DESIGN OF AN FFT PROCESSOR

Erik Nordhamn, Björn Sikström and Lars Wanhammar
Department of Electrical Engineering
Linköping University
S-581 83 Linköping, Sweden

Abstract

In this paper we present a structured approach to design fixed function DSP systems. The approach is applied to design an ASIC FFT processor. The emphasis is placed on synthesis of an efficient architecture for the processor. The synthesis starts from the system specification and goes through a number of design steps: specification, DSP algorithm design, partitioning, scheduling, resource allocation and architecture design. The remaining design steps for a complete VLSI chip, i.e., logic and circuit design followed by VLSI design are discussed in brief. The approach allow the designer to estimate, at each design step, the effect on the cost of the system in terms of silicon area and power consumption.

1 Introduction

This paper concerns on-going work, at the Department of Electrical Engineering at Linköping University, to develop design methodologies for fixed function (ASIC) digital signal processing systems with state-of-the-art performance. To evaluate different methodologies, we have chosen to work on a number of case studies: an FFT processor, a DCT processor and a multirate wave digital filter (WDF) used for interpolation of the sample frequency [7]. In this paper, we will only discuss the design of the FFT processor.

Most signal processing systems are so called hard real-time systems. This means that the system must respond to an input within a given time-frame. The system is not allowed to miss a deadline. Therefore this type of signal processing requires resource adequate systems in contrast to resource limited systems which are used for supercomputing applications. Fortunately, most DSP algorithms are regular in the sense that they do not contain conditional branching operations. This makes it possible to minimize the amount of required resources by using static schedule for the algorithm. Hence, the required peek performance and the average performance are the same.

2 The System Design Process

We advocate a top-down approach to design fixed function systems. The design follows the design strategy described below [2-4, 7].

1 The specification of the DSP properties are determined.
A suitable DSP algorithm is selected and tuned to the application. The algorithm is successively partitioned into a set of co-operating processes. Storage of data is also considered as a process. These processes will later be mapped onto virtual machines and finally assigned to PEs. Most DSP algorithms are formulated using a traditional sequential programing language. The algorithm must therefore be transformed from the original sequential description into a parallel description.

The processes are scheduled in time so that the specified performance is met and the required amount of resources is minimized. A life-time table for the memory variables is derived from the process schedule.

An adequate amount of resources are allocated. The processes are assigned to resources, i.e., PEs, memories, and communication channels in the resource assignment step. The methods that are used are based on clique-partitioning combined with a variant of the left-edge algorithm that is used for wire routing.

The generic architecture is a shared-memory architecture. In this step, we minimize the cost of the resources, i.e., PEs and memories. Often bit-serial PEs are used. The bit-serial PEs communicate with the memories through serial/parallel converters, consisting of a set of shift registers, that acts as cache memories. It is possible to balance the communication bandwidths throughout the architecture by choosing a proper word length for the memories.

Steps 3 and 7 are iterated until a satisfactory solution is found.

This step involves the logic design of the modules in the architecture, i.e., the PEs, memories, and control units.

In the circuit design step, the modules are designed at the transistor level. Transistor sizes are optimized with respect to performance in terms of speed, power consumption and chip area.

In the last step, the VLSI design phase, involves layout, i.e., floor-planning, placement, and wire routing. A key idea is that system or circuit design is not done at this stage.

The emphasis, in this paper, is on step 3 through 7.

We stress that design is an iterative process. We cannot expect to go through the steps above only once. Instead, we must redesign the system at a higher level if lower level reveal unsolvable problems. The estimate of the system parameters, e.g., chip area, power dissipation, will become more and more accurate as design progresses. In the first pass through a design level we may do a preliminary, crude design and successively refine the design in subsequent design iteration. Generally, we are also interested in exploring the design space by investigating a large variety of design alternatives. Simulation and other means of validation is required to validate and evaluate the design decisions at all levels.

3 Specification
We will not discuss the specification of the FFT processor in detail. Here we only assume that the speed and required accuracy of the FFT algorithm is given. The FFT processor shall compute a 1024-point Sande-Tukey FFT or IFFT. It will have a sustained throughput of more than 1000 FFTs per second.

The system will communicate with an operator through a host processor. A 32-bit I/O interface to a microprocessor is required. The I/O data rate will be at least 8 MHz. The input and output data word length will be 16 bits for the real and imaginary part, respectively. The internal data word length will be 21 bits. The word length for the coefficients, i.e., the twiddle factors, is 16 bits.

The FFT processor will be implemented as a self-contained system on one single chip. The chip area and power consumption of the system shall be minimized.

4 Partitioning of the FFT

The original algorithm is described in a sequential form, in this case using Pascal, as shown in Fig. 2. An inverse Fourier transform (IFFT) is performed if the real and imaginary parts of the input and output sequences are interchanged [Wanh91].

```pascal
Program ST_FFT;
const
  N = 1024;
  NU = 10;
  Nminus1 = 1023;

type
  Complex = record
    re : Double;
    im : Double;
  end;

var
  x : Complex[0..Nminus1];
  Stage, Ns, kNs, i, j, p, j, k : Integer;
  WCos, WSin, TwoPiN, TempRe, TempIm : Double;
begin
  (* READ INPUT DATA INTO x *)
  Ns := N;
  TwoPiN := 2 * Pi / N;
  for Stage := 1 to NU do
    begin
      k := 0;
      Ns := Ns div 2;
      for j := 1 to (N div (2 * Ns)) do
        begin
          for i := 1 to Ns do
```

![Fig. 1. The FFT processor.](image-url)
begin
  p := k * 2^(Stage - 1) mod (N div 2);
  W_Process(Wp, p);
  kNs := k + Ns;
  Butterfly(k, kNs, Wp); { Butterfly process }
  k := k + 1;
end;

end;

Unscramble;
{ OUTPUT DATA STORED IN x }
end.

Fig. 2. The original algorithm.

4.1 First Design Iteration

In the first design iteration, the system is partitioned into four communicating processes: input process, FFT process, output process, and memory process. The input process is responsible for reading data into the FFT processor from the outside world. The data are stored by the memory process. Conversely, the output process writes data, from the memory, to the outside world. The output process will also handle the unscrambling of the data array in the last stage of the FFT. The FFT process handles the interchange of real and imaginary parts of data that is required for computation of the IFFT. Hence, both of these tasks can be accomplished without using any extra processing time.

The total time required for input and output is estimated to:
\[ t_{I/O} = \frac{2 \cdot 1024}{8 \cdot 10^6} = 0.256 \text{ ms} \]

This is assuming that complex words are transferred sequentially to and from the FFT processor. The time remaining for the actual FFT computation with a throughput of 1000 FFTs per second, is:

\[ t_{FFT} = 0.744 \text{ ms} \]

The number of butterfly operations required in the FFT is:

\[ \frac{N}{2} \log_2(N) = 5120 \]

A bit-serial butterfly PE can be implemented using 24 clock cycles. Typical clock frequencies are about 110 MHz. Hence, the minimal number of butterfly PEs is:

\[ N_{PEb} = \frac{24 \cdot 5120}{0.744 \cdot 10^{-3} \cdot 110 \cdot 10^6} = 1.5 \]

Thus, we need at least two butterfly PEs to reach the necessary speed. We can also make estimates of the data rate to and from the memory process. For each butterfly operation we must read and write two complex data. Hence, the data rate will be:

\[ \frac{(2 + 2) \cdot 5120}{0.744 \cdot 10^{-3}} = 27.5 \cdot 10^6 \text{ complex words/s} \]

In principle, it is possible to use only one logical memory. Memories with this data rate can easily be implemented. However, we choose to use two logical memories. This will make the implementation of the memories simpler. Also, it is desirable that the memory clock frequency is a multiple of the I/O frequency. We therefore select the memory clock frequency to 16 MHz.

The following design iterations aim to transform the original sequential algorithm into a parallel description that can be efficiently mapped onto the hardware resources.

### 4.2 Second Design Iteration

In the second design iteration we explore the fact the twiddle factors can shared between the two PEs. In stage 1, of the 16-point Sande-Tukey FFT which is shown in Fig. 4. we have the following relation between \( W_p \) and \( W_{p+N/4} \):

\[ W_p + N/4 = W_{N/4} W_p = -j W_p \quad \text{, since } W_p = e^{-j2\pi p/N} \]
Hence, only one factor is required. In stage 2 to 4 it is possible to schedule the butterfly processes so that two butterflies, that use the same twiddle factor, are performed concurrently.

The sequential description is transformed into the form, shown in Fig. 5. This is still a sequential description but the two butterfly processes can in principle be performed in parallel.

Program ST_FFT;
begin
  { READ INPUT DATA INTO x }
  TwoPiN := 2 * Pi / N;
  Ns := N;
  for Stage := 1 to NU do
    begin
      Ns := Ns div 2;
      for m := 0 to N div 4 - 1 do
        begin
          Addresses(p, k, kNs, k2, k2Ns, m, Stage);
          W_Process(WCos, WSin, p);
          Butterfly1(k, kNs, Wp, Stage);
          Butterfly2(k2, k2Ns, Wp, Stage);
        end;
    end;
  Unscramble;
  { OUTPUT DATA STORED IN x }
end.

Fig. 5. Second sequential description.

4.3 Final Design Iteration of the Algorithm
In the final iteration, we map the sequential description onto a parallel description, as shown in Fig. 6. We also include memory transactions and control loops as processes.

\[
\text{TwoPiN := 2 * Pi / N;}
\]

\[
\text{Ns := N;}
\]

\[
\text{for Stage:= 1 to NU do}
\]

\[
\text{Ns := Ns \ div 2;}
\]

\[
\text{for m:= 0 to (N \ div 4 - 1) do}
\]

Fig. 6. Parallel description of the FFT.

5 Scheduling

As an example of the scheduling process, we will describe the scheduling of the inner loop [1]. The inner loop processes are shown in Fig. 7. Estimates of the execution time of the corresponding PEs are included in the figure. The precedence relations denoted with dashed arrows are precedence relations imposed for control purposes. Hence, they do not correspond to some interchange of data. The inner loop is scheduled according to Fig. 8.
The rectangles denote the life-time of the processes. The white area is the interval when the process is active. However, the result is not available until after the grey interval, because of pipelining of the PE that will execute the process.

6 Resource Allocation

Generally the resource allocation step is simple. A lower bound on the number of PEs can be found from the total amount of operations per second. The required amount have to be determined from the process schedule. The number of logical memories, or ports, is also determined from the schedule, and is equal to the maximal number of values that are read/written simultaneously [7].

7 Resource Assignment

In this step, the processes are assigned to specific resources, e.g., butterfly processes to butterfly PEs and variables to memories and memory cells. The chip area required for memory is in this application significant. We will therefore use an in-place FFT where the result of a butterfly operation is always written back to the same memory cells that were used as inputs. Using this scheme only 1024 complex-valued memory cells is required.

Several memory assignments are possible. Figure 9 shows two alternatives for a 16-point FFT. In the first alternative, the first half of the data variables are allocated to RAM 0 and the second half to RAM 1. In the second alternative, the variables are assigned so that a butterfly always receive input data from two different memories. The second
assignment alternative can be described by an EXOR function of all bits in the binary representation of memory address index $i$. The resulting assignment is also shown in Fig. 9.

There are also several PE alternatives for assignment of the butterfly PEs. One alternative is to assign the first half of the butterfly processes to PE0 and the second half to PE1, as indicated in Fig. 10. Another alternative, that assures that a data variable, $x(i)$, is always used as an input data to the same input port of the butterfly PE. The resulting assignment is also shown in Fig. 10. The mapping function of a process corresponds to the EXOR function of all bits in the binary representation of the butterfly number in a stage, counted from top to bottom.

7 Synthesis of an Optimal Architecture

First, we use a shared-memory architecture with bit-serial PEs and bit-parallel RAMs. To convert between bit-parallel and bit-serial we use a set of shift registers, one for each input/output of the PEs. These
registers are also used as cache memories to equalize the communication data rates to the main memories.

The implementation of the interconnection network (ICN) depends heavily on the PE and RAM assignment. Further, this part of the circuit can prove to be expensive in terms of chip area and power consumption. The scheduling and resource allocation and assignment steps usually optimize only resources as PEs and RAMs. In this section we will show that the resource assignment of PEs and RAMs will influence the ICN as well as the complexity of the control structures that are required.

In principle, we will have four different alternatives, due to the two different assignment alternatives for RAMs and PEs, respectively.

The first alternative consists of the simple RAM assignment and using the EXOR pattern for the PE assignment. Its main advantage is a simple address generation for RAM. However, the ICN contain switches on the bit-serial side of the architecture. It would be favorable to remove them.

The second architectural alternative is the combination of the simple type of both RAM and PE assignment.
The third architectural alternative is shown in Fig. 13. This architecture is the result of using an EXOR pattern for both the memory and PE assignments. The main advantage with this architecture is that the high-speed interconnection network on the bit-serial side is fixed. The only means to control of the ICN is the possibility to choose which of the S/P registers to write to or read from. Further, the address generation for the RAM can be designed so that this control becomes simple.

Fig. 13. Third architectural alternative.

The third architectural alternative is chosen as the final. In Fig. 14 we have included the control of the architecture.

Fig. 14. Final architecture.

The third architectural alternative is chosen as the final. In Fig. 14 we have included the control of the architecture.

8 Acknowledgements

This work was supported by the swedish board for technical development (STU).

9 References


