HLS-l: A High-Level Synthesis Framework for Latch-Based Architectures

Seungwhun Paik, *Student Member, IEEE,* Insup Shin, *Student Member, IEEE,* Taewhan Kim, *Senior Member, IEEE,* and Youngsoo Shin, *Senior Member, IEEE*

***Abstract*—Level-sensitive latches are widely used in high-performance custom designs while edge-triggered flip-flops are predominantly used in application-specific integrated circuits. We consider a latch as a basis for storage and address each step of high-level synthesis (HLS), including scheduling, allocation, and control synthesis. While the use of latches provides an opportunity to reduce the latency during the scheduling, the register allocation has to take extra conflicts caused by latch into account, and the control synthesis has to be tailored to support the latch-based data-path. Optimization potentials specific to this HLS are identified and solutions are proposed. Specifically, the register allocation can be improved by refining the operation schedule in a way to reduce the number of edges in a register conflict graph; the latency can be reduced by adjusting the clock duty cycle in a way to generate a tighter schedule. All the steps of HLS and optimization procedures were integrated into a framework called HLS-l. It was tested on benchmark designs implemented in 1.1-V, 45 nm complementary metal-oxide-semiconductor technology. Compared to the conventional HLS, HLS-l was able to reduce the latency by 18.2% on average with 9.2% less area and 16.0% less power consumption. The application of HLS-l to an industrial example is demonstrated through the design of a module extracted from H.264/advanced video coding.**

***Index Terms*—ASIC, dual-edge-triggered flip-flop, high performance, high-level synthesis, latch.**

I. Introduction

IT IS WELL-KNOWN that there is a large performance gap between custom designs and application-specific integrated circuit (ASIC) designs. Specifically, ASICs are routinely slower than their custom counterparts in the same technology node by a factor of six or more [2]. Several factors have been identified [2] which cause this large performance gap: only a limited flavor of microarchitectures is used in ASICs, the timing overhead due to clock tree and registers can be alleviated in custom designs but not in typical ASICs, static complementary metal-oxide-semiconductor (CMOS) is dominantly used in ASICs but high-speed dynamic logic is also used in custom designs, and so on. Higher performance has been mainly achieved by the technology scaling, but as the CMOS scaling becomes very difficult and uneconomical around the 22 nm node [3], it is important to extract higher performance in ASIC designs by employing methodologies used in the custom counterpart.

The timing overhead that stems from the clock tree and registers is an important factor to address for high-performance ASIC designs. ASICs typically have about 4 fanout-of-4 (FO4) delays of clock skew and jitter, which can be cut down to 1 FO4 in custom designs [2]. Employing a low-skew clock distribution network, e.g., buffered clock trees driving a clock grid [4], or combined clock skew scheduling and clock tree construction [5], may alleviate the timing overhead due to clock tree.

Edge-triggered flip-flops are predominantly used in ASICs as registers, which have three or four FO4 delays; high-speed custom designs typically use level-sensitive latches, which have two FO4 delays [6]. In the flip-flop-based design, each combinational block between flip-flops can be isolated in view of timing, making timing analysis and optimization very convenient for synthesis-based ASICs. This is not the case in the latch-based design, because some combinational blocks may use more than the clock period to compute, which has to be compensated for by some other blocks that use less than the clock period. The transparency of latch, however, offers flexibility in handling timing, which can be used to tolerate clock skews or to distribute timing slacks to combinational blocks for higher frequency.

Latch-based architectures have been used extensively in custom designs [7]–[11] and even in some ASICs [12], [13]. Only a few papers, however, have been published on the use of latches during high-level synthesis (HLS), which we review in Section II; even their use is limited to the register allocation. In this paper, we consider latches during HLS more radically, i.e., they are taken into account during the whole step of HLS. The complicated timing behavior of latches is made manageable through the proposed operation scheduling, register allocation, and control synthesis; the key idea commonly carried in these HLS steps is to prevent latches from being read and written at the same time while latches are transparent.

Manuscript received July 3, 2009; revised November 21, 2009. Current version published April 21, 2010. This work was supported by the Korea Science and Engineering Foundation (KOSEF) Grant, funded by the Ministry of Education, Science and Technology (MEST), no. R01-2007-000-20891-0. The work of T. Kim was supported by the Basic Science Research Program of MEST (no. 2009-0091236). This paper [1] was presented in part at the Design, Automation and Test in Europe Conference, Nice, France, April 20–24, 2009. This paper was recommended by Associate Editor, S. Nowick.

S. Paik, I. Shin, and Y. Shin are with the Department of Electrical Engineering, Korea Advanced Institute of Science and Technology (KAIST), Daejeon 305-701, Korea (e-mail: swpaik@dtlab.kaist.ac.kr; isshin@dtlab.kaist.ac.kr; youngsoo@ee.kaist.ac.kr).

T. Kim is with the School of Electrical Engineering and Computer Science, Seoul National University (SNU), Seoul 151-742, Korea (e-mail: tkim@ssl.snu.ac.kr).

Digital Object Identifier 10.1109/TCAD.2010.2043588

We introduce a concept of phase step (p-step), which refers to a clock phase, i.e., a period of the clock being high or low. We show that using p-step, as opposed to using the conventional clock step, as a time unit of the operation scheduling provides flexibility that helps reduce the latency. Care needs to be taken, however, in scheduling an operation having the same variable as its input and output because such a variable has a possibility of being read and written simultaneously; this leads us to define the feasibility of a schedule. List scheduling is taken as an example to extend it toward the p-step based scheduling (Section III-A).

Due to the transparent nature of latch, extra edges are introduced in a register conflict graph so that the same latch should not be read and written at the same time. We show that some of these edges can be removed, which helps reduce the number of registers, by re-scheduling some of operations (Section III-B).

The p-step-based scheduling forces to generate control signals at every clock phase. This is realized by using dual-edge-triggered flip-flops (Section III-C).

The duty cycle, the proportion of the clock being high, affects the p-step based scheduling, and thus it affects the latency. The method to determine the duty cycle that leads to a schedule of the minimum latency is proposed (Section IV). All the aforementioned methods were integrated into HLS-l, and the design flow based on it was developed starting from the behavioral description in VHDL down to the synthesized netlist. Its application was demonstrated on behavioral benchmark designs as well as an industrial example of H.264/advanced video coding (AVC) in 45 nm technology (Section V).

Our main contributions are summarized as follows.

1) A concept of p-step, which provides the flexibility in the operation scheduling that helps reduce the latency, and its application to list scheduling (Section III-A).
2) The formulation of the register allocation for latch-based registers, and refining schedule to improve the register allocation (Section III-B).
3) Controller synthesis to support the p-step based schedule by using dual-edge-triggered flip-flops (Section III-C).
4) Optimizing the clock duty cycle for a better operation schedule (Section IV).
5) Extensive experimental results from commercial 45 nm technology applied to behavioral benchmark designs to assess HLS-l in terms of the latency, area, and power consumption (Section V), as well as application of HLS-l to an industrial example of H.264/AVC (Section VI-B).

The remainder of this paper is organized as follows. In the next section, we briefly review related work. In Section III, we address each step of HLS-l, namely the operation scheduling, allocation, and control synthesis. The problem of optimizing the clock duty cycle is presented in Section IV. Experimental results are presented in Section V; the application of HLS-l to an industrial example, as well as the design flow using HLS-l, is discussed in Section VI. We draw a conclusion in Section VII.

II. Related Works

HLS-l is related to HLS using high-performance microarchitectures and, in particular, to latch-based HLS, which we review in this section.

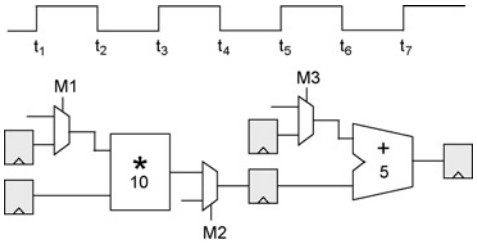

Fig. 1. Example data-path.

A. High-Performance Microarchitectures for HLS

There have been several research efforts to encompass various microarchitectures during HLS. Optimal selection of the clock period [14], [15] has been studied to minimize a waste of timing slack. Chaining and allowing multicycle operations are popular techniques to make operation schedule tighter, which have been extended to several directions [16]–[18]. A concept of complex functional unit (FU) has been introduced [17] to enforce operation chaining; this approach, however, limits potential FU sharings and requires a library of large set of complex FUs to cover various chaining combinations. Multicycle execution of chained operations has been combined with bit-level chaining [18], which exploits bit-level parallelism of FUs based on bit-by-bit computation, e.g., in a ripple carry adder; it, however, comes at a cost of complex controller.

B. Latch-Based Architectures for HLS

The methods of replacing flip-flops with latches have been proposed in [19], [20]. During the register allocation, they convert as many flip-flops as possible to latches, while preserving the original functionality, so that power consumption and area can be reduced.

More recently, the latch replacement has been addressed to improve timing yield [21]. Because latches are inherently more robust to process variations than flip-flops are, the latch replacement can improve the timing yield of designs.

However, these works consider latches only during the register allocation after performing the conventional scheduling, and thus do not take full advantage of latches during all the steps of HLS, which we target in HLS-l.

III. High-Level Synthesis of Latch-Based Architectures

A. Operation Scheduling Based on Phase Step

1) *Phase Step:* In the conventional HLS [22] using registers as storage, where registers are built from flip-flops, the operation scheduling is performed in a unit of clock step, also called a control step (c-step). Therefore, the execution delay d_i of operation i is given as the number of c-steps it takes

$$d_i = \left\lceil \frac{D_i}{T} \right\rceil \quad D_i = D_{\mathrm{FU}(i)} + T_{\mathrm{cq}} + \alpha \cdot T_{\mathrm{mux}} + T_{\mathrm{su}} \tag{1}$$

where T is the clock period, $D_{\mathrm{FU}(i)}$ is the maximum delay of a FU that executes i, T_{cq} and T_{su} are clock-to-Q delay and

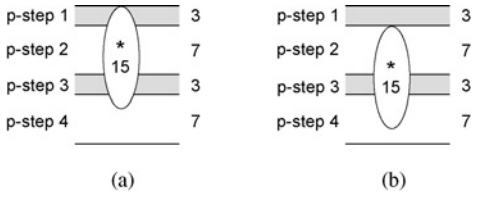

Fig. 2. Execution delay of a multiplication using p-steps: (a) delay is four p-steps when scheduled in the first p-step, (b) three p-steps when scheduled in the second p-step.

setup time of a register, respectively, and T_{mux} is the delay of a multiplexer. The number of multiplexers, α, on the timing path is determined after resource allocation on all data-path components. The controller delay can also be factored in (1) through estimating its delay [23]; or, it can be assumed to be negligible if all control signals are ready before data arrive at the input of multiplexers, which can be done via adjusting clock arrival times to the controller. Fig. 1 shows an example data-path. Let $T = 10$ time units, $T_{\text{cq}} = T_{\text{su}} = T_{\text{mux}} = 1$ time unit, and the numbers within FUs denote their delays in time units; the multiplication is scheduled in the first two c-steps because $\lceil (10 + 1 + 2 \cdot 1 + 1)/10 \rceil = 2$, which span the time from t_1 to t_5; the addition, whose execution delay is 1 because $\lceil (5 + 1 + 1 + 1)/10 \rceil = 1$, occupies the next c-step, which is from t_5 to t_7.

If we use positive level-sensitive latches for the registers, the multiplication can be initiated at any time between t_1 and t_2. Since the data-path has to be synchronized with the controller, e.g., the multiplexer select `M1` has to be made available for the register data to be steered to the multiplier, scheduling operations at arbitrary time point may yield very complicated controller. If we restrict operations to being scheduled only at the clock edges, the multiplication, which needs 14 time units for its execution, is now scheduled between t_1 and t_4 and the addition, which needs eight time units, is scheduled between t_4 and t_6 provided that the duty cycle is 0.5. This saves the amount of time corresponding to the period between t_6 and t_7 compared to the schedule using flip-flops for registers. Note that `M3` is generated after the falling-edge of the clock (t_4) while `M1` is generated after the rising-edge of the clock (t_1), i.e., the controller needs to generate control signals at both the clock edges, which can be implemented by using dual-edge-triggered flip-flops as we discuss in Section III-C. A similar schedule can be derived if we stick to flip-flops for registers but with twice the frequency, i.e., $T = 5$ time units; this approach, however, comes at a cost of roughly twice the power consumption of the clock network, more sequencing overhead, and the lack of optimization potential that comes from adjusting the duty cycle, which will be addressed in Section IV.

As a unit of the operation scheduling, we introduce a concept of *phase step* (p-step). In positive level-sensitive latches, p-step refers to either a transparent phase, which is the period of the clock being high, or a non-transparent phase, which is the period of the clock being low; vice versa in negative level-sensitive latches. A single c-step, therefore, is equivalent to two consecutive p-steps. The execution delay d_i is now given as the number of p-steps operation i takes. Let r_i be a residual delay, i.e., the remainder after D_i is subtracted by integer multiples of T, $r_i = D_i \bmod T$, and $\varphi(i)$ be an index of p-step when i is initiated. Then

$$d_i = \begin{cases} 2\left\lfloor \frac{D_i}{T} \right\rfloor + \left\lceil \frac{r_i}{T} \right\rceil \left(\left\lceil \frac{r_i - W}{T} \right\rceil + 1 \right) & \text{if } \varphi(i) \text{ is odd} \\ 2\left\lfloor \frac{D_i}{T} \right\rfloor + \left\lceil \frac{r_i}{T} \right\rceil \left(\left\lceil \frac{r_i - (T - W)}{T} \right\rceil + 1 \right) & \text{otherwise} \end{cases} \tag{2}$$

where W is a width of transparent phase with $T - W$ being a width of non-transparent phase. In (2), $\lceil r_i/T \rceil$ becomes 0 when $r_i = 0$ and 1 otherwise, since $r_i < T$; the expression $\lceil (r_i - W)/T \rceil + 1$ evaluates to 1 when $r_i < W$ and 2 otherwise, and similarly for $\lceil [r_i - (T - W)]/T \rceil + 1$, i.e., it evaluates to 1 when $r_i < T - W$ and 2 otherwise. Note that if the duty cycle W/T is 0.5, d_i is uniquely determined, otherwise d_i takes different values depending on whether i is scheduled in a transparent or in a non-transparent phase.

Example 1: Consider the multiplication in Fig. 2. Suppose that T is 10 time units, W is 3 time units, and D_i is 15 time units. The residual delay r_i is 5 time units. When the multiplication is scheduled in the first p-step, which is transparent, d_i is four p-steps as computed by (2). However, if it is scheduled in the second p-step as shown in Fig. 2(b), d_i becomes three p-steps. □

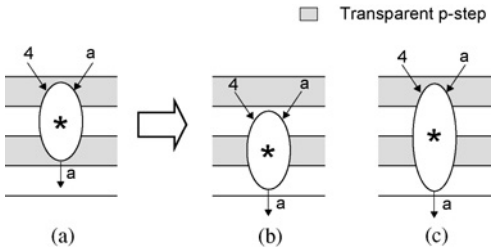

Fig. 3. (a) Infeasible schedule when CRWO completes in the transparent p-step; making it feasible (b) by scheduling it in the next p-step or (c) by extending its execution delay.

2) *Feasibility of Schedule and As-Soon-As-Possible (ASAP)/As-Late-As-Possible (ALAP) Scheduling:* In the p-step-based scheduling, an operation with one of its input operands being the same as its output operand, called a concurrent read/write operation (CRWO), is not allowed to complete its execution in a transparent p-step. Fig. 3(a) shows an example; the output operand a is written to a register in the third p-step, which is transparent; since a is also the input operand, it triggers a new multiplication; if the output from the multiplier (due to its min-delay) starts to appear within the third p-step, the register is contaminated with wrong value.

This can be avoided if we schedule a CRWO such that it completes in a non-transparent p-step as shown in Fig. 3(b); a is not loaded into the register in the fourth p-step since it is non-transparent. Another option to resolve the case of Fig. 3(a) is to arbitrarily extend the execution delay by one p-step as shown in Fig. 3(c) so that the operation can complete in a non-transparent p-step. Note that the solution in Fig. 3(b) is not always possible to achieve while the solution in Fig. 3(c) is. For example, if $W = 3$ time units, $T = 5$ time units, and $D_i = 8$ time units, an operation always completes in a transparent p-step, thus increasing d_i is the only option.

Definition 1: An operation schedule is called feasible if and only if all CRWOs complete their execution in non-transparent p-steps.

Example 1 and (2) show that d_i can vary in the p-step-based scheduling; thus, it raises a question of whether we need to take the varying operation delay into account while we perform the operation scheduling. Specifically, the multiplication completes its execution in the fourth p-step both in Fig. 2(a) and (b), even though it is initiated in the first p-step in the former and in the second in the latter. Since the first p-step is empty in Fig. 2(b), which may be used for scheduling other operations, the question therefore is whether Fig. 2(b) always yields better schedule than Fig. 2(a) does in terms of the latency (in resource-constrained scheduling) or in terms of the total number of FUs (in latency-constrained scheduling).

Fortunately, we do not need to worry about the varying operation delay for ASAP scheduling [24], which is usually used as a basis of other scheduling algorithms such as list [26] and force-directed scheduling [27]:

Proposition 1: In ASAP scheduling, scheduling an operation in the p-step l is not worse in terms of the latency than scheduling it in the p-step $l+1$, provided that the scheduling is feasible.

The claim is true to ALAP scheduling [24].

Proposition 2: In ALAP scheduling, for a given latency constraint, if scheduling an operation in the p-step l violates the latency constraint, then scheduling it in the p-step $k < l$ also violates the latency constraint, provided that the scheduling is feasible.

3) *List Scheduling With P-Step:* We address how the conventional list scheduling [26] can be extended toward the p-step-based scheduling; other scheduling algorithms such as force-directed scheduling [27] and ILP-based scheduling [16] can be handled similarly. *Resource-constrained-list* scheduling is shown in Fig. 4, where a_k denotes the maximum number of FUs of type k that are available. The candidate operations $U_{l,k}$ are the operations that are ready to be scheduled in the current p-step l (i.e., their predecessors are all scheduled and completed their execution before l) and can be executed by FU of type k (L4). The unfinished operations $T_{l,k}$ are the operations that are scheduled before l but do not complete their execution in l, and thus occupy FUs of type k (L5). Due to the resource constraint, a subset of candidate operations S_k is selected (L6); this is done by a priority assigned to each operation [24]. If S_k contains a CRWO (L7), we check whether scheduling it in l yields its completion in a transparent p-step (L8), which corresponds to the case of Fig. 3(a). We then remove it from S_k (L9) so that it can be scheduled in the next p-step [see Fig. 3(b)]; if it always completes in a transparent p-step, which corresponds to its execution delay when scheduled in l [denoted by $d_i(l)$] being different from that when scheduled in $l+1$ [denoted by $d_i(l+1)$], we simply increase its execution delay by one p-step (L10).

When we schedule each operation $i \in S_k$ (L11), if $d_i(l) = d_i(l+1)+1$ (see Fig. 2), we have an option to schedule it in l or in $l+1$. If we choose $l+1$, the current p-step is not filled in with i; thus, we should select other operation j, which is not in S_k but in $U_{l,k}$, having a lower priority than i to be scheduled in l. This, however, may forbid i to be scheduled in $l+1$ since j already occupies one of FUs; this in turn may increase the latency rather than decrease since operations of a higher priority are scheduled in the later p-steps. Thus, all the operations in S_k are scheduled in the current p-step l (L11).

Algorithm *Resource-Constrained-List* $(a_1, a_2, \ldots, a_n)$

```
L1   Initialize current p-step, l ← 1
L2   while there are un-scheduled operations do
L3     for each resource type k do
L4       Determine candidate operations U_{l,k}
L5       Determine unfinished operations T_{l,k}
L6       Select S_k ⊂ U_{l,k} such that |S_k| + |T_{l,k}| ≤ a_k
L7       for each CRWO i ∈ S_k do
L8         if l + d_i(l) − 1 is transparent p-step then
L9           if d_i(l) = d_i(l+1) then S_k ← S_k − {i}
L10          else d_i(l) ← d_i(l) + 1
L11      Schedule operations in S_k in l
L12    l ← l + 1
```

(a)

Algorithm *Latency-Constrained-List* (λ)

```
L13  Initialize current p-step, l ← 1
L14  Initialize FU number, a_k ← 1, i = 1, 2, ..., n
L15  Compute the latest possible start p-step t_i^L of all operations
     via ALAP(λ)
L16  while there are un-scheduled operations do
L17    for each resource type k do
L18      Determine candidate operations U_{l,k}
L19      Compute slacks s_i ← t_i^L − l, ∀i ∈ U_{l,k}
L20      Schedule i with s_i = 0 and update a_k
L21      Schedule i that requires no additional resources
L22    l ← l + 1
```

(b)

Fig. 4. Pseudo-code of (a) resource-constrained and (b) latency-constrained list scheduling algorithms.

Example 2: Consider a data-flow graph (DFG) shown in Fig. 5(a). Suppose that T is ten time units, W is seven time units, and the execution delay of multiplier, adder, and subtractor are 16 time units, 12 time units, and ten time units, respectively. Each type of FU can be used just once due to the resource constraint. The result produced by *resource-constrained-list* is shown in Fig. 5(b). Operation 1 is scheduled in the second, not in the first, p-step because it is a CRWO. Operation 5 is also a CRWO; it is scheduled in the first p-step with its execution delay extended because otherwise it always completes in a transparent p-step [recall Fig. 3(c)]. Since operation 1 is not allowed in the first p-step, we may consider operation 4 instead, even though 4 has a lower priority. This, however, increases the latency as shown in Fig. 5(c), because operation 1, which is more important in the latency, has to be delayed. □

Latency-constrained list scheduling *latency-constrained-list* is shown in Fig. 4, where λ is the latency constraint as the number of p-steps; overall, the algorithm is very similar to the conventional c-step-based list scheduling. Note that if $d_i(l) = d_i(l+1)+1$ for some operation $i \in U_{l,k}$, which corresponds to

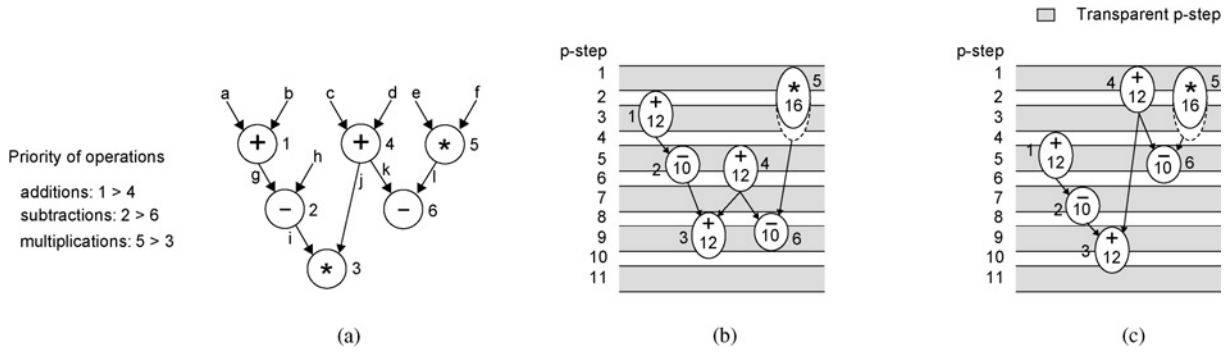

Fig. 5. (a) Example DFG. (b) Result produced by *resource-constrained-list*. (c) Schedule when operation 4 is scheduled before operation 1.

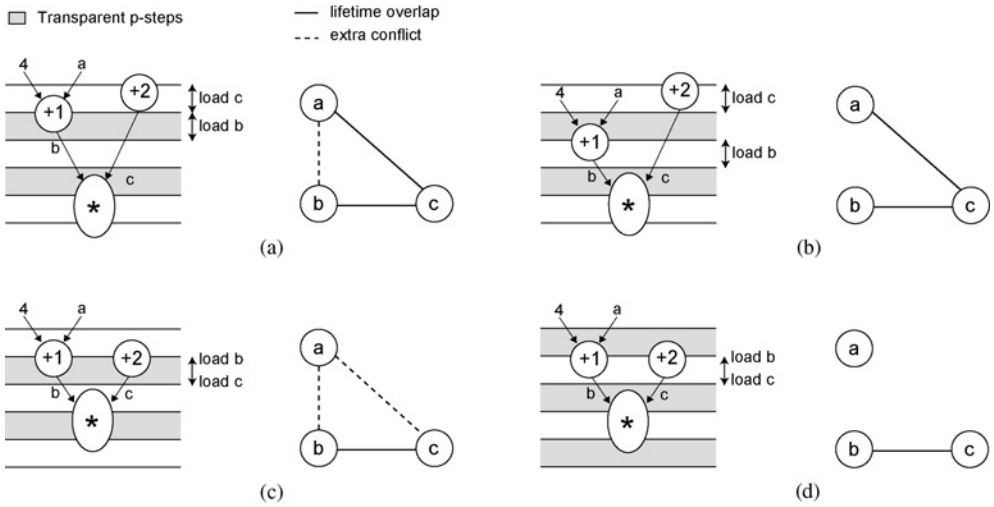

Fig. 6. (a) Variables a and b cannot share a register. (b) a and b can share a register when the adder completes its execution in a non-transparent p-step. (c) a and c (as well as a and b) cannot share a register. (d) a and c (as well as a and b) can share a register when two adders complete their execution in a non-transparent p-step.

Fig. 2, and if the latest possible start p-step of i determined by ALAP is $l+1$ (L15), it is scheduled in l only if there are resources available to use (L21), otherwise it is scheduled in l+1.

B. Allocation

1) *Register Allocation:* The register allocation for the conventional flip-flop-based registers is formulated as the vertex coloring problem on a register conflict graph G_R. Each vertex in G_R corresponds to the lifetime of a variable, and there is an edge between two vertices if their lifetimes overlap. If each of all lifetimes is a continuous interval, G_R belongs to an interval graph, which can be colored optimally by using left-edge algorithm [28], [29]. For the other cases, we resort to a heuristic [30].

The latch-based register allocation problem (under the p-step-based scheduling) can also be formulated in the same way, except that we have two extra conditions that forbid register sharing.

1) Input and output operands of the same operation cannot share a register if the operation completes its execution in a transparent p-step.
2) Input and output operands of two different operations, respectively, cannot share a register if they complete their execution in the same transparent p-step.

The first condition is similar to the case of CRWO (see Fig. 3) except that input and output operands are different variables. Fig. 6(a) shows an example. Variable b is loaded in the second p-step, which is transparent; therefore, if it shares a register with a, the value a is replaced by b, which triggers a new addition; if a new output from the adder (due to its min-delay) starts to appear within the second p-step, the register is contaminated with the wrong value. To forbid register sharing between a and b, even though their lifetimes do not overlap, we introduce an extra edge in G_R between the vertices corresponding to a and b as shown in Fig. 6(a). It should be noted that this extra edge makes G_R a non-interval graph, thus forces us to use a heuristic coloring algorithm. If the adder completes its execution in the third p-step, which is non-transparent, and a controller is made to generate load_enable only in that p-step as shown in Fig. 6(b), b is loaded only after the fourth p-step begins; therefore a and b can share a register. Note that the register sharing between a and b is always disallowed in the c-step-based scheduling; thus, the p-step-based scheduling helps in the register allocation as well as in reducing the latency.

Fig. 6(c) shows an example of the second condition. Variable c, the output of the second adder, is loaded in the second p-step, which is transparent; therefore, if it shares a register with a, the input of the first adder that also completes in the

Algorithm *Refine-Schedule*

L1 Create a register conflict graph $G_R = (V, E_o \cup E_e)$
L2 $L = \{v \in V | \exists e \in E_e : e$ is incident with $v\}$
L3 **while** $L \neq \{\}$ **do**
L4 Select $v \in L$ of max degree; $L \leftarrow L - \{v\}$
L5 Move $producer(v)$ one p-step earlier or later if possible
L6 **if** there is a change in resources or latency **then** restore
L7 **else** remove corresponding $e \in E_e$; update G_R
L8 *Vertex-Color* (G_R)

Fig. 7. Algorithm to refine schedule to reduce the extra edges of G_R.

second p-step, the value a is replaced by c, which triggers a new addition in the first adder thereby causing a similar problem as in Fig. 6(a). This can be resolved by adding an extra edge between the vertices corresponding to a and c. Note that when the second condition happens, the first condition also happens, which is why we have an extra edge between a and b as well. If the first adder completes its execution one or more p-steps earlier than the second adder does, a and c can share a register. Similar to Fig. 6(b), if two adders complete their execution in a non-transparent p-step as shown in Fig. 6(d) with load_enable being generated only in that p-step, a and c can share a register as well.

2) *Refining Schedule to Improve Register Allocation:* The register allocation is typically performed after the operation scheduling. Fig. 6, however, implies that if we can make a slight change on a schedule so that some cases as (a) are refined to (b) and some as (c) to (d), we can reduce the number of extra edges in G_R, which helps reduce the number of registers during the register allocation.

The algorithm *refine-schedule* is shown in Fig. 7. We first create a register conflict graph (L1); there are two groups of edges, one group E_o due to the overlap of lifetimes and the other E_e having extra edges (see Fig. 6). A list L of vertices, each of which is an incident with at least one extra edge from E_e, indicating a candidate for refinement, are derived (L2). We heuristically select the vertex of maximum degree from L (L4), since vertices of larger degree are more difficult to color, i.e., they are likely to increase the number of registers. We then try to move the operation (L5) that produces a variable corresponding to the selected vertex, denoted by $producer(v)$, one p-step earlier or later (so that $producer(v)$ can complete its execution in a non-transparent p-step), if this move does not interfere with the predecessors or successors of $producer(v)$. If the move violates the resource or the latency constraint, it is canceled (L6); otherwise the extra edges that become invalid due to the move are removed from E_e (L7) and G_R is updated (note that there can be a change in E_o as well after the move). The process (L4–L7) is repeated until L becomes empty. The new G_R is then submitted to the vertex coloring (L8) for the register allocation.

The results by *refine-schedule* on HLS benchmark designs [31] are shown in Table I, where the first two columns are the name and the resource constraint (as the number of multipliers and adders) for *resource-constrained-list*. The third column denotes the number of vertices of G_R. Columns 4–6 are the numbers without performing *refine-schedule*, where the number of registers is obtained from the heuristic coloring [30]; the next three columns report the difference of corresponding numbers after *refine-schedule*.

Since some operations are moved as a result of *refine-schedule*, there are slight changes in the overlap of lifetimes ($\Delta|E_o|$). The decrease of extra edges ($\Delta|E_e|$) is dependent on the operation schedule, i.e., smaller decrease for a tight schedule. The change in the number of registers (ΔReg.) is, however, very marginal. This is understandable if we check the number of registers without any extra edges, i.e., the vertex coloring of $G'_R = (V, E_o)$, which is shown in the last column as a difference from the sixth column; this can serve as a loose lower bound for the number of registers obtained by *refine-schedule* since ignoring extra edges is not implementable. Therefore, even though we may consider a radical approach such as a combined scheduling and register allocation rather than using the simple heuristic *refine-schedule*, there is not much room for improvement.

To assess our register allocation (called RA-l), we use the method of [20] (called RA-LT) as a reference of comparison. RA-LT modifies the lifetime of variables by increasing all lifetimes by one c-step, at a cost of increase in the number of registers, so that latches hold their values one more cycle after they are last used. This allows all the flip-flops to be safely replaced by latches. The results of two register allocation methods are compared in Table II; column 3 reports the number of registers from RA-l, which was performed after *refine-schedule*; columns 4 and 5 show the number of registers from RA-l and its difference from that of RA-l, respectively. RA-l achieves smaller numbers of registers in many circuits; this is because we consider the conflict of latch sharing during scheduling step so that such conflict can be minimized and conflict is assumed only when it is necessary (see Fig. 6) while [20] forces conflict between all pairs whose lifetimes abut.

3) *Allocation of Functional Units and Connections:* The allocation of FUs is also formulated as the vertex coloring on a resource conflict graph [25] for each type of operation; each vertex in one of these graphs corresponds to an operation, and there is an edge between two vertices if they cannot share the same FU, i.e., if there is an overlap of p-steps when they use their FUs.

The connection allocation uses multiplexers (or buses) and wires to connect registers to FUs and FUs back to registers. It is also formulated as the vertex coloring on a connection conflict graph for each destination of data transfer, which is either FU or register. The connections of the same color, i.e., data transfers that can be shared, are routed to the same destination using a multiplexer.

Multiplexers take a good portion of total chip area and power consumption; it is thus important to minimize the number of multiplexers as well as the size of each multiplexer. Optimizing multiplexers can be performed either during FU and register allocation [32] or after these allocations are done [33]. We used a heuristic [33] that tries to detect and remove redundant connections to multiplexers, which help

TABLE I
REGISTER ALLOCATION WITH AND WITHOUT *Refine-Schedule*

Bench.	(*, +)	$\|V\|$	Without *Refine-Schedule*			With *Refine-Schedule*			ΔReg. With
			$\|E_o\|$	$\|E_e\|$	Reg.	$\Delta\|E_o\|$	$\Delta\|E_e\|$	ΔReg.	(V, E_o)
Iir7	(1, 1)	36	182	19	22	−4	−10	0	−1
	(2, 1)	36	233	10	24	−1	−2	0	−1
	(2, 2)	36	224	12	24	−1	−2	0	−1
Fir11	(1, 1)	37	330	17	19	−7	−8	0	−1
	(2, 1)	37	318	20	20	−2	−2	0	−2
Fir7	(1, 1)	23	200	15	18	1	−7	0	−1
	(2, 1)	23	211	4	17	0	0	0	0
	(2, 2)	23	201	6	18	0	0	0	−1
Elliptic	(1, 1)	44	509	25	19	0	0	0	−1
	(1, 2)	44	476	12	18	−2	−2	0	0
Lattice	(1, 1)	32	237	20	13	−2	−4	0	0
	(2, 1)	32	210	28	13	0	0	0	−1
Volterra	(1, 1)	46	344	38	15	−3	−14	0	−1
	(2, 1)	46	347	39	16	−7	−6	0	−2
	(3, 1)	46	359	79	17	0	−2	0	−3
Wavelet	(1, 1)	66	881	27	32	−2	−26	−1	−1
	(2, 2)	66	898	42	34	−12	−30	−1	−2
	(3, 2)	66	965	55	36	−1	−2	0	−2
Wdf7	(1, 1)	66	1377	10	33	0	0	0	0
	(2, 2)	66	1357	40	34	−4	0	0	0
	(3, 2)	66	1328	47	35	1	−3	0	−1
Ar	(1, 1)	44	378	9	16	−2	−2	0	0
	(2, 1)	44	365	16	17	−4	−2	−1	−1
	(2, 2)	44	361	37	17	−4	−2	−1	−1
Diffeq	(1, 1)	18	72	4	12	0	0	0	0
	(2, 1)	18	72	8	13	2	−5	−1	−1
	(2, 2)	18	71	11	13	2	−5	−1	−1

TABLE II
COMPARISON OF RA-L AND RA-LT [20]

Benchmark	(∗, +)	RA-l	RA-LT	
		# Reg.	# Reg.	RA-LT − RA-l
Iir7	(1, 1)	22	23	1
	(2, 1)	24	25	1
	(2, 2)	24	25	1
Fir11	(1, 1)	19	19	0
	(2, 1)	20	20	0
Fir7	(1, 1)	18	18	0
	(2, 1)	17	19	2
	(2, 2)	18	20	2
Elliptic	(1, 1)	19	20	1
	(1, 2)	18	21	3
Lattice	(1, 1)	13	14	1
	(2, 1)	13	14	1
Volterra	(1, 1)	15	15	0
	(2, 1)	16	16	0
	(3, 1)	17	17	0
Wavelet	(1, 1)	31	32	1
	(2, 2)	33	34	1
	(3, 2)	36	35	−1
Wdf7	(1, 1)	33	34	1
	(2, 2)	34	36	2
	(3, 2)	35	36	1
Ar	(1, 1)	16	17	1
	(2, 1)	16	17	1
	(2, 2)	16	17	1
Diffeq	(1, 1)	12	14	2
	(2, 1)	12	14	2
	(2, 2)	12	14	2
Average				1

reduce the size of multiplexers; an example of such redundant connections is two variables stored in the same register, but routed to different operands of the same FU.

C. Control Synthesis

The control synthesis is responsible for generating control signals for the data-path, such as a multiplexer select, a function select of multifunction units such as ALU, and a load-enable for registers. This is accomplished by describing the behavior of a controller as a state transition graph (STG) followed by the conventional sequential and logic synthesis. The state corresponds to a c-step in the conventional HLS; it corresponds to a p-step in our approach, meaning that we need twice the number of states in STG for the same latency of schedule. Since the number of flip-flops to implement STG of n states is at least $\lceil \log_2 n \rceil$, we need one more flip-flop in our controller.

Since the controller needs to generate control signals at both edges of the data-path clock, we either have to use a separate clock for the controller with twice the frequency of data-path clock or use a dual-edge-triggered flip-flop (DETFF)[1] with the same data-path clock. Note that the former approach can be used only when the duty cycle W/T is 0.5, because otherwise one of the edges of the data-path clock is not synchronized

[1]DETFF is triggered, thus launches and captures data, at both the clock edges; the conventional flip-flop, which is triggered either at the rising-edge or at the falling-edge of the clock but not at both, is called a single-edge-triggered flip-flop (SETFF) [34].

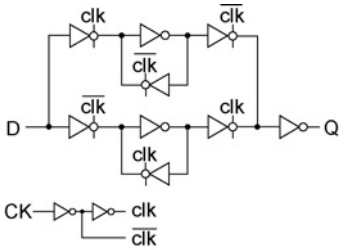

Fig. 8. Latch-mux implementation of D-type DETFF [35].

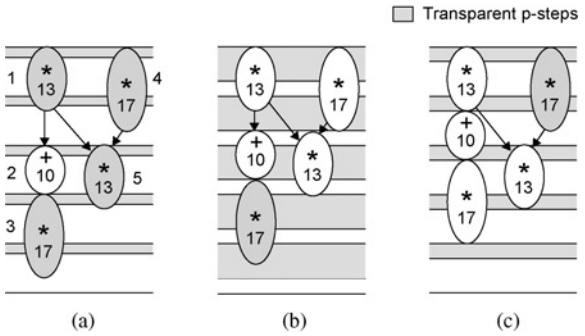

Fig. 9. Different schedules for different duty cycles for T = 10 time units. (a) W = 2 time units. (b) W = 7 time units. (c) W = 3 time units.

Algorithm *Optimize-Duty-Cycle*

L1 **for** each operation i **do**
L2 **if** $r_i = 0$ **then** $W_i \leftarrow [0, T]$
L3 **else**
L4 **if** $r_i < \frac{T}{2}$ **then** $W_i \leftarrow [r_i, T - r_i]$
L5 **else** $W_i \leftarrow [0, T - r_i] \vee [r_i, T]$
L6 $W \leftarrow \wedge W_i, \forall i$
L7 **if** $W = \emptyset$ **then** $W \leftarrow$ *Heuristic-Duty-Cycle* $(W_i, \forall i)$

(a)

Algorithm *Heuristic-Duty-Cycle* $(W_i, \forall i)$

L8 Perform initial scheduling with $W = T/2$
L9 **for** each operation i **do**
L10 $oc_i \leftarrow \{j | FU_t(j) = FU_t(i), slack(j) = 0\}$
L11 $weight_i = |oc_i|$
L12 **if** $r_i < T/2$ **then** $weight_i \leftarrow 2 \cdot weight_i$
L13 $W \leftarrow [0, T]$
L14 **for** each operation i in decreasing $weight_i$ **do**
L15 **if** $W \wedge W_i \neq \emptyset$ **then** $W \leftarrow W \wedge W_i$
L16 return W

(b)

Fig. 10. (a) An algorithm to find an optimal duty cycle and (b) a heuristic algorithm to determine the duty cycle when the optimal solution is intractable.

with the controller clock; there is also a potential to reduce the latency by adjusting the duty cycle as presented in Section IV. This leads us to adopt the latter approach for implementing the controller.

There are various implementations of DETFF [34], but we select the latch-mux structure [35] for its relatively low sequencing overhead. Fig. 8 shows the latch-mux style DETFF; it was sized so that its timing parameters are comparable to those of its SETFF counterpart as shown in Table III, which allows us to simply substitute DETFFs for SETFFs in the original synthesized gate-level netlist of the controller (see overall design flow of HLS-l in Section VI).

TABLE III
COMPARING TIMING PARAMETERS (IN PS) OF D-TYPE SETFF AND DETFF IN 45-NM TECHNOLOGY, V_{dd} = 1.1-V

	SETFF	DETFF	
		Rising-Edge	Falling-Edge
Setup time	14.0	18.4	14.2
Hold time	−10.8	−13.4	−11.7
Clock-to-Q delay	61.5	58.7	70.0

IV. OPTIMIZING DUTY CYCLE TO IMPROVE SCHEDULE

The duty cycle (W/T) is typically 0.5. However, the clock with the duty cycle other than 0.5 can be equally well generated without significant overhead [36]. This provides us the opportunity to minimize the latency because the p-step based execution delay (2) is a function of W for a fixed T. For example, in Fig. 9, if W is too small [Fig. 9(a)], the execution delays of shaded multipliers are all four p-steps, which yields the latency of ten p-steps. For large value of W shown in Fig. 9(b), only one multiplier takes four p-steps, but the latency is still nine p-steps. Fig. 9(c) shows the best schedule having eight p-steps. Using the duty cycle other than 0.5 forces the controller to have smaller delay, because the combinational logic in a DETFF-based implementation is constrained by the smaller of a transparent or a non-transparent phase of the clock. This, however, is not a significant problem, since typically the controller delay is already very small.

The algorithm *optimize-duty-cycle* is shown in Fig. 10. It seeks W that minimizes the latency for a given T. We have a better chance to minimize the latency when each operation has a minimum execution delay, even though achieving the minimum latency is not guaranteed because the scheduling problem, which determines an exact latency, belongs to the class of NP-complete problems in general. For each operation i, we want to determine the values of W, denoted by W_i, minimizing d_i (L1–L5). From (2), d_i becomes minimum if residual delay r_i is 0 regardless of the values of W_i; thus W_i can take any value from 0 to T (L2). When r_i is not 0 (L3), meaning that $\lceil r_i/T \rceil$ evaluates to 1, d_i is minimized when both $\lceil (r_i - W_i)/T \rceil$ and $\lceil [r_i - (T - W_i)]/T \rceil$ become 0, which imply $r_i - W_i < 0$ and $r_i - (T - W_i) < 0$. Solving for W_i yields

$$r_i < W_i < T - r_i \tag{3}$$

provided that $r_i < T - r_i$, i.e., $r_i < T/2$ (L4). If $r_i > T/2$ (L5), either $\lceil (r_i - W_i)/T \rceil$ or $\lceil [r_i - (T - W_i)]/T \rceil$ can be made 0 but not both, which implies that W_i consists of two intervals; one from 0 to $T - r_i$ and the other from r_i to T. In other words, d_i is unique when $r_i < T/2$; otherwise, d_i varies depending on whether i is scheduled in a transparent or in a non-transparent p-step.

Once we determine W_i that minimizes the execution delay of i for all the operations, we form their intersection, which

then becomes W (L6). If W is not empty and consists of a single continuous interval, any value from W yields the same schedule for a given scheduling method, thus yields the same latency. If W consists of more than one interval, it can be readily shown that the number of intervals is exactly two. The value from each interval yields different schedules, because at least one operation i has $r_i > T/2$, i.e., its execution delay is not unique (recall L5). In this case, therefore, we perform scheduling twice and select the one with a smaller latency.

If the intersection is empty (L7), there is no such W that minimizes the execution delay of all the operations. Thus, the problem now becomes intractable, because we cannot choose W minimizing the latency unless we perform the operation scheduling for every possible W value. The duty cycle, in this case, is determined by a heuristic algorithm *heuristic-duty-cycle*. Initial scheduling is performed with any value of W, say $T/2$ (L8). Then a weight, $weight_i$, is assigned to each operation i (L10–L12), which determines a priority of its W_i being considered when we decide on the value of W. For this purpose, we find a list oc_i of operations (including i itself) that are assigned to the same type of FU and have a zero slack (L10), i.e., operations cannot be moved without altering the schedule of other operations. The $weight_i$ is chosen as the cardinality of oc_i (L11), since those operations with more elements in oc_i are more likely to increase the latency. If $r_i < T/2$, the weight is made bigger (L12) because, when W_i is reflected in deciding W, d_i becomes unique (recall L4), which helps reduce the latency. We form the intersection of W_i starting from i having the largest $weight_i$ unless taking an intersection with W_i does not yield an empty interval (L14 and L15).

Example 3: Consider an example in Fig. 9. Following L1 to L5 in Fig. 10 yields

$$W_1 = [3, 7] \qquad W_2 = [0, 10]$$
$$W_3 = [0, 3] \vee [7, 10] \qquad W_4 = [0, 3] \vee [7, 10]$$
$$W_5 = [3, 7].$$

Taking their intersections (L6) gives $W = [3, 3] \vee [7, 7]$. Fig. 9(c) shows the schedule for $W = 3$ time units and Fig. 9(b) for $W = 7$ time units, and thus we select the former. □

V. Experimental Results

We carried out experiments on a set of behavioral benchmark designs [31] to assess the effectiveness of HLS-l, which integrates the procedures presented in Sections III and IV. It was also applied to a module extracted from H.264/AVC to demonstrate the application of HLS-l to industrial designs. HLS-l was implemented in C under Centos 5.0; it takes a behavioral VHDL as an input and outputs the register transfer level (RTL) description of the data-path and the controller, which is then synthesized [37] in a commercial 1.1-V, 45 nm bulk CMOS technology (see Fig. 14). The conventional c-step based HLS, which we call HLS, was also implemented as a reference of comparison.

Table IV summarizes circuit elements used in the experiment, each one with its area and delay. For all benchmark designs in Table V, we used 4.4 ns of the clock period for the operation scheduling; this is slightly larger than the maximum delay along the data-path consisting of adder/subtractor, registers, and two 10-to-1 multiplexers, which is $3417 + 61 + 60 + 2 \cdot 300 = 4138$ ps. The exact number of inputs of each multiplexer and the number of multiplexers along the data-path can only be determined after the register allocation and the connection allocation; thus, two 10-to-1 multiplexers are assumed as an upper bound for the delay through multiplexers. The clock period is adjusted for each design once the timing analysis is performed on the synthesized gate-level netlist.

TABLE IV
Area and Delay of (32-Bit) Circuit Elements Used in the Experiment

Circuits	Area (μm^2)	Delay (ps)
Adder/subtractor	355	3417
Multiplier	2398	5086
F/F register	189	75/65
Latch register	138	61/60
2-to-1 MUX	65	68
10-to-1 MUX	457	293

Register delay is denoted by clock-to-Q delay/setup time.

Since the adder/subtractor dictates the clock period, any operation that is assigned to it always takes a single clock period, i.e., two p-steps. Therefore, the optimum duty cycle is determined by the multiplier alone. The maximum data-path delay through the multiplier is $5086+61+60+2\cdot300 = 5807$ ps; its residual delay is thus $5807 - 4400 \approx 1400$ ps, meaning that the optimum duty cycle is simply any value between 0.32 (1.4/4.4) and 0.68 [(4.4 − 1.4)/4.4] for all benchmark examples (see L4 of Fig. 10). We will consider more variety of FUs to see the effect of the optimum duty cycle on the latency in Section VI-B.

A. Latency

We compared the latency obtained by HLS-l and HLS under the same resource constraints; *resource-constrained-list* in Fig. 4 is used for HLS-l and the conventional resource-constrained list scheduling [26] is used for HLS. The comparisons are summarized in Table V. The second column shows the resource constraint, expressed as the number of multipliers and adder/subtractors. The results produced by HLS are shown in the next three columns, and the results from HLS-l are shown in the following three columns. The last three columns show the reduction in the clock period, the latency in the number of c-steps, and the latency in ns achieved by HLS-l over HLS.

The clock period (T) is the maximum latch to latch delay in HLS-l and the maximum flip-flop to flip-flop delay in HLS, which were reported by the timing analysis [38] on the gate-level netlist. It has similar values in HLS-l and HLS, even though latch has a lower sequencing overhead than flip-flop. This makes sense because the sequencing overhead takes relatively small proportion of the clock period (see Table IV).

HLS-l reports the latency (L) as the number of p-steps, which is rounded up to the number of c-steps for comparison to HLS. The latency in ns, which is the product of T and L, is reduced by 18.2% on average as shown in the last column; L is reduced by 3.8 c-steps and T is reduced by 0.15 ns, on

TABLE V
COMPARISON OF HLS AND HLS-L

Benchmark	Resource Constraint (*, +)	HLS			HLS-l			Savings		
		T (ns)	L (c-step)	$T \cdot L$ (ns)	T (ns)	L (c-step)	$T \cdot L$ (ns)	ΔT (ns)	ΔL (c-step)	$\Delta(T \cdot L)$ (%)
Iir7	(1, 1)	3.9	31	122	3.7	24	88	0.3	7	27.9
	(2, 1)	3.9	20	79	3.9	18	70	0.1	2	11.6
	(2, 2)	4.0	18	72	3.9	16	63	0.1	2	12.9
Fir11	(1, 1)	4.0	23	92	3.4	18	61	0.6	5	33.3
	(2, 1)	3.7	13	48	3.7	12	45	0.0	1	8.2
Fir7	(1, 1)	3.8	16	61	3.7	14	51	0.2	2	16.6
	(2, 1)	3.7	11	41	3.9	10	39	−0.1	1	5.7
	(2, 2)	3.8	9	34	3.8	8	30	0.0	1	11.3
Elliptic	(1, 1)	4.3	28	121	4.0	28	111	0.3	0	7.9
	(1, 2)	4.2	22	92	4.0	19	75	0.2	3	18.4
Lattice	(1, 1)	3.9	20	79	3.7	16	59	0.2	4	24.7
	(2, 1)	3.9	12	46	3.8	11	42	0.1	1	9.8
Volterra	(1, 1)	3.8	36	138	3.7	28	102	0.2	8	25.7
	(2, 1)	4.0	20	79	3.7	16	60	0.2	4	24.5
	(3, 1)	3.9	15	58	3.9	14	54	0.0	1	7.4
Wavelet	(1, 1)	3.9	57	224	3.6	43	157	0.3	14	30.1
	(2, 2)	3.9	30	116	3.8	23	88	0.1	7	24.3
	(3, 2)	4.0	21	84	3.9	17	66	0.1	4	21.1
Wdf7	(1, 1)	4.1	39	160	4.0	29	115	0.1	10	28.2
	(2, 2)	4.1	20	83	3.9	17	66	0.3	3	20.1
	(3, 2)	4.0	18	72	3.9	16	63	0.1	2	13.3
Ar	(1, 1)	4.0	36	143	3.8	26	99	0.2	10	30.5
	(2, 1)	3.9	21	82	3.8	18	68	0.2	3	17.6
	(2, 2)	3.9	19	74	3.7	16	59	0.2	3	19.3
Diffeq	(1, 1)	3.8	15	57	3.7	13	48	0.1	2	15.2
	(2, 1)	3.8	11	42	3.7	10	37	0.1	1	11.2
	(2, 2)	3.8	9	34	3.6	8	29	0.1	1	13.7
Average								0.15	3.8	18.2

T is clock period, L is the latency in the number of c-steps, and $T \cdot L$ corresponds to the latency in ns.

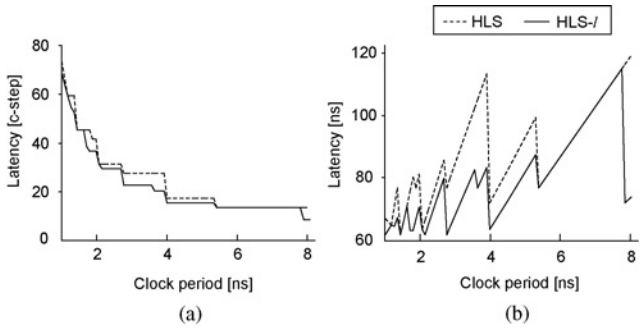

Fig. 11. Latency (a) as the number of c-steps and (b) as ns for design Iir7 for the varying clock period.

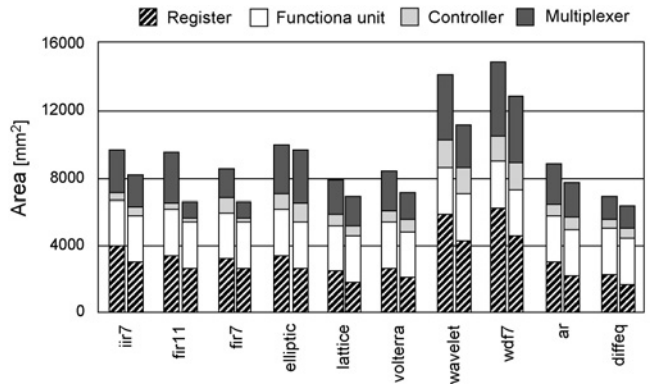

Fig. 12. Comparison of area of designs produced by HLS (left-hand bars) and HLS-l (right-hand bars) for one multiplier and one adder/subtractor as a resource constraint.

average. The extent of saving of each benchmark is determined by the number of multiplications on a critical path, a chain of operations that dictates the latency, because we take advantage of the p-step-based scheduling in multiplications but not in additions/subtractions (a multiplication takes three p-steps in HLS-l; it takes two c-steps, equivalent to four p-steps, in HLS). Design Fir11 with one multiplier and one adder/subtractor achieves the most saving as its critical path consists of 11 multiplications and one addition. For the same resource constraint, Elliptic cannot benefit from using the p-step-based scheduling, in terms of latency in c-step, because its critical path has 26 additions/subtractions but only one multiplication; its latency, however, is still reduced due to the low sequencing overhead of latches.

Latency strongly depends on the selection of the clock period [14], [15]. We performed HLS and HLS-l on Iir7 with two multipliers and two adder/subtractors as the resource constraint while we change the clock period from 1 ns to 9 ns with the duty cycle fixed at 0.5; we did not perform operation chaining in this example. Fig. 11(a) shows the latency as the number of c-steps and Fig. 11(b) as ns. The number of c-steps naturally decreases with the increasing clock period, but, in terms of ns, there are several local minima. The clock period

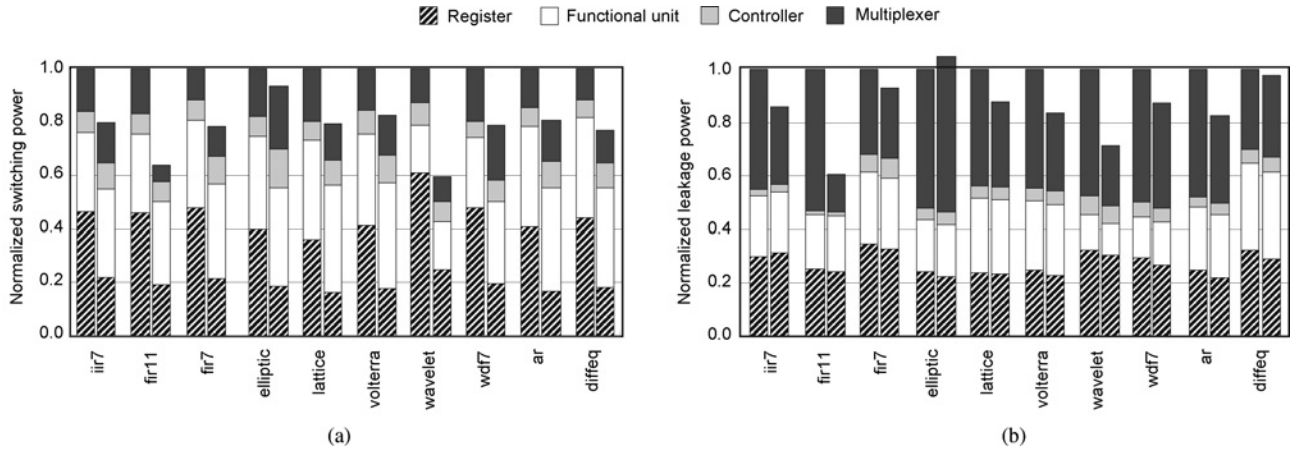

Fig. 13. Comparison of normalized (a) dynamic-power and (b) static-power consumption of designs produced by HLS (left-hand bars) and HLS-l (right-hand bars) for one multiplier and one adder/subtractor as a resource constraint.

we used for the experiment, 4.4 ns, is one of local minima, which justifies our selection of the clock period. The curves clearly show the benefit of HLS-l over HLS.

B. Area

The areas of each design, which is the sum of the areas of all the cells after logic synthesis, produced by HLS and HLS-l are compared; in Fig. 12, we report the area of designs with one multiplier and one adder/subtractor as a resource constraint.

The area is reduced by 9.2% on average, where saving mainly comes from registers. Note that the number of registers itself increases in HLS-l due to extra edges introduced in the register conflict graph, even though the extent of increase is very marginal as discussed in Section III-B1. Due to the smaller area of latch, however, registers produced by HLS-l occupy 23.5% less area on average than those produced by HLS. The area of controller increases by 10.6% on average due to the use of DETFFs and more number of states in HLS-l; the controller, however, occupies small proportion of the total area.

C. Power Consumption

The dynamic-power and static-power consumption are reported in Fig. 13, where the numbers are normalized to the power consumption of designs produced by HLS. The dynamic consumption is caused by switching computation; short-circuit current and leakage current constitute static consumption. The power consumption was obtained by simulating each circuit with a fast transistor-level circuit simulator [39] 100 times with different random input vectors and taking their average. Note that average power consumption during the same amount of time is compared in Fig. 13, meaning that the design produced by HLS-l is simulated for the amount of time equal to the latency of the same design produced by HLS.

The dynamic power is reduced by 16.6% on average, where saving mainly comes from registers similar to the area saving. The static power is reduced by 6.5% on average. In registers, leakage component decreases when latches are used, but short-circuit current increases due to many brief glitches, which explains seemingly no change in static power of registers. Multiplexers also suffer from many brief glitches, since they are typically fan-out of registers, which explains increasing proportion of multiplexers in static power consumption. The proportion of static power in total power consumption, however, is very small, about 6%. As a result, the total power consumption is reduced by 16.0% on average.

Note that the power consumption due to glitches is included in Fig. 13. Considering that latch-based designs typically suffer from more glitches than flip-flop-based ones, Fig. 13 implies that the reduction in capacitance of latches is large enough to outweigh the increase of power consumption due to glitches.

VI. Design Flow Using HLS-l and Case Study

A. Design Flow

The overall design flow based on HLS-l is shown in Fig. 14. A behavioral description written in VHDL is first analyzed [40] and is then transformed into a DFG [41]. The DFG will then be an input to HLS-l. The RTL description of data-path and controller generated by HLS-l go through a standard logic synthesis [37] to create an initial gate-level netlist. Commercially available IPs [42] are used for FUs.

Care needs to be taken in designing a controller. A controller is synthesized as a sequential circuit using SETFFs, because DETFFs are not typically supported by commercial synthesis tools [37]; SETFFs are thus manually replaced by DETFFs afterward as shown in Fig. 14. During synthesis, however, the timing constraint has to be properly set so that the controller functions correctly after we substitute DETFFs for SETFFs. This is done by arbitrarily setting the clock period of the controller as the smaller of the transparent (W) or the non-transparent phase ($T - W$) of data-path clock. Note that this is only for synthesis purpose; the data-path and the controller use the same clock in the implementation. Additional timing analysis is needed after we have DETFFs in the controller, to ensure that the controller satisfies the setup and hold-time constraints both at the rising-edge and falling-edge of the clock, since the corresponding timing parameters of SETFF and DETFF do not exactly match as shown in Table III. If timing analysis reports failure, an extra timing guardband is

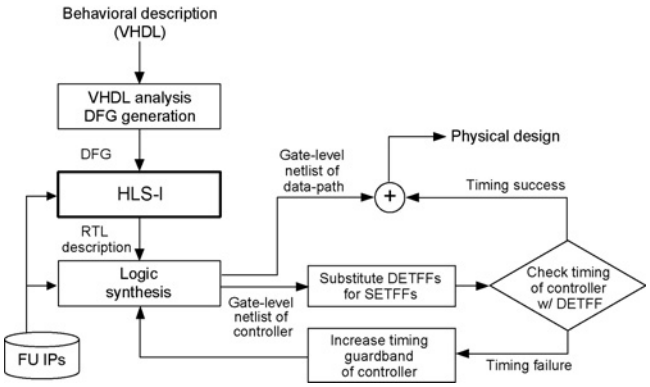

Fig. 14. Design flow using HLS-l.

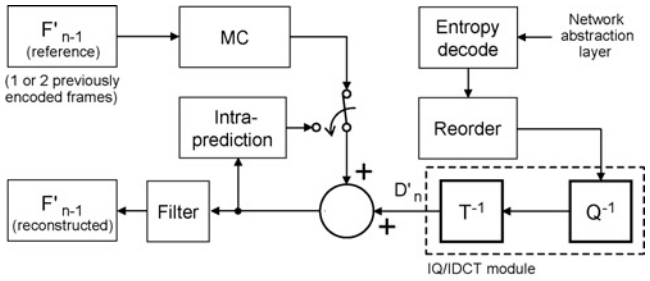

Fig. 15. Block diagram of the H.264/AVC decoder; the IQ/IDCT module is highlighted.

intentionally inserted into the controller, and re-synthesis is performed as shown in Fig. 14.

B. Case Study: H.264/AVC

We tested the design flow shown in Fig. 14 on H.264/AVC decoder, which implements one of the most popular international video coding standards [43]. Fig. 15 shows its block diagram, where we extracted a module named inverse quantization and inverse discrete cosine transform (IQ/IDCT) as a case study. The module is responsible for recovering the residual data from encoded data. The VHDL description of IQ/IDCT was transformed into a DFG consisting of 208 operations (64 multiplications, 64 additions, 64 subtractions, and 16 assignments).

The resource constraint was empirically determined as two fast multipliers, one slow multiplier, two fast adders, and one slow adder, where adder can also perform subtraction; D_i of each type of FU was 6.0 ns, 11.8 ns, 4.4 ns, and 6.8 ns, respectively. For the clock period of 4.4 ns, *optimize-duty-cycle* reported empty intersections of W_i (L6 of Fig. 10), i.e., there is no such width of transparent phase W that minimizes the execution delay of all the operations. *Heuristic-duty-cycle* then returned two intervals of W: $[1.6, 2.0]$ and $[2.4, 2.8]$. The scheduling was performed twice with the value from each interval, and $[1.6, 2.0]$ resulted in the smaller latency. To assess the selected interval of W, we also performed scheduling while we vary W from 1 ns to 4 ns; the result is illustrated in Fig. 16. Even though the interval of $[1.6, 2.0]$ was found out by the heuristic algorithm, it indeed was the optimal in this particular example. When the typical duty

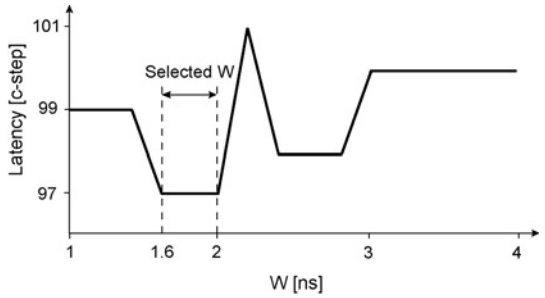

Fig. 16. Latency as the number of c-steps versus W.

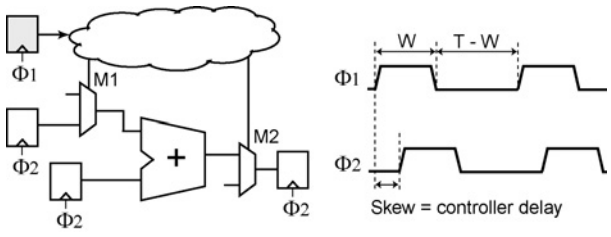

Fig. 17. Small amount of skew is assigned between the controller clock (Φ_1) and the data-path clock (Φ_2) so that multiplexer select M1 arrives in time to avoid its delay added into the data-path delay.

cycle of 0.5, which corresponds to $W = 2.2$ ns, is used, the latency becomes 101 c-steps, which is four c-steps worse than using the selected interval.

If we select 2.0 ns for W, for example, which is smaller than $T - W = 2.4$ ns, that value has to be used as maximum DETFF to DETFF delay when we synthesize the controller as we have discussed in Section VI. The controller delay may be factored in the data-path delay. As shown in Fig. 17, if the multiplexer select M1 is not available when an input data of the same multiplexer has arrived, the input data is effectively put on hold until M1 arrives, thereby increasing the data-path delay. This can be alleviated by applying a negative skew to Φ_1 shown in Fig. 17 (or equivalently by applying a positive skew to Φ_2) so that M1 arrives in time for the input data. The amount of skew that can be reliably realized is limited [44], because within-die variations affect extra buffers and wires inserted to implement large skews in randomly different amount, thereby causing uncertainties in skews. In our implementation, therefore, the controller was synthesized with very tight timing constraint (less than 2.0 ns), which resulted in 0.8 ns of maximum delay of the controller.

VII. Conclusion

We have proposed a comprehensive solution to the new problem of latch-based HLS. The main idea is the use of p-step, which enables scheduling at both edges of the clock to achieve a tighter schedule. It also helps in the register allocation, since scheduling operations so that they complete in a non-transparent p-step allows us to resolve read/write conflict inherent in latch-based registers. The controller supporting the proposed scheduling is implemented using DETFFs. In addition, we showed that optimizing the clock duty cycle can further reduce the latency. We tested HLS-l on several

benchmark designs implemented in 1.1-V, 45 nm CMOS technology. Compared with the conventional HLS, the latency was reduced by 18.2% on average with 9.2% less area and 16.0% less power consumption.

Latches are generally considered difficult to use for their complicated timing behavior. They are made manageable in HLS-l by combined efforts during the operation scheduling, allocation, and control synthesis, specifically by handling CRWOs, by introducing extra conflict edges to the register conflict graph, and by using load_enable per phase-step basis.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their constructive comments and suggestions, and I. Han of KAIST, Daejeon, Korea, for the help with the mux experiment.

REFERENCES

[1] S. Paik, I. Shin, and Y. Shin, "HLS-l: High-level synthesis of high performance latch-based circuits," in *Proc. Design Autom. Test Europe Conf. Exhibition*, Apr. 2009, pp. 1112–1117.

[2] D. Chinnery and K. Keutzer, "Introduction and overview of the book," in *Closing the Gap Between ASIC and Custom*. Kluwer Academic, 2002, pp. 4–28.

[3] V. Zhirnov, R. Cavin, J. Hutchby, and G. Bourianoff, "Limits to binary logic switch scaling: A Gendanken model," *Proc. IEEE*, vol. 91, no. 11, pp. 1934–1939, Nov. 2003.

[4] P. Restle, T. McNamara, D. Webber, P. Camporese, K. Eng, K. Jenkins, D. Allen, M. Rohn, M. Quaranta, D. Boerstler, C. Alpert, C. Carter, R. Bailey, J. Petrovick, B. Krauter, and B. McCredie, "A clock distribution network for microprocessors," *IEEE J. Solid-State Circuits*, vol. 36, no. 5, pp. 792–799, May 2001.

[5] S. Held, B. Korte, J. Maßberg, M. Ringe, and J. Vygen, "Clock scheduling and clocktree construction for high performance ASICs," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 2003, pp. 232–239.

[6] F. Klass, C. Amir, A. Das, K. Aingaran, C. Truong, R. Wang, A. Mehta, R. Heald, and G. Yee, "A new family of semidynamic and dynamic flip-flops with embedded logic for high-performance processors," *IEEE J. Solid-State Circuits*, vol. 34, no. 5, pp. 712–716, May 1999.

[7] M. Hamada, T. Terazawa, T. Higashi, S. Kitabayashi, S. Mita, Y. Watanabe, M. Ashino, H. Hara, and T. Kuroda, "Flip-flop selection technique for power-delay trade-off," in *Proc. IEEE Int. Solid-State Circuits Conf.*, Feb. 1999, pp. 270–271.

[8] S. Naffziger, G. Colon-Bonet, T. Fischer, R. Riedlinger, T. Sullivan, and T. Grutkowski, "The implementation of the itanium 2 microprocessor," *IEEE J. Solid-State Circuits*, vol. 37, no. 11, pp. 1448–1460, Nov. 2002.

[9] H. Ando, Y. Yoshida, A. Inoue, I. Sugiyama, T. Asakawa, K. Morita, T. Muta, T. Motokurumada, S. Okada, H. Yamashita, Y. Satsukawa, A. Konmoto, R. Yamashita, and H. Sugiyama, "A 1.3-GHz fifth-generation SPARC64 microprocessor," *IEEE J. Solid-State Circuits*, vol. 38, no. 11, pp. 1896–1905, Nov. 2003.

[10] C. Yeh, E. Hsu, K. Cheng, J. Wang, and N. Chang, "An 830 mw, 586 kbps 1024-bit RSA chip design," in *Proc. Design Autom. Test Eur. Conf. Exhibition*, Mar. 2006, pp. 24–29.

[11] X. Liang, D. Brooks, and G. Wei, "A process-variation-tolerant floating-point unit with voltage interpolation and variable latency," in *Proc. IEEE Int. Solid-State Circuits Conf.*, Feb. 2008, pp. 404–406.

[12] D. G. Chinnery, B. Nikolic, and K. Keutzer, "Achieving 550 MHz in an ASIC methodology," in *Proc. Design Autom. Conf.*, Jun. 2001, pp. 420–425.

[13] V. S. Sathe, J. C. Kao, and M. C. Papaefthymiou, "Resonant-clock latch-based design," *IEEE J. Solid-State Circuits*, vol. 43, no. 4, pp. 864–873, Apr. 2008.

[14] S. Narayan and D. Gajski, "System clock estimation based on clock slack minimization," in *Proc. Eur. Design Autom. Conf.*, Sep. 1992, pp. 66–71.

[15] A. Naseer, M. Balakrishnan, and A. Kumar, "Optimal clock period for synthesized data paths," in *Proc. Int. Conf. Very Large Scale Integr. Design*, Jan. 1997, pp. 134–139.

[16] C.-T. Hwang, J.-H. Lee, and Y.-C. Hsu, "A formal approach to the scheduling problem in high level synthesis," *IEEE Trans. Comput.-Aided Design*, vol. 10, no. 4, pp. 464–475, Apr. 1991.

[17] M. R. Corazao, M. A. Khalaf, L. M. Guerra, M. Potkonjak, and J. M. Rabaey, "Performance optimization using template mapping for datapath-intensive high-level synthesis," *IEEE Trans. Comput.-Aided Design*, vol. 15, no. 8, pp. 877–888, Aug. 1996.

[18] S. Park and K. Choi, "Performance-driven high-level synthesis with bit-level chaining and clock selection," *IEEE Trans. Comput.-Aided Design*, vol. 20, no. 2, pp. 199–212, Feb. 2001.

[19] T. Wu and Y. Lin, "Storage optimization by replacing some flip-flops with latches," in *Proc. Eur. Design Autom. Conf.*, Sep. 1996, pp. 296–301.

[20] W. Yang, I. Park, and C. Kyung, "Low-power high-level synthesis using latches," in *Proc. Asia South Pacific Design Autom. Conf.*, Jan. 2001, pp. 462–465.

[21] Y. Chen and Y. Xie, "Tolerating process variations in high-level synthesis using transparent latches," in *Proc. Asia South Pacific Design Autom. Conf.*, Jan. 2009, pp. 73–78.

[22] M. McFarland, A. Parker, and R. Camposano, "The high-level synthesis of digital systems," *Proc. IEEE*, vol. 78, no. 2, pp. 301–318, Feb. 1990.

[23] G. Gupta, M. Gupta, and P. R. Panda, "Rapid estimation of control delay from high-level specifications," in *Proc. Design Autom. Conf.*, Jul. 2006, pp. 455–458.

[24] G. De Micheli, "Scheduling algorithms," in *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994, pp. 187–193.

[25] G. De Micheli, "Resource sharing and binding," in *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994, pp. 230–245.

[26] T. C. Hu, "Parallel sequencing and assembly line problems," *Oper. Res.*, vol. 9, no. 6, pp. 841–848, Dec. 1961.

[27] P. G. Paulin and J. P. Knight, "Force-directed scheduling for the behavioral synthesis of ASICs," *IEEE Trans. Comput.-Aided Design*, vol. 8, no. 6, pp. 661–679, Jun. 1989.

[28] A. Hashimoto and J. Stevens, "Wire routing by optimizing channel assignment within large apertures," in *Proc. Design Autom. Workshop*, Jun. 1971, pp. 155–169.

[29] F. Kurdahi and A. Parker, "REAL: A program for register allocation," in *Proc. Design Autom. Conf.*, Jun. 1987, pp. 210–215.

[30] D. Brelaz, "New methods to color the vertices of a graph," *Commun. ACM*, vol. 22, no. 4, pp. 251–256, Apr. 1979.

[31] *High level synthesis benchmark* [Online]. Available: http://bears.ece.ucsb.edu/cad

[32] T. Kim and X. Liu, "Compatibility path-based binding algorithm for interconnect reduction in high level synthesis," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 2007, pp. 435–441.

[33] D. Chen and J. Cong, "Register binding and port assignment for multiplexer optimization," in *Proc. Asia South Pacific Design Autom. Conf.*, Jan. 2004, pp. 68–73.

[34] V. G. Oklobdzija, V. M. Stojanovic, D. M. Markovic, and N. M. Nedovic, "State-of-the-art clocked storage elements in CMOS technology," in *Digital System Clocking: High-Performance and Low-Power Aspects*. Wiley, 2003, pp. 180–186.

[35] R. Llopis and M. Sachdev, "Low power, testable dual edge triggered flip-flops," in *Proc. Int. Symp. Low Power Electron. Design*, Aug. 1996, pp. 341–345.

[36] Y.-J. Wang, S.-K. Kao, and S.-I. Liu, "All-digital delay-locked loop/pulsewidth-control loop with adjustable duty cycles," *IEEE J. Solid-State Circuits*, vol. 41, no. 6, pp. 1262–1274, Jun. 2006.

[37] *Design Compiler User Guide*, Synopsys, Inc., Mountain View, CA, Mar. 2007.

[38] *Prime Time User Guide*, Synopsys, Inc., Mountain View, CA, Dec. 2006.

[39] *NanoSim User Guide*, Synopsys, Inc., Mountain View, CA, Dec. 2007.

[40] T. Ahn, K. Kim, S. Park, and K. Choi, "Incremental analysis and elaboration of VHDL description," in *Proc. Asia Pacific Conf. Hardw. Description Languages*, Jan. 1996, pp. 128–131.

[41] J. Jeon, Y. Ahn, and K. Choi, "CDFG toolkit user's guide," Seoul Nat. Univ., Tech. Rep. SNU-EE-TR-2002-8, Aug. 2002.

[42] *DesignWare IP Family Reference Guide*, Synopsys, Inc., Mountain View, CA, Dec. 2007.

[43] *ITU-T Recommendation H.264*, ITU-T, Mar. 2009 [Online]. Available: http://www.itu.int/rec/T-REC-H.264

[44] K. M. Carrig, "Chip clocking effect on performance for IBMs SA-27E ASIC technology," *IBM Micronews*, vol. 6, no. 3, pp. 12–16, 2000.

Seungwhun Paik (S'07) received the B.S. degree in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, in 2006. He is currently working toward the Ph.D. degree from the Department of Electrical Engineering, KAIST.

His current research interests include computer-aided design for high-performance designs, low power designs, high-level synthesis, and structured application-specific integrated circuit.

Insup Shin (S'09) received the B.S. and M.S. degrees in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, in 2007 and 2009, respectively. He is currently working toward the Ph.D. degree from the Department of Electrical Engineering, KAIST.

His current research interests include very large scale integration design methodology and computer-aided design for low-power and high-performance integrated circuits.

Taewhan Kim (SM'08) received the B.S. degree in computer science and statistics and the M.S. degree in computer science from Seoul National University (SNU), Seoul, Korea in 1985 and 1987, respectively, and he received the Ph.D. degree in the field of computer science from the University of Illinois at Urbana-Champaign, Champaign, in 1993.

From 1993 to 1998, he was a Software Engineer with Lattice Semiconductor Corporation, Hillsboro, OR, and Synopsys, Inc., Mountain View, CA, where he was involved in logic and high-level synthesis. From 1998 to 2003, he was with the Department of Electrical Engineering and Computer Science, Korea Advanced Institute of Science and Technology, Daejeon. Currently, he is a Professor with the School of Electrical Engineering and Computer Science, SNU. His current research interests include computer-aided design of integrated circuits and combinatorial optimizations.

Youngsoo Shin (SM'05) received the B.S. and M.S. degrees in electronics engineering from Seoul National University, Seoul, Korea, and the Ph.D. degree in electronics engineering from Seoul National University, in 2000.

From 2000 to 2001, he was with the University of Tokyo, Tokyo, Japan, as a Research Associate. From 2001 to 2004, he was with the IBM T. J. Watson Research Center, Yorktown Heights, NY, as a Research Staff Member. He joined the Department of Electrical Engineering, Korea Advanced Institute of Science and Technology, Daejeon, in 2004, where he is currently an Associate Professor. His current research interests include computer-aided design with emphasis on low-power design and design tools, high-level synthesis, sequential synthesis, and structured application-specific integrated circuit.

Dr. Shin received the Best Paper Award at the International Symposium on Quality Electronic Design, in 2005, and was nominated for the Best Paper Award at the same conference, in 2007. He has been a Member of the Technical Program Committee and Organizing Committee of several technical conferences, including the Design Automation Conference, the International Conference on Computer-Aided Design, the International Symposium on Low Power Electronics and Design, the Asia and South Pacific Design Automation Conference, the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, the IEEE Symposium on Very Large Scale Integration, and the International Symposium on Circuits and Systems.