10

Advanced Model-Based FPGA Accelerator Design

10.1 Introduction

As described in Chapter 8, architectural synthesis of SFG models is a powerful approach to the design of high-throughput custom circuit accelerators for FPGA. This approach is one particular case of a wider trend toward design of high-performance embedded systems via the use of a model of computation (MoC), where a domain-specific modeling language is used to express the behavior or a system such that it is semantically precise, well suited to the application at hand and which emphasizes characteristics of its behavior such as timeliness (how the system deals with the concept of time), concurrency, liveness, heterogeneity, interfacing and reactivity in a manner that may be readily exploited for efficient implementation.

A plethora of MoCs have been proposed for modeling of different types of system (Lee and Sangiovanni-Vincentelli 1998), and determining the appropriate MoC for certain types of system should be based on the specific characteristics of that system. For instance, a general characterization of DSP systems could describe systems of repetitive intensive computation on streams of input data. Given this characterization, the dataflow MoC (Najjar *et al.* 1999) has been widely adopted and is a key enabling feature of a range of industry-leading design environments, such as National Instruments' LabVIEW and Keysight Technologies' SystemVUE.

This chapter addresses dataflow modeling and synthesis approaches for advanced accelerator architectures which fall into either of two classes. The first is that of *multidimensional* accelerators: those which operate on complex multidimensional data objects, or multiple channels of data. The second focuses on accelerators with an issue largely ignored by the SFG synthesis techniques of Chapter 8, where it is a heavy demand for high- capacity memory resource which must be accessed at a high rate.

The dataflow modeling of DSP systems is the subject of Section 10.2. The synthesis of custom accelerators is covered in Section 10.3, and this is extended to multidimensional versions in Section 10.4. Memory-intensive accelerators are covered in Section 10.5. A summary is given in Section 10.6.

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

10.2 Dataflow Modeling of DSP Systems

10.2.1 Process Networks

The roots of the most popular current dataflow languages lie in the Kahn process network (KPN) model (Kahn 1974). The KPN model describes a set of parallel processes (or "computing stations") communicating via unidirectional FIFO queues – the general structure of a KPN is shown in Figure 10.1. A computing station maps streams of data tokens impinging along its input lines, using localized memory, onto streams on its output lines.

In DSP systems, the tokens are usually digitized input data values. Continuous input to the system generates streams of input data, prompting the computing stations to produce streams of data on the system outputs. The semantics of mapping between streams of data in KPN makes this modeling approach a good match with the behavior of DSP systems. A KPN structure can be described as a graph $G = (V, E)$, where V is a set of vertices (the computing stations) and E a set of directed edges connecting the vertices. An edge connecting source and sink computing stations a and b respectively is uniquely identified using the tuple (a, b).

Lee and Parks (1995) developed this modeling framework further into the dataflow process network (DPN) domain. DPN models augment KPN computing stations with semantics which define how and under what conditions mapping between streams occurs. Specifically, a stream is said to be composed of a series of data tokens by invocation or *firing* of a dataflow actor; tokens input to an actor are translated to tokens output. Firing only occurs when one of a series of rules is satisfied. Each rule defines a pattern, such as the available number of tokens at the head of an edge FIFO, and when the pre-specified pattern for each input edge is satisfied, the actor may fire. When it does so, tokens are *consumed* from incoming edge FIFOs and resulting tokens *produced* on outgoing edges. Via repeated firing, each actor maps a succession, or a stream, of tokens on its input edges to streams on its output edges. Combined, the KPN and DPN models provide a functional modeling foundation with important properties, such as determinism (Lee and Parks 1995), a foundation upon which a series of more refined dataflow dialects have been devised. Three refinements of specific importance in this section are synchronous dataflow (SDF), cyclo-static dataflow (CSDF) and multidimensional synchronous dataflow (MSDF).

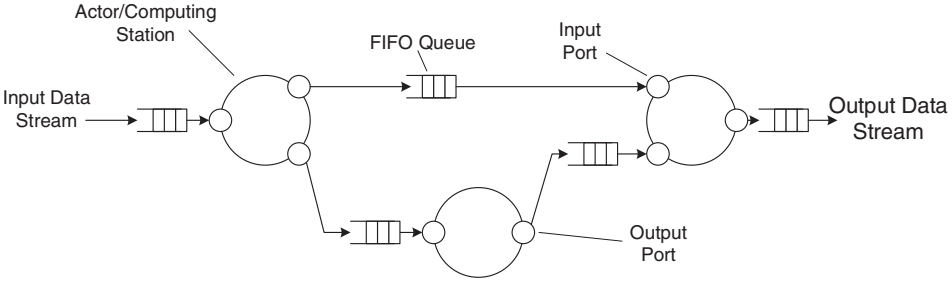


Figure 10.1 Simple KPN structure

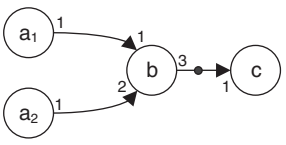


Figure 10.2 Simple SDF graph

10.2.2 Synchronous Dataflow

An SDF model is a DPN with highly restricted semantics. It specifies, for each actor, a single firing rule which states as a condition for actor firing a fixed, integer number of tokens required on its incoming edges (Lee and Parks 1995). Hence SDF is a domain where "we can specify a priori the number of input samples consumed on each input and the number of output samples produced on each output each time the block is invoked" (Lee and Messerschmitt 1987a). This restriction permits compile-time graph analysis with three powerful capabilities:

1. **Consistency**: It can be determined whether a graph is consistent, i.e. whether a program realizing the graph can be constructed which operates on infinite input streams of data within bounded memory.
2. **Deadlock Detection**: It may be determined whether a program realizing the graph operates without deadlock .
3. **Compile-Time Optimization**: Not only can a program implementing the graph be constructed in compile time, the schedule can be analyzed and optimized as regards, for example, buffer and code memory costs or communications costs (Bhattacharyya *et al.* 1999; Sriram and Bhattacharyya 2000).

These capabilities allow compile-time derivation of very low-overhead, efficient programs realizing the SDF model whose buffer memory cost may be highly tuned to the target platform. This capability has pioneered a large body of research into dataflow system modeling, analysis and implementation techniques (Bhattacharyya *et al.* 1999; Sriram and Bhattacharyya 2000). However, this advantage is gained at the expense of expressive power since the SDF forbids data-dependent dataflow behavior.

Each SDF actor exhibits a set of ports, via which it connects to and exchanges tokens with an edge. The number of tokens consumed or produced at a port for each firing of the actor is known as that port's *rate*, *r*. This value is quoted adjacent to the port, as illustrated in Figure10.2.[1] When all ports in the graph are equi-rate, the graph is known as a single-rate or *homogeneous* or single-rate dataflow graph (SR-DFG). Otherwise, the DFG is known as a multi-rate dataflow graph (MR-DFG). A simple SDF model is shown in Figure 10.2. Note the black dot on the edge (b, c); this denotes a delay, which in dataflow terms represents an initial token, i.e. a token resident in the inferred FIFO before any has been produced by the source actor.

If, for actor j connected to edge i, x_j^i (y_j^i) is the rate of the connecting port, an SDF graph can be characterized by a topology matrix Γ, given by

$$\Gamma_{ij} = \begin{cases} x_j^i & \text{if task } j \text{ produces on edge } i \\ -y_j^i & \text{if task } j \text{ consumes from edge } i \\ 0 & \text{otherwise.} \end{cases} \tag{10.1}$$

1 By convention, this annotation is omitted in cases where the rate is 1.

This topology matrix permits compile-time verification of consistency, specifically by determining a number of firings of each actor so that a program schedule may be derived which is balanced, i.e. it may repeat an infinite number of times within bounded memory. It does so by ensuring that the net gain in the number of tokens on each edge, as a result of executing an iteration of the schedule, is zero (Lee 1991). This is achieved by balancing the relative number of firings of each actor according to the rates of the ports via which they are connected. Specifically, for every actor a, which fires proportionally q_a times in an iteration of the schedule and produces r_a tokens per firing, connected to actor b, which fires proportionally q_b times and consumes r_b tokens per firing, since for operation in bounded memory an iteration of the schedule must see all FIFO queues return to their initial state (Lee and Messerschmitt 1987b), the equation

$$q_a r_a = q_b r_b \tag{10.2}$$

holds. Collecting such an equation for each edge in the graph, a system of balance equations is constructed, which is written compactly as

$$\Gamma \mathbf{q} = \mathbf{0}, \tag{10.3}$$

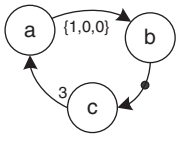
where the *repetitions vector*, $\mathbf{q}$, describes the number of firings of each actor in an iteration of the execution schedule of the graph and where q_i is the number of firings of actor i in the schedule.

10.2.3 Cyclo-static Dataflow

The CSDF model (Bilsen *et al.* 1996) notes the limitation of SDF actors to a single firing rule pre-specifying the availability of an integer number of tokens on each input edge. Due to this restriction, SDF actors can only perform one fixed behavior on each firing. CSDF attempts to broaden this capability to allow an actor to perform a multitude of predefined behaviors whilst maintaining the powerful compile-time analysis features of SDF. In CSDF, actors have cyclically changing actor behavior, whereby an actor j defines a firing *sequence* $\gamma = \{f_j(1), f_j(2), \ldots, f_j(P_j)\}$. Given this sequence, it is then said that the actor operates in one of P_j *phases* with the behavior of γ_i invoked during firing $i(\mathrm{mod}\, j)$.

In addition, the restriction imposed by SDF that the rate of each port be a scalar integer is similarly extended in CSDF to permit rates to be *sequences* of integer scalars. A simple example in shown in Figure 10.3. In this case, whilst b and c are SDF actors (or, more generally, CSDF actors with single-phase firing and rate sequences), a in this case is cyclic, operating a three-phase schedule, with the rate of its output port iterating over the sequence $\{1, 0, 0\}$.

Figure 10.3 Simple CSDF graph



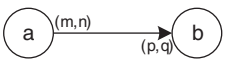


Figure 10.4 Simple MSDF graph

In general, for CSDF actor j connected to edge i, if $X_j^i(n)$ is the total number of tokens produced and $Y_j^i(n)$ the total number consumed during the first n firings of the actor, a CSDF topology matrix Γ is defined by

$$\Gamma_{ij} = \begin{cases} X_j^i(P_j) & \text{if task } j \text{ produces on edge } i \\ -Y_j^i(P_j) & \text{if task } j \text{ consumes from edge } i \\ 0 & \text{otherwise.} \end{cases} \tag{10.4}$$

10.2.4 Multidimensional Synchronous Dataflow

Both SDF and CSDF operate on the assumption that tokens are atomic: a firing of an actor cannot consume anything other than an integer number of tokens traversing along an edge. This restriction is alleviated in MSDF, a domain at its most beneficial for complex multidimensional tokens, first via work which elaborates a single MSDF graph into equivalent SDF structures based on rectangular lattice-shaped problems, such as matrices (Lee 1993a,b), but later further to arbitrary shaped lattices (Murthy and Lee 2002). In MSDF, rates are specified as M-tuples of integers and the number of balance equations per edge increased from 1 to M. An example MSDF graph is shown in Figure 10.4. Note that the form of a multidimensional token is expressed using braces. The balance equations for this graph are given by

$$\begin{aligned} q_{a,1}m &= q_{b,1}p, \\ q_{a,2}n &= q_{b,2}q. \end{aligned} \tag{10.5}$$

The generalization to multiple dimensions inherent in the MSDF model has a similar effect on the mathematical representations of rates and repetitions structures, both of which are generalized to matrices. MSDF provides an elegant solution to multidimensional scheduling problems in SDF graphs, and exposes additional intra-token parallelism for higher order dimension tokens (Lee 1993a).

10.3 Architectural Synthesis of Custom Circuit Accelerators from DFGs

Previous chapters have described how the register-rich programmable logic present in FPGAs makes them ideal for hosting pipelined custom circuit accelerator architectures for high-throughput DSP functions. Furthermore, the substantial body of research into automatically deriving and optimizing these structures from SFGs (SR-DFGs where all ports have a rate of 1) presents a perfect opportunity to enable automatic accelerator synthesis for DFG-based design approaches. A typical architectural synthesis approach deriving such accelerators from generalized MR-DFG models is outlined in Figure10.5.

As this shows, the MR-DFG is first converted to a single-rate equivalent, before undergoing architectural synthesis. This initial conversion is important. SFGs are more

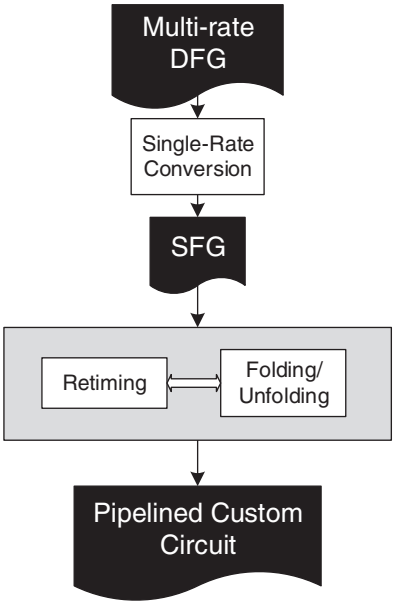


Figure 10.5 MR-DFG accelerator architectural synthesis

restricted than general MR-DFG models, including SDF, CSDF and MSDF. There are three key restrictions of note:

1. The port rates of all ports in the DFG are fixed at unity.
2. The each actor fires only once in an iteration of the schedule.
3. Port tokens are atomic.

Since MR-DFG models are semantically more expressive than SFGs, on conversion to variations in port rates, actor repetitions or token dimensions are manifest explicitly in the structure of the SFG and hence any accelerator derived from it. This means that an SFG accelerator can only realize one configuration of MR-DFG actor and that the designer does not have explicit control over the structure of their accelerator from the MR-DFG structure. In complex FPGA system designs, it is often desired to reuse components in multiple designs; but if an accelerator derived via the SFG route can only realize one MR-DFG actor configuration, how can such reuse be enabled and controlled? Given traditional SFG architectural synthesis techniques, it cannot.

10.4 Model-Based Development of Multi-Channel Dataflow Accelerators

Changing the MR-DFG operating context for an SFG accelerator, e.g. altering token dimensions or port rates, requires re-generation of the accelerator. In many cases, this is unavoidable, but in many others there may be an opportunity, given an augmented synthesis approach, to derive components which are reusable in numerous contexts.

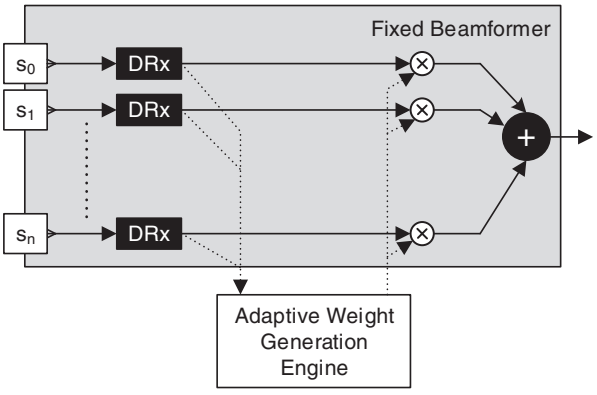


Figure 10.6 Beamformer architecture

Consider one such example. Beamformers are versatile array-processing components for spatial filtering for radar, sonar, biomedical and communications applications (Haykin 2013). A beamformer is typically used with an array of sensors which are positioned at different locations so that they are able to "listen" for a received signal by taking spatial samples of the received propagating wave fields. A block diagram representation of a beamformer structure is shown in Figure 10.6.

As shown, the signals emanating from each antenna element are filtered by a digital receiver (DRx) and scaled before being summed to produce the output signal. The upper portion, consisting of the DRx, scaling and sum components is known as a fixed beamformer (FBF), with the addition of an adaptive weight generation engine producing an adaptive beamformer system.

Consider the case where a custom circuit accelerator is created to realize the DRx component in an FBF. FBF systems can be of different scales and have differing throughput and latency constraints; for example, the number of antenna elements, n, may vary. In order to make sure these requirements are met with minimum cost, it is desirable to use m DRx accelerators, with the same DRx required to process multiple channels of data in the case where $m < n$. But since the original DRx is created to service only one channel, there is no guarantee that it can be reused for multiple channels.

A similar situation may arise in, for example, matrix multiplication. To demonstrate, consider multiplication of two matrices, M_1 and M_2 (of dimensions (m,n) and (n,p), respectively). In this case, assuming that an accelerator has been created to form the product of 3×3 matrices, how may that accelerator be used to multiply M_1 and M_2 when $(m, n, p) = (3, 3, 12)$? One possible approach is illustrated in Figure 10.7.

As this shows, by interpreting M_2 as a sequence of parallel column vectors, groups of columns of arbitrary size can be formed and individually multiplied by M_1 to derive M_3 by concurrent multiplication of M_1 by an array of y matrices $\{M_2^0, M_2^1 \cdots M_2^{y-1}\}$ where M_2^i is composed of the p column vectors $\{i \times \frac{p}{y}, \ldots, ((i+1) \times \frac{p}{y}) - 1\}$. The subdivision of M_2 into parallel submatrices for $p = 4$ is given in Figure 10.7. Note the regular relationship between the number of multipliers and the size of submatrix consumed by each. This kind of relationship could be exploited to regularly change the structure of the DFG,

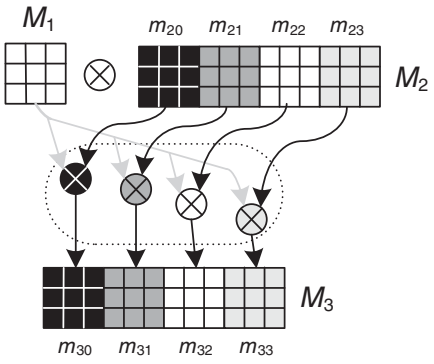


Figure 10.7 Parallel matrix multiplication

trading off the number of actors and the token dimensions processed at the ports of each; given an appropriate one-to-one correspondence between actors and FPGA accelerators, this would then permit explicit control of the number of accelerators and the token dimensions processed by each. It demands, however, accelerators which are sufficiently flexible to process multiple streams of data, trading resource usage with performance without accelerator redesign.

This capability is dependent on two enabling features:

1. Expressing a dataflow application in such a way that the number of actors and the channels processed by each are under designer control without variation affecting the behavior of the application.
2. Synthesizing accelerators which can support varying multi-channel configurations.

10.4.1 Multidimensional Arrayed Dataflow

The key issue with lack of explicit designer control on the structure of the implementation is the lack of structural flexibility in the MR-DFG itself. A single actor in standard dataflow languages like SDF or MSDF can represent any number of tasks in the implementation, rather than employing a close relationship between the number of DFG actors and number of accelerators in the solution. To overcome this structural inflexibility, the multidimensional arrayed dataflow (MADF) domain may be used (McAllister *et al.* 2006).

To demonstrate the semantics of the domain, consider the same matrix multiplication problem described at the beginning of this section. The MADF graph of this problem is given in Figure 10.8. In this formulation, M_1 and M_2 are sources for the operand matrices, whilst *mm* is the matrix multiply actor and M_3 is a sink for the product. In MADF, the notions of DFG actors and edges are extended to arrays. Hence an MADF graph

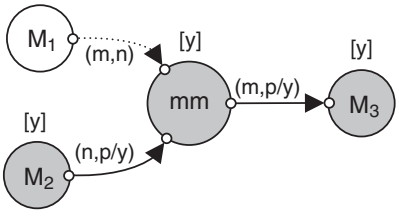


Figure 10.8 Matrix multiplication MADF

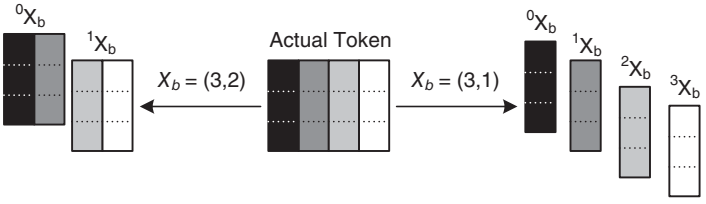


Figure 10.9 Matrix decomposition for fixed token size processing

$G = \{V_a, E_a\}$ describes arrays of actors connected by arrays of edges. Actor arrays are gray, as opposed to single actors (or actor arrays of size 1) which are white. Edge arrays are solid, as opposed to single edges (or edge arrays of size 1) which are dashed. The size of an actor array is quoted in brackets above the actor array.

In such a graph, the system designer controls parameters such as y in Figure 10.8. This is used to define the size of the M_1, M_2, *mm* and M_3 actor arrays, as well as the dimensions of the tokens produced/consumed by M_2, *mm* and M_3. Under a one-to-one translation between the number of, for example, *mm* actors and the number of accelerators, this enables direct graph-level control of the number of accelerators and token dimensions for each. However, as outlined, accelerators derived from SFGs have fixed port token dimensions and a mechanism must be established to allow processing of higher-order tokens.

Consider the case of the array of submatrices of M_2 input to *mm* in the matrix multiplication example of Figure 10.8. How may a single accelerator be made flexible enough to implement any size of input matrix on this input, given that the pipelined accelerator produced from an SFG description has fixed token dimensions?

As outlined at the beginning of this section, each of the y submatrices can be interpreted as a series of p column vectors, with the ith submatrix composed of the column vectors $\{i \times \frac{p}{y}, \ldots, ((i+1) \times \frac{p}{y}) - 1\}$ of M_2. As such, for the case where $y = 4$, the submatrix can be interpreted in two ways, as illustrated in Figure 10.9. As this shows, the matrix can be interpreted as an aggregation of *base* tokens. If the actor to process the submatrix can only process the base tokens, then the aggregate may be processed by using multiple firings of the actor, each of which processes a different component base token. In a sense, then, the actor is treating the aggregate as an array of base tokens over which it iterates.

To support this concept, MADF support variable-sized arrays of actor ports, each of which consumes identical base tokens, with the resulting accelerator derived to process the base token. To enable multiple iterations of the actor to process the multiple base tokens in the actual token, MADF actors may be cyclic (Section 10.2.3), with individual firings consuming one or more base tokens through each port in the array in turn.

Using this formulation, Figure 10.10 illustrates the full, fixed token processing version of the MADF matrix multiplication problem. Note the presence of differentiated arrays

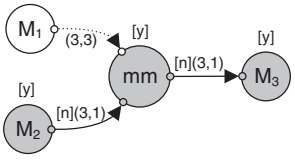


Figure 10.10 Full MADF matrix multiplication

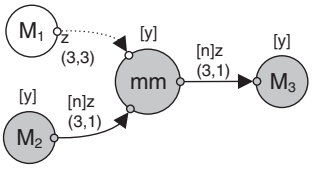


Figure 10.11 Block processing matrix multiplication

of ports (gray) and individual ports (white). In the case of an array of ports, note that the size of the array is annotated on the port using brackets; for instance, the array of ports on M_3 is of size dimension $[n]$.

10.4.2 Block and Interleaved Processing in MADF

Having exposed intra-token parallelism by separating the actual token processed across multiple streams transporting base tokens, further implementation exploration may be enabled. In the case where the port array is used to process a single token, *interleaved* processing of each port in the array is required, i.e. a single base token is consumed through each port in turn to form the full token. In this case, the rate of each port array element is 1. However, having opened up the token processing into a multi-stream processing problem, the generalized multi-rate nature of dataflow languages can be exploited to enable *block* processing via rates greater than 1 at each element of the port array.

At a port array, the ith element has a production/consumption vector of length p_{size} (the size of the port array) with all entries zero except the ith. These vectors exhibit a diagonal relationship (i.e. for the port array a, all entries in the consumption vector of a_0 are zero except the zeroth, all entries in the consumption vector for a_1 are zero except the first, and so forth. A generalized version of this pattern, for a port array with n elements with thresholds z is denoted by $[n]z$, as illustrated in Figure 10.11 for mm when $y = 3$. The value of z, the rate of each port array element, indicates whether interleaved or block processing is used ($z = 1$ for interleaved, $z > 1$ for block processing).

Given a one-to-one correspondence between the number of actors in an MADF graph, the designer then has the capability to control the number of accelerators in the realization. However, the number of accelerators and the characteristics of each are interlinked. For instance, in the case of the matrix multiplication arrangement in Figure 10.10, if the MADF model is to form the product of M_1 and M_2 when $(m, n, p) = (3, 3, 12)$ and mm has a $(3, 3)$ base token, then depending on the number of mm accelerators, y, the characteristics of each will change; in particular, n will vary as $\frac{12}{y}$. Similarly, the behavior will change with the rate of each port. As such, the traditional SFG architectural synthesis approach in Figure 10.5 needs to be augmented to produce accelerators which can process a variable number of streams at each port, and operate on the variable number of streams in either an interleaved or block-processed manner.

10.4.3 MADF Accelerators

When realized on FPGA, an array of MADF actors translates to an equi-sized array of dataflow accelerators (DFAs). The general structure of a DFA is illustrated in Figure 10.12.

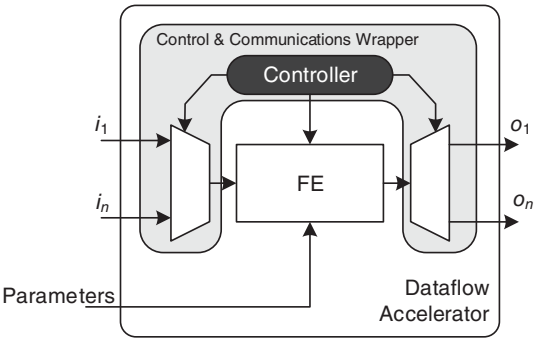


Figure 10.12 Dataflow accelerator architecture

The dataflow accelerator object is composed of three main elements:

1. **Functional engine (FE)**: The FE implements the functionality of the actor and is the accelerator portion of the unit. It can take any architecture, but here is a high-throughput pipelined accelerator. It is created to realize a specific MR-DFG actor, but is capable of performing that functionality on multiple streams of data in either an interleaved or block-processed manner.
2. **Control and communications wrapper (CCW)**: This implements a cyclic schedule to realize multi-stream operation and handles the arbitration of multiple data streams through the FE, in either an interleaved or block processed manner depending on the MADF actor port configuration. The read unit here also implements the necessary edge FIFO buffering for the MADF network.
3. **Parameter bank (PB)**: The PB provides local data storage for run-time constants for the accelerator, e.g. FIR filter tap weights. It is not an active part of the streaming application (i.e. the data stored here can be created and inserted off-line) and so it is not discussed further.

The pipelined FE accelerator part is flexible for reuse across multiple applications and MADF actor configurations, and as such may only require creation and reuse in an accelerator-based design strategy. Efficient generation of FIFO buffers and controllers for automatic generation of dedicated dataflow hardware is a well-researched area (Dalcolmo *et al.* 1998; Harriss *et al.* 2002; Jung and Ha 2004; Williamson and Lee, 1996). The remainder of this section addresses realization of the FE.

10.4.4 Pipelined FE Derivation for MADF Accelerators

The FE part of a dataflow accelerator is a pipelined accelerator, designed as a white box component (WBC) (Yi and Woods 2006). It is one whose structure is parameterized such that it may be reused in various forms in various systems. In the case of MADF accelerators, these parameterized factors are summarized in Table 10.1.

The WBC is derived via architectural synthesis of an SFG representing a specific multi-rate actor instance. To understand how this structure may be reused for multi-stream operation, consider an example two-stage FIR filter WBC, as illustrated in Figure 10.13. The WBC is composed of a computational portion, composed of all of the arithmetic

Table 10.1 WBC parameterization

Parameter	Significance
Streams	Number of streams realized by accelerator is to be shared
Blocking Factor	Blocking (factor > 1) or interleaved (factor = 1) modes.

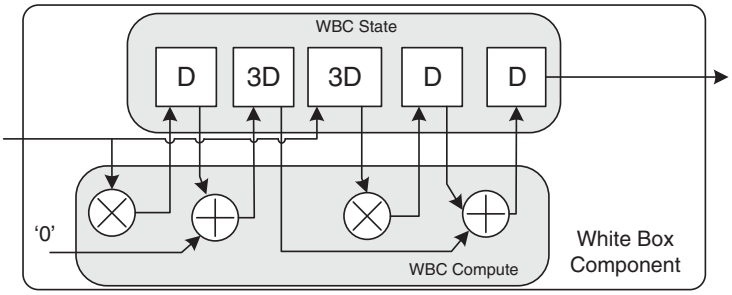


Figure 10.13 Two-stage FIR WBC

operators, and a state space including any delay elements, shift registers or memories.[2] A standard SFG synthesis process such as that in Yi and Woods (2006) will create both, but it is only the state space which restricts the result to a specific MADF actor configuration. The key to designing reusable, configurable accelerators lies in the proper arbitration of the state space and appropriate design of the circuitry such that the data relevant to multiple streams may be stored and properly arbitrated onto the computation portion to enable block or interleaved processing.

To create WBC structures, SFG architectural synthesis is undertaken to create compute and state-space portions for the base configuration, which is then augmented to give the WBC a flexible internal structure which may be regularly changed without redesign to achieve regular changes in MADF actor configuration.

The pipelined WBC architecture resulting from SFG architectural synthesis is merely a retimed version of the original SFG algorithm. The computational resource of the resource must effectively be time-multiplexed between each of the elements of the input stream array, with the entire computation resource of the SFG dedicated to a single stream for a single cycle in the case of interleaved processing, and for multiple cycles in the case of block processing.

To enable interleaved processing, the first stage in the WBC state space augmentation process requires k-slowing (Parhi 1999), where the delay length on every edge resulting from SFG architectural synthesis is scaled by a factor k, and in the case of interleaved processing of n input streams, $k = n$. This type of manipulation is known as *vertical*.

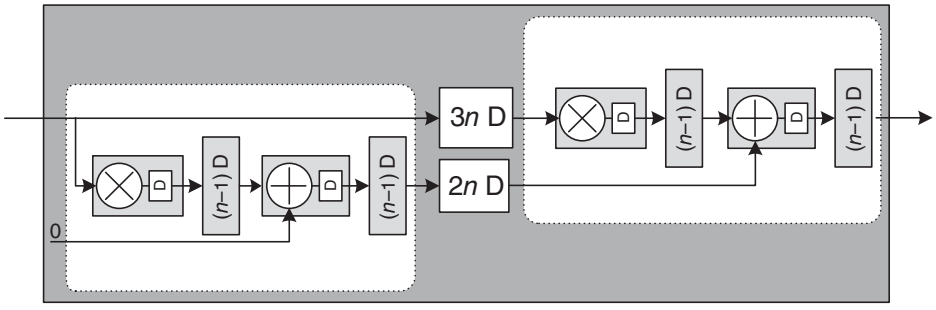
In the case where block processing is required, base tokens are consumed/produced from a single-port array element for a sustained number of cycles. Accordingly, the dataflow accelerator state space should have enough state capacity for all s streams,

2 Henceforth only delay elements are considered.

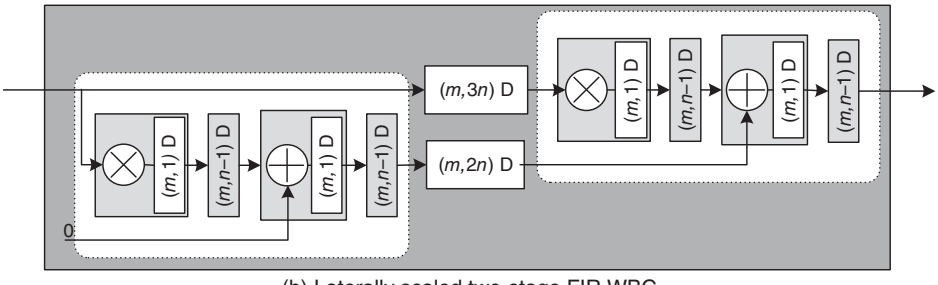
activating the state space associated with a single stream in turn, processing for an arbitrary number of tokens, before loading the state space for the next stream. This kind of load–compute–store behavior is most suited to implementation as a distributed memory component, with the active memory locations determined by controller schedule. This is known here as *lateral* delay scaling, where each SFG delay is scaled into an s-element disRAM.

Given the two general themes of lateral and vertical delay scalability, an architectural synthesis process for reusable WBC accelerators to allow multi-stream actor and accelerator reuse involves four steps:

1. **Perform MADF actor SFG architectural synthesis.** For a chosen MADF actor, C is fixed and defined as the base configuration C_b. This is converted to SFG for architectural synthesis. The MADF actor C_b is the minimum possible set of configuration values for which the resulting pipelined architecture, the base processor P_b, may be used, but by regular alteration of the parameterized structure the processor can implement integer supersets of the configuration. The lower the configuration values in the base, the greater the range of higher-order configurations that the component can implement. To more efficiently implement higher-order configurations, C_b can be raised to a higher value. For a two-stage FIR, $C_b = \{1, 1, 1\}$, the WBC of the P_b is shown in Figure 10.13.
2. **Vertical delay scalability for interleaved processing.** To implement k-slowing for variable interleaved operation, the length of all delays must be scaled by a constant factor m. All the lowest-level components (adder/multipliers) are built from predesigned accelerators which have fixed pipelined depths (in the case of Figure 10.13 these are all 1) which cannot be altered by the designer. To enable the scaling of these delays, these are augmented with delays on their outputs to complete the scaling of the single pipeline stages to that of length m. The resulting FIR circuit architecture for the pipelined FIR of Figure 10.13 is shown in Figure 10.14(a). The notation (m) D refers to an array of delays with dimensions $(1, m)$. Note that all delay lengths are now a factor of m, the vertical scaling factor, and note the presence of the added delay chains on the outputs of the lowest-level components. This type of manipulation is ideally suited to FPGA where long delays are efficiently implemented as shift registers (Xilinx 2005).
3. **Lateral delay scalability for block processing.** For block processing the circuit delays are scaled by a vertical scaling factor n post lateral scaling to allow combined interleaved/block processing if required. This results in arrays of delays with dimensions (m, n). The resulting FIR circuit architecture when this is applied to the circuit of Figure 10.14(a) is shown in Figure 10.14(b). Note the presence of the vertical scaling factor on all delay arrays. This kind of miniature embedded-RAM-based behavior is ideally suited to FPGA implementation, since these can implement small disRAM in programmable logic. These disRAMs have the same timing profile as a simple delay (Xilinx 2005), and as such do not upset edge weights in the circuit architecture.
4. **Retime structure to minimize lateral delay scalability.** When P_b is configured to implement a much higher-order MADF actor configuration than C_b, very large delay lengths can result. To minimize these, retiming is applied to the augmented processor architecture.



(a) Vertically scaled two-stage FIR WBC



(b) Laterally scaled two-stage FIR WBC

Figure 10.14 Scaled variants of two-stage FIR WBC

10.4.5 WBC Configuration

After creation of the WBC architecture P_b, it must be configured for use for specific MADF actor configuration. Consider a base processor created via SFG architectural synthesis P_b realizing a MADF actor with configuration $C_b = (\mathbf{r}_b, \mathbf{x}_b, \mathbf{s}_b)$, where $\mathbf{r}_b, \mathbf{x}_b, \mathbf{s}_b$ respectively represent the rates, token dimensions and number of streams processed by the actor in question, with pipeline period α_c created using SFG architectural synthesis (Parhi 1999). To realize an MADF actor P, where X is an n-dimensional token of size $x(i)$ in the ith dimension, the following procedure is used:

1. Determine the vertical scaling factor m, given by

$$m = \left\lceil \frac{1}{\alpha_c} \prod_{i=0}^{n-1} \frac{x(i)}{x_b(i)} \right\rceil . \tag{10.6}$$

2. k-slow P_b by the factor m.
3. Scale primitive output delays to length $(m-1) \times l$, where l is the number of pipeline stages in the primitive.
4. Scale all delays laterally by the scaling factor n, given by

$$n = \frac{s}{s_b}. \tag{10.7}$$

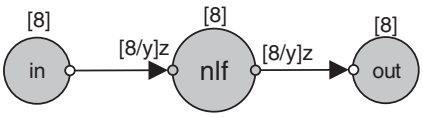


Figure 10.15 Eight-channel NLF filter bank MADF graph

10.4.6 Design Example: Normalized Lattice Filter

In order to demonstrate the effectiveness of this kind of architectural synthesis and exploration approach, it is applied to an eight-channel filter bank design problem, where each filter takes the form of a normalized lattice filter (NLF) (Parhi 1999). The MADF graph is shown in Figure 10.15, with accelerators realizing the *nlf* actors to be created.

As Figure 10.15 shows, *in* and *out* arrays generate an array of eight scalar tokens which are processed by the *nlf* array. The designer controls the size of the *nlf* actor array by manipulating the variable y on the graph canvas. This in turn determines n, the size of the port array of each element of the *nlf* actor array. To test the efficiency of this MADF synthesis and exploration approach the SFG architectural synthesis capability for P_b synthesis is limited to retiming (i.e. advanced architectural explorations such as folding/unfolding are not performed), placing the emphasis for implementation optimization entirely on the MADF design and exploration capabilities. The base processor P_b operates on scalar tokens with $C_b = (1, 1, 1)$ to maximize flexibility by maximizing the number of achievable configurations. The target device is the smallest possible member of the Virtex-II Pro^{TM} family which can support the implementation. This enables two target-device-specific design rules for efficient synthesis:

$$\mathrm{D_{type}} = \begin{cases} \text{FDE} & \text{if } (P, Q) = (1, 1) \\ \text{LUT RAM} & \text{if } P > 1 \\ \text{SRL16+FDE} & \text{otherwise.} \end{cases} \tag{10.8}$$

The SFG of the base NLF actor is shown in Figure 10.16(a), with the SFG of the NLF stage shown in Figure 10.17(a). If the lowest-level components (adders and multipliers) from which the structure is to be constructed are implemented using single-stage pipelined black box components (a common occurrence in modern FPGA), then a particular feature of the NLF structure is the presence of 36 recursive loops in the structure, with the critical loop (Parhi 1999) occurring when two pipelined stages are connected. For single-stage pipelined adders and multipliers, this has a pipeline period, α, of 4 clock cycles. Hence, by equation (10.7), $n = \frac{x_i}{4x_b}$.

The base processor P_b is created via hierarchical SFG architectural synthesis (Yi and Woods 2006), and produces the pipelined architecture of Figure 10.16(b), with the architecture of each stage as in Figure 10.17(b). After lateral and vertical delay scaling and retiming, the NLF and stage WBC architectures are as shown in Figure 10.16(c) and Figure 10.17(c), respectively.

Synthesis of the given architecture for three different values of y has been performed. ${}^{8}_{1}BS\text{-}NLF$, ${}^{2}_{4}BS\text{-}NLF$ and ${}^{1}_{8}BS\text{-}NLF$ are the structures generated when y is 1, 2 and 8 respectively, and each dataflow accelerator performs interleaved sharing over the impinging data streams, whilst results for a single dataflow accelerator processing a 68-element vector (${}^{68}_{1}BS\text{-}NLF$) are also quoted to illustrate the flexibility of the WBC architectures. A block-processing illustration of 16 streams of four-element vector tokens (${}^{1}_{4}BS\text{-}NLF_{16}$)

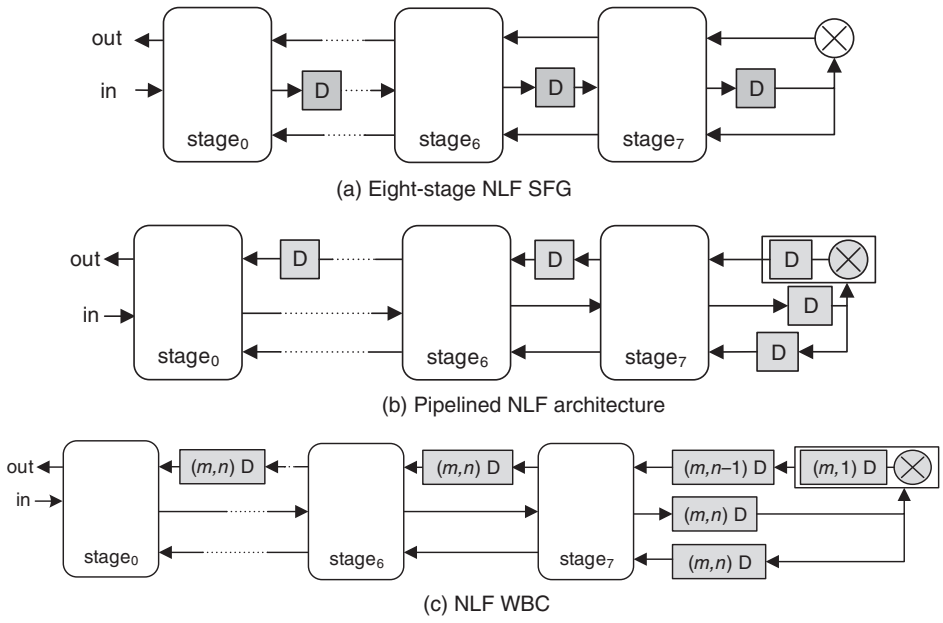


Figure 10.16 NLF SFG, pipelined architecture and WBC

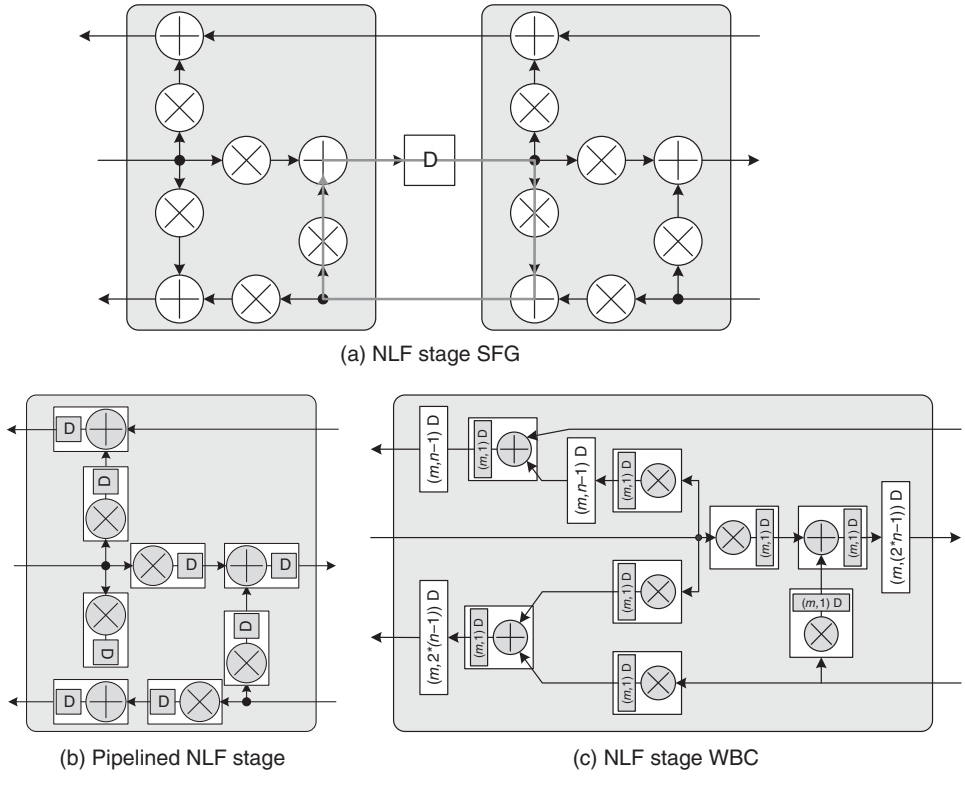


Figure 10.17 NLF stage SFG, pipelined architecture and WBC

Table 10.2 NLF post place and route synthesis results on Virtex-II Pro FPGA

	Logic				Throughput
	LUTs	SRL	DisRAM	mult18	(MSamples/s)
${}^{8}_{1}BS\text{-}NLF$	1472	–	–	312	397.4
${}^{2}_{4}BS\text{-}NLF$	368	–	–	78	377.9
${}^{1}_{8}BS\text{-}NLF$	186	207	–	39	208.6
${}^{68}_{1}BS\text{-}NLF$	186	207	–	39	208.6
${}^{1}_{4}BS\text{-}NLF_{16}$	188	7	576	39	135.8

is also quoted in Table 10.2. These illustrate the effectiveness of this approach for accelerator generation and high-level architecture exploration. Transforming the MADF specification by trading off number of actors in the family, token size per actor, and number of functions in the MADF actor cyclic schedule has enabled an effective optimization approach without redesign.

The initial implementation ($y = 8$, ${}^{1}_{8}BS\text{-}NLF$) created an right-element dataflow accelerator array. Given the large number of multipliers (mult18 in Xilinx Virtex-II) required for implementation, the smallest device on which this architecture can be implemented is an XCV2DA70. However, given the pipeline period inefficiency in the original WBC architecture, reducing y to 2 produces two four-element vector processors (${}^{2}_{4}BS\text{-}NLF$) with almost identical throughput, and enables a significant reduction in required hardware resource with little effect on throughput rate. This amounts to a throughput increase by a factor of 3.9 for each dataflow accelerator with no extra hardware required in the WBC. The large reduction in required number of embedded multipliers also allows implementation on a much smaller XC2DA20 device. Decreasing y still further to 1 produces a single eight-element vector processor (${}^{1}_{8}BS\text{-}NLF$). Whilst the throughput has decreased, a significant hardware saving has been made. The NLF array can now be implemented on a smaller XC2DA7 device.

This example shows that the MADF synthesis approach can achieve impressive implementation results via simple system-level design space exploration. Using a single pipelined accelerator, this approach has enabled highly efficient architectures (3.9 times more efficient than one-to-one mappings) to be easily generated, in a much simpler and more coherent manner than in SFG architectural synthesis. Furthermore, by manipulating a single DFG-level parameter, this design example can automatically generate implementations with wildly varying implementation requirements, offering an order-of-magnitude reduction in device complexity required to implement the design is desired. This illustrates the power of this approach as a system-level, accelerator-based design approach with highly efficient implementation results and rapid design space exploration capabilities.

10.4.7 Design Example: Fixed Beamformer System

The structure of the FBF is highly regular and can be represented in a very compact fashion using MADF, as shown in Figure 10.18. The structure of a beamformer is also included in Figure 10.19 for comparison.

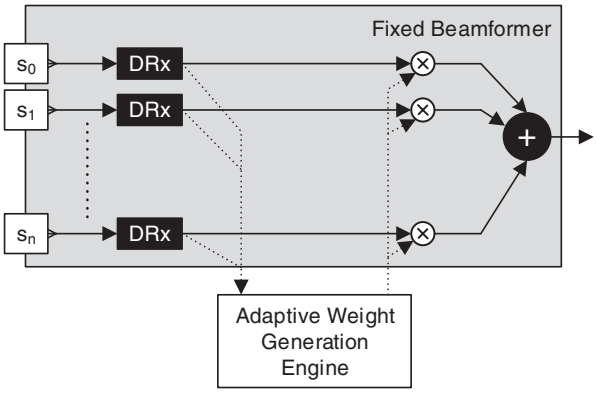


Figure 10.18 Fixed beamformer MADF graph

The MADF graph consists of an array of *n* inputs, one for each sensor in the array. This is tightly correlated with the number of members in the *DRx* and *gain* actor families, as well as the size of the port array *i* on the *sum* actor (again a port array is denoted in grey). Hence by altering the value of *n*, parameterized control of the algorithm structure is harnessed for a variable number of sensors. By coupling the implementation structure tightly to the algorithm structure, this gives close control of the number of *DRx* and *gain* accelerators in the implementation.

For the purposes of this design example, $n = 128$ and the design process targets a Xilinx Virtex-II Pro^{TM} 100 FPGA (Xilinx 2005). The accelerator library consists only of complex multiplication, addition and sum accelerators, and hence the entire system is to be composed from these. The length of the *DRx* filters is taken as 32 taps. Given that this structure then requires 16,896 multipliers, and it is desirable to utilize the provided 18-bit multipliers on the target device (of which only 444 are available) this presents a highly resource-constrained design problem.

To enable the exploration of the number of channels processed by each accelerator in the implementation, each actor must be able to process multiple channels in the MADF algorithm. This is enabled using the MADF structure of Figure 10.20. Here, a second parameter, *m*, has been introduced to denote the number of actors used to process the *n* channels of data. Note that the ports on the *DRx* and *multK* actors are now both families

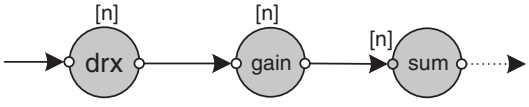


Figure 10.19 Fixed beamformer overview

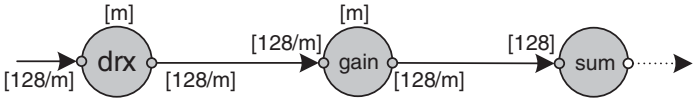


Figure 10.20 Fixed beamformer MADF graph

Table 10.3 FBF post place and route synthesis results on Virtex-II Pro FPGA

m(i)	Logic			mult18	Throughput (MSamples/s)
	LUTs	SRL	DisRAM		
1(i)	3493 (8%)	16128 (37%)	8448 (8%)	99 (22%)	1.45
2(i)	4813 (11%)	16128 (37%)	8448 (19%)	198 (45%)	3.18
4(i)	8544 (19%)	16128 (37%)	8448 (19%)	396 (89%)	6.19
1(b)	3490 (8%)	0 (0%)	24576 (56%)	99 (22%)	1.45
2(b)	4812 (11%)	0 (0%)	24576 (56%)	198 (45%)	3.51
4(b)	8554 (19%)	0 (0%)	24576 (56%)	396 (89%)	1.45

of size m to denote the sharing of the actor amongst $\frac{n}{m}$ data streams processed in a cyclic fashion (McAllister *et al.* 2006). On synthesis, a wide range of synthesis options are available for the FBF custom circuit system on a chosen device, with an accompanying wide range of real-time performance capabilities and resource requirements, and these are summarized in Table 10.3. The breakdown of the proportion of the programmable logic (LUT/FDE) by dataflow accelerator function (WBC, PB or CCW) is given in Table 10.4.

From an initial implementation consisting of a single accelerator process a 128-element vector (i.e. interleave shared across the 128 input streams), increasing the value of m by 2 and 4 has produced corresponding increases in throughput by factors of 2.2 and 4.3 respectively, and it should be noted that the architectures used for accelerator sharing amongst multiple streams exhibit minimal resource differences. This is a direct result of the abstraction of the accelerator architectures for target portability. Whilst the overheads in terms of LUTs (which may be configured as 16-bit SRL or disRAMs) for the WBC wrapping in the dataflow accelerator are high (up to 35%), the major part of this is required entirely for storage of on-chip filter tap and multiplier weights in the SFO parameter banks. This storage penalty is unavoidable without exploiting on-chip embedded BRAMs. In addition, the overhead levels decrease with increasing values of m since the number of tap weights remains constant independent of m. The CCW incurs little LUT overhead, instead exploiting the embedded *muxF5*/*muxF6*/*muxF7*/*muxF8* fabric of the FPGA (Xilinx 2005) to implement the switching. These are not used at all anywhere else in the design and hence are plentiful. Finally, it should be noted that all the accelerators in the system are 100% utilized depending on input data.

Table 10.4 FBF implementation resource breakdown

m(i)	LUT			FDE		
	%WBC	%CCW	%PB	%WBC	%CCW	%PB
1(i)	3.7	31.3	6.5	1.3	0	98.7
2(i)	3.6	28.7	69.5	0.9	0	99.1
4(i)	3.2	25.5	71.3	0.3	0	99.7
1(b)	3.7	31.3	6.5	1.3	0	98.7
2(b)	3.6	28.7	69.5	1.0	0	99.0
4(b)	3.2	25.5	71.3		0.3	99.7

Table 10.5 Memory resources available on Virtex-6

Number		**Off-chip 1**	**Pixel BRAM 215**
Capacity		2 GB	8 K
Access rate (pixels/s)	600 MHz	230 MB/s	600 Mpixel/s
	200 MHz	230 MB/s	200 Mpixel/s

10.5 Model-Based Development for Memory-Intensive Accelerators

The previous section addressed techniques for representing, deriving and optimizing FPGA accelerators for pipelined streaming operators. It is notable, though, that these operations are all performed on scalar data elements, which imposes a small overhead for storage of memory; for the most part memory can be realized using registers, SRLs or disRAMs. However, in a great many cases the demands for buffer memory are high, particularly in applications such as image and video processing where large frames of data are to be handled. In these situations, how are large quantities of buffer memory realized, handled and optimized?

Consider the memory resources on a Xilinx Virtex-6 FPGA in the context of a typical memory-intensive operation for video processing: full search motion estimation (FSME) on CIF 352×288 video frames at 30 frames per second. The resources at the designer's disposal with which this operation may be realized are summarized in Table 10.5.[3] When gauging the anticipated access rate of an on-chip BRAM, it is necessary to take into account the anticipated clock rate of the final synthesize accelerator and the entire system in which it resides; hence, whilst peak clock rates may reach 600 MHz, operation at around 200 MHz is much more reasonable. The designer's challenge is to collect these resources in such a fashion that: data are hosted off-chip where possible to minimize the on-chip buffering cost, and the final accelerator architecture meets real-time performance requirements.

10.5.1 Synchronous Dataflow Representation of FSME

A block diagram of the FSME operation is shown in Figure 10.21(a), and an SDF representation of the FSME operation is shown in, is shown in Figure 10.21(b).

Consider the behavior of the SDF model in Figure 10.21(b). The actors C and R are source actors to represent the stream of current and reference frames incident on the FSME operator. The dimensions of the output tokens indicate the size of the respective video frames. The current frame C is decomposed into 396 (16, 16) non-overlapping CBs via *cb*, with R decomposed into corresponding (48, 48) SWs, each of which is centered around the same center pixel as the corresponding CB. From each SW are extracted 1089 (16, 16) sub-blocks, with each compared with the *cb* in turn via a minimum absolute difference (MAD) operation, with the lowest of these 1089 metrics selected for

3 Assuming BRAM configuration as 8K × 4-bit BRAM, six of which are used to construct a three-byte "pixel BRAM".

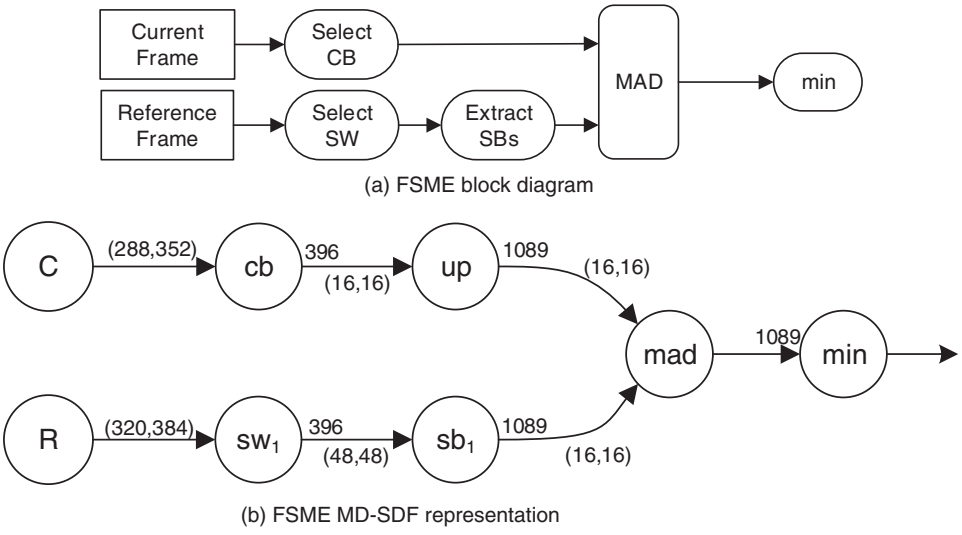


Figure 10.21 Full search motion estimation representations

computation of motion vectors. Accordingly, in order to ensure balanced dataflow, each CB is replicated 1089 times by *up*.

Consider now the memory costs of naive implementation of a model such as this on FPGA, where the FIFO buffering associated with each edge is realized using the available memory resource. The capacity and access rate requirements for the buffer relating to each edge are described in Table 10.6.

As Table 10.6 shows, the capacity and access requirements of each of $e_1, \ldots, e_4$ are such that each could be hosted in off-chip RAM, or indeed on-chip BRAM. However, in the case of e_5 and e_6 there is a major issue: whilst the access rate of these buffers can be realized using 17-pixel BRAM (102 BRAMs in total), many more BRAMs would be required to satisfy its capacity requirements than are available on the device. Similarly, the access rate requirements are such that these buffers cannot be realized using off-chip DRAM. Hence, an architecture such as this could not be realized using the Virtex-6 FPGA, necessitating refinement to overcome the capacity issues on (*up*, *mad*) and (*sb*, *mad*).

Table 10.6 SDF FSME memory requirements

	Capacity		Rate	
Edge	**pixels**	**BRAM**	**pixels/s**	**BRAM**
$e_1 : (C, cb)$	101.4 K	13	3.04 M	1
$e_2 : (R, sw)$	122.9 K	16	3.96 M	1
$e_3 : (cb, up)$	101.4 K	13	3.04 M	1
$e_4 : (sw, sb)$	912.4 K	115	27.37 M	1
$e_5 : (up, mad)$	110.4 M	13,800	3.3 G	17
$e_6 : (sb, mad)$	110.4 M	13,800	3.3 G	17

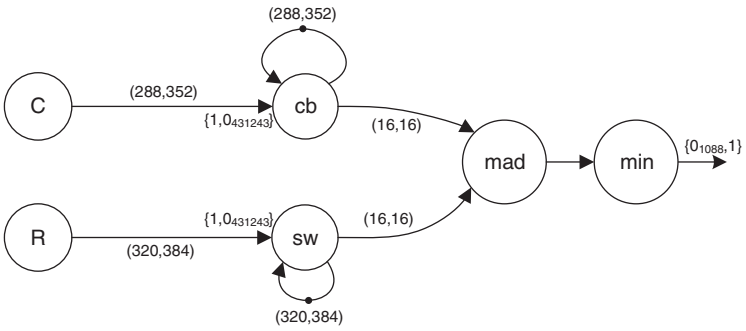


Figure 10.22 FSME modeled using CSDF

Table 10.7 CSDF FSME memory requirements

	Capacity		Rate	
Edge	**pixels**	**BRAM**	**pixels/s**	**BRAM**
$e_1 : (C, cb)$	101.4 K	13	3.04 M	1
$e_2 : (R, sw)$	122.9 K	16	3.96 M	1
$e_3 : (cb, cb)$	101.4 K	13	1.3 T	6500
$e_4 : (sw, sw)$	122.9 K	115	1.59 T	8000
$e_5 : (cb, mad)$	256	1	3.3 G	17
$e_6 : (sw, mad)$	256	1	3.3 G	17

10.5.2 Cyclo-static Representation of FSME

Despite being concise, a major issue in the SDF model in Figure 10.21 is duplication of information; in particular, the excessive capacity requirements of the edges (*up*, *mad*) and (*sb*, *mad*) mask the fact that the former houses 1089 copies of the same (16, 16) CB, whilst the latter contains successive SBs containing substantial amounts of duplicated pixels. In order to address this issue of duplication of pixels, consider the use of CSDF for modeling. A CSDF representation of FSME is shown in Figure 10.22.[4]

Note that *cb* and *sw* are now cyclic actors, each of which operates over 431,244 phases. During the first firing of each, the respective frames are consumed, with the appropriate sequences of CBs for *mad* producing over 431,244 firings. In order to satisfy the need to have access to the entire current and reference frames during each phase for extraction of the relevant CB and SW, these are "recycled" using the self-loops on each of *cb* and *sw*. Consider the capacities and access rates of the buffers associated with each edge in Figure 10.22, as detailed in Table 10.7.

As Table 10.7 shows, the capacity and access rates of e_1 and e_2 are such that these can be realized either off-chip or on-chip, whilst the access rate demands of e_5 and e_6 demand BRAM storage. In addition, the issue of very large-capacity buffers encountered in the SDF model has been avoided by exploiting the multi-phase production capabilities of CSDF – the edges impinging on *mad* now require sufficient capacity for only a single

4 Note that, henceforth, the notation m_n, as seen in Figure 10.22, represents a length-n sequence of elements, each of which takes the value m.

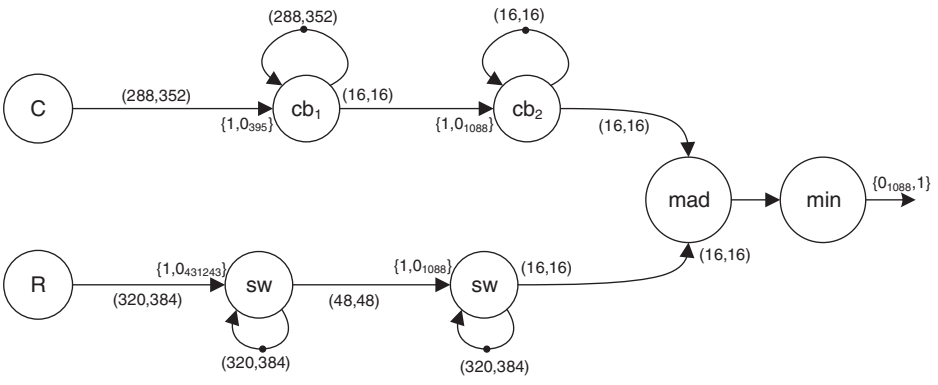


Figure 10.23 Modified FSME CSDF model

CB/SB: very large-capacity savings. Whilst this has come at the cost of an extra buffer for each of the self-loops on *cb* and *sw*, it still represents a very substantial capacity reduction.

However, it has also come at the cost of very high access rate requirements for the buffers representing e_3 and e_4. Indeed, these access rates cannot be realized either off-chip or on-chip and as such a capacity constraint encountered for the SDF network has been replaced by an access rate constraint in this CSDF formulation. Consider the alternative CSDF formulation, shown in Figure 10.23.

In this graph, *cb* and *sw* have both been subdivided into a sequence of two actors. cb_1 is a 396-phase CSDF actor which consumes C during the first phase and recycles it via a self-loop during the remaining 395 phases allowing it to produce a single CB per phase. Subsequently, cb_2 operates over 1089 phases, with a CB consumed during the first phase and recycled via a self-loop during the remaining 1088, allowing a single copy to be produced during each phase. The combination of sw_1 and sw_2 performs a single function on R, except that sw_2 extracts the distinct SBs from each SW rather than simply duplicating the input, as is the case in cb_2. Consider the memory costs for this CSDF formulation, detailed in Table 10.8.

As described, this approach has had a profound impact on the buffer structure for FSME. In this case, all buffers can be hosted off-chip, up to and including e_6 can be

Table 10.8 CSDF (2) full search motion estimation memory requirements

	Capacity		Rate	
Edge	**pixels**	**BRAM**	**pixels/s**	**BRAM**
$e_1 : (C, cb_1)$	101.4 K	13	3.04 M	1
$e_2 : (R, sw_1)$	122.9 K	16	3.96 M	1
$e_3 : (cb_1, cb_1)$	101.4 K	13	1.2 G	6
$e_4 : (sw_1, sw_1)$	122.9 K	115	1.5 G	8
$e_5 : (cb_1, cb_2)$	256	1	3.04 M	1
$e_6 : (sw_1, sw_2)$	2304	1	27.37 M	1
$e_7 : (cb_2, cb_2)$	256	1	3.3 G	17
$e_8 : (sw_2, cw_2)$	2304	1	29.8 G	149
$e_9 : (cb_2, mad)$	256	1	3.3 G	17
$e_{10} : (sw_2, mad)$	256	1	3.3 G	17

hosted off-chip, with the access rates of $e_7, \ldots, e_{10}$ dictating on-chip realization. Whilst the access rate of e_8 in particular is quite high and imposes a high BRAM cost, the situation is now that a realization of this form is at least feasible; this was not previously the case. In addition, the use of more advanced dataflow modeling approaches exploiting non-destructive read capabilities can be used to reduce the on-chip cost even further (Fischaber *et al.* 2010; Denolf *et al.* 2007).

10.6 Summary

This section has described techniques to enable system-level design and optimization of custom circuit accelerators for FPGA using dataflow application models.

The use of MADF as a modeling approach for DSP systems helps encapsulate the required aspects of system flexibility for DSP systems, in particular the ability to exploit data-level parallelism, and control how this influences the implementation. This has been shown to be an effective approach; for an NLF filter design example, impressive gains in the productivity of the design approach were achieved. In this example, this included an almost fourfold increase in the efficiency of the implementation via simple transformations at the DFG level, negating the need for complex SFG architectural manipulations.

Otherwise, this approach has proven effective at rapid design space exploration, producing NLF implementations of varying throughput and drastically different physical resource requirements (on order-of-magnitude variation in device complexity) simply by manipulating a single parameter at the graph level. Further, in an FBF design example the effectiveness of this approach was demonstrated by enabling rapid design space exploration, producing a variety of implementations for a specific device via manipulation of a single DFG parameter.

Similarly, the use of CSDF modeling has been shown to make feasible realization of FPGA accelerators which otherwise could not have been achieved. In particular, for memory-intensive accelerators which access large amounts of memory at high access rates, careful design is required to ensure that both the capacity and access rate requirements are met, whilst reducing cost if possible. Frequently, this means devising combinations of on-chip BRAM and off-chip DRAM in multi-level memory structures customized to the application and performance requirements. The design of an FSME accelerator has highlighted both capacity and access rate issues which have been overcome by intuitive employment of CSDF modeling, allowing an otherwise infeasible accelerator to be realized on the Virtex-6 FPGA.

Bibliography

Bilsen G, Engels M, Lauwereins R, Peperstraete J 1996 Cyclo-static dataflow. *IEEE Trans. on Signal Processing*, 44(2), 397–408.

Bhattacharyya SS, Murthy PK, Lee EA 1999 Synthesis of embedded software from synchronous dataflow specifications. *J. of VLSI Signal Processing*, 21(2), 151–166.

Dalcolmo J, Lauwereins R, Ade M 1998 Code generation of data dominated DSP applications for FPGA targets. In *Proc. Int. Workshop on Rapid System Prototyping*, pp. 162–167.

Denolf K, Bekooij M, Gerrit J, Cockx J, Verkest D, Corporaal H 2007 Exploiting the expressiveness of cyclo-static dataflow to model multimedia implementations. *EURASIP J. on Advances in Signal Processing*, 2007, 084078.

Fischaber S, Woods R, McAllister J 2010 SoC memory hierarchy derivation from dataflow graphs. *J. of VLSI Signal Processing*, 60(3), 345–361.

Harriss T, Walke R, Kienhuis B, Deprettere EF 2002 Compilation from Matlab to process networks realised in FPGA. *Design Automation for Embedded Systems*, 7(4), 385–403.

Haykin S 2013 *Adaptive Filter Theory*, 5th edn. Pearson, Upper Saddle River, NJ.

Jung H, Ha S 2004 Hardware synthesis from coarse-grained dataflow specification for fast HW/SW cosynthesis. In *Proc. Int. Conf. on Hardware/Software Codesign and System Synthesis*, pp. 24–29.

Kahn G 1974 The semantics of a simple language for parallel programming. *Proc. IFIP Congress*, pp. 471–475.

Lee EA 1991 Consistency in dataflow graphs. *IEEE Trans. on Parallel and Distributed Systems*, 2(2), 223–235.

Lee EA 1993a Multidimensional streams rooted in dataflow. *IFIP Transactions: Proceedings of the IFIP WG10.3. Working Conf. on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, A-23, pp. 295–306.

Lee EA 1993b Representing and exploiting data parallelism using multidimensional dataflow diagrams. In *Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, pp. 453–456.

Lee EA, Messerschmitt DG 1987a Synchronous data flow. *Proc. of the IEEE*, 75(9), 1235–1245.

Lee EA, Messerschmitt DG 1987b Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. on Computers*, 36(1), 24–35.

Lee EA, Parks TM 1995 Dataflow process networks. *Proc. of the IEEE*, 83(5), 773–801.

Lee EA, Sangiovanni-Vincentelli A 1998 A framework for comparing models of computation. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 17(12), 1217–1229.

McAllister J, Woods R, Walke R,d Reilly D 2006 Multidimensional DSP core synthesis for FPGA. *J. of VLSI Signal Processing Systems for Signal, Image and Video Technology*, 43(2), 207–221.

Murthy PK, Lee EA 2002 Multidimensional synchronous dataflow. *IEEE Trans. on Signal Processing*, 50(8), 2064–2079.

Najjar WA, Lee EA, Gao GR 1999 Advances in the dataflow computational model. *Parallel Computing*, 25(4), 1907–1929.

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

Sriram S, Bhattacharyya SS 2000 *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel-Dekker, New York.

Williamson MC, Lee EA 1996 Synthesis of parallel hardware implementations from synchronous dataflow graph specifications. In *Proc. 30th Asilomar Conf. on Signals, Systems and Computers*, 2, pp. 1340–1343.

Xilinx Inc. 2005 Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet. Available from http://www.xilinx.com (accessed June 11, 2015).

Yi Y, Woods R 2006 Hierarchical synthesis of complex DSP functions using IRIS. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 25(5), 806–820.