A Discrete Fourier-Cosine Transform Chip

MARTIN VETTERLI AND ADRIAAN LIGTENBERG

Abstract — **An 8-point Fourier-cosine transform chip designed for a data rate of 100 Mbits/s is described. The top-down design is presented step by step, including algorithm modification for VLSI suitability, architectural choices, testing overhead, internal precision assignments, mask generation, and finally, verification of the layout. A high-level language (C) design tool was developed concurrently with the layout. This tool allows mimicking exactly the different representations of the algorithm: software, mask, and chip. This provides an automatic cross-checking at all design stages. The VLSI environment created by this tool, as well as existing powerful CAD tools, made a fast design-time possible.**

I. Introduction

HIGH-SPEED computation of the discrete cosine transform (DCT) [1] is often required, typically in transform image coding [2], polyphase filter banks [3], and fast Fourier transform evaluation [4]. A VLSI chip realizing a small length DCT at very high speed is desirable, both for image coding and for discrete Fourier transform (DFT) evaluation. Implementations have been realized with assemblages of MSI hardware [5], [6], but, to our knowledge, VLSI implementations have only been proposed [7], [8]. The realized design described in this paper is not only different from the previously proposed ones, but its concepts can be easily extended to larger transform sizes and/or higher precision when finer mask design rules and associated chip processing become accessible.

Implementation of an 8-point DCT chip working at 100 ns per data sample with 10-bit input and 12-bit output precision is presented. It turned out that in the 2.5 μm technology that was available at the time of the design, the realized design was about the upper limit of what could be fitted onto 35 mm^2 of available silicon. Parallel arithmetic is used to satisfy the high data rate. Since latency is usually of no consequence in this type of computation, the chip is fully pipelined.

The fast Fourier-cosine transform algorithm (FFCT) [4] was chosen for its minimum number of multiplications. The flowgraph was modified in order to obtain a simpler structure and divided into pipeline stages. Each stage was designed with input memory, a permutation network, and an arithmetic unit which performs the required operations. With the help of simulations, the precision of constants and data was determined. A testing overhead was included which allows the testing of stages independently on the chip.

In parallel to the design of the actual chip, a software image in a high-level language (C) was made. This image allowed us to check all levels, e.g., algorithm modification, finite precision effects, and also to automatically generate test vectors, both for the layout and for the chip. A future paper will describe this design tool for mapping digital signal processing (DSP) algorithms into VLSI [9]. The concept of this tool, called MOVAL, was developed and partly implemented in parallel with the design of the actual chip.

The above approach together with the use of the powerful symbolic layout system MULGA [10] allowed a design time of less than 3 months (which is short considering the fact that we had no previous VLSI experience).

II. The FFCT Algorithm

The discrete cosine transform of length N for a real vector $x(0), x(1), \cdots x(N-1)$ is defined as

$$DCT(k, N, x) := \sum_{n=0}^{N-1} x(n) \cdot \cos\left(\frac{2\pi(2n+1)k}{4N}\right), \quad k = 1, \cdots, N-1. \quad (1)$$

For $k = 0$ there is an additional scaling constant of $1\sqrt{2}$.

The main features of the FFCT algorithm are that it allows evaluation of both the DFT or the DCT and that only real arithmetic is used. For sample point lengths which are powers of 2, it achieves the lowest known number of operations for the DCT, and for the DFT for real, complex, or symmetric signals, as well [4], [11]. Especially in the case of the DCT, it needs substantially fewer multiplications than the algorithm of Chen *et al.* [12]. This is crucial in the VLSI context, as will be pointed out.

The principle of the FFCT algorithm is briefly recalled (details are found in [4]). As shown in Fig. 1, a DFT of length N is mapped into a DFT of length $N/2$ and 2 DCT's of length $N/4$, and this at the cost of $3N/2-2$ additions. Fig. 2 shows that mapping of a DCT into a DFT of the same length and output rotations. As an example, the computation of a 32-point DFT with the FFCT algorithm is given in Fig. 3. The important points are that rotations represent the main computational load and that a chip performing a DCT of length N can be used as a building block for a processor performing a DFT of length $4N$.

Manuscript received October 1, 1984; revised February 20, 1985.
M. Vetterli is with Ecole Polytechnique Federale de Lausanne, CH-1007 Lausanne, Switzerland. He is on leave at AT&T Bell Laboratories, Holmdel, NJ 07733.
A. Ligtenberg is with AT&T Bell Laboratories, Holmdel, NJ 07733.
IEEE Log Number 8406180.

0733-8716/86/0100-0049$01.00

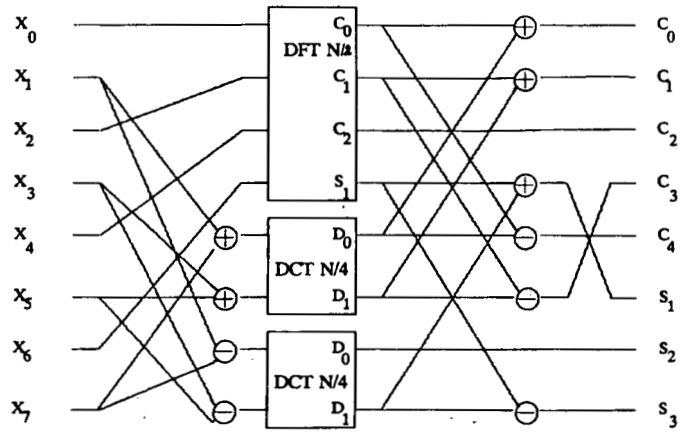


Fig. 1. Computation of the DFT by means of a smaller DFT and DCT's.

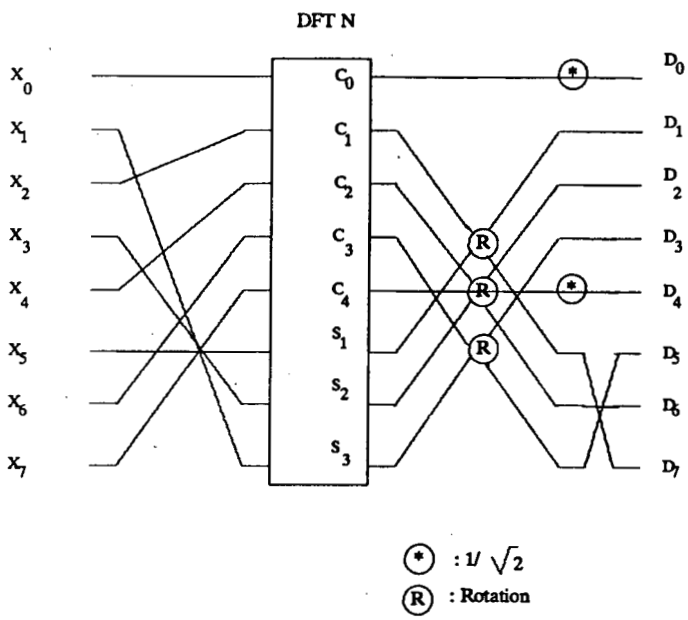


Fig. 2. Computation of the DCT with a DFT of same size and rotations.

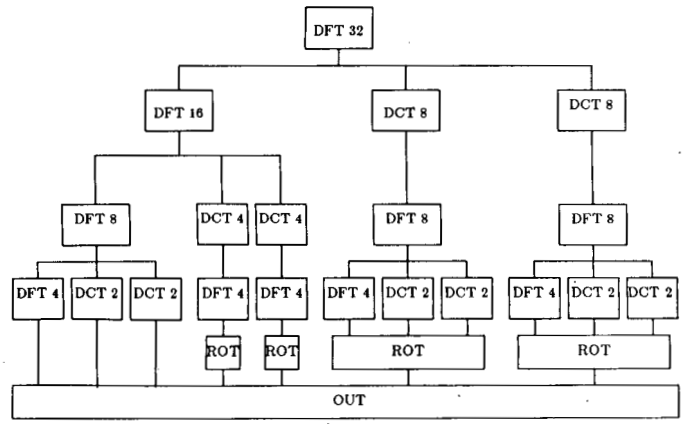


Fig. 3. A 32-point DFT computed with the FFCT algorithm.

DFT-N DFT of length N.
DCT-N DCT of length N (the inputs are obtained from sum and differences).
ROT Rotation.
OUT Output recombination (sum and differences of outputs).

For two-dimensional DFT's or DCT's the most efficient algorithm is based on polynomial transforms [13], [14]. But for simplicity of control, generally one uses a one-dimensional transform in a row-column fashion. Thus, a chip that computes a transform of length N is a useful building block for an $N \times N$ transform.

III. A Structured Tool to Implement DSP Algorithms Into VLSI

The block diagram containing the architectural information (see Section IV, Fig. 6) indicates that a number of arithmetic units and register/permutation networks need to be designed. We had less than three months to work together and concluded that, with this severe time limit, the chip was too complex to be laid out and verified completely by hand. Therefore, a structured design tool became not only desirable but absolutely necessary.

The design tool named MOVAL (modular versatile algorithmic language [10]) allows us to analyze the precision issues, to test automatically the layout (on switch-level simulation as well as for timing consideration), and to verify the working of the produced chip. Even more, it allows automatic generation of the layout of primitive functions such as the multiplier. In short, it is a topdown approach for structured VLSI design. These characteristics are realized by the creation of a software image for the different description levels of the algorithm with MOVAL. The highest level contains a description of the algorithm using floating point arithmetic, whereas at the lowest level an exact hardware description of the chip is formed. Different levels can be compiled together in order to test different parts of the hardware design. Every description level is automatically cross-checked against the highest level thus reducing the number of human errors to a minimum. For testing and simulation purposes, a data file is created containing the input data, clock and control signals, and the correct output data. The input data can be generated manually, but can also be obtained from real-world data. The choice of real input data is important to analyze the finite precision effects. For example, in the case that a DCT chip is designed to code pictorial data, real pictures can be used as input data to see the effect of finite precision arithmetic in order to make a tradeoff between available chip space and picture quality.

IV. Algorithm Modification for VLSI Suitability

A special-purpose machine allows a better match between required computational power and arithmetic units than would a general-purpose computer. Boldly stated, the philosophy is: put a processor whenever and wherever you need one. The DCT chip design makes an attempt to place computational elements only where they are required, thus maximizing their use. On the other hand, if the DCT algorithm runs on a general-purpose signal processor, some of its computational elements will stay idle at certain stages of the algorithm, thus making a bad use of time and/or silicon area. A first estimate shows that a commercial

signal processor (TMS320) would require at least a factor of twenty more times to solve the 8-point DCT transform than does the DCT chip.

In general, a design reflects the interplay between the algorithm and the outside-world constraints. For the DCT chip, the constraints were a one-chip design and a minimum data rate of 100 Mbit/s. The available technology was 2.5 μm CMOS, and preliminary investigations showed that putting an 8-point DCT on one chip would be a reasonable goal. (The available surface was about 35 mm^2.) Yet another constraint was a short design time (less than 3 months), which also dictated some design choices.

In the flowgraph of an 8-point DCT computed with the FFCT algorithm as shown in Fig. 4(a), a lack of regularity is apparent. In the VLSI context, this means that the control and the routing will be complex thus costly in size and time. Note as well that the output is not ordered, which is not desirable in general. With a post permutation one can obtain ordered output samples.

There are many possible flowgraph transformations that can be applied to a flowgraph like the one in Fig. 4(a) and that will retain the same arithmetic complexity. To our knowledge, there are no flowgraph transformations that will lower the operation count and no transform will result in an in-place in-order algorithm. Therefore, the flowgraph will be transformed in such a way that similar operations appear grouped. Actually, an increase in the number of operations is tolerated just for the sake of better structure.

The chosen solution moves on addition/subtraction operator from stage 3 [Fig. 4(a)] to the last stage and merges it with the scaling by $1\sqrt{2}$, thus producing a rotation [see Fig. 4(b)]. Now there are 4 output rotations, and stage 3 has only 2 multipliers left. While this means an implicit increase of the computational load by 1 multiplication and 1 addition, the regularity will pay off in implementation simplicity.

The precision issue has to be addressed at a very early state of the design. Floating point was out of the question due to the high speed and chip area constraints. Since image processing is a potential field of use for the chip, the aim was a precision of at least 8 bits. Simulations showed that multiplying constants having a bit length of 80 percent of that of the input data could be used with negligible loss of precision. Linked to word width is also the data growth issue. Since overflow cannot be tolerated (for example, in transform coding, it would mean that the relevant information would get lost), one has either to provide guard bits, or to truncate after each operation. The first method requires absurdly large data paths (each addition means a growth of 1 bit, and each multiplication with L- and M-bit inputs has an output of $L + M$ bits, where L and M are usually of the same order). The second method results in the fact that certain low-level input sequences simply disappear.

As might be expected, a compromise solution was chosen. While a certain number of guard bits are provided, the outputs of certain operations (like the multiplications) are truncated so as to bound the data growth. Both for the flowgraph manipulation and the data precision analysis, a software image was very helpful. This software image was written with floating point arithmetic for the flowgraph emulating version, and with appropriate integer arithmetic for the data precision analysis version.

V. Architectural Issues

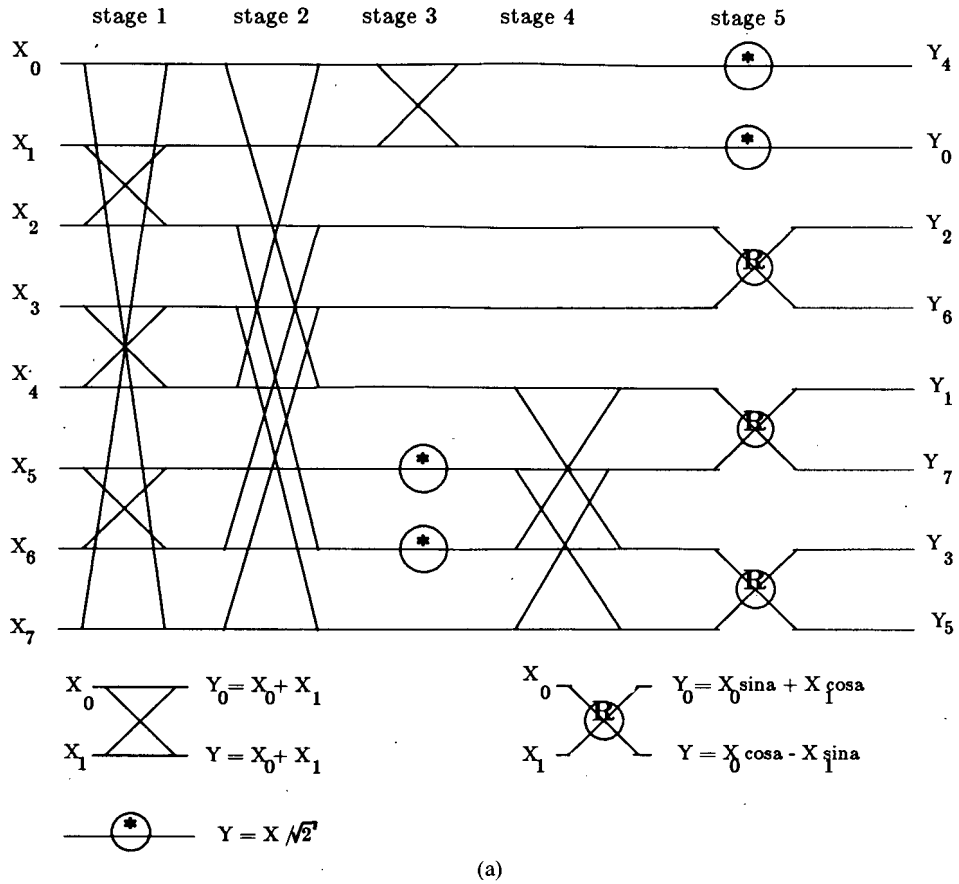
Since the chip is basically a number cruncher, the choice of two's complement number representation was an obvious one. Another early concern was what type of arithmetic should be used. The two opposite solutions are serial or parallel arithmetic. A mixing of both was excluded because the design would become too complex. The advantage of serial arithmetic is its simplicity and its good time/area performance when used appropriately. One of the drawbacks is that the rounding problem cannot be handled without a significant loss in performance (large number of guard bits) and rounding overhead. Multiplications require $2L$ partial adding times [17], which makes them slow. Therefore, high throughput requires many serial operators in parallel, which involves a large control overhead.

In contrast, parallel arithmetic allows high throughput without overhead. The design of parallel arithmetic is rather straightforward, and the rounding/truncating can be handled easily. Problems may occur when the word length is too large, resulting in a carry time that is not acceptable. Then the solution is to use look-aheads or pipelined arithmetic, but both represent a more involved design.

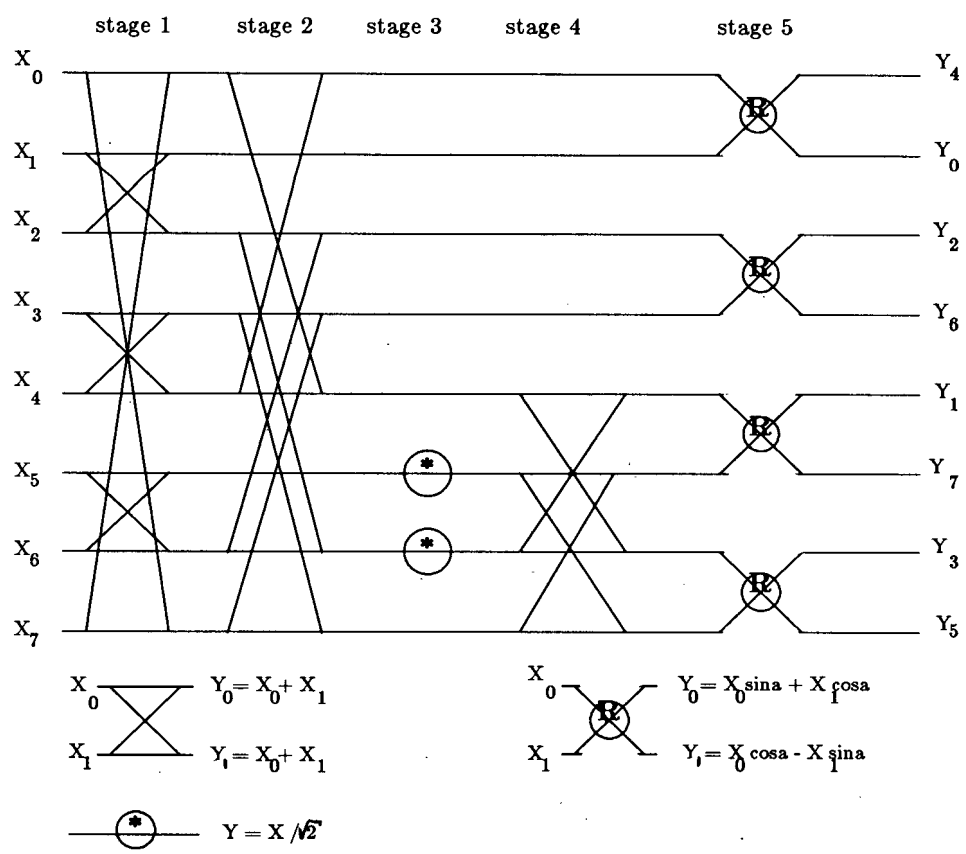
As an example the computational requirements of stage 5 are analyzed. The specification for the processing time of one sample is 100 ns. Four rotations containing 12 multiplications must be performed in 800 ns (8 samples). With a multiplication time in the range of 100–200 ns, at least three multipliers in parallel are necessary.

These considerations lead to the use of parallel arithmetic. Furthermore, all the stages have to be pipelined in order to achieve the desired performance (one rotation unit is already necessary just to meet the throughput requirement of stage 5). Pipelining is a natural way to go, since each stage has a specific computational task (e.g., plus and minus operation, rotation) that can be met with a matching arithmetic unit. The chip is, therefore, divided into 6 pipelined stages, where the first 5 realize the calculations and the 6th one is used to reorder the data. Stage 1, 2, and 4 will have add-subtract units, stage 3 a multiply-by-constant unit, and stage 5 a rotator unit. Once the two basic choices have been made, namely parallel arithmetic and pipelined computation, the design of the chip is set in a manageable context.

The dataflow is examined next. Besides the multiply-by-constant in stage 3 (which is a scaling) all other oper-



(a)



(b)

Fig. 4. Flowgraph of an 8-point DCT computed with the FFCT algorithm. (a) Original flowgraph. (b) Flowgraph after modification for VLSI suitability.

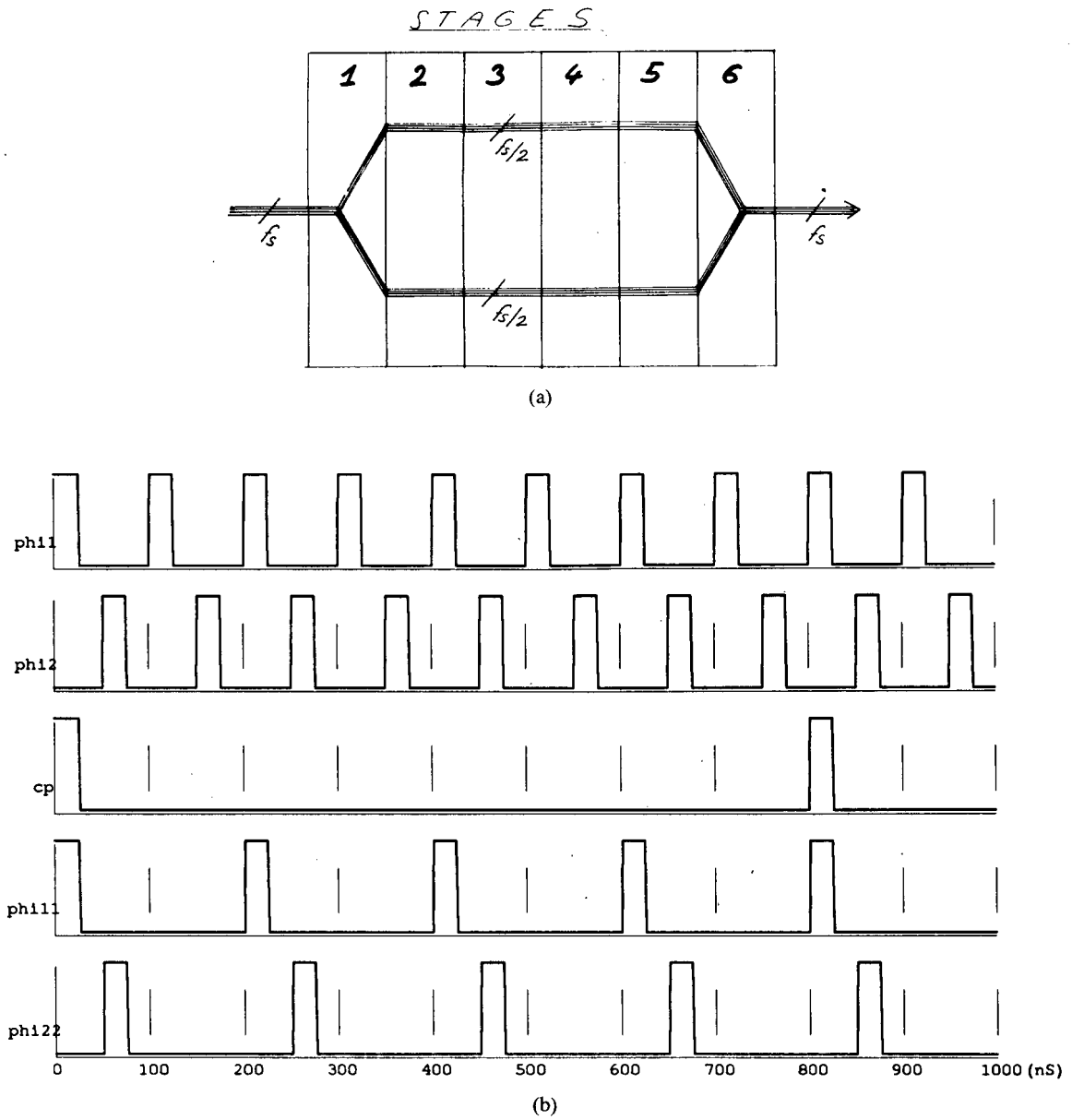


Fig. 5. Architectural concept. (a) Data flow concept through the six computational stages. *Fs* is the sampling rate of the data (10 MHz). (b) Basic control signals.

ations are "rotations" (sum and difference of 2 samples can be regarded as a rotation by 45° followed by a scaling by $\sqrt{2}$. One interesting property of a rotation is that it transforms 2 inputs into 2 outputs (in contrast to the more common addition or multiply operation which transform 2 inputs into 1 output), and that, therefore, a $2L$ datapath is quite natural when working with L-bit numbers. This parallelism also allows the arithmetic units to work at half the input data rate, which in the case of the general rotator of stage 5 is very welcome (it would be difficult to design parallel multipliers with a 100 ns multiply time, at least without going to pipelining the multiplier itself).

Timing and control are examined next. The usual two-phase nonoverlapping clocking scheme was chosen, which is the most straightforward clocking scheme for a synchronous design [15]. Phase 1 is used for data transmission, and phase 2 for data evaluation. Since both input and output were chosen bit parallel/sample serial, the first and last stage work at the data rate for the input and output, respectively. In between, the data rate is halved. Actually, phase 2 is chosen to be longer than phase 1, since the time-consuming part is the data evaluation in the parallel arithmetic. Besides the clocks, a clear signal is used to define the beginning of each new 8-samples sequence. Fig. 5 schematically shows the dataflow and data rate as well as the clocks (phi11 and phi22 are the half-rate clocks).

The overall dataflow and timing, and the need to pipeline the computation, leads to the need to separate the stages by registers. The registers also include the permutations required by the flowgraph. A first sketch of the resulting design is shown in Fig. 6.

Note that this architecture is by no means the only solution to the given initial problem of computing an 8-point DCT on one chip at a 100 Mbit/s data rate. But

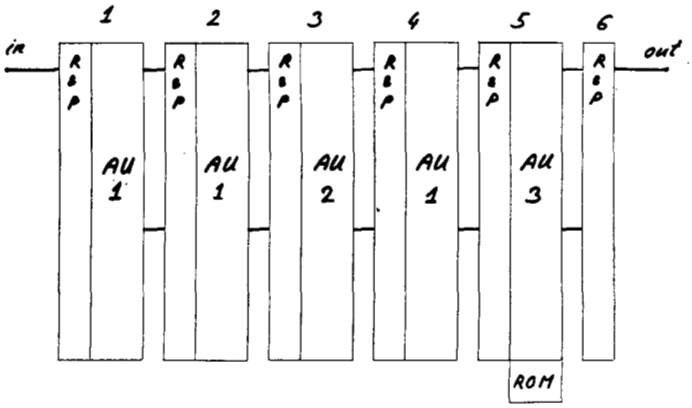


Fig. 6. Overall architecture of the DCT chip.

AU1 Adder and substractor.
AU2 multiply by $1/\sqrt{2}$.
AU3 rotator.
ROM Read only memory with the constants for the rotator.
R&P Register and permutation.

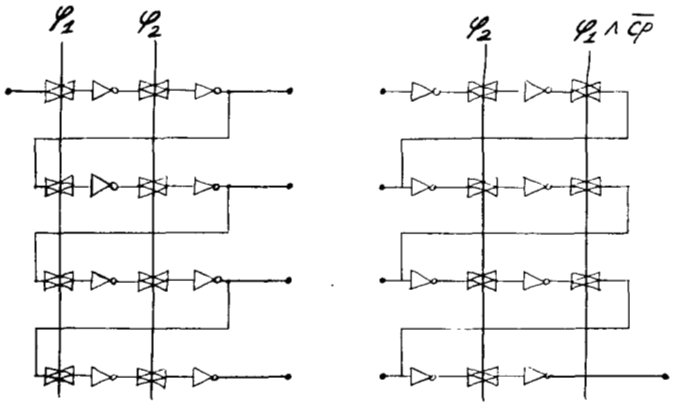


Fig. 7. Shift registers. (a) A 4-bit shift register with serial input and parallel output. (b) A 4-bit shift register with parallel input (which is written through passgates with a copy signal) and serial output.

one of its nice features is that it is trivial to change its arithmetic precision, and that the same concept can be applied to compute larger or other transforms (requiring different arithmetic units). Furthermore, it will be seen that registers provide a built-in testability feature that is quite useful when testing the layout and/or the chip.

VI. Building Blocks

In a dataflow concept with parallel arithmetic, it is advantageous to store and process bits having the same weight at the same physical location. Thus, the same bit parallel concept is used in the storage and the processing. Next, the main building blocks, registers, adders, subtractors, and multipliers are described.

Since the registers represent a substantial overhead in the pipelined architecture of Fig. 6, a compact design was important. Note that the permutations in Fig. 4(b) are rather complex, and that therefore, it was important to be able to realize arbitrary permutations between stages (i.e., that the register design does not restrict the possible permutations). Therefore, the registers were designed to function in three steps: 1) acquire the data from the previous stage, 2) permute the samples arbitrarily, and 3) put out the samples to the arithmetic units. Step 1 is realized with a shift register having serial input and parallel output; Step 2 with an arbitrary permutation network; and Step 3 with a shift register having parallel input and serial output. Note that the shift registers are of length 4 or 8 depending on which stage they are located, but that the permutation network always maps 8 input samples into 8 outputs (the flowgraph being not completely separable into 2 parallel paths). The 4-bit shift registers for steps 1 and 3 are depicted in Fig. 7. To realize a full register/permutation block (as required in Stage 2, for example), two such registers are put on each side of a permutation network. A copy signal (occurring every 800 ns) moves the content of the first set, through the permutation network, to the

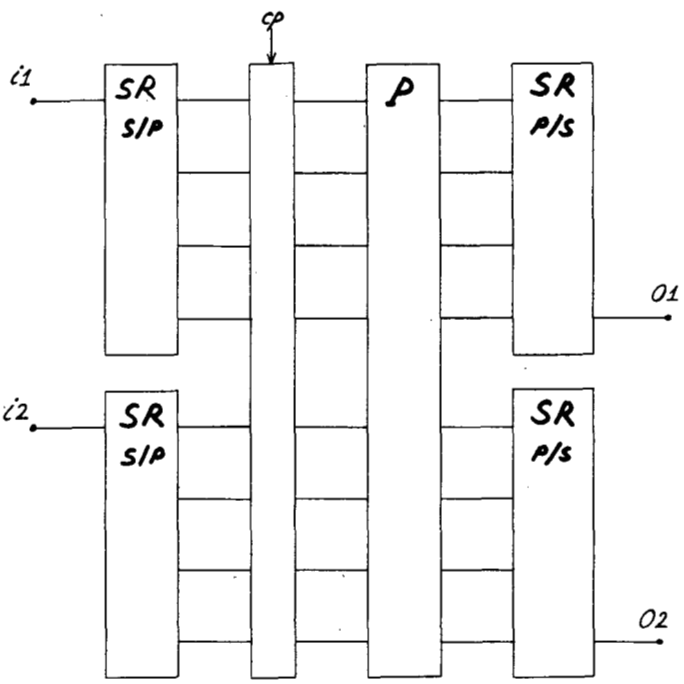


Fig. 8. Registers of stages 1–6.

SR Shift register, 4 bits.
S/P Serial in, parallel out.
P/S Parallel in, serial out.
CP Copy signal to activate pass gate array.
P Permutation.

second set as shown in Fig. 8 (where a one bit slice is shown). The permutation network can be realized with pass-gates (which are more flexible, since the permutation can be programmed), but for simplicity and compactness, a routing network was used in the chip.

Full adder designs are examined next. From the 3 inputs a, b, and c_{in}, (which can be interchanged), one devises s and c_{out} according to

$$s = a \oplus b \oplus c_{in}$$
$$c_{out} = ab + bc_{in} + ac_{in}. \quad (2)$$

The minimum gate CMOS implementation of the full

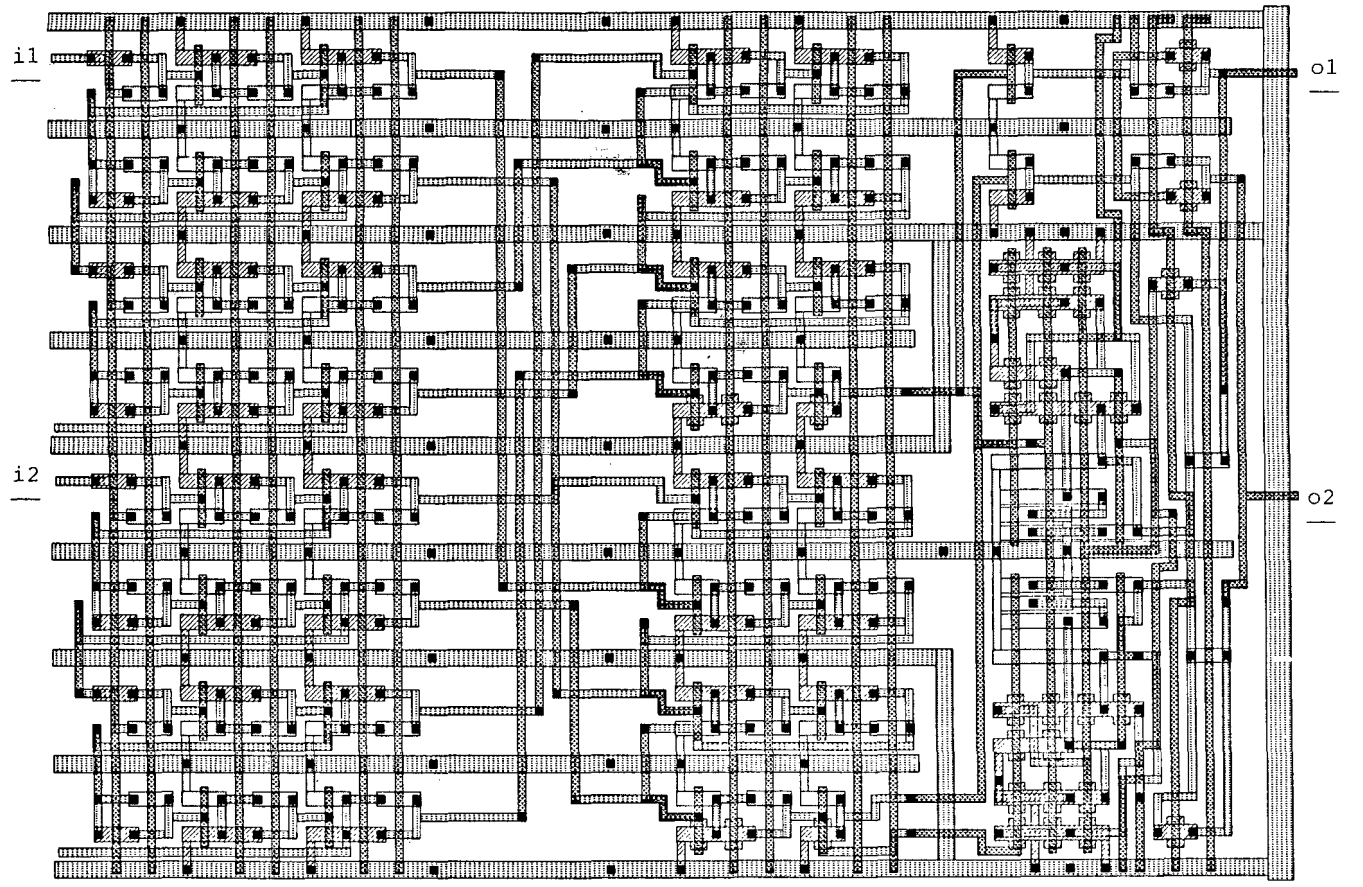


Fig. 9. Symbolic layout of 1-bit strip in stage 2. From left to right; shift registers, permutation, shift registers and 2 full adders.

adder provides $\bar{s}$ and $\bar{c}_{out}$, but, since it is easy to show that

$$\bar{s} = \bar{a} \oplus \bar{b} \oplus \bar{c}_{in}$$
$$\bar{c}_{out} = \bar{a}\bar{b} + \bar{b}\bar{c}_{in} + \bar{a}\bar{c}_{in} \quad (3)$$

one can cascade minimum full adders provided that every other bit of the input is reversed.

This results in the fact that every second output is reversed, but this can be accommodated in the next adder stage. Therefore, only the final and last arithmetic unit needs inverters for every other bit. A minimum adder cell was designed so that two of this type would match the registers (one to compute the sum, the other the difference of 2 samples). The blocks described so far complete basically 1 bit of the stages 1, 2, and 4. A remark is appropriate here. Stage 1 performs 4 addition/subtraction operations, while stage 2 and 4 perform 3 and 2 addition/subtraction operations, respectively. While in traditional complexity terms, stage 2 and 4 are less complex than stage 1, in VLSI it is just the opposite. Stage 1 has just an adder/subtractor cell, while stage 2 and 4 require additional routing and passgates in order to bypass the operations at certain times! In the FFCT, algorithm arithmetic complexity has been traded off against regularity, and thus another algorithm involving more operations can have a more regular structure.

Fig. 9 shows the symbolic layout of a one bit slice in stage 1. At left are the two 4-bit serial-in parallel-out shift registers, followed by the permutation network, and the two 4-bit parallel-in serial-out shift registers. At right are the two full adder cells (the subtraction is achieved by two's complementing the input to be subtracted, namely, take the inverse of the bits and add 1 at the lowest carry input). On top of the adders is the bypass.

Both the driving of the shift registers and the generation of addresses for the ROM's require a set of control signals which have to be devised from the three basic control signals phi1, phi2, and clear. This was realized with ring counters and the appropriate logic (signals and their complement being generated by complementary logic as well, so as not to incorporate any unnecessary delay in the control). The set of control signals is depicted in Fig. 10.

Another important building block is the multiplier. The four multipliers used on the chip cover a large part of the total chip area. Attention must be paid to the size of the multiplier cells because a linear growth of the number of bits results in a quadratic growth of the multiplier.

A two's complement multiplier was developed by modifying the Baugh–Wooley algorithm [18]. The modification consists of using a large number of different cells, as depicted in Fig. 11. This results in a smaller overall area. We could afford to have a large number of different cells because an automatic multiplier generator was built. How-

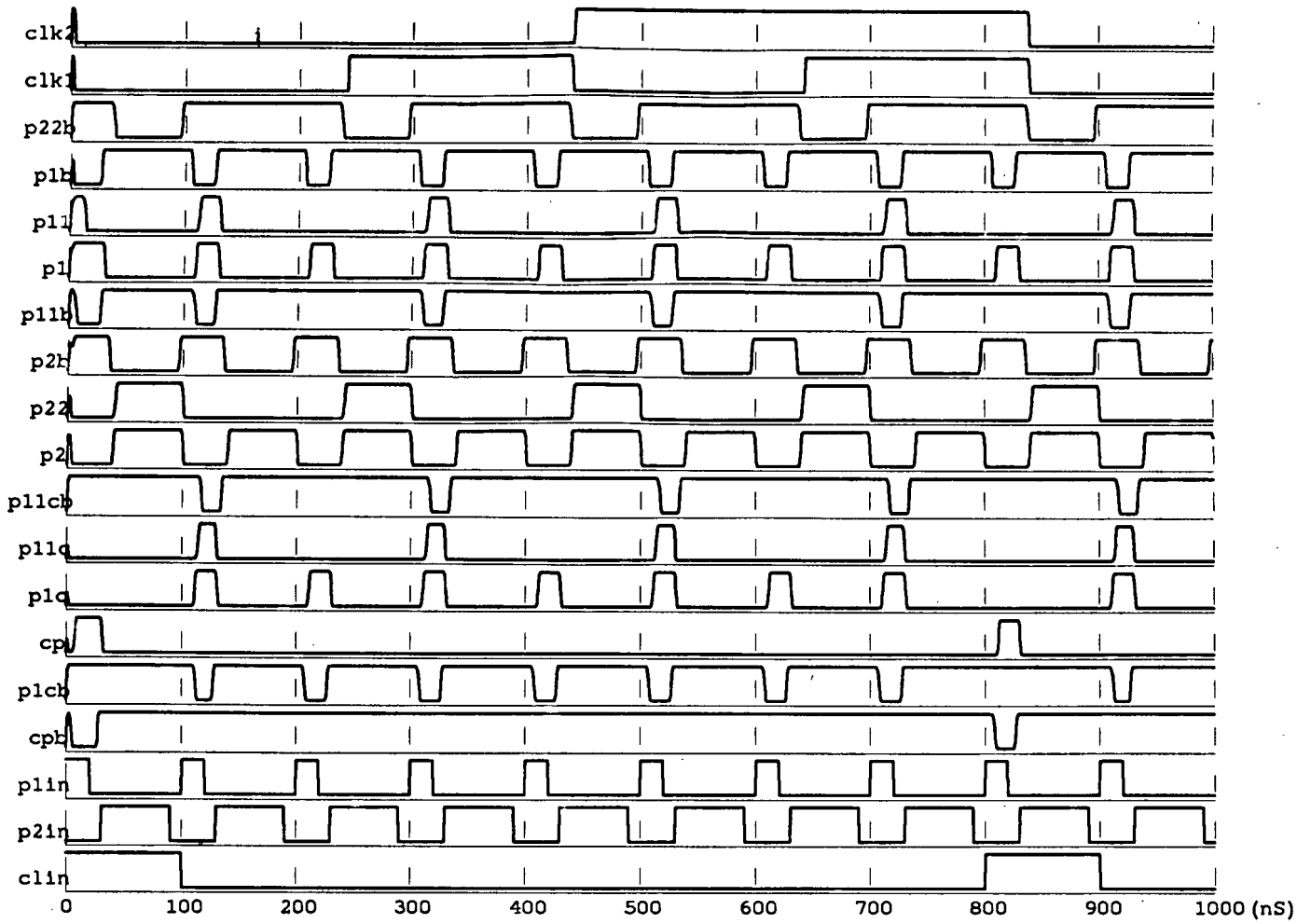


Fig. 10. Control signals derived from p1in, p2in, and c1in. This is the result of an analog simulation.

ever, the generator does not handle in a special way the multiplication by a constant. Such a situation occurs in stage 3, where a multiplication with a constant of $1\sqrt{2}$ or (0100100110) in 10 bit, two's complement data representation is performed. By stripping the columns that correspond to the zero's in the constant, a smaller multiplier can be obtained. The multiplication constants for stage 5 are stored in ROM's. These ROM's are adjacent to the multiplier blocks.

The MULGA system, which was used for the design of the cells, allows a hierarchy (by instancing cells or modules). Therefore, it is straightforward to assemble the above described building blocks to create the stages and, finally, the whole chip.

VIII. Testability

With high chip complexity, the testability issue has growing importance [18]. Thus it was decided to include built-in test capabilities [19]. This is achieved by using the existing registers. Only a small overhead was necessary in order to be able to write or read data in any of the registers to load test vectors or evaluate the computational results.

The principle is fairly simple. When the chip is in test mode, one can put all registers of a given stage in series, and then write data into them from an input pin (and this at an arbitrary slow speed). Once this is done, the chip can run at full speed through a full cycle of computations (800 ns in fact). Then, the set of registers can be put in a series and the result of the computation can be read out (again at slow speed). In that fashion, each stage can be tested individually. The principle is sketched in Fig. 12.

Again, the MOVAL tool is of great help because it can automatically generate the test vectors and check the result, both on the symbolic layout and on the chip.

A. An Example: The Rotator Design

The rotator is the most complex part of this chip. Given two inputs X_0 and X_1, it evaluates the outputs as

$$Y_0 = \cos\alpha X_0 - \sin\alpha X_1$$
$$Y_1 = \sin{:}\,\alpha X_0 + \cos\alpha X_1. \quad (4)$$

A parallel arithmetic approach rather than a CORDIC solution [20] was chosen, mainly because of design consistency.

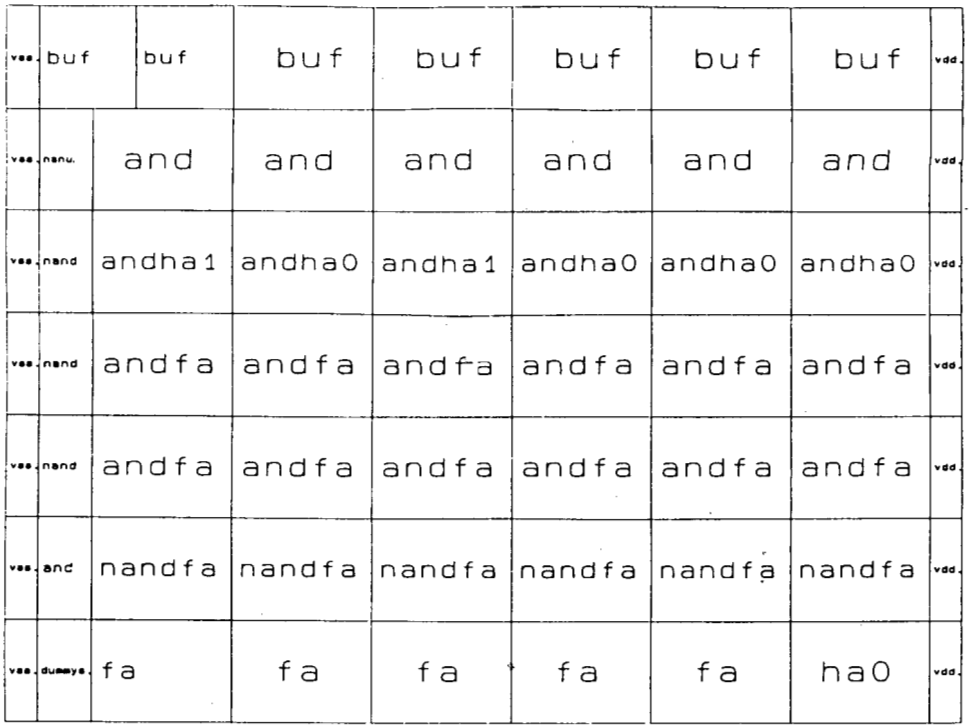


Fig. 11. Floor plan of a multiplier, where buf, andfa, andha0 and andha1 stand for buffer, AND with fulladder, AND with halfadder with input carry zero, and AND with halfadder with input carry one, respectively.

Instead of evaluating (3) which requires 4 multipliers and 2 adders, we resort to the well-known complex multiply algorithm [13], which requires only 3 multipliers and 3 adders. Equation (3) is evaluated as

$$\begin{aligned} Z_0 &= X_0 \\ Z_1 &= X_0 + X_1 \\ Z_2 &= X_1 \\ Z_3 &= Z_0 \cdot (\sin\alpha - \cos\alpha) \\ Z_4 &= Z_1 \cdot \cos\alpha \\ Z_5 &= Z_2 \cdot (-\cos\alpha - \sin\alpha) \\ Y_0 &= Z_4 + Z_5 \\ Y_1 &= Z_3 + Z_5. \end{aligned} \tag{5}$$

While this gain may seem futile in a software implementation, it pays off well in the VLSI context, since one multiplier is traded for an adder. Actually, a glance at the chip layout shows that the four-multiplier version would simply not fit with the used technology.

In order to obtain maximum throughput, the computation of the rotation is again pipelined, and additional registers are added at the end so that the latency of the rotator is 1 μs. This allows the output to be again in phase with the general 800 ns cycle. The architecture of the rotator stage is given in Fig. 13.

A remark on bit precision is now appropriate. Since the number of bits is small, the handling of truncation is delicate. On the one hand, unnecessary truncation produces too few significant bits, but overflow must be absolutely avoided. On the other hand, truncation is necessary in order to handle the data blow up. First, we note that all constants have to be scaled to fit within $(-0.5,\cdots,0.5)$. This means that the constants C_0 and C_2 have to be divided by 4 (considering only shifting oper-

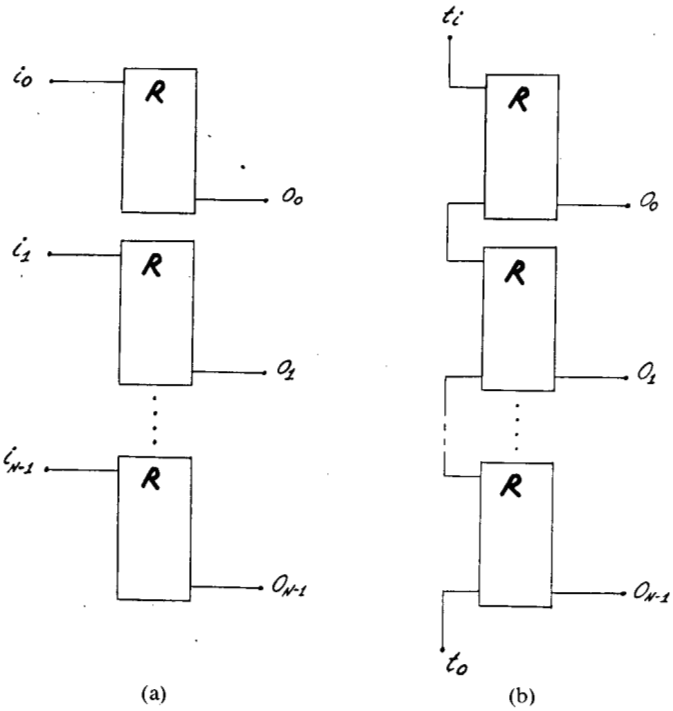


Fig. 12. Testing overhead. (a) Normal mode. The registers (R) are disconnected. (b) Test mode. The registers are put in series, and can be written from t_i or read from t_0.

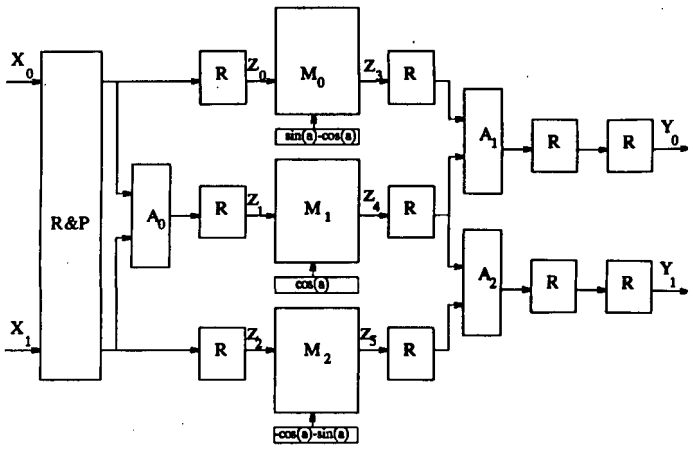


Fig. 13. Rotator block diagram.

R&P	Registers and permutations.
R	Register.
A_i	Adder.
M_i	Multiplier.

sin(a)-cos(a), cos(a), -cos(a)-sin(a), multiplicant constants C_0, C_1, C_2, respectively.

ations), but C_1 only by 2. But the input to M_1 (multiplier) is the result of an addition, and if we do not want M_1 to be larger than M_0 or M_2, we have to take the L-most significant bits of the addition (which is a division by 2). Then, since C_1 is only divided by 2, all multiplier outputs have the same scaling. Moreover, it is easy to see that, unless the constant is -0.5, the 2 most significant bits of the output are always the same. Since we know that this worst case never happens, we can disregard the most significant bit, and have only a division by 2. When looking at the adders A_1 and A_2, and knowing that their input has L significant bits, one might expect an $L+1$ bit result. Interestingly, this is not the case, because an input vector can have a growth in its length of at most $\sqrt{2}$ (for example, when $[-0.5, -0.5]$ is rotated by 45°), and since a division by 2 was already done, the result of $A_1 + A_2$ never produces a carry. These remarks should stress that a careful analysis allowed us to gain 2–3 bits of precision, which is non-negligible when working with small-bit data sizes.

The final design of stage 5 used the same registers and adders as in the previous stages, as well as 3 general multipliers with ROM's containing the various constants. Actually, a special-purpose 3-multiplier (made up of 3 interleaved multipliers) would better fit into the bit-parallel philosophy, but no time was available for such a specialized design. The layout of the stage 5 is shown in Fig. 14.

This rather complex stage will be used to demonstrate the use of MOVAL in handling its complexity. The rotation unit, described in floating-point arithmetic, is shown in Fig. 15. The multiplier constants are C_0, C_1 and C_2, in and out are arrays containing the input and output samples respectively, whereas the rest of the variables are used to share the intermediate results. This part of the program is used both to test the validity of the equations describing the reduced rotation and to crosscheck successively more detailed descriptions of the algorithm.

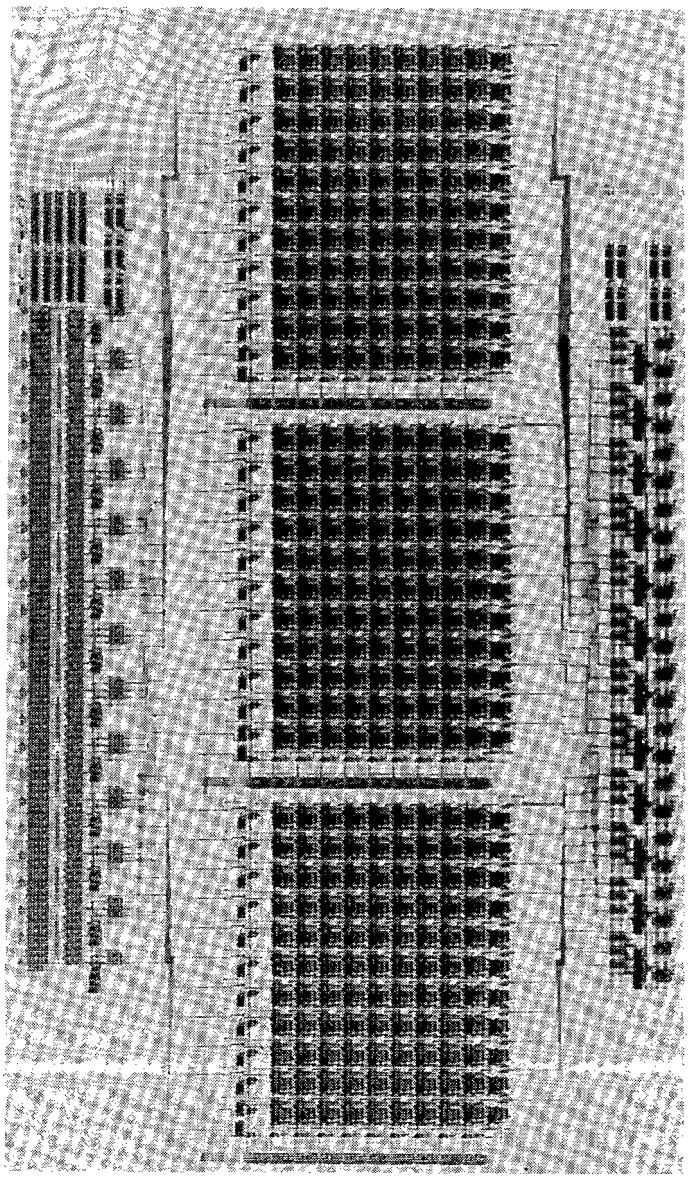


Fig. 14. Symbolic layout of rotator (stage 5). Note the ROM's which are attached to the bottom of each multiplier.

The exact software image of the rotator hardware is found in Fig. 16. All the variables are expressed in a two's complements bit-restricted data format. One can distinguish additions (add), registers (reg_2 phase), multiplexing routines (multiplex_4), two's complement multipliers (multi2c), and routing/masking networks (mask). Besides the need for input and output data, the routines also require control and timing information.

This description allows us to generate in a simple manner a data file containing test vectors. The same data file can be interfaced to different kinds of simulators or directly to the chip.

IX. Realization of the Chip

Having realized the symbolic layout of the stages, results from the compaction program that produces the mask showed that 12-bit arithmetic would be the upper limit of

```
ts = in[0] + in[1];
m1 = in[0] * c0;
m2 = ts * c1;
m3 = in[1] * c2;
out[0] = m2 + m3;
out[1] = m1 + m2;
```

For all variables, floating-point arithmetic is used.

Fig. 15. C program of the rotator in floating-point arithmetic.

what could fit on a single chip using the technology described earlier. Thus, the final implementation uses a 12-bit data path and 10-bit constants.

It also became clear that a software image of the chip in MOVAL was not only useful, but a requirement in order to be able to test it. Because of the data permutations and the bit reversals involved, it was just impossible to test the layout by hand, not to speak of the physical chip.

Fig. 17(a) shows the floorplan with the cell outlines, and Fig. 17(b) depicts the symbolic layout of the chip. Stages 3 and 5 are easily recognizable at the multipliers, and the control is on top of stage 3.

Because of the complexity of the chip (34 000 transistors) and the fact that a lot of different cells are used, the chip had to be assembled on the mask level (rather than on the symbolic layout level). This assembly is continuing, and a final mask set is expected soon for submittal on the next "silicon shuttle." Initial actual chips are expected in mid-1985.

X. Conclusions

The realization of a one-chip 8-point DCT processor with a data rate of 100 Mbits/s has been described. The chip uses 2.5 μm CMOS technology and contains about 34 000 transistors. It was shown how an efficient but rather involved fast algorithm was mapped into a VLSI circuit. Three factors were of great help in this design process: 1) a powerful symbolic layout system (MULGA), 2) a structured top-down design tool (MOVAL) that realized a software image of the chip being realized, and 3) a simple and clear design concept (parallel arithmetic, pipelined computational stages). We hope that this design example will motivate even more "algorithm" people to actually "siliconize" their ideas.

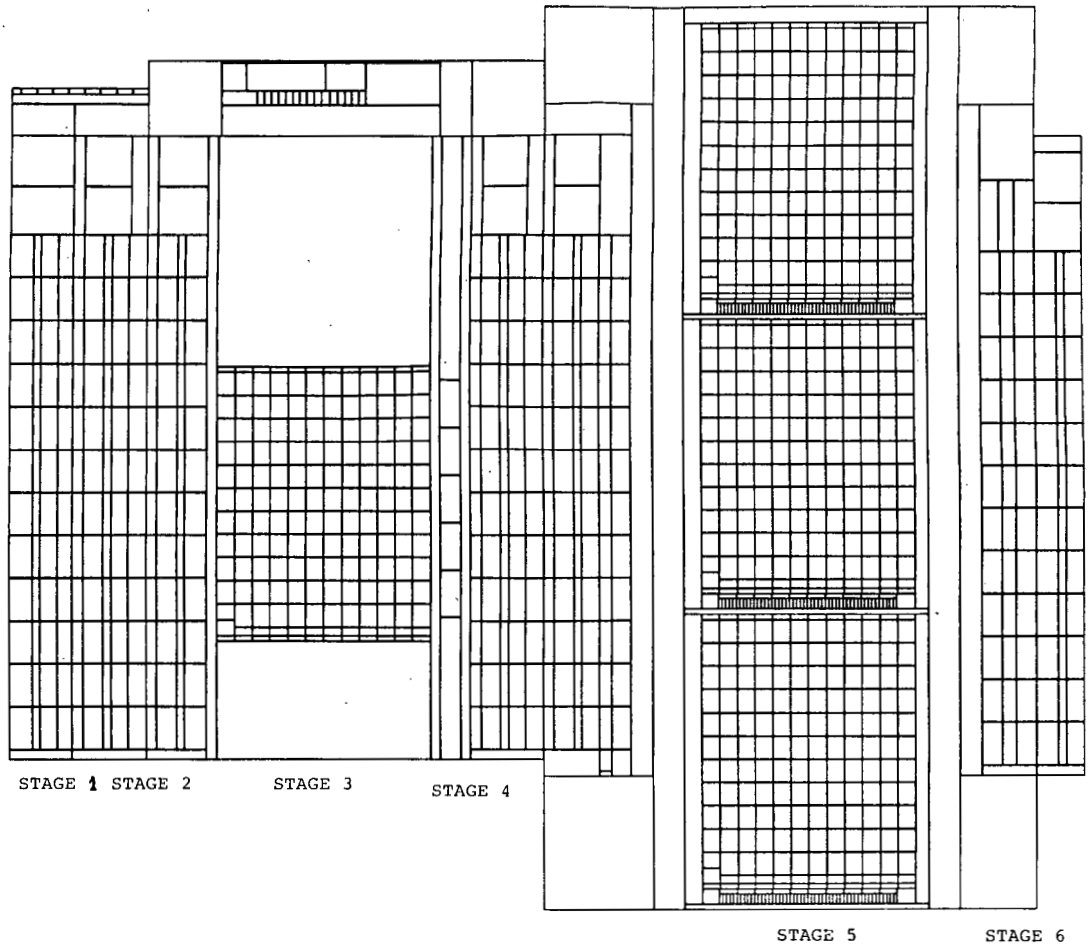
Appendix

Below, the linear code written in Pascal for an 8-point DCT is given. It uses nine memories for its computation, which is minimal. The operations are numbered (a_7 = 7th addition, m_5 = 5th multiplication) and the constants are given at the beginning. The output ordering is given at the end. The linear code for DCT's and FFT's for real input data is available on request for small lengths.

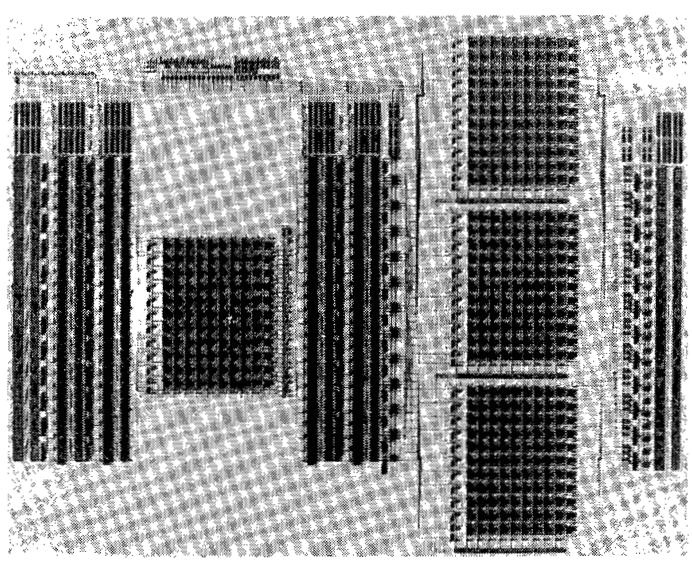
```
add(&d1, &d2, 0, &co, &tsum);
mask(&tsum,&tsum,1,12);

invodd(&d1, &d1);
invodd(&d2, &d2);
invodd(&tsum, &tsum);

reg_2phase(state5, &rd1, &d1, ph11[it], ph22[it]);
reg_2phase(state6, &rd2, &d2, ph11[it], ph22[it]);
reg_2phase(state7, &rtsum, &tsum, ph11[it], ph22[it]);

multiplex_4(&con1, &b0, &b1, &b2, &b3,
            counter_ph1[it], counter_ph22[it]);
mult2c(&rd1, &con1, &m1);
mask(&m1, &m1, 9, 20);

multiplex_4(&consum, &b4, &b5, &b6, &b7,
            counter_ph1[it], counter_ph22[it]);
mult2c(&rtsum, &consum, &msum);
mask(&msum, &msum, 9, 20);

multiplex_4(&con2, &b8, &b9, &b01, &b11,
            counter_ph1[it], counter_ph22[it]);
mult2c(&rd2, &con2, &m2);
mask(&m2, &m2, 9, 20);

reg_2phase(state8, &rm1, &m1, ph11[it], ph22[it]);
reg_2phase(state9, &rmsum, &msum, ph11[it], ph22[it]);
reg_2phase(state10, &rm2, &m2, ph11[it], ph22[it]);

invodd(&rm1, &rm1);
invodd(&rmsum, &rmsum);
invodd(&rm2, &rm2);

add(&rm1, &rmsum, 1, &co, &dadd1);
add(&rm2, &rmsum, 1, &co, &dadd2);
mask(&dadd1, &dadd1, 0, 11);
mask(&dadd2, &dadd2, 0, 11);
```

Fig. 16. Exact software image of the rotator hardware.

```
Procedure DCT 8:
const  PI = 3.141592654;
       c1d4 = cos(PI/4);
       c1d16 = cos(PI*1/16);
       cps1d16 = cos(PI*1/16) + sin(PI*1/16);
       smc1d16 = sin(PI*1/16) - cos(PI*1/16);
       c2d16 = cos(PI*2/16);
       cps2d16 = cos(PI*2/16) + sin(PI*2/16);
       smc2d16 = sin(PI*2/16) - cos(PI*2/16);
       c3d16 = cos(PI*3/16);
       cps3d16 = cos(PI*3/16) + sin(PI*3/16);
       smc3d16 = sin(PI*3/16) - cos(PI*3/16);

var x0, x1, x2, x3,  x4, x5, x6, x7, x8   : real;

(* 1   a1   *)  x8  :=  x0  +  x7   ;
(* 2   a2   *)  x7  :=  x0  -  x7   ;
(* 3   a3   *)  x0  :=  x4  +  x3   ;
(* 4   a4   *)  x3  :=  x4  -  x3   ;
(* 5   a5   *)  x4  :=  x8  +  x0   ;
(* 6   a6   *)  x0  :=  x8  -  x0   ;
(* 7   a7   *)  x8  :=  x2  +  x1   ;
(* 8   a8   *)  x1  :=  x2  -  x1   ;
(* 9   a9   *)  x2  :=  x5  -  x6   ;
(* 10  a10  *)  x6  :=  x5  +  x6   ;
(* 11  a11  *)  x5  :=  x8  +  x6   ;
(* 12  a12  *)  x6  :=  x8  -  x6   ;
(* 13  m1   *)  x6  :=  x6  *  c1d4;
(* 14  a13  *)  x8  :=  x1  +  x2   ;
(* 15  a14  *)  x2  :=  x1  -  x2   ;
(* 16  m2   *)  x2  :=  x2  *  c1d4;
(* 17  a15  *)  x1  :=  x4  +  x5   ;
(*18   a16  *)  x5  :=  x4  -  x5   ;
(* 19  a17  *)  x4  :=  x7  +  x6   ;
(* 20  a18  *)  x6  :=  x7  -  x6   ;
(* 21  a19  *)  x7  :=  x2  +  x3   ;
```



(a)



(b)

Fig. 17. DCT chip. (a) Cell layout of the chip. (b) Symbolic layout of the chip.

```
(* 22  a20  *)  x3  := x2  -  x3   ;
(* 23  m3   *)  x5  := x5  *  c1d4;
(* 24  a21  *)  x2  := x4  +  x7   ;
(* 25  m4   *)  x2  := x2  *  c1d16;
(* 26  m5   *)  x7  := x7  *  cps1d16;
(* 27  a22  *)  x7  := x2  -  x7   ;
(* 28  m6   *)  x4  := x4  *  smc1d16;
(* 29  a23  *)  x4  := x2  +  x4   ;
(* 30  a24  *)  x2  := x0  +  x8   ;
(* 31  m7   *)  x2  := x2  *  c2d16;
(* 32  m8   *)  x8  := x8  *  cps2d16;
(* 33  a25  *)  x8  := x2  -  x8   ;
(* 34  m9   *)  x0  := x0  *  smc2d16;
(* 35  a26  *)  x0  := x2  +  x0   ;
(* 36  a27  *)  x2  := x6  +  x3   ;
(* 37  m10  *)  x2  := x2  *  c3d16;
(* 38  m11  *)  x3  := x3  *  cps3d16;
(* 39  a28  *)  x3  := x2  -  x3   ;
(* 40  m12  *)  x6  := x6  *  smc3d16;
(* 41  a29  *)  x6  := x2  +  x6   ;
(* 42  m13  *)  x1  := x1  *  c1d4;

(* ordering:  1  7  8  3  5  6  0  4  *)
```

ACKNOWLEDGMENT

Without the help of a great many people, this project would never have been possible. We especially acknowledge the help of J. O'Neill (whose patience and help was infinite), B. Haskell (who made the project possible and carefully reread the manuscript), B. Ackland (for his encouragement on the MOVAL approach), and B. Ninke for his counseling during moments of frustration. Many thanks go to S. Fernandez for typing the manuscript. Very special thanks also to Swanica, R. Ligtenberg, and M. L. Renevey.

REFERENCES

[1] N. Ahmed, T. Natarajan, and K. R. Rao, "Discrete cosine transform," *IEEE Trans. Comput.*, vol. C-23, pp. 88–93, Jan. 1974.
[2] W. H. Chen and W. K. Pratt, "Scene adaptive coder," *IEEE Trans. Commun.*, vol. COM-32, no. 3, pp. 225–232.
[3] M. J. Narasimha and A. M. Peterson, "Design of a 24-channel transmultiplexer," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-27, no. 6, pp. 752–762.
[4] M. Vetterli and H. J. Nussbaumer, "Simple FFT and DCT algorithms with reduced number of operations," *Signal Processing*, vol. 6, no. 4, pp. 267–278.
[5] A. Jalai and K. R. Rao, "A high-speed FDCT processor for real-time processing of NTSC color TV signals," *IEEE Trans. Electromagn. Compat.*, vol. EMC-24, no. 2, pp. 270–286.
[6] G. Bertocci *et al.*, "An approach to the implementation of a discrete cosine transform," *IEEE Trans. Commun.*, vol. COM-30, no. 4, pp. 635–641.
[7] E. Arnould and J. P. Dugre, "Real-time discrete cosine transform: An original architecture," in *Proc. Intl. Conf. ASSP*, ICASSP-84, San Diego, CA, Mar. 1984.
[8] S. C. Knauer, "Distributed VLSI processors for picture coding," in *Proc. Intl. Conf. Commun.*, ICC-84, Amsterdam, The Netherlands, June 1984, pp. 718–723.
[9] A. Ligtenberg, M. Vetterli, and J. O'Neill, "MOVAL: A structured top-down design tool for mapping DSP algorithms into VLSI," to be published.
[10] N. Weste, "MULGA—An interactive symbolic layout system for the design of integrated circuits," *Bell Syst. Tech. J.*, vol. 60, no. 6, pp. 823–857.
[11] M. Vetterli, "FFT's of signal with symmetries and applications," submitted to MELECON-85, Madrid, Spain.
[12] W. H. Chen *et al.*, "A fast computational algorithm for the discrete cosine transform," *IEEE Trans. Commun.*, vol. COM-25, pp. 1004–1009, Sept. 1977.
[13] H. J. Nussbaumer, *Fast Fourier Transform and Convolution Algorithms*. Berlin, Germany: Springer Verlag, 1982.
[14] M. Vetterli, "Fast 2-D cosine transform algorithm," in *Proc. Intl. Conf. ASSP*, ICASSP-85, Tampa, FL, Mar. 1985.
[15] C. Mead and L. Conway, *Introduction to VLSI Systems*. Reading, MA: Addison-Wesley, 1980.
[16] C. R. Baugh and B. A. Wooley, "A two's complement parallel array multiplication algorithm," *IEEE Trans. Comput.*, vol. C-22, no. 12, pp. 1045–1097.
[17] K. Hwang, *Computer Arithmetic, Principles, Architecture, and Design*. New York: Wiley, 1979.
[18] Fred Guterl, "Testing," *IEEE Spectrum*, vol. 21, no. 9, pp. 40–46.
[19] J. A. Abraham, "Design for testability," in *Proc. 1983 Custom Integrated Circuits Conf.*, CICC-83, Rochester, NY, May 1983.
[20] H. Ahmed, J-H. Delosme, and M. Morf, "Highly concurrent computing structures for matrix arithmetic and signal processing," *Computer*, pp. 65–82, Jan. 1982.

Martin Vetterli was born in Switzerland. He received the B.A. degree at Neuchatel. In 1981, he received the Engineering Diploma from the Eidgenossische Technische Hoshschule de Zuriche (ETHZ), Zurich, Switzerland, and in 1982, the M.Sc. degree in electrical engineering from Stanford University, Stanford, CA. He then returned to Switzerland and received the Ph.D. degree from the Ecole Polytechnique Federale de Lausanne, for work on the efficient implementation of filter banks.

During 1984, he trained in VLSI conception at Bell Laboratories in Holmdel, NJ.

He is a member of the European Association of Signal Processing (EURASIP).

Adriaan Ligtenberg received the Engineer degree in electrical engineering in 1981 from Delft University of Technology, The Netherlands, and the Ph.D. degree in 1984 from the Ecole Polytechnique Federale de Lausanne, Switzerland.

Since 1984 he has been employed at AT&T Bell Labs, Holmdel, NJ. His research interests are the principles of algorithm integration.