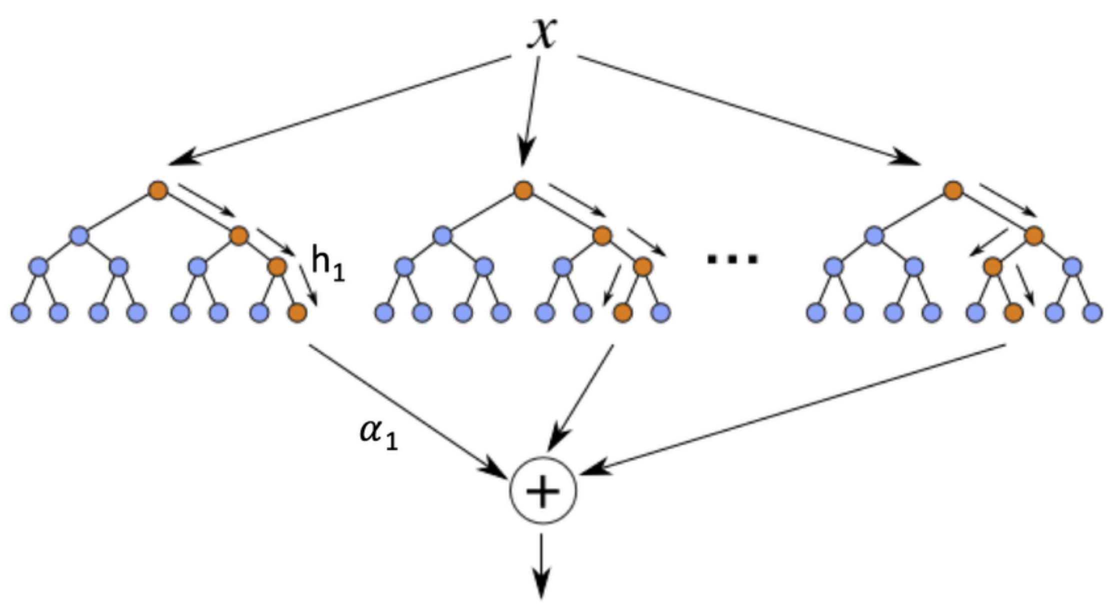
Under the Canopy: Exploring Conifer for Low-Latency Decision Forests on FPGAs

Sioni Summers

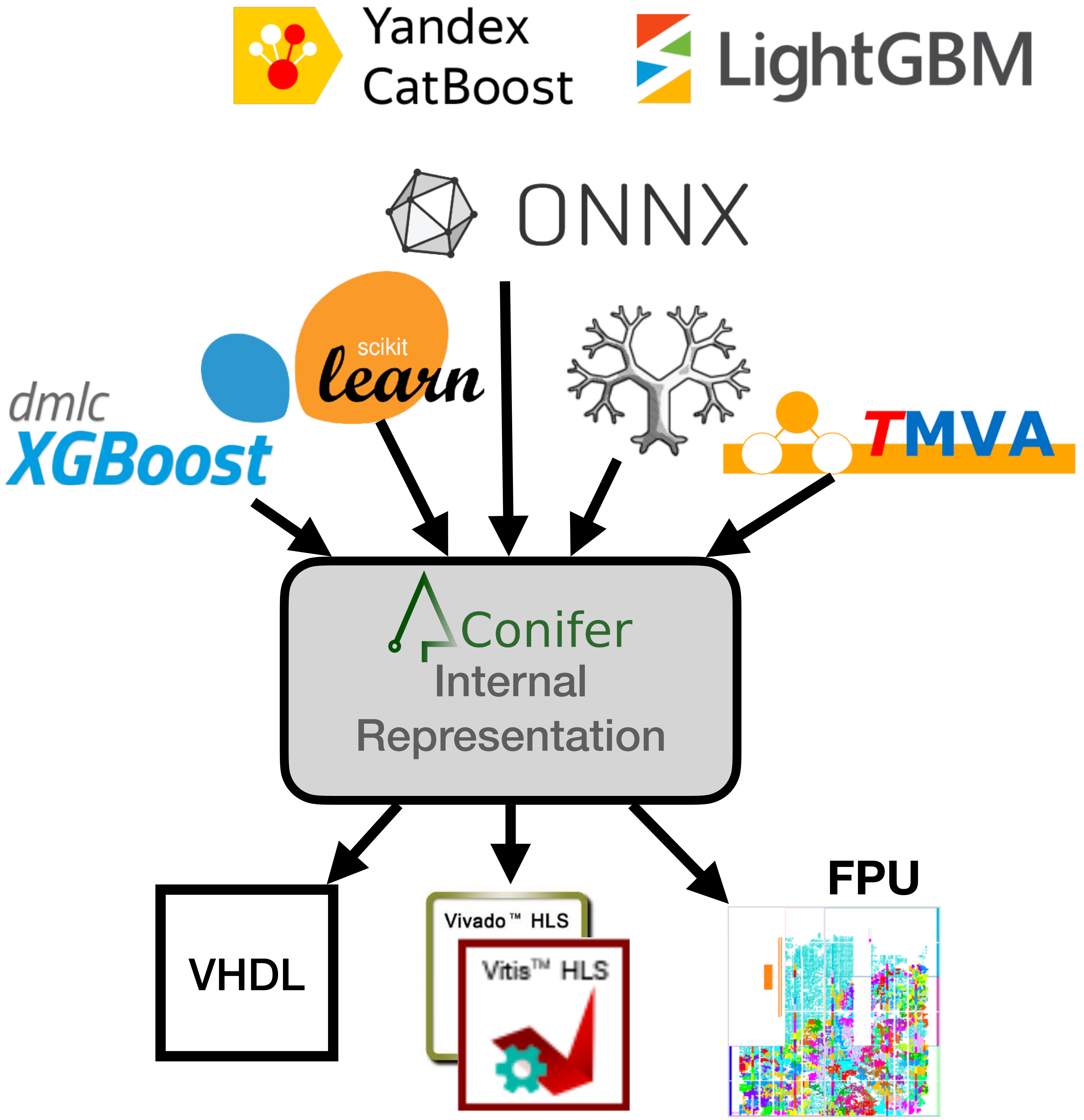
sioni@cern.ch sioni.web.cern.ch

12th June 2024

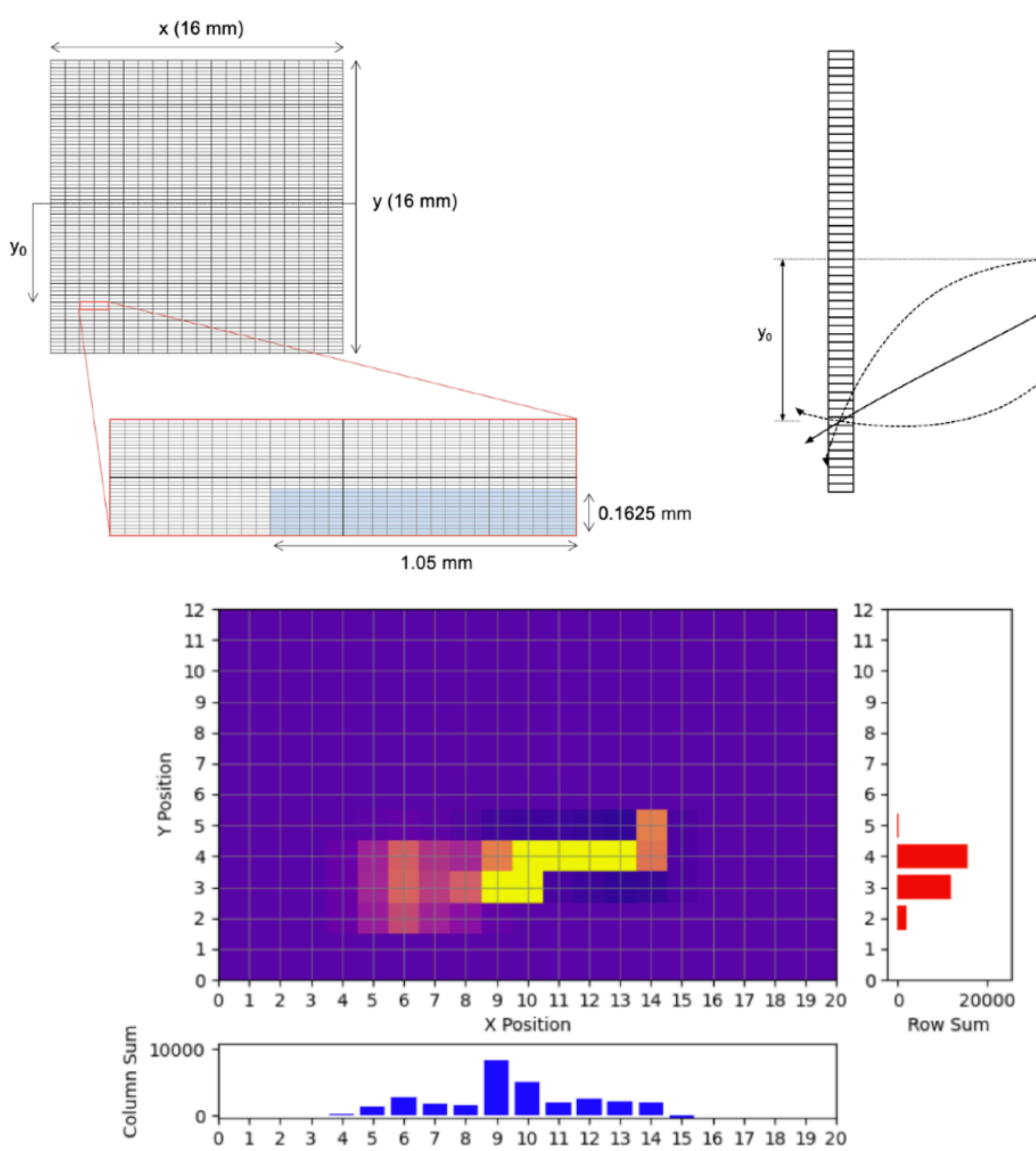
Conifer for Decision Forests

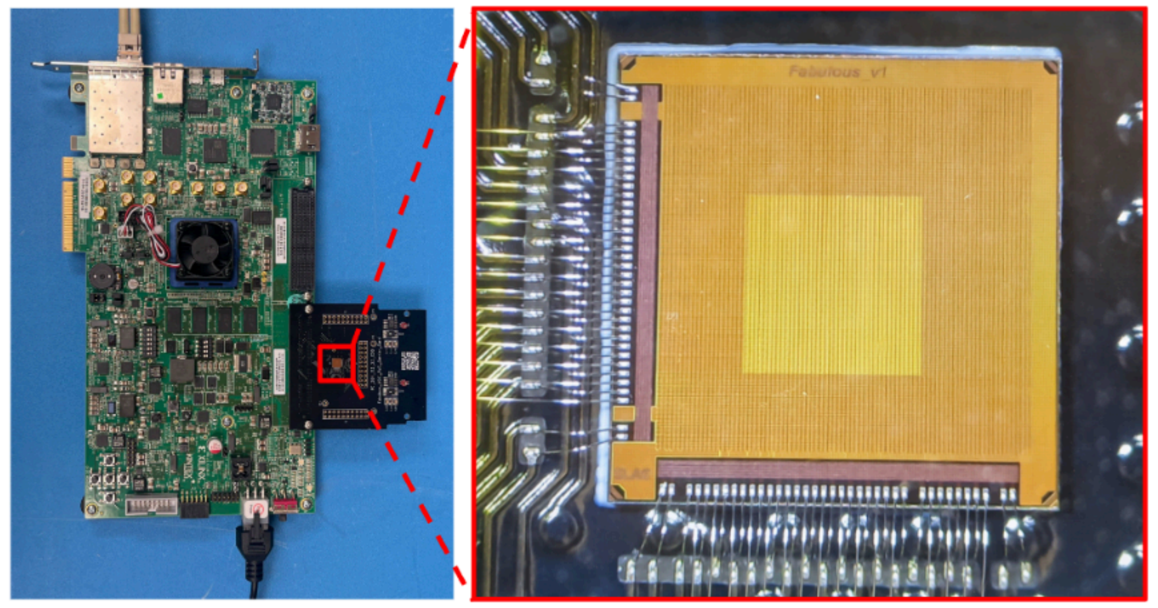


- Decision Forests are still relevant for edge / constrained ML:
  - Fast, lightweight, robust (arXiv:2207.08815, IML keynote)
- **conifer** is a tool to map DFs onto FPGA firmware
  - On Python Package Index: `pip install conifer`
- A Decision Tree *splits* on data variables until reaching a *leaf*
  - Leaves associate a *score* corresponding to prediction probability
- A Decision Forest is an ensemble of Decision Trees
  - Randomisation of each DT as a form of regularisation
  - Ensemble score is an aggregation over trees e.g. sum
- **conifer** reads from popular DF training tools and writes FPGA projects
  - Implemented with high parallelism for low latency and high throughput
- This talk will present the implementations and design considerations

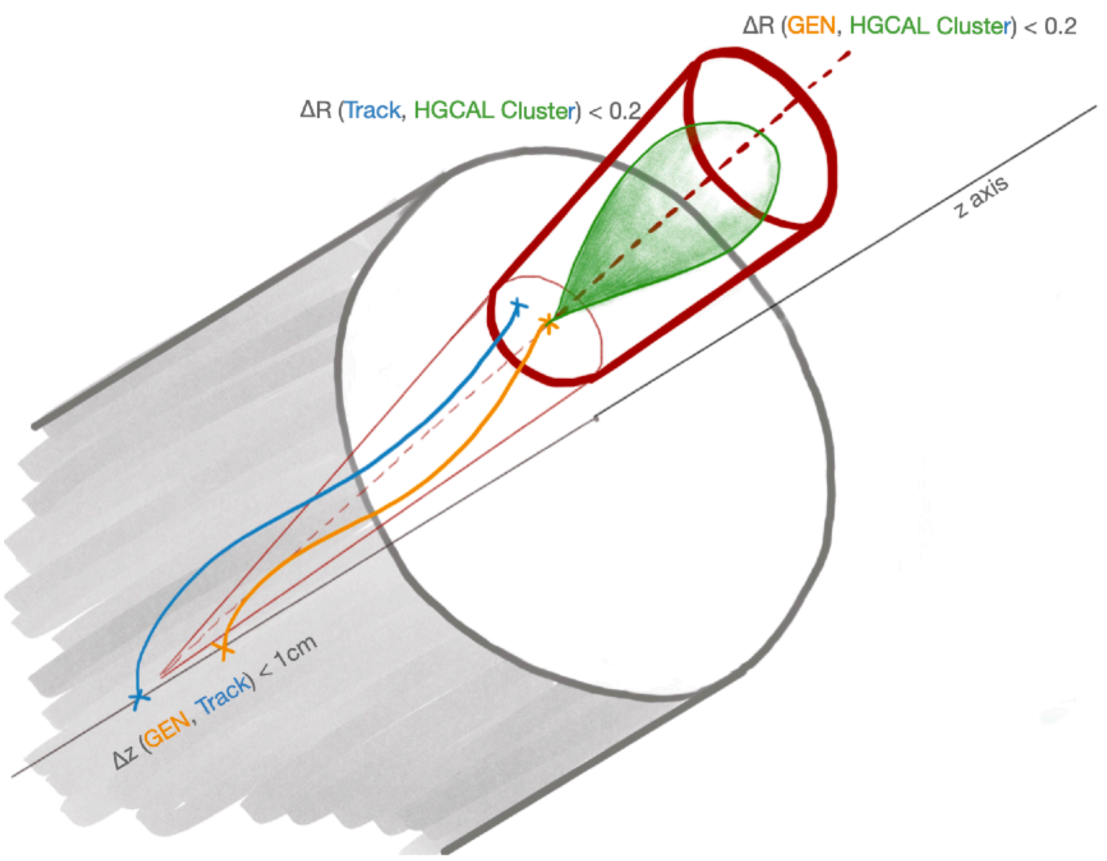


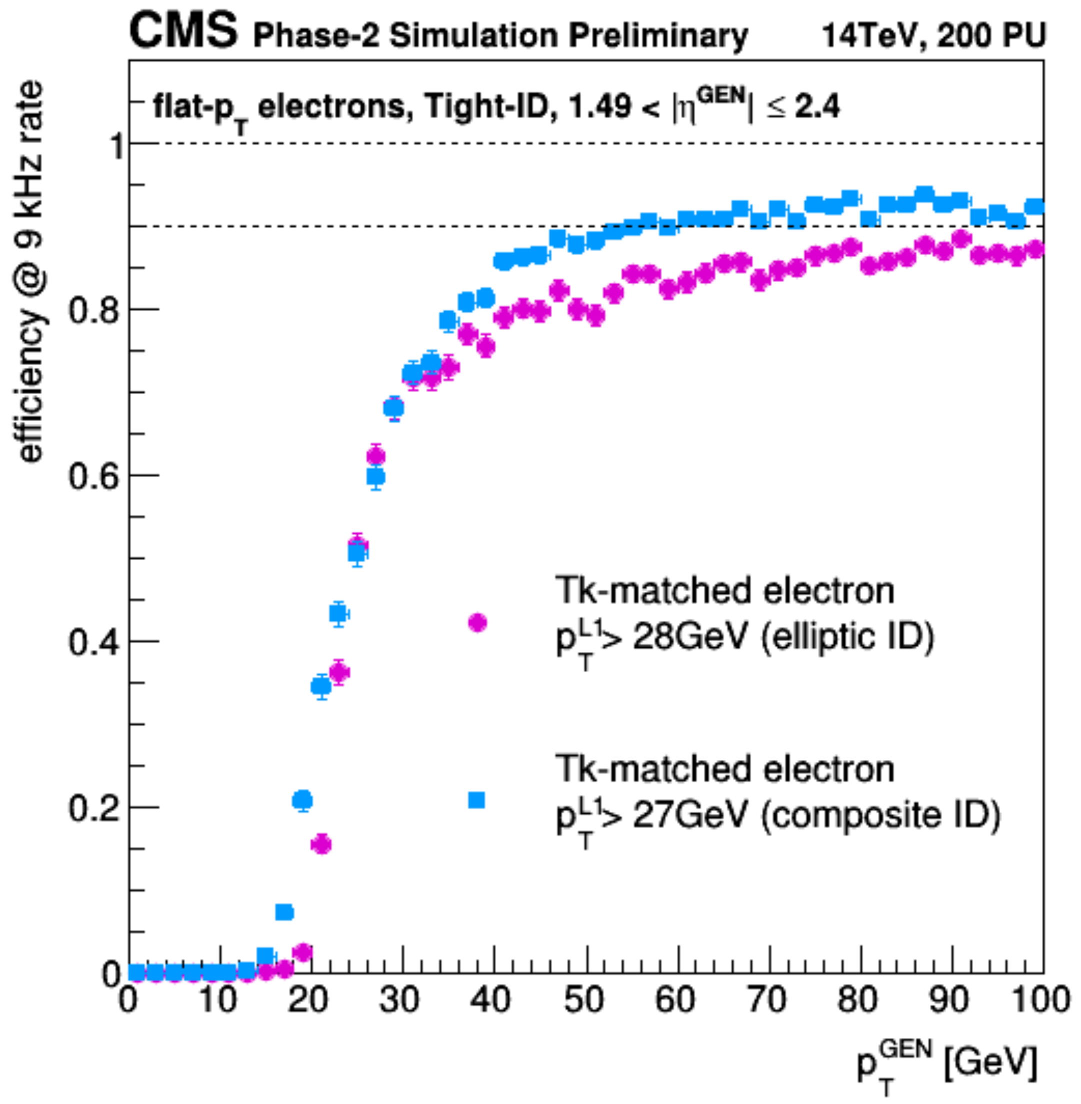
conifer applications





pT filtering in an eFPGA in a tracking detector frontend (25 ns latency, 500 LUTs)





Electron reconstruction in CMS Phase 2 Level 1 Trigger
(< 50 ns latency)

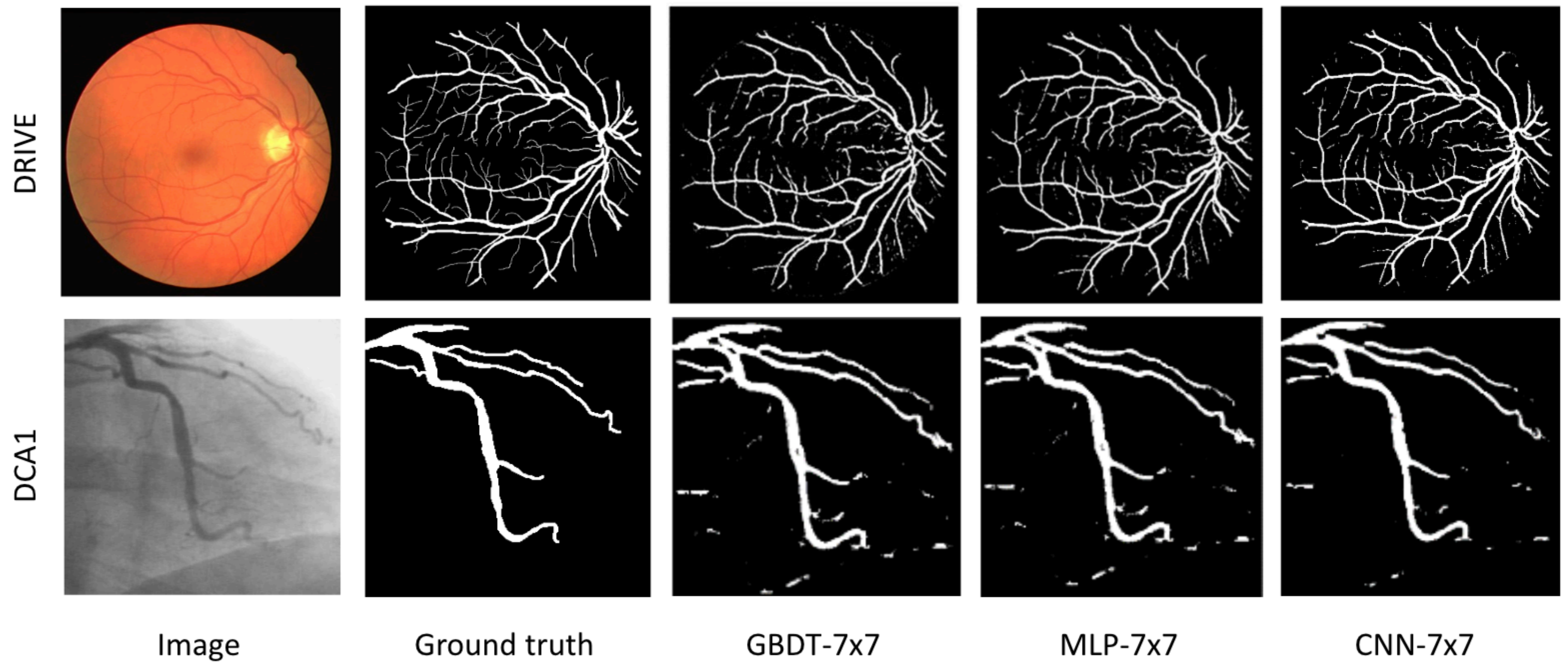
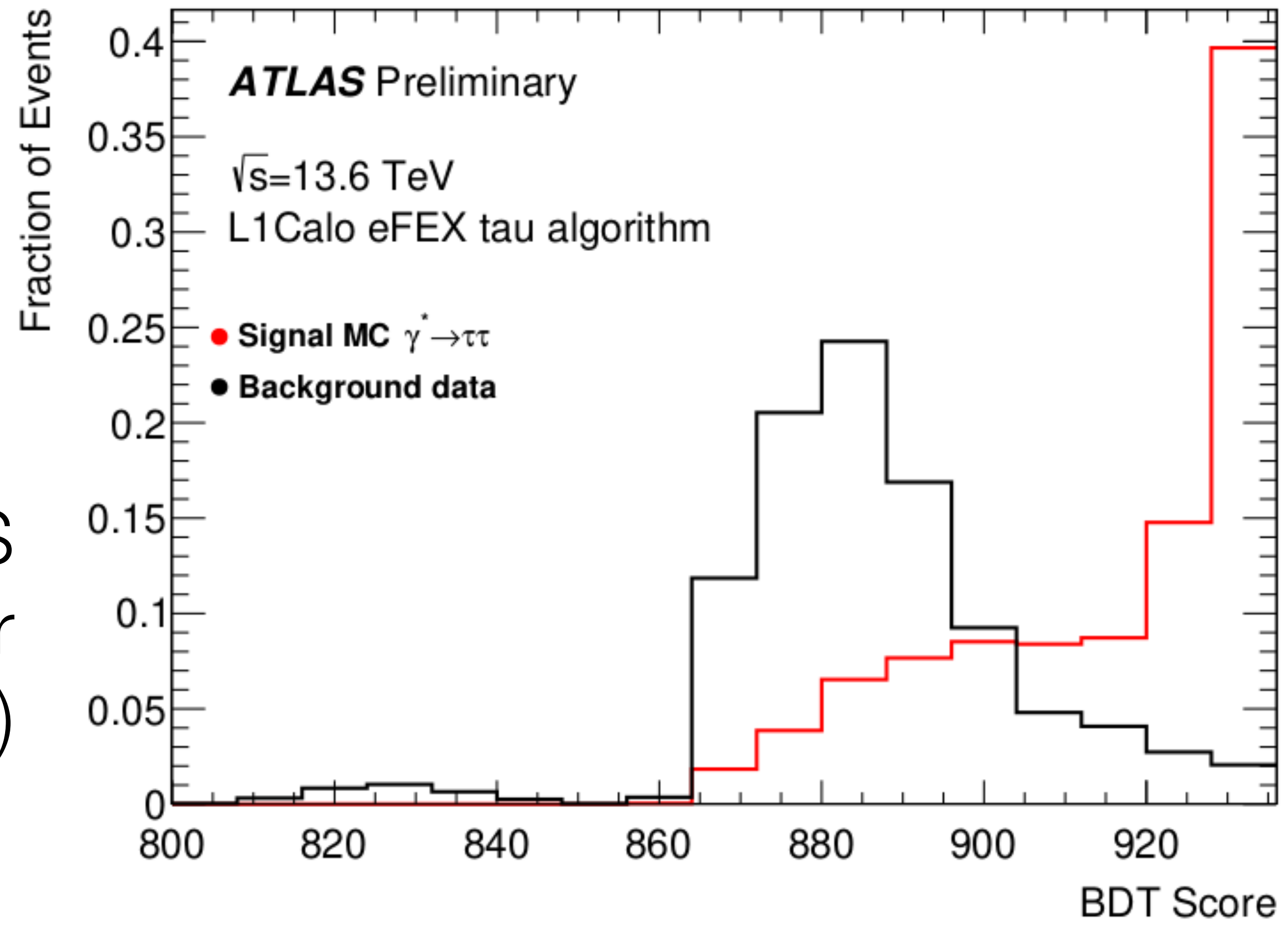


Image segmentation for blood vessels tracking in an embedded medical device (1779 FPS at 3.8 W)

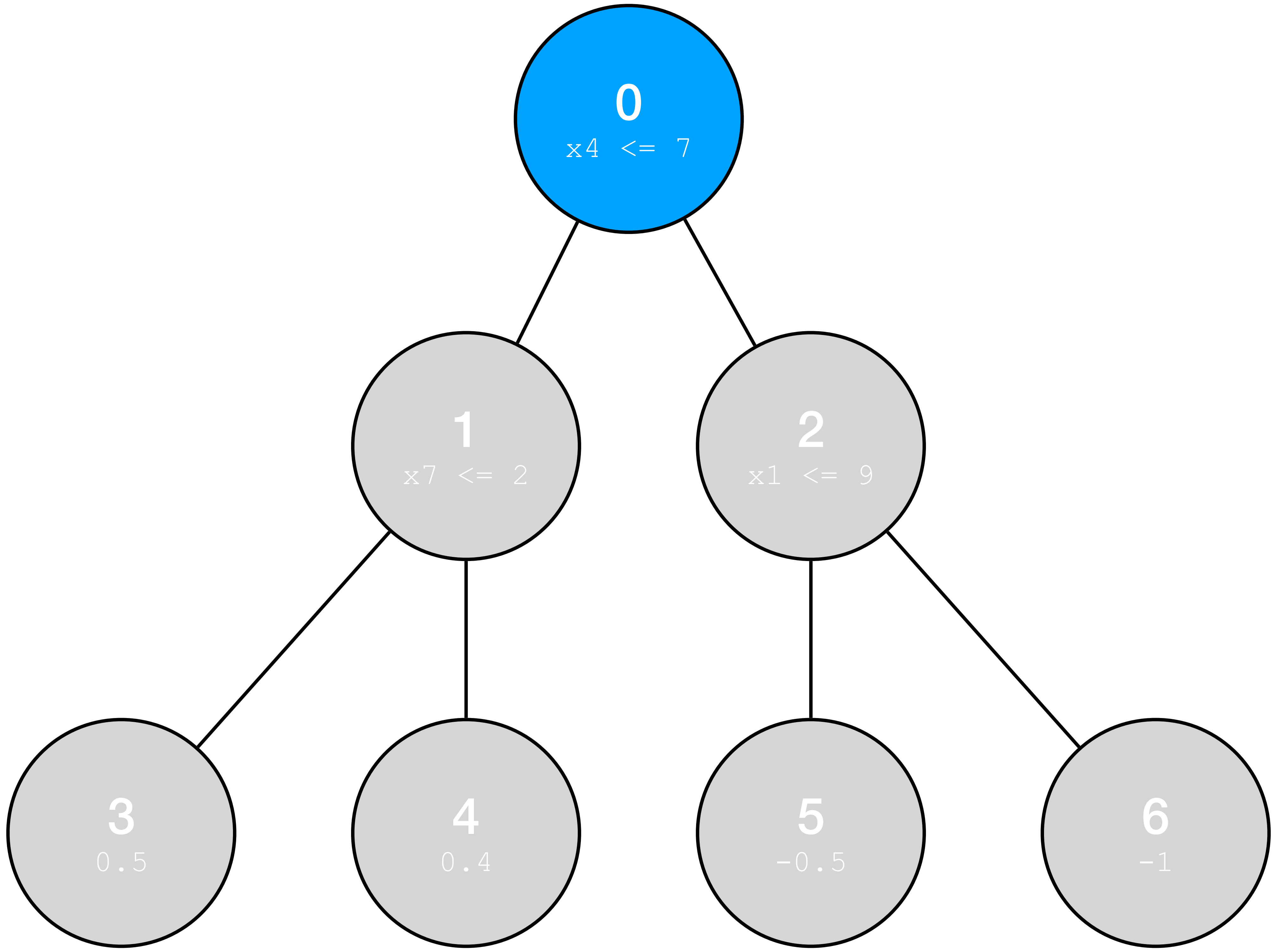
Tau reconstruction in ATLAS Run 3 calorimeter trigger (see David Reikher talk)



Decision Tree Inference

- Start at the root node - compare the selected feature with the threshold, go left or right depending on result

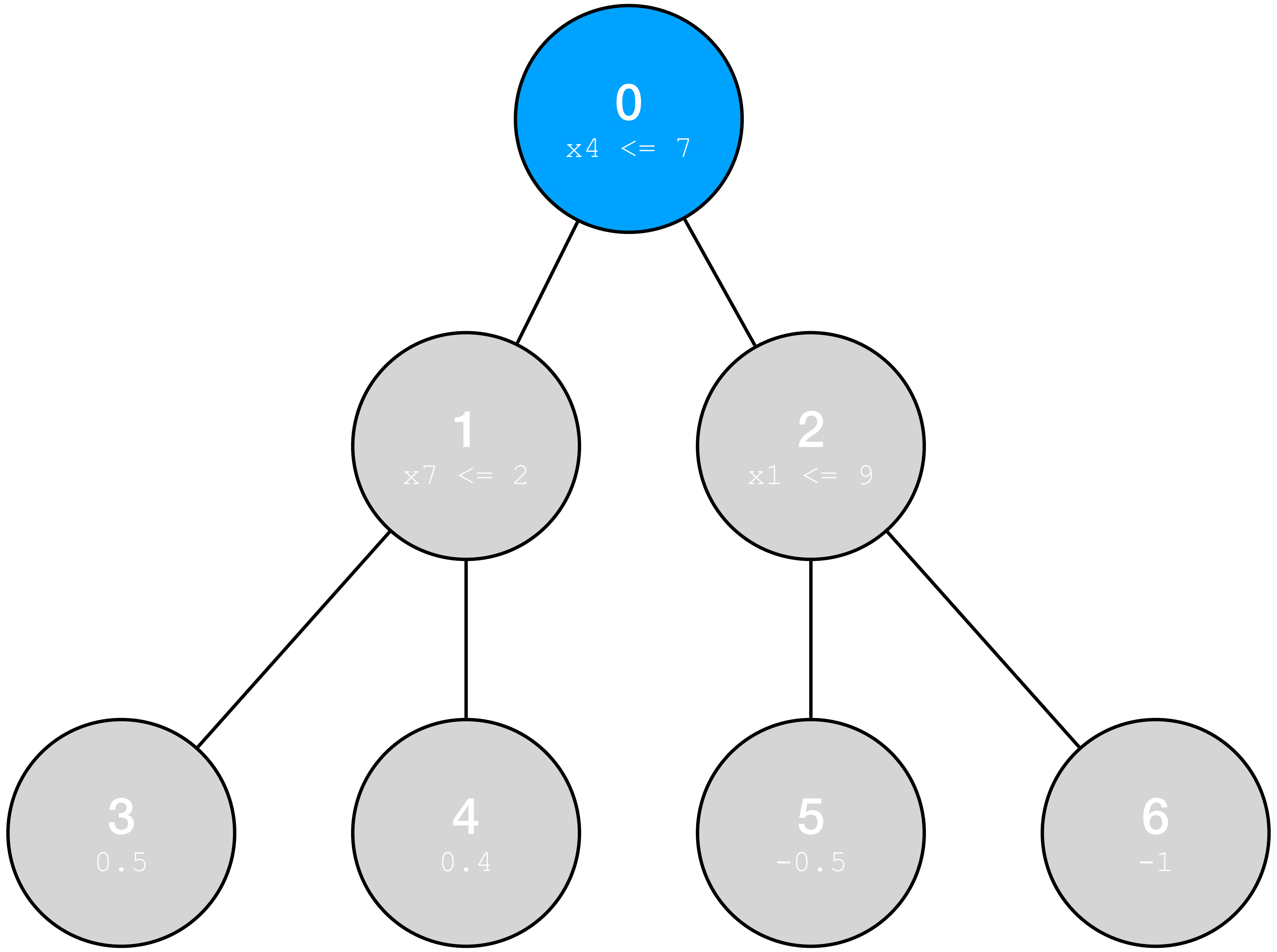
$$x = [\ x_0,\ x_1,\ x_2,\ x_3,\ x_4,\ x_5,\ x_6,\ x_7\ ]$$



Decision Tree Inference

- Start at the root node - compare the selected feature with the threshold, go left or right depending on result

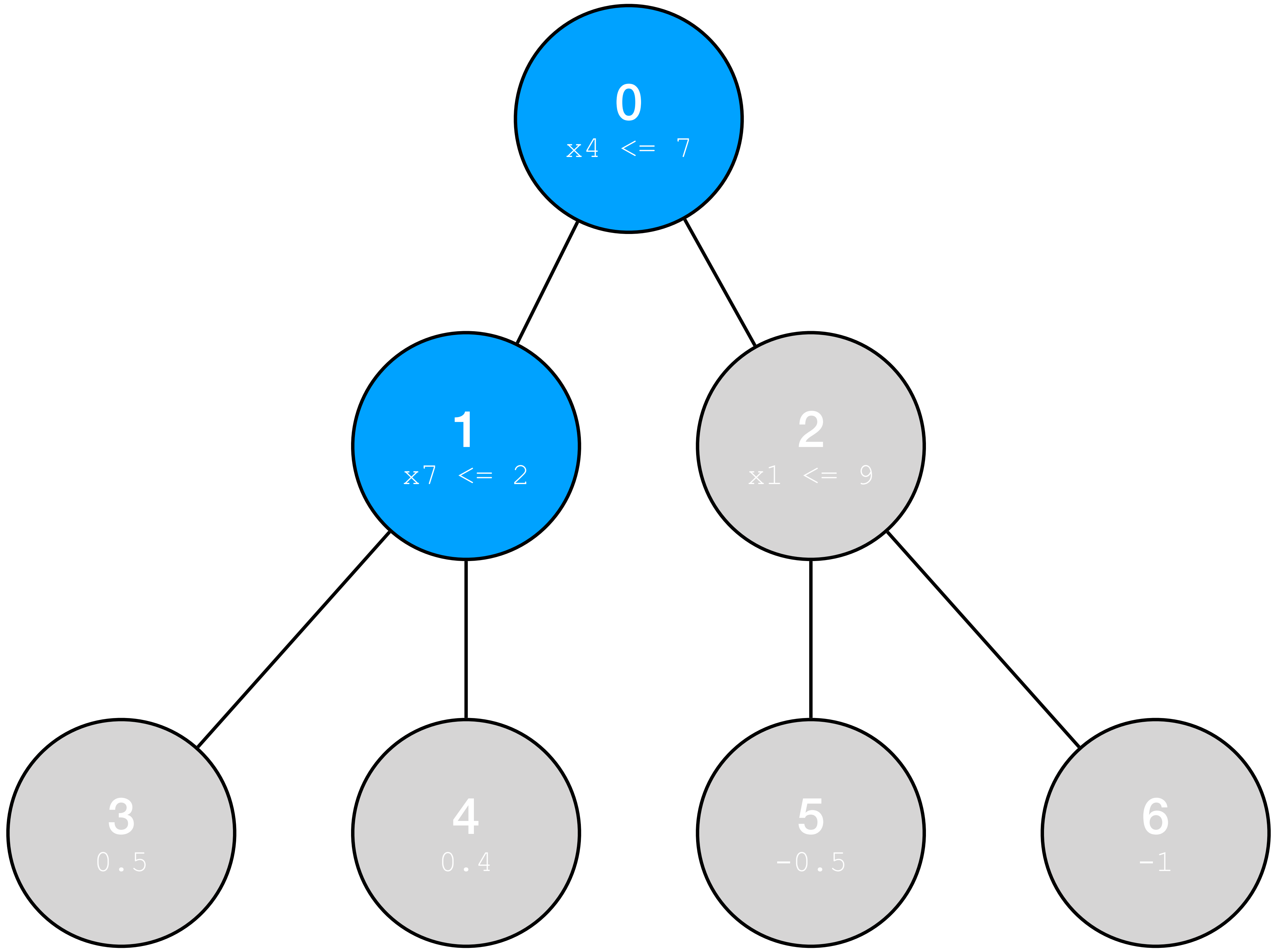
```
x = [ -, 12, -, -, 3, -, -, 5 ]
```



Decision Tree Inference

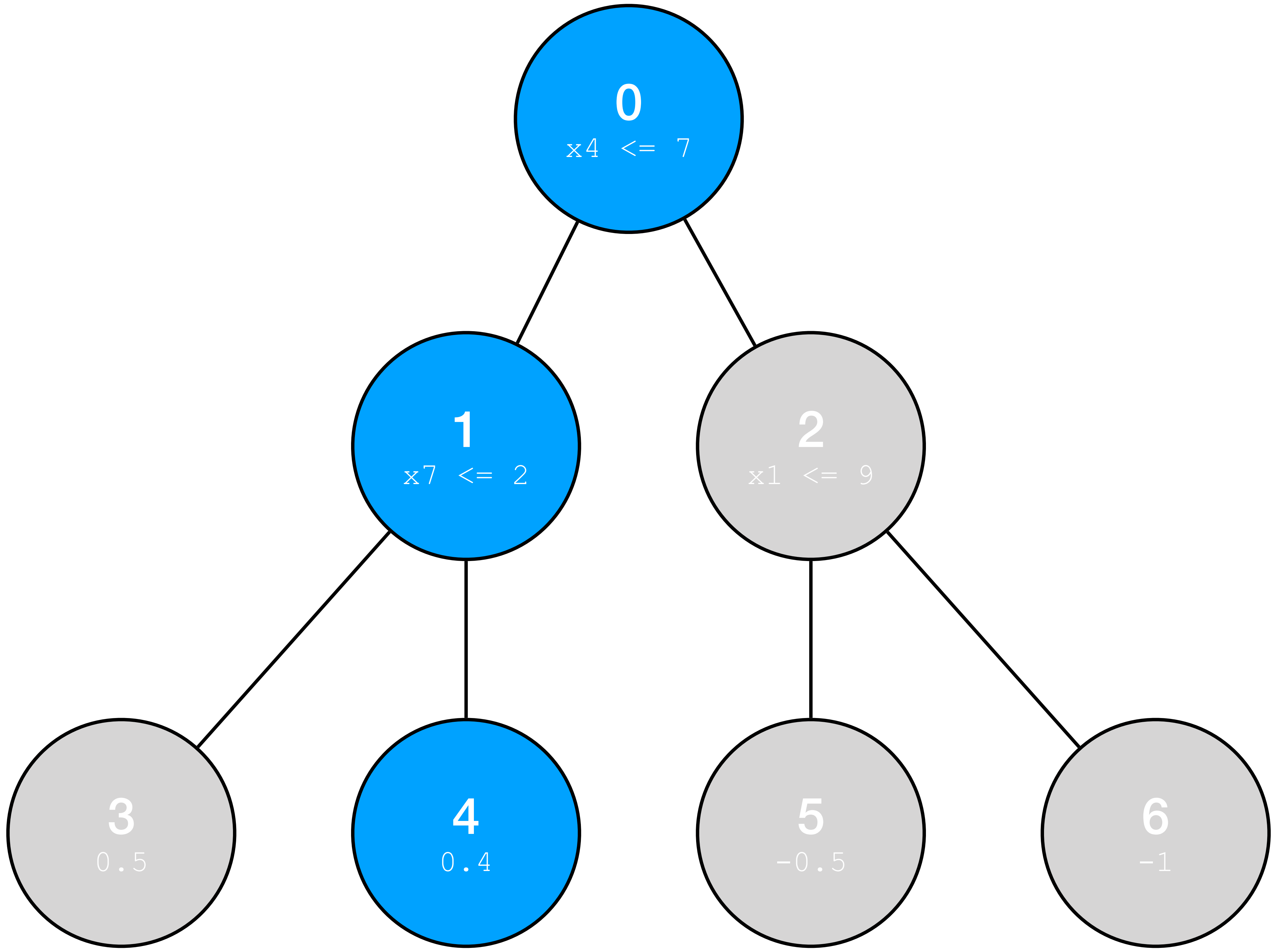
- Start at the root node - compare the selected feature with the threshold, go left or right depending on result
- Continue until reaching leaf - compare the selected feature with the threshold, go left or right depending on result

```
x = [ -, 12, -, -, 3, -, -, 5 ]
```



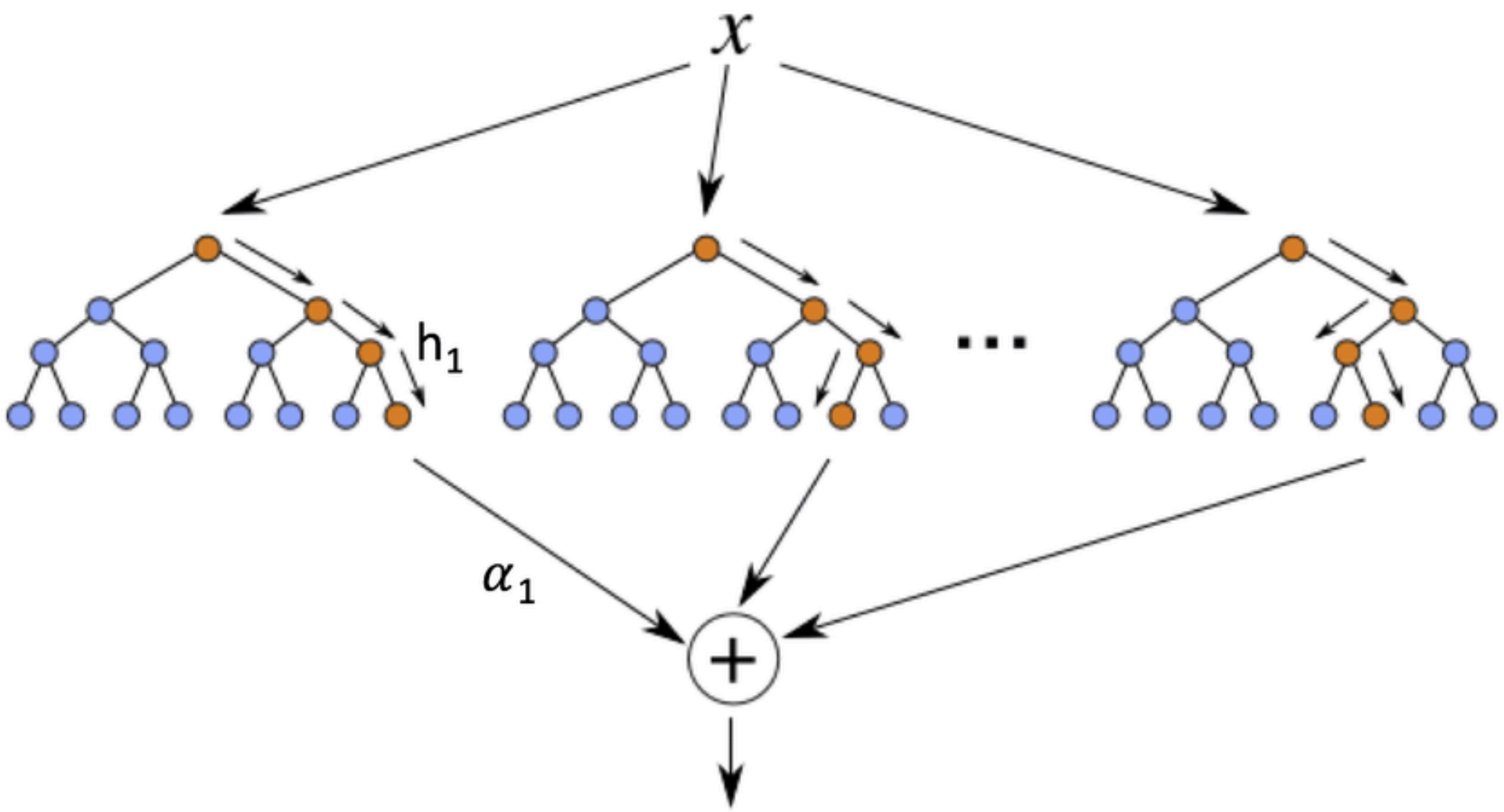
Decision Tree Inference

- Start at the root node - compare the selected feature with the threshold, go left or right depending on result
- Continue until reaching leaf - compare the selected feature with the threshold, go left or right depending on result
- The value of the terminal leaf is the tree prediction



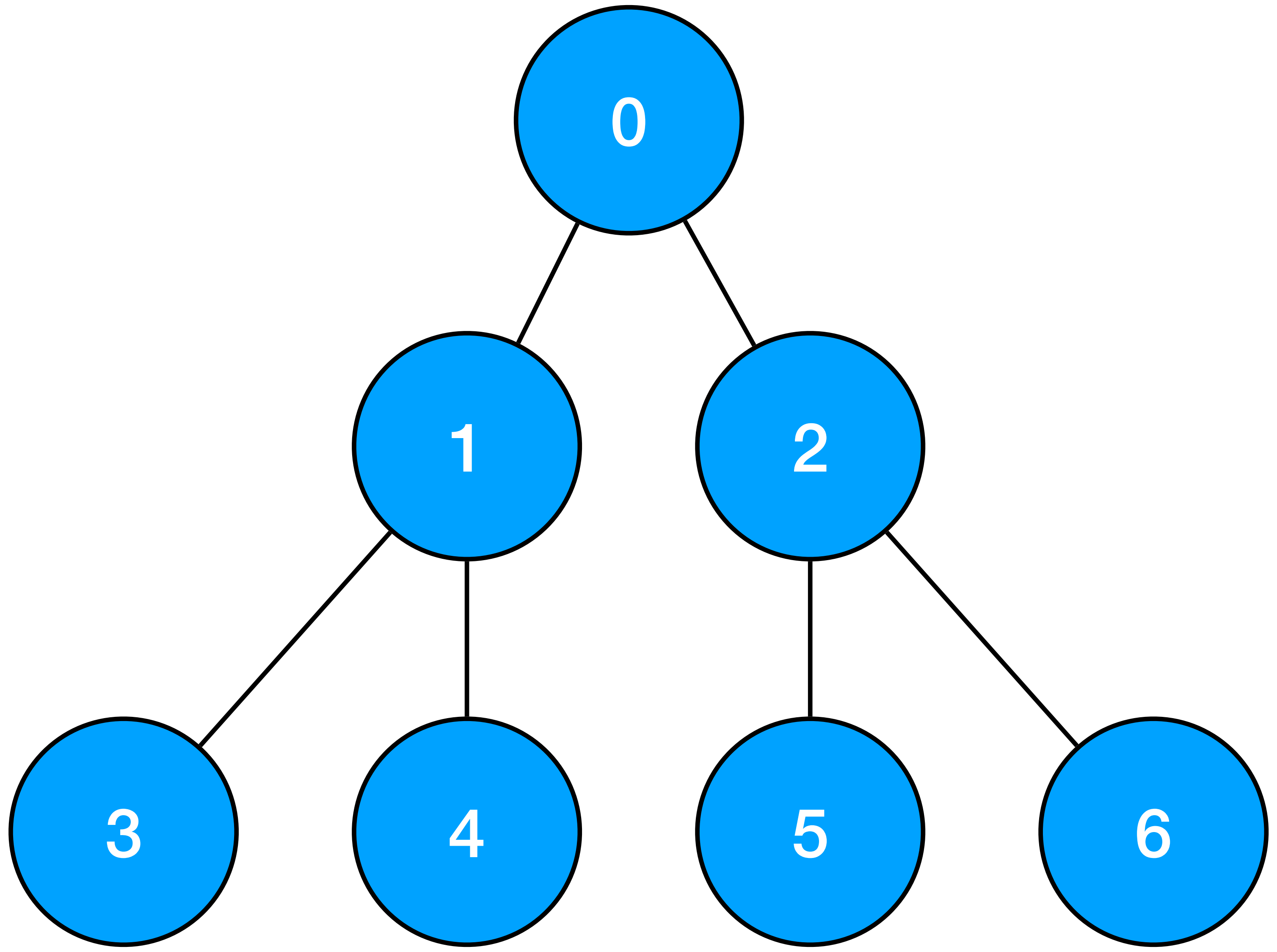
Decision Forest Inference

- Repeat the same procedure for every tree in the ensemble, sum up the tree scores for the BDT prediction
- Apply the inverse of the training loss function to obtain class probabilities



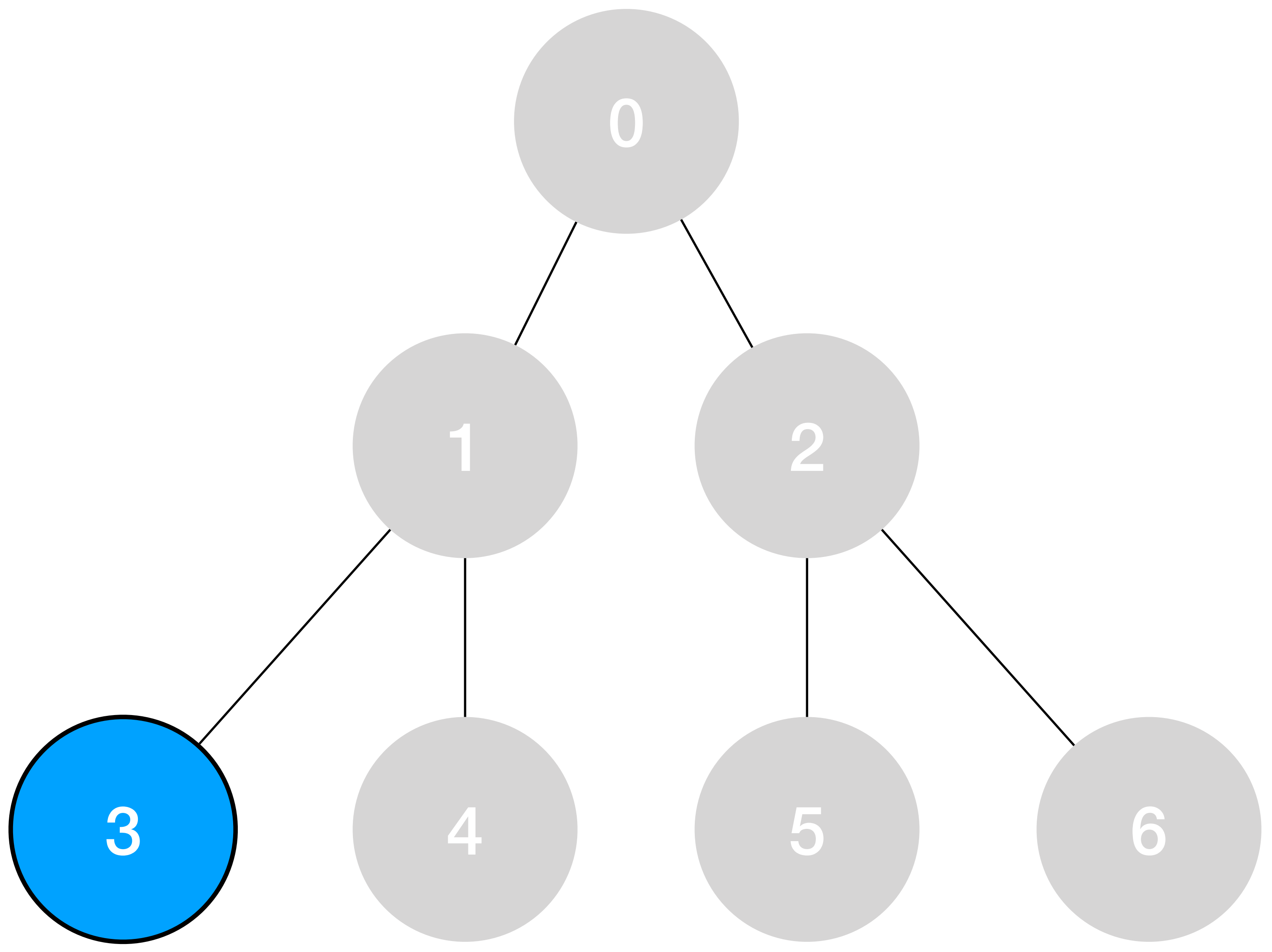
Conifer Implementation

- For a tree: find which leaf is reached given a data sample x
- ‘Invert’ the problem: for each node ask “does the decision path reach this node?” starting at the leaves



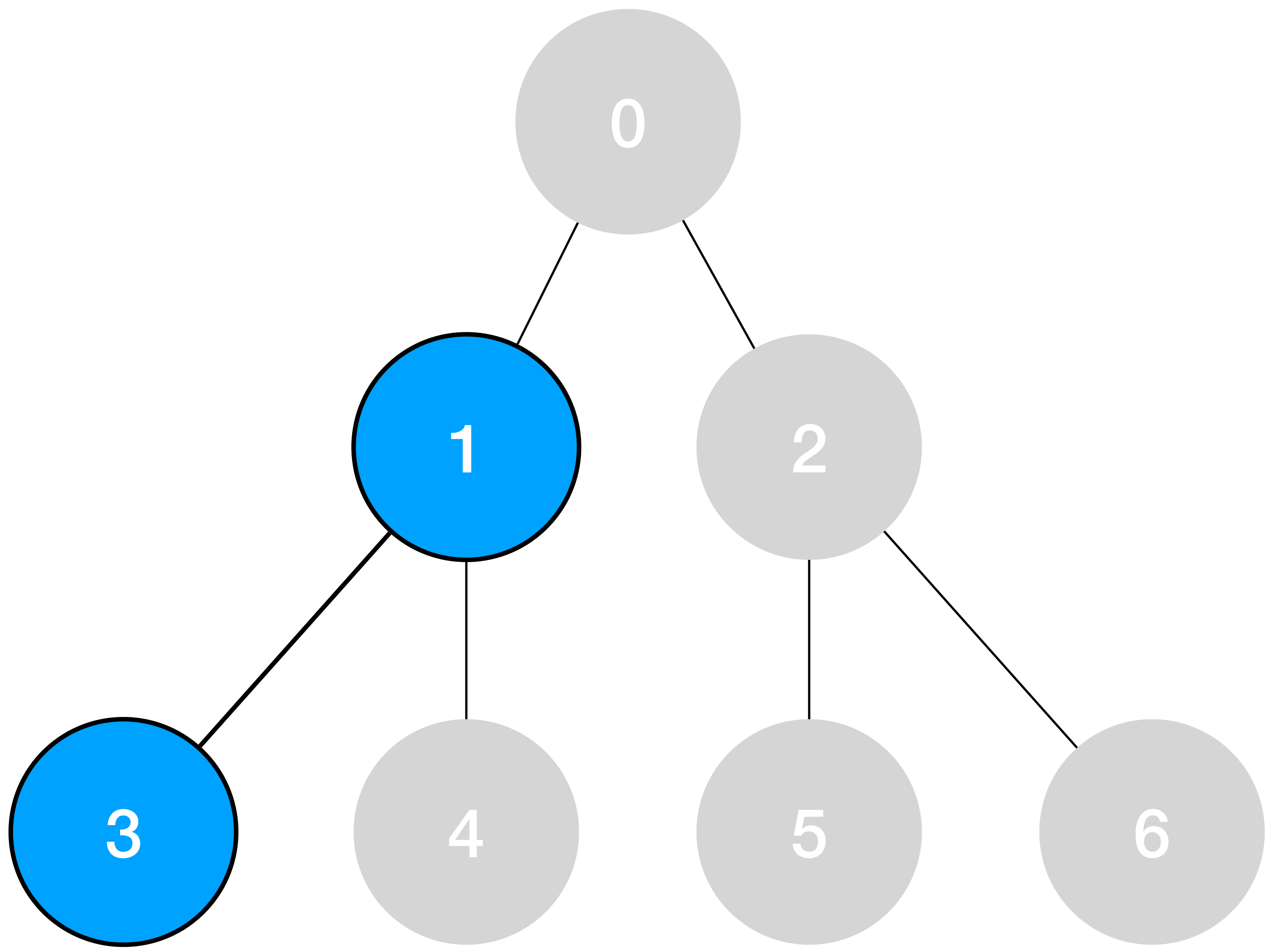
Conifer Implementation

- For a tree: find which leaf is reached given a data sample x
- ‘Invert’ the problem: for each node ask “does the decision path reach this node?” starting at the leaves
- For leaf node ‘3’:
  - The decision path reaches ‘3’ if: the decision path reached ‘1’ AND the comparison at ‘1’ goes ‘left’



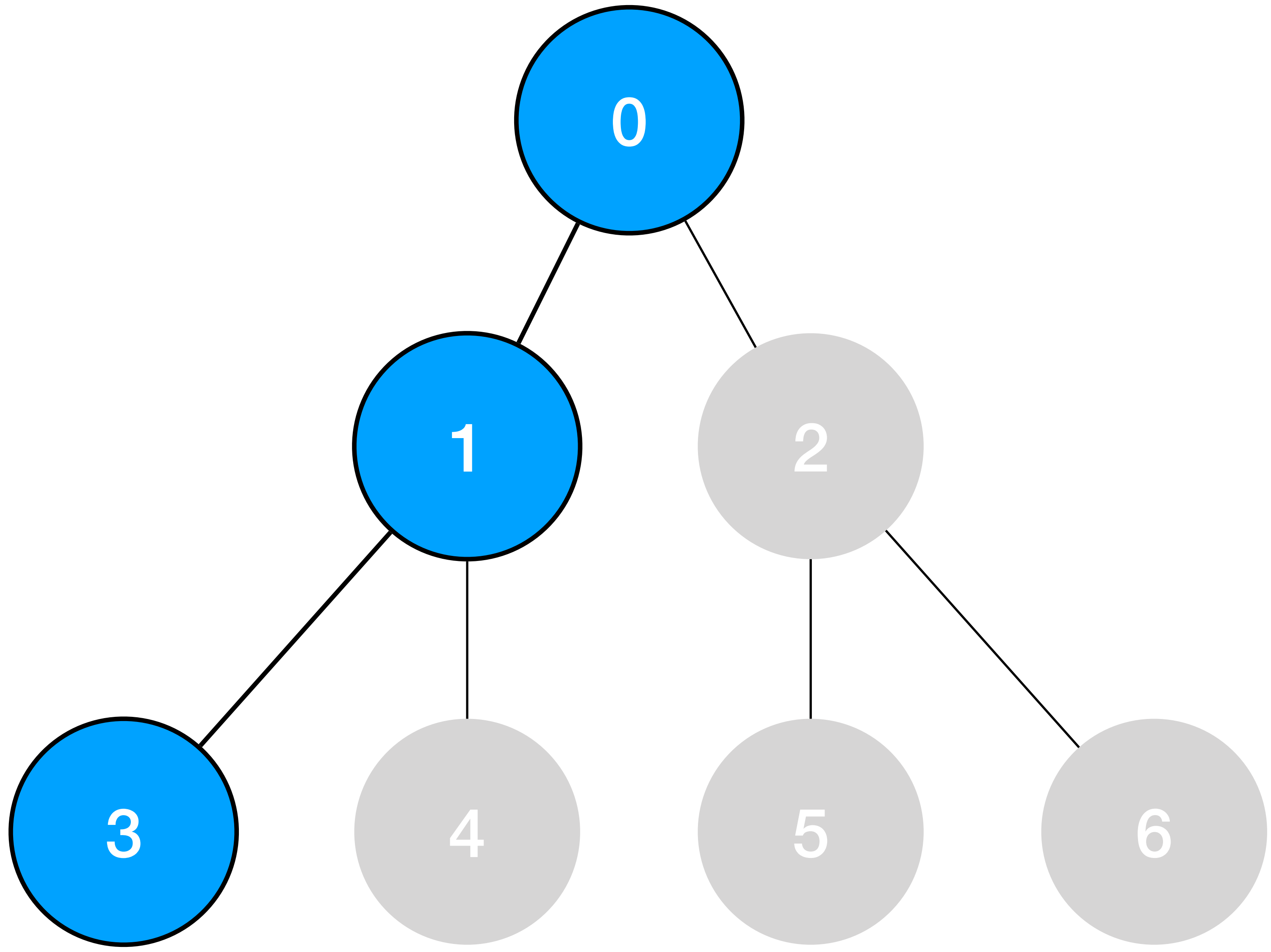
Conifer Implementation

- For a tree: find which leaf is reached given a data sample x
- ‘Invert’ the problem: for each node ask “does the decision path reach this node?” starting at the leaves
- For leaf node ‘3’:
  - The decision path reaches ‘3’ if: the decision path reached ‘1’ AND the comparison at ‘1’ goes ‘left’
- For node ‘1’:
  - The decision path reaches ‘1’ if: the decision path reached ‘0’ AND the comparison at ‘0’ goes ‘left’



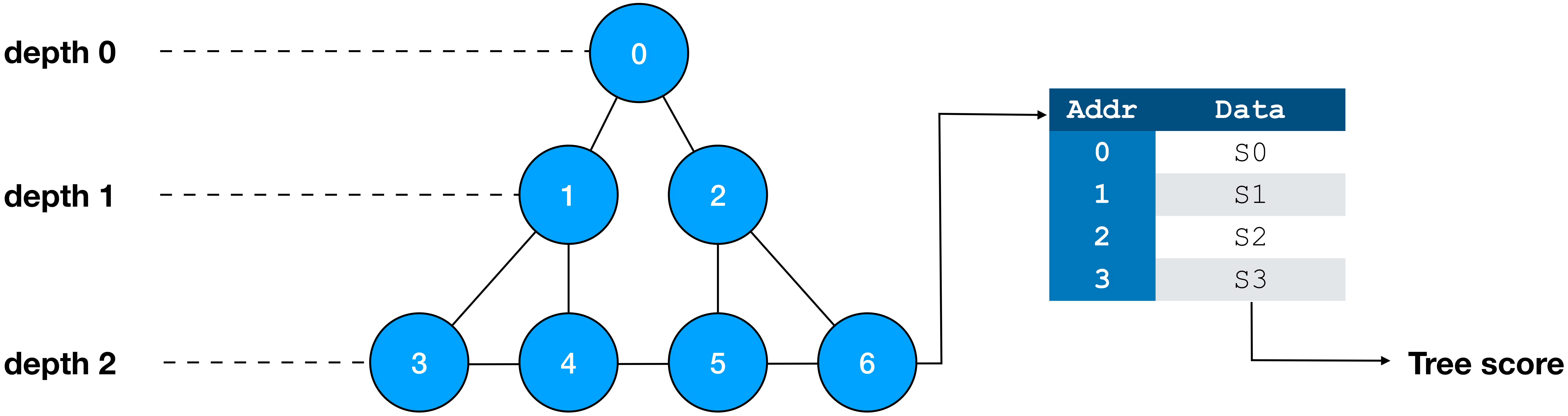
Conifer Implementation

- For a tree: find which leaf is reached given a data sample x
- ‘Invert’ the problem: for each node ask “does the decision path reach this node?” starting at the leaves
- For leaf node ‘3’:
  - The decision path reaches ‘3’ if: the decision path reached ‘1’ AND the comparison at ‘1’ goes ‘left’
- For node ‘1’:
  - The decision path reaches ‘1’ if: the decision path reached ‘0’ AND the comparison at ‘0’ goes ‘left’
- For node ‘0’:
  - The decision path always passes through the root node

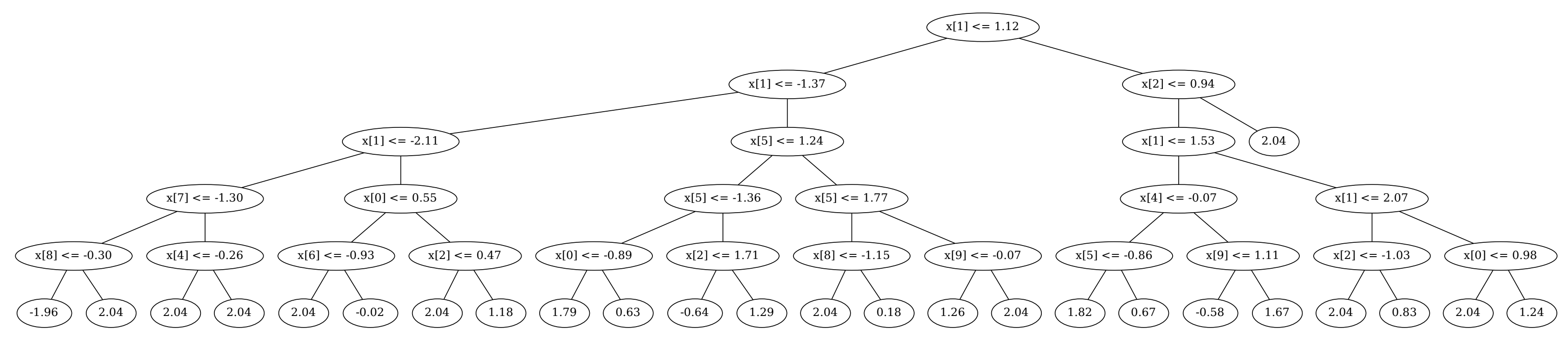


Conifer Implementation

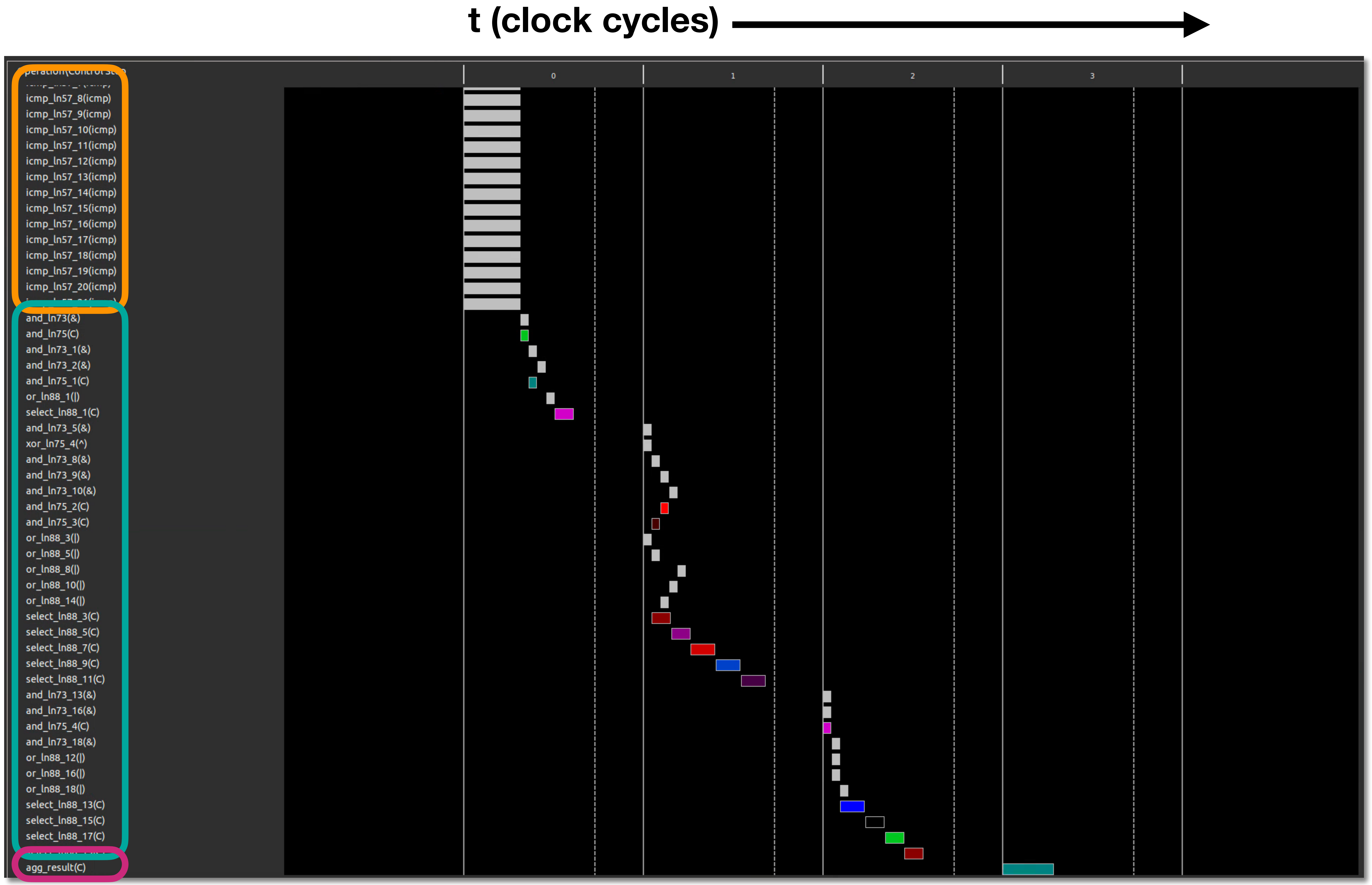
- For a tree: find which leaf is reached given a data sample x
- ‘Invert’ the problem: for each node ask “does the decision path reach this node?” starting at the leaves
- We can **parallelise** this over paths by brute force: evaluate all nodes at the same depth simultaneously
- We can **pipeline** this over different data: each node can do a comparison on new data with II=1
- For each leaf node we have a boolean: TRUE if the decision path reaches leaf, otherwise FALSE
- Concatenate the boolean for each leaf node → select the value corresponding to the leaf



Scheduling - Tree



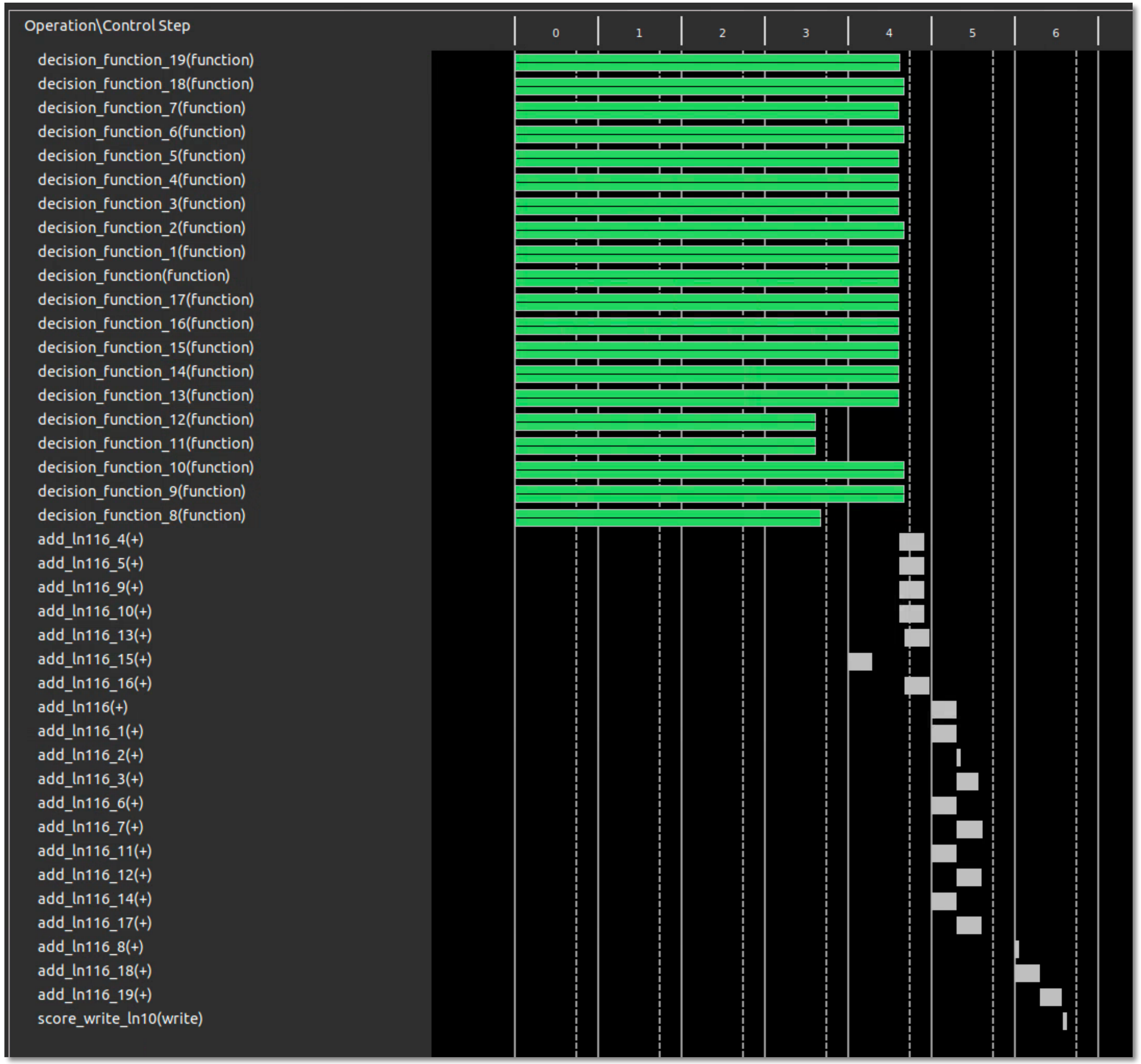
- Did we achieve what we described?
- Vitis HLS Schedule Viewer in GUI
    - Tree depth = 5, some sparsity
- All **comparisons** in parallel at the start
- Cascade of **boolean operations**
    - AND, OR, XOR, NOT
- ‘**Aggregate**’ at end



Scheduling - Forest

- Did we achieve what we described?
- Vitis HLS Schedule Viewer in GUI
  - Number of trees = 20
  - Tree from previous slides is one of them
- All tree inferences performed in parallel
- Tree scores summed in pairs
- Total latency: 7 clock cycles

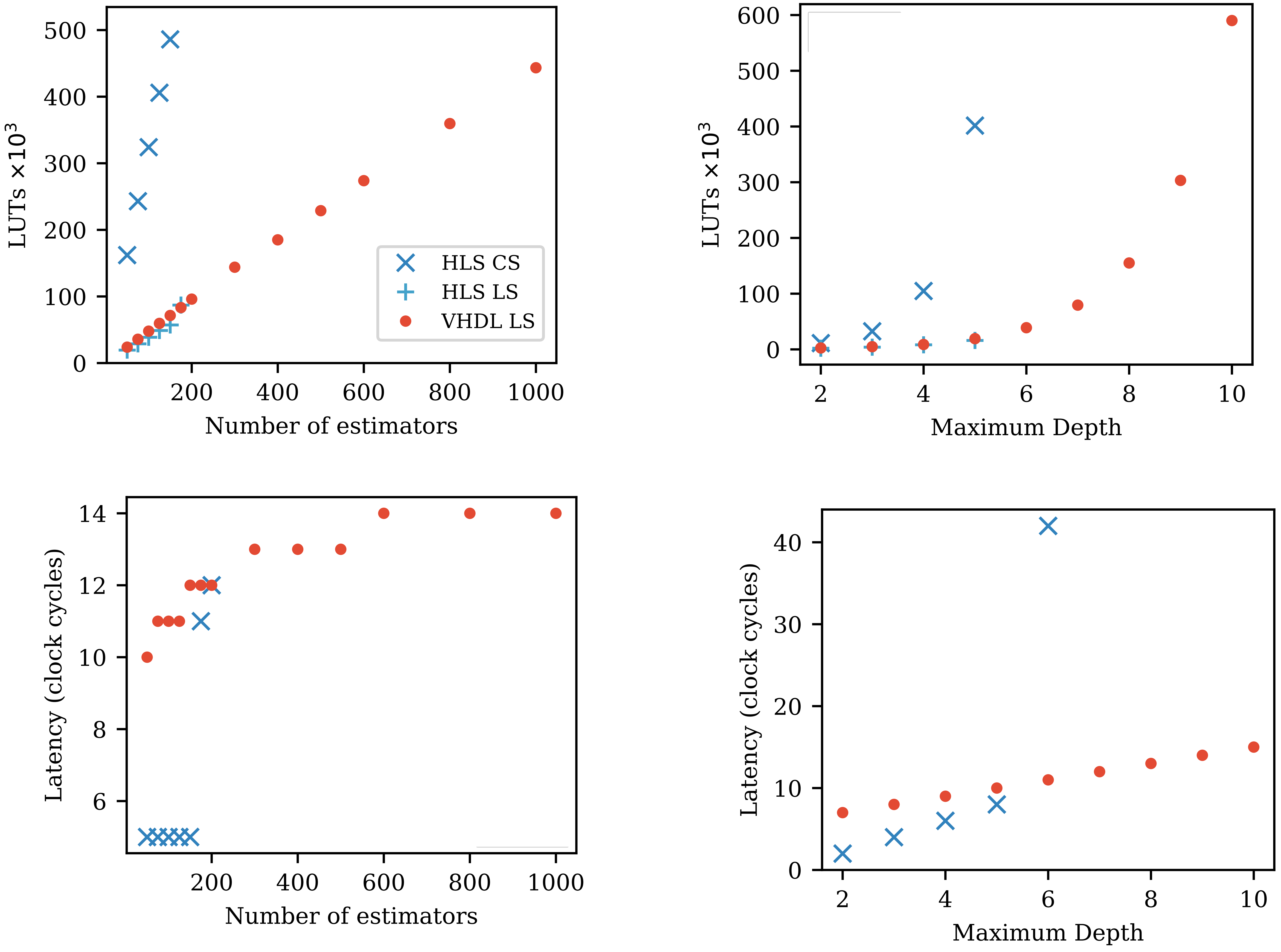
t (clock cycles)



Implementations

- Conifer has both HLS and VHDL implementations - both targeting the same architecture previously described and fully pipelined
- Within some limits the HLS achieves identical resources to the VHDL
  - After synthesizing the HLS-generated HDL
  - Caveat: plots are with Vivado HLS 2019.2. With recent Vitis HLS the performance is better
- The HLS latency can be lower than the VHDL
  - VHDL pipelining was done 'by hand'
- Resources and latency scales as expected:
  - Resource linear with trees, exponential with depth
  - Latency logarithmic with trees, linear with depth
- Latency within 10-100 ns is achievable

'CS' = C Synthesis
'LS' = Logic Synthesis

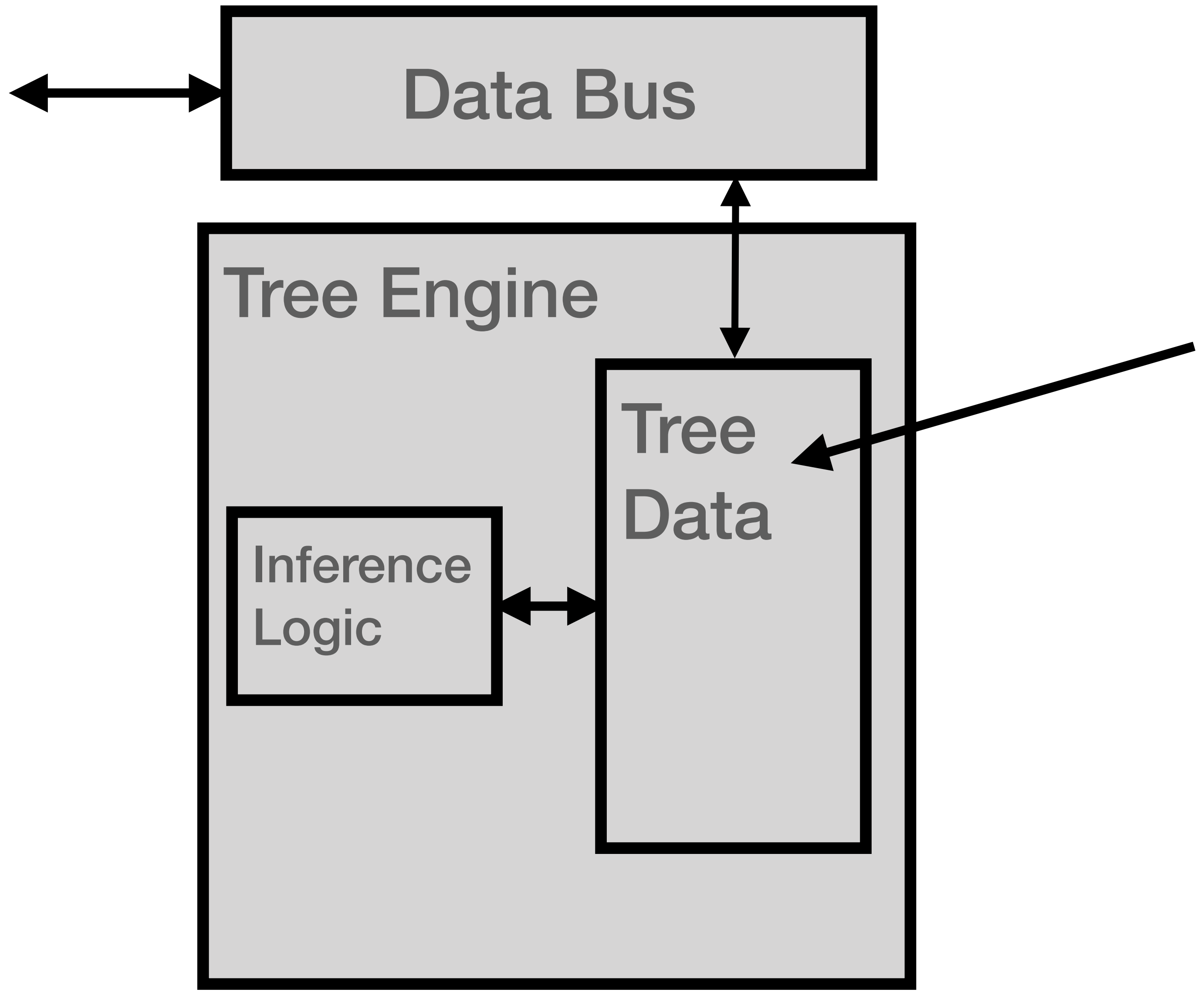


Forest Processing Unit

- So far we looked at ‘static’ BDT evaluation
  - One trained model ➝ one HLS function ➝ one IP ➝ one bitfile
  - So if the model changes at all, we need to rerun C Synthesis, Logic Synthesis and Implementation ➝ takes hours!
- In next section we will look at a more dynamic & reconfigurable implementation called “Forest Processing Unit” (FPU)
- We would like a base design that can perform inference of ~any BDT model afterwards (within some limits)
- And we would like to take advantage of the FPGA to get good performance (fast inference)
- **Idea 1**: represent the BDT as data, operate inference on that data, and load new data for a new model
- **Idea 2**: parallelise over trees by having independent ‘Tree Engines’, aggregate their output for the model

FPU Design

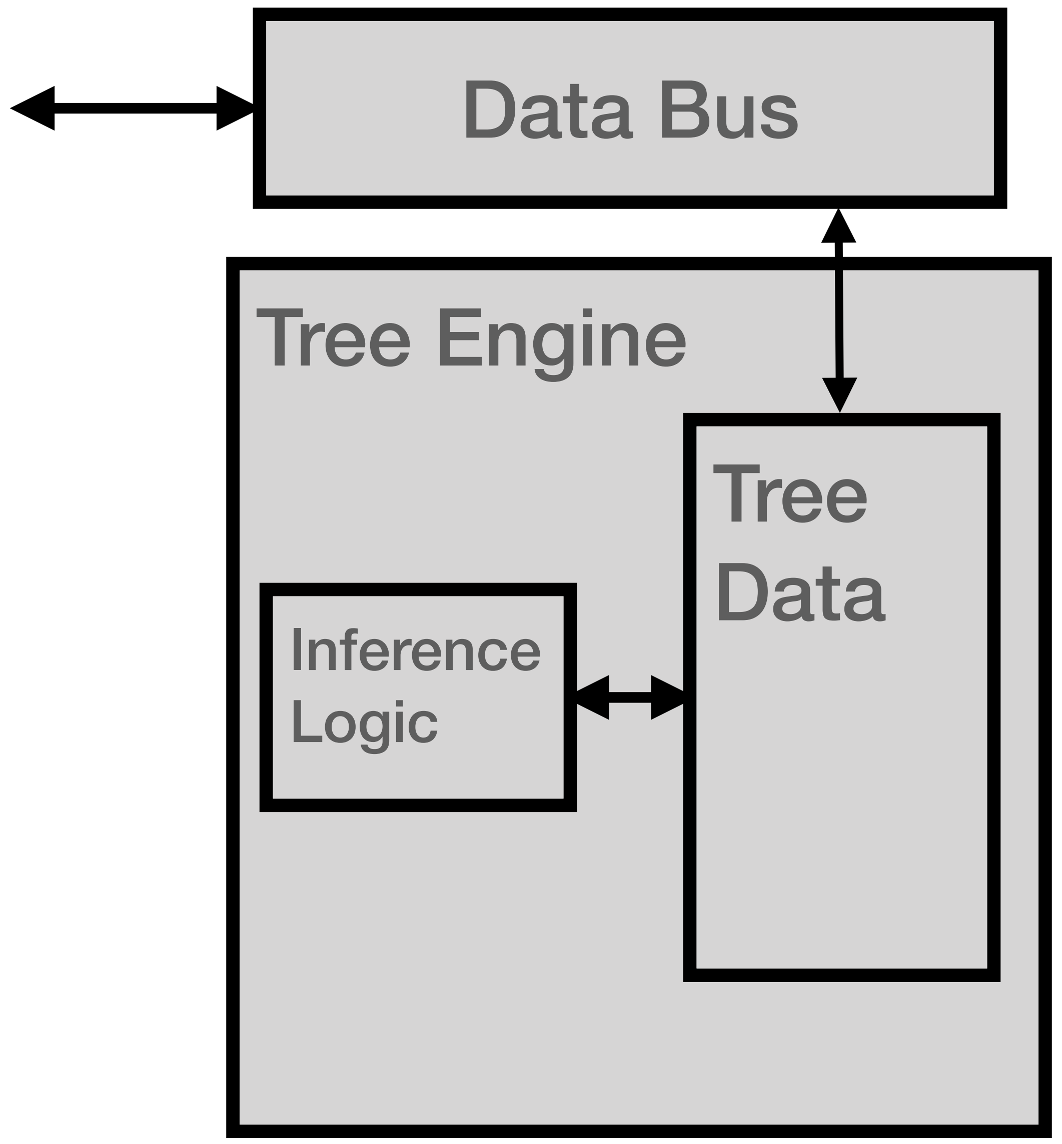
- **Idea 1**: represent the BDT as data, operate inference on that data, and load new data for a new model over a bus
- Store one node at one address, child indices are pointers to other addresses




```
Tree:
      index          : [ 0,  1,  2,   3,   4,    5,  6]
      children_left  : [ 1,  3,  5,  -2,  -2,   -2, -2]
      children_right : [ 2,  4,  6,  -2,  -2,   -2, -2]
      parent         : [ -1, 0,  0,   1,   1,    2,  2]
      feature        : [ 4,  7,  1,  -2,  -2,   -2, -2]
      threshold      : [ 7,  2,  9,  -2,  -2,   -2, -2]
      value          : [-1, -1, -1, 0.5, 0.4, -0.5, -1]
```

FPU Design

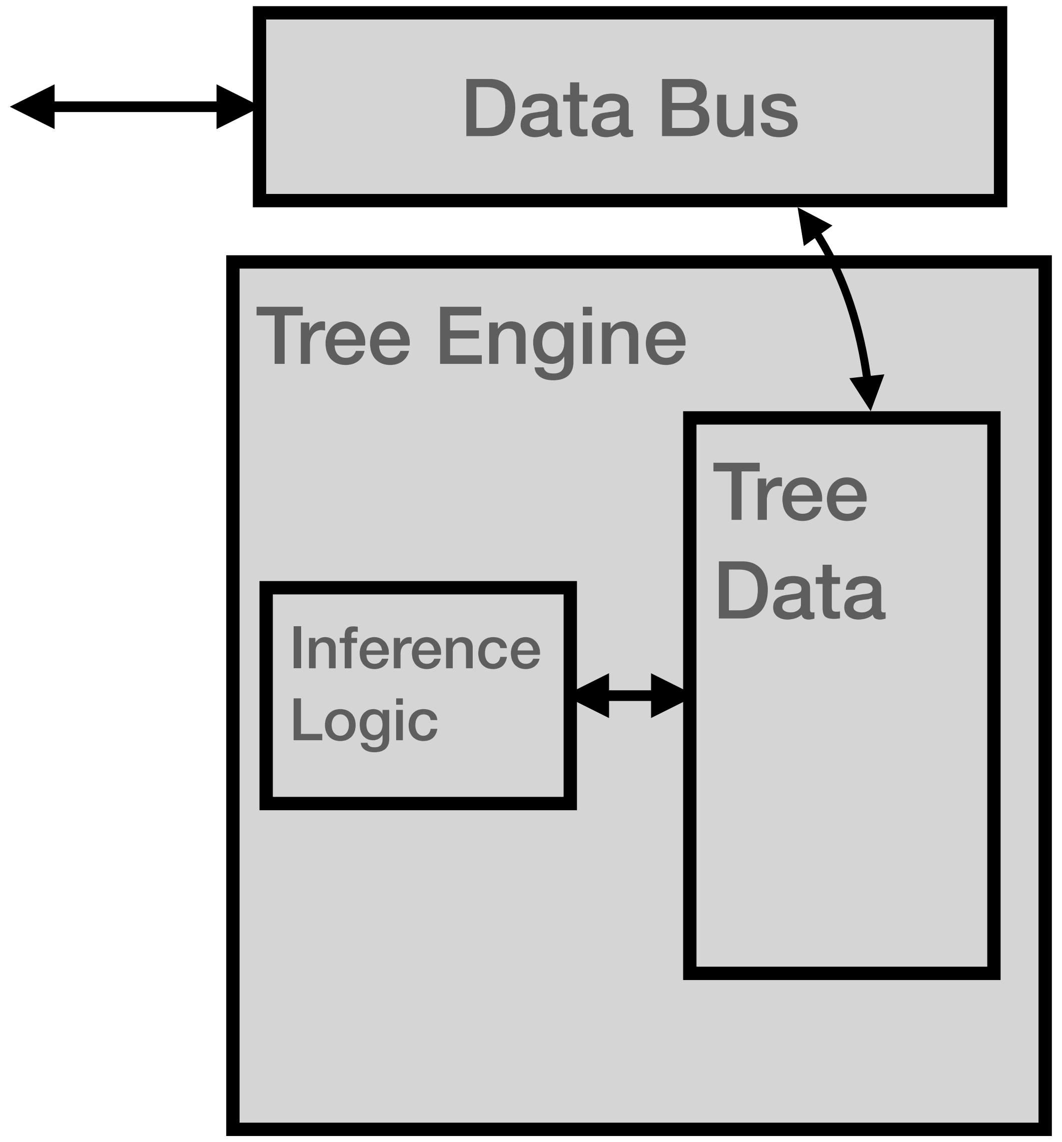
- **Idea 1**: represent the BDT as data, operate inference on that data, and load new data for a new model
- To perform inference of a model on some data we need to:
  - Read the next node
  - Compare the appropriate feature with the threshold
  - Get the pointer to the next node
- Upon reaching a leaf, return its score




```
void TreeEngine(T X[NVARS], DecisionNode nodes[NNODES], U& y){
  #pragma HLS pipeline
  ap_int<ADDRBITS> i = 0;
  auto node = nodes[i];
  node_loop : while(!node.is_leaf){
    #pragma HLS pipeline
    i = X[node.feature] <= node.threshold ?
                                node.child_left : node.child_right;
    node = nodes[i];
  }
  y = node.score;
}
```

FPU Design

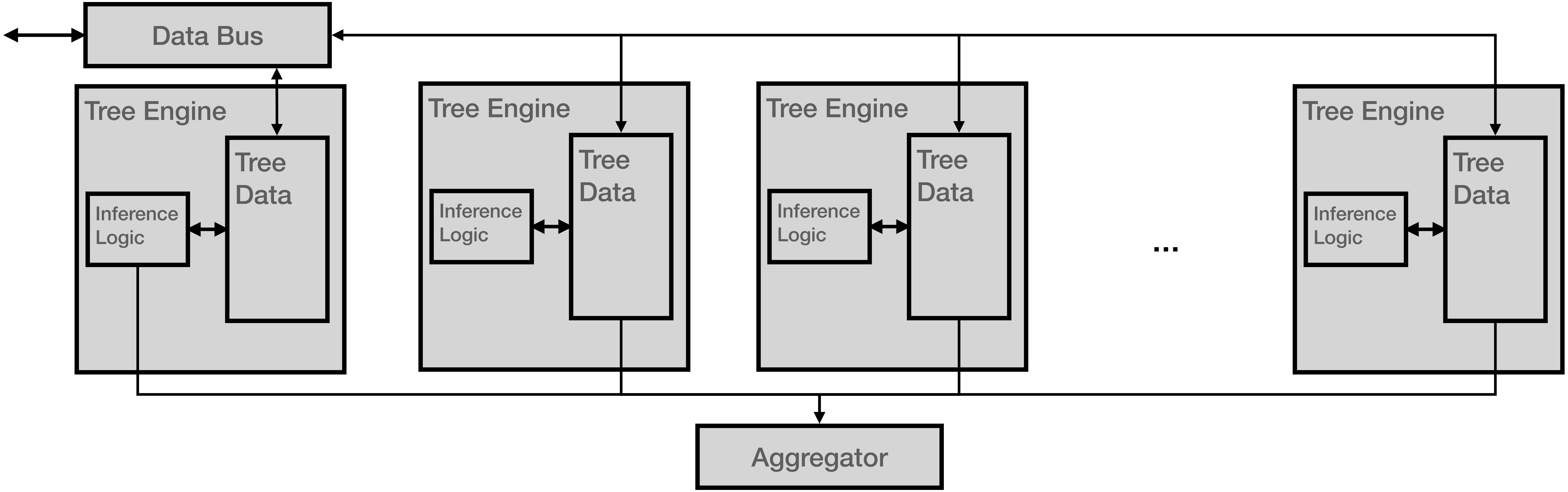
- **Idea 1**: represent the BDT as data, operate inference on that data, and load new data for a new model
- To perform inference of a model on some data we need to:
  - Read the next node
  - Compare the appropriate feature with the threshold
  - Get the pointer to the next node
- Iteration logic has a ‘loop carried dependency’ between iterations, and a data dependent latency



Loop Name	Latency (cycles) min	Latency (cycles) max	Iteration Latency	Initiation Interval achieved	Initiation Interval target	Trip Count	Pipelined
- node_loop	?	?	3	3	1	?	yes

FPU Design

- **Idea 2**: parallelise over trees by having independent 'Tree Engines', aggregate their output for the model
- Put as many Tree Engines as will fit and achieve timing closure in the FPGA
- Number of Tree Engines will constrain the model size that fits



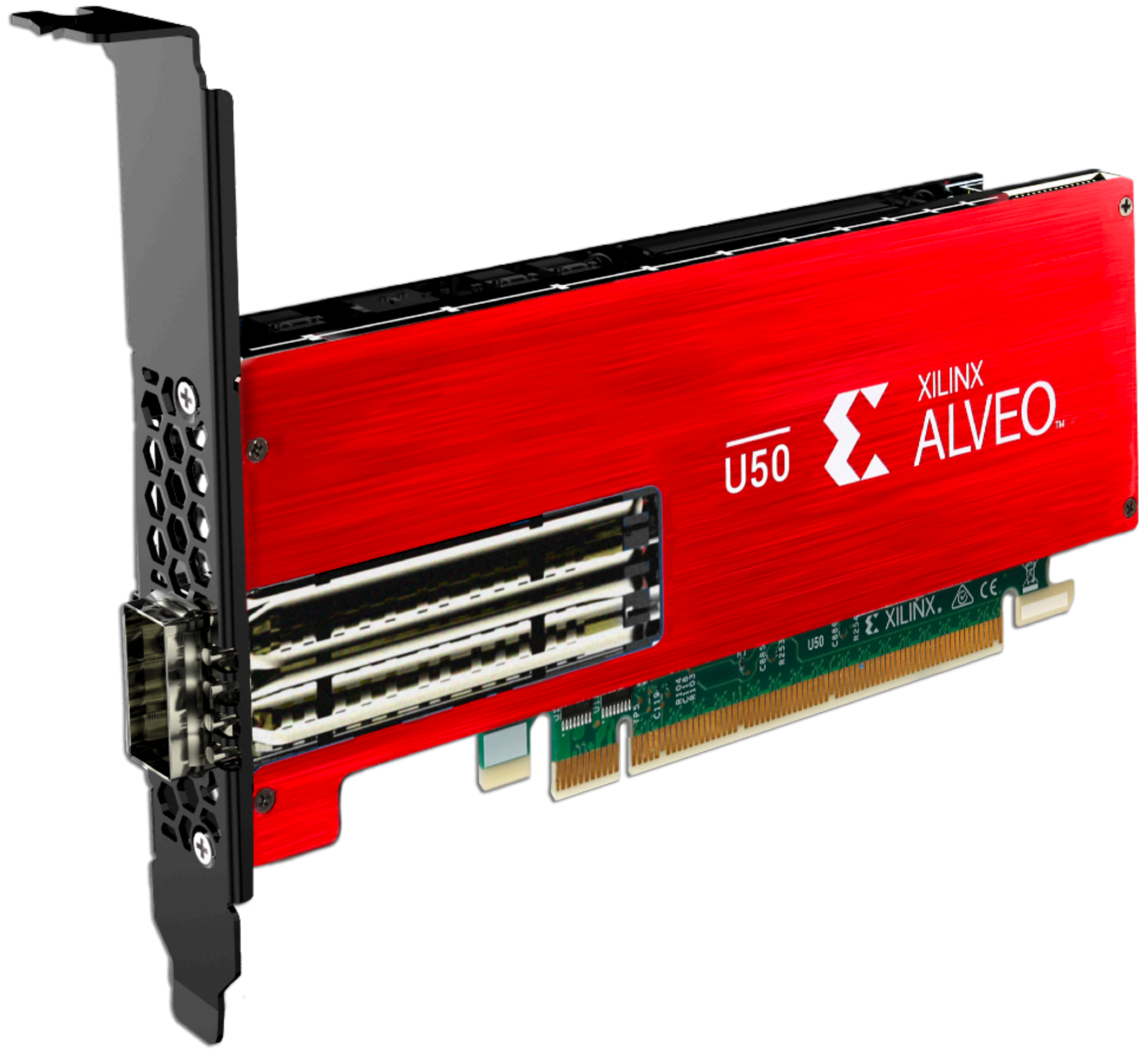
FPU System Design

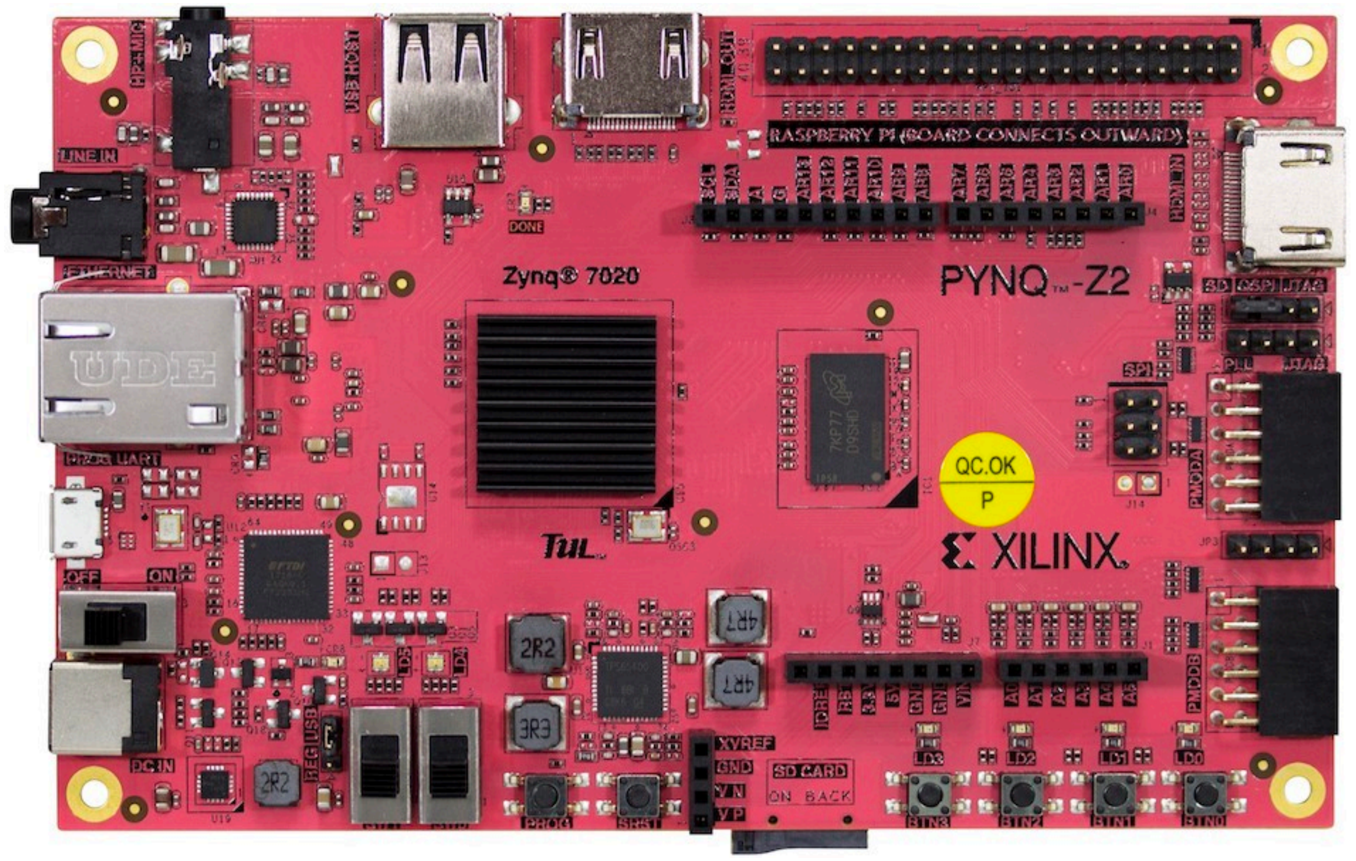
- Putting it together
  - One function that has arguments for both BDT-data and inference-data, and an ‘instruction’ parameter for what to do
- Define the node memories as static to keep the data in between function calls
  - Load nodes once, perform inference later whenever (multiple times)
  - Later load new nodes for a different model..
- This code is a simplified view of that:

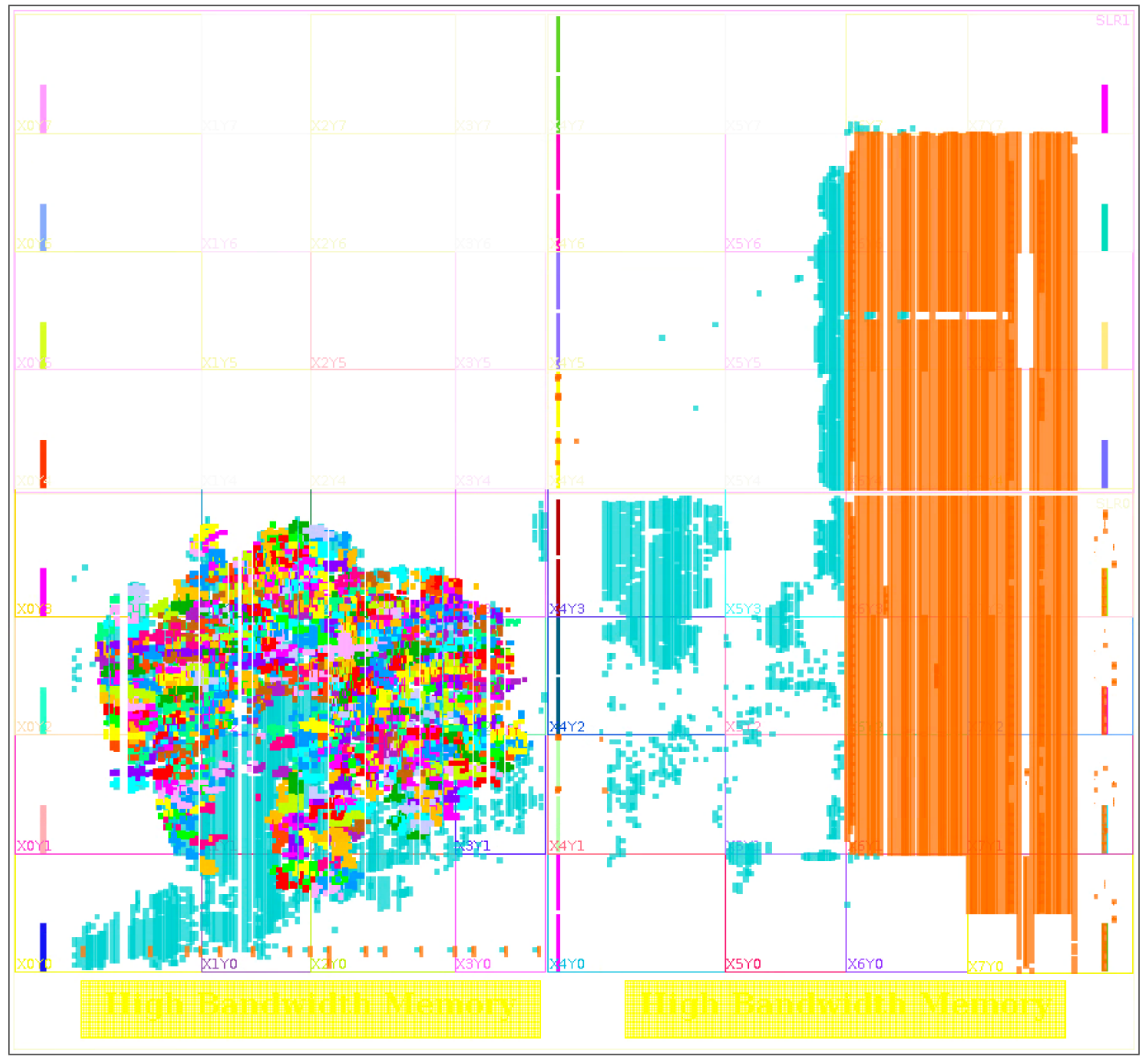
```
void fpu_top_level(int* X, int* y, int instruction, DecisionNode* nodes){
  #pragma interface …
  static DecisionNode nodes_internal[NTE][NNODES];
  #pragma HLS array_partition variable=nodes_int dim=1
  if(instruction == 0){
    load_nodes(nodes, nodes_internal);
  }
  if(instruction == 1){
    decision_function(X, y);
  }
}
```

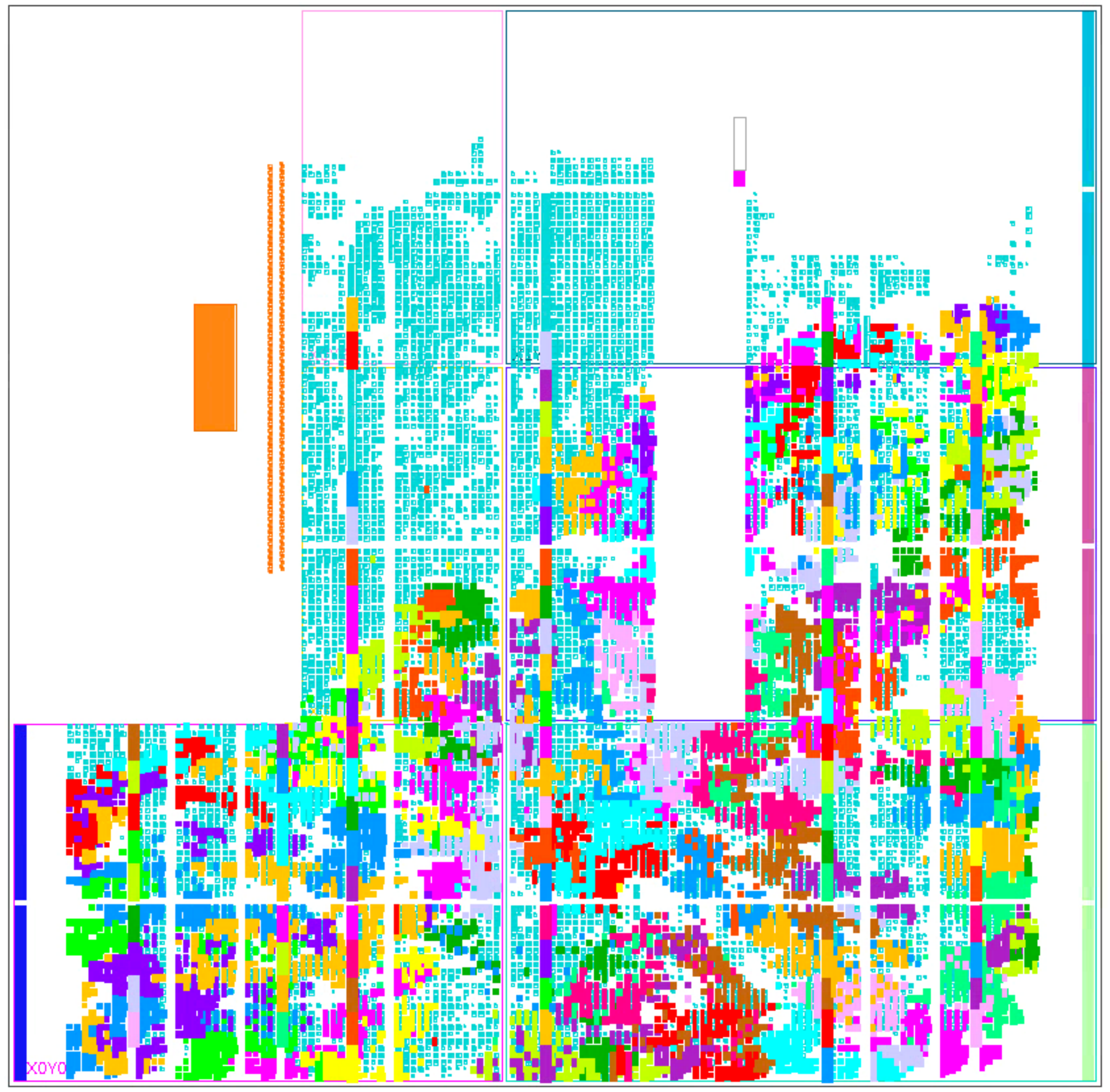
FPU Floorplan

- FPU with 200 Tree Engines in Alveo U50 (top) and 100 Tree Engines in pynq-z2 (bottom)
  - Each TE is highlighted in colour (with a repeating cycle)
- BRAMs for nodes are in columns
- Logic near BRAMs is TE inference logic
- AXI Interfaces used for data bus
  - Both for loading models and inference data
- Whole design is written with HLS
  - HLS as a productivity tool









Conclusions

- **conifer** is a tool to map Decision Forests onto FPGA firmware
  - `pip install conifer`
- In this talk we discussed:
  - Some applications: low latency triggering, embedded frontend
  - Conifer implementation and approach to executing branched prediction
  - HLS and VHDL performance
  - Forest Processing Unit: reconfigurable Decision Forest inference architecture designed with HLS

Backup

HLS Code 1 / 3

- Perform all the comparisons simultaneously: `unroll` the loop
- Store `boolean` results in a fully-partitioned array “`comparison`”

```
// Execute all comparisons
Compare: for(int i = 0; i < n_nodes; i++){
  #pragma HLS unroll
  // Only non-leaf nodes do comparisons
  // negative values mean is a leaf (sklearn: -2)
  if(feature[i] >= 0){
    comparison[i] = x[feature[i]] <= threshold[i];
  }else{
    comparison[i] = true;
  }
}
```

HLS Code 2 / 3

- Compute the node `activation` (`true` if decision path traverses node, otherwise `false`)

```
// Determine node activity for all nodes
int iLeaf = 0;
Activate: for(int i = 0; i < n_nodes; i++){
  #pragma HLS unroll
  // Root node is always active
  if(i == 0){
    activation[i] = true;
  }else{
    // If this node is the left child of its parent
    if(i == children_left[parent[i]]){
      activation[i] = comparison[parent[i]] && activation[parent[i]];
    }else{ // Else it is the right child
      activation[i] = !comparison[parent[i]] && activation[parent[i]];
    }
  }
  // Skim off the leaves
  if(children_left[i] == -1){ // is a leaf
    activation_leaf[iLeaf] = activation[i];
    value_leaf[iLeaf] = value[i];
    iLeaf++;
  }
}
```

HLS Code 3 / 3

- Compute the node `activation` (`true` if decision path traverses node, otherwise `false`)

```
for(int i = 0; i < n_leaves; i++){
    if(activation_leaf[i]){
      return value_leaf[i];
    }
  }
```

VHDL

- To the right is the VHDL version of the tree traversal is shown in HLS on the previous slides
- The main difference is that we have to do the scheduling of operations to clock cycles ourselves in VHDL
  - The latency of this section of code depends on the maximum depth of the tree
  - This VHDL is “over pipelined” compared to the HLS

```
activation(0) <= true; -- the root node is always active
GenAct:
for i in 1 to nNodes-1 generate
  LeftChild:
  if i = iChildLeft(iParent(i)) generate
    process(clk)
    begin
      if rising_edge(clk) then
        activation(i) <= comparisonPipe(depth(i))(iParent(i))
                         and activation(iParent(i));
      end if;
    end process;
  end generate LeftChild;
  RightChild:
  if i = iChildRight(iParent(i)) generate
    process(clk)
    begin
      if rising_edge(clk) then
        activation(i) <= (not comparisonPipe(depth(i))(iParent(i)))
                          and activation(iParent(i));
      end if;
    end process;
  end generate RightChild;
end generate GenAct;
```