Type

The Chipper Bool is used to represent the result of logical expressions:

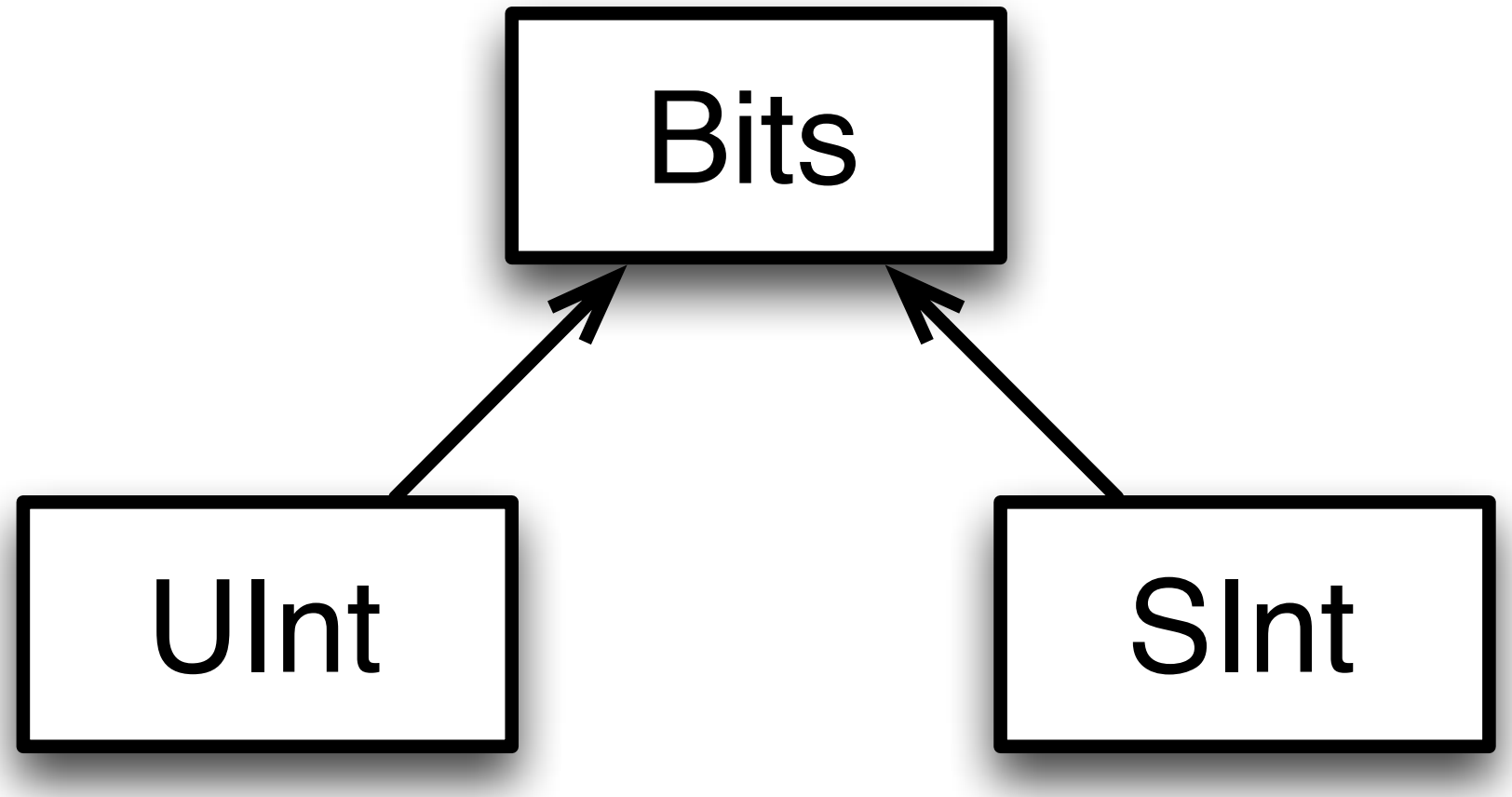
```
wire change = io.a === io.b ;; change gets Bool type
when change : ;; execute if change is true
  ...
```

You can instantiate a Bool value like this:

```
wire true_value  = UInt(true)
wire false_value = UInt(false)
```

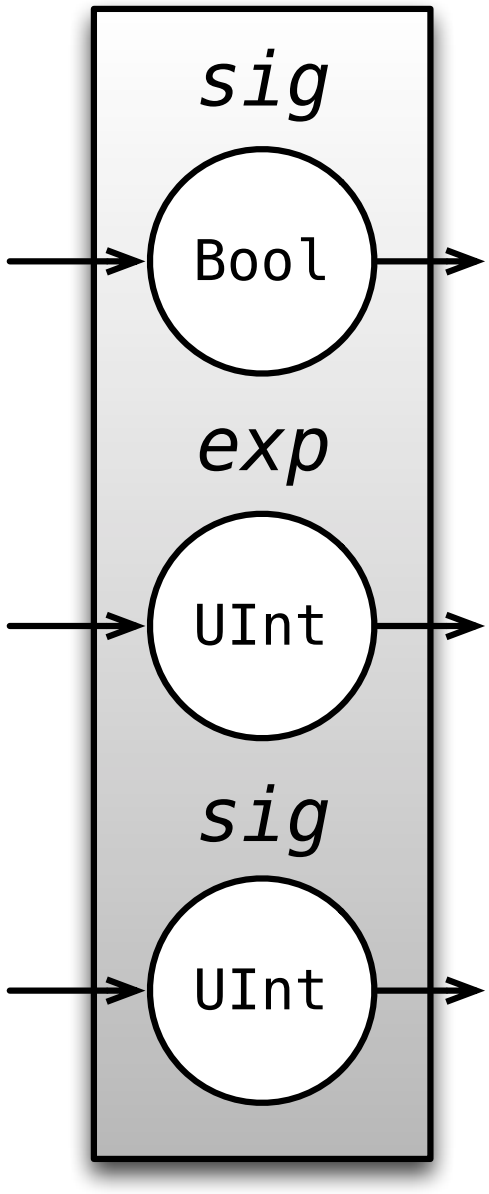
Bits Subtype Hierarchy

- `SInt` is a signed integer type



Bundles


```
defbundle MyFloat :
  output sign : UInt<1>
  output exponent : UInt<8>
  output significand : UInt<23>

wire x  : MyFloat
wire xs = x.sign
```

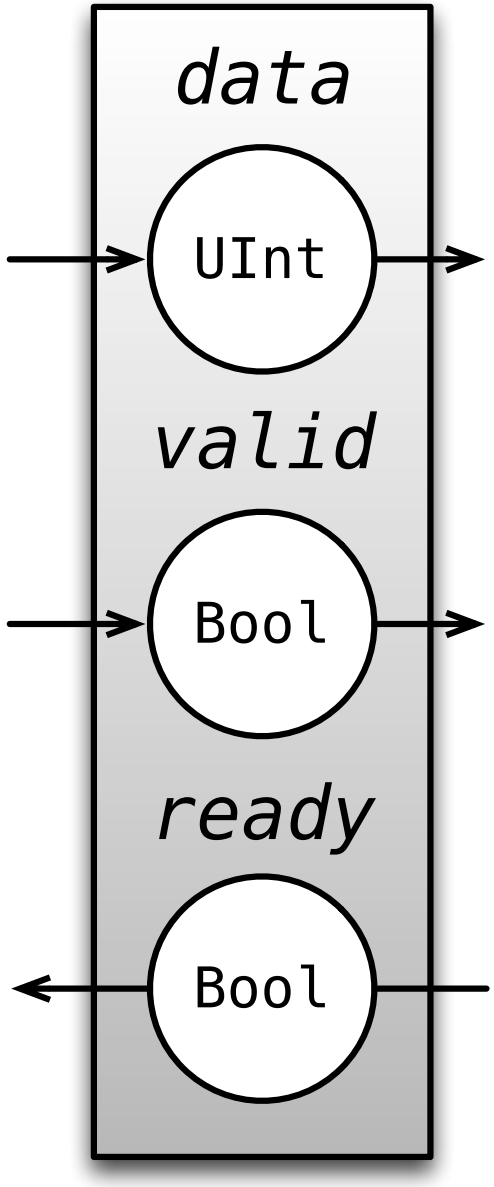


Data object with directions assigned to its members

```
defbundle Decoupled :
  output data : UInt<32>
  output valid : UInt<1>
  input ready : UInt<1>
```

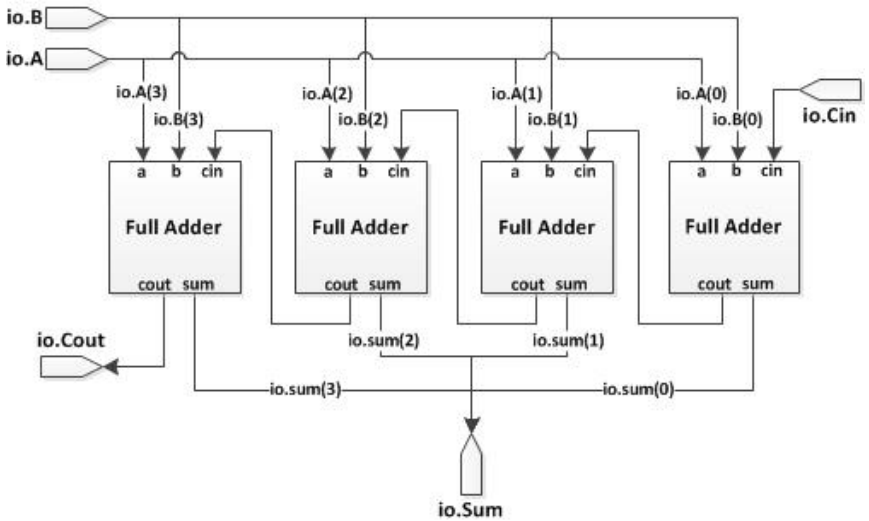
Direction assigned at instantiation time

```
defbundle ScaleIO :
  input in : MyFloat
  input scale : MyFloat
  output out : MyFloat
```



Instantiating Modules


```
;; A 4-bit adder with carry in and carry out
defmodule Adder4:
  input A : UInt<4>
  input B : UInt<4>
  input Cin : UInt<1>
  output Sum : UInt<4>
  output Cout : UInt<1>
  ;; Adder for bit 0
  inst Adder0 : FullAdder
  Adder0.a   := A[0]
  Adder0.b   := B[0]
  Adder0.cin := Cin
  wire s0 = Adder0.sum
  ;; Adder for bit 1
  inst Adder1 = FullAdder
  Adder1.a   := A[1]
  Adder1.b   := B[1]
  Adder1.cin := Adder0.cout
  wire s1 = Cat(Adder1.sum, s0)
  ...
  ;; Adder for bit 3
  inst Adder3 : FullAdder
  Adder3.a   := A[3]
  Adder3.b   := B[3]
  Adder3.cin := Adder2.cout
  Sum  := Cat(Adder3.sum, s2)
  Cout := Adder3.cout
```



- inherits from `Module` class,
- contains an interface stored in a port field named `io`, and
- wires together subcircuits in its constructor.

constructing vecs

```
wire myVec1 : <data type>[<number of elements>]
wire myVec2 = Vec(<elt0>, <elt1>, ...)
```

creating a vec of wires

```
wire ufix5_vec10 : UInt<5>[10]
```

creating a vec of regs

```
reg reg_vec32 : UInt<16>[32]
```

writing

```
reg_vec32[1] := UInt(0)
```

reading

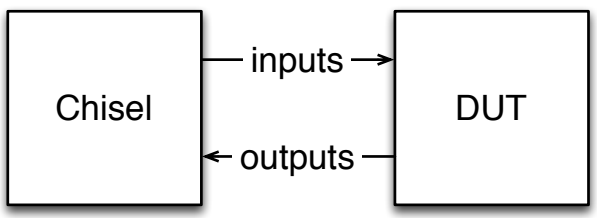
```
wire reg5 = reg_vec[5]
```

- add loadability to shift register
- change interface to use vec's

```
defmodule VecShiftRegister :
  input ins: UInt<w>[4]
  input load: UInt<1>
  input shift: UInt<1>
  output out: UInt<w>

  reg delays: UInt<w>[4]
  when load :
    ;; fill in here ...
  else when shift :
    ;; fill in here ...
  out := delays[3]
```

Defining a Tester


```
class ByteSelector extends Module {
  input in: UInt<32>
  input offset: UInt<2>
  output out: UInt<8>
  out := UInt<8>(0)
  ...

defn bs-tests () :
  with-tester [t, c] = ByteSelector() :
    val test_in = 12345678
    for i in 0 to 4 all? :
      poke(t, c.in,     test_in)
      poke(t, c.offset, t)
      step(1)
      val ref_out = (test_in >> (t * 8)) & 255
      expect(t, c.out, ref_out)
```




```
defclass Tester
defn Tester (dut:String) -> Tester
defmulti dut (t:Tester) -> String
defmulti reset (t:Tester, n:Int) -> Int
defmulti step (t:Tester, n:Int) -> Int
defmulti peek (t:Tester, data:Bits) -> Int
defmulti peek (t:Tester, data:Bits, index:Int) -> Int
defmulti poke (t:Tester, data:Bits, x:Int) -> Int
defmulti poke (t:Tester, data:Bits, i:Int, x:Int) -> Int
defmulti expect (t:Tester, data:Bits, target:Int) -> True|False
defn expect (good:Boolean, msg: Streamable) -> True|False

defn int(x: Boolean) -> BigInt
defn int(x: Int) -> BigInt
defn int(x: Bits) -> BigInt
```

which binds a tester to a module and allows users to write tests using the given debug protocol. In particular, users utilize:

- `poke` to set input port and state values,
- `step` to execute the circuit one time unit,
- `peek` to read port and state values, and
- `expect` to compare peeked circuit values to expected arguments.

Simulation Debug Output


```
> cd chipper-tutorial/examples
> make ByteSelector.out
STARTING ../emulator/problems/ByteSelector
---
RESET 1 -> 1
STARTING TESTS
WIRE-POKE ByteSelector.in = 16807 -> ok
WIRE-POKE ByteSelector.offset = 1 -> ok
STEP 1 -> 0
WIRE-PEEK ByteSelector.out -> 0x41
EXPECT ByteSelector.out VALUE 65 -> 65
WIRE-POKE ByteSelector.in = 548908249 -> ok
WIRE-POKE ByteSelector.offset = 2 -> ok
STEP 1 -> 0
...
WIRE-PEEK ByteSelector.out -> 0x35
EXPECT ByteSelector.out VALUE 53 -> 53
WIRE-POKE ByteSelector.in = 881157273 -> ok
WIRE-POKE ByteSelector.offset = 2 -> ok
STEP 1 -> 0
WIRE-PEEK ByteSelector.out -> 0x85
EXPECT ByteSelector.out VALUE 133 -> 133
SUCCESS
[success] Total time: 26 s, ...
```

In particular, users utilize:

- `poke` to set input port and state values,
- `step` to execute the circuit one time unit,
- `peek` to read port and state values, and
- `expect` to compare peeked circuit values to expected arguments.

- write a testbench for MaxN

```
defmodule MaxN (n: Int, w: Int) :
  input ins : UInt<w>[n]
  output out : UInt<w>
  defn Max2 (x: UInt, y: UInt) :
    mux(x > y, x, y)
  out := ins.reduce(Max2)
```

```
;; returns random int in 0..lim-1
val x = rnd.nextInt(lim)
```

```
defn MaxNTests () :
  with-tester [t,c] = MaxN() :
    for i in 0 to 10 do :
      for j in 0 to num(c) do :
        ;; FILL THIS IN HERE
        poke(t, c.ins[0], 0)
      ;; FILL THIS IN HERE
      step(1)
      expect(t, c.out, 1)
```

Dynamically Accessed Vec


```
defmodule MemorySearch (inits:Streamable<Int>) :
  input  isRd:   UInt<1>
  input  data:   UInt<4>
  output done:   UInt<1>
  output rdAddr: UInt<3>

  wire vals: UInt<4>[7]
  for (e in inits, i in 0 to false) do:
    vals[i] := UInt(e)
  reg index : UInt<4>
  wire elt    = vals[index]
  wire isDone = ! isRd & ((elt === data) | (index === UInt(7)))
  when isRd :
    index := UInt(0)
  else when ! isDone :
    index := index + UInt(1)
  done    := isDone
  rdAddr  := index
```

RAM is supported using the `cmem` construct

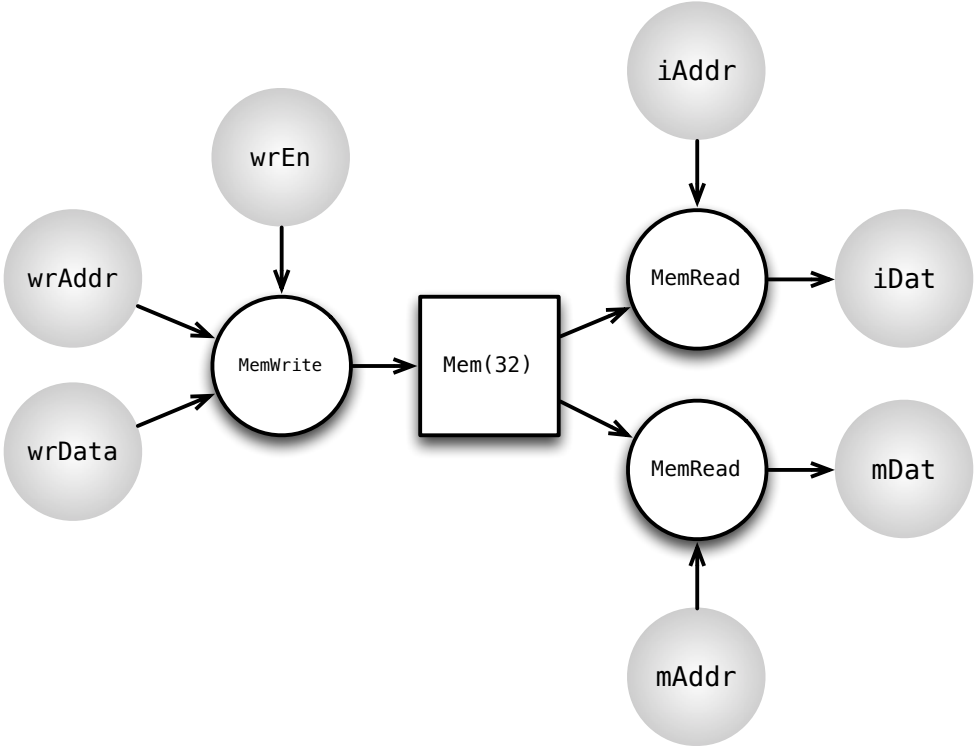
```
cmem m : UInt<32>[32]
```

where

- writes to Mems are positive-edge-triggered
- reads are either combinational or positive-edge-triggered
- ports are created by applying a `UInt` index

32-entry Register File


```
cmem regs : UInt<32>[32]
when wrEn :
  regs[wrAddr] := wrData
wire iDat = regs[iAddr]
wire mDat = regs[mAddr]
```



Load/Search Mem – DynamicMemorySearch.stanza 72

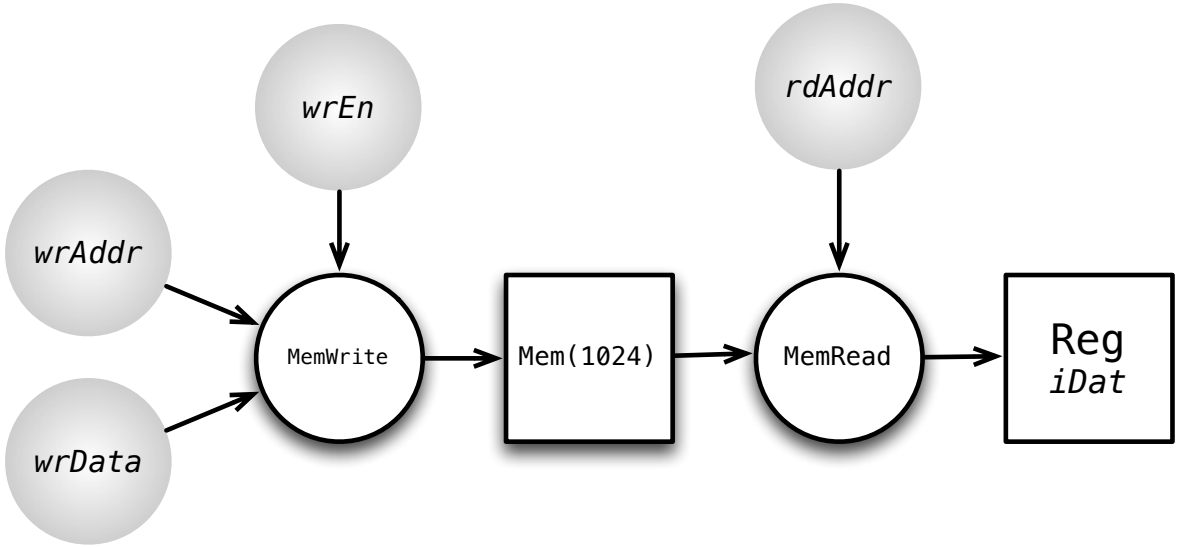
```
defmodule DynamicMemorySearch (n:Int, w:Int) :
  input  isWr:   UInt<1>
  input  wrAddr: UInt<sizeof(w)>
  input  isRd:   UInt<1>
  input  data:   UInt<w>
  output done:   UInt<1>
  output rdAddr: UInt<sizeof(w)>

  reg index = UInt<sizeof(w)>(0)
  ;; ...
  wire elt  = vals[index]
  wire isDone = ! (isRd | isWr) & ((elt === data) | (index === UInt(n - 1)))
  when isWr :
    vals[wrAddr] := data
  ;; ...
  else when ! isDone :
    index := index + UInt(1)
  done    := isDone
  rdAddr  := index
```

Sequential Read Ports

Sequential read ports are created using sequential memories:

- reads are delayed one cycle

```
smem ram1r1w : UInt<32>[1024]
when wen: ram1r1w[waddr] := wdata
when ren: eg_raddr := raddr
wire rdata = ram1r1w[reg_raddr]
```

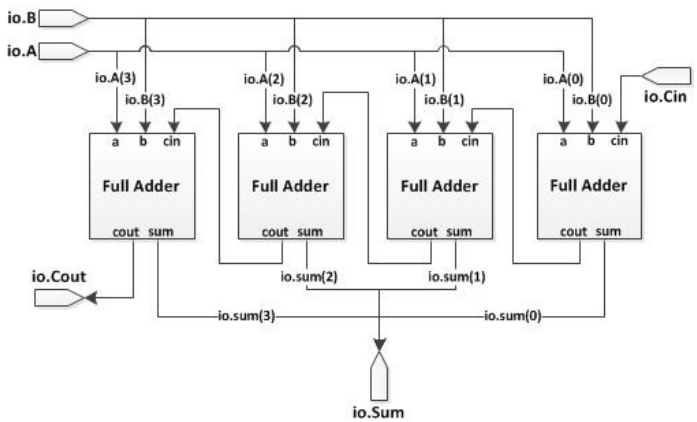


Stack

```
defmodule Stack (n:Int) :
  input push: UInt<1>
  input pop: UInt<1>
  input en: UInt<1>
  input dataIn: UInt<32>
  output dataOut: UInt<32>
  cmem stack_mem : UInt<32>[n] ;; declare stack memory
  reg sp : UInt<sizeof(n)> sp := UInt(0)
  wire out : UInt<32> out := UInt(0)
  when en :
    ;; Push condition - make sure stack isn't full
    when push & (sp < UInt(depth)) :
      stack_mem[sp] := dataIn
      sp := sp + UInt(1)
    ;; Pop condition - make sure the stack isn't empty
    else when pop & (sp > UInt(0)) :
      sp := sp - UInt(1)
    when sp > UInt(0) :
      out := stack_mem[sp - UInt(1)]
  dataOut := out
```

Scripting Hardware Generation


```
;; A n-bit adder with carry in and carry out
defmodule Adder (n: Int) :
  input A : UInt<n>
  input B : UInt<n>
  input Cin : UInt<1>
  output Sum : UInt<n>
  output Cout : UInt<1>
  ;; create a vector of FullAdders
  inst FAs   : FullAdder[n]
  wire carry = UInt<1>[n + 1]
  wire sum   = UInt<1>[n]

  ;; first carry is the top level carry in
  carry[0] := Cin

  ;; wire up the ports of the full adders
  for i in 0 to n do :
    FAs[i].a     := A[i]
    FAs[i].b     := B[i]
    FAs[i].cin   := carry[i]
    carry[i + 1] := FAs[i].cout
    sum[i]       := FAs[i].sum
  Sum  := to-uint(sum)
  Cout := carry[n]
```



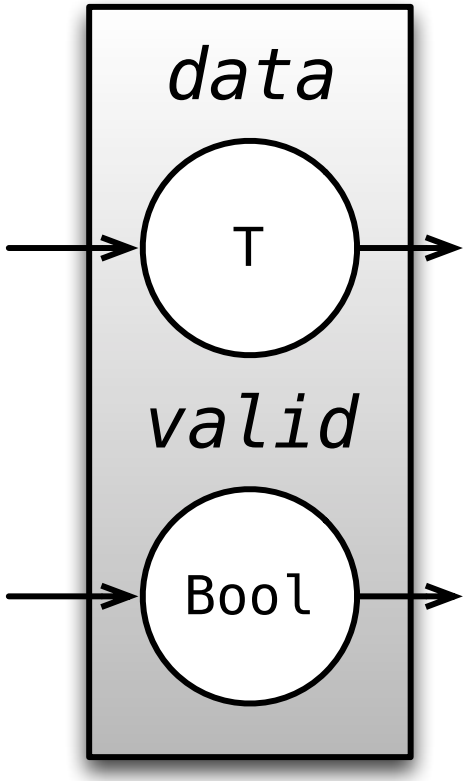
Mul Lookup Table Problem – `Mul.stanza`

- write 16x16 multiplication table using `Vec`

```
defmodule Mul :
  input x : UInt<4>
  input y : UInt<4>
  output z : UInt<8>
  wire tab : UInt<8>[256]

  ;; TODO: would like to do uint/cat here
  for i in 0 to 16 do :
    for j in 0 to 16 do :
      tab[(i << 4) | j] := UInt(i * j)

  z := tab[cat(x, y)]
```

Valid Wrapper


```
defbundle Valid<T> :
  output data : T
  output valid : UInt<1>

defmodule GCD :
  input a : UInt<16>
  input b : UInt<16>
  output out : Valid<UInt<16>>
  ...
  out.data  := x
  out.valid := y === UInt(0)
```



Bits Properities	`log2Up`, `log2Down`, `isPow2`, `PopCount`
Numeric Utilities	`LFSR16`, `Reverse`, `FillInterleaved`
Stateful Functions	`ShiftRegister`, `Counter`
Priority Encoding Functions	`UIntToOH`, `OHToUInt`, `Mux1H`
Priority Encoders	`PriorityEncoder`, `PriorityEncoderOH`
Queues and Pipes	`Decoupled`, `Queue`, `Valid`, `Pipe`
Arbiters	`ArbiterIO`, `Arbiter`, `RRArbiter`

- Required parameter `entries` controls depth
- The width is determined from the inputs.

```
defbundle QueueIO<T> (entries: Int) :
  input enq    : DecoupledIO<T>
  output deq   : DecoupledIO<T>
  output count : UInt<log2Up(entries + 1)>

defmodule Queue<T>
    (entries: Int,
     pipe: Boolean = false,
     flow: Boolean = false
     flushable: Boolean = false)
```

```
inst q : Queue<UInt>(16)
q.enq := producer.out
consumer.in := q.deq
```

directory structure

```
Hello/
  Hello.stanza # your source file
```

```
defmodule Hello :
  output out : UInt<8> }
  out := UInt(33)

defn hello-tests () :
  with-tester [t,c] = Hello() :
    step(t, 1)
    expect(t, c.out, 33)

defn main () :
  if commandline-arguments[1] == "--testing" :
    hello-tests(inst-of Hello)
  else :
    println(circuit-of Hello)

defn main () :
  chipper-main(
  if commandline-arguments[1] == "--testing" :
    hello-tests(inst-of Hello)
  else :
    println(circuit-of Hello)
```

Producing C++

```
chipperc Hello.stanza
```

Producing Verilog

```
chipperv Hello.stanza
```

Running the Chipper Tests

```
chipperc Hello.stanza
Hello "--testing"
```

set hello project up

```
cd ~
mkdir hello
cp -r ~/chipper-tutorial/hello/* hello
cd hello
sbt run
```

make a change

- make output a function of an new input

Chipper Workflow