# Area Optimization of Multi-Cycle Operators in High-Level Synthesis*

M. C. Molina, R. Ruiz-Sautua, J. M. Mendías, R. Hermida
Dpto. Arquitectura de Computadores y Automática
Universidad Complutense de Madrid
{cmolinap, mendias, rhermida}@dacya.ucm.es, rsautua@fdi.ucm.es

## Abstract

*Conventional high-level synthesis algorithms usually employ multi-cycle operators to reduce the cycle length in order to improve the circuit performance. These operators need several cycles to execute one operation, but the entire functional unit is not used in any cycle. Additionally, the execution of operations over wider multi-cycle operators is unfeasible if their results must be available in a smaller number of cycles than the functional unit delay. This obliges to add new functional resources to the datapath even if multi-cycle operators are idle when the execution of the operation begins.*

*In this paper a new design technique to overcome the restricted reusability of multi-cycle operators is presented. It reduces the area of these functional units allowing their internal reuse when executing one operation. It also expands the possibilities of common hardware sharing as it allows the partial use of multi-cycle operators to calculate narrower operations faster than the functional unit delay. This technique is applied as an optimization phase at the end of the high-level synthesis process, and can optimize the circuits synthesized by any high-level synthesis tool.*

## 1. Introduction

A High–Level Synthesis (HLS) process transforms the behavioural description of a circuit into a Register-Transfer-Level (RTL) implementation. It involves three major tasks: scheduling, allocation, and binding. Scheduling selects the number of clock cycles and their duration, and assigns operations of the behavioural description to them. Allocation selects a set of resources to execute the specified behaviour, and binding assigns operations to functional units (FUs), variables to storage elements, and data transfers to routing resources.

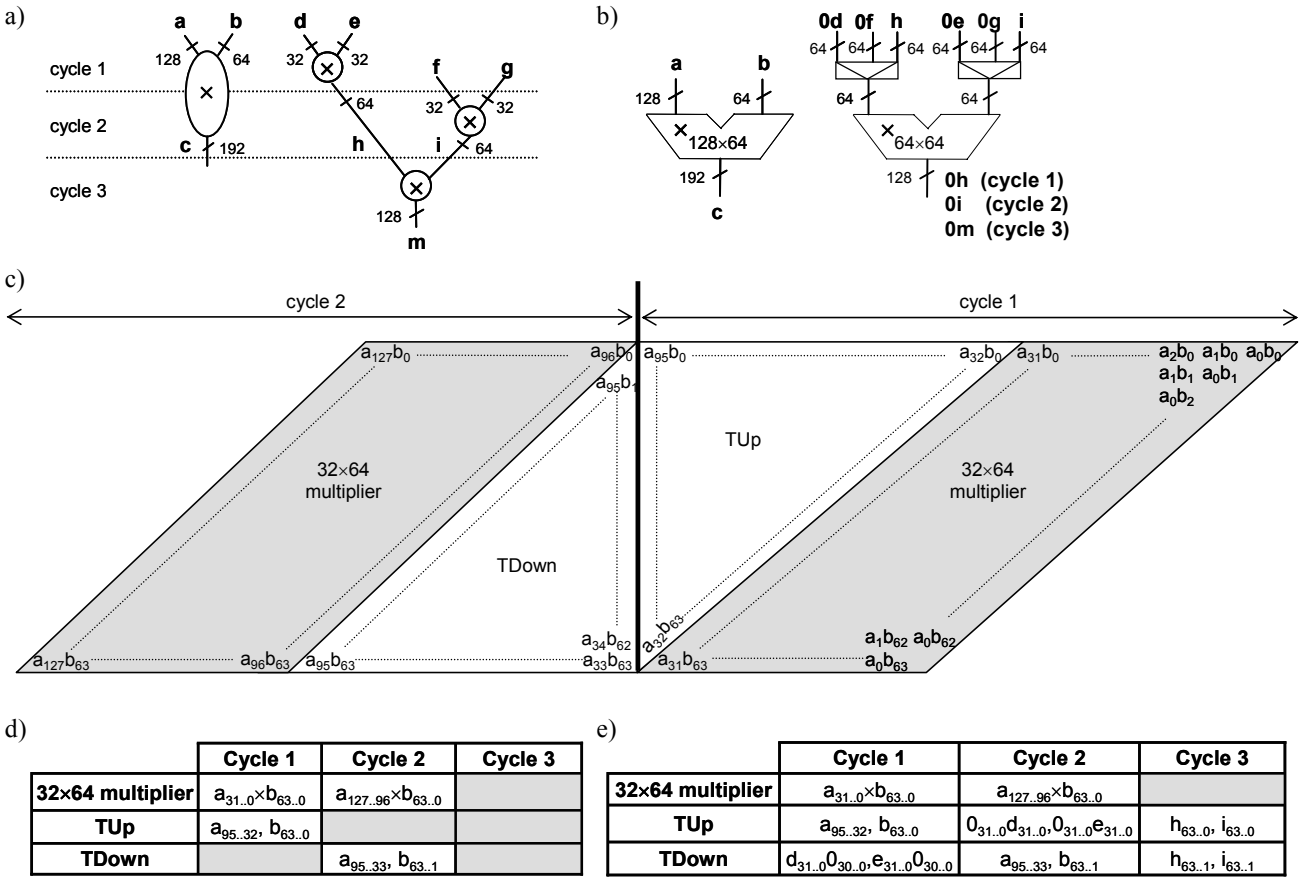
Many efforts in high-level scheduling have been concentrated on improving circuit performance (time required to execute all the specification operations). Traditionally, pipelining has been the preferred technique although it does not reduce the circuit latency [1-2]. Most scheduling algorithms have improved circuit performance with chaining [3-5] and multi-cycle [5-6] techniques. Chaining helps to reduce the number of clock cycles through the execution of several data-dependent operations in the same cycle. This technique requires more FUs as chained operations cannot share HW resources. Multi-cycle reduces the cycle duration through the execution of long operations across several consecutive cycles. Here, the results produced need several cycles to be available. Also, the reusability of multi-cycle operators is quite limited. A multi-cycle FU can execute as many specification operations as the circuit latency divided by the number of cycles needed to perform the calculus. But even in the best case, when the multi-cycle FU executes this maximum number of operations, it is only partially used in each cycle. Then, the best use of multi-cycle operators implies some HW waste that can be translated as redundant area.

In order to minimize the datapath area most allocation algorithms bind operations to wider FUs. Here, the time required to calculate the operation result is equivalent to the FU delay. In particular, the execution of one operation over a wider multi-cycle FU requires the same number of cycles as the FU, even if the operation could be calculated faster. This makes the execution of operations over wider multi-cycle operators impossible if they must finish in a smaller number of cycles than the FU delay.

In this paper, we propose an optimization technique that minimizes the HW waste in multi-cycle operators increasing their internal reuse. It reduces the area of the multi-cycle FUs without affecting their delays, and allows the execution of smaller operations faster than the operator delay. The proposed technique is applied as an optimization phase after the HLS process. Other optimization phases have been previously added at the end of the HLS process with successful results [7-10]. The post-synthesis optimizations can be applied to the RTL circuits synthesized by any HLS algorithm, and thus can benefit from future improvements in HLS tools.

* This work was supported by Spanish Government under grant CYCIT TIN 2005-5619

978-3-9810801-2-4/DATE07 © 2007 EDAA




| | Cycle 1 | Cycle 2 | Cycle 3 |
|---|---|---|---|
| 32×64 multiplier | $a_{31..0}\times b_{63..0}$ | $a_{127..96}\times b_{63..0}$ | |
| TUp | $a_{95..32}$, $b_{63..0}$ | | |
| TDown | | $a_{95..33}$, $b_{63..1}$ | |

| | Cycle 1 | Cycle 2 | Cycle 3 |
|---|---|---|---|
| 32×64 multiplier | $a_{31..0}\times b_{63..0}$ | $a_{127..96}\times b_{63..0}$ | |
| TUp | $a_{95..32}$, $b_{63..0}$ | $0_{31..0}d_{31..0}$,$0_{31..0}e_{31..0}$ | $h_{63..0}$, $i_{63..0}$ |
| TDown | $d_{31..0}0_{30..0}$,$e_{31..0}0_{30..0}$ | $a_{95..33}$, $b_{63..1}$ | $h_{63..1}$, $i_{63..1}$ |

**Figure 1. a) Scheduled DFG, b) conventional implementation, c) calculus matrix of a 128×64 multiplication in 2 cycles, d) availability of the new FUs, and e) binding of the operations executed over the 64×64 multiplier.**

## 2. Motivational example

This section provides the insight into the internal reuse of multi-cycle operators, and the new possibilities of HW sharing provided by the proposed technique. Its underlying idea is the substitution of every multi-cycle operator for the minimum HW necessary to execute the part of the operation calculated in every cycle, reusing as much as possible the same FUs in all the cycles. The following paragraphs introduce the proposed technique with the aid of a simple example.

Figure 1 a) shows a scheduled data flow graph (DFG) of a behavioural description formed by 4 multiplications. A conventional implementation using multi-cycle operators is illustrated in fig. 1 b). The datapath is formed by two FUs: one multi-cycle 128×64 multiplier, and one 64×64 multiplier. The 128×64 multiplier calculates $a\times b$ across cycles 1 and 2, and the 64×64 multiplier computes $d\times e$, $f\times g$, and $h\times i$ in cycles 1, 2, and 3, respectively. In the best case, a multi-cycle operator performs the same number of calculi in every execution cycle, in order to minimize the slack times wasted. Then, we can assume that the 128×64 multiplier calculates the 96 least significant bits (LSB) of the result in cycle 1, and the 96 most significant bits (MSB) in cycle 2, and the time spent performing the calculations coincides in both cycles. Figure 1 c) shows the calculus matrix of a 128×64 multiplication, and the vertical line divides the part of the multiplication computed in cycles 1 and 2. Inside every operation fragment we can identify the calculus matrix of one 32×64 multiplication, shown in grey colour. This means that we can use a 32×64 multiplier to execute $a_{31..0}\times b_{63..0}$ in cycle 1 and $a_{127..96}\times b_{63..0}$ in cycle 2. Then, the internal reuse of a 32×64 multiplier implies a 25% reduction of the FU area. In addition to the two new multiplications, we find two other new operations in the calculus matrix of a 128×64 multiplication. These new operations perform the calculus included inside an isosceles right-angled triangle. TUp is the operation included in the triangle with the right angle on the top, and TDown the one included in the triangle with the right angle on the bottom. In [11] a possible implementation of these operations based on adders and *AND* gates is presented. The multi-cycle multiplier is then decomposed

into one 32×64 multiplier, one TUp operator, one TDown operator and one adder used in cycles 1 and 2 to sum up the partial results. These additions do not increment the cycle length due to the rippling effect of the operations considered in the example.

In addition to the internal reuse of the adder and the 32×64 multiplier, it is possible to use the new FUs to execute other specification operations. In the example, if we are able to allocate all the operations bound to the 64×64 multiplier to the new FUs, then the 64×64 multiplier can be removed from the datapath. Figure 1 d) shows the availability of the new FUs, the grey cells correspond to the cycles where the FUs are idle. In order to remove the 64×64 multiplier, *d*×*e* must be executed over the TDown operator (the only one without allocated operations in cycle 1), *f*×*g* over TUp, and *h*×*i* over any of the FUs obtained from the decomposition of the multi-cycle multiplier.

Inside both the TUp and TDown operators it is possible to identify the calculus matrix of several smaller multiplications. In general, the execution of an *m*×*m* multiplication over an *n* bits TUp or TDown operator is possible if $2m-1 \le n$ . Operations *d*×*e* and *f*×*g* satisfy this equation and then can be executed over the idle FUs. Figure 1 e) shows the execution of both multiplications over the TDown and TUp FUs. In both cases, the multiplications do not occupy the overall operator and it is necessary to extend the operands to invalidate some of the calculus performed by the FU to obtain the correct result. Note that the TUp and TDown operators linked as shown in fig. 1 c) constitute one 64×64 multiplier. Thus, *h*×*i* can be executed in cycle 3 distributed over these two FUs. The operands supplied to every FU are shown in fig. 1 e). The results produced by the TUp and TDown FUs are summed up in the same adder used in cycles 1 and 2 to perform the 128×64 multiplication.

The optimized datapath is finally formed by one 32×64 multiplier, one 64 bits TUp operator, one 63 bits TDown operator and one adder. Table I shows the area and time results of both implementations measured by *Synopsys Design Compiler* after logic synthesis. The area values include the area of FUs, storage and routing resources and the controller. As shown in the table, the overall area has been significantly decreased, around 30%. As expected, the execution times coincide in both cases. Note that this technique requires the storage of the partial results calculated by the multi-cycle operator until, at least, the end of the operation. However, the new storage needs do not waste the area reductions achieved because, in most cases, the storage elements present in the original datapath are used inside the new multi-cycle operator.

**Table I. Comparison of implementations**

| | Area (gates) | Cycle length (ns) |
|---|---|---|
| Original datapath | 109668 | 39.592 |
| Optimized datapath | 77920 | 39.596 |

# 3. Proposed algorithm

The proposed technique is integrated into an automatic algorithm to optimize the circuit synthesized by a conventional HLS algorithm. The algorithm progressively transforms the synthesized implementation across two phases: decomposition of multi-cycle operators and removal of some datapath FUs. The first phase is executed only once, and the second one until the datapath satisfies some conditions. In the first phase every multi-cycle FU is substituted for a set of smaller FUs. Some of these new resources are used in several cycles of the multi-cycle operator (internal HW reuse). In the second phase some of the mono-cycle operations are bound to the new FUs with the aim of removing some of the FUs of the original datapth. The following subsections detail both phases.

## 3.1 Decomposition of multi-cycle operators

The synthesized datapath is transformed substituting every multi-cycle operator for a set of smaller FUs that are used in one or several cycles of the multi-cycle FU. The decompositions performed try to maximize the internal reuse of the smaller FUs, and thus to minimize the operator area. We have tried to find the largest common operation included in the fragments calculated in every execution cycle of the multi-cycle operator.

In the present version of the optimization algorithm we have considered multi-cycle multipliers that have been decomposed into mono-cycle multipliers, adders, and the two new operators TUp and TDown. These two new FUs compute the operations included in one isosceles right-angled triangle extracted from the calculus matrix of a multiplication. In [11] some implementations of both FUs are presented. In order to balance the set of calculations performed by the multi-cycle FU in every cycle, we have divided the FU to get the same number of result bits in every execution cycle. These partitions allow the fragments of the multi-cycle operator to share bigger FUs.

**3.1.1 Decomposition of 2-cycles multipliers.** The decomposition of one 2-cycles multiplier of *m*×*n* bits is performed fragmenting vertically the calculus matrix of the multiplication to get the $\lceil (m+n)/2 \rceil$ LSB of the result in the first cycle and the remaining ones in the second. We have distinguished the following two cases:

1) *m*×*m multiplication.* The multi-cycle FU is substituted for one mono-cycle multiplication of $\lceil m/2 \rceil \times \lceil m/2 \rceil$ bits that is used in cycles 1 and 2, two TUp FUs of $\lfloor m/2 \rfloor$ bits used in cycle 1, and two TDown FUs of $\lfloor m/2 \rfloor$-1 bits used in cycle 2. If *m* is an odd number, then it is necessary to insert a column of zeros between the two fragments obtained in the decomposition of the operator in order to have the cited set of FUs. The additional column in the calculus matrix of the operation does not increment its execution time because it does not enlarge the chain of carry propagations. Figure 2 a) illustrates this decomposition, the multiplier in grey colour is used in

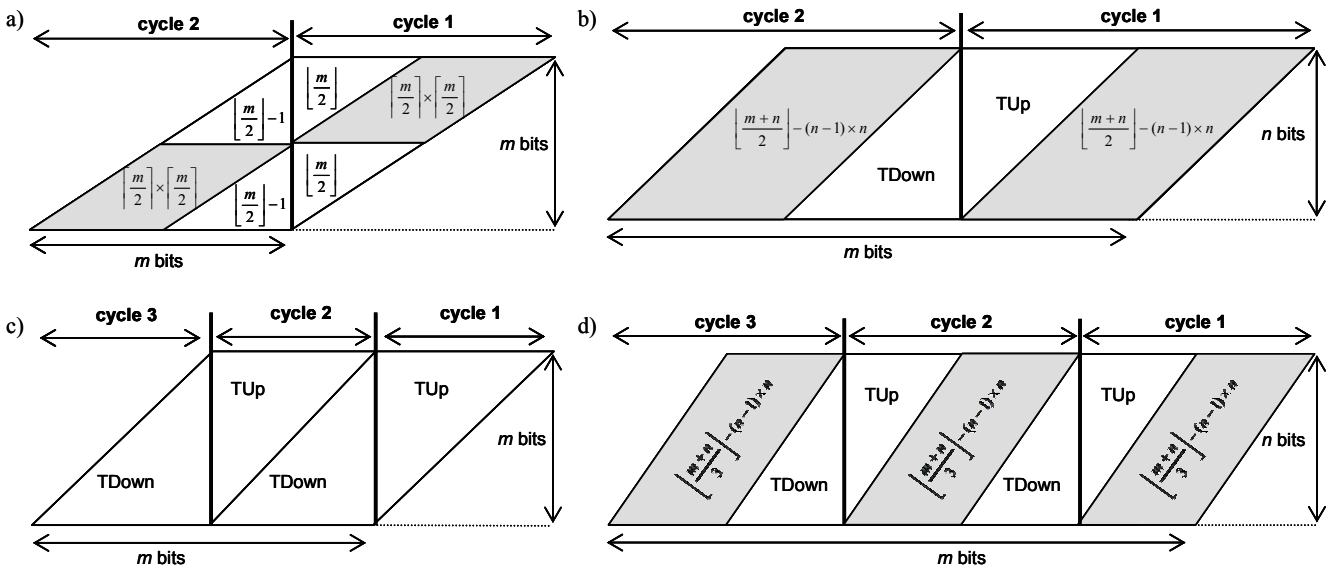



**Figure 2. a) Decomposition of a 2-cycles *m*×*m* multiplier, b) 2-cycles *m*×*n* multiplier, c) 3-cycles *m*×*m* multiplier, and d) 3-cycles *m*×*n* multiplier.**

cycles 1 and 2 (internal reuse). As shown in the figure, this decomposition reduces 25% the operator area (one of the multipliers in grey is removed).

2) *m×n multiplication, being m≠n.* The multi-cycle FU is substituted for one mono-cycle multiplication of $\lfloor (m+n)/2 \rfloor - (n-1) \times n$ bits that is used in cycles 1 and 2, one TUp FU of *n*-1 bits used in cycle 1, and one TDown FU used in cycle 2. If *m*+*n* is an even number, then the width of the TDown operator becomes *n*-2 bits, and in this case it is also necessary to insert a column of zeros between both operator fragments. If *m*+*n* is an odd number, then the width of the TDown operator is *n*-1 bits. Figure 2 b) shows this decomposition, with the multiplier in grey colour used in cycles 1 and 2. It also reduces 25% the operator area (one of the multipliers is removed).

**3.1.2 Decomposition of 3-cycles multipliers.** The decomposition of one 3-cycles multiplier of *m*×*n* bits is performed fragmenting vertically the calculus matrix of the multiplication to obtain the $\lceil (m+n)/3 \rceil$ LSB of the result in the first cycle, and the $\lfloor (m+n)/3 \rfloor$ MSB in the third one. Two different cases appear:

1) *m×m multiplication*. The multi-cycle FU is substituted for one TUp FU of $\lceil 2m/3 \rceil$ bits used in cycles 1 and 2, and one TDown FU of $\lceil 2m/3 \rceil$ bits used in cycles 2 and 3. Figure 2 c) shows the 50% area reduction achieved (one TUp FU, and one TDown FU removed).

2) *m×n multiplication, being m≠n*. The multi-cycle FU is substituted for one mono-cycle multiplication of $\lceil (m+n)/3 \rceil - (n-1) \times n$ bits that is used in cycles 1, 2, and 3, one TUp FU of *n*-1 bits used in cycles 1 and 2, and one TDown FU of *n*-1 bits used in cycles 2 and 3. Figure 2 d) shows the 40% area reduction achieved (two multipliers, one TUp FU, and one TDown FU removed).

The decompositions of multi-cycle multipliers that need more than 3 cycles to calculate the operation results are performed similarly to the 3-cycles operators. In these decompositions all the fragments take the same time to be executed. However, the algorithm can be easily modified to comply with non-integer multi-cycle FUs [7] that produce less result bits in the last execution cycle than in the previous ones. In this case, the vertical partitions performed are not equidistant but the extraction of the new FUs is carried out as explained above.

## 3.2 Removal of some datapath FUs

The main goal of this phase is the removal of some datapath FUs to reduce the overall circuit area. One FU can be removed from the datapath if all the operations bound to it can be executed over the remaining functional resources. In the previous phase the number of datapath FUs has been increased, and there are also more idle FUs left in every cycle. The algorithm tries to reallocate the specification operations over these idle FUs. To this purpose, it allows the execution of one operation over a wider FU, and also distributed over several chained FUs.

**3.2.1 Binding one operation to one idle FU.** The algorithm takes into account the following cases:

1) One *m*×*n* multiplier can execute a *k*×*p* multiplication if *k*≤*m* and *p*≤*n*.

2) One *m*×*n* multiplier (*m*≥*n*) can calculate one TUp/TDown operation of *k* bits (being *k*≤*n*) as illustrated in fig. 3 a).

3) One *m* bits TUp/TDown FU can execute one TUp/TDown operation of *n* bits (being *n*≤*m*).

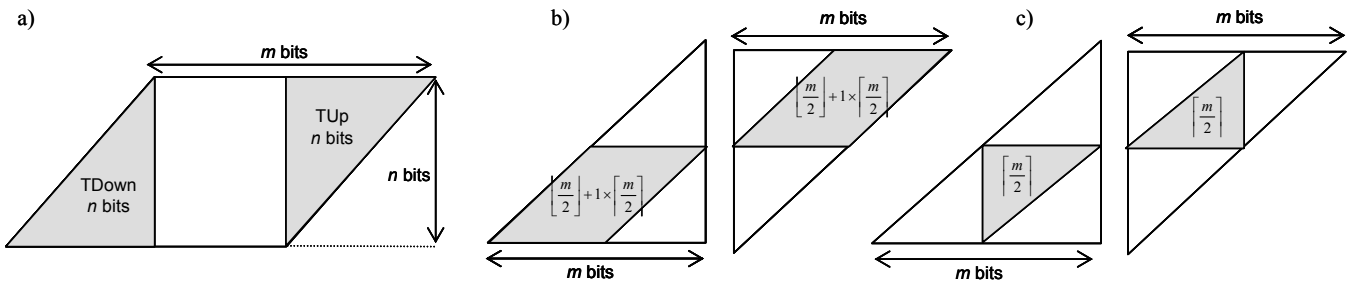



**Figure 3. a) Execution of one TUp/TDown operation over a multiplier, b) one multiplier over one TDown/TUp FU, and c) one TUp/TDown operation over one TDown/TUp FU.**

4) One $m$ bits TUp/TDown FU can calculate one $\lfloor m/2 \rfloor + 1 \times \lceil m/2 \rceil$ multiplication as in fig. 3 b).

5) One $m$ bits TUp/TDown FU can execute one TDown/TUp operation of $n$ bits (being $n \leq \lceil m/2 \rceil$) as illustrated in fig. 3 c).

**3.2.2 Binding one operation to several chained Fus.** The datapath FUs can be chained to execute wider operations. The following rules compose wider FUs:

1) An even number $n$ of triangles of the same width $m$ can be chained horizontally to form one multiplier of width $(n/2) \cdot (m+1) \times m$. The number of TUp and TDown operators must be the same, and every operator must be located between two FUs of the other type. Figure 4 a) illustrates this composition.

2) An even number $n$ of triangles of the same width $m$ can be chained vertically to other set of $n$ chained triangles of $m$ bits to form one multiplier of width $(n/2) \cdot (m+1) \times 2 \cdot m$. The number of TUp and TDown operators must be also the same, and every operator must be located between two FUs of the other type. Every row of chained triangles must be shifted $m$ bits to the left from the above row. Figure 4 c) shows this composition.

3) One $m \times n$ multiplication can be chained horizontally to one $k \times n$ to form a multiplication of $(m+k) \times n$ bits. Figure 4 d) illustrates this composition.

4) One $m \times n$ multiplication can be chained vertically to one $m \times k$ to form a multiplication of $m \times (n+k)$bits. The second multiplication must be shifted $n$ bits to the left from the above one as shown in fig. 4 b).

**3.2.3 Removal execution.** This phase consists of a loop. Every iteration tries to remove one of the datapath FUs, from the widest one to the narrowest. The FUs considered either in any cycle do not execute any operation, or calculate a narrower one. The algorithm tries to bind the operations assigned to the target FU to the remaining FUs in the datapath, with no change in its execution cycle. The first choice is always the binding of one operation to one FU, and as second choice it tries to bind one operation to several chained FUs. The previous rules are checked in order for every FU considered and every operation assigned to it. The phase finishes when either all the FUs have been considered, or all the datapath FUs are used in every cycle to execute operations of their same type and width. In this case, the removal of any datapath FU would make the execution of the operations scheduled in that cycle impossible. This second phase can be executed independently of the first one to optimize datapaths without multi-cycle operators. In this case, it tries to increase the internal reuse of FUs that execute smaller operations, binding them to several idle FUs chained as explained in the previous subsections.

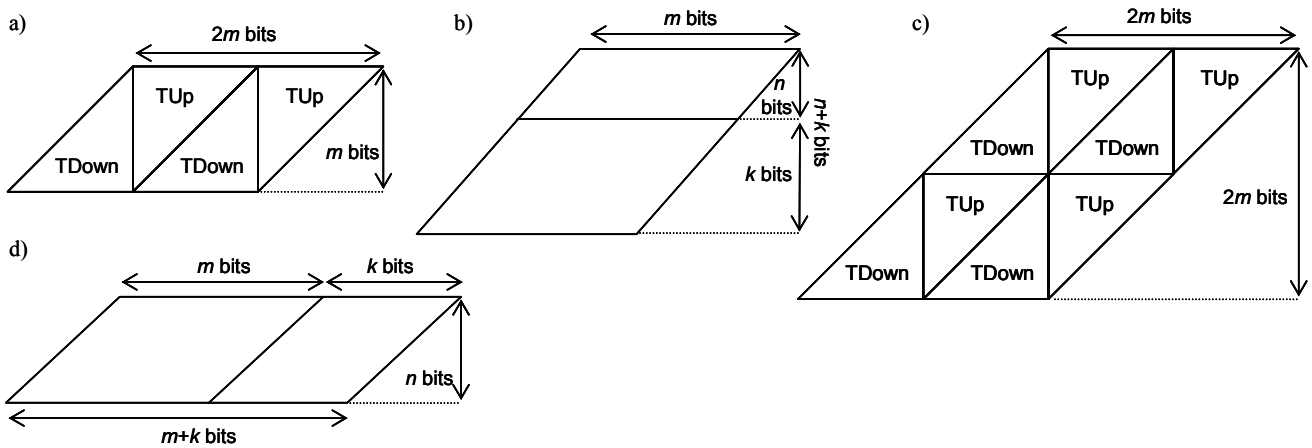



**Figure 4. a) Horizontal composition of triangles, b) vertical composition of multipliers, c) vertical composition of triangles, and d) horizontal composition of multipliers.**

## 4. Experimental results

The experimental work performs the optimization of some circuits synthesized by the commercial tool *Synopsys Behavioral Compiler* (BC), version 2001.08. Both RT-level implementations, the one produced by BC and the optimized one, are processed by *Synopsys Design Compiler* (DC) to obtain the area and time reports. The implementations of both the TUp and TDown operators have been added to the design library. Time results shown in this section are measured in nanoseconds and areas in number of equivalent gates. In all cases areas include the FUs, storage and routing units, glue logic, and controller.

We have synthesized and optimized twenty different synthetic specifications ranging from 50 to 100 operations (around 60% are multiplications). In the implementations synthesized by BC the percentage of area occupied by multi-cycle operators is around 20% of the total area. Figure 5 shows the average areas of these circuits grouped by the number of operations. In all the cases the optimized implementation is quite smaller than the original one, and the amount of area saved is 38% on average.

In addition to the synthetic specifications, we have synthesized the fifth order elliptical wave filter, and some modules of the ADPCM decoding algorithm described in Recommendation G.721 of the CCITT. Table II shows the datapath area and cycle length comparisons between the implementation obtained by BC and the optimized one, for three different latency values (λ). Areas have been decreased 40% on average, and reductions of up to 65% have been obtained. The ADPCM modules synthesized and optimized are: *Inverse Adaptative Quantizer* (IAQ), *Output PCM Format Conversion* (OPFC), *Synchronous Coding Adjustment* (SCA), and *Tone and Transition Detector* (TTD). Table III shows the areas of the implementation obtained by BC and our optimization. The circuit area has been reduced 24% on average. In all the experiments, the area reductions have incremented neither the latency nor the cycle length, which have remained constant.

**Table II. Synthesis of the fifth-order elliptic wave filter**

| λ | Circuit area (gates) | | | Cycle length (ns) | |
|---|---|---|---|---|---|
| | Original | Optimized | Saved | Original | Optimized |
| 8 | 76540 | 58190 | 23% | 58.63 | 58.68 |
| 11 | 70652 | 45926 | 35% | 51.59 | 51.54 |
| 16 | 67944 | 23840 | 65% | 32.27 | 32.36 |

**Table III. Synthesis of some modules of ADPCM decoder**

| Module | λ | Area (gates) | | |
|---|---|---|---|---|
| | | Original | Optimized | Saved |
| IAQ | 3 | 798 | 672 | 16% |
| TTD | 5 | 1873 | 1282 | 32% |
| OPFC + SCA | 12 | 1226 | 870 | 30% |

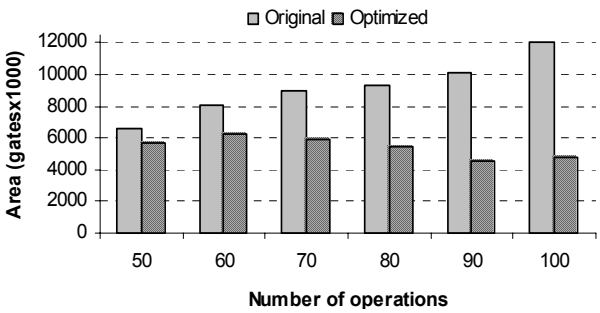



**Figure 5. Area results for the synthetic circuits.**

## 5. Conclusion

The proposed optimization technique reduces the area of the circuits synthesized by regular HLS algorithms using multi-cycle operators. It allows the internal reuse of the HW inside the multi-cycle FUs in their different execution cycles, and also their partial reuse to execute smaller operations faster than the FU delay. Furthermore, the proposed transformations can be used to increment the partial reuse of mono-cycle operators, and are compatible with recent techniques like bit-level chaining and non-integer multi-cycle. All the presented optimizations keep constant both the cycle length and the latency of the original implementation, therefore they do not waste the benefits obtained from the use of multi-cycle operators.

## References


[1] N. Park, and A. Parker. "Sehwa: A Software Package for Synthesis of Pipelines from Behavioural Specifications". IEEE Trans. on CAD, March 1988.

[2] K.S. Hwang, A.E. Casavant, C.T. Chang, and M.A. d'Abreu. "Scheduling and Hardware Sharing in Pipelined Data Paths". In Proc. ICCAD 1989.

[3] S. Ogrenci, R. Kastner, E. Bozorgzadeh, M. Sarrafzadeh. "A Scheduling Algorithm for Optimization and Early Planning in High-level Synthesis". ACM TODAES, January 2005.

[4] S. Gupta, T. Kam, M. Kishinevsky, S. Rotem, N. Savoiu, N. Dutt, R. Gupta, A. Nicolau. "Coordinated Transformations for High-Level Synthesis of High Performance Microprocessor Blocks". In Proc. DAC 2002.

[5] A. Kountouris, C. Wolinski. "Efficient Scheduling of Conditional Behaviors for High-level Synthesis". ACM TODAES, July 2002.

[6] J. Jeon, D. Kim, D. Shin, K. Choi. "High-level Synthesis under Multi-cycle Interconnect Delay". In Proc. ASP-DAC 2001.

[7] S. Park, and K. Choi. "Performance-Driven High-Level Synthesis with Bit-Level Chaining and Clock Selection". IEEE Trans. on CAD, February 2001.

[8] Z. Yu, K. Khoo, and A. Wilson, Jr. "The Use of Carry-Save Representation in Joint Module Selection and Retiming". In Proc. DAC 2000.

[9] V. Raghunathan, S. Ravi, and G. Lakshminarayana. "Integrating Variable-Latency Components into High-Level Synthesis". IEEE Trans. on CAD, October 2000.

[10] J. Zhu, and D. D. Gajski. "Soft Scheduling in High Level Synthesis". In Proc. DAC 1999.

[11] R. Ruiz-Sautua, M. C. Molina, J. M. Mendias, R. Hermida. "Pre-synthesis Optimization of Multiplications to Improve Circuit Performance". In Proc. DATE 2006.