UC Irvine
ICS Technical Reports

Title
Layout-driven allocation for high level synthesis

Permalink
https://escholarship.org/uc/item/4zc6k948

Authors
Wu, Allen C.H.
Gajski, Daniel D.

Publication Date
1991-04-09

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Z
699
C3
no. 91-30

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

Layout-Driven Allocation for High Level Synthesis

Allen C-H. Wu
Daniel D. Gajski

Technical Report #91-30
April 9, 1991

Dept. of Information and Computer Science
University of California, Irvine
Irvine, CA 92717
(714) 856-8059

Abstract

We propose a hypergraph model and a new algorithm for hardware allocation. The use of a hypergraph model facilitates the identification of sharable resources and the calculation of interconnect costs. Using the hypergraph model, the algorithm performs interconnect optimization by taking into account interdependent relationships between three allocation subtasks: register, operation, and interconnect allocations simultaneously. Previous algorithms considered these three tasks serially. Another novel contribution of our algorithm is the exploration of design space by trading off storage units and interconnects. We also demonstrate that traditional cost functions using the number of registers and the number of mux-inputs can not guarantee the minimal area. To rectify the problem, we introduce a new layout area cost function and compare it to the traditional cost functions. Our experiments show that our algorithm is superior to previously published algorithms under traditional cost functions.

TABLE OF CONTENTS

1. Introduction 1

2. The allocation problem 2

2.1 Hypergraph formation 2

2.2 Layout-area cost function 7

2.3 Interchange optimization 8

2.3.1 Hyperedge merging by node relocation 8

2.3.2 Hyperedge merging by node swapping 11

2.3.3 Interchange under global considerations 11

2.4 The overall algorithm 15

3. Results and Discussions 17

4. Conclusions 19

5. Acknowledgements 19

6. References 25

LIST OF FIGURES

Figure 1. Hypergraph formation: (a) Data flow graph and schedule, (b) Variable assignments, (c) Hypergraph of (a), (d) Hypergraph after hyperedge megering. 4

Figure 2. Hyperedge merging. 6

Figure 3. Hyperedge merging by node relocation (a) before, (b) after. 9

Figure 4. Group relocation. 10

Figure 5. Node swapping. 12

Figure 6. The data transfer model. 13

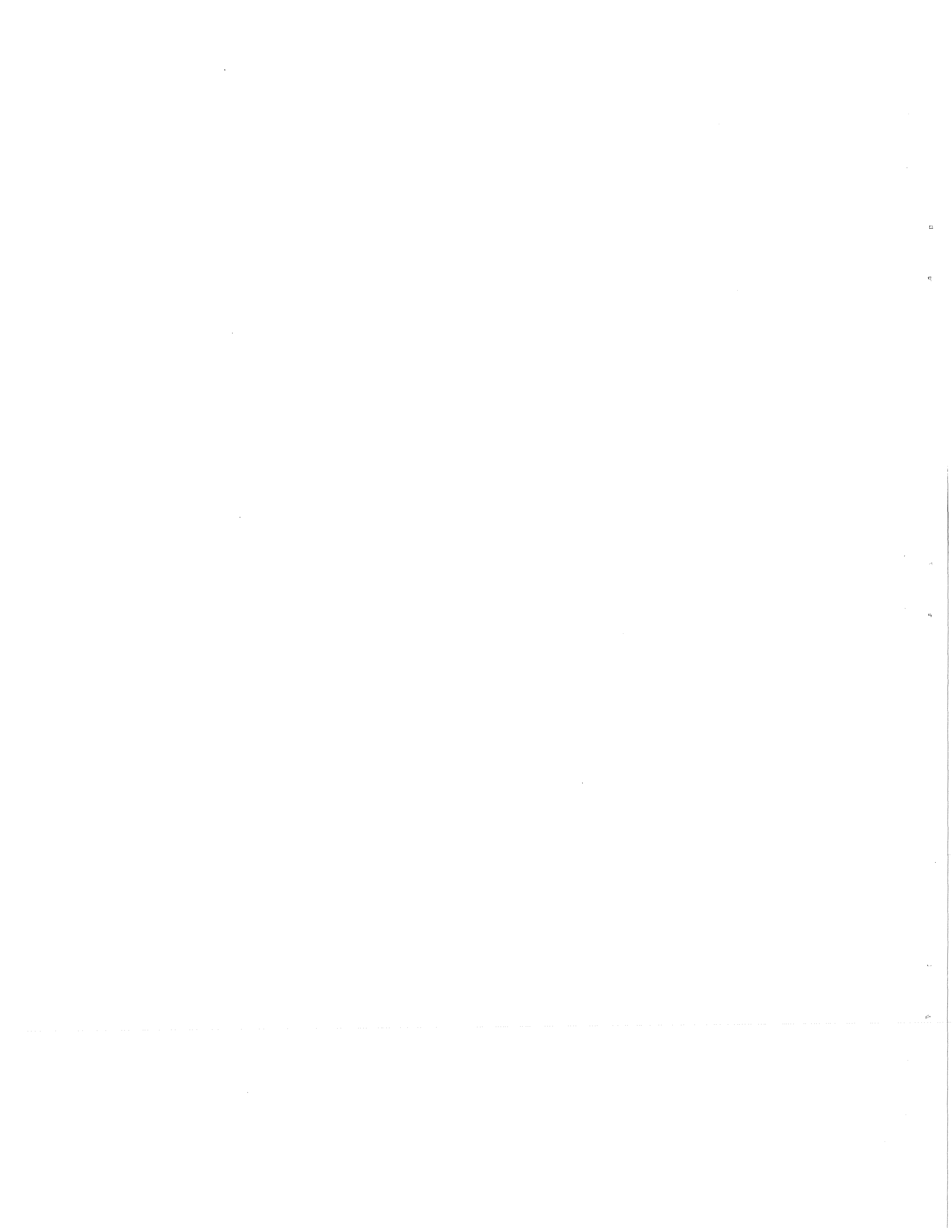
Figure 7. Interchange based on FU-REG-FU data path. 14

Figure 8. Figure 8. The 17-step Elliptic Filter example: (a) Schedule, (b) Structure, and (c) Variable and operation assignments. 20

Figure 9. The 19-step Elliptic Filter example: (a) Schedule, (b) Structure, and (c) Variable and operation assignments. 21

Figure 10. The lauoyt of a 16-bit 21-step Elliptic Filter example. 22

Figure 11. The relationships between area costs and actual areas. 23



LIST OF TABLES

Table 1. The area results of the Elliptic Filter example. 23

Table 2. Allocation results of the Elliptic Filter example. 24

1. Introduction

The purpose of high-level synthesis is to transform a behavioral description into a multistage register transfer design. Commonly, high-level synthesis consists of two phases: (i) scheduling and (ii) allocation. In the scheduling phase, operations are scheduled and assigned to the control steps to satisfy timing and resource constraints. In the allocation phase, operations are assigned to functional units, variables are assigned to storage elements, and required data transfers in each control step are assigned to interconnect units between functional units and storage elements. In this paper, we focus on the allocation problem.

Typically, unit allocation is divided into three subtasks: (i) register allocation, (ii) operator assignment, and (iii) connection allocation. In the register allocation phase, all variables are bound to a set of registers. Variables with nonoverlapping lifetimes can share the same register. In the operation assignment phase, operations are assigned to functional units such that no more than one operation within the same control step is assigned to the same functional unit. In the connection allocation phase, the communication paths (busses and multiplexers) are chosen so that the functional units and registers are connected to perform the required data transfers. The primary objective of unit allocation is to minimize the total hardware cost and to satisfy the design goal.

Most systems perform three allocation subtasks separately [2,3,4,5,7,8,9,10]. Since these three subtasks are interdependent, the result of one subtask may prevent other subtasks from finding an optimal solution. Hence, some other systems [1,13,15,16,17] perform functional units, registers, and interconnect units simultaneously. Traditionally, the design quality is evaluated using the number of registers and selector (mux) inputs. However, since the relationships between the structural design and the physical design have not been established, the traditional design quality measurement may not reflect the real physical design. The

approach described in this paper is different from previous approaches in three ways. First, we use a hypergraph model of design that facilitates the identification of sharable components and the calculation of interconnect costs. Second, using the hypergraph model, we formulate allocation as a partitioning problem. The partitioning problem is solved by using an interchange optimization technique which performs the three interdependent subtasks simultaneously. Third, we use a new layout-based cost function [20] to predict the real physical design. Moreover, this approach allows trading off storages and interconnections.

The remainer of this paper is organized as follows: Section 2 discusses the hypergraph model, the interconnect minimization technique, and the allocation algorithm. Section 3 presents our experimental results. Finally, Section 4 summarizes our approach.

2. The allocation problem

The objective of allocation is to assign operations to functional units and to assign variables to storage elements such that the storage and interconnect costs are minimized for a given set of functional units and storage elements. In this section, we first describe the hypergraph formation and the layout-area cost function. Then, we describe the hyperedge merging technique for interconnect minimization. Finally, we present the complete allocation algorithm.

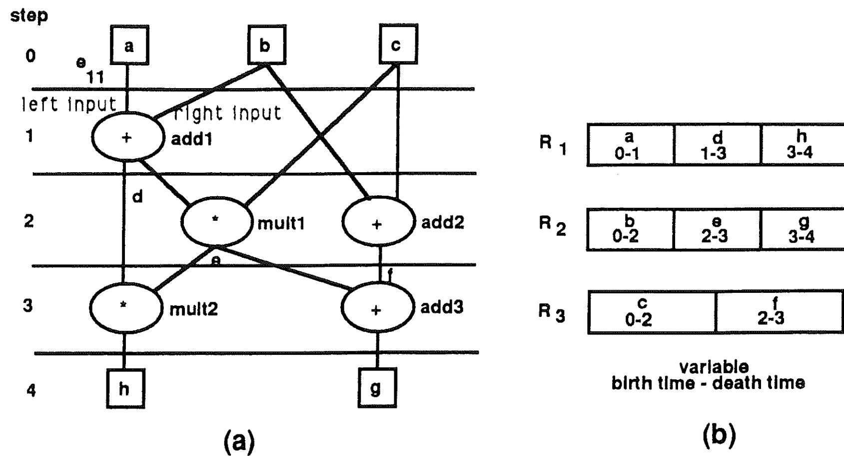
2.1. Hypergraph formation

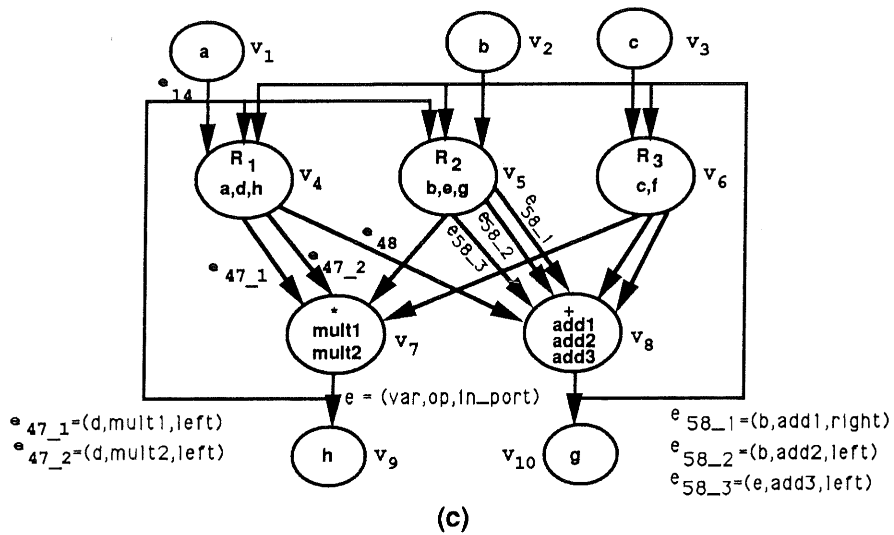
Let **G**=(**V**,**E**) denote a data flow graph (**DFG**), where $\mathbf{V}=\{\mathbf{v}_i \mid i=1..n\}$ is a set of operation nodes, and $\mathbf{E}=\{e_{jm} \mid j=1..n, m=1..n\}$, is a set of dependency edges. $\mathbf{Z}=\{\mathbf{z}_k \mid k=1..q\}$ is a set of variables in the **DFG**, where each variable corresponds to a set of edges as $z_k=\{e_{jm} \mid j=1..n, m=1..n\}$.

Let $\mathbf{H}=(V,E)$ denote a hypergraph, $V=V_{io} \cup V_{reg} \cup V_{op}$ in which there are three types of hypernodes: (i) input/output, (ii) register, and (iii) operation. $V_{io}=\{v_p \mid p=1..t\}$ is a set of i/o hypernodes which denote the input and output ports in the **DFG**. $V_{reg}=\{v_g \mid g=1..r\}$ denotes a set of register hypernodes and $v_g=\{\mathbf{z}_k \mid k=1..q\}$. Each register hypernode denotes a register which contains a set of variables. $V_{op}=\{v_c \mid c=1..s\}$ denotes a set of operation hypernodes and $v_c=\{\mathbf{v}_i \mid i=1..n\}$. Each operation hypernode denotes a single functional unit or a multi-functional unit such as adder/subtracter, multiplier, shifter, or ALU. Each operation hypernode contains a set of operation nodes in the **DFG**.

Let $E=\{e_{gc} \mid \{v_g,v_c\} \in V\}$, denote a set of hyperedges and $\mathbf{w}(e_{gc})$ is the weight of e_{gc}. Each hyperedge denotes the physical connection between two hypernodes which can be two functional units, a functional unit and a register, or two registers. The weight of a hyperedge is the number of dependency edges between two hypernodes, which also can be viewed as the number of variables (signals) communicating between two hypernodes. In addition, the hyperedge direction is depended on the data flow between hypernodes. For example, a hyperedge e_{12} is connected from v_1 to v_2 so that e_{12} is an outgoing hyperedge of v_1 while e_{12} is an incoming hyperedge of v_2. Since certain hypernode inputs are non-commutable, each hyperedge uses a flag to indicate the input position of the connecting hypernode.

For example, three registers, one functional unit +, and one functional unit * are given to perform data transfer for the data flow graph and its schedule shown in Figure 1(a). The variable assignment is shown in Figure 1(b). Operations **mult1** and **mult2** are assigned to the functional unit * and operations **add1**, **add2**, and **add3** are assigned to the functional unit +. Figure 1(c) shows an example of hypergraph formulation from the data flow graph shown in Figure 1(a). There is a set of input/output hypernodes $V_{io}=\{v_1,v_2,v_3,v_9,v_{10}\}$, and a set of three register hypernodes $V_{reg}=\{v_4,v_5,v_6\}$ where v_4 and v_5 consist of three variables, and v_6 consists of





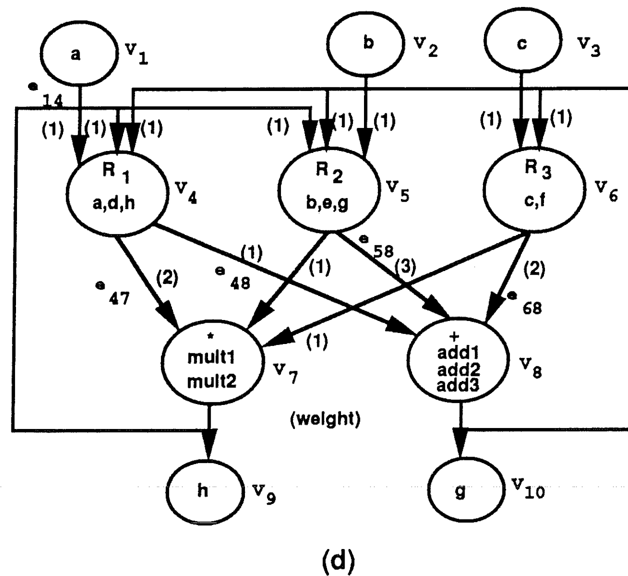


Figure 1. Hypergraph formation: (a) Data flow graph and schedule, (b) Variable assignments, (c) Hypergraph of (a), (d) Hypergraph after hyperedge megering.

two variables: v_4={**a**,**d**,**h**}, v_5={**b**,**e**,**g**}, and v_6={**c**,**f**}. In addition, there are two operation hypernodes $V_{op}=\{v_7,v_8\}$ where v_7={**mult1**,**mult2**}, v_8={**add1**,**add2**,**add3**}, **type**(v_7)=multiplier, and **type**(v_8)=adder.

At the time of hypergraph formation, each dependency edge is mapped to a pair of hyperedges. The process of hyperedge mapping from dependency edge consists of two parts: (i) from source node to register and (ii) from register to destination node. For example, $\mathbf{e}_{11}$ in Figure 1(a), is mapped to: (i) e_{14} from v_1 to v_4 and (ii) e_{48} from v_4 to v_8 in Figure 1(c).

After forming the hypergraph, the algorithm performs hyperedge merging to reduce interconnect cost. Hyperedge merging is important because it contributes to interconnect sharing and selector reduction, which will be described in the next section. Before merging, each hyperedge is labeled as a right data input or a left data input as shown in Figure 2. Hyperedge merging consists of two cases: (i) merging hyperedges on hypernodes with commutable inputs and (ii) merging hyperedges on hypernodes with non-commutable inputs. In both cases, two hyperedges can be merged if and only if: (i) they have same source and destination hypernodes, and (ii) they are entering the same input of the hypernode.

Consider Figure 2(a). If $\mathbf{e}_1$(op1_right) and $\mathbf{e}_2$(op2_right) are connected from $\mathbf{z}_3$ and $\mathbf{z}_4$ of the node **reg3** to right inputs of **op1** and **op2** respectively, $\mathbf{e}_1$(op1_right) is mapped to e_1, and $\mathbf{e}_2$(op1_right) is mapped to e_2. Since e_1 and e_2 are connected to the right input of **FU**, they can be merged. In Figure 2(b), if the operation hypernode inputs are commutable, two hyperedges e_1(op1_right) and e_2(op2_left) can be merged when they enter different inputs of the operation nodes (e_1 enters the right input of **op1** and e_2 enters left input of **op2**). Therefore, if the operation hypernode inputs are commutable, the hyperedges e_1(op1_left) and e_2(op2_right) can be commuted first and then merged into one hyperedge. On the other hand, in Figure 2(c) e_1(op1_right) enters the right input of **op1** and e_2(op2_left) enters the left input of **op2**. If the

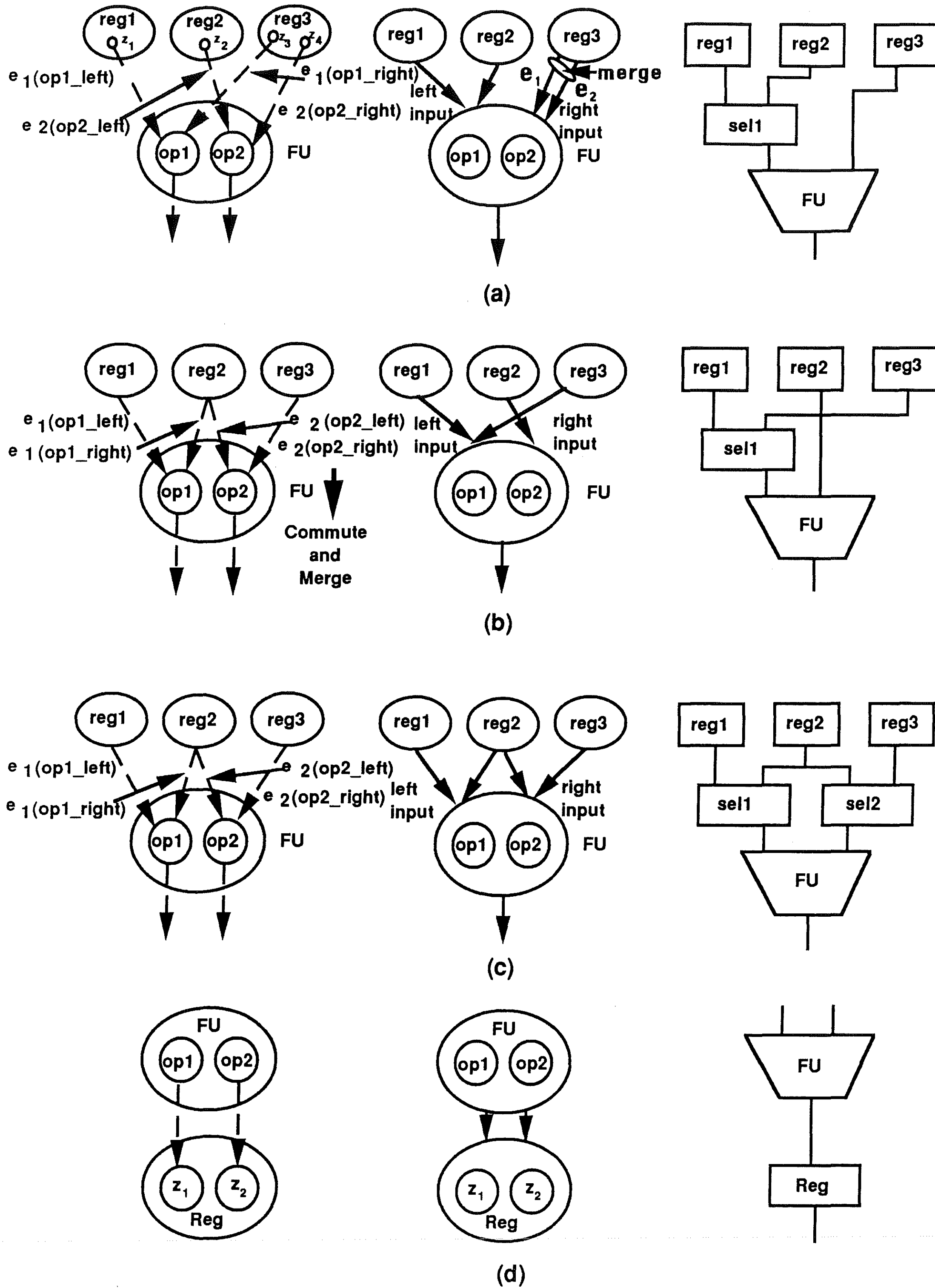


Figure 2. Hyperedge merging.

operation hypernode inputs are not commutable, then these two hyperedges can not be merged. Figure 2(d) shows that each register hypernode has only one input, the incoming hyperedges from the same operation hypernode of a register hypernode can be merged.

By applying hyperedge merging to the example of Figure 1(c), two hyperedges e_{47_1}=(**d**,**mult1**,**left**) and e_{47_2}=(**d**,**mult2**,**left**) can be merged into e_{47} with **w**(e_{47})=2 as shown in Figure 1(d). Since operation hypernode v_8 is an adder with commutable inputs, three hyperedges e_{58_1}, e_{58_2} and e_{58_3} can be merged into e_{58} with **w**(e_{58})=3.

2.2. Layout-area cost function

Using the hypergraph model, we first describe the interconnect cost for each hypernode in terms of the number of selectors and the number of inputs of selectors. We use a single-level interconnect model. If a hypernode has more than one incoming hyperedge on one of its input ports, then this input port needs a selector to select one input from several sources. This selector can be implemented with a mux or a bus with tri-state devices. For example, the left input port of **FU** in Figure 2(b) has two incoming hyperedges from two registers **reg1** and **reg3**. Thus, a two-input selector is needed for this input port. On the other hand, if an input port has only one incoming hyperedge, then a selector is not needed. Such a hyperedge is called the **minimal-cost** hyperedge. For example, the right input port of **FU** in Figure 2(b) has a minimal cost incoming hyperedge from **reg2** so that this input port does not need a selector. Furthermore, when an input port needs a selector, the number of inputs of this selector is equal to the number of incoming hyperedges on this input port.

To take into account the physical design effects, the area cost function is based on a bit-sliced stack architecture [11,18,19], which uses abutment to connect different bit slices, and over-the-cell routing for connecting different units inside one bit slice. The stack grows horizontally when the bit-width increases, and grows vertically when the number of units

increases. Using this layout architecture, the total area cost is the sum of four parts [20]: (1). Functional unit area, (2). Register area, (3). Interconnect unit area, and (4). Wiring area. We use the transistor counts as a function of the area consumptions. The transistor counts of the functional units and registers can be estimated by examing the component library [18]. The number of transistors in a selector is proportional to the number of inputs of the selector which also can be obtained from the component library. We use the sliced layout architecture [11] which has 13 over-the-cell routing tracks for each bit-slice. If the required routing tracks are less than 13 then the wiring area is not needed. The overall area cost is calculated as follows:

$$\mathbf{A}_{total} = c_1 \left(\sum_{k=1}^{n} \mathbf{trs}(\mathbf{FU}_k) + \sum_{j=1}^{m} \mathbf{trs}(\mathbf{Reg}_j) + \sum_{i=1}^{p} \mathbf{trs}(\mathbf{Sel}_i) \right) + \mathbf{A}_{wire}$$

$\mathbf{trs}(\mathbf{FU}_k)$ is the number of transistors in functional unit $\mathbf{k}$;
$\mathbf{trs}(\mathbf{Reg}_j)$ is the number of transistors in register $\mathbf{j}$;
$\mathbf{trs}(\mathbf{Sel}_i)$ is the number of transistors in selector $\mathbf{i}$;
c_1 is the transistor area coefficient (area/per transistor) which correlates to the layout technology and the layout system;
$\mathbf{A}_{wire}$ is the wiring area.

2.3. Interchange optimization

In the interchange optimization phase, the algorithm minimizes the interconnect cost by merging hyperedges. In the following sections, we first describe two possible ways to merge the hyperedges by interchanging variables and operations: (i) relocation and (ii) swapping. Then, we discuss the interchange technique under a global consideration.

2.3.1. Hyperedge merging by node relocation

The first possible way for hyperedge merging is to relocate variables among register hypernodes or operations among operation hypernodes. A variable $\mathbf{z}_k$ can be relocated from a source register hypernode v_{reg_source} to a destination register hypernode v_{reg_dest} if and only if v_{reg_dest} is free during the life time of $\mathbf{z}_k$. An operation $\mathbf{v}_i$ can be relocated from a source

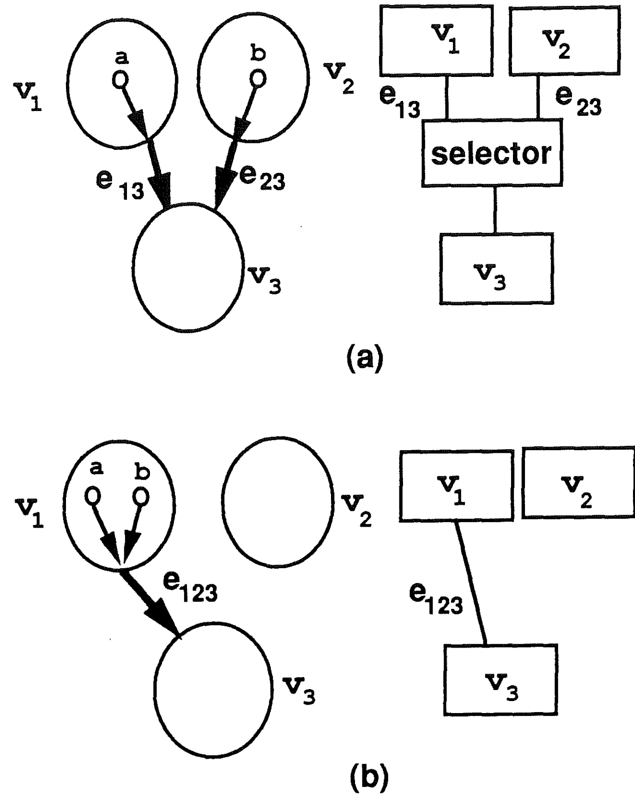


Figure 3. Hyperedge merging by node relocation (a) before, (b) after.

operation hypernode v_{op_source} to a destination hypernode v_{op_dest} if and only if: (i) v_{op_dest} can performs the same functions as $\mathbf{v}_i$, and (ii) there does not exist an operation $\mathbf{v}_j$ in v_{op_dest} such that $\mathbf{v}_j$ is assigned to the same control step as $\mathbf{v}_i$'s. We term the above conditions **relocation preconditions.** The node relocation can be performed if and only if the relocation preconditions are satisfied that is called a **feasible relocation.**

For a hypernode, the interconnect cost of this hypernode can be reduced by merging its incoming hyperedges. For example, consider the register hypernode v_3 in Figure 3. v_3 has two incoming hyperedges e_{13} and e_{23} from v_1 and v_2 respectively. Since v_3 has to select one input

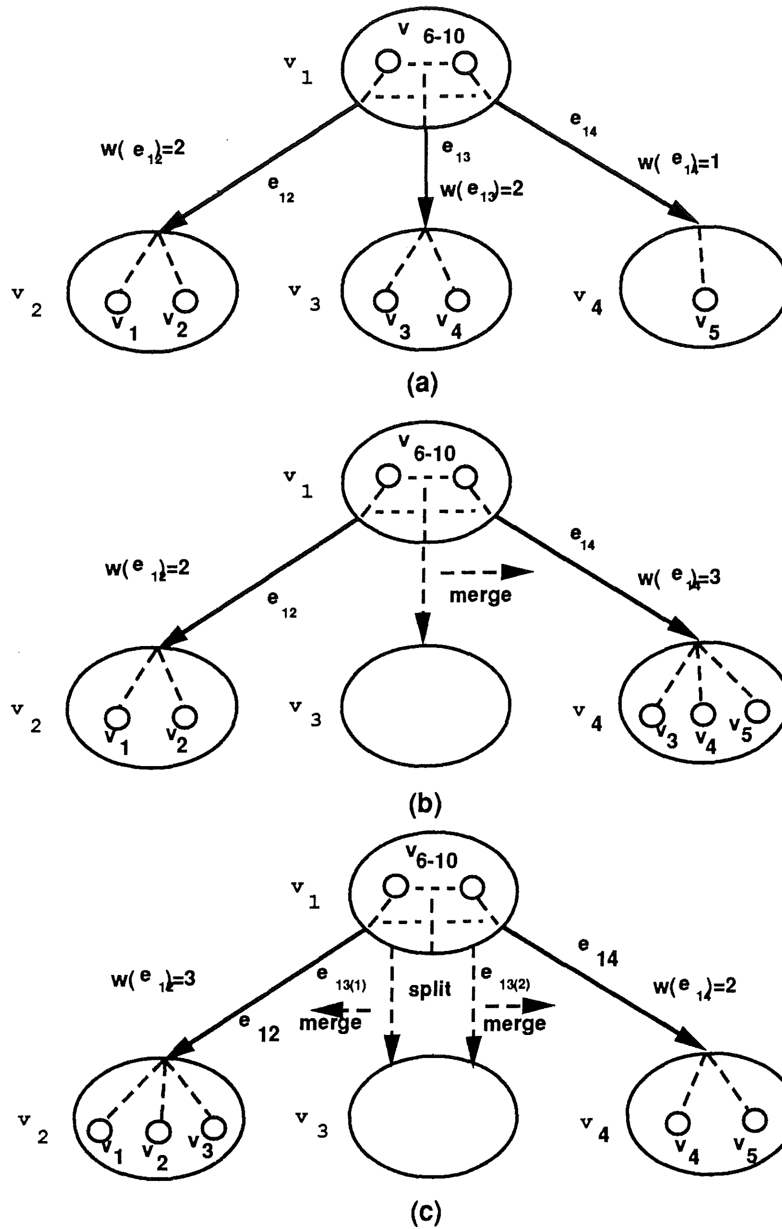


Figure 4. Group relocation.

from two sources v_1 and v_2, a 2-input selector is required for v_3 (Figure 3(a)). If node **b** of v_2 can be moved to v_1 then e_{13} and e_{23} can be merged into e_{123} (Figure 3(b)). As a result, v_3 does not need a selector for its input. The hyperedge merging in this case results in interconnect reduction.

The node relocation also allows group relocation, which relocates more than one node simultaneously. For example, in Figure 4(a), e_{13} can be merged with e_{14} by relocating a group of nodes $\mathbf{v}_3$ and $\mathbf{v}_4$ from v_3 to v_4 (Figure 4(b)), or splitting and relocating to different hypernodes as shown in Figure 4(c).

2.3.2. Hyperedge merging by node swapping

The second possible way for the hyperedge merging is to swap the variables between register hypernodes or to swap the operations between operation hypernodes. Node swapping can be viewed as a two-way node relocation problem. IN the node relocation, the node relocation is performed as relocating nodes from the same source hypernode to one or more destination hypernodes if there exists a feasible relocation. On the other hand, node swapping is performed when a one-way feasible relocation from a source hypernode to a destination hypernode can not be found; but a feasible relocation can be created by rearranging the nodes in the destination hypernode. For example, in Figure 5(a), assuming v_2 and v_3 are the operation hypernodes, e_{12} and e_{13} can be merged by relocating $\mathbf{v}_5$ from v_3 to v_2. If $\mathbf{v}_3$ in v_2 is assigned to the same control step as $\mathbf{v}_5$'s, then $\mathbf{v}_5$ can not be relocated from v_3 to v_2. However, e_{12} and e_{13} can be merged by swapping $\mathbf{v}_3$ and $\mathbf{v}_5$ as shown in Figure 5(b).

2.3.3. Interchange under global considerations

In general, a data path can be viewed as follows: a functional unit fetches data from a set of storage elements via a set of interconnect units, performs the data computations, then stores

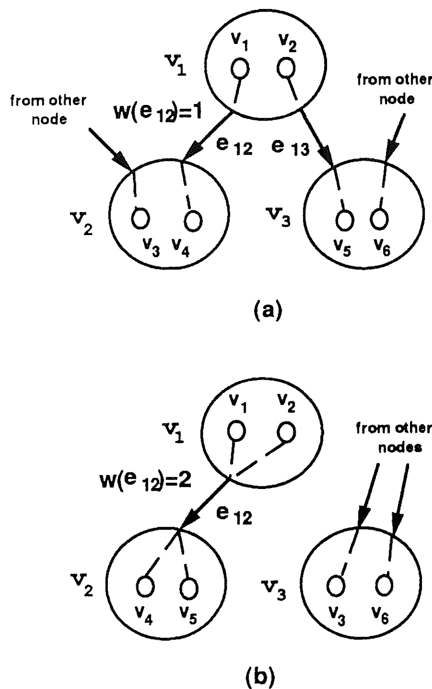


Figure 5. Node swapping.

the data back to the storage elements via a set of interconnect units as shown in Figure 6(a). Thus, a data path forms a closed loop relationship among the storages, functional units, and interconnect units. Because of this closed loop relationship, reducing the interconnect cost for a register or a functional unit by rearranging the variables or operations might increase the interconnect cost for other functional units or registers so that the total interconnect cost is unchanged or even increased.

To take into account the interdependent relationships between operation and register assignments, our interchange algorithm evaluates the node rearrangements by cutting the data

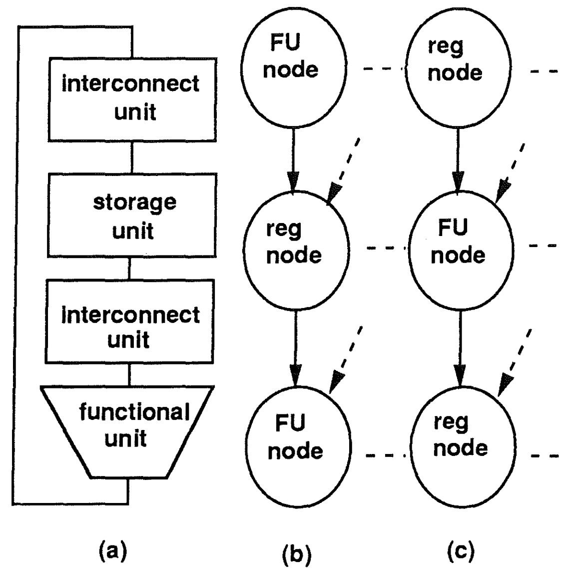


Figure 6. The data transfer model.

path loop into two basic forms: (i) register-functional unit-register (Figure 6(c)) and (ii) functional unit-register-functional unit (Figure 6(b)). In case (i), when the algorithm tries to reduce the interconnect cost of a functional unit node by rearranging the operations, the algorithm will take into account the register interconnect cost which will be affected by the node rearrangement. In case (ii), when the algorithm tries to reduce the interconnect cost of a register node by rearranging the variables, the algorithm will take into account the functional unit interconnect cost, which also will be affected by the node rearrangement.

Based on this global consideration, the algorithm tries to rearrange variables in the registers and operations in the functional units from a global scope so that the hyperedges can be merged, which results in selector inputs reductions. An example is shown in Figure 7(a). e_{36} and e_{46} can be merged by relocating variable z_2 from v_3 to v_4 so that the number of selector inputs of v_6 is reduced by 1. However, after relocating z_2 from v_3 to v_4, e_{13} has to be split into two hyperedges e_{13} and e_{14}, as shown in Figure 7(b), so that the number of selector inputs of v_4 is increased by 1. Thus, the total number of selector inputs are unchanged. However, if there

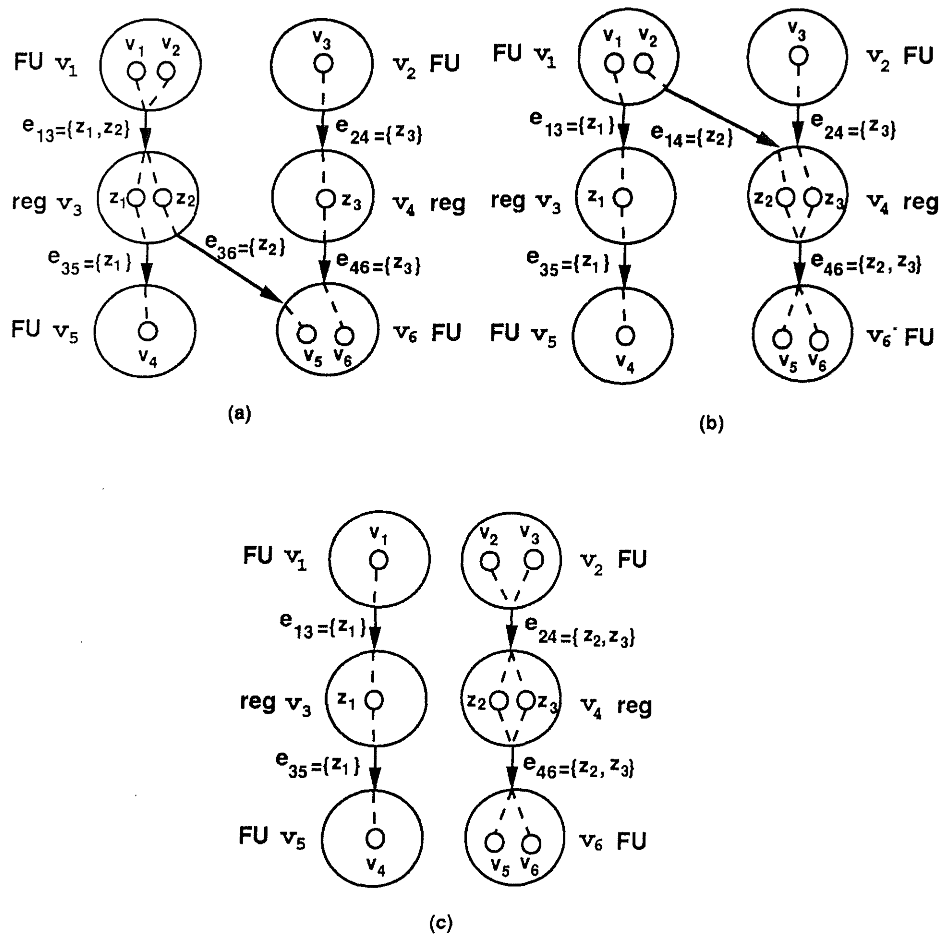


Figure 7. Interchange based on FU-REG-FU data path.

is a feasible relocation of moving $\mathbf{v}_2$ from v_1 to v_2, and this node relocation will not increase the overall selector inputs, then the algorithm finds a solution to achieve overall interconnect reduction. As a result, the algorithm relocates $\mathbf{z}_2$ and $\mathbf{v}_2$ to v_4 and v_2 respectively so that the overall interconnect cost is reduced (Figure 7(c)).

2.4. The overall algorithm

We assume that allocation is performed after scheduling with the available functional units and control time steps given. The algorithm consists of three phases: (i) initial register and operation assignments, (ii) hypergraph formation, and (iii) interchange optimization.

In the first phase, the algorithm determines the life time for all variables and sorts them in descending order according to their life span. The algorithm then determines the lower bound of required registers by using a left edge algorithm [6]. Starting with the minimal number of required registers, the algorithm assigns operations to the given functional units randomly such that no more than one operation within the same control step will be assigned to the same functional unit. In the second phase, the algorithm transforms the data flow graph into a hypergraph.

In the final phase, the algorithm minimizes the interconnect cost by interchanging the operations and the variables. The algorithm minimizes the incoming hyperedges of register hypernodes by relocating variables between register hypernodes. In addition, the algorithm minimizes the incoming hyperedges of operation hypernodes by relocating operations between operation hypernodes. A hypernode that has more than one incoming hyperedges is called a feasibly mergeable hypernode, and the incoming hyperedges of this hypernode are called feasibly mergeable hyperedges. For a feasibly mergeable hyperedge, the algorithm locates a set of variables or operations associated with this hyperedge, rearranges the nodes, and evaluates the area cost using the cost function described in the previous section. If a lesser area cost is

obtained, then a feasible merging solution has been found. The algorithm performs the allocation iteratively by incrementing the number of registers to trade-off the register and interconnect costs. For each allocation iteration, the algorithm runs repeatedly until no more improvement can be found.

Algorithm I Hardware Allocation

```
Let
    reg_count denote a given arbitrary number;
    P={e_h | h=1..w} denote a set of feasible merging hyperedges;
    Z={z_k | k = 1..q} denote a set of variables;
    F denote a set of given functional units;
    R={r_a | a=1..b} denote a set of registers;
    T denote a set of control steps for {v_i | i=1..n} in the data flow graph;
Hardware_Allocation(G,F,Z,T,reg_count){
    /*determine the lower bound of required registers*/
    R = left_edge_alg(Z);
    while (reg_count > 0){
        /*initial register and operation assignments*/
        init_reg_op_assignment(G,T,F,R);
        /*build hypergraph*/
        H = build_hypergraph(G,F,R);
        /*interchange optimization*/
        no_more_improve = FALSE;
        while (no_more_improve = FALSE){
            P = locate_feasible_merging_hyperedge(H);
            a_gain_merging = FALSE;
            for (h = 1 to w){
                /*relocate nodes associated with hyperedge e_h and evaluate the area cost*/
                gain = relocate_node(e_h);
                if (gain = TRUE){
                    rearrange_node(H,e_h);
                    a_gain_merging = TRUE;
                }
                if (a_gain_merging = FALSE)
                    no_more_improve = TRUE;
        }
        Output netlist and total area cost;
        /*incrementing register for next allocation run*/
        reg_count = reg_count - 1;
        if (reg_count > 0)
            R = R ∪ {r};
    }
}
```

Complexity analysis. Since the algorithm performs registers and selectors trade offs in several

runs (outer **while** loop), we consider only one allocation run which consists of three parts:

(1) Using the left edge algorithm, it takes O(**mlogm**) time to sort variables, and it takes O(**m**) time to assign variables, where **m** is the number of variables in the data flow graph. Therefore, it takes O(**mlogm**) to determine the lower bound of registers required for performing data transfer.

(2) The initial variable assignment takes O(**m**) time. The initial operation assignment takes O(**n**) time. In addition, it takes O(**2m+q**) time to build the hypergraph, where **n** is the number of operations, **m** is the number of edges in the data flow graph, and **q** is the number of hypernodes.

(3) In the interchange optimization procedure, it takes O(**pq**) time to locate feasible merging hyperedges, where **p** is the number of hyperedges. For each feasible merging hyperedge, it takes O(**r**) time to locate a set of nodes, rearranging nodes takes O(**rq**) time, and area estimation takes constant time, where **r** is the average number of variables or operations associated with the feasible merging hyperedge. Thus, each interchange optimization loop takes O(**prq**) time. In our experience, the local optimal (no_more_improve) state can be achieved by less than 20 iterations (interchange optimization **while** loop).

3. Results and Discussions

We have implemented the previously described algorithm using the C programming language on SUN4 workstations under the UNIX operating system. We have tested our algorithm on the elliptic filter benchmark with different control steps (19-step with 2-adder and 1-piped multiplier, 21-steps with 2-adder and 1-multiplier, 19-steps with 2-adder and 2-multiplier, and 17-steps with 3-adder and 2-piped-multiplier) collected from literatures [2,10,11] in order to compare our results to previously published results. Due to the paper length limitation, we only show the schedule, variable and operation assignment, and structure results for one 17-step and one 19-step examples which are shown in Figures 8 and Figures 9 respectively. The layouts were generated by [12]. Figure 10 shows a 16-bit Elliptic Filter example with 21-step and 10 registers.

We use the single-level multiplexer model described in Section 2.2. Moreover, we did not apply multiplexer optimization procedures, such as multiplexer merging. Table 2 (a), (b), (c), and (d) show the area results for 17-step, 19-step, 21-step, and 19-step with 2-adder and 1-piped multiplier designs respectively. Since the areas of multipliers for each design are same,

alg/system	c-step	#OP	#Reg.	#Mux i/ps	#Reg.	#Mux i/ps	#Reg.	#Mux i/ps	#Reg.	#Mux i/ps	Time(s)
Ours	17	2*p,3+	10	34	11	33	12	31	13	33	5.1
HAL	17	2*p,3+	10	n/a	11	n/a	12	31	13	n/a	120-480
CATREE	17	2*p,3+	10	n/a	11	n/a	12	38	13	n/a	n/a
REAL	17	2*p,3+	10	50	11	n/a	12	n/a	13	n/a	n/a
ELF	17	2*p,3+	10	n/a	11	28	12	n/a	13	n/a	n/a
ASYL	17	2*p,3+	10	38	11	25	12	24	13	n/a	n/a

alg/system	c-step	#OP	#Reg.	#Mux i/ps	#Reg.	#Mux i/ps	#Reg.	#Mux i/ps	#Reg.	#Mux i/ps	Time(s)
Ours	19	2*,2+	10	30	11	28	12	28	13	29	1.1
HAL	19	2*,2+	10	n/a	11	n/a	12	29	13	n/a	120-480
SAW	19	2*,2+	10	n/a	11	n/a	12	32	13	n/a	n/a
MIMOLA	19	2*,2+	10	n/a	11	30	12	n/a	13	n/a	n/a
REAL	19	2*,2+	10	39	11	n/a	12	n/a	13	n/a	n/a
ELF	19	2*,2+	10	n/a	11	30	12	n/a	13	n/a	n/a
ASYL	19	2*,2+	10	33	11	31	12	28	13	n/a	n/a

alg/system	c-step	#OP	#Reg.	#Mux i/ps	#Reg.	#Mux i/ps	#Reg.	#Mux i/ps	#Reg.	#Mux i/ps	Time(s)
Ours	19	1*p,2+	10	36	11	28	12	26	13	23	2.0
HAL	19	1*p,2+	10	n/a	11	n/a	12	26	13	n/a	120-480
REAL	19	1*p,2+	10	35	11	n/a	12	n/a	13	n/a	n/a
ELF	19	1*p,2+	10	n/a	11	30	12	n/a	13	n/a	n/a
ASYL	19	1*p,2+	10	30	11	28	12	26	13	n/a	n/a

alg/system	c-step	#OP	#Reg.	#Mux i/ps	#Reg.	#Mux i/ps	#Reg.	#Mux i/ps	#Reg.	#Mux i/ps	Time(s)
Ours	21	1*,2+	10	30	11	27	12	28	13	31	2.6
HAL	21	1*,2+	10	n/a	11	n/a	12	31	13	n/a	120-480
SPLICER	21	1*,2+	10	n/a	11	n/a	12	n/a	16	35	>50
ARYL-LYRA	21	1*,2+	10	30	11	n/a	12	n/a	13	n/a	0.65
ELF	21	1*,2+	10	n/a	11	24	12	n/a	13	n/a	n/a
ASYL	21	1*,2+	10	22	11	21	12	n/a	13	n/a	n/a

*: multiplier, *p: piped multiplier, +: adder.

Table 2. Allocation results of the Elliptic Filter example.

6. References

[1] S. Devadas and A. R. Newton, "Algorithms for Hardware Allocation in Data Path Synthesis," *IEEE Trans. on Computer-Aided Design*, vol. CAD-8, no. 7, pp. 768-781, 1989.

[2] E. Dirkes Lagnese and D. E. Thomas, "Architectural Partitioning for System Level Design," *Proc. 26th DAC*, pp. 62-67, 1989.

[3] B. S. Haroun and M. I. Elmasry, "Architectural Synthesis for DSP Silicon Compiler," *IEEE Trans. on Computer-Aided Design*, vol. 8, no. 4, pp.431-447, April 1989.

[4] Huang, C.Y., Chen, Y.S. et. al., "Data Path Allocation Based on Bipartite Weighted Matching", *Proc. 27th DAC*, pp. 499-504, June 1990.

[5] B. Pangrle, and D. Gajski, "Design Tools for Intelligent Silicon Compilation", *IEEE Trans. on Computer-Aided Design*, vol. CAD-6 no. 6, Nov. 1987.

[6] F. J. Kurdahi and A. C. Parker, "REAL: A Program for REgister ALlocation," *Proc. 24th DAC*, pp. 210-215, 1987.

[7] A. C. Parker, J. Pizarro and M. Mlinar, "MAHA: A Program for Datapath Synthesis," *Proc. 23rd DAC*, pp. 461-466, 1986.

[8] P. G. Paulin, J. P. Knight and E. F. Girczyc, "HAL: A Multi-Paradigm Approach to Automatic Data Path Synthesis," *Proc. 23rd DAC*, pp. 263-270, 1986.

[9] P. G. Paulin, and J. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASICs", *IEEE Trans. on Computer-Aided Design*, vol. CAD-8 no. 6, June 1989.

[10] C. J. Tseng and D. P. Siewiorek, "Automated Synthesis of Data Path in Digital Systems," *IEEE Trans. on Computer-Aided Design*, vol. CAD-5, no.3, pp. 379-395, 1986.

[11] Lawrence L. Larmore, D. D. Gajski, and Allen C-H Wu, "Layout Placement for Sliced Architecture," *IEEE Trans. on Computer-Aided Design*, to appear.

[12] Allen C-H Wu, G. D. Chen, and D. D. Gajski, "Silicon Compilation from Register-transfer Schematics," *Proc. ISCAS*, 1990.

[13] M. Balakrishnan and P. Marwedel, "Integrated Scheduling and Binding: A Synthesis Approach for Design Space Exploration," *Proc. 26th DAC*, pp.68-74, 1989.

[14] T. A. Ly, W. L. Elwood, and E. F. Girczyc, "A Generalized Interconnect Model for Data Path Synthesis," *Proc. 27th DAC*, pp.168-173, 1990.

[15] A. Mignotte and G. Saucier, "A Generalized Model for Resource Assignment," *Fifth International Workshop on High-Level Synthesis*, pp.37-43, 1991.

[16] K. Kucukcahar and A. C. Parker, "Data Path Tradeoffs Using MABEL," *4th International Workshop on High-Level Synthesis*," 1990.

[17] C. Hitchcock and D. Thomas, "A Method of Automatic Data Path Synthesis," *Proc. 20th DAC*, pp.484-489, 1983.

[18] "Data Path Library," VLSI Technology, INC., 1988.

[19] R. Jamier and A. Jeraya, "APPLON: A Datapath Compiler," *Proc. ICCD*, 1985.

[20] Allen C-H Wu, Viraphol Chaiyakul and D. D. Gajski, "Layout Models for High-Level Synthesis," Tech. Rpt. #91-31, ICS Dept., UC Irvine, 1991.

[21] C. H. Gebotys and M. I. Elmasry, "VLSI Design Synthesis with Testability," *Proc. 25th DAC*, pp.16-21, 1988.