

# Asynchronous Circuits for Low Power: A DCC Error Corrector

**CHIPWIDE ASYNCHRONOUS** operation has the potential for very low power consumption. Based on a programming and compilation approach, the design of asynchronous circuits may prove simpler and cheaper than the design of clocked circuits.

To check this theory, we chose to design an error corrector based on digital compact cassette (DCC) specifications<sup>1</sup> to show the following:

- A complex, industrially relevant function can be realized as a fully asynchronous circuit (comprising 155,000 transistors).
- Such functions can be programmed in the Tangram high-level VLSI programming language, which allows fully automatic compilation into asynchronous circuits.
- Asynchronous circuits may have a substantial power advantage over clocked circuits (80% less power dissipation for the error corrector).
- Systematic interfacing of asyn-

KEES VAN BERKEL

RONAN BURGESS

JOEP KESSELS

MARLY RONCKEN

FRITS SCHALIJ

Philips Research Laboratories

AD PEETERS

Eindhoven University of

Technology

*The authors describe a complete low-power digital compact cassette error corrector. Using Tangram, a high-level programming language, they designed two asynchronous circuits that correct errors on DCC specifications.*

chronous circuits according to existing (often synchronous) protocols can be realized (including dynamic RAM, FIFO, and I<sup>2</sup>S sound interfaces).

- Asynchronous circuits can be tested efficiently for most fabrication defects.

## Low power consumption

For portable products such as personal audio systems, mobile telephones, and games, a reduction in power consumption means longer battery life as well as lighter and smaller products. But low power consumption has advantages for nonportable products as well: cheaper and lighter power supplies, less expensive IC packages, and simpler power distribution.

The power consumption of a digital CMOS circuit is directly proportional to the amount of activity (the number of wires charged and discharged per unit of time). The energy required to charge and discharge a wire equals  $CV^2$ , where  $C$  denotes the wire's parasitic capacitance (and that of the transistor gates connected to the wire) and  $V$  represents the power supply voltage. (A short-circuit during switching and current leakage also dissipates some ener-

gy. These contributions amount to about 20% of the total power consumption, for carefully designed circuits.) Hence, we can reduce power consumption by implementing the following measures.<sup>2</sup>

- Lowering the supply voltage, which requires lower transistor threshold voltages and/or parallelism to compensate for the associated reduction in speed.
- Reducing (parasitic) capacitances, for example, by optimizing transistor dimensions and by keeping intensively used wires short.
- Reducing the number of gate-output transitions required for a given task. This factor adds to the previous two and is the link between "low-power" and asynchronous circuits.

In many synchronous ICs clock distribution dominates the power consumption. A high ratio of clock frequency to sample frequency (as is typical in digital audio) suggests wasted energy; many flip-flops receive new inputs during only a small fraction of the clock cycles (typically less than 10% for digital audio). Clock frequencies are often high because of high-throughput (low pin count) off-chip interfaces, the desire to share on-chip hardware resources, and the need to accommodate exceptions that may take a large fraction of the clock cycles per sample period.

CMOS asynchronous circuits, in contrast, only dissipate when and where active. That is, any subcircuit will return to standby mode (consuming leakage power only) whenever not in use. In addition, they favor distribution of control, leading to shallow control logic, as well as short status and control wires. Furthermore, the quasi-delay-insensitive circuits used are free of transients on wires: Each transition has its role. QDI circuits operate correctly independent of the delays in gates and wires, but assume negligible

skew in forking wires. In addition to a lower power consumption, we also expect favorable electromagnetic compatibility properties because the radiation spectra of asynchronous circuits contain less energy and show greater spread.

### VLSI programming

In general, asynchronous circuits are difficult to design and understand. They are prone to hazards, their reliability may be sensitive to variations in supply voltage and ambient temperature, and testing for fabrication faults is often impossible. Hence, we require a systematic approach for the design of complex asynchronous circuits, and approach the design of such circuits as a programming activity.

We have defined the Tangram programming language,<sup>3</sup> inspired by Hoare's Communicating Sequential Processes<sup>4</sup> and Occam.<sup>5</sup> Essentially, Tangram is a traditional imperative programming language similar to Pascal or C, with two additional features:

- statements and procedures may execute in parallel
- parallel statements (and procedures) can communicate along fixed channels

Tangram is attractive for VLSI programming because it allows the introduction of parallelism at any grain size, from two simple assignments to networks of processors. Simple primitives (such as sequential composition, parallel composition, selection, and repetition) allow concise expression of control without any concern for the duration of clock periods and the global synchronization implied by a common clock. Furthermore, Tangram allows sharing of hardware corresponding to functions and procedures. Together with a few simple operators for creating powerful data types, these properties make Tangram an expressive and general-

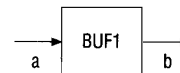


Figure 1. I/O structure of the one-place buffer BUF1.

purpose VLSI programming language. Others have also found CSP-based languages attractive for asynchronous circuit specification.<sup>6,8</sup>

For example, a simple Tangram procedure can describe a one-place buffer (see Figure 1).

$BUF1 = (a?W \& b!W).$

```
begin x: var W1
  forever do a?x ; b!x od
end
```

$W$  is an arbitrary type, such as a Boolean or an integer in the range 0 to 255. The opening pair of parentheses contains the declaration of the external ports of BUF1. Port  $a$  is an input of type  $W$ , and  $b$  is an output of the same type. The bracket pair **begin** ... **end** delineates the scope of variable  $x$ . The unbounded repetition **forever do** ... **od** comprises input statement  $a?x$  (accept an input along  $a$ , and store the incoming value in variable  $x$ ) followed by output statement  $b!x$  (send the value of  $x$  along  $b$ ).

The DCC error corrector uses buffers extensively to provide some slack between irregular production and/or consumption of data. Cascading one-place buffers results in economic buffers of limited capacity. For example, the Tangram procedure BUF2 describes a two-place buffer.

$BUF2 = (a?W \& b!W).$

```
begin m: chan W1
  BUF1(a,m) || BUF1(m,b)
end
```

Channel  $m$  connects the output of BUF1( $a,m$ ) to the input of BUF1( $m,b$ ).

Each communication along  $m$  synchronizes a send and a receive action.

Another practical variation on the one-place buffer is the ParSer parallel-to-serial converter. It repeatedly accepts a pair of messages of some type  $W$  along input channel  $a$  and outputs the messages sequentially along output channel  $b$ . The output rate is therefore twice the input rate. We express this behavior in Tangram as follows:

```
ParSer = (a?(W,W) & b!W).
begin x,y: var W
  forever do a?(x,y) ; b!x ; b!y
od
```

### Handshake circuits

The translation of Tangram programs into asynchronous circuits uses so-called handshake circuits as an intermediate architecture.<sup>9,10</sup> A handshake circuit is a network of components, connected by point-to-point channels. The only interaction among handshake components is by means of handshake signaling along these channels. There are no global signals. A handshake channel has an active (request) side and a passive (acknowledge) side. A

two-phase handshake protocol forms the basis of the handshake circuit formalism: The active side signals a request, and the passive side responds with an acknowledgment.

Typical examples of handshake components are the sequencer and the handshake latch. The sequencer controls the sequential execution of two Tangram statements. The handshake latch, corresponding to a Tangram variable, is a handshake component with a passive write port and a passive read port. The box below explains the behavior of these two components.

The handshake circuit of ParSer (shown in Figure 2) consists of 10 handshake components (depicted by circles), 12 handshake channels, and three external handshake ports (labeled  $\triangleright$ ,  $a$ , and  $b$ ). The handshake components correspond one-to-one to primitives in Tangram. The repeater (#) implements unbounded repetition (**forever do ... od**): An unbounded sequence of handshakes along  $c$  follows a request along  $\triangleright$ .

Accordingly, the repeater never acknowledges the handshake along  $\triangleright$ . Handshake latches implement Tangram

variables  $x$  and  $y$ . The so-called transferers  $T$  implement Tangram's input, output, and assignment. For instance, the transferer connected to port  $a$  responds to a request along  $d$  by actively fetching a message along  $a$  and passing this message along  $f$ . Similar to a traditional multiplexer, component "I" merges messages coming from  $x$  and  $y$ . Finally, component ">" splits messages incoming along  $f$  such that the two parts of the message can be passed to handshake latches  $x$  and  $y$ .

Observe that the structure of the handshake circuit reflects the syntactic structure of the Tangram program. The handshake components correspond to basic language constructs in Tangram. Hence, a small set of handshake component types (less than 30) suffices to fully implement Tangram. A few additional components implement interfaces to existing nonhandshaking ICs (as described in the Track-input buffer box). Handshake circuits are similar to macromodules<sup>11</sup> in that they construct VLSI circuits using weakly parameterized asynchronous building blocks, and that the wiring of these building blocks reflects the flow of control.

We can implement handshake circuits in various ways. Four-phase handshake signaling and double-rail encoding of data form the basis of our current libraries.<sup>7,12</sup> Four-phase signaling generally simplifies the implementation, because the wires involved return to their initial state after each handshake. Double-rail encoding (two wires per bit) leads to very robust designs. We discuss the CMOS realization of a handshake latch in the box on page 26.

### Testing

The prospects for testing QDI circuits for production defects are promising. We based our test approach on the stuck-at-output fault model. It takes advantage of the well-known property that a gate output stuck at 0 (or 1) manifests itself as a circuit deadlock when a hand-

### Examples of handshake components

A sequencer is a handshake component with one passive port  $a$  and two active ports  $b$  and  $c$  (Figure A1 and 2). Once activated along  $a$ , it will complete handshakes along  $b$  and  $c$  sequentially before completing the handshake along  $a$ . The active port marked with an asterisk is activated first. The state diagram depicts this behavior where the initial state has been marked with a fat dot. The subscripts  $r$  and  $a$  stand for request and acknowledge.

Handshake latch  $X$  returns an acknowledge signal after receiving a message along its write (input) channel  $wx$  (Figure B). Likewise, it outputs the contents of the latch after a request along its read channel  $rx$ . Read and write actions may not occur concurrently; they must be mutually exclusive.

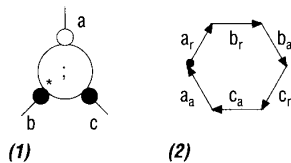


Figure A. A sequencer (1) and its state diagram (2).

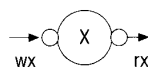


Figure B. Handshake latch.

shake involves transitions on that wire. Time-outs on the external output transitions can therefore detect defective circuits.

The problem of controllability remains: An acceptably short test run must exercise all wires. Complex circuits generally require additional test hardware to keep their test times acceptable. We still lack an overall solution for the controllability problem. Roncken and Saeijs present a solution for control-related circuits.<sup>13</sup> We extended this approach for one of the DCC ICs by adapting the scan principles from the synchronous test world.

### The Tangram toolbox

Translation of Tangram into hand-

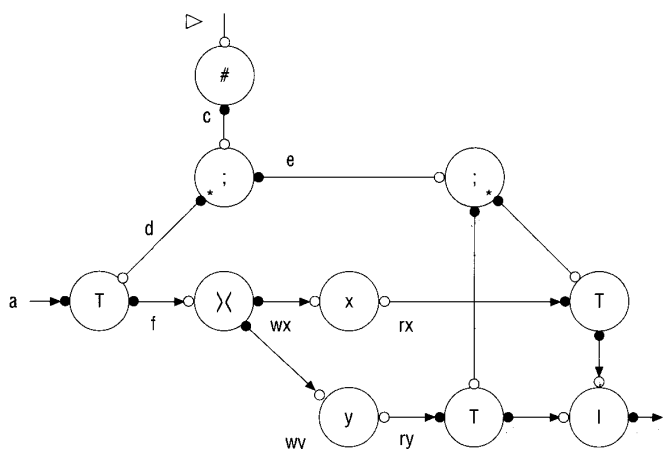


Figure 2. Handshake circuit for ParSer.

### Track-input buffer

Standard ICs, such as DRAMs and microprocessors, generally do not communicate by means of handshaking, and never use double-rail encoding for off-chip communication. So, we must address the issue of interfacing handshake circuits to existing ICs. Three hybrid handshake components

suffice to implement most of such interfaces:

- one to connect the state of a handshake latch to an output
- one to convert a single input transition into a full handshake
- one to sample the logic levels of a bundle of wires and convert those levels into a handshake message

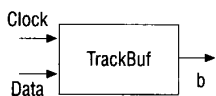


Figure C. TrackBuf with communication ports.

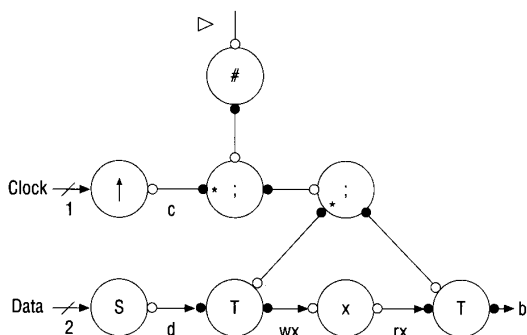


Figure D. Handshake circuit for TrackBuf.

Track-input buffer TrackBuf is a nice example that presents the latter two components. It repeatedly waits for a rising edge of clock, then samples the two data wires, stores their logical values in variable  $x$ , and sends the content of  $x$  along handshake channel  $b$  (Figure C). We express this in Tangram as

```
begin x: var <bool,bool> | forever do
  clock ↑ ; x:= data ; b!x od
end
```

$Clock$  denotes an input wire, and  $data$  represents a pair of input wires (their declarations have been omitted). Figure D shows the handshake circuit of TrackBuf.

Handshake component  $\uparrow$  acknowledges a passive handshake along  $c$  directly after a rising edge on wire  $Clock$ . Component  $S$  samples wire pair  $Data$  directly after a request along  $d$ , and outputs their logical values.

## Handshake latch

A typical handshake component is the handshake latch (see Figure E). The circuit allows two operations: a write handshake and a read handshake. A write handshake starts by

raising either wire  $w_0$  (write zero) or wire  $w_1$  (write one), and storing this value in a latch consisting of a pair of cross-coupled NOR gates. The latch then generates an acknowledgment by making  $w_a$  (write acknowledge) high, which indicates that the new val-

ue has arrived and is consistent with the contents of the latch (completion detection). The four-phase write handshake completes by making  $w_0$  (or  $w_1$ ) low again, followed by a downward transition of  $w_a$ .

Similarly, a read handshake starts by making  $r_r$  (read request) high. The handshake latch responds by making either  $r_0$  or  $r_1$  high, depending on the value stored in the latch. Then  $r_r$  becomes low, followed by a downward transition of  $r_0$  or  $r_1$ . After each write or read handshake all external wires ( $w_0$ ,  $w_1$ ,  $w_a$ ,  $r_r$ ,  $r_0$ , and  $r_1$ ) are low again.

Note that the handshake latch depends on a single latch; implementation does not require a master-slave operation. Hence, read and write handshakes must not overlap.

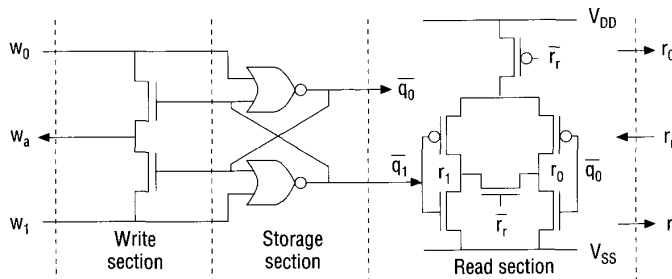


Figure E. CMOS circuit for a handshake latch.

shake circuits is syntax directed. For each production rule of Tangram's syntax, a similar translation rule exists.<sup>9,10</sup> This form of translation is highly transparent and provides the VLSI programmer information about the circuit cost,

speed, and power on the basis of the program text. It also means that the compiler processes a Tangram program directly into a specific handshake circuit. If unsatisfied with the cost or performance of the compiled circuit, the

designer must change the corresponding program. The transparency of the compilation scheme allows the VLSI programmer to develop a better program, in which powerful simulation and analysis tools provide assistance.

Figure 3 provides an overview of the Tangram tools. Boxes symbolize design representations, while arrows represent tools (an asterisk identifies commercially available tools). The first tool a VLSI programmer usually encounters is *B*, the translator from Tangram into behaviorally equivalent C code. Using simulator *C*, the programmer can verify the functional correctness of a program, including the input-output behavior and the absence of deadlocks. The simulator also provides coarse timing data to assist in the performance analysis.

Commercially available VHDL simulator *F* provides more accurate timing analysis and an estimation of the power consumption. This simulator uses the VHDL description generated by *E* from the handshake circuit and a library of

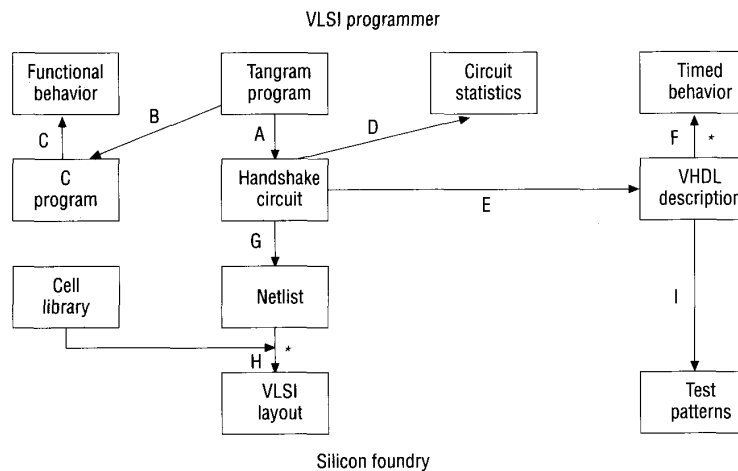


Figure 3. The Tangram toolbox.

models of handshake components. Standard cell layouts and layout statistics form the basis of the VHDL timing and energy consumption models, and simulation results can be analyzed with an interactive tool (not shown). Analysis tool *D* provides feedback on circuit and layout costs.

With feedback on functional behavior, area, timing, and power, VLSI programmers can iterate on the Tangram program. When satisfied, programmers can expand the handshake circuit into a netlist of standard cells, using *G*. Finally, using commercially available placement and routing tools *H*, the designer can convert this netlist into a custom VLSI layout. Tool *I* converts a Tangram program that describes a test into a sequence of test patterns, using the VHDL simulator.

### The DCC player

Three coding modules process the digital audio information in the DCC system (see Figure 4). The modules encode data streams from left to right in record mode. They decode data in the reverse order in play mode. In record mode, encoding compresses the audio information by a factor of four. Error encoding then adds parity bytes to protect the information against tape errors. Finally, channel encoding adds timing information by applying an 8-to-10 modulation. Observe that the sample rates in Figure 4 are all different. Also, they are almost two orders of magnitude smaller than the dominating clock rate of 6 MHz in the current synchronous realization. This suggests ample room for reducing power consumption.

We have designed an error corrector, consisting of three ICs, to operate in play mode only (see Figures 4 and 5). The detector accepts code words and outputs error information indicating possible corrections. The controller performs three simultaneous and loosely synchronized data transfers: from the channel decoder to memory, from memory

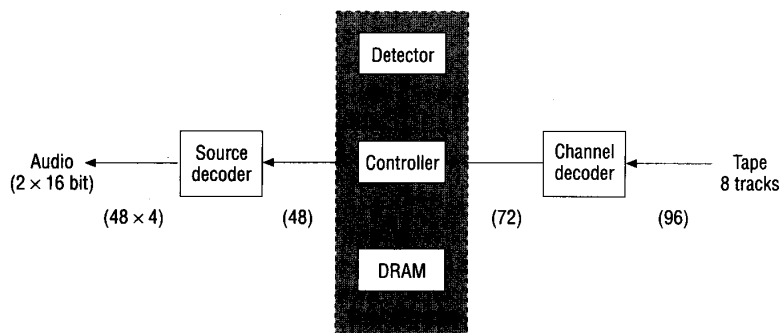


Figure 4. DCC codec in play mode (data rates in Kbytes/s).

to the detector for correction, and finally output of the corrected information to the source decoder. A commercially available 256K  $\times$  4-bit DRAM provides the memory. The rate of input data from the channel decoder depends on the motor speed, while a crystal oscillator keeps the output rate constant. Memory also serves as a buffer for motor speed control. When the buffer is nearly full, the motor slows down. Conversely, when the buffer is nearly empty, the motor speeds up.

We have programmed the detector and controller ICs in Tangram. Kessels et al.<sup>14</sup> provide a more detailed description of the design. For portable DCC

players, the power consumption of the chip set is a central concern; therefore we aimed at a design with minimal power usage. An experimental DCC player, used for research in digital magnetic recording, contains the error corrector. For this purpose, the controller collects and outputs extensive diagnostic information.

### Error-detector IC

A cross-interleaved Reed-Solomon code provides double protection against tape errors for the information on tape. Each 8-bit symbol is contained in two code words of different types—C1 and C2. C1 words contain 24 sym-

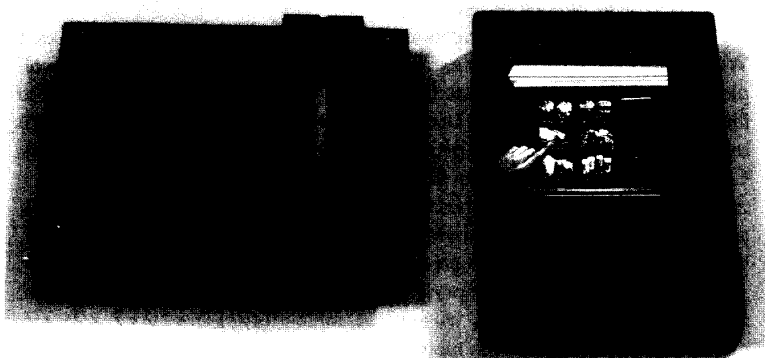
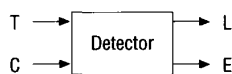


Figure 5. Photograph of the DCC error corrector, comprising a DRAM (top), the detector (right), and the controller (left). The DCC cassette beside the corrector provides an idea of the corrector's size.



**Figure 6.** Error detector with communications ports.

bols including four parity symbols; C2 words contain 32 symbols with six parity symbols. Due to the timing of a DCC player, the error detector must process approximately 3,000 C1 and 2,300 C2 words per second.

The detector processes 8-bit symbols that represent values from the Galois field  $GF(2^8)$ . The operations on these values differ from those on integers. For example, we obtain the sum of two  $GF(2^8)$  symbols by taking their bitwise exclusive OR. Tangram only supports two basic types, Booleans and integers, in a specified range. Therefore, programmers must introduce design-specific data types in the program. The Galois box describes four definitions of  $GF(2^8)$ .

Figure 6 shows the detector with its

communication ports. The detector has two input ports: *T* to receive the type of the code word and *C* to receive the symbols of a code word. The handling of a word starts by inputting the word type through *T*, after which *C* receives the codeword. The detector then computes the error information and outputs it through two output ports: *E* for the error values and *L* for the error status and location. Kessels et al.<sup>15</sup> describe a similar but much simpler design including a complete Tangram program.

The detector processes words in two phases. It computes so-called syndromes and then uses the result to compute error information. Syndrome computation occurs on-the-fly as the detector sequentially receives the symbols of the word. The syndrome computation is the same for all words, but the computation in the second phase depends on the correctness of the code word.

If the word is correct, the syndromes are zero and the error detection of the

word is complete. If the syndromes are not zero, the detector must perform an elaborate search of error values and locations. Hence, the circuit must be fast enough to deal with this worst-case situation.

Power simulations of the detector chip (Figure 7) show that a correct word requires only one third the energy of an incorrect one. Table 1 gives the measured timing and power characteristics of the fabricated IC at the normal operating voltage of 5V. It follows from these measurements and the required throughput that our chip dissipates 2.4 mW for correct words and 8.1 mW for worst-case incorrect words. As we expect that more than 98% of the words are correct, the average power consumption is approximately 2.5 mW.

Figure 8 shows the measured timing and power characteristics of the chip as a function of the supply voltage. We measure the time needed to process the mix of C1 and C2 code words relative to the value required for the DCC ap-

## Galois field arithmetic in Tangram

We can describe all the information about type  $GF(2^8)$  that the design of the detector requires in 60 lines of Tangram text. This description contains seven basic definitions introducing types, constants, and functions. We give four definitions as an example. The definitions are separated by the symbol `&`. A tuple of eight Booleans represents the new type *gfsym*. Constant *gfzero* represents the value zero in *gfsym*. Function *gfadd* has two parameters of type *gfsym* and yields their sum, which is also of type *gfsym*.

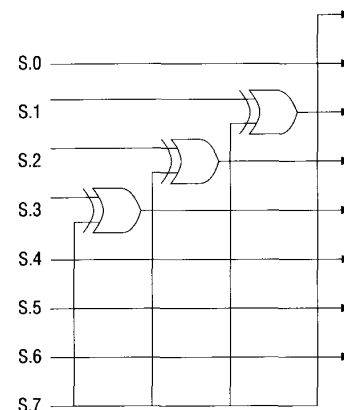
We obtain the result by comparing the corresponding elements in the operands, where element *i* in tuple *s* is denoted by *s.i*. Function *α* multiplies a *gfsym* value by the constant *α*. Here

*α* is a root of the irreducible polynomial  $X^8 + X^4 + X^3 + X^2 + 1$ . Figure F shows the circuit multiplying symbol *s* by *α*.

```
gfsym = type (bool, bool, bool,
             bool, bool, bool, bool, bool)
& gfzero = const (false, false, false,
                 false, false, false, false, false)
& gfadd = func (s, t : gfsym):
             gfsym.
             (s.0 ≠ t.0, s.1 ≠ t.1, s.2 ≠ t.2,
              s.3 ≠ t.3, s.4 ≠ t.4, s.5 ≠ t.5,
              s.6 ≠ t.6, s.7 ≠ t.7)
& alpha = func (s : gfsym) : gfsym.
             (s.7, s.0, s.1 ≠ s.7, s.2 ≠ s.7,
              s.3 ≠ s.7, s.4, s.5, s.6)
```

Note that `≠` for Booleans amounts

to their exclusive OR. Other basic functions include multiplication and inversion.



**Figure F.** Multiplication of *s* by *α*.

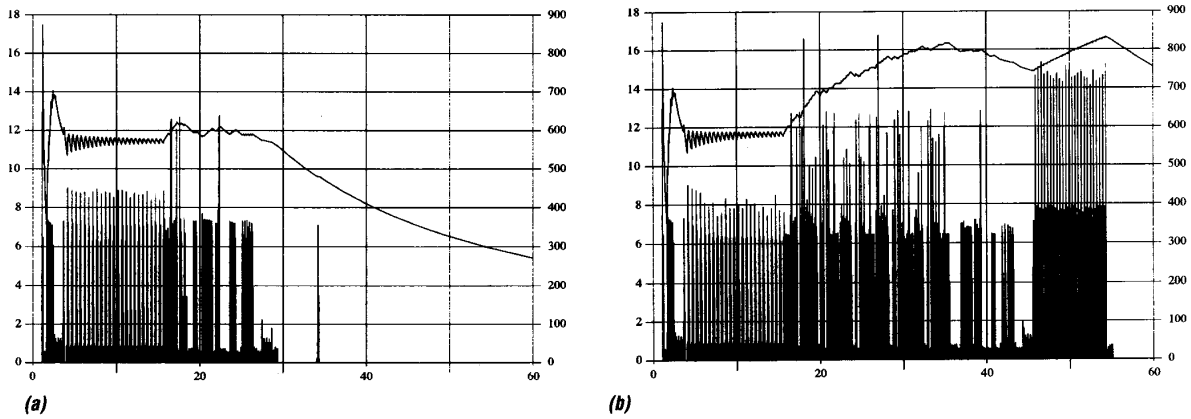


Figure 7. Simulated power consumption of a correct (a) and an incorrect (b) C1 word.

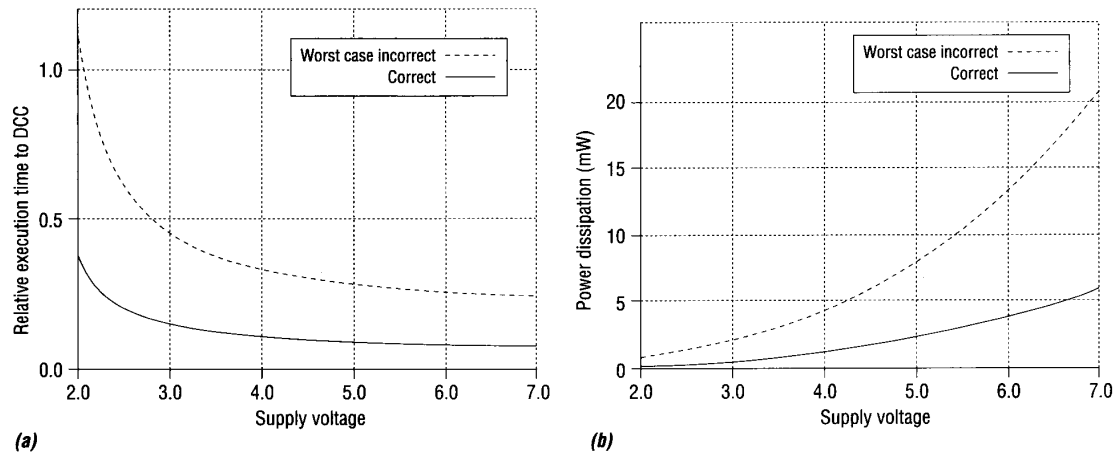


Figure 8. Measured timing (a) and power (b) characteristics of the detector IC.

Table 1. Measured execution time and energy for the two types of code words.

Word type	Rate (per s)	Time ( $\mu$ s)		Energy ( $\mu$ J)		Power (mW)	
		Correct	Worst case	Correct	Worst case	Correct	Worst case
C1	3,000	18	48	0.44	1.33	1.3	4.0
C2	2,300	18	64	0.49	1.79	1.1	4.1

plication. Therefore, the relative time should be less than one to be fast enough to deal with our worst-case scenario. The timing curves indicate that even at a supply voltage of 2V, the chip

is functionally correct and fast enough for correct words, but too slow if all words are incorrect. However, at 2.5V the IC also meets the speed requirements under worst-case conditions and

consumes only 0.3 mW. The IC is testable without additional hardware. The four code words that we selected give a stuck-at-fault coverage of 99.9% in less than 2-ms test time on an



HP82000 tester.

The detector program consists of 430 lines of Tangram text, including 60 lines on Galois field arithmetic. Kessels<sup>16</sup> provides a derivation of the program. We translated this program fully automatically (push-button) into a handshake circuit with 2,211 components. Basing the design on Euclid's algorithm led to a purely sequential program operating on two large variables of 64 bits. The two-wires-per-bit data encoding implies that each of the eight assignment channels to these variables requires 128 wires. This large amount of wiring leads to a circuit layout (see Figure 9) where the routing channels take more than two thirds of the area. The circuit contains 44,000 transistors, and its core size is 11.5 mm<sup>2</sup> in a 1.0-micron CMOS technology.

Compared to an existing (second-generation) clocked realization of a similar function, we estimate the asynchronous version to be about twice as large in area but five times more economic in power consumption. A modified Tangram program for the detector could further reduce the activity

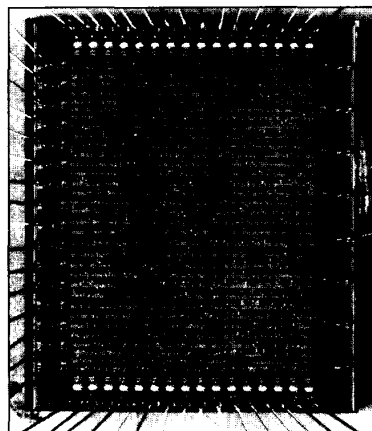


Figure 9. Photograph of a bonded version of the detector IC.

in the datapath. Switch-level simulations suggest a further reduction of 50% in power consumption with an area increase of only 5%.

### Controller IC

The controller is located between four neighboring modules. Its main function is to perform the following simultaneous and independent data

transfers between the modules:

- from the channel decoder to memory
- from memory to the detector (and vice versa)
- from memory to the source decoder

Since each of these transfers has its own timing characteristics, it is natural to have a distributed architecture in which a separate transferer performs each of these transfers. Figure 10 shows the global design of the controller in which we named each of the transferers after its communication partner. It also shows the Flagger module, which outputs diagnostic information gathered by other transferers.

The transferers operate in parallel, where loose synchronization prevents mutual overtaking. A memory controller arbitrates among the independent memory accesses. Each of the transferers contains a counter that cyclically generates memory addresses. One algorithmic cycle through the memory takes 680 ms.

Eight tape tracks store the audio information. Since the channel decoder handles each of the tracks separately, the controller receives eight track inputs, each with its own clock signal. We therefore designed the tape transferer as a distributed system, with eight autonomous track transferers and a common module. This system merges the eight track streams through arbitration. The arbitrated merging of data streams in both the tape transferer and the memory controller makes the IC nondeterministic regarding the order of the memory access.

The controller interfaces with four modules. Only communication with the detector goes through handshake channels. We based the interfaces with the channel decoder, the source decoder, and the DRAM on nonhandshake protocols that we programmed

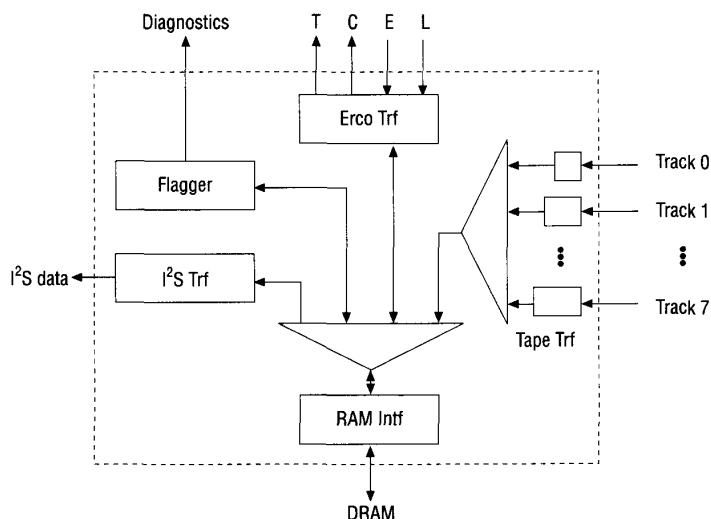


Figure 10. Block diagram of the controller IC.

in Tangram. (The earlier Track input buffer box gave an example.)

Both the nondeterministic behavior and the long algorithmic cycle time make designing a test strategy for the controller quite challenging. Since existing test equipment cannot deal with nondeterminism, we must make the test behavior deterministic. We have done this by designing the program to allow it to activate each transferer singly. By adding partial scan facilities to allow the setting of the address counters in the transferers, we have dealt with the long cycle time problem. The test strategy gives a 99.9% stuck-at fault coverage in 62-ms test time with 3% more area.

The controller program consists of 1,500 lines of Tangram text, resulting in a handshake circuit of 5,287 components. The distributed architecture leads to an efficient layout where the routing channels account for only 50% of the area. The circuit contains 111,000 transistors, and its core size is 18 mm<sup>2</sup> in a 0.8-micron CMOS technology. The chip consumes 8 mW at the normal operating voltage of 5V.

Because of functional differences, we cannot precisely compare the existing DCC IC. Estimates indicate a power reduction of 80%. The savings in power consumption comes from the distributed design, in which each module changes its state space at its own rate. The designer of a clocked revision of the same function must combine the state spaces of all the submodules into a large state space, which will make transitions at a rate high enough to deal with all events.

**WE HAVE DESIGNED** and fabricated two asynchronous ICs performing error correction on DCC specifications. Table 2 summarizes their main characteristics. A detailed account of a previous IC compiled from Tangram appears in van Berkel et al.<sup>17</sup>


**Table 2.** Characteristics of error corrector ICs.

Item	Detector	Controller
Design effort (person-years)	0.4	0.8
Tangram lines	430	1,500
Handshake components	2,211	5,287
Standard cells	3,385	11,474
Transistors	44,000	111,000
Technology (micron)	1.0	0.8
Core area (mm <sup>2</sup> )	11.5	18.0
Power @ 5V (mW)	2.4	8.0

At the gate and latch level, asynchronous circuits are more difficult to design than clocked circuits. Therefore, we believe the design of complex asynchronous circuits requires a high-level programming language such as Tangram and a transparent silicon compiler. Moreover, tools providing fast feedback on all relevant design aspects, such as functional behavior, timing, power, and area, are a prerequisite for obtaining correct and economic solutions.

Both asynchronous ICs we designed and fabricated nicely demonstrate why well-designed asynchronous CMOS circuits save power: They only dissipate *when* and *where* necessary. The timing holds for functions such as the detector, where the worst-case work load exceeds the average-case load. Where dissipation occurs becomes important for functions such as the controller, which allows a distributed architecture with submodules running at different rates. For example, the (second-generation) clocked realization operates at a clock frequency of 6 MHz. In both ICs, the various data rates are very low relative to this frequency. By accessing registers at their data rates rather than at a high clock rate, substantial savings occur.

In comparison with clocked circuits the 70-100% area overhead occurs largely as a result of the two-wires-per-bit data encoding. This is essentially the price for QDI operation.

In the ESPRIT project Exact, we are investigating less expensive alternatives to implement handshake circuits based on one-wire-per-bit encoding. These alternatives promise additional (and substantial) power savings, but require delay matching and, therefore, introduce new verification and test problems. 

#### Acknowledgments

We thank Peter Arts and John Sherry of Philips Consumer Electronics for introducing us to the specification of the DCC error corrector and its synchronous realization, Ad Denissen and Milton Ribeiro for providing the specification of the asynchronous version, Ton Kalker for explaining the mathematics behind error correction, Jos van Beers for assistance in testing the ICs, and Cees Niessen and Martin Rem for advice. We also thank Erik-Jan Marinissen and Jef van Meerbergen for reviewing our manuscript.

The European Commission funded part of this work under ESPRIT Contract 6143 (Exact).

## References

1. G.C.P. Lokhoff, "DCC—Digital Compact Cassette," *IEEE Trans. Consumer Electronics*, Vol. 37, No. 3, Aug. 1991, pp. 702-706.
2. A.P. Chandrakasan, S. Sheng, and R.W. Brodersen, "Low-Power CMOS Digital Design," *IEEE J. Solid-State Circuits*, Vol. 27, No. 4, 1992, pp. 473-483.
3. F. Schalij, "Tangram Manual," Tech. Report LR 008/93, Philips Research Laboratories, Eindhoven, The Netherlands, 1993.
4. C.A.R. Hoare, "Communicating Sequential Processes," *Comm. ACM*, Vol. 21, No. 8, 1978, pp. 666-677.
5. *Occam Programming Manual*, Inmos Limited, ed., *Series in Computer Science*, Prentice-Hall Int'l, 1984.
6. A.J. Martin, "Syntax-Directed Translation of Concurrent Programs into Self-Timed Circuits," *Proc. Sixth MIT Conf. Adv. Research in VLSI*, MIT Press, Cambridge, Mass., 1990, pp. 35-50.
7. A.J. Martin, "Programming in VLSI: from Communicating Processes to Delay-Insensitive Circuits," *UT Year of Programming: Institute on Concurrent Programming*, C.A.R. Hoare, ed., Addison-Wesley, Reading, Mass., 1989, pp. 1-64.
8. E. Brunvand and R. Sproull, "Translating Concurrent Programs into Delay-Insensitive Circuits," *Proc. IEEE Int'l Conf. Computer-Aided Design*, IEEE Computer Society Press, Los Alamitos, Calif., 1989, pp. 262-265.
9. K. van Berkel et al., "The VLSI-Programming Language Tangram and Its Translation into Handshake Circuits," *Proc. European Design Automation Conf.*, CS Press, 1991, pp. 384-389.
10. K. van Berkel, *Handshake Circuits: An Asynchronous Architecture for VLSI Programming*, Int'l Series on Parallel Computation 5, Cambridge University Press, Cambridge, England, 1993.
11. S.M. Omstein, M.J. Stucki, and W.A. Clark, "A Functional Description of Macromodules," *Proc. Sprint Joint Computer Conf.*, AFIPS, 1967, pp. 337-355.
12. C.L. Seitz, "System Timing," *Introduction to VLSI Systems*, C.A. Mead and L.A. Conway, eds., Addison-Wesley, 1980.
13. M. Roncken and R. Saeijs, "Linear Test Times for Delay-Insensitive Circuits: A Compilation Strategy," *Proc. IFIP WG 10.5 Working Conf. Asynchronous Design Methodologies*, 1993, pp. 13-27.
14. J. Kessels et al., "VLSI Programming of a Low-Power Asynchronous Error Corrector for the DCC Player," Tech. Report TN 023/94, Philips Research Laboratories, 1994.
15. J. Kessels et al., "An Error Decoder for the Compact Disc Player as an Example of VLSI Programming," *Proc. European Design Automation Conf.*, 1992, pp. 69-74.
16. J. Kessels, "Derivation of a Low-Power Reed-Solomon Decoder for the DCC Player," Tech. Report TN 034/94, Philips Research Laboratories, 1994.
17. K. van Berkel et al., "A Fully Asynchronous Low-Power Error Corrector for the DCC Player," *Proc. IEEE Int'l Solid-State Circuits Conf.*, 1994, p. TA5.4.

Address questions and comments on this article to Kees Van Berkel at Philips Research Laboratories, Building WAY4 099, Prof. Holstlaan 4, 5656 AA Eindhoven, The Netherlands; berkel@prl.philips.nl.



The asynchronous design team includes (from left): Joep Kessels, Kees van Berkel, Ronan Burgess, Ad Peeters, Frits Schalij, and Marly Roncken.

The authors cooperate on the VLSI Programming and Silicon Compilation project at the IC Design Center of Philips Research Laboratories, Eindhoven, The Netherlands. **Joep Kessels'** main field of interest is VLSI programming and possible applications of Tangram. He designed the Tangram programs for both the error detector and controller. **Kees van Berkel** invented the concept of handshake circuits and handled their VLSI implementation. He also coordinates the project. **Ronan Burgess** was responsible for the chip finishing and the

handshake circuit simulator. **Ad Peeters** is working toward his PhD on protocols in asynchronous designs at the Eindhoven University of Technology. He added the external interfaces to the controller. **Frits Schalij** works on Tangram and its compilation into handshake circuits. He built the high-level Tangram simulator and the compiler into handshake circuits. **Marly Roncken** researched asynchronous circuit testing. She developed the test approach for both ICs and handled the tests and measurements.