DEPARTMENT OF ELECTRICAL ENGINEERING,
IIT BOMBAY

A REPORT

ON

32-Bit Brent Kung Adder

June'21

AUTHOR'S NAME

Kanak Vjay (203070050)

Mohd. Faizaan Qureshi (203070062)

Introduction

First Order P and G values

P means propagate signal and G means generate signal. Basically idea is to analyze how carry depends on input carry at various stages.

Let's suppose a case , in a full adder, when A = 1, and B = 1, then irrespective of what the value of input carry is, output carry is always 1.

And other case is when A=0 and B=1, or A=1 and B=0, then output carry of the full adder depends on input carry, i.e. whatever is the input carry is going to be output carry.

And when A=0 and B=0, then output carry is going to be 0, i.e. it does not depend on input carry.

So this condition when A=1 and B=1 is therefore called generate (G) and condition when any one of A and B is 0 not both, then it is called propagate condition (P).

First order P and G values

As we have seen how output carry depends on A and B for its dependence on input carry.

So, $C_{i+i} = G_i + P_i . C_i$

Here C_{i+1} is output carry of i'th stage, and C_i is input carry to i'th stage.

Where $G_i = A_i . B_i$ and $P_i = A_i \oplus B_i$

Now we can clearly see that to produce above P and G value, we need only single bit of A and B, that's why it is called first order P and G value.

Now one important observation here is that carry of the stage (i+1) which is very next to i'th stage uses only first order P and G value. This is important to note here because when we design our Brent-Kung adder architecture then this

information is important to produce all the very next stage carry with previous stage carry with the help of first order P and G value.

Now if we put (i-1) in above carry equation then basically we will get carry equation of (i-1)th stage.

So, we will get $C_{\mathrm{i}} = G_{\mathrm{i}-1} + P_{\mathrm{i}-1} . C_{\mathrm{i}-1}$

Now put above equation in 1st equation, we will get =>

$C_{\mathrm{i}+1} = G_{\mathrm{i}} + P_{\mathrm{i}} . (G_{\mathrm{i}-1} + P_{\mathrm{i}-1} . C_{\mathrm{i}-1})$

Now we can see that after rearranging this equation, we will get just like the equation we got for the first orde case, i.e.

$C_{\mathrm{i}+\mathrm{i}} = (G_{\mathrm{i}} + P_{\mathrm{i}} . G_{\mathrm{i}-1}) + P_{\mathrm{i}} . P_{\mathrm{i}-1} . C_{\mathrm{i}-1}$

So on comparing with first order equation, we can say that =>

$G^2_{i,i-1} = G^1_i + P^1_i . G^1_{i-1}$ and $P^2_{i,i-1} = P^1_i . P^1_{i-1}$

And also after expressing like these, we observed that $G^2_{i,i-1}$, $P^2_{i,i-1}$ are independent of input carry.

And on clearly observing, we can say that to produce these P and G values, we need 2 first order P and G values of i'th and (i-1)th stage, it means we need 2 bits of A and B to produce these P and G values, so we call it as second order P and G vaues.

So now, expression simplifies to reduced form as =>

$C_{i+1} = G^2_{i,i-1} + P^2_{i,i-1} . C_{i-1}$

Here from above relation, we can say that from C_{i-1} given to us, we can find C_{i+1} directly from it (i.e. gap of 2, that means from carry available at i'th stage, we can find carry of (i+2)th stage.

Now let's move to next higher order, now we again we will take 2nd order equation of carry (i.e. expression containing 2nd order P and G values).

Now let's put (i-2) in place of i, to get value of C_{i-1} .

$C_{i-1} = G^2_{i-2,\ i-3} + P^2_{i-2,\ i-3} \ .\ C_{i-3}$

Now put this value in above equation, we will get = >

$C_{i+1} = G^2_{i,i-1} + P^2_{i,i-1} \ .\ (G^2_{i-2,\ i-3} + P^2_{i-2,\ i-3} \ .\ C_{i-3})$

Now we can see that it is again will become of the form which we have seen quite often now.

$G^3_{i,\ i-3} = G^2_{i,i-1} + P^2_{i,i-1} \ . G^2_{i-2,\ i-3}$ and $P^3_{i,\ i-3} = P^2_{i,i-1} \ .\ P^2_{i-2,\ i-3}$

So, these are called 3rd order P and G values, since it is producing P and G values based on 2nd order P and G values, which in turn uses 1st order values. So, if we see, then we will find that to produce 3rd order P and G values, there is a need of 4 bits of A and B (i.e grouping of 4 bits)

So if we generalize this observation, then we will that the group size over which the carry can be computed directly multiplies by two each time we use a higher order for G and P values. But since we compute higher order P and G values with the help of previous P and G values, then it will add delays. So the time to compute the required higher order G and P values increments by one gate delay. That's why we can say that ultimate time to generate the final carry being logarithmic in the number of bits being added.

Once we calculate higher order P and G values, the final carry can be computed in one step from input carry. What does it means is basically we do not require the internal carries at each bit for final result unlike in case of ripple carry adder, that's why critical carry propagation path in Brent-Kung adder has been significantly reduced in logarithmic style.

And for the sum bits, we just need to do one more step, which is to XOR all first order P's with carry at their stages. So what it means is essentially we need internal carry for the cmpuation of sum bits. So here in Brent-Kung case, output carry is produced of this adder before the sum of this adder.

$S_i = A_i \oplus B_i \oplus C_i = P_i \oplus C_i$

In logarithmic adders, therefore internal bit-wise carry may be available after the final carry. And as we have seen in above discussion that critical path now is not The generation of final carry unlike in the case of ripple carry adder, where we have to wait for the final carry for long time, since rippling is going on in case of ripple carry adder, but the critical path in Brent-Kung adder depends on bit wise sums.

Now let's see the time complexity of Brent-Kung adder. All of the operations which is generation of final carry and final sum result in times which are logarithmic functions of the number of the bits. For wide adders, these can be much faster than other architectures, because if we see the graph of linear curve and logarithmic curve, we can simply observe that log curve slows down for high value of argument on which it is being computed.

Worst case time of generation of final result $\alpha \ \ln(N)$

Now let's see the working of 32-bit Kung adder =>

All the P and G values (first order, second order, etc) are computed in a tree fashion.

1st oder P and G values :

$G_i^1 = A_i \ . \ B_i$ and $P_i^1 = A_i \oplus B_i$

2nd order P and G values:

$G_{i,i-1}^2 = G_i^1 + P_i^1 \ . \ G_{i-1}^1$ and $P_{i,i-1}^2 = P_i^1 \ . \ P_{i-1}^1$

3rd oder P and G values :

$G_{i,\ i-3}^3 = G_{i,i-1}^2 + P_{i,i-1}^2 \ . G_{i-2,\ i-3}^2$ and $P_{i,\ i-3}^3 = P_{i,i-1}^2 \ . \ P_{i-2,\ i-3}^2$

4th order P and G values :

$G^4_{i,\ i-7} = G^3_{i,i-3} + P^3_{i,i-3} \cdot G^3_{i-4,\ i-7}$ and $P^4_{i,\ i-7} = P^3_{i,i-3} \cdot P^3_{i-4,\ i-7}$

5th order P and G values :

$G^5_{i,\ i-15} = G^4_{i,i-7} + P^4_{i,i-7} \cdot G^4_{i-8,\ i-15}$ and $P^5_{i,\ i-15} = P^4_{i,i-7} \cdot P^4_{i-8,\ i-15}$

6th order P and G values :

$G^6_{i,\ i-31} = G^5_{i,i-15} + P^5_{i,i-15} \cdot G^5_{i-16,\ i-31}$ and $P^6_{i,\ i-31} = P^5_{i,i-15} \cdot P^5_{i-16,\ i-31}$

Now, once we have calculated all the required P and G values, we can calculate some of the internal carries directly with

$C_1 = G^1_0 + P^1_0 \cdot C_0$

$C_2 = G^2_{1,0} + P^2_{1,0} \cdot C_0$

$C_4 = G^3_{3,0} + P^3_{3,0} \cdot C_0$

$C_8 = G^4_{7,0} + P^4_{7,0} \cdot C_0$

$C_{16} = G^5_{15,0} + P^5_{15,0} \cdot C_0$

$C_{32} = G^6_{31,0} + P^6_{31,0} \cdot C_0$

Now, once we calculated these carries on the periphery of the Brent-Kung adder, now let's compute rest of the carries =>

C_2 ---- using 1st order P and G -------> C_3

C_4 ----- using 1st order P and G --------> C_5

C_4 ------- using 2nd order P and G -------> C_6

C_6 ----- using 1st order P and G ---------> C_7

C_8 ------- using 1st oder P and G --------> C_9

C_8 ------- using 2nd order P and G -------> C_{10}

C_{10} --------- using 1st order P and G ----------> C_{11}

C_8 ---------- using 3rd order P and G ---------> C_{12}

C_{12} ---------- using 1st order P and G ---------> C_{13}

C_{12} ---------- using 2nd order P and G -------> C_{14}

C_{14} ---------- using 1st order P and G --------> C_{15}

C_{16} ---------- using 1st order P and G ---------> C_{17}

C_{16} ---------- using 2nd order P and G ---------> C_{18}

C_{18} ---------- using 1st order P and G ----------> C_{19}

C_{16} ---------- using 3rd order P and G ----------> C_{20}

C_{20} ---------- using 1st order P and G ----------> C_{21}

C_{20} ---------- using 2nd order P and G ----------> C_{22}

C_{22} ---------- using 1st order P and G ----------> C_{23}

C_{16} ---------- using 4th order P and G ----------> C_{24}

C_{24} ---------- using 1st order P and G ----------> C_{25}

C_{24} ---------- using 2nd order P and G ----------> C_{26}

C_{26} ---------- using 1st order P and G ----------> C_{27}

C_{24} ---------- using 3rd order P and G ----------> C_{28}

C_{28} ---------- using 1st order P and G ----------> C_{29}

C_{28} ---------- using 2nd order P and G ----------> C_{30}

C_{30} ---------- using 1st order P and G ----------> C_{31}

So in this way we build a Brent Kung tree for 32-bit unsigned addition of 2 numbers.

Verilog Implementation

```
/////// Generating 1st order P's and G's signals ////////
assign P1 = A ^ B;
assign G1 = A & B;

//////// Generating 2nd order P's and G's signals ///////
genvar i;
generate
    for(i=0; i<=30; i=i+2) begin: second_stage  //32
        assign G2[i/2] = G1[i+1] | (P1[i+1] & G1[i]);
        assign P2[i/2] = P1[i+1] & P1[i];
    end
endgenerate

/////// Generating 3rd order P's and G's signals //////
generate
    for(i=0; i<=14; i=i+2) begin: third_stage   //16
        assign G3[i/2] = G2[i+1] | (P2[i+1] & G2[i]);
        assign P3[i/2] = P2[i+1] & P2[i];
    end
endgenerate

///////  Generating 4th order P's and G's signals /////
generate
    for(i=0; i<=6; i=i+2) begin: fourth_stage  //8
        assign G4[i/2] = G3[i+1] | (P3[i+1] & G3[i]);
        assign P4[i/2] = P3[i+1] & P3[i];
    end
endgenerate


///////  Generating 5th order P's and G's signals
generate
    for(i=0; i<=2; i=i+2) begin: fifth_stage  //4
        assign G5[i/2] = G4[i+1] | (P4[i+1] & G4[i]);
        assign P5[i/2] = P4[i+1] & P4[i];
    end
endgenerate

//////// Generating 6th order P's and G's signals
assign G6 = G5[1] | (P5[1] & G5[0]);
assign P6 = P5[1] & P5[0];


//////// Generating carry which can be calculated directly from input carry /////
assign C[1] = G1[0] | (P1[0] & Ci);
assign C[2] = G2[0] | (P2[0] & Ci);
assign C[4] = G3[0] | (P3[0] & Ci);
assign C[8] = G4[0] | (P4[0] & Ci);
assign C[16] = G5[0] | (P5[0] & Ci);
assign C[32] = G6 | (P6 & Ci);
```

```
/////// Now generating all carry signals at remaining stages ////////,
assign C[3] = G1[2] | (P1[2] & C[2]);

assign C[5] = G1[4] | (P1[4] & C[4]);
assign C[6] = G2[2] | (P2[2] & C[4]);
assign C[7] = G1[6] | (P1[6] & C[6]);

assign C[9] = G1[8] | (P1[8] & C[8]);
assign C[10] = G2[4] | (P2[4] & C[8]);
assign C[11] = G1[10] | (P1[10] & C[10]);
assign C[12] = G3[2] | (P3[2] & C[8]);
assign C[13] = G1[12] | (P1[12] & C[12]);
assign C[14] = G2[6] | (P2[6] & C[12]);
assign C[15] = G1[14] | (P1[14] & C[14]);
assign C[17] = G1[16] | (P1[16] & C[16]);
assign C[18] = G2[8] | (P2[8] & C[16]);      //2nd order => /2
assign C[19] = G1[18] | (P1[18] & C[18]);
assign C[20] = G3[4] | (P3[4] & C[16]); //3rd order = /4
assign C[21] = G1[20] | (P1[20] & C[20]);
assign C[22] = G2[10] | (P2[10] & C[20]);
assign C[23] = G1[22] | (P1[22] & C[22]);
assign C[24] = G4[2] | (P4[2] & C[16]); //4th order => /8
assign C[25] = G1[24] | (P1[24] & C[24]);
assign C[26] = G2[12] | (P2[12] & C[24]);
assign C[27] = G1[26] | (P1[26] & C[26]);
assign C[28] = G3[6] | (P3[6] & C[24]);
assign C[29] = G1[28] | (P1[28] & C[28]);
assign C[30] = G2[14] | (P2[14] & C[28]);
assign C[31] = G1[30] | (P1[30] & C[30]);

//////////////////////
assign S = P1 ^ {C[31:1],Ci};
assign Co = C[32];
```

Testing of Brent-Kung Adder

Now let's test Brent-Kung adder =>

4 test cases are =>

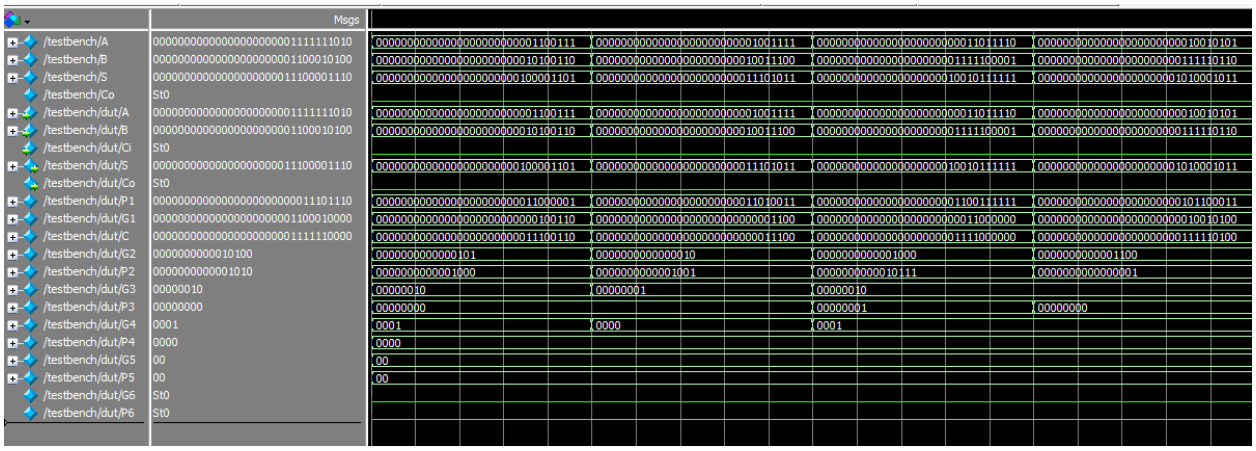
A = 103 & B = 166

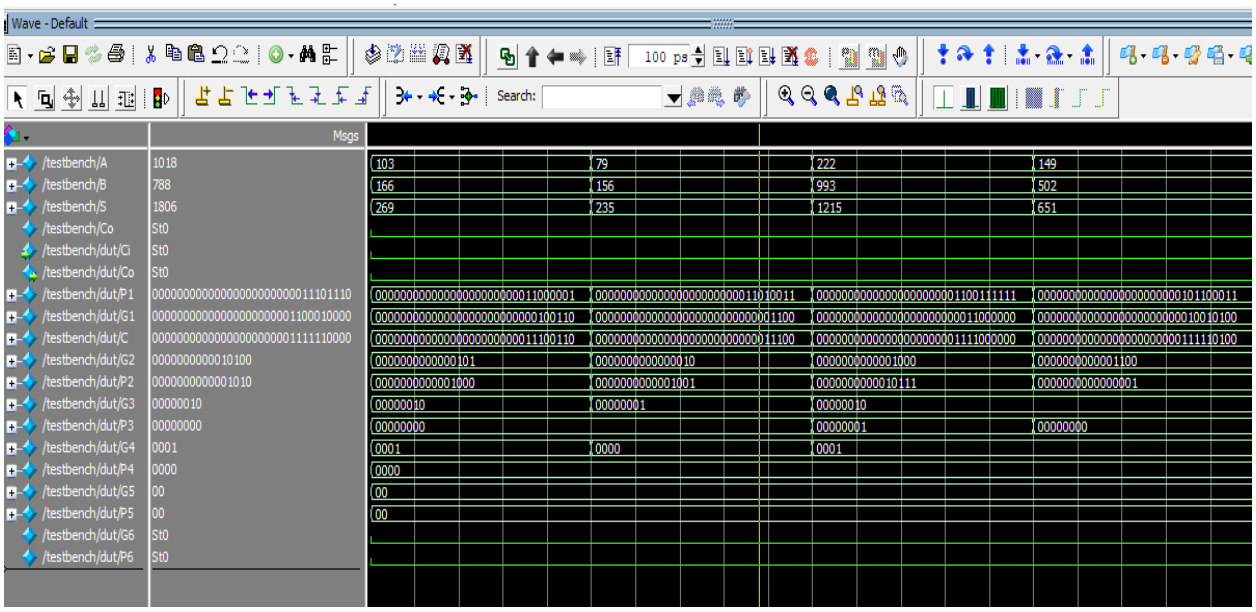
A = 79 & B = 156

A = 222 & B = 993

A = 149 & B = 502

And we assumed input carry to be 0 in all cases, then simulation in ModelSim looks like =>





Worst case delay calculation

Idea here is, if we put Brent-Kung adder which is combinational block between registers, then after doing STA (i.e. static timing analysis) on the circuit, we can find the worst case delay of the adder.

Quartus when doing STA uses various models to estimate delays

Slow 1200mV 85C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	215.05 MHz	215.05 MHz	clk	

And also Quartus is smart enough to analyze all the paths and tell us the potential failing paths =>

Top Failing Paths

	Slack	From	To	Recommendations
1	-3.650	Bt[4]	sum[31]~reg0	Report recommendations for this path
2	-3.648	At[4]	sum[31]~reg0	Report recommendations for this path
3	-3.564	Bt[7]	sum[31]~reg0	Report recommendations for this path
4	-3.561	At[7]	sum[31]~reg0	Report recommendations for this path
5	-3.550	Bt[4]	sum[23]~reg0	Report recommendations for this path

Also best part of Quartus is that it gives all the worst-case timing paths when we do STA on design with Quartus=>

Slack	From Node	To Node	Launch Clock	Latch Clock	Relationship	Clock Skew	Data Delay
-3.650	Bt[4]	sum[31]~reg0	clk	clk	1.000	0.291	4.936
-3.648	At[4]	sum[31]~reg0	clk	clk	1.000	0.291	4.934
-3.564	Bt[7]	sum[31]~reg0	clk	clk	1.000	0.291	4.850
-3.561	At[7]	sum[31]~reg0	clk	clk	1.000	0.291	4.847
-3.550	Bt[4]	sum[23]~reg0	clk	clk	1.000	-0.062	4.483
-3.548	At[4]	sum[23]~reg0	clk	clk	1.000	-0.062	4.481
-3.524	At[1]	sum[31]~reg0	clk	clk	1.000	0.290	4.809
-3.509	At[0]	sum[31]~reg0	clk	clk	1.000	0.290	4.794
-3.491	Bt[5]	sum[31]~reg0	clk	clk	1.000	0.291	4.777
-3.468	Bt[1]	sum[31]~reg0	clk	clk	1.000	0.290	4.753
-3.464	Bt[7]	sum[23]~reg0	clk	clk	1.000	-0.062	4.397
-3.461	At[7]	sum[23]~reg0	clk	clk	1.000	-0.062	4.394
-3.455	Bt[6]	sum[31]~reg0	clk	clk	1.000	0.291	4.741
-3.424	At[1]	sum[23]~reg0	clk	clk	1.000	-0.063	4.356

-3.423	At[2]	sum[31]~reg0	clk	clk	1.000	0.290	4.708
-3.411	Bt[2]	sum[31]~reg0	clk	clk	1.000	0.290	4.696
-3.409	At[0]	sum[23]~reg0	clk	clk	1.000	-0.063	4.341
-3.391	Bt[5]	sum[23]~reg0	clk	clk	1.000	-0.062	4.324
-3.376	At[12]	sum[31]~reg0	clk	clk	1.000	0.290	4.661
-3.374	At[5]	sum[31]~reg0	clk	clk	1.000	0.291	4.660
-3.368	Bt[1]	sum[23]~reg0	clk	clk	1.000	-0.063	4.300
-3.355	Bt[6]	sum[23]~reg0	clk	clk	1.000	-0.062	4.288
-3.355	At[9]	sum[31]~reg0	clk	clk	1.000	0.289	4.639
-3.351	At[8]	sum[31]~reg0	clk	clk	1.000	0.289	4.635
-3.350	At[13]	sum[31]~reg0	clk	clk	1.000	0.290	4.635
-3.337	Bt[4]	sum[30]~reg0	clk	clk	1.000	0.291	4.623
-3.335	At[4]	sum[30]~reg0	clk	clk	1.000	0.291	4.621
-3.330	Bt[4]	sum[29]~reg0	clk	clk	1.000	0.291	4.616
-3.328	At[4]	sum[29]~reg0	clk	clk	1.000	0.291	4.614
-3.323	At[2]	sum[23]~reg0	clk	clk	1.000	-0.063	4.255
-3.312	Bt[4]	sum[25]~reg0	clk	clk	1.000	0.290	4.597
-3.311	Bt[2]	sum[23]~reg0	clk	clk	1.000	-0.063	4.243
-3.310	At[4]	sum[25]~reg0	clk	clk	1.000	0.290	4.595
-3.286	At[6]	sum[31]~reg0	clk	clk	1.000	0.291	4.572
-3.279	At[3]	sum[31]~reg0	clk	clk	1.000	0.290	4.564
-3.276	At[12]	sum[23]~reg0	clk	clk	1.000	-0.063	4.208
-3.274	At[5]	sum[23]~reg0	clk	clk	1.000	-0.062	4.207
-3.262	Bt[4]	sum[11]~reg0	clk	clk	1.000	-0.062	4.195
-3.260	At[4]	sum[11]~reg0	clk	clk	1.000	-0.062	4.193
-3.256	Bt[0]	sum[31]~reg0	clk	clk	1.000	0.290	4.541
-3.255	At[9]	sum[23]~reg0	clk	clk	1.000	-0.064	4.186
-3.251	Bt[7]	sum[30]~reg0	clk	clk	1.000	0.291	4.537

-3.251	At[8]	sum[23]~reg0	clk	clk	1.000	-0.064	4.182
-3.250	At[13]	sum[23]~reg0	clk	clk	1.000	-0.063	4.182
-3.248	At[7]	sum[30]~reg0	clk	clk	1.000	0.291	4.534
-3.245	Bt[14]	sum[31]~reg0	clk	clk	1.000	0.290	4.530
-3.244	Bt[7]	sum[29]~reg0	clk	clk	1.000	0.291	4.530
-3.241	At[7]	sum[29]~reg0	clk	clk	1.000	0.291	4.527
-3.226	Bt[7]	sum[25]~reg0	clk	clk	1.000	0.290	4.511
-3.223	At[7]	sum[25]~reg0	clk	clk	1.000	0.290	4.508
-3.213	Bt[12]	sum[31]~reg0	clk	clk	1.000	0.290	4.498
-3.211	At[1]	sum[30]~reg0	clk	clk	1.000	0.290	4.496
-3.204	At[1]	sum[29]~reg0	clk	clk	1.000	0.290	4.489
-3.196	At[0]	sum[30]~reg0	clk	clk	1.000	0.290	4.481
-3.189	At[0]	sum[29]~reg0	clk	clk	1.000	0.290	4.474
-3.186	At[6]	sum[23]~reg0	clk	clk	1.000	-0.062	4.119
-3.186	At[1]	sum[25]~reg0	clk	clk	1.000	0.289	4.470
-3.181	At[10]	sum[31]~reg0	clk	clk	1.000	0.289	4.465
-3.179	At[3]	sum[23]~reg0	clk	clk	1.000	-0.063	4.111
-3.178	Bt[5]	sum[30]~reg0	clk	clk	1.000	0.291	4.464
-3.176	Bt[7]	sum[11]~reg0	clk	clk	1.000	-0.062	4.109
-3.173	At[7]	sum[11]~reg0	clk	clk	1.000	-0.062	4.106
-3.171	Bt[5]	sum[29]~reg0	clk	clk	1.000	0.291	4.457
-3.171	At[0]	sum[25]~reg0	clk	clk	1.000	0.289	4.455
-3.169	At[11]	sum[31]~reg0	clk	clk	1.000	0.289	4.453
-3.163	Bt[10]	sum[31]~reg0	clk	clk	1.000	0.289	4.447
-3.156	Bt[0]	sum[23]~reg0	clk	clk	1.000	-0.063	4.088
-3.155	Bt[1]	sum[30]~reg0	clk	clk	1.000	0.290	4.440
-3.154	Bt[4]	co~reg0	clk	clk	1.000	0.291	4.440
-3.153	Bt[4]	sum[27]~reg0	clk	clk	1.000	0.290	4.438

-3.153	Bt[5]	sum[25]~reg0	clk	clk	1.000	0.290	4.438
-3.152	At[4]	co~reg0	clk	clk	1.000	0.291	4.438
-3.151	At[4]	sum[27]~reg0	clk	clk	1.000	0.290	4.436
-3.148	Bt[1]	sum[29]~reg0	clk	clk	1.000	0.290	4.433
-3.145	Bt[14]	sum[23]~reg0	clk	clk	1.000	-0.063	4.077
-3.142	Bt[6]	sum[30]~reg0	clk	clk	1.000	0.291	4.428
-3.141	Bt[3]	sum[31]~reg0	clk	clk	1.000	0.290	4.426
-3.136	At[1]	sum[11]~reg0	clk	clk	1.000	-0.063	4.068
-3.136	Bt[8]	sum[31]~reg0	clk	clk	1.000	0.289	4.420
-3.135	Bt[6]	sum[29]~reg0	clk	clk	1.000	0.291	4.421
-3.130	Bt[1]	sum[25]~reg0	clk	clk	1.000	0.289	4.414
-3.129	At[15]	sum[31]~reg0	clk	clk	1.000	0.290	4.414
-3.127	Bt[1]	sum[27]~reg0	clk	clk	1.000	0.289	4.411
-3.123	Bt[1]	sum[15]~reg0	clk	clk	1.000	-0.064	4.054
-3.121	At[0]	sum[11]~reg0	clk	clk	1.000	-0.063	4.053
-3.117	Bt[6]	sum[25]~reg0	clk	clk	1.000	0.290	4.402
-3.113	Bt[12]	sum[23]~reg0	clk	clk	1.000	-0.063	4.045
-3.110	At[2]	sum[30]~reg0	clk	clk	1.000	0.290	4.395
-3.103	Bt[5]	sum[11]~reg0	clk	clk	1.000	-0.062	4.036
-3.103	At[2]	sum[29]~reg0	clk	clk	1.000	0.290	4.388
-3.098	Bt[2]	sum[30]~reg0	clk	clk	1.000	0.290	4.383
-3.091	Bt[2]	sum[29]~reg0	clk	clk	1.000	0.290	4.376
-3.085	At[2]	sum[25]~reg0	clk	clk	1.000	0.289	4.369
-3.081	At[10]	sum[23]~reg0	clk	clk	1.000	-0.064	4.012
-3.073	At[14]	sum[31]~reg0	clk	clk	1.000	0.290	4.358
-3.073	Bt[2]	sum[25]~reg0	clk	clk	1.000	0.289	4.357
-3.072	Bt[13]	sum[31]~reg0	clk	clk	1.000	0.290	4.357
-3.069	At[11]	sum[23]~reg0	clk	clk	1.000	-0.064	4.000
-3.068	Bt[7]	co~reg0	clk	clk	1.000	0.291	4.354
-3.067	Bt[6]	sum[11]~reg0	clk	clk	1.000	-0.062	4.000

What we have done is simply added some registers on input side and output side to have sequential kind of design (intentionally it is not), so now Quartus can do STA i.e static timing analysis. What it do in STA is it look for all reg-to-reg paths, and find maximum and minimum delay based on the information it has regarding standard cells and also use characteristics of registers i.e. setup and hold time, and find the maximum frequency which satisfy the setup and hold constraints. Now after all these analysis, what we get is max frequency, which we look it in other way, then we also get time period which directly corresponds to critical path delay. That's how we calculate critical path delay in any combinational logic.

Conclusion

We have successfully designed and implemented the 32-bit Brent Kung Adder in Verilog and simulated in ModelSim and also use Quartus to do STA on the design after adding ff/registers in the input and output side of Brent-Kung design to find the worst case delay in this adder.