A 1.0ns 64-bits GaAs Adder using Quad tree algorithm

Philippe ROYANNEZ
Laboratoire MASI/CAO-VLSI/UPMC
4, place Jussieu, 75252 Paris, FRANCE
Philippe.ROYANNEZ@masi.ibp.fr

Amara AMARA
Laboratoire de Microelectronique
21 Rue d’Assas, 75006 Paris, FRANCE
Amara.Amara@isep.fr

Abstract

This paper describes a full custom 64-bits adder targeting the VITESSE E/D MESFET process HGaAsIII. This adder which respects a bit slice topology is part of the project of GaAs data-path compiler for ALLIANCE CAD TOOLS. GaAs’s specific properties have been exploited in a full custom approach. Original architecture have been used to increase the parallelism of carries’ computation. The layout is portable using a symbolic approach and could also be used with other E/D MESFET process.

1. Introduction

High speed adders are very common elements both in microprocessors and DSP devices. They have to be as fast as possible for a minimal cost in term of area and power consumption. A good compromise could be achieved by a full custom approach using a Carry Lookahead Adder algorithm [7] which provides better layout density and delay performance.

Adders are often part of Arithmetic and Logic Unit. They are mostly integrated in a Data-Path structure with other vectorized operators. Based on these element’s, a bit-slice topology was chosen. Some physical constraints have been taken to insure the compatibility with an over-cell data-path router for CMOS developed at MASI laboratory [1].

High speed GaAs DCFL digital design arizes some specific limitations such as pass-gates and complex gates implementation [4]. This paper presents firstly the algorithm analysis adapted to this kind of circuits. Then, it details the description of the physical design and topologic problems involved. Finally, it deals with the design of a dedicated library of leaf cells in order to use an assembling software and concludes with the validation procedure and performances.

2. Algorithm

NOR gates are basic elements in DCFL logic. More precisely, NAND gates and PASS-GATE should be avoided for

\begin{align*}
\text{For the input stage:} & \quad \left\{ \begin{array}{l}
p_i = a_i \oplus b_i = P_i, \\
g_i = a_i b_i = G_i,
\end{array} \right. \\
\text{For propagation:} & \quad \left\{ \begin{array}{l}
P_{i,k} = P_{i,j} P_{j+1,k}, \\
G_{i,k} = G_{i,j} P_{j+1,k} + G_{j+1,k},
\end{array} \right. \\
\text{For the output stage:} & \quad s_i = p_i \oplus G_{0,i-1}
\end{align*}

Where $a_i$ and $b_i$ are the bits from the two operands. Intermediate carries are $C_i = G_{i,j-1}$ and $C_{out} = G_{0,N-1}$. Carry propagation can be seen as a binary tree (see figure 1).

For a CMOS implementation we take great advantage of using complex gates like "xor", "and", "OrAnd" which is not the case with DCFL logic. On the other hand these gates have low Fan-In (two or three). These two reasons suggest a modified algorithm. To increase the parallelism of the carry computation using gates with great Fan-In, we implement quad trees instead of binary ones (see figure 2).

This technic decreases the depth in term of number of gates and by the way provides higher speed computation. More precisely the algorithm is an $O(\log_4 N)$ while the BCLA is an $O(\log_2(N))$ where $N$ is the operand size. For
Figure 2. Carry out propagation in an 8-bits Quad CLA

N=8 figure 1 and 2 show that binary tree is deeper than quad tree.

Figure 3. 6th Carry out propagation in an 8-bits Quad CLA

However we must notice two important points. On one hand there is no need to increase parallelism of the carry out computation if we don’t increase the whole intermediate carries computation at the same rate. This idea suggests to use quad trees for each carry \( G_{0,i-1} \). But on the other hand it would be more judicious to avoid speeding up signals which are not on the critical path. Consequently we use both binary, ternary and quad convergences to provide best compromise between delay and power consumption. This solution provides customized trees for each carry computation. Figure 3 gives an example for \( G_{0,6} \). This tree includes three kinds of convergences.

Figure 4. Carry computation cells

This means that we need three types of cells (see figure 4) which logical equations follow:

- **PG2:** \[
P_{i,k} = P_{i,j}P_{j+1,k}, \quad G_{i,k} = G_{i,j}P_{j+1,k} + G_{j+1,k}
\]

- **PG3:** \[
P_{i,l} = P_{i,j}P_{j+1,k}P_{k+1,l}, \quad G_{i,l} = G_{i,j}P_{j+1,k}P_{k+1,l} + G_{j+1,k}P_{k+1,l} + G_{k+1,l}
\]

- **PG4:** \[
P_{i,m} = P_{i,j}P_{j+1,k}P_{k+1,l}P_{l+1,m}, \quad G_{i,m} = G_{i,j}P_{j+1,k}P_{k+1,l}P_{l+1,m} + G_{j+1,k}P_{k+1,l}P_{l+1,m} + G_{k+1,l}P_{l+1,m} + G_{l+1,m}
\]

Of course the input and output stages are the same as in the BCLA (see figure 5). The cost of this modified algorithm appears in the higher complexity of boolean equations. However, this complexity can be reduced by using four inputs NOR gates without important loss in delay performance. So, this algorithm does not increase only the parallelism of carries computation but also uses the advantages of the DCFL logic.

3. Logical structure

![Logical Structure Diagram](image)

Figure 6. Cells schematics

Implementation of those logical functions uses up to four inputs NOR gates and two inputs OR. To optimise the delay performances we use negative polarity for the whole propagate and generate signals. Figure 6 shows cells used in the quad adder. An example of an 8-bits implementation at cells level is shown on figure 7. This structure is easily extendable to 64 bits (see next section).

For a better understanding of this topology, figure 8...
gives the full description of carry propagation trees in an 8-bits adder. Each convergence on figure 8 is translated into its corresponding cell on figure 7. For example, G0,2 has a ternary convergence on figure 8 and corresponds to the first PG3 cell of the second column on figure 7. Unlike BCLA structure on figure 9 the quad trees implementation uses two new kind of cells (PG3 and PG4). Differences represented by figure 1 and 2 appear on figure 7 and 9 in term of number of columns. Except the input stage PG0 and the output stage XOR, the depth of the tree is the number of columns in between. In this example the depth of the quad trees is 2 instead of 3 for the BCLA.

4. Physical implementation

The layout is built by assembling cells from a dedicated library. They have to include both active area and wire for routing. This is one of the most important difficulty in this structure. Indeed, PG4 cells must be reached by eight input signals, PG3 needs six one and so on. Moreover, some cells compute only the generate signal because the output column doesn’t need the propagate one. Consequently, special cells must be designed to implement this side effect. And despite the fact that there are only six effective cells, we use about thirty elements in the library to treat every case of routing.

As far as the placement is concerned, it appears on figure 7 that the topology consists of a "carry propagation" block enclosed between an input column (which performs p_i and g_i) and an output column (which is a simple column of XOR gates to provide the final sum). In this architecture the carry propagation tree can be divided into several columns. Each one is built by repeating in a periodic manner four kinds of cells: PG1 (which just transmits signals P and G), PG2, PG3 and PG4. The number of repetition of each cell is 1 for the first column, 4 for the second, and more generally 4^j for the j^th column (0 ≤ j ≤ depth - 1). Thanks to the following expression: [(i/4^j)]mod4 + 1 where i denotes the row in...
dex and j the column index, the internal matrix can be built column after column. Indeed, this formula provides an array of number (see figure 10) which can be stored in a data structure in order to perform the placement. This placement uses a procedural language called "genlib" [2] which is a set of C-like functions.

Cells have been designed using symbolic approach in order to target both HGaAsII and HGaAsIII processes. The symbolic layout editor of ALLIANCE CAD TOOLS [2] has been used. This editor provides on line facilities such as equipotential extraction or design rule checking. Figure 11 gives the PG4 cell's layout.

5. Timing considerations

Gates have been designed to perform best compromise between delay, noise immunity and power consumption (see [6]). For transistor sizing we massively used automatic solutions.

Types of buffers (DCFL, BDCFL, SuperBDCFL, SquirBDCFL,...) and their transistors' sizes are automatically given by a special software. Gates, buffered or unbuffered, have been designed to assume rise and fall time about 120ps and static noise margin about 220mV with HGaAsIII vitesse technology. Some signals have great fanout and must be buffered. Those signals are always computed by a PG4 cell whose left neighbour is "empty" (PG1). We take advantage of this by inserting a super buffer.

The critical path between \( a(0) \) and \( s(N-1) \) running through the propagation tree is represented on figure 7 with a dash line. This path includes \( G_{0,N-2} \) computation.

6. Validation procedure

Except SPICE simulation, adder has been fully designed and validated with ALLIANCE CAD TOOLS [2]. Two levels of verification were performed. At the cell level we used both DRC, netlist extraction, functionnal abstraction and formal proof. At the block level, validation procedure is exactly the same. But at this level we also used vhdl simulation and automatically compared the results to ones obtained from the simulation of vhdl reference model.

7. Performances and conclusion

The adder provides a 64-bits sum in 1ns at worst case speed. This performance has been reached without using dynamic circuits (see [5]). The layout topology is a bit-slice one, in order to be used by a data-path compiler. It uses virtual connectors and respects stretchability constraint. This allows the data-path compiler to adjust dynamically power supply width or to add some tracks for over-cells routing. The height and the width of the circuit are respectively 3170\(\mu\) and 250\(\mu\).

The data sheet is the following:

<table>
<thead>
<tr>
<th>description</th>
<th>64-bits Carry Lookahead Adder</th>
</tr>
</thead>
<tbody>
<tr>
<td>technology</td>
<td>Vitesse HGaAsIII</td>
</tr>
<tr>
<td>Timing: From ( a_0 ) to ( s_{32} ): 1.0ns From ( a_0 ) to ( C_{out} ): 0.95 ns</td>
<td></td>
</tr>
<tr>
<td>Area: 0.795(mm^2)</td>
<td></td>
</tr>
<tr>
<td>Nb of transistors: 45077Tr</td>
<td></td>
</tr>
<tr>
<td>Density: 5669T/m(mm^2)</td>
<td></td>
</tr>
</tbody>
</table>

Part of the layout of this adder is presented on figure 13.
Acknowledgements

Thanks to the whole CAO-VLSI team at MASI laboratory, especially P. Remy, P. Renaud M. Louerat and L. Lucas. Thanks also to M. Desgigot.

References


