Chapter 2
Background

In this chapter, we provide necessary background to understand our works. We explain scheduling and assignment procedures for data path generation in high-level synthesis, test pattern generation and test response analysis for built-in self-test (BIST), and integer linear programming (ILP) for solving various tasks of high level synthesis. We review relevant works in high-level BIST synthesis.

2.1 High-Level Synthesis

High-level synthesis is a process of transforming a behavioral description into a structural description comprising data path logic and control logic [27]. First, a behavioral description is converted into a control data flow graph (CDFG). Then operations are scheduled in clock cycles (scheduling), a hardware module is assigned to each operation (module assignment), and registers are assigned to input and output variables (register assignment). A CDFG and basic tasks in high-level synthesis area such as scheduling, transformation, module assignment, register assignment, and interconnection assignment are described in the following subsections.

2.1.1 Control Data Flow Graph

High-level synthesis reads a high-level description and translates it into an intermediate form. An intermediate form should represent all the necessary information and be simple to be applicable in high-level synthesis. A control data flow graph (CDFG) refers to such an intermediate form. A data flow graph, in which control information is omitted from a CDFG, is used frequently for data path-intensive circuits (to which high-level synthesis applies) such as filters and signal processing applications. They often do not require control of the data flow. There are controller-intensive circuits in which control of the data flow is required. Control data flow graph representation is used for such circuits. More specific descriptions of the two types of circuits and their representations are described in the following two subsections.

2.1.1.1 DFG for Data Path Intensive Circuit

A 6th order FIR (Finite Impulse Response) filter, one of data path-intensive circuits, is represented as,

$$y = h_0 x_0 + h_1 x_1 + h_2 x_2 + h_3 x_3 + h_4 x_4 + h_5 x_5 + h_6 x_6$$

where h_n is a filter coefficient, and x_n is a delayed value of x_{n-1}. The equation consists of seven multiplications and one summation. Fig. 2.1 shows a block diagram of a 6th order FIR filter.

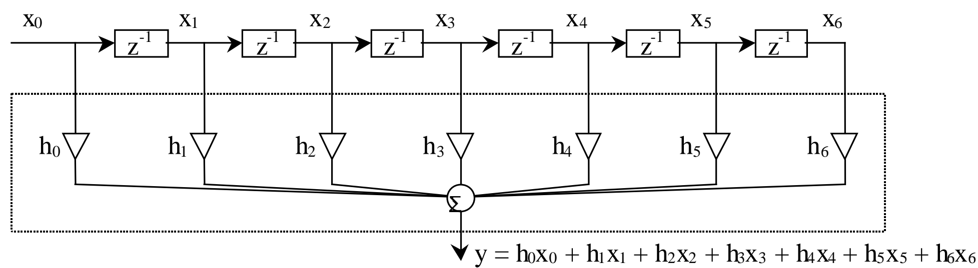


Figure 2.1. A 6th order FIR filter

Fig. 2.2 is a Silage program [45] that describes the behavior of the 6th order FIR filter. Delay elements (outside the dashed line in Fig. 2.1) are implemented as a shift register and are

not described in the Silage program. Fig. 2.3 shows an unscheduled data flow graph of the 6th order FIR filter.

```
#define num8 num<8,7>
#define h0 0.717.
…
func main(x0, x1, x2, x3, x4, x5, x6: num9) y: num8 =
begin
    t0 = num8(h0 * x0);
    t1 = num8(h1 * x1);
    t2 = num8(h2 * x2);
    t3 = num8(h3 * x3);
    t4 = num8(h4 * x4);
    t5 = num8(h5 * x5);
    t6 = num8(h6 * x6);
     y = (((((t0 + t1) + t2) + t3) + t4) + t5) + t6;
end;
```

Figure 2.2 Silage program of a 6th order FIR filter

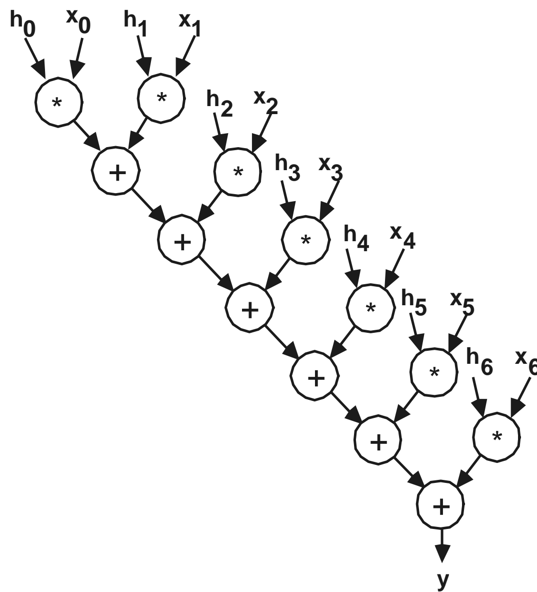


Figure 2.3 Unscheduled data flow graph of a 6th order FIR filter

2.1.1.2 CDFG for Controller-Intensive Circuits

A description for controller-intensive circuits has constructs which control the flow of data such as if-then-else, and for (while)-loop. The representation of such constructs affects greatly the performance of high-level synthesis tasks. A control data flow graph for control constructs, if-then-else and while-loop, are shown in Fig. 2.4 (a) and (b). Fig. 2.5 (a) shows description of a GCD (Greatest Common Divisor) [7], a controller-intensive MCNC high level benchmark circuit [49], in VHDL and in a control data flow graph.

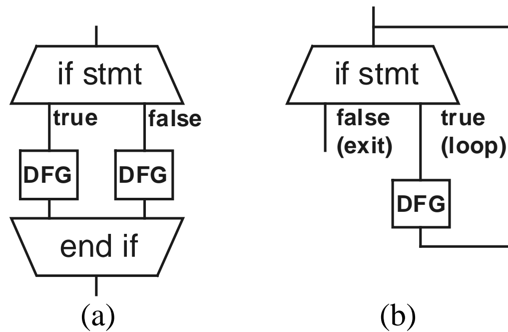
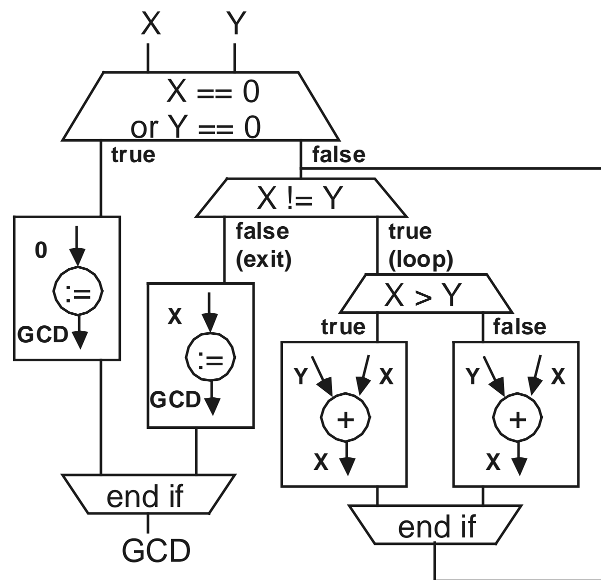


Figure 2.4 Basic control data flow graph elements for (a) if-then-else (b) while-loop

```
process
  begin
    X := PortX;
    Y := PortY;
    If ( X == 0 ) or ( Y == 0 ) then
        GCD := 0;
    else
        while ( X != Y ) loop
            if ( X > Y ) then
                X := X – Y;
            else
                Y := Y – X;
            end if; Y
        end loop;
        GCD := X;
    end if;
    PortGCD <= GCD;
  end process;
```



(a) (b)

Figure 2.5 GCD description in (a) behavioral VHDL (b) control data flow graph

Unlike a data flow graph, a control data flow graph can possibly have many different representations.

2.1.2 Scheduling

Scheduling determines the precise *start time* of each operation for a given data flow graph [27],[30]. The start times must satisfy the original dependencies of the graph, which limit the amount of parallelism of the operations. This means that the scheduling determines the concurrency of the resultant implementation, which, in turn, affects the performance. The maximum number of concurrent operations of any given type at any step of the schedule is a lower bound on the number of required hardware resources of the type. Therefore, the choice of a schedule affects the area and the BIST design.

Three commonly used scheduling algorithms are ASAP (As Soon As Possible), ALAP (As Late As possible), and list scheduling. In the ASAP scheduling, the start time of each operation is assigned its *as soon as possible* value. This scheduling solves an unconstrained minimum-latency scheduling problem in polynomial time. An example DFG (called paulin from [50]) and the corresponding ASAP scheduled one are shown in Fig. 2.6 and Fig. 2.7, respectively. In the ALAP scheduling, the start time of each operation is assigned its *as late as possible* value, and the scheduling is usually constrained in its latency. When it is applied to an unconstrained scheduling, the latency bound $\bar{\lambda}$ (which is the upper bound of latency) is the length of the schedule computed by the ASAP algorithm. When the ALAP algorithm is applied to the DFG in Fig. 2.6, the resultant DFG is obtained as shown in Fig 2.8. The latency for the DFG is 4.

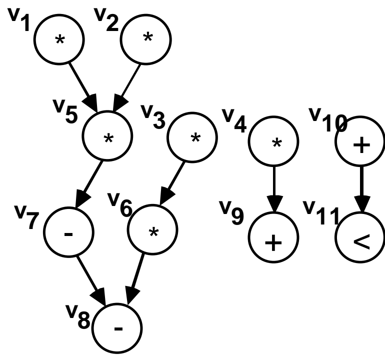


Figure 2.6 Paulin's DFG

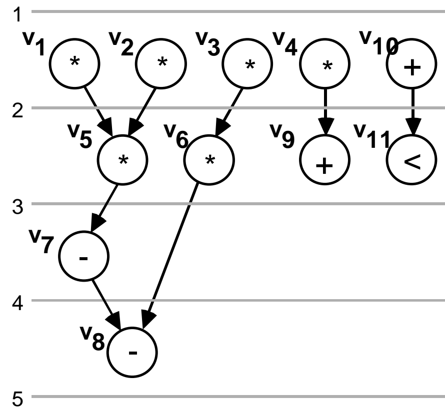


Figure 2.7 ASAP Scheduled DFG

The list scheduling [51], one of the most popular heuristic methods, is used to solve scheduling problems with resource constraints or latency constraints. A list scheduling maintains a priority list of the operations. A commonly used priority list is obtained by labeling each vertex with the weight of its longest path to the sink and ranking the vertices in the decreasing order. The most urgent operations are scheduled first. It constructs a schedule that satisfies the constraints. However, the computed schedule may not have the minimum latency. Fig. 2.9 shows the result of the list scheduling for the DFG in Fig. 2.6.

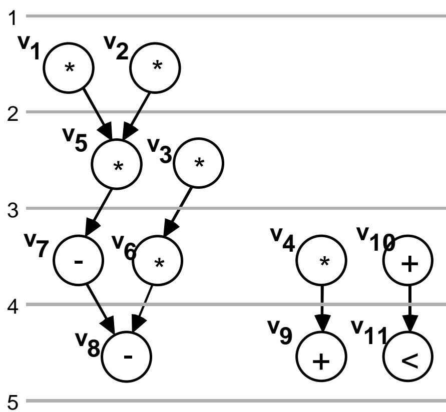


Figure 2.8 ALAP scheduled DFG

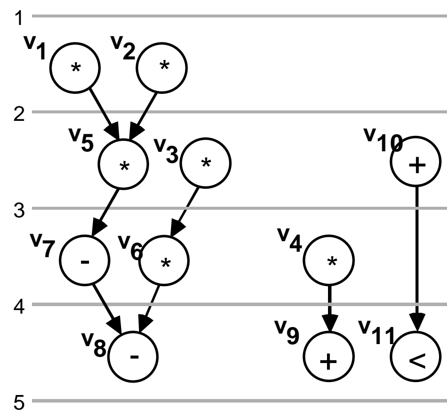


Figure 2.9 List scheduled DFG

2.1.3 Transformation

Transformation is to modify a given DFG, while preserving the functionality of the original DFG. A transformed DFG usually yields a different hardware structure, which leads to a different structure for testability and power consumption. Following techniques are available for a behavioral transformation of DFGs:

- Algebraic laws and redundancy manipulation:
  Associativity, commutativity, and common sub-expression
- Temporal Transformation:
  Retiming, pipelining, and rephasing
- Control (Memory) Transformation:
  Loop folding, unrolling, and functional pipelining
- Sub-Operation level Transformation:
  Add/shift multiplication and multiple constant multiplication

Among the above techniques, pipelining and loop folding/unrolling are most commonly used and are described in detail. There are two types of pipelining, structural pipelining and functional pipelining, which are also explained below.

2.1.3.1 Structural Pipelining

Pipelined modules are used in structural pipelining. If there is no data dependency exists between the operations and the operations (which implies they do not start in the same control step), pipelined modules can be shared, even if the corresponding operations overlap in their executions. For example, suppose that a pipelined multiplier resource, with execution delay of 2 and a *data introduction interval* of 1, is available. Then we can have a data flow graph with pipelined multipliers as shown in Fig. 2.10. Note that operations v6, v7, and v8 can share one pipelined multiplier.

2.1.3.2 Functional Pipelining

In a functional pipelining, it is assumed that there is a constraint on the global data introduction interval δ_0 representing the time interval between two consecutive executions. The number of required resources in pipelined implementations depends on δ_0. The smaller δ_0 is, the larger the number of operations executing concurrently is [30]. The result of the scheduling, using two stages and data introduction interval of two time steps is shown in Fig. 2.11. In the figure, operations in the stage 1 and stage 2 are executed concurrently and cannot be shared.

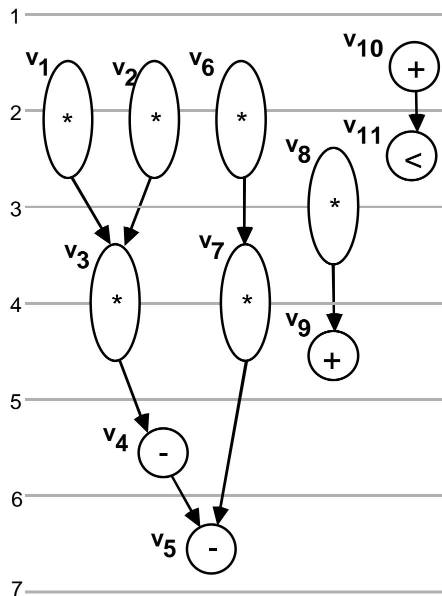


Figure 2.10 A Scheduled DFG using structural pipelining

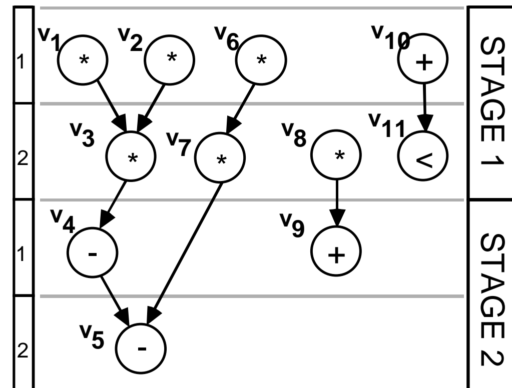
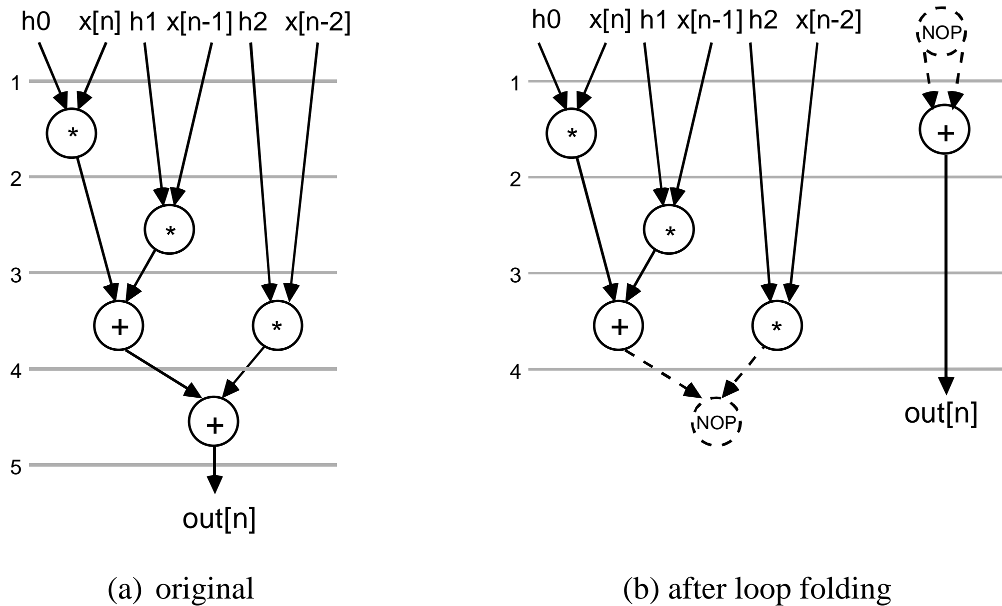


Figure 2.11 A Scheduled DFG using functional pipelining

2.1.3.3 Loop Folding

Loop folding is an optimization technique to reduce the execution delay of a loop. If it is possible to pipeline to the loop body itself with a local data introduction interval δ_l less than the latency of the loop body, the overall loop execution delay would be δ_l.

For example, a loop body of the data flow graph of FIR, shown in Fig. 2.12 (a), can be folded as shown in Fig. 2.12 (b) [52]. The loop execution delay would be reduced from 4 to 3. Note that a loop folding can be viewed as pipelining. However it is not a pipelined operation, since it can consume new data *only when the loop exit condition is met*.



(a) original (b) after loop folding

Figure 2.12 DFG for FIR

2.1.3.4 Loop Unrolling

Loop unrolling, or expansion, applies to an iterative construct with data-independent exit conditions. The loop is replaced by as many instances of its body as the number of operations. The advantage is to expand the scope of the other transformations. Consider a simple recursive computation of an IIR filter [53],

$$y_n = b_0 x_n + a_1 y_{n-1} \tag{6.6}$$

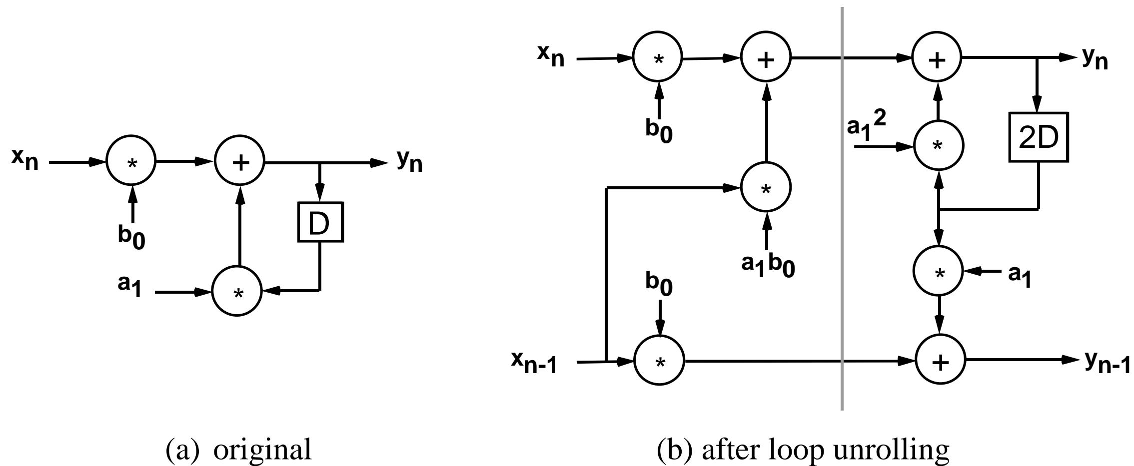
After loop unrolling it becomes,

$$y_n = b_0 x_n + a_1 b_0 x_{n-1} + a_1^2 y_{n-2} \tag{6.7}$$

$$y_{n-1} = b_0 x_{n-1} + a_1 y_{n-2}. \tag{6.8}$$

The data flow graphs of the original and unrolled computation are shown in Fig. 2.13 (a) and (b), respectively. After loop unrolling, several behavioral transformations such as pipelining and algebraic transformation may be applied to improve the performance such as reduction of power consumption.

Loop folding, which is the reverse operation of loop unrolling, is an optimization technique to reduce the execution delay of a loop. While the unrolling transformation is simpler, the folding transformation is more complex. This is because different folding descriptions result in different folded data-flow graphs while unrolling results in a unique data-flow graph.



(a) original (b) after loop unrolling

Figure 2.13 DFG for IIR

2.1.4 Module Assignment

Module assignment is to assign a module to each operation in a data flow graph, and it is applied after scheduling is completed. Often, a scheduling and a module assignment are performed simultaneously to provide better results. However, while performing scheduling, it is

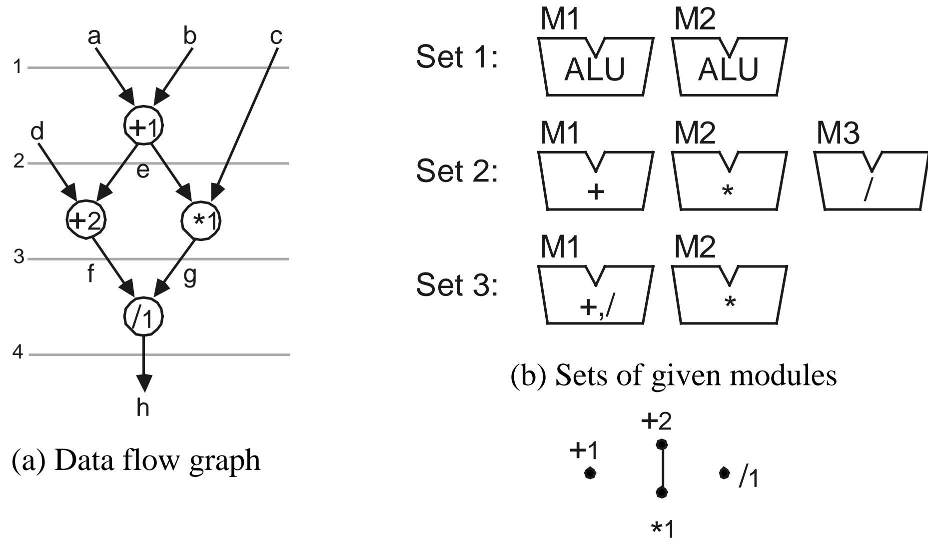
easy to figure out the minimum number of hardware resources such as modules and registers. Unlike the assignment operation that maps each operation (variable) to a specific module (register), allocation is to compute the number of necessary hardware resources for the given data flow graph.

The minimum number of modules and registers can be easily identified by counting the number of operations and variables exist in the same control step, respectively. Scheduling combined with allocation can generate an area-optimized data path without a complex assignment task. However, it is still necessary to combine assignment with scheduling for better testability or low power consumption.

Operations described in a behavioral language are mapped to proper data path modules available in the given library. Operations used in high-level synthesis can be classified as arithmetic and logical operations. Arithmetic operations include add, subtract, multiply, divide, and comparison. The corresponding register-level modules include adder, subtractor, multiplier, divider, comparator and so on. Logical operations include and, or, nand, nor, and so on. If a data-path library has modules with the same functionality but different characteristics, high-level synthesis can achieve better performance. For example, an "add" operation can be mapped to a ripple-carry adder, a carry-lookahead adder, or a carry-save adder. Similarly, a "multiply" operation can be mapped to an array multiplier, a wallace-tree multiplier, or a radix-n booth multiplier. The trade-off among different characteristics enables the synthesized circuit to have smaller area, higher performance, or less power consumption.

Operations in the same control step in a data flow graph should be assigned to different modules, and those operations are called *incompatible*. The incompatible operations cannot share the same modules. Fig. 2.14 (a) shows scheduled data flow graph, and Fig. 2.14 (b) shows three different sets of modules. When module set 1 is given, all the operations in the data flow graph can be assigned to any of the given modules unless there is incompatibility. The incompatibility graph of operations for the data flow graph for the module set 1 is shown in Fig. 2.14 (c). A vertex of the graph is an operation, and an edge exists between each pair of *incompatible* vertices. If multi-functional modules are not allowed as in module set 2 in Fig 2.14 (b), the

number of necessary modules increases to 3. If the module set 3 in Fig. 2.14 (b) is given, the number of necessary modules decreases to 2.



(a) Data flow graph

(b) Sets of given modules

(c) Incompatibility graph of operations for the set 1

Figure. 2.14. A data flow graph, given modules and incompatibility graph

2.1.5 Register Assignment

A data flow graph in which scheduling and module assignments have been completed is shown in Fig. 2.15 (a). Shaded lines in the data flow graph denote clock cycle boundaries. An input or output variable on a clock boundary should be stored in a register. In other words, a register should be assigned to each input or output variable, which is called register assignment. If two variables overlap at a clock boundary, the two variables are *incompatible*, and two incompatible variables cannot share the same register. The incompatibility graph of input and output variables for the data flow graph is shown in Fig. 2.15 (b). A vertex of the graph is a variable, and an edge exists between each pair of *incompatible* vertices. A compatibility graph of variables is readily derived from an incompatibility graph and is shown in Fig. 2.15 (c). In this case, an edge between two vertices indicates that the two vertices are *compatible*. During the

register assignment, a delayed variable spanning *n* clock cycles is often split into *n* variables to increase the flexibility of the register assignment [1].

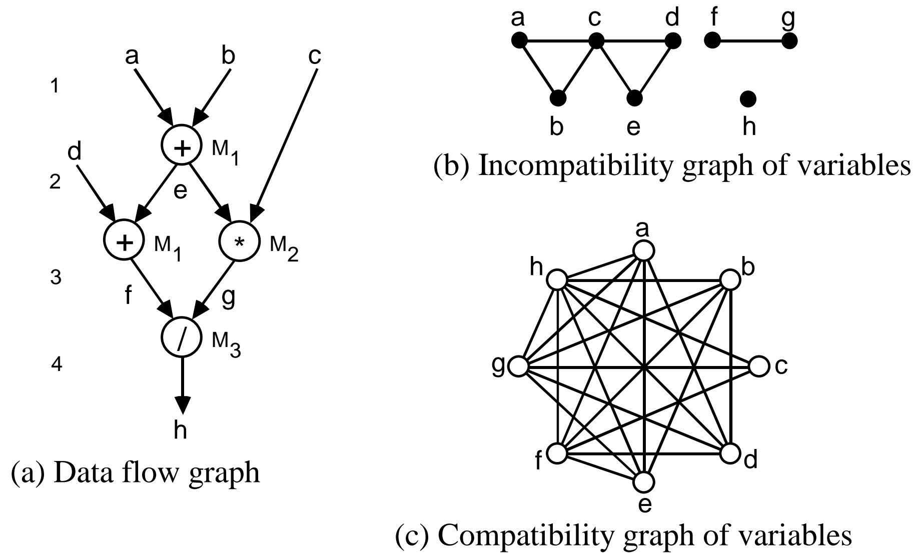


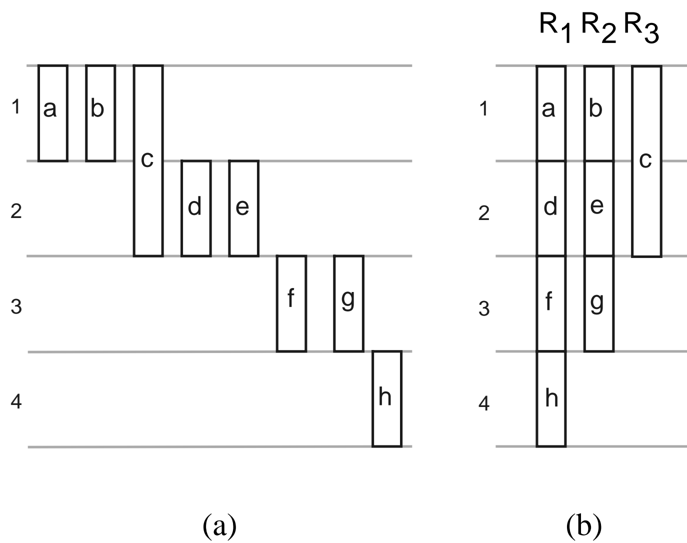
Figure. 2.15. A data flow graph and its incompatibility and compatibility graphs

Register assignment, including all the other assignments, can be solved by the graph coloring algorithm. For the graph coloring, there have been a number of algorithms such as left-edge algorithm [31], clique partitioning [32], perfect vertex elimination scheme (PVES) [33], bipartite-matching [54], and so on. All the methods mentioned above are based on heuristic, thus, they do not guarantee optimal solutions but generate results. In this section, PVES, left-edge algorithm, and clique partitioning are described briefly.

Perfect vertex elimination scheme was first described in [33] and was applied recently in the register assignment problem to enhance BIST testability [20]. It starts its operation by ordering the vertices in a graph. The ordering is determined by the maximum clique size of each vertex. For example, the ordering of the variables in the data flow graph in Fig. 2.15 (a) is {h, f, g, a, b, c, d, e}, and the maximum clique sizes are {1, 2, 2, 3, 3, 3, 3, 3}. The register assignment is performed in the following the order. If a variable is compatible with the variable(s) in a pre-assigned register, then the variable is assigned to the register. If not, it is assigned to a new

register. The register assignment for the example is R_1 = {h, f, a, d}, R_2 = {g, b, e}, and R_3 = {c}.

Left-edge algorithm was proposed by Hashimoto and Stevens in [31] to assign wires to routing tracks. The same idea has been applied to the register assignment by Kurdahi and Parker in [34], where variable intervals correspond to wires and routing tracks correspond to registers. The algorithm lists variable intervals as shown in Fig 2.16 (a). As long as the inverval of variable does not overlap, it is moved to the *left edge* as shown in Fig 2.16 (b). For example, we have register assignment as R_1 = {a, d, f, h}, R_2 = {b, e, g}, and R_3 = {c} as shown in the figure. Note that the assignment is the same as that for perfect vertex elimination scheme.



(a) (b)

Figure 2.16 Left edge algorithm (a) variable intervals (b) assignment results

Clique partitioning algorithm is an approach based on the heuristic proposed by Tseng and Siewiorek in [32]. Although it can be applied to any assignment, we describe it only for the register assignment.

A complete graph is one such that every pair of vertices has an edge. A clique of a graph is a complete subgraph. The size of a clique is the number of vertices of the clique. If a clique is

not contained by any other clique, it is called maximum. For the compatibility graph in Fig. 2.15 (c), the graph {a,d,f,h} is a clique of size 4 and is maximum. A clique partition is to partition a graph into a disjoint set of cliques. *Maximum clique partition* is a clique partition with the smallest number of cliques. To find a maximum clique partition is an NP-complete problem [30].

Two measurements, *common neighbor* and *non-common edge*, are used for clique partitioning. A vertex v is called *common neighbor* of a subgraph provided the vertex v is not contained in the subgraph and has an edge with every vertex of the subgraph. If a vertex v is not a common neighbor of a subgraph, but has an edge with at least one vertex of the subgraph, the edge is called *non-common edge*. For example, vertex h in the compatible graph in Fig. 2.15 (c) is a common neighbor of a subgraph {c,d,f}, but vertex g is not. As g has an edge c-g with vertex c, the edge c-g is a non-common edge for the subgraph. With common neighbor and non-common edge, all the vertices are ordered. For the example, all the vertices are assigned to registers in the same order as those in perfect vertex elimination scheme.

2.1.6 Interconnection Assignment

When module and register assignments are done, source and destination ports for interconnections are fixed in the data path. Depending on the interconnection architecture, there may be a conflict in accessing hardware resources. Thus, interconnection assignment is to connect source and destination ports without conflict while minimizing interconnection resources. The interconnection assignment is greatly affected by the target data path architecture. The data path architecture is based on multiplxers as shown in Fig 2.17 (a) or buses as shown in Fig 2.17 (b).

The multiplexer-based architecture shown in Fig 2.17 (a) is simpler in structure than other architectures. Hence, the problem space is small, which makes it easy to find an optimal solution for the structure. It is possible to employ layered multiplexers as shown in Fig 2.17 (c). In such a case the complexity of interconnections is reduced. However, as the structure becomes

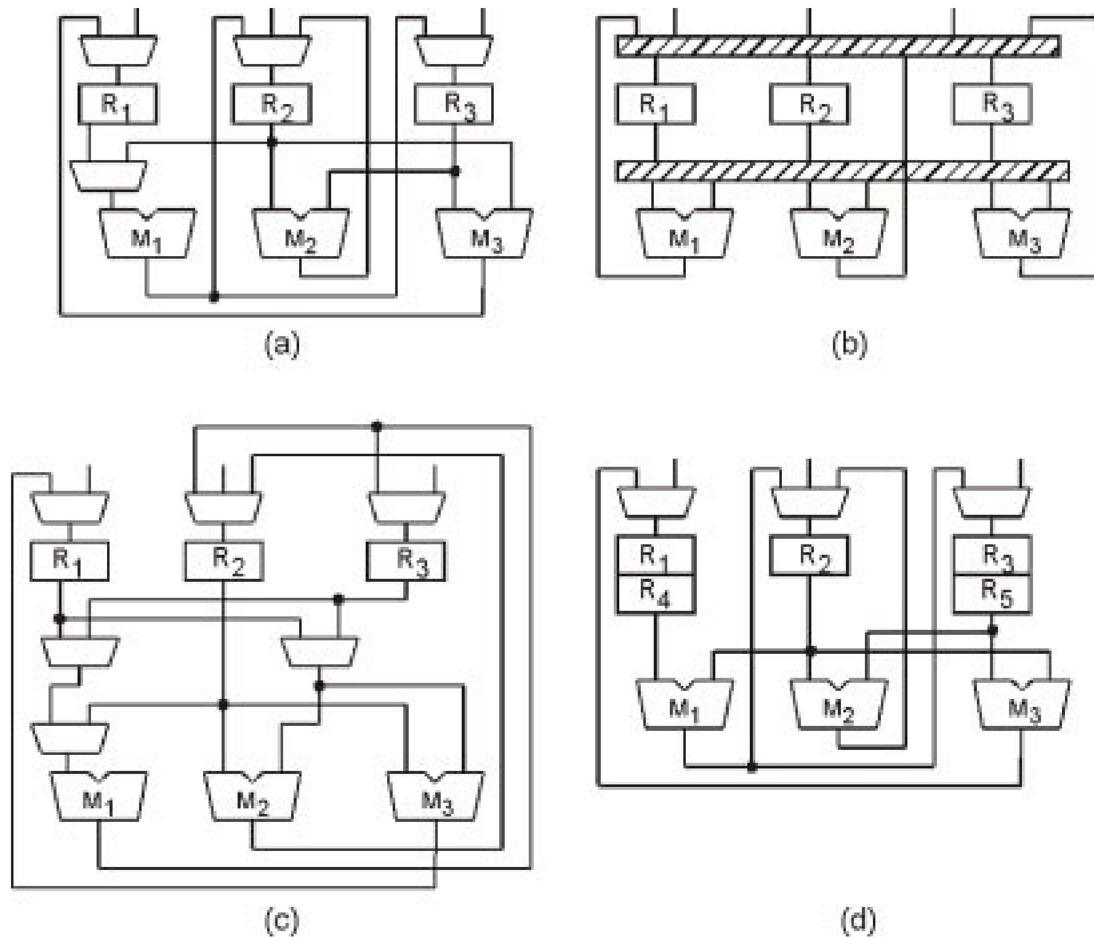
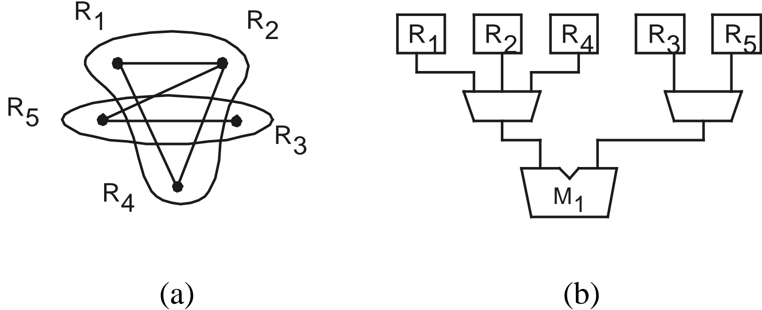


Figure 2.17 Various data path architectures

complicated, it is more difficult to find an optimal solution. The bus-based architecture can be used to minimize the number of interconnections. As multiple hardware resources share the same bus, tri-state buffers are used, and a conflict in using a bus should be resolved. The architecture as shown in Fig 2.17 (d) uses register files [55], or often referred to as multiport memory. The number of interconnections can be further reduced by using register files. However, register files require decoder circuits for multiport-access to degrade the performance.

One way to minimize the number of interconnections is to swap input ports of the module or of the operation in a data flow graph. For example, suppose that we have a data path that performs R_1+R_3, R_1+R_5, R_2+R_3, R_3+R_4, and R_4+R_5. Our objective is to swap operands, so that the resultant circuit has the smallest number of interconnections [56]. When there are two operands for an operation, both operands cannot be placed on one side, either left or right. Thus those operands have incompatibility, or no edge, in the compatibility graph as shown in Fig 2.18

(a). From the compatibility graph, we have two cliques, {R1, R2, R4} and {R3, R4}. These cliques lead to the smallest number of interconnections as shown in Fig 2.18 (b).



(a) (b)

Figure 2.18 Input swapping: (a) compatibility graph (b) simplified interconnection

2.2 Built-In Self-Test (BIST)

Built-in self-test (BIST) is one of design-for-test (DFT) techniques in which testing circuit, test pattern generators (TPGs) and signature analyzers (SAs), are embedded in the circuit [28]. A typical BIST structure is shown in Fig. 2. 19. A BIST is necessary to test hard-to-access internal nodes such as embedded memory. BIST has many advantages such that at-speed testing, no need for test pattern generation and test data evaluation, elimination of expensive automatic test equipment (ATE), and easy field testing.

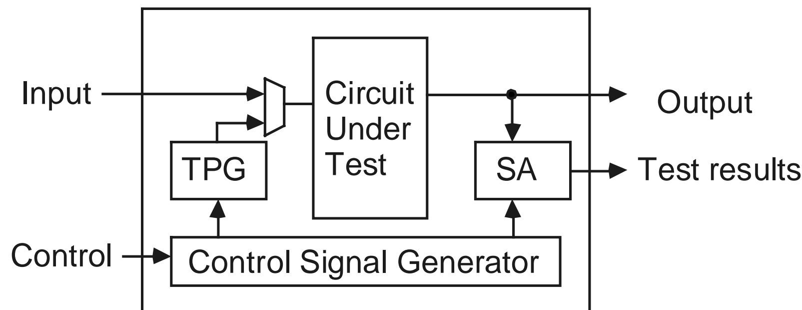


Figure 2.19 Typical BIST structure

When a BIST is performed under normal function, it is referred to *on-line* BIST. Whereas, BIST is performed in *off-line* BIST under test mode. The methods regarding BIST described in this thesis assume off-line BIST. Parallel BIST, which is based on random pattern testing, employs test pattern generators and test data evaluators for every module under test (which is usually a combinational circuit). Parallel BIST often achieves relatively high fault coverage compared with other BIST methods such as circular BIST [29]. In this thesis, we present a high-level BIST synthesis method that employs the parallel BIST structure.

The objective of the BIST synthesis considered in the thesis is to assign registers to incur the least area overhead for the reconfiguration of assigned registers, while the processing time is within the practical limit. Two constraints are imposed in our BIST synthesis (and in most BIST synthesis systems). *First, test pattern generators and signature registers are reconfigured from*

existing system registers. In other words, all test registers function as system registers during normal operation. *Second, extra paths are not added for testing*. The constraints can be met through the reconfiguration of existing registers into four different types of test registers described below.

A system register may be converted into one of four different types of test registers: a test pattern generator (TPG), a multiple input signature register (short for signature register), a built-in logic block observer (BILBO) [57], or a concurrent BILBO (CBILBO) [58]. If a register should be a TPG and a signature register (SR) at the same sub-test session, it should be reconfigured as a CBILBO. If a register should behave as a TPG and an SR, but not at the same time, it should be reconfigured as a BILBO. Reconfiguration of a register into a CBILBO requires twice the number of flip-flops of the register. Hence, it is expensive in hardware costs.

A TPG can be shared between modules as long as each input of a module receives test patterns from different TPGs. (Application of the same test patterns to two or more inputs of a module does not achieve a high fault coverage due to the dependence of test patterns.) However, an SR cannot be shared between modules tested in the same sub-test session. Therefore, *the number of sub-test sessions necessary for a test session is usually determined by the number of modules sharing the same SR*.

Consider a DFG in which all operators are assigned to N modules. Through an appropriate register assignment, it is possible that the N modules can be tested at least once using exactly *k* sub-test sessions where *k* is 1, 2, …N. Test registers are reconfigured to TPGs and/or SRs in each sub-test session, and a subset of modules is tested in a sub-test session. When a BIST design is intended to test all of the modules in *k* sub-test sessions, we say that the BIST design is for *k*-test session. As extreme cases, BIST design for 1-test session tests all modules in one sub-test session, while a BIST design for N-test session tests only one module in each sub-test session.

2.2.1 Test Pattern Register (TPG) and Linear Feedback Shift Register (LFSR)

For the test method, there are two approaches: exhaustive and pseudorandom testing. Exhaustive testing applies all 2^n test patterns generated by counters and linear feedback shift registers (LFSRs). While it can detect all combinational faults, it is applicable only to circuits with a small number of inputs. In pseudorandom testing, test patterns resemble random but are deterministic. The test patterns for pseudorandom testing are generated by LFSRs as shown in Fig. 2.20 (a). The connection polynomial function p(x) determines feedback points. Fig. 2.20 (b) shows an LFSR with a connection polynomial function, $p(x) = x^4+x+1$.

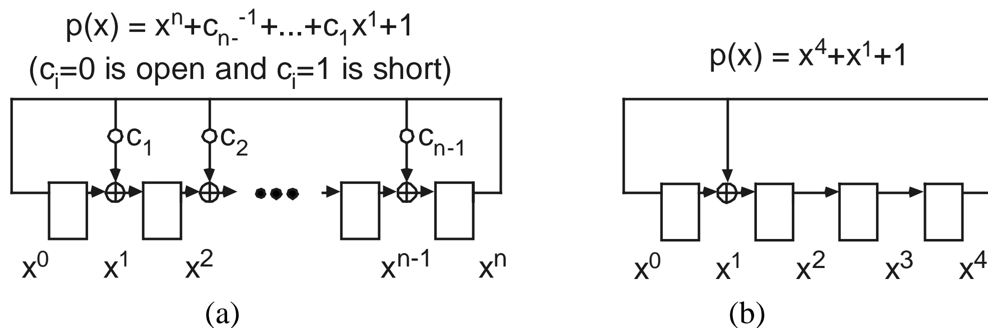


Figure 2.20 LFSR for (a) generic case and, (b) n=4

When a *primitive polynomial* is used to construct an LFSR, the resultant LFSR generates all patterns except the all-zero patterns. Such LFSR is called *maximum length* LFSR. There is at least one primitive polynomial for any order *n*. Table 2.1 shows primitive polynomials for some orders in the range of 2 to 32. From the table, $p(x) = x^{16}+ x^5+x^3+x^2+1$ is primitive for order 16, which is used for our 8-point discrete cosine transform circuit to be presented in Chapter 4.

Table 2.1 primitive polynomial table

Order	Term	Order	Term	Order	Term
2,3,4,6,7	1 0	10	3 0	14	12 11 1 0
5	2 0	11	2 0	15	1 0
8	6 5 1 0	12	7 4 3 0	16	5 3 2 0
9	4 0	13	4 3 1 0	32	28 27 1 0

2.2.2 Signature register (SR)

Since test patterns of BIST resemble random, BIST requires a large number of test patterns to achieve a reasonably high fault coverage. The task of storing and comparing a large number of test responses directly on the chip is impractical. Signature analysis is one of the most widely used compression techniques to compress test responses. Signature analyzer performs repetitive division operations and the remainder called signature is compared against the good signature.

In general, a circuit under test has multiple outputs. Thus, a signature analyzer with multiple-inputs, often referred to as multiple-input signature register (MISR) in BIST, is shown in Fig. 2.21, and is considered in this thesis.

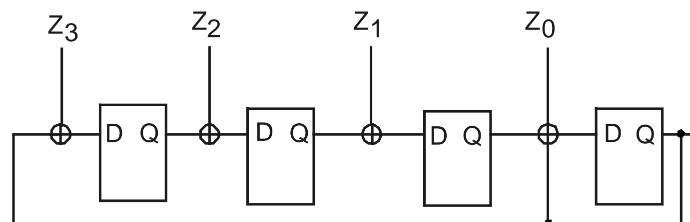


Figure 2.21 A schematic of signature register

2.2.3 Built-In Logic Block Observer (BILBO)

As described earlier, if a test register behaves as a TPG in a sub-test session and an SR in another session, the test register should be reconfigured as a BILBO as shown in Fig. 2.22. Scan in/out functionality of the circuit is necessary for initialization of LFSRs and retrieval of test responses. BILBO has four modes such that normal function, shift register (scan in/out), TPG/MISR, and reset, and are shown in Table 2.2.

Table 2.2 Operating modes of built-in block observer

B1	B2	Modes
1	1	Normal function
0	0	Shift register
1	0	TPG and MISR
0	1	Reset

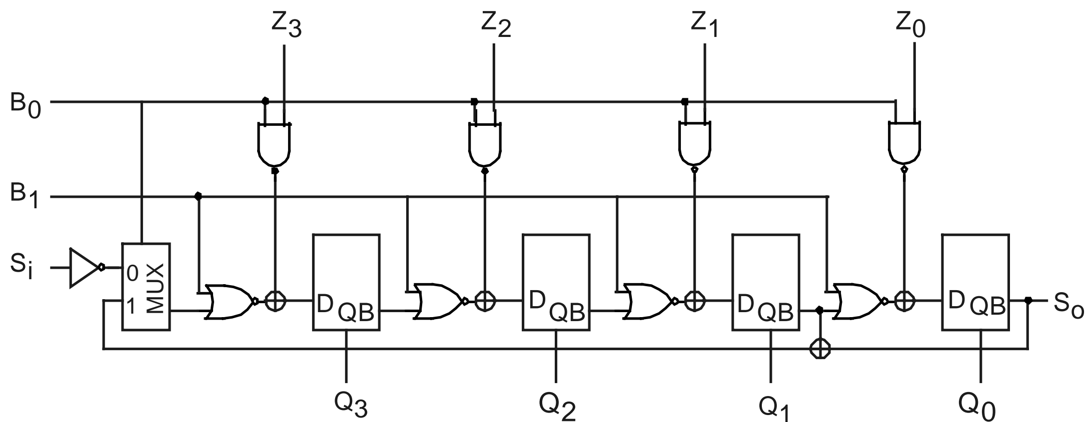


Figure 2.22 A schematic of BILBO

2.2.4 Concurrent BILBO (CBILBO)

If a test register should be a TPG and an SR in the *same* sub-test session, it should be reconfigured as a CBILBO. CBILBO, as shown in Fig 2.23, uses two layers of LFSRs to provide TPG and SR functionality simultaneously. CBILBO has three modes, as shown in Table 2.3, such that normal function, shift register (scan in/out), and TPG/MISR. CBILBO incurs high area overhead and hence should be avoided, if possible.

Table 2.3 Operating modes of concurrent built-in block observer

B0	B1	Modes
–	0	Normal function
1	1	Shift register
0	1	TPG and MISR

Figure 2.23 A schematic of CBILBO

2.2.5 BIST Register Assignment

If a cycle is formed for a module through a register, it is called self-adjacent register. Assignment of input and output variables of a module to the same register creates a self-adjacent

register, viz. register R_1 in Fig. 2.24. A self-adjacent register often requires a CBILBO. The main focus of early high-level BIST synthesis methods focused on the avoidance of self-adjacent registers [17], [18]. However, self-adjacent registers do not necessarily require CBILBOs. Consider part of a data flow graph given in Fig. 2.25 (a). The data path logic under a register assignment, R_1={c,e}, R_2={a,f}, and R_3={b,d}, is shown in Fig. 2.25 (b). The necessary reconfiguration of registers to test module M_1 is also indicated in the figure. Note that Register R_1 is self-adjacent, but is reconfigured to a SR, not a CBILBO.

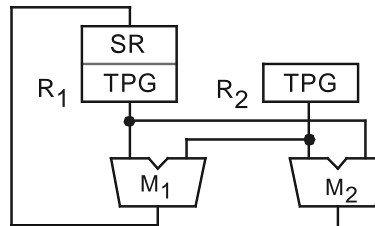
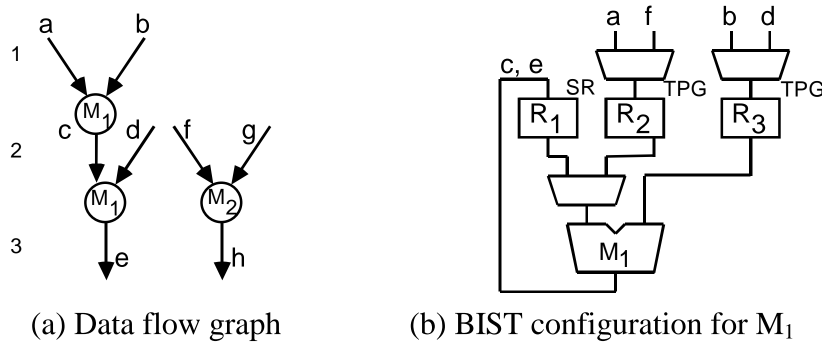


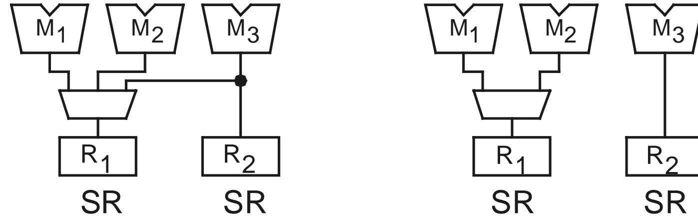
Figure. 2.24 Self-adjacent register



(a) Data flow graph (b) BIST configuration for M_1

Figure. 2.25 Self-adjacent register and its reconfiguration to a SR

Parulkar et al.'s method aims to reduce the overall area overhead by sharing registers in their maximum capacity [20]. However, excessive sharing of signature registers is unnecessary. In fact, it may result in higher area overhead. Consider the cases given in Fig. 2.26. Registers R_1 and R_2 are reconfigured into signature registers during testing. The sharing of R_1 with M_3 in Fig. 2.26 (a) is unnecessary for two test sessions as shown in Fig. 2.26 (b), and it creates an unnecessary path to incur higher area overhead.



(a) Excessive sharing (b) Proper sharing

Figure. 2.26. Allocation of signature registers

2.3 Integer Linear Programming

In this section, we illustrate an ILP model for high-level synthesis and describe necessary terminologies.

2.3.1 Scheduling with the ILP Model

The following nomenclature is used for the ILP model described in this thesis, otherwise specified. *All the examples given in this section are for the DFG and the data path in Fig. 2.6.*

- V is the set of operations: $V = \{v_1, v_2, \ldots, v_{11}\}$.
- E is the set of edges representing variables. E is represented as a set of ordered doubles (v_i, v_j), where v_i and v_j are associated operations of the edge: $E = \{(v_1, v_5), (v_2, v_5), \ldots\}$.
- C is the set of available control steps: $C = \{0, 1, 2, 3, \ldots\}$.
- K is the type of modules: $K = \{1, 2, 3, 4\}$. Note that this definition is used only in this section. It represents the set of test sessions in Chapter 3.
- $\mathrm{T}(v_i)$ represents resource type of operation v_i: $\mathrm{T}(v_1) = 1$.
- t_i^S represents a lower bound for operation v_i's start time measured in control steps set by ASAP scheduling: $t_4^S = 1$.

- t_i^L represents an upper bound for operation v_i's start time *measured in control steps* set by ALAP scheduling: $t_4^L = 3$.
- d_i represents execution delay of operation v_i: $d_4 = 1$.

Using an ILP model [30],[35],[36],[38], the minimum latency scheduling under a resource constraint and minimum resource scheduling under a latency constraint can be solved. Let us consider a minimum latency scheduling. A binary variable x_{il} is 1 only when the start time of operation v_i is assigned to control-step t. Note that upper and lower bounds on the start times can be computed using ASAP and ALAP algorithms. List scheduling (which is a fast heuristic scheduling) is used to derive an upper bound for latency, $\overline{\lambda}$, which, in turn, is used in ALAP scheduling. Using upper and lower bounds for the start time of operations, the number of ILP constraints can be reduced. For example, a binary variable, x_{it} is necessarily zero for $l < t_i^S$ or $l > t_i^L$ for any operation v_i, $i \in \{0,1, \ldots, n\}$ where n is the number of operations. We now consider ILP formulations for the minimum latency scheduling under a resource constraint in which the start time for each operation is unique. Thus,

$$\sum_t x_{it} = 1, \forall i . \tag{2.1}$$

Therefore, the start time of any operation v_i can be stated in terms of x_{it} as $t_i = \sum_t t \cdot x_{it}$. The data dependence relations represented in a data flow graph must be satisfied. Thus,

$$t_i \geq t_j + d_j \ \forall i, j : (v_j, v_i) \in E \tag{2.2}$$

which is the same as:

$$\sum_t t \cdot x_{it} \geq \sum_t t \cdot x_{jt} + d_j, \quad i, j \in \{0,1,\ldots,n\} \ : (v_j, v_i) \in E . \tag{2.3}$$

Finally, the number of the type k resources required by an operation v_i at a control step t should not exceed a resource constraint a_k. Thus,

$$\sum_{i:\mathrm{T}(v_i)=k} \sum_{m=t-d_i+1}^{t} x_{im} \le a_k, \quad \forall k \in K, \quad \forall t \in C, \tag{2.4}$$

Note that m can be larger than 1 for a multicycle operation. The objective of the formulation is to minimize latency that is to minimize $\sum_i t_i = \sum_i \sum_t t \cdot x_{it}$.

To illustrate the ILP formulations for scheduling, the data flow graph in Fig. 2.6 is used in the following. The start time of every operation is bound by the result of ASAP and ALAP schedulings. Let N_m, N_a, N_s and N_c be the number of multipliers, adders, subtractors and comparators, respectively. They are given as 2, 1, 1, and 1, respectively. First, all operations must start only once:

$$\begin{aligned}
x_{11} &= 1 \\
x_{21} &= 1 \\
x_{31} + x_{32} &= 1 \\
x_{41} + x_{42} + x_{43} &= 1 \\
x_{52} &= 1 \\
x_{62} + x_{63} &= 1 \\
x_{73} &= 1 \\
x_{84} &= 1 \\
x_{92} + x_{93} + x_{94} &= 1 \\
x_{10,1} + x_{10,2} + x_{10,3} &= 1 \\
x_{11,2} + x_{11,3} + x_{11,4} &= 1.
\end{aligned}$$

Then, by the resource constraints,

$$x_{1,1} + x_{2,1} + x_{3,1} + x_{4,1} \le 2 = N_m$$
$$x_{3,2} + x_{4,2} + x_{5,2} + x_{6,2} \le 2 = N_m$$
$$x_{4,3} + x_{6,3} \le 2 = N_m$$
$$x_{7,3} \le 1 = N_s$$
$$x_{8,4} \le 1 = N_s$$
$$x_{10,1} \le 1 = N_a$$
$$x_{9,2} + x_{10,2} \le 1 = N_a$$
$$x_{9,3} + x_{10,3} \le 1 = N_a$$
$$x_{9,4} \le 1 = N_a$$
$$x_{11,2} \le 1 = N_c$$
$$x_{11,3} \le 1 = N_c$$
$$x_{11,4} \le 1 = N_c.$$

Finally, data dependence relations, represented by data flow graph, are,

$$1 \cdot x_{3,1} + 2 \cdot x_{3,2} - 2 \cdot x_{6,2} - 3 \cdot x_{6,3} \le -1$$
$$1 \cdot x_{4,1} + 2 \cdot x_{4,2} + 3 \cdot x_{4,3} - 2 \cdot x_{9,2} - 3 \cdot x_{9,3} - 4 \cdot x_{9,4} \le -1$$
$$1 \cdot x_{10,1} + 2 \cdot x_{10,2} + 3 \cdot x_{10,3} - 2 \cdot x_{11,2} - 3 \cdot x_{11,3} - 4 \cdot x_{11,4} \le -1.$$

The objective of the ILP is to minimize $\sum_i \sum_t t \cdot x_{it}$. An optimum solution for the data flow graph in Fig. 2.6, is the latency with 4 control steps and is shown in Fig. 2.27.

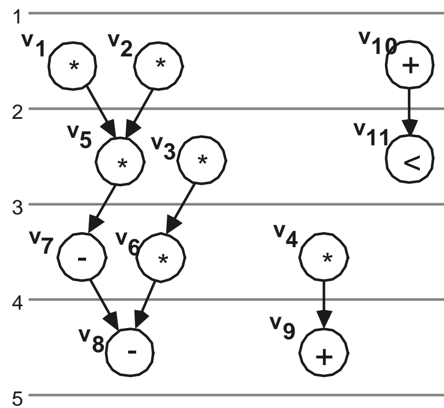


Figure 2.27 Scheduled data flow graph by ILP

Now, let us consider the minimum resource scheduling under a latency constraint for the example. The uniqueness constrains on the start time of the operations and the data dependence constraints are the same as before. However, the resource constraints are changed to unknown variables as follows:

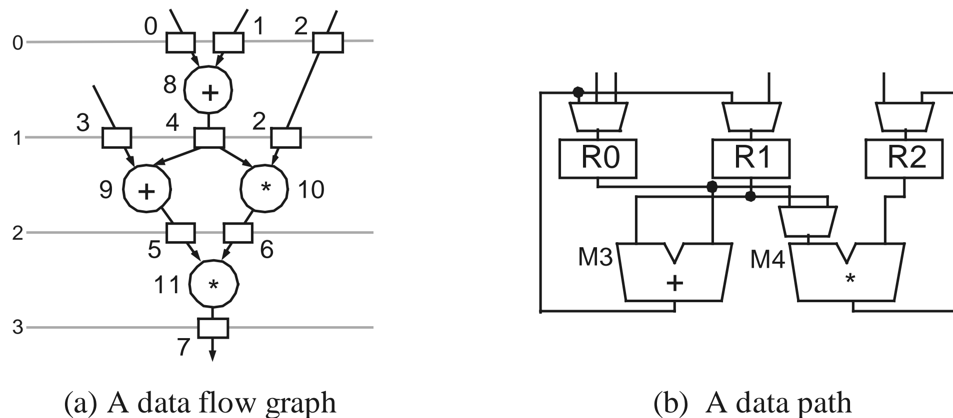
$$
\begin{aligned}
x_{1,1} + x_{2,1} + x_{3,1} + x_{4,1} &\le N_m \\
x_{3,2} + x_{4,2} + x_{5,2} + x_{6,2} &\le N_m \\
x_{4,3} + x_{6,3} &\le N_m \\
x_{7,3} &\le N_s \\
x_{8,4} &\le N_s
\end{aligned}
$$

and

$$
\begin{aligned}
x_{10,1} &\le N_a \\
x_{9,2} + x_{10,2} &\le N_a \\
x_{9,3} + x_{10,3} &\le N_a \\
x_{9,4} &\le N_a \\
x_{11,2} &\le N_c \\
x_{11,3} &\le N_c \\
x_{11,4} &\le N_c .
\end{aligned}
$$

2.3.2 Module and Register Assignments with the ILP Model

In this and the following sections, we illustrate an ILP model for assignments using an data flow graph (DFG) given in Fig. 2.28 and describe necessary terms to understand our method. We also discuss constraints imposed in our high-level BIST synthesis. *All the examples given in this and the following sections are for the DFG and the data path in Fig. 2.28.*



(a) A data flow graph (b) A data path

Figure 2.28 A data flow graph and a synthesized data path

Gray lines in the DFG denote clock cycle boundaries called control steps. All input and output variables on a clock boundary should be stored in a register. In other words, a register should be assigned to each input or output variable; this process is called register assignment. The data path in Fig. 2.28 (b) is obtained under a register assignment: R0 = {0, 4}, R1 = {1, 3, 6}, and R2 = {2, 5, 7}.

The *horizontal crossing* of a control step is the number of variables for the control step. Therefore, the minimum number of registers required for the synthesis of a DFG is equal to the maximal horizontal crossing of the DFG. For example, the horizontal crossing of the control step 0 and of the control step 1 is three and is maximal for the DFG. Hence, the minimum number of registers required for the synthesis of the DFG is three. The minimum number of modules necessary for a type of operation is obtained directly from the maximum concurrency of the operation. For example, if the maximum number of multiplication operations performed between

any two consecutive clock steps is three for a DFG, then at least three multipliers are needed to synthesize the DFG. The data path logic in Fig. 2.28 (b) contains the minimum number of registers (three) and the minimum number of modules (two). *We assume that the number of registers and modules to be used for the synthesis of a DFG are known a priori.*

The following nomenclatures are used for the ILP model described in the thesis.

- V_o is the set of operations: $V_o = \{8, 9, 10, 11\}$.
- V_v is the set of variables: $V_v = \{0, 1, 2, \ldots, 7\}$.
- *l* is the label of an input port of an operation. The leftmost input port is labeled as 0, the next port is 1, and so on. An input port is designated by its label.
- *I(o)* is the set of input ports for an operation *o*: $I(8) = \{0,1\}$ and $I(9) = \{0,1\}$.
- E_i is the set of ordered triples (*v*, *o*, *l*) defined for the *inputs* of all operations where *o* is an operation, *l* is an input port of the operation, and *v* is the variable on the input port *l:* $E_i =$ {(0,8,0), (1,8,1), (3,9,0), (4,9,1), (4,10,0), (2,10,1), (5,11,0), (6,11,1)}.
- E_o is the set of ordered doubles (*o*, *v*) defined for the *outputs* of all operations where *o* is an operation and *v* is the output variable of the operation: $E_o =${(8,4), ((9,5), (10,6), (11,7)}.
- *T* is the set of control steps: $T = \{0, 1, 2, 3\}$.
- *C* is the set of constants: $C = \emptyset$.

The following nomenclature is defined for a data path logic to be synthesized from a data flow graph. Input ports of modules are labeled in the same manner as those of operators.

- *R* is the set of the registers: $R = \{0, 1, 2\}$.
- *M* is the set of modules: $M = \{3, 4\}$.
- *I(m)* is the set of input ports of a module *m* where $m \in M$: $I(3) = \{0,1\}$ and $I(4) = \{0,1\}$.

A binary variable x_{vr} (x_{om}) is 1 only when a variable (operation) v (o) is assigned to a register (module) r (m) and is 0 otherwise. Each variable (operation) should be assigned to only one register (module). Hence,

$$\sum_{r \in R} x_{vr} = 1, \ \forall \ v \in V_v \tag{2.5}$$

$$\sum_{m \in M} x_{om} = 1, \ \forall \ o \in V_o. \tag{2.6}$$

From Eq. (2.5) and Eq. (2.6), we have the following set of equations:

$$\begin{aligned} x_{0,0} + x_{0,1} + x_{0,2} &= 1 \\ x_{1,0} + x_{1,1} + x_{1,2} &= 1 \\ &\dots \\ x_{8,3} + x_{8,4} &= 1 \\ x_{9,3} + x_{9,4} &= 1 \\ x_{10,3} + x_{10,4} &= 1 \\ x_{11,3} + x_{11,4} &= 1. \end{aligned}$$

If a variable is used to perform a necessary operation at a control step, the variable is called *active* at the control step. A binary constant a_{vt} (a_{ot}) is 1 only when a variable (operation) v (o) is active at a control step t and is 0 otherwise. For example, $a_{0,0} = a_{1,0} = a_{2,0} = 1$ and $a_{3,0} = a_{4,0}$ … $a_{7,0} = 0$ for the control step 0. Each active variable (operation) at a control step should be assigned to a different register (module). Hence,

$$\sum_{v \in V} x_{vr} \cdot a_{vt} \leq 1, \ \forall \ r \in R, t \in T, \ \sum_{o \in O} x_{om} \cdot a_{ot} \leq 1, \ \forall \ m \in M, t \in T. \tag{2.7}$$

From Eq. (2.7), the set of equations for the control step 0 is illustrated below.

$$
\begin{aligned}
x_{0,0} + x_{1,0} + x_{2,0} &\le 1 \\
x_{0,1} + x_{1,1} + x_{2,1} &\le 1 \\
x_{0,2} + x_{1,2} + x_{2,2} &\le 1 \\
&\dots
\end{aligned}
$$

Similar sets of equations are obtained for the other control steps 1, 2, and 3.

2.3.3 Interconnect Assignment with the ILP Model

A binary variable z_{rml} is 1 only if there is an interconnection, i.e., a wire, between a register r and an input port l of a module m, and is 0 otherwise. Let us consider a necessary interconnection for an edge $(v, o, l) \in E_i$. Suppose that the variable v and the module o are assigned to a register r and a module m, respectively. The existence of an edge between v and o necessitates an interconnection between r and the input port l of m. Hence, the variable z_{rml} should be 1. The relationship can be represented as

$$x_{vr} + x_{om} - z_{rml} \le 1,\ \forall r \in R, m \in M, (v,o,l) \in E_i. \tag{2.8}$$

From Eq. (2.8), the following set of equations is obtained for the interconnections for r=0 and m=3.

$$
\begin{aligned}
x_{0,0} + x_{8,3} - z_{0,3,0} &\le 1 \\
x_{1,0} + x_{8,3} - z_{0,3,1} &\le 1 \\
x_{3,0} + x_{9,3} - z_{0,3,0} &\le 1 \\
&\dots
\end{aligned}
$$

Similarly, a binary variable z_{mr} is 1 only if there is an interconnection between the output of a module m and a register r and is 0 otherwise. The constraint is represented as

$$x_{om} + x_{vr} - z_{mr} \le 1,\ \forall m \in M, r \in R, (o,v) \in E_o. \tag{2.9}$$

Commutative operations, in which the two input ports can be swapped, are modeled as follows [39]. Inputs are applied to input ports of the operation through pseudo-input ports. Let a binary variable $s_{l*,l,o} = 1$ if a connection exists between a pseudo-input port $l*$ and an input port l

of an commutative operation o. All of the possible connections between the pseudo and original inputs for operation 8 are shown in Fig. 2.29 (a).

The constraints for commutative operations can be written as

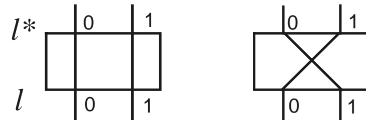
$$\sum_{l^* \in I(o)} s_{l^*,l,o} = 1, \ \forall \, l \in I(o) \tag{2.10}$$

$$\sum_{l \in I(o)} s_{l^*,l,o} = 1, \ \forall \, l^* \in I(o). \tag{2.11}$$

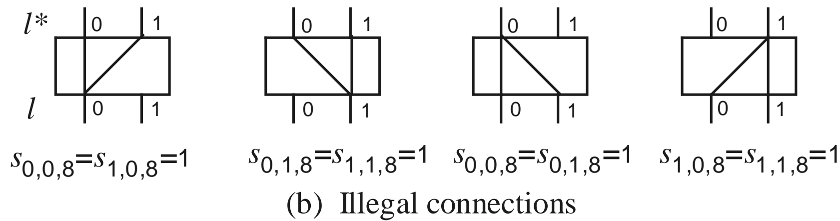
From Eq. (2.10) and Eq. (2.11), the constraints for operation 8 are obtained as

$$\begin{aligned} s_{0,0,8} + s_{1,0,8} &= 1 \\ s_{0,1,8} + s_{1,1,8} &= 1 \\ s_{0,0,8} + s_{0,1,8} &= 1 \\ s_{1,0,8} + s_{1,1,8} &= 1 \end{aligned}$$

The above set of equations exclude the illegal connections shown in Fig. 2.29 (b).



(a) Legal connections



(b) Illegal connections

Figure 2.29 Connections between pseudo-input ports and input ports

Hence, if a module is commutative, the constraints for interconnection can be written as

$$x_{vr} + x_{om} + s_{l^*,l,o} - 2 \cdot z_{rml} \geq 0, \ \forall \, r \in R, m \in M, l \in I(o), (v,o,l^*) \in E_i. \tag{2.12}$$

A synthesis procedure solves the above set of equations under a cost function such as area. The data path in Fig. 2.28 (b) is optimal in terms of the number of components and interconnections.

2.4 Literature Review

There are two major approaches to generate efficient data paths in terms of area, speed, testability, and power in high-level synthesis area. The first approach is based on heuristics and the other one is based on integer linear programming (ILP). A heuristic approach has been used extensively, since it can generate quality solutions within polynomial time. Whereas ILP based approach has been used to generate optimal solutions for small to midrange circuits. In this section, we review relevant existing methods based on heuristic or ILP in high level synthesis or high level BIST synthesis.

2.4.1 Heuristic Approach for High-Level BIST Synthesis

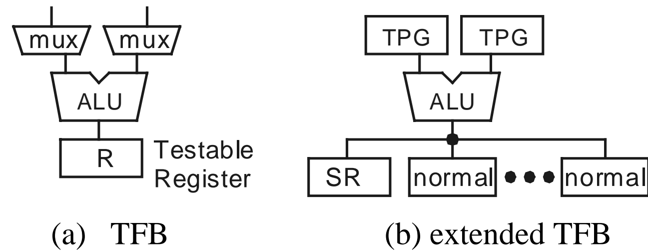
High-level BIST synthesis is to generate data path with BIST capability while minimizing test area overhead and/or test time. The heuristic approaches aimed to minimize test area overhead include SYNTEST [24], RALLOC [17], and BITS [20]. Whereas approach in [26] aims to generate a data path with minimum test time. Those works described in the following.

2.4.1.1 BIST Area Overhead Minimization

A. SYNTEST

One of the earliest high-level BIST synthesis methods based on the parallel BIST was proposed by Papachristou et al. [18]. In their method, operations and variables are assigned to a "testable functional block (TFB)", which consists of input multiplexers, an ALU, and output registers as in Fig. 2.30 (a). The objective of the assignment is to avoid self-adjacent registers, which incurs high area overhead. During the mapping operations in [18], if a newly assigned variable conflicts with assigned variables of a TFB, it requires an additional TFB. The problem

was later refined using a new structure called extended TFB as shown in Fig.2.30 (b), which further reduces the area overhead [24].



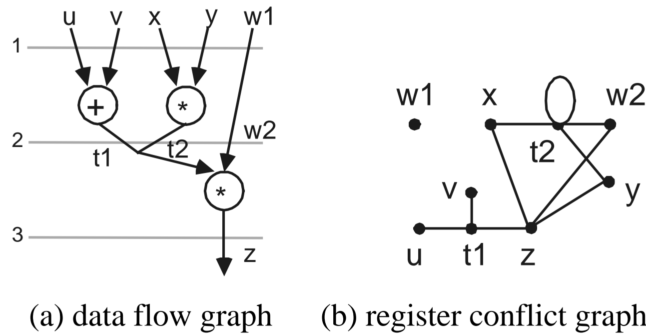
(a) TFB (b) extended TFB

Figure 2.30 A structure of testable functional blocks

B. RALLOC

Avra proposed a heuristic in [17] that guides the register assignment to avoid self-adjacent register whenever it is possible. The heuristic in based on a clique partitioning of a register conflict graph. A testability conflict edge is added between two nodes representing variables provided one node is an input to an operation and the other node is the output of the same operation. It is illustrated using the data flow graph shown in Fig. 2.31 (a). Suppose that the * operations in the data flow graph are assigned to the same module. If both variables x and z are assigned to the same register, then the register becomes self-adjacent register. The existence of an edge between x and z in the register conflict graph in Fig.2.31 (b) prevents such an assignment.

If two operations in consecutive clocks are assigned to the same block and if the output of one operation is an input to the other, the variable associated with both the input and the output of the block must be assigned to a CBILBO register. If a variable (or value) is active across several control steps, then it is split to minimize the number of CBILBOs. If splitting does not contribute to minimize the number of CBILBOs, it is merged back to save the number of multiplexer inputs. Thus, a weight is assigned to split variables (delayed values).



(a) data flow graph (b) register conflict graph

Figure 2.31 Register assignment in Avra's method

C. BITS

The register assignment in BITS [20] is performed using the perfect vertex elimination scheme (PVES). The order for the assignment is determined by a sharing degree and the size of maximum clique of a conflict graph. The sharing degree of a register is the number of different modules connected to the register. The goal of the assignment is to maximize the sharing. With highly shared data path logic, the number of TPGs and SRs necessary to test modules is reduced. In addition to that, the number of CBILBOs is minimized, by checking the condition of previously assigned registers and new variable to be assigned. The assignment of CBILBOs is performed only when there are no alternatives.

Let MCS(v) of a variable v denote the size of a clique containing v and SD(v) the sharing degree of variable v. The sharing degree in this case is the number of distinctive modules attached to the register to which the variable is to be assigned. Variables are ordered such that if a variable v appears before w, then $\mathrm{SD}(v) < \mathrm{SD}(w)$ or $\mathrm{MCS}(v) < \mathrm{MCS}(w)$ in case $\mathrm{SD}(v) = \mathrm{SD}(w)$. Following the order, variables are assigned such that if a variable conflicts with all existing registers then a new register is created. Otherwise, among the registers that do not conflict with the variable, pick a register, in the following priority:

- maximize the increase of sharing degree when adding new variable,
- maximize sharing degree of the register, and
- minimize the interconnection cost.

For example, a data flow graph in Fig. 2.32 (a) has a conflict graph of variables as shown in Fig. 2.32 (b). The ordering of PVES for the data flow graph is {h, a, b, e, f, g, d, c} with SDs and MCSs calculated as in Table 2.3 for each variable. The reverse order of that is {c, d, g, f, e, b, a, h} and by which register assignment is performed sequentially.

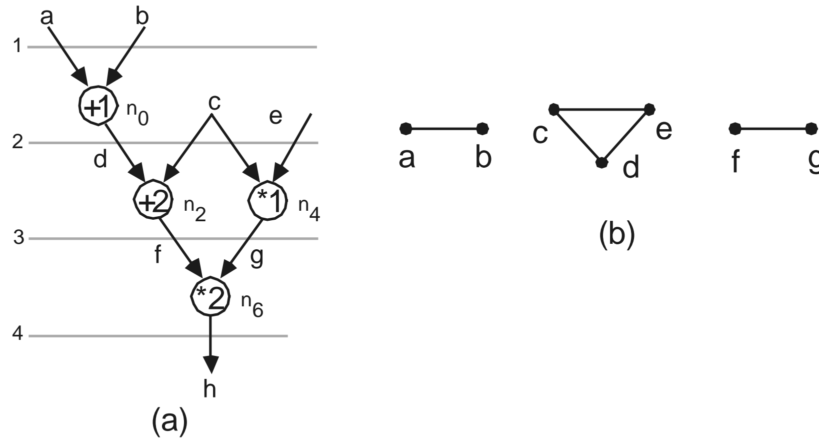


Figure. 2.32 Data flow graph and its conflict graph

Table 2.3 Ordering of variables in BITS

Variables v	a	b	c	d	e	f	g	h
Sharing degree SD(v)	1	1	2	2	1	2	2	1
MaxClique size MCS(v)	2	2	3	3	3	2	2	1
Reverse Order	7	6	1	2	5	4	3	8

2.4.1.2 BIST Test Session Minimization

All the above mentioned works focused on minimization of area overhead in BIST synthesis. For those methods, test time is not a concern in the design process, and is determined from the synthesized circuit through a post-process. In order to reduce test time in BIST, Harris and Orailoglu examined conditions which prevent concurrent testing of modules [25], [26]. They identified two types of conflicts; namely hardware conflict and software conflict. The synthesis

process is guided to avoid such conflicts for the synthesized circuit. They reported that test time for example circuits is reduced (presumably at the cost of higher area overhead) through the proposed method [25], [26].

2.4.2 ILP based Approach for High-Level BIST synthesis

Scheduling and assignment problems for a given data flow graph can be solved optimally using ILP. Advantages of ILP are that ILP can describe a new problem formally, new problem formulation can be added easily to existing ones, and it can be used to evaluate heuristic algorithms. However, an ILP method is limited to small or midrange problems, since the complexity of an ILP problem can grow exponentially. Thus, techniques to incorporate heuristics into ILP have been researched to solve problems within reasonable time. The method by Hwang et al. in [37] have used a zone scheduling technique to obtain a suboptimal solution for the scheduling problem. Rim et al [39] proposed a method which solves the assignment problem by construction, that is, the ILP formulation is solved control step by control step. Both approaches are described shortly in the following.

2.4.2.1 Suboptimal Scheduling Using ILP

Zone Scheduling (ZS) proposed by Hwang et al. divides the control steps and applies ILP to each divided zone successively. For example, consider the data flow graph shown in Fig 2.33, (which is a duplication of Fig. 2.6 (a)), and a variable interval graph in Fig. 2.35 (The variable interval graph is constructed by the same method in *Left-Edge Algorithm*.) The variable interval graph is divided into two zones as shown in Fig. 2. 34, so that the problem size of the each zone is small enough to be solved efficiently using the ILP approach. ZS solves the first zone followed the second zone by updating the time frame.

From the interval graph in Fig. 2.35, intervals of two variables {v1 and v2} are completely contained in the Zone1. The intervals of variables, {v3, v6, v7, v8, v9, v10, v11}, are cross the two zones. Processing of variables whose intervals across the zones is the major issue of the algorithm. Interested readers may refer to [37].

It is reported in [37] that when optimal ILP formulations are applied for a sixth-order elliptic band-pass filter, it takes more than 1800 seconds. When the proposed ZS is applied to the same filter, the processing time is reduced to 1 to 10 seconds. The cost for the speedup is the increase of control steps from 40 to 42, which is small.

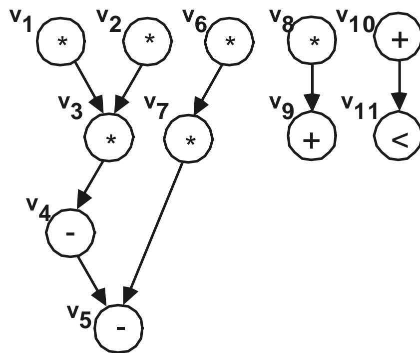


Figure 2.33 Data flow graph for Zone Scheduling

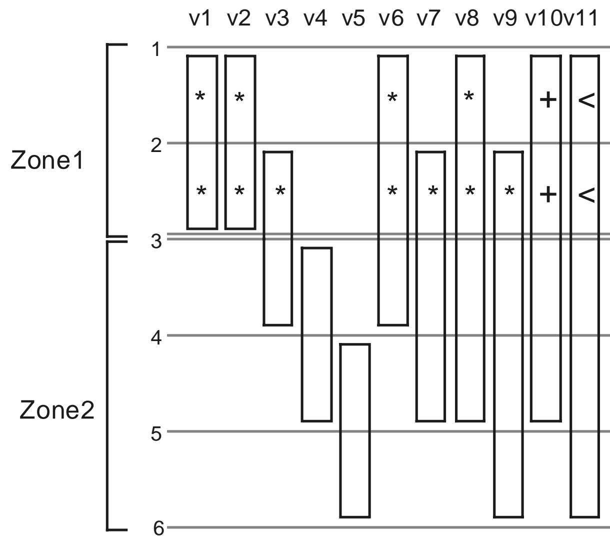


Figure 2.34 Divided data flow graph in Zone Scheduling

2.4.2.2 Suboptimal Assignment Using ILP

Rim et al. worked on ILP formulations for the assignment of modules, registers and interconnections [39]. The assignment problems in their approach are decomposed into several subproblems, one for each control-step. The processing starts from a null design or a partial design. Module, register, and interconnect assignments are performed concurrently for each control-step. They showed that an assignment problem for a single control step is identical to the transportation problem and thus solvable in polynomial time. By restricting the problem space to only one control step, ILP formulations for the assignments are simplified to result in short processing time.

According to the experimental results in [39], Rim et al.'s approach yields the same or nearly the same results in the register assignment compared with the optimal solution. However, the processing time for their approach is a small fraction of that for the optimal solution.