Synthesis of False Loop Free Circuits *

Shih-Hsu Huang† Ta-Yung Liu† Yu-Chin Hsu† Yen-Jen Oyang‡

†Department of Computer Science
University of California
Riverside, CA 92521

‡Department of Computer Science
National Taiwan University
Taipei, Taiwan

Abstract

In behavior synthesis, an improper resource sharing may result in a circuit containing false loops which is non-simulatable or non-timing-analyzable. Previous approaches solve this problem during the datapath allocation phase. To build a false loop free circuit, they may have to allocate additional functional units other than those defined in the resource constraints. In this paper, we present an approach to solve the problem during the scheduling phase. Our scheduling algorithm finds a schedule which guarantees to have a false loop free circuit mapping under the given resource constraints. Experiments show the proposed approach finds false loop free schedule for most of the examples without introducing extra control steps.

1 Introduction

A false path [1] of a combinational circuit is defined as a path which will never be traversed under any combination of input values. A false loop [2] is a special case of false path, where the start point and the end point of the false path are the same. Because most logic synthesizer, timing simulator, and timing analyzers cannot handle false loops, it is therefore desirable to generate a circuit which contains no false loops.

The HIS system [3] is the first behavior synthesis system to tackle the false loop problem. It solves the problem during the allocation phase. Given a scheduled code, HIS uses a loop prevention algorithm [2] to generate a false loop free circuit. Their approach has a drawback that in many cases a false loop free circuit cannot be built under given resource constraints. Additional resources have to added in order to build a false-loop free circuit.

Different from the HIS system, we propose an approach to solve the problem during the scheduling phase. We use a "resource allocation graph" to represent a circuit configuration. A designer first specifies resource constraints such as the number and type of functional units to be used in the data path. The scheduling algorithm incrementally constructs an acyclic resource allocation graph which corresponds to a false loop free circuit mapping under the specified resource constraints. The objective is to build an acyclic resource allocation graph using minimal number of control steps.

The remainder of the paper is organized as follows. Section 2 defines a directed resource allocation graph to model the false loop problem. A false loop free scheduling algorithm, which builds an acyclic resource allocation graph, is proposed in Section 3. A refinement algorithm is presented in Section 4. Section 5 presents a multiple passes approach to find a false loop free schedule using minimal number of control steps. Section 6 reports the experimental results. Finally, concluding remarks are made in Section 7.

2 Resource Allocation Graph

Given a resource constraint, a scheduling algorithm may find a schedule that is impossible for an allocator to come up with a false loop free circuit. Fig. 1 (a) gives such an example. Suppose we are given a resource contraint of two adders (*add1* and *add2*) and one subtracter (*sub1*). Assume the clock cycle time is 100 ns, and the execution delay of each functional unit is 30 ns. By applying the list scheduling [4] algorithm to the program, we may find a schedule as shown in Fig. 1 (b).

Suppose during data path binding, we assign operations 2, 4, and 7 to adder *add1*, operations 3 and 5 to adder *add2*, and operations 1, 6 and 8

*This work has been supported in part by National Semiconductor Inc., UC MICRO program, Quickturn co, Fujitsu America Inc., and SMOS systems.

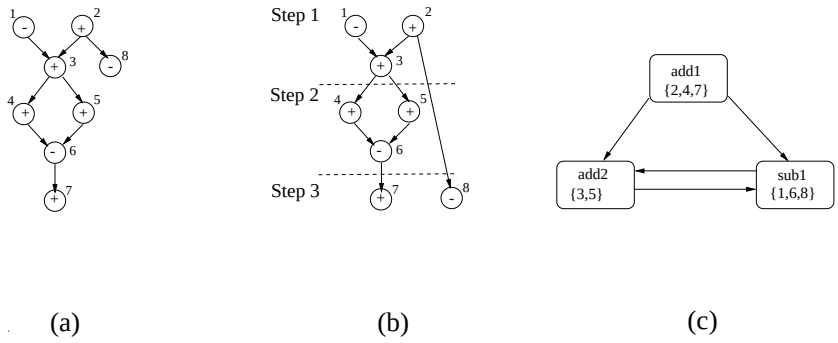

Figure 1: An Example. (a) The Data Flow Graph. (b) The Scheduled Code. (c) The Resource Allocation Graph.

to subtracter *sub1*. Because operation 3 uses the output of operation 1 and operation 6 uses the output of operation 5, there will be a connection between *add2* and *sub1* which in the real case will never be used. Thus there is a false loop between *add2* and *sub1* for this binding.

To detect false loops, we define a *resource allocation graph* to represent a circuit configuration at the higher abstraction level. Two operations o_i and o_j are said to be *chained* at control step s, if they are scheduled within control step s and there is a control and data dependency between operation o_i and operation o_j. Let $FU(o)$ denote the functional unit to which operation o is assigned. If operations o_i and o_j are chained in a control step, there must be a connecting path which passes the result of $FU(o_i)$ to the input of $FU(o_j)$ in the final hardware. We can model such a circuit configuration using a resource allocation graph at a higher abstraction level.

Definition 1 : *A resource allocation graph $G(V, E)$ is defined as a graph, where:*

- *each vertex in V represent a functional unit (FU) defined in the resource constraint; and*
- *a directed edge e connecting vertices $FU(o_i)$ to node $FU(o_j)$, if operations o_i and o_j are chained in a control step.*

The following theorem states that we can detect false loops on a resource allocation graph.

Theorem 1 : *There is a false loop in the final hardware, if and only if there is a cycle in the corresponding resource allocation graph.*
Proof: *Suppose there is a cycle in the resource allocation graph: $FU(r_1) \rightarrow FU(r_2) \rightarrow ... \rightarrow FU(r_n) \rightarrow FU(r_1)$. Since it is impossible to execute two different operations in functional unit $FU(r_1)$ within the same clock cycle, the loop will never be executed. Therefore, the loop in the final circuit is a false loop.* **Q.E.D.**

We can easily prove that *the scheduled code in Fig. 1 (b) contains false loops no matter what the binding is under the given resource constraints.* Since operation 6 uses the outputs of operations 4 and 5 at control step 2, we have directed edges $FU(4) \rightarrow FU(6)$ and $FU(5) \rightarrow FU(6)$ in the resource allocation graph. Note that $FU(6)$ is *sub1*. Since operations 4 and 5 are scheduled at the same control step, they must be assigned to different adders. Hence, we have directed edges $add1 \rightarrow sub1$, and $add2 \rightarrow sub1$ in the resource allocation graph. However, because operations 1 and 3 are chained at control step 1, a directed edge from $FU(1)$ (i.e., *sub1*) to $FU(3)$ (i.e., the adder executes operation 3) must be added to the resource allocation graph. Consequently, there is a cycle (i.e., false loop) between *sub1* and $FU(3)$. In other words, there is no binding for this schedule which is false loop free.

Thus, to prevent false loops in the final hardware, we should do it during the scheduling phase. We say a scheduled code is a *false loop free schedule* if there is a binding which is false loop free. The problem we study in this paper is: *given a behavior description (in a CDFG) and resource constraints, find a false loop free schedule using minimum number of control steps.*

3 False Loop Free Scheduling

In this section, we present a scheduling algorithm to find a false loop free schedule under the given resource constraints. A resource allocation graph is incrementally modified during

the scheduling process. Our objective is to find a schedule in which the corresponding resource allocation graph is acyclic.

3.1 The Algorithm

The core algorithm of the scheduling algorithm is based on the concept of list scheduling [4] [5]. In order to guarantee to have a false loop free circuit mapping, we perform module binding during the scheduling process. Initially, each functional unit in the resource constraint is given a label and the edge set of the resource allocation graph is empty. A directed edge which connects $FU(o_i)$ to $FU(o_j)$ is added, if operations o_i and o_j are chained at a control step. In order to ensure that false loops will never occur, we have to ensure there is no cycle in the resource allocation graph. To construct an acyclic resource allocation graph, the following rules are applied:

> *When an operation is scheduled into a control step, the module it is assigned to must not violate the topological ordering of the resource allocation graph. When two operations are chained into a control step, a new edge is added from its predecessors to represent the chaining between the operation and its predecessors.*

Let $AV(o, s)$ denote the set of available resources which can execute operation o at control step s, $P(o)$ denote the set of direct predecessors of operation o in the control/data flow graph, $I(s)$ denote the set of operations scheduled at control step s, and $Pred(r)$ denote the set of resources which precede resource r. The rules are elaborated in the following subsections.

3.1.1 Eliminating False Loops

If we don't consider the false loop condition, we can assign operation o to any resource in the set $AV(o, s)$. In order to avoid false loops, some bindings in $AV(o, s)$ must be prohibited. If operation o is scheduled into control step s, then it is chained with the operations in the set CP, where CP is $P(o) \cap I(s)$. Therefore, after we schedule operation o into control step s, for each operation o_i in the set CP a directed edge from $FU(o_i)$ to $FU(o)$ must be added to the resource allocation graph. Thus in order to avoid forming a cycle, the following condition, $FU(o) \notin \cup_{o_i \in CP} Pred(FU(o_i))$, must be true. In other words, we cannot assign operation o to any resource r in the set $\cup_{o_i \in CP} Pred(FU(o_i))$, even if $r \in AV(o, s)$.

3.1.2 Maintaining Resource Sharing

As mentioned earlier, in order to avoid creating false loops in the final hardware, we need to enforce an ordering to the resources during operation chaining. However, we want to maintain as much freedom as possible for the operations for future sharing. Because the resource allocation graph is acyclic, we can define a level $L(r)$ for each node r as follows:

1. if $Pred(r) = \emptyset$, $L(r) = 1$;
2. otherwise, $L(r) = MAX_{r_i \rightarrow r} L(r_i) \ + 1$.

In order to fully utilize the resources at each control step, the minimal level functional unit (in possible solutions) is chosen. This property maintains as much freedom as possible for the operation chaining and hence achieves the maximal resource sharing.

3.2 An Example

Let's apply the false loop free scheduling algorithm to the example given in Section 2. Assume we are given two adders (*add1* and *add2*) and one subtracter (*sub1*) as the resource constraint. Assume the clock cycle time is 100 ns, and the execution delay of each functional unit is 30 ns.

Initially, the set E of edges in resource allocation graph is empty. Thus the level of each node is 1. The scheduling starts with control step 1. Operations 1, 2, and 3 are assigned to *sub1*, *add1*, and *add2*, respectively. Operation 3 is chained with operations 1 and 2 at this control step. Therefore, we need to add directed edges $FU(1) \rightarrow FU(3)$ and $FU(2) \rightarrow FU(3)$ to the resource allocation graph. As a result, we have the ordering that *sub1* precedes *add2* and *add1* precedes *add2*.

Next, we move to control step 2. Operations 4 and 5 are assigned to *add1* and *add2*, respectively. Then, we move to operation 6. We have that $AV(6, 2)$ is *sub1*. Note that $P(6)$ is $\{4, 5\}$ and $I(2)$ is $\{4, 5\}$. It means operations 4 and 5 are the predecessors of operation 6 for chaining at this control step. Therefore, if we assign operation 6 to *sub1*, the edges $FU(4) \rightarrow sub1$ and $FU(5) \rightarrow sub1$ must be added into the resource allocation graph. However, since *sub1* precedes $FU(5)$ (i.e., *add2*), *sub1* cannot use the output of *add2*. Otherwise, a cycle between *add2* and *sub1* will be formed. Hence, we cannot assign operation 6 to the functional unit *sub1*. There are no binding available for operation 6. Therefore, we cannot schedule operation 6 in this control step, even if *sub1* has not been sealed. The next

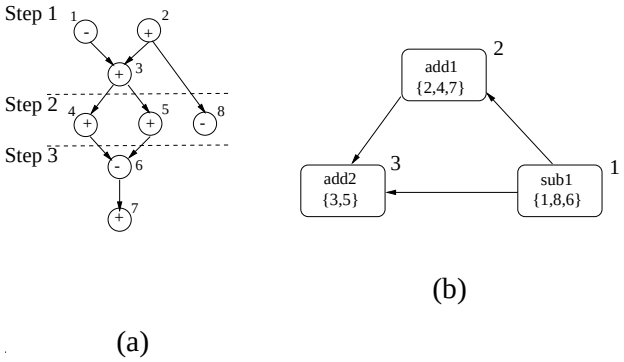

Figure 2: After False Loop Free Scheduling is Finished. (a) The Scheduled Code. (b) The Resource Allocation Graph.

operation to be scheduled in the ready queue is operation 8. Since it has no dependency predecessor scheduled at this control step, we can assign Operation 8 into the current control step.

Next, we move to control step 3. Operation 6 is chosen from the ready queue, Since $AV(6,3)$ is *sub1*, we assign operation 6 to *sub1*. Finally, operation 7 is chosen from the ready queue, The set $AV(7,3)$ is $\{add1, add2\}$. Note that neither *add1* nor *add2* precedes *sub1* in the resource allocation graph. Therefore, the set of possible solutions Sol is the same as $AV(7,3)$. Because $L(add1)$ is less than $L(add2)$, we assign operation 7 to *add1*; i.e. $FU(7)$ is *add1*. A new directed edge $FU(6) \rightarrow FU(7)$ is added to represent the chaining of operations 6 and 7.

After we complete the task of false loop free scheduling, we obtain the scheduled data flow graph as shown in Fig. 2 (a). Different from the scheduled code shown in Fig. 1 (b), operations 6 and 8 are respectively scheduled at control step 3 and 2 instead. A topological ordering of resources has been defined after the scheduling is finished. The ordering is $sub1 \rightarrow add1 \rightarrow add2$.

4 Resource Allocation Graph Refinement

The objective of resource allocation graph refinement is to improve the binding of module assignment during the scheduling phase. Let $o_i \stackrel{s}{\Rightarrow} o_j$ denote that operations o_i and o_j are chained at control step s. We define a *chaining path* c as follows: $c : o_1 \stackrel{s}{\Rightarrow} o_2 \stackrel{s}{\Rightarrow} ..... \stackrel{s}{\Rightarrow} o_n$, where o_1 has no predecessor and o_n has no successor within control step s. During the scheduling, a path $P1$: $FU(o_1) \rightarrow FU(o_2) \rightarrow ..... \rightarrow FU(o_n)$ is formed in the resource allocation graph to represent the chaining path c. Assume there is another path $P2 : r_1 \rightarrow r_2 \rightarrow ..... \rightarrow r_n$ in the resource allocation graph. Let $o(r_i, s)$ denote the operation to be executed on functional unit r_i at control step s. We can relocate the chaining path c from path $P1$ to path $P2$ by interchanging operations o_i and $o(r_i, s)$ for $1 \leq i \leq n$, if the following conditions are satisfied:

1. Functional unit r_i can execute operation o_i, for $1 \leq i \leq n$;

2. Functional unit $FU(o_i)$ can execute operation $o(r_i, s)$, for $1 \leq i \leq n$; and

3. After the interchange, no new edge is formed in the resource allocation graph.

Because the resource allocation graph obtained by the scheduling algorithm is acyclic, we can impose a topological ordering to the edges. We try to remove the edges according to the sequence of their topological ordering. Our goal is to remove as many edges as possible such that the number of edges is minimized. Let $C(e)$ denote the set of chaining paths which are realized by flowing through edge e. An edge e is removable if all the chaining paths in $C(e)$ can be relocated to other paths without flowing through e.

5 Multiple Passes Scheduling

As mentioned earlier, our goal is to find a false loop free schedule using a minimal number of control steps. Since the scheduling algorithm imposes limitations on the resource sharing, it may introduce extra control steps. In some cases, the extra control steps are unavoidable. However, in many cases the extra control steps can be avoided by reordering the execution of operations. In order to minimize the number of control steps, our idea is to make several passes of scheduling to compact the false loop free schedule.

The multiple passes scheduling algorithm is outlined in Fig. 3, where $CDFG$ is the control/data flow graph and $init\_RAG$ is the initial resource allocation graph. The set of edges in $init\_RAG$ is empty. The algorithm goes through

```
Procedure Multiple_Passes_Scheduling
Begin
 direction=top_down;
 (S_CDFG,S_RAG)= False_Loop_Free_Schedulng(
  CDFG,init_RAG,direction);
 S_opt_RAG=RAG_Refinement(S_RAG);
 loop
  (S'_CDFG,S'_RAG)= False_Loop_Free_Scheduling(
   CDFG,S_opt_RAG,inverse(direction));
  S'_opt_RAG=RAG_Refinement(S'_RAG);
  if (|S_CDFG| = |S'_CDFG|)
   then
    if (cost(S_opt_RAG) ≤ cost(S'_opt_RAG))
     then return (S_CDFG);
     else return (S'_CDFG);
   else
    S_CDFG=S'_CDFG;
    S_opt_RAG=S'_opt_RAG;
 forever;
End;
```

Figure 3: Multiple Passes Scheduling.

several passes of scheduling in top-down manner and bottom-up manner alternatively. The first pass performs false loop free scheduling in top-down manner and produces a scheduled code S_{CDFG}. During each pass in the loop iteration, S'_{CDFG} and S_{CDFG} represent the scheduled codes produced in this iteration and the previous iteration, respectively. The loop iteration terminates if the number of control steps does not decrease; i.e., $|S_{CDFG}| = |S'_{CDFG}|$. Upon termination of the algorithm, we select one from the two scheduled codes S_{CDFG} and S'_{CDFG} as our solution.

At the beginning of each pass, it invokes the subroutine $False\_Loop\_Free\_Scheduling$ to produce a false loop free schedule and an acyclic resource allocation graph. Next, the subroutine $RAG\_Refinement$ improves the resource allocation graph and then returns the refined one. The goal of each pass is to find a more compact false loop free schedule; i.e., to find a scheduled code S'_{CDFG} whose number of control steps is less than (at most, equal to) the number of control steps of S_{CDFG}. The goal can be achieved by using the scheduling strategy similar to [5].

6 Experiments

We have implemented the proposed scheduling algorithm in C running on a Sun Sparc workstation and integrated it into a behavior synthesis system [6]. A wide range of benchmarks from the open literature, including *Diffeq* [7], *Ellip* [7], *LPC* [8], a segment of *FFT* [9], *Filter* [10], *QRS* [11], and *Knapsack* [12], are used to test the effectiveness of the new approach. The experiment shows that the proposed approach synthesizes false loop free circuits without introducing extra control steps.

We compare the results produced by scheduling algorithms using and without using false loop avoidance. Table 1 tabulates the results with constraints on the number of adders (*#add*), the number of subtracters (*#sub*), the number of ALUs (*#alu*), and the number of multipliers (*#mul*). The column *Without_Avoidance* gives the scheduling result without false loop avoidance, including the number of control steps ($\#Steps_{ls}$), and if it contains a false loop which cannot be eliminated under the resource constraints or not (*floop*). The column *With_Avoidance* gives the scheduling result with false loop avoidance, including the number of control steps obtained by false loop free scheduling ($\#Steps_{flfs}$), the number of control steps obtained by multiple passes scheduling ($\#Steps_{mps}$), and the number of passes to perform multiple passes scheduling (*#passes*). The example *Ex* is the example given in Fig. 1 (a).

Experiments show that examples *Ellip*, *LPC*, *FFT*, *Filter* and *Ex* contain false loops if false loop avoidance is not used in the algorithm. However, using false loop avoidance heuristic in the algorithm, a false loop free schedule can be found using the same number of states. We were able to find false loop free schedules using minimum number of states in one pass except *LPC* and *Knapsack*. In examples *LPC* and *Knapsack*, the second pass in multiple-pass scheduling will reduce the number of control steps by one.

7 Conclusions

In this paper, we presented an approach to generate a schedule which guarantees to have a false loop free circuit mapping under the given resource constraints. A directed resource allocation graph is defined to model the circuit configuration, such as operation chaining and resource sharing, at the higher abstraction. The goal of the scheduling algorithm is to construct an acyclic resource allocation graph which corresponds to a false loop free circuit mapping, while using minimum number of states. Benchmark data shows that most false loop free circuits can be built without introducing extra control steps.

Design	Constraints				Without_Avoidance		With_Avoidance		
	#add	#sub	#alu	#mul	$\#Steps_{ls}$	floop	$\#Steps_{flfs}$	$\#Steps_{mps}$	#passes
Diffeq	0	0	2	2	3	no	3	3	2
Ellip	**3**	**0**	**0**	**1**	**8**	**yes**	**8**	**8**	**2**
	4	0	0	2	7	no	7	7	2
	5	**0**	**0**	**2**	**6**	**yes**	**6**	**6**	**2**
LPC	**0**	**0**	**2**	**2**	**11**	**yes**	**12**	**11**	**3**
FFT	**0**	**0**	**2**	**2**	**5**	**yes**	**5**	**5**	**2**
Filter	**4**	**0**	**0**	**1**	**9**	**yes**	**9**	**9**	**2**
	4	**0**	**0**	**2**	**7**	**yes**	**7**	**7**	**2**
	5	**0**	**0**	**2**	**6**	**yes**	**6**	**6**	**2**
QRS	1	1	0	0	37	no	37	37	2
	2	2	0	0	28	no	28	28	2
Knapsack	**0**	**0**	**2**	**1**	**18**	**yes**	**19**	**18**	**3**
Ex	**2**	**1**	**0**	**0**	**3**	**yes**	**3**	**3**	**2**

Table 1: Scheduling Results.

References

[1] D. Brand and V. Iyengar, "Timing Analysis using Functional Analysis". *IEEE Trans. on Computers*, pages 1309–1314, October 1988.

[2] L. Stok, "False Loops through Resource Sharing". In *Proc. of International Conference of Computer-Aided Design*, pages 345–348, November 1992.

[3] R. Camposano, R. Bergamaschi, C. Haynes, M. Payer, and S. Wu, "The IBM High-Level Synthesis System". *Kluwer Academic Publishers*, pages 79–104, 1991.

[4] S. Davidson, D. Landskov, B.D. Shriver, and P.W. Mallett, "Some Experiments in Local Microcode Compaction for Horizontal Machines". *IEEE Trans. on Computers*, pages 460–477, July 1981.

[5] S.H. Huang, C.T. Hwang, Y.C. Hsu, and Y.J. Oyang, "A New Approach to Schedule Operations across Nested-ifs and Nested-loops". *The EuroMicro Journal: Microprocessing and Microprogramming*, pages 37–52, April 1995.

[6] Y.C. Hsu, "MEBS User Guide: A Multiple-Entry Behavior Synthesis System for Digital System Rapid Prototyping". *Technical Report, Department of Computer Science, University of California, Riverside,* June 1994.

[7] R. Vemuri, P. Mamtora, and J. Roy, "Benchmarks for High-Level Synthesis". *TM-ECE-DDE-91-11,* Lab. for Digital Design Environments, Electrical and Computer Engineering, University of Cincinnati, June 1991.

[8] M.M. Jamali, P. Bumrungthum and N. Mohankrishnan, "A Parallel Algorithm for Linear Predictive Coding". *Signal Processing IV: Theories and Applications*, pages 759–762, 1988.

[9] R.C. Gonzalez and P. Wintz, "Digital Image Processing". *Addison Wesley*, 1987.

[10] S.Y. Kung, H.J. Whitehouse, and T. Kailath, "VLSI and Modern Signal Processing". *Englewood Cliffs, NJ: Prentice Hall*, pages 258–264, 1985.

[11] S.C. Roy, H.T. Nagle, M.G. McNamer, and W.T. Krakow, "QRS/BIST: A Reliable Cardiac Arrhythmia Monitor ASIC". In *Proc. of the 3rd Annual IEEE ASIC Seminar and Exhibit*, September 1990.

[12] E. Horowitz and S. Sahni, "Fundamentals of Computer Algorithms". *Computer Science Press*, page 355, 1978.