High-Level Synthesis of Digital Comparators

WALTER DOSCH
Institute for Software Technology and Programming Languages
Medical University of Lübeck
Seelandstraße 1a, D–23569 Lübeck
GERMANY
dosch@isp.mu-luebeck.de http://www.isp.mu-luebeck.de

Abstract: We present a unifying high-level synthesis of tree-structured and iterative networks for digital comparators. Based on the theory of list homomorphisms, we develop a standard implementation for a tree-structured module processing the input digits in parallel. The design is systematically specialized to iterative networks processing the input sequentially from the least resp. from the highest significant positions. Throughout the paper, we explicate functional methods for the design and the description of combinational circuits.

Key-Words: High-level synthesis, combinational circuit, comparator module, tree network, iterative network, functional hardware description

1 Introduction

The high-level synthesis of digital modules aims at formally deriving a description of the implementation from a behavioural specification. The formal design supports the “correct by construction” paradigm — in contrast to an “a posteriori” verification [3].

This paper presents the functional description and synthesis of tree-structured and iterative networks for digital comparators. Within a structured design methodology for modern VLSI technology [11] we abstract from timing issues and concentrate on algorithmic design principles. The synthesis explicates different comparator modules in terms of the design decisions leading to the implementation. The approach separates the structural design of the network from the realization of the basic modules. Important design steps consist in changing the representation from natural numbers to lists of digits and in introducing suitable decomposition strategies on the input list.

As our major contribution, we present a unifying synthesis for tree-structured and iterative networks of digital comparators. The common background is provided by the theory of list homomorphisms. When imposing a binary decomposition strategy on the input list, we obtain a tree-structured combinational network processing the input digits in parallel. The elementary circuitry implementing the associative combining operation is cascaded with logarithmic depth. The general design can systematically be specialized to linear digit recurrence algorithms. They describe iterative networks processing the input digits sequentially from the least resp. from the highest significant positions.

The combinational circuits are uniformly described in a functional style [12] both on the specification and the implementation level [2]; this offers a coherent framework for algebraic reasoning. On the specification level, the functional style abstracts from the module’s representation to the input/output function. On the implementation level, the functional approach provides a structural description, compare [9] for a topological approach and [10] for a relational approach.

The paper is organized as follows. Section 2 surveys the foundations of modeling combinational circuits with list functions. In Section 3 we introduce parallel homomorphisms for tree networks. In Section 4 we specialize them to linear homomorphisms and iterative networks. In Section 5 we specify the behaviour of a comparator module and refine the interface from natural numbers to lists of digits. Section 6 presents a unifying synthesis of different comparator modules in three design steps. The conclusion surveys the approach and outlines future research.
2 Combinational Networks and List Functions

This section surveys the foundations of modeling combinational circuits as list functions. We introduce list homomorphisms for describing tree-structured networks. Iterative networks arise from specializing the parallel recursion to a linear recursion.

2.1 Combinational Modules as List Functions

The behaviour of a combinational module at one clock cycle is described by a function \( f : \langle x_k, \ldots, x_0 \rangle \mapsto \langle y_l, \ldots, y_0 \rangle \) mapping \( k+1 \) input digits to \( l+1 \) output digits, compare Fig. 1. We achieve

\[
\begin{array}{c}
\begin{array}{c}
\vdots \\
\vdots \\
f \\
\vdots \\
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
x_k \ldots x_1 x_0 \\
y_l \ldots y_1 y_0
\end{array}
\end{array}
\]

Figure 1: Combinational module as a list function

A uniform treatment of modules with different arities by collecting the inputs \( x_i \) into an input list \( \langle x_k, \ldots, x_0 \rangle \) and the outputs \( y_j \) into an output list \( \langle y_l, \ldots, y_0 \rangle \). Lists will be denoted by capital letters, their items by small letters.

2.2 Composition of Modules

The sequential composition of two combinational modules corresponds to the function composition \((f \circ g)(X) \equiv f(g(X))\) of their behaviours, compare Fig. 2.

The parallel composition of two modules is represented by the function product \((f \times g)(X, Y) \equiv f(X) \& g(Y)\), compare Fig. 2. For convenience, we also use function tupling \((f \circ g)(X) \equiv f(X) \& g(X)\) that copies the input list to both modules.

\[
\begin{array}{c}
\begin{array}{c}
\vdots \\
\vdots \\
g \\
\vdots \\
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
\vdots \\
\vdots \\
f \\
\vdots \\
\end{array}
\end{array}
\]

Figure 2: Serial composition (left) and parallel composition (right) of two modules

A combinational network results from the parallel and serial composition of combinational modules.

3 Parallel Lists and Tree Networks

In this section we describe tree-structured networks by homomorphisms over parallel lists.

3.1 Parallel Lists

For a nonempty set \( A \) of digits, the equation

\[
A^+ \equiv A \cup A^+ \times A^+
\]

defines the set \( A^+ \) of parallel lists over \( A \) assuming that \( \times \) denotes the associative cartesian product. Parallel lists are constructed by forming singleton lists \( \langle \rangle : A \to A^+ \) and by concatenating \& \( : A^+ \times A^+ \to A^+ \) two sublists. Concatenation is associative, hence the decomposition of a parallel list into sublists is not unique.

3.2 Parallel Homomorphisms

Parallel homomorphisms are functions that promote through concatenation. Formally, a function \( h : A^+ \to B \) is called a parallel homomorphism, if

\[
h(\langle a \rangle) \equiv f(a) \quad (2)
\]

\[
h(A \& B) \equiv h(A) \oplus h(B) \quad (3)
\]

holds for an atomic function \( f : A \to B \) and a parallel combining operation \( \oplus : B^2 \to B \). We write \( h \equiv \text{hom}(f, \oplus) \) for the parallel homomorphism uniquely determined by \( f \) and \( \oplus \).

3.3 Special Parallel Homomorphisms

Combinational networks are based on two fundamental homomorphisms.

The functional map \( : [A \to B] \to [A^+ \to B^+] \) applies a function \( f : A \to B \) to all digits of an input lists:

\[
\text{map}(f) \equiv \text{hom}(\langle \rangle \circ f, \&) \quad (4)
\]

In the standard layout, the combinational module implementing the atomic function \( f \) is replicated for each input digit, compare Fig. 3.
The functional \( \text{red} : [\mathcal{B}^2 \to \mathcal{B}] \to [\mathcal{B}^+ \to \mathcal{B}] \) reduces an input list by combining its digits under an associative operation:

\[
\text{red}(\oplus) \equiv \text{hom}(\text{id}_\mathcal{B}, \oplus)
\]

(5)

The standard layout is a binary tree with logarithmic depth that cascades the combinational module for the combining operation, compare Fig. 4.

![Combinational network associated with the functional red](image)

Figure 4: Combinational network associated with the functional \( \text{red} \)

The recursive description of networks \cite{13} leads to a regular layout where the primitive combinational modules are replicated in a systematic way.

### 3.4 Networks Associated With Parallel Homomorphisms

The “homomorphism lemma” \cite{1} lays the basis for describing tree networks by parallel list homomorphisms.

Every parallel homomorphism can be “diffused” into the sequential composition of two modules. First the atomic function is applied in parallel to all digits of the input list, then the intermediate results are reduced under the parallel combining operation:

\[
\text{hom}(f, \oplus) \equiv \text{red}(\oplus) \circ \text{map}(f)
\]

(6)

The standard layout for parallel homomorphisms results from composing the standard layouts for the functionals \( \text{map} \) and \( \text{red} \), compare Fig. 5.

![Combinational module associated with a parallel homomorphism \( \text{hom}(f, \oplus) \)](image)

Figure 5: Combinational module associated with a parallel homomorphism \( \text{hom}(f, \oplus) \)

The high-level design of tree networks concentrates on finding suitable algebraic structures where the function to be implemented can be represented by a parallel homomorphism. The approach separates the structural design of the network from the realization of the primitive modules for the atomic function and the parallel combining operation.

### 4 Sequential Lists and Iterative Networks

Sequential lists and parallel lists denote the same set of objects. The different names reflect the way how the lists are constructed resp. decomposed. Left and right homomorphisms over sequential lists describe iterative networks.

#### 4.1 Left-Sequential Lists

For a nonempty set \( \mathcal{A} \), the equation

\[
\mathcal{A}^+ \equiv \mathcal{A} \cup \mathcal{A} \times \mathcal{A}^+
\]

(7)

defines the set of all left-sequential lists, also called “cons-lists”. They are generated by forming singleton lists \( \langle \cdot \rangle : \mathcal{A} \to \mathcal{A}^+ \) and by repeatedly prefixing \( \langle \cdot \rangle : \mathcal{A} \times \mathcal{A}^+ \to \mathcal{A}^+ \) an element to the front of the list.

#### 4.2 Left Homomorphisms

A left homomorphisms traverses a left-sequential list following its structure. Formally, a function \( h : \mathcal{A}^+ \to \mathcal{B} \) is called a left homomorphism, if

\[
h(\langle x \rangle) \equiv f(x)
\]

(8)

\[
h(x \triangleleft X) \equiv x \odot h(X)
\]

(9)

holds for a start function \( f : \mathcal{A} \to \mathcal{B} \) and a left combining operation \( \odot : \mathcal{A} \times \mathcal{B} \to \mathcal{B} \). Left homomor-
phisms show a linear recursion and process the list element by element.

Parallel homomorphisms can be specialized to left homomorphisms by defining the left combining operation as
\[ x \odot r = f(x) \oplus r \quad (10) \]

4.3 Left-Iterative Networks

Left homomorphisms describe left-iterative networks which replicate the module implementing the left combining operation in a regular chain. The input list is processed sequentially starting with the least significant digit, compare Fig. 6.

4.4 Right-Sequential Lists

Symmetrically, parallel lists can be specialized to right-sequential lists, also called “snoe-lists”:
\[ A^+ \equiv A^+ \times A \cup A \quad (11) \]

Right-sequential lists are constructed by forming singleton lists \([\cdot] : A \to A^+\) and by repeatedly attaching \(\triangleright : A^+ \times A \to A^+\) an element to rear of the list. The associated right homomorphism
\[ h([x]) = f(x) \quad (12) \]
\[ h(X \triangleright x) = h(X) \odot x \quad (13) \]

uses a right combining operation \(\odot : B \times A \to B\). Right homomorphisms show a linear recursion and process the list element by element.

Parallel homomorphisms can be specialized to right homomorphisms by defining the right combining operation as
\[ r \odot x = r \oplus f(x) \quad (14) \]

4.5 Right-Iterative Network

Right homomorphisms describe right-iterative networks that process the input digits sequentially starting with the highest significant position, compare Fig. 7.

Altogether, list homomorphisms provide a unifying algebraic framework for tree-structured and iterative combinational modules. Different types of iterative networks can systematically be designed by specializing the tree network. When the combining operation of the parallel list recursion is not commutative, the left and the right combining operations used for the iterative networks will be different.

5 Specification

In this section, we define the behaviour of a comparator module for natural numbers and transform it into a design specification based on lists of digits.

5.1 High-Level Specification

A comparator module indicates the magnitude relationship between two natural numbers. The function comparator : \(\mathbb{N} \times \mathbb{N} \to \{G, E, S\}\) associates with each two natural numbers the relationship \(S\) (smaller), \(E\) (equal) or \(G\) (greater):
\[ \text{comparator}(a, b) \quad \begin{cases} 
  S & \text{if } a < b \\
  E & \text{if } a = b \\
  G & \text{if } a > b 
\end{cases} \quad (15) \]

5.2 Design Specification

In the first development step, the high-level specification is transformed into a design specification based on lists of digits. We relate the abstract level — natural numbers — and the implementation level — lists of digits — by representation and abstraction functions [2].

5.2.1 Representation of Natural Numbers

As the first design decision, we represent natural numbers in the positional number system as nonempty lists of digits with the least significant digit standing on the right-hand side.

For a natural number \(p \geq 2\) as basis, the set \(D \equiv \{0, \ldots, p - 1\}\) denotes the set of digits used in \(p\)-adic
arithmetic. The abstraction function decode : \( \mathbb{D}^+ \to \mathbb{N} \) computes the natural number associated with a list of digits:

\[
\text{decode}(\langle x \rangle) \equiv x \\
\text{decode}(X \& Y) \equiv \text{decode}(X) \cdot p^{\text{length}(Y)} + \text{decode}(Y)
\]

The representation of natural numbers as lists of digits is unique up to leading zeros.

### 5.2.2 Representation of the Input

As the second design decision, the two input lists \( \langle x_k, \ldots, x_0 \rangle \) and \( \langle y_k, \ldots, y_0 \rangle \) of equal length are joined into a single input list \( \langle (x_k, y_k), \ldots, (x_0, y_0) \rangle \) of pairs of digits.

The functions \( \text{first} : (\mathbb{D}^\mathbb{D})^+ \to \mathbb{D}^+ \) and \( \text{second} : (\mathbb{D}^\mathbb{D})^+ \to \mathbb{D}^+ \) extract the representation of the first resp. the second operand:

\[
\text{first} \equiv \text{hom}(\langle \cdot \rangle \circ \pi_1, \&) \\
\text{second} \equiv \text{hom}(\langle \cdot \rangle \circ \pi_2, \&)
\]

Here \( \pi_i \) selects the \( i \)-th component from a tuple.

### 5.2.3 Representation of the Output

As the third design decision, we represent the result domain using a 1-out-of-3 coding. The representation function \( \text{code} : \{G, E, S\} \to \mathbb{B}^3 \) reads:

\[
\text{code}(S) \equiv (1, 0, 0) \\
\text{code}(E) \equiv (0, 1, 0) \\
\text{code}(G) \equiv (0, 0, 1)
\]

An easy derivation, compare Appendix A, confirms the characteristic properties:

\[
\pi_1 \circ \text{code} \circ \text{comparator} \equiv < \\
\pi_2 \circ \text{code} \circ \text{comparator} \equiv = \\
\pi_3 \circ \text{code} \circ \text{comparator} \equiv >
\]

### 5.2.4 Interface and Behaviour

With these design decisions, we refine the interface of the comparator module, compare Fig. 8. Its behaviour \( \text{comp} : (\mathbb{D}^\mathbb{D})^+ \to \mathbb{B}^3 \) is defined as the composition of three subtasks:

\[
\text{comp} \equiv \text{code} \circ \text{comparator} \circ ((\text{decode} \circ \text{first}) \circ (\text{decode} \circ \text{second}))
\]

First we decode the two operands from the input list, then we compare the resulting natural numbers, and finally we encode the ternary result of the comparison. This is a general method how to transfer a function from lists of digits to natural numbers.

\[
\langle x_k, y_k \rangle \quad \langle x_0, y_0 \rangle
\]

Figure 8: Design specification of a comparator module

For convenience, we name the three components of the result by individual functions \( \text{lt}, \text{eq}, \text{gt} : (\mathbb{D}^\mathbb{D})^+ \to \mathbb{B} : \)

\[
\text{comp} \equiv \text{lt} \circ \text{eq} \circ \text{gt}
\]

Using the characteristic properties (23)–(25), we reach an explicit definition of the three comparison functions, compare Appendix B:

\[
\text{lt} \equiv < \circ (\text{decode} \circ \text{first}) \circ (\text{decode} \circ \text{second}) \\
\text{eq} \equiv = \circ (\text{decode} \circ \text{first}) \circ (\text{decode} \circ \text{second}) \\
\text{gt} \equiv > \circ (\text{decode} \circ \text{first}) \circ (\text{decode} \circ \text{second})
\]

### 6 Synthesis

The synthesis of the comparator module proceeds in three major steps. In the first step, we derive
digit recurrence algorithms for the three comparison functions. Then we fuse the individual comparison functions into a list homomorphism. Finally, we implement the atomic function and the combining operation of the parallel homomorphism by some binary logic. The derivation completely separates the structural properties of the network from Boolean algebra.

### 6.1 List Recurrence Algorithms

For each of the three comparison functions, we derive a digit recurrence algorithm by eliminating the auxiliary functions in their definitions (28)–(30). To this end, we use fold and unfold transformations along with algebraic simplifications, compare Appendix C:

\[
\begin{align*}
lt((x, y)) & \equiv x < y \\
lt(P \& Q) & \equiv \lt(P) \lor \eq(P) \land \lt(Q) \\
\eq((x, y)) & \equiv x = y \\
\eq(P \& Q) & \equiv \eq(P) \land \eq(Q) \\
\gt((x, y)) & \equiv x > y \\
\gt(P \& Q) & \equiv \gt(P) \lor \eq(P) \land \gt(Q)
\end{align*}
\]

For saving brackets we assume that conjunction binds tighter than disjunction. The equality function forms a genuine list homomorphism. The less-than and the greater-than functions, however, do not form homomorphisms. The result of a composite list cannot be computed from the results of the sublists alone; rather it also depends on the equality of the left sublist.

### 6.2 List Homomorphic Solution

We fuse the hierarchical system \(\{lt, eq, gt\}\) of the three individual comparison functions into a single three-valued comparator function \(\text{comp} : (\mathbb{D}^2)^+ \rightarrow \mathbb{B}^3\), compare Appendix D. The resulting comparator algorithm

\[
\text{comp}((x, y)) \equiv x \odot y
\]

\[
\text{comp}(X \& Y) \equiv \text{comp}(X) \oplus \text{comp}(Y)
\]

forms a list homomorphism. The atomic function \(\odot : \mathbb{D}^2 \rightarrow \mathbb{B}^3\) with

\[
\odot \equiv < \circ \circ >
\]

compari es each pair of input digits. The combining operation \(\oplus : \mathbb{B}^3 \rightarrow \mathbb{B}^3\) with

\[
(l, e, g) \oplus (l', e', g') \equiv (l \lor e \land l', e \land e', g \lor e \land g')
\]

combines the partial results from the left and right sublists to form the result of the overall input list. Note that the combining operation does not depend on the basis of the positional number system. By construction, it is associative, but due to the dominance of higher significant positions it is not commutative.

### 6.3 Specialization to Binary Digits

The high-level design was parameterized with the basis of the positional number system.

Now we instantiate the basis to \(p = 2\) and specialize the atomic function to binary digits using some Boolean logic:

\[
x \odot y \equiv (\neg x \land y, x \land y \lor \neg x \land \neg y, x \land \neg y)
\]

The basic building blocks for tree-structured comparator networks are shown in Fig. 9. They implement the two parameters of the parallel homomorphisms, viz. the atomic function and the parallel combining operation.

### 6.4 Combinational Networks

So far we established the comparator function as a homomorphism on the input list:

\[
\text{comp} \equiv \text{red}(\oplus) \circ \text{map}(\odot)
\]

We also implemented its two parameters by some basic circuitry. Now we can apply all the results from Sections 3 and 4 to generate standard layouts for digital comparators.

Following Fig. 5, a tree-structured comparator hierarchically cascades the combining operation with logarithmic depth. It applies the atomic function in parallel to all positions of the input list.
Fig. 6 shows the standard layout for a left-iterative network processing the input digits sequentially from the least significant position. The left combining operation \( \odot: D^2 \times B^3 \rightarrow B^3 \) reads:

\[
(x, y) \odot (l', e', g') = \begin{cases} 
(x < y \lor x = y \land l'), & \\
x = y \land e', & \\
x > y \lor x = y \land g') & 
\end{cases}
\]

Symmetrically, Fig. 7 presents the standard layout for a right-iterative network processing the input digits sequentially from the most significant position. The right combining operation \( \odot: B^3 \times D^2 \rightarrow B^3 \) reads:

\[
(l, e, g) \odot (x, y) = \begin{cases} 
(l \lor e \land x < y), & \\
e \land x = y, & \\
g \lor e \land x > y) & 
\end{cases}
\]

Also other types of comparator networks [6] can systematically be derived from the list homomorphic properties.

7 Conclusion

The formal synthesis of digital circuits [4, 5] reveals the algorithmic principles underlying hardware realizations. The derivation exhibits the algebraic properties necessary to establish the correctness of the circuit. The synthesis consists of many simple mechanical steps guided by recursion patterns. The synthesis is supported by algorithmic reasoning taking efficiency considerations into account.

The algorithms are written using higher-order concepts [8] to be independent of the technological progress with respect to the input size. They are parameterized with the basis of the positional number system to isolate the particular effects of binary coding.

The framework is well suited for the specification, design and analysis of synchronous digital circuits. The initial transformation changing the representation from natural numbers to lists of digits involves the factorization theorem [7]. The subsequent transformations outline a general proceeding how to parallelize digit recurrence algorithms. Future research attempts to formalize other general design principles for combinational circuits, for example multi-level module design, group-level design and carry by-pass design.

The formal synthesis — although not fully mechanizable — is a promising method for designing provably correct digital circuits on an abstract level.

Acknowledgements

I thank S. Magnusson, A. Stümpel and A. Vox for carefully reading draft versions of this paper.

References


Appendix A

Case 1.1 \(a < b\).

\[
\begin{align*}
\pi_1(\text{code}(\text{comparator}(a, b))) & \equiv \pi_1(\text{code}(S)) \equiv \pi_1(1, 0, 0) \equiv 1 \quad \text{fold} < \\
\equiv a < b
\end{align*}
\]

All other cases 1.2 – 3.3 are proved in a similar way.

Appendix B

\[
\begin{align*}
lit & \equiv \pi_1 \circ \text{comp} \\
& \equiv \pi_1 \circ \text{code} \circ \text{comparator} \circ (\text{decode} \circ \text{first}) \circ (\text{decode} \circ \text{second}) \\
& \quad \{ \text{use (23)} \} \\
& \equiv < \circ ((\text{decode} \circ \text{first}) \circ (\text{decode} \circ \text{second})) \\
\end{align*}
\]

The other comparison functions are treated in a similar way.

Appendix C

Induction basis

\[
\begin{align*}
\text{comp}(\langle x, y \rangle) & \quad \{ \text{use (27)} \} \\
\equiv \langle \text{comp}(\langle x, y \rangle), \text{eq}(\langle x, y \rangle), \text{gt}(\langle x, y \rangle) \rangle \\
& \quad \{ \text{use (31), eq (33), and gt (35)} \} \\
\equiv (x < y, x = y, x > y) \\
& \quad \{ \text{introduce } \circ \} \\
\equiv x \circ y
\end{align*}
\]

Induction step

\[
\begin{align*}
\text{comp}(P \& Q) & \quad \{ \text{use (27) and } \circ \} \\
\equiv \langle \text{lit}(\langle x, y \rangle), \text{eq}(\langle x, y \rangle), \text{gt}(\langle x, y \rangle) \rangle \\
& \quad \{ \text{lit (31), eq (34), and gt (36)} \} \\
\equiv \langle \text{lit}(P \lor \text{eq}(P) \land \text{lt}(Q)), \text{eq}(P) \land \text{eq}(Q), \text{gt}(P) \lor \text{eq}(P) \land \text{gt}(Q) \rangle \\
& \quad \{ \text{introduce } \oplus (40) \} \\
\equiv \langle \text{lit}(P), \text{eq}(P), \text{gt}(P) \rangle \oplus \langle \text{lt}(Q), \text{eq}(Q), \text{gt}(Q) \rangle \\
& \quad \{ \text{fold } \circ \text{and } \text{comp (27)} \} \\
\equiv \text{comp}(P) \oplus \text{comp}(Q)
\end{align*}
\]