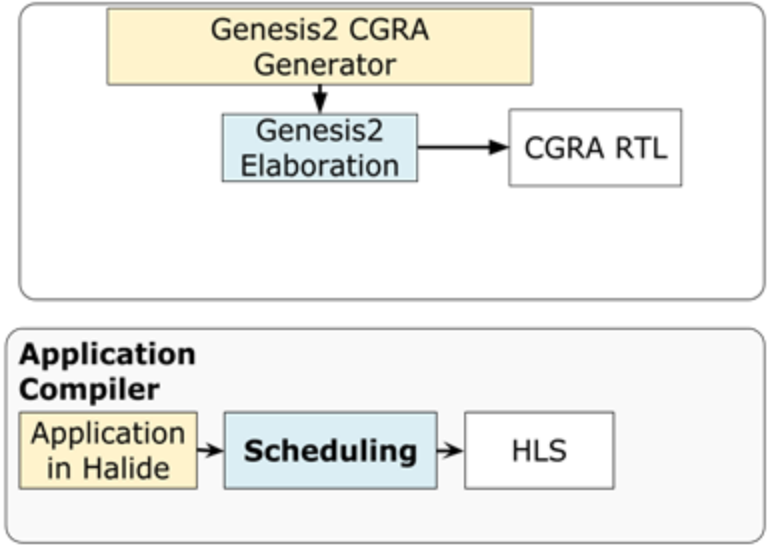
Enabling Agile Hardware Development with the PEak Programming Language

Caleb Donovick & Jackson Melchert

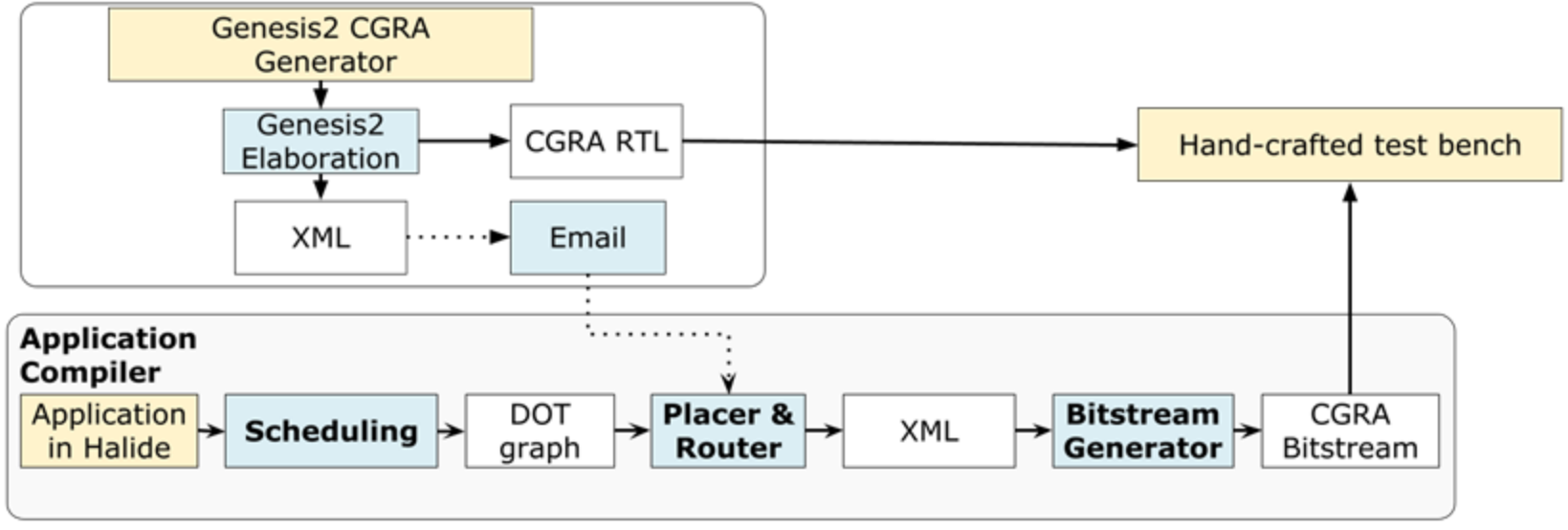
Stanford University

The Stanford Agile Hardware Project (Fall 2016)



Vasilyev, et al. 2016. Evaluating programmable architectures for imaging and vision applications
Pu, et al. 2017. Programming Heterogeneous Systems from an Image Processing DSL

The Stanford Agile Hardware Project (Summer 2017)



PE Spec

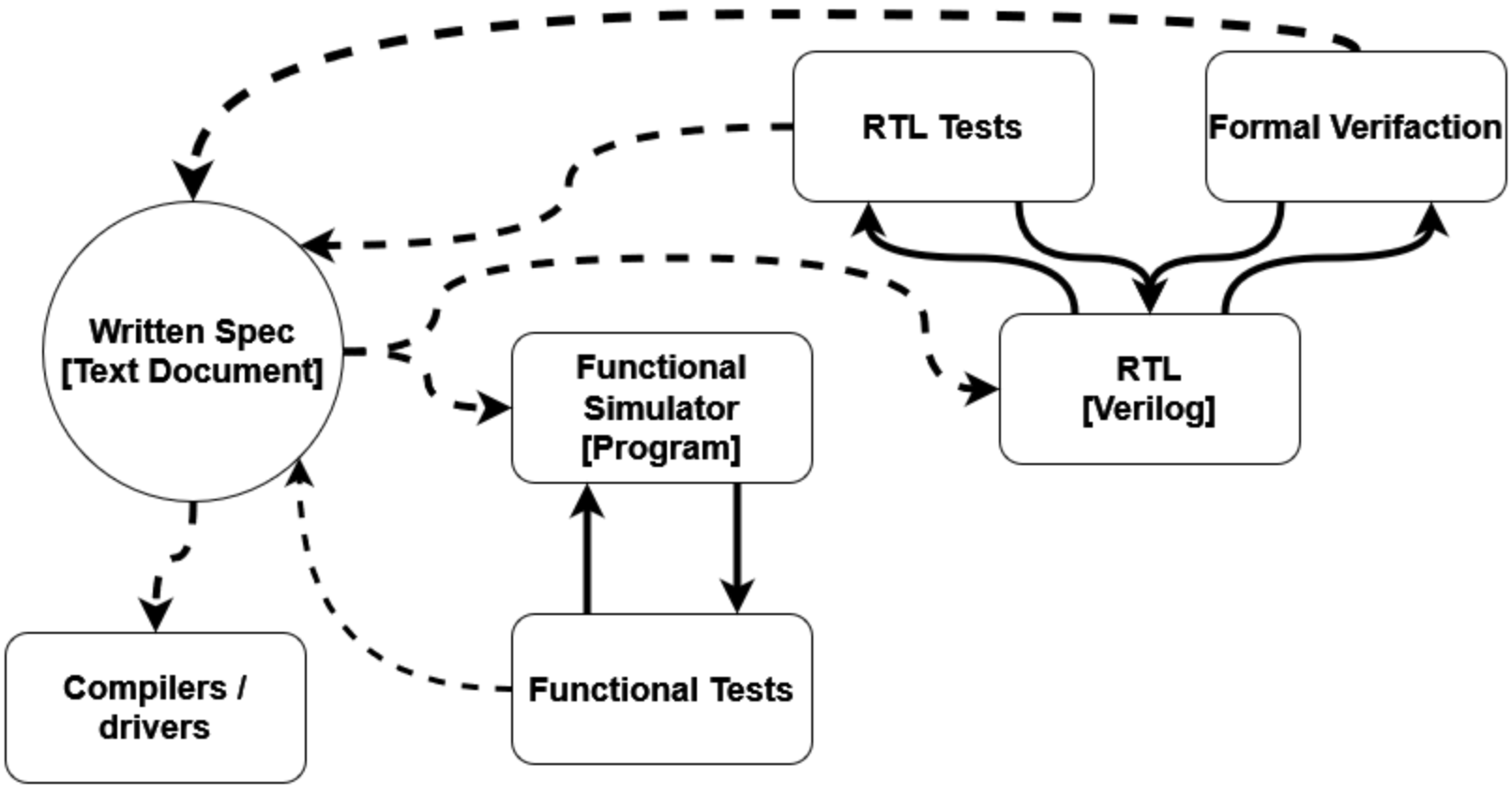
steveri edited this page on Jan 16 · 25 revisions

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

Typical Design Flow

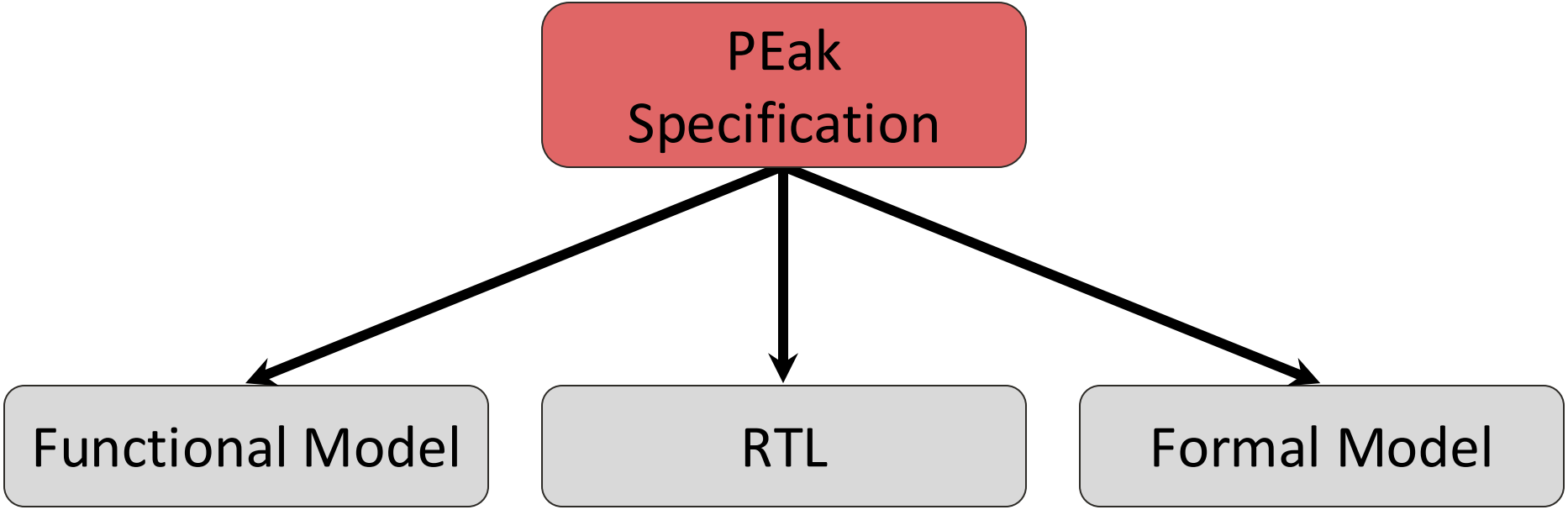


No way to test text-based specification

What are the features of an ideal specification?

- Easy to read and write
- Executable to enable agile test debug
- Generates RTL
- Is a formal model

PEak: The Single Source of Truth



Hardware Types

A Uniform Abstraction for Combinational Logic

Hardware Types Overview

- Abstract interfaces and type constructors for a number of types and kinds
  - BitVector
  - Bit
  - Algebraic data types
  - …
- Semantics derived from SMT-LIB
- Concrete implementations
  - Python
  - SMT
  - Magma

Hwtypes Example

```
from hwtypes import BitVector, SMTBitVector


def add3(a, b, c):
    return a + b + c

x = BitVector[8](1)
y = BitVector[8](2)
z = BitVector[8](3)

print(add3(x,y,z))
# outputs:
# 6
```

```
x = SMTBitVector[8](name='x')
y = SMTBitVector[8](name='y')
z = SMTBitVector[8](name='z')

print(add3(x,y,z))
# outputs:
# (bvadd (bvadd x y) z)
```

Requirements of Magma

- Hwtypes functions must be invoked inside a circuit definition or generator context
- Must specify ports

```
class Add3(m.Circuit):
    io = m.IO(
        a=m.In(m.Bits[8]),
        b=m.In(m.Bits[8]),
        c=m.In(m.Bits[8]),
        out=m.Out(m.Bits[8]),
    )
    io.out @= add3(io.a, io.b, io.c)
```

```
// compiled to verilog using MLIR
module Add3(
  input  [7:0] a,
               b,
               c,
  output [7:0] out
);

  assign out = a + b + c;
endmodule
```

Metaprogramming (generators)

- Embedding in python facilitates metaprogramming
  - Recursion
  - Loops
  - Functional operators (map, fold, …)

Hwtypes Metaprogramming Example

```python
def add_n(bvs):
    n = len(bvs)
    if n == 0:
        raise ValueError()
    elif n == 1:
        return bvs[0]
    else:
        return bvs[0] + add_n(bvs[1:])

x = BitVector[8](1)
y = BitVector[8](2)
z = BitVector[8](3)
w = BitVector[8](4)

print(add_n([x,y,z,w]))
# outputs:
# 10
```

```python
x = SMTBitVector[8](name='x')
y = SMTBitVector[8](name='y')
z = SMTBitVector[8](name='z')
w = SMTBitVector[8](name='w')

print(add_n([x,y,z,w]))
# outputs:
# (bvadd x (bvadd y (bvadd z w)))
```

PEak

A Single Source of Truth

Stanford University

PEak: Behavioral Hardware Types

- Defines circuits as a class
  - Subcomponents declared in __init__ (constructor)
  - Behavior defined in __call__ (function call syntax overload)
  - Ports inferred from type signature
- “Wiring” by calling subcomponents
- If statements generate muxes
- Register access through attributes
- Automatically wraps magma circuits

PEak Design Example

```
import hwtypes as ht
from peak import Peak

class Opcode(ht.Enum):
    Add = 0
    Sub = 1
    Neg = 2
```

```
Word = ht.BitVector[16]

class ALU(Peak):
    def __call__(self, inst: Opcode,
                 I0: Word, I1: Word) -> Word:
        if inst == Opcode.Add:
            return I0 + I1
        elif inst == Opcode.Sub:
            return I0 - I1
        else:
            return -I1
```

Testing

```
import hwtypes as ht
from peak import Peak

class Opcode(ht.Enum):
    Add = 0
    Sub = 1
    Neg = 2

Word = ht.BitVector[16]

class ALU(Peak):
    def __call__(self, inst: Opcode,
                 I0: Word, I1: Word) -> Word:
        if inst == Opcode.Add:
            return I0 + I1
        elif inst == Opcode.Sub:
            return I0 - I1
        else:
            return -I1

return ALU
```

```
alu = ALU()

i0 = ht.BitVector.random(16)
i1 = ht.BitVector.random(16)

# simulate alu
out = alu(Opcode.Add, i0, i1)
assert out == i0+i1
```

Testing

```
import hwtypes as ht
from peak import Peak

class Opcode(ht.Enum):
    Add = 0
    Sub = 1
    Neg = 2

Word = ht.SMTBitVector[16]

class ALU(Peak):
    def __call__(self, inst: Opcode,
                 I0: Word, I1: Word) -> Word:
        if inst == Opcode.Add:
            return I0 + I1
        elif inst == Opcode.Sub:
            return I0 - I1
        else:
            return -I1

return ALU
```

```
alu = ALU()

i0 = Word()
i1 = Word()

out = alu(Opcode.Add, i0, i1)

from pysmt import shortcuts as sc

with sc.Sover("z3") as s:
    s.add_assertion((out != i0+i1).value)
    if s.solve():
        print("Counter example found")
    else:
        print("Verified add")
```

Main Takeaways

- PEak gets us closer to the ideal specification:
  - PEak is python so it's easy to read and write
  - Through hwtypes we can execute and verify our design
  - PEak generates RTL through Magma
  - PEak provides easy and direct access to a formal model