# 11

# Adaptive Beamformer Example

The material in Chapter 9 highlighted the importance of IP in the design of complex FPGA-based systems. In particular, the use of soft IP has had a major impact in creating such systems. This has created a new market, as witnessed by the Design & Reuse website (http://www.design-reuse.com/) which has 16,000 IP cores from 450 vendors, and the open source cores available from the OpenCores website (opencores.org). This, along with the major FPGA vendors' cores (LogiCore from Xilinx and MegaCore® from Altera) as well as their partners' programs, represents a core body of work.

A lot of companies and FPGA developers will have invested a lot of effort into creating designs which well match their specific application. It may then seem relatively straightforward to extend this effort to create soft IP for a range of application domains for this function. However, the designer may have undertaken a number of optimizations specific to a FPGA family which will not transfer well to other vendors. Moreover, the design may not necessarily scale well to the functional parameters.

The ability to create an IP core requires a number of key stages. Firstly, the designer needs to generate the list of parameters to which the core design should scale. The architecture should then be designed such that it scales effectively across these parameters; to be done effectively, this requires a detailed design process. The description is considered in this chapter for a QR-based IP core for adaptive beamforming. It is shown how an original architecture developed for the design can then be mapped and folded to achieve an efficient scalable implementation based on the system requirements.

Section 11.1 provides an introduction to the topic of adaptive beamforming. Section 11.2 outlines the generic process and then how it is applied to adaptive beamforming. Section 11.3 discusses how the algorithm is mapped to the architecture and shows how it is applied to the squared Givens rotations for RLS filtering. The efficient architecture design is then outlined in Section 11.4 and applied to the QR design example. Section 11.5 outlines the design of a generic QR architecture. A key aspect of the operation is the retiming the generic architecture which is covered in Section 11.6. Section 11.7 covers the parameterizable QR architecture, and the generic control is then covered in Section 11.8. The application to the beamformer design is then addressed in Section 11.9, with final comments given in Section 11.10.

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

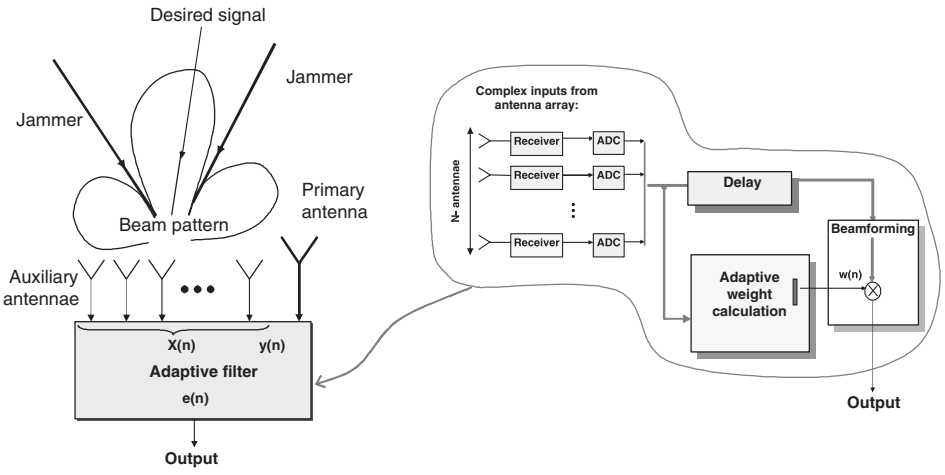



**Figure 11.1** Diagram of an adaptive beamformer for interference canceling

## 11.1 Introduction to Adaptive Beamforming

Adaptive beamforming is a form of filtering whereby input signals are received from a number of spatially separated antennae, referred to as an antenna array. Typically, its function is to suppress signals from every direction other than the desired "look direction" by introducing deep nulls in the beam pattern in the direction of the interference. The beamformer output is a weighted linear combination of input signals from the antenna array represented by complex numbers, therefore allowing an optimization both in amplitude and phase due to the spatial element of the incoming data.

Figure 11.1 shows an example with one primary antenna and a number of auxiliary antennae. The primary signal constitutes the input from the main antennae, which has high directivity. The auxiliary signals contain samples of interference threatening to swamp the desired signal. The filter eliminates this interference by removing any signals in common with the primary input signal. The input data from the auxiliary and primary antennae are fed into the adaptive filter, from which the weights are calculated. These weights are then applied on the delayed input data to produce the output beam.

There are a range of applications for adaptive beamforming, from military radar applications to communications and medical applications (Athanasiadis *et al.* 2005; Baxter and McWhirter 2003; Choi and Shim 2000; de Lathauwer *et al.* 2000; Hudson 1981; Shan and Kailath 1985; Wiltgen 2007). Due to the possible applications for such a core, this chapter investigates the development of an IP core to perform the key computation found in a number of such adaptive beamforming applications.

## 11.2 Generic Design Process

Figure 11.2 gives a summary of a typical design process followed in the development of a single use implementation. It also gives the additional considerations required in generic IP core design. In both cases, the process begins with a detailed specification

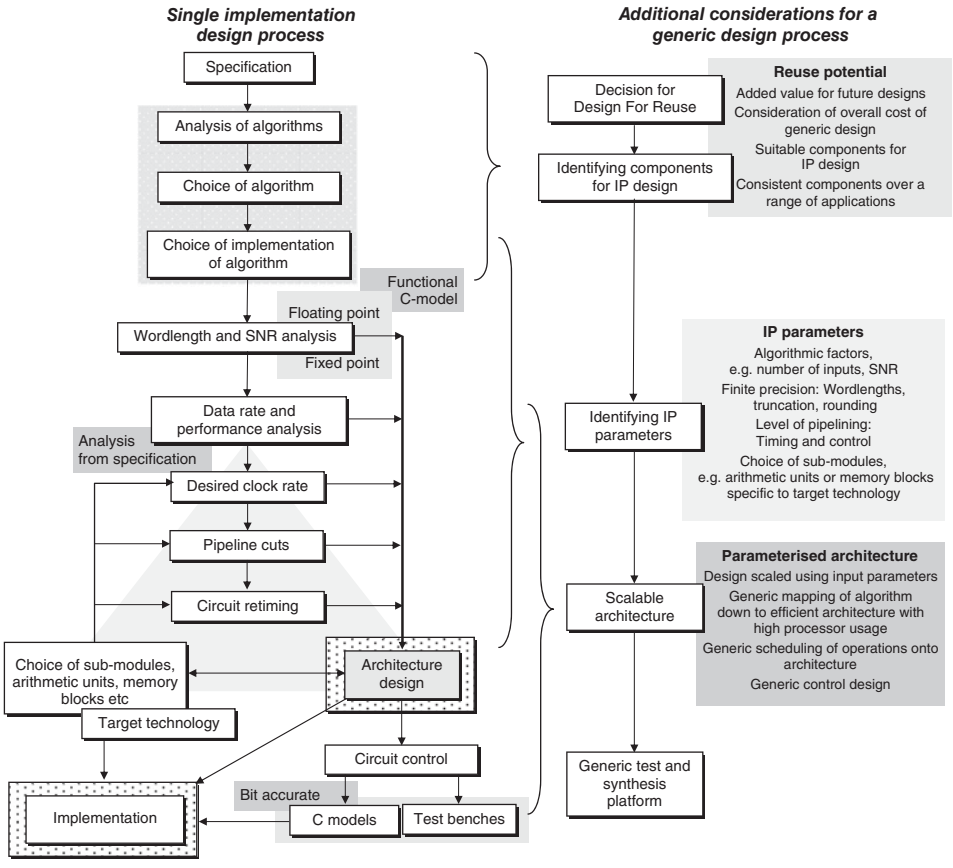



**Figure 11.2** Generic design process

of the problem. At this point, consideration may be given to employing a design-for-reuse strategy to develop a generic product. This is based on a number of factors, the most prevalent being whether such a generic core would be worthwhile to have. Would it be applicable over a range of applications, or is it a one-off requirement? There is an initial extra cost in terms of money and time in the development of a generic core, so it is essential that this cost will be more than recouped if the IP core is used in future designs.

Once defined, an analysis is performed to determine the most suitable algorithm. Once chosen, the method for implementation of this algorithm is key as, even with simple addition, there are a number of different arithmetic techniques; these impact on the overall performance of a circuit, in terms, for example, of area compactness, critical path, and power dissipation.

Within a design, there may only be a number of key components that will be suitable to implement as IP cores. These are the parts of the design that will have a level of consistency from application to application. It is imperative to determine expected variations for future specifications. Are all these variables definable with parameters within

the generic design, or would other techniques be required to create the flexibility of the design? An example of this could be hardware and software co-design. Here, the fixed components could be implemented as IP cores driven by a software harness adding the needed flexibility for further developments.

The choice of fixed-point or floating-point arithmetic is also vital. From the data rate and performance analysis, a decision can be made regarding certain issues for the architecture design. A desired clock rate may be required to meet certain data rate requirements. Again this will relate to the target technology or specific FPGA device. Clock rate and area criteria will also influence the choice of submodules within the design and the level at which they may need to be pipelined so as to meet circuit speeds.

What we have is an interlinked loop, as depicted in Figure 11.2, with each factor influencing a number of others. With additional pipeline cuts there will be effects on circuit timing and area as well as the desired improvement in clock rate. All these factors influence the final architecture design. It is a multidimensional optimization with no one parameter operating in isolation.

Within a generic design, different allowable ranges may be set on the parameters defining the generated architectures. Different wordlength parameters will then have a knock-on effect on the level of pipelining required to meet certain performance criteria. The choice of submodules will also be an important factor. The target technology will determine the maximum achievable data rates and also the physical cost of the implementation.

Again, for a generic design, choices could be made available for a range of target implementations. Parameters could be set to switch between ASIC-specific and FPGA-specific code. Even within a certain implementation platform, there should be parameters in place to support a range of target technologies or devices, so as to make the most of their capabilities and the availability of on board processors or arithmetic units.

There may also be a need for a refined architecture solution meeting the performance criteria but at a reduced area cost. This is the case when the algorithm functionality is mapped down onto a reduced number of processors, the idea being that the level of hardware for the design could be scaled to meet the performance criteria of the application. With scalable designs comes the need for scalable control circuitry and scheduling and retiming of operations. These factors form the key mechanics of a successful generic design. Generating an architecture to meet the performance criteria of a larger design is one thing, but developing the generic scheduling and control of such a design is of a different level of complexity.

Software modeling of the algorithm is essential in the design development. Initially, the model is used to functionally verify the design and to analyze finite precision effects. It then forms the basis for further development, allowing test data to be generated and used within a testbench to validate the design. For the generic IP core, the software modeling is an important part of the design-for-reuse process. A core may be available to meet the needs of a range of applications; however, analysis is still required from the outset to determine the desired criteria for the implementation, such as SNR and data wordlengths. The software model is used to determine the needs for the system, and from this analysis a set of parameters should be derived and used to generate a suitable implementation using the IP core.

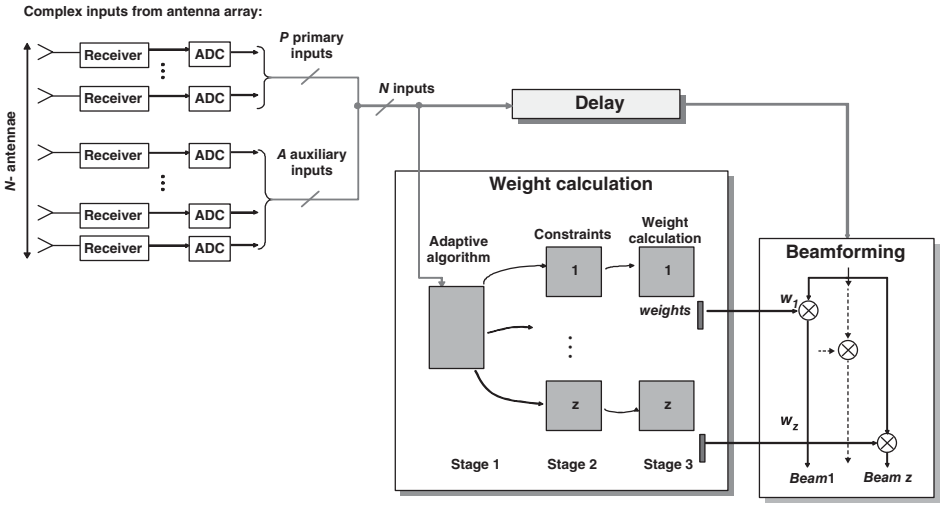



**Figure 11.3** Multiple beam adaptive beamformer system

### 11.2.1 Adaptive Beamforming Specification

Adaptive beamforming is a general algorithm applicable in a range of applications, from medical separation of signals to military radar applications. The key factor for development of a generic design is to determine the key component within a range of adaptive beamforming applications that would be consistent to some degree and could therefore be suitable to develop as an IP core. To widen the potential of the core, it will need to be able to support a varied range of specifications dealing with issues such as the following:

**Number of inputs:** A varying number of auxiliary and primary inputs need to be supported (Figure 11.3). The weights are calculated for a block of the input data coming from $N$ antennae and then applied to the same input data to generate the beamformer output for that block. For the final design, a more efficient post-processor is developed to extract the weights such as that described in Shepherd and McWhirter (1993).

**Supporting a range of FPGA/ASIC technologies:** By including some additional code and parameters the same core design can be re-targeted to a different technology. Doing this could enable a design to be prototyped on FPGA before targeting to ASIC.

**Support for performance criteria:** The variation in adaptive beamformer applications creates a wide span of desired features. For example, mobile communications power considerations and chip area could be the driving criteria, while for others a high data rate system could be the primary objective.

**Scalable architecture:** May need to be created to support a range of design criteria. Some key points driving the scalable architecture are desired data rate, area constraints, clock rate constraints and power constraints.

**Clock rate performance:** Depends on the architecture design and target technology chosen. Specifying the system requirements enables the designer to make a choice regarding the target technology and helps reach a compromise with other performance criteria such as power and area.

**Wordlength:** As different applications require different wordlengths, a range of wordlengths should be supported.

**Level of pipelining:** The desired clock rate may rely on pipelining within the design to reduce the critical path. Giving a choice of pipelining within the submodules of the design will greatly influence performance.

These values will form the basis from which to develop the adaptive beamformer solution from the generic architecture. The surrounding software models and testbenches should include the same level of scalability so as to complete the parameterization process.

### 11.2.2 Algorithm Development

The function of a typical adaptive beamformer is to suppress signals from every direction other than the desired "look direction" by introducing deep nulls in the beam pattern in the direction of the interference. The beamformer output is a weighted combination of signals received by a set of spatially separated antennae. An adaptive filtering algorithm calculating the filter weights is a central process of the adaptive beamforming application.

The aim of an adaptive filter is to continually optimize itself according to the environment in which it is operating. A number of mathematically and highly complex algorithms exist to calculate the filter weights according to an optimization criterion. Typically the target is to minimize an error function, which is the difference between a desired performance and the actual performance. Figure 11.4 highlights this process.

A great deal of research has been carried out into different methods for calculating the filter weights (Haykin 2002). The algorithms range in complexity and capability, and detailed analysis is required in order to determine a suitable algorithm. However there is no distinct technique for determining the optimum adaptive algorithm for a specific

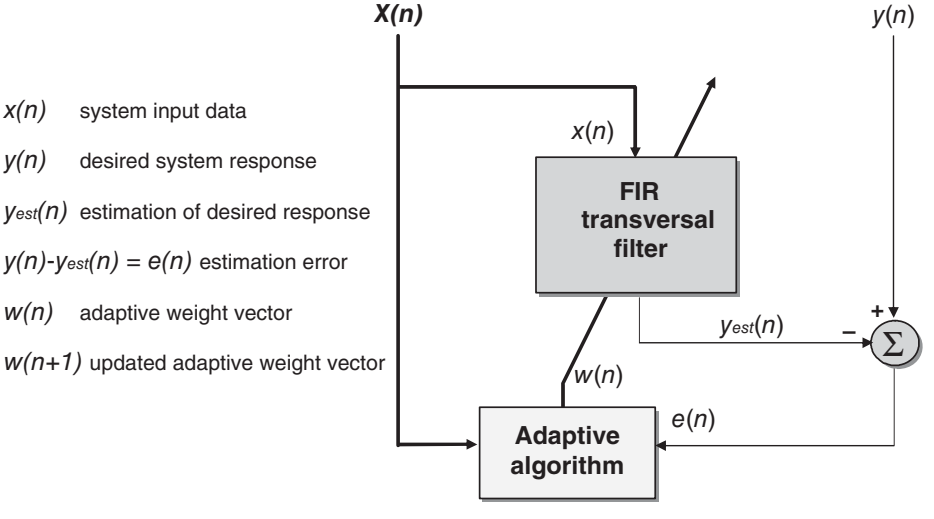



**Figure 11.4** Adaptive filter system

application. The choice comes down to a balance of the range of characteristics defining the algorithms, such as:

- rate of convergence, i.e. the rate at which the adaptive algorithm reaches an optimum solution;
- steady-state error, i.e. the proximity to an optimum solution;
- ability to track statistical variations in the input data;
- computational complexity;
- ability to operate with ill-conditioned input data;
- sensitivity to variations in the wordlengths used in the implementation.

As was discussed in Chapter 2, two methods for deriving recursive algorithms for adaptive filters use Wiener filter theory and the method of least squares, resulting in the LMS and the RLS algorithms, respectively. Whilst the LMS algorithm is simpler, its limitations lie in its sensitivity to the condition number of the input data matrix as well as slow convergence rates. In contrast, the RLS algorithm is more elaborate, offering superior convergence rates and reduced sensitivity to ill-conditioned data. On the negative side, the RLS algorithm is substantially more computationally intensive than the LMS equivalent, although it is preferred here.

In particular, the QR-RLS decomposition is seen as the algorithm for adaptively calculating the filter weights (Gentleman and Kung 1982; Kung 1988; McWhirter 1983). It reduces the computation order of the calculations and removes the need for a matrix inversion, giving a more stable implementation.

## 11.3 Algorithm to Architecture

A key aspect of achieving a high-performance implementation is to ensure an efficient mapping of the algorithm into hardware. This involves developing a hardware architecture in which independent operations are performed in parallel so as to increase the throughput rate. In addition, pipelining may be employed within the processor blocks to achieve faster throughput rates. One architecture that uses both parallelism and pipelining is the systolic array (Kung 1988). As well as processing speed, Chapter 13 highlights its impact on power consumption. The triangular systolic array (Figure 11.5), first introduced in Chapter 2, consists of two types of cells, referred to as BCs and ICs. Figure 11.6 illustrates the process from algorithm to architecture for this implementation.

It starts with the RLS algorithm solved by QR decomposition, shown as equations. The next stage depicts the RLS algorithm solved through QR decomposition using a sequential algorithm; at each iteration a new set of values are input to the equations, thus continuously progressing towards a solution. The new data are represented by $\underline{x}^T(n)$ and $y(n)$, where $x$ is the input data (auxiliary) matrix and $y$ is the desired (primary) data. The term $n$ represents the iteration of the algorithm. The QR operation can be depicted as a triangular array of operations. The data matrix is input at the top of the triangle and with each row another term is eliminated, eventually resulting in an upper triangular matrix.

The dependence graph (DG) in Figure 11.6 depicts this triangularization process. The cascaded triangular arrays within the diagram represent the iterations through time, i.e.

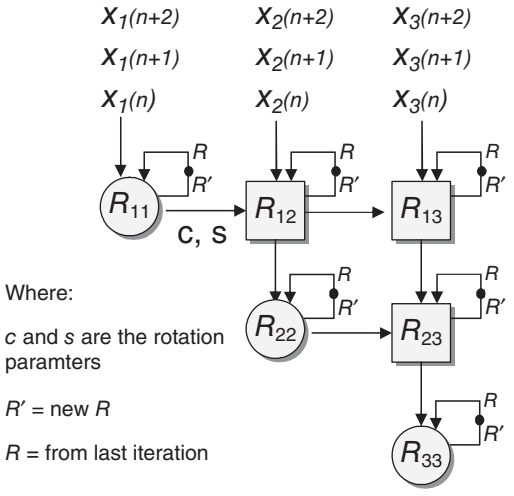



**Figure 11.5** Triangular systolic array for QRD RLS filtering

each one represents a new iteration. The arrows between the cascaded arrays highlight the dependency through time.

### 11.3.1 Dependence Graph

The dependencies between data can be identified in a DG. This allows the maximum level of concurrency to be identified by breaking the algorithm into nodes and arrows. The nodes outline the computations and the direction of the arrows shows the dependence of the operations. This is shown for the QR algorithm by the three-dimensional DG in Figure 11.7. The diagram shows three successive QR iterations, with arcs

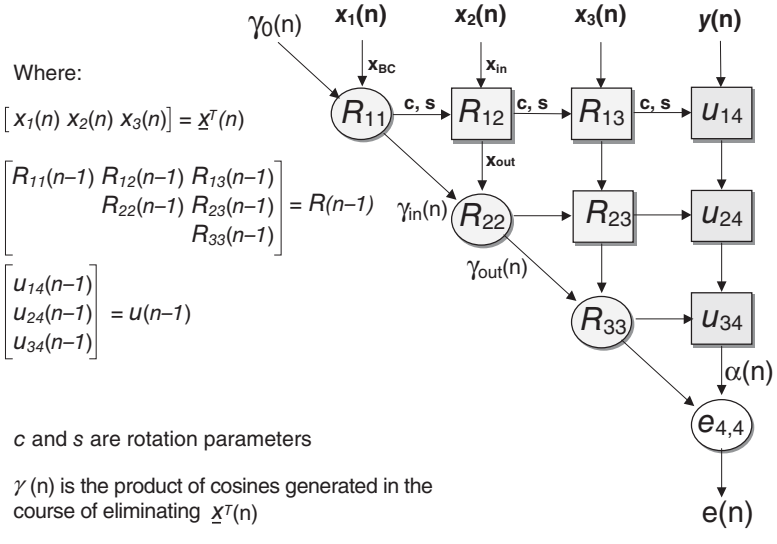



**Figure 11.6** From algorithm to architecture

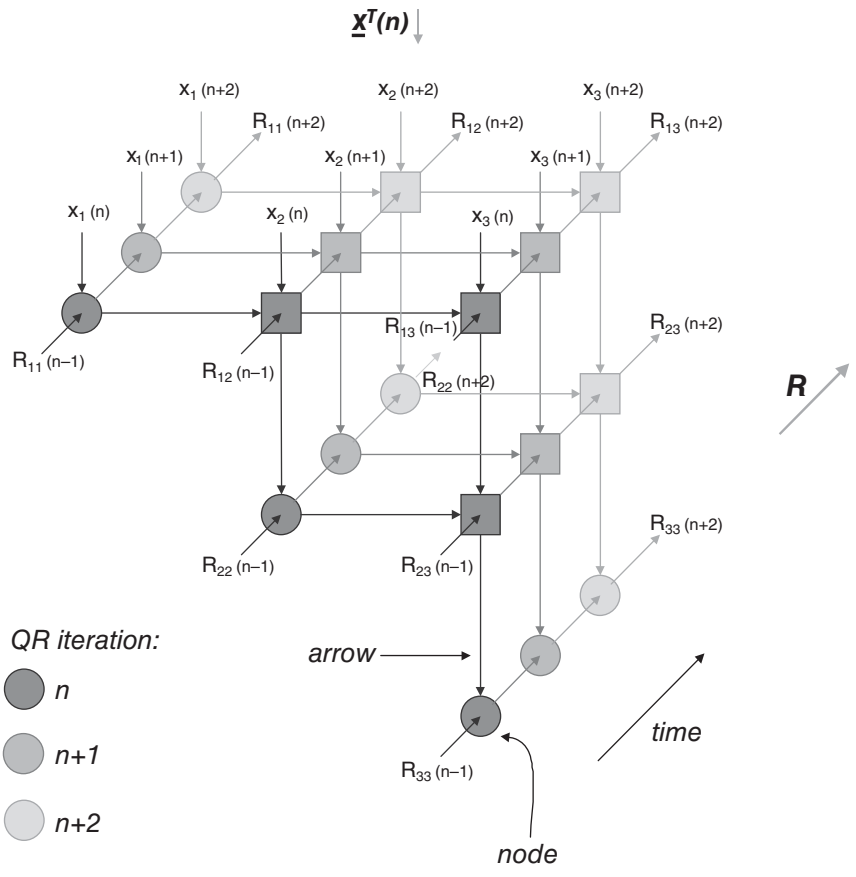



**Figure 11.7** Dependence graph for QR decomposition

connecting the dependent operations. Some of the variable labels have been omitted for clarity.

In summary, the QR array performs the rotation of the input $\underline{x}^T(n)$ vector with $R$ values held within the memory of the QR cells so that each input $x$ value into the BCs is rotated to zero. The same rotation is continued along the line of ICs via the horizontal arrows between QR cells. From this DG, it is possible to derive a number of SFG representations. The most obvious projection, which is used here, is to project the DG along the time (i.e. $R$) arrows.

### 11.3.2 Signal Flow Graph

The transition from DG to SFG is clearly depicted in Figure 11.8. To derive the SFG from the DG, the nodes of the DG are assigned to processors, and then their operations are scheduled on these processors. One common technique for processor assignment is linear projection of all identical nodes along one straight line onto a single processor, as indicated by the projection vector **d** in Figure 11.8. Linear scheduling is then used to determine the order in which the operations are performed on the processors.

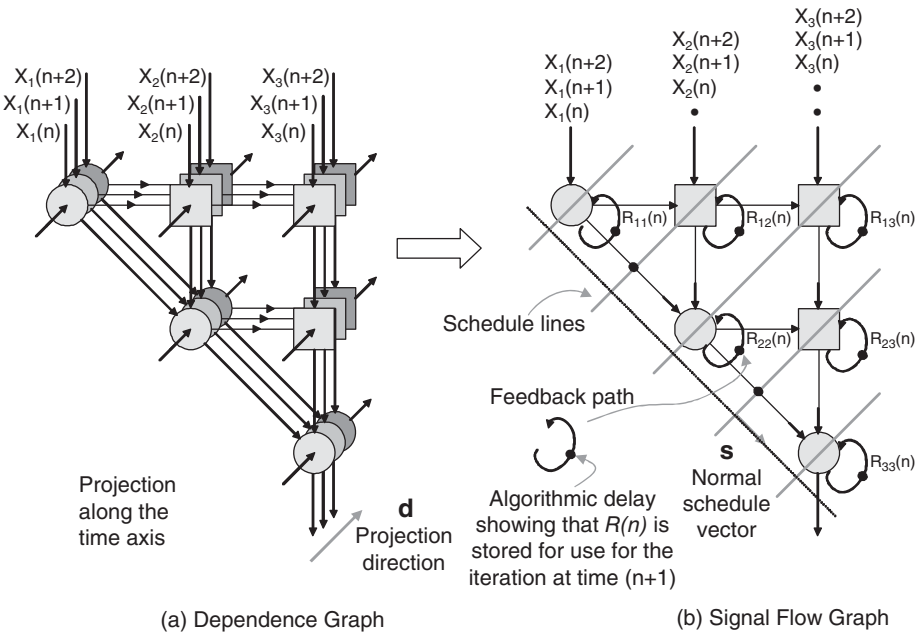



**Figure 11.8** From dependence graph to signal flow graph

The schedule lines in Figure 11.8 indicate the operations that are performed in parallel at each cycle. Mathematically they are represented by a schedule vector **s** normal to the schedule lines, which points in the direction of dependence of the operations. That is, it shows the order in which each line of operations is performed.

There are two basic rules that govern the projection and scheduling, and ensure that the sequence of operations is retained. Given a DG and a projection vector **d**, the schedule is permissible if and only if:

- all the dependence arcs flow in the same direction across the schedule lines;
- the schedule lines are not parallel with the projection vector **d**.

In the QR example in Figure 11.8, each triangular array of cells within the DG represents one QR update. When cascaded, the DG represents a sequence of QR updates. By projecting along the time axis, all the QR updates may be assigned onto a triangular SFG as depicted in Figure 11.8(b). In the DG, the *R* values are passed through time from one QR update to another, represented by the cascaded triangular arrays. This transition is more concisely represented by the loops in Figure 11.8(b), which feed the *R* values back into the cells via an algorithmic delay needed to hold the values for use in the next QR update. This is referred to as a recursive loop.

The power of the SFG is that it assumes that all operations performed within the nodes take one cycle, as with the algorithmic delays, represented by small black nodes, which are a necessary part of the algorithm. The result is a more concise representation of the algorithm than the DG.

The rest of this chapter gives a detailed account of the processes involved in deriving an efficient architecture and hence hardware implementation of the SFG representation

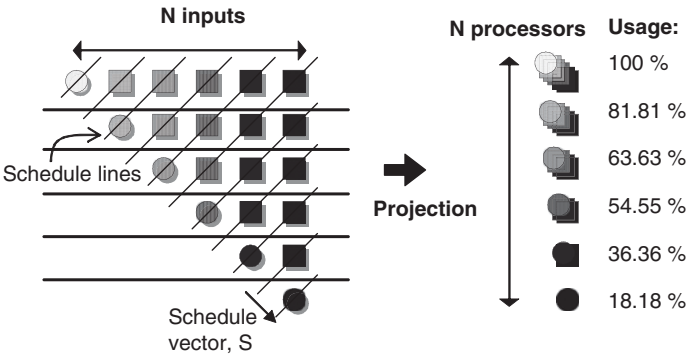



**Figure 11.9** Simple linear array mapping

of the algorithm. In particular, the emphasis is on creating an intuitive design that will be parameterizable, therefore enabling a fast development for future implementations.

## 11.4 Efficient Architecture Design

With the complexity of the SGR QR-RLS algorithm, coupled with the number of processors increasing quadratically with the number of inputs, it is vital to generate efficient QR array architectures tailored to the applications that meet desired performance with the lowest area cost. This is achievable by mapping the triangular functionality down onto a smaller array of processors. Deriving an efficient architecture for this QR array is complicated by its triangular shape and the position of the BCs along the diagonal. A simple projection of operations from left to right onto a column of $N$ processors leads to an architecture where the processors are required to perform both the IC and BC operations (which were described in Chapter 2). In addition, while the first processor is used 100% efficiently, this rate usage decreases down the column of processors such that the $N$th processor is only used once in every $N$ cycles. This results in an overall efficiency of about 60% as shown in Figure 11.9.

Rader (1992 1996) solved the issue of low processor usage by mirroring part B in the $x$-axis (see Figure 11.10) and then folding it back onto the rest of the QR array. Then, all the

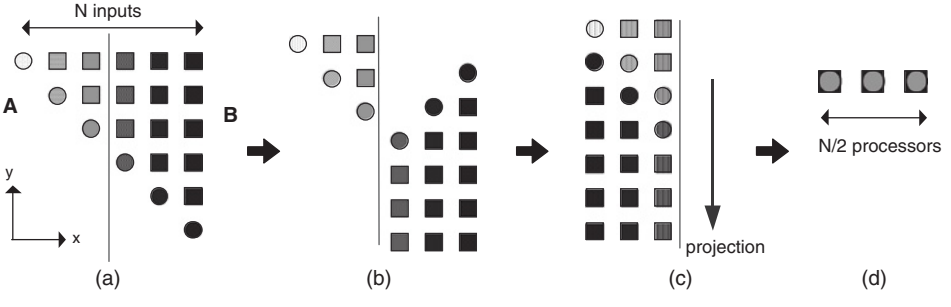



**Figure 11.10** Radar mapping (Rader 1992, 1996)

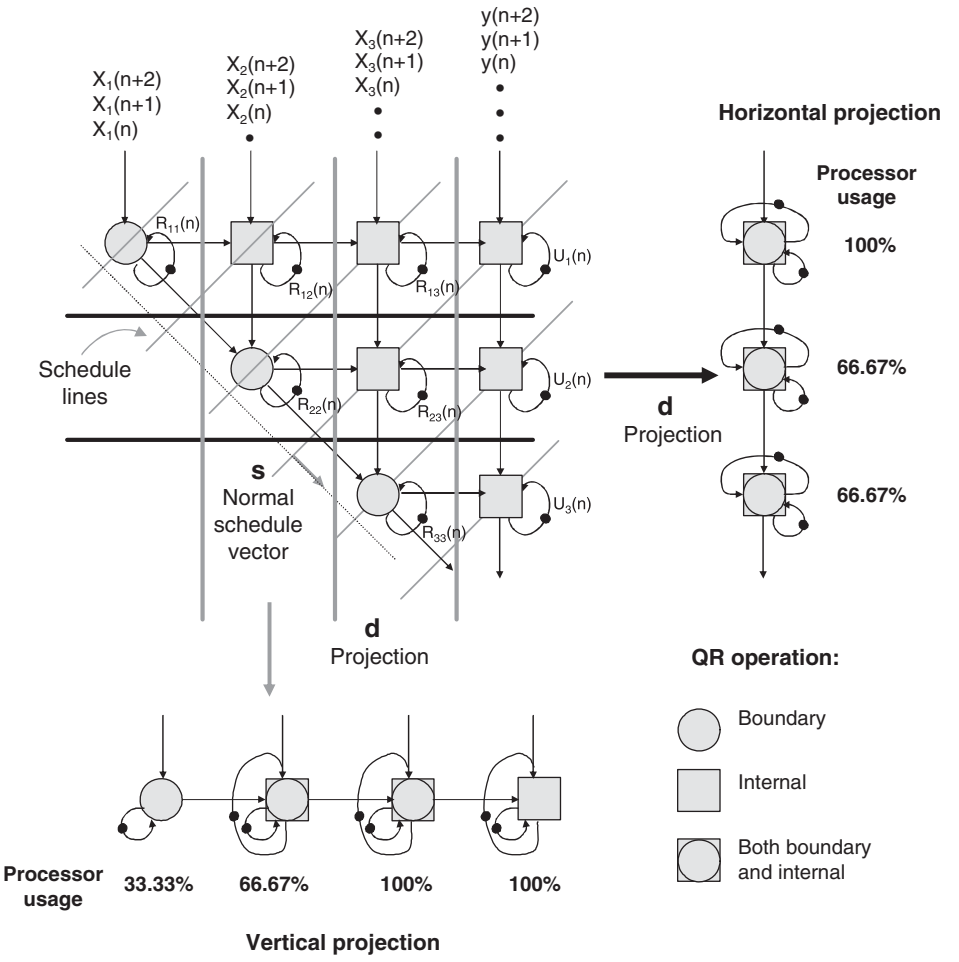



**Figure 11.11** Projecting the QR array onto a linear architecture

operations were mapped down onto a linear architecture of $N/2$ processors. This works quite effectively but the BC and IC operations still need to be performed on the same processor, involving the design of a generic cell architecture or an implementation based on the CORDIC algorithm, see Hamill (1995). Another solution (Tamer and Ozkurt 2007) used a tile structure on which to map the QR cells.

Figure 11.11 gives an example of how the QR operations need to be scheduled. It shows a simplified QR array with just three auxiliary ($x$) inputs and one primary ($y$) input. The schedule lines show the sequence in which the QR operations need to be performed due to the dependence on variables passing between the cells. On each schedule line, there are a number of operations that can be performed at the same time. The normal schedule vector, **s**, then depicts the order of the operations, that is, the order of the schedule lines. Two examples are given for the projection vector, **d**. There is a horizontal projection of the QR operations onto a column of three processors. Likewise, a vertical

projection is possible down to four processors. As with the example above, the resulting architectures require that both the BC and IC operations are performed on the same processor.

Another mapping (Walke 1997) solves this issue of requiring QR cells that perform both operations. The mapping assigns the triangular array of $2m^2 + 3m + 1$ cells (i.e. $N = 2m + 1$ inputs) onto a linear architecture consisting of one BC processor and $m$ IC processors. It folds and rotates the triangular array so that all the BC operations may be assigned to one processor, while all the IC operations are implemented on a row of separate processors. All processors in the resulting linear architecture are locally interconnected and used with 100% efficiency, thus displaying the characteristics of a systolic array and hence offering all the advantages associated with these structures. This procedure is depicted in Figure 11.12 for a seven-input triangular array (for a more detailed description, see Lightbody 1999; Lightbody *et al.* 2003; Walke 1997).

For clarity, each QR operation is assigned a coordinate originating from the $R$ (or $U$) term calculated by that operation, i.e. the operation $R_{1,2}$ is denoted by the coordinate 1, 2, and $U_{1,7}$ is denoted by 1, 7. To simplify the explanation, the multiplier at the bottom of the array is treated as a BC, denoted by 7, 7.

The initial aim of mapping a triangular array of cells down onto a smaller architecture is to maneuver the cells so that they form a locally interconnected regular rectangular array. This can then be partitioned evenly into sections, each to be assigned to an individual processor. This should be done in such a way as to achieve 100% cell usage and a nearest neighbor connected array. Obtaining the rectangular array is achieved through the following four stages. The initial triangular array is divided into two smaller triangles, $A$ and $B$. A cut is then made after the $(m + 1)$th BC at right angles to the diagonal line of BCs (Figure 11.12(a)). Triangle $A$ forms the bottom part of a rectangular array, with $m + 1$ columns and $m + 1$ rows.

Triangle $B$ now needs to be manipulated so that it can form the top part of the rectangular array. This is done in two stages. By mirroring triangle $B$ first in the $x$-axis, the BCs are aligned in such a way that they are parallel to the BCs in the triangle $A$, forming a parallelogram, as shown in Figure 11.12(b). The mirrored triangle $B$ is then moved up along the $y$-axis and left along the $x$-axis to above $A$ forming the rectangular array (Figure 11.12(c)). As depicted, the BC operations are aligned down two columns and so the rectangular array is still not in a suitable format for assigning operations onto a linear architecture.

The next stage aims to fold the large rectangular array in half so that the two columns of BC operations are aligned along one column. This fold interleaves the cells so that a compact rectangular processor array (Figure 11.12(d)) is produced. From this rectangular processor array, a reduced architecture can be produced by projection down the diagonal onto a linear array, with all the BC operations assigned to one BC processor and all the IC operations assigned to a row of $m$ IC processors (Figure 11.12(e)). The resulting linear architecture is shown in more detail in Figure 11.13.

The lines drawn through each row of processors in Figure 11.12(e) (labeled $1, \ldots, 7$), represent the set of QR operations that are performed on each cycle of the linear array. They are used to derive the schedule for architecture, as denoted more compactly by a schedule vector $\mathbf{s}$, normal to the schedule lines. In Figure 11.13, it is assumed that registers are present on all processor outputs to maintain the data between the cycles of the schedule. Multiplexers are present at the top of the array so that system inputs to the

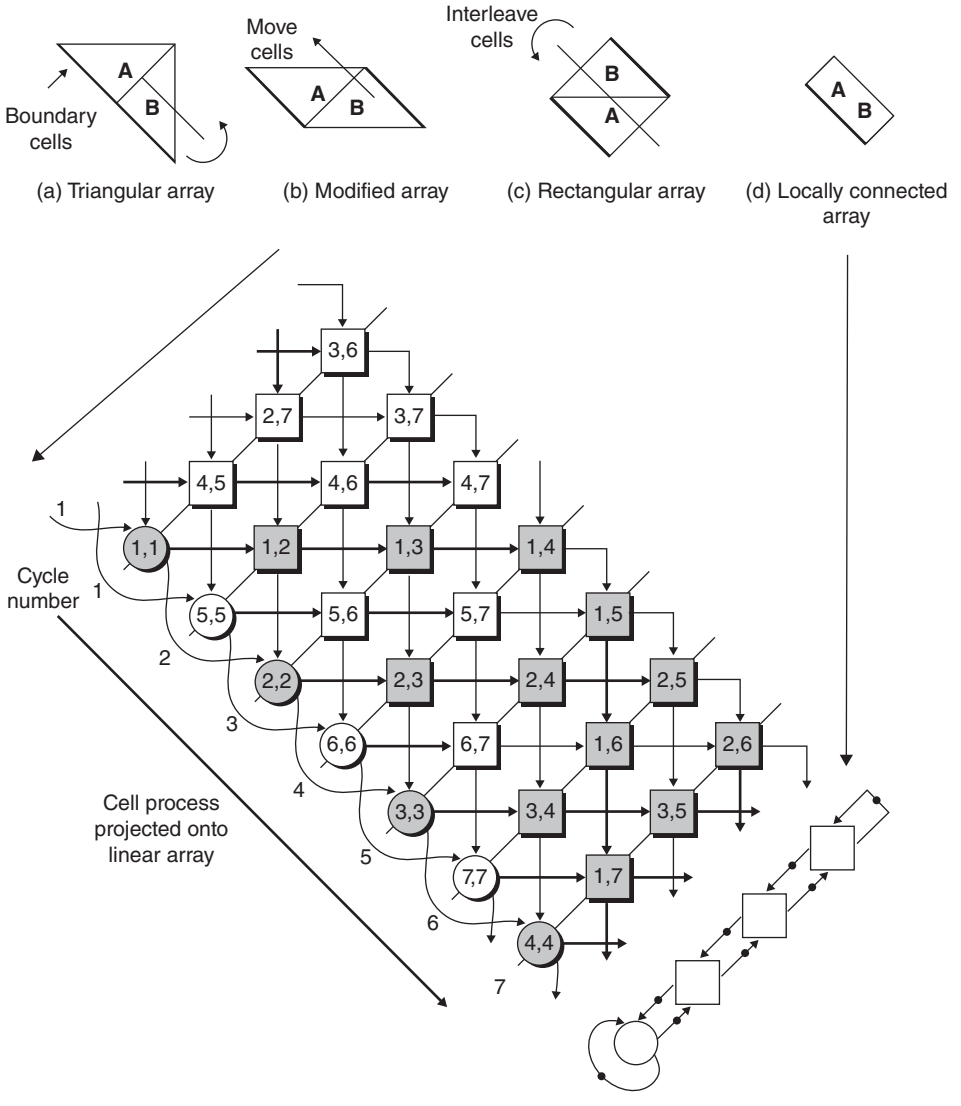



**Figure 11.12** Interleaved processor array

QR array can be supplied to the cells at the right instance in time. The linear array has only local interconnections, so all the cell inputs come from adjacent cells. The bottom multiplexers govern the different directions of dataflow that occur between rows of the original array.

The folding of the triangular QR array onto an architecture with reduced number of processors means that the $R$ values need to be stored for more than one clock cycle. They are held locally within the recursive data paths of the QR cells, rather than external

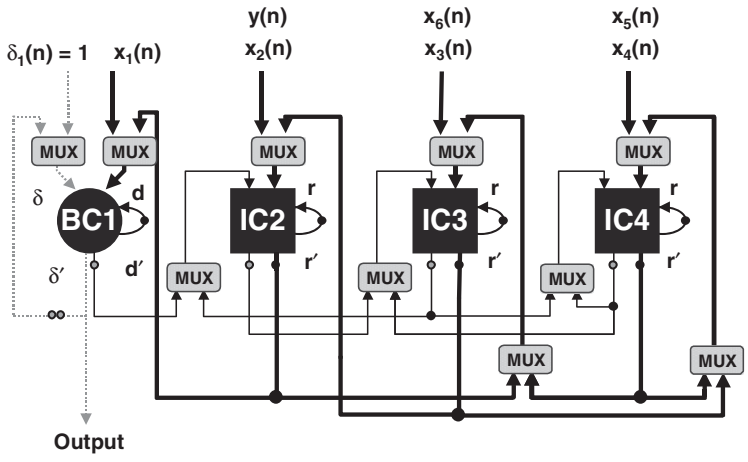



**Figure 11.13** Linear architecture for a seven-input QR array

memory (i.e. the values are pipelined locally to delay them until they are needed). Some of the required delays are met by the latency of existing operations within the loop and the remainder are achieved by inserting additional registers.

### 11.4.1 Scheduling the QR Operations

The derivation of the architecture is only a part of the necessary development as a valid schedule needs to be determined to ensure that the data required by each set of operations are available at the time of execution. This implies that the data must flow across the schedule lines in the direction of the schedule vector. The rectangular processor array in Figure 11.12(d) contains all the operations required by the QR algorithm, showing the sequence in which they are to be implemented on the linear architecture. Therefore, this diagram can be used to show the schedule of the operations to be performed on the linear architecture.

An analysis of the scheduling and timing issues can now be refined. Looking at the first schedule line, it can be seen that operations from two different QR updates have been interleaved. The shaded cells represent the current QR update at time $n$ and the unshaded cells represent the previous unfinished update at time $n - 1$. Effectively the QR updates have been interleaved. This is shown in more clarity in Figure 11.14. The first QR operation begins at cycle 1, then after $2m + 1$ cycles of the linear architecture the next QR operation begins. Likewise, after a further $2m + 1$ cycles the third QR operation is started. In total, it takes $4m + 1$ cycles of the linear architecture to complete one specific QR update.

From Figure 11.14, it can be seen that the $x$ inputs into the QR cells come from either external system data, i.e. from the snapshots of data forming the input $x(n)$ matrix and $y(n)$ vector, or internally from the outputs of other processors. The external inputs are fed into the linear architecture every $2m + 1$ clock cycles.

If each QR cell takes a single clock cycle to produce an output, then there will be no violation of the schedule shown in Figure 11.12. However, additional timing issues must be taken into account as processing units in each QR cell have detailed timing

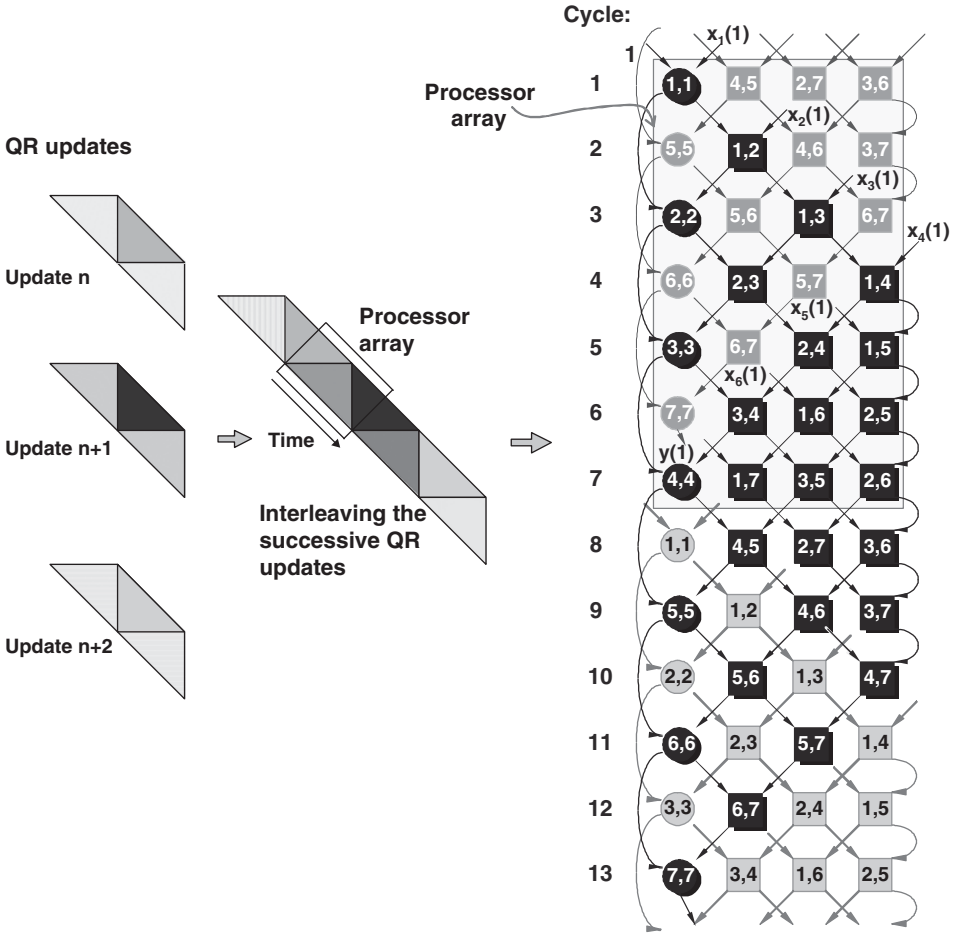



**Figure 11.14** Interleaving successive QR operations. (Source: Lightbody 2003. Reproduced with permission of IEEE.)

requirements. The retiming of the operations is discussed in more detail later on in this chapter.

Note that the processor array highlighted in Figure 11.14 is equivalent to the processor array given in Figure 11.12(d). This processor array is the key starting point from which to develop a generic QR architecture.

## 11.5 Generic QR Architecture

The technique shown so far was applied to a QR array with only one primary input. More generally, the QR array would consist of a triangular part and a rectangular part (Figure 11.15(a)), the sizes of which are determined by the number of auxiliary and primary inputs, respectively. Typically, the number of inputs to the triangular part is at least a

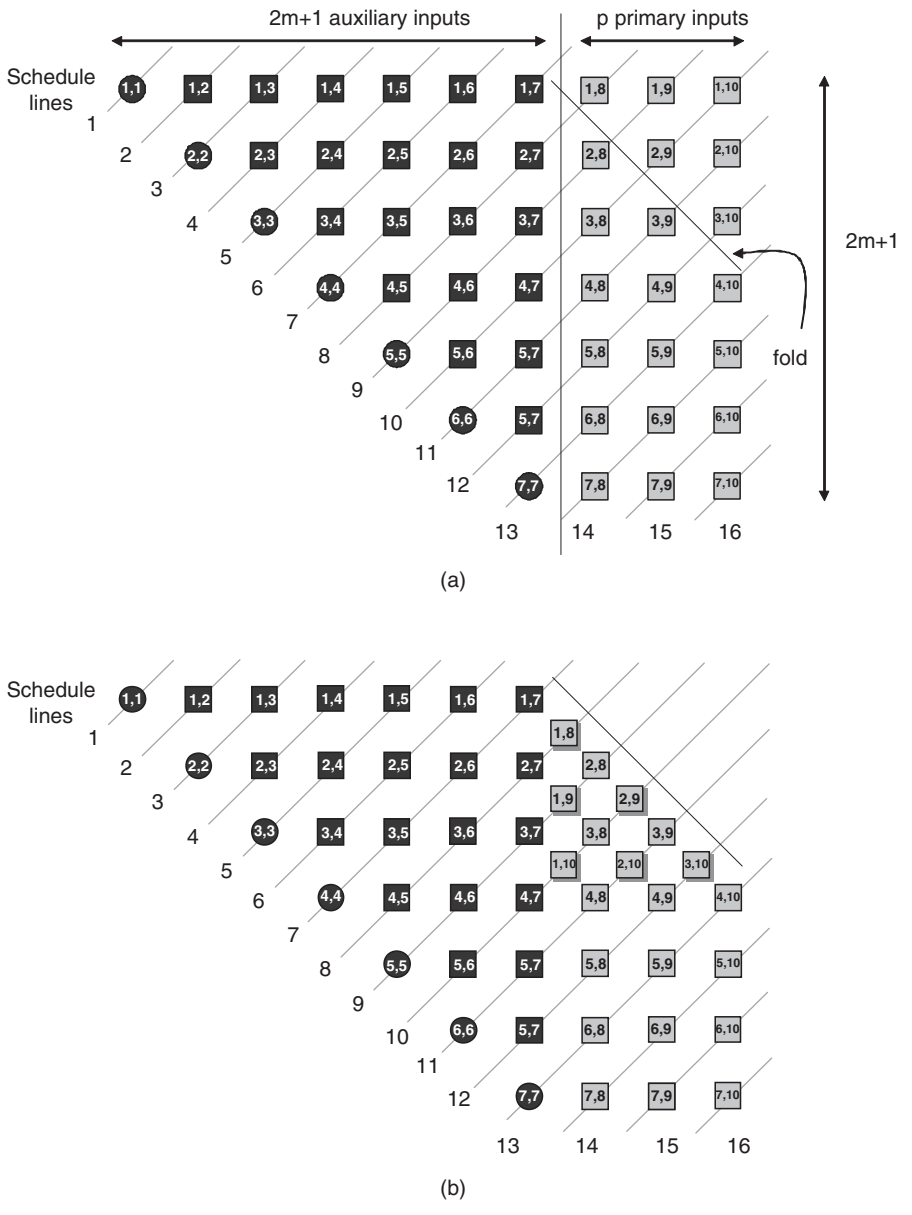



**Figure 11.15** Generic QR array. (Source: Lightbody 2003. Reproduced with permission of IEEE.)

factor greater than the number of inputs to the rectangular part, with example numbers for radar being 40 inputs for the triangular part and only 2 for the rectangular part.

The mapping procedure presented in this section implements both the triangular and rectangular components of the QR array in a single architecture. As before, the BC and IC operations are kept to two distinct processors. The additional factor presented by the generic mapping technique is that a choice of linear or rectangular architectures is

available. The number of IC processors may be reduced further, allowing more flexibility in the level of hardware reduction. However, at least one BC processor is required, even if the number of ICs is reduced below one row. Note that the connections have been removed from Figure 11.15 and in later following diagrams in order to reduce the complexity of the diagram and aid clarity.

### 11.5.1 Processor Array

In the previous section, the triangular structure of the QR array was manipulated into a rectangular processor array of locally interconnected processors, as shown in Figure 11.12(d). From this starting point, the operations can be mapped onto a reduced architecture. A simplified method for creating the processor array is demonstrated in the following example.

The processor array is obtained through two steps. Firstly, a fold is made by folding over the corner of the array after the *m*th cell from the right-hand side, as depicted in Figure 11.15. The cells from the fold are interleaved between the rows of unfolded cells as shown. The next stage is to remove the gaps within the structure by interleaving successive QR updates in the same manner as shown in Figure 11.14. The choice of position of the fold and the size of the triangular part of the array are important. By placing the fold after the *m*th cell from the right-hand side, a regular rectangular array of operations can be produced.

This is shown in greater detail in the Figure 11.16, which shows that there is a section which repeats over time and contains each of all the required QR operations. This section is referred to as the processor array. It is more clearly depicted in Figure 11.17, which shows just the repetitive section from Figure 11.16.

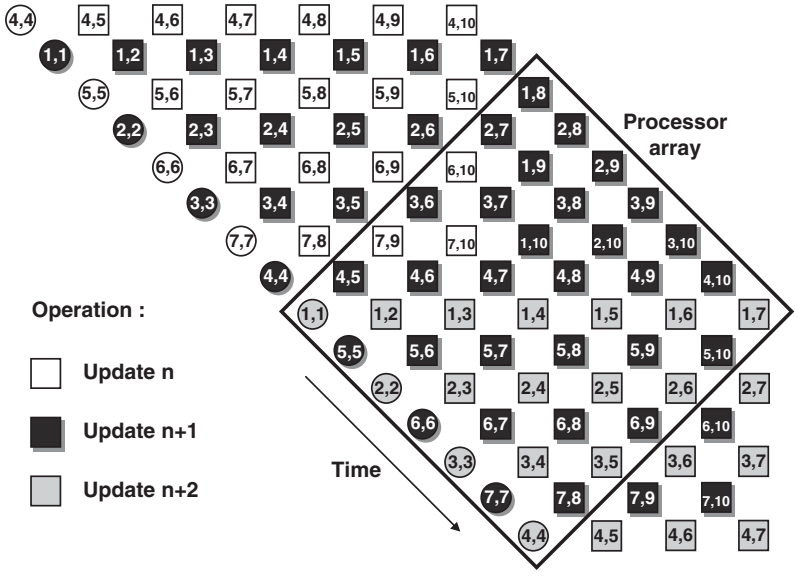



**Figure 11.16** Repetitive section

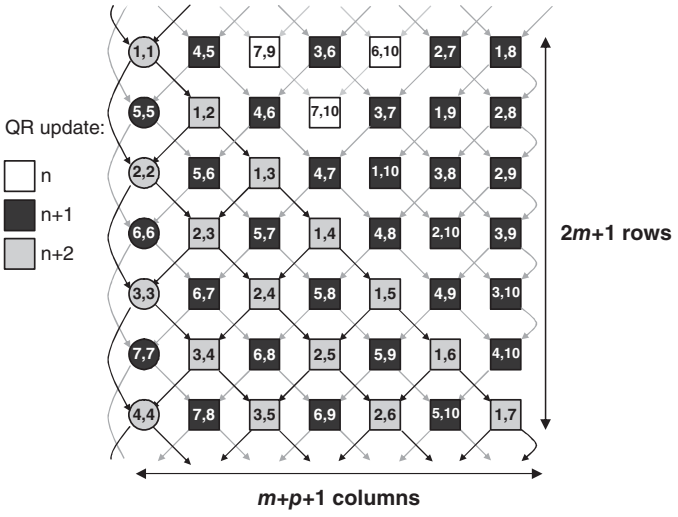



**Figure 11.17** Processor array. (Source: Lightbody 2003. Reproduced with permission of IEEE.)

In this example, the processor array contains QR operations built up from three successive QR updates, represented by the differently shaded cells. The interconnections are included within this diagram, showing that all cells are locally connected. The size of the processor array is determined by the original size of the triangular QR array, which in turn is governed by the number of auxiliary and primary inputs, $2m + 1$ and $p$, respectively. The resulting processor array has $2m + 1$ rows and $m + p + 1$ columns. As expected, the product of these two values gives the number of operations within the original QR array. From the processor array, a range of architectures with reduced number of processors can be obtained by dividing the array into partitions and then assigning each of the partitions to an individual processor. There are several possible variants of QR architecture:

**Linear architecture:** The rectangular array is projected down onto a linear architecture with one BC and $m + p$ ICs.

**Rectangular architecture:** The rectangular array is projected down onto a number of linear rows of cells. The architecture will have $r$ rows (where $1 < r \leq 2m + 1$), and each row will have one BC and $m + p$ ICs.

**Sparse linear architecture:** The rectangular array is projected down onto a linear architecture with one BC and less than $m + p$ ICs.

**Sparse rectangular architecture:** The rectangular array is projected down onto a number of linear rows of cells. The architecture will have $r$ rows (where $1 < r \leq 2m + 1$), and each row will have one BC and less than $m + p$ ICs.

### Linear Array

The linear array is derived by assigning each column of operations to an individual processor, as shown in Figure 11.18. In total, it takes $4m + p + 1 = 16$ cycles of the linear array to complete each QR operation. In addition, there are $2m + 1 = 7$ cycles between the start of successive QR updates. This value is labeled as $T_{\mathrm{QR}}$. Note that so far the

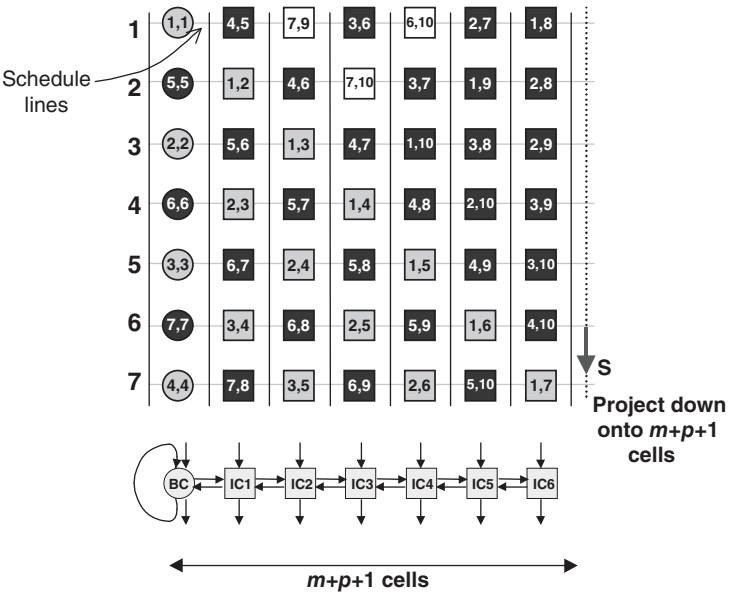



**Figure 11.18** Linear array. (Source: Lightbody 2003. Reproduced with permission of IEEE.)

latency of the QR cells is considered to be one clock cycle, i.e. on each clock cycle one row of QR operations is performed on the linear architecture. Later sections will examine the effect of a multi-cycle latency, which occurs when cell processing elements with detailed timings are used in the development of the generic QR architecture.

### Sparse Linear Array

A further level of hardware reduction is given in Figure 11.19, resulting in a sparse linear array. Here the number of IC processors has been halved. When multiple columns (i.e. $N_{IC}$ columns) of IC operations are assigned to each processor then the number of iterations of the architecture is increased by this factor. Hence, for the sparse linear array, $T_{QR}$ is expressed as the product of $2m + 1$ (used in the linear array) and $N_{IC}$. The schedule for the sparse linear array example is illustrated in Figure 11.20.

### Rectangular Array

The processor array can be partitioned by row rather than by column so that a number of rows of QR operations are assigned to a linear array of processors. The example below shows the processor array mapped down on an array architecture. As the processor array consisted of 7 rows, 4 are assigned to one row and 3 are assigned to the other. To balance the number of rows for each linear array, a dummy row of operations is needed and is represented by the cells marked by the letter *D*.

On each clock cycle, the rectangular array processor executes two rows of the original processor array. Each QR iteration takes 18 cycles to be completed, two more clock cycles than for the linear array due to the dummy row of operations. However, the QR

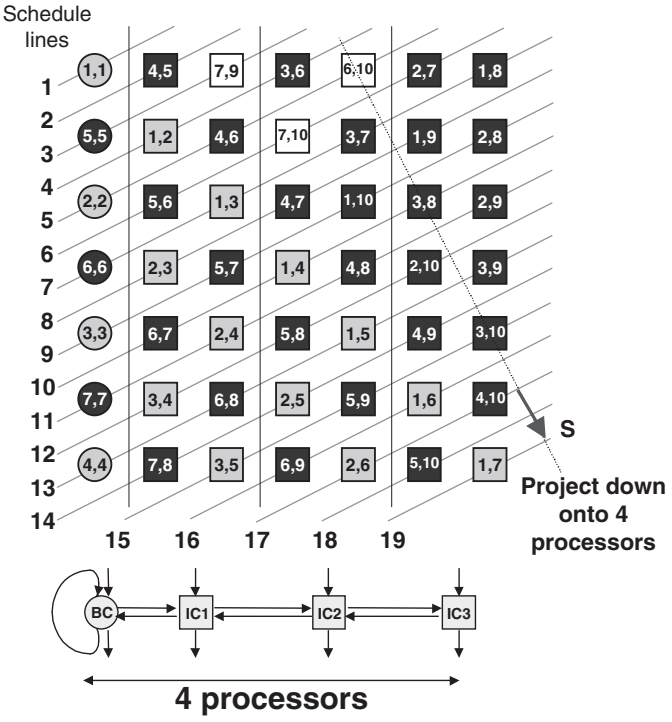



**Figure 11.19** Sparse linear array

updates are started more frequently. In this case $T_{\text{QR}}$ is 4, compared to the linear array which took 7 cycles. For the array architecture, $T_{\text{QR}}$ is determined by

$$T_{\text{QR}} = \frac{(2m+1) + N_D}{N_{\text{rows}}},$$

where $N_{\text{rows}}$ is the number of lines of processors in the rectangular architecture, and $N_D$ is the number of rows of dummy operations needed to balance the schedule. The resulting value relates to the number of cycles of the architecture required to perform all the operations within the processor array.

### Sparse Rectangular Array

The sparse rectangular array assigns the operations to multiple rows of sparse linear arrays. A number of rows of the processor array are assigned to each linear array. The columns are also partitioned so that multiple columns of operations are assigned to each IC processor, as shown in Figure 11.22.

The QR update takes 34 cycles for completion and each update starts every 7 cycles, i.e. $T_{\text{QR}} = 7$. Including the term $N_{\text{IC}}$, the equation for $T_{\text{QR}}$ becomes

$$T_{\text{QR}} = \frac{((2m+1) + N_D)N_{\text{IC}}}{N_{\text{rows}}}.$$

For example, $T_{\text{QR}} = ((2 \times 3 + 1 + 0) \times 2)/2 = 7$ cycles.

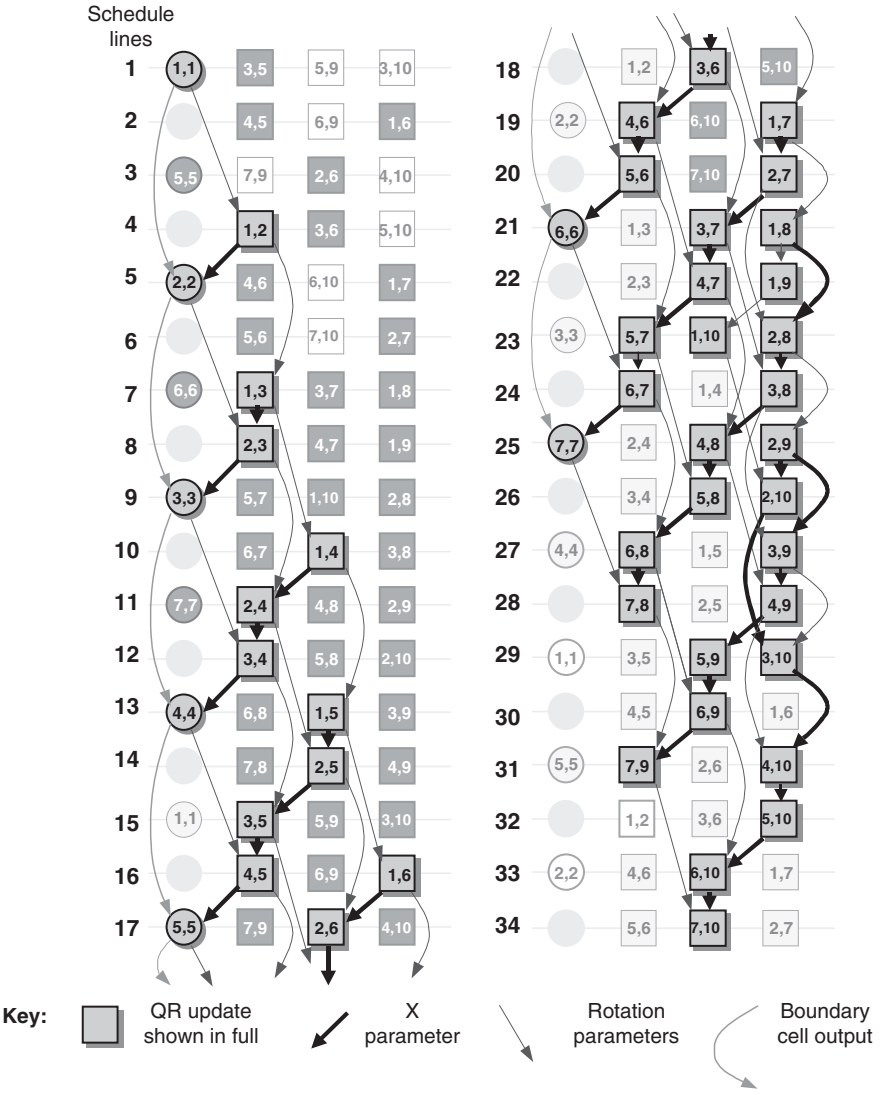



**Figure 11.20** One QR update scheduled on the sparse linear array. (Source: Lightbody 2003. Reproduced with permission of IEEE.)

The discussion to date has concentrated on mapping QR arrays that have an odd number of auxiliary inputs. The technique can be applied to an array with an even number with a slight reduction in overall efficiency.

## 11.6 Retiming the Generic Architecture

The QR architectures discussed so far have assumed that the QR cells have a latency of one clock cycle. The mapping of the architectures is based on this factor; hence there will be no conflicts of the data inputs. However, the inclusion of actual timing details

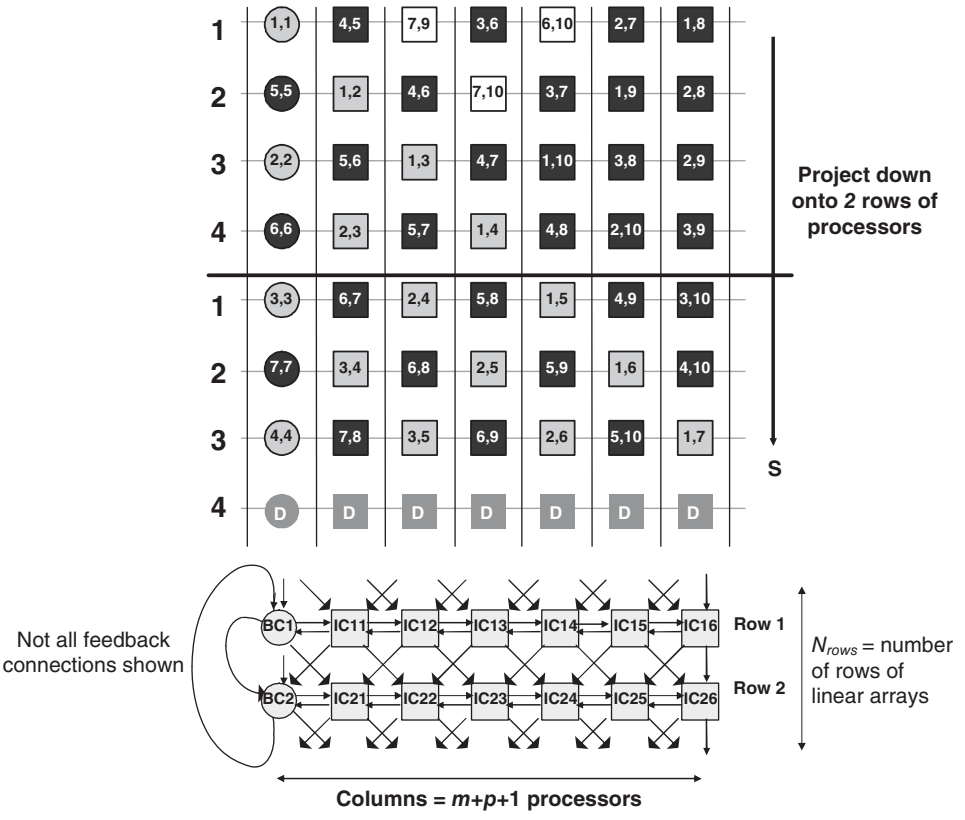



**Figure 11.21** Rectangular array

within the QR cells will affect this guarantee of a valid data schedule. The arithmetic IP processors (McCanny *et al.* 1997; Northeastern University 2007), used to implement the key arithmetic functions such as multiplication, addition and division involve timing details which will impact the overall circuit timing. Embedding processor blocks with specific timing information, coupled with the impact of truncation and internal word growth, means that detailed retiming of the original SFGs of the QR cells must be performed before the processors can be used to implement the QR architecture (Trainor *et al.* 1997). The overall effect of retiming is to incur variable latencies in the output data paths of the QR cells. The effect of real timing information within the QR cells is discussed in this section.

The choice for the QR array was to use floating-point arithmetic to support the dynamic range of the variables within the algorithm. The floating-point library used supported variable wordlengths and levels of pipelining, as depicted in Figure 11.23.

In adaptive beamforming, as with many signal processing applications, complex arithmetic representations are needed as incoming signals contain a magnitude and phase component. This is implemented using one signal for the real part and another for the imaginary part, and gives the BC and IC operations shown in the SFGs depicted in Figure 11.24.

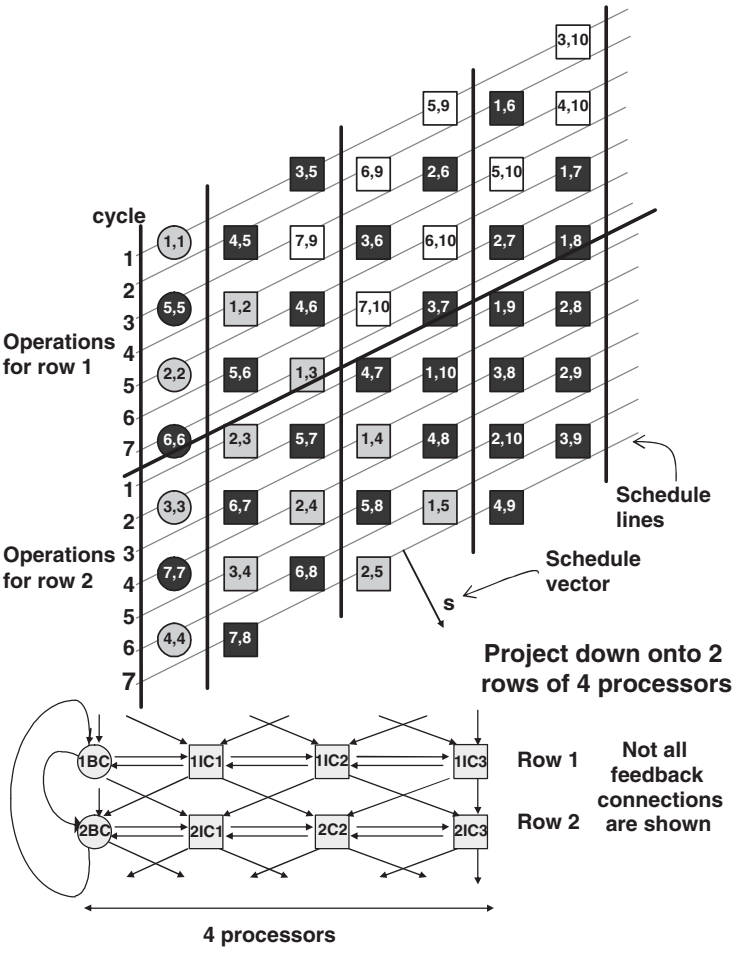



**Figure 11.22** Sparse rectangular array. (Source: Lightbody 2003. Reproduced with permission of IEEE.)

The floating-point complex multiplication is built up from four real multiplications and two real additions: $(a + jb)(c + jd) = (ac - bd) + j(ad + bc)$. An optimization is available to implement the complex multiplication using three multiplications and fives additions/subtractions as illustrated in Section 6.2.2. However, given that an addition is of a similar area to multiplication within floating-point arithmetic due to the costly exponent calculation, this is not beneficial. For this reason, the four-multiplication version is used. The detail of the complex arithmetic operations is given in Figure 11.25.

The SFGs for the BCs and ICs are given in Figures 11.26 and 11.27, respectively. These diagrams show the interconnections of the arithmetic modules within the cell architectures. Most functions are self-explanatory, except for the shift-subtracter. For small values of $x$, the operation $\sqrt{1-x}$ can be approximated by $1 - x^2$ which may be implemented by a series of shifts denoted by $D = A - \text{Shift}(A, N)$. This operation is used to implement the forgetting factor, $\beta$, within the feedback paths of the QR cells. This value, $\beta$, is close to 1, therefore $x$ is set to $1 - \beta$ for the function application.

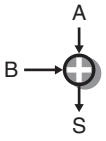
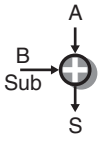
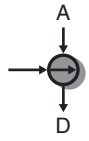
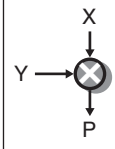
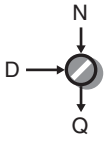
| Floating point block | Add | Sub Add | Shft Sub | Mult | Div | Round |
|---|---|---|---|---|---|---|
| **Function** | Addition S = A+B | Adder/sub S = A+B when Sub = 0 else S = A–B | Shift-subtractor: *D* = *A*–*Shift* (*A*, *N*). | Multiplier: *P* = *X*×*Y* | Divider: *Q* = *N*/*D* | Rounder |
| **Symbol** |  |  |  |  |  |  |
| **Latency** | 0 – 3 | 0 – 3 | 0 – 1 | 0 – 2 | 0 – Mbits+1 | 0 – 1 |
| **Label for Latency** | $\mathbf{P_A}$ | $\mathbf{P_A}$ | $\mathbf{P_S}$ | $\mathbf{P_M}$ | $\mathbf{P_D}$ | $\mathbf{P_R}$ |

**Figure 11.23** Arithmetic modules. (Source: Lightbody 2003. Reproduced with permission of IEEE.)

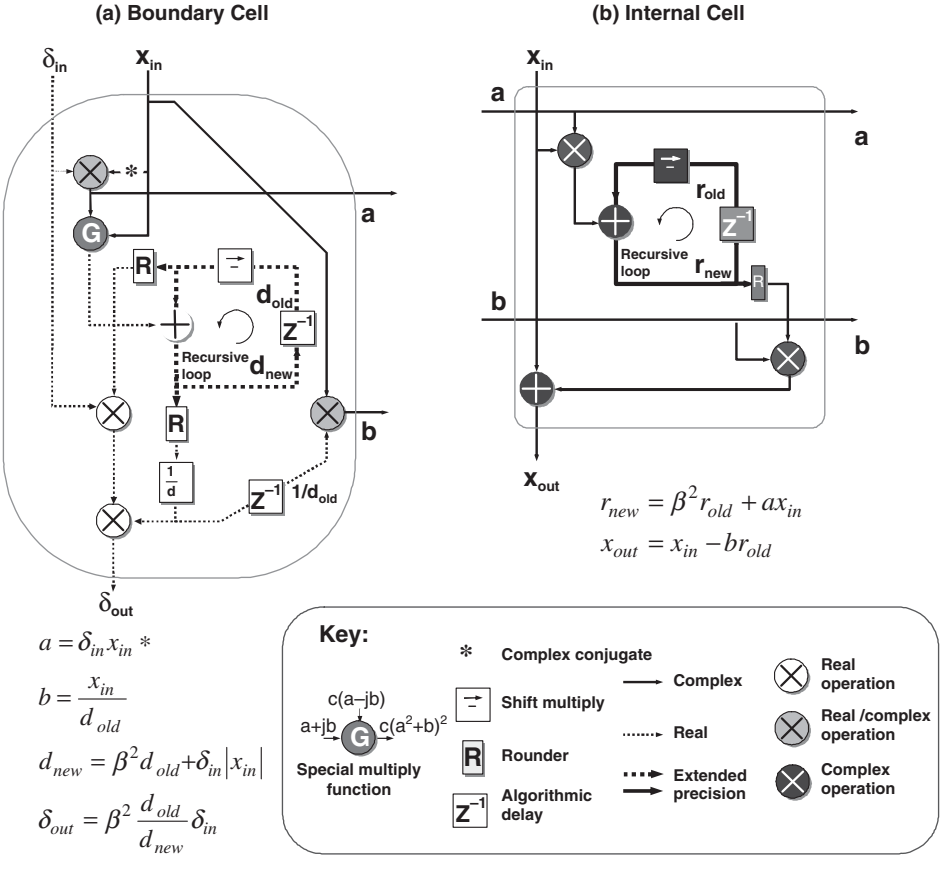



**Figure 11.24** Cell SFGs for the complex arithmetic SGR QR algorithm. (Source: Lightbody 2003. Reproduced with permission of IEEE.)

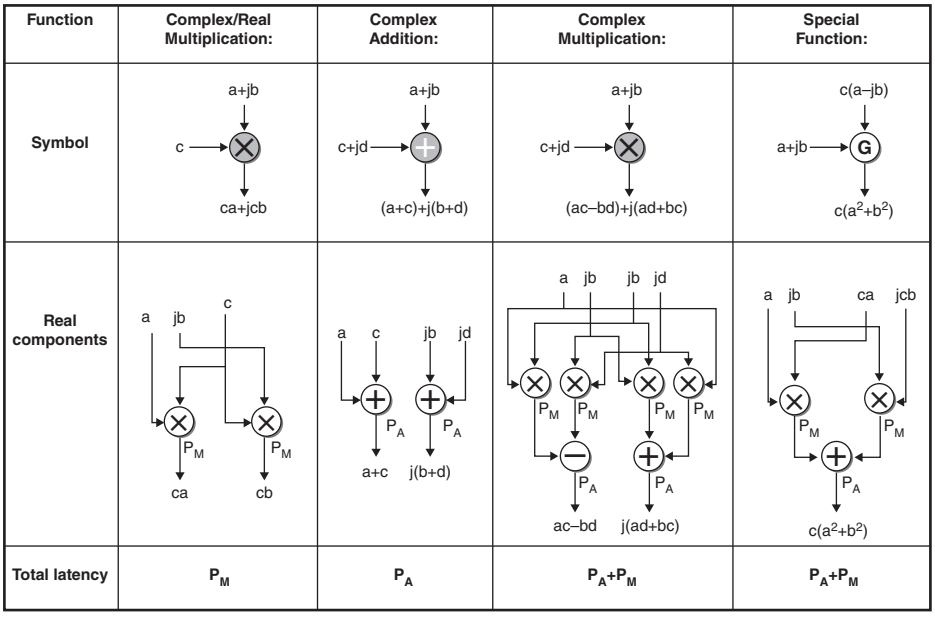



**Figure 11.25** Arithmetic modules. (Source: Lightbody 2003. Reproduced with permission of IEEE.)

There are a number of feedback loops within the QR cells, as shown in Figures 11.26 and 11.27. These store the $R$ values from one RLS iteration to the next. These loops will be a fundamental limitation to achieving a throughput rate that is close to the clock rate and, more importantly, could lead to considerable inefficiency in the circuit utilization. In other words, even when using a full QR array, the delay in calculating the new $R$ values will limit the throughput rate.

Figures 11.26 and 11.27 show the QR cell descriptions with generic delays placed within the data paths. These are there to allow for the re-synchronization of operations due to the variable latencies within the arithmetic operators, i.e. to ensure correct timing. The generic expressions for the programmable delays are listed in Tables 11.1 and 11.2 for the BC and IC, respectively.

Secondly, to maintain a regular data schedule, the latencies of the QR cells are adjusted so that the $x$ values and rotation parameters are output from the QR cells at the same time. The latency of the IC in producing these outputs can be expressed generically using a term $L_{\mathrm{IC}}$. The latencies of the BC in producing the rotation parameters, $a$ and $b$, are also set to $L_{\mathrm{IC}}$ to keep outputs synchronized. However, the latency of the BC in producing the $\delta_{\mathrm{out}}$ is set to double this value, $2L_{\mathrm{IC}}$, as this relates back to the original scheduling of the full QR array, which showed that no two successive BC operations are performed on successive cycles. By keeping the structure of the data schedule, the retiming process comes down to a simple relationship.

### 11.6.1 Retiming QR Architectures

This subsection continues with the discussion of retiming issues and how to include them in a generic architecture.

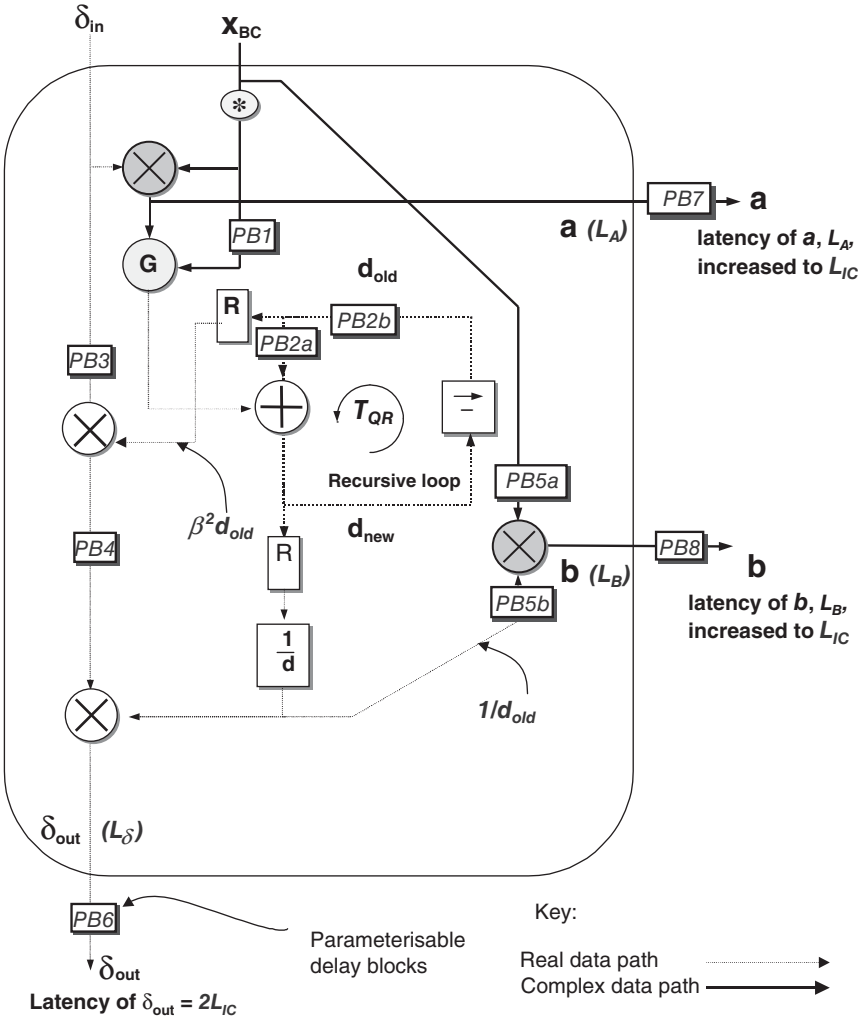



**Figure 11.26** Generically retimed BC

### Retiming of the Linear Array Architecture

The latency has the effect of stretching out the schedule of operations for each QR update. This means that iteration $n = 2$ begins $2m + 1$ clock cycles after the start of iteration $n = 1$. However, the introduction of processor latency stretches out the scheduling diagram such that iteration $n = 2$ begins after $(2m + 1)L_{\mathrm{IC}}$ clock cycles. This is obviously not an optimum use of the linear architecture as it would only be used every $L_{\mathrm{IC}}$th clock cycle. A factor, $T_{\mathrm{QR}}$, was introduced in the previous section as the number of cycles between the start of successive QR updates, as determined by the level of hardware reduction.

It can be shown that a valid schedule which results in a 100% utilization can be achieved by setting the latency $L_{\mathrm{IC}}$ to a value that is relatively prime to $T_{\mathrm{QR}}$. That is,

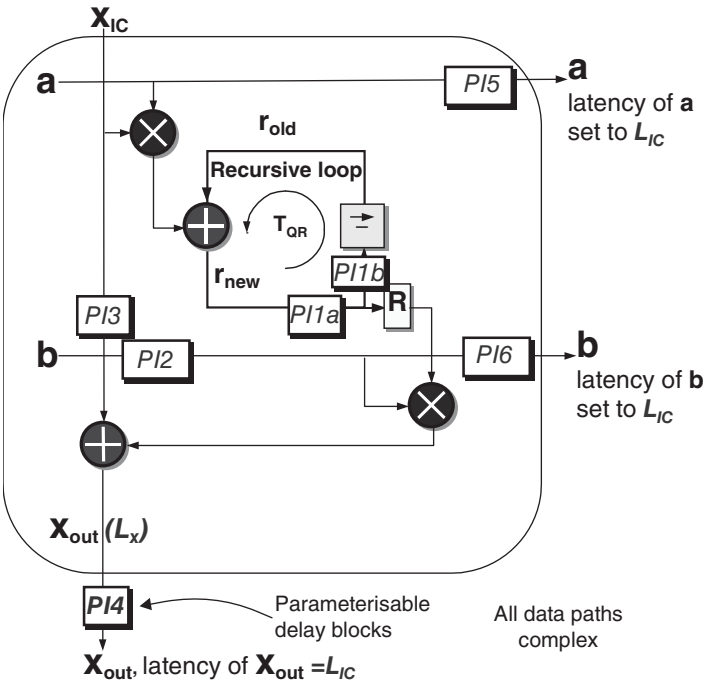



**Figure 11.27** Generically retimed IC

if the two values do not share a common factor other than 1 then their lowest common multiple will be their product. Otherwise there will be data collisions at the products of $L_{\mathrm{IC}}$ and $T_{\mathrm{QR}}$ with their common multiplies. Thus if $T_{\mathrm{QR}} = \mathrm{mod}\, c$ and $L_{\mathrm{IC}} = \mathrm{mod}\, c$ then $T_{\mathrm{QR}} = d \times c$ and $L_{\mathrm{IC}} = e \times c$, giving $c = T_{\mathrm{QR}}/d = L_{\mathrm{IC}}/e$, where $c$ is a common multiple of $T_{\mathrm{QR}}$ and $L_{\mathrm{IC}}$ and a positive integer other than 1, and $d$ and $e$ are factors of $T_{\mathrm{QR}}$ and

**Table 11.1** BC generic timing

| BC | Delay Value |
|---|---|
| $\mathrm{B_{RL}}$ | $2P_A + 2P_M + P_R + P_S - T_{\mathrm{QR}}$ |
| PB1 | $P_M$ |
| PB2 | $T_{\mathrm{QR}} - P_A - P_B$ |
| PB2a | If $B_{\mathrm{RL}} < 0$, then $-B_{\mathrm{RL}}$, otherwise, 0 |
| PB2b | If $B_{\mathrm{RL}} < 0$, then $PB2 - PB2a$, otherwise $PB2$ |
| PB3 | If $B_{\mathrm{RL}} > 0$, then $B_{\mathrm{RL}}$, otherwise, 0 |
| PB4 | $2P_A + P_M + P_R + P_D - PB3$ |
| PB5 | $2P_A + 2P_M + P_R + P_D - T_{\mathrm{QR}}$ |
| PB5a | If $PB5 < 0$, then $PB5$, otherwise, 0 |
| PB5b | If $PB5 > 0$, then $PB5$, otherwise, 0 |
| PB6 | $L_{\mathrm{IC}} - L_\delta$ |
| PB7 | $L_{\mathrm{IC}} - L_a$ |
| PB8 | $L_{\mathrm{IC}} - L_b$ |

**Table 11.2** IC generic timing

| IC | Delay Value |
|---|---|
| $I_{\mathrm{RL}}$ | $2P_A + P_M + P_R - T_{\mathrm{QR}}$ |
| PI1 | $T_{\mathrm{QR}} - P_A - P_S$ |
| PI1a | If $I_{\mathrm{RL}} < 0$, $-I_{\mathrm{RL}}$, otherwise, 0 |
| PI1b | If $I_{\mathrm{RL}} < 0$, $PI1 - PI1a$, otherwise, $PI1$ |
| PI2 | If $I_{\mathrm{RL}} > 0$, $I_{\mathrm{RL}}$, otherwise, $PI1$ |
| PI3 | $PI2 + P_A + P_M$ |
| PI4 | $L_{\mathrm{IC}} - L_x$ |
| PI5 | $L_{\mathrm{IC}}$ |
| PI6 | $L_{\mathrm{IC}} - PI2$ |

**Table 11.3** Generic expressions for the latencies of the BC and IC

| Latency | Value |
|---|---|
| $La$ | $P_M$ |
| $Lb$ | $P_M + PB5$ |
| $L_\delta$ | $PB3 + PB4 + 2P_M$ |
| $L_X$ | $PI3 + P_A$ |

$L_{\mathrm{IC}}$ respectively. Hence, there would be a collision at $T_{\mathrm{QR}} \times e = L_{\mathrm{IC}} \times d$. This means that the products of both $T_{\mathrm{QR}} \times e$ and $L_{\mathrm{IC}} \times d$ must be less than $T_{\mathrm{QR}} \times L_{\mathrm{IC}}$. Therefore, there is a collision of data. Conversely, to obtain a collision free set of values, $c$ is set to 1.

The time instance $T_{\mathrm{QR}} \times L_{\mathrm{IC}}$ does not represent a data collision as the value of $T_{\mathrm{QR}}$ is equal to $2m + 1$, as the QR operation that was in line to collide with a new QR operation will have just been completed. The other important factor in choosing an optimum value of $T_{\mathrm{QR}}$ and $L_{\mathrm{IC}}$ is to ensure that the processors are 100% efficient.

The simple relationship between $T_{\mathrm{QR}}$ and $L_{\mathrm{IC}}$ is a key factor in achieving a high utilization for each of the types of structure. More importantly, the relationship gives a concise mathematical expression that is needed in the automatic generation of a generic QR architecture complete with scheduling and retiming issues solved.

Figure 11.28 shows an example schedule for the seven-input linear array that was originally shown in Figure 11.12 where $L_{\mathrm{IC}}$ is 3 and $T_{\mathrm{QR}}$ is 7. The shaded cells represent the QR operations from different updates that are interleaved with each other and fill the gaps left by the highlighted QR update. The schedule is assured to be filled by the completion of the first QR update; hence, this is dependent on the latency, $L_{\mathrm{IC}}$.

## 11.7 Parameterizable QR Architecture

The main areas of parameterization include the wordlength, the latency of arithmetic functions, and the value of $T_{\mathrm{QR}}$. Different specifications may require different finite precision, therefore the wordlength is an important parameter. The QR cells have been built up using a hierarchical library of arithmetic functions, which are parameterized in terms of wordlength, with an option to include pipelining to increase the operation speed as

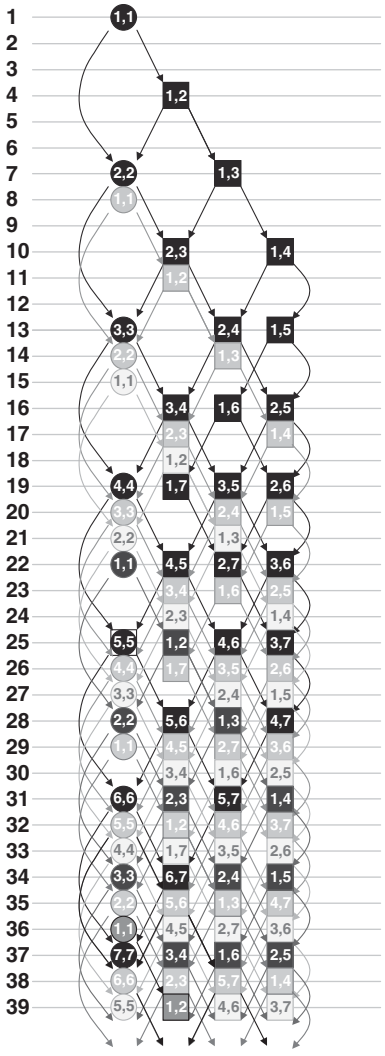



**Figure 11.28** Schedule for a linear array with an IC latency of 3

required. These parameters are passed down through the hierarchy of the HDL description of the QR cells to these arithmetic functions. Another consideration is the value of $T_{\mathrm{QR}}$, which determines the length of the memory needed within the recursive loops of the QR cells which hold the $R$ and $u$ values from one QR update to the next. Both $T_{\mathrm{QR}}$ and the level of pipelining within the arithmetic functions are incorporated in generic timing expressions of the SGR QR cells.

### 11.7.1 Choice of Architecture

Table 11.4 demonstrates the process for designing a QR architecture when given a specific sample rate and QR array size. The examples below are for a large QR array with 45 auxiliary inputs and 4 primary inputs, i.e. $m = 22$ and $p = 12$. The resulting processor array is $2m + 1 = 45$ rows by $m + p + 1 = 35$ columns. For a given sample throughput

**Table 11.4** Other example architectures (clock speed = 100 MHz)

| Arch. | Details | Number of processors | | | $T_{QR}$ | Data rate MSPS |
|---|---|---|---|---|---|---|
| | | BC | IC | total | | |
| Full QR | Processor for each QR cell | 45 | 1170 | 1215 | 4 | 25 |
| Rectangular 1 | Processor array assigned onto 12 linear arrays, each responsible for 4 rows | 12 | 312 | 324 | 4 | 25 |
| Rectangular 2 | Processor array assigned onto 3 linear arrays, each responsible for $45/3 = 15$ rows | 3 | 78 | 81 | $(2m + 1)/3$ (15) | 6.67 |
| Rectangular 3 | Processor array assigned onto 15 linear arrays (13 ICs), each responsible for $45/3 = 15$ rows 2 columns of ICs to each | 15 | 195 | 210 | $(2m + 1)/15$ | 16.67 |
| Sparse rectangular | 2 columns of ICs to each IC processor of 3 linear arrays | 3 | 39 | 42 | $2(2m + 1)/3$ (30) | 3.33 |
| Linear | 1 BC and 26 ICs | 1 | 26 | 27 | $2m + 1$ (45) | 2.22 |
| Sparse linear | 2 columns of ICs assigned to each IC processor of a linear array | 1 | 13 | 14 | $2(2m + 1)$ (90) | 1.11 |

rate and clock rate, we can determine the value for $T_{\mathrm{QR}}$, as depicted in the table. Note that the resulting value for $T_{\mathrm{QR}}$ and $L_{\mathrm{IC}}$ must be relatively prime, but for these examples we can leave this relationship at present.

The general description for $T_{\mathrm{QR}}$, as shown above, can be rearranged to give the following relationship:

$$\frac{N_{\mathrm{IC}}}{N_{\mathrm{rows}}} = \frac{T_{\mathrm{QR}}}{2m+1}.$$

This result is rounded down to the nearest integer. There are three possibilities:

- If $\frac{T_{\mathrm{QR}}}{2m+1} > 1$ then a sparse linear array is needed.
- If $\frac{T_{\mathrm{QR}}}{2m+1} = 1$ then a linear array is needed.
- If $\frac{T_{\mathrm{QR}}}{2m+1} < 1$ then a rectangular array is needed.

Depending on the dimensions of the resulting architecture, the designer may decide to opt for a sparse rectangular architecture.

Note that the maximum throughput that the full triangular array can meet is limited to 25 MSamples/s due to the four-cycle latency within the QR cell recursive path for the

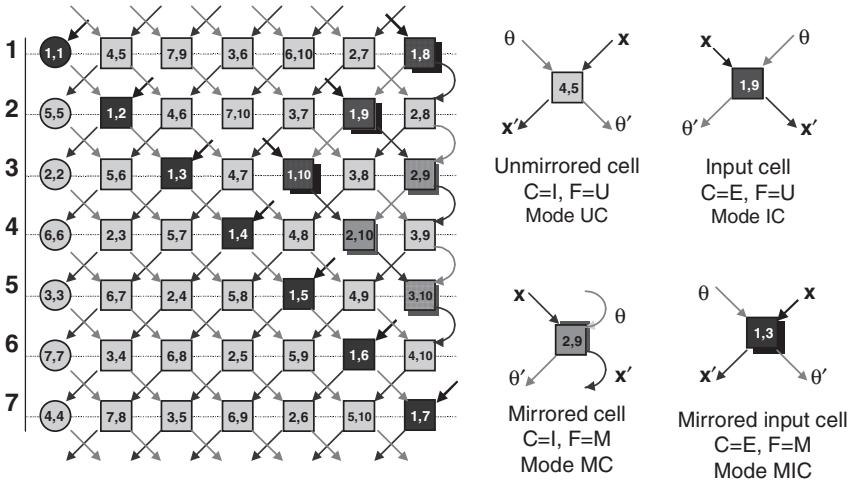



**Figure 11.29** Types of cells in processor array

specific implementation listed in Lightbody et al. The first rectangular array solution is meeting the same throughput performance as the full QR array using only 408 ICs and 12 BCs, instead of the full array which requires 1530 ICs and 45 BCs.

### 11.7.2 Parameterizable Control

A key aspect of the design of the various architectures is the determination of the control data needed to drive the multiplexers in these structures. Due to the various mappings that have been applied, it is more relevant of think of the IC operation as having four different modes of operation: input, mirrored input, unmirrored cell and mirrored cell (Figure 11.29). The mirrored ICs are the result of the fold used to derive the rectangular processor array from the QR array and simply reflect a different dataflow. The cell orientation is governed by the multiplexers and control, and is therefore an issue concerning control signal generation.

The four modes of operation can be controlled using two control signals, $C$, which determines whether the $x$ input is from the array (I) or from external data (E), and F, which distinguishes between a folded (M) and an unfolded operation (U). The latter determines the source direction of the inputs. The outputs are then from the opposite side of the cell. A mechanism for determining the control of each architecture is given next.

### 11.7.3 Linear Architecture

The control signals for the linear architecture were derived directly from its data schedule. The modes of operation of the cells were determined for each cycle of the schedule, as shown in Table 11.5. Figure 11.30 shows the QR cells with the applied control and multiplexers, and the control signals for a full QR operation for this example are given in Table 11.6. The control and timing of the architectures for the other variants become more complex – in particular, the effect that latency has on the control

**Table 11.5** Modes of operation of the QR cells for the linear array

| Cycle | BC1 | IC2 | IC3 | IC4 | IC5 | IC6 | IC7 |
|---|---|---|---|---|---|---|---|
| 1 | **IC** | UC | UC | UC | UC | UC | **MIC** |
| 2 | UC | **IC** | UC | UC | UC | **MIC** | UM |
| 3 | UC | UC | **IC** | UC | **MIC** | UC | **MIC** |
| 4 | UC | UC | UC | **IC** | UM | **MIC** | UM |
| 5 | UC | UC | UC | UC | **IC** | UM | **MIC** |
| 6 | UC | UC | UC | UC | UM | **IC** | UM |
| 7 | UC | UC | UC | UC | UM | UM | **IC** |

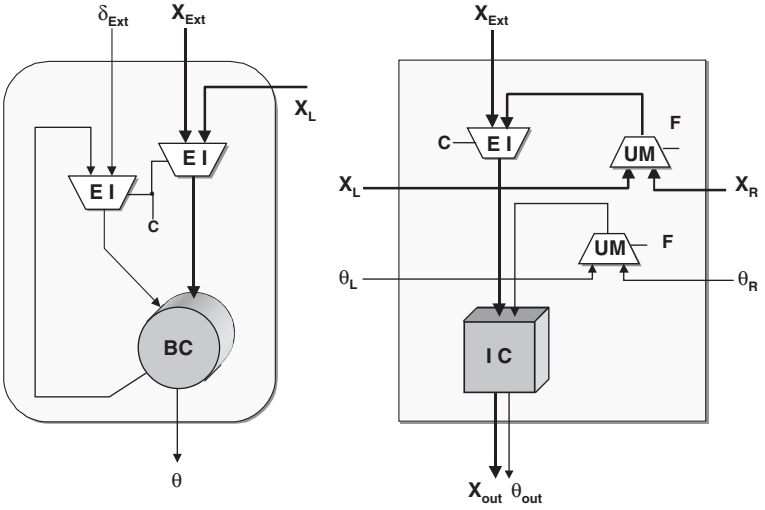



**Figure 11.30** QR cells for the linear architecture

**Table 11.6** Linear array control for the $x$-inputs and for mirrored/not mirrored cells

| Cyc. | C1 | C2 | C3 | C4 | C5 | C6 | C7 | F1 | F2 | F3 | F4 | F5 | F6 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | **E** | I | I | I | I | I | **E** | U | U | U | U | U | **M** |
| 2 | I | **E** | I | I | I | **E** | I | U | U | U | U | **M** | U |
| 3 | I | I | **E** | I | **E** | I | I | U | U | U | **M** | U | **M** |
| 4 | I | I | I | **E** | I | I | I | U | U | U | U | **M** | U |
| 5 | I | I | I | I | **E** | I | I | U | U | U | U | U | **M** |
| 6 | I | I | I | I | I | **E** | I | U | U | U | U | U | U |
| 7 | I | I | I | I | I | I | **E** | U | U | U | U | U | U |
| 8 | **E** | I | I | I | I | I | **E** | U | U | U | U | U | **M** |
| 9 | I | **E** | I | I | I | **E** | I | U | U | U | U | **M** | U |
| 10 | I | I | **E** | I | **E** | I | I | U | U | U | **M** | U | **M** |
| 11 | I | I | I | **E** | I | I | I | U | U | U | U | **M** | U |
| 12 | I | I | I | I | **E** | I | I | U | U | U | U | U | **M** |
| 13 | I | I | I | I | I | **E** | I | U | U | U | U | U | U |

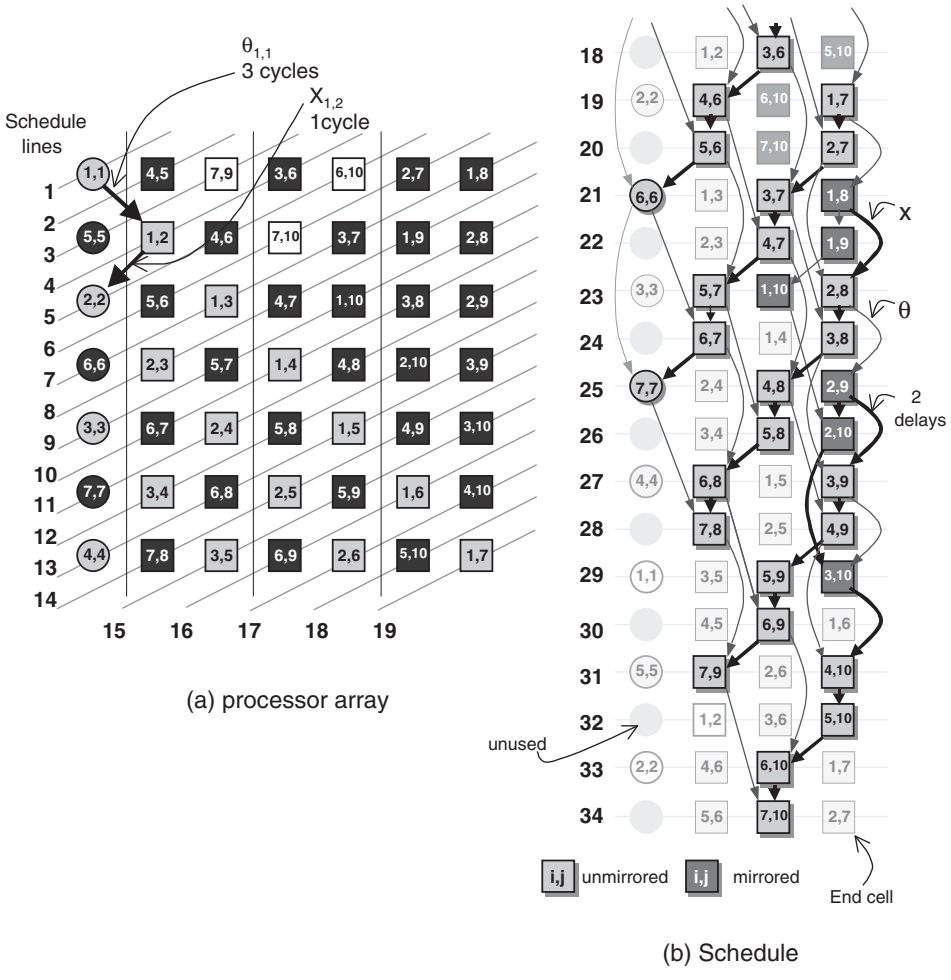



**Figure 11.31** Sparse linear array schedule

sequences. In the sparse variants, extra delays need to be placed within the cells to organize the schedule, and in the rectangular variants, the cells need to be able to take $x$ and $\theta$ inputs from the cells above and below as well as from adjacent cells. Each of these variants shall be looked at in turn.

### 11.7.4 Sparse Linear Architecture

Figure 11.31(a) shows two columns of operations being assigned onto each IC. From the partial schedule shown in Figure 11.31(b), it can be seen that the transition of a value from left to right within the array requires a number of delays. The transfer of $\theta_1$ from BC(1, 1), to the adjacent IC(1, 2) takes three cycles. However, the transfer of $X_{12}$ from the IC to the BC only takes one cycle.

The example in Figure 11.32 shows the partitioning of three columns of ICs. Scheduling them onto a single processor requires their sequential order to be maintained. The

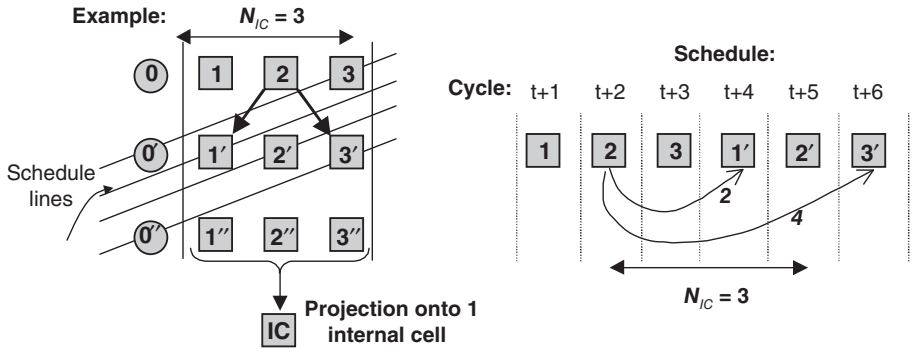



**Figure 11.32** Example partitioning of three columns onto one processor. (Source: Lightbody 2003. Reproduced with permission of IEEE.)

IC operations have been numbered 1, 2, 3 for the first row, and 1′, 2′, 3′ for the second row. The outputs generated from operation 2 are required for operations 1′ and 3′. Because all the operations are being performed on the same processor, delays are needed to hold these values until they are required by operations 1′ and 3′. Operation 3 is performed before operation 1′, and operations 3, 1′ and 2′ are performed before operation 3′, which relates to 2 and 4 clock cycle delays, respectively. This has been generically defined according to the number of columns of operations within the processor array assigned to each IC, $N_{\mathrm{IC}}$, as shown in Figure 11.33.

Two output values, $x$ and $\theta$, are transferred from operation $c + 1$ to $c$ and $c + 2$. The value that is fed to a specific operation depends on whether the cells perform the folded or unfolded modes of operation as summarized in Table 11.5. If the data is transferred

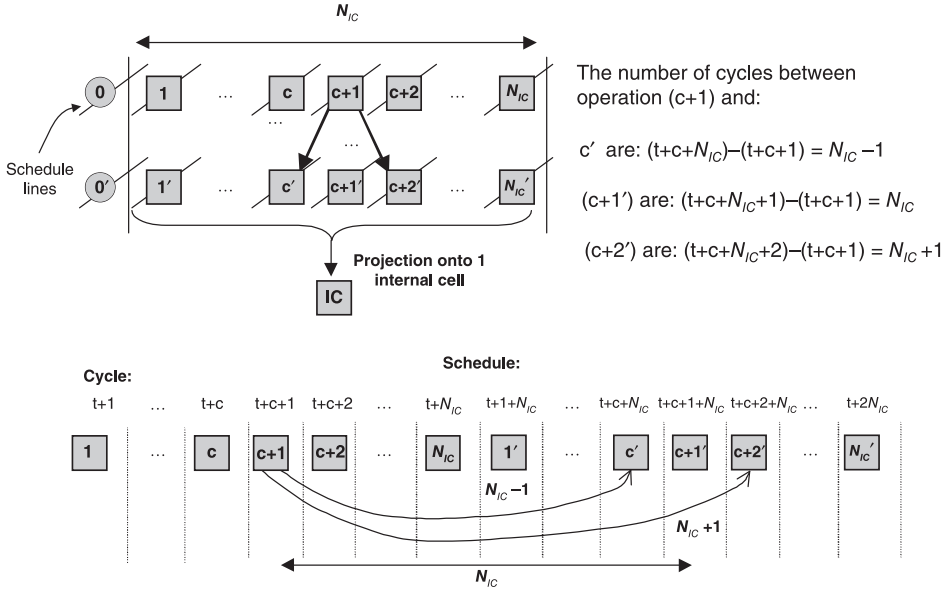



**Figure 11.33** Generic partitioning of $N_{\mathrm{IC}}$ columns onto one processor

**Table 11.7** Required delays for sparse linear array (U, not mirrored; M, mirrored)

| Data transfer | Direction in terms of QR operation | Dataflow direction | Delays | Label |
|---|---|---|---|---|
| U → U | $(i,j) \rightarrow (i,j+1), \theta$ | → | $N_{IC}+1$ | $D1$ |
| | $(i,j) \rightarrow (i+1,j), x$ | ← | $N_{IC}-1$ | $D2$ |
| M → M | $(i,j) \rightarrow (i+1,j), \theta$ | ← | $N_{IC}-1$ | $D2$ |
| | $(i,j) \rightarrow (i,j+1), x$ | → | $N_{IC}+1$ | $D1$ |
| U → M(end cell) | $(i,j) \rightarrow (i,j+1), \theta$ | ↓ | $N_{IC}$ | $D3$ |
| M → U(end cell) | $(i,j) \rightarrow (i+1,j), x$ | ↓ | $N_{IC}$ | $D3$ |

between the same type of cell (i.e. $U \rightarrow U$, or $M \rightarrow M$) then the delay will be either $N_{IC}-1$ or $N_{IC}+1$, according toTable 11.7. However, if the data transfer is between different types of cell (i.e. U → M, or M → U, as in the case of the end processor), then the number of delays will be $N_{IC}$. This is summarized in Table 11.7.

These delays are then used within the sparse linear architecture to keep the desired schedule as given in Figure 11.34. The three levels of delays are denoted by the square blocks labeled D1, D2 and D3. These delays can be redistributed to form a more efficient QR cell architectures as shown in Figure 11.35. The extra $L$ and $R$ control signals indicate the direction source of the inputs, with $E$ and $I$ control values determining whether the inputs come from an adjacent cell or from the same cell. $EC$ refers to the end IC that differs slightly in that there are two modes of operation when the cell needs to accept inputs from its output. The control sequences for this example are given in Figure 11.36.

From Figure 11.36, it can be seen that $EC$ is the same as $R$ and is the inverse of $L$. In addition, the states alternate between $E$ and $I$ with every cycle, therefore, one control

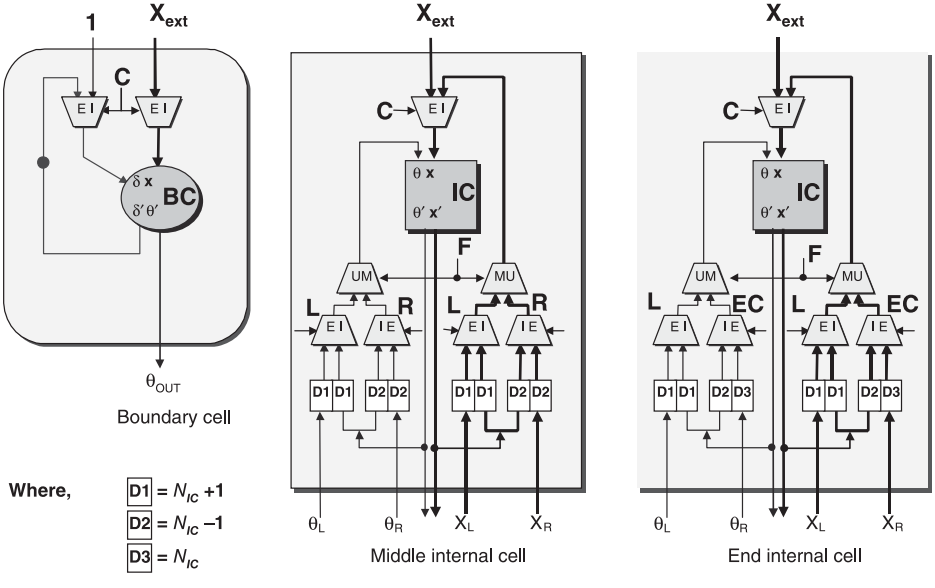



**Figure 11.34** Sparse linear array cells

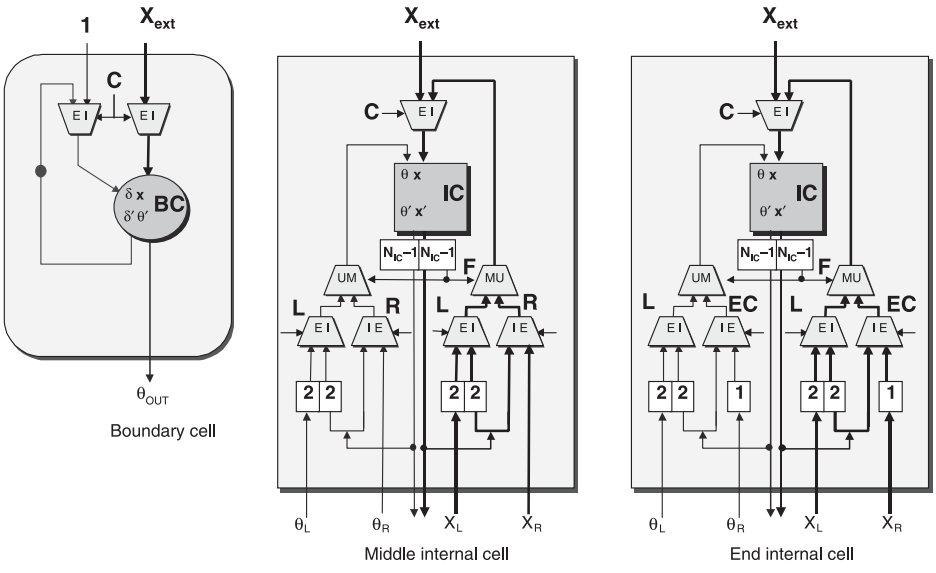



**Figure 11.35** Redistributed delays for sparse linear array cells

sequence could be used to determine the control of the internal inputs. This control value has been labeled *D*. The control signals shall be categorized as external input control, $C_i$, fold control, $F_i$, array control, $L_i$ and internal input control, $D_i$ The subscripts are coordinates representing the cells to which the control signals are being fed.

One of the key issues with the sparse linear array is the effect of the latencies in the QR cells on the schedule (which previously assumed a one-cycle delay). With the linear architecture, the schedule was scaled by the latency. However, with the sparse linear array, there was a concern that the delays $N_{\mathrm{IC}} - 1$, $N_{\mathrm{IC}}$, $N_{\mathrm{IC}} + 1$ would also need to be

| Cycle | External Input | External Input Control | | | | Fold Control | | | Internal Input Control | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | C1 | C2 | C3 | C4 | F2 | F3 | F4 | L | R | EC |
| 1 | $X_1(1)$ | E | I | I | I | U | U | M | E | I | E |
| 2 | $X_6(0)$ | I | I | I | E | U | U | U | I | E | I |
| 3 | | I | I | I | I | U | U | U | E | I | E |
| 4 | $X_2(1)$ | I | E | I | I | U | U | U | I | E | I |
| 5 | $X_7(0)$ | I | I | I | E | U | U | U | E | I | E |
| 6 | | I | I | I | I | U | U | U | I | E | I |
| 7 | $X_3(1)$, $X_8(0)$ | I | E | I | E | U | U | M | E | I | E |
| 8 | $X_9(0)$ | I | I | I | E | U | U | M | I | E | I |
| 9 | $X_{10}(0)$ | I | I | E | I | U | M | U | E | I | E |
| 10 | $X_4(1)$ | I | I | E | I | U | U | U | I | E | I |
| 11 | | I | I | I | I | U | U | M | E | I | E |
| 12 | | I | I | I | I | U | U | M | I | E | I |
| 13 | $X_5(1)$ | I | I | E | I | U | U | U | E | I | E |
| 14 | | I | I | I | I | U | U | U | I | E | I |

**Figure 11.36** Control sequence for sparse linear array

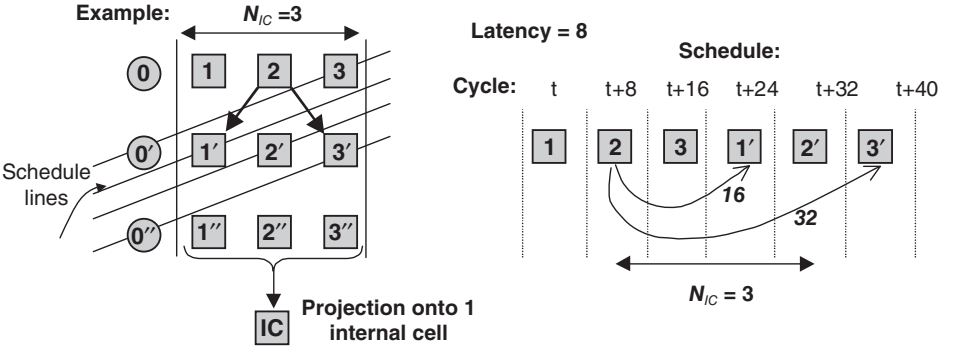



**Figure 11.37** Possible effect of latency on sparse linear array schedule

scaled in order to keep the structure of the original schedule, which would cause inefficiency. This is depicted in Figure 11.37 for a latency of 8. This is not the case, as the delays $N_{\mathrm{IC}} - 1$, $N_{\mathrm{IC}}$, $N_{\mathrm{IC}} + 1$ can be applied using the existing latency within the QR cells. The minimum allowed number of clock cycles between successive operations is the latency. By setting $N_{\mathrm{IC}} - 1$ to this minimum value, and then setting $N_{\mathrm{IC}}$ to be one clock cycle more and $N_{\mathrm{IC}} + 1$ to be two clock cycles more, a valid and efficient schedule can be achieved. This is depicted in Figure 11.38.

In the example given in Figure 11.39, the latency of the IC is 3, so this gives the minimum value for $N_{\mathrm{IC}}$ as 4. $N_{\mathrm{IC}} + 1$ is therefore 5 and $N_{\mathrm{IC}} - 1$ is 3 clock cycles. The shaded cells in Figure 11.39 show one complete QR update with interconnection included. The rest of the QR operations are shown but with limited detail to aid clarity. Since it is most probable that the latency of the IC will exceed the number of columns assigned to each processor, it figures that the delays within the linear sparse array will depend on $L_{\mathrm{IC}}$, i.e. the $N_{\mathrm{IC}} - 1$ delay will not be needed and the schedule realignment will be performed by the single- and double-cycle delays shown in Figure 11.35. The highlighted cells represent a full QR update, while the other numbered cells represent interleaved QR operations. The faded gray BCs with no numbers represent unused positions within the schedule.

### 11.7.5 Rectangular Architecture

The rectangular architecture consists of multiple linear array architectures that are concatenated. Therefore, the QR cells need to be configured so that they can accept inputs

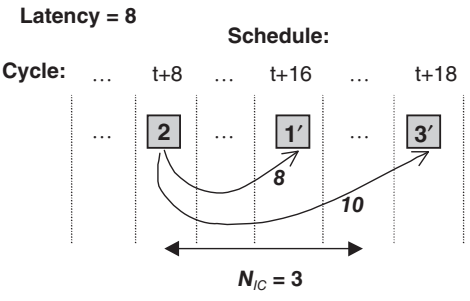



**Figure 11.38** Merging the delays into the latency of the QR cells

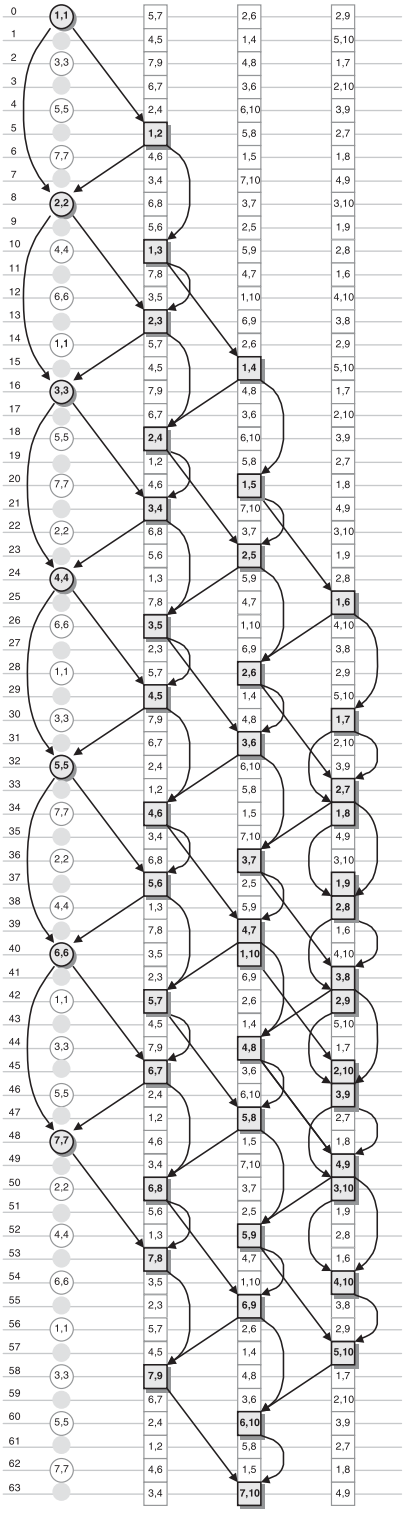


**Figure 11.39** Effect of latency on schedule for the sparse linear array ($N_{IC} = 3$)

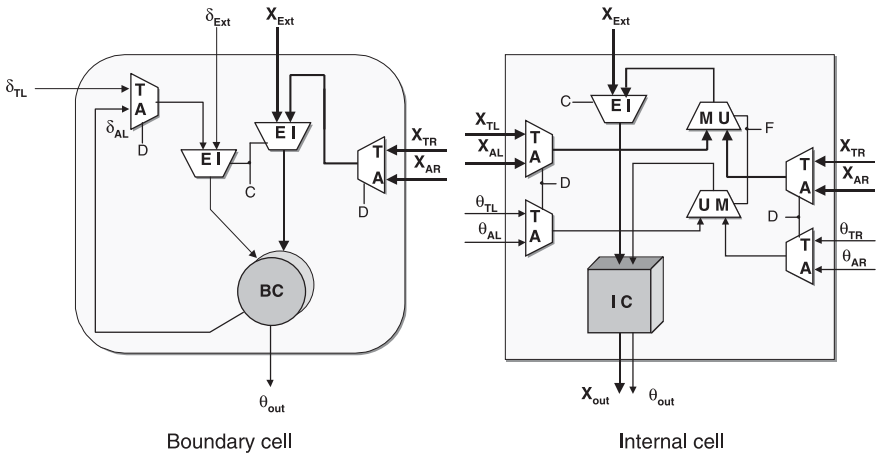



**Figure 11.40** QR cells for the rectangular array

from the above linear array. In addition, the top linear array needs to be able to accept values from the bottom linear array. The QR cells are depicted in Figure 11.40. The control signals, *E* and *I*, decide on whether the *X* inputs are external (i.e. system inputs) or internal. The control value, *T*, refers to inputs from the above array and *A* refers to inputs from adjacent cells. When used as subscripts, *TR* and *TL* refer to values coming from the left and right cells of the array above. *AR* and *AL* refer to the values coming from the right and left adjacent cells within the same linear array.

### 11.7.6 Sparse Rectangular Architecture

The QR cells for the sparse rectangular array need to be able to feed inputs back to themselves, in addition to the variations already discussed with the linear and rectangular architectures. The extra control circuitry is included in the QR diagrams shown in Figure 11.41. The control and the delays required by the sparse arrays to realign the schedule are brought together into LMR multiplexer cells (Figure 11.42) that include delays needed take account of the retiming analysis demonstrated in this section.

It was discussed with the sparse linear array how certain transfer in data values required the insertion of specific delays to align the schedule. This also applies to the rectangular array and the same rules can be used.

The starting point for determining the schedule for the sparse rectangular array is the schedule for the sparse linear array. From this, the rows of operations are divided into sections, each to be performed on a specific sparse linear array. The control, therefore, is derived from the control for the linear sparse version. The next section deals with parametric ways of generating the control for the various QR architectures. In addition to the control shown so far, the next section analyzes how latency may be accounted for within the control generation.

### 11.7.7 Generic QR Cells

The sparse rectangular array QR cells, shown in Figure 11.41, can be used for all of the QR architecture variants, by altering the control signals and timing parameters.

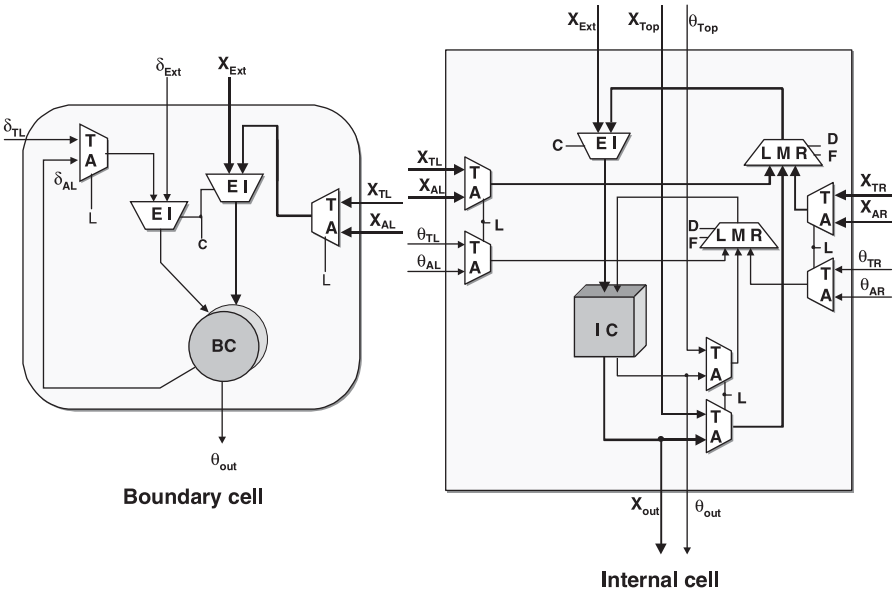



**Figure 11.41** QR cells for the sparse rectangular array

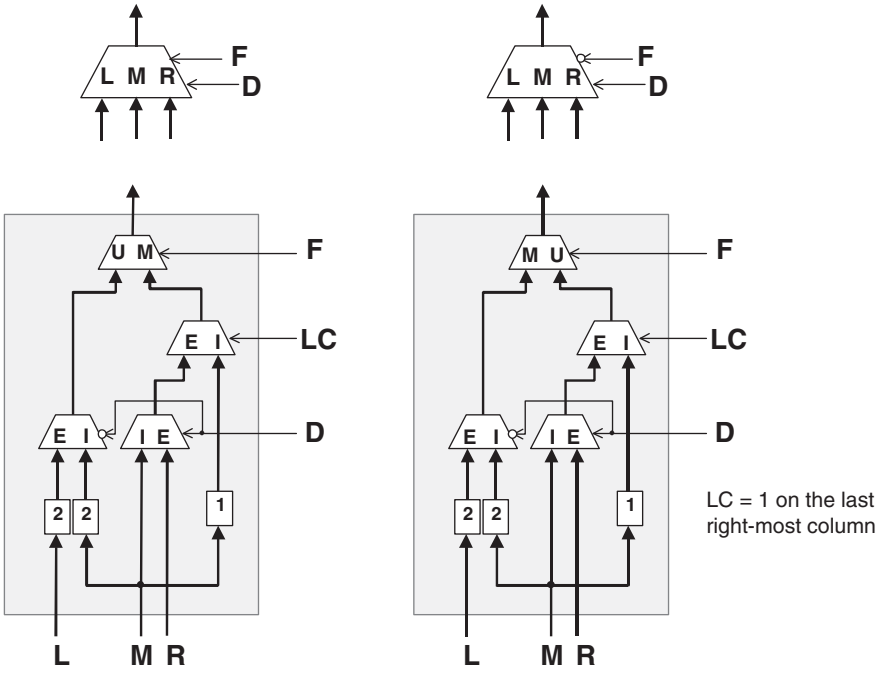



**Figure 11.42** LMR control circuitry for sparse arrays

However, in the sparse variants, there are added delays embedded within the LMR control cells. These can be removed for the full linear and rectangular array versions, by allowing them to be programmable so that they may be set to zero for the non-sparse versions. The key to the flexibility in the parameterizable QR core design is the generic generation of control signals. This is discussed in the following section.

## 11.8 Generic Control

The previous section detailed the various architectures derived from the QR array. Some detail was given of the control signals needed to operate the circuits. This section looks at generic techniques for generating the control signals that may be applied to all the QR architecture variants. It is suggested that a software interface is used to calculate each control sequence as a bit-vector seed (of length $T_{\mathrm{QR}}$) that may be fed through a linear feedback register which will allow this value to be cyclically output bit by bit to the QR cells.

The first stage in developing the control for the QR array is to look at the generic processor array which gives the control needed for the linear array. From this, the control signals may be folded and manipulated into the required sequence for the sparse linear arrays. The control for the rectangular versions may be generated quite simply from the control for the linear architectures.

### 11.8.1 Generic Input Control for Linear and Sparse Linear Arrays

A new external $x$-input is fed into a cell of the linear array on each clock cycle, starting from the leftmost cell, reaching the leftmost cell and then folding back until all the $2m + p + 1$ inputs are fed into the array for that specific QR update. This is highlighted for one set of QR inputs in Figure 11.43. The next set of inputs follow the same pattern but start after $T_{\mathrm{QR}}$ cycles. The result is a segment of control signals that repeat every $T_{\mathrm{QR}}$ cycles (which is 7 for the linear array example and 14 for the sparse linear array

| Cycle | Input | Linear array | | | | | | | Input | Sparse linear array | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | C1 | C2 | C3 | C4 | C5 | C6 | C7 | | C1 | C2 | C3 | C4 |
| 1 | $X_1(1)$, $X_8(0)$ | E | I | I | I | I | I | E | $X_1(1)$ | E | I | I | I |
| 2 | $X_2(1)$, $X_9(0)$ | I | E | I | I | I | E | I | $X_6(0)$ | I | I | I | E |
| 3 | $X_3(1)$, $X_{10}(0)$ | I | I | E | I | E | I | I | | I | I | I | I |
| 4 | $X_4(1)$ | I | I | I | E | I | I | I | $X_2(1)$ | I | E | I | I |
| 5 | $X_5(1)$ | I | I | I | I | E | I | I | $X_7(0)$ | I | I | I | E |
| 6 | $X_6(1)$ | I | I | I | I | I | E | I | | I | I | I | I |
| 7 | $X_7(1)$ | I | I | I | I | I | I | E | $X_3(1)$, $X_8(0)$ | I | E | I | E |
| 8 | $X_8(1)$, $X_1(2)$ | E | I | I | I | I | I | E | $X_9(0)$ | I | I | I | E |
| 9 | $X_9(1)$, $X_2(2)$ | I | E | I | I | I | E | I | $X_{10}(0)$ | I | I | E | I |
| 10 | $X_{10}(1)$, $X_3(2)$ | I | I | E | I | E | I | I | $X_4(1)$ | I | I | E | I |
| 11 | $X_4(2)$ | I | I | I | E | I | I | I | | I | I | I | I |
| 12 | $X_5(2)$ | I | I | I | I | E | I | I | | I | I | I | I |
| 13 | $X_6(2)$ | I | I | I | I | I | E | I | $X_5(1)$ | I | I | E | I |
| 14 | $X_7(2)$ | I | I | I | I | I | I | E | | I | I | I | I |

**Figure 11.43** Control for the external inputs for the linear QR arrays

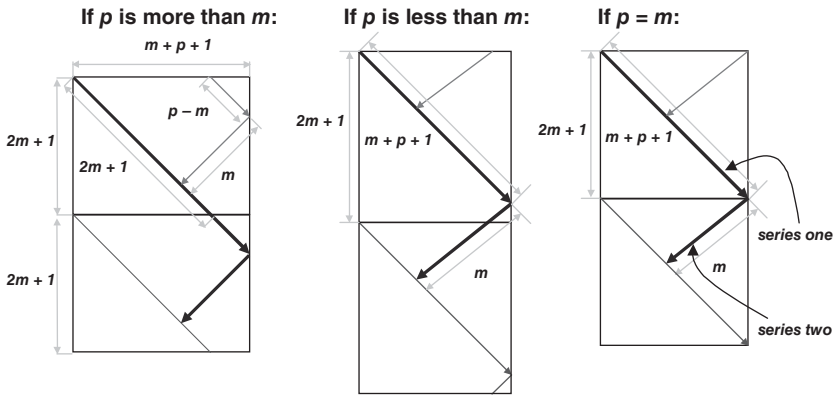



**Figure 11.44** External inputs

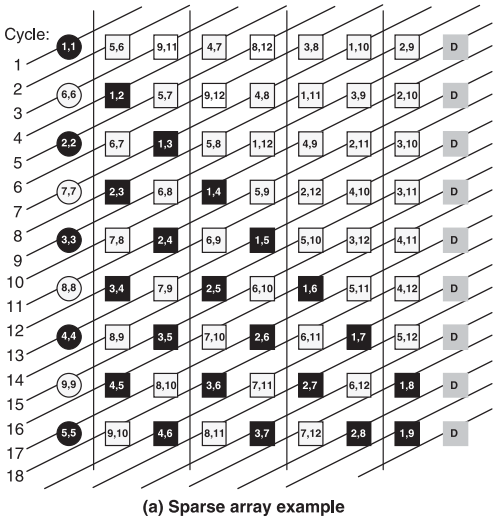
example). The aim is to automatically generate vectors, containing $T_{QR}$ bits, which represent the repeating sections for each of the control signals, C1 to C7. The key point is to determine when the next set of QR inputs starts in relation to the previous set. This can be determined mathematically from the dimensions of the original QR array and the resulting processor array from which the QR architectures are derived, i.e. a QR array with $2m + 1$ auxiliary inputs and $p$ primary inputs leads to a processor array with $2m + 1$ rows and $m + p + 1$ columns. This relationship is depicted by Figure 11.44. The heavy lines indicate the series of inputs for one QR update, and relate to the highlighted control for the external inputs for the linear array example in Figure 11.43.

Software code can be written to generate the control signals for the external inputs for the linear and sparse linear array. The inputs are broken down into two series (see Figure 11.44), one dealing with the inputs going from left to right, and the other dealing with the inputs from right to left (the change in direction being caused by the fold).

The code generates the position of the control signals within the control vector for each input into each processor. If the vector number is larger than the vector, then the vector size is subtracted from this value, leaving the modulus as the position. However, after initializing the operation of the QR array, it is necessary to delay this control signal by an appropriate value.

### 11.8.2 Generic Input Control for Rectangular and Sparse Rectangular Arrays

The control from the rectangular versions is then derived from these control vectors by dividing the signals vectors into parts relating to the partitions within the processor array. For example, if the control seed vectors for the linear array are eight bits wide and the rectangular array for the same system consists of two rows, then each control vector seeds would be divided into two vectors each four bits wide, one for the first rectangular array and the other for the second. The control seed for the sparse rectangular array is derived in the same manner from the control of the sparse linear array with the same value of $N_{IC}$. The same code may be edited to include the dummy operations that may be required for the sparse versions. Figure 11.45(a) shows an example sparse linear array mapping with $m = 4$, $p = 3$ and $N_{IC} = 2$. The control in Figure 11.45(b) can be divided




(a) Sparse array example

| cycle | Input | BC1 | IC2 | IC3 | IC4 | IC5 |
|---|---|---|---|---|---|---|
| 1 | $X_1(1)$, $X_7(0)$ | E | \| | \| | E | \| |
| 2 | | \| | \| | \| | \| | \| |
| 3 | | \| | \| | \| | \| | \| |
| 4 | $X_2(1)$, $X_8(0)$ | \| | E | \| | \| | E |
| 5 | | \| | \| | \| | \| | \| |
| 6 | $X_9(0)$ | \| | \| | \| | \| | E |
| 7 | $X_3(1)$, $X_{10}(0)$ | \| | E | \| | E | \| |
| 8 | $X_{11}(0)$ | \| | \| | \| | E | \| |
| 9 | $X_{12}(0)$ | \| | \| | E | \| | \| |
| 10 | $X_4(1)$ | \| | \| | E | \| | \| |
| 11 | | \| | \| | \| | \| | \| |
| 12 | | \| | \| | \| | \| | \| |
| 13 | $X_5(1)$ | \| | \| | E | \| | \| |
| 14 | | \| | \| | \| | \| | \| |
| 15 | | \| | \| | \| | \| | \| |
| 16 | $X_6(1)$ | \| | \| | \| | E | \| |
| 17 | | \| | \| | \| | \| | \| |
| 18 | | \| | \| | \| | \| | \| |

Control for Row 1

Control for Row 2

(b) External input control

**Figure 11.45** Partitioning of control seed

into two sections for implementing a sparse rectangular array consisting of two rows of the sparse linear array.

### 11.8.3 Effect of Latency on the Control Seeds

The next stage is to determine the effect that latency has on the control vectors. As discussed in this section, the sparse array needs delay values $D1$, $D2$ and $D3$, to account for assigning multiple columns, $N_{IC}$, of operations to each IC processor. For a system with a single cycle latency, $D1 = N_{IC} - 1$, $D2 = N_{IC}$ and $D3 = N_{IC} + 1$. However, in the real system the processors have multiple latency. It is assumed that the latency of the IC, $L_{IC}$, will be greater than these delays, so the delays are added onto the latency such that the appropriate delays become $D1 = L_{IC}$, $D2 = L_{IC} + 1$ and $D3 = L_{IC} + 2$. For the linear array the values $D1$, $D2$ and $D3$ are all set to $L_{IC}$. Then the code may be used to generate the control vectors. The only difference is when the position of the control value exceeds the width of the vector. With the single latency version, this was accounted for by subtracting the value $T_{QR}$ from the value (where the width of the vector seed is $T_{QR}$).

When latency is included within the calculations, it is not sufficient to reduce the value to within the bounds of the vector width. Alternatively, the position of the control value within the vector is found by taking the modulus of $T_{QR}$. An analysis of the effect of latency on the control vectors is shown through an example linear array where $m = 3$, $p = 5$ and $T_{QR} = 2m + 1 = 7$.

One point to highlight is the fact that there may be several cycles of the control vector before the required input is present. For example, the vector in the above example for C4 is [I I E I I I I ], but the first required input is at time 10, not 3. Therefore it is necessary to delay the start of this control signal by 7 cycles. The technique relies on the use of initialization control signals to start the cycling of the more complicated control vectors for the processors. However, the method discussed offers a parametric way of dealing with control and allows the majority of the control to be localized. In addition, the same

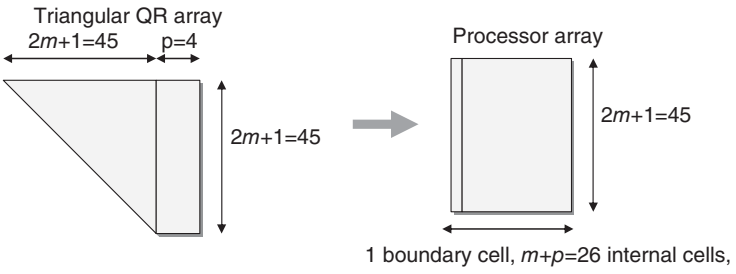



**Figure 11.46** Example QR architecture derivation, $m = 22$, $p = 4$

principles used to develop the control signals for the timing of the external inputs may be applied for the rest of the control signals, i.e. the fold, internal input, and row control.

## 11.9 Beamformer Design Example

For a typical beamforming application in radar, the values of $m$ would range from 20 to over 100. The number of primary inputs, $p$, would typically be from 1 to 5 for the same application. An example specification is given in Figure 11.46. One approach is to use the QR array. Assuming the fastest possible clock rate, $f_{\mathrm{CLK}}$, the fundamental loop will dictate the performance and result in a design with 25% utilization. Thus the major challenge is now to select the best architecture, mostly closely matching the throughput rate with the best use of hardware. For the example here, a desired input sample rate of 15 MSPS with a maximum possible clock rate of 100 MHz is assumed.

The value for $T_{\mathrm{QR}}$ can be calculated using the desired sample rate, $S_{\mathrm{QR}}$, and the maximum clock rate, $f_{\mathrm{CLK}}$:

$$T_{\mathrm{QR}} = \frac{f_{\mathrm{CLK}}}{S_{\mathrm{QR}}} = \frac{100 \times 10^6}{15 \times 10^6} = 6.67.$$

This value is the maximum number of cycles allowed between the start of successive QR updates, therefore, it needs to be rounded down to the nearest integer. The ratio $N_{\mathrm{rows}}/N_{\mathrm{IC}}$ can be obtained by substituting for the known parameters into the relationship below:

$$\frac{N_{\mathrm{rows}}}{N_{\mathrm{IC}}} = \frac{2m+1}{T_{\mathrm{QR}}} = \frac{45}{6} = 7.5,$$

where $1 \leq N_{\mathrm{rows}} \leq 2m + 1$ (i.e. 45) and $1 \leq N_{\mathrm{IC}} \leq m + p$ (i.e. 26). Using these guidelines, an efficient architecture can be derived by setting $N_{\mathrm{IC}} = 2$, and hence $N_{\mathrm{rows}} = 15$. The operations are distributed over 15 sparse linear architectures, each with 1 BC and 13 ICs, as shown in Figure 11.47.

Also note that the circuit critical path within the circuit must be considered to ensure that the core can be clocked fast enough to support the desired QR operation. Here, additional pipeline stages may be added to reduce the critical path and therefore improve the clock rate. However, this has the effect of increasing the latencies and these must then be included in the architecture analysis.

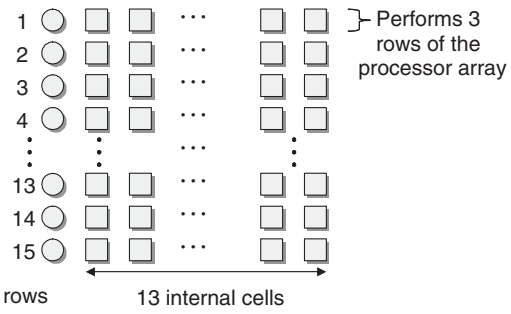



**Figure 11.47** Example architecture

Each row of processors is responsible for three rows of operations within the processor array, therefore, $T_{\mathrm{QR}} = 6$, resulting in an input sample rate of 16.67 MSPS, which exceeds the required performance. The details of some example architectures for the same QR array are given in Table 11.8.

The value for $T_{\mathrm{QR}}$ for the full QR array implementation is determined by the latency in the recursive loop of the QR cells (consisting of a floating- point addition and a shift subtract function). For the example shown, the QR array needs to wait four clock cycles for the calculation of the value in the recursive loop, which therefore determines the sample rate of the system. This example emphasizes the poor return of performance of the full QR implementation at such a high cost of hardware. The same performance can be achieved by using the first rectangular array example with only about one quarter the number of processors.

**Table 11.8** Selected architectures (clock speed = 100 MHz)

| Arch. | Details | Number of processors | | | $T_{\mathrm{QR}}$ | Data rate MSPS |
|---|---|---|---|---|---|---|
| | | BC | IC | total | | |
| Full QR | Processor for each QR cell | 45 | 1530 | 1575 | 4 | 25 |
| Rectangular 1 | Processor array assigned onto 12 linear arrays, each responsible for 4 rows | 12 | 408 | 420 | 4 | 25 |
| Rectangular 2 | Processor array assigned onto 3 linear arrays, each responsible for $45/3 = 15$ rows | 3 | 102 | 105 | $(2m+1)/3$ (15) | 6.67 |
| Sparse rectangular | 3 columns of ICs to each IC processor of 3 linear arrays | 3 | 51 | 54 | $2(2m+1)/3$ (30) | 3.33 |
| Linear | 1 BC and 34 ICs | 1 | 34 | 35 | $2m+1$ (45) | 2.22 |
| Sparse linear | 2 columns of ICs assigned to each IC processor of a linear array | 1 | 17 | 18 | $2(2m+1)$ (90) | 1.11 |

## 11.10 Summary

The goal of this chapter was to document each of the stages of development for an IP core for adaptive beamforming. The main aspects covered were the design choices made with regard to:

- the decision to use design-for-reuse strategies to develop an IP core;
- determination of the algorithm;
- determination of a suitable component to design as an IP core;
- specifying the generic parameters;
- algorithm to architecture development;
- scalable architectures;
- scalable scheduling of operations and control.

Each stage listed above was detailed for the adaptive beamforming example. Background information was supplied regarding the RLS choice of algorithm decided upon for the adaptive weight calculations. The key issue with the algorithm used is its computational complexity. Techniques and background research were summarized showing the derivation of the simplified QR-RLS algorithm suitable for implementation on a triangular systolic array. Even with such reduction in the complexity there may still be a need to map the full QR array down onto a reduced architecture set.

This formed a key component of the chapter, giving a step-by-step overview of how such a process can be achieved while maintaining a generic design. Consideration was given to architecture scalability and the effects of this on operation scheduling. Further detail was given of the effects of processor latency and retiming on the overall scheduling problem, showing how such factors could be accounted for upfront. Finally, examples were given on how control circuitry could be developed so as to scale with the architecture, while maintaining performance criteria. It is envisaged that the principles covered by this chapter should be expandable to other IP core developments.

## Bibliography

Athanasiadis T, Lin K, Hussain Z 2005 Space-time OFDM with adaptive beamforming for wireless multimedia applications. In *Proc. 3rd Int. Conf. on Information Technology and Applications*, pp. 381–386.

Baxter P, McWhirter J 2003 Blind signal separation of convolutive mixtures. In *Proc. IEEE Asilomar Conf. on Signals, Systems and Computers*, pp. 124–128.

Choi S, Shim D 2000 A novel adaptive beamforming algorithm for a smart antenna system in a CDMA mobile communication environment. *IEEE Trans. on Vehicular Technology*, 49(5), 1793–1806.

de Lathauwer L, de Moor B, Vandewalle J 2000 Fetal electrocardiogram extraction by blind source subspace separation. *IEEE Trans. on Biomedical Engineering*, 47(5), 567–572.

Gentleman W, Kung H 1982 Matrix triangularization by systolic arrays. In *Proc. SPIE*, 298, 19–26.

Hamill R 1995 VLSI algorithms and architectures for DSP arithmetic computations. PhD thesis, Queen's University Belfast.

Haykin S 2002 *Adaptive Filter Theory*, 4th edn. Prentice Hall, Upper Saddle River, NJ.

Hudson J 1981 *Adaptive Array Principles*. IET, Stevenage.
Kung S 1988 *VLSI Array Processors*. Prentice Hall, Englewood Cliffs, NJ.
Lightbody G 1999 High performance VLSI architectures for recursive least squares adaptive filtering. PhD thesis, Queen's University Belfast.
Lightbody G, Woods R, Walke R 2003 Design of a parameterizable silicon intellectual property core for QR-based RLS filtering. *IEEE Trans. on VLSI Systems*, 11(4), 659–678.
McCanny J, Ridge D, Hu Y, Hunter J 1997 Hierarchical VHDL libraries for DSP ASIC design. In *Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, pp. 675–678.
McWhirter J 1983 Recursive least-squares minimization using a systolic array. In *Proc. SPIE*, 431, 105–109.
Northeastern University 2007 Variable precision floating point modules. http://www.coe.neu.edu/Research/rcl/projects/floatingpoint/index.htmlprojects (accessed November 7, 2016).
Rader C 1992 MUSE –a systolic array for adaptive nulling with 64 degrees of freedom, using Givens transfomations and wafer scale integration. In *Proc. Int. Conf. on Application Specific Array Processors*, pp. 277–291.
Rader C 1996 VLSI systolic arrays for adaptive nulling. *IEEE Signal Processing Magazine*, 13(4), 29–49.
Shan T, Kailath T 1985 Adaptive beamforming for coherent signals and interference. *IEEE Trans. on Acoustics, Speech and Signal Processing*, 33(3), 527–536.
Shepherd TJ, McWhirter JG 1993 Systolic adaptive beamforming. In Haykin S, Litva J, Shepherd TJ (eds) *Array Signal Processing*, pp. 153–243. Springer, Berlin.
Tamer O, Ozkurt A 2007 Folded systolic array based MVDR beamformer. In *Proc. Int. Symp. on Signal Processing and its Applications*, pp. 1–4.
Trainor D, Woods R, McCanny J 1997 Architectural synthesis of digital signal processing algorithms using IRIS. *J. of VLSI Signal Processing*, 16(1), 41–55.
Walke R 1997 High sample rate Givens rotations for recursive least squares. PhD thesis, University of Warwick.
Wiltgen T 2007 Adaptive beamforming using ICA for target identification in noisy environments. Master's thesis, Virginia Tech.