# Device Resources - M2S010/M2GL010

Existing Smartfusion2 device: M2S010/M2GL010

|  |  |
| --- | --- |
| Resource | Available |
| LSRAM 18432bit (512x36, 1024x18) | 2 x ? = 21 |
| µSRAM 64 x 18 bit | 2 x 11 = 22 |
| MACC DSP 18x18 signed mult, 9x9+9x9 DOTP | 2 x 11 = 22 |

# FFT

## Theory

### Layman’s Terms

### Math

## Structure

### Big Picture

### Stage

### Butterfly

## Determining Data Width

The 3 considerations for this project are:

1. Data storage
2. Data math (mainly multiplication)
3. Result accuracy

### 1 Data Storage

µSRAM blocks in Smartfusion2 devices can store up to 18 bits in a single memory location and have dual read ports. In order to maximize resource utilization I’ve elected to store 1 complex number in a µSRAM location as concatenated signed bits. e.g.

Where & indicates concatenation of bits, RS & RD represent the signed 9 bit real value of a complex number, and IS & ID represent the signed 9 bit imaginary value of a complex number.

This would allow 1 µSRAM location to store 1 complex number used in FFT calculations and the dual port nature of the µSRAM block to feed 1 FFT butterfly in a single clock cycle.

### 2 Data Math

The DSP math blocks within a Smartfusion2 device allows for up to 18bit x 18bit signed multiplications. This is more than enough data width for the data being stored in RAM and indeed is more than the data width I expect to be sending to this FFT.

More interestingly is the Dot Product (DOTP) mode of the DSP blocks. This allows a single DSP block to perform the calculation:

Where A, B, C, and D are 9 bit signed numbers and P is the signed 18 bit result.

This is useful for finding the complex magnitude of the result so it can be sent to a human readable interface (LCD screen) or some other use. This also fits very nicely with the configuration of the data in the µSRAM blocks and all told allows 1 DSP block (fully utilized in DOTP mode) to provide the final complex magnitude result for 1 frequency bin.

Having decided that the data within the FFT will occupy pairs of 9 bit signed numbers. We must work backwards and determine the input data widths.

|  |  |
| --- | --- |
| Decimal | Binary |
| 255 | 1111\_1111 |
| 65025 | 1111\_1110\_0000\_0001 |

Remember that the twiddle factors represent cosine and sine values that range from 1 to -1. This is key. When 2 numbers of 8 bits are multiplied together, the result will be a 16 bit number:

More generally the bit width is but we’re intending to use same length numbers.

Therefore, on any butterfly multiplication we are multiplying our data at 9bits signed with a sine or cosine value that has been scaled up (literally or ) to be a 9bit signed number. However, as mentioned before, our sine and cosine values are really only between 1 and -1, I have scaled them up so they can be represented in an integer format. Therefore, each result of the multiplication can be divided by the same amount:

;

;

;

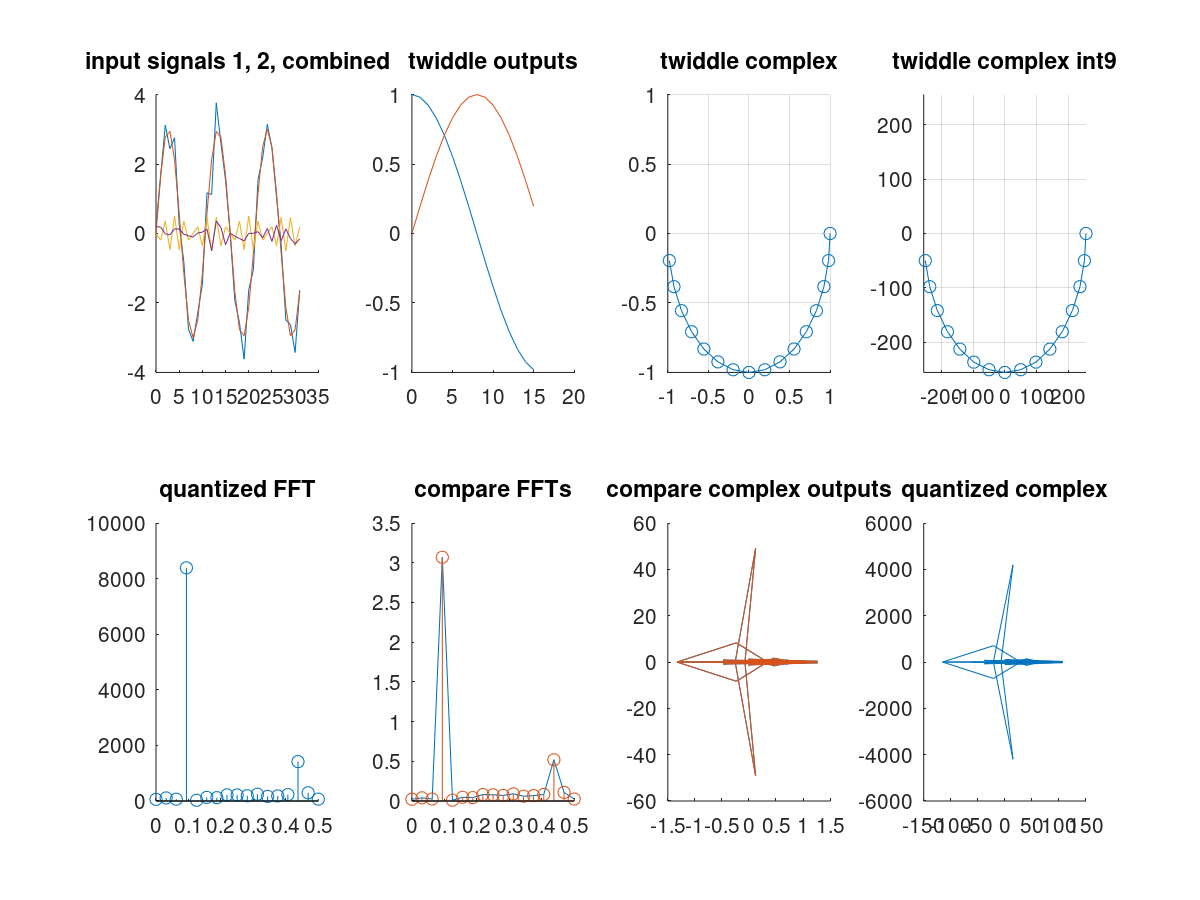
;

;

As can be seen this is not an exact process but the results should remain within the resolution of the 9bits signed, it’s the best that can be done.

### 3 Result Accuracy

Limiting performing on an FFT transform on 5 bit signed data seems like it would be limiting.



* Data bits = 8
* Input signals:
* Twiddle outputs:
  + Visualize the parts of the Cosine and Sine used as twiddle factors
* Twiddle complex:
  + Cosine and Sine values represent real and imaginary values respectively of complex numbers. This plot visualizes the twiddle factors used along the unit circle.
* Twiddle complex int9:
  + Cosine and Sine values quantized to represent 9bit signed values. This plot visualizes the scaled and quantized twiddle factors along the scaled unit circle.
* Quantized FFT:
  + Results of GNU Octave (Matlab) script to determine FFT results using integer operations. This is also scaled to my desired output resolution of 5 bits.
* Compare FFT:
  + Results of the built in FFT() function as well as a floating point version of my FFT implementation. My FFT version was then modified to produce the quantized FFT plot.
* Compare Complex Outputs:
  + Similar to the Compare FFT plot, this takes the real and imaginary parts of the FFT results and graphs them, just to see what differences there might be.
* Quantized Complex:
  + Plot of the complex results of the quantized FFT.

### Conclusion

## Implementation

### Example Project

A proof of concept project that will use this FFT core takes samples from an LVDS based Delta Sigma Analog to Digital Converter (Δ∑ ADC) on a Smartfusion2 device. This ADC core will be set to produce 8 bit unsigned samples at a sample rate of 195.313 kHz.

#### Output Frequency Information

Our FFT output will produce frequency bins centered on where and N is the input sample size. Each frequency bin will have a bandwidth of . An exception exists for the first and last frequency bins which have an effective bandwidth that is half of the others resulting in a bin bandwidth of .

I will be using an FFT with 256 input sample length which will give 128 frequency bins with a core bin bandwidth of 762.9Hz. The reasoning for this FFT length is because the output LCD only has 84 pixel columns so any information beyond 84 bins is going to be thrown out anyway.

#### Speed Considerations

Since this project doesn’t have any hard requirements for processing time. I’ll start by determining how many clock cycles it takes to fill the FFT sample memory and see if I can get the FFT processing time to stay within that window.

I’ve already determined that ADC sample frequency will be 195.313kHz. At 256 samples, this will take . With a main clock speed of 100Mhz I will have clock cycles to perform the FFT.

Assuming 1 cycle to read from the µSRAM blocks into working registers (remember we’re overwriting the µSRAM locations so we need a temporary location), 1 cycle to perform the DSP calculation and write the result into 1 µSRAM location, 1 cycle to write to the other location; each butterfly calculation including reads and writes should take 3 clock cycles.

With N = 256 samples, we have 8 FFT stages and 128 butterfly calculations per stage. This gives a total number of butterfly calculations as . At 3 clock cycles per butterfly we get 3072 clock cycles.

3072 clock cycles is far below the 131,072 clock cycle window and provides the option to increase the FFT length or reduce the number of DSP blocks used if desired.

Another approach is to take the window and work in the opposite direction. 131,072 clock cycles can support 43,690 butterfly calculations. Using the formula from above the nearest N value that gives a butterfly count below 43,690 is giving a total butterfly count of 24,576. This is also before any parallelization (beyond the internals of the butterfly).

#### Resource Usage

256 samples fits nicely into 4 µSRAM blocks, and since each µSRAM block has 2 read ports, they can directly feed a total of 4 butterfly multiplications (1 DSP per µSRAM) in a single clock cycle. A bottleneck is that there is only 1 write port per µSRAM block, so regardless of any other speedup options, any math operation the modifies 2 µSRAM locations (e.g. FFT butterfly) will require 2 clocks to write that data back into storage.

### Module Descriptions

#### RAM Interfacing

RAM blocks will be passed around to multiple modules: sample loading, the FFT process, and output ports. The following is a common set of interface ports that each module must accommodate.

|  |  |  |  |
| --- | --- | --- | --- |
| Port | Size | Dir | Function |
| Ram\_stable | 1 | In | Indicates the RAM connections are all connected. Indicates situations where synchronous reset makes sense. |
| Ram\_ready | 1 | In | Indicates the RAM block is ready to be written to. Indicates situations where the process should pause. |
| Ram\_valid | 1 | Out | Sets the current RAM block as containing invalid/corrupted data |
| Ram\_adr | A | Out | Array of vectors that set the RAM address to be read or written for each RAM port |
| Ram\_w\_en | B | Out | Array of pins the control the write enable for each RAM port |
| Ram\_dat\_w | C | Out | Array of vectors for the data being written to each RAM port, synchronous based on ram\_adr and ram\_w\_en. |
| Ram\_dat\_r | C | In | Array of vectors for the data being read from each RAM port, asynchronous based on ram\_adr. |

1. RAM port count \* address width
2. RAM port count \* 1
3. RAM port count \* data width \* 2

#### Sample Loader

This module accepts data write requests from an external sample generator and writes the data into the RAM block connected to it by the FFT controller.

#### Butterfly

The key component of the butterfly component is the hardware MACC block within the Smartfusion2 FPGA fabric. These blocks have a variety of configurations but the one we’re interested in is the Dot Product mode. This allows the equation to be performed as a single “operation” within the Math block.

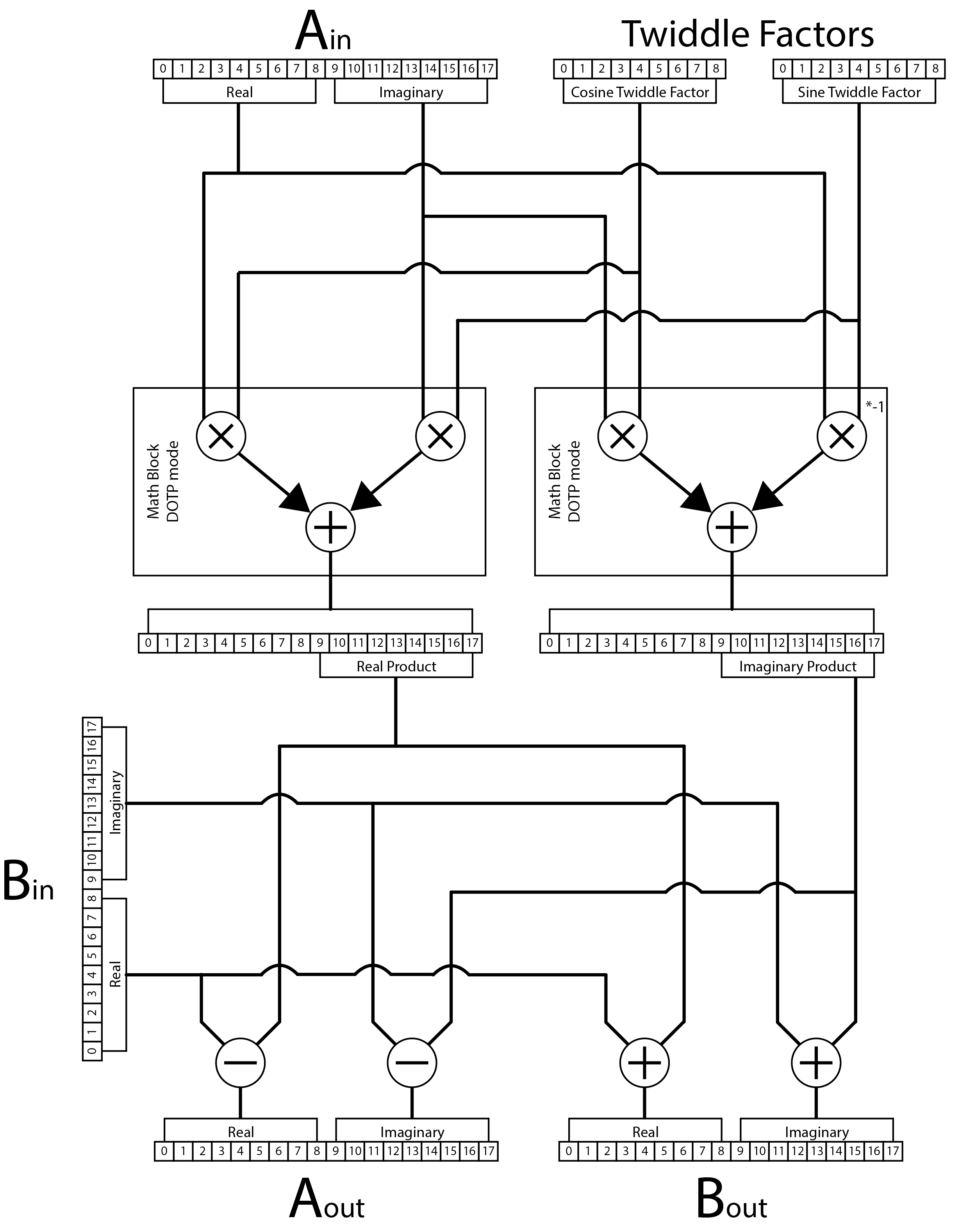
There are 2 major hurdles to designing this however.

The first is that the second equation that gives the imaginary result is , notice the subtraction sign. The DOTP mode does not accommodate subtraction in the necessary step and so we must find a work around. The Fabric User Guide provided by Microsemi gives a solution:

The overarching method is to take the two’s complement of Q and feed that in, thereby introducing the necessary negative sign. While the negation step is cheap, adding the +1 requires arithmetic blocks and waiting for the carry bit to propagate. The Math block has a C input which feeds directly into the internal 3-input adder. Since we’re already limited by the internals of the Math block, we might as well use all of it.

The second hurdle is in writing the code. Synplify does not support “Dynamic add/sub support in Dot Product mode.” This is what we’re doing and so we must find the necessary components and add them directly to the Butterfly component list ourselves.

Synthesizing the butterfly gives the slowest path as being from a flipflop that feeds directly into the Math block to the flipflop that stores the output. The result is a path delay that is 62% from logic and an estimated maximum frequency of 212Mhz. Hopefully this will be the bottleneck in the design when it comes to the frequency.



#### The Transform

The transform is governed by several key values that are iterated over as nested loops. In a sequential programming language (C, Python, etc.) this would be 3 for loops nested. The outer loop iterates over each stage of the FFT. The Middle loop iterates over each DFT in the current stage, and the Inner most loop iterates over each Butterfly calculation in the DFT.

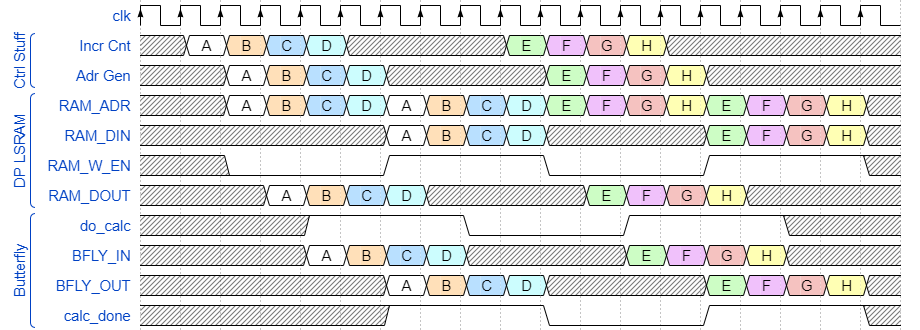
One characteristic of the FFT is that within each stage, the total number of butterfly calculations remains the same. A simple explanation is that each sample is acted upon exactly once each stage. Additionally as each butterfly calculation takes 2 samples, this means that the number of butterfly calculations per stage can be calculated as .

Part of the FFT process is to increase the size of the DFT each stage. The first stage will have 2 sample DFTs while the last stage will have a single N sample DFT. As the sample size, and therefore butterfly count, remains the same, the butterfly calculations per DFT will have to change to remain consistent.

|  |  |  |  |
| --- | --- | --- | --- |
| Variable Identity | Equation | Bits Required | Implementation Notes |
| Stage Count | ; aka Sample Exponent |  | Set by constant |
| DFT Count |  |  | Start all ‘1’, shift right |
| BFly Count |  |  | Start 0, shift left add 1 |
| BFly Sample A adr |  |  | Start DFT shifted left, add BFly each loop |
| BFly Sample B adr |  |  | Start 1, Shift left and add Sample A |
| Twiddle Index |  |  | Start 0, add TwiddleStep each BFly |

In order to maximize throughput our goal is to reduce the clock cycles between sending samples to our Butterfly component.

There are 2 ways I see to approach this. As our butterfly is pipelined, both approaches will incorporate pipelines to feed the butterfly. The important difference is what determines when the feeding stops, and the reading begins. Below is a wave chart for a pipeline implementation using a single dual port SRAM and a single butterfly.

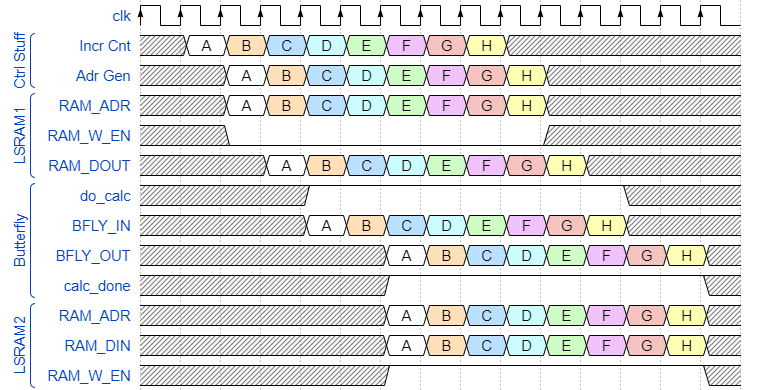


Notice the RAM\_ADR line is at capacity. This is our system bottleneck when it comes to the number of clocks. Every clock cycle the SRAM is either writing or reading from its memory. This is fine so long as there isn’t a strict limit on the number of clock cycles available to perform the transform.

Something you may also notice is that the beginning of the pipeline depends on the end of the pipeline.

One final aspect that is not apparent from the wave chart is that our input module is operating as a circular memory buffer. If it was a fixed size buffer with no wrap around we could do a simple bit reverse on the address when the data is written, thereby writing the data directly into bit reverse sorted order. The circular memory complicates this, if we do not include a separate step to do the sorting, the design will have to accommodate the shifted out of order addressing throughout the FFT Core. As there’s only 2 ports to work with.

The second approach is to use a second LSRAM and bookend the butterfly pipeline.



This can accommodate random order on either side so sorting can be done in line. There is no stopping of the pipeline feed. And the throughput is doubled.

As mentioned previously, there are parts of the FFT process in which 4 butterfly calculations will require the results of multiple sub DFTs. An example is during the very first stage of the FFT where each DFT only performs a single butterfly calculation. This means to achieve maximum throughput we cannot spend clock cycles between sub DFTs. This is important when thinking about how to implement the DFT in code.

Another reminder is that each sample is only used once in a given stage. This means if we can calculate the butterfly inputs based on both the DFT and butterfly count in a single cycle, we’re golden. A concern is the logic depth.

# Resources

|  |  |  |  |
| --- | --- | --- | --- |
| Fabric User Guide | Libero SoC | Real Operation | Imaginary Operation |
| AH | A1 | Real | Imag |
| BL | B0 | Cos | Cos |
| BH | B1 | Sin | -Sin |
| AL | A0 | Imag | Real |

## FFT

<http://www.dspguide.com/>

<https://www.ecse.rpi.edu/~rjradke/dspcourse.html>

<http://www.ws.binghamton.edu/Fowler/Fowler%20Personal%20Page/EE302_files/FFT%20Reading%20Material.pdf>