MULLET - A PARALLEL MULTIPLIER GENERATOR

K. H. Tsoi and P. H. W. Leong

Computer Science and Engineering Department,
The Chinese University of Hong Kong
email: {khtsoi, phwl}@cse.cuhk.edu.hk

ABSTRACT

A module generator called Mullet for producing near-optimal parallel multipliers in a technology independent manner is presented. Using this tool, a large number of candidate designs can be generated in order to find combinations of primitive elements which produce the best multiplier. The process of multiplication is broken down into a partial product generator (PPG) and a partial product summer (PPS). Both of these tasks can be done in a number of different ways and the best solution depends on the size of the required multiplier as well as the technology used. Mullet can be combined with a searching algorithm to find the best multiplier based on some objective. It can generate high quality multipliers for irregular architectures such as FPGAs and use features such as the dedicated multipliers. The tool can also be used to explore tradeoffs between architectures and to callibrate timing models of the primitive components. Synthesized examples using Xilinx FPGA devices are comparisons are made with those produced by the Xilinx CoreGenerator and XST tools.

1. INTRODUCTION

Although a wealth of knowledge exists about parallel multiplier design, the best architecture is dependent on the desired multiplier size and the technology which is used. For example, for a small multiplier, the partial products (PPs) might be best generated using a simple AND structure and ripple carry adders used to accumulate them. For larger sizes, a Wallace tree might be faster. Furthermore, the crossover point where the Wallace tree is faster depends on the VLSI technology used as well as whether the design is on an application specific integrated circuit (ASIC) or a field programmable gate array (FPGA).

Many different parallel multiplier architectures have been proposed in the literature (e.g. [1, 2]). High speed multipliers typically reduce the number of PPs in the partial product generator (PPG) stage via Booth's encoding and reduce the number of logic levels in the partial product summer (PPS) using tree structures. Different kinds of adders can also be used in the PPS stage. Some FPGA devices have hardwired dedicated multiplier units and practical multiplier module generators should use them when appropriate. Given the bewildering number of choices, it is difficult even for an expert to find an optimal multiplier without investing a large amount of time to the task.

In this work we describe an automatic multiplier generator called *Mullet* (MULtpLiEr Tool) that can generate multipliers which are a combination of simpler primitive elements. A search through the different combinations can easily explore tradeoffs. Furthermore, by synthesizing a number of designs and recording their performance, Mullet can determine its own timing, area and power model parameters and callibrate itself. To the best of our knowledge, no other multplier module generator is able to consider all of these issues in a unified manner. We apply this system to the generation of parallel multipliers for Xilinx Virtex FPGAs [3] and show that the multipliers generated by our tool are better than those of the Xilinx CoreGenerator and XST tools for large multiplier sizes.

The rest of the paper is organized as follows: In Section 2, we review parallel multiplier architectures which are used as primitive elements in our tool. In Section 3 we describe our tool in detail. In Section 4, we present experimental results obtained on FPGAs. Finally we draw concludes about this work in Section 5.

2. MULTIPLIER ARCHITECTURES

In this section we briefly review the basic architectures of the multiplier primitive elements which are used in our tool. For a more extensive treatment, we refer readers to the references cited and computer arithmetic textbooks [1, 4, 5, 2].

In this work, we assume that inputs are in two's complement format and we perform parallel signed multiplication of an n-bit *multiplicand* A with an m-bit *multiplier* B. The resulting product P is $n + m$ bits in size. Figure 1 shows the basic architecture of a 4-bit parallel multiplier. The multiplier can be broken down into two independent units, the PPG and PPS.

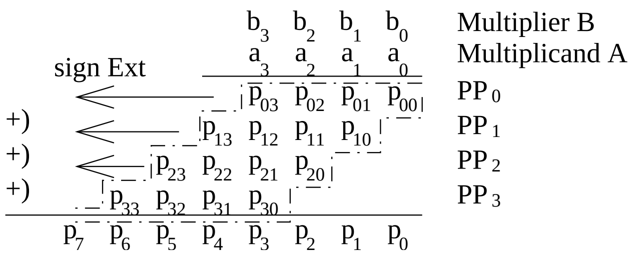


Fig. 1. A 4-bit parallel multiplier showing the partial product generator and summer.

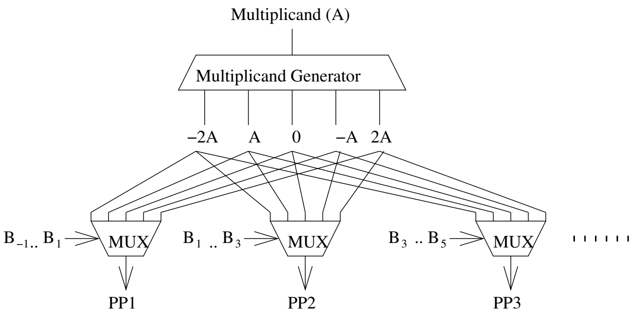


Fig. 2. Radix-4 MBE circuit.

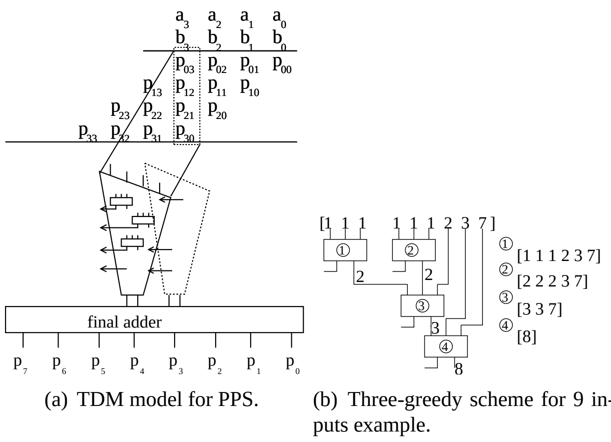
2.1. Partial Product Generators (PPGs)

AND scheme

In Figure 1, the partial products $PP_0 - PP_3$ are computed by forming the bitwise AND of b_i with A, i.e. $PP_i = b_iA$. Using this method, the number of PPs generated is m and the length of each PP is n. We call this method for generating the partial products the *AND* scheme. For signed multiplication, the PPs should be sign extended as shown in the figure.

Modified Booth Encoding (MBE)

The modified Booth's algorithm [6] considers multiple bits of B. If two bits are considered (radix-4), the partial products are generated according to a coding table. Figure 2 shows the circuit for the modified Booth encoding (MBE) PPG, with a lookup table being used to produce the appropriate multiplexor selection according to three bits of multiplier B. PP_i is formed from bits B_{2i+1}, B_{2i} and B_{2i-1} ($B_{-1} = 0$) so only $\lceil m/2 \rceil$ partial products are generated, half as many as for the AND scheme. The scheme can be generalized to higher radixes, a radix-8 MBE scheme requiring only $\lceil m/3 \rceil$ partial products. This is, of course, at the expense of a more complex partial product generation scheme. Variants of Booth's algorithm can further improve performance by introducing more complicated encoders [7] and conditional-sum adders [8].



(a) TDM model for PPS. (b) Three-greedy scheme for 9 inputs example.

Fig. 3. TDM model and 3-greedy scheme.

2.2. Partial Product Summers (PPSs)

Weighted Sum (WS)

The PPs produced by a PPG must be summed in order to form the final result. A straightforward way to do this is to use an array of adders to form the weighted sum of the PPs as show in Figure 1.

The array can be constructed using simple carry ripple adders (CRAs) or faster schemes such as carry look-ahead or carry select adders. For ripple adders, the critical path is the Manhattan distance from the LSB of the first PP to the carry out from the MSB of the last PP. This delay can be modeled as a carry chain of length $n + m$ and is shown as the dotted line in Figure 1.

Three Dimensional Method (TDM)

The three dimension method (TDM) proposed in [9] and [10] uses compressor trees to sum the partial products and a delay balancing scheme so that signal delays are minimized in a globally optimal manner. For each weight, trees are used to produce two equal weight bits of output, shown as vertical lines connected to the final adder of Figure 3(a).

An optimal method for interconnecting the compressors to reduce the global delay for the TDM has been reported by Stelling [10]. Unfortunately, the computational requirements are extremely high. So this method is not suitable for schemes such as ours in which a search over many different multipliers is proposed. We employ the three-greedy algorithm [9, 10] which produces multipliers of similar quality but is several orders of magnitude faster. Figure 3(b) shows an example of using the three-greedy algorithm to compress 9 inputs. Circled numbers represent the order of compressor generation and numbers beside the signals represent the delay of the line. The delays of the inputs to the compressors are (1,1,1,1,1,1,2,3,7). The updated available input delay list after each compressor was generated are also shown on the right. It can be seen that inputs which have large delay are placed in positions with minimum delay to the output. The technique just described uses 3:1 compressors but this can be generalized to deal with higher compression ratio.

3. MULLET ARCHITECTURE

Mullet combines the primitive elements described in the previous section to create multipliers of arbitrary size. In this section, the architecture of Mullet is described in detail in a bottom-up fashion.

To isolate the PPG and PPS parts of a multiplier circuit, we create a generalized *PP* object in CAST. A *PP* object represents a partial product which has no logic or circuitry associated with it. The attributed associated with the *PP* object include the weight of the LSB and the maximum delay from the primary input of the circuit, which is used in the TDM design.

Hardware Multipliers (HWMs)

Modern FPGA devices such as the Xilinx Virtex-II have dedicated hardware signed multipliers of fixed input size [3]. These do not use the logic resources of the FPGA and are usually faster than a similar multiplier built from the FPGA's logic resources. The HWM element is represented as a primitive object in CAST. For the Xilinx Virtex II devices considered in this work, the multiplier is 18×18-bit signed multiplier which can be used as a 17×17-bit unsigned multiplier. Larger multipliers can be constructed from HWMs.

In order to break a large multiplier into smaller ones the system first partitions the multiplier and multiplicand into several smaller bit segments. If the input segment includes the MSB, it is signed extended to 18-bits. Otherwise, a 17-bit (or smaller) unsigned HWM is used. For maximum speed and minimum logic utilization, a HWM should be used where possible. Unfortunately, the number of HWM resources on an FPGA device is limited and there are often situations in which the user may want to save some of the HWMs for other parts of the design. In Mullet, the user can specify how many HWMs to use. The system will assign the HWMs to the least significant segments first and thus reducing the critical path delay of the circuit.

Modified Booth Encoding

Mullet currently supports radix-4 and radix-8 MBE primitives which are called MBE3 and MBE4 respectively since they scan 3 and 4 bits at a time. In the MBE3 example, the $2A$ output is generated by shifting the input A and has no logic delay. Output $-A$ is generated by 2's complementing A and requires an n-bit adder. The $-2A$ output is generated by shifting the $-A$ value. The total cost of multiplicand generator is an n-bit adder in MBE3 and a 5-to-1 MUX. Mullet will first generate the $\pm$ multiplies from the multiplicand. It then segments the multiplier B according the number of bits to be scanned (currently 3 or 4). The final step is to make connections to the MUXs.

Weighted Sum (WS)

The *weight_sum* object in Mullet will accept two PP objects and output a PP object. The circuit for *weight_sum* is dynamically generated in CAST according to the width and weight of the two inputs. The inputs will be appropriately sign extended and aligned before they are summed.

Compression Tree

The most simple compressor is a $3:1$ compressor implemented as a full adder. There are different ways to implement the full adder which lead to different area and delay models. In [10], the full adder delays are modeled as an XOR gate count where the carry out delay is 1 XOR gate delay and the sum output is 2 XOR gate delays. In most FPGA architectures, this is not true due to their implementation using a 4-input LUT and fast carry logic.

We can build larger compressors by interconnecting standard $2:1$ and $3:1$ compressors. CAST will make use of LUT4 and F5 primitives in the FPGA to optimize area and speed when implementating the high ratio compressors. The delay model for these compressors is determined by the number of levels of LUT required.

The original TDM algorithm was proposed for unsigned multiplication. We modified the algorithm to accept signed numbers.

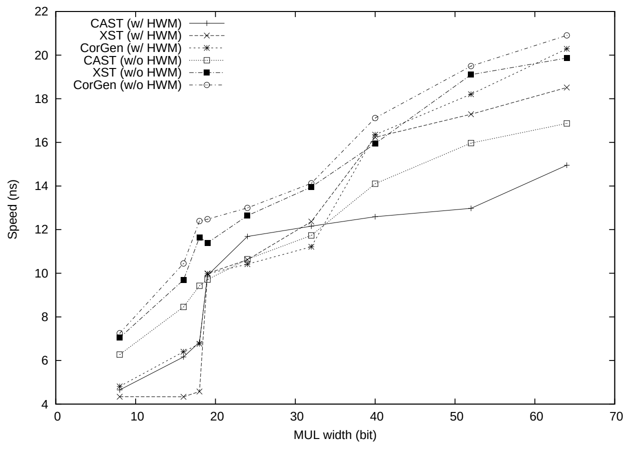
Multplier Generator

The multiplier generator accepts a set of configuration parameters as input and generate a multiplier. The PPG can be one of AND, MBE and HWM. The PPA can be either WS or TDM. The choices of PPG and PPA are independent.

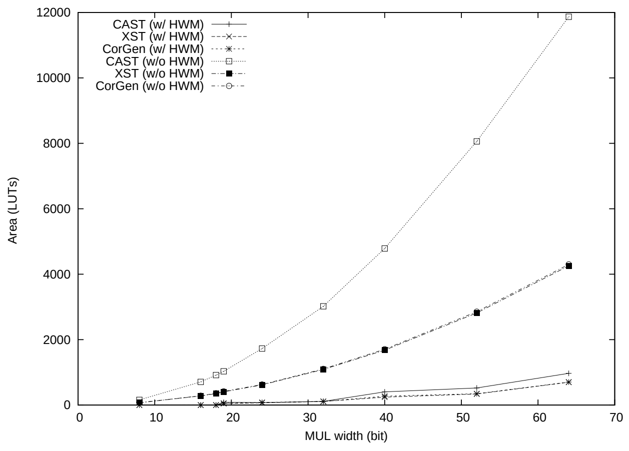
To implement the TDM algorithm, the system is able to obtain delay and other information from the circuit objects in the CAST system. Every object has its own delay model which is used to compute the maximum delay at each output. These delays are then propagated through the connections.

4. RESULTS

Multiplier performance for different input size using different schemes are shown Figure 4. All results are collected with the tools set to the highest optimization effort. The generated results were compared with the Xilinx CoreGen system as well as a multiplier directly generated using the "*" operator in XST on a Xilinx XC2V6000-6 FPGA. The correctness of a multiplier can be verified both by simulation in CAST by compiling the program with a C++ compiler and/or VHDL simulation. In the verification process, we exhaustively test all the possible inputs for a 8×8 multiplier for all possible configurations by comparing the results against



(a) MUL Speed.



(b) MUL Area.

Fig. 4. Performance of differnet multiplier schemes for different input sizes.

software multiplications. Random input vectors were used to verify larger multipliers. In this section we present experimental results based on Xilinx FPGA devices. The VHDL codes generated by Mullet were first synthesised using the Xilinx Synthesis Tools (XST) and then implemented using the ISE 6.2i tools.

The delays are measured between input and output registers of the multipliers. The configurations shown in Figure 4 are optimized for speed. As shown in the table, the performance of the generated circuit is better then those from XST and CoreGen when the input width is large. In our experiments, circuits using TDM3 performed better for multipliers larger than 40 bits because of the reduced number of logic levels. The Xilinx CoreGen can only accept input up to 64 bits so the last two entries for CoreGen are missing. For the 19 bit multiplier, our tool uses 1 MULT18X18 HWM while the other two both use 4 HWMs. The resulting speed is faster at the expense of requiring more LUTs.

In practice, we often need to find out what is the best implementation scheme for a given sized multiplier. The user may wish to optimize for speed, area or both. Using Mullet a user can easily explore tradeoffs associated with different schemes. A 52x52 bit muliplier is used as an example and the results agree with our expectation for different configurations.

5. CONCLUSION

In this paper, we presented a system that can be used to generate different parallel multiplier structures for a reconfigurable platform. It is shown that the we can combine the advantages from different algorithms and obtain an improved result, a task which is very difficult without automatic design tools.

The MG system has built in area and speed estimation functions to evaluate the generated circuits. These features allow different search methods to optimize multiplier circuits automatically. Even without the searching algorithms, it can be used to explore the complete design space in an efficient way.

6. REFERENCES

[1] I. Koren, *Computer Arithmetic Algorithms*, 2nd ed. A.K. Peters, 2002.

[2] M.D. Ercegovac and T. Lang, *Digital Arithmetic*. Morgan Kaufmann, 2004.

[3] *Virtex-II Platform FPGAs: Complete Data Sheet*, Xilinx, Inc., 2004, version 3.3.

[4] S. Waser and M. J. Flynn, *Introduction to arithmetic for digital systems designers*. Holt, Rinehart and Winston, 1982.

[5] M. J. Flynn, *Advanced computer arithmetic design*. Wiley, 2001.

[6] A. D. Booth, "A signed binary multiplication technique," *Quart. J. Mechanical and Applied Math.*, vol. 4, pp. 235–240, 1951.

[7] O. L. MacSorley, "High speed arithmetic in binary computers," *Proc. IRE*, vol. 49, pp. 67–91, 1961.

[8] W.-C. Yeh and C.-W. Jen, "High-speed booth encoded parallel multiplier design," *IEEE Transactions on Computers*, vol. 49, pp. 692–701, 2000.

[9] V. G. Oklobdzija, D. Villeger, and S. S. Liu, "A method for speed optimized partial product reduction and generation of fast parallel multipliers using an algorithmic approach," *IEEE Trans. Comput.*, vol. 45, no. 3, pp. 294–306, 1996.

[10] P. F. Stelling, C. U. Martel, V. G. Oklobdzija, and R. Ravi, "Optimal circuits for parallel multipliers," *IEEE Trans. Comput.*, vol. 47, no. 3, pp. 273–285, 1998.