Clock Period Optimization During Resource Sharing and Assignment

Subhrajit Bhattacharya [*]
Dept. of Computer Science
Duke University
Durham, NC 27706

Sujit Dey
C&C Research Labs
NEC USA
Princeton, NJ 08540

Franc Brglez [†]
CBL, Dept. of ECE
North Carolina State Univ
Raleigh, NC 27695

Abstract - This paper analyzes the effect of resource sharing and assignment on the clock period of the synthesized circuit. We focus on behavioral specifications with mutually exclusive paths, due to the presence of nested conditional branches and loops. It is shown that even when the set of available resources is fixed, different assignments may lead to circuits with significant differences in clock period.

We provide a comprehensive analysis of how resource sharing and assignment introduces long paths in the circuit. Based on the analysis, we develop an assignment algorithm which uses a high-level delay estimator to assign operations to a fixed set of available resources so as to minimize the clock period of the resultant circuit. Experimental results on several conditional-intensive designs demonstrate the effectiveness of the assignment algorithm.

I. Introduction

Resource sharing has been used effectively by several high level synthesis approaches to reduce the area required to implement a design. Besides resources being shared amongst operations in different states of a schedule, conditional statements in the behavioral description allows sharing of resources amongst mutually exclusive operations in the same state. Several resource sharing approaches have been proposed which are applicable to conditional-intensive designs [1, 2, 3, 4, 5].

It has been observed [3, 4, 6] that, as resource sharing is increased, the area of the synthesized circuit decreases but the clock period may increase. Various aspects of the resource sharing problem have been addressed in [3, 4, 5, 6]. Area optimization under a fixed set of resources has been addressed in [3]. The increase in delay due to sharing is offset by using faster modules in [4]. However, faster modules typically also require larger area. In [5], an area-delay cost function has been used to synthesize circuits initially described at the functional level to minimize hardware cost and delay.

We show that even when the set of resources to be shared is fixed, the clock period of the circuit can vary significantly, depending upon the assignment. In Section II, we discuss assignment of operations of the *Blackjack dealer* process [7] to a fixed set of resources resulting in circuits with clock period 87.14 ns and 43.74 ns for two different assignments, a difference of 50%. We give a comprehensive analysis of the relationship between assignment and clock period in conditional-intensive designs which allow sharing of not only arithmetic units but also comparators. We present an assignment algorithm which minimizes the clock period, given a fixed allocation of resources and a schedule.

Section III analyzes sources of long (critical) paths due to resource sharing and assignment, suggesting assignment techniques to generate circuits whose critical paths have small delays. In Section IV, we outline an assignment algorithm to perform resource sharing such that the clock period of the synthesized circuit is minimized, while satisfying the resource constraints. The algorithm is applied to several VHDL descriptions which have numerous conditional branches and loops. The experimental results in Section V demonstrate the effectiveness of the algorithm in minimizing the clock period of the synthesized circuits.

II. Motivation

Sharing resources is a common technique used for minimizing the area of a design. Before analyzing the effect of sharing resources on the area and delay of the synthesized designs, it is instructive to look at Table 1 which reports the area and delay of the 8-bit and 16-bit implementations of several commonly used resources, using SIS technology mapper with fanout optimization [8], OASIS layout tools [9], and the lib2.genlib standard cell SCMOS 2.0 library [10]. Assigning multiple operations to the same resource (sharing the resource) may require additional multiplexors to select the inputs of the operations being shared. Hence, sharing a low-area unit like a $(=)$Comp unit, with an area similar to a mux, does not result in area savings unless the $(=)$ operations to be shared have the same inputs. In contrast, sharing a $(<)$Comp, a $(<, =, >)$Comp, an adder or an ALU, can result in significant savings in area.

We next motivate the need for assignment techniques to minimize the increase in delay resulting from sharing of resources. Consider the VHDL code fragment shown in Figure 1. It has been taken from a VHDL description of the *dealer* process, one of the processes in the behavioral description of the Blackjack chip [7]. While the complete schedule for the description of the dealer process is omitted for lack of space, Figure 2(a) shows the operations of the *dealer* process scheduled to be executed in one of the states.

Table 1: Area/delay for library modules in SCMOS 2.0 technology.

Unit	**8 bit**		**16 bit**	
	Area $[\mathbf{mm^2}] * \mathbf{100}$	**Delay [ns]**	**Area** $[\mathbf{mm^2}] * \mathbf{100}$	**Delay [ns]**
Comp$(=)$	8.5	5.54	17.5	6.71
Comp$(<)$	17.2	10.69	41.9	19.96
Comp$(<, =, >)$	19.5	12.65	45.2	21.75
adder	19.6	12.33	46.1	22.02
alu$(+, -)$	31.4	13.44	71.8	23.43
mux	7.5	4.19	17.4	4.85

[*] S. Bhattacharya was supported by a grant from CCRL, NEC USA.
[†] F. Brglez was supported in part by a grant from SRC.

```
entity DealerChip is
    port ( Ready: out Bit;
        Deal: in Bit;
        Dealt: out Bit;
        CardValue: out Value;
        CardSuit: out Suit);
end DealerChip;

architecture Behavior of DealerChip is

begin

Dealer: process

type DeckSuit is array (DeckIndex) of Suit;
type DeckValue is array (DeckIndex) of Value;

variable TheSuits: DeckSuit;
variable TheValues: DeckValue;
variable Seed: DeckIndex := 23;
variable Card: DeckIndex;
variable Limit: DeckIndex;

begin

    . . .
6.  Limit := DeckSize - Seed;
7.  Asuit := NoSuit;
8.  PresentSuit := TheSuits[Card];
9.  While ASuit < Hearts loop
10.     Asuit := Asuit + 1;
11.     Avalue := Ace;
12.     While AValue <= King loop
13.         While PresentSuit /= NoSuit loop
14.             if Card < DeckSize then
15.                 Card := Card + 1;
                else
16.                 Card := 1;
                end if;
17.             PresentSuit := TheSuits[Card]
            end loop;
18.         TheSuits (Card) := ASuit;
19.         TheValues(Card) := AValue;
20.         if Card > Limit then
21.             Card := Card − Limit;
            else
22.             Card := Card + Seed;
            end if;
23.         Avalue := Avalue + 1;
24.         PresentSuit := TheSuits[Card];
        end loop;
    end loop;
. . .
end process;
end Behavior;
```

Figure 1: VHDL code for the Blackjack *dealer* process.

Let us assume that the allocated resources are three comparators and two ALUs, and resource sharing is performed. Two possible assignments of the operations of Figure 2(a) to the available resources, and the resultant RT-level circuits are shown in Figures 2(b) and 2(c) respectively. The Finite State Machine (FSM) implementing the schedule consists of the state Flip Flops (FFs) and the next state logic, shown in Figure 2(b) and 2(c) as SFF and NS respectively.

In assignment 1, the comparison operation 12 (op_{12}) is assigned to a comparator unit, cmp_1. Since op_{13} and op_9 are mutually exclusive, they can be shared and have been assigned to comparator cmp_2. Similarly, op_{14} and op_{20} have been assigned to cmp_3, op_{21} and op_{22} to ALU_1 and op_{15} and op_{10} to ALU_2. Since op_{13} and op_9 are on mutually exclusive branches of op_{12}, the output of cmp_1 controls the muxes at the inputs of cmp_2. Consequently, there is a path from cmp_1 to cmp_2 as shown in the circuit of Figure 2(b). We say that two units *X* and *Y* are **chained**, if there exists a path from unit *X* to unit *Y*, like cmp_1 and cmp_2 in the circuit of Figure 2(b). Similarly, cmp_2 is chained to cmp_3 and cmp_3 to ALU_1. The longest path thus consists of three comparators, one ALU and two muxes. Using the delay values for 8 bit units given in Table 1, the delay of the longest path in the circuit of Figure 2(b) is 63.96 ns. This path is critical in determining the clock period. As shown in Table 2, the clock period of the circuit implementing the complete *dealer* process of Figure 1 under assignment 1, is 87.14 ns.

In contrast, the longest path in the circuit corresponding to assignment 2 shown in Figure 2(c) has only two comparator units and two muxes. The corresponding delay of the longest path is 34.47 ns. The clock period of the complete implementation under assignment 2, as shown in Table 2, is 43.74 ns, a reduction of 50% ! This example illustrates that, even when the number of resources is fixed, different assignments can lead to circuits with significant difference in clock period. Consequently, an effective assignment algorithm is needed which, given a fixed set of resources, assigns the operations to minimize the clock period of the final circuit.

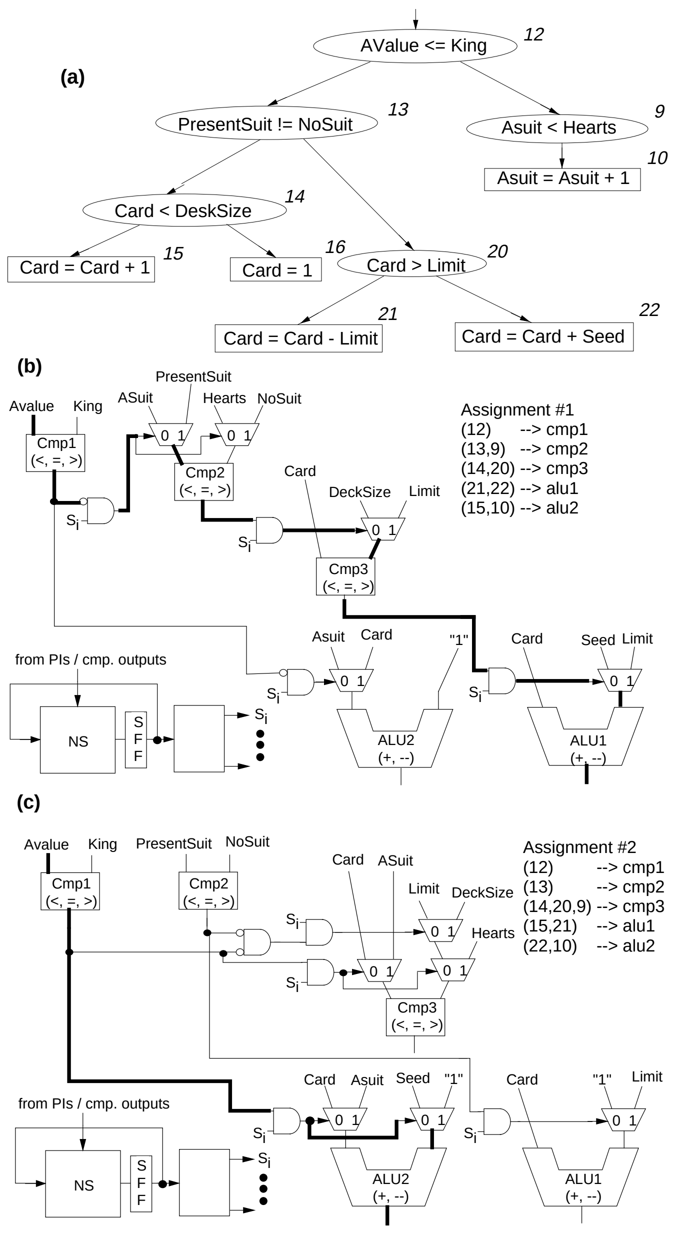


Figure 2: (a) Operations in a state of a schedule of the *dealer* process. (b) Circuit corresponding to assignment 1. (c) Circuit corresponding to assignment 2.

III. Analysis and Minimization of the Effect of Resource Sharing on Clock Period

We analyze the effect of different types of resource sharing on the clock period of the synthesized circuit. We show that in spite of having a fixed number of resources to share, assignment plays a critical role in determining the clock period of the final circuit. Three fundamental ways of sharing and their role in *chaining* resource units to create long paths are detailed. The *transitivity* of chaining is illustrated. Assignment techniques to derive circuits with short critical paths are suggested. We begin the section with a few definitions with respect to an acyclic, directed and rooted graph.

The **Common Ancestor (CA)** of a pair of nodes v_i and v_j is a comparison node v_k, such that there are disjoint paths from v_k to v_i and v_j. The common ancestor of a set of nodes S, is the set CA of all comparison nodes v_k, such that there exists disjoint paths from v_k to at least any two nodes in S. In Figure 2(a), CA$(\{16, 22\}) = \{13\}$, while CA$(\{16, 22, 9\}) = \{13, 12\}$.

The **Level Number (level)** of a node v_i in the graph is defined as:
level(v_i) = 1, v_i does not have a successor.
level(v_i) = MAX$\{$level(v_j) $+ 1 \mid v_j$ is a successor of $v_i\}$
level(S) = MAX$\{$level(v_i) $\mid v_i \in$ S$\}$, where S is a set of nodes.

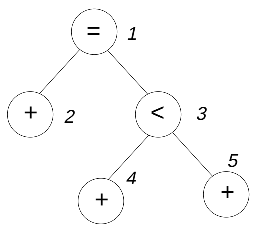
In the graph of Figure 6(a), the level numbers are shown in parentheses at the left of the vertices, while the operation numbers are shown on the right. For example, the level number of operation 15 is 1, and the level number of operation 14 is 2.

A pair of operations can share the same resource, that is, they are **compatible**, if (a) they are never executed in the same clock cycle and, (b) it is possible to assign them to the same resource. Two operations are never executed in the same clock cycle if (a) they are scheduled in separate states, or (b) they are in the same state but on mutually exclusive paths. Compatible operations can be shared in three fundamental ways: across mutually exclusive paths, across states and shared *implicitly*.

A. Sharing Mutually Exclusive Operations

Let two operations on mutually exclusive paths be assigned to the same unit u_2, and their common ancestor (CA) to unit u_1. There is a combinational path from unit u_1 to unit u_2 (chaining from u_1 to u_2) if and only if, *the two operations being shared have at least one operand different.* The chaining arises because the output of u_1 has to decide which of the different operands should be an input to u_2. In Figure 2(b), considering assignment 1, op_{13} and op_9 are assigned to cmp_2 and their CA, op_{12} to cmp_1. Since op_{13} and op_9 have different inputs, circuit 1 (Figure 2(b)) has a combinational path from cmp_1 to cmp_2. Similarly, assignment 1 creates a path from cmp_2 to cmp_3 and yet another from cmp_3 to ALU_1. By *transitivity* of combinational paths, we get a long combinational path $(cmp_1 \rightarrow mux \rightarrow cmp_2 \rightarrow mux \rightarrow cmp_3 \rightarrow mux \rightarrow ALU_1)$

We propose the assignment rules **R1** to **R4**, to perform resource sharing while avoiding the formation of the resource chains that create long paths in the resultant circuit.
(**R1**) *Share operations with all inputs common.* Since input selection is not necessary, there will be no chaining with the ancestor's output.
(**R2**) *Share those operations such that the sum of the delay of the resource unit on which the operations can be implemented and the delay of the unit on which their CA is implemented is minimum.* Figure 3 shows the CFG of the operations scheduled in one of the states of a schedule for the benchmark Fancy [11] and the available resources. Note that all the add operations, op_2, op_4 and op_5, can be shared.



Resources available
1 (=) unit
1 (<) unit
2 (+) units

Assignment #1:
(1) -> cmpeq, (3) -> cmplt
(2) -> add1, (4, 5) -> add2

Assignment #2:
(1) -> cmpeq, (3) -> cmplt
(2, 4) -> add1, (5) -> add2

Figure 3: The Fancy example: Illustrating sharing of operations whose CA has smaller delay.

Sharing op_4 and op_5 chains a $(<)$ unit with an adder through a mux (delay 46.83 ns, from 16 bit column of Table 1). However, sharing op_2 and op_4 creates a chain from the $(=)$ unit to an adder through a mux with delay 33.58 ns. This example illustrates the advantage of sharing operations whose CA has the least delay. The next two rules effectively break up long chains of resource units.
(**R3**) *Share operations whose common ancestor (CA) has the highest level number*. Referring to the datapath corresponding to assignment 1 in Figure 2(b), there is a long chain $(cmp_1 \rightarrow cmp_2 \rightarrow cmp_3 \rightarrow ALU_1)$. If, instead of sharing op_{21} and op_{22} (CA(21,22) has level number 2) as in assignment 1, op_{15} was shared with op_{21} (CA(15,21) has level number 3) as in assignment 2, cmp_2 would be directly chained with ALU_1, thus bypassing the intermediate cmp_3. This illustrates that, sharing operations whose CA is closer to the root of the CFG creates smaller chains.
(**R4**) *Share operations which have smaller level numbers.* Consider sharing operations (20,9,14) where all three have level number 2, and assign op_{13} which has level number 3 to a separate comparator as done in assignment 2 (Figure 2(c)), as opposed to sharing operations (9,13) and (14,20) as in assignment 1. Since op_{13} assigned to cmp_2 is not shared at all, the output of cmp_1 is no longer chained with cmp_2. This further breaks up the chain $(cmp_1 \rightarrow cmp_2 \rightarrow cmp_3)$ of the circuit produced by assignment 1 (Figure 2(b)), to generate the circuit shown in Figure 2(c) where the longest chain is $(cmp_1 \rightarrow ALU_1)$.

B. Implicit Sharing Inside a State

We define implicit sharing inside a state with respect to the CFG of operations scheduled in that state. *A functional unit* F *is said to be implicitly shared inside a state if and only if, there exists two or more mutually exclusive paths in the CFG of the state from a conditional operation O_i, to an operation O_k that is assigned to* F*, and the inputs to O_k differ depending upon the path executed.* Let the conditional operation O_i deciding which path to execute be assigned to a comparator unit *C*. Implicit sharing creates a combinational path from the comparator unit *C* to the implicitly shared unit *F*.

An example CFG is shown in Figure 4(a). The CFG contains the

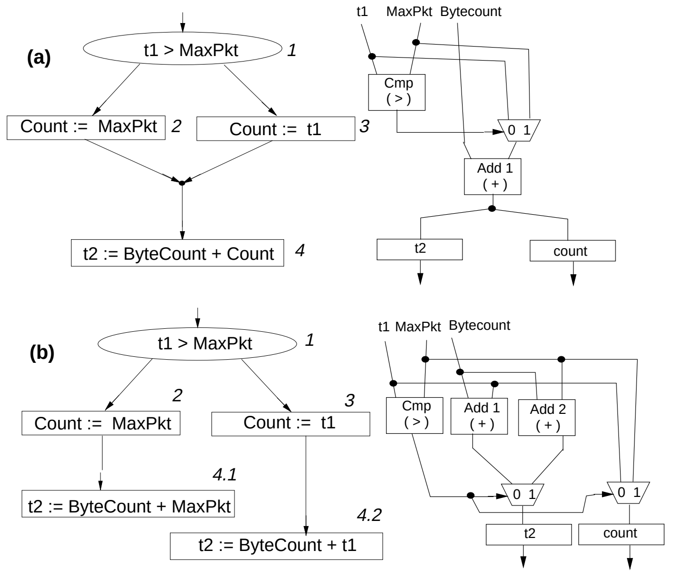


Figure 4: (a) Flowgraph to illustrate implicit sharing inside a state. (b) Duplicating operations to avoid implicit sharing.

operations to be executed in one of the states of a schedule of the *send* process of a network protocol X.25 [12]. If the comparison operation op_1 is assigned to a comparator, and op_4 is assigned to an adder, from the above definition we have implicit sharing of the adder and there will be a path from the comparator to the adder, as is shown in the datapath of Figure 4(a). However, consider Figure 4(b), where op_4 has been split into two operations, one for each mutually exclusive path to op_4. If each copy of op_4, $op_{4.1}$ and $op_{4.2}$ were to be assigned to a separate adder, there would be no implicit sharing. Also, there would be no chaining from the output of the comparator to which op_1 has been assigned to any of the two adders, as is shown in the datapath of Figure 4(b). Splitting implicitly shared operations converts a CFG into a tree and is hence referred to as a **control flow tree (CFT)**. Given a CFT, the assignment algorithm makes the choice whether the split operations should be assigned to the same unit or not, depending upon the availability of resource units.

C. Implicit Sharing across States

Existing resource sharing techniques consider sharing operations across states. However, sharing across states can also lead to implicit sharing. *In a behavioral description with mutually exclusive paths, it is possible for an operation to be scheduled in more than one state. When the occurrences of this operation in two or more states is assigned to the same unit, implicit sharing across states take place.* Implicit sharing across states can create resource chaining resulting in long combinational paths.

In Figure 5(a), we show a fragment of a CFG extracted from the control software code for the AutoPilot of an Unmanned Aerial Vehicle (UAV), written for the Motorola MC68HC11 microcontroller [13]. The available resources are two comparators, and two $(+/-)$ALUs. Note from Figure 5(b), op_6 could not be scheduled in state s_0 along path 1,3,5,6, since it would violate the resource constraint of 2 ALUs. We show that for the given schedule, implicit sharing across states creates a long path. An alternative assignment is given which creates shorter paths by avoiding implicit sharing across states.

In Figure 5(b), operation 6 is scheduled in two states, s_0 and s_2. Two possible assignments and the resultant datapaths are given in Figures 5(c) and (d). Let s_i^j be an instance of operation *i* scheduled in state s_j. Since assignment 1 in Figure 5(c) assigns s_0^6 and s_2^6 to ALU_2, by definition we have implicit sharing across states, and the path $cmp1 \rightarrow ALU_1 \rightarrow ALU_2 \rightarrow cmp2$ is created. However, if s_2^6 is assigned to ALU_1, two smaller chains $cmp1 \rightarrow ALU_1 \rightarrow ALU_2$, and $cmp1 \rightarrow ALU_1 \rightarrow cmp2$ are created as shown in Figure 5(d). Instead of assuming that the occurrences of the same operation in different states will be assigned to the same unit, we treat each occurrence as a separate operation. The assignment algorithm chooses the best assignment to minimize the clock period.

IV. An Assignment Algorithm for Minimizing the Clock Period

We present an assignment algorithm which, given a scheduled CFG and a set of allocated resources, synthesizes a circuit with minimum clock period. The assignment algorithm uses an assignment graph (AG) which has the following components.

(1) For each state of the scheduled CFG there is a corresponding CFT in the AG. All assignment nodes of the form $x \leftarrow y$ are removed from the CFT's. Each node in a CFT is assigned a level number.

(2) There exists a compatibility edge between each pair of compatible operations, with an associated edge weight. The operations can be on the same CFT (mutually exclusive) or on different CFT's (across states). The weight is an estimate of the delay penalty of sharing the two operations connected by the compatibility edge [14].

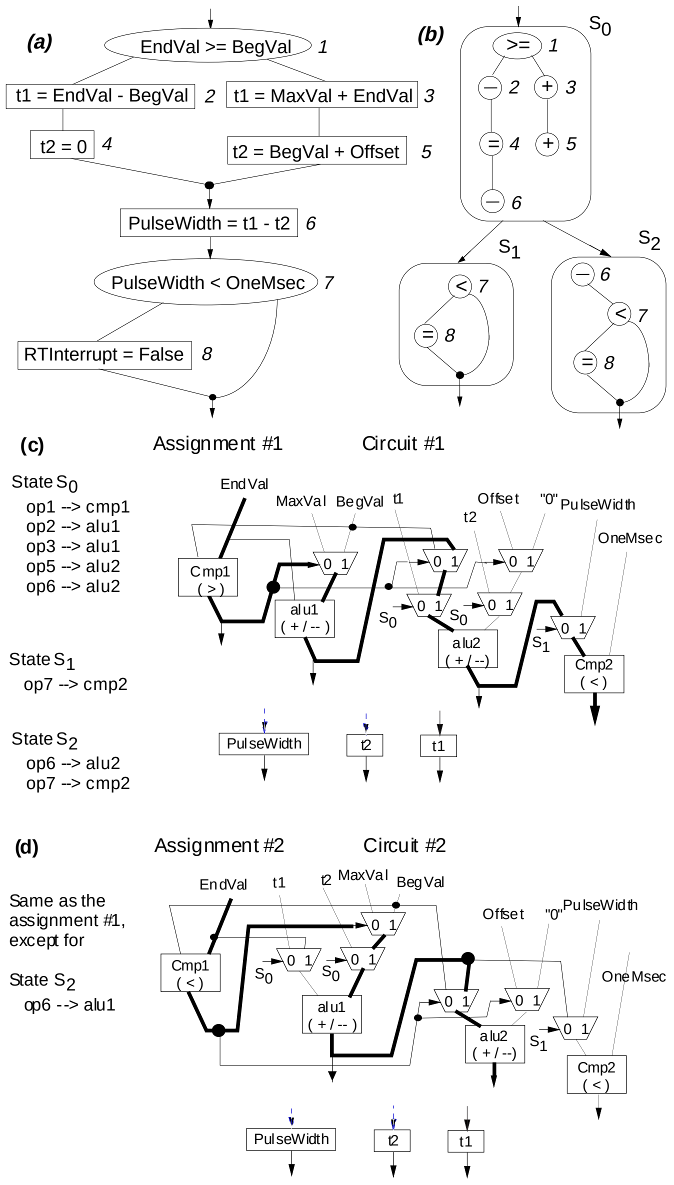


Figure 5: UAV AutoPilot: Illustrating implicit sharing across states.

Figure 6(a) shows the assignment graph corresponding to the CFG in Figure 2(a). The level numbers are shown on the left of each node inside parentheses in Figure 6(a), the node numbers are shown on the right. The dashed edges are the compatibility edges. Since op_{15} and op_{21} are on mutually exclusive branches of op_{13}, and they can be assigned to the same adder, there is a compatibility edge between them. The weight on the compatibility edge, also called the delay penalty, is the delay of the longest combinational path that results from sharing the two operations connected by the compatibility edge. The delay penalty of sharing op_{15} and op_{21} includes the delay of the adder, a mux at the input of the adder, and another comparator unit to which op_{13} is assigned to control the mux and is 30.28ns (using the delay values in the 8 bit column of Table 1). Details of the delay

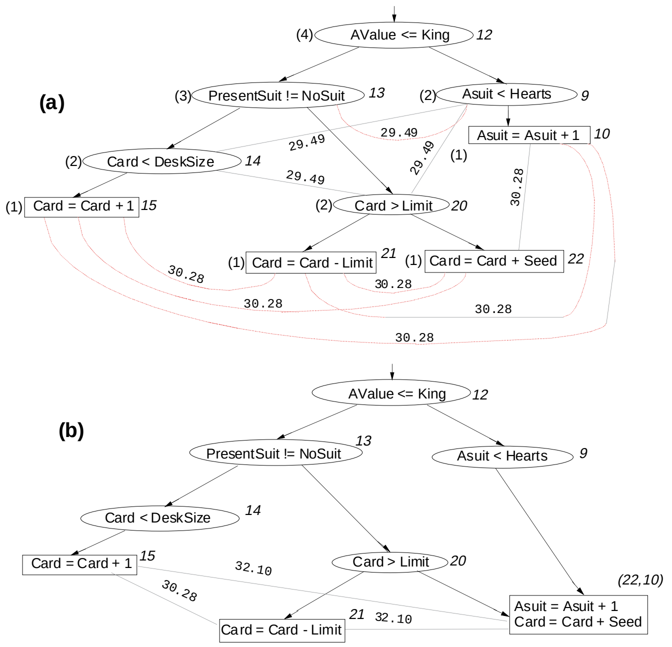


Figure 6: (a) Assignment graph for the CFG of Figure 2. (b) Assignment graph after assigning op21 and op22 to an alu unit (only relevant portions shown).

estimation algorithm is given in [14].

A. The Assignment Algorithm

We present an algorithm *ClkMin* which, given an assignment graph (AG) and a set of allocated resources, assigns the operations in the graph to the resources, so as to minimize the clock period of the resultant circuit. The assignment graph consists of the CFT of each state of a given schedule. Initially, each node in the AG contains one operation. Procedure *ClkMin* iteratively merges compatible nodes in AG such that in the final AG, each node (containing one or more operations) can be assigned to a separate available resource unit. The procedure *Select_Nodes_To_Share* uses assignment rules (R2) through (R4) (Section III) to select a pair of nodes to be shared such that the clock period of the final implementation is minimized. Since sharing can introduce false combinational loops [15] it is ensured that sharing the selected pair does not create any false loop in the circuit. Whenever a pair of AG nodes are chosen to be shared, the nodes are merged and the AG is updated (routine Update). If possible, a subset of the nodes of the updated AG are assigned to available resource units. The above process of selecting, updating and partial assignment is repeated until all AG nodes are assigned.

Select_Nodes_To_Share. This function which forms the core of our assignment algorithm chooses a compatible pair to be shared such that the final clock period is minimized. While rule (R1) of Section III is implemented by line 1 in *ClkMin*, rules (R2), (R3) and (R4) are implemented by lines 4 to 6 of *Select_Nodes_To_Share*. Line 4 of the code implements rule (R4) since the average of the level numbers of nodes with smaller level numbers will be smaller than the average of nodes with higher level numbers. Referring to Figure 6, applying line 4 results in operations (14, 20, 9) sharing the same resource (as in assignment 2), as opposed to operations (9, 13) and (14, 20) sharing resources (as in assignment 1). Line 5 directly implements (R3). Line 6 implements rule (R2). Since the compatibility edge weight is an estimate of the cost of sharing the two (sets of) operations, it includes the delay of the CA of the two operations and the cost of the unit on which the operations to be shared can be assigned as well as the cost of the mux which will have to be used to switch inputs into the resource unit. Hence, by picking the pair whose compatibility edge has minimum weight, it implements rule (R2).

Update. After a pair of nodes v_i, v_j have been selected to be shared, the assignment graph is updated as follows:

• The nodes are merged into a new node $v = (v_i, v_j)$, such that v contains all the operations previously contained in v_i and v_j.

• A compatibility edge is introduced between the new node v and node v_k, *if and only if*, there were compatibility edges between v_k and v_i and also between v_k and v_j before merging. Consider the Assignment Graph shown in Figure 6(a). After merging nodes 14 and 9, an edge is introduced between the new node (14,9) and 20, but not between (14,9) and 13.

• The compatibility edge weights need to be updated. In the AG shown in Figure 6(a) assume node 10 and node 22 have been merged. This implies they will be assigned to the same unit (as in ALU_2 of assignment 2 shown in Figure 2). Note that the edge between node 15 and the merged operations (22,10) shown in the updated AG in Figure 6(b) now has weight 32.10. The updated weight reflects the extra delay for sharing three operations on the same resource unit (due to more muxes at the input), than sharing two operations on one unit (as reflected by the weight between node 15 and node 21 which is 30.28). The updating of the edge weights is done by the delay estimation algorithm described in [14].

• The new node $v = (v_i, v_j)$, is assigned a level number which is given by MAX(level(v_i), level(v_j)).

Safe_Assign. Let V be a set of nodes in AG and R a subset of the allocated resource types. There can be more than one unit of a particular resource type available. The procedure *Safe_Assign* assigns the nodes in V to the units of R if the following two conditions hold:
(1) Each node in V can be assigned to a separate resource unit of a type which is in R, and
(2) no operation nodes which is in AG but outside V is implementable on a resource type belonging to R.
Condition (1) assures that the assignment is feasible. Condition (2) makes it *safe*. It is safe, since nodes in (AG−V) cannot be implemented on any resource of type R. Hence, if an assignment of nodes in (AG−V) existed before assigning the nodes in V to resources of type R, the assignment would exist even after the nodes in V were assigned. After assignment, the nodes in AG corresponding to the nodes in V are marked assigned and the resource units in R are removed from the list of resources, *resource_list*.

The pseudo-code of the assignment algorithm is given below.

```
   ClkMin(resource_list, AG) {
1.       merge all nodes in AG which have common inputs;
2.       Safe_Assign(AG, resource_list);
3.       while (∃ nodes in AG that have not yet been assigned) {
4.         $(v_i, v_j)$ = Select_Nodes_To_Share(AG);
5.         if (no sharable pair $(v_i, v_j)$ exists)
6.           return ASSIGNMENT_FAILED;
7.         Update($v_i$, $v_j$, AG);
8.         Safe_Assign(AG, resource_list);
         }
9.       return assignment;
   }
```

```
    **Select_Nodes_To_Share(AG)** {
1.      found := FALSE;
2.      while ((found == FALSE) & ($\exists$ compatible nodes in AG)) {
3.          $C_0 \leftarrow$ all compatible pairs $(v_i, v_j)$ in AG
            s.t. both $v_i, v_j$ are not assigned;
4.          $C_1 \leftarrow$ all $(v_i, v_j) \in C_0$
            s.t. $\mid level(v_i) + level(v_j) \mid /2$ is minimum;
5.          $C_2 \leftarrow$ all $(v_i, v_j) \in C_1$
            s.t. level(common_ancestor($v_i, v_j$)) is maximum;
6.          $C_3 \leftarrow$ all $(v_i, v_j) \in C_2$
            s.t. compat_edge_weight($v_i, v_j$) is minimum;
7.          $C_4 \leftarrow$ all $(v_i, v_j) \in C_3$
            s.t. sharing them does not introduce combinational
            false loops; delete compatibility edges between pairs
            which introduce false loops;
8.          if ($C_4 \neq \emptyset$) found := TRUE;
        }
9.      return a pair $(v_i, v_j)$ from $C_4$ if any, else return $(\emptyset, \emptyset)$;
    }
```

V. Experimental Results

To evaluate the effectiveness of the proposed assignment technique, we have synthesized the following conditional-intensive VHDL descriptions: the dealer process of Blackjack [7], Fancy [11], and the controller for the AutoPilot of an Unmanned Aerial Vehicle (UAV) [13]. Table 2 shows the synthesis results.

Each description is scheduled to satisfy the resource constraints specified in column *Resources.* Assignment 1 is a feasible assignment which satisfies the resource constraints, but does not focus on minimizing clock period. Assignment 2 has been obtained by the proposed algorithm *ClkMin* to minimize the clock period of the resultant circuit. The relevant portions of the CFG's for the Blackjack, Fancy and UAV AutoPilot processes, and the mapping of the CFG operations under the two assignments, are illustrated in Figures 2, 3, and 5.

Each RT-level circuit generated by the assignments is subjected to technology-dependent delay optimization, including fanout optimization, using the SIS technology mapper [8] and the *lib2.genlib* standard cell SCMOS 2.0 library [10]. Subsequently, OASIS [9] place and route tools are used to obtain the standard cell layout. Columns *Clock Period* and *Area* report the delay and area of the final implementations.

The results also demonstrate the effectiveness of the assignment algorithm in minimizing the clock period of the resultant circuit. For instance, in the case of Blackjack, assignment 1 produces a circuit with a clock period of 87.14 ns and an area of 121.8 mm^2. In contrast, the circuit produced by the proposed assignment algorithm *ClkMin* has a clock period of 43.74 ns and an area of 125.0 mm^2. A clock period reduction of 50% could be achieved by *ClkMin* for a nominal area penalty of 2.6%. On an average, a clock period reduction of 33% could be achieved. The change in area is marginal, because the clock period reduction was not effected by increasing the resources used, but by different assignments to the same resources.

Table 2: Results of appying *ClkMin* on several benchmarks.

Benchmark	**Resources**	**Assignment**	**Clock Prd [ns]**	**Area[mm^2] $*$ 100**
Blackjack (dealer) (8 bit)	3 $(<, =, >)$ 2 $(+, -)$	Assignment1	87.14	121.8
		Assignment2	43.74	125.0
Fancy (16 bit)	1$(=)$ 1$(<)$ 2$(+)$	Assignment1	53.96	2.91
		Assignment2	37.11	2.95
UAV AutoPilot (8 bit)	2$(<)$ 2$(+/-)$	Assignment1	49.27	3.29
		Assignment2	39.69	3.11

VI. Conclusions

This paper investigated the effect of resource sharing and assignment on the clock period of circuits synthesized from conditional-intensive behavioral descriptions. Under a fixed set of available resources, different assignments can lead to circuits with significant differences in clock period. An assignment algorithm has been proposed to identify an assignment which leads to a minimal clock period implementation. Experimental results demonstrate the feasibility of synthesizing high-speed circuits, not by using more resources or faster modules, but by effective assignment using a fixed set of resources.

Acknowledgments: We thank T. Misawa, K. Wakabayashi, and W. Wolf for helpful discussions.

References

[1] K. Wakabayashi and T. Yoshimura. A Resource Sharing and Control Synthesis Method for Conditional Branches . In *Proc. of the IEEE ICCAD*, 1989.

[2] T. Kim, J. W. S. Liu, and C. L. Liu. A Scheduling Algorithm For Conditional Resource Sharing . In *Proc. of the IEEE ICCAD*, 1991.

[3] R. A. Bergamaschi, R. Camposano, and M. Payer. Data-Path Synthesis Using Path Analysis. In *Proc. of the 28th ACM/IEEE DAC*, 1991.

[4] B. Gregory, D. MacMillen, and D. Fogg. ISIS: A System for Performance Driven Resource Sharing. In *Proc. of the 29th ACM/IEEE DAC*, June 1992.

[5] E. A. Rundensteiner and D. D. Gajski. Functional Synthesis Using Area and Delay Optimization. In *Proc. of the 29th ACM/IEEE DAC*, 1992.

[6] A. C. Parker, P. Gupta, and A. Hussain. Effects of Physical Design Characteristics on the Area-Performance Tradeoff Curve. In *Proc. of the 28th ACM/IEEE DAC*, 1991.

[7] *CLSI Users Guide*.

[8] E.M. Sentovich, K.J. Singh, C. Moon, H. Savoj, R.K. Brayton, and A. Sangiovanni-Vincentelli. Sequential Circuit Design using Synthesis and Optimization. In *Proceedings of the International Conference on Computer Design*, October 1992.

[9] K. Kozminski (ed.). *OASIS Users Guide*. MCNC, MCNC, Research Triangle Park, N.C. 27709, 1992.

[10] S. Yang. Logic Synthesis and Optimization Benchmarks, User Guide 3.0. In *Intl. Workshop on Logic Synthesis*, MCNC, Research Triangle Park, NC, May 1991.

[11] S. Bhattacharya, F. Brglez, and S. Dey. Transformations and Resynthesis for Testability of RT-Level Control-Data Path Specifications. *IEEE Trans. on VLSI Systems*, 1(3), September 1993.

[12] A. S. Tanenbaum. *Computer Networks*. Prentice Hall, Englewood Cliffs, N.J., 1989.

[13] K. Hintz and D. Tabak. *Microcontrollers: Architecture, Implementation, and Programming*. McGraw-Hill, New York, NY 10020, 1992.

[14] S. Bhattacharya, S. Dey, and F. Brglez. Clock Period Estimation and Optimization During Resource Sharing and Assignment. Technical Report 93-C013, C&C Research Labs, NEC USA, April 1993.

[15] L. Stok. False Loops Through Resource Sharing. In *Proc. of the IEEE ICCAD*, August 1992.