Signal Flow Graphs and Data Flow Graphs

Keshab K. Parhi and Yanni Chen

Abstract This chapter first introduces two types of graphical representations of digital signal processing algorithms including signal flow graph (SFG) and data flow graph (DFG). Since SFG and DFG are in general used for analyzing structural properties and exploring architectural alternatives using high-level transformations, such transformations including retiming, pipelining, unfolding and folding will then be addressed. Finally, their real-world applications to both hardware and software design will be presented.

1 Introduction

Signal processing programs differ from the traditional computing programs in the sense that these programs are referred to as non-terminating programs. In other words, input samples are processed periodically (typically with a certain iteration period or sampling period) and the tasks are repeated infinite number of times. A traditional dependence graph representation of such a program would require infinite number of nodes. Signal flow graphs and data flow graphs are powerful representations of signal processing algorithms and signal processing systems because these can represent the operations using a finite number of nodes.

K.K. Parhi (✉)
University of Minnesota, Department of Electrical and Computer Engineering, 200 Union St. S.E., Minneapolis, MN 55455, USA
e-mail: parhi@umn.edu

Y. Chen
Marvell Semiconductor Inc., 5488 Marvell Lane, Santa Clara, CA 95054, USA
e-mail: yannic@marvell.com

Signal flow graphs have been used for a long time to analyze transfer functions of linear systems. Data flow graphs are more general and are used to represent both linear and non-linear systems. These can also be used to represent multi-rate systems that contain rate-altering components such as interpolators and decimators. The z^{-1} elements in signal flow graphs and delay elements in data flow graphs describe *inter-iteration* precedence constraints, i.e., constraints between two tasks of different iterations. A simple edge without any z^{-1} or delay element represents a precedence constraint within the same iteration. These are referred as *intra-iteration* precedence constraints. Both types of precedence constraints describe the causality constraints among different operations. These causality constraints are important both in hardware designs [1] and software implementations. In a hardware implementation, the critical path time is the time needed for satisfying the causality constraints among all tasks within the same iteration, and is a lower bound on the clock period. In a software implementation, the scheduling algorithm must satisfy these causality constraints by satisfying both intra-iteration and inter-iteration constraints.

Another important property of these flow graphs is that these can be transformed into equivalent forms by high-level transformations such as pipelining, retiming, unfolding and folding. These equivalent forms have same input-output characteristics but have different sets of constraints. Pipelining and retiming can be used to reduce clock period in a circuit. These transformations can also be used as a preprocessing step for folding where the objective is to design time-multiplexed or folded architectures. Unfolding transformation can lead to lower iteration periods in software implementations as it can unravel some of the concurrency hidden in the original data flow graph. In hardware implementations, unfolding leads to parallel implementations that can increase the effective sample speed while the clock speed remains unaltered. Both pipelined and parallel implementations can be used to reduce power consumption if achieving higher speed is less important. Alternatively, in deep submicron technologies with low supply voltage, pipelining and parallel processing can be used to meet the critical path requirements.

This chapter provides a brief overview of signal flow graphs and data flow graphs, and is organized as follows. Section 2 introduces signal flow graphs, and illustrates how signal flow graphs are used for deriving transfer functions, either by using Mason's gain formula or by using a set of equations. This section also illustrates retiming and pipelining of signal flow graphs. Section 3 addresses single-rate and multi-rate data flow graphs, and illustrates how an equivalent single-rate data flow graph can be obtained from a multi-rate data flow graph. Section 4 provides an overview of unfolding and folding transformations, and their applications to hardware design. Section 5 illustrates how the causality constraints imposed by the intra-iteration and inter-iteration constraints are exploited for scheduling in software implementations. Sections 3–5 are adapted from the text book [2].

2 Signal Flow Graphs

In this section, the notation of signal flow graph (SFG) is first overviewed. Then two useful approaches, i.e., the Mason's gain formula and the equation-solving, are explained in detail to derive the corresponding transfer function for a given SFG.

2.1 Notation

Signal flow graphs have been used for the analysis, representation, and evaluation of linear digital networks, especially digital filter structures. An SFG is a collection of nodes and directed edges [3], where the nodes represent computations or tasks and a directed edge (j,k) denotes a branch originating from node j and terminating at node k. With input signal at node j and output signal at node k, the edge (j,k) denotes a linear transformation from the signal at node j to the signal at node k. An example of SFG is shown in Fig. 1, where both nodes j and k represent summing operations and the edge (j,k) denotes a unit gain transformation.

Two types of nodes exist in SFG, source nodes and sink nodes. A source node is a node with no entering edges, and is used to represent the injection of external inputs into a graph. A sink node is a node with only entering edges, and is used to extract outputs from a graph.

2.2 Transfer Function Derivation of SFG

For a given SFG, there are mainly two approaches to derive its corresponding transfer function. One is the Mason's gain formula, which provides a step-by-step method to obtain the transfer function. The other is the equation-solving approach by labeling each intermediate signal, writing down the equation for that signal with dependency on other signals, and then solving the multiple equations to

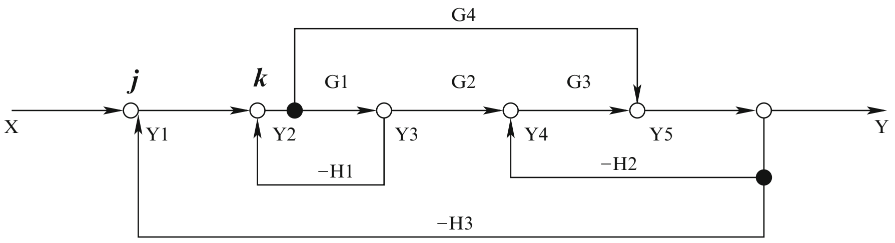


Fig. 1 An example of signal flow graph

represent the output signal *only* in terms of the input signal. Note that the variables used in the signal flow graphs and the equations correspond to frequency-domain representations of signals.

2.2.1 Mason's Gain Formula

First of all, a few useful terminologies in Mason's gain formula have to be defined related to an SFG.

- Forward path: a path that connects a source node to a sink node in which no node is traversed more than once.
- Loop: a closed path without crossing the same point more than once.
- Loop gain: the product of all the transfer functions in the loop.
- Non-touching or non-interacting loops: two loops are nontouching or noninteracting if they have no nodes in common.

In general, Mason's gain formula [4] is presented as below:

$$M = \frac{Y}{X} = \frac{\sum_{j=1}^{N} M_j \Delta_j}{\Delta} \tag{1}$$

where

- M = transfer function or gain of the system
- Y = output node
- X = input node
- N = total number of forward paths between X and Y
- Δ = determinant of the graph $= 1 - \sum$ loop gains + $\sum$ non-touching loop gains taken two at a time$-\sum$ non-touching loop gains taken three at a time + …
- M_j = gain of the jth forward path between X and Y
- Δ_j = 1-loops remaining after eliminating the jth forward path, i.e., eliminate the loops touching the jth forward path from the graph. If none of the loops remains, Δ_j = 1.

To illustrate the actual usage of Mason's gain formula, the transfer function for the example SFG shown in Fig. 1 is derived by following the steps below:

1. Find the forward paths and their corresponding gains
   Two forward paths exist in this SFG:
   $M_1 = G_1G_2G_3$ and $M_2 = G_4$
2. Find the loops and their corresponding gains
   There are four loops in this example:
   $Loop_1 = -G_1H_1$,
   $Loop_2 = -G_3H_2$,

$Loop_3 = -G_1G_2G_3H_3$,
$Loop_4 = -G_4H_3$

3. Find the Δ_j
   If we eliminate the path $M_1 = G_1G_2G_3$ from the SFG, no complete loops remain, so $\Delta_1 = 1$. Similarly, if the path $M_2 = G_4$ is eliminated from the SFG, no complete loops remain neither, so $\Delta_2 = 1$ as well.
4. Find the determinant Δ
   Only one pair of non-touching loops is in this SFG, i.e., $Loop_1$ and $Loop_2$, thus $\sum$ non-touching loop gains taken two at a time = $(-G_1H_1)(-G_3H_2)$.
   Therefore,
   $\Delta = 1 - \sum$ loop gains + $\sum$ non-touching loop gains taken two at a time
   $= 1 - (-G_1H_1 - G_3H_2 - G_1G_2G_3H_3 - G_4H_3) + (-G_1H_1)(-G_3H_2)$
   $= 1 + G_1H_1 + G_3H_2 + G_1G_2G_3H_3 + G_4H_3 + G_1G_3H_1H_2$
5. The final step is to apply the Mason's gain formula using the terms found above

$$M = \frac{Y}{X} = \frac{\sum_{j=1}^{N} M_j\Delta_j}{\Delta} = \frac{G_1G_2G_3 + G_4}{1 + G_1H_1 + G_3H_2 + G_1G_2G_3H_3 + G_4H_3 + G_1G_3H_1H_2} \tag{2}$$

2.2.2 Equations-Solving Based Transfer Function Derivation

As an alternative approach, the equations-solving based method follows different set of steps. Note that the intermediate signals have already been appropriately labeled in Fig. 1.

1. Write down the equations for each labeled signal with dependency on other signals:
   $Y_1 = X - YH_3$
   $Y_2 = Y_1 - Y_3H_1$
   $Y_3 = G_1Y_2$
   $Y_4 = G_2Y_3 - YH_2$
   $Y_5 = G_3Y_4 + G_4Y_2$
   $Y = Y_5$
2. Solve all the equations above and derive the relationship between output node Y and input node X:
   $Y = Y_5 = G_3Y_4 + G_4Y_2 = G_3(G_2Y_3 - YH_2) + G_4Y_2$
   $= G_3(G_2G_1Y_2 - YH_2) + G_4Y_2 = -G_3H_2Y + (G_1G_2G_3 + G_4)Y_2$

   Therefore,

$$Y = \frac{G_1G_2G_3 + G_4}{1 + G_3H_2}Y_2 \tag{3}$$

   Note that

$$Y_2 = Y_1 - Y_3H_1 \tag{4}$$

By substituting both Y_1 and Y_3 into (4), we obtain

$$Y_2 = X - YH_3 - Y_3H_1 = X - YH_3 - G_1H_1Y_2 \tag{5}$$

Consequently,

$$Y_2 = \frac{X - YH_3}{1 + G_1H_1} \tag{6}$$

Then by substituting (6) into (3),

$$\begin{aligned} Y &= \frac{G_1G_2G_3 + G_4}{1 + G_3H_2} \cdot \frac{X - YH_3}{1 + G_1H_1} = \frac{(G_1G_2G_3 + G_4)(X - YH_3)}{(1 + G_3H_2)(1 + G_1H_1)} \\ &= \frac{G1G2G3 + G4}{(1 + G_3H_2)(1 + G_1H_1)}X - \frac{(G_1G_2G_3 + G_4)H_3}{(1 + G_3H_2)(1 + G_1H_1)}Y \end{aligned}$$

As a result,

$$\begin{aligned} 1 + \frac{(G_1G_2G_3 + G_4)H_3}{(1 + G_3H_2)(1 + G_1H_1)}Y &= \frac{G_1G_2G_3 + G_4}{(1 + G_3H_2)(1 + G_1H_1)}X \\ \frac{Y}{X} &= \frac{G_1G_2G_3 + G_4}{(1 + G_3H_2)(1 + G_1H_1) + (G_1G_2G_3 + G_4)H_3} \end{aligned}$$

Finally, the transfer function between the output node Y and input node X is

$$M = \frac{G_1G_2G_3 + G_4}{1 + G_1H_1 + G_3H_2 + G_1G_3H_1H_2 + G_4H_3 + G_1G_2G_3H_3} \tag{7}$$

which is exactly the same as the derived transfer function using Mason's gain formula in (2).

3 Data Flow Graphs

In this section, the notation of data flow graph (DFG) is introduced and is followed by an overview of the single-rate DFG and the multi-rate DFG. How to construct an equivalent single-rate DFG from the multi-rate DFG is then explained in detail. After that, the concepts of retiming and pipelining are briefly introduced to derive equivalent DFGs.

3.1 *Notation*

In data flow graph representations, the nodes represent computations or functions or subtasks and the directed edges represent data paths (communications between nodes) and each edge has a nonnegative number of delays associated with it.

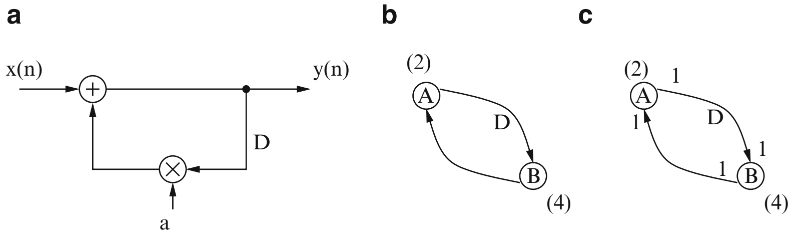


Fig. 2 (**a**) Block diagram description of the computation $y(n) = ay(n-1) + x(n)$. (**b**) Conventional DFG representation. (**c**) Synchronous DFG representation

For example, Fig. 2b is the DFG corresponding to the block diagram in Fig. 2a. Compared to the SFG in Sect. 2.1, DFG can be seen as a more generalized form of SFG in that it can effectively describe both linear single-rate and nonlinear multi-rate DSP systems.

DFG describes the data flow among subtasks or elementary computations modeled as nodes in a signal processing algorithm. Similar to SFG, various DFGs derived for one algorithm can be obtained from each other through high-level transformations. DFGs are generally used for high-level synthesis to derive concurrent implementations of DSP applications onto parallel hardware, where subtask scheduling and resource allocation are of major concerns.

3.2 *Synchronous Data Flow Graph*

A synchronous data flow graph (SDFG) is a special case of data-flow graph where the number of data samples produced or consumed by each node in each execution is specified a priori [5]. For example, Fig. 2c is an SDFG for the computation $y(n) = ay(n-1) + x(n)$, which explicitly specifies that one execution of both nodes A and B consumes one data sample and produces one output sample, which is a single-rate system.

In addition, the SDFG can describe multi-rate systems in a simple way. For example, Fig. 3a shows an SDFG representation of a multi-rate system, where nodes A, B and C are operated at different frequencies f_A, f_B and f_C, respectively. Note that A processes f_A input samples and produces $3f_A$ output samples per time unit. Node B consumes five input samples during each execution, hence consumes $5f_B$ input samples per time unit. Using the equality $3f_A = 5f_B$, we have $f_B = 3f_A/5$. Similarly, the operating rate of node C can be computed as $f_C = 2f_B/3 = 2f_A/5$. For a specified input sampling rate, the operating frequencies for nodes A, B and C can be computed. An equivalent single-rate DFG for the multi-rate DFG in Fig. 3a is shown in Fig. 3b. In contrast, this single-rate DFG contains 10 nodes and 30 edges, as compared to 3 nodes and 4 edges in the SDFG representation.

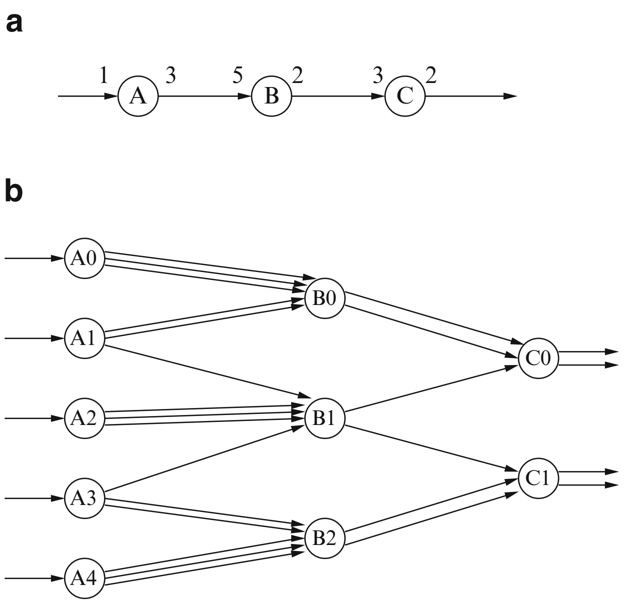
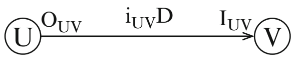


Fig. 3 Multi-rate DFG in (**a**) can be converted into single-rate DFG in (**b**), which can then be represented using linear SFG

Fig. 4 An edge $U \rightarrow V$ in a multi-rate DFG



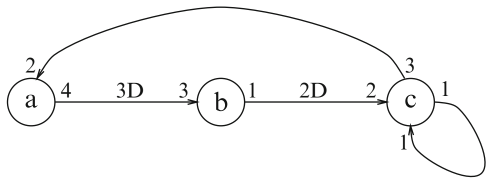
3.3 Construct an Equivalent Single-Rate DFG from the Multi-Rate DFG

By definition, each node in a single-rate DFG (SRDFG) is executed exactly once per iteration. In contrast, multi-rate DFG (MRDFG) allows each node to be executed more than once per iteration, and 2 nodes are not required to execute the same number of times in an iteration. However, SRDFG can still be used to represent multi-rate systems by first unfolding the multi-rate systems to single-rate.

An edge from the node U to the node V in an MRDFG is shown in Fig. 4, where the value O_{UV} is the number of samples produced on the edge by an invocation of the node U, the value I_{UV} is the number of samples consumed from the edge by an invocation of the node V and the value i_{UV} is the number of delays on the edge.

If the nodes U and V are invoked k_U times and k_V times in one iteration, respectively, then the number of samples produced on the edge from the node U to the node V in this iteration is $O_{UV}k_U$, and the number of samples consumed from the edge by the node V in this same iteration is $I_{UV}k_V$. Intuitively, to avoid a buildup or deficiency of samples on the edge, the number of samples produced

Fig. 5 A multi-rate DFG



in one iteration must equal the number of samples consumed in one iteration. This relationship can be described mathematically as

$$O_{UV} k_U = I_{UV} k_V \tag{8}$$

An algorithm for constructing an equivalent SRDFG from an MRDFG is described as follows:

1. For each node U in the MRDFG
2. For $k = 0$ to $k_U - 1$
3. Draw a node U^k in the SRDFG with the same computation time as U in the MRDFG
4. For each edge $U \overset{i_{UV}}{\to} V$ in the MRDFG
5. For $j = 0$ to $O_{UV} k_U$-1
6. Draw an edge $U^{j/O_{UV}} \to V^{((j+i_{UV})/I_{UV})\%k_V}$ in the SRDFG with $(j+i_{UV})/(I_{UV}k_V)$ delays

To determine how many times each node must be executed in an iteration, the set of equations found by writing (8) for each edge in the MRDFG must be solved so the number of invocations for the nodes are coprime. For example, the set of equations for the MRDFG in Fig. 5 is

$$\begin{aligned} 4k_a &= 3k_b \\ k_b &= 2k_c \\ k_c &= k_c \\ 3k_c &= 2k_a \end{aligned}$$

which has a solution of $k_a = 3$, $k_b = 4$, $k_c = 2$. Once the number of invocations of the nodes has been determined, an equivalent SRDFG can be constructed for the MRDFG. For the MRDFG in Fig. 5, the equivalent SRDFG is shown in Fig. 6.

3.4 Equivalent Data Flow Graphs

Data flow graphs can be transformed into different yet equivalent forms. Two common techniques to derive the equivalent DFGs are introduced here, one is the retiming and the other is pipelining. The transfer functions in these equivalent

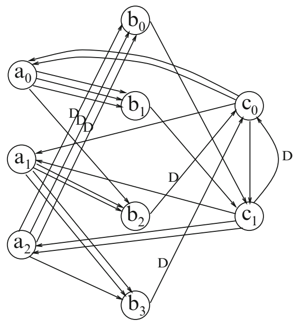


Fig. 6 An equivalent SRDFG for the MRDFG in Fig. 5

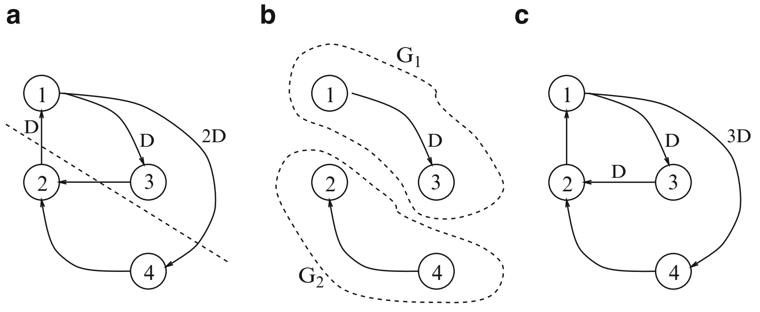


Fig. 7 (**a**) The unretimed DFG with a cutset shown as a *dashed line*. (**b**) The 2 graphs $G1$ and $G2$ formed by removing the edges in the cutset. **(c)** The retimed graph found using cutset retiming with $k = 1$

forms are either unaltered or differ only by a factor of z^{-i}, i.e., they may contain i additional delay elements. Note that both retiming and pipelining transformations can be applied to DFGs as well as SFGs in an identical manner.

3.4.1 Retiming

Retiming [6] is a transformation technique that changes the locations of delay elements in a circuit without affecting its input/output characteristics. For example, although the DFGs in Fig. 7a, c have different number of delays at different locations, they share the same input/output characteristics. Furthermore, these two DFGs can be derived from one another using retiming.

Retiming has many applications in synchronous circuit design including reducing the clock period of the circuit by reducing the computation time of the critical path, decreasing the number of registers in the circuit, reducing the dynamic power consumption of the circuit by placing the registers at the inputs of nodes with large capacitances to reduce the switching activity, and logic synthesis.

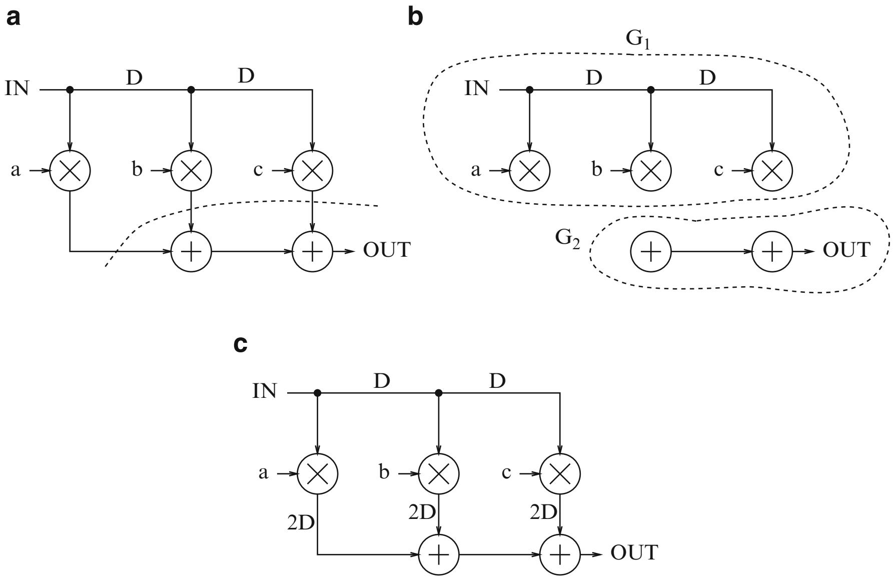


Fig. 8 (**a**) The unretimed DFG with a cutset shown as a *dashed line*. (**b**) The 2 graphs $G1$ and $G2$ formed by removing the edges in the cutset. **(c)** The graph obtained by cutset retiming with $k = 2$

Cutset retiming is a special case of retiming and it only affects the weights of the edges in the cutset, which is a set of edges that can be removed from the graph to create two disconnected subgraphs. If these two disconnected subgraphs are labeled G_1 and G_2 as depicted in Fig. 7b, then cutset retiming consists of adding k delays to each edge from G_1 and G_2, and removing k delays from each edge from G_2 to G_1. In the case of $k = 1$, the DFG in Fig. 7a can then be transformed to the DFG in Fig. 7c by using cutset retiming technique.

3.4.2 Pipelining

Pipelining is a special case of cutset retiming where there are no edges in the cutset from the subgraph G_2 to the subgraph G_1 as shown in Fig. 8b, i.e., pipelining applies to graphs without loops. These cutsets are referred to as feed-forward cutsets, where the data move in the forward direction on all the edges of the cutset. Consequently, registers can be arbitrarily placed on a feed-forward cutset without affecting the functionality of the algorithm. If two registers are inserted to each edge, the DFG in Fig. 8a can then be transformed into the DFG in Fig. 8c by applying pipelining technique. Therefore, complex retiming operations can be described by multiple simple cutest retiming or pipelining operations applied in a step-by-step manner.

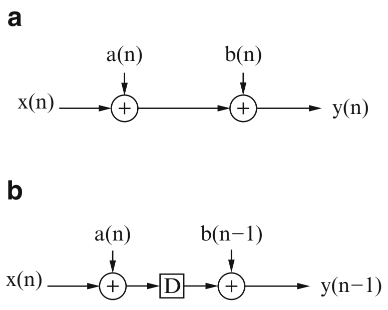


Fig. 9 (**a**) A datapath. (**b**) The 2-level pipelined structure of (**a**)

Pipelining transformation leads to a reduction in the effective critical path by introducing pipelining registers along the datapath, which can be exploited to either increase the clock speed or sample speed or to reduce power consumption at same speed. Consider the simple structure in Fig. 9a, where the computation time of the critical path is $2T_A$. Figure 9b shows the 2-level pipelined structure, where one register is placed between two adders and hence the critical path is reduced by half.

Obviously, in an M-level pipelined system the number of delay elements in any path from input to output is $(M-1)$ greater than that in the same path in the original sequential circuit. While pipelining offers the benefit of critical path reduction, its two drawbacks lie in the increase in the number of registers and the system latency, which is the time difference in the availability of the first output data in the pipelined system and the sequential system.

4 Applications to Hardware Design

In this section, two DFG-based high-level transformations applicable to practical hardware design such as field programmable gate array (FPGA) or application specific integrated circuit (ASIC) implementations are introduced, one is the unfolding transformation, and the other is the folding transformation. Examples will be given to demonstrate their usage in hardware design. For more details on how to map the decidable signal processing graphs to FPGA implementation, the reader is referred to [1].

4.1 Unfolding

Unfolding is a transformation technique that can be applied to a DSP program to create a new program describing more than one iteration of the original program. More specifically, unfolding a DSP program by the unfolding factor J creates a new program that describes J consecutive iterations of the original program. As a result, in unfolded system each delay is J-slow.

Unfolding has applications in designing high-speed and low-power VLSI architectures. One application is to unfold the program to reveal hidden concurrencies so that the program can be scheduled to a smaller iteration period, thus increasing the throughput of the implementation. Another application is to design parallel architectures at the word level and bit level from serial counterpart to increase the throughput or decrease the power consumption of the implementation.

4.1.1 The DFG Based Unfolding

Two approaches can be used to derive the J-unfolded DFG. One is to write equations for the original and the J-unfolded programs and then draw the corresponding unfolded DFG. This method could be tedious for large value of J. The other approach is to use a graph-based technique which directly unfolds the original DFG to create the DFG of the J-unfolded program without explicitly writing the equations describing the original unfolded system.

For each node U in the original DFG, there are J nodes with the same function as U in the J-unfolded DFG. Additionally, for each edge in the original DFG, there are J edges in the J-unfolded DFG. Consequently, the DFG of the J-unfolded program contains J times as many nodes and edges as the DFG of the original DFG. The following two-step algorithm could be used to construct a J-unfolded DFG:

1. For each node U in the original DFG, draw the J nodes $U_0, U_1, \dots, U_J - 1$.
2. For each edge $U \rightarrow V$ with w delays in the original DFG, draw the J edges $U_i \rightarrow V_{(i+w)\%J}$ with $\lfloor \frac{i+w}{J} \rfloor$ delays for $i = 0, 1, \dots, J-1$. Apparently, if an edge has $w < J$ delays in the original DFG, unfolding produces $J - w$ edges with no delays and w edges with 1 delay in the J-unfolded DFG.

To demonstrate the unfolding algorithm, the DFG in Fig. 10b that corresponds to the DSP algorithm in Fig. 10a will serve as an example, where the nodes A and B represent input and output, respectively, and the nodes C and D represent addition and multiplication by a, respectively. To unfold this DFG in Fig. 10b by unfolding factor of 2 to obtain the 2-unfolded DFG as shown in Fig. 10c, the two steps of the unfolding algorithm are performed:

1. The 8 nodes A_i, B_i, C_i and D_i for $i = 0, 1$ are first drawn according to the 1st step of the unfolding algorithm.
2. After these nodes have been drawn, for an edge $U \rightarrow V$ such as $D \rightarrow C$ with no delays, this step reduces to drawing the J edges $U_i \rightarrow V_i$ with no delays. Additionally, for the edges $C \rightarrow D$ with $w = 9$ delay, there are the edges $C_0 \rightarrow D_{(0+9)\%2} = D_1$ with $\lfloor (\frac{0+9}{2}) \rfloor = 4$ delays and $C_1 \rightarrow D_{(1+9)\%2} = D_0$ with $\lfloor (\frac{1+9}{2}) \rfloor = 5$ delays.

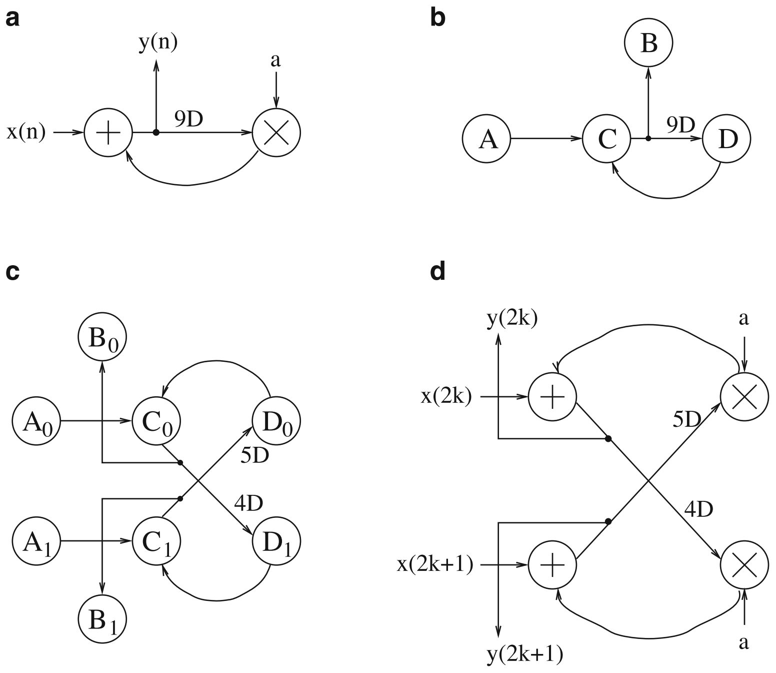


Fig. 10 (**a**) The original DSP program describing $y(n) = ay(n-9) + x(n)$ for $n = 0$ to ∞. (**b**) the DFG corresponding to DSP program in (**a**). **(c)** The 2-unfolded DFG. **(d)** The 2-unfolded DSP program describing $y(2k) = ay(2k-9) + x(2k)$ and $y(2k+1) = ay(2k-8) + x(2k+1)$ for $n = 0$ to ∞

Referring to Fig. 10c, the nodes C_0 and C_1 in the 2-unfolded DFG represent addition as the node C in the original DFG. Similarly, the nodes D_0 and D_1 in the 2-unfolded DFG represent multiplications as the node D in the original DFG. The node A in the original DFG represents the input $x(n)$. The k-th iteration of the node A_i in the unfolded DFG executes the $Jk+i$-th iteration of the node A in the original DFG for $i = 0, 1, \ldots, J-1$ and $k = 0$ to ∞. Similarly, the node B_0 corresponds to the output samples $y(2k+0)$ and the node B_1 corresponds to the output sample $y(2k+1)$. Therefore, the 2-unfolded DFG in Fig. 10c corresponds to the 2-unfolded DSP program in Fig. 10d.

4.1.2 Applications to Parallel Processing

A direct application of the general unfolding transformation is to design parallel processing architectures from serial processing architectures. At the word level, this means that word-parallel architectures can be designed from word-serial architectures. At the bit level, it means that bit-parallel and digit-serial architecture can be designed from bit-serial architectures.

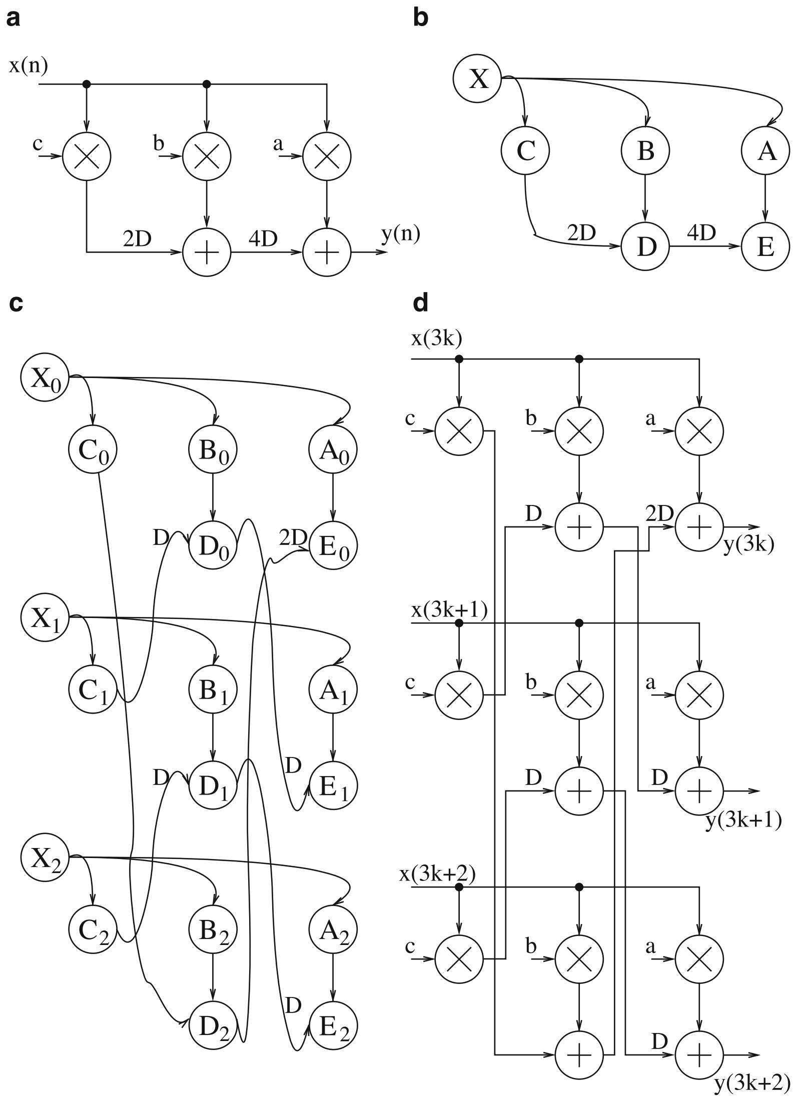


Fig. 11 (**a**) The original DSP program. (**b**) The DFG for (**a**). **(c)** The 3-unfolded DFG. **(d)** The 3-parallel DSP program

Word-Level Parallel Processing

In general, unfolding a word-serial architecture by J creates a word-parallel architecture that processes J words per clock cycle. As an example, consider the DSP program $y(n) = ax(n) + bx(n-4) + cx(n-6)$ shown in Fig. 11a. To create an architecture that can process more than 1 word per clock cycle, the first step is to

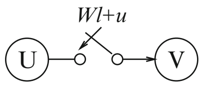


Fig. 12 A switch

draw a corresponding DFG as in Fig. 11b. The next step is to unfold the DFG to the 3-unfolded DFG as in Fig. 11c by following the steps described in 4.1.1. The final step is to draw the corresponding 3-unfolded DSP program as in Fig. 11d. The exact details are omitted here and left to the reader as an exercise.

Bit-Level Parallel Processing

Assume the wordlength of the data is W bits, the hardware implementation could have following possible architectures:

- Bit-serial processing: One bit is processed per clock cycle and hence a complete word is processed in W clock cycles.
- Bit-parallel processing: one word of W bits is processed every clock cycle.
- Digital serial processing: N bits are processed per clock cycle and a word is processed in W/N clock cycles, which is referred to as the digit size.

Most bit-serial architecture contains an edge with a switch, which corresponds to a multiplexer in hardware. Consider the edge $U \rightarrow V$ in Fig. 12. To unfold this edge with unfolding factor J, two basic assumptions are made:

- The wordlength W is a multiple of the unfolding factor J, i.e., $W = W'J$
- All edges into and out of the switch have no delays.

With these two assumptions in mind, the edge in Fig. 12 can be unfolded using the following two steps:

1. Write the switching instance as
   $Wl + u = J(W'l + \lfloor \frac{u}{J} \rfloor) + (u\%J)$
2. Draw an edge with no delays in the unfolded graph from the node $U_{u\%J}$ to the node $V_{u\%J}$, which is switched at time instance $(W'l + \lfloor \frac{u}{J} \rfloor)$.

If the switch has multiple instances, then each switching instance is treated separately. In addition, if an edge contains a switch and a positive number of delays, a dummy node can be used to reduce this problem to the case where the edge contains no delay elements.

Using these techniques for unfolding switches, bit-parallel and digit-serial architectures can be systematically designed from bit-serial architectures. This is demonstrated using the bit-serial adder in Fig. 13a. A DFG corresponding to this adder is shown in Fig. 13b. The 2-unfolded version of this DFG is shown in Fig. 13c and the architectures corresponding to this unfolded DFG is in Fig. 13d, where the 2-unfolded architecture is a digit-serial adder with digit size equal to 2.

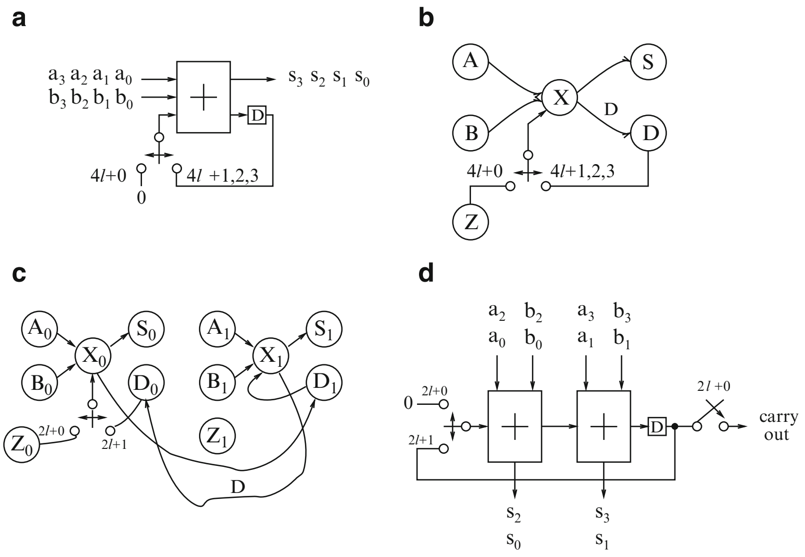


Fig. 13 (**a**) Bit-serial addition $s = a + b$ for wordlength $W = 4$. (**b**) The DFG corresponding to the bit-serial adder. (**c**) The DFG resulting from unfolding the DFG using unfolding factor of $J = 2$. (**d**) The digit-serial adder designed by unfolding the bit-serial adder using $J = 2$

For more details on how to unfold the DFG when the unfolding factor J is not the divisor of the wordlength W, the reader is referred to Sect. 5.5.2.2 in [2].

4.1.3 Infinite Unfolding of DFG

Any DFG can be unfolded by a factor of ∞. This infinitely unfolded DFG explicitly represents all intra-iteration and inter-iteration constraints. These DFGs correspond to dependence graphs (DGs) of traditional terminating programs. The DG or the infinitely unfolded DFG cannot contain any delay elements. Figure 14a shows a DFG and Fig. 14b shows the corresponding infinitely unfolded DFG.

4.2 Folding

The folding transformation is used to systematically determine the control circuits in DSP architectures where multiple algorithm operations such as additions are time-multiplexed to a single functional unit such as pipelined adder. By executing multiple algorithm operations on a single functional unit, the number of functional units in the implementation is reduced, resulting in an integrated circuit with low

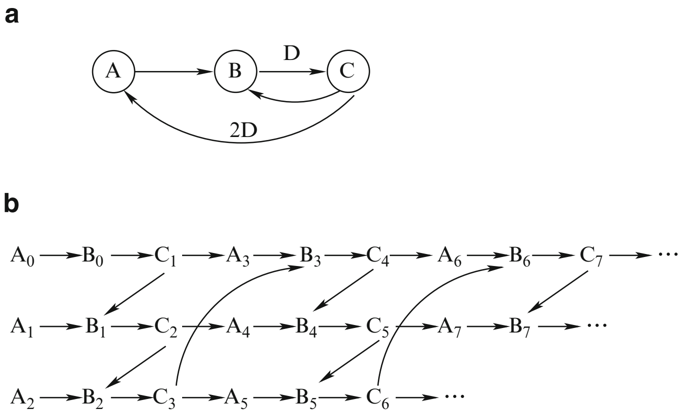


Fig. 14 (**a**) The original DFG. (**b**) The infinitely unfolded DFG

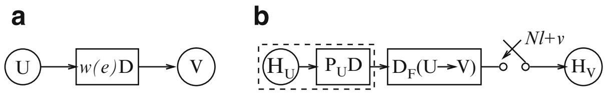


Fig. 15 (**a**) An edge $U \overset{e}{\rightarrow} V$ with $w(e)$ delays. (**b**) The corresponding folded datapath. The data begin at the functional unit H_U which has P_U pipelining stage, pass through $D_F(U \overset{e}{\rightarrow} V)$ delays, and are switched into the functional unit H_V at the time instances $Nl+v$

silicon area. In general, folding can be used to reduce the number of hardware functional units by a factor of N at the expense of increasing the computation time by a factor of N.

Consider the edge e connecting the nodes U and V with $w(e)$ delays, as shown in Fig. 15a. Let the executions of the l-th iteration of the nodes U and V be scheduled at the time units $Nl+u$ and $Nl+v$, respectively, where u and v are folding orders of the nodes U and V that satisfy $0 \le u, v \le N-1$ and N is the folding factor. The folding order of a node is the time partition to which the node is scheduled to execute in hardware. The functional units that execute the nodes U and V are denoted as H_U and H_V, respectively. If H_U is pipelined by P_U stages, then the result of the l-th iteration of the node U is available at the time unit $Nl+u+P_U$. Since the edge $U \overset{e}{\rightarrow} V$ has $w(e)$ delays, the result of the l-th iteration of the node U is used by the $(l+w(e))$-th iteration of the node V, which is executed at $N(l+w(e))+v$. Therefore, the result must be stored for

$$D_F(U \overset{e}{\rightarrow} V) = [N(l+w(e))+v] - [Nl+P_U+u] = Nw(e) - P_U + v - u \quad (9)$$

time units, which is independent of the iteration number l. The edge $U \overset{e}{\rightarrow} V$ is implemented as a path from H_U to H_V in the architecture with $D_F(U \overset{e}{\rightarrow} V)$ delays, and data on this path are input to H_V at $Nl+v$, as in Fig. 15b.

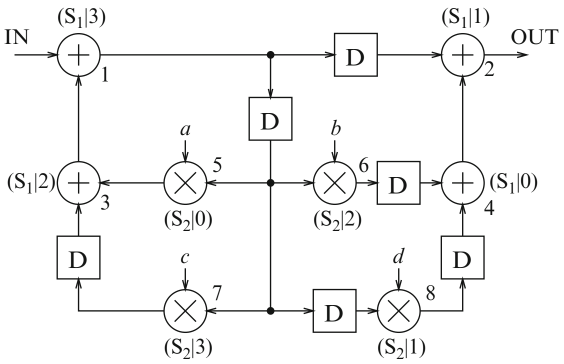


Fig. 16 The retimed biquad filter with valid folding sets assigned

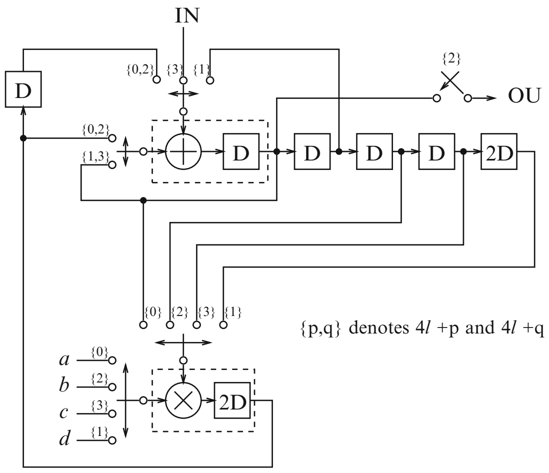


Fig. 17 The folded biquad filter using the folding sets given in Fig. 16

A folding set is an ordered set of operations executed by the same functional unit. Each folding set contains N entries, some of which may be null operations. The operation in the j-th position within the folding set, where j goes from 0 to $N-1$, is executed by the functional unit during the time partition j. The biquad filter example shown in Fig. 16 is folded with folding factor of $N=4$ and the folding sets shown in the figure can be written as $S_1 = \{4,2,3,1\}$ and $S_2 = \{5,8,6,7\}$, where the folding set S_1 and S_2 contains only addition operations using the same hardware adder and multiplication operation using the same hardware multiplier, respectively. To obtain the folded architecture as shown in Fig. 17 corresponding to the DFG in Fig. 16, the folding equations for each of the 11 edges are written as below:

$$D_F(1 \to 2) = 4(1) - 1 + 1 - 3 = 1$$

$$D_F(1 \to 5) = 4(1) - 1 + 0 - 3 = 0$$

$$D_F(1 \to 6) = 4(1) - 1 + 2 - 3 = 2$$

$$D_F(1 \to 7) = 4(1) - 1 + 3 - 3 = 3$$
$$D_F(1 \to 8) = 4(2) - 1 + 1 - 3 = 5$$
$$D_F(3 \to 1) = 4(0) - 1 + 3 - 2 = 0$$
$$D_F(4 \to 2) = 4(0) - 1 + 1 - 0 = 0$$
$$D_F(5 \to 3) = 4(0) - 2 + 2 - 0 = 0$$
$$D_F(6 \to 4) = 4(1) - 2 + 0 - 2 = 0$$
$$D_F(7 \to 3) = 4(1) - 2 + 2 - 3 = 1$$
$$D_F(8 \to 4) = 4(1) - 2 + 0 - 1 = 1$$

For a folded system to be realizable, $D_F(U \xrightarrow{e} V) \geq 0$ must hold for all of the edges in the DFG. Once valid folding sets have been assigned, retiming can be used to either satisfy this property or determine that the folding sets are infeasible.

In general, the original DFG and the N-unfolded version of the folded DFG are retimed and/or pipelined versions of each other. Furthermore, an arbitrary DFG can be unfolded by factor N and the unfolded DFG can be folded with many possible folding sets to generate a family of architectures. In order to obtain the original DFG from the unfolded DFG via folding transformation, an appropriate folding set has to be chosen.

5 Applications to Software Design

In this section, the precedence constraints in DFG will first be introduced followed by the definition of critical path and the iteration bound. A DFG based scheduling algorithm is then explained in detail.

5.1 Intra-iteration and Inter-iteration Precedence Constraints

The DFG captures the data-driven property of DSP algorithms where any node can fire (perform its computation) whenever all the input data are available. This implies that a node with no input edges can fire at any time. Thus many nodes can be fired simultaneously, leading to concurrency. Conversely, a node with multiple input edges can only fire after all its precedent nodes have fired. The latter case imposes the precedence constraints between two nodes described by each edge. This precedence constraint is an *intra-iteration* precedence constraint if the edge has no delay elements or an *inter-iteration* precedence constraint if the edge has one or more delays. Together, the intra-iteration and inter-iteration precedence constraints specify the order in which the nodes in the DFG can be executed.

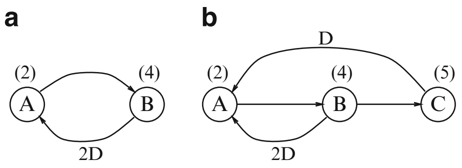


Fig. 18 (**a**) A DFG with one loop that has a loop bound of $6/2 = 3$ u.t. The iteration bound for this DFG is 3 u.t. (**b**) A DFG with iteration bound $T_\infty = max\{6/2, 11/1\} = 11$ u.t.

For example, the edge from node A to node B in Fig. 2b enforces the inter-iteration precedence constraint, which states that the execution of the k-th iteration of A must be completed before the $(k+1)$-th iteration of B. On the other hand, the edge from B to A enforces the intra-iteration precedence constraint, which states that the k-th iteration of B must be executed before the k-th iteration of A.

5.2 *Definition of Critical Path and Iteration Bound*

The critical path of a DFG is defined to be the path with the longest computation time among all paths that contain no delay elements. The critical path in the DFG in Fig. 18a is the path $A \to B$, which requires 6 u.t. Since the critical path is the longest path for combinational rippling in the DFG, the computation time of the critical path is the minimum computation time for one iteration of the DFG, which is the execution of each node in the DFG exactly once.

A loop is a directed path that begins and ends at the same node and the amount of time required to execute a loop can be determined from the precedence relation described by the edges of the DFG. According to these precedence constraints, iteration k of the loop consists of the sequential execution of A_k and B_k. Given that the execution times of nodes A and B are 2 and 4 u.t., respectively, one iteration of the loop requires 6 u.t. This is the loop bound, which represents the lower bound on the loop computation time. Formally, the loop bound of the l-th loop is defined as $\frac{t_l}{w_l}$, where t_l is the loop computation time and w_l is the number of the delays in the loop. As a result, the loop bound for the loop in Fig. 18a is $6/2 = 3$ u.t.

As another example, the DFG in Fig. 18b contains two loops, namely, the loops $l_1 = A \to B \to A$ and $l_2 = A \to B \to C \to A$. Therefore, the loop bounds for l_1 and l_2 are 6/2 = 3 u.t. and 11/1 = 11 u.t., respectively. The loop with the maximum loop bound is called the critical loop and its corresponding loop bound is the iteration bound of the DSP program, which is the lower bound on the iteration or sample period of the DSP program regardless of the amount of the computing resources available. Formally, the iteration bound is defined as

$$T_\infty = max_{l \in L}\{\frac{t_l}{w_l}\} \tag{10}$$

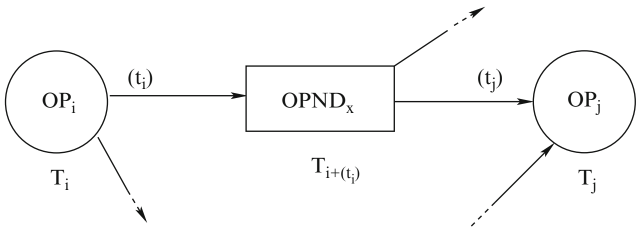


Fig. 19 A pair of operators showing the timing information

where L is the set of the loops in the DFG, t_l and w_l are the computation time and the number of delays of the loop l, respectively.

To compute the iteration bound of a DFG by locating all the loops and directly compute T_∞ by using (10) is rather straightforward. However, the number of loops in a DFG grows exponentially with the number of nodes, and therefore polynomial-time algorithms are desired for computing the iteration bound. Examples of polynomial-time algorithms include longest path matrix algorithm [7] and the minimum cycle mean algorithm [8].

5.3 Scheduling

DFGs are generally used for high-level synthesis to derive concurrent implementations of DSP applications onto parallel hardware, where a major concern is subtask scheduling which determines when and in which hardware units nodes can be executed.

Scheduling algorithm consists of assigning a scheduling time to every operator in an architecture, where the time represents when the operation will normally take place. Each operator must be scheduled at a time after all of its inputs become available. Consequently, the scheduling problem can be formulated as a linear programming problem. Furthermore, because all scheduled time of the operators must be integers, the scheduling algorithm must find the optimal integer solution to this problem.

5.3.1 The Scheduling Algorithm

Consider a pair of operators joined by an edge shown in Fig. 19. One of the outputs produced by the operator OP_i is the operand $OPND_x$, which in turn is the input to the operator OP_j. The scheduled times of operators OP_i and OP_j are denoted by T_i and T_j, respectively. In addition, the timing specifications of the relevant output and input ports of OP_i and OP_j are denoted by t_i and t_j, respectively. Since OP_i is scheduled at time T_i, the output $OPND_x$ will become available at time $T_i + t_i$.

Further, the same operand will be required as an input to the operator OP_j at time $T_i + t_j$. By the requirement that the operand can not be used by operator OP_j before it is produced by operator OP_i, the following inequality can be derived:

$$T_j + t_j \geq T_i + t_i \tag{11}$$

Such an inequality holds for each pair of operands joined by an edge in the DFG. In these inequalities, T_i and T_j are the unknowns, where the value $t_i - t_j$ is a known constant. A solution to the set of inequalities can be determined by using common techniques for solving linear programming problems. Once a solution is found to this set of inequalities, the circuit may be correctly synchronized by inserting a delay equal to $T_j - T_i - (t_i - t_j)$ clock cycles between the operators OP_i and OP_j.

In general, many solutions exist to satisfy the set of inequalities for a given architecture and hence there are many ways to synchronize the circuit. The goal of optimal scheduling is to generate a solution that provides the minimal cost, where cost is defined to be the total number of shift-register delays required to properly synchronize the circuit. If a linear cost function can be defined, the minimum cost problem can be easily formulated as a linear programming problem.

5.3.2 Minimum Cost Solution

Minimizing the total number of synchronization delays required for reach edge between functional units of a circuit is not sufficient. Note that there exists the possibility of multiple fanout from any functional unit. Therefore, the delays from a multiple fanout output should be allocated sequentially instead of in parallel. Consider a simple case where an output of some operator OP_o is used as an input to three other operators, OP_A, OP_B and OP_C as shown in Fig. 20a with delays of 10, 12 and 25 clock cycles. respectively. The total number of delays is equal to $10 + 12 + 25 = 47$. An alternative sequential arrangement of the delays is shown in Fig. 20b, where the total number of delays is 25, which is the length of the longest delay. Therefore, the total number of delays that need to be allocated to any node in the circuit is equal to the maximum delay that must be allocated.

Let D_x represents the maximum delay that must be allocated to an operand $OPND_x$ of width w_x. A total cost function to be minimized can now be defined:

$$Cost = \sum_x D_x w_x \tag{12}$$

where the sum is over each operand node in the circuit. For each node as shown in Fig. 20b, there exists a constraint equivalent to (11),

$$T_j - T_i \geq t_i - t_j \tag{13}$$

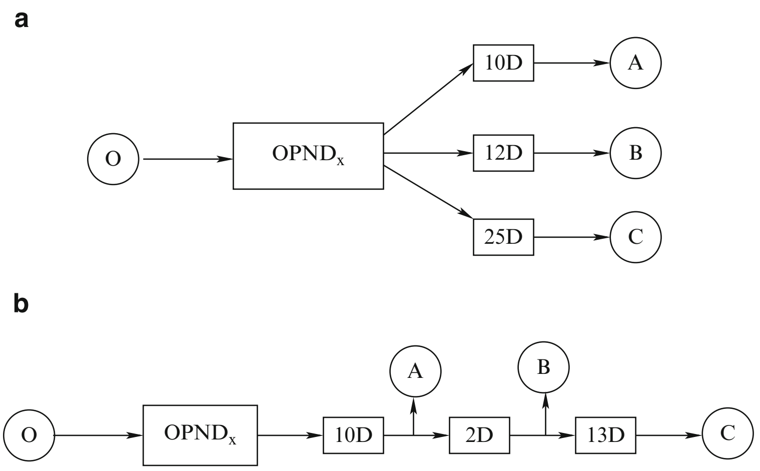


Fig. 20 (**a**) Operator O with a fanout of three and no delay sharing. (**b**) Operator O with a fanout of three and delay sharing

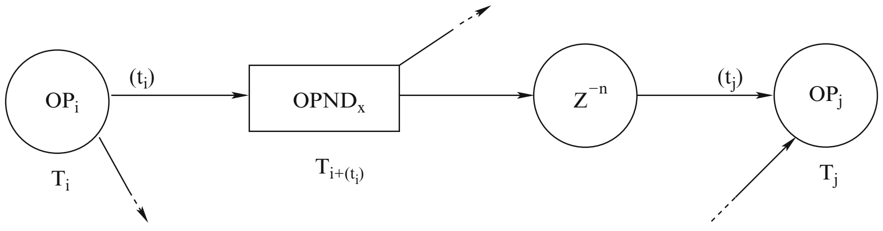


Fig. 21 General code to be scheduled including $z^{-1}operators$

In addition, the maximum delay on operand OPNDx from OP_i to OP_j is less than the maximum delay D_x, which is described by the constraint:

$$T_j - T_i - (t_i - t_j) \leq D_x \tag{14}$$

These two constraints along with the cost function that will be minimized as in (12), describe a linear programming problem capable of providing the minimum cost scheduling.

5.3.3 Scheduling of Edges with Delays

The scheduling algorithm will generate optimal solutions for DFGs consisting of delay-free edges. To handle edges that contain delays, a preferable method is to incorporate the word delays right into the linear programming solutions, which is achieved by slightly modifying equations to take into account the presence of z^{-1} operators. Specifically, Fig. 21 shows a general situation in which the output of some operator OP_i undergoes a z^{-n} transformation before being used as an input to the operator OP_j. The modified equations describing these scheduling constraints thus become:

$$T_j - T_i \geq t_i - t_j - nW \tag{15}$$

and

$$T_j - T_i - (t_i - t_j) + nW \leq D_x \tag{16}$$

where W is the number of clock cycles in a word or wordlength. In this case, $T_j - T_i - (t_i - t_j) + nW$ is the delay applied to the connection shown in the diagram, and D_x is the maximum delay applied to this variable.

6 Conclusions

This chapter has introduced the signal flow graphs and data flow graphs. Several transformations such as pipelining, retiming, unfolding and folding have been reviewed, and applications of these transformations on signal flow graphs and data flow graphs have been demonstrated for both hardware and software implementations. It is important to note that any DFG that can be pipelined can be operated in a parallel manner, and *vice versa*. Any transformation that can improve performance in a hardware system can also improve the performance of a software system. Thus, preprocessing of the DFGs and SFGs by these high-level transformations can play a major role in the system performance.

Acknowledgements Many parts of the text and figures in this chapter are taken from the text book in [2]. These have been reprinted with permission of John Wiley & Sons, Inc. The authors are grateful to John Wiley & Sons, Inc., for permitting the authors to use these figures and parts of the text from [2]. They are also grateful to George Telecki, associate publisher at Wiley for his help in this regard.

References

1. Woods, R.: Mapping decidable signal processing graphs into FPGA implementations. In: S.S. Bhattacharyya, E.F. Deprettere, R. Leupers, J. Takala (eds.) Handbook of Signal Processing Systems, second edn. Springer (2013)
2. Parhi, K.: VLSI Digital Signal Processing Systems, Design and Implementation. John Wiley & Sons, New York (1999)
3. Crochiere, R., Oppenheim, A.: Analysis of linear digital networks. Proc. IEEE **64**(4), 581–595 (1975)
4. Bolton, W.: Newnes Control Engineering Pocketbook. Newnes, Oxford, UK (1998)
5. Lee, E., Messerschmitt, D.: Synchronous data flow. Proc. IEEE, special issue on hardware and software for digital signal processing **75**(9), 1235–1245 (1987)

6. Leiserson, C., Rose, F., Saxe, J.: Optimizing synchronous circuitry by retiming. In: Third Caltech Conference on VLSI, pp. 87–116 (1983)
7. Gerez, S., Heemstra de Groot, S., Herrmann, O.: A polynomial-time algorithm for the computation of the iteration-period bound in recursive data flow graphs. IEEE Trans. on Circuits and Systems-I: Fundamental Theory and Applications **39**(1), 49–52 (1992)
8. Ito, K., Parhi, K.: Determining the minimum iteration period of an algorithm. Journal of VLSI Signal Processing **11**(3), 229–244 (1995)