Storage Binding in RTL synthesis

Pei Zhang
Daniel D. Gajski

Technical Report ICS-01-37
August 10th, 2001

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

{pzhang, gajski}@ics.uci.edu

Abstract

In this report, we present the implementation of storage binding which is one key process in high-level (RTL) synthesis. In previous related works, storage binding is based on isolated register, or use 0-1 integer linear programming (ILP) for multiple port memories to get optimal result. In this report, we introduce two new approaches that use graph-partitioning algorithm and grouping method to map variables into register files and memory that are normally used in industry.

Contents

1 Introduction **1**

2 Target architecture and five styles in RTL model **2**

2.1 Target architecture …………………………………………………………………….. 2

2.2 Five styles in RTL model……………………………………………………………… 2

3 Project goal **3**

4 Implementation of register binding 3

4.1 Data structure of our implementation ………………………………………………… 3

4.2 Approach one ………………………………………………………………………… 4

4.2.1 Get variable information……………………………………………………….. 4

4.2.2 Bind arrays to memories ……………………………………………………….. 4

4.2.3 Clique-partitioning……………………………………………………………... 4

4.2.4 Group cliques to register files ………………………………………………….. 6

4.2.5 Adjustment…………………………………………………………………….. 6

4.2.6 Example ……………………………………………………………………….. 7

4.3 Approach two………………………………………………………………………… 8

4.3.1 Split all variables into small groups…………………………………………… 9

4.3.2 Clique-Partitioning in small groups…………………………………………… 9

4.3.3 Adjustment…………………………………………………………………….. 10

4.3.4 Example ……………………………………………………………………… 10

5 Experiments **10**

6 Summary **12**

7 Future works **12**

Reference **13**

List of Figures

1 Target architecture …… 1
2 Synthesis tasks in RTL synthesis …… 3
3 CDFG data structure …… 4
4 The procedure of storage binding approach 1 …… 4
5 Get variable information …… 4
6 Graph-partitioning algorithm …… 6
7 CDFG of the example …… 7
8 Lifetimes of variables …… 8
9 Clique-partitioning result …… 8
10 Binding result using approach 1 …… 8
11 The procedure of storage binding approach 2 …… 8
12 Result of sorted lifetime variables …… 10
13 Result of clique-partitioning in each L_i …… 10
14 Binding result using approach 2 …… 11
15 Square-root approximation …… 11
16 Process of graph-partitioning …… 12
17 Experiment results (Approach 1) …… 12
18 Experiment results (Approach 2) …… 12

List of Tables

1 Simple example of 3 cliques …… 8
2 Variables Lifetime Table …… 11
3 Priority weight Table …… 11
4 Comparison of different approaches (1) …… 12
5 Comparison of different approaches (2) …… 12

List of Algorithm

1. Approach 1 of Storage Binding Using Multiple Port Register Files …… 5
2. Approach 2 of Storage Binding Using Multiple Port Register Files …… 9

Storage Binding in RTL Synthesis

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

Abstract

In this report, we present the implementation of storage binding which is one key task in high-level (RTL) synthesis. In previous related works, storage binding is based on isolated register, or uses 0-1 integer linear programming (ILP) for multiple port memories to get optimal result. In this report, we introduce two new approaches that use clique-partitioning algorithm and grouping method to map variables into register files and memories that are normally used in industry.

1 Introduction

High-level (RTL) synthesis is normally divided into four separate tasks: scheduling, storage binding, functional unit binding and interconnection binding. The storage binding binds the variables to the storage units, such as registers, register files and memories. Recently, there is a trend for designer to use register files other than isolated registers in the storage binding. There are several approaches for the storage binding using register files. But all of them are too complicated, time-consuming and not feasible for the large scale designs.

In this report, we describe some news approaches of storage binding using register files in high-level (RTL) synthesis.

The rest of the report is organized as follows: section 2 shows the 5 levels in RTL description; Section 3 gives the goal of this project; Section 4 describes implementation, algorithm, data structure and of the RTL storage binding. Finally, experiment results for our algorithms are given in section 5. Section 6 makes a conclusion and section 7 gives some directions of future

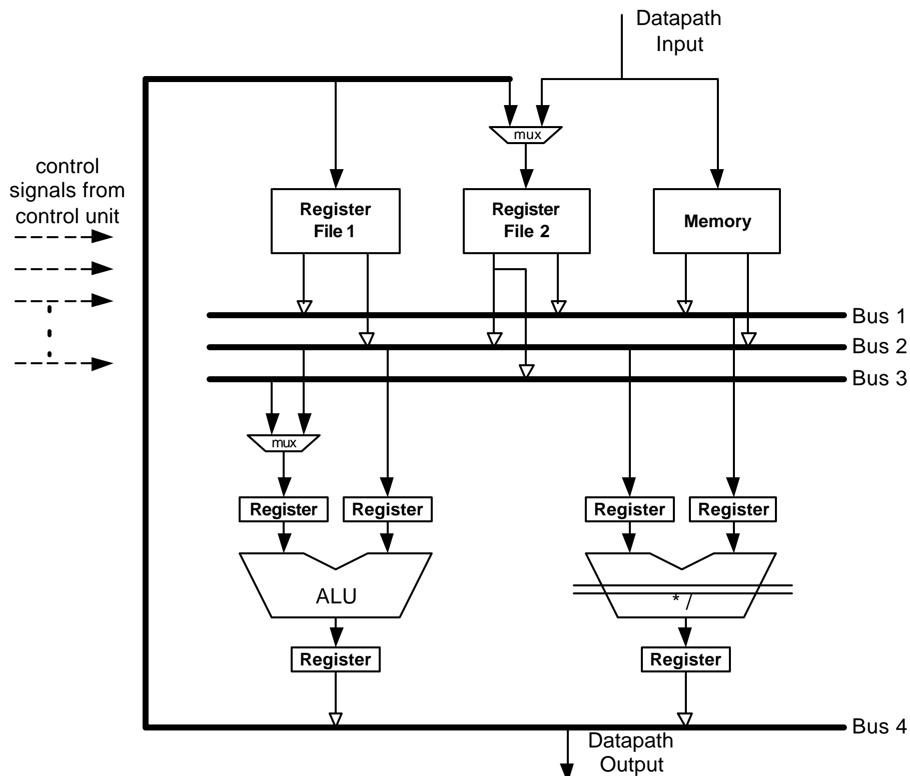

Figure 1: Target architecture

works.

2 Target architecture and five styles in RTL model

2.1 Target architecture

Our architecture is shown in figure 1. Since the RTL synthesis is focus on data-path, we only show the data-path part of a general design, which will be controlled by the control signals from control unit. It is composed with register files, memories, busses, functional units and multiplexers. Register files and memories get data from inputs and internal busses, send data to other internal busses. Then the data from busses is the input of functional units. After functional units finish their tasks, they send data to busses for Register files and memories' input, or for data-path outputs.

Here, instead of isolated register, our design is based on register files and memories as storage units because register files and memories can be more structured, modular and dense, and requires less chip area because of its regular layout structure. The registers above and below the functional units, as well as latch between the functional units, are for the purpose of pipeline.

2.2 Five styles in RTL model

We use Finite State Machine with Data (FSMD) to describe the RTL model. FSMD is an FSM with assignment statement added to each state.

The RTL model has two views: a behavioral RTL view and a structural RTL view. The behavioral RTL view specifies the operations performed in each clock cycle with explicitly modeling the units in the component's data-path and is obtained by scheduling the operations in the C code into clock cycles. The structural RTL view explicitly models the scheduling of register transfers into clock cycles the allocation and binding of operations, variables and interconnections to functional units, register files/memories and internal busses respectively. From behavior RTL view to structural RTL view, it includes the three tasks of high-level synthesis: storage binding, functional unit binding and interconnection binding.

In [GAJS00], the RTL model is divided into 5 well-defined styles to represent the refinement steps like scheduling, storage binding, functional unit binding and interconnection binding from behavioral RTL view (style 1) to structural RTL view (style 5)

- Style 1: Behavioral RTL (unmapped RTL). Behavioral RTL only specifies the change of values for some variables in each state. States, transitions and assignment statements are in no way related to any implementation. Scheduling task focuses on this style.

- Style 2: Storage-mapped RTL. The variables in style 1 can be of two types. One type is variables whose value is used in the same state in which that value is assigned. These variables will be implemented as wires or busses in the final implementation. The other type is variables whose values are assigned in one state and used in other state. The states between the value assignment and its last usage define the lifetime of each variable. These variables must be mapped to storage units such as register, register files, and memories in the final implementation. Thus style 2 represents RTL description in which the second type of variables with non-overlapping lifetimes are grouped and assigned to storage units. Storage binding task will implement the transfer from style 1 to style 2.

- Style 3: Function-mapped RTL. In style 3, the operators and/or functions with non-overlapping lifetimes are grouped into functional units, and a control encoding is assigned to each operation in the functional unit. Functional unit binding task will implement the transfer from style 2 to style 3.

- Style 4: Connection-mapped RTL. Similarly to style 2, the variables, with non-overlapping life times, that represent wires as well as inputs and outputs to storage elements and functional units are grouped and assigned to busses. Syntactically, there is no difference between wires and buses. Interconnection binding task will implement the transfer from style 3 to style 4.

- Style 5: Exposed-control (structural) RTL. In style 5, the FSMD implementation is

described in two parts: netlist of data-path components and a control unit that control the data-path components using control signals in each state.

3 Project goal

Due to the 5 styles in RTL, we divide the high-level (RTL) synthesis into several separate tasks, which include scheduling, storage binding, functional unit binding and interconnection binding (as shown in figure 2). The sequences of storage binding, functional unit binding and interconnection binding can be any order of these three tasks to make the whole synthesis task more freely.

From RTL style 1 to RTL Style 2, variables with non-overlapping lifetimes need to be grouped and assigned to storage units, such as register, register files and memory. Since the storage units usually occupy a substantial silicon area in a microchip, we generally try to reduce the number of storage units by merging several variables into a storage unit, which will lead to smaller area.

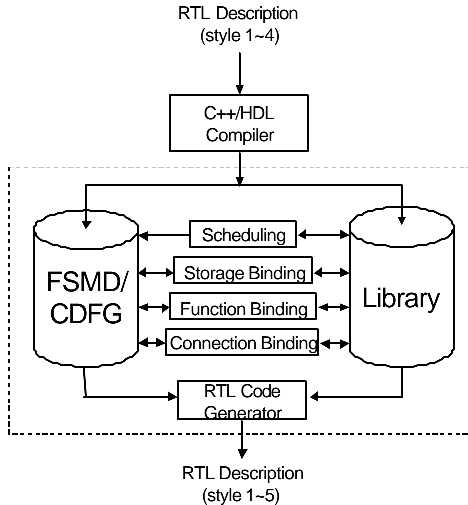

Figure 2: Synthesis tasks in RTL synthesis

Traditional storage bindings use isolated registers, which is not efficient. Here we use multi-port register files and memories as storage units.

Generally, the goal of storage binding when using register files is to design procedures to enable **fast** automatic variables to storage units binding. After storage binding, each variable with non-overlapping lifetimes will assign to the **minimum** register file modules and **minimum** number of register in each register file, as well as the **minimum cost** of interconnection.

[AHCH92] uses 0-1 integer linear programming (ILP) to group variables into multi-port memories, which is very difficult and time-consuming when the design size is in large scale. Here we introduce another two approaches, which use clique-partitioning and grouping to make the whole task much easier and to get the similar results.

In our implementation, the number of register files and memory is fixed before the storage binding task, that is so-called resource constraint storage binding. So the quality metrics is not the minimum number of register files. We can use the following metrics to compare the results:

1. The number of registers in each register files and the total number of registers in all register files;
2. The ports usage in every register files;

After the storage binding, we can give users the feedback of binding results. Users then can add/remove the number of register files or change the size of register files to have higher usage of resources.

In our implementations, we have the following assumptions:

1. All variables have the same type;
2. All given register files are the same;
3. The number of given register files are sufficient for storage binding, so we don't consider to minimize the number of the register files.

4 Implementation of storage binding

Instead of ILP, we have two approaches for the storage binding using register files. The two approaches are all based on clique-partitioning and grouping method. They are similar except that they have difference orders of step in each procedure.

4.1 Data structure of our implementations

In order to perform our works, we use CDFG as our basic data structure, which is also used in

other parts in high-level synthesis . Figure 3 give the class data structure of our CDFG.

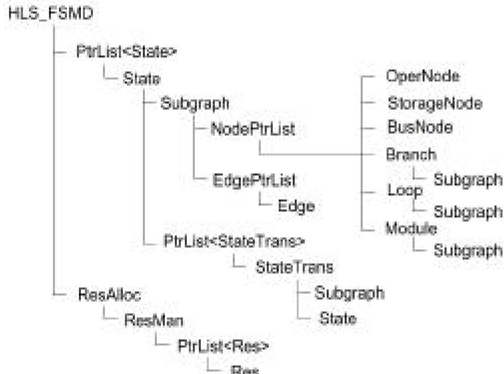

Figure 3: CDFG data structure

The detail of the CDFG data structure can be found in [DOGA01].

4.2 Approach one - Grouping after clique-partitioning

The procedure of the storage binding approach one is described by the figure 4. The algorithm related to register files is shown in algorithm 1.

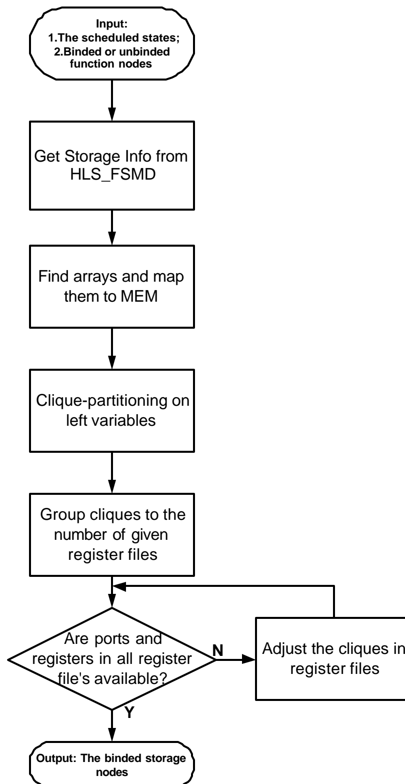

Figure 4: The procedure of storage binding approach 1

There are five major steps in this approach:

1. Get variables information which will be used in the following steps;
2. Bind arrays to memory;
3. Clique-partitioning the variables;
4. Group the cliques to the register files;
5. Adjustment;

4.2.1 Get variable information

First, we will get all variables information from class HLS_FSMD The procedure of getting variable information is explain in figure 5.

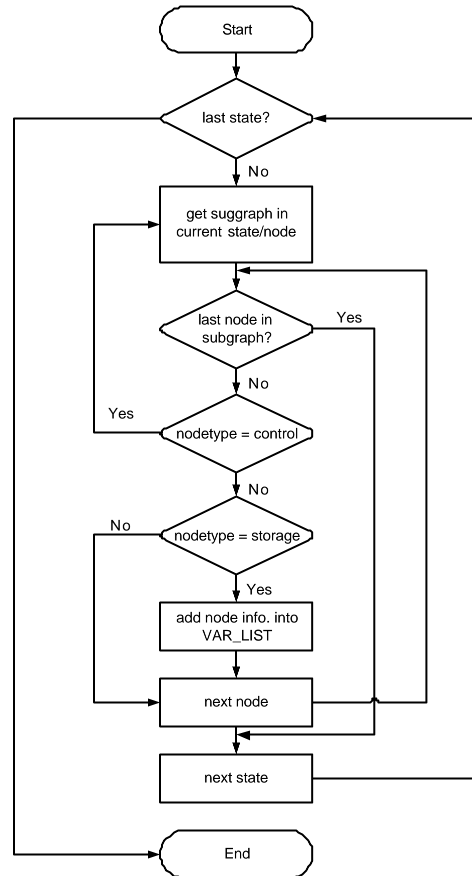

Figure 5: Get variable information

4.2.2 Bind arrays to memories

Since the structure of memories is very suitable for the arrays, we find the arrays in all variables and assign them to the given me mories.

4.2.3 Clique-partitioning

The following steps are for the left variables that will be assigned to register files.

Let $\boldsymbol{L}$ be the set of all variables needed to be binded:

$\boldsymbol{L}$={ $v_1\ v_2 ... v_m$ } m: the number of variables

The procedure of clique-partitioning algorithm [GAJS97] is described as figure 6.

Here we have to give the definition of the lifetime of a variable. It is defined as the set of states in which that variable is alive. The alive states includes the state following that state in which it is assigned a new value (**write state**), every state in which it is used on the right-hand side of a assignment statement (**read state**), and all states on each path between the write state and a read state.

```
for all v ∈ L do                                // Get lifetimes of variables
        Start(v);
        End(v);
endfor

C = CliquePartitioning(L);                      // Clique-Partitioning first for all variables

for all r ∈ C do                                // Get lifetimes of cliques (registers)
        Start(r);
        End (r);
endfor
SORT(C);                                        // sort the cliques (registers) in C in ascending order with
their
                                                // start  times, Start(r), as the primary key and end times,
                                                // End(r), as the secondary key
n = NumofRF(RF);
Set NumofRinRF(i) = number of registers in each RF_i; // i= 1,2..n
Set Availableport(i) = number of ports in each RF_i; // i= 1,2..n

C_1,C_2...C_n = Φ;
reg_index(i)=0;
while C ≠ Φ do                                  //Divide C into C_1,C_2..C_n, n= Number of Register Files
        for i=1; i++; i<=n do
                temp_reg = first r in C;
                ADD( C_i ,temp_reg);
                reg_index(i)++;
                C=DELETE(C, temp_reg);
                if C = Φ then
                        break;
                endif
        endfor
endwhile

// Check Register number and port number
While any NumofRinRF(i) ≤ reg_index(i) or Availableport(i) ≤port(i) do
        if NumofRinRF(i) ≤ reg_index(i) or Availableport(i) ≤port(i) then
                MOVE( C_i, C_j ,oneReginC_i);           // C_j has  the least number of cliques (registers)
        endif
endwhile
```

Algorithm 1: Approach 1 of Storage Binding Using Multiple Port Register Files

Then the compatibility graph will be generated. The compatibility graph consists of nodes and edges, in which each node represents a variable and each edge between two nodes represents compatibility (priority edge) or incompatibility (incompatibility edge) in merging variables represented by these two nodes.

The incompatibility edge indicates variables with overlapping lifetimes, while the priority edge indicates variables with non-overlapping lifetimes. Each priority has a weight *w* that can be represented as:

$$w = s + d \qquad (1)$$

s*: The number of different functional units that use both nodes as left or right operands;*
d*: The number of different functional units that generate results for both nodes.*

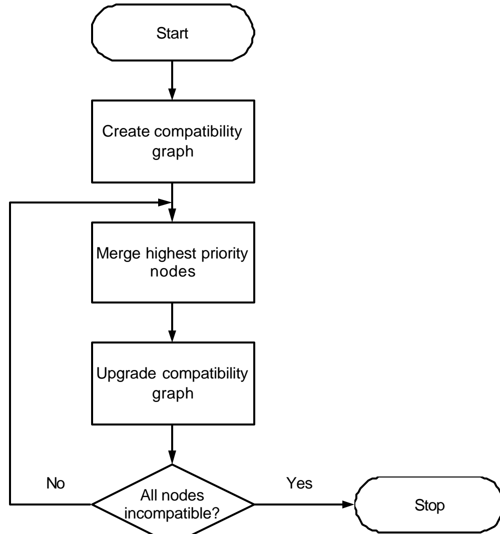

Figure 6: Clique-partitioning algorithm

Here we should consider two conditions when we calculate *w*: before functional units binding and after functional units binding. If before functional units binding, we can think the every operation as different functional units if they have different symbolic description, for examples, + and /. But if after functional units binding, the same symbolic operation descriptions may be mapped to different functional units, so they can also be thought as different functional units.

Using clique-partitioning algorithm, we can get the minimum number of required number of registers.

After graph partitioning, the variables are grouped into different cliques, which form the set ***C***. Every clique has its own lifetime that is equal the total of the lifetimes of the variables in the clique. The lifetime in the clique may be not continuous states. For examples, clique 1 have two variables whose lifetime are state 1~3 and state 5~7 respectively, then the lifetime of this clique is the state 1~3 and 5~7. The lifetime of cliques will be used in the following step.

4.2.4 Group cliques to register files

Register file includes several register and multiples in/out port. Since the number of register files is user-given and fixed, we should find a method to assign the cliques to the register files and make all register files do not have registers and ports conflicts.

Suppose the number of give register files is **n**.

Here we use sorted cliques lifetime to distribute the cliques. We have the following steps to distribute the cliques:

1. We sort the cliques (registers) in ***C*** in ascending order with their start times, *Start(r)*, as the primary key and end times, *End(r)*, as the secondary key;
2. We use **n** as the module, split ***C*** into small groups, $C_1, C_2 ... C_n$. The first clique in C is assign to C_1, the second one goes to C_2... then the n+1th clique to C_1 again. Repeat this process until all cliques in ***C*** are used.

$C_1, C_2 ... C_n$ will correspond to register file 1, register file2...register file n respectively.

4.2.5 Adjustment

We should consider not only number of registers in each register file, but also the ports of each register file.

The cliques number in each C_i maybe more than the number of registers in register files. The ports of register files also could have conflicts. When we assign the different cliques to the registers in register files, these cliques can have overlapping lifetimes. It will work fine if in every state, the number of lifetime overlapping cliques which

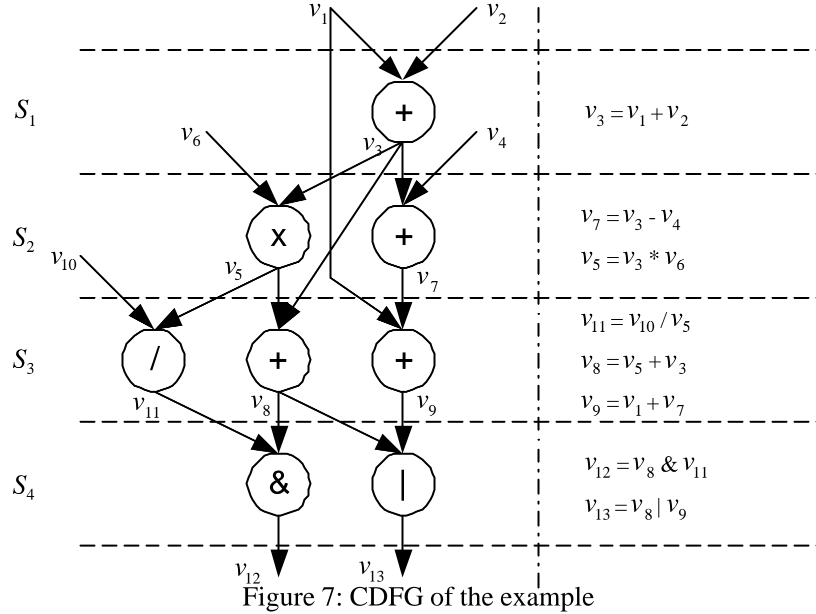

Figure 7: CDFG of the example

include this state do not exceed the number of ports in the target register files.

For examples, if there are three cliques whose life times are showed in table 1.

	S1	S2	S3	S4	S5
Clique 1	X	X	X		
Clique 2			X	X	
Clique 3			X	X	X

Table 1: Simple example of 3 cliques

In state 3, we can see all 3 cliques have overlapping lifetimes. So if given a register file having two ports, only two of these three can be assigned to this register file.

Besides lifetime, we also should consider in ports of the register file that is corresponding to the write state of the variables.

So in this step, we will make some adjustment of the cliques in each register file.

1. Find the C_i that have registers or ports conflicts;
2. Find the C_j which has the least number of register and least number of ports usage;
3. Move clique that cause conflict in C_i to C_j, check if any conflict in both C_i and C_j; If so, go to 1 again.

4.2.6 Example

Here we use an example to explain the approach 1. Figure 7 gives the CDFG of the example.

The lifetimes of each variable are given in figure 8. Here $\boldsymbol{L}$={ v_1 v_2 v_3 … v_{13} }.

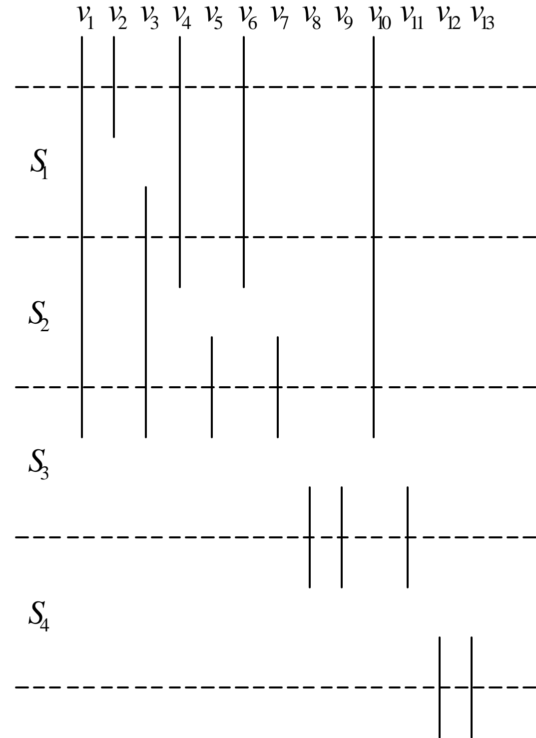

Figure 8: Lifetimes of variables

Then we do clique-partition and get the results shown in figure 9. We group all variables into 5 cliques. We let $\boldsymbol{C}$={ r_1 r_2 r_3 r_4 r_5 }.

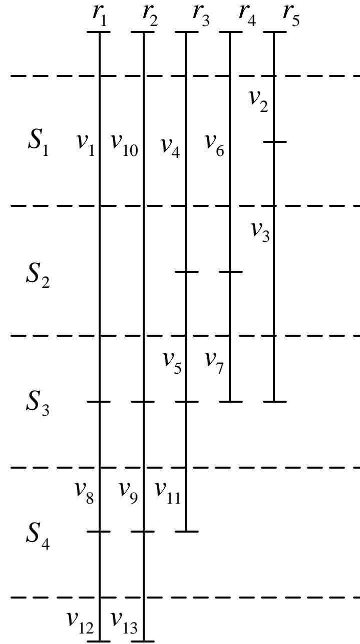

Figure 9: Clique-partitioning result

Three register files with one inport, two outports and four registers are give as storage units. So we should divide C into C_1, C_2, C_3 using the method in 4.2.4. We get

$$C_1 = \{ r_1 \; r_4 \},\; C_2 = \{ r_2 \; r_5 \} \; C_3 = \{ r_3 \}$$

Since there are no conflicts in all register files, the final binding result is shown in figure 10.

4.3 Approach two - Clique-partitioning after grouping

Besides approach one, we have another approach for the storage binding. The procedure is shown in figure 11. The algorithm related to register files is shown in algorithm 2.

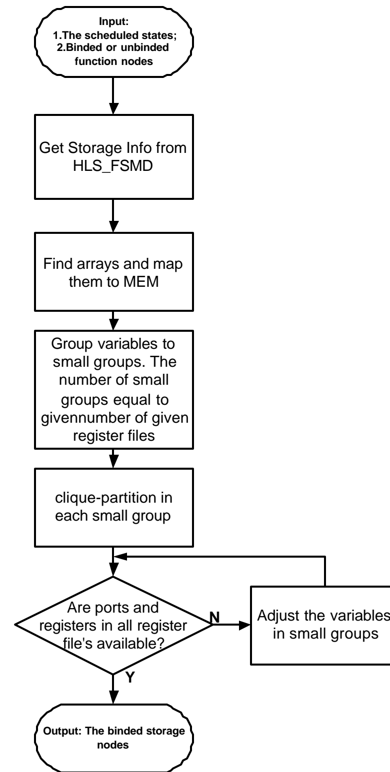

Figure 11: The procedure of storage binding approach 2

There are also five major steps in this approach:

1. Get variables information which will be used in the following steps;
2. Bind arrays to memory;
3. Split all variables into small groups. The Number of small groups equals to the number of given register files;
4. Clique-partitioning the variables in each groups;
5. Adjustment;

The first two steps are the same as approach one.

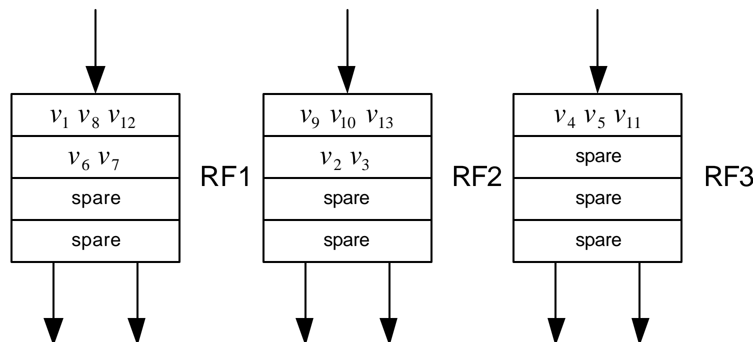

Figure 10: Binding results using approach 1

4.3.1 Split all variables into small groups

Instead of doing clique-partitioning first in approach one, we split variables first. We let ***L*** be the set of all variables needed to be binded:

L={ v_1 v_2 ... v_m } m: the number of variables

Suppose the number of give register files is **n**. Then we will split L into $L_1, L_2 ... L_n$.

We also use sorted variables life to distribute variables.

1. We sort the cliques (registers) in ***L*** in ascending order with their start times, *Start(v)*, as the primary key and end times, *End(v)*, as the secondary key;
2. We use **n** as the module, split ***C*** into small groups, $L_1, L_2 ... L_n$. The first clique in C is assign to L_1, the second one goes to L_2... then the n+1th clique to L_1 again. Repeat this process until all cliques in ***C*** are used.

$L_1, L_2 ... L_n$ will correspond to register file 1, register file2...register file n respectively.

Using this method, we can spread the variables by their lifetimes, which will be helpful in the following steps.

4.3.2 Clique-Partitioning in small groups

Then in each small group, we do clique-

```
for all v ∈ L do                              // Get lifetimes of variables
        Start(v);
        End (v);
endfor

SORT(L);                                      // sort the variables in L in ascending order with their start
                                              // times, Start(v), as the primary key and end times, End(v), as
                                              // the secondary key
n = NumofRF(RF);
Set NumofRinRF(i) = number of registers in each RF_i; // i= 1,2..n
L_1, L_2...L_n = Φ;
while L ≠ Φ do                                //Divide L into L_1, L_2...L_n, n= Number of Register Files
        for i=1; i++; i<=n do
                temp_var = first v in L;
                ADD( L_i,temp_var);
                L=DELETE(L, temp_var);
                if C = Φ then
                        break;
                endif
        endfor
endwhile

for i=1; i++; i≤n do                          // Clique Partitioning in every L_i
        Clique(i) = CliquePartitioning( L_i );// Get number of cliuqes in each L_i
endfor

While any NumofRinRF(i) ≤ Clique(i) do
        if NumofRinRF(i) ≤ Clique(i) then
                MOVE( L_i, L_j ,oneVarin L_i);        // L_j has the least number of cliques (registers)
                Clique(i) = CliquePartitioning( L_i );
                Clique(j) = CliquePartitioning( L_j );
        endif
endwhile
```

Algorithm 2: Approach 2 of Storage Binding Using Multiple Port Register Files

partitioning algorithm that is described in 4.2.3.

4.3.3 Adjustment

We also need some adjustments, which is similar as 4.2.5, in this approach to deal with registers and ports conflicts. The differences between them are that here we move variable other than clique and we need to do clique-partitioning again after variable movement.

4.3.4 Example

We also use the same example as approach one and given three register files with one inport, two outports and four registers.

First we get the lifetime of all variables ***L*** and sort them. The result is shown in figure 12.

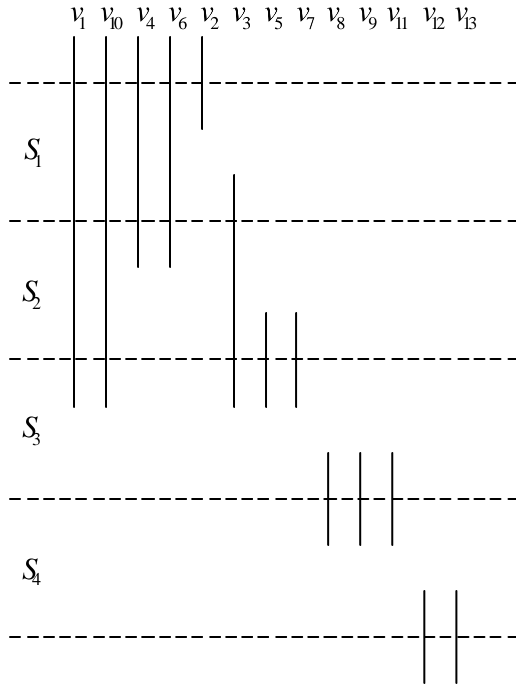

Figure 12: Result of sorted lifetime variables

Then using 4.3.1 method, we get:

$$L_1=\{ v_1\ v_4\ v_7\ v_{10}\ v_{13} \}$$

$$L_2=\{ v_2\ v_5\ v_8\ v_{11} \}$$

$$L_3=\{ v_3\ v_6\ v_9\ v_{12} \}$$

The results of clique-partitioning in each L_i are shown in figure 13.

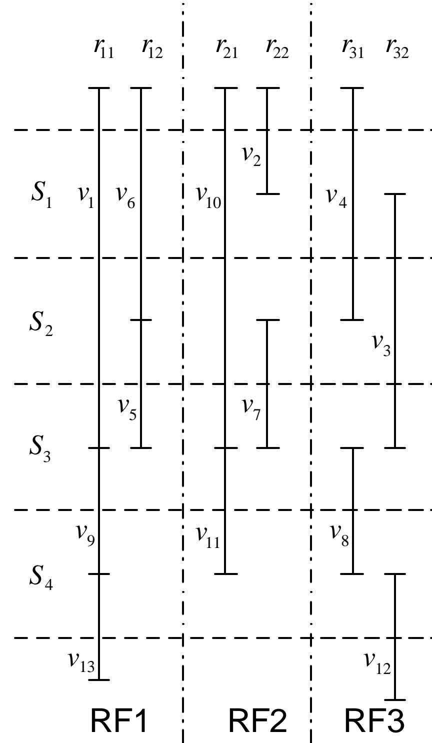

Figure 13:Results of clique-partitioning in L_i

Last, We get the binding result in figure 14

5 Experiment

To test our implementation, we use square-root approximation (SRA) as our example.

$$\sqrt{a^2+b^2} \approx \max((0.875x+0.5y),x) \qquad (2)$$

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

The procedure of SRA is described in figure 15 that has 8 states.

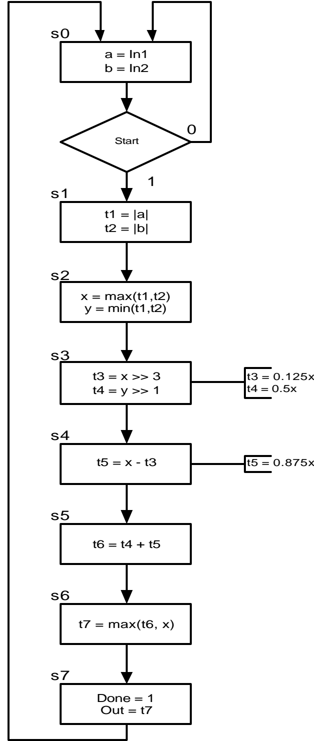

Figure 15: Square-root approximation

Table 2 and table 3 give the variables lifetime and priority weight respectively.

	S0	S1	S2	S3	S4	S5	S6	S7
a		X						
b		X						
t1			X					
t2			X					
x				X	X	X	X	
y				X				
t4					X	X		
t3					X			
t5						X		
t6							X	
t7								X

Table 2: Variables Lifetime Table

	a	**b**	**t1**	**t2**	**x**	**y**	**t4**	**t3**	**t5**	**t6**	**t7**
a	-										
b		-									
t1			-		1					1	
t2				-	1					1	
x			1	1	-						1
y						-					1
t4							-				
t3								-	1		
t5								1	-	1	
t6			1	1					1	-	
t7					1	1					-

Table 3: Priority weight Table

For the approach one, figure 16 give the clique-partitioning of the SRA.

After clique-partitioning, we got three cliques:
Clique 1(t4), lifetime:s4~s5
Clique 2(b,t1,x,t7), lifetime: s1~s7
Clique 3 (a,t2,t3,t5,t6,y), lifetime:s1~s6.

Then we sort the lifetimes of these cliques, we got a squence of *Clique 2, Clique 3, Clique 1*.

The given resources are two register files with two out ports and one in port. So we group *Clique 2* and *Clique 1* to register file 1, and *Clique 3* to register file 2 After checking the registers and ports conflicts, we found they are all feasible.

From another angle of views, we can see, all cliques' lifetimes include state s4 and s5. That is the s4 and s5 have overlapping 3 times. So, these three cliques can not assign to the same register file. Also, clique 1 and clique 3, clique 2 and clique 3 can not be assigned into the same register file since there write state are overlapped.

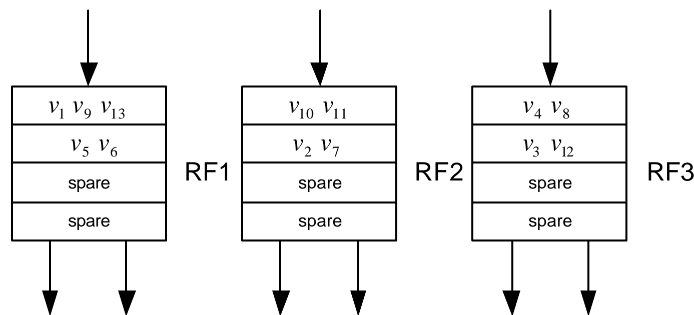

Figure 14: Binding results using approach 2

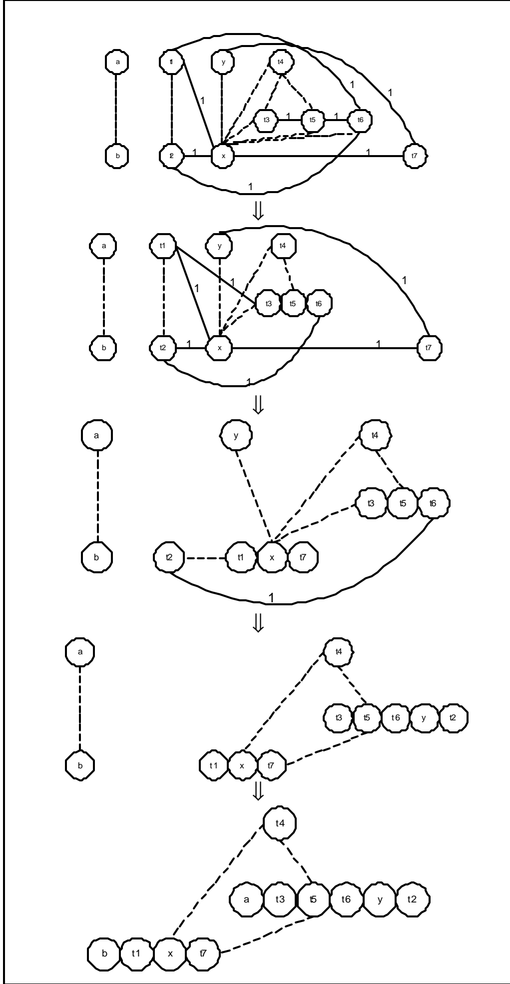

Figure 16: Process of clique-partitioning

After consider these issues, the final binding results is showed in figure 17.

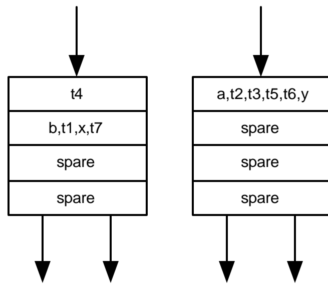

Figure 17: Experiment Results (Approach 1)

If we use approach 2, we can get the following results (figure 18):

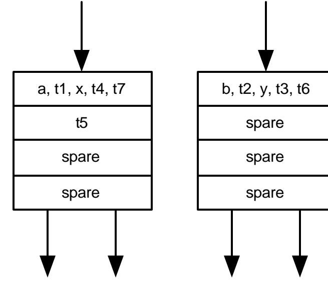

Figure 18: Experiment Results (Approach 2)

Table 4 and 5 give the comparison of different approaches.

	Total number of registers	Register usage in RF1	Register usage in RF2
Approach 1	3	50%	25%
Approach 2	3	50%	25%

Table 4: Comparison of different approaches (1)

	Inport usage in RF1	Outport usage in RF1	Inport usage in RF2	Outport usage in RF2
Approach 1	62.5%	43.75%	75%	37.5%
Approach 2	75%	43.75%	62.5%	37.5%

Table 5: Comparison of different approaches (2)

6 Summary

In this report, we use register files in the storage binding in the high-level (RTL) synthesis. In the implementation, the clique-partitioning algorithm and grouping method are used to get the minimum number of register and assign these registers in different register files. Different approaches have similar results. Results show that using our implementation, we can get decent results.

7 Future works

Here, we also give some directions on the future works. First, we can use different types of variables and register files to do storage binding, which is general in real designs. Second, we can consider the interconnection cost when do storage binding. Third, We should decide which ports of register file should be used for which register in each state (port binding). These works should be combined with scheduling and interconnection binding works.

References

[GAJS00] D. Gajski: *RTL Design and Methodology*, University of California, Irvine, Technical Report ICS-00-35, November 2000

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

[AHCH92] I. Ahmad, C. Y. R. Chen: *Grouping Variables into Multiport memories for ASIC Data Path Synthesis*, ASIC Conference and Exhibit, 1992.

[DOGA01] Dongwan Shin et al.: *CDFG Representation for SpecC RTL*, University of California, Irvine, Technical Report ICS-01-50, June, 2001

[GAJS97] D. Gajski: *Principles of Digital Design*, Prentice-Hall, Inc, 1997