Extremely High Speed Digital Down Converter Design for Wireless Communication Applications

Daniel Besse

Microelectronic Systems Laboratory (LSM)

Professor: Yusuf Leblebici
Supervisor: Tugba Demirci

Lausanne, Switzerland
January 12, 2016
Extremely High Speed Digital Down Converter Design for Wireless Communication Applications

Abstract. More than 4 GHz semi-custom design of a digital down converter (DDC) targeting time-interleaved analog-to-digital converters in wireless receivers for communication applications is reported using STMicroelectronic 28 nm fully depleted Silicon on insulator (FDSOI) technology. Deep pipelining combined to an optimal synthesis and place and route flow enables the reach of the minimal delay for the given technology and standard cell library. Circuit structure using multiplier-like compression trees is adapted to this design with an automated generation algorithm proposed to place full-adders based on the three-greedy method. Final delays more than 30% faster than with an ideal adder-based structure are reported for similar area. Dynamic data flip-flops (DFF) developed specifically to reduce setup time and clock-to-Q delay have been designed and characterized based on the true single phase clock (TSPC) structures. Their efficiency for high speed circuits is demonstrated with 16% improvement in maximal sustainable clock frequency and have been made fully compatible with the traditional semi-custom flow for easy reuse in other designs. Extended verification in extracted views of the layouts have been performed to ensure correct delay metrics, highlighting pitfalls with wire-load modeling for metal routing. Complete test interface is also detailed and a test chip has been sent for tape-out to STMicroelectronic to allow Silicon measurements including two DDC cores, one with and one without TSPC DFFs, and a frequency divider test block. Results of these measurements are expected to provide confirmation of the high working frequency of the DDC designs as well as quantify the improvements unlocked by the switch to dynamic logic for DFFs (these results are not part of this report though as manufactured chips will not be available before mid-2016). Thanks to the use of an entirely semi-custom flow, most of this design remains generic and highlighted guidelines can be used to reproduce similar results in different circuits. It constitutes therefore a proof of feasibility for such high speed circuits.

Keywords. Digital Down Converter (DDC); Time-Interleaved ADC (TI-ADC); High Speed Digital Design; Semi-Custom Flow; True Single Phase Clock (TSPC) Flip-Flops; Three-Greedy Method; Compression Tree

Master thesis report performed between September 14, 2015 and January 12, 2016 at the Microelectronic Systems Laboratory (LSM, EPFL) under the supervision of Tugba Demirci and Prof. Yusuf Leblebici as a partial requirement towards the degree of Masters of Science in Electrical Engineering from École Polytechnique Fédérale de Lausanne (EPFL).

Lausanne, le ____________________

Daniel BESSE
## Contents

**Abstract** iii  

1 **Introduction** 1  

2 **Polyphase DDC for TI-ADC** 3  
  2.1 Overall architecture ................................................. 3  
  2.2 Polyphase DDC .......................................................... 5  
  2.3 Compression tree ....................................................... 7  

3 **Circuit generation and optimization** 9  
  3.1 Structural generation .................................................. 9  
  3.1.1 Tree generation and reduction .................................... 9  
  3.1.2 Three-greedy implementation ...................................... 10  
  3.2 Synthesis and pipelining ............................................ 14  
  3.3 Place and Route ....................................................... 16  
  3.4 Note on wire-load modeling ......................................... 16  

4 **Dynamic flip-flops** 19  
  4.1 TSPC flip-flops generics ............................................... 19  
  4.2 Design and metrics .................................................. 21  
  4.2.1 TSPC flip-flop .................................................... 21  
  4.2.2 Double precharge TSPC flip-flop ................................ 22  
  4.3 Characterization and semi-custom flow integration ............ 25  

5 **Test-interface for tape-out** 27  
  5.1 Top-level architecture and layout ................................... 27  
  5.2 Frequency divider .................................................... 30  
  5.3 Test procedure ....................................................... 31  
  5.3.1 Digital down converters .......................................... 31  
  5.3.2 Frequency divider ................................................ 33  
  5.3.3 Ring oscillator .................................................... 33  

6 **Discussion of results and outlook** 35  
  6.1 Taped-out design weaknesses ........................................ 35  
  6.2 Design optimization .................................................. 35  
  6.2.1 RTL optimization and pipeline length .......................... 36  
  6.2.2 Standard cells in dynamic logic ................................. 36
Introduction

Bandwidth demand from mobile communications is ever increasing and experts seem to agree that a shift towards higher frequency bands such as mm-wave will already be required for 5G [1–3]. The Nyquist frequency of such a signal with a carrier in the 30 GHz range is technologically not sustainable for a classical single-channel analog-to-digital converter (ADC) in the receive path with direct digital down conversion. Time interleaving of converters has been proposed long ago to circumvent this problem as well as soften constraints on ADCs even for lower frequency carriers [4]. The resulting architecture requires recombination of the channels to retrieve the in-phase (I) and quadrature (Q) signals. This recombination can be expressed as finite impulse response (FIR) filters for each subchannel followed by a final addition tree and results to the same operation on the signal as the famous cascaded integrator comb (CIC) segments introduced by Hogenauer [5] which are often used for this type of applications [6–10]. Adequate choice of the sampling frequency also allows removal of the mixers, with the mixing operation implicitly performed by the time interleaving switches of the converters and the correct attribution of 0, 1, or −1 coefficients to each channel.

The implementation of the digital down converter (DDC) and optimization techniques to operate at very high speed is the main focus of this work. Its structure is analyzed and compared with adder trees found in multipliers for which optimization methods have been developed [11–14]. Optimization of this circuit is therefore proposed on algorithmic level followed by an efficient and deeply-pipelined synthesis to achieve the fastest speed possible. Registers form an important part of the design when reaching the limits of pipelining in order to reach the minimal delay defined by the technology. As such, faster versions of data flip-flops (D flip-flops or DFF) have been created using dynamic logic. These dynamic flip-flops have been fully characterized for inclusion in the standard synthesis followed by place and route flow (semi-custom flow). Verification and speed simulations have been performed on extracted views of the complete DDC incorporating therefore layout parasitics in the final reported results. A test interface for silicon measurements has also been implemented and the whole circuit plus some small test blocks such as frequency dividers for DFF testing have been included in a test chip sent to STMicroelectronics (ST) [15] in mid-December 2015 for tape-out in their 28 nm fully depleted Silicon on insulator (FDSOI) technology. Typical production delays in the order of 6 months will make Silicon results available only after the defense of this work though.

This report starts by detailing the particularities of polyphase DDCs in chapter 2.
Chapter 3 explains their optimal implementation as compression trees (which is design specific) and the generic semi-custom flow used for such a high-speed design. Standard cells optimization by the conception of dynamic flip-flops forms chapter 4. Namely, true single phase clock (TSPC) flip-flops from [16, 17] are adapted to the used technology and their integration in the generic semi-custom flow is detailed. The test structure developed for tape-out is explained afterwards in chapter 5. Finally, chapter 6 highlights recommendations for high-speed circuits derived from the outcomes of this design, concluding with improvement ideas to reach even higher clock frequencies.
Polyphase DDC for TI-ADC

There has been recently a general trend in favor of reconfigurable transievers [18] in wireless communications to adapt to multiple standards. Digital processing of signals is therefore preferred over analog treatment thanks to its easy reconfigurability. This is a typical application for ADCs with high sampling rates such as time-interleaved ADCs. The logical choice for this project is therefore to base the overall structure on a direct down conversion to baseband with the ADC as soon as possible in the receive path (only preceded by a low-noise amplifier (LNA) for instance). The resulting system is described in the following sections and mainly based on the proposal by Kim et al [19].

2.1 Overall architecture

The classical block diagram of a direct down conversion receiver is given in Figure 2.1 with its time-interleaved equivalent in Figure 2.2. It consists of an antenna whose signal is fed to a number $R$ of time-interleaved sub-channel ADCs. Each sub-channel converter has a resolution of $S_{in}$ bits and works at a conversion frequency $F_s$. To avoid the hardware overhead of constant multipliers as mixers, this conversion frequency (corresponding to the sampling frequency in this case) is made equal to four times the carrier frequency $F_c$ of the incoming signal. Indeed, we will show that the only coefficients remaining are then sines and cosines of multiples of $\pi/2$, which are just 0, 1 or

![Figure 2.1](image-url)

Figure 2.1: Receive path for quadrature direct down conversion with a single ADC. In this example, the ADC is only preceded by a low-noise amplifier. Both in-phase and quadrature path require a mixer, a low-pass filter (LPF) and a decimation by $R$. For the mixer, a local oscillator is needed with a 90° phase splitter.
CHAPTER 2. POLYPHASE DDC FOR TI-ADC

-1. This results in the removal of mixers from the design, leaving us simply with an alternating in-phase and quadrature sub-channels as well as the introduction of some negation as will be highlighted in the next section. The presence of only I or Q sub-filter but never both for each sub-channel is also a great resource saving in terms of silicon area and power. The only condition for this to be implementable is that the number of sub-channels must be a positive multiple of 4, or \( R = 4 \cdot k \) with \( k \in \mathbb{N} \).

Right after the time-interleaved ADC comes the polyphase DDC, which will be described in more details afterwards. Nevertheless, it is important to note already that this DDC can be built using only a single clock provided a particular alignment is done at the interface between the sub-ADCs and the DDC as shown in [19]. Even when the time difference \( \Delta t = 1/RF_s \) from the last phase clock to the main clock is too small to be handled by the setup time and clock-to-Q time of the registers, it has been demonstrated that a simple fix can be implemented with two sets of aligning registers. This additional set reuses only clocks already generated for the time-interleaving of the ADCs and does not create a high overhead.

The final I and Q outputs can then be treated in a compensation FIR filter if the flatness of the passband is not sufficient for the targeted application. This filter is not considered in this work but it is expected to work as fast as the cascaded integrator comb section in any given technology thanks to pipelining as will be argued in chapter 3. The reader is referred to specific FIR filter design texts or the introduction by Lyons on CIC filters and their compensation in this regard [20].
2.2 Polyphase DDC

The polyphase DDC which is the main subject of this work is best described by its Laplace transfer function based on the equivalent cascaded integrator comb circuit. For a decimation factor equal to the number of sub-channels $R$, the resulting function of order $N$ is

$$H(z) = \frac{(1-z^{-R})^N}{(1-z^{-1})^N} = \left(\sum_{k=0}^{R-1} z^{-k}\right)^N$$

(2.1)

where Hogenauer’s differential delay $M$ (see [5]) has been taken as 1, as is often the case. In our case, this transfer function has to be broken down to consider each sub-channels $k \in [0; R-1]$. We rewrite it therefore as

$$H(z) = \sum_{k=0}^{R-1} z^{-k} H_k(z^R)$$

(2.2)

with the sub-filter equations depending on the order $N$,

$$H_k(z') = \sum_{n=0}^{N-1} c_{k,n} z'^{-n},$$

(2.3)

and $c_{k,n}$ being positive integer coefficient obtained by polynomial expansion.

For the rest of this work, the example will be carried out with $N = 4$ and $R = 8$ for which we have the coefficients in Table 2.1. For any other pair of parameters $R$ and $N$, the coefficients can easily be obtained by convolution of a vector of 1 of length $R$ with itself $N$ times as in the MATLAB snippet of Listing 2.1.

Listing 2.1: Subfilter coefficients.

```matlab
p = ones(1, R);
if N==1
    Hz_coeffs = p;
else
    Hz_coeffs = conv(p, p);
    for i=3:N
        Hz_coeffs = conv(Hz_coeffs, p);
    end
end
% c_k,n = Hz_coeffs(1+n*R+k)
```

Recombination of all the sub-channels into $I$ and $Q$ output signals can be obtained by a simple linear combination. Using the advantage from the choice $F_s = 4F_c$, the in-phase signal $I$ will only take the output of odd sub-channels and alternate sign between them as in Figure 2.3, yielding

$$I = \sum_{k=0}^{R-1} \sin\left((R-1-k)\cdot \frac{\pi}{2}\right) H_k(z') = \sum_{i=0}^{R/2} (-1)^{i+1} H_{2i+1}(z').$$

(2.4)

Similar treatment for the quadrature output but with the even channels leads to

$$Q = \sum_{i=0}^{R/2} \cos\left((R-1-k)\cdot \frac{\pi}{2}\right) H_k(z') = \sum_{i=0}^{R/2} (-1)^{i+1} H_{2i}(z').$$

(2.5)
CHAPTER 2. POLYPHASE DDC FOR TI-ADC

Figure 2.3: (a) Block diagram of the digital down-converter for \( R = 8 \) sub-channel inputs from the time-interleaved ADCs, each being filtered in a sub-filter before being combined by means of additions. (b) Implementation of the sub-filter transfer function \( H_k(z') \).
2.3. COMPRESSION TREE

Table 2.1: (a) Coefficients $c_{k,n}$ for each sub-filter $H_k(z')$ in the case $N = 4$, $R = 8$ ($k \in [0,7]$ denotes the sub-channel index, $n \in [0,3]$). (b) Canonical signed digit representation used in replacing multiplications by additions and logical shifts.

<table>
<thead>
<tr>
<th>$k$</th>
<th>$c_{k,0}$</th>
<th>$c_{k,1}$</th>
<th>$c_{k,2}$</th>
<th>$c_{k,3}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>161</td>
<td>315</td>
<td>35</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>204</td>
<td>284</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>246</td>
<td>246</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>284</td>
<td>204</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>35</td>
<td>315</td>
<td>161</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>56</td>
<td>336</td>
<td>120</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>84</td>
<td>344</td>
<td>84</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>120</td>
<td>336</td>
<td>56</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$k$</th>
<th>$c_{k,0}$</th>
<th>$c_{k,1}$</th>
<th>$c_{k,2}$</th>
<th>$c_{k,3}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000000000+</td>
<td>00+0+0000+</td>
<td>0+0+000-0-</td>
<td>0000+00+0-</td>
</tr>
<tr>
<td>1</td>
<td>00000000+0</td>
<td>0+0-0-0-00</td>
<td>0+0+00-0-</td>
<td>00000+00+0</td>
</tr>
<tr>
<td>2</td>
<td>00000000+0</td>
<td>0+00000-0-</td>
<td>0+00000-0-</td>
<td>00000+00+0</td>
</tr>
<tr>
<td>3</td>
<td>00000+00+0</td>
<td>0+00000-0-</td>
<td>0+000+00+0</td>
<td>00000000+0</td>
</tr>
<tr>
<td>4</td>
<td>0000+00+0-</td>
<td>0+0+000-0-</td>
<td>00+0+000+0</td>
<td>00000000+0</td>
</tr>
<tr>
<td>5</td>
<td>00+0000+0</td>
<td>0+0+0000+0</td>
<td>00+0000-0</td>
<td>00000000+0</td>
</tr>
<tr>
<td>6</td>
<td>00+0000+0</td>
<td>0+0+0000+0</td>
<td>00+0000-0</td>
<td>00000000+0</td>
</tr>
<tr>
<td>7</td>
<td>00+0000+0</td>
<td>0+0+0000+0</td>
<td>00+0000-0</td>
<td>00000000+0</td>
</tr>
</tbody>
</table>

If full precision is retained, the output signals will require a bit length of

$$S_{\text{out}} = S_{\text{in}} + \left\lceil (N-1) \cdot \log_2 R \right\rceil + \left\lceil \frac{R}{4} \right\rceil$$  \hspace{1cm} (2.6)

where \([\star]\) denotes the ceiling function. The two additional terms account respectively for the increase in the sub-filters of each channel and the combination of all channels together by means of additions. Note that the ceiling functions are not necessary in the case where $R$ is a multiple of 4, meaning that the full output range will be used. In our example of $N = 4$, $R = 8$ and $S_{\text{in}} = 8$, an output of $S_{\text{out}} = 19$ bits is consequently required.

2.3 Compression tree

For hardware resource optimization, the multiplications by the coefficients $c_{k,n}$ are replaced by additions and shifting operations as the coefficients are fixed when the parameters $N$ and $R$ of the down-converter are set. Canonical signed digit representation of the coefficient is used for this with a maximal amount of 3 operations required in the case of $N = 4$ and $R = 8$ as shown in Table 2.1b (additions and subtractions, shifts being only wiring). The general method is illustrated in Figure 2.4 together with the most complex coefficient for that particular case, $c_{6,1} = 344$.

The consequence of this transformation is that the full circuit consist of only additions and negations in the logical point of view. It can therefore be considered as an addition tree. These kind of trees are often encountered in multipliers after the generation of partial products and it is well known from Wallace's [22] and Dadda's [23] works that they are best treated by means of compressions until only 2 operands are remaining, those being finally combined by a full adder. Multiple optimized configurations have been reported in literature for the compression part with most notably integer linear programming work by lenne's group [12, 24–26], the three dimensional method...
Figure 2.4: (a) Generic circuit replacing multiplication by a constant coefficient with additions and shifts. (b) Practical example of (a) for \( c_{6,1} = 344 \).

[11] and its extension as the three-greedy method [13]. After tests and extensive comparisons, it was decided to go on with the three-greedy method that uses principally full adders (or 3:2 counters in the generic notation). Larger counters or generalized compressors such as 5:5:4 can be very advantageous on field programmable gate arrays (FPGA) if they match the lookup table sizes for instance but do not present such a big advantage on application specific integrated circuits (ASIC) where the synthesis tool will anyway break them down to smaller gates. For the final adder, it is clear that a fast parallel prefix adder such as a Kogge-Stone should be used. These are in general very well handled by the synthesis tools though and thus do not need to be implemented manually in the register transfer level (RTL) code.
Circuit generation and optimization

The full structure of the circuit has already been given in Figure 2.3. Implementing it as is in structural VHDL code is straightforward and can serve as a reference starting point. It is however not the best way of doing it and building an optimized compression tree with judicious placement of full adders is the focus of this chapter. Synthesis with automated pipeline registers placement and place-and-route scripts targeting high-speed designs are also given.

3.1 Structural generation

To build the optimal compression tree, a full compression tree containing all elements is first created and then simplified according to logical rules. At this step, compressors are placed using the three-greedy method using a specifically developed C++ code implementing the algorithm proposed by Stelling et al [13] with modifications to reduce run-time. The full procedure is as follows.

3.1.1 Tree generation and reduction

First, starting from Table 2.1b, an initial addition tree is generated with one line per non-zero symbol in the canonical signed digit table. Negative numbers are already transformed at this stage respecting the two’s complement representation (by inverting all the bits and adding 1). All lines created with this procedure are then extended to the required length at the output by repeating the most significant bit (MSB) bit towards the left. In the case of $N = 4$ and $R = 8$, the total number of lines at this stage is 42 for the $Q$ tree and 34 for the $I$ tree. All of them are not required though as, for each coefficient having more than one non-zero symbol in its CSD representation, same bits of information are placed multiple times. Knowledge of this correlation can consequently be used to perform simplifications in the circuit even before placing full adders.

Indeed, we know that addition of two bits $x$ and $y$ corresponds to the following logical operations:

\[ s = x \oplus y; \quad \text{and} \]
\[ c = x \cdot y, \]

where $s$ is the sum bit (same weight) and $c$ the carry bit (double weight).
As such, when a single column contains either two times the same bit or the constant 1, the simplification rules from [27] can be used with the sum result $s$ being placed in the same column and the carry output $c$ in the next one on the left. The rules are:

$$s = x \oplus x = 0 \quad \text{and} \quad c = x \cdot x = x; \quad \text{or} \quad (3.3)$$

$$s = x \oplus \overline{x} = 1 \quad \text{and} \quad c = x \cdot \overline{x} = 0; \quad \text{or} \quad (3.4)$$

$$s = 1 \oplus 1 = 0 \quad \text{and} \quad c = 1 \cdot 1 = 1; \quad \text{or} \quad (3.5)$$

$$s = x \oplus 1 = \overline{x} \quad \text{and} \quad c = x \cdot 1 = x; \quad \text{or} \quad (3.6)$$

$$s = \overline{x} \oplus 1 = x \quad \text{and} \quad c = \overline{x} \cdot 1 = \overline{x}. \quad (3.7)$$

The three first rules represent a simplification of two similar bits ($x$ with $x$ or 0 with 0, or 1 with 1) leading to a reduction in the number of elements kept ($0$ can be removed). The remaining two consist of a propagation to the next column by combining a 1 together with another bit. Consequently, it is not surprising that best results from this reduction rules have been found when the reduction is first done for all similar bits and only then combination of a constant 1 with a bit already appearing in the next column is done. The procedure is looped as long as some reduction happened on the tree as the propagation resulting from the combination of a 1 with an element can allow further optimization. The algorithm is therefore similar to the pseudo-code of Listing 3.1.

**Listing 3.1: Reduction algorithm.**

```c
bool reduce() {
    do {
        reduce_sames(); // reduces all similar bits
    } while (reduce_propagate_1()); // use single 1s to propagate bits
    // already present in next column
}
```

The resulting final tree is given in Figure 3.1 for the $Q$ path (which, in this case, happens to be more complicated than the $I$ path for which all coefficient are multiples of 4, meaning that column 0 and 1 are always 0 and can be dropped). The maximal height is of 40 elements for column 8, not much smaller than the original 42 height but more significant bit columns were largely compacted.

### 3.1.2 Three-greedy implementation

Starting with the reduced addition tree, full-adders (or 3:2 compressors) are placed from the least significant bit (LSB) column up to the most significant (MSB) column in order according to the original three-greedy procedure [13]. When treating one column, each input bit is assigned a delay ($d_x = 0$ when coming from the ADC, even when inverted, as inverted outputs are often available from flip-flops). The basic principle of the method is then to propagate the delay from inputs to outputs of the compressors according to the rules in Figure 3.2 to try and reach the minimum delay profile for the last two numbers which will be fed to the unique full adder.

For this, all the bits in the input list for that column are ordered in increasing delay. In the case where the number of bits in the input list is odd, a half adder is first placed taking the two first bits in the list to make it even. Otherwise, the procedure can start...
Figure 3.1: Addition tree for $Q$ path in the case $N = 4$, $R = 8$ and $S_{\text{in}} = 8$. A bullet • represent one bit coming at time $t = 0$ from the ADC while an overlined bullet • is a bit that needs to be inverted compared to the data coming from the ADC (they do not all represent the same bit of course).

\[
d_s = 1 + \max(d_0, d_1) \quad \quad (3.8)
\]
\[
d_c = 0.5 + \max(d_0, d_1) \quad \quad (3.9)
\]

(a) Half Adder (2:2 compressor)

\[
d_s = \max(2 + d_1, 1 + d_2) \quad \quad (3.10)
\]
\[
d_c = 1 + d_2 \quad \quad (3.11)
\]

(b) Full Adder (3:2 compressor)

Figure 3.2: Half adder and full adder blocks with delay propagation rules in units of XOR delay in the technology used, assuming $a_0$ to $a_2$ are ordered in delay, $d_0 \leq d_1 \leq d_2$. 
repeatedly as full adders will take 3 inputs from the list and re-add a new one at the end (the sum bit \( s \)) and the number will stay even. It will be applied, respecting the rules outlined thereafter, until only two bits are remaining in the list. The first rule is to make the full adder two greedy, ie. it should take for its first two input \( a_0 \) and \( a_1 \) the two first elements in the list while the third one \( a_2 \) can have a higher delay.

**Rule 1** (Two greedy). A compressor always takes as its first two inputs \( a_0 \) and \( a_1 \) the two first bits in the input list ordered in increasing delay values.

This creates a lot of solutions though, even when keeping only one version of equivalent solutions (all new inputs having same delay). Therefore, we use the lemmas 3.1, 3.4, 3.6 and 3.7 of [13] to exclude some solutions that can be shown to be no better than the best one meeting the criteria of those lemmas. Rules from these lemmas are repeated here for convenience:

**Rule 2** (Topologically ordered). For a full-adder \( j \) placed after a full adder \( i \), the delays should meet

\[
\begin{align*}
d_{i,0} &< d_{j,0}; & \text{or} \\
d_{i,0} &= d_{j,0} \quad \text{and} \quad d_{i,1} < d_{j,1}; & \text{or} \\
d_{i,0} &= d_{j,0} \quad \text{and} \quad d_{i,1} = d_{j,1} \quad \text{and} \quad d_{i,2} \leq d_{j,2}.
\end{align*}
\]

**Rule 3** (\( d_2 \) ordered). For a full-adder \( j \) placed after a full adder \( i \), the delays \( d_2 \) should be ordered, ie

\[
d_{i,2} \leq d_{j,2}.
\]

**Rule 4** (\( d_{i,2} \) maximal value). For a full-adder \( j \) placed after a full adder \( i \) for which \( d_{i,2} \leq d_{j,1} \), \( d_{i,2} \) is limited by

\[
d_{i,2} \leq d_{j,0}.
\]

**Rule 5** (\( d_{j,2} \) maximal value). For a full-adder \( j \) placed after a full adder \( i \) for which \( d_{j,1} < d_{i,2} \leq d_{j,2} \), \( d_{j,2} \) is limited by

\[
d_{j,2} < d_{i,2} + 1.
\]

After each placement of a full adder, any solutions dominated by another can be removed, where by dominated it is meant that all the delays for each columns are higher or equal to the ones from the dominant solution. Nonetheless, the number of solutions kept with this method (which is the original three-greedy) is still relatively high to be handled in a reasonable time by a normal desktop computer and reduction of the solution set is advisable. For this, a bound is put on the delay profile (maximal delay of the two bits from each columns fed to the final full adder) and all solutions exceeding that bound in any column are disregarded. From Table 2 in [13], the maximal bound should be no higher than 12 for the \( Q \) path and 11 for the \( I \) path (having respectively 40 and 34 elements in their highest column). If the search is started with over-constrained delay profiles and relaxed step by step as long as no solutions are found by the three-greedy algorithm, the optimal solution can be found in reasonable time. The final delay profiles in our example are those in Table 3.1 where they have
Table 3.1: Delay profiles for $I$ and $Q$ path of the addition tree before entering the final 2-input full adder. Delays are expressed in units of typical XOR delay in the technology.

<table>
<thead>
<tr>
<th></th>
<th>col. 17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Q$ tree</td>
<td>10</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>11</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$I$ tree</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2: Number of half or full adders placed in each column of the $I$ and $Q$ compression trees.

<table>
<thead>
<tr>
<th></th>
<th>col. 17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Q$ tree</td>
<td>9</td>
<td>11</td>
<td>15</td>
<td>18</td>
<td>22</td>
<td>26</td>
<td>29</td>
<td>31</td>
<td>35</td>
<td>36</td>
<td>33</td>
<td>29</td>
<td>25</td>
<td>21</td>
<td>17</td>
<td>12</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>$I$ tree</td>
<td>5</td>
<td>7</td>
<td>10</td>
<td>16</td>
<td>20</td>
<td>25</td>
<td>28</td>
<td>32</td>
<td>31</td>
<td>28</td>
<td>24</td>
<td>20</td>
<td>16</td>
<td>12</td>
<td>7</td>
<td>3</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

been optimized using this methodology to have the predetermined maximal bound in the most significant column possible. The number of half or full adders placed in each column from this solution is given in Table 3.2. Note that, from the construction of the filter, overflows do not appear and therefore last column is composed only of XORs and is not included in the table.

The complete algorithm can be summarized with the pseudo-code in Listing 3.2 and the template for the three-greedy function for a single column ($tg\_reduce\_1\_col$) in Listing 3.3.

Listing 3.2: Generation procedure.

```c
// for each columns from LSB to MSB
for(unsigned int col_i(0); col_i < Sout-1; ++col_i)
{
    // create empty solution set
    Solutions new_sols;

    // solve for all current solutions
    for(unsigned int i(0); i < solutions.size(); ++i)
    {
        // adjust parity in case of odd number of elements
        if(inputs[i].size()%2 == 1)
            solutions[i].adjust_parity(); // add half-adder

        // apply three-greedy method on column col_i and solution i
        // and store all new solutions
        new_sols.add(tg_reduce_1_col(inputs, col_i, delay_bound[col_i],
            solutions[i]));
    }

    if(new_sols.size() ==0)
        return; // no solutions

    new_sols.remove_dominated_and_equivalent();
}
```

solutions = new_sols; // update solution set
}

Listing 3.3: Three-greedy algorithm for one column.

```c
Solutions tg_reduce_1_col(inputs, col_i, delay_bound, solution)
{
    Solutions solutions;
    // done with column if less than 3 elements remaining
    if(inputs[col_i].size() < 3)
        return solution;
    // inputs[col_i][0] and [1] are taken (two-greedy)
    // all others need to be tested
    for(unsigned int i (2); i < inputs[col_i].size(); ++i)
    {
        // do not test equivalent solutions
        if(i != 2 && inputs[col_i][i].delay == old_delay)
            continue;
        old_delay = inputs[col_i][i].delay;
        // test if inputs[col_i][i] is valid
        if(!respect_lemmas(inputs[col_i][0], inputs[col_i][1], inputs[col_i][i], → solution)
            continue;
        // compute delays and place next full-adder recursively
        new_inputs = update_inputs(inputs, col_i, i);
        new_solution = add_FA(solution, col_i, i);
        new_sols = tg_reduce_1_col(new_inputs, col_i, delay_bound, new_solution)
        → ;
        new_sols.remove_dominated_and_equivalent();
        solutions.insert(new_sols);
    }
}
```

3.2 Synthesis and pipelining

Once the RTL code is defined from the output of previous section, the only critical point remaining is the definition of the pipelining structure and length. It is not recommended to place pipelining registers manually in the design as they will end up with unbalanced delays and the overall maximal sustainable frequency for the circuit will be limited by the longest one. Allowing them to move during mapping and synthesis to equate all delays is therefore the route to follow and this is indeed a recommendation from Synopsys in their *Coding Guidelines for Datapath Synthesis* [28] (Synopsys Design Compiler 2013.03 being used for synthesis). The procedure for this is to place all pipelining registers in the end of the design and then just after compilation refine their placement for optimal timing with the *optimize_registers* command as in the tcl extract from Listing 3.4.
3.2. SYNTHESIS AND PIPELINING

Listing 3.4: Registers placement optimization.

```plaintext
# elaboration, constraints, etc
# ...

# compilation
puts "-i Compile_design"
ungroup -flatten -all
compile_ultra

report_resources -nosplit -hierarchy > RPT/$[DESIGN]_resources.rpt

# Pipeline optimization (keeping in/out registers)
puts "-i Optimizations"
set_dont_retime [get_cells "*out_I_reg_reg *"] true
set_dont_retime [get_cells "*out_Q_reg_reg *"] true
set_dont_retime [get_cells "*delay_reg_reg[0]*"] true
optimize_registers

# ...
# Verilog/SDF export, reports, etc
```

For interfacing with other blocks, input and output registers are kept as first, respectively last, logical blocks with the `set_dont_retime` command. Note that resources report for the type of final full adder used by the synthesis (which should be a parallel prefix adder for high speed) has to be done before re-optimization for correct results while other reports such as area, power, timing and list of references are better done after that step as they will be modified by the retiming operation.

Leaving the placement of registers in the design to the automated tool, only the pipeline length has to be chosen. Obviously, increasing the length from a combinational design starting point (equivalent to length 0) will reduce the delay while increasing area to account for the new sequential cells inserted. This trade-off will continue for a while up to a deflection point where only one or a few gates compose the logical path between two registers. Logic cannot be broken into smaller pieces anymore and adding more registers will only increase area. Area at this point is of course dependent on the RTL design synthesized, but delay and frequency are theoretically only limited by the technology and standard cell library used. The bottom limit on delay is indeed set by gate delays in addition to setup and clock-to-Q times of flip-flops, as will be highlighted in chapter 4 on TSPC flip-flops. In our example design (for \( R = 8, N = 4 \) and \( S_{in} = 8 \)), the optimal pipeline length is obtained around length 10 as illustrated in Figure 3.3 and this is the value that will be used. Negative slack of 110 ps on 100 ps clock constraint could theoretically be achieved, leading to \( f_{max} = 1/210 \text{ps} = 4.7 \text{GHz} \) maximal clock frequency. This value is however obtained without any wire-load from routing and is therefore greatly inflated as a first estimation. Nevertheless, this diagram justifies the choice of considering only one set of parameters for the design as others would be very close in terms of speed.

The rest of the flow in the complete script (see section A.1) follows standard procedure, similar to Vachoux’s tutorial [29] for instance. In the script given in appendix, standard DFF cells are marked `don’t use` and a custom library is included to force the use of custom designed TSPC flip-flops as explained in chapter 4.
CHAPTER 3. CIRCUIT GENERATION AND OPTIMIZATION

Figure 3.3: Finding the optimal pipeline length for fastest clock frequency on a synthesis for a clock constraint of 100 ps. Important to notice is that the lower limit on delay is technology dependent and not design dependent as can be seen by varying the parameters of the filters \((R = 8\) for all curves). Note that these are results on a version of the compression tree not using the latest three-greedy implementation but for which parameter variations were studied; values are very similar to \(N = 4\) and \(S_{\text{in}} = 8\) though for the final RTL used and length 10 was kept.

3.3 Place and Route

The place and route script (see section A.2) follows standard procedure as well, with the exception that the first and last rows in ST 28 nm FDSOI need to be made of fillers only to pass design rule check (DRC) rules. Note that no IR drop analysis was performed but it would probably have been better in terms of reliability to put larger power rings around the design and more stripes in the middle. These should have been made in higher metal levels as metal M1 and M2 as done in this script, which is included as used for reference and was not modified and rerun due to time constraints. Clock skew could be introduced at this point for fine-tuning of slack on all delay paths but this technique was finally not used as tests were not as convincing as expected (refer to subsection 6.2.4 for more explanations).

3.4 Note on wire-load modeling

Wire-load modeling and capacitance tables were not available in the process design kit (PDK) installation during the design procedure. Synthesis as well as place and route were therefore performed under zero wire-load model which imposes zero load from
3.4. *NOTE ON WIRE-LOAD MODELING*

Table 3.3: Results for DDC designs with $N = 4$, $R = 8$ and $S_{in} = 8$ under typical process conditions but 0.8 V supply and 125°C. A pure structural VHDL (using plus signs in the code) used as reference is compared to the optimized compression tree design showing 30% improvement in speed for low area increase. Both use a pipeline length of 10.

<table>
<thead>
<tr>
<th>Design</th>
<th>Core area [µm²]</th>
<th>Max. frequency [GHz]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference</td>
<td>13864</td>
<td>2.8</td>
</tr>
<tr>
<td>Three-greedy compression tree</td>
<td>14298</td>
<td>3.7</td>
</tr>
</tbody>
</table>

These wire-load issues also showed some contradictory results at first with high variability. For instance, the reference design based on Figure 2.3 showed after synthesis a slack almost similar to the three-greedy optimized compression tree design. The reason is clearly a deeper routing structure already included in the VHDL component instantiations when using only 1-bit full-adder blocks compared to when full-adders on 17 or 19 bits are used. As such, delay degradation is worse for the reference design after import in Cadence Virtuoso [32] and simulation with Spectre [33] including parasitic caps. In the end, the three-greedy method indeed shows much better results (30% improvement for same pipeline length) for a limited increase in area as shown in Table 3.3: under typical process conditions but worst case voltage supply (0.8 V instead of 1 V) and temperature (125°C), maximal sustainable frequency went from 2.8 GHz up to 3.7 GHz.
Dynamic flip-flops

For a very high speed deeply pipelined design, sequential elements have an important impact on key metrics such as area, power and speed. In particular, for this design where the goal is to reduce delay as much as possible, any improvement on timing characteristics of the flip-flops used is highly valued and could push the delay limit in pipelining further down. Traditionally available in standard cell libraries are static types of flip-flops; the structures used by ST microelectronics make no exception. Dynamic flip-flops have an unbeatable advantage regarding delay though and the most used is the true-single-phase clock (TSPC) one by Yuan and Svensson [16]. Therefore we show how this cell has been adapted for ST 28 nm technology and became fully compatible with their 8-track standard cell library for use in the semi-custom flow presented in last chapter.

4.1 TSPC flip-flops generics

The classical static flip-flop as used in ST’s standard cell library uses clock-enabled inverters with 2-inverter loops for data storage as in Figure 4.1. This cell is positive edge triggered on a signal CP, which passes through inverters to create first CPN and then CPI for internal use. Reset is active low from signal RN and RNN is the inverted version of this signal. With the 26 transistors used internally for logic, the total number of transistor per cell is 32 with those inverters for signal generation.

The TSPC cell [16] on the other hand uses a single clock signal internally (CNN) and is based on the usual precharge-evaluate alternating phases of dynamic logic structures. In the schematic of Figure 4.2, precharge happens when CNN is low for the output of the second stage (node D2) and evaluation is done when it switches to high value. If CNN is taken as the output of an inverter and not directly from the clock distribution network to avoid adding too much load to it, this flip-flop will be negative edge triggered. Reset has the same polarity as in the regular DFF, i.e. it is active low. This brings the number of transistor to 17 (13 internally and 4 for CNN and RNN inverters).

Another slight variant of this design is the double precharge TSPC DFF (DP TSPC) [17] which precharges both first and second stages (nodes D1N and D2). This version can be made faster than the traditional TSPC flip-flop while requiring one less transistor in theory (see Figure 4.3a). Its disadvantage is that it creates a racing condition between first and second stages when D is high which can lead to drops in voltages and increase in leakage currents in the following stages. Three additional transistors acting
CHAPTER 4. DYNAMIC FLIP-FLOPS

Figure 4.1: Schematic of the standard static CMOS D flip-flop using 26 transistors internally and 6 for CPN, CPI and RNN generation.

Figure 4.2: Schematic of the TSPC D flip-flop using 13 transistors internally and 4 for CNN and RNN generation.
4.2. DESIGN AND METRICS

Figure 4.3: Schematic of (a) the original double precharge TSPC D flip-flop with added reset capability and (b) its modification with keepers to reduce power consumption using respectively 12 and 15 transistors internally and 4 for CNN and RNN generation.

as keepers as in Figure 4.3b can be added to restore correct $V_{DD}$ voltages to the nets without perturbing the speed of evaluation. Detailed working principle of those cells is explained in the following sections together with the sizing and key timing metrics.

4.2 Design and metrics

4.2.1 TSPC flip-flop

The main advantage of the TSPC DFF comes from the precharging of node D2 during the low phase of clock CNN. This yields a very short evaluation path on the rising edge of CNN. Referring to Figure 4.2, $D = 1$ evaluation (or low-to-high transition) requires D2 to be high as well which is already the case and therefore is only limited by the two NMOS in series of the third stage. $D = 0$ evaluation (high-to-low transition) is made by the two NMOS in series of the second stage which discharge node D2 and allow the PMOS of the third stage to charge D3N. These five transistors are therefore the only
Table 4.1: Schematic (sch.) and parasitic-extracted (etxr.) delays in ps for the three different flip-flops considered, measured at 5 ps input rise/fall time and 0.2 fF load with setup taken for 10% increase in clock-to-Q time and hold as failure at 100 MHz clock. Important metric for high-speed design is the sum of clock-to-Q and setup for which TSPC and DP TSPC are respectively almost 50% and 70% better than static version.

(a) Falling edge.

<table>
<thead>
<tr>
<th>metric (in [ps])</th>
<th>ST (static)</th>
<th>TSPC</th>
<th>DP TSPC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sch.</td>
<td>extr.</td>
<td>sch.</td>
</tr>
<tr>
<td>clk-to-Q</td>
<td>38.5</td>
<td>46.8</td>
<td>22.6</td>
</tr>
<tr>
<td>setup</td>
<td>13.2</td>
<td>15.0</td>
<td>3.4</td>
</tr>
<tr>
<td>clk-to-Q + setup</td>
<td>51.7</td>
<td>61.8</td>
<td>26.0</td>
</tr>
<tr>
<td></td>
<td>-49%</td>
<td>-48%</td>
<td>-75%</td>
</tr>
<tr>
<td>hold</td>
<td>-5.4</td>
<td>-0.2</td>
<td>2.5</td>
</tr>
</tbody>
</table>

(b) Rising edge.

<table>
<thead>
<tr>
<th>metric (in [ps])</th>
<th>ST (static)</th>
<th>TSPC</th>
<th>DP TSPC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sch.</td>
<td>extr.</td>
<td>sch.</td>
</tr>
<tr>
<td>clk-to-Q</td>
<td>31.6</td>
<td>47.6</td>
<td>21.0</td>
</tr>
<tr>
<td>setup</td>
<td>11.4</td>
<td>10.3</td>
<td>-4.0</td>
</tr>
<tr>
<td>clk-to-Q + setup</td>
<td>43.0</td>
<td>57.9</td>
<td>17.0</td>
</tr>
<tr>
<td></td>
<td>-60%</td>
<td>-61%</td>
<td>-67%</td>
</tr>
<tr>
<td>hold</td>
<td>-8.6</td>
<td>-9.0</td>
<td>7.5</td>
</tr>
</tbody>
</table>

critical ones for speed and their sizes can be increased while the others can be kept small (for a fixed size of the output inverter serving as buffer). Lowest delay and good symmetry between high-to-low and low-to-high transitions is found for $W_n \approx 110 \text{nm}$, $W_p \approx 3 \cdot W_n$ and the 5 critical transistors having their size doubled. Signals during transitions are shown in Figure 4.5 (in red) and show more than 30% improvement in setup and clock-to-Q times (see Table 4.1) compared to ST’s C8T28SOI_LL_DFPRQX19_P0 standard cell from library C28SOI_SC_8_CORE_LL which uses the same strength X19 inverter as buffer for the Q output. Thanks to the important reduction in the number of transistors used, area of the standard cell is also greatly shrunk with the layout in Figure 4.4b using only 11 poly columns (including the dummies on both sides) versus 15 for ST’s equivalent cell (see Figure 4.4a).

4.2.2 Double precharge TSPC flip-flop

The double precharge cell tries to improve even more the delay from D to Q by realizing that the slower transition is generally the high-to-low for which D2 needs to discharge. The solution employed for this is to start discharging D2 in any case and only stop this discharge when $D = 1$; solution achieved by precharging D1N as well. The issue is then to correctly control the race condition to make sure that D1N discharge will stop the discharge of D2 soon enough for correct and fast evaluation at stage 3. This works well
4.2. DESIGN AND METRICS

Figure 4.4: DFF standard cell layouts with output buffer strength X19 compatible with ST 8-track standard cell libraries. Dynamic versions on the bottom row to be compared with the static DFF cell C8T28S0I LL DFPRQX19 P0 from library C28S0I SC 8 CORE LL on the top row. Poly lines (in red) are separated by 136 nm and cell height is 1.6 µm.
Figure 4.5: Schematic simulations of the different flip-flops on a 5 GHz clock, including internal nodes for the ones based on the TSPC structure. Inputs are common for all (with the polarity of CLK reversed for the static flip-flop to make it negative-triggered as well). Improvements in delay for dynamic versus static is already clearly visible on Q output. All vertical scales are in volts with $V_{DD} = 1$ V.

if the NMOS transistor with input D is made the same size as the one on D1N. Moreover, both NMOS of third stage as well as the common clock transistor for stage 1 and 2 are made $W_n \approx 440 \text{ nm}$ to reach similar clock-to-Q as the usual TSPC cell. As shown in Figure 4.5 at the first transition (in light green, around 80 ps), the problem with this structure is that node D2 can drop a bit during the time it takes D1N to go low and the drop can be large enough when using low threshold voltage (low $V_T$, or LVT) transistors to induce relatively high leakage on stage 3 (the effect is even worse on extracted simulation than on those schematic simulations). The solution to this is a keeper that restore $V_{DD}$ voltage on node D2 when D3N is low. To avoid impacting the delay, it is placed in series with a transistor on D1N to make sure it is active only when evaluation has already begun in the right direction. Note that this transistor on D1N alone would not be enough as it could induce a short in reset mode. Finally, in extracted simulations, D1N showed a drop on the rising edge of CNN due to parasitic caps of stage 2 NMOS transmitting the discharge of the common node between stage 1 and 2 (this drop is already noticeable on schematic simulation from Figure 4.5 at the second transition around 280 ps and, once again, is much worse on extracted simulations). To compensate for this, and considering space limitation in layout, the best solution was to include a shorted NMOS of size $W_n = 222 \text{ nm}$ on node D1N which imposes a simi-
Table 4.2: Results for DDC designs with $N = 4$, $R = 8$ and $S_{in} = 8$ under the same conditions as before: typical process, 0.8V supply and 125$^\circ$C. Standard static DFF cell from Table 3.3 (three-greedy compression tree row) is used as reference and compared to the TSPC designs with or without wire-load. All design use a pipeline length of 10.

<table>
<thead>
<tr>
<th>Design</th>
<th>Core area $[\mu m^2]$</th>
<th>Max frequency [GHz]</th>
<th>diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDC</td>
<td>14298</td>
<td>3.7</td>
<td></td>
</tr>
<tr>
<td>TSPC</td>
<td>11807 $-17%$</td>
<td>3.8</td>
<td>2%</td>
</tr>
<tr>
<td>TSPC wl</td>
<td>15060 $5%$</td>
<td>4.3</td>
<td>16%</td>
</tr>
</tbody>
</table>

lar change on D1N by parasitic caps as well but in the opposite direction. The cleaner signals obtained this way are given in Figure 4.5 (in dark green) with the layout in Figure 4.4c. Cell area is in-between ST’s static cell and TSPC cell with 13 poly columns and clock-to-Q times are similar to TSPC while setup times are furthermore improved leading to a sum of these two metrics about 30% better than the TSPC (see Table 4.1).

In both TSPC and DP TSPC, hold times are worse than in the standard static DFF. This is not a problem though as hold fixes are easily handled by synthesis or place-and-route tools.

4.3 Characterization and semi-custom flow integration

TSPC and double precharge TSPC have already been characterized in Table 4.1 for one specific load and input slew for comparison with the standard static DFF. For complete integration in the semi-custom flow, a more complete characterization is needed though and this is better done in a specialized software. Therefore, SPICE-extracted netlists of the different layouts were created and passed to Cadence Liberate [35] with the script in section A.3 for automated characterization with the same parameters as the static cell extracted from ST’s library for input slew and output load. The results from this characterization run confirmed the great improvements already noticed. For instance, the TSPC cell when compared to the static one re-extracted and re-characterized using the same flow shows an average of 30% improvement on clock-to-Q and a change of sign (200% improvement) on setup time constraint. General metrics are not taken from there though as some points measured are clearly not representative and will not be used in our design (such as a clock skew of 1.325 ns when we target more than 3 GHz clock frequency).

Nevertheless, .lib files can be created as such and imported into Synopsys Design Compiler (after trivial conversion to .db format) and Cadence Encounter for place and route. The place and route tools also require the knowledge of cell geometry as well as already present routing in metals M1 or M2 that will block global routing. For this, LEF files are exported from Cadence Virtuoso [32] (using the abstract generator tool) and can be included in the Encounter setup at the same time as the global libraries. Surprisingly, the first version of the DDC design synthesized and placed with these new cells did not show much improvement in terms of speed (but did in terms of area) as shown in Table 4.2. The reason for this was a too large load at the output of the new
TSPC flip-flops even though they use the same output buffers as the standard ones. Two solutions were used for this. Firstly, the maximal output capacitance in the `.lib` file was reduced from what Liberate found down to what ST’s library was using (i.e. by a factor of two). Secondly, a second synthesis pass was performed with extracted wire-load information from the first placed and routed design. A real gain in frequency was observed this time, reaching an impressive 4.3 GHz. This had the consequence of inserting larger gates or buffers almost everywhere though and therefore lost the benefits in terms of area. It is highly probable thus that there remains a difference in the creation of the `.lib` file between ST’s procedure and my TSPC characterization.
Test-interface for tape-out

Post-layout designs were intensively simulated not only with SDF-annotated digital simulation as is often done but also with full analog simulation from extracted view in Cadence Virtuoso. As such, confidence is relatively high (assuming the models are correct) that the working frequency of the best DDC circuit would overpass 4 GHz which is already impressive for a semi-custom design. Nevertheless, the confirmation of the gains expected from design architecture optimization and TSPC cell development can only come from real production and Silicon measurements. As such, a design with two DDC cores (one with only ST’s standard cells and one including TSPC flip-flops) and a few test blocks was encapsulated in a test structure for tape-out.

5.1 Top-level architecture and layout

The working frequency for the main DDC blocks is too high for direct interfacing with I/O pins and would also require too many pins in parallel. Consequently a memory interface with serial writing (reading) and parallel reading (writing) was added around the DDC blocks to feed the inputs (read back the outputs). A frequency divider (described in next section) was also added to test the maximal frequencies of the flip-flops and two ring oscillators were also packaged in one test block. The overall structure is therefore the one in Figure 5.1 and consists of:

**DDC STD**  One DDC using only ST’s standard cells, which is the last row of Table 3.3 and is expected to work up to 3.7 GHz.

**DDC TSPC**  One DDC using TSPC flip-flops, which is the last row of Table 4.2 and is expected to work up to 4.3 GHz.

---


3 The ring oscillator block is not used for the design in this report but just added to benefit from the same tape-out and test structure. It consists of two programmable ring oscillators, one with very classical inverters, and one with cascoded inverters and regulated gate and body biases. The goal is to measure the impact of drain-source voltage on jitter noise.
CHAPTER 5. TEST-INTERFACE FOR TAPE-OUT

Figure 5.1: Toplevel block diagram of the structures under test with the test controller and interface to the chip pads for probing with a SPI slave module. Data flow in bold lines, clock and configuration signals in thin lines. \( \text{clk}_\text{low} \) and \( \text{clk}_\text{high} \) are low and high frequency clock inputs. Slave select negative ss\(_n\), master-out/slave-in MOSI and master-in/slave-out MISO are SPI signals. sig\(_n\) is an output from the test blocks RO and DIV. Finally, bidir\(_{io\_en}\) is used to toggle the direction of a bidirectional pin combining MOSI (input) and MISO or sig\(_n\) (output) depending on the state.

**MEM in/out** Input and output memories for serial to parallel conversions for the DDC blocks. For space considerations these have been made for 16 test vectors before requiring reprogramming (which allows for any test length anyway as the pipeline length is of only 10). Their respective sizes are therefore \( 16 \cdot R \cdot S_{\text{in}} = 1024 \text{bits} \) and \( 16 \cdot 2 \cdot S_{\text{out}} = 608 \text{bits}\). Note that in the taped-out design the memories were created using an address-selectable RTL similar to random access memory (RAM) but that shift registers would probably be easier to implement and to make sure that they would work at high frequency.

**DIV** One frequency divider to compare ST and TSPC flip-flops. It contains one fast ring oscillator and is programmable to use either static or dynamic flip-flops to divide its high frequency output. This block is described in more details in the following section.

---

1. Reduced to \( 16 \cdot (2S_{\text{out}} - 2) = 576 \text{bits} \) effectively as I output is a multiple of 4, i.e. it has bits 0 and 1 always tied to ground.
RO One ring oscillator block for noise measurement (not related to this project/report).

Test interface One test interface controlling when each other block is activated and programming each block with the correct set of parameters depending on internal registers that can be written externally.

SPI slave One serial peripheral interface (SPI) slave for external communication and programmability of the registers inside the test interface. This slave is used for writing the input memory and reading back the output memory as well.

Figure 5.2: Toplevel layout of the taped-out design on the left with the insert on the right zooming on the part containing the logic. DDC cores and RO and DIV blocks have been highlighted, the rest of the insert consists of the test interface, the memories and some power routing and fillers. Decoupling capacitors on the top link $V_{DD}$, $V_N$, $V_P$, $V_{bn}$ and $V_{bp}$ to $V_{GND}$ for supply filtering. On the bottom left, metals that will connect to the input/output ring are noticeable.

The final layout taped-out is shown in Figure 5.2. The two DDC cores account for the majority of the area used and have been placed on top of each other. The RO and DIV blocks are placed on the left of the DDC cores, respectively on the very top and the very bottom; they are very small anyway. The space around is used by the test interface and the memories, together with the main routing (primarily power routing). On top, decoupling capacitors have been used to fill the space remaining and filter as much as possible supply noise.
5.2 Frequency divider

One of the major improvements in this work is the switch from static to dynamic flip-flops; it would also benefit other high-speed designs. As such, testing the flip-flops on their own is quite important and can provide a better understanding of the relative gains. This test can further act as a verification of the modeling of dynamic or charge-based structures.

Figure 5.3: Clock frequency dividers realizing a division by 2 using a single or two edge-triggered flip-flops. If only a single polarity is available at the output (Q or QN), an inverter can be used in the feedback loop as one inversion is required. Output clock can be taken at Q or QN without importance and at first or second stage when using two flip-flops. Reset RN acts as an enable signal.

The most obvious block for this test is a frequency divider as it is quasi-exclusively made of flip-flops. Its maximal working frequency is also a direct quantification of the sum of setup time and clock-to-Q time, reflecting perfectly the important metrics for high-speed designs. Figure 5.3 shows the basic divide-by-2 block based on a single flip-flop or on two flip-flops. The first version is the classical one which is the traditional way of realizing a frequency division by 2. The second version is in fact a division by four but doubles the input frequency by using the two edges of the input clock. It is used only to double the frequency constraint and extend the range of the ring oscillator used as frequency generation. It must be noted that the duty cycle should be close to 50% for the constraint to be effectively doubled and the delay of the single inverter for the first flip-flop should be matched with the two inverters leading to the second flip-flop. One inversion (path from QN to D) is always included in the loop for correct functionality. One version of each was added in both static and dynamic flip-flops (TSPC only as DP TSPC would require special treatment to account for hold time which is approximately equal to setup plus clock-to-Q time), yielding four different division paths with only one of them active at each time. Direct output from the ring oscillator is also selectable as fifth possibility to probe the input frequency. The output is then divided by a factor between 4 and 32 before being connected to the output pin as the frequency would be too high for output pins otherwise.

Maximal working frequencies expected from simulations in Cadence Virtuoso are given in Table 5.1, but a warning should be given that these are highly dependent on process, voltage or temperature corners.
Table 5.1: Maximal working frequencies in simulations for the frequency divider blocks. Simulations were performed in the typical corner at 125°C to account for self-heating which would occur at high switching frequency.

<table>
<thead>
<tr>
<th></th>
<th>static</th>
<th>TSPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>schematic</td>
<td>15 GHz</td>
<td>35 GHz</td>
</tr>
<tr>
<td>extracted</td>
<td>12 GHz</td>
<td>27 GHz</td>
</tr>
</tbody>
</table>

5.3 Test procedure

To test the different blocks, they first need to be programmed through the SPI slave interface. For this, the low frequency SPI clock (SCLK) is used with the master-out/slave-in (MISO) line. A programming sequence is started by bringing the slave select line (SS_N) low during a high phase of SCLK. Direction and address of the test register is then provided in the first 8 bits and data comes next in chunks of 8 bits as well. Example transactions are given in Figure 5.4 with the following first 8 bits:

<table>
<thead>
<tr>
<th>A7</th>
<th>A6</th>
<th>A5</th>
<th>A4</th>
<th>A3</th>
<th>A2</th>
<th>A1</th>
<th>A0</th>
</tr>
</thead>
<tbody>
<tr>
<td>T&lt;sub&gt;]&lt;/sub&gt;</td>
<td>ADDR</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

where T<sub>]</sub> is 1 to transmit data (read a register or the output memory, master-in/slave-out or MISO) and 0 to receive data (write a register or the input memory, master-out/slave-in or MOSI) with the reference at the slave. The address requires only 3 bits leaving the last four as don’t care. In the taped-out design, MISO and MOSI lines were combined on a bidirectional pin with the direction defined as a NOR operation between SS_N and V<sub>N</sub>, with both low for MOSI transaction. V<sub>N</sub> must therefore always be kept low during the first four bits and must be charged high during the lowest four bits if MISO transaction will follow. The 3 address bits link to the registers below, all filled with zeros by default after a reset.

<table>
<thead>
<tr>
<th>ADDR</th>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>RUN</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>CK</td>
<td>SEL</td>
</tr>
<tr>
<td>001</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td></td>
<td>DDC_INPUT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td></td>
<td>DDC_OUTPUT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td>DIV_SEL</td>
<td>RO_SEL</td>
<td></td>
<td></td>
<td>SEL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td></td>
<td>REG_SEL</td>
<td>REG</td>
<td>STD_SEL</td>
<td>STD</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In general, all blocks are off by default not to generate unnecessary noise or power consumption. The test procedure for each block is detailed in the following sections.

5.3.1 Digital down converters

The DDC blocks are programmed with the first four registers. The input memory is first programmed with register 2, starting with the first sample of the first channel (MSB first). Data is then continuously provided, going up in channel numbers and then in
Figure 5.4: SPI transactions for transmit and receive. $V_N$ sets the direction of the bi-directional MISO/MOSI pin. Data can come in burst mode for which block of 8 bits are repeated until all memory is read or written. Data toggles are best done at rising of SCLK for MOSI and will come at falling edge from slave for MISO.

sample numbers from 0 to 15. $8 \times 16$ packs of 8 bits of data can therefore be provided in burst mode to program the full memory, using the fact that the pointer register is automatically incremented. When only a few bits of data need to be changed, register 1 can be used to set the pointer first and then write only one particular sample with register 2. The second step is to choose the DDC block to activate with the first register; SEL at 0 links the standard DDC output to the output memory, whereas SEL at 1 uses the TSPC DDC output. RUN is write only and has to be set to 1 to start the DDC using the fast clock, coming in this test structure directly from a voltage-controlled oscillator (VCO) and whose frequency should be selected in the respective block.\footnote{CK bit was first supposed to select a frequency division factor but was not used. It has consequently no effect in this design.} The DDC then runs on the fast clock and stops after 26 clock cycles of the fast clock to account for 10 latency cycles and 16 inputs. To avoid synchronization issues the test controller is not sensitive to the fast clock though and simply waits then in an idle state until 5 SCLK cycles have passed. This means that the slow clock should be at least 5.2 times slower than the fast one (it can be stopped for a while in the middle if needed as SCLK has no other influence during this time). Once this time has passed, a new transaction can be issued to read register 3 and get the output data. This data comes with the 19 bits of the $Q$ output first and then the 19 bits of the $I$ output (LSB first for both). Two aligning bits are then issued to align on a multiple of 8 bits (40 bits in this case) and this is repeated in burst mode for the 16 output vectors.

An example of testbench is given in section B.1 with functional simulation in Figure B.1. It applies two sinusoidal inputs (one in phase and one in quadrature) following the procedure described above (i.e. writing the input memory, configuring the DDC, letting it run, reading the output memory).
5.3. TEST PROCEDURE

5.3.2 Frequency divider

The frequency divider described in the previous section is programmed by register 4. As previously mentioned, the output can be selected as coming from multiple channels which are selected by the SEL bits: 0 meaning everything is off, 1 for the ring oscillator output, and 2 to 5 for division by 4 (two blocks in series) using, in order, static flip-flops regular divide-by-2, TSPC flip-flops regular divide-by-2, static flip-flops input-doubled divide-by-2 and TSPC flip-flops input-doubled divide-by-2 blocks. When the SEL bits are in the range 2 to 5 and one of the dividing channel is on, the ring oscillator is obviously turned on as well for correct functionality.

The output of the selected channel is then fed to a dividing chain based on divide-by-2 blocks again and of programmable length from 5 down to 2 based on DIV_SEL bits (the two extremes are 0 yielding length 5, or division by a factor $2^5 = 32$, and 3 yielding length 2, or division by a factor $2^2 = 4$).

Finally, the ring oscillator used for this block is programmable with 3 bits (RO_SEL). It consists of a chain of inverters of sizes $W_N = 150\,\text{nm}$ and $W_P = 400\,\text{nm}$ with typical delay in the order of $t_d = 15$ to $20\,\text{ps}$. RO_SEL bits define the oscillation frequency through chain length in 8 steps from RO_SEL = 0, length 17 or $\approx 3.8\,\text{GHz}$, up to RO_SEL = 7, length 3 or $\approx 15\,\text{GHz}$. These frequencies are simulated values at 1 V supply but are modifiable by tuning the supply voltage as the ring oscillator is tied to $V_N$ instead of $V_{DD}$. Output is then provided on the sig_out line.

5.3.3 Ring oscillator

Finally, the ring oscillator block uses register 5. Standard and cascoded (REG) versions are turned on by respectively bit 0 and 4. Their length is also programmable by the SEL bits from 71 (SEL = 0) down to 57 (SEL = 7) to make it easy to have the same frequency for both of them. In addition to these bits, frequency of the cascoded version is also tunable with gate voltage $V_N$, $V_P$ and bulk voltages $V_{bn}$ and $V_{bp}$. The sizing was done for an output frequency of around 400 MHz for the standard one with $V_{DD} = 1\,\text{V}$ and similar values for the cascoded one for the same supply $V_{DD} = 1\,\text{V}$ and $V_N = 0.75\,\text{V}$, $V_P = 0.25\,\text{V}$ and $V_{bn} = V_{bp} = 0\,\text{V}$. The output is then divided by 4 before being provided on the sig_out line similarly to the frequency divider output. It is therefore recommended to have only one of the two blocks on at the same time and at most one of the two ring oscillator on when this block is chosen.
Discussion of results and outlook

As mentioned earlier, the core itself of both DDC versions included in the test design have been intensively optimized to reach frequencies of 4 GHz and more. General remarks on how such a high speed was made possible are highlighted in this chapter with additional ideas for further improvements. Some comments on the weaknesses of the taped-out design (not fixed due to time constraints) must be mentioned before though.

6.1 Taped-out design weaknesses

This design was initially thought as a test to try and push the speed limits of the DDC design as far as possible (with an initial goal of at least 1.5 GHz to 2.5 GHz). As such, optimization time and effort were mostly concentrated on the DDC core with the tape-out specifics only handled when results exceeding 3 GHz to 4 GHz were obtained. As a first consequence, power distribution in the core and in the core ring lacks intensive analysis and $I_R$ drop might be high enough to slow down the design. Additional stripes in topmost metals were added in the final design to try and counter this but connections could not be made on every row due to intermediate routing already densely present. The second consequence of the tape-out structure added afterwards is the absence of a proper failure analysis capability. In this design, it will indeed be difficult to locate the failure if the Silicon measurements do not provide the expected results; even just knowing whether the problem happened inside the DDC core or in the interface between the core and the input or output memories is hardly doable. Scan chain insertion as well as possibility to write the output memory and read the input memory directly through the SPI interface should have been added for better testing abilities. Finally, the high-frequency clock used as input in the design may not be as ideal as tested. Notably, it may not be symmetric and stable as time constraints forced us to take it from the output of a voltage-controlled oscillator (VCO) and not a phase-locked loop (PLL) as this block was not ready.

6.2 Design optimization

As highlighted in this report, optimization requires the following steps in a semi-custom flow. First, the structure should be optimized through RTL code to unlock the best
performance metrics (notably area and power). Pipelining length should then be defined to reach the minimal delay bound defined by technology and standard cell library thanks to optimal registers placement. Finally, further improvements can be made by borrowing some aspects to the full-custom flow and designing specific standard cells that can reduce the minimal delay bound.

6.2.1 RTL optimization and pipeline length

The most important point to mention is that most of the optimization methods used are generic and applicable for all high-speed designs. Indeed, as was shown in Figure 3.3, the maximal achievable speed is mostly dependent on the technology used and the standard cell library at disposition but not on the specificity of the circuit. Therefore, by optimizing the pipeline length, this frequency can theoretically be reached for all circuits under similar conditions. In practice however, there is never a single metric that needs to be optimized (as it was done for speed) but always a suitable point in a given specification set to find. The trade-offs for speed in case of pipelining are obviously area and power which will increase from the addition of new registers and new toggling nets. Any area increase will indeed have a direct repercussion on die cost and also power, while power increase will reduce battery lifetime for mobile usage in wireless communication applications. These metrics will however greatly depend on the design and consequently on parameters such as the input size $S_{in}$, the order $N$ or the channel number $R$ for a DDC as defined in this work. The other main factor defining area and power is the RTL code used and how the synthesizer can optimize the logic structure extracted from the code.

The previous paragraph seems very contradictory at first with the results from Table 3.3 and the 30% delay improvement noted only from a RTL change. The reason is that the pipeline length was kept the same in both designs whereas the reference design with adders would probably have benefited from a longer pipeline length at the cost of an area increase. Nevertheless, this was not easy to include in the synthesis flow as the routing load was hidden in the structure of the design itself as noted in the paragraph about wire-load modeling from chapter 3. As such, if synthesized in a tool with better placement and physical knowledge, it would be expected that both designs could reach the same speed but the compression tree implementation would keep the advantage of a more compact structure.

6.2.2 Standard cells in dynamic logic

After RTL and pipeline optimizations, the only remaining options to improve delay or area are improvements in the standard cells themselves (assuming a fixed technology). This was proven in this work by modifying the registers from static flip-flops to dynamic flip-flops, yielding an additional 16% increase in the maximal sustainable frequency. Of course, the advantage of the semi-custom flow means that this improvement is directly available to every other design in the same technology as the cells were all made compatible with ST’s 8-track library.

Similarly to this, any other logic gates could also be changed from static to dynamic logic if necessary to improve delay. It would for instance be possible to build half-adder or full-adder cells, maybe directly with a flip-flop included, and use directly those in the
compression tree implementation. For high-speed designs, gates used should be kept relatively small though as the logical depth has to be small between two consecutive registers. Whether this technique should be applied to half- and full-adders or NOR and NAND gates remains to be evaluated thus; it may also be that most of the gains have already been achieved by changing the most-used cell, the flip-flop.

6.2.3 Fully depleted SOI

Another promising idea that was not pursued in this work due to time constraints is to use the specifics of the ST's fully depleted SOI technology to boost the speed. Low threshold voltage (LVT) transistors in a flipped well are obviously used for high-speed designs and of course benefit from the full depletion of the well. Two other options to modulate the threshold voltage are however available and were not used. The first one, the poly biasing, or an increase in poly gate width above the active area, can modify the cell performances. Poly biases from 0 nm to 16 nm are available with only 0 nm cells used. Poly bias decreases leakage but also lowers performance; large delay gains are therefore not expected even though it could be use for some precharge or non delay-sensitive devices, notably for power optimization. The second option is to use the body biasing and forward-bias the body to increase performance. As the NMOS device is much stronger than the PMOS device, forward bias is only applied to PMOS bodies by default (with both NMOS and PMOS bodies tied to ground). Dynamic cells require a strong NMOS for evaluation but can live with a weaker PMOS for precharge. They could therefore greatly benefit from a forward bias on NMOS bodies as well.

6.2.4 Clock skew

When reaching the limits of pipelining, logic depth in-between registers is very small and often limited to a single up to a few gates. Nevertheless, propagation delay may not be exactly the same on all paths. The maximal frequency in this situation is defined by the path having the worst delay as it is the one that will fail first. Introducing skew, or delay, in the clock distribution can be a handy solution to harmonize slack between data delay and clock delay on all registers, henceforth optimizing the maximal working frequency. Cadence SOC Encounter has an option allowing the insertion of skew in the clock tree to perform such optimizations called useful skew. The small differences in delays from different paths observed in the DDC design together with the approximate wire-load modeling from routing made the results not convincing. With better capacitance modeling (for instance, with capacitance tables for routing) this option could be applied successfully though, allowing maybe an improvement of the last tens of picoseconds needed to meet timing constraint.

6.3 Outlook and conclusion

All the previously described optimization steps have been performed in our case for a digital down converter with with parameters $N = 4$, $R = 8$ and $S_{in} = 8$. Nevertheless, results should be the same for other designs or DDC with different parameters in terms of speed. As expected, pipeline length indeed showed the $1/x$-like curve allowing the choice of the adequate number of registers to reach the maximal frequency. A major
difference was experienced though: namely, the structural change going from adders to an optimal compression tree led to a speed improvement of more than 30% when theory predicts only area changes. As explained, this particularity is probably the consequence of a lack of routing load information by the tools used in the semi-custom flow. Synthesis and place and route are then not able to perform correct choices during mapping and optimization, leading to sub-optimal timing results. These variability in results was also noticed by the differences in delay when comparing tool reports versus extracted simulations in Cadence Virtuoso. Wire-load modeling appears to counter some of the problems but is far from being an optimal solution as it is purely statistical and implies a general increase in driving strength and therefore area for only a few paths that would require it. Another disadvantage is the increase in development time with the need of a second path through the flow. Similarly, extracted simulations, due to their analog nature, require a lot more effort and time compared to purely digital SDF-annotated simulations. Improvements in this regard would consequently be greatly appreciated. They can probably only come with a physically-knowledgeable flow from the start and it is certainly a reason for the emergence of new tools combining both synthesis and physical placement for better optimization. Lou for instance claims in [31] a better timing target with Cadence PKS versus the traditional synthesis followed by place and route in Cadence Encounter.

Timing gains are not much better than what is obtained from in-place optimization or multiple passes though. Nonetheless, they have a big advantage: full integration in one tool and no iteration. The consequence at the end of the day is a non-negligible gain in design time, which in turn would allow spending more time on library optimization. Dynamic logic inside flip-flops for instance proved itself worth it with 16% improvement on the maximal frequency, pushing it to 4.3 GHz. Other similar changes in standard gates or usage the particularities of the FDSOI technology could be implemented and probably increase this number even more.

Ultimately, all those modifications would still abut against a minimal delay bound. Full-custom design can be tried to overpass it. It is however very time consuming and not as generic as this design was thought; even though it is true that standard cell development of the flip-flops goes into this direction, it is still considered more generic as they are directly compatible with other circuits as well. The first logical continuation of this work is consequently to adapt the changes made to the flip-flops to other gates as well, leading eventually to the creation of a specialized cell library targeting high-speed. Meanwhile, decrease of design time thanks to better delay predictability early in the design flow would be highly valued. Entirely statistical approach such as wire-load modeling has shown its weaknesses in this work and is probably going to be replaced by a unified semi-custom flow with better physical knowledge during synthesis already. With routing accounting for a significant part of load and delay in smaller technology nodes, this seems unavoidable anyway. Last but not least, it is important to remember that a real design should balance this speed optimization against all the other metrics to reach the optimal trade-off for the targeted application; paraphrasing Mahatma Gandhi:

\textit{There is more to life than simply increasing its speed.}
Bibliography


A

Scripts for EDA tools

A.1 Synthesis

The full synthesis script used in Synopsys Design Compiler 2013.03 [34] is given here for reference.

Listing A.1: syn_generic_opt_nDFF.tcl

```tcl
set DESIGN_ENTITY "$(ENTITY_NAME)_${SUFFIX}" set DESIGN "$(DESIGN_ENTITY)_clk$(CLK_PERIOD)ns"

set target_library [insert $target_library 0 "../CDS_VISO/liberateRunDir/DDC_ss_0.80V_125C.db"]

set link_library "*$_target_library$_synthetic_library"

set wireload 1
if { $wireload } {
    set link_library "../CDS_SOCE/DDC_wireload_flat2.db*$_target_library$_synthetic_library"
}

# clean
remove_design -all
file delete -force -- DLIB/DDC/

file delete -force -- DLIB/WORK/

define_design_lib -path DLIB/DDC DDC

# analyse VHDL sources
puts "-- Analyze VHDL_sources"
analyze -format vhdl $(VHDL_SOURCES) -library DDC

# elaborate
puts "-- Elaborate design"
elaborate $(ENTITY_NAME) -architecture $(ARCH_NAME) \
    -parameter $(ELAB_PARAMS) \
    -library DDC -update

if { $wireload } {
    set_wire_load_mode "enclosed"
    set_wire_load_model -name DDC_8ch_flat [get_designs DDC_8ch]
}
```

43
APPENDIX A. SCRIPTS FOR EDA TOOLS

```plaintext
# constraints
set_max_area 0
create_clock ${CLK_NAME} -period ${CLK_PERIOD}
set_fix_multiple_port_nets -all

# link
puts "-i-Link_design"
link

set_dont_use [get_lib_cells C28SOI_SC_8_CORE_LL/C8T•DF* ]
set_dont_use [get_lib_cells C28SOI_SC_8_COREPBP4_LL/C8T•DF* ]
set_dont_use [get_lib_cells C28SOI_SC_8_COREPBP10_LL/C8T•DF* ]
set_dont_use [get_lib_cells C28SOI_SC_8_COREPBP16_LL/C8T•DF* ]

# compile
puts "-i-Compile_design"
link

set_dont_use [get_lib_cells C28SOI_SC_8_CORE_LL/C8T•DF* ]
set_dont_use [get_lib_cells C28SOI_SC_8_COREPBP4_LL/C8T•DF* ]
set_dont_use [get_lib_cells C28SOI_SC_8_COREPBP10_LL/C8T•DF* ]
set_dont_use [get_lib_cells C28SOI_SC_8_COREPBP16_LL/C8T•DF* ]

set_dont_retime [get_cells "*out_I_reg_reg *"] true
set_dont_retime [get_cells "*out_Q_reg_reg *"] true
set_dont_retime [get_cells "*delay_reg_reg[0]*"] true
optimize_registers

# reports
puts "-i-Generate_reports"

report_constraint -nosplit -all_violators > RPT/${DESIGN}_allviol.rpt
report_area -hierarchy > RPT/${DESIGN}_area.rpt
report_timing > RPT/${DESIGN}_timing.rpt
report_resources -nosplit -hierarchy > RPT/${DESIGN}_resources.rpt
report_reference -nosplit -hierarchy > RPT/${DESIGN}_references.rpt

# generate Verilog netlist
#
puts "-i-Generate_Verilog_netlist"
change_names -rules verilog -hierarchy
write -format verilog -hierarchy -output HDL/GATE/${DESIGN}_mapped.v

# generate SDF timing file for Verilog
puts "-i-Generate_SDF_file_for_Verilog_netlist"
write_sdf -version 2.1 TIM/${DESIGN}_mapped_vlog.sdf

# generate design constraint file
puts "-i-Generate_SDC_design_constraint_file"
write_sdc -nosplit SDC/${DESIGN}_mapped.sdc
```
A.2  Place and route

The full place and route script used in Cadence SoC Encounter 11.13 [30] is given here for reference.

Listing A.2: pr_generic.tcl

```tcl
if { ![ file exists RPT/$DESIGN] } { mkdir RPT/$DESIGN }

#setMultiCpuUsage −localCpu max

# prepare/init
create_constraint_mode −name constraint_typ −sdc_files ..:/SNPS_DC/SDC/$ {DESIGN} −_mapped.sdc
create_delay_corner −name delay_corner_typ −library_set libs_typ −rc_corner − rc_typ
create_delay_corner −name delay_corner_best −library_set libs_best −rc_corner − rc_typ
create_delay_corner −name delay_corner_worst −library_set libs_worst −rc_corner − rc_typ
create_analysis_view −name analysis_typ −constraint_mode constraint_typ − delay_corner delay_corner_typ
create_analysis_view −name analysis_best −constraint_mode constraint_typ − delay_corner delay_corner_best
create_analysis_view −name analysis_worst −constraint_mode constraint_typ − delay_corner delay_corner_worst

set init_verilog HDL/GATE/$ {DESIGN}_mapped.v
set init_top_cell STOPCELL
init_design −setup {analysis_worst} −hold {analysis_best}
setDrawView fplan
fit

saveDesign DB/$DESIGN−import.enc

# floor plan
floorPlan −r 1 {$fp_core_util} {$fp_core_to_left} {$fp_core_to_bottom} {$fp_core_to_right} {$fp_core_to_top}
setFinishFPlanMode −activeObj macro −direction xy −override false
finishFloorplan −autoHalo

# power ring
set sprCreateeleRingNets {}
set sprCreateeleRingLayers {}
set sprCreateeleRingWidth 1.0
set sprCreateeleRingSpacing 1.0
set sprCreateeleRingOffset 1.0
set sprCreateeleRingThreshold 1.0
set sprCreateeleRingJogDistance 1.0
setAddRingMode −stacked_via_top_layer IB −stacked_via_bottom_layer MB
```
APPENDIX A. SCRIPTS FOR EDA TOOLS

```
addRing −skip_via_on_wire_shape Noshape −skip_via_on_pin Standardcell −center 1
−type core_rings −jog_distance ${ring_dist} −threshold ${ring_dist} −nets
−[concat $init_gnd_net $init_pwr_net] −follow core −layer {bottom M1 top M1
−right M2 left M2} −width ${ring_width} −spacing ${ring_spacing} −offset $−
{ring_dist} −extend_corner {bl br rb lb}

# power stripes
set sprCreateIeStripeNets {}
set sprCreateIeStripeLayers {}
set sprCreateIeStripeWidth 10.0
set sprCreateIeStripeSpacing 2.0
set sprCreateIeStripeThreshold 1.0
setAddStripeMode −stacked_via_top_layer IB −stacked_via_bottom_layer M1
addStripe −skip_via_on_wire_shape Noshape −block_ring_top_layer_limit M3
−max_same_layer_jog_length 4 −padcore_ring_bottom_layer_limit M3
−set_to_set_distance ${stripe_dist} −skip_via_on_pin Standardcell
−padcore_ring_top_layer_limit M3 −spacing ${ring_spacing} −xleft_offset ${−
stripe_dist} −merge_stripes_value ${ring_dist} −layer M2
−block_ring_bottom_layer_limit M1 −width ${ring_width} −nets [concat
$init_gnd_net $init_pwr_net]

saveDesign DB/$DESIGN −fplan.enc

# Add filler cells on top and bottom rows to meet DRC rules GR111_or (Proximity
−effect design rule)
set Region_xll [dbDBUToMicrons [expr [lindex [dbFPlanCoreBox [dbHeadFPlan] ] 0]]]
set Region_yll [dbDBUToMicrons [expr [lindex [dbFPlanCoreBox [dbHeadFPlan] ] 1]]]
set Region_xur [dbDBUToMicrons [expr [lindex [dbFPlanCoreBox [dbHeadFPlan] ] 2]]]
set Region_yur [dbDBUToMicrons [expr [lindex [dbFPlanCoreBox [dbHeadFPlan] ] 3]]]

set TopRow_yll [expr $Region_yur −1.20]
set BotRow_yur [expr $Region_yll +1.20]

addFiller −cell $FILLER_TB_CELLS −markFixed −prefix FILLTOP −area $Region_xll
−$TopRow_yll $Region_xur $Region_yur
addFiller −cell $FILLER_TB_CELLS −markFixed −prefix FILLBOT −area $Region_xll
−$Region_yll $Region_xur $BotRow_yur

# Add WellTaps on all rows with 50 micron spacing to meet all LUP rules (Latchup
−rules)
addWellTap −cell $WELLTAP_CELLS −cellInterval 50 −inRowOffset 10.0 −prefix
−WELLTAP

# placing
setMultiCpuUsage −localCpu 4 −cpuPerRemoteHost 1 −remoteHost 0 −keepLicense true
setDistributeHost −local
setPlaceMode −timingDriven true
placeDesign −prePlaceOpt
testDrawView place
```
saveDesign DB/$DESIGN−placed.enc

cHECKPLACE RPT/$DESIGN/place.rpt

# limit routing up to Metal 8 (M9 and M10 are for power and would issue error because they are in the same direction as M8)
setNanoRouteMode −routeTopRoutingLayer 8

# pre–CTS optimization
timeDesign −preCTS −idealClock −pathReports −slackReports −numPaths 5 \\ −prefix timing_preCTS_setup −outDir RPT/$DESIGN
timeDesign −preCTS −hold −idealClock −pathReports −slackReports −numPaths 5 \\ −prefix timing_preCTS_hold −outDir RPT/$DESIGN

setOptMode −fixCap true −fixTran true −fixFanoutLoad false

saveDesign DB/$DESIGN−placed−topt−preCTS.enc

# clock tree
createClockTreeSpec −bufferList $CT_BUFFER −file CTS/$ {DESIGN} _spec.cts
clockDesign −specFile CTS/$ {DESIGN} _spec.cts −outDir RPT/$DESIGN \\ −fixedInstBeforeCTS

setOptMode −fixCap true −fixTran true −fixFanoutLoad false

optDesign −postCTS

clockDesign −specFile CTS/$ {DESIGN} _spec.cts −outDir RPT/$DESIGN \\ −fixedInstBeforeCTS

saveDesign DB/$DESIGN−cts.enc

# filler

addFiller −cell $FILLER_CELLS −prefix FILLER

saveDesign DB/$DESIGN−filled.enc

# global nets connections

globalNetConnect $init_pwr_net −type ppgin −pin $init_pwr_net −inst * −module {} \\ −verbose
globalNetConnect gnd −type ppgin −pin vdds −inst * −module {} −verbose
globalNetConnect $init_gnd_net −type ppgin −pin $init_gnd_net −inst * −module {} \\ −verbose
globalNetConnect gnd −type ppgin −pin gnds −inst * −module {} −verbose
globalNetConnect $init_pwr_net −type tiehi −module {}
globalNetConnect $init_gnd_net −type tieio −module {}

applyGlobalNets

# power routing

sroute −connect { blockPin corePin } −layerChangeRange { M1 M4 } −blockPinTarget \\ { nearestRingStripe nearestTarget } −checkAlignedSecondaryPin 1 \\ −allowlogging 1 −nets [concat $init_gnd_net $init_pwr_net] \\ −allowLayerChange 1 −blockPin useLef −targetViaBottomLayer M1 \\ −targetViaTopLayer M4 −crossoverViaBottomLayer M1 −crossoverViaTopLayer M4

saveDesign DB/$DESIGN−power.enc

# routing design

setNanoRouteMode −routeWithTimingDriven true −routeTdrEffort 7
APPENDIX A. SCRIPTS FOR EDA TOOLS

− routeTopRoutingLayer 8
routeDesign − globalDetail

setOptMode − fixCap true − fixTran true − fixFanoutLoad false
# optDesign − postRoute
optDesign − postRoute − hold

addFiller − cell $FILLER_CELLS − prefix FILLER
saveDesign DB/$DESIGN− routed.enc

# verify
verifyConnectivity − type all − report RPT/$DESIGN/ connectivity.rpt − error 100
− warning 50
verifyGeometry − report RPT/$DESIGN/ geometry.rpt
verifyProcessAntenna − report RPT/$DESIGN/ processAntenna.rpt

# report
summaryReport − outdir RPT/$DESIGN/ summary
report_power − outfile RPT/$DESIGN/ power.rpt
report_timing > RPT/$DESIGN/ timing_final.rpt

# export final design
write_sdf − view analysis_worst TIM/$ {DESIGN} _ pared.sdf
saveNetlist − excludeLeafCell HDL/GATE/$ {DESIGN} _ pared.v
streamOut DEX/$ {DESIGN} .gds − mapFile $MAP_FILE − libName $DESIGN − units 1000
− mode ALL

A.3 Cell characterization

The characterization script used in Cadence Liberate 13.14 [35] is given here for reference.

Listing A.3: liberate_DDC.tcl

# get cells and conditions
set rundir $env(PMD)

if { [string equal $condition "ss"] } {
    set models "$rundir/../CORNERS_local/corners_ss.scs"
    set voltage 0.80
    set temp 125
    set suffix ss_0.80V_125C
}

if { [string equal $condition "tt"] } {
    set models "$rundir/../CORNERS_local/corners.scs"
    set voltage 0.90
    set temp 25
    set suffix tt_0.90V_25C
}

if { [string equal $condition "ff"] } {
    set models "$rundir/../CORNERS_local/corners_ff.scs"
    set voltage 1.05
}
A.3. CELL CHARACTERIZATION

```tcl
set temp -40
set suffix ff_1.05V_m40C
}
source liberate_DDC_template.tcl
set Operating_condition -voltage $voltage -temp $temp
set_var extsim_model_include $models
set_var extsim_cmd "spectre"
set_var extsim_deck_dir decks_DDC_$suffix
set_var extsim_save_passed deck
define_leafcell -type nmos {lvtnfet}
define_leafcell -type pmos {lvtpfet}
set spice_netlists [list $models]
set csz [llength $cells]
for {set c 0} { $c < $csz } { incr c 1 } {
    set cell [lindex $cells $c]
    lappend spice_netlists $cell.sp
}
read_spice -format spectre $spice_netlists
char_library -extsim SPECTRE -cells $cells
add_lib_attribute { default_max_fanout : 20; 
}
write ldb DDC_$suffix.ldb
write_library -filename DDC_$suffix.lib DDC_stdcells
```

The cell template is the following to match the default characterization by ST for their equivalent static cell.

**Listing A.4: liberate_DDC_template.tcl**

```tcl
# DDC_stdcells
set_var slew_lower_rise 0.2
set_var slew_lower_fall 0.2
set_var slew_upper_rise 0.8
set_var slew_upper_fall 0.8
set_var measure_slew_lower_rise 0.2
set_var measure_slew_lower_fall 0.2
set_var measure_slew_upper_rise 0.8
set_var measure_slew_upper_fall 0.8
set_var delay_inp_rise 0.4
set_var delay_inp_fall 0.6
set_var delay_out_rise 0.4
set_var delay_out_fall 0.6
set_var def_arc_msg_level 0
set_var max_transition 2.175e-09
```
set_var min_transition 3e-12
set_var min_output_cap 2e-16

# set_var constraint_delay_degrade_absol 2e-12
# set_var constraint_search_bound -1e-10
set_var constraint_glitch_peak 0.5

set cells {
    DDC_DFPRQX19_1
}

# index1: slew on Data (ns), index2: slew on Clock (ns)
define_template -type constraint \
    -index_1 [0.003 0.019 0.037 0.15 0.3] \ 
    -index_2 [0.003 0.066 0.13 0.53 1.06] \ 
    constraint_template_5x5

define_template -type constraint \
    -index_1 [0.003 0.019 0.037 0.15 0.3] \ 
    -index_2 [0.003 0.066 0.13 0.53] \ 
    constraint_template_5x4

define_template -type constraint \
    -index_1 [0.019] \ 
    -index_2 [0.019] \ 
    constraint_template_2

# index1: input slew (ns), index2: output load (pF)
define_template -type delay \
    -index_1 [0.003 0.022 0.043 0.085 0.17 0.33 0.66 1.325] \ 
    -index_2 [0.0002 0.0022 0.0044 0.0088 0.018 0.035 0.07 0.2086] \ 
    delay_template_8x8

define_template -type delay \
    -index_1 [0.003 0.022 0.043 0.085 0.17 0.33 0.66 1.325] \ 
    -index_2 [0.0002 0.0023 0.0047 0.0094 0.019 0.037 0.11175] \ 
    delay_template_8x7_17

# index1: input slew (ns), index2: output load (pF)
define_template -type power \
    -index_1 [0.003 0.022 0.043 0.085 0.17 0.33 0.66 1.325] \ 
    -index_2 [0.0002 0.0022 0.0044 0.0088 0.018 0.035 0.07 0.2086] \ 
    power_template_8x8

define_template -type power \
    -index_1 [0.003 0.022 0.043 0.085 0.17 0.33 0.66 1.325] \ 
    -index_2 [0.0002 0.0023 0.0047 0.0094 0.019 0.037 0.11175] \ 
    power_template_8x7_17

define_cell \
    -clock { CP } \ 
    -async { RN } \ 
    -input { D } \ 
    -output { Q } \ 
    -pinlist { CP D RN Q } \ 
    -delay delay_template_8x8 \ 
    -power power_template_8x8 \
And finally, the .lib file created needs to be translated into a .db file for use in Synopsys Design Compiler, which can be done by the following script in a separate instance of Synopsys DC.

Listing A.5: dc_lib2db.tcl

```tcl
# set file_name DDC_ff_1.05V_m40C
set file_name DDC_tt_0.90V_25C
# set file_name DDC_ss_0.80V_125C

5 set lib_name DDC_stdcells

enable_write_lib_mode
read_lib ${file_name}.lib
write_lib ${lib_name} -f db -o ${file_name}.db

exit
```
Test and verification examples

B.1 Testbench

A testbench using the SPI protocol to program the test interface and run the DDC is given here as example. It first writes the input memory with the samples generated from two sinusoidal signal, one in phase and one in quadrature. The DDC is then configured and started. Finally, after waiting for the DDC to finish, the output memory is read back and output samples are verified as soon as they are collected thanks to a comparison with the outputs of the equivalent MATLAB code in section B.2.

Listing B.1: DDC_chip_tb.vhd

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.math_real.all;

library DDC;
use DDC.polyphase_DDC_package.all;

entity DDC_chip_tb is
end entity DDC_chip_tb;

architecture bench of DDC_chip_tb is
  -- constants
  constant CLK_H_PER : time := 4 ns;
  constant CLK_L_PER : time := 40 ns;

  -- signals
  signal clk_low, rst : std_logic := '0';
  signal clk_high : std_logic := '0';
  signal sub_adc_in_vec : input_vector(N_CHANNEL-1 downto 0);
  signal output_I, output_Q : std_logic_vector(NBITS_OUT-1 downto 0);
  signal ss_n : std_logic := '1';
  signal mosi, miso : std_logic;
  signal bidir_io_en : std_logic;
  signal command : std_logic_vector(7 downto 0);
  constant CMD_WR_DDC_IN : std_logic_vector(7 downto 0) := "00100000";
  constant CMD_RD_DDC_OUT : std_logic_vector(7 downto 0) := "10110000";
  constant CMD_WR_DDC_CFG : std_logic_vector(7 downto 0) := "00000000";
end architecture bench;
```
constant CMD_DDC_CFG_RUN : std_logic_vector(7 downto 0) := "10000000";

signal correct : boolean := true;
signal stop : boolean := false;

-- components
component chiptop is
  generic (---
    SPI_WIDTH: natural := 8;
    ADDR_SIZE: natural := 4;
    NTESTS : natural := 16);
  port (---
    I/O interface
    signal clk_high : in std_logic;
    signal clk_low, rst : in std_logic;
    signal ss_n : in std_logic;
    signal data_in : in std_logic;
    signal sig_out : out std_logic;
    signal bidir_io_en : out std_logic
  )
end component chiptop;

constant NTESTS : natural := 16;
constant ADDR_SIZE : natural := integer(cei l (log2(real(NTESTS))));
constant LATENCY : natural := 10;
constant N_LOAD_CYCLES : natural := NTESTS * N_CHANNEL * NBITS_IN;
constant FREQ_RATIO : real := real(CLK_L_PER / 10 ps) / real(CLK_H_PER / 10 ps
  --);
constant N_RUN_CYCLES : natural := integer(cei l (real(NTESTS+LATENCY+1))
  -- FREQ_RATIO));

-- verif for:
-- Function: value := sin(MATH_2_PI*real(8*i+k)/101.0);
-- N = 3, R = 8, NBITS_IN = 8, N_TEST = 128
signal correct_I_out : integer_array(0 to 127) := (
  1700, 37136, 133000, 217400, 249560, 221296, 139164, 23356, 249776, 213840, 126040, 7684,
  169676, 236972, 246360, 196104, --196104,
  -246360, -236972, -169676, -61363, 61363, 169676, 236972, 246360, 196104, --98272,
  -23356, -139164, -221296, -249560, -217400, -132556, -15508, 105272, 200532,
  247124, 233908, 163844, 54012, -68980, -175336, -239156, -245072, -191204,
  -91188, 31080, 145584, 225156, 249776, 213840, 126040, 7684, -112352,
  -205244, -248144, -230916, -157784, -46288, 76364, 180560, 240876, 242900, 185992,
  -38624, -151860, -228264, -249352, -209808, -119300, 0, 119300, 209808,
  -249352, 228264, 151860, 38624, -83824, -185992, -242900, -240876, -180560, -76364,
  46288, 157784, 230916, 248144, 205244, 112352, -7684, -126040, -213840,
  -249776, -225156, -145584, -31080, 91188, 191204, 245072, 239156, 175336, 68980,
  -54012, -163844, -233908, -247124, -200532, -105272, 15508, 132556, 217400, 249560,
  221296, 139164, 23356, -98272, -196104, -246360, -236972, -169676, -61363,
  61363, 169676, 236972, 246360, 196104, 98272, -23356, -139164, -221296,
  -249560, -217400, -132556, -15508) ;
\[\text{Function: } \text{value} := \sin(\text{MATH}_2 \cdot \text{PI} \cdot \text{real}(8*i+k)/97.0);\]

\[N = 3, R = 8, \text{NBITS}_\text{IN} = 8, \text{N_TEST} = 128\]

\[
\text{signal correct_Q_out : integer_array(0 to 127) := (}
2003, 39323, 137736, 222064, 248689, 209659, 115890, -7805, -129967, \\
-218014, -248725, -123394, 0, 123394, 214184, 248689, -222064, -137166, -16134, 109161, 205658, 248203, \\
225272, 143273, 24259, -101392, -200757, -247101, -228751, -32197, \\
94424, 196038, 246176, 231614, 39781, -86606, -190557, -244970, \\
-234763, -162858, 79313, 185817, 243528, 237116, 168301, 55899, -71154, \\
-180095, -241362, -239479, -174618, -63855, 63855, 174618, 239479, 241362, 180095, \\
71154, -55899, -168301, -237116, -243528, -185817, -79313, 48130, 162858, 234763, \\
244970, 190557, 86606, -39781, -156362, -231614, -246176, -196038, -94424, \\
32197, 150274, 228751, 247101, 200757, 101392, -24259, -143273, -225272, \\
-248203, -205658, -109161, 16134, 137166, 22064, 248689, 209659, 115890, \\
-7805, -129967, -218014, -248725, -214184, -123394, 0, 123394, 214184, 248725, \\
-218014, 129967, 7805, -115890, -209659, -248689, -222064, -137166, -16134, 109161, \\
205658, 248203, 225272, 143273 \);}

\begin{verbatim}
begin
  assert correct report "Wrong_output" severity failure;

  DUT_c: component chiptop
  port map(
    clk_high ,
    clk_low ,
    rst ,
    ss_n ,
    mosi ,
    miso,
    bidir_io_en
  );

  -- signal generation
  clk_high <= not clk_high after CLK_H_PER/2 when not stop;
  clk_low <= not clk_low after CLK_L_PER/2 when not stop;
  rst <= '0', '1' after CLK_L_PER/4, '0' after 7*CLK_L_PER/4;

  SIGGEN_p: process
  procedure apply_in (i: in natural) is
    variable value_I, value_Q: real;
    variable value_sampled: real;
  begin
    for k in 0 to N_CHANNEL-1 loop
      value_I := sin(MATH_2*PI*real(8*i+k)/101.0);
      value_Q := sin(MATH_2*PI*real(8*i+k)/97.0);
      value_sampled := value_I*cos(MATH_2*PI*real(8*i+k)/4.0) + value_Q * sin(MATH_2*PI*real(8*i+k)/4.0);
      sub_adc_in_vec(k) <= std_logic_vector(to_signed(integer(round(value_sampled*real(2**(NBITS_IN-1)-1))), NBITS_IN));
    end loop;
  end procedure apply_in;

  process
  begin
    for i in 0 to 127 loop
      apply_in(i);
    end loop;
  end process;
end
\end{verbatim}
end procedure apply_in;

procedure serialize_in is begin
  for k in 0 to N_CHANNEL-1 loop
    for bit_i in NBITS_IN-1 downto 0 loop
      wait until falling_edge(clk_low);
      mosi <= sub_adc_in_vec(k)(bit_i);
    end loop;
  end loop;
end procedure serialize_in;

procedure apply_cmd is begin
  wait until rising_edge(clk_low);
  wait for CLK_L_PER/4;
  ss_n <= '0';
  for k in 7 downto 0 loop
    if k /= 7 then
      wait for CLK_L_PER;
    end if;
    mosi <= command(k);
  end loop;
end procedure apply_cmd;

procedure collect_output is begin
  variable temp_out : std_logic_vector(NBITS_OUT-1 downto 0);
  for k in 0 to NBITS_OUT-1 loop
    wait until rising_edge(clk_low);
    temp_out(k) := miso;
  end loop;
  output_I <= temp_out;
  for k in 0 to 7-(2*NBITS_OUT mod 8) loop
    wait until rising_edge(clk_low);
  end loop;
end procedure collect_output;

begin
  sub_adc_in_vec <= (others => (others => '0'));
  wait until falling_edge(rst);
  -- write input memory
  command <= CMD_WR_DDC_IN;
  apply_cmd;
  for test_i in 0 to NTESTS-1 loop
    apply_in(test_i);
    serialize_in;
  end loop;
  wait until falling_edge(clk_low);
  wait until falling_edge(clk_low);
  wait for CLK_L_PER/4;
  ss_n <= '1';
B.2. TEST VECTOR GENERATION

Test vectors such as those used in the example testbench above can be generated with the following MATLAB [21] codes executing the same function as the digital down converter designed.

Listing B.2: dec_filter.m

```matlab
function [I, Q, Hz_coeffs, channel] = dec_filter(input, n_channel, R, N)
% decimation filter with:
% - I and Q: in phase and quadrature outputs
% - Hz_coeffs: subfilter coefficients
% - channel: sub-channel outputs
```
APPENDIX B. TEST AND VERIFICATION EXAMPLES

(a) Full waveform with the two cursors at 42,100 ns and 42,340 ns marking the switch of phases between writing the input memory, running and reading the output memory.

(b) Zoom of subfigure (a) between the two cursors at 42,100 ns and 42,340 ns to see the running phase of the DDC.

Figure B.1: Waveforms from testbench simulation configuring the DDC and verifying the outputs. Note how the running phase (parallel and on the fast clock $\text{clock\_high}$) is short and fast compared to the serial input/output phase on the low frequency clock $\text{clock\_low}$. 
B.2. TEST VECTOR GENERATION

input_size=size(input);
if(input_size(1)~=n_channel)
    error('Input should be an array of vectors of size n_channel')
end
if(mod(n_channel, 4)~=0)
    error('n_channel should be a multiple of 4')
end

% build coeffs of H(z)
p = ones(1, R);
if N==1
    Hz_coeffs = p;
else
    Hz_coeffs = conv(p, p);
    for i=3:N
        Hz_coeffs = conv(Hz_coeffs, p);
    end
end

% each channel transfer function
channel = zeros(n_channel, input_size(2));
for i=1:n_channel
    tmp = conv(Hz_coeffs(i:R:end), input(1+n_channel-i,:),:);
    channel(i,:) = tmp(1:input_size(2));
end

% Q and I outputs
if (n_channel > 4)
    Q = sum(channel(3:4:end,:)) - sum(channel(1:4:end,:));
    I = sum(channel(4:4:end,:)) - sum(channel(2:4:end,:));
else
    Q = channel(3,:) - channel(1,:);
    I = channel(4,:) - channel(2,:);
end

Listing B.3: tb_dec_filter.m

% options: sampling/decimation filter
n_channel = 8;
f_clk = 1.5e9;
fs = f_clk*n_channel;
R = 8;
N = 4;
ns = 128;
nbits_in = 8;

% input function
input_func_I = @(t) sin(2*pi*fs/101*t);
input_func_Q = @(t) sin(2*pi*fs/97*t);
fc = fs/4;
t_sampling=0:1/fs:(ns*n_channel-1)/fs;
input_I=input_func_I(t_sampling).*cos(2*pi*fc*t_sampling);
input_Q = input_func_Q(t_sampling) .* sin(2*pi*fc*t_sampling);

input = round(((input_I + input_Q) .* (2^(nbits_in-1)-1));

input_shaped = reshape(input, n_channel, length(input) / n_channel);

% decimation
[I, Q, Hz_coeffs, channel] = dec_filter(input_shaped, n_channel, R, N);