UC Irvine
ICS Technical Reports

Title
Back-annotation for interactive data path synthesis

Permalink
https://escholarship.org/uc/item/29w198zm

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

Publication Date
1991-04-09

Peer reviewed

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

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

Z
699
C3
no. 91-29

Back-Annotation for Interactive Data Path Synthesis

Allen C-H. Wu
Viraphol Chaiyakul
Daniel D. Gajski

Technical Report #91-29
April 9, 1991

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

Abstract

In order to take into account physical design effects, a designer needs a feedback mechanism during interactive data path synthesis. In this paper, we propose a hypergraph model and a back-annotation algorithm which provide a feedback mechanism for back-annotation from physical designs to behavioral descriptions. Given a control-data flow graph and its structural design, this back-annotation technique can not only evaluate the design quality but can also feedback the delay to each edge and node in the graph. Therefore, a designer can identify the critical paths and improve the design. The hypergraph model and the back-annotation algorithm allow us to bridge the gap between the behavioral description and the physical design.

Contents

1 Introduction **2**

2 Formulation **4**

2.1 The relationships between DFG and structure 4

2.2 The hypergraph 6

2.3 DFG-hypergraph formation 6

3 Back annotation of physical design **10**

3.1 Layout model 11

3.2 Delay model 13

3.3 The back-annotation procedure 14

4 Experiments and results **16**

4.1 A walk-through example 17

4.2 An application example 20

5 Conclusions **20**

6 Acknowledgements **21**

List of Figures

1 (a) Traditional data path synthesis, (b) Data path synthesis with feedback of physical information. 2

2 (a) The data flow between two operations and it's structural model, (b) The **var** node insertion, (c) The data transfer model. 5

3 Hypergraph formation: (a) Data flow graph and schedule, (b) Variable and operation assignments, (c) **var** node insertion, (d) Structural netlist, (e) Hypergraph. 7

4 The edge mapping (a) case 1 and (b) case 2. 8

5 Two data path layout architectures 11

6 The clock estimation model. 15

7 The layout of the Figure 2. example (a). Routing track assignments and (b). The final layout. 18

8 (a) Back-annotation of wire lengths and component delays and (b) Back-annotation of delay information to the DFG. 19

9 The layout of the Elliptic Filter example. 22

10 The Area-Clock tradeoffs of the Elliptic Filter example: (a) 4-bit, (b) 8-bit, (c) 16-bit, and (d) 32-bit. 23

1 Introduction

Data path synthesis for digital systems has been an active research topic in recent years ([3],[6],[7],[20],[22],[25],[27]). Data path synthesis usually 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 mapped into functional units, storage elements are allocated for variables, and connections are assigned to wires between units.

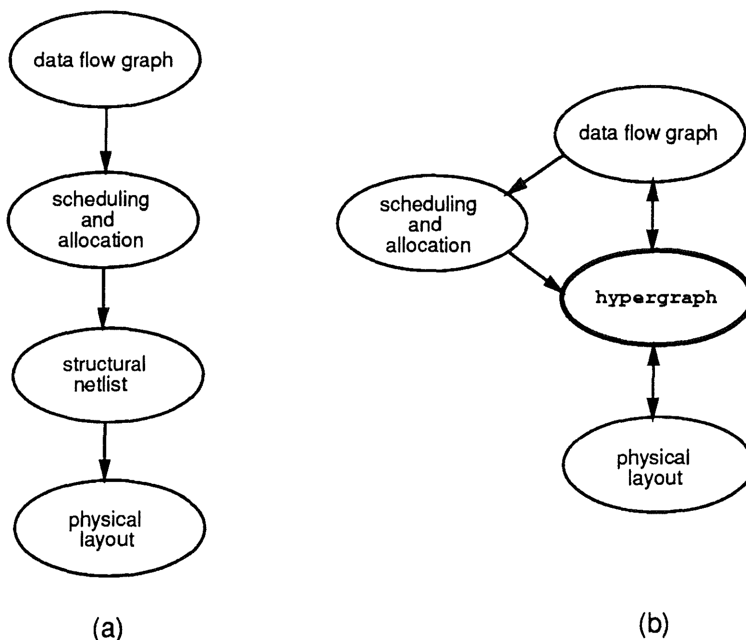


Figure 1: (a) Traditional data path synthesis, (b) Data path synthesis with feedback of physical information.

Much research has been reported for solving the scheduling and allocation problems ([8],[14],[19],[21],[22],[24],[12],[18]). Most of the research deals with scheduling and allocation separately. However, some approaches have also tried to perform the scheduling and allocation problems simultaneously [2]. Furthermore, very little syn-

thesis research has taken into account physical design effects ([4],[17],[29]). Critics point out that performing scheduling and allocation tasks without considering physical design effects results in impractical design. This has driven more recent interest in including layout information into the scheduling and allocation tasks. Since scheduling and allocation are interdependent, the results of one task will affect the other. On the other hand, the detailed physical layout information (including interconnect and register costs in terms of area and delay) can not be obtained until allocation process is completed. One practical approach [13] is to perform initial scheduling first, then retrieve physical layout information, and finally reschedule by taking into account physical design effects. This problem becomes even more difficult when manual or interactive synthesis is considered. A designer usually must find a control path and reassign several operations to different steps or to different execution units. Therefore, fast feedback is needed to assess consequences of such local changes.

Traditionally, data path synthesis performs in a straight line fashion as shown in Figure 1(a). The data path synthesizer first takes a data flow graph as input and performs scheduling and allocation tasks. Then, the synthesizer produces a structural netlist and passes it to a layout generator to produce the final layout. However, this methodology does not provide a feedback mechanism to back-annotate the layout information. In order to retrieve physical information, a feedback mechanism for data path synthesis is required.

In this paper, we describe a back-annotation technique to feedback physical information for a given data flow graph and its corresponding structural design. We use a hypergraph model which provides the mapping capability between the data flow graph and the corresponding physical design (Figure 1(b)). We describe the formulation from a structural netlist to a hypergraph. In addition, we describe the mapping between the data flow graph, the hypergraph, and the structural netlist. Furthermore, we describe the back-annotation procedure of physical layout including

area and delay. We also describe using the back-annotated information to estimate clock rates.

The remainder of this paper is organized as follows: Section 2 describes the relationships and mapping techniques between the data flow graph, the structure, and the hypergraph. Section 3 discusses the procedure of back-annotation of the physical design. Section 4 presents the experimental results. Finally, Section 5 summarizes our approach.

2 Formulation

In this section, we first describe the relationships between data flow graph and structure. Second, we describe the hypergraph model. Finally, we discuss the transformation procedure from the data flow graph to the hypergraph.

2.1 The relationships between DFG and structure

In general, a data-transfer-path can be viewed as follows: a functional unit fetches variables from a set of storage elements (registers, memories) via a set of interconnect units, performs the data computations, then stores the output variable back to the storage elements via a set of interconnect units. From the above observations, the data transfer between two operations in the data flow graph can be viewed as follows (Figure 2(a)): The function unit **FU1** generates the output (a variable) and stores in a register **reg** via a set of interconnect units. The functional unit **FU2** then fetches that variable via a set of interconnect units from the register **reg**. Thus, an edge in the data flow graph is equivalent to a path of **interconnect_unit1⇒reg⇒interconnect_unit2**.

For mapping the data flow graph to the data-tranfer-path, we introduce an in-

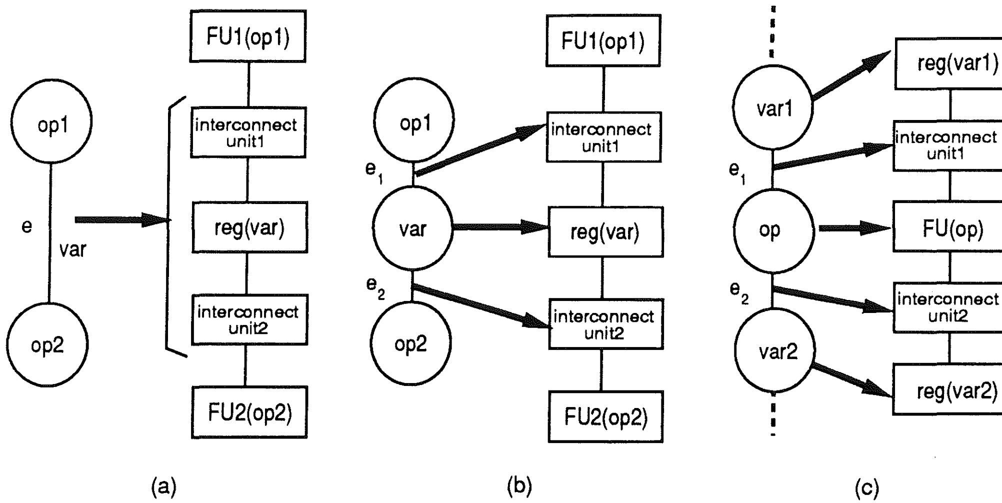


Figure 2: (a) The data flow between two operations and it's structural model, (b) The **var** node insertion, (c) The data transfer model.

termediate **var** node for each edge between two operations in the data flow graph as shown in Figure 2(b). By adding the **var** node, each edge is transformed to an **edge⇒var node⇒edge** path. For example, the edge **e** in Figure 2(a) is transformed into the $\mathbf{e_1}$⇒**var**⇒$\mathbf{e_2}$ path shown in Figure 2(b). Edge $\mathbf{e_1}$ corresponds to **interconnect_unit1** while edge $\mathbf{e_2}$ corresponds to **interconnect_unit2**. Using the intermediate nodes, the data-transfer model is shown in Figure 2(c). Each **var** node corresponds to a storage element, while each edge corresponds to an interconnect unit. For example, nodes **var1** and **var2** in Figure 2(c) correspond to **reg(var1)** and **reg(var2)** respectively, while $\mathbf{e_1}$ and $\mathbf{e_2}$ correspond to **interconnect_unit1** and **interconnect_unit2** respectively.

2.2 The hypergraph

Let $\mathbf{H} = (V, E)$ denote a hypergraph, $V = V_{io} \cup V_{reg} \cup V_{iug} \cup V_{op}$, in which there are four types of hypernodes: (i) input/output, (ii) register, (iii) interconnect, and (iv) operation. $V_{io} = \{v_k \mid k = 1..t\}$ is a set of i/o hypernodes which denote the input and output ports in the **DFG**. $V_{reg} = \{v_g \mid g = 1..u\}$ denotes a set of register hypernodes where each register hypernode contains a set of variables : $v_g = \{\mathrm{z_k} \mid \mathrm{k}=1..\mathrm{q}\}$. Each register hypernode denotes a register which contains a set of non-overlapping variables. $V_{iu} = \{v_f \mid f = 1..r\}$ denotes a set of interconnect hypernodes. Each interconnect hypernode denotes an interconnect unit such as multiplexer or bus. $V_{op} = \{v_c \mid c = 1..p\}$ denotes a set of operation hypernodes. Each operation hypernode denotes a particular functional unit such as adder/subtracter, multiplier, or shifter. Each operation hypernode contains a set of operation nodes in the **DFG** such as $v_c = \{\mathbf{v_i} \mid \mathrm{i}=1..\mathrm{n}\}$.

Let $E = \{e_{gc} \mid \{v_g, v_c\} \in V\}$, denote a set of hyperedges and $\mathbf{w}(e_{gc})$ denote the weight of e_{gc}. Each hyperedge represents the physical connection between two hypernodes. The weight of a hyperedge is the number of dependency edges between two hypernodes. This also can be viewed as the number of variables communicating between two hypernodes.

2.3 DFG-hypergraph formation

We are given a data flow graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$, a set of variables $\mathbf{Z}$, and a structural netlist with variable and operation assignments, where $\mathbf{V} = \{\mathbf{v_i} \mid \mathrm{i}=1..\mathrm{n}\}$ is a set of operation nodes, $\mathbf{E} = \{\mathbf{e_{jm}} \mid \{\mathbf{v_j}, \mathbf{v_m}\} \in \mathbf{V}\}$ is a set of dependency edges, and $\mathbf{Z} = \{\mathbf{z_k} \mid \mathrm{k}=1..\mathrm{q}\}$ is a set of variables in **DFG**, $\mathrm{z_k} = \{\mathbf{e_{jm}} \mid \{\mathbf{v_j}, \mathbf{v_m}\} \in \mathbf{V}\}$, where each edge corresponds to a variable. $\mathbf{V_{var}}$ is a set of **var** nodes. In addition, $\mathbf{G}' = (\mathbf{V}', \mathbf{E}')$, where $\mathbf{V}' = \mathbf{V} \cup \mathbf{V_{var}}$ and $\mathbf{E}' = \{\mathbf{e_{j'm'}} \mid \{\mathbf{v_{j'}}, \mathbf{v_{m'}}\} \in \mathbf{V}'\}$.

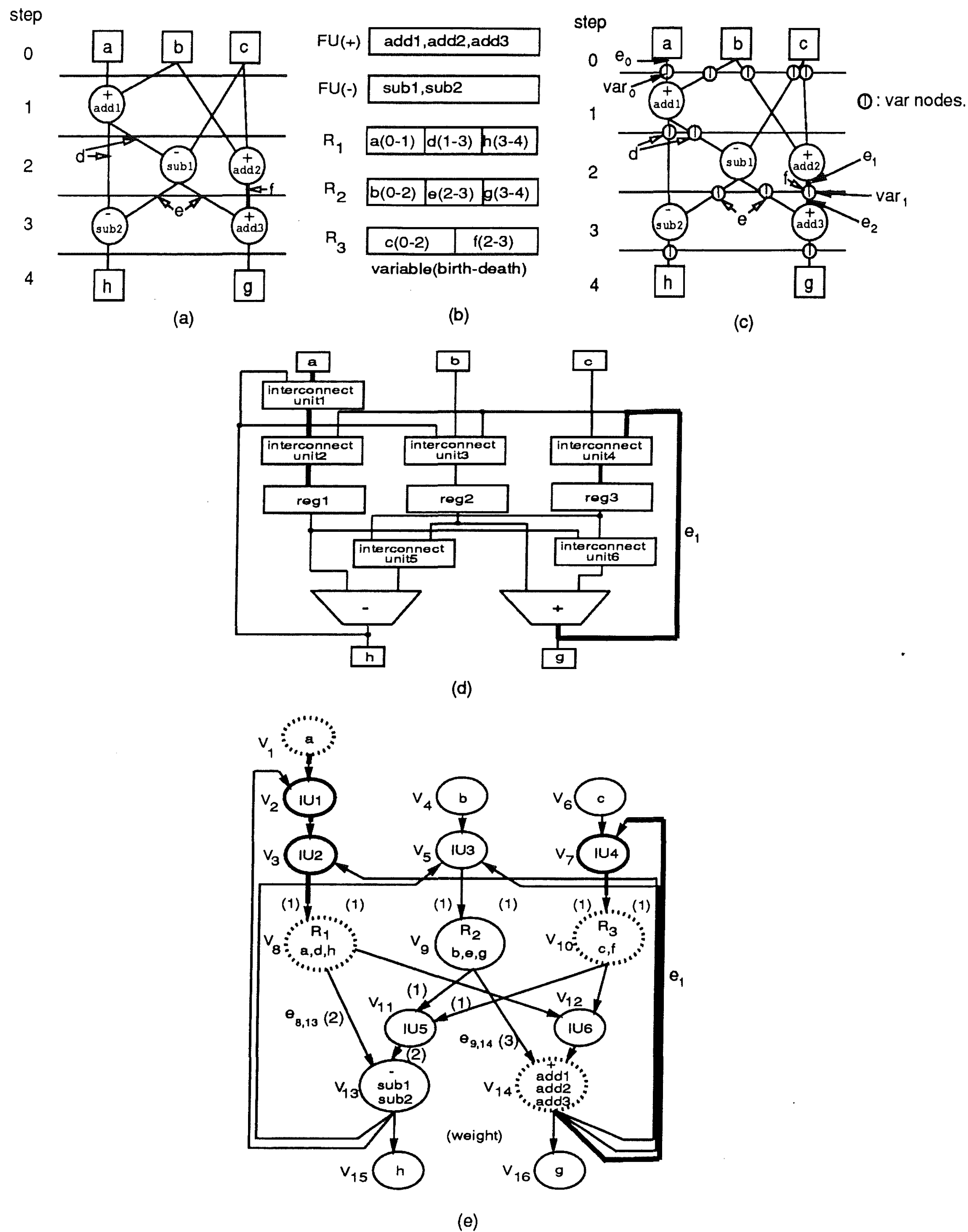


Figure 3: Hypergraph formation: (a) Data flow graph and schedule, (b) Variable and operation assignments, (c) **var** node insertion, (d) Structural netlist, (e) Hypergraph.

The transformation from a structural netlist to a hypergraph is a one-to-one mapping. The input/output ports, functional units, interconnect units, and registers are first mapped to a set of hypernodes. Each wire in the structural netlist is then mapped to a hyperedge. For example, Figure 3(e) shows a hypergraph which is mapped from the structural netlist in Figure 3(d).

The mapping from a data flow graph to a hypergraph consists of two steps: (1) operation and variable node mapping and (2) edge mapping. As described in the previous section, we first insert an intermediate **var** node on each edge in the DFG as shown in Figure 3(c).

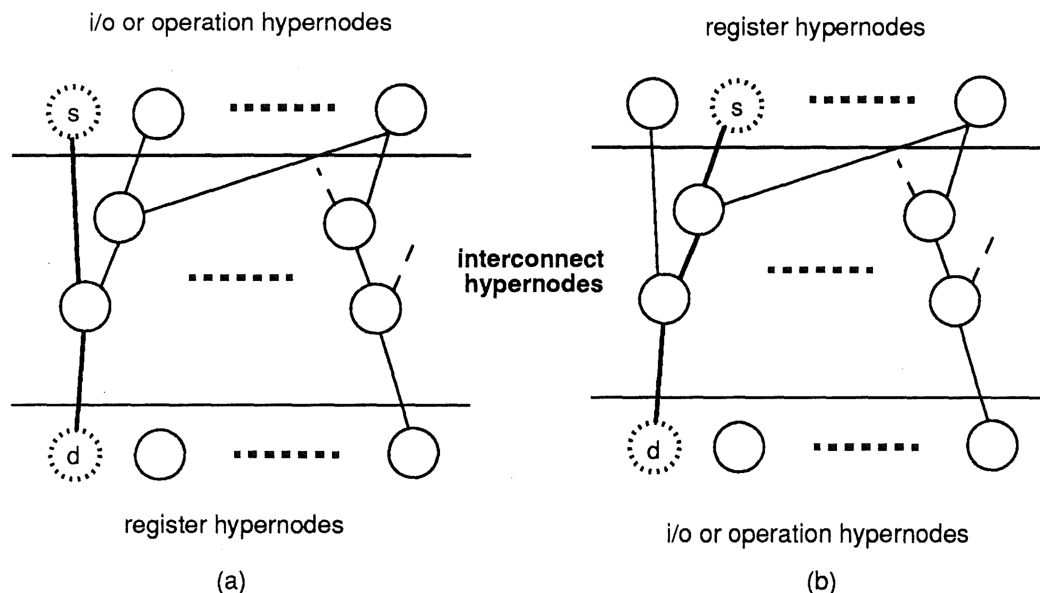


Figure 4: The edge mapping (a) case 1 and (b) case 2.

In the node mapping step, the variables and operations are assigned to their corresponding hypernodes. In addition, the variables and their corresponding register hypernodes are assigned to the corresponding **var** nodes in the DFG. For example, Figure 3(a) shows a data flow graph and its schedule. In this example, three

registers, one adder, and one subtracter are used to perform data transfer. The variable and operation assignments are shown in Figure 3(b). Operations **sub1** and **sub2** are assigned to the subtracter and operations **add1**, **add2**, and **add3** are assigned to the adder. The hypergraph shown in Figure 3(e) consists of a set of input/output hypernodes V_{io}=$\{v_1,v_4,v_6,v_{15},v_{16}\}$, and a set of three register hypernodes V_{reg}=$\{v_8,v_9,v_{10}\}$ where v_8 and v_9 consist of three variables, and v_{10} consists of two variables: v_8=$\{\mathbf{a},\mathbf{d},\mathbf{h}\}$, v_9=$\{\mathbf{b},\mathbf{e},\mathbf{g}\}$, and v_{10}=$\{\mathbf{c},\mathbf{f}\}$. In addition, there are two operation hypernodes V_{op}=$\{v_{13},v_{14}\}$ where v_{13}=$\{\mathbf{sub1},\mathbf{sub2}\}$, v_{14}=$\{\mathbf{add1},\mathbf{add2},\mathbf{add3}\}$. Furthermore, there are a set of interconnect hypernodes V_{iu}=$\{v_2,v_3,v_5,v_7,v_{11},v_{12}\}$.

In the edge mapping step, each edge is mapped to a set of interconnect hypernodes. The process of edge mapping consists of two cases: (1) from an operation or i/o node to a **var** node (Figure 4(a)) and (2) from a **var** node to an operation or i/o node (Figure 4(b)). Each edge mapping performs depth-first search from the source hypernode to the destination hypernode via a set of interconnect hypernodes. For example, the edge $\mathbf{e_1}$ between operation **add2** and **var** node $\mathbf{var_1}$ (Figure 3(c)) is mapped to v_7 (**interconnect unit4**) (Figure 3(d and e)). On the other hand, the edge $\mathbf{e_0}$ between input node **a** and **var** node $\mathbf{var_0}$ is mapped to $\{v_2, v_3\}$ since input node **a** stores variable **a** to register $\mathbf{R_1}$ via **interconnect unit1** and **interconnect unit2**.

Algorithm I. Hypergraph formation

Let

S=(N,W) be a given netlist where **N** is a set of components such as registers, interconnect units,i/o ports, and functional units and **W** is a set of wires;

hypergraph_formation(G,S,Z){

*/*hypergraph mapping*/*

H = hypergraph_mapping(**S**);

*/*variables and operations mapping*/*

var_op_mapping(**Z**,**V**,**H**);

*/*edge mapping*/*

for ($\mathbf{e_{jm}}$∈**E**){

```
        /*locate e_jm's corresponding variable*/
        z_k = find_variable(e_jm);
        /*locate the hypernode v_s such that v_j∈v_s*/
        v_s = find_hypernode(v_j);
        /*path searching from v_s to the hypernode v_d such that z_k∈v_d*/
        v_d = depth_first_search(v_s,z_k);
        /*insert var node v_p and map v_p to v_d*/
        insert_var_node(e_jm,v_d,v_p);
        /*map edge e_jm to a set of corresponding interconnect units*/
        interconnect_mapping(v_p, V_iu);
        v_s = v_d;
        /*locate the hypernode v_d such that v_m∈v_d*/
        v_d = find_hypernode(v_m);
        /*path searching from v_s to the hypernode v_d such that v_m∈v_d*/
        v_d = depth_first_search(v_s,v_m);
        /*map edge e_jm to a set of corresponding interconnect units*/
        interconnect_mapping(v_m, V_iu);
    }
}
```

Complexity analysis. The DFG-Hypergraph mapping consists of three procedures: (1) Hypergraph mapping, (2) Variable and operation mapping, and (3) Edge mapping. The complexity analysis of the three procedures is as follows:

(1). *Hypergraph mapping* takes $\mathbf{O}(V+E)$ time since the structure to hypergraph is a one-to-one mapping.

(2). *Variable and operation mapping* takes $\mathbf{O(V+E)}$ time, where $\mathbf{V}$ and $\mathbf{E}$ are the number of edges and the number of operations in the data flow graph respectively.

(3). *Edge mapping.* Using depth-first search, each edge mapping takes $\mathbf{O}(V+E)$ time. Since each edge is split into two edges by inserting a **var** node, the edge mapping takes $\mathbf{O}(\mathbf{2E}(V+E))$ time.

3 Back annotation of physical design

The main objective of the back annotation is to retrieve the physical design of a netlist back to its corresponding data flow graph. The back annotation consists of two parts: (i) Area and (ii) Delay. In this section, we first describe the layout and electrical

models. Then, we present the back annotation procedure. Finally, we describe the clock estimation using back-annotated physical information. In this paper, we only focus on the back-annotation of physical designs in data paths.

3.1 Layout model

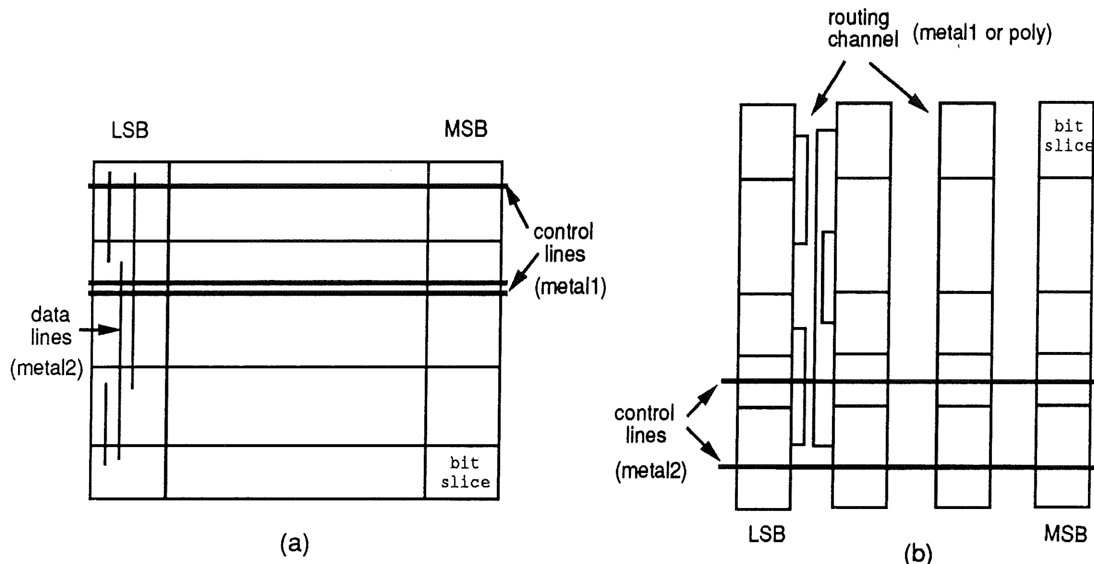


Figure 5: Two data path layout architectures

There are two common layout architectures for data paths: (1) bit slices with abutment and (2) macrocells with routing channel ([10],[16],[15],[1],[26]). The first layout architecture is shown in Figure 5(a), it uses abutment to connect different bit slices (in metal1) and over-the-cell routing (in metal2) to connect different units inside one bit slice. In this architecture, each bit-slice has the same width, but unit heights vary with the unit functionality. The stack grows horizontally when the bit-width increases, and vertically when the number of units increases. Data signals run vertically in second metal over the bit slices. Power, ground, and control lines are routed in the first metal or poly between the bit slices. In the second layout

architecture (Figure 5(b)), bit-sliced macrocells or standard cells of each bit slice are placed vertically and a routing channel is used for connecting different cells inside one bit slice (in metal1). Using either layout architecture, the area cost consists of four parts [32]: (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 obtained by examining the component library [28]. 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. Since the first architecture (abutment) has a fixed number of available over-the-cell routing tracks, if the required routing tracks are less than or equal to the available tracks then the wiring area is not needed. On the other hand, the second architecture needs a wiring area which is always proportional to the number of required tracks. In addition, we treat multiplier as a separate macroscell using our layout model. The overall area cost is:

$$\mathbf{A_{total}} = \mathbf{w}\ (\alpha\ (\sum_{\mathbf{i=1}}^{\mathbf{n}} \mathbf{trs(FU_i)} + \sum_{\mathbf{j=1}}^{\mathbf{m}} \mathbf{trs(REG_j)} + \sum_{\mathbf{k=1}}^{\mathbf{p}} \mathbf{trs(IU_k)}) + \mathbf{A_{wire}})$$

where

$\mathbf{A_{total}}$ is the area of the data path;
α is the transistor area coefficient correlating to the layout technology;
$\mathbf{trs(FU_i)}$ is the number of transistors in functional unit $\mathbf{i}$;
$\mathbf{trs(REG_j)}$ is the number of transistors in register $\mathbf{j}$;
$\mathbf{trs(IU_k)}$ is the number of transistors in interconnect unit $\mathbf{k}$;
$\mathbf{n}$ is the number of functional units;
$\mathbf{m}$ is the number of registers;
$\mathbf{p}$ is the number of interconnect units;
$\mathbf{A_{wire}}$ is the area of routing channel;
$\mathbf{w}$ is the bit widths of the data path.

3.2 Delay model

We divide the delay calculation into two parts: (i) Component delay and (ii) Path delay.

Component delay. The delay of a component is due to the internal and the total load capacitances. The load capacitances of a component consists of two elements: (i) the input capacitances of the fanout components and (ii) the total wire capacitances connected from the component to its fanout components. The total delay of a component **i** is:

$$\mathbf{d_{total(i)}} = \mathbf{d_{int(i)}} + \mathbf{cC_{load(i)}}$$

and

$$\mathbf{C_{load(i)}} = \mathbf{C_{wire}} + \sum_j \mathbf{C_{in(j)}}, \ \mathbf{j} \in \text{fanout components}$$

where

$\mathbf{d_{total(i)}}$ is the total delay of component **i**;
$\mathbf{d_{int(i)}}$ is the delay due to component internal capacitances;
c is a constant;
$\mathbf{C_{load(i)}}$ is the total capacitive load of component **i**;
$\mathbf{C_{wire}}$ is the interconnect capacitances;
$\mathbf{C_{in(j)}}$ is the input capacitances of component **j**.

Path delay. The path delay is the total delay from a source to a destination in a data path. The delay of a path **k** is:

$$\mathbf{D_{total(k)}} = \sum_m \mathbf{d_{total(m)}}, \ \mathbf{m} \in \text{the components along the path } \mathbf{k}$$

where

$\mathbf{D_{total(k)}}$ is the total delay of path **k**.

3.3 The back-annotation procedure

The back-annotation procedure consists of four steps described as follows:

(1). *Hypergraph formation.* We are given a data flow graph **G**, a structural netlist **S** with variable and operation assignments. The algorithm first performs hypergraph transformation as described in Section 2.3.

(2). *Stack placement.* The algorithm uses the min-cut partitioning to perform stack placement, and then uses the left-edge algorithm to assign routing tracks.

(3). *Delay calculation.* The algorithm calculates the wire length for each net and the capacitive load and delay for each unit. The delay of each unit is the unit logic delay plus the delay from the unit's driven loads using the previously described delay model.

(4). *Back annotation.* The algorithm back-annotates the delay information to each node and edge in the data flow graph. For example, the delay of $\mathbf{e_0}$ in Figure 3(c) is the sum of the delays of components **interconnect unit1** and **interconnect unit2** (Figure 3(d)) while the delay of the operation **add2** is equal to the delay of the adder.

Using the back-annotated information, the algorithm can also performs clock estimation. The delay of each data transfer is calculated using the **reg**$\Rightarrow$ **interconnect_unit** $\Rightarrow$ **FU** $\Rightarrow$ **interconnect_unit** $\Rightarrow$**reg** model as shown in Figure 2(c). Using this data transfer model, the data transfer of an operation $\mathbf{op_i}$, $\mathbf{i= 1..q}$ in the data flow graph can be formulated as follows (Figure 6): the function unit fetches input data from n registers via interconnect units $\mathbf{IU_{in_1}..IU_{in_n}}$, performs computation, and stores data to registers via interconnect units $\mathbf{IU_{out_1}..IU_{out_m}}$. Thus, the delay of operation $\mathbf{op_i}$ is:

$$\mathbf{delay(op_i) = Max(delay(IU_{in_j})) + delay(FU) + Max(delay(IU_{out_k}+delay(reg_k)))}$$

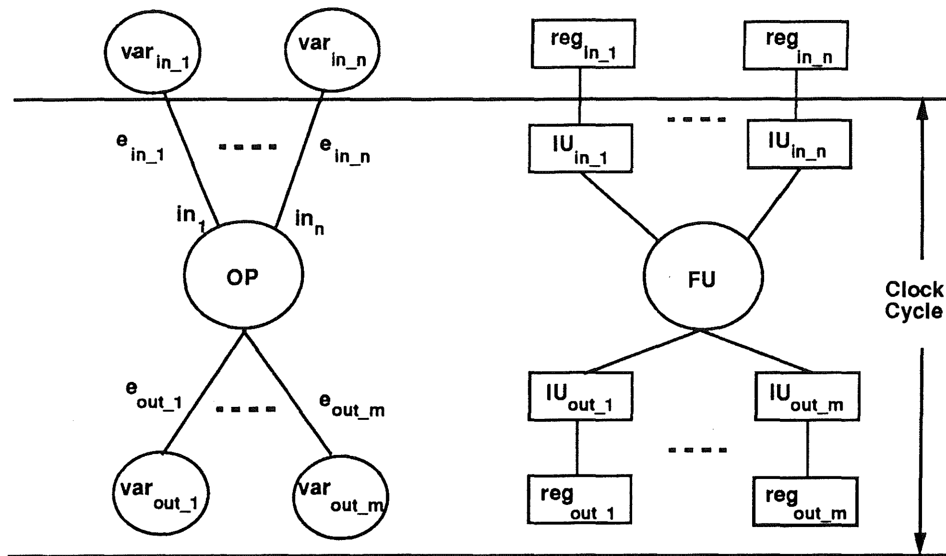


Figure 6: The clock estimation model.

where $\mathbf{j=1..n}$ and $\mathbf{k=1..m}$

and the minimal clock cycle for this design is:

$$\mathbf{Clock = Max(delay(op_i)),\ i = 1..q}$$

Algorithm II. Back Annotation

```
Let
   delay(v_i) be the delay of node v_i;
   V_interconnect be a set of interconnect hypernodes;
Back_Annotation(G,S,Z){
   {H, G'} = hypergraph_formation(G,Z,S);
   stack_place_route(H);
   unit_delay_calculation(H);
   /*back_annotation*/
   for (v_i∈V'){
      /*locate hypernode v_s where v_i ∈v_s*/
      v_s = find_hypernode(v_i);
      delay(v_i) = delay(v_s);
   }
   for (e_j'm'∈E'){
```

```
        /*locate interconnect hypernodes associated with e_j'm' */
        V_interconnect = find_hypernode(e_j'm');
        delay(e_j'm') = Σ_k delay(v_k) where v_k ∈ V_interconnect;
    }
    /*clock estimation*/
    for (v_m ∈ V_op){
        delay(v_m) = delay_calculation(v_m);
    }
    clock = Max{delay(v_m) | v_m ∈ V_op};
}
```

complexity analysis. The complexity analysis consists of five parts:

(1). *Hypergraph formation.* Described in the previous section.

(2). *Stack placement.* Our algorithm performs stack placement using the KLFM ([5],[11]) partitioning algorithm to minimize routing density. Using the bucket list data structure [5] which has the complexity of $\mathbf{O(p)}$ time where $\mathbf{p}$ is the number of pins in the netlist. In addition, the algorithm performs routing track assignments using the left-edge algorithm. The complexity of routing track assignments takes $\mathbf{O(mlogm)}$ time where $\mathbf{m}$ is the number of nets in the netlist.

(3). *Delay calculation.* It takes $\mathbf{O}(E)$ time to calculate the wire lengths of nets. In addition, it takes $\mathbf{O}(V)$ time to calculate delays for all components.

(4). *Back annotation.* Since each edge in the data flow graph is divided into a **edge⇒var-node⇒edge** path, it takes $\mathbf{O(V+3E)}$ time to annotate physical information to the data flow graph.

(5). *Clock estimation* takes $\mathbf{O}(V)$ time.

4 Experiments and results

In this section, we first present a walk-through example showing how to back-annotate the physical information using the Figure 3 example. Next, we show an application example using the back-annotation technique.

4.1 A walk-through example

We assume the data path bit-width of this example is eight. The final layout (area = 1040μm X 1960μm) shown in Figure 7 is generated by [31]. In the delay calculation, we only take into account the data path delay. The back-annotation of the delay information consists of five steps as follows:

(1). The algorithm performs hypergraph transformation and stack placement.

(2). After performing the placement, the algorithm calculates the wire length for each connection. For example, $\mathbf{c_9}$ (Figure 8(a)) is connected from **mux1** to **reg1**, is 230μm.

(3). The algorithm calculates component delay by taking into account wire capacitance. The algorithm first calculates the total capacitive load for each component. For example, the capacitive load of **reg1** in Figure 8(a) is the sum of the wire capacitance of $\mathbf{c_{11}}$, the input capacitance of **mux5**, and the input capacitance of the subtracter. Then, the algorithm calculates the component delay using the described delay model. For example, the delay of **reg1** is 5.8ns.

(4). The algorithm retrieves delay for each edge of the data flow graph. For example, the delay of edge $\mathbf{e_1}$ (Figure 8(b)) between input node **a** and **var** node **var1** is 10ns which is equal to the delays of **mux0** and **mux1**.

(6). Clock estimation. The delay of operation **add1** in Figure 8(b) is $\mathbf{Max(delay(e_2), delay(e_{10}))+delay(adder)+Max((delay(e_3) + delay(var_6)), (delay(e_4) + delay(var_7)))}$ such that **delay(add1)=Max**(4.8, 0)+22+**Max**((5.3+5.8), (5.3+5.8))=37.3ns. The delays of operations **add2, add3, sub1,** and **sub2** are 38.2ns, 37.3ns, 36.5ns, and 41.8ns respectively. As a result, the minimal clock cycle for this design is 41.8ns.

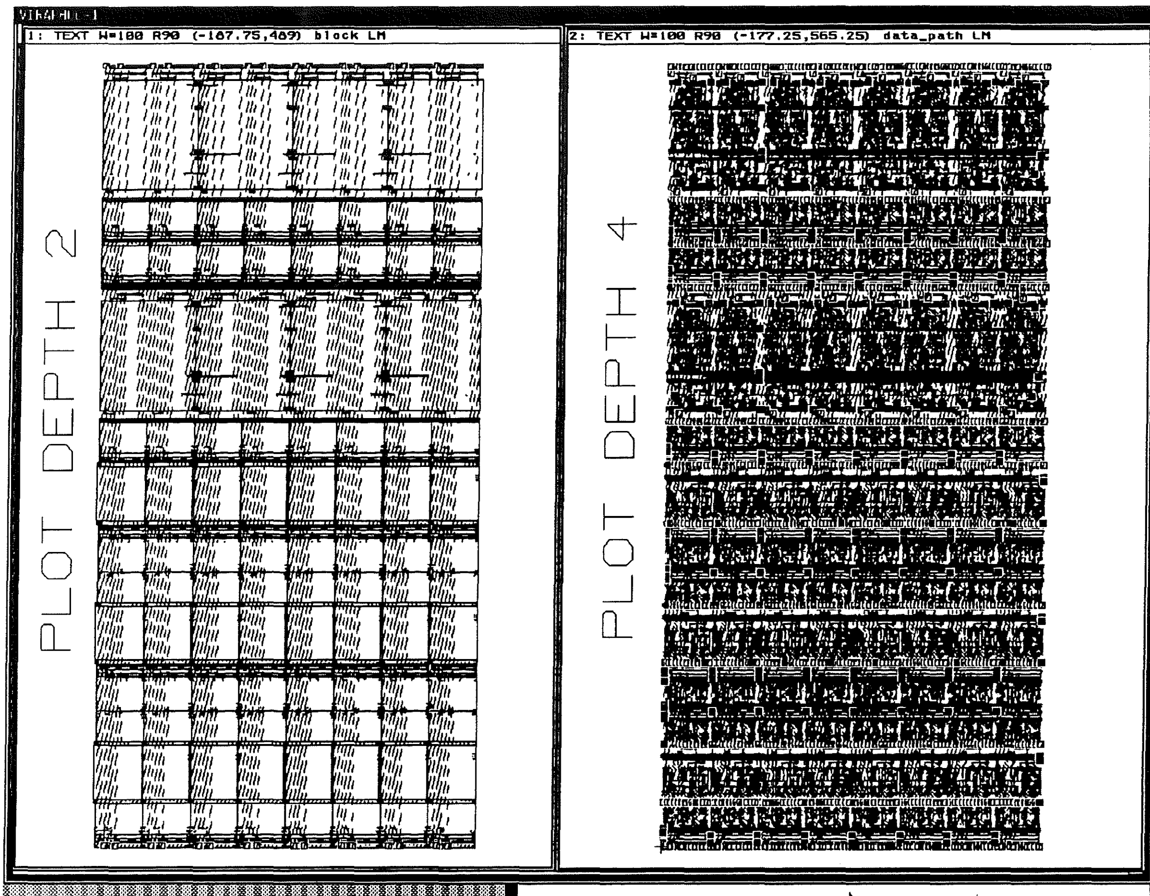
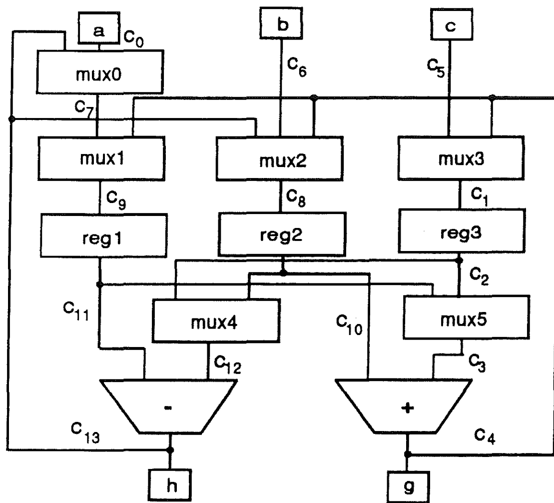


Figure 7: The layout of the Figure 2. example (a). Routing track assignments and (b). The final layout.

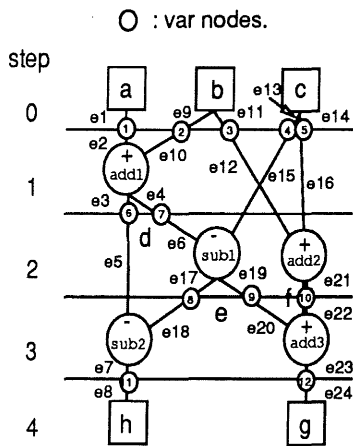


connection	wire length(um)
c0	130
c1	130
c2	1,040
c3	260
c4	2,040
c5	10
c6	660
c7	280
c8	230
c9	230
c10	410
c11	1,160
c12	500
c13	1,760

component	delay(ns) *
mux0	5.0
mux1	5.0
mux2	5.3
mux3	5.2
mux4	5.0
mux5	4.8
reg1	5.8
reg2	5.2
reg3	5.7
-	21.0 **
+	22.0 **

*: Delay calculation by taking into account wire capacitance.
**: Delay with 8-bit component.

(a)



connection	delay(ns)
e1	10.0
e2	4.8
e3	5.3
e4	5.3
e5	0.0
e6	0.0
e7	10.0
e8	0.0
e9	5.3
e10	0.0
e11	5.3
e12	5.3
e13	5.2
e14	5.2
e15	5.0
e16	4.8
e17	5.3
e18	5.0
e19	5.3
e20	0.0
e21	5.2
e22	4.8
e23	5.3
e24	0.0

var-node	delay(ns)
1	5.8
2	5.2
3	5.2
4	5.7
5	5.7
6	5.8
7	5.8
8	5.2
9	5.2
10	5.7
11	5.8
12	5.2

(b)

Figure 8: (a) Back-annotation of wire lengths and component delays and (b) Back-annotation of delay information to the DFG.

4.2 An application example

In exploring the design space, it is possible to tradeoff the registers and interconnect units. Back-annotation allows us to select the best solution that satisfies the given constraints. Using the hypergraph model, we first used an allocation algorithm [30] to perform registers and interconnect units tradeoffs. Then, we back-annotated the physical information using the described back-annotation procedure. We applied two single-level interconnect models, multiplexer and bus.

Table 1 shows the results of the 19-step Elliptic Filter example with a 2-adder 1-piped multiplier. Figure 9 shows the layout of the Elliptic Filter example which was generated using [31]. Using the multiplexer model, the results show that the design with 11 registers and 6 multiplexers produces the minimal area. Using the bus model, the design with 13 registers and 5 multiplexers produces the minimal design. For 4-bit, 8-bit, and 16-bit designs, the design with 13 registers and 5 multiplexers produces the shortest delay using either multiplexer or bus model (Figure 10(a) (b) (c)). However, for the 32-bit design, the designs with 11, 12, and 13 registers have the same delay for both interconnect models (Figure 10(d)).

5 Conclusions

We have proposed a layout back-annotation technique for data path synthesis. The use of a hypergraph provides a feedback mechanism for back-annotation of the physical design. We have described a back-annotation procedure to retrieve the detailed physical information including area and delay. Given a data flow graph and its structural design, this back-annotation technique can not only evaluate the design quality but can also feedback the delay to each edge and node in the data flow graph. This allows designers to perform scheduling/allocation by identifying the critical paths and improving the design. Using this back-annotation technique, we can perform the

The 19-step Elliptic Filter Example with 2-adder and 1-piped multiplier					Clock (ns) (Mux/bus)			
# of Reg.	# of Mux.	# of Mux i/ps	Area (sq. um / bit) (multiplier)	Area (sq. um / bit) (Mux/bus)	4-bit	8-bit	16-bit	32-bit
10	10	36	145,680	125,376/148,896	31.4/25.0	37.4/31.0	54.4/48.0	94.3/91.0
11	6	28	145,680	113,136/133,536	29.4/23.2	35.4/29.2	52.4/46.3	90.3/88.3
12	6	26	145,680	115,696/132,576	29.7/23.8	35.7/29.8	52.7/46.8	90.3/88.3
13	5	23	145,680	113,696/129,216	25.2/22.4	31.2/28.4	48.3/46.3	90.3/88.3

Table 1. The results of the 19-step, 2-adder, 1-piped multiplier Elliptic Filter example.

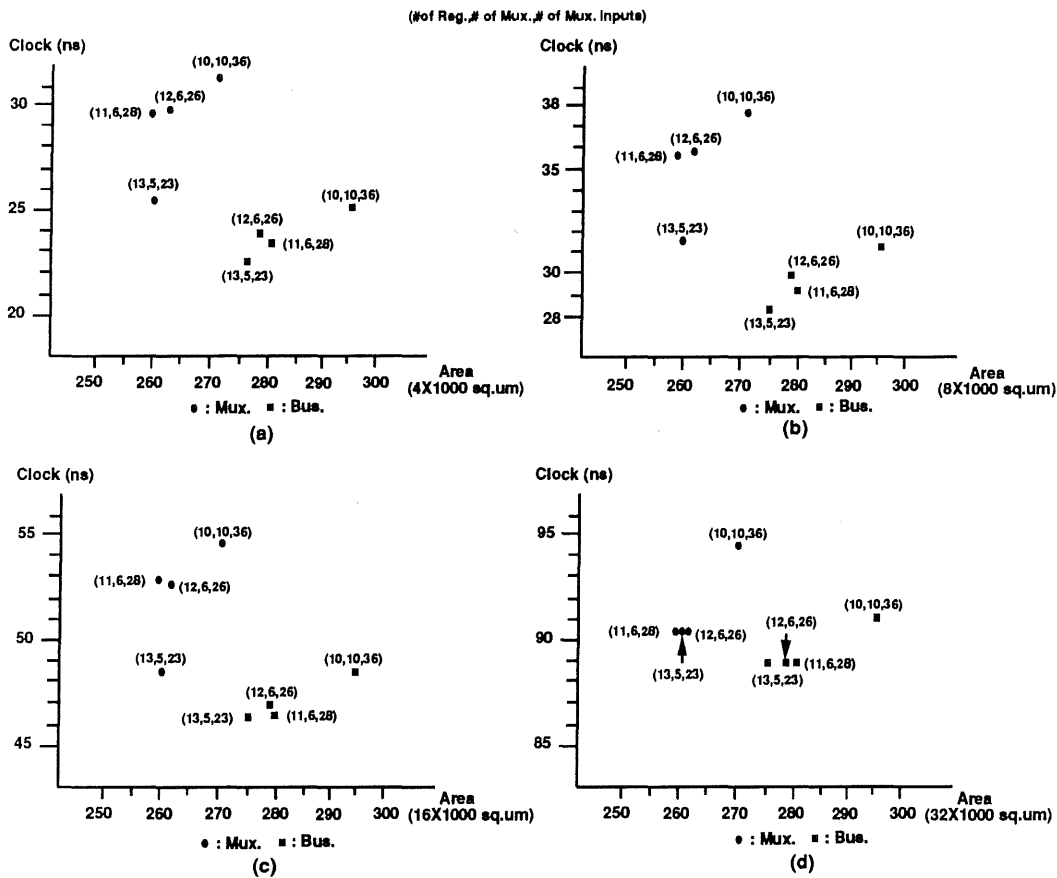


Figure 10: The Area-Clock tradeoffs of the Elliptic Filter example: (a) 4-bit, (b) 8-bit, (c) 16-bit, and (d) 32-bit.

References

[1] H. Cai, S. Note, P. Six, and H. De Man, "A data Path Layout Assembler for High Performance DSP Circuits," *Proc. 27th DAC.*, pp.306-311, 1990.

[2] R. J. Cloutier and D. G. Thomas, "The Combination of Scheduling,Allocation, and Mapping in a Single Algorithm," *Proc. 27th DAC*, pp. 71-76, 1990.

[3] H. De Man, et. al., "CATHEDRAL II - A Computer-Aided Synthesis System for Digital Signal Processing VLSI Systems", *IEE CAE Journal*, April 1988.

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

[5] C. M. Fiduccia and R. M. Mattheyses, "A Linear-Time Heuristic for Improving Network Partitions," *Proc. 19th DAC*, pp. 175-181, 1982.

[6] 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.

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

[8] C. T. Hwang, Y. C. Hsu, and Y. L. Lin., "Optimum and Heuristic Data Path Scheduling," *Proc. 27th DAC*, pp. 65-70, June 1990.

[9] K. S. Hwang, A. Casavant, et. al., "Scheduling and Hardware Sharing in Pipelined Data Paths", *Proc. IEEE Intl. Conf. on Computer-Aided Design*, Nov. 1989.

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

[11] K. H. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell System Technical Journal*, vol. 49, no. 2, pp. 291-307, Februry, 1970.

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

[13] D. W. Knapp, "Feedback-Driven Datapath Optimization in Fasolt," *Proc. IEEE Intl. Conf. on Computer-Aided Design*, 1990.

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

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

[16] Luk, W. K. and Dean, A. A., "Multi-Stack Optimization for Data-Path Chip (Microprocessor) Layout," *Proc. 26th DAC*, pp.110-115, 1989.

[17] M. C. McFarland., "Using Bottom-Up Design Techniques in the Synthesis of Digital Hardware from Abstract Behavioral Descriptions," *Proc.23th DAC*, June, 1986.

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

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

[20] N. Park, and A. Parker, "Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications", *IEEE Trans. on Computer-Aided Design*, vol. CAD-7 no. 3, March 1988.

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

[22] 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.

[23] 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.

[24] R. Potasman et. al., "Percolation Based Synthesis," *Proc. 27th DAC*, pp. 444-449, 1990.

[25] D. Thomas, et. al., "Methods of Automatic Data Path Synthesis", *IEEE Computer*, December 1983.

[26] M. T. Trick and S. W. Director, "Lassie: Structure to Layout for Behavioral Synthesis Tools," *Proc. 26th DAC.*, pp.104-109, 1989.

[27] 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.

[28] "Data path Library," VLSI Technology, INC., 1988.

[29] J. P. Weng and A. C. Parker, "3D Scheduling: High-Level Synthesis with Floorplanning," *Fifth International Workshop on High-Level Synthesis*, pp.1-7, 1991.

[30] Allen C-H Wu and D. D. Gajski, "Layout-Driven Allocation for Data Path Synthesis," Tech. Rpt. No. 91-30, ICS Dept., UC Irvine, 1991.

[31] Allen C-H Wu, G. D. Chen and D. D. Gajski, "Silicon Compilation from Register-Transfer Schematics," *Proc. ISCAS*, pp.2576-2579, 1990.

[32] Allen C-H Wu, Viraphol Chaiyakul and D. D. Gajski, "Layout Models for High-Level Synthesis," Tech. Rpt. No. 91-31, ICS Dept., UC Irvine, 1991.

3 1970 00882 4358