6.111 - Introductory Digital Systems Laboratory

Final Project Report:

128-point FFT Computation

(Previously part of Wireless EKG team project)

written by Vanessa F. Hsu Chen

Prof. Anantha Chandrakasan

TA: Jia Fu Cen

May 13, 2004

Abstract

The project consists of computing the Fast Fourier Transform (FFT) of a 128-point signal fed through an A/D converter, and outputted to a 256-address RAM. The algorithm is based on the radix-2 FFT, which computes the FFT for a power of 2, N, number of inputs. The actual algorithm is implemented inside a Field Programmable Gate Array (FPGA), the Altera Flex10k70. The system has an input RAM to store the last 128 input samples, an output RAM where the calculated FFT and a ROM to store pre-computed twiddle factors needed for the FFT algorithm. All these memory blocks are embedded within the FPGA. After each computation of the FFT, the output RAM addresses are read one by one to output the result.

1. Introduction

The project's initial purpose was to receive an EKG signal from an analog EKG sensor, transmit it wirelessly from the remote station to a base station, and compute its FFT, outputting the result to a display. My part of the project was specifically the FFT computation. For this, I decided to do a 128-point FFT, considering both performance speed and resolution as counterweighing factors.

Beyond the scope of the initial project application, the FFT has numerous applications in digital signal processing. There is a constant need for faster and better FFT algorithms. Although I was not aiming to build the most optimal FFT algorithm there is, the implementation of the FFT itself is a good familiarization with such an important algorithm, as well as a considerably non-trivial digital design problem. Namely, the FFT algorithm relies on a divide-and-conquer methodology, which divides the N coefficient points into smaller blocks in different stages. This iterative nature of the algorithm with individual computation of smaller blocks is ideal for a major-minor FSM design. The major FSM in the design controls the main loop of the system, as we go from one stage to the next, while a minor FSM controls the computation of the coefficients within each block.

2. Overview

As mentioned in the introduction, the FFT involves separating the N points into smaller groups. We compute the first stage with groups of two coefficients, yielding N/2 blocks, each computing the addition and subtraction of the coefficients scaled by the corresponding twiddle factors (called a "butterfly" for its cross-over appearance). These results are used to compute the next state of N/4 blocks, which will then combine the results of two previous blocks (combining 4 coefficients at this point). This process repeats until we have one main block, with a final computation of all N coefficients.

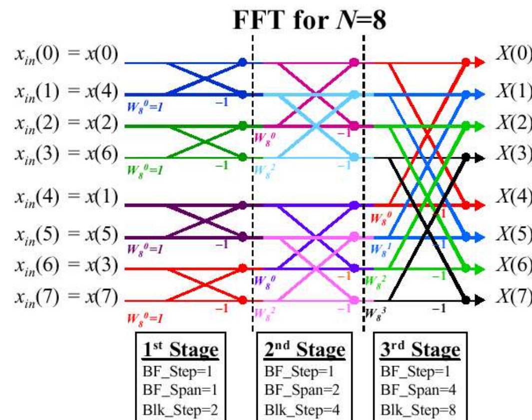

Figure 1. Illustration of FFT stages for an 8-pt FFT

In Figure 1, we can see the different stages. In stage 1, there are 4 blocks, with one butterfly-per-block. In stage 2, there are two blocks with 2 butterflies each; and finally, in stage 3, there is only one block, combining all 8 coefficients with 4 butterflies.

Using a major-minor FSM approach, we have one major FSM, the `fft` block, which controls the general flow of the system. Waiting for the signal, a `BFcomputation` block computes to start the computation within each block, given the right parameters. The minor block can take arguments, such as how many butterflies per block, the step size between different butterflies, or the step size of each block. These arguments allow the `BFcomputation` FSM to be a scalable block that can be re-used for each stage.

Beyond these two main blocks, the system also has to control several memory blocks, which store the twiddle factors as well as the input and output coefficients.

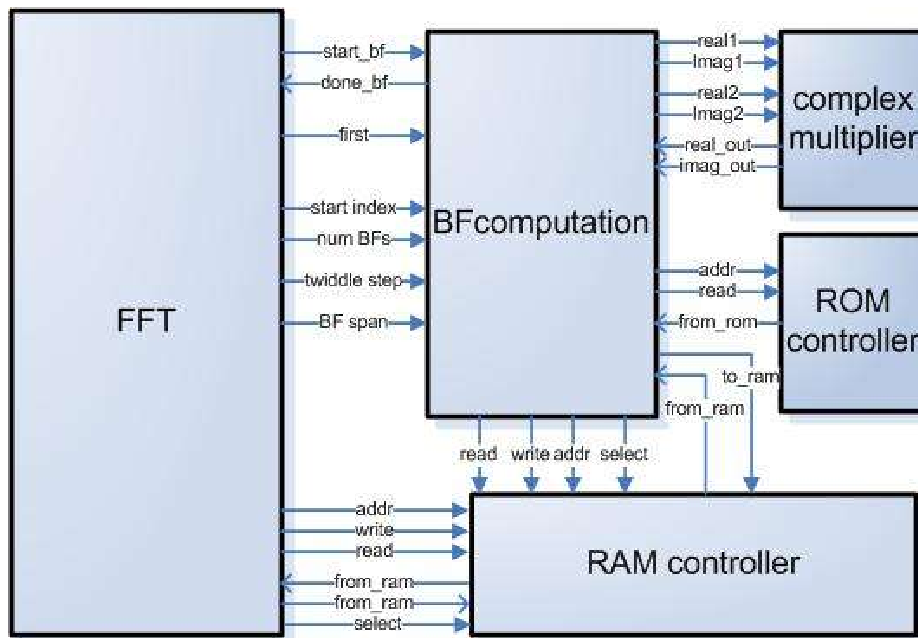

Figure 2. Block diagram for system

3. Design Methodology

As seen in Figure 2, there are smaller modules that ensure the functionality of the FFT and block computation modules. Specifically, the `BFcomputation` module uses a ROM where the twiddle factors are stored.

The twiddle factors are simply the factors $W_N = e^{-j2\pi/N}$. They contain both a real and an imaginary part, making the output coefficients complex numbers as well. These values are stored in a ROM controlled by a `ROMcontroller` minor FSM. This, in turn, is encapsulated in the `BFcomputation` module, since it does not need to be accessed by any other part of the system. Since these numbers repeat themselves after N/2 with a reversed sign, for a 128-point FFT, we only need the first 64 factor. Exactly which factors are used at each point is determined by the algorithm and the explanation for this can be studied with any of the many existing documents about the FFT.[1]

To be able to do complex multiplication, the `BFcomputation` module also includes a `complexmult` module inside, which computes the real and imaginary part of the product of two complex numbers, given their real and imaginary parts. Put together, these modules are in charge of the mechanics 'inside' each of the blocks. The `complexmult` module itself uses the asynchronous multiplier that had been used in a previous lab and problem set.

Finally, there is also a `RAMcontroller` module, which reads and writes from two internal RAM modules. One of these is where the input values are stored, and the other one where the output coefficients are progressively computed and ultimately stored in. The RAM is written on and read by both the `fft` and the `BFcomputation` modules. When the `fft` stores the new input value it writes on the input RAM, and when it finishes the computation of the FFT,

---

[1] http://www.ws.binghamton.edu/fowler/fowler%20personal%20page/EE302_files/Ch_05_09a%20FFT%20Write%20Up_2004.PDF

it reads the output coefficients at the end of the routine to ensure proper functionality. Figure 2 shows the basic block diagram of the entire system. Please note that the embedded Altera RAM and ROM modules are not in the diagram.

The `fft` FSM major starts off in an `init` state, when it is first reset. Here it sets the current address to be 0. After this, we enter the `idle` state, where each time there is an enable signal, sample, the RAM address is driven to be the current address, and the data to the RAM is set to be the input data. Following this, the FSM enters a `write` state where it waits for the RAM to finish writing. After the write, the system is ready to start the `fft` computation. Firstly, it sets up the parameters for the computation of the block. These are: `st_i` (the starting index for the block), `tw_step` (is the difference between the indexes of the twiddle factors used), `nBF` (the number of butterflies in the block) and `bf_span` (the difference of indexes between the two coefficients that the butterfly will take). The system goes through all the coefficients by doing as many blocks as there are per stage (for stage 1 of a 128-pt FFT, it is 64), and then goes to the next stage.

As the algorithm completes each stage, it updates these values to correspond to the appropriate values for that stage. Below is a table of the values for each stage in a 128-pt FFT

Stage →	0	1	2	3	4	5	6
# blocks	64	32	16	8	4	2	1
# BF/blk	1	2	4	8	16	32	64
BF_span	1	2	4	8	16	32	64
tw_step	64	32	16	8	4	2	1
blk_step	2	4	8	16	32	64	128

Table 1. Parameters for block computation

After each block starts, the `fft` FSM waits for it to be done before calling the next block. Once done with each block of stage 1 (k = 0), the parameters are updated in the `iter` state; and the system moves to stage 2, in this case. When all 7 stages are finished, the `fft` FSM goes to an out_ram mode where it reads the current address and continues to read each of the addresses in the output RAM to ensure proper functionality.

In terms of the `BFcomputation` FSM, the system just goes through a sequence of RAM reads (call them c_i and c_j) , and then computes their addition and subtraction, making these the new coefficients. The general control flow of the FSM goes as follows:

- Setup address for real part of c_i, wait for the RAM to be done reading
- Setup address for imaginary part of c_i, wait for the RAM to be done reading
- Multiply this first complex number by (127 + 0i)
- Read the real and imaginary parts of the corresponding twiddle factor
- Likewise, read real and imaginary parts of c_j, and then multiply c_j by the twiddle factor
- After this is done, the system just writes the results from of the real and imaginary parts of c_i and c_j writing it one at a time, and again waiting for the RAM to be done each time.

It is worth noting that the first stage of computation has two special conditions. One is that the coefficient indexes are the bit-reversed indexes from the ones where the output of the block goes. So that the ordering of the inputs is: 0, 2, 4, 6, 1, 3, 5, 7 for an 8-pt FFT, for example, while the outputs are still in regular order. This means that the first time, when the input coefficients are used to compute the coefficients at stage 1, the address should be inverted.

Furthermore, during the first stage, the FFT takes the 'base' coefficients from the input RAM. As the computation progresses though, the FFT takes the last stage of the computed coefficients. This means that the `BFcomputation` block receives a signal telling it if it is

computing a block in the first stage or not. If it is, then it reads from the input RAM and stores in the output RAM. Otherwise, it should read *and* write to the output RAM.

It is also important to see that the first coefficient is multiplied by 127. The reason for this is that the twiddle factors are in reality between 0 and 1 in magnitude. However, to have a floating-point number in fixed-point representation, each point was scaled by 127. The scaling of the coefficient without a twiddle factor keeps things in the same order of magnitude. The twiddle factors are scaled by 127 because they are 8-bit numbers in sign-magnitude formula.

Finally, the multiplier takes in an 8-bit number in sign-magnitude form and one in two's complement form. This is because the twiddle factors were pre-computed in Matlab, scaled and finally put in sign-magnitude form. These values were used to initialize the ROM by placing them in a .mif file.

4. *Testing*

The testing of the system involved very detailed simulation of each of the modules separately. This included the complex multiplier the ROM and RAM controllers at the lowest level. After these were working correctly, I simulated and tested the `BFcomputation` block, and finally the FFT. I chose to wrap the complex multiplier and `ROMcontroller` modules by instantiating them inside the `BFcomputation` module. This ensured that once it worked as a whole, the entire functionality could be abstracted as a unit. Likewise, the `RAMcontroller` module controlled both the input and the output RAM by simply using a select signal, which worked as a mux that determined which RAM was being read or written.

The modules all worked fairly easily by themselves as I built up the system, from multiplier to complex multiplier to `BFcomputation`, for example. However, the integration of the three top level blocks: `fft`, `BFcomputation` and `RAMcontroller` was the most difficult part.

The first reason was that once put together, the system took a very long time to simulate and it was hard to check for bugs with only the logic analyzer. Even the actual FFT takes fairly long since it is basically a large number of reads and writes to memory, and this, of course, takes time. In order to test that each of the states was going through correctly, I created another top file ommiting the `BFcomputation` block and always declaring done = 1. This way the simulation could bypass the actual computation of the coefficients and just see if the `FFT` and the `RAMcontroller` worked well together.

Testing is still underway since I have not been able to even output any of the values in the RAM. After the FFT clearly goes through each of the 7 stages, with the different blocks being computed at each stage, the outputted RAM values are all zero. This is strange, since the RAMs are initialized with random values, and even if it was not outputting the right values, it should output values different from zero. These problems still need to be fixed in order to check for proper computation for the FFT. Since my partner did not end up working on the project, there was not an appropriate output for the output coefficients and I had to improvise a way to read from the RAM (going through the addresses one by one to output to a logic analyzer).

5. Conclusions and Future Work

In short, this has been a long and arduous process,l satisfying in some senses and very frustrating in others. I was not able to complete the FFT computation. The reason behind this seems to be that each time a multiplication was computed with a coefficient by the corresponding twiddle factor, the RAM only stores the 8 MSBs from the 16-bit output. The reason behind this is that otherwise, the coefficients would double in width with each stage (for 7 stages, we would end up with coefficients of 512 points!). This is not desiarable. However, taking the 8MSBs seems to assume the producs are big enough, and this does not seem to be the case because the

output is always 0. Since the block are computed by the same module, regardless of what stage it is at, I believe the error lies in the data trimming.

Beyond these final problems, the project ran into other challenges along the way concerning group dynamics and project management. Despite all this, most of the FFT algortihm is correctly implemented, except for the internal data handling at the moment of multiplying the factors. The system does go through each of the 7 stages, varying the number of blocks per stage, and other arguments to compute each state correctly. For example, for the first stage, we take 64 blocks that compute 1 butterfly each, then the next stage has 32 blocks with 2 butterflies each (meaning using 4 points), and so no. Each block computation involves several reads and writes to the memory modules. Since all three memory blocks are internal to the FPGA, these should not return any output at all if there were problems with the interface. It is unfortunate that the FFT is so dependent on just its output, as opposed to a system of several individually tested parts, because I feel like it is an all or nothing outcome.

Another concluding remark is that this FFT implementation is by far not the most efficient one. The computation of 128 points takes several seconds, and ideally a faster FFT would be used with time-sensitive appliations such as the medical monitoring device we first envisioned. It did serve, however, as a good design exercise due to its several layers of nested iterative loops and their good adaptability to the major-minor FSM methodology.

A. Appendix

Note: Code included here is trimmed to show most relevant parts, omitting parameter declarations for example. This is not run-able code!!

```
module bfcomputation (clk, reset, start, first, select, st_i, nBF, bf_span, tw_step,
cur_addr, done_ram, from_ram, to_ram, write_ram, read_ram, done, addr_ram, state);

input clk, reset, start, first;
input [7:0] st_i, nBF, bf_span, cur_addr;
input [6:0] tw_step;
input done_ram;
input [7:0] from_ram;
output write_ram, read_ram, done, select;
output [7:0] to_ram, addr_ram;
output [4:0] state;

[…]

//instantiation of complex multiplier and rom controller modules inside block //
computation block
complexmult mult (….);

romcontroller twiddles ( ….);

[…]

//offset between the real and imaginary parts of a given index
//for example, real-i is in address i, while imag-i is in address (i + 64) for the rom
//and (i + 128) for the ram
parameter rom_offset = 8'd64;
parameter ram_offset = 8'd128;

always @ (posedge clk or negedge reset)
  begin
    if (!reset) state <= IDLE;
    else state <= next;
        select <= select_int; […]
        i <= i_int;
  end

always @ (state or start or done_ram or done_rom)
  begin
        //defaults:
[…]
    case (state)

    IDLE:
      begin
        done_int = 1;
        i_int = 8'b0;
               addr_ram_int = 8'b0;
               addr_rom_int = 8'b0;
               if (start) next = SETUP_C1_RE;
               else next = IDLE;
          end

        SETUP_C1_RE:
          begin
               // first denotes the first stage, where the information is taken from
               the // input. All other stages take it from the output ram (cumulative sum)
               if (first)
                 begin
                       select_int = 1;
                       addr_ram_int2 = st_i + i + cur_addr;
               //bit reversal for first stage
               addr_ram_int =
               {addr_ram_int2[0],addr_ram_int2[1],addr_ram_int2[2],addr_ram_int2[3],
               addr_ram_int2[4],addr_ram_int2[5],addr_ram_int2[6],addr_ram_int2[7]};
                 end
               else
                 begin
```

```
               select_int = 0;
               addr_ram_int = st_i + i + cur_addr;
          end
        read_ram_int =1;
        next = WAIT1;
  end

WAIT_C1_RE:
  begin
        if (first)
               select_int = 1;
        else select_int = 0;
        if (done_ram) begin
          in_re = from_ram;
          next = SETUP_C1_IM; end
        else next = WAIT_C1_RE;
  end

SETUP_C1_IM:
  Begin
        if (first)
          begin
[…]//similar to the past setup
  end
[…] //now setup values for the complex multiplier

WAIT_MULT1:
  begin
        if (done_mult) begin
        next = SETUP_C2_RE;
        c1_re = c_re;
        c2_re = c_im; end
        else next = WAIT_MULT1;
  end
SETUP_C2_RE:

[…] //same as the states to get real and imaginary parts of c1

WAIT_MULT2:
  begin
        if (done_mult) begin
        next = WRITE_C1_RE;
        c2_re = c_re;
        c2_re = c_im; end
        else next = WAIT_MULT2;
  end

//write the new coefficients now, onto the output ram
WRITE_C1_RE:
  begin
        select_int = 0;
        c1_new = c1_re + c2_re;
        addr_ram_int = st_i + i;
        to_ram_int = {c1_new[8:1]};
        write_ram_int = 1;
        next = WAIT13;
  end

WAIT13: next = WAIT14;
WAIT14:        next = WRITE_C1_IM;

WRITE_C1_IM:
  begin
        select_int = 0;
        if (done_ram) begin
          c1_new = c1_im + c2_im;
          addr_ram_int = st_i + i + ram_offset + cur_addr;
          to_ram_int = {c1_new[8:1]};
          write_ram_int = 1;
          next = WRITE_C2_RE; end
        else next = WRITE_C1_IM;
  end

[…] //again for c2
// finally update the block to do the next butterfly in the block
// if all of them have been done, then return to idle state BLOCK DONE!
```

```
        UPDATE:
          begin
                if (i == nBF) next = IDLE;
                else next = SETUP_C1_RE;
          end

        default: next = IDLE;
        endcase
  end
endmodule

module fft (clk, reset, sample, data_in, start_bfcomp, write_ram, first, select, done,
read_ram, cur_addr,
                      to_ram, addr_ram, done_ram, done_bfcomp, st_i, nBF, bf_span,
tw_step, k, m, state);

input clk, reset, sample;
input [7:0] data_in;
input done_bfcomp, done_ram;

output start_bfcomp, write_ram, first, select, done, read_ram;
output [7:0] st_i, nBF, bf_span, tw_step, m;
output [7:0] to_ram, addr_ram, cur_addr;
output [2:0] k;
output [3:0] state;

reg start_bfcomp, write_ram, read_ram, first, select, start_bfcomp_int,
               write_ram_int, read_ram_int, first_int, select_int, done, done_int;
reg [7:0] to_ram, to_ram_int, addr_ram, addr_ram_int, cur_addr, cur_addr_int, write_addr,
write_addr_int;
reg [7:0] st_i, nBF, bf_span, tw_step, nBlk, blk_step, m;
reg [7:0] st_i_int, nBF_int, bf_span_int, tw_step_int, nBlk_int, blk_step_int, m_int;
reg [3:0] state, next;
reg [2:0] k, k_int;

parameter INIT                = 0;
parameter IDLE                = 1;
parameter WRITE               = 2;
parameter SETUP               = 3;
parameter START_BLK    = 4;
parameter WAIT1               = 5;
parameter WAIT2               = 6;
parameter WAIT_BLK     = 7;
parameter ITER         = 8;
parameter OUT_RAM      = 9;
parameter WAIT_READ = 10;
parameter WAIT_READ1 = 11;
parameter WAIT_READ2 = 12;
parameter NEXT_ADDR1 = 13;
parameter NEXT_ADDR2 = 14;

always @ (posedge clk or negedge reset)
  begin
    if (!reset) state <= INIT;
    else state <= next;
        start_bfcomp <= start_bfcomp_int;
        write_ram <= write_ram_int;
        read_ram <= read_ram_int;
        first <= first_int;
        select <= select_int;
        to_ram <= to_ram_int;
        addr_ram <= addr_ram_int;
        cur_addr <= cur_addr_int;
        write_addr <= write_addr_int;
        st_i <= st_i_int;
        nBF <= nBF_int;
        bf_span <= bf_span_int;
        tw_step <= tw_step_int;
        nBlk <= nBlk_int;
        blk_step <= blk_step_int;
        k <= k_int;
        m <= m_int;
        done <= done_int;
  end

always @ (state or sample or done_ram or done_bfcomp)
```

```
  begin
    //defaults
       start_bfcomp_int = 0;
       write_ram_int = 0;
       read_ram_int = 0;
       select_int = 0;
       addr_ram_int = addr_ram;
       cur_addr_int = cur_addr;
       write_addr_int = write_addr;
       to_ram_int = to_ram;
       st_i_int = st_i;
       nBF_int = nBF;
       bf_span_int = bf_span;
       tw_step_int = tw_step;
       nBlk_int = nBlk;
       blk_step_int = blk_step;
       k_int = k;
       m_int = m;
       done_int = 0;
    case (state)

       INIT:
         begin
              cur_addr_int = 8'b0;
              write_addr_int = 8'b0;
              k_int = 3'b0;
              m_int = 8'b0;
              next = IDLE;
         end

       IDLE:
         begin
              k_int = 3'b0;
              if (sample) begin
                select_int = 1;
                addr_ram_int = write_addr;
                to_ram_int = data_in;
                write_ram_int = 1;
                next = WRITE;  end
              else next = IDLE;
         end

       WRITE:
         begin
              select_int = 1;
              cur_addr_int = write_addr + 1;
              if (done_ram)
                if (write_addr == 8'd127) write_addr_int = 8'b0;
                else write_addr_int = write_addr + 1;
                next = SETUP;
         end

       SETUP:
         begin
              m_int = 8'b0;
              if (k == 3'b0) begin
                nBlk_int = 8'd64;
                tw_step_int = 8'd64;
                blk_step_int = 8'd2;
                nBF_int = 8'd1;
                bf_span_int = 8'd1;
                first_int = 1; end
              else begin // k > 0
                blk_step_int = {blk_step[6:0], 1'b0};
                bf_span_int = {bf_span[6:0], 1'b0};
                nBF_int = {nBF[6:0], 1'b0};
                nBlk_int = {1'b0, nBlk[7:1]};
                tw_step_int = {1'b0, tw_step[7:1]};
                first_int = 0;  end
              next = START_BLK;
//            next = ITER;
         end

       START_BLK:
         begin
              st_i_int = blk_step * m;
```

```
                start_bfcomp_int = 1;
                next = WAIT1;
      end

        WAIT1:
          begin
                next = WAIT2;
          end

        WAIT2:
          begin
                next = WAIT_BLK;
          end

        WAIT_BLK:
          begin
                if (done_bfcomp)      begin
                  next = ITER; end
                else next = WAIT_BLK;
          end

        ITER:
          begin
          m_int = m + 1;
            if (m == nBlk)   begin
                  k_int = k + 1;
                  if (k == 7) begin
                        addr_ram_int = cur_addr;
                        next = OUT_RAM; end
                  else
                        next = SETUP;  end
//              else next = ITER;
                else next = START_BLK;
          end

        OUT_RAM:
          begin
                read_ram_int = 1;
                next = WAIT_READ;
          end

        WAIT_READ:
                next = WAIT_READ1;

        WAIT_READ1:
                next = WAIT_READ2;

        WAIT_READ2:
          begin
                if (done_ram) begin
                        addr_ram_int = addr_ram + 1;
                        next = NEXT_ADDR1; end
                else next = WAIT_READ2;
          end
        NEXT_ADDR1: next = NEXT_ADDR2;

        NEXT_ADDR2:
          begin
                if (addr_ram == cur_addr)
                  next = IDLE;
                else begin
                  next = OUT_RAM; end
          end

        default: next = IDLE;

    endcase
  end
endmodule
```