Chapter 21

Designing Special-purpose Co-processors Using the Data-Flow Paradigm

Bilha Mendelson
Baiju Patel
Israel Koren

21.1 Introduction

Design of general purpose computers using the data-flow paradigm has been an area of intensive research for over a decade, whereas control driven architecture has been the prime choice for the design of application specific ICs (ASICs) [4]. This research concentrates on designing high-performance ASICs using the data-flow paradigm. An ASIC designed using this approach is intended to be a co-processor attached to a host computer for executing a specific application.

The application, given in the data-flow language SISAL [10], is first translated to a data-flow graph (DFG). The DFG represents all the parallelism inher-

Supported in part by SRC under contract 89-DJ-125.

The authors are with the Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA 01003.

ent in the algorithm and therefore, the user is not required to identify the parallelism explicitly. The DFG for a given application is then "directly" mapped onto VLSI such that each node in the DFG corresponds to a cell in VLSI. The operation of the VLSI cells follows the data-driven execution model, i.e., each cell executes its task as soon as all the operands are available and the output buffer is free. The "direct" mapping of the DFG onto VLSI and the data-driven execution make it possible to exploit all the inherent parallelism in the algorithm without having the user identify it. Moreover, it is naturally suitable for pipelined operation. Therefore, this approach to ASIC design can provide high levels of concurrency and pipelining for general applications. The ASICs generated using this paradigm are referred to as *data-driven* ASICs throughout this chapter.

The rest of the chapter is organized as follows. Section 21.2 contains a brief description of the design process. Each step of the design process is then discussed in detail in sections 21.3 to 21.5. In section 21.3, the performance estimation of the application program is described. In section 21.4, the area minimization process is discussed in detail and it is shown that a large number of implementations can be generated for different performance requirements. Section 21.5 illustrates the layout generation step, and final conclusions are presented in section 21.6.

21.2 Design Process

The complete procedure for designing data-driven ASICs is shown in figure 21.1. These steps are outlined in what follows.

SISAL to DFG Translation: The application program is specified in SISAL [10], which is then translated using an optimizing compiler to a DFG. The nodes of the DFG represent the computations performed on operands or data values that flow through the directed arcs, which connect nodes. A typical DFG is composed of arithmetic (e.g., add, subtract, multiply, compare, etc.), logical (e.g., AND, OR, etc.) and control (e.g., *switch, merge, True, False, etc.*) nodes. The control nodes are used to implement the flow control for conditional and loop constructs. The DFG is then simulated using an event driven simulator, PARET [12], to verify the correctness of the application program.

Performance Estimation: Before the detailed design of the ASIC is initiated, a preliminary estimate of the performance of the application program is obtained. If the estimated performance is not satisfactory, then a new algorithm must be developed for the application. The performance of

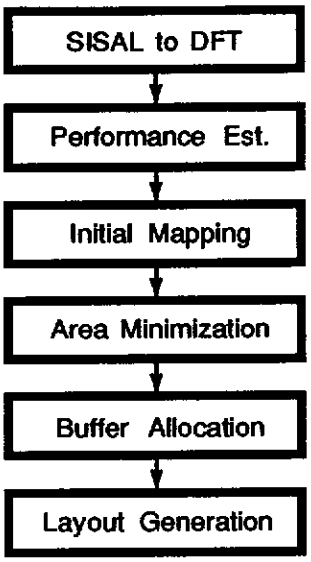


Figure 21.1: The design process

the ASIC is estimated hierarchically based on the estimated performance of arithmetic, logical, conditional, and loop constructs that constitute the DFG. At this stage, the implementation details are ignored.

Initial Mapping: The first step in mapping a DFG to VLSI is to assign an implementation to each of the nodes of the DFG. Since we are interested in designing very high-performance ASICs, initially, the fastest available implementation is chosen for each node. This, however, may result in an area wasteful implementation. Therefore, alternate implementations for certain nodes are then chosen to minimize the overall area within given performance constraints.

Area Minimization: Different operands at a multi-input node may arrive at different time instances by traversing different paths. Since the operand that arrives earlier has to wait for the other operands to arrive, the delay of the nodes on the shorter paths can be increased without affecting the overall performance. This, in turn, will reduce the area of the ASIC. Two algorithms for area minimization have been developed. The first is a heuristic algorithm which is very fast, and the second is a branch and bound algorithm which yields optimal results.

Buffer Allocation: If a DFG with non-uniform path lengths is directly mapped onto VLSI, it may not be optimally pipelinable. In such an event, buffers may be added to shorter paths of the DFG. The buffer allocation problem for this architecture is more difficult than the similar problem for synchronous pipelines [8] and static data-flow computers [5] because data-driven ASICs allow variable execution time for nodes and variable delay for buffers. This problem has been mapped to a quadratic programming problem that can be solved using well-known methods. The details of the buffer allocation algorithm are not provided here for the sake of brevity.

Layout Generation: The final layout is generated using OCT tools [18] developed at the University of California at Berkeley. A library of behavioral descriptions of the different node implementations is prepared. Then, the entire ASIC is synthesized from this library of behavioral descriptions using standard cell based generators.

21.3 Performance Estimation

The compiler translating the program written in SISAL to a DFG identifies the basic structures in the program and for each one of them creates the appropriate subgraph. Estimates for the potential performance of the program are obtained concurrently while creating the DFG in a hierarchical fashion. These estimates are implementation independent and therefore, do not account for any implementation overheads.

We use two performance measures: *latency*, which is the time, in clock cycles, from entering the input operands until the output is produced (measures the potential parallelism), and *pipeline period*, which is the mean time between successive results (measures the throughput which is its reciprocal).

We distinguish between two types of latencies: *worst case latency* (the input to output latency if the longest possible execution path is taken) and *average latency* (the average input to output latency, based on branch probabilities and estimates of iteration count in loop structures). Similarly, we define two types of pipelining measures: *worst case pipeline period* (the elapsed time between successive results if the longest operation in the algorithm is always executed) and *average pipeline period* (the average pipeline period based on branch probabilities and estimates of iteration counts).

We decompose the DFG into three types of structures: arithmetic/logic expressions, conditional expressions and loops, and estimate the potential parallelism and pipelining for each one of them. By combining the performance estimates for the basic structures hierarchically, we analyze the performance of the complete application.

In what follows, we present the data-flow graphs of the basic structures and derive expressions for their performance measures.

21.3.1 Arithmetic/logic expressions

A common way to implement an arithmetic/logic expression and achieve the best performance is through a balanced computation tree [1, 17]. The data has to pass through all possible paths in the computation tree of the arithmetic expression. Therefore, the latency of the arithmetic/logic expression is given by the length of the critical path.

The latency of an expression is denoted by $L(expression)$. The execution time of an operation is denoted by $EX(op)$. Given the estimated latencies of two sub-expressions, the estimated latency of the compound expression is calculated by the following recursive formula:

$$L(expression) = Max\{L(sub\_expression\_1), L(sub\_expression\_2)\} + EX(op)$$

The above formula assumes binary operation but it can be easily extended to n-ary operations.

The pipeline period of an expression is given by the execution time of the longest operation in the expression.

21.3.2 Conditional expressions

A DFG representation of a general conditional expression is shown in figure 21.2. This structure is composed of three parts: computing the condition, executing the *Then* or *Else* part, and routing the result of either branch through a *Merge* node. Routing of input data to either the *Then* or *Else* part is achieved using the *True* (T) and *False* (F) nodes. These nodes receive a data input and a Boolean control input. When the control value is true (false) the T (F) node passes the data to the outgoing arcs or otherwise consumes it.

The conditional structure is not a deterministic expression in the sense that the computation performed depends on the input data, and the path taken by the computation can not be determined a priori. Therefore, both the average and worst case performance measures need to be computed.

The worst case latency of an expression is denoted by $WL(expression)$. The latency of the *Then* and *Else* parts is equal to $WL(exp1) + EX(T)$ and $WL(exp2) + EX(F)$, respectively. Because of their similarity, we can assume that $EX(T) = EX(F)$ and we use the notation $EX(T)$. The worst case latency of the conditional structure is

$$\begin{aligned} WL(if\_then\_else) &= Max\{WL(exp1), WL(exp2)\} \\ &\quad + EX(T) + WL(Cond) + EX(Merge) \end{aligned}$$

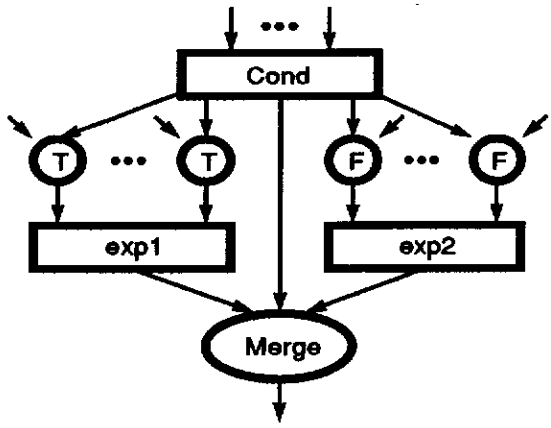


Figure 21.2: If-then-else structure

The probability of passing through the *Then* and the *Else* parts is denoted by p and $(1-p)$, respectively. The average latency of the *Then* part is equal to $AL(exp1) + EX(T)$ and that of the *Else* part is equal to $AL(exp2) + EX(T)$. The average latency of the complete if-then-else structure is therefore

$$\begin{aligned} AL(if\_then\_else) &= p * AL(exp1) + (1-p) * AL(exp2) \\ &\quad + EX(T) + AL(Cond) + EX(Merge) \end{aligned}$$

The worst case pipeline period is given by the longest operation in the structure. The calculation of the average pipeline period of a conditional structure is based on the average pipeline periods of the *Then* and *Else* paths. We call the path with the largest pipeline period the "long" path while the other is called the "short" path, and we denote the corresponding average pipeline periods by $AP(long)$ and $AP(short)$. Note that the *Cond* part belongs to both paths and consequently, $AP(long) \geq AP(Cond)$ and $AP(short) \geq AP(Cond)$. Let the probability of passing through the "short" path be denoted by p_s (p_s is either p or $1-p$). Assuming geometrical distribution of the selected paths, the average number of times that the "short" path will be selected successively is denoted by D and is equal to $D = \frac{p_s}{1-p_s}$.

Consecutive computations are at least $AP(Cond)$ clock cycles apart. Therefore, several computations in the "short" path can overlap a single computation in the "long" path only during $AP(long) - AP(Cond)$ clock cycles. We denote the ratio between the maximum overlap time and the average pipeline period

of the "short" path by R, i.e., $R = \frac{AP(long)-AP(Cond)}{AP(short)}$.

If D is smaller than R, then there is a complete overlap between the D computations in the "short" path and the single preceding computation in the other branch. Therefore, the average pipeline period will be equal to $AP(long)$ divided by $(D+1)$. If, however, D is larger than R, then the time to complete $D+1$ consecutive computations is determined by the time needed to complete the D computations in the "short" path. The computation in the "short" path can start only $AP(Cond)$ time units after the computation in the "long" path has started. Hence, we need to add this term to the overall computation time. The average pipeline period of the if-then-else structure is [11]:

$$AP(if\_then\_else) = \begin{cases} \frac{AP(long)}{D+1} & \text{if } D < R \\ \frac{D*AP(short)+AP(Cond)}{D+1} & \text{otherwise} \end{cases}$$

The above estimations are based on the assumption that the incoming data is always available when needed, i.e., the input rate is not smaller than the internal throughput. Consequently, the calculated estimates tend to be optimistic.

21.3.3 Loop structures

A DFG of a loop structure is composed of two parts: the control part, which determines the number of iterations to be executed, and the body which contains the computation that has to be repeated.

In many cases, the number of iterations needed is not known at compile time. Therefore, in these cases we estimate the latency and pipeline period based on user supplied values[1] of the average or worst case number of iterations, which will in turn yield average or worst case estimations, respectively. Therefore, we use the same notation for average and worst case measures.

In general, a loop may generate a single result or a stream of results. We next estimate the performance when the loop structure produces a single result and then we analyze the other case.

Single result loop structure

The DFG of a typical loop structure with a single result is shown in figure 21.3. The control signal passes to the synchronization nodes, which control the input streams, (stream_1,...,stream_I). These synchronization nodes are *Stream Modulo* nodes (**S.Mod**) that route the incoming data stream to the appropriate outgoing link in a *Round Robin* fashion. By replicating the loop body, we can

[1] These values may be determined through simulations on typical input data.

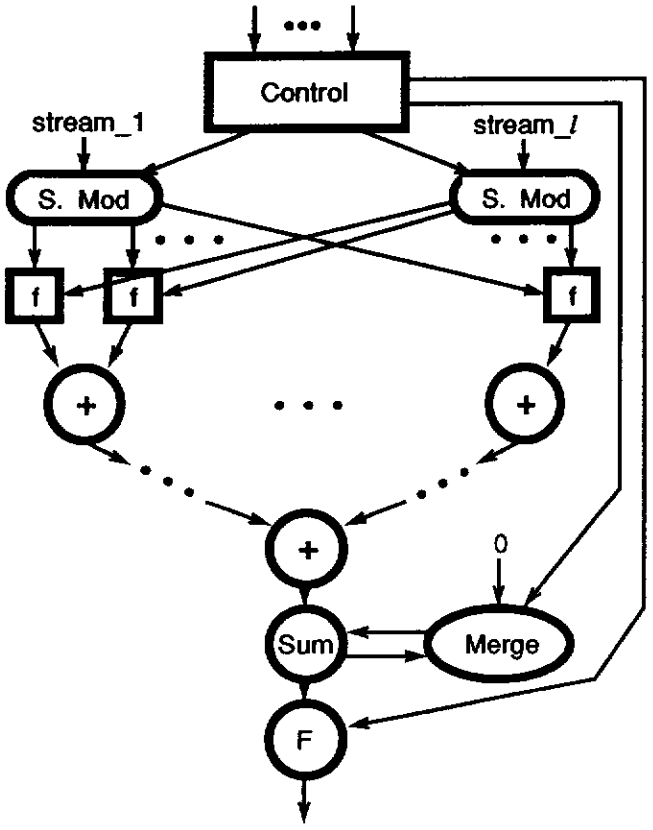


Figure 21.3: Single result loop structure

achieve a better performance and therefore we analyze a loop structure containing several replicas of the body, denoted by f blocks in figure 21.3. Detailed analysis, including the derivation of the optimal number of replications needed to maximize performance, appears in [11]. The partial results generated by all copies of the f block have to be accumulated to produce the loop output. This task is accomplished in two steps: add the partial results using a computation tree (e.g., *plus* (+) nodes) and then accumulate the partial results of the various iterations (e.g., *Sum* node). In summary, the loop body consists of m replicas of the f block and a summation tree as shown in figure 21.3.

The input to the loop body is a stream of data elements with inter-arrival time t. For simplicity, we assume that t is a constant. If the inter-arrival time is not a constant we may use its expected value.

The latency of the loop body depends on the latency of an f block, denoted by $L(f)$, the number of f block replications, m, and the latency of the summation tree, denoted by $L_{sum.\ tree}$.

We can divide the summation tree into two sub-trees; one is a complete binary tree with $\lfloor \log m \rfloor$ levels and the second is a partial tree. We can divide the partial tree again into two sub-trees; one is a complete tree with $m - 2^{\lfloor \log m \rfloor}$ leaves and the other is a partial tree. We may continue this process until the partial tree will be either a complete tree or will be empty. Therefore, the latency of the summation tree is given by the following recursive formula:

$$L_{sum.\ tree}(m) = Max\{LC(m), LP(m)\} + \delta(m)EX(+)$$

where $LC(m)$ is the latency of the complete tree and is given by

$$LC(m) = (2^{\lfloor \log m \rfloor} - 1)t + EX(+)\lfloor \log m \rfloor$$

$LP(m)$ is the latency of the partial tree and is given by

$$LP(m) = \begin{cases} 0 & m = 0 \text{ or } m - 2^{\lfloor \log m \rfloor} = 0 \\ 2^{\lfloor \log m \rfloor} t + L_{sum.\ tree}(m - 2^{\lfloor \log m \rfloor}) & \text{otherwise} \end{cases}$$

and

$$\delta(m) = \begin{cases} 0 & m = 2^k \quad (k \text{ is an integer}) \\ 1 & \text{otherwise} \end{cases}$$

The latency of the body can therefore be estimated as

$$L(body, m) = (\lceil \frac{n}{m} \rceil - 1)P(body) + CL + L_{sum.\ tree}(m)$$

where n is the original number of iterations,

$$CL = L(f) + EX(S.Mod) + EX(Sum) + EX(F)$$

and $P(body)$ is the pipeline period of the body determined by $Max\{IP(body),\ mt\}$. $IP(body)$ is the execution time of the longest operation in the body.

The optimal number of f block replications necessary to achieve the minimum possible execution time of a single result loop structure is

$$m_{opt} = \begin{cases} \left\lceil \frac{IP(body)}{t} \right\rceil & \text{if } \lceil \frac{IP(body)}{t} \rceil \le \frac{1}{2} + \sqrt{\frac{1}{4} + \frac{n \cdot IP(body)}{t + EX(+) + IP(body)}} \\ \left\lceil \frac{n}{\left\lceil \frac{n}{\lceil \frac{IP(body)}{t} \rceil} \right\rceil} \right\rceil & \text{if } \frac{1}{2} + \sqrt{\frac{1}{4} + \frac{n \cdot IP(body)}{t + EX(+) + IP(body)}} < \lceil \frac{IP(body)}{t} \rceil < n \\ n & \text{otherwise} \end{cases}$$

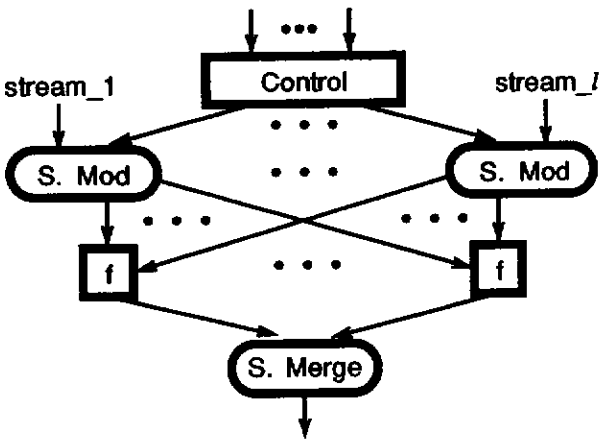


Figure 21.4: Stream of results loop structure

The corresponding minimal latency of the body is $nt + LP + L(sum.\ tree)$ [11].

Finally, the latency of the single result loop structure is

$$L(loop) = L(control) + L(f)$$

where $L(control)$ denotes the latency of the control part of the loop. The pipeline period of the single result loop is the same as its latency.

Stream of results loop structure

A stream of results loop structure is shown in figure 21.4. For synchronization purposes we use two nodes: *Stream Modulo* (**S.Mod**), which synchronizes the stream of inputs, and a *Stream Merge* (**S.Merge**), which guarantees the proper ordering of the results.

By replicating the f block, we allow overlapping of consecutive computations within the loop. This way, the loop structure can produce new results at a rate that may be smaller than $EX(S.Mod)$ and $P(f)$, where $P(f)$ is the pipeline period of the f block. As is shown in [11], the optimal number of replications is

$$m_{opt} = \lceil \frac{Max\{P(f), EX(S.Mod)}{Max\{t, P(control), EX(S.Merge)\}} \rceil$$

where $m_{opt} \leq n$. The minimal pipeline period of the loop structure is

$$P(stream\ of\ results\ loop\ structure) = Max\{t, P(control), EX(S.Merge)\}$$

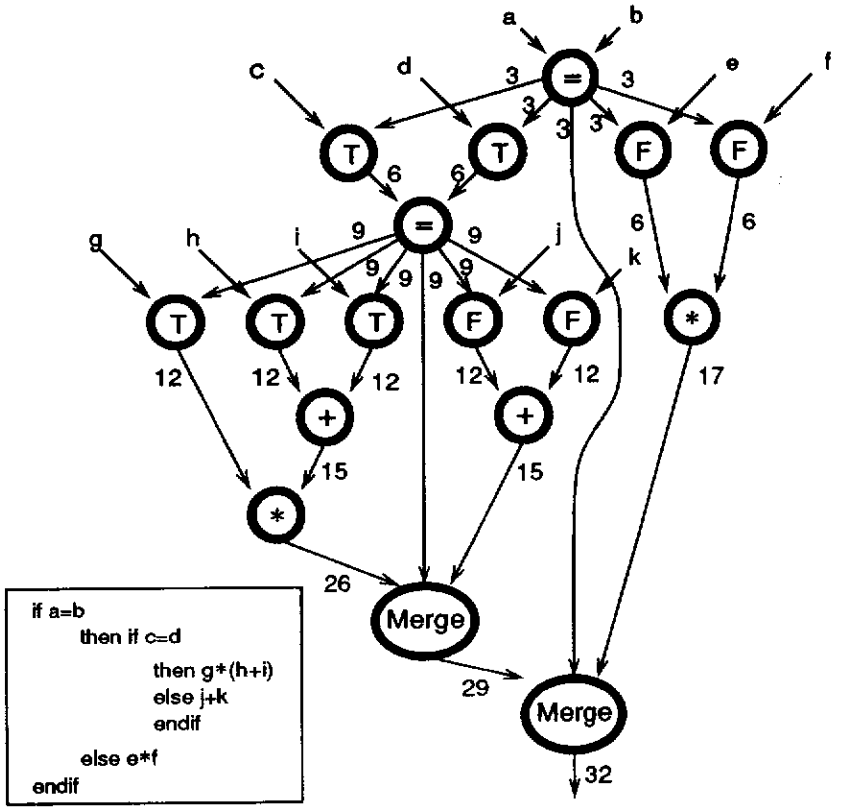


Figure 21.5: Nested if-then-else expression and its DFG representation

and the corresponding latency is

$$L(first\ result) = L(f) + EX(S.Mod) + EX(S.Merge) + L(control)$$

21.3.4 Examples

We demonstrate the performance estimation method through a simple nested if-then-else program. Figure 21.5 shows the program and its corresponding DFG generated by our compiler.

For the analysis of this example, we use the execution times from [7]. The number marked on each arc represents the accumulated worst case latency at that point. As can be seen from the figure, the latency of the complete DFG in

the worst case is equal to 32 clock cycles and the pipelining period is 11 clock cycles. The above results correspond to the length of the critical path and the longest operation in the graph, respectively.

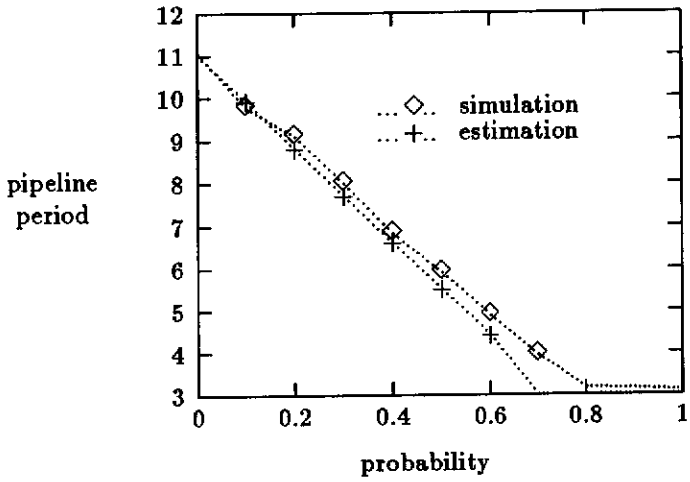
In figure 21.6, we compare the estimated values of the average pipeline period of the example in figure 21.5 to the simulation results obtained using the event simulator, PARET [12]. Figure 21.6 shows the average pipeline period as a function of the probability of taking the *Then* path of the outer conditional. The *Then* path is the "short" path. In figure 21.6(a), the probability to pass through the *Then* path of the inner conditional is 0.2. We can see that as the probability to pass through the outer *Then* path increases, the average pipeline period approaches $AP(short)$. In figure 21.6(b), the probability to pass through the *Then* path of the inner conditional structure is equal to 0.8 instead of 0.2. In this case, as the probability of passing through the outer *Then* path increases, the average pipeline period decreases. This continues as long as there is a complete overlap between the *Else* and *Then* branches of the outer conditional.

Further increase in the probability of taking the outer *Then* path reduces the overlap and results in an increase in the average pipeline period. As can be seen from the figure, the estimated values are very close to the simulation results and therefore, lengthy simulations may be avoided. In both cases, the worst case pipeline period is 11 clock cycles, which is substantially higher than the average case pipeline period.

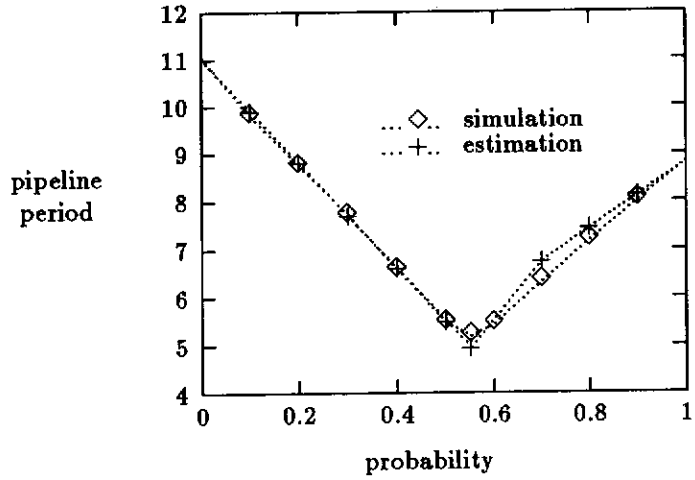
Figure 21.7 shows a first order impulse response filter [6] as an example for a loop structure where the current iteration depends on the previous iteration. Because of the dependency between successive iterations, the pipeline period of the body is: $EX(*) + EX(+) + EX(Merge)$. Replicating the body will not reduce this pipeline period and therefore the latency of the loop structure. In this example, the result of the first iteration is produced after 26 clock cycles, which is the accumulated execution time of the operations along the critical path. The second result, however, is produced 17 clock cycles later and not 11, which is the execution time of the longest operation in the graph (the multiply operation). Here, the pipeline period of the loop structure is determined by a sequence of operations that cannot be overlapped, which includes the *multiply*, *add*, and *Merge* nodes.

21.4 Area Minimization

The area minimization procedure starts with the initial mapping and reassigns smaller implementations to some of the nodes to minimize the overall area for the desired performance. For a pipelined ASIC, the single most important performance measure is the throughput. Since the pipeline period of the designed



(a) The probability to pass through the *Then* path of the inner conditional is 0.2



(b) The probability to pass through the *Then* path of the inner conditional is 0.8

Figure 21.6: Comparing the estimated pipeline period to simulation results for the example in figure 21.5

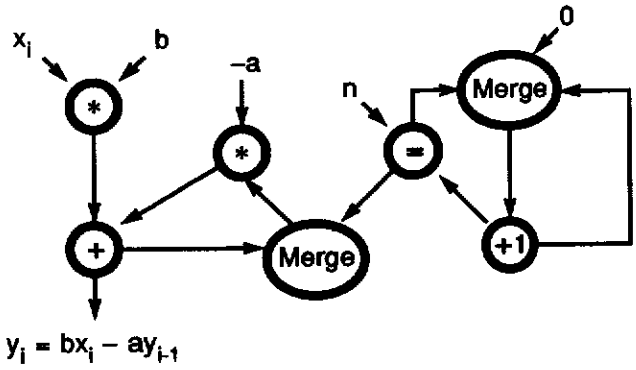


Figure 21.7: First order impulse response filter

ASIC can never be smaller than the delay of the slowest node, the smallest (slowest) implementation that can be used for a node is restricted by the desired pipeline period. The area minimization process will not select an implementation for any node of the DFG such that its delay is larger than the allowed pipeline period. The need for area minimization can be illustrated through the example in figure 21.8. Let the *multiply* (MULT) node be a sequential multiplier that takes 16 cycles for the computation. Let the *add* (ADD1 and ADD2) nodes be parallel 16-bit adders that take one cycle to execute. Therefore, for the initial mapping, the result of *MULT* is available 15 cycles later than the result of *ADD1*. Since *ADD2* cannot execute until both operands are available, the result of *ADD1* has to wait for 15 cycles after it has been computed. However, an alternate bit-serial implementation (with execution time of 16 cycles) may be used for node *ADD1* without affecting the overall performance of the ASIC. Moreover, the area of the interconnections for *ADD1* would be reduced as well. A parallel adder should still be used for *ADD2* because any slower implementation will increase the length of the critical path and thus increase the latency.

In general, there may be many different implementations for a node, and the area minimization will select an implementation for each node such that the overall area of the ASIC is minimal. If the area of the final design is not acceptable, the design process will be repeated for a larger value of latency to further reduce the overall area.

The above area minimization, in general, is an NP-complete problem since, for every node, there are many implementations and each of these may lead to a different system area and execution time. For example, if the DFG contains n

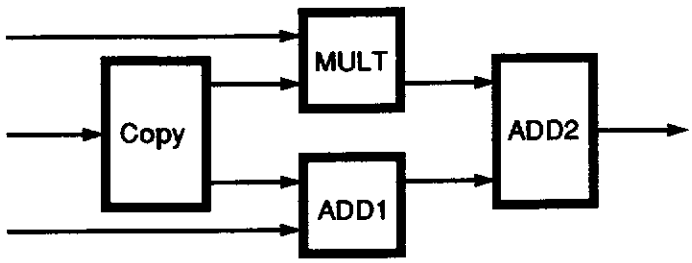


Figure 21.8: A simple DFG

adders and there are five implementations of the adder, then there are 5^n possible designs. Therefore, a greedy algorithm was developed to obtain a "good" solution while a branch and bound algorithm is used for optimal solution. At present, our algorithms ignore the area required for the interconnections between the nodes. Hence, the final layout may be larger than the area estimated by the area minimization step. However, we observe that in most cases, the interconnection area reduces with the smaller (e.g., serial) implementation of a node.

21.4.1 A greedy algorithm

From the initial mapping, a node is selected and its implementation is replaced by a smaller one such that the total area is reduced, but the overall latency remains unchanged. This process is repeated until no such node exists. If the area of the resultant ASIC is unacceptably large, the length of the critical path (latency) is increased and the process is repeated. The quality of the solution depends primarily upon the order in which nodes are selected for replacement and on the set of implementations available for each type of node.

Preliminaries

A node cannot start its execution before all the required inputs are available; therefore, the earliest time it can execute is the maximum of the earliest time instances at which the results from the predecessor nodes are available. Similarly, the node must complete its execution before the smallest of the latest time instances at which its output must be available to the successor nodes. These two time instances are denoted by *asap* (as soon as possible) and *alap* (as late as possible). The *asap* and *alap* times, also referred to as *slack* and *surplus* time, have been used for several scheduling problems (e.g., [9]). We define the *freedom* of a node as the largest amount by which its execution time

Algorithm: Area Minimization

1. Compute the freedom for each node.
2. Let S be the set of candidate nodes for which there are implementations such that when replaced, the increase in the delay is not larger than their freedom.
   If S is empty then exit.
3. Compute the *area savings* for each v in S.
4. Let $S' = \{v \mid v \in S$ and v has maximum area savings$\}$
5. Choose node v from S' which smallest freedom.
6. Replace current implementation by a smaller one.
7. Go to step 1.

Figure 21.9: Greedy algorithm for area minimization

can be increased without increasing the execution time of the overall design (which is given by the difference between the *alap* and *asap* times). Obviously, the freedom of a node on the critical path is zero.

Node Selection

In order to minimize the overall area, some of the nodes in the fastest implementation of the DFG are replaced by slower ones. The implementation of a node can be substituted by a smaller one if and only if the increase in its execution time is not larger than its freedom. The order in which the nodes are replaced will affect the solution and hence, determine the overall size of the final design.

The greedy algorithm attempts to minimize the overall area by selecting the node that provides maximum area savings to be replaced first. If there is more than one such node, the one with the smallest freedom is selected. Whenever the execution time for a node is increased, the freedom for some other nodes may be reduced. Therefore, if the node with the smallest freedom is chosen for replacement, then the freedom of the other nodes (with larger freedom) may decrease, but need not become zero. The above greedy algorithm is summarized in figure 21.9. A more detailed discussion of this algorithm appears in [15].

21.4.2 Branch and bound algorithm

An implementation of a node can be replaced by a smaller one only if the increase in the execution time is smaller than its freedom. Therefore, the lower

time(clocks)	1	2	4	8
bits	16	8	4	2
area(mm^2)	0.957	0.750	0.577	.500

Table 21.1: The area and execution time for different implementations of a 16-bit adder

bound for the area of the DFG is evaluated by choosing the smallest implementation within the above constraint. The area lower bound is used during both the branching and bounding steps. At the branching step, out of the nodes that can be replaced by smaller implementations, the node that leads to the smallest lower bound is selected. A tie in the selection is broken in favor of the node with the smallest freedom and largest area savings. Bounding takes place when the current smallest solution is smaller than the area lower bound.

21.4.3 Examples

To demonstrate the practical nature of the area minimization algorithms we present an example taken from [14], as shown in figure 21.10, where "S" and "M" nodes are *Switch*[2] and *Merge* nodes, respectively. The add/subtract nodes were designed using 2μ technology. The area and execution time for different implementations of these nodes are shown in table 21.1. The "bits" indicate the number of bits that are operated simultaneously, i.e., 8 bits indicate that 16-bit addition is performed by adding 8 bits at a time in two clock cycles. Both the greedy and branch and bound algorithms for area minimization use an 8-bit adder implementation (delay of two clock cycles) for the *ADD3, ADD4, ADD5, SUB2, SUB4,* and *SUB5*, and a 4-bit adder (delay of four clock cycles) for *SUB1*. The rest of the nodes are 16-bit adder implementations. Consequently, the total area is reduced from initial area of 15.32 mm^2 to 13.69 mm^2 while the overall execution time remains unchanged (16 clock cycles). The area-time tradeoff curve for this example obtained using the worst case latency measure is shown in figure 21.11.

In all the examples we tried (including the one above), the overall area obtained using the greedy algorithm is no more than 5% larger than the optimal area obtained using the branch and bound algorithm. Therefore, we suggest first generating an approximate set of solutions for different area and latency using the greedy algorithm. Once a solution is selected, it may be optimized using the branch and bound algorithm.

[2] A switch node is equivalent to a pair of *True* and *False* nodes.

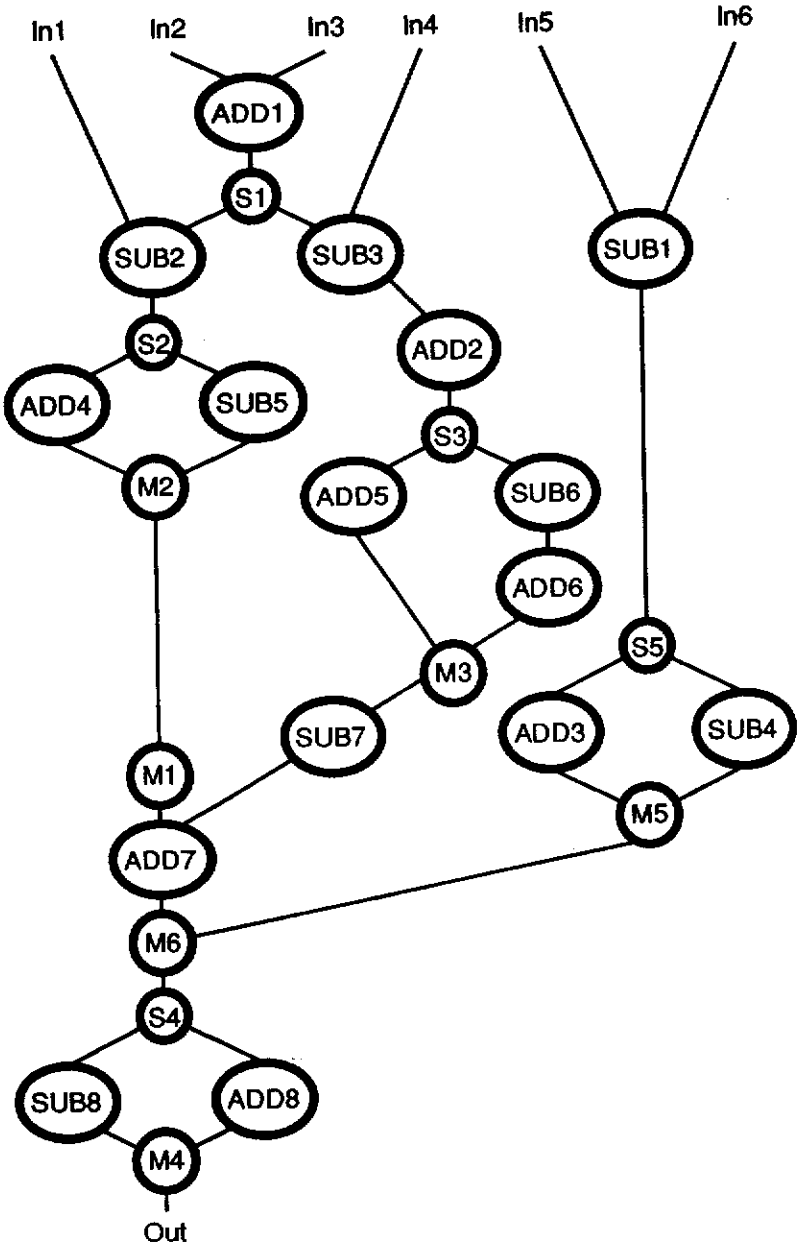


Figure 21.10: An example from MAHA [14]

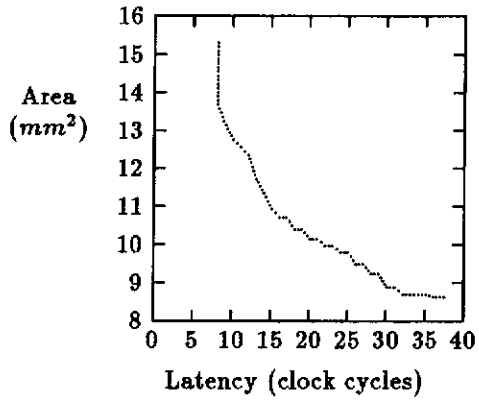


Figure 21.11: Area vs. time for the DFG of figure 21.10

21.5 Layout Generation

The final step in the synthesis of data-driven ASICs is the layout generation. The DFG for the application is translated into a netlist from which the ASIC is generated using the OCT tools [18]. The cells in the final layout may either be predesigned or generated as needed. The alternatives are standard cell or macrocell based layout generation schemes, as outlined below.

The first approach generates the final layout using standard cells only (all the cells are of uniform height). A library of the behavioral description of all the nodes is prepared in the language BDS [18]. The BDS description of a parallel adder (16-bit) is shown in figure 21.12. The BDS description of a node is translated into a netlist using BDSYN and MisII [2]. The netlist for the complete ASIC is then obtained by combining the netlists of the nodes and the netlist of the DFG using BDNET. This netlist, when placed and routed using the standard cell based placement and routing program Wolfe [16], generates the complete layout for the ASIC.

We have generated different layouts for the conditional statement of figure 21.13, and their area and latency are shown in figure 21.14. The "final area" in this figure indicates the area of the ASIC after placement and routing, and the "estimated area" indicates the area calculated by the area minimization algorithm (ignoring interconnects). It can be observed that the nature of both curves is similar. The timing analysis using Crystal [13] shows that all of these initial implementations will operate at least at 5MHz clock. One sample layout is shown in figure 21.15.

We have also generated layouts for several implementations of the DFG of

```
MODEL add_16
        out_1<16:1>,                        !the output
        send_o1<0>,                         !send signal for output
        ack_i1<0>, ack_i2<0>,               !ack signal for each input
=       in_1<16:1>, in_2<16:1>,             !two inputs
        send_i1<0>, send_i2<0>              !send for each input
        ack_o1<0>                           !ack signal at output port
ROUTINE add;
        ack_i1<0> = ack_o1<0>;
        ack_i2<0> = ack_o1<0>;
        send_o1<0> = send_i1<0> AND send_i2<0>;
        out_1<16:1> = in_1<16:1> + in_2<16:1>;
ENDROUTINE;
ENDMODEL;
```

Figure 21.12: The BDS description of a 16-bit adder

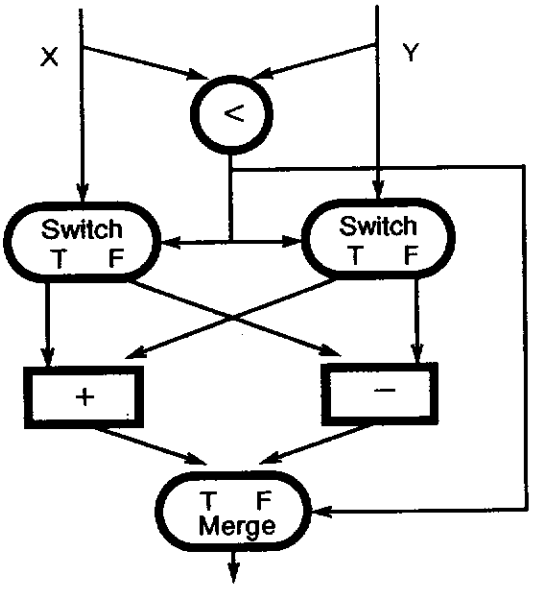


Figure 21.13: A conditional statement

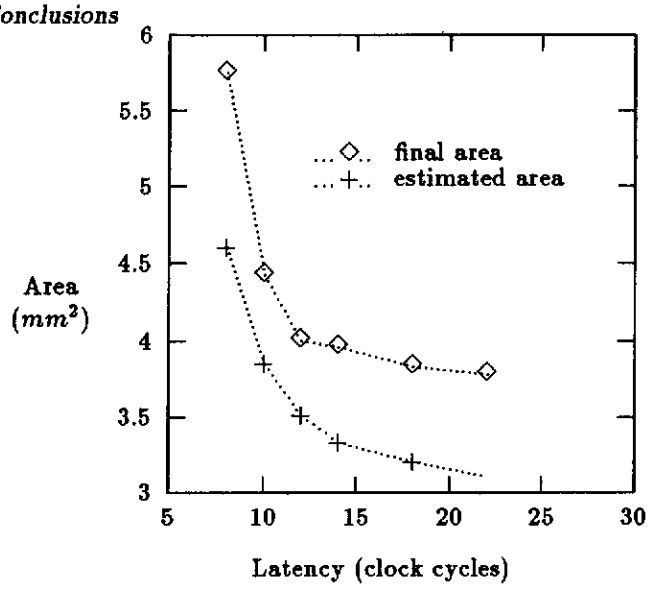


Figure 21.14: Area vs. time for the DFG of figure 21.13

figure 21.10. For example, the total area required for two designs with latencies of 12 and 16 cycles are 19.4 mm^2 and 17.9 mm^2, respectively. This clearly demonstrates the applicability of this approach to large DFGs.

The requirement that the complete ASIC be generated using a standard cell based generator may result in a suboptimal design. Instead, frequently used cells can be predesigned for optimal area/performance ratio and others may be generated using different target architectures (such as PLA). Finally, a macrocell based placement and routing tool (e.g., Mosaico [3]) should be used to place and route the separate cells.

21.6 Conclusions

An innovative approach to the design of ASICs has been presented in this paper. The designed ASICs operate in a data-driven mode that supports fine grain parallelism and pipelining. The developed CAD tool includes a compiler for translating the application specified in SISAL to a data-flow graph. This compiler also provides estimates for the performance of the ASIC. In the next step, the area of the ASIC is minimized and then the final layout is generated. Examples illustrating the various steps in the design of the ASIC have been presented.

Our preliminary experiments show that a DFG with about 50 to 100

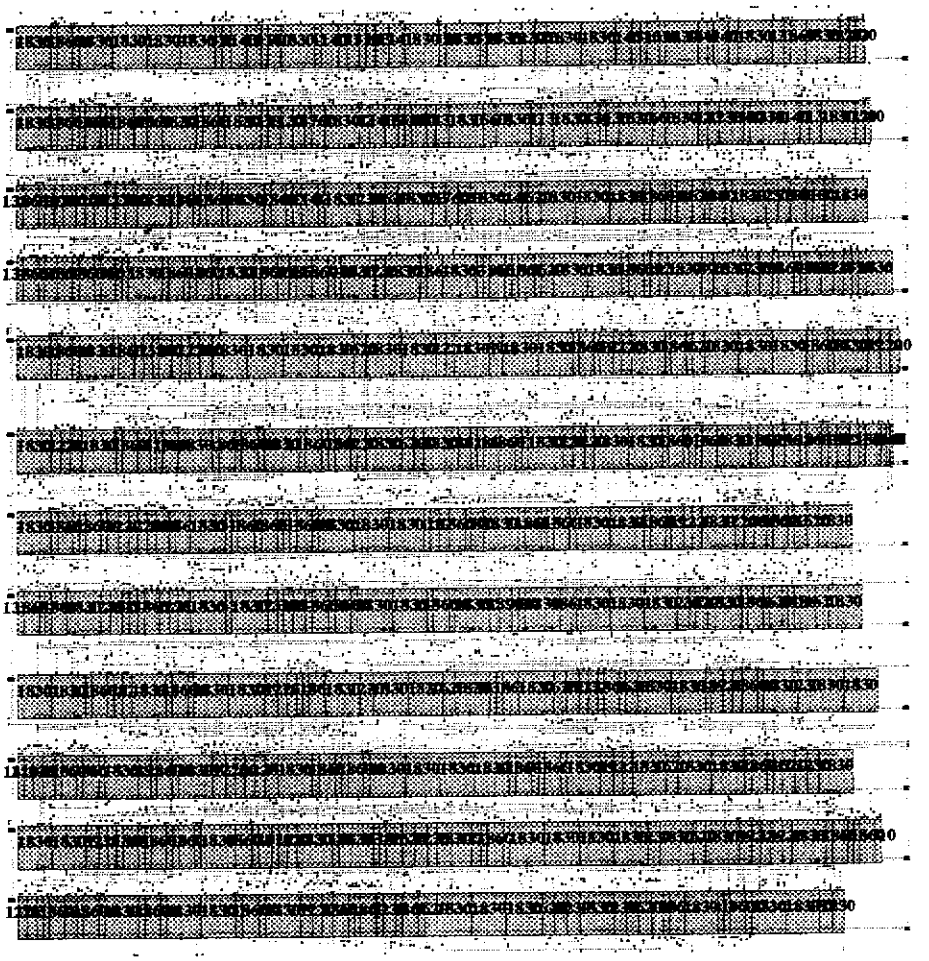


Figure 21.15: Layout of the DFG of figure 21.13

add/subtract operators can easily be implemented on a 1 cm^2 chip. Also, the timing analysis using Crystal and Spice shows that these ASICs can be operated at 5MHz to 10MHz clock rate.

References

[1] A.V. Aho and S.C. Johnson. Optimal code generation for expression trees. *J. ACM*, 23:488–501, November 1976.

[2] R.K. Brayton et al. MIS: A Multiple-Level Logic Optimization System. *IEEE Transactions on CAD*, CAD-6(6):1062–1081, November 1987.

[3] J. Burns et al. Mosaico: An integrated macro-cell layout system. In C.H Sequin ed., editor, *Proc. of VLSI '87*, Vancouver, Canada, 1987.

[4] D.D. Gajski. *Silicon Compilation.* Addison-Wesley Publishing Co., 1987.

[5] G.R. Gao. Algorithmic aspects of balancing techniques for pipelined data flow code generation. *Journal of Parallel and Distributed Computing*, 1(6):39–61, February 1989.

[6] L.B. Jackson. *Digital Filters and Signal Processing.* Kluwer Academic Publishers, 1986.

[7] I. Koren, B. Mendelson, I. Peled, and G.M. Silberman. A data-driven VLSI array for arbitrary algorithms. *Computer*, 21(10):30–43, October 1988.

[8] C.E. Leiserson and J.B. Saxe. Optimizing synchronous systems. *J. of VLSI and Computer Systems*, 1(1):41–67, January 1983.

[9] M.C. McFarland, A.C. Parker, and R. Camposano. Tutorial on high–level synthesis. In *Proc. of 25rd Design Automation Conference*, pages 330–336, 1988.

[10] J. R. McGraw et al. SISAL: Streams and iteration in a single assignment language: Reference manual version 1.2. Manual M-146, Rev. 1, Lawrence Livermore National Laboratory, Livermore, CA, March 1985.

[11] B. Mendelson and I. Koren. Estimating the potential parallelism and pipelining of algorithms. Technical Report TR-90-CSE-5, ECE Dept. University of Massechusetts, Amherst, 1989.

[12] K.M. Nichols and J.T. Edmark. Modeling multicomputer systems with PARET. *Computer*, 21(5):39–48, May 1988.

[13] J.K. Ousterhout. A switch-level timing verifier for digital MOS VLSI. *IEEE Transactions on CAD*, CAD–4(3):336–348, July 1985.

[14] A.C. Parker et al. MAHA: A program for datapath synthesis. In *Proc. of 23rd Design Automation Conference*, pages 461–466, 1986.

[15] B. Patel, I. Koren, and D.K. Pradhan. Designing highly pipelined ASICs using the data flow paradigm. Technical Report TR-89-CSE-9, ECE Dept. University of Massechusetts, Amherst, 1989.

[16] C. Sechen and A. Sangiovanni-Vincentelli. The TimberWolf placement and routing package. *IEEE Journal of Solid State Circuits*, 20(2):510–522, April 1985.

[17] R. Sethi and J.D. Ullman. The generation of optimal code for arithmetic expression. *J. ACM*, 17:715–728, October 1970.

[18] R. Spickelmier, editor. *Oct Tools Distribution 3.0.* University of California, Berkeley, March 1989.