Creating an Agile Hardware Accelerator Design Flow

Priyanka Raina

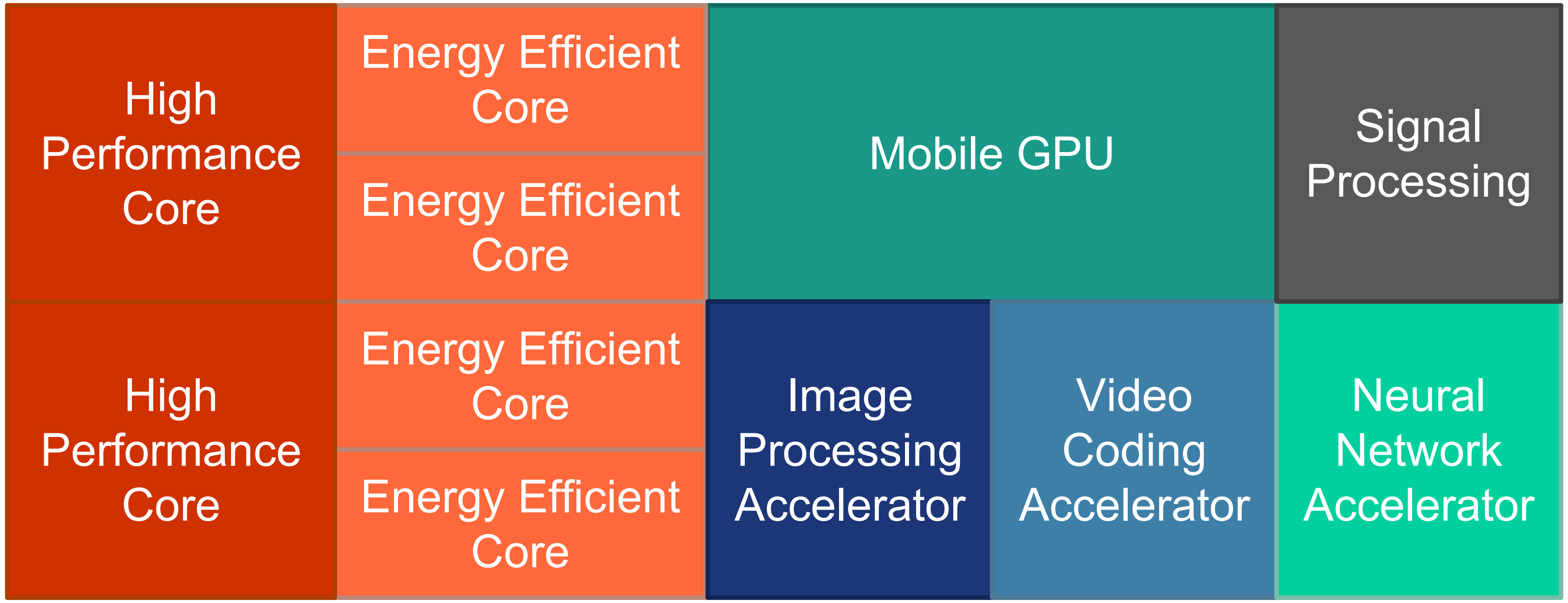
Stanford AHA! Agile Hardware Center

Stanford | ENGINEERING
Electrical Engineering

Nov 27, 2019

Need for hardware accelerators

- With the slowdown of technology scaling we cannot achieve higher performance and energy-efficiency with general purpose hardware
- Instead we are relying on specialized hardware aka accelerators to meet performance and energy demands

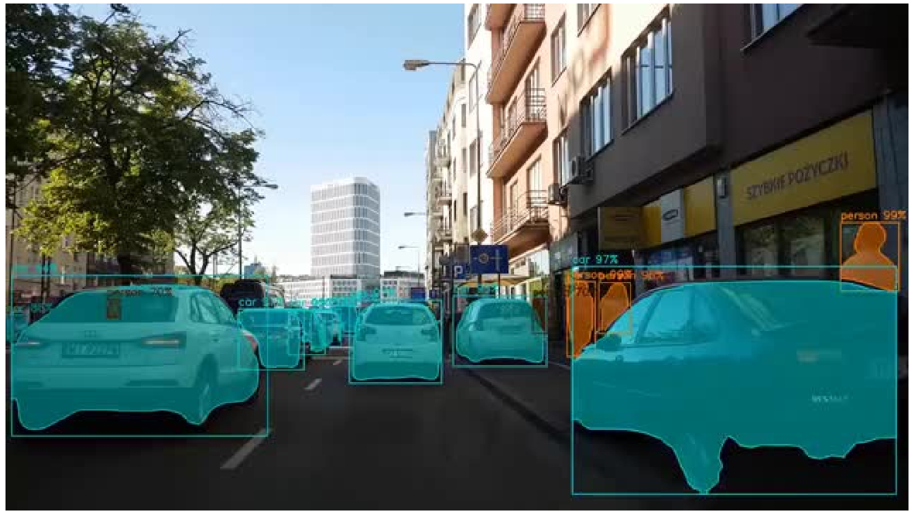


How hardware design is done today

Study the application

Design hardware

Write software



How hardware design is done today

Study the application

Design hardware

Write software

```
// Is this the last line in the thing? Valid_out should be gated based on the stencil
logic [15:0] vg_ctr;
logic valid_gate;
logic valid_int;
logic threshold;

assign valid_gate = (stencil_width == 0) ? 1 : vg_ctr >= (stencil_width - 1);
assign valid_int = (num_words_mem >= (depth - 1)) & wen & (depth > 0) & threshold;
assign valid = valid_gate & valid_int;
assign ren_to_fifo = (num_words_mem >= (depth - 1)) & wen & (depth > 0);

always_ff @(posedge clk or posedge reset) begin
   if(reset) begin
      threshold <= 0;
   end
   else if(clk_en) begin
      if(flush) begin
         threshold <= 0;
      end
      else if ((num_words_mem == (depth - 1)) & wen) begin
         threshold <= 1;
      end
   end
end

always_ff @(posedge clk or posedge reset) begin
    if(reset) begin
       vg_ctr <= 0;
    end
    else if(clk_en) begin
        if (flush) begin
            vg_ctr <= 0;
        end
        else begin
          if(valid_int) begin
            if(vg_ctr == (depth-1)) begin
                vg_ctr <= 0;
            end
            else begin
                vg_ctr <= vg_ctr + 1;
            end
          end
        end
    end
end
```

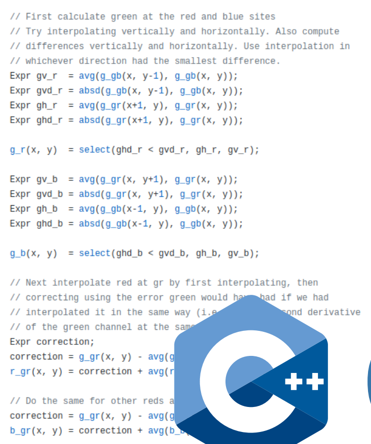
Verilog, VHDL, SystemVerilog...

How hardware design is done today

Study the application

Design hardware

Write software



```
// First calculate green at the red and blue sites
// Try interpolating vertically and horizontally. Also compute
// differences vertically and horizontally. Use interpolation in
// whichever direction had the smallest difference.
Expr gv_r  = avg(g_gb(x, y-1), g_gb(x, y));
Expr gvd_r = absd(g_gb(x, y-1), g_gb(x, y));
Expr gh_r  = avg(g_gr(x+1, y), g_gr(x, y));
Expr ghd_r = absd(g_gr(x+1, y), g_gr(x, y));

g_r(x, y)  = select(ghd_r < gvd_r, gh_r, gv_r);

Expr gv_b  = avg(g_gr(x, y+1), g_gr(x, y));
Expr gvd_b = absd(g_gr(x, y+1), g_gr(x, y));
Expr gh_b  = avg(g_gb(x-1, y), g_gb(x, y));
Expr ghd_b = absd(g_gb(x-1, y), g_gb(x, y));

g_b(x, y)  = select(ghd_b < gvd_b, gh_b, gv_b);

// Next interpolate red at gr by first interpolating, then
// correcting using the error green would ha   had if we had
// interpolated it in the same way (i.e         cond derivative
// of the green channel at the sam
Expr correction;
correction = g_gr(x, y) - avg(g
r_gr(x, y) = correction + avg(r

// Do the same for other reds a
correction = g_gr(x, y) - avg(g
b_gr(x, y) = correction + avg(b_
```

Problems with this waterfall approach

- Change of application requirements

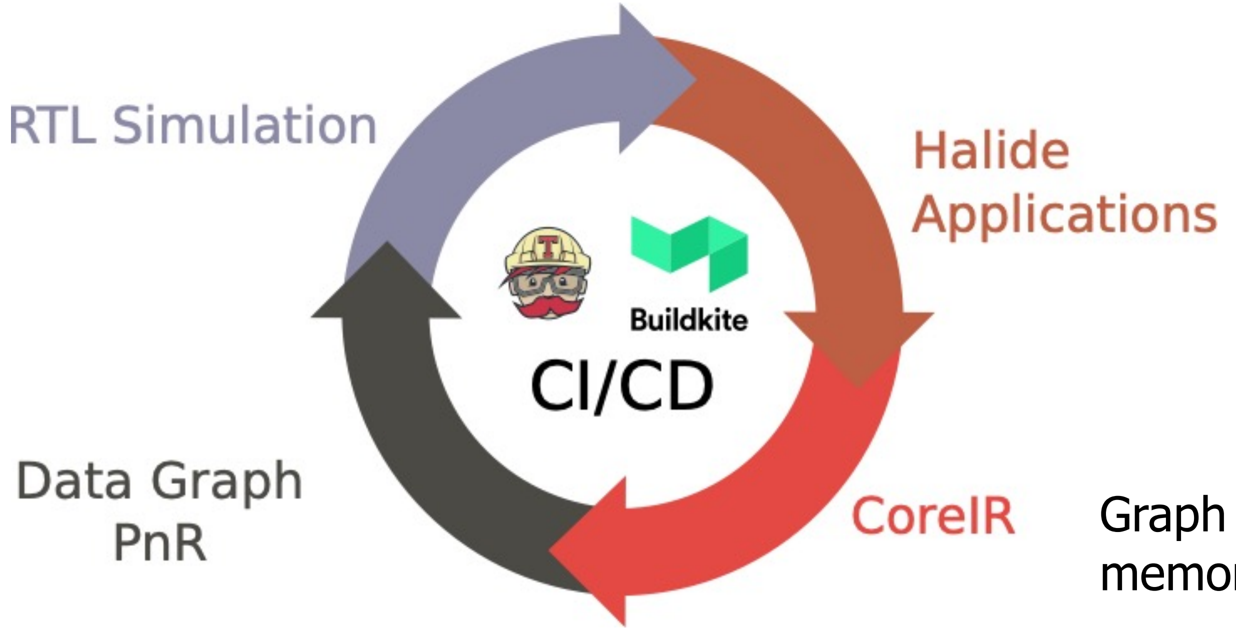
- Incomplete knowledge/understanding of the problem
  - The only software that works is the software you use
  - New software never works well

An Agile Approach to Hardware Design

Our vision of agile hardware design

- Create an end-to-end system
  - From Halide application code to working CGRA based accelerators

- Evolve that system to make it more efficient



Algorithm + Schedule

Graph of operations and memory blobs

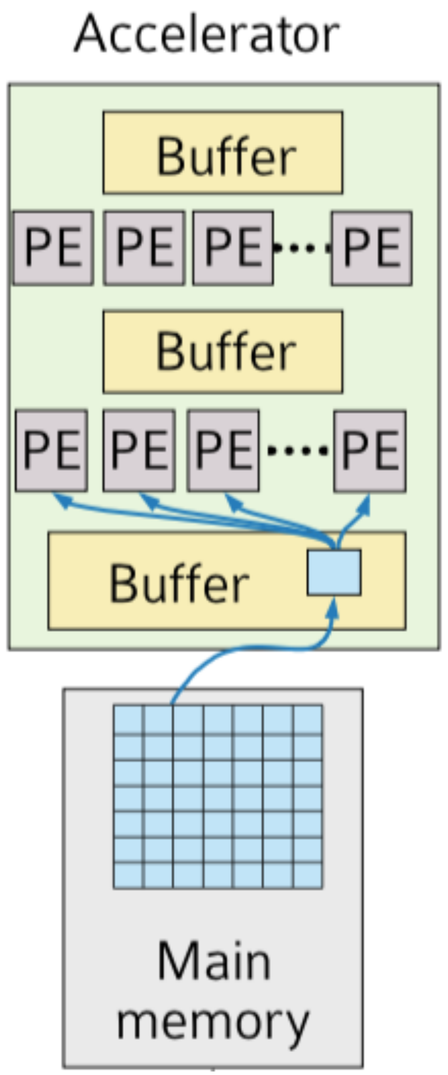
Halide Application Example – MobileNet Layer

```
// Algorithm
RDom r_dw(-1,3, -1,3), r_pw(0, 32);
dw_conv(x, y, c) += input(x+r_dw.x, y+r_dw.y, c)
                  * w1(1 + r_dw.x, 1 + r_dw.y, c);
pw_conv(x, y, k) += dw_conv(x,y, r_pw.c)
                 * w2(r_pw.c, k);
// Schedule
pw_conv.split(x, xo, xi, 16)
       .split(y, yo, yi, 16)
       .reorder(k, xi, yi, r_pw.c, xo, yo);
dw_conv.reorder(x, y, c);
dw_conv.compute_at(pw_conv, xi)
       .store_at(pw_conv, xo)
pw_conv.accelerate({input}, xo)
dw_conv.unroll(r_dw.x, 3).unroll(r_dw.y, 3);
pw_conv.unroll(k, 32);
```

Tiling

On-chip buffering

Parallel compute



Halide to CoreIR

Algorithm

```
Var x, y, xi, yi, xo, yo;
ImageParam input;
Func conv, output;
RDom win(0, 3, 0, 3);

conv(x, y)  = 0.0;
conv(x, y) += input(x + win.x, y + win.y) *
              weights(win.x, win.y);
output(x, y) = conv(x, y);
```

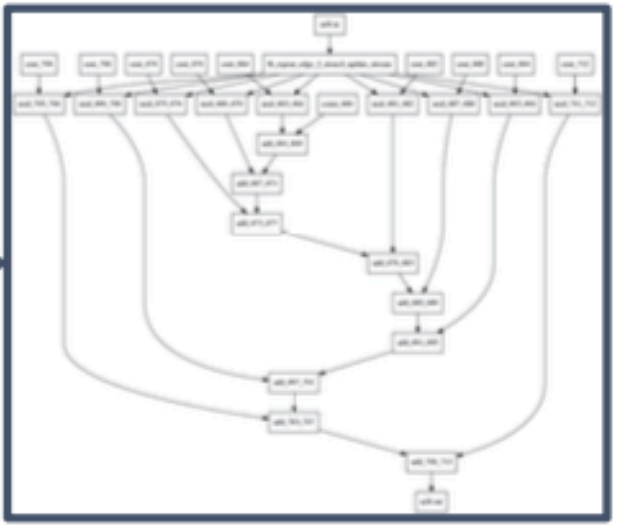
Schedule

```
output.tile(x, y, xo, yo, xi, yi, 64, 64);
output.hw_accelerate(xo, xi);
conv.compute_at(output, xi)
conv.update()
    .unroll(win.x)
    .unroll(win.y);

input.in()
     .store_at(output, xo)
     .compute_at(output, xi)
     .stream_to_accelerator();
```

Halide IR loop nest

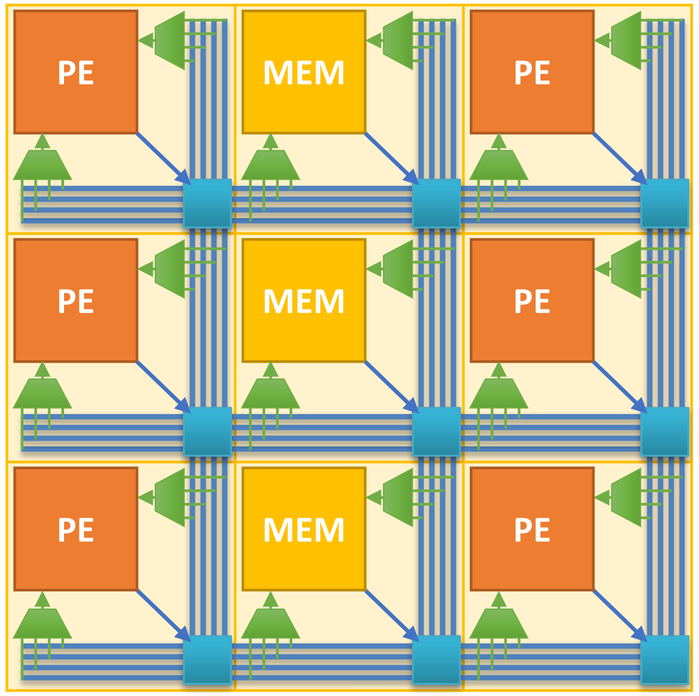
```
GENERATED_HARDWARE(xo, yo):
   StorageUnitToBeSpecified weights;
   Register<int> tmp_conv;
   LineBuffer<size=2x64, stencil=3x3, thruput=1>
         tmp_input;

   for output.yi = 0 to 64:
     for output.xi = 0 to 64:

      Load one new pixel into tmp_input

      tmp_conv = tmp_input[3x3 stencil] * weights

      // streaming store to memory
      STORE tmp_conv to output(xo*64+xi, yo*64+yi)
```

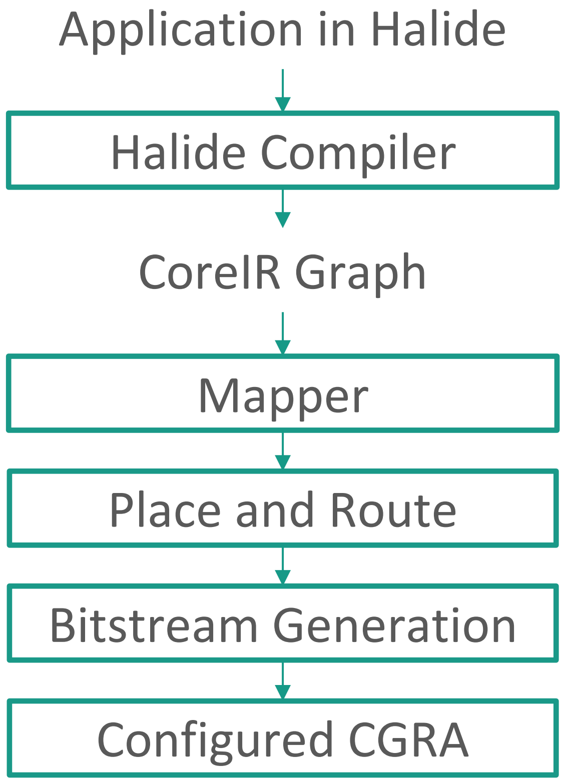
CoreIR circuit



An experiment on building accelerators

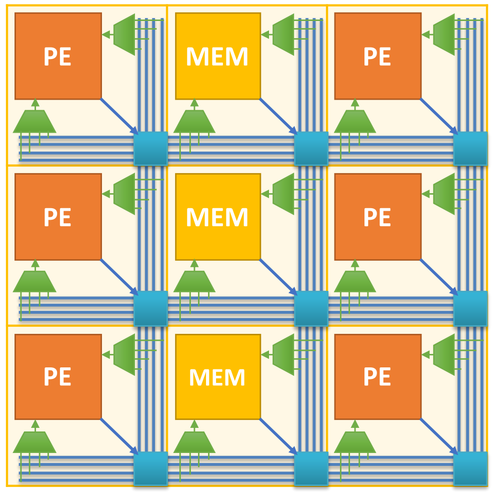
- Choose coarse-grained reconfigurable array (CGRA) as our base architecture
  - Since acceleration comes from exploiting parallelism in compute and locality in memory references
- Create accelerators through specialization of the base CGRA

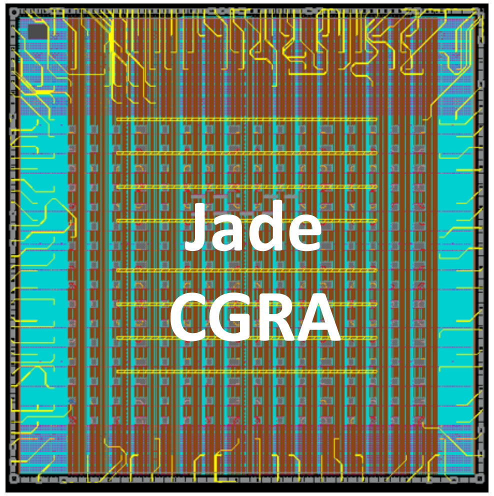


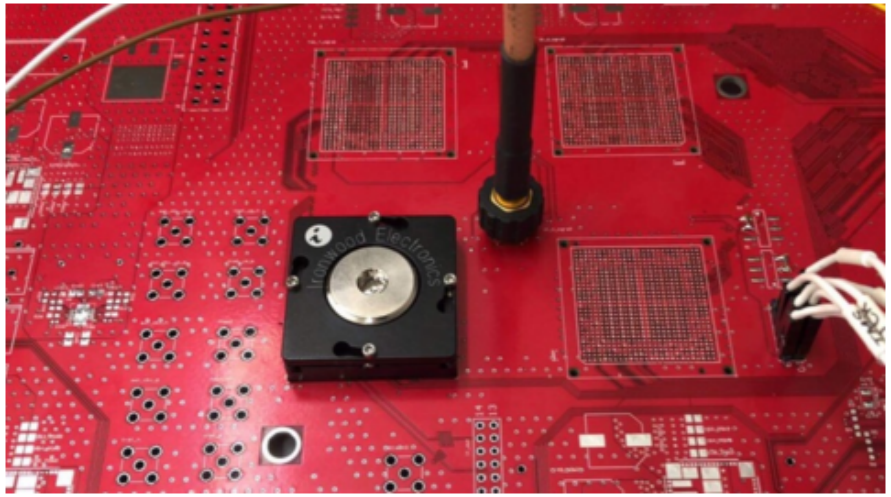


Jade: Our first generation CGRA

- A 16x16 island-style CGRA with simple processing elements (16 bit integer ALU, registers, LUT) and memories (2 KB – SRAM, FIFO, line buffer) for image processing applications
- Built with Genesis2, a hardware generation framework that uses Perl to meta-program hardware modules written in SystemVerilog
- Taped out in Summer 2018, received packaged parts in January 2019. Chip is fully functional.

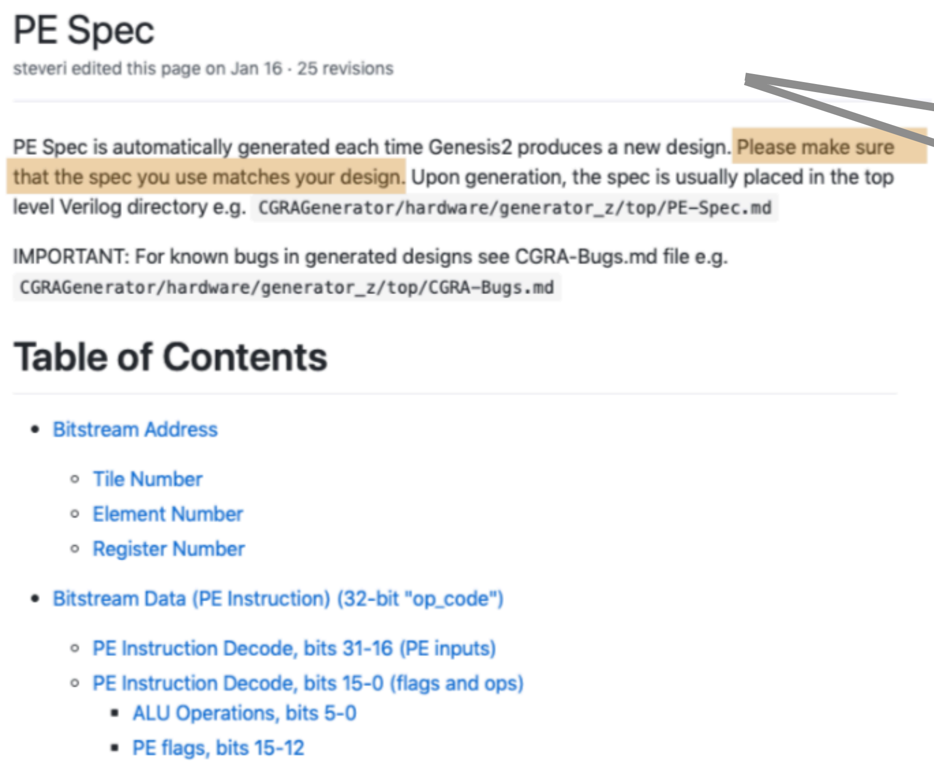






Lesson 1: Single source of truth

- For Jade, we really had two flows – one for application mapping, and other for CGRA generation
  - Loosely coupled – changes to hardware required manual updates to all the tools



PE Spec

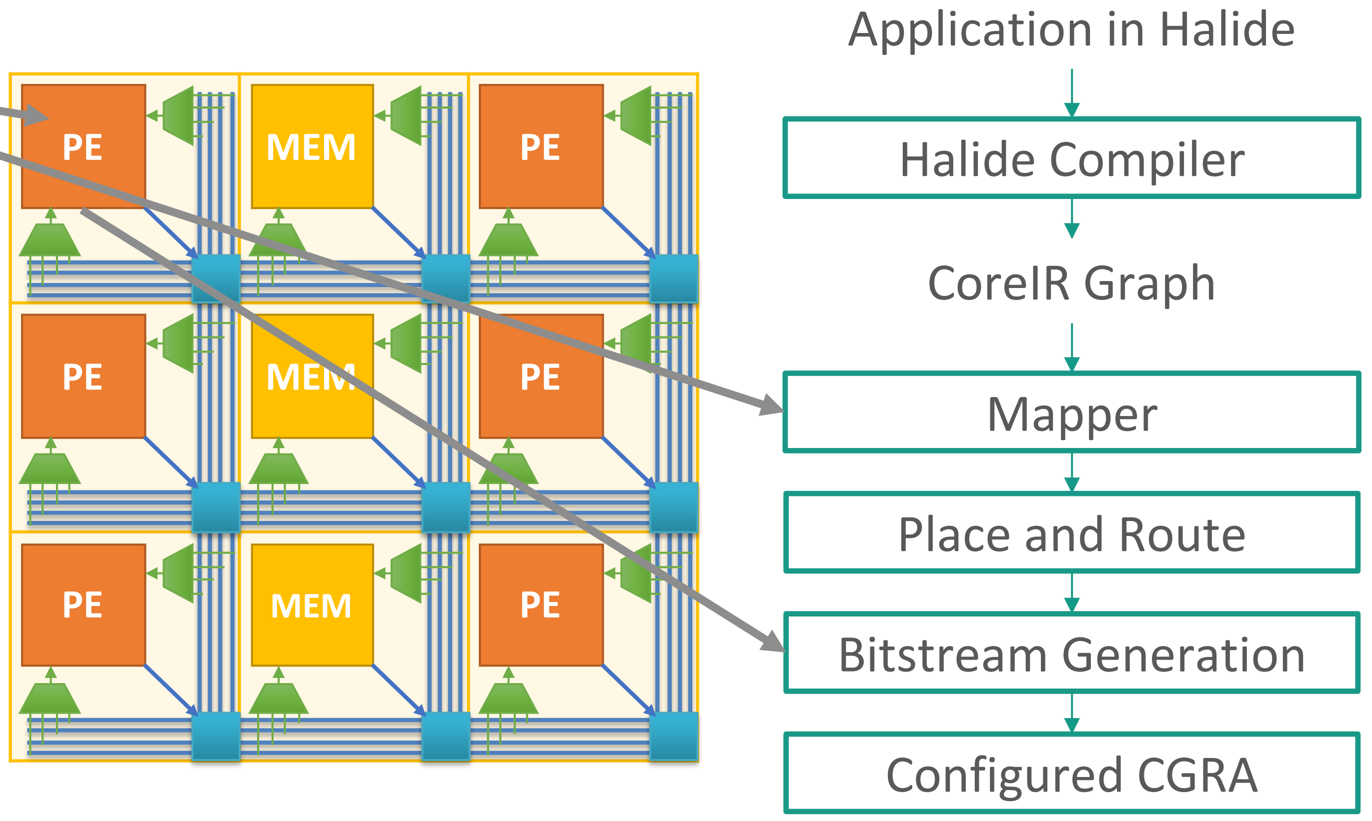
steveri edited this page on Jan 16 · 25 revisions

PE Spec is automatically generated each time Genesis2 produces a new design. Please make sure that the spec you use matches your design. Upon generation, the spec is usually placed in the top level Verilog directory e.g. `CGRAGenerator/hardware/generator_z/top/PE-Spec.md`

IMPORTANT: For known bugs in generated designs see CGRA-Bugs.md file e.g. `CGRAGenerator/hardware/generator_z/top/CGRA-Bugs.md`

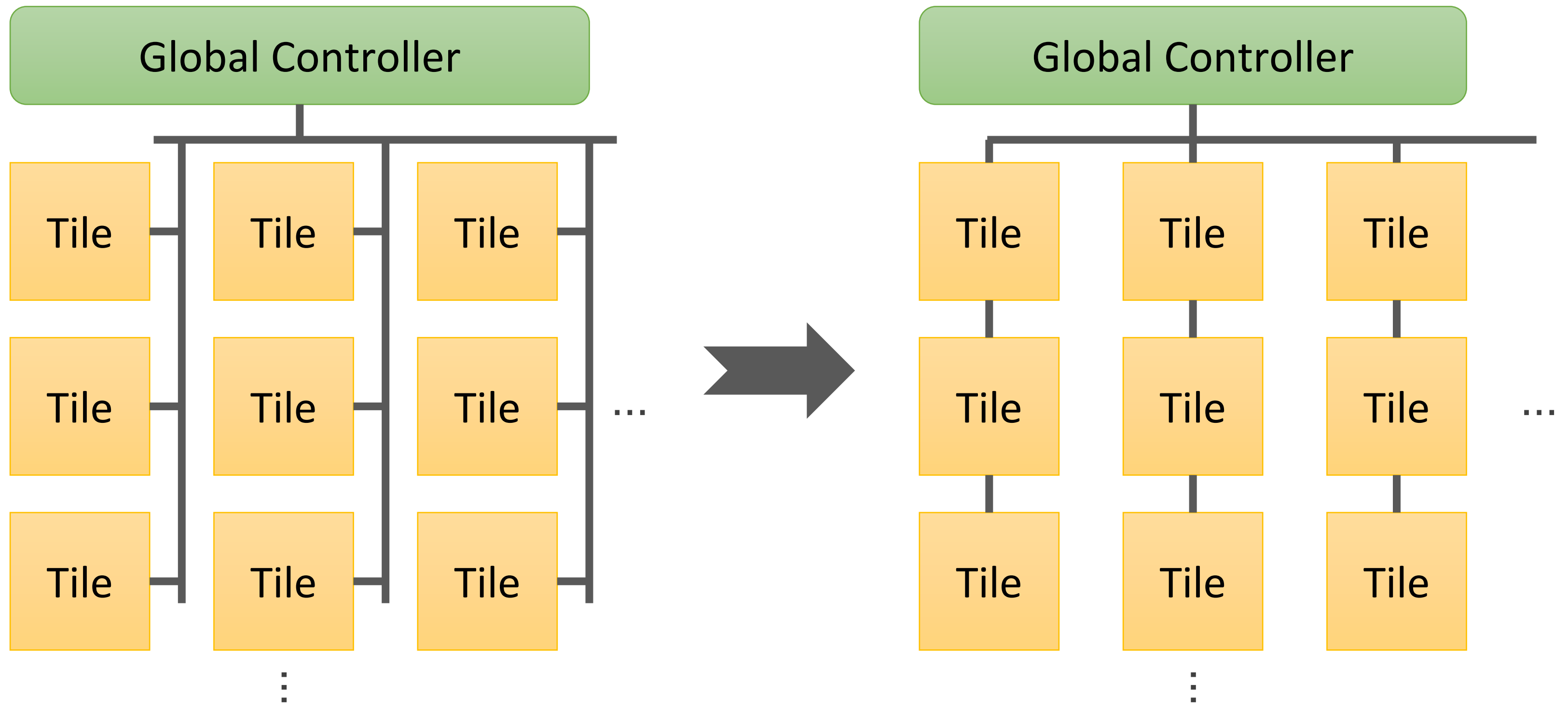
Table of Contents

- Bitstream Address
  - Tile Number
  - Element Number
  - Register Number
- Bitstream Data (PE Instruction) (32-bit "op_code")
  - PE Instruction Decode, bits 31-16 (PE inputs)
  - PE Instruction Decode, bits 15-0 (flags and ops)
    - ALU Operations, bits 5-0
    - PE flags, bits 15-12



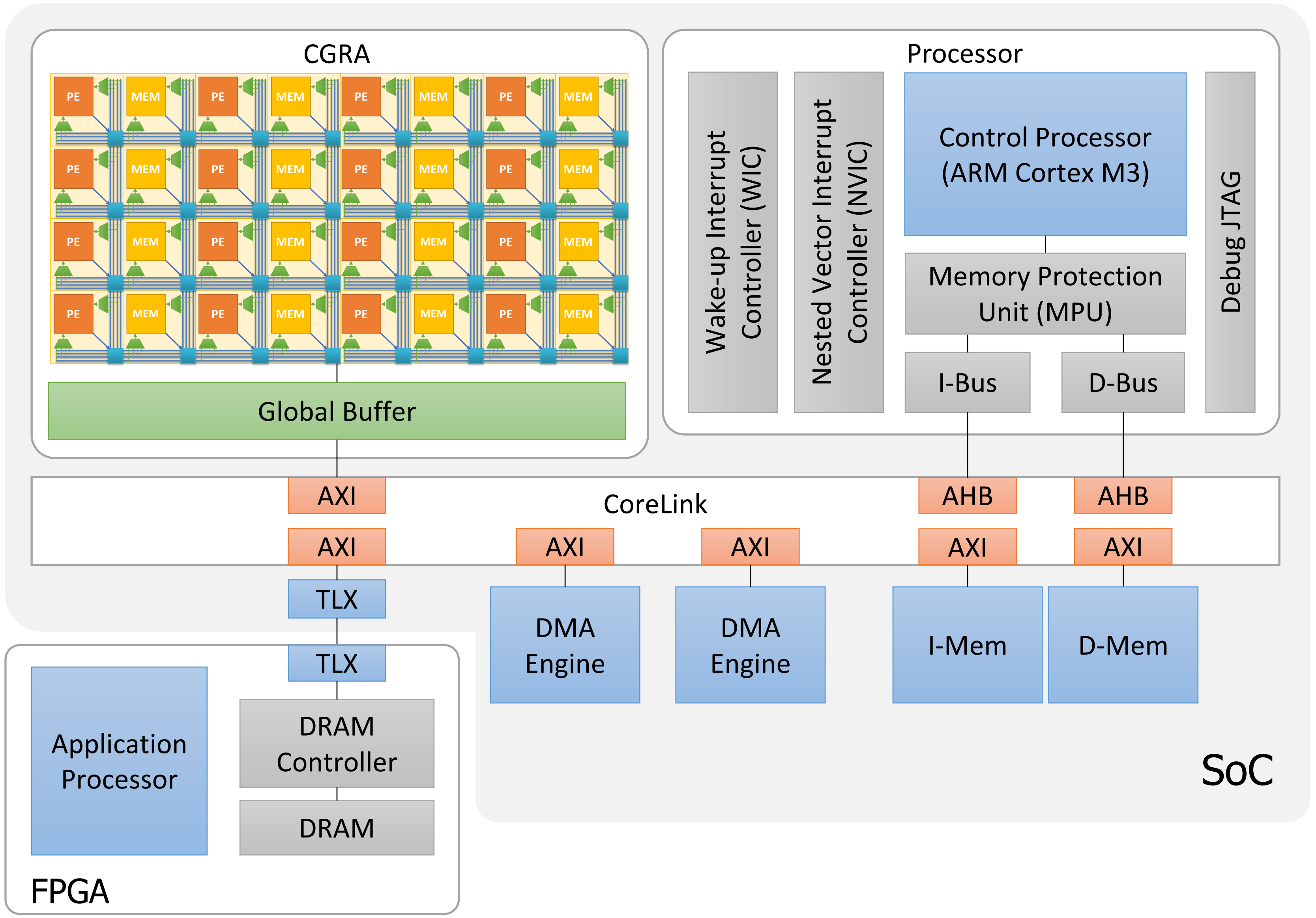
Lesson 2: Staged generation of design

- We wanted to make several changes to the logical design for physical design concerns
  - We had several global signals in the design – place and route tools were unsuccessful in routing them to all the tiles in narrow channels – leading to low area efficiency



Garnet: Our second generation CGRA SoC

- Garnet is more complex
  - PEs with floating point
  - MEMs with unified buffers support image + NN apps
  - Memory hierarchy: MEMs → Global Buffer → DRAM
  - Fast reconfiguration
  - Configurable power domains
- Full processor sub-system



A DSL-based hardware-software generation framework

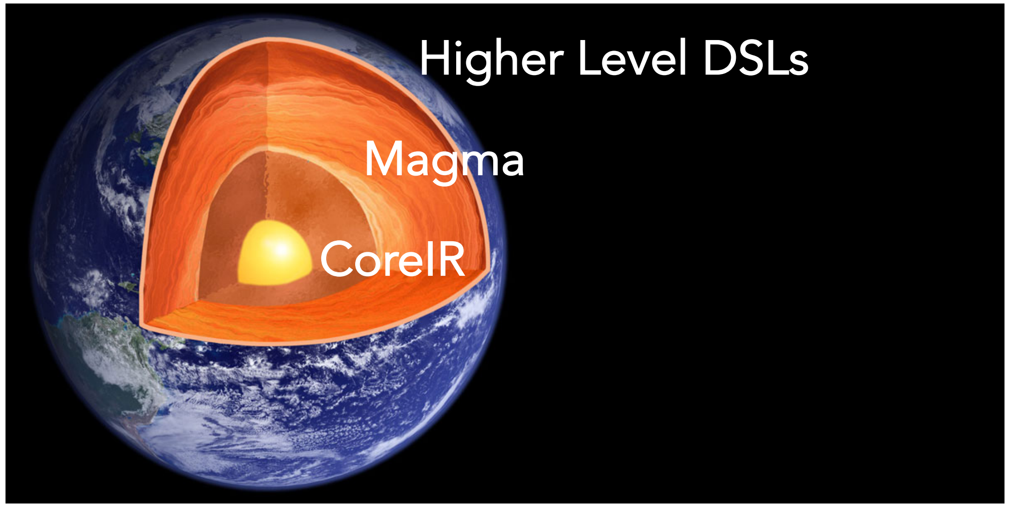
- Goal is to generate hardware and software from a higher-level specification (single source of truth)
- But generating arbitrary hardware from arbitrary higher-level specification is an extremely difficult problem
- We divide the problem into generating different specific types of hardware – like processing elements, memories and interconnect

A DSL-based hardware-software generation framework

- We create DSLs that easily express functionality of specific types of hardware
  - **PEak** for processing elements
  - **Lake** for memories
  - **Canal** for interconnect
  - **Gemstone** for CGRAs
  - These sit on top of our
    - Python-embedded HDL - **Magma**
    - Hardware intermediate representation – **CoreIR**

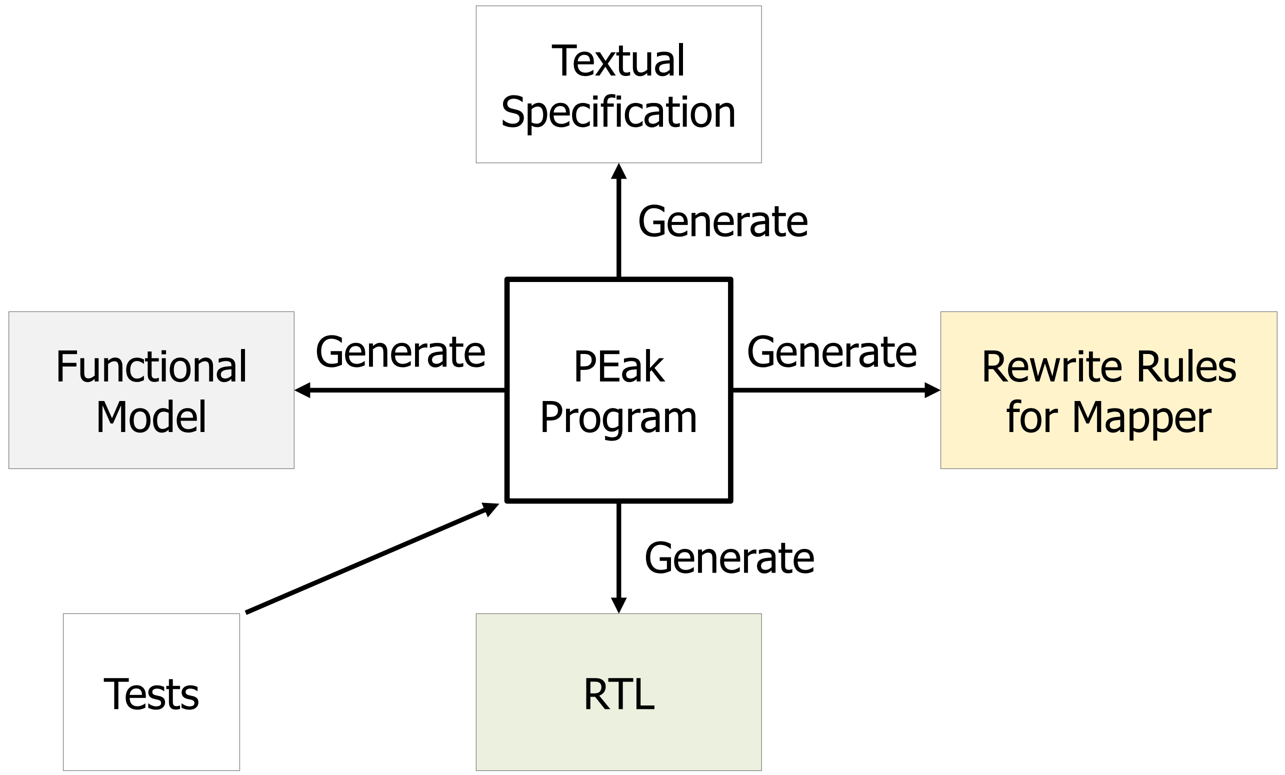
- Generate collateral for all tools in the flow from a single source of truth

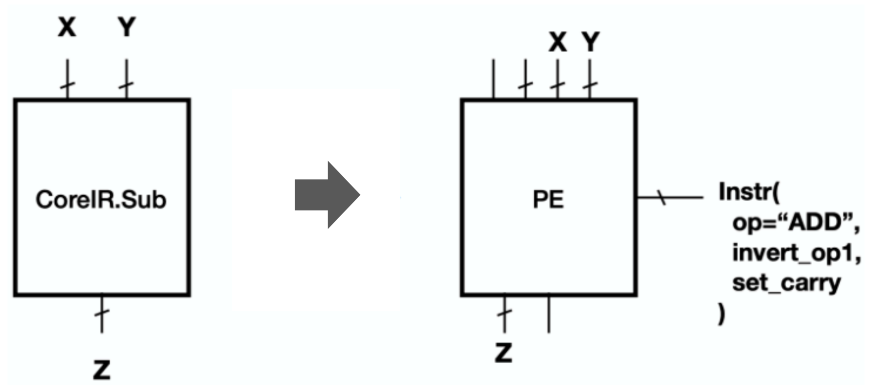
- Allow passes/staged generation for separation of concerns



PEak: DSL for Processing Elements (PEs)

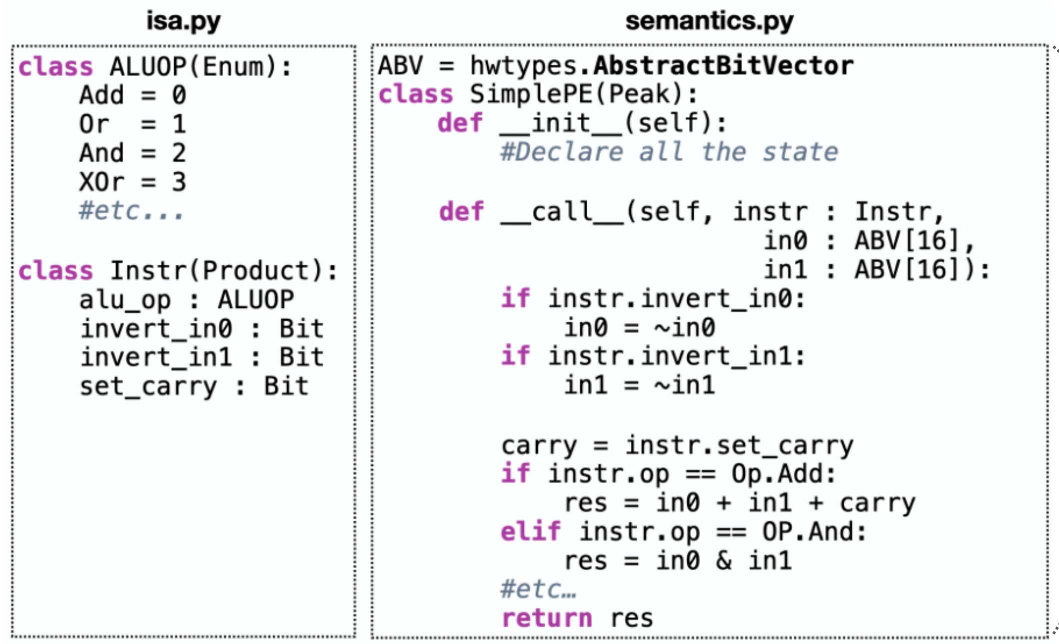
- Python-embedded DSL for specifying PEs
  - Defines an instruction set – using algebraic data types
  - Declares all state
  - Precisely describes instruction semantics





Rewrite Rule

Multiple interpretations of the same PEak program



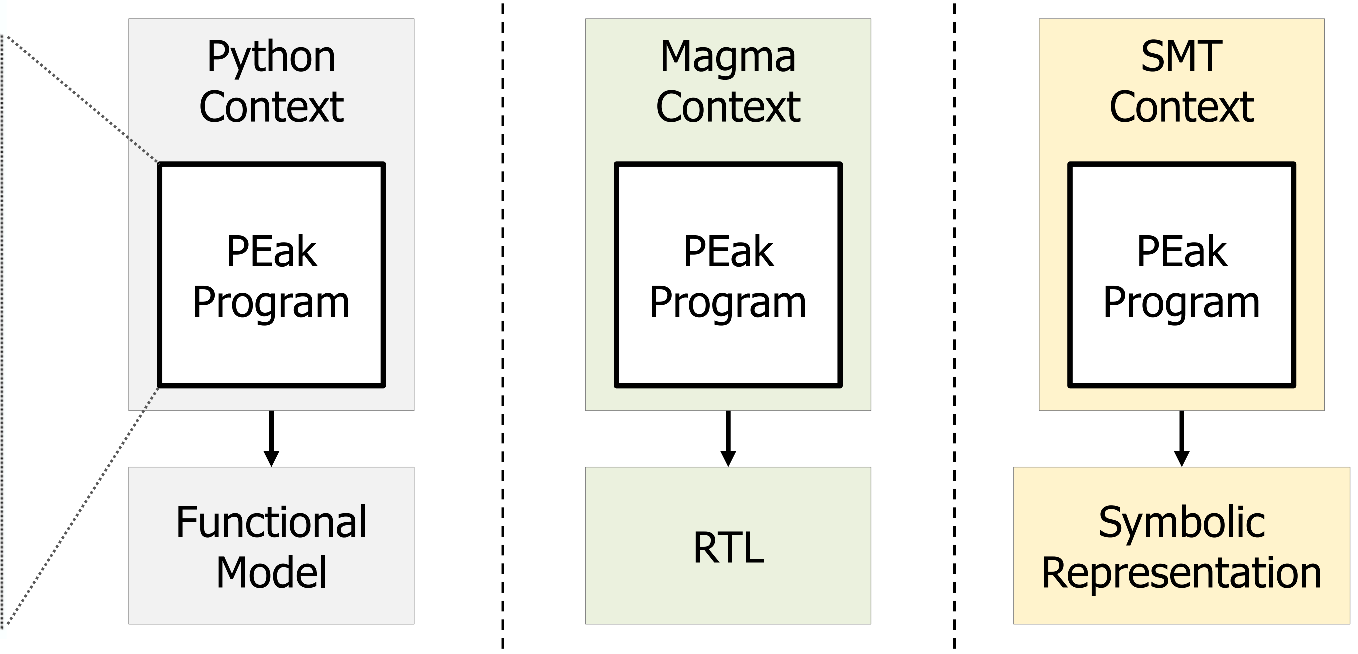
isa.py

```
class ALUOP(Enum):
    Add = 0
    Or  = 1
    And = 2
    XOr = 3
    #etc...

class Instr(Product):
    alu_op : ALUOP
    invert_in0 : Bit
    invert_in1 : Bit
    set_carry : Bit
```

semantics.py

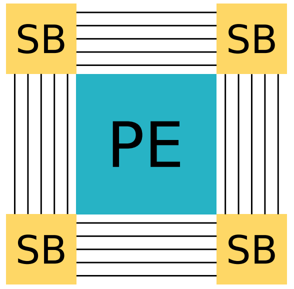
```
ABV = hwtypes.AbstractBitVector
class SimplePE(Peak):
    def __init__(self):
        #Declare all the state

    def __call__(self, instr : Instr,
                          in0 : ABV[16],
                          in1 : ABV[16]):
        if instr.invert_in0:
            in0 = ~in0
        if instr.invert_in1:
            in1 = ~in1

        carry = instr.set_carry
        if instr.op == Op.Add:
            res = in0 + in1 + carry
        elif instr.op == OP.And:
            res = in0 & in1
        #etc…
        return res
```

PEak Program



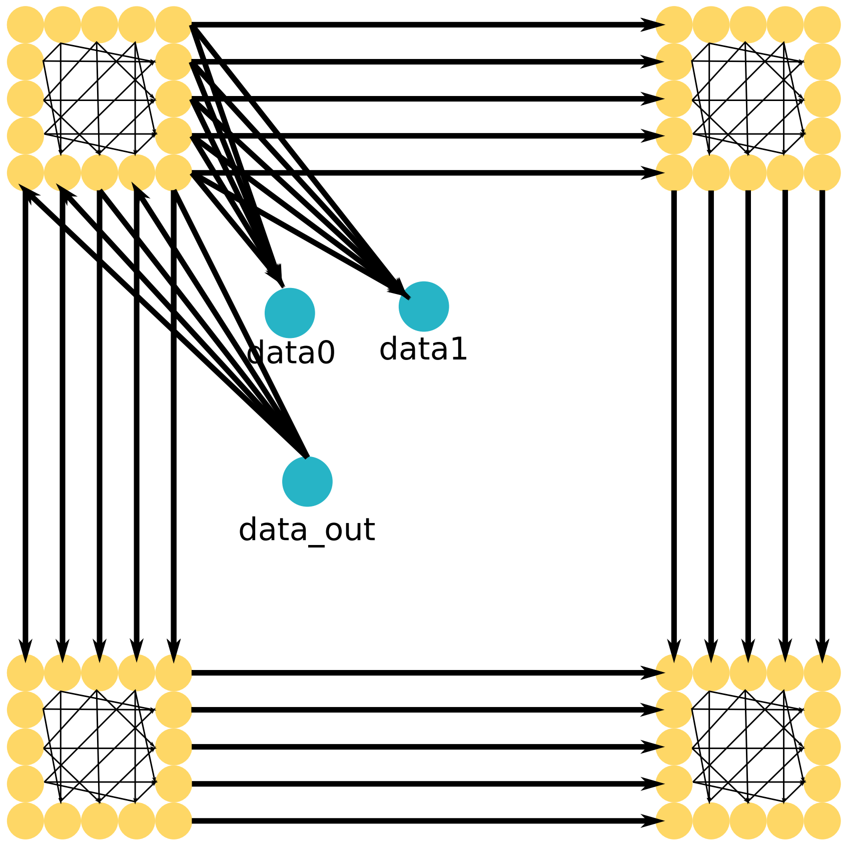
Canal: DSL for Interconnect

- Canal DSL specifies interconnect using a directed graph (DiGraph)
  - Anything connectable in RTL is a node in the DiGraph
  - It introspects the PEak/Lake core to obtain IO information automatically
  - Allows multiple passes to construct and transform the graph



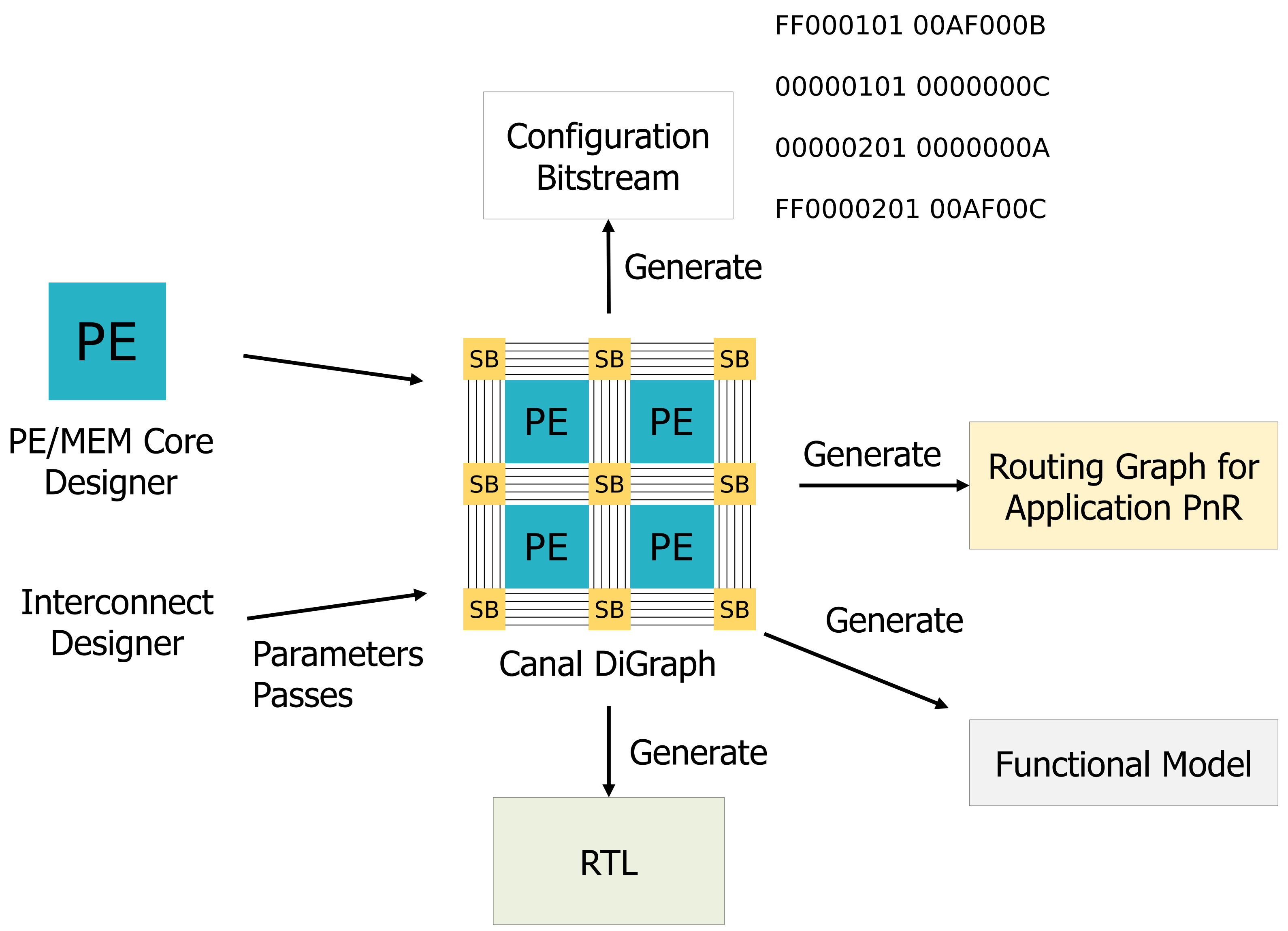
Traditional

Representation



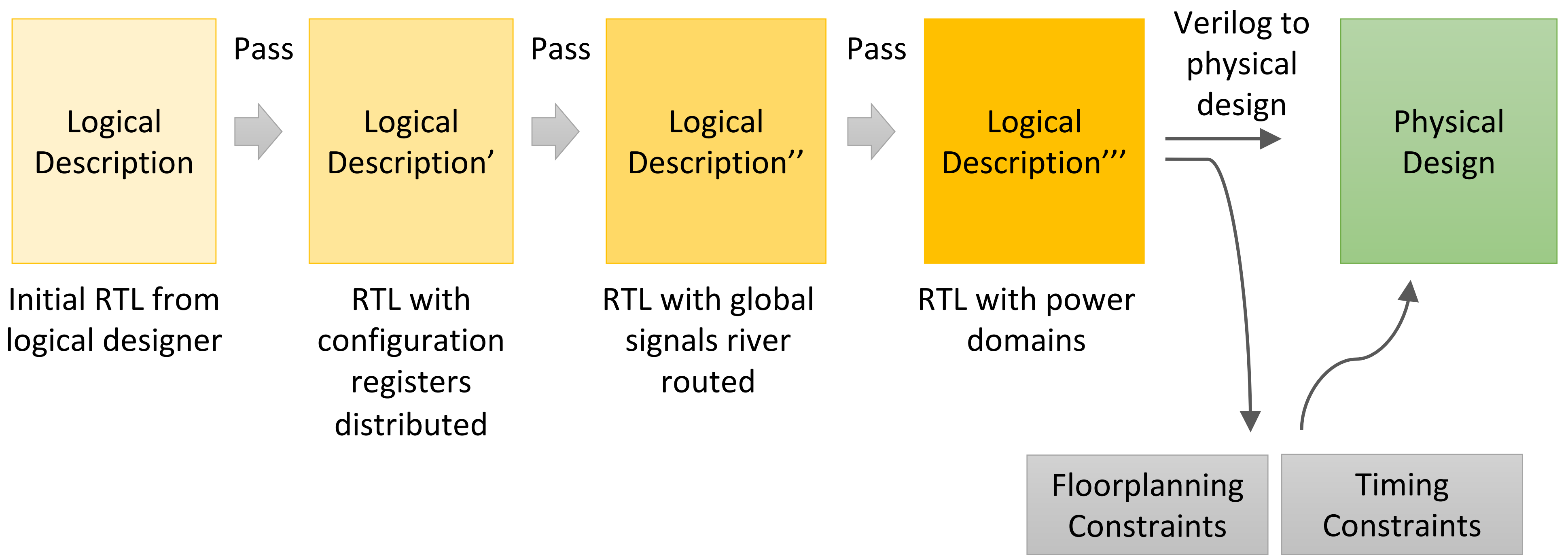
DiGraph Representation

Automatically generates the hardware and collateral for software tools



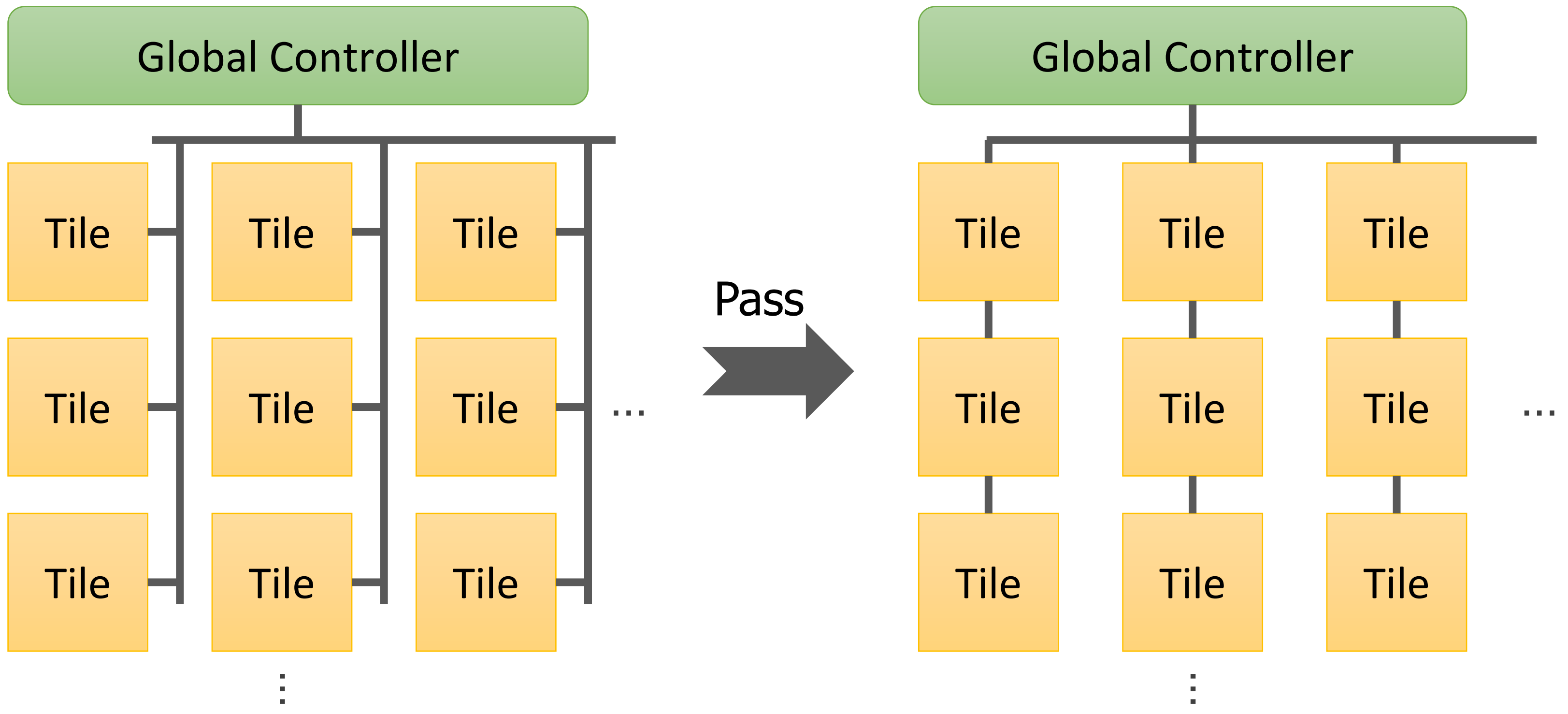
Gemstone: A staged generator

- All our DSLs create RTL in the form of gemstone circuit generator objects
- Gemstone allows multiple passes to
  - Change the RTL
  - Generate non-RTL collateral
- Well-defined primitives on circuit generator objects, such as add/remove ports and instantiate generators/circuits



Logical Description
Pass
Logical Description'
Pass
Logical Description''
Pass
Logical Description'''
Verilog to physical design
Physical Design
Initial RTL from logical designer
RTL with configuration registers distributed
RTL with global signals river routed
RTL with power domains
Floorplanning Constraints
Timing Constraints

Passes modify gemstone generator objects



An example pass that changes fanout global signals to river-routed global signals

Multiple end-to-end flows for the SoC system

- CGRA array-level application tests
- CGRA with control logic (AXI/JTAG) and second-level memory
- CGRA with ARM M3 (full SoC) tests
- These end-to-end flows are required to be “green” throughout the agile development process.

Summary

- We are creating an agile design methodology for highly-efficient accelerator systems (hardware + software) where we
  - Evaluate current system and make incremental improvements
  - Always keep an end-to-end application flow running
  - Maintain separation of concerns in the design process
- We are doing this by creating domain-specific languages (DSLs) that
  - Easily express functionality of specific types of hardware (like processors, memories and interconnect)
  - Generate collateral for all tools in the flow from a single source of truth
  - Allow passes/staged generation for separation of concerns
- We have designed an SoC called Garnet with this approach for image, vision and machine learning applications

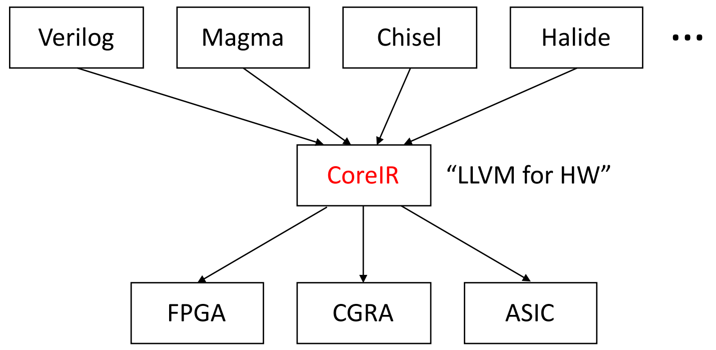
Stanford AHA (Agile HArdware Center)

https://aha.stanford.edu
https://github.com/StanfordAHA

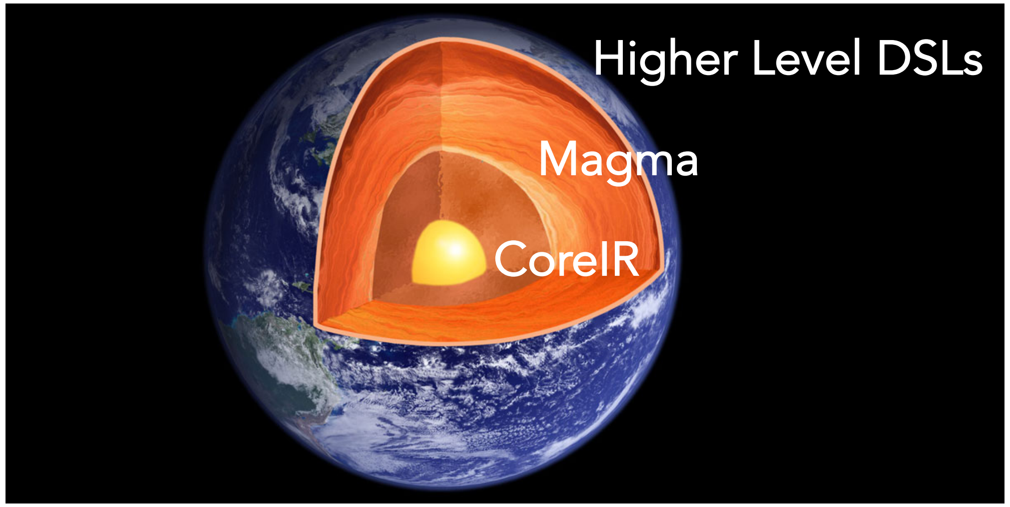
- **Faculty**: Myself, Mark Horowitz, Pat Hanrahan, Clark Barrett, Kayvon Fatahalian
- **Students**: Nikhil Bhagdikar, Alex Carsello, Ross G Daly, Caleb Donovick, David Durst, Kathleen Feng, Teguh Hofstee, Dillon Huff, Taeyoung Kong, Qiaoyi Liu, Makai Mann, Ankita Nayak, Aina Niemetz, Gedeon Nyengele, Raj Setaluri, Jeff Setter, Maxwell Strange, James Thomas, Leonard Truong, Keyi Zhang
- **Advisors**: Rick Bahr, Stephen Richardson

Collaboration Opportunities

- All our DSLs, tools and architectures are open source
- **CoreIR**: Hardware intermediate representation



- **Magma**: Python-embedded HDL
- **Gemstone**: Generator infrastructure on top of Magma
- **Fault**: Unified testing + formal verification for Magma
- **CoSA**: SMT based model checker
- **Peak**: DSL for PEs
- **Lake**: DSL for memories
- **Canal**: DSL for interconnect
- **Halide-to-Hardware**: Application compiler for our SoC/FPGAs
- **Jade**, **Garnet**: Our SoCs



Higher Level DSLs
Magma
CoreIR