A DATA STRUCTURE FOR CIRCUIT NET LISTS

Steve Meyer

Independent Consultant • 2124 Kittredge Street, #125 • Berkeley, CA 94704

Abstract: A data structure for storing and processing electrical circuit net lists is described. The basic data structure is not new, but the version described here is novel in three specific ways. It adds separate structures (arrays) for cell type and I/O pad specific information, stores net lists defined in terms of primitive elements or cells as two superimposed symmetric incidence list form directed graphs, and separates primitive element input and output lists to allow signal flow traversal. This paper concentrates on computer program level implementation details and on various practical problems arising in circuit net list processing. Finally, the structure's construction cost and algorithmic efficiency is discussed.

Keywords: Data Structure Implementation, CAE Tool Building, Software Engineering.

1. Introduction

When it is possible to store an electronic circuit net list in terms of basic primitives or cells, there seems to be one data structure that is the storage scheme of choice in almost all situations. The purpose of this paper is to describe that data structure at the computer program implementation level. The basic data structure is not new and variations have been used in network partitioning for a number of years. The basic data structure is most clearly described by Fiduccia and Mattheyses [6]. They attribute the data structure to Schweikert and Kernighan [11], but in fact, even though Schweikert and Kernighan realized that it is physically incorrect to represent an N pin signal net as M, 2 pin nets where M is the number of pairwise combinations of the N pins (number of edges in the complete graph of N pins), they still stored the circuit net list as a connectivity matrix rather than as incidence lists. See [2] or [5] for a discussion of the storage of mathematical graphs. The data structure has surely also been implicitly used in many CAD programs (see [3] for example).

The circuit partitioning data structure has been improved to handle signal flow directionality and to allow the association of additional information with various circuit elements. The improvements increase its usefulness in problems other than network partitioning. This paper also defines C programming language[8] structure templates, presents an example C function, and discusses a number of implementation details.

The data structure is usable in layout (placement, routing, and partitioning), net list translation, design verification (cell interconnection rule checking), path analysis, and circuit expansion (compilation). In the case of circuit expansion, the data structure can be used to store the individual sub-circuits, and the final flattened network. The expansion process maps sub-circuit data structures into more flattened circuits. This data structure is not applicable to circuit design approaches that directly generate transistor level IC masks because such designs are not decomposable into non-overlapping primitives.

2. Data Structure Description

The basic idea is to store each signal net as a member of the net array, each cell (primitive) as a member of the cell array, and to use two interconnected lists. The net-pin list contains an entry for every pin connected to a given net and contains the cell array index to which that pin connects. The cell-pin list contains an entry for every pin connected to a given cell and contains the net array index to which that pin connects. Notice that the two lists are symmetric. The index values can also be pointers, but the use of pointers makes storage management somewhat more difficult.

The data structure is improved from the circuit partitioning version in three ways (see section 3 for C structure declarations). First, a cell type array is added. The type array contains one entry for every cell type used in the network and contains at least a cell type name field, an instance count field for consistency checking, and two ordered pin lists. One pin list gives the pin name and pin type for every cell type input pin, and one list gives the pin name and pin type for every type output pin. Remember there may be many pin types in addition to the simple input or output categories. Possibilities are: tristate, bidirectional, wired-and, etc. The pin lists are ordered according to the pin order used in the cell type's definition. This allows the processing of net list forms using positional pin connectivity definitions. Delay calculation programs can use the type array for generic cell type dependent delay values.

Second, the net-pin and cell-pin lists are broken down into two lists: one for cell and net inputs and one for cell and net outputs. By breaking the lists into two sublists, the circuit network has been converted into two superimposed directed graphs. Forward source to drain signal flow can be traced by traversing cell outputs, net inputs, net outputs, and then inputs of the next cell. Backward drain to driving source connectivity can be traced by traversing first input then previous element

output lists. Notice that any path contains alternating cells and nets.

The cell-pin list is ordered identically to the corresponding cell type pin list and each cell array entry contains the index in the type array of its type. When a new cell is added to the cell array, its pin list order is copied from the corresponding type pin list. It is sometimes beneficial to also keep the net-pin list ordered.

Third, an I/O pad signal array is added. The I/O signal array contains one entry for every off-module (off-chip, or off-board) connection and contains at least a signal name, an I/O pad type field, and the index in the net array of the connected net. This array allows special processing for I/O pads and is useful for storing temporary data during circuit expansion (flattening).

3. Data Structure Templates

A number of storage assumptions have been made in the following C programming language [8] structure definitions. It is assumed that all element names can be of arbitrary length and are stored in a common dynamic storage area. This allows each pin name to be stored exactly once and pointed to from 3 places. All arrays are assumed to be stored in dynamically allocated memory. In C they are still accessed as normal arrays. It is possible to change the array index **int** types to **short int** for circuits with less than 32K cells. The structure templates might look something like:

```
struct ctype_pin_t {              /* one list item per type pin */
  char *ctpnam;                   /* pin name */
  struct ctype_pin_t *ctpnxt;     /* next pin in order */
};

struct ctype_t {                  /* one element per type */
  char *ctnam;                    /* type name */
  int howmany;                    /* no. of instances of type */
  struct ctype_pin_t *ctipins;    /* pointer to list of input pins */
  struct ctype_pin_t *ctopins;    /* and to list of output pins */

  ... other problem specific fields ...
};
extern struct ctype_t *ctype;     /* array of allocated types */

struct cell_pin_t {               /* one list item per cell pin */
  char *cpnam;                    /* pin name */
  int cni;                        /* index in net[] of pin */
  struct cell_pin_t *cpnxt;       /* next pin */

  ... other problem specific fields ...
};
```

```
struct cell_t {                   /* one element per instance */
  char *cnam;                     /* instance name */
  int cti;                        /* index in ctype[] of instance */
  struct cell_pin_t *cipins;      /* header of input pin list */
  struct cell_pin_t *copins;      /* header of output pin list */

  ... other problem specific fields ...
};
extern struct cell_t *cell;       /* array of allocated instances */

struct net_pin_t {                /* one list item per net pin */
  char *npnam;                    /* name of cell pin */
  int nci;                        /* index in cell[] of pin */
  struct net_pin_t *npnxt;        /* unordered next pin on net */

  ... other problem specific fields ...
};

struct net_t {                    /* one array element per net */
  char *nnam;                     /* name of net */
  /* signal class can be (WIRE, WAND, WOR, ...) */
  char typflg;                    /* net's signal class */
  struct net_pin_t *nipins;       /* list of net drivers */
  struct net_pin_t *nopins;       /* list of net fan-out */

  ... other problem specific fields ...
};
extern struct net_t *net;         /* allocated  array of nets */
```

4. Data Structure Construction

First, read and sort by type name the cell type library if it is available. It usually contains less than a few hundred elements. If it is not available, the cell type array can be built incrementally as the cell array is built. Next, read either the cell list or net list depending on how the net list input file is coded. While reading either the net or cell lists, whichever is more convenient, build either the net-pin list or the cell-pin list, again depending on whichever is easiest. Finally build the other symmetric pin list.

On systems supporting dynamic memory reallocation (such as the UNIX realloc function [4]), the cell and net arrays can be allocated in small, a few hundred entry, pieces and the reallocation function can be used to expand the arrays incrementally when needed. The incremental allocation has almost no cost on computers that provide block memory move instructions. See section 8 below for construction times for circuit net list languages.

5. Example

The following routine processes each cell driven by a given cell. It is useful in delay calculation or transmission gate connectivity checking. Notice that cells may be processed more than once if more than one cell output drives the same signal net. It is written in C and uses the template structures defined above.

```
process_driven_cells(ci)
  int ci;
{
  register struct cell_pin_t *cpp;
  register struct net_pin_t *npp;
  int ni, dci;

  cpp = cell[ci].copins;
  while (cpp)
    {
     /* if unconnected cell pin do nothing */
     if ((ni = cpp->cni) != -1)
       {
        npp = net[ni].nopins;
        while (npp)
          {
           /* if unconnected net pin ignore */
           if ((dci = npp->nci) != -1)
             problem_specific_action(dci);
           npp = npp->npnxt;
          }
       }
     cpp = cpp->cpnxt;
    }
}
```

6. Practical Considerations

The removal of circuit elements during net list processing is quite frequently necessary. Many net list input files contain point nets (meaningless nets containing only one pin) usually caused by dangling wires in circuit schematics. If the net list input is net pin list based, point nets can be ignored as they are read. For cell based input forms, the existence of point nets is not known until the net list has been read. One good approach is to access the net list only through an index array. Point nets are then not put into the index. This index array can also be sorted to allow various types of cross reference listings to be produced such as a list of nets sorted by decreasing delay times. The cell and net arrays themselves cannot be reordered without updating the net-pin and cell-pin entry, cell or net index values.

Some net list problems require that all extra cells and nets that do not affect design outputs be removed. This process is sometimes called a "gate eater" and works by recursively deleting all cells whose outputs are all unconnected. One way to handle this problem is to first mark all deletable cells and then construct indices into the cell and net arrays for non-deleted cells. The various list elements should also have their net index or cell index fields set to unconnected when they point to deleted elements.

7. Use in a Scientific Discovery

This data structure was used in a program that analyses the connectivity of cell based layout benchmark circuits coded in YAL (Yet Another Language) [10]. The program used the data structure's ability to simultaneously traverse net pin and cell pin lists to discover one reason why one of the circuits used in the 1987 benchmarks [10] turned out to be easier to place than expected. In terms of active area, seventeen percent of the unexpectedly easy benchmark connected directly to I/O pad cells that are situated along the chip periphery. More specifically, the primary benchmark circuit called CIRCUITX with 833 instances, 904 nets, and using 2672 gates has 169 or 22.5 percent of its cells and 17.5 percent of its area connected to I/O pads. The larger and more difficult benchmark called RPROC has 9.7 percent of its cells and 11.3 percent of its area connected to I/O pad cells. A cell connects to an I/O pad cell if the two cells share at least one net. Cells connecting to I/O pads are easy to place because they can be located at the nearest point along the masterslice area without increasing wire length or congestion.

8. Cost of Data Structure Construction

The net list data structure can be constructed quite efficiently. If the cells or nets are sorted and reasonably efficiently coded in a file, the data structure can be built in about twice the time it takes to read every byte in the net list file or about the same time it takes to break elements into tokens. For more human readable net list description languages, the construction time is generally two to three times the tokenization time since the various lists must be sorted and element "reference before definition" must be handled.

See table 1 for a comparison of construction times for net lists coded in popular hardware description languages. The TDL test circuit is a medium sized flattened circuit used for testing gate array TDL connectivity patterns. The hierarchical circuit is a small signal correlator with 591 instances in 33 modules. The circuit contains 40 I/O signals. The build time in this case is the time to build all 33 module net lists. The glue logic circuit is a flattened gate array of 8400 gates that provides microprocessor interface logic. The instances are gate array macro cells. The memory board circuit is the flattened logic part of a memory board. The board net list contains simulator primitive level instances plus three instances for gate arrays and eight instances for the RAM.

The first two circuits are coded in both TDL [12] and SDL [13]. Notice that it takes more space to store the SDL circuit but data structure construction is faster. The TDL language requires more time because it is impossible to tell whether an element is a cell or net until the entire net list has been read. The third and fourth circuits are coded in the Valid system schematic compiler output language [9]. Since this language allows sized instances, the glue logic input contains 1232 instances that become 2403 in the final data structure. For the memory board 1828 sized parts become 3772 cells.

The first two circuits were both coded in two different languages and run on a standard IBM PC/XT using an 8088 micro processor with a 70ms. hard disk. The second two circuits were run on a diskless Sun 3/50 with a 68020 microprocessor. The tests were run on a lightly loaded network, and the time is total elapsed time. The CPU

utilization was always between 92 and 94 percent. It takes 29 seconds to just copy the memory board file.

Circuit	No. of Cells	Input Format	Size in Kilo-bytes	Com-puter	Token-ization Time in Sec.	D.S. Build Time in Sec.
TDL Test	895	TDL	36	8088	36	104
		SDL	46	8088	45	86
Hierar-chical	591	TDL	43	8088	40	77
		SDL	59	8088	51	73
Glue Logic	2403	Valid	467	68020	26	48
Memory Board	3772	Valid	1047	68020	58	99

Table 1

9. Algorithmic Efficiency

Once the net list data structure is constructed, it is time efficient in the sense that it allows direct implementation of the efficient graph theory algorithms [7, 5, 1]. The inclusion of both cell and net nodes increases the length of any path by a factor of two, but both types of nodes are required when path lists must include all cell-pin and net-pin identities.

The data structure is space efficient in the sense that incidence lists are a good way to store sparsely connected graphs such as circuit networks [7]. If only cell to cell connectivity with no requirement for pin or signal flow information is of interest, the net array and net-pin lists can be removed. The cell-pin list becomes a cell-to-cell incidence list. However, the inclusion of net nodes may actually reduce net list size since the size of the incidence list for large nets is reduced from the product of the fan-in and the fan-out to the sum of the same. Busses and clock nets both have high fan-in and fan-out. Notice that algorithms involving either forward or backward traversal require no searching since each net node is immediately available.

Acknowledgements

Zahir Syed made the original suggestion that I try to improve net list storage methods. John Sanguinetti and the designers at Ardent Computer provided difficult and interesting circuits that led to the data structure in its present form. The manuscript was prepared by A Micro Assist.

10. References

1. Aho, A. V., Hopcroft, J. E., and Ullman J. D. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
2. Baase, S. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley, 1978.
3. Diss, W. C., Ott, R., and Nelson, D. W. Integration of a hardware simulator into an IC design system. IEEE Proceedings of ICCAD-85, 1985, 158-160
4. Bell Telephone Laboratories. *UNIX Programmer's Manual Seventh Edition*. Vol. 1, Holt Rinehart, 1983, 275.
5. Even, S. *Graph Algorithms*. Computer Science Press, 1979.
6. Fiduccia, C., and Mattheyses, R. A. A linear-time heuristic for improving network partitions. Proceedings 18th Design Automation Conference, 1982, 175-181.
7. Hopcroft, J. E., and Tarjan, R. E. Efficient algorithms for graph manipulation. *Communications of the ACM*, 16, 6(1973), 372-378.
8. Kernighan, B. W., and Ritchie, R. M. *The C Programming Language*. Prentice hall, 1978.
9. McWilliams, T. M., and Widdoes, L. C. SCALD: structured computer-aided logic design. Proceedings 15th Design Automation Conference, 1978, 271-277.
10. Preas, B. Benchmarks for cell-based layout systems. Proceedings 24th Design Automation Conference, 1987, 319-320.
11. Schweikert, D., and Kernighan, B. W. A proper model for the partitioning of electrical circuits. Proceedings 9th Design Automation Workshop, June 1972, 57-62.
12. Szygenda, S. A. TEGAS 2 - Anatomy of a general purpose test generation and simulation system. Proceedings 9th Design Automation Conference, 1972, 116-127.
13. Van Cleemput, W. M. An hierarchical language for the structural description of digital systems. Proceedings 14th Design Automation Conference, 1977, 377-385.