Internal Report, No. CAS-LA05.1

Implementation of an 18-point IMDCT on FPGA

Huibert J. Lincklaen Arriëns, June 2005 (last update: July 1, 2005).

1 Introduction

Given the tremendous interest in MP3 decoders, in which short Inverse Modified Discrete Cosine Transforms (IMDCTs) are applied (on blocks of either 12 or 36 samples), recently several publications [Bri01], [Lee01], [Nik04] have addressed the issues of MDCTs and IMDCTs . Generally, these publications are written to obtain an efficient solution in a software environment and are mainly concerned with the MDCT. The IMDCT is said to be obtained by reversed operations and is not described in detail.

In our research, we are interested in hardware implementations for stand-alone or co-processor like architectures to support and speed-up the software. In the following, the 18-point IMDCT (block size 36) for implementation on an FPGA will be described.

2 The IMDCT

2.1 Definition

The backward MDCT or IMDCT is defined as

$$\hat{x}_m = \frac{2}{N} \sum_{k=0}^{(N/2)-1} X_k \cdot \cos\left[\frac{\pi}{2N}(2k+1)\left(2m+1+\frac{N}{2}\right)\right], \quad with\, m = 0, 1, 2, \cdots, N-1. \tag{1}$$

Generally, since we are dealing with a lapped transform, the recovered data sequence $\{\hat{x}_m\}$ does not correspond to the original data sequence $\{x_m\}$. To obtain the correct $\{x_m\}$ the outputs of consecutive transforms have to be combined.

It can be seen that $N/2$ (non redundant) input values result in N output values (of course the MDCT reads N input values and results in $N/2$ output values). Since it is not completely clear whether Equation 1 should be called an N-point IMDCT or an $N/2$-point IMDCT, in the following we'll identify these transforms given the number of inputs. We will be describing an 18-point IMDCT that delivers 36 output values, thus length N will be 36.

2.2 Literature

Especially interesting is the article from Britanak and Rao[Bri01], who described a detailed algorithm to calculate 12 and 36 point MDCTs. In the same year, Szu-Wei Lee [Lee01] proposed a different approach, which was claimed to cost less multiplications and additions than Britanak's algorithm. However, Lee's approach is less suitable for a hardware implementation since it uses several recursive statements that consequently cannot be synthesized with concurrently operating hardware.

More recently, Nikolajevic and Fettweis [Nik04] discovered that Britanak's implementation was not optimal and could be improved upon, resulting in about the same number of multiplications and additions as were needed by Lee's solution. In this case, however, without any recursiveness.

Still, all proposed soltions are mainly aimed at an in-depth treatment of the forward MDCT while being relatively incomplete in their description of the IMDCT. Generally the conclusions are that all operations should be reversed.

In the following, a description will be given of a dedicated hardware implementation of an 18-point IMDCT which can be used in an MP3 decoder (e.g. for retrieving 36 point time samples out of 18 frequency lines).

2.3 Realization

Our goal is to find a realization similar to those used in the afformentioned MDCT descriptions. The main reasons therefore are

- a similar structure may facilitate resource resusage, and
- the forward DCT-II 'module' that Britanak describes for his MDCT solution needs less resources than his inverse DCT-II to be used in his IMDCT, viz. 10 multiplications and 34 additions against rep. 12 and 36.

Figure 1 shows the basic setup that will be used for our IMDCT.

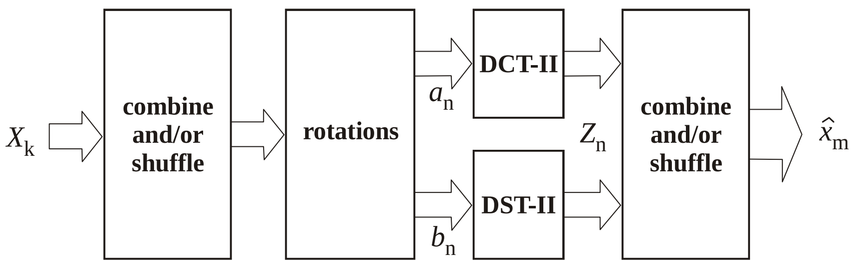


Figure 1: Basic setup of the IMDCT.

Since we defined $N = 36$, we start from an 18 values input sequence: $\{X_0, X_1, \cdots, X_{17}\}$.

The outputs of the 'rotations'-block are given by:

$$a_n = X_n \cos\left[\frac{\pi}{2N}(2n+1)\right] + X_{N/2-1-n}\sin\left[\frac{\pi}{2N}(2n+1)\right]$$

$$b_n = X_n \sin\left[\frac{\pi}{2N}(2n+1)\right] - X_{N/2-1-n}\cos\left[\frac{\pi}{2N}(2n+1)\right] \qquad n = 0, 1, 2, \cdots, \frac{N}{4} - 1.$$

The leftmost 'combine and shuffle'-block is thus nothing more than a reverse ordering of the second half of the input data.

The cos- and sin- angles $\theta = \frac{\pi}{2N}(2n+1)$ involved, are $\theta = \frac{\pi}{72}, \frac{3\pi}{72}, \frac{5\pi}{72}, \cdots, \frac{17\pi}{72}$.

Next, we perform a 9-point DCT-II on the a_n vector and a 9-point DST-II on the b_n vector, which delivers us

- $Z_n =$ DCT-II_9p (a_n)

  (with the DCT-II_9p given by $Z_j = \sum_{n=0}^{8} a_n \cdot \cos\left[9j\pi\left(n + \frac{1}{2}\right)\right], \quad j = 0, 1, \cdots, 8.$).

- $Z_{(N/4)-1+n} =$ DST-II_9p (b_n):

  [Wan82] described that this DST_II can favourably be performed by a similar DCT-II_9p as has been used to obtain Z_n, considering that the input and output data need to be negated partly, and be reordered at the output (Figure 2). The procedure to follow here is: first, negate the even terms $(n = 0, 2, \cdots, N/4 - 1)$ of b_n

  - $b'_{n=even} = -b_{n=even}$, then
  - $Z_{(N/2)-1-n} =$ DCT-II_9p (b'_n)

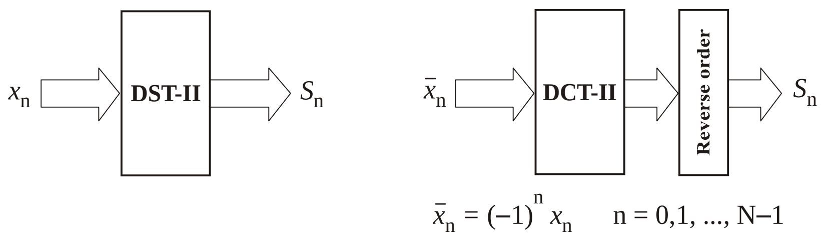


Figure 2: Constructing a DST-II from the DCT-II.

In the rightmost 'combine and shuffle'-block, y_k can be derived from these Z's as

$$y_k = \begin{cases} -Z_k\,, & \text{for } k = 0 \text{ and } k = N/2-1 \\ Z_{\frac{N}{4}-1+\frac{k}{2}} - Z_{\frac{k}{2}}\,, & \text{for } k = 2,4,6,\cdots,N/2-2 \\ -Z_{\frac{N}{4}-1+\frac{k+1}{2}} - Z_{\frac{k+1}{2}}\,, & \text{for } k = 1,3,5,\cdots,N/2-3 \end{cases}$$

Finally, $\{\hat{x}_m\}$ can be found from

$$\hat{x}_m = \begin{cases} -y_{\frac{N}{4}-1-m}\,, & \text{for } m = 0,1,\cdots,N/4-1 \\ y_{m-\frac{N}{4}}\,, & \text{for } m = N/4,\cdots,3N/4-1 \\ y_{\frac{5N}{4}-1-m}\,, & \text{for } m = 3N/4,\cdots,N-1 \end{cases}$$

When for computing the 9-point DCT-II's, a slightly modified version of Britanak's description (see Appendix A) is used, the above leads to a total number of 52 multiplications and 102 additions.

2.4 Implementation

For the hardware implementation, we assume the availability of multipliers, ALUs and registers. The ALUs can be instructed to either act as an adder or as a subtractor, depending on a single 'opcode' bit.

To avoid negations –which would cost rather ineffective hardware– the algorithm above has been rewritten slightly. In fact, no negations are needed upto and including the calculation of the 18 y_k values. These y_k's will also be the output of this implementation. It is assumed that extending y_k to $\hat{x}_m$ can be taken care of in the hardware or software following this IMDCT block by a.o. correctly indexing in the windowing calculations. The final set-up that was to be implemented is shown in Figure 3.

Several programs and intermediate programs have been written to stepwise verify the design process.

1. For the ultimate reference, data for the IMDCT can be generated with the MATLAB program `imdct_def.m`, which is a one-to-one calculation of the IMDCT definition formula given in Equation 1.
2. Then, based on Britanak's elaborate description of his DCT-II algorithm (see Appendix A), a text file `imdct_18p_sh.cir` has been created (a sequence graph with I/O's is shown in Figure 4). By dividing all cos- and sin-coefficients by $N/2$ the scaling factor needs no additional multipliers.
3. This `cir`-file was first checked by reading and evaluating it with `my_imdct_18p_ref.m` which in fact just provides the input values for the `cir`-file. This resulted in the same floating point output values as has been found as reference.
4. Next, our in-house scheduling software (`schedGUI.m`) used this `cir`-file to find a resource constrained scheduling scheme, based on the List Scheduling Method. This scheduling software generates

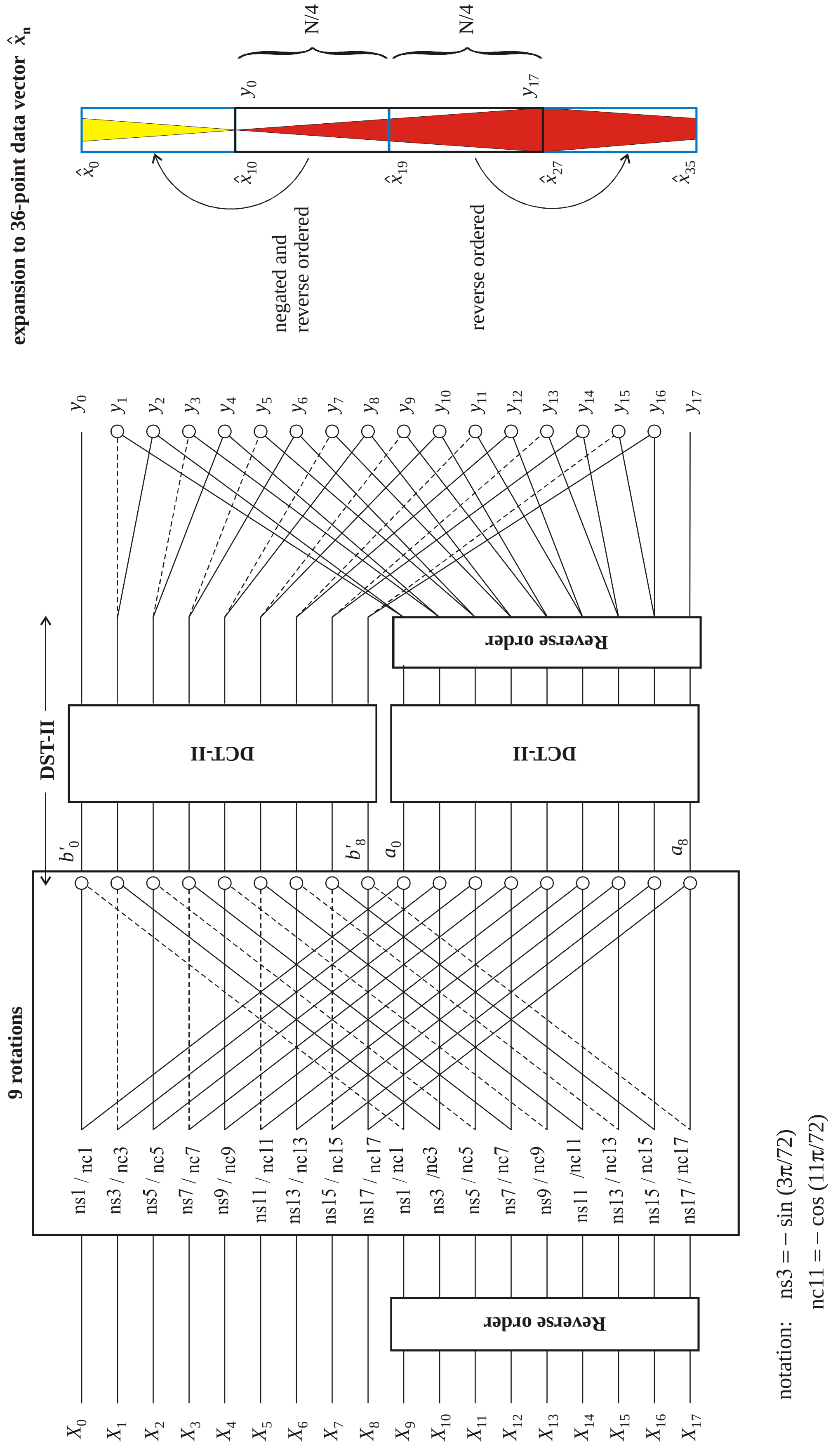


Figure 3: Efficient implementation of an 18 point IMDCT. Full lines represent a transfer factor $+1$, dotted lines a transfer factor -1, and a ◯ represents an addition (or subtraction)

– a MATLAB testbench for debugging and evaluation purposes (scheduled, fixed point output),
– VHDL code for synthesis and
– a VHDL testbench that should result in bit-true and state-true output when compared with the MATLAB testbench.

The version of the software used, had first been augmented to be able to translate multiplications by powers of 2 into arithmetic shifts. In hardware this leads to only differently connected (possibly duplicated to cope with sign extension) signal connections, thus saving on hardware multiplier usage. There is another benefit made possible by this feature, viz. that we can scale the cos- and sin-coefficients by N instead of $N/2$, and perform the divide by 2 at the very end of the scheme by a right shift of all y_k values. This will improve the overall accuracy of the implementation as will be shown later on.

A second modification in the new version is that now each resource (multiplier, ALU) is followed by its own register (asynchronous reset, clock-enable input) and needs only one clock cycle to compute its output. By properly writing VHDL code tailored to Xilinx Spartan-3 FPGAs, the registers following the multipliers are incorporated in the dedicated multiplier hardware (MUL18X18S components will be instantiated), so again some area will be saved. If intermediate computational results that last longer than one clock state.need to be stored, this will invoke additional registers, also with an async reset and a clock-enable

3 Design Space

Using the scheduling software with sets of different multiplier/ALU combinations can give an indication about the feasible number of Clock States against the number of resources (= area) that are needed (see Table 1).

Table 1 xx/yy indicates: number of States/number of additional Registers

	ALUs									
MULtipliers	1	2	3	4	5	6	7	8	9	10
1	109/26	61/23	55/20	54/19	53/17	53/18	53/18	53/18	53/18	53/18
2	106/26	55/24	39/23	**31/20**	**29/18**	29/17	28/15	28/15	28/15	28/16
3	105/26	54/24	37/22	29/21	25/19	22/17	21/16	20/15	20/16	19/13
4	104/26	53/24	37/22	28/20	24/18	20/18	18/16	**17/15**	16/14	16/16
5	104/26	53/24	36/22	28/20	23/19	20/17	18/15	16/14	16/14	14/13
6	104/26	53/24	36/22	28/20	23/19	19/17	17/16	16/16	14/15	13/12

Note that for the Spartan 3 FPGAs a number of dedicated multipliers are already available on the dye (p.e. 40 18x18 bits multipliers on an XC3S2000 device) and that they are either in use by other parts of the hardware or unused and thus available to us. Also, the number of bits used for data in our IMDCT determines how much hardware multipliers are combined to calculate the results of one multiplication, e.g. a 32 bits data buswidth takes up 4 multiplier components per multiplication).

As a starting point, a setup was chosen with 4 multipliers and 8 ALUs (functioning as an adder or as a subtractor). According to the scheduling process, the outputs of the IMDCT will be available after 17 clock states while15 additional register banks will be needed.

Figure 5 shows the resulting Scheduled Sequence Graph (SSG), and Figure 6 depicts the distribution of the resources versus the clock states.

Also shown is a map of the resource usage and a map of the additional registers in resp. Figures 7 and 8.

The two other bold faced combinations (2MULs,4ALUs and 2MULs,5ALUs) in Table 1 are referred to later on in this document.

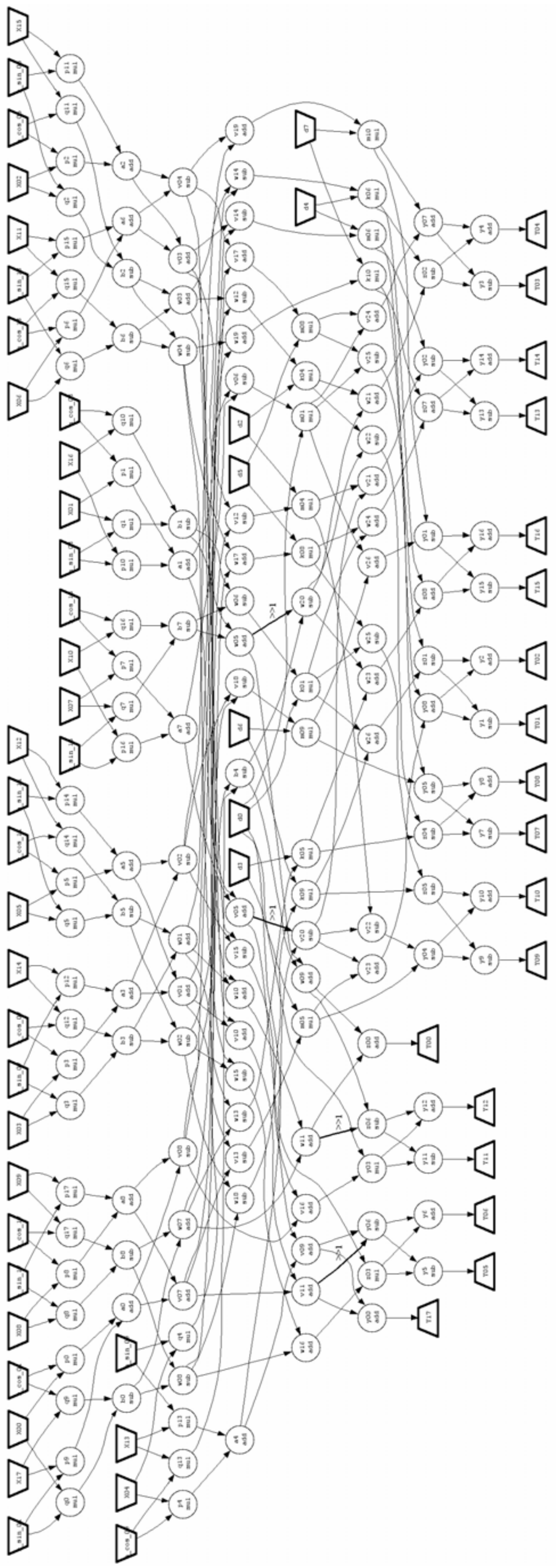


Figure 4: Interconnection of the operations and I/Os.

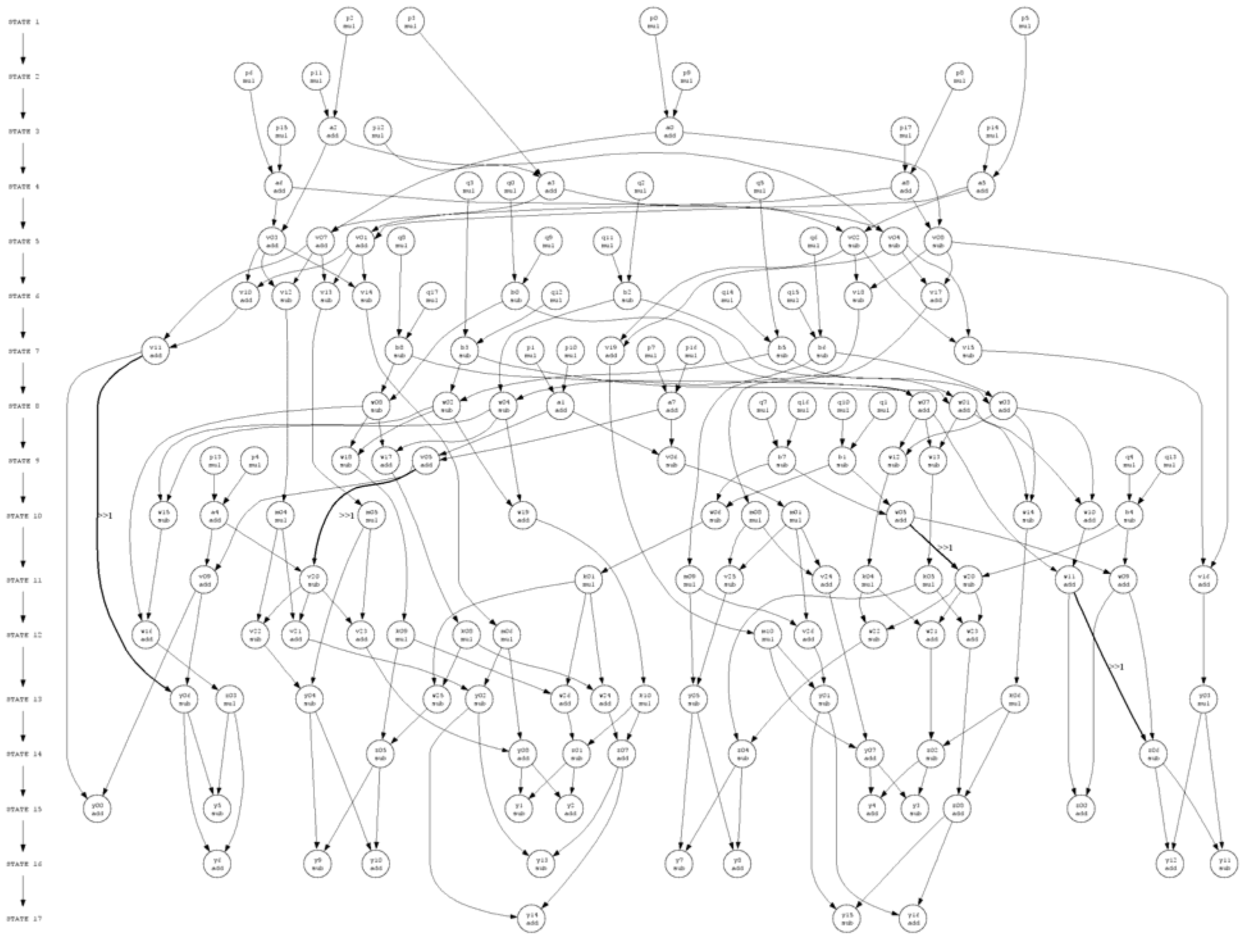


Figure 5: Scheduled Sequence Graph (SSG) after application of the List Scheduling Method with 4 multipliers and 8 ALUs.

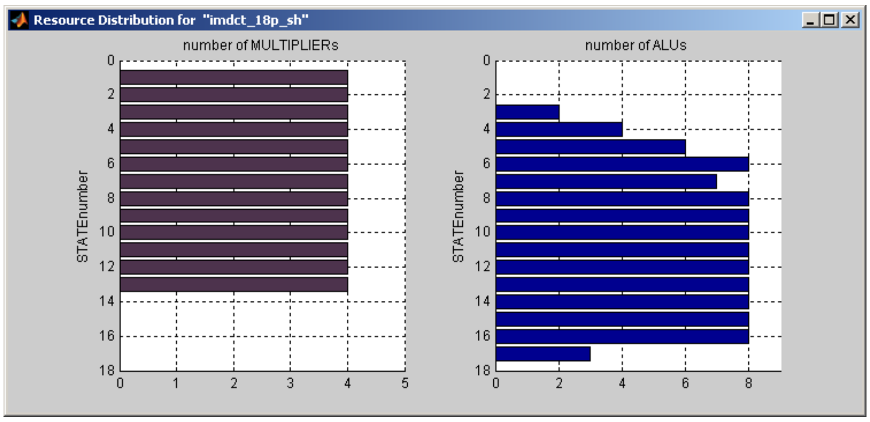
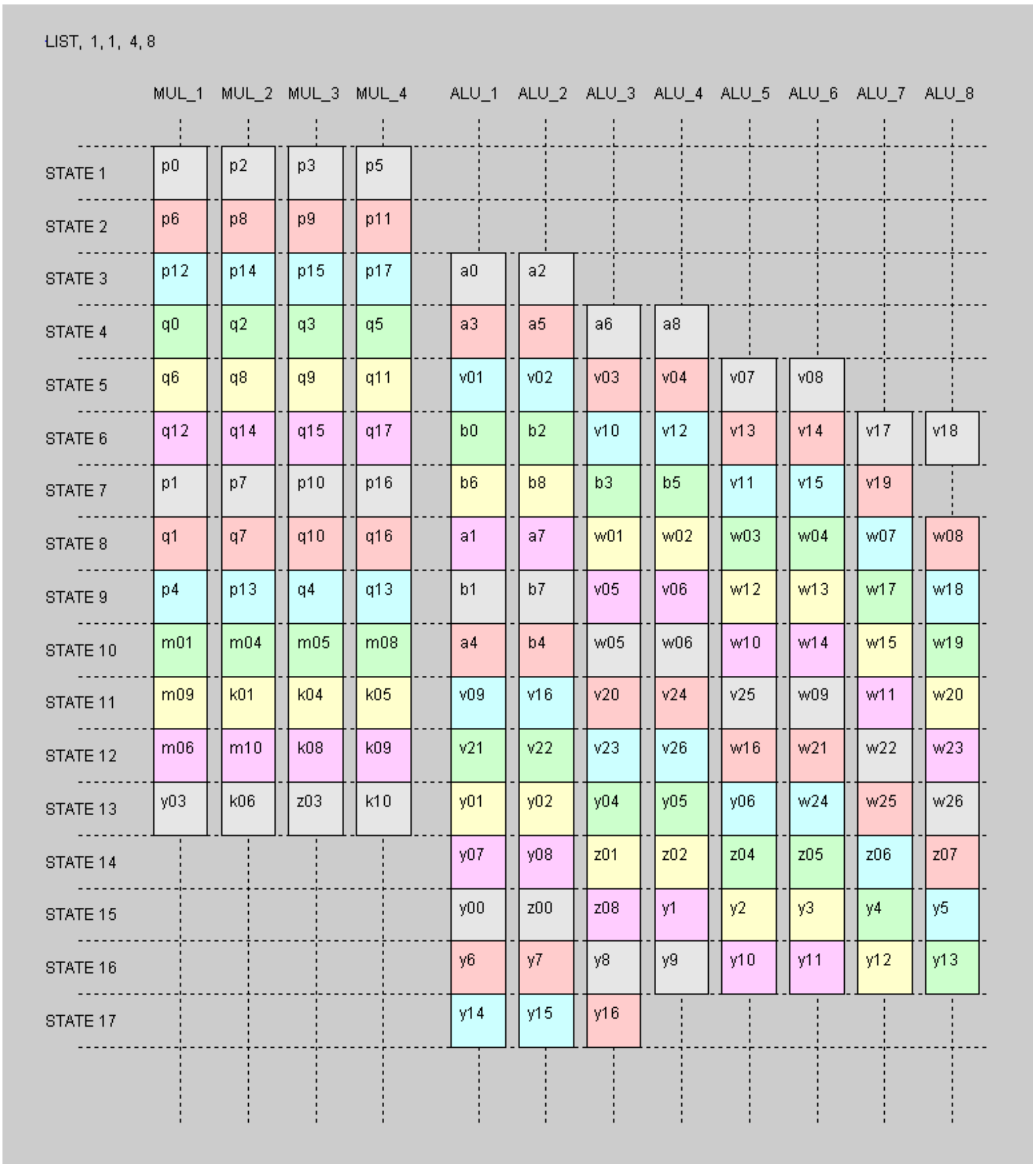


Figure 6: Distribution of the available resources.



LIST, 1, 1, 4, 8

	MUL_1	MUL_2	MUL_3	MUL_4	ALU_1	ALU_2	ALU_3	ALU_4	ALU_5	ALU_6	ALU_7	ALU_8
STATE 1	p0	p2	p3	p5								
STATE 2	p6	p8	p9	p11								
STATE 3	p12	p14	p15	p17	a0	a2						
STATE 4	q0	q2	q3	q5	a3	a5	a6	a8				
STATE 5	q6	q8	q9	q11	v01	v02	v03	v04	v07	v08		
STATE 6	q12	q14	q15	q17	b0	b2	v10	v12	v13	v14	v17	v18
STATE 7	p1	p7	p10	p16	b6	b8	b3	b5	v11	v15	v19	
STATE 8	q1	q7	q10	q16	a1	a7	w01	w02	w03	w04	w07	w08
STATE 9	p4	p13	q4	q13	b1	b7	v05	v06	w12	w13	w17	w18
STATE 10	m01	m04	m05	m08	a4	b4	w05	w06	w10	w14	w15	w19
STATE 11	m09	k01	k04	k05	v09	v16	v20	v24	v25	w09	w11	w20
STATE 12	m06	m10	k08	k09	v21	v22	v23	v26	w16	w21	w22	w23
STATE 13	y03	k06	z03	k10	y01	y02	y04	y05	y06	w24	w25	w26
STATE 14					y07	y08	z01	z02	z04	z05	z06	z07
STATE 15					y00	z00	z08	y1	y2	y3	y4	y5
STATE 16					y6	y7	y8	y9	y10	y11	y12	y13
STATE 17					y14	y15	y16					

Figure 7: Mapping of the operations on the available resources according to the List Scheduling Method.

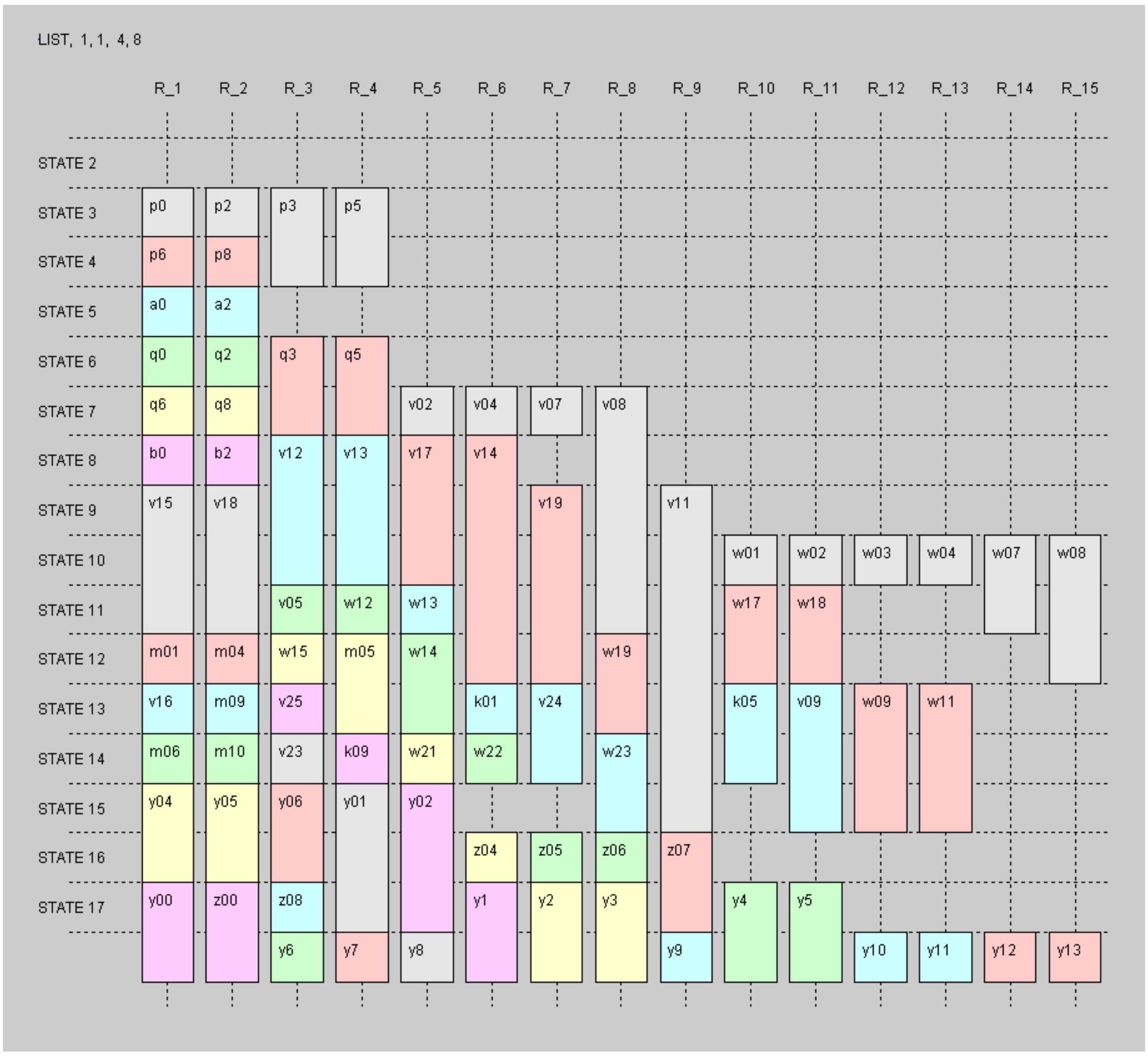


Figure 8: Additional registers needed to account for the lifetimes of computational results.

4 Verification

The input values for the tests mentioned hereafter were choosen to be $X_k = \dfrac{k}{20}$ with $k = 1, 2, \cdots, 18$.

In Table 2, the (18 relevant) reference output values y_k for $k = 0, 1, \cdots, 17$ which correspond with $\hat{x}_m$ for $m = 9, 10, \cdots, 26$ are listed in the leftmost column.
The middle columns (decimal and hex representations) show the output of the generated MATLAB testbench when all calculations are performed in a fixed point format: the total buswidth here is 32 bits, of which the lower 30 bits are regarded to represent the binary fraction. All –scaled– multiplication constants and inputs have also been converted to the [32 30] format using a rounding function.

The relative error between both outputs, $\dfrac{y_{testbench} - y_{ref}}{y_{ref}}$, is listed in the rightmost column.

Figure 9 lists the values that result from simulation of the VHDL files with the ModelSim simulator. They can be seen to be exactly the same as the ones predicted by MATLAB. Figure 10 additionally shows a timing simulation of a 'half scale' implementation. Appendix D finally, shows the input and resulting output after simulation in graphical format.
In Table 3 the effect of the distributed scaling can be seen, viz. using sin- and cos-coefficients scaled by only half the scaling factor and a final shift right of the output values.It would be tempting to try to obtain even better results by additional left and right shifting, and this will indeed succeed for the input values used throughout this report. For a worst case input consisting of all ones (all frequencies present at maximum level), however, overflow in the ALUs will occur. Expanding the buswidth can overcome this problem, but it doesn't seem worth the additional area since the errors won't drop significantly being determined by the number of fraction bits of the final result.

In Figure 11 can be seen what the effect is of different fixed point formats on the relative error, both for the 'full scale' and the 'half scale' methods.

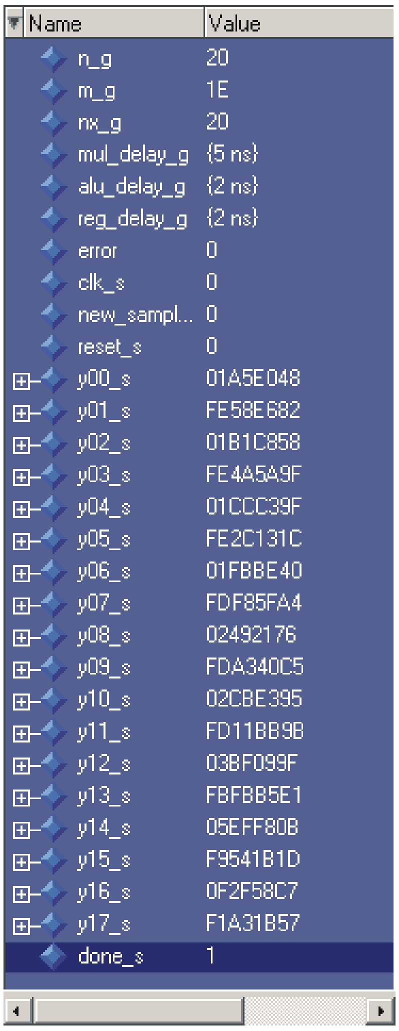
Table 2

reference (floating point)	MATLAB testbench ($2/N$) (fixed point calculations) [32 30.0]		relative error $*10^{-6}$
0.0257493	0.0257493	`01A5E048`	0.0564
-0.0258239	-0.0258239	`FE58E682`	0.0915
0.0264760	0.0264760	`01B1C858`	0.1013
-0.0267118	-0.0267118	`FE4A5A9F`	0.0988
0.0281228	0.0281228	`01CCC39F`	-0.1054
-0.0285599	-0.0285599	`FE2C131C`	-0.0565
0.0309902	0.0309902	`01FBBE40`	0.0458
-0.0317155	-0.0317155	`FDF85FA4`	0.1003
0.0357135	0.0357135	`02492176`	0.0439
-0.0369108	-0.0369108	`FDA340C5`	0.0859
0.0436944	0.0436944	`02CBE395`	0.0448
-0.0457927	-0.0457927	`FD11BB9B`	0.0259
0.0585350	0.0585350	`03BF099F`	-0.0267
-0.0627618	-0.0627618	`FBFBB5E1`	-0.0596
0.0927715	0.0927715	`05EFF80B`	-0.0218
-0.1042416	-0.1042416	`F9541B1D`	-0.0043
-0.2372648	-0.2372648	`0F2F58C7`	-0.0123
-0.2244197	-0.2244197	`F1A31B57`	0.0365

Table 3

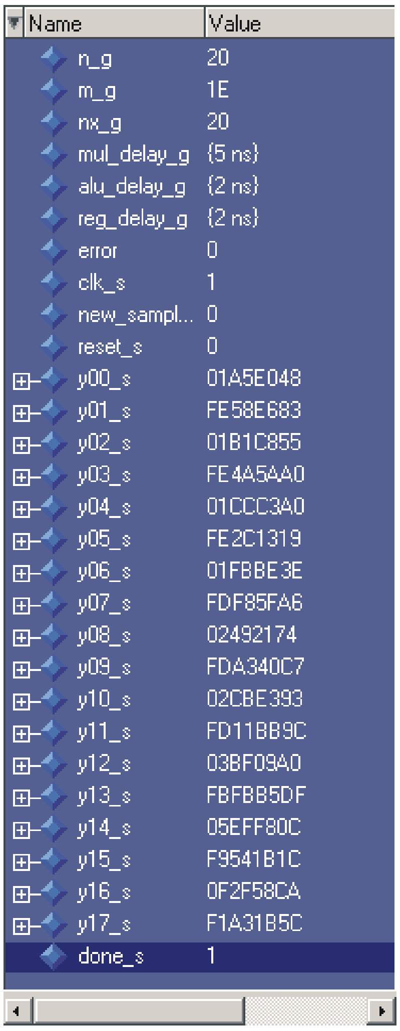
MATLAB ($4/N \gg 1$) (fixed point calculations) [32 30.0]	relative error $*10^{-7}$
`01A5E048`	0.5643
`FE58E683`	0.5546
`01B1C855`	-0.0423
`FE4A5AA0`	0.6391
`01CCC3A0`	-0.7226
`FE2C1319`	0.4136
`01FBBE3E`	-0.1432
`FDF85FA6`	0.4155
`02492174`	-0.0825
`FDA340C7`	0.3539
`02CBE393`	0.0213
`FD11BB9C`	0.0551
`03BF09A0`	-0.1078
`FBFBB5DF`	-0.2988
`05EFF80C`	-0.1179
`F9541B1C`	0.0464
`0F2F58CA`	-0.0053
`F1A31B5C`	0.1580

a)



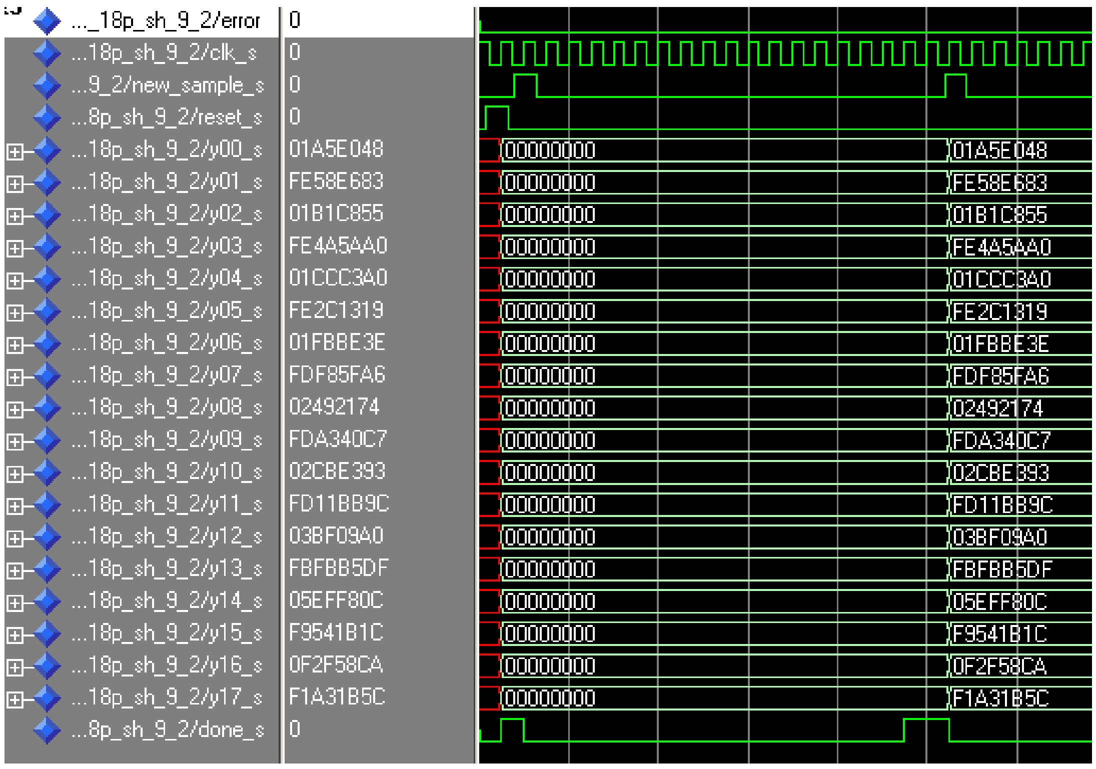
Name	Value
n_g	20
m_g	1E
nx_g	20
mul_delay_g	{5 ns}
alu_delay_g	{2 ns}
reg_delay_g	{2 ns}
error	0
clk_s	0
new_sampl...	0
reset_s	0
y00_s	01A5E048
y01_s	FE58E682
y02_s	01B1C858
y03_s	FE4A5A9F
y04_s	01CCC39F
y05_s	FE2C131C
y06_s	01FBBE40
y07_s	FDF85FA4
y08_s	02492176
y09_s	FDA340C5
y10_s	02CBE395
y11_s	FD11BB9B
y12_s	03BF099F
y13_s	FBFBB5E1
y14_s	05EFF80B
y15_s	F9541B1D
y16_s	0F2F58C7
y17_s	F1A31B57
done_s	1

b)



Name	Value
n_g	20
m_g	1E
nx_g	20
mul_delay_g	{5 ns}
alu_delay_g	{2 ns}
reg_delay_g	{2 ns}
error	0
clk_s	1
new_sampl...	0
reset_s	0
y00_s	01A5E048
y01_s	FE58E683
y02_s	01B1C855
y03_s	FE4A5AA0
y04_s	01CCC3A0
y05_s	FE2C1319
y06_s	01FBBE3E
y07_s	FDF85FA6
y08_s	02492174
y09_s	FDA340C7
y10_s	02CBE393
y11_s	FD11BB9C
y12_s	03BF09A0
y13_s	FBFBB5DF
y14_s	05EFF80C
y15_s	F9541B1C
y16_s	0F2F58CA
y17_s	F1A31B5C
done_s	1

Figure 9: Output values according to Modelsim after simulation of the testbench until the done-bit is high (all states succesfully executed). All cos- and sin -coefficients are divided by a) the scaling factor $N/2$ and b) divided by $N/4$ with an additional (SHR 1) at the output.



```
..._18p_sh_9_2/error      0
...18p_sh_9_2/clk_s       0
...9_2/new_sample_s       0
...8p_sh_9_2/reset_s      0
...18p_sh_9_2/y00_s       01A5E048    00000000    01A5E048
...18p_sh_9_2/y01_s       FE58E683    00000000    FE58E683
...18p_sh_9_2/y02_s       01B1C855    00000000    01B1C855
...18p_sh_9_2/y03_s       FE4A5AA0    00000000    FE4A5AA0
...18p_sh_9_2/y04_s       01CCC3A0    00000000    01CCC3A0
...18p_sh_9_2/y05_s       FE2C1319    00000000    FE2C1319
...18p_sh_9_2/y06_s       01FBBE3E    00000000    01FBBE3E
...18p_sh_9_2/y07_s       FDF85FA6    00000000    FDF85FA6
...18p_sh_9_2/y08_s       02492174    00000000    02492174
...18p_sh_9_2/y09_s       FDA340C7    00000000    FDA340C7
...18p_sh_9_2/y10_s       02CBE393    00000000    02CBE393
...18p_sh_9_2/y11_s       FD11BB9C    00000000    FD11BB9C
...18p_sh_9_2/y12_s       03BF09A0    00000000    03BF09A0
...18p_sh_9_2/y13_s       FBFBB5DF    00000000    FBFBB5DF
...18p_sh_9_2/y14_s       05EFF80C    00000000    05EFF80C
...18p_sh_9_2/y15_s       F9541B1C    00000000    F9541B1C
...18p_sh_9_2/y16_s       0F2F58CA    00000000    0F2F58CA
...18p_sh_9_2/y17_s       F1A31B5C    00000000    F1A31B5C
...8p_sh_9_2/done_s       0
```

Figure 10: Output of timing simulation with ModelSim.

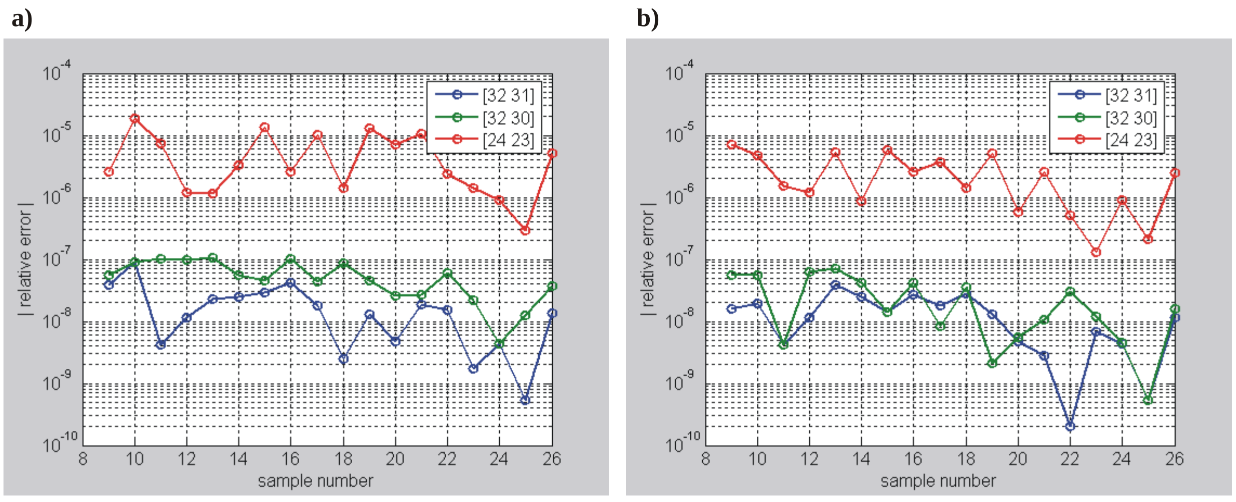


Figure 11: Errors for some different buswidths and number of fraction bits if all cos- and sin -coefficients are divided by a) the scaling factor $N/2$ and b) if divided by $N/4$ with an additional (SHR 1) at the output.

5 Hardware Utilization

In Table 4 the hardware utilization of a number of synthesized implementations, differing in design parameters, are listed.

All implementations have been scheduled with the List Scheduling Method, assuming 1 clock cycle for both multiplications and additions/subtractions (L11xxxx). The other parameters were:

L1148_18:	4 multipliers,	8 ALUs,	one scale factor of 18
L1148_9_2:	4 multipliers,	8 ALUs,	distributed scale factors 9 and 2
L1125_9_2:	2 multipliers,	5 ALUs,	distributed scale factors 9 and 2
L1124_9_2:	2 multipliers,	4 ALUs,	distributed scale factors 9 and 2

In all cases, the target device was a Xilinx Spartan 3 XC3S2000 -5 FG676 FPGA.

Synthesis was done with Synplfy Pro 8.0 from Synplicity, Inc. in the Auto Constrained Frequency mode, which results in a higher estimated frequency but also more used resources than the default 1MHz mode (generally, a higher estimated frequency involves more of the FPGA's components).

It turns out that the '_9_2' designs, apart from being slightly more accurate, also need slightly less area than the '_18' designs.

Table 4

	L1148_18	L1148_9_2	L1125_9_2	L1124_9_2
MULT18x18S	16	16	8	8
Total FDs	774	979	811	861
Total MUXFs	1794	1108	981	989
Total LUTs	7279 (17%)	6632 (16%)	5322 (12%)	5788 (14%)
Estimated Frequency [MHz]	60.2	58.7	61.4	63.4

Determined by the scheduler (see also Table 1, repeated here for convenience)

	L1148_18	L1148_9_2	L1125_9_2	L1124_9_2
Number of Clock States	17	17	29	31
Additional Registers	15	15	18	20

6 Conclusions

It is shown that, based on the basic scheme given in Figure 1 a synthesizable, hardware IMDCT implementation can be realized. The implementation described here needs less computations than the ones found in the literature up to now (see Table 5). Although the design was intended to be optimal from a hardware point of view, it thus may also be interesting for software implementations. In that case –when floating point calculations of exceptable accuracy are available, and/or when multiplications take more cycles then additions – it may be advantageous to replace the rotations as described here (viz. 4 muls/2 adds) by their 3muls/3adds equivalents. This will result in figures in Table 4 that are better comparible to those of the other software implementations, i.e. resp. 11/21 and 43/111.

Table 5

IMDCT resources	$N = 12$ mul/add	$N = 36$ mul/add
Britanak	13/33	51/151 *)
S-W. Lee	11/23	43/115
Nikolajevic	13/21	47/115 **)
this implementation	14/18	52/102 ***)

*) In some publications erronuously quoted as 47/151, since Britanak himself mentions a number of 51/151.

**) This should certainly also be 51 multiplications

***) ... or resp. 11/21 and 43/111 when the 4/2 rotations are replaced by their 3/3 equivalents.

7 Future Work

There are still a few questions that remain

- Can anyone write down the mathematical prove that the algorithm given above is correct, and –hopefully– in any way optimal?
- Is it possible to re-use the same hardware of the 18-point IMDCT for the 6-point version in one circuit?
- How would an MDCT/IMDCT combination look like?
- Is it possible to efficiently combine the IMDCT with the circuitry following it in the MP3 decoder, e.g. the windowing and reconstruction process, and the 32-point MDCT as a whole?

There are two interesting articles originating from completely different approaches, that need further investigation. They are

- Mu-Huo Cheng and Yu_Hsin Hsu, *"Fast IMDCT and DCT Algorithms – A Matrix Approach"*, IEEE Trans on Signal Processing, Vol. 51, No. 1, January 2003

and

- Oraintara, S. & Krishnan, T.: *" The integer MD*CT *and its application in the MPEG layer III audio"*, Proceedings of the 2003 International Symposium on Circuits and Systems, 2003. ISCAS '03. 25-28 May 2003.

8 References

[Bri01] V. Britanak and K.R. Rao, *"An Efficient Implementation of the Forward and Inverse MDCT in MPEG Audio Coding"*, IEEE Signal Processing Letters, Vol. 8, No. 2, February 2001.

[Lee01] Szu-Wei Lee, *"Improved Algorithm for Efficient Computation of the Forward and Backward MDCT in MPEG Audio Coder"*, IEEE Trans on CAS-II:Analog and Digital Signal Processing, Vol. 48, No. 10, October 2001.

[Nik04] V. Nikolajevic and G. Fettweis, *"Improved Implementation of MDCT in MP3 Audio Coding"*, 10th Asia-Pacific Conference on Communications and 5th International Symposium on Multi-Dimensional Mobile Communications.

[Wan82] Zhong-De Wang, *"A Fast Algorithm for the Discrete Sine Transform Implemented by the Fast Cosine Transform"*, IEEE Trans on Acoustics, Speech and Signal Processing, Vol. ASSP-30, No. 5, October 1982.

Appendix A. 9-Point forward DCT-II Module

Britanak [Bri01] proposed the following algorithm to compute the 9-Point DCT-II.
Given an input data sequence $\{x_0, x_1, \cdots, x_8\}$, the output data sequence $\{C_0^{II}, C_1^{II}, \cdots, C_8^{II}\}$ can be computed with

$$\begin{array}{ll}
a_1 = x_3 + x_5 & a_{11} = a_{10} + a_7 \\
a_2 = x_3 - x_5 & a_{12} = a_3 - a_7 \\
a_3 = x_6 + x_2 & a_{13} = a_1 - a_7 \\
a_4 = x_6 - x_2 & a_{14} = a_1 - a_3 \\
a_5 = x_1 + x_7 & a_{15} = a_2 - a_4 \\
a_6 = x_1 - x_7 & a_{16} = a_{15} + a_8 \\
a_7 = x_8 + x_0 & a_{17} = a_4 + a_8 \\
a_8 = x_8 - x_0 & a_{18} = a_2 - a_8 \\
a_9 = x_4 + a_5 & a_{19} = a_2 + a_4 \\
a_{10} = a_1 + a_3 &
\end{array}$$

$$\begin{array}{ll}
m_1 = -d_1 \cdot a_6 & m_6 = -d_5 \cdot a_{14} \\
m_2 = d_2 \cdot a_5 & m_7 = -d_1 \cdot a_{16} \\
m_3 = d_2 \cdot a_{11} & m_8 = -d_6 \cdot a_{17} \\
m_4 = -d_3 \cdot a_{12} & m_9 = -d_7 \cdot a_{18} \\
m_5 = -d_4 \cdot a_{13} & m_{10} = -d_8 \cdot a_{19}
\end{array}$$

$$\begin{array}{ll}
a_{20} = x_4 - m_2 & a_{24} = m_1 + m_8 \\
a_{21} = a_{20} + m_4 & a_{25} = m_1 - m_8 \\
a_{22} = a_{20} - m_4 & a_{26} = m_1 + m_9 \\
a_{23} = a_{20} + m_5 &
\end{array}$$

$$\begin{array}{ll}
C_0^{II} = a_9 + a_{11} & C_5^{II} = a_{25} - m_9 \\
C_1^{II} = m_{10} - a_{26} & C_6^{II} = m_3 - a_9 \\
C_2^{II} = m_6 - a_{21} & C_7^{II} = a_{24} + m_{10} \\
C_3^{II} = m_7 & C_8^{II} = a_{23} + m_6 \\
C_4^{II} = a_{22} - m_5 &
\end{array}$$

where

$$\begin{array}{llll}
d_1 = \frac{\sqrt{3}}{2} & d_2 = 0.5 & d_3 = \cos\left(\frac{8\pi}{9}\right) & d_4 = \cos\left(\frac{4\pi}{9}\right) \\
d_5 = \cos\left(\frac{2\pi}{9}\right) & d_6 = \sin\left(\frac{8\pi}{9}\right) & d_7 = \sin\left(\frac{4\pi}{9}\right) & d_8 = \sin\left(\frac{2\pi}{9}\right)
\end{array}$$

needing a total of 10 multiplications and 34 additions.

Since $d_2 = 0.5$ (or 2^{-1}), m_2 and m_3 can be discarded when a_{20} and C_6^{II} are changed into respectively

$$\begin{array}{l}
a_{20} = x_4 - (a_5 >> 1) \\
C_6^{II} = (a_{11} >> 1) - a_9
\end{array}$$

Then, noting that the implementation of an arithmetic shift on (FPGA) hardware doesn't involve any resources, the result will be 8 multiplications and 34 additions.

Appendix B. Description of the cir-file

This is the listing of the `cir`-file that has been used with sin- and cos-coefficients scaled by $\frac{4}{N}$ (Note the additional shifts at the very and of the listing). These coefficients are identified with `n_sin_01`, `n_cos_01`, etc. and are passed through the I/O connections in the VHDL-file in the fixed point format specified for a particular design.

```
iocDef = 'XYdn';     % which means:
                     % Variables starting with an X are defined to be inputs,
                     % starting with an Y are outputs, with a d or n are constant
                     % coefficients, while all others are operations (multiplier
                     % or ALU resources)

 p0 = X00 * n_cos_01;
 p1 = X01 * n_cos_03;
 p2 = X02 * n_cos_05;
 p3 = X03 * n_cos_07;
 p4 = X04 * n_cos_09;
 p5 = X05 * n_cos_11;
 p6 = X06 * n_cos_13;
 p7 = X07 * n_cos_15;
 p8 = X08 * n_cos_17;
% -- reverse ordering of X00 - X17 done here
 p9 = X17 * n_sin_01;
p10 = X16 * n_sin_03;
p11 = X15 * n_sin_05;
p12 = X14 * n_sin_07;
p13 = X13 * n_sin_09;
p14 = X12 * n_sin_11;
p15 = X11 * n_sin_13;
p16 = X10 * n_sin_15;
p17 = X09 * n_sin_17;

 a0 =  p0 +  p9;
 a1 =  p1 + p10;
 a2 =  p2 + p11;
 a3 =  p3 + p12;
 a4 =  p4 + p13;
 a5 =  p5 + p14;
 a6 =  p6 + p15;
 a7 =  p7 + p16;
 a8 =  p8 + p17;

 q0 = X00 * n_sin_01;
 q1 = X01 * n_sin_03;
 q2 = X02 * n_sin_05;
 q3 = X03 * n_sin_07;
 q4 = X04 * n_sin_09;
 q5 = X05 * n_sin_11;
 q6 = X06 * n_sin_13;
 q7 = X07 * n_sin_15;
 q8 = X08 * n_sin_17;
% -- reverse ordering of X00 - X17 done here, too
 q9 = X17 * n_cos_01;
q10 = X16 * n_cos_03;
```

```
q11 = X15 * n_cos_05;
q12 = X14 * n_cos_07;
q13 = X13 * n_cos_09;
q14 = X12 * n_cos_11;
q15 = X11 * n_cos_13;
q16 = X10 * n_cos_15;
q17 = X09 * n_cos_17;

 b0 =  q0 -  q9;
 b2 =  q2 - q11;
 b4 =  q4 - q13;
 b6 =  q6 - q15;
 b8 =  q8 - q17;
%-----
 b1 = q10 -  q1;
 b3 = q12 -  q3;
 b5 = q14 -  q5;
 b7 = q16 -  q7;

% DCT_II
v01 =  a3 +  a5;
v02 =  a3 -  a5;
v03 =  a6 +  a2;
v04 =  a6 -  a2;
v05 =  a1 +  a7;
v06 =  a1 -  a7;
v07 =  a8 +  a0;
v08 =  a8 -  a0;
v09 =  a4 + v05;
v10 = v01 + v03;
v11 = v10 + v07;
v12 = v03 - v07;
v13 = v01 - v07;
v14 = v01 - v03;
v15 = v02 - v04;
v16 = v15 + v08;
v17 = v04 + v08;
v18 = v02 - v08;
v19 = v02 + v04;

m01 =  d0 * v06;
m04 =  d2 * v12;          % m02 and m03 replaced by shift rights
m05 =  d3 * v13;
m06 =  d4 * v14;
y03 =  d0 * v16;          % m07 directly connected to y03
m08 =  d5 * v17;
m09 =  d6 * v18;
m10 =  d7 * v19;

v20 =  a4 - (v05 >> 1);  % m02 -> v05/2;
v21 = v20 + m04;
v22 = v20 - m04;
v23 = v20 + m05;
v24 = m01 + m08;
v25 = m01 - m08;
v26 = m01 + m09;

y00 = v09 + v11;
```

```
y01 = m10 - v26;
y02 = m06 - v21;
y04 = v22 - m05;          % y03 already defined to replace m07
y05 = v25 - m09;
y06 = (v11 >> 1) - v09;   % m03 -> v11/2
y07 = v24 + m10;
y08 = v23 + m06;

% modified DST_II
w01 =  b3 +  b5;
w02 =  b3 -  b5;
w03 =  b6 +  b2;
w04 =  b6 -  b2;
w05 =  b1 +  b7;
w06 =  b1 -  b7;
w07 =  b8 +  b0;
w08 =  b8 -  b0;
w09 =  b4 + w05;
w10 = w01 + w03;
w11 = w10 + w07;
w12 = w03 - w07;
w13 = w01 - w07;
w14 = w01 - w03;
w15 = w02 - w04;
w16 = w15 + w08;
w17 = w04 + w08;
w18 = w02 - w08;
w19 = w02 + w04;

k01 =  d0 * w06;
k04 =  d2 * w12;          % k02 and k03 replaced by shift rights
k05 =  d3 * w13;
k06 =  d4 * w14;
z03 =  d0 * w16;          % k07 directly connected to z03
k08 =  d5 * w17;
k09 =  d6 * w18;
k10 =  d7 * w19;

w20 =  b4 - (w05 >> 1);   % k02 -> w05/2
w21 = w20 + k04;
w22 = w20 - k04;
w23 = w20 + k05;
w24 = k01 + k08;
w25 = k01 - k08;
w26 = k01 + k09;

z00 = w09 + w11;
z01 = k10 - w26;
z02 = k06 - w21;
z04 = w22 - k05;          % z03 already defined to replace k07
z05 = w25 - k09;
z06 = (w11 >> 1) - w09;   % k03 -> w11/2
z07 = w24 + k10;
z08 = w23 + k06;

 y1 = y08 - z01;
 y2 = y08 + z01;
 y3 = y07 - z02;
 y4 = y07 + z02;
 y5 = y06 - z03;
 y6 = y06 + z03;
 y7 = y05 - z04;
 y8 = y05 + z04;
 y9 = y04 - z05;
y10 = y04 + z05;
y11 = y03 - z06;
y12 = y03 + z06;
y13 = y02 - z07;
y14 = y02 + z07;
y15 = y01 - z08;
y16 = y01 + z08;

Y00 = (z00 >> 1);
Y01 = ( y1 >> 1);
Y02 = ( y2 >> 1);
Y03 = ( y3 >> 1);
Y04 = ( y4 >> 1);
Y05 = ( y5 >> 1);
Y06 = ( y6 >> 1);
Y07 = ( y7 >> 1);
Y08 = ( y8 >> 1);
Y09 = ( y9 >> 1);
Y10 = (y10 >> 1);
Y11 = (y11 >> 1);
Y12 = (y12 >> 1);
Y13 = (y13 >> 1);
Y14 = (y14 >> 1);
Y15 = (y15 >> 1);
Y16 = (y16 >> 1);
Y17 = (y00 >> 1);
```

Appendix C. The VHDL entity of the Scheduled Sequence Graph (SSG)

The automatically generated description of the SSG entity will look as follows.

```
entity imdct_18p_sh_9_2_SSG is
    generic ( NX_g :  positive := 32;
              M_g  :  positive := 30;
              MUL_delay_g :  Time := 5 ns;
              ALU_delay_g :  Time := 2 ns;
              REG_delay_g :  Time := 2 ns );
    port (  clk :  in std_logic;
          reset :  in std_logic;
          start :  in std_logic;
          X00 :  in std_logic_vector(NX_g-1 downto 0);
          X01 :  in std_logic_vector(NX_g-1 downto 0);
          X02 :  in std_logic_vector(NX_g-1 downto 0);
          X03 :  in std_logic_vector(NX_g-1 downto 0);
          X04 :  in std_logic_vector(NX_g-1 downto 0);
          X05 :  in std_logic_vector(NX_g-1 downto 0);
          X06 :  in std_logic_vector(NX_g-1 downto 0);
          X07 :  in std_logic_vector(NX_g-1 downto 0);
          X08 :  in std_logic_vector(NX_g-1 downto 0);
          X09 :  in std_logic_vector(NX_g-1 downto 0);
          X10 :  in std_logic_vector(NX_g-1 downto 0);
          X11 :  in std_logic_vector(NX_g-1 downto 0);
          X12 :  in std_logic_vector(NX_g-1 downto 0);
          X13 :  in std_logic_vector(NX_g-1 downto 0);
          X14 :  in std_logic_vector(NX_g-1 downto 0);
          X15 :  in std_logic_vector(NX_g-1 downto 0);
          X16 :  in std_logic_vector(NX_g-1 downto 0);
          X17 :  in std_logic_vector(NX_g-1 downto 0);
          Y00 :  out std_logic_vector(NX_g-1 downto 0);
          Y01 :  out std_logic_vector(NX_g-1 downto 0);
          Y02 :  out std_logic_vector(NX_g-1 downto 0);
          Y03 :  out std_logic_vector(NX_g-1 downto 0);
          Y04 :  out std_logic_vector(NX_g-1 downto 0);
          Y05 :  out std_logic_vector(NX_g-1 downto 0);
          Y06 :  out std_logic_vector(NX_g-1 downto 0);
          Y07 :  out std_logic_vector(NX_g-1 downto 0);
          Y08 :  out std_logic_vector(NX_g-1 downto 0);
          Y09 :  out std_logic_vector(NX_g-1 downto 0);
          Y10 :  out std_logic_vector(NX_g-1 downto 0);
          Y11 :  out std_logic_vector(NX_g-1 downto 0);
          Y12 :  out std_logic_vector(NX_g-1 downto 0);
          Y13 :  out std_logic_vector(NX_g-1 downto 0);
          Y14 :  out std_logic_vector(NX_g-1 downto 0);
          Y15 :  out std_logic_vector(NX_g-1 downto 0);
          Y16 :  out std_logic_vector(NX_g-1 downto 0);
          Y17 :  out std_logic_vector(NX_g-1 downto 0);
          done :  out std_logic;
          error :  out std_logic
    );
end imdct_18p_sh_9_2_SSG;
```

The computations are started by a positive going edge at the `start` input, while the `done` output goes high when the result is available.

Appendix D. Graphical view of simulation output

In Figure 12, the output of the ModelSim VHDL simulator is shown in graphical format for an input for the simulator given with

$$X_k = \begin{cases} 1\,, & \text{for } k = 0 \\ 0\,, & \text{for } k = 1, 2, \cdots, N/2-1 \end{cases} .$$

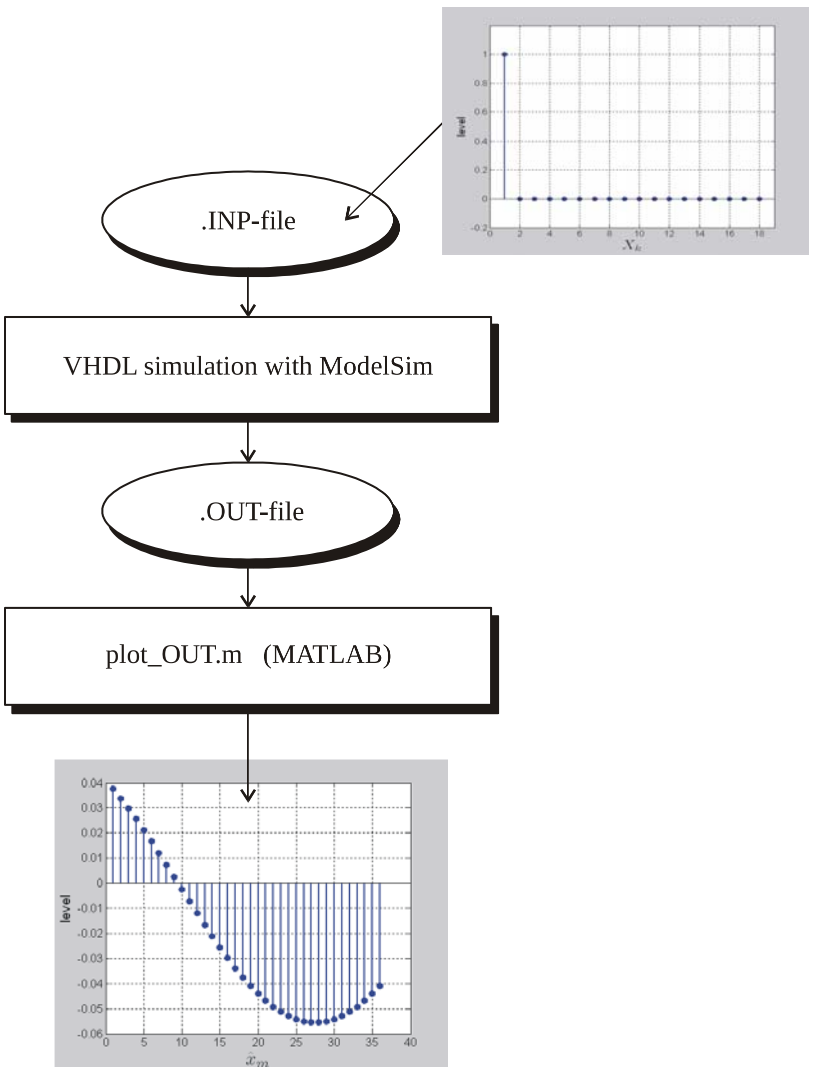


Figure 12: Simulation output for an input vector with $X_0 = 1$ and $X_{1\cdots17} = 0$ (first basis function).

This page intentionally left blank