A generic binding model for concurrently optimizing interconnection and functional units

En-Shou Chang and Daniel D. Gajski

Technical Report #97-42
October 10, 1997

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425
(714) 824-8059

echang@ics.uci.edu

Abstract

A generic binding model which can provide complete information for binding task is presented in this paper. With information provided by this model, people can do interconnection optimization concurrently with operation and variable binding, as well as find better combination of FUs. Our method is not only much faster but also can obtain better or competitive binding than complex previous works.

Categories and Subject Descriptors: B.5.2 [**Hardware**]: REGISTER-TRANSFER-LEVEL IMPLEMENTATION

Key Word: binding, allocation, source exchange

1 Introduction

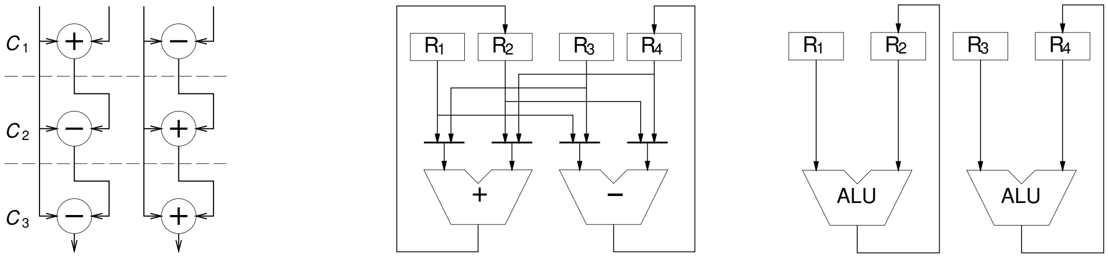
In recent years, **behavioral (high-level) synthesis**[1, 2] has been recognized as one of the major design methodologies. It allows us to specify a design in a purely behavioral form, devoid of any implementation details. For example, we can describe a design using Boolean equations, finite-state machines and other conceptual models, then an implementation for the design can be generated by automatic synthesis tools, instead of tedious manual design.

Behavioral synthesis involves the transformation of a design specification into a set of interconnected **RT-components**[2] which can perform the specified behavior and satisfy the specified constraints, such as area, performance, power, etc. To achieve these goals, three major synthesis tasks are applied during the transformation[1]: allocation, scheduling, and binding. The **allocation** determines the number of resources, such as registers, buses, and functional units(FUs), which will be used in the implementation. The **scheduling** partitions the behavioral description into time intervals, called **control steps**. During each control step, data are fetched from some registers, transformed in FUs, and written back to some other registers. The **binding** assigns variables to storage units, assigns operations to FUs, and makes sure that there are distinct communication paths assigned for every datum transferred between the storage and FUs. Here we define a **data-transfer** as the movement of a set of data transferred from a storage unit to an input port of an FU or from an output port of an FU to a storage unit.

In this paper, we assume the binding task is performed after scheduling is done. The binding task not only assigns the operations, variables, and data-transfers to hardware components such that the hardware can perform the specified behavior correctly, but also makes the assignment toward a better design quality, such as smaller hardware cost(area), lower power consumption, higher performance, or a combination of some of the aboves. Figure 1 shows an example of how the binding task affects design quality. We assume area is the major design quality concerned here. Figure 1(b) and Figure 1(c) are two possible implementations for the scheduled control/data-flow graph(CDFG) in Figure 1(a). Either Figure 1(b) or Figure 1(c) can be the optimal result. For example, under technology 1 shown in Figure 1(d), Figure 1(c) is the optimal result; on the other hand, under technology 2, Figure 1(b) is the optimal result.

Owing to the complexity of the binding task, most early synthesis systems[3, 4, 5, 6, 7, 8, 9, 10] decompose the binding task into several phases or subtasks, for example, operation-to-FU assignment, variable-to-storage assignment, and interconnection optimization, then find the optimal solution or a near-optimal solution in each phase or subtask. However, since there are inter-dependencies among these subtasks, no optimal solution is guaranteed even if all of the subtasks are solved optimally. Some other synthesis systems[11, 12, 13, 14] start with an empty datapath and build the datapath gradually by adding FUs, storages, or interconnections as necessary. However, their algorithms can't foresee better consequences, thus these constructive approaches often lead to not so good results. A time-consuming **iterative improvement**[2] phase is usually required by these systems[13].

Integer Linear Programming(ILP) model[15, 16, 17, 18] is the only way to find the optimal solution for the binding task. However, since an ILP problem is NP-complete[19], it would be very time-consuming to solve a large design using ILP model[18]. On the other hand, some researchers[20, 21, 22, 23] proposed heuristic methods which can find good enough implementations in reasonable time. They formulate the binding



(a) Scheduled CDFG **(b)** Implementation 1 **(c)** Implementation 2

	Tech 1	Tech 2
register	10	10
select bit	8	5
adder	20	20
subtractor	25	27
ALU	27	36

(d) Component area of two technologies

	Imp. 1	Imp. 2
Tech 1	117	94
Tech 2	107	112

(e) Area of each implementation

Figure 1: An example of binding

task as a 2-D placement problem. However, these 2-D placement models have to start with *pre-determined resource allocation*. Thus, some alternative implementations, for example, using ALUs to replace adders and subtractors or vise versa as shown in Figure 1 can not be evaluated to find a better implementation.

In this paper, we present a generic binding model which explicitly expresses the relation among variables, operations, and data-transfers, as well as the relation among partially bound data structure and components. Our model can provide following advantages:

- Starts *without* pre-determined resource allocation, where all possible implementations can be explored.

- Binds operations, variables, and data-transfers concurrently to globally search the best implementation.

- Works for arbitrary architectures, accepts arbitrary component libraries, and can be employed by most algorithms.

The rest of this paper is organized as following: We define our binding model in Section 2. Then, we illustrate how popular design rules are formulated by our binding model in Section 3. A simple binding algorithm and a source exchange algorithm based on our binding model are proposed in Section 4 and Section 5. Our model is so powerful that even a simple algorithm can produce competitive or better results than previous complex binding algorithms. The experimental results are shown in Section 6. Finally, we conclude our contributions in Section 7.

2 The binding model

In general, the binding task tries to find an assignment with best design quality for variables, operations, and data-transfers to implement a scheduled CDFG. To perform the binding task, we first transform the scheduled CDFG into a register-transfer flow

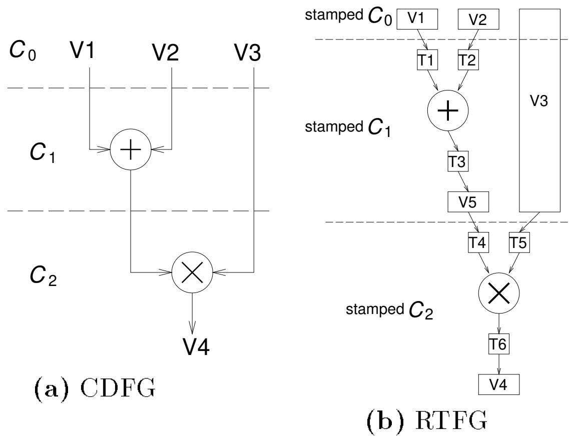


Figure 2: A scheduled CDFG and its corresponding RTFG

graph(RTFG) which explicitly describes the relation among variables, operations, and data-transfers, then group those vertexes which will be bound to the same hardware component into one cluster. The RTFG is defined as following:

Definition 1 *A* **register-transfer flow graph (RTFG)** *is a directed graph $\langle V, E\rangle$ where*

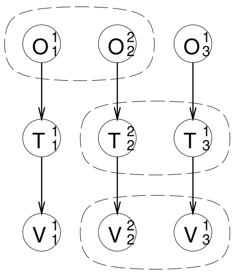
1. *A vertex $v \in V$ represents an operation, a variable, or a data-transfer.*
2. *An edge $e \in E$ indicates a relation between a sink of a vertex and a source of another vertex.*
3. *Each vertex is associated with one or more clock stamps indicating its life time.*

Figure 2 shows an example of the transformation from a scheduled CDFG to its corresponding RTFG: Figure 2(a) is a CDFG which has two operations($+$ and $\times$) and four input/output variables(V1, V2, ... V4). Including the initial control step, it is scheduled into three control steps(C_0, C_1, and C_2). Figure 2(a) is transformed into a 13-vertex RTFG as Figure 2(b). In Figure 2(b), vertex $+$ and vertex $\times$ represent operations, vertexes V1, V2, ... V5 represent variables, and vertexes T1, T2, ... T6 represent data-transfers. Each operation or variable in Figure 2(a) is transformed into a vertex associated with the corresponding clock stamps in Figure 2(b). For examples, the operation $+$ in Figure 2(a) is transformed into the vertex $+$ associated with a clock stamp C_1 in Figure 2(b); the variable V3 in Figure 2(a) is transformed into the vertex V3 associated with clock stamps C_0 and C_1 in Figure 2(b). In addition, vertexes represent temporary variables are added into the RTFG, for example, V5 associated a clock stamp C_1 is added into the RTFG. Finally, each data-transfers between variables and operations is transformed into a vertex associated with a corresponding time stamp. For example, the data-transfer from variable V3 to operation $\times$ in Figure 2(a) is transformed into the vertex T5 associated with a clock stamp C_2 in Figure 2(b).

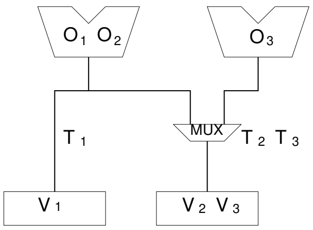
Once the scheduled CDFG is transformed into a RTFG, we can formulate the binding task as a **clustering problem** where we cluster all of the vertexes in the RTFG into clusters such that vertexes which are clustered together are bound to the same hardware component. In most cases, this formulation implies:

1. Vertexes associated with same clock stamps can not be clustered together, since a hardware component can't perform more than one job at the same time.
2. Operations can not be clustered together if there is no FU which can execute all of those operations available.

However, *there are some exceptions.* For example, the component library has a super-scalar FU which can execute two additions concurrently, then two addition-vertexes associated with the same clock stamp can be clustered together.

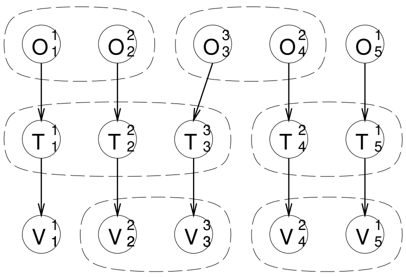


(a) RTFG clustering

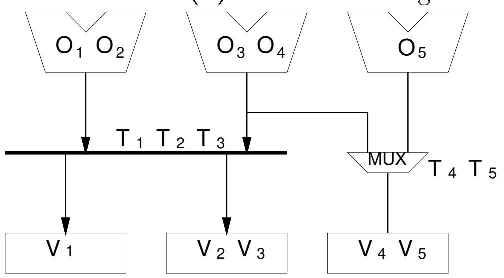


(b) Corresponding implementation

Figure 3: Design using single-level MUX



(a) RTFG clustering



(b) Corresponding implementation

Figure 4: Design using both MUX and bus

3 Some paradigms

Our binding model can work for arbitrary architectures, accept arbitrary component libraries, and be employed by arbitrary algorithms. Any design rules can be translated into corresponding clustering rules. We now illustrate how popular design rules are formulated in our binding model below.

3.1 Interconnection style

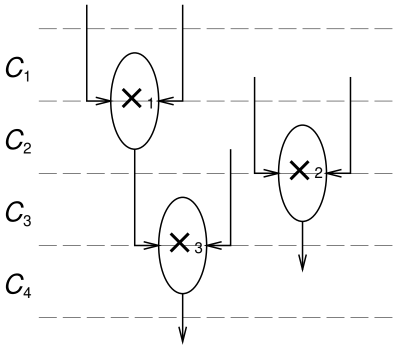
In case the design rule allows using only **one-level MUX interconnection** between registers and FUs, data-transfer vertexes should be clustered according to their successors. Figure 3 shows an example of MUX-base design: O_i is operation vertex, T_i is data-transfer vertex, and V_i is variable vertex. The upper-right index is clock stamp. T_2 and T_3 are clustered after V_2 and V_3 have been clustered.

On the other hand, when the design rule uses both **MUX and bus interconnection**, any data-transfer vertexes whose clock stamps don't conflict with each other can be clustered together to form an interconnection circuit. A data-transfer cluster with only one successive cluster is bound to a MUX; otherwise, bound to a bus. Figure 4 shows an example of this interconnection style.

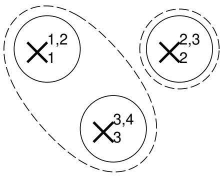
Moreover, **generalized interconnection model**[6] developed by Ly et. al. can also be implemented in our model. In this case, any data-transfer vertexes can be clustered together even they have same clock stamps. We can cluster any data-transfer vertexes together if the clustering is beneficial, under generalized interconnection style.

3.2 Pipeline FU

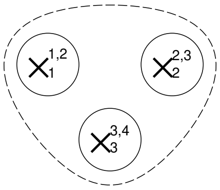
We use Figure 5 to show how pipeline components are interpreted in our model. In case there is no pipeline multiplier in the component library, two multiplication vertexes which have coincident clock stamps can not be grouped into a cluster. For example, three operation vertexes from the scheduled CDFG in Figure 5(a) can only be clustered as in Figure 5(b) if there is no pipeline multiplier available. On the other hand, in case there are pipeline multipliers in the component library, two multiplication vertexes which have coincident clock stamps can be grouped into a cluster as long as their first clock stamps are different. For example,



(a) A scheduled CDFG



(b) Operation clustering in case of no pipeline multiplier



(c) Operation clustering in case of pipeline multiplier available

Figure 5: Clusterings with and without pipeline FU

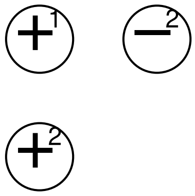
three operation vertexes for the scheduled CDFG in Figure 5(a) can be clustered as in Figure 5(c) if there are pipeline multipliers available.

3.3 Multi-function unit

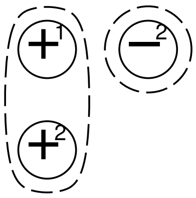
Our model allows using multi-function units, for example, implementing an addition by using an adder, add-subtractor, or ALU. Figure 6 shows an example of using multi-function units. In case there is no multi-function unit in the component library, an RTFG as shown in Figure 6(a) is clustered as in Figure 6(b). On the other hand, in case there are add-subtractors available, the RTFG shown in Figure 6(a) can be clustered either as in Figure 6(b) or as in Figure 6(c), depend on whose cost is smaller.

3.4 Multiclock and chaining

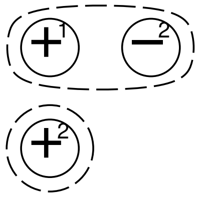
Our model can adopt multiclock and chaining. Once an operation is scheduled to be executed in more than



. **(a)** Part of an RTFG



(b) Clustering 1



(c) Clustering 2

Figure 6: Two clusterings

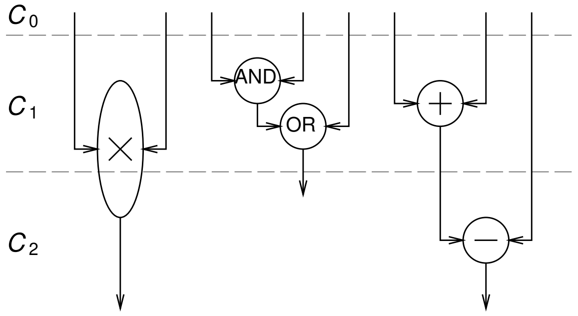
one clock cycles, more than one clock stamps are attached to the corresponding operation vertex in RTFG. On the other hand, once two operations are scheduled to be chained into one clock, the same clock stamp is attached to both corresponding operation vertexes and no variable vertexes are inserted between two operation vertexes. Figure 7 shows an example of multiclock and chaining in our binding model.

3.5 Custom-built components

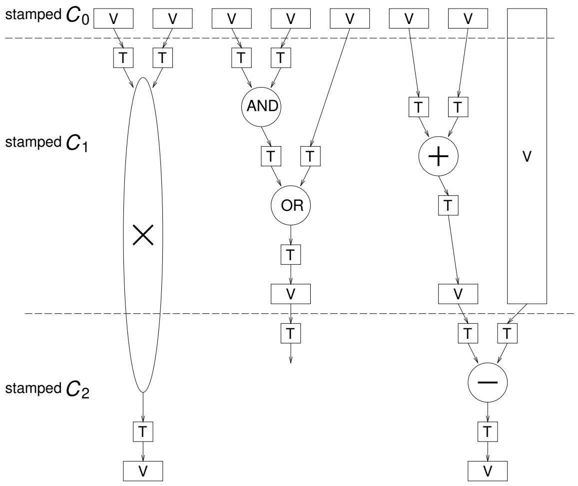
Special custom-built components can be formulated by certain rules on clustering, too. For example, given a component library which has register that can perform shifting, we can allow vertexes which represent variables and vertexes which represent shift operations to be clustered together. Similarly, superscaler FUs or multiplication-adder can be formulated in this way, too.

3.6 Adopting popular binding algorithms

Basicly, what a binding algorithm does is to assign operations and variables to hardware components and organize data-transfers into interconnection circuits. Thus, we can directly translate "assign an operation



(a) Scheduling with multiclock and chaining



(b) Corresponding RTFG

Figure 7: Scheduled with multiclock and chaining

Algorithm Simple

```
begin
  clusters = initial_RTFG_clustering;
  repeat
    i = find_a_seed( clusters );
    repeat
      k = max_gain_cluster( i );
      i = merge_cluster( i, k );
    until no_gain_on_growing_seed( i );
  until binding_completed;
  return( clusters );
end
```

Figure 8: A simple binding algorithm based on our model

or variable e to a hardware component C" in any algorithms into "put the vertex e into the cluster C" in our binding model, using the same process, cost functions, priorities, etc. Similarly, we can directly translate "combine data-transfers $e_1, e_2, \ldots, e_n$ into interconnection C" into "put the vertexes $e_1, e_2, \ldots, e_n$ into the cluster C". Therefore, our binding model can work with any algorithms. However, our model can cluster all the operations, variables, or data-transfers concurrently. Therefore we can find better solution than previous works did. In addition, the hardware type for a cluster is determined at the end in our model. Thus optimal allocation for the binding can be obtained.

4 A simple algorithm

In this paper, we use a simple algorithm to show how our binding model works. More algorithms based on our binding model can be found in[24].

Figure 8 shows the simple binding algorithm. In general, this algorithm starts with an initial RTFG clustering in which each cluster contains only one vertex. However, the initial clustering also can be a clustering transformed from a partially completed implementation. In this case, those operations, variables, or data-transfers which have been bound to same component are grouped into same cluster in the initial clustering.

The simple algorithm grows one cluster at a time. The procedure `find_a_seed` returns a seed cluster which is going to be grown. The seed is created by merging two clusters which have the best gain. The procedure `max_gain_cluster` returns the best cluster which can be merged with the seed. Once a seed is determined, the algorithm keeps growing the seed until no more gains can be obtained. This select-seed-and-grow loop(outer loop) is repeated until the whole binding is completed.

Time complexity of the algorithm is $O(n^2)$ where n is number of vertexes in the input RTFG. We compute a gain table at the first time we execute `find_a_seed`. The complexity of computing the gain table is $O(n^2)$. Once the gain table is built, we only have to update part of it(less than n items) at each time we execute `find_a_seed` afterward. Thus, the complexity of `find_a_seed` is $O(n)$ except the first time. Moreover, the inner loop is executed totally at most n times despite how many times the outer loop is executed, since we need to merge clusters at most n times. Therefore,

$$\begin{aligned}
&\text{the complexity of the algorithm} \\
= &\text{ compute gain table} \\
&+ \text{number of outer loop} \times \texttt{find_a_seed} \\
&+ \text{total number of inner loop} \times \texttt{max_gain_cluster} \\
= &\ O(n^2) + O(n) \times O(n) + O(n) \times O(n) \\
= &\ O(n^2)
\end{aligned}$$

We assume the maximum number of input/output of an operation is constant m. When transform CDFG into RTFG, at most m variable vertexes plus m data-transfer vertexes can be created for each operation. Thus, number of vertexes in the RTFG is less than number of operations in the original CDFG multiplied by constant $(2m + 1)$. That is, the complexity of the algorithm is also $O(k^2)$ where k is number of the operations in the original CDFG.

The cost function we use to compute the gain for merging clusters is not the design quality measures like total area or a weighted sum of numbers of FUs, registers, and MUX. Actually, *it is not necessary to use the design quality measure as the cost function for proceeding binding.* As long as it can lead to better binding, any metrics can be used to direct the binding process. Moreover, we may switch around several metrics when necessary. More complicate cost function, for example, using expert systems and incorporating estimation tools to determine the gain for each merging, can obtain even better results. On the other hand, through different cost functions, our model not only can work for area optimization binding, but also can do power optimization binding or performance optimization binding.

However, we only use simple cost functions in this paper. To compete with previous works, which usually evaluate design quality by number of drivers and selector bits, the cost function we use is the number of common sources and sinks of operation and variable clusters as the primary key and component area as the secondary key. On the other hands, to obtain designs toward smaller area, the cost function we use is the number of common sources and sinks as the primary key and component area as the secondary key.

5 Source exchange

There are a lot of two-input operations whose sources are **commutative**[25], for examples, additions, multiplications, logic operators, etc. Properly exchanging the sources of these operations can save a considerable amount of selectors.

Figure 9 shows an example of how exchanging the

(a) Operations bound to the adder:

$$add_1(\text{ bus } 1 \text{ , bus } 2 \text{ })$$
$$add_2(\text{ bus } 2 \text{ , bus } 3 \text{ })$$
$$add_3(\text{ bus } 3 \text{ , bus } 1 \text{ })$$

(b) **(c)**

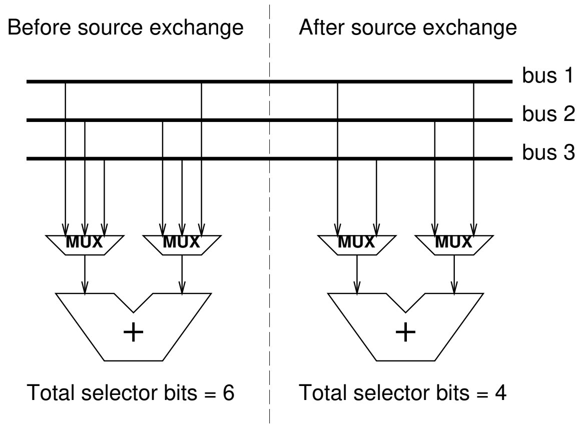


Figure 9: An example of saving selectors by exchanging sources

sources can save a number of selectors. Three additions as shown in Figure 9(a) are bound to an adder. They source three buses. Under straightforward implementation as shown in Figure 9(b), six selector bits are required. However, we can exchange the sources of add_2 as shown in Figure 9(c), where the functionality of the design remains totally the same and only four selector bits are required. In this example, two selector bits are saved after the source exchange.

In previous works, Ly et. al.[6] exchange sources of commutative operations according some heuristics before running their binding algorithms. On the other hand, Choi and Levitan[23] randomly exchange sources of commutative operations trying to obtain better results during their binding process. Different from their works, we propose a linear-time post-binding algorithm. We optimize arrangement of commutative sources after operations, variables, and data transfers are bound to hardware components.

In our work, we first construct an incompatible

(a) Four operations which are in a cluster:

$$Op_1(S_1, S_2)$$
$$Op_2(S_1, S_3)$$
$$Op_3(S_2, S_4)$$
$$Op_4(S_2, S_3)$$

(b) The corresponding incompatible graph

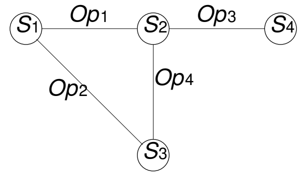


Figure 10: An incompatible graph example

graph defined below according to the relation among the sources of the operations which are clustered together(going to be bound to the same FU). Then, we use a red-black coloring algorithm to decide which input-port a source component should connect to.

Definition 2 *An* **incompatible graph** $\langle V, E \rangle$ *for the operations in a cluster is an undirected graph where*

1. *A vertex* $v \in V$ *represents a component sourced by the operations.*
2. *An edge* $e \in E$ *represents an operation in the cluster.*
3. *Two vertexes connected by an edge* e *are two sources of the operation. Two vertexes(sources) are said to be* **incompatible** *if they are connected by an edge.*

Since two sources of an operation can't connect to the same input-port, two incompatible vertexes(sources) should be bound(connect) to different input-ports.

Figure 10 shows an example of the incompatible graph. As shown in Figure 10(a), four operations $Op_1 \ldots Op_4$, which source four components $S_1 \ldots S_4$, are in a cluster. Figure 10(b) shows the corresponding incompatible graph. Each edge in Figure 10(b) represents an operation in Figure 10(a). For example, the edge Op_1 which connects S_1 and S_2 in Figure 10(b) represents the operation Op_1 which sources S_1 and S_2 in Figure 10(a).

Algorithm Red-black-coloring

```
begin
  repeat  /* breadth-first spanning-tree */
    v = select_seed( input_incompatible_graph );
    v.color = red ;
    repeat  /* coloring-neighbor process */
      if v.color = black then
        neb_color = red
      else
        neb_color = black;
      for i ∈ neighbor( v ) do
        if i.color = no_color then
          if ∃ j ∈ neighbor( i ),
            j.color = neb_color then
            i.color = bi_color;
          else
            begin
              i.color = neb_color;
              add_queue( i );
            end
      v = pop_queue();
    until v = φ;
  until all_vertexes_are_colored;
end
```

Figure 11: Red-black-coloring algorithm

Once we have the incompatible graph, we color each vertex with red or black to decide which input-port the vertex(source) should be bound(connect) to. Two incompatible vertexes are colored with different colors and vertexes(sources) have same color are bound(connect) to the same input-port. However, there are some sources which inevitably have to connect to both input-ports. The correspond vertexes of these sources are colored with both colors. For example, one of the sources S_1, S_2, and S_3 in Figure 10(a) has to connect to both input-ports. Similarly, one of the vertexes S_1, S_2, and S_3 in Figure 10(b) has to be colored by both colors. Finally, if we can find a coloring for the incompatible graph with least number of bi-colored vertexes, we can obtain a source arrangement with the least number of selector bits.

To find a coloring with least number of bi-colored vertexes for the incompatible graph is NP-complete. Thus, we use a quick and effective red-black coloring algorithm which can find a near optimal solution in linear time complexity.

Figure 11 shows our coloring algorithm. It is basicly a breadth-first spanning-tree[26] algorithm. We first select a un-colored vertex `v` as the seed and color it red. Then, we color all of `v`'s neighbors `i` with the other color `neb_color` or bi-color. If coloring an `i` with `neb_color` causes color conflict with one of the `i`'s neighbors `j`, we color the `i` with bi-color. Otherwise, we color the `i` with `neb_color`. Once we have colored all of the neighbors of the seed `v`, we follow breadth-first criterion to select a newly uni-colored(black or red) vertex as the next seed and repeat the coloring-neighbor process(inner repeat loop). Meanwhile, a queue is used to keep the list of seed candidates. All newly uni-colored vertexes are stored into the queue. One candidate is popped out each time the inner repeat loop needs a new seed. Once all candidates are popped out, we again select a vertex with no color as the next seed and start another outer repeat loop. The outer loop continues until all of the vertexes are colored. Moreover, the function `select_seed` selects a vertex with no color as the seed and the function `neighbor` returns all of the neighbors of the vertex given.

Algorithm Red-black-coloring can find a near optimal solution within $O(n)$ time complexity, where n is number of vertexes in the incompatible graph. Actually, we obtained optimal solutions in almost all of our experiments.

6 Experimental results

There are two major advantages of our work over previous':

	clk		MI	reg	bus
bus-base	4	SCHALLOC	12	6	10
		Ours	8	6	6
	6	SCHALLOC	11	6	11
		Ours	8	6	7
MUX-base	4	HAL	13	10	
		Splicer	11	10	
		LYRA	9	9	
		Ours	9	8	

Table 1: Differential equation of HAL — use 1 adder, 1 subtractor, 2 multipliers, and 1 comparator

1. We do interconnection optimization concurrently with variable and operation binding.
2. Our method can find better trade-off among combination of FUs and interconnection.

To show the first advantage, we compare experimental results of previous works with ours, in terms of number of components required and interconnection cost. To demonstrate the second advantage, we show some examples of using different combination of FUs. Our experimental results are obtained by using both of the algorithms in Section 4 and Section 5. We use VCC4DP3 Datapath Library[27] to generate our final implementations.

6.1 Comparison with previous works

To show the performance of our method, we compare our results with previous works Schalloc[28], HAL[5], Splicer[29], LYRA[8], SPAID[10], Su and Xue[30], STAR[13], and M&M[23].

Table 1 shows experimental results of the differential equation example of HAL[5], using 1 adder, 1 subtractor, 2 multipliers, and 1 comparator. Both bus-base(linear) and MUX-base(random topology) interconnection styles are tested. The forth column are numbers of total MUX input, including selectors and drivers. Table 2 shows experimental results of the 5th order elliptic filter of Kung[31], using register files and

clk		FU	MI	RF	mem	bus
17	STAR	2+ 2$\times$	29	3	11	5
	SPAID	3+ 2$\times$	26	6	17	6
	MandM	2+ 2$\times$	25	6	9	4
	Ours	2+ 2$\times$	22	5	10	8
	STAR	2+ 1$\times_p$	26	3	11	5
	SPAID	3+ 2$\times_p$	26	6	17	6
	Ours	2+ 1$\times_p$	20	5	10	8
28	STAR	1+ 1$\times_p$	15	2	11	3
	Ours	1+ 1$\times_p$	12	3	11	5

$\times_p$: pipeline multiplier

Table 2: Elliptic filter of Kung — bus interconnection

clk		FU	MI	reg
17	HAL	3+ 3$\times$	31	12
	Su	3+ 3$\times$	38	10
	Ours	2+ 2$\times$	32	10
	HAL	3+ 2$\times_p$	31	12
	Su	3+ 2$\times_p$	37	10
	Ours	2+ 1$\times_p$	28	10

$\times_p$: pipeline multiplier

Table 3: Elliptic filter of Kung — MUX interconnection

clk†		FU	MI	RF	mem	bus
8	STAR	2+ 2− 2$\times$	61	7	14	12
	Ours	2+ 2− 2$\times$	53	8	12	17
13	STAR	1+ 1− 2$\times$	50	6	12	8
	Ours	1+ 1− 2$\times$	40	7	14	13
16	STAR	1+ 1− 1$\times$	38	5	15	6
	Ours	1+ 1− 1$\times$	23	5	11	11

† input interval

Table 4: Pipelined FDCT kernel of Mallon

clk†		FU	MI	RF	mem	bus
4	STAR	4+ 2$\times$	42	8	23	12
	Ours	4+ 2$\times$	35	8	19	12
5	STAR	3+ 2$\times$	25	6	20	10
	Ours	3+ 2$\times$	23	6	17	11
6	STAR	3+ 2$\times$	42	5	21	8
	Ours	3+ 2$\times$	30	5	15	9
8	STAR	2+ 2$\times$	27	4	23	6
	Ours	2+ 2$\times$	22	4	13	6

† input interval

Table 5: Pipelined FIR of Park

FU	area(μm^2)
adder	40.0K
subtractor	41.6K
comparator	32.0K
add-sub	48.0K
ALU	83.2K
multiplier	112.0K

Table 9: Size of FUs in VCC4DP3 Datapath Library

bus-base interconnection. Table 3 shows experimental results of the 5th order elliptic filter, using registers and MUX-base interconnection. Table 4 and Table 5 show experimental results of pipelined FDCT(Fast Discrete Cosine Transform)[22] kernel and pipelined FIR filter[32], using register files and bus-base interconnection. From these tables, we can see our method can obtain similar or better results than previous'.

Since we don't use iterative improvement, our method is relatively fast in addition to producing competitive results. For example, using 110 MHz SUN Sparc5, it takes less than one second to obtain each FDCT implementation. Compared to STAR[13], which can obtain results similar to ours, it takes twelve minutes in average to obtain an FDCT implementation using SUN Sparc 1. Moreover, most of our designs are obtained within 0.1 second.

6.2 Using multi-function units

It is a trade-off between using multi-function FUs and mono-function FUs. Using multi-function FUs can reduce the number of FUs required, since the utilization of FUs is increased. However, a multi-function FU may be much more expensive than a mono-function FU. Thus, total FU cost for using less number of multi-function FUs may be higher than total FU cost for using more mono-function FUs.

In addition to the cost for FUs, there is another even bigger impact when using multi-function FUs. Once we have multi-function FUs in a design, there are more candidates for each operation to be bound to, therefore we can choose an arrangement which needs less number of MUX input and then better interconnection can be obtained. As the example shown in Figure 1 (in Section 1), two ALUs are more expensive than one adder plus one subtractor. However, each operation can be bound to either FU as in Figure 1(c) instead of bound to a specific FU as in Figure 1(b). Thus, an implementation with smaller total area may be obtained, even though using multi-function units doesn't improve FU utilization a little bit.

Table 6, 7, 8 show experimental results on trade-off multi-function FUs and mono-function FUs. The area of each FU we used is shown in Table 9. The add-subtractor can execute only addition and subtraction, and cost only a little higher then the adder or the subtractor. On the other hand, the ALU can execute all general operations except multiplication and division, and cost much higher then the adder, the subtractor, or the comparator. Since there is no add-sub-comparator available in VCC4DP3 Datapath Library, using much expensive ALU is the only way for the comparison operation to share FU with other operations.

In Table 6, we test each design using (1) only mono-function units, (2) mono-function units and add-subtractor, (3) all types of FUs. The sixth column is total area of all FUs used in each implementation. We can see total FU cost for implementation using multi-function unit is higher than that of implementation using mono-function unit. However, due to saving in MUXes, better implementations may still be obtained by using multi-function units. Similarly, Table 7 and Table 8 show more experimental results under various design styles.

inter-conn	FU	bus	reg	MI	total FU cost(μ^2)	total area(μ^2)	FU used
bus-base	$1+\ 1-\ 2\times\ 1cmp$	6	6	8	337.6K	708K	mono
	$1+\ 1\pm\ 2\times\ 1cmp$	6	6	6	344.0K	676K	add-sub
	$1+\ 1\pm\ 2\times\ 1cmp$	6	6	6	344.0K	676K	all
MUX-base	$1+\ 1-\ 2\times\ 1cmp$	–	8	9	337.6K	728K	mono
	$1+\ 1\pm\ 2\times\ 1cmp$	–	8	6	344.0K	676K	add-sub
	$1+\ 2\times\ 1alu$	–	8	3	347.2K	620K	all

Table 6: FU sharing of the differential equation example (4 C-step scheduling)

clk	inter-conn	FU	bus	reg	MI	total FU cost(μ^2)	total area(μ^2)	FU used
8	bus-base	$3+\ 3-\ 2\times$	19	10	55	468.8K	2106K	mono
		$5\pm\ 3\times$	21	10	33	576.0K	1800K	all
8	MUX-base	$2+\ 2-\ 2\times$	–	10	44	387.2K	1784K	mono
		$5\pm\ 2\times$	–	10	31	464.0K	1620K	all
13	bus-base	$1+\ 1-\ 2\times$	12	13	51	305.6K	1948K	mono
		$3\pm\ 2\times$	13	14	41	368.0K	1868K	all
13	MUX-base	$1+\ 1-\ 2\times$	–	14	50	305.6K	1970K	mono
		$3\pm\ 2\times$	–	13	41	368.0K	1826K	all
16	bus-base	$1+\ 1-\ 1\times$	8	11	35	193.6K	1404K	mono
		$3\pm\ 1\times$	11	11	24	256.0K	1262K	all
16	MUX-base	$1+\ 1-\ 1\times$	–	10	32	193.6K	1302K	mono
		$3\pm\ 1\times$	–	11	21	256.0K	1202K	all

Table 7: FU sharing of the pipelined FDCT kernel using registers

clk	inter-conn	FU	bus	RF	MI	mem	total FU cost(μ^2)	total area(μ^2)	FU used
8	bus-base	$3+\ 3-\ 2\times$	19	8	34	12	468.8K	2082K	mono
		$3+\ 3-\ 2\times$	19	8	34	12	468.8K	2082K	all
8	MUX-base	$2+\ 3-\ 2\times$	–	8	31	13	428.8K	1984K	mono
		$5\pm\ 2\times$	–	8	20	13	464.0K	1808K	all
13	bus-base	$1+\ 1-\ 2\times$	13	7	40	14	305.6K	1938K	mono
		$3\pm\ 2\times$	13	7	21	15	368.0K	1648K	all
13	MUX-base	$1+\ 1-\ 2\times$	–	7	35	15	305.6K	1850K	mono
		$2\pm\ 2\times$	–	6	34	14	320.0K	1752K	all
16	bus-base	$1+\ 1-\ 1\times$	11	5	23	11	193.6K	1254K	mono
		$3\pm\ 1\times$	9	5	9	11	256.0K	1052K	all
16	MUX-base	$1+\ 1-\ 1\times$	–	5	19	11	193.6K	1174K	mono
		$1+\ 3\pm\ 1\times$	–	5	5	11	296.0K	1022K	all

Table 8: FU sharing of the pipelined FDCT kernel using register files

7 Concluding remarks

A binding model which can formulate any architectures is presented in this paper. This model provides complete binding information that can help people working on implementation details and fine tune their binding algorithms.

With complete binding information provided by our model, we have two major advantages: (1) We can do interconnection optimization concurrently with operation and variable binding. (2) We can find better trade-off among combination of FUs and interconnection.

A simple binding algorithm is proposed to demonstrate the performance of our model. Using our binding model, even simple algorithms can obtain better or competitive results than complex previous works.

References

[1] D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and design of embedded systems.* New Jersey: Prentice Hall, 1994.

[2] D. Gajski, N. Dutt, C. Wu, and Y. Lin, *High-Level Synthesis: Introduction to Chip and System Design.* Boston, Massachusetts: Kluwer Academic Publishers, 1991.

[3] C.Tseng and D. Siewiorek, "Automated synthesis of datapaths in digital systems," *IEEE Transactions on Computer-Aided Design*, pp. 379–395, July 1986.

[4] D. Thomas, E. Dirkes, R. Walker, J. Rajan, J. Nestor, and R. Blackburn, "The system architect's workbench," in *Proceedings of the Design Automation Conference*, 1988.

[5] P. Paulin and J. Knight, "Scheduling and binding algorithms for high-level synthesis," in *Proceedings of the Design Automation Conference*, pp. 1–6, 1989.

[6] T. Ly, W. Elwood, and E. Girczyc, "A generalized interconnect model for data path synthesis," in *Proceedings of the Design Automation Conference*, pp. 168–173, 1990.

[7] F. Kurdahi and A. Parker, "Real: A program for register allocation," in *Proceedings of the Design Automation Conference*, 1987.

[8] C. Huang, Y. Chen, Y. Lin, and Y. Hsu, "Datapath allocation based on bipartite weighted matching," in *Proceedings of the Design Automation Conference*, 1990.

[9] M. McFarland, "Using bottom-up design techniques in the synthesis of digital hardware from abstract behavioral descriptions," in *Proceedings of the Design Automation Conference*, 1986.

[10] B. Haroun and M. Elmasry, "Architectural synthesis for dsp silicon compilers," *IEEE Transactions on Computer-Aided Design*, pp. 431–447, April 1989.

[11] K. Kucukcakar and A. Parker, "Datapath tradeoffs using MABAL," in *Proceedings of the Design Automation Conference*, 1990.

[12] B. Pangrle and D. Gajski, "Design tools for intelligent silicon compilation," *IEEE Transactions on Computer-Aided Design*, pp. 1098–1112, November 1987.

[13] F. Tsai and Y. Hsu, "Star: An automated data path allocator," *IEEE Transactions on Computer-Aided Design*, pp. 1053–1064, September 1992.

[14] J. Septièn, D. Dozos, F. Tirado, R. Hermida, and M. fernàndez, "Heuristics for branch-and-bound

global allocation," in *Proceedings of the International Conference on VLSI Design*, pp. 334–340, 1992.

[15] M. Rim, R. Jain, and R. D. Leone, "Optimal allocation and binding in high-level synthesis," in *Proceedings of the Design Automation Conference*, pp. 120–123, 1992.

[16] C. Gebotys and M. Elmasry, "Global optimization approach for architectural synthesis," *IEEE Transactions on Computer-Aided Design*, pp. 1266–1278, September 1993.

[17] B. Landwehr, P. Marwedel, and R. Dömer, "Oscar: Optimum simultaneous scheduling, allocation and resource binding based on integer programming," in *Proceedings of the European Design Automation Conference (EuroDAC)*, pp. 90–95, 1994.

[18] M. Rim, A. Mujumdar, R. Jain, and R. D. Leone, "Optimal and heuristic algorithms for solving the binding problem," *IEEE Transactions on Computer-Aided Design*, pp. 211–225, June 1994.

[19] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* San Francisco, CA: Freeman, 1979.

[20] S. Devadas and A. Newton, "Algorithms for hardware allocation in data path synthesis," *IEEE Transactions on Computer-Aided Design*, vol. 8, no. 7, pp. 768–781, 1989.

[21] G. Krishnamoorthy and J. Nestor, "Data path allocation using an extended binding model," in *Proceedings of the Design Automation Conference*, pp. 279–284, 1992.

[22] D. Mallon and P. Denyer, "A new approach to pipeline optimisation," in *Proceedings of the European Design Automation Conference (EuroDAC)*, pp. 83–88, 1990.

[23] K. Choi and S. Levitan, "A robust datapath allocation method for realistic system design," in *Proceedings of the International Conference on VLSI and CAD*, 1995.

[24] E.-S. Chang and D. D. Gajski, "A connection-oriented binding model for binding algorithms." UC Irvine, Dept. of ICS, Technical Report 96-49,1996.

[25] C. Liu, *Elements of discrete mathematics.* New York: McGraw-Hill, 1977.

[26] E. Horowitz and S. Sahni, *Fundamemtals of Data Structures.* Computer Science Press, Inc., 1982.

[27] VLSI Technologies Inc., *0.8-Micron Datapath Library (VCC4DP3)*, 1992.

[28] N. Berry and B. Pangrle, "Schalloc: An algorithm for simultaneous scheduling and connectivity binding in a datapath synthesis system," in *Proceedings of the European Design Automation Conference (EuroDAC)*, pp. 78–82, 1990.

[29] B. M. Pangrle, "Splicer: A heuristic approach to connectivity binding," in *Proceedings of the Design Automation Conference*, pp. 536–541, June 1988.

[30] M. Su and H. Xue, "An allocation algorithm for data path synthesis," in *Proceedings of the International Conference on VLSI and CAD*, 1995.

[31] S. Kung, H. Whitehouse, and T. Kailath, *VLSI and Modern Signal Processing.* Prentice-Hall, 1985.

[32] N. Park and A. Parker, "Sehwa: A software package for synthesis on pipelines from behavioral specifications," *IEEE Transactions on Computer-Aided Design*, pp. 356–370, March 1988.