HLS-l: High-Level Synthesis of High Performance Latch-Based Circuits

Seungwhun Paik Insup Shin Youngsoo Shin

Department of Electrical Engineering, KAIST
Daejeon 305-701, Korea
{swpaik, isshin}@dtlab.kaist.ac.kr, youngsoo@ee.kaist.ac.kr

Abstract

An inherent performance gap between custom designs and ASICs is one of the reasons why many designers still start their designs from register transfer level (RTL) description rather than from behavioral description, which can be synthesized to RTL via high-level synthesis (HLS). Sequencing overhead is one of the factors for this performance gap; the choice between latch and flip-flop is not typically taken into account during HLS, even though it affects all the steps of HLS. HLS-*l is a new design framework that employs high-performance latches during scheduling, allocation, and controller synthesis. Its main feature is a new scheduler that is based on a concept of phase step (as opposed to conventional control step), which allows us scheduling in finer granularity, register allocation that resolves the conflict of latch being read and written at the same time, and controller synthesis that exploits dual-edge triggered storage elements to support phase step based scheduling. In experiments on benchmark designs implemented in 1.2 V, 65-*nm* CMOS technology,* HLS-*l reduced latency by 16.6% on average, with 9.5% less circuit area, compared to the designs produced by conventional HLS.*

1 Introduction

It is well-known that there is a large gap of performance between custom designs and ASICs. In particular, ASICs are slower than custom designs in the same technology node by a factor of six or more [3]. Several factors have been identified [3] that cause this large gap of performance: microarchitecture, timing overhead, logic style, cell design and their tuning, layout, and coping with process variation.

High-level synthesis (HLS), which automatically derives architecture (typically in data-path and controller) from behavioral specification of algorithm, inherits the same limitation of ASICs compared to custom designs. In fact, the inability to derive various high-performance architectures is one of the reasons why many designers still start their designs directly from architecture, i.e. register transfer level (RTL), instead of from behavior.

Much effort has been devoted for using advanced microarchitectures in HLS. Optimal selection of clock period [13] has been studied to minimize the waste of timing slack, thus to obtain higher frequencies. Chaining and allowing multicycle operations are popular techniques for tight scheduling of operations, and have been extended in several directions [4, 8, 15]. A concept of complex functional units (FUs), which are essentially a set of chained FUs, has been proposed [4] to map a set of operations into a single FU; this approach, however, requires a library of large set of FUs to cover all chaining possibilities. Multicycle execution of chained operations has been combined [15] with bit-level chaining, which exploits bit-level parallelism of FUs based on bit-by-bit computation such as ripple carry adder, to reduce latency; it comes at the cost of increasing complexity of controller.

Besides microarchitecture, timing overhead from clock and sequential elements plays an important role in performance gap between custom designs and ASICs. ASICs typically have about 4 fanout-of-4 (FO4) delays of clock skew and jitter, which can be reduced to 1 FO4 delay in custom designs [3]. High-speed custom designs typically employ latches with 2 FO4 delays while ASICs exclusively use flip-flops with 3 or 4 FO4 delays [10]. Each combinational block between flip-flops can be identified and its validity of timing constraints can be verified independently from other blocks that allows independent timing optimization, which is why synthesis-based ASICs rely on flip-flops. Level-sensitive sequential circuits, on the other hand, make timing verification very difficult, since combinational blocks are not isolated from each other due to transparent nature of latches. This transparency, however, offers more flexibility: latches allow combinational block to have a delay more than a clock period; clock skew can be tolerated if transparency window, shifted by skew, can still capture the data.

Latch-based HLS, called HLS-l, is proposed to synthesize architectures of high-performance: smaller clock period and smaller number of control steps. The main idea is to schedule operations at both (rising and falling) clock edges as opposed to scheduling at rising (or falling) edge of clock alone in conventional HLS. The use of latch-based registers may increase the total number of registers due to additional restriction on the sharing of variables [18, 19]. Our choice of scheduling at both clock edges alleviates this restriction by generating load-enable signals for registers only at non-transparent period of time. We employ dual-edge triggered storage elements in controller synthesis to generate control signals at both edges of clock. Experiments from 65-nm technology shows that latency is reduced by 16.6% on average while cutting area by 9.5%, compared to the designs produced by conventional HLS. Our main contributions are:

- A complete framework for latch-based HLS, HLS-l, that covers design processes from scheduling, allocation, and controller synthesis (Sections 2 and 3).
- A concept of *phase step* (Section 3.1) that allows scheduling in finer granularity, which helps reduce la-

978-3-9810801-5-5/DATE09 © 2009 EDAA

Authorized licensed use limited to: Korea Advanced Institute of Science and Technology. Downloaded on December 10, 2009 at 00:13 from IEEE Xplore. Restrictions apply.

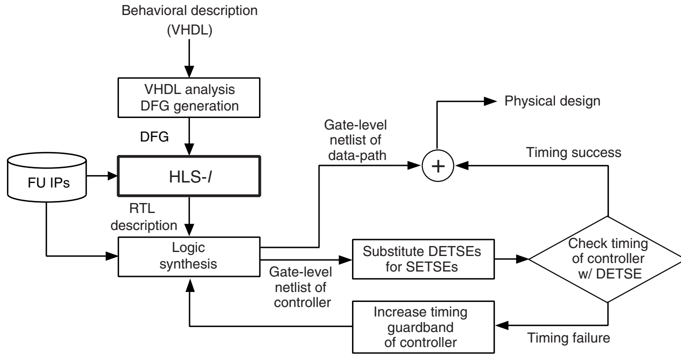

Figure 1. Overall design flow based on HLS-l.

tency, and register allocation with less conflict, which helps reduce the number of registers.

- A method to select period of transparent phase-step to minimize total latency (Section 4).

2 Overall Design Flow Based on HLS-l

The overall design flow based on HLS-l is shown in Figure 1. A behavioral description written in VHDL is first analyzed and is then transformed into a DFG [9]. The DFG will then be an input to HLS-l, which will be described in Section 3. The RTL description of data-path and controller generated by HLS-l then goes through a standard logic synthesis [16] to create an initial gate-level netlist. Commercial IPs [17] of functional units are used during both HLS and logic synthesis.

We use dual-edge triggered storage elements (DETSEs) [14] in our controller to support operation scheduling both at rising- and falling-edges of clock, which will be explained in Section 3. This is done by initially synthesizing a controller[1] in conventional single-edge triggered storage elements (SETSEs), in particular edge-triggered flip-flops, which is followed by substituting DETSEs for SETSEs in the controller component of the netlist. Additional timing check is needed to ensure that the controller satisfies setup- and hold-time constraints both at rising- and falling-edges of clock; if timing is violated, extra timing guardband is intentionally put into the controller, which is followed by its re-synthesis. The final netlist can then be submitted to physical design.

3 High-Level Synthesis of Latch-Based Circuits

3.1 Scheduling

In conventional register file-based architecture, where each register consists of edge-triggered flip-flops, operation scheduling is performed in a unit of control step (c-step). Control step is the period of cycle time led by clock edge, i.e. the interval from one rising edge of clock to the next when rising-edge flip-flops are used or from one falling edge of clock to the next when falling-edge flip-flops are used. During operation scheduling, the execution delay of operation i is given as the number of control steps it takes:

$$d_i = \left\lceil \frac{D_i}{T_{clk}} \right\rceil, \;\; D_i = D_{FU(i)} + D_{margin} \qquad (1)$$

[1] Clock period of controller clock is set to be the minimum of transparent period and non-transparent period of data-path clock.

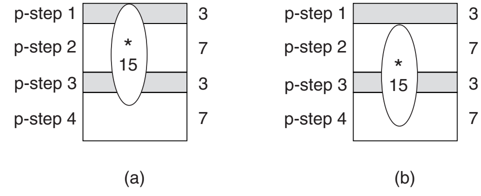

Figure 2. Execution delay of a multiplication using p-steps: (a) delay is 4 when scheduled in the first p-step, and (b) delay is 3 when scheduled in the second p-step.

where T_{clk} is clock period, $D_{FU(i)}$ is the longest path delay of a functional unit (FU) executing i, and D_{margin} is the timing margin to accommodate extra delay through data-path, which includes sequencing overhead (setup time and clock-to-Q delay) of register and multiplexer delay. This cycle time-based scheduling may yield a data-path, where some functional units finish their executions too early leaving excessive slacks, which implies that some control steps are under-utilized.

In our architecture, where each register consists of level-sensitive latches, an operation can be scheduled any time when register is transparent, i.e. when clock is high for positive level-sensitive. Since data-path has to be synchronized with controller (e.g., register data can be steered to target functional unit only when multiplexer select signal arrives from the controller), scheduling operations at arbitrary time points may yield very complicated controller. Hence, for the sake of controller implementation, we assume that operations can be scheduled only at clock edges, i.e. either at rising- or falling-edge. The controller then needs to generate control signals at both rising- and falling-edges of clock, which can be realized using DETSEs as illustrated in Section 3.3.

Our decision of operation scheduling at both edges of clock leads us to introduce a concept of *phase step* (p-step), which is the period of clock being high (transparent phase) or the period of clock being low (non-transparent phase) for positive level-sensitive, as a unit of operation scheduling. The execution delay of operation i, which is expressed using c-steps in (1), should be now re-expressed using p-steps:

$$d'_i = \begin{cases} 2\left\lfloor \frac{D_i}{T_{clk}} \right\rfloor + \left\lceil \frac{r_i}{\xi_i} \right\rceil & \text{if } 0 \le r_i \le \xi_i \\ 2\left\lfloor \frac{D_i}{T_{clk}} \right\rfloor + 2 & \text{if } \xi_i < r_i \end{cases} \qquad (2)$$

where r_i is a residual delay, i.e. the remainder of D_i after it is subtracted by integer multiples of T_{clk}

$$r_i = D_i \bmod T_{clk}, \qquad (3)$$

and ξ_i corresponds to the interval of P_i, a p-step where i is scheduled (we assume that the first p-step is a transparent one):

$$\xi_i = \begin{cases} T_{tr} & \text{if } P_i \text{ is odd} \\ T_{clk} - T_{tr} & \text{otherwise} \end{cases} \qquad (4)$$

where T_{tr} is a period of time when latches are transparent (thus, $T_{clk} - T_{tr}$ is a non-transparent period of time). Note that, when $0 \le r_i \le \xi_i$ in (2), the second term ($\lceil \frac{r_i}{\xi_i} \rceil$) evaluates either to 0 when $r_i = 0$ or to 1 when $0 < r_i \le \xi_i$.

Authorized licensed use limited to: Korea Advanced Institute of Science and Technology. Downloaded on December 10, 2009 at 00:13 from IEEE Xplore. Restrictions apply.

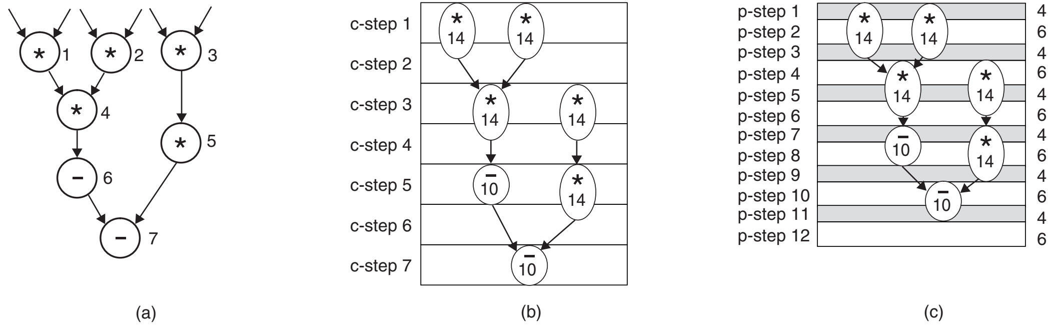

Figure 3. (a) An example DFG, (b) resource-constrained (2 MUL, 1 ALU) list scheduling with control steps, and (c) resource-constrained list scheduling with phase steps.

Example 1 Consider an operation of multiplication shown in Figure 2. We assume that T_{clk} is 10, T_{tr} is 3, and D_i is 15 (D_{margin} is assumed to be 0 for simplicity). The residual delay r_i is 5. When the multiplication is scheduled in the first p-step as shown in Figure 2(a), ξ_i is 3 and thus d'_i is 4 as computed by (2). When the multiplication is scheduled in the second p-step as shown in Figure 2(b), however, it takes 3 p-steps. □

From (2) and Example 1, we see that the execution delay of operation may vary (by one p-step) depending on whether it is scheduled in odd p-step or in even p-step. This raises a question of whether we need to take varying operation delay into account, i.e. whether we need to choose a p-step (either odd or even), during operation scheduling. In ASAP scheduling [5], it can be shown that scheduling operations in as early as possible p-steps (without regarding varying operation delay) always guarantees the smallest latency; similarly, in as late as possible p-steps in ALAP scheduling [5]. The same holds for list scheduling, which we use in this paper: in resource-constrained list scheduling, scheduling a selected operation (an operation whose predecessors have all finished their operations and a resource is left for its execution) in current p-step is no worse (in terms of total latency) than scheduling it in the next p-step (even though the operation delay is smaller when scheduled in the next p-step).

Example 2 Consider an example DFG shown in Figure 3(a). We assume that T_{clk} is 10, and multiplier and ALU take 14 and 10 for their executions, respectively. In c-step based scheduling, the execution delay of multiplication is 2 and that of subtraction is 1. Resource-constrained list scheduling with constraints of two multipliers and one ALU is shown in Figure 3(b); total latency is 7 control steps.

For p-step based scheduling, T_{tr} is assumed to be 4. It can be readily shown that the delay of multiplication is 3 both when scheduled in odd p-step and when scheduled in even p-step; subtraction takes 2 p-steps both from odd and even p-steps. Resource-constrained list scheduling with the same resource constraints is shown in Figure 3(c); total latency is 11 p-steps (or 6 c-steps), which is 1 control step less than Figure 3(b). □

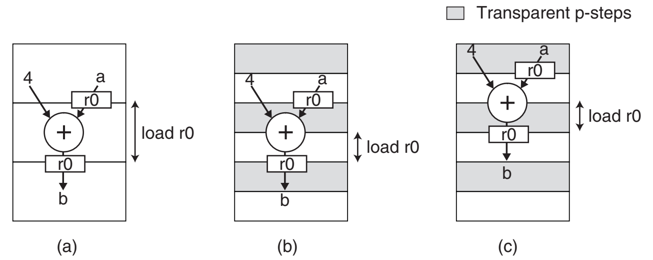

Figure 4. (a) Variables a and b cannot share the same latch-based register in c-step based scheduling; in p-step based scheduling, (b) they can share the register when operation finishes in non-transparent p-step, (c) but cannot share the register when operation finishes in transparent p-step.

3.2 Allocation

3.2.1 Register Allocation

Latch-based register allocation is similar to conventional flip-flop-based one, except that an operation cannot read from and write to the same register. Figure 4(a) shows an example. Assume that the input operand a and output operand b of addition operation are allocated to the same register $r0$. If the short-path delay (contamination delay) of adder is smaller than the period of register being transparent, wrong value is written to the register while it is being read. Note that the lifetimes of a and b do not overlap each other; they can be allocated to the same register if register is conventional flip-flops-based one. Moreover, a variable that is an output operand of an operation cannot share the register with variables that are input operands of other operations finishing at the same p-step.

This problem can be resolved by introducing an extra edge (between a vertex indicating the lifetime of a and a vertex for lifetime of b) to the conventional register conflict graph [18], where each vertex corresponds to the lifetime of a variable and each edge denotes the existence of overlap between lifetimes of vertices connected to it. Register allocation then can be solved by heuristic vertex coloring algorithm [2]. Note that we cannot use the exact left-edge algorithm [6, 11], since the register conflict graph does not belong to interval graph due to extra edges we introduce (recall that we have an extra edge between two vertices even though their lifetimes do not overlap).

In our operation scheduling based on p-steps, we do not always introduce extra edges for the two cases mentioned above,

Authorized licensed use limited to: Korea Advanced Institute of Science and Technology. Downloaded on December 10, 2009 at 00:13 from IEEE Xplore. Restrictions apply.

which allows us more freedom in coloring, thus potentially less number of registers. As shown in Figure 4(b), when an operation is scheduled to finish at non-transparent p-step, a and b can be allocated to the same register (thus, we do not introduce an extra edge); this is possible by generating load-enable signal of register only at the p-step when an operation finishes, i.e. we force the register to be read but not to be written during transparent p-steps by not generating load-enable signal during those p-steps. When an operation finishes at transparent p-step as shown in Figure 4(c), a and b cannot share the register, and thus we introduce the extra edge to the register conflict graph.

The case of Figure 4(c) can be avoided by re-scheduling operations so that they finish at non-transparent p-steps, which is left for future work. When a and b are the same variable, meaning that they have to be allocated to the same register, we simply shift an operation by one p-step above or below (together with its predecessors or successors, if necessary), which may increase the total latency.

3.2.2 Allocation of Functional Units and Connection

The allocation of functional units is also formulated as vertex coloring of a resource conflict graph [5] 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 functional unit, so that they have to be scheduled in different p-steps. Since, the resource conflict graph in our DFG belongs to interval graph, which can be colored optimally, we use the exact left-edge algorithm [6, 11].

The connection allocation, which uses multiplexers to connect registers to functional units and functional units to registers is also formulated as vertex coloring of a connection conflict graph, which is again performed by the left edge algorithm.

3.3 Control Synthesis

Control synthesis is responsible for generating control signals for data-path, such as data-select inputs for multiplexers, function-select inputs for multi-function units (e.g., ALU), load-enable inputs for registers, and so on. This can be accomplished by describing the behavior of a controller as a state transition graph (STG), which is followed by FSM synthesis that outputs a hard-wired sequential circuit. In conventional data-path that is based on edge-triggered registers, control signals are generated only at clock edges (either rising or falling); thus, the number of states in STG is typically equal to the number of control steps [5]. In our data-path, on the other hand, control signals have to be generated at both clock edges, meaning that the number of states gets doubled, which in turn implies that we need one more storage element in our controller[2].

Since our controller has to generate control signals at both edges of clock that is used in data-path, we either need to use the clock (for controller) of twice the frequency of data-path clock or use DETSEs (for storage elements of controller) with the same data-path clock. The first approach is only viable when duty cycle of data-path clock is 50%; otherwise, one of the edges of data-path clock cannot be synchronized with controller clock. We adjust duty cycle of data-path clock as will be shown in Section 4, since it affects the total latency obtained by scheduling; thus, we take the second approach in this paper.

[2] Note that the number of storage elements required to implement a given STG is equal to $\lceil log_2(\text{\# of states})\rceil$.

Table 1. Comparison of D flip-flop and Latch-mux DETSE in 65-nm technology, V_{dd}=1.2 V, T=125°C

	SETSE (D flip-flop)	DETSE Rising clock edge	DETSE Falling clock edge
Setup time (ps)	51	45	53
Hold time (ps)	-29	-24	-25
Clock-to-Q delay (ps)	148	165	164

There are several varying implementations of DETSE [14]; we implemented latch-mux structure [12] for its relatively small sequencing overhead and simple architecture; it was sized so that it has similar timing parameters as SETSE (see Table 1), which allows us to substitute DETSEs for SETSEs in the controller component of the netlist (see Section 2 and Figure 1) without causing too much timing violation.

4 Selecting Period of Transparent P-Step

In order to minimize the latency of operation scheduling based on p-steps, the period of time when latches are transparent (T_{tr}) should be carefully selected: too small value of T_{tr} may yield large delay of FUs (e.g., 4 p-steps for the shaded multipliers shown in Figure 5(a)); too large value of T_{tr} may also yield large delay of FUs (e.g., 4 p-steps for the shaded multiplier shown in Figure 5(b))

We first perform a scheduling with arbitrary T_{tr} (e.g., $\frac{T_{clk}}{2}$) and find a set of critical paths that contribute to total latency. Then we examine the operations on the critical paths and assign cost to each operation type based on (5). Each operation type has a range of T_{tr} that minimizes its operation delay. So we try to find T_{tr} that minimizes the operation delay for all operation types, and favor the operation type with higher cost if there is any conflict. The cost of operation type k is defined as follows:

$$cost_k = w_k \cdot occur_k. \quad (5)$$

where w_k is the weight for each operation type and $occur_k$ is the number of occurrence of the operation type on the critical paths. We use 2 for w_k if $0 < r_k \leq \frac{T_{clk}}{2}$ and 1 for w_k if $r_k > \frac{T_{clk}}{2}$ because the former has potential to minimize the operation delay regardless of scheduled p-step whereas the latter has potential to minimize the operation delay only when the operation is scheduled either in odd or even p-step. (When $r_k = 0$, the operation delay is fixed regardless of T_{tr}, and thus is not considered.)

Example 3 Consider a scheduled DFG in Figure 5(c). We assume that T_{clk} is 10 and fast multiplier, slow multiplier and ALU takes 13, 17 and 10 for their execution, respectively. There are 3 operations, one per each operation type, in the critical path, and thus the cost of fast multiplication is 2 and that of other operation types are 1. In this case, there is no conflict in finding T_{tr} to minimize the operation delay for both type of multiplications, and the candidate values of T_{tr} are 3 and 7.[3]

[3] To minimize the operation delay, T_{tr} should satisfy $3 \leq T_{tr} \leq 7$ for fast multiplier and either $T_{tr} \geq 7$ (P_i is odd) or $T_{tr} \leq 3$ (P_i is even) for slow multiplier. Addition is not considered as its $r_k = 0$.

Authorized licensed use limited to: Korea Advanced Institute of Science and Technology. Downloaded on December 10, 2009 at 00:13 from IEEE Xplore. Restrictions apply.

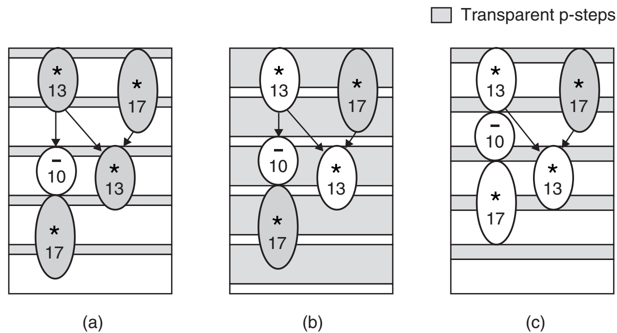

Figure 5. Latency depends on the selection of T_{tr}. Either (a) too small or (b) too large T_{tr} may increase the latency from (c) the minimum latency.

We try scheduling with both values of T_{tr} and select 3, which gives the minimum latency as slow multiplier on the critical path is scheduled in even p-step, over 7. □

5 Experimental Results

We took a set of behavioral benchmark designs [1] and performed experiments to assess the effectiveness of HLS-l in reducing the number of control steps, latency, and area. HLS-l was implemented in C under Centos 5.0, and each design, once transformed into RTL as shown in Figure 1, was synthesized [16] in commercial 1.2 V, 65-nm bulk CMOS technology. For scheduling, clock period (T_{clk}) was set to 8.2 ns, which is the sum of ALU delay (6.85 ns) and timing margin (1.35 ns) to accommodate extra delay from sequencing overhead of register and multiplexer delay, and transparent period of clock was set to 2.5 ns; once gate-level netlist is obtained (see Figure 1), clock period was adjusted, which we report in Table 2.

5.1 Effectiveness of HLS-l on Latency

We compared results from HLS-l and conventional high-level synthesis, which we simply denote as HLS, under the same resource constraints. List scheduling [7] was used for operation scheduling in both approaches, with difference of HLS-l relying on p-step while HLS using c-step. The results are summarized in Table 2. The second column is resource constraint, which is expressed as the number of multipliers and ALUs. The results produced by HLS are shown in the next three columns, where T_{clk} is clock period and L is latency in terms of control steps; the results from HLS-l are shown in the following three columns; the next three columns show the savings obtained by HLS-l over HLS.

The latency (L) is reduced by 3.8 control steps on average, which, combined with clock period (T_{clk}) that is reduced by 0.2 ns due to less sequencing overhead, yields reduction in total latency ($T_{clk} \cdot L$) by 16.6% on average. The latency improvement is more significant when the number of occurrence of multiplications on a set of critical paths is high because T_{tr} has been selected to reduce the delay of multiplication. Example `wavelet` with resource constraint of one ALU and one multiplier obtains the largest reduction on latency as its critical path consists of multiplications alone. Example `elliptic` benefits the least because its critical path is dominated by additions and subtractions (26 in total) rather than by multiplications (only 1), when resource constraints is one ALU and one multiplier.

5.2 Effectiveness of HLS-l on Area

The area of designs (the sum of the areas of all the cells after logic synthesis) produced by HLS and HLS-l is compared in Figure 6. The area of designs from HLS-l is reduced by 9.5% on average, where main contributor is register that uses latches occupying 24.6% less area than flip-flops. Furthermore, the number of registers, after register allocation illustrated in Section 3.2.1, in HLS-l are increased (recall that we introduce extra edges in register conflict graph, which translates into potentially more registers) but only uses 1.1 more registers on average. Area is more reduced in the designs with larger proportion of area from registers, such as `wavelet` and `wdf7` as shown in Figure 6.

6 Conclusion

We have presented a method of high-level synthesis of latch-based circuits, focusing on the primary problem of minimizing latency by scheduling operations at both edges of clock. The latency was reduced by 3.8 control steps on average of several benchmark designs, which, combined with reduced clock period, translates into 16.6% reduction in latency. The area was reduced by 9.5% on average, mainly due to smaller area of latches. The choice of scheduling operations at both edges of clock helps register allocation, since less conflicts are introduced by generating load-enable signals only at non-transparent period of time. The controller that support the proposed scheduling can be implemented by using dual-edge triggered storage elements.

Rescheduling operations during register allocation can further reduce the number of registers; varying operation delay can possibly be used as a priority function in list scheduling, which are left for future work.

Acknowldegment

This work was supported by the Korea Science and Engineering Foundation (KOSEF) grant funded by the Korea government (MEST) (NO. R01-2007-000-20891-0).

References

[1] High level synthesis benchmark. http://bears.ece.ucsb.edu/cad.
[2] D. Brelaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22(4):251–256, Apr. 1979.
[3] D. Chinnery and K. Keutzer. *Closing the Gap Between ASIC & Custom*. Kluwer Academic Publishers, 2002.
[4] M. Corazao et al. Performance optimization using template mapping for datapath-intensive high-level synthesis. *IEEE Trans. on Computer-Aided Design*, 15(8):877–888, Aug. 1996.
[5] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, Inc., 1994.
[6] A. Hashimoto and J. Stevens. Wire routing by optimizing channel assignment within large apertures. In *Proc. Design Automation Workshop*, pages 155–169, June 1971.
[7] T. C. Hu. Parallel sequencing and assembly line problems. *Operation Research*, 9(6):841–848, Dec. 1961.
[8] C.-T. Hwang, J.-H. Lee, and Y.-C. Hsu. A formal approach to the scheduling problem in high level synthesis. *IEEE Trans. on Computer-Aided Design*, 10(4):464–475, Apr. 1991.
[9] J. Jeon, Y. Ahn, and K. Choi. CDFG toolkit user's guide, Aug. 2002. Technical Report No. SNU-EE-TR-2002-8.

Authorized licensed use limited to: Korea Advanced Institute of Science and Technology. Downloaded on December 10, 2009 at 00:13 from IEEE Xplore. Restrictions apply.

Table 2. Comparison of results produced by HLS and our HLS-l. T_{clk} is clock period, L is latency in terms of control steps, and $T_{clk} \cdot L$ corresponds to latency in ns.

Benchmark	Resource constraint (MUL, ALU)	HLS			HLS-l			Savings		
		T_{clk} (ns)	L (c-step)	$T_{clk} \cdot L$ (ns)	T_{clk} (ns)	L (c-step)	$T_{clk} \cdot L$ (ns)	T_{clk} (ns)	L (c-step)	$T_{clk} \cdot L$ (%)
IIR7	(1,1)	7.9	31	246	7.7	24	186	0.2	7	24.4
	(2,1)	8.0	20	159	8.0	18	143	0.0	2	10.1
	(2,2)	8.1	18	146	7.9	16	126	0.2	2	13.6
FIR11	(1,1)	8.0	23	183	7.8	18	140	0.2	5	23.5
	(2,1)	7.6	13	99	7.6	12	91	0.1	1	8.5
FIR7	(1,1)	7.9	16	126	7.8	14	109	0.1	2	13.3
	(2,1)	7.9	11	86	7.9	10	79	0.0	1	9.0
	(2,2)	7.8	9	70	7.7	8	61	0.2	1	13.0
ELLIPTIC	(1,1)	8.2	28	228	8.1	28	226	0.1	0	1.2
	(1,2)	8.2	22	180	8.1	19	153	0.1	3	15.0
LATTICE	(1,1)	8.0	20	159	7.7	16	123	0.3	4	22.8
	(2,1)	7.8	12	94	7.8	11	85	0.1	1	9.3
VOLTERRA	(1,1)	7.9	36	285	7.6	28	212	0.3	8	25.6
	(2,1)	8.1	20	162	7.7	16	124	0.3	4	23.4
	(3,1)	8.0	15	120	7.8	14	109	0.3	1	9.6
WAVELET	(1,1)	8.0	57	457	7.4	43	319	0.6	14	30.1
	(2,2)	8.1	30	244	7.8	23	178	0.4	7	26.7
	(3,2)	8.0	21	168	8.0	17	136	0.0	4	19.2
WDF7	(1,1)	8.2	39	318	8.0	29	233	0.1	10	26.6
	(2,2)	8.2	20	163	8.1	17	137	0.1	3	15.8
	(3,2)	8.1	18	138	7.9	16	127	0.2	2	8.4
AR	(1,1)	7.9	36	285	7.8	26	203	0.1	10	28.7
	(2,1)	7.9	21	166	7.7	18	138	0.3	3	17.0
	(2,2)	7.8	19	149	7.7	16	124	0.1	3	16.8
DIFFEQ	(1,1)	7.8	15	117	7.7	13	100	0.1	2	14.3
	(2,1)	7.9	11	86	7.8	10	78	0.1	1	10.1
	(2,2)	7.8	9	70	7.7	8	62	0.0	1	11.5
Average								**0.2**	**3.8**	**16.6**

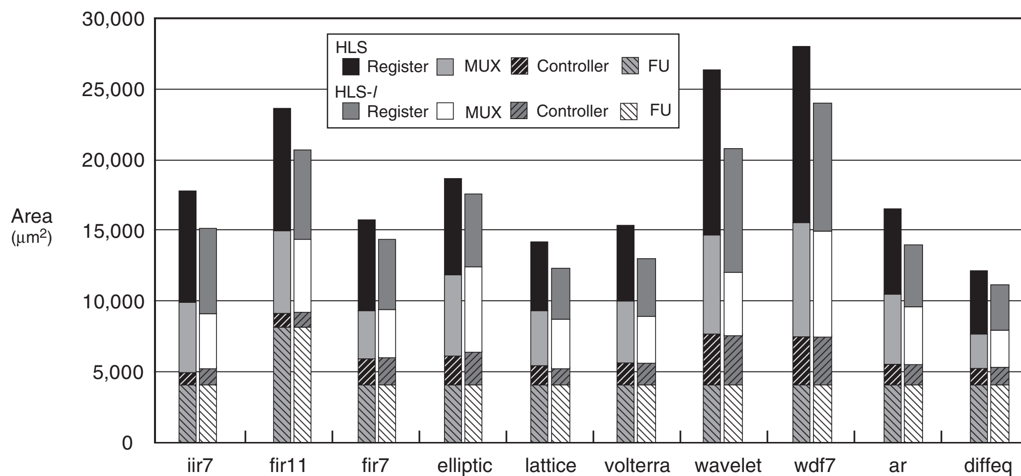

Figure 6. Comparison of the area of designs produced by HLS (left-hand bars) and HLS-l (right-hand bars) for resource constraint of 1 multiplier and 1 ALU.

[10] F. Klass et al. A new family of semidynamic and dynamic flip-flops with embedded logic for high-performance processors. *IEEE Journal of Solid-State Circuits*, 34(5):712–716, May 1999.

[11] F. Kurdahi and A. Parker. REAL: a program for register allocation. In *Proc. Design Automation Conf.*, pages 210–215, June 1987.

[12] R. Llopis and M. Sachdev. Low power, testable dual edge triggered flip-flops. In *Proc. Int. Symp. on Low Power Electronics and Design*, pages 341–345, Aug. 1996.

[13] A. Naseer, M. Balakrishnan, and A. Kumar. Optimal clock period for synthesized data paths. In *Proc. Int. Conf. on VLSI Design*, pages 134–139, Jan. 1997.

[14] V. G. Oklobdzija, V. M. Stojanovic, D. M. Markovic, and N. M. Nedovic. *Digital System Clocking: High-Performance and Low-Power Aspects*. John Wiley & Sons, Inc., 2003.

[15] S. Park and K. Choi. Performance-driven high-level synthesis with bit-level chaining and clock selection. *IEEE Trans. on Computer-Aided Design*, 20(2):199–212, Feb. 2001.

[16] Synopsys. Design Compiler User Guide, Mar. 2007.

[17] Synopsys. DesignWare IP Family Reference Guide, Dec. 2007.

[18] T. Wu and Y. Lin. Storage optimization by replacing some flip-flops with latches. In *Proc. European Design Automation Conf.*, pages 296–301, Sept. 1996.

[19] W. Yang, I. Park, and C. Kyung. Low-power high-level synthesis using latches. In *Proc. Asia South Pacific Design Automation Conf.*, pages 462–465, Jan. 2001.

Authorized licensed use limited to: Korea Advanced Institute of Science and Technology. Downloaded on December 10, 2009 at 00:13 from IEEE Xplore. Restrictions apply.