Storage Optimization by Replacing Some Flip-Flops with Latches[*]

Tsung-Yi Wu   Youn-Long Lin

Department of Computer Science
Tsing Hua University, Hsin-Chu, Taiwan 30043

Abstract

Conventionally, when a synchronous sequential circuit is synthesized, storage units are implemented in either edge-triggered flip-flops or level-sensitive latches, but not both, depending on the clocking scheme (one- or two-phase) used. We propose that, in the former case, some of the flip-flops can be replaced with latches. Since a latch is generally smaller, faster and less power-consuming than a flip-flop, this replacement leads to improvements in circuit area, performance and power consumption. Whether a flip-flop can be replaced with a latch depends on not only its structural context but also its temporal behavior. In this paper, we first present conditions under which a straightforward replacement can be made; then, we propose two retiming-based transformations that increase the number of replaceable flip-flops. We have implemented the proposed idea in a software called FF2Latch. Experimental results on a set of control-dominated circuits from the high-level synthesis benchmark set [1] show that a large number of the flip-flops can be replaced with latches. Up to 22% reduction in the circuit area and up to 73% reduction in the power consumption have been achieved.

1. Introduction

Research in high-level synthesis (HLS) has drawn a lot of attention because of its potential in significantly increasing the productivity of VLSI design [7][8][12]. Experience has shown that HLS tools have better chance to be accepted in such specific application domains as digital signal processing (DSP) circuits [15] and control-dominated circuits [6][18]. Control-dominated circuits are used in a wide variety of applications including protocol converters, network interfaces, ...etc. Therefore, automatic synthesis from a high-level behavioral description of control-dominated circuits is a worthy research. This work addresses storage synthesis, one of the important tasks in the HLS of control-dominated circuits.

The behavior of a control-dominated circuit is usually specified in a hardware description language (e.g., Verilog [3] or VHDL [2]) and compiled into an internal representation such as the behavioral finite state machine (BFSM) [18]. During each transition from one state of the machine to another, a set of data manipulation and movement operations take place. The synthesizer has to allocate for each variable or signal a storage element of appropriate word length. Depending on the clocking scheme used, contemporary synthesizers implement storage using either edge-triggered flip-flops (for one-phase clock) or level-sensitive latches (for two-phase clock), but not both. The former is larger but is easier to design; while the later is smaller and faster at the expense of greater design difficulty.

Papaefthymiou and Randall [14] compared the usage of flip-flops and latches in the context of retiming [11]. Their software, called TIM, replaces each flip-flop in a one-phase circuit with a pair of cascade, two-phase-clocked level-sensitive latches and subsequently retimes the resulting two-phase circuit. They have shown that, for performance-driven retiming, a latch-based circuit is more area-efficient than a flip-flop-based one. But, for area-driven retiming, there is no gain in using the latch-based approach. In this paper, however, we show that some of the flip-flops in a one-phase-clock system can be replaced by latches while preserving the circuit's functionality. Although the timing (temporal behavior) of some of the internal nodes will be modified, *the timing of all output ports remains unchanged.* Therefore, this type of replacement leads to area and power-consuming reduction in the synthesized circuit.

The rest of this paper is organized as follows. In the next section, we will describe the motivation behind this work. Then in section 3 we will first describe a straightforward replacement method, and then present two local transformations that increase the replacement opportunity. Some experimental results on a set of control-dominated circuits from the HLS benchmark set [1] are given in section 4. Finally, we draw a conclusion and point to possible directions for future research in section 5.

2. Motivation

To optimize the performance or the area of a circuit, a high-level synthesizer can modify the behavior of any part of the circuit so long as the circuit's function matches what is specified in the behavioral description. However, because of their emphasis on the so called *What You Synthesize is What You Simulate* (WYSWYS [5][17]) feature, contem-

[*]Supported in part by a grant from the National Science Council of R.O.C. under contract no. NSC-85-2221-E-007-032

porary HLS tools usually over-synthesize by making sure that every node of the synthesized circuit exhibits identical timing behavior as that of the corresponding node in the behavioral specification. If we relax this WYSWYS restriction on the circuit interior, the synthesizer will have more freedom in performing the optimization.

Compared with a flip-flop, a latch is smaller, faster and less power-consuming. Therefore, we should choose latches over flip-flops wherever possible. Unfortunately, in a synchronous sequential circuit, the choice between using flip-flops or latches is dependent on the underlining clocking scheme (one- or two-phased). We will be able to save more chip area if some of the flip-flops in a one-phase-clocked circuit can be replaced with latches.

When a synchronous Mealy machine[1] [9] is described in VHDL [2], the synchronization mechanism is usually expressed using a wait statement (e.g., "wait until (clock='1')" [4][5]). Upon the execution of the wait statement, the machine must wait for the system clock to raise. In other words, each wait statement in the behavioral description leads to a state in the corresponding synchronous Mealy machine. For instance, in the 2-state machine synthesized from the behavioral VHDL description for the PREFETCH benchmark circuit depicted in Figure 1, the two states are derived from the two wait statements (line 16 and line 25), respectively. Note that because actions are performed concurrently during state transitions, "<=" rather than ":=" [2] is used to represent each assignment action in a state-transition diagram (STD) that represents a synchronous Mealy machine.

Straightforwardly, every variable/signal in the left-hand side of any assignment statements following a wait statements will be given a set of flip-flops triggered by the same event waited for by the wait statement. For instance, in Figure 1(a), statement "ir := ibus" (line 28) following the second wait statement (line 25) causes the synthesizer to allocate a set of positive-edge-triggered flip-flops to variable *ir* (Figure 1(c)), and the value of *ir* will be updated to that of *ibus* at a positive edge of the system clock when the current state is s_1 while ire is true. This implementation style matches the timing semantics of every part of the specification.

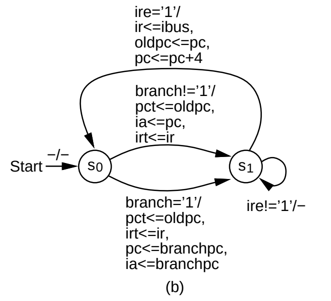
If we replace the flip-flops implementing variable *ir* with active-low latches (Figure 1(d)), some internal signals will exhibit different timing behavior as shown in Figure 2. We call the flip-flops ir_{ff} and the latches ir_{lt}. When the current state is s_1 and ire is true, ir_{ff} gets the value of *ibus* at a rising edge of the system clock (i.e., t_4). ir_{lt} will transport the value of *ibus* when the system clock is low (i.e., from t_1 to t_a and from t_3 to t_4).

In Figure 2, the positive edge of the system clock waited for by the second wait statement appears at both time t_2 and t_4 of the current execution cycle and t_2' of the previous execution cycle. Let $ibus_{t_x}$ be the value of *ibus* at time t_x.

[1]In this paper, we rigidly define a *synchronous Mealy machine* is such a FSM that state transitions occur at positive edges of the *system clock*, and actions are concurrently executed during state transitions.

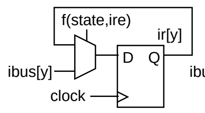
```
 1: use work.vvectors.all;
 2:
 3: entity prefetch is
 4:   port ( clock               : in bit;
 5:          branchpc, ibus : in bit32;
 6:          branch, ire         : in bit;
 7:          pct, irt, ia        : out bit32);
 8: end prefetch;
 9:
10: architecture behavior of prefetch is
11: begin
12:   process
13:     variable ir, oldpc, pc : bit32 := 0;
14:          --Initial values are ignored.
15:   begin
16:     wait until (clock = '1'); --state s0
17:     pct <= oldpc;
18:     ia <= pc;
19:     irt <= ir;
20:     if (branch = '1') then
21:       pc := branchpc;
22:       ia <= pc;
23:     end if;
24:     test_ire : loop
25:       wait until (clock = '1'); --state s1
26:       exit test_ire when (ire = '1');
27:     end loop;
28:     ir := ibus;
29:     oldpc := pc;
30:     pc := pc + 4;
31:   end process;
32:
33: end behavior;
```

(a)



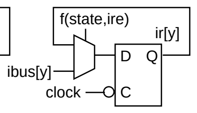
(b)

state	ire	output of MUX
s_0	–	ir[y]
s_1	0	ir[y]
s_1	1	ibus[y]



(c)

state	ire	output of MUX
s_0	–	ir[y]
s_1	0	ir[y]
s_1	1	ibus[y]



(d)

Figure 1. The PREFETCH benchmark: (a) VHDL description; (b) the STD that represents the synthesized synchronous Mealy machine; (c) the flip-flop implementation of the y-th bit of ir; (d) the latch implementation of the y-th bit of ir.

ir_{lt} latches $ibus_{t_a}$ during the time slot between t_a and t_3 because the state is s_1 and ire switches from logic 1 to 0 at t_a; however, ir_{ff} still holds $ibus_{t_2'}$ during this time slot because ire is logic 0 at t_2 when the machine transits from s_1 to the next state of s_1. Both ir_{ff} and ir_{lt} will be updated to $ibus_{t_4}$ at t_4, but, before t_4, only ir_{lt} is continuously following the value of *ibus* during the time interval between t_3 and t_4. Although ir_{lt} and ir_{ff} may hold different values during the time interval between t_1 and t_4, it will not affect the circuit's behavior at its I/O ports because no read access is made to variable ir during that interval. ir is read only at the transition from s_0 to s_1.

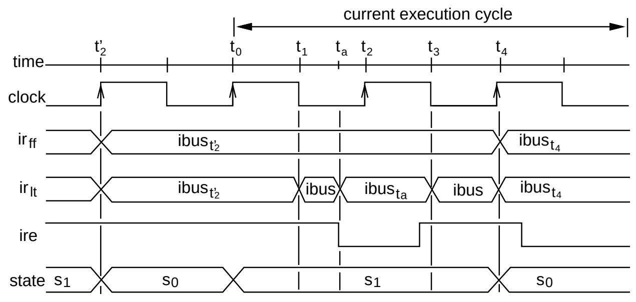


Figure 2. Timing diagram for ir_{ff} and ir_{lt} of the illustrative example.

From the above-described example, we know that a flip-flop can be replaced with latch if the replacement does not affect the I/O behavior. That is, a positive(negative)-edge flip-flop can be replaced by an active-low(-high) latch under certain conditions.

3. Replacement

The condition under which the flip-flops implementing a register can be replaced with latches depends on the relationship among the register, some I/O ports and some combinational blocks. Some replacements can be done directly; while others need some pre-processing. In this section, we first present a simple condition under which a flip-flop can be directly replaced with a latch. Then, we describe two local transformations that make more flip-flops replaceable.

3.1. Straightforward Replacement

In a synchronous Mealy machine, latches can be used to replace flip-flops implementing a register r if and only if the following conditions are met.

C_1: Writing to and reading from r never take place at the same time.

C_2: r does not feed any output port through only true combinational paths.

C_3: Let $p = [s_a, e_{a,b}, s_b, \ldots, s_i, e_{i,j}, s_j]$ be a path in the STD and f_1, f_2, f_3, f_4, and f_5 are arbitrary functions. If either action $x <= f_1(r, \ldots)$ or condition $f_2(r, \ldots)$ is associated with $e_{i,j}$, $r <= f_3(\ldots)$ is not in $p - e_{a,b}$, and there exists an edge $e_{a,m}$ ($\notin p$) conditioned by some asynchronous signals and associated with action $r <= f_4(\ldots)$, then there exists an action $r <= f_5(\ldots)$ associated with $e_{a,b}$.

C_1 is necessary because latches cannot implement a register that is read and written at the same time in a synchronous Mealy machine.

C_2 is necessary for the following reason. A register r is called r_{ff} if it is implemented by positive-edge flip-flops, and is called r_{lt} if it is implemented by active-low latches. If r feeds an output port through only combinational paths, the output port fed by r_{ff} and the output port fed by r_{lt} will exhibit different timing behavior. The reason is that an active-low latch will continuously update its value during the low level of the system clock, while a positive-edge flip-flop will update its value only at the positive-edge of the system clock. For instance, if ir shown in Figure 1(a) is declared as an output port, the output port implemented by ir_{ff} and the output port implemented by ir_{lt} will exhibit different timing behavior shown in Figure 2.

C_3 is necessary for the following reason. Let the path p shown in Figure 3 be part of a STD and the machine represented by the STD transit from state s_a to state s_b. If action $r <= f_5(\ldots)$ does not associate with $e_{a,b}$, then r_{lt} may latch the wrong value, $f_4(\ldots)$, during state s_a, s_b, ..., and s_i. The reason is that, before the transition from s_a to s_b, r_{lt} continuously updates its value to $f_4(\ldots)$ during state s_a when $cond_m$ is true and the system clock is 0. The wrong value may propagate to edge $e_{i,j}$ and be accessed by the edge's "condition/actions". On the other hand, if there exists an action $r <= f_5(\ldots)$ associated with $e_{a,b}$, the error cannot occur because the wrong value of r is overwritten by $f_5(\ldots)$.

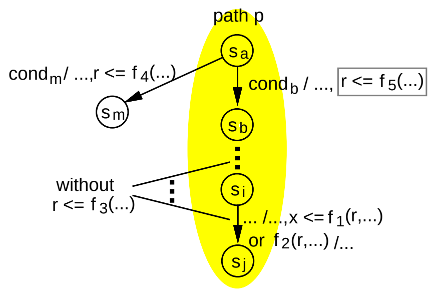

path p
s_a
$cond_m$/ ...,r <= f_4(...)
s_m
$cond_b$ / ..., r <= f_5(...)
s_b
without
r <= f_3(...)
s_i
... /...,x <= f_1(r,...)
or f_2(r,...) /...
s_j

Figure 3. An illustration for condition C_3: r can be implemented with latches only if "$r <= f_5(\ldots)$" exists.

A formal proof to the correctness of these conditions is given in [19].

A *concurrent register* is one that violates C_1. For instance, the register for pc in "pc <= pc+4" shown in Figure 1(b) is concurrent. Another example of concurrent register is one that appears in the right-hand side of an assignment statement and the left-hand side of another associated with the same edge. For instance, if both actions "a <= b" and "b <= a" are associated with the the same state transition to perform a swapping, then *a* and *b* are concurrent.

Simultaneous read and writing accesses by a single machine to a register can only take place during the same state transition. Therefore, to identify concurrent registers in a single machine, we only have to examine every state transition one at a time. A register is concurrent if and only if it appears in the left-hand side of any statement and in either the condition or the right-hand side of any statement associated with the same state transition.

A register satisfies condition C_2 if and only if the register is not declared an output port. To check whether a register satisfies condition C_3 we analyze all related paths in a synchronous Mealy machine.

A register implemented with flip-flops is replaced with latches if it satisfies conditions C_1, C_2, and C_3.

The replacement for register ir shown in Figure 1 is straightforward. Register ir is not concurrent because it is only read during the transition from state s_0 to s_1, and written during the transition from state s_1 to s_0 (Figure 1(b)). It is a variable rather than an output port according to the description given in Figure 1(a). Therefore, it satisfies C_2. It also satisfies C_3 by checking the diagram shown in Figure 1(b).

3.2. Replacement after Local Transformation

Some flip-flop that cannot be replaced with latches directly can be replaced after some local transformations are performed on its surrounding circuit. In this section, we present methods that make more flip-flops replaceable. We first describe how a flip-flop is generally implemented. Then, we describe a method that makes a non-replaceable flip-flop replaceable by moving it across some combinational portion of the circuit. Finally, we show that for some flip-flops their master latches can be retimed and then eliminated.

3.2.1. Storage Implementations

A flip-flop will load new data at some clock cycles and hold old data at the others. There are two methods to implement the loading/holding circuit. The first is to gate the clock by ANDing it with a control signal decoding from the state register. This produces a clock edge at the flip-flop only when it needs to load the data. For instance, Figure 4(a) shows a circuit for a flip-flop that will load data y at the rising edge of the system clock if signal $load$ is logic 1. The second method inserts a multiplexer in front of the flip-flop's data input. The output of the flip-flop will be fed back to the input in case of no loading of new data. For instance, in Figure 4(b) the flip-flop will hold its original data as long as signal $load$ is logic 0; on the other hand, it will load the new data, y, at the rising edge of the clock signal when signal $load$ is logic 1. The second method is preferred by contemporary synthesis tools because it does not suffer as much from the clock skew problem.

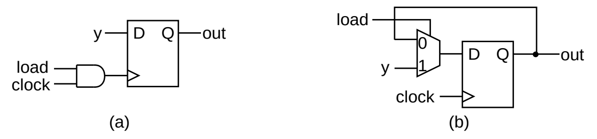


Figure 4. Methods of loading new data and holding old data at a flip-flop: (a) the gated clock method; (b) the multiplexed input method.

3.2.2. Transformation by Moving Flip-Flops

We use the circuit of Figure 5 to illustrate the idea. The circuit (CKT_{org}) shown in Figure 5(a) is synthesized by Synopsys's Design Compiler for register bit $r[m]$. $r[m]$ will retain its data during state s_p under condition c_p, but it will load new data $y[m]$ during state s_q under condition c_q. CKT_{org} can be retimed to CKT_{rt} (Figure 5(b)) by moving the flip-flop across the multiplexer. *Note that the control signals for the multiplexer have effectively also been retimed by decoding the next state function.*

The retiming is feasible as long as there is no contradiction in the resulting control table. That is, no more than one input is to be routed to the multiplexer's output simultaneously. If the retiming is infeasible, we will apply the technique described in Section 3.2.3.

Every flip-flop of CKT_{rt} other than FF_p can be removed if its input data remains unchanged for at least one clock period before its output data is routed to the multiplexer's output ($r[m]$). For instance, FF_q can be removed if data $y[m]$ is synchronous and is stable from state s_q to state $next(s_q,\ c_q)$. If all flip-flops of CKT_{rt} except FF_p can be removed, CKT_{rt} can be transformed into CKT_{trf} (Figure 5(c)). Flip-flip FF_p of circuit CKT_{trf} will be concurrently read and written only during state $next(s_p,\ c_p)$. FF_p can be replaced with a latch because what is read and what is written are the same. Figure 5(d) shows the optimized circuit by replacing FF_p of CKT_{trf} with a latch. Clearly, the circuit of Figure 5(d) is less expensive than that of Figure 5(a).

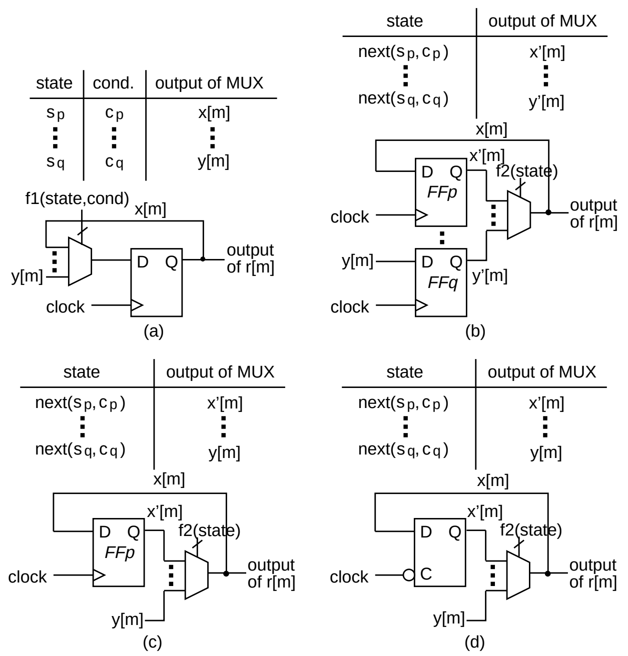


Figure 5. The implementation of register bit $r[m]$: (a) the original implementation, CKT_{org}; (b) the retimed implementation, CKT_{rt}; (c) the transformed implementation, CKT_{trf}; (d) the optimized implementation, CKT_{opt}.

Our algorithms finds any register that matches the pattern of CKT_{org} and transforms it to that of CKT_{opt}. Figure 6 shows the process when the technique is applied to the y-th bit of register irt of the PREFETCH benchmark.

3.2.3. Transformation by Moving Master Latches

When a flip-flop cannot be moved as described previously, we still can do some transformations. We use Figure 7 to illustrate the idea. CKT_{org} shown in Figure 7(a) can be retimed to CKT_{rt} in Figure 7(b) by moving the master latch of the flip-flop FF_p across the multiplexer. Note that the control signals are retimed.

We now show that ML_p can be removed from CKT_{rt} without changing the circuit function. ML_p acts as a short circuit when the clock signal is low. When the clock signal is high, ML_p is useful only when slave latch SL_p is latching the output value of ML_p. However, the content of ML_p is same as that of SL_p. Therefore ML_p is also useless during the clock signal being logic 1. We conclude that ML_p can be removed from CKT_{rt} without changing the circuit function as shown in Figure 7(c).

In CKT_{trf}, any master latch feeding the multiplexer can be removed if its input data is stable while its output data is being routed to the multiplexer's output. For instance, latch ML_q in CKT_{trf} can be removed if signal $y[m]$ is synchronous and is stable during the transition from state s_q to state $next(s_q, c_q)$.

Figure 7(e) shows the possible timing difference between signal $y[m]$ and signal $y'[m]$ in CKT_{trf} during the transi-

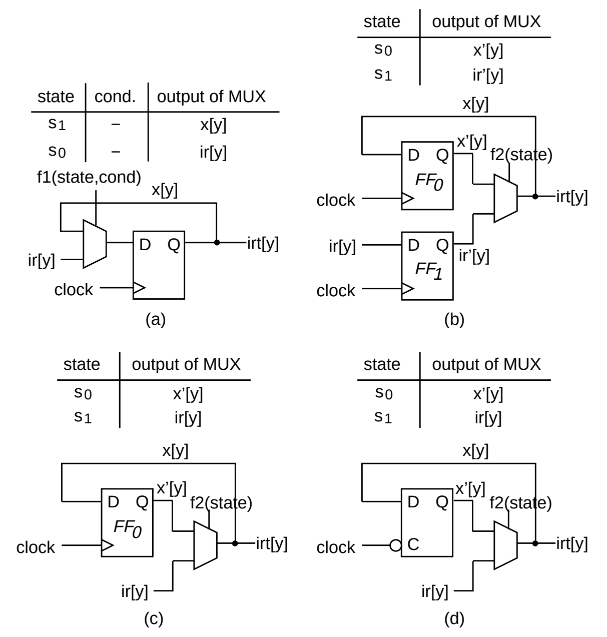


Figure 6. The implementation of register bit $irt[y]$ of PREFETCH: (a) the original implementation, CKT'_{org}; (b) the retimed implementation, CKT'_{rt}; (c) the transformed implementation, CKT'_{trf}; (d) the optimized implementation, CKT'_{opt}.

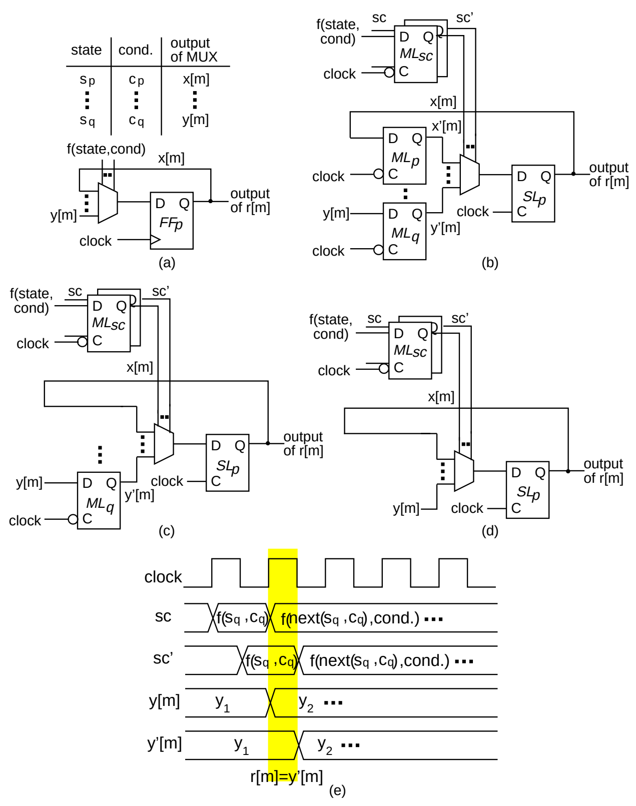


Figure 7. The implementation of register bit $r[m]$: (a) the original implementation, CKT_{org}; (b) the retimed implementation, CKT_{rt}; (c) the transformed implementation, CKT_{trf}; (d) the optimized implementation, CKT_{opt}; (e) the timing diagram for CKT_{trf}.

tion from state s_q to state $next(s_q, c_q)$. Register bit $r[m]$ will latch value $y'[m]$ when the clock signal is logic one and the state is $next(s_q, c_q)$, i.e., during the *gray interval*. Therefore, $r[m]$ will latch value y_1 during state $next(s_q, c_q)$. Obviously, the behavior matches that of CKT_{org}. However, $r[m]$ will latch value y_2 instead of value y_1 if ML_q is removed. Removing ML_q does not change the function of CKT_{trf} if value y_1 is equal to y_2 during the gray interval. In other words, latch ML_q in CKT_{trf} is redundant if signal $y[m]$ does not change during the gray interval. It is easy to know whether $y[m]$ switches during the gray interval by inspecting the STD. Figure 7(d) shows the circuit, CKT_{opt}, derived from CKT_{trf} by removing all latches that directly feed the multiplexer.

CKT_{opt} use a latch rather than a flip-flop as in CKT_{org} to implement $r[m]$. ML_{sc} is the control overhead. Despite the overhead, CKT_{opt} is generally less expensive because ML_{sc} are sharable among multiple bits.

Our goal is to find the register that matches the pattern of CKT_{org} and transforms it to that of CKT_{opt} if this transformation can reduce area cost. For instance, irt shown in Figure 6 can also be transformed with this method as shown in Figure 8. The transformed circuit uses 32 fewer flip-flops and 33 more latches.

4. Experimental Results

We have implemented the proposed idea in a software program called *FF2Latch*. Several circuits have been used to test the effectiveness of *FF2Latch*. They include a counter (COUNTER [1]), a greatest common divisors (GCD_ALT [1]), a prefetch subcircuit (PREFETCH [1]), the Fancy chip (FANCY [1]), a highway traffic light controller (HIGHWAY [1]), a bar-code reader circuit (BARCODE), a first-in first-out stack (FIFO [13]), and a transceiver (TRANSCEIVER).

Each benchmark is firstly synthesized using the Synopsys Design Compiler. Then, *FF2Latch* identifies the set of flip-flops that can be replaced with latches. Each circuit takes *FF2Latch* and Design Compiler a few minutes to execute on a SUN SPARC-20.

The experimental results are summarized in Table 1. Columns 2, and 3 show the total register bits and the number of flip-flops that have been identified by *FF2Latch* as replaceable, respectively. Column 4 shows the total cell area of the synthesized result by Synopsys Design Compiler. Column 5 shows the total cell area of the synthesized result with the postprocessing of *FF2Latch*. Column 6 shows the percentage of reduction in the total circuit area under the assumption that a flip-flop is twice as large as a latch as specified by the Synopsys's class library.

Column 7 shows the power (microWatts) consumed by

	Register		Circuit area			Power consumption (uW)		
Benchmark	#bits	#repl.	Synopsys	Synopsys + FF2Latch	reduc.	Synopsys	Synopsys + FF2Latch	reduc.
COUNTER	8	0	120	120	0	686	686	0
GCD_ALT	25	9	543	514	5.3%	3377	3047	9.8%
PREFETCH	192	128	2241	1800	19.6%	15687	87942	43.9%
FANCY	50	26	1261	1168	7.4%	–	–	–
HIGHWAY	9	6	159	137	13.8%	1014	765	24.6%
BARCODE	43	1	618	615	0.5%	4227	4188	0.9%
FIFO	151	128	1988	1540	22.5%	12085	3201	73.5%
TRANSCEIVER	24	16	256	200	21.9%	1906	1014	46.7%

Table 1. Experimental results.

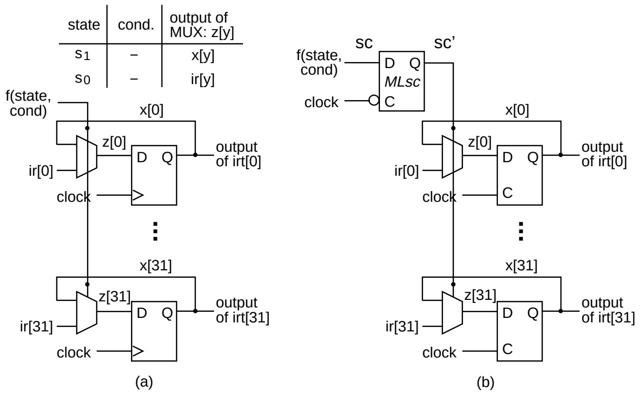


Figure 8. The implementation of register irt: (a) the original implementation; (b) the transformed implementation.

each synthesized circuit. Column 8 shows the power consumption after replacement. The last column shows the percentage of reduction in the power consumption. Power consumption was estimated using SIS-1.3 [16] with command "power_estimate –t S –s U". Note that SIS-1.3 failed to report on the power consumption of FANCY because of insufficient memory in our workstation.

5. Conclusions

We have proposed a method to replace some of the flip-flops with latches in a one-phase-clocked synchronous sequential circuit. Latches are in general smaller, faster and less power-consuming than flip-flops. We have implemented the proposed approach in a software called *FF2Latch*. Experiment on a set of control-dominated benchmark circuits show that up to 22% of the circuit area and 73% of the power consumption can be saved by applying *FF2Latch*.

A possible direction for future research is to study how to reschedule actions in the synchronous Mealy machine so that the number of registers with concurrent R/W is reduced; hence, the number of registers that can be implemented in latches is increased.

References

[1] "Benchmarks for the 6th International Workshop on High-Level Synthesis," Available through anonymous ftp at ics.uci.edu, 1992.

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

[3] *Verilog-XL Reference Manual*, CADENCE Design Systems, Inc., version 1.6, vol. 1, 1991.

[4] R. A. Bergamaschi and A. Kuehlmann, "A System for Production Use of High-Level Synthesis," *IEEE Transactions on VLSI Systems*, pp. 233-243, Sep. 1993.

[5] J. Biesenack, et al., "The Siemens High-Level Synthesis System CALLAS," *IEEE Transactions on VLSI Systems*, pp. 244-252, Sep. 1993.

[6] R. Camposano, "Path-Based Scheduling for Synthesis," *IEEE Transactions on CAD of Integrated Circuits and Systems*, pp. 85-93, Jan. 1991.

[7] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.

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

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

[10] Z. Kohavi, *Switching and Finite Automata Theory*, McGraw-Hill, second edition, 1978.

[11] C. Leiserson, F. Rose, and J. Saxe, "Optimizing Synchronous Circuits by Retiming," *Third Caltech Conference on VLSI*, pp. 87-116, 1983.

[12] M. C. McFarland, A. C. Parker, and R. Camposano, "Tutorial on High-Level Synthesis," *25th ACM/IEEE Design Automation Conference*, pp. 330-336, 1988.

[13] D. E. Ott and T. J. Wilderotter, *A Designer's Guide to VHDL Synthesis*, Kluwer Academic Publishers, pp. 215-225, 1994.

[14] M. C. Papaefthymiou and K. H. Randall, "Edge-Triggering vs. Two-Phase Level-Clocking," *Research on Integrated Systems: Proceedings of the 1993 Symposium*, Mar. 1993.

[15] J. Rabaey, H. D. Man, J. Vanhoof, G. Goossens, and F. Catthoor, "CATHEDRAL II: A synthesis system for multiprocessor DSP," in *Silicon Compilation* (D. Gajski, editor), pp. 311-360, Addison-Wesley, 1988.

[16] E. M. Sentovich, et al., *SIS: A System for Sequential Circuit Synthesis*, Department of Electrical Engineering and Computer Science, UC Berkeley, May 1992.

[17] A. Stoll and P. Duzy, "High-level Synthesis with Exact Timing Constraints," *29th ACM/IEEE Design Automation Conference*, pp. 188-193, 1992.

[18] W. Wolf, A. Takach, C. Y. Huang, and R. Manno, "The Princeton University Behavioral Synthesis System," *29th ACM/IEEE Design Automation Conference*, pp. 182-187, 1992.

[19] T.-Y. Wu and Y.-L. Lin, "Storage Optimization by Replacing Some Flip-Flops with Latches," *Technique Report*, Tsing Hua University, 1995.