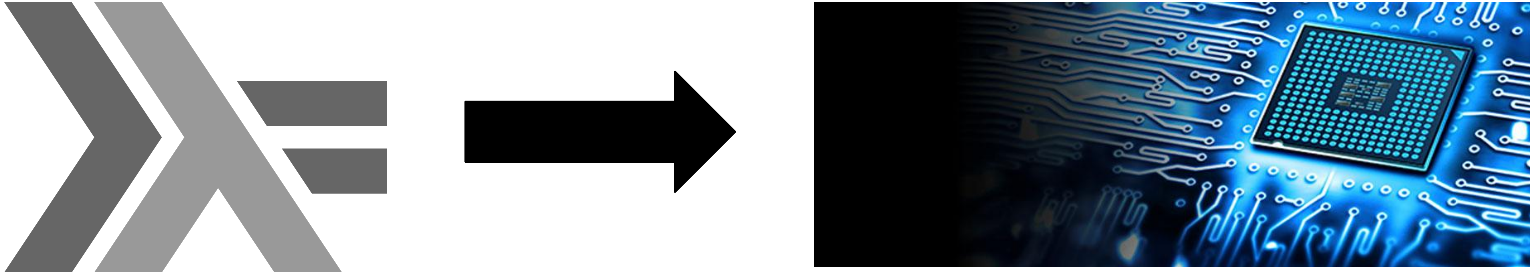
Translating Haskell to Hardware

Lianne Lairmore
Columbia University

FHW Project

Functional Hardware (FHW)

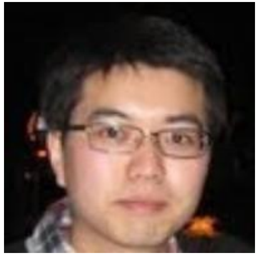


Martha Kim

Stephen Edwards

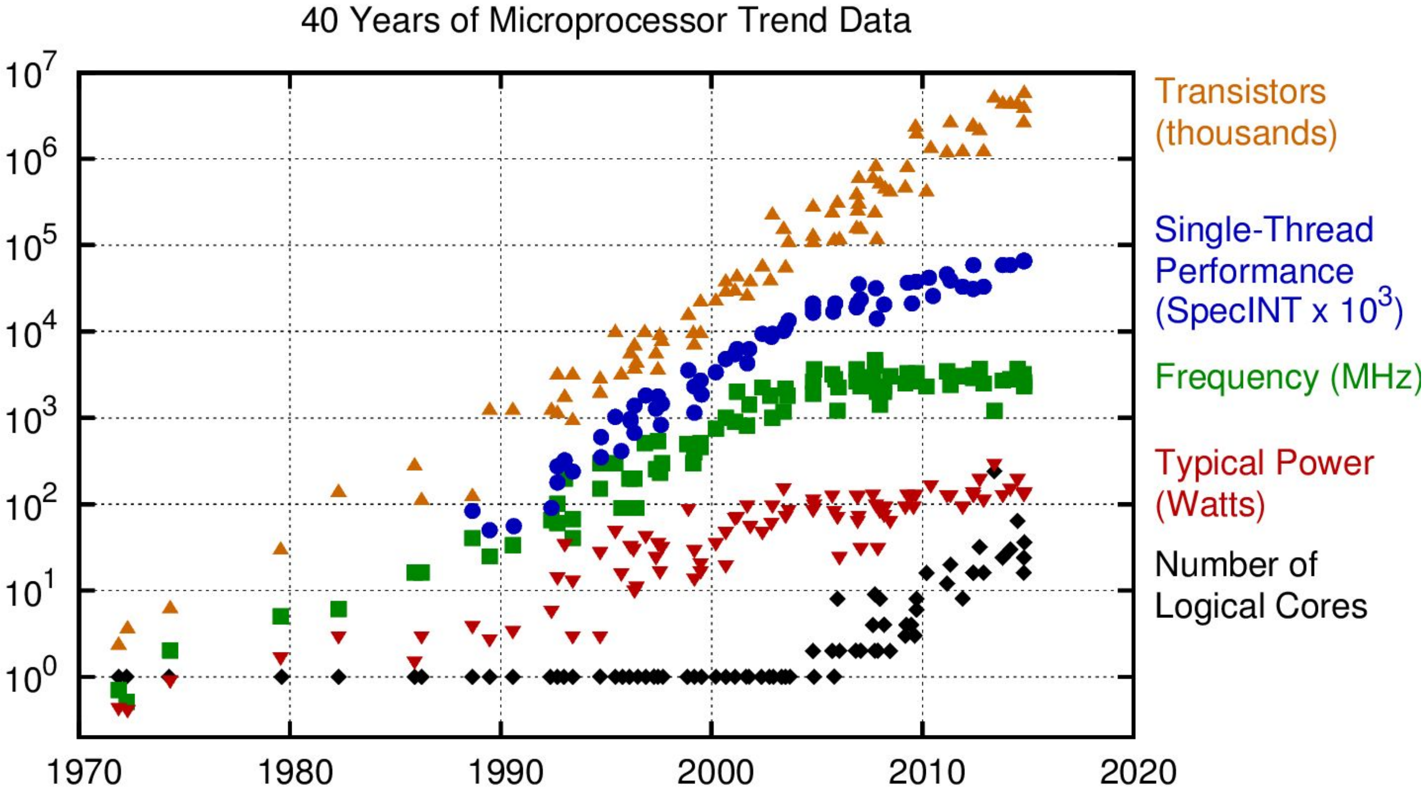
Richard Townsend

Lianne Lairmore



Kuangya Zhai

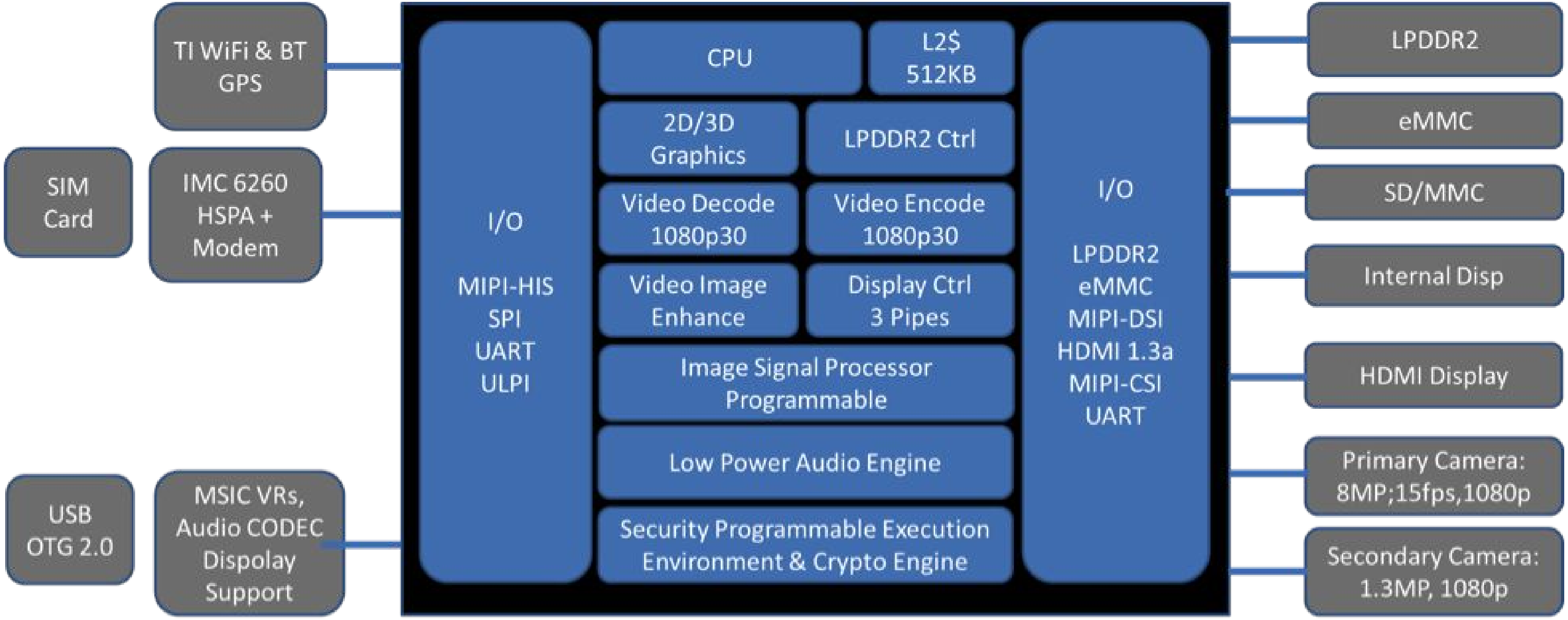
CPUs



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

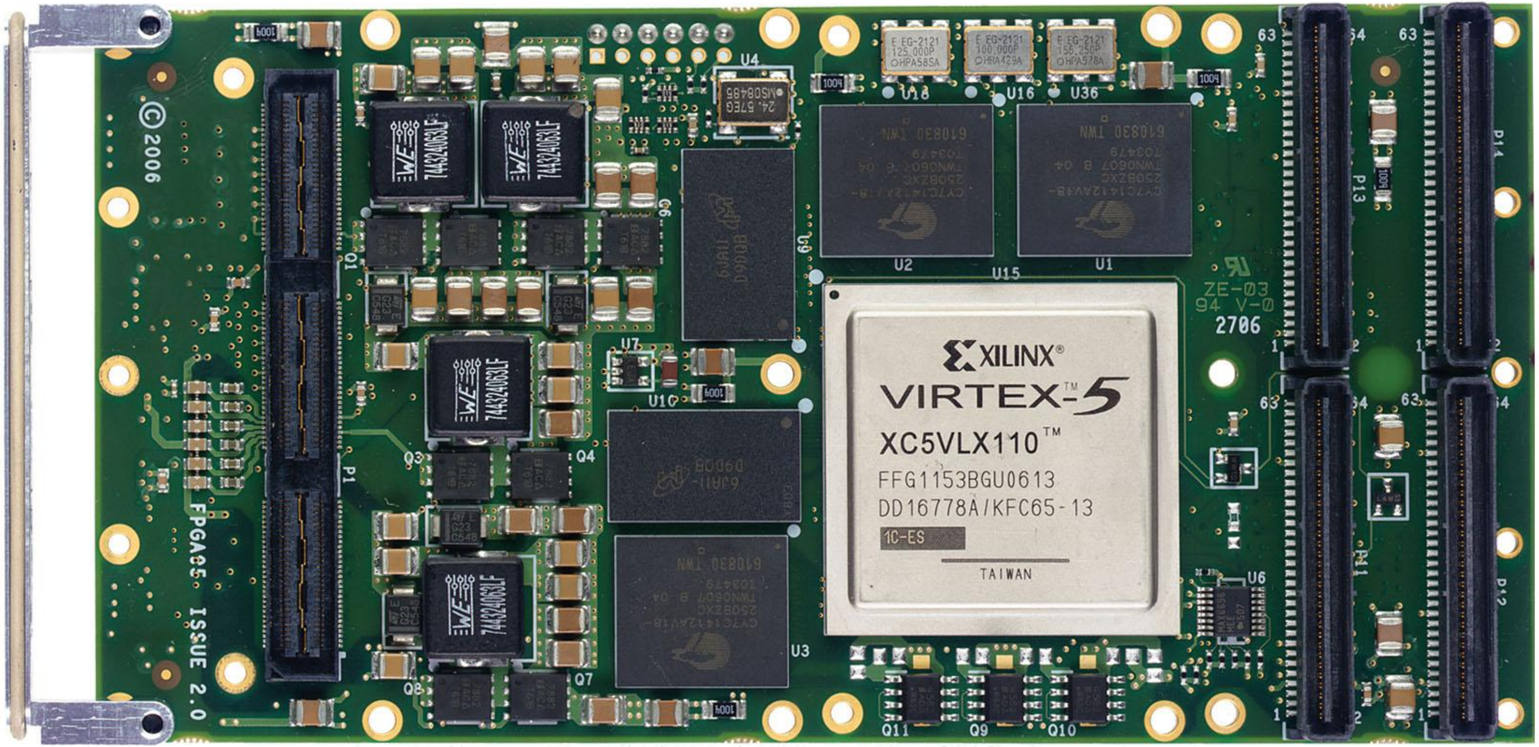
Power Consumption

Dedicated Hardware



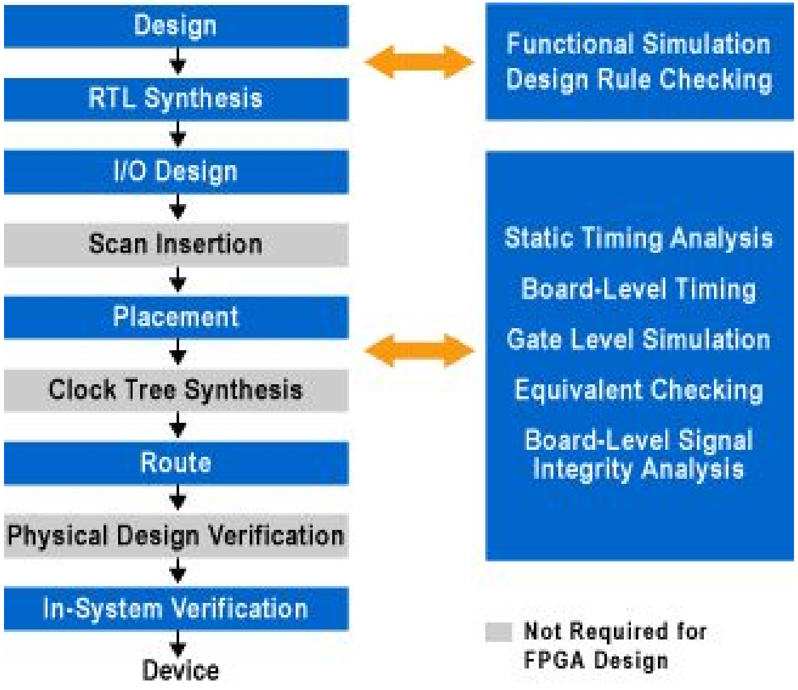
Intel® Atom™ Processor Z2610, formerly known as Medfield

What is an FPGA?



Why FPGAs?

FPGAs vs. ASIC



Design

RTL Synthesis

I/O Design

Scan Insertion

Placement

Clock Tree Synthesis

Route

Physical Design Verification

In-System Verification

Device

Functional Simulation
Design Rule Checking

Static Timing Analysis

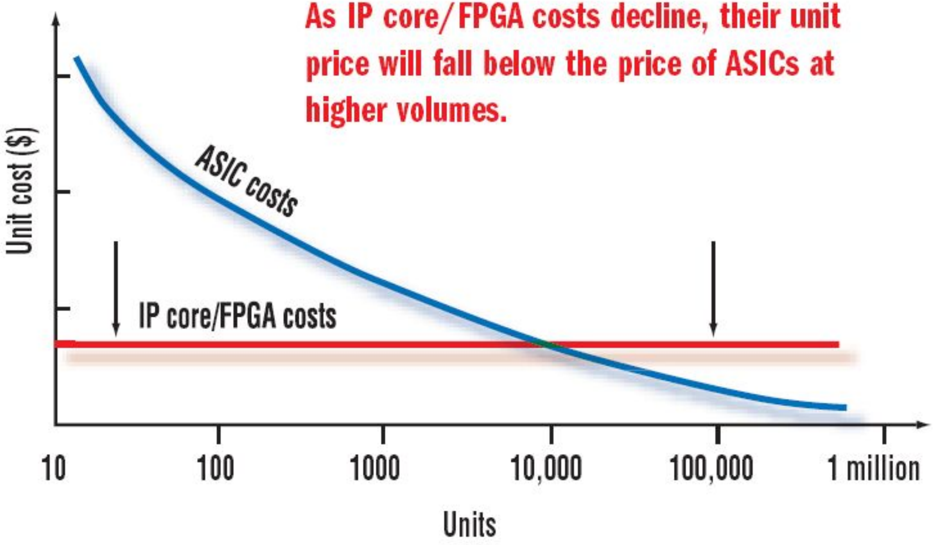
Board-Level Timing

Gate Level Simulation

Equivalent Checking

Board-Level Signal Integrity Analysis

▨ Not Required for FPGA Design



As IP core/FPGA costs decline, their unit price will fall below the price of ASICs at higher volumes.

Why FPGAs?

FPGAs vs. GPUs

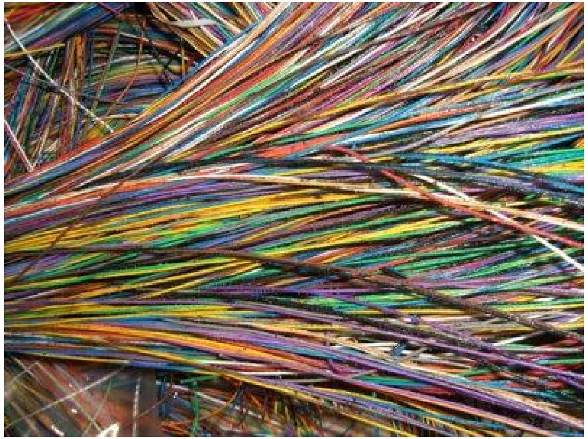
FPGA

- Energy efficient
- Faster for some algorithms
- **Hard to program!**

GPU

- Uses a lot of energy
- Floating Point fast
- Parallel code fast
- Poor performance on algorithms with irregular memory access

Designing Hardware Today



+

Verilog

```
module first_counter (input clock , input
reset , input enable ,
output[3:0] counter_out);

   wire clock ;
   wire reset ;
   wire enable ;
   reg [3:0] counter_out ;

   always @ (posedge clock)
   begin
     if (reset == 1'b1) begin
       counter_out <= 4'b0000;
     end
     else if (enable == 1'b1) begin
       counter_out <= counter_out + 1;
     end
   end
 endmodule
```

- low level
  - wires
  - registers
  - every bit defined
- timing complex
  - sync with a clock
  - sync with other functions

SystemC

```
#include "systemc.h"
SC_MODULE (first_counter) {
   sc_in_clk     clock ;
   sc_in<bool>   reset ;
   sc_in<bool>   enable;
   sc_out<sc_uint<4> > counter_out;
   sc_uint<4>  count;
   void incr_count () {
     if (reset.read() == 1) {
        count = 0;
        counter_out.write(count);
     } else if (enable.read() == 1) {
       count = count + 1;
       counter_out.write(count);
     }
   SC_CTOR(first_counter) {
     SC_METHOD(incr_count);
     sensitive << reset;
     sensitive << clock.pos();
   }
  };
```

- library in C++
- allows higher level design than Verilog/ VHDL but no complete automatic translation to synthesizable code
- need to understand hardware and SystemC framework

Why Haskell?

Functional Languages map well to hardware

- referential transparency/side-effect freedom make formal reasoning about programs vastly easier
- inherently concurrent and race-free (Church and Rosser)
- immutable data structures makes it vastly easier to reason about memory in the presence of concurrency

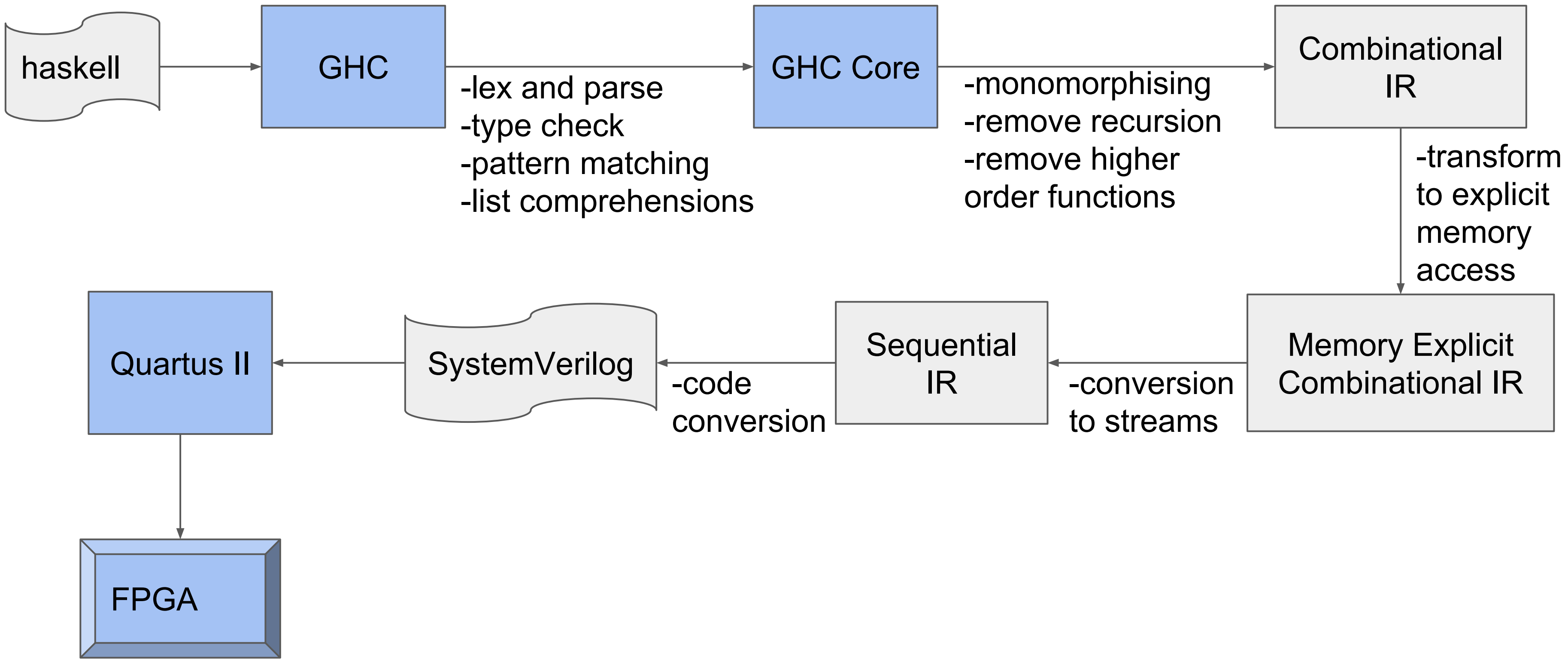
λ

Functional HDLs

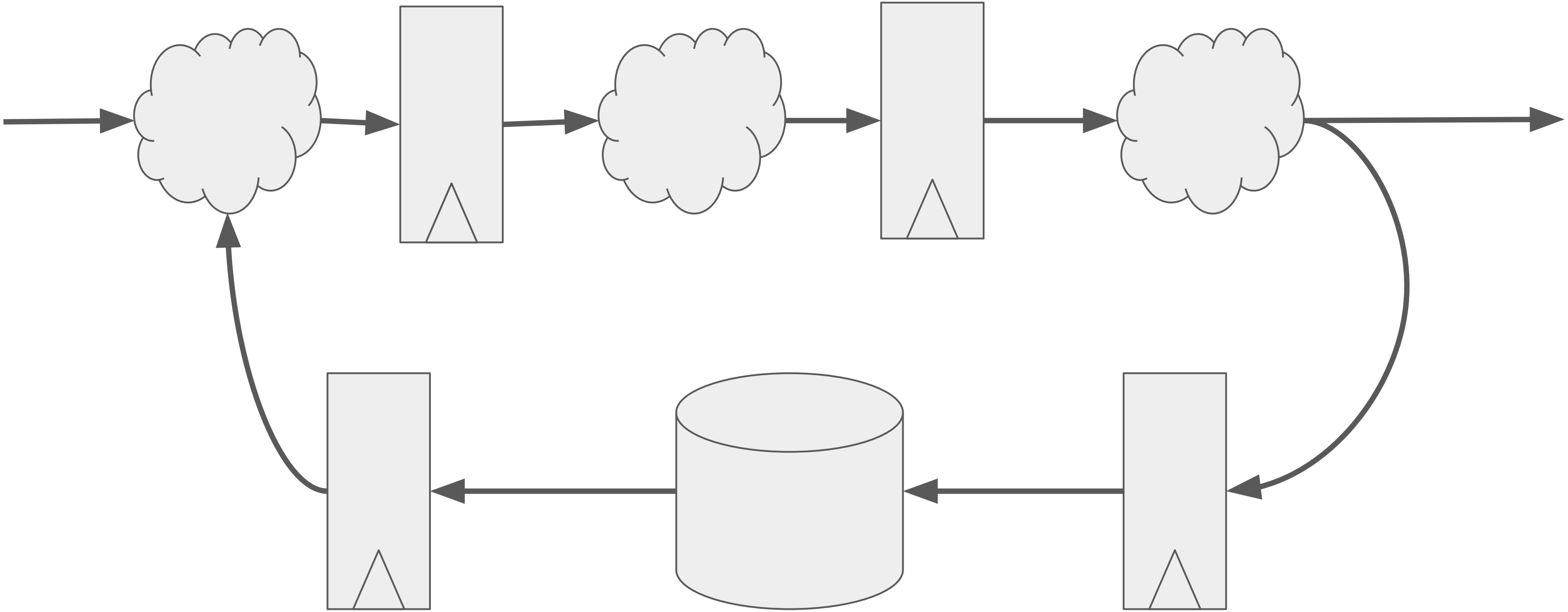
- µFP
    - Mary Sheeram at Oxford University
    - Functional, untyped, low level
- Lava
    - A continuation of the µFP project
    - Embedded in Haskell
    - Supports simulation, synthesis, and verification.
    - Version developed in Xilinx by Satnam Singh
- Bluespec
    - Created by Arvind at MIT
    - Proprietary HDLs to describe specific types of circuits
- C λaSH
    - Available as part of GHC
    - Allows complex concepts like higher order functions and type inference

We are not creating a functional hardware description language!

Compiler Overview



End Goal

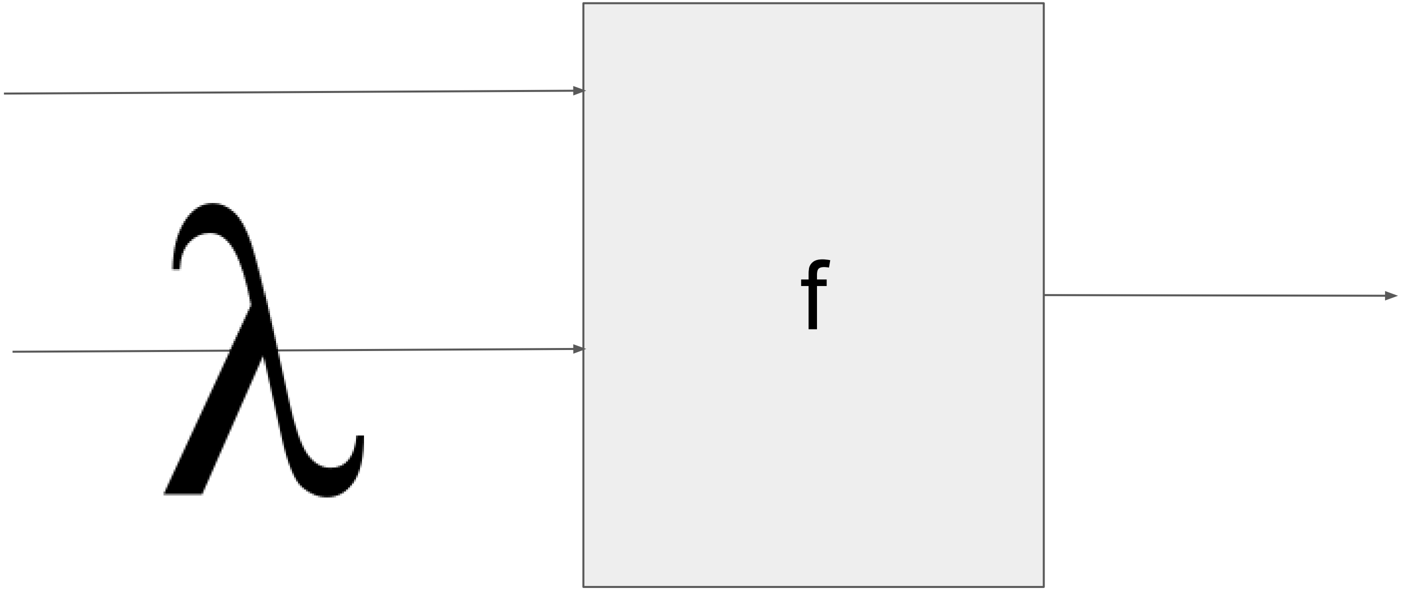


GHC frontend

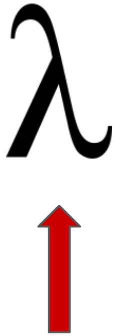
GHC frontend takes a complex and rich syntax and translates it to a typed core with relatively simple syntax tree

- lexer/parser
- inferred types
- type checker
- pattern matching
- syntactic sugar (list constructors)

Removing Higher Order Functions



Removing Higher Order Functions



- First lambda lift to ensure there are no free variables

f x y = ~~z~~ + x + y

- Passed functions become types

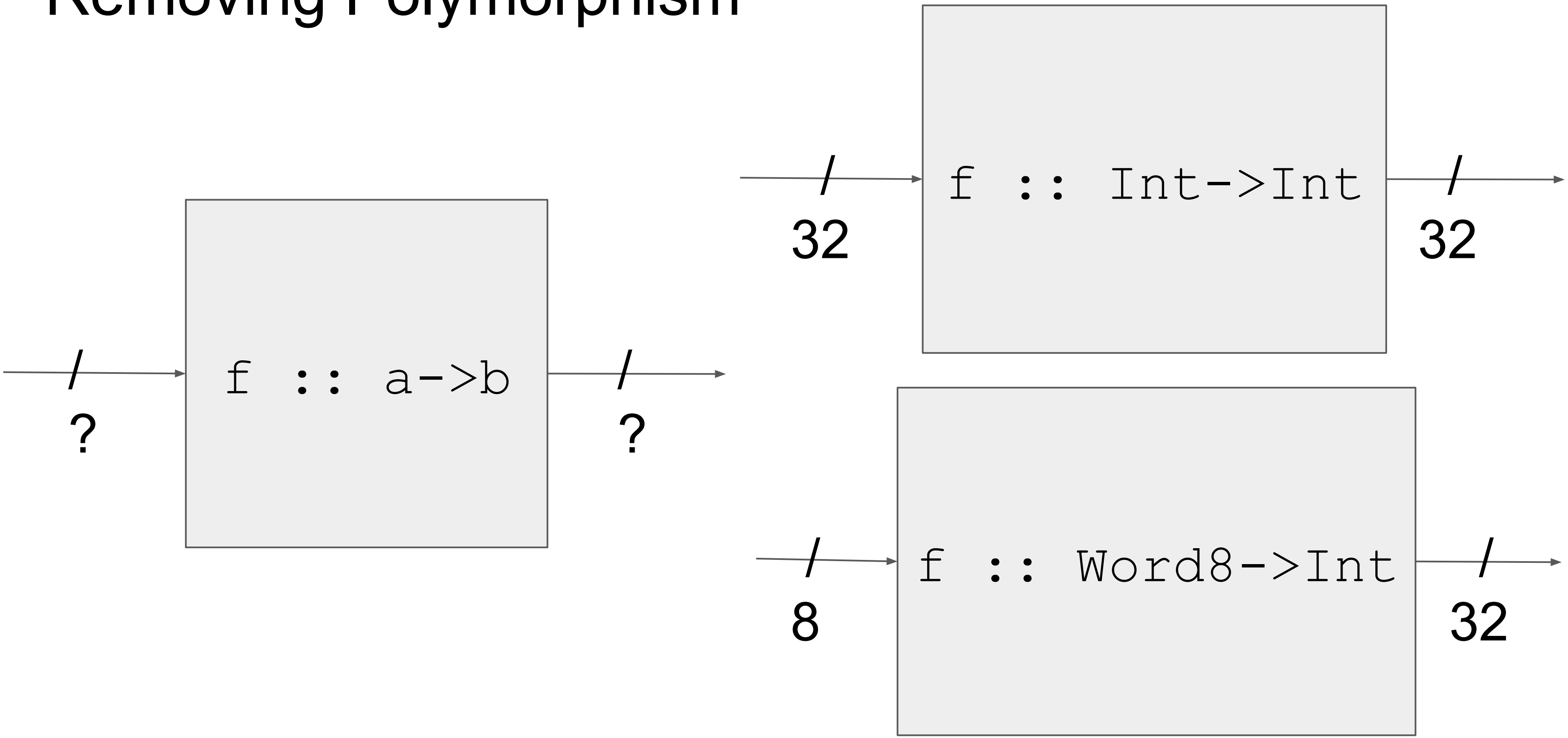
```
f x = x  → data F = F
```

- Apply function applies the new function type to values

```
apply F x = f x
```

- Functions that have functions as arguments now have new types as arguments and use the apply function when using the function type!

Removing Polymorphism

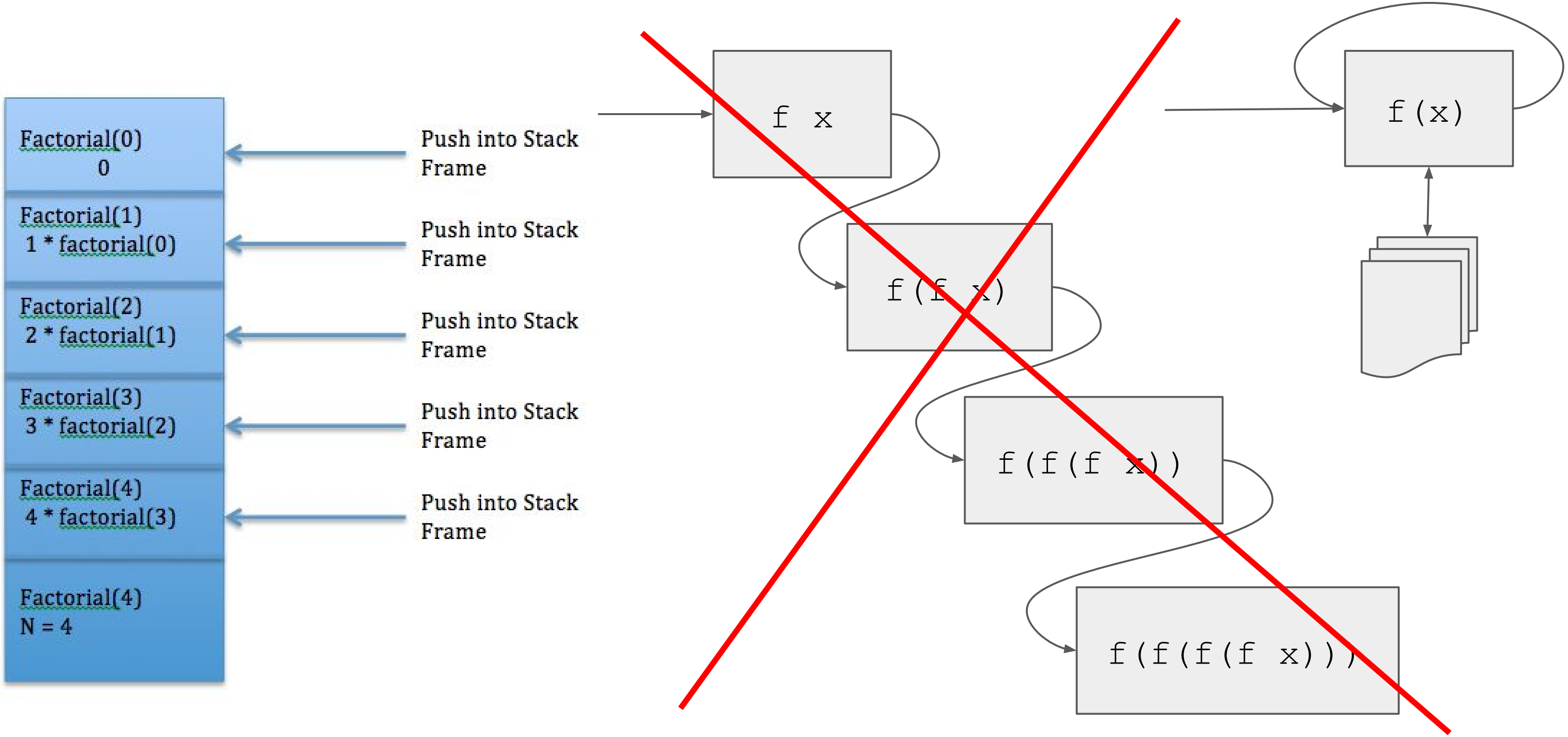


? `f :: a->b` ?

32 `f :: Int->Int` 32

8 `f :: Word8->Int` 32

Removing Recursion



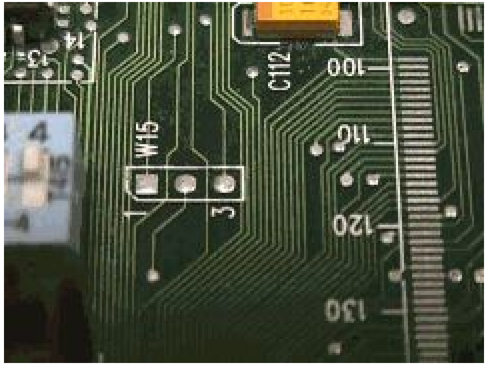
Remove Recursion

- Combine mutually recursive functions
- Sequence recursive call sites using CPS
- Translate Continuations into types (exactly like removing higher order functions)
- Define Operation type to separate going down recursive calls and and coming back up
- Transform Continuation type so it is no longer recursive
- Map Operation types to pushes and pops

Hardware Synthesis from a Recursive Functional Language

Translating Types to Bitvectors

```
data Shape =
    Circle Int Int
  | Square Int
  | Rectangle Int Int
  | Triangle Int Int
```



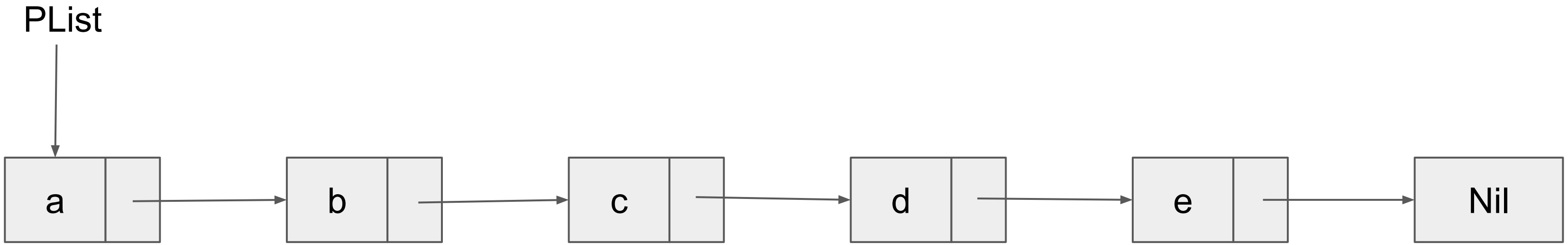
tag bits	first int bits	second int bits
0..1	2..33	34..65

Recursive Data types

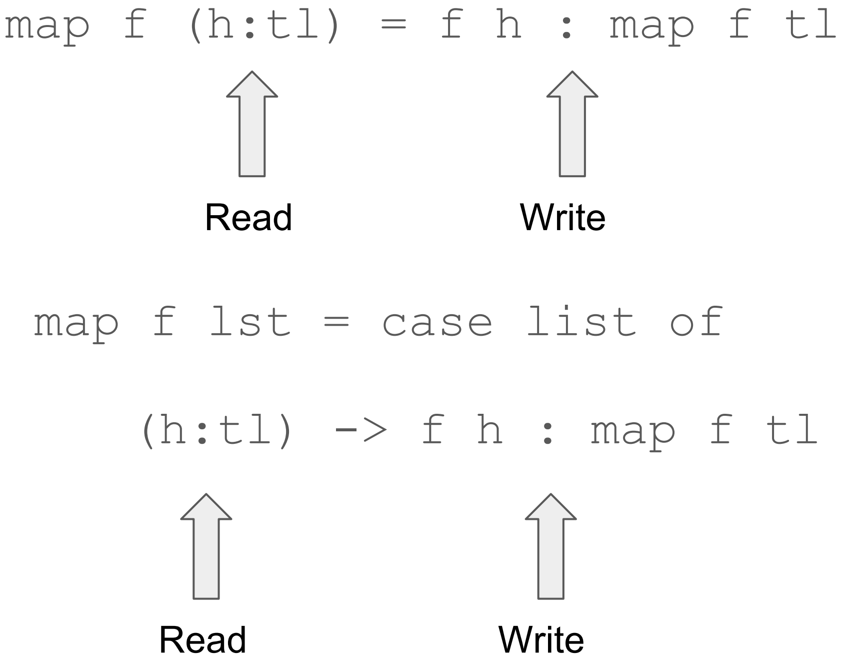
```
data List a = Cons a (List a) | Nil
```

⇓

```
data PList a = PCons a Ptr | PNil
```



Identifying Memory operations



```
map f (h:tl) = f h : map f tl
```

```
 map f lst = case list of

     (h:tl) -> f h : map f tl
```

Memories

- Type specific
- Independent
- Immutable

AST

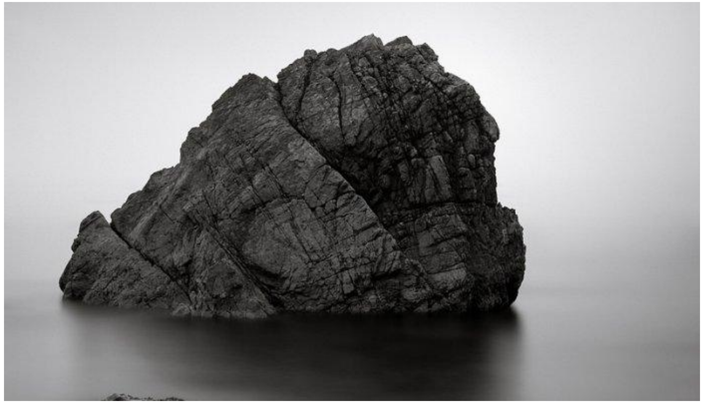
BTree Int

List Int

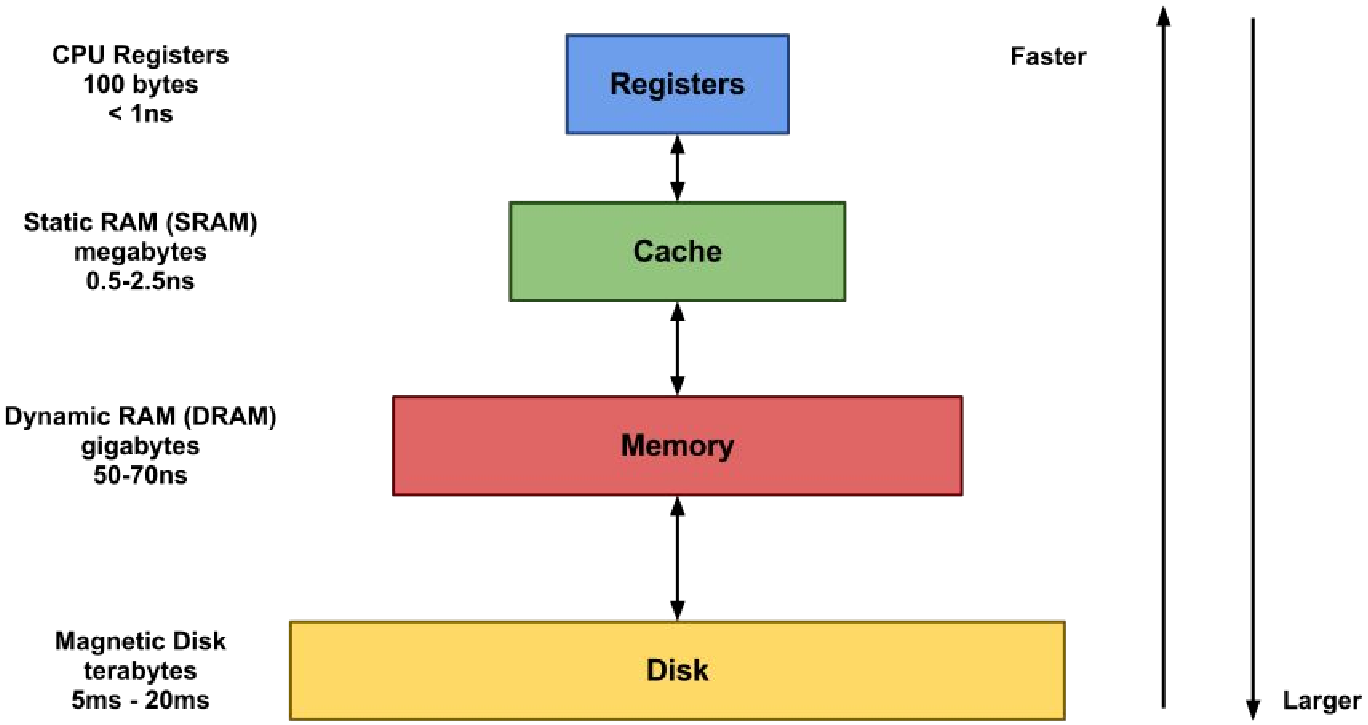
Map Word8 Int

BTree Word8

List Word8



Memories



CPU Registers
100 bytes
< 1ns

Registers

Static RAM (SRAM)
megabytes
0.5-2.5ns

Cache

Dynamic RAM (DRAM)
gigabytes
50-70ns

Memory

Magnetic Disk
terabytes
5ms - 20ms

Disk

Faster

Larger

A New Dimension

λ→t

```
newList = map (*2) [1..10]
```

Build List: Map (*2) Wait Nil

Read 1 **Push** 10

(2*1) Write Cons 10 ptr Write Cons 20 ptr

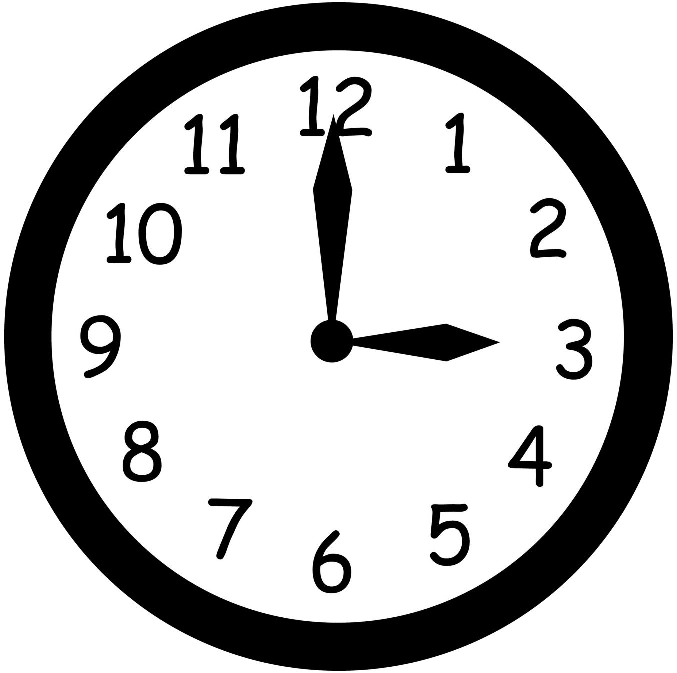
Push 2 ... 10

Pop 2 ... 1

Read 10 Write Cons 2 ptr Write Cons 1 ptr

(2*10)

Push 20



Sequencing Haskell

```
data Stream a = a :> Stream a

count = 1 :> count + 1 -- produces a stream of 0,1,2,3…

delCount = 10 :> count

addC = count + delCount
```

cycle	0	1	2	3	4	5	6	7	8	9
count	1	2	3	4	5	6	7	8	9	10
delCount	10	1	2	3	4	5	6	7	8	9
addC	11	3	5	7	9	11	13	15	17	19

FHW - Streaming Haskell

```haskell
infixr 5 :>
data Stream a =  a :> Stream a

repeat :: a -> Stream a
repeat a = s where s = a :> s

map :: (a -> b) -> Stream a -> Stream b
map f (h :> t) = f h :> map f t

zipWith :: (a -> b -> c) -> Stream a -> Stream b -> Stream c
zipWith f (a :> as) (b :> bs) = f a b :> zipWith f as bs

scanl :: (b -> a -> b) -> b -> Stream a -> Stream b
scanl f init_state input_stream = next_state
    where
    next_state = zipWith f present_state input_stream
    present_state = init_state :> next_state
```

Memories in FHW

```
memory# ::   Word#                -- ^ Size in words
         -> word                  -- ^ Initial contents
         -> Stream Bool           -- ^ Write commands
         -> Stream Word32         -- ^ Addresses
         -> Stream word           -- ^ Write data
         -> Stream word           -- ^ Read data
memory# size init write addr wdata = init :> zipWith readOp memoryState addr
    where
    updatedMemory = zipWith writeOp memoryState (zipWith3 (\a b c -> (a,b,
c)) write addr wdata)
    memoryState = initialMem :> updatedMemory

    size' :: Word32
    size' = W32# size     -- Get away from the primitive type
    initialMem = array (0, size' - 1) [(a,init) | a <- [0..size'-1]]

    writeOp arr (True, a, d) = arr // [(a, d)]
    writeOp arr _         = arr

    readOp arr ad = arr ! ad
```

Recap

- CPUs aren’t getting much faster
- Hardware designed to do specific work can save energy and improve performance
- FPGAs are a type of hardware that can be reprogrammed and cost less for smaller deployments
- Really need a way to program FPGAs that doesn’t suck
- Haskell is really cool and has properties that make transformations to hardware easier than other things
- Our group wants to make writing programs on FPGAs accessible to people who don’t know how hardware works

Questions

Thank you!