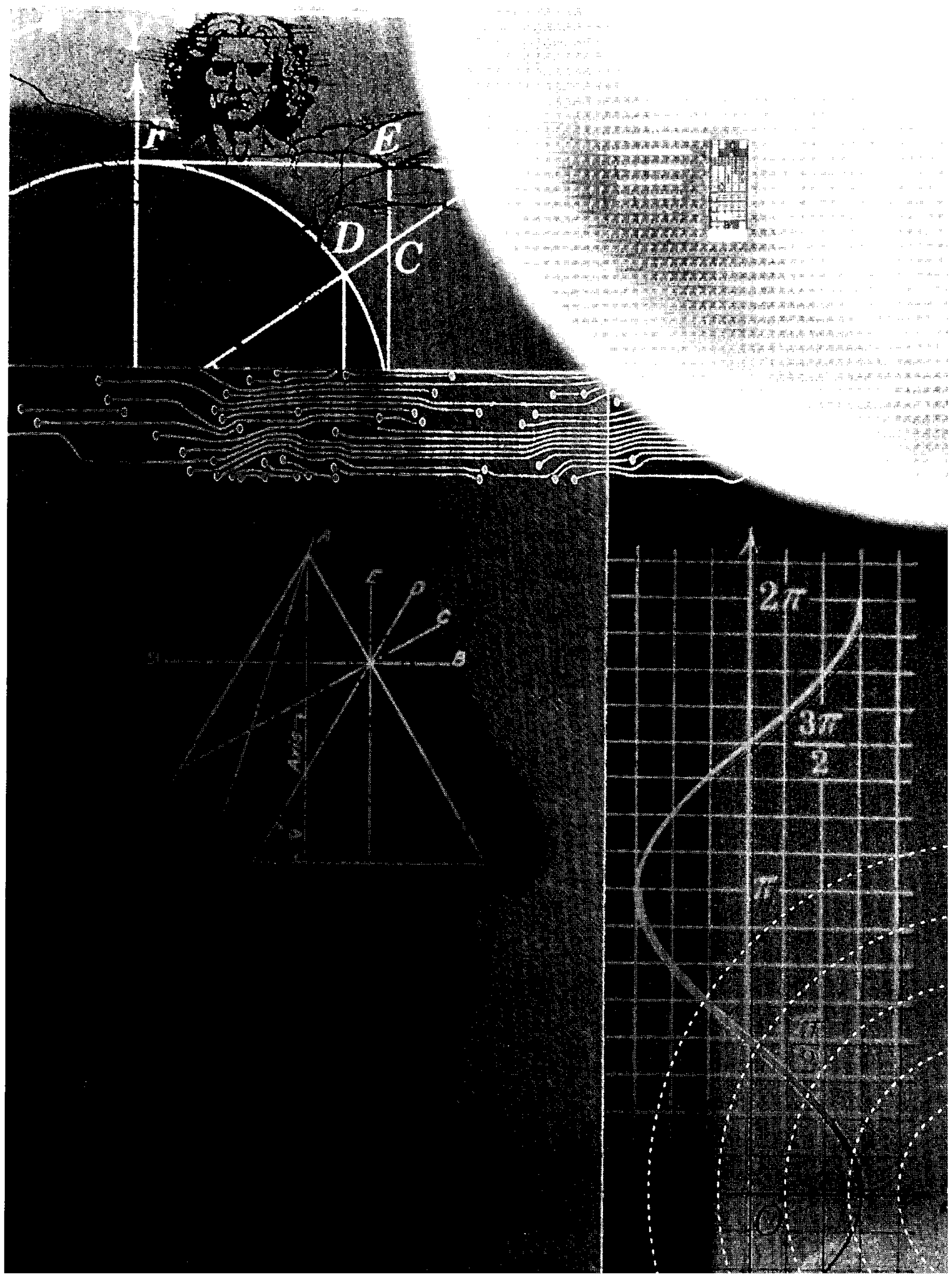
SADIQ M. SAIT, BENTEN MS, KHAN AMT
DESIGNING ASICS WITH UAHPL
IEEE CIRCUITS AND DEVICES MAGAZINE 11 (2): 14-24 MAR 1995

Designing ASICs with UAHPL



The Image Bank · Steven Hunt

Prototype system for standard cell based ICs will ultimately reduce 'time to market'

Reducing the 'time to market' is one of the greatest challenges facing the electronics industry today. This includes the time to design, manufacture, and test new products. Reducing design time, in particular, is actually more important than optimizing the area or performance of an IC. Saving time is especially important in the case of Application Specific Integrated Circuits (ASICs) since, unlike a microprocessor, they are not programmable, and have a limited market.

Reducing design time requires a high degree of automation. Automation can be achieved by standardizing the design process. Such a design automation environment for chips, developed at The King Fahd University of Petroleum and Minerals (KFUPM), helps to reduce design time and effort at both the architectural and layout levels.

The KFUPM design automation system, called Universal AHPL (A Hardware Programming Language), is a blend of tools developed locally and those developed at the University of Arizona, University of California at Berkeley, University of Washington, and the Microelectronics Center of North Carolina (MCNC). The UAHPL system is suitable for design of synchronous digital systems in VLSI. It has been successfully used to synthesize designs such as data compression chips, protocol processors, programmable CRC checkers, digital controllers, computer arithmetic algorithms, and small microprocessors. In addition to providing workable designs with short turnaround time, the system is an excellent educational tool to convey concepts related to digital system design, synthesis, and VLSI design automation.

Literature Review

Logic synthesis has become the norm for today's ASIC design. These have evolved from schematic capture based systems, in which the designer expresses the design graphically, to hardware description language (HDL) systems, which take a HDL representation of the system and produce a netlist of logic gates and flip-flops. These systems are coupled with physical design systems that produce the layout of mask geometries required for fabrication. The logic synthesis systems are analogous to software compilers.

A number of such systems have been proposed. Some of them are targeted towards specific application domains such as digital signal processing (DSP). In the domain of data flow applications, DSP has been at the forefront of high-level synthesis systems. One of the most industrially successful HLS system has been CATHEDRAL [1]. Other systems have also been built [2]. These mostly used applicative programming languages such as SILAGE, SDL, and so on to specify the design.

The control flow oriented systems are fewer, and not as successful. Most of these have adopted VHDL as the specification language due to its success in hardware description domain. The main developments in this area have come from large companies. IBM, for example, presented its high-level IBM synthesis system (HIS) [3], and a Siemens research team developed CALLAS [4]. Contributions from the universities include AMICAL [5] and ALLIANCE [6]. Each of these systems have their own subset of VHDL as the synthesis specification, resulting in different design spaces. They handle the issues of allocation and scheduling to synthesize a circuit from abstract VHDL descriptions.

Among the commercially available systems, a number of them are based on VHDL/Verilog as the HDL. These include Synergy from Cadence Design Systems Inc., Epoch/Finesse from Cascade Design Automation, and ASIC Synthesizer from Compass Design Automation Inc. [7]. Many of these systems provide the capability to program PLDs apart from synthesizing ASICs.

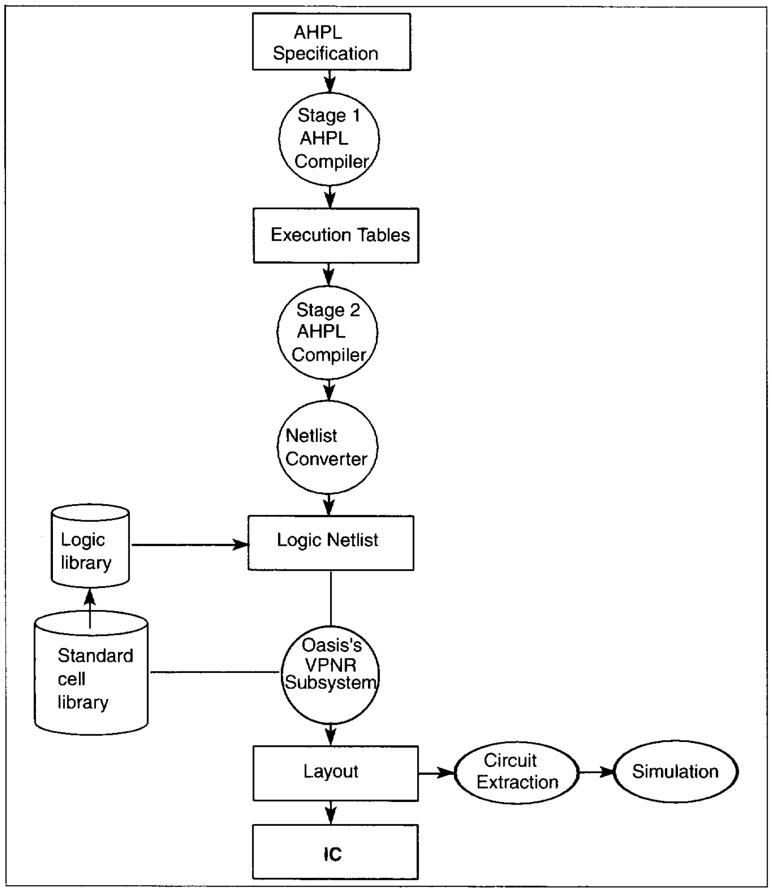
by Sadiq M. Sait, Muhammad S.T. Benten, and Asjad M.T. Khan

UAHPL System

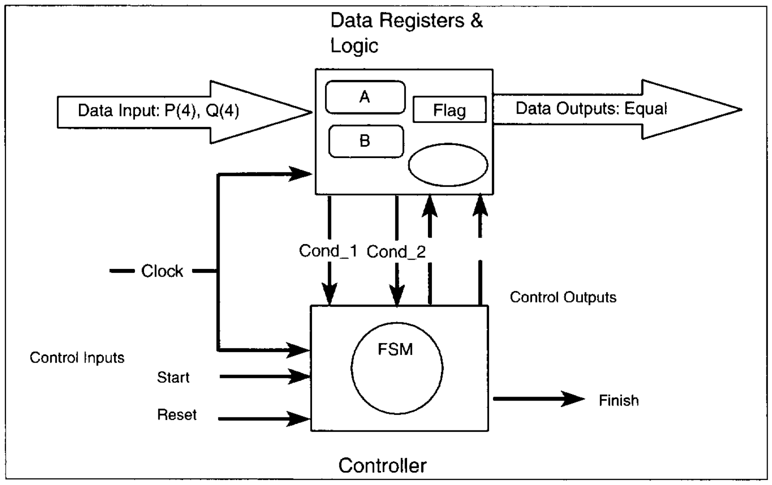
Register transfer or RT-level HDLs have an established formal foundation and have been used widely in the industrial world. The UAHPL system belongs to this category of systems. It takes input at the register transfer level. This specification undergoes *logic synthesis* to give a gate level specification in terms of a netlist of standard gates and flip-flops. Simulation is done both at the register transfer and the gate level and results are compared to verify the translation process. This finishes the synthesis task. The netlist is given to a physical design subsystem which has placement, routing, and graphic layout manipulation tools along with a standard library. The layout produced is checked for design rules and the circuit is extracted. The extracted circuit is again simulated and the results compared with the results of the functional simulation at the register transfer level. A detailed schematic diagram of the UAHPL system is shown in Fig. 1.

Overview

The conventional design approach (which uses finite state machine model of synchronous digital systems) is inadequate for designing large digital systems. These systems typically have a number of registers, many bits wide, for storing data operands. Including these in the model would make the number of states large and render the conventional techniques computationally impossible. Extended state tables provide an extension to the FSM model. The system is partitioned into two sequential circuits, one for the data path and the other for the controller. The data path consists of registers and interconnected combinational logic. This part sends the status of computation as input to the controller for branching information. Other signal information is introduced at the controller's external inputs, and the controller produces control signals that are passed along the data path to control



1. Block diagram of current UAHPL based design automation system.



2. Block diagram of the Equality_Detector circuit.

register transfers. This extended-FSM model operates at the register transfer level, in contrast to the classical FSM model, which deals with each flip-flop and all possible states.

The extended state table specifies the two partitions. It specifies the controller using the FSM model, and the register transfers in the data path are specified at each state. UAHPL is basically an extension to the extended state machine model. The basic ideas will be developed by means of an illustrative example.

Consider the example of a serial comparator. Testing for equality of two n-bit vectors serially is possible by testing for equality of the LSBs and shifting the vector right. At most, n shifts are required until a decision can be made. A possible algorithmic description of the digital system explained above is:

```
Step 1: Load A and B.
Step 2: If both A and B are zero
then FLAG = TRUE, goto Step 5.
Step 3: If LSB(A) ≠ LSB(B)
then FLAG = FALSE, goto step 5.
Step 4: Shift right A and B, goto Step 2.
Step 5: OUTPUT = FLAG.
```

The above algorithm can be mapped to hardware using the extended state table model. In contrast to the conventional design procedure, where one has to come up with a state diagram from the functional description, the design process here is limited to algorithm development. The process of mapping an algorithm to an extended state table involves the identification of the hardware components required to implement the data path. Assumptions are required about the word length of operands **A** and **B**. To store these operands, two registers, say of 4-bits each, are required. Further, a flip-flop is needed to store the result of the computation. Since the time of completion of comparison is data dependent, two outputs are used, namely FINISH and EQUAL. Line FINISH is made high to indicate the completion of comparison, and the value on line EQUAL (when FINISH is raised) will specify if the two vectors are equal or not. The extended state table model for the equality detector is shown in Fig. 2.

With the above model, the extended state table for the equality detector can be specified as in Table 1. In the extended state model, the first column specifies the state of the controller, the second multicolumn

Table 1. Extended state table of serial Equality_Detector

State	Inputs (Start, Cond_1, Cond_2								Transfer
	000	001	011	010	110	111	101	100	
C_1	C_1	C_1	C_1	C_1	C_2	C_2	C_2	C_2	*A = P; B = Q*
C_2	C_3	C_3	C_5	C_5	C_5	C_5	C_3	C_3	*Flag = 1*
C_3	C_4	C_5	C_5	C_4	C_4	C_5	C_5	C_4	*Flag = 0*
C_4	C_2	C_2	C_2	C_2	C_2	C_2	C_2	C_2	*ShiftRight(A); ShiftRight(B)*
C_5	C_5	C_5	C_5	C_5	C_5	C_5	C_5	C_5	*Equal = Flag; Finish = 1*

specifies the next state for all the possible combinations and the third column specifies the register transfers that are to occur for that particular state. The controller uses a Moore type FSM model. An output associated with each state controls the transfers in the data path.

This operation is represented in UAHPL succinctly as shown in Fig 3. Each statement in the UAHPL code corresponds to a row of the extended state table and specifies the state. One statement in UAHPL consists of two parts, one identified by:

=(condition)/(next_state#)

This means 'if condition is true, then branch to next_state number,' and gives the next state information. The second part specifies the transfers associated with that state. These are denoted by the following syntax:

destination $\leq$ src

Note that the next state information does not contain all possible input combination and their next states; rather, only the condition that is significant is listed. It is possible to list all conditions, however. As is usual in most practical circuits, there are situations where not all inputs are significant in a particular state (incompletely specified machines), and UAHPL provides the flexibility of specifying only that condition that is required for a decision. Furthermore, although this example gives the next state information in all the states, it is not necessary to do so. The execution should be sequential, unless a branching information is given. The controller goes to next state sequentially in the next clock pulse if no next states are specified. This information is built into the compiler [8].

The UAHPL language provides a programming language based method for specifying the design. The notations used are based on the language APL. However, unlike the current standard in HDLs, that is, VHDL, UAHPL has been designed to enable synthesis. We will present the basic ideas about the synthesis procedure by illustrating how hardware can be derived for the example of an equality detector.

UAHPL based synthesis assumes that the system uses a single clock and that all the memory elements consist of negative-edge-triggered D flip-flops. These assumptions have effect on modeling as well as synthesis. The controller extraction from the UAHPL sequence is achieved using a Moore type FSM model. Each statement of the UAHPL corresponds to a state.

A straightforward method of constructing the controller is to use one-hot encoding (one flip-flop per state). Hence, each flip-flop represents a control state. The output from these flip-flops can be used to control the transfers in the data path. The control part can be extracted from the UAHPL description by converting the branchinginformation (=(condition/(next_state #)) in each statement to inputs of the D-type flip-flops.

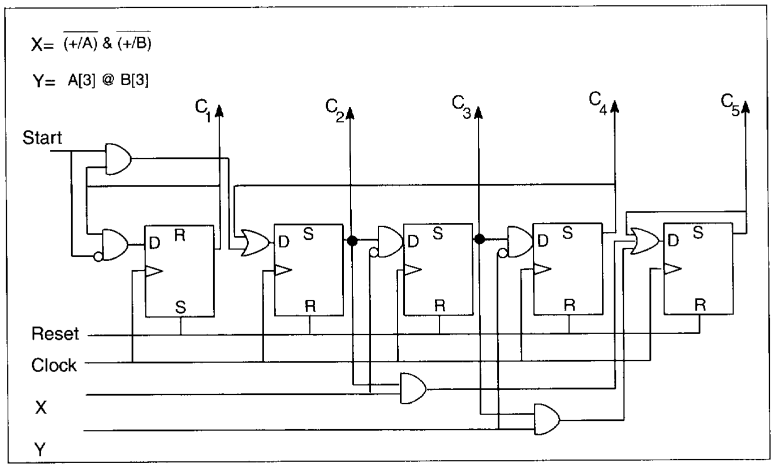
```
MODULE: EQUALITY_DETECTOR.
MEMORY: FLAG;A[4];B[4].
EXINPUTS; START; RESET; CK1; P[4];Q[4].
OUTPUTS; EQUAL; FINISH.

BODY SEQUENCE; CK1.
1 A<=P; B+Q; => ^START/(5).
2 FLAG<=1$1;  => (^(+/A)& ^ (+/B))/(5).
3 FLAG<=1$0;  => (B[3]@A[3])/(5).
4 B<=1$0,B[0:2]; => A<=1$0,A[0:2]; =>(2).
5 EQUAL=FLAG; FINISH=1$1;=>(5).

ENDSEQUENCE
CONTROLRESET (RESET)/1.
END.
```

3. Complete UAHPL model of a serial Equality-Detector.

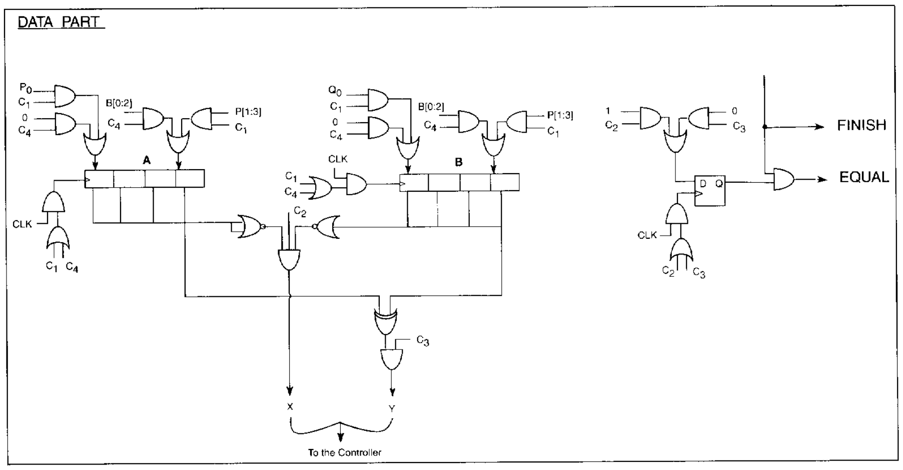
Each D flip-flop is fed by a number of inputs OR'ed together. Each input represents a state AND'ed with the condition that would cause the transition to this state (flip-flop). This method would not lead to a minimum realization. However, it is simple and can be realized from the UAHPL program using simple rules. The complete controller realization for the equality detector is shown in Figure 4.

Realization of the data path from the UAHPL specification uses the control signals for each control state. All the registers and flip-flops are not clocked by a single clock. The con trolled clock approach is used. That is, the data path is fed by a single external clock, the same as the one feeding the controller. However, the registers are enabled by clocks gated with the output signals from control flip-flops called CSLs (control signal levels).

The data path realization is achieved by con trolling the clock and inputs of the registers. A register can receive its input from many sources; therefore, each of these are AND'ed with the appropriate control signal for which they should occur, and then OR'ed and fed as input. The register is clocked only when a transfer to the register is required. This is done by OR'ing all the control states for which a transfer should occur, and using it to gate the clock. The complete data path for the equality detector is shown in Fig. 5.



4. Controller of the serial Equality_Detector circuit.



5. Data path of the serial Equality_Detector circuit.

Again, this technique would not necessarily lead to optimal/minimal hardware. However, it can be derived from the UAHPL specification by means of simple rules, which are:

(1) Find the hardware used (this is usually declared), look for each register the control states in which it is modified, OR those states and generate the gated clock for the register. (2) Look for the sources to the register, AND each source with the appropriate control state signal, then OR all the sources, and feed the output of the OR gate to the register input. In this language, data assignment to the registers is achieved synchronously by an implicit clock.

In UAHPL, digital designs are described using interactive concurrent modules. Iterative combinational networks such as adders, decoders, and so on can be described as Combinational Logic Units (CLUs). The language does not support timing mechanisms. Assigning values to buses have immediate effect, while values to registers become effective at the trailing edge of the clock.

In this section, we introduced UAHPL at a rudimentary level. There are a number of features like conditional transfers, handling of memory, combinational circuits, and so on, some of which have been investigated [8,9].

Example: Programmable CRC Generator
Another example of an UAHPL model of a programmable CRC generator is given in Fig. 6. Referring to the UAHPL model, the initialization is done in the 1st state. In the 2nd state, a 64-bit message is supplied sequentially on line MESIN and a 16-bit CRC pattern is simultaneously generated and stored in a register, CREG. When all the message bits have been processed, the CRC pattern can be serially appended to the message data stream for transmission.

The CRC pattern is serially appended to the message in the 3rd state. The generator then goes to the initial state to perform the same task described for another message, if any. The 16-bit CRC pattern is available on the 66th clock pulse. The transmission of the 16th bit takes place on the 81st clock pulse. Thus, the number of clock cycles required to generate and transmit any CRC in this implementation, is the sum of the size of the message and the degree of the generator polynomial.

As seen in Figure 6, every UAHPL description consists of basically three parts: a declaration, a procedural section describing the state machine, and a non-procedural section. The numbered steps between the keywords SEQUENCE and ENDSEQUENCE form the procedural section defining the states of the sequential machine. In this section, a statement is active only when the machine is in the corresponding step (state). The non-procedural section follows the keyword ENDSEQUENCE. Statements in this part are always active regardless of the state of the control sequencer.

UAHPL is based on the fact that any digital system can be partitioned into a data part and a control part. The data part consists of registers, buses, CLUs, and some basic gates. The control part consists of logic, which provides signals to control the operations in the data part. Similar to other HDLs, UAHPL has its own set of conventions for transfers, connections, register indexing, and so on.

Conventionally, in UAHPL, all transfers into registers take place at the trailing edge of the clock pulse. Also, transitions between states of the finite state machine take place at the trailing edge. However, transfers to buses or input/output lines, called connections, are active for the entire duration of the clock pulse.

To aid in the construction of an efficient model, exercise hardware tradeoff, and to verify the logic of design, the environment is supported by a functional UAHPL simulator.

```
MODULE: CRCGENERATOR.
MEMORY: CRCREG{16};COUNT{6}.
EXBUSES: C;Z.
BUSES: X{6};Y;ZOUT;CRCRDY.
EXINPUTS: CLK;RESET;START.
EXINPUTS: MESIN;SELECT.
CLUNITS: INC{6}  <: INCR <. 6 . >.
BODY SEQUENCE: CLK.
   1 COUNT<=6$0;
     CRCREG<=16$0;
     =>¯(START)/(1).
   2 ZOUT=MESIN;
     Y=MESIN@CRCREG{15};
     COUNT,<=X;
     CRCREG,=(Y,CRCREG{0:3),CRCREG{4}@Y,CRCREG{5:10},
                    CRCREG{11}@Y,CRCREG{12:14} !
        (Y,CRCREG{0},Y@CRCREG{1},CRCREG{2:13},
                    CRCREG{14}@Y)*(SELECT),¯SELECT);
     =>¯(&COUNT) /2.
   3 COUNT<=X;CRCRDY=\1\ ;
     CRCREG<=\0\ ,CRCREG{0:14};
     ZOUT=CRCREG{15};
     =>(¯(&/COUNT{2:5}),COUNT{1})/(3,1).
   ENDSEQUENCE
CONTROLRESET (RESET)/(1);
        X=INC(COUNT);
        Z=ZOUT;
        C=CRCRDY.
END
CLU: INCR(X) <. I .>.
   "I-BIT INCREMENTER CONSTRUCTED WITH  EXCLUSIVE OR GATES"
   "AND GATES AND AN INVERTER"
   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. "INCR"
```

6. UAHPL model of a 64-bit programmable CRC generator.

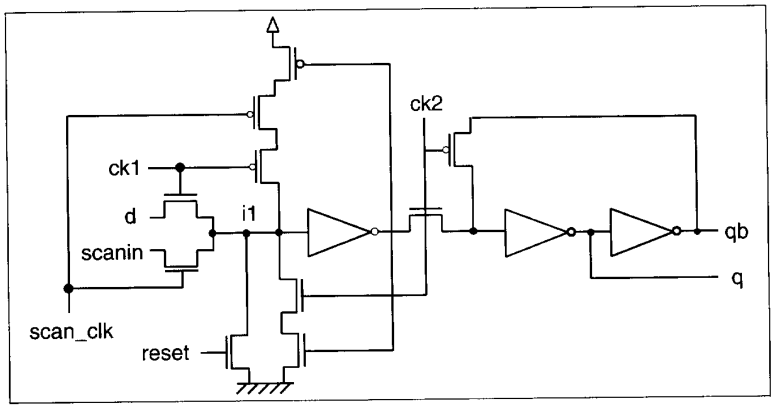
UAHPL Compiler and Functional Simulator

As shown in the outline of the system (Fig. 1), the UAHPL program serves as the input to the compiler. This basically performs the task of breaking down the input into tokens (lexical analysis) syntax analysis on the input (parsing), and storing the input in an intermediate form made of tables (semantics actions).

Full details can be found in [8], as can the full syntax of the UAHPL language. The task of syntax analysis is performed using a bottom up, table driven SLR parser. It uses two stacks for parsing. The parser calls the lexical analyzer to get the tokens from the input UAHPL program. When the syntax analyzer encounters the left side of text, it calls the semantic analyzer. If entries can be applied then, the program does so in the tables, which is an intermediate step in representing the language. In all, 16 tables are produced. One of the tables, the symbol table, is stored as a four column array, while

```
Begin
Allocate $k$ flip-flops with input of $i_{th}$ flip-flop being $D_i$ and output $Q_i$
Allocate $k$ OR gates with output of $i_{th}$ OR gate being $OR_i$

For $i$=1 to $k$
    1. Connect $OR_i$ to $D_i$
    2. **If** UAHPL step $i$ has no brance statement **Then** connect
       $Q_i$ to input of $OR_{i+1}$.
    3. In case of unconditional branch from step $i$ to $j$,
       connnect $Q_i$ to $OR_j$.
    4. **For** every conditional branch in step $i$ to step $l$ **Do**
            Synthesize comnbinational logic for the condition (Say $f$)
            Connect (($f$).AND.$Q_i$) to input $OR_l$
            Connect (^($f$).AND.$Q_i$) to next setp in sequence.

    Connect $Q_i$ to $C_i$.
**Endfor**
Replace all single input $OR$ gates with a direct connection.
**End.**
```

7. Systematic construction of control part.



8. Transfer level circuit diagram of the basic D flip-flop (dsr2s).

the others are kept in common storage. The tables are dynamic in nature and the required memory is allocated dynamically. Sharing of data is implemented by means of a director array for each table.

If the design is found to be error free, syntactically, and the entire system has been compiled, the stage1 dumps the contents of the table. If there are errors in the design, it flags the errors and produces a listing to help in debugging.

Once the UAHPL description has been translated into intermediate form, it is possible to verify the behavior of the designed system. A functional simulator at the RTL level is fast and provides feedback about the design that can be conveniently incorporated. This is in contrast to the higher level systems, where the feedback from design is difficult to incorporate.

Full details of simulator implementation can be found in [10]. The simulator takes its input through a command file. The input directives to the simulator include information on initial state, initialization of memory/registers, set watch on input/output lines, and so on. There are two components to the simulator. The first component performs lexical, syntax and semantic analysis of the input and produces an intermediate representation of the command file. The actual simulator uses the output of stage1, which is an intermediate form of the UAHPL program and the compiled intermediate form of the simulation directives. It produces a log of the simulation output.

Logic Synthesis and Simulation

After a design is functionally correct, we can proceed to do logic synthesis. The intermediate representation produced from stage1's compiler is used for this purpose. As explained in the Overview, the task of synthesis involves the identification of the data and control parts of the system. The description of the language implicitly assumes a number of features of the underlying hardware implementation. These include a synchronous, negative-edge-triggered system. The exact algorithms used in the synthesis have been published earlier [8], and the previous section provided an intuitive insight in the process. Figure 7 provides the algorithm for extracting the control part of the system.

The stage2 compiler produces a logic level network interconnection list that is technology-independent. A doubly linked list structure is used to store the network.

Each element (gate/flip-flop) is represented as a node in the network. The data structure used to represent a node includes information on its type, inputs, outputs, and pointers to the input and output nodes. This stage generates two lists, a gate list containing information about each node of the network; and an IO list, which contains the information on the inputs and outputs connected to the gates. Each line in the gate list has seven entries. These are the gate number, gate type, ilink (pointer to the IO-list for the list of inputs to the element), olink (pointer to the IO/list for the outputs connected to the element), symlink (symbolic list), siginp (sum of node numbers input to it), and siglink (pointer to another element with the same or a multiple of 10000 siginp number).

The first two parameters are used in optimization. The IO-list has rows with a pointer number for each row. The first number on a row is its pointer address, the next two numbers denote the element numbers that are either input/output (depending on whether the pointer is used under ilink or olink, and the fourth number is a pointer address for continuation. If the list has no continuation, the last entry is a zero.

Thus, once the logic of the UAHPL model is verified and the functional correctness guaranteed, the model is converted to hardware. This is done using the UAHPL hardware synthesizer. The output of the compiler is a net/list that contains the logic gates, flip-flops, and their interconnections. The netlist syntax can be found in [8, 9].

After logic synthesis, simulation can be done to verify the logic synthesis process. This requires a logic level simulator. RNL [12] is a simulator distributed with VLSI design tools [13], it was acquired along with the other tools and found to be sufficient. In order to integrate the RNL simulator, two programs were required:

1. A technology mapping program (the UAHPL netlist and the RNL cell library are different; and

2. A program that translates the UAHPL netlist into a format acceptable by RNL.

The task of technology mapping was required because the UAHPL logic synthesis tool is technology independent. It may, for instance, use an 8-input AND gate, which a technology may not support. RNL contains a library of basic logic gates. These two goals were achieved using locally developed programs. The first objective is met by the program postprocessor, and the second by the netlist converter.

Currently, The OASIS standard cell library is used in our implementation. Corresponding to each layout cell of the cell library is a logical/switch level model that can be simulated using the RNL logic level simulator.

The pre-processor program maps the UAHPL compiler generated netlist into the cells avail able in the cell library. The post-processor then translates these to netlist format accept able to RNL. Once the netlist is verified at the switch level, it is mapped to a layout.

Cell Libraries and Technology Mapping

In the standard cell approach, the cell library is the core of the physical design process. In our work, we used the "vanilla place and route" (VPNR) subsystem in the OASIS design environment [14]. The OASIS system comes with a standard cell library. The VPNR system uses cell based approach and it works with the OASIS cell library. The VPNR system is flexible in the sense that it is possible to define another standard cell library as long as some restrictions are met.

```
macro dsr2 (m_d_m_scanin m_reset m_ck1 m_scan_clk m_ck2 m_qb m_q)
(local n)
(cond
((== stdc 1) (printf "dsr2s INS%S (%S,%S,%S,%S,%S,%S,%S,%S);\n"
                n m_d m_scanin m_reset m__ck1 m_scan_clk m_ck2 m_qb m_q))
((== sctest 1) (printf "dsr2s INS%S (%S,%S,%S,%S,%S,%S,%S,%S);\n"
                  n m_d m_scanin m_reset m__ck1 m_scan_clk m_ck2 m_qb
m_q))
(t
(etrans m_ck1 n.13 m_d 3 2) ; m9
(etrans m_ck2 n.11 n.12 3 2) ; m2
...   ...   ...   ...   ...   ...
...   ...   ...   ...   ...   ...
(ptrans m_ck1 n.16 n.13 3 2) ; m15
(ptrans m_ck2 n.11 m_qb 3 2) ; m7
...   ...   ...   ...   ...   ...
...   ...   ...   ...   ...   ...
(capacitance m_d 0.002); d
(capacitance m_scanin 0.003); scanin
...   ...   ...   ...   ...   ...
...   ...   ...   ...   ...   ...
)))
```

9. Partial transistor level RNL circuit of dsr2s cell.

```
; _______________________________
; deff.net
; _______________________________
(macro deff
(d clk1 enable q clk2 scantest scanin Reset)
(local ffinput ffinput_bar ld_bar q_bar)
(il enable ld_bar)
(aoi22 d enable q ld_bar ffinput_bar)
(i1 ffinput_bar ffinput)
(dsr2 ffinput scanin Reset clk1 scantest clk2 q_bar q)
)
; End of macro deff
```

10. RNL macro for d flip-flop with enable (e\deff).

The VPNR system takes a logic level netlist of cells in vpnr format (RNL format files can be translated into vpnr format). The netlist is made from the constituents of a logic-level cell library. This logic-cell library reflects the lower level layout cell library. For each cell in the layout library, there exists a corresponding cell in the logic-cell library. Further, the netlist may consist of macro cells written in terms of the logic-cells. The VPNR program takes the input netlist and unfolds it in terms of the logic-cells to create a flat netlist made of logic-cells. Corresponding to these cells, fully characterized cells exist in the layout cell library. The placement program then proceeds to place these layout cells on the floor.

One of the constituents of the OASIS cell library is the D flip-flop with reset inputs. The logic-cell library consists of a logic-cell corresponding to the D flip-flop. This is obtained by extracting the layout. The circuit diagram for the flip-flop is shown in Fig. 8 and the transistor level logic-cell (RNL format) for it is shown in Fig. 9.

The library uses cells with a 2-phase clock, and all the cells are scannable (for testability). The UAHPL logic synthesizer produces logic netlist using the UAHPL primitive gates and flip-flops. As mentioned previously, the UAHPL generated netlist assumes an underlying hard ware that uses negative edge triggered D flip-flops for implementation. All of these flip-flops should also be resettable via a single external reset signal. Further, the UAHPL design assumes a single phase clock.

As part of the simulation process, we had translated the UAHPL netlist into RNL by defining macros for these gate types in terms of the RNL primitives. These primitives included all the standard gates. However, this netlist cannot be directly fed to a physical design system. The netlist input to VPNR system should contain only those leaf cells that have their layouts in the layout cell library.

This requires a netlist mapper. An input to this mapper is the UAHPL logic-cell library. The library contains the models of all the gates/flip-flops that are required by the UAHPL netlist in terms of OASIS logic-cell library natives. For example, the UAHPL system uses three types of D flip-flops: a D flip-flop with enable (data part), a D flip-flop with set (control flip-flop for initial state), and a D flip-flop with reset (for all other control flip-flops). The OASIS cell library contains only one resettable D flip-flop. The UAHPL logic-cell library contains macros for all the three types of flip-flops in terms of the OASIS D flip-flop primitive cell, dsr2s. The OASIS native D flip-flop is shown in Fig. 9, and the UAHPL macro of the D flip-flop with enable is shown in Fig. 10.

Physical Design

Standard-cell placement

Once the netlist is ready, it can be fed to the VPNR's placement program. This programs performs a pre-processing of the netlist before placement. It removes the clock and global signals from the netlist. Then it proceeds to layout the cells in rows (the number of rows and aspect ratios are user controllable). For placement, it uses the quadrisection min-cut placement algorithm [15]. This algorithm partitions the netlist into four quadrants recursively until each quadrant contains exactly one cell. The placement is accompanied and directed by approximate global routing.

Routing

After the placement of layout cells in rows and preliminary routing, the full routing phase is done in VPNR. First, the global routing of signal nets is done, then routing of global signals (e.g., clock etc.), and then channel routing. For global routing, VPNR considers each net sequentially. It constructs a minimum spanning tree for each net, finds exact locations for nets that need to cross rows, inserts feed-through cells in the row (if required), and assigns sub-nets in channels. The global signals are assumed to follow a fixed routing scheme. These extend horizontally in the channels on either side of a vertical rail. Channel routing is done to generate the routed circuit. VPNR has two choices for the channel router, a greedy router [16] and a left-edge based router with channel compaction [17].

Layout Assembly

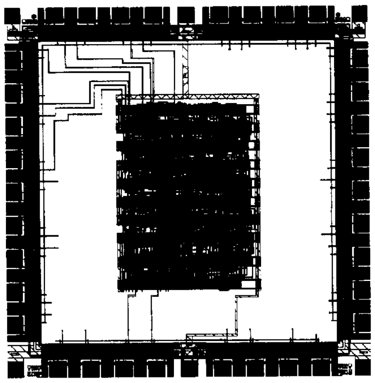
Once the channel routing is completed, the task of generating the complete layout is done using the Magic layout system. The description of the placed circuit is converted from the vpnr format into Magic's format. The placed cells are interconnected using the channels and interconnection wire, which are written out in Magic format by the routing program. Magic is used to assemble the layout and to connect the vertical power rails.

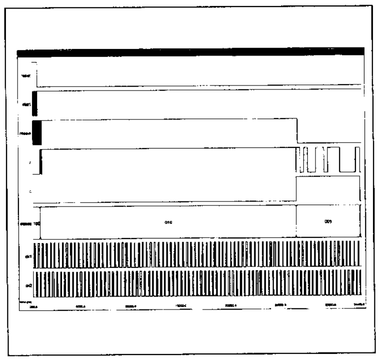
After the layout is ready, the design is placed in a padframe, and the leads are bonded. Many foundries offer their own pad frames or utilities to generate pad frames. For the general case, one can provide the pads and choose from the standard packaging dies of that particular foundry.

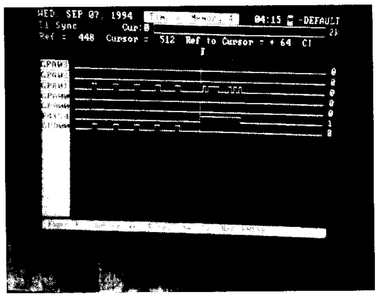
Layout Extraction and Verification

Although the translation of transistor netlist to layout is error free, the accuracy of the layout cannot be guaranteed, even if the netlist is functionally correct. Wiring delays, for example, may introduce clock skews, thus resulting in a broken design. Simulation of the layout can be done to verify the behavior of the system. In order to do that, the circuit under lying the layout must be extracted.

The design automation system uses Magic's circuit extractor. The extractor is


11. Layout of programmable CRC chip.


12. Simulation of the extracted programmable CRC circuit.


WED. SEP 07, 1994 04:15 -DEFAULT
T1 Sync Cur:
Ref = 448 Cursor = 512 Ref to Cursor = + 64 C1

13. Logic analyzer output ofr the fabricated CRC chip.

both incremental (only part of the layout is re-extracted after any change) and hierarchical. The extractor produces a separate file (*ext* format) for each Magic file (*mag* format) in a hierarchical design. Currently, very few tools (SPICE, RNL) can support the *ext* format. Format conversion utilities such as ext2spice and ext2sim adapt the extracted circuits to SPICE and RNL, respectively. In addition, the *ext*'s extracted circuit can be used with Magic's interactive simulator, IRSIM. Details of the operation are given in [11].

If there is to be correlation between extracted parameters and the fabrication process, the process parameters have to be obtained from the foundry, where the design is fabricated. These include their process parameters for simulation of the circuit using different type of simulators. In our case, IRSIM and SPICE parameter formats were used. The system can use two simulators, RNL and IRSIM. The RNL logic level simulator can also be used to simulate the extracted circuit. RNL takes the *sim* files, which may be either extracted from the layout or translated from the logical level *netlist* of the circuit. RNL's advantage is that the same simulation stimulus input file can be used at the logic and extracted layouts, and the results compared.

IRSIM is an *event* driven simulator with two supported modes. In the switch mode, each transistor is modeled as a voltage controlled switch. In the linear mode, each transistor is modeled as a resistor in series with a voltage controlled switch. It uses a different model for computing node values and transition times than RNL. IRSIM accepts a parameters file defining the electrical parameters of the simulated devices. It defines the capacitance of the various layers, transistor resistances, threshold voltages, etc. This data is obtained from the foundry fabricating the design. The simulation thus accurately reflects the parameters of the devices and the functionality of the circuit can be verified before fabrication.

IRSIM offers a number of valuable features. Among them is the facility to use it interactively in the Magic layout editor. Thus, instead of specifying node names, one can choose a node in the layout and find its logical value. It can also be run non-interactively. In addition, it is supported by a logical analyzer tool for graphically representing the simulation results. It is integrated well with the Magic layout system.

A typical finished layout, in this case the programmable CRC chip previously discussed, is shown in Figure 11. The simulation result is shown in Fig. 12. Simulation was performed using process parameters generated at Orbit Semiconductor Inc., where this circuit was later fabricated.

Fabrication and Testing

Once the design has been verified, it can be fabricated. At the moment, the design automation system has no generalized method of test ing the ICs. Depending on the functionality, a test circuit is breadboarded and tested using microcontroller based pattern generator and logic analyzer. For the programmable CRC chip, the output of the logic analyzer is shown in Fig. 13.

Summary and Conclusion

In this article, we discussed the progress made at KFUPM in putting together an UAHPL based design automation system using software tools developed locally and in U.S. universities. The UAHPL language is used as the front-end specification medium because of its close relation to hardware implementation issues. The system is modular and technology independent so that future extensions and specific implementation issues can be added and modified.

The work carried out so far has targeted semicustom design and standard cell methodologies. The front-end of the system, up to the netlist, is independent of technology and architecture. The output of this stage can be conveniently mapped to programmable devices (PLDs) and field programmable gate arrays (FPGAs). Current research on the extensibility includes the task of synthesis from algorithmic specification. The approaches under consideration include the use of high level languages, e.g., Pascal and C subsets for input specification. The use of a VHDL subset is also under investigation. The results from these studies can be easily integrated into this design automation system to convert it from an RTL level to an algorithmic level system. Other work includes the investigation of a formal synthesis option that will do away with the comparative simulation approach to verification.

A number of existing RT-level systems such as OASIS incorporate the concept of design for testability (DFT) and test vector generation. With the current complexity of systems, how ever, it is nearly impossible to design test vectors manually. Incorporating testability into a design is done in OASIS, and ASIC Synthesizer using automatic insertion of scan flip-flops. The choice of OASIS tools led to development of these features in the physical design. However, the logic synthesized by our UAHPL-based system was sometimes redundant. As a result, some designs were not fully testable. The area of synthesis for testability, especially at the RTL level, is receiving significant attention today, and work has been done on re-synthesis for testability [18]. Other work is focused on exploring techniques for redundancy removal from the synthesized UAHPL netlists such that the designs can be fully test able. A rapid prototyping facility would be invaluable to support the design automation system. There are proposals for adopting FPGAs for rapid in-house prototyping, and the use of FPGA based reconfigurable system for testing the designs [19, 20].

Acknowledgments

The authors wish to thank King Fahd University of Petroleum and Minerals, which supported our work. We also thank the King Abdul Aziz City for Science and Technology (KACST) for support in the form of research grant AR 11-21. Research teams at MCNC, North Carolina (OASIS); University of California, Berkeley (Magic) and (VLSI design tools); University of Arizona (UAHPL compiler, simulator) are also gratefully acknowledged. **CD**

Sadiq M. Sait is an Associate Professor in the Department of Computer Engineering at King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia. He specializes in digital design automation, VLSI system design, and computer architecture. Professor Sait is the author of *VLSI Physical Design Automation: Theory and Practice*, McGraw-Hill Book Co., Europe, 1995.

Muhammad S. Benten is Dean of the College of Computer Sciences and Engineering at King Fahd University of Petroleum and Minerals. His areas of expertise include parallel processing, computer networks and VLSI design automation.

Asjad M. T. Khan is currently a lecturer in the Computer Engineering Department at King Fahd University of Petroleum and Minerals.

References/Further Reading

1. J. Rabaey, H. De Man, J. Vanhoof and F. Catthoor: "CATHEDRAL II: a synthesis system for multiprocessor DSPs," in *Silicon Compilation*, D. D. Gajski, Ed. Reading, MA: Addison-Wesley, 1988, pp 311-360.

2. G. Zimmermann, "MDS: The Mimola design method," *Journal of Digital Systems*, **1**(3), pp 337-369, 1980.

3. Reinaldo A. Bergamaschi and Andreas Kuehlmann, "A system for production use of high-level synthesis," *IEEE Transactions on VLSI*, **1**(3), pp 233-243, September 1993.

4. Jorg Biesenack et al., "The Siemens high-level synthesis system, CALLAS," *IEEE Transactions on VLSI*, **1**(3), pp 244-253, September 1993.

5. A. A. Jerraya et al., "AMICAL: interactive high level synthesis environment," *Proc. EDAC 93*, Paris, Feb. 1993.

6. A. Greiner, and F. Pecheux. ALLIANCE: a complete set of CAD tools for teaching VLSI design, *3rd Eurochip Workshop on VLSI Design Training*, pp 230-237, Grenoble, September 1992, available via ftp from ftp.ibp.fr under the directory /ibp/softs/masi/alliance Feb. 1994.

7. M. J. S. Smith, "More logic synthesis for ASICs," *IEEE Spectrum*, pp 44-48, November 1992.

8. M. Masud, Modular Implementation of a Digital Hardware Design Automation System," Ph.D Dissertation, University of Arizona, 1981.

9. Fredrick. J. Hill and G. R. Peterson, *Digital Systems: Hardware Organization and Design*, 2nd edition, John Wiley and Sons, New York, 1978.

10. M. Al-Sharif, AHPL Simulator, M.S. Thesis. Electrical Engineering Department. Arizona State University, 1983.

11. Robert N. Mayo, et al., 1990 DECWRL/Livermore Magic Release, Digital Western Research Laboratory, September 1990.

12. C. Terman, "RSIM - A Logic-level Timing Simulator," Proceedings of IEEE International Conference on Computer Design, pp 437-440, October 1983.

13. VLSI Design Tools Reference Manual, Release 3.1, NW Laboratory for Integrated Systems, FR-35, University of Washington, February 1987.

14. K. Kozminski, Ed. "OASIS: open architecture silicon implementation system, Users Guide," Version 2.0, MCNC, Research Triangle Park, North Carolina, October 1992.

15. P. R. Suaris and G. Kedem, "A new approach to standard cell layout," *International Conference on Computer Aided Design*, pp 474-477, November 1987.

16. J. E. Rose, Greedy Algorithms for Wiring in VLSI," Masters Thesis, Department of Computer Science, North Carolina State University, 1985.

17. M. J. Lorenzetti, M. S. Nifong and J. E. Rose, Channel routing for compaction," *Proceedings of the International Workshop on Placement and Routing*, May 1988.

18. S. Bhattacharya, Franc Brglez, and S. Dey, "Transformations and resynthesis for testability of RT-Level control-data path specifications," *IEEE Transactions on VLSI*, **1**(3), pp 304-318, September 1993.

19. A. El-Gamal et al., "An architecture for electrically configurable gate arrays," *IEEE Journal of Solid State Circuits*, **24**, pp 394-398, April 1989.

20. David E. Van Den Bout et al., "AnyBoard: an FPGA-Based, reconfigurable system," *IEEE Design and Test*, pp 21-30, September 1992.

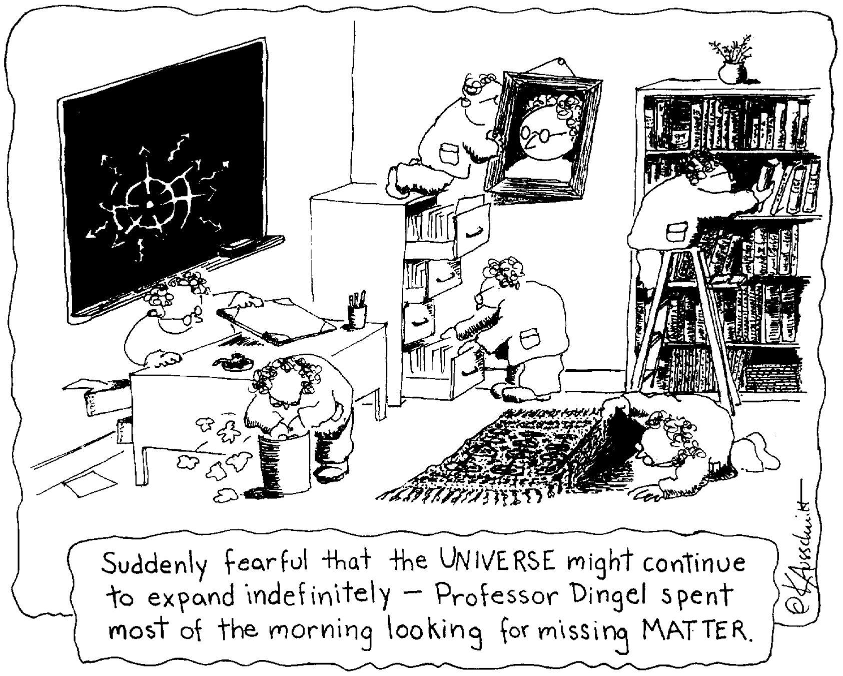
21. M. Masud and Sadiq M. Sait, "Universal AHPL— a language for VLSI design automation," *IEEE Circuits and Devices Magazine*, September 1986.

22. Sadiq M. Sait, "Integrating UAHPL-DA system with VLSI design tools to support VLSI DA courses," *IEEE Transactions on Education*, **35**(4), pp 321-330, 1992.

23. Sadiq M. Sait and Habib Youssef, *VLSI Design Automation: Theory and Practice*, McGraw Hill Book Co., Europe, 1994.

24. Sadiq M. Sait and Shahid M. Tanvir, "VLSI layout generation of a programmable CRC chip," *IEEE Transactions on Consumer Electronics*, pp 911-916, November 1993.

25. Sadiq M. Sait and Muhammad S. T. Benten, An Integrated Design Automation System for Generating VLSI Layouts. Progress Reports #1 to #5, KACST Project AR 11-21.


Suddenly fearful that the UNIVERSE might continue to expand indefinitely — Professor Dingel spent most of the morning looking for missing MATTER.