Function Binding in RTL Design Methodology

Qiang Xie
Daniel D. Gajski

Technical Report ICS-01-36
June 28th, 2001

Center for Embedded Computer Systems
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{qxie, gajski}@ics.uci.edu

Function Binding in RTL Design Methodology

Qiang Xie
Daniel D. Gajski

Technical Report ICS-01-36
June 28th, 2001

Center for Embedded Computer Systems
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{qxie, gajski}@ics.uci.edu

Abstract

This report describes the function binding algorithm in RTL synthesis. It describes the RTL design methodology and implement our function binding algorithms in our RTL design refine tool. We proposed two algorithms here, one algorithm is based on the clique partitioning algorithm and the other is based on the seed constructive based algorithm. Our algorithms are resource constraint algorithm and they are focused to minimize the cost ofinterconnections needed for the datapath and can be performed at different RTL refine steps.

Contents

List of figures

List of tables

Function Binding in RTL Design Methodology

Qiang Xie, Daniel D. Gajski
Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA

Abstract

This report describes the function binding algorithm in RTL synthesis. It describes the RTL design methodology and implement our function binding algorithms in our RTL design refine tool. We proposed two algorithms here, one algorithm is based on the clique partitioning algorithm and the other is based on the seed constructive based algorithm. Our algorithms are resource constraint algorithm and they are focused to minimize the cost ofinterconnections needed for the datapath and can be performed at different RTL refine steps.

1 Introduction

With the ever increasing complexity and time-to-market pressures in the design of embedded systems, designers have moved the design to higher levels of abstraction in order to increase productivity. However, each design must be described, eventually, at the lowest level (e.g. layout masks) through various synthesis processes. Register-transfer level (RTL) synthesis has been recognized as one of the major design methodology.

RTL synthesis involves the transformation of behavioral description of the design into a set of interconnected RT components which satisfy the behavior and some specified constraints, such as the number of resources, timing etc. Three major synthesis tasks are applied during the transformation: allocation, scheduling and binding. Allocation determines the number of resources, such as registers, buses, and function units, that will be used in the implementation. Scheduling partitions the behavioral description into time intervals. Binding assigns variables to storage units (register binding), assigns operations to function units (function binding), and interconnections to buses (bus binding).

This report focuses on the function binding in our RTL synthesis approach. The purpose of function binding is to map the operations in behavioral description into a set of selected functional units and minimize the cost of the design. In this report, we will present our function binding algorithm, which is based on clique partitioning algorithm [GWDL92]. This algorithm attempts to minimize the complexity of the interconnection by bind operations with the same inputs or outputs to the same functional unit. Our algorithm can be performed at any styles of the RTL implementation model and determine the exact mapping of the operations into the function units.

The rest of this report is organized as follows: section 2 is the motivation of this project, section 3 describes the RTL design methodology and its implementation model with different styles. Section 4 and 5 describes our data structure and function binding algorithm. Experimental results for our algorithms are given in section 6 and conclusions are made in section 7.

2 Motivation

Much research for High-level/RTL synthesis has been done since 1980s. Currently, many commercial and academical High-level/RTL synthesis tools exists in Electronic Design Automation (EDA) industry but the design community wouldn't integrate them into its design methodology and design flow, because 1)they can support only several limited architecture, 2) they are lack of interaction between them and the designers, 3) the quality of the design which they generated is worth than that of manual design. TO make them popularly used in the design, we should tackle these problems. Our RTL synthesis approach supports the more advanced architecture like bus-based architecture, as shown in figure 1, instead of the mux-based architecture whose performance is not good in large design.
Our RTL deisgn refinement tool also provides the interaction between designer and refinement tool. To support this interaction, we use the finite state machine with datapath (FSMD) as our RTL implementation model representation and define 5 different styles of RTL[Acc01] description according to the RTL refinement steps. The function binding algorithm in our approach should minimize the function units and the

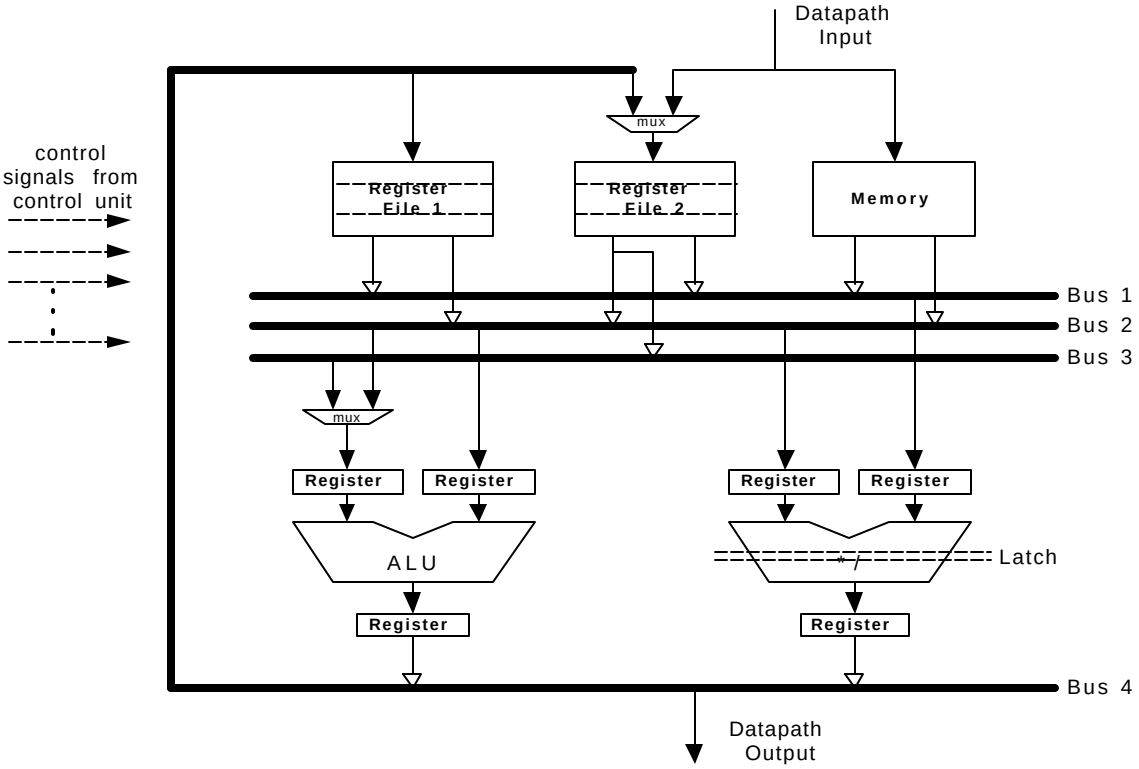


Figure 1. Bus-based target architecture

interconnection in a datapath and can be performed in different styles of RTL implementation model accordingly.

3 RTL design methodology

The RTL design is modeled by Finite State Machine with Datapath (FSMD) [Acc01], which is FSM model with assignment statements added to each state. The FSMD can completely specify the behavior of an arbitrary RTL design. The variables and functions in FSMD may have different interpretations which in turn defines several styles of RTL semantics.

The register-transfer level (RTL) implementation model has two views: a behavioral RTL view and a structural RTL view [GERS00]. The behavioral RTL specifies the operations performed in each clock cycle with explicitly modeling the units in the component's datapath and is obtained by scheduling the operations in the C code into clock cycles. The structural RTL view of the implementation model explicitly models the allocation of RTL components, the scheduling of register transfers into clock cycles, and the binding of operations, variables and assignments to functional units, register, memories and components busses. The RTL implementation model can be divided into 5 well defined styles as behavior RTL (style 1), Storage-mapped RTL (style 2), function-mapped RTL (style 3), Connection-mapped RTL (style 4), and structural RTL (style 5). These different styles represent the different refinement steps like scheduling, register binding, function binding and bus binding from behavior RTL (style 1) to structural RTL (style 5) as Figure 1 shows.

Figure 2 describes the RTL refine flow in our RTL design methodology. We generate the FSMD/CDFG as our internal representation for refinement. Each refine step is based on this FSMD model, which contains a list of states and each state has its own Control/Data Flow Graph (CDFG)[DshinG01]. The scheduling task separates the state into sub-state based on resource constraint. The storage, function, and interconnection binding are performed considering each state transition. However, due to the interdependence of scheduling, allocation, and binding, the order of these three binding steps should be interchangeable to get the RTL implementation model in different styles.

The netlist mapper generates the style 5 exposed-control RTL from style 4 RTL. Then the style 5 RTL description can be used as input for gate-level synthesis tool such as Synopsys Design Compiler.

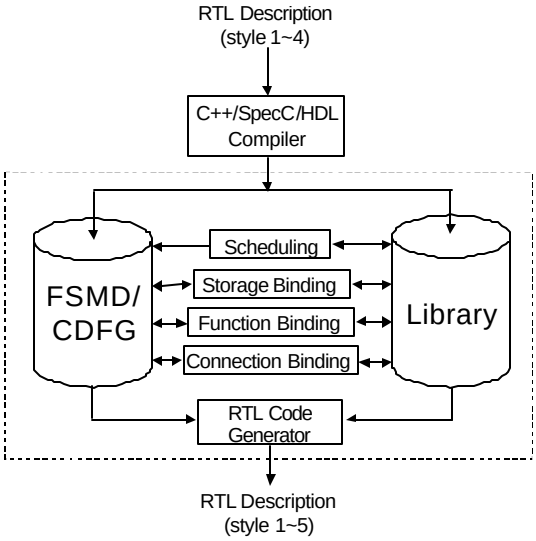


Figure 2 RTL design refinement flow

The RTL component library has the information about datapath modules, such as ALUs, multipliers, register files, memories and buses. When each synthesis step is performed, it refers to the RTL component library to get the information about resource constraint.

4 Internal representation

Our RTL model is represented by FSMD, which has a set of states and transition among them. Each state has its own CDFG. This section describes the CDFG representation, which is selected for internal data representation for our RTL model.

The CDFG is the hierarchical graph which has the data flow information to describe the operations and their depepencies and has the control flow information which is related to branching and iteration constructs. The CDFG has been used for the internal representation of high-level synthesis tool since mid-1980s and has many variations. It can be hierachical or non-hierarchical, polar or non-polar, and cycle or acycle.

We made the novel CDFG structure to represent the RTL model and to perform the RTL refinement steps. Our CDFG is hierarchical, acyclic polar graph, which is shown in Figure 3. Given a behavioral RTL description shows as follows:

```
// Behavioral RTL(style1)
a=b+c;
for(I = 0; I < 32; I=I+1)
  d[I:I] ^= e[I:I];
if (start == 0)
 state = S0;
else
state = S1;
```

we generate a CDFG graph as shown in figure 3. The acycle graph makes it easy to implement the graph algorithm, because it has no loop. The polar graph has the single-entry and single exit

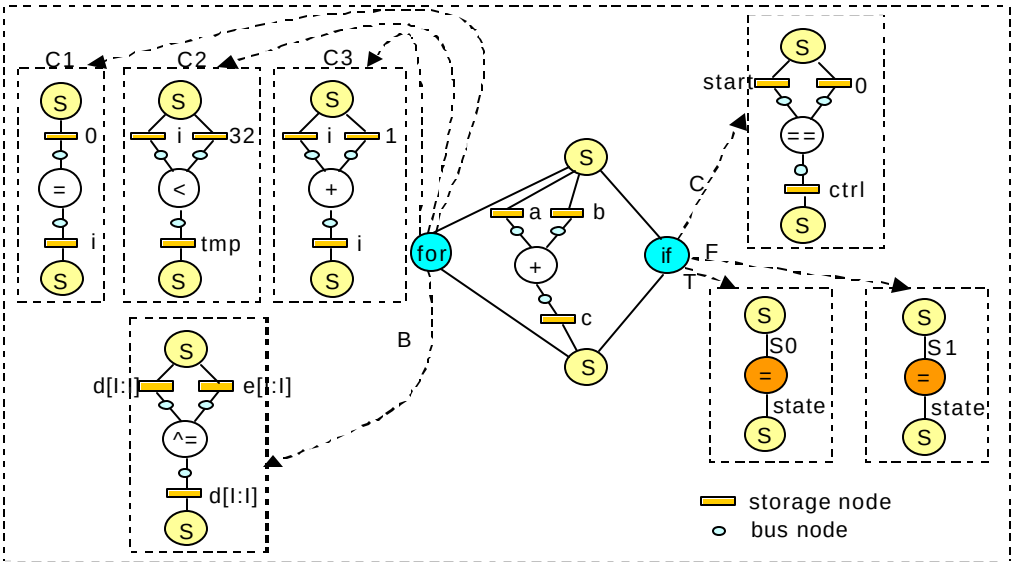


Figure 3. CDFG example (style 1)

property using no-operation (source node/sink node in our graph) and makes it easy to build hierachical graph. The node S in the center represents the no-operation node. The top S is the source node and the bottom S is the sink node.

In this graph, the edge has the dependency information between nodes such as control dependency and data dependency. The node has all information except the flow information. The node is decomposed of the non-hierachical node and the hierarchical node. The non-hierachical node has the datapath operation information, such as operation node to perform arithmetic/logic operation, storage node to store the data, bus node to transfer the data between functional unit and storage unit, and control node to generate the status information of datapath, and state transition node to store state transition information in finite state machine. In Figure 3 shows the operation node which is the white circle node, storage node which is the shaded rectangular node, bus node which is the small shaded circle node between operation node and storage node. The hierarchical node is divided to the module node to represent the structural hierarchy in SpecC description, branch node to represent branching information and loop node to represent the iteration information. The branch node (if node) and loop node (for node) are shown in center part of the figure .

5 Function binding

In our RTL synthesis approach, the function binding is one of the major tasks of the binding process. The function binding is done in the corresponding FSMD which is generated from the RTL description. In this binding process, the operations nodes in each CDFG in the FSMD are extracted to form a list of operations with proper information (e.g. state it belonged), then we generate a graph based on this list, and perform function binding on this graph. We develop two algorithms for the binding, one is based on the clique partition and the other is based on a seat based algorithm. We will discuss these algorithm in the rest of this sections.

Our approach is based on the following assumption:

1. Within any given state, a datapath will not perform all the operations;
2. The operations performed in different control steps can be mapped to the same function unit if this function unit can perform these types of operations and they are not conflicted in control steps;
3. In order to minimize the number of interconnections, it is beneficial to bind those operations which have the same input or the same output;
4. Each data I/O port is connected to a register through a input/output bus, thus we assume that there isn't the situation in which a data I/O port is the input/output of an operation;

5.1 Binding Model

The function binding algorithm begins with an input FSMD, which is generated from the RTL description. And after resource allocation, the given resource has been associated with the FSMD model. An input FSMD model for function binding can be defined as follows:

Definition 1 *A **FSMD model for function binding** is a member of*

set ***FSMD*** *{*
S : *State*
Res : *Resource*
}

where S is a set of states, and each state $s_i \in S$ *contains a set of CDFG which contains a set of nodes N; Res is the set of resources that is allocated to this FSMD.*

The set Res is a set of resources given after allocation. We call it a *resource set*, which can be defined as follows:

Definition 2 *A **resource set** is a member of*

set ***Res*** *{*
SU : *Storage unit*
FU : *Function unit*
BU: *Bus unit*
}

where SU is a set of storage units; FU is a set of function unit; BU is a set of bus unit.

As described above, each state contains a set of CDFGs. And each CDFG contains a set of nodes, we call it a *node set* which can be defined as follows:

Definition 3 *A **node set** N is a member of*

set ***N*** *{*

V : Variable
O : Operation
I: Interconnection
C: Control
}

where V is a set of storage units; O is a set of opeartion units; I is a set of interconnection units; and C is a set of control signals.

In definition 2, the element *OU* contains a set of operation nodes[DshinGa01] in a CDFG. To implement our function binding algorithm, we need to create a list of the operations with necessary information in the whole FSMD. We extend each operation node in the CDFG structure to a new node, super-operation node. A super-operation node is defined as follows:

Definition 4 *A **super-operation node** is a member of*

set ***SN*** *{*
op : operation
id : int
state: int
if_flag: int
clks: int
}

where op is the operation node in the CDFG; id is an unique identifier for each operation node in the FSMD; state is the state where the operation is performed; if_flag indicates that whether this operation is in a branch statement; clks is the execution time for the function unit to complete this operation.

The function binding is performed based on the the input FSMD(S, Res). The function binding problem can be formulated as follows:

Definition 5 *Given certain resources**, the Function binding problem** is to map each operation in the FSMD to a function unit* $fu_j \hat{I}$ *FU, while minimize the cost of the interconnections.*

5.2 Function Binding Process

Our binding process is performed after scheduling, where each variable, operation, and interconnection will be allocated to a resource type, such as register file, ALU, and bus. The compiler will generate a FSMD model as the input for the binding. The function binding process is shown in figure 4.

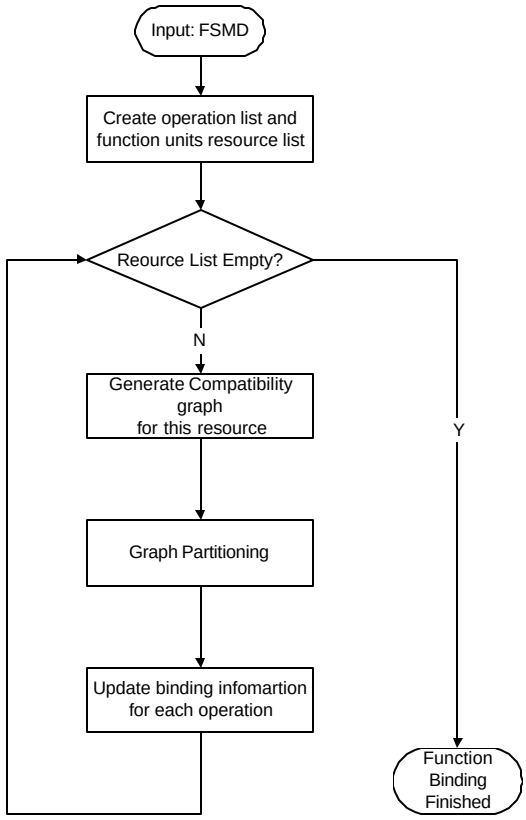


Figure 4. Function Binding Process

At the beginning of the binding process, the operation nodes in the CDFG are extracted from the input FSMD model, together with their allocation information. We create a super-operation list for all the operation nodes and a resource list for the allocated resources. After the super-operation node list is created, for each resource type in the resource list, we generate a compatibility graph for the operations that are allocated to the same resource type. And based on the compatibility graph, we perform graph partition and get the result for function binding. At last we update the binding information in each operation node. We repeat these step for each resource type in the resource list until all the resource types allocated to operations have been processed.

5.3 Function binding algorithm

This section will discuss our proposal algorithm which is based on the graph partition, and then

we will explain this algorithm by an illustrative example.

5.3.1 Function binding algorithm

The function binding algorithm is shown in figure 5. The proposal algorithm is base on the clique partition algorithm and we extend the algorithm by using a compatibility graph with weighted edges.

As in our RTL design process, a FSMD(S, Res) is generated from the RTL description and the resource allocation is given by the user, where S is a set of state, and Res is a set of resource allocation information. Each state $s_i \in S$ contains a set of CDFG C_i, and each CDFG contains node set N_i . OpList is a set of the super-operation nodes as defined in definition 4.

Let G = (V, E) denote a compatibility graph, where $V=\{v_i\}$ is the set of vertices which represents the identification number of operations and $E = \{e_{ij}\}$ is the set of weighted edges which are defined as follows:

.

Definition 6 *A **weighted edge** is an edge that link two compatible operation nodes with a weight value associated with it, where a weight represents the common inputs or common outputs of the two operation nodes that linked by the edge.*

An edge $e_{ij} \in E$ is used to link two vertices v_i and v_j, which represent two operations and they can be mapped to the same function unit without conflict. CliqueList is a set of clique generated from the graph G, where each clique $C_i \subseteq$ CliqueList contains a set of vertices.

The function Add2OpList(OpList, n_j, s_i) is used to add a super-operation node sn_i, as described in definition 4, to create a list of super-operation nodes. The function Add2ResList(ResList, n_k) add different allocated resource type and correspondence information to ResList to form a resource list. The function Add2Graph(G, i, j, e_{ij}, res) is used to add an weighted edge to a graph G(V, E) for the resource type *res*. And function Clique_partition(G, res) perform partitioning on the graph G , it returns a list of cliques as CliqueList. The function SetBindingInfo(sn_i) updates the resource binding information in the nodes sn_i, where each operation will be mapped to a instance of allocated resource type.

The function binding begins with the input FSMD model FSMD(S, Res). We first create a list of the super-operation nodes OpList, together with the resource list ResList which is related to allocated resource information. Then based on the super operation list, we compare each pair of the super-operation nodes which are allocated to the same resource type, calculate their edges and weight of the edges, and create a compatibility graph G for the operation that allocated to the same resource type. The compatible edge and its weight are calculated as following:

1. If two operations are compatible, i.e. they are performed in different control steps and they can be mapped to the same function unit, and they have common input or common output, then a non-zero weight is assigned to the edge that links these two nodes. Note that here we consider the two-

Algorithm 1. Graph partition based algorithm

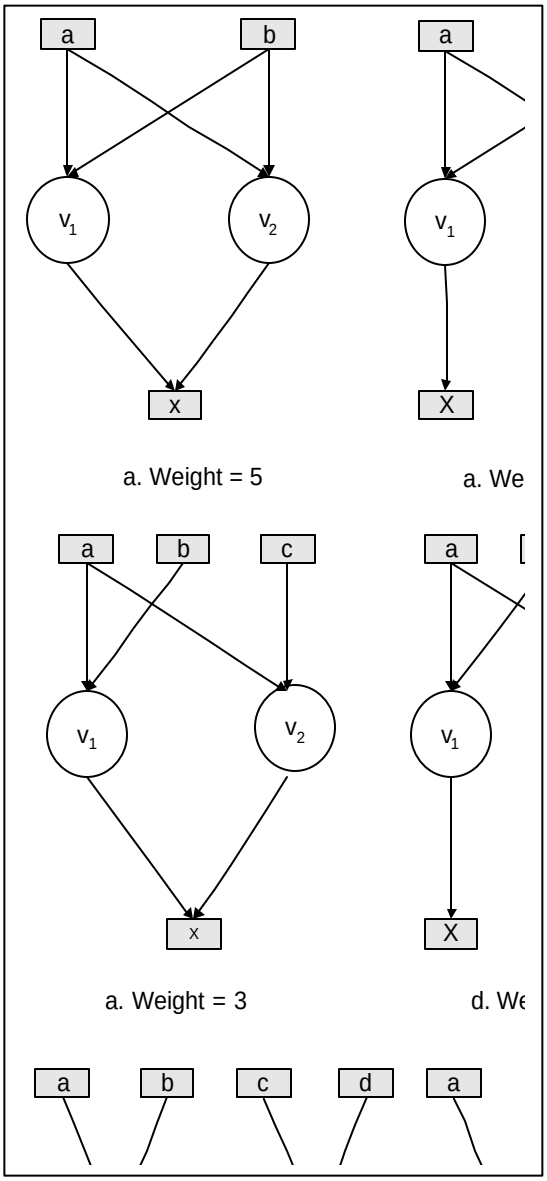
```
Input: FSMD(S, Res)
/* create super-operation list OpList */
OpList = ∅;
ResList = ∅;
for each s_i ∈ S do
    for each c_j ∈ C_i do
        for each n_k ∈ N_j do
            if n_k is operation node then
            Add2OpList(OpList, n_k, s_i );
            Add2ResList(ResList, n_k );
          endif
        endfor
    endfor
endfor

for each res ∈ ResList do
/*create a graph G(V, E) */
 G = ∅;
 for each sn_i ∈ OpList do
    for each sn_j ∈ OpList do
        weight = CompWeight(sn_i, sn_j);
        if(weight =-1) continue;
        e_ij = weight;
        Add2Graph(G, i, j, e_ij, res);
    endfor
 endfor

 /* clique partitioning */
 CliqueList = Clique_partition(G, res);

 /* Update binding information */
 for each C_i ⊆ CliqueList do
    for each sn_i ∈ OpList and sn_i ∈ C_i do
        SetBindingInfo(sn_i);
    endfor
 endfor
endfor
```

Figure 5. Function Binding Algorithm

inputs operations whose sources are **communicative**[Cliu77], e.g. additions, multiplications, etc;

2. If two operations are compatible, but they have no common source/sink, then a weight 0 is assigned to the edge that link these two nodes.

Figure 6 shows different level of weights between two compatible operation nodes. We can see from it that the weight of an edge values from 5 to 0, which is determined by the number of common inputs and common output of two operation nodes. For example, the two compatible nodes that have two common inputs and one common output will be assigned a highest weight as 5, and the two compatible



nodes that have no common input and output will be assigned a lowest weight as 0. By setting the weight threshold, we can limit the complexity of the clique graph and reduce the complexity of the interconnections at will.

As in graph partition, we may merge two compatible nodes and generate a new node, and the edges that link to the new node have to be calculated. There are different cases when merge two nodes. For examples, node x and node y will be merged to a new node xy. Node z have one common input with node x, and one common input with node y. Then node z may have two common inputs or one common input with the new node xy depend on whether edge e_{xz} and e_{yz} are pointed to the same input. Hence, to provide sufficient information for the graph partitioning performed followed, we extend these different weights to 12 cases as explained following:

Case 1:
No common input and common output;
Case2:
Common output and no common input;
Case 3:
No common output, the right input of node x is the left input of nodey;
Case 4:
No common output, the left input of node x is the right input of nodey;
Case 5:
Common right input and no common output;
Case 6:
Common left input and no common output;
Case 7:
Common output, the right input of node x is the left input of nodey;
Case 8:
Common output, the left input of node x is the right input of nodey
Case 9:
Common right input and common output;
Case 10:
Common left input and common output;
Case 11:
Two common inputs and no common output;
Case 12:
Two common inputs and common output;

On the other hand, to adapt our algorithm to different styles of RTL implementation mode, we have 3 methods to calculate the common inputs and common output of each pair of two operation nodes since in different styles, the inputs and output of the operations are different.

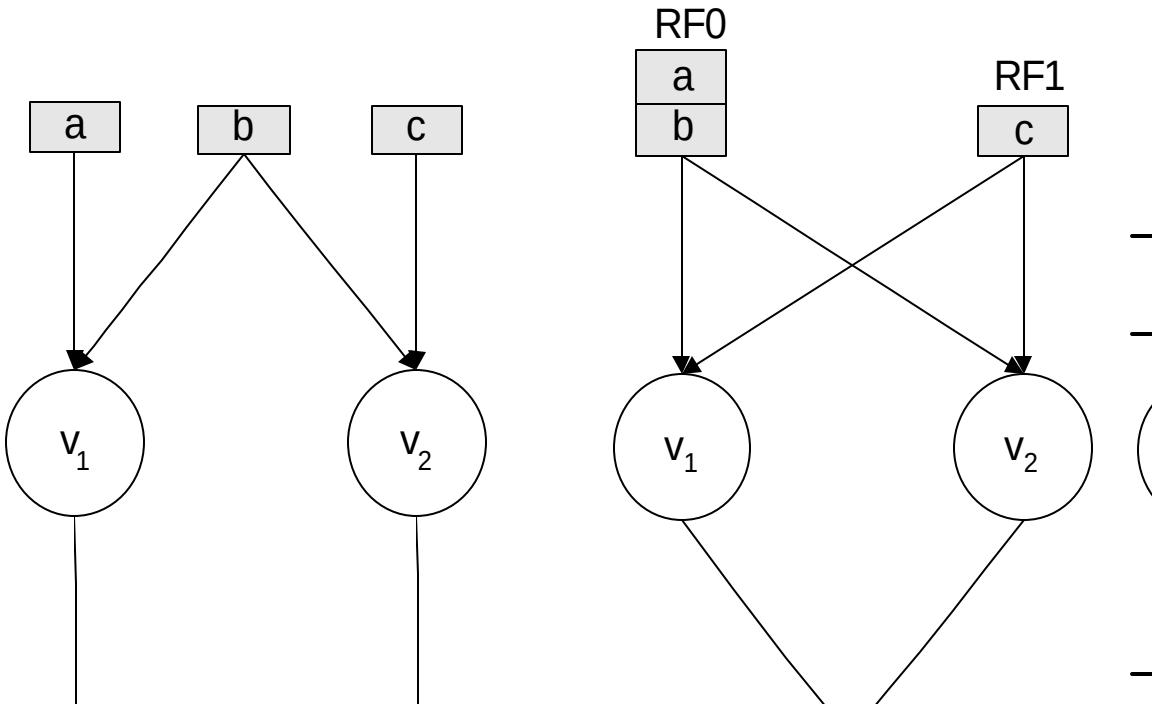


Figure 7 shows the different situation when we perform function binding in different styles.

As we can see from figure 7, when function binding is performed first before both storage unit and bus bindings, the inputs and output of an operation are linked to variables. In this case, we will compare the variables that link to the operation nodes to calculate the weight of each two operations. In figure 7(a), the weight of these two nodes is 2. When function binding is performed after storage unit binding, but before bus binding, the inputs and output of an operation are linked to storage units, such as register files, memories etc. In this case, we will compare the storage units and their ports that link to the operations to calculate the weight of each two operations. In the example of figure 7(b), the weight of these two nodes is 5. When function binding is performed after bus binding, the inputs and output of an operation are linked to buses. In this case, we will compare the buses that link to the operations to calculate the weight between each two operations. As in figure 7(c), the weight of the two operation nodes is 5. We can see that when performed in different styles, we may get different result for the function binding.

After a graph G is created, we used the graph partition algorithm, which is based on the clique partition algorithm [TSSi86], to partition the graph to a set of cliques, where the number of cliques equal the number of the resources allocated. The detail of the graph partitioning algorithm will be discussed in the following section. After partitioning, we will get a list of cliques, CliqueList, where each clique contains a set of operations that can be mapped to the same instance of function unit. Finally, we will update the binding information in each operation node by mapping the operations in the same clique to the same instance of the allocated resource type. For each allocated resource type for the operations, we generate the compatibility graph for it and repeat the binding steps for each allocated resource as described above, finally we will get the binding result.

5.3.2 Graph partitioning algorithm

The graph partitioning algorithm is performed on the compatibility graph for the operation nodes allocated to the same resource type. Figure 8 shows the detail of this algorithm.

The input of this algorithm is a compatibility graph G(V, E, Res), where V is a set of vertices, E is a set of edges, and Res is the allocated resource information, i.e. the number of the allocated resource, for this graph. The function MaxNeighbor (G, weight) returns an edge that has the maximum number of common neighbors among the edges which have the weight value as *weight*. A node $v_i \in V$ is a common neighbor of the two nodes v_j and $v_k \in V$, if there exist edges e_{ij} and $e_{ik} \in E$. The function MergeNode (G, $e_{ij,}$ weight) returns a new super-node snode after merges the two nodes v_i, v_j that linked by the

Algorithm 2: Graph partitioning alogrithm

```
Input: G(V, E, Res)
/* clique partitioning */
clique_number = node_number;
done =0;
for weight =5; weight >= 0 and done == 0 do
  while E ≠ ∅ and done == 0 do
      e_{i,j} = MaxNeighbor (G, weight);
      snode = MergeNode (G, e_{ij,} weight);
      clique_num = clique_num –1;
      if clique_num > res_num then
        while e_{i,j} =MaxNeighbor(G, weight,
snode) and clique_num > res_num do
           MergeNode(G, e_{ij}, weight);
          clique_num = clique_num –1;
        endwhile
      endif
      if clique_num == res_num then
        done =1;
      endif
 endwhile
  weight = weight –1;
endfor
```

Figure 8. Graph Partition

edge e_{ij}, and it also update the edge information in graph G that link to the new generated super-node. The function MaxNeighbor(G, weight, snode) has the same function as MaxNeighbor(G, weight), but it only search the edges that link to the super-node snode.

To minimize the complexity of the interconnections, the partitioning is performed from the highest weight to the lowest weight. The weight used by these iterative steps is called processing weight. At the beginning, we calculate the common neighbors of an edge at the current processing weight value, and try to find the edge that has the maximum number of common neighbors. Then we the two nodes that linked by the edge are merged to generate a new super-node, snode. Thus, the criterions to find two nodes be merged in a compatibility graph is as follows:

1. The nodes that have edges with the highest weight will be merged first;
2. The nodes that have the maximum number of neighbors will be merged first;
3. If there are nodes that have the same maximum number of maximum neighbors, then the nodes with the highest sum of the weight with other operations will be merged first;

When two nodes are merged to a new node, we update the graph information by deleting the merged nodes, add new super-node and new edges. As we described in the above section, for each weighted edge, we associate a case number to it, then we can easily calculate the new edge information from the two old edges. For example, node x and node y are merged together, a node z is linked to both these two nodes with edge e_{xz} and e_{yz}. We have a case number 9 for edge e_{xz} and 10 for edge e_{yz}. Then after the two nodes merged, the edge that link node z and the new node will have weight as 5 and a case number as 12.

After a super-node generated, since the new generated edges may have higher weight value than current processing weight, we need to merge these nodes before processing to next weight. In this step, the new edges that link to this node are calculated, and the nodes linked by edge that has higher weight value and maximum number of common neighbors will be merged. This step is repeated until there is no edge that has higher weight than current processing weight.

The above steps are iterative, and eventually, the graph is partitioned and the nodes are merged to a set of cliques, CliqueList. When the number of the cliques equal the number of the resource allocated, we stop partition and get the binding result. The result is optimal in reduce the interconnections since we use interconnections, i.e. weight, as our first criterion to merge nodes.

5.3.3 Illustrative example

We will illustrate the above algorithm by a simple example in the rest of this section. As shown below, we have part of the RTL description as follows:

```
S1:  x = a - b;
…
S2: if(x>=b) x=x-b;
      else x = x +b;
…
S3:  c = a - x;
      y = b + a;
…
```

The above RTL description shows the RTL statements which contain operations in this code. And it is a style 1 RTL description in which the storage unit binding, function binding, and bus binding have not been done yet.

In the beginning of the algorithm, based on this code, we first generate a FSMD model from this

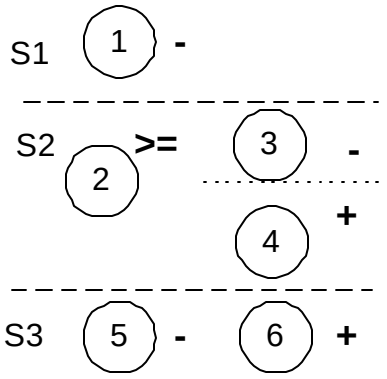


Figure 9. Operation list

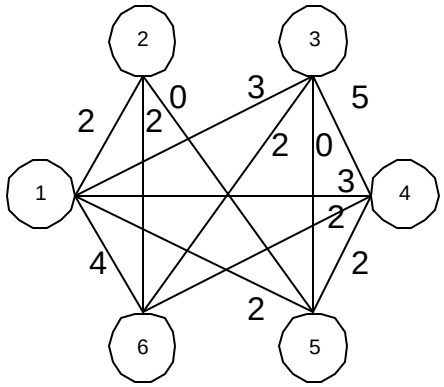


Figure 10. Compatibility Graph

RTL description with our compiler. And user will give us the resource constraint, for example, 3 ALUs, for this FSMD. Then based on this FSMD, a list of the super operation node, OpList, is created, as shown in figure 9. In the list of super operation node, each operation is assigned an identification number, which is shown in the circle, and associated with proper control step information, e.g. state information and branch statement information. Suppose that the function unit ALU can complete all kinds of operations in 1 control step, then a compatibility graph G is created based on the super operation node list OpList. Figure 10 shows the compatibility of this compatibility graph.

After the graph is created, we partition the compatibility graph into cliques as the above algorithm described. Figure 11 shows the partitioning process. The compatibility graph is partitioned from the highest weight 5 to the lowest weight 0. As shown in figure 11(a), in the beginning we partition the edges with the highest weight 5. In this weight, there are only one edge $e_{3,4}$, then the two nodes v_3 and v_4 are merged to formed a new super-node as figure 11(b) shows. Note that node v_3 and v4 can be mapped to the same resource even they are in the same state, because they are in different branch of an **IF** statement, which means they are in different control steps. After update the graph information, there is no edges with weight 5 left in the graph, so we advance to edges with weight 4. Then from figure 11(b) we can see that the edge link node v_1 and v_6 have the weight of 4. So we merge node v_1 and node v_6, and update the graph information again as shown in figure 11(c). After update the graph information, there is no edge of weight 4 left in the graph, and we advance to edges with weight 3. The two super-nodes v_{16} and v_{34} that have an edge link them with weight 3, hence they are merged to formed

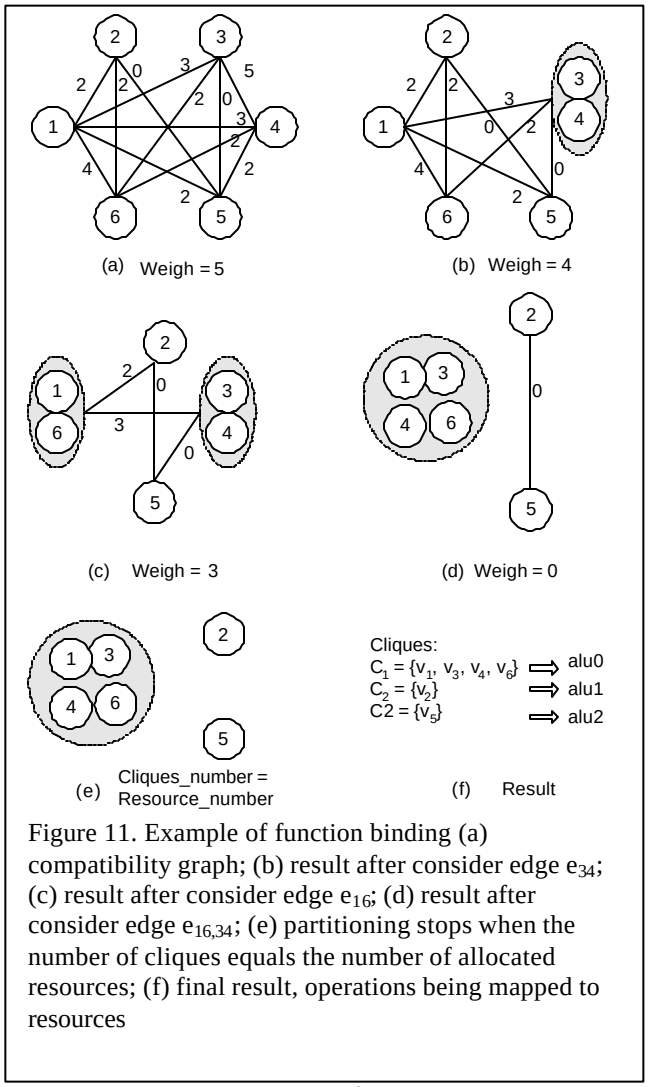


Figure 11. Example of function binding (a) compatibility graph; (b) result after consider edge e_{34}; (c) result after consider edge e_{16}; (d) result after consider edge $e_{16,34}$; (e) partitioning stops when the number of cliques equals the number of allocated resources; (f) final result, operations being mapped to resources

super-node v_{1346} as shown in figure 11(d). In all above merge steps, we compare the clique number to the allocated resources number after each merge is completed. When we find the clique number equals the allocated resources

number, we stop merge nodes and output the result. As in figure 11(d), the node v_2 and v_6 could be merged further more, but since the clique number is 3, which equals to the allocated resources number, we stop partitioning and generated a list of cliques as in figure 11(e). The binding result give us the cliques $C_1 = \{v_1, v_3, v_4, v_6\}$, $C_2 = \{v_2\}$, $C_2 = \{v_5\}$. Finally we update the information in each operation node and map the operations in the same clique to the same resource unit, as shown in figure 11(f). And we map this result to the target architecture as shown in figure 12.

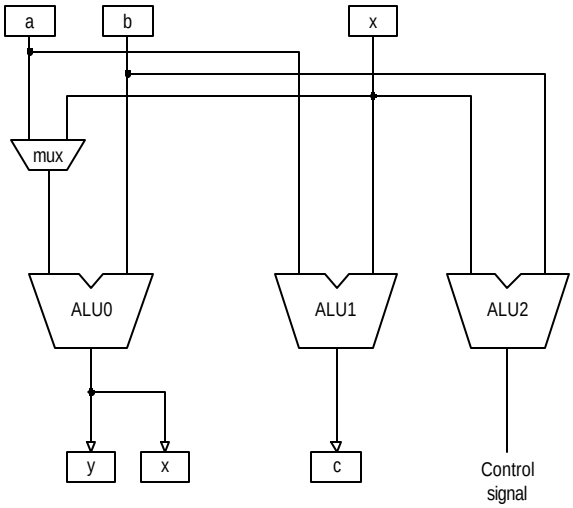


Figure 12: Target architecture by using the graph partition based algorithm with resource constraint(3 ALUs)

5.4 Summary

The graph partition based algorithm is based on the clique partition algorithm. However, it is different with the tradition clique partition algorithm in the following 2 points:

1. The compatibility graph in our algorithm extend the edge with several 6 weight threshold, which is similar to the extended clique algorithm by Paulin and Knight [PaKn89];
2. We apply resource constraint, i.e. the number of function units, in our algorithm.

While the resource unconstraint function binding algorithms try to minimize the number of resources utilized, our goal is to minimize the complexity of the interconnection. This is due to the fact that in the embedded system design, the number of the interconnections is much more than the number of the function units. Hence, we can save more cost by minimizing the interconnections than minimizing the function units.

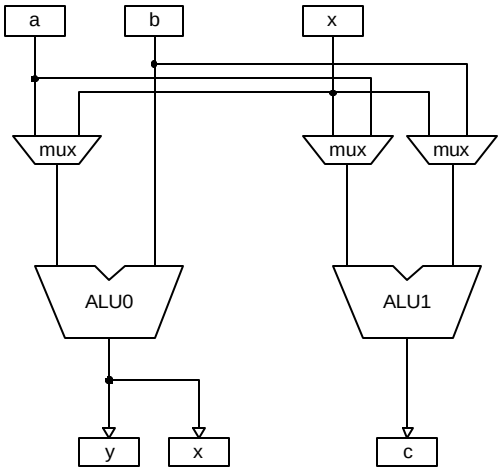


Figure 13. Target architecture by using the unconstraint clique partition algorithm

As for the above example, the clique partition without resource constraint will give us the result with two cliques: $C_1 = \{v_1, v_3, v_4, v_6\}$ and $C_2 = \{v_2, v_5\}$. And the generated target architecture is shown in figure 13 (suppose we do function binding at style 1 RTL model before storage unit binding and bus binding).

Compare figure 12 to figure 13, we can see that figure 12 has less mux-gates and wires than figure 13. Table 1 summarizes the results of these two approach, resource unconstraint and resource constraint. From the table, we can see that by using an additional ALU, we can reduce 2 mux gates in the architecture. Hence it is quite possible that in a large scale design, the gain in the interconnections may larger than the cost of several additional function units.

	No. FUs	No. Muxes	No. Drivers
Resource Unconstraint	2	3	3
Resource constraint	3	1	3

Table 1. Resource unconstraint/constraint result

We can modify our algorithm furthermore by calculating the cost of the interconnections and the cost of function units, and we select the minimum cost solution as our result.

Example	No. of states	No. of operators	No. of Fus(after binding)		No. of Interconnections (after binding)	
			No. of ALU	No. Of Shifter	No. of Mux	No. Of bus driver
No resource constraint	8	4	1	1	2	3
Resource constraint(2 ALUs)	8	4	2	1	0	3
Resource constraint(3 ALUs)	8	4	3	1	0	3

Table 2. Experimental results for One's counter

6 Experimental results

We implemented our function binding algorithm, which uses the graph partition based algorithm, in C++ codes and integrated it to our RTL synthesis tool, which can perform scheduling, storage unit binding, function binding, and interconnection binding in arbitrary order.

We implement a simple RTL library which contains several basic RTL components, such as register, register files, ALU, multiplier, buses etc. Then we applied our algorithm on several examples.

Table 2 and figure 14 gives the result on the One's Counter example. One's Counter is used to count the number of '1' for a given number. The RTL description for One's Counter has 8 states. Our result shows that we can perform function binding on different styles of RTL successfully. A comparison of these results with different allocated ALUs is given in table 2. As in table 2, the second and third columns are the total numbers of states and operators in the RTL description. The fourth column shows the function units used in the result. The fifth column is the total number of interconnections, i.e. muxs, bus drivers, after function is performed. We can see that with constraint of 2 ALUs, we can get a gain of 2 mux gates in the result.

7 Conclusion

In this report, we discuss our function binding algorithm in RTL synthesis. We proposed two algorithms to do function binding in the synthesis flow. Our goal is performing binding with resource constraint and minimize the cost of the interconnections. Our two algorithms are graph partition based algorithm and seed constructive based algorithm. Our comparison shows that the former algorithm can give us a better result, though the seed constructive based one has major problem on select proper seeds.

We integrated our graph partition based algorithm in our RTL synthesis tool. The experimental results show that the function binding algorithm can work effectively in different styles of RTL description. It also can work in different in different order of binding process in RTL synthesis. And it can reduce the cost of the interconnection effectively. The future works include support multi-cycle operations which is not supported by now.

References

[GZDG00]D. Gajski er al. *SpecC: Specification Language and Design Methodology*, Kluwer Academic Publishers, 2000

[GERS00]A.Gerstlauer: *SpecC Modeling Guidelines* , University of California, Irvine, 2000

[GWDL92]D. Gajski er al. *High level synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers, 1992

[Acc01] Accellera C/C++ working Group. RTL semantics: Draft Specification. Feb. 2001

[DShinGa01] D. Shin, D. Gajski:

, ICS technical report [01-50], University of California, Irvine, 2001

[TsSi86]Chia-Jeng Tseng,Daniel P.Siewiorek: *"Automated Synthesis of Data Paths on Digital Systems"*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and

Systems, vol. CAD-5, no.3, pp.379-395, July, 1986

[HaoyGa01] H. Yu, D. Gajskier: *Bus Bindnign in RTL synthesis*, ICS technical report[01-37], University of California, Irvine, 2001.

[CLiu77] C.Liu: *Elements of discrete mathematics.* New York , McGraw-Hill, 1977

[PaKn89] P.G.Paulin and J.P. Knight: *"Force-Directed Scheduling for the Behavioral Synthesis of ASIC's"*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol.8, no.6, pp.661-679, June 1987