High Level Approaches to VLSI Design

by

F. Mavaddat

Department of Computer Science
University of Waterloo
Waterloo, Ontario

Research Report CS-83-25

September, 1983

High Level Approaches to VLSI Design

F. Mavaddat

Department of Computer Science

University of Waterloo

Waterloo, Ontario

ABSTRACT

The complexity of VLSI design and the human limitations in handling this complexity are discussed. Structured and high level concepts of design, borrowed from software engineering, are applied to the handling of VLSI complexity. Emphasis is placed on several abstract levels of VLSI design. Objects of design at each level and some known results are discussed.

"we tell ourselves that what we can do once, we can also do twice and by induction we fool ourselves into believing that we can do it as many times as needed" E.W. Dijkstra*

Limits to Growth

The design of Very Large Scale Integrated (VLSI) circuits is a complicated and highly error prone process. Manufacturing complications are beyond the scope of this paper. Here we will only consider the complexity of logical design subject to the constraints of VLSI technology. We will demonstrate the need for a structured approach in order to increase design efficiency and designer productivity.

The complexity and communication requirements of a typical integrated circuit (IC) of today's technology is not unlike a road network covering the Los Angeles basin, i.e. a 100 x 100 km area with 200 m blocks [SEISTV79]. It is easy to calculate that the same urban density will cover an area of the size of the North American continent with the technologies envisioned for the not too distant future. Those familiar with programming activities, perhaps, can better appreciate the potential human errors which can be introduced in unstructured plans of this size and complexity. As a design becomes more complex, it is more difficult to conceptualize and the designers understand less of the interdependencies of its parts. The design iterations due to errors introduced at one stage of design and caught at a subsequent stage can reduce designer's productivity to unacceptably low levels. Such small productivities will lead to poor morale and may completely destroy the efforts [LOSCAD80].

Another limiting factor to unaided design is the required time. The productivity of a layout designer is estimated at 5 to 20 devices per day. At the average rate of 10 devices per day (some consider this figure optimistic) a microprocessor of today's complexity would

*From "Notes on Structured Programming", in *Structured Programming,* A.P.I.C. Studies in Data Processing, No. 8, Academic Press, 1972.

require some sixty man-years of effort to complete. Reported figures confirm this estimate [WEIVDM79, LOSCAD80]. Since this level of effort is difficult to manage, it means that the technology will (if it has not yet) outrun the manufacturer's ability to use it in a timely manner [LATVDM79].

Managing the Complexity

The road network analogy is a tangible indicator of VLSI design complexity. Planning a city of such large dimensions with detailed planning of every intersection is indeed a mind boggling task. Let us assume that we want to control the traffic lights at every intersection of this very large scale city in order to control the overall harmonious flow of traffic between any two points of this area, all simultaneously.

The right of way at every interaction can be represented by a binary variable. We will assume that every one of these variables contribute to the overall operation of the system in a significant way and a single error will fail the system requirements—a condition amply true for an IC design. The total number of binary control variables for an area the size of the Los Angeles basin with city blocks of 200 m is about 250,000. Compared to the number of bits in a typical Operating System (OS), this figure is roughly an order of magnitude smaller, yet the significance of every bit to the successful operation of any OS of any size is completely valid.

Computer specialists have been managing this complexity rather well for a number of years. We would like to investigate how the management of this complexity is achieved and what lessons have been learned.

The early experience of OS writers is almost similar to that of present day VLSI designers. The history of OS writing is full of failures and OS releases with a never-ending sequence of bugs found in them. Lessons learned from these failures resulted in the acceptance of a set of guidelines necessary for the successful writing of large software systems and the emergence of the discipline of software engineering.

By far the most important of these guidelines is that of structured and hierarchical design. The strategy in structured design is basically to divide and conquer. Through this strategy the subsystems of a system are identified and designed as independent parts and are then glued together to form the desired system. These subsystems must be further divided until a point is reached where the design of any leaf of the structure becomes a manageable task. The central assumption of this approach is the existence of subsystems with little interaction between them [SIMACO62]. Fortunately, digital hardware provides ample opportunity for this kind of division. In fact, with the ever increasing complexity of VLSI, it is expected that such large ICs will never possess many random logic elements. The geometric patterns seen with the naked eye on the surface of ICs is in itself proof that the area is divided (and subdivided) into a number of subsystems.

The need for dividing the system into a reasonable number of subsystems is due to "our inability to do much" [DIJNSP72] at any one time. The complexity of managing the whole task without a breakdown into a number of manageable and understandable subtasks is just beyond the capabilities of the human intellect. This is in line with other observations which indicate dramatic qualitative changes due to quantitative changes of two or three orders of magnitude.

Dividing the VLSI design process into a number of subtasks is being exploited in most present designs. The main reason for its ready acceptance is the earlier work in dividing digital systems into subsystems such as ALUs, memories, and control functions. These earlier divisions were independent of VLSI implementation or other mediums.

Another dimension of a structured approach is that of "layered" design. A divide and conquer approach is a necessary pre-requisite to managing complexity, however, for most large systems it is probably not sufficient. Dijkstra refers to the divide and conquer style without a "layered" discipline as the "approach via unordered modules" [DIJHOS71]. He argues that in this way "one makes a long list of all the functions of the operating system to be performed, for each function a module is programmed and finally these modules are glued together in the fervent hope that they will co-operate correctly and will not interfere disastrously with each others activity. It is this approach which has given rise to the assumed law of nature that complexity grows as the square of the number of program components." In the layered approach to writing operating systems, one starts with a given hardware configuration and covers it with successive layers of software, each creating a slightly more attractive and easy to program virtual machine.

It is this gradual ease of programming at higher abstract levels which is in sharp contrast to the approach of unordered modules, where adding new functions further complicates the design [DIJHOS71].

It is this approach of layered design which has not found enough attraction amongst VLSI designers, especially in industry, and will be our focus of attention in the rest of this paper.

Hierarchy of VLSI Design Levels

In the design of operating systems (or other large software systems) every level of software either completely hides one or more of the difficult and error prone features of the bare hardware or presents it in a more friendly and easy to use form to the next higher level.

Those who have experienced the software handling of interrupts from several sources are familiar with the complexity of this endeavour. As the first layer of software in the "THE" operating system [DIJSTM68] Dijkstra successfully hid these interrupts under the cover of friendlier objects called "semaphores" and presented a new view of the hardware which had as many virtual processors as needed by the processes active at any time. Above this level, the user never has to worry about the fact that his many processes are really running on a single processor. Our point in presenting this rather detailed example of software techniques in a report on VLSI design is to demonstrate, through systematic layered design, how a very difficult problem can be solved once and for all. It is this feature which prompted Dijkstra to predict an ever increasing friendly environment of design, through increased modules, rather than one which gets progressively worse. The question to address now is: In what way can the layering technique help in VLSI design, or in other words, what is the equivalent of a helping software layer in the hardware design process? To answer this we have to focus more on the nature and role of any layer.

At every layer the designer has many options subject to constraints in achieving his goal. A good design is the choice of those options, within ranges permitted by the constraints, which produce the "best" result. The goodness of the result is usually measured according to certain criteria. If one of the design options is independent enough of the others, and the designer is able to gain enough insight into the rules controlling that option, then he is in a good position to isolate that option from the others and treat it in an independent way. The treatment, in the case of software, is to program the rules of that option into a layer of software and relieve the designer of the burden of worrying about that option from that point on. In VLSI design any such option, whose rules are well understood, can be placed in a design aid package. Having access to this design aid tool the designer would no longer be concerned about that aspect of design and would only have to concentrate on the remaining options.

An accumulation of these design aid tools constitutes the backbone of a design automation (DA) system. Any designer having access to a design automation system will describe his design in terms of only those options left for him to decide. His high level design, which describes his choice of the remaining options and is expressed in a suitable notation, is passed to the design automation package. The DA package expands the design by adding its own choices for omitted options and produces a final design ready for implementation in silicon. The similarity between this process and that of a compiler, which accepts an algorithm expressed in a high level language and expands it to object code by adding detailed hardware options, has prompted many to name these DA packages Silicon Compilers.

The first layers to implement are those easiest to understand and automate, and are mostly of a clerical nature. As higher levels are reached, understanding of the underlying human design processes increases. There is considerable controversy over whether these layers could ever be automated in a reasonably successful way.

In VLSI Design the automation of the following layers are found to be reasonable steps toward hiding the more cumbersome design considerations from the designer and toward presenting him with a friendlier environment of options in which he can concentrate on more creative aspects of the design process. We will discuss these layers in the order which seems to be most logical to implement them.

While discussing each level we will present some of the issues relevant to the specification of design at that level, the requirements of a DA package for expanding the design at that level to the previous level, and we will discuss some directions of future work whenever possible.

Geometric Layout

At this level, the process of VLSI design is abstracted into the design of a number of templates with windows through which selected photoresist areas of circuit surface are exposed. All the processing know-how below this level is left to the process engineer. Early in the development of VLSI technology, designers were able to separate the processing know-how from that of area layout. This in turn leads to the complete separation of activities between the layout designers and process engineers. The abstract world of geometric layout is a multilayer universe of planar geometric objects subject to certain constraints for minimum width, minimum separation and so forth. These constraints are the design rules (DR). By announcing the DRs relevant to a particular technology the process engineers can guarantee the creation of diffusion, polysilicon, and metal paths with the necessary interconnections.

Initially, mask layouts were prepared by draughtsmen and were communicated to the mask generation systems using tedious entry methods such as coordinate digitizers. Recently, more convenient methods of layout design have been used or proposed. Two primary methods of geometric layout generation involve the use of "interactive graphics" and "layout languages". Both methods have been pursued by the universities and industry and there is some controversy over the suitablity of each method in meeting the requirements of layout design procedures. Most graphic languages are in fact plotter-driver extensions of some existing algorithmic language. Embedding graphic function extensions in a general purpose hihg level language (HLL) is one of the strong points of its application. The algorithmic nature of specification is particularly suitable for parametric specifications like that of an n-input gate or a transistor whose dimensions can be changed in order to adjust its speed. The execution of a HLL layout specification generates a representation of the layout in a data-base. LAP and GAP are examples of SIMULA based layout languages [WALGUM80]. NET is LISP-based [TERUGN82], EARL is APL related [KINEIC82], and CHISEL is an extension of the C language [KARCEP83].

Graphic languages for the specification of layout are basically graphics editors. The use of commercially available graphics systems reduces the design cycle time but lacks layout-specific features. Interactive graphics provides "instant plotting" which enables the designer to iterate through the design quickly. It provides a natural form for creating geometric shapes and placing them conveniently on a design sheet. Interactive graphics also allows convenient ways of pointing to the design, usually through an appropriate pointing device such as a light pen, a mouse, or a tablet. ICARUS [FAIIII78] and IGS [INFIGS78] are examples of interactive graphic systems specifically designed for IC layout purposes. Advocates of the graphic approach point to the non-intuitive nature of geometric specification in algorithmic languages, the reluctance of layout designers to program in a HLL, and the inability of HLL design specification to help with design rule verification [WALWID80]. On the other hand, supporters of the HLL design of IC layouts emphasize the flexibility of this approach in the parametric representation of layout and the inability of graphical input for the specification of behavior [AYRISL79].

The complementary nature of the two approaches resulted in the proposal of the SAM system which combines the best of both in a single package [TRICGL81]. SAM provides the user with a vertical split viewing window. On one side the program view of the design being edited is shown. On the other side the graphics view of it is displayed. Any change in either window is reflected immediately in both windows.

Symbolic layout

Geometric layout can be separated into two concurrent activities: those of planar layout of functions and interconnections, and detailed consideration of geometric rules. Both of these activities are subject to constraints imposed by the process technology. Planar layout is constrained by the limitations of connectivity using only one or two metal layers and the high cost of communication in terms of speed and silicon area. The mapping of a circuit graph into a planar layout is a difficult task in need of considerable creativity.

Detailed geometric considerations are determined by the process technology and are a set of rules to be observed by the designer regarding minimum widths, and separations. If the enforcement of these rules, which are often well defined, is programmed as a layer of software separating the designer from the manufacturing process, then the designer can concentrate on the more creative activity of planar layout. This new environment of layout, in which the designer is not subject to the consideration of design rules, is referred to as symbolic layout. Therefore, symbolic design incorporates a simplified notation for defining circuit topology without any immediate concern for the geometrical details of the mask. Such a symbolic representation is also more suitable for design verification purposes. This independence from fabrication considerations also allows for design portability.

As with geometric layout, a symbolic layout can also be specified using either textual or graphical means. Textual formats are based on the assumption that a geometric layout can be scaled to a grid structure [GIBSSL76, BEYDMB82]. Once scaling into small rectangular areas is done, a distinct symbol can be used to represent each kind of area.

In graphical approaches coloured lines are used to represent the interconnections in each mask and the circuit property is usually defined through the cross over of lines of different colours. The use of graphical techniques in symbolic representation dates back to the work of Williams at MIT on STICKS [WILSNA77, WILSGC78]. This work subsequently received widespread acceptance through the classical work of Mead and Conway [MEAIVS80]. STICK diagrams are easy to read and a pleasure to design with. Their power becomes apparent in their application to the NMOS fabrication process. It is more difficult to apply them to other technologies. Elmasry recently proposed a technique for the representation of bipolar technology [ELMSLN83]. This technique uses both the colour line drawings and a small number of symbols to specify certain intersections of coloured lines.

Textual specification does not require any special purpose colour graphic terminal. Portions of large systems can be printed on regular printers and cut and taped together in order to form a single large system. Textual specification also permits the designer to have control over the width and height of the elements of a device by repeating a symbol horizontally or vertically as needed. This is a useful control on the performance of a device by trading off between speed and power. Matrices of alphanumeric symbols representing layout data are not very easy to read or, for that matter, to debug.

Functional Arrays Level

The central activity of symbolic layout of digital circuits is concerned with the creation and interconnection of switches. Again, this task has two orthogonal or independent components. The first component deals with the design of switches and the topology of their interconnections in order to realize a particular logic function. The second component tries to map the switches and their interconnections into a plane subject to the interconnection constraints of the technology under consideration. As with the previous two levels, if we are able to delegate one of these two activities to a new level of software, then the designer only has to deal with the other component and can therefore concentrate on more creative aspects of design.

Again, the component to be automated is the one whose procedure and objectives are better defined. The choice is to automate the layout, and this is not necessarily the obvious choice. Layout problems are very hard [SAHCDA80], and one has to settle for heuristic approximations. On the other hand, they are an ever present problem in all levels of design above that of symbolic layout, and ignoring them at this level will only aggravate the problem at higher levels.

Once the burden of layout is removed, the designer is left with the task of interconnecting a series of two state switches (transistors) in order to realize the function of his design. Design activities at this level are known as the functional array design. The design of VLSI systems at the functional array level has not received enough attention. Designs at this level are more efficient than those of the more accepted levels based on AND or OR gates [CARMLD72]. This level is an unlikely candidate for the design of total systems. It can be effectively used for the design of those parts of a system in which the layout efficiency is important and a simple AND/OR based design may lead to an inefficient design. A particularly useful scheme for the optimal layout of CMOS functional arrays is reported in [UEHOLC79]. Here, the CMOS implementation of logic functions results in an array of series-parallel switches to which graph theoretical results can be applied for optimal layout.

Logic Gates Level

This level and those above it were the recognized levels of digital systems design prior to the advent of integrated circuits. The building blocks of this level are the familiar logic gates and it is the designer, through the proper interconnection of these gates, who must realize his system. Design entry at this level has enjoyed considerable recognition in the past and a large number of systems based on this concept have been designed and implemented in academic and industrial environments for internal and external use [BEREED80, ODAPSI81, SAICSL81]. The underlying significance of gate level design, as the next level after the functional array, may not be immediately obvious. To appreciate the kind of abstraction used we have to remember that functional arrays are in fact series-parallel networks of switches. It is not difficult to see that the AND and OR gates are, respectively, the abstract representations of the "series" and "parallel" branches in the series-parallel graphs.

Widespread acceptability of gate level logic design can be attributed to at least three factors. The first, and probably the foremost, is the applicability of Boolean Algebra and the accompanying simplification techniques. The second is the widespread availability of standard logic families which has enormously simplified the task of logic design. The third is the fact that through logic gates the designer is freed from the concerns of physical implementation. He is able to proceed with the manipulation of his design and in the end, with a minimum amount of effort (often simple change of definition between positive and negative logic), implement his design using available physical gates off the shelf.

Recently, two new reasons have also attracted the VLSI designer into this level of design activity. The first is the promise of VLSI based PLA design which greatly enhances the appeal of designs expressed in terms of logic functions. So far, considerable effort has been focused on the efficient implementation of PLAs in silicon [AYRSCH79, KANAPS81, PAIOPA81, SUNCAD81, and WILHCB81]. The second reason is the widespread acceptance of gate arrays as a viable alternative to the headaches of fully custom-designed circuits [GRADGA82].

Similar to design entry at other levels, logic gates can also be entered either through the use of special purpose textual languages or through the facilities of a graphics editor. In both cases the usual reasons for preferring one method to another are valid. The use of hierarchical descriptions in both forms are very common.

Register Transfer Level

It is the common experience of many logic systems designers that their design can be broken into two components: that of a data path and a sequence controller. The microcode level of a computer system is a good example of this claim. Any such data path is a closed network of registers and ALUs through which the data is transferred under the signalling operations of a sequence controller. This observation, which successfully describes the internal operation of many digital systems of reasonable complexity, has prompted many into suggesting its use for modelling digital systems. At least one manufacturer has introduced a line of products on the basis of these concepts.

The acceptance of register transfer (RT) level design is still limited and more work must be done prior to its serious acceptance by those who apply CAD Technology. Some recent work in the direction of formalizing the RT level representation is promising and may help the RT level as Boolean Algebra did for gate level design [HAFFMS83, GORMRT82]. Many textual languages are proposed for RT level specification. Some of these languages are non-procedural and closely reflect the separation of a system into a data path and sequence controller. Others follow more closely the traditional concepts of procedural languages. Currently at the University of Waterloo, we are working on the development of an RT language with strong algebraic and graphical features. Its graphical version will be used as a hierarchical input language to a CAD system.

Summary and Conclusions

We discussed the inherent complexity of VLSI and pointed to another system of similar complexity, with similar initial problems, which has enjoyed the power of a disciplined design method for a number of years with impressive results. This was followed by an enumeration of several levels of design, along with a discussion of their powers and limitations.

The price for this increased ease of design is paid in terms of some inefficiency at the silicon level. Considering the ever increasing cost of the designer and the gradual decrease in the cost of silicon real estate, this is probably a penalty which can be tolerated without much grief. Finally, the high level design of VLSI circuits will resemble that of software through the need for a mixed mode of levels. A mixed mode design will allow the designer to get close to the silicon for those parts of a system in which silicon efficiency is crucial and to stay at higher levels when designer efficiency is more important.

Acknowledgements

This research was supported by grant number A5515 of the Natural Sciences and Engineering Research Council of Canada.

References

AYRISL79: Ayres, R., "IC Specification Language," Proc. 16th Design Auto. Conf., San Diego, Calif., June 1979.

AYRSCH79: Ayres, R., "Silicon Compilation—A Hierarchical use of PLAs," Proc. 16th Design Auto. Conf., San Diego, Calif., June 1979.

BEREED80: Bering, D. E., "The Electronics Engineers Design Station," Proc. 17th Design Auto. Conf., Minneapolis, Minn., June 1980.

BEYDMB82: Beyls, A. M., Hennion, B., Lecourvoisier, J., Mazare, G., and Puissochet, A., "A Design Methodology Based upon Symbolic Layout and Integrated CAD Tools," Proc. 19th Design Auto. Conf., Las Vegas, Nevada, June 1982.

CARMLD72: Carr, W. N., Mize, J. P., *MOS/LSI Design and Application,* McGraw-Hill Book Co., 1972.

DAHSPR72: Dahl, O.J., Dijkstra, E.W., Hoare, C.A.R., *Structured Programming,* A.P.I.C. Studies in Data Processing, No. 8, Academic Press, 1972.

DIJHOS71: Dijkstra, E.W., "Hierarchical Ordering of Sequential Processes," Acta Informatica 1, pp. 115-138, Springer-Verlag, 1971.

DIJNSP72: Dijkstra, E.W., "Notes on Structured Programming," in *Structured Programming,* A.P.I.C. Studies in Data Processing, No. 8, Academic Press, 1972.

DIJSTM68: Dijkstra, E.W., "The Structure of the 'THE'-Multiprogramming System", Comm. ACM, Vol. 11, No. 5, May 1968.

FAIIII78: Fairbairn, D.G., Rowson, J.A., "Icarus: An Interactive Integrated Circuit Layout Program", Proc. 15th Design Auto. Conf., Las Vegas, Nevada, June 1978.

GIBSSL76: Gibson, D., Nance, S., "SLIC—Symbolic Layout of Integrated Circuits", Proc. 13th Design Auto. Conf., San Francisco, Calif., pp. 434-440, June 1976.

GORMRT82: Gordon, M., "A Model of Register Transfer Systems with Application to Microcode and VLSI Correctness", CSR-82-81, Univ. of Edinburgh, Dept. of Comp. Sci., March 1981, Revised May 1982.

GRADGA82: Gray, J.P., Buchanan, I., Robertson, P.S., "Designing Gate Arrays Using a Silicon Compiler", Proc. 19th Design Auto. Conf., Las Vegas, Nevada, June 1982.

HAFFMS83: Hafer, L.J., Parker, A.C., "A Formal Method for the Specification, Analysis, and Design of Register-Transfer Level Digital Logic", IEEE Trans. on Computer-Aided Design, Vol. CAD 2, No. 1, Jan. 1983.

INFIGS78: Infante, B., Bracken, D., McCalla, B., Yamakoshi, S., Cohen, E., "An Interactive Graphics System for the Design of Integrated Circuits," Proc. 15th Design Auto. Conf., Las Vegas, Nevada, June 1978.

KANAPS81: Kang, S., VanCleemput, W.M., "Automatic PLA Synthesis from DDL-P Description," Proc. 18th Design Auto. Conf., Nashville, Tenn., June 1981.

KARCEP83: Karplus, K., "Chisel An Extension to the Programming Language C for VLSI Layout," Report No. Stan-CS-82-959, Dept. of Comp. Sci., Stanford University, Feb. 1983.

KINEIC82: Kingsley, C., "Earl: An Integrated Circuit Design Language," Technical Memo #4710, Dept. of Comp. Sci., California Inst. of Tech., Feb. 1982.

LATVDM79: Lattin, W., "VLSI Design Methodology, The Problem of 80's for Microprocessor Design," Proc. 16th Design. Auto. Conf., pp. 548,9, San Diego, Calif., June 1979.

LOSCAD80: Losleben, P., "Computer Aided Design for VLSI," in *Very Large Scale Integration VLSI*, edited by D.F. Barbe, Springer-Verlag, 1980.

MEAIVS80: Mead, C., Conway, L., *Introduction to VLSI Systems*, Addison Wesley, 1980.

ODAPSI81: Odawara, G., Kurishima, S., Aoyama, H., Kanaya, Y., "PAC-CIP An Interactive Logic Design System," Proc. 18th Design Auto. Conf., Nashville, Tenn., June 1981.

PAIOPA81: Paillotin, J.F., "Optimization of the PLA Area," Proc. 18th Design Auto. Conf., 1981.

SAHCDA80: Sahni, S., Bhatt, A., "The Complexity of Design Automation Problems," Proc. 17th Design Auto. Conf., Minneapolis, Minn., June 1980.

SAICSL81: Saito, T., Uehara, T., Kawato, N., "A CAD System for Logic Design Based on Frames and Demons," Proc. 18th Design Auto. Conf., Nashville, Tenn., June 1981.

SEISTV79: Seitz, C.L., "Self-Timed VLSI Systems," Proc. Caltech Conf. on Very Large Scale Integration, Pasadena, Calif., Jan. 1979.

SIMACO62: Simon, H.J., "The Architecture of Complexity," Proc. of the American Philosophical Society, Vol. 106, No. 6, Dec. 1962.

SUNCAD81: Suwa, I., Kubitz, W.J., "A Computer-Aided-Design System for Segmented-Folded PLA Macrocells," Proc. 18th Design Auto. Conf., Nashville, Tenn., June 1981.

TERUGN82: Terman, C.J., "User's Guide to NET, PRESIM, and RNL/NL," M.I.T. Laboratory for Comp. Sci., July 1982.

TRICGL81: Trimberger, S., "Combining Graphics and a Layout Language in a Single Interactive System," Proc. 18th Design Auto. Conf., Nashville, Tenn., June 1981.

UEHOLC79: Uehara, T., VanCleemput, W.M., "Optimal Layout of CMOS Functional Arrays," Proc. 16th Design Auto. Conf., San Diego, Calif., June 1979.

WALGUM80: Walker, H., "GAP User Manual," VLSI Document V030, Dept. of Comp. Sci., Carnegie-Mellon University, Feb. 1980.

WALWID80: Walker, H., "Why IC Design is NOT Like Programming," VLSI Document V043, Dept. of Comp. Sci., Carnegie-Mellon University, Oct. 1980.

WEIVDM79: Weimann, W., "VLSI Design at Motorola," unpublished presentation at 1979 Design, Automation Workshop, East Lansing, MI., Oct. 1979.

WILHCB81: Williams, D.D., "A Hardware Compiler for Boolean Logic Function," Memo No. 81-51, Dept. of E.E. and Comp. Sci., Massachusetts Inst. of Tech., June 1981.

WILSGC78: Williams, J.D., "Sticks—A Graphical Compiler for High Level LSI Design," National Computer Conf., pp. 289-295, 1978.

WILSNA77: Williams, J.D., "Sticks—A New Approach to LSI Design," M.I.T., Dept. of E.E., MS Thesis, May 1977.