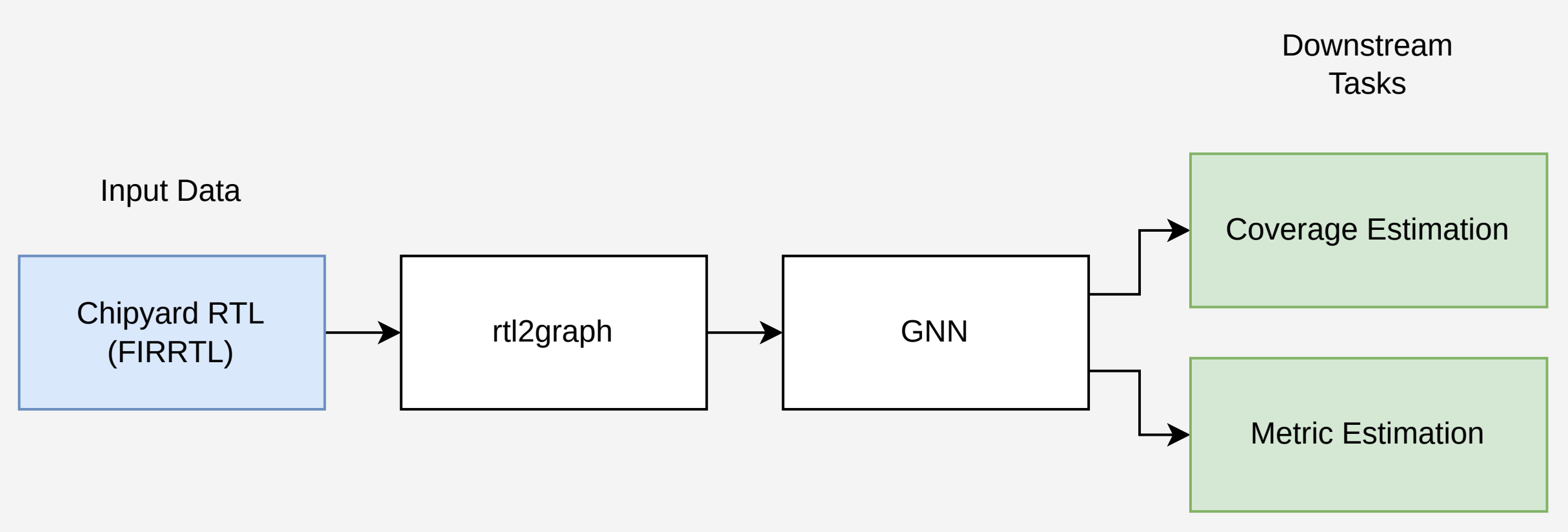
rtl2graph: Circuit Representation Learning using GNNs

Viansa Schmulbach*, Nikhil Jha*, Josh Kang, Vighnesh Iyer, John Wawrzynek, Bora Nikolic
UC Berkeley

Motivation

- Running digital simulation and elaboration tools to check test coverage and get estimates for power and area is time-consuming.
- Digital designs are hierarchical in how we design them, so leveraging this hierarchy may allow self-supervised learning (SSL) techniques to learn useful latent representations of netlists.
- Different information is available at different levels of abstraction (hi-FIRRTL vs lo-FIRRTL), and so we may need to use different techniques to get the best representation for each.

Our Proposal



- Because collecting a full dataset with critical path and coverage data is expensive, we first train the model with pre-text tasks relevant to learning the structure and semantics of netlists (e.g learning to predict randomly masked node attributes).
- Then, we fine-tune the pre-trained model on downstream tasks of predicting verification test coverage and estimating power and area metrics without running expensive synthesis.

We built a tool (`rtl2graph`) that transforms a FIRRTL netlist into a graph that can be used for GNN-based representation learning.

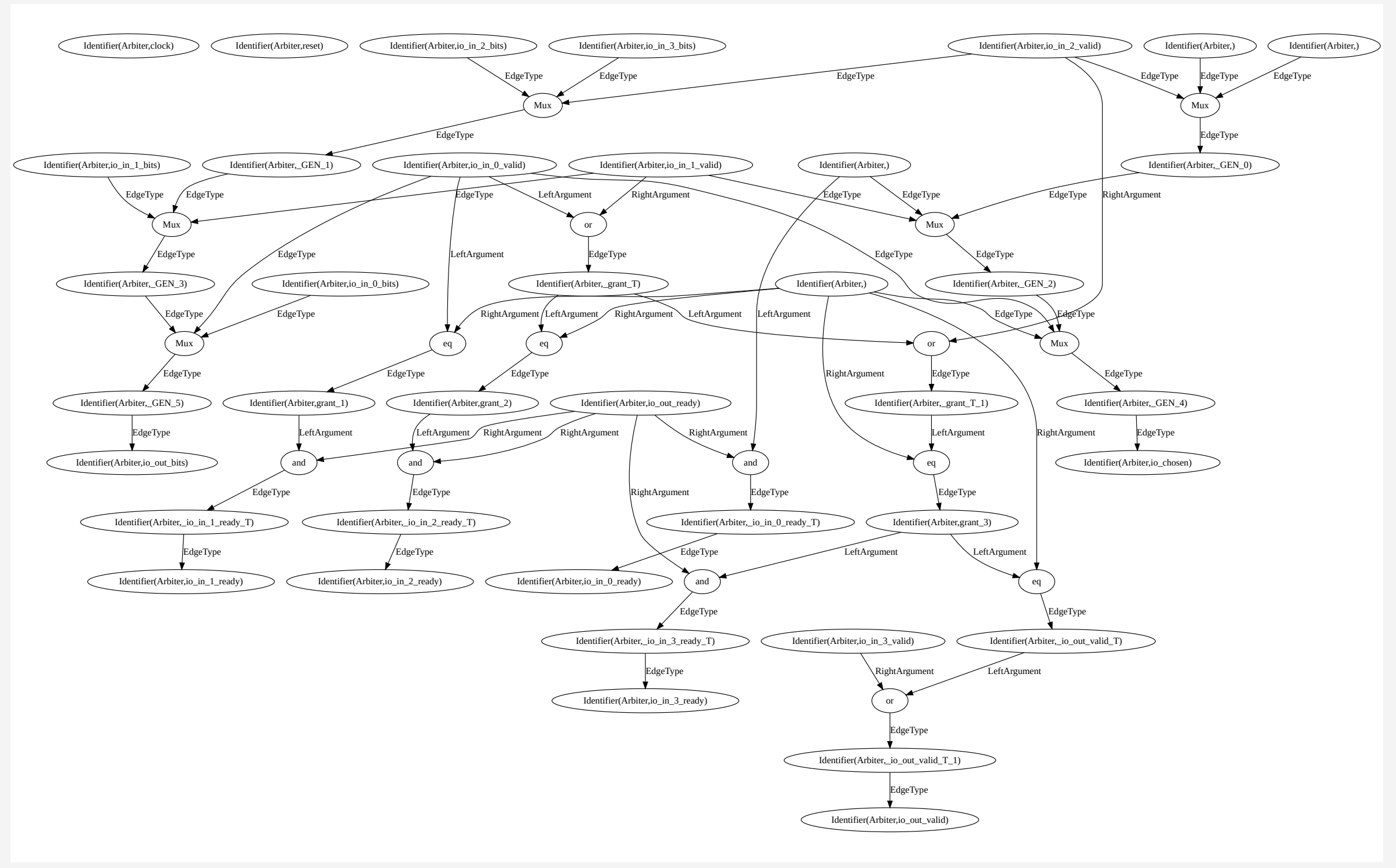
Graph Transform Implementation

The lo-FIRRTL AST is analyzed in two passes

1. The first pass instantiates nodes for each hardware structure (IOs, operators, registers, memories)
2. The second pass creates edges to represent inter-node connections

Chisel Arbiter

```
class Arbiter[T](gen: T, n: Int) extends Module {
  val io = IO(new Bundle {
    val in = Flipped(Vec(n, Decoupled(gen)))
    val out = Decoupled(gen)
  })
  // ... more RTL
}
```



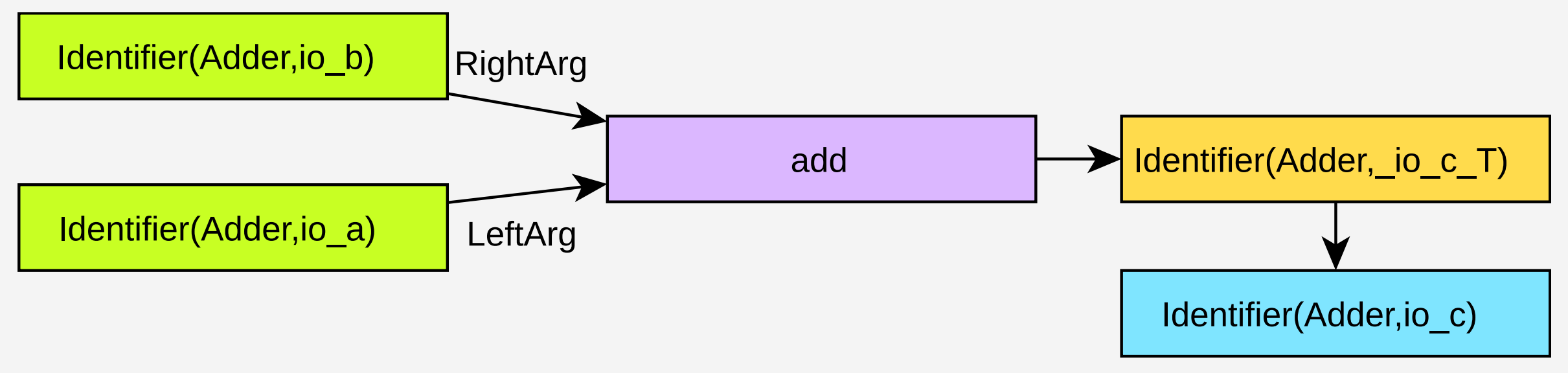
Node and Edge Representations

- Different node types represent module inputs, module outputs, primary operators, muxes, registers, and memory so that the model can distinguish between these components. Left inputs and right inputs, as edges, are also distinguished where applicable.
- Combinatorial primary operators (e.g. `and`, `or`) are represented by the same node type but have an node attribute to distinguish different operators.

```
class Adder extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(8.W))
    val b = Input(UInt(8.W))
    val c = Output(UInt(9.W))
  })
  io.c := io.a +& io.b
}
```

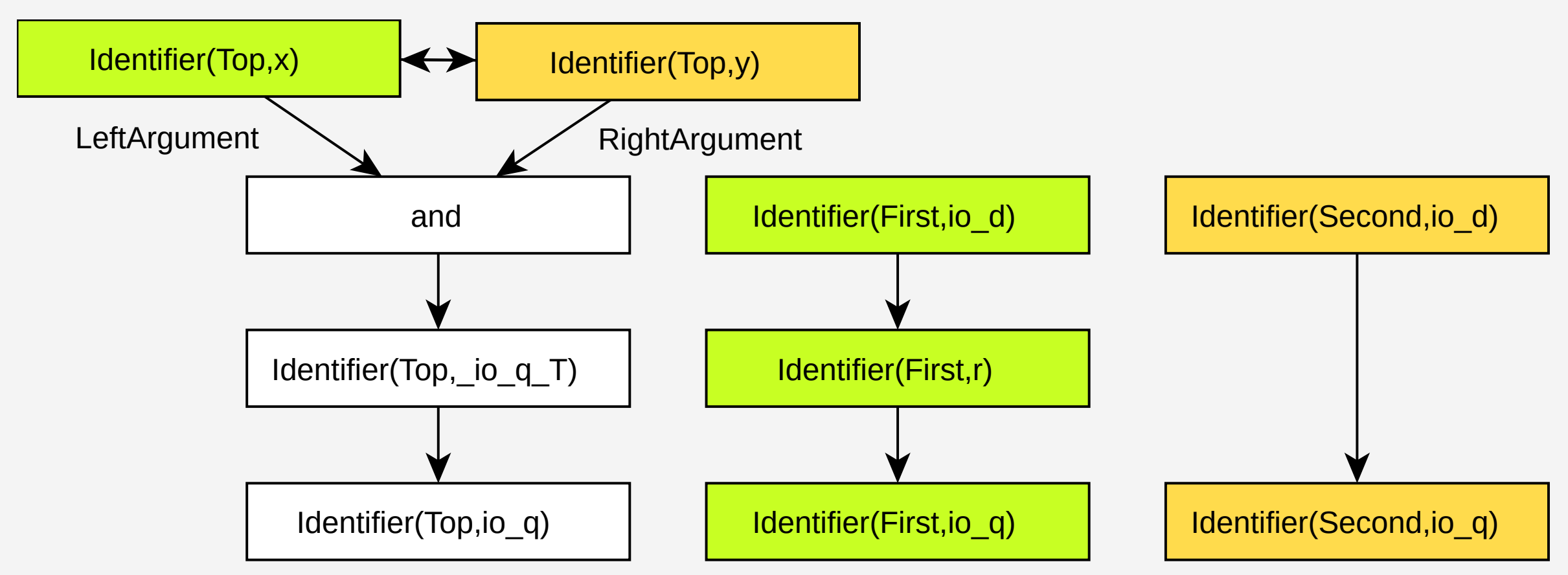
```
circuit Adder :
  module Adder :
    input io_a : UInt<8>
    input io_b : UInt<8>
    output io_c : UInt<9>

    node _io_c_T = add(io_a, io_b)
    io_c <= _io_c_T
```



Subfield Reference Simplification

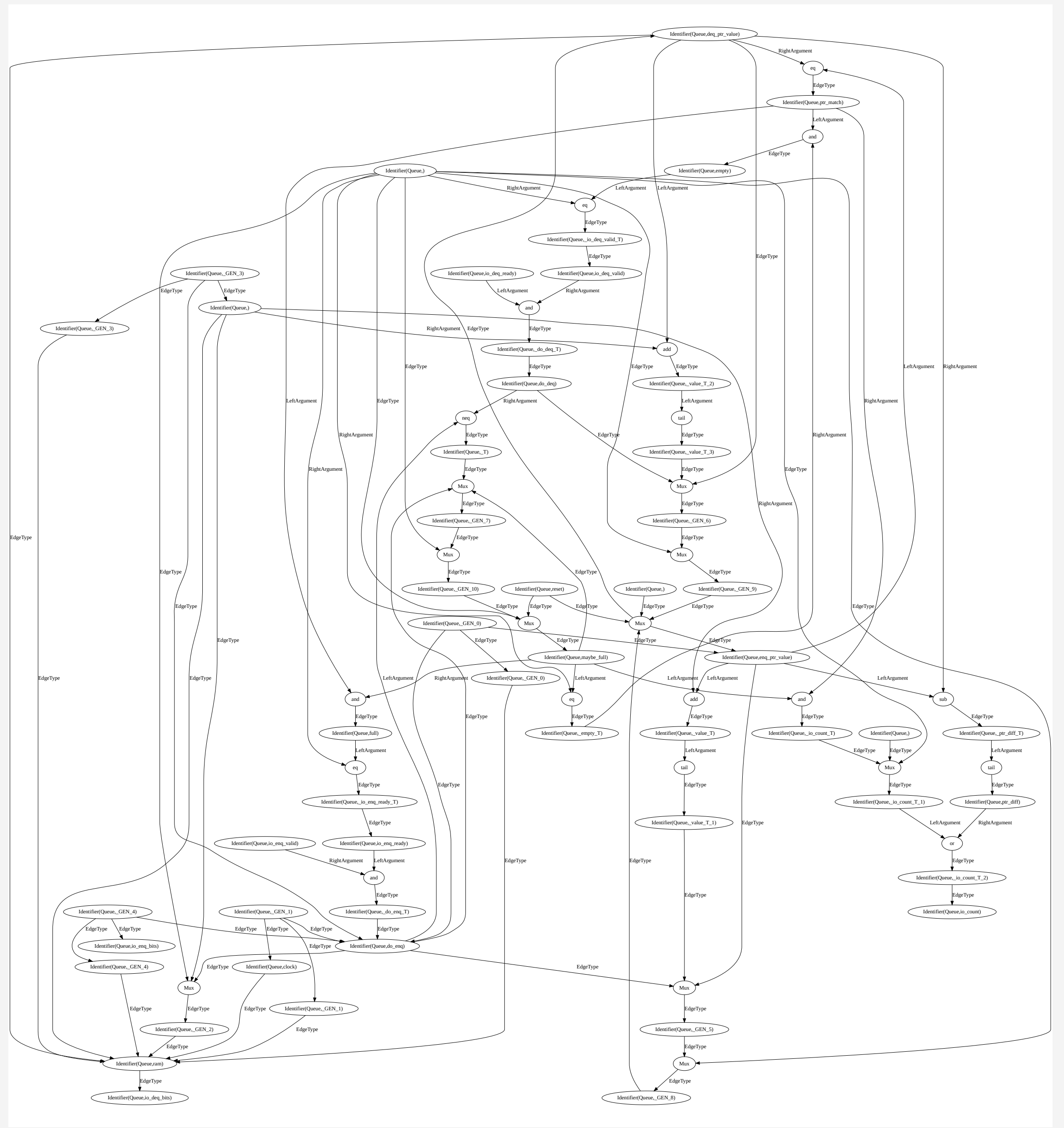
Subfield references (ex. `a.b.c` ← `d`) are "unwrapped" such that the expression's right-hand side is connected to the topmost expression of the left-hand side (d is connected as an input to a).



Edges to and from SRAMs are modeled uniformly, without regard to their underlying semantics (address, data, mask).

Chisel Queue

```
class Queue[T](gen: T, entries: Int) extends Module {
  val io = IO(new QueueIO(gen, entries, hasFlush))
  val ram = SyncReadMem(entries, gen)
  val enq_ptr = Counter(entries)
  val deq_ptr = Counter(entries)
  when(io.enq.fire) {
    ram(enq_ptr.value) := io.enq.bits
    enq_ptr.inc()
  }
  when(do_deq) {
    deq_ptr.inc()
  }
  io.deq.valid := !empty
  io.enq.ready := !full
  // ... more RTL
}
```

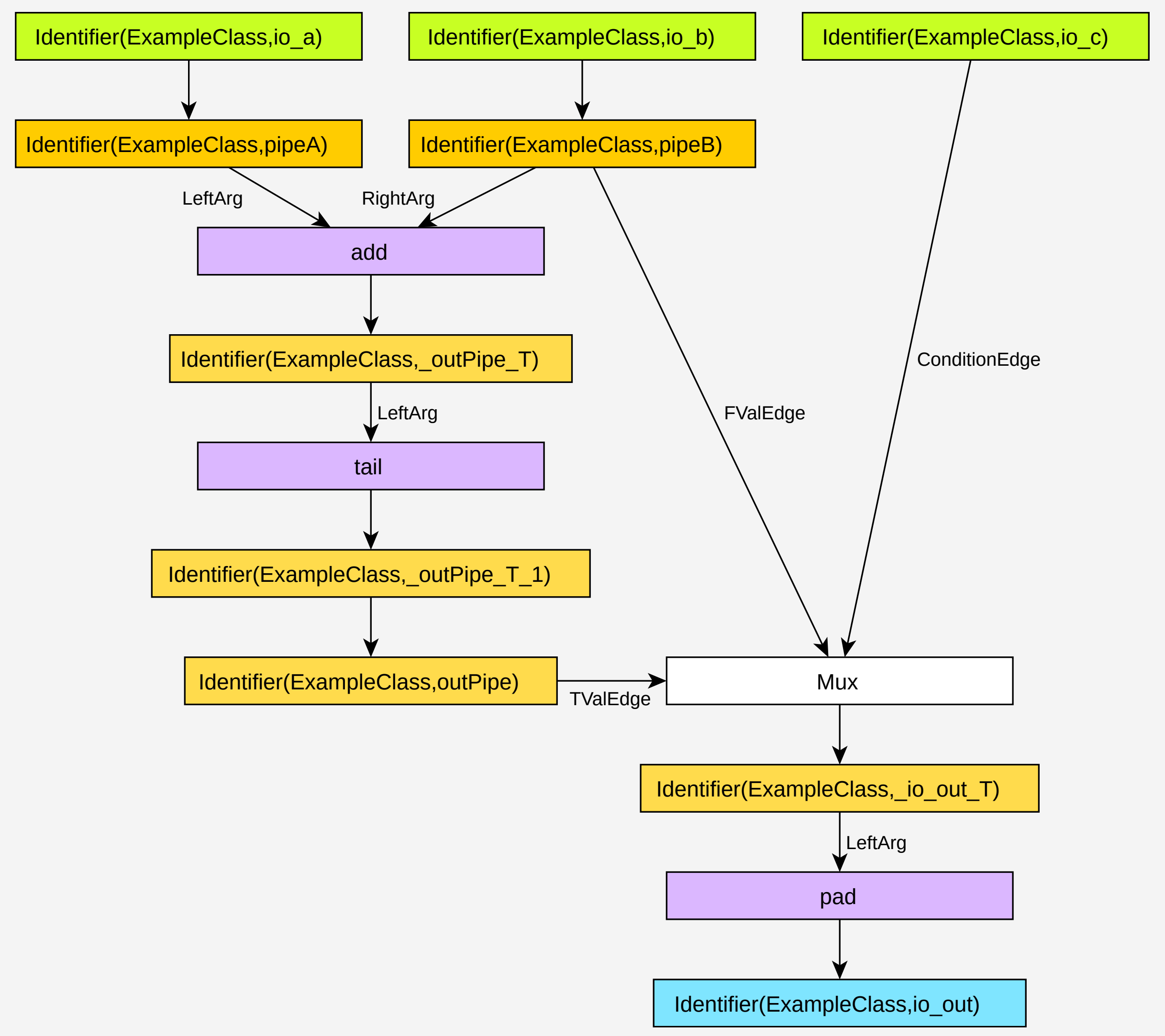


Complete Example (with Chisel + FIRRTL)

```
class ExampleClass extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(8.W))
    val b = Input(UInt(8.W))
    val c = Input(Bool())
    val out = Output(UInt(16.W))
  })
  val pipeA = RegNext(io.a)
  val pipeB = RegNext(io.b)
  val outPipe = RegNext(pipeA + pipeB)
  io.out := chisel3.Mux(io.c, outPipe, pipeB)
}
```

```
module ExampleClass :
  [declarations omitted]
  node _outPipe_T = add(pipeA, pipeB)
  node _outPipe_T_1 = tail(_outPipe_T, 1)
  node _io_out_T = mux(io_c, outPipe, pipeB)

  io_out = pad(_io_out_T, 16)
  pipeA = io_a
  pipeB = io_b
  outPipe = _outPipe_T_1
```


Identifier(ExampleClass,io_a)
Identifier(ExampleClass,io_b)
Identifier(ExampleClass,io_c)
Identifier(ExampleClass,pipeA)
Identifier(ExampleClass,pipeB)
LeftArg
RightArg
add
Identifier(ExampleClass,_outPipe_T)
LeftArg
tail
Identifier(ExampleClass,_outPipe_T_1)
FValEdge
ConditionEdge
Identifier(ExampleClass,outPipe)
TValEdge
Mux
Identifier(ExampleClass,_io_out_T)
LeftArg
pad
Identifier(ExampleClass,io_out)

Evaluation

- For very small graphs, tests were implemented by ensuring the output graph matched a manually constructed expected graph.
- For larger graphs, test were done by exporting the graph to Graphviz format and inspecting the output graph.

Next Steps

- Pre-train a GNN for circuit representation embeddings by masking out patches of the graphs we have generated.
- Fine-tune or augment the GNN for our target downstream tasks (metric estimation, coverpoint estimation).

https://github.com/vighneshiyer/rtl2graph/

{ansa, nikhiljha, minwoo_kang, vighnesh.iyer, johnw, bora}@berkeley.edu

SLICE Retreat, Summer 2023