Pipelined FFT Example

Technical Report #96-45
October 4, 1996

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425
(714) 824-7063

pgrun@ics.uci.edu
wpan@ics.uci.edu
sbakshi@ics.uci.edu
gajski@ics.uci.edu

Abstract

Fourier transform techniques are very popular and practical for DSP applications. Among them, Discrete Fourier Transform (DFT) and its fast algorithm (FFT) are best known and most important. In this report, we define two general communication models and a corresponding handshaking protocol for the FFT chip. We explore multiple design alternatives using efficient pipelining techniques, and show what algorithm transformations are needed. From this example we derive a methodology for applying the pipelining techniques in a time-constrained formulation. Parameterization for a more general FFT chip is also discussed.

Contents

List of Figures

List of Tables

Pipelined FFT Example

Peter Grun, Wenwei Pan, Smita Bakshi, Daniel D. Gajski
Department of Information and Computer Science
University of California, Irvine, CA 92697-3425

Abstract

Fourier transform techniques are very popular and practical for DSP applications. Among them, Discrete Fourier Transform (DFT) and its fast algorithm (FFT) are best known and most important. In this report, we define two general communication models and a corresponding handshaking protocol for the FFT chip. We explore multiple design alternatives using efficient pipelining techniques, and show what algorithm transformations are needed. From this example we derive a methodology for applying the pipelining techniques in a time-constrained formulation. Parameterization for a more general FFT chip is also discussed.

1 Introduction

In most Digital Signal Processing (DSP) applications, performance is a very important characteristic. Pipelining is one of the techniques which can decrease the execution time of a chip without a significant impact on the cost.

In this report, we give a detailed example showing how to design pipelined digital systems from behavioral description. The example is a custom ASIC for Fast Fourier Transform (FFT), which is used in the many DSP applications.

Section 2 describes the FFT example, section 3 shows two different communication patterns and their design implications. In section 4 we give the components library used throughout the report. Section 5 discusses different pipelining techniques, and section 6 shows the details of the implementation. Section 7 shows the advantages and disadvantages of a parametrized approach. We summarize the design methodology derived from this example in section 8, and finally we present the conclusions.

2. Example Description.

The mathematical definition for DFT [5] is:

$$X(k) = \sum_{n=0}^{N-1} x(n) W^{nk}$$

where x(n) represents input discrete sequence in the time domain, X(k) represents its transformation in the frequency domain. Generally, both the x(n) and X(k) are complex numbers. N is the transform length, and the DFT coefficients used in the DFT kernel, W, are:

$$W = e^{-\frac{j2\pi}{N}}$$

Knowing that the W factor in DFT is periodic and symmetric if N is a power of 2, the FFT algorithm reduces the complexity for computing the same kernel. The direct DFT takes N^2 complex operations. The radix-2 FFT computes the discrete fourier transform in N * $\log_2 N$ complex operations.

In the following, We use the Decimation in Time FFT algorithms as an example for our work. If not explicitly stated, the FFT example in the report will be a 64-point complex number FFT.

Knowing that the W factor in DFT is periodic and symmetric if N is a power of 2, the FFT algorithm reduces the complexity for computing the same kernel. The direct DFT takes N^2 complex operations. The radix-2 FFT computes the discrete fourier transform in $N * \log_2 N$ complex operations.

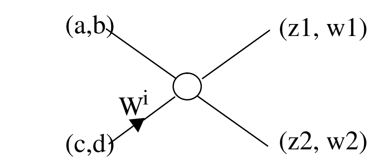

Figure 1 : Butterfly operation for FFT

The basic operation in the FFT algorithm is called butterfly. In Figure 1, we show the butterfly operation for the Decimation-in-time FFT.

In the butterfly operation in Figure 1, (a,b) and (c,d) are 2 input complex numbers. (z1,w1) and (z2,w2) are two output numbers. In the in-place FFT algorithm, the output of the butterfly can be stored in the positions where the input were. W^i ($0 <= i <= N/2 - 1$) are the coefficients. The relationship between the input and output of the butterfly operation is:

$$
\begin{aligned}
tr &= wr * c - wi * d; \\
ti &= wr * d + wi * c; \\
z1 &= a - tr, \; w1 = b - ti; \\
z2 &= a + tr, \; w2 = b + ti;
\end{aligned}
$$

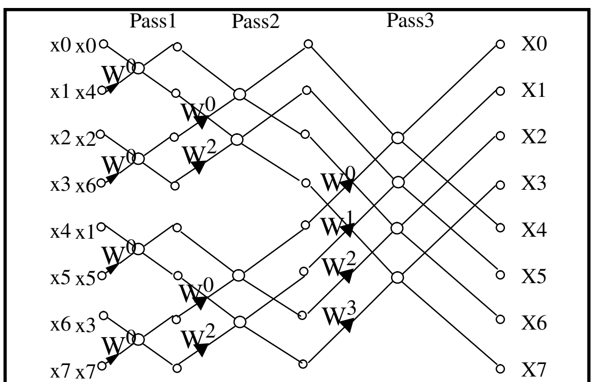

Figure 2: Decimation-in-time (DIT) flow graph

Knowing the basic FFT operation, the easiest way to understand the FFT algorithm is to look at the flow graph. The flow graph for Decimation-in-time 8-point FFT algorithm is shown in Figure 2. For N-point FFT, generally, there are N passes. In each pass, there are N/2 butterfly operations to be executed. In Figure 2, first, the 8-point input sequence is bit-reversed. Then, after each pass, a new 8-point sequence is derived and stored in the same place with the old sequence. After pass 3, the new sequence is already the transformed sequence in the frequency domain.

It is also noted that for N-point FFT, only N/2 - 1 W coefficients are needed. Knowing this rule can simplify the ROM design for the FFT chip.

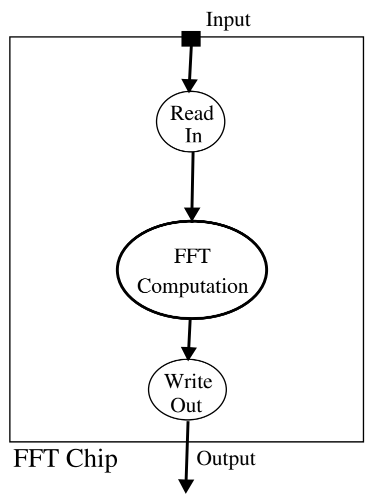

Figure 3: FFT Dataflow.

In Figure 3, a FFT chip is defined. It has 2 data ports: data input port and data output port. It consists of 3 stages: Read in, FFT computation and Write out. For decimation-in-time FFT algorithm, the input data sequence is shuffled while being read in from the input data bus under certain protocol. Then, after the in-place computation, the output data sequence can be written to the output data bus.

3. Communication Models.

The FFT chip we defined in Section 2 has no knowledge how the data is sending to or receiving from the chip. We consider two communication models.

3.1. Model 1: Distributed In Distributed Out

In this model, the FFT chip is considered as 3 stage pipelined including input sample, compute sample and output sample. New sample sequence arrives while the FFT chip is processing the old sequence. Data is arriving and leaving at certain rate in burst mode.

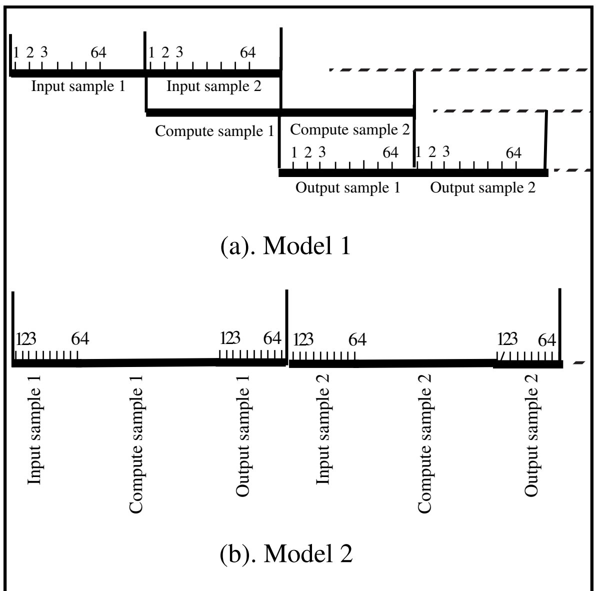

Figure 4: Communication models.

3.2. Model 2: Burst In Burst Out

In this model, the FFT chip still consists of 3 stages but they are not pipelined. It starts to compute new sample sequence only when the old sample sequence has been received.

Figure 5 shows the difference of the FFT chip using these two different models. In the model 1 block diagram, there are 2 RAMs between stage 1 and stage 2. One RAM is reading the data while the other is being read. After every amount of delay of a FFT block level pipelining stage, the roles of these 2 RAMs are swapped. The same situation exists between stage 2 and stage 3.

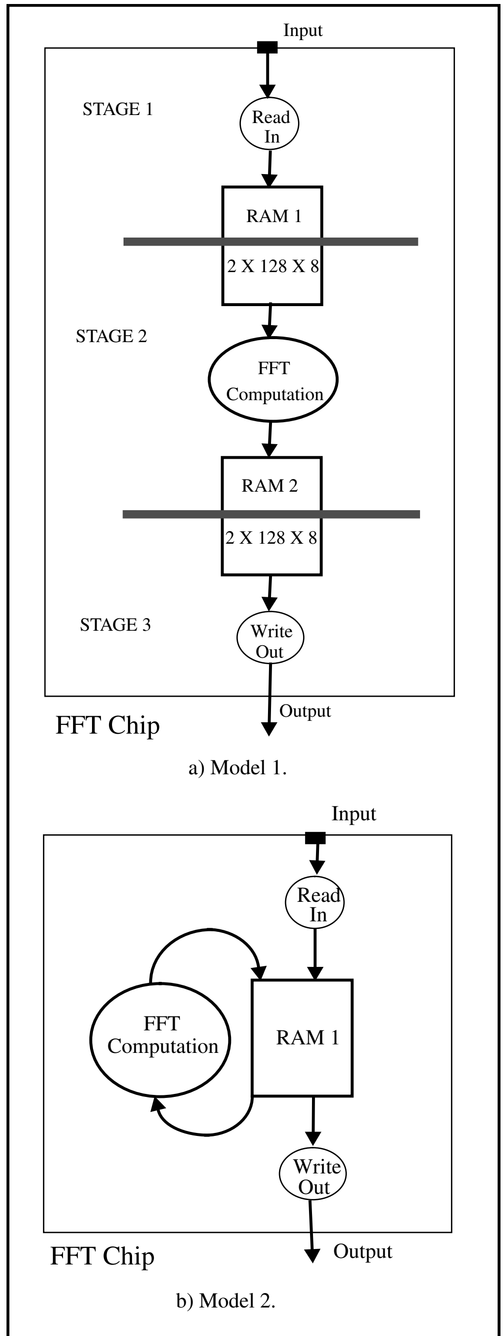

Figure 5: Block diagram for FFT chip.

In the model 2 block diagram, only one RAM is needed. The reason is because decimation-in-time algorithm is in-place, and the 3 stages in this FFT chip is non-pipelined.

4. Components Library.

The components used in our design are from [3]. Due to technology improvement [6], the delay of an inverter is 0.1 ns which is 10% of the nominal delay of 1 ns of the inverter from [3]. Therefore, we scaled down the delays for all the components by a factor of 10. Table 1 shows the delay and cost of the components in terms of ns and number of transistors respectively.

No	Component	Delay (ns)	Cost (trans.)
1	8 bit CLA adder	1.5	537
2	16 bit CLA adder	2.1	1074
3	8 bit booth multiplier	5.6	3562
4	8 bit booth multiplier (2 stage pipe)	3.5	4210
5	8 bit booth multiplier (4 stage pipe)	2.7	5218
6	8 bit register	0.4	256
7	128x8 RAM	3.5	6144
8	32x16 ROM	3.5	2048

Table 1: Components library.

For the pipelined components, the delay represents the delay of the longest stage. For the storage components it represents the average between the reading and writing time.

In column four we show the cost of each component. We use 1 literal/2 transistors estimation for the basic gates (*nand, nor, inverter*), which we use subsequently in the computation of the cost of the rest of components, in a bottom-up fashion.

5. Design Exploration.

In this section we present different design alternatives for the FFT chip. We trade-off the cost of the chip against performance. For each design we show the behavioral description, and outline the structural representation, as a datapath and controller. Finaly, we show the design space which has been explored, both as a table and a diagram.

We perform the exploration by using different resource constraints and different levels of pipelining. For most DSP applications, pipelining improves the performance of the computation without significantly affecting the cost of the chip. If two or more operations are performed repeatedly on different data samples, they can be pipelined by inserting latches between the operations. The cost increase is represented only by the additional latches, whereas the number of functional units remains the same. On the other hand, when using different amounts of resources (such as functional units), we increase the degree of parallelism in the computation. The cost of the chip as well as the performance increases correspondingly.

The pipelining can be performed at different levels of the behavioral description. Along with the different FU allocation and schedules, these alternatives represent the design points in our exploration.

5.1 No pipelining.

In the first design alternative, there is no pipelining performed. Nevertheless, depending on the number of functional units available, some operations can be done in parallel. E.g, if we have more adders/subtractors, the operations inside the inner loop in the FFT can be performed in less time. Figure 6 shows the behavioral description of the non-pipelined FFT.

```
ReadIn();
nn:=64; n:=128;
mmax:=2;
while (mmax<n)
loop
        istep:=2*mmax;
        theta:=math_pi/real(isign*mmax);
        sinth:=sin(theta);
        wsr:=-2.0*sinth*sinth;
        wsi:=in(theta*2.0);
        wr:=1.0;
        wi:=0.0;
        m:=0;
        while(m<mmax)
        loop
                i:=m;
                while(i<n)
                loop
                        j:=i+mmax;
                        tr:=(wr*data(j))-(wi*data(j+1));
                        ti:=(wr*data(j+1))+(wi*data(j));
                        data(j):=data(i) - tr;
                        data(j+1):=data(i+1) - ti;
                        data(i):=data(i) + tr;
                        data(i+1):=data(i+1) + ti;
                        i:=i+istep;
                end loop;
                tr:=wr;
                wr:=wr+(wr*wsr)-(wi*wsi);
                wi:=wi+(wi*wsr)+(tr*wsi);
                m:=m+2;
        end loop;
        mmax:=istep;
end loop;
WriteOut();
```

Figure 6: Completely sequential behavior.

Scheduling and pipelining are ortogonal issues. On one hand, we can obtain different points in the design space by performing scheduling (and allocation) with different resource constraints. The performance and the cost of each design depends on the level of parallelism present in the particular schedule. On the other hand, any particular scheduled design, can be pipelined in different ways.

We considered 2 scheduling/allocation alternatives for the FFT. By scheduling the computation part of the non-pipelined FFT using 1 multiplier and 8 adders/subtractors, we obtain a total delay of 8601.6ns. By scheduling the computation part of the FFT with 4 multipliers and 8 adders/subtractors we obtain a delay of 3225.6ns. The delay of the ReadIn and WriteOut parts depend on the communication model used. The total delay for the whole circuit is the sum of the communication delays, and the computation delay.

5.2 Process Pipelining.

The first level of pipelining is the process pipelining. In a behavioral description, each process is comprised of a set of HDL instructions (as shown in Figure 6, in the FFT example these instructions are: the call to ReadIn, some assignments, the while loop, and the call to WriteOut). Based on the estimated execution time of the instructions we partition them into pipe stages, so that each stage has approximately the same delay. Figure 6 shows that in the case of FFT we obtain 3 process level pipe stages. The first one reads the input, the second one computes the FFT, and the third one writes the output to the output port.

```
-- Read data                      STAGE 1
ReadIn();
-------------------------------------------
-- FFT Computation
nn := 64; n:= 128;
mmax:=2;
while (mmax<n)                    STAGE 2
loop
      ............................

end loop;
-------------------------------------------
-- Write data
WriteOut();
                                  STAGE 3
FFT Chip
```

Figure 6: Process pipelining.

In Figure 7 we show the timing diagram for the process pipelining. After the start-up period, the three stages are executing in parallel for consecutive samples of data.

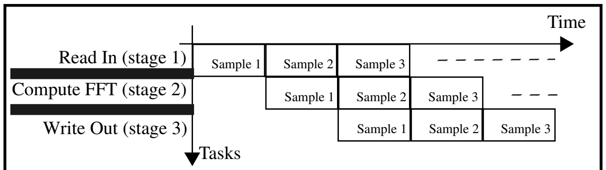

Figure 7: Time diagram for process pipelining

We considered 2 scheduling/allocation alternatives for the FFT. By scheduling the computation stage of the FFT using 1 multiplier and 8 adders/subtractors, we obtain a stage delay of 8601.6ns. By scheduling the computation stage of the FFT with 4 multipliers and 8 adders/subtractors we obtain a stage delay of 3225.6ns. Since the ReadIn and WriteOut are done in parallel with the computation (different pipeline stages), the throughput of the whole circuit is given by the computation stage delay. The cost of the circuit is increased due to the extra pipeline registers needed to store the information between the pipe stages, but the perfromance is also increased.

5.2 Loop Body Pipelining.

The loops from a behavioral description can be also pipelined. If the loop body consists of more than one instruction, and we know the time each of these instructions takes to execute, we can divide the body of the loop into equal delay stages which will form the pipeline. These pipe stages will execute in parallel, on consecutive iterations of the loop.

Any of the instructions forming the body of the loop can be a loop at its turn. In case that the number of iterations this inner loop executes is variable, we cannot determine the total execution time of this loop. Therefore we do not have enough information to divide the body of the outer loop into equal pipe stages. Thus the body of a loop cannot always be pipelined.

The initial FFT specification is such a case. To solve this, we transformed the algorithm so that the execution time of the instructions composing the body of the loops is known and constant. Due to the loop transformations, we have to recompute the loop indexes. The transformed algorithm is shown in Figure 8.

As mentioned above, we know the execution time for all the instruction from the body of LOOP 2, since none of them is a loop. Therefore the body of LOOP 2 can be pipelined. We divide it into 5 pipe stages of approximately equal delay, and we obtain the pipelining as shown by the light dividing lines in Figure 8.

On the other hand, the body of LOOP 1 is composed of 4 assignments plus the LOOP 2. The execution time of the LOOP 2 is much higher than the execution time of the assignments. Since we cannot divide these instructions into equal delay stages, it doesn't make sense to pipeline the body of this loop.

In Figure 9 we show the timing diagram for loop body pipelining for the FFT example. The second stage of the process level pipelining comprises the 5 loop body pipe stages. While the process level pipe stages perform on different samples of data, the loop body pipe stages execute on different iterations of the loop, for the same input sample.

```
ReadIn();                                                                                    STAGE 1
----------------------------------------------------------------------------------------------------
nn := 64; n:= 128;
mmax:=2;
while (mmax<n)    ---- *********** LOOP 1 ************
loop
      istep:=2*mmax;
      m:=0; m_rom := 0;
      i:=0;
      for k:=0 to nn/2-1 loop      ---- ************** LOOP 2 **************
            i_save:=i;
            j_save:=i+mmax;


            wr:=ROM_real(m_rom);                        -- MEMORY READ
            wi:=ROM_imag(m_rom +1);
            c := data(j_save); d := data(j_save+1);                        stage 1
            ------------------------------------------------------------------
            e := wr * c; f := wi * d;                   -- MULTIPLICATIONS  stage 2
            g := wr * d; h := wi * c;

            i := i + istep;
            ------------------------------------------------------------------    STAGE 2
            tr := e - f; ti := g + h;                   -- ADD/SUB          stage 3
            a := data(i_save); b := data(i_save+1);

            if (i>=n) then
                  m_rom:=m_rom+nn/mmax;  m := m + 2;
            end if;
            ------------------------------------------------------------------
            z1 := a - tr; w1 := b - ti;                 -- ADD/SUB          stage 4
            z2 := a + tr; w2 := b + ti;

            if (i>=n) and (m>=mmax) then m:=0; m_rom := 0;
            end if;
            ------------------------------------------------------------------
            data(i_save) := z2; data(i_save+1):= w2;    -- MEMORY WRITE     stage 5
            data(j_save) := z1; data(j_save+1) := w1;

            if (i>=n) then
                  i:=m;
            send if;
      end loop;
      mmax:=istep;
end loop;
----------------------------------------------------------------------------------------------------
WriteOut();                                                                                  STAGE 3
```

FFT Chip

Figure 8: Loop body pipelining.

The LOOP 1 and LOOP 2 control structures induce the repetitive execution of the stages 1 through 5 of loop body pipelining. Therefore the execution time for the stage 2 from process pipelining is given by the total execution time for all the iterations of LOOP 1 and LOOP 2. Since LOOP 1 and LOOP 2 are nested loops, the total number of iterations is equal to the product of the iterations of the two loops.

Between consecutive iterations of the inner loop (LOOP 1), there are no loop carried dependences, because they operate on different elements of the array data. On the other hand consecutive iterations of the outer loop (LOOP 2) execute on the same array data, therefore there may be dependences.

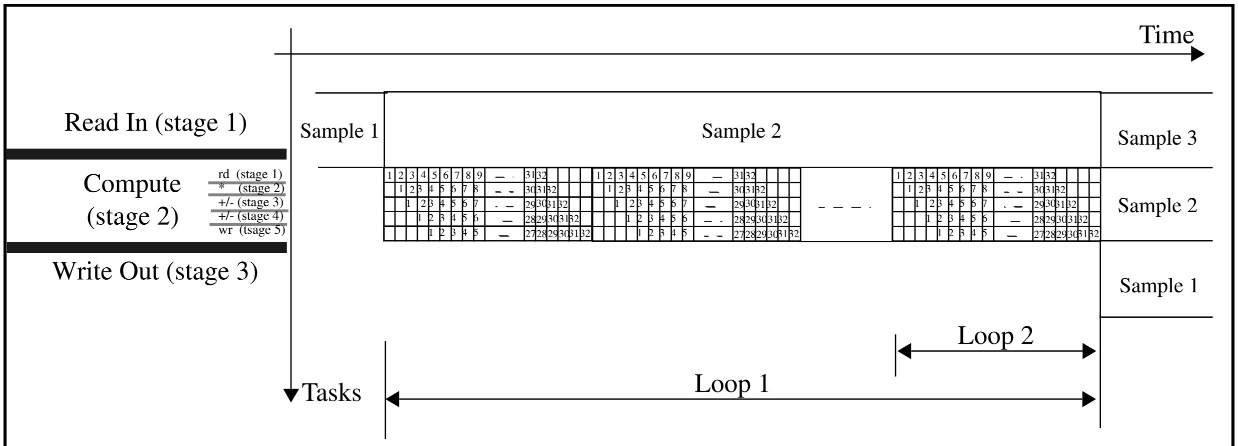

Figure 9: Timing diagram for loop body pipelining

The access pattern of the array data is not constant, therefore, to avoid memory conflicts we do not allow overlapping of different iterations of the outer loop. The approach is conservative, but the decrease in speed is not significant. Thus, as shown in Figure 9, a stall is needed between different iterations of the body of LOOP 2.

Since at every new iteration of LOOP 1 we spend the time for a new start up of the pipeline, the speed loss due to this stall is proportional to the ratio between the number of pipe stages - 1, and the number of iterations of LOOP 2.

After completing the design for 1 multiplier and 6 adders/subtractors using loop body pipelining, we obtain a throughput of 4569.6ns, which is significantly better than the one obtained from the totaly non-pipelined design (with the same number of multipliers). By increasing the number of multipliers to 4, we obtain an even higher performance of 1209.6ns.

5.3 Functional Unit Pipelining.

The functional units implementing the operations from the HDL description can also be pipelined. During functional unit allocation, a pipelined multiplier from the components library can be chosen. For example for the multiplication from the FFT algorithm we can use a 4 pipe-stage multiplier, as shown in Figure 10. The lightest dividing lines show the pipe stages of the multiplier. In this case the number of stages of the body of LOOP 2 increases from 5 to 8. The FFT computation presented here is pipelined at all the three levels: process, loop body and functional unit.

Figure 11 shows the timing diagram for the functional unit pipelining. Besides the process and loop body pipelining, the stage which contains the multiplier is divided into four pipe stages, which execute in parallel on different data. The multiplier is fed with consecutive iterations of the inner loop, therefore the pipe stages of the multiplier will execute in parallel on different iterations of the loop. Thus, the functional unit pipelining operates on the same data granularity as the loop body pipelining.

Functional unit, loop body and process pipelining can be combined in any way. For brevity we only presented three design options.

```
ReadIn();                                                                              STAGE 1
-----------------------------------------------------------------------------------------------
nn := 64; n:= 128;
mmax:=2;                                                                               STAGE 2
while (mmax<n)    ---- *********** LOOP 1 ************
loop
      istep:=2*mmax;
      m:=0; m_rom := 0;
      i:=0;
      for k:=0 to nn/2-1 loop      ---- ************** LOOP 2 **************
            i_save:=i;
            j_save:=i+mmax;


            wr:=ROM_real(m_rom);                         -- MEMORY READ
            wi:=ROM_imag(m_rom + 1);
            c := data(j_save); d := data(j_save+1);                          stage 1
            -----------------------------------------------------------------------
            e := wr * c; f := wi * d;  g := wr * d; h := wi * c;              stage 2
                                                                              stage 3
                                                                              stage 4
                                                                              stage 5
            i := i + istep;
            -----------------------------------------------------------------------
            tr := e - f; ti := g + h;                    -- ADD/SUB           stage 6
            a := data(i_save); b := data(i_save+1);

            if (i>=n) then
                  m:=m+2; m_rom := m_rom + nn/mmax;
            end if;
            -----------------------------------------------------------------------
            z1 := a - tr; w1 := b - ti;                  -- ADD/SUB           stage 7
            z2 := a + tr; w2 := b + ti;

            if (i>=n) and (m>=mmax) then m:=0; m_rom := 0;
            end if;
            -----------------------------------------------------------------------
            data(i_save) := z2; data(i_save+1):= w2;     -- MEMORY WRITE    stage 8
            data(j_save) := z1; data(j_save+1) := w1;

            if (i>=n) then
                  i:=m * 2;
            send if;
      end loop;
      mmax:=istep;
end loop;
-----------------------------------------------------------------------------------------------
WriteOut();                                                                            STAGE 3
```

FFT Chip

Figure 10: Functional unit pipelining.

The design in Figure 6 shows only process pipelining, the one in Figure 8 shows process and loop body pipelining, whereas the description in Figure 10 presents process, loop body and functional unit pipelining.

The design using FU, loop and process pipelining with 1 pipelined multiplier has a throughput of 3855.6ns, the one with 4 pipelined multipliers has 819ns, and the one with 8 pipelined multipliers has 483ns. This last design is the fastest from our exporation. It has the highest cost both in terms of functional units, and pipeline registers.

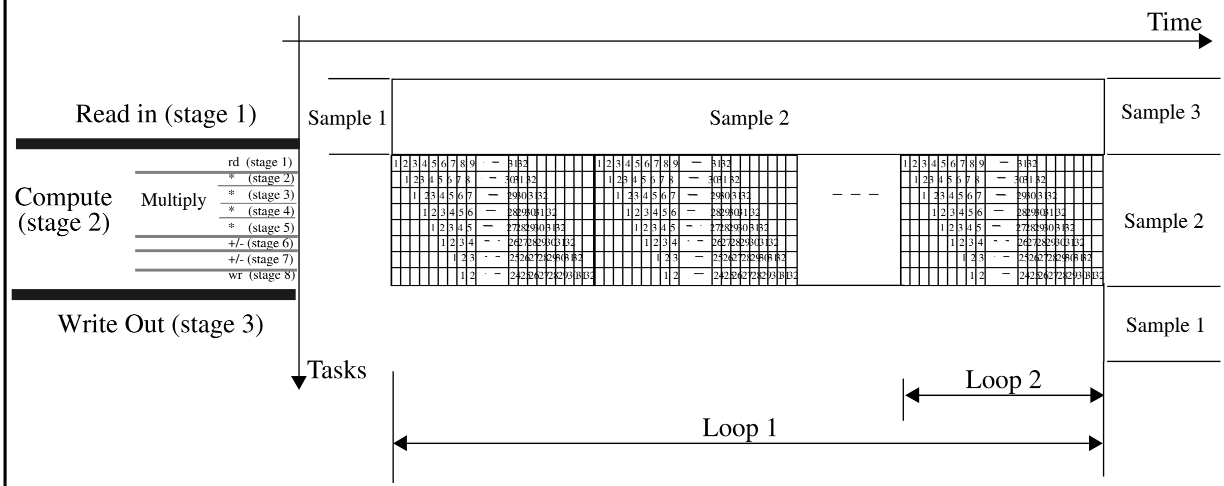

Figure 11: Timing diagram for functional unit pipelining

6. Datapath and Controller.

In section 3 we presented two different communication models. On the other hand, any communication model can be combined with any pipelining alternative. For brevity, we only show the detailed implementation of the communication model 1 using process and loop body pipelining.

At process level the algorithm is divided into 3 pipe stages. These stages execute in parallel on consecutive data samples. To store the sample computed by stage 1 for subsequent use of stage 2, we need a memory unit. We denote it as RAM1. Since the stage 1 writes in the memory unit while the stage 2 reads it, to avoid conflict we have to divide the memory unit into 2 parts: one for reading and one for writing. A swapping mechanism between the two parts can be implemented. The same considerations apply for the memory between stage 2 and stage 3. This memory is denoted RAM2.

6.1 Datapath and Controller of the FFT Computation.

As shown in section 2, the FFT algorithm consists of a set of butterfly computations, where the input is multiplied with a set of coefficients and then stored back into the memory. We consider the coefficients to be read from a ROM memory, containing both the real and imaginary parts of the values.

At process level, the computation represents the stage two of the algorithm. It reads the data from RAM1 and writes it to RAM2, and consists of two nested loops, with the innermost computing one butterfly of the FFT. We consider the body of the inner loop pipelined, as shown in section 5.2.

We divided the FFT computation datapath into two parts: the address generation, and the butterfly computation. The address generation part comprises the calculation of the indexes of the loops, along with the computation of the final addresses for the RAM and ROM memories.

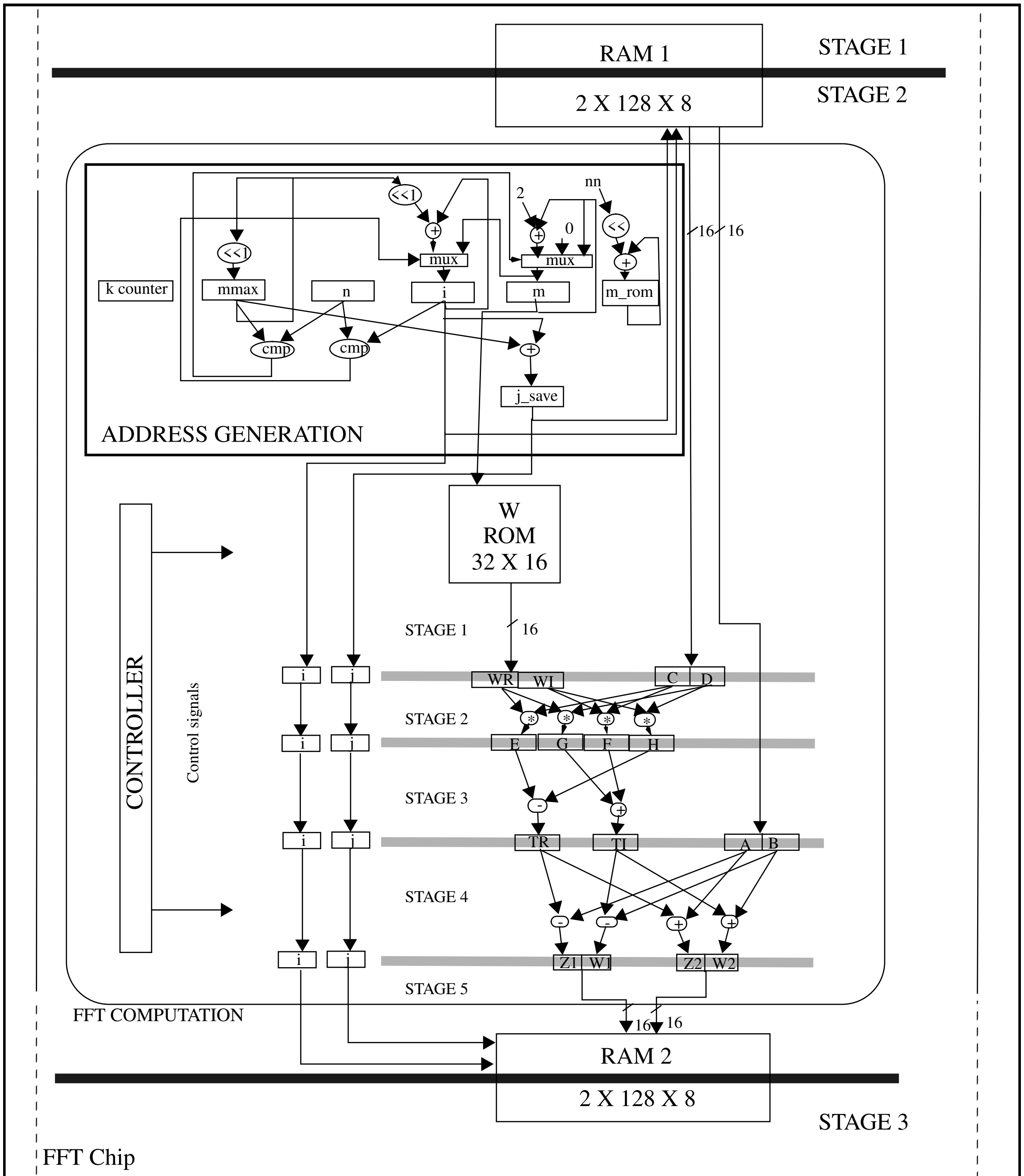

Figure 12: FFT computation datapath.

The butterfly computation consists of the multiplications and additions performed in a 5-stage pipeline.

Figure 12 shows the datapath of the FFT computation.

6.2 Interface between FFT chip and its environment

In this section, we show schematic diagram for the interface between the FFT chip and its data sender and receiver.

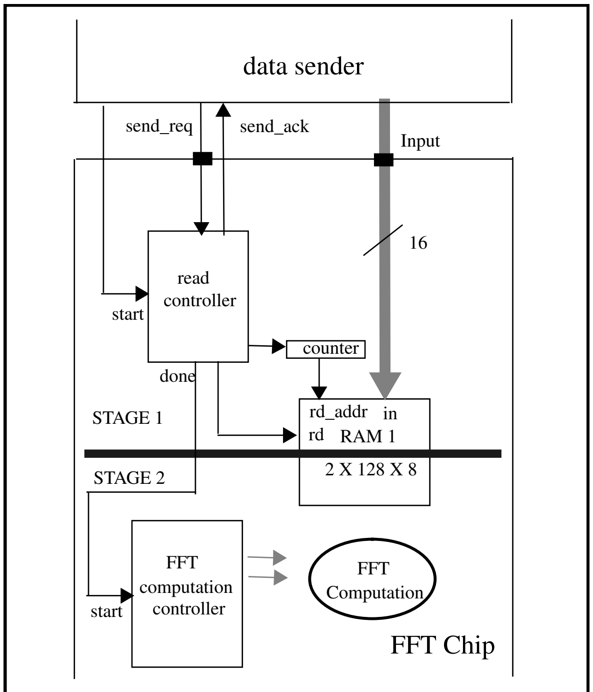

Figure 13: Interface between Sender and FFT Chip

6.2.1 Input Interface

Figure 13 shows the interface between the FFT chip and its data sender. For simplicity, only the stage 1 and stage 2 of the FFT chip are shown in the diagram. the data bus consists only of 16-bit input data which is a complex word for input sample sequence. The control bus consists of 3 wires: the start signal starts the FFT chip, while the send_req and send_ack signals work for 2-wire handshaking protocol. The FFT computation starts only when its controller sees a start signal event, which is the done signal of the first stage.

6.2.2 Output Interface

We now give a detailed description for the data receiving protocol and the design for the write controller for writing the data to the output data bus.

Figure 14 shows the output interface between the FFT chip and its receiver.

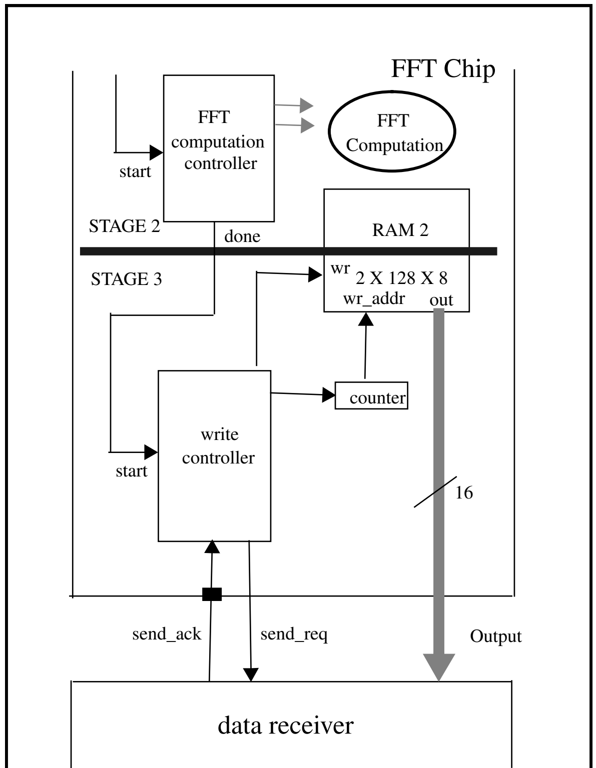

Figure 14: Interface between FFT chip and receiver

Figure 15 shows the timing diagram of the handshaking protocol between the FFT chip and its data receiver. First, the FFT chip sends a start signal which is the done signal of the third stage. Then, after a start-up time (start-up = n cycles), it sends a send_req signal pulse to its receiver to show it is ready to send data. Only when there is a pulse in the send_ack wire, the FFT begins to send data. It put a 16-bit complex word to the output data bus at a certain rate. Here, rate = m cycles. The counter 1 works as the address register for the data RAM.

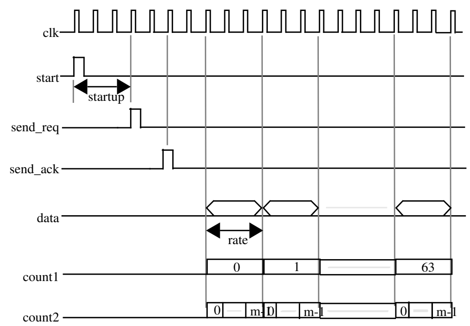

Figure 15: Timing digram for receive protocol

Figure 16 shows the state diagram for the write controller.

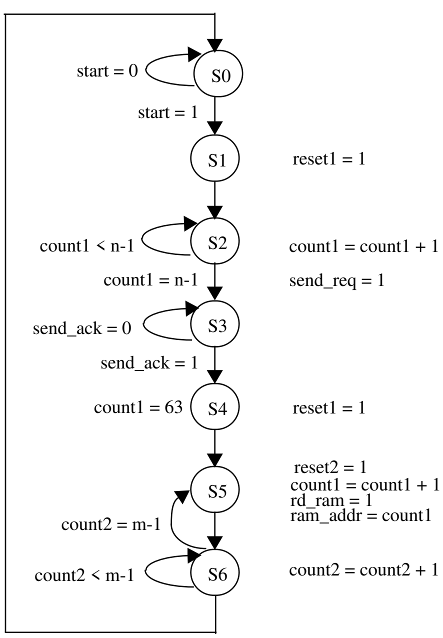

Figure 16: State diagram for FFT chip write controller

7 Parametrization.

The FFT implementation shown in the previous sections assumed that the length of the input and output data are fixed (to a power of 2). The algorithm can be implemented also for a variable length of data. In this section we present the advantages and disadvantages of such an approach.

The length of the data can be read from the input every time the FFT is executed, and it can be stored into a register. Using the value from this register, the number of iterations for the loops, as well as the addressing of the memories can be determined during execution. Therefore, the number of iterations of the loops is also variable.

The FFT coefficients stored in the ROM are different for different length of the data. Therefore, for a variable length we have to store in the ROM the coefficients for all the possible lengths. The addressing of the ROM has to be modified also correspondingly, by adding an offset depending on the current length of the data.

The RAM memories have to correspond to the biggest length of the data allowed.

The execution time of the computation will be variable, depending on the length of the data. The relationship between them is shown in Table 2. Intuitively, for a higher number of elements in the input data, the computation will take longer.

In Table 2 we show the cost and throughput of the FFT computation for variable length data. The value nn in the throughput represents the current length of the FFT input.

No	Pipelining	Clock Cycle	Pipe stage delay	# of pipe stages	Throughput	Multipliers
9	loop 2 body pipelining	5.6	5.6	5	((nn/2) + 4) * log2(nn) * 5.6 ns	4 non-pipe
10	loop 2 body + FU pipelining	3.5	3.5	8	((nn/2) + 7) * log2(nn) * 3.5 ns	4 pipelined

Table 2: Pipelining exploration for variable length FFT computation.

In the parametrized version of the FFT a more complicated addressing of the coefficients is necessary, as well as an increase ROM. Also the total execution time will not be constant. In the case of process pipelining,. this will create an uneven distribution of computation between pipe stages. On the other hand, the parametrized implementation offers more flexibility and increases the reusability of the circuit.

8. Methodology.

Figure 17 shows the design methodology. Starting from an HDL specification of the algorithm, we generate the pipelined RTL description of the implementation.

Determining the communication model depends on the input sample rate and communication requirements of the application.

By pipelining at finer granularity we mean progressively perform process level, loop body and functional unit pipelining, if necessary.

For the parametrized version we determine the execution time for the highest data length allowed.

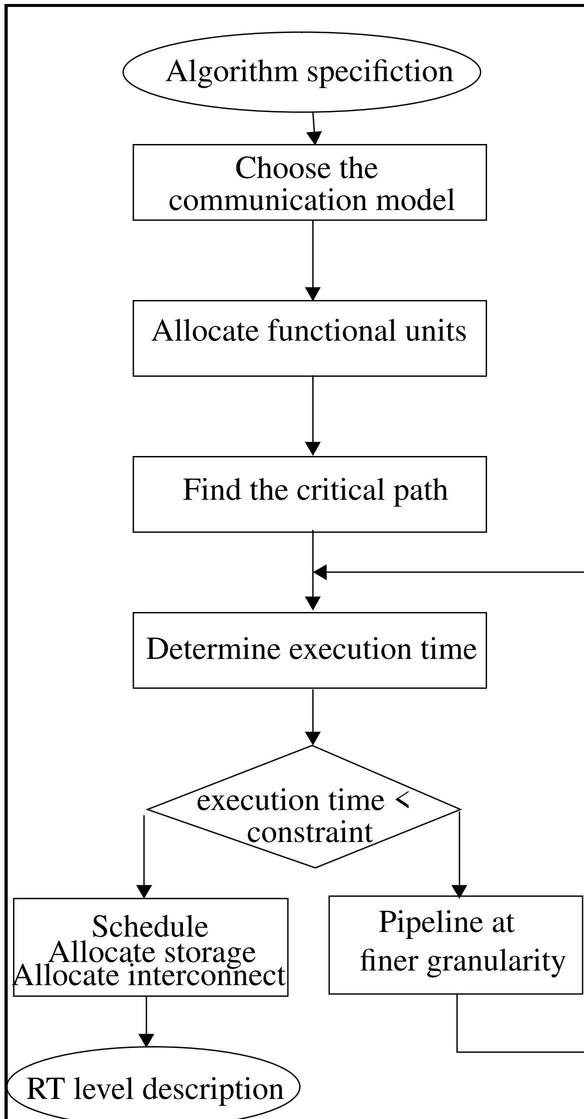

Figure 17: Design methodology

No	Pipelining	Performance				Cost	
		Clock Cycle (ns)	Pipe Stage Delay (ns)	Number of pipe stages	Throughput (ns)	Multipliers	Adders + Subtractors
1	non-pipelined	5.6	-	-	8601.6	1 non-pipe	8
2	loop 2 body pipelined	5.6	5.6*4	3	4569.6	1 non-pipe	6
3	loop 2 body + FU pipelining	3.5	3.5*7	3	3855.6	1 pipelined	6
4	non-pipelined	5.6	-	-	3225.6	4 non-pipe	8
5	loop 2 body pipelined	5.6	5.6	5	1209.6	4 non-pipe	10
6	loop 2 body + FU pipelining	3.5	3.5	8	819	4 pipelined	10
7	loop 2 body pipelining	5.6	5.6	5	672	8 non-pipe	12
8	loop 2 body + FU pipelining	3.5	3.5	8	483	8 pipelined	12

Table 3: Pipelining exploration for the 64 point FFT computation.

9. Lessons Learned.

As shown in the design methodology, starting from a totally non-pipelined design, by progressively performing pipelining at different granularity levels, we obtain different cost/performance trade-offs. Intuitively, the more the design is pipelined, the higher the performance and the cost.

On the other hand, by performing more operations in parallel, we can obtain even higher performance trading off higher cost.

To illustrate this, Table 3 shows the throughput and area of the FFT design for a fixed input length of 64. Rows 1, 2 and 3 represent implementation with one multiplier. Rows 4, 5, and 6 represent 4 multiplier implementations, whereas 7 and 8 use eight multipliers.

10. Conclusions.

We define the FFT chip which can operate in two communication models. We present different pipelining techniques and apply them to the FFT example, showing the algorithm transformations needed. Given a components library we explore multiple design alternatives and compare their performance and cost. Starting from this example, we derive a methodology to apply the pipelining techniques to get the cheapest design while still satisfying the given time constraint.

We also showed the advantages and disadvantages of a parametrized design approach. An increased flexibility and reusability of the chip is obtained against an increase in cost.

11. References.

[1] D. D. Gajski, “Principles of Digital Design”, Prentice Hall 1996.

[2] D. D. Gajski, N. Dutt, A. Wu, and S. Lin, “High Level Synthesis: Introduction to Chip and System Design”, Kluwer Academic Publishers, 1992.

[3] W. Pan, P. Grun, D.D. Gajski, “Behavioral Exploration with RTL Library”, Technical Report, UCI ICS #96-34, July 29, 1996.

[4] D.D. Gajski, P. Grun, W. Pan, “Design Exploration for Pipelined IDCT”, Technical Report, UCI ICS #96-41, September 12, 1996.

[5] P.M.Embree, B. Kimble, “C Language Algorithms for Digital Signal Processing.”, Prentice Hall 1991.

[6] LCB 500K, Preliminary Design Manual, LSI Logic, June 1995.


```
while (mmax<n)
loop
      istep:=2*mmax;
      theta:=math_pi/real(isign*mmax);

sinth:=math.generic_math_functions.sin(theta)
;
      wsr:=-2.0*sinth*sinth;

wsi:=math.generic_math_functions.sin(theta*
2.0);
      wr:=1.0;
      wi:=0.0;
      m:=0;
      while(m<mmax)
      loop
            i:=m;
            while(i<n)
            loop
                  j:=i+mmax;
                  tr:=(wr*data(j))-(wi*data(j+1));
                  ti:=(wr*data(j+1))+(wi*data(j));
                  data(j):=data(i) - tr;
                  data(j+1):=data(i+1) - ti;
                  data(i):=data(i) + tr;
                  data(i+1):=data(i+1) + ti;
                  i:=i+istep;
            end loop;
            tr:=wr;
            wr:=wr+(wr*wsr)-(wi*wsi);
            wi:=wi+(wi*wsr)+(tr*wsi);
            m:=m+2;
      end loop;
      mmax:=istep;
end loop;
for i in 0 to data1'length-1 loop
      data1(i).r:=data(2*i);
      data1(i).i:=data(2*i+1);
end loop;

end cdft;

procedure cdfti (data:inout array1fc;
nn:in integer;
isign:in integer) is

variable s:real;
variable i:integer;
variable u:fcomplex;

begin

      cdft(data,nn,isign);
      s:=1.0/real(nn);
      crmult(data(0),data(0),s);
      for i in 1 to nn/2 loop
            crmult(u,data(i),s);
            crmult(data(i),data(nn-i),s);
            data(nn-i):=u;
      end loop;
end cdfti;

end dsp;
```