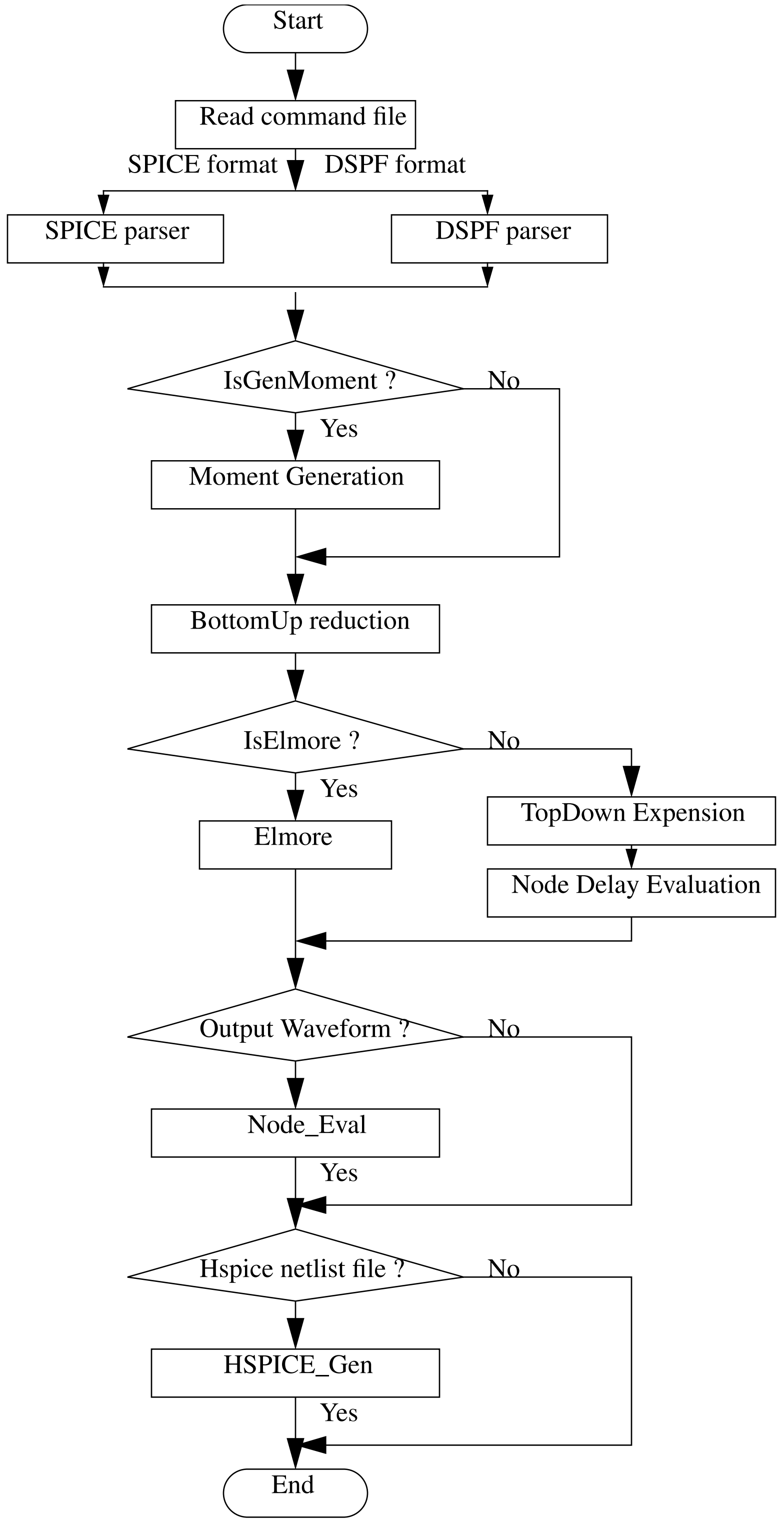
Hurwitz Interconnect Delay Evaluation - HIDE

Programmer's Manual

Zhanhai Qin
Chung-Kuan Cheng

VLSI Lab, Computer Science & Engineering Department
University of California, San diego
Nov. 18, 2000

Flow Chart


Start
Read command file
SPICE format
DSPF format
SPICE parser
DSPF parser
IsGenMoment ?
No
Yes
Moment Generation
BottomUp reduction
IsElmore ?
No
Yes
Elmore
TopDown Expension
Node Delay Evaluation
Output Waveform ?
No
Node_Eval
Yes
Hspice netlist file ?
No
HSPICE_Gen
Yes
End

Data Structure & Routines

Note: The functionality of the program is described in HIDE User’s manual.

All the macros, data structures, and global variables are defined in *HIDE/inc/HDstruct.h.*

Macros

define **NP** 2 */* Define the number of poles */*
define **NM** 4 */* Define the number of moment orders */*
define **MAXCOLUMN** 100 */* Define the maximal number of characters in any lines of an ASCII file */*
define **MAXIT** 10 */* Define the maximal number of iterations in Newton-Ralphson approach */*
define **XACC** 1e-4 */* Define the error upper bound for Newton-Ralphson approach */*

define DRIVER_OUT 0 */* Define the positions of a test node in test-node list */*
define BRANCH 1
define RECEIVER_IN 2
define SINK 3

define TBLSIZE 1024 */* Define the size of the Hash table to store nodes */*

define MAX(x1,x2) ... ... */* Get the maximum of x1 and x2 */*
define MIN(x1,x2) ... ... */* Get the minimum of x1 and x2 */*

define SQR(x) ... ... */* Get x square */*
define CUBE(x) ... ... */* Get x cube */*

define NewString(x,l) ... ... */* Allocate an l long string to char-typed pointer variable x with initial values zero */*
define NEWNODE(x) ... ... */* Allocate an node to NODEPTR-typed pointer variable x with initial values zero */*
define NEWTESTNODE(x) ... ... */* Allocate a new testnode to (TESTNODE *)-typed pointer variable x with initial values zero */*
define NEWTABLE(x,y) ... ... */* Allocate a hash table with y entries to NODEPTR-typed pointer variable x with initial values zero */*

define STRCPY(x,y) ... ... */* Allocate space for char-typed pointer variable x and copy the content in y into x */*

Data structures

```
typedef struct rc{
   Real r;
   Real c;
}RC_Pair;
```

RC parameters from parent node to the current node. See **NODE** structure for details.

```
typedef struct pi_model{
   Real C1;
   Real R;
   Real C2;
}PI_Model;
```

Save π model parameters. Literally speaking, they are two capacitors and one resistor.

```
typedef struct transfer_function{
   Real  a[2];
   Real  b[3];
}TranFun;
```

Save the coefficients of every transfer function. **b[]** is for denominator coefficients, and **a[]** is for numerator coefficients.

```
typedef struct two_pole{
  Real     a[2], b[3];
  Real     P[3], K[3];
}Two_Pole;
```

Save poles and residues in **P[]** and **K[]** respectively.

```
typedef struct admittance_coefficents{
 Real y[4];
}Adm_Coeff;
```

Save the first four admittance coefficients.

```
typedef struct node{

/* Node information */
   char   * NID;            /* Name of the node */
   char   * PID;            /* Name of the parent */
   int      Nchild;         /* Number of the node's Child(ren) */
   int      Nreduction;     /* Number of reduced branches of the node */
   struct node * parent;    /* Parent ptr of the node*/
   struct node * child;     /* The first child ptr of the node */
   struct node * brother;   /* The next brother ptr of the node */
   RC_Pair  RC;             /* RC Parameter from the parent to the present node; Load capacitance included*/
   struct node * next;      /* Used to link synonyms in Hash table*/

/* Reduction Parameter */
   PI_Model    Down_PI;     /* π equivalent circuit of the downstream tree structure */
   PI_Model    Up_PI;       /* Reduced π equivalent circuit seen from the parent of the node */
   Adm_Coeff   Down_AC;     /* Total Admittance Coefficients of the downstream tree structure*/
   Adm_Coeff   Up_AC;       /* Admittance Coefficients seen from the parent of the node */
   Two_Pole    TP;          /* Two Pole Model*/
   TranFun     Up_TF;       /* Transfer Function from the parent of the node*/

/* Parsed command lines */
   Real     trstart;        /* start point for rise time */
   Real     trend;          /* end point for rise time */
   Real     tdelmore;       /* Elmore delay from source */
   Real     td2m;           /* D2M delay */
   Real     td;             /* 50% delay from the source to the present node */

/* For moment evaluation use */
   Real     I;              /* Branch current */
   Real     m[NM];          /* Saved nodal moments */
   Real     Y[NM];          /* DownStream AC */
}NODE;

typedef NODE * NodePtr;
```

Node related information is saved in this data structure.

```
typedef struct signal{
   char  *  func;        /* Specify the function name, exp, ramp, or step. */
   Real     Vdd;         /* Supply Voltage */
   Real     tr;          /* Rise time of the signal */
    char  *  SrcNode;    /* Name of the Source Node, e.g. IN */
    NodePtr  root;       /* root of the interconnect tree */
}SIGNAL;
```

Input Signal information is saved in this data structure.

```
typedef struct stack{
  NodePtr Pnode;
  struct stack * NextPtr;
}Stk;

typedef Stk * StkPtr;
```

A stack structure used in RC bottom-up reduction.

Global variables

```
char * isDebug ;                 /* Specify if the program is in debug mode or not */

char * isGenMoment ;             /* Specify if the program evaluates moments and outputs them */

char * isGenHspiceNetlist ;      /* Specify if the program needs to generate the equivalent Hspice netlist file */

char * isOutputWaveform;         /* Specify if the program outputs the waveform for test node(s) */

char * isElmore;                 /* Specify if the program outputs Elmore delay for user-concerned node(s) */

char * isD2M;                    /* Specify if the program outputs D2M delay for user-concerned node(s) */


char * FinFormat;                /* Specify the input netlist file format: spice or DSPF */

char * isOutputAll;              /* Specify if all nodes need to be printed out */

char * isRiseTime;               /* Specify if the program evaluates the rise time for user-concerned node(s) */

char * isUseDefaultTestNode;     /* Specify if test node list is default, or defined by users */

Real   Vtdth;                    /*  Define delay threshold. A real number between 0.0 and 1.0 */

Real   VtrStart;                 /*  Define output signal’s starting threshold. A real number between 0.0 and 0.5 */

Real   VtrEnd;                   /* Define output signal’s ending threshold. A real number between 0.5 and 1.0 */

FILE * Fin;                      /* File handle of input netlist file */

FILE * Fout;                     /* File handle of delay result output file */

FILE * Fspice;                   /* File handle of Hspice-formatted input netlist file if needed */

int      Ntotal_node;            /* Number of all nodes */

int      Ntestnode;              /* Number of all test nodes */

TESTNODE * TestNode;             /* Head of test node list */

SIGNAL   * Src;                  /* Root of everything: input netlist, stimulus, etc. */

Real      tstep,tstop;           /* Step size and stop point of simulation time. Used in waveform evaluation *

int       Total_iter;            /* Newton-Ralphson iteration control */

NodeNameList * AssignedTestNodeName;  /* Head of user-defined test node name list */
```

Subroutine Definitions

Please find the formal user-defined types in the section above.

Main.c : controls the whole program flow.

Interface : By including the header HDstruct.h.

Internal Routines : Routines used within this module;
`main` - main routine of the whole program, controlling the program running flow.

Referred External Routines : Routines defined outside to be used within this module;
ReadInput - Given HIDE configuration file, configure run-time settings.
SPICE_Parser - Given an input netlist file handle, returns StkPtr structure.
DSPF_Parser - Given the input netlist file handle, returns StkPtr structure.
BottomUpReduction - Reduce a given RC tree into an equivalent RC PI circuit.
Driver_Eval - Calculate a stable transfer function for driving point.
TopDownExpansion - Propagate transfer functions in a top-down fashion.
Elmore - Given an node, evaluate Elmore delay for itself and its downstream nodes.
Moment - Evaluate up to *NM*-th moments for the present node and its downstream nodes.
Delay - Given the root of a netlist, evaluate the delay for all the nodes in the netlist.
TestNodeDelay - Evaluate the delay for all the nodes in test node list.
Node_Eval - Evaluate the output response for test nodes.
HSPICE_Gen - Generate an HSPICE netlist file, which can be used for delay verification.
Data_Output - Output customized delay results for each user-interested node.

Routine(s) : description for each routine in this file;
main (argc, argv)
int argc; */* The number of arguments from Unix Shell */*
char *argv[]; */* The argument strings from Unix Shell */*

Usage : given HIDE configuration file from Unix Shell, return delay evaluation result.

Algorithm :
1. Check if the arguments from Unix Shell are in good format. Exit if they are not;
2. Call ReadInput() to parse the given HIDE configuration file;
3. Call SPICE_Parser() or DSPF_Parser() to parse the given input netlist file;
4. Call Moment() to evaluate nodal moments if needed;
5. Call BottomUpReduction() to reduce the given netlist;
6. Call Elmore() to evaluate nodes' Elmore delay if needed;
7. If Elmore() is not called, call Driver_Eval() and TopDownExpansion() to evaluate the driver and perform TopDown Expansion for transfer function propagation;
8. Call Delay() or TestNodeDelay to evaluate delay for all nodes or only selected test nodes;
9. Call Data_Output() to print delay evaluation results into files;
10. Call Node_Eval() to evaluate nodes' delay waveform if needed;
11. Call HSPICE_Gen() to generate equivalent HSPICE netlist file if needed.

ReadInput.c : parses the given HIDE configuration file.

Interface : By including the header HDstruct.h.

External Routines : Routines defined here to be used outside;
ReadInput - given HIDE configuration file, configure run-time settings.

Macros : macros used within this module;
NewNodeNameList(x) - allocates a NodeNameList structure to pointer variable *x*.

Routine(s) : description for each routine in this file;
void ReadInput (Finitial)
FILE *Finitial; */* Handle of the given HIDE configuration file */*

Usage : Given the file handle, configure HIDE run-time settings.

Algorithm :
1. retrieve ASCII strings line by line from Finitial;
2. parse each line by assigning a value to the corresponding run-time setting variable(s)

Parser.c : parses the given input netlist file in SPICE format.

Interface : By including the header HDstruct.h.

External Routines : Routines defined here to be used outside
SPICE_Parser - given the input netlist file handle, returns a well-filled StkPtr structure.

Internal Routines : Routines used within this module
Parse_spice - given the input netlist file handle, returns a well-filled Src structure.

Referred External Routines : Routines defined outside to be used within this module;
Hash - map an node name to a hash table entry;
InsertNode - given an node name, insert the node into the Hash table;
SearchNode - given an node name, return its entry ptr in the Hash table;
SortNode - link entries in the Hash table and return the input node as root;
Find_Sink_Node - build stack for later RLC reduction;
Display_Node - print out all the node in the Hash table (debug use);
Display_Sink_Node - print out all the nodes in the stack (debug use);

Routine(s)
void SPICE_Parser (Fin, StkTop)
FILE *Fin; */* Handle of the given input netlist file */*
StkPtr *StkTop; */* Top pointer of the stack containing all the nodes */*

Usage : Given input netlist file handle, returns the top entry pointer of node stack StkTop.

Algorithm :
1. Call Parse_spice() to parse input netlist file and build the structure Src;
2. Call Find_Sink_Node() to push all nodes into StkTop in bottom-up fashion;
3. If program is in Debug mode, call Display_Node() and Display_Sink_Node() to show retrieved nodal information in Src structure and StkTop structure, respectively.

int Parse_spice (Fin, Src)
FILE *Fin; */* Handle of the given input netlist file */*
SIGNAL *Src; */* Structure with all circuit and stimulus information */*

Usage : Given input netlist file handle, parse the file and returns Src structure.

Algorithm :
1. Skip empty and commented lines if any;
2. Allocate Hash table with *TBLSIZE* entries to *TblHead*;
3. Resistor definitions, Capacitor definitions, Independent voltage source definitions, *.tran* statements, and *.print* statements will be identified and parsed using a *switch* statement. Insert if the second node in Resistor definition is not in *TblHead* yet; also insert if the first node in Capacitor definition is not in *TblHead* yet. Link lists will be used as an solution to HASH confliction. Meanwhile, *Src->Vdd*, *Src->tr*, *tstep*, and *tstop* global variables will be assigned. Global variable *TestNode* will be updated if there is any *.print* statements. On the presence of *.end* statement, call SortNode() to link all the nodes in the Hash table and routine returns 1.

DSPF_Parser.c : parses the given input netlist file in DSPF format.

Interface : By including the header HDstruct.h.

External Routines : Routines defined here to be used outside
- DSPF_Parser - given the input netlist file handle, returns StkPtr structure;
- Hash - map an node name to an hash table entry;
- InsertNode - given an node name, insert the node into the Hash table;
- SearchNode - given an node name, return its entry ptr in the Hash table;
- SortNode - link entries in the Hash table and return the input node as root;
- Find_Sink_Node - build stack for later RLC reduction;
- Traverse_tree - traverse the whole tree in prefix order, label nodes if necessary;
- Display_Sink_Node - print out all the nodes in the stack (debug use);
- Display_Node - print out all the node in the Hash table (debug use);
- Push_Stk - create an node and push it into the stack;
- Pop_Stk - free the top node in the stack and return its *Pnode* field value.

Internal Routines : Routines used within this module
- Parse_dspf - given the input netlist file handle, returns well-filled Src structure.

Routine(s)
void DSPF_Parser (Fin, StkTop)
FILE *Fin; */* Handle of the given input netlist file */*
StkPtr *StkTop; */* Top pointer of the stack containing all the nodes */*

Usage : Given input netlist file handle, returns the top entry pointer of node stack StkTop.

Algorithm :
1. Call Parse_dspf() to parse input netlist file and build the structure Src;
2. Call Find_Sink_Node() to push all nodes into StkTop in bottom-up fashion;
3. If program is in Debug mode, call Display_Node() and Display_Sink_Node() to show retrieved nodal information in Src structure and StkTop structure respectively.

int Parse_dspf (Fin, Src)

FILE *Fin; */* Handle of the given input netlist file */*
SIGNAL *Src; */* Structure with all circuit and stimulus information */*

Usage : Given input netlist file handle, parse the file and returns Src structure.

Algorithm :

1. Skip empty and commented lines if any;
2. Allocate Hash table with *TBLSIZE* entries to *TblHead*;
3. Resistor definition lines, Capacitor definition lines, and interface node definition lines (starting with X...) will be identified and parsed using a *switch* statement. Insert if the second node in Resistor definition lines is not in *TblHead* yet; also insert if the first node in Capacitor definition lines is not in *TblHead* yet. Link lists will be used as an solution to HASH confliction. Global variable *TestNode* will be defined using output nodes (the nodes with names starting with 'O') if there is any *X...* statement. On the occurrence of *.ends* statement, call SortNode() to link all the nodes in the Hash table and routine returns 1.

void Traverse_tree (root)
NodePtr root;

Usage : Given an root node as input, all the downstream nodes will be visited in prefix order.

Algorithm :

1. call Traverse_tree recursively. Return if root = Null;
2. If root has no child, create a new test node with StateVar labeled as SINK and insert the test node into global variable TestNode, which is a test node list;
3. If root has only one child, nothing will be done but call Traverse_tree(root->child);
4. If root has more than one child, create a new test node with StateVar labeled as BRANCH and insert the test node into global variable TestNode, which is a test node list;
5. After visiting all the child(ren), if any, call Traverse_tree(root->brother) to visit its brothers.

NodePtr SortNode (TblHead)
NodePtr TblHead; */* point to the first entry of the Hash table */*

Usage : Link entries in the Hash table into a tree structure and return the root of the tree.

Algorithm :

1. For every nonempty entry in the Hash table, label it as root if it has no parent (*PID* field is *NULL*). Otherwise call SearchNode() to find its parent entry in the Hash table and make its *parent* field point to that entry. Also update the parent's *child* field at the same time.

int Hash (name)
char *name; */* Node name as input */*

Usage : Given an node name as input, apply an Hash function and return the value.

Algorithm :

1. $s = 1$ and k = strlen(name) - 1;
2. Hash function: for $i = 0$ to k $\quad s = 2 * (s + (*(name + i)))$

NodePtr InsertNode (TblHead, name)
NodePtr *TblHead; */* point to the first entry of the Hash table */*
char *name; */* Node name */*

Usage : Given an node name, insert the node into the Hash table;

Algorithm :
1. Call Hash() to get an Hash entry *i*;
2. If *i*th entry in the Hash table is empty, copy *name* to *NID* field of the entry;
3. Otherwise create a new node, copy *name* to *NID* field of the node, and insert the node into the link list of the *i*th entry;
4. Newly created node or the *i*th entry will be returned.

NodePtr SearchNode (TblHead, name)
NodePtr *TblHead; */* point to the first entry of the Hash table */*
char *name; */* Node name */*

Usage : given an node name, return its entry ptr in the Hash table. The ptr could point to an node in a Hash entry list.

Algorithm :
1. Call Hash() to get an Hash entry *i*;
2. If the *i*th entry is empty, return NULL;
3. Otherwise if the *i*th entry's *NID* = *name*, return the *i*th entry ptr;
4. Otherwise search through the *i*th entry's link list, return NULL if *name* is not matched, and return the node ptr if *name* is matched.

void Find_Sink_Node (TOP, StkTop)
NodePtr TOP; */* the root of the netlist in the Hash table */*
StkPtr StkTop; */* the top of the stack */*

Usage : Traverse the tree and push every node into the stack in an bottom-up fashion.

Algorithm :
1. If *TOP* is not a sink node, apply Find_Sink_Node on each of its children;
2. Otherwise push this sink node (*TOP*) into the stack by calling Push_Stk().

Push_Stk (StkTop, Ptr_Node)
StkPtr *StkTop; */* top of the stack */*
NodePtr *Ptr_Node; */* ptr of the node going to be pushed in */*

Usage : Given the top of the stack and the node ptr which is going to be pushed in, do push action on the stack and return the updated top ptr.

Algorithm :
1. Create an Stk structure;
2. Link its *Pnode* field to *Ptr_Node*;
3. Update *StkTop.*

NodePtr Pop_Stk (StkTop)
StkPtr *StkTop; */* top of the stack */*

Usage : free the top node in the stack and return its *Pnode* field value.

Algorithm :
1. Return the *StkTop->Pnode*;
2. Update *(*StkTop*) and free the original *Stk* structure.

void Display_Sink_Node (StkTop)
StkPtr *StkTop; */* top of the stack */*

Usage : Traverse all the nodes in the stack and print them out.

Algorithm :
1. From the top to the bottom of the stack, print out every entry's *Pnode->NID, Pnode->RC.r, Pnode->RC.c.*

Display_Node (TOP)
NodePtr TOP; */* top of the stack */*

Usage : Recursively traverse all the nodes in the Hash table in prefix order and print them out.

Algorithm :
1. If *TOP* is an sink node, print out *TOP->NID, TOP->RC.r, and TOP->RC.c* and return;
2. Otherwise if *TOP* has child(ren), print out *TOP->NID, TOP->RC.r, and TOP->RC.c* and apply Display_Node on each of its children.

Moment.c : Evaluate moments for each node in the netlist.

Interface : By including the header HDstruct.h.

External Routines : Routines defined here to be used outside;
Moment - evaluate up to *NM*-th moments for the present node and its downstream nodes.

Internal Routines : Routines used within this module;
Moment_Cal - Evaluate the *i*th order moment of a node and its downstream nodes;
DownStreamCurrent - Evaluate the *i*th order branch current of a node.

Routine(s)
void Moment (Node)
NodePtr Node;

Usage : Given an node ptr, evaluate up to *NM*-th moments for itself and all the downstream nodes.

Algorithm :
1. $i = 0;$
2. Call DownStreamCurrent() to calculate *i*th order branch current of *Node*;
3. Call Moment_Cal() to calculate the *i*th moment of *Node*;
4. Repeat 2 and 3 until $i = NM$.

void Moment_Cal (Node, i)
NodePtr Node; *\/* node to be evaluated */*
int i; */* ith moment to be evaluated */*

Usage : Given an node ptr and integer *i,* evaluate the *i*th order moment of the node itself and its downstream nodes.

Algorithm :

1. AWE approach to evaluate moments. Moments are evaluated in a top-down (root to leaves) fashion.

void DownStreamCurrent (Node, i)
NodePtr Node; */* node to be evaluated */*
int i; */* ith moment to be evaluated */*

Usage : Given an node ptr and integer *i*, evaluate the *i*th order branch current, which is the sum of all downstream currents and current from its capacitor.

Algorithm :

1. AWE approach is used here. Branch currents are evaluated in a bottom-up (leaves to root) fashion.

RC_Reduction.c : Do hierarchical tree reduction.

Interface : By including the header HDstruct.h.

External Routines : Routines defined here to be used outside
BottomUpReduction - `Reduce the given RC tree into an equivalent RC PI circuit.`

Internal Routines : Routines used within this module
Series_Reduction - Reduce an RC segment with a PI load into one equivalent PI load;
Branch_Mergence - Merge all downstream PI loads into one equivalent PI load.

Referred External Routines : Routines defined outside to be used within this module;
Push_Stk - create an node and push it into the stack;
Pop_Stk - free the top node in the stack and return its *Pnode* field value.

Routine(s)
void BottomUpReduction (StkTop)
StkPtr * StkTop;

Usage : Given the top of the stack, do bottom-up RC reduction. As a result, every node's *Up_AC, Down_AC, Up_PI, Down_PI, Up_TF* fields will be updated.

Algorithm :

1. Call Pop_Stk() to get the top node. Routine ends if the node is *Src->root*;
2. Do backward RC reduction on this node by calling Series_Reduction() until the branch root occurs;
3. Check if all the children of the branch root have been reduced. If not, go back to step 1;
4. Otherwise call Branch_mergence() to merge all the branches of the branch root and push the merged equivalent node into the stack. Go back to step 1.

void Series_Reduction (Nodel1, Node0)
NodePtr Node1; */* the former node in the series */*
NodePtr Node0; */* the latter node in the series */*

Usage : Given two serialized lumped segments, reduce them into a new PI load.

Algorithm : [1]
1. Use *Node0*'s admittance coefficients to evaluate *Node1*'s admittance coefficients;
2. Realize the elements of the new PI load;
3. Evaluate a stable transfer function for *Node1* from realized reduction.

void Branch_Mergence (Node)
NodePtr Node;

Usage : Given an node ptr, merge all its branches into a new PI load.

Algorithm :
1. Sum up every branch node's *i*th admittance *y[i]*, which is *Node*'s *i*th admittance *y[i]*;
2. Realize the elements of the new PI load.

DELAY.c : calculate node delay.

Interface : By including the header HDstruct.h.

External Routines : Routines defined here to be used outside
- Delay - Given the root of a netlist, evaluate the delay for all the nodes in the netlist.
- TestNodeDelay - Evaluate the delay for all the nodes in test node list.
- Elmore - Given an node, evaluate Elmore delay for itself and its downstream nodes.

Internal Routines : Routines used within this module
- Delay_Cal - Given an node, evaluate the 50% delay of it.
- V_dV - Given time *t*, evaluate *V* and *dV* for the given node at that moment.
- Td_ini - Given an node ptr, return the initial value for its delay.
- Search_Node - Given an node ptr and an node ID, try to find if the node is in the subtree with *Top* as root.

Newton_Ralphson - A well-know approach to solve nonlinear equations. Given a root boundary and equation forms, root will be returned.

Routine(s)
void Delay (Node)
NodePtr Node;

Usage : Given the root of a netlist, evaluate the delay for all the nodes in the netlist.

Algorithm :
1. Call Delay_Cal() to evaluate the delay of the root;
2. Apply Delay() recursively on the root's children.

void TestNodeDelay ()

Usage : Evaluate the delay for all the nodes in test node list.

Algorithm :

1. If *IsUseDefaultTestNode* setting is “no” in HIDE configuration file, build test node list *TestNode* from the test node list given in the configuration file;
2. Apply Delay_Cal() on each node in the list.

```
void Delay_Cal ( node )
NodePtr node;
```

Usage : Given an node, evaluate the 50% delay of it.

Algorithm :

1. If the parent node delay is available, use that value as the delay boundary of the present node's delay; otherwise call Td_ini() to get the boundary;
2. Call Newton_Ralphson() method to evaluate the node delay;
3. If *isRiseTime* setting is “yes” in HIDE configuration file, call Newton_Ralphson() method to evaluate the starting point delay and the end point delay.

```
void Elmore ( Node )
NodePtr Node;          /* node for Elmore delay evaluation */
```

Usage : Given an node, evaluate Elmore delay for itself and its downstream nodes.

Algorithm :

1. If *Node* is an sink node and it is not the child of the root, evaluate its Elmore delay = *parent Elmore delay + Node->RC.r * (Node->Rc.c + downstream capacitors)* and return;
2. If *Node* is the child of the root, Elmore delay = *Node->RC.r * (Node->Rc.c + downstream capacitors);*
3. If Node is not an sink node, nor the child of the root, Elmore delay = *parent Elmore delay + Node->RC.r * (Node->Rc.c + downstream capacitors);*
4. Apply Elmore() on each of its children.

```
void V_dV( t, V, dV, node, Vth, Order )
Real      t;
Real      *V;
Real      *dV;
NodePtr   node;
Real      Vth;
int       Order;
```

Usage : Given time *t*, evaluate *V* and *dV* for the given node at that moment.

Algorithm :

1. Due to different input signal forms, three branches are used to evaluate *V* and *dV* at the given moment. Basically, the method we use is symbolic formula, which is equivalent to convolution. However, we don't have to do convolution since it is very expensive (O(n^2)) and our input signals are step, ramp or exponential functions only, we can use symbolic formula to do in O(1) time.

Real Td_ini (Node)
NodePtr Node;

Usage : Given an node ptr, return the initial value for its delay.

Algorithm :

1. Apply D2M approach to initialize the node’s delay value.

NodePtr Search_Node (Top, NodeNumber)
NodePtr Top;
char *NodeNumber;

Usage : Given an node ptr and an node ID, try to find if the node is in the subtree with *Top* as root.

Algorithm :

1. Traverse the subtree with *Top* as root, and return the node ptr if the node with the name *NodeNumber* is found in this subtree, otherwise NULL is returned.

Real Newton_Ralphson (x1, x2, Xacc, f_df, Vth, node, Order)
Real x1;
Real x2;
Real Xacc;
void (*f_df) (Real, Real *, Real *, NodePtr, Real, int);
Real Vth;
NodePtr node;
int Order;

Usage : A well-know approach to solve nonlinear equations. Given a root boundary and equation forms, root will be returned.

Algorithm :

1. Use bisection method to refine the given root boundary;
2. Newton-Ralphson formulation: $\varphi(x) = x - \frac{f(x)}{f'(x)}$

Pole_Eval.c : Evaluate poles for each node.

Interface : By including the header HDstruct.h.

External Routines : Routines defined here to be used outside
Driver_Eval - Calculate a stable transfer function for driving point.
TopDownExpansion

Internal Routines : Routines used within this module
AdjacentNodeExpansion
PoleResidueEval
Hier_Moment

Routine(s)
void Driver_Eval ( )

Usage : Calculate a stable transfer function for driving point.

Algorithm : [1]
1. Please refer to reference [1] and Series_Reduction ().

void TopDownExpansion (node)
NodePtr node;

Usage : Propagate the transfer function from driver node to all the sink nodes, and meanwhile every node's poles will be evaluated.

Algorithm :
1. Call AdjacentNodeExpansion (*node, node->child*) to expand the transfer function from *node* to *node->child*;
2. Call TopDownExpansion (*node->child)* to recursively expand the transfer function;
3. If *node* still have children untouched , go back to step 1.

void AdjacentNodeExpansion (node0, node1)
NodePtr node0;
NodePtr node1;

Usage : Given upstream node *node0* and downstream node *node1,* evaluate transfer function for *node1*.

Algorithm : [1]
1. Refer to Theorem 2 of reference [1] on how to merge two transfer functions, keeping it Hurwitz stable.

void PoleResidueEval (node)
NodePtr node; /* node to evaluate poles and residues */

Usage : Given an node ptr, evaluate poles and residues of its transfer function.

Algorithm :
1. Given transfer function and admittance coefficients, it is easy to evaluate poles of the transfer function and also the residue. Depending on the polynomial order of the denominator, poles can be obtained with closed forms or by calling RealPole(), which uses Newton Ralphson method. Currently the program is using 2-pole model, so RealPole() is not used yet.

void Hier_Moment (node)
NodePtr node;

Usage : Given an node ptr, compute moments locally.

Algorithm :
1. Arbitrary order of the moments can be trivially obtained because we know the transfer function of the node. Just do fraction division.

Data_Output.c : output delay, risetime, Elmore delay or moments if needed.

Interface : By including the header HDstruct.h.

External Routines : Routines defined here to be used outside
Data_Output - Output customized delay results for each user-interested node.

Internal Routines : Routines used within this module
OutputAll - Output customized delay results for each node;
OutputTestNode - Output delay and rise time for each test node.

Routine(s)
void Data_Output ( )

Usage : Output customized delay results for each node or just delay and rise time for each test node.

Algorithm :

1. Due to the setting *"isOutputAll"* from HIDE configuration file, call *OutputAll()* or *OutputTestNode()* respectively.

void OutputAll (Node)
NodePtr Node;

Usage : Given an node ptr, output customized delay results for this node.

Algorithm :

1. If *"isElmore"* is set as "yes", output Elmore delay; otherwise output HIDE delay.
2. If "*isD2M"* is set as "yes", output D2M delay;
3. If *"isGenMoment"* is set as "yes", output its 0th to *NM*th moments;
4. Apply OutputAll on each of its children.

void OutputTestNode ( )

Usage : Output delay and rise time for each node in the test node list.

Algorithm :

1. Output delay for test node *i;*
2. If "*isRiseTime*" is set as "yes", output its rise time as well;
3. *i* becomes the next node in the test node list, until all the test nodes are visited.

TDWE.c : Time domain waveform evaluation.

Interface : By including the header HDstruct.h.

External Routines : Routines defined here to be used outside
Node_Eval - evaluate the output response for test nodes.

Internal Routines : Routines used within this module
F2T - transfer the output response from frequency domain to time domain.

Routine(s)
void Node_Eval ( )

Usage : evaluate the output response for test nodes.

Algorithm :
1. Apply F2T() on each test node.

void F2T (node, Order)
NodePtr node;
int Order;

Usage : evaluate the node output waveform from 0.0 sec till *tstop*.

Algorithm :
1. We use the same approach - symbolic formula, which we use in V_dV().

HSPICE_GEN.c : generate a HSPICE file for delay verification.

Interface : By including the header HDstruct.h.

External Routines : Routines defined here to be used outside
HSPICE_Gen - generate an HSPICE netlist file, which can be used for delay verification.

Internal Routines : Routines used within this module
write_Hspice - write RC netlist.

Routine(s)
void HSPICE_Gen (Top)
NodePtr Top; */* root of the netlist */*

Usage : Given the root ptr of the netlist, generate an equivalent HSPICE netlist file.

Algorithm :
1. Write the root's RC values in HSPICE file;
2. Call write_Hspice() to write RC netlist in HSPICE file;
3. Write stimulus information in HSPICE file;
4. Write test node in HSPICE file (.print...).

void write_Hspice (Node, Fspice)
NodePtr Node;
FILE *Fspice;

Usage : Given the root of the netlist and HSPICE handle *Fspice*, write RC netlist in HSPICE file.

Algorithm :
1. Write *Node*'s RC values in *Fspice*;
2. Apply write_Hspice on each of *Node*'s children.

Function_Lib.c : define common routines.

Interface : By including the header HDstruct.h.

External Routines : Routines defined here to be used outside
Error_exit - Print an error message corresponding to the given error code and exit;
OpenFile - Given a file name and open mode, return the file handle;
PrintUssage - Print usage information.

Routine(s)
void Error_exit (Error_Code)

Usage : Given an error code, print out the corresponding error message and program exits.

FILE * OpenFile (file, attribute)
char * file; *\/* given file name *\/*
char * attribute; *\/* open file mode *\/*

Usage : Given the file name and requested open mode, open the file and return the handle. Program will exit if file is not found.

void PrintUssage ( )

Usage : Call to print the helpful information on how to run HIDE.

Timer.c : evaluate program's running time.

External Routines : Routines defined here to be used outside
BeginTiming - Start clock;
EndTiming - End clock and return how much time the clock run.

Macros : macros used within this module;
#define MAXClockNum 5 /* Control the maximal number of clocks allowed simultaneously */

Variables : variables only used throughout this module;
struct rusage ru[MAXClockNum]; /* system resource usage statistics */
struct timeval start_time[MAXClockNum]; /* start time of each clock */
struct timeval end_time[MAXClockNum]; /* end time of each clock */

Routine(s)
void BeginTiming (*i)*
int i; /* clock ID */

Usage : Given a clock ID, start a clock and identify it using the ID.

```
double  Endtiming ( i )
int i;            /* Clock ID */
```

Usage : given a clock ID, end the clock and return how much time the clock run.

References

[1] Xiao-Dong Yang, C.K. Cheng, "*Hurwitz Stable Reduced Order Modeling for RLC Interconnect Tree*", IEEE/ACM ICCAD, Nov. 2000, p.p. 222-8.
[2] L. T. Pilage, R. A. Rohrer, "*Asymptotic Waveform Evaluation for Timing Analysis*", IEEE Trans. on CAD, Apr. 1990, vol 9, p.p. 352-66.