Register Minimization beyond Sharing among Variables*

Tsung-Yi Wu Youn-Long Lin

Department of Computer Science
Tsing Hua University, Hsin-Chu, Taiwan 30043, R.O.C.
E-mail: {dr818304, ylin}@cs.nthu.edu.tw

Abstract

Traditionally, it is assumed that every variable in the input HDL (Hardware Description Language) behavioral description needs to be held in a register; A register can be shared by multiple variables if they have mutually disjoint lifetime intervals. This approach is effective for signal-flow-like computations such as various DSP algorithms. However, it is not the best for the synthesis of control-dominated circuits, which usually have variables/signals of different bit-width as well as very long lifetime. To go beyond register minimization by lifetime-analysis-based sharing, we propose holding some variables in the state registers, some signal nets, or some unclocked sequential networks. We have implemented the proposed method in a software program called VReg. Experimental results have demonstrated that VReg minimizes the number of registers more effectively than the lifetime-analysis-based approach does. Better register minimization also leads to both smaller area and faster designs.

Key Words: High-Level Synthesis; Control-Dominated Circuit; Storage Synthesis.

1 Introduction

The ultimate goal of high level synthesis research and development is to enable the designer specifying what he wants via a behavioral description in an HDL (Hardware Description Language). Despite all the potential advantages (e.g., higher productivity, correct-by-construction, ...etc) promised by the HLS (High Level Synthesis) community, the acceptance of HLS by the mainstream designers lies on whether the quality (i.e., performance, area, power consumption, and testability) of the synthesized results is competitive compared with that of manual designs. Therefore, it is important to make every effort to optimize every aspect of the synthesis process.

*Supported in part by a grant from the National Science Council of R.O.C. under contract no. NSC84-2215-E-007-045.

32nd ACM/IEEE Design Automation Conference ®

Storage elements such as registers and memory usually occupy a significant portion of the chip area. Therefore, it is important to maximize their utilization. All high level synthesis systems found in the open literature employ one technique or another for storage synthesis.

Traditionally, register allocation and variable-to-register binding are done based on the lifetime analysis of variables [3]. The lifetime of a variable is the time interval between the first definition and the last use of the variable. Multiple variables can share the same register if their lifetimes do not overlap with one another. Many techniques [6][7][8][9] have been developed for allocating as few registers as possible for all variables taking advantage of the sharing opportunity.

The lifetime-analysis-based approach is effective for certain application domains (e.g., DSP) where many temporary variables are used to hold intermediate computation results. Recently, HLS for control-dominated circuits [11] has attracted more and more attention because of their widely use in control, communication, and embedded systems. While some techniques for some subtasks in the HLS of computation-intensive circuits are still applicable to HLS of control-dominated circuits, we realize that the lifetime-analysis-based register allocation technique is inadequate because most control-dominated circuits have drastically different characteristics from a computation-intensive one in terms of their variables in the HDL program.

In a control-dominated circuit, variables are mostly used to hold commands or control signals. Their word-lengths vary greatly from a single bit to multiple bytes. Because very little computation is performed, temporary variables are seldom used. Hence, most variables have very long lifetime. Therefore, a lifetime-analysis-based approach usually performs poorly because not much sharing opportunity is there.

In this contribution, we try to minimize the register cost for a control-dominated circuit using a different approach. Because a variable might be a function of some other variables, the controller state, some signal nets, or a mixture of above, we can replace the register and its associated combinational circuit with some simple circuit.

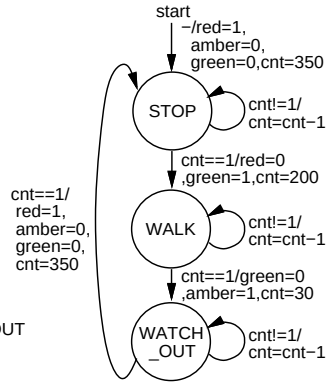
An example is the traffic light controller [10]. In its Verilog HDL description (Figure 1(a)), a variable, *red*, represents the status of the red light which will stay ON (logic one) for 350 clock periods before the green light is turned on. A traditional synthesizer will allocate for *red* a

register due to the existence of the *procedural assignment statements* "*red = 0*" and "*red = 1*" in the HDL code. However, we observe that the allocation is unnecessary because the value of *red* is a function of the controller state. Variable *red* is ON if, and only if, the controller is in state STOP (Figure 1(b)). Therefore, *red* can be controlled by, instead of a dedicated register, a simple combinational circuit decoding the state register of the controller.

```
module TLC_Vg(clock,red,amber,green);
input clock;
output red,amber,green;
reg red,amber,green;

always
begin
  red = 1;
  amber = 0;
  green = 0;
  // wait 350 clock period; state STOP
  repeat (350) @(posedge clock);
  red = 0;
  green = 1;
  // wait 200 clock period; state WALK
  repeat (200) @(posedge clock);
  green = 0;
  amber = 1;
  // wait 30 clock period; state WATCH_OUT
  repeat (30) @(posedge clock);
end
endmodule
```

(a)



(b)

Figure 1: A Traffic Light Controller Example: (a) its behavioral description in Verilog HDL, (b) its state transition graph.

In the next section, we will define the state transition graph (STG) that describes a circuit input to our register minimization algorithm. The basic idea behind our proposed approach is presented in Section 3. An algorithm implementing the proposed approach is described in Section 4. In Section 5, we illustrate how the algorithm works using a walk-through example. Comparison with three synthesizers on a set of benchmarks is presented in Section 6. Finally, we conclude this paper in Section 7.

2 State Transition Graph

We use a state transition graph (STG) to capture the behavior of a circuit described in HDL. The STG is essentially a *synchronous Mealy machine* [4]. In the graph, each vertex represents a state derived from a synchronization statement such as "*@(posedge clock)*" in Verilog or "*wait until clock'event and clock='1'*" in VHDL. Each directed edge represents a state transition and is labeled with a *condition/action* pair.

To satisfy the timing specification, the machine represented by the STG will transit itself from one state to the next upon the occurrence of the positive edge of the system clock, and all actions will be performed only during the transition. Hence, the content of a register in a synchronous Mealy machine can only be updated during a state transition.

For instance, the Verilog description shown in Figure 1(a) is compiled into the STG depicted in Figure 1(b). In the graph, variable *cnt* is created for statement *repeat* as a counter, and states STOP, WALK, and WATCH_OUT are generated for statements "*repeat(350) @(posedge clock)*", "*repeat(200) @(posedge clock)*", and "*repeat(30) @(posedge clock)*", respectively.

3 Basic Idea

The execution of a Verilog *procedural assignment statement* [10] (equivalent to a VHDL *sequential assignment statement* [5]), $a = f(b_1, b_2, \ldots, b_l)$, at time t will give variable a the value $f(b_1^t, b_2^t, \ldots, b_l^t)$, where b_j^t $(j = 1, 2, \ldots, l)$ is the value of variable b_j at time t. Because the value of $f(b_1^t, b_2^t, \ldots, b_l^t)$ may be different from the value of $f(b_1^{t+\Delta t}, b_2^{t+\Delta t}, \ldots, b_l^{t+\Delta t})$, it is necessary to allocate a register for variable a.

However, a Verilog *continuous assignment statement* [10] (equivalent to a VHDL *concurrent signal assignment statement* [5]), *assign* $a = f(b_1, b_2, \ldots, b_l)$, will always assign $f(b_1^t, b_2^t, \ldots, b_l^t)$ (where t is the current time) to variable a. In this case, variable a can be directly implemented with a combinational logic that performs $f(b_1, b_2, ..., b_l)$ without any register.

Under certain conditions, a procedural assignment statement for a variable can be converted into a continuous assignment statement. For instance, the procedural assignment "$a = b \ \& \ d$" associated with edge $e_{0,1}$ in Figure 2(a) propagates its assignment value "$b \ \& \ d$" to both state s_1 and state s_2 as the function of variable a as shown in Figure 2(b). With this result, the procedural assignment "$a = b \ \& \ d$" affecting state s_1 and state s_2 can be represented by the continuous assignment statement "*assign* $a = (current\_state == s_1)$ & $(b \ \& \ d)$ | $(current\_state == s_2)$ & $(b \ \& \ d)$".

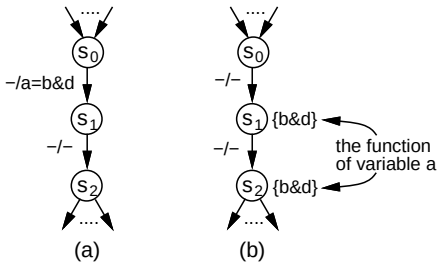


Figure 2: Propagation of a procedural assignment statement: (a) The original STG, (b) The modified STG after propagation.

After each procedural assignment statement for each variable has been translated into an *assignment action* of the STG, the value that will be latched by $a.reg[m]$, the m-th bit of register $a.reg$, can be formally described by the following *Boolean expression*:

$$\sum_{i,j \in state\ space} b(s_i) \cdot c_{i,j} \cdot f_{i,j}^a[m], \tag{1}$$

where s_i is the i-th state, $b(s_i)$ is 1(0) if the current state is (is not) s_i, $c_{i,j}$ is the condition controlling the transition along the edge $e_{i,j}$ from s_i to s_j, and $f_{i,j}^a[m]$ is the function that would be assigned to $a.reg[m]$ when the transition does indeed take place. Note that $f_{i,j}^a[m]$

is equivalent to $a.reg[m]$ if there is not assignment statement for $a.reg[m]$ alone $e_{i,j}$. Upon every positive edge of the system clock, $a.reg[m]$ latches $\sum b(s_i) \cdot c_{i,j} \cdot f^a_{i,j}[m]$ be it new or unchanged.

For example, the Boolean expression of the value latched by register *red.reg* of variable *red* in Figure 1(b) is

$$b(s_0) \cdot (cnt \neq 1) \cdot red.reg + b(s_0) \cdot (cnt == 1) \cdot 0 + \\ b(s_1) \cdot (cnt \neq 1) \cdot red.reg + b(s_1) \cdot (cnt == 1) \cdot red.reg + \\ b(s_2) \cdot (cnt \neq 1) \cdot red.reg + b(s_2) \cdot (cnt == 1) \cdot 1$$

if states STOP, WALK, and WATCH_OUT are denoted by s_0, s_1, and s_2, respectively.

A register *a.reg* can be minimized if the value of each of its bit, $a.reg[m]$, can be described by the following Boolean expression:

$$\sum_{i \in state\ space} b(s_i) \cdot f^a_i[m], \qquad (2)$$

where $f^a_i[m]$ is the *function* of $a.reg[m]$ during state s_i. The value of variable $a[m]$ (associated with $a.reg[m]$) can be represented by a continuous assignment "$assign\ a[m] = (b(s_0)\ \&\ f^a_0[m]) \mid (b(s_1)\ \&\ f^a_1[m]) \mid ... \mid (b(s_j)\ \&\ f^a_j[m])$" if the state space is $\{s_0,s_1,...,s_j\}$; Therefore, the register can be eliminated.

For instance, after assignment propagation of variable *red* (Figure 3), the value of *red.reg* is represented by the Boolean expression

$$b(s_0) \cdot 1 + b(s_1) \cdot 0 + b(s_2) \cdot 0$$

if states STOP, WALK, and WATCH_OUT are denoted by s_0, s_1, and s_2, respectively. Therefore, *red.reg* can be eliminated and, thus, variable *red* is represented by

$$assign\ red = b(s_0)\ \&\ 1 \ \mid \ b(s_1)\ \&\ 0 \ \mid \ b(s_2)\ \&\ 0$$

or, after logic minimization, $assign\ red = b(s_0)$.

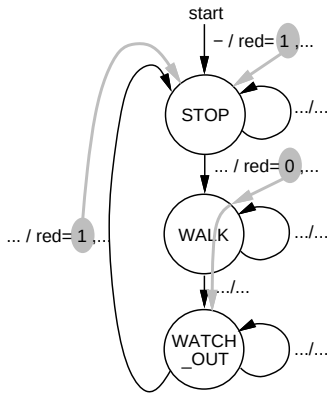


Figure 3: The process of function Find_Func for *red*.

Unclocked sequential networks can also be used to replace certain registers. We illustrate it with an example. If variable a has value $f(b_1, b_2, \ldots, b_l)$ during the current state, s_i, and will not change its value during the next state, s_j, then it can be represented by $f(b_1, b_2, \ldots, b_l)$ during state s_i. However, we cannot do so during state s_j if any of b_1, b_2, ..., or b_l is updated during state s_j. Using a lifetime-analysis-based method, it would be impossible to implement the circuit for variable a without using any register for holding the old value. However, via an unclocked feedback network, variable a can hold its original value during state s_j. Hence, variable a can be described by

$$assign\ a = b(s_i)\ \&\ f(b_1, b_2, ..., b_l) \ \mid \ b(s_j)\ \&\ a \ \mid \ \ldots$$

which corresponds to the unclocked feedback network depicted in Figure 4.

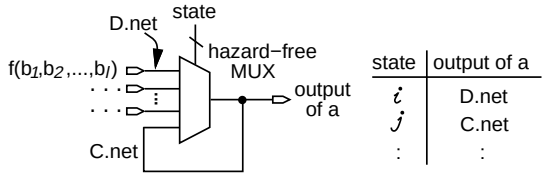


Figure 4: The implementation of "$assign\ a = b(s_i)\ \&\ f(b_1, b_2, ..., b_l) \ \mid \ b(s_j)\ \&\ a \ \mid \ \ldots$" as an unclocked feedback network.

The drawback of using an unclocked feedback network is the difficulty in initializing the output to an arbitrary value.

In summary, a register allocated for variable a can be eliminated if all procedural assignment statements for variable a can be transformed into and described by a single continuous assignment. However, one must be aware of that the cost of the combinational circuit for implementing formula (1) may be more expensive than that for formula (2).

4 Algorithm

In this section, we propose an algorithm that implements the approach described in the previous section. We assume that register allocation has been done beforehand. Our objective is to improve its quality.

Figure 5 shows a pseudo-code description of the proposed algorithm. First, the algorithm finds all functions of each register bit during each state using subroutine *Find_Func*. Each assignment action, $r_m = f(b_1, b_2, \ldots, b_l)$, of edge $e_{j,k}$ will propagate $f(b_1, b_2, \ldots, b_l)$ to all states reachable from state s_k till the value of any of r_m, b_1, b_2, ..., or b_l is updated. If this termination is not due to the updating to the value of r_m, then r_m is propagated to all reachable states till the value of r_m is updated.

A function can be used to replace a register. If f(b_1, b_2, ..., b_l) is a function of a register r_m during state s_j, then $b_1 \neq r_m$, $b_2 \neq r_m$, ..., and $b_l \neq r_m$. If either $b_1 == r_m$, $b_2 == r_m$, ..., or $b_l == r_m$, the function f(b_1, b_2, ..., b_l) cannot be implemented without register r_m. We term r_m irreducible if either $b_1 == r_m$, $b_2 == r_m$, ..., or $b_l == r_m$.

Register r_m is also irreducible if any of b_1, b_2, ..., or b_l is updated and the assignment $r_m = f(b_1, b_2, \ldots, b_l)$ takes place along the same edge.

A register assigned by a function of any asynchronous signals is irreducible because an asynchronous signal can

change its value any time while a register can only be updated during state transitions.

Furthermore, if a register r_i assumes multiple functions during a state s_j and these functions are directly propagated from the assignment action along an incoming edge of s_j, then r_i cannot be reduced (this special case can only be caused by multiple fan-ins of state s_j) because the machine cannot determine which function to use. The problem is solved by splitting a state into multiple states such that each state can determine its own function. The task of state splitting is done by subroutine *State_Split* shown in Figure 6.

Figure 7 depicts a state-splitting example. Because the functions of register a during state s_2 are "$\{b+d, x+y\}$", it is necessary to split s_2 into two states. One state ($s_{5'}$) will give $b+d$ to register a, while the other state ($s_{6'}$) will give $x+y$ to register a.

The innermost for-loop of the algorithm chooses proper resources to replace a register r_i during each state s_j. The resources include registers other than r_i, nets, states and unclocked sequential networks.

Each register r_i that can be eliminated is marked and implemented with a *virtual register* by subroutine *Replace_Register_by_Virtual_Register*. A virtual register is either a combinational circuit or an unclocked sequential network and will be implemented by a multiplexer with the input signals from array *in*. The multiplexer steers signal *in*[*j*] to the output if s_j is the current state. If *in*[*j*] of register r_i is r_i itself, then the output is fedback to the input. Obviously, the data-path of a virtual register can be described using a Verilog *continuous assignment statement*.

5 A Walk-Through Example

In this section, we use the traffic light controller example of Figure 1 to illustrate how the proposed algorithm works. Initially, for the STG shown in Figure 1(b) four registers (red.reg, amber.reg, green.reg, and cnt.reg) are allocated to variables red, amber, green and cnt, respectively. If states STOP, WALK, and WATCH_OUT are denoted as s_0, s_1, and s_2, respectively, the data path (before logic minimization) for red.reg would be like that in Figure 8(a).

By function *Find_Func*, logic 1 in statement "red = 1" is propagated to state STOP (s_0), and logic 0 in statement "red = 0" is propagated to both state WALK (s_1) and state WATCH_OUT (s_2) as the function of register *red.reg* (Figure 3). Figure 8(b) depicts this result. Via the innermost loop of the algorithm, the array *in* is obtained as depicted in Figure 8(c). Finally, the function *Replace_Register_by_Virtual_Register* can implement the description for variable *red* with the circuit (before logic minimization) depicted in Figure 8(d). The circuit shown in Figure 8(d) implements the same behavior (including the timing and function) as the circuit depicted in Figure 8(a). However, the circuit shown in Figure 8(d) does not use any register.

```
Algorithm Register_Minimization(STG)
 Find_Func(STG);
 for each reducible register r_i  /* eliminate register r_i */
  for each state s_j
   if (all functions of r_i at s_j are equal)
    then begin
     if (∀ fan-in edges of s_j, there is no assignment to r_i
     and the function of r_i isn't constant)
      then set in[j] to r_i;
      /* minimize by unclocked sequential network */
     else set in[j] to the function of r_i;
     /* minimize by registers, nets or states*/
    end
   else if (∀ fan-in edges of s_j, there is no assignment to r_i)
    then set in[j] to r_i;
    /* minimize by unclocked sequential networks */
    else begin
     State_Split(STG,r_i,s_j);
     exit and then redo Register_Minimization(STG);
    end
  endfor
  Replace_Register_by_Virtual_Register(array in);
 endfor
end Register_Minimization
```

Figure 5: The pseudo-code description of the *Register_Minimization* algorithm.

```
Subroutine State_Split(STG,r_i,s_j)
 for each function f_k' assigning value to r_i along any edge
 entering s_j
  create a new state s_k';
  for each edge e_x,j that has the action: r_i=f_k'
   replace e_x,j with e_x,k';
  endfor
 endfor
 if (∃ an edge entering s_j and there is no assignment to r_i
 along the edge)
  then begin
   create a new state s_q';
   for each edge e_x,j that has no assignment to r_i
    replace e_x,j with e_x,q';
   endfor
  end
 for each edge e_j,y
  for each newly created state s_r'
   create a new edge e_r',y with the same label as e_j,y;
  endfor
 endfor
 eliminate s_j;
end State_Split
```

Figure 6: The pseudo-code description of the *State_Split* subroutine.

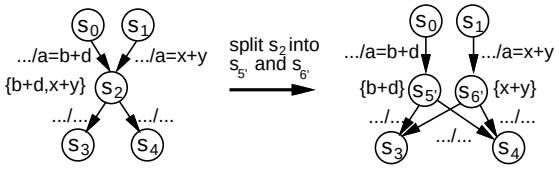


Figure 7: A state-splitting example.

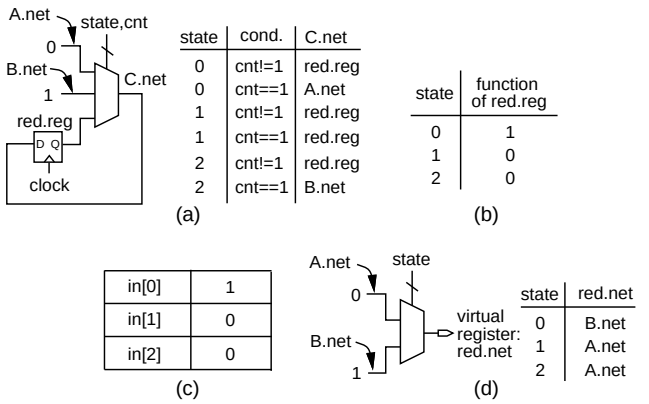


Figure 8: (a) A straightforward implementation for register red.reg, (b) functions for red.reg during each state, (c) array *in*, (d) the implementation for the virtual register of red.reg.

6 Results

We have implemented the proposed method in a software program called VReg. Several control-dominated circuits have been used to test the effectiveness of the proposed approach. They include a prefetch subcircuit (PREFETCH [1]), a counter (COUNT [1]), three versions of traffic light controllers (TLC [1], TLC_M [4] and TLC_Vg [10]), two versions of greatest common divisors (GCD [1] and GCD_M [4]), the FANCY chip (Fancy [1]), an LED display (LED_Dp [4]), a synchronous mealy machine(Smealy [4]), a "01" Pattern Detector (01PD), and a 3-phase generator (3-phase). Note that only benchmark PREFETCH, GCD series and Fancy make use of unclocked sequential networks.

Among the benchmarks, PREFETCH, COUNT, TLC and GCD have been synthesized by the CALLAS synthesizer [2]. VReg minimizes the register resource better than CALLAS did as shown in Table 1. Columns 2 and 3 give the number of register bits of the data-paths generated by CALLAS and VReg, respectively, for the four test cases. Column 4 shows the percentage of register bit reduction achieved by VReg over CALLAS.

In the next experiment, we compare VReg and MEBS [4] in terms of their effectiveness in register minimization and circuit performance. The measurement is done by a MEBS subroutine based upon MEBS's cell library. MEBS converts a behavioral VHDL description into a gate level netlist. The results are summarized in Table 2. Columns 2-5 give MEBS's synthesized results including the number of states, the maximum rising and falling delays of the critical path, the number of register bits of the data-paths, and the total cell area. Columns 6-9 gives the

Table 1: Experiment results I.

Design	#Register bits		Reduction
	CALLAS	VReg	
PREFETCH	162	96	41%
COUNT	10	8	20%
TLC	13	7	46%
GCD	49	33	33%

results when VReg is used as a preprocessor before MEBS is invoked. In all cases except COUNT not only the total cell area but also the critical path delay have been improved. The reductions of the register bits and the total cell area are shown in columns 10 and 11, respectively. For PREFETCH, TLC_M, GCD_M, and Fancy, no measurement was reported because MEBS does not accept the description style after VReg's preprocessing for those particular cases. For LED_Dp, Smealy, and 01DP, the critical path delays are significantly shorten because VReg has simplified their respective output port circuits.

In the final experiment, we compare VReg against a commercial logic synthesis tool, the Design Analyzer™ (Version v3.1a) from Synopsys. The cell library used is Synopsys's class library, and all benchmarks are rewritten in Verilog HDL. Some circuits were left out of this experiment because of the Design Analyzer's constraints on description style. The experimental results are shown in Table 3. Like previous experiment, both the number of register bits and total cell area have been significantly reduced when the circuit is first preprocessed before feeding to the Design Analyzer.

Finally, we show the schematic drawing of one of the synthesized results, LED_Dp. Figure 9(a) shows the result of directly applying the Design Analyzer, while Figure 9(b) depicts the results due to the combined effort of VReg and the Design Analyzer. Simulated by the Verilog-XL simulator from Cadence, both circuits exhibit the same timing behavior.

7 Conclusions

We have proposed a new method to minimize the number of register bits for the synthesis of control-dominated circuits described in behavioral HDL. The method is based on the relationships between variables and variables, between variables and states, and between variables and signal nets. A register holding a variable can be minimized if the variable can be derived, with a simple logic, from other resources.

We have realized the proposed approach in a software called VReg. We have conducted a series of experiments to evaluate its effectiveness. The experimental results demonstrated that VReg is indeed a valuable preprocessor to a behavioral synthesizer. Minimizing the number of register bits also lead to both reduction in the total circuit size and improvement in critical path delay.

Table 2: Experiment results II.

Benchmarks	MEBS				VReg + MEBS				reduction	
	#states	MR/MF†(ns)	#reg	area‡	#states	MR/MF(ns)	#reg	area	#reg	area
PREFETCH*	3	173.8/174.5	192	2336	3	–	96	–	96	–
COUNT	1	44.9/45.6	8	110	1	44.9/45.6	8	110	0	0%
TLC_M	6	85.8/89.8	13	304	6	–	7	–	6	–
GCD_M	2	117.2/105.3	24	871	2	–	16	–	8	–
Fancy	6	♮	72	1504	7	♮	63	–	9	–
LED_Dp	12	64.6/65.2	10	288	12	35.6/25.8	0	214	10	26%
Smealy	5	21.7/23.1	1	58	7	19.6/19.6	0	49	1	16%
01PD	3	20.2/17.9	1	27	3	14.9/17.2	0	16	1	41%

*: rewritten as the acceptable style of MEBS. ‡: The area of a inverter cell is 1. ♮: MEBS failed to report it.
†: MR and MF are the maximum rising and falling time of the critical path, respectively.

Table 3: Experiment results III.

Benchmarks	Synopsys				VReg + Synopsys				reduction	
	#states	#nets	#reg	area	#states	#nets	#reg	area	#reg	area
COUNT	1	36	8	120	1	36	8	120	0	0%
Fancy	6	488	72	1344	7	468	63	1274	9	5%
LED_Dp	12	60	10	171	12	27	0	69	10	60%
Smealy	5	30	1	71	7	22	0	54	1	24%
01PD	3	12	1	38	3	10	0	26	1	32%
TLC_Vg	3	88	12	234	3	84	9	205	3	12%
3-phase	3	13	3	58	3	10	0	26	3	55%

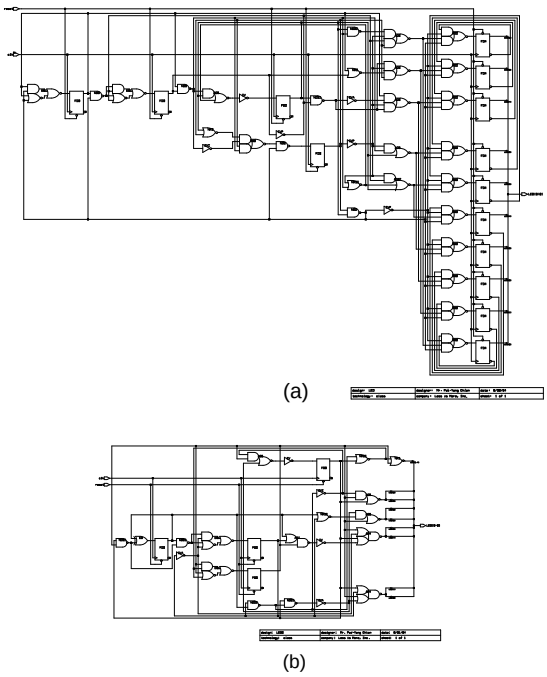


Figure 9: The schematic of the synthesized LED_Dp: (a) by the Design Analyzer, (b) by VReg and the Design Analyzer.

References

[1] "Benchmarks for the 5th & 6th International Workshop on High-Level Synthesis," Available through electronic mail at HLSW@ics.uci.edu.

[2] J. Biesenack, M. Koster, etc., "The Siemens High-Level Synthesis System CALLAS," *IEEE T. VLSI Systems*, Vol. 1, No. 3, pp. 244-252, September 1993.

[3] D. Gajski, N Dutt, A. Wu, and S. Lin, *High Level Synthesis: Introduction to Chip and System Design,* Kluwer Academic Pub., 1992.

[4] Y.C. Hsu, *MEBS VHDL Reference Manual*, University of California, Riverside, 1994.

[5] *IEEE Standard VHDL Language Reference Manual*, IEEE, 1989.

[6] F.J. Kurdahi, A.C. Parker, "REAL: A Program for Register Allocation," *Proc. of the DAC*, pp. 210-215, 1987.

[7] D. Lanneer, M. Cornero, G. Goossens, H.D. Man, "Data Routing : a Paradigm for Efficient Data-Path Synthesis and Code Generation," *7th ACM/IEEE HLSS.*, 1994.

[8] P.G. Paulin, J.P. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASICs," *IEEE Trans. on CAD/ICAS*, Vol. CAD-8, No. 6, pp. 661-679, June 1989.

[9] C.J. Tseng, D.P. Siewiorek, "Automated Synthesis of Data Paths in Digital Systems," *IEEE Trans. on CAD/ICAS*, Vol. CAD-5, No. 3, pp. 379-395, July 1986.

[10] *Verilog-XL Reference Manual*, CADENCE, Inc., version 1.6, Vol. 1, 1991.

[11] W. Wolf, A. Takach, C. Y. Huang and R. Manno, "The Princeton University Behavioral Synthesis System," *Proc. of the DAC*, pp. 182-187, 1992.