Integrating UAHPL–DA Systems with VLSI Design Tools to Support VLSI DA Courses

Sadiq M. Sait

Abstract—**Because of rapid growth in areas related to digital design automation (DA) of very large scale integration (VLSI) systems, it has become necessary to introduce related courses into the university curriculum. In order to support effective teaching and laboratory courses, it is essential to have a complete operational environment established with the help of state-of the-art tools. In this paper, we explain the establishment of such an environment, which is accomplished with the integration of two systems: 1) a DA system which automatically produces VLSI layouts of digital systems modeled in Universal Hardware Programming Language (UAHPL); and 2) a set of VLSI tools, which in addition to several other functions can be used for simulation and verification of layout designs. Compared with other approaches, the integrated DA system provides a very simple user interface, fast turnaround time, no restriction on the final structure of the layout, and simulation and verification of all phases of design. The new environment, called UAHPL-based VLSI DA, is excellent for teaching and research at universities.**

I. Introduction

SCIENTIFIC advancements in certain directions lead to the creation of new disciplines. This forces the updating of curricula, and the introduction of new courses and teaching environments at universities. In recent years, one such area that has seen rapid growth is related to the design automation (DA) of digital systems and very large scale integration (VLSI).

With the increase in complexity of electronic circuits and the tremendous advancement made in the field of VLSI, it has become evident that DA is the exclusive viable way to handle digital system design and VLSI implementation. The number of components and their interconnections have been growing exponentially. Without DA, the design effort and possibility of errors would also grow at the same rate. Thus, the reason for our interest in DA research arises from its direct application to the area of labor-intensive digital integrated circuit (IC) design.

In order to support teaching and research in VLSI DA, it is essential for universities to have a DA environment that supports all phases of digital design from digital system modeling to layouts for fabrication. The environment must provide a framework in which a student learns disciplined, structured design techniques and obtains a complete view of the overall design process [1], [2].

The focus of this paper will be to present the Universal Hardware Programming Language (UAHPL) [3]-based VLSI DA environment. The basic components of the UAHPL–DA system required for VLSI DA are discussed. A subset of tools from the toolkit "VLSI Design Tools, Release 3.1" [4], distributed by the Northwest Laboratory of Integrated Systems at the University of Washington, is also presented. The combination of the UAHPL tools and the VLSI toolkit will establish an environment for automatic implementation of IC's modeled in UAHPL.

The paper is organized as follows. Section II discusses the salient features of the language and a sample UAHPL model of a digital system. It also highlights the multistage UAHPL compiler and the functions of the various stages. Section III briefly discusses the subset of VLSI design software taken from the toolkit that is required by the UAHPL–DA system. Section IV is dedicated to a general discussion pertaining to the acquisition, installation, hardware requirements, and use in VLSI DA courses. Some research projects that were accomplished with the UAHPL-based VLSI DA system are also mentioned. Section V summarizes the paper.

Manuscript received March 1991; revised September 1991.

The author is with the Department of Computer Engineering, King Fahd University of Petroleum and Minerals, Dhahran, 31261 Saudi Arabia.

IEEE Log Number 9203280.

II. UAHPL–DA System

DA tools primarily replace the designer in tasks that are tedious, well understood, and where no design decisions are to be made. They assist the designer in evaluating the merits of various design alternatives and in verifying the correctness of design. They also help in improving quality and productivity.

Integration of various hardware design tools into a complete design automation system enables designers to gracefully interact and move from one design phase to another. A fully integrated DA system can smooth on and speed up the flow of digital system design from specification to manufacture. It also provides an excellent teaching environment for DA of VLSI, a discipline that has been introduced in the undergraduate/graduate curricula of several universities [1], [2].

The UAHPL–DA system uses a high-level computer hardware description language (CHDL) called UAHPL to model the complex digital system to be implemented [3]. A multistage, multiapplication compiler which supports a wide spectrum of design and test activities has been implemented for UAHPL [5]. The design described by the user in UAHPL is translated automatically into internal tables by processors in different stages. These tables are used for various activities of the IC design process. This approach of having a single high-level description for all activities of the design process which include modeling, simulation, testing, implementation, etc. reduces the possibility of design errors. The following subsections will discuss the language UAHPL and the structure of the UAHPL–DA system.

A. Universal AHPL (UAHPL)

AHPL (a hardware programming language) is a register transfer-level language (RTL) which has been used for almost two decades [6] as an educational tool for communicating concepts of digital systems and computer design. The language is very simple and easy to learn, extremely powerful for modeling, and is sufficient to express the description of both simple and complex digital systems. Such a range spans combinational circuits to parallel processors and data flow machines. However, AHPL lacks certain constructs and features which have little pedagogical significance, but are necessary for efficient realization and testing of digital systems. UAHPL, an enhancement of AHPL, was designed after reviewing the requirement of a wide spectrum of design and test activities in various environments. This language gives more freedom in the choice of register types, clocking options, and bus types. It accommodates pass transistors, wired-OR gates, temporary registers, and other elements which are useful for an efficient realization of a circuit in any desired technology [5].

The UAHPL language, for the merits mentioned above and the fact that its compiler and simulator are easily available free of charge to all academic institutions in and out of the U.S., is a good candidate for the hardware description front-end. Moveover, familiarity with one hardware description language would enable one to migrate to another language such as VHDL with minimal effort.

In UAHPL, sequential automata are described as MODULES. Iterative combinational networks, such as adders, decoders, etc., can conveniently be described as combinational logic units (CLU's). Circuits with memory but no sequential control such as shift registers, counters, etc., may be described as functional registers. This classification allows one to model and use available digital IC's or predesigned layouts from the cell library in a convenient way. The module description contains three parts: declaration, procedural, and nonprocedural.

In the declaration part, registers are declared as MEMORY. INPUTS, OUTPUTS, and BUSES are communication lines used among the modules of the same system. EXINPUTS and EXBUSES are used for communication with other systems or the outside world. BUSES and MEMORY elements are local to the module. CLUNIT and FNREG are declarations of CLU and functional registers, to be incorporated into the module.

The procedural part (describing the state machine sequence) is divided into numbered steps. Each step takes one clock period to execute. A step may have zero or more transfer ' <= ' or connection ' = ' statements, followed optionally by a conditional or unconditional branch ' => '. The step numbers and branch statements define the sequencer or the control unit of the module. These can be implemented by means of flip–flops and logic gates.

The nonprocedural part follows the keyword ENDSEQUENCE. Statements in this part are always active regardless of the state of the control sequencer.

Only operators that have a direct hardware correspondence such as Boolean And '&', Or '+', Exclusive Or '@', and Complement ' ~ ' are permitted in UAHPL. APL [7]-type vector notation for Boolean reduction, concatenation, and bit selection are used. UAHPL does not have primitive operators for addition, subtraction, incrementing, decoding, etc. The reason is that, depending on engineering tradeoffs, these devices may be designed in different ways. For example, we may have a serial adder, a ripple carry adder, or a carry lookahead adder. Rather than restricting the choice to a few built-in options, UAHPL provides the facility to describe such devices as CLU's.

```
MODULE    : SEQSEL.
MEMORY    : A[4];B[4];C[4];F[4].
EXINPUTS  : X[4];Y[2];P;Q;CLOCK;RESET.
BUSES     : BUS1[4].
EXOUTPUTS: Z[4].
CLUNITS   : DCD[4]<: DCDER{4}.
CLUNITS   : INC[4]<: INCRE{4}.

BODY
SEQUENCE:CLOCK

1  =>(P,Q,~P&~Q)/(2,4,1).
2  =>(P)/(2).
3  A<=X; B<=A; C<=B;  =>(1).
4   F<=INC(BUS1);     =>(1).

ENDSEQUENCE
  CONTROLRESET(RESET)/(1);
  Z=F;
  BUS1=(A!B!C!(|1,1,1,1|))*DCD(Y).
END.

CLU: INCRE(X) {I}.
INPUTS: X[I].
OUTPUTS: Y[I].
 BODY
   FOR J=(I-1) TO 0 STEP -1
     CONSTRUCT
      IF J=I-1 THEN Y[J]=~X[J].
         ELSE Y[J]=X[J]@(&/X[J+1:I-1])
      FI
   ROF.
 END.
```

Fig. 1. UAHPL description of SEQSEL.

B. A Design Example

The example chosen is a sequential circuit SEQSEL. It illustrates several basic constructs and features of the language. Its UAHPL description is shown in Fig. 1. This is the model of a system that receives a four-bit vector from an external source, selects one among three such vectors or a constant vector, and displays that one. The circuit contains four 4–bit registers and accepts a 4–bit vector as input data. Registers A, B, and C act as a stack to save new input. The data at the output is selected from one of the registers or binary vector $|1,1,1,1|$ by the 2–bit external input Y. The vector chosen is stored in register F, an output holding register, until a new selection is made. Two additional inputs, P and Q, indicate when the input vector and selecting signals are valid.

In the declaration section, delimited by keywords MODULE and BODY (refer to Fig. 1), the size of all memory elements

(registers) and CLU's such as decoder (DCD) and incrementer (INC) are given. EXINPUTS such as CLOCK and RESET, EXOUTPUTS, and BUSES are also specified.

The numbered steps between the keywords SEQUENCE and ENDSEQUENCE form the procedural part defining the state of the sequential machine. In this part, a statement is active only when the machine is in the corresponding step. A statement may be a transfer ' <= ', connection ' = ', or a branch ' => '. The destination of a transfer statement is always a memory element, and that of the connection is a nonmemory element like a bus or a set of output lines. Connection is active throughout the corresponding control step, whereas transfer is assumed to take place at the trailing edge of the clock in that step.

Statements appearing after the keyword ENDSEQUENCE are always active and form the nonprocedural part. In this segment, the CONTROLRESET statement gives the step number to which the circuit will reset when the line RESET is activated. The other two statements in the nonprocedural part of the given example are for making permanent connections.

The example uses two CLU's DCD (a 4–bit decoder) and INC (a 4–bit incrementer). The identifier following the delimiter ' <: ' gives the generic name of the CLU. If the CLU has already been compiled and its layout exists in the cell library, then it is not necessary to describe it again; otherwise, it must be described explicitly. In this example, the generic CLU INCRE is described but the DCDER is not. The number enclosed in curly brackets is used as a parameter, the use of which is explained below.

The generic CLU description is used as a template by the compiler to generate copies of combinational circuit. The Algol-like syntax chosen for the CLU description is convenient for describing iterative networks. The parameter enclosed in curly brackets allows the compiler to generate CLU's of varying sizes from the same template. In this example, the parameter I is used to specify the bit size of the INCRE unit. To do the incrementing, the least significant bit is complemented, the other output bits are obtained by EXclusive ORing the corresponding input bit with the logical AND of all the lower significant input bits. FOR and IF statements are compiler directives which are used by the compiler to generate iterative networks.

Fig. 2 is another example of a CLU in UAHPL. The AND '&' symbol is used to represent a passgate in MOS. The equivalent circuit of the model is a chain of four pass transistors. Nesting of CLU's is possible, and this chain can be used to make more complex circuits such as a multiplexer. This digital circuit is used to illustrate various applications of the "VLSI Design Tools" that are discussed in Section III.

C. Compilation and Logic Synthesis

The block diagram of the UAHPL–DA system is given in Fig. 3. The compilation process is divided into stages [5]. The first stage, called Stage 1, accepts UAHPL circuit description, performs syntax analysis and semantics checking, and decomposes the source text into a tabular representation of the circuit. In all, fourteen tables are produced.

```
CLU: PASS4(A) {I}.
INPUTS: A[I], B[I].
OUTPUTS: Y.
 BODY
   FOR J=1 TO I
     CONSTRUCT
     A[J+1]=A[J]&B[J]
   ROF
     Y=A[I+1]
 END.
```

Fig. 2. UAHPL Model of CLU PASS4.

The Stage 2 compiler uses the tables generated by Stage 1 to create the interconnection list; that is, to synthesize the circuit. This list is the actual gate-level description of the digital circuit expressed in UAHPL.

Figs. 4 and 5 show the internal structure of the linked list, comprising GATELIST and its corresponding IOLIST produced by Stage 2 for a circuit. All nodes of the circuit are stored as records in the GATELIST. The first cell in Fig. 4 is the gate number, and the second specifies the type of element (for example AND, OR, NAND, D flip–flop, etc.). ILINK, OLINK, SYMLNK, and SIGINP are pointers to IOLIST (Fig. 5). Each node of IOLIST has four elements. The first element is an index pointed by either ILINK, OLINK, SYMLNK, or SIGINP. The second and third fields are information elements, and the fourth is a pointer to the next IOLIST record which contains similar information. The fourth field is a zero if there is no successor record.

The pointers of the element nodes mentioned above point to the IOLIST. For example, element 1 of Fig. 4 has gate-type 4509, which is a D flip–flop. Its ILINK, OLINK, SYMLNK, and SIGINP point to Entries 2, 157, 1, and 381 of the IOLIST, respectively. Following the IOLIST Record 2 and the succeeding record (7) pointed by it, one can find that Elements 19, 8, 16, −1 (Vcc), and −1 (Vcc) are connected to the inputs of Element 1, since it is referred to by ILINK; the output of Element 1 is connected to Element 11, which is of type 4201 (AND gate). The circuit for the segment explained above is shown in Fig. 6.

D. Stage 3 Processors

The outputs of the first two stages are technology independent, and are common to all applications of the compiler. These are further manipulated by various Stage 3 processors, to produce outputs pertaining to specific applications or activities. Currently available CAD tools, marked as Stage 3 processors in Fig. 3, provide an environment for modeling, synthesis, simulation, design, testing, and automatic layout generation.

Stage 3A performs functional simulation at the register transfer level [8], [9]. It uses Stage 1 tables and simulator directives to produce simulation output. If errors are detected in the output, then the UAHPL circuit description must be modified. This is shown by the upward arrow in Fig. 3.

Stage 3B is a test sequence generator [10]. Heuristics of the program are guided by the fact that UAHPL partitions a circuit

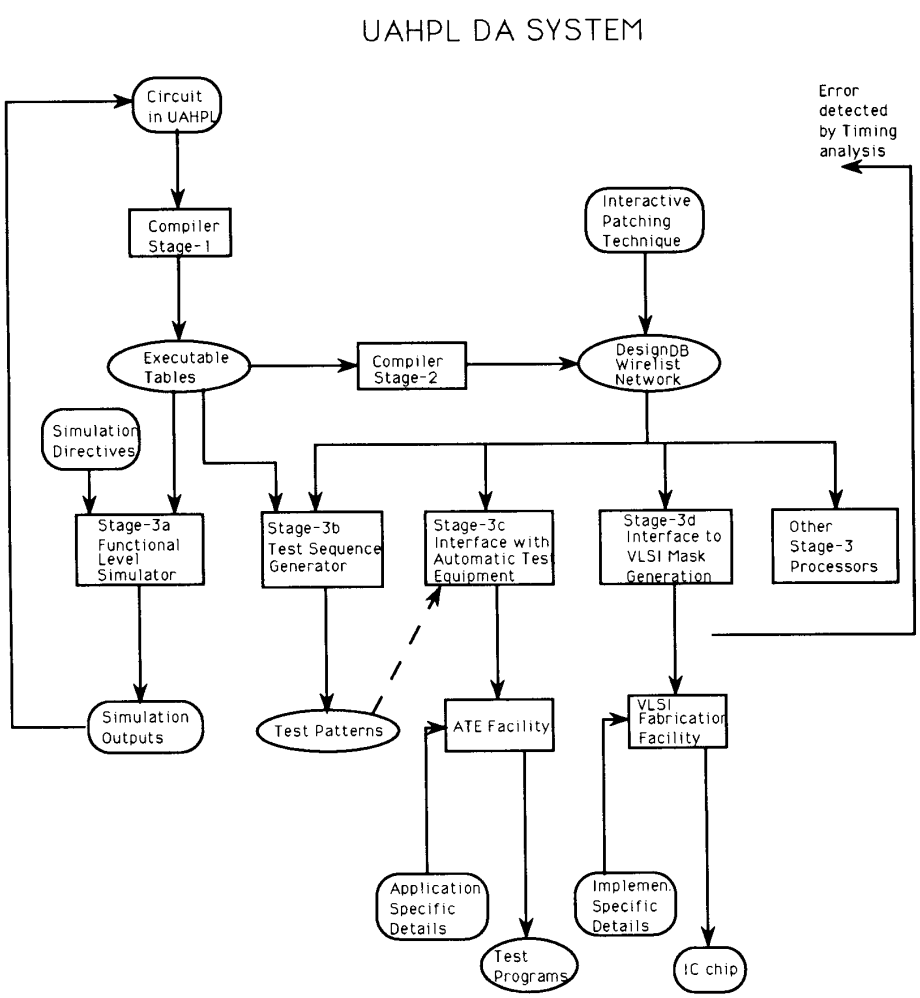


Fig. 3. Block diagram of a UAHPL-DA system.

GATE#	GATE TYPE	ILINK	OLINK	SYMLNK	SIGINP
1	4509	2	157	1	381
2	4509	25	160	24	383
3	4509	47	158	46	355
4	4509	70	161	69	360
5	4018	0	123	91	0
.	....	.	...	..	.
8	4018	0	6	94	0
.	....	.	...	..	.
11	4201	156	163	0	202
.	....	.	...	..	.
16	4506	98	124	97	379
.	....	.	...	..	.
19	4229	122	126	191	238
.	....	.	...	..	.

Fig. 4. Sample of a GATELIST.

PTR#	ELEMT#1	ELEMT#2	NEXT_PTR
1	2	0	0
2	19	8	7
7	16	-1	11
11	-1	0	0
24	2	1	0
157	11	0	0
158	11	0	210
159	2	4	0
160	12	0	0
161	12	0	211
162	11	18	0
...	..	..	..

Fig. 5. Sample of an IOLIST.

logically into control and data units. For a satisfactory level of fault coverage in highly sequential circuits, it is necessary to exercise the control section thoroughly, while the faults in the data section may be searched in a simpler way.

Stage 3C is an interface with commercial test program generation software [11], [12]. It is an important link between the design process and testing, which can be further strengthened by providing a path from Stage 3B as shown by the dotted line in Fig. 3. Stages 3B and 3C allow the designer to determine the testability of a circuit before it is committed to hardware. Testing and test sequence generation, thus, become an integral part of the design process rather than a separate effort by a group of people who were not a part of the design team, and who have to reconstruct the circuit model often from its imprecise description.

Stage 3D is for generating semicustom [13] VLSI layouts.

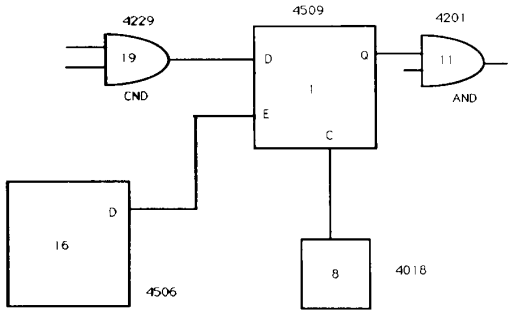


Fig. 6. Partial circuit diagram of GATELIST in Fig. 4.

The components of the Stage 3D processor are elaborated on in the next section.

E. Stage 3D Processor

The task of this stage is to generate the layouts of systems modeled in UAHPL. The various subtasks include preprocessing, automatic placement, and automatic routing. The preprocessor modifies the technology-independent and implementation-independent interconnection list generated by Stage 2 to another format suitable for implementation. In addition, depending on the target technology, it takes care of excessive fanout on pins that may cause performance degradation. The placement software decides on locations of cells on the layout floor corresponding to the various logic elements of the GATELIST. The procedure for placement of cells from the cell library on the layout consists of a heuristic initial placement followed by iterative improvement [14]. The iterative improvement consists of a procedure similar to piecewise interchange with an important difference that the dimensions of cells are taken into account. Information about the dimensions of layout cells is obtained from the cell library. The procedure allows a large cell, such as a macrocell (e.g., ALU or a large flip–flop), to interchange positions with many small cells such as two input gates. This results in reduced interconnection wire length and chip area, which are the objective functions. The procedure may be automatic or may be manually intervened for refinements to optimize timing problems in the final design. Interconnections between cells are made by a grid router which uses a modified Lee's algorithm [15]. Two-layer $H-V$ routing is used. To achieve 100% connectivity, additional channels are automatically added between cells.

The cell library contains layouts of all types of functions required to implement any digital circuit described in UAHPL. It is possible to add large macrocells by first modeling their function in UAHPL. Then, the above-mentioned placement and routing software are used to create new layouts.

The automatically produced fully-routed layout for the UAHPL description of Fig. 1 is given in Fig. 7. The four blocks on the top-most row represent the four output lines $Z[4]$. Below the output lines is the register $F[4]$, whose outputs are connected directly to the output lines. Register $C[4]$ is placed above register $A[4]$ and next to $B[4]$. The four square blocks are the control flip–flops. Ten small blocks

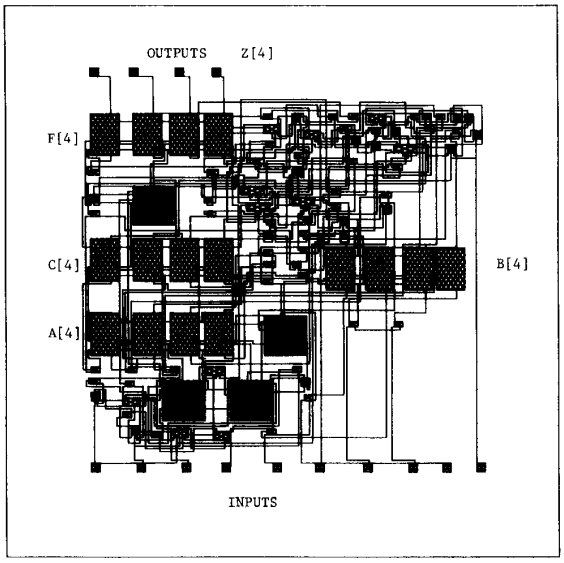


Fig. 7. UAHPL-DA generated layout of SEQSEL.

in the bottom row represent input lines $X[4]$, $Y[2]$, P, Q, CLOCK, and RESET.

We can now summarize some of the salient features of the UAHPL DA system. First, the level of abstraction provided to functionally describe the digital system is adequately high. Second, the various Stage 3 processors use the same tabular and interconnection list representation of the circuit. In other words, the circuit designed does not have to be expressed again and again for the various Stage 3 applications. This saves time and effort as well as eliminating the possibility of discrepancies among various descriptions. Third, the entire DA system is modular, open-ended, and can be easily expanded and modified. More applications can be conveniently added as new Stage 3 processors without making changes in the language or existing software. Application-specific details can be directly provided to the Stage 3 processors. Separating application-specific tradeoffs and parameters from the language and its compiler has enhanced the flexibility of the automation system.

The UAHPL–DA has been tested for several digital systems including multipliers, sequential circuits, the Am2911 sequencer, and a Meggit decoder with encouraging results [13].

III. VLSI Design Tools

The UAHPL–DA system discussed in the previous section enables an automatic layout generation of digital systems modeled at a high level of abstraction. It supports automatic generation of hardware [5], functional-level simulation [8], [9], testing [10]–[12], and automatic layout generation [13].

In the creation of layouts using the UAHPL–DA system (like the one shown in Fig. 7), it was assumed that all layouts in the cell library have been checked for design-rule violations (both physical and electrical) and that there are no timing problems. Also, no mention of input/output

pads for interconnection of the layouts to the outside world was mentioned. Before submitting the layout for fabrication, several verifications have to be made using simulators to ensure correctness of design.

For the above simulations and checks, tools available from a comprehensive toolkit "VLSI Design Tools Release 3.1" [4] have been obtained. In this section, we will discuss tools that are required by the UAHPL–DA system to complete an environment that will support automatic design of VLSI chips and thus support teaching of VLSI DA.

A. Cell Library

The cell library is the heart of the DA system in which layouts in Caltech Intermediate Form (CIF) [16] format are stored. At the time of development of the UAHPL–DA system, the cell library contained only the primitive cells of nMOS layouts. These were handcrafted using a standard layout editor. A set of design-rule-checked cells in CMOS–PW technology is available in the toolkit; for details, see the "Standard Cell Library Guide" in [4]. These include basic gates and flip–flops. Large cells can be made by using the available cells, with the help of a standard layout editor. The approach discussed in Section II is also used for automatic layout and interconnection for the creation of complex cells. Information about the cell dimensions required by the placement software and about the input/output points of cells required by the router are obtained from the cell library. The final layout is also stored back in CIF and can be used for other designs.

Layouts for structured devices created using "generator" programs, and cells from other standard libraries have also been included. The toolkit contains software that can be used to convert between different formats (for example, CIF and Caesar formats). This enables inclusion of designs created under other environments. The use of layout generators is elaborated on in the next section.

The cell library now contains the standard set of basic logic gates such as NOR, NAND, XOR, and complex circuits such as ADDERS, multiplexers, PLA's (programmable logic arrays), ALU's, etc. It also contains input/output pads and circuits required to translate between internal voltages and those required off the chip.

B. VLSI Design Generators

A large portion of routine circuitry can be automatically produced by CAD software. Programs called "generators" can be used to create designs from input parameters. These are suitable for the creation of layouts which have a well-defined structure (e.g., PLA's, multiplexers, registers, etc.). Generator programs available with the toolkit include **buffer** (generates a static buffer), **decoder** (generates a decoder), **mult** (generates a multiplier), **pads** (generates pads and padframe), **plac** (generates a static or dynamic PLA), **ram** (generates a single/dual port register file), **rom** (generates a static/dynamic ROM), **rshift** (generates an n-bit, m-stages shift register), and **tmux** (generates a transmission gate MUX). All layouts are in 3 μ bulk CMOS technology. More details are available in [4, Sect. I-G].

Coordinate Free Lap (CFL) [17] is a library of subroutines written in C language intended to facilitate construction of VLSI circuit layouts. The above generators are CFL-based programs and require a number of leaf cells in Caesar format [18], which comes with the toolkit. Layouts can be produced by these generators and are included in the cell library.

C. Design-Rule Checker

A design rule checker (DRC) is a piece of software that is used to verify that all design rules of a technology have been met. A DRC software called **lyra** [19], available in the toolkit, is used for checking design rules pertaining to overlaps and separation of different rectangles in the layout. **Lyra** takes as input the Caesar description of the design and produces as output an indication of the geometrical design errors that occur with regard to the given set of design rules. It allows both batch hierarchical check and interactive check when used with the Caesar layout editor [18]. **Lyra** currently supports two nMOS rulesets (Berkeley nMOS rules and Mead & Conway rules) and one CMOS ruleset (MOSIS 3 μ bulk CMOS process rules) [16], [20]. One advantage of using the **lyra** checker is that it can be retargeted to a new design ruleset. This is done by first preparing the ruleset in **lyra** format. The rule specification is then compiled by the **lyra** rule compiler to generate an executable file for checking the specified rules. This executable file is invoked by the **lyra** front end to check designs against the ruleset [21].

D. Circuit Extraction

A circuit extractor is a program that takes a layout description and produces as output an underlying electrical circuit consisting of the list of transistors and node connectivity information [22]. Other information such as the dimensions of the devices, resistances, and capacitances of the connecting wires can also be derived.

The circuit extractor available in the toolkit is called **Mextra** (Manhattan circuit extractor for VLSI simulation) [4]. **Mextra** reads the layout file in CIF and creates a circuit description. As an example, for the UAHPL model in Fig. 2 (a chain of four pass transistors), Fig. 8 shows its layout and the corresponding CIF file. The extracted transistor circuit description is given in Fig. 9. This extracted description provided by **Mextra** can be translated using translation software available with the toolkit to convert **Mextra** outputs to acceptable **Spice** (a simulation program with integrated circuit emphasis) [23] and **Rnl** (NetLisp, timing and logic simulator for VLSI) [24] inputs.

E. Circuit Simulation

The output of circuit extractors is used by simulators to verify functional correctness and for transient analysis of MOS circuits. Depending on the type of simulation task, the extraction of parasitic capacitance can be made optional.

Spice [23] is a general-purpose circuit simulation program. Circuits may contain resistors, capacitors, inductors, transistors, etc. **Rnl** [24] is a timing logic simulator for digital MOS circuits with a lisp-like command interpreter. It is even-

```
DS 111 250 1;
9 PASS4;
L NP ;
B  2 6  5, 5;
B  2 6  9, 5;
B  2 6 13, 5;
B  2 6 17, 5;
L ND;
B 18 2 11, 5;

94 A(1)  3 5;
94 A(2)  7 5;
94 A(3) 11 5;
94 A(4) 15 5;
94 A(5) 19 5;
DF;
C 111;
End
```

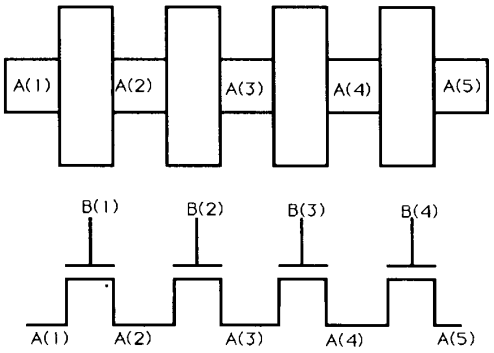


Fig. 8. Chain of four pass transistors and their CIF file.

```
units: 1      tech: nmos format: UCB
e 4 A(5) A(4) 500 500 4000 1000
e 3 A(4) A(3) 500 500 3000 1000
e 2 A(3) A(2) 500 500 2000 1000
e 1 A(2) A(1) 500 500 1000 1000
L 1 0 0 0 0 750000 4000 0 0 0 0
L 2 0 0 0 0 750000 4000 0 0 0 0
L 3 0 0 0 0 750000 4000 0 0 0 0
L 4 0 0 0 0 750000 4000 0 0 0 0
L A(1) 0 0 0 0 0 0 249875 1997.5 0 0
L A(2) 0 0 0 0 0 0 249750 1997.5 0 0
L A(3) 0 0 0 0 0 0 249750 1997.5 0 0
L A(4) 0 0 0 0 0 0 249750 1997.5 0 0
L A(5) 0 0 0 0 0 0 249875 1997.5 0 0
```

Fig. 9. Mextra output for CIF input of Fig. 8.

driven and uses a simple RC model of the circuit to estimate node transition times and the effects of charge sharing. **Spice** and **Rnl** input files can be obtained from the layout by using **Mextra** and format translators **pspice** and **presim**, respectively. Fig. 10 is the **Spice** deck created by **pspice** (a preprocessor for Spice) from **Mextra** output.

F. Pad Frame and Pads

In order to communicate with the outside world, the EXINPUTS and EXBUSES declared in the UAHPL model are to be connected to input/output pads. To facilitate this, the toolkit consists of software called **pads**. This is a CFL-based module generator program for the MOSIS 3 μ bulk CMOS padframe layout. Leaf cells derived from MIT pads received from MOSIS are used to produce a padframe in Caesar format. Specifications are read from standard input/output and consist of a frame specifier and pad specifiers. Pad specifiers (consisting of pad number and pad type) are used to determine the location and type of circuitry to place on specific pad sites. More details of pads, that is their dimensions, types, usage, etc., are given in [4].

The VLSI toolkit is very comprehensive, and only a small subset of the tools was discussed previously. The integrated VLSI DA system is illustrated in Fig. 11. **Simscope**, shown in Fig. 11, is a program used to view the timing diagrams produced by outputs of **Spice** and **Rnl**. **Vic**, shown in Fig. 11, is a program used to view layouts in Caesar format. The toolkit also contains several supporting files such as **config-files** and **technology-files**, which contain various parameters obtained from device modeling. The above-discussed VLSI design tools are available on the first of the two files that come on one tape. The second file and its documentation, which are not discussed in this report, also contain essential software such a **Crystal** [25] for timing analysis and **Magic Tools** [26]. The inclusion of tools from this second file will considerably enhance the power of the DA system.

IV. Discussion

The two preceding sections described two important software packages that can be used for the establishment of an integrated system to teach VLSI DA. With the recent lifting of the ban on the export of VLSI design software from the U.S., it is now possible for universities and departments all over the world to obtain the above-described software at no cost. In this section, we briefly present the hardware and software requirements for establishment of the VLSI DA system depicted in Fig. 11, and outline its use in teaching related undergraduate and graduate courses. Some of the projects that were accomplished using the DA system are also listed.

A. Hardware and Software Requirements

Most of the work related to UAHPL was done at the University of Arizona and King Fahd University of Petroleum and Minerals. The package has been successfully installed on both an IBM 3033 mainframe and a VAX 11/780 that runs a VMS operating system. The source code is in standard Fortran and with little effort can be ported to run on other machines. Some UAHPL tools like the Stage 1 compiler and functional simulator are also available for IBM PC's.

```
*** SPICE DECK created from x.sim, tech=nmos
M1    5   4 6   7 NMOSE L=5.0U W=5.0U
M2    9   8 5   7 NMOSE L=5.0U W=5.0U
M3   11  10 9   7 NMOSE L=5.0U W=5.0U
M4   13  12 11  7 NMOSE L=5.0U W=5.0U
C5   12   0 0.003000PF
C6   10   0 0.003000PF
C7    8   0 0.003000PF
C8    4   0 0.003000PF
C9   13   0 0.004496PF
C10  11   0 0.004495PF
C11   9   0 0.004495PF
C12   5   0 0.004495PF
C13   6   0 0.004496PF
.MODEL NMOSE NMOS LEVEL=2.00000 LD=0.211698U TOX=635.000E-10
+NSUB=3.779887E+15 VTO=1.13877 KP=4.145038E-05 GAMMA=0.494661
+PHI=0.600000 UO=300.000 UEXP=1.001000E-03 UCRIT=441843.
+DELTA=1.26380 VMAX=100000. XJ=5.27683U LAMBDA=2.385822E-02
+NFS=2.356687E+12 NEFF=1.001000E-02 NSS=0.000000E+00 TPG=1.0000
+RSH=25.4 CGSO=1.6E-10 CGDO=1.6E-10 CJ=1.1E-4 MJ=0.5 CJSW=5E-10
+MJSW=0.33
VE 7    0 DC 0
.END
```

Fig. 10. Spice deck created by pspice from Fig. 9.

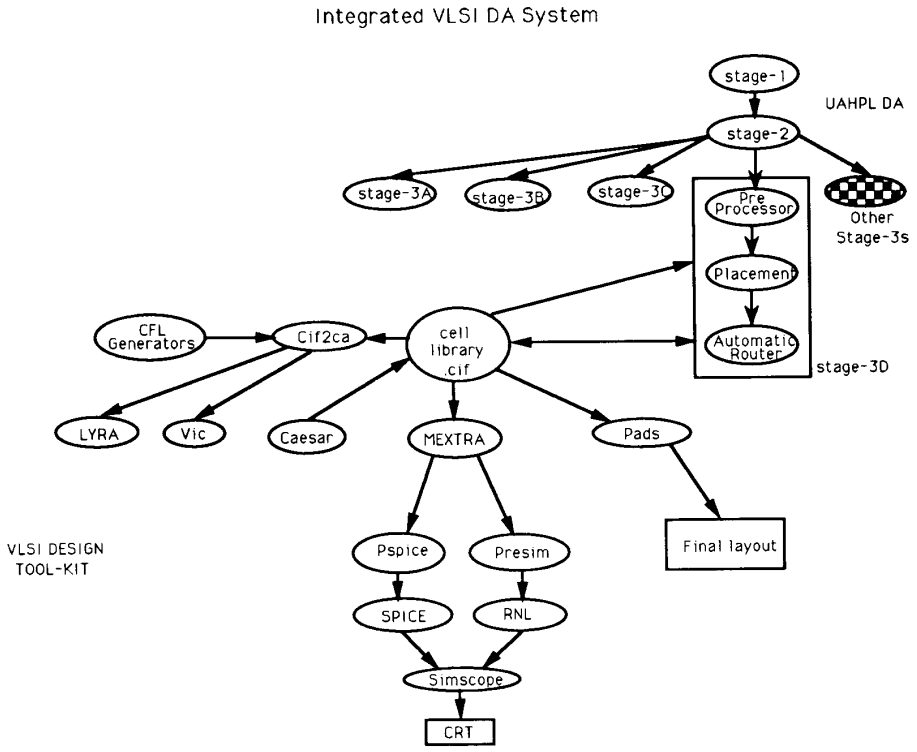


Fig. 11. Integrated UAHPL-based VLSI DA system.

"VLSI Design Tools, Release 3.1" runs on VAX 11/780 with Berkeley's 4.2 or 4.3 UNIX and on Sun3 under Sun OS 3.2. Some of the tools require plotting and graphic devices. Both **Simscope** and **Vic** are designed to display outputs on Tektronix 4105 or GP19 graphics terminals. **Simscope** is used to display timing diagrams and **Vic** for displaying layouts. IBM PC's connected to VAX through Kermit can also be used to display using **Simscope** and **Vic**. Depending on the hardware facilities on site, drivers and emulators can be developed in-house. If hardware facilities like DEC workstations, etc., are

available, then emulators can be commercially obtained.

Universities that have more than one VAX 11/780 interconnected, and run both the operating systems (i.e., VAX VMS and Berkeley UNIX) can have a distributed DA system. The UAHPL–DA can run on one VAX (VMS) and the VLSI Design Tools on the other (UNIX). This is the current configuration at our school. However, the UAHPL tools written in standard Fortran can be ported with little effort to run under UNIX.

B. VLSI DA Courses

The toolkit offers a large number of options on how a DA course can be conducted. The approach taken will greatly depend on the experience and dedication of the instructor, the support and competence of the teaching assistants, and the composition and level of the students and their background in electronics and VLSI.

At the undergraduate level, it is recommended that the course be conducted with a laboratory session that meets once a week. The theory sessions could be dedicated to the discussion of the UAHPL-based DA, other VLSI design methodologies, and VLSI DA tools with less emphasis on algorithms for DA. The laboratory sessions may be dedicated to experimentation with various packages to teach circuit simulation, design-rule checking, and timing analysis, since this is the best way to teach undergraduate students about real-world engineering. Other experiments may include the creation of tiles for structured layouts, the building of **lyra** rulesets for a given set of design rules, and the design of simple generator programs using CFL. The course could also include a project that uses most of the tools discussed. One such undergraduate course project was the design and implementation of a package for automatic mapping of CLU's in UAHPL to layouts using Weinberger arrays [27], [28]. Inclusion of projects induces creativity and gives a better overall view of the VLSI DA system.

A graduate-level course that uses the VLSI DA system could concentrate on the design of underlying DA algorithms and their implementation. Topics of discussion could include algorithms for automatic hardware synthesis, partitioning netlists, automatic placement and routing, circuit extraction, design-rule checking, circuit simulation, and timing analysis. Course projects may be related to design and implementation of new algorithms for DA, the performance of which can be compared with those of the corresponding tools in the toolkit. Research and graduate class projects are presented in the next section.

C. UAHPL–DA Projects

Most of the VLSI DA projects that have been implemented thus far have been related to the automatic mapping of UAHPL circuit descriptions to various structured layout targets. The final implementations have been in PPLA's (path programmable logic arrays) [29], the SLA (storage logic array) [30], and the Weinberger arrays [27].

Mapping netlists to new layout structures, implementation using ULM's (Universal logic modules) [31], investigating transformations of hardware generated by the UAHPL compiler to another form suitable for efficient implementation in VLSI, delay modeling of layout cells in the cell library, and investigation of new device technologies for implementation are some research projects worth pursuing.

Several digital DA projects that are not related to VLSI have also been completed. One of them was the automatic mapping of UAHPL circuits to a printed circuit board (PCB) layout and implementation using the TTL 7400 series [32]. A database to help manage the VLSI design process was also attempted [33]. Another successful project was the implementation of a tool for automatic microcode generation for controllers of systems modeled in UAHPL [34].

There is no limit to the number of research projects that can be investigated, and it is obvious that the presence of the above DA system will definitely have a great impact on the students of DA courses.

UAHPL tools can be obtained by writing to Prof. F. J. Hill at the University of Arizona or to Prof. M. Masud at the University of Petroleum and Minerals. "VLSI Design Tools" can be obtained by writing to V. Palm at the University of Washington or to The Industrial Liason Program at the University of California, Berkeley.

V. Conclusions

This paper presents a powerful language-based integrated environment for teaching DA of VLSI. Various aspects of design, from modeling using a high-level language to final implementation in VLSI, can best be explained with the help of tools available in the UAHPL-based VLSI DA system. The DA system can be used for laboratory experiments related to modeling, simulation, and verification of different phases of the digital design process. There is no restriction on the structure of the final product. The presence of excellent tools in the UAHPL-based VLSI DA system allows students to build real designs and verify their correctness in a relatively short time. Additional tools may be procured or developed and attached to the modular DA system without making any changes in the language or the existing software.

Acknowledgment

Acknowledgments are due to the King Fahd University of Petroleum and Minerals, Dhahran, and the King Abdul Aziz City for Science and Technology (under Project AR 11–21) for their support. The author also wishes to acknowledge the help of Dr. M. S. T. Benten and A. M. T. Khan for their help in the installation of software, testing of packages, and preparation of this manuscript.

References

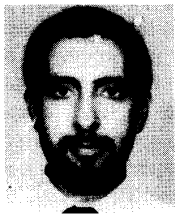
[1] M. T. O'Keefe, J. C. Lindenlaub, S. C. Bass, and T. P. Waheln, "Introducing VLSI computer-aided design into the EE curriculum: A case study," *IEEE Trans. Educ.*, vol. 32, no. 3, Aug. 1989.

[2] K. F. Smith and J. Gu, "Fast structured design of VLSI circuits," *IEEE Trans. Educ.*, vol 32, no. 3, Aug. 1989.

[3] M. Masud and S. M. Sait, "Universal AHPL—A language for VLSI design automation," *IEEE Circ. Dev. Mag.*, Sept. 1986.

[4] *VLSI Design Tools Reference Manual, Release 3.1,* NW Lab. for Integ. Syst., Univ. of Washington, Feb. 1987.

[5] M. Masud, "Modular implementation of a digital hardware design automation system," Ph.D dissertation, Dept. Electric. Eng., Univ. of Arizona, 1981.
[6] F. J. Hill and G. R. Peterson, *Digital Systems: Hardware Organization and Design.* New York: Wiley, 1973.
[7] K. Iverson, *A Programming Language.* New York: Wiley, 1962.
[8] M. M. Alsharif, "Functional level simulator for universal AHPL," M. S. Thesis, Univ. of Arizona, 1983.
[9] Z. Navabi, "Hardware program simulator," M. S. Thesis, Univ. of Arizona, 1978.
[10] C. H. Chiang, F. Hill, A. Mohseni, and D. Chen, "Fault detection test generation at the register transfer level," in *Proc. IEEE First Ann. Phoenix Conf. on Comp. and Commun.,* May 1982, pp. 58–63.
[11] K. Wacks, F. Hill, M. Masud, and P deBruyn Kops, "An integrated system for LSI device modeling," in *Proc. IEEE Test Conf.,* Philadelphia, PA, Nov. 1980, pp. 473–478.
[12] K. Wacks, P deBruyn Kops, F. Hill, and M. Masud, "Model LSI devices from manufacturer's data," *Electron. Design,* pp. 103–108, Nov. 1980.
[13] S. M. Sait, "VLSI mask generation from register transfer level descriptions: An automated approach," Ph.D. dissertation, King Fahd Univ. of Petroleum and Minerals, Nov. 1986.
[14] S. M. Sait, "A general cell placement procedure for UAHPL based DA system," in *Proc. CompEuro '87,* Hamburg, Germany, May 1987, pp. 513–514.
[15] C. Y. Lee, "An algorithm for path connections and its applications," *IRE Trans. Electron. Comput.,* vol. EC-10, no. 3, pp. 346–365, Sept. 1961.
[16] C. Mead and L. Conway, *Introduction to VLSI Systems.* New York: Addison Wesley, 1980.
[17] W. Becket, *Coordinate Free LAP Reference Manual, Version 1.2, in VLSI Design Tools Reference Manual, Release 3.1,* NW Lab. for Integ. Syst., Univ. of Washington, Feb. 15, 1987.
[18] J. Ousterhout, *Editing VLSI Circuits with Caesar.* Computer Science Div., Univ. of California.
[19] M. H. Arnold, "Corner-based geometric layout rule checking for VLSI circuits," Univ. of Calif. at Berkeley Rep. UCB/CSD 36/264, Ph.D Thesis, 1985.
[20] *MOSIS Scaleable and Generic CMOS Design Rules, Revision 6.* USC Inform. Science Inst., U.S.C., Feb. 1988.
[21] M. H. Arnold, "Specifying design rules for lyra," In *VLSI Design Tools Reference Manual, Release 3.1.*
[22] D. T. Fitzpatrik, "Exploring structure in the analysis of integrated artwork," Univ. of Calif. at Berkeley Rep. UCB/CSD 38/130, Ph.D. Thesis, 1983.
[23] L. W. Nagel, "SPICE 2: A computer program to simulate semiconductor circuits," Memo ERL-M520, Energy Res. Lab., Univ. of Calif., Berkeley, May 1975.
[24] C. Terman, *Users Guide to NET, PRESIM and RNL/NL.* Cambridge, MA: Laboratory for Computer Science, M.I.T.
[25] J. K. Ousterhout, "Using crystal for timing analysis," in *1986 VLSI Tools: Still More Works by the Original Artists,* Rep. UCB/CSD 86/272, Univ. of Calif. at Berkeley, 1985.
[26] J. K. Ousterhout, G. T. Hamachi, R. N. Mayo, W. S. Scott, and G. S. Taylor, "A collection of papers on magic," Rep. UCB/CSD 83/154, Univ. of Calif. at Berkeley, Dec. 1983.
[27] S. M. Sait and F. A. Al-Khulaiwi, "Automatic Weinberger synthesis from a UAHPL description," *Int. J. Electron.,* vol. 69, no. 2, pp. 211–224, 1990.
[28] A. Weinberger, "Large scale integration of MOS complex logic: A layout method," *IEEE J. Solid State Circ.,* vol. SC-2, no. 4, pp. 182–190.
[29] T. A. Olson, "Automatic AHPL description to path programmable logic array," M.S. Thesis, Univ. of Arizona, 1984.
[30] F. Hill, Z. Navabi, C. Chiang, D. Chen, and M. Masud, "Hardware compilation from an RTL to a storage logic array target," *IEEE Trans. CAD/ICAS,* vol. 3, no. 3 pp. 208–217, July 1984.
[31] P. I. Jennings, S. Hurst, and A. McDonald, "A highly routable ULM gate array and its automated customization," *IEEE Trans. Comp. Aided Design,* vol. 3, no. 1, Jan. 1984.
[32] M. Masud, J. Yazdani, and S. M. Sait, "PCB layouts from RTL descriptions: An automated approach," in *Proc. 11th Nation. Comput. Conf. and Exhibit.,* King Fahd Univ. of Petroleum and Minerals, Dhahran, Saudi Arabia, Mar. 1989.
[33] S. Khan, "Design and implementation of a database for managing chip design," M.S. Thesis, King Fahd Univ. of Petroleum and Minerals, 1987.
[34] S. M. Sait, A. Y. Yaagoub, and M. Masud, "A CAD tool for the automatic generation of microprograms for systems modeled in UAHPL," *J. Microprocess. and Microsyst.,* Oct. 1988.



Sadiq M. Sait received the B.S. degree in electronics from Bangalore University in 1981, and the M.S. and Ph.D. degrees in electrical engineering from King Fahd University of Petroleum and Minerals (KFUPM), Dhahran, Saudi Arabia, in 1983 and 1987, respectively

He was awarded a Research/Teaching Assistantship by the Department of Electrical Engineering during 1981–1986. Since 1987, he has worked at the Department of Computer Engineering as an Assistant Professor, teaching graduate and undergraduate courses in the areas related to Digital Design Automation, VLSI System Design, and Computer Architecture. In 1981, he received the Best Electronic Engineer award from the Institute of Electrical Engineers, Bangalore, India. In 1990, he was awarded the Distinguished Researcher Award by KFUPM. In 1988, 1989, and 1990, he was nominated by the Department of Computer Engineering for the Best Teacher Award. He has authored over 30 research papers in international journals and conferences. He has also contributed two chapters to a book entitled *Progress in VLSI Design.* He served on the editorial board of the *International Journal of Computer Aided Design* between 1988–1990, and was invited to serve as a Guest Editor for their Special Issue on Hardware Description Languages. He has also served as a referee for several leading journals. He has given lectures at The University of Erlangen in Nurenberg, and at the IBM Research Center in Rushlikon. His current areas of interest are in digital design automation, VLSI system design, and high-level synthesis.