Area-Time Optimal Adder with Relative Placement Generator

Abstract:

This paper presents the design of a generator, for the production of area-time-optimal adders. A unique feature of this generator is that, it integrates synthesis and layout by providing relative placement information.. Relative placement information provides better support for structured layout and easier integration flow with data-path placer. Adders generated using the proposed generator are dynamically configured for a given technology library, wire-load model, delay, and area goal. Adders of sizes 1 to 1024 bits are produced. The adder architecture used in this generator is a hybrid of Brent & Kung, carry select, and ripple carry adders. When compared with Synopsys' fast adders, a 20-50% reduction in area with comparable delays are produced. This generator has been integrated into Synopsys high-performance datapath design tool Module Compiler.

1. Introduction

Addition is an important operation affecting the speed and performance of digital signal processors. High-speed adders can be realized using most widely used carry look-ahead (CLA) [1], carry select [2], or binary carry look-ahead [3,4] techniques. Due to time-to-market constraints in VLSI industry, design of, high-speed area efficient circuits in minimal time is a major challenge. There are many full custom designed adders [][][][][][], which offer very high speed and low area for a specific process technology and complex circuit techniques. Due to the high cost of full custom adders, standard cell based designs are widely used [][][][][][] but the performance of these adders is also limited to a specific cell library and design constraints.

There is another class of adders, which are produced using software programs called generators [][][][]. Using generators, adders can be generated for a variety of design goals, bit-widths, and technologies. First attempt to produce area-time optimal adders using software programs was reported by xx in []. The proposed adder architecture was based on Ladner and Fischer's parallel prefix computation model, which is essentially a look-ahead addition. The design is formulated as a dynamic programming problem and optimized for area and delay. In [], xxx propose a multi-dimensional programming paradigm to control the block sizes of carry skip adders and CLA for minimizing the delay. In the generator proposed in [], conditional sum adders which include testability features are generated. All the above generators have a major drawback that they did not consider the wiring effects. With the shrinking VLSI dimensions, especially today's deep sub-micron technologies interconnect delays play a dominant role in overall performance of the circuit. Moreover the digital design paradigm has moved from logic domain to physical domain, none of the above mentioned generators provide any information for the placement or tiling of adder cells.

In this paper we present an efficient algorithm to generate area-time optimal adders, with relative placement (RP) information. Relative Placement (RP) assigns an instance/gate to a row and column position. RP information is used by the placement tool for fast,

efficient and structured placement of instances in layout. The adders generated are targeted for low area, with speeds comparable to Synopsys' fast adders. The proposed generator takes into consideration the wire-load models along with other parameters such as delay, area, and operating conditions (temperature and voltage). The proposed generator that generates adders upto 1024 bits has been integrated with Synopsys' datapath synthesis tool, which allows further enhancement such as, pipelining, retiming, GUI interface, and adder instantiation in a Verilog like language called MCL (module compiler language).

The adder architecture used in this generator is a hybrid of Brent & Kung [i, ii], carry select [], and ripple carry adders. In order to achieve low area with high-speed, area optimized ripple carry adders are used along with carry select adders (CSA) and carry generation logic. The whole adder is composed of 4-bit ripple carry and/or carry select adder blocks. The 4-bit ripple carry adder blocks are used for fast arriving carry signals (in the beginning of the adder) and CSA are used for slow arriving signals. The carry generation logic (carry chain) has been implemented using the '*o*' operator as described by xxxx in [i, ii] with a modification that four-bit groups are used, instead of two bits. By using four-bit groups, the number of wires and hardware is reduced by *1/2*, keeping the adder delay proportional to *O(log n)* [iii] (where '*n*' is the number of bits of the adder) Also, timing analysis is used to equalize the arrival of the entire sum and reduce adder hardware by inserting/using maximum number of ripple carry adders. The adders produced by the generator are similar in topology to [iii, iv, v] but due to programming area-delay optimal adders are produced using different technology libraries, wire-load models, area-delay goals.

The paper is organized as the following: In Section 2 the architecture of the proposed adder is described. In Section 3 the generator is described in detail. In Section 4 the performance comparison of the adders generated and RP layout of a 64-bit adder using the proposed generator is presented. Finally Section 5 concludes this paper.

2. Adder Architecture:

Let, $A = a_{n-1}, a_{n-2} ..... a_1, a_0$, and $B = b_{n-1}, b_{n-2}, ... b_1, b_0$ be the two input operands, with a_{n-1} and b_{n-1} be the most significant bits. The generate and propagate signal at bit position "*i*" are given by; $g_i = a_i \quad b_i$, and $p_i = a_i \quad b_i$,(where: = AND operation and = XOR operation). The Carry out from bit position "*i*" is given by; $C_i = g_i + g_{i-} \quad p_i$ (where + = OR operation) provided $C_0 = 0$. The "o" operator as defined by [3] is given as follows:

(g, p) o (g', p') = (g + (p g'), p p') (1)

The group Generate (G) and Propagate (P) are given by:

$(G_i, P_i) = (g_0, p_0)$ if i =0 &

(g_i, p_i) o (g_{i-1}, p_{i-1}) if 0 < i < n (2)

In [3,4,5], using (1), the generate and propagate signals for each level (*k*) of the adder are generated using the following combination:

$(G_{i+2k}, P_{i+2k}) = (g_{i+2k}, p_{i+2k})$ o (g_i, p_i) for $0 < k < \log n$ (3)

In the proposed implementation for '*n*' bits, at $k = 0$ (first level) $n/2$ generate and propagate signals are produced using the following combination:

$(G_{2i+1}, P_{2i+1}) = (g_{2i+1}, p_{2i+1}) \circ (g_{2i}, p_{2i})$ for $0 < i < n/2$ (4)

At the second level n/4 signals are produced (by grouping the signals generated at the first level) using (4) but limiting i *to* $n/4$. These signals are the four-bit group generate and propagate signals, their value for 4-bit case is given below, and their grouping is shown in Fig. 1.

$(g_{10}, p_{10}) = (g_1, p_1) \circ (g_0, p_0)$ and

$(g_{32}, p_{32}) = (g_3, p_3) \circ (g_2, p_2)$ at k = 0 (at first level) (5)

$(G_{30}, P_{30}) = (g_{32}, p_{32}) \circ (g_{10}, p_{10})$ (at second level) (6)

In this realization no (g_2, p_2) or intermediate even carry is generated, because these are generated within the conditional sum adders. Once we have the 4-bit group carries, the carries in multiplies of 4 are generated using (2). This technique results in minimum wiring and area, for *n* bits, approximately $2n/2^k$ signals are generated at each level of the adder in contrast to [3, 5] which requires $2(n-2^k)$ signals. The wiring and area of Brent's adder [3,5] increases exponentially with the increase in the number of bits. While using four-bit groups offer linear increase in wiring and area, keeping the delay equivalent to Brent's adder [].

Fig. 1 shows the block diagram of a 32-bit adder. As shown in this figure, the carry tree generates the carry inputs in multiples of four. The carry tree is made up of 3-input AND-OR and 2-input AND gates, implementing $g_{i.i-1} = g_i + g_{i-1} \bullet p_i$ and $p_{i.i-1} = p_{i-1} \bullet p_i$ (filled circle) respectively, at each level of the tree. Since inverted logic is faster then non-inverted logic therefore alternate polarities of g and p are produced at each level. The carries generated by the tree are then used to generate the final sum using either a ripple carry adder or a CSA. In the following sections we explain the design of the area-delay-optimized ripple and CSA adders.

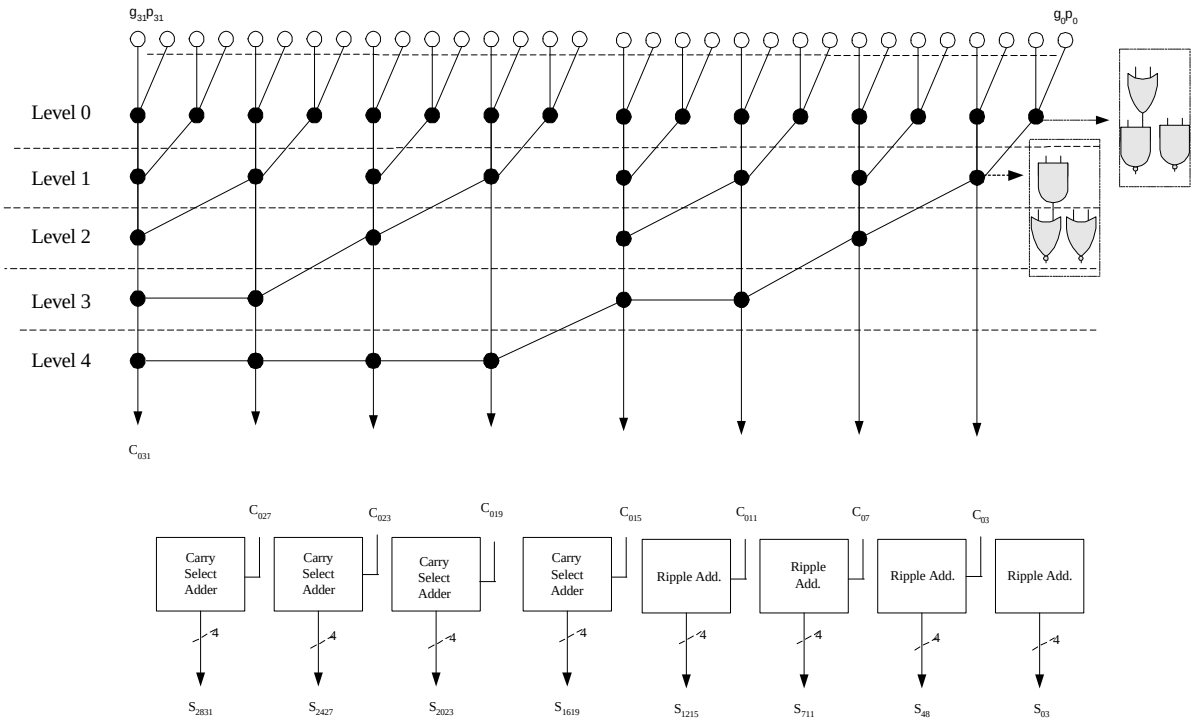

Fig. 1. 32-Bit adder block diagram.

Basic Cells

Fig. 2 shows the area optimized ripple carry adder. The area of the adder is optimized by utilizing generate (*g*) and propagate (*p*) signals produced in the carry tree. This adder requires only 9-gates (counting shaded AND-OR gates as one gate) for 4-bits.

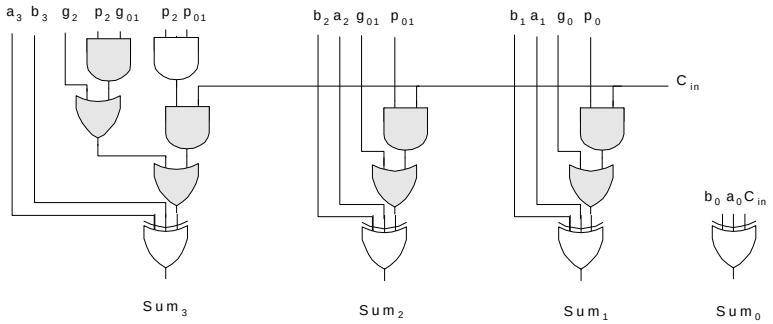

Fig. 2. Four bit area optimised ripple carry adder.

Since, the four bit carry select adders are not on the critical path, they could be designed using two sets of four bit ripple carry adders, with $C_{in} = 0$ for one set , and $C_{in} = 1$ for the other. However, we have found that it is possible to reduce the hardware by merging the two adders together.

Fig. 3 shows the schematic diagram of the 4-bit merged carry select adder (CSA). The hardware of this merged CSA is approximately 40% smaller then the hardware required by two separate four-bit ripple carry adders. In VLSI implementation, the 4-bit CSA is combined with the 4-bit carry generate circuit.

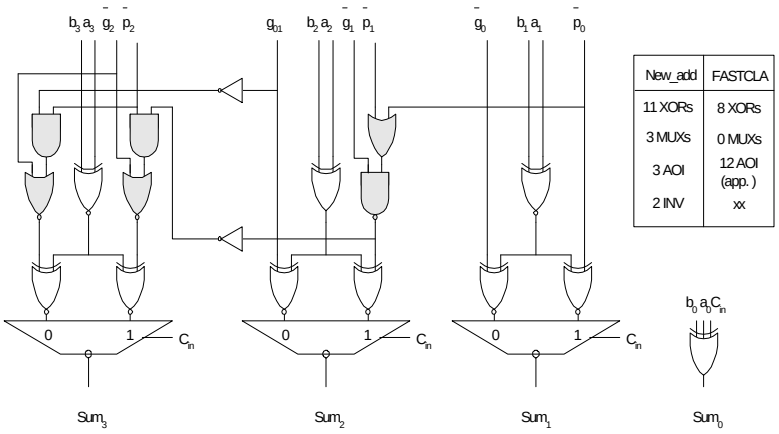

Fig. 3. Four bits carry select adder.

3. Generator

In this section we present the generator architecture. The generator is described in C++ in a modular fashion and contains the following modules:

1. A Logic generator.
2. A Delay calculator.
3. An Adder optimizer.

Functions, such as, reading the technology library, parsing MCL input, adder parameters (bitwidth, delay, area goals), and GUI interface are performed by the module compiler (MC) synthesis engine. Below we explain the design of the individual generator modules.

Logic generator

The main function of the logic generator is to instantiate, place, and interconnect logic cells and generate relative placement information. The MC synthesis engine parses the input and then passes a data structure to the logic generator. This data structure contains information regarding the adder type, number of bits, and pointers to input/output operands, desired area and delay.

Logic generator creates a data structure containing '*n*' link lists (columns) containing input operand bits and their associated delay in ascending order. After this the generator performs following operation in the same order: Each instantiation of the gate updates RP data structure with bit/column number and the level of the adder. The nets produced following this operation are again placed in the link list, which is once again sorted for delays.

1. Generate the GP terms for each bit using NAND and NOR gates.
2. Generate a binary tree using equation 6, instantiate AND-OR and AND gate for each node. Since inverted logic is faster then non-inverted logic therefore alternate polarities of g and p are produced at each level of the GP tree, in order to get full advantage of inverting logic cells.

3. Generate intermediate GP terms, which are not power of 2.
4. Instantiate the first ripple carry adder shown in Fig.2 for the first four bits.
5. Instantiate CSA adders (Fig.3) for the rest of the bits.

Adder optimizer

Can you please explain this in words??

```
/*------------------------------------------------------------
 Begin adder optimization
 Determine the maximum delay the Adder
------------------------------------------------------------*/
        int delay[BigMaxBits];
        int maxDelay = 0;
        int maxGroupDelay = 0;
        for (i=4; i < origBits; i++)
        {
                delay[i] = gp_tree[MaxLevel][i].delay;
                if (delay[i] > maxDelay)
                        maxDelay = delay[i];
        }

/*------------------------------------------------------------
        Now Start adder optimization (why can't this be merged above?)
        Now Replace the CSA adders
------------------------------------------------------------*/
        //fast car_in path, area optimized ripple adder
        for (i=4; i<origBits; i = (i + 4))
        {
             if ( origBits-i < 4)
                        groupWidth =  origBits-i-1;
             else
                        groupWidth =  3;
             reduceCol=AdddbFALSE;
             GenerateSumRip(inp,i,(claLog2(i-1)),groupWidth,
reduceCol);
             for (j=i; j < i+1+groupWidth; j++)
             {
                        delay[j] =  gp_tree[MaxLevel][j].delay;
                        if (delay[j] > maxGroupDelay)
                                maxGroupDelay = delay[j];
             }
             reduceCol=AdddbTRUE;
             if(maxDelay > (maxGroupDelay+5))
             GenerateSumRip(inp,i,(claLog2(i-1)),groupWidth,
reduceCol);
                else
             GenerateSumCSA(inp,i,(claLog2(i-1)),groupWidth,
reduceCol);

        }
 }
```

Delay calculator

The delay calculator module provides detailed timing calculation information about the specified cell or net timing arc. Both, the operating conditions, and wire load models are taken into account when making delay calculations. For CMOS models, the delay is broken into four parts: intrinsic delay (D_i), transition delay (D_t), slope delay (D_s), connect delay (D_c). The total delay is therefore given by:

$$D_{total} = D_i + D_t + D_s + D_c.$$

The intrinsic delay D_i is the delay through the element, which is independent of the load (also known as the base delay).

The transition delay D_t is a function of the driver resistance and the capacitance of the wire and the pin. $D_t = R_{driver}(C_{wire} + C_{pins})$ where R_{driver} is the rise or fall resistance (also known as the Load Factor)

The slope delay D_s is the input to output delay due to the transition time on the input. The equation for this is $D_s = S_s * D_t$ (previous stage); where S_s is the rise slope or fall slope.

The connect delay D_c is a rise or fall delay, also known as the time-of-flight delay or the time it takes for a waveform to propagate along a wire and is based on one of three types of models:

Worst Case RC generic CMOS tree $D_c = R_{wire}(C_{wire} + sum(C_{load\_pins}))$

Worst Case RC CMOS2 tree $D_c = (R_{wire}/N) * (C_{wire} + sum (C_{load\_pins}))$

The above models the case where the load pin is at the extreme end of the wire.

Balanced RC tree $D_c = (R_{wire}/N) * (C_{wire}/N + sum (C_{load\_pins}))$

The above models the case where all load pins are on separate, equal branches of the interconnect wire. Each load pin incurs an equal portion of the wire capacitance.

Best Case RC tree $Dc = R_{wire}(C_{wire} + C_{pin}) = 0$

Models the case where the load pin is physically adjacent to the driver.

The wire resistance R_{wire} is determined from a wire load model. The length of the net is computed from a global estimation function, which is based on the number of fanouts for that net.

The wire capacitance C_{wire} is determined for the head of the timing arc. The length is computed using the actual number of fanout pins on the net and the fanout_lengths defined in the wire_load group in the library. If no wire_load group is defined, then C_{wire} is 0. The pin capacitance C_{pin} is the library capacitance defined in the pin group for the load pin.

For a cell arc, the pin names are associated with an input pin and an output pin for the same cell. For a net arc, the pin names are for a driver pin to a load pin on the same net.

Calculating the Driving Cell Delay

Assuming nonlinear delay library modeling for the specified timing arc (that is, libcell-input-pin to libcell-output-pin timing relationship), the cell delay is a function of both the transition time on the input pin and the capacitive load on the output pin. Delay calculation uses the first timing arc it encounters for the driving cells specified output pin, and it uses zero for the input transition time.

The total capacitive load seen by the driving cells output pin is the sum of the wire capacitances and the pin capacitances that it drives. These capacitances can exist both externally and internally to an input port. Both external wire capacitance and external pin capacitance contribute to the load seen by the driving cell. Internal capacitances are modeled in four ways:

- The wire load model calculation of the nets wire capacitance based on the number of loads it drives
- The net's wire capacitance annotated on pins or ports through set_rtl_load (for details, see the man page for set_rtl_load) (what man page?)
- The net's pin capacitance for each leaf cell input pin it drives
- The nets wire capacitance annotated by using set_load (this capacitance

The conventional approach to calculate pre-layout wire load during synthesis, in an ASIC environment, has been a specification of a statistical wire load model based on total die size and fanout capacitance. For example, assume you have the following values:

C_{pin} - pin capacitance external to input port.

C_{wire} - wire capacitance external to input port.

C_{wlm} - calculated wire capacitance internal to input port based upon wire load model

C_{fan} - pin capacitance internal to input port based upon individual library cell input pin descriptions

The total capacitive load seen by the output pin of our driving cell would be

$$C_{total} = (C_{pin} + C_{wire}) + (C_{wlm} + C_{fan})$$

Calculating the Interconnect Delay External to the Input Port

The wire delay external to an input port is the product of the external wire resistance and all capacitance seen after the external pin capacitance. The external and internal capacitances used here are modeled in the same manner as described above in the calculation of the driving cell delay. The difference, here, is that the external pin capacitance does not contribute to the external wire delay. Only the external wire capacitance and all internal capacitances contribute to this delay.

For example, assume we have the following values

R_{ip} - resistance value external to input port

C_{pin} - pin capacitance external to input port

C_{wire} - wire capacitance external to input port

R_{wlm} - calculated wire resistance internal to input port based on the wire load model

C_{wlm} - calculated wire capacitance internal to input port based on wire load model

C_{fan} - pin capacitance internal to input port based on individual library cell input pin descriptions

The wire delay external to an input port would be

$D_{ext} = R_{ip} * (C_{wire} + C_{wlm} + C_{fan})$

The total wire delay both external and internal to an input port would be

$D_{total} = D_{ext} + (R_{wlm} * (C_{wire} + C_{wlm} + C_{fan}))$

Note that Cpin does not affect the wire delay. However, it still presents an additional load to the driving cell and, therefore, affects the driving cells delay as seen above.

4. Performance Comparison

The performance of the adders synthesized by the generator are compared with the existing Module Compiler (MC) FASTCLA [], and Design Ware (DW) BK adder []. All the MC adders are synthesized using MC and then post processed in Design Compiler (DC), while the DW adders were synthesized using only DC. Each adder is synthesized using ten different libraries worst case operating conditions and wire load models. Fig. 4 and Fig. 5 shows the delay and area comparison between different adders using IBM_CMOS6S_SC library. It is clear from the synthesis results that in terms of delay adders generated by the proposed generator are comparable (good in some cases and slightly slow in some cases) to SYNOPSYs' best adders (DW_BK, and MC_FASTCLA). While, in terms of area adders generated by the proposed generator are always 20-50% better.

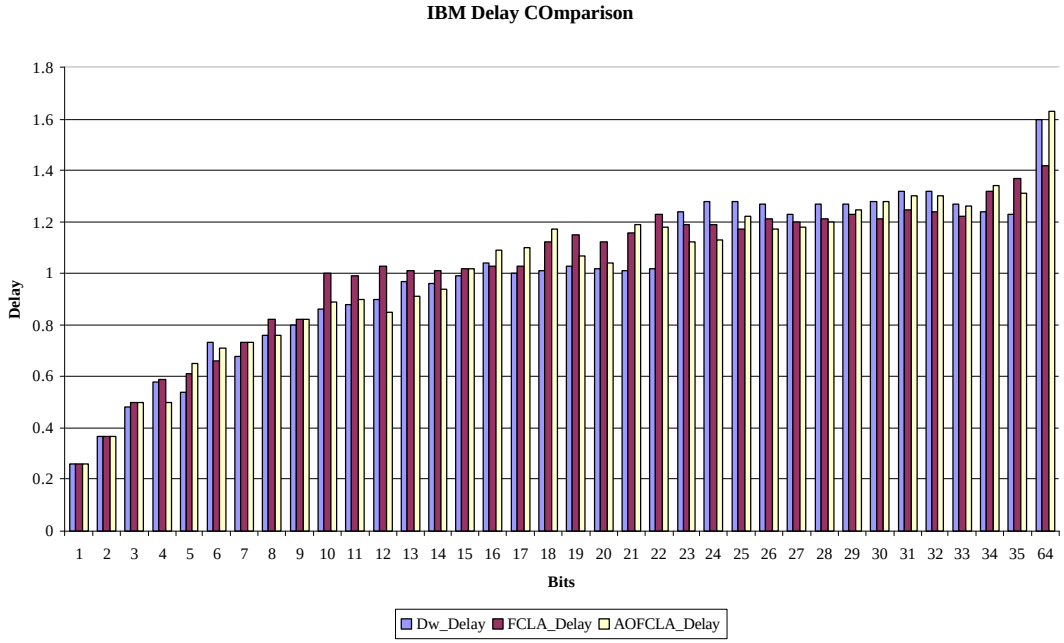

Fig. 4. Delay comparison of DW_BK, MC_FCLA, and NEW adder in IBM 0.18u Tech.

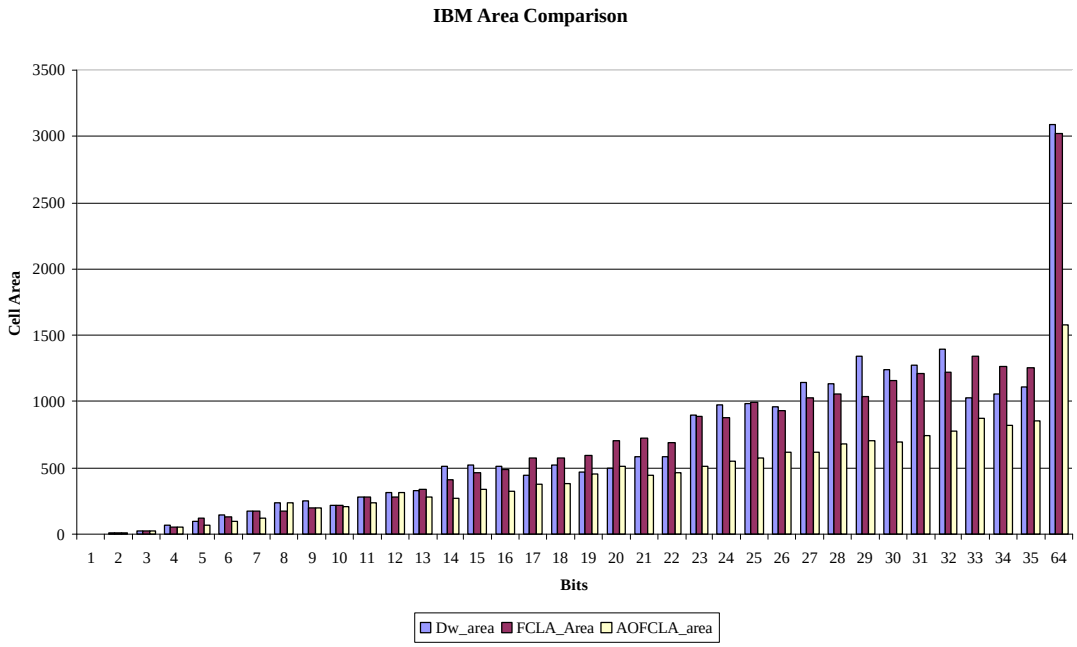

Fig. 5. Area comparison of DW_BK, MC_FCLA, and NEW adder in IBM 0.18u Tech.

5. Conclusions

Reference:

i. Brent R., "On the addition of binary numbers", IEEE Trans. Computers, C-19, 758 (1970).

ii. Brent, R.P.; Kung, H.T., “A regular layout for parallel adders”, IEEE Transactions on Computers, vol.C-31, (no.3), March 1982. p.260-4.

iii. A. A Farooqui, V. G. Oklobdzija, F. Chehrazi, "Multiplexer based adder for media signal processing", Proceedings of the 1999 International Symposium on VLSI Technology, Systems, and Applications, Taipei, Taiwan, R.O.C., Page(s): 100 -103, 8-10 June 1999.

iv. A. A Farooqui, V. G. Oklobdzija, F. Chehrazi, "64-Bit Media Adder", Proceedings of the 1999 IEEE International Symposium on Circuits and Systems, (ISCAS '99), Orlando, Florida, USA, 30 May-2 June 1999.

v. A. A Farooqui, V. G. Oklobdzija, F. Chehrazi, "A Mutliplexer Based Parallel n-Bit Adder Circuit for High Speed Processing”, SONY-50N3106, filed March 1999.