Chapter 3

Study of Common DSP Algorithms and Their Architectures

Contents

3.1 Finite Impulse Response Filter **47**
  3.1.1 FIR Filter Architectures 48
3.2 Discrete Fourier Transform **53**
  3.2.1 Fast Fourier Transform 54
  3.2.2 Fast Fourier Transform Architectures 58
3.3 Discrete Cosine Transform **64**
  3.3.1 Reported pipelined DCT Implementations 66
3.4 Viterbi Decoding **69**
  3.4.1 Convolutional Coding 69
  3.4.2 The Viterbi Algorithm 69
  3.4.3 Proposed Parallel-Architecture Viterbi Decoder 71
  3.4.4 Viterbi decoding using the R-2SDF architecture 76
3.5 Proposed Specialized Reconfigurable Architectures **78**
  3.5.1 The Proposed Reconfigurable Size MDCT Processor 78
  3.5.2 A Reconfigurable FIR Filter Realization 82
  3.5.3 The Proposed R2MDF Architecture 87
  3.5.4 Reconfigurable R2^2MDF FFT Processors 94
3.6 Concluding Remarks **99**

In the past few decades, developments in field of Digital Signal Processing (DSP) combined with the advancements in process technology and computer arithmetic technology has paved the way for the realization of a magnitude of products and applications that were not previously possible to realize. Applications in areas ranging from automatic control to cryptography and multimedia processing became now possible to realize by utilizing the advancements in the aforementioned fields.

An economical and common solution is to implement the DSP algorithm of interest in software and have it run it on a μp. However, μps may not be a suitable solution when processing speed is demanded.

This is because typically many DSP applications involve a considerable amount of arithmetic operations. Such heavy arithmetic computation requirements need relatively long computation times given the limited amount of arithmetic functional units available in most μps. Hence, μp performance will not suffice many DSP applications demands.

In this context, specialized ASICs that are capable of delivering the required computations within the required time constraints are utilized, yielding thus orders of magnitude of performance increase. As was discussed in Chapter 1. This is a solution that lacks flexibility and involves increasing amounts of NRE costs.

There are several strategies for designing processors for DSP algorithms. One way is to design an accelerator implementing the most computationally expensive parts of the algorithm. This approach —although it usually saves hardware— requires extra control and data management resources especially when the implemented accelerator needs to be invoked several times within the computation. This control is mostly implemented in software which in tern elongates the total time of computation.

An other design strategy is to implement the complete algorithm in hardware investing thus in area to increase the performance. The DSP algorithm can either be implemented as either a *"parallel architecture"* or as *"pipelined architecture"*. In the parallel architecture approach parallelism in the DSP application is exploited and usually blocks of data are processed and produced at each clock cycle. This approach usually needs minimal control since data is consumed, processed and produced in parallel with no need for intermediate storage and operation scheduling because the complete *S*ignal *F*low Graph (*SFG*) is implemented. However parallel architectures are very expensive in terms of area requirements and thus are only used when high performance is critical.

In the so called pipelined approach data is consumed and produced serially. Therefore, scheduling of intermediate data and operations is required to allow operation on the appropriate pair (or set) of data supplied to different computational elements. The control resources needed are usually of moderate size especially when the SFG is regular. Area requirements of both data path and control resources are typically considerably lower than that of the parallel approach. In general, pipelined architectures are widely used and exhibit adequate performance.

The algorithms chosen for discussion in this Chapter span a wide range of DSP applications and thus a wide range of VLSI architectures as well. Finite Impulse Response filtering was chosen for its simplicity and regularity, the Fast Fourier Transform for its relative complexity (involving extensive butterfly operations) and regularity, the Discrete Cosine Transform especially for its irregularities and Viterbi Decoding for its irregularity and involvement of bit operations. All of the four above algorithms are widely used and a lot of work on several VLSI architectures have been reported in the literature.

In the course of this work not only the abovementioned DSP and their VLSI architectures were studied, but moreover, some of them were realized and in some cases improved and extended. The goal of these realizations was to design and test designs of these DSP algorithms featuring similar and repetitive modules. This modularity of internal components is a feature that can help clarify what computational features are included in such modules and how is the data flowing within the data path. This knowledge is very useful in designing the target CGRC for DSP applications: its processing elements, routing strategies and other features.

In the next sections the above DSP algorithms are briefly described, their implementations are discussed and our contributions are also presented. In the next Section the Finite Impulse Response filter and its implementations are discussed. This is followed by a similar discussion on the Fast Fourier Transform and its implementations in Section 3.2. The Discrete Cosine Transform and its implementations are discussed in Section 3.3. In section 3.4 Viterbi decoding and its implementations are discuses. Section 3.5 presents some of out contributions in the development of VLSI architectures for the above algorithms. A summary and remarks in Section 3.6 concludes this chapter.

3.1 Finite Impulse Response Filter

Filtering in general means allowing some parts of the input signal to pass to the output according to given characteristics. In signal processing filtering allows the removal uninteresting parts of the input signal such as random noise, or components of a given frequency content. In digital communications filtering can be either analog or digital. Analog filtering operates on the analog signal before analog to digital conversion. Alternatively, filtering can take place on the digitized signal. Digital filtering in general is more suited for fabrication since it can be more easily fabricated in standard C-MoS processes. In addition, digital filters are less sensitive to ambient conditions changes and enjoy good stability properties.

The Finite Impulse Response Filter (*FIR*) is a very common algorithm used in spectral shaping, motion estimation, noise reduction channel equalization among many other DSP applications. Because of its bounded output characteristics it takes its name “Finite Impulse Response”.

The FIR filter is a generalization of the on going average filter. The FIR filter's output is computed by multiplying current and past inputs by coefficients such as shown in the FIR filter formula below:

$$y(n) = \sum_{k=0}^{N} h_k x(n-k), \tag{3.1}$$

or equivalently, the FIR filter's transfer function is given by

$$H(z) = \sum_{k=0}^{N} h_k z^{-k}. \tag{3.2}$$

The size of the FIR filter is determined by the number of coefficients. A FIR filter of size N will have N coefficients and at least $N-1$ delay elements to store past values of the input needed to be operated on to produce the output. The size of the FIR filter is some types expressed in *taps* which is the number of delay elements + 1.

3.1.1 FIR Filter Architectures

- *Canonical Form Structure*
  The Canonical form FIR filter structure is directly derived from Equation 3.1 above. In the canonical form structure, partial products of current and previous inputs are fed to a multiple input adder. Practical realizations of such canonical form structure is known as the *adder tree* architecture. For an N tap FIR filter, the adder tree architecture depicted in Figure 3.1 needs $N-1$ registers for storing previous inputs, $N-1$ adders and N multipliers.

- *Direct Form Structure*
  The direct form implementation of the FIR filter can be also directly obtained from its formula. As shown in Figure 3.2 delayed inputs are scaled by factors producing partial products that are in tern summed to evaluate the output. Notice that this architecture of the FIR filter can be divided to similar portions each containing a delay element, a multiplier and an adder each representing a tap.

- *Transposed Form Structure*
  Figure 3.3 SFG of the direct form FIR filter of Figure 3.2. The transposition theory states that: *"Reversing the direction of all the edges in a given SFG and interchanging the input and output ports preserves the functionality of the system"* [74]. The transposed SFG of the direct form FIR filter is depicted in Figure 3.4. Thereupon, the transposed form FIR filter is shown in Figure 3.5 where the input is broadcaster

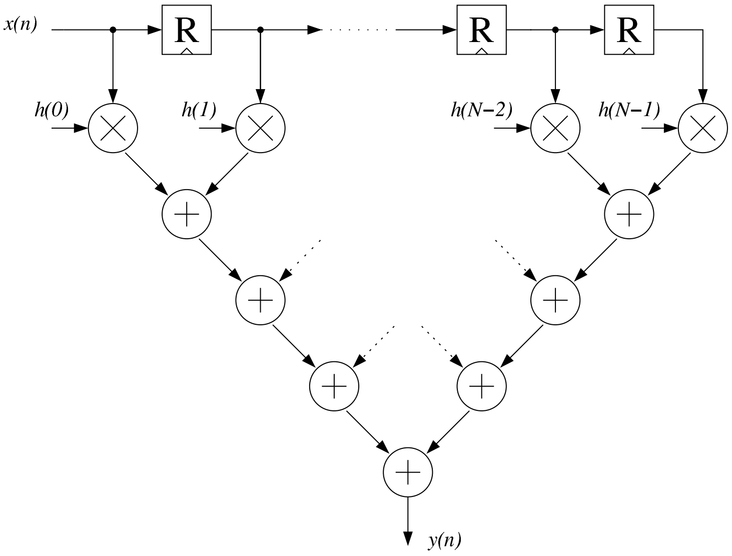

Figure 3.1: The adder tree architecture

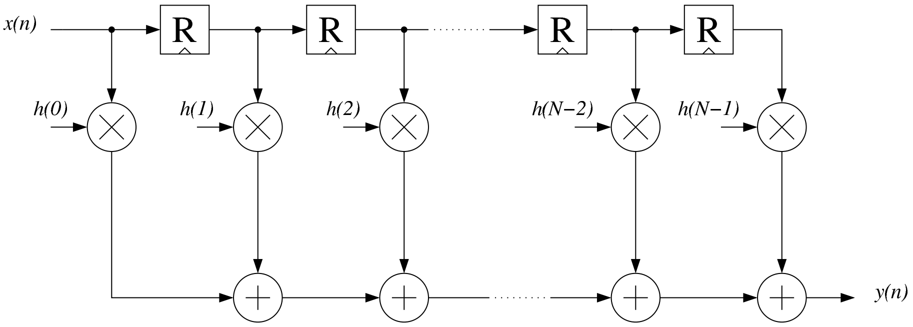

Figure 3.2: The direct form implementation of the FIR filter

to the multipliers with the coefficients in reverse order and hence the transposed FIR filter implementation is also o known as the *broadcast implementation*. Here we can also partition the FIR filter into taps each having a multiplier and an adder.

One of the advantages of the transposed form is that the registers shown in the bottom of Figure 3.5 serve the purpose of delaying partial products as well as pipelining and thus saving pipelining registers pipelining hardware. However, broadcasting implementations are not preferred in VLSI implementations because of higher parasitics and cross-talk potential [1] which makes the Direct Form FIR filter implementation preferred for VLSI implementations. An other advantage of the Direct Form implementations is that it allows the implementation of bigger FIR filters than that which can be implemented by the available hardware. This can be done done by storing partial data in intermediate memory and then

---

[1] resulting from the long input broadcasting wires

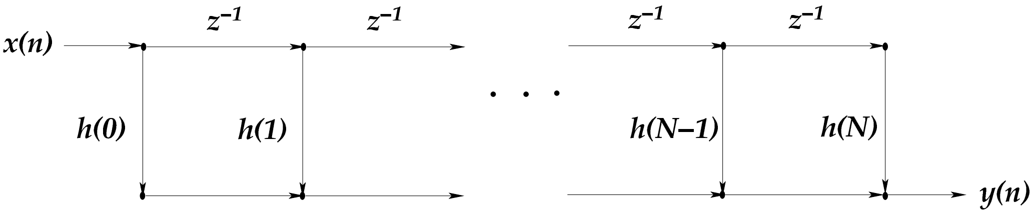

Figure 3.3: SFG of the direct form FIR filter

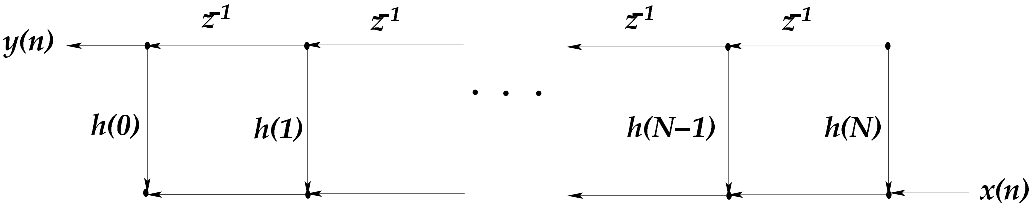

Figure 3.4: SFG of the transposed direct form FIR filter

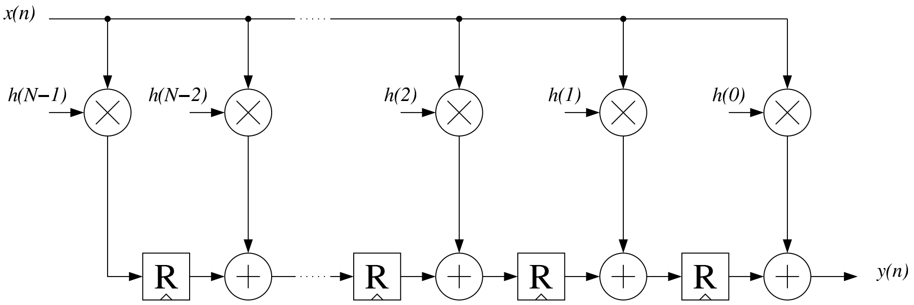

Figure 3.5: The transposed direct form implementation of the FIR filter

the same hardware is used once again to complete the FIR filter computations since the computations in later taps of the FIR filter is independent of the inputs. In contrast, such partitioning is not possible in the the Transposed Form implementation since the old inputs are needed to operate on all taps and thus additional memory is needed to store old inputs.

For linear phase filtering the FIR filter's coefficient are symmetric and therefore,the number of multiplies can be reduced to the half and a symmetric FIR filter can be realized as shown in Figure 3.6.

- *Cascade Form Structure*
  The summation in Equation 3.2 can be factorized and thus the transfer function can be represented by:

$$H(z) = h_0 \prod_{k=0}^{K} (1 + b_{1k} z^{-1} + b_{2k} z^{-2}), \tag{3.3}$$

where $K = N/2$ if N is even and $K = (N+1)/2$ if N is odd with $b_{2K} = 0$.

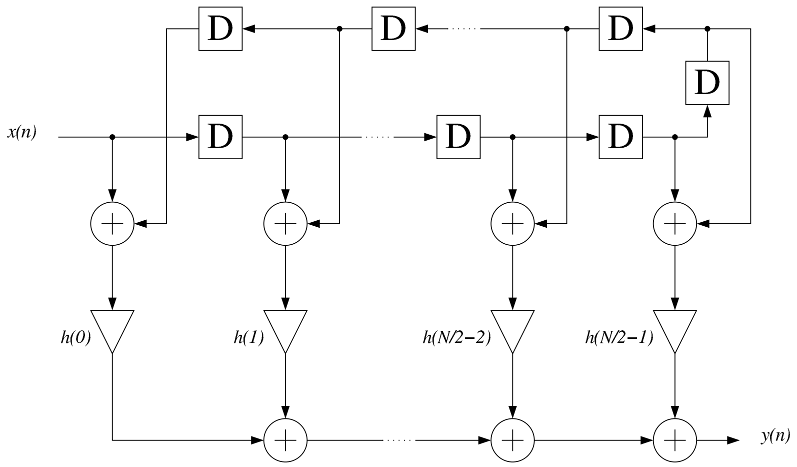

Figure 3.6: The linear phase transposed direct form implementation of the FIR filter

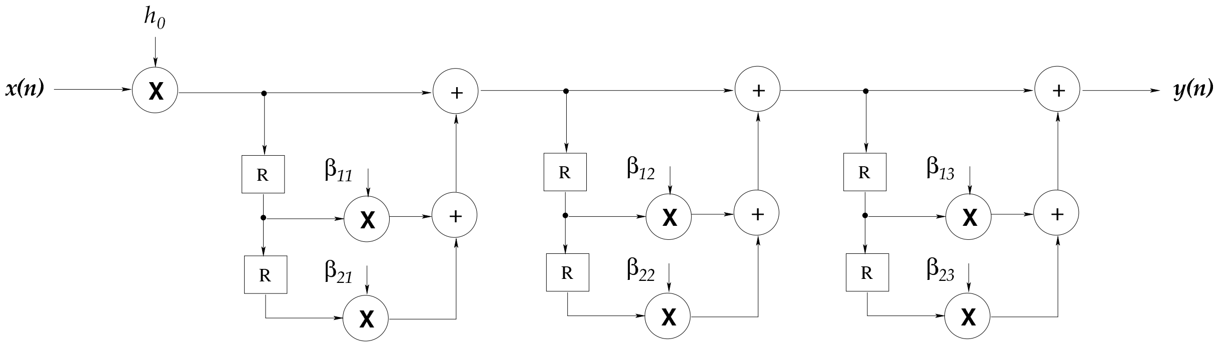

Figure 3.7: The cascade FIR filter realization

The cascade FIR filter realization represented by Equation 3.3 is illustrated in Figure 3.7.

- *Polyphase Structure*
  To explain the polyphase structure consider the transfer function of a 9 tap FIR filter expanded below:

$$\begin{aligned} H(z) =& h_0 + h_1 z^{-1} + h_2 z^{-2} + h_3 z^{-3} + \\ & h_4 z^{-4} + h_5 z^{-5} + h_6 z^{-6} + h_7 z^{-7} + h_8 z^{-8}. \end{aligned} \tag{3.4}$$

The terms in Equation 3.4 can be grouped into say three terms such as:

$$\begin{aligned} H(z) =& (h_0 + h_3 z^{-3} + h_6 z^{-6}) + \\ & (h_1 z^{-1} + h_4 z^{-4} + h_7 z^{-7}) + \\ & (h_2 z^{-2} + h_5 z^{-5} + h_8 z^{-8}), \end{aligned} \tag{3.5}$$

and the 9 tap FIR filter transfer function can be rewritten as:

$$H(z) = E_0(z^3) + z^{-1} E_0(z^3) + z^{-2} E_3(z^3), \tag{3.6}$$

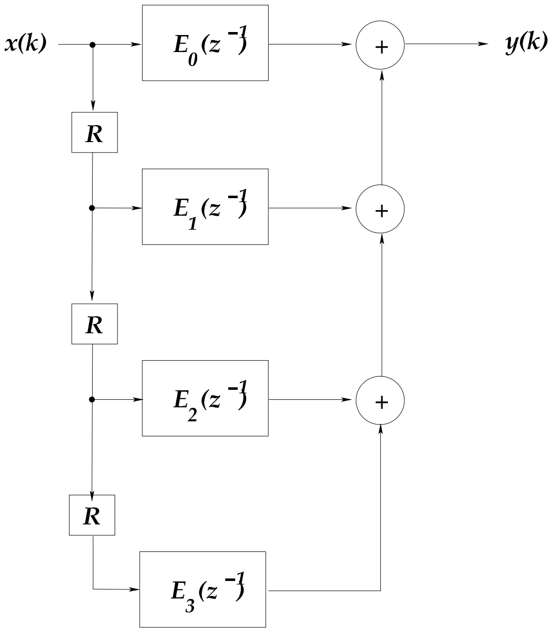

Figure 3.8: The Polyphase FIR filter structure

where,

$$\begin{aligned} E_0(z) &= h_0 + z^{-1}h_3 + z^{-2}h_6, \\ E_1(z) &= h_1 + z^{-1}h_4 + z^{-2}h_7, \\ E_3(z) &= h_2 + z^{-1}h_5 + z^{-2}h_8. \end{aligned} \tag{3.7}$$

Each of the terms is in Equation 3.7 represents a FIR filter producing a partial result and the final result is computed by adding the partial results as illustrated in Figure 3.8.

- *Cascaded Lattice Structure*
  An elegant realization of the FIR filter is the cascaded lattice structure shown in Figure 3.9. The cascaded lattice filter assumes a FIR transfer function of the form:

  $$H_N(z) = 1 + \sum_{n=0}^{N} p_n z^{-n}. \tag{3.8}$$

  An arbitrary FIR can be implemented by normalizing its factors such that $h_0 = 1$ and multiplying with h_0 at the input hence realizing the function: $H(z) = h_0 H_N(z)$ with $p_n = h_n/h_0$

  In this structure, butterfly-like processing units each featuring a register, a couple of multipliers in the feed-forward butterfly branches and a couple of adders at the output of the processing units.

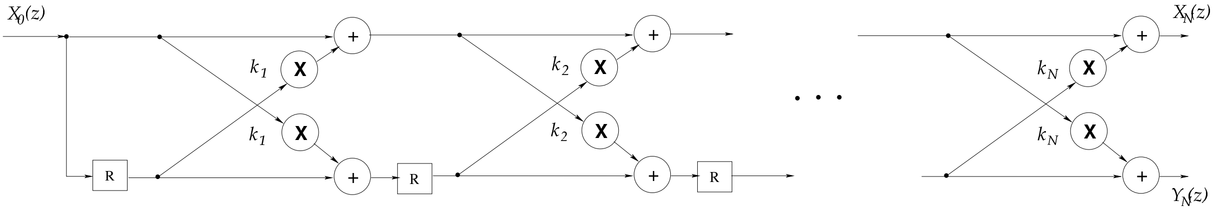

Figure 3.9: The FIR filter cascaded lattice structure

The coefficients of the multipliers are computed recursively by having the first coefficient equal to the highest index coefficient p_N and then computing the new coefficients according to the following relation:

$$p'_n = \frac{p_n - k_N p_{N-n}}{1 - k_N^2}, \qquad 1 \leq n \leq N-1$$

where k_N is the coefficient of the highest index from the previous stage, p_n are the remaining coefficients and p'_n are the new generated coefficients. The maximum indexed generated coefficient is taken as the next coefficient and so on.

We notice from Figure 3.9 that the cascaded lattice structure implementation of FIR filters is more expensive than others in terms of the amount of hardware resources required. Lattice structures –slightly different than the one shown in Figure 3.9– are commonly used to implement other types of digital filters.

3.2 Discrete Fourier Transform

Transformations are very powerful tools in signal processing. Dealing with signals transformed domains eases particular computations and allows better understanding of the properties of the signal. The Fourier transform is one of the most popular transformations representing the signal in the frequency domain. The Fourier transform is used in many DSP applications such as radar implementations, multimedia processing, Orthogonal Frequency Division Multiplexing (*OFDM*) and many other mobile communication applications. The Discrete Time Fourier Transform (*DTFT*) is given by the following formula:

$$X(e^{j\omega}) = \sum_{n=-\infty}^{\infty} x(n)e^{-j\omega n}. \tag{3.9}$$

Computation of such a formula is obviously very expensive; assuming however, that the input signal is bounded and of length N instances and zero otherwise we reach the

Discrete Fourier Transform (*DFT*) representation of the signal using the DFT equation below:

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn}, \qquad 0 \leqslant k \leqslant N-1, \tag{3.10}$$

where the complex numbers W_N^{kn} are known as the twiddle factors and are given by:

$$W_N^{kn} = e^{-j\frac{2\pi kn}{N}} \quad = e^{-j\frac{\omega k}{N}}, \tag{3.11}$$

and its inverse (*IDFT*) is given by:

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) W_N^{-kn}, \qquad 0 \leqslant n \leqslant N-1, \tag{3.12}$$

The amount of computations needed for each sample of the transformation in Equation 3.12 is N complex multiplications and $N-1$ complex additions which yields N^2 complex multiplications and $N^2 - N$ complex additions [2]. This high number of complex operations ($O(N^2)$) makes it prohibitively expensive to build a feasible DFT processor because of the huge amount of multipliers and address requirement.

3.2.1 Fast Fourier Transform

The Cooley and Tukey Fast Fourier Transform (FFT) algorithm [16] made use of the symmetry and periodicity properties of the complex twiddle factors which resulted in considerable reductions in the amount of complex computations bringing down the order of complex computations to $O(Nlog_2N)$.

Assuming a DFT size of a power of 2, the Cooley Tukey FFT algorithm uses a divide and conquer technique to reduce the number of computations one step at a time. The two main FFT algorithms are denoted the *Decimation-in-Time* and *Decimation-in-Frequency* algorithms.

- The Decimation-in-Time Algorithm:
  In the decimation-in-time (*DIT*) algorithm, the time-domain input values are split

---

[2] this is equivalent to $4N^2$ real multiplications and $4N^2 - 2N$ real additions

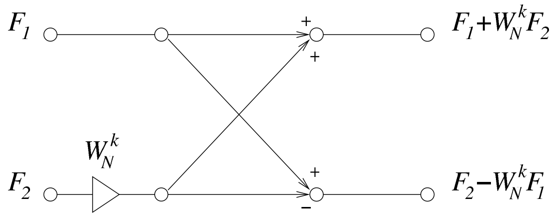

Figure 3.10: Simplified Basic butterfly SFG for the DIT FFT

into odd and even parts,

$$
\begin{aligned}
X(k) = & \sum_{n=0}^{N/2-1} x(2n)W_N^{2kn} + \sum_{n=0}^{N/2-1} x(2(n+1))W_N^{2k(n+1)} \\
= & \sum_{n=0}^{N/2-1} x(2n)e^{-j\frac{\omega}{N}2kn} + \sum_{n=0}^{N/2-1} x(2n+1)e^{-j\frac{\omega}{N}k(2n+1)} \\
= & \sum_{n=0}^{N/2-1} x(2n)e^{-j\frac{\omega}{N}2kn} + \sum_{n=0}^{N/2-1} x(2n+1)e^{-j\frac{\omega}{N}2kn}e^{-j\frac{k\omega}{N}} \\
= & \sum_{n=0}^{N/2-1} x(2n)e^{-j\frac{\omega}{N/2}kn} + W_N^k \sum_{n=0}^{N/2-1} x(2n+1)e^{-j\frac{\omega}{N/2}kn} \\
= & \underbrace{\sum_{n=0}^{N/2-1} x(2n)W_{\frac{N}{2}}^{kn}}_{N/2\ point\ FFT\ for\ even\ inputs} + W_N^k \underbrace{\sum_{n=0}^{N/2-1} x(2n+1)W_{\frac{N}{2}}^{kn}}_{N/2\ point\ FFT\ for\ odd\ inputs}
\end{aligned}
$$

Each of the resulting odd and even parts FFT can be further broken into two parts until no further splitting is possible when only two input values are available. This reduction results in the basic SFG simplified as illustrated in 3.10 which is referred to as the Radix-2 decimation-in-time basic butterfly.

Furthermore, this recursive splitting operation results in dividing the N-point DFT operation into log_2N stages each stage consisting of $N/2$ basic butterfly elements. The output of even butterflies feeding the upper branches input of the next stage of butterflies and the odd butterflies feeding the lower branches input of the next stage of butterflies as the decimation-in-time FFT SFG shows in Figure 3.11. It is worth mentioning that the recursive splitting into odd and even sets results in the input being sorted in the bit-reversed order [3] while the output's original order is preserved as shown in Figure 3.11

- The Decimation-in-Frequency Algorithm:

---

[3] i.e. when representing the indices in binary reversing its order putting the MSB at the right and the LSB at the left

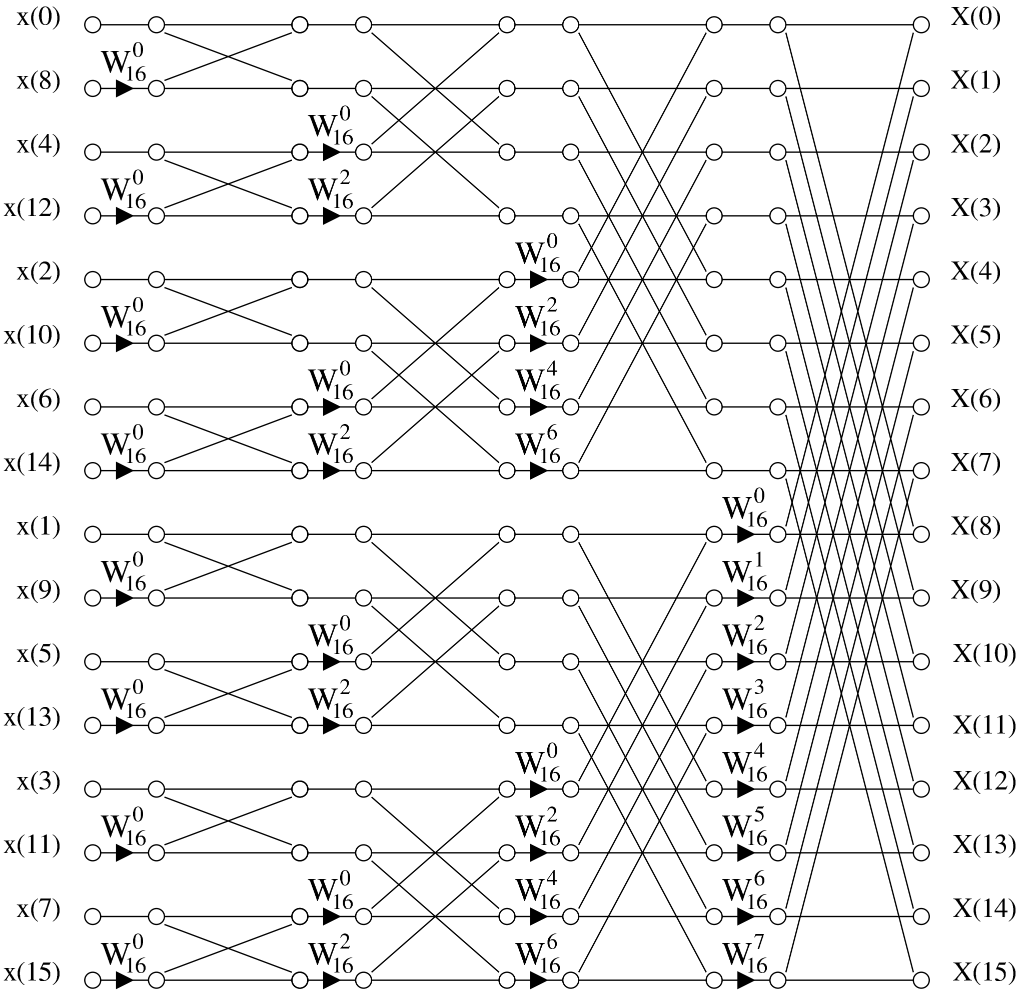

Figure 3.11: The decimation-in-time FFT SFG

In the Decimation-in-Frequency (*DIF*), the resulting frequency components of the DFT are split into two halves: top and bottom as follows:

$$X(k) \;=\; \sum_{n=0}^{N/2-1} x(n)W_N^{kn} + \sum_{n=N/2}^{N-1} x(n)W_N^{nk},$$

performing a time domain translation on the right hand term we get,

$$\begin{aligned} X(k) &= \sum_{n=0}^{N/2-1} x(n)W_N^{kn} + W_N^{\frac{N}{2}k} \sum_{n=0}^{N/2-1} x(n+\frac{N}{2})W_N^{nk} \\ &= \sum_{n=0}^{N/2-1} x(n)W_N^{\frac{N}{2}k} x(n+\frac{N}{2})W_N^{nk}, \end{aligned}$$

but since $W_N^{\frac{N}{2}}k = -1^k$,

$$X(k) = \sum_{n=0}^{N/2-1} [x(n) + (-1)^k x(n+\frac{N}{2})]W_N^{nk},$$

then for even $k = 2l$ we get,

$$\begin{aligned} X(2l) &= \sum_{n=0}^{N/2-1} [x(n) + x(n+\frac{N}{2})]W_N^{2nl} \\ &= \sum_{n=0}^{N/2-1} [x(n) + x(n+\frac{N}{2})]W_{\frac{N}{2}}^{nl}, \qquad 0 \leqslant l \leqslant \frac{N}{2} - 1, \end{aligned}$$

denote $X(2l) = X_0(k)$ which is the $\frac{N}{2}$ point DFT of $x_0(n) = x(n) + x(\frac{N}{2} + n)$, similarly, for odd $k = 2l + 1$ we have,

$$\begin{aligned} X(2l+1) &= \sum_{n=0}^{N/2-1} [x(n) - x(n+\frac{N}{2})]W_N^{2n(l+1)} \\ &= \sum_{n=0}^{N/2-1} [x(n) - x(n+\frac{N}{2})]W_{\frac{N}{2}}^{nl} W_N^n, \qquad 0 \leqslant l \leqslant \frac{N}{2} - 1, \end{aligned}$$

and denote $X(2l+1) = X_1(k)$ which is the $\frac{N}{2}$ point DFT of $x_1(n) = [x(n) + x(\frac{N}{2} + n)]W_N^n$. Thus, performing the above split recursively we reach the basic butterfly SFG element shown in 3.12 leading to the Radix-2 decimation in frequency SFG illustrated in 3.13. Here again bit reverse ordering is observed but this time at the output side and the input is provided in normal order.

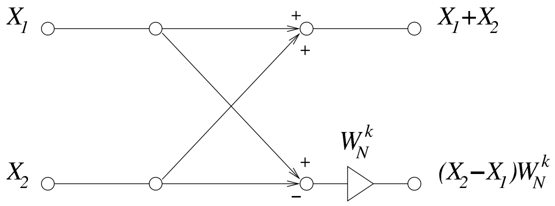

Figure 3.12: The basic butterfly SFG of the decimation-in-frequency FFT

3.2.2 Fast Fourier Transform Architectures

Parallel implementation of the SFGs shown in Figures 3.11 and 3.13 of the Radix-2 decimation-in-time and decimation-in-frequency is unquestionably costly. especially for large FFT sizes. Moreover, most applications do not require very fast FFT computations. This makes parallel implementations of the FFT excessively powerful and expensive. On the other hand, pure sequential implementations performance do not suffice many applications requirements.

In this context the *"pipelined"* implementations of the FFT present a trade off between pure parallel and pure sequential implementations of the FFT.

In the literature there is a wide variety of VLSI architectures proposed for implementing the above and other FFT algorithm. In the following some of the Radix-2 architectures will be presented and briefly described. We are more concerned with Radix-2 architectures because their basic building element is the Radix-2 butterfly which incorporates adders and multipliers of two operands and a single output which is more likely to coincide with our future reconfigurable coarse-grained processing elements of our proposed CGRC coprocessor.

- *The Radix-2 Single-Path Delay Feedback Architecture*
  The Radix-2 Single-Path Delay Feedback Architecture (*R2SDF*) [79] uses a very elegant strategy to schedule the computations of either the SFG of Figure 3.11 or 3.13. Figure 3.14 show the R2SDF implementation of the DIF FFT SFG The basic component is the *Dynamic Butterfly* whose operation is shown in Figure 3.15.

  The basic idea of the R2SDF architecture operation is based on the observation that the SFG of an $N = C^2$ point FFT can be divided into C clusters each having a set of butterflies whose inputs are separated by a power of two clock cycles (assuming that the inputs are streaming in serially). Therefore, the received input should be saved for a power of two, say d clock cycles and then operated on with the second input arriving after d cycles. A shift register (or a FIFO) is then needed to store the input data periodically and providing them at the proper instant to the dynamic butterfly input.

  The sizes of the feedback FIFOs therefore, start with 2^{C-1} and end with 1 for an N-point decimation-in-frequency FFT to coincide with the corresponding butter-

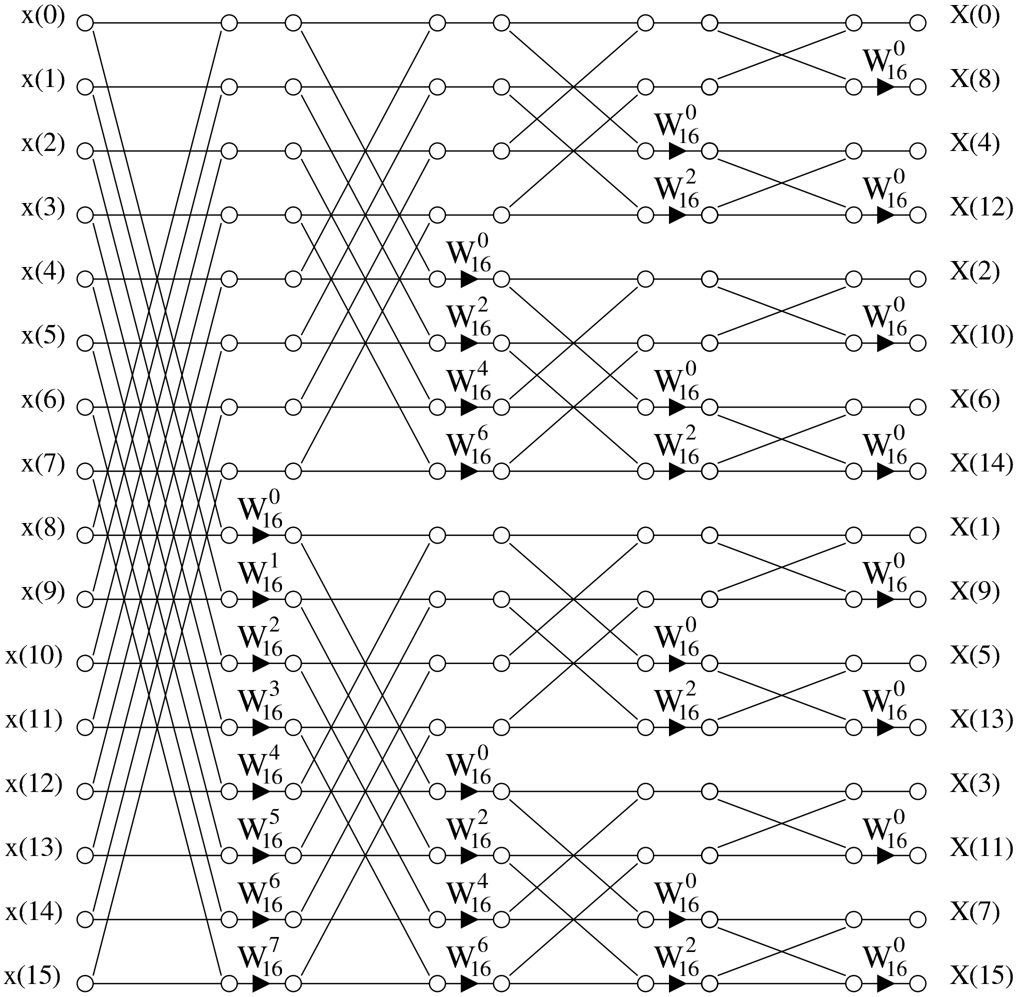

Figure 3.13: The decimation-in-frequency FFT SFG

fly input and output displacement of each stage. Similarly, for a decimation-in-time R2SDF FFT processor, the FIFO sizes start with 1 and end with 2^{C-1} as is suggested by their SFGs.

As such, each dynamic butterfly stores the upper branch input data in the first phase of its operation and in the second phase it carries out the butterfly as the bottom branch inputs arrive operation.

While in the butterfly operation phase, the dynamic butterfly passes to the next butterfly stage the upper branch outputs while utilizing the FIFO to shift in the lower outputs that are provided to the next stage in the next phase of operation ensuring thus correct ordering of data input to each of the butterfly stages.

Since the separation between the inputs of butterfly at each stage is two times that of the next butterfly in the DIF FFT SFG (or half that of the next stage in the DIT FFT SFG) and since the dynamic butterflies are half the time active and half

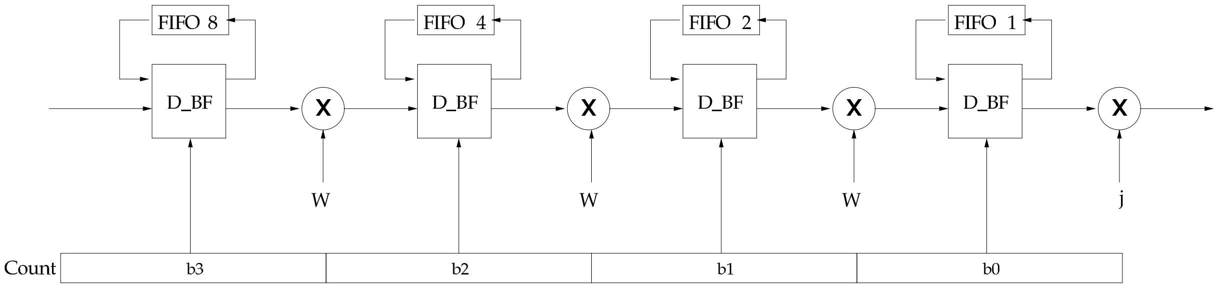

Figure 3.14: The R2SDF implementation of the DIF FFT SFG

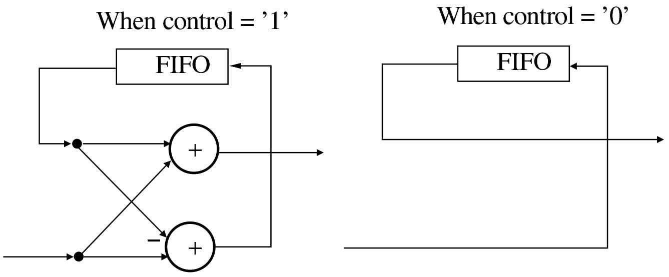

Figure 3.15: The Dynamic Butterfly component of the R2SDF architecture

the time passing data the dynamic butterflies can be controlled by simply a serial counter count bits with each bit connected to the appropriate butterfly stage.

Each dynamic butterfly is cascaded with a multiplier to multiply the result with the appropriate twiddle factor stored in a ROM whose addresses are provided also by the counter.

The R2SDF architecture described above, requires the following hardware resources:

- $N-1$ dynamic butterfly stages each with two complex adders/subtracters and multiplexors to select inputs and outputs.
- $N-1$ FIFOs of different sizes collectively $\sum_{s=1}^{N} 2^{s-1}$ words of storage.
- $C-1$ complex multipliers.
- $C-1$ ROMs for storing the twiddle factors each $N/2$ words of size.

- *The Radix-2^2 Single-Path Delay Feedback Architecture*
  The Radix-2^2 Single-Path Delay Feedback Architecture (*R2^{2}SDF*) FFT was introduced in [42, 41, 40]. The R2^{2}SDF algorithm and architecture aims at reducing the hardware requirements of multipliers to $(N-1)/2$. Using a change of variable technique putting $n =< \frac{N}{2}n_1 + \frac{N}{4}n_2 + n3 > N$ and $n =< k_1 + 2k_2 + 4k_3 > N$, as in [42, 41, 40] reaching the following relation of the FFT transform:

$$X(k_1+k_2+k_3) = \sum_{n_3=0}^{N/4-1} [H(k_1,k_2,n_3)W_N^{n_3(k_1+k_2)}]W_{\frac{N}{4}}^{n_3k_3},$$

where,

$$H(k_1,k_2,n_3) =$$

$$\underbrace{\overbrace{[x(n_3)+(-1)^{k_1}x(n_3+\frac{N}{2})]}^{BF\ I}+(-j)^{k_1+2k_2}\overbrace{[x(n_3+\frac{N}{4})+(-1)^{k_1}x(n_3+\frac{3}{4}N)]}^{BF\ I}}_{BF\ II}. \tag{3.13}$$

As seen from Equation above the R2^{2}SDF frequency elements has are composed of a big butterfly elements (*BF II*) joining its two inputs with a j multiplication and both elements are in tern formed by throughly a normal butterfly. The resulting R2^{2}SDF SFG is shown in Figure 3.16. Note that the j multiplication takes place only at the lower branch output of only the bottom half butterflies which necessitates more complicated control. Fortunately this control requirement can be still met using the simple counter lines but this time with an extra and gate and multiplexers as shown in Figure 3.17. The complete R2^{2}SDF FFT processor structure is shown in Figure 3.18.

- *The Radix-2 Multi-Path Delay Commutator Architecture*
  The Radix-2 Multi-Path Delay Commutator Architecture (*R2MDC*) FFT pipelined architecture [79] operates by rescheduling the butterfly inputs through feed-forward shift registers. As Figure 3.19 illustrates, the R2MDC architecture consists of delay elements, dynamic direct pass/criss-cross switches, static butterfly units and multipliers. The operation of the dynamic switches is shown in Figure 3.20. Since the butterflies receive and produce data in pairs, they need not be dynamic like their R2SDF counterparts. Rather data rescheduling is carried out by the delay elements and the dynamic switches.

  As data enters the R2MDC network, the upper input branch holds the first $N/2$ inputs (where N is the number of points of the FFT) and when the second N inputs arrive the correct pair of data is processed in the butterfly unit.

  Note that from Figure 3.13 multiplication is performed only at the bottom output branch of the butterflies. Therefore, the multipliers are placed only at the bottom branch output. As can be also observed from the SFG of Figure 3.13 the upper branch outputs and lower branch outputs supply the two half size butterflies of the next stage such that the upper butterfly cluster of the next stage inputs are supplied solely from the upper branch outputs of the preceding butterfly stage. Similarly, the bottom cluster of samples its inputs only from the bottom outputs

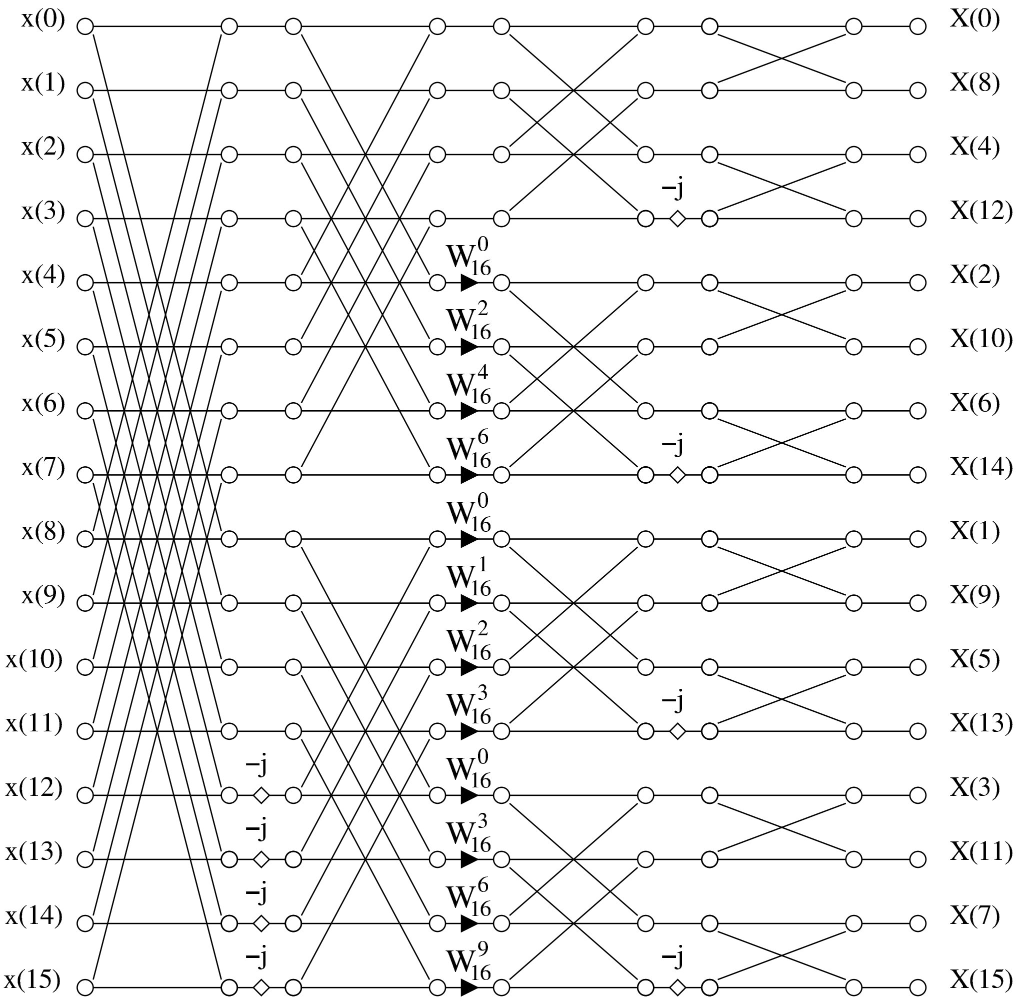

Figure 3.16: The R2^{2}SDF SFG [40]

of the preceding butterfly stage. Thus, as the switch is in the direct pass mode, the upper branch outputs are shifted in the delay elements and after $N/4$ cycles $N/4$ samples are stored and the switch is toggled to the criss-cross mode providing correct ordered input pairs to the next butterfly stage and simultaneously storing the bottom outputs of the preceding butterfly stage that were already preserved for $N/4$ to be used as the upper branch inputs in the bottom butterfly cluster.

Obviously, since the switching activity frequency increases by a factor of two for every stage, the control of the R2MDC architecture can also be generated by a counter. The R2MDC architecture is widely used in many applications but it sufferers from a low (25%) hardware utilization.

- *The Radix-2^2 Multi-Path Delay Commutator Architecture*
  In [85] the authors work with the DFT formula reaching a SFG very much similar to that of Figure 3.16 proposed by [42, 41, 40]. The principle profit of this

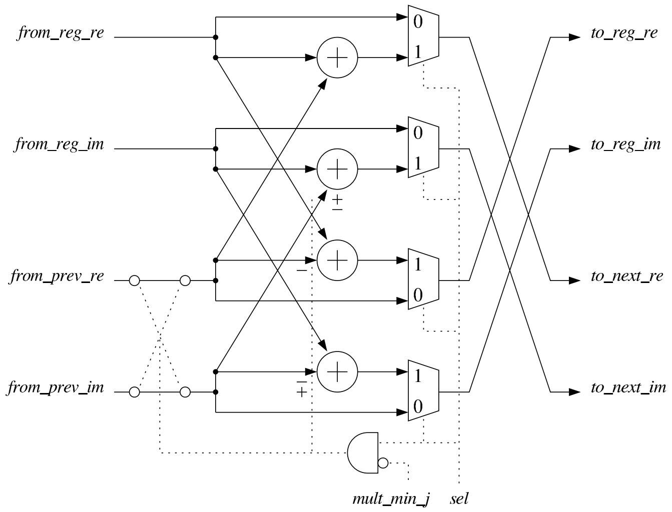

Figure 3.17: Basic butterflies of the R2^2SDF FFT [40]

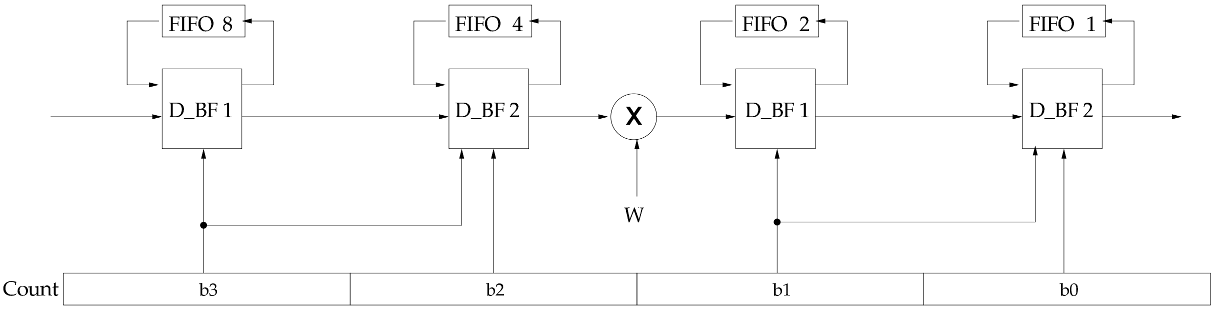

Figure 3.18: The R2^2SDF FFT processor structure [40]

approach is to gather the multiplications cascaded after two butterfly stages and place them instead after one butterfly switch and cascading a regularly clustered trivial $-j$ multiplication after the other.

In contrast to [42, 41, 40], the authors of [85] choose to implement the resultant FFT in an R2MDC structure. We denote this implementation the R2^2MDC architecture to differentiate between it and the regular R2MDC one and to point out the resemblance in this approach and its SFG to the R2^2SDF architecture and SFG discussed above.

In the R2^2MDC architecture the trivial $-j$ multiplications are carried out in special branches at the bottom output of each butterfly unit as ruled by the SFG. A switch selects to pass an un-manipulated output or the $-j$ multiplied one. The R2^2MDC architecture is illustrated in Figure 3.21. Because of the regularity of the SFG of the R2^2MDC SFG the switching activity control of the structure of Figure 3.21 can be simply generated by a counter.

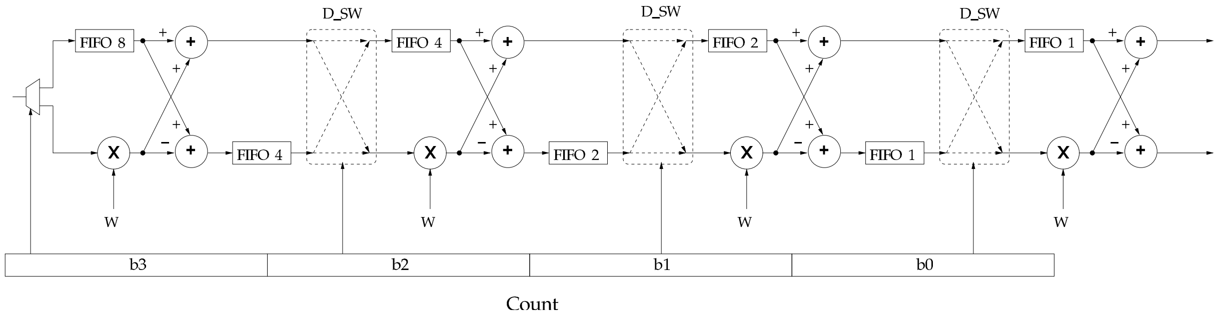

Figure 3.19: The FFT R2MDC architecture [79]

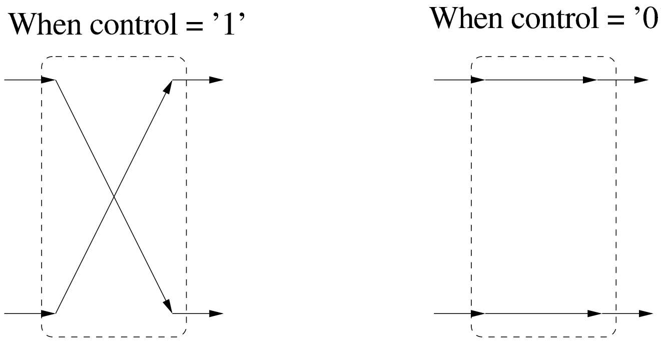

Figure 3.20: Operation of the dynamic switches of the R2MDC architecture [79]

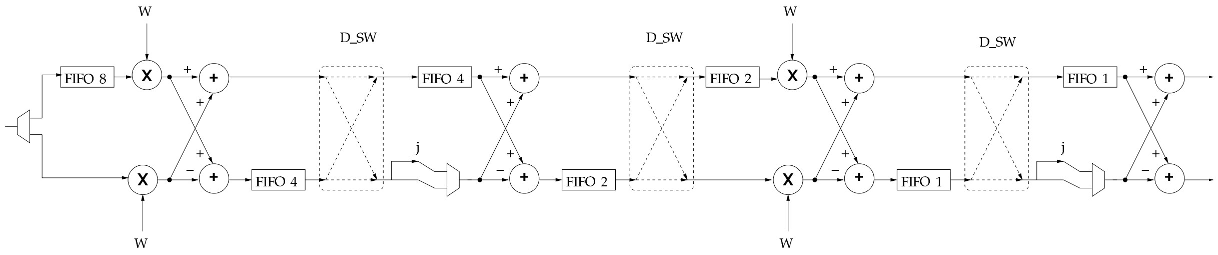

Figure 3.21: The R2^2MDC architecture [85]

3.3 Discrete Cosine Transform

The Discrete Cosine Transform *DCT* is a popular transformation that is used nowadays in many applications. From a mathematical point of view the DCT inferior to the DFT since it lacks many of the DFT's elegant mathematical properties. However the DCT has other features that make it very attractive especially in image, audio and video compression applications.

Performing the DCT on a digital image produces a compacted form of the original image which can be efficiently compressed using lossy compression techniques. Compression's main profit is the resulting reduction in the amount of memory needed for storage, therefore, the reduction in the time required to process data and increases the channel bandwidth efficiency.

Although DFT can also be used in compression algorithms the DCT offerers better properties that makes it preferable for data compression. The DCT computations for example involve only real operations in contrast to the DFT that whose operations are all complex. The DCT energy compaction properties are also better than those of the FFT. The above and other features qualified the DCT to be a popular tool used in many applications in video teleconferencing applications, in ISDN multimedia communications, in satellite video transmission and in digital facsimile transmission.

The DCT can be expressed in vector form as:

$$\mathbf{X_c} = [C_N]_{mn}\mathbf{x}, \qquad m,n = 0,\ 1,\ ...,\ N-1 \tag{3.14}$$

where x is the input vector of length N and $\mathbf{X_c}$ is the resultant transformed vector. $[C_N]$ is the $m \times n$ transformation coefficients matrix.

The DCT is classified into four main types: I, II, III, and IV and their transform matrices are given below:

$$[C^{I}_{N+1}]_{mn} = \sqrt{\frac{2}{N}}\left[b_m b_n cos\left(\frac{mn\pi}{N}\right)\right], \qquad m,n = 0,\ 1,\ ...,\ N \tag{3.15}$$

$$[C^{II}_{N}]_{mn} = \sqrt{\frac{2}{N}}\left[b_m cos\left(\frac{m(n+\frac{1}{2})\pi}{N}\right)\right], \qquad m,n = 0,\ 1,\ ...,\ N-1 \tag{3.16}$$

$$[C^{III}_{N}]_{mn} = \sqrt{\frac{2}{N}}\left[b_n cos\left(\frac{(m+\frac{1}{2})n\pi}{N}\right)\right], \qquad m,n = 0,\ 1,\ ...,\ N-1 \tag{3.17}$$

$$[C^{IV}_{N}]_{mn} = \sqrt{\frac{2}{N}}\left[cos\left(\frac{(m+\frac{1}{2})(n+\frac{1}{2})\pi}{N}\right)\right], \qquad m,n = 0,\ 1,\ ...,\ N-1 \tag{3.18}$$

where, the DCT type is shown as superscripts of C matrices and,

$$b_m = \begin{cases} \frac{1}{\sqrt{N}}, & \text{if } m = 0 \text{ or } m = N \\ 1, & \text{otherwise} \end{cases} \tag{3.19}$$

The DCT-Type-I of Equation 3.15 is symmetric and therefore, its inverse is the same. This is also the case of the DCT-Type-IV given by Equation 3.18. The case is however not the same for DCT-Type-II and DCT-Type-III where the transformation matrices are the transpose of each other and they are therefore the inverse of each other.

The main difficulty of DCT implementations is that the transformation matrices do not enjoy all the attractive properties that the DFT twiddle factors posses a matter that makes minimizations similar to those that led to the FFT algorithms not possible. As a result, the DCT SFGs do not exhibit the regularity features of the FFT SFGs. In this context a lot of work has been done to implement fast DCT algorithms via simplifying the DCT flow graphs. For additional information the reader is referred to [66] where a good survey of different fast DCT algorithms are introduced. Irregularities make the realization of efficient pipelined processor difficult. This is mainly because the input data of different processing nodes are of variable distances apart. Moreover, the operations at different stages of the SFG are not necessarily similar. The aforementioned problems cause the need of having different types of processing elements to suit all types of operations at different portions of the SFG. Moreover, there is also the need of the addition of more complicated control to help time and schedule operations and ensures smooth and continuous data flow through the processor. obviously, these special considerations result in designs of lower efficiencies.

3.3.1 Reported pipelined DCT Implementations

In [38, 47] Hsiao et al. produced the 8-point DCT-Type-II SFG as well as its inverse (DCT-Type-II) SFG. What is interesting in this realization is that the irregularities in the SFG are separated from the regular parts. This paves the way for designing simple pipelined data path for the regular part and cascade it with a more complicated pipelined data path to carry out the computation for irregular parts of the SFGs. Moreover, similarities in the regular parts of the aforementioned SFGs facilitate the reuse of parts of the hardware to implement both the DCT and its inverse in one circuit.

The DCT/IDCT pipelined architecture was implemented in [38, 47] using a clever Radix-2 Single Path Delay Commutator network. A couple of multiplexors are used here to choose between for DCT or IDCT processing. The processing elements of the regular parts of the DCT/IDCT SFG are controlled multiplexor passes data of correctly displaced time stamps to the adder/subtracter unit. Different stages of the regular part of the SFG have different time displacements between its branches and therefore, the number of delay elements at every processing element stage differs.

In the pre and post irregularities in the DCT/IDCT SFGs data is either passed directly or saved so pairs of data (formed current and old displaced by a given amount of time) are processed together. There, controlled multiplexors are used to either directly pass input data or processed data pairs.

Also, multiplications with the transformation coefficients is done by multipliers placed in the correct positions of the pipelined architecture. The coefficients are supplied through ROMs passing coefficients in the correct order. The control of the whole network could be generated by a counter.

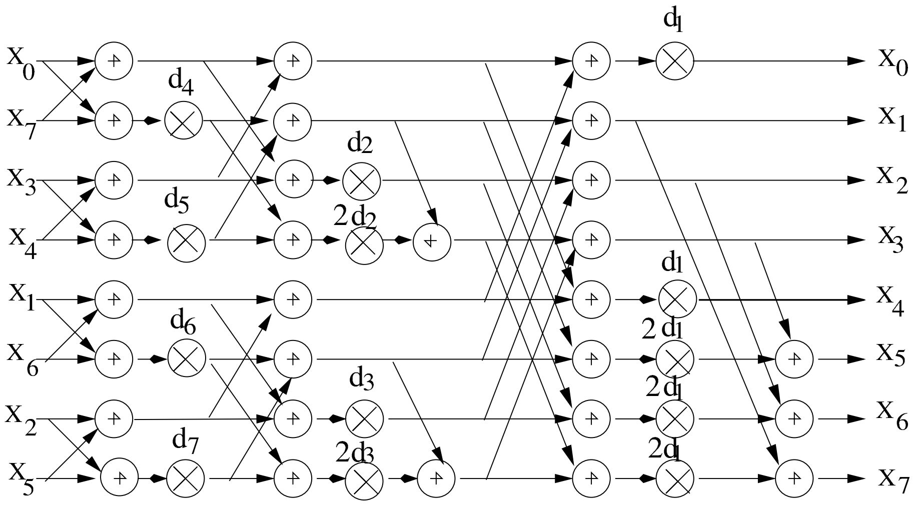

Figure 3.22: DCT-Type-II SFG Type A adopted by [87]

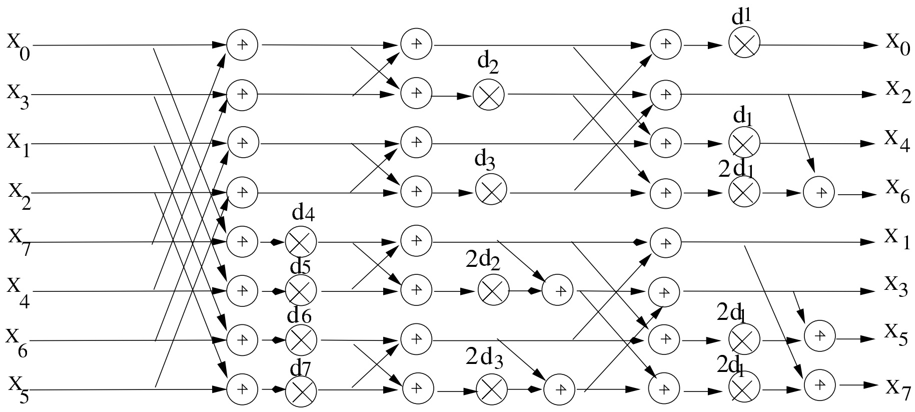

Figure 3.23: DCT-Type-II SFG Type B adopted in [87]

Another DCT-Type-II implementation are proposed in [66, 88, 67, 87] although here 2-dimensional DCT transformation is tackled. The 2-dimensional $N \times N$ DCT transformation is carried out by performing the DCT operation on an $N \times N$ input matrix rows and then performing the DCT transformation on the resultant columns. Again, tuned SFGs of the DCT are utilized for simplifying the pipelined design process.

The reference DCT-Type-II SFGs considered in [87] for example is shown in Figures 3.22 showing the DCT II type a SFG and 3.23 showing the DCT II type B SFG. Firstly, input data should be reordered to be supplied serially to the pipelined DCT processor. The reordering process is carried out with a cascaded combinations of a very interesting R2SDF-like processing units denoted Shift-Exchange Units (*SEU*s). Inputs or previously stored data are either stored or passed through according to a control signal thus shuffling the input data.

The DCT operation of Figure 3.22 can then performed on the serially streaming input

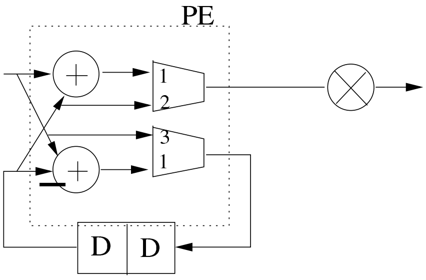

Figure 3.24: R2SDF butterfly unit proposed by [87]

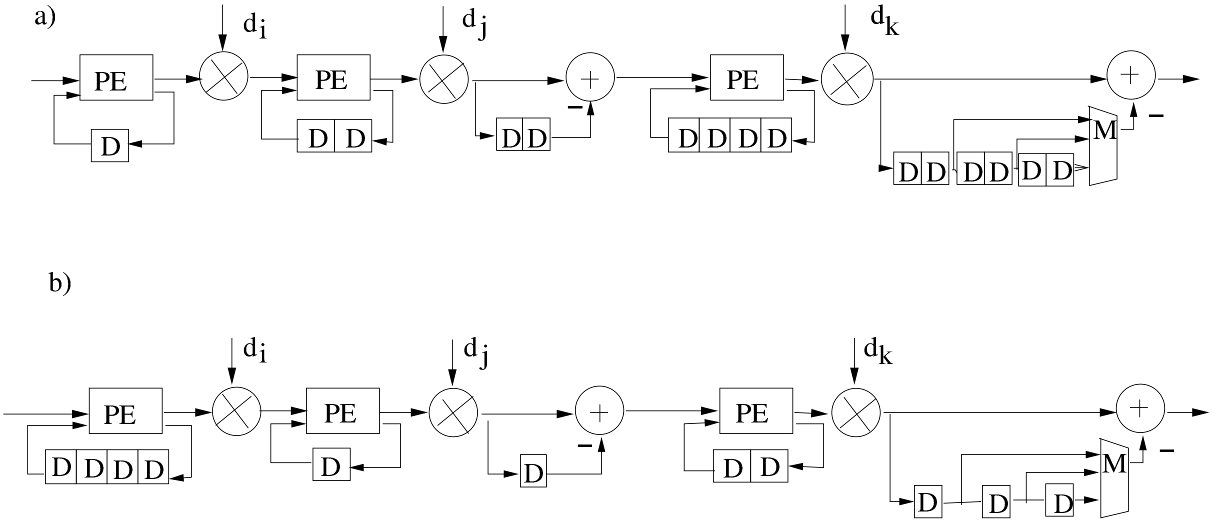

Figure 3.25: The R2SDF processing of the DCT II (a) for the type-A SFG and b) processes for the type B SFG shown in Figures 3.22 and 3.23) as in [87]

data with the aid of specialized R2SDF pipelined butterfly units shown in Figure 3.24. Naturally a multiplier is attached to the output of the butterfly units for the multiplication with the transform coefficients.

The DCT processed data is then transposed and reordered with a cascaded combinations of SEU of suitable sizes. The transposed data is then processed again with a pipelined DCT unit implementing the SFG of Figure 3.23. The DCT pipelined units implementing the SFGs of Figures 3.22 and 3.23are shown in Figure 3.25.

In [88, 67] a similar approach is used although here the basic butterfly operation is done with a feed-forward Radix-2 Single path Delay Commutator. It is worth mentioning that all the control signals can be generated by a counter.

3.4 Viterbi Decoding

Shannon's theory of channel capacity limit states that *"For a given channel there exists a channel capacity limit for error-free communications"*. The channel capacity limit is given by:

$$C = Blog(\frac{S}{N} + 1).$$

where, C is the channel capacity in $bits/sec$.
B is the channel bandwidth in Hz.
$\frac{S}{N}$ is the linear signal to noise ratio.

Hence, error correction coding techniques are exploited and developed to reach the sought theoretical channel capacity for error-free communications limit. There are two types of error correction codes: *Linear Codes* and *Convolutional Codes*. In Linear Codes the complete message (original data and redundancy bits) should be completely received before decoding starts. In Convolutional Codes decoding can start before the reception of the complete message. The Viterbi algorithm, introduced in [96], emerged as a very popular approach for decoding convolutionally encoded messages.

3.4.1 Convolutional Coding

A convolutional encoder is a series of cascaded storage elements shifting from its input end the original message. The output encoded message is generated by modulo adding (XORing) samples from the current and delayed input data (XOR) according to given generator polynomials expressed usually in octal numbers. The code rate r is defined as the ratio of the number of input raw bits by the number of output encoded bits $r = \frac{No.\text{ of inputs}}{No.\text{ of inputs}}$. The convolutional encoder is characterized by the constraint number which is one plus the number of storage elements $K = 1 + FF_{No}$ An example for a $K = 3, r = 1/2$ with generator polynomials $7_8, 5_8$ convolutional encoder is shown in Figure 3.26

The state diagram of the encoder illustrated in Figure 3.26 is shown in Figure 3.27. The state digram can also be represented as in Figure 3.27. Solid lines indicate a '1' input and broken lines indicate a '0' input. Encoded outputs (or codewords) are associated with each line.

3.4.2 The Viterbi Algorithm

In a Viterbi algorithm decoding is carried out by finding the most probable state at a given point of time. This is done by measuring the difference between the received

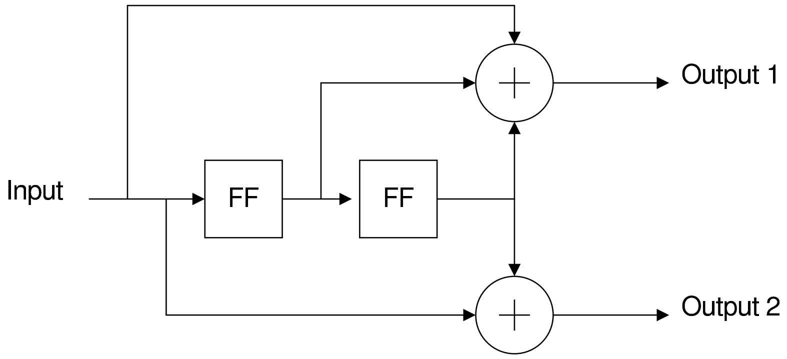

Figure 3.26: An example $K = 3, r = 1/2$, generating polynomials $7_8, 5_8$ convolutional encoder.

data and the code word of every branch. Then this measure (or metric) is added to that of the corresponding previous state. This way, the metric of every branch carries information about its own distance from the input and "inherits" information from its predecessor state. The best metric of each branch is then selected and becomes the new state metric. By the knowledge of the initial state and paths from each state to the other, the decoded data can be produced as depicted in Figure 3.27.

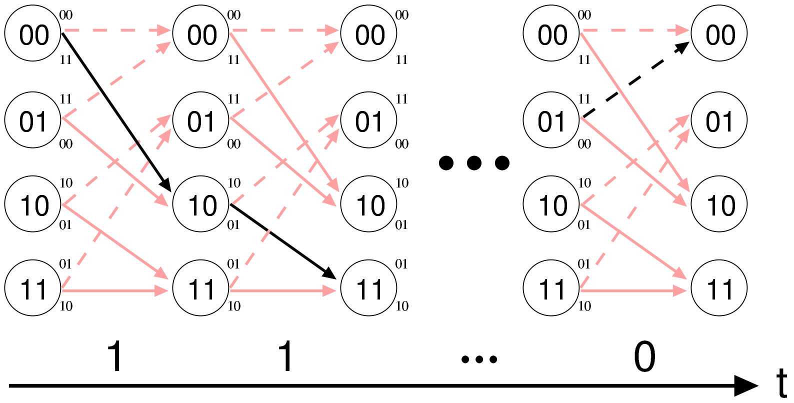

Figure 3.27: The decoding process in trellis diagram of a $K = 3, r = 1/2$ with generator polynomials of $7_8, 5_8$ convolutional encoder.

A trellis diagram is formed by concatenating many state (or code trellis) diagrams. Knowing the initial state and the decision or path from each state, the most probable state path of the encoder can be known and hence the data can be decoded. Figure 3.27 shows the forming of a survived path after tracing back the trellis diagram.

There are two main blocks in a Viterbi decoder: the Add Compare Select (ACS) block and the Trellis Window (TW) block. The ACS is a computational-intensive block and is a major contributor in the area of the decoder. The feedback of previous metrics prohibits pipelining of the ACS unit making it the main speed bottleneck. A convolutional encoder of a constraint number K has $S_n = 2^{(K-1)}$ states. A parallel Viterbi decoder for such an encoder would therefore contain S_n ACS cells for metrics computation. Each ACS cell uses the input data and other internal metrics to compute the new state metric.

The later block, namely the TW block, is mainly a block saving information about previously received data needed to produce the output. It consists of a memory that stores the ACS units' output information and some simple logic for decoding. The probability of the correctness of the guess on the output depends on the length of the trellis window. Obviously, the on-chip real estate is directly affected by TW length. Following we introduce briefly the two main Viterbi decoder's architectures: the Trace Back (TB) and the Register Exchange (RE).

- **The Trace-Back Technique** In the TB technique, (see Figure 3.28) the current best state is used to predict the previous state by referencing the corresponding value of the first column of the TW. This process is repeated for each computed state and the corresponding column till the end of the table. The decision bit selected by the latest computed state is passed as the output.

- **The Register-Exchange Technique** The RE technique is a straightforward technique for managing the decision vectors. In this technique, the TW is constructed of a bank of registers connected in the same manner as the trellis diagram. The newest decision bits are inserted in the left column of the TW as the oldest bits shift out at the right of the window. Each state is assigned a row and the newest decision bit is used as the select line of the multiplexers connecting the states. Each row now contains the survivor path of its corresponding state. The surviving paths then merge in the table with high probability and if the window is large enough the outputs of all the states should be the same. The output then can be sampled from any state. The structure in Figure 3.30 illustrates the Register-Exchange method in addition to the majority counter technique discussed in Section 3.4.3.

3.4.3 Proposed Parallel-Architecture Viterbi Decoder

In both the TB and the RE techniques discussed above the trellis table's length is theoretically infinite. Moreover, since the metrics' magnitudes increase constantly, the metrics' bit widths should also be infinite. For the TB approach, metrics of all states should be compared to choose the best initial state which is costly especially for large constraint lengths. Below we propose feasible solutions –that offers minimal or no loss in the decode quality– for the stated problems.

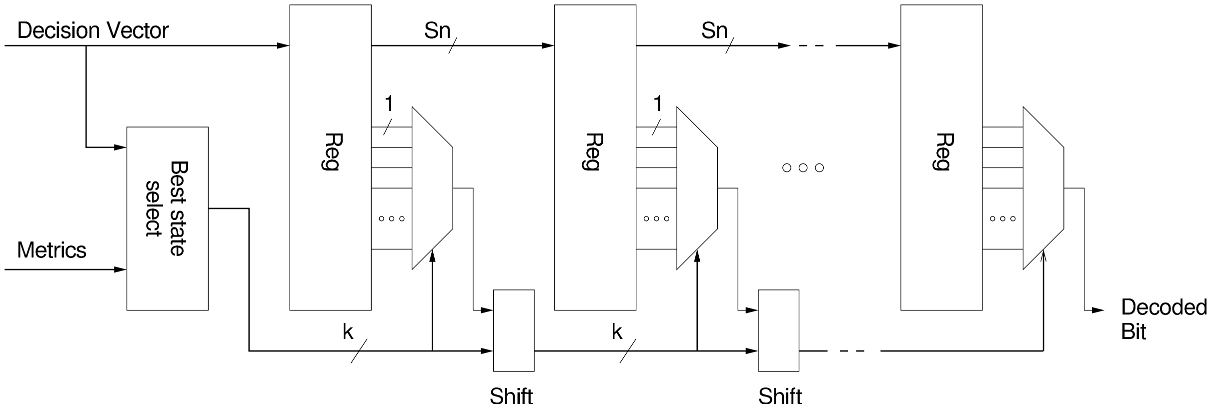

Figure 3.28: The TB architecture.

- *The Add Compare Select (ACS) Unit* For high constraint number parallel Viterbi decoders, the area, the power and the time consumed by the ACS unit is significant. To reduce this area, a Branch Metric Unit (BMU) that computes all possible 4 branch metrics needed is separated as common unit that provides all ACS units with the appropriate branch metrics. Another practical problem is that the magnitudes of the metrics are constantly increasing. Channel noise and accumulation of metrics are the reasons behind this increase. To solve this problem, the metrics should be frequently normalized. Attempting to keep the metrics values under a certain threshold by naive "shift-right" approaches will not work since this division will result in an imbalance between old and new metrics' weights and hence leads to wrong decisions. Therefore, the normalization process implies the insertion of an extra adder in each ACS unit. This accounts for a serious increase in area and hence, power consumption but moreover will degrade the speed. We propose to normalize only the MSBs of the metrics in the 2's complement representation. This way, one small size adder can be used instead of a full size adder. In our case, we needed only 3-bit adders to normalize 16-bit metrics. Testing only the 3 MSBs of any metric sufficed to signal for normalization. Figure 3.29 depicts the proposed ACS architecture.

- *Register Exchange* In a systolic RE architecture the trellis window is the main contributor for power consumption. This is because the data is being shifted through the complete table which results in a huge transition activity. To reduce the size of the window a majority counter was implemented as illustrated in Figure 3.30. Our simulations show a significant improve in the decode quality with that majority counter.

- *The proposed Multi-Path Trace-Back Architecture*
  In TB decoding the starting (initial) state can be found, ideally, by comparing the metrics of all states. The best metric corresponds to the best state number.

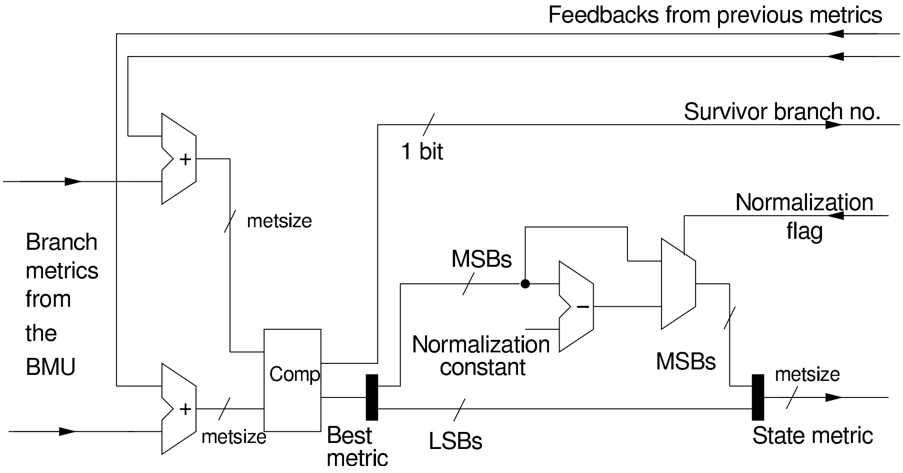

Figure 3.29: The proposed ACS unit.

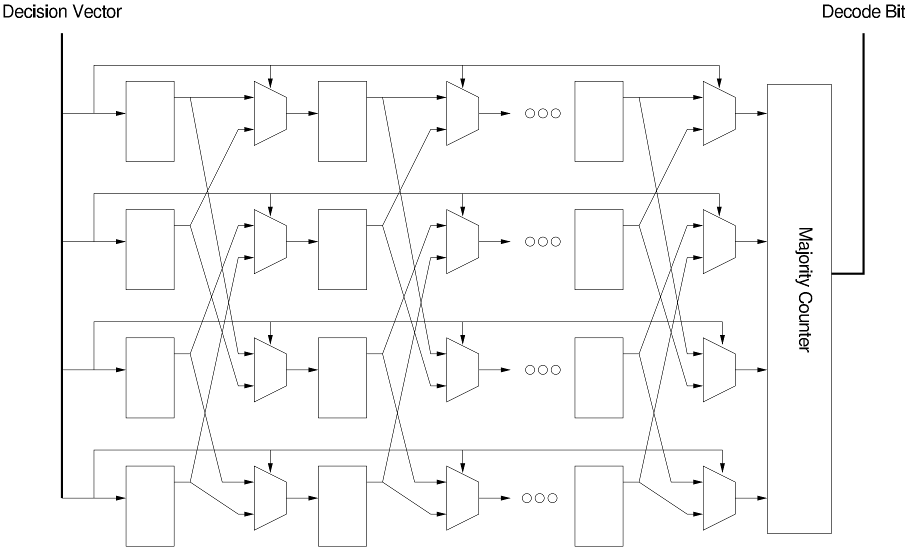

Figure 3.30: The proposed RE Viterbi decoder architecture.

Although it can be assumed that the initial state choice can be fixed supposing that the chosen path will eventually merge in the large enough TW, the fulfillment of this assumption will lead to a large TW and a decode quality close to that of the smaller RE approach. On the other hand, it is impractical and expensive to process $S_n = 2^{K-1}$ states in order to find the best one. One, therefore, needs an intelligent, yet cost and speed effective technique that facilitates choosing the initial state with no or minimal degradation in the Bit Error Rate (BER). Some approaches such as the M algorithm were proposed [12, 26]. A main drawback of that approach is the need of expensive sorting hardware.

The proposed idea is to guess the current state number (CS) based on our knowledge of the previous state number (PS). According to the structure of the convolutional encoder, given the PS the CS number can be either of the two following possibilities: $S(\frac{PS}{2}+0)$ or $S(\frac{PS}{2}+\frac{2^{K-1}}{2})$ since the new state is generated by shifting an input bit into the encoder's shift register. Hence, if the previous best state is known at any point of time, the current best state can be computed by comparing the metrics of the two possible child states. But since the decision taken depends on the metrics of which values are affected by the channel conditions, a manipulated input can result in making the wrong decision and choosing the wrong path. Although this path should merge with the correct one later in the decode process, there is no guarantee that the TW will be wide enough for the merge. There is a need, therefore, to revise our previous decisions in the following cycles to make sure that we have chosen the correct path. We need to look back in time and choose among not only the (PS) child states, but moreover, the child states of the (PS)'s sister state that was neglected. Or even more, we might need to look a few cycles back in time and check all the possible resulting descendants of an original parent state. Let LBL be the look back levels, the CS can be chosen from a set of candidate states as follows:

$$CS = best\{S(\frac{PS}{2^{LBL}} + i * 2^{K-LBL-1})\}, i = 0..2^{LBL} - 1$$

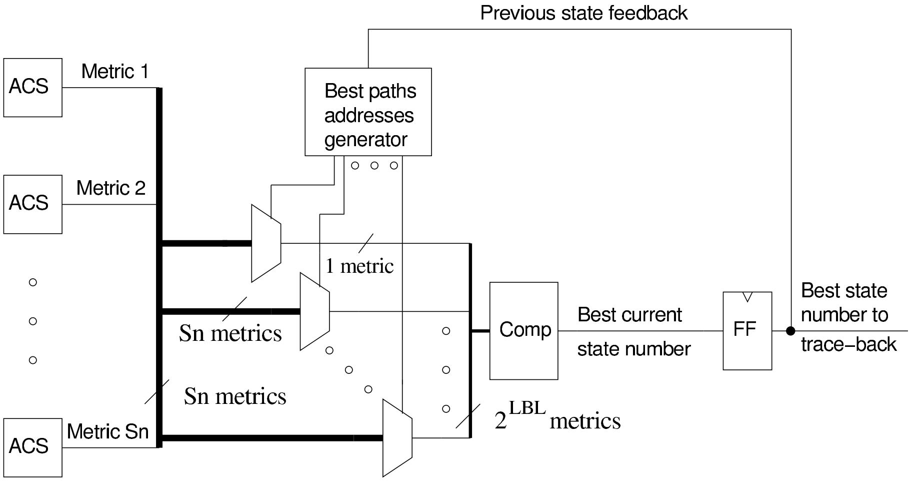

Figure 3.31: The proposed best path selection architecture.

- *Synthesis Results*

  To validate our approach, TB Viterbi decoders with LBL of 1 to 6 and RE Viterbi decoders were modeled in VHDL at the RTL level and simulated. All Viterbi decoders were of $K = 7, S_n = 2^{K-1} = 64$ states and a metric size of 16. Simulations were based on an accurate channel model and punctured coding. Synthesis was carried out using Synopsys Design Analyzer ^{TM} using $0.25\mu m$ technology libraries. Figures 3.33 and 3.34 depict a significant improvement in the decode quality of the 3 LBL TB decoder over that of the proposed RE decoder with a majority counter. Note that this proposed majority counter approach offers a decode quality roughly 30 window lengths better than the classical RE approach as depicted in 3.32. The RE decoder, on the other hand, is much faster and smaller in area (see Table 3.1). While a window length of 70 was used for the synthesis reports shown in Table 3.1, a window length of 40 was used in the simulations comparing between the decode quality of TB decoders of different LBLs. Figure 3.35 shows that the decode quality of the 3 LBL TB decoder is almost the same as the 6 LBL TB decoder. Note that the performance of the 6 LBL TB decoder is theoretically the best for a given WL since the descendants states of a parent state 6 generations back are all the 64 states. Note also that there is a significant difference in area and speed between the 3 LBL TB decoder and the larger LBL ones with virtually no improvement in the BER. The $3LBL$ TB decoder is thus an acceptable compromise.

Table 3.1: Timing and Speed reports for the RE and the 1, 3, 6 LBL decoders synthesized using a $0.25\mu m$ technology.

Block name	RE	1 LBL	3 LBL	6 LBL
Area (mm^2)				
ACS top	2.38	2.56	2.56	2.56
Best State	===	0.21	0.23	0.35
TW	1.28	3.04	3.08	3.23
Top	3.69	5.62	5.66	5.81
Timing (ns)				
ACS top	3.4	3.61	3.61	3.61
Path selection	===	3.96	10.54	84.22
TW	4.12	4.73	12.29	84.35
Top	4.4	4.75	12.29	86.41
Speed (MHz)	227.3	210.5	81.4	11.6

A very important drawback of our Multi LBL approach is that the best state selection becomes a determining factor in the speed of the decoder, since this unit cannot be pipelined because of the feed-back of the PS needed to compute the CS. Nevertheless, high throughput in excess of 80 Mbps can be achievable as shown in Table 3.1. If high speed is the goal, the RE with a majority counter approach can be used.

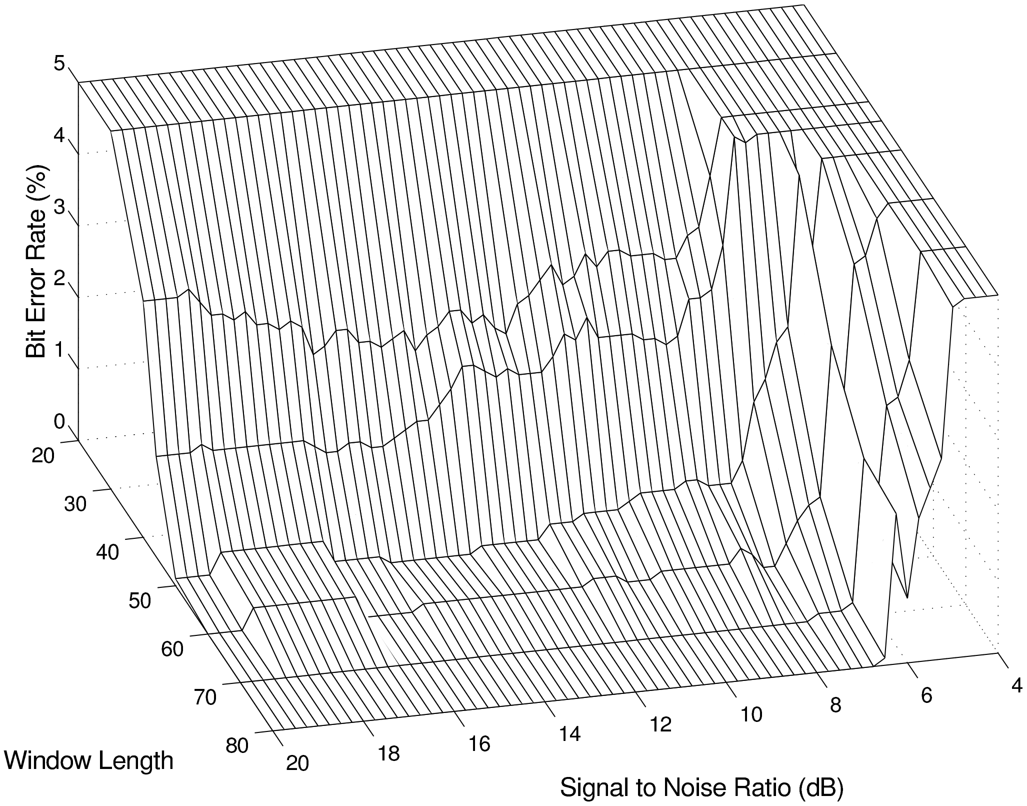

Figure 3.32: RE decoder without a majority counter

3.4.4 Viterbi decoding using the R-2SDF architecture

In a Viterbi decoder the Add-Compare-Select (*ACS*) units' output metrics are fed back to the inputs to reflect the state transition represented by the trellis diagram of the convolutional encoder. As such, the ACS unit produces decision bits and metrics representing the probability for a given state to be visited by the convolutional encoder. For the sake of clarity, we illustrate our architecture for a simple case of a convolutional encoder of a small constraint number.
Consider for example a convolutional encoder of a constraint number, $K = 4$. Such an encoder has a number of flip-flops of $C = K - 1 = 3$ and number of states $S_n = 2^C = 8$ states. The trellis diagram of a Viterbi decoder for this encoder is shown in Figure 3.27.

By reordering the right hand side states forming horizontal connections between the states on each side of the trellis diagram and continuing this reordering process for a number of stages we observe the recovery of the original order of states after C stages. This reordering of states is known as in-place state replacement [9]. The trellis diagram after in-place replacement shown in Figure 3.36 is very similar to the FFT butterfly data flow graph. This interesting analogy unveils opportunities of exploitation of well studied FFT architectures and fitting them for efficient realizations of Viterbi decoders.

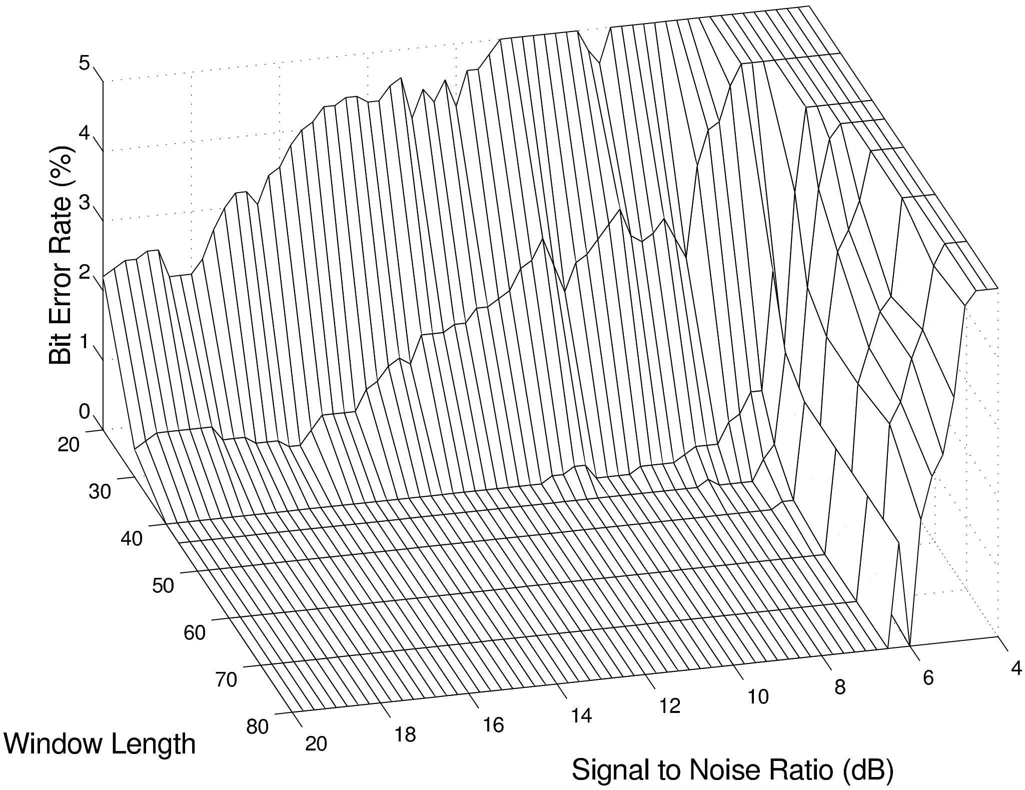

Figure 3.33: The proposed RE decoder with a majority counter

By adjusting the architecture of the R-2SDF FFT implementation, Jia et. al formulated an interesting realization of the Viterbi decoder's ACS unit [49].

While the FFT operation has a fixed number of butterfly stages in the data flow graph, the ACS operation, however, repeats indefinitely and the computed values are reused. The current computed metrics are stored to calculate the next metrics. Observe also that every dynamic butterfly unit computes metrics for one stage of the trellis diagram. Thus the C dynamic butterfly units shown in Figure 3.37 cover C sets of S_n inputs to the decoder. After C stages of butterflies the resulting metrics original order is recovered as shown in Figure 3.36 and fed back thereafter to the first stage. Unfortunately, because of this feedback, the aforementioned architecture cannot be pipelined since pipelining will delay the feedback data which is strongly correlated with the streaming input to the first stage, and that will result in wrong decoding. Pipelining and slowing down the input stream to wait for the appropriate feedback data defeats the main merit of pipelining which is speeding up. This is the sole reason that makes the ACS unit the speed limiting factor of any Viterbi decoder implementation. Nevertheless, the overall throughput is still in the expected range since all C butterflies stages produce metrics simultaneously compensating for the C stags of ACS delays.
In Section 3.5 we propose a method that enhances the throughput and allows for trellis decoding realization using the same architecture and clock frequency as well.

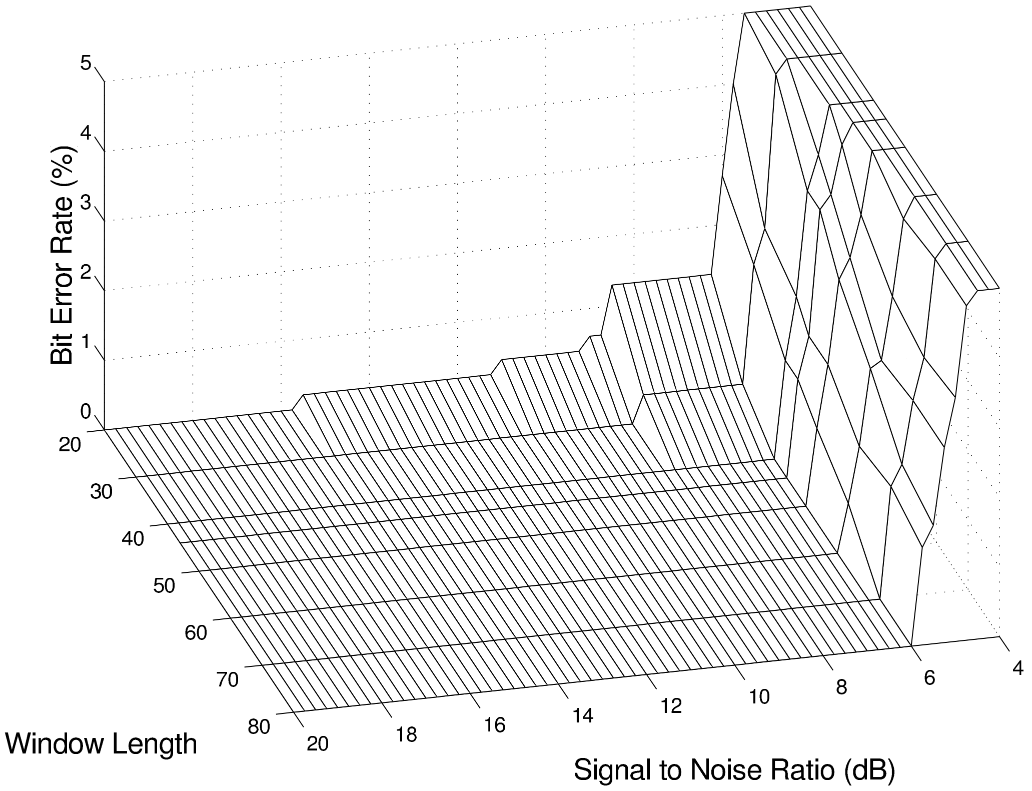

Figure 3.34: The proposed 3 LBL TB decoder

3.5 Proposed Specialized Reconfigurable Architectures

In the coarse of our study of several DSP VLSI architectures, we have developed pipelined VLSI architectures for some of the aforementioned DSP algorithms. Our intention was to find simply controlled modular architectures composed of a repetitive structure of processing elements. Such architectures are most beneficial in understanding the requirements of a general purpose reconfigurable architecture for DSP applications. In this Section some of the our results and contribution is this area are presented.

3.5.1 The Proposed Reconfigurable Size MDCT Processor

Some applications such as the Ogg Vorbis uses a Modified DCT (*MDCT*) which is a variant of the type 4 DCT. The transformation formulas are as follows: MDCT:

$$X_t(m) = \sum_{k=0}^{n-1} f(k)x_t(k)\cos\big(\frac{\pi}{2n}(2k+1+\frac{n}{2})(2m+1)\big) \qquad (3.20)$$

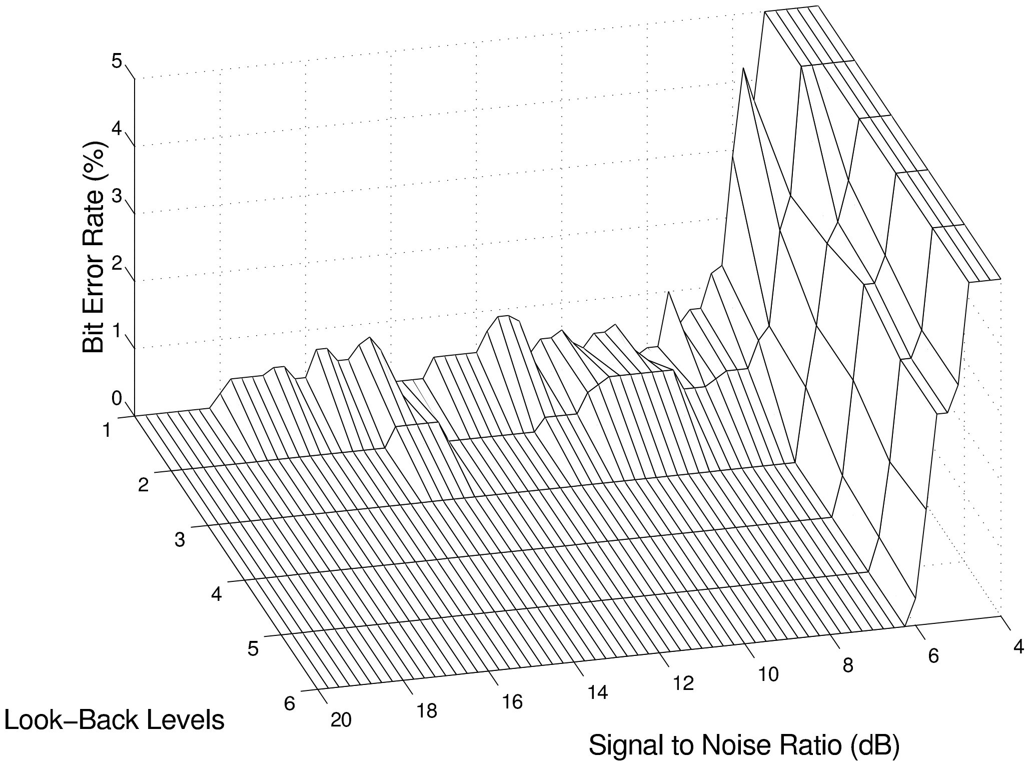

Figure 3.35: The proposed TB decoders with different LBLs.

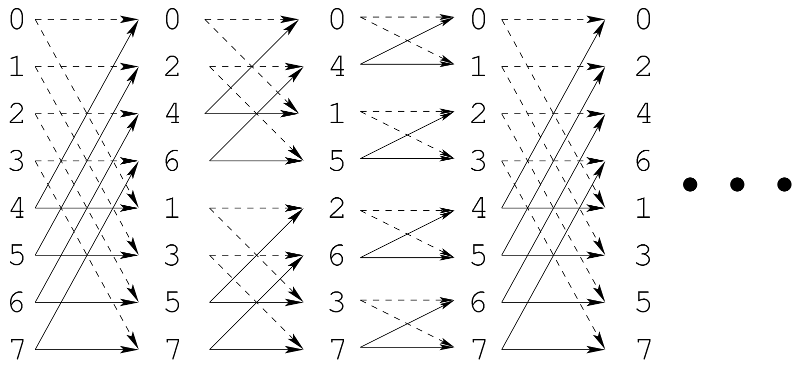

Figure 3.36: Trellis diagram of a $K = 4$ Viterbi decoder with in-place replacement

for $m = 0..\frac{n}{2} - 1$

IMDCT:

$$y_t(p) = f(p)\frac{n}{4}\sum_{m=0}^{\frac{n}{2}-1} X_t(m)\cos\big(\frac{\pi}{2n}(2p+1+\frac{n}{2})(2m+1)\big) \tag{3.21}$$

for $p = 0..n - 1$

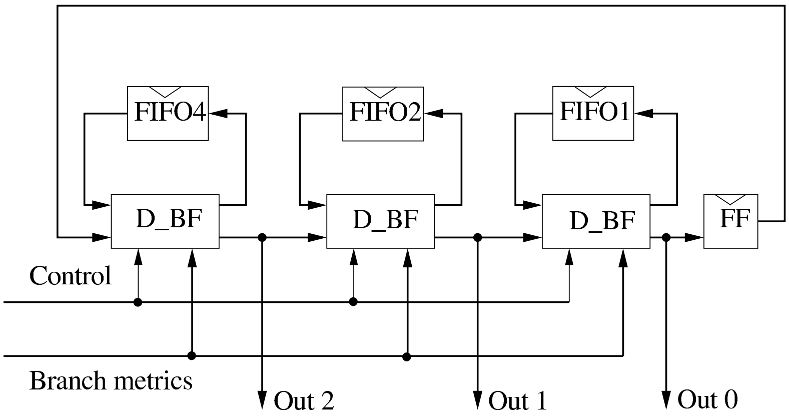

Figure 3.37: The R2SDF ACS Architecture of a $K = 4$ Viterbi decoder

The MDCT SFG is shown in Figure 3.38. An example of one of the irregular blocks is shown in Figure 3.40. Following, for each column of the SFG in Figure 3.38 we used a corresponding block dealing with the data and passing them to the appropriate data path at the correct instance.

As was discussed above, the basic idea behind pipelined structure is the correct scheduling of the input data arriving serially vector after another. This is done effectively with the dynamic butterflies. Figure 3.39 shows the dynamic butterfly architecture used for the DCT computation.

Not all irregularities are similar. Figure 3.42 for example shows the irregularity 16 SFG. We needed therefore to design different irregular block processor for each irregularity stage. However, we have found some similar features in some of the irregularities SFGs namely in irr_16, irr_32 and irr_64. With some minor differences in the signs of the inputs we have categorized them as XN, XP, PX XOP, NX and WL as shown in the Figures 3.42 and 3.38. Our solution was to design a separate data path for each and then select the correct data path for the data according to the time position of the streaming in data. The select line addresses is supplied by the controller through a simple counter. Figure 3.41 shows for example the irregularity of the irr_8 block.

Ogg Vorbis needs two sizes of MDCT 1024 points and 128 points. We have designed a reconfigurable data path that supports both sizes by feeding the input through either the complete data path or to activate only the 128 point data path (see Figure 3.43). With such an architecture more MDCT sizes can be supported by allowing the input to be fed at the input of each butterfly and irregular block pair.

The MDCT processor shown in Figure 3.43 was simulated and its correctness was proven by comparing the output files to that of the C code.

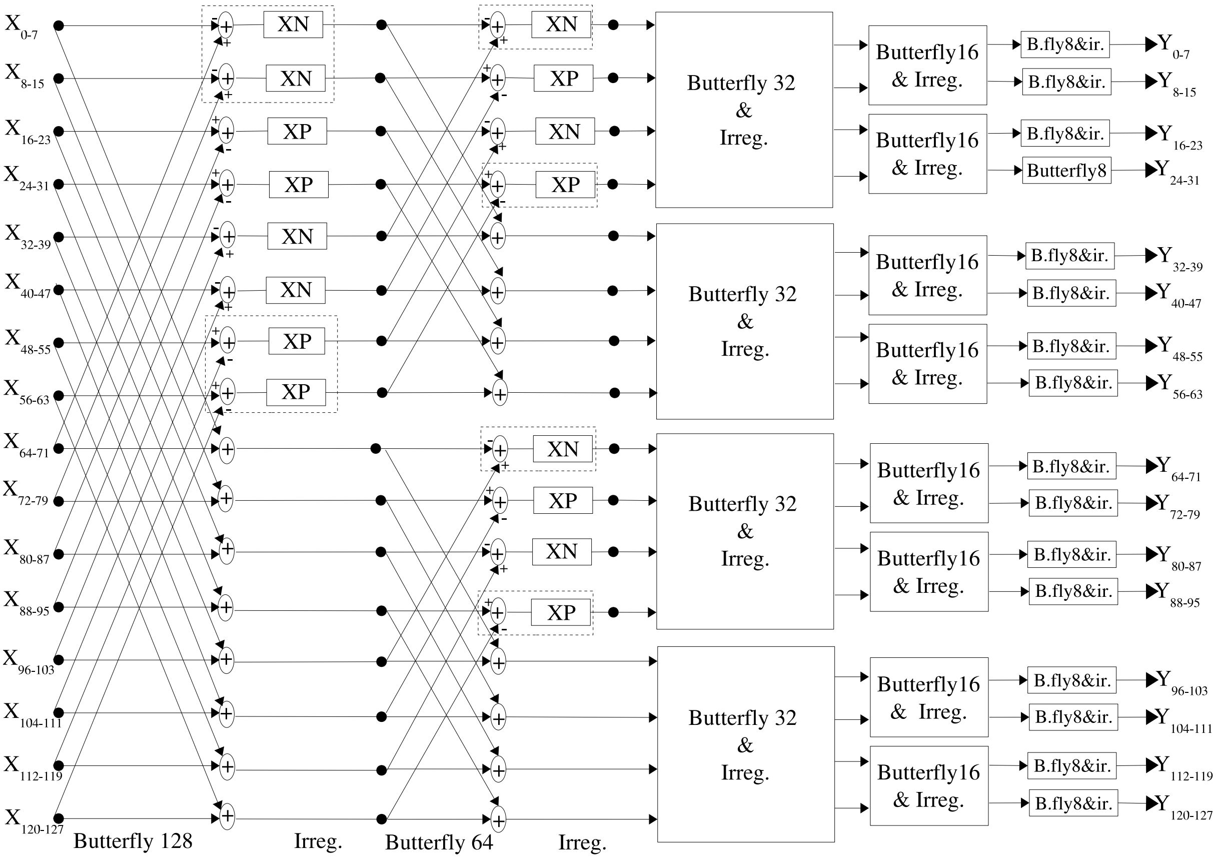

Figure 3.38: The SFG of the 128 point DCT

3.5.1.1 Synthesis Results

The reconfigurable size accelerator was modeled in RTL VHDL simulated and synthesized on a a XILINX Virtex2 devise (using an XC2V8000 board) using the XILINX ISE tools. Special coding techniques were used to utilize the multipliers and internal RAM to implement the FIFOs and the twiddle factors memory. Table 3.4 summarizes the synthesis results.

Table 3.2: Synthesis Results

Component Names	# of Components	% of Usage
Slices	5131	11%
Multipliers	112	66%
BSRAMs	9	5%
Flip Flops	5819	6%

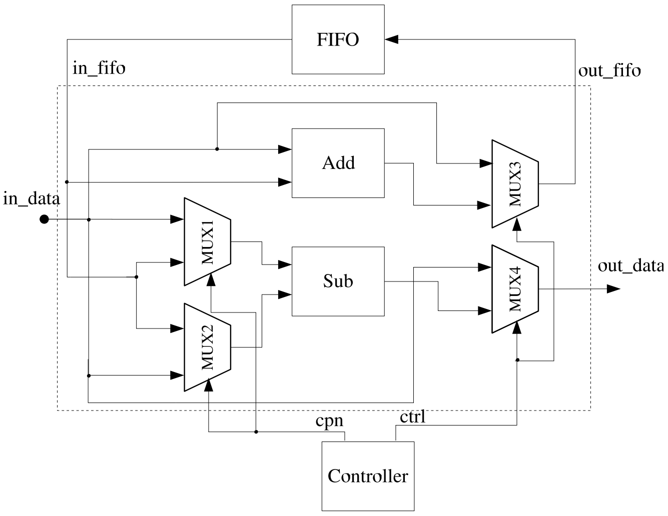

Figure 3.39: The Dynamic butterfly architecture

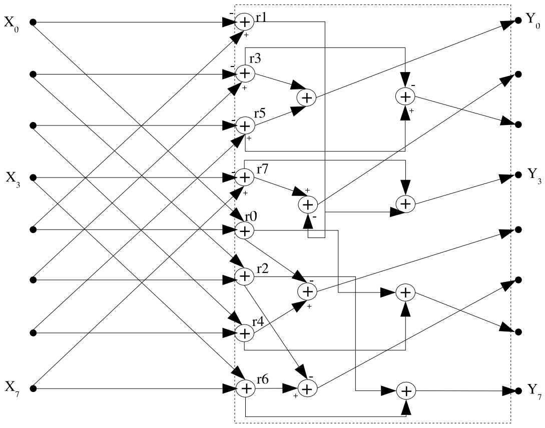

Figure 3.40: The irregularity 8 SFG

3.5.2 A Reconfigurable FIR Filter Realization

Most of the FIR filter structures reviewed above have similar hardware requirements and regular structures. The direct form and the transposed form are most interesting because of their simplicity and regularity.

To explore the requirements of building our reconfigurable architecture that can efficiently realize FIR filters, we experimented with a reconfigurable architecture for FIR filters. We chose to take the transposed form structure since it offers hardware savings when it comes to implementing a fully pipelined FIR filter as mentioned above. A fully pipelined transposed form FIR filter structure is shown in Figure 3.44. From Figures 3.44 and 3.6 we see that a reconfigurable transposed form filter structure capable of realizing general or linear-phase FIR filters should have the following features:

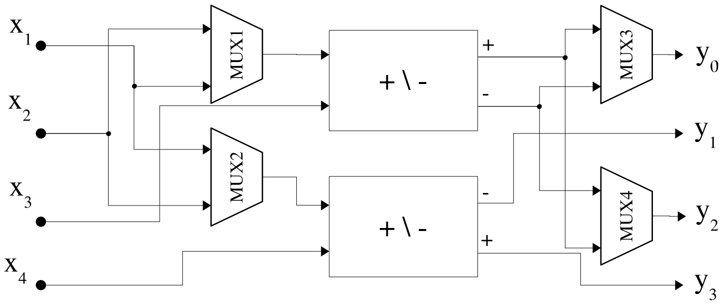

Figure 3.41: The irregularity 8 block

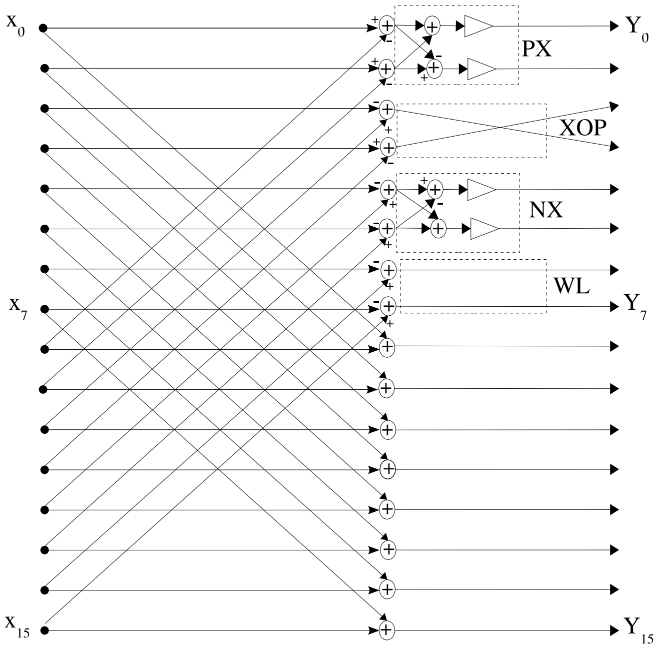

Figure 3.42: The irregularity 16 SFG

- A basic processing element representing a *tap*.
- Each tap should consist of a multiplier, an adder, a delay element to store partial products, a register for pipelining after the multiplier and a register to store the filter coefficient.
- Each tap should be architected such that it can be cascaded with a similar tap to form bigger or smaller filters.
- Furthermore, for a general purpose reconfigurable FIR filter, there should be an efficient way to pass and store coefficients.

Now to design a complete reconfigurable FIR filter taking the above points into consideration, we choose a modular architecture that can also be cascaded with similar

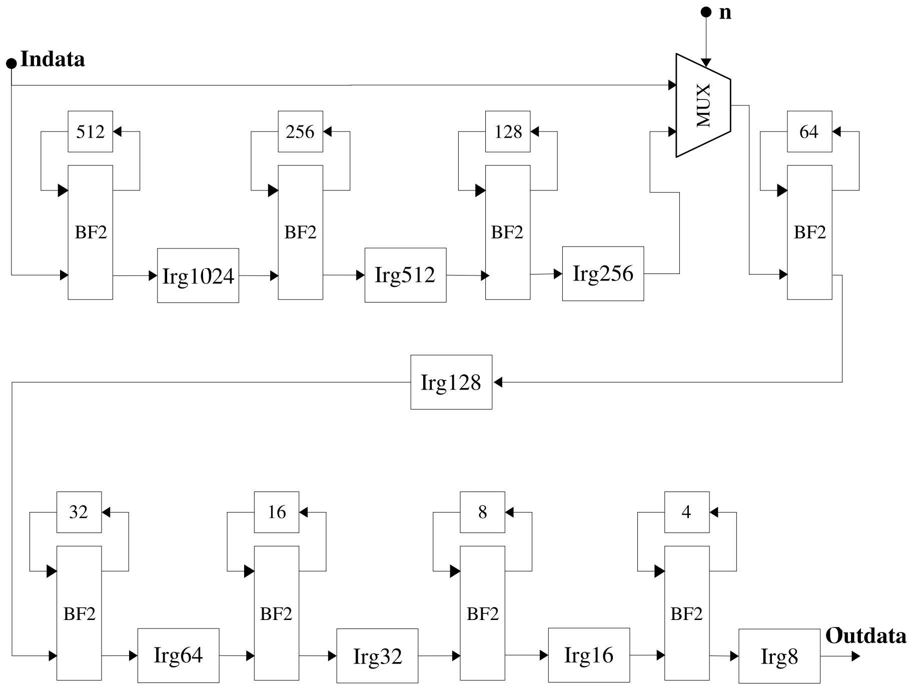

Figure 3.43: The block diagram of the reconfigurable IMDCT accelerator

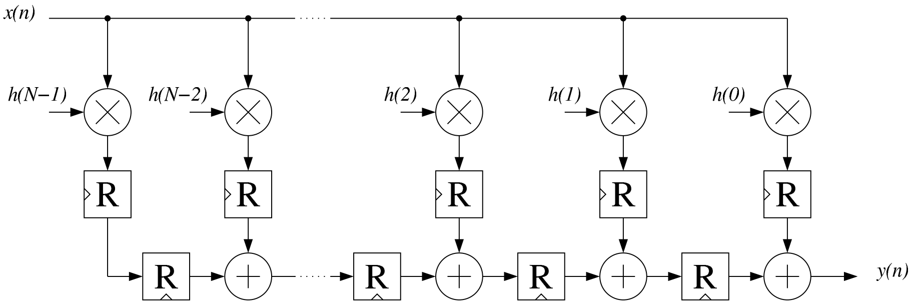

Figure 3.44: A fully pipelined transposed form FIR filter structure.

elements forming larger and smaller filters. Moreover, odd, even and various FIR filter sizes are required to be realizable. All of the above requirements were taken into account and a reconfigurable size FIR filter shown in Figure 3.45 was realized. The organization of the reconfigurable FIR filter data path unit shown in Figure 3.45 illustrates a two row structure where the input and output of the filter are at one end of the filter. Bigger sized filters to be realized. The data path shown above can realize a maximum tap number of 6. As such, realize 6-tap, 12-tap, 18-tap, etc. sized FIR filters can be realized by cascading several of the units shown in Figure 3.45. Finer tap granularity can be realized as well by the use of the multiplexors located in the bottom half of the Figure above. To store the filter coefficients new coefficients are shifted through the registers located at the input of the multipliers as depicted in Figure 3.45.

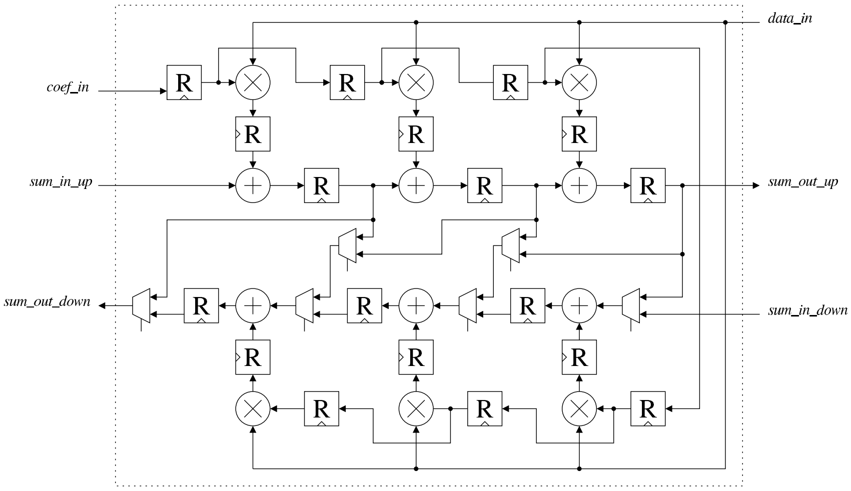

Figure 3.45: The reconfigurable TDF FIR filter data path.

3.5.2.1 Reconfigurable Single-Cycle FIR Filter Processor

Several of the data path units shown in Figure 3.45 can be cascaded to in a large data path coinciding with the biggest intended FIR filter implementation form the main processing unit shown in Figure 3.46.

This data path can realize also smaller FIR filters when reconfigured by the control unit. The control unit manages also the initialization of the data path unit and storage of the filter coefficients as well as the operation duration of the reconfigurable FIR filter. As such the architecture of the system illustrated in Figure 3.46 can perform computations for various sizes FIR filters limited only by the size of the data path unit.

3.5.2.2 Reconfigurable Multi-Cycle FIR Filter Processor

The modular reconfigurable design explained above can be used to synthesize reconfigurable architectures of different sizes. But when synthesized, the above reconfigurable architecture can be used to realize filters with a tap number less than or equal to that of the synthesized reconfigurable FIR filter. To realize bigger filters partial results should be stored in an intermediate memory and operated on again using the same hardware to produce the final result. Such a system should include intermediate memory and a more complicated controller along side the FIR data path unit as shown in Figure 3.47.

Here, the controller determines the number of times it needs to use the given data path to conclude the complete FIR filter computations and manages storage of partial data to be used again in remaining parts of th FIR filter computation. The state diagram of the controller is shown in Figure 3.48. As illustrated in the Figure, the controller has 4 states: IDLE, INIT, OUTPUT, OUTPUT_STORE and COMP_STORE. If the size of the

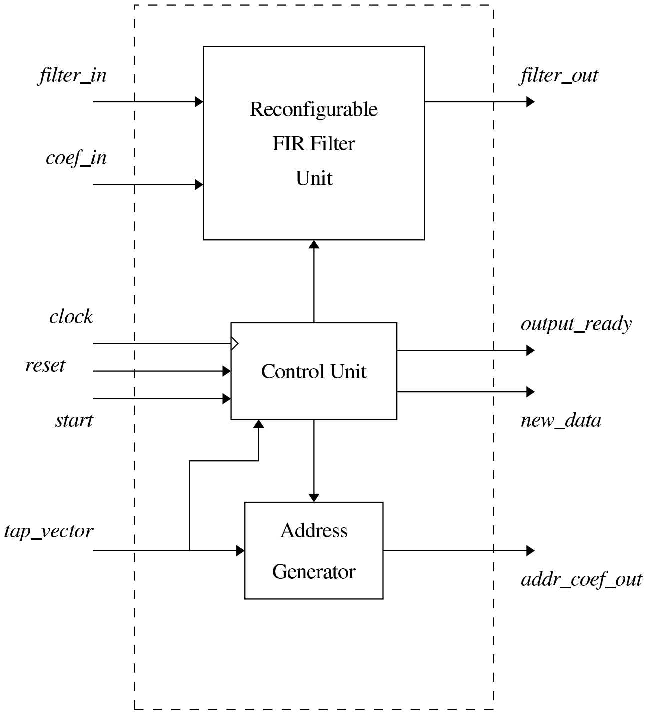

Figure 3.46: Block diagram of the Single-Cycle reconfigurable FIR filter.

required filter is less than or equal to the data path unit available the data path unit is initialized with the proper coefficients in the INIT state and the output is completed and produced directly in the OUTPUT state. If the required FIR filter is larger than the available data path the coefficients corresponding to the first cycle of computation are stored in the data path unit in the INIT state and then partial computations are carried out and partial results are stored in the intermediate memory in the COMP_STORE state. When all cycles of computations are done the final results are made ready in the OUTPUT_STORE state.

- *Synthesis Results*
  The question of the more feasible realization of the reconfigurable FIR filter being the single cycle or the more complicated single cycle approach depends more or less on the application requirements. While the single cycle approach is simpler and require less control and memory resources, generally larger reconfigurable data path unit to support larger FIR filters. On the other hand, the multi cycle approach require more area for intermediate memories and more complicated control resulting in area and speed penalties. Table 3.3 summarizes the synthesis results of a 64-tap single-cycle reconfigurable FIR filter and a 32-tap multi-cycle reconfigurable FIR filter. Table 3.3 also shows synthesis results of a normal non-reconfigurable pipelined transposed form 64-tap FIR filter to give an idea about overhead resulting from the reconfiguration resources invested.

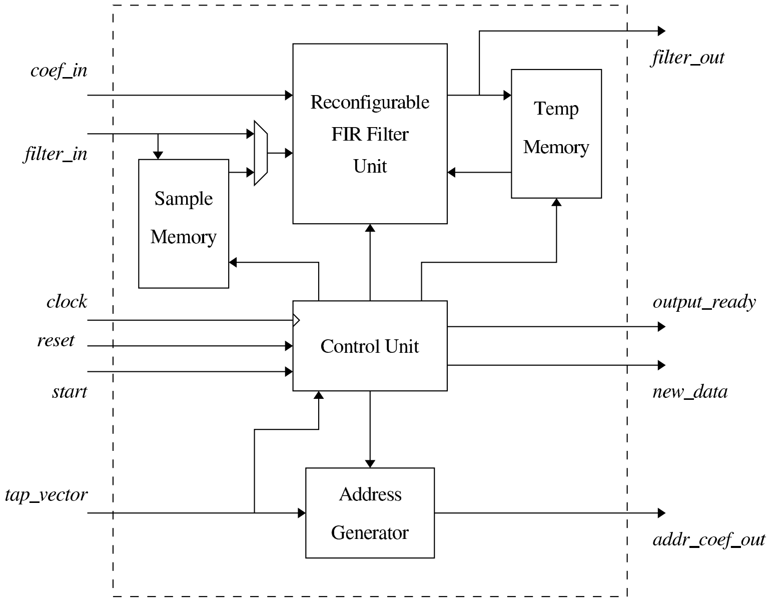

Figure 3.47: Block diagram of the Multi-Cycle reconfigurable FIR filter.

FIR Architecture	Area (μm^2)	Frequency (MHz)
non-reconfigurable pipelined (64-taps)	3776560	171.8
Single-Cycle(64-taps)	3899282	120.6
Multi-Cycle (32-taps and 1k of memory)	7699733	75.6

Table 3.3: Synthesis results of various implementations of the FIR filter (using UMC 0.25 μm libraries)

3.5.3 The Proposed R2MDF Architecture

The main problem in architectures such as that shown in Figure 3.37 is throughput. In [107] the proposed State Parallel (SP) Viterbi decoder was intended for the HiperLAN 2 standard. The achieved performance is over 200 Mbps, while the required speed is 56 Mbps. Surely, this performance is gained at the cost of investing in hardware having S_n ACS units. Building a sequential Viterbi decoder with only one butterfly element to save in silicon area will not achieve the targeted minimum throughput. The R-2SDF architecture proposed in [49] is also not fast enough. Therefore, it will be beneficial to explore the design space between the SP approach and the R-2SDF approach to reach a compromise between area and speed. To overcome the throughput problem we propose to split the butterflies vertically thus doubling the number of inputs and outputs.

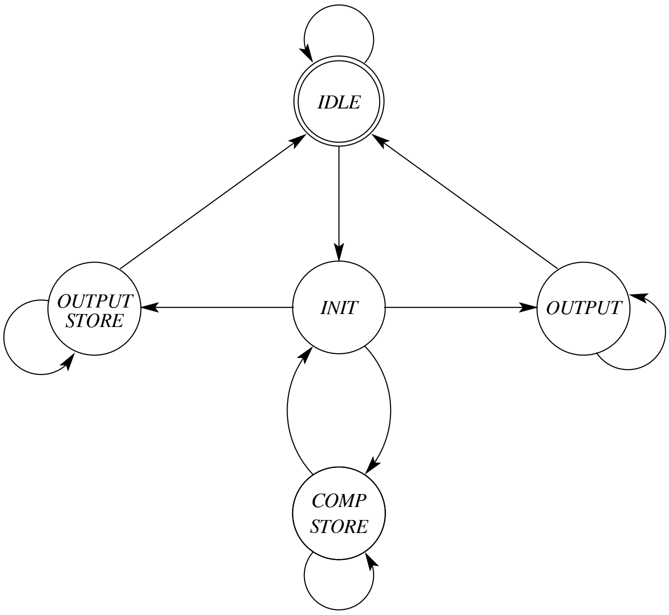

Figure 3.48: The Controller of the Multi-Cycle reconfigurable FIR filter of Figure 3.47

3.5.3.1 Parallelizing the butterflies

By splitting the butterflies, every dynamic butterfly unit is replaced by two in parallel. This way the number of inputs of the R-2SDF ACS unit is doubled, the work at every butterfly stage is divided by two and consequently, the number of outputs is also doubled. Since the computation at every stage is split between two units, odd and even, data are fed in parallel. Thus, the FIFOs sizes are halved. The stage that had a FIFO of size 1 is replaced with an always active static butterfly unit binding the two split halves of the data flow graph and producing 2 outputs in parallel. As depicted in Figure 3.49 the number of outputs is now 6 rather than 3 meaning that the effective throughput of the R-2SDF unit is now 6 outputs per cycle rather than 3, naturally at the expense of two more butterfly units but reduced FIFO sizes.

Taking this parallelization technique one step further we arrive at the configuration of Figure 3.51. Here, with the smaller FIFO sizes and a total of 8 butterfly units we achieve a throughput of 12 outputs per cycle which is more than that of the normal SP implementations, again at the expense of additional hardware.
Although the butterfly count is doubled with each parallelization level the sizes of the FIFOs are decreased. Moreover, the utilization percentage of the butterflies increases as well. We denote this architecture the **R**adix-**2** **M**ulti-path **D**elay **F**eedback (R-2MDF) architecture.

The area requirement issue is an important one to address for two reasons: Firstly, the knowledge of the amount of hardware prior to synthesis and fabrication is important since this helps to decide on levels of parallelism feasible to be invested in especially in

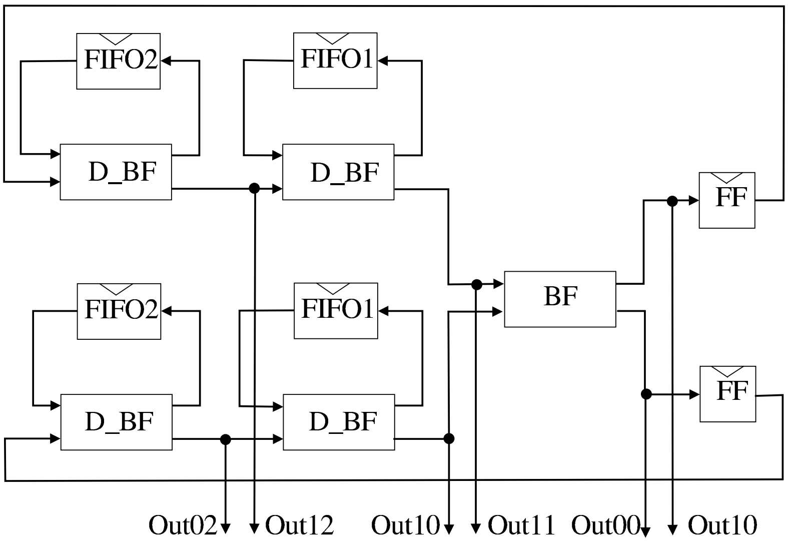

Figure 3.49: The proposed R-2MDF ACS unit for a $K = 4$ Viterbi decoder parallelized once ($l = 2$)

such a design paradigm that grows exponentially with parallelization levels and constraint number. Secondly, when designing a reconfigurable hardware aimed at solving such problems, it is good to have an idea about the amount of hardware that should be thrown in to fit synthesis of the targeted problems.

Figure 3.50 shows different arrangements of the butterfly units for possible parallelization levels of a $K = 5$ ACS unit using the proposed technique. The dots represent butterfly units and for simplicity no connectivity or FIFOs are shown.

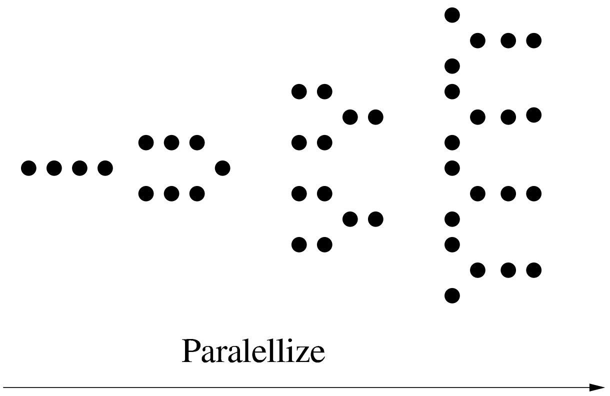

Figure 3.50: Possible levels of parallelization for a $K = 5$ Viterbi decoder

3.5.3.2 Quantitative analysis

In general, the number of inputs to the proposed super parallelized R-2MDF ACS unit is $InNo = 2^l$ and the number of outputs generated is $OpNo = C * 2^l$ where $l = 1, 2, ...K$ is the number of parallel inputs to the R-2MDF unit.

Proposition 1 *The number of butterflies of an R-2MDF ACS with 2^C points and 2^{l-1} outputs is given by:*

$$BF_{No} = 2^{l-2}(2C - l + 1),$$

and the number of flip-flops is given by:

$$FF_{No} = 2^C - 2^{l-1}.$$

Proof 1 *Looking at the butterfly elements we find that there are C stages of dynamic butterflies. The number of dynamic butterflies stages (columns) is reduced by 1 with every parallelization level but the number of the dynamic butterflies rows is doubled. Hence, the number of dynamic butterfly elements D_BF_{No} as a function of C and l can be expressed as:*

$$D\_BF_{No} = (C - (l - 1))2^{l-1}.$$

For the static always active butterflies, the number of stages increases with l starting with 0 when $l = 1$ and the number of rows is also doubled with l. Therefore, the number of static butterfly elements S_BF_{No} as a function of C and l can be expressed as:

$$S\_BF_{No} = (l - 1)2^{l-2}.$$

The total number of butterflies is then

$$\begin{aligned} &BF_{No} = D\_BF_{No} + S\_BF_{No}. \\ &= (C - (l - 1))2^{l-1} + (l - 1)2^{l-2}. \\ &= 2^{l-2}(2C - l + 1). \end{aligned}$$

The FIFO sizes per row is a power of 2 series starting with 2^0 up to 2^{k-1} where k is the number of stages. Since the flip-flops are attached only to the dynamic butterflies and since the number of rows increases parallelization level a factor of 2, we deduce,

$$\begin{aligned} &FF_{No} = 2^{l-1}.\sum_{m=0}^{C-l} 2^{C-m-l}. \\ &= 2^C - 2^{l-1}. \end{aligned}$$

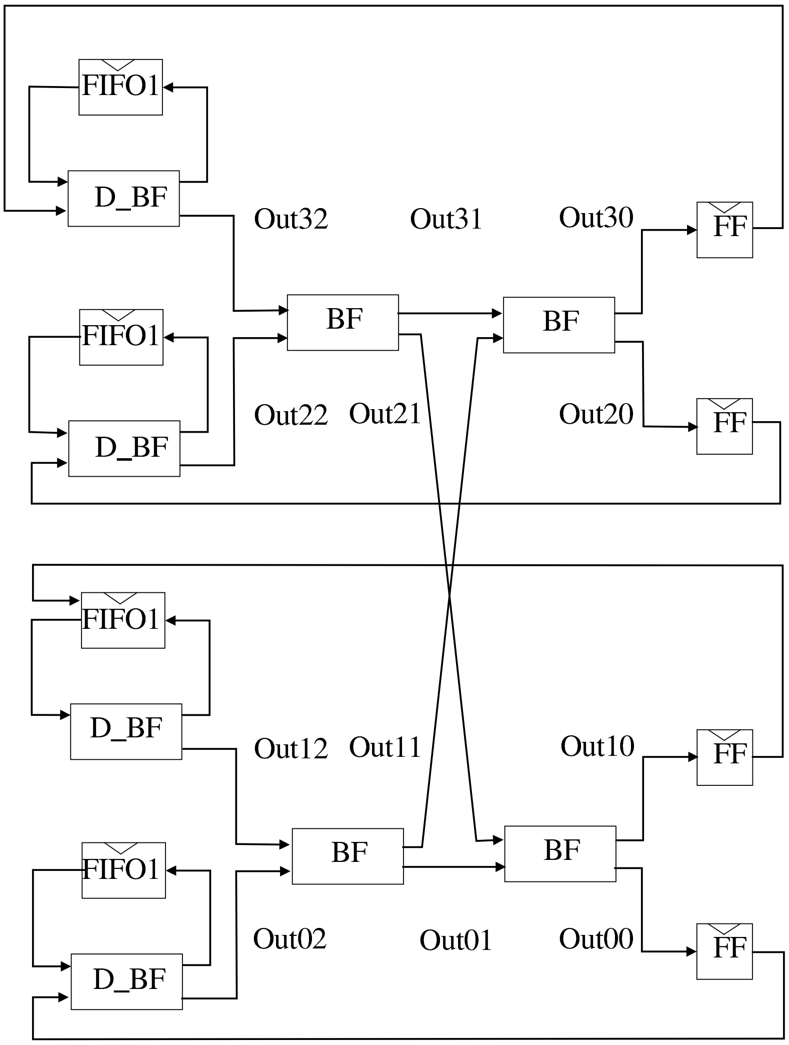

Figure 3.51: The proposed R-2MDF ACS unit for a $K = 4$ Viterbi decoder parallelized twice ($l = 3$)

3.5.3.3 The trellis table realization

It is important to note that the outputs produced by the aforementioned method are scrambled and need to be reordered prior to decoding in the trellis window. This scrambling of data is due to the in-place state replacement methodology used in constructing this ACS unit. Hence, the output of each butterfly should be managed such that the trellis window unit can make use of it.

There are two main ways to decode the output decision bits produced by the ACS unit: the trace-back (*TB*) and the Register Exchange (*RE*) methods. In the TB method by

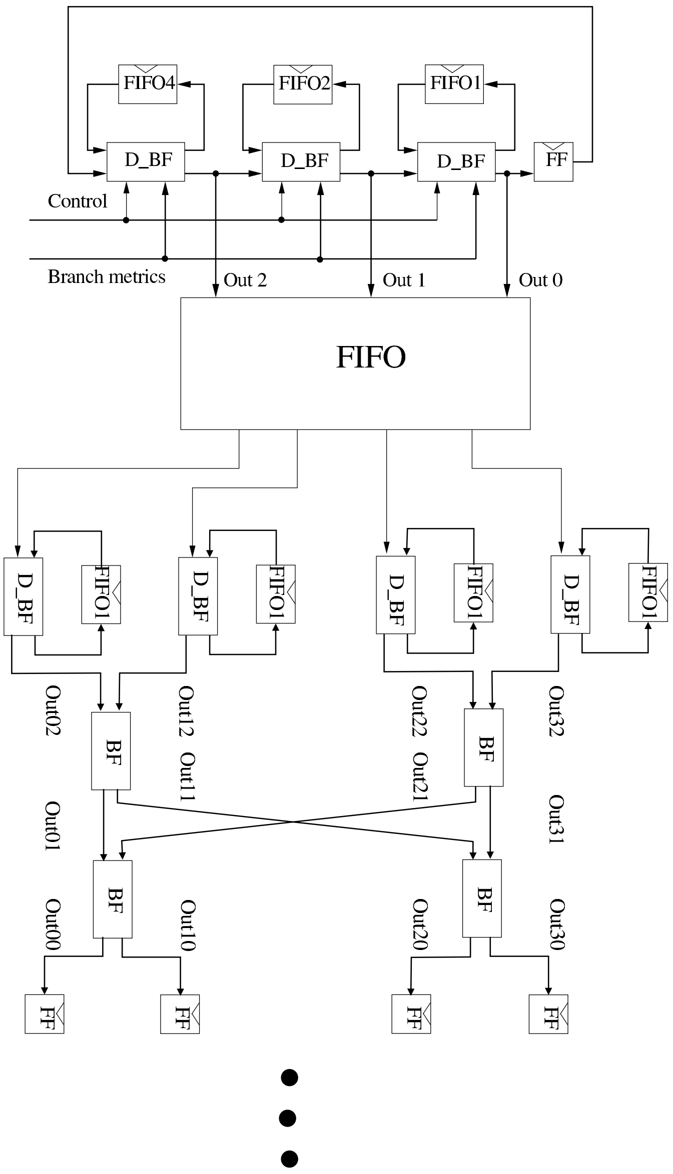

Figure 3.52: The complete R-2MDF based Viterbi decoder architecture

selecting an initial state – which can be chosen according to the metrics – the decision bit corresponding to this state is chosen and a new state is found. This process is continued through the trellis table. On the other hand, in the RE method the trellis table is arranged like the trellis diagram shown in Figure 3.27 with multiplexers and registers at every node and with the select lines of the multiplexers connected to the newest decision vector generated by the ACS unit. Because of this type of connections we propose to use the same R-2MDF architecture for solving the RE problem.

Since the decoding quality is a function of the realized size of the trellis table, a number of R-2MDF RE units can be cascaded to meet the required decoding quality. Longer

tables are generally preferred to achieve better decoding quality. Clearly, using too long trellis tables is not the best solution due to chip area and power consumption requirements. A good approach is to have a dynamic length trellis table as proposed in [110]. Our proposed architecture here facilitates dynamic enabling or disabling of some or parts of the cascaded R-2SDF RE window according to channel conditions.
The trellis table decode throughput should match the ACS computation speed. Thus, the parallelization level of the register-exchange trellis must be such that the number of inputs of the register-exchange trellis is grater or equal to that of the ACS unit. In some cases the number of inputs and the number of outputs can be matched by using different parallelization levels. An example of that is a $K = 5$ decoder with the ACS unit parallelized 2 times and the trellis window unit parallelized 4 times. From the equations given above, the ACS unit will have 8 outputs and the trellis window 8 inputs. In case an equal number of inputs is not possible, the next greater number is chosen and as shown in Figure 3.52, a FIFO could be used to manage the flow of data.

3.5.3.4 Synthesis results

We have developed a parameterizable VHDL IP at the RTL level for the abovementioned architecture. The idea is by simply changing some constants in the package, different Viterbi decoders with completely different specifications can be synthesized. Furthermore, various levels of parallelization can be selected by editing the corresponding constant in the package. This approach facilitated synthesizing more designs with no need for re-coding.

3.5.3.5 Performance analysis

Note that a straightforward comparison of different architectures results cannot be made. This is because these architectures do not have the same processing power. The fastest implementation is the SP one with a throughput of $S_n = 2^{K-1}$ outputs/cycle. The throughput of the R-2SDF architecture is $C = K - 1$ outputs/cycle. Because of the feedback from the last stage to the first stage pipelining is not possible and therefore, the clock cycle will be C times slower than that of the SP which means that the R-2SDF speed normalized to that of the SP one will be: $\frac{1}{S_n}$ outputs/SP cycle. Each parallelization level in the R-2MDS architecture doubles the throughput while maintaining the slow R-2SDF cycle. Thus, the normalized R-2MDF architecture's speed is $\frac{1}{S_n}.2^{l-1}$ outputs/SP cycle. Since l ranges between 1 and K, the maximum achievable speed will be that of the SP implementation. It can be shown that at a certain level of parallelization the amount of hardware invested will be more than that of the SP but unfortunately must operate at slower speed. The above formulas are therefore useful in determining the points at which the R-2MDF does not provide an optimum solution given the constraint number and parallelization levels.

Table 3.4: Timing and area reports for different realizations of a $K = 7$ Viterbi decoder's ACS unit. synthesized using a $0.25\mu m$ technology.

Architecture	Cycle (ns)	Area (mm^2)	Butterflies number	Throughput metrics/cycle	Norm. throughput	Norm. speed
R-2SDF ($l = 1$)	21.12	0.72	6	6	0.966	0.015
R-2MDF ($l = 3$)	20.07	1.89	20	24	4.065	0.064
R-2MDF ($l = 5$)	21.30	5.53	64	96	15.324	0.239
SP	3.4	2.38	32	64	64	1.0

It is worth mentioning that the above analysis holds only for the ACS unit implementation because of the feedback while for other applications such as FFT and DCT any level of parallelization will be rather an adequate solution if higher throughput is needed. Furthermore, because of the decrease in the sizes of the FIFOs and the number of dynamic butterflies, each parallelization level will result in yet a more efficient architecture in terms of hardware utilization.

Table 3.4 shows timing and area results for different parallelization levels of a $K = 7$ Viterbi decoder. Synthesis was carried out using Synopsys Design Analyzer ^{TM} and $0.25\mu m$ technology libraries. The last row shows the results obtained from a systolic SP Viterbi decoder reported in [106, 107] while the previous ones are for different levels of parallelization. The last two columns give a comparison between throughputs and speeds of different levels of parallelization of the proposed R-2MDF architecture all normalized to those of the SP realization. Note that for 4 levels of parallelization ($l = 5$) the amount of hardware needed exceeds the SP one but still at 23.9% of its performance. Clearly, this is a case where the R-2MDF is a suboptimal solution. Working with the formulas above, we find that the R-2MDF with 2 levels of parallelization ($l = 3$) is the maximum parallelization choice with less hardware than the SP. The speed, although 6.4% of the SP one, is still much faster than any sequential realization.

Although the clock cycle results for all the R-2MDF are almost the same, the throughput of the 2 levels of parallelization is $2^2 = 4$ times that of no parallelization since the number of output metrics per cycle is quadrupled.

We base our ACS design here on the architecture proposed by [106, 107]. The parallelization technique discussed here facilitates producing more outputs in parallel thus achieving higher speeds.

3.5.4 Reconfigurable R2^2MDF FFT Processors

In case only the size is the problem is to be configured, simply one extra butterfly stage along with a FIFO of the appropriate size can be cascaded at (or remover from) the left and then we have the new FFT as is proposed in [59]. Here because the throughput

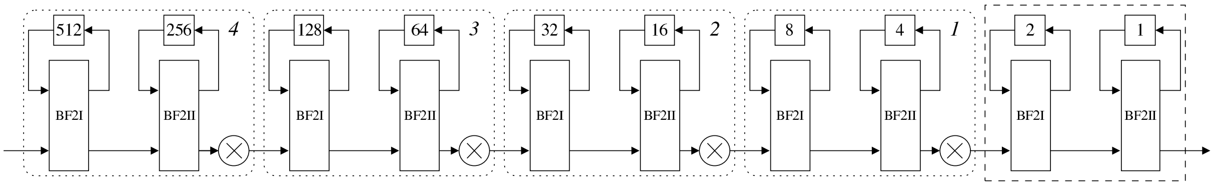

Figure 3.53: 1 output per cycle configuration ($l = 1$)

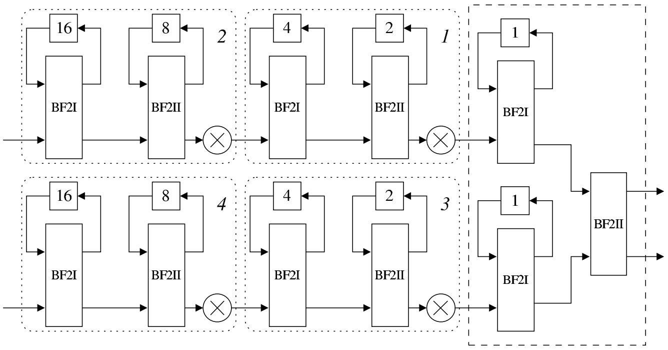

Figure 3.54: 2 outputs per cycle configuration ($l = 2$)

should also be reconfigurable we have to take care of several more points: the variable FIFO sizes, the functionality of the dynamic butterflies, the twiddle factors and the interconnectivity as they change according to the amount of parallelization levels and the FFT size sought.

Given a fixed number of processing elements (butterflies in our case) and because of the regularity of the R2^{2}MDF architecture there are two possible approaches to go around designing a reconfigurable FFT processor. The first approach is to design a reconfigurable FFT processor with enough hardware resources for realizing the largest FFT to be supported. Smaller variants can be realized by the use of a simple reconfiguration controller managing the reconfiguration. The other approach saves on the hardware and processes a part of the data flow graph storing intermediate data and then processing the intermediate data to produce the outputs but at the expense of latency. The following sections introduce these approaches.

3.5.4.1 Single Cycle R2^{2}MDF Architecture

To test this approach we realized a 1024-point R2^{2}MDF FFT processor with a simple controller Figure 3.53 and 3.55 show different reconfiguration possibilities.

Since there are two butterfly types in the R2^{2}SDF which is the base of our proposed R2^{2}MDF architecture our main building block is chosen to be two dynamic butterflies of type I and II with a complex multiplier as illustrated in the figures below.

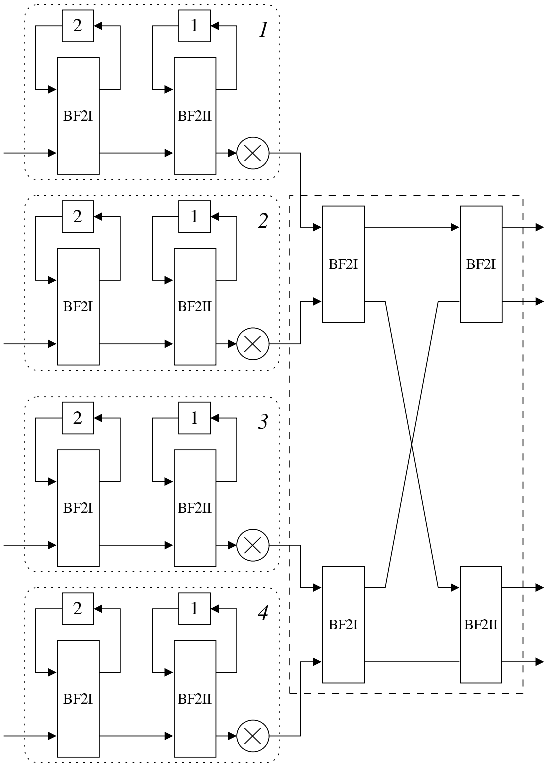

Figure 3.55: 4 outputs per cycle configuration ($l = 3$)

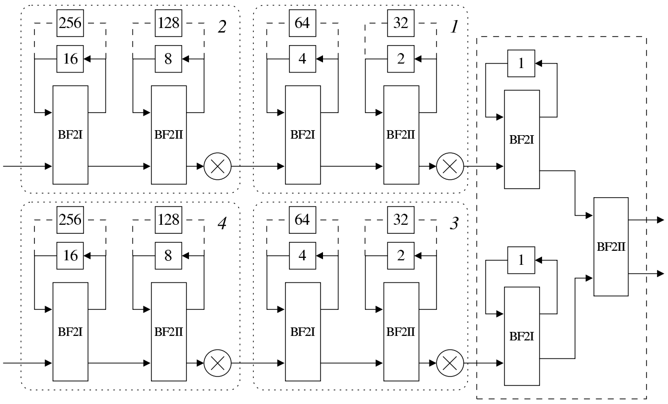

Figure 3.56: The smaller data path/multi-cycle reconfigurable data path

The twiddle factors addressing problem is solved by using a partial addressing technique to get the proper factor depending on the size of the FFT.

3.5.4.2 Multi Cycle R2^{2}MDF Architecture

To investigate this approach, we have implemented the configurable FFT processor depicted in Figure 3.58. Input data can be processed in stages carrying out partial computation of the FFT flow graph and storing partial results in intermediate memory. This process is repeated until completing the FFT flow graph. Theoretically this method can implement any FFT size but the two limiting factors are the latency per computed output and the size of the memory needed (FIFOs and intermediate) required for the computations. The operation controller Figure 3.57 orchestrates this operation of this realization facilitating 8 to 1024-point FFT was implementations. In this paradigm extra ROMs and FIFOs have to be added.

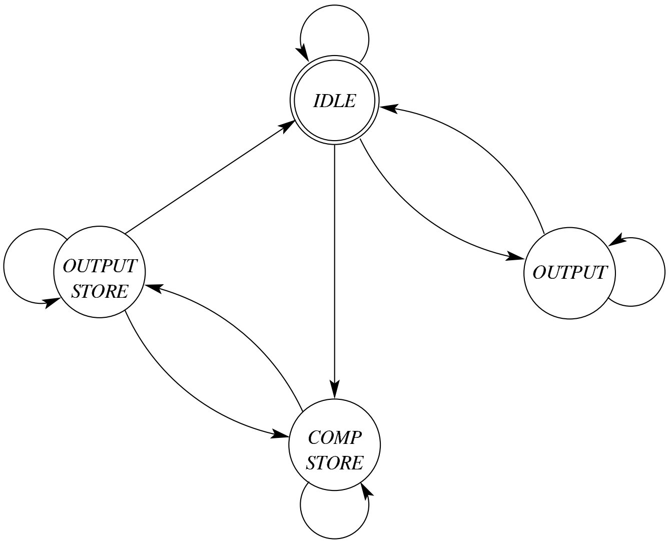

Figure 3.57: Small Data path reconfigurable FFT operation controller

3.5.4.3 Synthesis Results

VHDL models at the RTL level for both paradigms were developed and also a model for a conventional non-reconfigurable FFT processor as well as a non-reconfigurable two output per cycle 1024-point FFT were developed as a reference to the proposed approach. All designs were simulated and their correctness were proven. The code was then synthesized using Synopsys Design Analyzer with a $0.25\mu m$ standard cell libraries

Table 3.5 shows the comparison between a single output 1024-point R2^{2}SDF FFT processor and our proposed reconfigurable R2^{2}MDF architecture presented in Section 3.5.4.1 which is capable of performing FFTs of different sizes and throughputs at a very low extra cost.

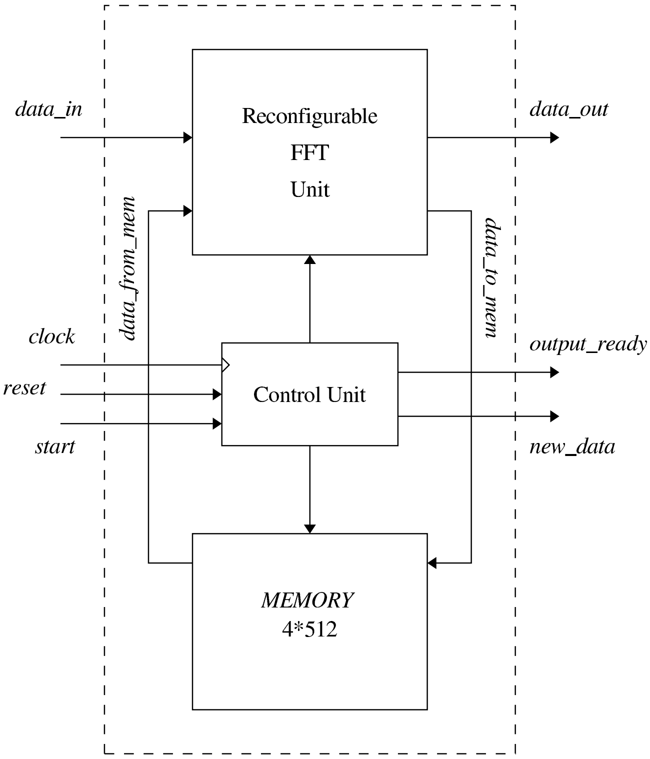

Figure 3.58: Small Data path Reconfigurable FFT Processor

Table 3.5: Normal and Simple Control FFT Comparison (UMC 0.25 μm)

FFT type	Area (μm^2)	Frequency (MHz)
Normal S2^2DF 1024-point	5509478	22.5
Proposed S2^2MF 1024-point (simple control)	5688091	20.3
Proposed S2^2MF (up to 1024 points multi-cycle control)	15282254	31.8

On the other hand Table 3.5 shows the R2^2MDF FFT processor with the approach discussed in Section 3.5.4.2. Most of the area is consumed in the extra memory which is composed of both the extra FIFOs and the intermediate data storage memory.

From the Tables above we observe that the Simple control approach is more appealing because of the flexibility it offers at a minimal price and because the size of the needed intermediate memory is large in comparison with the savings on butterflies.

3.6 Concluding Remarks

The nature of DSP applications is extensive processing of data. Within a given algorithm –as the studied SFGs suggest– the number of computations explode. Pure parallel implementations result in very high throughputs at the expense of enormous hardware resources especially for bigger sized algorithms. On the other hand, pipelined architectures intend to trade hardware resources of parallel architectures for speed. This is done by unrolling and re-rolling their SFGs swapping therein intermediate data in dynamic data paths.

Some applications such as digital filtering are pipelined and regular in nature and thus often do not require as much effort to optimize their architectures as required by other irregular architectures. The other studied algorithms above, namely: FFT, DCT and Viterbi decoding are among algorithms of higher degrees of irregularity where optimizations in their implementations can result in important gains.

The R2SDF, R2MDC architectures and a couple of their variants were studied to get a better understanding of th design challenges and requirements needed for the implementation of such structures. Although additional hardware resources are needed to construct the final pipelined architectures and there is always less than 100% utilization of hardware, the resulting architectures are usually orders of magnitude smaller in areathan their parallel counterparts, yet performing at acceptable throughputs.

In some cases parallel architectures provide much higher throughputs than needed while pipelined solutions throughputs do not suffice the needed application requirements. In this context, the proposed R2MDF architecture provide an intermediate solution that can be used for the implementation of several DSP algorithms' architectures. The main idea of the R2MDC architecture is to partition vertical loops in the R2SDF architecture by splitting them to two or more parallel data paths where possible and joining them again with a higher throughput data path resources. The separation of and combination of the data paths are guided by their SFGs. The resulting R2MDF architecture feature better hardware utilization than the traditional R2SDF one.

Studying the antecedent and other DSP algorithms architectures, we observe that these architectures exhibit the following aspects: they are PE based structures, the involve dynamic switching between data paths, they need simple control data paths utilizing counters, they have different types of PEs and they sometimes involve bit operations.

In the preceding discussion about these architectures in this Chapter, an obvious conclusion is that practical and optimized DSP algorithms architectures are not trivial. As a matter of fact their development is a result deep understanding of the algorithms, hardware design and system requirements. The resulting structures and architectures involve a good deal of human intellectual design efforts.

Chapter 4

Problem Definition: What Features a Reconfigurable DSP Processor Should Posses

Contents

4.1 Observations and Extraction of Primary Architectural Features of the Previewed DSP Architectures . . . 102

4.2 The Relation Between Key Reconfiguration Parameters to the Target DSP Applications . . . 104

4.2.1 Fine-Grain vs. Coarse Grain Reconfigurable Solutions . . . 104

4.2.2 Timing Issues . . . 105

4.2.3 Routing . . . 105

4.2.4 RTR Issues . . . 106

4.2.5 Partial Reconfigurability . . . 106

4.2.6 Programming . . . 106

4.3 Suitability of Studied Reconfigurable Architectures to DSP Applications . . . 107

4.3.1 Reviewed FG FPGA Architectures . . . 107

4.3.2 Reviewed CGRC Architectures . . . 107

4.4 Problem Solving Methodology . . . 109

4.5 Architectural and Reconfiguration Parameters Observed in the Design of the Proposed CGRC Solution . . . 111

4.6 Concluding Remarks . . . 114

In Chapter 3, carefully crafted architectures of various DSP algorithms were introduced. Common features of these architectures were also discussed. These observations lead us to recommending primary features and functions a CGRC solution for such applications should enjoy. By supporting these features we aim at finding an efficient CGRC solution architecture where feasible realizations of DSP algorithms can be achieved.

In this Chapter the required architectural features are extracted and discussed in Section 4.1. Then RC architectures features introduced Chapter 2 and how they relate to the extracted features are discussed in Section 4.4. Section 4.3 discusses the suitability of the reconfigurable architectures introduced in Chapter 2 for high speed processing of the DSP algorithms studied in Chapter 3. This then leads to the proposition of the main features of our CGRC processor in Section 4.4. Finally the summary and remarks Section concludes this Chapter.

4.1 Observations and Extraction of Primary Architectural Features of the Previewed DSP Architectures

In the Previous Chapter the following common architectural features were observed:

- *Regular Structure of Processing Elements:*
  Regular and repetitive structure of processing elements is clearly observed in most of the architectures studied in Chapter 3. This feature is most clear in the various FIR filters structures. This is due to the simple SFG that FIR filters possess.

  The FFT SFG however is more complicated. This increase in complexity result in more complicated structures such as dynamic butterflies and FIFOs. Nevertheless, a repetitive structure of these elements consisting of adders, subtracters, multipliers and FIFOs is observed.

  For more complicated DSP algorithms such as the DCT and Viterbi decoding, the repetitive nature of processing elements structure is less obvious because of the higher degree of irregularity. This irregularity not only noted in the more complex structures of processing blocks but also in irregular pattern of interconnectivity. The irregular patterns of interconnectivity are remedied by more complicated structures for data rescheduling control.

- *Interconnectivity:*
  In the architectures review in the last chapter, the following types of connections can be noted:

  - *Left-to-Right Row Connections:* In most of the reviewed architectures, we note that the data is processed in general from one side of the architecture to the other with processing elements handing processed data to be further processed by the adjacent ones. Therefore, most connections are found to connect processing elements located in the same row from left to right. This observations also support the *"Connection by abutment"* proposed by [36, 34, 37].
  - *Other Left-to-Right Connections:* i.e. from cells in different rows to cells in the adjacent columns at the left. This type of connections is needed to realize butterfly connections for example.
  - *Top-to-Bottom & Bottom-to-Top Connections:* This type of is needed for vertical passing of data like constants passing.
  - *Backward (Right-to-Left) Connections:* This type of connections although needed, is very rare and does not represent a key feature.
  - *Broadcasting of Data:* This is needed in some cases where inputs or other operands are needed to be shared by several processing elements for example in the transposed form FIR filter or in the ACS operation of the Viterbi decoder.
  - *Connections between different types of PEs:* This is needed to facilitate inter processing of data between storage and arithmetic units.

- *Arithmetic Operations:*
  Most of the operations carried out previously are vector addition, subtraction and multiplications. In multiplication truncation is needed to keep the resulting vectors within the fixed vector size to allow further usage of the results by other processing elements. However other operations are needed for special applications like Viterbi decoding. Such operations include comparison and selections but more importantly bit operations. Bit operations can be used to propagate the carry bit and produce decision bits used in the Viterbi ACS operation.

- *Storage Resources:*
  A key feature found in most of the studied architectures is local storage. Three main types of storage facilities were noticed:

  - *Storage for Holding and Passing Coefficients:* this feature observed in FIR, FFT and DCT implementations.
  - *FIFOs:* this feature is needed for scheduling data in pipelined architectures.
  - *Registering and Pipelining:* for pipelining and retiming of data and temporal storage of data.

- *Dynamic Data Routing:*
  The core feature of all the studied pipelined architectures is dynamic data routing. This enables operation scheduling and execution at the correct point of time.

This feature is associated with control flags provided by the scheduling control discussed next.

- *Routing and Operation Scheduling Control:*
  The generation of the control flags needed to govern switching and routing of data is the price traded for flexibility and area savings of non-parallel implementations. Fortunately, this is usually based on simply a counter as was illustrated by the architectures discussed in the previous Chapter.

- *Bit Manipulation Operations:*
  In error correction and other similar algorithms bit manipulations operations are common. The decision bits of the Viterbi decoder and the trellis window decode operations are two examples that has been covered in Chapter 3.

4.2 The Relation Between Key Reconfiguration Parameters to the Target DSP Applications

Chapter 2 has introduced a collection of RC solutions architectures. It should be pointed that DSP acceleration was probably not the main factor that influenced the organization of these solutions. However, several DSP applications were used to demonstrate the speedups achieved using these architectures. In this Section the key reconfiguration features that help better implementation of DSP applications are presented.

4.2.1 Fine-Grain vs. Coarse Grain Reconfigurable Solutions

As was discussed previously in Chapter 1, there is a tradeoff between computational flexibility and price in terms of area and processing speed requirements. A μp gains its versatility from the highly complex control structures it possesses and the software computational paradigm it adopts. These are the very factors that cause high area real estate and degrade computation time required for a given set of tasks.

On the other hand, FGRC solutions lose some of its flexibility for higher computational speeds. However, since FGRC solutions support no specific computational paradigm in particular, a high level of flexibility and close to maximum granularity are offered to ensure their versatility. This in tern causes in low effective area inefficiencies. In [20, 17] it is reported that a cost-performance factor ranging from 20 to 100 is penalized for circuits implemented in fine-grained FPGAs.

To reduce these inefficiencies fine-grained FPGA manufactures started to implement dedicated carefully crafted CG blocks within the FG FPGA to help achieve higher area efficiencies. These blocks include adders, multipliers, FIFO/RAM configurable units and even complete microprocessors as was noted in Chapter 2. In addition, FPGA vendors tend to carve their FPGA building blocks and routing architectures to suit most popular applications such as the realization of basic arithmetic operations and dedicated routing for the carry bits. Moreover, special design models and macros are provided to ensure efficient functional units implementations. These models include even μp soft cores.

4.2.2 Timing Issues

In CGRC solutions registering outputs is of great importance because this represents the feature clock frequency of the CGRC solution. As a matter of fact the target frequency was the factor deciding on the size of the basic functional blocks and influencing their architectures of the RAW introduced in Section .

On the other hand, the Radix 2 Feedback architectures discussed in the previous Chapter prohibits registering the functional units within the dynamic butterfly unit. This is because insertion of delays in feedback paths will result in breaking timing integrity of the feedback loop and hence will result in wrong processing of data. This feature is found in most FG FPGAs functional blocks.

In most CGRC solutions, outputs of processing elements are registered. Even more, functional blocks are shaped as small μps, and thus more than one clock cycle delay may result depending on the depth of their execution pipeline. This structure of processing elements may result in considerable speed degradation in the complete DSP algorithm computation and obstructs efficient implementation of well studied DSP algorithms architectures.

4.2.3 Routing

Routing facilities dramatically affect both the area efficiency and flexibility of any RC solution. Low flexibility in routing facilities result in higher effective area efficiency. On the other hand, low routing flexibility usually leads to arrays with poor usability. In [36] it states that flexible routing is a facility worth investing in. In this context efficient and feasible routing facilities topologies are essential in realizing efficient and versatile CGRC solutions.

For DSP applications as noted above, most connections are between adjacent processing elements. This observation was noted in [36, 34, 37] and thereupon, the concept of connection by abutment was adopted. In addition, long distance connections as well as broadcasting of operands are sometimes needed and therefore global buses are also implemented in the KressArray.

Some CGRC solutions adopt event-based computing paradigm. i.e. operations are carried out only when the valid operands arrive at the consumption ports of the target processing element. This paradigm is not a good solution when high throughput processing of DSP algorithms is sought. Moreover, realization of the well studied efficient data architecture also is not made possible in such a computational environment.

4.2.4 RTR Issues

RTR capabilities maximize the functional diversity efficiency of the CGRC solution [62]. The main benefits of RTR support are:

- *Facilitation of Operation Scheduling & Synchronization:* This is essential to realize pipelined architectures such as those discussed in Chapter 3 and many more.
- *Facilitation of Event Based Operation:* Which is a feature that can aid in coding and cryptographic algorithms.
- *Facilitation of Fast Switching Different Between Algorithms Configurations:* Assuming the availability of configuration cache, valuable processing time can be saved if fast switching between configurations is supported.

4.2.5 Partial Reconfigurability

Partial reconfigurability is surely a very interesting feature that opens opportunities for many computational capabilities. When associated with RTR, partial reconfigurations new endeavors of reconfigurable computing possibilities can be exploited. Partial reconfigurations can take place within the same algorithm such as those of the aforementioned pipelined architectures where dynamic butterflies are reconfigured independently of the rest of the architecture and of each other.

When possible, and at the availability of enough unconfigured resources, partial configuration enables more than one task operation. This way variable speed ups can be achieved by parallel processing of several tasks.

4.2.6 Programming

In many of the reported CGRC solutions a high level language compiler is provided to efficiently implement and schedule selected portions of the tasks on the CGRC coupled solution. These efforts aim at bridging the gap between the system programmer and the utilization of rather complex and powerful yet difficult to program reconfigurable arrays. Compilers however are not expected to recognize the rather finely crafted specialized architectures such as those introduced previously.

4.3 Suitability of Studied Reconfigurable Architectures to DSP Applications

In Chapter 2 several fine and coarse- grained reconfigurable architectures were surveyed. In this section their suitability for high-speed DSP algorithms processing is evaluated.

4.3.1 Reviewed FG FPGA Architectures

Fine-grained FPGAs reviewed in Section 2.3 suffer form the classical fine-grain disadvantages discussed previously in this thesis. However, it is worth mentioning that the architectures of the newest FPGA generations enjoy several features that remedy some of the classical fine-grained FPGAs shortcomings.

A variety of special blocks for RAM & FIFO implementations are now integrated in XILINX, Actel and ALTERA FPGAs. In addition, specialized DSP blocks of different flavors also appear in XILINX and ALTERA FPGAs. These DSP blocks feature multipliers with rounding/truncation capabilities. Special lines dedicated for fast carry propagation and more optimally tuned arithmetics blocks are also introduced.

Earlier generations of these FPGAs (also discussed in Chapter 2) did not enjoy the abovementioned improvements. This demonstrates that pure FG architectures although are very flexible, do not represent the most practical computational solution. FPGAs themselves are not pure AND-OR planes capable of implementing any number of 2 or more variable logic functions. As a matter of fact, they can be viewed as an elegant collection of coarser-grained *"logic"* modules.

4.3.2 Reviewed CGRC Architectures

Reconfigurable solution discussed in Section 4.2.2 represent good examples of state of the art CGRC architectures proposed in literature and in the market.

Coarse-Grained RC architectures presented in Section 4.2.2 have a strong FPGA flavor. But unfortunately, not all of the FPGA features is suitable for data flow applications. Nevertheless, deeper study of these architectures and their advocated features and concepts reveal that they mostly have been tuned and organized with huge distributed processing computational tasks in mind rather than pure data flow like processing. An example of that is the MorphoSys architecture that can efficiently realize SIMD topologies. In this context, the previewed architectures offer truly powerful computational solutions with processing power far more superior than the fastest μps. Unfortunately, all this processing power remains to be experimental because of the software gap between the CGRC solution and the system programmer [7, 28, 31]. This gap, if bridged, can allow the exploitations of the most intriguing RC capabilities.

Our goal here however is to find a feasible CGRC architecture for high speed DSP processing. In this context, The architectural features of Section 4.2.2 will be assessed from that perspective.

The μp like strategy of most of the introduced CGRC architectures can have the classical μp disadvantages discussed in Chapter 1. Moreover, the μp processing element strategy also result in duplicating several resources to meet RTR requirements such in the KressArray for example. However, because of the interconnectivity of the processing elements the data-fetch and the write-back penalties can be partially eliminated. However, this processing paradigm is not optimal for all high speed DSP applications because it does not coincide with the architectural based savings of specialized VLSI architectures as mentioned earlier.

The same argument holds for data-driven processing style adopted by the KressArray and the PACT XPP architectures. Here, important timing and synchronization cycles can be broken resulting thus in slower processing. In addition, data driven processing can result in data loss problems, although vendors such as PACT claim that they implement techniques that ensure prevention of such problems.

On the other hand, architectures introduced in Section 4.2.2 enjoy many advantages including the incorporation of local data storage facilities tat are in the form of register files in the KressArray, MATRIX, RAW, MorphoSys whereas PACT has a special type of a flexible processing element of memory operations.

Global and hierarchical routing are features exhibited by all of the aforementioned architectures. These features give a good deal of flexibility to the architecture. Nearest neighbor connections as well are of great importance and they are implemented by the abovementioned architectures. Anyhow, for DSP data flow applications full duplex connections such as that exhibited by the RAW architecture are not needed since their implementations mean extra area with no clear utilization in the studied family of DSP algorithms architectures.

Both dynamic and partial reconfiguration are adopted by all the discussed architectures. They are realized either by context memory of several layers or as instructions in the instruction memory of μp based architectures. Table 4.1 summarize the points for and against the aforementioned architectures from high processing speed of DSP algorithms point of view.

4.4 Problem Solving Methodology

Our goal is to design an efficient CGRC solution for DSP algorithms acceleration purposes. As it was discussed in Chapter 1, a rather complete general purpose computation SoC will integrate μps, data storage facilities, ASICs, fine-grained FPGAs and CGRC solutions resources. In this context we view our proposed CGRC solution as a flexible and a dynamic block taking care of specific DSP algorithms computations efficiently and making the results available to the rest of the SoC at minimal overhead costs.

A general purpose CGRC solution design is difficult to achieve. It is important to stress again that various design considerations as well as the computational scheme of the sought architecture will differ according to target computational tasks. In other words general purpose distributed computing solutions result in arrays of μp and event based computing such as the RAW and PACT architectures. Such architectures do not achieve maximum performance for many DSP algorithms and do not necessarily offer good effective area efficiencies. As a matter of fact, within a single CGRC array family, several design parameters could be tuned according to the family of algorithms to be implemented in order to achieve more efficient designs [34]. Therefore, a practical solution for finding more efficient architectures is to optimize the CGRC solution's architecture for a given set of applications [29, 30]. The DSP architectures studied in the preceding Chapter thus have inspired our design of the proposed CGRC solution.

A variety of DSP VLSI architectures were chosen for study in Chapter 3. The reason behind our choice of these algorithms and their architectures was that they cover a wide variety of features that span a very wide range of structures. The selected architectures exhibited features ranging from being simple and regularity to more complex with butterfly operations to irregular structures to dynamic scheduling to memory and bit operations. Therefore, most other DSP algorithms and their architectures are believed to be possible to implement using the our resultant CGRC solution.

Looking at the different SFGs of the DSP algorithms studied in Chapter 3 we note that within a given algorithm the complexity of computations and data dependencies increase dramatically within the algorithm. Hence, an accelerator implementing the complete algorithm although may need to have a larger data path can save substantial control efforts needed to partition the SFG and schedule its operation taking data dependencies in account. This claim is supported by the experimentations and findings discussed previously in Section 3.5 of the large data path and single cycle computation vs. the smaller data path and multi cycle computation for both FIR filtering and FFT operations implementations . Recall that the additional hardware for implementing multi-cycle operation were impractical and thus it was recommended that this extra control should be implemented in software. Still, the insertion of a software component in the algorithm computation may very well result in additional loss of computation time. At any rate, the very software computational paradigm is what results in the inefficient computational scheme of μps. Thus we target a CGRC solution design that solves an entire DSP algorithm with minimal need of additional control or aid.

As noted in Section 4.2.6 specialized compilers do not provide efficient realizations of the target DSP algorithms. The reason behind this is that only the mathematical formulae of the algorithm do not at the first glance reveal all the reduction possibilities that can be advantageous in finding feasible architectures. Indeed, synthesis of DSP algorithms involve clever mathematical reductions and derivations as well as thorough understanding of SFG and its data dependencies.

Moreover, special architectural features such as on-the-fly scheduling of operation and routing of data and RTR exploitation are features difficult to be efficiently achieved by current compilation tools. As a matter of fact, compilation on its own is a non-trivial task and complete instruction level parallelism extraction is not achieved by current superscalar or VLIW machines compilers. Adding extra synthesis tasks involving the above considerations plus complicated placement and routing tasks will further complicate the compilation problem. In fact, current synthesis tools do not always provide efficient results [28].

A programming paradigm that adopt library based configurations of DSP algorithms seems therefore to be a feasible solution for DSP applications. In such a paradigm well studied DSP algorithms' configurations are stored in a library and loaded when the specific algorithm is called. Here, the configuration files can be provided by the CGRC solutions vendors who have thorough understanding of both their CGRC architecture and the target DSP algorithms that may have influenced their design. Thus, the efforts of synthesizing them become transparent to the system programmer. The system programmer views the programming of the CGRC solution as accessing a library or a class.

RTR is a very attractive feature to be implemented. In most pipelined architectures dynamic reconfigurations and data routing are essential features [50]. As discussed in Chapter 2 some CGRC solutions employ multiple configuration contexts local to the processing elements and routing resources. This technique permits not only fast switching between configurations, but moreover, can hide the reconfiguration time needed to load inactive planes of contexts as the CGRC architecture is running. RTR should be as fast as possible to reduce the overall computation time. Local reconfiguration operation is required in both switching between configuration contexts background loading of configurations.

The architectures studied in Chapter 3 contained different computational and storage resources. Embracing a heterogeneous solution with different types of processing elements can result in smaller processing elements, meaning less area and thus higher effective area efficiency while maintaining higher functionality. Heterogeneity therefore result in high area efficiency yet with rich functionality [15, 98, 99, 76, 77, 20].

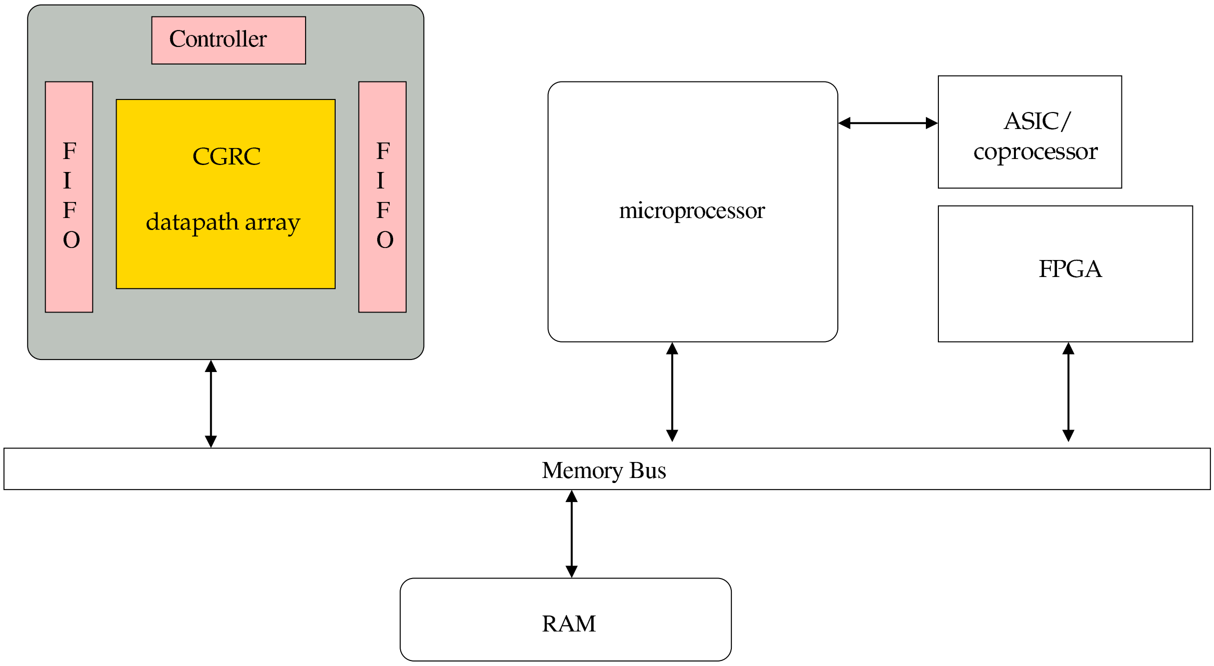

Figure 4.1: General architecture of a comprehensive computational SoC solution.

4.5 Architectural and Reconfiguration Parameters Observed in the Design of the Proposed CGRC Solution

In Section 4.1 of this Chapter the key features implemented in the DSP algorithms architectures were introduced. The relation between the main reconfiguration parameters and the extracted key architectural features of Section 4.2 were discussed. Next in Section 4.3 the suitability of reviewed reconfigurable solutions in Chapter 2 was discussed. Thereafter, our assumption and approach for our design of the proposed CGRC Solution was introduced. In the following, we list the resulting features to be supported by our proposed CGRC solution.

1. **General Architecture:**
   The proposed CGRC solution is assumed to be a part of a SoC integrating a μp (one or more), a memory as well as several other peripherals as shown in Figure 4.1. We assume also lose coupling between the μp and the CGRC block.

2. **Interfacing:**
   As stated above, we propose that our CGRC solution is a part o a more complete system. In this context it should have enough resources to operate with minimal need of external supervision. This important feature is intended to formulate a rather practical solution that is very much independent from the platform it operates in.

   2.1 *General System Concept:* In General, the CGRC solution should have the basic structure illustrated in Figure 4.1.

   2.2 *Input & Output FIFOs:* Since the CGRC system is integrated in a SoC, FIFOs are needed to maintain constant streaming of input and output as much as possible.

   2.3 *Control Resources:* The control of reconfiguration, operation will obviously be needed. More on the control is given below.

3. **Heterogeneity:**
   For higher area efficiencies while maintaining a high degree of flexibility.

   3.1 *Arithmetic-Logic Operations:* can be implemented using configurable ALUs possibly of different types.

   3.2 *Local Storage:* local to the configurable ALUs for coefficient storage.

   3.3 *FIFOs:* of variable sizes to aid in data scheduling.

4. **Mixed Granularity:**
   Some times bit or higher vector size operations are required therefore mixed granularity may be useful.

   4.1 *Vector Operations:* for targeted dominant operations operations.

   4.2 *Larger Vector Sizes Support:* by interconnecting several processing elements.

   4.3 *Bit Operations:* for coding operations such as Viterbi decoding.

5. **Routing:**
   Routing is a key element determining the flexibility of the design. Nonetheless, it should be as efficient and versatile as possible to keep the area efficiency in an acceptable range. The following features are chosen to optimize routing to the target DSP applications.

   5.1 *Data-Flow Rout ability:* Most connections as seen in DSP architectures are between adjacent elements.

   5.2 *Inter-Row Rout ability:* To allow butterfly-like connectivity.

   5.3 *Feedback Paths:* This feature is not frequently needed but nevertheless should be supported.

   5.4 *Global Data Sharing:* To enable data broadcast and operands sharing.

6. **Reconfiguration:**
   Incorporation of effective reconfiguration techniques can maximize the profitability of the CGRC solution. Agile reconfiguration techniques for managing various reconfiguration aspects are to be supported. The following are the main reconfiguration aspects to be observed.

6.1 *ROM Context Library:* We assume supported configurations are stored in a ROM. When needed configurations can be called from the ROM context library and loaded to the CGRC architecture..

6.2 *Reconfiguration Control:* For monitoring active and inactive contexts and to tag contexts either needed or not needed and need to keep or allow the update of contexts.

6.3 *Multi Context:* This feature permits the support of most of the following reconfiguration parameters.

6.4 *Background Reconfiguration:* Loading contexts in the background can hide the effect of reconfiguration time on the total computation time.

6.5 *Fast Reconfiguration:* Although RTR and background reconfiguration is supported, fast reconfiguration is still required to minimize the total computation time.

6.6 *RTR Considerations:*
To provide a good deal of flexibility and suitability for DSP applications we consider supporting the following RTR features:

   i. *Fast Switching Between Contexts:* 1-cycle switching feature is necessary for the implementation of structures such as the R2SDF without loss in timing integrity.
   ii. *Global RTR Control:* for enabling or disabling configuration switching depending on the operation conditions is required. Also providing timing stamps is essential to orchestrate the operation and scheduling of pipelined architectures.
   iii. *Support of Event Based Switching:* Time stamp flags or other event flags should be utilized to control context switching.
   iv. *Processing Elements and Routing Reconfiguration:* Complete dynamic reconfiguration should be allowed for both the operations and data routing of various processing elements.

6.7 *Partial Reconfigurability:*
At least partial configuration within a given task should be allowed to enable configuration of only the interesting portions of the data path.

7. **Operation and Control:**
Reconfiguration and complete system management need control resources to manage them. The following are the main points to be considered by the controller.

7.1 *Global Event Flags:* Such as time stamps needed for some architectures.

7.2 *Control of Operation Durations:* to keep records of needed and expired contexts.

7.3 *Conflict Handling and Computation Halting:* Assuming that the CGRC solution is connected to a SoC cases of input or output conflicts has to be covered. For example when valid inputs are not available, computations should be halted while holding the data in the middle of the configured pipeline until this input conflict is resolved.

4.6 Concluding Remarks

Because of the considerable savings achieved by CGRC solutions, a CGRC solution is to be adopted. However since in some applications such as Viterbi decoding, bit operations are needed, a mixed grained or at least limited fine grained solution is recommended. Bit operations are not only needed to produce data but moreover, to generate RTR governing event flags needed in many applications.

From Chapters 2 and 3 we deduce that for a CGRC solution for DSP computations we need a heterogeneous data path array architecture. RTR and partial reconfiguration features are essential to implement structures requiring for example the realization of dynamic butterflies.

To adhere to data flow timing considerations, an ASIC like solution is to be adopted. In such a solution, PEs and are considered to be coordinated together rather than independently and their data are assumed to be closely correlated. Therefore, optional registering of processing elements outputs is essential to ensure preserving breaking timing loops within the computation.

To manage operation and reconfiguration controllers are needed to orchestrate loading of contexts as well as the progress of operations. In this context, time stamp signals are needed to conduct RTR in various parts of the realized data path.

For a realistic CGRC solution operating as a part of a bigger SoC computational solution, the continuous stream of data in and out of the system is not guaranteed. In this context, FIFOs for input and output can be used to remedy this problem. In Section 5.1, these and more aspects are discussed.

Table 4.1: The main advantages and disadvantages of the reconfigurable architectures studied in Chapter2 from fast DSP algorithms processing perspective.

Architecture	Points for	Points against
KressArray	1. Local memory (register files) 2. Global buses 3. Connection by abutment 4. Several context layers 5. Dynamic and partial Configuration 6. Fast switching between contexts	1. Data-driven processing 2. Large rDPU 3. μp-like processing elements 4. Multiple layers of peripherals requirement
MATRIX	1. Local memory (register files) 2. Global buses 3. Multi Context (instructions) 4. Nearest neighbor connections 5. Dynamic and partial reconfiguration 6. Fast switching between contexts	1. μp-like processing elements
RAW	1. Local memory (register files) 2. Several routing networks 3. Multi Context (instructions) 4. Local data & instruction cache 5. Nearest neighbor connections 6. Dynamic and partial reconfiguration 7. Fast switching between contexts	1. μp-like processing elements 2. Full duplex interconnections
MorphoSys	1. Local memory (register files) 2. Global buses 3. Several context layers 4. Nearest neighbor connections 5. Dynamic and partial reconfiguration 6. Fast switching between contexts 7. Background loading of contexts	1. SIMD-like topology implementation
PACT XPP	1. Configuration manager 2. I/O interfacing 3. ALU and RAM type processing elements 4. Multi context memory 5. Dynamic and partial Configuration 6. Background loading of contexts	1. Data-driven processing

Chapter 5

The HPad

Contents

5.1 General Organization 118
  5.1.1 Integration Provisions 119
  5.1.2 Target Computational Paradigm 120
  5.1.3 The HPad Data Path Array Architecture 120
  5.1.4 Reconfiguration Techniques 121
5.2 The HPad DPA Architecture 123
  5.2.1 Arithmetic Processing Elements 123
  5.2.2 Memory Manipulation Processing Element 126
  5.2.3 Data Sharing Buses 127
  5.2.4 Reconfiguration Interface Sockets 129
5.3 Reconfigurable FIFOs 134
5.4 The HPad Control Architecture 136
  5.4.1 Operation Control 137
  5.4.2 Reconfiguration Control 137
5.5 The HPad Reconfiguration Mechanisms 141
5.6 The HPad Routing Topology 143
5.7 Synthesis Results 145
5.8 Area Efficiency Analysis 146
  5.8.1 Block wise Area Efficiency 146
  5.8.2 Overall Area Efficiency 147
5.9 Scalability 147
5.10 Concluding Remarks 151

In Chapter 4 both the functionality requirements of the processing elements as well as the reconfiguration aspects were studied together with DSP algorithms architectures in mind, reaching in conclusion the features listed in Section 5.1. The proposed CGRC solution, the *"HPad"* was designed observing the aforementioned features.

The HPad design was carried out through VHDL modeling. The HPad was completely modeled in Synthesisable Register Transfer Level (*RTL*) VHDL code. The developed VHDL models are parameterizable. This parameterization approach permitted the experimentation of a variety of design features simply by modifying some constants and recompiling generating thus different designs of different parameters. This powerful parameterization or *"Genericity"* is a property making VHDL modeling superior to Verilog modeling in terms of parameterization possibilities. The complete adequately commented VHDL model size exceeded 6500 lines of code.

In this Chapter the HPad RTR coprocessor architecture and features are described and analyzed. In Section 5.1 the assumed computational environment is posted. The general architecture of the HPad is introduced in Section 5.1 as well. Section 5.2 then presents a detailed description of the HPad's data path array and its different components. Section 5.4 discusses the reconfiguration and operation control parts of the HPad. In Section 5.7 synthesis results are presented and discussed. Section 5.8 a discussion on the HPad's area efficiency is presented. In Section 5.9 the scalability of the HPad is discussed. Finally, Section 5.10 conclude this Chapter.

5.1 General Organization

Observing the architectural parameters listed in Section we reach the proposed CGRC solution architecture shown in Figure 5.1. We name our proposed CGRC solution the Heterogeneous Pad (*HPad*). The heterogeneity characteristic of the HPad comes from its different types of building blocks: data path and control as well as the different types of processing elements within its data path array. In addition, FIFOs and a reconfiguration Context libraries are needed. The FIFOs are needed to ensure the continuous streaming of data to and from the HPad's Data Path Array. The Context library can be realized as any arbitrary type of ROM: EPROM, EEPROM, Flash or any type of non volatile memory storing pre defined configurations. Since - as will be discussed below in this Chapter - configurations are written line by line to the data path array, there is a notable similarity between it an a note pad that one uses for solving a problem, writing thus line by line and discarding the used page when starting a new problem (analogous to reconfiguration).

The HPad feature three main types of blocks:

- The data path processing array,
- FIFOs for input and output interfacing, and

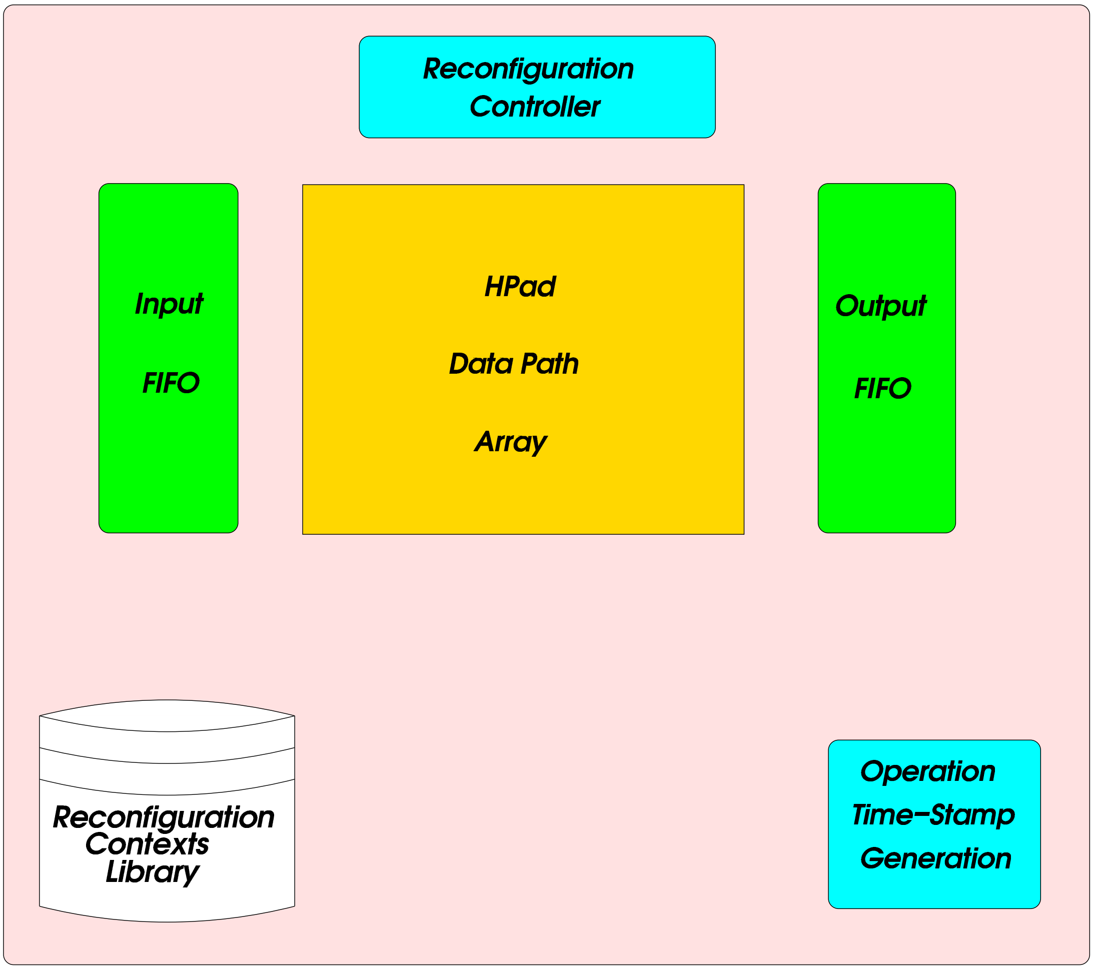
Reconfiguration Controller
Input FIFO
HPad Data Path Array
Output FIFO
Reconfiguration Contexts Library
Operation Time-Stamp Generation

Figure 5.1: A simplified structure of the HPad.

- Reconfiguration and operation control.

5.1.1 Integration Provisions

Since we have designed the HPad to be loosely integrated in a SoC, we expect that it accepts instructions and likewise, consumes and produces data from and to its surrounding environment. Accordingly, we suppose that the outside environment is not completely aware of the operation progress and conditions of the HPad. Similarly, we deduce that, the HPad needs not to be fully aware of the operation conditions of its surroundings. What the rest of the computational SoC components should know about the HPad is the set of supported operations that the HPad can perform.

From the HPad side, it should accept the computational tasks when it is available and signal if the required task can be accepted or not and likewise also signal when it is completed. Hence, a simple form of handshaking is required to assign new tasks to the HPad and to know if the new task is accepted and whether or not is the computational task is finished.

When a computational task is accepted, the HPad should be configured and then the operation starts and continues on the assigned scope of data. As the computations are in progress, special synchronization flags are generated to help schedule pipelined-like architectural implementations. The abovementioned tasks are assigned to the control and configuration resources of the HPad.

In the same context, The HPad is assumed to be integrated with the rest of the SoC through a shared memory bus. As the bus may be used by other components of the SoC, it is not expected to be available to the HPad continuously.

Since the HPad is to realize high speed DSP algorithms architectures, it needs a continuous supply of inputs and correspondingly, produces continuous output for consumption. In order for it to operate correctly and efficiently, this contentious streaming of data is to be sustained. To keep the HPad data path array running at the maximum pace, flexible FIFOs are used at the input and output ends of the HPad data path processing array. The FIFOs are to be charged and discharged with data in a fashion that keeps the HPad running at the maximum possible speed. In case a read or write conflict still occurs, the control unit should intervene and put the operation of the HPad on hold until this conflict is resolved in order to ensure the validity of the produced outputs.

5.1.2 Target Computational Paradigm

The HPad is to operate in a library-based computational paradigm computing for complete DSP algorithms. In such a paradigm, the user calls, say, an FFT operation to be operated on a specified region of data. This function call triggers the HPad. The HPad configuration controller retrieves the appropriate configuration from the configuration ROM, loads it in the HPad array and begins computations.

This computational paradigm simplifies to a great extent the compilation problem, leaving the difficult task of algorithm synthesis to the hardware designer as have been discussed in Chapter 4. In addition, the μp and other on board peripherals are very much let free to perform their own required tasks.

5.1.3 The HPad Data Path Array Architecture

The core of HPad is its data path dynamically reconfigurable *D*ata *P*ath *A*rray (*DPA*). The HPad array consists of three main elements:

- Arithmetic-Logic Processing Elements,

- Memory Manipulation Processing Elements ,and
- Data Sharing Bus Elements.

As Figure 5.2 illustrates, the HPad array is organized with rows of memory and arithmetic units processing elements placed interchangeably. Data sharing bus elements are placed horizontally between rows and vertically between columns allowing thus for long distance data routing both vertically and horizontally. Section 5.2.3 describes the data sharing bus elements in more details.

The arithmetic-logic processing elements take care of arithmetic and logic operations. They read tow vector operands and a carry in bit and produce a vector in addition to a single bit outputs. several of these arithmetic-logic units can be interconnected to form arithmetic units of bigger vector size inputs and outputs. In Section 5.2.1 a more detailed discussion about the arithmetic-logic processing elements is given.

Memory manipulation elements serve the purpose of local storage as well as and FIFO operations. Several f the memory processing elements can be interconnected to form various sized FIFOs. Section 5.2.2 provides more details about the memory processing elements.

All of the above elements are interconnected and interfaced through the reconfiguration sockets. Each reconfiguration socket encompasses an element and manages its reconfiguration and connectivity with other components of the HPad array.

5.1.4 Reconfiguration Techniques

The reconfiguration problem was tackled at two levels of abstractions fronts: global and local. Globally, the supported configurations are stored in the configuration ROM that supply the proper contexts to each of the reconfiguration interface sockets. Locally, reconfiguration interface sockets mange multiple context words and switch between them also according to global or local events. Global events can be generated from the reconfiguration controller or from time stamps. Local events are produced either depending on the processing element results or can be also chosen according to passed bits. In Section 5.4 a more detailed discussion about the reconfiguration techniques is provided.

The idea of the use of reconfiguration sockets has fortunately facilitated multi-context support, dynamic reconfiguration (RTR), partial reconfigurability and background loading of contexts as follows:

- *Multi Context Support:* Reconfiguration interface sockets are supplied with multiple reconfiguration context registers to store more than one context.
- *RTR Reconfiguration:* RTR is permitted with the aid of the special switching facilities of the reconfiguration sockets as is described in Section 5.2.

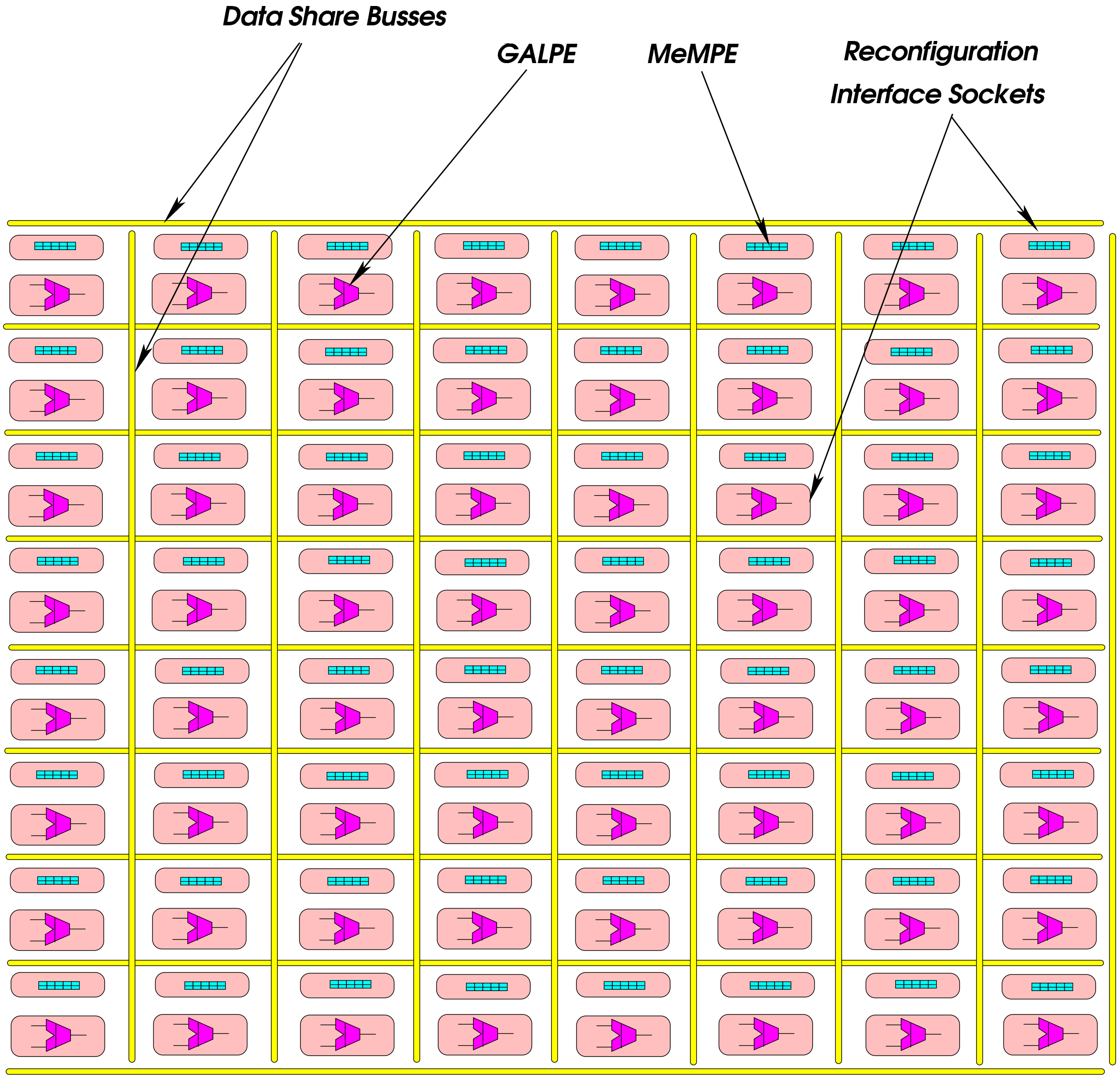

Figure 5.2: A simplified structure of the HPad DPA, here the current implementation size of 8×8 array is shown.

- *Partial Reconfigurability:* In addition, since the contexts containing the operation and routing information of each element is provided to it locally through its reconfiguration interface socket, individual sockets can select arbitrarily any of the contexts stored locally in it and thus provides partial configuration capabilities.
- *Background Reconfigurability:* Since multi contexts is supported, an obsolete context can be updated in the background while processing continues with another active context.

5.2 The HPad DPA Architecture

The DPA architecture of the HPad is its core processing resource. The HPad DPA shown in Figure 5.2 is arranged as a square array with the number of processing elements of the same type is the same column and row wise. The HPad data path array is designed to ensure high speed of operation for high computational power demanding DSP algorithms. To serve this purpose, the HPad data path array encompasses two types of processing elements arithmetic-logic processing elements and memory manipulation processing elements. The routing and interconnectivity of the processing elements was influenced by the studied architectures in Chapter 3. For extra routing flexibility, horizontal and vertical data sharing buses allow for long distance and feedback data routing.

The VHDL description of the HPad DPA was written with parameterization considerations. A package containing the most important parameters as constants forms the parameters of the HPad DPA. The VHDL code of the HPad array is carefully written to avoid conflicts in compilation and results as these constants in the package are modified in order to obtain a new array with different parameters. The most important of these parameters are:

- The number of processing elements per row,
- The data vector size,
- The number of vector storage positions in the memory processing elements,
- The input size of the reconfigurable FIFOs,
- The output size of the reconfigurable FIFOs, and
- The instruction word sizes of each of the elements.

In the following, more details about the various types of elements composing the HPad DPA is described. The description of each element's reconfiguration interface socket is described as well.

5.2.1 Arithmetic Processing Elements

The arithmetic-logic elements are obviously the most important type of building elements composing the HPad DPA. The following main points were observed when designing this ALU-like unit:

- It should support all the arithmetic and logic operations encountered in the course of our study DSP algorithms architectures.

- It should also support limited bit operations namely:
    - carry,
    - comparison flag (needed for the Viterbi decoding ACS unit)
    - zero flag, and
    - negative flag.
- It should allow optional registering of outputs.
- Formation of larger vector size operation should be possible.
- Multiplication and producing a standard vector size output should be allowed.

In addition, re usage of computed results is also permitted to add extra flexibility and to allow for MAC like instructions.

As is the case with all ALU units, the arithmetic logic units of the HPad operates on two vector operands and produces one vector result. For a regular and usable structure of the HPad DPA architecture, the operands and products vector sizes should be constant throughout the array. Therefore, the output data range should be assumed to remain representable within the input operands data width.

Now since multiplication operation on a pair of $N-bit$ vectors result in a $2N-bit$ vector result. The selection of the $N-bit$ final processing element's result out of the complete $2N-bit$ product is an issue to be resolved. In case both operands were integers, the most significant $N-bits$ are selected. However, this is not always the case. In the FFT operation, for example, all the coefficients are fractions less than one which means that the selection of the most significant $N-bits$ will result in huge truncation error. In the DCT operation, the coefficients are also small numbers but can exceed one. Multipliers used in the architectures presented in Chapter 3 were attached to truncation units that select the best $N-bits$ of the product depending on the typical values of its operands. Truncation and selection of the output vector range is therefore essential and was implemented with multiplication. To aid in the selection of the final result bits a multiplexor of log_2N select bits is utilized for full range selection of the final result.

The area required by multipliers is obviously much larger than that of adders. In the studied architectures in Chapter 3 the ratio of multipliers to adders ranges between $4:6$ in the FFT implementations and rises to $1:1$ in most of the the FIR implementations. To save in area, several types of the arithmetic units were developed. The first version, the *A*rithmetic-*L*ogic *P*rocessing *E*lement (*ALPE*) possessed no multiplication capabilities, instead it possessed simpler add/subtract, ACS and other logic operations. The second version the *M*ultiplication capable ALPE, (*MALPE*) has multiply and *M*ultiply *Ac*cumulate (*MAC*) capabilities but in the other hand has less logic support operations than that of the ALPE. A third version is the *G*ross *ALPE* (*GALPE* that possesses all the ALPE and MALPE capabilities.

Several topologies of the HPad were tested for their efficiencies and suitability for the implementation of some of the DSP algorithms. The GALPE based DPA was found to have the highest area efficiency for the studied DSP algorithms. Detailed instruction word organization and description of the GALPE is provided in Appendix B.

A simplified illustration of the GALPE unit is given in Figure 5.3. The input operands are fed to the GALPE unit where they are operated on according to the given instruction. The truncation information is also provided by the instruction. The results (vector and bit) are produced. Operation result flags (zero_flag and neg_flag) are generated afterwards. The Bit-out provides either the carry out or, in case of a comparison operation, an indicator bit pointing to biggest operand (by being set to '0' when the X operand is larger and '1' if the Y operand is larger). The final results are then optionally registered and delivered as outputs of the GALPE block.

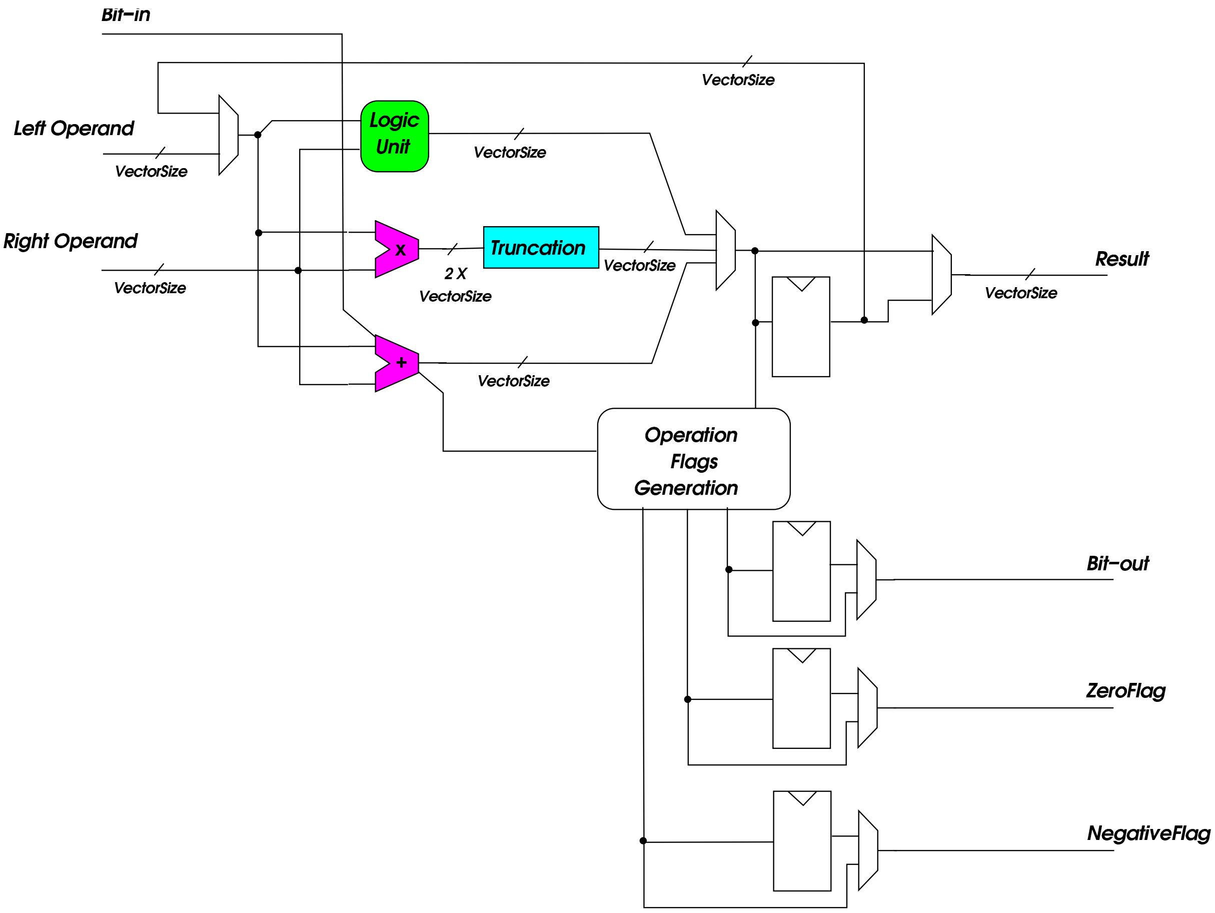

Figure 5.3: A simplified block digram of the GALPE unit.

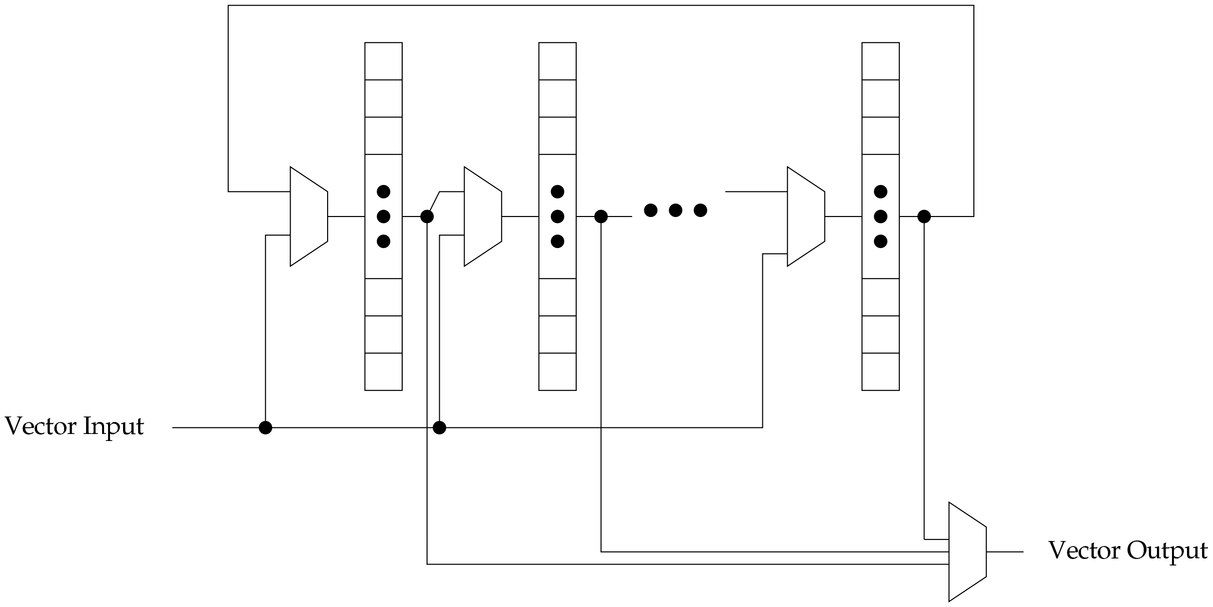

Figure 5.4: The vector functionality of the MeMPE units

5.2.2 Memory Manipulation Processing Element

The *M*emory *M*anipulation *P*rocessing *E*lement (*MeMPE*) serves the following main purposes:

- to act like local memory for coefficient or data storage,
- to be configured as FIFOs for data synchronization, and
- to perform several fine-grained functions that require bit-level access.

The later fine-grained operations include bit shift, storage and update and most interestingly the Viterbi decoders TB decoding function. The number of vector storage positions of the MeMPE block is kept generic. The operations of the MeMPE include complete memory content transfer to neighboring MeMPE elements, vector shifting and loading, bit shifting and loading and the Viterbi decoder TB operation. In addition, each vector or bit can be accessed for input or output.

The vector functionality of the MeMPE is illustrated in Figure 5.4. A vector can be loaded in either of the array positions. It can be also shifted left in the array. In addition, complete array of vectors can be updated. Any of the vectors can be selected as output as well. This configuration allows the MeMPE units to be used as RAM as is discussed above or as register file in the vicinity of the GALPE elements placed in an interleaving manner as depicted in Figure 5.2 above.

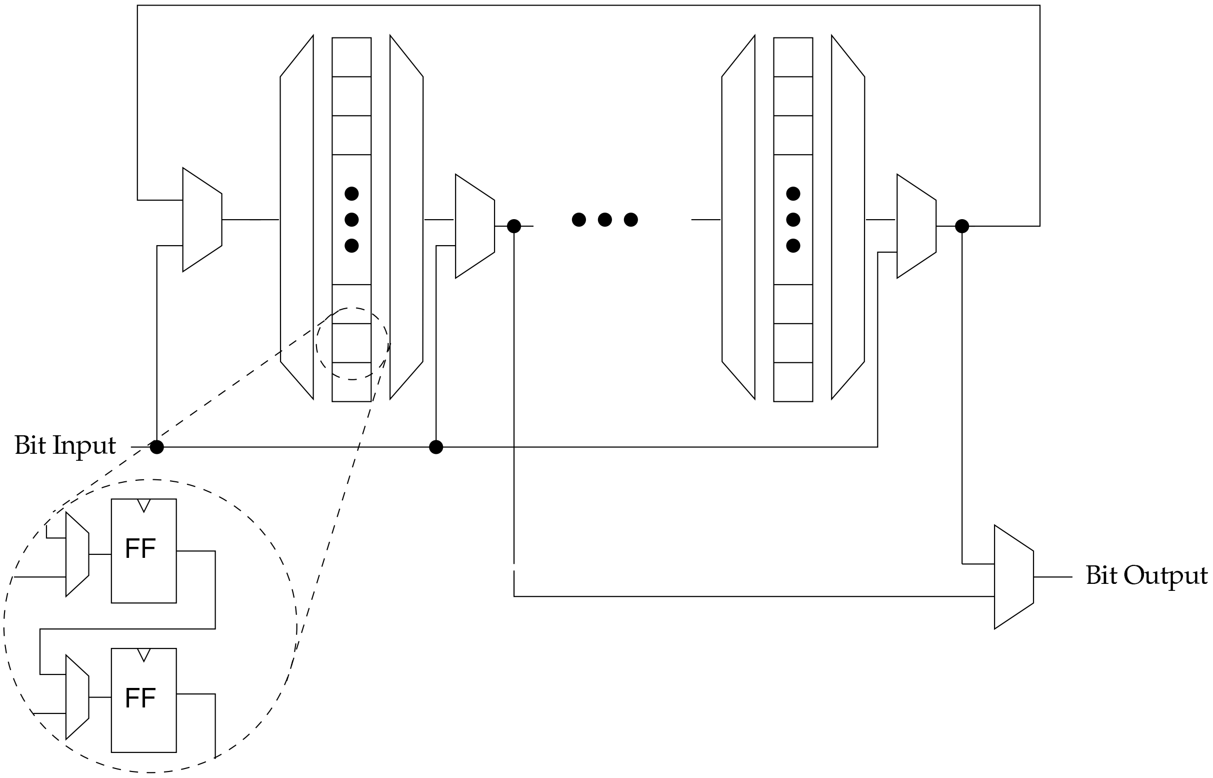

Figure 5.5: The bit functionality of the MeMPE units

A simplified illustration of the bit operations is shown in Figure 5.5. As shown in the Figure, bit load and bit shift operations are allowed . Each of the vector storage locations is composed of several single bit registers. Each of the registers input is selected from the adjacent more significant bit or from outside. The enable signal of the single bit registers are used to enable or disable the registers which facilitates the update of either one bit or more of the stored data.

The inputs of the MeMPE unit include bit address and vector address. In the current implementation of the HPad two vectors of data registers are incorporated in each MeMPE unit. This number of vectors is thought to be adequate for the targeted DSP applications. This assumption is validated later in Chapter 6 where several DSP algorithms were implemented using the HPad with this parameter. This particular parameter and others are listed in Table 5.1 later in this Chapter. The instruction set of the MeMPE unit is provided in Appendix C.

5.2.3 Data Sharing Buses

The *D*ata *S*haring *B*us (*DSB*) main function is the provision of long range data routing. Two versions of the data sharing bus were developed: a long range bus covering the

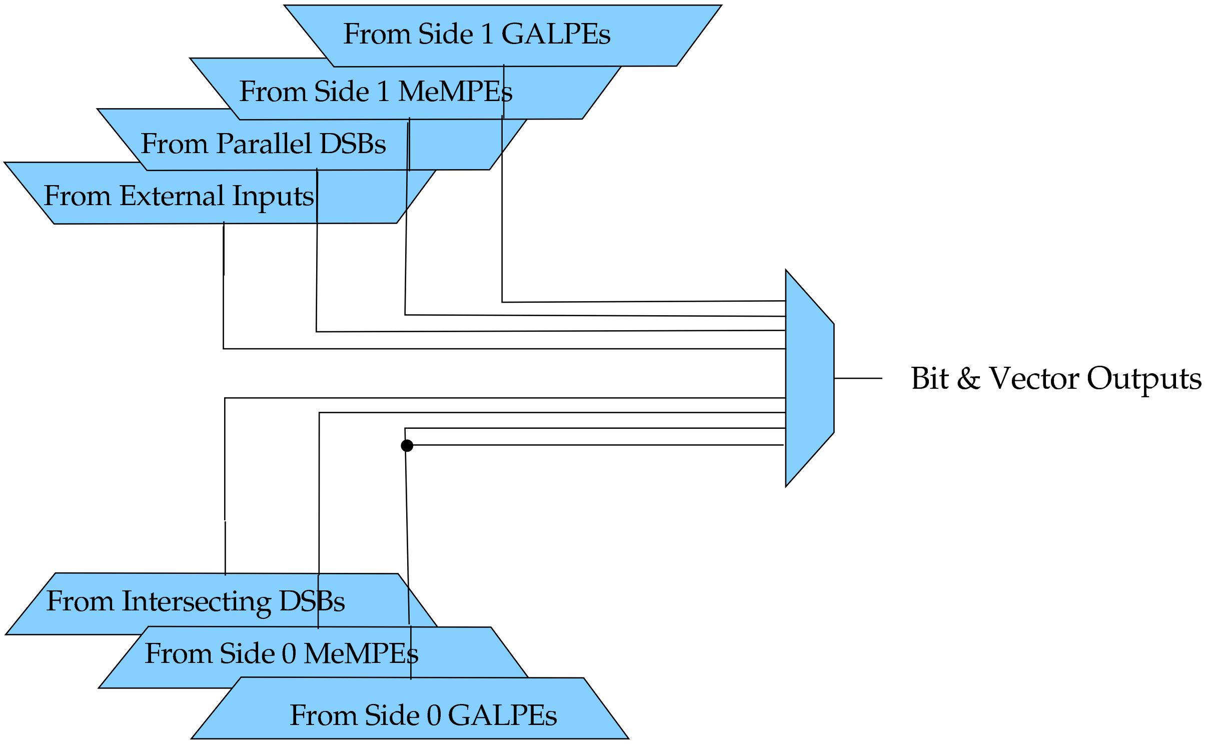

Figure 5.6: structure of the DSB

complete DPA and a short range one spanning only a few processing elements. As the length of the long range bus is parameterizable by the number of element per row in the HPad DPA, the span of the short range data sharing bus is parameterizable as well. As will be shown in Chapter 6, the long range DSB sufficed for the target applications.

The number of elements per row in the current implementation of the HPad is 8. This parameter and others fixed for the current implementation of the HPad are listed in Table 5.1 shown later with the synthesis results in Section 5.7.

The same architecture of the DSB is used to form the horizontal and vertical buses as shown in Figure. The buses select its single vector output from the outputs of the GALPE, MeMPE units along both of its sides and the intersecting buses outputs. The data sharing bus is capable also of routing a single bit in a similar manner. Dynamic reconfiguration of the DSBs is also possible through their reconfiguration interface sockets. DSBs can be interconnected to each other and can also broadcast external inputs.

Figure 5.6 depicts the structure of the DSB. More information about the instruction set (addressing information) of the DSB is given in Appendix D.

5.2.4 Reconfiguration Interface Sockets

The *R*econfiguration *I*nterface *S*ockets (*RIS*)of the various building elements of the HPad are the key elements facilitating RTR, partial configuration and local context storage. The RIS of each of the building element of the HPad implements not only the reconfiguration of the function of the hosted processing element, but moreover, it is responsible of the input data routing to the hosted processing element.

Each reconfiguration interface host dispenses of one vector and one bit output to the rest of the HPad data path processing array. This simplifies the routing problem being solved now at only the input side of the RIS. This topology also permits sharing the output of any element by one or more other elements: GALPEs, MeMPEs and DSBs.

Each RIS is supplied with a number of data operands (depending on its type and the type of the element it hosts), control event flags from the event flag generator, several signals form the Configuration controller, and global signals such as the *reset* and *clock* signals. The signals coming from the Configuration Controller include configuration control signals and configuration contexts.

Each RIS in principle can carry several context registers. In the current implementation of the HPad two contexts registers per elements is used. This and other parameters of the current realization of the HPad are listed in Table 5.1. The aforementioned couple of context word registers can carry either two configurations of separate static implementations algorithms (i.e. where no RTR operations are required), or the configuration information of an RTR implementation of a given algorithm.

The management of contexts loading and updating in the RIS is illustrated in Figure 5.7. The configuration shown in Figure 5.7 facilitate fast loading of the HPad DPA with contexts. The RIS can keep both of the current context vectors, update one of them or shifts new contexts in both context registers as depicted in Figure 5.7. As shown in Figure 5.7 new contexts can be shifted through the upper context registers in each row of RIS units, shifted through the lower context registers in each row of RIS units, or shifted upper to lower in each RIS to the upper of the next RIS in the same row and so on in a zig-zag pattern. The above functions are determined by a two-bit instruction explained in Appendix A. More on configuration contexts management and configuration loading topology is given in Section 5.4.2.

Each RIS selects the proper context dynamically according to the seven most significant bits of the context vector as shown in Figure 5.9. The control bit deciding on the configuration number is selected from one of the following signals:

- signal provided by the reconfiguration controller,
- selected input bit,
- negative flag,

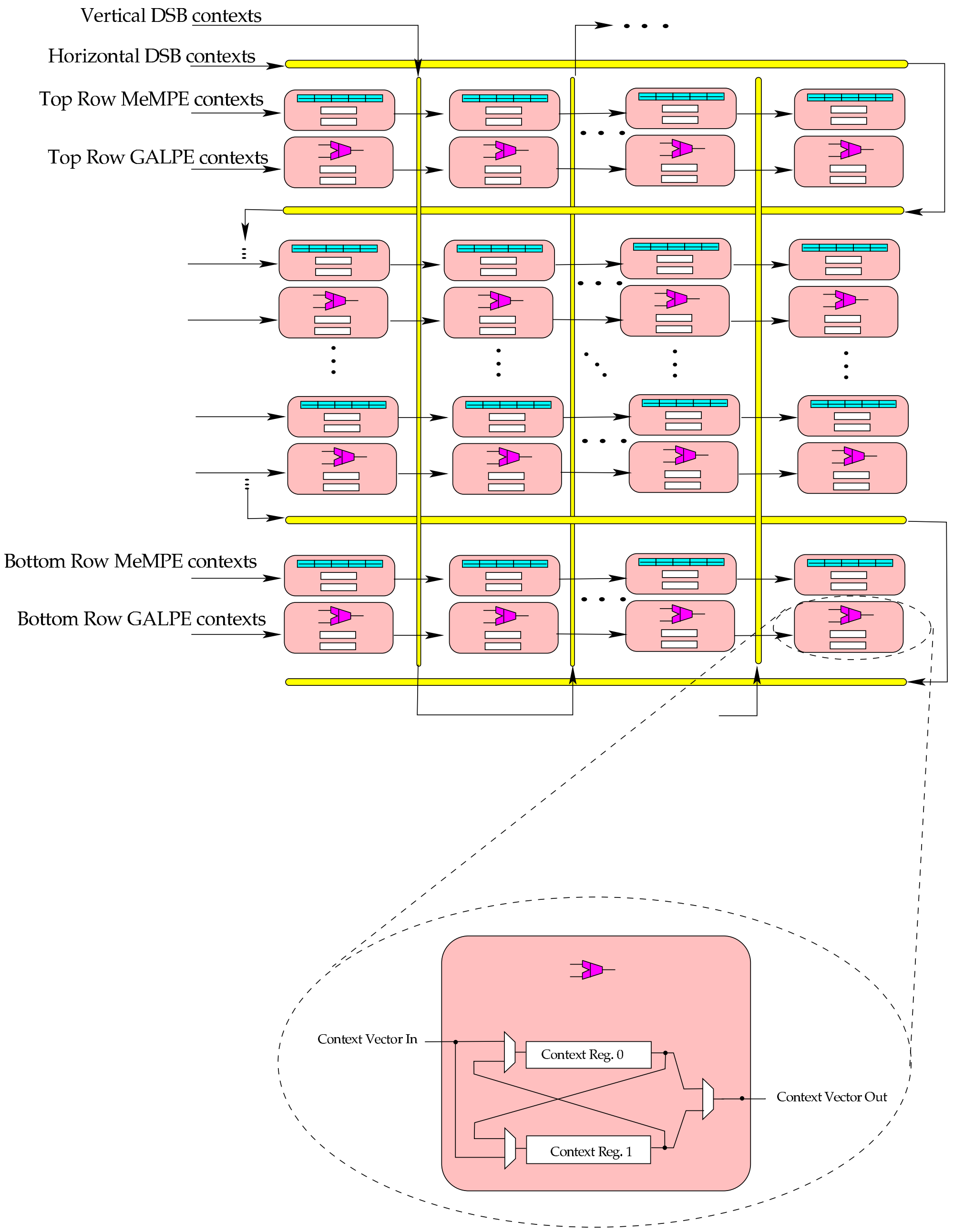

Figure 5.7: Configuration contexts updating in the HPad.

- zero flag or
- one of the $2 \times ElementsPerRow$ bits generated by the operation controller

For static configurations, the 3 MSBs of the context vector should be 000 forcing thus the number of the selected context to come directly from the configuration controller. Alternatively, the selected control bit can be passed from a previous processing element or even form the same encompassed processing element (MeMPE or GALPE). Additionally, control bits generated externally by the operation controller from time stamps can be selected by setting the MSB of the context to 1 and the individual bit address is stored in the $log_2 ElementsPerRow + 1$ bits at the left of the 3 MSBs of the reconfiguration context vector.

The above discussion described the common features of the RIS of the different building elements of the HPad DPA. Of course, each building element of the HPad DPA resides in a different type of reconfiguration RIS designed for it specifically. In the HPad data path array we have three different types of RIS: one for GALPE units, one for MeMPE units and one one for the DSBs. The main features of these RIS are given below.

5.2.4.1 Arithmetic Precessing Elements RIS

The GALPE RIS manages the routing and reconfiguration of the input signal to the GALPE unit. The GALPE RIS selects both operands for the GALPE unit from 16 neighboring PEs.

The different routing possibilities of a GALPE unit within a RIS is shown in Figure 5.8. As discussed in Chapter 4 the HPad architecture is oriented towards efficient realizations of DSP algorithms data paths. Therefore, connections allowing data to flow from one side of the HPad DPA to the other are implemented. Four connections to the nearest neighbor GALPE RIS of the adjacent left column, from the same row, one row above, tow rows above and one row below. For efficient implementations of structures involving coefficients stored in local memories or FIFOs for data management, local access to MeMPE units are made available in a similar manner. Furthermore, for more efficient realizations of butterfly−like structures, connections of the above and below GALPE and MeMPE are implemented. For long range connections, access of data share buses above and below the GALPE RIS is also supported.

All RIS can switch between two of the contexts as discussed above. The context vector is organized as depicted in Figure 5.9. As mentioned above, the MSB 7 bits are used for defining the RTR switching scheme. The next 4 bits are used to define the address of the first operand and the following 4 bits are used to define the second operand's address. To reduce the context size the bit in operand to the GALPE unit is assumed to have the same address as the first operand. The next 4 bits are used to define the truncated portion of the result in case of a multiply operation. Finally, the least significant 6 bits contain the instruction passed to the GALPE unit.

The output of the GALPE unit is passed though the GALPE RIS to the rest of HPad DPA.

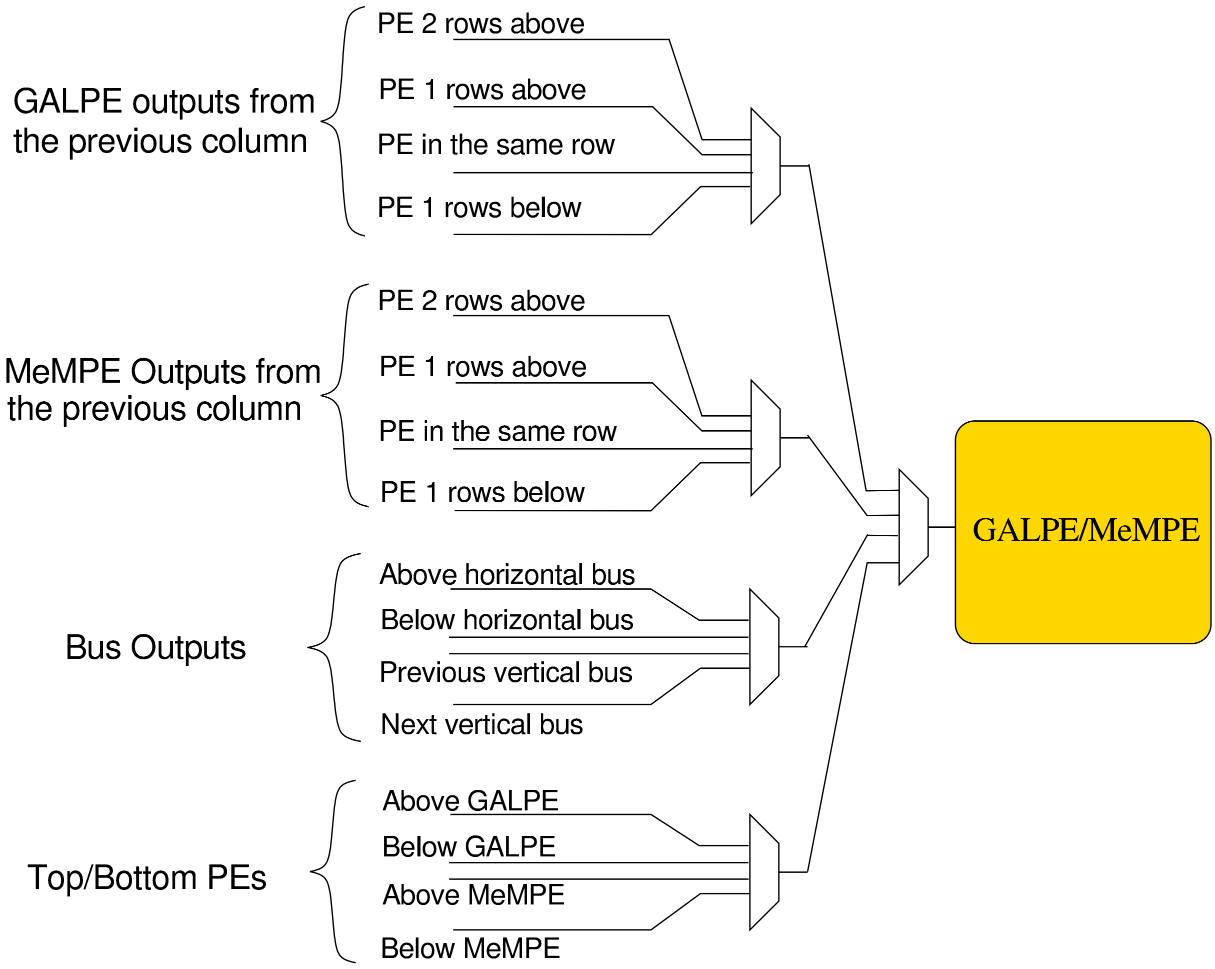

Figure 5.8: Routing of operands to the GALPE and MeMPE units.

5.2.4.2 Memory Manipulation Processing Element RIS

The MeMPE RIS also manages the routing and reconfiguration of the MeMPE units. In contrast to the GALPE RIS, the MeMPE manages only one vector and one bit input to the encompassed MeMPE unit.

The MeMPE RIS units also route an array of vectors to and from the MeMPE units to the neighboring MeMPE RIS units of the same row at the right and left of it. This is to facilitate the transfer the complete stored values in the MeMPE units through the array. Such a capability might be needed for some applications such as the TB operation of the Viterbi decoding algorithm. Although the transfer of the complete array of data of the MeMPE units may seem a bit expensive, practical sizes of the MeMPE units are not expected to exceed 4 data vector registers, or 8 at the most. As a matter of fact and as has been mentioned above, the current implementation of the HPad has MeMPE units

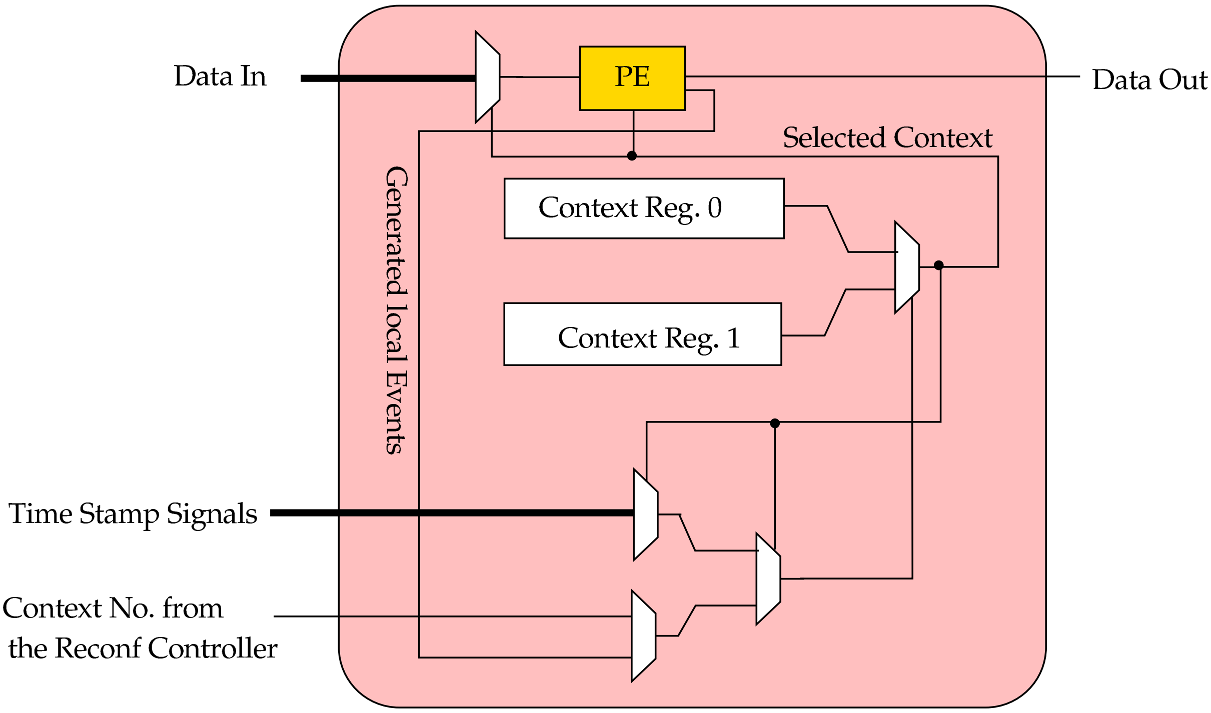

Figure 5.9: Context selection in RIS of the GALPE and MeMPE units.

of only 2 data vector registers. This size of the MeMPE units seems to be sufficient as is validated later in Chapter 6.

In addition, the routing of the MeMPE RIS unit is similar to that of the GALPE RIS unit shown in Figure 5.8 . This scheme permits feasible implementations of the studied architectures of Chapter 3 including local storage of constants, access of several constants periodically, formation of larger ROMs and FIFOs by cascading of several of the MeMPE RIS units. All this is possible along with local access of operands from and to neighboring units.

The organization of the context vector of the MeMPE RIS units is shown in Figure 5.9. The 7 MSBs are used to define the RTR switching scheme. The next eight (4 and 4) bits are used to store the output and input bit addresses and respectively. Similarly, the following two (1 and 1) bits are used to specify the output vector addresses respectively. Here again the selection address of the input vector and bit are assumed to be the same to reduce the reconfiguration area overheads. The selection address of the input vector (and bit) is specified by the following 4 bits. The least significant 4 bits holds the instruction of the encompassed MeMPE.

The MeMPE RIS passes the output bit and vector (in addition to the complete array) to the rest of the HPad DPA. Other control inputs such as the time stamp vector is used to manage the RTR switching of contexts similar to all other RIS.

5.2.4.3 DSBs RIS

The DSB RIS manages the routing of the DSB. This routing can also be changed dynamically at run time.

The organization of the DSB RIS context vector is slightly different than that of the GALPE and MeMPE RIS.

Next to the 7 bits needed for RTR configuration, a single bit is used to specify if the input is sampled directly from the input of the HPad DPA. The next bit is used to specify the top or bottom inputs of the DSP are selected. The following 3 bits are used to select the input bit address of the DSB which can be different from the address of the input vector specified by the least significant 3 bits.

5.3 Reconfigurable FIFOs

For a loosely coupled reconfigurable coprocessor design of the HPad (which is thought to be more general and versatile) input and output FIFOs are needed for the following two main reasons:

1. *to maintain a continuous flow of data to the HPad data path array:*
   In a complete SoC computational solution it is unrealistic to assume that all computational blocks have dedicated access to the memory or source of data. We therefore assume a shared memory organization of the SoC. As depicted in Figure 4.1, the memory is accessed by the HPad through a data bus. At his stage the type of data bus is not yet defined since the other components of the SoC are not decided on.

   As a matter of fact, we aim at designing a system independent reconfigurable coprocessor solution which is more versatile and helps us to concentrate on the general computational problems of the targeted HPad. A FIFO stores as much input data when the bus access to the memory is granted to make it available to the HPad data path array continuously. This is important since the HPad is designed to implement pipelined DSP algorithms architectures which assumes continuous data input stream.

   The same above discussion also applies for the output data flow. Therefore an output FIFO is also required to manage the outputs.

2. *to adapt the size of the HPad data path area to the size of the memory buses:*
   After reducing the sizes if the inputs and outputs of the HPad data path array to a manageable number of 16 (8 inputs and 8outputs) as discussed in Section 5.6, we need to consider interfacing it with the memory bus that is not necessarily of the same size as the HPad DPA. Hence, the utilization of sage of FIFOs accepting and delivering data vectors of different sizes can solve this problem.

Since a couple of FIFO units are needed (one for output and one for input) and since the type and size of the data bus is not defined, we decided to design a reconfigurable In/Out FIFO whose size of the input and output vectors is configurable. The general architecture of the reconfigurable FIFO is shown in Figure 5.10.

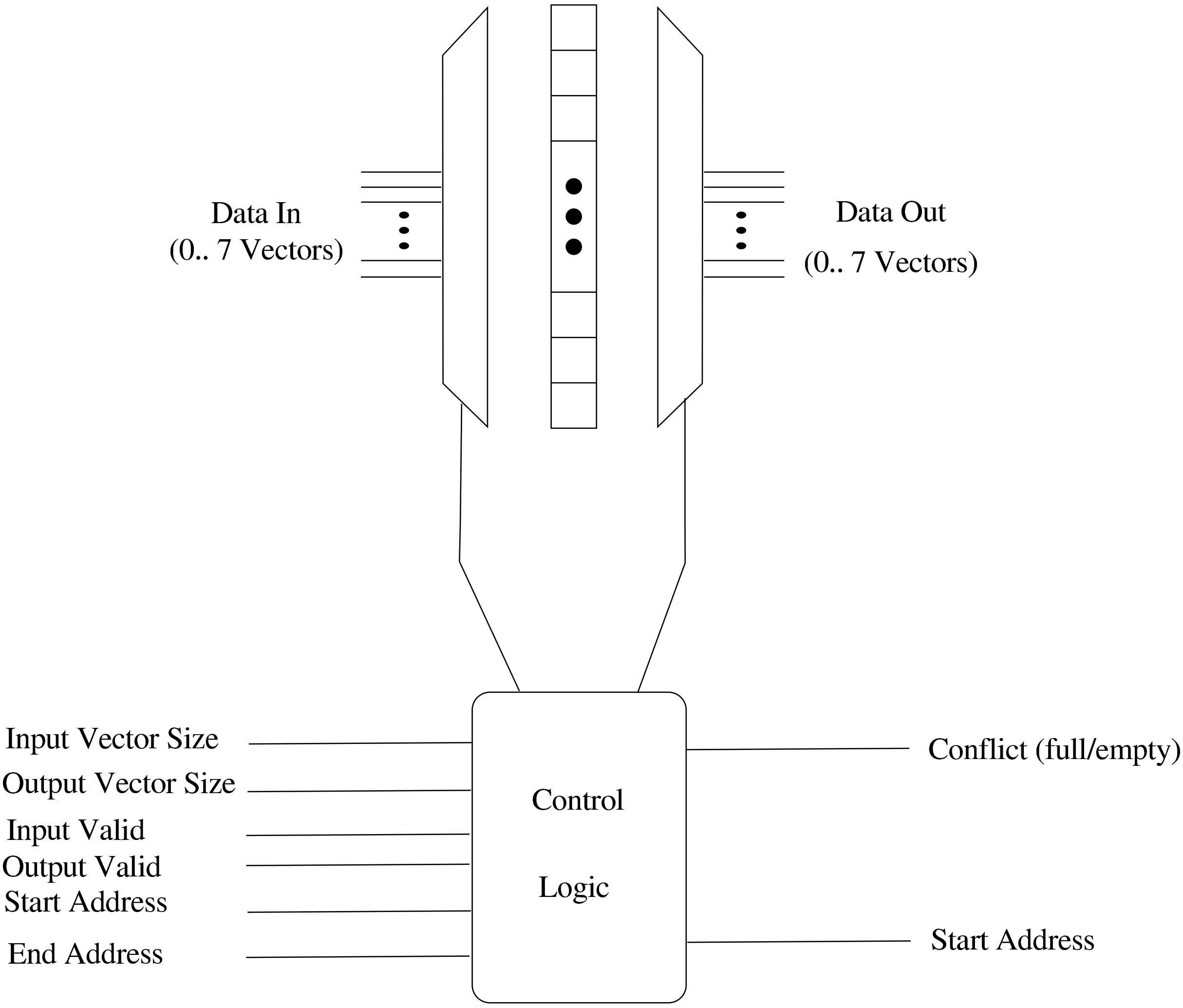

Figure 5.10: The general architecture of the reconfigurable FIFO.

As shown in Figure 5.10, the inputs of the reconfigurable FIFO is composed of an array of $CellsPerRow$ [1] input vectors and a similar sized array of output vectors. These input and output arrays of vectors can be considered as input and output vectors of larger sizes. Through the use of multiplexors, the FIFO's memory registers [2] can be updated (or read) by the input (or output) vectors.

---

[1] this value is generic and can be parameterized in the VHDL model's constants package.

[2] which is also parameterizable.

The needed input and size of the FIFO (depending on the device it is attached to either at the input and output sides) can be configured [3] to range from an array of size one vector to an array of $CellsPerRow$ vectors (maximum size). The input and output array sizes are independent and are configured through the reconfigurable FIFO $InputSize$ and $OutputSize$ inputs respectively.

The problem of updating the proper memory range given the variable sizes of the input is solved by *enabling* according to the $InputSize$ vector only the appropriate number of registers in the portion of the memory the current inputs are directed to discarding thus the *dangling* inputs that are not needed. In addition, the memory address increment is adjusted according to the $InputSize$ vector as well. The proper outputs are read from the FIFO memory also by adjusting the output address increment according to the $OutputSize$ vector and neglecting the uninteresting vectors from the output array of the reconfigurable FIFO.

The reconfigurable FIFO control unit orchestrates the operation of the FIFO enabling constant flow of data in to and out of the reconfigurable FIFO as long as no empty-memory conflict occurs while a read out request is asserted or full-memory conflict occurs while a write in request is asserted. The reconfigurable FIFO memory addressing is managed in a circular manner allowing reading and writing to loop from the end to the beginning of the memory when applicable. The reconfigurable FIFO control unit also controls the read and write address generation counters. Two conflict flags: $FIFO_{Empty}$ and $FIFO_{Full}$ are also generated by the reconfigurable FIFO control unit.

5.4 The HPad Control Architecture

The reconfiguration control architecture of the HPad is the tool enabling the usage of the dynamic reconfiguration features embedded in the HPad DPA discussed above. The HPad control facilities manages both the internal operations of the HPad RTR coprocessor as well as overall operation management including external communications management. The control tasks of the HPad involve the management of the following tasks:

1. RTR operations
2. Background loading of contexts
3. Tasks processing management
4. Operation management of the HPad components
5. Generation and usage of time stamps signals

---

[3] Note that configuration takes place in run time, while parameterization is fixed at compile time.

Items 1, 2, 3 and 4 are carried out in the Reconfiguration Controller discussed in Section 5.4.2. While the generation of the timing stamps is carried out by the Operation Controller of Section 5.4.1.

5.4.1 Operation Control

The Operation Controller is basically a sequencer responsible for generating time sequences. Throughout this thesis we denote the generated sequences "time-stamp signals" since they are mainly used to synchronize the operation of DPA and to aid in the scheduling of the RTR events. The Operation Controller generates $2 \times CellsPerRow$ time stamp bits. Half of the generated bits ($CellsPerRow$) are generated directly by a serial counter counting the clock input. These time stamps are essential for the operation of a wide variety of architectures such as the R2SDF, R2^2SDF and R2MDC of the FFT and other similar algorithms to name a few.

The other half of the generated time-stamp signals are generated via $CellsPerRow$ LUTs in order to generate functions of time stamps for some irregular structures such as some pipelined architectures for the DCT algorithm. The address of the LUTs are the count bits generating thus signals that are functions of time. These function of time signals were not needed for the examples listed in Chapter 6 but they are likely to be needed in applications that involve events at time points that are not a power of 2.

The block digram of the Operation Controller is shown in Figure 5.11. At reconfiguration time the $CellsPerRow$ LUTs are loaded with the specific time functions. In case of operation conflicts, the counting process is frozen to ensure the integrity and correctness of the generated time-stamp signals relative to the operation progress of the HPad coprocessor.

5.4.2 Reconfiguration Control

The heart of the HPad coprocessor control is the reconfiguration controller. The reconfiguration controller manages the reconfiguration and operation of the HPad coprocessor including triggering of its operation, freezing the operation, and accepting new tasks to be processed in the HPad coprocessor.

As illustrated in Figure 5.12, the Reconfiguration Controller inputs the task number which is a pre-defined task number stored in the reconfiguration ROM, the amount o data to be processed and the start and end addresses for both reading and writing in the external memory of the assumed SoC solution.

To keep track of the configuration and/or operation progress a couple of counters are needed. The read and write from and to the outside world conflict flags are passed from the input/output FIFOs. At an occurrence of a conflict the Reconfiguration Controller generates the appropriate signals putting the operation of the HPad coprocessor

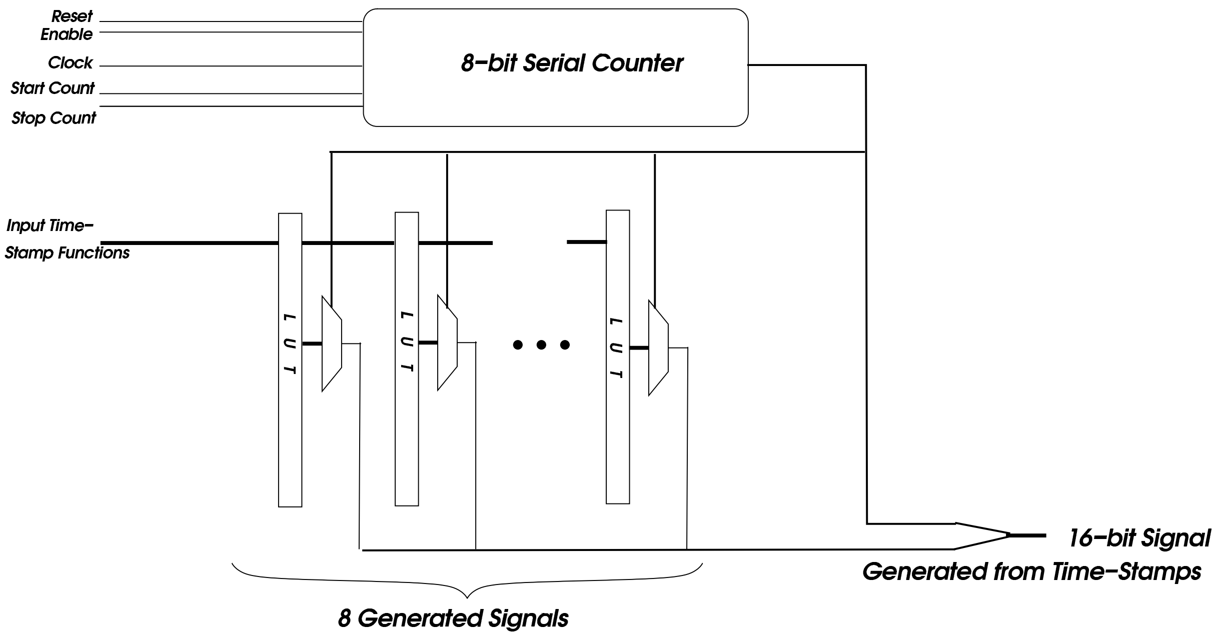

Figure 5.11: General Architecture of the Operation Control unit (with $CellsPerRow =$ 8).

on hold until the conflict is resolved. The input and output data address are supplied to the input and output FIFOs.

The Reconfiguration Controller also takes care of the book keeping of needed and obsolete contexts in the HPad DPA and thus if it can or not load in new configurations and also whether loading of these configuration can be carried out in the background while the HPad is still performing a current operation.

Taking all the above considerations in account, 8 different states of the HPad coprocessor are defined. These states and their transitions are depicted in the state digram in Figure 5.13.

The state diagram of 5.13 shows the following states:

- the idle (*"NOP"*) state,
- the *Conf*iguration (*"Cfg"*) state,
- the *S*ample *N*ew *T*ask (*"SNT"*) state,
- the *"Run"* state,
- the *F*reeze while possibly *C*onfiguring (*"FzC"*) state,
- the *S*ample ne*X*t *T*ask (*"SXT"*) state,

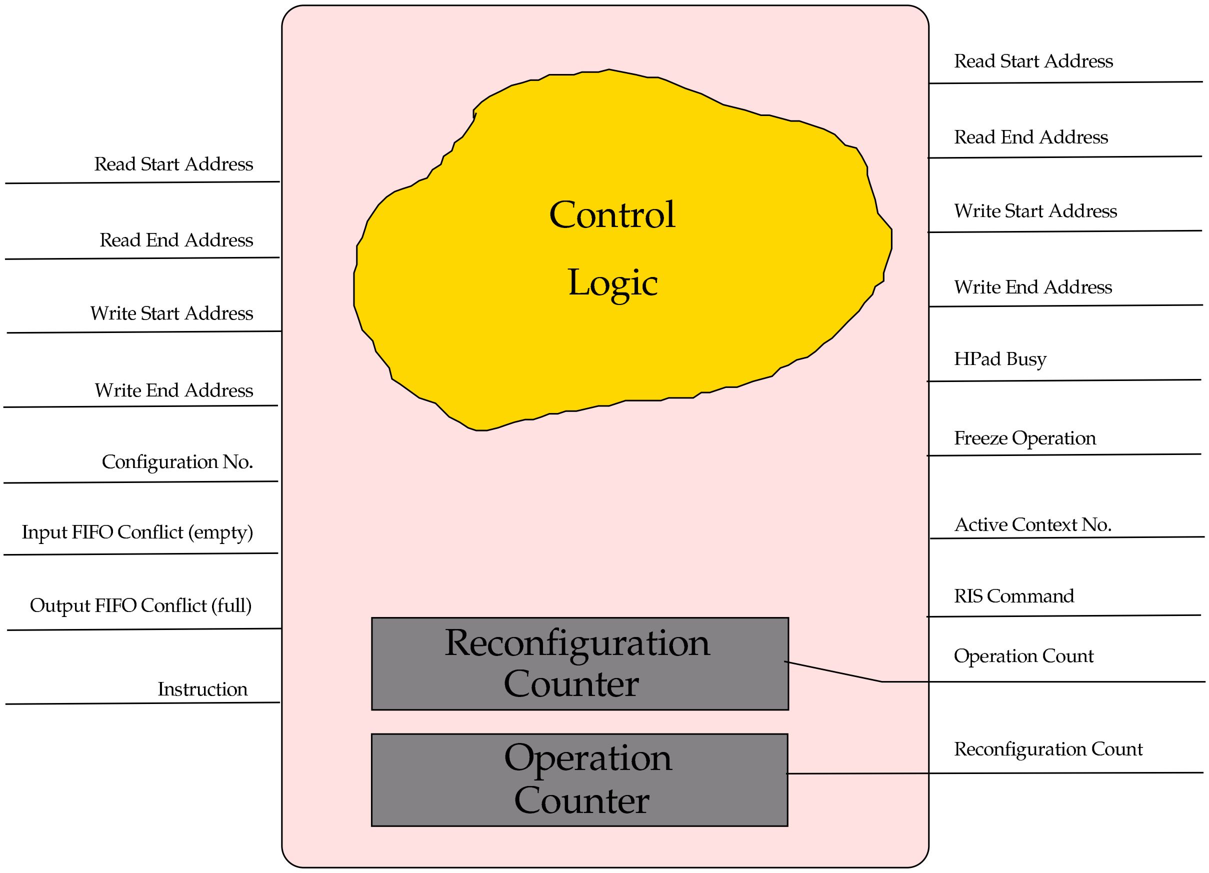

Figure 5.12: The HPad's reconfiguration controller.

- the *R*un while ReConfiguring *P*artially (*"RCP"*) state, and
- the *F*reeze while possibly *P*artially Configuring (*"FzP"*) state.

The transition conditions between the above states is summarized in the following:

The HPad reconfigurable processor starts with the *NOP* state. At the event of a new task request, the Reconfigurable Controller switches to the *SNT* state. After storing the new task information: the task name, the number of data points to be operated on and the read and write beginning and end addresses in the external SoC memory the HPad coprocessor switches to the *Cfg* state.

In the *Cfg* state the appropriate contexts are loaded form the Reconfiguration ROM in the HPad DPA. If a static configuration is to be loaded, only one of the two context registers in each RIS should be updated whilst both context registers should be loaded when an RTR configuration is to be configured. According to the type of configuration i.e. static or dynamic, the Reconfiguration Controller asserts the appropriate RIS

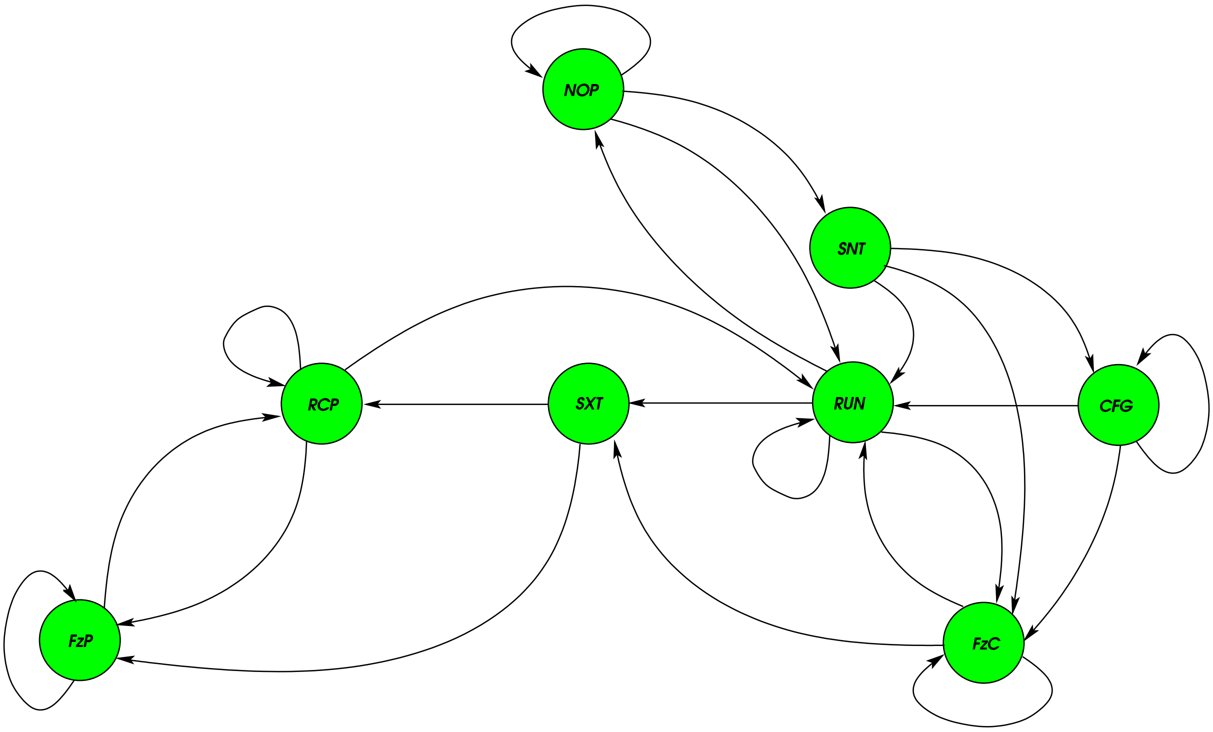

Figure 5.13: The state transition digram of the of the HPad Reconfiguration Controller.

configuration instruction and sustains the *Cfg* state until the contexts are loaded in the proper positions.

Subsequent to the reconfiguration of the HPad DPA the HPad is switched to the *Run* state unless there is a read or write conflict. In case of a conflict the HPad coprocessor switches from the *Cfg* state to the *FzC*. In the *FzC* state the reconfiguration of the HPad DPA continues but it does not transit to the *Run* state unless the read/write conflict is resolved. The *FzC* is also visited directly from the *SNT* state, the *Cfg* state and the *Run* state in case of a read/write conflict.

In the *Run* state, the configured task is executed until the number of data points to be operated on is reached. A special counter keeps track of the data consumption progress. Obviously, at a conflict event, the data consumption progress counter is put on hold so long as the conflict situation is sustained.

When the required amount of data is processed the HPad coprocessor transits back to the *NOP* state. The Reconfiguration Controller remembers the type of loaded configuration whether *complete* for RTR architectures implementations, or *partial* for static architectures implementations. The reconfiguration controller remembers as well as the execution progress of the stored contexts. In case of vacancy in the context registers, the HPad coprocessor can accept a new task. While in the *Run* state, and at the event of a new task request, if there is vacancy for a new task, [4] the HPad coprocessor shifts to the*SXT* state.

The *SXT* state can be directly visited from the *FzC* state if a new task request is placed. From the *SXT* state the HPad coprocessor transits to the *RCP* state where the operation of the current task continues as the loading of the new tasks contexts takes place. Again, in case of a read/write conflict, the HPad switches from the*SXT* state to the *FzP* where the partial configuration is carried out while the current task execution is frozen.

When the read/write conflict is resolved the HPad coprocessor turns to the *RCP* state.

After the the partial configuration is concluded the HPad coprocessor returns to the *Run* state.

When the current task computations are accomplished, the HPad switches back to th *NOP* state. The HPad coprocessor remains in the *NOP* state unless either there is a new task request or if the second reconfigured task is yet to be attained to. If the second task is yet to be undertaken the HPad controller updates the current context number and switches directly to the *Run* state and so on.

5.5 The HPad Reconfiguration Mechanisms

In this Section the reconfiguration mechanisms of the HPad are summarized. The HPad has *three* modes of reconfigurations:

1. *Complete HPad reconfiguration:*
   This type of reconfiguration is the slowest. Implementations involving RTR operations reconfiguration in the HPad are always of this type because they require the usage of contexts stored in both the context registers in each RIS. Therefore, this type of configurations always takes place in the foreground and can not take place in the background. Conceptually, by adding more context registers implementations involving RTR operations can be also carried out in the background. However, because of the additional overhead in the sizes of the RIS and reconfiguration control we choose to live with this complete reconfiguration latency if we managed to keep it in an acceptable range. The HPad DPA was therefore organized and interconnected to minimize the latency caused by reconfiguration.

   The RIS are cascaded one after the other in row basis i.e. each RIS context input comes from the left adjacent RIS and the left most RIS of each row get their configurations directly from the reconfiguration control unit. The reconfiguration control unit passes the proper contexts from the reconfiguration contexts ROM.

---

[4] both configurations should be static

The above topology guarantees low latencies for loading the complete HPad data path processing array with new configurations. In this case, both context registers need to be updated and thus the latency is doubled to $Latency_{Dyn} = 2 \times CellsPerRow \ \ (cycles)$.

The above reconfiguration technique is offers a fast reconfiguration scheme considering, for example, an 8 by 8 HPad array that is good enough for a 16 point FFT as will be shown in Chapter 6. Here, only 16 cycles are needed to fully configure the RTR R2SDF architecture.

2. *Background reconfiguration:*
   Background reconfiguration can take place in the HPad when both the current and next configurations are static with no RTR requirements. Obviously, due to the organization discussed above, the reconfiguration latency of background reconfiguration (and static implementation reconfiguration) is expected to be shorter than that of the complete reconfiguration.

   In case of static configurations, only one context register per RIS needs to be updated. As the new contexts are shifted from one RIS to the other the total latency will thus be $Latency_{Stat} = CellsPerRow \ \ (cycles)$.

   The above topology is offers again fairly short background reconfiguration latency considering, for example, an 8 by 8 HPad array that is good enough for a 32-tap FIR filter. For the FIR filter which needs a static configuration only 8 cycles are required to carryout loading of the reconfiguration contexts which can take place in the background while an other static task is being executed.

3. *RTR:*
   When an RTR implementation is loaded in the HPad DPA, RTR operations is triggered at the occurrence of the selected reconfiguration event. The reconfiguration happen in only *one* cycle since the RIS selects between the context registers.

Complete reconfiguration is managed by the reconfiguration control unit. The computation of the HPad DPA is frozen until loading of contexts in the HPad DPA is completed. Background reconfiguration is also managed by the reconfiguration control unit. The computation of the HPad DPA continues during the loading of contexts in the HPad DPA operation.

In contrast to complete and background reconfigurations, RTR involves no loading of reconfiguration contexts. The decision of RTR action is taken locally at each RIS without the intervention of neither the reconfiguration control unit nor the operation control unit. The operation controller, however, makes time stamps and other timing signals available to the RIS units to aid them in deciding on the context registers to choose. In case of the static implementations, RTR is disabled by setting the MSBs of the contexts to 00 and in this case the number of active context register is supplied by the reconfiguration control unit.

5.6 The HPad Routing Topology

In this Section the routing topology of the HPad DPA is summarized. One of the targets of the organization of the HPad routing topology is to tailor it to fit common DSP algorithms' architectures while simplifying the routing effort as much as possible. This was achieved by recognizing that most of the needed routing will be between neighboring elements in a data flow manner i.e. with the data flowing mostly in one direction. This obviously reduces the number of inputs to each processing element which reduces in tern the reconfiguration overheads (area, power consumption and size of the reconfiguration contexts).

As mentioned above, each processing element has direct access to the processing elements of the neighboring cells at its left provided they are in the range from two rows above to one row below. Each processing element can also sample data from the four buses surrounding it (above and bottom horizontal and left and right vertical DSBs).

To support butterfly connections and efficient usage of both types of processing elements connections of the top and bottom processing elements are made available. This was proven beneficial in some of our experimentations.

In addition to unifying the direction of connections from the column on the left to that at its right and so forth, feedback provisions have also to be made. For this both horizontal and vertical buses are provided. These buses save also the purpose of broadcasting data or passing them to far away parts of the HPad DPA.

An external inputs per row is available for use. Any of the cells of the HPad can have access to an arbitrary external input through buses (that have access to the external input coinciding with its number) provided that the needed buses are available.

Moreover, to maximize the usability of the HPad DPA, feedbacks from the end of the array can be selected at the right side of the array by choosing between them and the external inputs. Connections to the above and bottom rows are also made circular in case the connected cell is close to the top edge or at the bottom edge of the HPad DPA. Figure 5.14 summarizes the routing possibilities of the HPad DPA.

An important problem to be solved is finding a feasible solution how to make use of the grand processing power of the HPad DPA. Consider the current implemented size of the HPad DPA of 8×8 which is suitable for implementations algorithms of realistic sizes as is discussed in Chapter 6. The number of elements of such an array is $8 \times 8 = 64$ GALPE units, and also $8 \times 8 = 64$ MeMPE units, as well as 8 horizontal buses and 8 vertical buses. This makes up a total of 144 elements each with an output [5] that may be needed to be picked as an HPad output. Obviously, this number is not realistic for practical usage in a dynamically reconfigurable system. Therefore, sampling outputs of the HPad array is restricted to the outputs of GALPE and MeMPE units of the last

---

[5] each of the outputs is composed of a vector and a single bit outputs

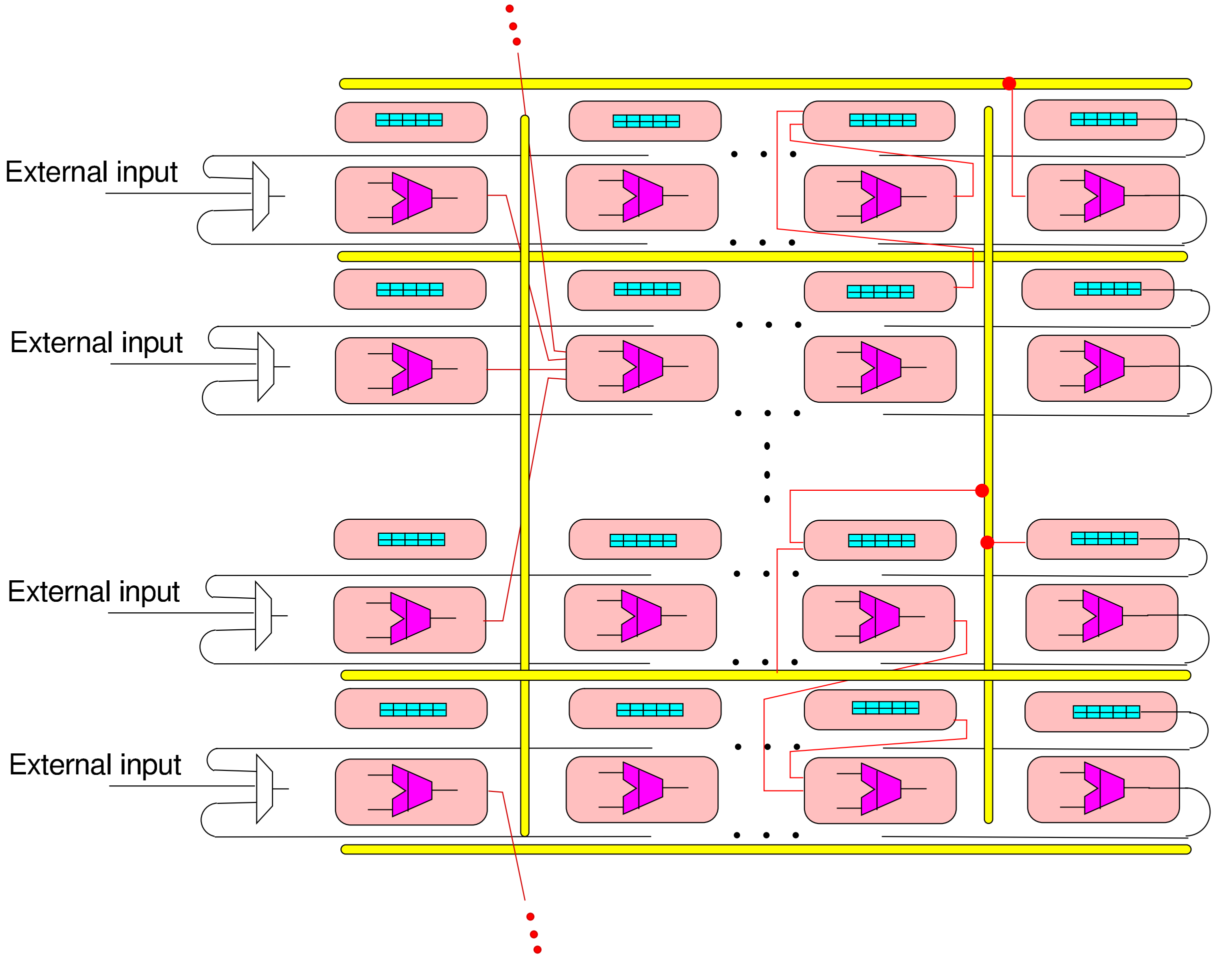

Figure 5.14: A simplified illustration showing the routing topology of the HPad DPA.

column in the HPad DPA as well as all the horizontal and vertical buses which adds up to $4 \times 8 = 32$ outputs.

Now after having selected 32 outputs from the HPad array, further reduction of the output size is achieved by selecting 1 output out of 4 thus having now only 8 outputs from the HPad DPA. This number of outputs is not only practical but moreover, allows scaling the size of the HPad coprocessor by cascading several of the HPad DPAs since the number of inputs and the number outputs of the HPad DPA are the in this case same.

5.7 Synthesis Results

As was mentioned in Section 5.2 the HPad VHDL model was written in a parameterizable manner allowing the generation of different designs by simply changing a few constants in the constants package.

The current implementation of the HPad, however, has its parameters set set according to Table 5.1 below:

Table 5.1: Current HPad parameters.

Parameter	Value
Array width	8
Data vector size (bits)	16
Size of vector storage in the MeMPE units	2
Number of context word registers in each RIS unit	2
Reconfigurable FIFOs input size (bits)	Array width × Data vector size
Reconfigurable FIFOs output size (bits)	Array width × Data vector size

The HPad was designed in RTL level VHDL code. Over 6500 lines of code was written for the HPad. The HPad VHDL RTL model was synthesized using the UMC $0.25\mu m$ technology with Synopsys Design Analyzer. Table 5.2 below summarizes these synthesis area and timing results. No timing or area constraints were set at synthesis time i. e. the following results are relaxed.

Table 5.2: Area and Propagation delay reports for the HPad synthesized using a $0.25\mu m$ technology.

Block name	Area (mm^2)	Timing (ns)
GALPE	0.074	12.97
GALPE RIS	0.116	25.14
MeMPE	0.030	1.78
MeMPE RIS	0.062	7.23
DSB	0.050	2.41
DSB RIS	0.064	1.04
HPAD DPA	12.531	====
OpController	0.040	2.04
ReconfController	0.048	5.04
ReconfFIFO	0.899	7.96
RTR_HPadTop	14.543	====

The timing results for the HPad DPA and the top RTR_HPadTop are not defined because they depend on the clocking option of the GALPE units and the routing as well.

5.8 Area Efficiency Analysis

5.8.1 Block wise Area Efficiency

Since all PEs in the HPad DPA are surrounded by RIS blocks which mange reconfiguration and routing, the area efficiency of each processing element can be expressed by:

$$\eta_{PE\_area} = \frac{Area_{PE}}{Area_{RIS}} \times 100. \tag{5.1}$$

Nevertheless, the above measure of area efficiency is not really expressive. This is because each PE has additional resource enabling it to perform several functions. To have a better measure of PE area efficiency we synthesized an experimental PE performing all the possible operations of the actual PEs in parallel. The ratio between the area of this experimental PE and that of the actual PE can express the effective area efficiency since it compares between two elements of the same processing power indicating thus the multiple functional flexibility penalty effect. The experimental PE is obviously not usable since the address of the useful output is not known at fabrication time and thus can not be routed.

Table 5.3 shows the synthesis results of the experimental arithmetic/logic and the memory manipulations processing elements.

Table 5.3: Area and Propagation delay reports for the experimental PEs used to find the minimum theoretical area for area efficiency analysis synthesized using a $0.25\mu m$ technology.

Block name	Area (mm^2)	Timing (ns)
Arithmetic/logic unit	0.071	===
Memory unit	0.814	0.29

The effective area efficiency can then be computed by:

$$\eta_{PE\_Eff\_area} = \frac{EffNonReconfArea_{PE}}{Area_{RIS}} \times 100. \tag{5.2}$$

From the Equation 5.2 and Tables 5.3 and 5.2 above we compute the effective area efficiency of the GALPE unit to be $\eta_{GALPE\_Eff\_area} = \frac{NonReconfPE}{Area_{GALPE\_RIS}} \times 100 = 61.37\%$ while the naive measure of Equation 5.1 gives an area efficiency measure of 63.55%.

The experimental memory manipulation units were chosen to be only memory providing thus an absolute reference for us to the minimum storage area of the same type provided by the technology. Accordingly, applying Equation 5.2 for the MeMPE units we compute $\eta_{MeMPE\_Eff\_area} = 13.13\%$ while Equation 5.1 results in an area efficency of 47.61%.

5.8.2 Overall Area Efficiency

As was discussed above, the HPad coprocessor consists of the HPad DPA, the controllers, the input/output FIFOs and the reconfiguration ROM. Obviously, all computations take place in the HPad DPA. Hence the ratio between the area of the HPad DPA and the rest of the HPad coprocessor can be considered as a measure of the area efficiency of the RTR HPad coprocessor solution. From here the data path to system area efficiency is given by

$$\eta_{Sys\_area} = \frac{Area_{DPA}}{Area_{Sys}} \times 100. \tag{5.3}$$

The HPad DPA on the other hand is not purely a data path unit, but as a matter of fact it possesses a number of routing and reconfiguration resources. The DSBs present reconfigurable routing resources and all types of RIS units present both reconfiguration and routing resources. The net effective HPad area efficiency is therefore given by:

$$\eta_{NetDPA\_area} = \frac{EffectivePE\_Area_{PE}}{Area_{HPad}} \times 100. \tag{5.4}$$

From Equation 5.4 above we get: $\eta_{NetDPA\_area} = 40.26\%$ and from Equation 5.3 which is an optimistic figure we get $\eta_{DPA\_area} = 52.63\%$.[6]

Furthermore the finally configured HPad coprocessor does not utilize 100% of the different processing elements of the HPad not to mention complete computational power of the processing elements. From here we can compute the actual application area efficiency according to:

$$\eta_{App\_area} = \frac{No_{UsedPEs}}{No_{PE\_total}} \times \eta_{NetDPA\_area} \times 100. \tag{5.5}$$

5.9 Scalability

The HPad DPA can be scaled to form larger or smaller array by scaling the DSBs. As will be shown in Chapter 6 an HPad of size 8×8 is a realistic size suitable for common

---

[6] These efficiency figures do not take into account the FIFOs and context ROM areas, since their sizes are arbitrary and are platform dependant.

DSP algorithms sizes. It will also be shown that the use of single global horizontal and vertical DSBs spanning the whole HPad-DPA suffices for the implementation of the studied algorithms.

From an architectural point of view (not considering wire delays), the use of global DSBs is better to using segmented DSBs for the following reasons:

1. *Global DSBs are less expensive in terms of area:*
   As shown in the simplified illustration of the segmented DSB implementation in Figure 5.15, a segmented DSB needs a pair of additional inputs to be able to transfer data from DSBs at both sides. In that case the number of multiplexers needed is given by

$$\begin{aligned} SegDSB_{MuxNr} &= 3 \times Segments_{Nr} \\ &= \frac{3}{2} \times Nr_{Inputs}, \qquad Nr_{Inputs} = 2, 4, 6... \end{aligned}$$

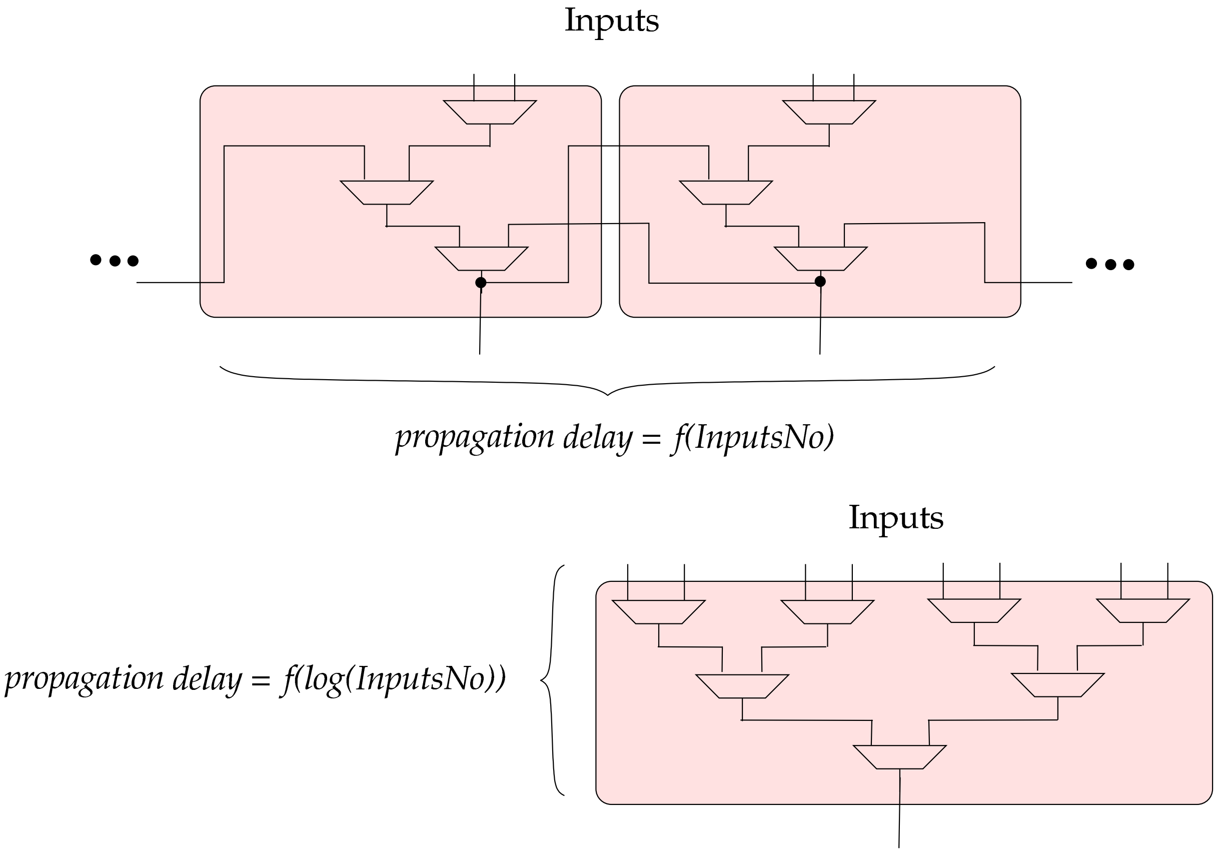
Inputs
•••
•••
propagation delay = f(InputsNo)
Inputs
propagation delay = f(log(InputsNo))

Figure 5.15: Global v.s segmented bus structures.

while the number of multiplexers of the global DSB is given by $GlobDSB_{MuxNr} = Nr_{Inputs} - 1$. The number of multiplexers of the segmented DSB in terms of that of the global DSB is therefore,

$$SegDSB_{MuxNr} = \frac{3}{2}(GlobDSB_{MuxNr} + 1),$$

which is clearly larger than that of the global DSB.

2. Global DSBs offers shorter propagation delay times:
The global DSB propagation delay is:

$$GlobDSB_{PropDel} = Mux_{Delay} \lceil \log_2(Nr_{Inputs}) \rceil.$$

The maximum segmented DSB, however, propagation delay is:

$$\begin{aligned} SegDSB_{PropDel} &= 2 \times Mux_{Delay} \times Stages_{Nr} \\ &= 2 \times Mux_{Delay} \times \frac{Nr_{Inputs}}{2}, \qquad Nr_{Inputs} = 2, 4, ... \\ &= Mux_{Delay} \times Nr_{Inputs} \end{aligned}$$

This delay is proportional to the array size and thus is ill flavored and inappropriate for usage. Moreover, the resultant delay is considerable larger than that of the global DSB.

3. *Registering the outputs of the segmented DSBs may also lead undesirable results:*
As we discussed in Chapter 4 registering might result in undesirable delays that might break timing loops. Additionally, pipelining segmented DSBs will cause unsuitable and irregular latencies in data transfers which may be problematic when synchronizing data.

The obvious disadvantage of using global DSBs is that the DSB spanning the complete length or width of the HPad is exhausted when used for only one data transfer. This problem can be solved by using additional DSBs per row or column. However, as will be discussed in Chapter 6 investing in one DSB per row and one per column was adequate for the studied algorithms. According to the above discussion, we chose the Global bus configuration for the HPad DPA[7]

Due to having the same number of inputs and outputs of the HPad data path array, several HPad DPA units can be cascaded by connecting the outputs of preceding HPad DPA nits to the succeeding ones. Clearly, the size of the configuration memory needed to configure the larger structure is proportional to the number of HPad DPA units used.

---

[7] The above discussion does not take into account the propagation line delays that are non linear and also function of the fabrication technology.

This increase in the configuration memory size may likewise result in also reconfiguration times proportional to the number of HPad DPA units. Reconfiguration control may turn be more complicated too because of the need of management of longer reconfigurations.

To work around this problem we split the reconfiguration memory with each portion attached to its corresponding HPad DPA as suggested in Figure 5.16. This way, the the increase in the HPad coprocessor is linear with the number of HPad DPAs implemented[8], both controllers of the HPad coprocessor are intact and so is the case with input and output FIFOs.

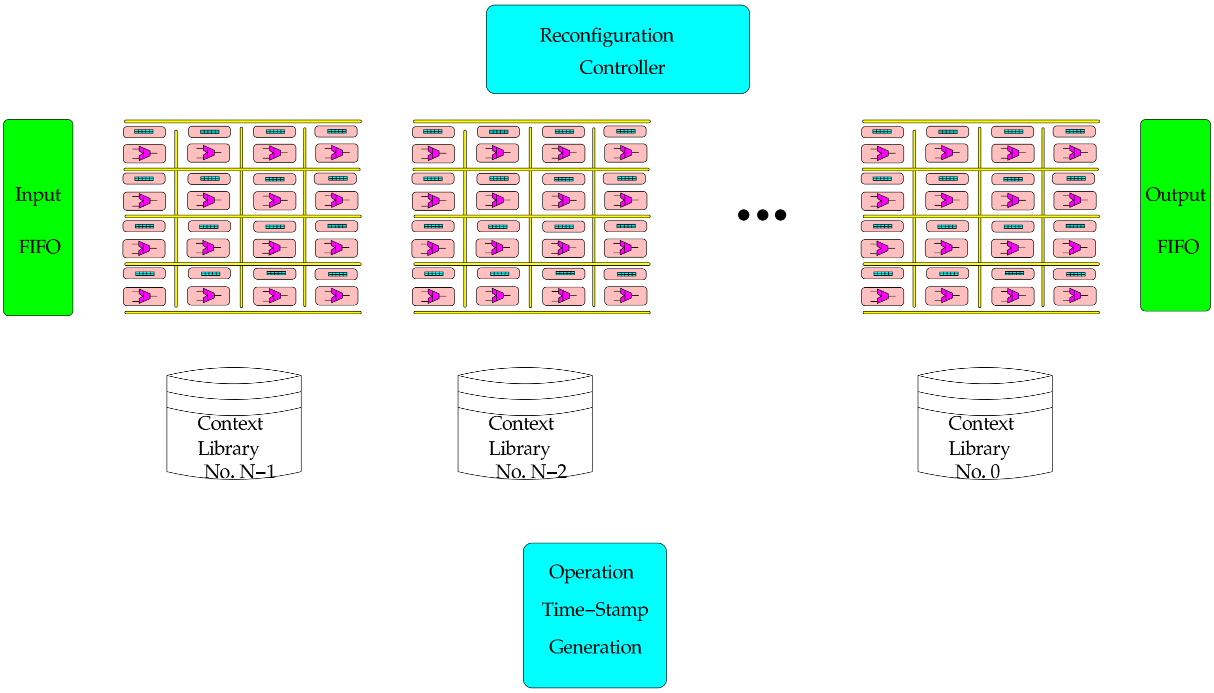

Figure 5.16: Scalability of the HPad.

The only disadvantage of this scaling approach is that it can result in *longer* but not *wider* reconfigurable data path processing arrays. This is because installing additional HPad data path array rows will result in the need of using In (or Out) FIFOs of wider outputs (or inputs) which is not scalable. However, because of the pipelined nature of most DSP algorithms implementations the number of the HPad data path processing array rows is thought to have minimal or no effect especially considering the availability of row data sharing buses which can help in transferring input data to any of the HPad DPA units.

---

[8] the reconfiguration memory sizes is reduced since now they are addressed independently

5.10 Concluding Remarks

A practical CGRC solution is to be operated within in a SoC containing other processing subsystems such as μps, ASICs and FG FPGAs. We assume a library based computational paradigm where a ROM library storing various configurations charges the proposed CGRC HPad solution with new contexts. This ROM library can be at fabrication time realized as any type of non volatile memory: EPROM, EEPROM, FLASH etc. The HPad is also assumed to receive and deliver data through a shared data bus. Therefore, the HPad feature integrated reconfigurable FIFOs and control data paths. In this context, the HPad control can turn the complete HPad computational units to a *freeze* mode upon the occurrence of an input/output conflict. In addition, the HPad is capable of RTR operation as well partial and background reconfiguration. The HPad is composed of a DPA, a couple of Reconfigurable FIFOs for input and output, a Reconfiguration Controller and an Operation Controller.

The DPA is a heterogeneous array of different types of PEs. The PEs are of two types: GALPE for arithmetic and logic operations, MeMPE for memory manipulation operations such as RAM/FIFOs. Most routing in the HPad is local between PEs although long distance routing can be achieved through the global DSB elements. To support implementation of structures that include feedback loops optional registering of the PEs outputs is provided.

Both the function and routing of each PE are possible to change dynamically at run time. Partial and run-time well as multi-context reconfiguration are supported by the HPad. This is facilitated by RIS units encompassing each of the aforementioned elements of the DPA.

The reconfigurable FIFOs can be configured with a range of input and output data sizes. This enables them to adapt to different data bus widths as well the delivery (or consumption) of data from the HPad DPA (depending on the implemented algorithm). In case of input/output conflict, the reconfigurable FIFOs generate flags that are read by the Reconfiguration Controller which manages the HPad accordingly.

The Reconfiguration Controller provides the essential interfacing control of the HPad with the external world. The HPad only needs bus availability information to operate, minimizing thus the effort of integrating the HPad with the rest of the SoC. the Reconfiguration Controller also manages full and partial reconfiguration of the HPad. To do that the Reconfiguration Controller takes care of the book keeping of fresh and obsolete configuration contexts. In a case of a vacant space in the configuration contexts registers, background recognition can take place while the HPad DPA is crunching numbers for the current active context. To tag a context fresh or obsolete, the Recognition Controller also keeps track of the progress of computation.

The Operation Controller generates time stamp signals to aid in RTR of the HPad DPA. The Operation Controller is based on a counter and LUTs to generate time stamps and time stamp functions. These signals as well as locally generated event flags are managed within each RIS unit of the HPad DPA according to the active context to carry out a predefined RTR action.

Chapter 6

Implementation Examples and Validation

Contents

6.1 Basic Operations **154**
    6.1.1 Larger Vector Operations 154
    6.1.2 Complex Operations 154
    6.1.3 FIFO Realizations 155
    6.1.4 Butterfly Operations 156
6.2 FIR Filter Realizations **160**
    6.2.1 TDF FIR Filter Mapping 160
    6.2.2 DF FIR Filter Mapping 161
6.3 FFT Realization **163**
6.4 DCT Realizations **165**
6.5 Viterbi Decoding Realization **166**
6.6 Concluding Remarks **168**

In the early Chapters of this thesis, general knowledge about reconfigurable computing and several DSP algorithms architectures were presented. This led us to the extraction of needed features of the targeted RTR architecture in Chapter 4. In the previous Chapter the architecture of the proposed RTR HPad was presented.

In this Chapter, we analytically demonstrate how can several of the discussed DSP algorithms' architectures be realized: mapped, routed and operated on the HPad. In The first section we discuss the mapping possibilities of some the basic blocks such as larger vector operations, basic complex operations, FIFO and butterfly realizations that are important for complete algorithm implementations. The later sections describe the realizations of FIR, FFT, DCT and Viterbi decoder. This Chapter is finally concluded in Section 6.6.

6.1 Basic Operations

The very basic arithmetic and logic operations supported by the HPad were already included in the instructions of the GALPE and MeMPE units. The realization, however, of more complex functions used repetitively in various DSP algorithms' architectures are constructed by utilization of several of the HPad coprocessor components. In this Section we demonstrate how some of the basic functions or *MacroFunctions* can be realized by the HPad. Some of these MacroFunctions such as larger vector sizes operations and complex operations involve usage and routing of more than one of the HPad DPA path array units together. Some such as the *Dynamic Butterfly* operation involve the usage of several of the Pad data path array units along with some of the operation control resources to facilitate RTR operations. Throughout this Section we will discuss the implementations of larger vector operations, complex operations, FIFO and ROM realization and the Dynamic Butterfly operations.

6.1.1 Larger Vector Operations

Almost all logic functions of larger vectors can be constructed by simply dividing the large input vector (or vectors) to two or more standard size vectors and carrying out the logic operations on them. However, this will not suffice for some logic operations such as shift and rotate operations. In that case, the bit-in and bit-out of the processing elements should be inter-routed such as the proper results are achieved. Such an implementation is within reach given the routing resources of the HPad DPA. Easier and more efficient implementation is a straight-forward task when utilizing MeMPE units. Recall that the MeMPE units can shift a bit recursively within its vector array.

The addition operation of larger-sized vectors can be obviously implemented by routing the carry out (bit-out) signal from the GALPE unit operating on the *L*east-*S*ignificant (*LS*) portion of the input vectors to the carry in (bit-in) GALPE unit processing for the *M*ost-*S*ignificant (*MS*) portion of the input vectors.

6.1.2 Complex Operations

Complex addition and multiplication operations can also be easily implemented on the HPad DPA.

The addition/subtraction operation can be constructed by using two lines routing the real and imaginary addition/subtraction separately.

The complex multiplication, however, is a little bit more complicated. The complex multiplication operation is given by the following well known formula:

$$(A_r + jA_i)(B_r + B_i) = (A_rB_r - A_iB_i) + j(A_iB_r + A_rB_i) \tag{6.1}$$

The implementation of Equation 6.1 can be directly realized as shown in Many of the Figures later in this Chapter. Again here we find that the routing resources and the GALPE units organization suffice and can efficiently realize the complex multiplication operation given by Equation 6.1.

6.1.3 FIFO Realizations

The MeMPE units are designed to efficiently implement ROMs and FIFOs that are common parts of the architectures of various DSP algorithms. ROMs can be used to statically store constants such as the FIR filter factors. In such a case the proper constant is stored in a MeMPE unit oriented in the vicinity of the GALPE unit needing to use this factor. In that case, the MeMPE unit is configured such that it selects from its array of registers the stored vector and thus only one vector register of the MeMPE array of registers is used.

An other possibility is to use one or more MeMPE units to store several constants for some pipeline architectures such that one of them is used at each clock cycle. In such a configuration, a number of MeMPE units can be used to store these constants and the proper vector is passed to the output at the correct instant of time. Possible implementations for such a scheme are shown in several of the Figures of the FFT, DCT and Viterbi decoder mapping examples below.

A configuration good for storing 4 constants is used in the FFT implementation discussed below. To dynamically select the proper input to the GALPE unit, the GALPE unit changes its input from MeMPE unit 1 to MeMPE unit 2 every other cycle, while both of the MeMPE units change their output between both the stored vectors every cycles. An other possibility illustrated in the Figures of FFT, DCT implementations in Sections 6.4 and 6.4 There, several MeMPE units are interconnected such as the vectors are rotating between them by shifting-in vectors from one MeMPE unit to the other and at the end routing the last vector to be shifted in the first unit again. The GALPE unit samples as input the routed vector from the bus. By this configuration large rotating output ROMs can be constructed.

Configuration of FIFOs is very much similar to ROMs. FIFOs of depths ranging from one vector to several can be constructed by shifting in the inputs from one side of the *configured* FIFO unit and the out put can be *read* from the other end.

6.1.4 Butterfly Operations

Butterfly structure present the basic component in architectures of many of the studied DSP algorithms. The butterfly operation can be either dynamic or static. Static butterfly structures are mostly used in parallel implementations. The routing topology of the HPad DPA permits direct butterfly routing of data placed 4 rows apart with no need for DSB usage.

Dynamic butterfly structure, however, are not very trivial to implement because of the run-time switching between direct passing of data to or carrying out the butterfly operation as described previously in Chapter 3. The implementation of dynamic butterfly operation require the observation of the following considerations:

- Control of the dynamic switching,
- The availability of routing lines for the inputs and
- The availability of routing lines for the outputs

Each configured dynamic butterfly unit has one output rather than two and is closely coupled to a FIFOs unit to schedule the operation of the streaming data as shown in Figure 3.15. The availability of local FIFOs for “juggling” of input and output data is essential to the dynamic butterfly operation. This is clearly an advantage of having MeMPE units placed in the vicinity of GALPE units making them easily usable as FIFOs.

The configuration of dynamic butterfly units with 1, 2, 4 and 8 FIFO units depths are shown in Figure 6.8 within the FFT implementation. As shown in the Figure, no DSB are needed to realize the dynamic butterfly units for FIFO sizes less than or equal to 4, while DSB are needed to realize larger butterfly structures. Dynamic switching between inputs is triggered by the time-stamp flags asserted by the operation control unit.

6.1.4.1 Dynamic Butterfly with FIFO of size 2 Configuration

To demonstrate how a dynamic butterfly can be realized in the HPad, consider the following brief example: Consider a simple dynamic butterfly with a FIFO of 2 data vectors depth. The basic operation of the butterfly is shown in Figure 6.1.

According to the control, the input data vector is either shifted in the FIFO wile the oldest stored data in the FIFO is passed to the output, or with the aid of two PEs (with PE A performs the $+$ operation and PE B performs the $-$ operation), the input data is operated on with the oldest of the stored value in the FIFO while the output of the bottom PE is shifted in the FIFO and the top PE is produced as output.

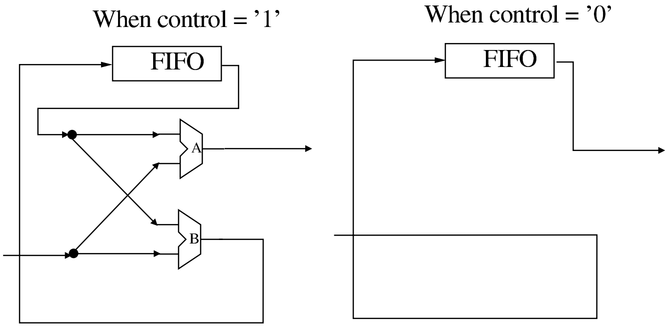

Figure 6.1: The dynamic butterfly operation.

As such the implementation of such a structure on the HPad's DPA can be implemented as shown in Figure 6.2. Here, as illustrated in Figure 6.2 the FIFO is implemented using the MeMPE unit and the two PEs are the GALPEs. The MeMPE unit is chosen to be the one between the GALPE units to be accessible by both the GALPE units as the HPad's DPA routing facilities dictate. A multiplexor indicates the chosen routing according to the control signal coming from the operational controller. The top GALPE unit in Figure 6.2 function toggles also between the function of the PE A operation and data transfer operation.

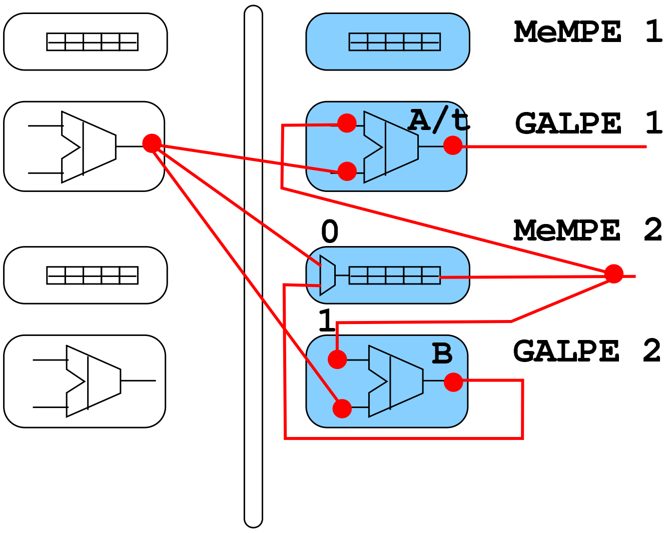

Figure 6.2: The dynamic butterfly operation.

When the selected time stamp (control) signal is 0, the input is directed to the FIFO's input and the output of the FIFO is transfered to the output of the dynamic butterfly. The GALPE A unit –which passes the output of the dynamic butterfly– is at this stage only transferring the output of the MeMPE unit (FIFO). At the other half of the operation cycle the selected time stamp signal from the Operational controller will change forcing the GALPE A unit to operate as an adder and produce the output of the dynamic butterfly. Meanwhile, the MeMPE unit (FIFO) selects now the its input from the bottom GALPE unit that is acts as a subtracter operating on the FIFO output and the external butterfly input.

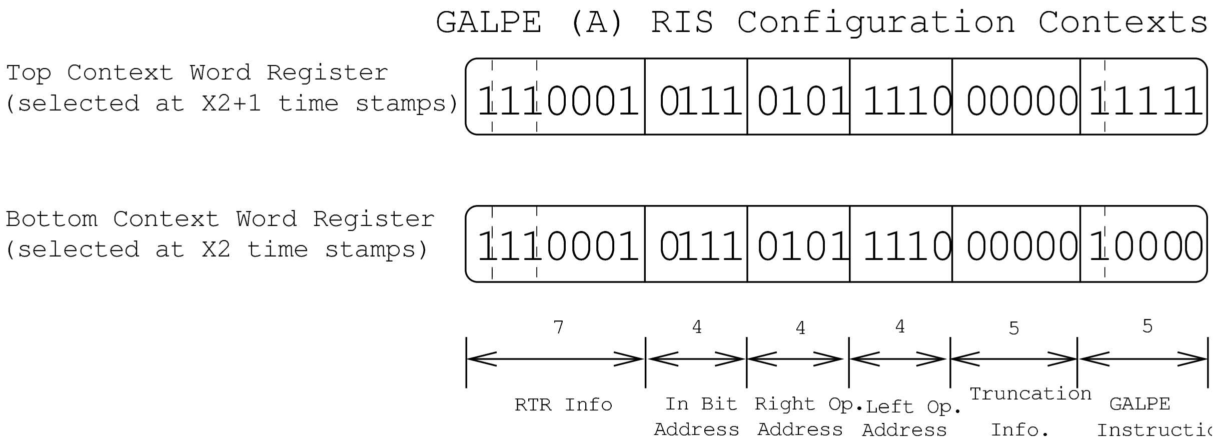

Figure 6.3: Contxt words of the GALPE A unit of Figure 6.2.

In Figures 6.3, 6.4 and 6.5 the corresponding contexts of both the GALPE units as well as the MeMPE units are shown. The top and bottom contexts are selected alternatively according to the selected time stamp signal.

Note from Figure 6.3 that all 7 MSBs of the 6 contexts are the same indicating that they are subject to the same RTR conditions. The MSB of these 7 bits is one to choose synchronous RTR operation with the aid of the Operational Controller's time stamp signals. The 2 bits on its left do not matter in that case, they are chosen to be ones. The 4 LSBs of the RTR info. 7 bits are set to 0001 to select the time stamp signal number 1 being toggled every two cycles. This is the proper trigger to be used with the 2 vectors deep dynamic butterfly.

As shown in Figure 6.2 the routing of the GALPE A unit remains the same even in both cycles of its operation, however, its function changes. Therefore, GALPE A's context word are the same except for the GALPE A's instruction which occupy the 5 LSB position of the context word. The MSB of these 5 instruction word indicates whether the output needs to be register or not. Here we choose to register the output setting this bit to 1 for both cases to be consistent and correct. In the first cycle of operation, the GALPE A unit task is only to transfer its X input out and thus the corresponding instruction is chosen according to Table B.2 to be 1111 as is depicted in Figure 6.3. The second cycle of operation, the GALPE A unit performs act as an adder with 0000 as its instruction.

The input routing addresses of the GALPE A unit are also shown in Figure 6.3. There the X operand addresses are selected to be 0101 since in that case, the input comes from the adjacent GALPE unit on the left. The Y input address is set to 1110 to select the butterfly connection as to the MeMPE unit below.

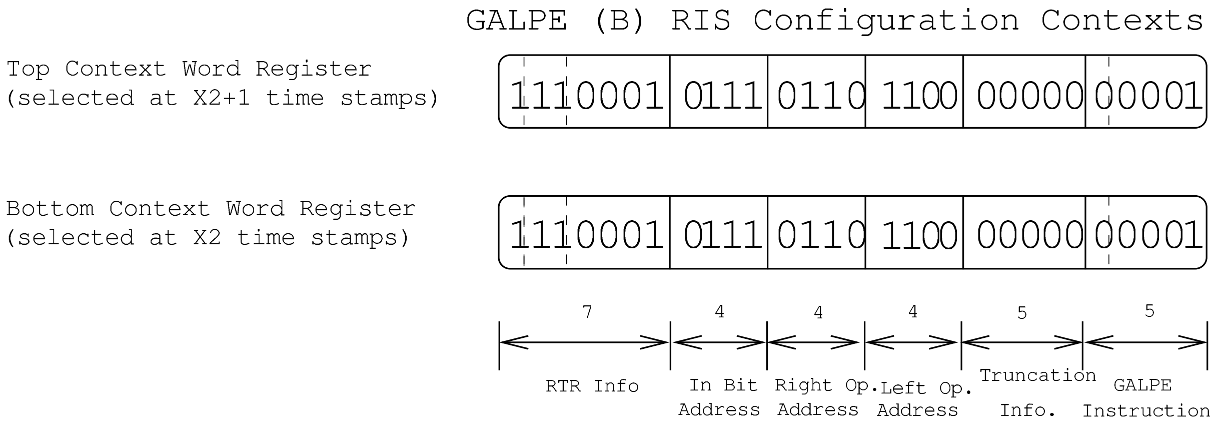

Figure 6.4: Contxt words of the GALPE B unit of Figure 6.2.

The GALPE B unit operation remains to be the same and therefore, its context words are the same. the addresses of the operands show that the butterfly connection to the above MeMPE unit is selected as the X input and hence the X operand address field is loaded with 1100. The Y input is routed to the GALPE unit from the adjacent column, one row above is selected by setting the corresponding field to 0110

No truncation is needed in this particular configuration for both GALPE A and GALPE B units since only simple addition/subtraction operations are performed. Hence, the truncation bit fields are set to 0000 as shown in Figure 6.4.

The MeMPE unit in Figure 6.2 is carrying out the job of a FIFO in both cycles of operation and therefore the 4 LSBs holding the instruction are set to 0010. This instruction indicates that the MeMPE unit is to shift the input data vector in. The shift in operation takes place from the indicated memory address to the least significant vector stored in the MeMPE unit that id delivered as an output. For a 2 vector deep FIFO the corresponding vector memory address is 1 (remember that the current implementation of the HPad incorporates MeMPE units of 2 vector positions).

The input vector routing changes in each cycle of operation. In the first cycle of operation, the output from the GALPE unit from the column on the left one row above is selected with 0110 in the corresponding filed as shown in the Figure. The second context word holds 1111 to rout the output of the GALPE B unit.

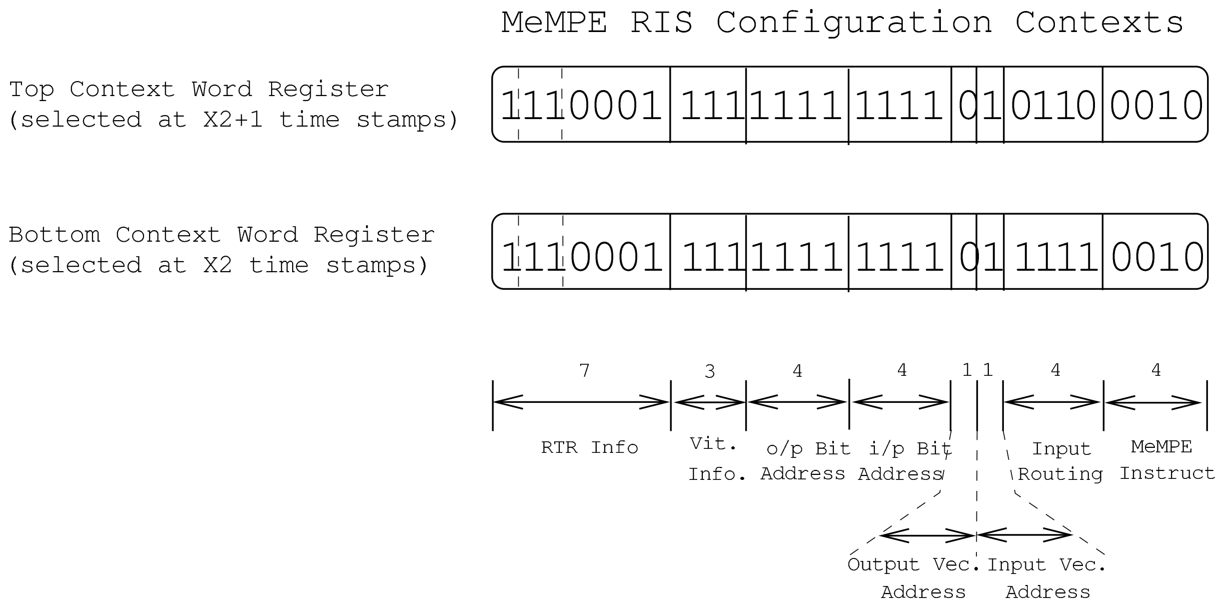

Figure 6.5: Contxt words of the MeMPE unit of Figure 6.2.

6.2 FIR Filter Realizations

FIR filter structures, as was shown in Chapter 3, are very regular. Mapping of these FIR structures on the HPad DPA is therefore very efficient. Both the DF and TDF structures of the FIR filter have the "tap" as the basic building block that is replicated many times according to the size of the target FIR filter. Each tap consists of an adder, a multiplier and a storage element holding the filter coefficient. The former two items can be implemented in the HPad by the GALPE units and the lated can be implemented by a MeMPE unit.

6.2.1 TDF FIR Filter Mapping

Mapping of the TDF form FIR filter is shown in Figure 6.6. As depicted in Figure 6.6 DSBs are used to "*broadcast*" the streaming input to each tap. Taps are mapped by using two GALPE units aligned one on top of the other. The top one is used for multiplication with a MeMPE unit serving as a ROM to hold the coefficient. Two rows of the HPad DPA are needed thus to implement $ElementsPerRow$ taps in the square configuration of the HPad DPA.

Here the "butterfly" connections are utilized to realize the connection between the multiplier and adder of each tap. Butterfly connections are also utilized to deliver the output of the local MeMPE units (storing coefficients) to the multiplier GALPE units. The input vector is made available locally to each tap by the DSB as mentioned above.

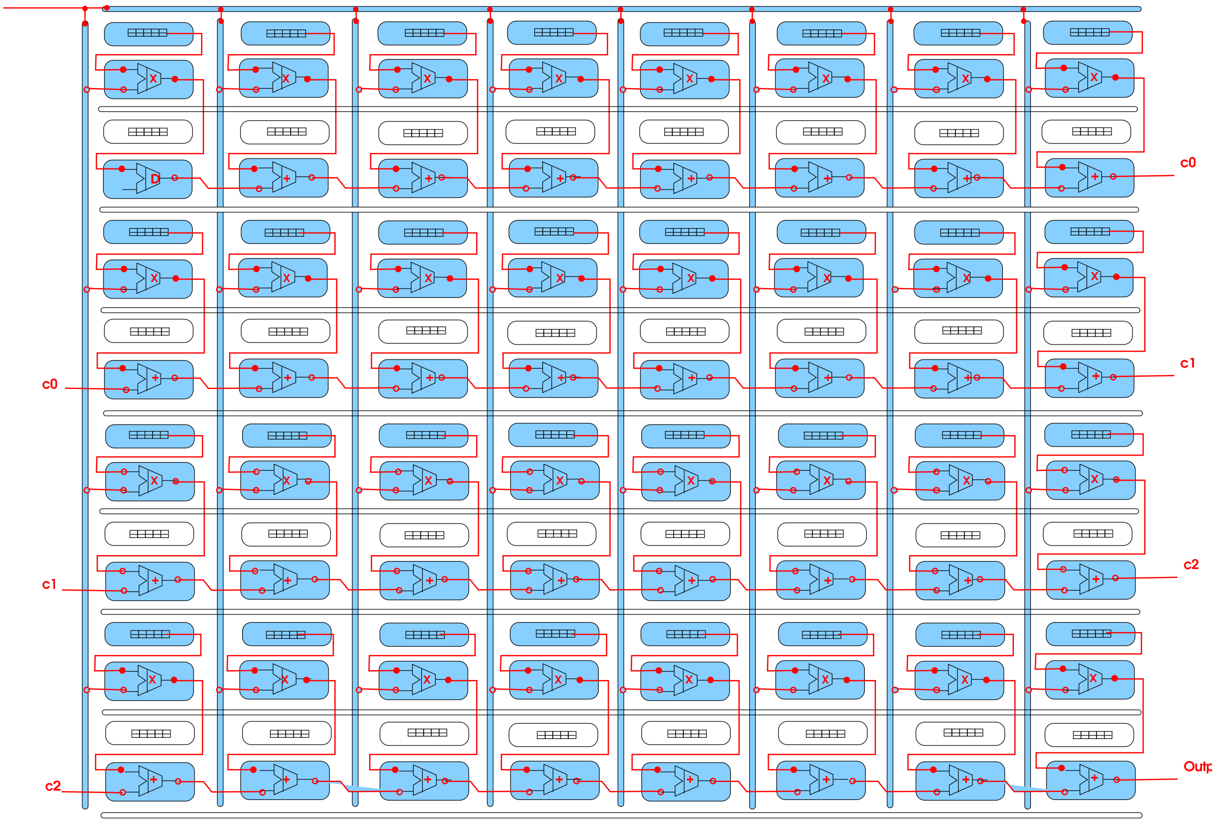

Figure 6.6: Mapping of the TDF form FIR filter on the HPad DPA.

Routing between taps is implemented by adjacent GALPE to GALPE connections. The output data at the end of the "*double row*" of the HPad DPA forming a portion of the FIR filter is *rolled* to the next double row using this useful routing facility of the HPad DPA. Pipelining of the TDF FIR filter is implemented by choosing the optional registering feature of the outputs of the GALPE units.

This configuration of the TDF FIR filter results in utilizing 100% of the GALPE units, 56% of the DSBs units and 50% of the MeMPE units providing thus very high utilization of the HPad DPA and realizing a 32-tap FIR filter.

6.2.2 DF FIR Filter Mapping

Mapping of the DF FIR filter is illustrated in Figure 6.7. Here again two of the GALPE units oriented on top of each other along with a couple of local MeMPE units are used to implement each tap. One GALPE unit is configured as a multiplier and the other is configured as an adder while one of the MeMPE units is used as a ROM storing the tap coefficient and the other is used to delay the input data to be delivered to each tap as input at the correct instant.

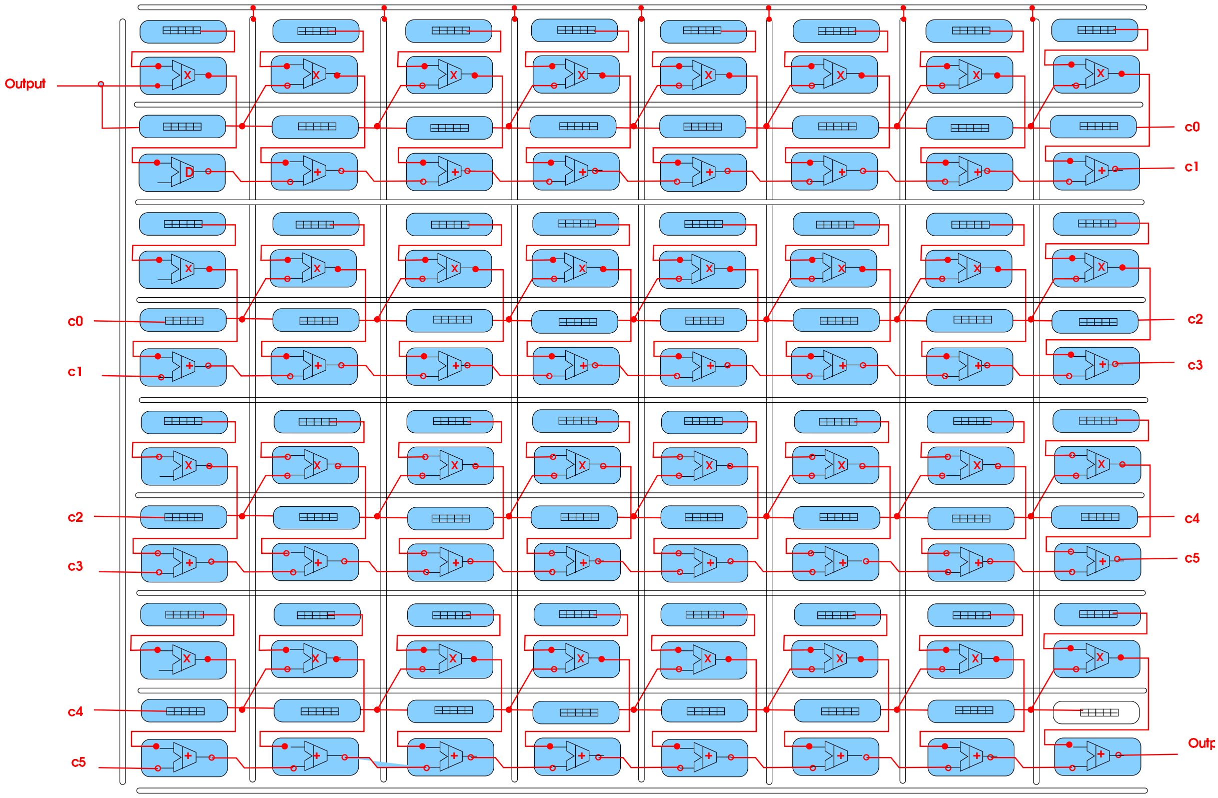

Figure 6.7: Mapping of the DF form FIR filter on the HPad DPA.

Butterfly connections provide here a valuable resource enabling connecting the multipliers to both top and bottom MeMPE cells in order to access coefficients and delayed inputs. The same type of connections are also used to pass products from the GALPE multiplier units to the bottom adders. Direct connections are also used here to connect between the adders input and outputs. Routing of partial results from left side of the HPad DPA to the right side of it is also implemented here in a similar manner to the rolling of data implementation discussed above in the TDF example. Pipelining is implemented by both the registering of output option of the GALPE units as well a retiming of the input data delays using both of the MeMPE units registers shifting data in them for two cycles.

The above DF FIR filter configuration uses 100% of the GALPE units, 100% of the MeMPE units and no bussed are needed. This presents again a very high area efficiency for this configuration.

6.3 FFT Realization

Mapping of FFT processors on the HPad DPA is obviously more complicated than that of FIR filters for the reasons discussed previously in this thesis. Butterfly connections combined with complex operations result in mapping and routing challenges given the limited routing resources of the HPad DPA (or any other reconfigurable array) since the complete pipelined algorithm is to be implemented and not only small parts of the algorithm -say the butterfly operation while the rest implemented by software- as been implemented in some systems.

In the following we show how the HPad can be used to realize both the data path along with the RTR requirements of the R2SDF implementation of a $16\ pt.$ FFT. Considering the current size of the HPad DPA, the $32\ pt.$ FFT is almost impossible to implement because it needs almost 100% of the HPad DPA resources (GALPE, MeMPE and DSB units).

Figure 6.8 illustrates mapping of a $16\ pt.$ FFT on the HPad DPA. The configuration of Figure 6.8 show the implementation of the R2SDF architecture for a DIF FFT algorithm.

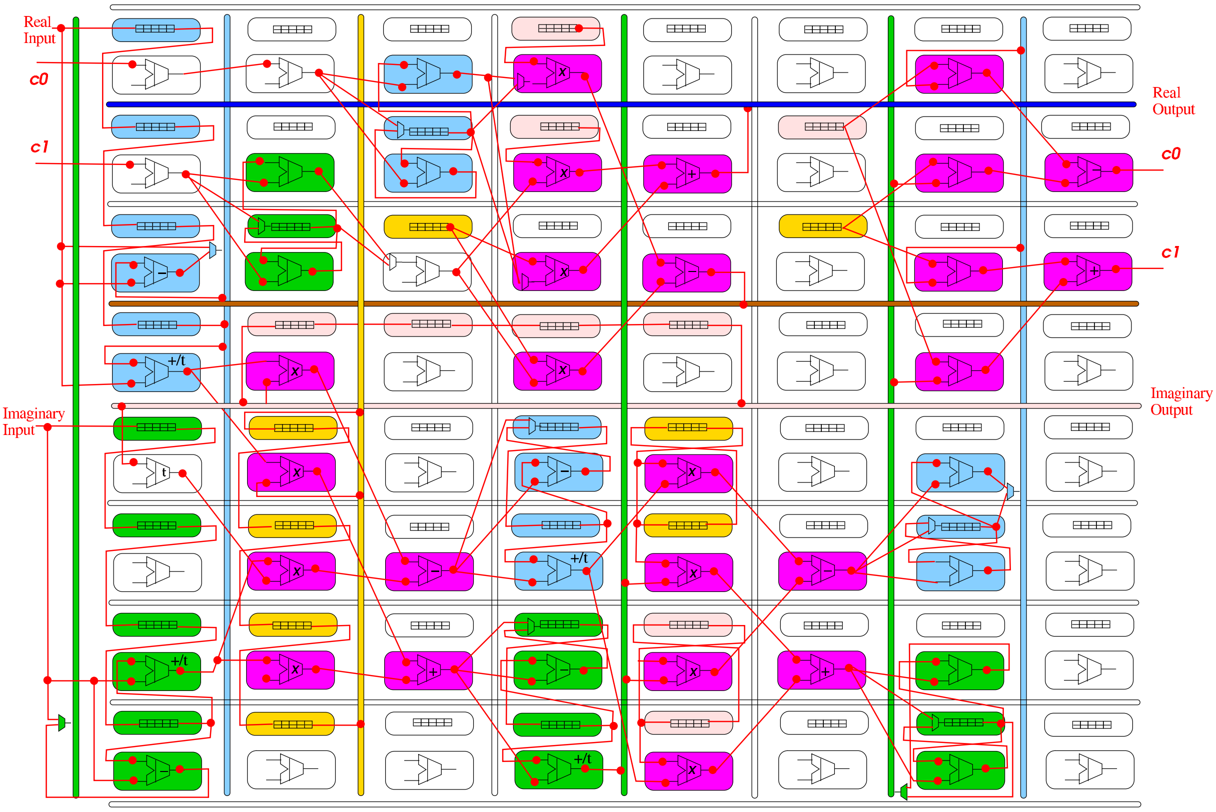

Figure 6.8: R2SDF $16\ pt$ FFT implementation on the HPad.

The real and imaginary inputs of the 16 *pt.* FFT are passed to the inputs at the left side of the HPad DPA. The first stage dynamic butterflies are implemented as discussed in previously Section 6.1.4 using a vertical DSB, 4 MeMPE units and a couple of GALPE units for computations. The DSB switches between passing in newly received information data or partial results produced by the butterfly GALPEs to the FIFOs formed by the MeMPE units. Both real and imaginary dynamic butterflies are mapped to the first column of the HPad DPA utilizing both buses at the right and left of that column one DSB for each dynamic butterfly.

After the real and imaginary dynamic butterflies follows the complex multiplication operation realized by 6 GALPE units. The inputs of the complex multiplier are the real and imaginary outputs produced by both the preceding dynamic butterflies and the twiddle factors stored in ROM cells rotating their inputs to provide the complex multipliers with the proper inputs at each instant. Two of these ROMs (for the real and imaginary parts of the twiddle factors) are realized by MeMPE cells interconnected in vertical and horizontal topologies. A horizontal and a vertical DSBs are used for long transfers of the twiddle factors being rotated in the ROMs. The complex multipliers produce real and imaginary results to be consumed by the next stage in the realized R2SDF architecture.

Succeeding the complex multipliers come the two 4-vector deep dynamic butterflies processing the real and imaginary products produced by the complex multiplier. In this case of 4-vector deep butterflies no DSB are needed to realize the FIFOs, rather, a couple of MeMPE units are routed vertically together utilizing the butterfly connections facilities to realize the complete dynamic butterfly including also a couple of GALPE units located in the same cell positions of the MeMPE units. This locality maximizes thus the utilization of local interconnection possibilities provided in the HPad DPA.

Subsequent to the 4-vector deep dynamic butterflies complex multiplier is mapped. For this multiplier stage only 4 twiddle factors are needed. Here again they are implemented by four MeMPE units (two MeMPE units for real parts of the twiddle factors and two for the imaginary). Both real and imaginary ROMs are implemented by vertically interconnecting MEMPE units without the need of additional DSB elements.

Next come two of the 2-vector deep dynamic butterflies each using a couple of GALPE units and a single MeMPE unit located between them. At this point the end side of the HPad DPA is approached with only one free column is left at the right which is not enough for the complex multiplier. We then transmit the real and imaginary result of the butterfly units vertically to the top rows of the HPad DPA.

On top of both the 2-vector deep dynamic butterflies the multiplier stage is placed making use of the butterfly results made available by the vertical buses at the right and left sides of the dynamic butterflies. the real and imaginary twiddle factors stored in MeMPE units in the column left of the complex multiplier stage. The complex multiplier implemented in two columns: one for multipliers and the second for adders producing the real and imaginary products at the end side of the HPad DPA. These products are rolled back to the left side of the HPad DPA.

The rolled data are sampled and transferred by the two GALPE units at the top right side of the HPad DPA. Two 1-vector deep dynamic butterflies are implemented in two adjacent columns because of the occupancy of MeMPE and GALPE units with the early R2SDF stages. GALPE units are used to transfer and delay data to ensure the preservation of timing integrity.

Later the last multiplier stage is implemented with several local MeMPE units used to pass twiddle factors to the multipliers. The final results are passed to the output through two horizontal DSBs.

RTR operations of the dynamic butterfly units are implemented by having both the storage and processing stages of the butterflies written in the RIS of the various building elements of the HPad DPA. This is indicated by multiplexers in Figure 6.8. The select lines of these multiplexers are provided by the OpControl unit time stamp flags as discussed in the previous Chapter.

The FFT realization on the HPad discussed above shows efficient usage of all types of routing resources, processing elements and DSBs as well as RTR with the aid of the OpControl unit.

6.4 DCT Realizations

DCT implementations exhibit a higher degree of irregularity in comparison to the FFT. However this disadvantage is minimized because of the only real operations of the DCT and its typical smaller sizes in common applications. A typical DCT size is the $8-point$ DCT discussed in [38, 47, 87, 67, 88, 66]. In this section we discuss the mapping and routing of one of the DCT architectures proposed in the references above on the HPad.

A simple structure of the DCT is the R2SDF architecture shown in [87]. The top part of Figure 6.9 shows the implementation of the R2SDF DCT architecture on the HPad. While inverse DCT implementation on the HPad is illustrated in the bottom part of Figure 6.9.

Both realizations use the same concepts discussed in Section except here an additional GALPE is used to select and pipeline the output of each dynamic butterfly since a lot of resources are available in the HPad DPA since the $8 - point$ DCT does not require a lot of resources.

A new simple element implementing the irregularities of the DCT SFG. These are realized with MeMPE units implementing delays and GALPE units for addition. The last stage of both the DCT and IDCT R2SDF structures involve selection between 3 signals. This is implemented by using two GALPE units for selection since the current

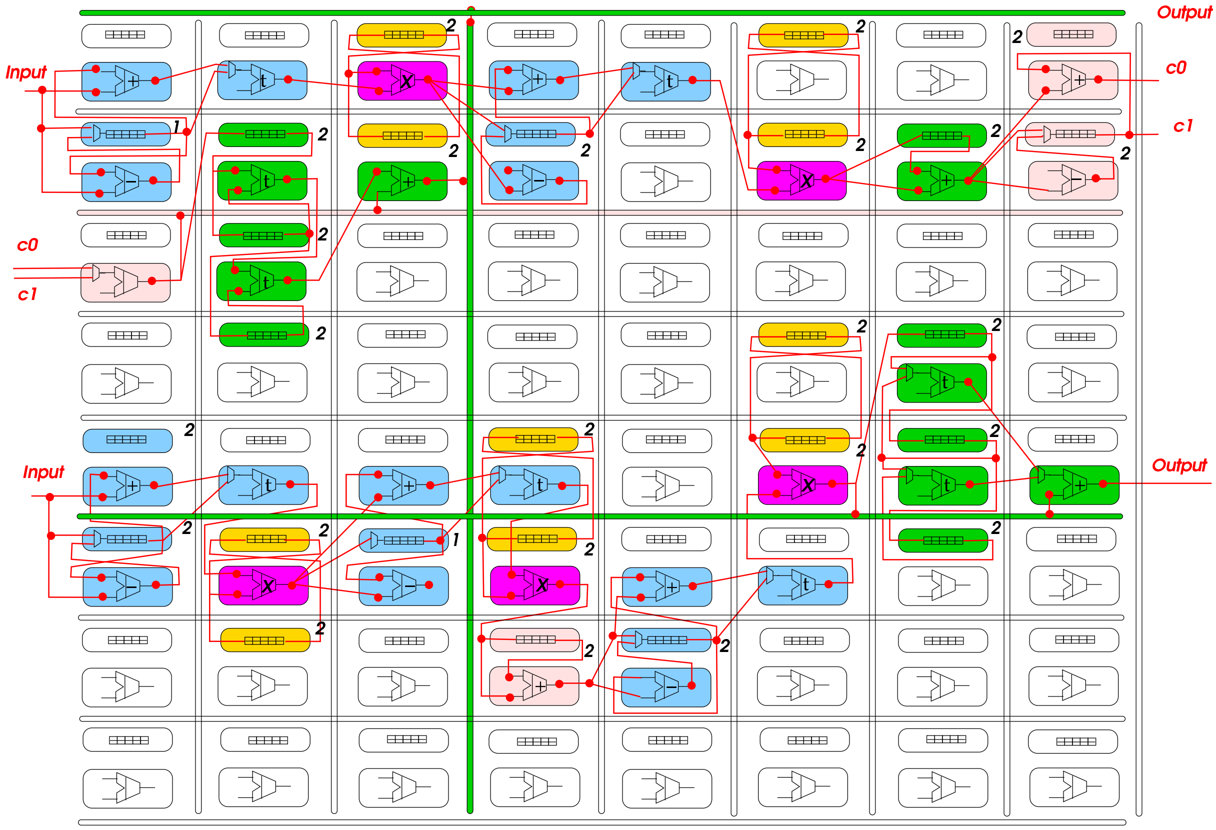

Figure 6.9: R2SDF DCT structure of [87] implemented on the HPad.

HPad implementation has only two fields of contexts in each RIS. As shown in Figures Figure 3.25 a) and b) real multipliers interleave the aforementioned elements. GALPE units are used for multiplication and interconnected MeMPE units are used for rotating multiplication factors.

6.5 Viterbi Decoding Realization

The realization of the Viterbi decoder may be the most difficult one because of the high amount of irregularities in its SFG and the involvement of bit operations as has been pointed out earlier in this work. One of the most difficult problems in the Viterbi decoder implementation is its irregularly spaced butterfly operations sizes. This was remedied by the in-place replacement approach introduced in Chapter 3. In the following we present the implementation of a small Viterbi decoder for a $K = 3$ i.e. 4 state convolutional encoder. To illustrate how can even such applications be realized by the HPad we choose to implement an R2SDF Viterbi decoder with a RE trellis window decoding as proposed in [105]. The mapping and routing of this architecture is depicted in Figure 6.10

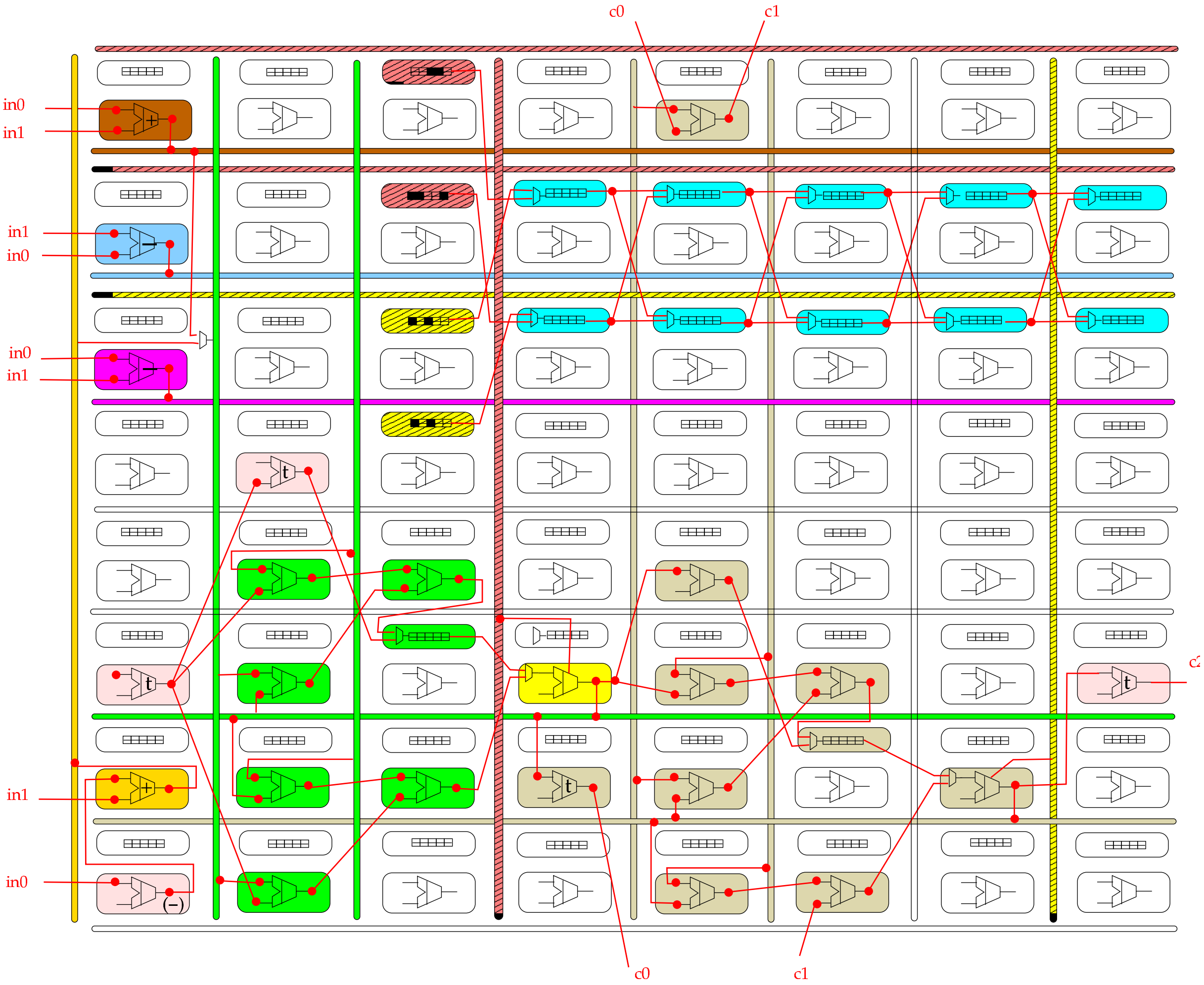

Figure 6.10: Mapping and routing of a $K = 3$ Viterbi decoder on the HPad. Shaded DSBs represent their bit transfer.

As was mentioned in Chapter 3 Viterbi decoders consist of three main blocks: branch metric computations, path metric computations (or ACS) and trellis window decoding.

The branch metrics can be produced by the use of 5 GALPE units in the first column of the HPad DPA producing all possible branch metrics. The dynamic ACS butterfly units consist of 6 GALPE units, four of them for adding branch metrics and two for comparing. Additionally, a number of MeMPE units are used to implement FIFOs of the appropriate sizes.

The problem with the R2SDF ACS units is that they do not only need path metrics as inputs, but moreover, they need to read the branch metrics corresponding to the current path metrics processed at each point of time. To achieve this the outputs of the branch metrics units are passed through horizontal buses through the HPad DPA. The two appropriate branch metrics are then passed to the ACS units through the two intersecting vertical buses at both sides of the GALPE units performing the addition operation. These vertical buses dynamically select the appropriate branch metrics and pass them to the ACS units according to the time stamp flags producer by the OpController unit.

An interesting point to mention is that the vertical cylindrical routing possibilities proved useful because it helped realize the second dynamic butterfly that needed one more GALPE unit after reaching the bottom side of the HPad DPA. A GALPE unit from the top row is simply utilized without changing the usual dynamic butterfly routing pattern. Subsequent to the second dynamic butterfly ACS unit, the output is rolled back to the first ACS stage at the input side of the HPad DPA implementing thus the big feedback loop of the R2SDF structure of the Viterbi decoder.

At the output of each dynamic butterfly stage the decision bits from the GALPE units are transmitted through vertical DSBs. On the top side of the HPad DPA the RE structure a couple of the horizontal DSBs select both the decision bits to be used in the RE operation.

In the RE operation, the produced decision bits are used to select the new trellis window bits either from the top or the bottom of the of the previous stage bits. Here each MeMPE unit is used to store a single bit of the trellis window bits and produces them to the next stage. The input of each MeMPE units is either the top or bottom previous bit. In this case the selection of the input is run-time reconfigured according to the corresponding decision passed on from the selected DSB.

6.6 Concluding Remarks

In this Chapter, the suitability of the HPad to DSP applications was validated. Architectures of several DSP algorithms were mapped and routed on the HPad. RTR capabilities of the HPad showed also suitability of the HPad to the studied DSP algorithms architectures.

Basic operations such as double-sized arithmetic is achievable using the HPad DPA PE and routing capabilities.

As examples of complete algorithms, FIR filters, FFT processors, DCT processors and Viterbi decoding were mapped and routed on the HPad. The above mentioned algorithms present a good range of DSP applications showing thus the suitability of the HPad to a wide range of DSP applications.

Regular architectures such as FIR filtering showed a high (approaching full) utilization of the HPad DPA. Others that are less regular and with more complicated routing characteristics were also implemented but their utilization is expected to be lower because of their routing irregularities as well as the large variance in their basic functional blocks sizes.

RTR operation was utilized in the mapped pipelined architectures of the FFT, DCT and Viterbi decoding. The time stamp flags generated by the Operation Controller was used in most applications. For Viterbi decoding local events (selected bits) were used to trigger RTR activity in the trellis table decoding part of the algorithm.

In the above routing and mapping examples, all of the routing types of the HPad DPA namely all left column connections, all butterfly connections and DSB signal broadcasting were used. This shows once again the suitability of the routing topology of the HPad DPA.

In general, in the above discussed mapping and routing examples, all of the proposed HPad components were used as was intended in the HPad design.

Chapter 7

Conclusions

Reconfigurable Computing for all its flexibility and power presents a promising solution not only closing the various design gaps, but moreover, for bringing about a multitude of computational possibilities. Due to several challenges, the full capacity of Reconfigurable Computing is yet to be achieved by the given solutions either in the market or in the literature.

The challenges facing the development of Reconfigurable Computing fall mainly in two categories: classical and self induced. Classical problems lie in the higher consumption of area and power as well as slower operation as compared to finely crafted ASICs. These problems are being remedied in research mainly by finding optimal processing elements (or logic blocks) architectures that exhibit high functionality and lower area and power consumption. As a result, reconfigurable solutions are being polarized to fine-grained and coarse-grained solutions in order to best meet the target applications. Within each category, further flavors and sub categories are found, again each seeking more efficient solutions for the intended range of problems. From the technological side efforts are made for finding reconfiguration devices of smaller size with lower cost and power consumption. In this context, several programming technologies such as SRAM, EEPROM and Anti fuse were introduced and are constantly being improved.

As a matter of fact, while observing the evolution of reconfigurable architectures, one can see how the designers' experience and the market demand factors reshaped the newest reconfigurable computing products. It is notable that there is a trend to use coarse grained blocks even in state of the art fine grained FPGAs seeking thus more efficient realizations of potential applications. Additionally, in fine grained solutions, bigger logic blocks are being introduced.

On the other hand, the self induced design challenges result from the very nature of reconfigurable computing: reconfigurability & computation!. From a computational point of view having a very powerful array of processors capable of very high throughputs may not be useful at all if there is no feasible way to use it. Software support and integration along with other system issues present true bottlenecks. From the reconfiguration point of view new concepts such as RTR operation, partial reconfigurability and speed of reconfiguration are among the new concepts that were not previously known. Finding efficient design schemes addressing the above mentioned concepts have triggered quite an amount of research.

The goal of this thesis is to design a reconfigurable device capable of efficiently realizing a family of DSP applications by emulating ASIC architectures. By that we seek to reach comparable performance to ASICs yet permitting flexibility and reduction of NRE costs. A pragmatic methodology to engage such a design task is to study different DSP applications and their implemented VLSI architectures. Hence, architectural features to be supported in the target reconfigurable platform are extracted. It is worthy of stressing that VLSI DSP architectures are do always directly represent the original DSP algorithm, but on the contrary, the design of high speed DSP algorithms structures involve in many cases craftsmanship and design skills as well as comprehensive understanding of the DSP algorithm at hand. We, therefore, assume a library based computational paradigm where predefined configurations are stored in a ROM and loaded when needed.

After studying a number of the reported architectures of popular DSP algorithms that were selected to cover a wide range of DSP applications, we drew a number of conclusions. These conclusions can summarized in the following design requirements are needed for the target reconfigurable solution: heterogeneity, coarse grain with some fine grain features and RTR capabilities.

The proposed HPad takes its name from *H*eterogeneity and that its resemblance to the writing *Pad* that when used to solve a problem, it is written into line by line; and so is the HPad configured. The HPad could be categorized as a loosely coupled attached processing or an accelerator unit that can be attached to a system with minimal integration efforts.

In the HPad, RTR operations are tackled from three angles: globally, locally and from the context point of view. *Globally* when the RTR operation is permitted, and at a relative operation time point, each of the HPad processing elements can *locally* and through Reconfigurable Interface Sockets decide on switching to another predefined configuration (an other locally stored context). Each processing element can also switch to another configuration according to a local flag or a received bit. The *context* register include a $7-bit$ field summarizing the RTR conditions. The above imply multi context as well as dynamic and partial reconfigurability.

To support the aforementioned requirements, the HPad is organized in the following blocks: the HPad Data Path Array (*DPA*), the Reconfiguration Controller, the Operation Controller and the Reconfigurable IN/OUT FIFOs. All the above sub systems operation are orchestrated by the Reconfigurable Controller.

The Operation Controller is a sequencer that generates global time stamp flags that may be needed to schedule RTR actions. The FIFOs help maintaining contentious streaming of data in and out the HPad DPA. However, at a read or write conflict event, the Reconfigurable controller *freezes* the operation of the system to ensure data integrity and the correctness of results. The Reconfiguration Controller also manages full and partial reconfiguration of the HPad.

The net area efficiency in terms of the effective processing area data path to the total area exceeds 40% which is more than one order of magnitude better than its fine grained FPGA counter parts. Furthermore, several architectures of DSP applications were routed and mapped on the HPad. The presented mapping and routing examples show good utilization of the HPad PEs, routing resources and RTR features.

For future work, integration of the HPad with several μps and testing its operation within the μp system running other applications as well can help in further tuning its architectural features. In addition, other applications such as block error correction, data compression and encryption and the suitability of the HPad architecture to their realization can be studied. The above mentioned studies may result in further improvements of the HPad architecture and may point out more issues that were not previously brought into attention by the studied DSP applications.

Zusammenfassung

Rekonfigurierbares Rechnen stellt durch seine hohe Flexibilität und Leistungsfähigkeit nicht nur eine vielversprechende Lösung zur Schließung verschiedener Design Gaps dar, sondern bietet außerdem eine Vielzahl von Berechnungsmöglichkeiten. Die volle Kapazität rekonfigurierbaren Rechnens wurde bisher aufgrund verschiedenster Herausforderungen jedoch noch nicht voll ausgeschöpft, weder von auf dem Markt befindlichen Lösungen noch von solchen aus der Literatur.

Die Herausforderungen angesichts der Entwicklung rekonfigurierbarer Systeme fallen hauptsächlich in zwei Kategorien: klassische und spezifische Probleme. Klassische Probleme sind der höhere Flächen- und Leistungsverbrauch sowie langsamere Rechengeschwindigkeiten verglichen mit optimierten ASICs. Diese Probleme werden in der Forschung vor allem durch die Entwicklung von optimalen Architekturen für Rechenelemente (oder Logikblöcke) angegangen, welche hohe Funktionalität sowie niedrigen Flächen- und Energieverbrauch aufweisen. Dies hat eine Polarisierung rekonfigurierbare Systeme in grob- und feingranulare Architekturen zur Folge, um das System möglichst gut an die Anforderungen der Zielapplikation anzupassen. Innerhalb jeder Kategorie können weitere Nuancen und Unterkategorien gefunden werden, die ihrerseits auf eine noch effizientere Lösung der betrachteten Probleme abzielen. Von der technologischen Seite bemüht man sich, immer kleinere, kostengünstigere und energiesparendere rekonfigurierbare Architekturen zu finden. In diesem Zusammenhang wurden verschiedene Rekonfigurationstechnologien eingeführt, die beispielsweise auf SRAM, EEPROM oder Anti-fuse basieren und laufend verbessert werden.

Beobachtet man die Entwicklung rekonfigurierbarer Architektur, kann man tatsächlich feststellen, wie die Erfahrung der Designer und die Anforderungen des Marktes die neusten Produkte rekonfigurierbarer Rechensysteme gewandelt haben. Es ist bemerkenswert, dass es einen Trend zum Einsatz grobgranularerer Architekturen gibt, selbst bei aktuellen feingranularen FPGAs, wodurch eine noch effizientere Realisierung potentieller Anwendungen angestrebt wird. Außerdem werden auch in feingranularen Architekturen größere Logikblöcke eingeführt.

Andererseits ergeben sich spezifische Designherausforderungen gerade aus dem rekonfigurierbaren Rechnen selbst: Rekonfiguration & Rechnen! Von Seiten der Berechnung her hat ein Array aus sehr leistungsstarken Prozessoren wenig Nutzen, wenn deren Rechenleistung nicht in geeigneter Weise verwendet werden kann.

Die Softwareunterstützung und Integration mit anderen Systemkomponenten sind entscheidende Problembereiche. Von Seiten der Rekonfiguration sind neue Konzepte wie Rekonfigurationsgeschwindigkeit, dynamische und partielle Rekonfiguration aufgetaucht, die vorher unbekannt waren. Effiziente Lösungen für diese Konzepte zu finden hat eine Vielzahl von dokumentierten Forschungsaktivitäten ausgelöst.

Das Ziel dieser Doktorarbeit ist, eine rekonfigurierbare Architektur zu entwickeln, die die Fähigkeit zur effizienten Realisierung einer Reihe von DSP Anwendungen besitzt. Dabei wird eine mit ASICs vergleichbare Performanz angestrebt, während gleichzeitig Flexibilität und eine Reduzierung der Entwicklungsfixkosten gewährleistet werden soll. Eine pragmatische Methodik zur Lösung solch einer Entwurfsaufgabe ist, verschiedene DSP Anwendungen und deren VLSI-Implementierungen zu untersuchen. Dadurch können Architektureigenschaften für die rekonfigurierbare Zielplattform extrahiert werden. Es muss betont werden, dass VLSI Architekturen für DSP Anwendungen nicht immer direkt aus dem eigentlichen DSP Algorithmus ersichtlich sind, sondern dass häufig viel Entwurfserfahrung und umfassendes Verständnis der DSP Algorithmen erforderlich sind, um geeignete Strukturen für Hochgeschwindigkeits-DSP-Anwendungen zu entwerfen. Daher wird angenommen, dass für alle Berechnungen eine Bibliothek mit vordefinierten Konfigurationen zugrunde liegt, die in einem ROM gespeichert sind und bei Bedarf geladen werden können.

Aus der Untersuchung verschiedener Architekturen von gängigen DSP Algorithmen, die ein weites Spektrum an DSP Anwendungen abdecken, wurden verschiedene Schlüsse gezogen. Zusammengefasst kann gesagt werden, dass eine effiziente rekonfigurierbare Systemlösung heterogen, grobgranular und dynamisch rekonfigurierbare sein sollte.

Das vorgeschlagene HPad trägt seinen Namen aufgrund seiner Heterogenität sowie seiner Ähnlichkeit mit einem Schreibblock (engl.: pad) in dem Sinne, dass es ebenfalls zur Problemlösung benutzt wird und zeilenweise beschrieben bzw. konfiguriert wird. Das HPad kann als lose gekoppelte Beschleunigereinheit klassifiziert werden, deren Integration in ein System nur minimalen Aufwands bedarf.

Dynamische Rekonfiguration wird im HPad aus drei Richtung angegangen: global, lokal und kontextabhängig. Wenn dynamische Rekonfiguration global gestattet ist, kann jedes Rechenelement des HPad zu vorgegebenen Zeitpunkten lokal und mithilfe einer fest definierten Rekonfigurationsschnittstelle entscheiden, ob zu einer anderen vordefinierten Konfiguration (einem anderen lokal gespeicherten Kontext) gewechselt werden soll. Jedes Element kann außerdem abhängig von einem lokalen Flag oder einem empfangenen Bit Kontextwechsel durchführen. Ein Kontextregister beinhaltet dazu ein Feld von 7 Bit, in denen die Bedingungen für einen Kontextwechsel zusammengefasst sind. Die vorgestellte Lösung setzt eine Multi-Kontext-Architektur sowie dynamische und partielle Rekonfiguration voraus.

Zur Realisierung dieser Funktionalität ist das HPad in folgende Komponenten gegliedert: Das HPad DPA, eine Rekonfigurationssteuerung, eine Ablaufsteuerung sowie rekonfigurierbare FIFOs an den Ein- und Ausgängen. Die Rekonfigurationssteuerung sorgt hierbei für das reibungslose Zusammenspiel dieser Subsysteme.

Die Ablaufsteuerung dient zur Generierung globaler Zeitmarker, die zur zeitlichen Ablaufplanung der dynamischen Rekonfiguration genutzt werden können. Die FIFOs sorgen für einen so konstant wie möglich gehaltenen Datenstrom zum HPad DPA. Tritt dennoch ein Schreib- oder Lesekonflikt auf, so friert die Rekonfigurationssteuerung die Verarbeitung im System zeitweilig ein, um die Datenintegrität und die Richtigkeit der Ergebnisse zu gewährleisten. Die Rekonfigurationssteuerung kontrolliert außerdem die vollständige oder partielle Rekonfiguration des HPad.

Die Flächeneffizienz, ausgedrückt durch das Verhältnis von effektiv zur Datenverarbeitung genutzten Fläche zur Gesamtfläche, übersteigt 40% und ist somit um eine Größenordnung besser als feingranulare FPGAs. Ferner wurden mehrere Architekturen von DSP Anwendungen auf das HPad abgebildet. Die präsentierten Beispiele zeigen eine gute Ausnutzung der Rechenelemente, der Verbindungsressourcen und der Rekonfigurationsmechanismen des HPad.

Für die Zukunft können die Integration des HPad mit mehreren Mikroprozessoren sowie Funktionstests in einem Mikroprozessorsystem zusammen mit anderen laufenden Anwendungen dabei helfen, die Architektureigenschaften des HPad weiter zu verfeinern. Zudem können weitere Anwendungen wie Blockverfahren zur Fehlerkorrektur, Datenkompression und Verschlüsselung sowie deren Eignung zur Realisierung auf dem HPad untersucht werden. Dies könnte zu einer weiteren Verbesserung des HPad führen sowie weitere Aspekte aufzeigen, die in den bisher untersuchten Anwendungen nicht in Erscheinung getreten sind.

Appendix A

Reconfigurable Interface Sockets: General

In this appendix the common features of all types of the RIS blocks of the HPad DPA are presented.

A RIS block encompasses either a PE or DSB. The RIS functions as an adaptor passing the appropriate context to the PE and routing the required signals to it. This is achieved by selecting the proper context word holding routing information, instruction and RTR configuration information.

In addition to the data inputs, RIS blocks input External control signals from the operational controller, control signals from the reconfiguration controller (including the reconfiguration instruction discussed below) and receive and pass context vectors.

As shown in Figure A.1, each RIS unit holds a couple of context words in its context registers. The selection of the context word is either determined by the reconfiguration controller unit for static configuration or by a local or a global event chosen by the RTR_Info field of the current context word. Table A.1 below summarize the RTR reconfiguration possibilities.

The seven most significant bits of the context word determine the RTR action. The layout of these bits are shown in Figure A.2 and described in the Tables below.

Loading of context words is controlled by the SocConf signal according to Table A.2. The SocConf signal specifies the target context register address and the reconfiguration loading operation of the RIS.

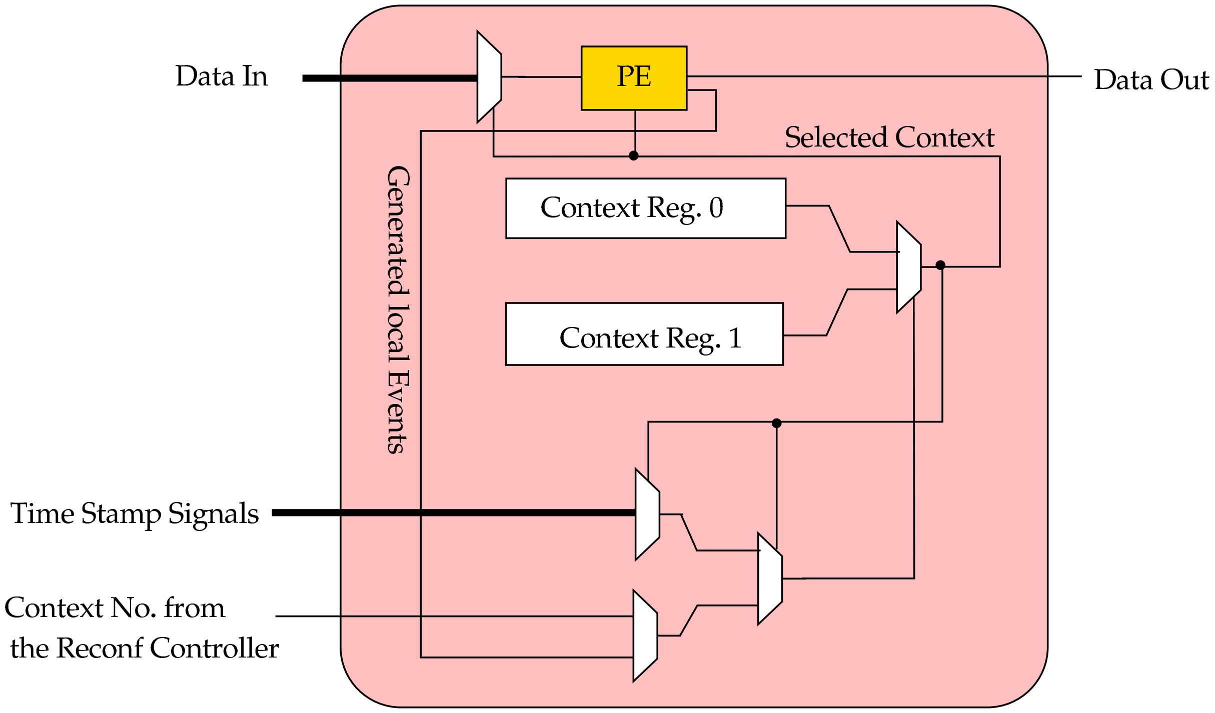

Figure A.1: A block digram depicting the context switching mechanism in the RIS units of the HPad.

Table A.1: RTR control instruction. The selected control bit is obtained according to the address specified by the ExtSel field of the current context word.

RTR control instruction	No. of selected context word
000	Control signal from the reconfiguration controller
001	Selected input bit to the PE
010	Negative flag produced by the PE
011	Zero flag produced by the PE
1XX	Selected external control bit from the operation controller

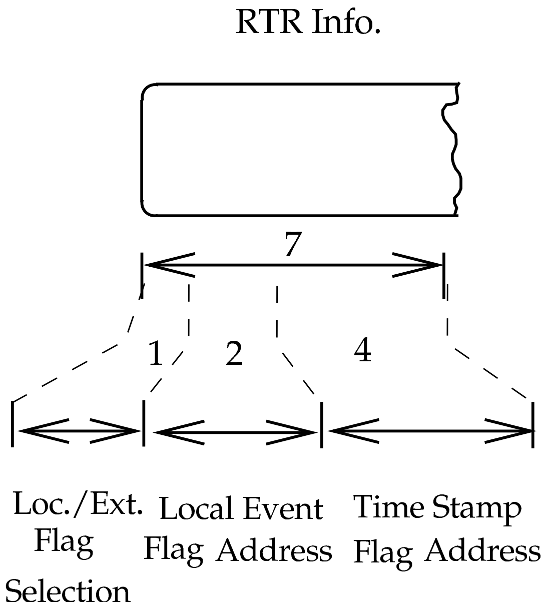

Figure A.2: Different fields in the RTR portion of the context words.

Table A.2: Context words loading instruction description.

SocConf value	Action
00	Control signal from the reconfiguration controller Keep old contexts (default)
01	Load configuration to the top context register shifting the old value to the next RIS.
10	Load configuration to the bottom context register shifting the old value to the next RIS.
11	Shift in the new configuration to the top context register through the bottom one shifting the old value to the next RIS.

Appendix B

The GALPE RIS Unit

In this appendix the GALPE RIS structure, its instructions as well as its stored context words are described. Within this description, the instruction set of the GALPE is also provided.

The GALPE RIS context word is 29 bits wide. Figures 5.9 and 5.8 show the most interesting functions of the GALPE RIS unit. The most significant 7 are used for RTR purposes and the rest are described in Table B.1 and its general structure is shown in Figure B.1.

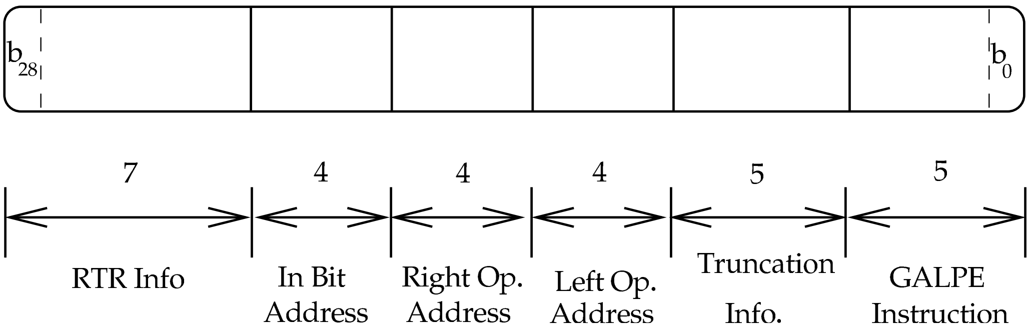
GALPE RIS Configuration Context

Figure B.1: Different Fields in the GALPE configuration context.

The 6-bit GALPE instructions are described in Table B.2 below.

Table B.1: GALPE RIS context words description.

bits range	description
28-22	RTR info (see Appendix A)
21-18	Bit inputs address
17-14	Right vector operand address
13-10	Left vector operand address
10- 6	Truncation address
5- 0	GALPE instruction

Table B.2: GALPE instruction set.

bits range	description	
5-5	Clocked result?	
4-4	Reuse previous clocked result?	
3-0	Instructions:	
	0000	Z = X + Y
	0001	Z = X - Y
	0010	Z = -X
	0011	Z = X AND Y
	0100	Z = X OR Y
	0101	Z = X XOR Y
	0110	Z = NOT X
	0111	Z = Min(X, Y) , co = 0 if X, 1 otherwise suitable for the Viterbi ACS operation
	1000	Z = SRA X
	1001	Z = SLA X
	1010	Z = ROR
	1011	Z = ROL
	1100	Z = X * Y
	1101	Z = X * Y + Z(t-1)
	default	Z = Xm

Appendix C

The MeMPE RIS Unit

In this appendix the MeMPE RIS structure, its instructions as well as its stored context words are described. The description of the MeMPE instructions is also listed.

The MeMPE RIS context word is 28 bits wide. Figures 5.9 and 5.8 show the most interesting functions of the GALPE RIS unit. Similar to the GALPE RIS and DSB RIS the most significant 7 bits are used for RTR purposes and the rest are described in Table C.1 and the context word's general structure is shown in Figure C.1.

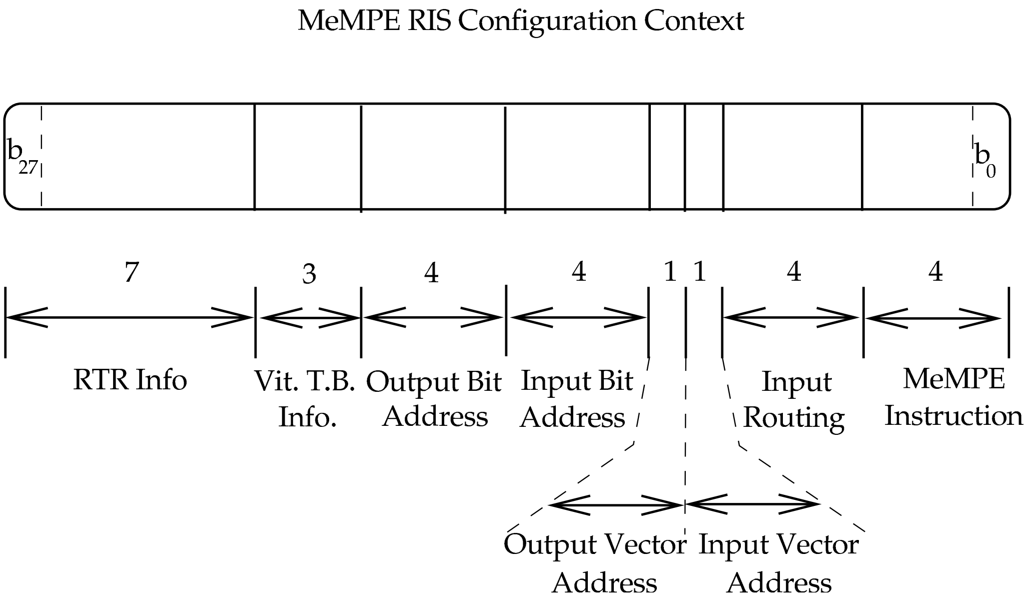

Figure C.1: Different Fields in the MeMPE configuration context.

The 4-bit MeMPE instructions are described in Table C.2 below. In addition to storage of data the MeMPE unit can perform other useful applications such as shifting and rotating as listed below.

Table C.1: MeMPE RIS context words description.

bits range	description
21-27	RTR info (see Appendix A)
18-20	Viterbi trace back update signal
14-17	Memory bit output address
10-13	Memory input bit address
9	Memory vector output address
8	Memory vector input address
4-7	External input vector routing address
0-3	MeMPE instruction

Table C.2: MeMPE instruction set.

Instruction	Description
0000	NOP (default)
0001	Load Vector
0010	Shift in Vector
0011	Shift BitIn left in all vectors
0100	Shif bit left throgu the complete memory with BitIn at the most left
0101	Shift BitIn left in only one vector
0110	Load BitIn left in only one vector
0111	Load BitIn left in all vectors
1000	Shift in the complete memory
1001	RoR Vector
1010	RoL Vector
1011	RoR Bit recursively in the complete memory
1100	RoL Bit recursively in the complete memory
1101	RoR Bit in the selected vector
1110	RoL Bit in the selected vector

Appendix D

The DSB RIS Unit

In this appendix the DSB RIS structure, its instructions as well as its stored context words are described. In the following the both the DSB and its RIS configuration are discussed.

Unlike the MeMPE and GALPE units the DSB does not perform any application rather than routing. The DSB RIS function is, therefore, management of the context words, their storage and their selection while the routing is naturally taken care of in the DSB itself.

The selection of the active context is slightly different than that of the GALPE and MeMPE RIS discussed in Appendix A. Here, the context number will be determined by the selected bit by the DSB if the 3 MSBs of the current context are 011 and for the other $0XX$ cases the control signal from the reconfiguration controller is selected. Similar to the description in Appendix A the RTR decision bit is selected by the addressed generated bits from the operational control when the MSB of the current context is 1.

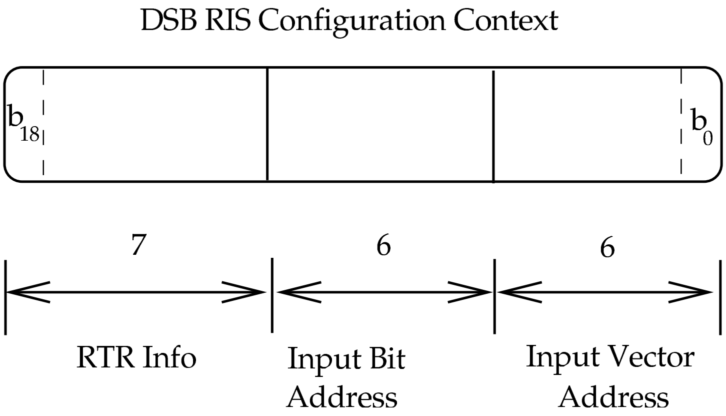

Figure D.1: Different Fields in the DSB configuration context.

Figure D.1 depicts the DSB's configuration context fields. Detailed listing of the different fields of the DSB context is given in Table D.1. The DSB RIS context word is 19 bits wide. Similar to the GALPE RIS and DSB RIS the most significant 7 bits are used for RTR purposes.

Table D.1: DSB RIS context words description.

bits range	description
18-12	RTR info (see Appendix A)
11-6	Bit address
5-0	Vector address.

No instructions are passed to the DSB since its only function is routing.

References

[1] http://www.vorbis.org.

[2] http://www.xiph.org.

[3] Actel. Axcelerator Family FPGAs. Technical Report v2.5, Actel, May 2005.

[4] ALTERA. Stratix II Device Handbook. Technical Report SII5V1-3.0, ALTERA, 2005.

[5] L. Azuara and P. Kiatisevi. Design of an Audio Player as System-on-a-Chip. Master's thesis, University of Stuttgart, 2002.

[6] V. Baumgarte, G. Ehlers, F. May, A. Nueckel, M. Vorbach, and M. Weinhardt. PACT XPP - A Self-Reconfigurable Data Processing Architecture. *Journal of Supercomputing archive*, 26(2):167 – 184, September 2003.

[7] J. Becker, R. Hartenstein, M. Herz, and U. Nageldinger. Parallelization in co-compilation for configurable accelerators-a host/accelerator partitioning compilation method. In *Proceedings of the Asia and South Pacific Design Automation Conference ASP-DAC '98.*, 1998.

[8] A. Berkeman, V. Owall, and M. Torkelson. A low logic depth complex multiplier using distributed arithmetic. *IEEE Journal of Solid-State Circuits*, 35:656 – 659, 2000.

[9] M. Biver, H. Kaeslin, and C. Tommansini. In-Place Updating of Path Metrics in Viterbi Decoders. *IEEE Jornal of Solid State Circuits*, 33(3):473–482, March 1989.

[10] K. Bondalapati. *Modeling and Mpping for Dynamically Reconfigurable Hybrid Architectures*. PhD thesis, University of Southern California, August 2001.

[11] V. Bondalapati, K.; Prasanna. Reconfigurable computing systems. *Proceedings of the IEEE*, 90:1201 – 1217, 2002.

[12] E. Boulton and L. Gonzalez. Trace Back Techniques Adapted to The Surviving Memory Management in the M Algorithm. In *Proceedings of the 2000 international Conference on Acostics, Speech and Signal processing (ICASSP2000)*, pages 3366–3369, 2000.

[13] S. D. Brown, R. J. Francis, J. Rose, and Z. G. Vranesic. *Field Programmable Gate Arrays*. Kluwer Academic Publishers, 1992.

[14] B. L. Bullough. Analysis of Field-Programmable Gate Array Implementations of Constant Coefficient Finite Impulse Responce Filters. Master's thesis, Birgham Young University, August 2002.

[15] K. Compton and S. Hack. Reconfigurable Computing: A Survey of Systems and Software. *ACM Computing Surveys*, 34(2):171–210, June 2002.

[16] J. W. Cooley and J. W. Tukey. An algorithm for Machine Calculation of Complex Fourier Siries. *Math. Computaion*, 19:297–301, 1965.

[17] A. DeHon. Reconfigurable Architectures for General Purpouse Computing. Technical Report Report no. ARTI 1586, MIT AI Lab, 1996.

[18] C. Deltoso, C. Johanblanq, M. Cand, and P. Senn. Fast Prototyping Based on Generic and Synthesizable VHDL Models a Case Study: Punctured Viterbi Decoders. In *IEEE Seventh International Conference on Rapid System Prototyping*, pages 158–163, 1996.

[19] T. J. Ding, J. V. McCanny, and Y. Hu. Synthesisable FFT Cores. In *IEEE Workshop on Signal Processing Systems*, pages 351–363, 1997.

[20] C. Ebeling, D. Cronquist, and P. Franklin. Configurable computing: the catalyst for high-performance architectures. In *Proceedings., IEEE Application-Specific Systems, Architectures and Processors*, 1997.

[21] A. El-Khashab and J. Swartzlander, E.E. A modular pipelined implementation of large fast Fourier transform. In *Signals, Systems and Computers*, 2002.

[22] R. H. et al. A Datapath Synthesis System for the Reconfigurable Datapath Architecture. In *Design Automation Conference, ASP-DAC'95*, 1995.

[23] G. Feygin, P. G. Gulak, and P. Chow. Generalized cascade Viterbi decoder-a locally connected multiprocessor with linear speed-up. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 2, pages 1097–1100, 1991.

[24] V. George and J. Hui Zhang andRabaey. The design of a low energy FPGA. In *International Symposium on Low Power Electronics and Design*, pages 188 – 193, 1999.

[25] V. S. Gierenz, O. Weiss, T. G. Noll, I. Carew, J. Ashley, and R. Karabed. A 550 Mb/s Radix 4 Bit-level Pipelined 16-State 0.25 mu/m CMOS Viterbi Decoder. In *IEEE International Conference on Applications-Specific Systems, Architectures and Processors*, pages 195–201, 2000.

[26] L. Gonzalez and E. Boulton. Simplified Path metric Update in the M Algorithm for VLSI Implementation. In *Proceedings of the 2000 international Conference on Acostics, Speech and Signal processing (ICASSP2000)*, pages 3366–3369, 2000.

[27] R. Hartenstein. Wozu Noch Mikrochips? *ITpress Verlag*, 1994.

[28] R. Hartenstein. The microprocessor is no longer general purpose: why future reconfigurable platforms will win. In *Second Annual IEEE International Conference on Innovative Systems in Silicon*, 1997.

[29] R. Hartenstein. A Decade of Reconfigurable Computing: a Visionary Retrospective. In *Design, Automation and Test in Europe*, pages 642 – 649, 2001.

[30] R. Hartenstein. Coarse grain reconfigurable architectures. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2001.

[31] R. Hartenstein. Trends in reconfigurable logic and reconfigurable computing. In *International Conference on Electronics, Circuits and Systems*, 2002.

[32] R. Hartenstein, J. Becker, M. Herz, and U. Nageldinger. A novel sequencer hardware for application specific computing. In *IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pages 392 – 401, 1997.

[33] R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinge. Using the KressArray for Configurable Computing. In *Conference on Configurable Computing: Technology and Applications*, 1998.

[34] R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger. Using the KressArray for Configurable Computing. In *Proceedings of SPIE Conference on Configurable Compting: Technology and Applications*, volume 3526, Nov. 1998.

[35] R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger. KressArray Xplorer: a New CAD Environment to Optimize Reconfigurable Datapath Array Architectures. In *Asia and South Pacific Design Automation Conference*, pages 163 – 168, 2000.

[36] R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger. KressArray Xplorer: a new CAD environment to optimize reconfigurable datapath array architectures. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2000.

[37] R. Hartenstein and R. Kress. A datapath synthesis system for the reconfigurable datapath architecture. In *IFIP International Conference on Hardware Description Languages*, 1995.

[38] S.-F. Hasio, W.-R. Shiue, and J.-M. Tsing. A Cost Effcient Fully-Pipelined Architecture for DCT/IDCT. *IEEE Transactions on Consumer Electronics*, 45(3):515–525, August 1999.

[39] S. Haykan. *Communication Systems*. Wiley, 1994.

[40] S. He and M. Torkelson. A new approach to pipeline FFT processor. In *The 10th International Parallel Processing Symposium*, pages 766 –770, 1996.

[41] S. He and M. Torkelson. Design and Implementation of a 1024-point Pipeline FFT Processor. In *IEEE Custom Integrated Circuits Conference*, pages 131 –134, 1998. MyPhDFFT4.

[42] S. He and M. Torkelson. Designing pipeline FFT processor for OFDM (de)modulation. In *URSI International Symposium on Signals, Systems, and Electronics*, 1998.

[43] M. Herz. *High Performance Memory Communication Architectures for Coarse-grained Reconfigurable Computing Systems*. PhD thesis, Universitaet Kaiserslautern, 2001.

[44] M. Herz, T. Hoffmann, U. Nageldinger, and C. Schreiber. Interfacing the MoM-PDA to an Internet-based development system. In *Annual Hawaii International Conference on System Sciences*, 1999.

[45] S.-F. Hsiao, W.-R. Shiue, and J.-M. Tseng. A cost-efficient and fully-pipelinable architecture for DCT/IDCT. In *IEEE Transactions on Consumer Electronics*, pages 515–525, 1999.

[46] S.-F. Hsiao, W.-R. Shiue, and J.-M. Tseng. Design and implementation of a novel linear-array DCT/IDCT processor with complexity of order log2N. In *IEE proceedings Vision, Image and Signal Processing*, pages 400–408, 2000.

[47] S.-F. Hsiao, W.-R. Shiue, and J.-M. Tseng. Design and implementation of a novel linear-array DCT/IDCT processor with complexity of order log2N. *IEE Proceedings- Vision, Image and Signal Processing*, 147:400 – 408, 2000.

[48] Z. Huang, S. Malik, N. Moreano, and G. Araujo. The design of dynamically reconfigurable datapath coprocessors. *ACM Transactions on Embedded Computing Systems (TECS)*, 3(2):361–384, May 2004.

[49] L. Jia, Y. Gao, J. Isoaho, and H. Tenhunen. Design of A Super-Pipelined Viterbi Decoder. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pages I133–I136, 1999.

[50] H. Kalte, D. Langen, E. Vonnahme, A. Brinkmann, and U. Ruckert. Dynamically reconfigurable system-on-programmable-chip. In *Euromicro Workshop on Parallel, Distributed and Network-based Processing*, 2002.

[51] I. Kang and A. W. Jr. Low Power Viterbi Decoder for CDMA Mobile Terminals. *IEEE Jornal of Solid State Circuits*, 24(4):1158–1160, August 1989.

[52] P. Kelly and P. Chau. A Flexible Constraint Length, Folfable Viterbi Decoder. In *IEEE Global Telecommunications Conference Including a Communications Theory Mini-Conference*, volume 1, pages 631–635, 1993.

[53] J. I. L. Jia, Y. Gao and H. Tenhunen. A New VLSI Oriented FFT Algorithm and Implementation. In *Proceedings of the Eleventh Annual IEEE International ASIC Conference*, pages 337 –341, 1998.

[54] B. P. Lathi. *Modern Digital and Analog Communication Systems*. Holt, Rinehart and Witson, second edition edition.

[55] K. Leijten-Nowak. *Template-Based Embedded Reconfigurable Computing*. PhD thesis, Technische Universiteit Eindhoven, 2004.

[56] G. Lu. *Modling, Implementation and Scalability of the MorphoSys Dynamically Reconfigurable Computing Architecture*. PhD thesis, University of California, Irvine, 2000.

[57] G. Lu, H. Singh, M.-H. Lee, N. Bagherzadeh, F. Kurdahi, E. Filho, and V. Castro-Alves. The MorphoSys Dynamically Reconfigurable System-on-Chip. In *Proceedings of the First NASA/DoD Workshop on Evolvable Hardware*, pages 152 – 160, July 1999.

[58] T. A. M. Hasan and J. Thompson. Scheme for reducing coefficient memory in FFT processor. In *Electronic Letters*, volume 38, pages 163 –164, Feb 2002.

[59] T. A. M. Hasan and J. Thompson. A delay spread based low power reconfigurable FFT processor architecture for wireless receiver. In *Proceedings of the IEEE International Symposium on System-on-Chip*, pages 135 –138, 2003.

[60] R. Maestre, M. Fernandez, R. Hermida, Bagherzadeh, F. Kurdahi, and H. Sing. A Framework for Reconfigurable Computing: Task Sheduling and Context Management. *IEEE Transactions on Very Large Scale Integration Systems*, 9(6):858–873, 2001.

[61] D. Manners and T. Makimoto. *Living with the Chip*. Chapmann & Hall, 1995.

[62] E. Mirsky and A. DeHon. MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resource. In *. IEEE Symposium on FPGAs for Custom Computing Machines*, pages 17–19, April 1996.

[63] E. A. Mirsky. Coarse-Grain Reconfigurable Computing. Master's thesis, Massachusetts Institute of Technology, May 1996.

[64] S. Mujtaba. An Area-Efficient Architecture of the Viterbi Decoder for Reverese Link IS-95 (CDMA) Interface. In *The Forth International Conference on Signal Processing*, pages 525–528, 1998.

[65] S. Mujtaba. An area Efficient VLSI architecture of the Viterbi Decoder for reverse link IS-95 CDMA. In *IEEE Forth International Conference on Signal Processing*, volume 1, pages 525–528, 1998.

[66] J. Nikara. *Application-Specific Parallel Structures for Discrete Cosine Transform and Variable Lenght Decoding*. PhD thesis, Tempre University of Technology, 2004.

[67] J. Nikara, J. Takala, D. Akopian, and J. Saarinen. Pipeline Architecture for DCT/IDCT. In *IEEE International Symposium on Circuits and Systems (ISCAS 2001)*, pages 902–905, 2001. MyPhDDCT4.

[68] M. Nivioja, J. Isoaho, and L. V. Design and implementation of Viterbi Decoder with FPGA. *Jornal of VLSI Signal Processing*, 21:5–14, 1999.

[69] PACT. The XPP White Paper. Technical Report 2.1.1, PACT, March 2002.

[70] PACT. Smart Media Processing with XPP. Technical Report ver.1, PACT, April 2003. rev.1.

[71] PACT. Reconfiguration on XPP-IIb Cores. Technical Report ver.1.0.1, PACT, March 2005.

[72] J. Palicot and C. Ronald. FFT: a basic function for a reconfigurable receiver. In *IEEE 10 th International Conference on Telecommunications*, pages 898 –902, 2003.

[73] P. Pandita and S. K. Roy. Design and Implementation of a Viterbi decoder Using FPGAs. In *IEEE Twelvth International Conference on VLSI Design*, pages 611–614, 1999.

[74] K. K. Parhi. *VLSI Digital Signal Processing Systems, Design and Implementation*. WILEY Interscience, 1999.

[75] L. Perez. *Architectures VLSI pour le Codage Conjoint Source-Canal en Trellis*. PhD thesis, Ecole Nationale Superieure des Telecommunications, 2000.

[76] J. Rabaey. Reconfigurable processing: the solution to low-power programmable DSP. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, pages 275 – 278, April 1997.

[77] J. Rabaey. Hybrid reconfigurable processors-the road to low-power consumption. In *Eleventh International Conference on VLSI Design*, pages 300 – 303, Jan. 1998.

[78] J. Rabaey, A. Abnous, Y. Ichikawa, K. Seno, and M. Wan. Heterogeneous reconfigurable systems. In *EEE Workshop on Signal Processing Systems*, pages 24 – 34, Nov. 1997.

[79] L. R. Rabiner and B. Gold. *Theory and Application of Digital Signal Processing*. Prentice-Hall, 1975. DSP book with the FFT pipelined architectures.

[80] B. Radunovic. An overview of advances in reconfigurable computing systems. In *Proceedings of the 32nd Annual Hawaii International Conference on System Sciences*, pages 1–10, Jan. 1999.

[81] H. Sing, M.-H. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh, and F. Chaves. MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Iintensive Applications. *IEEE Transactions on Computers*, 49(5):456–481, 2000.

[82] H. Singh. *Reconfigurable Architectures for Multimedia and Data-Parallel Application Domains*. PhD thesis, University of California, Irvine, 2000.

[83] S. Sriam, R. Tessier, D. Goeckel, and W. Burleson. A Dynamically Reconfigurable Adaptive Viterbi Decoder. In *Tenth ACM International Symposium on Field Programmable Gate Arrays*, pages 227–236, 2002.

[84] S. Sridharan and L. Carley. A 110 MHz 350 mW 0.6 μm CMOS 16-State Generalized Target Viterbi Drcoder. *IEEE Jornal of Solid State Circuits*, 35(3):362–370, March 2000.

[85] R. Storn. Radix-2 FFT-pipeline architecture with reduced noise-to-signal ratio. *IEE Proceedings- Vision, Image and Signal Processing*, 141:81 – 86, 1994.

[86] J. Takala, D. Akopian, J. Astola, and J. Saarinen. Constant Geometry Algortim for discrete Cosine Transform. *IEEE Transactions on Signal Processing*, 48(6):1840–1843, June 2000.

[87] J. Takala, J. Nikara, D. Akopian, J. Astola, and J. Saarinen. Pipeline architecture for 8X8 discrete cosine transform. In *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '00)*, pages 3303–3306, 2000. MyPhDDCT3.

[88] J. Takala, J. Nikara, and K. Punkka. Pipeline Architecture for Two Dimentional Discrete Cosine Transform and Its Inverse. In *IEEE International Conference on Electronics, Circuits and Systems (ISCAS 2002)*, pages 947–950, 2002. MyPhDDCT5.

[89] E. Tau, I. Eslick, D. Chen, J. Brown, and A. DeHon. A First Generation DPGA Implementation. In *n Proceedings of the Third Canadian Workshop on Field- ProgrammableDevices*, page 138â143, May 1995.

[90] M. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The Raw microprocessor: a computational fabric for software circuits and general-purpose programs. *IEE Micro*, 22:25 – 35, 2002.

[91] R. Tessier and W. Burleson. Reconfigurable Computing for Digital Signal Processing: A Survey. *Journal of VLSI Signal Processing, Kluer Academic Publishers*, 28:7–27, 2001.

[92] N. Tredennick. Technology and Buisness: Forces Driving Microprocessor Evolution. volume 83,12, Dec 1995.

[93] N. Tredennick. The Case of Reconfigurable Computing. Microprocessor Report 10, 10, Aug 1996.

[94] T. K. Truong, M. Sih, I. Reed, and E. H. Satorius. A VLSI Design for A Trace-Back Viterbi Decoder. *IEEE Jornal of Solid State Circuits*, 35(3):362–370, March 2000.

[95] C. Tsui, R. Cheng, and C. Ling. Low Power ACS Unit Design for the Viterbi Decoder. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pages I137–I140, 1999.

[96] A. J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, IT-13:260–269, April 1967.

[97] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it All to Software: RAW Machines. *IEEE Computer*, 30(9):86 – 93, Sept 1997.

[98] M. Wan, Y. Ichikawa, D. Lidsky, and J. Rabaey. An energy conscious methodology for early design exploration of heterogeneous DSPs. In *Proceedings of the IEEE, Custom Integrated Circuits Conference*, pages 111 – 117, May 1998.

[99] M. Wan, H. Zhang, M. Benes, and J. Rabaey. A low-power reconfigurable data-flow driven DSP system. In *IEEE Workshop on Signal Processing Systems*, pages 191 – 200, Oct. 1999.

[100] W. Weng and W. Yang. The CPLD Implementation of Viterbi Algorithm in Grand Alliance ATCS System. *IEEE Transactions on Industrial Electronics*, 48(5):898–903, October 2001.

[101] XILINX. Virtex-4 User Guide. Technical Report UG070 (v1.3), XILINX, April 2005.

List of Publications

[102] M. Glesner, H. Hinkelmann, T. Hollstein, L. Indrusiak, T. Murgan, A. M. Obeid, M. Petrov, T. Pionteck, and P. Zipf. Reconfigurable Embedded Systems: An Application-Oriented Prespective on Architectures and Design Techniques. In *Embedded Computer Systems: Architectures, Modeling and Simulation, 5th International Workshop, SAMOS 2005*, volume 3553, pages 12–21. Springer, July 2005.

[103] T. Murgan, A. M. Obeid, A. Guntoro, P. Zipf, M. Glesner, and U. Heinkel. Design and Implementation of a Multi-Core Architecture for Overhead Processing in Optical Transport Networks. In *Reconfigurable Communication-centric SoCs (ReCoSoC) Workshop*, June 2005.

[104] A. Obeid, T. Murgan, A. Taadou, and M. Glesner. HW/SW Design and Realization of a Size-Reconfigurable DCT Accelerator. In *12th IEEE International Conference on Electronics, Circuits and Systems*, 2005.

[105] A. M. Obeid, A. Garcia, and M. Glesner. A Constraint Length and Throughput Parameterizable Architecture for Viterbi Decoders. In *The 16th International Conference on Microelectronics (ICM 04)*, 2004.

[106] A. M. Obeid, A. Garcia, R. Ludewig, and M. Glesner. Prototyping of A High Performance Generic Viterbi Decoder. In *IEEE 13th International Conference on Rapid System Prototyping*, pages 42–47, 2002.

[107] A. M. Obeid, A. Garcia, M. Petrov, and M. Glesner. A Multi-Path High Speed Viterbi Decoder. In *10th IEEE International Conference on Electronics, Circuits and Systems*, 2003.

[108] A. M. Obeid, A. G. Ortiz, and M. Glesner. A Parameterizable Constraint Length and Throughput Architecture for Viterbi Decoders. *Elsvier Microelectronics Journal*, (accepted).

[109] M. Petrov, T. Murgan, A. Obeid, C. Chitu, P. Zipf, J. Brakensiek, and M. Glesner. Dynamic Power Optimization of the Trace-Back Process for the Viterbi Algorithm. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, 2004.

[110] M. Petrov, A. Obeid, T. Murgan, and et al. An Adaptive Trace-Back Solution for State-Parallel Viterbi Decoders. In *12th IFIP International Conference on Very Large Scale Integration*, 2003.

Supervised Theses

[111] S. Azzam. Synthese ausgewaehlter DSP-Algorithmen auf einer hybriden grobgranularen rekonfigurierbaren Architektur. Bachelorarbeit, Darmstadt University of Technology, Aug. 2005.

[112] M. Barzan. Architecture and Design for FIR Filter and FFT Accelerators. Master thesis, Darmstadt University of Technology, Jul. 2004.

[113] H. Chokr. Entwicklung einer rekonfigurierbaren Pipeline-architektur FFT/DCT. Bachelorarbeit, Darmstadt University of Technology, 11 2004.

[114] I. Gercek. Realization and Verification of a Low-Power Viterbi Decoder. Diplomarbeit, Darmstadt University of Technology, Oct. 2005.

[115] R. Hartmann. FPGA-Prototyping von Komponenten eines HiperLAN/2 Empfaengers. Studienarbeit, Darmstadt University of Technology, May 2002.

[116] W. Li. Development of an Embedded Coprocessor for Compact Flash Interfacing. Diplomarbeit, Darmstadt University of Technology, 2005.

[117] Y. Li. Entwicklung und Implementierung Eines M algorithm Viterbi Decoder. Studienarbeit, Darmstadt University of Technology, Feb. 2005.

[118] A. Taadou. Hardware/Software Codesigh eines Ogg Vorbis Players. Diplomarbeit, Darmstadt University of Technology, March 2005.

[119] P. Vajravelu. Development and Optimization of a Leon-based DSP Accelerator. Masterarbeit, Darmstadt University of Technology, Oct. 2005.

Lebenslauf

Zur Person:	
Geburtsdatum:	13.03.1971
Geburtsort:	Riad, Saudi Arabien
Ausbildung:	
1988 bis 1994	Gymnasium "Jesus Maria" und "Hermanos Maristas" Abschluß: Abitur
1994 bis 1996	Studium der Nachrichtentechnik an der Polytechnischen Universität Valencia (Vertiefungsrichtung Elektrotechnik)
1994	Abschluß: B. S. in Electrical Engineering
1994 bis 1996	Wissenschaftlicher Mitarbeiter am Electronics and Computers Research Institute der King Abdulaziz City for Science and Technology in Riad, Saudi Arabient
1996 bis 1999	Masterstudium der Elektrotechnik am Michigan State University in East Lansing, MI, USA
1999	Abschluß: M. S. in Electrical Engineering
1999 bis 2000	Wissenschaftlicher Mitarbeiter am Electronics and Computers Research Institute der King Abdulaziz City for Science and Technology in Riad, Saudi Arabient
seit 01.09.2000	Doktorand am FG Mikroelektronische Systeme der Technischen Universität Darmstadt
Beruflicher Werdegang:	
1994 bis 1996 & 1999 bis 2000	Wissenschaftlicher Mitarbeiter am Electronics and Computers Research Institute der King Abdulaziz City for Science and Technology in Riad, Saudi Arabient
seit 01.09.2000	Wissenschaftlicher Mitarbeiter am FG Mikroelektronische Systeme der Technischen Universität Darmstadt