# Chapter 7.

# Simulation and Design of OFDM Receivers

## Contents

**7.1. Simulation of OFDM Systems . . . 161**
7.1.1. An Overview of the IEEE 802.11a Standard . . . 162
7.1.2. Untimed Dataflow Modeling . . . 163
7.1.3. Functionality Embedding and Inter-Module Communication . . . 167
7.1.4. Symbol-Based Token Transfers . . . 169
**7.2. Design of an FFT Block for OFDM Systems . . . 170**
7.2.1. FFT Architectures Review . . . 170
7.2.2. FFT in OFDM Applications . . . 174
7.2.3. Advances in FPGA and Synthesis Technology . . . 176
7.2.4. Implementation Details . . . 177

## 7.1. Simulation of OFDM Systems

This section presents a systematic approach to modeling and simulating an OFDM physical layer (PHY) transceiver using SystemC. On the one hand, it shows the problems associated with using pure untimed dataflow models, suggesting different solutions for circumventing them and add run-time control features to modules. On the other hand, it proposes a method for modeling latency (add timing information) with minimal overhead on the model complexity. For both the timed and the untimed dataflow models, two approaches to transferring data between modules are presented: sample-based and symbol-based, which results in four possible modeling scenarios.

In order to illustrate our ideas, we have chosen a real application, a transceiver for the OFDM-based IEEE 802.11a wireless LAN standard. However, the insights presented here are also

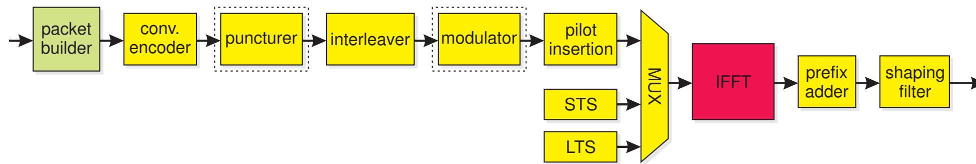


Figure 7.1.: Structure of an IEEE 802.11a transmitter

applicable to other standards employing OFDM, either for wireless networking or for broadcasting. As modeling language we have selected SystemC [67, 68] because it is free, it simulates relatively fast and allows modules described at different levels of abstraction to coexist, which makes it ideal for architectural exploration and refinement.

The section begins with a short overview of the IEEE 802.11a standard, with emphasis on those features which are of special interest for modeling the transceiver chain. We then describe the modeling of the chain using a sample-based untimed dataflow approach. A special attention is given to embedding control information in a pure dataflow simulation using control flags that accompany data tokens, targeted specifically to OFDM systems. A discussion on functional modeling for maximum reusability is presented next, by separating the functionality of a module from the communication with other modules.

We then go on to present a flexible method for adding clocks and timing information to the modules (latencies), with minimal impact on the model complexity. If the modeling style from the previous section is used, most of the code from the untimed models can be reused. Furthermore, in order to increase the simulation speed, a symbol-based approach (both untimed and timed) is presented in the next section, which complements the classic sample-based approach.

### 7.1.1. An Overview of the IEEE 802.11a Standard

The IEEE 802.11a standard [38] defines an OFDM-based wireless LAN that supports raw data rates between 6 and 54 Mbps in the 5 GHz band. The block structure of the transmitter is shown in **Figure 7.1**. Multiple data rates are achieved by employing two puncturing patterns (of rate 2/3 and 3/4) in addition to the basic convolutional code of rate 1/2, as well as by using four different modulation schemes (BPSK, QPSK, 16-QAM, 64-QAM). Of the twelve possible combinations, only eight are defined by the standard.

As with any OFDM system, the data stream is organized in symbols, which are converted to time domain using IFFT, then appended a cyclic prefix. The adjacent time-domain cyclic-prefixed symbols are passed through a shaping filter and sent to the up-converter for the 5 GHz band. In 802.11a, a symbol has 64 samples, while the cyclic prefix is 1/4 of the symbol length. The sampling period is 50 ns, resulting in a data rate of 16 Msps for the IFFT, which is a characterizing parameter for an OFDM system.

The payload data to be transmitted (max. 4096 bytes) is received from the upper MAC layer,

together with the desired data rate and transmission power. Each transmitted packet starts with two short training symbols (STS), followed by two long training symbols (LTS). The packet builder block appends a SIGNAL field to the payload, which contains the data rate used for encoding the payload and the length thereof. The SIGNAL field is always encoded using the lowest rate, regardless of the selected rate for the payload. **Figure 7.2** shows the structure of the packet.

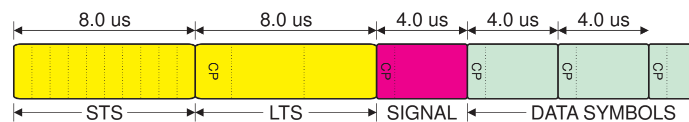


Figure 7.2.: IEEE 802.11a packet structure

The receiver will first detect the packet start, perform time and frequency synchronization, and extract the symbols from the received data stream by using STS and LTS. After converting the symbols back to frequency domain using FFT, a channel estimation and equalization is performed to compensate for the channel frequency selectivity. Further processing steps are inverses of the corresponding operations in the transmitter. The structure of the implemented receiver is shown in **Figure 7.3**.

The fact that different data rates have to be supported makes the flow control between modules an important implementation issue. Using multiple clock domains and inter-module asynchronous FIFO's is one of the possible solutions. On the receiver side, the complete SIGNAL symbol has to be decoded in order to know the data rate with which the following symbols are encoded. Since the first data-rate-dependent block is the soft demodulator, a large FIFO will be required at its input to buffer the incoming data until the rate has been decoded. The size of this FIFO will depend on the latency of the data path between demodulator and packet extractor.

## 7.1.2. Untimed Dataflow Modeling

The first step in modeling a system is usually to start with a first order untimed description of the system, based on the initial specifications. A very general solution is to use Kahn process

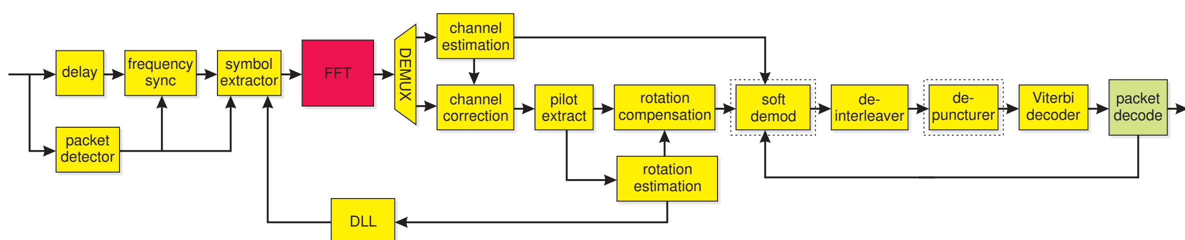


Figure 7.3.: Structure of an IEEE 802.11a receiver

networks (KPN) [47], which is an effective method for modeling signal processing algorithms for communication applications. In this model of computation (MOC), processes execute concurrently and communicate through infinite FIFO channels, using atomic data units called tokens. Processes write to and read tokens from FIFO's using blocking functions. Moreover, there is no concept of time. The functionality in each process is separated into three stages: reading a number of input tokens, processing them, then writing the resulting tokens to the output. The Ptolemy framework [76] is an example of a modeling environment which supports this MOC natively.

In SystemC, we can use `sc_fifo` channels and `SC_THREAD` processes to model KPN. SystemC does not support infinite FIFO's, therefore explicit upper bounds of the FIFO sizes have to be specified. This particular case of KPN is known as untimed dataflow modeling. If token generation and consumption are not balanced on average, FIFO's can become full and/or empty and the blocking read/write operations can lead to simulation stalls through starvation (lack of events) [27]. More information about dataflow modeling can be found in [53] and [47].

Untimed dataflow modeling is also used by Synopsys CoCentric System Studio, when using *PRIM* models. In fact, an initial implementation of our 802.11a chain has been done using this environment. Unfortunately, a number of limitations have led us to abandon it in favor of the free SystemC reference solution [67]. System Studio supports both static and dynamic dataflow modeling. If the number of tokens that a process will read and write each time is known at compile time, the model is said to be static. Upon compilation, the tool will analyze the network and create static execution schedules for processes. Moreover, FIFO sizes can be computed at compile-time. The simulation will execute much faster since dynamic scheduling is avoided.

In SystemC, the scheduler is not aware and does not take advantage of the fact that some modules may employ static dataflow. The processes are independent threads which access FIFO's using blocking read and write methods. Synchronization is implicit, threads being automatically suspended and resumed, depending on the status of the FIFO channels. This guarantees that no tokens get lost.

As long as a pure dataflow simulation is all that is needed, modeling is straightforward. A number of tokens are read from the input FIFO, processed, then the resulting tokens written to the output FIFO. As soon as we want to add some control capabilities, the limitations of this modeling methodology become evident. In the 802.11a transmitter for instance, at least the puncturer, the interleaver, and the modulator need to know the data rate with which each symbol is encoded, in order to select the appropriate mode of operation. In order to determine the data rate of each symbol, these modules consume first a token from the control channel. Once they have this information, they know how many samples have to be read from the input data FIFO for the current symbol, using a look-up table. We will denote the data-rate token with RATE.

For each module, care must be taken to ensure that the number of control tokens is equal to the number of symbols, or else the simulation will stall. We have experimented with two solutions

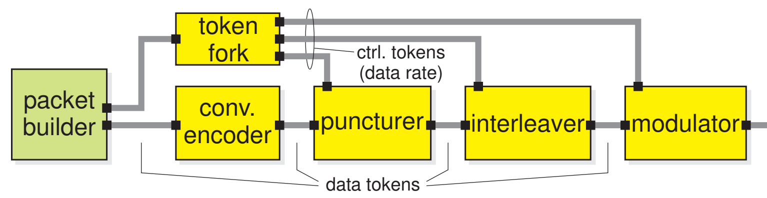


Figure 7.4.: Distributing controls tokens through a *fork* module

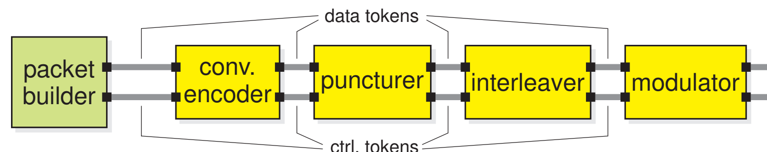


Figure 7.5.: Point-to-point forwarding of controls tokens

to this problem. In both approaches, the control tokens are generated by the packet builder block. Since this is where the symbols are created, it is easy to ensure that a control sample is produced for each symbol. In the first approach, the control tokens (data-rates) are distributed to all modules that require to know the data-rate using a fork module, as shown in **Figure 7.4**. A parameterizable fork module has been created, which takes the number of replicated output streams and the token type as template parameters.

In the second approach, the control tokens are passed from one module to the following, in the same manner as the data tokens. This resembles the *synchronous piggy-backed dataflow* (SPBD) model proposed in [69] and has the advantage of maintaining the point-to-point nature of the chain. Besides data token processing, each module is now responsible of forwarding the control token to the next module. The disadvantage is that even the most simple modules, e.g. convolutional encoder, need to be aware of the symbol-oriented nature of the data flow and keep a table with the symbol sizes for all possible data rates, which adds unwanted complexity to the modules. The principle is illustrated in **Figure 7.5**. Besides data rates, other control tokens are flags like first-symbol (FSYM) and last-symbol (LSYM). FSYM is used to control the training symbol multiplexer, while LSYM can be used by the modules to know when they can enter a low-power mode.

We propose therefore to pass control information locally, without the need of a central state machine. **Figure 7.6** shows the symbol sizes for all FIFO channels in the sender, for all possible data rates. An additional benefit of such a table is that it presents a clear overview on the inter-module communication requirements requirements in all stages of the design. The peak and average data rate is an important parameter in selecting an appropriate IP core. Using the values in the table, the absolute data rates can be determined by a multiplication with the symbol rate, which is a fixed parameter defined in the standard. For 802.11a, the symbol rate is 250 KSym/s.

The above solutions tend to complicate the design and are prone to simulation stalls if the number of control tokens does not match the number of symbols. The models are also complicated

| channel (block output) | data rate index | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | |
| **packet_builder** | 24 | 36 | 48 | 72 | 96 | 144 | 192 | 216 | bits |
| **encoder** | 48 | 72 | 96 | 144 | 192 | 288 | 384 | 432 | bits |
| **puncturer** | 48 | | 96 | | 192 | | 288 | | bits |
| **interleaver** | 48 | | 96 | | 192 | | 288 | | bits |
| **modulator** | 48 | | | | | | | | samples |
| **pilot_insert** | 52 | | | | | | | | samples |
| **ifft** | 64 | | | | | | | | samples |
| **prefix_add** | 80 | | | | | | | | samples |

Figure 7.6.: Symbol sizes for various blocks of the transmitter

due to the fact that each module has to keep track of the symbol sizes for the data it processes. A better solution is to have control information accompany each data token (sample). This can be modeled by creating a new structure type like the one shown in **Listing 7.1**, which encapsulates the data token type together with an integer and a vector of Boolean flags. Such structures will be transmitted as tokens between modules. In order for the new data type to be used as tokens in SystemC, at least the `<<` streaming operator and the `sc_trace` function had to be overloaded.

Listing 7.1: Tagged data class

```
1  enum flag_t {
2    SOS = 0, // start of symbol
3    EOS,     // end of symbol
4    FSYM,    // first symbol
5    LSYM,    // last symbol
6    EN,      // enable
7    MAX_BITS
8  };
9
10 template <class T, int BITS = MAX_BITS> class tagged
11 {
12 public:
13   tagged(); // default ctor
14   tagged(T value); // initializing ctor
15   tagged(const tagged<T>& value); // copy ctor
16
17   bool operator==(const tagged& rhs) const;
18   tagged& operator=(const tagged& rhs);
19
20 public:
21   T            data; // main data
22   bitset<BITS> bits; // boolean flags
23   int          tag;  // for RATE, etc.
24   string       text; // for debug info
25 };
```

This resembles more closely the way real hardware operates, as the additional control information can be thought of as tags to the data samples. The modules will no longer have to keep track of the symbol size, but instead the symbol boundary samples will have to be marked accordingly. For this purpose, we added the start-of-symbol (SOS) and end-of-symbol (EOS) tags, which qualify individual samples. On the other hand, FSYM, LSYM, and RATE qualify a symbol and will not change throughout the symbol. The modules will now have to parse the

token stream to identify symbol boundaries.

A problem we encountered here was to devise a way to parse the input stream to identify symbol boundaries. The most efficient solution was to use a one-token look-ahead in the input stream, i.e. we need to be able to read the first token available, without extracting it from the stream. Unfortunately, the existing `sc_fifo` channels in SystemC do not provide such a feature. That is why we had to extend the interface of the `sc_fifo` class with a `peek()` function, which supplies the missing feature. At this point we also extended the `sc_fifo` class to write every sample to a data dump file, which is extremely useful for debugging purposes. This feature is only active when the program is compiled with the `DEBUG` option.

### 7.1.3. Functionality Embedding and Inter-Module Communication

In order to increase the reusability factor, the approach is to separate the functionality of a module from the inter-module communication. The natural solution offered by the object oriented nature of C++ is to encapsulate the functionality of the modules in classes, which can then be reused among modules that employ different inter-module communication strategies. The behavior of a class is accessed through member functions, which can be roughly divided into three categories: 1) data input/output, 2) computation, and 3) (re)configuration.

If properly designed, the same class can be instantiated in the SystemC modules as they evolve through the communication refinement process. Furthermore, a class can be even reused when designing a new communication chain for another standard, if enough configuration parameters are provided. For example, the Viterbi decoder class we designed is completely configurable, supporting different constraint lengths, polynomials, trace-back window lengths, as well as different operating modes, such as stream or block mode.

In general, there are three methods for configuring a class: template parameters, constructor parameters, and configuration functions. Using template parameters, the class is parameterized upon instantiation at compile-time. Using constructor parameters, the configuration is defined at run-time. However, once simulation started, it cannot be changed anymore. If full run-time configuration is desired (reconfiguration), the class should provide a special configuration function to be called during simulation.

From an inter-module communication point of view, modules belong to different categories: sample-based with memory (convolutional encoder, filters) or without memory (CORDIC), block-based (FFT, channel estimation), or mixed (symbol extractor, channel correction, Viterbi decoder). This is a very important aspect that has to be considered when designing the communication profiles.

The first step towards refinement is to create a timed functional model, or performance model, obtained by annotating the initial untimed functional model with delays, which can take any desired value. We have therefore a concept of time, but still no clocks. In our modeling, we

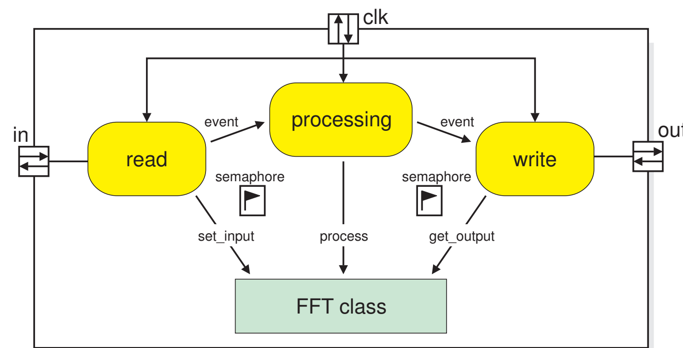


Figure 7.7.: Three-process clocked model

will skip this step and go directly to the next, which is to consider the system sequential by adding clocks. In order to ensure the generality of the discussion, each module is provided with a separate clock. The latency will now be expressed in clock cycles.

In the case of the untimed models, the input read, processing, and output write tasks were preformed sequentially, which is reasonable as long as there is no concept of time. In a real circuit, however, these three processes (`read`, `processing`, and `write`) normally execute in parallel. All modules that have a clock will need to model this parallelism. Our approach is to use three SystemC processes which communicate through events and share common data through semaphores, as shown in **Figure 7.7**.

One semaphore is associated with the input data and one with the output data. When the `read` process is reading data, it takes ownership of the input semaphore. In the case of the 64-sample FFT used in the 802.11a standard, the read process will read 64 data samples from the input data stream and call the `set_input` function of the `FFT` class for each sample read. Once the 64 samples have been read, the `read` process issues an event to notify the `processing` process that new data is available, at the same time releasing the semaphore.

When the `processing` process is triggered by that event, it will take the ownership of both semaphores and call the processing function of the class, which executes in zero time from a simulation point of view. Then, in order to model the latency, it waits for a number of clock cycles, given as a parameter. Once completed, the process releases the semaphores and issues an event to notify both the `read` and the `write` process. When notified, the `write` process will claim the output semaphore and write the FFT results to the output, releasing the semaphore after completion.

Modeling the latency accurately is crucial, since the latency of the receiver/transmitter pair is an important performance metric of a wireless LAN, affecting the response time to a received packet. The main contributors to the latency in an OFDM receiver are the FFT and the Viterbi decoder, which are also the most computationally intensive and take up most of the silicon area.

### 7.1.4. Symbol-Based Token Transfers

When modeling a communication chain, there are usually two directions of interest: communication refinement or functional simulation only. Consequently, there are two different strategies to pursue. For refinement purposes, we need to model the control information as accurately as possible, while for functional simulation only, the simulation speed is the parameter of concern. As we already dealt with optimizing the chain for communication refinement in **Subsection 7.1.2**, we will now show how the simulation efficiency can be increased.

In SystemC, transferring data through a channel, such as `sc_fifo`, involves two assignment operators (one at the producer for writing, one at the consumer for reading) and a copy constructor (inside the channel). Moreover, if the FIFO is full when a blocking write occurs, the producer task the write function has been called from will be suspended (context switch) until data is read from the FIFO by the consumer. The same happens if the FIFO is empty on a blocking read. Suspending and resuming a task incurs an execution time penalty, which can affect the simulation speed significantly if context switches occur too often.

In order to increase the simulation speed, the number of tokens to be transferred has to be minimized. Since data in OFDM systems is organized in symbols, it is therefore more efficient to transfer whole OFDM symbols as tokens instead of transferring individual samples. Thus, the number of tokens in the system is decreased significantly. The more samples in an OFDM symbol, the less symbols are needed for a given data rate. The simulation speed-up can be significant for systems featuring large symbols, such as DVB-T (2048 or 8192 samples/symbol) [20].

As with the sample-based approach, each token is a tagged data structure. In this case, the data member of the structure is a vector of dynamically allocated data samples. However, unlike the sample-based approach, where each data sample is accompanied by its own tag vector, only one tag vector is now required for the whole symbol. This reduction in data size further contributes to reducing the simulation time. The sample-level `SOS` and `EOS` tags are not needed anymore, only `FSYM`, `LSYM`, and `RATE` need to be kept.

A further solution for accelerating the simulation is to transfer pointers to the symbol data structure instead of transferring the structure itself. Thus, a symbol is dynamically allocated and its elements populated by the producer process, then the pointer to the symbol is sent through the FIFO channel. The consumer process receives the pointer, reads the data and ultimately frees the allocated memory to prevent leaks. This technique poses some serious risks. For instance, if we have a fork module which duplicates the data flow, both consumers connected at its output would attempt to deallocate the same memory region, which results in crashing the simulation. If such a technique is to be employed, the designer has to avoid using fork modules, i.e. communication must be point-to-point.

An additional benefit of the symbol-based approach is that the modules no longer need to parse the incoming data stream to identify the symbol boundaries. Their design and maintenance

is thus simplified. Timing information can still be added, as presented in the previous section. One disadvantage is, however, that the resulting chain no longer lends itself to further communication refinement. Depending on the particular application, the system designer can choose between the sample-based approach, which is more suitable for refinement, and the symbol-based approach, which simulates faster.

## 7.2. Design of an FFT Block for OFDM Systems

The FFT is the core of any OFDM system, accounting together with the Viterbi FEC decoder for the highest percentage of area and power consumption in an OFDM receiver. In this section we examine four OFDM standards with regard to their FFT computational requirements and investigate various FFT architectures with emphasis on their complexity and throughput. For a specified standard, the goal is to select the most area-efficient architecture for a target technology. We focus here on FPGA implementations, in light of the new features offered by modern devices, such as embedded RAM blocks, multipliers, and shift registers. Two architectures have been selected and implemented, a pipelined and a sequential one, each being optimized for the smallest area in their class.

We begin with an FFT overview and a comparative analysis of various architectures regarding their hardware complexity and throughput. The next point considers four OFDM standards and shows how the specifications affect the FFT requirements, providing also concrete implementation figures. We then go on to discuss the new advances in FPGA architectures and synthesis and how they facilitate the implementation of FFT cores. Finally, the implementation of the selected architectures is addressed, showing through concrete figures that all considered standards can be implemented in FPGA.

### 7.2.1. FFT Architectures Review

The discrete Fourier transform $X(k)$ of an N-point sequence x(n) is by definition:

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk}; \quad k = 0, 1, \ldots, N-1 \tag{7.1}$$

where $W_N^k = e^{-j2\pi k/N}$ are the complex roots of $-j$, equally spaced on the unit circle, also known as 'twiddle factors'.

FFT is an algorithm for computing the discrete Fourier transform, which reduces the computational complexity from $O(N^2)$ to $O(N \log N)$ by successive decompositions. Each decomposition step produces two (radix-2) or four (radix-4) smaller transforms. Depending on the decomposition direction, the FFT algorithm is said to be with decimation-in-time (DIT) or decimation-in-frequency (DIF). We use only the DIF variant for our analysis, both for radix-2

and radix-4. The graph of a radix-2 DIF 16-point FFT is shown in **Figure 7.8**. At the output, the values are arranged in 'bit reverse' order.

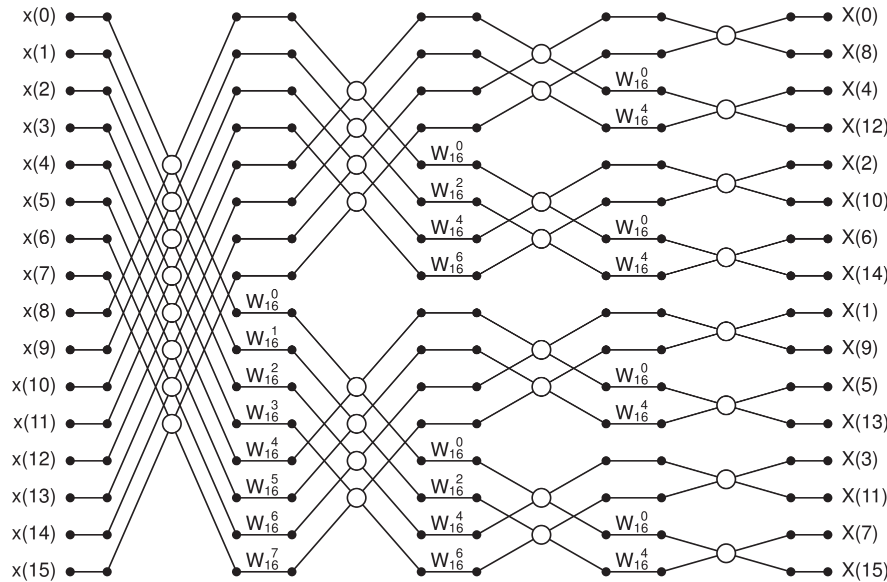


Figure 7.8.: Radix-2 DIF graph for the 16-point FFT

The basic operation in FFT is known as a butterfly. For radix-2 and radix-4 type, the main properties are summarized in **Table 7.1**. The number of radix-4 butterflies is 1/4 of the number of radix-2 butterflies for the same FFT size. However, radix-4 butterflies contain three times more multiplications and five times more additions, and the FFT size must be a power-of-4 for a pure radix-4 decomposition. If this is not the case, radix-4 and radix-2 stages can be mixed.

| | Radix-2 butterfly | Radix-4 butterfly |
|---|---|---|
| # Butterflies/stage | $N/2$ | $N/4$ |
| # Stages | $\log_2 N$ | $\log_4 N$ |
| # Butterflies | $(N/2)\log_2 N$ | $(N/4)\log_4 N$ |
| Operations (complex) | 2 2-input additions<br>1 multiplication | 4 4-input additions<br>3 multiplications |
| Building blocks | 6 adders<br>4 multipliers | 30 adders<br>12 multipliers |

Table 7.1.: Properties of radix-2 and radix-4 FFT butterflies

Various FFT architectures that provide different trade-offs between throughput and hardware complexity have been proposed in the literature. Depending on the degree of parallelism, they can be roughly divided into three main classes: systolic, pipelined, and sequential. Their complexity and timing properties are shown in **Table 7.2**, using the $O(\cdot)$ notation. $O(1)$ signifies

constant order, i.e. independent of $N$. Systolic architectures offer the highest throughput by exploiting the highly parallel structure of the FFT, but require considerable hardware resources. Sequential architectures have the smallest area but the lowest throughput, since a fixed number of butterflies is used. Pipelined architectures offer higher throughput at the cost of increased hardware complexity.

| | Complexity | | Timing | |
|---|---|---|---|---|
| Architecture | Storage | Processing | Throughput | Latency |
| Systolic | 0 | $O(N \log N)$ | $O(N)$ | $O(\log N)$ |
| Pipelined | $O(N)$ | $O(\log N)$ | $O(1)$ | $O(N)$ |
| Sequential | $O(N)$ | $O(1)$ | $O(1/\log N)$ | $O(N \log N)$ |

Table 7.2.: Properties of FFT architecture families

Pipelined FFT architectures fall into two main classes: delay commutator and delay feedback. The delay commutator solutions are usually multi-path and achieve a higher throughput, 2x for radix-2 and 4x for radix-4, as they process multiple data streams in parallel. On the other hand, delay feedback architectures are single-path (SDF), three variants thereof being shown in **Figure 7.9**: radix-2 (R2SDF), radix-4 (R4SDF), and radix-$2^2$ (R$2^2$SDF).

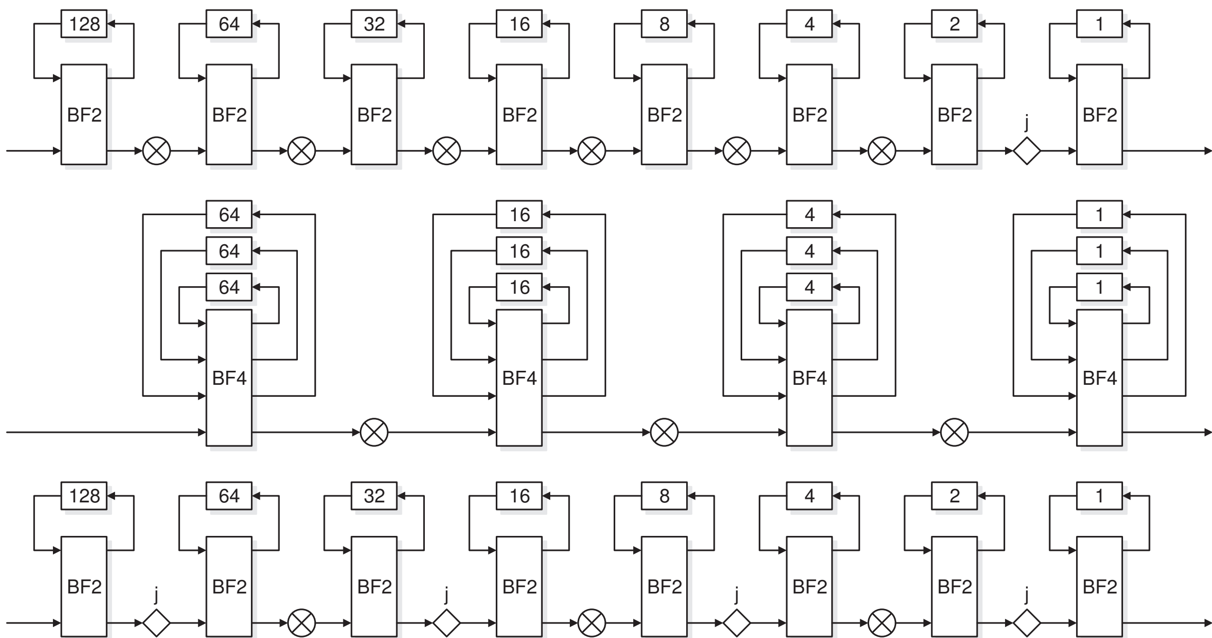


Figure 7.9.: Pipelined SDF FFT architectures (data-path only)

All these architectures use a single clock and offer the same throughput at a given operating frequency. Their hardware requirements are summarized in **Table 7.3**. The R$2^2$SDF architecture, proposed by He [29], combines the reduced number of multipliers of the radix-4 with

the simplified butterfly structure of the radix-2 solution. All SDF architectures have exactly the same memory requirements, although the number of delay lines is 50% larger for radix-4. These considerations lead us to select the R2$^2$SDF architecture for implementation as optimal in its class. FPGA implementation results are presented in **Subsection 7.2.4**.

| | | # Adders | | Delay storage | |
|---|---|---|---|---|---|
| Architecture | # Multipliers | In cmult. | In butt. | # Blocks | Samples |
| R2SDF | $4(\log_2 N - 2)$ | $2(\log_2 N - 2)$ | $4 \log_2 N$ | $\log_2 N - 1$ | $N - 1$ |
| R4SDF | $4(\log_4 N - 1)$ | $2(\log_4 N - 1)$ | $24 \log_4 N$ | $\log_4 N - 1$ | $N - 1$ |
| R2$^2$SDF | $4(\log_2 N - 1)$ | $2(\log_4 N - 1)$ | $4 \log_2 N$ | $\log_2 N - 1$ | $N - 1$ |

Table 7.3.: Hardware requirements of various SDF FFT architectures

Unlike pipelined architectures, sequential architectures reuse one physical butterfly to perform all the operations in the FFT graph. They require a minimum of hardware resources, at the cost of a decreased throughput. The amount of storage required is the same as for the pipelined version ($N$ complex samples). However, instead of multiple delay lines of sizes in geometrical progression, they require only a memory block of size $N$. This memory block can be partitioned into one, two, or four banks, depending on the architecture employed.

For radix-2 butterflies, the memory is divided into two banks, which are actually two distinct memories of size $N/2$. Every clock cycle, two samples are read and other two samples are written, one from/to each bank. Memory read, butterfly computation, and memory write form a 3-cycle pipeline. In order to minimize the memory size, butterfly outputs are written to the same addresses from which the inputs have been read. The computation is said to be in-place. If two memory banks are used, the two operands must be located in distinct banks so that they can be read at the same time. If radix-4 butterflies are used, four memory banks are needed.

Butterfly scheduling and memory allocation have made the object of extensive research in the past. Cohen [12] and Johnson [45] were the first to propose efficient hardware implementations for radix-2 and radix-4 respectively, an arguably improved memory access scheme being also proposed by Ma [58]. All solutions rely on the fact that the addresses of the butterfly operands differ in parity, as shown by Pease [73].

In some situations where the number of available memory blocks is limited, e.g. FPGA, it might be convenient to use a single memory block and read the butterfly operands sequentially. In this case, butterfly utilization decreases to 50% for radix-2 and to 25% for radix-4, with processing time increasing accordingly. This solution has been selected for our case study since it is the most efficient in terms of hardware resources. **Table 7.4** shows the execution times of various sequential architectures. It can be seen that the radix-4 butterfly with one RAM block is equivalent with the radix-2 butterfly with two RAM blocks, which allows for a trade-off between the number of RAM blocks and the number of multipliers.

| | RAM blocks | | |
|---|---|---|---|
| | 1 | 2 | 4 |
| Radix-2 | $N \log_2 N$ | $(N/2) \log_2 N$ | — |
| Radix-4 | $(N/2) \log_2 N$ | $(N/4) \log_2 N$ | $(N/8) \log_2 N$ |

Table 7.4.: Processing time for sequential FFT architectures

| OFDM standard | $f_s$ | $N_U$ | $N_G$ | Comments |
|---|---|---|---|---|
| IEEE 802.11a | 20 MHz | 64 | 16 | |
| DVB-T | 64/7 MHz | 8192 | 256 | 8K mode, 1/32 prefix |
| DAB | 2.048 MHz | 2048 | 504 | Mode I |
| DRM | 24 KHz | 512 | 128 | Mode B |

Table 7.5.: FFT requirements for various OFDM standards

### 7.2.2. FFT in OFDM Applications

FFT is an essential component of any OFDM receiver, being used for extracting the orthogonal subcarriers from the time-domain signal. Together with the Viterbi decoder, it accounts for the highest area and power consumption in an OFDM receiver, as they are very computationally intensive. In this section, we investigate the relationship between the OFDM parameters and the FFT requirements. As an example, we have chosen four mainstream OFDM standards, one for wireless LAN (IEEE 802.11a), and three for broadcasting (DVB-T, DAB, and DRM). The parameters that affect the FFT requirements are listed in **Table 7.5**. $N_U$ and $N_G$ are the number of samples in the useful part of the OFDM symbol and the guard interval respectively, while $f_s$ represents the sampling frequency of the baseband OFDM signal.

The raw data rate depends directly on the sampling frequency and the relative length of the guard interval. The absolute length thereof is chosen to be slightly longer than the channel impulse response in the environment for which the system is designed. The useful symbol size is then chosen as the lowest power-of-two that ensures an acceptable efficiency. The longer the symbol, the higher the efficiency, but at the cost of increased receiver complexity and decreased robustness against Doppler effects. Ultimately, the symbol size is directly proportional to the channel delay spread and the desired data rate.

Where a standard defines more than one mode, only the most demanding for the FFT core has been considered, which is the mode with the longest useful symbol and the shortest guard interval. For instance, DVB-T has a 2K and a 8K symbol mode, each with a choice of four different cyclic prefixes. The worst case occurs for 8K symbols and a 1/32 cyclic prefix. For DAB, four modes are specified, with symbol sizes of 2048, 512, 256, and 1024 respectively.

DRM specifies four modes: A, B, C, and D, all with very low data rates. The symbol size is a power-of-two only for Mode B. For the other modes, the FFT algorithm cannot be applied directly. Instead, a prime factor algorithm can be used.

In the following analysis, we intend to determine which architecture is best suited for the implementation of a particular OFDM standard in FPGA. We consider only the two architectures selected in **Subsection 7.2.1**, i.e. the pipelined R2$^2$SDF and the single-bank sequential radix-2. While the pipelined architecture is able to process a continuous data stream, the sequential one processes data symbol wise. The processing time of the sequential radix-2 architecture is actually $N(\log_2 N + 2)$ if one RAM bank is used, and $N/2(\log_2 N + 2)$ for two banks. The extra cycles are required by the write in and read out phases.

Since the operation of the sequential architecture is not continuous, a symbol buffer is needed at the input. Also, the clock frequency of the FFT, $f_{clk}$, will have to be higher than the sampling rate of the incoming OFDM stream $f_s$ so that an FFT transform can be completed within one symbol duration $(T_U + T_G)$. Having only one clock domain, $f_{clk}$ will have to be a multiple of $f_s$. Our goal is to determine the lowest clock frequency that satisfies these conditions. The first condition can be written as:

$$\frac{1}{f_{clk}} N_U(\log_2 N_U + 2) \leq \frac{1}{f_s}(N_U + N_G) \tag{7.2}$$

Thus, the integer $f_{clk}/f_s$ ratio will have the following expression, where $\lceil \cdot \rceil$ denotes rounding to the next higher integer:

$$\frac{f_{clk}}{f_s} = \left\lceil \frac{N_U}{N_U + N_G}(\log_2 N_U + 2) \right\rceil \tag{7.3}$$

The resulting clock frequencies are shown in **Table 7.6**. These results show that all OFDM standards considered can be implemented using sequential FTT architectures. If higher data rates are needed, pipeline architectures should be used instead, as they operate at the sampling rate of the OFDM signal. Unlike sequential architectures, whose throughput decreases with the FFT size $(O(1/\log N))$, pipeline architectures are characterized by a constant throughput given by the operating frequency alone. Hardware complexity, however, increases $O(\log N)$ with FFT size. An important aspect in the design of such architectures is the pipelining, which can be more or less aggressive depending on the target frequency.

Due to the guard interval between symbols, continuous operation is not possible. Therefore, the pipeline cannot be controlled by a simple counter, as it is the case with most implementations. Once a symbol started, it has to be processed through the pipeline regardless of the guard interval. Our solution consists in using a distributed control scheme that employs a separate counter for each pipeline stage. Because of the local $2^k$-cycle delays, each stage $k$ is enabled for at least $N + 2^k$ cycles in a row to allow for the initial buffering.

| | | | $f_{clk}$ | |
|---|---|---|---|---|
| OFDM standard | $f_s$ | FFT size | radix-2 single-bank | radix-2 dual-bank |
| IEEE 802.11a | 20 MHz | 64 | 140 MHz | 80 MHz |
| DVB-T | 64/7 MHz | 8192 | ≈137 MHz | ≈73 MHz |
| DAB | 2.048 MHz | 2048 | ≈22.5 MHz | ≈12.3 MHz |
| DRM | 24 KHz | 512 | 216 KHz | 120 KHz |

Table 7.6.: Minimum clock frequencies for sequential FFT architectures

### 7.2.3. Advances in FPGA and Synthesis Technology

Modern FPGA's offer enhanced functionality that can benefit DSP applications. On the one hand, the elementary configurable logic blocks (CLB) have been extended and include now carry and multiplexer logic, as well as shift registers. On the other hand, dedicated hardware blocks have been provided on chip, such as multipliers, RAM blocks, and clock managers. As an example, we have chosen Xilinx's Spartan-3 and Altera's CycloneII low-cost FPGA's.

Both FPGA families feature dedicated 18x18 multipliers, that can be used in DSP operations like filtering or FFT. Depending on the device, Spartan-3 has between 4 and 104, while CycloneII between 26 and 250. Besides, each multiplier in CycloneII can be also used as two independent 9x9 multipliers. Both families provide registers at the output of the multiplier blocks (also at the input for CycloneII) that can be taken advantage of for pipelining. For example, our complex multiplier VHDL module has 'generic' parameters that controls register insertion at the inputs, after multipliers, or at the outputs.

An essential resource is also the on-chip RAM. Spartan-3 features between 4 and 96 RAM blocks of 16Kbit each, while CycloneII between 26 and 250 blocks of 4Kbit each, depending on the device. The actual sizes are slightly larger and accommodate extra parity bits. However, their use is subject to many restrictions and the inference from HDL is not supported. Both RAM blocks are true dual-port (two reads and two writes) and support variable aspect ratios, with data widths between 1 and 32. Their operation is fully synchronous, with independent clocks for each port.

Another interesting feature of Spartan-3 FPGAs is that the two LUTs in a slice can be configured as 16-bit shift registers. Depending on the device, between 1536 and 66560 are available. These shift registers are ideally suited for implementing the delay lines in pipelined FFT architectures. Alternatively, larger delay blocks can be implemented using RAM in conjunction with an adder and a register. Our VHDL delay line design is completely generic with regard to its depth and width, having an additional boolean parameter that selects between a register and a RAM-based implementation.

These dedicated FPGA resources can be readily used in VHDL, either by hand-instantiating vendor-specific primitives or by direct inference from VHDL constructs. Hand-instantiation has the advantage that all the features in a certain block can be used, but the resulting VHDL code is vendor dependent and not parameterizable, which seriously restrict its reusability. A much better approach is to let the synthesis tool infer these blocks automatically. Thus, the design remains fully generic and platform independent.

Modern synthesis tools are able to infer RAM, ROM, shift registers, and multipliers automatically and use the dedicated FPGA resources. Depending on the size and the aspect ratio of a VHDL memory block, the optimal number of RAM blocks will be inferred and automatically connected in the appropriate configuration. Also, if the operands of a multiplication exceed 18 bit, 4 multipliers will be used instead of one. If the design is completely generic, the designer can choose the best parameters by taking the architectural features into account. Special attention must be paid to minimizing the number of RAM blocks and multipliers, as they are a scarce resource.

Besides the aforementioned inference capabilities, modern synthesis tools support an increasing number of VHDL features and constructs that improve the reusability of the designs considerably. Among them are the support for variable slices, recursive instantiation, or floating point 'generics' and intermediate results in static functions. One example where intermediate floating point values are needed is the generation of a fully parameterizable sine table, among whose applications is the storing of the FFT twiddle factors. As of this writing, only XST from Xilinx supports this feature and is therefore the only tool able to synthesize a fully generic FFT design.

### 7.2.4. Implementation Details

As mentioned in **Subsection 7.2.1**, we have selected the R2$^2$SDF pipelined and the single-bank sequential architectures for reference implementation. The designs have been described in VHDL, and are completely parameterizable through 'generics', such as FFT size and the input/internal/output word-lengths. Both architectures employ the same twiddle-factor generation scheme, using a ROM with the sin/cos values for the first quadrant, then exploiting the symmetry to generate the values for the other quadrants. Depending on their size, the twiddle-factor ROMs can be implemented with LUT or with block RAM.

Synthesis has been performed using Xilinx's XST synthesis tool. After various tests with data-path widths between 8 and 16 bits, we have seen that the resulting clock frequency only varies within 10%. That is why only the results for a 12-bit data-path are given in **Table 7.7**. All figures are shown as reported by the synthesis tool (before physical implementation), using a fast-grade Spartan-3 part.

The pipelined architecture allows for different pipelining solutions. The default strategy is to add a pipeline register every other stage, after each butterfly preceded by a $j$ multiplication. If higher clock frequencies are needed, more aggressive pipelining can be used. However, the

| | Pipelined FFT | | | | Sequential FFT | | | |
|---|---|---|---|---|---|---|---|---|
| Results | 64 | 256 | 1024 | 4096 | 64 | 256 | 1024 | 4096 |
| # Slices | 631 | 1024 | 1854 | 4444 | 105 | 111 | 132 | 143 |
| # Slice FF | 359 | 491 | 632 | 789 | 119 | 125 | 131 | 135 |
| # Block RAM | 1 | 2 | 3 | 5 | 2 | 2 | 3 | 8 |
| # Block MULT | 8 | 12 | 16 | 20 | 4 | 4 | 4 | 4 |
| $f_{clk}$ | 65 ... 133 MHz | | | | 200 MHz | | | |

Table 7.7.: Synthesis results for Spartan-3 FPGA

maximum achievable value will be limited by the embedded block multiplier and by the FPGA routing. For large FFT sizes, the shift registers in the feedback delays will use an increasingly percentage of the slices. If needed, slices can be saved by implementing the delays with RAM, starting from a threshold stage index. In our design, this threshold is given as a generic parameter. For Spartan-3 FPGAs, the limiting factor for large FFTs is the number of available block multipliers.

The proposed simplified sequential architecture is shown in **Figure 7.10**, while **Figure 7.11** shows the controller. The synthesis results reveal a maximum clock frequency of about 200 Mhz, due to pipelining and the absence of long routing delays. The number of slices grows with $\log N$ (due to the controller logic), while the memory size grows linearly with $N$. For large FFT sizes, the limiting factor becomes the number of available RAM blocks. This architecture exhibits an excellent performance/area ratio for FPGAs, being able to meet the specifications of the most demanding OFDM standards. For instance, an FFT for the 8K mode of DVB-T can be implemented in a low-cost XC3S400 FPGA. All 16 RAM blocks are used, while 12 out of 16 block multipliers and 94% of the slices remain available.

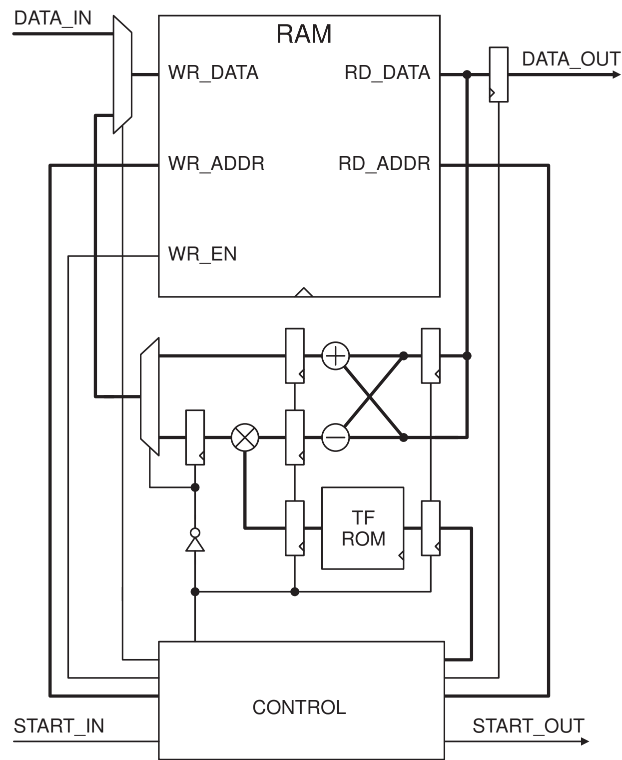


Figure 7.10.: Single-bank sequential FFT architecture

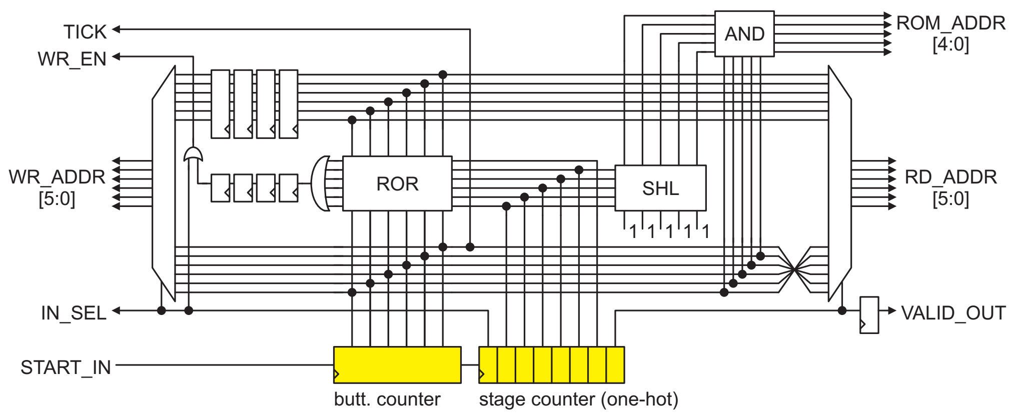


Figure 7.11.: Simplified control logic for the sequential FFT architecture