8

Architecture Derivation for FPGA-based DSP Systems

8.1 Introduction

The technology review in Chapter 4 and the detailed circuit implementation material in Chapter 5 clearly demonstrated the need to develop a circuit architecture when implementing DSP algorithms in FPGA technology. The circuit architecture allows the performance needs of the application to be captured effectively. One optimization is to implement the high levels of parallelism available in FIR filters directly in hardware, thereby allowing a performance increase to be achieved by replicating the functionality in FPGA hardware. In addition, it is possible to pipeline the SFG or DFG heavily to exploit the plethora of available registers in FPGA; this assumes that the increased latency in terms of clock cycles, incurred as a result of the pipelining (admittedly at a smaller clock period), can be tolerated. It is clear that optimizations made at the hardware level can have direct cost implications for the resulting design. Both of these aspects can be captured in the circuit architecture.

In Chapter 5 it was shown how this trade-off is much easier to explore in "fixed architectural" platforms such as microprocessors, DSP processors or even reconfigurable processors, as appropriate tools can be or have been developed to map the algorithmic requirements efficiently onto the available hardware. As already discussed, the main attraction of using FPGAs is that the available hardware can be developed to meet the specific needs of the algorithm. However, this negates the use of efficient compiler tools as, in effect, the architectural "goalposts" have moved as the architecture is created on demand! This fact was highlighted in Chapter 7 which covered some of the high-level tools that are being developed either commercially or in universities and research labs. Thus, it is typical that a range of architecture solutions are explored with cost factors that are computed at a high level of abstraction.

In this chapter, we will explore the direct mapping of simple DSP systems or, more precisely, DSP components such as FIR or IIR filters, adaptive filters, etc. as these will now form part of more complex systems such as beamformers and echo cancelers. The key aim is to investigate how changes applied to SFG representations can impact the FPGA realizations of such functions, allowing the reader to quickly work in the SFG domain rather than in the circuit architecture domain. This trend will become

FPGA-based Implementation of Signal Processing Systems,
Second Edition. Roger Woods, John McAllister, Gaye Lightbody and Ying Yi.

increasingly prevalent throughout the book as we attempt to move to a higher-level representation. The later chapters demonstrate how higher levels of abstraction can be employed to allow additional performance improvements by considering system-level implications.

Section 8.2 looks at the DSP characteristics and gives some indication of how these map to FPGA. The various representations of DSP systems are outlined in Section 8.3. Given that a key aspect of FPGA architecture is distributed memory, efficient pipelining is a key optimization and so is explored in detail in Section 8.4. The chapter then goes on to explore how the levels of parallelism can be adjusted in the implementation in order to achieve the necessary speed at both lower or higher area costs; duplicating the hardware is formally known as "unfolding" and sharing the available hardware is called "folding," and both of these techniques are explored in Section 8.5. Throughout the chapter, the techniques are applied to FIR, IIR and lattice filters and explored using the Xilinx Virtex-5 FPGA family. This material relies heavily on the excellent text by Parhi (1999).

8.2 DSP Algorithm Characteristics

By their very nature, DSP algorithms tend to be used in applications where there is a demand to process high volumes of information. As highlighted in Chapter 2, the sampling rates can range from kilohertz, as in speech environments, right through to megahertz, as in the case of image processing applications. It is vital to clearly define a number of parameters with regard to system implementation of DSP systems:

- *Sampling rate* can be defined as the rate at which we need to process the DSP signal samples for the system or algorithm under consideration. For example, in a speech application, the maximum bandwidth of speech is typically judged to be 4 kHz, and the Nyquist rate indicates a sampling rate of 8 kHz.
- *Throughout rate* (TR) defines the rate at which data samples are processed. In some cases, the aim of DSP system implementation is to match the throughput and sampling rates, but in systems with lower sampling rates (speech and audio), this would result in underutilization of the processing hardware. For example, speech sampling rates are 8 kHz, but the speeds of many DSP processors are of the order of hundreds of megahertz. In these cases there is usually a need to perform a large number of computations per second, which means that the throughput rate can be several times (say, p) the sampling rate. In cases where the throughput is high and the computational needs are moderate, there is the possibility of reusing the hardware, say p times. This is a mapping that would need to be applied in FPGA implementation.
- *Clock rate* defines the operating speed of the system implementation. It used to be a performance figure quoted by computing companies, although it is acknowledged that memory size, organization and usage can be more critical in determining performance. In DSP systems, a simple perusal of DSP and FPGA data sheets indicates that clock rates of FPGA families are 550–600 MHz, whereas TI's TMS320C6678 DSP family can run up to 1.25 GHz. It would appear that the DSP processor is faster than the FPGA, but it is the amount of computation that can be performed in a single cycle that it is important (Altera 2014). This is a major factor in determining the throughput rate, which is a much more accurate estimate of performance, but is of course application-dependent.

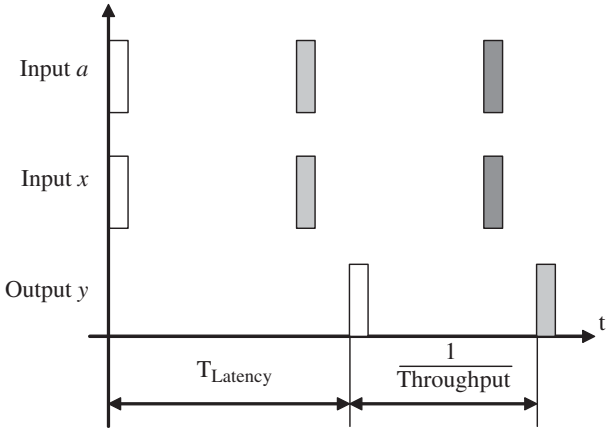


Figure 8.1 Latency and throughput rate relationship for system $y(n) = ax(n)$

Thus, it is clear that we need to design systems ultimately for throughput and therefore sampling rate, as a first measure of performance. This relies heavily on how efficiently we can develop the circuit architecture. As Chapter 5 clearly indicated, this comes from harnessing effectively the underlying hardware resources to meet the performance requirements. In ASIC applications the user can define the processing resources to achieve this, but in FPGAs the processing resources are restrictive in terms of their number and type (e.g. dedicated DSP blocks, scalable adder structures, LUT resources, memory resource (distributed RAM, LUT RAM, registers)) and interconnection (e.g. high-speed Rocket IO, various forms of programmable interconnect). The aim is to match these resources to the computational needs, which we will do here based initially on performance and then trading off area, if throughput is exceeded.

In DSP processors, the fixed nature of the architecture is such that efficient DSP compilers have evolved to allow high-level or C language algorithmic descriptions to be compiled, assembled and implemented onto the platform. Thus, the implementation target is to investigate if the processing resources will allow one iteration of the algorithm to be computed at the required sampling rate. This is done by allocating the processing to the available resources and scheduling the computation in such a way as to achieve the required sampling rate. In effect, this involves reusing the available hardware, but we intend not to think about the process in these terms. In a FPGA implementation, an immediate design consideration is to consider how many times we can reuse the hardware and whether this allows us to achieve the sampling rate. This change of emphasis in creating the hardware resource to match the performance requirements is the reason for a key focus of the chapter.

8.2.1 Further Characterization

Latency is the time required to produce the output, $y(n)$ for the corresponding $x(n)$ input. At first glance, this would appear to equate to the throughput rate, but as the computation of $y(n) = ax(n)$ shown in Figure 8.2 clearly demonstrates, this is not the case, particularly if pipelining is applied. In Figure 8.2, the circuit could have three pipeline stages and thus will produce a first output after three clock cycles, hence known as the

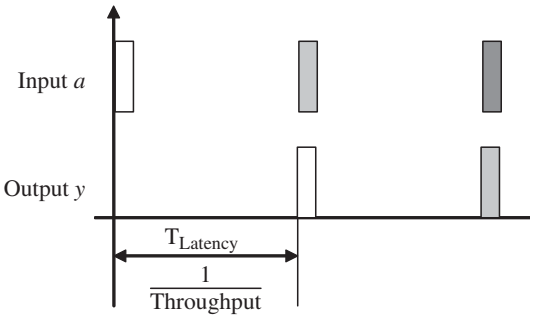


Figure 8.2 Latency and throughput rate relationship for system $y(n) = ay(n-1)$

latency; thereafter, it will produce an output once every cycle which is the throughput rate.

The situation is complicated further in systems with feedback loops. For example, consider the simple recursion $y(n) = ay(n-1)$ shown in Figure 8.2. The present output $y(n)$ is dependent on the previous output $y(n-1)$, and thus the latency determines the throughput rate. This means now that if it takes three clock cycles to produce the first output, then we have to wait three clock cycles for the circuit to produce each output and, for that matter, enter every input. Thus it is clear that any technique such as pipelining that alters both the throughput and the latency must be considered carefully, when deriving the circuit architectures for different algorithms.

There are a number of optimizations that can be carried out in FPGA implementations to perform the required computation, as listed below. Whilst it could be argued that parallelism is naturally available in the algorithmic description and not an optimization, the main definitions here focus on exploitation within FPGA realization; a serial processor implementation does not necessarily exploit this level of parallelism.

Parallelism can either naturally exist in the algorithm description or can be introduced by organizing the computation to allow a parallel implementation. In Figure 8.3, we can

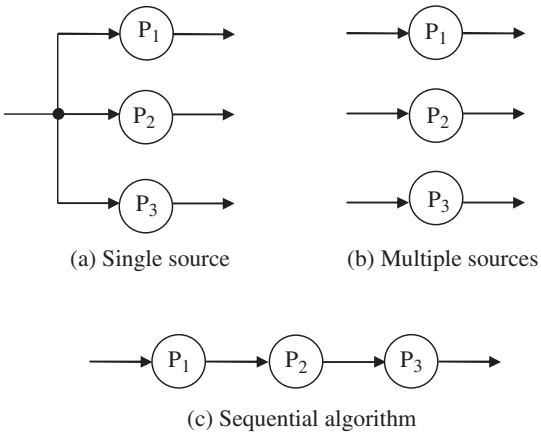
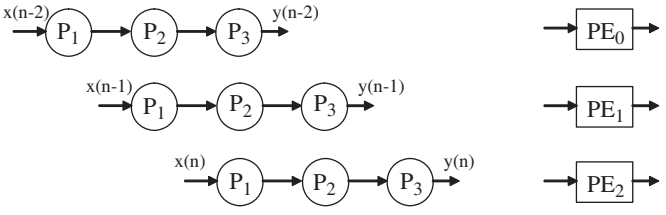
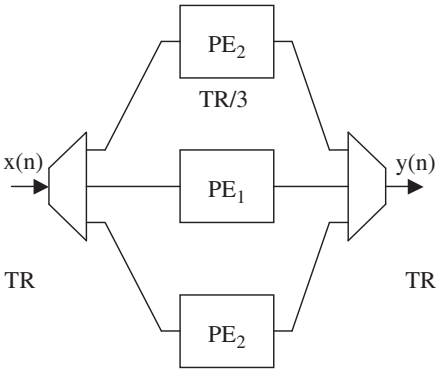


Figure 8.3 Algorithms realizations using three processes P_1, P_2 and P_3



(a) Demonstration of interleaving



(b) Interleaving realization

Figure 8.4 Interleaving example

realize processes P_1, P_2 and P_3 as three separate processors, PE_1, PE_2 and PE_3, in all three cases. In Figure 8.3(a) the processes are driven from a single source, in Figure 8.3(b) they are from separate sources and in Figure 8.3(c) they are organized sequentially. In the latter case, the processing is inefficient as only one processor will be used at any one time, but it is shown here for completeness.

Interleaving can be employed to speed up computation, by sharing a number of processors to compute iterations of the algorithm in parallel, as illustrated in Figure 8.4 for the sequential algorithm of Figure 8.3(c). In this case, the three processors PE_1, PE_2 and PE_3 perform three iterations of the algorithm in parallel and each row of the outlined computation is mapped to an individual processor, PE_1, PE_2 and PE_3.

Pipelining is effectively another form of concurrency where processes are carried out on separate pieces of data, but at the same time, as illustrated in Figure 8.5. In this case, the three processes PE_1, PE_2 and PE_3 are performed at the same time, but on different

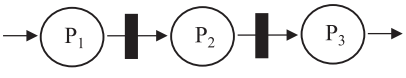
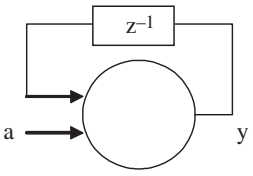
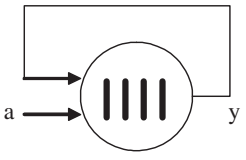


Figure 8.5 Example of pipelining



(a) Original recursive computation (clock rate, fc, and throughput rate, f)



(b) Pipelined version (clock rate, $4\,fc$, and throughput rate, f)

Figure 8.6 Pipelining of recursive computations $y(n) = ay(n-1)$

iterations of the algorithm. Thus the throughput is now given as t_{PE_1} or t_{PE_2} or t_{PE_3} rather than $t_{PE_1} + t_{PE_2} + t_{PE_3}$ as for Figure 8.3(c). However, the application of pipelining is limited for some recursive functions such as the computation $y(n) = ay(n-1)$ given in Figure 8.6. As demonstrated in Figure 8.6(a), the original processor realization would have resulted in an implementation with a clock rate f_c and throughput rate f. Application of four levels of pipelining, as illustrated in Figure 8.6(b), results in an implementation that can be clocked four times faster, but since the next iteration depends on the present output, it will have to wait four clock cycles. This gives a throughput rate of once every four cycles, indicating a nil gain in performance. Indeed, the flip-flop setup and hold times now form a much larger fraction of the critical path and the performance would actually have been degraded in real terms.

It is clear then that these optimizations are not a straightforward application of one technique. For example, it may be possible to employ parallel processing in the final FPGA realization and then employ pipelining within each of the processors. In Figure 8.6(b), pipelining did not give a speed increase, but now four iterations of the algorithm can be interleaved, thereby achieving a fourfold improvement. It is clear that there are a number of choices available to the designer to achieve the required throughput requirements with minimal area requirements such as sequential versus parallel, trade-off between parallelism/pipelining and efficient use of hardware sharing. The focus of this chapter is on demonstrating how the designer can start to explore these trade-offs in an algorithmic representation, by starting with an SFG or DFG description and then carrying out manipulations with the aim of achieving improved performance.

8.3 DSP Algorithm Representations

There are a number of ways of representing DSP algorithms, ranging from mathematical descriptions, to block diagrams, right through to HDL descriptions of implementations.

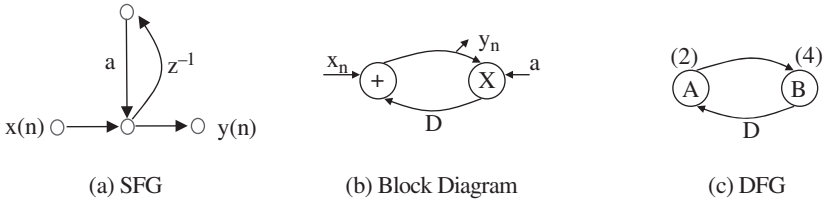


Figure 8.7 Various representations of simple DSP recursion $y(n) = ay(n-1) + x(n)$

In this chapter, we concentrate on an SFG and DFG representation as a starting point for exploring some of the optimizations briefly outlined above. For this reason, it is important to provide more detail on SFG and DFG representations.

8.3.1 SFG Descriptions

The classical description of a DSP system is typically achieved using an SFG representation which is a collection of nodes and directed edges, where a directed edge (j, k) denotes a linear transform from the signal at node j to the signal at node k. Edges are usually restricted to multiplier, adder or delay elements. The classical SFG of the expression $y(n) = ay(n-1) + x(n)$ is given in Figure 8.7(a), while the block diagram is given in Figure 8.7(b). The DFG representation shown in Figure 8.7(c) is often a more useful representation for the retiming optimizations applied later in the chapter.

8.3.2 DFG Descriptions

In DFGs, nodes represent computations or functions and directed edges represent data paths with non-negative numbers associated with them. Dataflow captures the data-driven property of DSP algorithms where the node can fire (perform its computation) when all the input data are available; this creates precedence constraints (Parhi 1999). There is an *intra*-iteration constraint if an edge has no delay; in other words, the order of firing is dictated by DFG arrow direction. The *inter*-iteration constraint applies if the edge has one or more delays and will be translated into a digital delay or register when implemented.

A more practical implementation can be considered for a three-tap FIR filter configuration. The SFG representation is given in Figure 8.8. One of the transformations that can be applied to SFG representation is that of transposition. This is carried out by reversing the directions in all edges, exchanging input and output nodes whilst keeping edge gains or edge delays unchanged as shown in Figure 8.8(b). The reorganized version is shown in Figure 8.8(c). The main difference is that the dataflow of the $x(n)$ input has been reversed without causing any functional change to the resulting SFG. It will be seen later that the SFG of Figure 8.8(c) is a more appropriate structure to which to apply pipelining.

The dataflow representation of the SFG of Figure 8.8(b) is shown in Figure 8.9. In Figure 8.9 the multipliers labeled as a_0, a_1 and a_2 represent pipelined multipliers with two levels of pipeline stages. The adders labeled as A_0 and A_1 represent pipelined adders

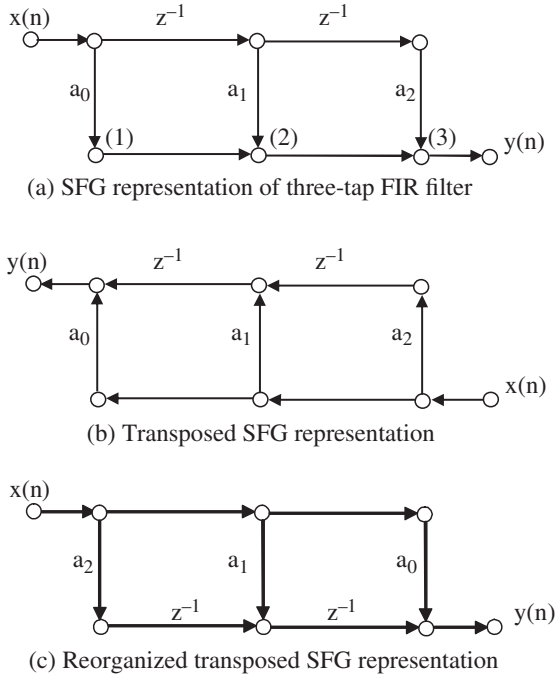


Figure 8.8 SFG representation of three-tap FIR filter

with a pipeline stage of 1. The D labels represent single registers with size equal to the wordlength (not indicated on the DFG representation). In this way, the dataflow description gives a good indication of the hardware realization; it is clear that it is largely an issue of developing the appropriate DFG representation for the performance requirements needed. In the case of pipelined architecture, this is largely a case of applying suitable retiming methodologies to develop the correct level of pipelining, to achieve the performance required. The next section is devoted to retiming because, as will be shown, recursive structures, i.e. those involving feedback loops, can present particular problems.

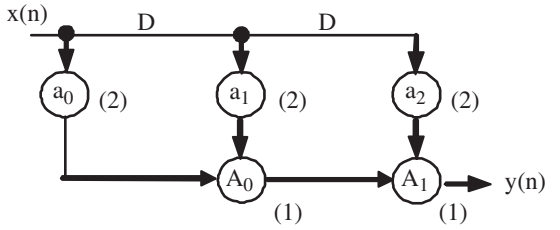


Figure 8.9 Simple DFG

Table 8.1 FIR filter timing

Address Clock	Input	Node 1	Node 2	LUT contents Node 3	Output
0	$x(0)$	$a_0x(0)$	$a_0x(0)$	$a_0x(0)$	$y(0)$
1	$x(1)$	$a_0x(1)$	$a_0x(1) + a_1x(0)$	$a_0x(1) + a_1x(0)$	$y(1)$
2	$x(2)$	$a_0x(2)$	$a_0x(2) + a_1x(1)$	$a_0x(2) + a_1x(1) + a_2x(0)$	$y(2)$
3	$x(3)$	$a_0x(3)$	$a_0x(3) + a_1x(2)$	$a_0x(3) + a_1x(2) + a_2x(1)$	$y(3)$
4	$x(4)$	$a_0x(4)$	$a_0x(4) + a_1x(3)$	$a_0x(4) + a_1x(3) + a_2x(2)$	$y(4)$

8.4 Pipelining DSP Systems

One of the main goals in attaining an FPGA realization is to determine the levels of pipelining needed. The timing of the data through the three-tap FIR filter of Figure 8.8(a) for the nodes labeled (1), (2) and (3) is given in Table 8.1. We can add a delay to each multiplier output as shown in Figure 8.8(a), giving the change in data scheduling shown in Table 8.2. Note that the latency has now increased, as the result is not available for one cycle. However, adding another delay onto the outputs of the adders causes failure, as indicated by Table 8.3. This is because the process by which we are adding these delays has to be carried out in a systematic fashion by the application of a technique known as retiming. Obviously, retiming was applied correctly in the first instance as it did not change the circuit functionality but incorrectly in the second case. Retiming can be applied via the cut theorem as described in Kung (1988).

8.4.1 Retiming

Retiming is a transformation technique used to move delays in a circuit without affecting the input/output characteristics (Leiserson and Saxe 1983). Retiming has been applied in synchronous designs for clock period reduction (Leiserson and Saxe 1983), power consumption reduction (Monteiro *et al.* 1993), and logical synthesis. The basic process of retiming is given in Figure 8.10 (Parhi 1999). For a circuit with two edges U and V and ω delays between them, as shown in Figure 8.10(a), a retimed circuit can be derived with ω_r delays as shown in Figure 8.10(b), by computing the ω_r value as

$$\omega_r(e) = \omega(e) + r(U) - r(V), \tag{8.1}$$

where $r(U)$ and $r(V)$ are the retimed values for nodes U and V, respectively.

Table 8.2 Revised FIR filter timing

Address Clock	Input	Node 1	Node 2	LUT contents Node 3	Output
0	$x(0)$	$a_0x(0)$			
1	$x(1)$	$a_0x(1)$	$a_1x(0)$	$a_0x(0)$	$y(0)$
2	$x(2)$	$a_0x(2)$	$a_1x(1) + a_1x(0)$	$a_0x(1) + a_1x(0)$	$y(1)$
3	$x(3)$	$a_0x(3)$	$a_1x(2) + a_2x(1)$	$a_0x(2) + a_1x(1) + a_2x(0)$	$y(2)$
4	$x(4)$	$a_0x(4)$	$a_1x(3) + a_2x(2)$	$a_0x(3) + a_1x(2) + a_2x(1)$	$y(3)$

Table 8.3 Faulty application of retiming

Address Clock	Input	Node 1	Node 2	LUT contents Node 3	Output
0	$x(0)$	$a_0x(0)$			
1	$x(1)$	$a_0x(1)$	$a_0x(0)$		
2	$x(2)$	$a_0x(2)$	$a_0x(1) + a_1x(0)$	$a_0x(0)$	$y(0)$
3	$x(3)$	$a_0x(3)$	$a_0x(2) + a_1x(1)$	$a_0x(1) + a_1x(0) + a_2x(0)$	
4	$x(4)$	$a_0x(4)$	$a_0x(3) + a_1x(2)$	$a_0x(2) + a_1x(1) + a_2x(1)$	

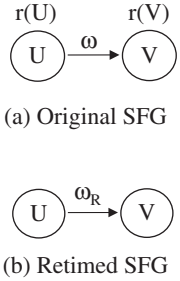
Retiming has a number of properties which can be summarized as follows:

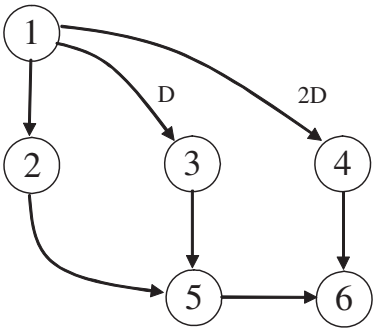
1. The weight of any retimed path is given by Equation (8.1).
2. Retiming does not change the number of delays in a cycle.
3. Retiming does not alter the iteration bound (see later) in a DFG as the number of delays in a cycle does not change.
4. Adding the constant value j to the retiming value of each node does not alter the number of delays in the edges of the retimed graph.

Figure 8.11 gives a number of examples of how retiming can be applied to the FIR filter DFG of Figure 8.11(a). For simplicity, we have replaced the labels a_0, a_1, a_2, A_0 and A_1 of Figure 8.9 by 2, 3, 4, 5 and 6, respectively. We have also shown separate connections between the $x(n)$ data source and nodes 2, 3 and 4; the reasons for this will be shown shortly. By applying equation (8.1) to each of the edges, we get the following relationships for each edge:

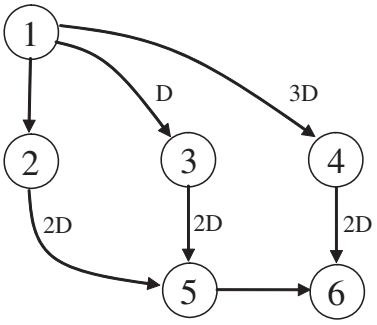
$$\begin{aligned}
\omega_r(1 \to 2) &= \omega(1 \to 2) + r(2) - r(1),\\
\omega_r(1 \to 3) &= \omega(1 \to 3) + r(3) - r(1),\\
\omega_r(1 \to 4) &= \omega(1 \to 4) + r(4) - r(1),\\
\omega_r(2 \to 5) &= \omega(2 \to 5) + r(5) - r(2),\\
\omega_r(3 \to 5) &= \omega(3 \to 5) + r(5) - r(3),\\
\omega_r(4 \to 6) &= \omega(4 \to 6) + r(6) - r(4),\\
\omega_r(5 \to 6) &= \omega(5 \to 6) + r(6) - r(5).
\end{aligned}$$

Figure 8.10 Retiming example

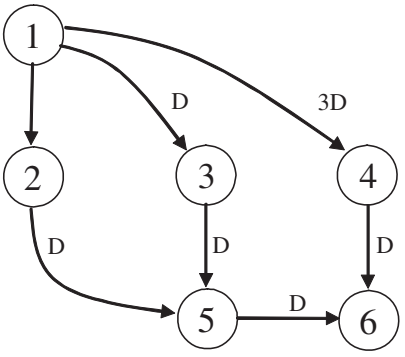




(a) DFG of original 3-tap FIR filter



(b) FIR filter DFG retimed with
$r(1) = -2$, $r(2) = -2$, $r(3) = -2$, $r(4) = -2$, $r(5) = 0$, $r(6) = 0$



(c) FIR filter DFG retimed with
$r(1) = -2$, $r(2) = -2$, $r(3) = -2$, $r(4) = -1$, $r(5) = -1$, $r(6) = 0$

Figure 8.11 Retimed FIR filter

Using a retiming vector $r(1) = -1$, $r(2) = -1$, $r(3) = -1$, $r(4) = -1$, $r(5) = 0$, $r(6) = 0$ above, we get the following values:

$$\begin{aligned}
\omega_r(1 \to 2) &= 0 + (-1) - (-1) = 0,\\
\omega_r(1 \to 3) &= 1 + (-1) - (-1) = 1,\\
\omega_r(1 \to 4) &= 2 + (-1) - (-1) = 2,\\
\omega_r(2 \to 5) &= 0 + (0) - (-1) = 1,\\
\omega_r(3 \to 5) &= 0 + (0) - (-1) = 1,\\
\omega_r(4 \to 6) &= 0 + (0) - (-1) = 1,\\
\omega_r(5 \to 6) &= 0 + (0) - (0) = 0.
\end{aligned}$$

This gives the revised diagram shown in Figure 8.11 which gives a circuit where each multiplier has two pipeline delays at the output edge. A retiming vector could have been

applied which provides one delay at the multiplier output, but the reason for this retiming will be seen later. Application of an alternative retiming vector,

$$\omega_r(1 \rightarrow 2) = 0 + (-2) - (-2) = 0,$$
$$\omega_r(1 \rightarrow 3) = 1 + (-2) - (-2) = 1,$$
$$\omega_r(1 \rightarrow 4) = 2 + (-1) - (-2) = 3,$$
$$\omega_r(2 \rightarrow 5) = 0 + (-1) - (-2) = 1,$$
$$\omega_r(3 \rightarrow 5) = 0 + (-1) - (-2) = 1,$$
$$\omega_r(4 \rightarrow 6) = 0 + (0) - (-1) = 1,$$
$$\omega_r(5 \rightarrow 6) = 0 + (0) - (-1) = 1,$$

namely $r(1) = -1$, $r(2) = -1$, $r(3) = -1$, $r(4) = -1$, $r(5) = -1$, $r(6) = 0$, gives the circuit of Figure 8.11(c) which gives a fully pipelined implementation. It can be seen from this figure that the application of pipelining to the adder stage required an additional delay, D, to be applied to the connection between 1 and 4. It is clear from these two examples that a number of retiming operations can be applied to the FIR filter. A retiming solution is feasible if $w \geq 0$ holds for all edges.

It is clear from the two examples outlined that retiming can be used to introduce inter-iteration constraints to the DFG, manifested as a pipeline delay in the final FPGA implementation (Parhi 1999). However, the major issue would appear to be the determination of the retiming vector which must be such that it moves the delays to the edges needed in the DFG whilst at the same time preserving the viable solution, i.e. $w \geq 0$ holds for all edges. One way of determining the retiming vector is to apply a graphical methodology to the DFG which symbolizes applying retiming. This is known as the cut-set or cut theorem (Kung 1988).

8.4.2 Cut-Set Theorem

A cut-set in an SFG (or DFG) is a minimal set of edges which partitions the SFG into two parts. The procedure is based upon two simple rules.

Rule 1: *Delay scaling*. All delays D presented on the edges of an original SFG may be scaled by D', where $D' \rightarrow \alpha D$; the single positive integer α is also known as the pipelining period of the SFG. Correspondingly, the input and output rates also have to be scaled by a factor of α (with respect to the new time unit D'). Time scaling does not alter the overall timing of the SFG.

Rule 2: *Delay transfer* (Leiserson and Saxe 1983). Given any cut-set of the SFG, which partitions the graph into two components, we can group the edges of the cut-set into inbound and outbound, as shown in Figure 8.12, depending upon the direction assigned to the edges. The delay transfer rule states that a number of delay registers, say k, may be transferred from outbound to inbound edges, or vice versa, without affecting the global system timing.

Let us consider the application of Rule 2 to the FIR filter DFG of Figure 8.11(a). The first cut is applied in Figure 8.13(a) where the DFG graph is cut into two distinct regions or sub-graphs: sub-graph #1 comprising nodes 1, 2, 3 and 4; and sub-graph #2 comprising 5 and 6. Since all edges between the regions are outbound from sub-graph #1 to sub-graph #2, a delay can be added to each. This gives Figure 8.13(b). The second cut splits the DFG

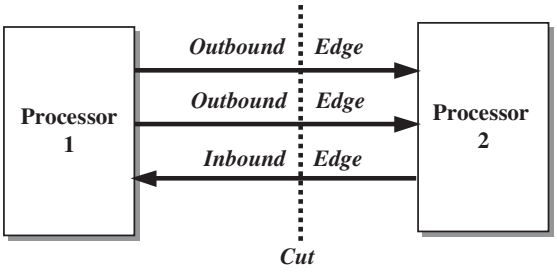


Figure 8.12 Cut-set theorem application

into sub-graph #3, comprising nodes 1, 2, 3 and 5, and sub-graph #4, comprising nodes 4 and 6. The addition of a single delay to this edge leads to the final pipelined design, as shown in Figure 8.11(c).

These rules provide a method of systematically adding, removing and distributing delays in an SFG and therefore adding, removing and distributing registers throughout a circuit, without changing the function. The cut-set retiming procedure is then employed, to cause sufficient delays to appear on the appropriate SFG edges, so that a number of delays can be removed from the graph edges and incorporated into the processing blocks, in order to model pipelining within the processors; if the delays are left on the edges, then this represents pipelining between the processors.

Of course, the selection of the original algorithmic representation can have a big impact on the resulting performance. Take, for example, the alternative version of the SFG shown initially in Figure 8.8(c) and represented as a DFG in Figure 8.14(a); applying an initial cut-set allows pipelining of the multipliers as before, but now applying the cut-set between nodes 3 and 5, and nodes 4 and 6, allows the delay to be transferred, resulting in a circuit architecture with fewer delay elements as shown in Figure 8.14(c).

8.4.3 Application of Delay Scaling

In order to investigate delay scaling, let us consider a recursive structure such as the second-order IIR filter section given by

$$y(n) = a_0x(n) + a_1x(n-1) + a_2x(n-2) + b_1y(n-1) + b_2y(n-2) \tag{8.2}$$

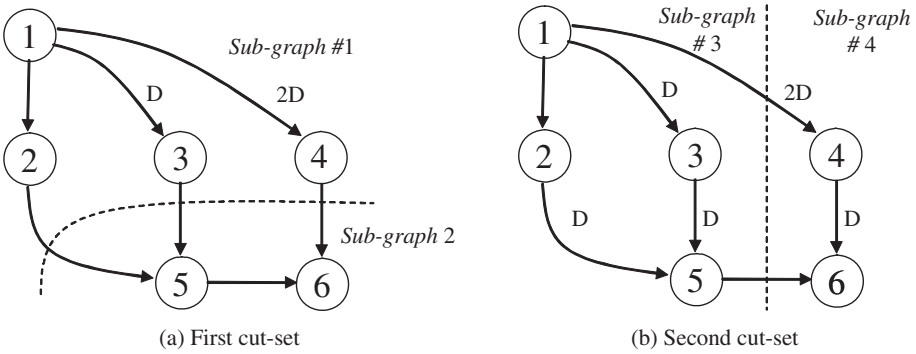


Figure 8.13 Cut-set timing applied to FIR filter

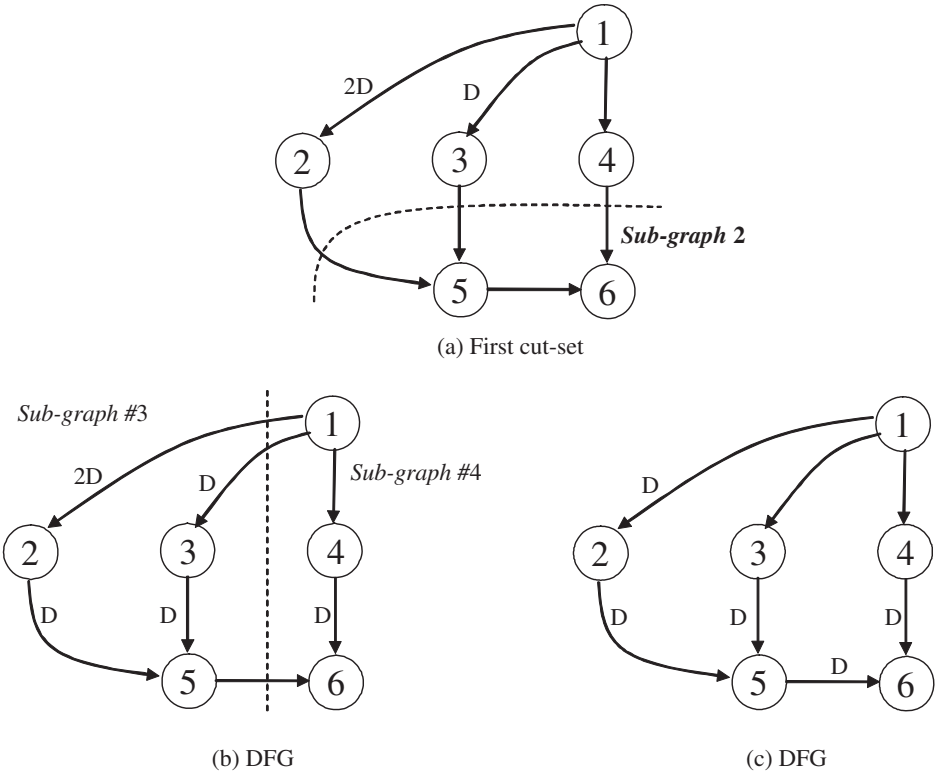


Figure 8.14 Cut-set timing applied to FIR filter

The block diagram and the corresponding DFG is given in Figures 8.15(a) and 8.15(b), respectively. The target is to apply pipelining at the processor level, thereby requiring a delay D on each edge. The problem is that there is not sufficient delay in the $2 \rightarrow 3 \rightarrow 2$ loop to apply retiming. For example, if the cut shown in the figure were applied, this would end up moving the delay on edge $3 \rightarrow 2$ to edge $2 \rightarrow 3$. The issue is

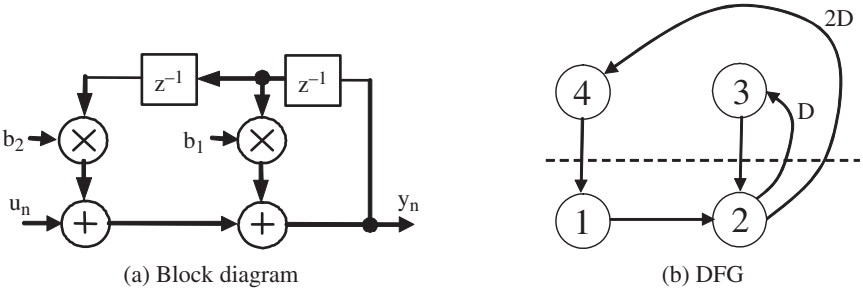


Figure 8.15 Second-order IIR filter

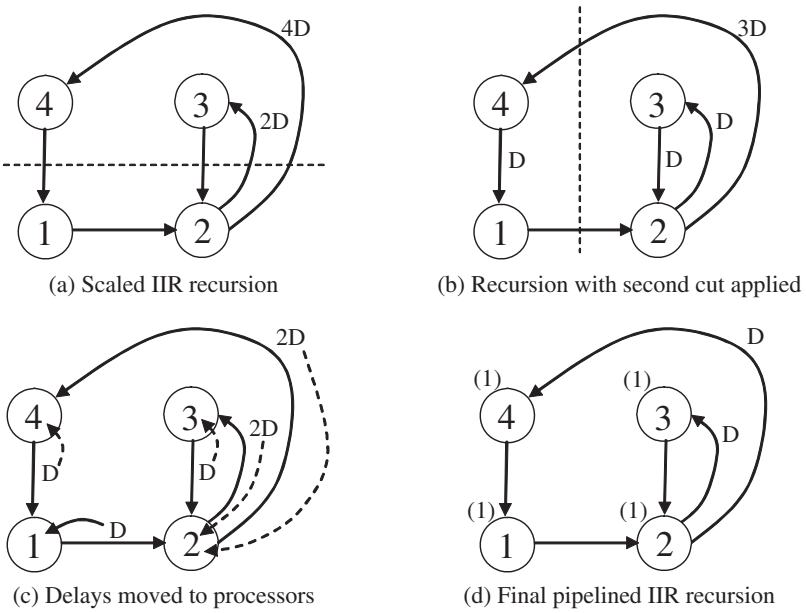


Figure 8.16 Pipelining of a second-order IIR filter. Source: Parhi 1999. Reproduced with permission of John Wiley & Sons.

resolved by applying time scaling, by working out the worse-case pipelining period, as defined by

$$\alpha_c = \frac{B_c}{D_c}, \tag{8.3}$$

$$\alpha = \max \alpha_c. \tag{8.4}$$

In equation (8.3), the value B_c refers to the delays required for processor pipelining and the value D_c refers to the delays available in the original DFG. The optimal pipelining period is computed using equation (8.4) and is then used as the scaling factor. There are two loops as shown, giving a worst-case loop bound of 2. The loops are given in terms of unit time (u.t.) steps:

$1 \rightarrow 2 \rightarrow 4 \rightarrow 1$ $(3u.t.)$
$2 \rightarrow 3 \rightarrow 2$ $(2u.t.)$
$Loopbound\#1(3/2 = 1.5u.t.)$
$Loopbound\#2(2/1 = 2u.t.).$

The process of applying the scaling and retiming is given in Figure 8.16. Applying a scaling of 2 gives the retimed DFG of Figure 8.16(a). Applying the cut shown in the figure gives the modified DFG of Figure 8.16(b) which then has another cut applied, giving the DFG of Figure 8.16(c). Mapping of the delays into the processor and adding the numbers to show the pipelining level gives the final pipelined IIR recursion in Figure 8.16(d).

Table 8.4 Retiming performance in the Xilinx Virtex-5

	Area		Throughput	
Circuit	**DSP48**	**Flip-flops**	**Clock (MHz)**	**Data rate (MHz)**
Figure 8.15(b)	2	20	176	176
Figure 8.16(d)	2	82	377	188

The final implementation has been synthesized using the Xilinx Virtex-5 FPGA and the synthesis results can be viewed for the circuits of Figure 8.15(b) and Figure 8.16(d) in Table 8.4.

8.4.4 Calculation of Pipelining Period

The previous sections have outlined a process for first determining the pipelining period and then allowing scaling of this pipelining period to permit pipelining at the processor level. This is the finest level of pipelining possible within FPGA technology, although, as will be seen in Chapter 13, adding higher levels of pipelining can be beneficial for low-power FPGA implementations. However, the computation of the pipelining period was only carried out on a simple example of an IIR filter second-order section, and therefore much more efficient means of computing the pipelining period are needed. A number of different techniques exist, but the one considered here is the longest path matrix algorithm (Parhi 1999).

8.4.5 Longest Path Matrix Algorithm

A series of matrices is constructed and the iteration bound is found by examining the diagonal elements. If d is the number of delays in DFG, then create $L^{(m)}$ matrices, where $m = 1, 2, \ldots, d$, such that element $l^1_{1,j}$ is the longest path from delay element d which passes through exactly $m - 1$ delays (not including d_i and d_j); if no path exists, then $l^1_{i,j} = -1$. The longest path can be computed using the Bellman–Ford or Floyd–Warshall algorithm (Parhi 1999).

Example 1. Consider the example given in Figure 8.17. Since the aim is to produce a pipelined version of the circuit, we have started with the pipelined version indicated by the (l) expression included in each processor. This can be varied by changing the expression to (0) if the necessary pipelining is not required, or to a higher value, e.g.

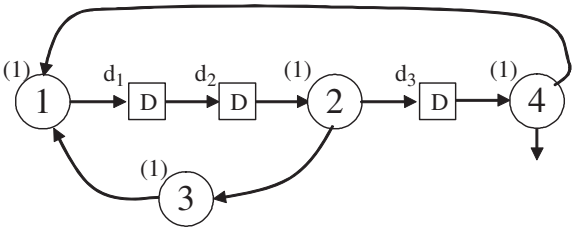


Figure 8.17 Simple DFG example (Parhi 1999)

(2) or (3), if additional pipelined delays are needed in the routing to aid placement and routing or for low-power implementation.

The first stage is to compute the $L^{(m)}$ matrices, beginning with $L^{(1)}$. This is done by generating each term, namely $l^1_{i,j}$, which is given as the path from delay d_i through to d_j. For example, d_1 to d_1 passes through either 1 ($d_1 \to d_2 \to 2 \to 3 \to 1 \to d_1$) or 2 delays ($d_1 \to d_2 \to 2 \to d_4 \to 1 \to d_1$), therefore $l^1_(1,1) = -1$. For $l^1_{3,1}$, the path d_3 to d_1 passes through nodes (4) and (1), giving a delay of 2; therefore, $l^1_{3,1} = 2$. For $l^1_{2,1}$, the path d_2 to d_1 passes through nodes (2), (3) and (1), therefore $l^1_{2,1} = 3$. This gives the matrix

$$\begin{pmatrix} -1 & 0 & -1 \\ 7 & -1 & 3 \\ 3 & -1 & -1 \end{pmatrix}.$$

The higher-order matrices do not need to be derived from the DFG. They can be recursively computed as

$$l^{m+1}_{i,j} = \max_{k \in K}(-1, l^1_{i,j} + l^m_{k,j}),$$

where K is the set of integers k in the interval $[1, d]$ such that neither $l^1_{i,k} = -1$ nor $l^m_{i,k} = -1$ holds. Thus for $l^2_{1,1}$ we can consider $K = 1, 2, 3$ but $K = 1, 3$ include -1, so only $K = 2$ is valid. Thus

$$l^2_{1,1} = \max_{k \in 3}(-1, 0 + 7).$$

The whole of $L^{(2)}$ is generated is this way as shown below:

$$\underbrace{\begin{pmatrix} -1 & 0 & -1 \\ 7 & -1 & 3 \\ 3 & -1 & -1 \end{pmatrix}}_{L^{(1)}} \underbrace{\begin{pmatrix} -1 & 0 & -1 \\ 7 & -1 & 3 \\ 3 & -1 & -1 \end{pmatrix}}_{L^{(1)}} = \underbrace{\begin{pmatrix} 7 & -1 & 3 \\ 6 & 7 & -1 \\ -1 & 3 & -1 \end{pmatrix}}_{L^{(2)}}.$$

While $L^{(2)}$ was computed using only $L^{(1)}$, the matrix $L^{(3)}$, is computed using both $L^{(1)}$ and $L^{(2)}$ as shown below, with the computation for each element given as

$$l^3_{i,j} = \max_{k \in K}(-1, l^1_{i,j} + l^2_{k,j})$$

as before. This gives the computation of $L^{(3)}$ as

$$\underbrace{\begin{pmatrix} -1 & 0 & -1 \\ 7 & -1 & 3 \\ 3 & -1 & -1 \end{pmatrix}}_{L^{(1)}} \underbrace{\begin{pmatrix} 7 & -1 & 3 \\ 6 & 7 & -1 \\ -1 & 3 & -1 \end{pmatrix}}_{L^{(2)}} = \underbrace{\begin{pmatrix} 6 & 7 & -1 \\ 14 & 6 & 10 \\ 10 & -1 & 6 \end{pmatrix}}_{L^{(3)}}.$$

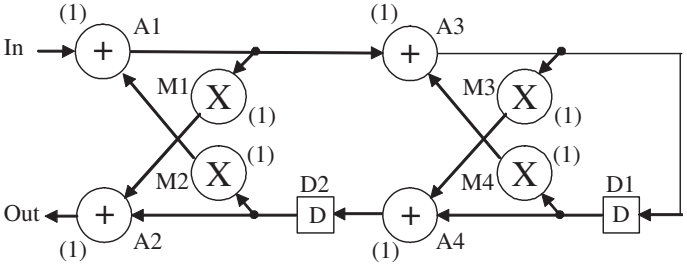
Once the matrix $L^{(m)}$ is created, then the iteration bound can be determined from the equation

$$T_\infty = \max_{i,m \in 1,2,\ldots,D} \left\{ \frac{l^m_{1,l}}{m} \right\}. \tag{8.5}$$

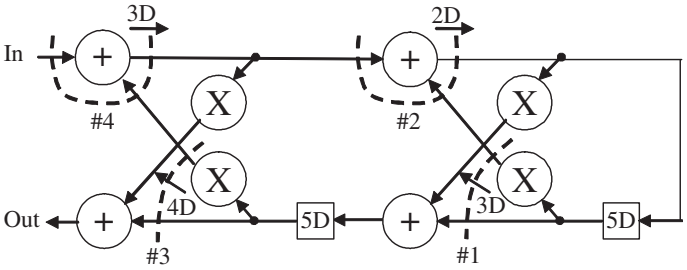
In this case, $m = 3$ as there are three delays, therefore $L^{(3)}$ represents the final iteration. For this example, this gives

$$T_\infty = \left\{ \frac{7}{2}, \frac{7}{2}, \frac{6}{3}, \frac{6}{3}, \frac{6}{3} \right\}.$$

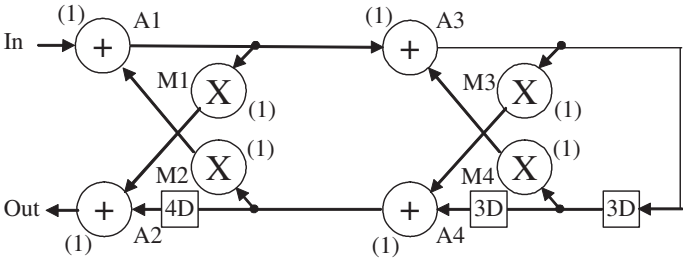
Example 2. Consider the lattice filter DFG structure given in Figure 8.18(a). Once again, a pipelined version has been chosen by selecting a single delay (1) for each processor.



(a) DFG



(b) DFG with cuts indicated



(c) Delays moved to processors

Figure 8.18 Lattice filter

The four possible matrix values are determined as follows:

$$D1 \rightarrow M3 \rightarrow A3 \rightarrow D1$$
$$D1 \rightarrow A4 \rightarrow D2 \text{ and } D1 \rightarrow M4 \rightarrow A3 \rightarrow M3 \rightarrow A4 \rightarrow D2$$
$$D2 \rightarrow M2 \rightarrow A1 \rightarrow A3 \rightarrow D1$$
$$D2 \rightarrow M2 \rightarrow A1 \rightarrow A3 \rightarrow M2 \rightarrow A4 \rightarrow D2,$$

thereby giving

$$\begin{pmatrix} 2 & 4 \\ 3 & 5 \end{pmatrix}.$$

The higher-order matrix L^2 is then calculated as shown below:

$$\underset{L^{(1)}}{\begin{pmatrix} 2 & 4 \\ 3 & 5 \end{pmatrix}} \underset{L^{(1)}}{\begin{pmatrix} 2 & 4 \\ 3 & 5 \end{pmatrix}} = \underset{L^{(2)}}{\begin{pmatrix} 7 & 9 \\ 8 & 10 \end{pmatrix}}$$

This gives the iteration bound

$$T_\infty = \max_{i,m \in 1,2} \left\{ \frac{l_{1,l}^m}{m} \right\}. \tag{8.6}$$

For this example, this gives

$$T_\infty = \left\{ \frac{2}{1}, \frac{5}{1}, \frac{7}{2}, \frac{10}{2} \right\} = 5.$$

Applying this scaling factor to the lattice filter DFG structure of Figure 8.18(b) gives the final structure of Figure 8.18(c), which has pipelined processors as indicated by the (1) expression added to each processor. This final circuit was created by applying delays across the various cuts and applying retiming at the processor level to transfer delays from input to output.

8.5 Parallel Operation

The previous section has highlighted methods to allow levels of pipelining to be applied to an existing DFG representation, mostly based on applying processor-level pipelining as this represents the greatest level applicable in FPGA realizations. This works on the principle that increased speed is required, as demonstrated by the results in Table 8.4, and more clearly speed improvements with FIR filter implementations. Another way to improve performance is to parallelize up the hardware (Figure 8.19). This is done by converting the SISO system such as that in Figure 8.19(a) into a MIMO system such as that illustrated in Figure 8.19(b).

This is considered for the simple FIR filter given earlier. Consider the four-tap delay line filter given by

$$y(n) = a_0 x(n) + a_1 x(n-1) + a_2 x(n-2) + a_3 x(n-3). \tag{8.7}$$

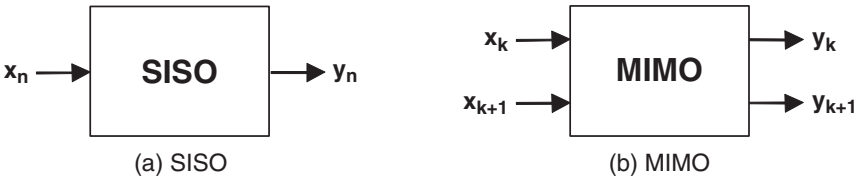


Figure 8.19 Manipulation of parallelism

Assuming blocks of two samples per clock cycle, we get the following iterations performed on one cycle:

$$y(k) = a_0x(k) + a_1x(k-1) + a_2x(k-2) + a_3x(k-3),$$
$$y(k+1) = a_0x(k+1) + a_1x(k) + a_2x(k-1) + a_3x(k-2).$$

In these expressions, two inputs, $x(k)$ and $x(k+1)$, are processed and corresponding outputs, $y(k)$ and $y(k+1)$, produced at the same rate. The data are effectively being processed in blocks and so the process is known as block processing, where k is given as the block size. Block diagrams for the two cycles are given in Figure 8.20. Note that in these structures any delay is interpreted as being k delays as the data are fed at twice the clock rate. As the same data are required at different parts of the filter at the same time, this can be exploited to reduce some of the delay elements, resulting in the circuit of Figure 8.20(b).

The FIR filter has a critical path of $T_M + (N-1)T_A$ where N is the number of filter taps which determines the clock cycle. In the revised implementation, however, two samples are being produced per cycle, thus the throughput rate is $2/T_M + (N-1)T_A$. In this way, block size can be varied as required, but this results in increased hardware cost.

Parhi (1999) introduced a technique where the computation could be reduced by reordering the computation as

$$y(k) = a_0x(k) + a_2x(k-2) + z^{-1}(a_1x(k+1) + a_3x(k-1)).$$

By creating two tap filters, given as $y(1k) = a_0x(k) + a_2x(k-2)$ and $y(2k) = a_1x(k+1) + a_3x(k-1)$, we recast the expressions for $y(k)$ and $y(k+1)$ as

$$y(k) = y(1k) + z^{-1}(y(2(k+1))),$$
$$y(k+1) = (a_0 + a_1)(x(k+1) + x(k)) + (a_2 + a_3)(x(k-1) + x(k-2)) - a_0x(k) - a_1x(k+1) - a_2x(k-2) - a_3x(k-1).$$

This results in a single two-tap filter given in Figure 8.21, comprising a structure with coefficients $a_0 + a_1$ and $a_2 + a_3$, thereby reducing the complexity of the original four-tap filter. It does involve the subtraction of two terms, namely $y(k)$ and $y(2k+1)$, but these were created earlier for the computation of $y(k)$. The impact is to reduce the overall multiplications by two at the expense of one addition/subtraction. This is probably not as important for an FPGA implementation where multiplication cost is comparable to addition for typical wordlengths. More importantly, though, the top and bottom filters are reduced in length by $2(N/2)$ taps and an extra $2-(N/2)$-tap filter is created to realize the first line in each expression. In general terms, filters have been halved, thus

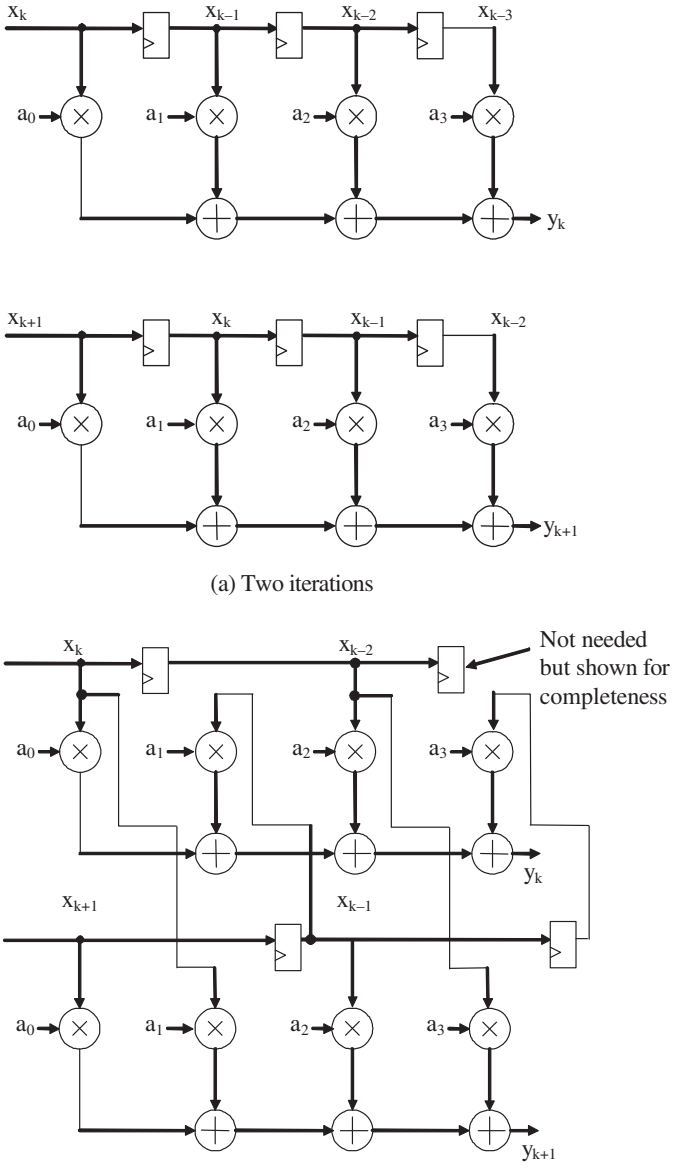


Figure 8.20 Block FIR filter

the critical path is given as $T_M + (N/2)T_A + 3T_A$ with three adders, one to compute $x(k) + x(k+1)$, one to subtract $y(1k)$ and one to subtract $y(2(k+1))$:

$$y(k+1) = (a_0 + a_1)(x(k+1) + x(k)) + (a_2 + a_3)(x(k-1) + x(k-2)) \\ -y(1k) - y(2(k+1)).$$

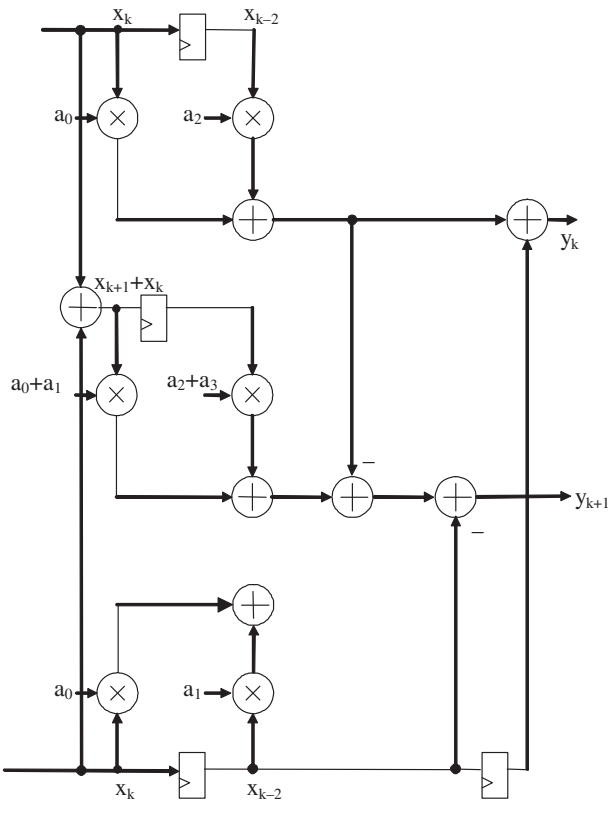


Figure 8.21 Reduced block-based FIR filter

8.5.1 Unfolding

The previous section indicated how we could perform parallel computations in blocks. Strictly speaking, this is known as a transformation technique called unfolding, which is applied to a DSP program to create a new program that performs more than one iteration of the original program. It is typically described using an unfolding factor J which describes the number of iterations by which it is unfolded. For example, consider unfolding the first-order IIR filter section, $y(n) = x(n) + by(n-1)$ by three, giving the expressions below:

$$\begin{aligned} y(k) &= x(k) + by(k-1), \\ y(k+1) &= x(k+1) + by(k), \\ y(k+2) &= x(k+2) + by(k+1). \end{aligned}$$

The SFG and DFG representation is given in Figure 8.22(a), where the adder is replaced by processor A and the multiplier by B. The unfolded version is given in Figure 8.22(b), where A_0, A_1 and A_2 represent the hardware for computing the three additions and B_0, B_1 and B_2 that for computing the three multiplications. With unlooped expressions, each delay is now equivalent to three clock cycles. For example, the previous value needed at

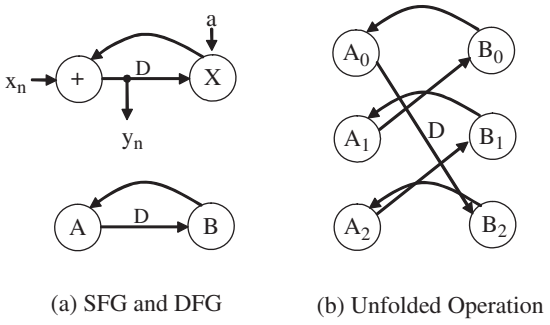


Figure 8.22 Unfolded first-order recursion

processor B_0 is $y(n-1)$ which is generated by delaying the output of A_0, namely $y(n+2)$, by an effective delay of 3. When compared with the original SFG, the delays would appear to have been redistributed between the various arcs for $A_0 - B_0$, $A_1 - B_1$ and $A_2 - B_2$.

An algorithm for automatically performing unfolding is based on the fact that the kth iteration of the node $U(i)$ in the J-unfolded DFG executes the $J(k+i)$th iteration of the node U in the original DFG (Parhi 1999):

1. For each node U in the original DFG, draw the J nodes $U(0), U(1), \ldots, U(J-1)$.
2. For each edge $U \rightarrow V$ with ω delays in the original DFG, draw the J edges $U(i) \rightarrow V(i+\omega)/J$ with $(i + w\%J)$ delays for $i = 0, 1, \ldots, J-1$, where % is the remainder.

Consider the FIR filter DFG, a DFG representation of the FIR filter block diagram of Figure 8.23(a). Computations of the new edges in the transformed graphs, along with the computation of the various delays, are given below for each edge:

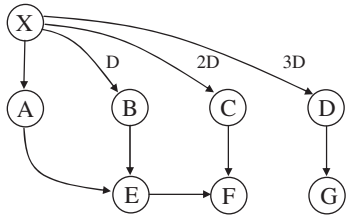
$$
\begin{aligned}
X0 \rightarrow A(0+0)\%2 &= A(0), \quad \text{Delay} = \lfloor 0/2 \rfloor = 0 \\
X1 \rightarrow A(1+0)\%2 &= A(1), \quad \text{Delay} = \lfloor 1/2 \rfloor = 0 \\
X0 \rightarrow B(0+1)\%2 &= B(1), \quad \text{Delay} = \lfloor 1/2 \rfloor = 0 \\
X1 \rightarrow B(1+1)\%2 &= A(2), \quad \text{Delay} = \lfloor 2/2 \rfloor = 1 \\
X0 \rightarrow C(0+2)\%2 &= C(0), \quad \text{Delay} = \lfloor 2/2 \rfloor = 1 \\
X1 \rightarrow C(1+2)\%2 &= C(1), \quad \text{Delay} = \lfloor 3/2 \rfloor = 1 \\
X0 \rightarrow D(0+3)\%2 &= D(1), \quad \text{Delay} = \lfloor 3/2 \rfloor = 1 \\
X1 \rightarrow D(1+3)\%2 &= D(0), \quad \text{Delay} = \lfloor 4/2. \rfloor = 2
\end{aligned}
$$

This gives the unfolded DFG of Figure 8.23(b) which equates to the folded circuit given in Figure 8.23(a).

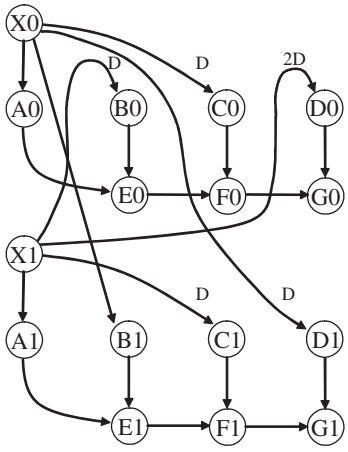
8.5.2 Folding

The previous section outlined a technique for a parallel implementation of the FIR filter structure. However, in some cases, there is a desire to perform hardware sharing, i.e. folding, to reduce the amount of hardware by a factor, say k, and thus also reduce the sampling rate. Consider the FIR filter block diagram of Figure 8.24(a). By collapsing the

Figure 8.23 Unfolded FIR filter-block



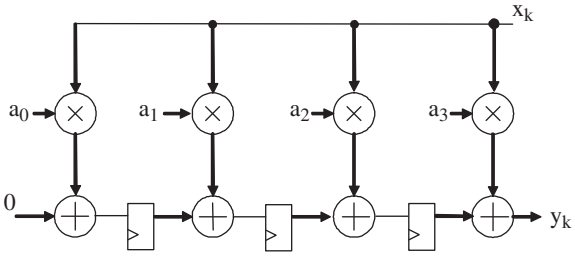
(a) Original block diagram



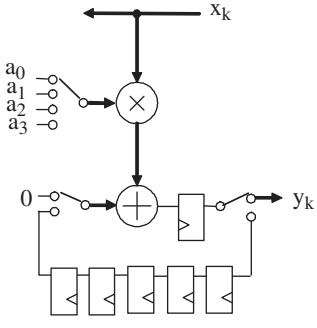
(b) Unfolded operation

filter structure onto itself four times, i.e. folding by four, the circuit of Figure 8.24(b) is derived. In the revised circuit, the hardware requirements have been reduced by four with the operation scheduled onto the single hardware units, as illustrated in Table 8.5.

The timing of the data in terms of the cycle number number is given by 0, 1, 2 and 3, respectively, which repeats every four cycles (strictly, this should by k, $k + 1$, $k + 2$ and



(a) Folded FIR filter section



(b) Folded circuit

Figure 8.24 Folded FIR filter section

Table 8.5 Scheduling for Figure 8.24(b)

Cycle clock	Adder input	Adder input	Adder output	System output
0	a_3	0	$a_3x(0)$	$y(3)'''$
1	a_2	0	$a_2x(0)$	$y(2)''$
2	a_1	0	$a_1x(0)$	$y(1)'$
3	a_0	0	$a_0x(0)$	$y(0)$
4	a_3	0	$a_3x(1)$	$y(4)'''$
5	a_2	$a_2x(1)$	$a_2x(1) + a_3x(0)$	$y(3)''$
6	a_1	$a_1x(1)$	$a_1x(1) + a_2x(0)$	$y(2)'$
7	a_0	$a_0x(1)$	$a_1x(1) + a_2x(0)$	$y(1)$
8	a_3	0	$a_3x(2)$	$y(5)'''$
9	a_2	$a_2x(1) + a_3x(0)$	$a_2x(1) + a_2x(1) + a_3x(0)$	$y(4)''$

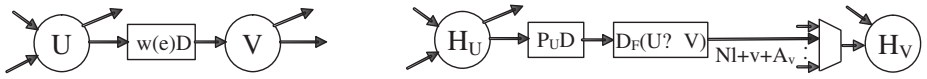
$k + 3$). It is clear from the table that a result is only generated once every four cycles, in this case on the 4th, 8th, ..., cycle. The partial results are shown in brackets as they are not generated as an output. The expression $y(3)'''$ signifies the generation of the third part of $y(3)$, $y(3)''$ means the second part of $y(3)$, etc.

This folding equation is given by

$$D_F(U \overset{e}{\rightarrow} V) = Nw(e) - P_u + v - u, \tag{8.8}$$

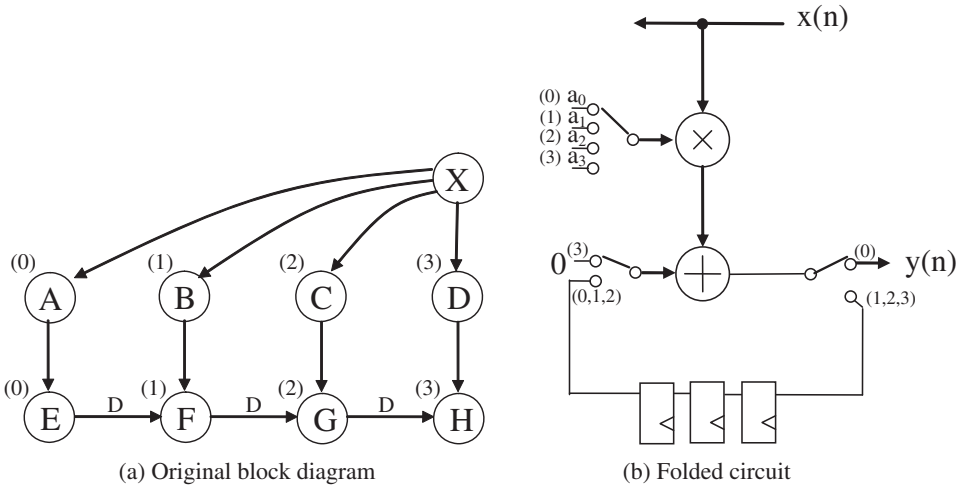
where all inputs of a simplex component arrive at the same time and the pipelining levels from each input to an output are the same (Parhi 1999). In equation (8.8), $w(e)$ is the number of delays in the edge $U \overset{e}{\rightarrow} V$, N is the pipelining period, P_u is the pipelining stages of the H_u output pin, and u and v are folding orders of the nodes U and V that satisfy $0 \leq u, v \leq N - 1$. Consider the edge e connecting the nodes U and V with $w(e)$ delays shown in Figure 8.25(a), where the nodes U and V may be hierarchical blocks. Let the executions of the ith iteration of the nodes U and V be scheduled at time units $NL + u$ and $NL + v$ respectively, where u and v are folding orders of the nodes U and V that satisfy $0 \leq u, v \leq N - 1$.

The folding order of a node is the time partition to which the node is scheduled to execute in hardware (Parhi 1999). H_u and H_v are the functional units that execute the nodes U and V, respectively. N is the folding factor and is defined as the number of operations folded onto a single functional unit. Consider the lth iteration of the node U. If the H_u output pin is pipelined by P_u stages, then the result of the node U is available at the time unit $Nl + u + P_u$, and is used by the $(l + w(e))$th iteration of the node V. If the minimum value of the data time format of H_u input pin is A_v, this input pin of the node V is executed at $N(l + w(e)) + v + A_v$. Therefore, the result must be stored for $D_F(U \overset{e}{\rightarrow} V) = [N(l + w(e)) + v + A_v] - [Nl + P_u + A_v + u]$ time units. The path from H_u to H_v needs



(a) An edge $U \rightarrow U$ with w(e) delays (b) Corresponding folded data path

Figure 8.25 Folding transformation



(a) Original block diagram (b) Folded circuit

Figure 8.26 Folding process

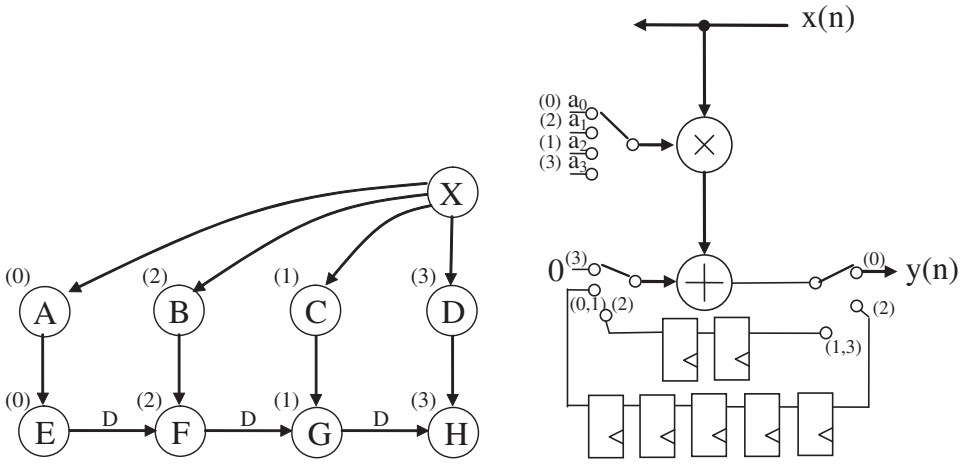
$D_F'(U \overset{e}{\rightarrow} V)$ delays, and data on this path are inputs H_v at $Nl + v + A_v$, as illustrated in Figure 8.25(b). Therefore, the folding equation for hierarchical complexity component is given by

$$D_F(U \overset{e}{\rightarrow} V) = Nw(e) - P_u + A_v + v - u. \tag{8.9}$$

This expression can be systematically applied to the block diagram of Figure 8.25(a) to derive the circuit of Figure 8.25(b). For ease of demonstration, the DFG of Figure 8.26(a) is used. In the figure, an additional adder, H has been added for simplicity of folding. In Figure 8.26(a), we have used a number of brackets to indicate the desired ordering of the processing elements. Thus, the goal indicated is that we want to use one adder to implement the computations $a_3x(n)$, $a_2x(n)$, $a_1x(n)$ and $a_0x(n)$ in the order listed. Thus, these timings indicate the schedule order values u and v. The following computations are created as below, giving the delays and timings required as shown in Figure 8.26(a):

$$\begin{aligned}
D_{F(A\rightarrow H)} &= 4(0) - 0 + 0 - 0 = 0\\
D_{F(B\rightarrow E)} &= 4(0) - 0 + 1 - 1 = 0\\
D_{F(C\rightarrow F)} &= 4(0) - 0 + 3 - 3 = 0\\
D_{F(D\rightarrow G)} &= 4(0) - 0 + 4 - 4 = 0\\
D_{F(H\rightarrow E)} &= 4(1) - 0 + 1 - 2 = 3\\
D_{F(E\rightarrow F)} &= 4(1) - 0 + 2 - 3 = 3\\
D_{F(F\rightarrow G)} &= 4(1) - 0 + 3 - 4 = 3.
\end{aligned}$$

Figure 8.27(a) shows how a reverse in the timing ordering leads to a slightly different folded circuit in Figure 8.27(b) where the delays on the feedback loop have been changed and the timings on the multiplexers have also been altered accordingly. This example



(a) Revised block diagram (b) Changed schedule

Figure 8.27 Alternative folding

demonstrates the impact of changing the time ordering on the computation. The various timing calculations are shown below:

$$
\begin{aligned}
D_{F(A\rightarrow H} &= 4(0) - 0 + 0 - 0 = 0 \\
D_{F(B\rightarrow E} &= 4(0) - 0 + 2 - 2 = 0 \\
D_{F(C\rightarrow F} &= 4(0) - 0 + 1 - 1 = 0 \\
D_{F(D\rightarrow G} &= 4(0) - 0 + 3 - 3 = 0 \\
D_{F(H\rightarrow E} &= 4(0) - 0 + 0 - 2 = 2 \\
D_{F(E\rightarrow F} &= 4(0) - 0 + 2 - 1 = 5 \\
D_{F(F\rightarrow G} &= 4(0) - 0 + 1 - 3 = 2.
\end{aligned}
$$

The example works on a set of order operations given as (1), (3), (2) and (4), respectively, and requires two different connections between adder output and input with different delays, namely 3 and 6.

The application of the technique becomes more complex in recursive computations, as demonstrated using the second-order IIR filter example given in Parhi (1999). In this example, the author demonstrates how the natural redundancy involved when a recursive computation is pipelined, can be exploited to allow hardware sharing to improve efficiency.

8.6 Conclusions

The chapter has briefly covered some techniques for mapping algorithmic descriptions, in the form of DFGs, into circuit architectures. The initial material demonstrates how we could apply delay scaling to first introduce enough delays into the DFGs to allow

retiming to be applied. This translates to FPGA implementations where the number of registers can be varied as required.

In the design examples presented, a pipelining of 1 was chosen as this represents the level of pipelining possible in FPGAs at the processor level. However, if you consider the Xilinx DSP48E2 or the Altera DSP block as a single processing unit, these will allow a number of layers of pipelining as outlined in Chapter 5. Mapping to these types of processor can then be achieved by altering the levels of pipelining accordingly, i.e. by ensuring inter-iteration constraints on the edges which can then be mapped into the nodes to represent pipelining. The delays remaining on the edges then represent the registers needed to ensure correct retiming of the DFGs.

The chapter also reviews how to incorporate parallelism into the DFG representation, which again is a realistic optimization to apply to FPGAs, given the hardware resources available. In reality, a mixture of parallelism and pipelining is usually employed in order to allow the best implementation in terms of area and power that meets the throughput requirement.

These techniques are particularly suitable in generating IP core functionality for specific DSP functionality. As Chapter 11 illustrates, these techniques are now becoming mature, and the focus is moving to creating efficient system implementations from high-level descriptions where the node functionality may already have been captured in the form of IP cores. Thus, the rest of the book concentrates on this higher-level problem.

Bibliography

Altera Corp. 2014 Understanding peak floating-point performance claims. Techical White Paper WP-01222-1.0. Available from www.altera.com (accessed May 11, 2016)].

Leiserson CE, Saxe JB 1983 Optimizing synchronous circuitry by retiming. In *Proc. 3rd Caltech Conf. on VLSI*, pp. 87–116.

Kung SY 1988 *VLSI Array Processors*. Prentice Hall, Englewood Cliffs, NJ.

Monteiro J, Devadas S, Ghosh A 1993 Retiming sequential circuits for low power. *Proc. IEEE Int. Conf. on CAD*, pp. 398–402.

Parhi KK 1999 *VLSI Digital Signal Processing Systems: Design and Implementation*. John Wiley & Sons, New York.