A Hardware Architecture of Reconfigurable Decision Tree Model

> Wang Hanyu @ 2021.01.07

Summary

In this study, we design a light-weight reconfigurable decision tree structure based on **Von Neumann Architecture**. We set:

1. Truth table model
2. MUX Tree model

as our benchmarks, and our model has much **smaller area** comparing to truth table model and much more **flexibility** comparing to MUX Tree model. This report is organized as follows:

Part 1: Problems and Need

1. K(i)L-LUT

Machine Learning methods could improve hardware's performance in many application. However, the area of the circuit is always the main concern. The complexity of the model brings a huge circuit which is not able to fit into target clock cycle, unrealistic to be deployed on the chip, or brings too much heat consumption. Therefore, a light-weighted architecture to store a trained model is crucial.

The problem could be formulated as: design a hardware structure that is able to store a given map from inputs to outputs. We denote the parameters as K(i)L:

- **K** : number of inputs
- **L** : number of outputs
- **i**: degree of freedom

Note that the variable **i** represents the degree of freedom, which means the minimum number of variable to represents the function. For example, a constant function has **i** = 1 and a step function has **i** = 3 (1 for threshold value, and 2 for the output below and above).

As a summary, the problem is to implement a reconfigurable K(i)L -LUT with area as small as possible.

Part 2: Benchmarks

1. Truth Table Model

Truth table is a simple implementation of a look-up-table. For the `K(i)L` model, the truth table use that K-bit input as the index and stores all the corresponding output in that table, and each entry is an L-bit data. The trained model is stored into the memory and the whole truth table is refreshed. After that, given a set of input values, the truth table is able to return the output value by look up the data using input value as index.

The advantage of this implementation is the flexibility and the delay. First, it is flexible because all of the Boolean function can be expressed by its truth table. Hence, all models can be stored in this structure. Second, since the output is derived by address interpretation, which has been optimized by most of the libs. The customized RAM could run the read and write operations faster.

However, the flaw of this implementation is the circuit area. Note that the truth table could be implemented using a customized RAM or DFF as register arrays. For both implementation, the cost of memory could be huge as it grows exponentially when the input size increase.

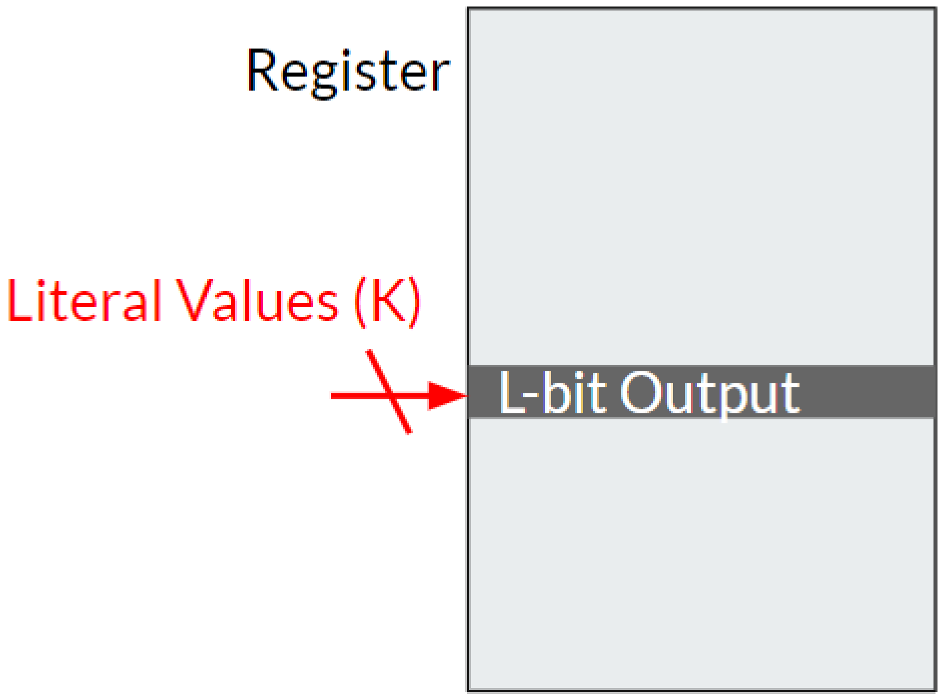


Figure above shows the register, it has L bits per entry and 2^k entries.

Verilog HDL Code

```
module ram(
    input                   clk_i,
    input                   rst_i,
    input                   wr_en_i,
    input                   rd_en_i,
    input [14:0]            addr_i,
    input [6:0]             data_i,
    output reg [6:0]        data_io
);
```

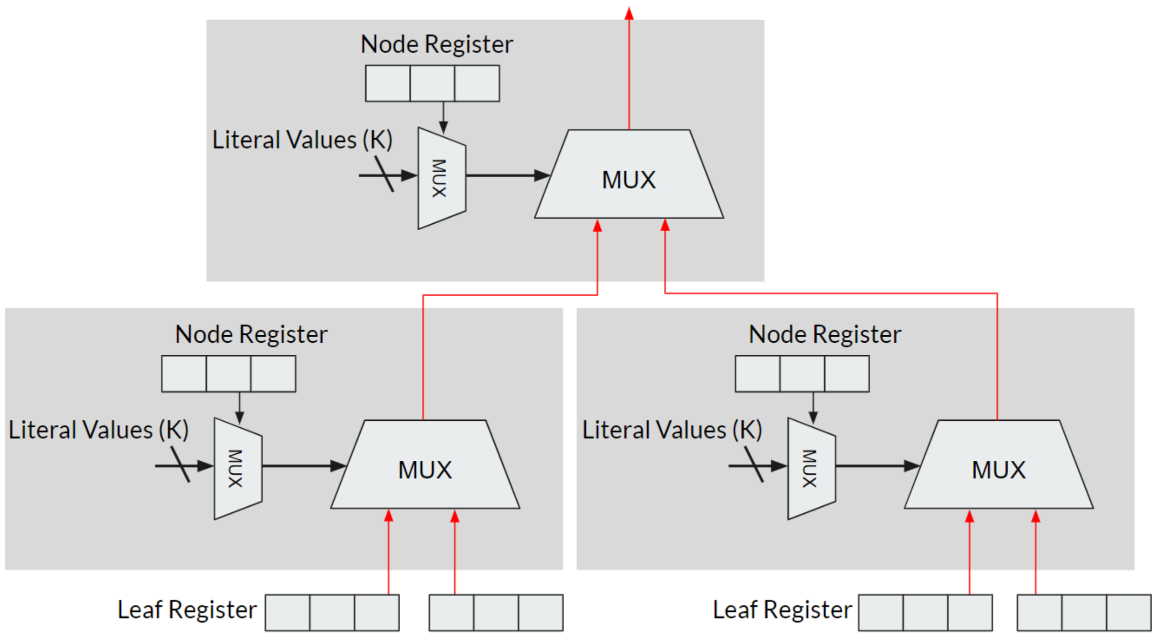
```
    reg [6:0]       mem[1023:0];
    integer i;
    always @(posedge clk_i or negedge rst_i)
    begin
       if (rst_i == 1'b0)
            for (i=0;i<1024;i=i+1)
            begin
                mem[i] = 7'b0;
            end
       else if (wr_en_i)
            mem[addr_i] <= data_i;
    end

    always @(posedge clk_i or negedge rst_i)
    begin
        if (rst_i == 1'b0)
            data_io <= 7'b0;
        else if (rd_en_i)
            data_io <= mem[addr_i];
    end
endmodule
```

2. MUX Tree Model

The tree model is a direct interpretation of a decision tree. Remember that for a decision tree, each tree node has a criteria and 2 children: left child for False and right child for True, and the leaves are a set of values. Note that the reconfigurability requires the tree model to adjust its literals and its leaf values, which need us to allocate registers for those variables: **node register** and **leaf register**. All the literal values are passed to every node and a selection signal is read from the node register so that one specific literal is activated on one node.

After training the model, the software should call some kernel functions and write the trained data to those register. The literal should be assigned to each node and the value at each leaf should be given. A skewed tree may have useless MUX (2-MUX with data_0 equals to data_1). However, the tree's height should be enough to adapt all kinds of trees. Therefore, we have to prepare a complete binary tree with **2^(k+1)-1** nodes and **2^(k+1)** leaves.

As mentioned, the area is too large. However, a compromise could be made. We could set an upper bound of tree depth and sacrifice some accuracy for area. Notice that the area is approximately exponentially related to the depth. Therefore, MUX tree model provides us with a valid way to shrink the circuit area by slightly reducing the accuracy.



A sketch of a mux tree with depth equals to 2 is shown above.

Verilog HDL Code

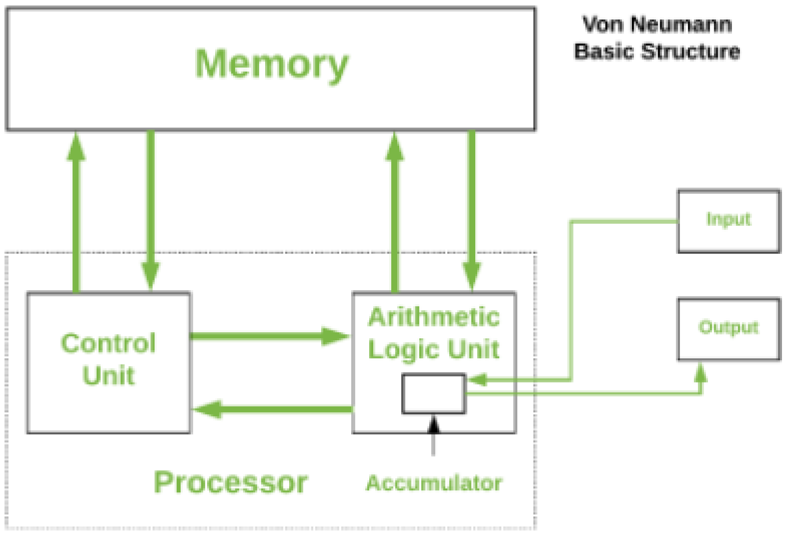
```
module mux_node(
    input [9:0] input_val,
    input [3:0] index,
    input [6:0] data_0,
    input [6:0] data_1,
    output reg [6:0] output_val
    );

    wire sel;
    assign sel = input_val[index];

    always@(sel, data_0, data_1) begin
        case(sel)
            1'b0: output_val = data_0;
            1'b1: output_val = data_1;
            default:;
        endcase
    end
endmodule
```

```
module mux_tree_10_7_2(
    input clk,
    input rst,
    input wr_leaf_en,
    input [2:0] wr_leaf_addr,
    input [6:0] wr_leaf_val,
    input wr_node_en,
    input [1:0] wr_node_addr,
    input [3:0] wr_node_val,
    input [9:0] input_val,
    output [6:0] output_data
    );

    reg [3:0] index_mem[2:0];
    reg [6:0] leaf_value[3:0];
    wire [6:0] data_2_1;
    wire [6:0] data_2_2;

    mux_node node_1_1(
        .input_val(input_val),
        .index(index_mem[0]),
        .data_0(data_2_1),
        .data_1(data_2_2),
        .output_val(output_data)
    );

    mux_node node_2_1(
        .input_val(input_val),
        .index(index_mem[1]),
        .data_0(leaf_value[0]),
        .data_1(leaf_value[1]),
        .output_val(data_2_1)
    );

    mux_node node_2_2(
        .input_val(input_val),
        .index(index_mem[2]),
        .data_0(leaf_value[2]),
        .data_1(leaf_value[3]),
        .output_val(data_2_2)
    );

    // mux node
    integer i;
    always @(posedge clk or negedge rst)
    begin
        if (rst == 1'b0)
            for (i=0;i<3;i=i+1)
                index_mem[i] <= 4'b0;
        else if (wr_node_en)
            index_mem[wr_node_addr] <= wr_node_val;
    end

    // leaf node
    always @(posedge clk or negedge rst)
    begin
        if (rst == 1'b0)
            for (i=0;i<4;i=i+1)
                leaf_value[i] <= 7'b0;
        else if (wr_leaf_en)
            leaf_value[wr_leaf_addr] <= wr_leaf_val;
    end

endmodule
```

Part 3: A New Architecture

1. Background

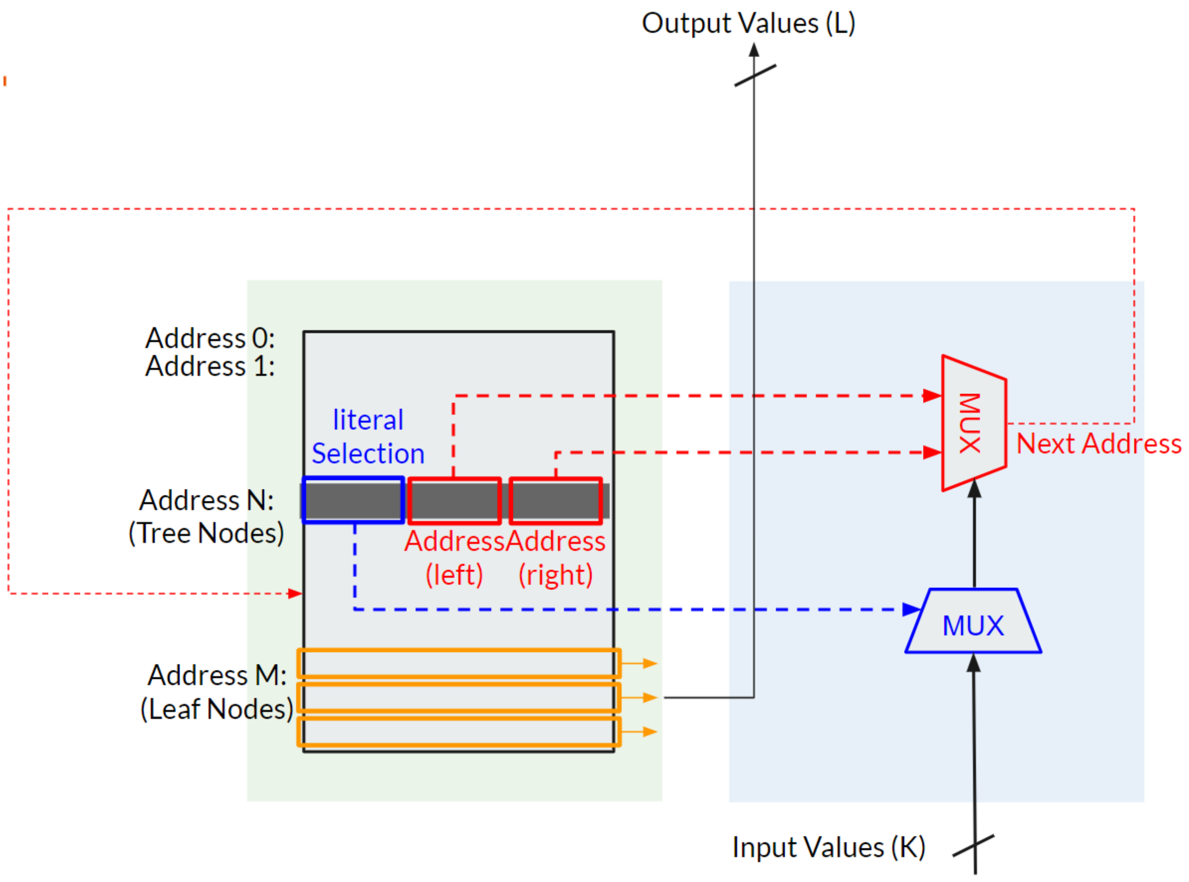
Von Neumann Architecture is the foundation of modern computers. Its idea is to divide a computer into an arithmetic and logic unit (ALU), and a memory. Note that both instructions and data are stored in the memory, and in each iteration, the CPU first fetches instructions, does some operation based on the command stored in such instruction, and calculates the next instruction to run.



picture reference: https://www.geeksforgeeks.org/computer-organization-von-neumann-architecture/

2. Overview

The truth table is flexible but has large area, and the MUX tree have to sacrifice its flexibility to reduce the area. Is there a way to keep both the performance and the flexibility? Here we come up with a new structure to solve this problem.

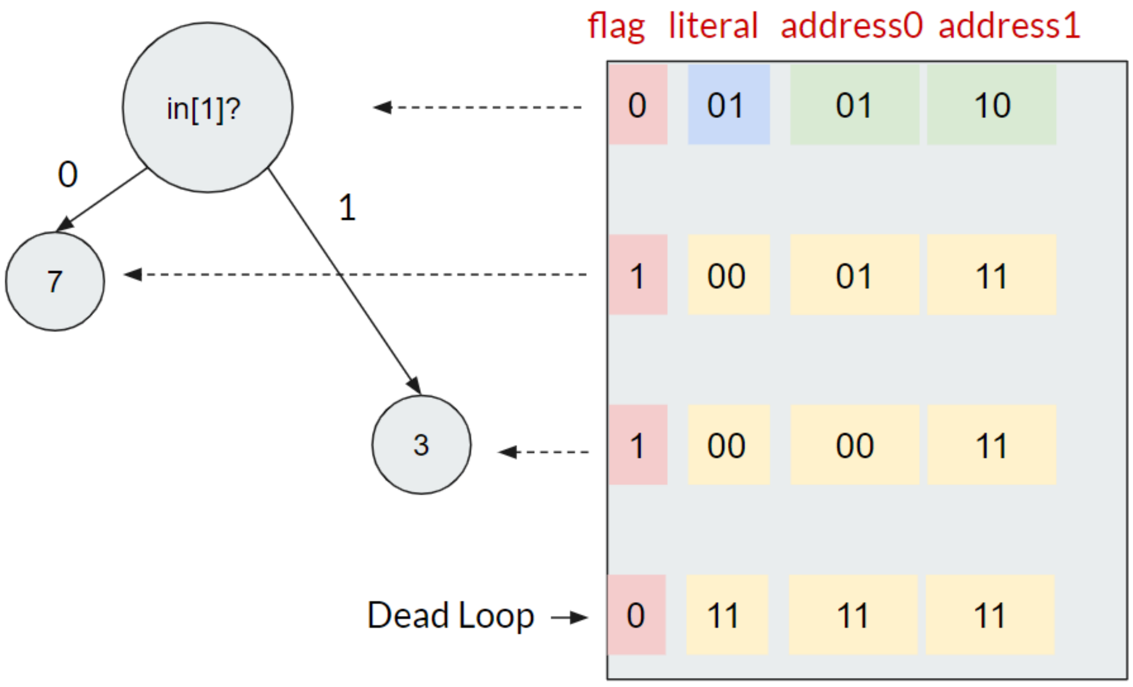


Our structure is shown above. It has 2 components, a register and a combinational logic part. Similar to von Neumann Architecture, the register stores both the instructions (calculations and decisions to made) and data (the value of leaf nodes). And similarly, the combinational part on the left is similar to ALU, in each cycle, it not only make the decision, but also return the next `$ip` value, which is the next decision to run.

3. An Example

The representation of a tree model with depth 1 is shown below. In this model we set `i` = 1, so there are 1 tree node and 2 leaf nodes. Therefore 3 entries are stored in the register. The work flow is:

- **Step 1:** Always starts from address = 0
- **Step 2:** Read the corresponding data from the memory and get `literal` = 1, `address 0` = 1 and `address 2` = 2.
    - if `in[1]` is False, go to `address0`
    - if `in[1]` is True, go to `address1`
- **Step 3:** Since `flag` is `1`, we have reached a leaf node. Read the following digits as an output.
- **Step 4:** Set the next address to `11` and it will lock the tree model. Then the output value will not change.



3. Verilog HDL Code

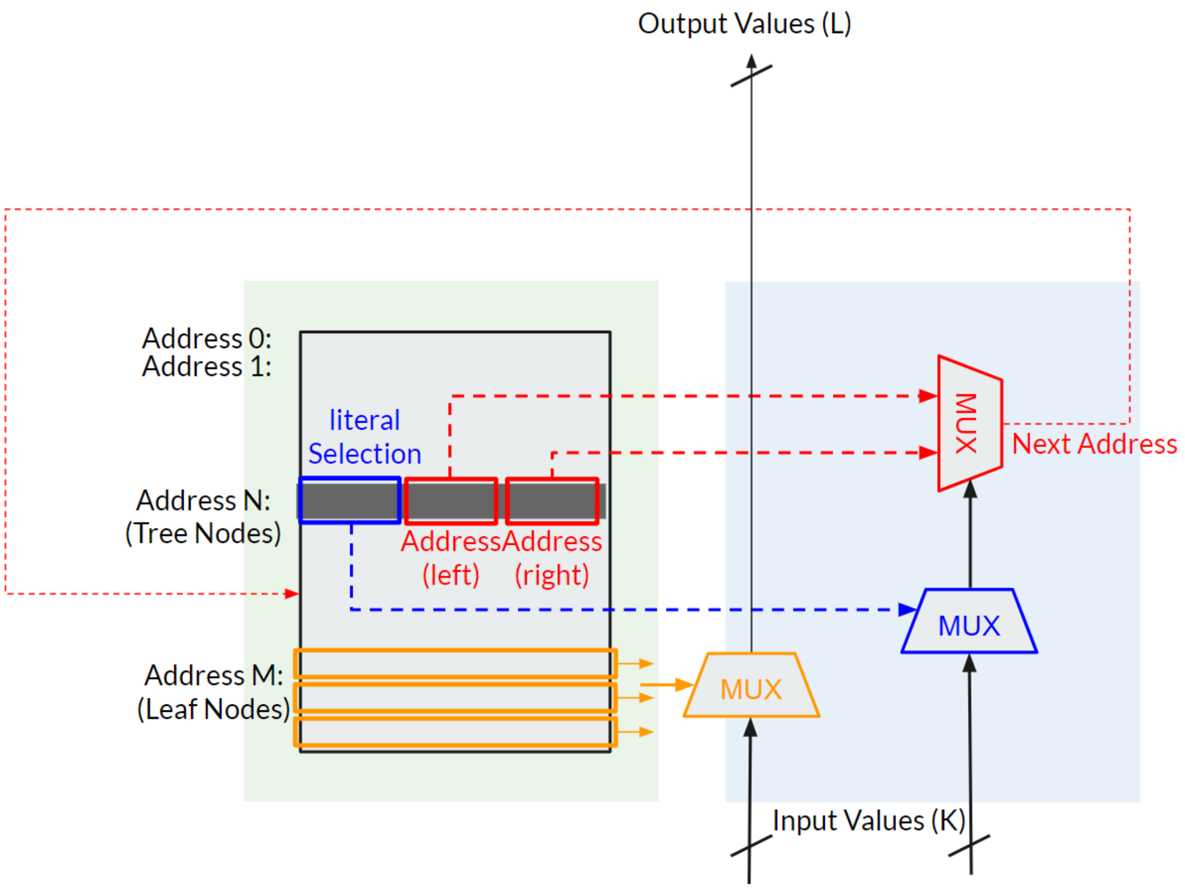
```
module logic(
    input [12:0]        data,
    input [15:0]        i_val,
    output reg [3:0]    address,
    output              flag
);
    wire [3:0] sel;
    reg add_sel;
    wire [3:0] add_0;
    wire [3:0] add_1;
    assign sel = data[3:0];
    assign add_0 = data[7:4];
    assign add_1 = data[11:8];
    assign flag = data[12];
    always @ (sel, i_val) begin
        add_sel = i_val[sel];
    end
    always @ (add_sel, add_0, add_1) begin
        case (add_sel)
            0: address = add_0;
            1: address = add_1;
            default:;
        endcase
    end
endmodule

module top(
    input                   clk_j,
    input                   rst_j,
    input                   wr_en,
    input                   rd_en,
    input [15:0]            i_val,
    output reg [11:0]       o_val
);
    reg [3:0] address;
    wire [3:0] next_address;
    wire [12:0] data;
    wire flag;
    ram my_ram(
        .clk_i(clk_j),
        .rst_i(rst_j),
        .wr_en_i(wr_en),
        .rd_en_i(rd_en),
        .addr_i(address),
        .data_io(data)
    );

    logic my_logic(
        .data(data),
        .i_val(i_val),
        .address(next_address),
        .flag(flag)
    );
    always @(posedge clk_j or negedge rst_j)
    begin
        if (rst_j == 1'b0) begin
            address <= 4'b0;
            o_val <= 12'b111111111111;
        end
        else if (flag == 1'b1) begin
            address <= 4'b1111;
            o_val <= data;
        end
        else begin
            address <= next_address;
            o_val <= 12'b111111111111;
        end
    end

endmodule
```

Part 4: An Advanced Model

Thanks to the flexibility of our model, many patches could be added on this model to implement different functions.

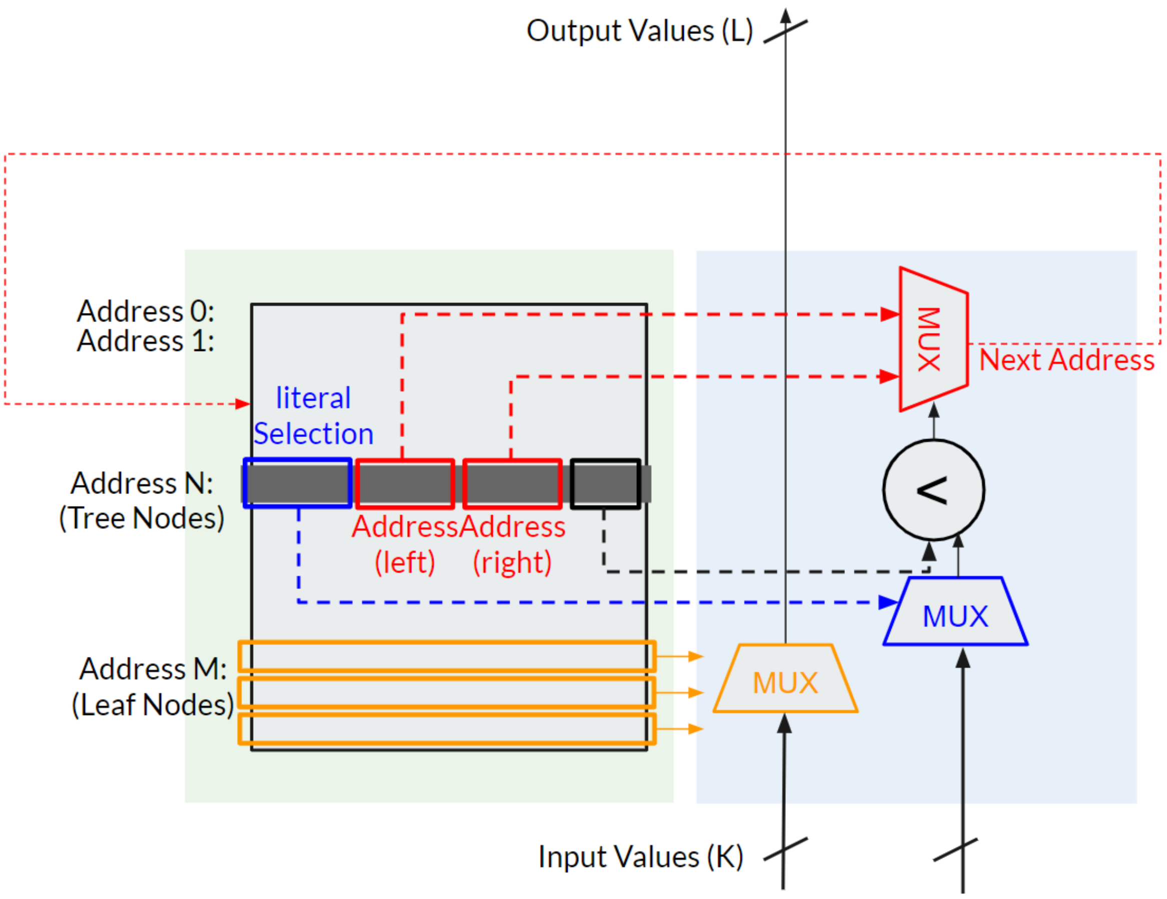
1. Linear Model

A straight-forward modification is to store literal instead of constant at each leaf node. When deriving the output, we simply use a MUX to select the corresponding input.



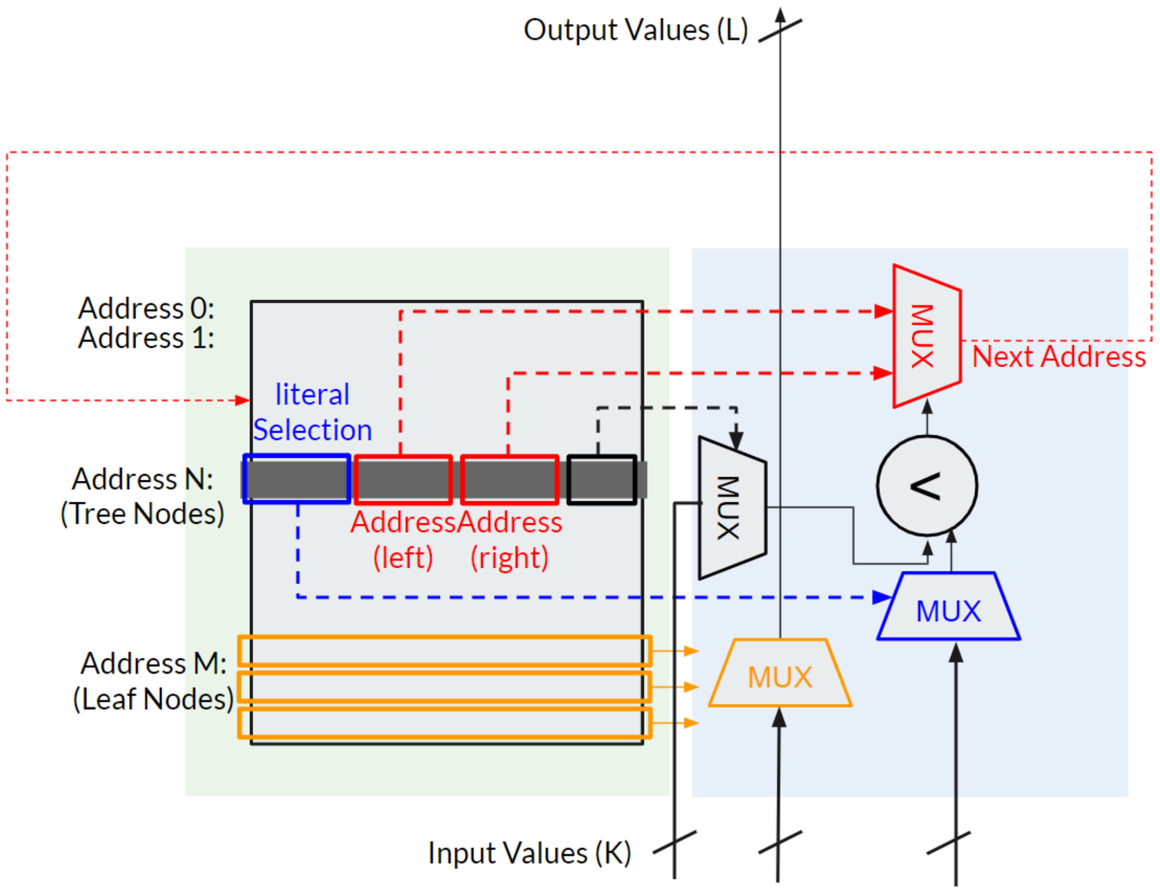
2. Non-bitwise Comparisons

Notice that the original model directly use one bit of input value and make decisions based on its value (0 or 1). However, since most of the inputs are multi-bits, non-bitwise comparisons and decision could be more effective. To achieve non-bitwise comparison, we pass the result of a comparison to the MUX instead of the raw input value. Besides, we have to store the constant for each decision in the memory.



3. Literal Comparisons

To enable comparisons between literals, we use the constant stored in each entry for literal selection instead of passing it directly to the comparator.



4. More Flexibility

We have introduced many kinds of ALU designs so far, linear functions, multi-bit comparisons and literal comparisons. But the type of model have to be pre-defined, which is to say the linear function model shown above can not be used to implement step function model. A feasible solution is to define the operation code as CPU does. The operation code can be achieved using only 1 bit.

```
reg opcode;
always @ (...) begin
    case (opcode)
        0: addr_sel = input_val[literal_sel] < comp_val;
        1: addr_sel = input_val[literal_sel] < input_val[comp_val];
        default:;
    endcase
end
```

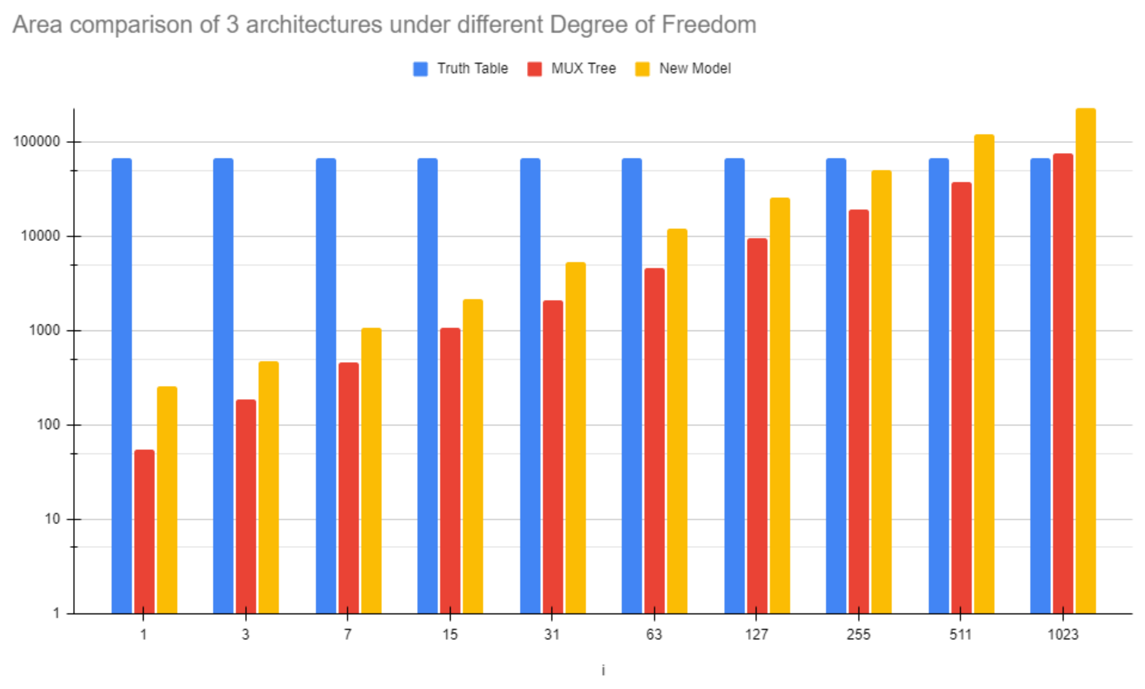
The size of register will increase obviously as we store more information in each entry. The size of opcode will increase if we allow more kinds of calculations. We have to consider the trade-off between area and flexibility.

Part 5: Experiments

We synthesis the implementation using ASAP 7nm library. Notice that the library does not contain 2-MUX, arithmetic unit, or any customized RAM. Therefore, our memory is implemented using DFF and 2-MUX using ANDs and ORs.

Top Module Name	**Combinational Area (um^2)**	**Sequential Area (um^2)**	**Total Area (um^2)**
10 input Truth table (1024*7 RAM)	24681.10	43508.58	68189.69
MUX Tree, depth = 0	11.90	42.46	54.25
MUX Tree, depth = 1	77.68	109.18	186.86
MUX Tree, depth = 2	220.68	242.61	463.29
MUX Tree, depth = 3	594.87	485.22	1080.09
MUX Tree, depth = 4	1041.13	1043.23	2084.36
MUX Tree, depth = 5	2598.04	2086.46	4684.50
MUX Tree, depth = 6	5240.87	4221.43	9462.30
MUX Tree, depth = 7	10593.01	8491.39	19084.40
MUX Tree, depth = 8	21144.97	17031.31	38176.27
MUX Tree, depth = 9	42443.20	34111.13	76554.33
MUX Tree, depth = 10	84805.68	68270.79	153076.47
New Architecture, `i` = 1	77.21	182.89	260.10
New Architecture, `i` = 3	150.23	320.99	471.23
New Architecture, `i` = 7	394.94	679.31	1074.25
New Architecture, `i` = 15	722.00	1419.28	2141.28
New Architecture, `i` = 31	2208.93	3096.56	5305.49
New Architecture, `i` = 63	5171.12	6835.57	12006.69
New Architecture, `i` = 127	11014.32	15017.83	26031.95
New Architecture, `i` = 255	18195.14	32874.75	51069.89
New Architecture, `i` = 511	51200.53	71752.79	122952.79
New Architecture, `i` = 1023	74682.80	155594.03	230226.82

The plot is shown below. The y-axis is the log of area and the x-axis is the degree of freedom (`i` as mentioned before). The blue line is the area of truth table model, which is a constant. The red bar is MUX tree model, the value of which grows exponentially with the tree depth. The yellow bar is area of our model, roughly 2 times the area of MUX tree with same number of nodes.



Part 6: Conclusions

In this report, we introduce an architecture of reconfigurable model. Comparing to the