Using Precomputation in Architecture and Logic Resynthesis*

Soha Hassoun
Tufts University
Medford, MA 02155
soha@eecs.tufts.edu

Carl Ebeling
University of Washington
Seattle, WA 98195-2350
ebeling@cs.washington.edu

Abstract – Although tremendous advances have been accomplished in logic synthesis in the past two decades, in some cases logic synthesis still cannot attain the improvements possible by clever designers. This, in part, is a result of logic synthesis not optimizing across register boundaries. In this paper we focus on precomputation as a resynthesis technique capable of resynthesizing across register boundaries. By using precomputation, a critical signal is computed earlier in time, thus allowing it to be combinationally optimized with logic from previous pipeline stages. Precomputation automatically discovers some standard circuit transformations like bypassing and lookahead. In addition, precomputation can be used in conjunction with combinational logic synthesis to resynthesize a circuit to obtain better performance.

This paper contributes to the understanding and development of precomputation. First, it provides a synthesis algorithm for precomputation. Second, it demonstrates how precomputation can be used to improve sequential logic resynthesis and reports the results of applying a heuristic to a subset of the MCNC benchmarks. Third, it illustrates how precomputation generalizes and unifies bypassing and lookahead – two important and practical architectural transformations often used in processor design and high-level synthesis of DSP processors. Finally, it clarifies the relationships among precomputation, retiming, and implicit retiming.

1 Introduction

In spite of the tremendous success of logic synthesis over the past two decades, there are still situations where logic synthesis cannot attain the required performance. This is, in part, a result of logic synthesis not optimizing across register boundaries. Although some sequential synthesis techniques have been developed to address this problem [8, 3, 2, 1], better sequential synthesis techniques are needed to achieve performance comparable to that possible using complex transformations typically employed by clever designers.

This paper focuses on precomputation as a general sequential synthesis technique. Precomputation is a technique whereby a critical signal is computed earlier in the computation. This has the effect of shortening the critical path by changing when the critical signal is computed. This in turn exposes logic on the critical path to combinational logic optimizations. It is important to note that precomputation can be performed both at the logic level and at the RTL/microarchitecture level. By performing transformations at the RTL level, optimizations can be applied that would otherwise be obscured after the circuit has been flattened to the logic level. Precomputation automatically discovers some standard (micro)architectural transformations like bypassing and lookahead. In addition, precomputation can be used in conjunction with combinational logic synthesis to resynthesize a circuit to obtain better performance.

The notion of precomputation was recently formalized via architectural retiming [5]. Architectural retiming is a technique for pipelining a latency-constrained path while preserving a circuit's latency and functionality. Latency constraints arise frequently in practice, and they are due to either cyclic dependencies or explicit performance constraints. Architectural retiming is comprised of two steps. First, a register is added to the latency-constrained path. Second, the circuit is changed to absorb the increased latency caused by the additional register. To preserve the circuit's latency and functionality, we use the negative register concept. A normal register performs a shift forward in time, while a negative register performs a shift backward in time. A negative register/normal register pair reduces to a wire. Resulting performance improvements are due to increasing the number of clock cycles available for the computation, which allows for a smaller clock.

Two implementations of the negative register are possible: precomputation and prediction. In precomputation, the negative register is synthesized as a function that precomputes the input to the added pipeline register using signals from the previous pipeline stage. In prediction, the negative register's output is predicted one clock cycle before the arrival of its input.

We focus in this paper on the precomputation implementation of architectural retiming. While earlier work formalized the notion of precomputation, we present here practical necessary details, and we demonstrate the benefits of applying precomputation both at the sequential logic level and at the architectural level.

We begin by describing our circuit model and reviewing the definition of precomputation-based architectural retiming. We then discuss using precomputation in sequential logic resynthesis. We describe a heuristic to combine precomputation with resynthesis. We then present an example and experiments to evaluate our heuristic. Next, we describe using precomputation in architectural synthesis and describe how precomputation synthesizes two architectural transformations: bypassing and lookahead. We illustrate this using an example. We conclude by reviewing related work and summarizing the contributions of this paper.

*This work was supported by the ARPA/CSTO Microsystems Program under an ONR-monitored contract (DAAH04-94-G-0272)

2 Background

2.1 Model

All registers in a circuit are edge-triggered registers clocked by the same clock. Time is measured in terms of clock cycles and the notation x^t denotes the value of a signal x during clock cycle t, where t is an integer clock cycle. Values of signals are referenced after all signals have stabilized and it is assumed that the clock period is sufficiently long for this to happen. A register delays its input signal y by one cycle. Thus, $z^{t+1} = y^t$, where z is the register's output.

Each function f in the circuit is a single-output function of N input variables (or signals) $x_0, x_1, \ldots, x_{N-1}$ computing a variable y. In a specific cycle t, the variable y is assigned the value computed by the function f using specific values for $x_0, x_1, \ldots, x_{N-1}$, that is, $y^t = f(x_0^t, x_1^t, \ldots, x_{N-1}^t)$.

The set of fan-in signals of a function f is the set of f's input variables. The set of combinational transitive fan-in signals of a function f, denoted by CT_f, is defined recursively as follows. For each fan-in variable x_i of f, if x_i is a primary input, then $x_i \in CT_f$. If x_i is an output of a register, then $x_i \in CT_f$. Otherwise, g_i computes x_i, i.e. $x_i = g_i()$ and CT_{g_i} is in CT_f. The set of sequential transitive fan-in signals across one register boundary, ST_f^1, is defined the same as CT_f except that the input path is allowed to cross one register.

2.2 Precomputation: Definition

We review the definition of precomputation introduced in [5]. Let us assume the input to the negative register is the variable y computed by the function $f(x_0, x_1, \ldots, x_{N-1})$. From the definition of the negative register,

$$z^t = y^{t+1} = f(x_0^{t+1}, x_1^{t+1}, \ldots, x_{N-1}^{t+1}).$$

The values x_i^{t+1} for $0 \leq i \leq N-1$ are, of course, not available at time t. If the value of $x_i, 0 \leq i \leq N-1$, at time $t+1$ can be computed directly from values available at time t, then we can recompute the function f as a function f' of values at time t. That is, $f(\ldots, x_i^{t+1}, \ldots) = f'(\ldots, g_i(\ldots, y_{i_j}^t, \ldots), \ldots)$, where $x_i^{t+1} = g_i(\ldots, y_{i_j}^t, \ldots)$ for $0 \leq i \leq N-1$ and $0 \leq j \leq M-1$, where M is the cardinality of $ST_{x_i}^1$. The function f' is then derived from the definition of the original circuit by the recursive collapsing (or flattening) of each x_i into the variables in $ST_{x_i}^1$ or the primary inputs in CT_{x_i} for which f is a don't care value. The recursive collapsing across a register boundary consists of substituting the input variable to the register for its output variable.

Precomputation can be done only if there is sufficient information in the circuit to allow precomputing the value one cycle ahead of time. Precomputation is possible if none of the signals in the combinational transitive fan-in set are primary inputs.

3 Precomputation in Logic Resynthesis

The idea behind precomputation is simple. The negative register is implemented as a function f' that computes the output of the negative register one cycle before the input actually arrives – relying on signals available from the previous pipelined stage. The function f' includes more than one clock cycles' worth of computation along the latency-constrained path. We illustrate this with an example.

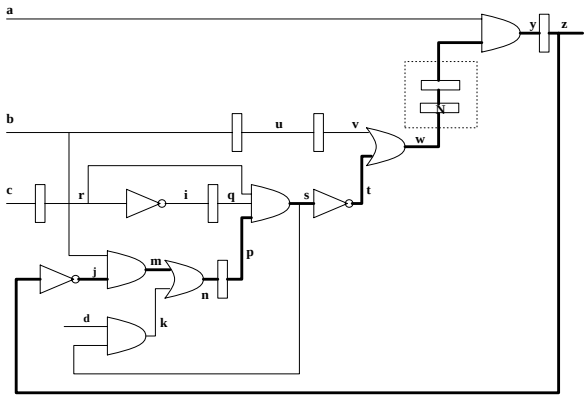

Figure 1: Example circuit. The edges along the critical cycle are drawn in bold. The negative/pipeline register pair (in the dotted box) is added along the critical cycle.

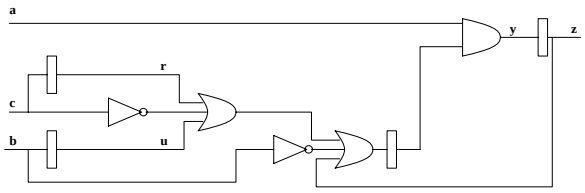

Figure 2: Example circuit after precomputing at w.

Figure 1 applies precomputation to an example circuit. A delay of one time unit is assumed for each gate, and zero delay is assumed for each register. The clock period is four time units. The critical cycle in the circuit consists of the vertices driving signals z, j, m, n, p, s, t, w, and y. The lower bound on the clock period due to the critical cycle is 3.5. Even when using level-clocked latches and skewing the clock [4], retiming cannot reduce the clock period below 3.5. Precomputing and then optimizing w:

$$\begin{aligned} w^{t+1} &= v^{t+1} + \neg(r^{t+1} \cdot q^{t+1} \cdot p^{t+1}) \\ &= u + \neg(c \cdot \neg r \cdot (b \cdot \neg z + d \cdot p \cdot q \cdot r)) \\ &= u + \neg(c \cdot \neg r \cdot b \cdot \neg z) \\ &= u + \neg c + r + \neg b + z \end{aligned}$$

The final circuit is shown in Figure 2. The delay of the critical cycle is now two time units; it is no longer a performance bottleneck.

We now describe our heuristic. In a sequential circuit, several critical paths may limit the performance of the circuit. Each critical path, which may be along a latency-constrained path (a critical cycle or a path from a primary input to a primary output), must be modified to improve performance. To use precomputation in sequential resynthesis, we thus precompute along critical paths.

The procedure for using precomputation in sequential logic resynthesis is comprised of four steps. The first step is identification of points in the circuit where negative/pipeline register pairs must be added. Second is synthesis of precomputation functions representing the added negative registers. Third is the addition of the precomputation functions to the original circuit and the removal of the logic rendered useless

by these new functions. A resynthesis and retiming step completes the precomputation optimization procedure. We describe each of these steps, present an example, and then evaluate the heuristic for integrating precomputation in sequential resynthesis.

3.1 Identifying Locations for Added Register Pairs

Precomputation duplicates logic in the circuit and increases the number of registers along the critical paths. We must therefore select a minimum number of points to precompute while trying to maximize performance gain. The following factors should be considered when deciding where to add the negative/pipeline register pair.

- Precomputation exposes two adjacent pipeline stages for combinational optimizations. At least one of these stages should have a critical path.
- For precomputation to be effective, it must be applied along all critical single-register cycles.
- If a node, v, along a critical combinational path is precomputed, there is no need to precompute any nodes that drive v through a combinational path: by precomputing v, a larger combinational cone has been exposed for combinational resynthesis.
- If precomputing a node v affects more critical paths than precomputing a node u, then v should be preferred for precomputation over u. Thus, if node v drives through combinational paths more registers and primary outputs, which have a small slack value, than does node u, then v is chosen first for precomputation.
- Finally, to reduce logic duplication, we precompute all outputs of a node v, rather than precomputing along one of the output edges.

Taking the preceding factors into consideration, we develop a heuristic for choosing the nodes to precompute. Our heuristic consists of first determining which nodes, if any, benefit from precomputation, and then sorting the nodes by (a) decreasing benefit and (b) decreasing arrival times.

Every edge driven by a node with slack value less than *slack_limit*, a value provided by the user, is considered critical. The slack of a node is the difference between the required and arrival times of the signal at the output of that node. We refer to the critical nodes and their fan out edges as the critical graph. We refer to a single-register cycle with all critical nodes as a critical single-register cycle.

A value, *benefit*, is assigned to each node in the circuit. A benefit of 1 is initially assigned to all nodes with a slack value less than *slack_limit*. The benefit for v is further increased by 1 for each critical single-register cycle that passes through it. Finally, the benefit for v is increased by the number of registers and primary outputs v drives through a combinational path in the critical graph. The algorithm for computing the benefit values is presented next. We assume that the slack values are known. We pre-calculate a matrix that has the shortest weight between all pairs of nodes in the circuit, which in turn allows us to identify single-register cycles. The procedure "compute_PIR(v)" counts the number of critical register inputs or primary outputs that are driven by v through a combinational path. The heuristic ultimately favors nodes on single-register cycles that fan out through combinational paths to many registers and primary outputs.

Algorithm: Computing Benefit Values

```
for each vertex v in V[G]
benefit[v] := 0
for each vertex v in V[G]
    if slack(v) < slack_limit,
        benefit[v] := benefit[v] + 1
    for each critical single-register cycle passing through v,
        benefit[v] := benefit[v] + 1
    benefit[v] := benefit[v] + compute_PIR(v)
```

Next, the nodes are sorted by decreasing benefit, and then by decreasing arrival times. The heuristic thus favors nodes that are further forward in a combinational stage.

3.2 Synthesizing the Precomputation Function

We now present an algorithm for synthesizing the negative register as a precomputation function. We assume that the negative/pipeline register pair is added along an edge, e_c.

A graph, PG, representing the precomputation function f', is synthesized using an algorithm based on the derivation in Section 2.2. Vertices and edges in the original circuit graph are replicated while performing a depth-first, backward traversal in the circuit. The traversal begins at e_c and ends when it reaches the inputs of the previous pipeline stage. Thus, before traversing a register boundary, the algorithm traverses the vertices in the same combinational stage as e_c. That stage is referred to as the current stage, or stage 1. Once the algorithm traverses a register boundary, the traversal is in stage 2, or the previous pipeline stage. The algorithm for building a precomputation graph, PG, is presented in Procedures 1 and 2 below.

Algorithm: Extracting The Precomputation Function

Procedure 1: Build Precomputation Logic (edge e_c)

```
for each vertex u in V[G]
    Color[u, 1] := white
    Color[u, 2] := white
Map := {}
create new primary output vertex vpo in V[PG]
return Build Precomputation Logic Visit(source(e_c), vpo, 1)
```

Procedure 2: Build Precomputation Logic Visit (v, $target$, $stage$)

```
Color[v, stage] := black
if v is a primary input
    if (stage = 1), then no precomputation is possible
        return Failure
    else,
        create new primary input vertex vpi in V[PG]
        Map := Map ∪ ((v, stage) ⇒ vpi)
        create new edge in E[PG] from vpi to target
if v is a register
    if (stage = 1), then for all in-edges e of v
        if (Color[source(e), 2] = white), then
            Build Precomputation Logic Visit(source(e),
            target, stage+1)
        else
            create new edge in E[PG] from
            Map(source(e), 2) to target
    else, (v is a register and stage equals 2)
        create new primary input vertex vpi in V[PG]
        Map := Map ∪ ((v, stage) ⇒ vpi)
        create new edge in E[PG] from vpi to target
if v is combinational logic
    create new combinational logic vertex vf in V[PG]
    create new edge in E[PG] from vf to target
    Map := Map ∪ ((v, stage) ⇒ vf)
    forall in-edges e of v
        if (Color[source(e), stage] = white), then
```

```
        Build Precomputation Logic Visit(source(e), vf, stage)
    else
        if (source(e) is a register and stage = 1), then
            create new edge in E[PG] from
            Map(source(in-edge(source(e))), stage+1) to vf
        else
            create new edge in E[PG]
            from Map(source(e), stage) to vf
```

There are several important points to make about the algorithm.

- A vertex can be visited as part of the traversal of the current pipeline stage, the previous pipeline stage, or both. If both are traversed, the vertex is replicated twice in PG. To keep track of the traversal in each stage, a color variable is associated with each vertex for each stage traversal.
- The variable Map is a set of mappings from a vertex in G and a stage number to a vertex in PG. Map is initialized to be the empty set. It is updated whenever a new vertex in PG is created in Procedure 2.
- Procedure 2, Build Precomputation Logic Visit, returns failure if precomputation is not possible.

3.3 Adding Precomputation Logic and Removing Redundant Logic

Before adding the precomputation function back to the circuit, logic in the original graph that duplicates each precomputation function is removed. The output edges of the precomputed node, v, are first removed. Then, a backward traversal occurs in a depth-first fashion, beginning with v. While traversing the circuit backwards, a vertex is removed if all its output edges are removed, and an edge is removed if all its target vertices are removed. The removal process stops at the connection points where the precomputation function will be reconnected in the circuit.

3.4 Resynthesizing and Retiming

Once the precomputation function and the added pipeline register are merged again with the existing circuit, we apply combinational optimizations to all logic blocks affected by the precomputations to further optimize the circuit's area and speed. Retiming is then performed to optimally place the registers in the circuit. The resulting circuit can be further optimized by using the precomputation procedure again, if needed.

3.5 Resynthesis Example – Revisited

We apply the preceding procedure to the circuit in Figure 1 – first with a *slack_limit* of 0, and then with a *slack_limit* of 1.

Assume first a *slack_limit* of 0. Nodes computing signals p, s, t, w, and y comprise the critical path; they have zero slack. Their benefit is thus increased by one. Their benefit is increased by an additional one because they drive y, a register input, through a combinational path in the critical graph.

The nodes are first sorted by benefit values and then by arrival times – recommending applying precomputation in the following order: y, w, t, s, p. Precomputing y is not possible because a is a primary input. The heuristic thus chooses to precompute w, which was shown earlier. Vertices

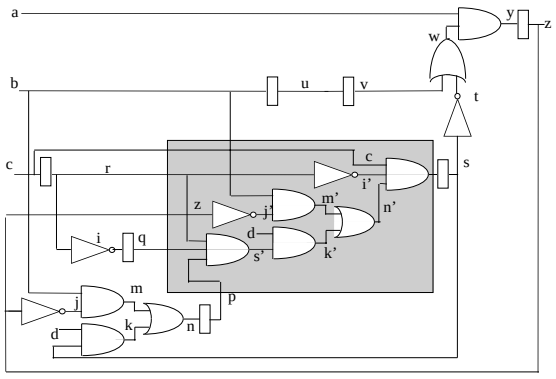

Figure 3: Precomputing at node s.

that drive w, v, and t are removed from the original function. Before adding the optimized precomputation function back to the circuit, vertices driving p, n, m, j, k, s, q, and i are removed from the circuit, because connection points p and q are don't cares when optimizing the precomputation function. The final circuit was shown in Figure 2.

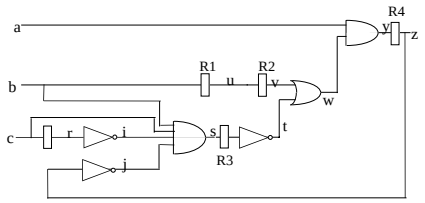

Figure 4: Final circuit when precomputing at node s.

Consider a slack limit of 1. The single-register cycle involving p is part of the critical graph. Moreover, signals s and p affect through combinational paths two register inputs. The benefit values for the nodes computing s and p are thus larger than those for the nodes associated with signals t and w. When sorting the nodes by benefit values first and then by arrival times, s is chosen for precomputation.

Precomputing s, the single-register cycle involving p is partially unrolled, as it was when precomputing w and as shown in Figure 3. Signal s is precomputed as follows:

$$\begin{aligned} s^{t+1} &= r^{t+1} \cdot q^{t+1} \cdot p^{t+1} \\ &= c \cdot \neg r \cdot (b \cdot \neg z + d \cdot p \cdot q \cdot r) \\ &= c \cdot \neg r \cdot b \cdot \neg z \end{aligned}$$

Once the precomputation function is added to the original circuit and the redundant circuitry is removed, we obtain the circuit shown in Figure 4. The critical cycle has a delay of four time units, with two registers. With retiming, the achievable clock period is 2.

The circuits in Figure 4 and in Figure 2 are equivalent. This is evident by retiming locally and moving registers R2 and R3 to the output of the node computing w, and then re-optimizing the logic driving w. By precomputing w, however, fewer iterations of retiming and resynthesis were required to produce the optimal circuit.

circuit	# ins	# outs	# nodes	# regs	T_{clk}	area
mm4a	7	4	35	12	28.50	410176
mm9a	12	9	720	27	46.51	783696
mult16a	17	1	147	16	46.03	406928
mult16b	17	1	218	30	7.04	410176
s1196	14	14	529	18	18.66	815248
s208.1	10	1	104	8	10.66	138272
s27	4	1	10	3	7.44	30160
s298	3	6	119	14	11.14	223648
s344	9	11	160	15	13.22	234320
s349	9	11	161	15	13.22	238496
s382	3	6	158	21	14.35	287680
s386	7	7	159	6	15.83	262624
s400	3	6	162	21	15.23	309488
s420.1	17	1	218	16	12.86	294176
s444	3	6	181	21	15.94	322944
s526	3	6	193	21	12.67	393936
s641	20	14	379	19	20.43	341504
s713	20	13	393	19	20.53	348464

Table 1: Circuits from the MCNC benchmarks used to evaluate precomputation. We list the number of primary inputs; primary outputs; logic functions (.names in blif format); registers; the clock period, T_{clk}; and area of the initial circuit.

3.6 Experiments

To show that precomputation is a viable option in sequential logic resynthesis, we implemented our heuristic and compared resynthesis that employs precomputation with one that does not.

Our set of circuits, summarized in Table 1, was selected from the MCNC sequential multilevel circuits. We used only a subset of the 40 circuits, eliminating large circuits that demanded large memory or long CPU time in SIS [11], a logic optimization program we used to resynthesize, map, and retime the circuits.

Because our intention is to use precomputation in resynthesis, i.e. not during initial optimization, the circuit given to our tool is an optimized circuit. It was obtained by first running the SIS script script.rugged, recommended for area minimization. This script was then followed by script.delay[1], which synthesizes a circuit for a final implementation that is optimal with respect to speed. The circuit was then remapped and retimed to minimize the clock period while also minimizing the register area. The optimized circuit obtained via this one step resynthesis-retiming procedure is reported in Table 2, column a. We used the lib2.genlib and lib2_latch.genlib libraries in SIS for mapping the initial circuit and reporting results.

The experiment consisted of applying up to ten precomputation steps, if possible, to each optimized circuit that was obtained via one step of synthesis and retiming. Next, we resynthesized the resulting 10 circuits using script.rugged and script.delay and retimed them for optimal performance. The circuit with the smallest clock period is reported in column c in Table 2.

Examining Table 2, we note that one resynthesis-retiming step (column a) reduces in most cases both the clock period and area. Applying an additional step of resynthesis and retiming (column b) further reduces the clock period. This is possible because the previous retiming step changed the boundaries of the combinational logic, thus creating new logic optimization opportunities. Applying precomputation before the second step of resynthesis and retiming, however, results in most cases (column c) in a circuit with a smaller clock period than obtained via two steps of retiming and resynthesis by themselves. Thus, when combined with precomputation, the same retiming and resynthesis effort can produce faster circuits.

Note that resynthesis (with or without precomputation) may increase the clock period. This occurs because of the resynthesis script we used has no target clock period; there was no reason for the synthesis process to reject an optimized circuit. It also occurs because of the script used the reduce_depth command, which restructures a technology-independent network rather than a mapped circuit, the delay of each combinational stage after mapping could be larger or smaller than it was before resynthesis. The results may have been different had we instead used the technology-dependent command speed_up. Some circuits, such as s208.1 and s420.1, have single-register cycles that were not tackled by precomputation.

The area increase for the circuits obtained using precomputation was larger than that obtained using only resynthesis and retiming. This is because precomputation duplicates logic in the circuit, and resynthesis cannot always reclaim that area.

The geometric means in Table 2 show that we can almost double the benefit obtained via an additional resynthesis and retiming step when precomputation is employed. We have demonstrated that resynthesis with precomputation is better than resynthesis without it.

	(a) resy-ret		(b) 2 resy-ret		precomputation			
Circuit Name	T_{clk}	area	T_{clk}	area	(c) T_{clk}	(c) area	(d) steps	(e) max. steps
mm4a	0.81	0.62	0.75	0.59	0.66	0.76	3	5
mm9a	0.87	0.91	0.97	0.94	0.72	1.33	8	9
mult16a	0.33	1.62	0.28	1.59	0.25	1.85	2	10
mult16b	0.98	0.93	1.20	1.26	1.13	0.96	1	10
s1196	0.88	1.10	0.82	1.09	0.86	1.01[2]	1	2
s208.1	1.07	0.90	1.03	0.89	1.09	0.93	1,2	2
s27	0.52	0.92	0.55	1.00	0.56	1.62	1	2
s298	0.93	0.87	0.84	0.94	0.78	1.23	7	7
s344	1.13	1.13	0.92	1.13	0.83	2.11	8	10
s349	1.00	1.18	0.90	1.17	0.87	1.73	3	10
s382	0.92	0.96	0.74	1.04	0.77	0.98	1	1
s386	0.88	0.79	0.89	0.81	0.79	1.52	6	7
s400	0.74	1.02	0.68	1.00	0.63	1.48	8	10
s420.1	1.59	0.82	1.48	0.81	1.68	0.88	1-4	4
s444	0.76	0.93	0.83	0.98	0.62	1.71	9	10
s526	1.05	0.82	0.93	0.98	0.74	0.97	6	8
s641	0.84	1.00	0.81	1.08	0.74	1.18	1	4
s713	0.85	0.96	0.80	1.05	0.76	1.23	6	6
geometric mean	0.93	0.88	0.90	0.89	0.84	1.26		

Table 2: Results of applying precomputation. The minimum normalized clock period and corresponding normalized area after (a) one resynthesis-retiming step, (b) two resynthesis-retiming steps, (c) a resynthesis-retiming step followed by precomputation and an additional resynthesis-retiming step. The last two columns indicate: (d) number of precomputations in the circuit that resulted in the smallest clock period, and (e) the maximum number of precomputation steps that were possible.

4 Using Precomputation in Architectural Resynthesis

The circuits described thus far are comprised of combinational logic and registers. The clocked elements in the circuit, however, can also be arrays of registers. Register arrays

[1] The script was run without the redundancy removal command, because SIS was unable to perform redundancy removal reliably.

are essential in designing systems, typically used as temporary storage for either input, output, or intermediate results of a computation. Examples of register arrays are many: register files, look-up tables, FIFO queues, and caches. Details of array implementations – such as the basic array cells, array sizes and organizations – differ. However, all arrays share common characteristics that designers often exploit to improve performance. This section begins by isolating and modeling the basic characteristics of register arrays, and we then describe how precomputing an array's output synthesizes a bypass transformation. Next, we describe how precomputation results in synthesizing a lookahead transformation. We conclude this section with an example.

4.1 Model – Adding Register Arrays

A register array consists of clusters of registers that are accessed via a specific mechanism. Arrays perform read and write operations. The basic array structure is shown in Figure 5(a). An array can be modeled hierarchically, as shown in Figure 5(b). Node A is a register vertex that represents the actual array registers. The registers can be updated only through the $Write$ node, and read through the $Read$ node. The $Write$ node models the write operation, and the $Read$ node models the read operation. The dotted edges are internal to the array node. It is not possible to apply architectural retiming or place registers along those edges. The path from rd_addr to rd_data is a combinational path, while the path from the write port to rd_data has a delay of one cycle.

The timing and functionality of an array, A, are specified by the following equations:

$$\begin{aligned} A^{t+n}[wr\_addr^t] &= wr\_data^t \\ rd\_data^{t+m} &= A^t[rd\_addr^t] \end{aligned}$$

For simplicity, we deal only with arrays with one read and write port. We also assume that n equals one and m equals zero – typical number for reasonably sized register arrays in current technologies.

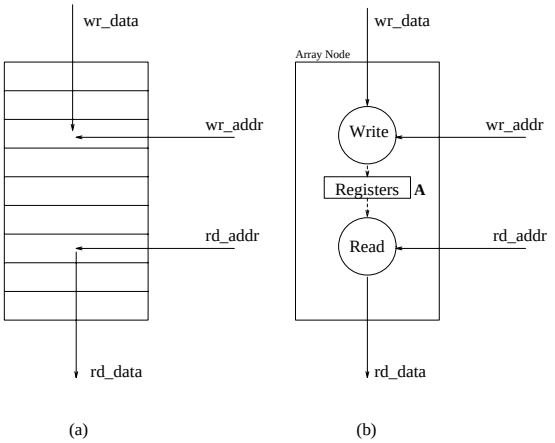

Figure 5: Register Arrays. (a) Basic structure. (b) Model.

4.2 Synthesizing Bypasses

Architectural retiming can add a negative register followed by a pipeline register along any of the edges leading into an array node, or along the output of the array. When added along the output edge, rd_data, the negative register's implementation can be deduced based on the equations in Section 4.1. The value of the output of the negative register, G, in cycle t is the input shifted forward in time. Thus:

$$\begin{aligned} G^t &= rd\_data^{t+1} \\ &= A^{t+1}[rd\_addr^{t+1}] \\ &= \begin{cases} A^t[rd\_addr^{t+1}] & \text{if } rd\_addr^{t+1} \neq wr\_addr^t \\ wr\_data^t & \text{otherwise} \end{cases} \end{aligned}$$

Based on the result of comparing the precomputed rd_addr and the wr_addr in cycle t, signal rd_data equals either the input data, wr_data, in cycle t, or the array contents from the previous cycle. The resulting modification is a bypass, or a forwarding circuit. We illustrate bypass synthesis using the example in Section 4.4.

4.3 Synthesizing the Lookahead Transformation

Applying precomputation along a single-register cycle results in an interesting transformation, known as lookahead [6, 9]. In this case, there is no previous physical pipeline stage; the execution of the computation in the previous cycle constitutes the previous pipeline stage.

When applying precomputation, the net result is unrolling the computation along the single-register cycle, as shown in Figure 6. Two consecutive iterations of the computation can now be exposed to combinational optimizations. Originally, $A(n)$ computes based on the value of B from the previous iteration, $B(n-1)$. With the unrolling and the proper initialization of the added pipeline register, $A(n)$ becomes a function of $B(n-2)$. Along this new cycle is twice as much delay and twice as many registers. If we can optimize the delay along the cycle to compensate for the delay of the added pipeline register, we have effectively reduced the clock period.

The second application of architectural retiming to the example in Section 4.4 shows how precomputation synthesizes a lookahead transformation that exposes two iterations of the computation to combinational optimizations.

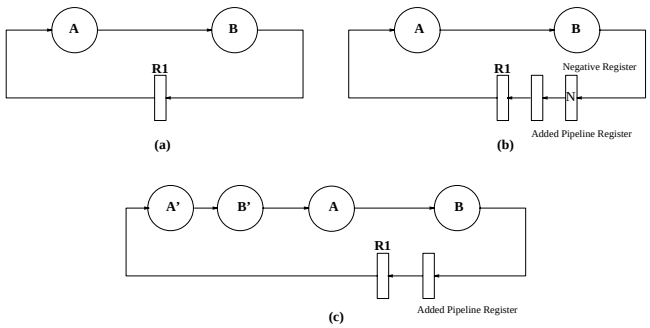

Figure 6: Lookahead synthesis via precomputation. (a) Original circuit. (b) Adding the negative/pipeline register pair. (c) Final circuit.

4.4 Example: Bypass and Lookahead Synthesis

The example presented in this section illustrates: (a) bypass synthesis, (b) lookahead synthesis, and (c) the performance improvement possible with the repeated application

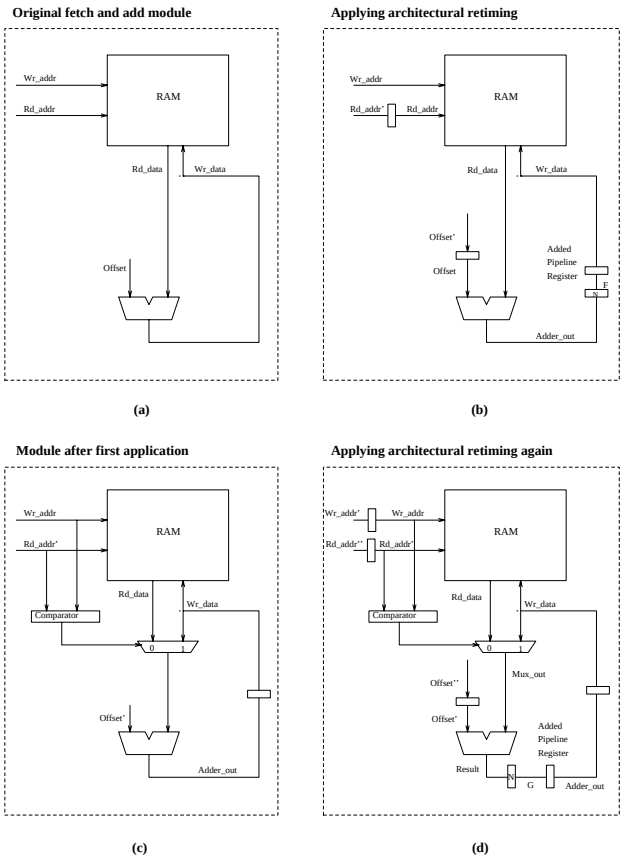

Figure 7: Fetch and add circuit: Applying architectural retiming twice, and resynthesizing resulting circuits.

of architectural retiming. The example circuit[3], illustrated in Figure 7(a), consists of a RAM and an adder. Each clock cycle data is read from the RAM and added to an offset.

The problem with the initial implementation is that the circuit cannot run at the specified clock period because of delays along the cycle – the delays involved in reading the RAM, performing the addition, and writing the result.

Pipelining the cycle solves this problem; however, pipelining prevents executing consecutive fetch and add operations, because the result of one add operation is not available as an operand for the next addition. Precomputation, however, exposes optimizations that permit reducing the clock period while allowing a new fetch and add operation each clock cycle. We describe the application of precomputation.

We assume that the offset signal and the read and write addresses are available in earlier cycles. The circuit's behavior can be described as follows:

$$\begin{aligned} RAM^{t+1}[Wr\_addr^t] &= Wr\_data^t \\ Rd\_data^t &= RAM^t[Rd\_addr^t] \\ Wr\_data^t &= Rd\_data^t + Offset^t \end{aligned}$$

Performance can be improved by applying architectural retiming. A negative/pipeline register pair is inserted at the output of the adder, as shown in Figure 7(b). The output of the negative register can be computed as:

$$\begin{aligned} F^t &= Adder\_out^{t+1} = Offset^{t+1} + Rd\_data^{t+1} \\ &= Offset^{t+1} + RAM^{t+1}[Rd\_addr^{t+1}] \end{aligned}$$

[3]This example was provided by Bob Alverson of Tera Computers.

The values of the signals in cycle $t+1$ can be evaluated based on signals available from the previous pipeline stages. The value of signals Rd_addr and $Offset$ in clock cycle $t+1$ is the same as the values of $Rd\_addr'^t$ and $Offset'^t$, respectively. The state of the FIFO in clock cycle $t+1$ is the same as it was in the previous cycle except for the one location that is updated.

The implementation of the circuit after synthesizing this precomputation as a bypass circuit is illustrated in Figure 7(c). The specifications for this new circuit are:

$$\begin{aligned} RAM^{t+1}[Wr\_addr^t] &= Wr\_data^t \\ Rd\_data^t &= RAM^t[Rd\_addr'^t] \\ Wr\_data^{t+1} &= Adder\_out^t \\ Adder\_out^t &= \begin{cases} Offset'^t + Wr\_data^t \\ \quad \text{if } (Rd\_addr'^t = Wr\_addr^t) \\ Offset'^t + RAM^t[Rd\_addr'^t] \\ \qquad \text{otherwise} \end{cases} \end{aligned}$$

Signal Rd_data in the original circuit differs from that in the modified circuit, as the latter is read one clock cycle earlier. This first application of architectural retiming reduces the clock period to 54% of that of the original, at an area increase of 24%.

If the performance at this point is not satisfactory, we apply architectural retiming once again. We insert a negative and a pipeline register at the output of the adder, as shown in Figure 7(d), and precompute the output of the added negative register, G:

$$\begin{aligned} G^t &= Result^{t+1} = Offset'^{t+1} + Mux\_out^{t+1} \\ &= \begin{cases} Offset'^{t+1} + Wr\_data^{t+1} \\ \quad \text{if } Rd\_addr'^{t+1} = Wr\_addr^{t+1} \\ Offset'^{t+1} + RAM^{t+1}[Rd\_addr'^{t+1}] \\ \qquad \text{otherwise} \end{cases} \end{aligned}$$

To express G in terms of signals available in cycle t, the following signals are precomputed:

$$\begin{aligned} Offset'^{t+1} &= Offset''^t \\ Rd\_addr'^{t+1} &= Rd\_data''^t \\ Wr\_data^{t+1} &= Offset'^t + Mux\_out^t \\ &= \begin{cases} Offset'^t + Wr\_data^t \\ \quad \text{if } (Rd\_addr'^t = Wr\_addr^t) \\ Offset'^t + RAM^t[Rd\_addr'^t] \\ \qquad \text{otherwise} \end{cases} \\ RAM^{t+1}[Rd\_addr'^{t+1}] &= \begin{cases} Wr\_data^t \\ \text{if } (Rd\_addr'^{t+1} = Wr\_addr^t) \\ RAM^t[Rd\_addr'^{t+1}] \\ \qquad \text{otherwise} \end{cases} \end{aligned}$$

Substituting the preceding expressions, G^t is synthesized to have one of the following values:

$$G^t = \begin{cases} Offset''^t + Offset'^t + Wr\_data^t \\ Offset''^t + Offset'^t + RAM^t[Rd\_addr'^t] \\ Offset''^t + Wr\_data^t \\ Offset''^t + RAM^t[Rd\_addr''^t] \end{cases}$$

If the current operands are not dependent on the results from the previous two clock cycles, the negative register simply adds the $Offset$ and the proper RAM data (last case in the equation for G^t). The rest of the cases cover the potential dependencies between the current operands and the results of the previous two additions. Reviewing the preceding equations, it should be clear that an add operation starts two cycles earlier than it did in the original circuit. In addition to synthesizing the value of the output of the negative register, the control signals (not shown) are also synthesized appropriately.

The final circuit is shown in Figure 8. The RAM is read either using Rd_addr'' or Rd_addr'. The negative register can be implemented efficiently as a three-input adder. The adder's inputs are selected appropriately based on comparison results of the proper read and write addresses. The critical cycle now has two registers instead of one, and the clock period can be smaller. This second application of precomputation-based architectural retiming reduces the clock period by an additional 14% of that of the original, at an area increase of about 200%. This example is a real circuit; the final synthesized solution is the same one obtained and used by the designer.

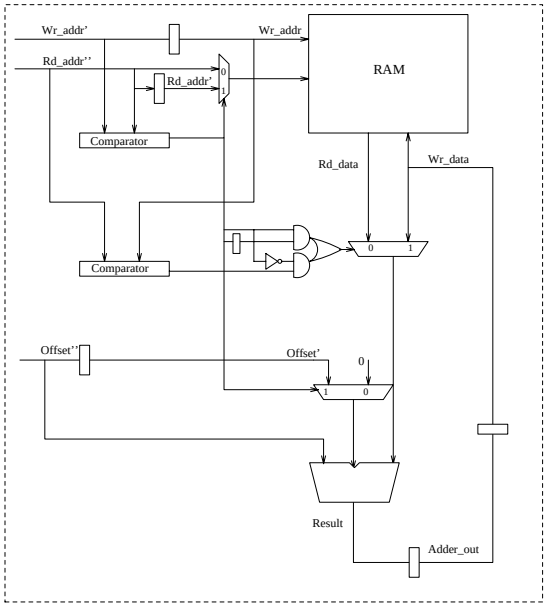

Figure 8: Final fetch and add circuit after applying architectural retiming twice.

5 Related Work

The idea of modifying register boundaries and optimizing resulting combinational blocks using combinational optimization techniques has been previously proposed. The register movements may be large, such as in peripheral retiming [8], which attempts to expose the whole circuit for combinational resynthesis. The register movements could also be restricted to a portion of the circuit. For example, Dey et al. identify subcircuits with equal-weight reconvergent paths (petals) to which peripheral retiming can be applied [3]. Sequential optimization techniques based on more fine-grain register movements include applying local algebraic transformations across register boundaries [2] and using implicit retiming, where a local retiming move is applied to specific kernels whose support (input) variables are register outputs [1]. Although precomputation in this paper was applied only to structural sequential circuits, it could have been applied before technology mapping, as in implicit retiming. Architectural retiming synthesizes the same results for the example circuits described in Bommu's thesis [1].

Note that in some instances precomputation results in a simple local forward retiming [7] move. This occurs if there is no interaction between the current and previous pipeline stage, and if none of the vertices duplicated in the precomputation function drives vertices that are not part of that function. Improvement is still possible if the precomputation function can be optimized.

Lookahead and bypassing are certainly not new optimizations. Lookahead has been used extensively in DSP high-level synthesis algorithms [6, 9]. Bypassing is a practical technique used in modern day processors [10].

6 Conclusion

This paper contributed to the understanding, synthesis, and evaluation of precomputation. Using one pair of negative and pipeline registers and optimizing across the boundaries of the previous pipeline stages, we have demonstrated that precomputation is a powerful technique in the optimization circuits at both the architectural and logic levels. We have shown how to incorporate precomputation in sequential logic resynthesis. We have also shown that precomputation encompasses two important architectural transformations: lookahead and bypassing, two effective optimizations used often in processor designs and DSP synthesis.

References

[1] S. Bommu. "Sequential Logic Optimization with Implicit Retiming". Master's thesis, University of Massachusetts, 1996.

[2] G. De Micheli. "Synchronous Logic Synthesis: Algorithms for Cycle-Time Minimization". IEEE Transactions on Computer-Aided Design, 10(1):63–73, Jan. 1991.

[3] S. Dey, F. Brglez, and G. Kedem. "Partitioning Sequential Circuits for Logic Optimization". In IEEE International Conference on Computer Design, pages 70–6, 1991.

[4] J. Fishburn. "A Depth-Decreasing Heuristic for Combinational Logic; or How to Convert a Ripple-Carry Adder into a Carry-Lookahead Adder or Anything in-between". In Proc. 31th ACM-IEEE Design Automation Conf., pages 361 –4, June 1990.

[5] S. Hassoun and C. Ebeling. "Architectural Retiming: Pipelining Latency-Constrained Circuits". In Proc. of 33th ACM-IEEE Design Automation Conf., June 1996.

[6] P. Kogge. The Architecture of Pipelined Computers. McGraw-Hill, 1981.

[7] C. Leiserson, F. Rose, and J. Saxe. "Optimizing Synchronous Circuitry by Retiming". In Proc. of the 3rd Caltech Conference on VLSI, pages 87–116, Mar. 1983.

[8] S. Malik, E. Sentovich, R. Brayton, and A. Sangiovanni-Vincentelli. "Retiming and Resynthesis: Optimizing Sequential Networks with Combinational Techniques". IEEE Transactions on Computer-Aided Design, 10(1):74–84, Jan. 1991.

[9] K. Parhi. "Look-ahead in Dynamic Programming and Quantizer Loops". In IEEE International Symposium on Circuits and Systems, pages 1382–7, 1989.

[10] D. Patterson and J. Hennessy. "Computer Architecture : A Quantitative Approach". Morgan Kaufmann Publishers, 1990.

[11] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli. "SIS: A System for Sequential Circuit Synthesis". Technical Report UCB/ERL M92/41, University of California, Dept. of Electrical Engineering and Computer Science, May 1992.