Automated fitting of MUX and adder in LUTs

Group Members :

Ankita Pasad (12D070021)
Madhu Lekha G (11D070063)

Problem Statement :

As known already, the concept of Look-up Table (LUT) is commonly used to reduce processing time for applications that uses complex calculations. Basically, the LUT contains data or results from the complex calculations needed by application, which was done beforehand—once. By keeping the results in the LUT, when the application needs the values, instead of having to do the calculations, it can just refer to the LUT and retrieve the values from it; bypassing the calculations.

Our goal is to model a multiplexer of order n (specified by the user) in k-input LUTs (which is also set by the user).

Input/Output Description :

Input: k (no of inputs for an LUT that are allowed)

n (order of MUX)

MUX data lines/inputs (in order to verify the output)

MUX select lines (to select which mux input has to appear at the output)

Output: number of LUTs required

output of the LUT-based implementation of MUX for the given inputs.

Assumptions :

- k,n are positive integers (k>1)
- All MUX inputs, select lines are binary values i.e. 0 or 1
- An LUT can have only one output

Algorithm pseudo-code :

Sub-optimal solution

Algorithm : mux_implementation

Inputs : k,n,mux_inputs,select_inputs

calculate : no_inputs = 2^n;

Functions :

int mux_final(n,k) : // Diverting to cases according to relation between k,n

case 1 if k ==n+1;

case 2 if k < n+1;

case 3 if k >n+1;

int case1() :

for(int i=0; i<no_inputs; i++)

{

give MUX input to LUT function one-by-one;

function call to single_op_LUT_case_1 which actually implements the LUT;

set output[i] to the value obtained by above function call;

Increment the number of LUTS;

Increment no_OR which denotes the number of inputs that are to be finally ORed;

}

function call final_OR;

set answer to the value of above function call;

return answer;

int case2() :

find number of levels in each channel = ceil(n/k-1) //there will be no_inputs channel in all

for(int i=0; i<no_inputs; i++){

set input as i^{th} mux_input;

function call to single_op_LUT_case_2 which implements all the LUTs in that particular channel;

Increment the number of LUTs by no_level;

Increment no_OR, which denotes the number of inputs that are to be finally ORed, by 1;

function call final_OR;

set answer to the value of above function call;

return answer;

}

int case3() :

find number of stage1 LUTS = ceil(no_inputs/k-n);

while(i<no_inputs)

{

prepare MUX inputs for LUT by windowing to length of k-n;

function call to single_op_LUT_case_3 which actually implements the LUT;

set output[i] to the value obtained by above function call;

Increment the number of LUTS;

Increment no_OR which denotes the number of inputs that are to be finally ORed;

i = i+k-n;

}

if number of stage1 LUTs = 1, then answer = output[0];

else **function call** final_OR;

set answer to the value of above function call;

return answer;

int single_op_LUT_case_1(index of input, mux_input, select_inputs)

{

flag = 0;

Convert input index to binary;

for(int i = 0; i<n; i++){

Compare binary value of input index with select inputs;

if all match, do nothing ;

else flag =1, break ;

}

```
if flag = 0, then answer = mux_input;
else answer = 0 ;
return answer;
}

int single_op_LUT_case_2(index of input,mux_input,select_inputs for that LUT, no of levels)
{
flag = 0;
convert index_ip to binary of n bits
for(int i=0; i<no_levels; i++){ //finding the answer of LUT at each level
compare (k-1) bits of select lines and appropriate (this appropriate number is decided by
the level number)  k-1 bits of mux_input index converted to binary
If they all perfectly match then set the output as the corresponding mux_input and break
the loop, else keep the output as zero
}

int single_op_LUT_case_3(index of input, mux_input, select_inputs,  number of stage1 LUTS )
{
for(int j = 0; j<k-n; j++){ // i.e for the inputs in the window of length
Convert input index to binary;
for(int i = 0; i<n; i++){
Compare binary value of input index with select inputs;
if all match, do nothing ;
else flag =1, break ;
}
if flag = 0, then answer = mux_input;
else answer = 0 ;
}
return answer;
}
```

int* int_to_binary(int number)

{

Converts an integer to binary;

}

int final_OR(result)

{

Find the number of OR_LUTs required from k and in_OR

no_OR_LUT = ceil((in_OR-k)/(k-1)) + 1;

update the no_LUT to no_LUT + no_OR_LUT;

Divide the in_OR number of inputs in no_OR_LUT LUTs, forming a chain from previous OR_LUT outputs and so on;

Every time, select which k number of inputs are to be passed to OR_LUT which is designed to implement a k-input OR gate;

return the final answer;

}

Thus, we have total number of LUTs used in the implementation stored in no_LUT (it is a global variable being updated in the appropriate loops), and the final output of mux via LUT implementation.

Optimal solution

Algorithm : mux_implementation

Inputs : k,n,mux_inputs,select_inputs

calculate : no_inputs = 2^n;

Functions :

int mux_final(n,k) : // Diverting to cases according to relation between k,n

calculate no of select inputs each LUT of size k is to be given ;

no of MUX inputs at each LUT = k-no of select inputs each LUT;

Calculate number of LUT stages (excluding or_LUTs)

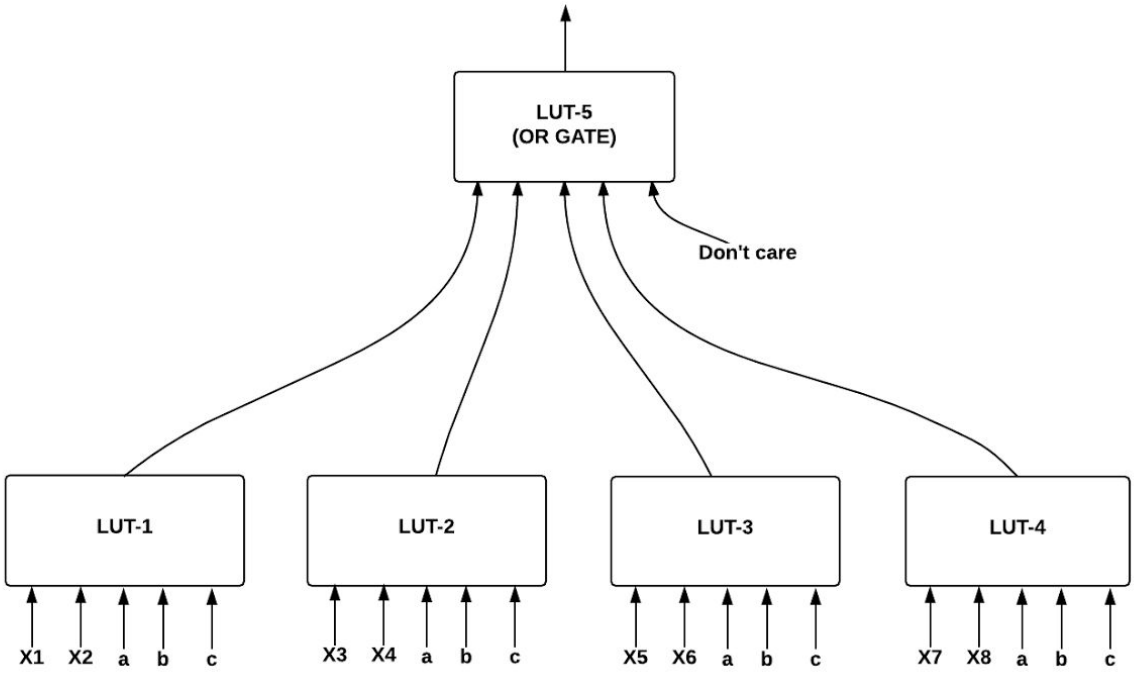
```
 = ceil(float(n)/float(no_of_select_input_lut))
for(int stage=1; stage<=no_of_LUT_stages; stage++)
{
Find number of LUTs that are required in the stage;
while (i < number of LUTs in previous stage)
{
Find the select input indices that are to be fed into the LUTs;
Note the binary values held by those select input container cells;
prepare MUX inputs for LUT by windowing their indices to length of no of MUX inputs;
function call to single_op_LUT which implements the LUT actually;
set output[i] to the value obtained by above function call;
Increment the number of LUTs;
i = i+k-n;
}
Number of inputs to be sent to further OR gates = no of output from last stage of LUTs
calculated above ;
if  number of stage 1 LUTs = 1,  then answer = output[0];
else function call final_OR;
set answer to the value of above function call;
return answer;
}

int single_op_LUT_(windowed indices of mux_inputs, select input indices, select input values,
no_mux_inputs)
{
Compare binary value of bits of input index that correspond to indices of select inputs;
 if all match, do nothing ;
else flag =1, break ;
if flag = 0, then answer = mux input index;
else answer = -1 ;
return answer;
}
```

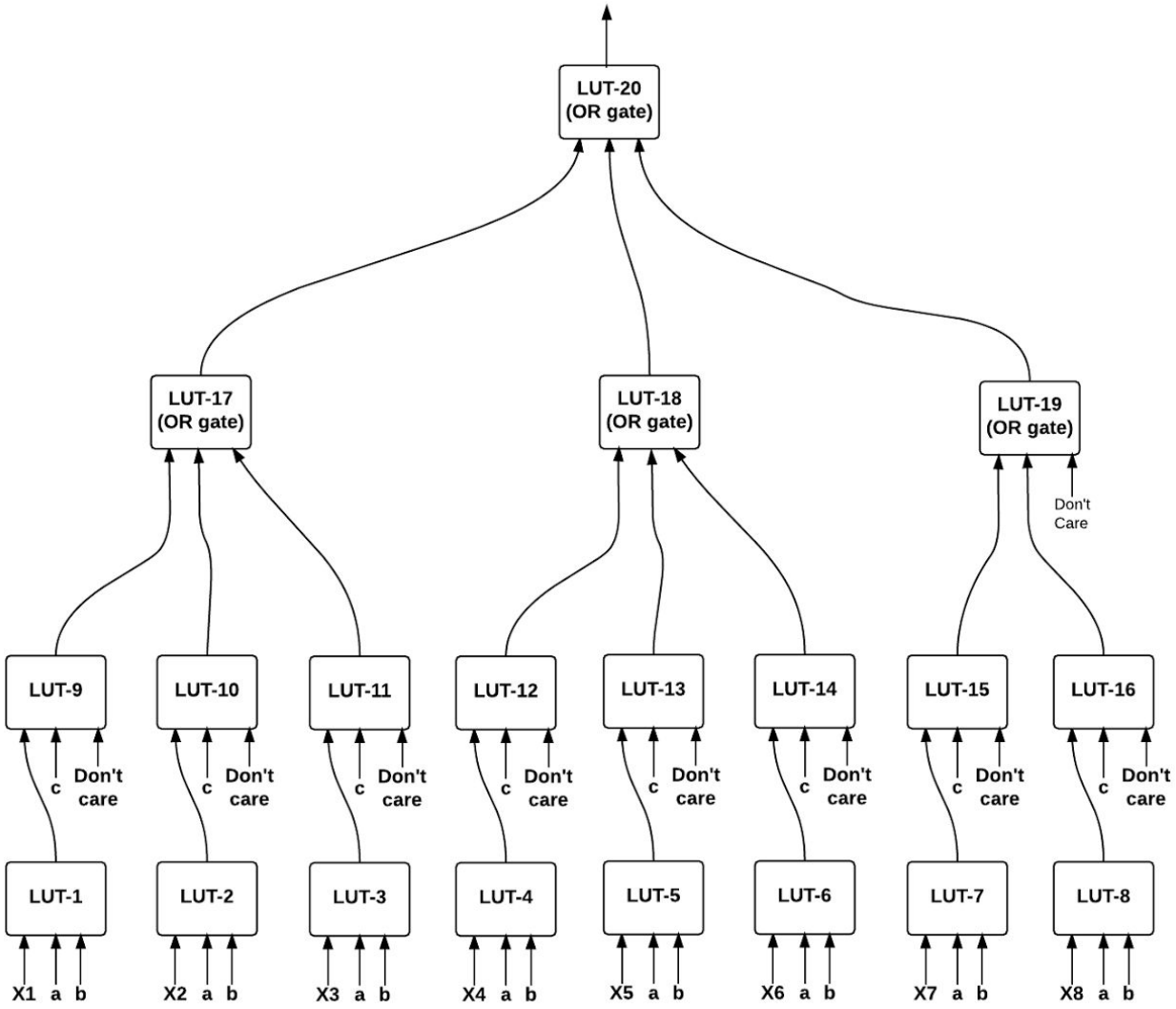
```
int* int_to_binary(int number)
        {
        Converts an integer to binary;
        }
int final_OR(result)
        {
        Find the number of OR_LUTs required from k and in_OR
        no_OR_LUT = ceil((in_OR-k)/(k-1)) + 1;
        update the no_LUT to no_LUT + no_OR_LUT;
        Divide the in_OR number of inputs in no_OR_LUT LUTs, forming a chain from
        previous OR_LUT outputs and so on;
        Every time, select which k number of inputs are to be passed to OR_LUT which is
        designed to implement a k-input OR gate;
        return the final answer;
        }
```

Adder :

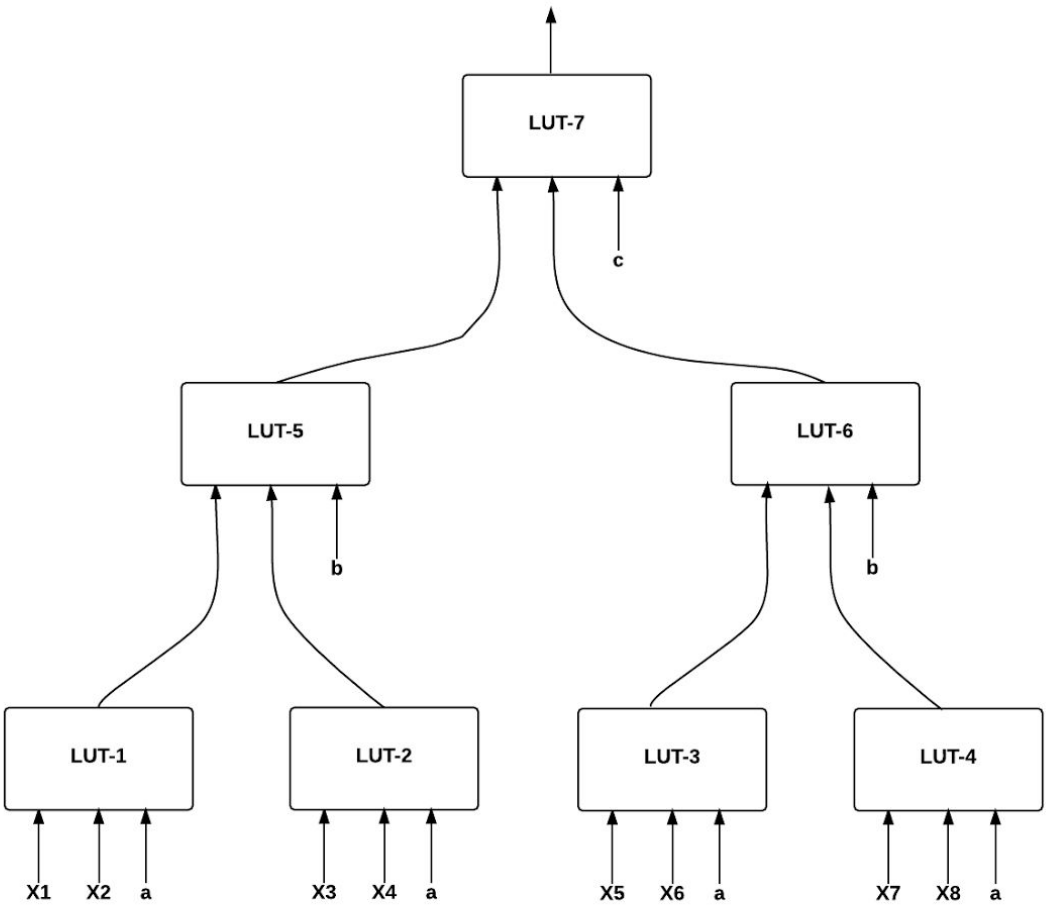
A representative case for k>n+1 for the suboptimal algorithm - (k=5, n=3)



A representative case for k<n+1 for the suboptimal algorithm - (k=3, n=3)



A representative case for the optimal algorithm - (k=3, n=3)



Issues faced :

As such there are no issues, however we started with a very inefficient code, then a better version clicked, which is the sub-optimal one described in the report. The sub-optimal one is efficient for cases k>n+1, but still underperforms with k<n+1. We came to an optimal solution trying to solve for the case k<n+1, but the code has some minute errors which we were not able to debug being a last minute thought !

References :

- https://en.wikipedia.org/wiki/Lookup_table
- http://www2.engr.arizona.edu/~rlysecky/courses/ece274-05f/lectures/lecture23.pdf
- Class notes of Patkar sir