Area and Performance Estimation from System-level Specifications

Sanjiv Narayan

Daniel D. Gajski

Technical Report ICS-92-16

December 20, 1992

Dept. of Information and Computer Science

University of California, Irvine

Irvine, CA 92717-3425

(714) 856-8059

narayan@ics.uci.edu

Abstract

An important task in the system level synthesis process is estimating design parameters such as area and performance that will characterize any implementation of the design. Estimates enable the designer to rapidly explore large search spaces and provide him/her with valuable feedback on the effect and feasibility of implementing certain design decisions. In this report, we present techniques for estimating the area and performance of a design specified using the SpecChart language which is a combination of hierarchical/concurrent state diagrams and VHDL.

Contents

1 Introduction **5**

1.1 Motivation 5

1.2 Inputs to the Estimators 7

2 Design Model for Estimation **8**

2.1 Datapath 8

2.2 Control Unit 9

3 Performance Estimation **10**

3.1 Performance Estimation Approach 10

3.2 Computing Performance for Leaf States 12

3.2.1 Constructing Basic Blocks from VHDL statements 13

3.2.2 Example : Creating Basic Blocks from VHDL statements 14

3.2.3 Basic Block with Assignment Statements – Operator Use Method 14

3.2.4 Example : Creation of macro nodes 17

3.2.5 Example : Estimating microstates from Macro Nodes 18

3.2.6 Basic Block with a Wait statement 19

3.2.7 Basic Block with a Procedure call 20

3.2.8 Computing Total Performance for Leaf State 20

3.3 Probability-based Analysis of Control Flow Graph Models 22

3.3.1 Determining the Branching Probabilities 22

3.3.2 Determining Node Execution Frequencies, $Freq(N_i)$ 23

3.3.3 Example: Determining Execution Frequencies based on Branch probabilities 24

3.3.4 Applying Probability Based Analysis to Estimation 25

3.4 Estimating performance for a hierarchy of sequential states 25

4 Area Estimation **27**

4.1 Area Estimation Approach 27

4.1.1 Terminology and Notation . . . 28
4.2 Estimating the Number of Registers Required . . . 28
4.2.1 Live Variable Analysis Method . . . 28
4.2.2 Example : Live Variable Analysis . . . 29
4.3 Estimating the Multiplexers required . . . 30
4.3.1 Determining the Paths in the design . . . 30
4.3.2 Example : Determining Paths in the design . . . 31
4.3.3 Mux Estimation for Register Inputs . . . 31
4.3.4 Mux Estimation for Operators - Average Usage Method . . . 32
4.3.5 Comparing Average Case Estimate to the Worst Case . . . 33
4.3.6 Example : Average Usage Mux estimation . . . 33
4.4 Estimating the number of Microstates . . . 34
4.5 Estimating the Control Logic Area . . . 35
4.5.1 Implementing the control in Random Logic . . . 35
4.5.2 Determining the number of active states for Control Lines . . . 36
4.5.3 Estimating the Size and Number of AND / OR Gates . . . 37
4.5.4 Implementing the control using ROM . . . 39
4.6 Calculating the Design Area . . . 40
4.6.1 Computing Datapath Area . . . 40
4.6.2 Computing the Control Logic Area . . . 42
4.6.3 Computing the Memory Area . . . 43
4.6.4 Computing the Total Area . . . 44

5 Results **45**
5.1 Fifth Order Elliptical Filter . . . 45
5.2 Differential Equation Example . . . 45
5.3 Am2901 Microprocessor . . . 47
5.4 Kalman Filter . . . 47

5.5 Am2910 Microprogram Sequencer 47

5.6 Traffic Light Controller 48

6 Conclusion **49**

7 Acknowledgements **49**

8 References **49**

A On-line Manual Pages **51**

A.1 SCESTAREA : Area Estimator 51

A.2 SCESTPERF : Performance Estimator 53

List of Figures

1 Domain of System Level Synthesis 5

2 SpecChart of a Simple Computer 7

3 Design Model for Estimation 8

4 A sample SpecChart 11

5 Computing execution time for a leaf behavior 12

6 Building Basic Blocks from VHDL statements (a) VHDL code segment (b) basic blocks constructed 14

7 Scheduling the basic block into macro nodes for HAL differential equation example 17

8 Estimating microstates in the Basic Block from HAL example: (a) Given functional unit allocations and delays (b) Basic block scheduled into macro nodes (c) Calculating microstates for each macro node and and for the entire block 18

9 Estimating Microstates in Basic Block with Wait Statement (clock cycle = 40 ns) 19

10 Constructing equivalent Control Flow Graph Model for a set of VHDL statements 21

11 Obtaining a set of Linear Equations from The Control Flow Graph Model 24

12 Applying Probability based Analysis for estimation 25

13 Division of design area into memories, datapath blocks and control unit 27

14 Live Variable Analysis applied to the Differential Equation example. 29

15 Determining design paths from the specification (a) VHDL code segment, (b) 3-address statements, (c) paths between components 31

16 Average and Best Case Mux Estimates for $w_i = 12, n_i = 2$ 33

17 2-level AND-OR implementation of Control Logic 35

18 (a) Datapath Layout plan (b) Transistor counts for various components 41

19 Estimation Results 46

20 Comparing Estimated area with the actual layouts 48

List of Tables

1 Average Case Mux Area Estimates Compared with the Worst Case 34

1 Introduction

System level synthesis refers to synthesis at the computer system level. The primary aim is to convert a system's specification into a set of one or more interconnected chips. This involves determining the number of chips necessary to implement the design, distributing the specifications among the chips, finding a well-defined structure for each chip, and creating proper interchip interfaces. The domain of system-level synthesis is shown in Figure 1.

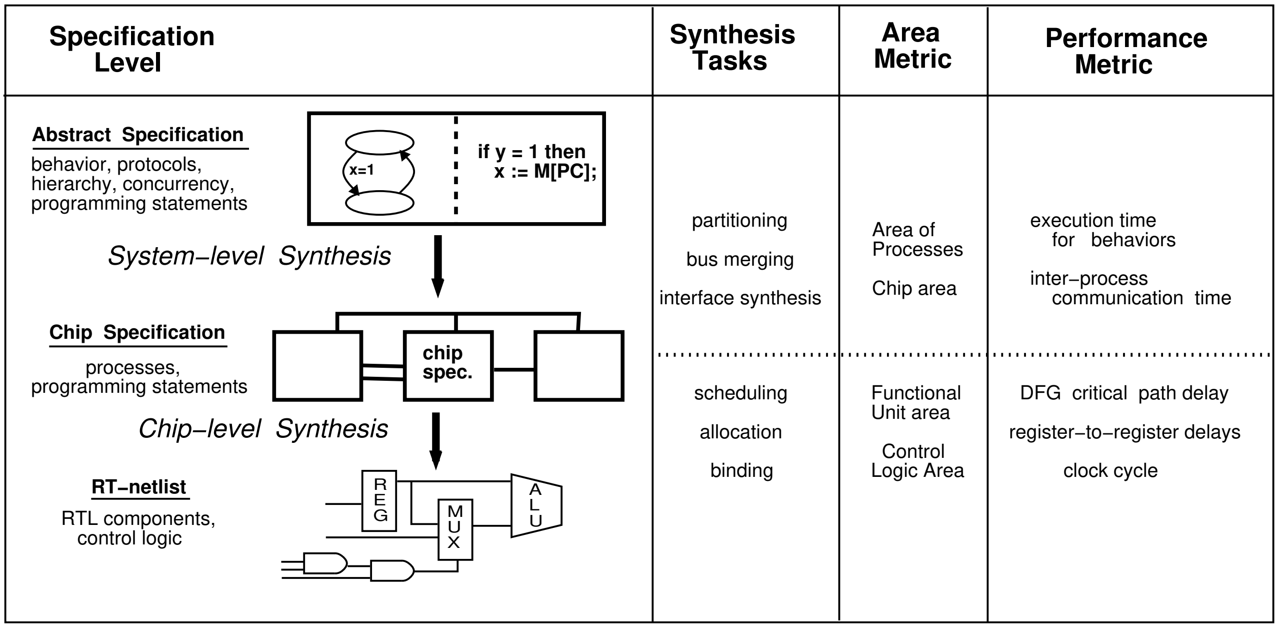


Figure 1: Domain of System Level Synthesis

1.1 Motivation

System-level synthesis involves tasks such as partioning behaviors into chips, grouping variables into memories and partitioning the set of interprocess communication channels into a set of busses. Due to the large search spaces associated with system-level design, it becomes a necessity to have the capability of obtaining estimates of design parameters such as area and performance that will characterize any implementation of the design. In the absence of estimates, the designer cannot make synthesis decisions and perform tradeoffs without actually synthesizing the design and then evaluating the implementation.

Estimates of design parameters assist the designer in two ways:

1. Estimates provide the designer with the capability of exploring large search spaces in a relatively shorter time. The savings in time are evident if the designer were to synthesize the design completely before realizing that an undesirable design decision had been made early in the design cycle.
2. Estimates enable the designer to evaluate the effects or desirability of any design decision. For example the designer might like to know whether a clock cycle of 50 ns will lead to a faster design as compared to a 100 ns clock cycle. Based on the performance estimates available from the estimator, the designer can proceed with other synthesis tasks after selecting a clock cycle which will give him a faster implementation.

The estimators presented in this report is intended to estimate the area and performance of a design which has been captured using the SpecCharts language [1].

For system level synthesis (Figure 1), *cost (area)* metrics are concerned with the area of entire processes or behaviors and consequently with the chip area. Thus, the designer will be able to determine as to how much area will a particular behavior will require or whether a set of given behaviors be assigned to the same chip without violating the area constraints specified for that chip. For any specified behavior (or state), the estimator `scestarea` computes the cost in terms of area of any specified behavior and all its descendant sub-behaviors.

Similarly, the *performance* metrics at this level of design are concerned with the execution time for processes and inter-process communication times as shown in Figure 1. This will enable the designer to obtain answer to questions like - How long does it take to fetch, decode and execute an instruction in a CPU or how would the execution times of two communicating processes be affected if they communicate using a full-handshaking protocol as opposed to using a half-handshake ? For any specified behavior (or state), the estimator `scestperf` computes the cost in terms of the time required for executing that behavior and all its descendant sub-behaviors.

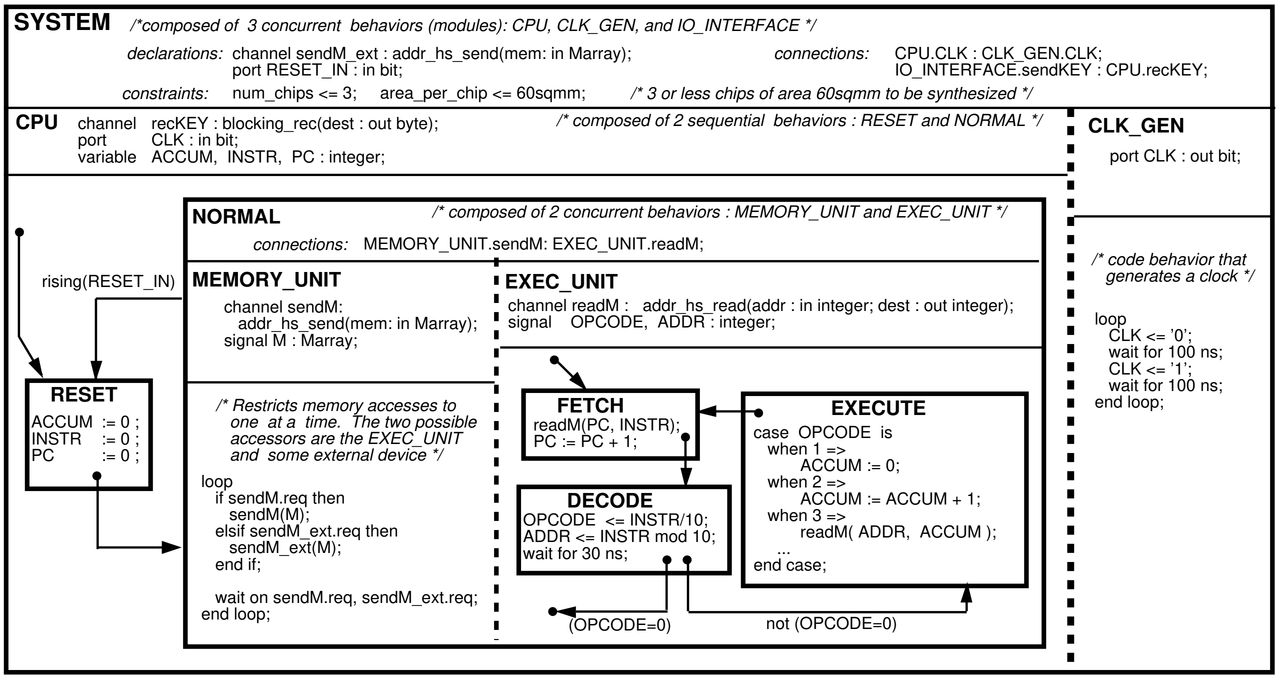


Figure 2: SpecChart of a Simple Computer

1.2 Inputs to the Estimators

The inputs to the area and performance estimators outlined in this report are:

1. A **SpecChart** representing the design for which the area has to be estimated. A SpecChart consists of hierarchical/concurrent state diagrams built on top the VHDL language. An example of a SpecChart for a simple computer is given in Figure 2. Details on the language and its constructs may be found in [1].

2. An **allocation list** consisting of number of available operators of each type to implement the design and their delays. If no allocation list is specified, the estimator will allocate one operator of each class (adder, multiplier etc.) needed in the design.

3. The **clock cycle** which will be used to determine the number of microstates in the design. The clock cycle can be selected by the designer or computed using the Slack Minimization method, presented in [2]. If a clock value is not specified, the estimator will compute the system clock using the Clock Slack Minimization method [2].

It must be mentioned here that the estimators serves as a separate program which simply returns the area/performance estimate for a design, given the three inputs above. It does **not** attempt to optimize the cost or performance nor does it attempt to estimate the clock cycle.

In Section 2, we present the design model that forms the basis for our estimation algorithms. Performance and area estimation are discussed in Section 3 and Section 4 respectively. The results of our estimation our presented in Section 6.

2 Design Model for Estimation

The design model for estimation, shown in Figure 3 is similar to the one presented in [3]. A datapath/control logic model is used for implementing a given design.

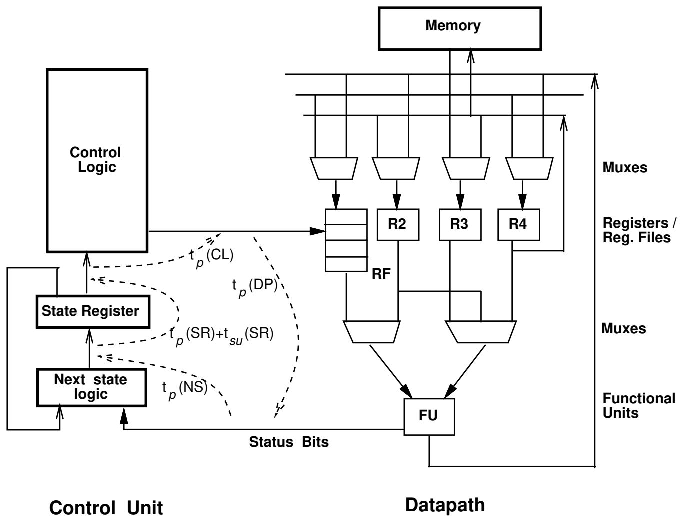


Figure 3: Design Model for Estimation

2.1 Datapath

The datapath consists of storage and functional units. Storage units in the datapath can consist of registers and register files. Since access time for memories can take several cycles and we assume that all accesses to the memory will be through a register (i.e. memory read will write a register in the datapath with the value read, memory write will write to the memory a value stored in a register in the datapath). A two-level multiplexer structure is assumed for the interconnecting the storage and functional units.

A typical register-to-register transfer in the datapath involves operands being read from registers or register files, an operation performed on the operands, and the results stored in another register. The delays associated with this type of register-to-register transfer are the following :

1. Delay through the source register for reading the operands,
2. Delay associated with two levels of multiplexers,
3. Delay of the functional unit executing the operation, and
4. Setup time of the destination register.

Let $t_p(DP)$ represent the total delay in a register-to-register transfer involving a functional unit, FU. Then,

$$t_p(DP) = t_p(Reg) + t_p(Mux) + t_p(FU) + t_p(Mux) + t_{su}(Reg) \quad (1)$$

where,

$t_p(Reg)$ is the delay through the source registers or register files,

$t_p(Mux)$ is the multiplexer delay,

$t_p(FU)$ is the delay of functional unit, and

$t_{su}(Reg)$ is the setup time for destination register or register file.

2.2 Control Unit

The control unit consists of a state register and control logic. The control logic has outputs for each control line needed to control storage, functional and interconnect units. The control logic and the next state logic can be implemented using a ROM or random logic implementation laid out using standard cells.

Let $t_p(CU)$ represent the total delay associated with the control unit. Then,

$$t_p(CU) = t_p(NS) + t_p(SR) + t_{su}(SR) + t_p CL \tag{2}$$

where,

$t_p(NS)$ is the delay through the next state logic,

$t_{su}(SR)$ is the setup time for the state register,

$t_p(SR)$ is the delay through the state register, and

$t_p(CL)$ is the delay through the control logic.

Wire delays are currently ignored for the purpose of performance estimation. We assume that there is a unique functional unit to implement each and every operation type in the given design. The system clock cycle, t_{clk}, is then given by sum of the datapath and control unit delay, as shown in Figure 3, i.e.,

$$t_{clk} = t_p(DP) + t_p(CU) \tag{3}$$

3 Performance Estimation

3.1 Performance Estimation Approach

We adopt a bottom-up strategy for the estimation of performance for a given SpecChart behavior. The overall estimation strategy is illustrated with the aid of Figure 4. below:

1. First, the estimator analyzes the leaf behaviors which are descendents of the behavior for which the performance is being estimated. The VHDL code of these leaf behaviors is used to determine

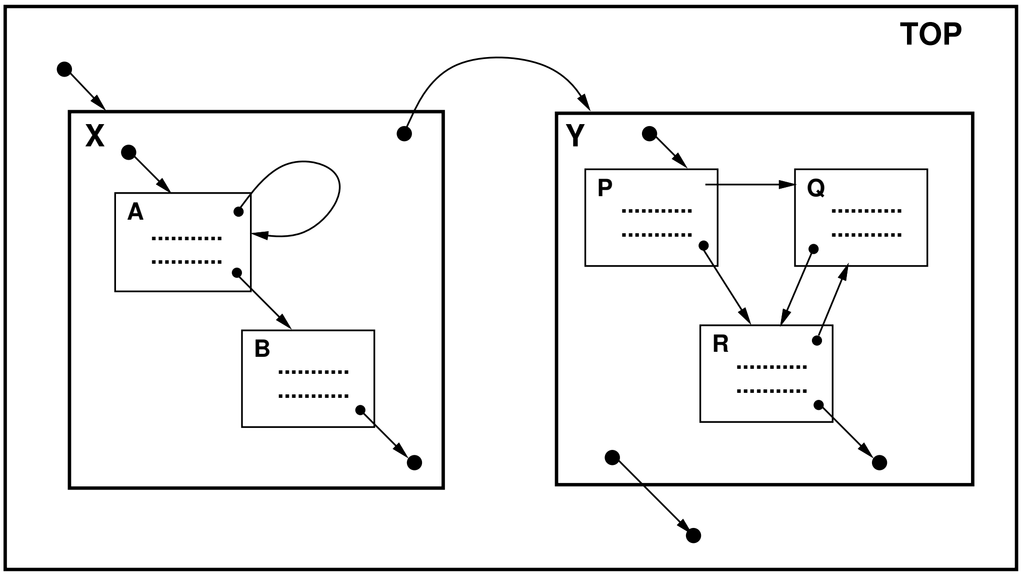


Figure 4: A sample SpecChart

the number of microstates (or clock cycles) each leaf behavior needs to execute. In Figure 4, to estimate the execution time of state TOP, first the execution time for each of the leaf behaviors (A, B, P, Q, R) are first estimated.

2. The leaf behavior execution times and the *transition arc probabilities* are used to determine the execution time of the next higher behavior in the hierarchy. The leaf behavior estimates in Figure 4 are used to estimate the execution time of behavior X and Y.

3. Step 2 is repeated for higher levels of the hierarchy until we have the performance estimate of the given behavior. The estimates of X and Y and the branching probabilities between them are used to determine the execution time of the behavior TOP in Figure 4.

Essentially, the task of estimating the performance of a SpecChart behavior consists of the following sub-tasks:

1. **Computing the execution time for each leaf behavior** : This involves constructing basic blocks for the VHDL code in each leaf behavior, estimating the execution time for each basic block, and combining the estimates for each basic block along with the conditional branch probabilities to obtain the average execution time for the leaf behavior.

2. **Computing the execution time for behaviors in the hierarchy** : At any level of the hierarchy, this involves combining the estimates for each of the sub-behaviors along with the transition probabilities to obtain the average execution time for the behavior

It must be mentioned here that a behavior, TOP, for which performance is being estimated, may have a descendant sub-behavior which does not have a *stop dot*. In this case the behavior TOP will never finish executing and consequently the execution time returned for that behavior is an arbitrarily large number.

---

procedure ESTIMATE_LEAF_BEHAVIOR_PERFORMANCE (L : leaf behavior)
return microstates is

begin

Compute, B, the set of basic blocks in VHDL code

for (all $b \in B$) do
  Compute, $ExecTime(b)$, the number of microstates required by block b
endfor

for (all $b \in B$) do
  Compute, $Freq(b)$, the average frequency of execution for block b
endfor

Compute the total execution time for leaf behavior L,

$$ExecTime(L) = \sum_{for\ all\ b \in B} (ExecTime(b) \times Freq(b))$$

return($ExecTime(L)$);

end

Figure 5: Computing execution time for a leaf behavior

3.2 Computing Performance for Leaf States

A *microstate* is defined as the set of operations that can be executed in exactly one clock. Given a leaf behavior with VHDL sequential statements, we need to estimate the average number of microstates required by a given set of sequential VHDL statements to execute. Since exact scheduling is computationally expensive to perform, we employ the *Operator Use* method for microstate estimation, *based on the clock cycle and resource constraints.* The steps required for estimating the performance of leaf states is summarized in Figure 5.

3.2.1 Constructing Basic Blocks from VHDL statements

We first divide the given VHDL code segment into *basic blocks.* A *basic block* [4] is a sequence of consecutive VHDL statements in which flow of control enters at the beginning and leaves at the end without halting or the possibility of branching, except at the end. To determine the basic blocks, we first determine the set of *leaders*, the first statement of a basic block. The rules we use to determine leader statements are the following -

1. The first statement of the code segment is a leader.
2. All *wait* statements and procedure calls are leaders.
3. Any statement which is the target of a conditional statement (if, loop, case) is a leader. The target of a conditional statement is any statement to where control could possibly be transferred on evaluating the condition.
4. Any statement that immediately follows a conditional statement, wait statement, or a procedure call is a leader.

For each leader determined above, its basic block consists of the leader and all statements upto but *not including* the next leader (or the end of the given VHDL code segment). A basic block will then contain one of the following:

1. a set of assignment statements, or
2. a single wait statement, or
3. a single procedure call.

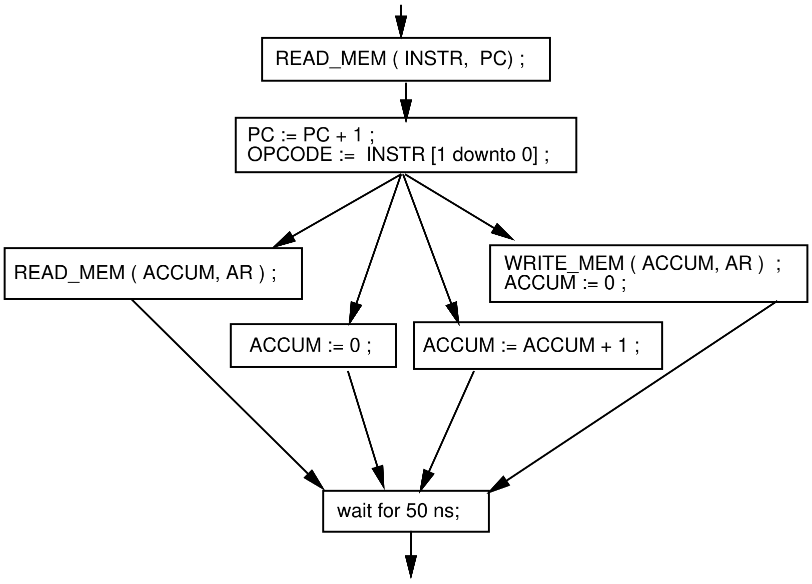
Once basic blocks have been determined, we can estimate the number of microstates required by each basic block to execute. The estimation procedure depends on the organization of the basic block.

```
1 ->    READ_MEM ( INSTR,  PC) ;
4 ->    PC := PC + 1 ;
        OPCODE :=  INSTR [1 downto 0] ;

        if (OPCODE = "00") then
3 ->       ACCUM := 0 ;
        elsif  (OPCODE = "01") then
3 ->       ACCUM := ACCUM + 1 ;
        elsif  (OPCODE = "10") then
3 ->       READ_MEM (ACCUM,  AR) ;
        elsif  (OPCODE = "11") then
3 ->       WRITE_MEM (ACCUM, AR) ;
           ACCUM := 0
        end if;

2 ->    wait for 50 ns;
```

(a)



(b)

Figure 6: Building Basic Blocks from VHDL statements (a) VHDL code segment (b) basic blocks constructed

3.2.2 Example : Creating Basic Blocks from VHDL statements

Figure 6 shows how the basic blocks can be constructed from a given set of sequential VHDL statements. In Figure 6(a), the leaders are denoted by horizontal arrows along with the corresponding rule number used to determine the leader. Figure 6(b) shows the basic block structure of the VHDL code.

3.2.3 Basic Block with Assignment Statements – Operator Use Method

In the case where the basic block consists of a set of assignment statements only, we apply the following steps to determine the number of microstates required by the basic block for execution:

1. **Creating Statement nodes**: The basic block is organized as a collection of statement nodes. A *statement node* is defined as a node which has zero, one or more statements in it. Initially each statement in the basic block is placed in a separate statement node .
2. **Scheduling statements into Macro nodes** : The statements are first merged together using the ASAP (as-soon-as-possible schedule) scheduling so as to achieve the earliest completion time.

Statement B is said to be *dependent* on statement A if the result computed by statement A (the variable on the left hand side of statement A) is used in statement B. Thus, statement A should be scheduled before B. As a result of scheduling, statements are grouped together in nodes provided that no dependencies which existed in the original basic block are violated. We will refer to nodes thus obtained (with several statements in them) as *macro nodes*.

All the statements in the macro node could possibly be executed concurrently because they are independent of each other. It should be mentioned here that the statements are grouped together *independent* of the clock rate or the resources allocated to the design. We ignore the occurrence of an *after* clause in signal assignment statements.

3. **Computing microstates required by each macro node**: Given an operator allocation, clock rate, and the operator delay (all of which are inputs to the estimator), the number of microstates for *each* macro node can be estimated by the **Operator Use method**.

Non-pipelined Functional Units

The number of microstates required by all the operations in any macro node is given by :

$$MS_{mn(j)} = MAX_{all\ operator\ types\ t_i}(\ \lceil u(t_i) \div n(t_i) \rceil \times ClkDelay(t_i) \ ) \tag{4}$$

where,

$MS_{mn(j)}$ is the number of microstates in macro node $mn(j)$.
$u(t_i)$ is the number of *uses* or occurrences of operator type t_i in the macro node.
$n(t_i)$ is the number of functional units of type t_i allocated to the design.
$ClkDelay(t_i)$ is the delay (in clocks) of the functional unit implementing operation type t_i.

The idea behind the Operator Use Method is simple. If there are $u(t_i)$ uses (i.e. occurrences) of operator type t_i in a macro node, and only $n(t_i)$ operators of that type are available to implement the design, we need at least $(u(t_i) \div n(t_i))$ *states* or control steps to perform the computation. Each stage represents the delay of the operator in terms of the number of clock cycles. For each operator used in the macro node, the total delay given by Equation 4 is calculated. The maximum of all these clock delays gives the minimum number of microstates needed to implement that macro node.

Pipelined Functional Units

Consider an operation of type t_i which is implemented by a pipelined functional unit. Let $stages(t_i)$ represent the number of pipeline stages in the functional unit. If there are $u(t_i)$ uses of operator type t_i in a macro node, then each of the $n(t_i)$ functional units allocated will on the average perform $\lceil u(t_i) \div n(t_i) \rceil$ operations. The functional unit pipeline will produce the result of the first operation after $stages(t_i)$ steps, and the results of the remaining $(\lceil u(t_i) \div n(t_i) \rceil - 1)$ operations will require an additional $(\lceil u(t_i) \div n(t_i) \rceil - 1)$ steps. Thus, the total number of microstates required for execution by a macro node is given by:

$$MS_{mn(j)} = MAX_{all\ operator\ types\ t_i} \left[\, (\lceil u(t_i) \div n(t_i) \rceil - 1 + stages(t_i)) \times ClkDelay(t_i) \, \right] \qquad (5)$$

where,

$MS_{mn(j)}$ is the number of microstates in macro node $mn(j)$.
$u(t_i)$ is the number of *uses* or occurrences of operator type t_i in the macro node.
$n(t_i)$ is the number of functional units allocated to the design.
$stages(t_i)$ is the no. of pipeline stages in functional unit implementing operation type t_i.
$ClkDelay(t_i)$ is the delay (in clocks) of each *stage* of pipelined functional unit implementing operation type t_i.

It is interesting to note that Equation 5 reduces to Equation 4 if we were to consider each non-pipelined functional unit as being a *single-stage* pipelined functional unit ($stages(t_i) = 1$).

Memory Accesses in basic blocks

Memory accesses within the basic block also contribute to the microstates required for execution of the statements in the basic block. We can extend the Operator Use method for memories by treating the memory as an operator. Accesses to the memory are then identical to the "uses" of an operator, the number of ports available for accessing the memory are equivalent to the number of functional units allocated for implementing the design and the memory access time is equivalent to the delay of the functional unit.

Thus, for each memory M accessed within a basic block, define an operator t_M such that,

$u(t_M)$ = number of accesses to M within the basic block.
$n(t_M)$ = number of access ports available for M.
$ClkDelay(t_M)$ = memory access delay (in clocks).

We can now treat memory M as any other operator and apply Equation 4 to incorporate memory accesses into the basic block microstates calculation.

4. **Computing microstates for entire Basic Block**: The clock cycles needed for each macro node are added to give the total number of microstates needed to execute the basic block i.e.

$$MS_{BasicBlock} = \sum_{all\ mn(j)\ \in\ BasicBlock} MS_{mn(j)} \tag{6}$$

where,

$MS_{BasicBlock}$ is the number of microstates in the basic block.
$MS_{mn(j)}$ is the number of microstates in macro node $mn(j)$.

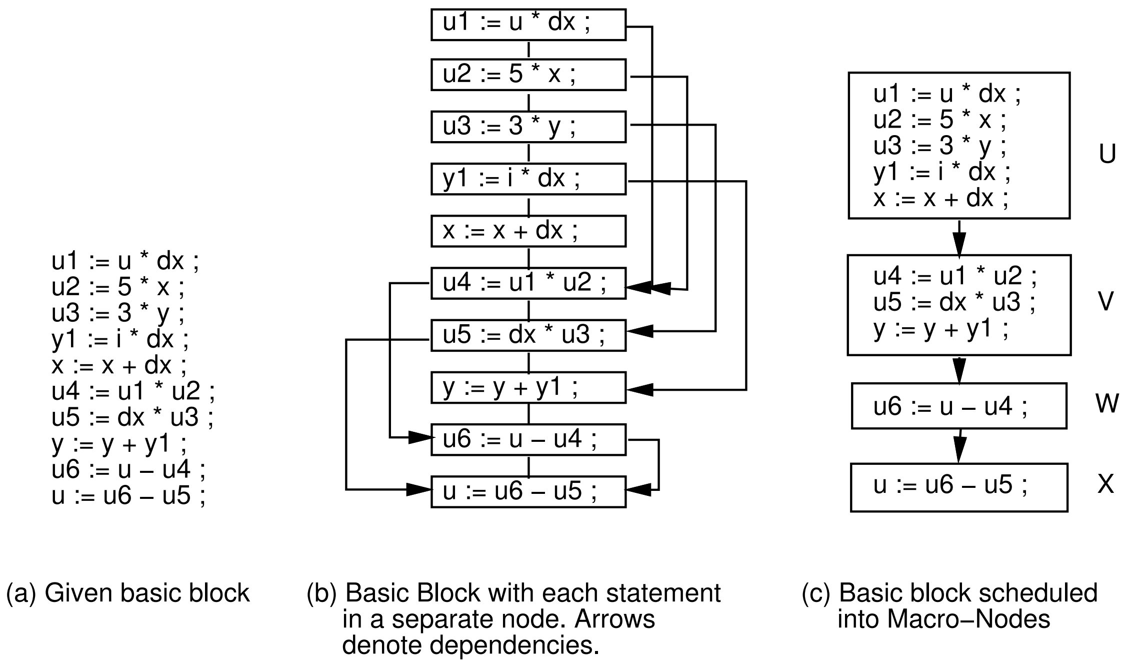


Figure 7: Scheduling the basic block into macro nodes for HAL differential equation example

3.2.4 Example : Creation of macro nodes

Figure 7 shows how the macro nodes can be formed for a given basic block consisting of assignment statements from the Differential Equation example [5]. The basic block is given in Figure 7(a). The corresponding basic block statement nodes are shown in Figure 7(b), where each statement is placed in separate statement node. Figure 7(c) shows the result of ASAP scheduling using dependency analysis. The statement ``u2 := 5 * dx'' is not dependent on the assignment to the variable $u1$ in

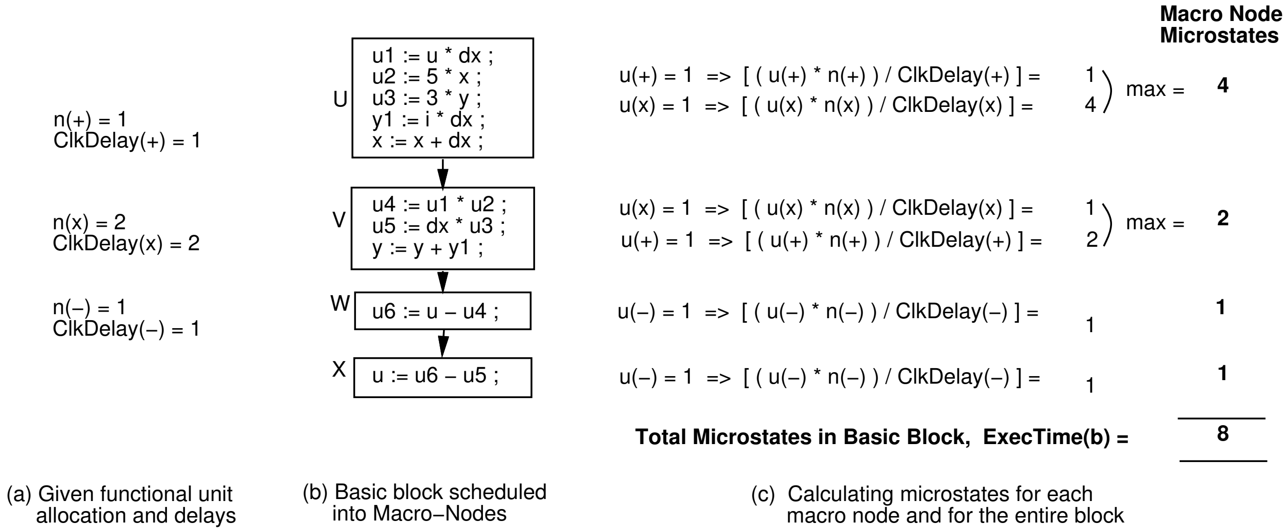


Figure 8: Estimating microstates in the Basic Block from HAL example: (a) Given functional unit allocations and delays (b) Basic block scheduled into macro nodes (c) Calculating microstates for each macro node and and for the entire block

the statement ``u1 := u * dx''. Hence it is moved up to macro node U. The scheduled basic block has four macro nodes U, V, W and X. While the example of Figure 7 has only binary expressions, the same method can be applied to statements with more complex expressions to create macro nodes.

3.2.5 Example : Estimating microstates from Macro Nodes

The scheduled basic block for the HAL example is shown in Figure 8. Figure 8(a) gives the allocation and clock delays of the operators used. The functional units in this example are non-pipelined. In Figure 8(b), the HAL statements have been grouped together into four macro nodes, such that they contain statements which are not dependent on each other. Figure 8(c) shows the microstate calculation for each macro node. For example, in macro node U, there are four ($u(x) = 4$) multiply operations and 2 multipliers ($n(x) = 2$) are allocated to the design. Thus, each multiplier performs 2 multiplications, and since multiplication takes 2 microstates ($ClkDelay(x) = 2$), the macro node requires at least 4 microstates to perform multiplications. The total number of microstates in the Differential Equation example is 8.

3.2.6 Basic Block with a Wait statement

The *wait* statement in SpecCharts has the same semantics as that in VHDL. In case the basic block has a wait statement, estimating the number of microstates for which execution will be suspended depends on the type of the wait statement.

1. **Wait :** In the absence of a timeout clause, the execution time is set to a very large number like MAXINT, the largest integer supported by the host machine.

2. **Wait on** $S_1, ...S_n$ **until** $C_1, ...C_2$ **for T ns** : The execution time for a wait statement with only a *timeout clause* (i.e. *for* clause) is equivalent to the smallest multiple of the clock period higher than T. The effect of conditions (e.g. C1) and an event on signals in the sensitivity list (e.g. S1) are ignored.

3. **Wait on** $S_1, ...S_n$ **until** $C_1, ...C_2$ i.e wait statement without a timeout clause. If the wait statement has a sensitivity list or condition clause but no timeout clause then the wait statement requires at least one control step to be executed. The effect of conditions (e.g. C1) and an event on signals in the sensitivity list (e.g. S1) are ignored.

Examples of estimations for basic blocks with wait statement are presented in Figure 9.

Wait Statement	**Estimate of time spent Waiting**
wait for 60 ns ;	60 ns wait ≡ 2 clock periods ≡ **80 ns**
wait on RD, WR for 90 ns ;	90 ns wait ≡ 3 clock periods ≡ **120 ns**
wait until (A=2) or not(P'stable);	No Timeout clause Estimated Wait = 1 clock period = 40 ns
wait ;	Indefinite Wait ≡ MAXINT clock periods

Figure 9: Estimating Microstates in Basic Block with Wait Statement (clock cycle = 40 ns)

3.2.7 Basic Block with a Procedure call

The third type of basic block is one which has a single procedure call. The execution time for the procedure can be determined by treating the body of the procedure declaration as a leaf state. This time is then used as the execution time of the basic block which contained the procedure call.

3.2.8 Computing Total Performance for Leaf State

Once the number of microstates required by each basic block have been computed, we can compute the overall performance for the entire leaf state VHDL code.

Leaf State without Conditional Branching

If the leaf state VHDL code has no conditional branching, each basic block of statements is executed exactly once, and thus the total number of microstates required by the leaf state for execution is the sum of the number of leaf states required by each basic block.

$$ExecTime(L) = \sum_{for\ all\ b \in B} (ExecTime(b)) \tag{7}$$

where,

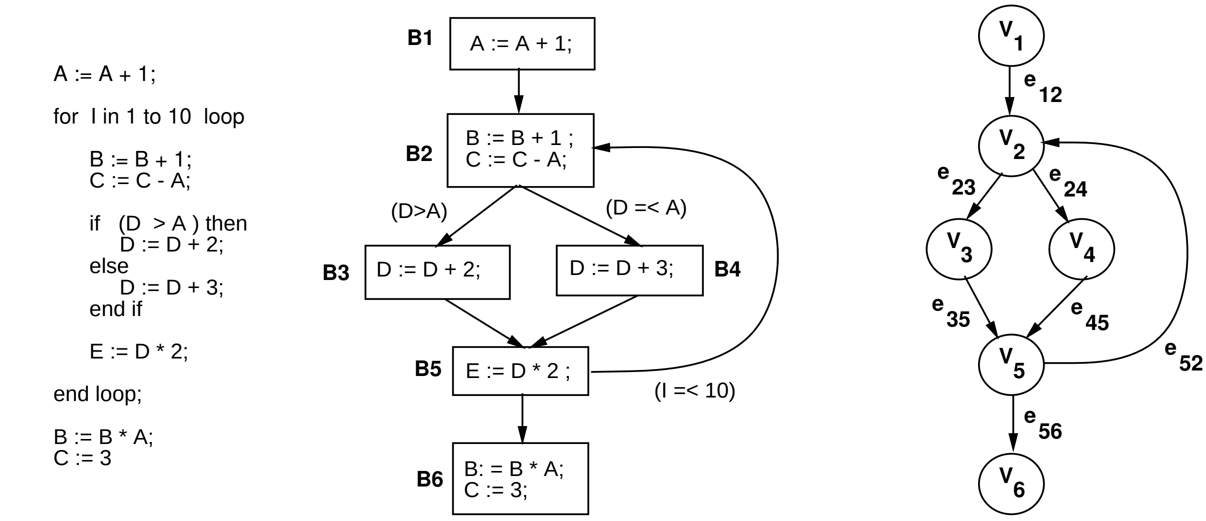
$ExecTime(L)$ is the total execution time in terms of microstates required by a leaf state, L.

$ExecTime(b)$ is the execution time required by basic block $b \in B$.

B is the set of basic blocks in the leaf state.

Leaf State with Conditional Branching

If the VHDL code contains conditional branching, then we need to weigh the number of microstates required by the constituent basic block by its *frequency of execution* to calculate the execution time required by the entire code segment. The frequency of execution, $Freq(b)$, for a basic block b is defined as the number of times that the basic block will be executed on the average, during the execution of the VHDL code segment to which it belongs.



(a) Leaf State VHDL code (b) Basic Blocks for VHDL code (c) Equivalent Control Flow Graph Model

Figure 10: Constructing equivalent Control Flow Graph Model for a set of VHDL statements

In order to calculate the frequency of execution for each basic block, we must first create an equivalent *control flow* graph model for the basic blocks in the leaf state. Let $G = (V, E)$ be a graph, where V is the set of vertices v_i, and E is the set of *directed* edges e_{ij} connecting vertex v_i to v_j.

For each basic block b_i in the VHDL code, there exists a corresponding vertex v_i in the graph G. The target block of a conditional statement (if, case or loop) is any basic block to where control could possibly be transferred on evaluating the condition. For any basic block b_j which is the target of a conditional statement following basic block b_i, the set E has a directed edge e_{ij} from vertex v_i to vertex v_j in G. Figure 10 shows how such a graph model can be constructed from a set leaf state VHDL statements.

After the control flow graph model has been constructed for the basic blocks, we use the **Probability-based Analysis method** (explained in the next section) method for computing execution frequency, $Freq(b)$, for each basic block b in the VHDL code.

Having determined the execution frequencies of each basic block, we can now calculate the total number of microstates required by the leaf state.

$$ExecTimeL = \sum_{for\ all\ b \in B} (ExecTime(b) \times Freq(b)) \quad (8)$$

where:

$ExecTime(L)$ is the total execution time in terms of microstates required by a leaf state, L.

$ExecTime(b)$ is the execution time required by basic block $b \in B$.

B is the set of basic blocks in the leaf state.

$Freq(b)$ is the frequency of execution of the basic block b, determined by the Probability based Analysis method.

3.3 Probability-based Analysis of Control Flow Graph Models

Given a graph which represents control flow in a portion of the design, we wish to determine the execution frequencies of each of its nodes based on the branch probabilities. By determining the execution frequencies of the nodes, we can obtain a lot of useful information about the design by associating with each node in the graph, *weights* representing some aspect of the design.

For example, if each node weight represents the execution time of a basic block derived from the VHDL code of a leaf state, then we can use the execution frequency of the nodes to determine the total execution time for the leaf state by taking a weighted sum of of these two quantities. If the graph represents a set of sequential substates and each node weight represents the number of time a variable is accessed by a state, then we can use the execution frequencies of the states to determine the total number of time the variable is accessed by the parent state.

3.3.1 Determining the Branching Probabilities

Transition arcs between the nodes each have a probability assigned to them. Arc probabilities may be determined in the following ways –

1. **Equal Probabilities** : In case of branching, we assign equal probabilities to all the arcs leading out of the basic block. Thus, if there are three arcs emerging from a node, all of them are assigned

a probability of 1/3.

2. **User Defined Probabilities** : The user may be requested to specify the branch probabilities either in the SpecChart description using annotations or interactively at runtime .

3. **Simulation based Probabilities** : For every simulation performed, a record is kept of the branches that were taken. From this observed behavior, the probabilities of the branches can be derived.

3.3.2 Determining Node Execution Frequencies, $Freq(N_i)$

The execution frequency of a node is defined as the number of times on the average that the node will be executed in a single execution of the graph. We analyze the transition arc probabilities between the nodes to determine execution frequencies. This is done according to the steps shown below –-

1. Determine the branch probabilities using one of the methods outlined above.

2. A *start node*, S, preceding the first node in the graph, is added. Its execution frequency, $Freq(S)$ is set to 1 since this node is executed exactly once whenever the control flow graph is executed.

3. The execution frequency for any node Y depends on the weighted execution frequency of all its immediate predecessor nodes. The execution frequency for each predecessor node X is multiplied by the branch probability of the edge between X and Y, Prob(e_{xy}). For each node in the graph, we first formulate the equation for the node execution frequency.

$$Freq(N_j) = \sum_{all\ predecessor\ nodes\ N_i} Freq(N_i) \times Prob(e_{ij}) \tag{9}$$

4. We then solve the set of equations formulated in step 3 to obtain the individual execution frequencies using any of a variety of methods : Gaussian Elimination, LU decomposition, Chomsky's method. Currently, the performance estimator uses the Gaussian Elimination method.

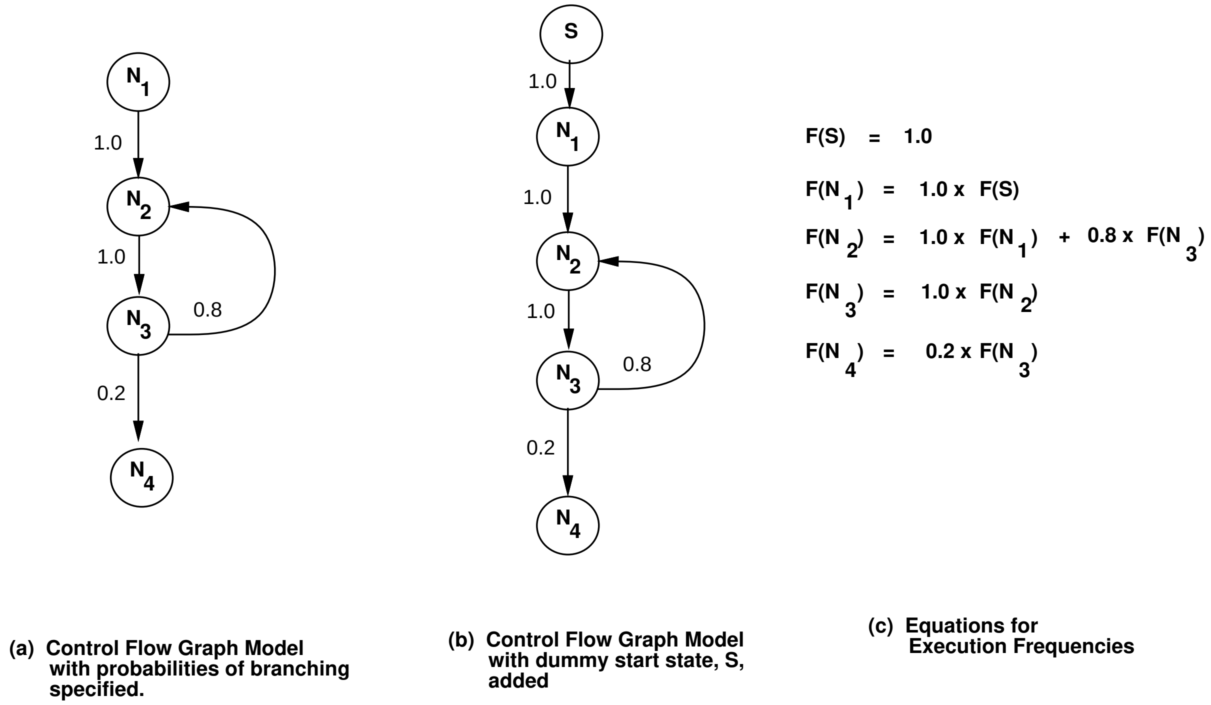


Figure 11: Obtaining a set of Linear Equations from The Control Flow Graph Model

3.3.3 Example: Determining Execution Frequencies based on Branch probabilities

In Figure 11(a), a control flow graph with the branch probabilities is shown. Figure 11(b) shows the same graph with the added dummy state, S. Figure 11(c) shows how the equations for the execution frequency of each node are formulated. As an example, consider node N_2. It has two predecessor nodes - N_1 and N_3. The probabilities of the edges e_{12} and e_{32} are 1.0 and 0.8 respectively. Thus the equation for the execution frequency for N_2 is :

$$F(N_2) = 1.0 \times F(N_1) \; + \; 0.8 \times F(N_3) \tag{10}$$

Solving the equations in Figure 11c yields the following values for execution frequency :

$$F(N_1) = 1, \quad F(N_2) = 5, \quad F(N_3) = 5, \quad F(N_4) = 1$$

These are the expected execution frequencies returned after Probability based analysis of the control flow graph.

3.3.4 Applying Probability Based Analysis to Estimation

The nodes in the flow-graph can represent the basic blocks of a leaf state VHDL code or sequential substates of a state. By associating different quantities with the weight of each node, the Probability based Analysis method can be used to provide various useful performance related estimates. These are summarized in Figure 12.

Nodes of Flow–Graph	Node Weight	Estimate Obtained after applying Probability based Analysis
Basic blocks of VHDL code of Leaf State	Execution time of Basic block	Total execution time of a Leaf state
Sequential substates	Execution Time of a substate	Total execution time of a parent state
Sequential substates	Number of accesses to variable by substate	Total accesses to variable by parent state
Sequential substates	Number of accesses to procedure by substate	Total accesses to procedure by parent state
Sequential substates	Number of accesses to channel by substate	Total accesses to channel by parent state

Figure 12: Applying Probability based Analysis for estimation

For example, if the flow-graph represents the substates of a state and the weights associated with each node are the number of times a substate accesses a communication channel, then after applying probability based analysis and taking a weighted sum (product of node weights and execution frequencies), we can obtain the total number of accesses made by the parent state to the communication channels. This channel access frequency estimate is used while determining which channels can be merged into buses.

3.4 Estimating performance for a hierarchy of sequential states

Once execution times have been estimated for each of the leaf states as shown above, we must now merge the performance estimates of the leaf states to yield the the performance estimate of the next higher state in the hierarchy. This process is repeated until we determine the performance estimate of the topmost state in the SpecChart.

The approach adopted is similar to that of merging the performance estimates of basic blocks to obtain performance estimates for the leaf states. At any level of the hierarchy, we will first determine the performance estimates of the children states and then use probability based analysis to combine these estimates to determine the performance estimate of the parent state. Thus, in Figure 4 to estimate the execution time of state TOP, first the execution time for each of the leaf states (A, B, P, Q, R) are first estimated as outlined in Section 3.2. The estimates of A and B are used to estimate the performance of state X. The estimates of P, Q and R are used to compute the performance of state Y. Finally, the estimates obtained for X and Y are used to determine the performance of state TOP.

Once again, to estimate the performance for a state, $State_{parent}$, we create a control flow graph model for its children states whose performance estimates are already known. Let $G = (V, E)$ be a graph, where V is the set of vertices v_i, and E the set of *directed* edges e_{ij} connecting vertex v_i to v_j.

For each of the substates, Sub_i, of $State_{parent}$, there exists a corresponding vertex v_i in the graph G. For every transition arc between the two substates Sub_i and Sub_j, the set E has a directed edge e_{ij} from vertex v_i to vertex v_j in G. After the control flow graph model has been constructed for the substates, we can use the *Probability-based Analysis* method for computing execution frequencies for each substate Sub_i. Details of this method were presented in Section 3.3.

The total execution time for the parent state, $ExecTime(State_{parent})$ in terms of microstates, is:

$$ExecTime(State_{parent}) = \sum_{i=1}^{N_{ss}} [\ ExecTime(Sub_i) \times Freq(Sub_i)\ ] \tag{11}$$

where,

$ExecTime(Sub_i)$ is the execution time of substate Sub_i.

$Freq(Sub_i)$ is the execution frequency of Sub_i.

N_{ss} is the number of substates of $State_{parent}$

In case a state at any level of the hierarchy has concurrent substates, then the execution time of that state is same as the maximum execution time amongst all the concurrent substates. *EI* (exit-immediately) arcs, are assigned a zero probability during probability based analysis since they represent external events like resets or power failures which will occur infrequently.

4 Area Estimation

4.1 Area Estimation Approach

The strategy for area estimation is based on dividing the total design area into the following components :

1. *Datapath* blocks, which consist of registers, function units such as adders and interconnect units such as multiplexers. The datapath blocks assumed to consist of a stack of bit-sliced components.

2. *Control Unit*, which controls the data transfers within the datapath components. The control unit could be a random logic implementation consisting of two level AND-OR gates or as a ROM.

3. *Memories* which are used to implement the arrays in the design specification.

The Figure 13 shows the various blocks in the design. The components in the datapath are determined first. This is then followed by an estimation of the control logic and finally, the size of the memory components.

The size of the components is measured by the associated transistor costs as obtained by examining a component library. The total number of transistors in the design is the sum of the transistors in the datapath, control logic and memories.

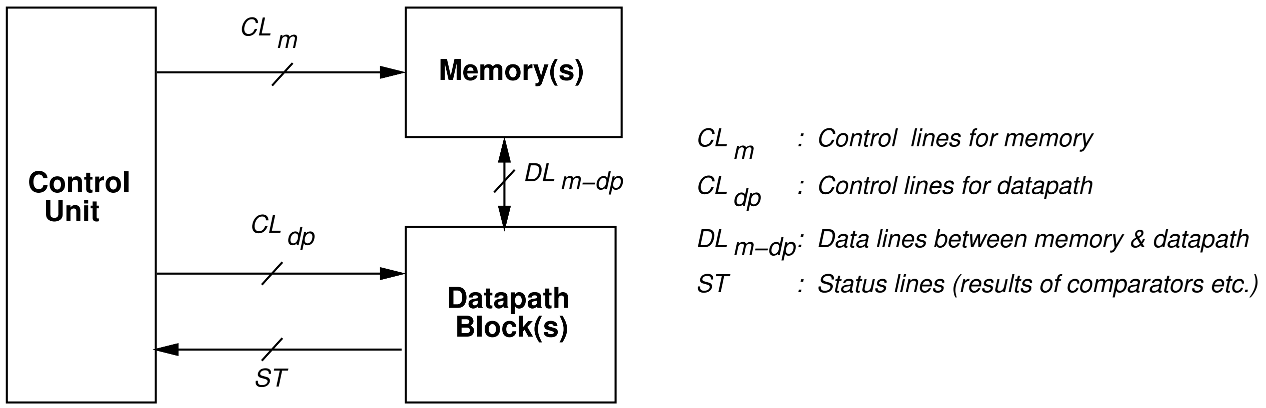


Figure 13: Division of design area into memories, datapath blocks and control unit

4.1.1 Terminology and Notation

A few terms which are commonly used throughout this document are outlined here.

(a) Though the input to the estimator is a SpecChart, we have used the terms set of sequential statements, program, code and pseudo-code interchangeably when describing examples.

(b) The *writes* to a variable or a register is the same as its *assignment* and *definition* . All these terms refer to the existence of the variable on the left hand side of an assignment statement.

(c) The *reads* from a variable or a register is the same as its *use*. All these terms refer to the existence of the variable on the right hand side of an assignment statement.

4.2 Estimating the Number of Registers Required

We can determine the number of registers required for implementing the SpecChart by performing *live variable analysis*. Since the SpecChart for which we are estimating the area may consist of a hierarchy of sequential substates, it is difficult to apply live variable analysis directly to the SpecChart. The area estimator first flattens the SpecChart hierarchy and generates a leaf state containing the VHDL code for the flattened SpecChart. Live variable analysis is then applied to this VHDL code to determine the registers needed to implement the SpecChart.

4.2.1 Live Variable Analysis Method

This method examines the *life* of a variable in terms of its definitions and uses. A variable is said to be *live* over a set of sequential statements from the point of an assignment to it (i.e. definition) to the last read (i.e last use before the next write) from it. Thus in Figure 14, variable $u3$ is written to in statement 4, and is last accessed in statement 8. Thus it is said to be live over statements 4 through 8 and this is indicated by the vertical line drawn in the column for u3. Since variable x is written to in statement 6, but last used in the statement 6 of the next iteration , it is considered to be live over the entire set of statements. This method can be summarized as follows -

1. Determine the *lifetimes* of all the variables in the given program.

2. Represent the lifetime of each variable as a vertical line.

3. Find the maximum number of overlapping lifetimes by determining the maximum number of vertical lines, N, intersecting with any horizontal cutline.

4. Then N is the minimal number of registers needed for implementing the design. This follows from the fact that if N variables are still live, we need to store them in N registers for further *use*. *Register sharing* is used wherein several of the variables may be mapped to the same register.

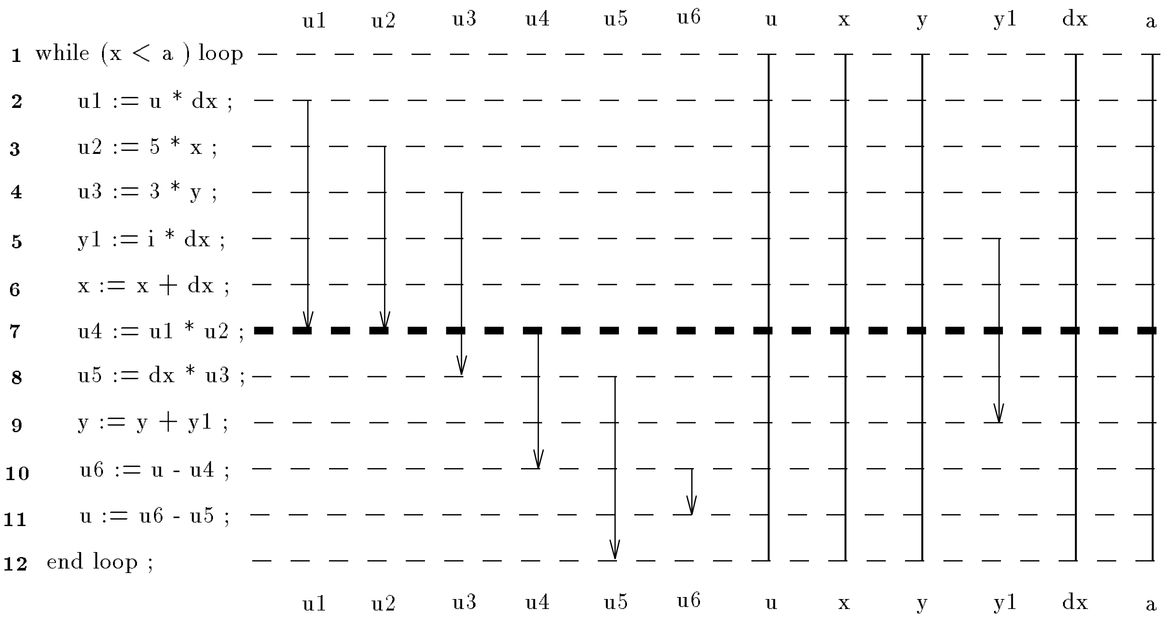


Figure 14: Live Variable Analysis applied to the Differential Equation example.

4.2.2 Example : Live Variable Analysis

Figure 14 shows the application of live variable analysis to the Differential Equation pseudo code. Originally there are 12 variables in the set of statements. The lifetimes for each variable is shown by the vertical lines. The bold dotted line at statement 7 has the largest number of vertical lines (i.e., 9) reaching or crossing it. Thus, we can implement the design using only 9 registers. Some of the variables are mapped to the same register. A possible implementation would assign registers to the following groups of variables - (x), (a), (u), (dx), (y), ($u3$), ($u1,\ u4$), ($u2,\ u5$) and ($u6,\ y1$).

The area of each register depends on the number of bits of the variables mapped to that register. The area of array variable is a function of the number of words in the memory and the number of bits in each word.

4.3 Estimating the Multiplexers required

The number of muxes needed for a datapath component can be determined by examining the paths determined earlier for the design. The size of the mux required for a component depends on the number of input paths to that component.

4.3.1 Determining the Paths in the design

The first step in datapath area estimation is determining the *unique paths between components* in the design. This will enable us to determine the size and number of multiplexers required in the design and the interconnection between the components. Initially, it is assumed that each variable is implemented as a separate register. As mentioned earlier, all data read (written) from (to) memories is stored in a register. Thus, there are no direct paths from the memories to operators like adders and multipliers.

Given a SpecChart, we determine all the paths by examining the VHDL code in all the leaf states. There are be several types of paths in the datapath:

1. register to register (RR path)
2. register to operator (RO path)
3. operator to register (OR path)
4. register to memory (RM path)
5. memory to register (MR path)

A path P, from component S to component T, is represented as $(T < S)$, where: T is the *target component* of path P, S is the *source component* and $<$ shows the direction of the path .

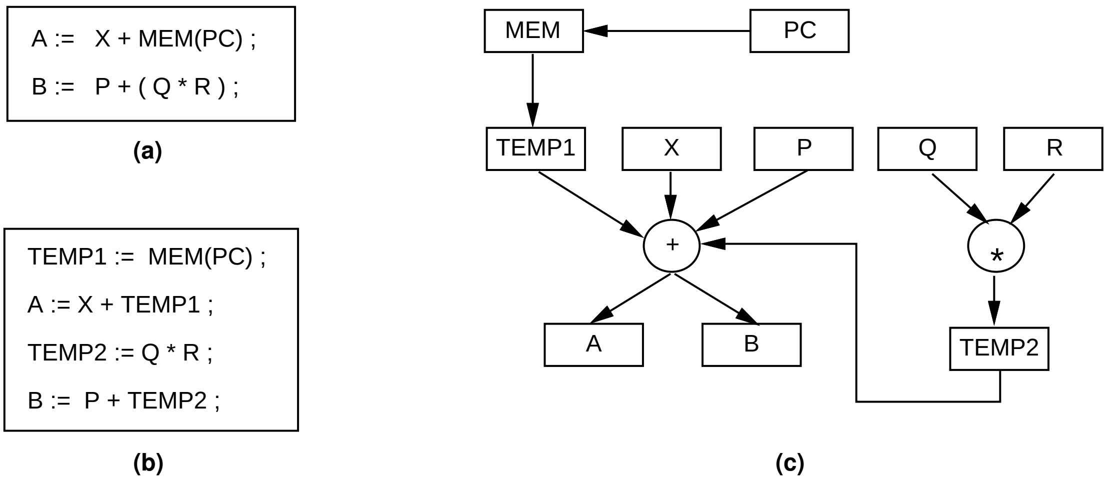


Figure 15: Determining design paths from the specification (a) VHDL code segment, (b) 3-address statements, (c) paths between components

4.3.2 Example : Determining Paths in the design

The example of Figure 15 illustrates how the paths in a design can be determined from a given VHDL code segment. Consider the sequential statements given in Figure 15(a). For each statement, we first generate 3-address statements in which at most one operation exists on the right hand side. The 3-address statements for the code segment is given in Figure 15(b). All memory accesses are performed through intermediate registers. Thus, in Figure 15(b), $MEM(PC)$ is first read into register $TEMP1$. The arrows in Figure 15(c) represent all the paths for the code segment. Thus, the VHDL code segment has the following paths:

1. register to operator : $(+ < TEMP1), (+ < X), (+ < P), (+ < TEMP2), (* < Q), (* < R)$
2. operator to register : $(A < +), (B < +), (TEMP2 < *)$
3. register to memory : $(MEM < PC)$
4. memory to register : $(TEMP1 < MEM)$

4.3.3 Mux Estimation for Register Inputs

For registers, the size of the mux depends on the number of unique sources that have a *write path* to it. The write paths to all variables mapped to the same register (after live variable analysis) are used to determine the size of the mux required for inputs to that register. If there are a total of n paths that have as a target, variables mapped to a register R, then a n-to-1 mux is needed for register R.

In the Differential Equation example, the live variable analysis of Section 4.2.2 mapped the variables $u6$ and $y1$ to the same register. Each of these two variables have a single path leading to them from the multiplier and subtractor respectively. Consequently, the register implementing these two variables will require a 2-to-1 mux.

4.3.4 Mux Estimation for Operators - Average Usage Method

In case there is only one operator of a given type, allocated to the design, the mux size estimation is identical to the procedure outlined above for registers (the number of write paths to the operator are split into two equal sets, each of which is connected through a mux to one of the inputs of the functional unit implementing that operation). If several operators of the same type are present in the design, calculation of the size and number of muxes is not as simple. This is due to the fact, that we do not wish to perform *binding* of the allocated operators to operator nodes in the data flow graph of the design. Instead, will use the Average Usage Mux Estimation Method for operators.

As the name suggests, the Average Usage method assumes that all unique inputs to an operator of a given type are shared equally amongst all operators of that type allocated to the design. Furthermore, once the average number of inputs for a single operator are determined, these are assumed to be divided equally into two sets each of which is connected to one of the two functional unit inputs. The size of the mux required at each of the two inputs of the functional unit implementing operation type t_i is given by:

$$s(t_i) = \left(\frac{w(t_i)}{2 \times n(t_i)} \right) \tag{12}$$

where,

$s(t_i)$ is the size of the mux needed on each of the two inputs of the functional units implementing the operation of type t_i,

$w(t_i)$ is the number of writes to an operator of type t_i, i.e., the number of time the operator is a target (or destination) of a design path (determined in Section 4.3.1.

$n(t_i)$ is the number of operators of type t_i allocated to the design.

4.3.5 Comparing Average Case Estimate to the Worst Case

The Average Usage Mux Estimation Method (average case) attempts to estimate mux sizes by averaging the number of paths in which the operation is a destination, over all the available operators of that type. The *best case* is defined as the case when all the sources which write to the operator are connected to a single input of the operator. In the best case (with respect to the fewest number of multiplexers used), only one functional unit is used to implement the operation with all but one input being mapped to the same input of the functional unit.

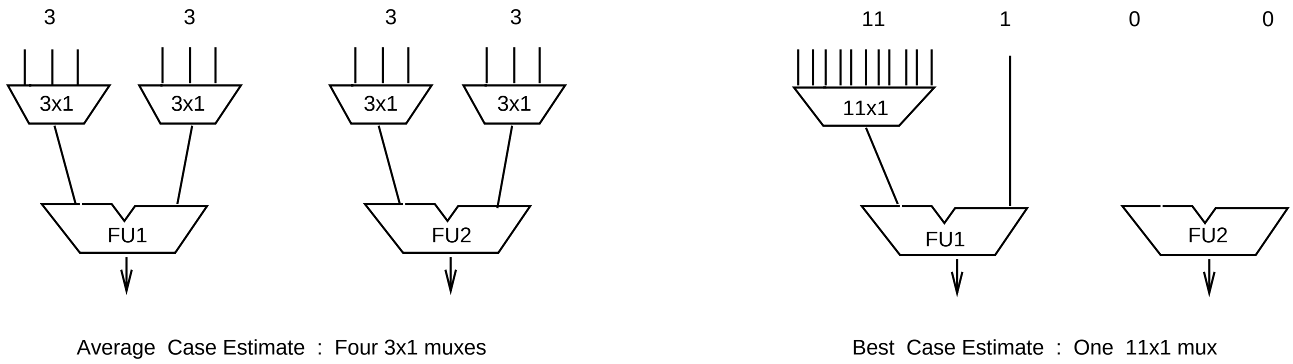


Figure 16: Average and Best Case Mux Estimates for $w_i = 12, n_i = 2$

4.3.6 Example : Average Usage Mux estimation

The best and average cases for $w_i = 12$ and $n_i = 2$ are shown in Figure 16.

Table 1 gives some average case estimate and best case mux areas for some typical values of w_i, and n_i. The mux areas were calculated by examining the layout of the standard cells generated by GDT. The multiplexers used are assumed to have one control line per mux input. The percentages represent the error involved in the best case when compared to the average case.

As a result of mux estimation, the set of datapath components is augmented by the muxes required for operators and registers. Also the set of paths the set of paths in the design is modified to reflect any new register-mux and operator-mux paths that may have been created.

w_i, n_i	Average Case			Best Case			Percent Error
	Muxes	Size	Total Area	Muxes	Size	Total Area	
$w_i = 6, n_i = 1$	2	3 to 1	272	1	5 to 1	206	24%
$w_i = 16, n_i = 4$	8	2 to 1	808	1	15 to 1	556	31%
$w_i = 18, n_i = 3$	6	3 to 1	816	1	17 to 1	626	23%

Table 1: Average Case Mux Area Estimates Compared with the Worst Case

4.4 Estimating the number of Microstates

A *microstate* is defined as the set of operations that can be executed in exactly one clock. We wish to estimate the number of microstates in the SpecChart so as to be able to estimate the number of bits in the state register of the control unit and the control logic area.

The method employed to determine the number of microstates in the design is identical to that presented in Section 3.2. The only difference is that in case the leaf state VHDL code has conditional branching like those caused by *if* and *case* statements, the maximum number of microstates in any branch is used to find the total number of microstates in the given program (i.e. microstates are shared across conditional branches). Probability based analysis is not needed here to estimate the number of microstates in the leaf state. The number of microstates in the design is the sum of the microstates of all the leaf behaviors in the SpecChart

The width of the state register, SR, in the control unit can then be determined as:

$$BitWidth_{SR} = log_2 (MS) \tag{13}$$

where,

MS is the total number of microstates in the SpecChart.

$BitWidth_{SR}$ is the bitwidth of the state register, SR.

4.5 Estimating the Control Logic Area

The Control Unit may be implemented using random logic or a ROM.

4.5.1 Implementing the control in Random Logic

The general organization of the control logic is given in Figure 17. The value for the next state lines and the control signals is computed using a sum-of-products expression, where the sum and product terms are implemented with OR and AND gates respectively. States are encoded using binary values stores in the state register, SR.

We assume that at most *one* datapath conditional line (results of comparators, carry bit of ALUs etc.) may affect the value of any control line. Thus the total number of inputs on which the value of a control line depends is $(BitWidth_{SR} + 1)$, where $BitWidth_{SR}$ is the width of the state register, SR.

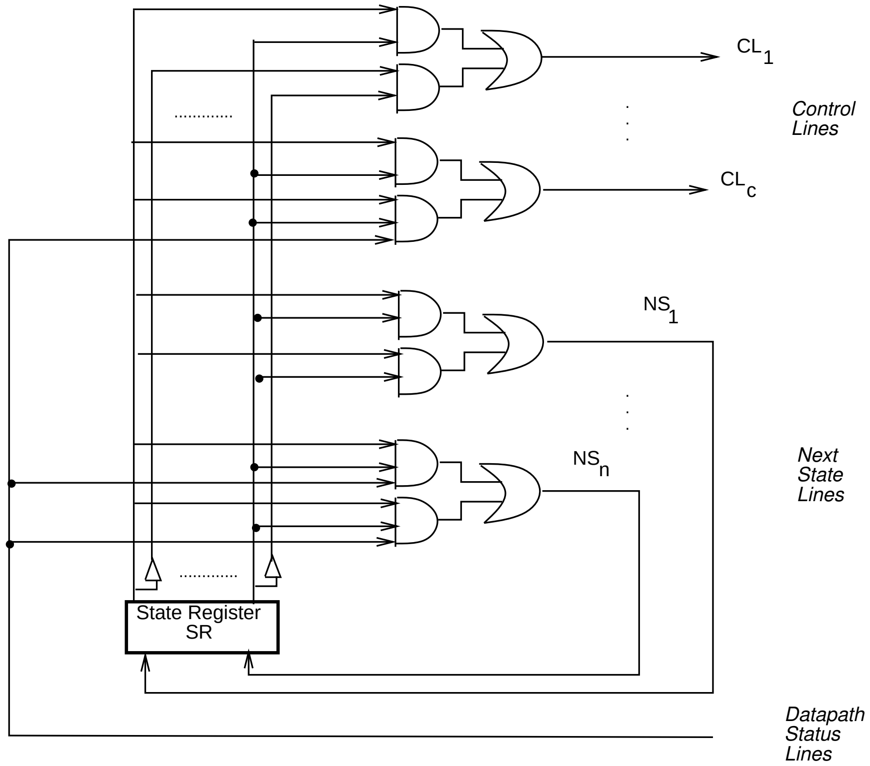


Figure 17: 2-level AND-OR implementation of Control Logic

4.5.2 Determining the number of active states for Control Lines

The number of AND gates required for implementing the logic for a given control line depends on the number of control states when that control line is *active*, i.e. the number of control states it is used to control a component in the datapath. For e.g., when a register is written to in any state, then the load line for the register is said to be active in that state. The number of active states for a control line can be estimated depending on the type of component it controls. The method outlined below assumes that the number of control lines for a multifunctional unit is the same as the number of functions in the unit. In addition, each control line in a multiplexer selects exactly one of its inputs.

(a) **Register Control Lines** : The active states for load lines are those in which the register is written to. If the register has a reset line, it will be activated whenever the register is cleared. Since we do not have an exact schedule of operations in the design, we ignore the writes to the same register in different branches of a conditional statement, while determining the number of active states for the load line of a register.

(b) **Mux Control Lines** : A Mux is considered to be active whenever the component for which it selects data is active. An optimization that reduces the number of control line active states for muxes is to assign the *all zero* select combination to the source for the component (driven by the mux) which is most often used.

(c) **Operator Control Lines** : Operator select lines will be active whenever the operator is used. If more than one operator of a given type is allocated for the design, then the number of active states can be averaged over the available operators. For e.g. if there are 24 additions in the specifications, and 3 adders are allocated for the design, then each adder control line will be assumed to be active for 8 states. In case the same component implements several operations, then the number of active states for its control lines is the sum of the active states for each of several operations that are implemented by the component. An optimization that would reduce the number of control line active states for such multi function units would be to assign the select combination of *all zeroes* to the operator that occurs most frequently.

(d) **Memory Control Lines** : The number of reads from a memory is the number of states that the control line RD is active. Similarly the WR line active whenever the memory is written to.

4.5.3 Estimating the Size and Number of AND / OR Gates

Determining the number of AND gates

The minterms in the sum of products expression for each control line represent a condition when the control line is "active". Thus, the number of AND gates required is the sum total of all the microstates during which a control line is active, i.e.,

$$N_{and} = \sum_{i=1}^{c} \begin{pmatrix} \textit{no. of microstates during} \\ \textit{which } CL_i \textit{ is active} \end{pmatrix} \tag{14}$$

where,

N_{and} is the number of AND gates in the control logic.

CL_i represents control line i.

c is the total number of control lines needed to drive the datapath and memories.

Assuming that at most one conditional status line from the datapath affects any control line, the number of AND gates has an *upper bound* of $2^{(BitWidth_{SR}+1)}$. This upper bound represents the maximum number of minterms that can be generated with $(BitWidth_{SR} + 1)$ boolean variables.

Determining the number of inputs for each AND gate

The number of inputs of an AND gate is depends on the number of bits in the state register and the number of datapath conditional status lines. We assume that at most *one* conditional status line affects a control line. Thus, the number of inputs to an AND gate is calculated as:

$$INP_{and} = BitWidth_{SR} + 1 \tag{15}$$

where,

INP_{and} be the number of inputs for each AND gate.

$BitWidth_{SR}$ is the bitwidth of the state register, SR.

Determining the number of OR gates

The control unit must generate c control lines for the datapath/memories and n next state lines which determine the next state for state register SR. Each of these lines is implemented as a sum of minterms (the minterms are derived from the current state and conditional status lines from the datapath). Thus, the number of OR gates required are given by:

$$N_{or} \;=\; c \;+\; n \tag{16}$$

where,

N_{or} is the number of OR gates in the control logic.

c is the number of control lines needed to drive the datapath and memories.

n is the number of next-state lines which is the same as $BitWidth_{SR}$, the bitwidth of the state register, SR.

Determining the number of inputs for each OR gate

For a given control line, the number of inputs the corresponding OR gate will require is equal to the number of minterms in the sum of products expression for that control line. Thus, the number of inputs that an OR gate for control line CL_i has is given by,

$$INP_{or}(CL_i) = \left(\begin{array}{c} no.\ of\ microstates\ in \\ which\ CL_i\ is\ active \end{array} \right) \tag{17}$$

where,

$INP_{or}(CL_i)$ be the number of inputs for the OR gate for control line, CL_i.

CL_i represents control line i.

We assume that each next-state line toggles the corresponding state register bit at each state. Then, each next-state line has the value '1' on half the states and value '0' in the remaining states.

Thus, each next-state line is generated using an OR gate with a number of inputs equal to half the number of states in the design. The number of inputs that an OR gate for next-state line NS_i has is given by,

$$INP_{or}(NS_i) = 2^{Bitwidth_{SR}} \div 2 \tag{18}$$

where,

$INP_{or}(NS_i)$ be the number of inputs for the OR gate for next-state line, NS_i.

$BitWidth_{SR}$ is the bitwidth of the state register, SR.

Determining the number of NOT gates

For state decoding we also have inverters for each of the bits of the state register. Thus, the number of NOT gates required for state decoding is

$$N_{not} = BitWidth_{SR} \tag{19}$$

where,

N_{not} is the the number of inverters required for state decoding.

$BitWidth_{SR}$ is the bitwidth of the state register, SR.

4.5.4 Implementing the control using ROM

The control logic may also be implemented using a ROM. Each microstate in the design must have corresponding word in the ROM. The bitwidth of the ROM should be the sum of the control lines required to drive the datapath/memory and the next state lines needed.

Thus we will require a $W \times B$ ROM as computed by:

$$W = 2^{BitWidth_{SR}} \tag{20}$$

$$B = c + n \tag{21}$$

where,

W is the number of words in the ROM.

B is the number of bits in a word

$BitWidth_{SR}$ is the bitwidth of the state register SR.

c is the number of control lines for the datapath/memory.

n is the number of next-state lines.

4.6 Calculating the Design Area

Having determined the datapath components and the control logic, we now determine the total area of the design.

4.6.1 Computing Datapath Area

The datapath layout architecture is a bit-sliced macro standard cell [6, 7]. This architecture is shown in Figure 18(a). The standard cells of each bit-slice are placed vertically and a single routing channel is used to connect different units within the same bit-slice. Control lines run horizontally in second metal and power and data lines run vertically in ploy or first metal. The number of transistors in each component is computed according to the equations of Figure 18(b).

The number of transistors in the datapath are computed as:

$$tr(DP) \;=\; \sum_{i=1}^{r} tr(REG_i) \;+\; \sum_{i=1}^{f} tr(FU_i) \;+\; \sum_{i=1}^{m} tr(MUX_i) \tag{22}$$

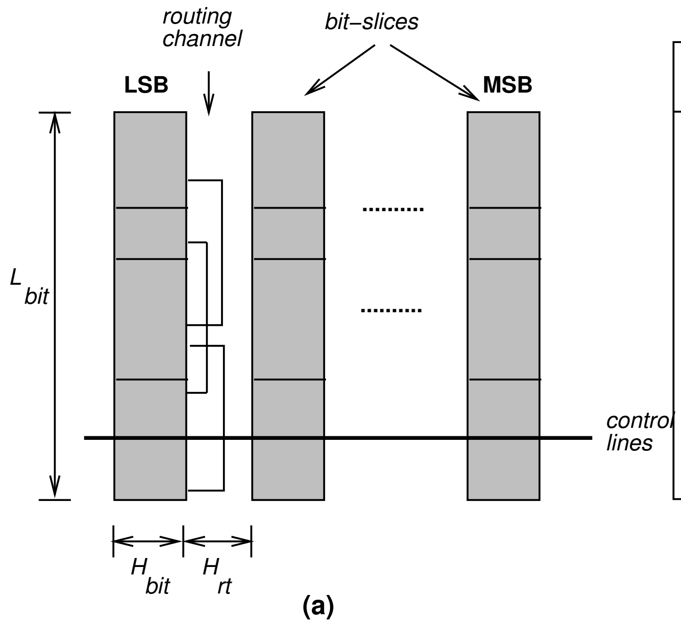
where,

$tr(DP)$ is the total number of transistors in each bit-slice of the datapath.

$tr(REG_i)$ is the no. of transistors in a single bit of register i.

$tr(FU_i)$ is the no. of transistors in a single bit functional unit i.

$tr(MUX_i)$ is the no. of transistors in a single bit of multiplexer i.



(a)

Component	No. of transistors
ADD, SUB	35*b
MULT	6*b*b + 14*(b–1) + 34*(b–1)*(b–2)
MUX	4*b*(i+1)
INV	2
AND, OR	2*i + 2
NAND, NOR	2*i
REGISTER	24*b
MEMORY	6*b*w + w*(2*log w + 2) + 2*log w

b = No. of bits, i = No. of inputs, w = No. of words

(b)

Figure 18: (a) Datapath Layout plan (b) Transistor counts for various components

r is the number of registers in the datapath (determined in Section 4.2).

f is the number of functional units in the datapath (input to estimator).

m is the number of multiplexers in the datapath (determined in Section 4.3).

The length of each bit slice in Figure 18(a) is then calculated as:

$$L_{bit} \;=\; \alpha \times tr(DP) \tag{23}$$

where,

L_{bit} is the length of each bit-slice in the datapath.

$tr(DP)$ is the total number of transistors in each bit-slice of the datapath (Equation 22).

α is the transistor pitch coefficient in μm/transistor.

The area of each bit-slice can then be computed as:

$$A_{bit} \;=\; L_{bit} \times H_{bit} \tag{24}$$

where,

A_{bit} is the area of each bit-slice in the datapath.

L_{bit} is the length of each bit-slice in the datapath.

H_{bit} is the fixed height of the cells in the bit slice.

The area of the routing channel associated with each bit-slice can be calculated as:

$$A_{routing} = L_{bit} \times \beta \times \#nets \times pitch \tag{25}$$

where,

$A_{routing}$ is the area of the routing channel.

L_{bit} is the length of each bit-slice in the datapath.

β represents the number of tracks required per net (incorporates the sharing of tracks to implement interconnections).

$\#nets$ is the number of estimated nets in the datapath.

$pitch$ is the wiring pitch.

Finally, the total area of the datapath can then be determined as:

$$A(DP) = bw \times (A_{bit} + A_{routing}) \tag{26}$$

where,

bw is the datapath bitwidth.

A_{bit} is the area of each bit-slice in the datapath.

$A_{routing}$ is the area of the routing channel.

4.6.2 Computing the Control Logic Area

The number of transistors in the control logic can be calculated from the knowledge of the gates determined in Section 4.5. We assume that all the gates in the control logic are laid out in a single row.

The number of transistors in the control logic are computed as:

$$tr(CL) = \left[\begin{array}{c} tr(SR) + N_{not} \times tr(INV) + N_{and} \times tr(AND_{INP_{and}}) + \\ \sum_{i=1}^{c} tr(OR_{INP_{or}(CL_i)}) + \sum_{i=1}^{n} tr(OR_{INP_{or}(NS_i)}) \end{array} \right] \tag{27}$$

where,

$tr(CL)$ is the total number of transistors in the control logic.

$tr(SR)$ is the total number of transistors in the state register, SR.

c is the number of control lines required for datapath and memories.

n is the number of next-state lines.

N_{and} is the number of AND gates (Equation 14).

$tr(AND_{INP_{and}})$ is the number of transistors in an AND gate with INP_{and} inputs (Equation 15).

$tr(OR_{INP_{or}(CL_i)})$ is the number of transistors in an OR gate with $INP_{or}(CL_i)$ inputs (Equation 17).

$tr(OR_{INP_{or}(NS_i)})$ is the number of transistors in an OR gate with $INP_{or}(NS_i)$ inputs (Equation 18).

N_{not} is the number of inverters (Equation 19).

$tr(INV)$ is the number of transistors in an inverter.

The area of the control logic can then be computed as:

$$A(CL) \;=\; \gamma \times tr(CL) \tag{28}$$

where,

$A(CL)$ is the area of the control logic.

γ is the transistor area coefficient in μm^2/transistor.

$tr(CL)$ is the total number of transistors in the control logic.

4.6.3 Computing the Memory Area

The number of transistors in the memories are computed as:

$$tr(MEM) \;=\; \sum_{i=1}^{N_{mem}} tr(MEM_i) \tag{29}$$

where,

$tr(MEM)$ is the total number of transistors in all the memories.

N_{mem} is the number of memories in the design.

$tr(MEM_i)$ is the number of transistors in memory MEM_i (equation in Figure 18(b)).

The area of the memory can then be computed as:

$$A(CL) \;=\; \gamma \times tr(MEM) \tag{30}$$

where,

$A(MEM)$ is the area of the memories.

γ is the transistor area coefficient in μm^2/transistor.

$tr(MEM)$ is the total number of transistors in the memories.

4.6.4 Computing the Total Area

The total number of transistors and the area of the design can now be calculated as:

$$TOTAL\ TRANSISTORS \;=\; tr(DP) \;+\; tr(CL) \;+\; tr(MEM) \tag{31}$$

$$TOTAL\ AREA \;=\; A(DP) \;+\; A(CL) \;+\; A(MEM) \tag{32}$$

The coefficients α, β and γ are determined empirically and depend on the technology and the layout system used. For this report, we used the GDT 3 micron technology component library [8]. The values of these coefficients were:

α = 13.4 μm/transistor,

β = 0.46 routing tracks/net,

γ = 1100 μm^2/transistor.

In addition, H_{bit}, the height of the cells in the datapath bit-slice was 82 microns.

5 Results

We tested the SpecChart estimators on a variety of examples which were adapted from [5]. The results of estimating the area and performance of these designs are tabulated in Figure 19. For each example, we have compared with the actual design the number of registers, the number of multiplexer inputs, the total number of microstates or control steps in the design, and the number of transistors in the datapath, control unit and the entire design. We will examine each example separately. In Figure 19, the third row for each example represents the percentage error by which the estimate of a design parameter differed from the *actual* or *measured* value of that parameter. The actual values were determined from either the layout of the design (if available) or from the netlists produced by the application manual or automatic synthesis. The estimator has a runtime of the order of few seconds running on a Sun Sparc2 workstation.

5.1 Fifth Order Elliptical Filter

The estimation results for the elliptic filter example were compared with the layouts reported in [9]. For example, the error in the estimation for the number of registers in the design was 9% whereas the error in the number of microstates (or control steps) in the design was 16%. The overall transistor count for the design had an error of 8%.

5.2 Differential Equation Example

The estimation results for the differential equation example were compared with the RT-netlist produced for the datapath and control units by the VHDL Synthesis System [10]. The estimated overall transistor count is within 6% of the measured value.

Design		# Registers in datapath	# Mux Inputs	# Microstates	# Datapath Transistors	# Control Logic Transistors	#Total Transistors
ELLIPTICAL FILTER	**Estimated**	12	33	22	26,124	856	26,980
	Actual	11	28	19	25,164	740	25,904
	% Error	**9 %**	**18 %**	**16 %**	**4 %**	**16%**	**4 %**
DIFFERENTIAL EQUATION	**Estimated**	8	15	14	6,836	362	7,198
	Actual	7	24	13	7,788	336	8,124
	% Error	**14 %**	**37 %**	**8 %**	**12 %**	**8 %**	**11 %**
AM 2901	**Estimated**	6 Reg 16 x 4 Reg File	30	2	3,172	750	3,922
	Actual	3 Reg 16 x 4 Reg File	17	1	2,472	684	3,156
	% Error	**–**	**76 %**	**100 %**	**28 %**	**10 %**	**24 %**
KALMAN FILTER	**Estimated**	4 Reg 6 Reg Files	37	33	110,886	892	111,778
	Actual	4 Reg 6 Reg FIles	29	23	108,954	606	109,560
	% Error	**0 %**	**27 %**	**43 %**	**2 %**	**47 %**	**2 %**
AM 2910	**Estimated**	4 registers	20	2	5,244	714	5,958
	Actual	1 inc, 1 dec 1 inc/dec	11	1	3,888	552	4,440
	% Error	**–**	**81 %**	**100 %**	**35 %**	**29 %**	**34 %**
TRAFFIC LIGHT CONTROLLER	**Estimated**	4	23	1	705	240	945
	Actual	4	15	1	840	–	840
	% Error	**0 %**	**53 %**	**0 %**	**–16 %**	**–**	**12 %**

Figure 19: Estimation Results

5.3 Am2901 Microprocessor

The estimates produced for the Am2901 were compared with the layout of the Am2901 [9]. The number of estimated mux inputs is 76% more than the actual number of mux inputs in the Am2901. The reason for this is the fact that several temporary variables were used in the description which are not present in the design. Consequently, the datapath area (which contains the extra muxes estimated) and control unit area (which has the control logic for driving the control lines for the muxes) are greater than the actual.

The largest disagreement between the estimated and the measured value occurs in the case of the number of microstates in the design. The estimator calculated 2 microstates whereas the actual design has only 1. The reason for this disparity becomes evident if we were to examine the description of the Am2901 in [5]. The estimation of microstates, as explained in Section 3 created basic blocks from VHDL statements. The sum of temporary variables is calculated and subsequently is used in a later statements. This causes the estimator to create two basic blocks and consequently two microstates are estimated for the design.

5.4 Kalman Filter

The results of estimation for the Kalman filter [11] were compared with a manually synthesized design. The Kalman filter is interesting in that it involves multiple matrix multiplications and thus is one of the few designs which has significant amount of storage organized as memories. The small estimation error is partly due to these large memories. If we exclude the the memories from the transistor counts, the error in the total number of transistors is approximately 15%.

5.5 Am2910 Microprogram Sequencer

The Am2910 is a microprogram address sequencer intended for use in high-speed microprocessor applications. The estimates for this example were compared with the layout reported in [9]. The

Am2910 has a multiplexor which selects data from a microprogram counter (uPC), a register/counter (R), a top of stack pointer (TOS) and direct data input. As was the case in AM2901 above, we see a 100% error in the number of microstates estimated for the AM2910. The reasons for this are similar to those for Am2901 above.

While counting the mux inputs for the actual AM2910 design, we have included muxes needed to implement the register/counter (2 mux inputs - one each for load and decrement operations), microprogram counter (2 mux inputs) and stack pointer (2 mux inputs - one each for push and pop operations). The error is mainly due to the use of the temporary variables in the description which the estimator maps to registers with multiplexers at the inputs.

5.6 Traffic Light Controller

The estimates for the Traffic Light Controller were compared to a manually synthesized design. The manually synthesized design was a single state machine with only a datapath.

The most significant error is in the number of mux inputs. The reason for this is that the estimator used a 3-bit comparator for comparing the value of the state variable $current\_state$, whereas the same was implemented using simple gates in the manual design.

Design	Estimated Area	Actual Layout (ICCAD 92)	% Error
Elliptical Filter	66.46	55.82	19 %
AM2901	7.89	10.01	21 %
AM2910	11.97	12.87	7 %

Figure 20: Comparing Estimated area with the actual layouts

In Figure 20, we compare the estimated area with the actual layout area reported in [9]. The area was estimated as outlined in Section 4.6 using the GDT 3 micron standard cell library.

6 Conclusion

In this report we have presented estimators for the performance and area of system-level designs. The estimators provide the designer with instant feedback as to how any design decision might affect these quality metrics. Since the input to the estimators is a high-level HDL specification (as opposed to a netlist, CDFG's etc.), the designer has the capability of performing system-level synthesis tasks (such as partitioning the specification into chip-level specifications) without having to perform partial synthesis in order to obtain estimates. The time required for each iteration involving a design decision and evaluation is significantly reduced. This enables the designer to search a larger design space which may lead to faster and/or smaller designs.

7 Acknowledgements

This work was supported by the National Science Foundation (grant #MIP-8922851) and the Semiconductor Research Corporation (grant #91-DJ-146). We are grateful for their support. The authors would wish to thank Frank Vahid for testing the estimators extensively and for his helpful suggestions. We are grateful to Chamapaka Ramachandran for her valuable assistance with examples used to evaluate the estimators. We also wish to acknowledge the several discussions and suggestions from Logie Ramachandran and Viraphol Chaiyakul.

8 References

[1] S. Narayan, F. Vahid, and D. Gajski, "System Specification and Synthesis with the SpecCharts Language," in *Proceedings of the International Conference on Computer-Aided Design*, 1991.

[2] S. Narayan and D. Gajski, "System Clock Estimation based on Clock Slack Minimization," in *Proceedings of the European Design Automation Conference*, 1992.

[3] D. Gajski, N. Dutt, C. Wu, and Y. Lin, *High-Level Synthesis: Introduction to Chip and System Design.* Kluwer Academic Publishers, 1991.

[4] A. Aho, R. Sethi, and J. Ullman, *Compilers Principles, Techniques, and Tools.* Addison Wesley, 1988.

[5] N. Dutt and C. Ramachandran, "Benchmarks for the 1992 High Level Synthesis Workshop." UC Irvine, Dept. of ICS, Technical Report 92-107,1992.

[6] C. Wu, V. Chaiyakul, and D. Gajski, "Layout-Area Models for High-Level Synthesis," in *Proceedings of the International Conference on Computer-Aided Design*, 1991.

[7] C. Wu, V. Chaiyakul, and D. Gajski, "Layout-Area Models for High-Level Synthesis." UC Irvine, Dept. of ICS, Technical Report 91-31,1991.

[8] , *GDT 3 micron Standrad Cell Datapath Library.* Mentor Graphics, 1990.

[9] C. Ramachandran, F.J. Kurdahi, D. Gajski, C.H. Wu and V. Chaiyakul, "Accurate Layout Area and Delay Modeling for System-level Design," in *Proceedings of the International Conference on Computer-Aided Design*, 1992.

[10] J. Lis, *Behavioral Synthesis from VHDL Using Structured Modeling.* PhD thesis, University of California, Irvine, January 1992.

[11] D. Thomas, et. al., *Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench.* Kluwer Academic Publishers, 1990.

A On-line Manual Pages

A.1 SCESTAREA : Area Estimator

```
SCESTAREA(L)             LOCAL COMMANDS             SCESTAREA(L)


NAME
     scestarea - estimate area for given SpecChart

SYNOPSIS
     scestarea [ -fvd ] [ -t componentfile ] specchartfile  statename

DESCRIPTION
     scestarea estimates the area of the specified state  in  the
     given SpecChart.

     The component file contains the clock cycle and the  follow-
     ing  information for each component type used in the design:
     class, type, delay, area/bit, control lines, and  number  of
     components  of  that  type allocated to the design.  In case
     the component  file is not specified using the '-t'  option,
     the   area   estimator   will   look   for   a  file  called
     'specchart.cmp' in the current directory. Thus, if  a  Spec-
     Chart is stored  'system.sc' and no component file is speci-
     fied, the estimator will look for the file 'system.cmp'.  If
     it  does  not  find 'system.cmp', it will be created, having
     one entry for each of the operator types in VHDL and default
     values for the fields mentioned above.

     If no options are specified, the estimator by  default  pro-
     duces a single-line non-verbose output.

OPTIONS
     -t compfile
          File containing the component information. The  default
          file is 'specchart.cmp'.

     -d   Output  the  results  on  three  files  :   ",tmpcomp",
          ",tmpdata",  and  ",tmpstate" which contain the results
          of the estimation for use by xscestarea.

     -f   Estimate the area by  first  building  a  special  data
          structure  containing  various  pieces  of  information
          about the SpecChart, and then using it to  compute  the
          area. Not observable speedups  if this option is used.

     -v   Verbose output which also lists the  number  of  micro-
          states and components.

FILES
     ./*.sc              SpecChart files.
     ./*.scp             SpecChart packages.
     ./*.cmp             File containing component information.
     ./,tmpcomp          File containing the list  of  components
                         when the '-d' flag is used.
     ./,tmpdata          File containing the data such as  number
                         of  bits in state register etc. when the
                         '-d' flag is used.
     ./,tmpstate         File containing  the  number  of  micro-
                         states  required  by  each leaf behavior
                         when the '-d' flag is used.
```

SEE ALSO
    xscestarea

AUTHOR
    Sanjiv Narayan (narayan@ics.uci.edu)

A.2 SCESTPERF : Performance Estimator

```
SCESTPERF(L)              LOCAL COMMANDS              SCESTPERF(L)


NAME
     scestperf - estimates performance  for  a  given  SpecChart
     state or procedure.

SYNOPSIS
     scestperf [ -t compfile ] [ -dipv ] specchart statename

DESCRIPTION
     Computes the execution time of the  specified  procedure  or
     state  in  terms  of clock cycles. For a non-leaf state, the
     estimator  performs  control-flow  analysis   of  the  state
     sequencing   at  each  hierarchical  level.  The state being
     estimated must not have any descendant concurrent substates.
     Each non-leaf state  MUST have "stop" dot in it.

     The component file contains the  clock cycle and the follow-
     ing   information   for  each   component  type  used in the
     design: class, type, delay,  area/bit,  control  lines,  and
     number  of   components   of   that  type  allocated  to the
     design.  In case the component  file is not specified  using
     the '-t' option, the  estimator  will look for a file called
     'specchart.cmp' in the current directory. Thus, if  a  Spec-
     Chart  is   stored    'system.sc'  and  no component file is
     specified,  the   estimator   will   look   for   the   file
     'system.cmp'.  If  it does not find 'system.cmp', it will be
     created, having one entry for each of the operator types  in
     VHDL and default values for the fields  mentioned  above.

     If no options are specified, the estimator by  default  pro-
     duces  a  non-verbose output without inlining procedures and
     using the component file 'specchart.cmp'.

OPTIONS
     -t   File containing the component information. The  default
          file is 'specchart.cmp'.

     -d   Dumps results to files, for use by xscestperf.

     -i   Inline procedures. If this flag is omitted, the estima-
          tor  will ignore all procedure calls encountered in the
          VHDL code.

     -p   Indicates that the "state_name" parameter is actually a
          procedure declared in the SpecChart. The performance of
          the procedure  is then estimated.

     -v   Verbose mode, prints  estimates  at  each  hierarchical
          level.

FILES
     ./*.sc      SpecChart files
     ./*.scp     Packages required by the specchart.
     ./*.cmp     File containing component information.

BUGS
     Currently, channels are treated as procedure calls.


SEE ALSO
     xspecsyn(L), xscest(L), xscestperf(L)
```

AUTHOR

Sanjiv Narayan (narayan@ics.uci.edu), Computer Science Dept, University of California, Irvine