Synthesis Using Sequential Functional Modules (SFMs)

Samit Chaudhuri Michael Quayle
Cadence Design Systems
2655 Seely Road, Bldg. 6, MS 6B1
San Jose, CA 95134

Abstract

This paper presents a new method used in our RTL-synthesis tool to perform technology mapping with Sequential Functional Modules (SFMs) such as counters, accumulators, shift-registers, or rotators from any target or macro library. If the library contains SFMs, the method automatically recognizes them. If an RTL design contains patterns that can be implemented on SFMs, the method maps them to the SFMs found in the target library. This mapping reduces the design time by leveraging the library developer's effort, leads to more regular and often smaller and faster designs, and helps to reduce timing and routing problems at later stages of the design process.

1 Introduction

Most high-level synthesis systems [7] map data path operations to generic Register-Transfer Level (RTL) modules to achieve technology independence. However, in the world of industrial design, high-level synthesis tools need to map operations to RTL modules that exist in technology specific target or macro libraries.

Some RTL synthesis tools [8, 10], on the other hand, are capable of mapping data path operations to combinational functional modules of technology specific target libraries. Such modules include adders, subtracters, incrementers, decrementers, or comparators. Tools with similar capability also exist in the area of embedded processor design [6].

However, most synthesis tools lack the capability of mapping to Sequential Functional Modules (SFMs), such as counters, shift registers, or accumulators. SFMs are distinguished from sequential storage modules (eg., registers, latches) in that they perform some complex combinational function on data before storing its value.

In a general purpose synthesis tool, capability of mapping to SFMs is desirable because many recent industrial target and macro libraries (both FPGA and ASIC libraries) contain different counters, accumulators, shift registers, and rotators. By taking advantage of these modules, it is possible to synthesize more regular and often smaller and faster designs.

1.1 Motivation

As an example, consider synthesizing the Verilog RTL design shown in Figure 1. In this design, the value of `i` is incremented in each clock cycle. In netlists synthesized by most RTL synthesis tools, `i` is stored in a register (which is a simple sequential module), and is incremented using an incrementer (which is a combinational functional module). However, if the synthesis tool can map to a counter in the target library, then `i` can be stored and incremented using the counter (which is an SFM). Similarly `q` can be implemented with a shift register. This implementation is more regular and usually has smaller area than the register-incrementer and register-shifter implementation.

```
always @(posedge clock)
begin
    for (i = 0; i < 4'b1001; i=i+1)
    begin
        @(posedge clock)
        q = q << 1;
    end
end
```

Figure 1: RTL-level Verilog Description of a Design

Automatic inference and mapping of SFMs during RTL synthesis provides several advantages: (1) The SFMs in the library are usually carefully designed and pre-routed, enabling the designer to leverage the library developer's effort and reduce the design time. (2) Typically, SFMs are not processed during logic-level optimization and technology mapping. This reduces the CPU and memory requirement when performing logic synthesis. (3) Designs using SFMs often have smaller area. (4) An SFM encapsulates a register, some complex combinational logic, and their interconnections. Such encapsulation leads to more regular data paths and helps reduce timing and routing problems at later stages of the design process. (5) A netlist containing SFMs is easier to examine; the designer can easily inspect its structure and perhaps even substitute one SFM implementation for another.

1.2 Preliminaries and Previous Work

This paper describes a method used in our RTL synthesis tool to perform technology-mapping with SFMs of any target or macro library. The set of supported SFMs currently includes counters, accumulators, shift registers, and rotators, and can be extended to support additional modules.

In our method, the behavior of both the design and the library SFMs are described in a Hardware Description Language (HDL) such as Verilog or VHDL. The method recognizes patterns in the HDL description that are suitable for SFMs. In general terms, such patterns arise from variables

ICCAD '96
1063-6757/96 $5.00 © 1996 IEEE

that are stored across clock boundaries, and are often updated through arithmetic operations such as addition, subtraction, increment, or left shift.

If an SFM pattern is recognized in a library module, the module is further examined to determine whether it is really an SFM, and not an arbitrarily complex module that happens to contain an SFM pattern. If an SFM pattern is recognized in a design, then the pattern is mapped to an appropriate SFM from the target or macro library.

The problem addressed here involves automatic recognition and mapping with SFMs, where both the design and the library are described in an HDL. Formally, the behavior of an SFM can be described as a graph template containing cycles, conditional branches, and storage operations. Therefore, although the problem is closely related to template matching, tree-matching techniques presented by Aho and Ganapathi [1], or Keutzer [5] cannot be used. Work by Corazao *et al.* [3] does not consider templates with conditionals and storage operations.

Some synthesis tools such as [2, 9] recognize repeated patterns in a behavior. However, instead of mapping the recognized patterns onto SFMs in the target library, these tools implement them as interconnections of simpler library modules.

Tools such as DTAS [4] can automatically map generic counters onto target library counters. For example, it can implement a 8-bit generic counter using two 4-bit counters from the target library. However, DTAS does not automatically map patterns in an HDL design to 8-bit generic counters, nor can it automatically recognize 4-bit counters in the target library (the user has to manually find the counter and describe it in a special language).

The rest of the paper describes three major phases of our method:

Recognition: 1. SFM Pattern recognition

2. Library SFM recognition

Mapping: Mapping SFM patterns of the design to SFMs of the library

Binding: Determining the expressions for the control and data inputs of the selected SFMs

2 SFM-Pattern Recognition

The first step in our method consists of SFM pattern recognition in target library modules and the RTL design. Formally, an SFM pattern can be represented by a graph template containing cycles, conditional branches, and storage operations. Such a representation, although very general, does not necessarily lead to an efficient solution. As a matter of fact, a general graph-matching algorithm will be too expensive to be used on real-life designs.

Therefore we use a special-purpose algorithm that efficiently solves the problem at hand, and we choose a simpler representation of SFM patterns suitable for the algorithm. This representation is based on the understanding that an SFM implements both the LHS and the RHS of an assignment[1]. So both the LHS and the RHS should be considered in an SFM pattern. The notion of an SFM pattern can then be formalized with a (LHS, RHS) pair, called *c-pair* , which is introduced through the following definitions.

Definition 1 (*c-op*) A *c-op* denotes any arithmetic operation that an SFM can perform, and must be one of the following: addition, subtraction, increment, decrement, left-shift, right-shift, or concatenate.

Definition 2 (*s-op*) An *s-op* denotes any storage operation that an SFM can perform, and must be one of the following: synchronous (resp. asynchronous) set, reset, constant-load (e.g.,`x=4'b7`), or parallel-load (e.g.,`x=a`), and no-op (i.e.,`x=x`).

Definition 3 (*c-rhs*) A *c-rhs* is any expression involving a c-op; Verilog examples are: `x+1'b1`, `x + a`, `x << 1'b1`, and `{x[3:0], 1'b0}`. (see Table 1 for more examples).

Definition 4 (*c-lhs*) x is a *c-lhs* of an expression $f(x)$, if

1. x needs to be stored across clock boundaries, and
2. $f(x)$ is a c-rhs, and has one of the patterns listed in Table 1.

For a Verilog example, consider a c-rhs`{x[3:0], 1}`. If `x[4:0]` needs to be stored across clock boundaries, then it is a c-lhs for `{x[3:0], 1}`. Note that an assignment of the form `x[4:0] = {x[3:0], 1}` can be implemented on a shift register.

Definition 5 (*c-assign*) If $f(x)$ is a c-rhs and x is a c-lhs of $f(x)$, then assignment $x = f'(x)$ is called a *c-assign* when $f'(x)$ is either (1) $f(x)$, or (2) a conditional expression with $f(x)$ on one branch. For a Verilog example, if `x` is a c-lhs of `x+1`, then the following assignment is a c-assign

```
x = (ready == 1'b1) ?  x + 1 :  a;
```

Definition 6 (*c-pair*) The pair $(x, f(x))$ is called a *c-pair*, if there exists a c-assign $x = f'(x)$. In the example used in the c-assign definition, (`x, x+1`) is a c-pair.

The following example shows how c-pairs can be used in recognizing SFM patterns.

Example 1 *Suppose a variable* `x` *appears in the following Verilog description:*

```
@(posedge clock)
    x = x + 1;
    a = y + b;
@(posedge clock)
    x = x + a;
    y = x + a;
```

Although the assignment `x = x+a` *can be implemented using an accumulator, the assignment* `y = x+a` *requires* `x+a` *to be implemented on an adder because the value of* `x+a` *is not available at the accumulator output in the same clock cycle. Thus it is more desirable to implement* `x = x+1` *with a counter,* `x+a` *with an adder, and connect the data inputs of* `x` *and* `y` *registers to the adder's output.*

[1] in contrast, a combinational functional module implements only the RHS of an assignment, and a sequential storage module implements only the LHS of an assignment.

c-rhs for counter:
$x \pm u$
c-rhs for accumulator:
$x \pm b$
$x + b_1 + \cdots + b_n$
$(x + b_1 + \cdots + b_{k-1}) - (b_k + \cdots + b_n)$
c-rhs for shift-register (left):
$x << u$
$\{x, u\}$
$\{x[msb-1:lsb], u\}$
$\{x[msb-1:lsb], x[msb]\}$
c-rhs for shift-register (right):
$x >> u$
$\{u, x[msb-1:lsb]\}$
$\{x[lsb], x[msb-1:lsb]\}$

Table 1: Forms for c-rhs considered in Phase 1. In the above c-rhs expressions, we assume that (i) the range of u is a subset of $\{-1, 0, 1\}$, and (ii) the range of b is not a subset of $\{-1, 0, 1\}$

To deal with situations as above, it is more convenient to keep track of two c-pairs, (`x, x+1`) and (`x, x+a`). We can first nominate both pairs as candidates: the first pair for both a counter and an accumulator, and the second for an accumulator only. Later we select a counter implementation for the first pair, and reject the accumulator implementation for the second pair.

Our recognition algorithm consists of two phases: the first phase, *Nomination*, finds all possible c-pairs; the second phase, *Selection*, selects the nominated c-pairs based on a heuristic. These phases are described below.

2.1 Phase 1: Nomination of C-Pairs

The nomination phase finds all possible c-pairs in the design. It traverses each c-rhs, checks if any of its operands is a c-lhs, and checks if the c-lhs and c-rhs constitute a c-pair. Each such c-pair is then nominated as a candidate. The pseudo-code of this search procedure is given in Figure 2.

During nomination of c-pairs, we consider the semantics of each c-op (eg., sign and bit-width of the result) as specified in the particular HDL. Our current implementation requires the library modules to be unsigned, although it can be enhanced to handle signed library modules as well. If the design uses a different number representation, then SFM mapping is performed only when possible. For example, if a ranged-integer variable contains a constant sign-bit, then it may be possible to implement all bits but the msb of the variable with an unsigned counter.

2.2 Phase 2: Selection of C-Pairs

This phase examines the c-pairs nominated in phase 1, and selects the appropriate ones for SFM implementation. Not every c-pair should be implemented on an SFM. In Example 1, the c-pair (`x, x+a`), although nominated, should not be implemented on an accumulator, because then `x+a` would have to be implemented twice: once inside the accumulator, and once on an adder. Thus implementing a c-pair on an SFM may involve a trade-off:

Nominate C-Pairs:
(1) for each c-op τ do
(2) for each c-rhs exp of type τ do
(3) for each operand x of exp do
(4) if x is a c-lhs of of exp then
(5) if there exists a c-assign $x = g(exp)$ then
(6) nominate (x, exp) as a c-pair
(7) end
(8) end
(9) end

Figure 2: Algorithm for Nominating C-Pairs

SFM Trade-off: An SFM implementation of $(x, f(x))$ saves area, but also restricts the use of the result produced by $f(x)$ as follows. If $(x, f(x))$ is implemented on an SFM, the result produced by $f(x)$ can be used to update x only, and is not immediately available to the rest of the data path. Therefore if $f(x)$ is needed elsewhere, such as to update another variable (eg., $y = f(x)$), or to compute another expression (eg., $f(x) < a$), then either (1) $f(x)$ will again have to be computed using a different module, or (2) the computation will have to be delayed by one clock cycle. In case (1), hardware will be allocated twice for computing the same expression $f(x)$, and may lead to a larger-area design. In case (2), computations will be delayed and may lead to a slower design.

The above observation suggests that, while selecting a c-pair $(x, f(x))$, all expressions of x must be examined to determine SFM trade-offs. In other words, the design must be globally analyzed before selecting any c-pair.

2.2.1 Global-Analysis Heuristic

For each c-pair $(x, f(x))$, the entire design is searched for determining SFM trade-offs. The search visits every assignment whose *rhs* contains x, computes two quantities called *clash* and *overlap* and uses them to update the selection information. This process is described below.

The quantity, *clash*, is computed to determine whether the result produced by $f(x)$ is needed to compute *rhs*. Some examples of *clash* are illustrated in Table 2, and the pseudo-code for computing *clash* is presented in Figure 3.

$f(x)$	*rhs*	*clash*
`x + 1`	`{x, 2'b0}`	1
`x + 1`	`x + 1 - a`	∞
`x + a`	`x + a`	0
`x + a`	`(a == 1'b1) ? y : x + a`	0

Table 2: Verilog expressions for $f(x)$ and *rhs*, and their *clash*

The value of *clash* is interpreted as follows:

CLASH($f(x)$, $g(x)$):
if $g(x)$ is a conditional expression then
  for each branch $g_i(x)$ of $g(x)$ do
    brClash ← **CLASH**$(f(x),\ g_i(x))$
    if $brClash = 0$ then
      clash ← 0
    else if *brClash* $= \infty$ then
      clash ← ∞
    return *clash*
  end
else if $f(x)$ and $g(x)$ are identical then
  clash ← 0
else if $f(x)$ is a subexpression of $g(x)$ then
  clash ← ∞
else
  clash ← 1
return *clash*

Figure 3: Calculation of Clash between $f(x)$ and $g(x)$

1. A *clash* of ∞ implies that *rhs* (or at least one branch of *rhs*, if *rhs* is conditional) uses the value produced by $f(x)$; consequently, as explained in SFM trade-off, c-pair $(x, f(x))$ is not selected.

2. A *clash* of 1 implies that *rhs* does not use the value produced by $f(x)$; therefore the selection of c-pair $(x, f(x))$ is not affected by *rhs*.

3. A *clash* of 0 implies that *rhs* (or at least one branch of *rhs*, if *rhs* is conditional) is the same as $f(x)$. The selection of $(x, f(x))$ then depends on *overlap* which is described below.

When the *clash* between $f(x)$ and *rhs* is 0, the *overlap* between x and the left-hand side of the assignment is computed; *overlap*s are illustrated in Table 3.

x	***lhs***	***overlap***
x	y	none
x[7:4]	x[3:0]	none
x[7:3]	x[3:0]	partial
x[7:3]	x[7:2]	partial
x[7:3]	x[7:3]	full

Table 3: Verilog expressions for x and *lhs*, and their *overlap*

Example 2 *We show an example of how, during selection of c-pair (*`x[3:0], x[3:0]+1`*), clash and overlap are used when we visit the following assignments:*

1. For assignment `y = x[3:0]+1+a`*, clash is* ∞*, and we disqualify (*`x[3:0], x[3:0]+1`*).*

2. For assignment `x[3:0] = x[3:0]+1`*, clash is 0; so we compute overlap as "full" (see Table 3), and we favor the selection of (*`x[3:0], x[3:0]+1`*).*

WEIGH(x, $f(x)$, *assign*):
lhs ← left-hand side of assign
rhs ← right-hand side of assign
clash ← **Clash** ($f(x)$, *rhs*)
switch (*clash*)
case ∞:
  weight ← $-\infty$
case 0:
  switch (overlap of x and lhs)
  case full:
    weight ← 10
  case partial:
  case none:
    weight ← $-\infty$
  end switch
case default:
  weight ← 0
end switch
return *weight*

Figure 4: Routine for Assigning *weight* to a C-Pair

3. For assignment `x[3:0] = x[3:0]+1`*, clash is 0; so we compute overlap as "none" (see Table 3), and we disqualify (*`x[3:0], x[3:0]+1`*).*

Depending on the *overlap* and *clash*, a *weight* is added to the current *weight* of $(x, f(x))$. The pseudo-code for calculating *weight*s is given in Figure 4. A high *weight* will encourage the implementation of the c-pair on an SFM. Thus the *weight*s guide the selection of c-pairs for SFM implementation.

After the design has been searched for all the c-pairs, only those with *weight*s above a certain threshold are selected.

3 Library SFM Recognition

Library SFM recognition follows the SFM-pattern recognition process described in Section 2. If a library module contains any selected SFM, then library SFM recognition is performed to decide whether the module is really an SFM, and not an arbitrarily complex module that happens to contain an SFM pattern. Obviously, library SFM recognition is performed only during library compilation, and not during design synthesis.

To qualify as an SFM, a library module must contain a c-pair, and it must also meet other constraints, some of which are described below:

1. Module contains exactly one state
2. Module contains exactly one c-lhs, and no other register node
3. Each operation on the c-lhs is a c-op or an s-op

```
module accum(din,reset,clk,ld,en,udn,q);
  input [3:0] din;
  input udn, ld, en, clk, reset;
  output [3:0] q;
  reg [3:0] q;
  always @(reset)
    if (reset == 1'b0)
      assign q = 4'b0000; // async. reset
    else
      deassign q;
  always @(posedge clk)
  begin
    casex({ld,en,udn})
      3'b1??:  q = din;    // sync. parallel-load
      3'b011:  q = q+din; // add
      3'b010:  q = q-din; // subtract
     default:  q = q;      // no-op
    endcase
  end
endmodule
```

Figure 5: Behavioral Description of an Accumulator in a Target Library

For example, the module in Figure 5 satisfies the above constraints as follows: it contains exactly one c-lhs, `q`, and each operation on `q` is either a c-op or s-op, as indicated in the comments.

Furthermore, certain additional information about the library module is also extracted during SFM recognition. Such information includes (i) control expression for each c-op and s-op, and (for example, the control expression for the add operation in Figure 5 is {`ld, en, udn`}), and (ii) data ports such as parallel-in, carry-in (resp. shift-in) and carry-out (resp. shift-out) for counters/accumulators (resp. shift registers). This additional information is used while binding assignments to an SFM, as will be shown in Section 5.

4 Mapping

Section 2 described a method for selecting the c-pairs in a design. Attempts are then made to map the selected c-pairs onto SFMs from the target or macro library. This section describes our method of choosing SFMs from the library to map the selected c-pairs.

Each c-lhs in the design is examined in turn, and is mapped to the least expensive SFM in the library that can implement all the c-ops and s-ops that are performed on the c-lhs. The c-ops (e.g., increment, add, and left-shift) performed on the c-lhs can be found from the c-pairs associated with the c-lhs. Furthermore, the s-ops (e.g., set, reset) performed on the c-lhs can be found by examining all the assignments to the c-lhs. Thus a set of required c-ops and s-ops is computed.

For each required c-op or s-op, a weight is assigned to every SFM in the library. Then the SFM with the maximum total weight is chosen.

```
always @(posedge clk)
  casex(m+n)
    4'd08:  m = y;          // sync. parallel-load
    4'd06:  m = m+p;        // add
    4'd10:  m = m-q;        // subtract
    4'd07:  m = u;          // sync. parallel-load
  default:  m = m;          // no-op
  endcase
```

Figure 6: Design Fragment to be Mapped to an Accumulator

Each weight indicates how a particular c-op or s-op can be implemented on a library module. For example, if the c-lhs requires an no-op (no-operation; just retain the value from the previous state), then a library module with no-op capability will get a high weight. However, a library module with no no-op can also implement no-op by using parallel load to feed back the data from the previous state. Thus such a library module will receive a lower weight. If the c-lhs requires an asynchronous reset, then the library module must have the capability for asynchronous reset; otherwise the module will not be considered for mapping (i.e., receive a weight of $-\infty$).

Figure 6 shows part of a design containing a c-lhs `m`. The required c-ops are add and subtract, and the required s-ops are synchronous parallel load and no-op. The library accumulator of Figure 5 has capabilities for all the above operations, and hence is a good candidate for mapping.

5 Binding

After a c-lhs in the design has been mapped to an SFM in the library, the expressions for both the data and control inputs of the SFM have to be specified; this process is called binding. For example, Section 4 mentioned that the c-lhs of Figure 6 can be mapped to the library accumulator of Figure 5. For this mapping, the subsequent binding process is illustrated below.

The design of Figure 6 requires that when `m+n = 4'd06`, the library accumulator must perform the addition `x+p`. This addition can be performed on the library accumulator of Figure 5 by connecting the data input `din` to `p`, and the control inputs `ld, en,` and `udn` to `0, 1,` and `1`, respectively. A complete specification of the expressions for the data and the control inputs of the accumulator are shown in a concise form in Figure 7.

The above example illustrates a general method for determining the expressions of the input ports required to bind an assignment to an SFM. Expressions for the individual ports are later minimized using boolean optimization.

6 Results

This section presents some experimental results to show the effect of using SFMs on the synthesized designs.

Table 4 shows the area and timing reports of some designs synthesized both with and without SFMs. Design 1

```
// data inputs
casex ({m+n})
  4'd08:  din = y;
  4'd06:  din = p;
  4'd10:  din = q;
  4'd07:  din = u;
default:  din = 4'bxxxx;
endcase
// control inputs
casex ({m+n})
  4'd08:  {ld,en,udn} = 3'b1??;
  4'd06:  {ld,en,udn} = 3'b011;
  4'd10:  {ld,en,udn} = 3'b010;
  4'd07:  {ld,en,udn} = 3'b1??;
default:  {ld,en,udn} = 3'b00?;
endcase
```

Figure 7: Expressions for Control and Data Inputs of the the Library Module of Figure 5.

Design	with SFM		without SFM	
	Area	Longest Path (ns)	Area	Longest Path (ns)
Design 1	1406	20.55	1502	20.55
Design 2	503	5.75	925	8.36
Design 3	2848	2.62	2911	10.15
Design 4	2390	2.24	2687	10.15
Design 5	1237	2.21	1418	1.86

Table 4: Synthesis Results with and without SFMs

and 2 were synthesized using an ASIC target library that contains SFMs, while Design 3, 4, and 5 were synthesized using a different target library in conjunction with an ASIC macro library. The designs are industrial examples and we used commercially available ASIC libraries.

Each design in Table 4 contains some variables that can be implemented on SFMs. Such variables usually exist in address generator, clock divider, or event counting modules. Loop index variables also often get mapped to SFMs.

The results in Table 4 demonstrate that the area and timing of the synthesized designs are improved when SFMs are used. An apparent exception is Design 5, where timing in fact degrades when SFMs are used. This is because of the inefficient design of the counter in the macro library, where the longest path goes through a control signal. Such inefficiencies can be avoided by careful design of SFMs in the library.

7 Conclusions and Future Work

In this paper, we have demonstrated that using SFMs in RTL synthesis often improves the area and timing of the synthesized design. This work can be extended in the following ways to more effectively exploit the advantages of SFMs.

Experimental results in Section 6 showed that the timing of a design may be adversely affected by using SFMs that are not well-designed. SFMs in a target library, which are not well-designed, should be avoided.

Furthermore, using complex SFMs can affect resource sharing. For example, an accumulator encapsulates an adder and a register, and the adder and the register can not be shared freely (because the register stores only one particular variable, while the adder performs only those additions that involve that variable). The effect of this restriction on resource sharing needs to be addressed in the future.

8 Acknowledgment

The authors would like to acknowledge the contribution of Dr. Richard J. Cloutier in this project. His initial work, and his suggestions through later long discussions constitute an essential part of this work.

References

[1] A. V. Aho, M. Ganapathi, and S. W. K. Tjiang. Code Generation Using Tree Matching and Dynamic Programming. *ACM Trans. on Programming Languages and Systems*, 11(4):491–516, 1989.

[2] C. T. Chang, K. Rose, and R. A. Walker. Cluster-Oriented Scheduling in Pipelined Data Path Synthesis. In *Proc. of the IEEE International Conference on Computer Design*, Cambridge, Massachusetts, Oct. 3-6 1993. IEEE Computer Society Press.

[3] M. Corazao, M. Khalaf, L. Guerra, M. Potkonjak, and J. M. Rabaey. Instruction Set Mapping for Performance Optimization . In [11], pages 518–521.

[4] N. D. Dutt and J. R. Kipps. Bridging High-Level Synthesis to RTL Technology Libraries. In *Proc. of the 28th ACM/IEEE Design Automation Conf.*, pages 526–529, San Fransisco, California, June 17-21 1991. IEEE Computer Society Press.

[5] K. Keutzer. DAGON: Technology Binding and Local Optimization by DAG Matching. In *Proc. of the 24th ACM/IEEE Design Automation Conf.*, Miami Beach, Florida, June 1987. IEEE Computer Society Press.

[6] P. Marwedel. Tree-Based Mapping of Algorithms to Predefined Structures. In [11], pages 586–593.

[7] M. C. McFarland, A. C. Parker, and R. Camposano. The High Level Synthesis of Digital Systems. *Proceedings of the IEEE*, 78(2):301–318, Feb. 1990.

[8] M. Quayle and C. L. Huang. Complex Operator Synthesis. In *Proc. of the IEEE International Conference on Computer Design*, pages 514–517, Cambridge, Massachusetts, Oct. 10-12 1994. IEEE Computer Society Press.

[9] D. S. Rao and F. J. Kurdahi. Partitioning by Regularity Extraction. In *Proc. of the 29th ACM/IEEE Design Automation Conf.*, pages 235–238, Anaheim, California, June 8-12 1992. IEEE Computer Society Press.

[10] B. Sharma, M. Mahmood, N. V. Zanden, and A. Ginetti. Flexible Datapath Synthesis Using Parameterized HDL Components. In *Proc. of 3rd Asia Pacific Conf. on Hardware Description Language*, pages 82–85, Bangalore, India, Jan. 8–10 1996.

[11] *Proc. of the IEEE/ACM International Conference on Computer-Aided Design*, Santa Clara, California, Nov. 7-11 1993. IEEE Computer Society Press.