Scheduling Verification in High-Level Synthesis - Implementation of a Normalizer and a Code Motion Verifier

SYNOPSIS OF
DUAL DEGREE – M. TECH PROJECT

" YOGA KARMASU
KAUSALAM "

Under the guidance of

Prof. C. R. Mandal
Department of Computer Science and Engineering,
Indian Institute of Technology Kharagpur
And
Prof. Dipankar Sarkar
Department of Computer Science and Engineering,
Indian Institute of Technology Kharagpur

Submitted by:
Pramod Kumar (03CS3019)

Introduction

High level synthesis is the process of generating the register transfer level (RTL) design from the behavioral description. The synthesis process consists of several interdependent phases:

1. Preprocessing: Translation of the input control data flow graph (CDFG) to an intermediate representation (IR) and calculation of necessary information for scheduling.
2. Schedule of the operations and the transfer of variables in minimum number of control steps for a given architectural specification. The scheduler accomplishes functional unit formation.
3. Allocation and binding of variables to registers.
4. Data path generation from the schedule of operations, bus transfers and the variable mapping to the registers.
5. Generation of synthesizable Verilog code (RTL).

A High-level synthesis tool, called Structured Architecture Synthesis tool (SAST), has been developed which support hand-in-hand synthesis and verification. The existing framework is the SAST, and this work is to enhance this tool by incorporating a Normalizer and a Code Motion Verifier

The complexity of present-day VLSI systems is very high. The specification is given at a high level of abstraction compared to that of the output. In addition several optimization and transformations may be made at each phase to improve the performance of the design. Hence it is important to ensure that after each phase the behavior of the original specification is preserved. Hence is the need of phase-wise verification.

Verification of high level synthesis is a formal method for checking the equivalence between two descriptions of the target system, one before a particular phase and the other after that phase. The descriptions are represented as Finite State Machines with Data paths (FSMD).The basic principle is to show that any computation of one FSMD is covered by a computation on the other.

While finding the equivalent path for a path, it is required to check the equivalence of the respective conditions as well as the data transformations of the paths. Since the condition of execution and the data transformation of a path involve the whole of integer arithmetic, checking equivalence of paths reduces to the validity problem of first order logic which is undecidable; thus, a canonical form does not exist for integer arithmetic. There we use a normalized form for conditional expression and data transformation expression.

It may be possible to transform the input behavior to some equivalent description, by incorporating several high-level code transformation techniques, which results in amore efficient scheduling behavior. Thus the need to enhance the verifier to handle various code motion techniques while verification.

[1] NORMALIZATION :

While finding the equivalent path for a path, it is required to check the equivalence of the respective conditions as well as the data transformations of the paths. Since the condition of execution and the data transformation of a path involve the whole of integer arithmetic, checking equivalence of paths reduces to the validity problem of first order logic which is undecidable; thus, a canonical form does not exist for integer arithmetic.

The normalization process reduces many computationally equivalent formulas syntactically identical as it forces all the formulas to follow a uniform structure. In the following, the normal form chosen for the formulas and the simplification carried out on the normal form during the normalization phase are briefly described.

A condition of execution (formula) of a path is a conjunction of relational and Boolean literals. A Boolean literal is a Boolean variable or its negation. A relational literal is an arithmetic relation of the form s r 0, where s is a normalized sum and r belongs to { <=, >=, ==, != }. The relation > (<) can be reduced to >= (<=) over integers. For example, x – y > 0 can be reduced to x - (y – 1) >= 0.

The data transformation of a path is an ordered tuple < ei > of algebraic expressions such that the expression ei represents the value of the variable vi after execution of the path in terms of the initial data state. So, each arithmetic expression in data transformation can be represented in the Normalized Sum form. A normalized sum is a sum of terms with at least one constant term; each term is a product of primaries with a non-zero constant primary; each primary is a storage variable, an input variable or of the form abs(s), mod(s1, s2), exp(s1, s2) or div(s1, s2), where s, s1, s2 are Normalized Sums. These syntactic entities are defined by means of production of the following grammar.

Grammar Of The Normalized Sum :

1. $S \rightarrow S + T \mid c_s$, *where* c_s *is any integer.*

2. $T \rightarrow T * P \mid c_t$, *where* c_t *is any integer.*

3. $P \rightarrow S \uparrow C_e \mid abs\,(S) \mid (S)\ mod\ (S) \mid S \div C_d \mid c_m$, *where* c_m *is a symbolic constant.*

4. $C_e \rightarrow S \uparrow C_e \mid S$

5. $C_d \rightarrow S \div C_d \mid S.$

Various simplifications that can be carried out at the normalization phase are as follows:

Simplification at the arithmetic expression (normalized sum) level:

- Any expression involving only integer constants is immediately evaluated, e.g., (5 / 2) is evaluated to 2.
- In an expression, common sub expressions are collected together. Ex.

$x^2 + 3x + 5z + 4x$ is reduced to $x^2 + 7x + 5z$.

Simplifications at the relational expression (relational literal) level:

- Any relational expression built from constant arithmetic expressions may be immediately evaluated to TRUE or FALSE.

  For example, 4 – 1 >= 0 is evaluated to TRUE.

  $3x^2 + 9xy + 6z + 7 \geq 0$ is

- Common constant factors are extracted from the normalized sum and the relational expression is consequently simplified. For example,

$$3x^2 + 9xy + 6z + 7 \geq 0 \text{ is}$$

is mapped to

$$x^2 + 3xy + 2z + 2 \geq 0, \text{ where } \lfloor 7 \div 3 \rfloor = 2.$$

Simplification at the formula level:

Some literals of the formula can be deleted by the rule " if (A --> B) then (A&&B is equivalent to A) ". For this step of simplification, it becomes necessary to detect implication among literals. It is possible to detect whether a relational literal implies another relational literal when they involve the same non-constant sums. Let the literals be l1: (s1 + c1) R1 0 and l2 : (s2 + c2) R2 0. If s1 == s2 == s, then table depicts the relationship between the constants c1 and c2 depending upon R1 and R2, which must be satisfied for l1 to imply l2. Removal of repetitions of literals in a formula is possible using this rule as for any literal l1, l1 → l2 is always TRUE. For example, the literal (A >= B) has multiple occurrences in the formula (A >= B && C <= D && A >= B). So, this formula is simplified to (A >= B && C <= D).

		$R_2 \rightarrow$			
		$=$	$\geq$	$\neq$	$\leq$
$R_1 \downarrow$	$=$	$c_1 = c_2$	$c_2 \geq c_1$	$c_1 \neq c_2$	$c_2 \leq c_1$
	$\geq$		$c_2 \geq c_1$	$c_2 > c_1$	
	$\neq$			$c_1 = c_2$	
	$\leq$			$c_2 < c_1$	$c_2 \leq c_1$

Table: Conditions on c1 and c2 for which (s1 + c1) R1 0 implies (s2 + c2) R2 0

[2] ENHANCING THE VERIFIER TO HANDLE CODE MOTION TECHNIQUES :

1) **Reverse Speculation**: In reverse speculation the operations before a conditional Block are moved into the blocks subsequent to the conditional block In a special case of Reverse Speculation the scheduler may move an operation, say 'O', before the conditional block into only one of the conditional branch.This is possible when the operations in the other branch as well as all the operations following the merging of the conditional branches are not dependent on the result of the operation 'O'.

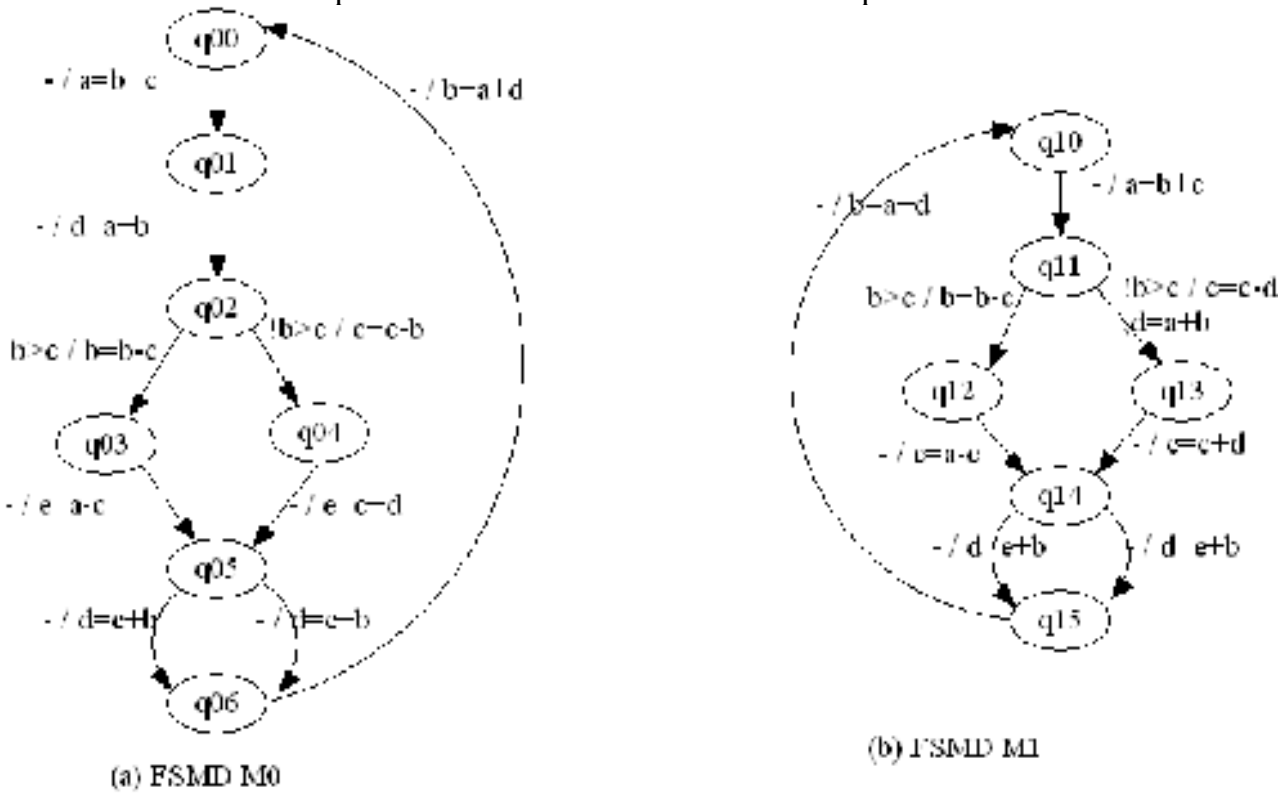


figure 1.1: Reverse Speculation

2) **Early Condition Execution:** This transformation involves restructuring the original code so as to execute the conditional operations as soon as possible. This, in effect, means that the conditional operation is moved-up in the design, and hence, all the operations before the conditional operation are reverse speculated into the conditional branches.

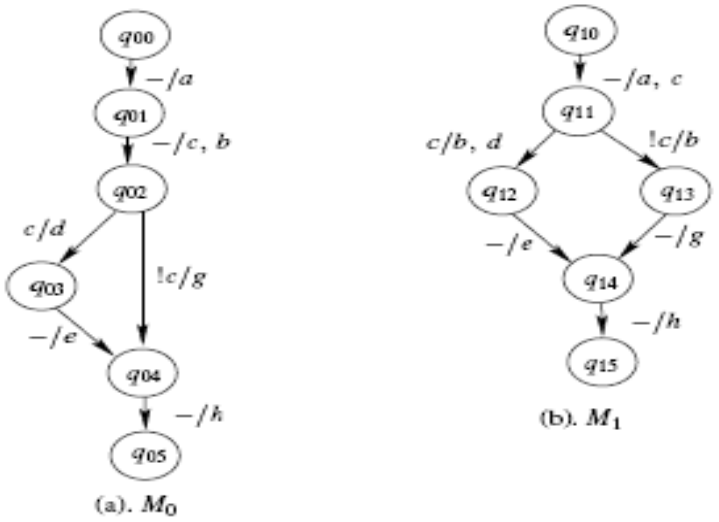


figure 2.1: Early Condition Execution

In the above figure, the conditional statement operation ‘c’ is executed one step early in the scheduled behavior and the operation ‘b’ is reverse speculated in the conditional branches. This is also a kind of Reverse Speculation and can be handled.

3) **Speculation:** Speculation refers to the unconditional execution of instructions that were originally supposed to be executed conditionally. In this approach, the result of a Speculated operation is stored in a new variable. If the condition under which the operation was to execute evaluates to true, then the stored result is copied to the variable from the original operation, else the stored result is discarded.

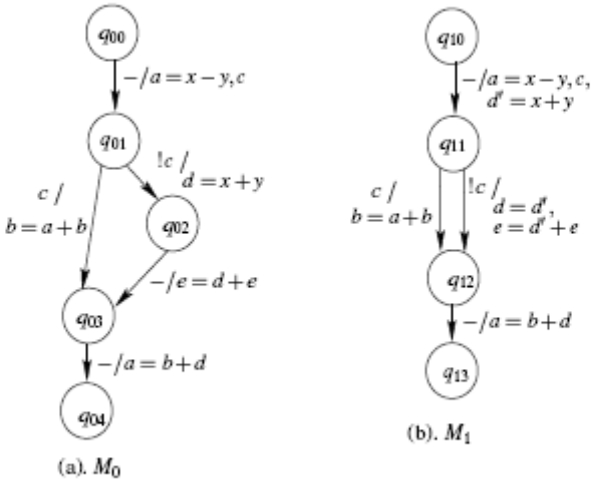


figure 3.1: Speculation

In the above figure, the operation d=x+y is speculated out of the branch with condition '!c' of the FSMD M0 and the result of the operation is stored in d'. It may be noted that if we do not store the value in d', then the variable 'a' gets the wrong value (by the operation a=b+d) when the execution is through the branch with condition 'c' of the FSMD M1.

4) **Loop Shifting and Compaction:** Loop Shifting is a technique whereby an operation 'op' is moved from the beginning of the loop body to the end of the loop body. To preserve the correctness of the program, a copy of the operation 'op' is also placed before the start of the loop. Shifting an operation results in execution of the operation one more time than in the original code. This situation is similar to the speculation and can be handled.

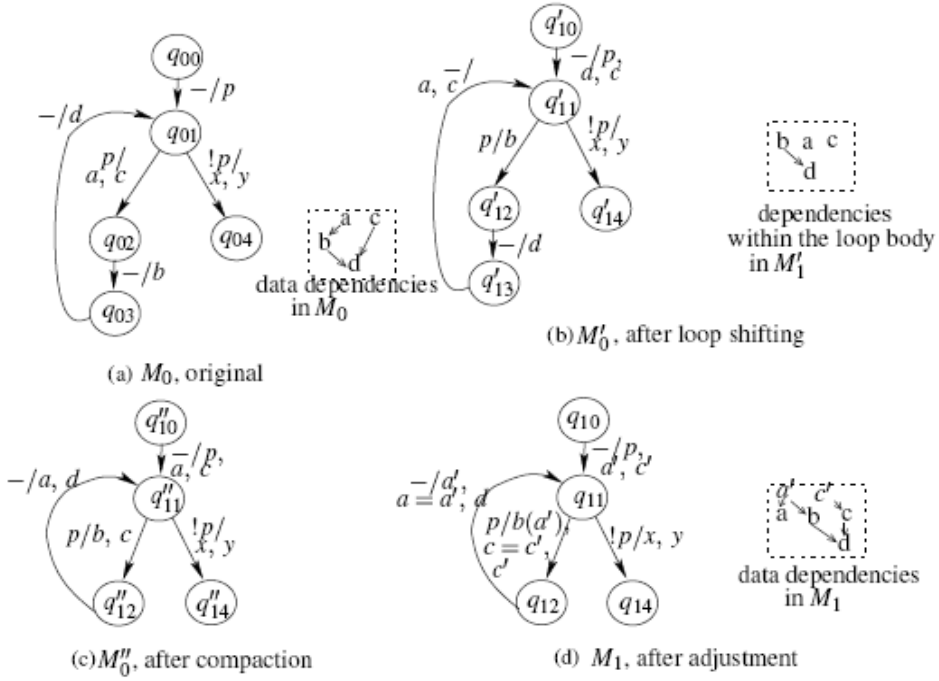


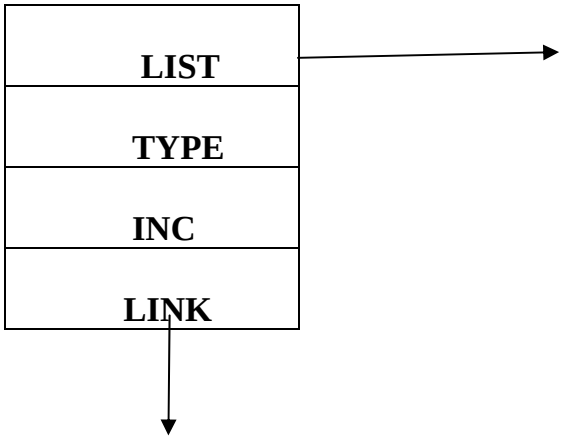
figure 4.1: Loop Shifting and Compaction

Work Done:

1. Implementation of the Normalizer:

1.1 Implementation of Structure for Normalized Form (A Normalized Cell) :

```
struct normalized_cell
{
   NC *list;
   char type;
   int inc;
   NC *link;
};
```



Normalized Expression: A Normalized Condition is represented as [(s + c) R 0], where 's' is a Normalized Sum, 'c' is an integer constant, 'R' is a Relational operator.Example: Normalized Condition: [(3 + 1 * x – 2 * y) >= 0].

1.2. **Implementation of all the normalization routines for conditional expressions and the transformation expressions.**

2. Implementation of the Code Motion Verifier:

2.1 Handling Reverse Speculation and Early Condition Execution:

Let ‘β’ be a path in M0 of the form < q0i => q0j > and < q0i, q1k > be a corresponding state pair. When the existing algorithm fails to find the equivalent path for ‘β’, then let there exist a path starting from q1k in M1, say ‘α’, whose condition of execution matches with that of ‘β’ but the data transformations does not match. In such case, we will check whether there is any variable of ‘V0 ∩ V1’ (V0 contains the list of variables present in FSMD M), and V1 contains the list of variables present in FSMD M1) which is modified along ‘β’ but not modified in the path ‘α’. Let ‘v’ be such a variable. Without any loss of generality, let the values of all the variables in ‘V0 ∩ V1’ other than ‘v’ at the end of execution of ‘α’ be the same as those for ‘β’. Now, if we can show that the transformed value of ‘v’ in ‘β’ is not used in any execution path starting from state ‘q0j’, then ‘α’ is equivalent to ‘β’.

We convert the FSMD M0 into an equivalent Kripke structure by some logical transformations. A dummy state will be added for every transition of M0. There would be two propositions, ‘D_v’ (defined ‘v’) and ‘U_v’ (used ‘v’), for each variable in ‘V0 ∩ V1’. The proposition ‘D_v’ will be true in a dummy state if the variable ‘v’ is defined by some operation in the corresponding transition in the M0. Similarly, ‘U_v’ will be true in a dummy state if the variable ‘v’ is used in some operation in the corresponding transition in M0. by convention if any proposition is not present in any state of the Kripke structure, then the negation of the proposition is true in that state. The required property that there does not exist any path in which ‘v’ has not been defined before it is used can be written as the CTL formula “ ! E [(! D_v) W U_v] “, were E represents there exists and W represents weak until operator. This formula can be verified using CTL model checker ex. NuSMV. If this formula is true in the state ‘q0j’, then ‘β’ is equivalent to ‘α’.

2.2 Handling Speculation:

While finding the equivalent of a path, say ‘β’ of M0 in M1, paths starting from the corresponding state of the start node of ‘β’ are considered one by one. Let path ‘β’ be of the form < q0i => q0j > and < q0i, q1k > be the corresponding state pair. So, the paths starting from q1k will be checked one by one until an equivalent path is found, failing which it is concluded that no equivalent path exists for that path. Let ‘α’ be a path which starts from q1k. If it is found that some variables not belonging to ‘V0 ∩ V1’ ( ) are used before they are defined along path ‘α’ during the computation of R_α and r_α, then we will find the set of paths from P1’ of FSMD M1 which terminates in state ‘q1k’. Next the last operations defining these variables in those paths will be found out by backward breadth first

search from q1k and the right hand side expressions of those operations will be used as the initial symbolic values of these variables.

2.3. **Handling loop Shifting and compaction:**
In order to do so, it is required to perform the following steps. Let the shifted operation be 'v <= f( )'. The first step is to required to create an operation 'w <= f( )' in place of the shifted operation, where 'w' is a new variable. In the second step, all the instances of the operation 'v <= f( )' in the loop body in the original behavior are placed by two operations 'v <= w' and 'w <= f( )' in parallel. Finally, the operations that use the variables 'v' in the loop body will now use the variable 'w' instead of variable 'v'. It is demonstrated in figure (d). This situation is similar to the speculation and can be handled.

REFERENCES :

[1] D. Sarkar and S. C. De Sarkar, ***"Some Inference Rules for Integer Arithmetic for Verification of Flowchart Programs on Integers",*** IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 15**,** NO. I. JANUARY 1989.

[2] D. Sarkar and S. C. De Sarkar **, *"A Theorem Prover for Verifying Iterative Programs Over Integers",*** IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 15, NO. 12, DECEMBER 1989.

[3] Chandan Karfa, ***"Hand-in-hand verification and Synthesis of Digital circuits"***, M. S. Thesis, I.I.T. KHARAGPUR, 2007.

[4] Chandan Karfa, Chittaranjan Mandal, Dipankar Sarkar, Pramod Kumar, ***"An Equivalence Checking Method for Scheduling Verification in High – Level Synthesis"***, IEEE Transactions on COMPUTER-AIDED DESIGN of Integrated Circuits and Systems, VOL. 27, No. 3, March-2008.