Chipper Bootcamp

Jonathan Bachrach

EECS UC Berkeley

July 17, 2015

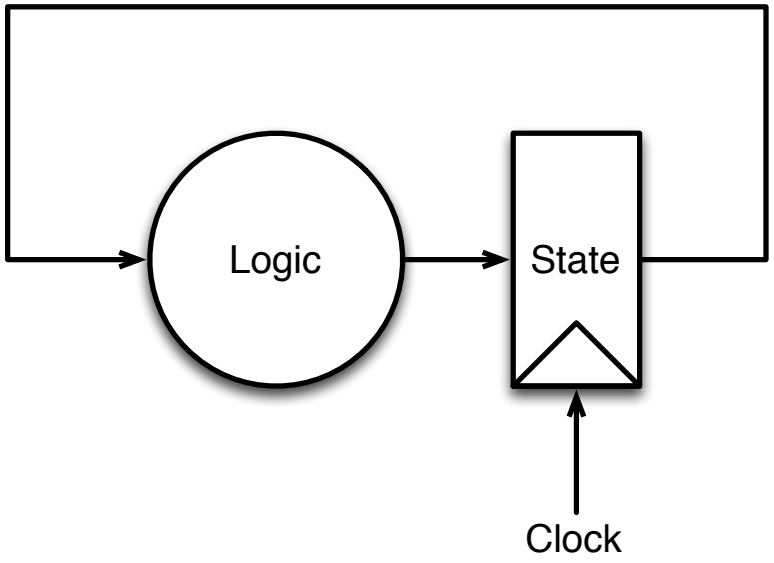
Goals for Bootcamp

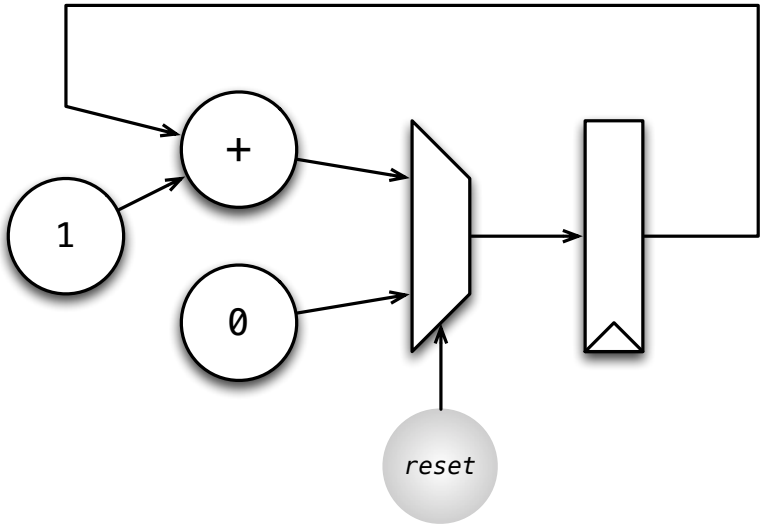
- introduction to reconfigurable computing
- introduction to hardware design
- introduce you to stanza
- get you started with Chipper
- get a basic working knowledge of Chipper
- learn how to think in Chipper
- how to write an accelerator
- know where to get more information

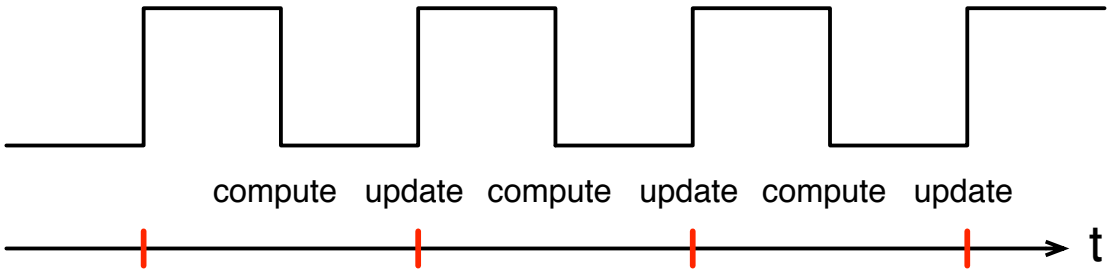
Register Transfer Level = RTL

- logic and state
- state machines

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

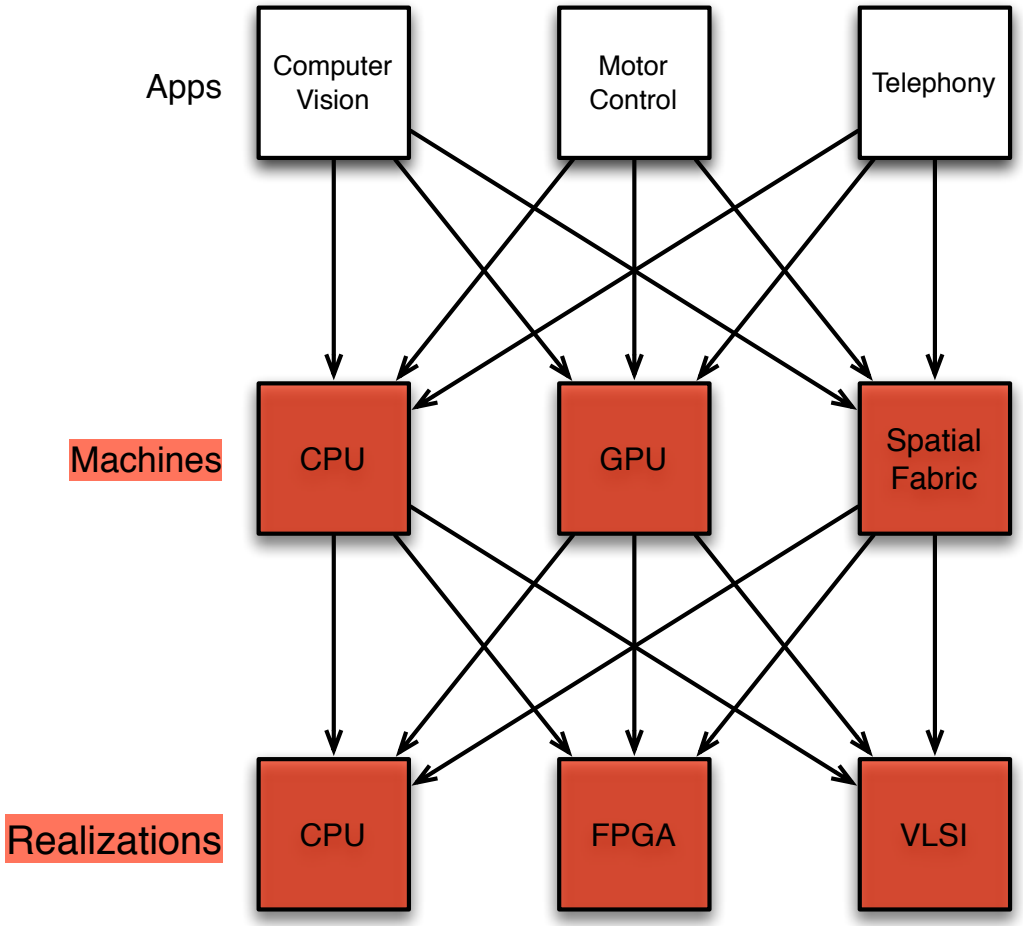






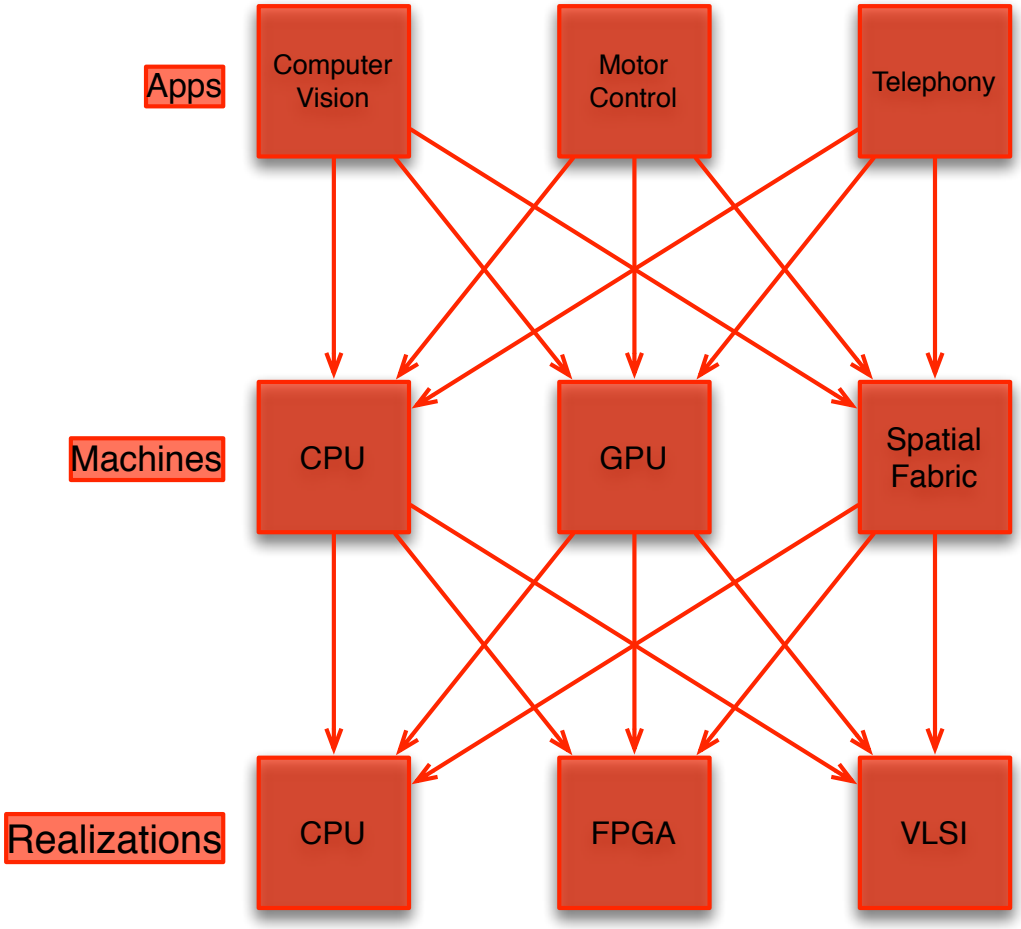
synchronous clock

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

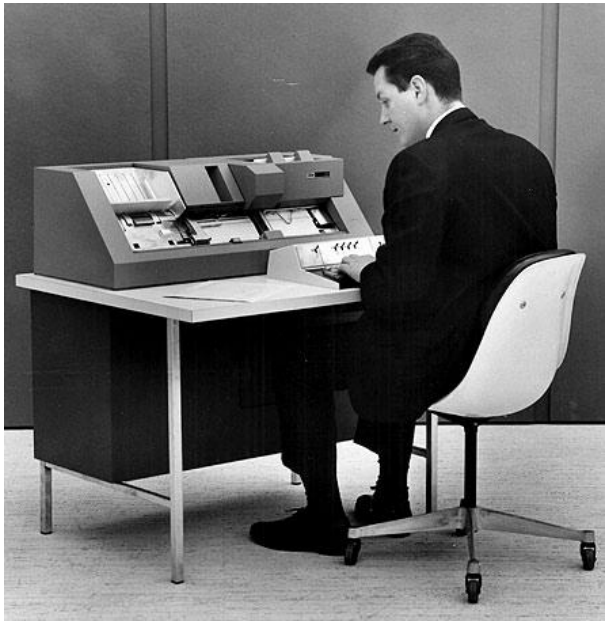
- slow hardware design
  - 1980's style languages and baroque tool chains with ad hoc scripts
  - manual optimization obscuring designs
  - minimal compile-time and run-time errors
  - army of people in both CAD tools and design – costs $10Ms
- slow and expensive to synthesize
  - takes order days
  - 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

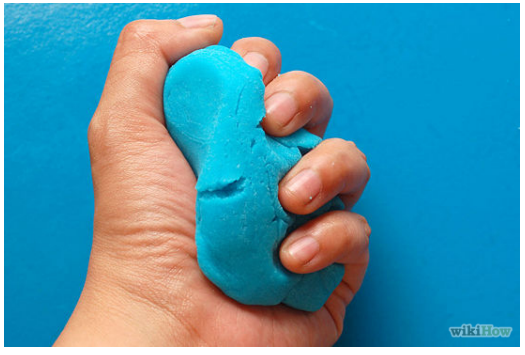
- Software startups would be rare
- CS enrollment would go way down!



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*

Reconfigurable Landscape

- speed / cost / programmability
- what is Reconfigurable Computing?
- what are FPGAs?
- how to program FPGAs?
- FPGA strength and weaknesses
- FPGA system architectures
- how to speed up programmability?
- opportunities

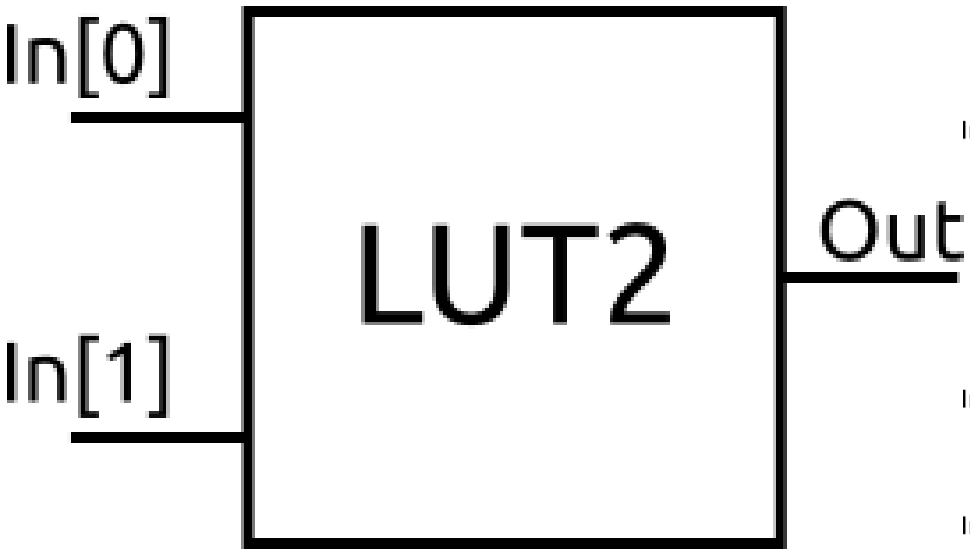
Speed / Cost / Power / Programmability

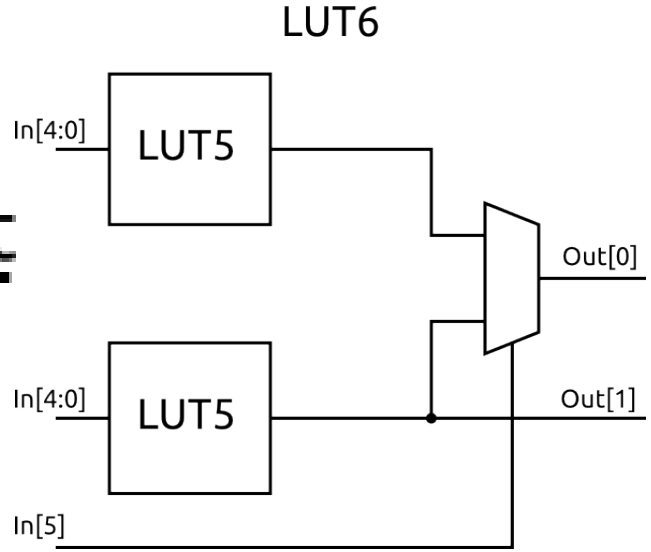
dim	CPU	ASIC
speed	1	100-1000x
cost	1	>1000000x
power	100-1000x	1
prog	>1000x	1

What is Reconfigurable Computing?

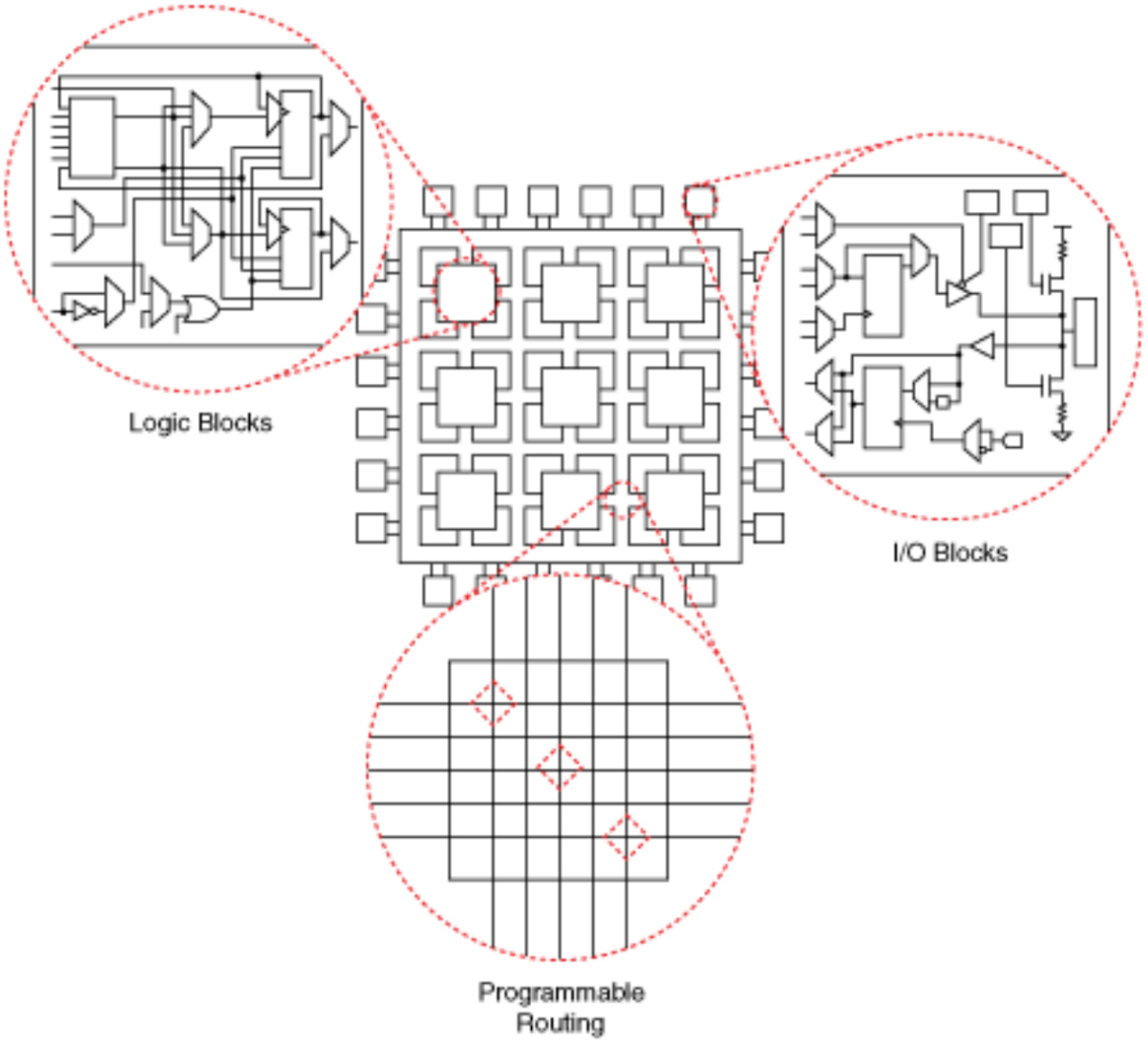
- Reprogrammability
- Higher Speed Lower Power with Less Cost
- Offerings
  - Programmable Logic
    - CPLDs
    - PLA
    - FPGAs
  - Cellular Automata
  - Coarse Grain Reconfigurable Arrays

name	implementation	description
logic	LUTs	4 input lookup tables
state	FLOPs	flip flops
wires	CLBs	circuit switched network
io	IO blocks	dir / voltage programmable





What are FPGAs?



name	implementation	description
arith	DSPs	add + mul (e.g., 48b)
state	Block Rams	larger memories (e.g., 20KB)
logic	Hard Blocks	memory controllers
sla	CPUs	control fabric
arith	FPUs	floating point

- Design Circuit in Verilog
- Configure Chip with functions, clocks and pins
- Place and Map circuit elements on FPGA elements
- Route signals between elements

FPGA Strengths

Good for

- lots of parallelism
- lots of memory access
- bit level manipulations or low bit width
- hard real time
- low power
- embedded

FPGA Weaknesses

Bad because

- extremely slow to program
- painful to debug
- hard to interoperate with
- double floating point
- bandwidth intensive compute
- proprietary and quirky tools

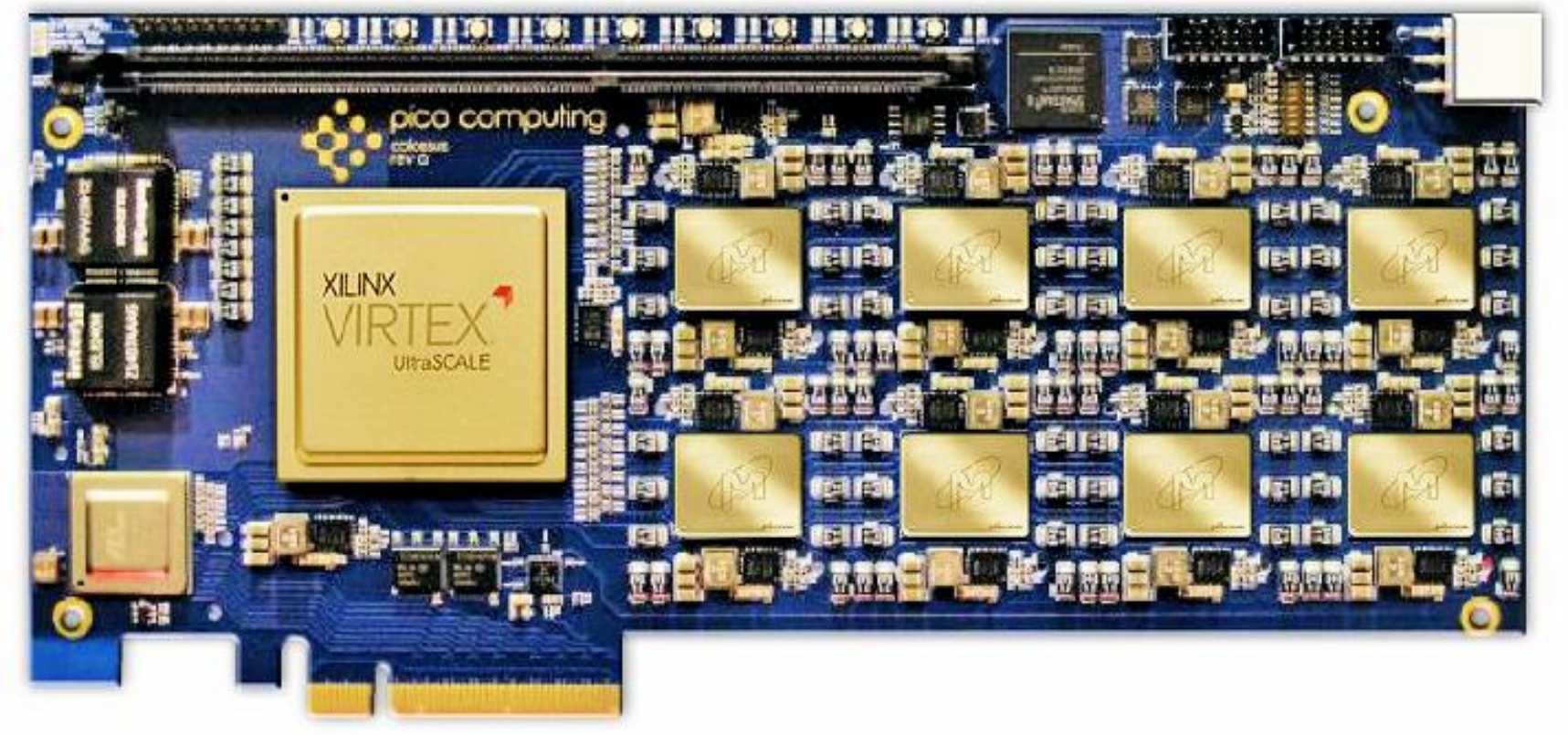
Speed / Cost / Power / Programmability

dim	CPU	FPGA	ASIC
speed	1	1-100x	100-1000x
cost	1	10-100x	>1000000x
power	100-1000x	10-100x	1
prog	>1000x	>10x	1

- how do you control FPGA?
- how do you talk to FPGA?

- ingredients
  - GPIO
  - Mem Controller
  - Serdes
- how to control?
  - you don't: just state machines
  - connect with embedded controller

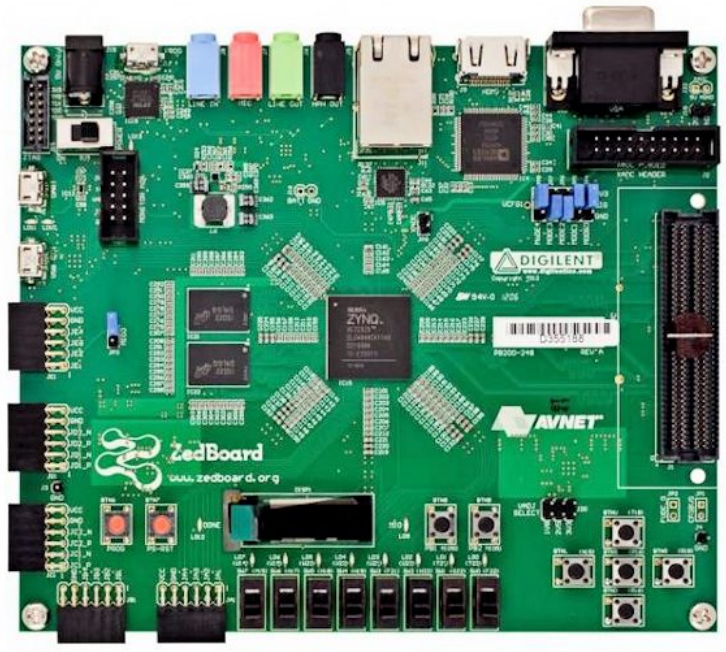
PicoComputing FPGA



- ingredients
  - soft processor
- how to control?
  - processor talks to fabric through various means
  - vendor specific cores and tools

- ingredients
  - hard processor
  - FPGA
- how to control?
  - network
  - memory

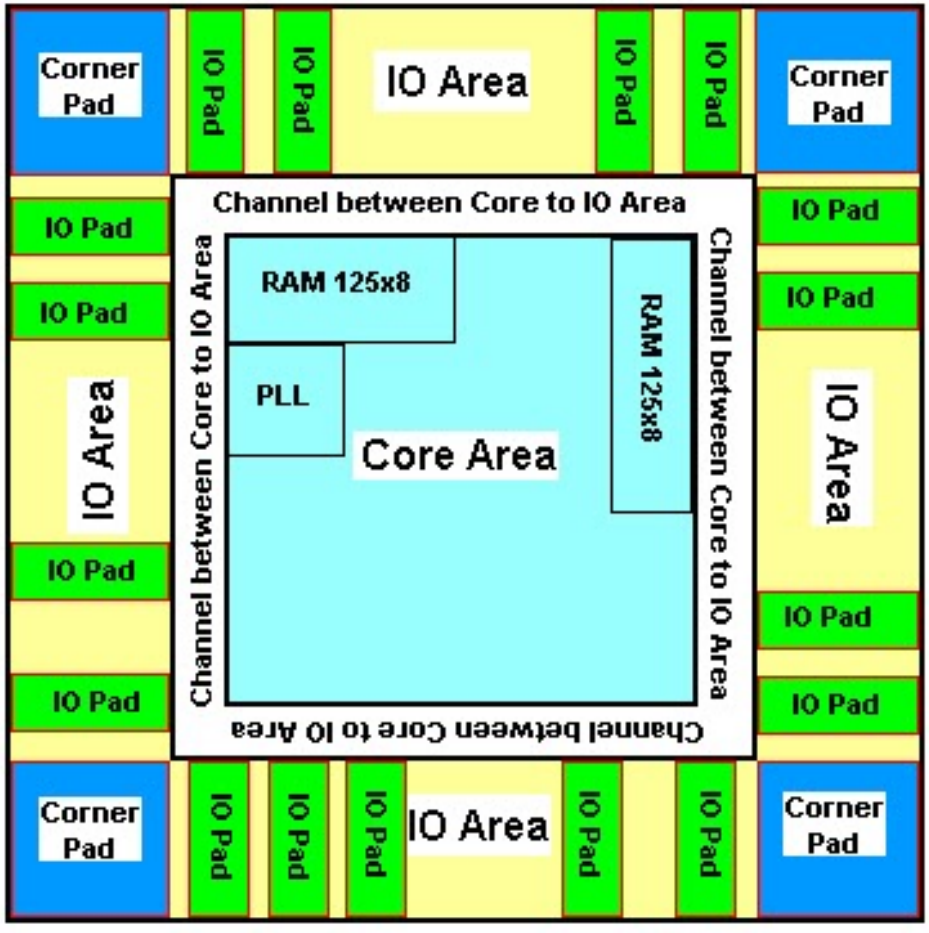
kind	name	throughput	latency	coherent	virtual
net	AXI / ZYNQ	low	high	no	no
net	PCIe / pico	high	high	no	no
mem	ACP/L2 ZYNQ	high	lower	yes	no
mem	ACP/DRAM ZYNQ	higher	low	yes	no
mem	QPI / HARP	higher	lower	yes	yes



- peek/poke interface with AXI lite
- DMA over streaming interface with device driver
- mem mapping with physical addresses

- write CSR words
- memory based locking primitives (AMOs)
- page pinning
- cache line access
- random access

- partial programming
- floor planning
- overlays

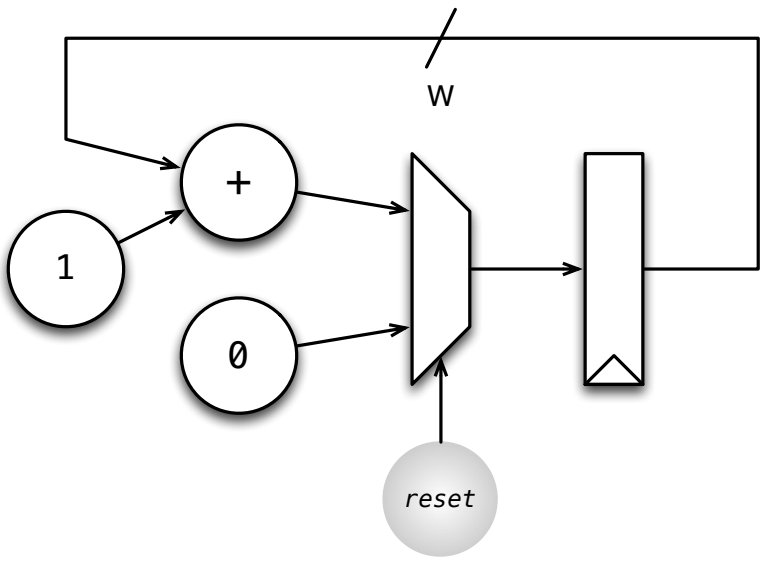


Future Reconfigurable Computing?

- Hard Core with QPI
- Coarse Grain Reconfigurable Arrays
- Open Source FPGA?

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

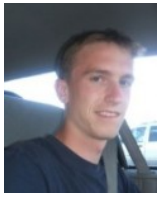
```
module counter (clk, reset);
  input clk;
  input reset;
  parameter W = 8;
  reg [W-1:0] cnt;
  always @ (posedge clk)
  begin
    if (reset)
      cnt = 0
    else
      cnt = cnt + 1
  end
endmodule
```



Berkeley Chisel Team

jonathan bachrach

chris celio

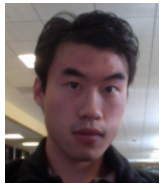


henry cook

palmer dabbelt

adam izraelivitz

donggyu kim

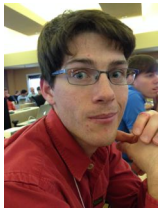


patrick li

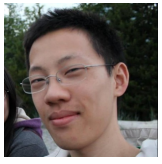
yunsup lee

richard lin

jim lawson



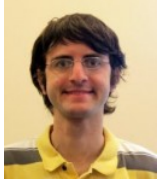
albert magyar



howie mao

colin schmidt

danny tang



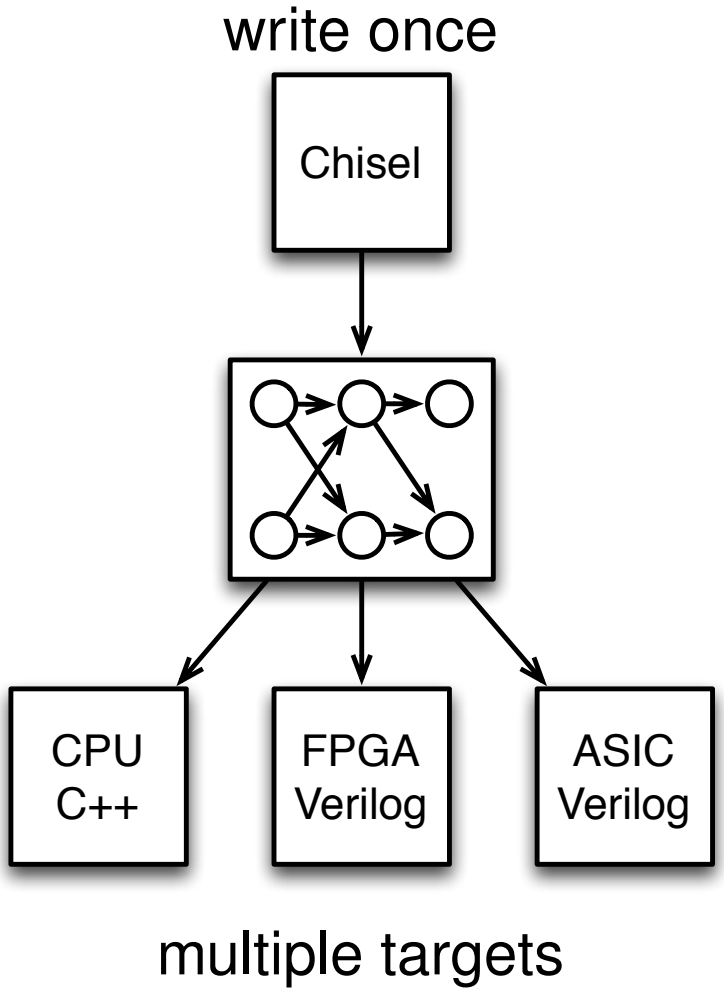
stephen twigg

andrew waterman

john warzynek

krste asanovic

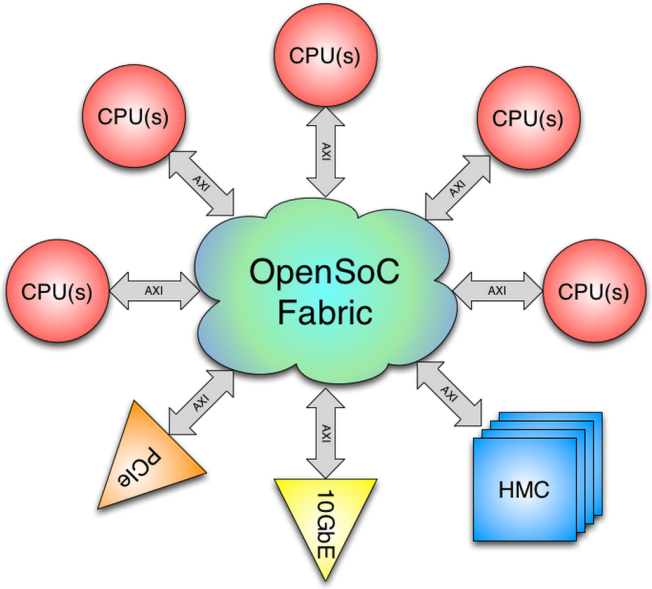
- A hardware construction language
  - “synthesizable by construction”
  - creates graph representing hardware
- Embedded within Scala language to leverage mindshare and language design
- Best of hardware and software design ideas
- Multiple targets
  - Simulation and synthesis
  - Memory IP is target-specific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



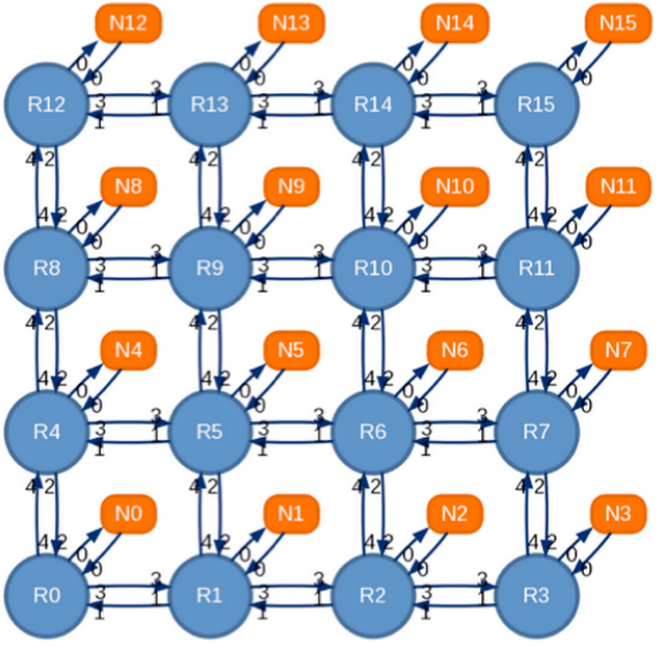
- Open Source Berkeley ISA
- Supports GCC, LLVM, Linux
- Accelerator Interface
- Growing Adoption – LowRisc etc
- Rocket-Chip is Processor Generator in Chisel

http://www.riscv.org

- NOC
- Memory Blocks
- IO / AXI interfaces



SOC



NOC

http://opensocfabric.lbl.gov/

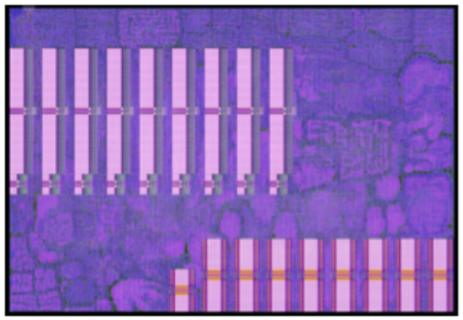
Other Generators

- ZSCALE – lee ...
- BOOM – celio
- FFT – twigg
- Radio – rigge ...
- Spectrometer – bailey
- Sha-3 – schmidt + izraelivitz + ...
- CS250 accelerators – berkeley EECS graduate students
- many more

Our Story

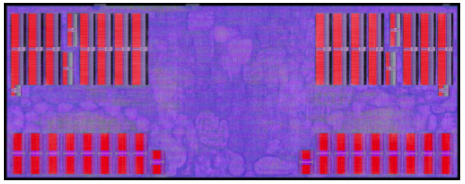
- six chips taped out one every six months
- small teams of (6-8) graduate students
- using our own design software and agile hardware design approach

eos-14



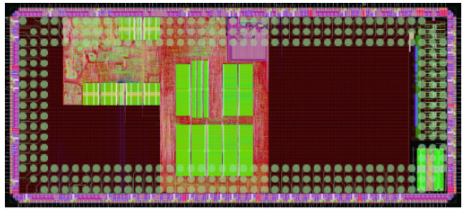
2011-04-01, IBM 45nm, 1GHz+

eos16



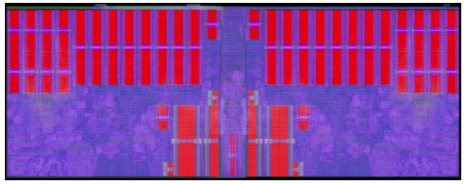
2012-08-17, IBM 45nm, 1GHz+

raven-2



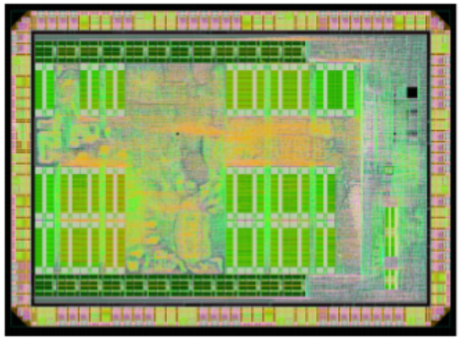
2012-08-22, ST 28nm, 900MHz+

eos18



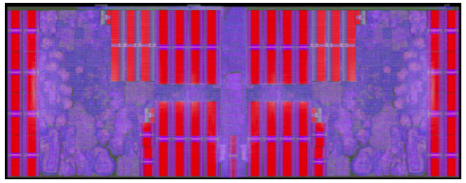
2013-02-06, IBM 45nm, 1.5GHz+

raven-3



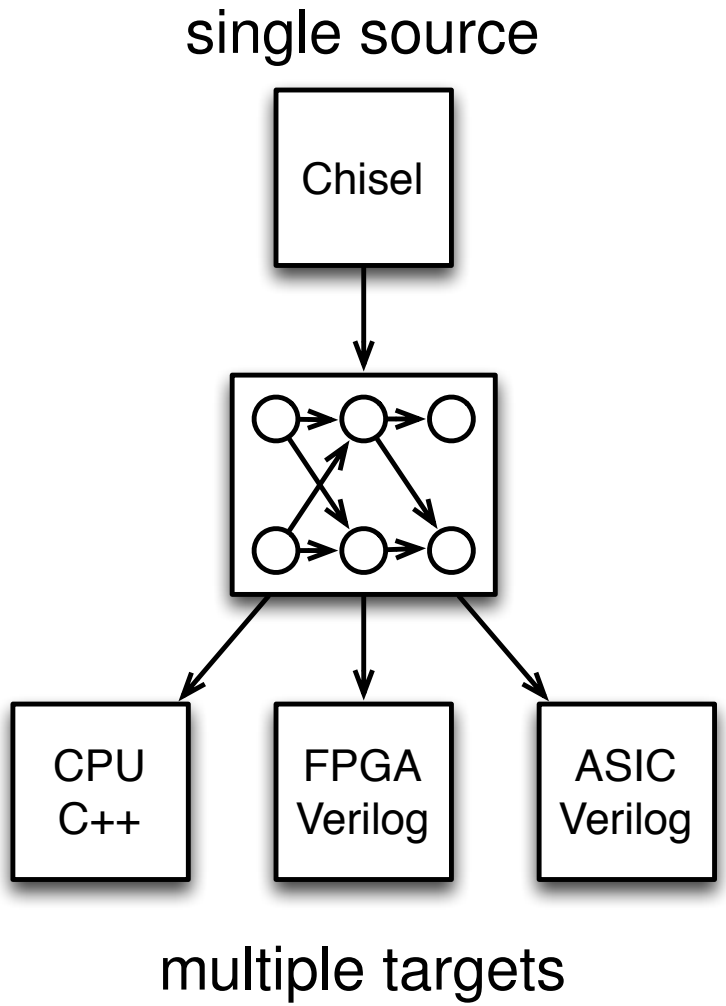
2013-09-26, ST 28nm, 1GHz+

eos20



2013-07-03, IBM 45nm, 1.5GHz+

- 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



Stanza* Essence

- best of scripting and production languages
  - easy to understand and powerful to use
  - gradual types -> easy parameteric types
- simple orthogonal concepts
  - functions, objects, pipes, and namespace separated
  - use concepts in unlimited ways – serendipity
  - entire language including optimizing native compiler in 20K LOC
- powerful macros for conventional syntax
  - almost entire stanza syntax written as macros
  - better DSL hosting language

* developed by Patrick Li @ EECS Berkeley

Stanza by Contrast

- like Python but
  - types and on and on and on ...
- like Scala but
  - thinner – native compiler + runtime in 20K LOC
  - simpler – fewer base concepts
  - more separated – orthogonal functions / objects
- like Clojure but
  - conventional syntax – love the parens but ...
  - thinner – native
  - more powerful macro system – not just name macros
- like Dylan but
  - improved gradual types – parameteric types
  - better multimethod namespaces – fewer name clashes
  - more powerful macros – syntax written in it
  - has pipes – generalized control flow mechanism

- values
- types
- ...
- macros

Stanza Values


```
;; Ints
println(42)
println(plus(32, 42))
println(32 + 42)

;; Int Type
42 : Int

;; Named Value
val x : Int = 42
val x = 42
```

```
;; Chars
println('A')
to-int('A') ==> 65

;; Strings
println("ABC")
length("ABC") ==> 3
length: (String) -> Int
val s = "ABC"
get(s, 0) ==> 'A'
s[0] ==> 'A'
```

```
;; Variables
var x = 42

;; if statement
if 10 < 20 :
  x = 2
else :
  x = 4

;; while statement
var i = 0
while i < 10 :
  println(i)
  i = i + 1

;; for statement
for i in 0 to 10 do :
  println(i)
```

Stanza Collections


```
;; Range
val els = Range(0, 5, 1, false)
val els = 0 to 5

;; Array's
val tbl = Array(256, 0)
tbl[0] = 32
val y = tbl[0]
val n = length(tbl)

;; Vector's
val buf = Vector()
add(buf, 12)
val z = buf[0]
val l = length(buf)
```

```
;; Tuples
val els = [1, 2, 3]
val [a, b, c] = els
val m = length(els)

;; List's
val els = list(1, 2, 3)
val a = head(els)
val b = head(tail(els))
val m = length(els)
```

```
;; Parametric Array's
val tbl = Array<Int>(256, 0)

Array: <T> . (Int) -> Array<T>

;; Captured types
get: <?T> . (Array<?T>, Int) -> T
set: <?T> . (Array<?T>, Int, T) -> T
defn reverse<?T> (xs: Array<?T>) -> Array<T> : ...
```

```
val s = to-stream(0 to 5)
;; supports more? and next

while (more?(s)) :
println(next(s))

;; Streamable === implements to-stream
to-stream: (Range) -> Stream<Int>
to-stream: (String) -> Stream<Char>
to-stream: <?T> . (Array<T>) -> Stream<T>
to-stream: <?T> . (Stream<T>) -> Stream<T>
```

Streamable Iteration

```
val tbl = Array<Int>(256, 0)

;; loop over all indices
for i in 0 to length(tbl) do :
  tbl[i] = i

;; loop of each sequence element
var sum = 0
for e in tbl do :
  sum = sum + e

;; parallel loop
for (e in tbl, i in 0 to false) do :
  tbl[i] = e + 10

;; create second table with doubled elements
val tbl2 = generate<Int> :
  for i in 0 to 16 do: yield(tbl[i] * 2)
```

```
;; simple scaling function, e.g., x2(3) => 6
defn x2 (x:Int): 2 * x
```

```
val xs = [1, 2, 3]
;; produce list of 2 * elements
map(x2, xs)              ==> [2, 4, 6]
map(fn (x): 2 * x, xs) ==> [2, 4, 6]
map({2 * _}, xs)         ==> [2, 4, 6]
```

```
;; sum all elements using pairwise reduction, e.g., sum([1, 2, 3]) => 6
reduce(plus, 0, xs) ==> 6
defn sum (xs: Streamable<Int>) : reduce(plus, 0, xs)
```

Stanza Multis and Methods

```
defmulti deposit (amount:Int, account:String|Int) -> Int

defmethod deposit (amount:Int, account:Int) -> Int :
  println-all(["DEPOSITING" amount " in " count])

defmethod deposit (amount:Int, account:String) -> Int :
  println-all(["DEPOSITING" amount " in " count])
```

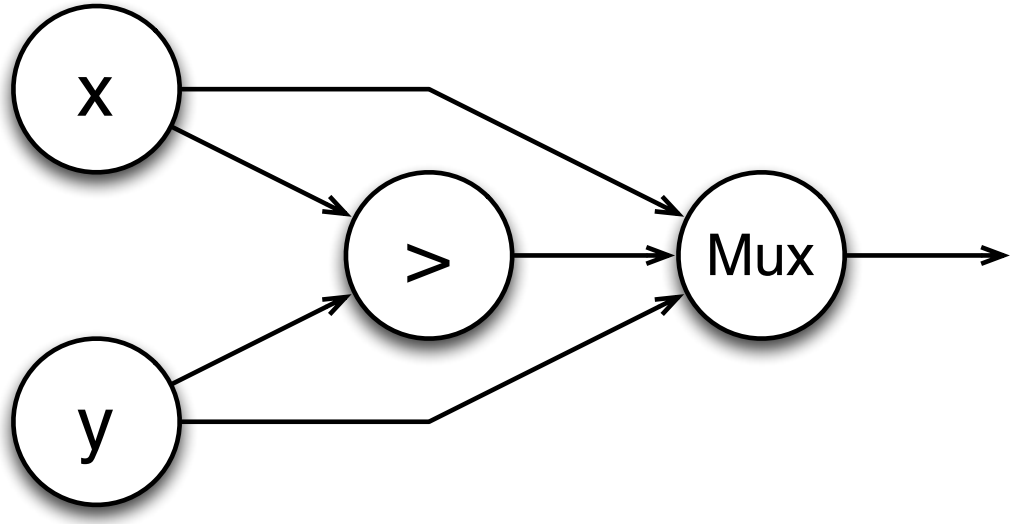
```
definterface Blimp
defmulti rad (b:Blimp) -> Float
defn Blimp (r:Float) :
  println("Another Blimp")
  new Blimp :
    defmethod rad (this) : r

definterface Zep <: Blimp
defmulti hydogren? (b:Zep) -> True|False
defn Zep (r:Float, h?:True|False) :
  new Zep :
    defmethod rad (this) : r
    defmethod hydogren? (this) : h?
```

More Stanza

for more information, check out

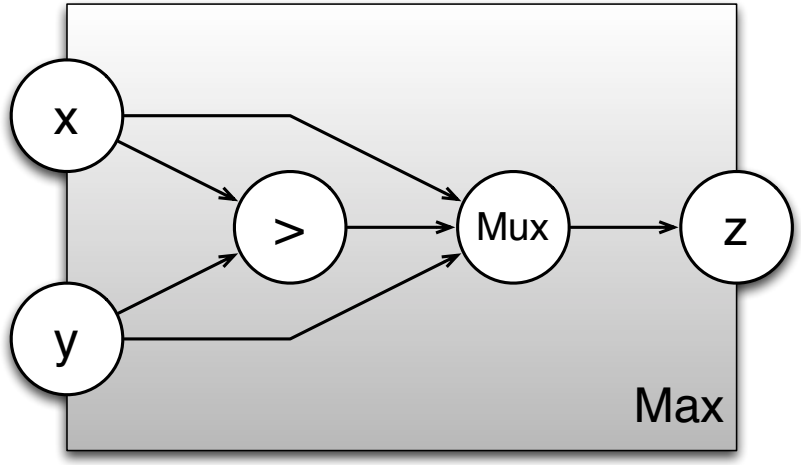
- `http://www.lbstanza.org`
- “stanza by example” online by Patrick Li
- `stanza/core/Core.stanza` + `stanza/core/Verse.stanza` for API
- major release end of this summer
- papers to come on type, macro, coroutine systems

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



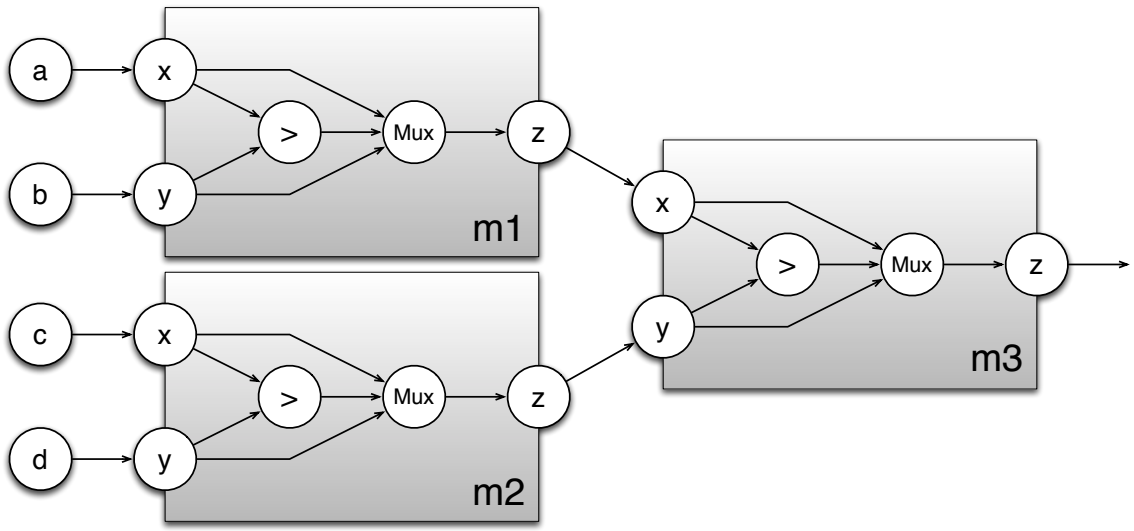
Creating Module


```
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

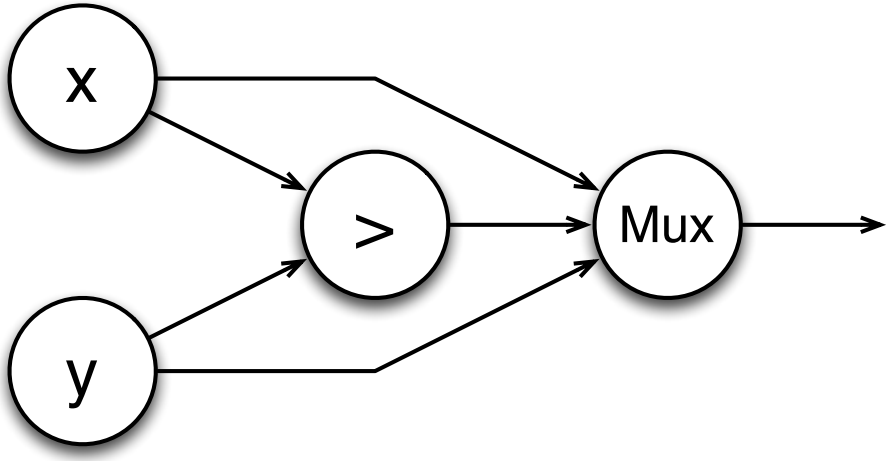

```
inst m1 : Max2()
m1.x := a
m1.y := b
inst m2 : Max2()
m2.x := c
m2.y := d
inst m3 : Max2()
m3.x := m1.z
m3.y := m2.z
```



Defining Construction Functions

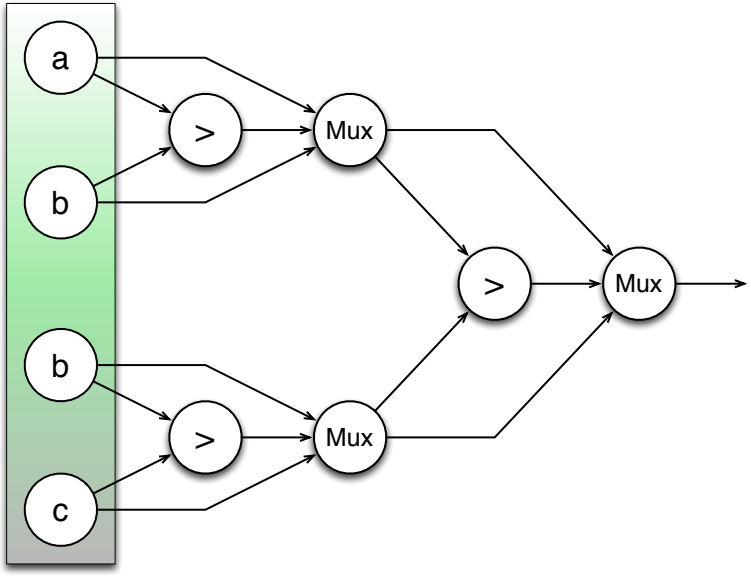

```
defn max2 (x, y): mux(x > y, x, y)
```

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



Functional Construction


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



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
cd ../chipper
git pull
```

```
cd $TUT_DIR/examples
make ByteSelector.out
```

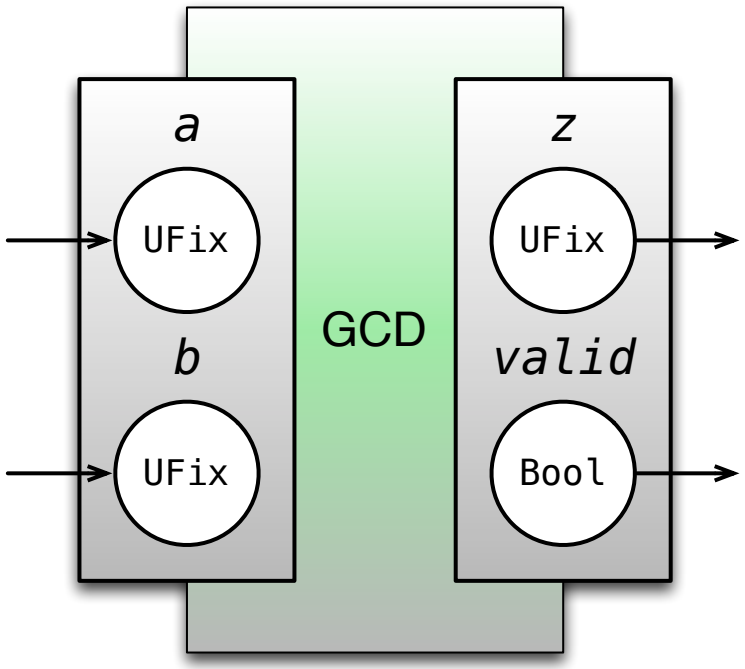
If your system is set up correctly, you should see a messsage `SUCCESS`

Get This


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

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
```

```
...
SUCCESS
```

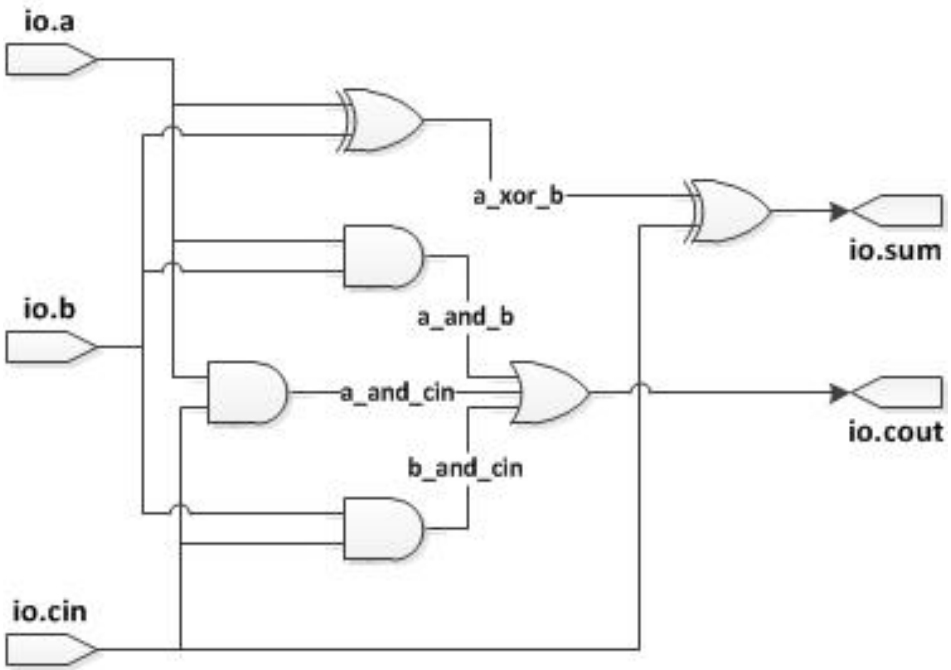
```
cd ~/chipper-tutorial/examples
make GCD.v
```

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
```

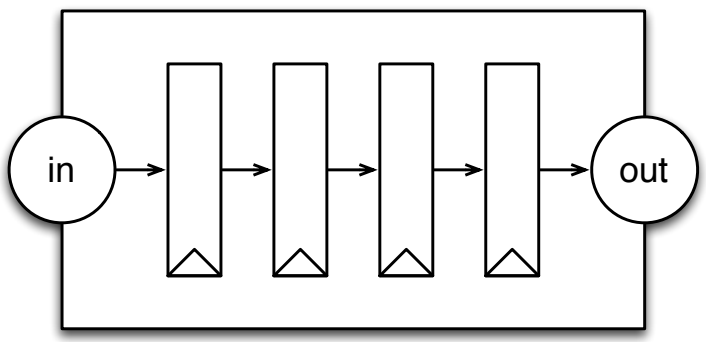
```
module FullAdder(
    input [1:0] io_a,
    input [1:0] io_b,
    input [1:0] io_cin,
    output[1:0] io_sum,
    output[1:0] io_cout);
  wire[1:0] T0;
  wire[1:0] a_and_cin;
  wire[1:0] T1;
  wire[1:0] b_and_cin;
  wire[1:0] a_and_b;
  wire[1:0] T2;
  wire[1:0] a_xor_b;

  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
```

```
// 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

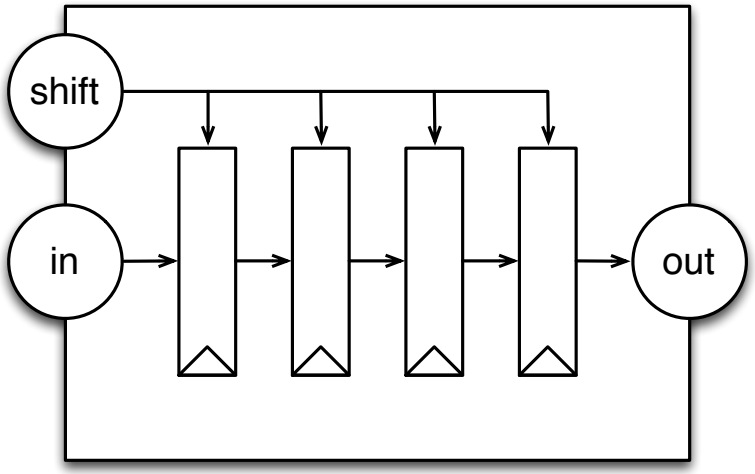

```
defmodule ShiftRegister :
  input in : UInt<1>
  output out : UInt<1>
  reg r0 := in
  reg r1 := r0
  reg r2 := r1
  reg r3 := r2
  out := r3
```




```
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 ShiftRegisterCond :
  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 ShiftRegisterCondInit :
  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` 72

- 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)
```

```
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
|	Bitwise OR
===	Equal
!==	Not Equal
>	Greater
<	Less
>=	Greater or Equal
<=	Less or Equal

Bit Extraction

```
;; extracts the lo through hi bits of value
wire x_to_y = value[hi, lo]
```

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

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

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

Bit Concatenation and Filling

You concatenating bits using `Cat`:

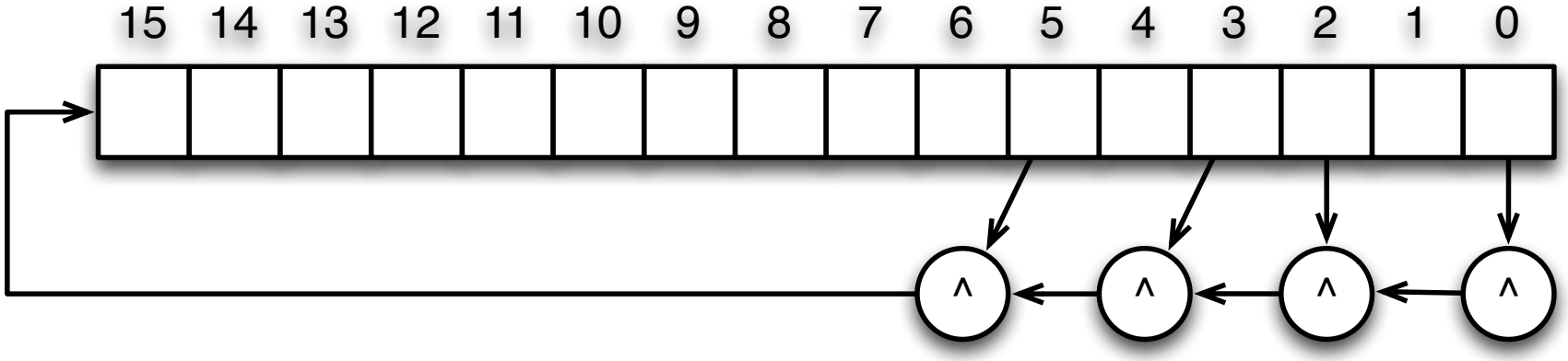
```
wire A   : UInt<32>
wire B   : UInt<32>
wire bus = cat(A, B) ;; concatenate A and B
```

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, [], ^
- 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

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

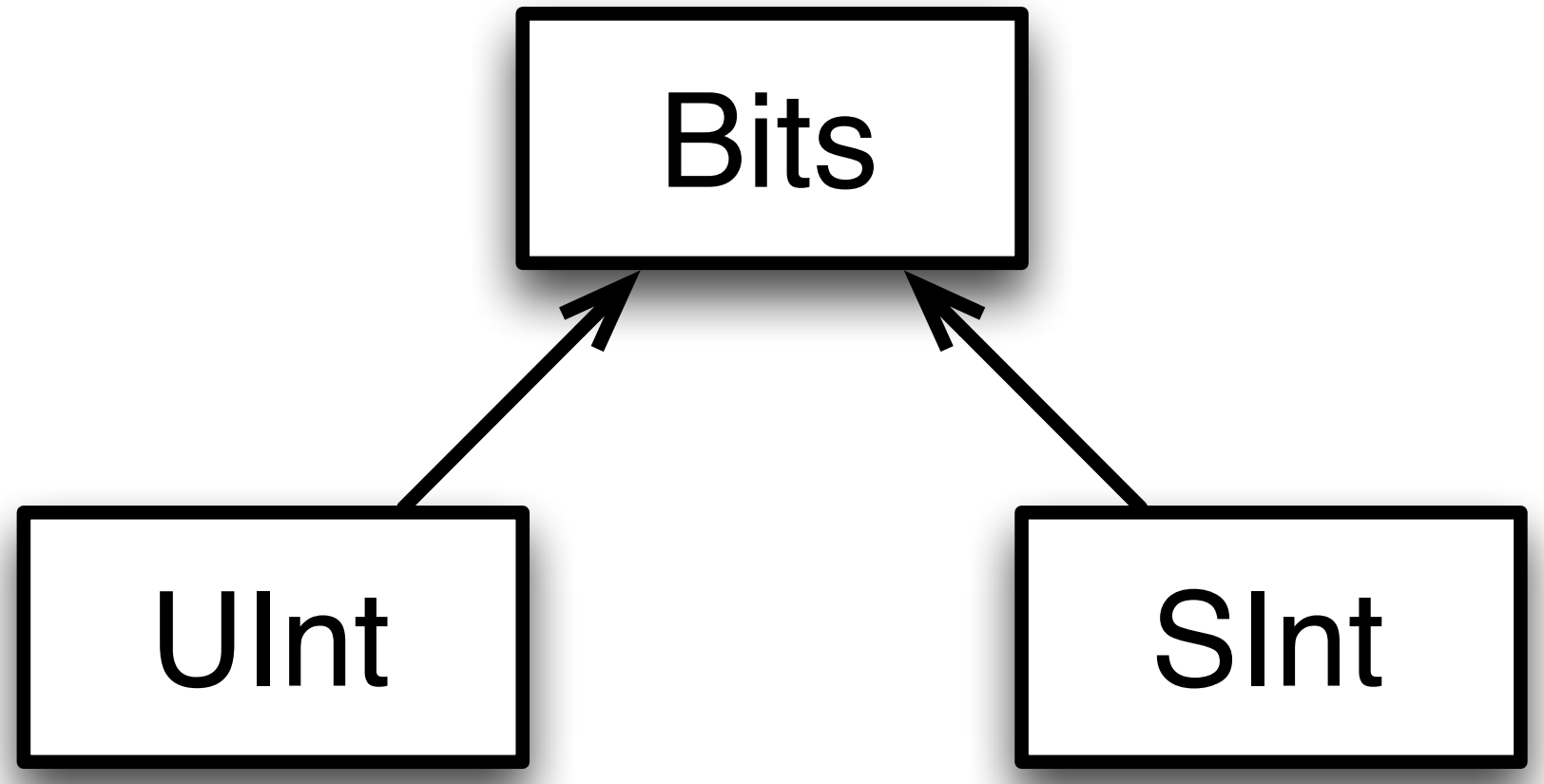
```
wire change = io.a === io.b ;; change gets Bool type
when change : ;; execute if change is true
  ...
```

You can instantiate a Bool value like this:

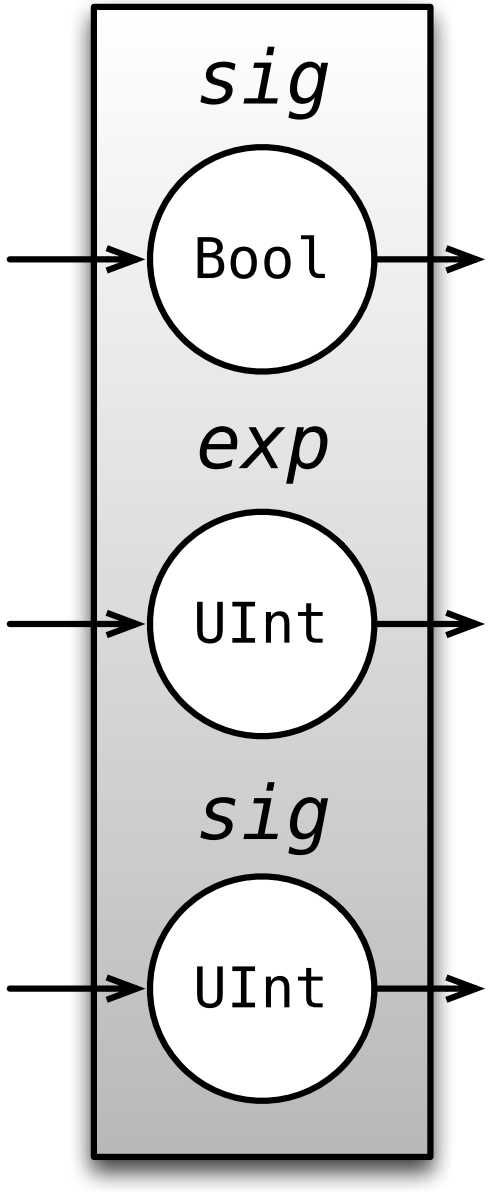
```
wire true_value  = UInt(true)
wire false_value = UInt(false)
```

Bits Subtype Hierarchy

- `SInt` is a signed integer type



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

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

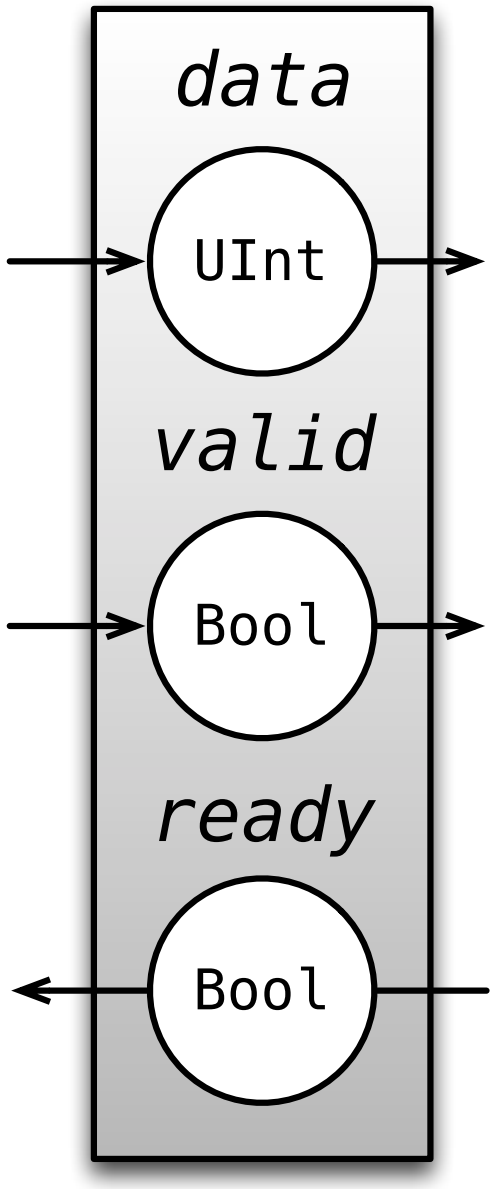


Data object with directions assigned to its members

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

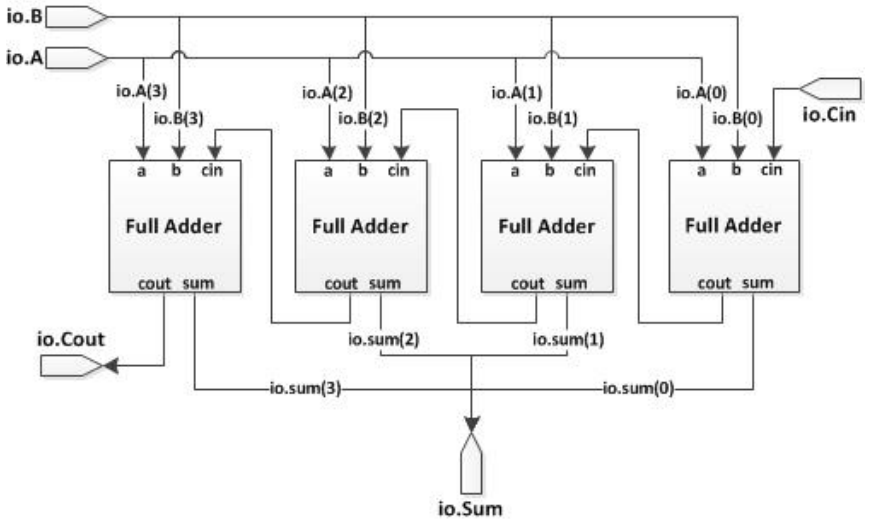
Direction assigned at instantiation time

```
defbundle ScaleIO :
  flip in : MyFloat
  flip scale : MyFloat
  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
```



- defines interface as series of ports,
- 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

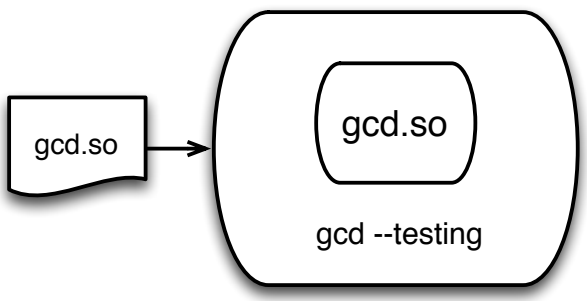
```
wire reg5 = reg_vec[5]
```

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

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

  reg delays: UInt<1>[4]
  when load :
    ;; fill in here ...
  else when shift :
    ;; fill in here ...
  out := delays[3]
```

Defining a Tester


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

defn byte-selector-tests () :
  with-tester [t, c] = ByteSelector() :
    for i in 0 to 8 all? :
      val in  = rand(1 << 31)
      val off = rand(4)
      poke(t, c.in,  in)
      poke(t, c.off, off)
      step(t)
      expect(t, c.out, (in >> (off * 8)) & 255)
```

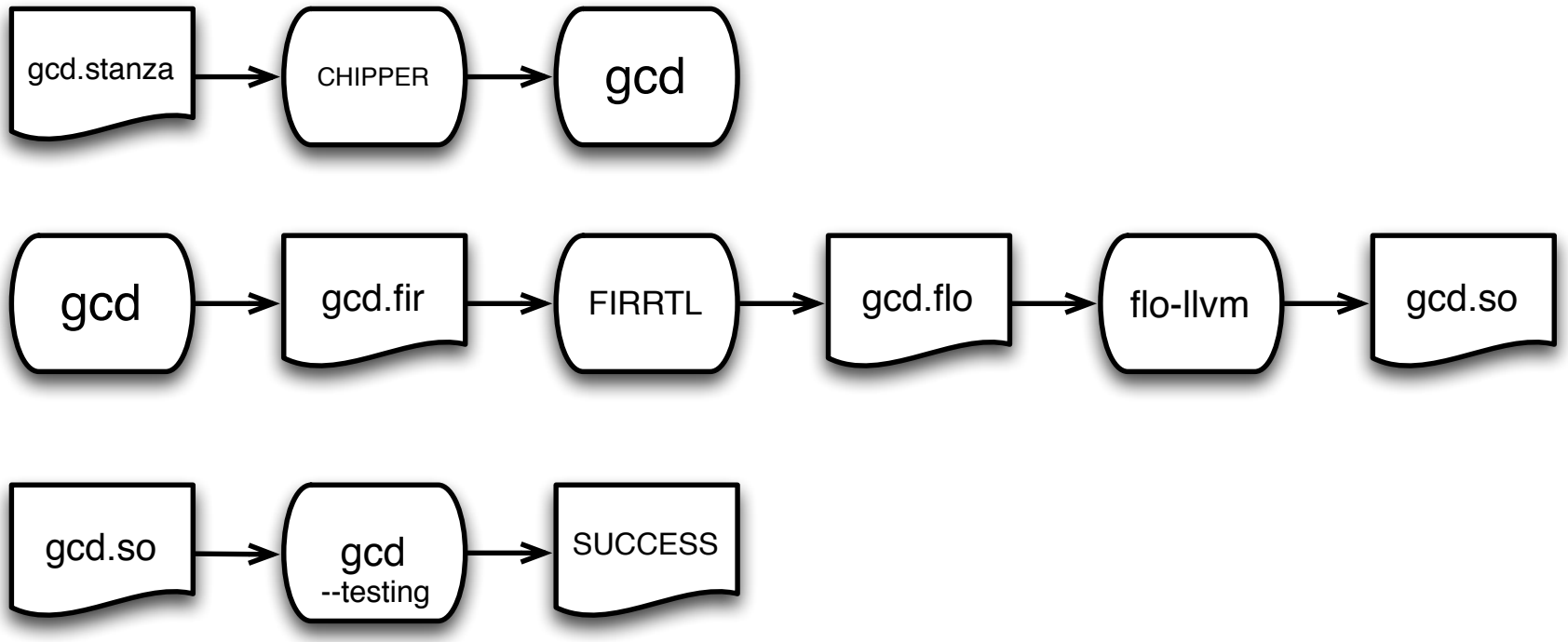



```
defclass Tester
defn Tester (dut:String) -> Tester
defmulti dut (t:Tester) -> String
defmulti step (t:Tester) -> Int
defmulti peek (t:Tester, data:Bits) -> Int
defmulti peek (t:Tester, data:Bits, index:Int) -> Int
defmulti poke (t:Tester, data:Bits, x:Int) -> Int
defmulti poke (t:Tester, data:Bits,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

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.

Chipper Workflow



Simulation Debug Output


```
> cd chipper-tutorial/examples
> make ByteSelector.out
RESET 1 -> 1
WIRE-POKE ByteSelector.in = 0x41a7
WIRE-POKE ByteSelector.off = 0x1
STEP[0] 1 -> 0
WIRE-PEEK ByteSelector.out -> 0x41
EXPECT ByteSelector.out 0x41 -> 65
WIRE-POKE ByteSelector.in = 0x60b7acd9
WIRE-POKE ByteSelector.off = 0x2
STEP[1] 1 -> 0
WIRE-PEEK ByteSelector.out -> 0xb7
EXPECT ByteSelector.out 0xb7 -> 183
WIRE-POKE ByteSelector.in = 0x4431b782
WIRE-POKE ByteSelector.off = 0x0
STEP[2] 1 -> 0
WIRE-PEEK ByteSelector.out -> 0x82
EXPECT ByteSelector.out 0x82 -> 130
...
WIRE-POKE ByteSelector.in = 0x6a5d128c
WIRE-POKE ByteSelector.off = 0x2
STEP[6] 1 -> 0
WIRE-PEEK ByteSelector.out -> 0x5d
EXPECT ByteSelector.out 0x5d -> 93
WIRE-POKE ByteSelector.in = 0x6d7d4b3
WIRE-POKE ByteSelector.off = 0x3
STEP[7] 1 -> 0
WIRE-PEEK ByteSelector.out -> 0x06
EXPECT ByteSelector.out 0x6 -> 6
SUCCESS
```

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 = rand(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 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)
  ;; DEFINE MEM HERE
  wire elt  = UInt(0)
  wire isDone = !(isRd | isWr) & ((elt === data) | (index === UInt(n - 1)))
  ;; FILL IN HERE
  when isRd :
    index := UInt(0)
  else when !(isDone) :
    index := index + UInt(1)
  done    := isDone
  rdAddr  := index
```

RAM is supported using the `cmem` construct

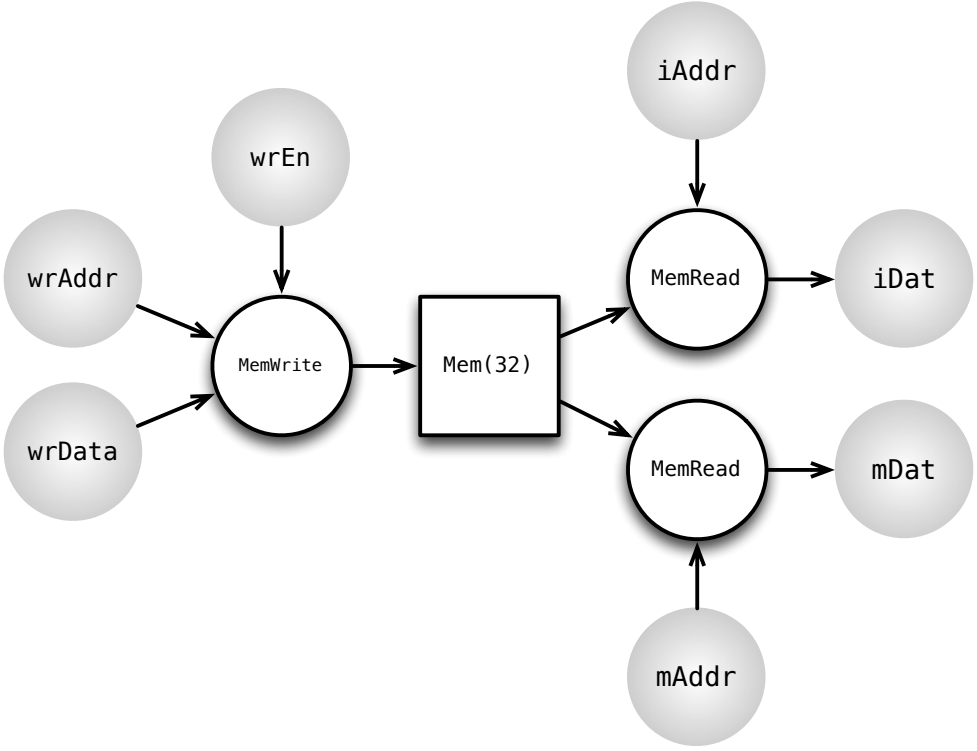
```
cmem m : UInt<32>[32]
```

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 96

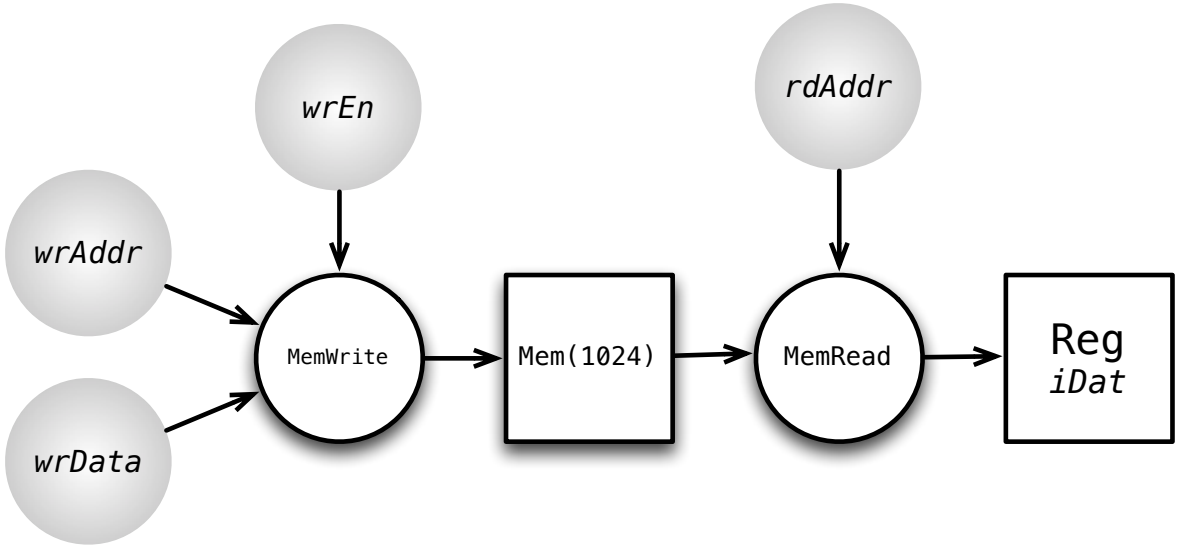
```
defmodule DynamicMemorySearch (n:Int, w:Int) :
  input  isWr:   UInt<1>
  input  wrAddr: UInt<sizeof(w)>
  input  isRd:   UInt<1>
  input  data:   UInt<w>
  output done:   UInt<1>
  output rdAddr: UInt<sizeof(w)>

  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

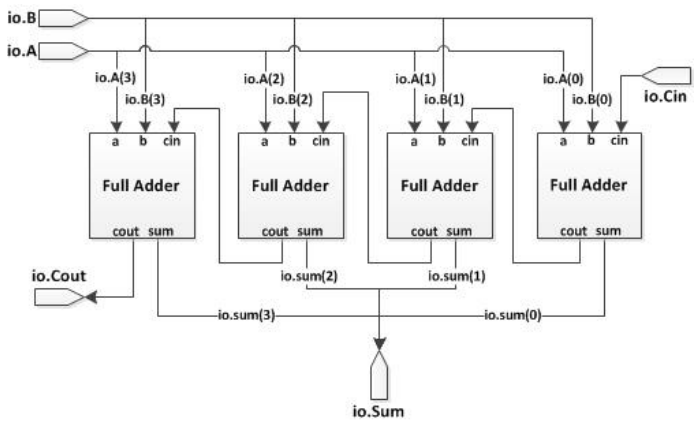
```
smem ram1r1w : UInt<32>[1024]
when wen: ram1r1w[waddr] := wdata
when ren: reg_raddr := raddr
wire rdata = ram1r1w[reg_raddr]
```



```
defmodule Stack (depth: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>[depth]
  reg sp = UInt<sizeof(depth)>(0)
  reg out = UInt<32>(0)

  when en :
    when push & ((sp + UInt(1)) < UInt(depth)) :
      stack_mem[sp] := dataIn
      sp := sp + UInt(1)
    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>

   wire carry: UInt<1>[n + 1]
   wire sums: UInt<1>[n]
   carry[0] := cin
   for i in 0 to n do :
      inst fa : FullAdder
      fa.a := a[i]
      fa.b := b[i]
      fa.cin := carry[i]
      carry[i + 1] := fa.cout
      sums[i] := fa.sum

   sum := reduce(cat, sums)
   cout := carry[n]
```



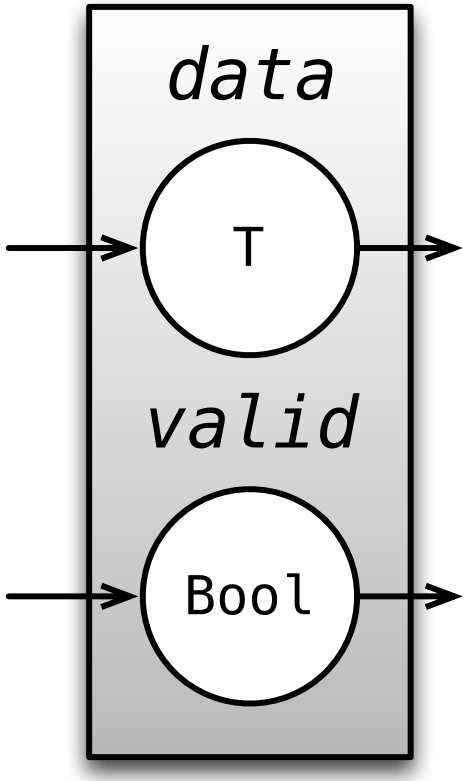
Mul Lookup Table Problem – `Mul.stanza`

- 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]

  ;; CALC Z := x * y BY FILLING IN TAB

  z := UInt(0)
```

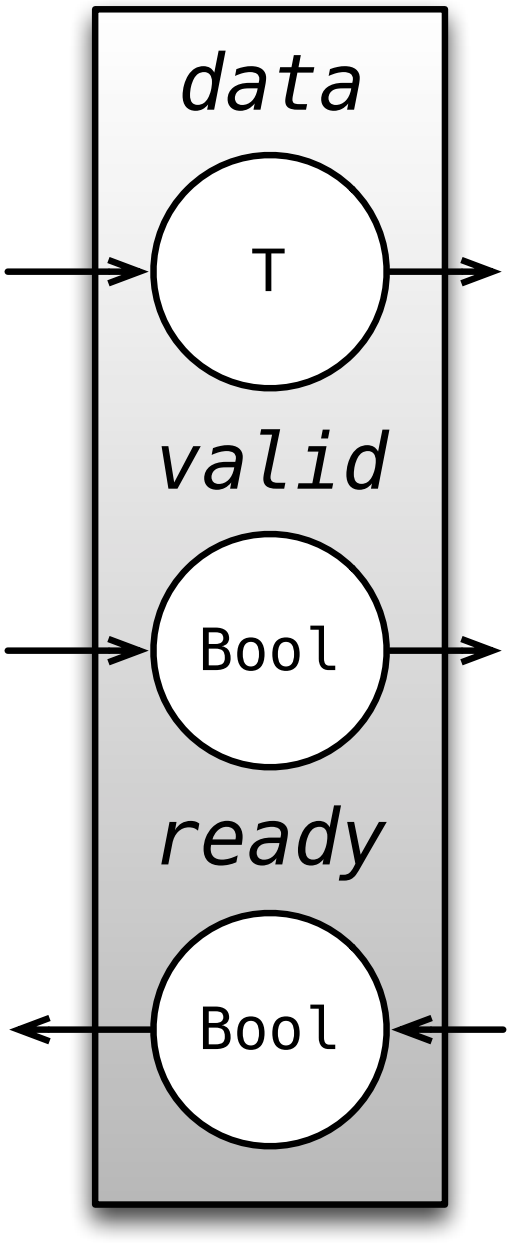
Valid Wrapper


```
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)
```



Decoupled Wrapper


```
defbundle Decoupled<T> :
  data : T
  valid : UInt<1>
  flip ready : UInt<1>

defmodule DecoupledGCD :
   input  a: DecoupledIO<UInt<16>>
   input  b: DecoupledIO<UInt<16>>
   output z: DecoupledIO<UInt<16>>

   reg computing = UInt(false)
   reg x: UInt<16>
   reg y: UInt<16>

   a.ready := !(computing)
   b.ready := !(computing)
   z.bits  := UInt(0)
   z.valid := UInt(false)

   ;; ...
```



- map queues to DecoupledIO's
- sources pop from queue and push onto circuit
- sinks pop data from circuit and push onto queue and
- step now first does needed circuit peek/poke's and queue pop/add's

Sources and Sinks are created and added to tester

```
defn Source<?T> (t:Tester, d:DecoupledIO<?T&Data>) ...
defn Sink<?T> (t:Tester, d:DecoupledIO<?T&Data>) ...
defmulti add-ios (t:Tester, ios:Streamable<SmartIO<Data>>)
```

Sources and Sinks support

- `add` – push onto queue
- `pop` – pop and return top of queue
- `peek` – return top of queue
- `clear` – remove all queue elements
- `length` – size of queue

```
defn decoupled-gcd-tests (n:Int) :
  defn gcd (a:Int, b:Int) -> Int :
    if b == 0: a else: gcd(b, a % b)
  with-tester [t, c] = DecoupledGCD() :
    ;; connect queues to decoupled io
    val as = Source(t, c.a)
    val bs = Source(t, c.b)
    val zs = Sink(t, c.z)
    add-ios(t, [as, bs, zs])
    ;; tests and remember correct answers
    val ezs = Vector<Int>()
    for i in 0 to n do :
      val [a, b] = [rand(2, 256), rand(2, 256)]
      add(ezs, gcd(a, b))
      add(as, [a])
      add(bs, [b])
    ;; wait for circuit to consume inputs and produce all answers
    while length(as) > 0 or length(bs) > 0 or length(zs) < length(ezs) :
      step(t)
    ;; check answers
    for i in 0 to length(ezs) all? :
      val rz = pop(zs)
      expect(rz[0] == ezs[i], ["Sum " ezs[i] " GOT " rz[0]])
```

```
public defbundle SumReq  :
  v   : UInt<32>
  len : UInt<32>

public defbundle SumResp  :
  data : UInt<32>

public defbundle MemReq :
  wr   : UInt<1>
  tag  : UInt<8>
  addr : UInt<32>
  data : UInt<32>

public defbundle MemResp :
  tag  : UInt<8>
  data : UInt<32>
```

```
public defmodule Sum :
  input  sreq: DecoupledIO<SumReq>
  output srsp: DecoupledIO<SumResp>
  input  mrsp: DecoupledIO<MemResp>
  output mreq: DecoupledIO<MemReq>
  reg a   : UInt<32>
  reg ea  : UInt<32>
  reg n   : UInt<32>
  reg sum : UInt<32>
  reg computing  = UInt(false)
  ...
  when computing :
    sreq.ready := UInt(false)
    when mrsp.valid :
      sum := sum + mrsp.bits.data
      n   := n - UInt(1)
    when a < ea :
      mreq.valid := UInt(true)
      mreq.bits.addr := a
      when mreq.ready :
        a := a + UInt(1)
    else when n === UInt(0) :
      srsp.valid := UInt(true)
      srsp.bits.data := sum
      when srsp.ready :
        sum       := UInt(0)
        computing := UInt(false)
  else when sreq.valid :
    a  := sreq.bits.v
    ea := sreq.bits.v + sreq.bits.len
    n  := sreq.bits.len
    computing := UInt(true)
```

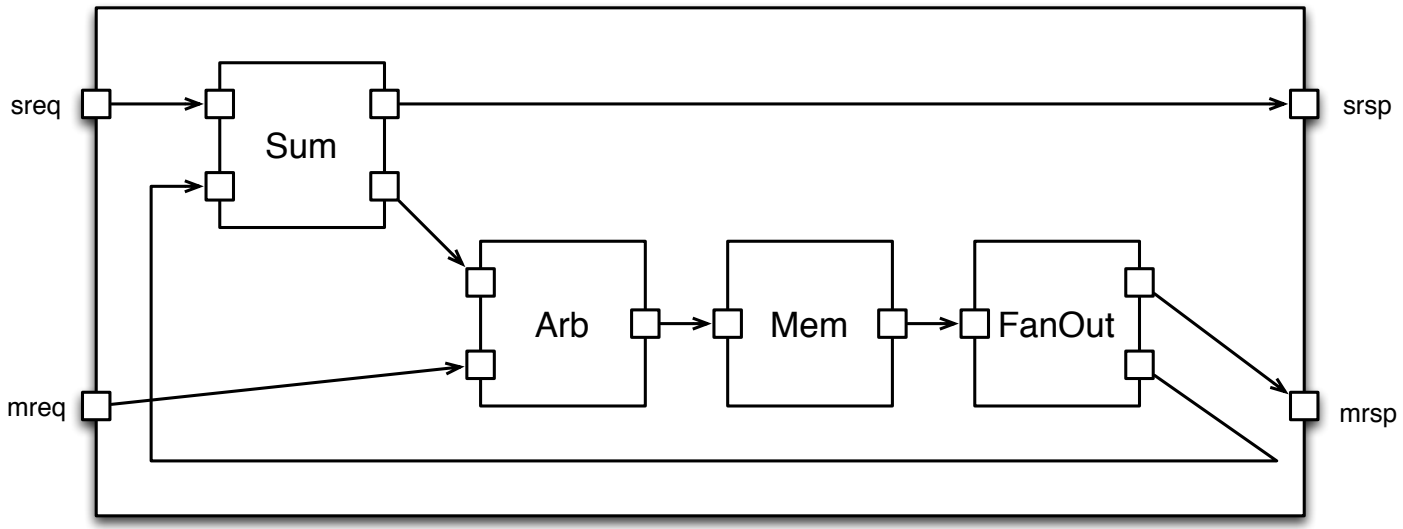
```
defmodule SumCore :
  input  sreq: DecoupledIO<SumReq>
  output srsp: DecoupledIO<SumResp>
  input  mrsp: DecoupledIO<MemResp>
  output mreq: DecoupledIO<MemReq>
  inst sum : Sum
  sum.sreq := sreq
  srsp     := sum.srsp
  mreq     := sum.mreq
  sum.mrsp := mrsp
```

```
defn sum-core-tests (n:Int) :
  with-tester [t, c] = SumCore() :
    val mrsps = Source(t, c.mrsp)
    val mreqs = Sink(t, c.mreq)
    val sreqs = Source(t, c.sreq)
    val srsps = Sink(t, c.srsp)
    add-ios(t, [mreqs, mrsps, sreqs, srsps])
    for i in 0 to 10 all? :
      val v = to-array(stream({ rand(256) }, 0 to (rand(n - 1) + 1)))
      val r = reduce(plus, 0, v)
      add(sreqs, [0, length(v)])
      while length(sreqs) > 0 or length(srsps) < 1 :
        while length(mreqs) > 0 :
          val mreq = pop(mreqs)
          val [wr, tag, addr, data] = [mreq[0], mreq[1], mreq[2], mreq[3]]
          if wr == 0 :
            add(mrsps, [tag, v[addr]])
        step(t)
      val srsp = pop(srsps)
      expect(srsp[0] == r, ["Sum " r " GOT " srsp[0]])
```

```
public defbundle MemReq :
  wr   : UInt<1>
  tag  : UInt<8>
  addr : UInt<32>
  data : UInt<32>

public defbundle MemResp :
  tag  : UInt<8>
  data : UInt<32>
```

```
public defmodule Memory (n: Int) :
  input  req : DecoupledIO<MemReq>
  output rsp : DecoupledIO<MemResp>
  cmem mem : UInt<32>[n]
  req.ready      := rsp.ready
  rsp.valid      := UInt(false)
  rsp.bits.data := UInt(0)
  rsp.bits.tag  := UInt(0)
  when req.valid :
    rsp.valid    := UInt(true)
    rsp.bits.tag := req.bits.tag
    when req.bits.wr :
      rsp.bits.data      := req.bits.data
      mem[req.bits.addr] := req.bits.data
    else :
      rsp.bits.data := mem[req.bits.addr]
```

```
defmodule SumSys (n:Int) :
  input  sreq: DecoupledIO<SumReq>
  output srsp: DecoupledIO<SumResp>
  input  mreq: DecoupledIO<MemReq>
  output mrsp: DecoupledIO<MemResp>
  inst sum   : Sum
  inst mem   : Memory(n)
  val [req, chosen] = arbiter([mreq, sum.mreq])
  fan-out(chosen, mem.rsp, [mrsp, sum.mrsp])
  mem.req     := req
  srsp        := sum.srsp
  sum.sreq    := sreq
```



Borrowing from Compiler Field

- Apple Programming Stack
- NVidia Cuda
- OpenCL
- Cray Chapel
- Haskell
- Many backends

```
define i32 @mul_add
    (i32 %x, i32 %y, i32 %z) {
entry:
  %tmp = mul i32 %x, %y
  %tmp2 = add i32 %tmp, %z
  ret i32 %tmp2
}
```

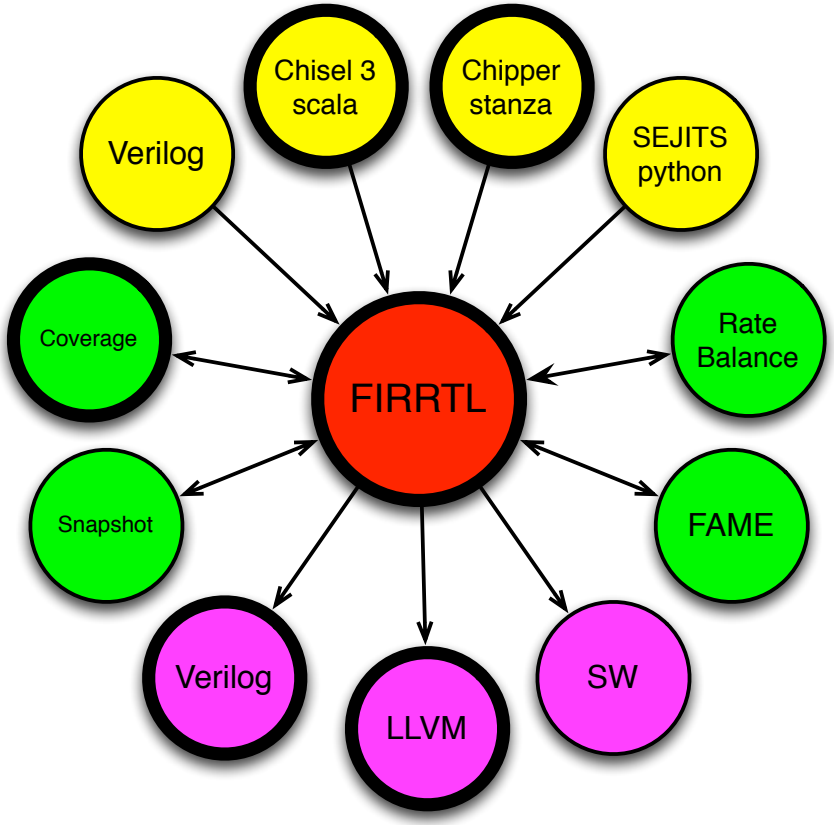
- SSA-based IR
- Primitive RISC ISA
  - Regular instructions
  - Easy to write backends
- Compiler organized as transformations that take LLVM IR and produce LLVM IR
- IR exposed as file format and API
- Supports User-defined Transformations

- **F**lexible **I**ntermediate **R**epresentation for **RTL**
- Focussed Synthesizable RTL
- Precise File Format and API
- Micro Passes for Robustness
- User defined passes
- Chew Off Biggest Piece of Chipper and make it modular

```
circuit GCD :
  module GCD :
    input a : UInt<16>
    input b : UInt<16>
    input e : UInt<1>
    output z : UInt<16>
    output v : UInt<1>
    reg x1 : UInt<16>
    reg y2 : UInt<16>
    node tmp3 = gt(x1, y2)
    when tmp3 :
      node tmp4 = sub-wrap(x1, y2)
      x1 := tmp4
    else :
      node tmp5 = sub-wrap(y2, x1)
      y2 := tmp5
    when e :
      x1 := a
      y2 := b
    z := x1
    node tmp6 = eq(y2, UInt<1>(0))
    v := tmp6
```

EDEN Ecosystem

- Modular
- Language Neutral
  - SEJITS $\rightarrow$ FIRRTL
  - Verilog $\rightarrow$ FIRRTL
  - chisel3 $\rightarrow$ FIRRTL
- Backends
  - LLVM
  - verilog
  - target code
- Transformations
  - FAME
  - Rate Balancing
  - Coverage
  - Snapshot
- Tools
  - Coverage
  - Visualization



Electronic **D**esign of **E**verything **N**icer

- macros
  - creates elegant representation
  - creates scaffolding
- handles with AST IR
  - handles are typed and also point to IR
  - IR collected in circuits and modules
  - IR tree data structure easy to print, read, walk
  - explicit file format

Embedding Interface

chipper

```
definterface CHExp
defmulti firrtl-exp (e:CHExp) -> fExpression
defmulti type<?T> (e:?T&CHExp) -> Type<T>

definterface Type<T>
defmulti firrtl-type (t:Type) -> fType
defmulti handle<?T> (t:Type<?T>, e:fExpression) -> T
```

firrtl

```
defstruct Circuit :
   modules: List<Module>
   main: Symbol
defstruct Module :
   name: Symbol
   ports: List<Port>
   body: Stmt
definterface Stmt
defstruct DefWire <: Stmt :
   name: Symbol
   type: Type
...
defstruct Connect <: Stmt :
   loc: Expression
   exp: Expression
...
definterface Expression
defmulti type (e:Expression) -> Type
defstruct Ref <: Expression :
   name: Symbol
   type: Type
defstruct Subfield <: Expression :
   exp: Expression
   name: Symbol
   type: Type
...
definterface Type
defstruct UIntType <: Type :
   width: Width
...
```

Bundle Expansions

```
defbundle Double :
   a: UInt
   b: UInt
```

... expands to ...

```
defclass Double <: CHExp
defmulti a (d:Double) -> UInt
defmulti b (d:Double) -> UInt

defn Double (e:Expression) :
   new Double :
      defmethod firrtl-exp (this) : e
      defmethod a (this) :
         UInt(Subfield(e, 'a))
      defmethod b (this) :
         UInt(Subfield(e, 'b))

defclass DoubleType <: CHType
defn DoubleType () :
   new DoubleType :
      defmethod handle (this, e:Expression) :
         Double(e)
```

```
defmodule Dual :
   input a: UInt
   output b: UInt
   mybody
```

... expands to ...

```
defclass Dual <: CHExp
defmulti a (d:Dual) -> UInt
defmulti b (d:Dual) -> UInt

defn Dual (e:Expression) :
   new Dual :
      defmethod firrtl-exp (this) : e
      defmethod a (this) :
         UInt(Subfield(e, 'a))
      defmethod b (this) :
         UInt(Subfield(e, 'b))

defclass DualModule <: CHModule
defn DualModule () :
   new DualModule :
      defmethod handle (this, e:Expression) :
         Dual(e)
      defmethod firrtl-module (this) :
         ModuleExp(
            'Dual,
            fn () : mybody)
```

- hardware types are different than software ones
- no type complexity
- no perfect symmetry between compile-time run-time

HW	SW
UInt	BigInt
SInt	BigInt
Vec	Array
Bundle	???

- seamlessness
  - looks like domain
  - no leakage
  - good error reporting
- embedded plus eclipse support
  - want embedding to leverage facilities
  - want automatic tooling
- better hosting language
  - easy to understand
  - powerful
  - base
  - easy/powerful to embed