Report on

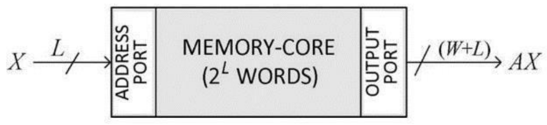
LUT Optimization for Memory-Based Computation

By: Priyansh Kasyap

Along with the progressive device scaling, semiconductor memory has become cheaper, faster, and more power-efficient. Embedded memories will have dominating presence in the system on-chips (SoCs), which may exceed 90% of the total SoC content. It has also been found that the transistor packing density of memory components is not only higher but also increasing much faster than those of logic components. Because of these reasons, Memory based computations are rapidly gaining in popularity .

Apart from that, memory-based computing structures are more regular than the multiply–accumulate structures i.e easy to lay out and offer many other advantages, e.g., greater potential for high-throughput and low-latency implementation and less dynamic power consumption. This kind of computation is really helpful for DSP applications which involve multiplication with a fixed set of coefficients.

A conventional lookup-table (LUT)-based multiplier is shown in Fig. 1, where A is a fixed coefficient, and X is an input word to be multiplied with A. Assuming X to be a positive binary number of word length L, there can be 2^L possible values of X, and accordingly, there can be 2^L possible values of product C = A · X.

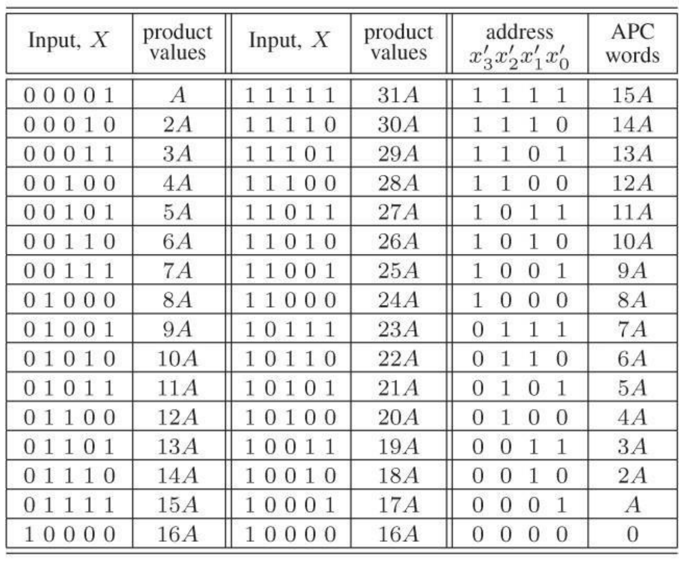


Thus an LUT of 2^L words, consisting of precomputed product values corresponding to all possible values of X, is conventionally used. The product word A · Xi is stored at the location Xi for $0 \le Xi \le 2^L - 1$, such that if an L-bit binary value of Xi is used as the address for the LUT, then the corresponding product value A · Xi is available as its output.

In this project, using combined techniques of APC (Antisymmetric Product Coding) and OMS (Odd Multiple Storage) scheme, a reduction in LUT size to one-fourth of the conventional LUT is achieved where APC provides reduction by 2 and OMS provides a further reduction by 2.

APC Technique

TABLE I
APC WORDS FOR DIFFERENT INPUT VALUES FOR $L = 5$



Input, X	product values	Input, X	product values	address $x'_3x'_2x'_1x'_0$	APC words
0 0 0 0 1	A	1 1 1 1 1	$31A$	1 1 1 1	$15A$
0 0 0 1 0	$2A$	1 1 1 1 0	$30A$	1 1 1 0	$14A$
0 0 0 1 1	$3A$	1 1 1 0 1	$29A$	1 1 0 1	$13A$
0 0 1 0 0	$4A$	1 1 1 0 0	$28A$	1 1 0 0	$12A$
0 0 1 0 1	$5A$	1 1 0 1 1	$27A$	1 0 1 1	$11A$
0 0 1 1 0	$6A$	1 1 0 1 0	$26A$	1 0 1 0	$10A$
0 0 1 1 1	$7A$	1 1 0 0 1	$25A$	1 0 0 1	$9A$
0 1 0 0 0	$8A$	1 1 0 0 0	$24A$	1 0 0 0	$8A$
0 1 0 0 1	$9A$	1 0 1 1 1	$23A$	0 1 1 1	$7A$
0 1 0 1 0	$10A$	1 0 1 1 0	$22A$	0 1 1 0	$6A$
0 1 0 1 1	$11A$	1 0 1 0 1	$21A$	0 1 0 1	$5A$
0 1 1 0 0	$12A$	1 0 1 0 0	$20A$	0 1 0 0	$4A$
0 1 1 0 1	$13A$	1 0 0 1 1	$19A$	0 0 1 1	$3A$
0 1 1 1 0	$14A$	1 0 0 1 0	$18A$	0 0 1 0	$2A$
0 1 1 1 1	$15A$	1 0 0 0 1	$17A$	0 0 0 1	A
1 0 0 0 0	$16A$	1 0 0 0 0	$16A$	0 0 0 0	0

For $X = (0\ 0\ 0\ 0\ 0)$, the encoded word to be stored is $16A$.

For simplicity of presentation, we assume both X and A to be positive integers. The product words for different values of X for L = 5 are shown in Table I. It may be observed in this table that the input word X on the first column of each row is the two's complement of that on the third column of the same row. In addition, the sum of product values corresponding to these two input values on the same row is 32A.

Let the product values on the second and fourth columns of a row be u and v, respec tively. Since one can write u = [(u + v)/2 − (v − u)/2] and v = [(u + v)/2+(v − u)/2], for (u + v) = 32A, we can have

$$u = 16A - \left[\frac{v-u}{2}\right] \quad v = 16A + \left[\frac{v-u}{2}\right].$$

The product values on the second and fourth columns of Table I therefore have a negative mirror symmetry. This behavior of the product words can be used to reduce the LUT size, where, instead of storing u and v, only [(v − u)/2] is stored for a pair of input on a given row. The 4-bit LUT addresses and corresponding coded words are listed on the fifth and sixth columns of the table, respectively. Since the representation of the product is derived from the antisymmetric behavior of the products, we can name it as antisymmetric product code(APC) .

4-bit address $X' = (x'_3x'_2x'_1x'_0)$ of the APC word is given by

$$X' = \begin{cases} X_L, & \text{if } x_4 = 1 \\ X'_L, & \text{if } x_4 = 0 \end{cases} \qquad (2)$$

Where X_L' is 2's complement of X_L and X_L is the 4 LSBs of X.

$$\text{Product word} = 16A + (\text{sign value}) \times (\text{APC word}) \qquad (3)$$

where sign value $= 1$ for $x_4 = 1$ and sign value $= -1$ for $x_4 = 0$. The product value for $X = (10000)$ corresponds to APC value "zero," which could be derived by resetting the LUT output, instead of storing that in the LUT.

OMS Technique

TABLE II
OMS-BASED DESIGN OF THE LUT OF APC WORDS FOR $L = 5$

input X' $x'_3x'_2x'_1x'_0$	product value	# of shifts	shifted input, X''	stored APC word	address $d_3d_2d_1d_0$
0 0 0 1	A	0	0 0 0 1	$P0 = A$	0 0 0 0
0 0 1 0	$2 \times A$	1			
0 1 0 0	$4 \times A$	2			
1 0 0 0	$8 \times A$	3			
0 0 1 1	$3A$	0	0 0 1 1	$P1 = 3A$	0 0 0 1
0 1 1 0	$2 \times 3A$	1			
1 1 0 0	$4 \times 3A$	2			
0 1 0 1	$5A$	0	0 1 0 1	$P2 = 5A$	0 0 1 0
1 0 1 0	$2 \times 5A$	1			
0 1 1 1	$7A$	0	0 1 1 1	$P3 = 7A$	0 0 1 1
1 1 1 0	$2 \times 7A$	1			
1 0 0 1	$9A$	0	1 0 0 1	$P4 = 9A$	0 1 0 0
1 0 1 1	$11A$	0	1 0 1 1	$P5 = 11A$	0 1 0 1
1 1 0 1	$13A$	0	1 1 0 1	$P6 = 13A$	0 1 1 0
1 1 1 1	$15A$	0	1 1 1 1	$P7 = 15A$	0 1 1 1

From this table, its clear that it is sufficient to store only the odd multiple APC words as the even multiples can be obtained by left shifting accordingly as given in the table.

$$\text{Product word} = 16A + (\text{sign value}) \times (\text{APC word})$$

We want to have product values as 0 and 32 A as well. So, need to store APC word 16 A. However, if 16A is not derived from A, only a maximum of three left shifts is required to obtain all other even multiples of A.

TABLE III
PRODUCTS AND ENCODED WORDS FOR $X = (00000)$ AND (10000)

input X $x_4x_3x_2x_1x_0$	product values	encoded word	stored values	# of shifts	address $d_3d_2d_1d_0$
1 0 0 0 0	$16A$	0	$---$	$--$	$---$
0 0 0 0 0	0	$16A$	$2A$	3	1 0 0 0

A maximum of three bit shifts can be implemented by a two-stage logarithmic barrel shifter, but the implementation of four shifts requires a three-stage barrel shifter. Therefore, it would be a more efficient strategy to store 2A for input X = (00000), so that the product 16A can be derived by three arithmetic left shifts. The product values and encoded words for input words X = (00000) and (10000) are separately shown in Table III. For X = (00000), the desired encoded word 16A is derived by 3-bit left shifts of 2A [stored at address (1000)]. For X = (10000), the APC word "0" is derived by resetting the LUT output, by an active-high RESET signal given by RESET = (x0 + x1 + x2 + x3)′ · x4.

So, the possible product values we can get:

0,A,2A,….,31A.

For the mappings , refer to the Paper.

Code:-

```
module modified_LUT(input [4:0] X,output [3:0] d, output [10:0] product,output reg [9:0]
LUT_OUT,output [9:0] barrel_out,output s0,s1,output [8:0] w1 );

//made by Priyansh Kasyap

//W refers to the no.of bits in the binary

// representation of A. Here A is 32. Thus, W=6

//L refers to the size of the bus of  input x. Here, L=5 is taken.

parameter W=6;

parameter A=32;

address_generation_unit aa(X,d);   // This block will map the input address X to the address which goes to
the 4 to 9 decoder

// notation used same as in the paper.

//" LUT Optimization for Memory-Based Computation"

wire reset;
```

```
four_to_nine_decoder(d,w1); // w1 is the address given to the LUT . In paper, it is represented as w.

control_ckt(X,d[3],s0,s1,reset); // produces the s0,s1 and reset signal

 barrel b11( LUT_OUT ,{s1,s0} ,barrel_out); // barrel shifter which produces the required left shifter
 // s1,s0 determine the amount of shifting required.



assign product = (X[4])? (16*A + barrel_out):(16*A - barrel_out) ;



always @(w1,X,reset)
begin
  if(reset==1)
  LUT_OUT=0;
  else
  begin
  case (w1)
  9'b000000001: LUT_OUT=A;
  9'b000000010: LUT_OUT=3*A;
  9'b000000100: LUT_OUT=5*A;
  9'b000001000: LUT_OUT=7*A;
  9'b000010000: LUT_OUT=9*A;
  9'b000100000: LUT_OUT=11*A;
  9'b001000000: LUT_OUT=13*A;
  9'b010000000: LUT_OUT=15*A;
  9'b100000001: LUT_OUT=2*A;   // 2A is stored here. Why? Explanation in paper
  endcase
  end
```

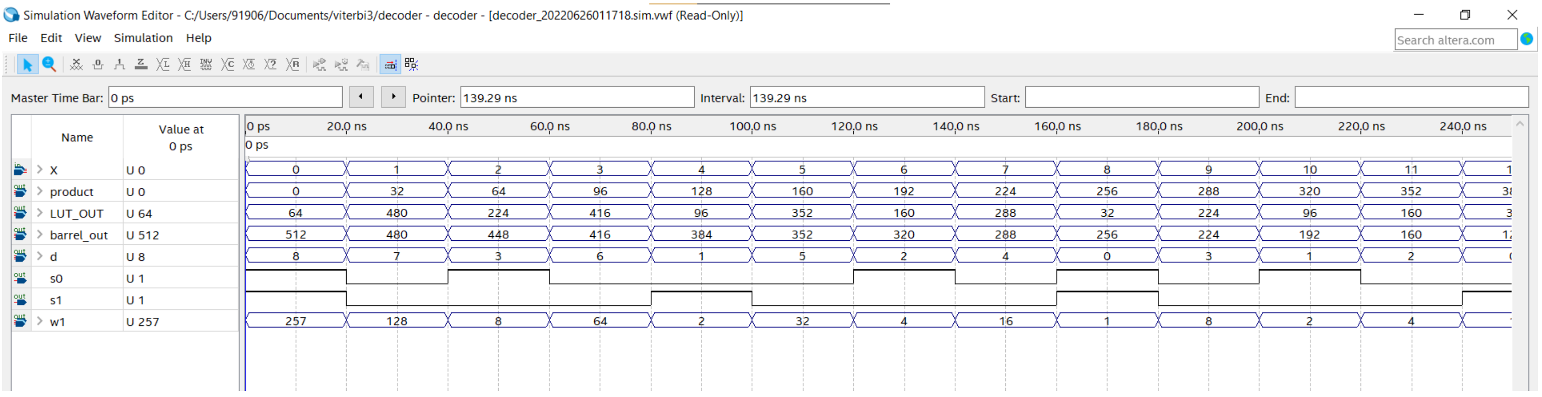
```
end




endmodule



module control_ckt(input [4:0] x,input d3,output s0,s1,reset);

assign s0= ~(t1|x[0]);
assign t1=~(~x[2]| x[1]);

assign s1=~(x[0]| x[1]);

assign reset=d3 & x[4];

endmodule




module address_generation_unit(input [4:0] X,output reg [3:0] d,Xbarbar,output [3:0]Xbar);

wire [3:0] Xlbar;
two_comp t1(X[3:0] , Xlbar);
 // Xlbar stores the 2's complement of the 4 LSBs of input X
assign Xl=X[3:0]; // Xl stores just the 4LSBs of X
//notations as per the paper
assign Xbar=X[4]?X[3:0] :Xlbar;

always @(Xbar)
```

```
//defining Xbarbar here
begin

 if (Xbar[2:0]==0)
  Xbarbar={3'b0,Xbar[3]};
  else if (Xbar[1:0]==0)
  Xbarbar={2'b0,Xbar[3:2]};
  else if ( Xbar[0]==0)
  Xbarbar={1'b0,Xbar[3:1]};
  else
  Xbarbar=Xbar;

end

// mapping of Xbarbar to d

integer i;
always @(Xbarbar)
begin
 d[3]= ~(Xbarbar[0]);
for(i=0;i<=2;i=i+1)
begin
 d[i]=Xbarbar[i+1];
 end
 end
endmodule
```

```
//operand size for barrel shifting is W+4=10 here.

//need to implement maximum shifting =3






module barrel( input [9:0] d_in ,input [1:0] control , output [9:0] y); //barrel shifter architecture from Prof.
Dinesh Sharma's Class


wire [9:0] q;

row_of_mux r1(d_in,{d_in[7:0],2'b0},control[1],q);


row_of_mux r2(q,{q[8:0],1'b0},control[0],y);




endmodule


module mux_2to1(input a,input b, input sel ,output out1);


 assign out1=sel?b:a;


 endmodule


module row_of_mux(input [9:0] a,input [9:0] b ,input sel, output [9:0] out1);



 mux_2to1 f7(   a[9],b[9],   sel,  out1[9]);

 mux_2to1 f8(   a[8],b[8],         sel,  out1[8]);

 mux_2to1 f9(   a[7],b[7],         sel,  out1[7]);

 mux_2to1 f10(  a[6],b[6],   sel,  out1[6]);

 mux_2to1 f11(  a[5],b[5],   sel,  out1[5]);

 mux_2to1 f12(  a[4],b[4],   sel,  out1[4]);
 mux_2to1 f13(  a[3],b[3],   sel,  out1[3]);
 mux_2to1 f14(  a[2],b[2],   sel,  out1[2]);
 mux_2to1 f15(  a[1],b[1],   sel,  out1[1]);
 mux_2to1 f16(  a[0],b[0],   sel,  out1[0]);
 endmodule


module two_comp(input [3:0] in, output [3:0] out);

assign out=~in +1;

endmodule



module four_to_nine_decoder(input [3:0] d,output [8:0] w);


// Why 9 outputs required? Because we will store A,3A,5A,.....,15A and then 2A.



three_to_eight_decoder(d[2:0],w[7:0]);

assign w[8]=d[3] & w[0];

endmodule
```

```
module three_to_eight_decoder(input [2:0] d, output reg [7:0] w);


always @(d)

begin

 case(d)


 0:begin w[0]=1;w[7:1]=0; end

   1:begin w[1]=1;w[7:2]=0; w[0]=0; end

 2:begin w[2]=1;w[7:3]=0; w[1:0]=0; end

 3:begin w[3]=1;w[7:4]=0; w[2:0]=0; end

   4:begin w[4]=1;w[7:5]=0; w[3:0]=0; end

   5:begin w[5]=1;w[7:6]=0; w[4:0]=0; end

   6:begin w[6]=1;w[7]=0; w[5:0]=0; end

   7:begin w[7]=1;w[6:0]=0;  end

    endcase


end

endmodule
```

<u>Results</u>:



```
Simulation Waveform Editor - C:/Users/91906/Documents/viterbi3/decoder - decoder - [decoder_20220626011718.sim.vwf (Read-Only)]

Master Time Bar: 0 ps     Pointer: 139.29 ns     Interval: 139.29 ns     Start:     End:

Name         Value at 0 ps   0 ps  20.0 ns  40.0 ns  60.0 ns  80.0 ns  100.0 ns  120.0 ns  140.0 ns  160.0 ns  180.0 ns  200.0 ns  220.0 ns  240.0 ns
X            U 0             0     1        2        3        4        5         6         7         8         9         10        11
product      U 0             0     32       64       96       128      160       192       224       256       288       320       352
LUT_OUT      U 64            64    480      224      416      96       352       160       288       32        224       96        160
barrel_out   U 512           512   480      448      416      384      352       320       288       256       224       192       160
d            U 8             8     7        3        6        1        5         2         4         0         3         1         2
s0           U 1
s1           U 1
w1           U 257           257   128      8        64       2        32        4         16        1         8         2         4
```

We can observe that the outputs are as expected.