The FPGA, AXI, Etc…

Administration

- Pset 09 was due today
- No more ”Psets”….what are a series of final project items (in parallel with the las few lab pieces)
- Lab 04B Due tonight
- Lab 05 is out tonight. Due in TWO WEEKS (last lab)

Final Projects

- Teaming to be organized by this weekend.
- Project Abstract (few paragraphs) due by next Tuesday the 18th…go from there.
- Details on the final projects page on site.

What to do for a Final Project?

- Something that an FPGA would Actually get used for…
  - Codec (mp4, mp3, jpeg, and many others!)
  - Accelerators (do some task efficiently)
  - Real-time audio processing (today is simple example)
  - Graphics
  - Signal Processing (graphical or audio)
  - Vision (object detection, tracking)
  - Prototype CPU, TPU, GPU architectures
  - Cryptography
  - High Speed Controller
  - Communication (ethernet…lab05)
  - Inference/detection
  - Decisions

What to do for a Final Project?

- Something an FPGA would not get used for in real life:
  - Video game…
  - Video game…

If you want to do a game…

- If you want to do a game, go hard with it:
- Try to explore more FPGA-relevant topics such as:
  - 3D graphics?
  - Ray-casting
  - Video Processing?
  - Inference

Complexity

- The complexity must come from stuff you do!
- You cannot take lab04b and then lab05, and stitch them together and have an A project.
- The final project will be graded on what you did and contributed.

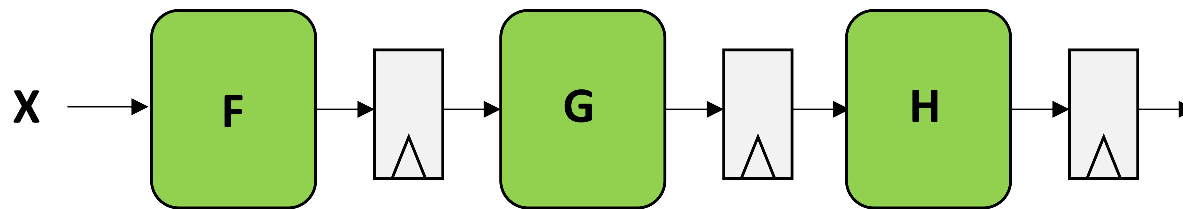
More Details and Timeline on

- https://fpga.mit.edu/6205/F22/final_projects

More Advance Pipelining

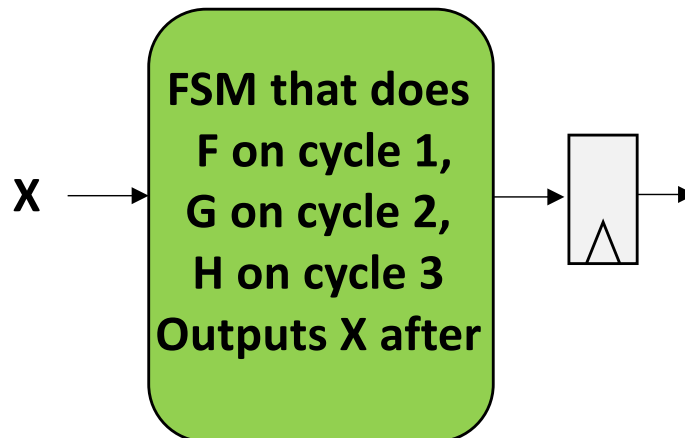
Do We Fully Pipeline an Algorithm or Not?

- Let's say we need to compute F(G(H(X))). Do we build our hardware like this?:



Latency: $3*T_{clk}$
Throughput: $1/ T_{clk}$
Uses more resources

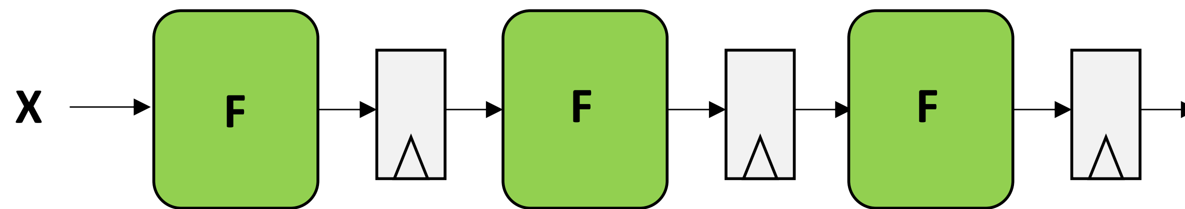
- Or like this:?



Latency: $3*T_{clk}$
Throughput: $1/ (3*T_{clk})$
MIGHT use fewer resources

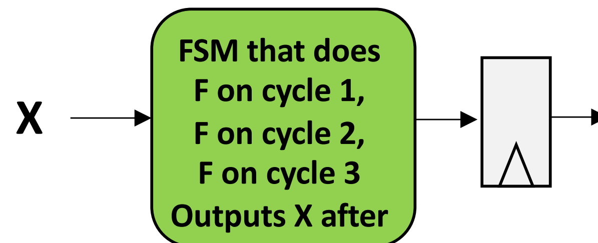
A lot of Algorithms are Repeition-Based though

- Let's say we need to compute F(F(F(X))). Do we build our hardware like this?:



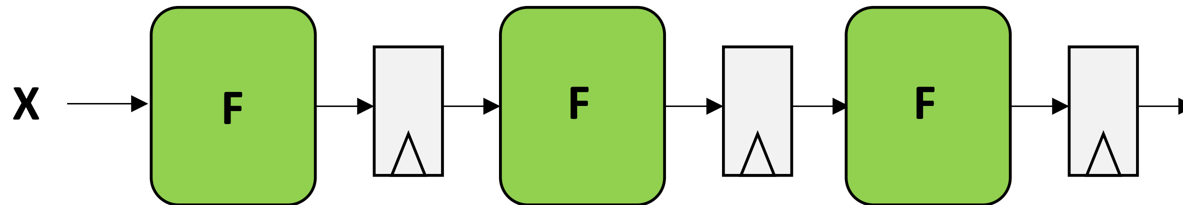
Latency: 3*T_{clk}
Throughput: 1/ T_{clk}
Uses more resources

- Or like this:?



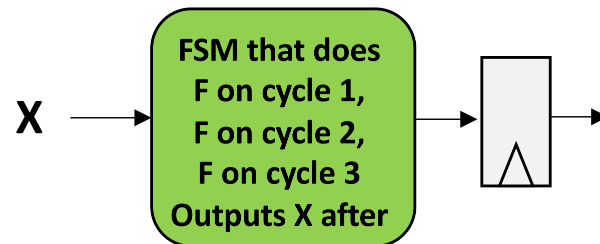
Latency: 3*T_{clk}
Throughput: 1/ (3*T_{clk})
Likely uses fewer resources

This is the Great Tradeoff!



More resources,
Better Throughput
Same Latency

OR

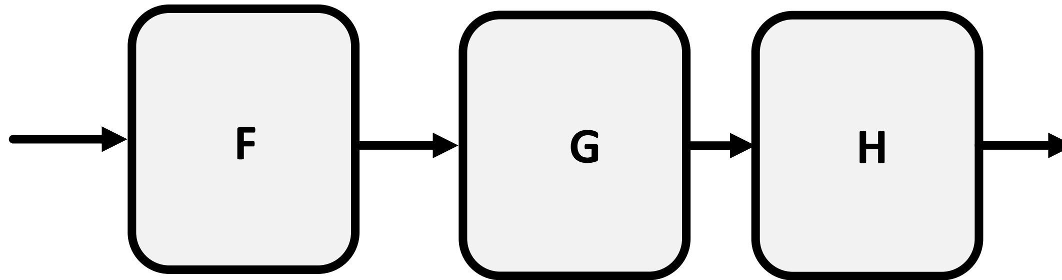


Fewer resources,
Worse Throughput
Same Latency

- Base on what you need for the design!

Pipelining II

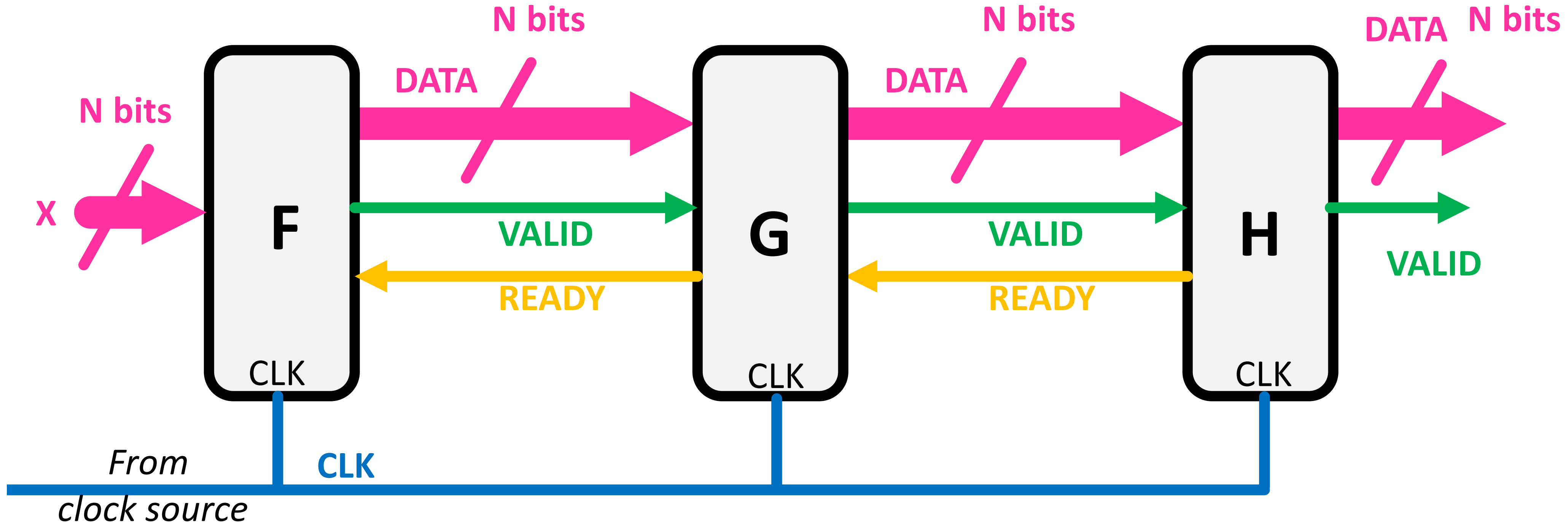
- As we make larger-level systems



- As we make larger-level systems we need to pipeline data through systems which might take varying amounts of time

Pipelining II

- Mixing our Major/Minor FSMs with Pipelining!
- Need a way to send data *downstream*, but also convey preparedness *upstream*

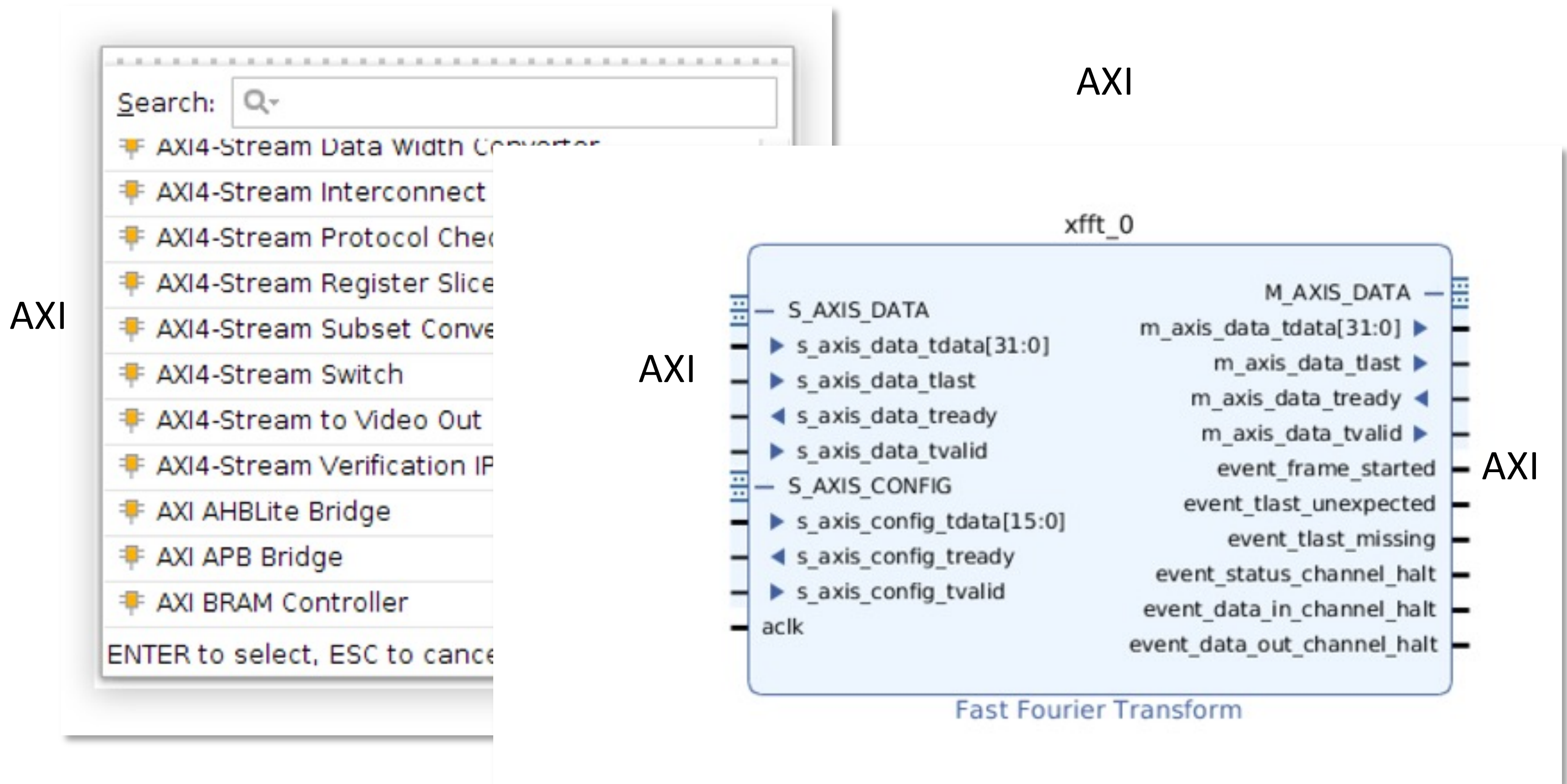


What is IP?

- In Vivado there are IP which you can instantiate.
- These basically let you specify an extremely parameterizable module
- There's a ton of effort that goes into enabling a particular circuit in a modifiable way
- Some companies actually do this:
  - Create a particular design-development platform
    - Example: a pipelined algorithm implementation
  - Sell/lease to Xilinx
  - When people use your design process in their products they give you licensing fees.

AXI Everywhere

- There's lot of neat IP (FFT, more complicated math, etc…)
- Xilinx IP and many others generally use an AXI communication protocol



Advanced Microcontroller Bus Architecture (AMBA)

- Version 1 released in 1996 by ARM
- 2003 saw release of **A**dvanced e**X**tensible **I**nterface (AXI3)
- 2011 saw release of AXI4
- There are no royalties affiliated with AMBA/AXI so they're used a lot.
- It is a general, flexible, and relatively free* communication protocol for development

https://en.wikipedia.org/wiki/Advanced_Microcontroller_Bus_Architecture

Three General Flavors of AXI4

- **AXI4 (Full AXI):** For memory-mapped links. Provides highest performance.
  1. Address is supplied
  2. Then a data burst transfer of up to 256 data words
- **AXI4 Lite:** A memory-mapped simplified link supporting only one data transfer per connection (no bursts). (also restricted to 32 bit addr/data)
  1. Address is supplied
  2. One data transfer
- **AXI4 Stream:** Meant for high-speed streaming data
  - Can do burst transfers of unrestricted size
  - No addressing
  - Meant to stream data from one device to another quickly on its own direct connection

From the Zynq Book

Note on Terminology

- In device-to-device communication, it is common to have:
  - one device labeled the ”Master” and
  - one labeled the “Slave”
  - the Master controls the Slave(s) in these settings.
- Trace history of this naming terminology back to 1940s
- This should be changed…happening slowly I’ve seen some alternatives suggested: Leader/Follower, Main/Secondary (other ideas?), but this naming scheme persists in the field and on data sheets
- Movement from this terminology is slowly happening…Petition for SPI communication here!!:
  - https://www.oshwa.org/a-resolution-to-redefine-spi-signal-names/

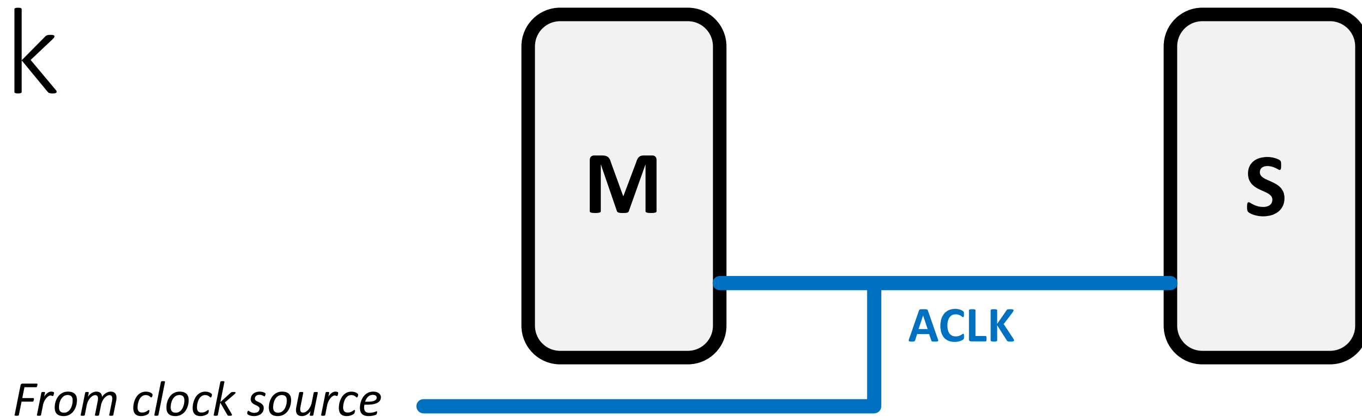
Note on Terminology

- The Xilinx AXI protocol uses this terminology
- In 6.205 I'm going to just use Main/Secondary or just "M" and "S"
- This way we can keep using the datasheets
- And then push Xilinx to change it

AXI:

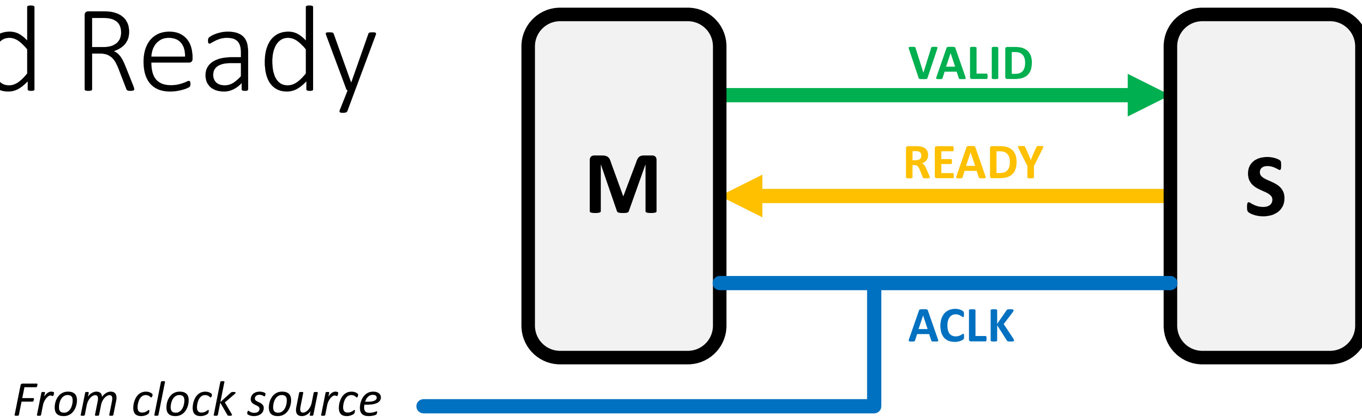
- Made up of wires
- These wires serve specific purposes.
- Some are universal to all AXI4S channels, and others are specific

AXI Clock



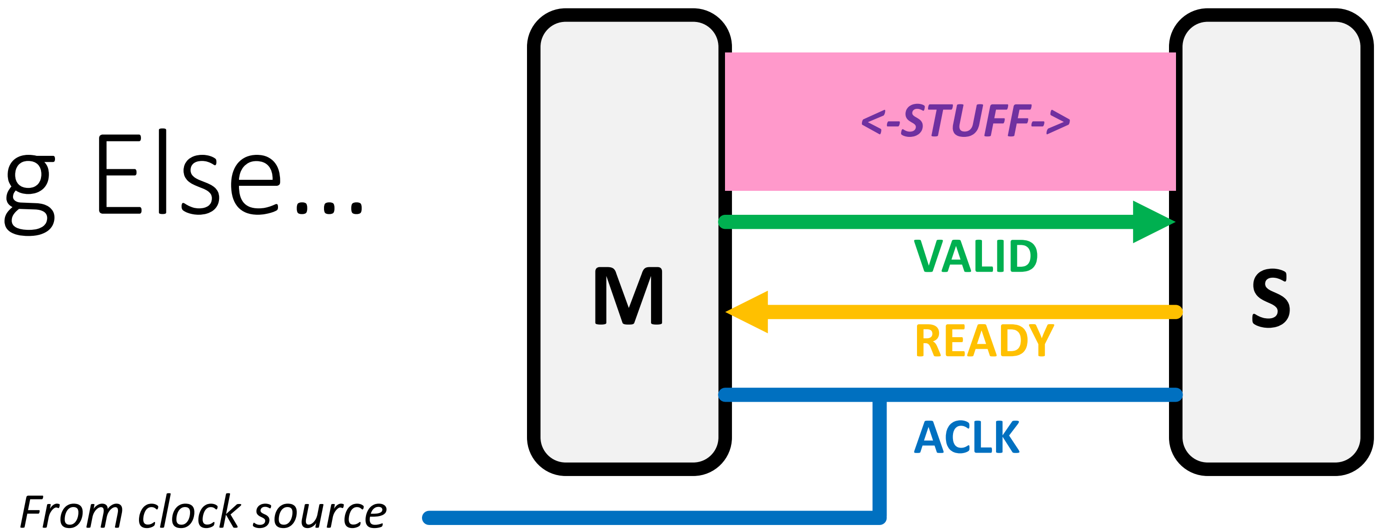
- Everything in system will run off of AXI clock usually called **ACLK** in documentation
- No combinatorial paths between inputs and outputs. Everything must be registered.
- All signals are sampled **on rising edge**
- AXI modules should also have Reset pins. AXI work ACTIVE LOW so the Reset pin is usually called **ARSTn** or **ARESETn** (meaning it is normally high)

Valid and Ready



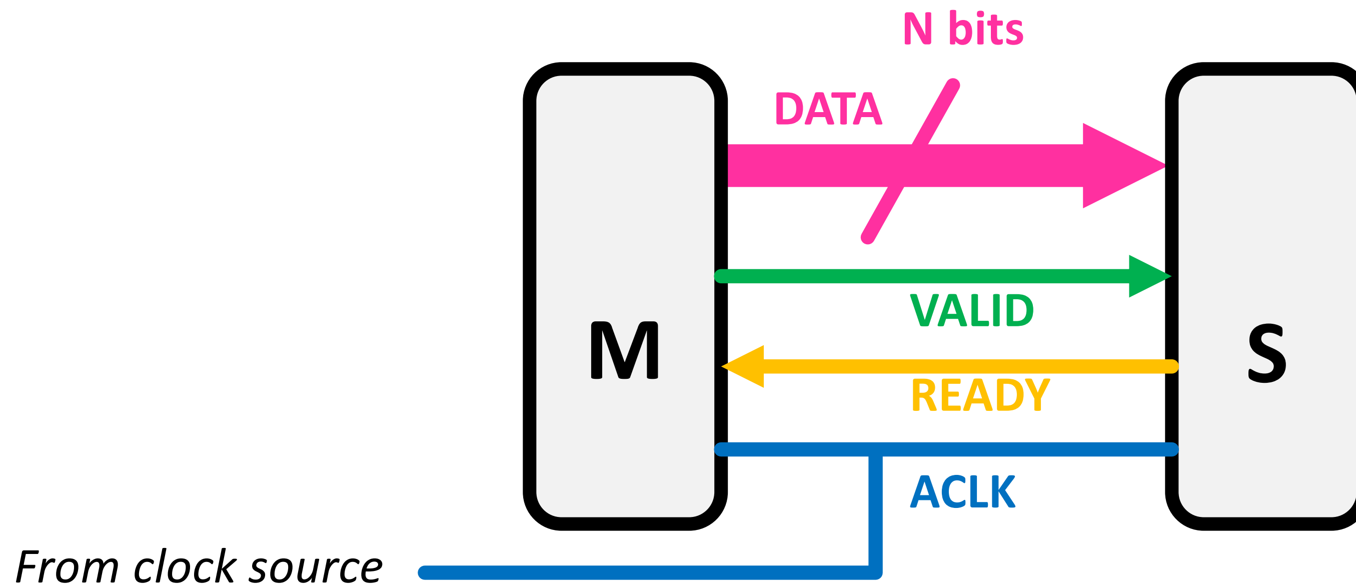
- All of AXI uses the same handshake procedure:
- The *source* of a data generates a **VALID** signal
- The *destination* of data generates a **READY** signal
- Transfer of data only occurs when both are high
- Both M and S Devices can therefore control the flow of their data as needed

Everything Else…



- Everything else is information and depends on what is needed in situation. Could be:
  - Address
  - Data
  - Metadata
  - Other specialized wires like:
    - STRB (used to specify which bytes in current data step are valid, sent by Main along with data payload to Secondary)
    - RESP (sort of like a status)
    - LAST (sent to indicate the final data clock cycle of data in a burst)

In a Pipelined Streaming Situation



Generalized Transaction

- All Channel Interactions follow same high-level structure

Table A3-1 Transaction channel handshake pairs

Transaction channel	Handshake pair
Write address channel	AWVALID, AWREADY
Write data channel	WVALID, WREADY
Write response channel	BVALID, BREADY
Read address channel	ARVALID, ARREADY
Read data channel	RVALID, RREADY

Sending One ”beat” of data (one clock-cycle of data)

Keep in mind this could be 64 parallel wires of 1’s and 0’s of info or 8 bytes for example… Or it could be something else

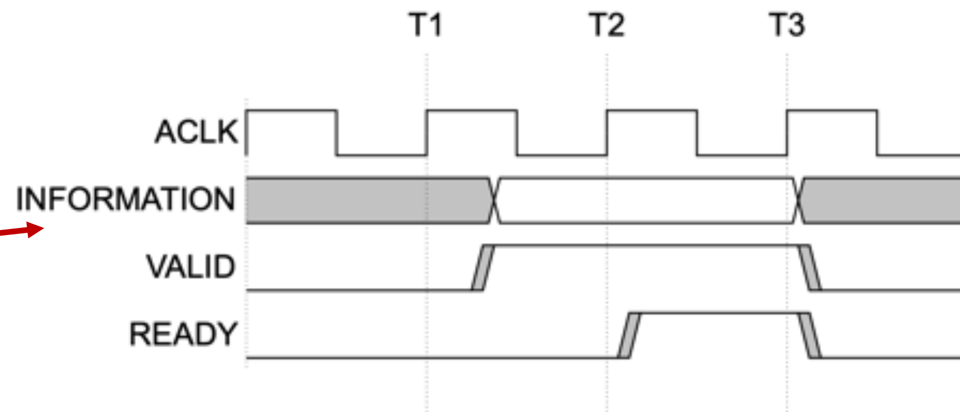


Figure A3-2 VALID before READY handshake

Generalized Transaction

- All Channel Interactions follow same high-level structure

Table A3-1 Transaction channel handshake pairs

Transaction channel	Handshake pair
Write address channel	AWVALID, AWREADY
Write data channel	WVALID, WREADY
Write response channel	BVALID, BREADY
Read address channel	ARVALID, ARREADY
Read data channel	RVALID, RREADY

Sending One "beat" of data (one clock-cycle of data)

Keep in mind this could be 64 parallel wires of 1's and 0's of info or 8 bytes for example… Or it could be something else

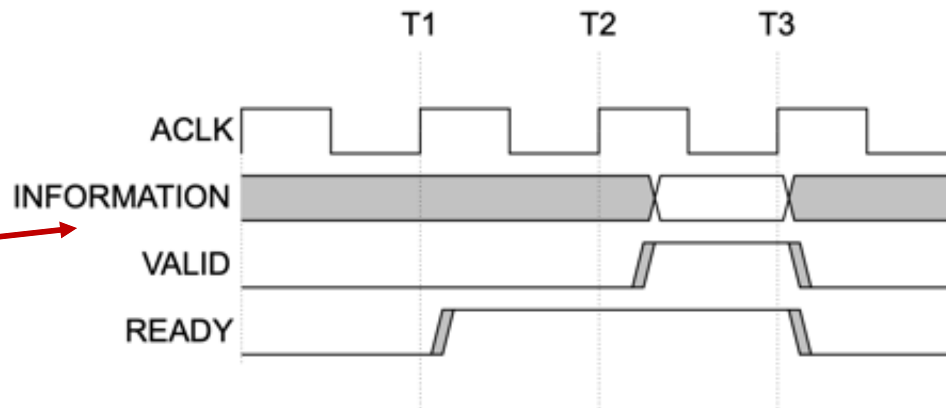


Figure A3-3 READY before VALID handshake

Generalized Transaction

- All Channel Interactions follow same high-level structure

Table A3-1 Transaction channel handshake pairs

Transaction channel	Handshake pair
Write address channel	AWVALID, AWREADY
Write data channel	WVALID, WREADY
Write response channel	BVALID, BREADY
Read address channel	ARVALID, ARREADY
Read data channel	RVALID, RREADY

Sending One ”beat” of data (one clock-cycle of data)

Keep in mind this could be 64 parallel wires of 1’s and 0’s of info or 8 bytes for example… Or it could be something else

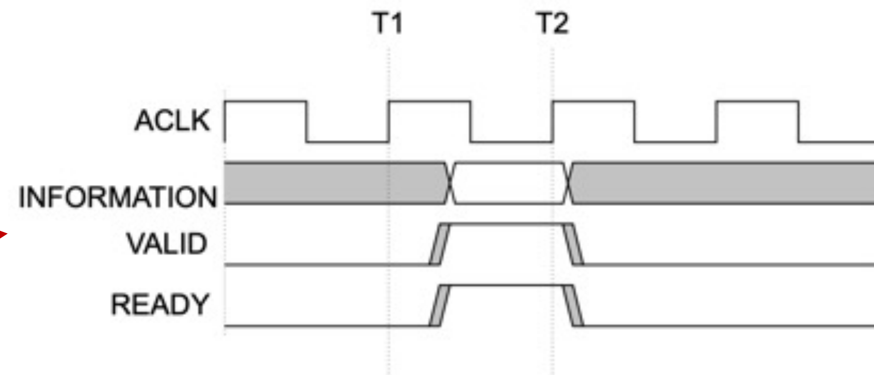


Figure A3-4 VALID with READY handshake

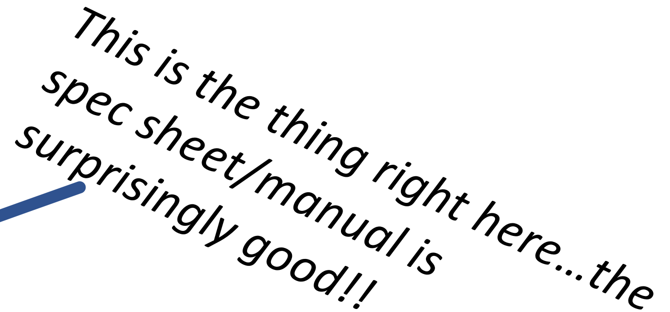
Other Things to Keep in Mind

- the **VALID** signal of the AXI interface sending information must not be dependent on the **READY** signal of the AXI interface receiving that information
- an AXI interface that is receiving information can wait until it detects a **VALID** signal before it asserts its corresponding **READY** signal.
- Fail to Follow these rules and could have devices wait infinitely.
  - Like when two people keep going “no, after you at a door”

Others than AXI?

- There are other generalized bus protocols out there:
  - Wishbone, some Open cores use this
  - Avalon: used in some Altera sets (proprietary)
- AXI is a good one to be familiar with, not just because it is used in Xilinx stuff a lot

Sources

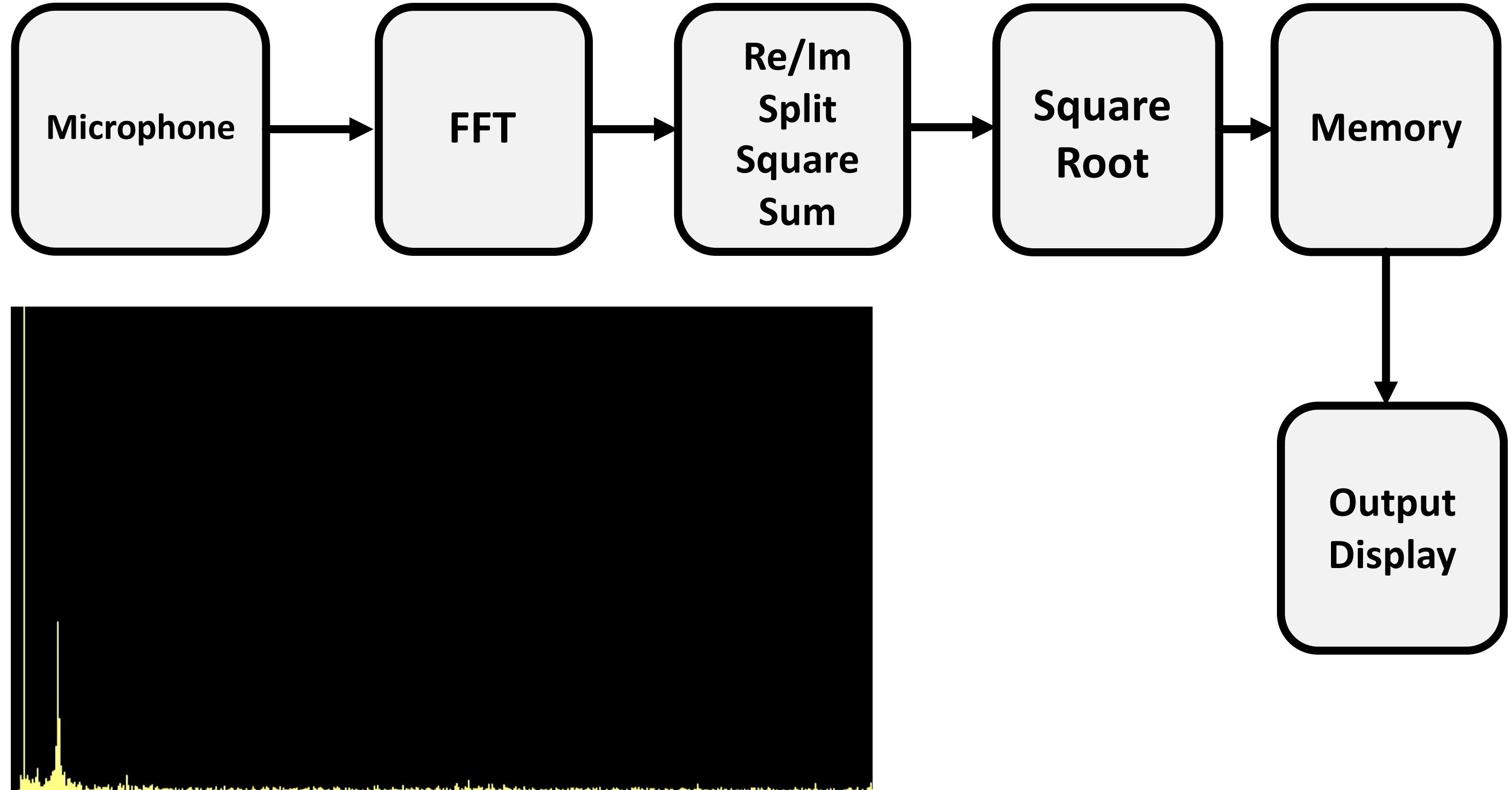


- **"AMBA® AXITM and ACETM Protocol Specification",** ARM 2011
- **"The Zynq Book", L.H. Crockett, R.A. Elliot, M.A. Enderwitz, and R.W. Stewart, University of Glasgow**
- **"Building Zynq Accelerators with Vivado High Level Synthesis" Xilinx Technical Note**
- **Some material from ECE699 Spring 2016 https://ece.gmu.edu/coursewebpages/ECE/ECE699_SW_HW/S16/**

Crack open the AXI spec sheet with a few data sheets for some Xilinx IP cores (like the CORDIC, FFT, etc…) and you should be able to start making sense of it.

Real-time Audio Spectrograph

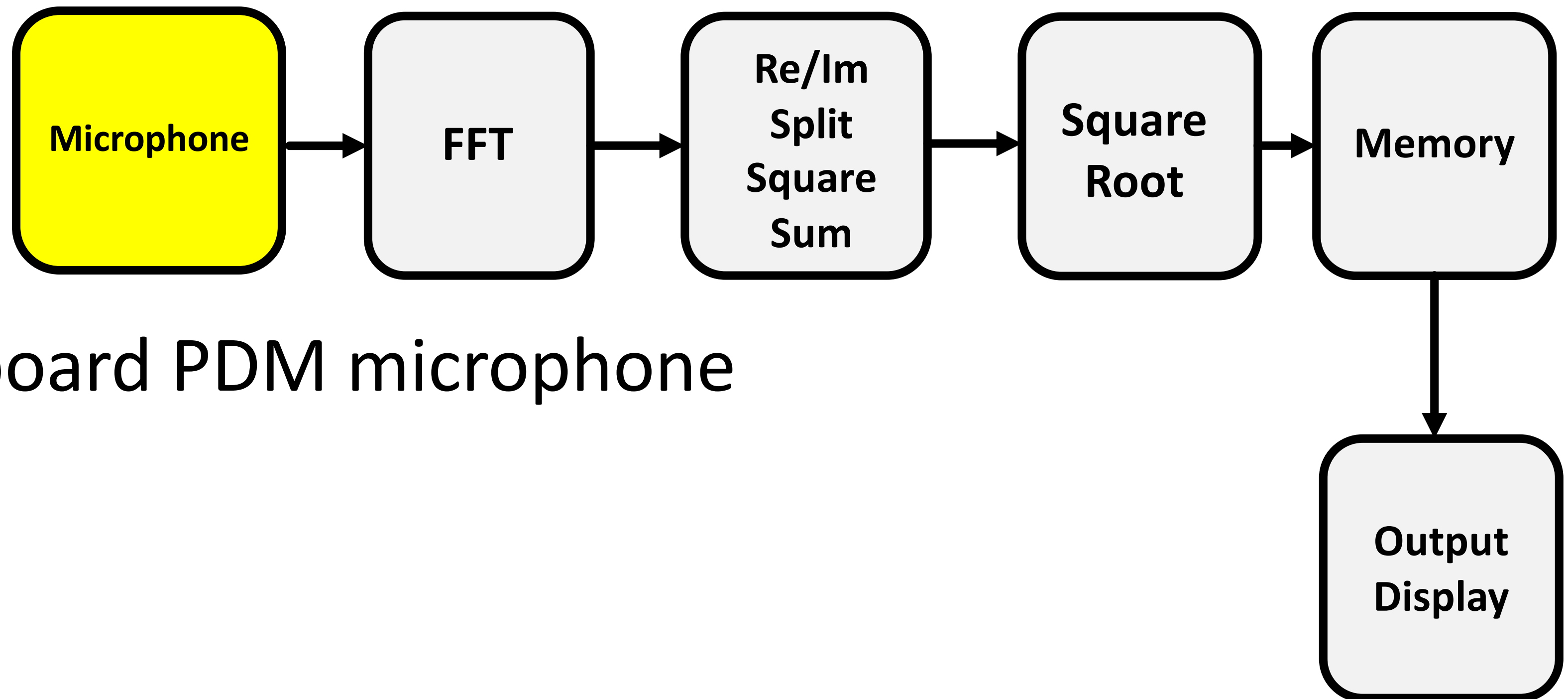
- Let’s do an example!!!



Real-time Audio Spectrograph

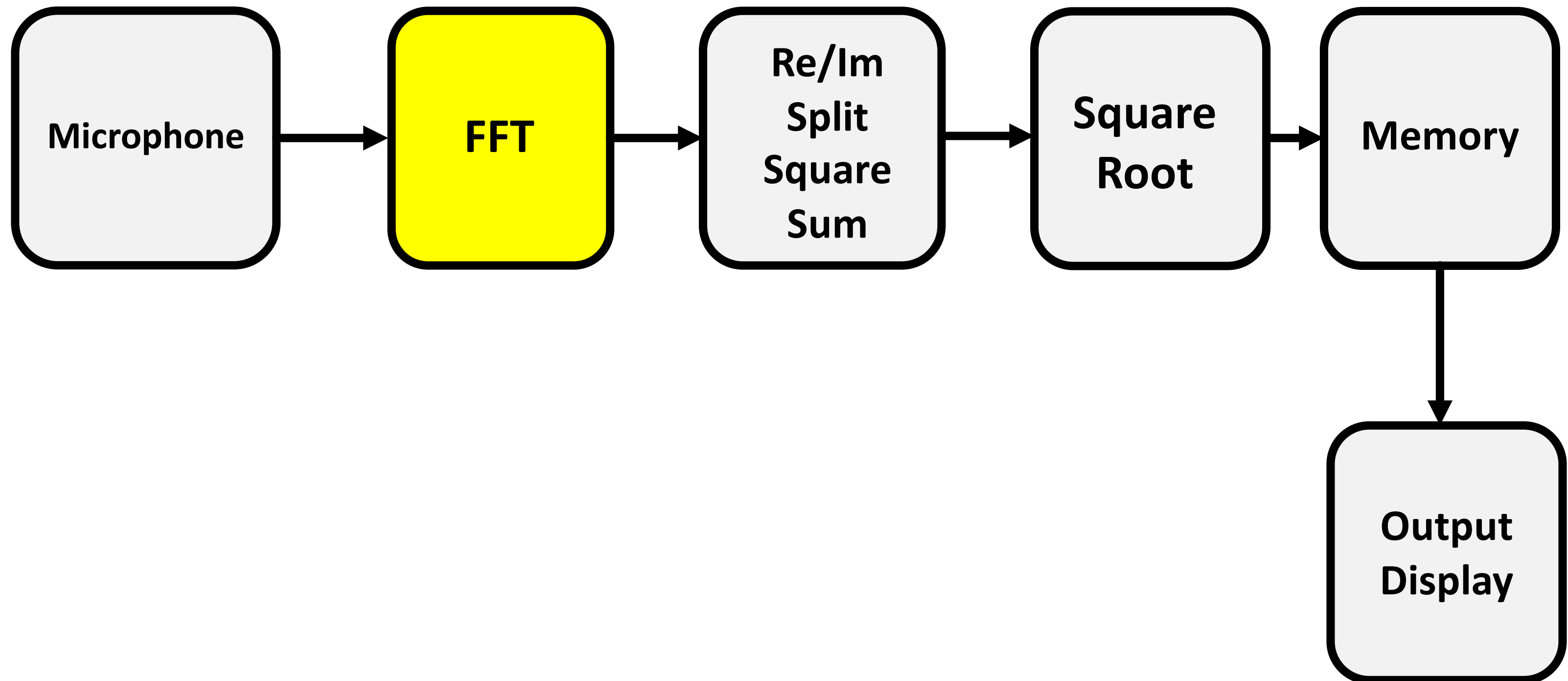
- Collect audio from microphone (use Analog-to-digital Converter)
- Convert time-series data to frequency series
- Take Magnitude of it
- Store it in memory
- Render it on screen as a bargraph
- **RESULT:**
  - Render the energy of the frequency spectrum in real time

Real-time Audio Spectrograph



- On-board PDM microphone

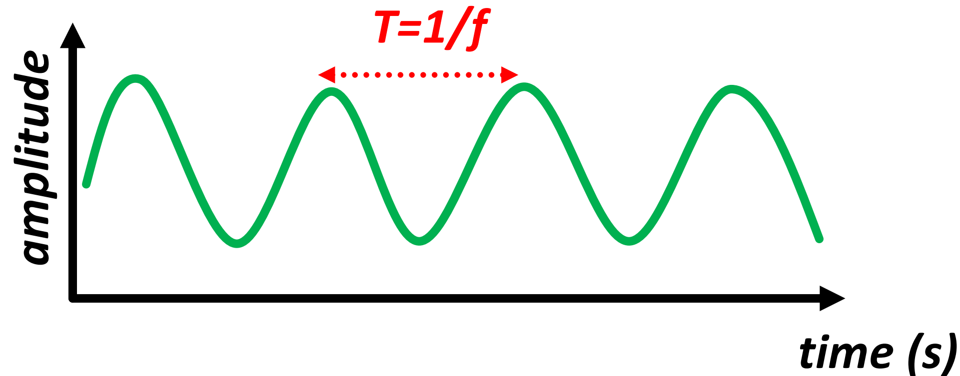
Real-time Audio Spectrograph



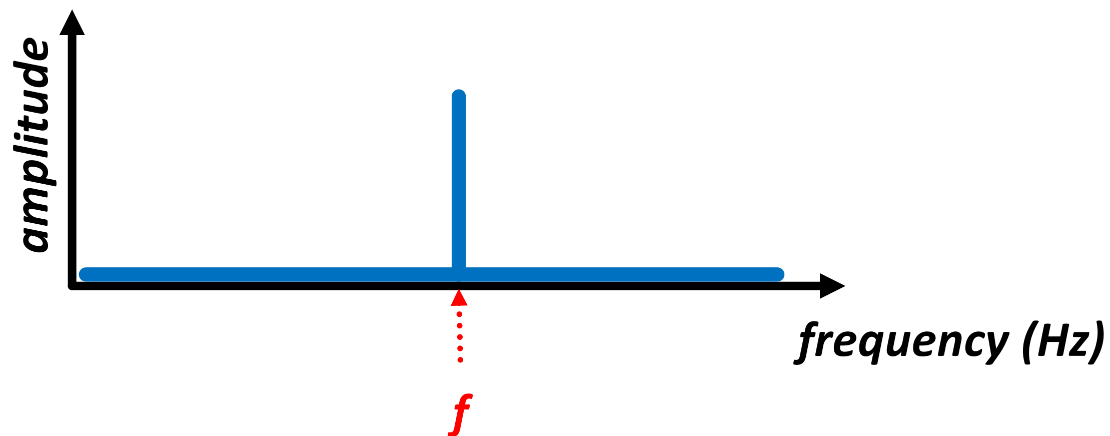
- Computer the Fourier Transform of a Time Series of audio measurements

Fourier Transform

- Convert a time-domain signal:



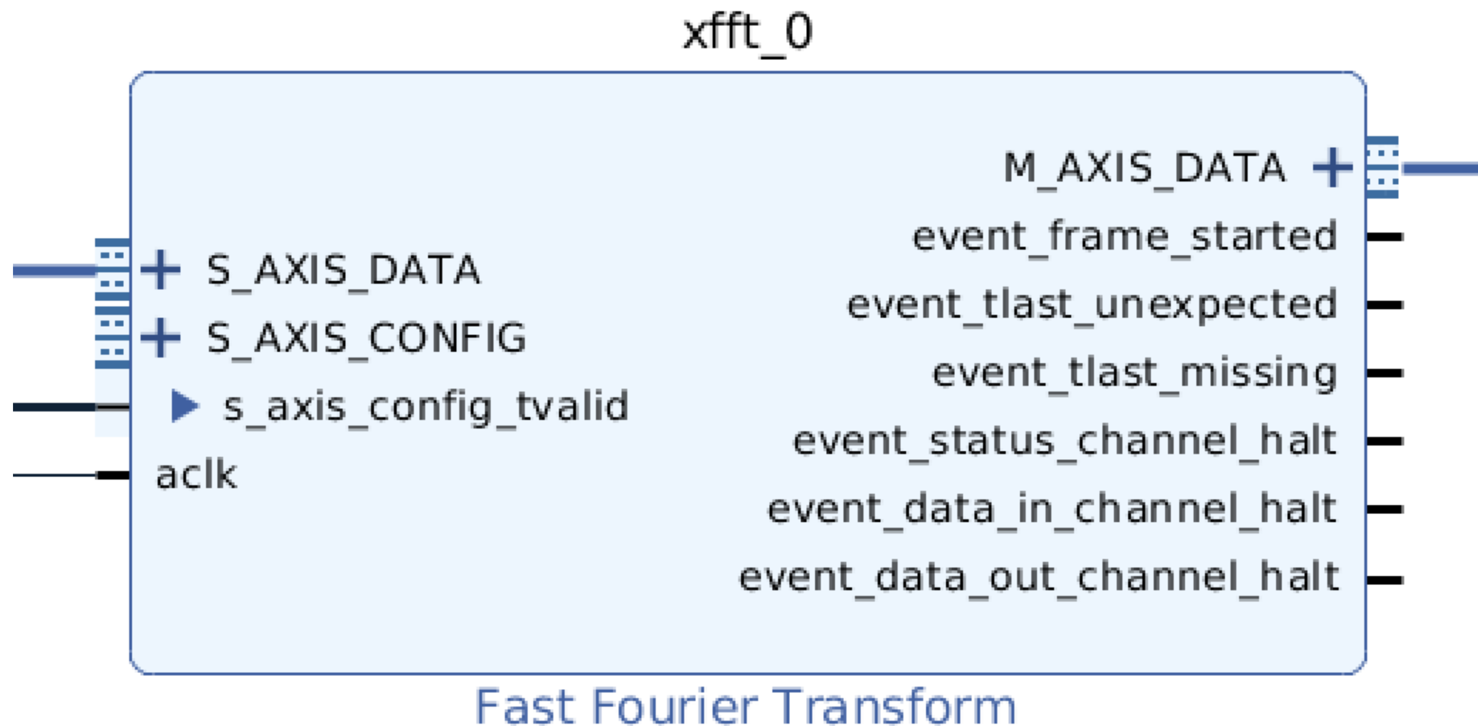
- Into its frequency domain representation:



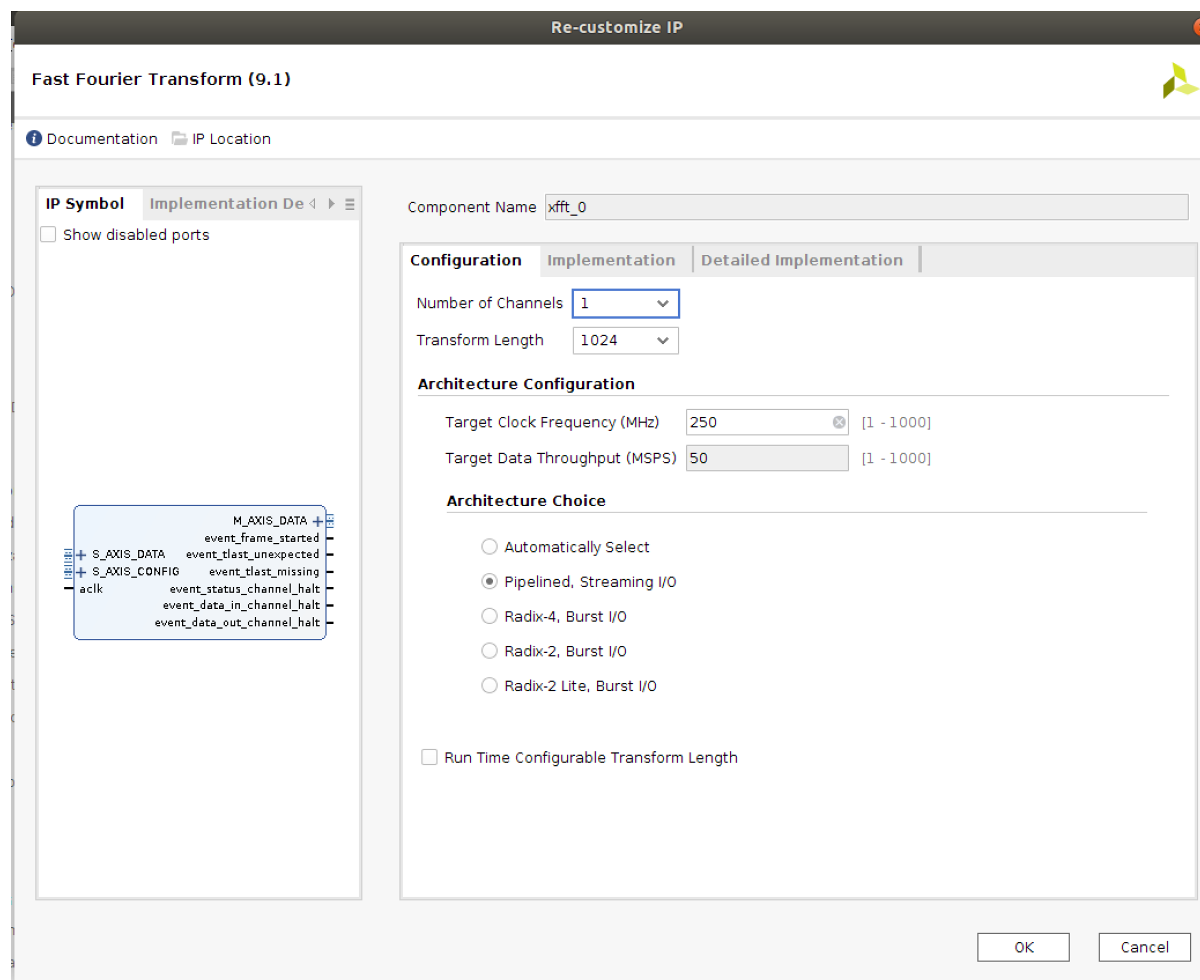
Fast Fourier Transform

- A computationally efficient means of generating the Fourier Transform
- We'll do a 1024 point Fourier Transform (pretty small)
- The bigger the *N*, the "better" the Fourier transform, but the number of multiply adds you need to will scale with $\boldsymbol{N^2}$…this becomes problematic very quickly
- A Fast Fourier Transform is a class of algorithm that takes advantage of symmetries/periodicities in all of the multiplications that you do in order to simplify the overall work.
- These simplifications allow the work to scale with $\boldsymbol{N\log(N)}$
- Further pipelining and parallel structures in hardware allow you to stream into an FFT. Lots of repetition in FFT…great for pipelining vs. Blocking FSM debate/choice

Fast Fourier Transform



FFT



All the way up to 65536 point FFT (theoretically)…never built one myself, but it should be possible

FFT

Configuration | Implementation | Detailed Ir

Number of Channels: 1

Transform Length: 1024

Architecture Configuration

Target Clock Frequency (MHz): 250

Target Data Throughput (MSPS): 50

Architecture Choice

- Automatically Select
- Pipelined, Streaming I/O (selected)
- Radix-4, Burst I/O
- Radix-2, Burst I/O
- Radix-2 Lite, Burst I/O

- **Pipelined, Streaming I/O** – Allows continuous data processing.
- **Radix-4, Burst I/O** – Loads and processes data separately, using an iterative approach. It is smaller in size than the pipelined solution, but has a longer transform time.
- **Radix-2, Burst I/O** – Uses the same iterative approach as Radix-4, but the butterfly is smaller. This means it is smaller in size than the Radix-4 solution, but the transform time is longer.
- **Radix-2 Lite, Burst I/O** – Based on the Radix-2 architecture, this variant uses a time-multiplexed approach to the butterfly for an even smaller core, at the cost of longer transform time.

Figure 2 illustrates the trade-off of throughput versus resource use for the four architectures. As a rule of thumb, each architecture offers a factor of 2 difference in resource from the next architecture. The example is for an even power of 2 point size. This does not require the Radix-4 architecture to have an additional Radix-2 stage.

All four architectures may be configured to use a fixed-point interface with one of three fixed-point arithmetic methods (unscaled, scaled or block floating-point) or may instead use a floating-point interface.

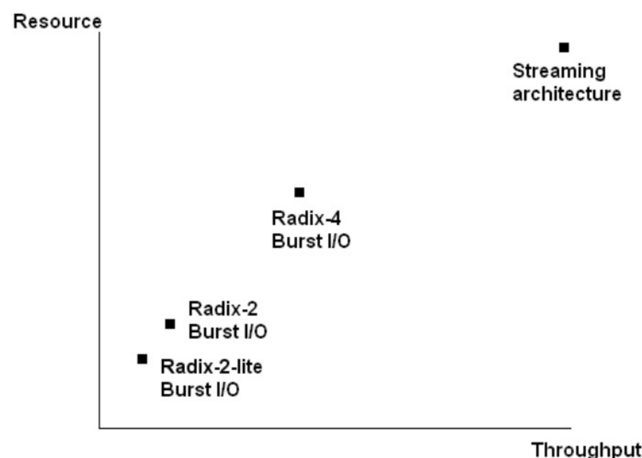


Figure 2: **Resource versus Throughput for Architecture Options**

https://www.xilinx.com/support/documentation/ip_documentation/xfft_ds260.pdf

FFT Latency

IP Symbol | Implementation Details | Latency

Transform Length	Transform Cycles	Latency(μs)
1024	3191	31.910

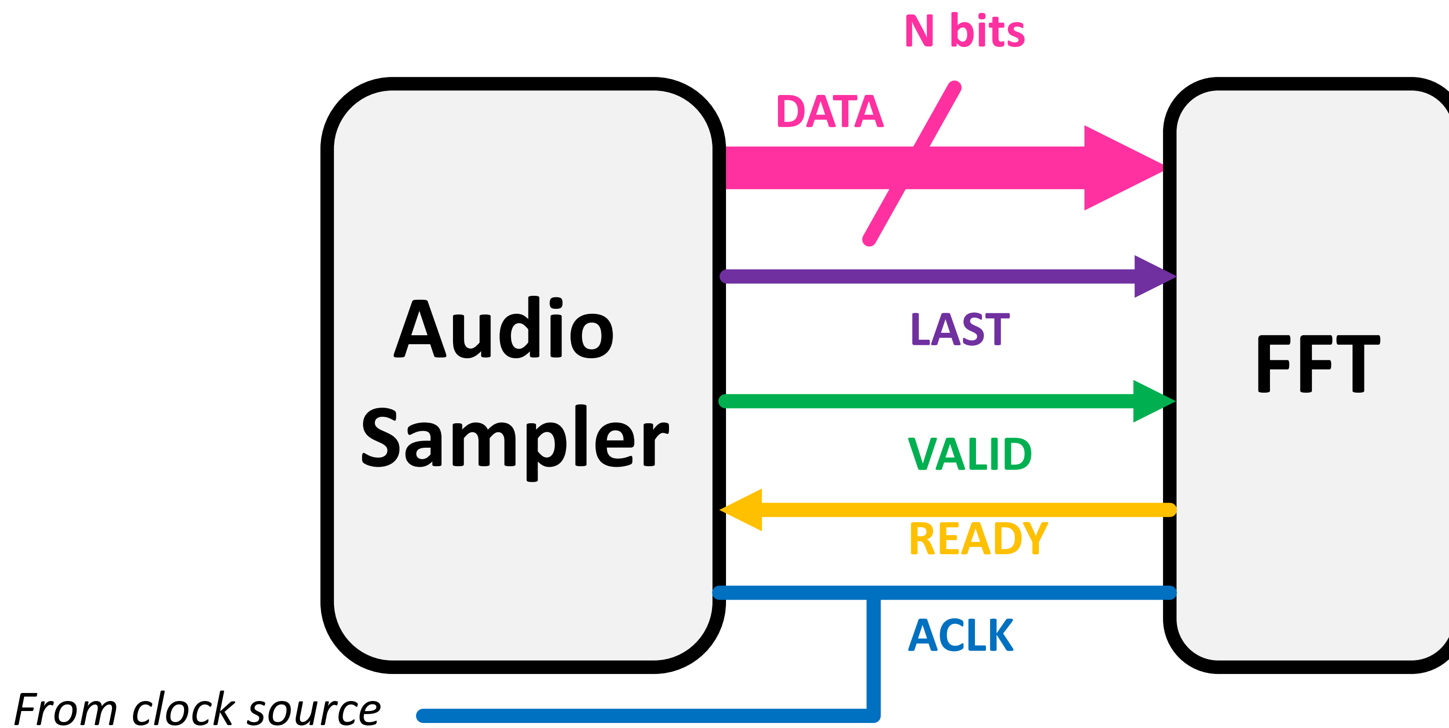
- Calculates entire FFT in 31.910 μs !
- Needs all 1024 input samples before it starts outputting

Do we need ready?

- Sometimes you can ignore it maybe…
- If data is coming in no matter what

TLAST

- Since we're sending 1024 samples one after the other (serially) we need a way to tell the FFT we're at the end of a frame!
- Use a LAST signal (tells FFT we're on last sample)



FFT Input

If audio sample ready,
give it a sample,
Otherwise don't

```
always_ff @(posedge axi_clk)begin
  if (audio_sample_valid)begin
    fft_valid = 1;
    fft_data = {audio_data,8'b0};
    fft_data_counter <= fft_data_counter +1;
    fft_last <= fft_data_counter==1023;
  end else begin
    fft_valid = 0;
  end
end
```

FFT Instance:

```
1  xfft_0 my_fft (.aclk(clk_100mhz), .s_axis_data_tdata(fft_data),
2                 .s_axis_data_tvalid(fft_valid),
3                 .s_axis_data_tlast(fft_last), .s_axis_data_tready(fft_ready),
4                 .s_axis_config_tdata(0),
5                  .s_axis_config_tvalid(0),
6                  .s_axis_config_tready(),
7                 .m_axis_data_tdata(fft_out_data), .m_axis_data_tvalid(fft_out_valid),
8                 .m_axis_data_tlast(fft_out_last), .m_axis_data_tready(fft_out_ready));
```

Already “breaking” AXI

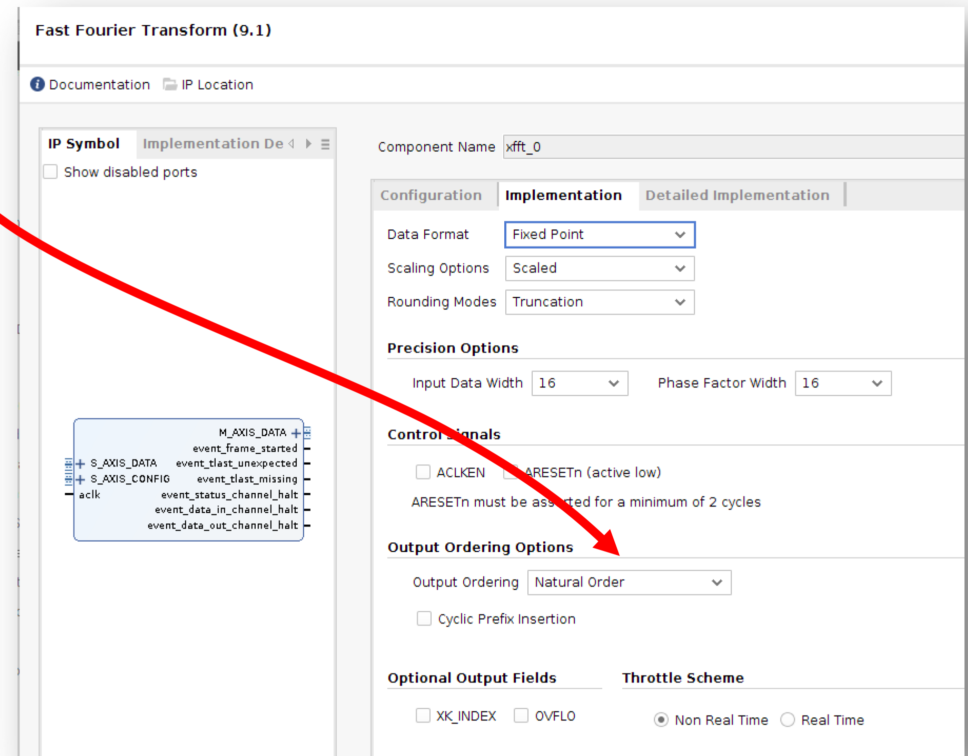
- This code is not monitoring whether the FFT is **READY** for data.
- Realistically we are generating data so slowly that this will never actually matter (discuss at end)

```
always_ff @(posedge axi_clk)begin
  if (audio_sample_valid)begin
    fft_valid = 1;
    fft_data = {audio_data,8'b0};
    fft_data_counter <= fft_data_counter +1;
    fft_last <= fft_data_counter==1023;
  end else begin
    fft_valid = 0;
  end
end
```

FFT

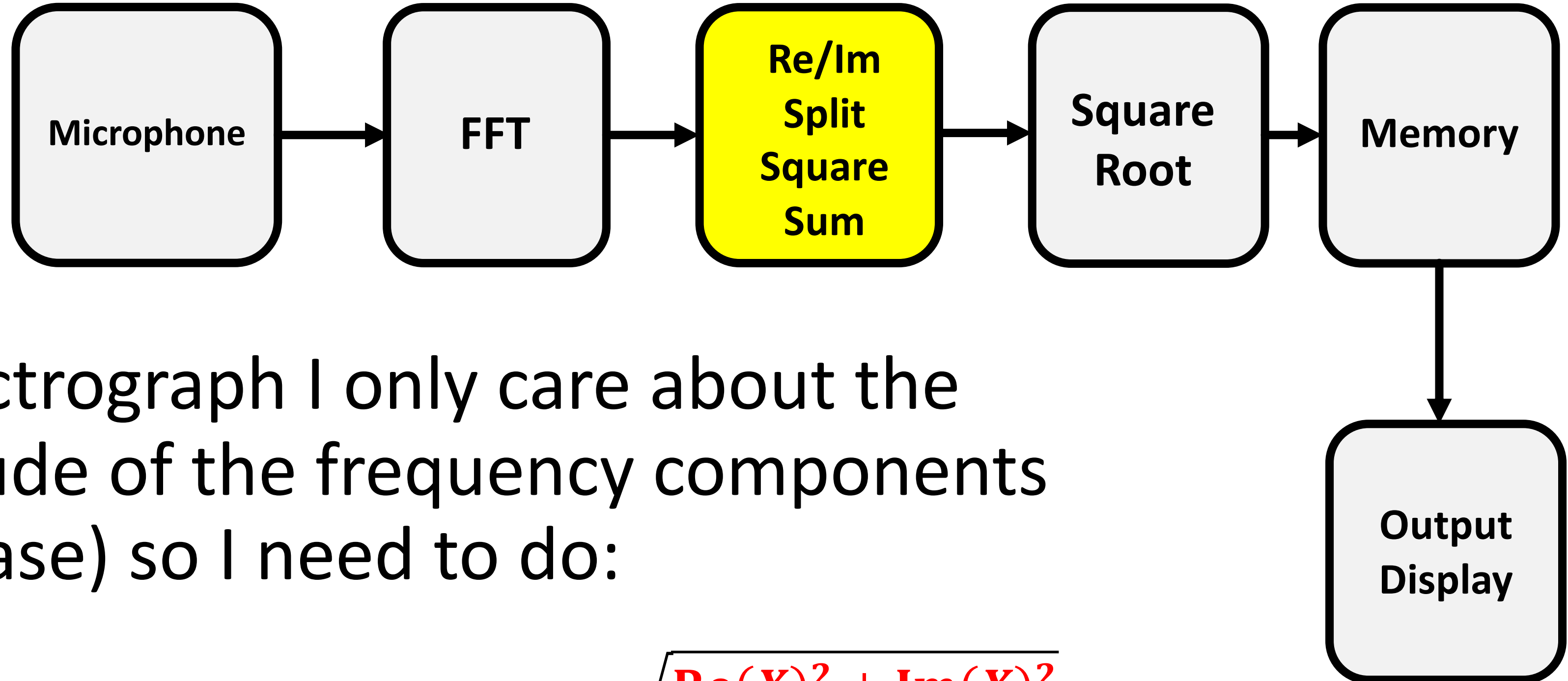
- Because of how an FFT is calculated the first known values are not the lowest frequency values
- I blow an extra 1200 cycles to have FFT organize its outputs in order of frequency (“Natural Order”)

- Having individual labels for each data sample could let me do this.



Real-time Audio Spectrograph

- FFT outputs 32 bits of a complex number:
  - 16 bits real component
  - 16 bits imaginary component

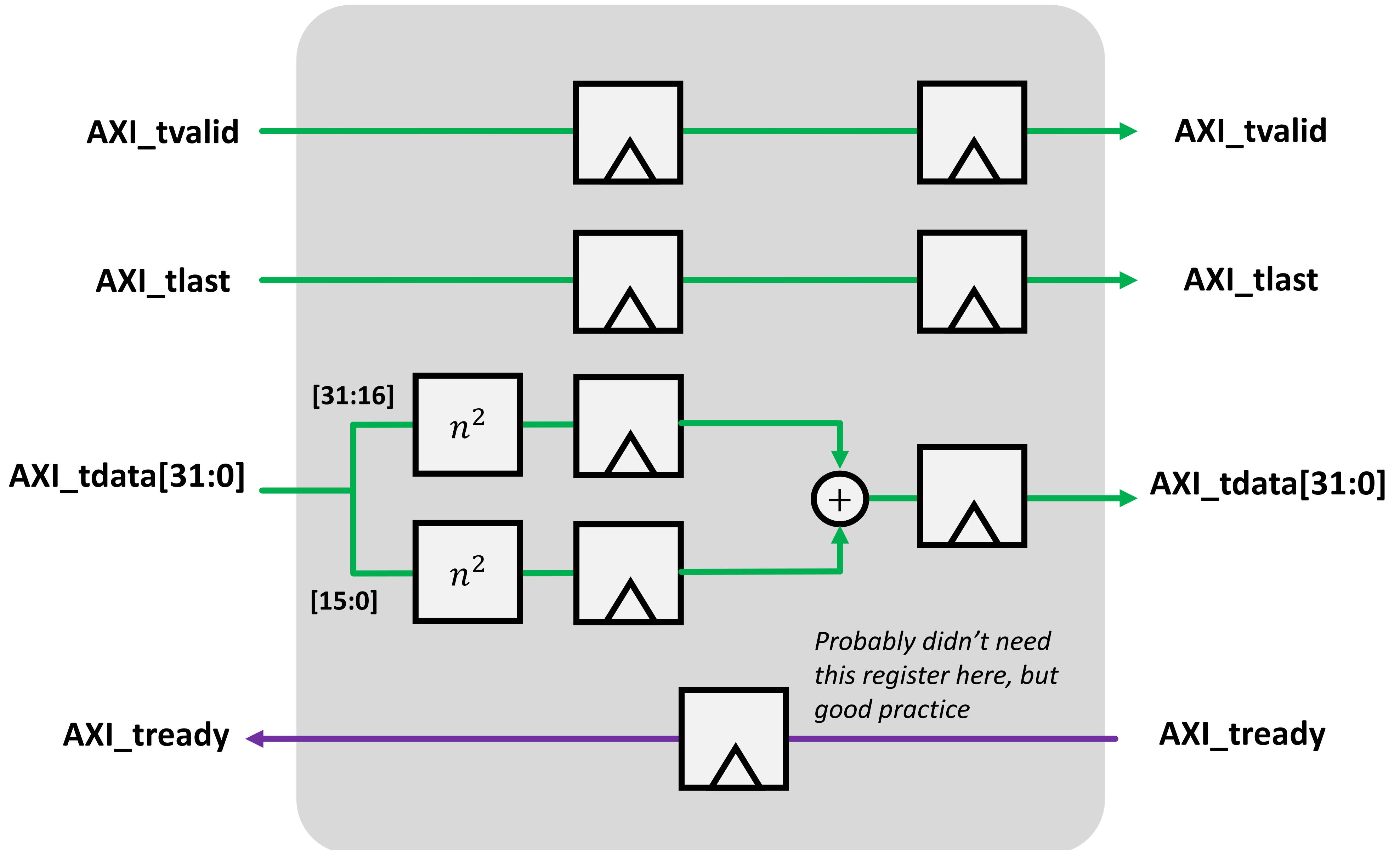


For spectrograph I only care about the magnitude of the frequency components (not phase) so I need to do:

$$\sqrt{\mathbf{Re}(X)^2 + \mathbf{Im}(X)^2}$$

Split→Square→Sum

M → S
M ← S



Split→Square→Sum

```
51      reg s00_axis_tready_reg;
52      reg signed [31:0] real_square;
53      reg signed [31:0] imag_square;
54
55      wire signed [15:0] real_in;
56      wire signed [15:0] imag_in;
57      assign real_in = s00_axis_tdata[31:16];
58      assign imag_in = s00_axis_tdata[15:0];
59
60      assign m00_axis_tvalid = m00_axis_tvalid_reg;
61      assign m00_axis_tlast = m00_axis_tlast_reg;
62      assign m00_axis_tdata = m00_axis_tdata_reg;
63      assign s00_axis_tready = s00_axis_tready_reg;
64
65      always @(posedge s00_axis_aclk)begin
66          if (s00_axis_aresetn==0)begin
67              s00_axis_tready_reg <= 0;
68          end else begin
69              s00_axis_tready_reg <= m00_axis_tready; //if what you're feeding data to is ready, then you're ready.
70          end
71      end
72
73      always @(posedge m00_axis_aclk)begin
74          if (m00_axis_aresetn==0)begin
75              m00_axis_tvalid_reg <= 0;
76              m00_axis_tlast_reg <= 0;
77              m00_axis_tdata_reg <= 0;
78          end else begin
79              m00_axis_tvalid_reg_pre <= s00_axis_tvalid; //when new data is coming, you've got new data to put out
80              m00_axis_tlast_reg_pre <= s00_axis_tlast; //
81              real_square <= real_in*real_in;
82              imag_square <= imag_in*imag_in;
83
84              m00_axis_tvalid_reg <= m00_axis_tvalid_reg_pre; //when new data is coming, you've got new data to put out
85              m00_axis_tlast_reg <= m00_axis_tlast_reg_pre; //
86              m00_axis_tdata_reg <= real_square + imag_square;
87          end
88      end
89
```

Split the real imaginary parts

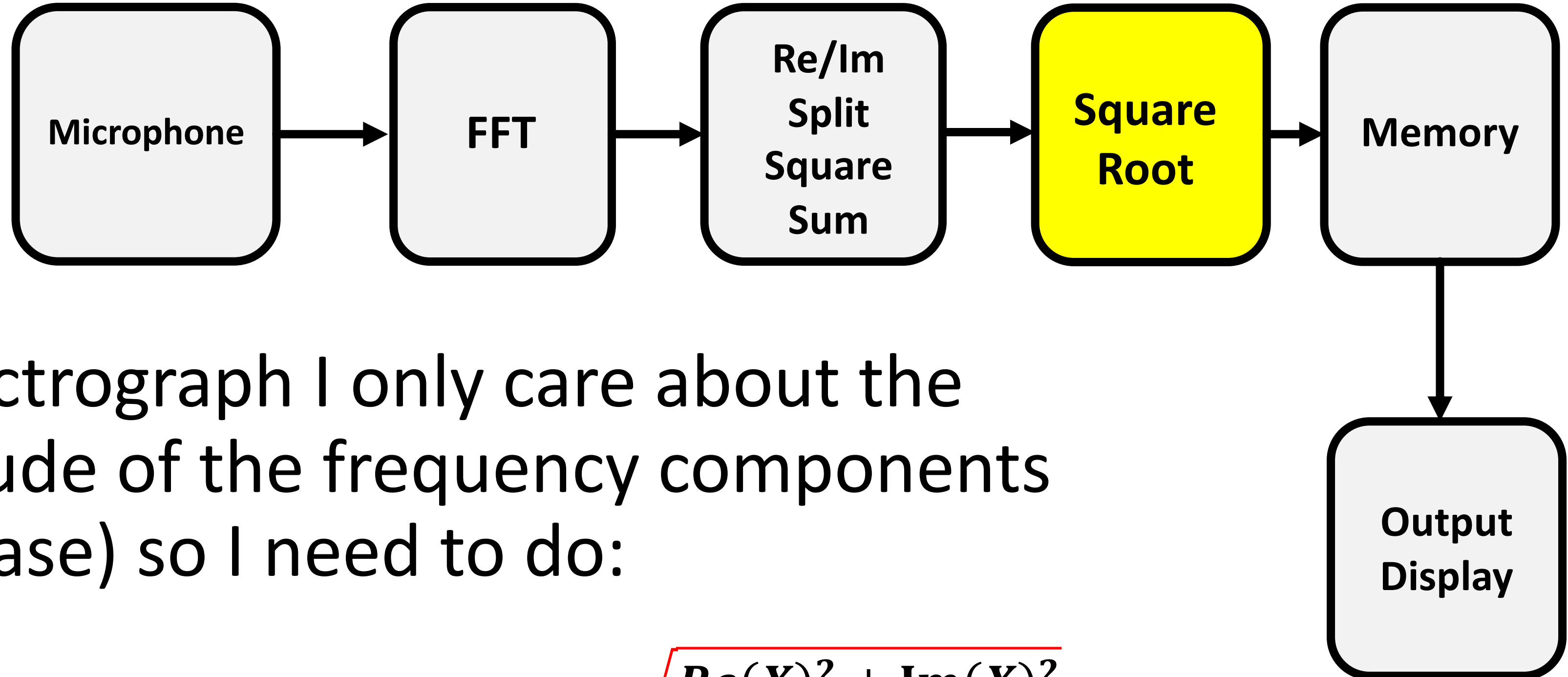
Two-Cycle Latency Pipeline

Square the real, imag parts on one cycle

Sum them on next cycle

Real-time Audio Spectrograph

- FFT outputs 32 bits of a complex number:
  - 16 bits real component
  - 16 bits imaginary component



For spectrograph I only care about the magnitude of the frequency components (not phase) so I need to do:

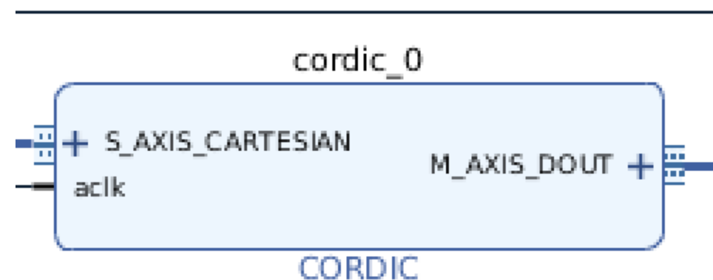
$$\sqrt{Re(X)^2 + \mathrm{Im}(X)^2}$$

CORDIC

- Generalized Mathematical operations (mostly trig and hyperbolics, but square roots too), done using only adds, subtracts, shifts, and some lookups

- Basically works by guessing and checking in iteratively smaller leaps to arrive at answer!

- Is really cool: https://en.wikipedia.org/wiki/CORDIC



CORDIC Configure…specify input/output size

CORDIC (6.0)

Documentation | IP Location

IP Symbol | Implementation Details

Show disabled ports

S_AXIS_CARTESIAN | M_AXIS_DOUT | aclk

Component Name: cordic_0

Configuration Options | AXI4 Stream Options

Configuration Parameters

Functional Selection	Square Root
Architectural Configuration	Parallel
Pipelining Mode	Maximum
Data Format	UnsignedInteger
Phase Format	Radians

Input/Output Options

Input Width	48	[8 - 48]
Output Width	25	[5 - 48]
Round Mode	Truncate	

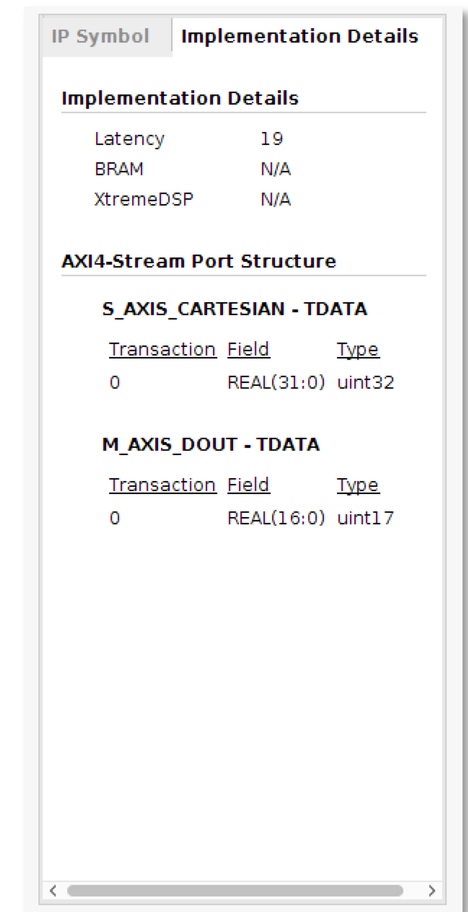
Advanced Configuration Parameters

Iterations	0	[0 - 48]
Precision	0	[0 - 48]

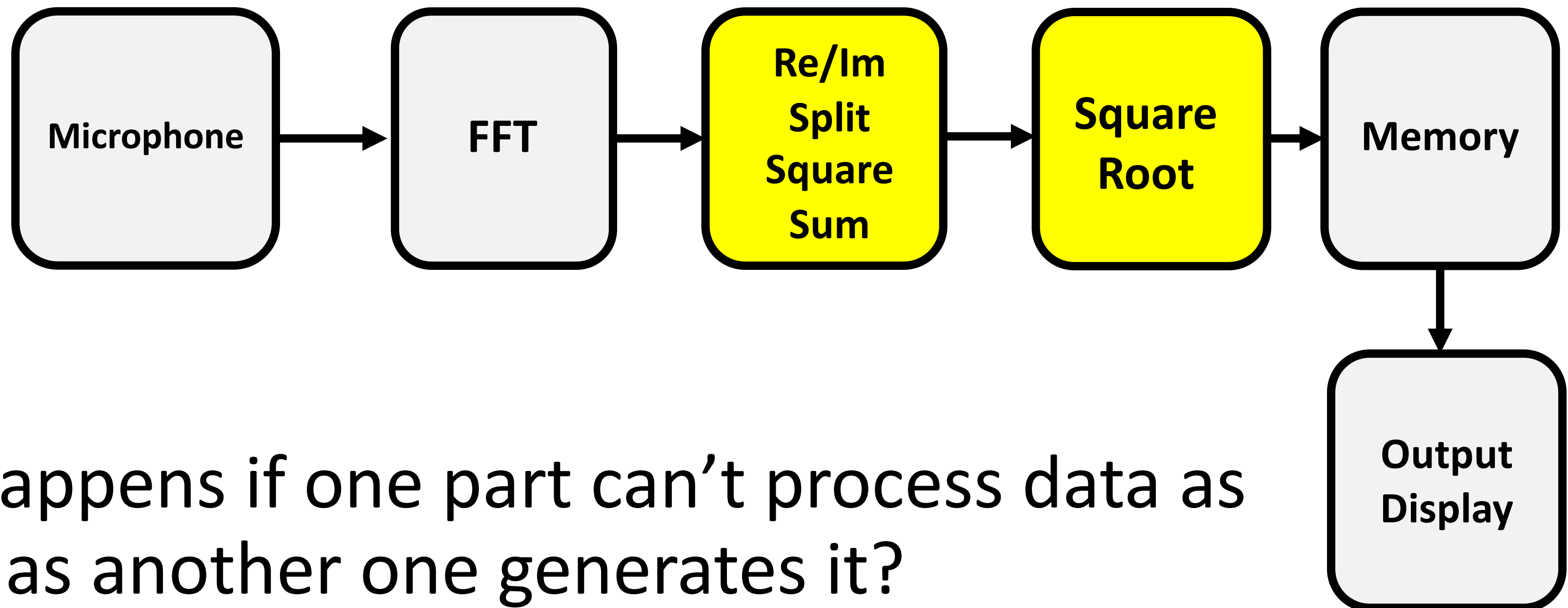
Coarse Rotation

Compensation Scaling: No Scale Compensation

OK | Cancel



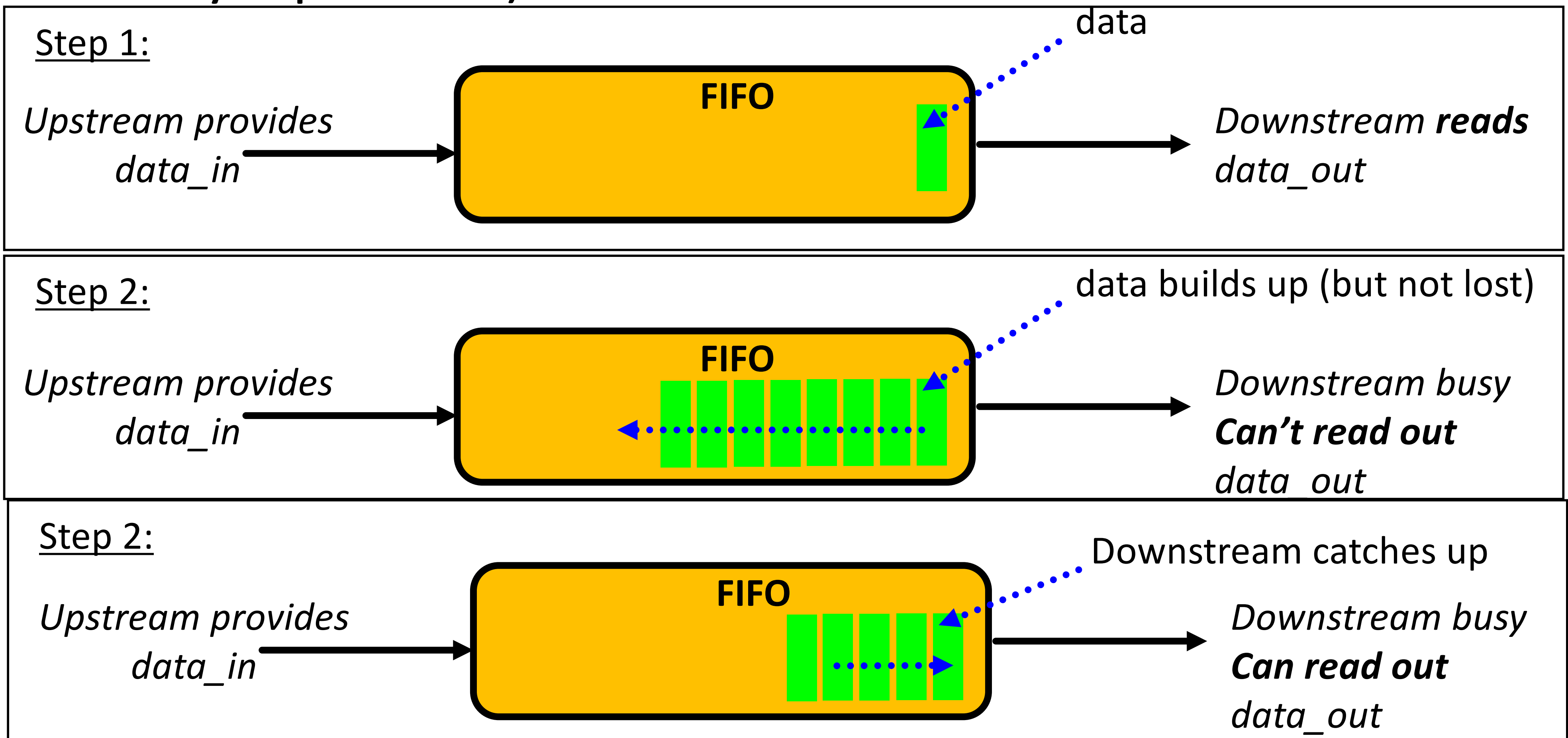
Real-time Audio Spectrograph



- What happens if one part can't process data as quickly as another one generates it?
- Hopefully the backpropagation of **t_READY** over an AXI bus should help with this, but might be good to add some breathing room

First-In-First-Out (FIFO)

- An ordered temporary holding tank of data
- Made of Two-port BRAM with a few pointers (like C-style pointers) variables

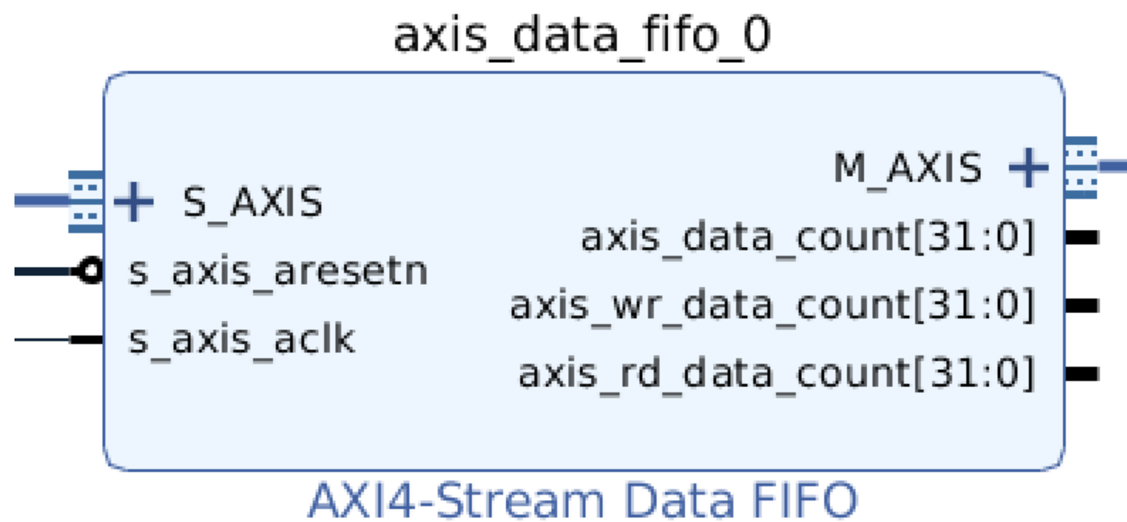


FIFOs

- If upstream produces measurements at 100 MHz and downstream processes at 50 MHz, FIFOs **will not help!**
- They only help to resolve momentary buildups of data!
- The FFT doesn't periodically generate output:
  - Much of runtime its output is silent and THEN it generates a burst of data

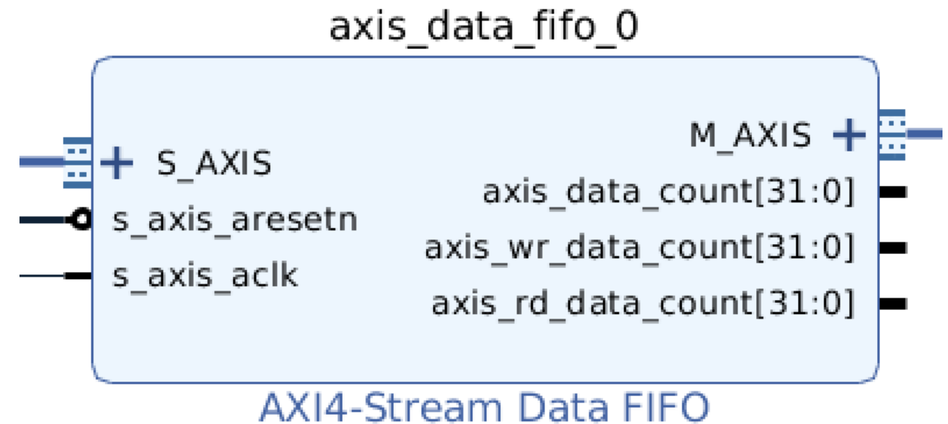
FFT Data Output

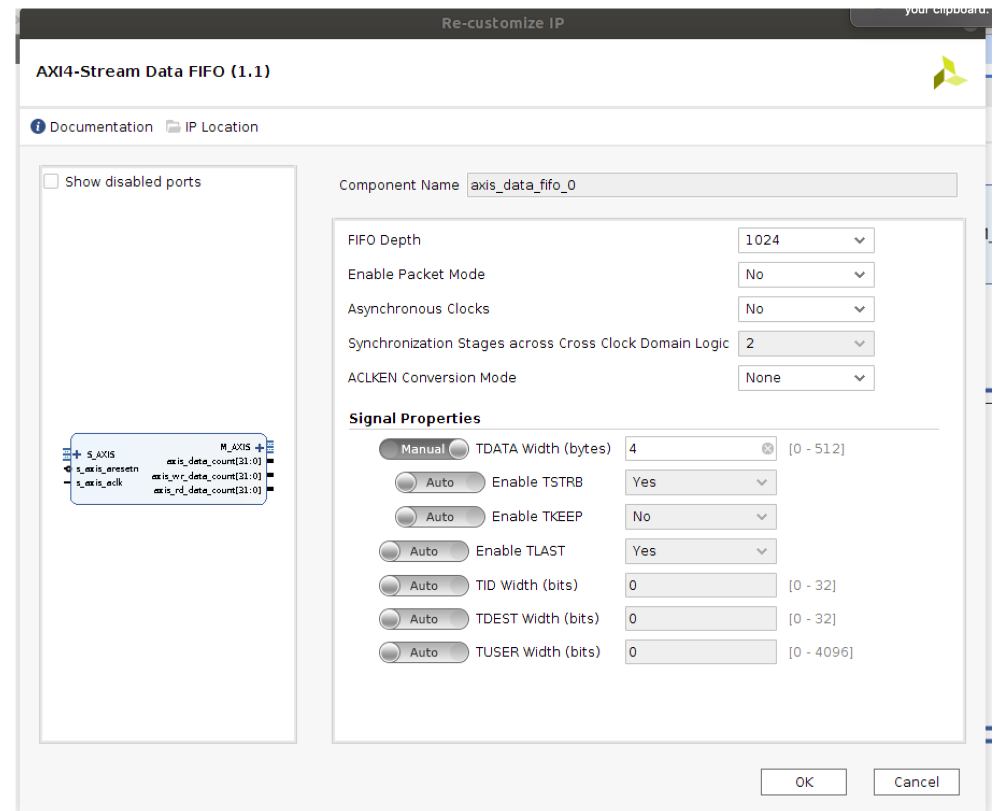
AXI4S FIFO



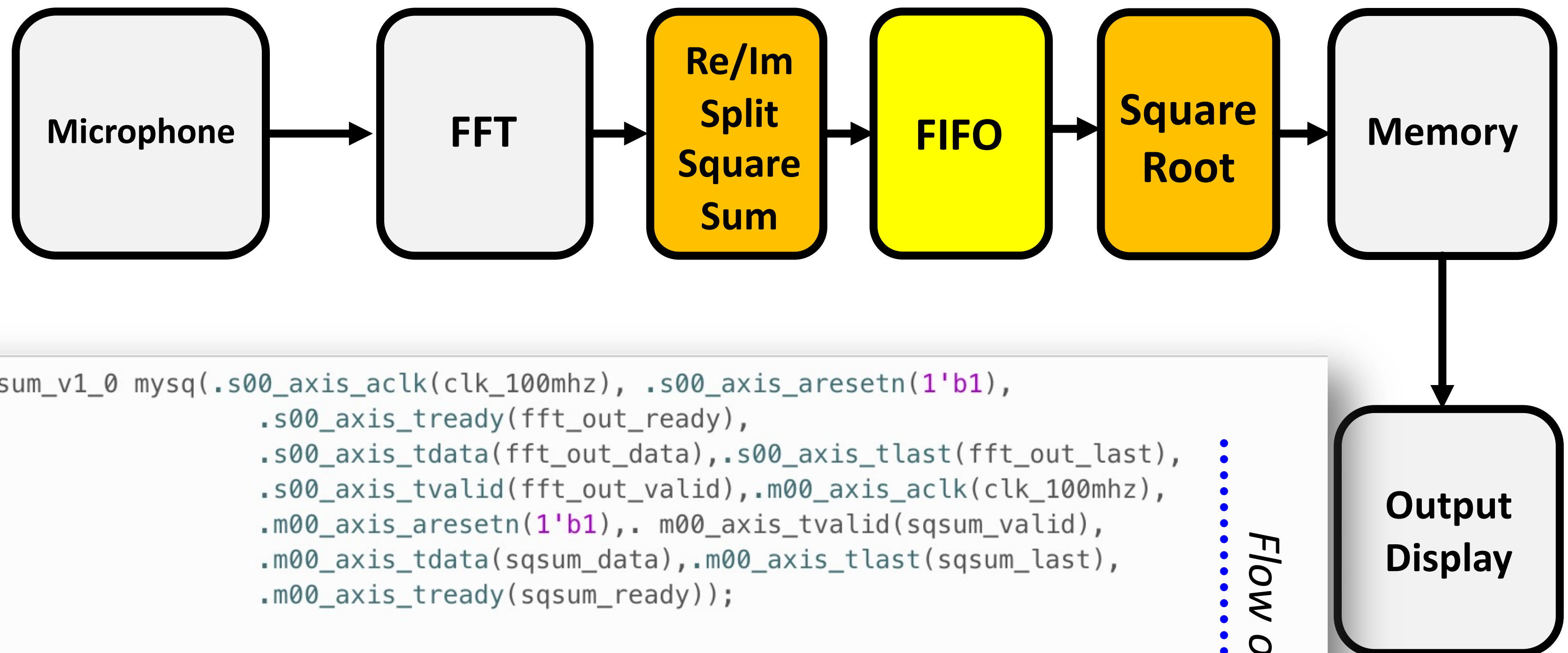
AXI4S FIFO

- Added in between because my original square version was blocking and not pipelined
- Switched to fully pipelined mode





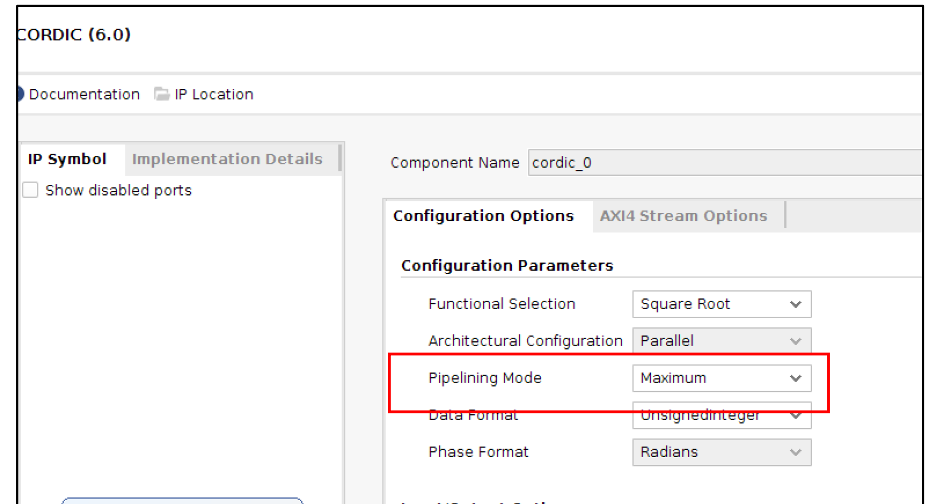
Real-time Audio Spectrograph




```
1  square_and_sum_v1_0 mysq(.s00_axis_aclk(clk_100mhz), .s00_axis_aresetn(1'b1),
2                             .s00_axis_tready(fft_out_ready),
3                             .s00_axis_tdata(fft_out_data),.s00_axis_tlast(fft_out_last),
4                             .s00_axis_tvalid(fft_out_valid),.m00_axis_aclk(clk_100mhz),
5                             .m00_axis_aresetn(1'b1),. m00_axis_tvalid(sqsum_valid),
6                             .m00_axis_tdata(sqsum_data),.m00_axis_tlast(sqsum_last),
7                             .m00_axis_tready(sqsum_ready));
8
9
10 axis_data_fifo_0 myfifo (.s_axis_aclk(clk_100mhz), .s_axis_aresetn(1'b1),
11                          .s_axis_tvalid(sqsum_valid), .s_axis_tready(sqsum_ready),
12                          .s_axis_tdata(sqsum_data), .s_axis_tlast(sqsum_last),
13                          .m_axis_tvalid(fifo_valid), .m_axis_tdata(fifo_data),
14                          .m_axis_tready(fifo_ready), .m_axis_tlast(fifo_last));
15
16 cordic_0 mysqrt (.aclk(clk_100mhz), .s_axis_cartesian_tdata(fifo_data),
17                  .s_axis_cartesian_tvalid(fifo_valid), .s_axis_cartesian_tlast(fifo_last),
18                  .s_axis_cartesian_tready(fifo_ready),.m_axis_dout_tdata(sqrt_data),
19                  .m_axis_dout_tvalid(sqrt_valid), .m_axis_dout_tlast(sqrt_last));
```

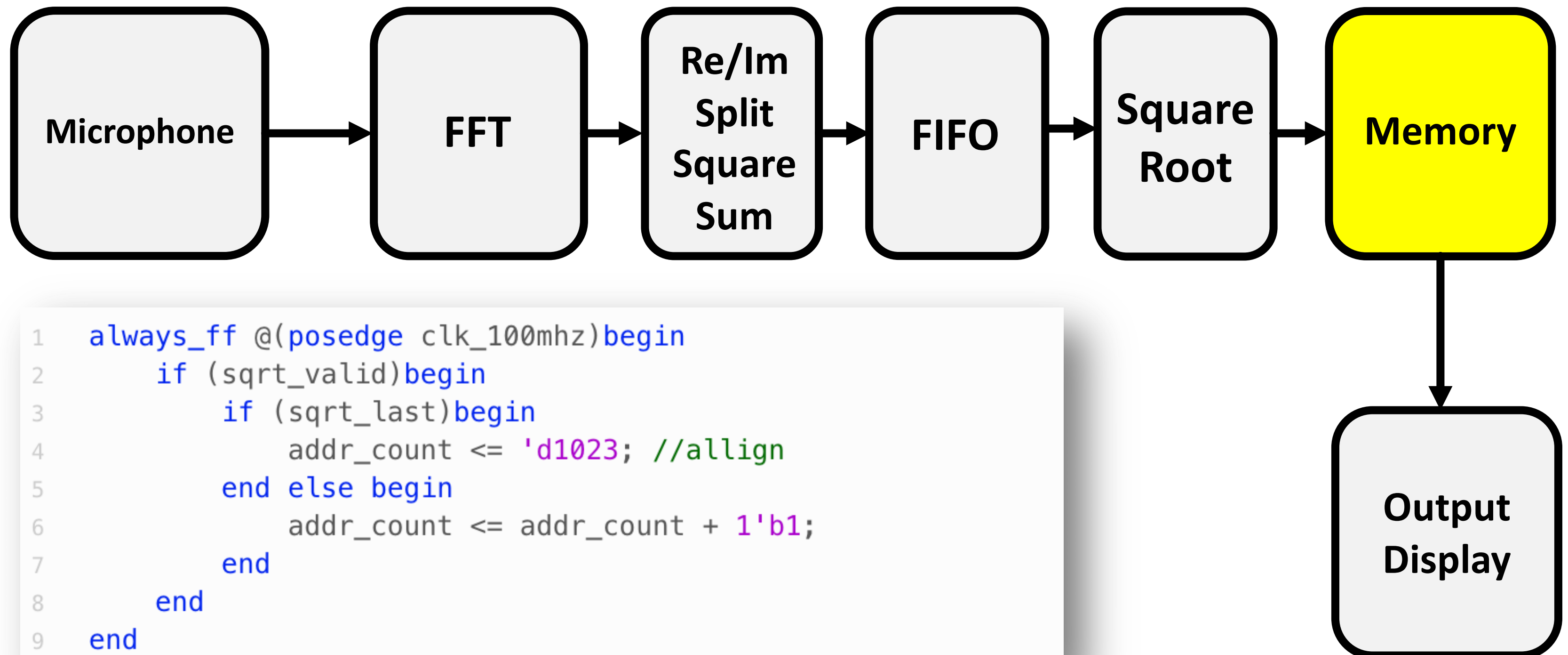
Do we need a FIFO here?

- No. Our Square root is maximally pipelined so it can accept data on every clock cycle.
- I put it in as example here.



- If running low on resources and made CORDIC minimal hardware footprint (so worse throughput) a FIFO could help data buildup from FFT burst.

Real-time Audio Spectrograph




```
always_ff @(posedge clk_100mhz)begin
    if (sqrt_valid)begin
        if (sqrt_last)begin
            addr_count <= 'd1023; //allign
        end else begin
            addr_count <= addr_count + 1'b1;
        end
    end
end
```

```

value_bram mvb (.addra(addr_count+3), .clka(clk_100mhz), .dina({8'b0,sqrt_data}),
                .douta(), .ena(1'b1), .wea(sqrt_valid),.dinb(0),
                .addrb(draw_addr), .clkb(pixel_clk), .doutb(amp_out),
                .web(1'b0), .enb(1'b1));
```

Two Port BRAM

- Calculations Written In as they are created
- Calculations Read Out as needed for video display
- Example of a frame-buffer
- Avoids having to synchronize FFT generation too tightly with video drawing (lab04a/b)

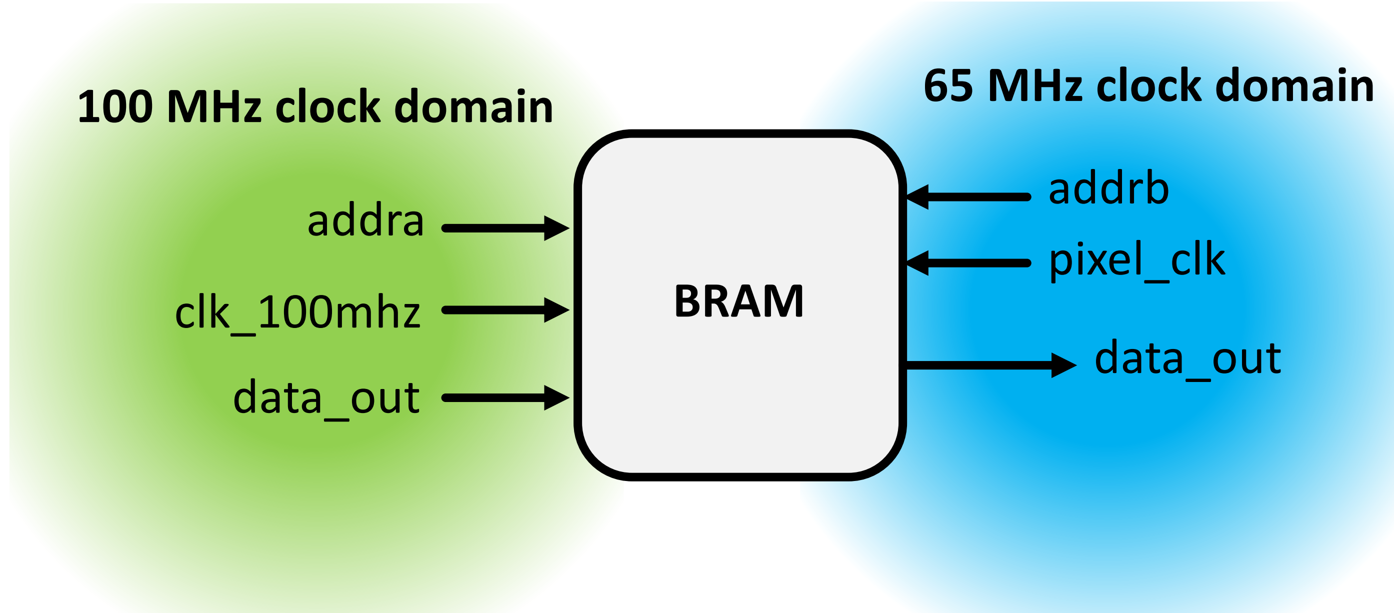
```
xilinx_true_dual_port_read_first_2_clock_ram #(
  .RAM_WIDTH(32),
  .RAM_DEPTH(1024))
  frame_buffer (
  //Write Side (100MHz)
  .addra(addr_count+3),
  .clka(axi_clk), //NEW FOR LAB 04B
  .wea(sqrt_valid),
  .dina({8'b0,sqrt_data}),
  .ena(1'b1),
  .regcea(1'b1),
  .rsta(btnd),
  .douta(),
  //Read Side (65 MHz)
  .addrb(draw_addr),
  .dinb(16'b0),
  .clkb(pixel_clk),
  .web(1'b0),
  .enb(1'b1),
  .rstb(btnd),
  .regceb(1'b1),
  .doutb(amp_out)
);
```

1024 X 32 bit Memory

Why 1024? There's 1024 FFT values to store!
Why 32 bit? Each magnitude is 32 bits

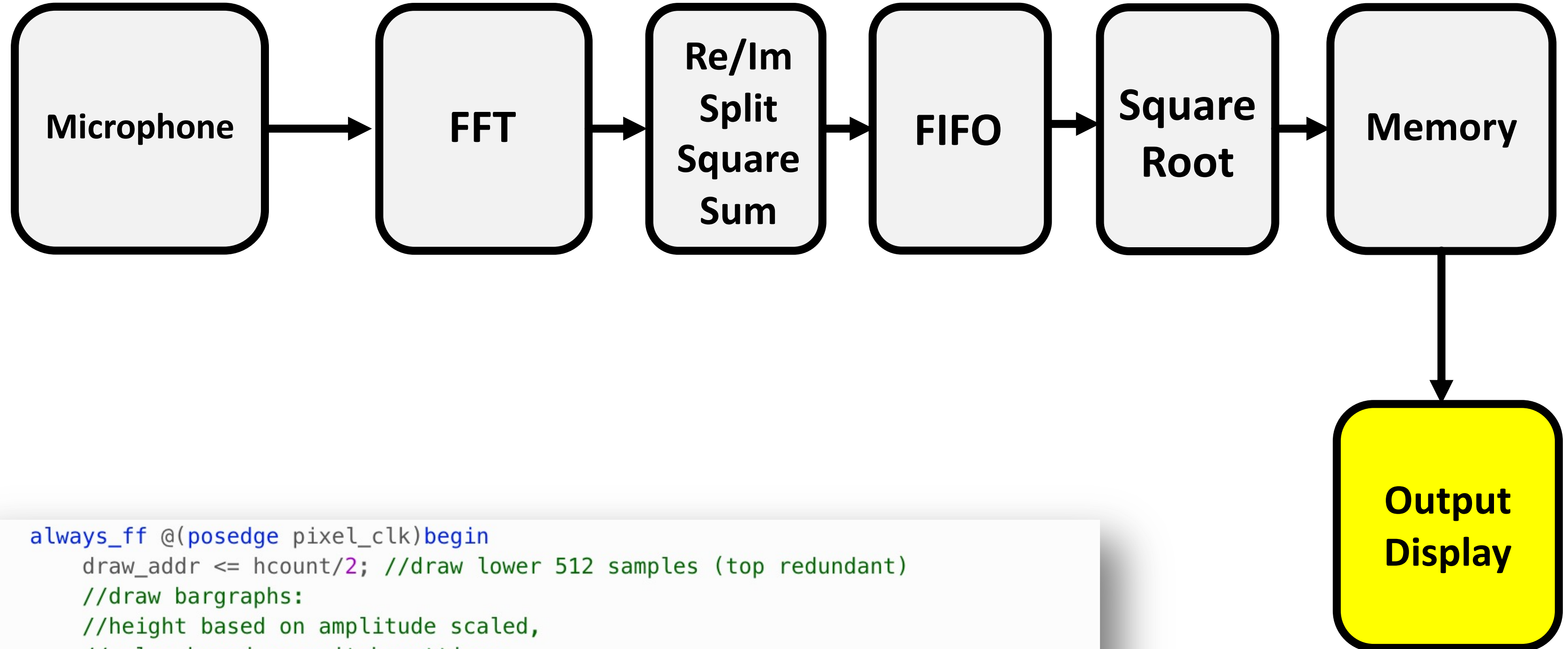
Video Memory

- Two Port Block RAM:
  - Each side separately clocked!
  - Don’t have to worry about running upstream at video clock rate!



Real-time Audio Spectrograph

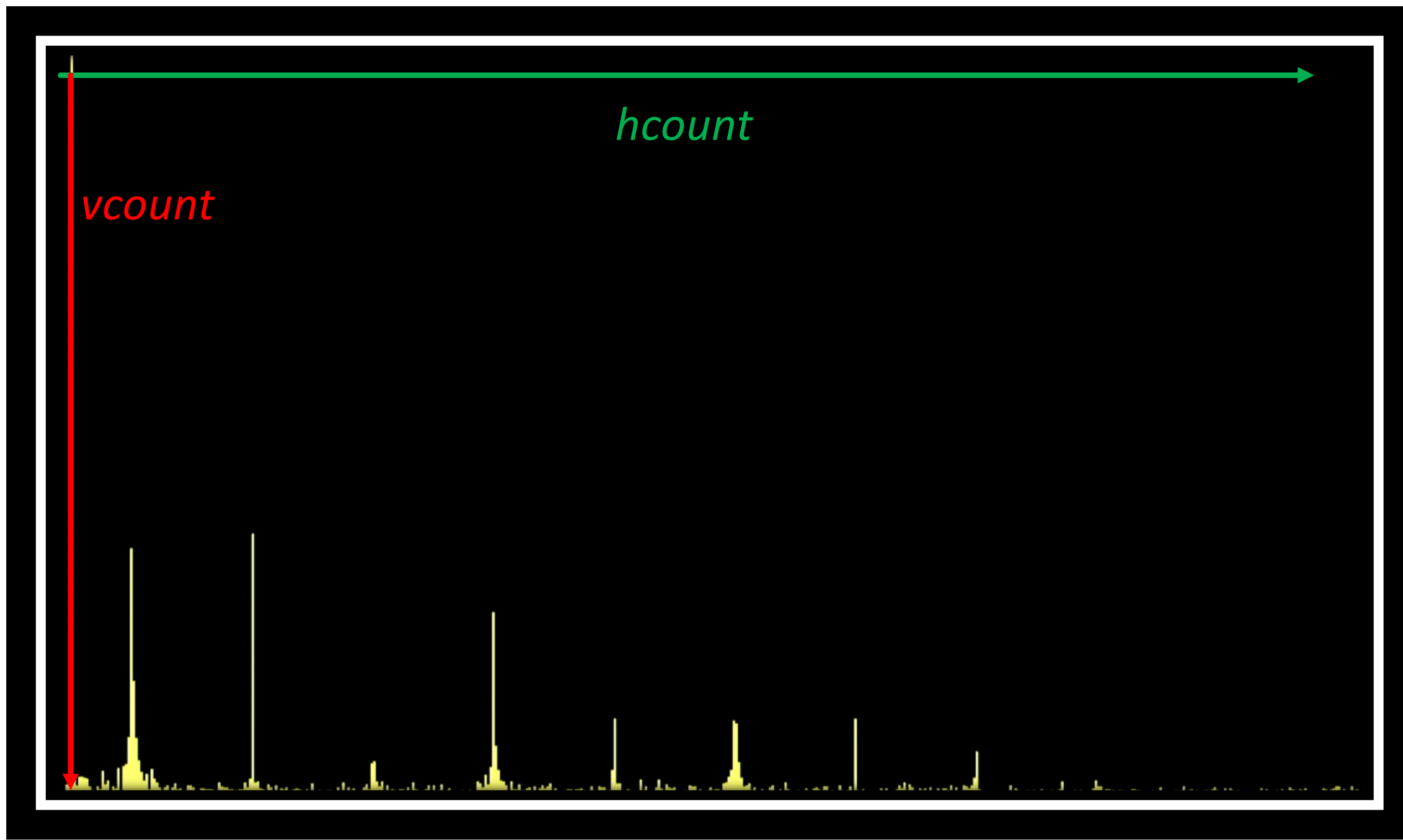
- The last step!




```
always_ff @(posedge pixel_clk)begin
    draw_addr <= hcount/2; //draw lower 512 samples (top redundant)
    //draw bargraphs:
    //height based on amplitude scaled,
    //color based on switch settings
    rgb <= ((amp_out>>sw[3:0])>='d768-vcount)?sw[15:4]:12'b0000_0000_0000;
end
```

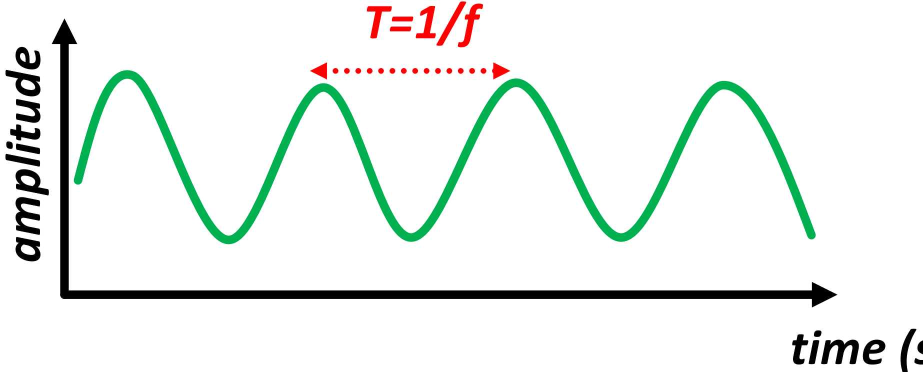
Display Output

```
always_ff @(posedge pixel_clk)begin
    draw_addr <= hcount/2; //draw lower 512 samples (top redundant)
    //draw bargraphs:
    //height based on amplitude scaled,
    //color based on switch settings
    rgb <= ((amp_out>>sw[3:0])>='d768-vcount)?sw[15:4]:12'b0000_0000_0000;
end
```



Sine Waves In

*The square waves in later



Cat

Me

Beyoncé

20th Century Fox

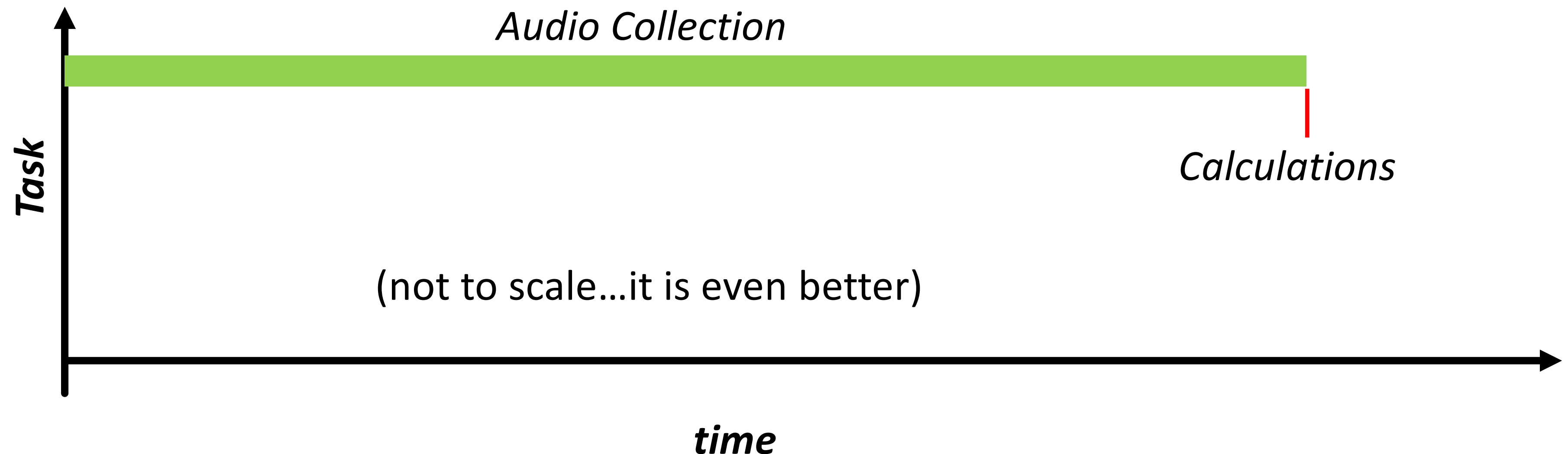
Celine Dion

How Quick to calculate FFT?

- Collect 1024 audio measurements :
  - @19 KHz. Every 52 microseconds (so ~52 milliseconds total)
- Compute 1024 point FFT:
  - 3191 clock cycles (31.91 μs)
- Square and Sum:
  - 2 cycles (20 ns)
- FIFO:
  - 3 cycles overhead latency (30 ns)
- Root:
  - 26 cycles (260 ns)

How Quick?

- After audio clip captured, FFT generated and ready to render in 32.2 μs (3,222 clock cycles)
- Our audio samples are measured *every* 52 μs (LOL)
- We can calculate our entire FFT in between individual audio samples



Are we good on timing

- Report say, “yes”

```
Timing Report

Slack (MET) :              4.459ns  (required time - arrival time)
```

Resource Usage?

- Very minimal!

```
2. Slice Logic Distribution
---------------------------

+-------------------------------------------+------+-------+-----------+-------+
|                 Site Type                 | Used | Fixed | Available | Util% |
+-------------------------------------------+------+-------+-----------+-------+
| Slice                                     | 1049 |     0 |     15850 |  6.62 |
|   SLICEL                                  |  653 |     0 |           |       |
|   SLICEM                                  |  396 |     0 |           |       |
| LUT as Logic                              | 2176 |     0 |     63400 |  3.43 |
|   using O5 output only                    |    5 |       |           |       |
|   using O6 output only                    | 1474 |       |           |       |
|   using O5 and O6                         |  697 |       |           |       |
| LUT as Memory                             |  674 |     0 |     19000 |  3.55 |
|   LUT as Distributed RAM                  |   32 |     0 |           |       |
|     using O5 output only                  |    0 |       |           |       |
|     using O6 output only                  |   32 |       |           |       |
|     using O5 and O6                       |    0 |       |           |       |
|   LUT as Shift Register                   |  642 |     0 |           |       |
|     using O5 output only                  |   20 |       |           |       |
|     using O6 output only                  |  197 |       |           |       |
|     using O5 and O6                       |  425 |       |           |       |
| Slice Registers                           | 4499 |     0 |    126800 |  3.55 |
|   Register driven from within the Slice   | 3425 |       |           |       |
|   Register driven from outside the Slice  | 1074 |       |           |       |
|     LUT in front of the register is unused |  570 |       |           |       |
|     LUT in front of the register is used  |  504 |       |           |       |
| Unique Control Sets                       |   44 |       |     15850 |  0.28 |
+-------------------------------------------+------+-------+-----------+-------+
* Note: Available Control Sets calculated as Slice Registers / 8, Review the Control Sets Report for
more information regarding control sets.
```

Resource Usage?

- Very minimal!

3. Memory

Site Type	Used	Fixed	Available	Util%
Block RAM Tile	5	0	135	3.70
RAMB36/FIFO*	2	0	135	1.48
RAMB36E1 only	2			
RAMB18	6	0	270	2.22
RAMB18E1 only	6			

* Note: Each Block RAM Tile only has one FIFO logic available and therefore can accommodate only one FIFO36E1 or one FIFO18E1. However, if a FIFO18E1 occupies a Block RAM Tile, that tile can still accommodate a RAMB18E1

4. DSP

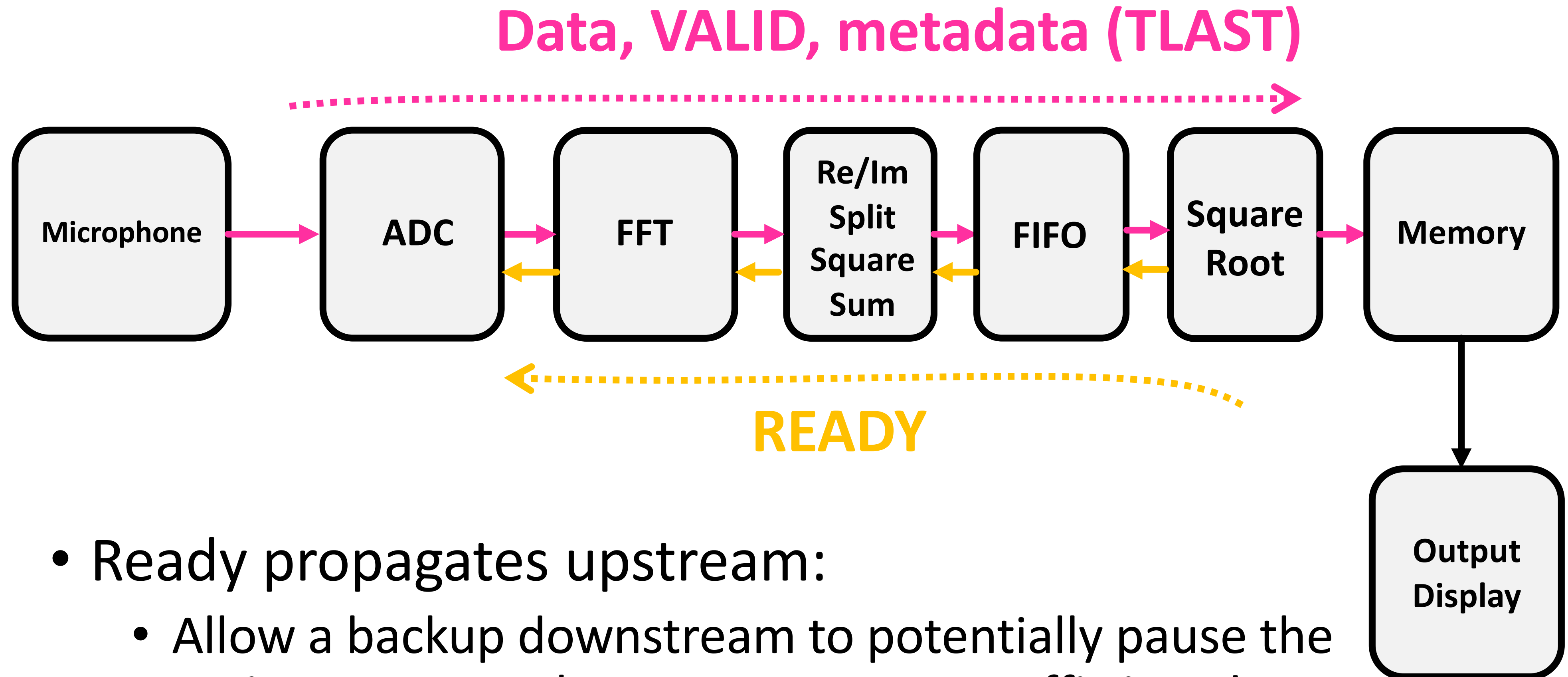
Site Type	Used	Fixed	Available	Util%
DSPs	14	0	240	5.83
DSP48E1 only	14			

Make it much better

- This was a 1024 point FFT over 10 kHz
- Could totally do way more points at differing sample rates to get better frequency resolution
  - Many lower musical notes are differentiated by **< 1Hz**
- Could also do:
  - Overlapping Frames/do some averaging
  - Notch filter that 60 Hz
  - Window samples to cut down on artifacts
  - Many other things

Different Directions

- Data Propagates downstream:



- Ready propagates upstream:
  - Allow a backup downstream to potentially pause the entire system at the start to prevent traffic jams!

Usefulness of Metadata or markers

- If data takes a really long time you can also activate a USER field to send along with DATA
- USER values will be unchanged but will get pipelined properly along with the corresponding data they're sent in with