Interconnection Binding in RTL Design Methodology

Haobo Yu
Daniel D. Gajski

Technical Report ICS- 01-38
June 27, 2001

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

{ haoboy, gajski}@ics.uci.edu
http://www.ics.uci.edu/~haoboy

Abstract

Bus-based architecture has better performance than mux-based architecture in large design. Unfortunately no commercial High-Level/RTL synthesis tools exists today in EDA industry enable the bus based architecture synthesis. This report describes the interconnection binding part in our RTL refinement tool. The proposed algorithm is based on the clique partitioning algorithm to minimize the total number of buses needed for the datapath. It integrates with other modules in our RTL refinement tool to generate RTL style 2-4 code automatically.

List of Figures

Interconnection Binding in RTL Design Methodology

Haobo Yu, Dianel D. Gajski
Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA

Abstract

Bus-based architecture has better performance than mux-based architecture in large design. In this paper we introduce interconnection binding in a new RTL design methodology. The proposed methodology uses the bus-based architecture and supports pipelined /multi-cycle operations and storage units. By using the tools supporting our methodology, the user can explore the bus-based architecture design space efficiently.

1. Introduction

Much research for High-level/RTL synthesis has been done since 1980s. Currently, many commercial and academical high-level/RTL synthesis tools exist in electronic design automation market but the design community wouldn't integrate them into its design methodology and design flow, because 1) they can support only several limited architectures, 2) they are lack of interaction between them and the designers, and 3) the quality of the design which they generates is worse than that of manual design.

Until now, much research on high level synthesis was focused on mux-based target architecture, however, bus-based architecture has better performance than mux-based architecture in large design. Unfortunately no commercial synthesis tool exists today in EDA industry enables the bus based architecture synthesis. Also, there's limited architecture support, for example, the registers are used as storage unit rather than register files and memories, which is commonly used in manual design.

To make the synthesis tools accepted by the design community, we introduce a new RTL design methodology with corresponding RTL refinement tool [ShGa01B]. The proposed RTL design methodology supports the RTL semantics by Accellera C/C++ working Group[Acc01].

This paper focuses on the interconnection binding in the purposed RTL design methodology. The new methodology enables automatic bus binding and let the user explore the design space. This methodology differs from previous interconnection binding approaches in that it does not try to minimize the total number of buses used in target architecture, rather, the number of buses in target architecture is assigned by the user. By this way, our methodology can let the user rather than the synthesis tools explore the interconnection binding design space. Thus, the synthesis result will be better and the design quality will be close to that of manual design.

2. RTL Design methodology

This section describes the RTL design methodology and the corresponding test tool. The proposed methodology has the following features that make the tool achieve good synthesis quality:

- Comply with the proposed Accella RTL semantics[Acc01];
- Support bus based universal processor architecture;
- Support user-tool interaction, enables user directed design space exploration ;

2.1 RTL refinement flow

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 and is obtained by scheduling the operations in the behavioral code into clock cycles. The structural RTL view of the implementation model explicitly models the allocation of RTL components, the

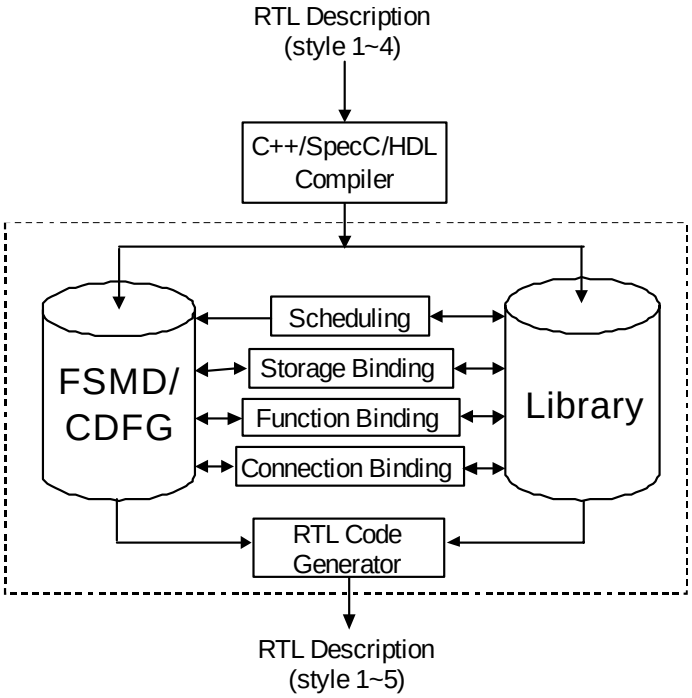

Figure 1 RTL design refinement flow

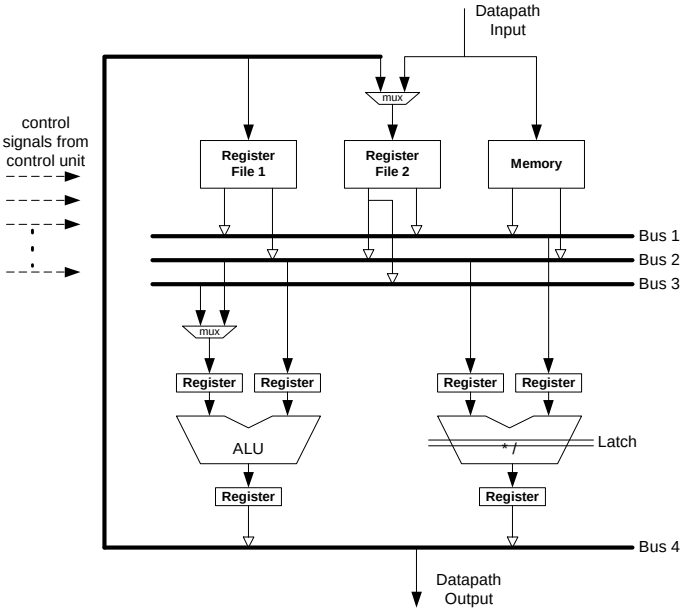

Figure 2 Bus based architecture

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 proposed by Accellera RTL semantics[Acc01]: 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 1 describes the RTL refine flow in our RTL design methodology. We generate the FSMD/CDFG from the C++/SpecC/HDL input as our internal representation for refinement. Each refine step is based on FSMD/CDFG data structure, where each state has its own Control/Data Flow Graph [ShGa01A]. 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 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.

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.

2.2 Target architecture

The target architecture of our RTL design methodology is bus-based universal processor architecture[Acc01], as shown in Figure 2. It consists of storage units, function units, buses, bus drivers and multiplexers. The storage units can be composed of registers, register files and memories with different latency and pipeline scheme. The function units are pipelined or multi-cycled.

2.3 User interaction and design space exploration

Most high level synthesis tools were built to do everything automatically. Research was focused on how to minimize the number of operation units,

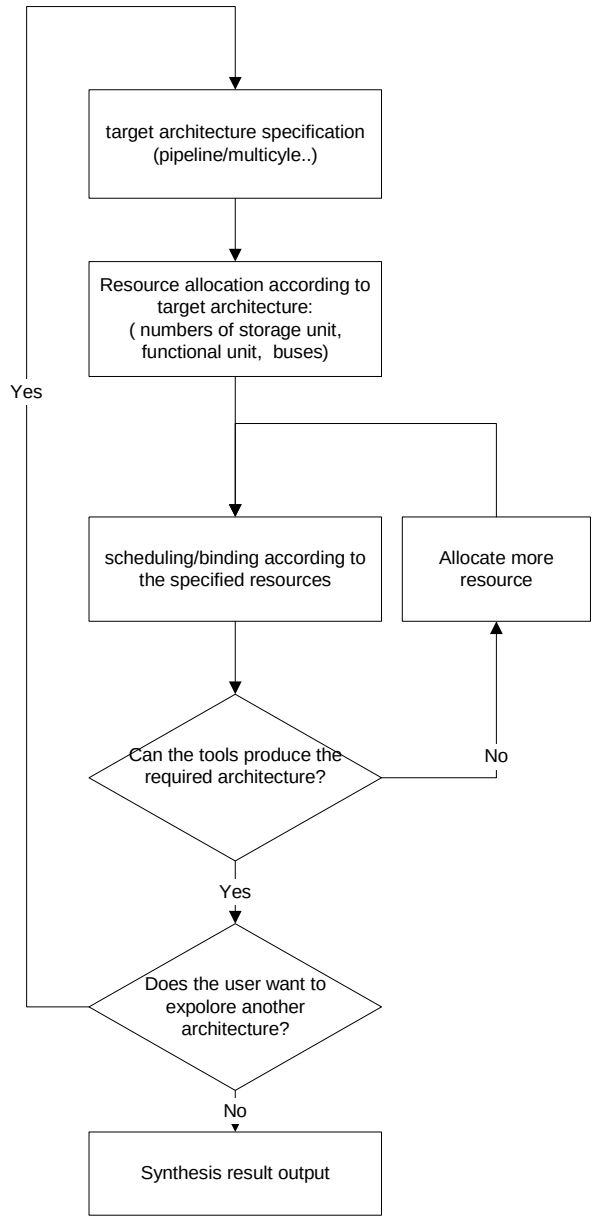

Figure 3: user interaction and synthesis procedure

storage units and interconnection units (multiplexes and number of connections) [TsSi83]. Nearly all the synthesis tools are trying to explore the design space automatically without human intervention.

But all these automatical approaches, though good in intention, failed to achieve satisfactory synthesis quality. These automatic tools can't explore such broad design space by themselves. We need the designer to participate in the design space exploration process, because human has more specific knowledge and experience about the direction of exploration.

Figure 3 shows our user directed design space exploration. First, the user specifies the target architecture and allocates the corresponding resource according to the target architecture, then our synthesis tool try to do scheduling/binding based on these specified resources, if our tool failed to produce the synthesis result, the user allocate more resources, this interaction is repeated until the tool can produce the required architecture. Then, the user can try another target architecture and the whole process is repeated again, by this way, we give the user more freedom to explore the design space. Since the experienced designer has much knowledge about the design, his input will lead to better target and synthesis result than the automatic process.

3. Interconnection binding

In the datapath synthesis, we need to transfer each register output to the input of a functional unit and each functional unit output to the input of a register. Interconnection binding maps the data transfers to the interconnection paths (in our target architecture, they are buses). The objective of interconnection binding is to maximize the sharing of interconnection units, while still supporting the conflict-free data transfers required by the register-transfer description. Since the connections of a datapath usually occupy a substantial silicon area in a microchip, we can reduce the cost of datapath connection by merging several connections into a bus, which occupies less area.

In our design methodology, the number of buses is pre-allocated by the designer, and our tool tries to map all the data transfer paths to specified number of buses. Thus our goal differs from other bus binding approaches, which try to achieve minimal number of buses in the target architecture.

3.1 Quality Metrix

The quality measure for interconnection binding is to achieve minimal silicon area for the interconnections. But in the RTL level, we only have a coarse estimation of the cost, however, the number of bus drivers and multiplexers give a reasonable measurement to the interconnection cost. So we use the following measure:

Cost = *w1*Num(busdriver) + w2 *Num(mux)*

where *w1, w2* is the weight for the bus driver and multiplexer to account for the different cost between

Interconnection Binding Algorithm:
Input: FSMD(*S, N*) , *BusNumber*
Output: FSMD(*S,N*) where
$\forall$ $n_i \in$ *WireNode*,busID(n_i) =j , j=1,2$\cdots$*BusNumber*;

```
/* create a list WireList containing all wires of the datapath */
WireList  = ∅;
for each s_i ∈ S do
        for each n_j ∈ N_i do
            if n_j ∈ WireNode then
            Add2WireList(WireList, n_j, s_i );
          endif
        endfor
endfor

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

/* Partition the graph G into BusNumber number of Cliques */
Graph_partition(G, CliqueList, BusNumber);

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

Figure 4. Interconnection binding algorithm

busdriver and multplexer.

3.2 Interconnection binding algorithm

Since the number of buses to be bound is pre-allocated by the user, our algorithm differs from other interconnection binding algorithm. We do not seek to achieve minimal number of buses in target architecture, rather, we will try to bind the data transfer paths into the specified number of buses, at the same time, we try to minimize the quality matrix of the bus binding result.

The proposed interconnection binding algorithm is based on two facts about data transfer paths:

1. Data transfer paths which are never used simultaneously can be bound together to connect to the same bus;
2. In order to minimize the quality matrix, it is beneficial to bind those wires which have the same source or destine to the same sink;

There are two input to the interconnection algorithm: the first one is the CDFG data structure [ShGa01], which is generated from RTL style 3(after storage binding and function binding) code. The data structure includes CDFG data structure for every state in FSMD. Each CDFG data structure contains the storage node, wire node(representing data transfer path in each state) and function unit node [ShGa01]. The second input is the number of bus available for interconnection binding. This number is determined in the allocation process, based on the datapath architecture specified by the user.

In our interconnection binding algorithm, the wire nodes in each CDFG in the FSMD are extracted to form a super-wire list with proper information (e.g. states in which it belongs and a unique ID for each wire), then we generate a graph based on this super-wire list and partition this graph into several cliques(the total number of cliques equals the input bus number). The wire nodes in each clique are mapped into same bus.

Figure 4 describes our interconnection binding algorithm. The algorithm begins with an input FSMD data structure generated from the RTL description. Let F = (*S, N*) denote a FSMD, where *S* is the set of the states in FSMD and *N* is the set of nodes (storage nodes, operation nodes, and wire nodes). Each state $s_i \in S$ contains a subset of nodes $N_i \subseteq N$. The designer specify the total bus number (*BusNumer*) for the target architecture. The problem of interconnection binding is to map each wire node $n_i \in N$ to a bus resource {busID| busID = *1…BusNumer*} and minimize the total number of buses used. The input is FSMD F = (*S, N*) generated from the RTL description. Each state s_i of the FSMD($s_i \in S$) contains a subset of nodes $N_i \subseteq N$. *WireNode* is the set of all wire nodes in FSMD and *WireList* is the set of the super-wire nodes in which every super-wire node contains the corresponding wire node, the states in which the wire is used, and a unique identification number for this node. G = (*V, E*) is a graph, where *V*={v_i| *i*=1,2¡-total number of elements in *WireList*} is the set of all super-wire nodes in *WireList* and *E* = {e_{ij}| *i,j*= 1,2¡-total number elements in *WireList*} is the set of all weighted edges which link vertices in V. The edge e_{ij} exists in the graph if and only if two vertices v_i and v_j (i.e. two wire nodes in the CDFG data structure) can be mapped to the same bus without conflict. Also, all the edge has weight. We assign weight 1 to an edge if the two vertices (corresponding to two wires) connected by this edge have the same source or destination. A *CliqueList* is a set of clique, where each clique $C_i \subseteq$ *CliqueList* contains a set of vertices in which every two vertices are compatible with each other(i.e. the two wires are used in different clock cycles)

```
CompWeight(sn_i, sn_j)
{
if  UsedInSameState(sn_i, sn_j)
   if SameSourceDest(sn_i, sn_j)
         weight=1;
   else
         weight=-1;
   endif
else
   if SameSourceDest(sn_i, sn_j)
         weight=1;
   else
         weight=0;
   endif
endif
return weight;
}
```

Figure 5. Weight calculation

- Function Add2WireList(*WireList, n_j, s_i*) first create a super-wire node sn_i, from the wire node n_i and the state information *si*, then add the super-wire sn_i to *WireList*.
- Function CompWeight(sn_i, sn_j) check whether two node sn_i, sn_j is compatible with each other , and calculate the weight of the ege connecting these two nodes if they are compatible;
- Function Add2Graph(*G, i, j,* e_{ij}) adds an weighted edge e_{ij} to a graph G(*V, E*), where *i* and *j* are two vertices of the graph *G* .
- Function Graph_partition(*G, CliqueList, BusNum*) parititions the graph into *BusNum* number of Cliques. *CliqueList* contains the partitioned clique list of graph *G*.
- Function SetBindingInfo(sn_j, *i*) updates the binding information in the nodes sn_j by assign the bus resource number *i* to that node.

3.3. Compatibility graph

We generate a compatibility graphs for all the wires(data transfer paths). Each vertex in the graph represents a wire between the register and the functional unit. If two wires are not used in the same time, the two corresponding vertices in the compatibility graph are connected by an edge. For every two wires sharing the common source or destination in the datapath, we assign weight 1 to the edge which connects the corresponding two vertices in the compatibility graph. The remaining edges are

```
Graph_partition(G, CliqueList, BusNumber);
    weight=1;
    while ((weight >= 0 ) and (CliqueNumber > BusNumber))do
      while ( (Edge(weight) ) and (CliqueNumber > BusNumber))do
          e_{i,j} = Pick2Node (G, weight);
          CliqueNumber = MergeNode (G, e_{ij,} weight, CliqueList);
       endwhile
    weight = weight -1;
    endwhile
```

Figure 6 graph partitioning algorithm

assigned weight 0. Also, those interconnection wires with the common source or destination, even when they are used concurrently, can still share the same bus. Therefore, we should assign weight 1 to the corresponding edges in the graph. Figure 5 is the function to calculate the weight in the compatibility graph.

3.4. Graph partitioning algorithms

After we get the compatibility graph, we will partition the graph into cliques, each clique contains a subset of the vertices in which every vertex is compatible with each other. Our algorithm differs from other clique partition algorithm for interconnection binding in that we do not seek to get the minimal number of cliques for the compatibility graph. In our algorithm, the number of cliques is predetermined and we should partition the graph into specified number of cliques.

The algorithm is a modified algorithm proposed by [TsSi83]. We will try to combine the vertices that are connected by edges with weight 1 first, that means the wires with common source or common destination will share the same bus, thus saving bus drivers or mux used in datapath.

The algorithm is shown in figure 6, it works as follows:

1. In the subgraph of G where all edges is weight 1, the function Pick2Node(*G, weight*) pick two vertices and return the edge $e_{i,j}$ connecting these two nodes;
2. MergeNode (*G, $e_{ij,}$ weight*) deletes all edges that are connected to the two vertices connected by $e_{i,j}$ and combines the two nodes to form a super vertex, then recalculate the weight of all the edges that are connected to the new super vertex in graph *G*;
3. Check the total number of cliques in graph G, if the number is same as *BusNumber*, the algorithm stops and return the *CliqueList*;
4. Repeat 1-3 until there's no weight 1 edge in the graph;
5. Now we got a graph with only weight 0 edge, repeat the same 1,2,3,4 process on this graph until there's no edge in the graph or the total numer of cliques is same as *BusNumber*

3.4.1 Weight calculation after merge two node

In step 2,we see that after combining two nodes, we should calculate the weights of edges that connect the new super vertex with the other vertex in graph *G*. For example, if we combine *node1, node2* into a new vertex *nodex* in the graph,

nodex= combination (node1,node2);

for any *nodey* ∈graph we want to calculate weight(nodex, nodey) there are 9 conditions that will occur:

case 1:
weight(node1,*nodey*)=1 and
weight(node2,*nodey*)=1
case 2:
weight(node1,*nodey*)=1 and
weight(node2,*nodey*)=0
case 3:
weight(node1,*nodey*)=0 and
weight(node2,*nodey*)=1

in these three cases , we still want to combine nodey with the nodex first,so we make

weight(nodex,*nodey*)=1;
case 4:
weight(node1,*nodey*)=0 and
weight(node2,*nodey*)=0
of course,the result is
weight(nodex,*nodey*)=0;

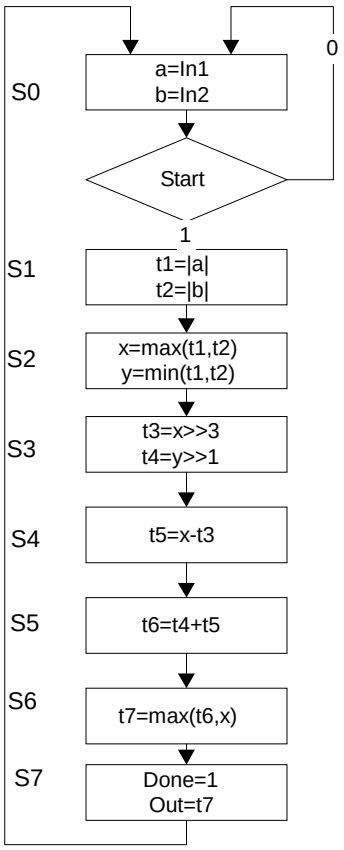

Figure 7 SRA ASM Chart

	S0	S1	S2	S3	S4	S5	S6	S7
A								X
B			X				X	
C		X	X				X	
D			X		X			
E						X		
F		X	X		X	X		
G				X				
H				X				

Figure 8 Connectivity usage table

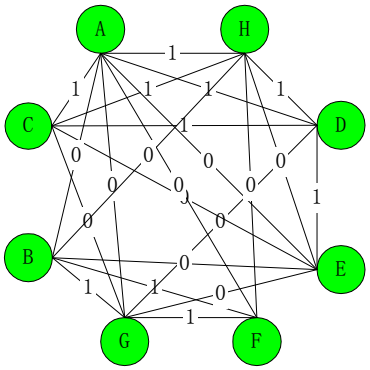

Figure 10 Compatibility graph

case 5:
weight(node1,*nodey*)=1 and
(node2,*nodey*) is not an edge;
case 6:
weight(node1,*nodey*)=0 and
(node2,*nodey*) is not an edge;
case 7:
(node1,*nodey*) is not an edge;
weight(node2,*nodey*)=0 ;
case 8:
(node1,*nodey*) is not an edge;
weight(node2,*nodey*)=1 ;
case 9:
(node1,*nodey*) is not an edge;
(node2,*nodey*) is not an edge;
nodex is in conflict with nodey

we do not make an edge between nodex and nodey in the resulting graph;

Also, the function MergeNode (*G, e_{ij}, weight*) in step 2 select two nodes from the subgraph of G where all edges have weight value *weight.* Here's the criteria for selecting candidate nodes to merge.

3.4.2 Node selecting criterias:

1. Total number of common neighbors, each supervertex is counted one time, if there are several candidate node pairs which have same number of common neighors, then go to criteria 2;
2. For each of the candidate vertex pairs, calculate the weight sum of those edges which connect the vertex in each candidate vertex pair and the common neighbor node, select the vertex pair with maximum weight sum; if there are several candidate node pairs which have same weight sum, then go to criteria 3:
3. We want to distribute the vertices uniformly among the cliques, so we choose to combine the supervertex pairs with less total number of vertices, if there are several candidate super vertex pairs which have same total number of vertices, then go to criteria 4;
4. We calculate the total number of common neighbors again, each supervertex is counted N times, where N is total number of vertices in

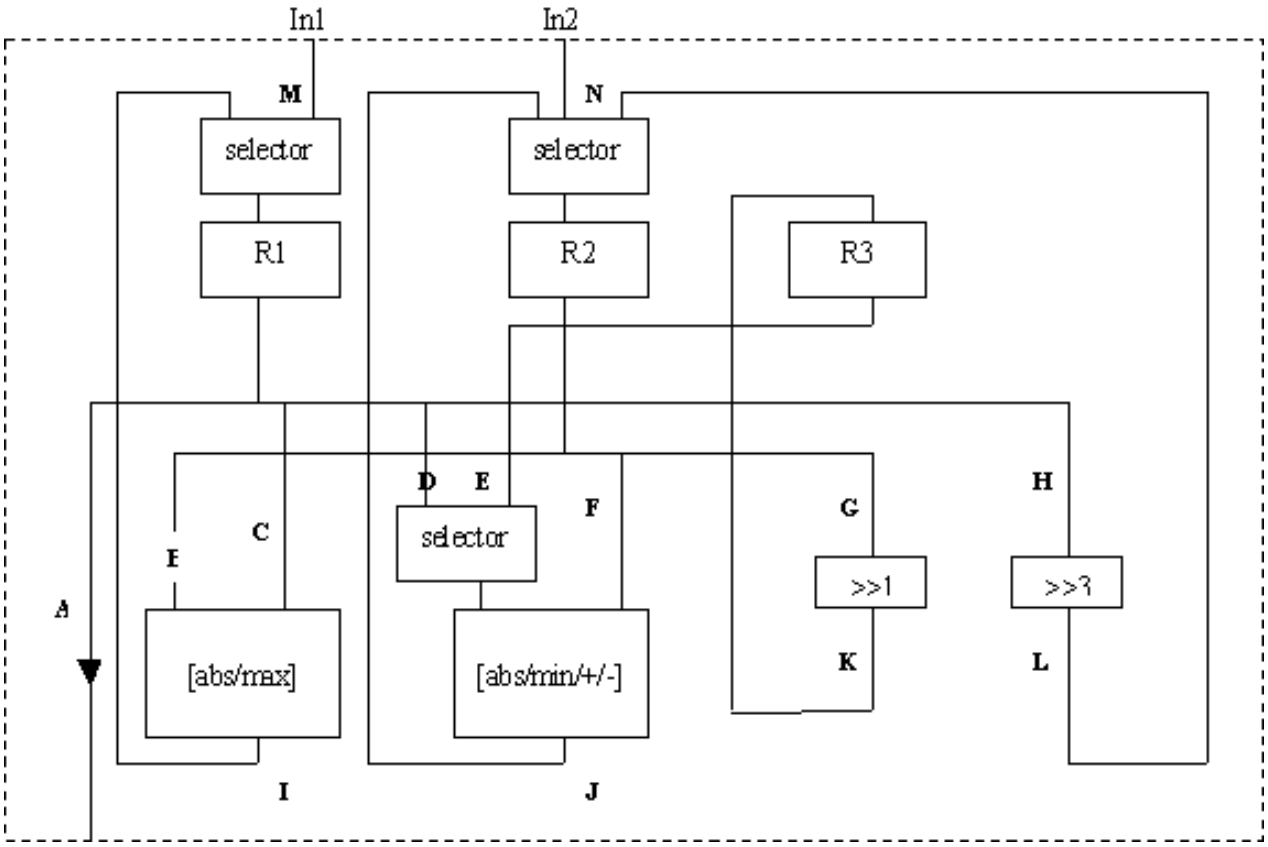

Figure 9 Datapath for SRA

each supernode;

If after applying criteria 1, 2,3, 4 , we still have several candidates, we select the vertex pair to combine randomly.

4. An illustrative example

We will illustrate our interconnection binding algorithm by the square root approximation (SRA) example from the book [Gajs97]. The SRA ASIC is designed to compute the square root approximation of two signed integers, a and b, by the following formula:

max(((0.875x + 0.5y),x)

where x =max(|a|, |b|), and y = minx(|a|, |b|).

The ASM chart[Gajs97] of the SRA ASIC is shown in figure 7. This ASIC has two input ports, In1 and In2 which are used to read integers a and b, and one output port Out. This ASIC reads the input ports and starts the computation whenever the input control signal Start becomes equal to 1. In state s1, it computes the absolute values a and b and in s2 it assigns the maximum of these two values to x and the minimum to y. In state s3 it shifts x three positions to the right to obtain 0.125x and y one position to obtain 0.5y. The ASIC calculates the 0.875x by subtracting 0.125x from x in state s4. In state S5 it adds 0.875x and 0.5y, while in state s7 it computes the maximum of x and the expression 0.875x +0.5y. In state s7, the ASIC produces the result and makes it available through the Out port for one clock cycle. At the same time, it sets the control signal Done to 1, in order to signal to the environment that the data that has appeared at the Out port is a valid result.

After we did register binding and function binding, the datapath for SRA is shown in figure 8, where we have three registers R1 , R2, R3 to hold the value of x,y,t1-t7, and two ALU(abs,min/max/+/-)

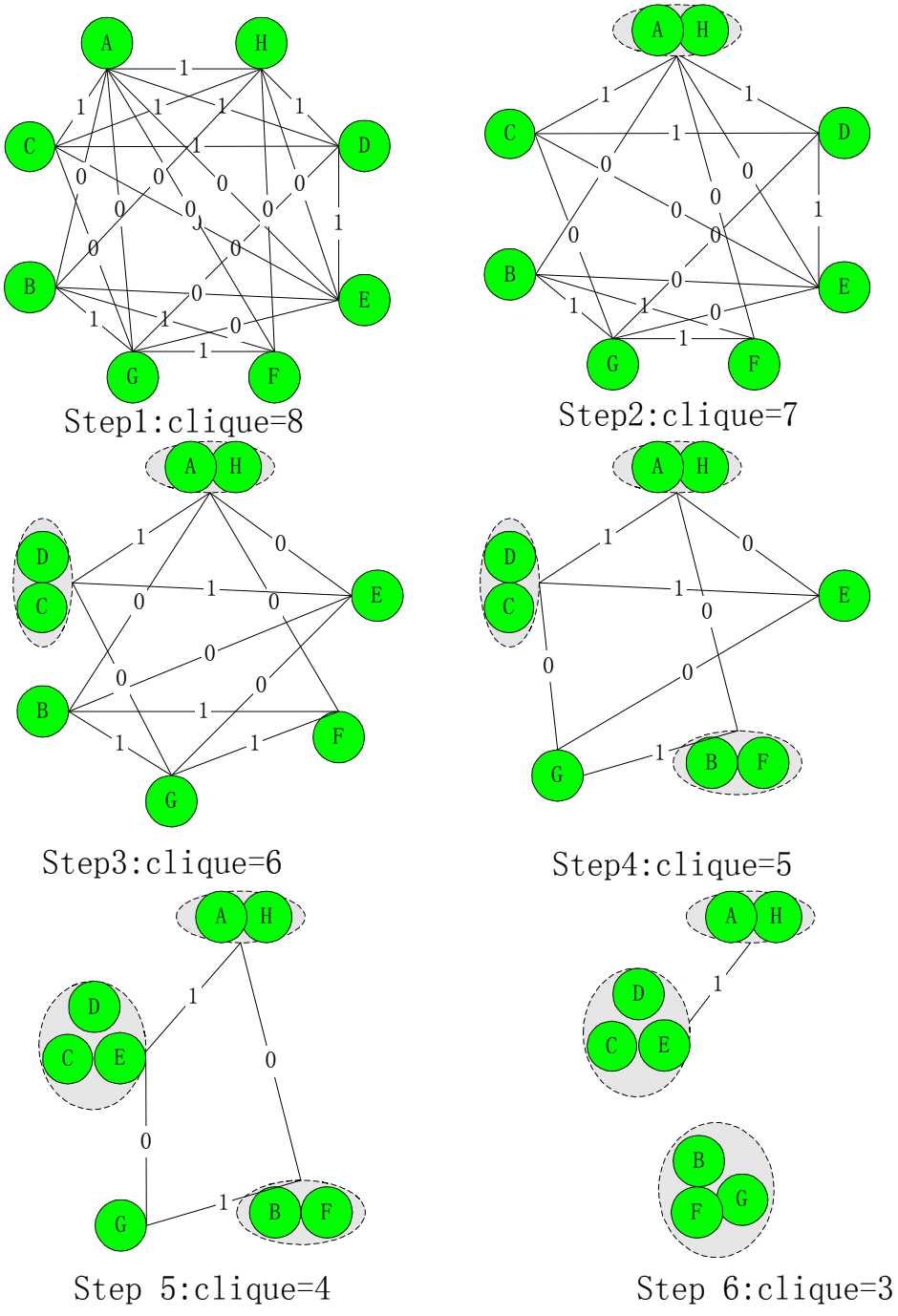

Figure 11: graph paritioning

and two shifter(>>1,>>3) as functional units.

Now we need to bind the wires A, B, C, D, E, F, G,H,I,J,K,L,M in the datapath.Since we only need this example to demonstrate the algorithm itself, for simplity's sake, we only bind the input wiress A, B, , D, E, F, G, H in the datapath.Our interconnection binding algorithm works as follows:

1. Create the connection usage table, as shown in figure 9;
2. Create the super-wire node (by adding state

information associated with each wire and unique identification number) for wire A, B, C, D, E, F,G, H and add them to WireList ;Add these 8 vertices to the graph G;

3. From the SRA datapath, we can see that A,C,D,H share the same source R1; B,G,F share the same source R2 and D, E share the same destination ,we assign the edges that connecting these vertices weight 1;
4. From the connectivity usage table(Figure 6) we can see that A is compatible with B,E,F,G, so we assign weight 0 to edges AB,AE,AF,AG; for the same reason, we assign weight 0 to edge BE, BH, CG, CE, CH, DG, EG, EH, FH, now we have a compatibility graph G ,as shown in figure 9;

Now we will do the clique partitioning on graph G,suppose we want to bind them to three buses, that is ,there will be three cliques after graph partitioning. The whole process is illustrated in figure 11:

1. In step 1, we have a compatibility graph for the input wires of the datapath for SRA,there's 8 cliques in this graph G;
2. We will select two vertices in the graph G(step 1) to merge(the two vertices should connected by an weight 1 edge) according to the criteria in section 6.1.2: (A,C) has common neighbor D,H,G, (A,H) has common neighbor C,D,B,E F, (H,D) has common neighbor A,C,E, (C,D) has common neighbor A,H,G,E, so the pair (A,H) is selected to merge first. We delete all the edges that are connected with vertex A and H, then combine vertex A and H and form a new super vertex AH, calculate the edge weight between vertex AH and the other vertices according to the rules of section 6.1.1, the new graph G is show in step 2, where the total number of cliques is 7;
3. we choose two vertices in the graph G(step 2) to merge(the two vertices should connected by an weight 1 edge): (AH, C) has common neighbor D, E, (B,G) has common neighbor F,E, (B,F) has common neighbor AH, G; (C,D) has the common number AH, G, E, so we select to merge (C, D), and the merged graph is shown in step 3, where the total number of cliques is 6;
4. In the step 3 graph, among the vertices connected by weight 1 edge, (B,F) and (B,G) all have 2 common neighbors, but (B,F) has common neighor AH, G, (B,G) has common neighbor E, F , according to criteria 4 in 3.4.5 , we should select (B,F) to merge, we get step 4 graph, where the total number of cliques is 5;
5. In the step 4 graph, among the vertices connected by weight 1 edge, (CD,E) have the maximal number of common neighbors, so (B,F) is selected to merge, we get step 5 graph, where the total number of cliques is 4;
6. In the step 5 graph, there's two edges have weight 1: (CDE, AH) and (BF, G) ,they have same total number of common neighbors, but (CDE,AH) , if combined, will have 5 nodes in the clique, while (BF,G), if combined, will have 3 nodes,according to criteria 3 in 3.4.5, we choose to combine (BF,G) first, we get step 6 graph;, where the total number of cliques is 3;

Now, we have get the three cliques, we can assign bus1 to {A,H }, bus 2 to {C,D,E}, bus 3 to {B,G,F}.

5. Data structure

Our binding algorithm is based on the internal representation of SpecC RTL data structure. The most important data structure is the FSMD/CDFG data structure. It is designed according to accellara RTL semantics.Detailed explanation of the CDFG data structure can be found on [ShGa01].

The interconnection binding algorithm begins with input FSMD/CDFG data structure, which is generated from the RTL description. And after resource allocation (register binding, function unit binding and interconnection binding), the given resource has been added to the FSMD/CDFG data structure. Finally, the netlist generater can produce style 5 RTL code from the FSMD/CDFG data structure.

Also, In the clique partition algorithm, the graph is implemented as two list data structure: one for all the nodes in the graph, the other for all the edges in the graph.

6. Experimental Results

We implemented our interconnection binding algorithm on 1500 lines of C++ code. Our code is a part of the RTL refinement tool, which is used to demonstrate our RTL design methodology. The RTL refinement tool can perform the scheduling, storage binding, function binding and interconnection binding in arbitrary order. Because our code is based on the same RTL internal representation [ShGa01A], our work integrate easily with scheduling, register binding and function binding code of the RTL refinement tool.

Examples	One's Counter						SRA		
Resources	1RF,1 ALU,1 Shifter, 3 buses			2 RF, 2 ALU,1 shift, 6 buses			2 RF, 2 Shift , 3 bus, 2 ALU		
Binding	# bus driver	# MUX	Cost	# bus driver	# MUX	Cost	# bus driver	# MUX	Cost
Our algorithm	4	2	5	6	3	7.5	8	2	9
Manual	4	2	5	6	3	7.5	6	2	7
Full connectivity	6	4	8	10	4	12	13	4	15

Table 1: Experimental Result

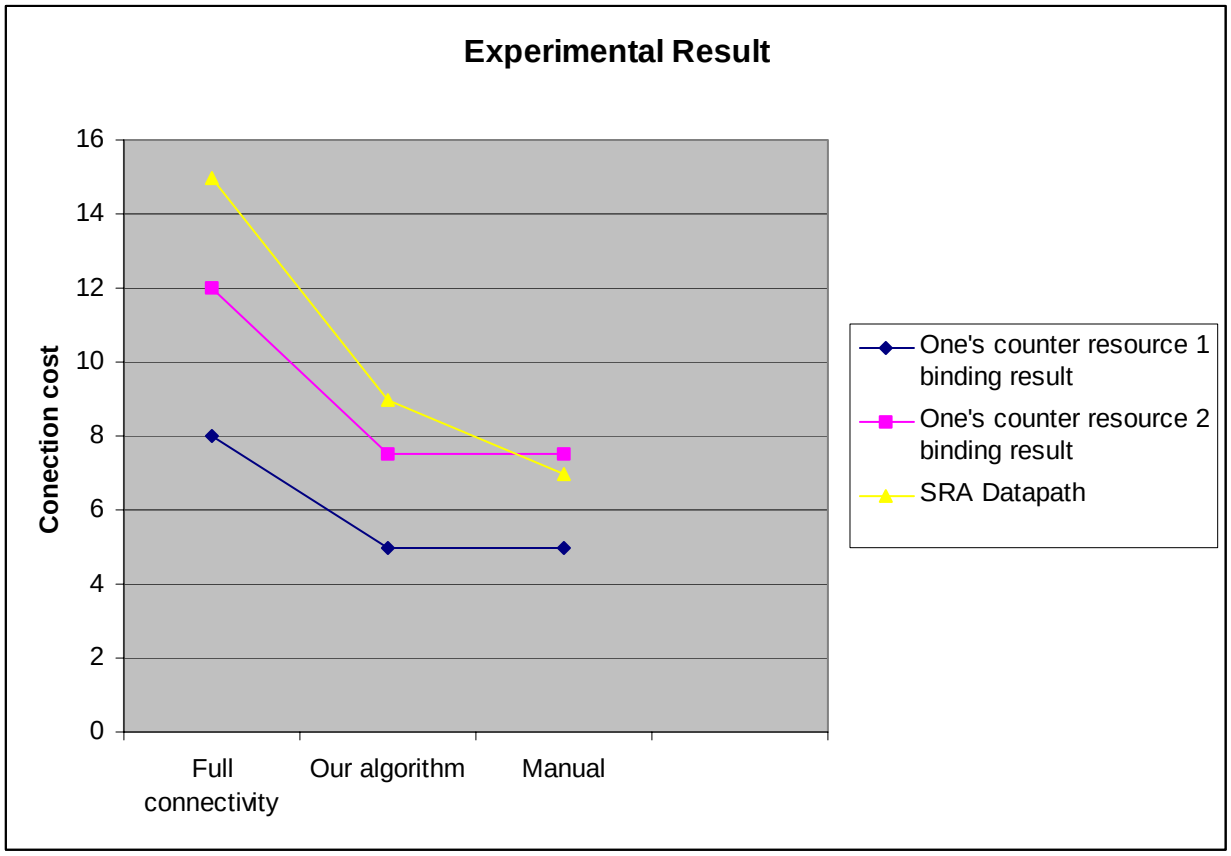

Figure 12 compare the different binding approaches

The first test example is the ones' counter, which is used to count the number of '1' in a given number. The second is the SRA. We applied our algorithm to convert the style 3 RTL code for one's counter to style 4 RTL code based on different resource allocation input.

Table 1 shows the binding result for three examples using different resources as input. In figure 12, we compare the binding cost using three approaches: full connectivity, our algorithm and manual binding. It can be seen from this graph that our algorithm can get nearly the same binding cost as manual binding.

10. Conclusion

We have presented a procedure to do interconnection binding targeting bus-based architecture in the datapath. This procedure is combined with scheduling, register binding and function binding procedure to form a RTL level refinement system.

Further research work will be extended to enable the datapath/control pipeline, where the function unit can accept data at every clock cycle to enable maximum output.

References

[Acc01]Accellera C/C++ working group. RTL Semantics : Draft Specification, Febuary ,2001.

[TsSi83]Chia-Jeng Tseng, Daniel P.Siewiorek, Facet:A Procedure for the Automated Ssynthesis of Digital Systems: 20th Design Automation Conference

[ShGa01A]Dongwan Shin. D.D. Gajski, *Internal Representation for SpecC RTL*, University of California, Irvine, Technical Report ICS-00-50, June 2001

[ShGa01B]Dongwan Shin. D.D. Gajski, *Scheduling in RTL Design Methodology*, University of California, Irvine, Technical Report ICS-00-51, June 2001

[Gajs00]D. Gajski *RTL Design and Methodology*, University of California, Irvine, Technical Report ICS-00-35, November 2000

[Gajs97] D. Gajski, Principles of Digital Design , Pretence Hall, 1997

[GDLW92]D. Gajski er al. *High level synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers, 1992

[Gers00] A.Gerstlauer: *SpecC Modeling Guidelines* , University of California, Irvine

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