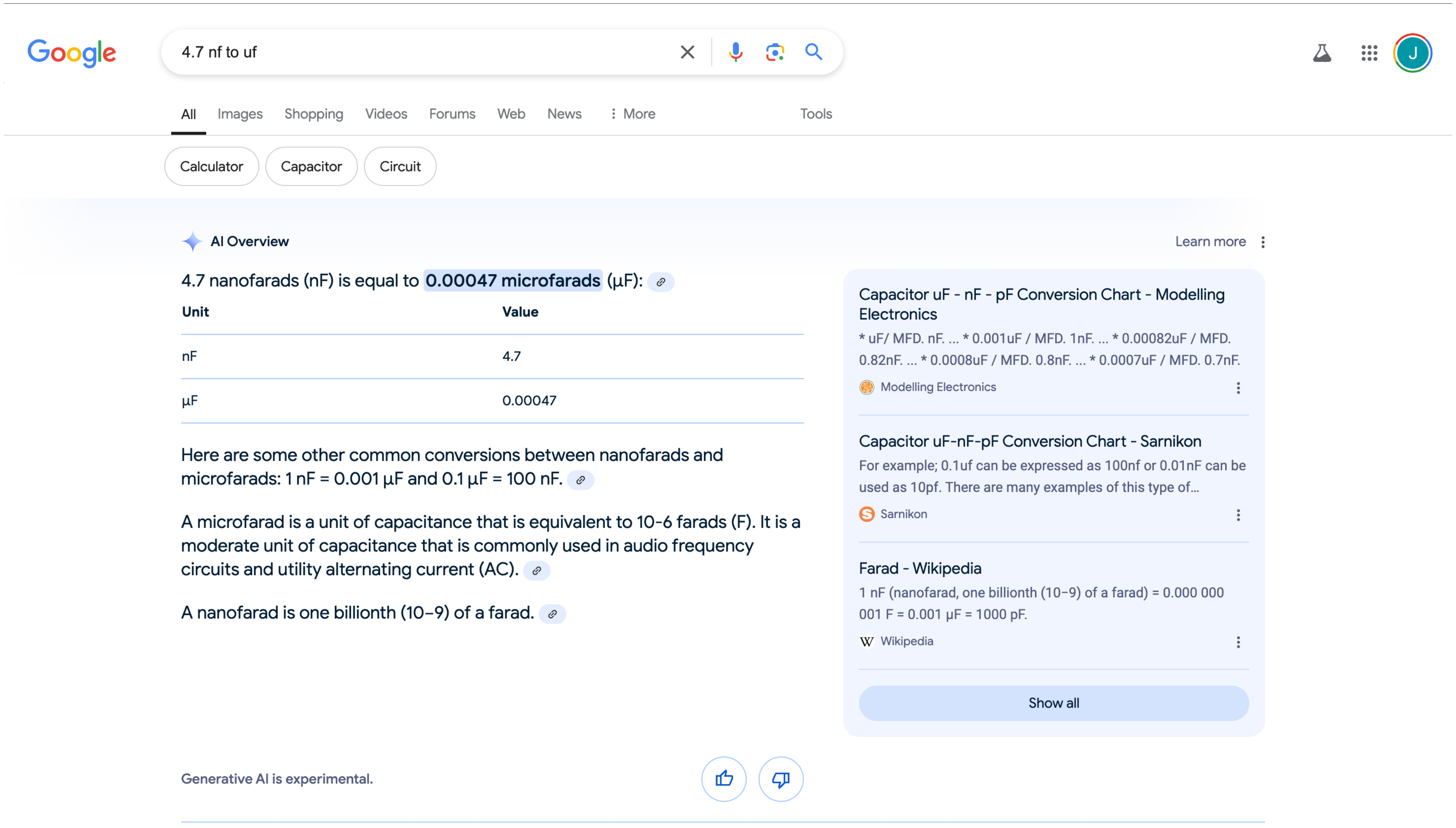
The FPGA, AXI, Etc…



4.7 nanofarads (nF) is equal to 0.00047 microfarads (μF):

Unit	Value
nF	4.7
μF	0.00047

Here are some other common conversions between nanofarads and microfarads: 1 nF = 0.001 μF and 0.1 μF = 100 nF.

A microfarad is a unit of capacitance that is equivalent to 10-6 farads (F). It is a moderate unit of capacitance that is commonly used in audio frequency circuits and utility alternating current (AC).

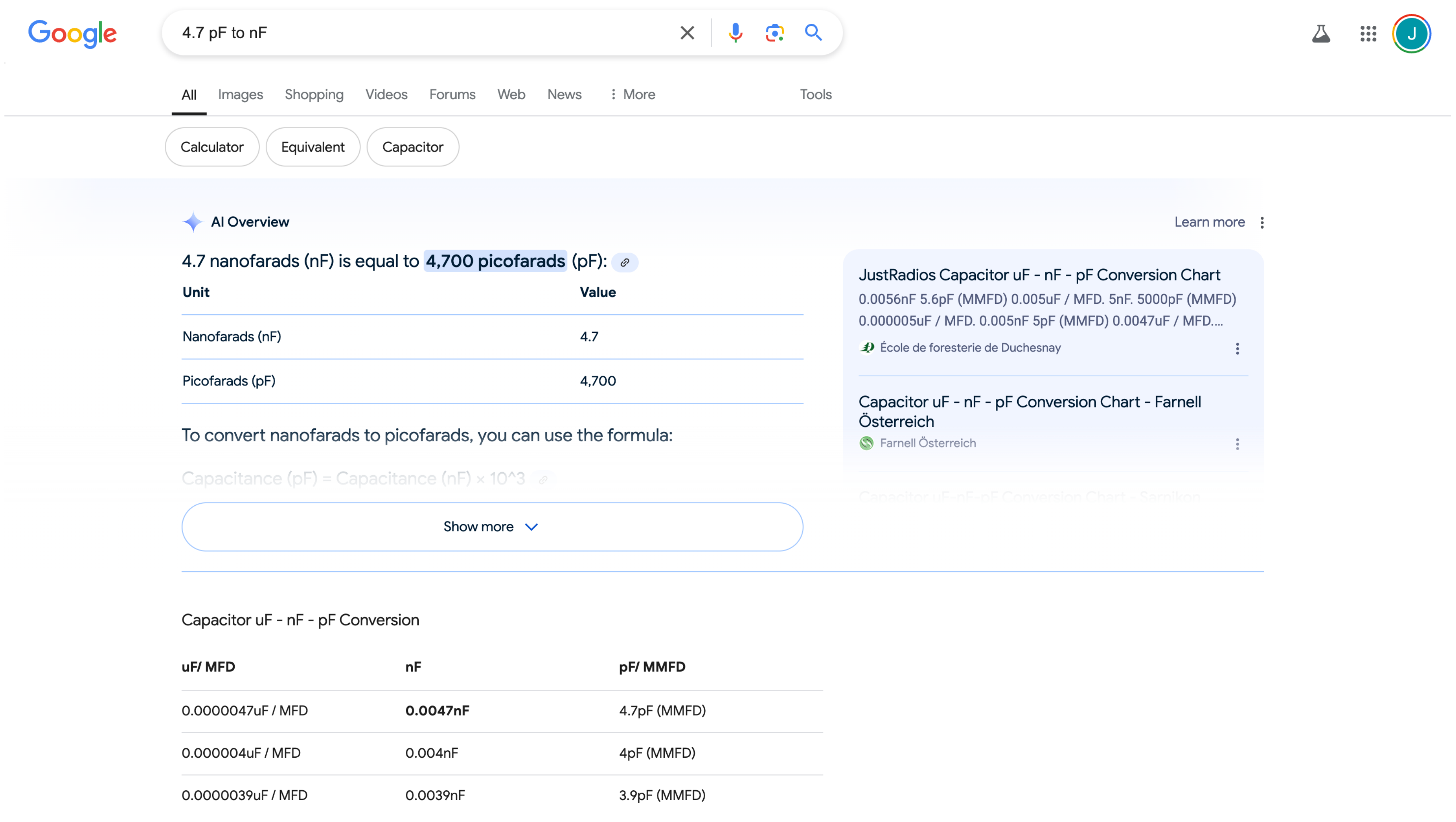
A nanofarad is one billionth (10−9) of a farad.

Capacitor uF - nF - pF Conversion Chart - Modelling Electronics
* uF/ MFD. nF. ... * 0.001uF / MFD. 1nF. ... * 0.00082uF / MFD. 0.82nF. ... * 0.0008uF / MFD. 0.8nF. ... * 0.0007uF / MFD. 0.7nF.
Modelling Electronics

Capacitor uF-nF-pF Conversion Chart - Sarnikon
For example; 0.1uf can be expressed as 100nf or 0.01nF can be used as 10pf. There are many examples of this type of...
Sarnikon

Farad - Wikipedia
1 nF (nanofarad, one billionth (10−9) of a farad) = 0.000 000 001 F = 0.001 μF = 1000 pF.
Wikipedia

Generative AI is experimental.



4.7 pF to nF

All | Images | Shopping | Videos | Forums | Web | News | More | Tools

Calculator | Equivalent | Capacitor

AI Overview

Learn more

4.7 nanofarads (nF) is equal to 4,700 picofarads (pF):

Unit	Value
Nanofarads (nF)	4.7
Picofarads (pF)	4,700

To convert nanofarads to picofarads, you can use the formula:

Capacitance (pF) = Capacitance (nF) × 10^3

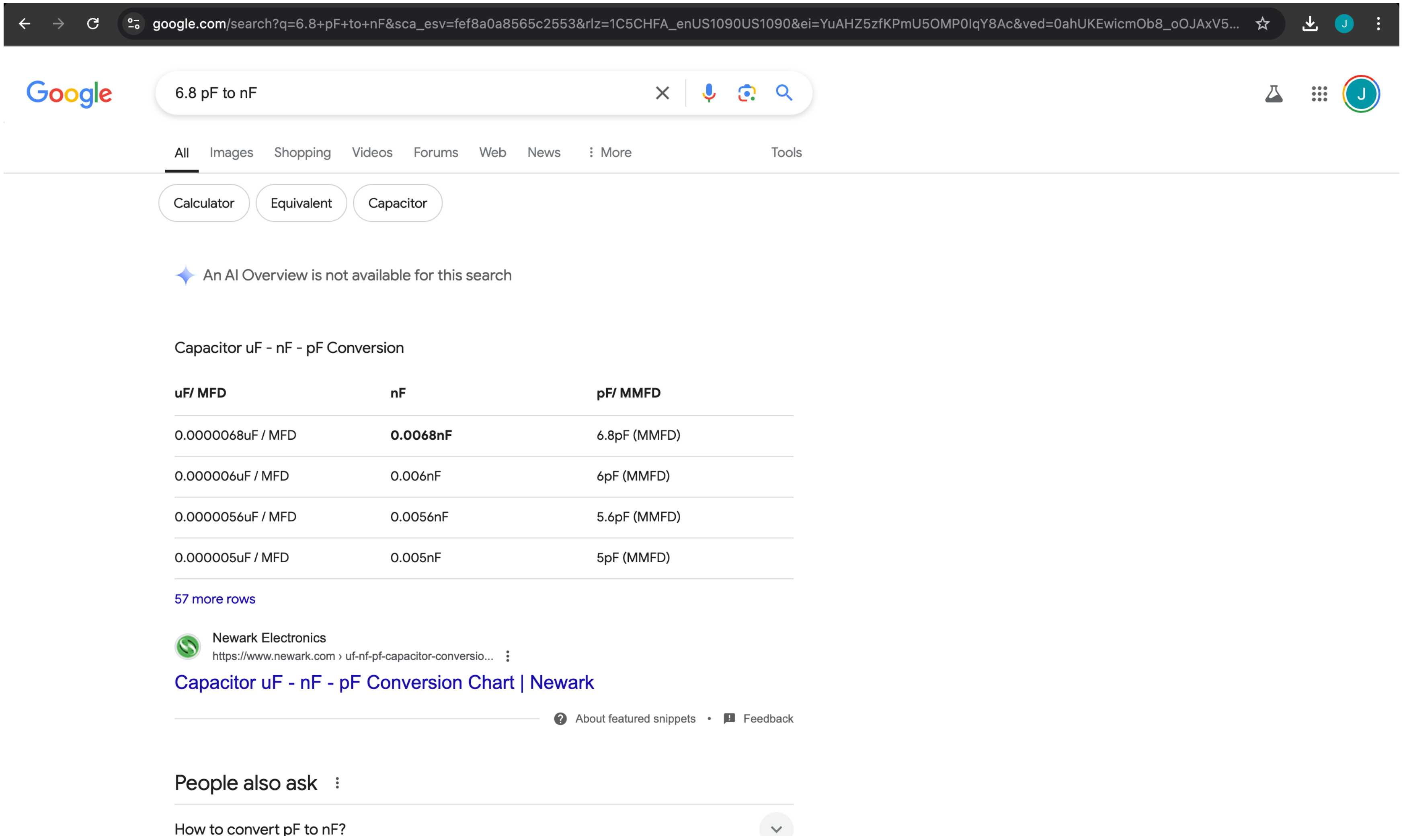
Show more

JustRadios Capacitor uF - nF - pF Conversion Chart
0.0056nF 5.6pF (MMFD) 0.005uF / MFD. 5nF. 5000pF (MMFD) 0.000005uF / MFD. 0.005nF 5pF (MMFD) 0.0047uF / MFD....
École de foresterie de Duchesnay

Capacitor uF - nF - pF Conversion Chart - Farnell Österreich
Farnell Österreich

Capacitor uF - nF - pF Conversion

uF/ MFD	nF	pF/ MMFD
0.0000047uF / MFD	**0.0047nF**	4.7pF (MMFD)
0.000004uF / MFD	0.004nF	4pF (MMFD)
0.0000039uF / MFD	0.0039nF	3.9pF (MMFD)


google.com/search?q=6.8+pF+to+nF&sca_esv=fef8a0a8565c2553&rlz=1C5CHFA_enUS1090US1090&ei=YuAHZ5zfKPmU5OMP0lqY8Ac&ved=0ahUKEwicmOb8_oOJAxV5...
Google
6.8 pF to nF
All
Images
Shopping
Videos
Forums
Web
News
More
Tools
Calculator
Equivalent
Capacitor

An AI Overview is not available for this search

Capacitor uF - nF - pF Conversion

uF/ MFD	nF	pF/ MMFD
0.0000068uF / MFD	**0.0068nF**	6.8pF (MMFD)
0.000006uF / MFD	0.006nF	6pF (MMFD)
0.0000056uF / MFD	0.0056nF	5.6pF (MMFD)
0.000005uF / MFD	0.005nF	5pF (MMFD)

57 more rows

Newark Electronics
https://www.newark.com › uf-nf-pf-capacitor-conversio...

Capacitor uF - nF - pF Conversion Chart | Newark

About featured snippets • Feedback

People also ask

How to convert pF to nF?

Administration

- Week 05 due last night
- Week 06 out after class today (might be delayed by a couple hours)…it is short.
  - two pages
- Week 07 (next week) will involve some convolution/image processing (regular length)
- Week 08 will be short after that, look at soft processing cores*
- Then final project time

*that's the plan anyways

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…)
  - 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…

However if you want to do a video 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
- Or if you want to make a simple game, then you really need push it the limits.

Excellent “simple” game

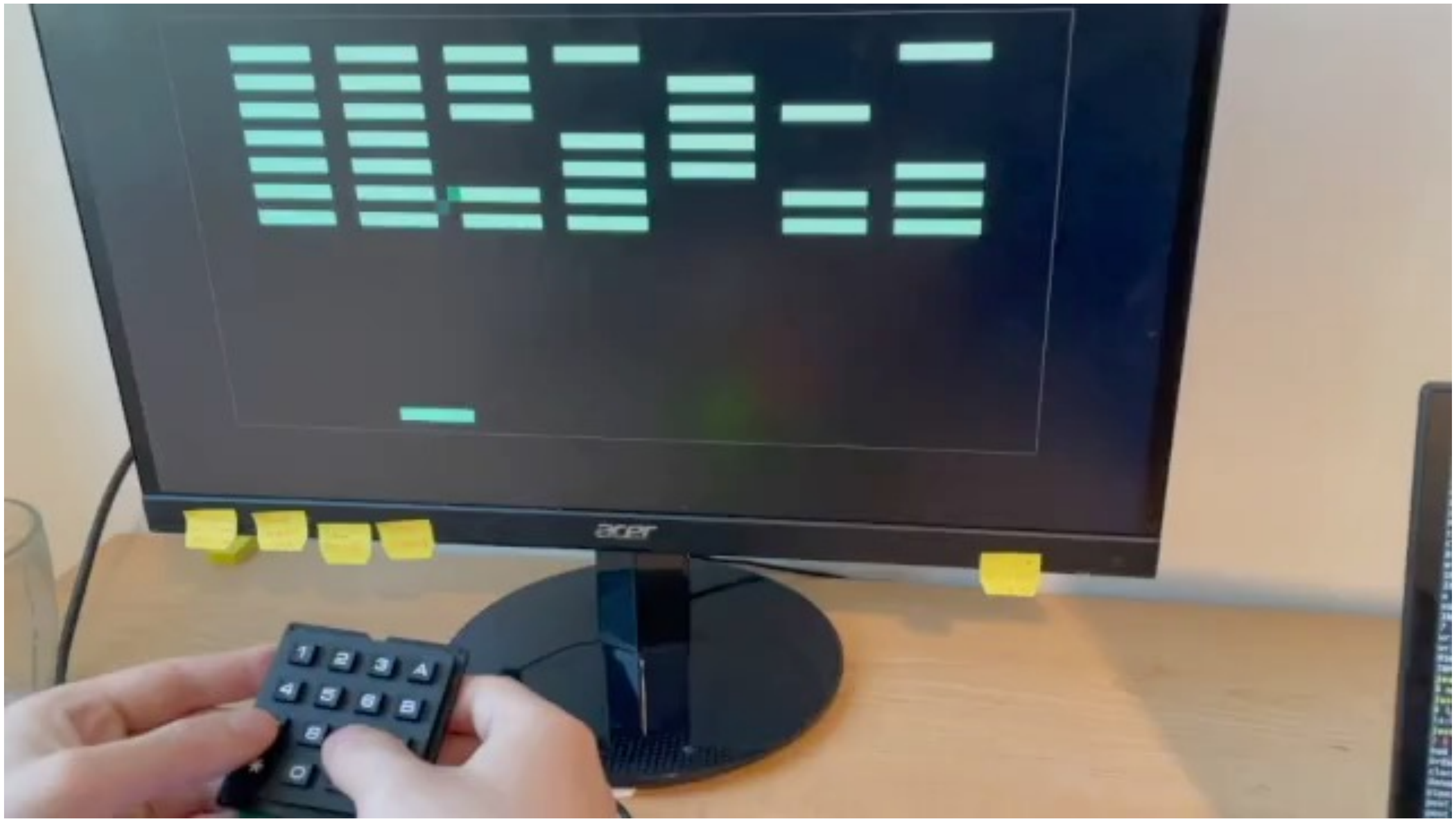
Pacman Extreme

- Used basically all the resources on that FPGA
  - Partially through poor planning on their part
  - Partially through over-pipelining and over-parallelization
- But the attention to detail and overall depth, was extreme
- And some poor choices with utilization resulted in them having to be very clever with how a lot of aspects of their higher-level design worked out
- Team built supplemental tools to aid in design:
  - Kim wrote a javascript app that would make .mem files of all their custom sprites since she got so sick of making them manually, for example

Complexity

- The complexity must come from stuff you do!
- You cannot take week 05’s stuff and week 07’s stuff and glue them together and have an A-level project.
- Using UART to talk to a device that ”does wifi” does not actually have much technical merit…and does not mean you made a wifi system.
- The final project will be graded on what you did and contributed.

Chip 8 Emulator

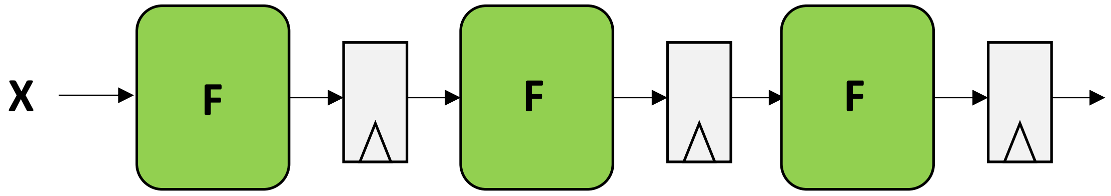


Chip 8 Emulator

- Chip8 is like 50 years old/early attempt at a virtual machine/game engine
- Has a large online following because it is weird and is a great first emulator to write since the instruction set is very tiny (and because once you get it working you have tons of stuff to test on it)
- Many people write emulators and write games for it.
- This team built an emulator and then did all the emulator tuning stuff and then ran a bunch of them in parallel (FPGA strength)…something most people can’t do with a software simulation/emulation

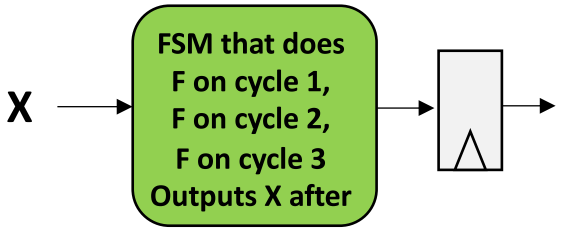
More Advanced Pipelining

This is the Great Tradeoff!



More resources,
Better Throughput
Same Latency

OR

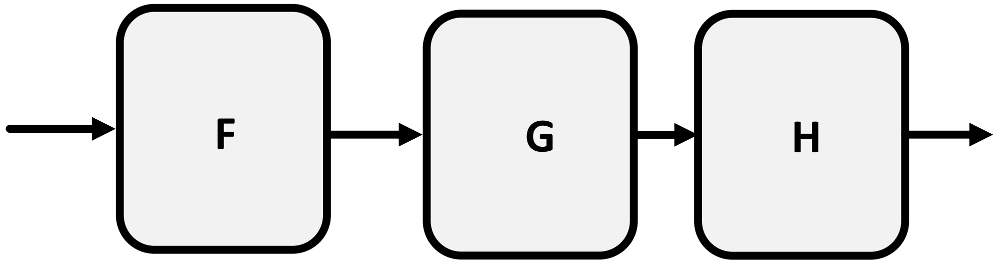


Fewer resources,
Worse Throughput
Same Latency

- Base on what you need for the design!

Pipelining II

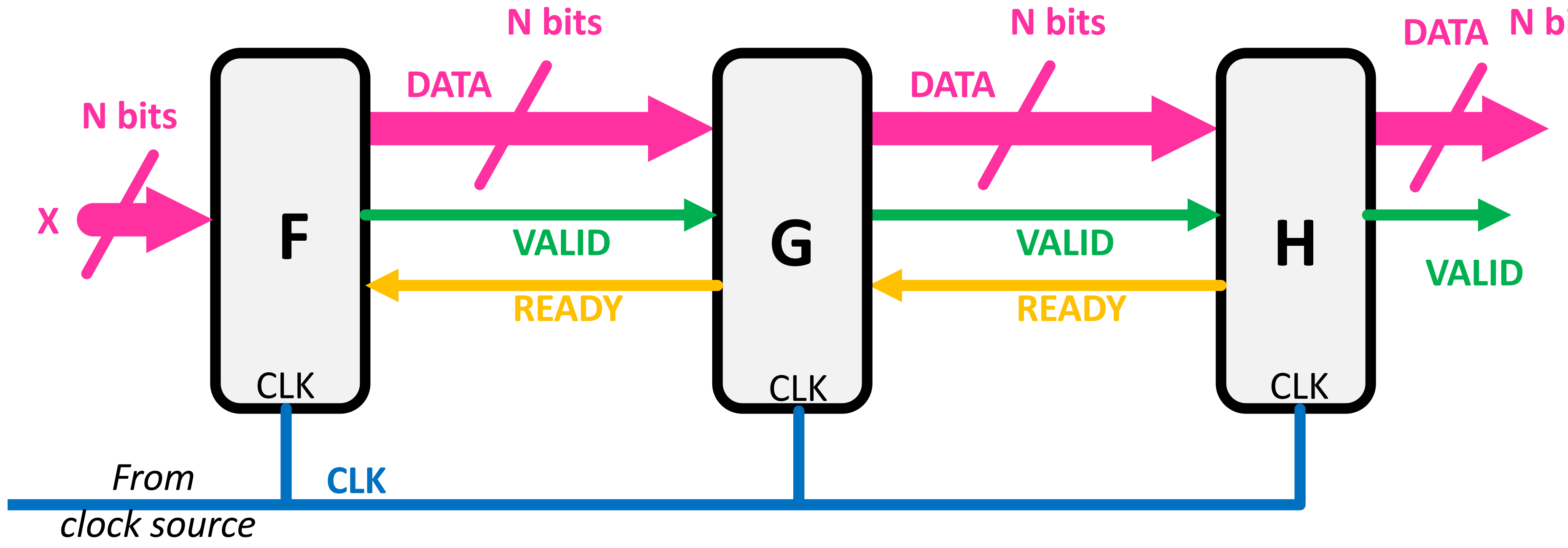
- As we make larger-level systems



- As we make larger-level systems we need to pipeline data through systems which might take varying amounts of time
- And the cycles of latency can become 1000’s of cycles

Pipelining II

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



What is IP?

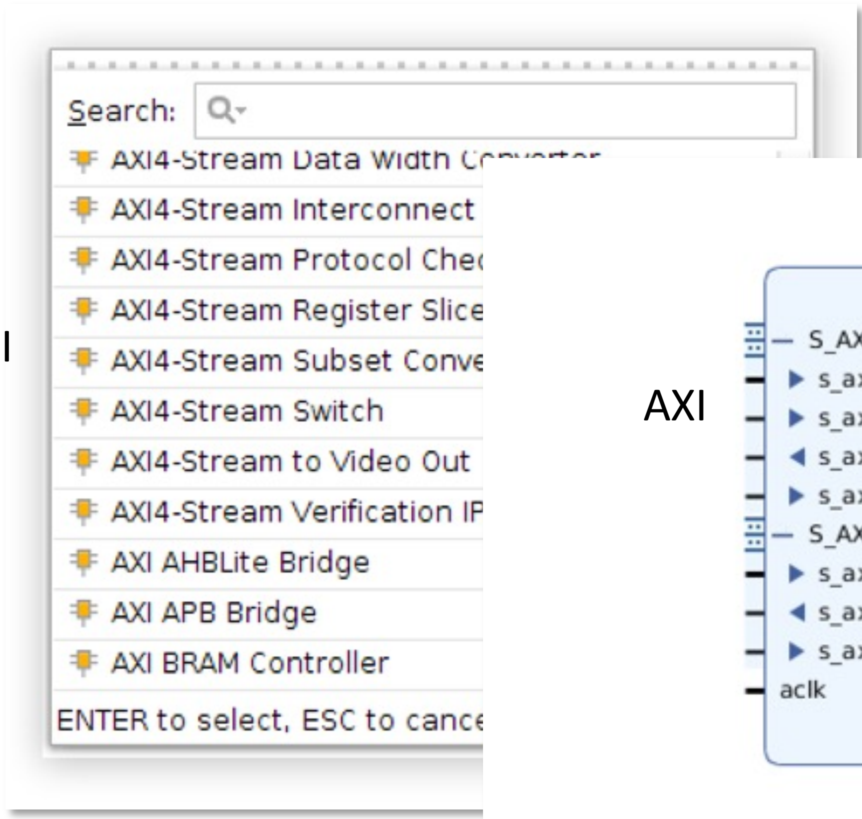
- Often times you'll hear people call a module they made "IP"…short for "intellectual property"
- These basically let you specify an extremely parameterizable module
- In Vivado there are IP which you can instantiate.
- There's a ton of effort that goes into enabling a particular circuit in a modifiable 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.

What are some attributes of extensible modules?

- Well documented, or at least some attempt at documentation, or at least the ability to read the source code
- Speak a common language…
  - Accept inputs in a commonly accepted way
  - Generate results in a commonly accepted way
- We need standards!

AXI Everywhere

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


Search:

AXI4-Stream Data Width Converter

AXI4-Stream Interconnect

AXI4-Stream Protocol Chec

AXI4-Stream Register Slice

AXI4-Stream Subset Conve

AXI4-Stream Switch

AXI4-Stream to Video Out

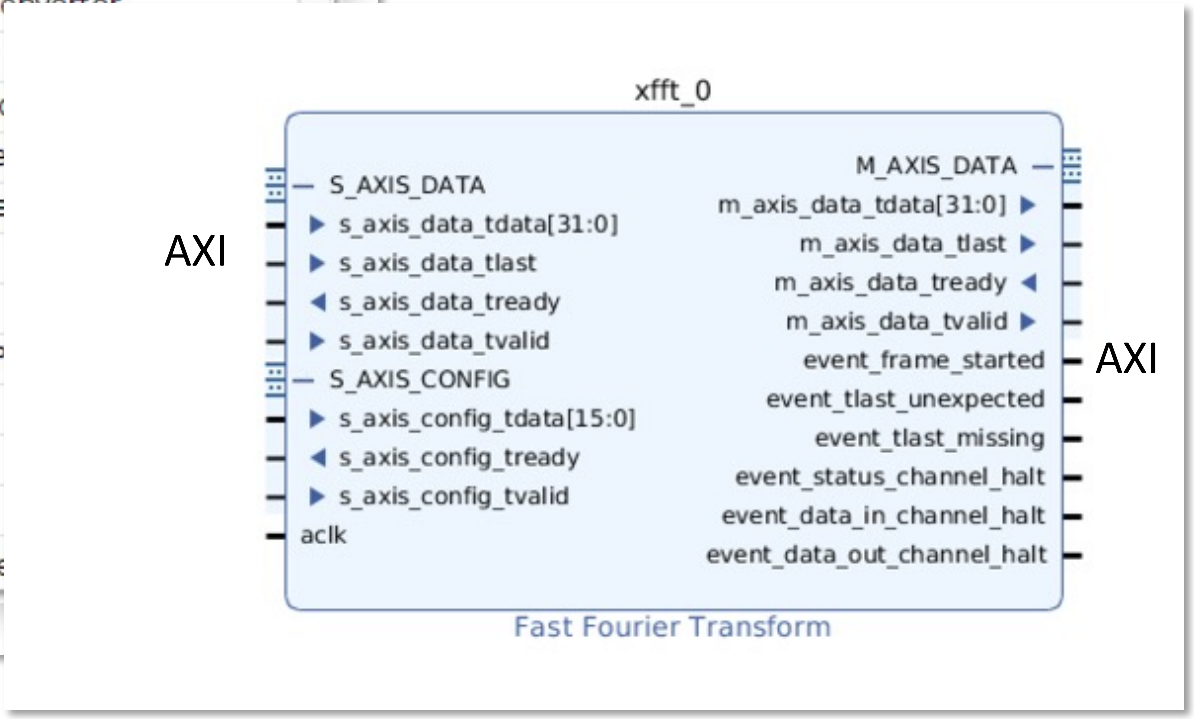
AXI4-Stream Verification IP

AXI AHBLite Bridge

AXI APB Bridge

AXI BRAM Controller

ENTER to select, ESC to cance



AXI

AXI

AXI

AXI

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

AXI Life

- A lot of modules written for FPGA or ASIC application build towards AXI interfaces
- Doing this allows things to be more plug-and-play than if you rolled your own
- So we should go over how it works!

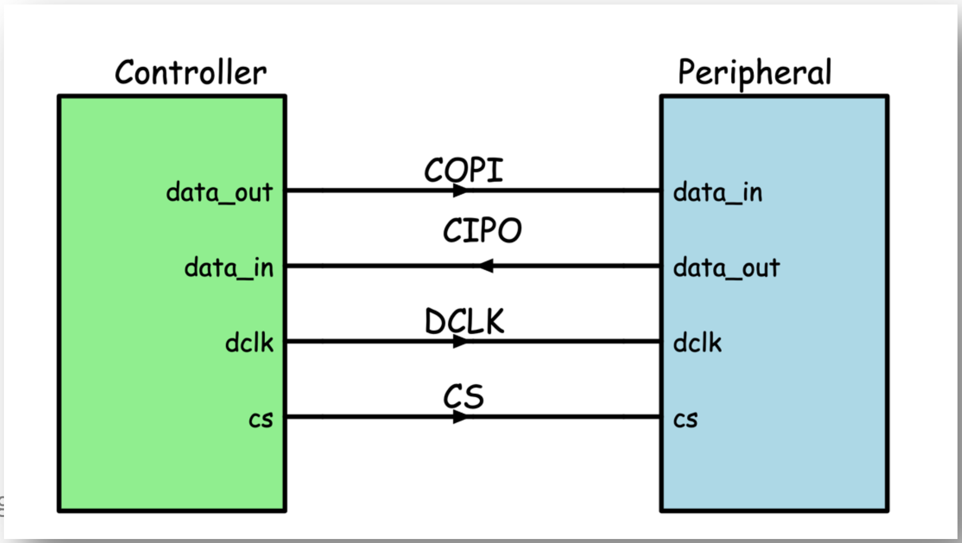
Three General Flavors of AXI4

- **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
- There has been successful transition to **Controller** and **Peripheral** in some areas
- Lab 2!!!



Note on Terminology

- The Xilinx AXI protocol uses this Master/Slave terminology
- And continues to do so into 2024.
- In 6.205 I’m going to just use Main/Secondary or just “M” and “S”, but the docs and even some port names distinctly use Master/Slave.
- This way we can keep using the datasheets.
- And then continue to push AMD/Xilinx to change it.

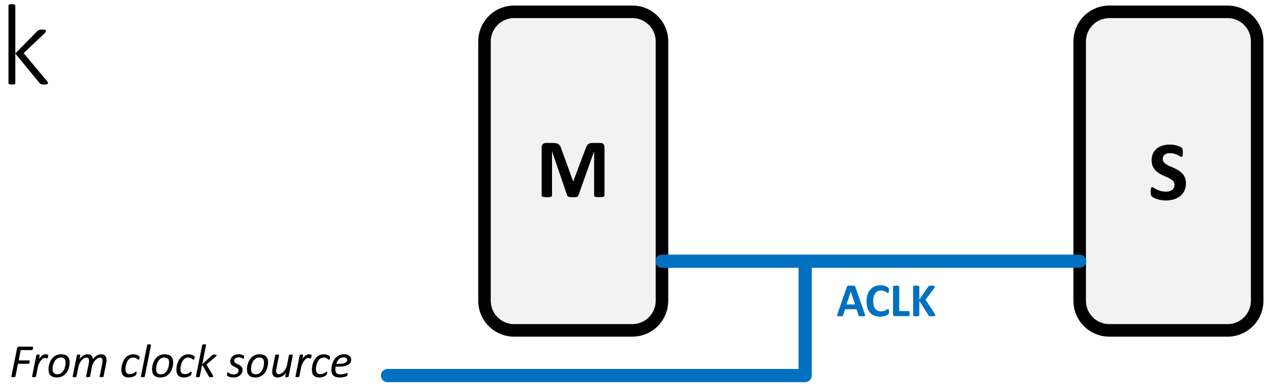
Others than AXI?

- 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….so that’s what we’ll look at.

So the AXI Protocol!

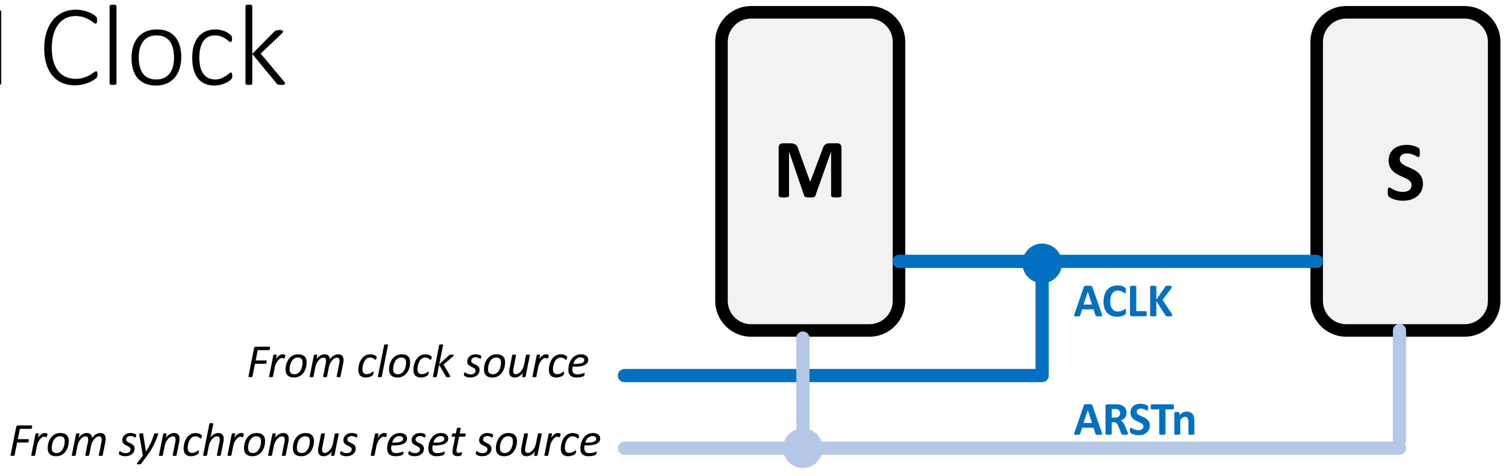
- Made up of wires
- These wires serve specific purposes.
- Some are universal to all AXI4S channels, and others are specific

AXI Clock



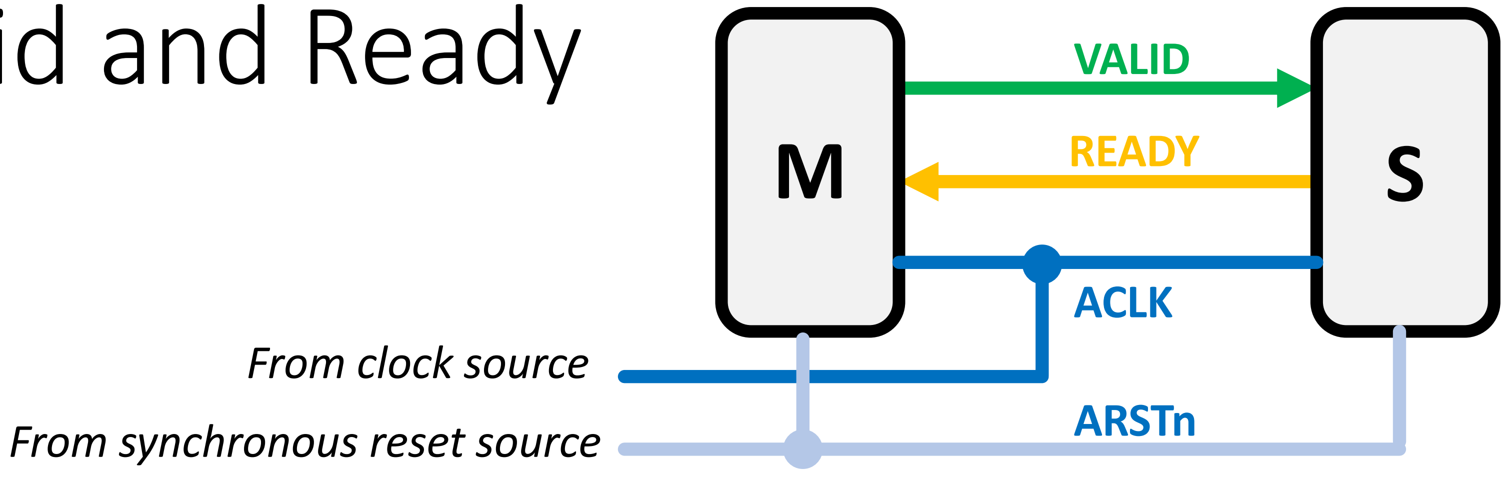
- Everything in system will run off of AXI clock usually called **ACLK** in documentation
- No combinatorial paths between inputs and outputs. Everything must be registered.
- All signals are sampled **on rising edge**

AXI Clock



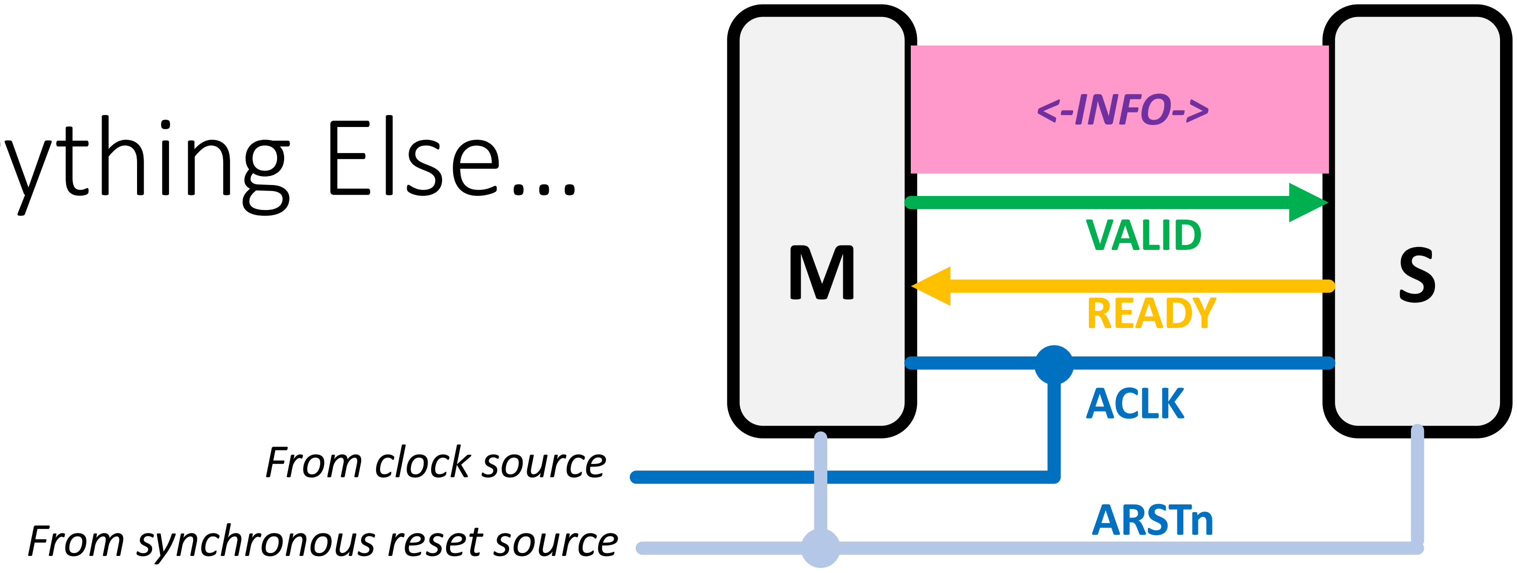
- Everything in system will run off of AXI clock usually called **ACLK** in documentation
- No combinatorial paths between inputs and outputs. Everything must be registered.
- All signals are sampled **on rising edge**
- AXI modules should also have Reset pins. AXI work **<u>ACTIVE LOW</u>** so the Reset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

Valid and Ready



- All of AXI uses the same handshake procedure:
- The *creator* of a data "M" generates a **VALID** signal
- The *destination* of data "S" 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/sets of 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)

Generalized Transaction

- All Channel Interactions follow same high-level structure
- Data is handed off IF AND ONLY IF VALID and READY are high on the rising edge of the clock
- If that happens, both parties must realize that data transfer has happened

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

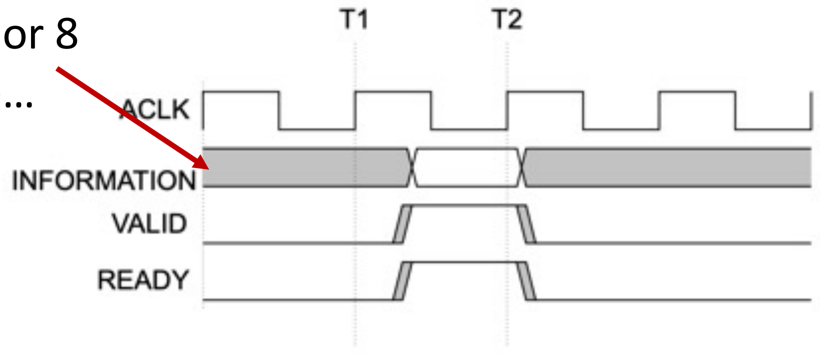


Figure A3-4 VALID with READY handshake

VALID then READY

- Valid can be high first
- Then ready can show up later
- Only when both are high is data exchanged

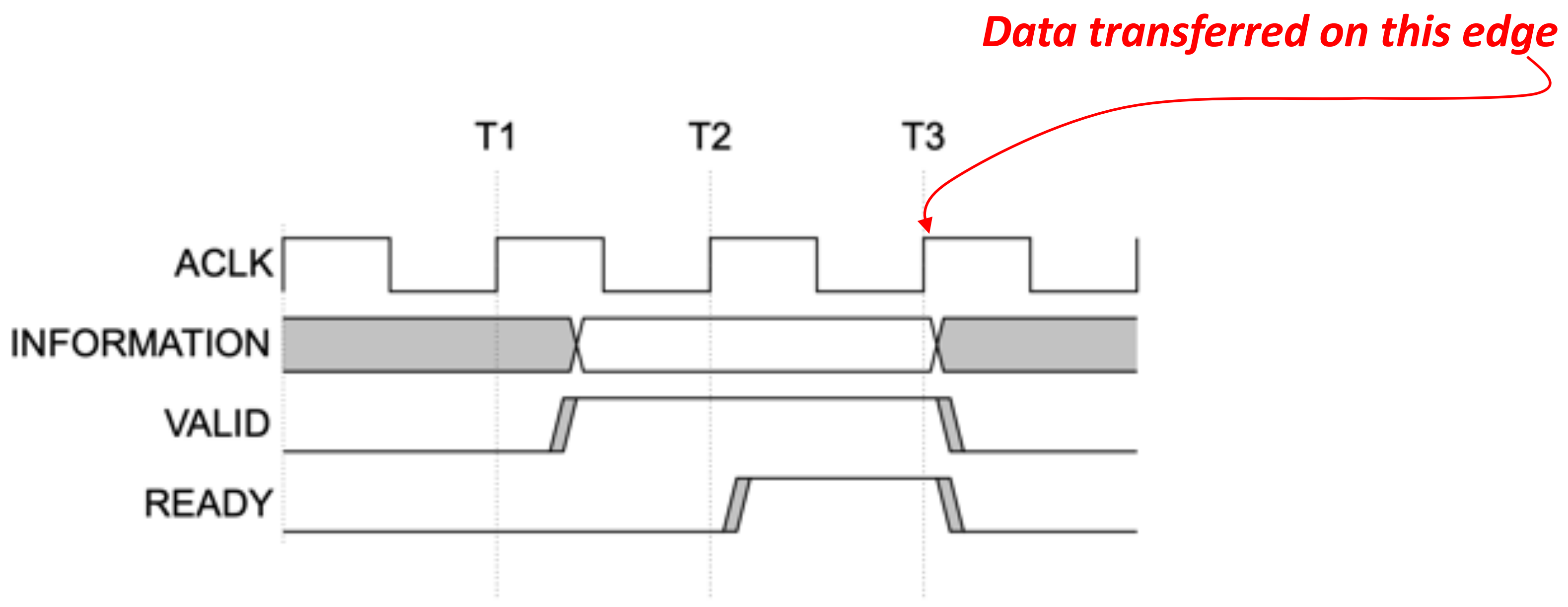


Figure A3-2 VALID before READY handshake

READY then VALID

- Ready can be high first
- Then Valid can show up later
- Only when both are high is data exchanged

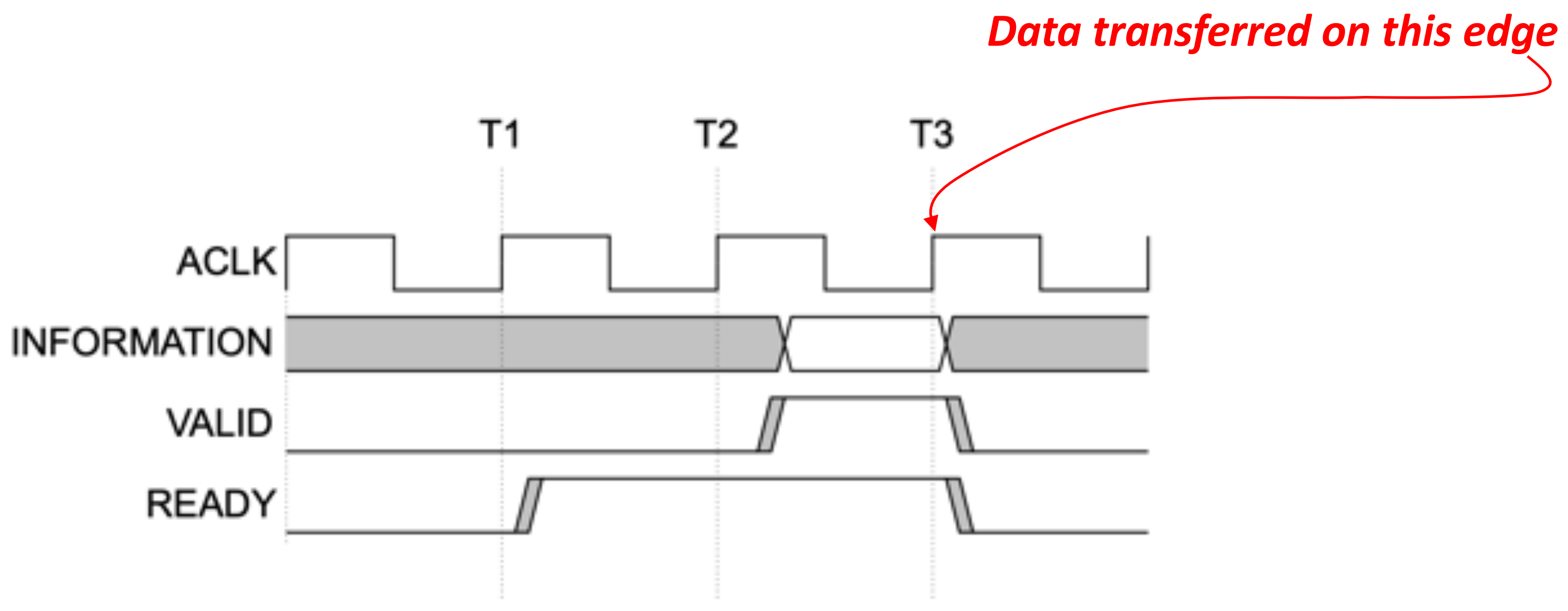


Figure A3-3 READY before VALID handshake

READY WITH VALID

- Ready and Valid come high at the same time
- Totally allowed
- Data is exchanged on that clock edge

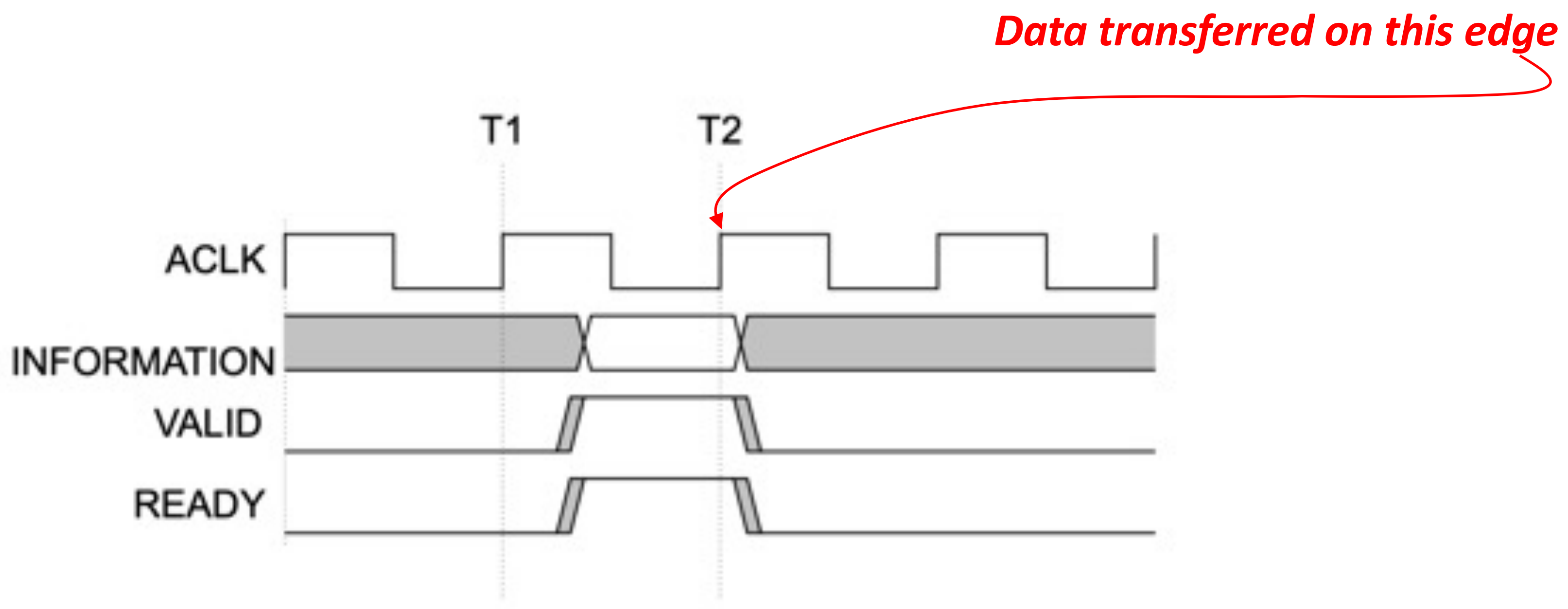


Figure A3-4 VALID with READY handshake

Generalized Transaction

- Can have multiple channels
- They all follow the same spec though
- All Channel Interactions follow same high-level structure

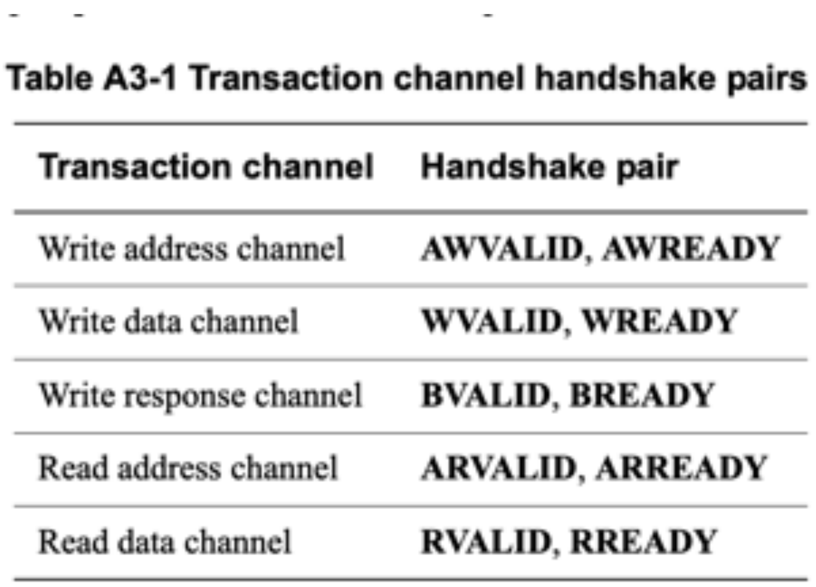


Table A3-1 Transaction channel handshake pairs

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**

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 *may* wait until it detects a **VALID** signal before it asserts its corresponding **READY** signal.
- In other words **READY** can depend on **VALID**, but not the other way around.
- Failure to adhere to this can lead to what's known as **"dead-lock"**
- Fail to Follow these rules and could have devices wait infinitely.
  - Like when two people keep going "no, after you" at a door.

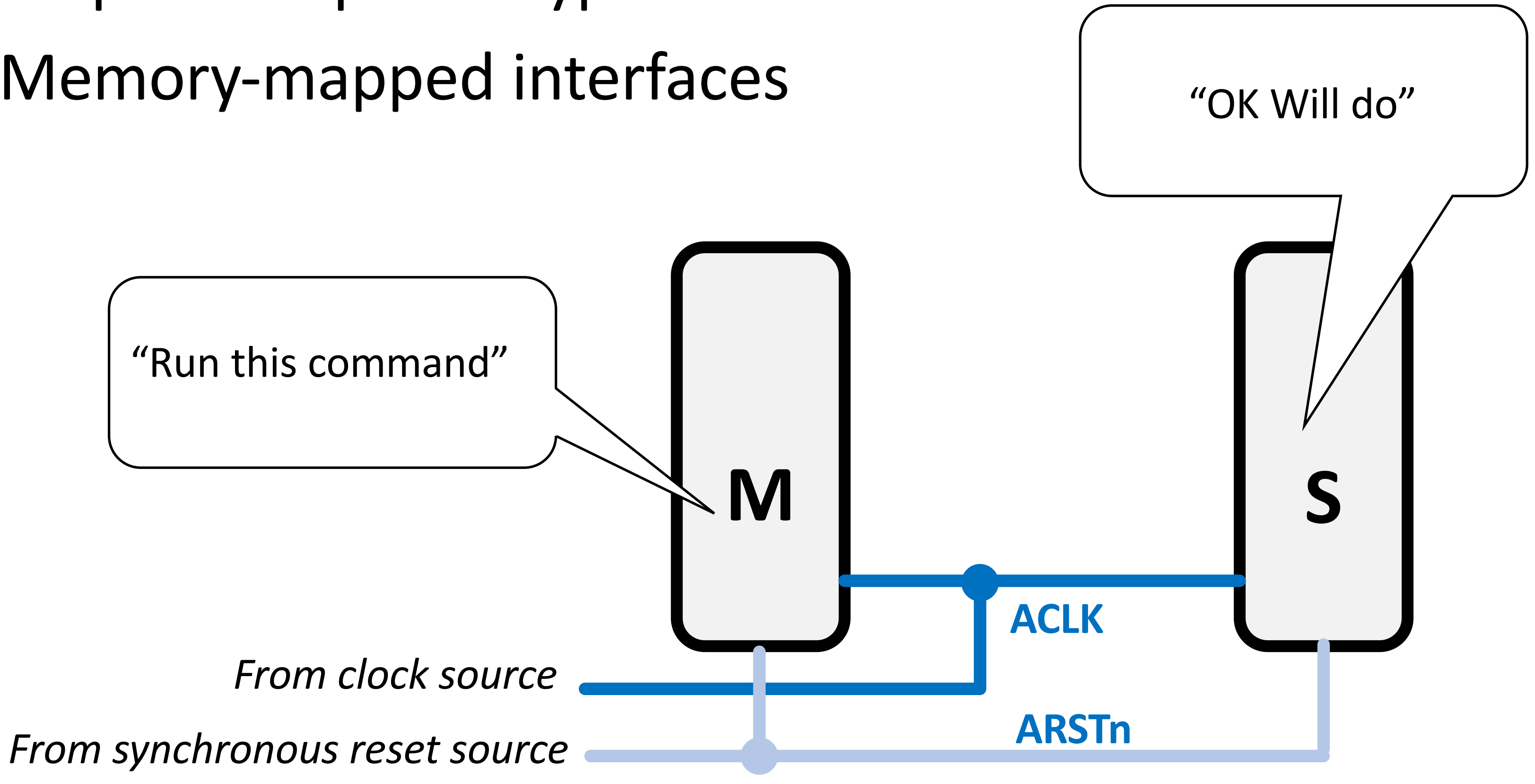
Three General Flavors of AXI4

- **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

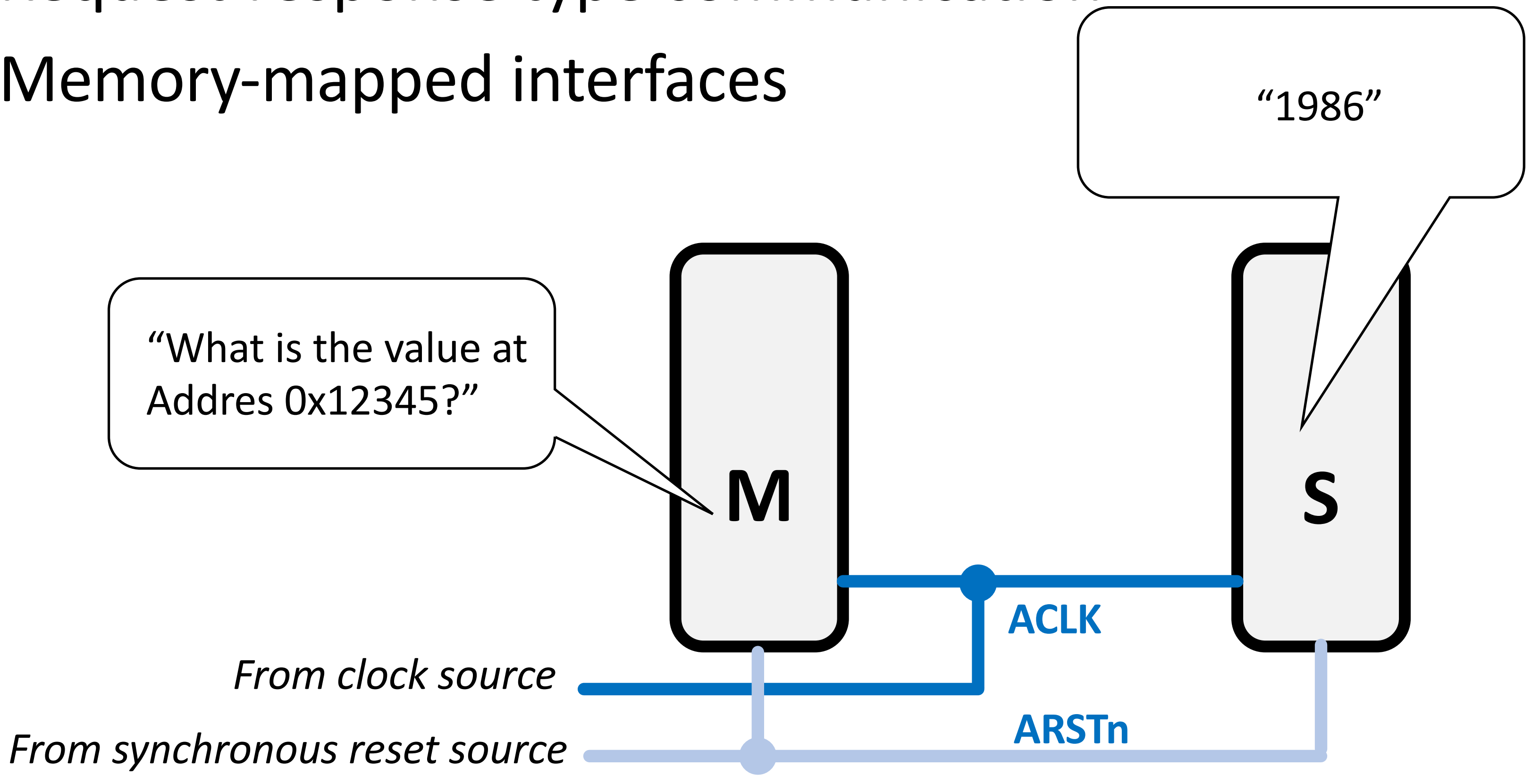
Full AXI and AXI Lite

- Meant for back-and-forth communication
- Request-response type communication
- Memory-mapped interfaces



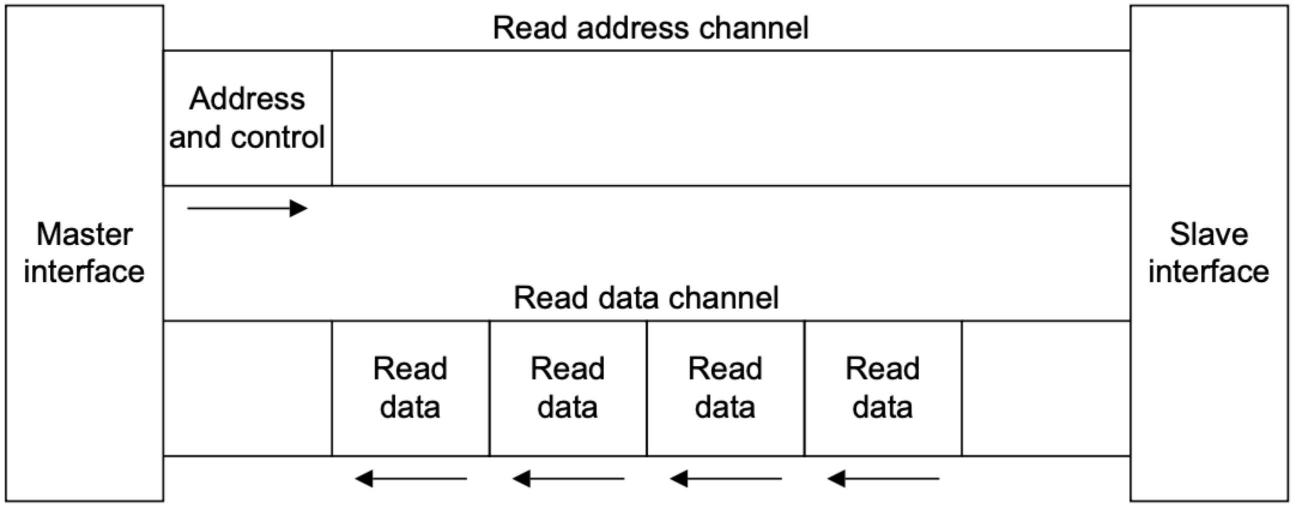
Full AXI and AXI Lite

- Meant for back-and-forth communication
- Request-response type communication
- Memory-mapped interfaces


"What is the value at Addres 0x12345?"
"1986"
M
S
ACLK
ARSTn
From clock source
From synchronous reset source

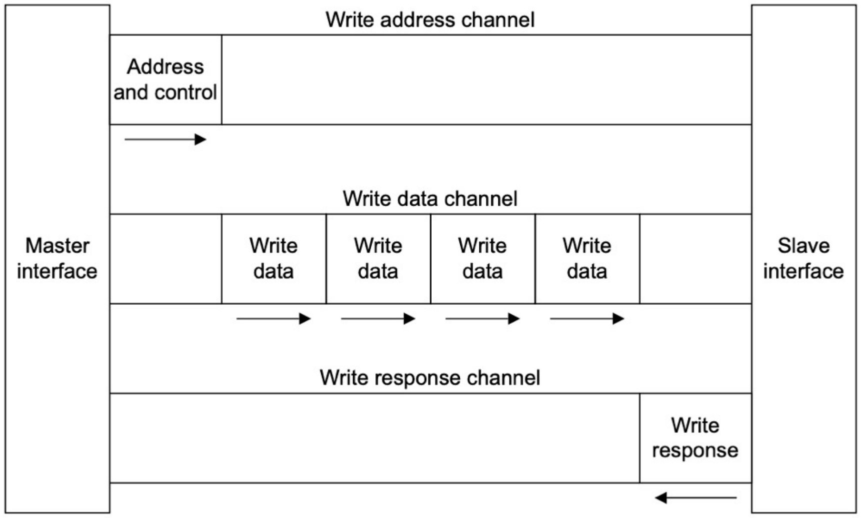
Full AXI and AXI Lite Read

- Will involve multiple channels (Each with their own ready, valid, clock, data path, etc…)
- A Read interface will have two AXI channels:
    - One that transfers address info from Master to Slave
    - One that transfers response data from Slave to Master



Full AXI and AXI Lite Write

- Will involve multiple channels (Each with their own ready, valid, clock, data path, etc…)
- A Write interface will have three AXI channels:
  - One that transfers address info from Master to Slave
  - One that transfers data to write from Master to Slave
  - One that transfers response data from Slave to Master



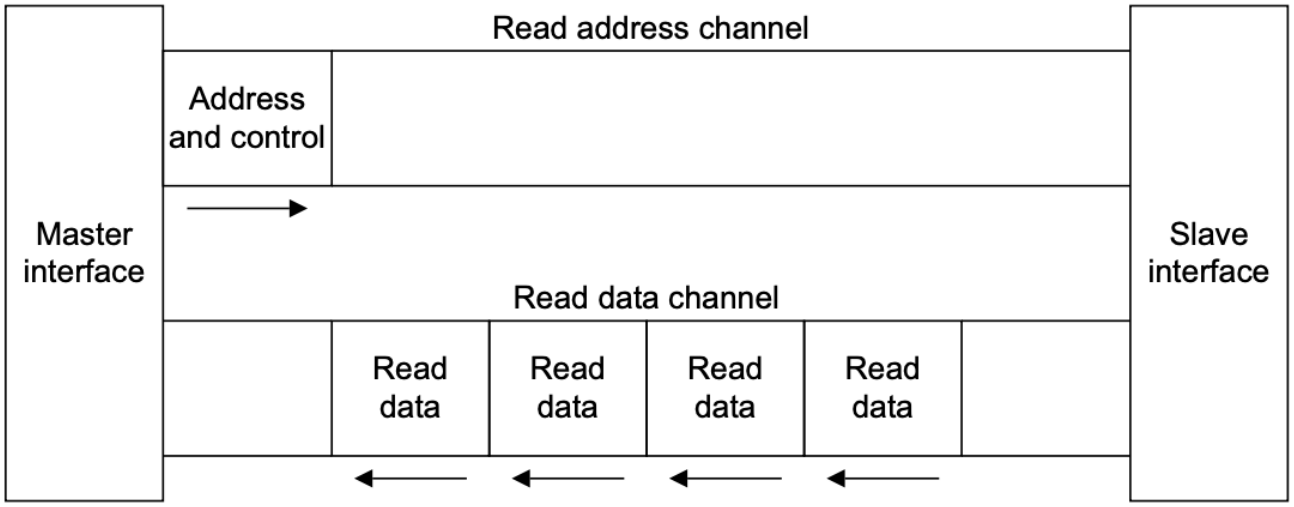
All Channels are AXI

- Then for specific tasks, they can have specific additional signals

- Think of generic AXI as a root class
- The “read address channel” is a subclass of standard AXI

Full AXI and AXI Lite Read

- Will involve multiple channels (Each with their own ready, valid, clock, data path, etc…)
- A Read interface will have two AXI channels:
  - One that transfers address info from Master to Slave
  - One that transfers response data from Slave to Master



Read Address Chanel

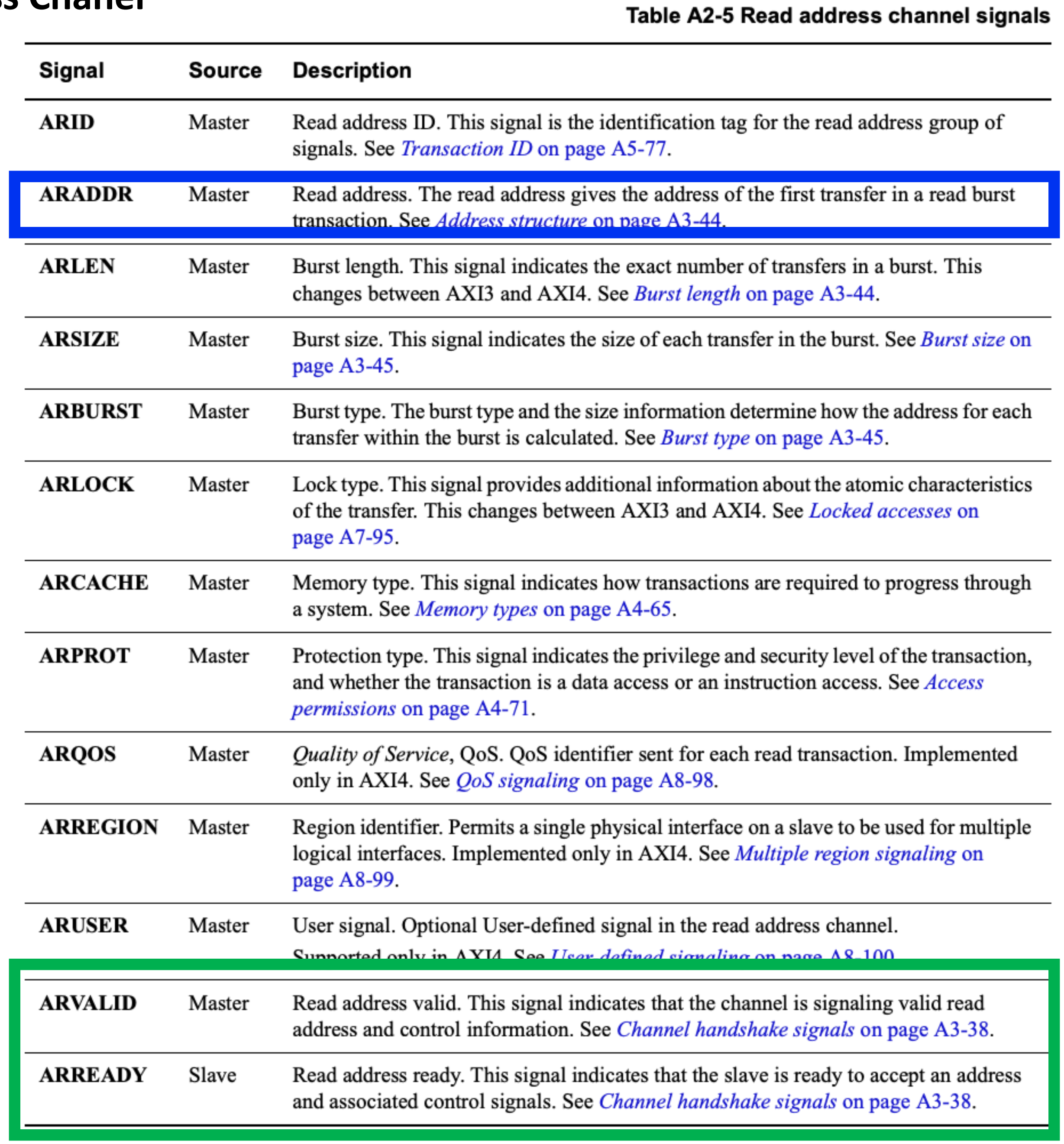


Table A2-5 Read address channel signals

Payload

CORE

Signal	Source	Description
ARID	Master	Read address ID. This signal is the identification tag for the read address group of signals. See *Transaction ID* on page A5-77.
ARADDR	Master	Read address. The read address gives the address of the first transfer in a read burst transaction. See *Address structure* on page A3-44.
ARLEN	Master	Burst length. This signal indicates the exact number of transfers in a burst. This changes between AXI3 and AXI4. See *Burst length* on page A3-44.
ARSIZE	Master	Burst size. This signal indicates the size of each transfer in the burst. See *Burst size* on page A3-45.
ARBURST	Master	Burst type. The burst type and the size information determine how the address for each transfer within the burst is calculated. See *Burst type* on page A3-45.
ARLOCK	Master	Lock type. This signal provides additional information about the atomic characteristics of the transfer. This changes between AXI3 and AXI4. See *Locked accesses* on page A7-95.
ARCACHE	Master	Memory type. This signal indicates how transactions are required to progress through a system. See *Memory types* on page A4-65.
ARPROT	Master	Protection type. This signal indicates the privilege and security level of the transaction, and whether the transaction is a data access or an instruction access. See *Access permissions* on page A4-71.
ARQOS	Master	*Quality of Service*, QoS. QoS identifier sent for each read transaction. Implemented only in AXI4. See *QoS signaling* on page A8-98.
ARREGION	Master	Region identifier. Permits a single physical interface on a slave to be used for multiple logical interfaces. Implemented only in AXI4. See *Multiple region signaling* on page A8-99.
ARUSER	Master	User signal. Optional User-defined signal in the read address channel. Supported only in AXI4. See *User-defined signaling* on page A8-100.
ARVALID	Master	Read address valid. This signal indicates that the channel is signaling valid read address and control information. See *Channel handshake signals* on page A3-38.
ARREADY	Slave	Read address ready. This signal indicates that the slave is ready to accept an address and associated control signals. See *Channel handshake signals* on page A3-38.

The Read Data Channel:

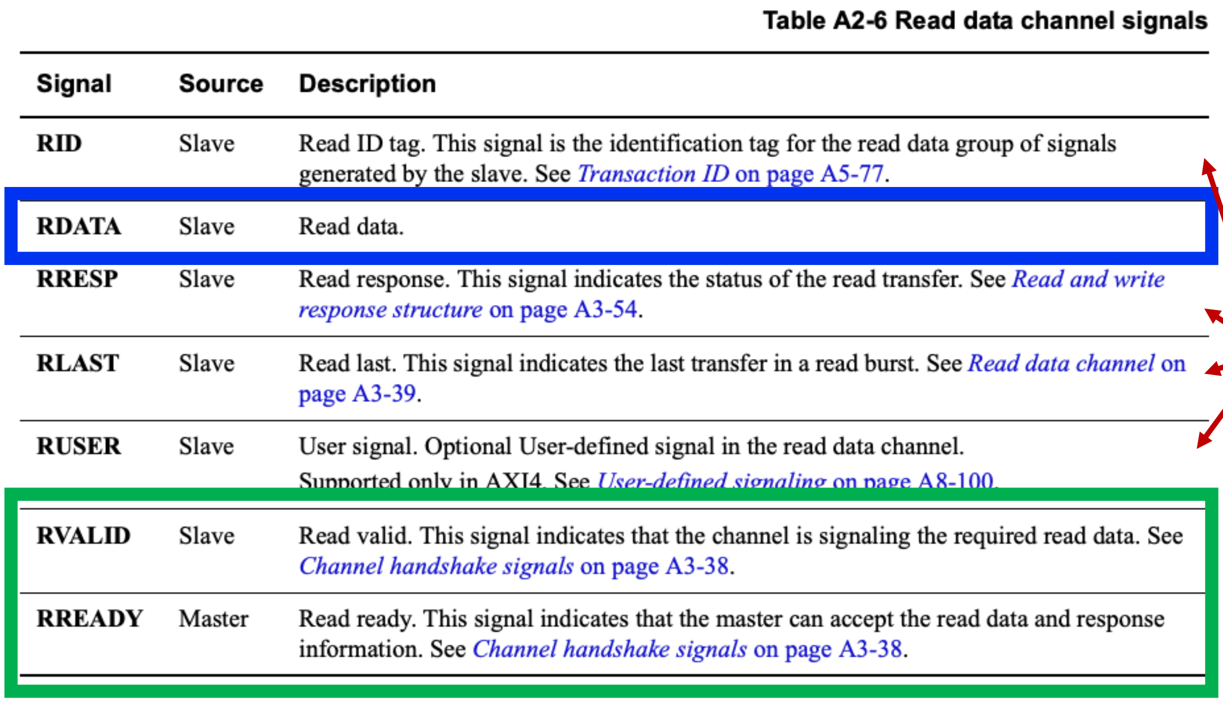


Table A2-6 Read data channel signals

Signal	Source	Description
RID	Slave	Read ID tag. This signal is the identification tag for the read data group of signals generated by the slave. See *Transaction ID* on page A5-77.
RDATA	Slave	Read data.
RRESP	Slave	Read response. This signal indicates the status of the read transfer. See *Read and write response structure* on page A3-54.
RLAST	Slave	Read last. This signal indicates the last transfer in a read burst. See *Read data channel* on page A3-39.
RUSER	Slave	User signal. Optional User-defined signal in the read data channel. Supported only in AXI4. See *User-defined signaling* on page A8-100.
RVALID	Slave	Read valid. This signal indicates that the channel is signaling the required read data. See *Channel handshake signals* on page A3-38.
RREADY	Master	Read ready. This signal indicates that the master can accept the read data and response information. See *Channel handshake signals* on page A3-38.

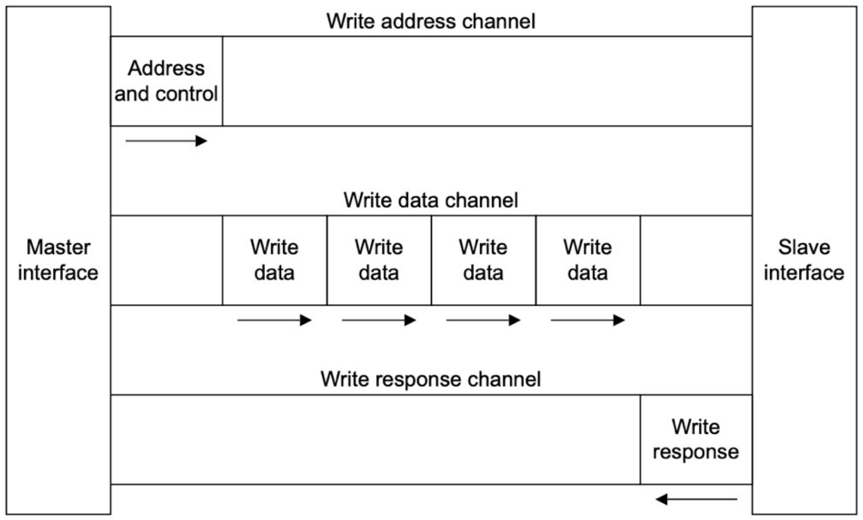
Payload

Supplemental Stuff

CORE

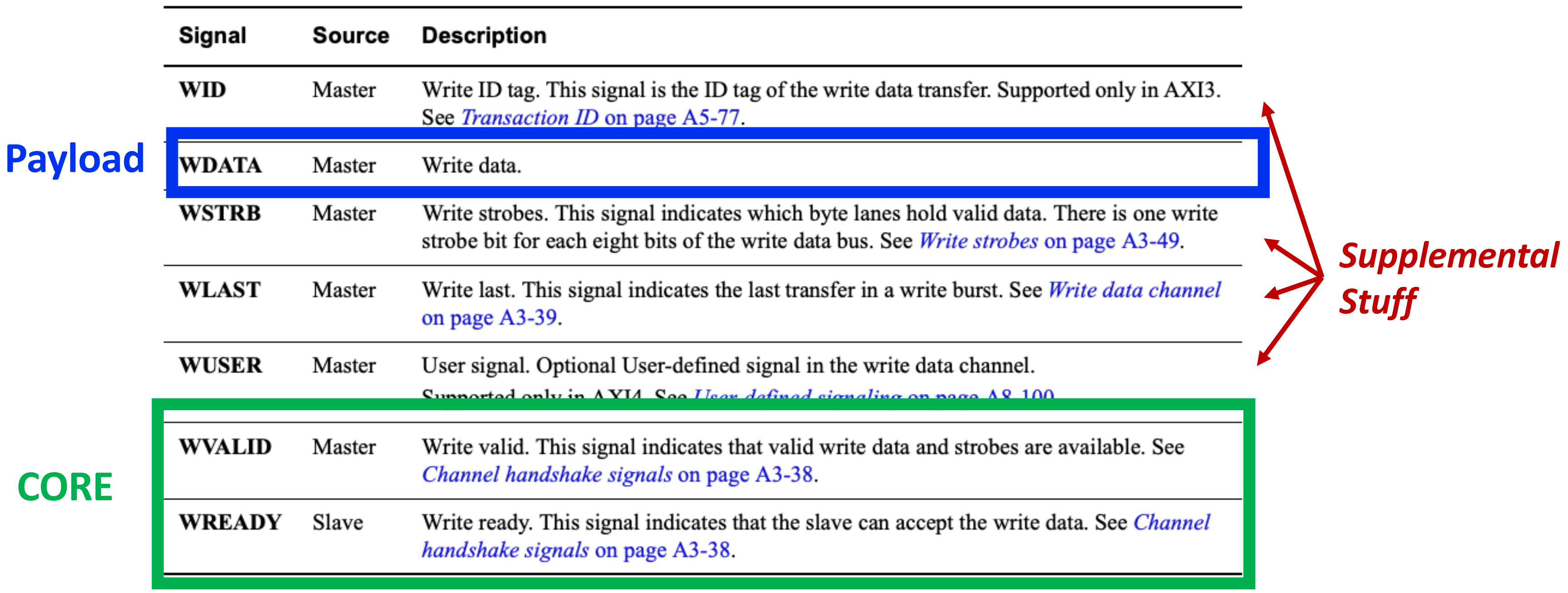
Full AXI and AXI Lite Write

- Will involve multiple channels (Each with their own ready, valid, clock, data path, etc…)
- A Write interface will have three AXI channels:
    - One that transfers address info from Master to Slave
    - One that transfers data to write from Master to Slave
    - One that transfers response data from Slave to Master



Each channel has its own subset of "stuff" that goes along with those core signals shared by all

For example, the Write Data Channel ("W" channel)



Signal	Source	Description
WID	Master	Write ID tag. This signal is the ID tag of the write data transfer. Supported only in AXI3. See *Transaction ID* on page A5-77.
WDATA	Master	Write data.
WSTRB	Master	Write strobes. This signal indicates which byte lanes hold valid data. There is one write strobe bit for each eight bits of the write data bus. See *Write strobes* on page A3-49.
WLAST	Master	Write last. This signal indicates the last transfer in a write burst. See *Write data channel* on page A3-39.
WUSER	Master	User signal. Optional User-defined signal in the write data channel. Supported only in AXI4. See *User-defined signaling* on page A8-100.
WVALID	Master	Write valid. This signal indicates that valid write data and strobes are available. See *Channel handshake signals* on page A3-38.
WREADY	Slave	Write ready. This signal indicates that the slave can accept the write data. See *Channel handshake signals* on page A3-38.

Write Address Channel

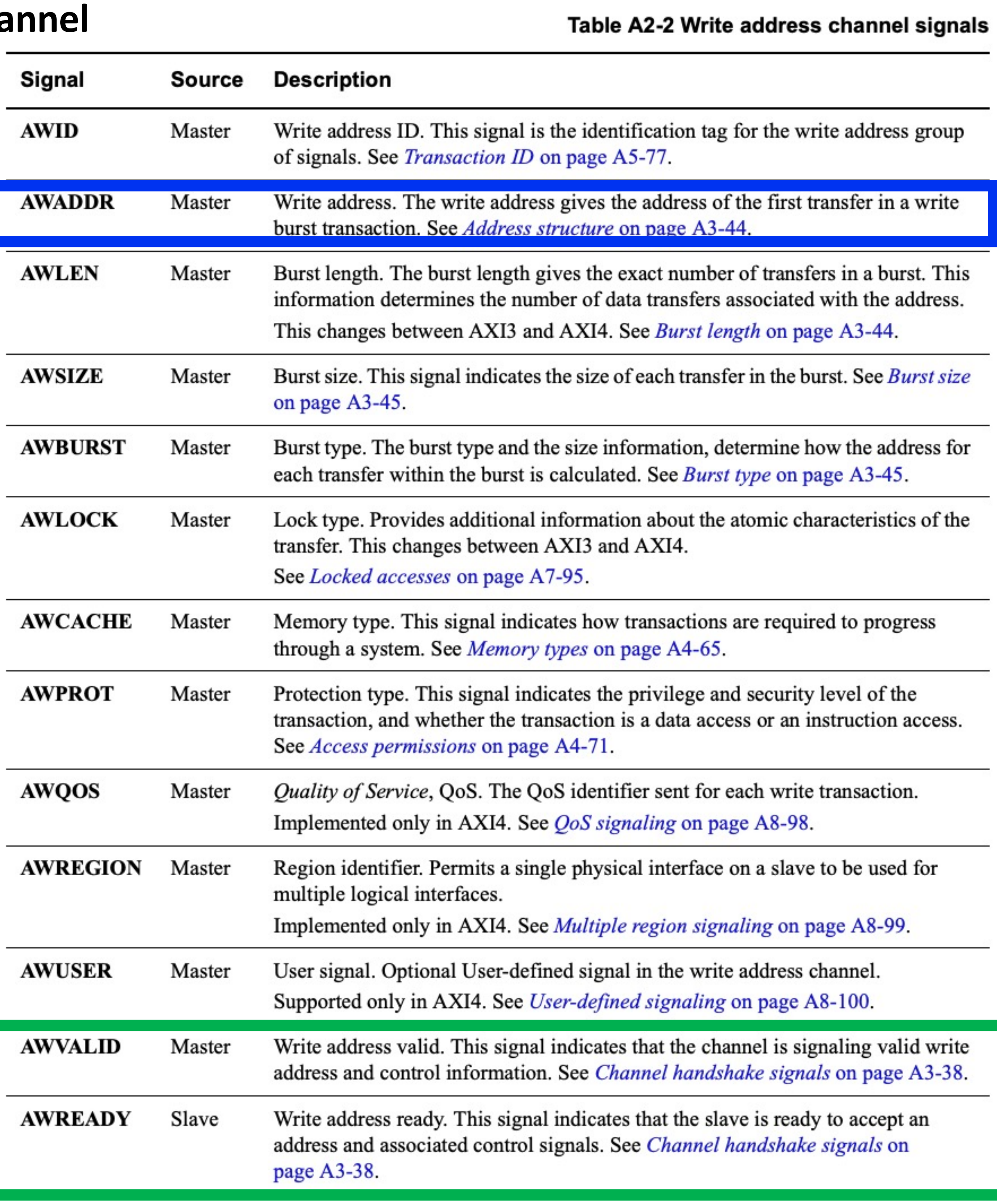


Table A2-2 Write address channel signals

Signal	Source	Description
AWID	Master	Write address ID. This signal is the identification tag for the write address group of signals. See *Transaction ID* on page A5-77.
AWADDR	Master	Write address. The write address gives the address of the first transfer in a write burst transaction. See *Address structure* on page A3-44.
AWLEN	Master	Burst length. The burst length gives the exact number of transfers in a burst. This information determines the number of data transfers associated with the address. This changes between AXI3 and AXI4. See *Burst length* on page A3-44.
AWSIZE	Master	Burst size. This signal indicates the size of each transfer in the burst. See *Burst size* on page A3-45.
AWBURST	Master	Burst type. The burst type and the size information, determine how the address for each transfer within the burst is calculated. See *Burst type* on page A3-45.
AWLOCK	Master	Lock type. Provides additional information about the atomic characteristics of the transfer. This changes between AXI3 and AXI4. See *Locked accesses* on page A7-95.
AWCACHE	Master	Memory type. This signal indicates how transactions are required to progress through a system. See *Memory types* on page A4-65.
AWPROT	Master	Protection type. This signal indicates the privilege and security level of the transaction, and whether the transaction is a data access or an instruction access. See *Access permissions* on page A4-71.
AWQOS	Master	*Quality of Service*, QoS. The QoS identifier sent for each write transaction. Implemented only in AXI4. See *QoS signaling* on page A8-98.
AWREGION	Master	Region identifier. Permits a single physical interface on a slave to be used for multiple logical interfaces. Implemented only in AXI4. See *Multiple region signaling* on page A8-99.
AWUSER	Master	User signal. Optional User-defined signal in the write address channel. Supported only in AXI4. See *User-defined signaling* on page A8-100.
AWVALID	Master	Write address valid. This signal indicates that the channel is signaling valid write address and control information. See *Channel handshake signals* on page A3-38.
AWREADY	Slave	Write address ready. This signal indicates that the slave is ready to accept an address and associated control signals. See *Channel handshake signals* on page A3-38.

Payload

CORE

Write Response

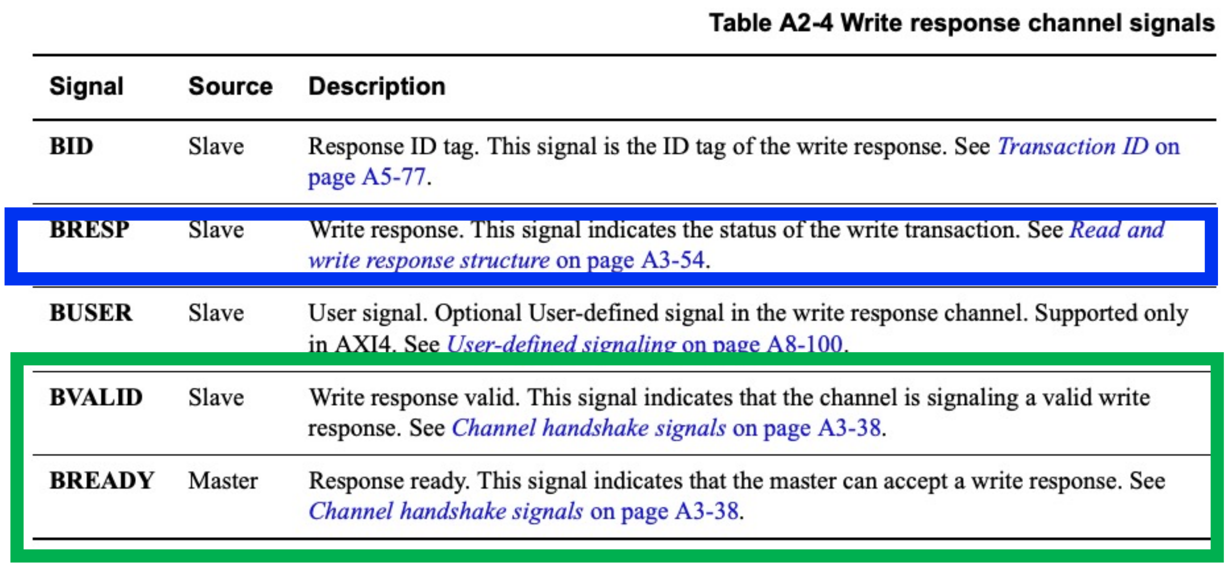


Table A2-4 Write response channel signals

Signal	Source	Description
BID	Slave	Response ID tag. This signal is the ID tag of the write response. See *Transaction ID* on page A5-77.
BRESP	Slave	Write response. This signal indicates the status of the write transaction. See *Read and write response structure* on page A3-54.
BUSER	Slave	User signal. Optional User-defined signal in the write response channel. Supported only in AXI4. See *User-defined signaling* on page A8-100.
BVALID	Slave	Write response valid. This signal indicates that the channel is signaling a valid write response. See *Channel handshake signals* on page A3-38.
BREADY	Master	Response ready. This signal indicates that the master can accept a write response. See *Channel handshake signals* on page A3-38.

Payload

CORE

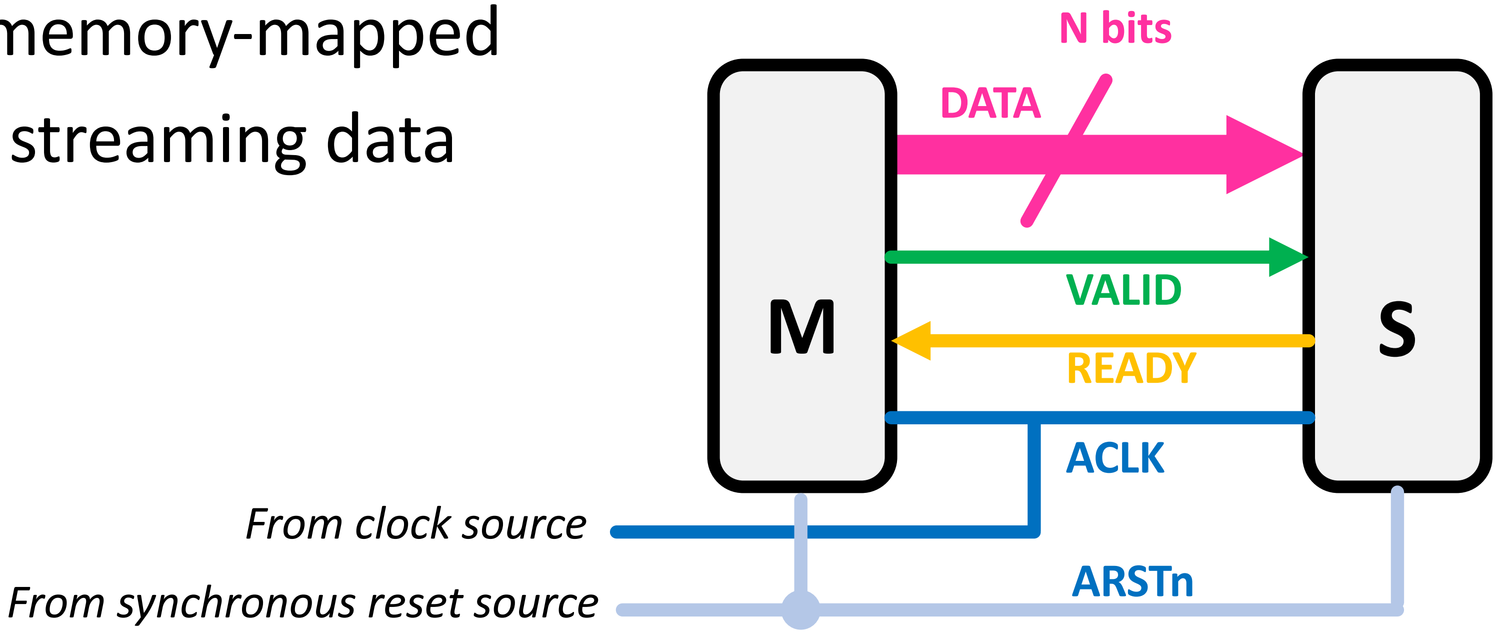
Three General Flavors of AXI4

- **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

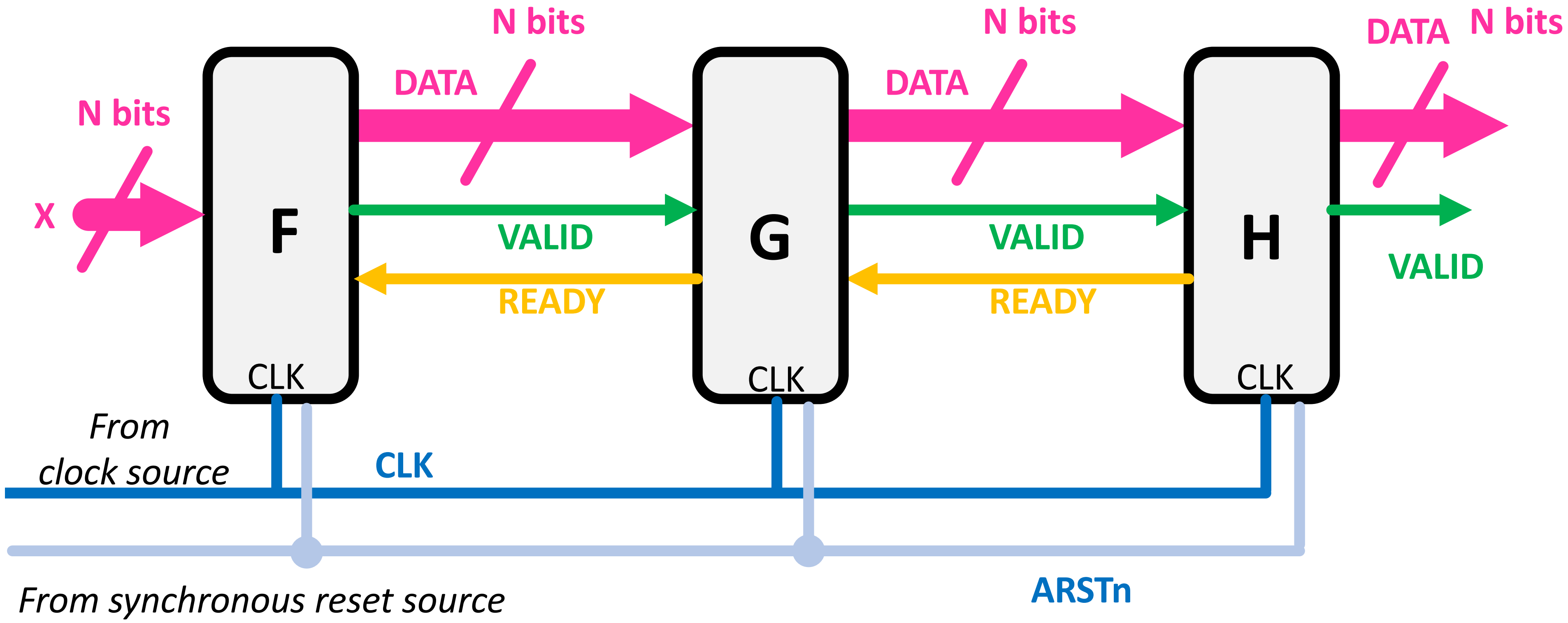
In a AXI Streaming Situation

- Uni-Directional Movement of data
- No call-response
- No memory-mapped
- Just streaming data



AXI Stream

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



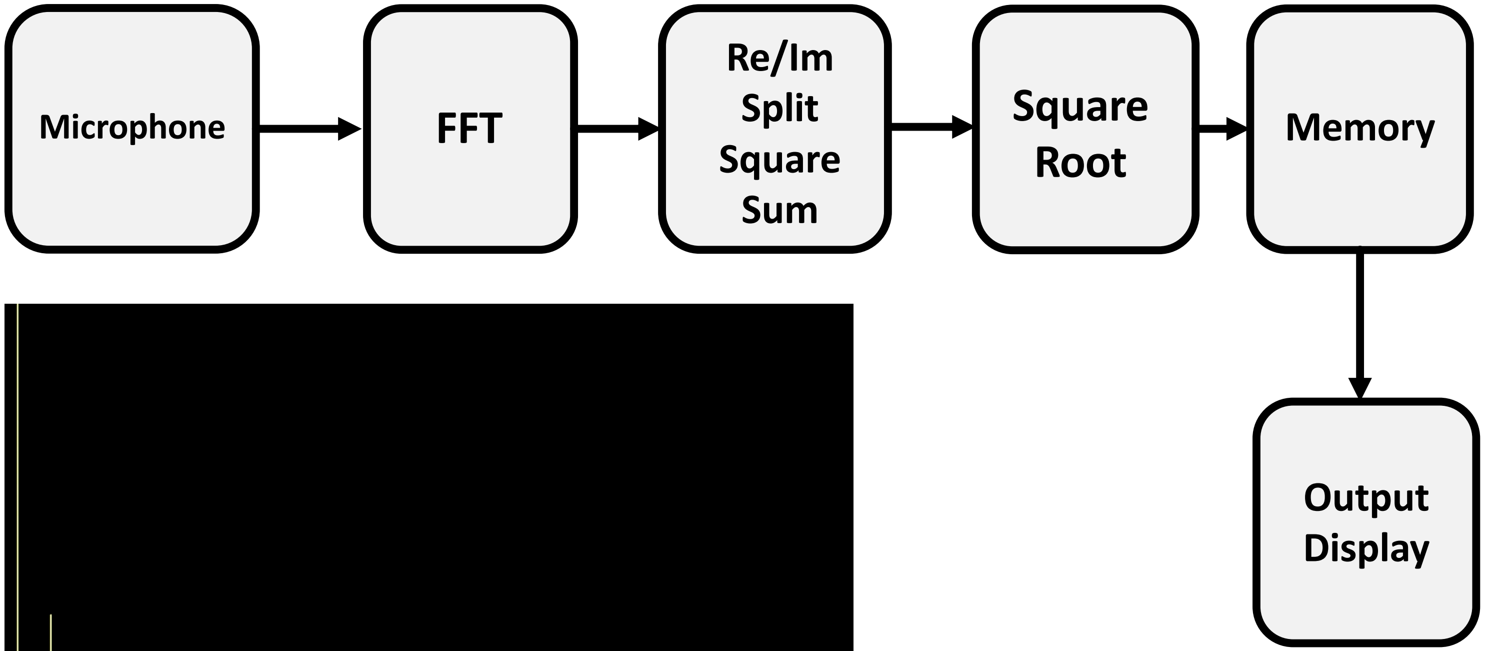
Complexity

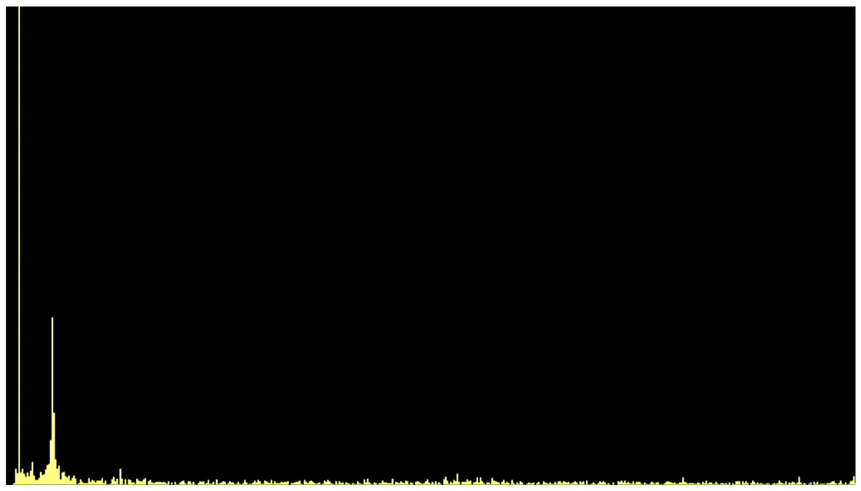
- In terms of wires and options, Full-AXI is the most complex
- AXI-LITE has a lot less options (single data beat so all the supplemental stuff that specifies burst characteristics gets skipped)
- AXI-STREAM has even less…basically a high-speed write channel (Few options), but often needs that extra TLAST signal

$$\begin{tikzcd}
\text{Full-AXI4} \arrow[d] \\
\text{AXI-LITE} \arrow[d] \\
\text{AXI-STREAM}
\end{tikzcd}$$

Real-time Audio Spectrograph

- Let’s do an example!!!

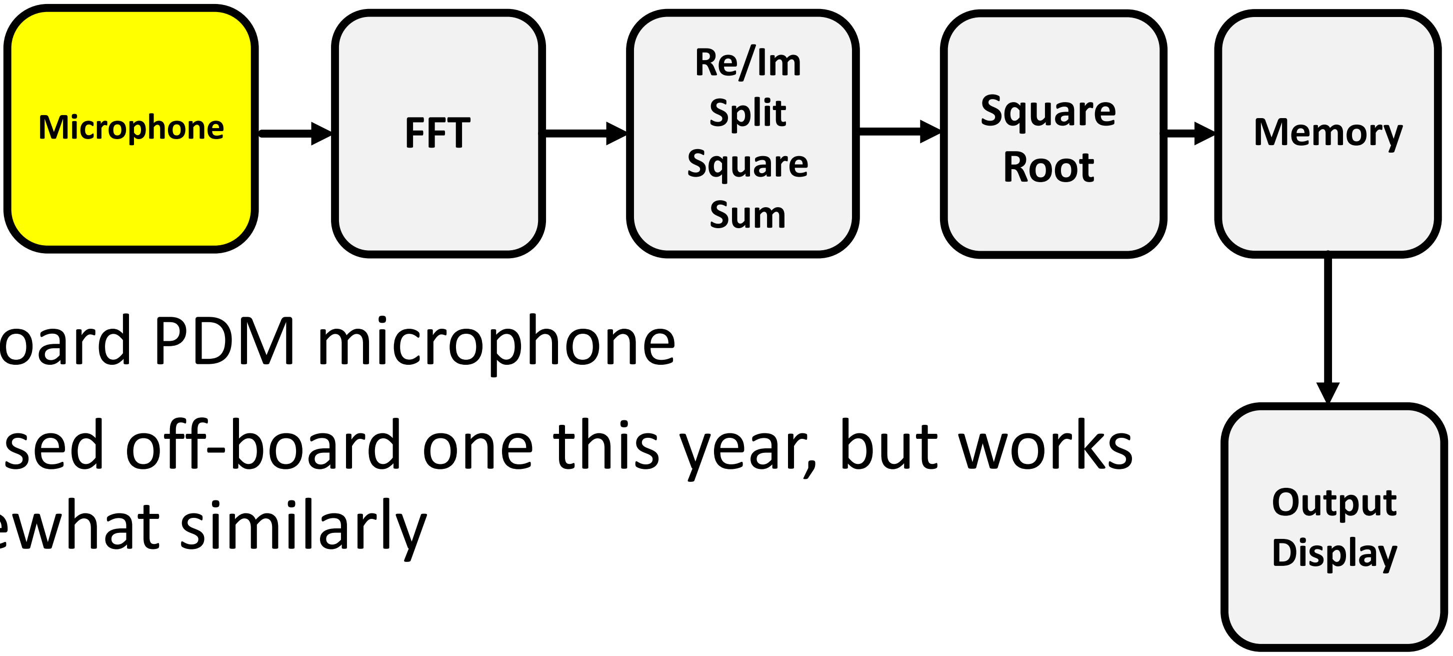




Real-time Audio Spectrograph

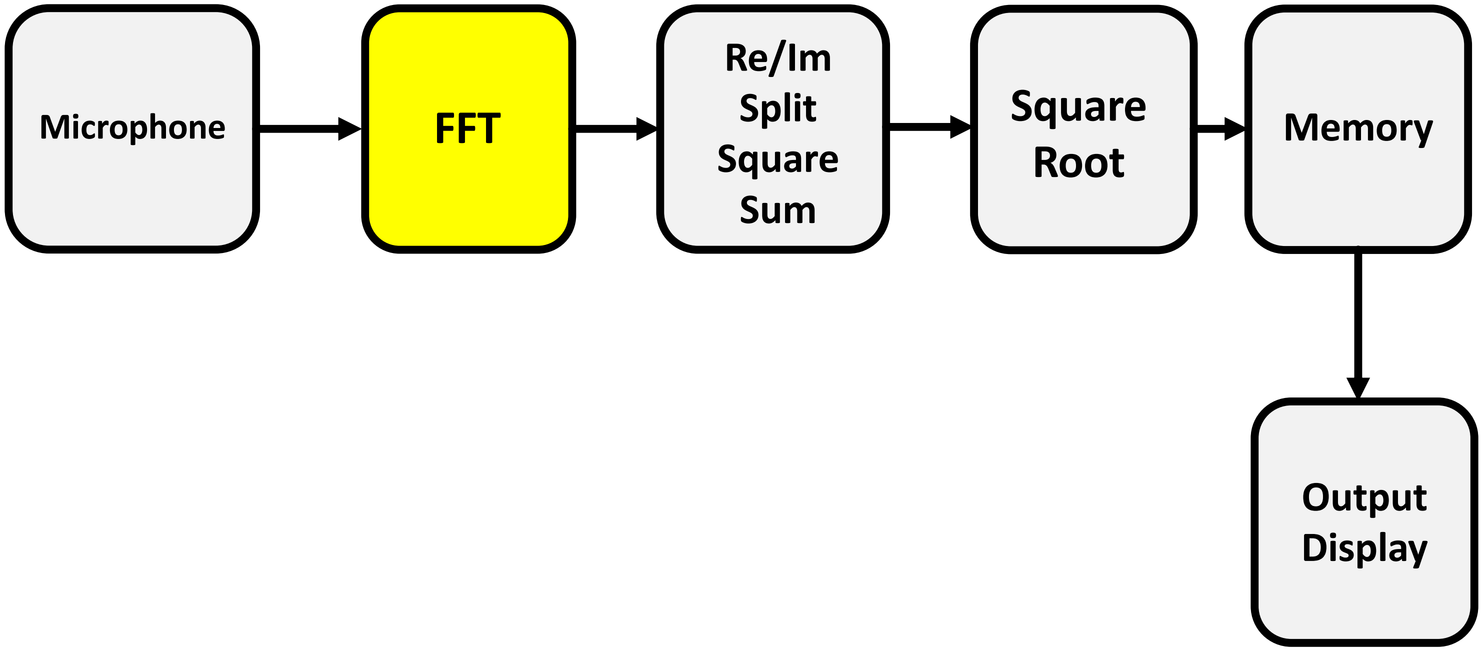
- Collect audio from microphone (use Analog-to-digital Converter)
- Convert time-series data to frequency series
- Take Magnitude of it
- Store it in memory
- Render it on screen as a bargraph
- **RESULT:**
  - Render the energy of the frequency spectrum in real time

Real-time Audio Spectrograph



- On-board PDM microphone
- We used off-board one this year, but works somewhat similarly

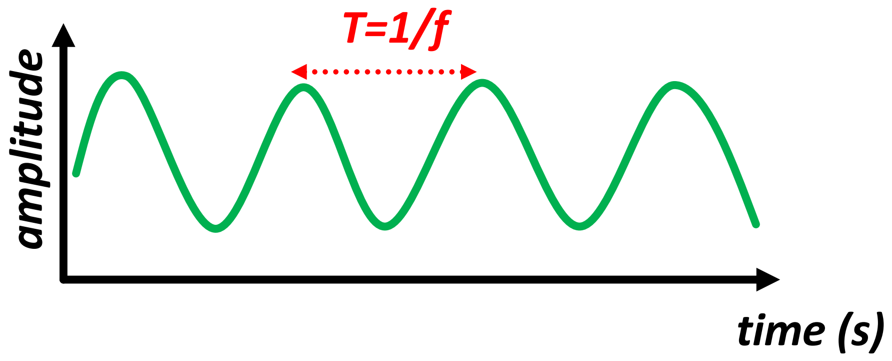
Real-time Audio Spectrograph



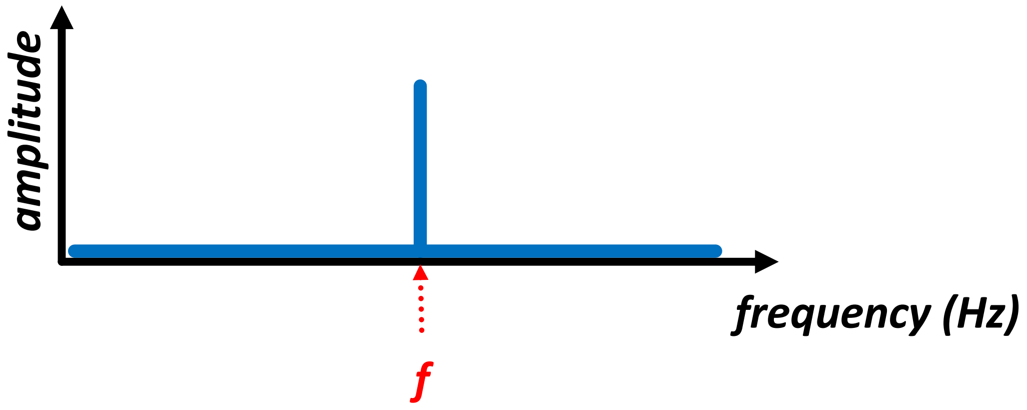
- Computer the Fourier Transform of a Time Series of audio measurements and do so in real time

Fourier Transform

- Convert a time-domain signal:



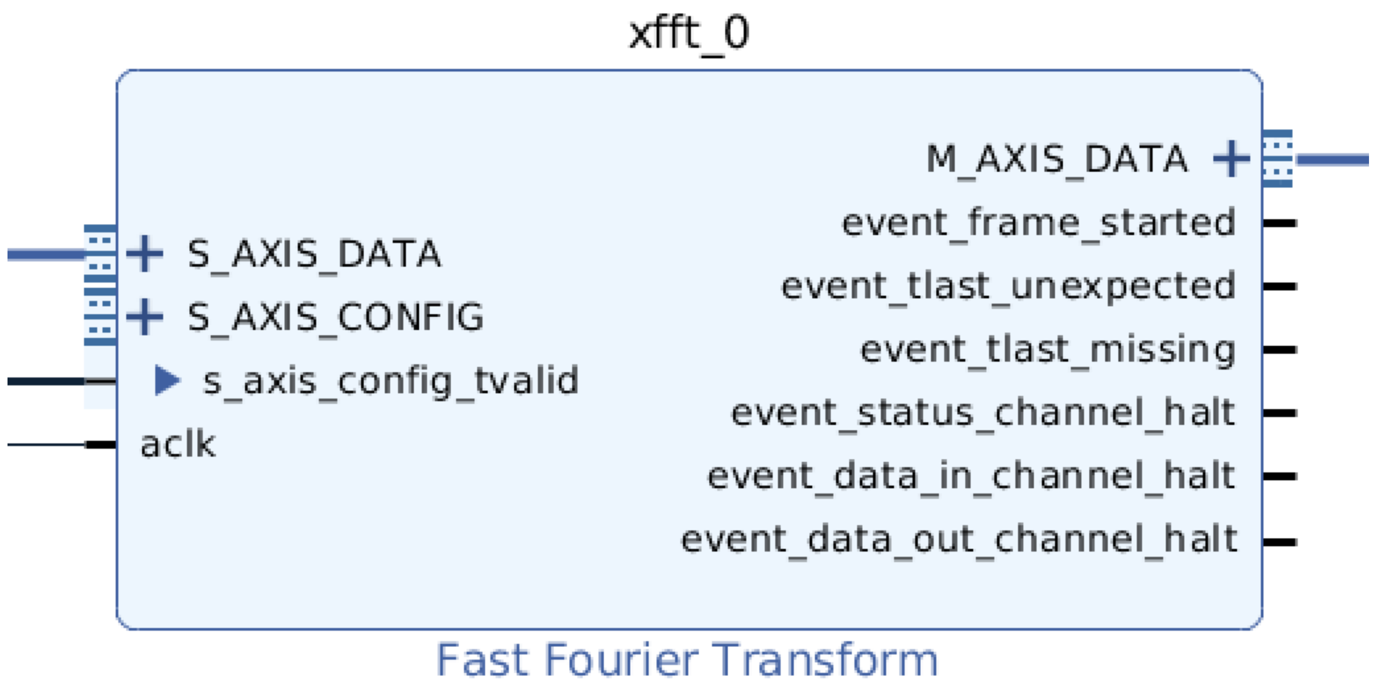
- Into its frequency domain representation:



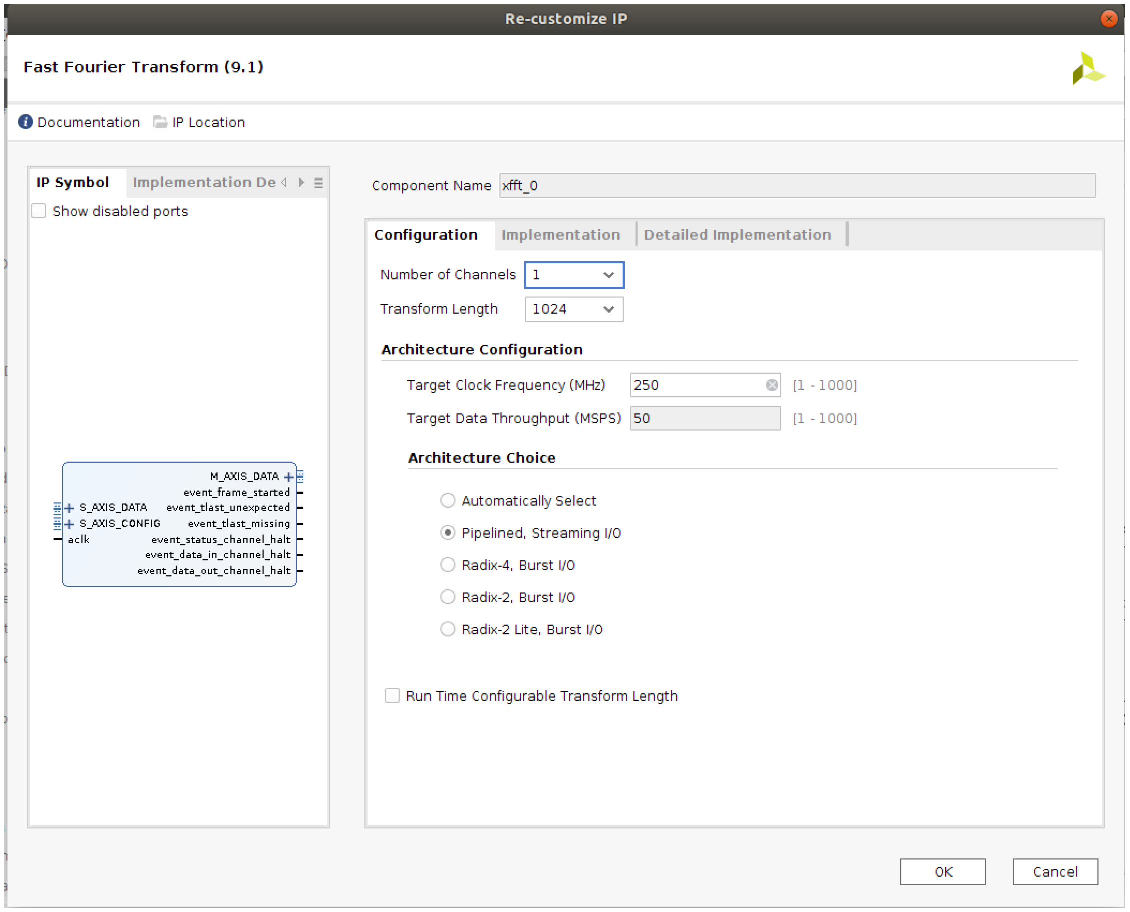
Fast Fourier Transform

- A computationally efficient means of generating the Fourier Transform
- We'll do a 2048 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



Re-customize IP

Fast Fourier Transform (9.1)

Documentation IP Location

IP Symbol | Implementation De

Show disabled ports

Component Name xfft_0

Configuration | Implementation | Detailed Implementation

Number of Channels 1

Transform Length 1024

Architecture Configuration

Target Clock Frequency (MHz) 250 [1 - 1000]

Target Data Throughput (MSPS) 50 [1 - 1000]

Architecture Choice

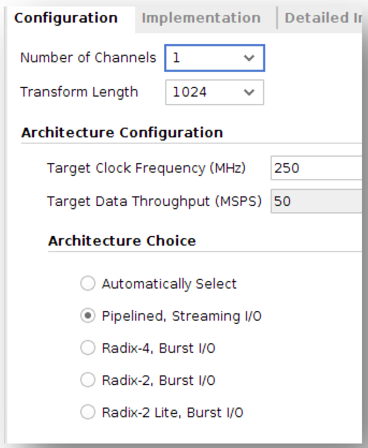
- Automatically Select
- [✓ Pipelined, Streaming I/O]
- Radix-4, Burst I/O
- Radix-2, Burst I/O
- Radix-2 Lite, Burst I/O

Run Time Configurable Transform Length

OK Cancel

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

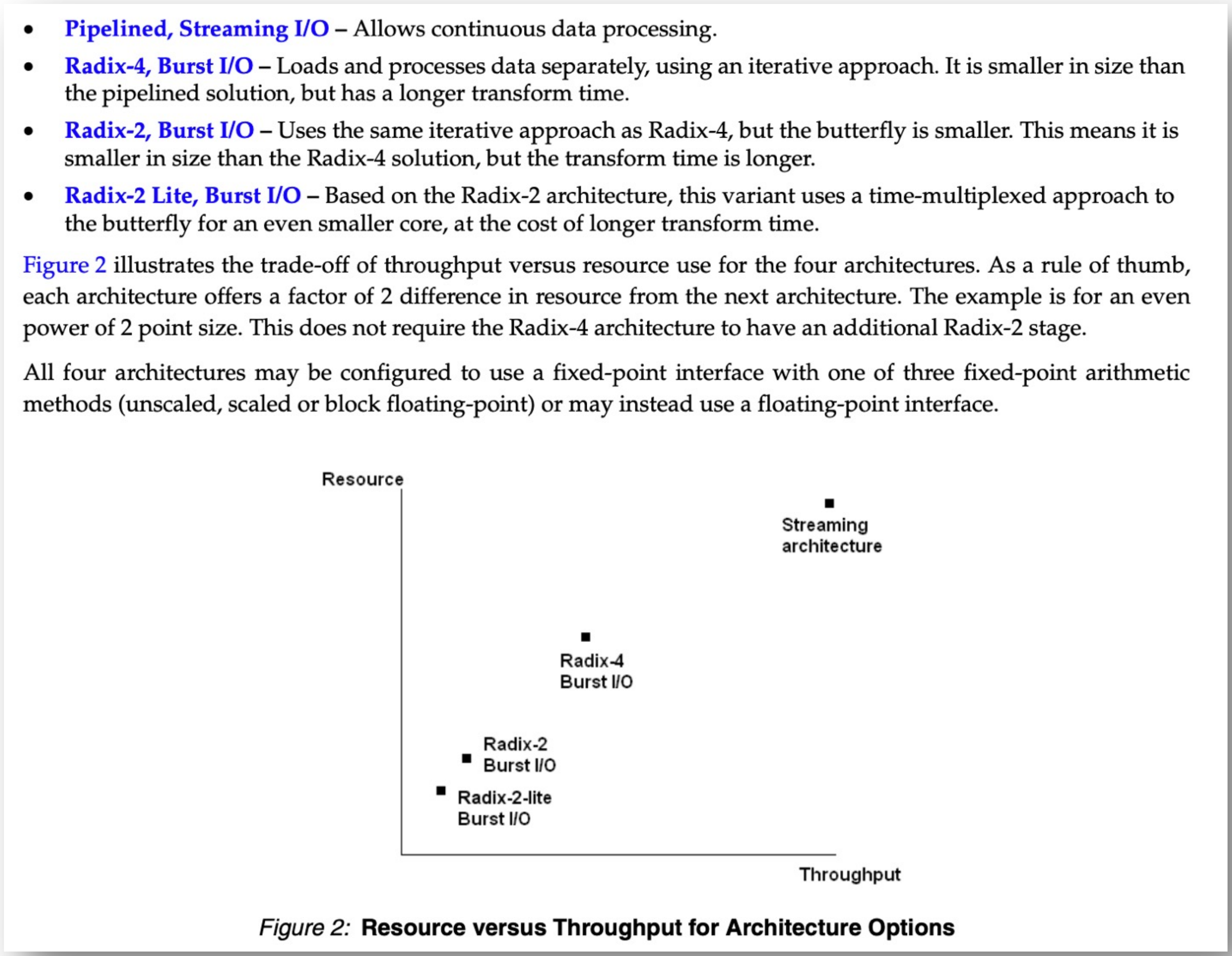
Architecture Configuration

Target Clock Frequency (MHz): 250

Target Data Throughput (MSPS): 50

Architecture Choice

- ( ) Automatically Select
- [✓ Pipelined, Streaming I/O]
- ( ) 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.

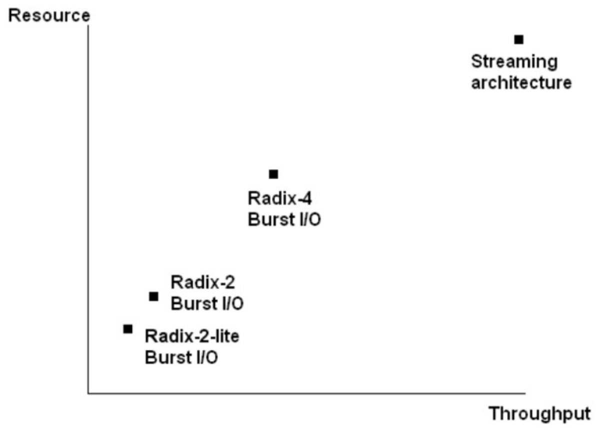
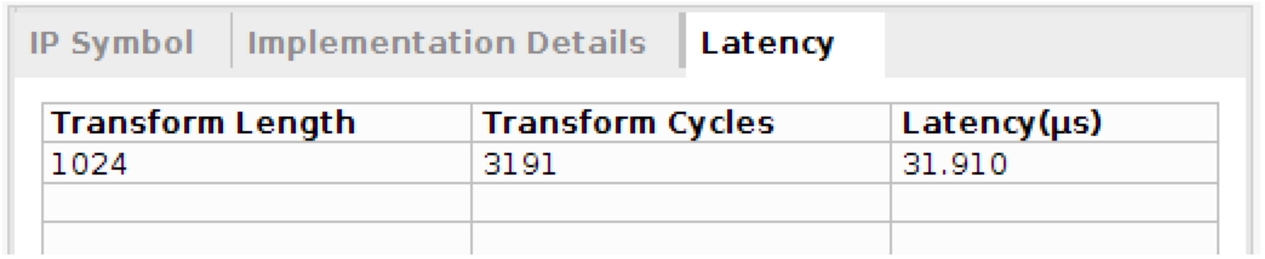


Figure 2: **Resource versus Throughput for Architecture Options**

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

FFT Latency

1024 FFT on 100 MHz clock…

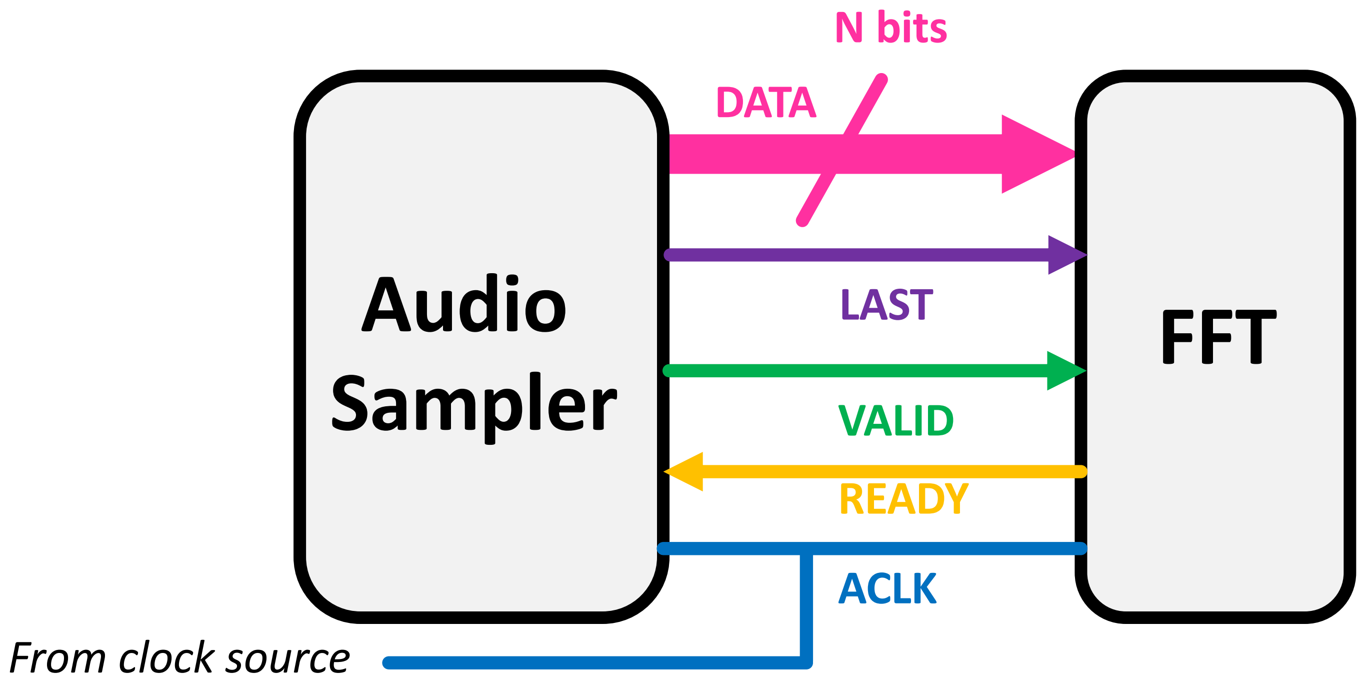


Transform Length	Transform Cycles	Latency(µs)
1024	3191	31.910

- For this year…
- At the clock I ran it: 148.5MHz that is:
  - 6273 clock cycles @ 148.5MHz (42.25 µs)
- Needs all 2048 input samples before it starts outputting

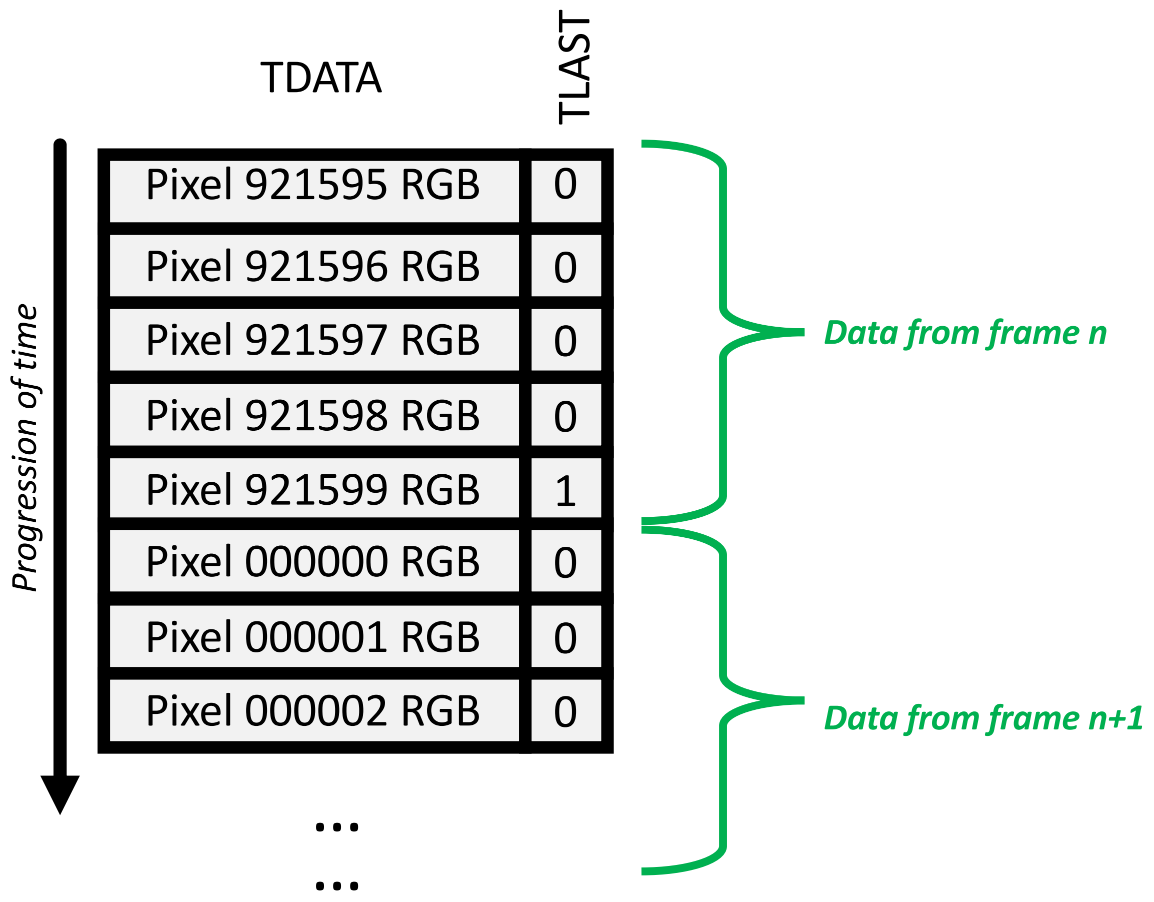
TLAST

- Since we're sending 2048 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)



TLAST is important

- Since data is sent serially, TLAST allows us to know where to place data with respect to other data



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==2047;
  end else begin
    fft_valid = 0;
  end
end
```

FFT Instance:

```
xfft_0 my_fft (.aclk(clk_100mhz), .s_axis_data_tdata(fft_data),
                .s_axis_data_tvalid(fft_valid),
                .s_axis_data_tlast(fft_last), .s_axis_data_tready(fft_ready),
                .s_axis_config_tdata(0),
                 .s_axis_config_tvalid(0),
                 .s_axis_config_tready(),
                .m_axis_data_tdata(fft_out_data), .m_axis_data_tvalid(fft_out_valid),
                .m_axis_data_tlast(fft_out_last), .m_axis_data_tready(fft_out_ready));
```

Already “breaking” AXI

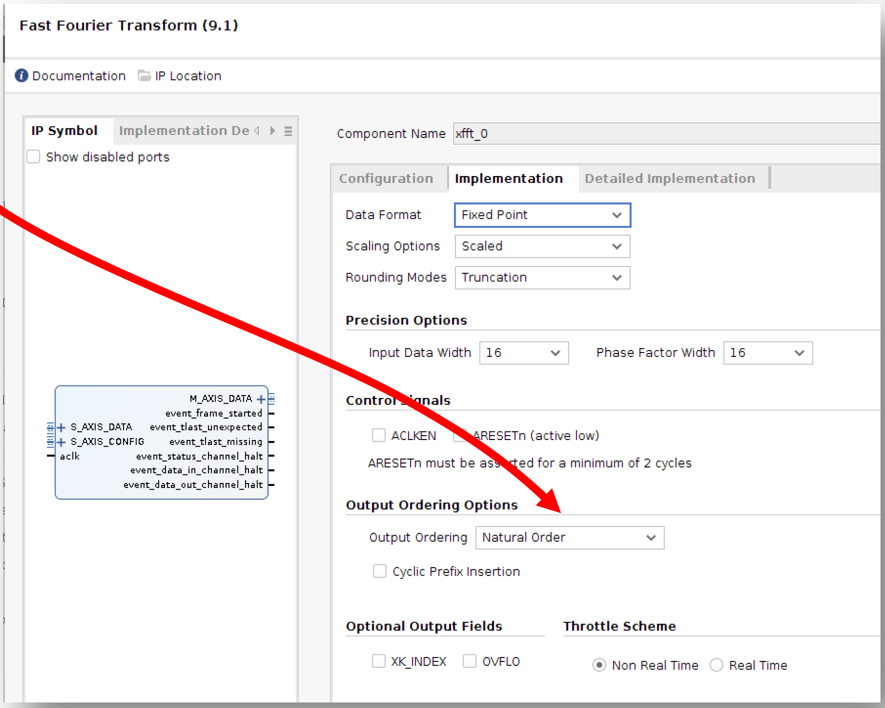
- This code is not monitoring whether the FFT is **READY**.
- Realistically we are generating data so slowly that this will never actually matter (discuss at end)
- Also we’re not storing this data anywhere

```
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==2047;
  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.



Fast Fourier Transform (9.1)

Documentation   IP Location

Show disabled ports

Component Name: xfft_0

Configuration | **Implementation** | Detailed Implementation

Data Format: Fixed Point
Scaling Options: Scaled
Rounding Modes: Truncation

Precision Options

Input Data Width: 16   Phase Factor Width: 16

Control Signals

☐ ACLKEN   ☐ ARESETn (active low)

ARESETn must be asserted for a minimum of 2 cycles

Output Ordering Options

Output Ordering: Natural Order

☐ Cyclic Prefix Insertion

Optional Output Fields

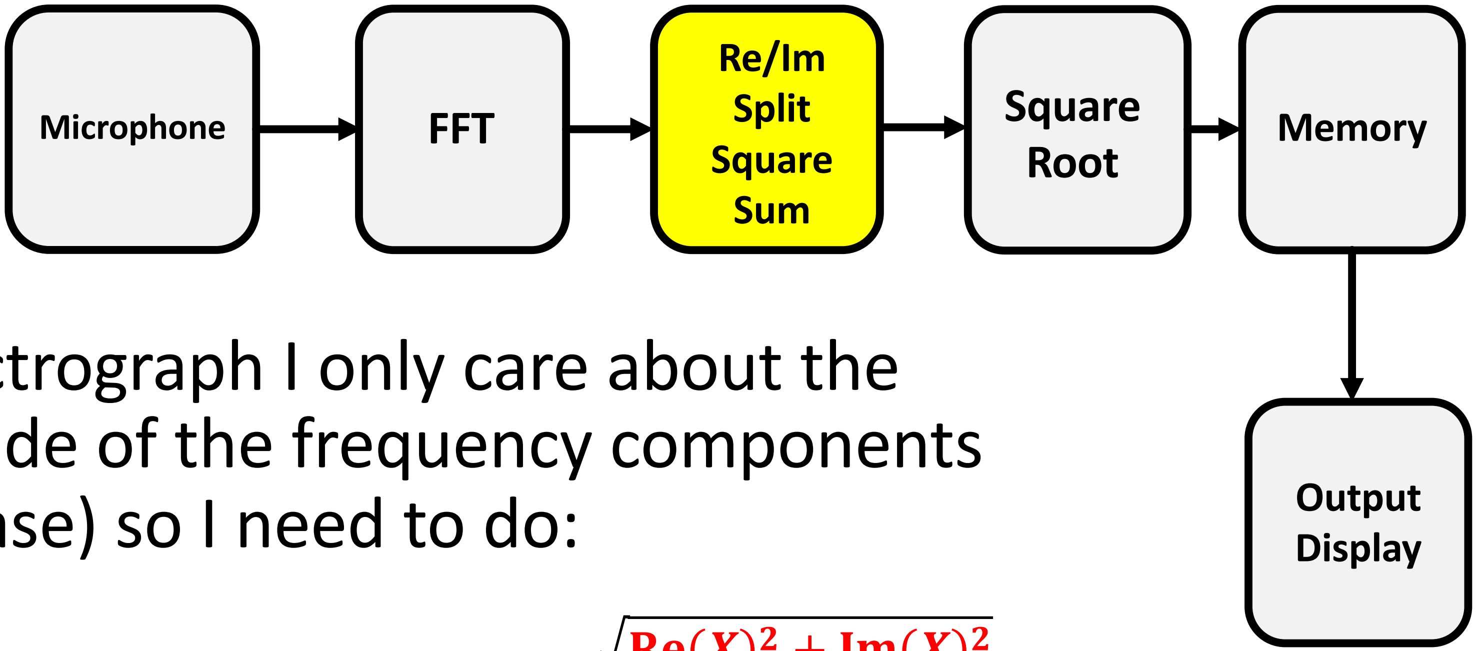
☐ XK_INDEX   ☐ OVFLO

Throttle Scheme

● Non Real Time   ○ Real Time

Real-time Audio Spectrograph

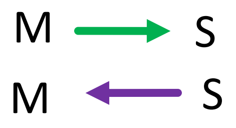
- FFT outputs 32 bits of a complex number:
  - 16 bits real component
  - 16 bits imaginary component

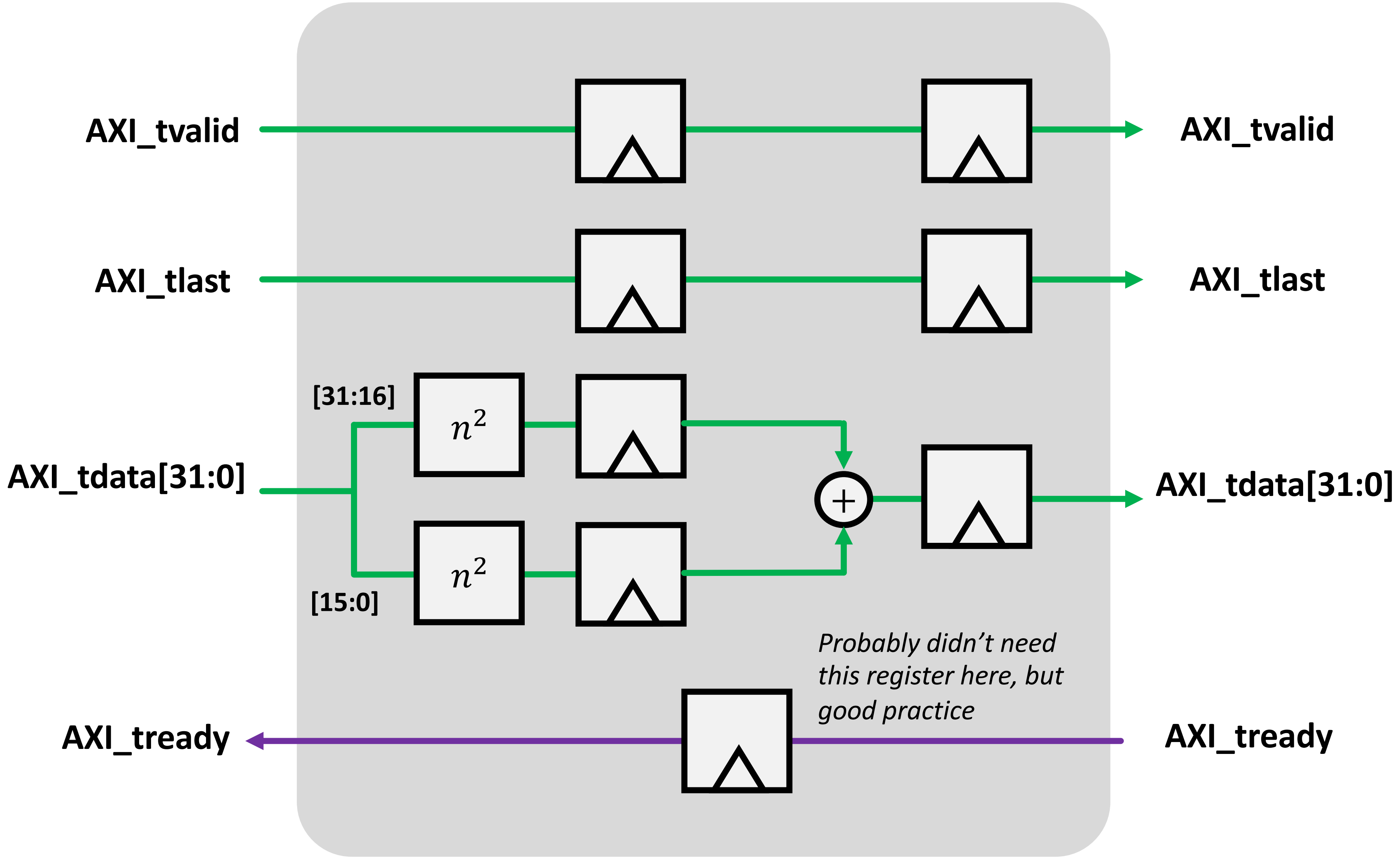


For spectrograph I only care about the magnitude of the frequency components (not phase) so I need to do:

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

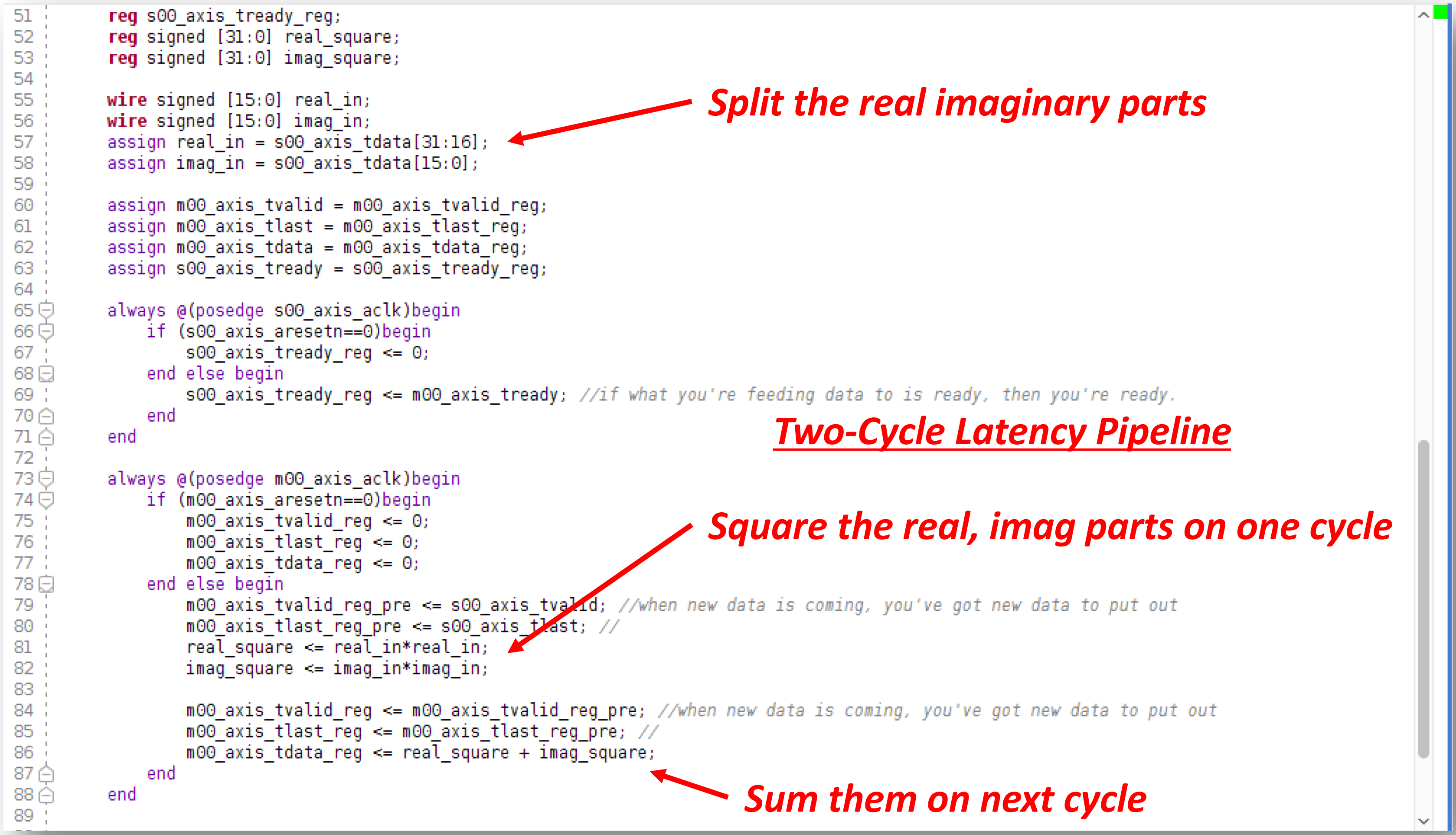
Split→Square→Sum





Probably didn't need this register here, but good practice

Split→Square→Sum



```
    reg s00_axis_tready_reg;
    reg signed [31:0] real_square;
    reg signed [31:0] imag_square;

    wire signed [15:0] real_in;
    wire signed [15:0] imag_in;
    assign real_in = s00_axis_tdata[31:16];
    assign imag_in = s00_axis_tdata[15:0];

    assign m00_axis_tvalid = m00_axis_tvalid_reg;
    assign m00_axis_tlast = m00_axis_tlast_reg;
    assign m00_axis_tdata = m00_axis_tdata_reg;
    assign s00_axis_tready = s00_axis_tready_reg;

    always @(posedge s00_axis_aclk)begin
        if (s00_axis_aresetn==0)begin
            s00_axis_tready_reg <= 0;
        end else begin
            s00_axis_tready_reg <= m00_axis_tready; //if what you're feeding data to is ready, then you're ready.
        end
    end

    always @(posedge m00_axis_aclk)begin
        if (m00_axis_aresetn==0)begin
            m00_axis_tvalid_reg <= 0;
            m00_axis_tlast_reg <= 0;
            m00_axis_tdata_reg <= 0;
        end else begin
            m00_axis_tvalid_reg_pre <= s00_axis_tvalid; //when new data is coming, you've got new data to put out
            m00_axis_tlast_reg_pre <= s00_axis_tlast; //
            real_square <= real_in*real_in;
            imag_square <= imag_in*imag_in;

            m00_axis_tvalid_reg <= m00_axis_tvalid_reg_pre; //when new data is coming, you've got new data to put out
            m00_axis_tlast_reg <= m00_axis_tlast_reg_pre; //
            m00_axis_tdata_reg <= real_square + imag_square;
        end
    end
```

Split the real imaginary parts

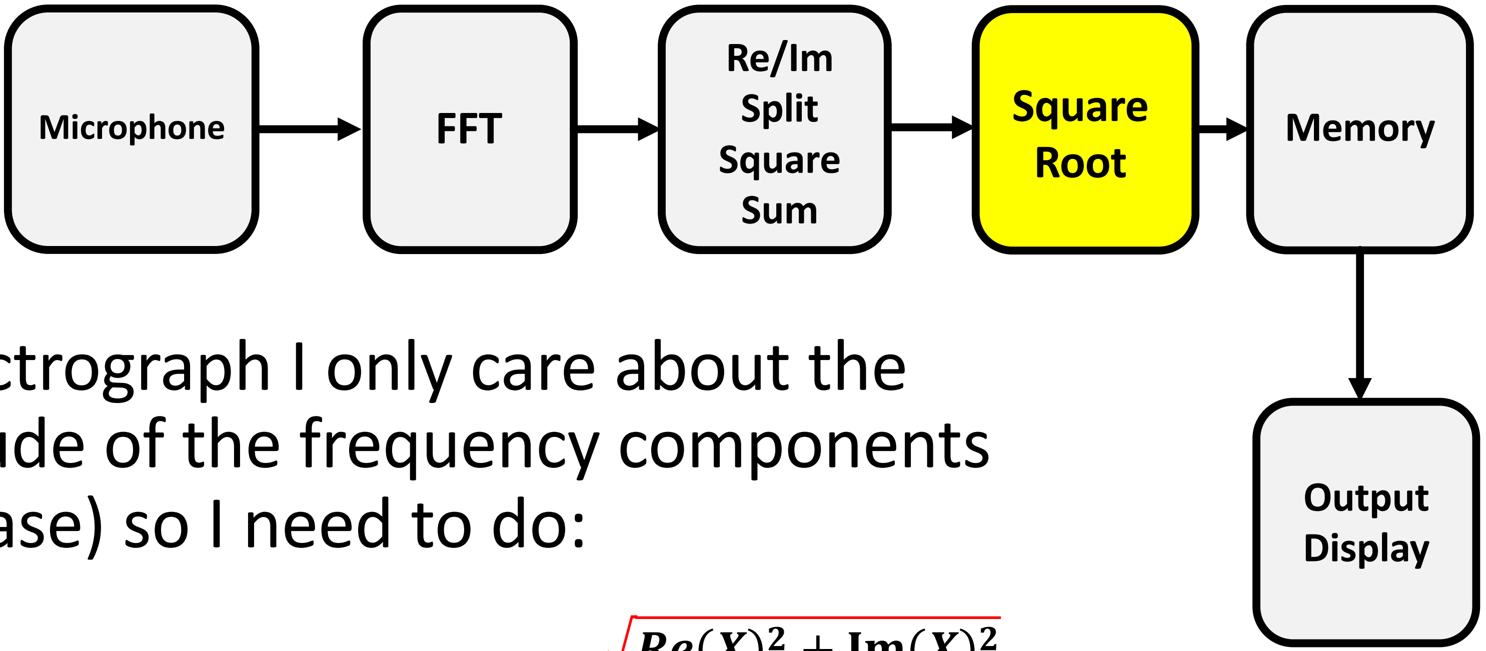
<u>Two-Cycle Latency Pipeline</u>

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

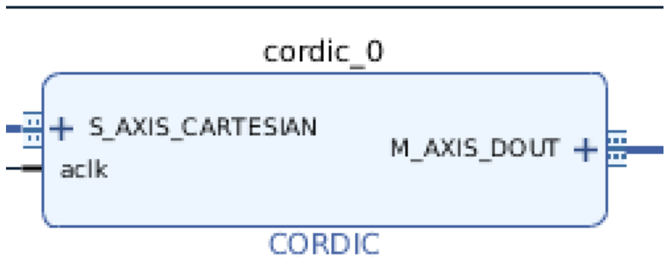


For spectrograph I only care about the magnitude of the frequency components (not phase) so I need to do:

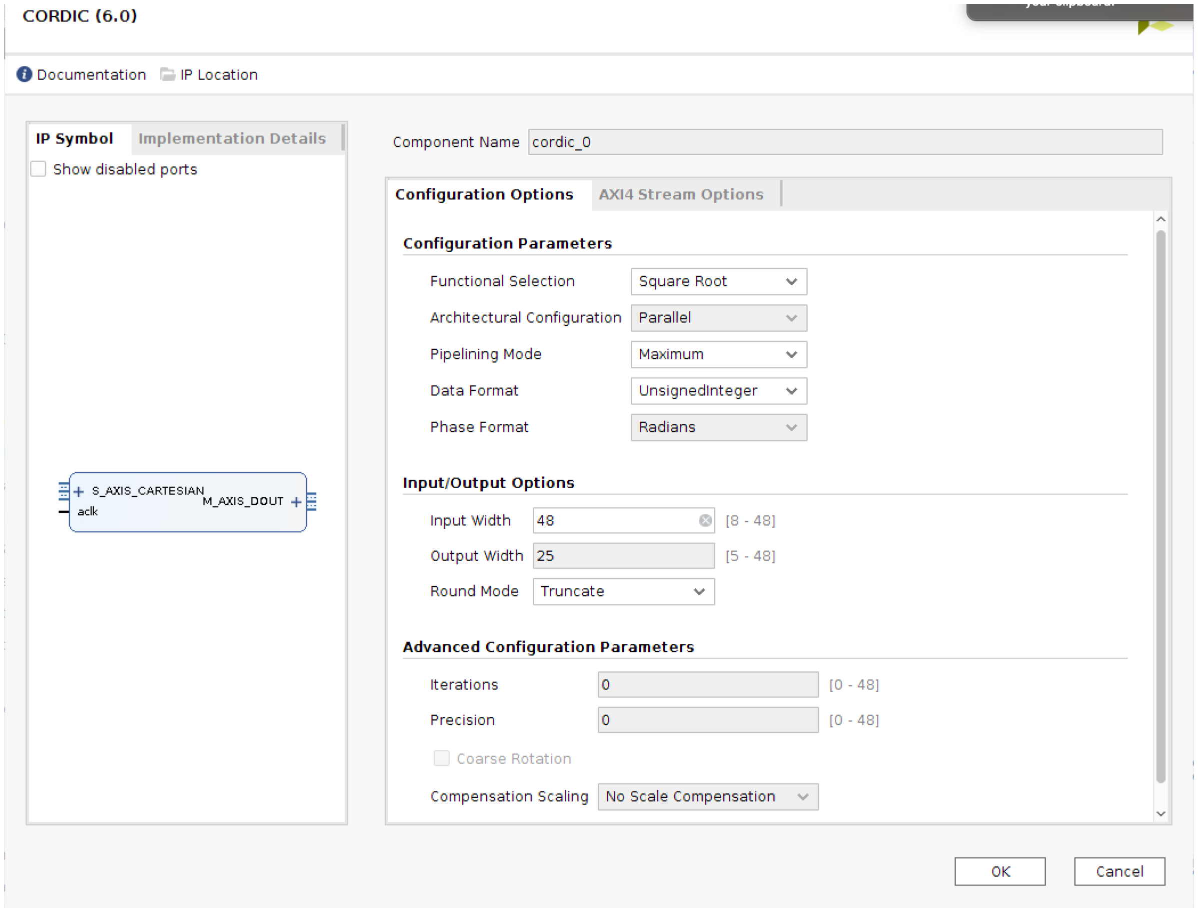
$$\sqrt{Re(X)^2 + \mathrm{Im}(X)^2}$$

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

Component Name: cordic_0

Configuration Options | AXI4 Stream Options

Configuration Parameters

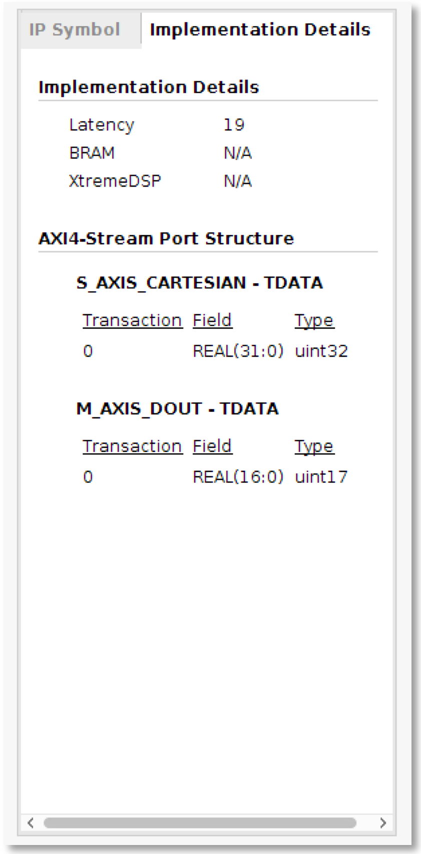
Functional Selection	Square Root
Architectural Configuration	Parallel
Pipelining Mode	Maximum
Data Format	UnsignedInteger
Phase Format	Radians

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	


IP Symbol | Implementation Details

Implementation Details

Latency	19
BRAM	N/A
XtremeDSP	N/A

AXI4-Stream Port Structure

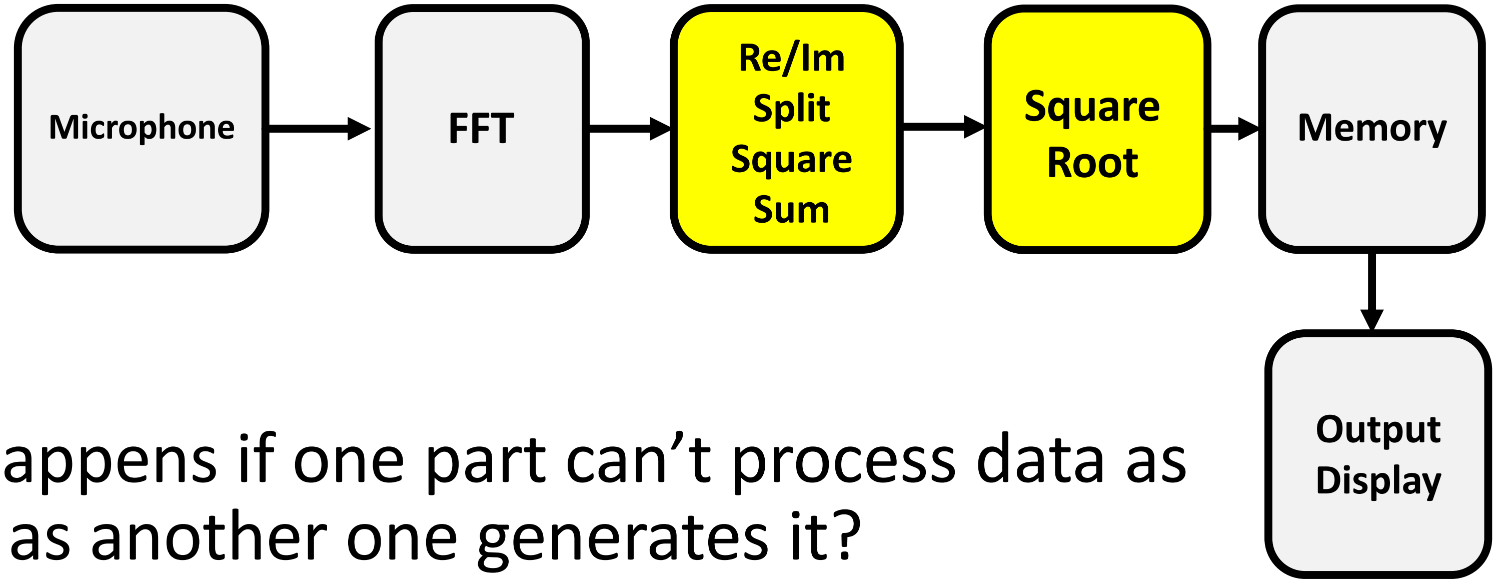
S_AXIS_CARTESIAN - TDATA

Transaction	Field	Type
0	REAL(31:0)	uint32

M_AXIS_DOUT - TDATA

Transaction	Field	Type
0	REAL(16:0)	uint17

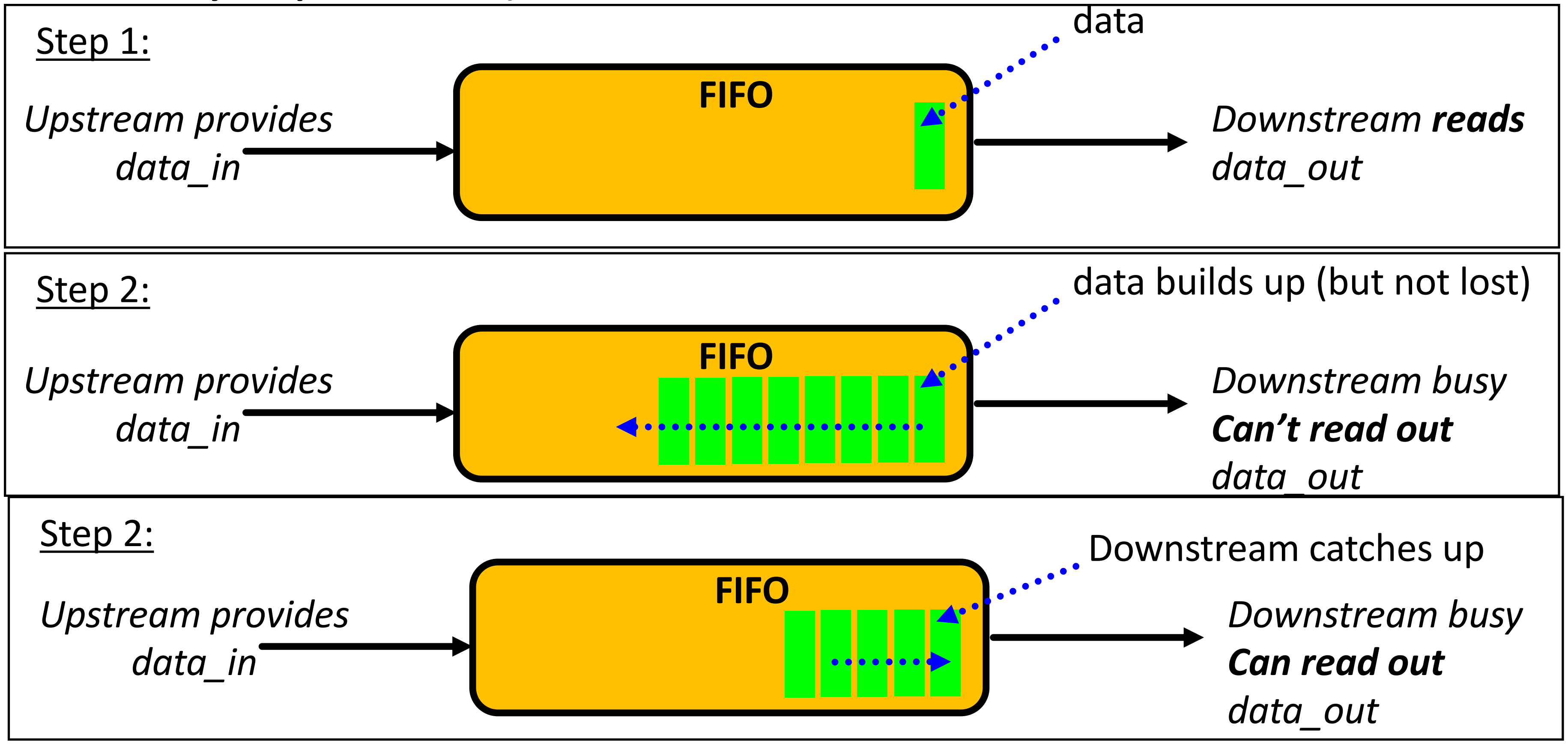
Real-time Audio Spectrograph



- What happens if one part can't process data as quickly as another one generates it?
- Hopefully the backpropagation of **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

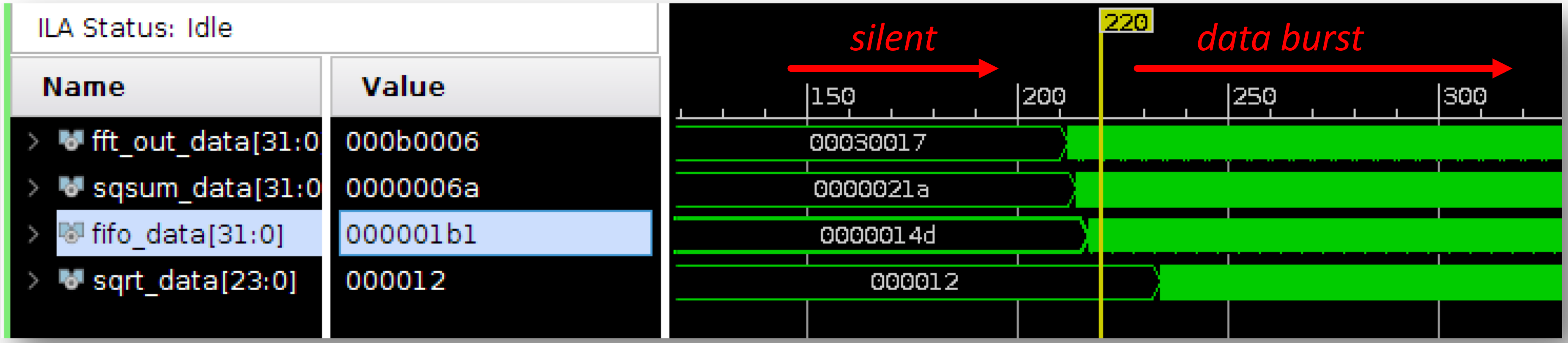


Step 1:
Upstream provides data_in
FIFO
data
Downstream **reads** data_out
Step 2:
Upstream provides data_in
FIFO
data builds up (but not lost)
Downstream busy
Can't read out data_out
Step 2:
Upstream provides data_in
FIFO
Downstream catches up
Downstream busy
Can read out data_out

FIFOs

- If upstream produces measurements at 100 MHz and downstream processes at 50 MHz, FIFOs **<u>will not help!</u>**
- 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

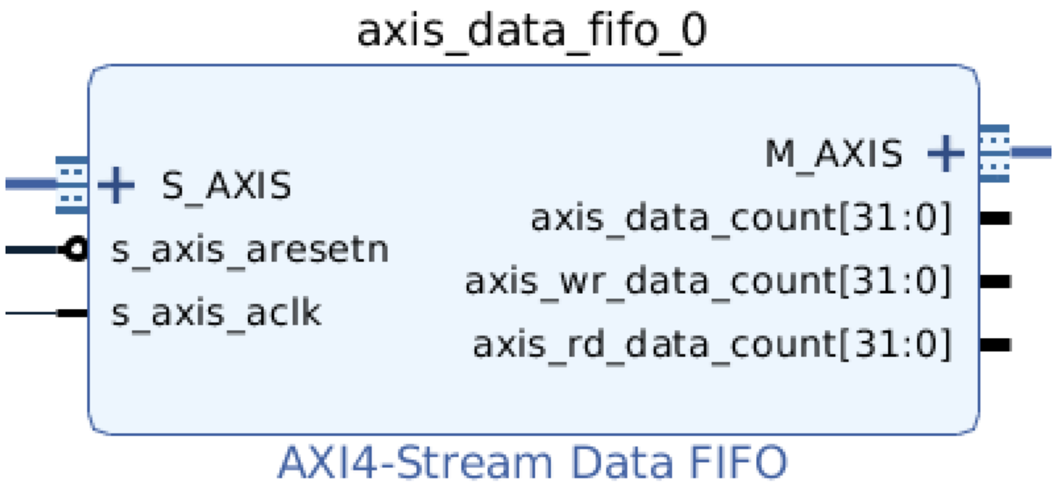
FFT Data Output



ILA Status: Idle

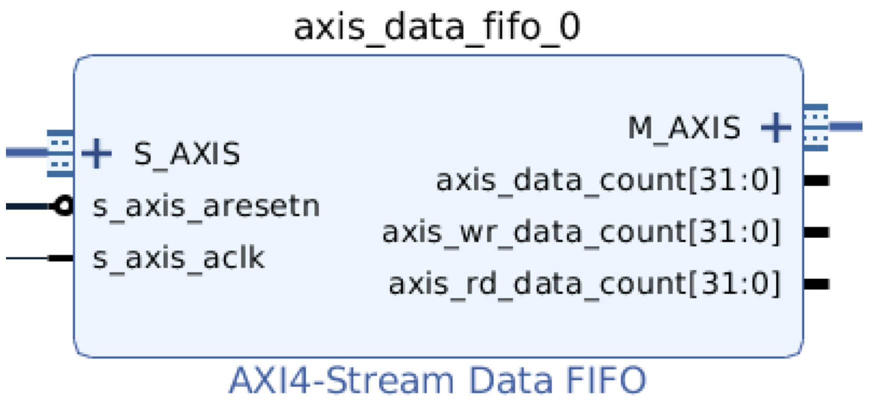
Name	Value
fft_out_data[31:0	000b0006
sqsum_data[31:0	0000006a
fifo_data[31:0]	000001b1
sqrt_data[23:0]	000012

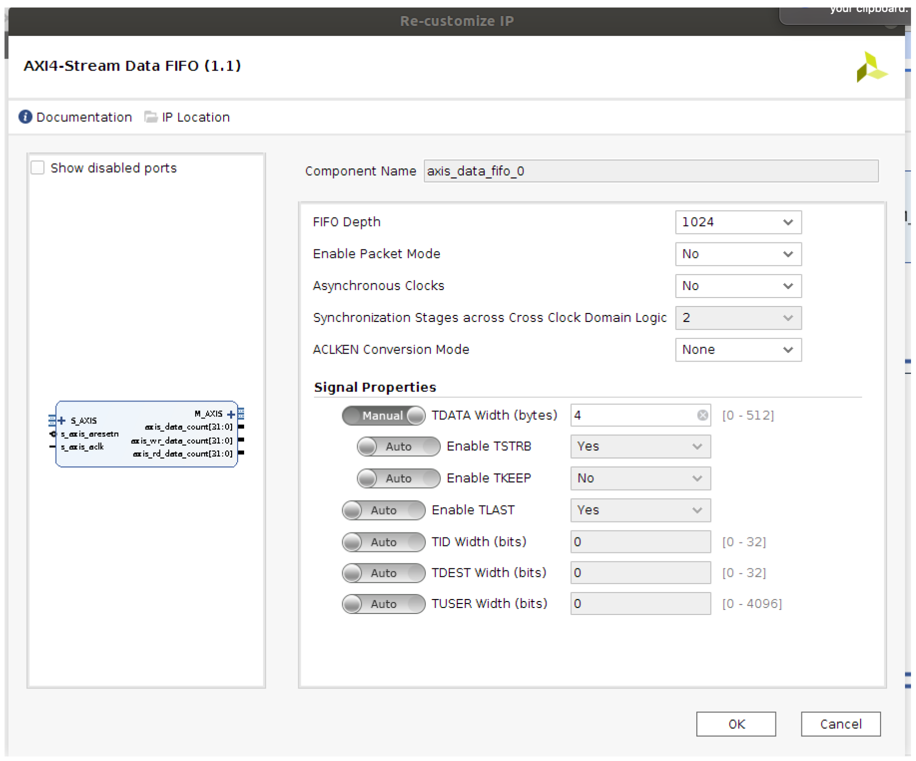
AXI4S FIFO


axis_data_fifo_0
S_AXIS
s_axis_aresetn
s_axis_aclk
M_AXIS
axis_data_count[31:0]
axis_wr_data_count[31:0]
axis_rd_data_count[31:0]
AXI4-Stream Data FIFO

AXI4S FIFO

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





AXI4-Stream Data FIFO (1.1)

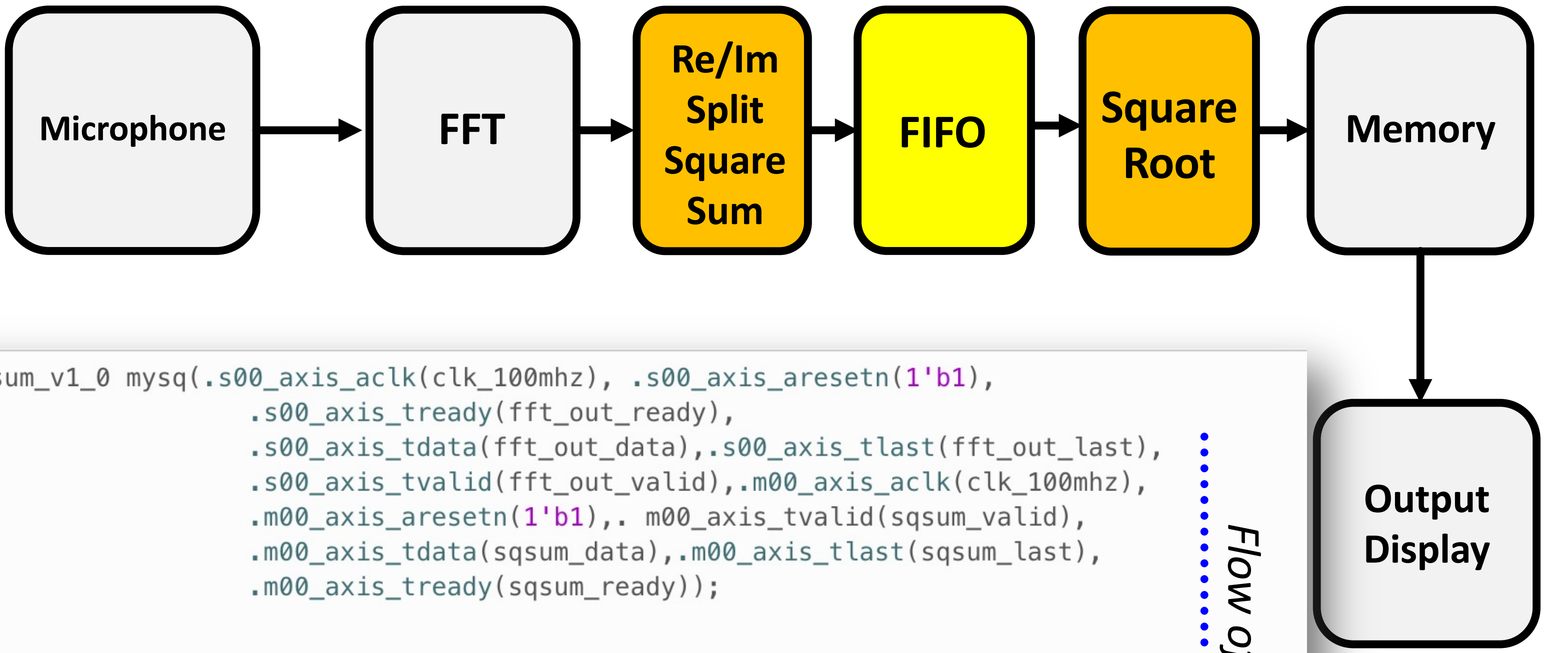
Component Name: axis_data_fifo_0

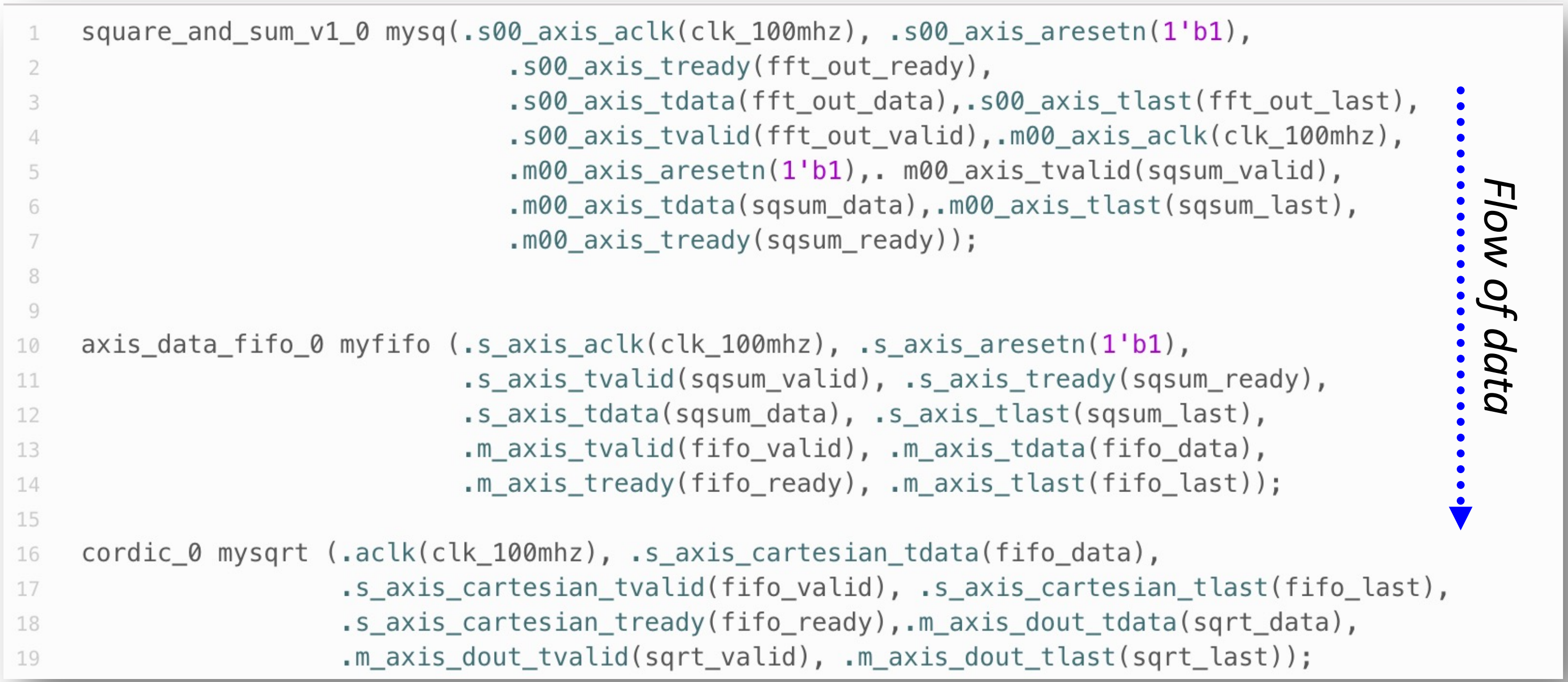
FIFO Depth	1024
Enable Packet Mode	No
Asynchronous Clocks	No
Synchronization Stages across Cross Clock Domain Logic	2
ACLKEN Conversion Mode	None

Signal Properties

Manual	TDATA Width (bytes)	4	[0 - 512]
Auto	Enable TSTRB	Yes	
Auto	Enable TKEEP	No	
Auto	Enable TLAST	Yes	
Auto	TID Width (bits)	0	[0 - 32]
Auto	TDEST Width (bits)	0	[0 - 32]
Auto	TUSER Width (bits)	0	[0 - 4096]

Real-time Audio Spectrograph



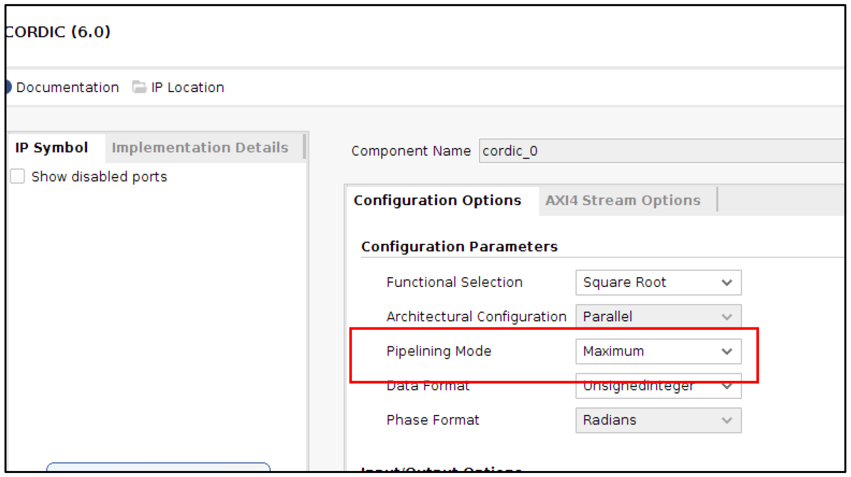

```
square_and_sum_v1_0 mysq(.s00_axis_aclk(clk_100mhz), .s00_axis_aresetn(1'b1),
                           .s00_axis_tready(fft_out_ready),
                           .s00_axis_tdata(fft_out_data),.s00_axis_tlast(fft_out_last),
                           .s00_axis_tvalid(fft_out_valid),.m00_axis_aclk(clk_100mhz),
                           .m00_axis_aresetn(1'b1),. m00_axis_tvalid(sqsum_valid),
                           .m00_axis_tdata(sqsum_data),.m00_axis_tlast(sqsum_last),
                           .m00_axis_tready(sqsum_ready));


axis_data_fifo_0 myfifo (.s_axis_aclk(clk_100mhz), .s_axis_aresetn(1'b1),
                         .s_axis_tvalid(sqsum_valid), .s_axis_tready(sqsum_ready),
                         .s_axis_tdata(sqsum_data), .s_axis_tlast(sqsum_last),
                         .m_axis_tvalid(fifo_valid), .m_axis_tdata(fifo_data),
                         .m_axis_tready(fifo_ready), .m_axis_tlast(fifo_last));

cordic_0 mysqrt (.aclk(clk_100mhz), .s_axis_cartesian_tdata(fifo_data),
                 .s_axis_cartesian_tvalid(fifo_valid), .s_axis_cartesian_tlast(fifo_last),
                 .s_axis_cartesian_tready(fifo_ready),.m_axis_dout_tdata(sqrt_data),
                 .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.

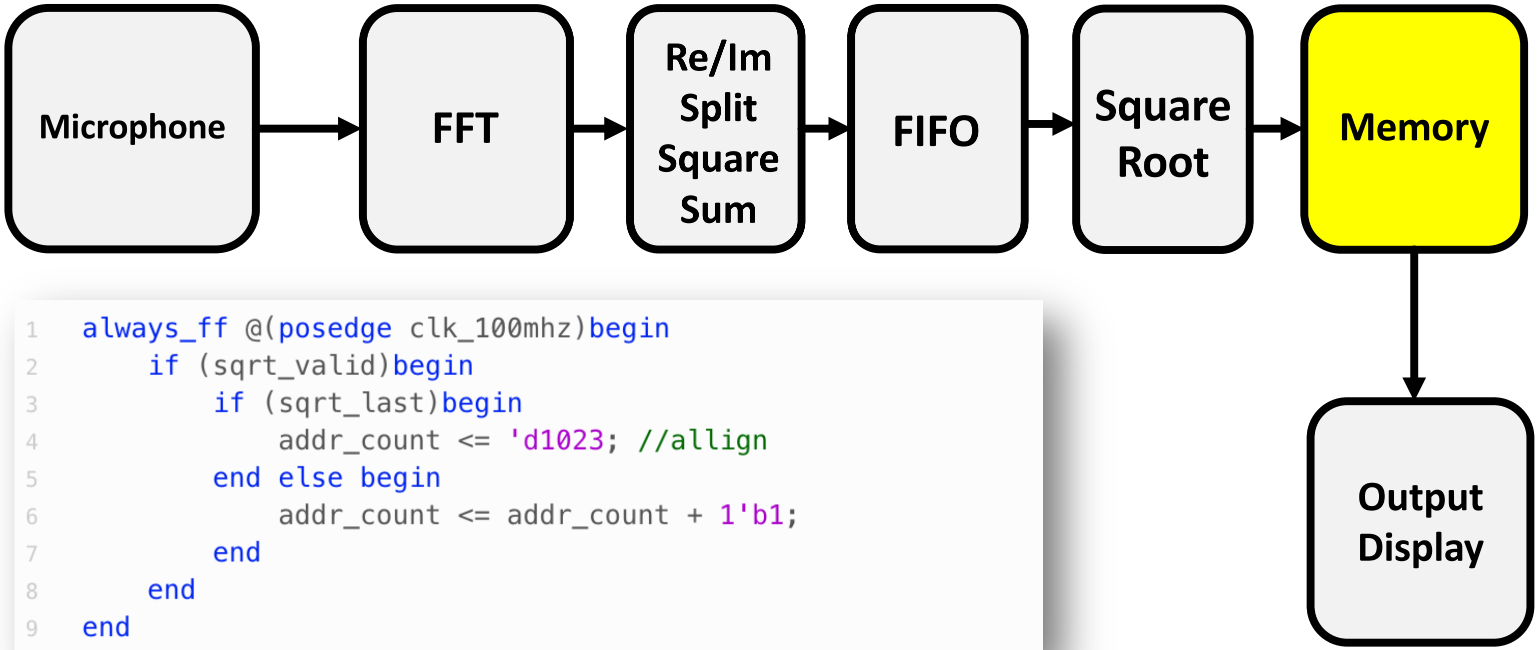

CORDIC (6.0)

Component Name: cordic_0

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

- 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



```verilog
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
```

```verilog

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 week 05)

```
xilinx_true_dual_port_read_first_2_clock_ram #(
  .RAM_WIDTH(32),
  .RAM_DEPTH(2048))
  frame_buffer (
  //Write Side (148.5 MHz)
  .addra(addr_count),
  .clka(axi_clk),
  .wea(sqrt_valid),
  .dina({8'b0,sqrt_data}),
  .ena(1'b1),
  .regcea(1'b1),
  .rsta(btnd),
  .douta(),
  //Read Side (74.25 MHz)
  .addrb(draw_addr+3), //lazy pipelining
  .dinb(16'b0),
  .clkb(pixel_clk),
  .web(1'b0),
  .enb(1'b1),
  .rstb(btnd),
  .regceb(1'b1),
  .doutb(amp_out)
);
```

2048 X 32 bit Memory

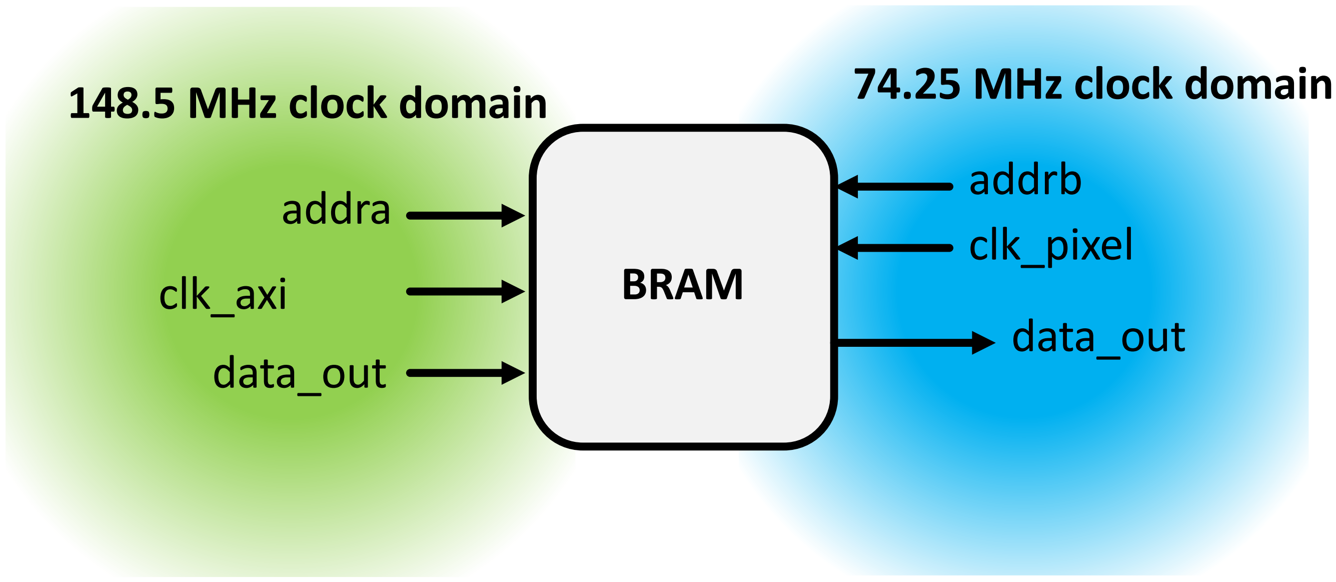
Why 2048? There's 2048 FFT values to store!
Why 32 bit? Each magnitude is 32 bits

Use AXI if you need a bus

- There's some somewhat decent critiques of the AXI protocol…
- But usually most boil down to incomplete compliance of particular modules…
  - Even in 6.S965 (6.205++) we found some AMD/Xilinx IP is not actually AXI compliant
- It is pretty well thought out tbh, so don't necessarily assume you can do better, especially in this class.

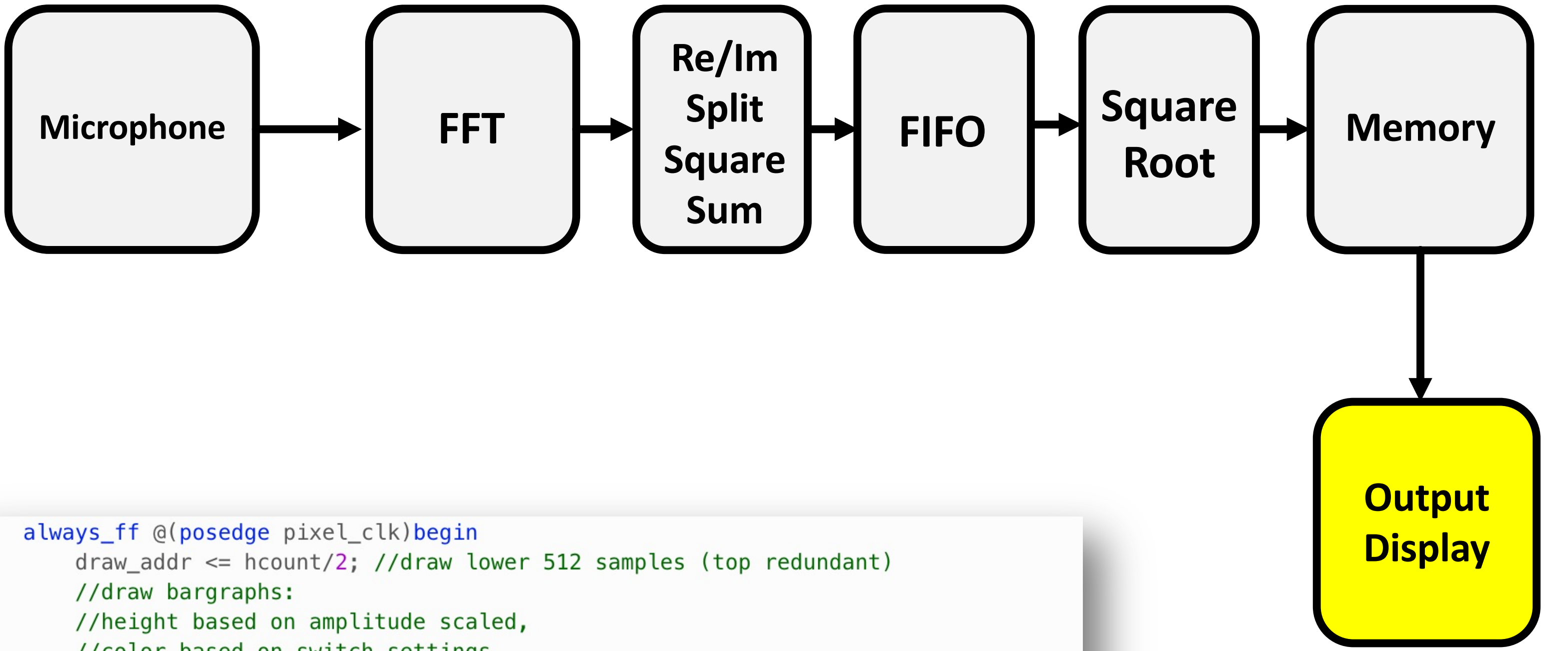
Video Memory

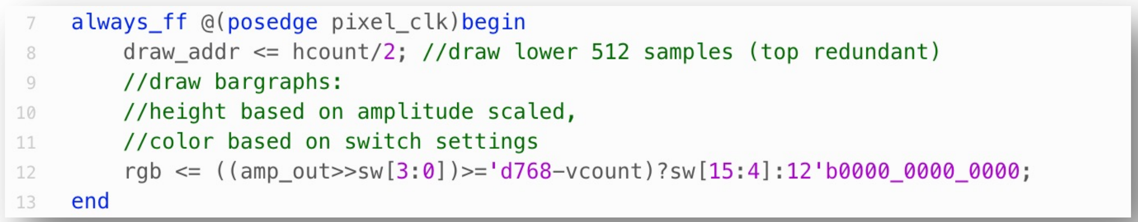
- Two Port Block RAM:
  - Each side separately clocked!
  - Don't have to worry about running upstream at video clock rate!



Real-time Audio Spectrograph

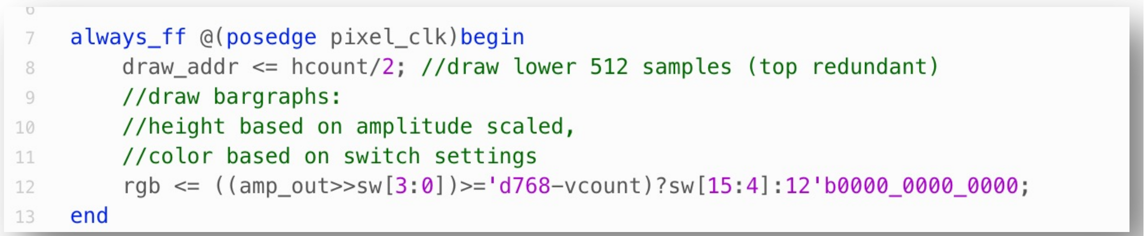
- The last step!



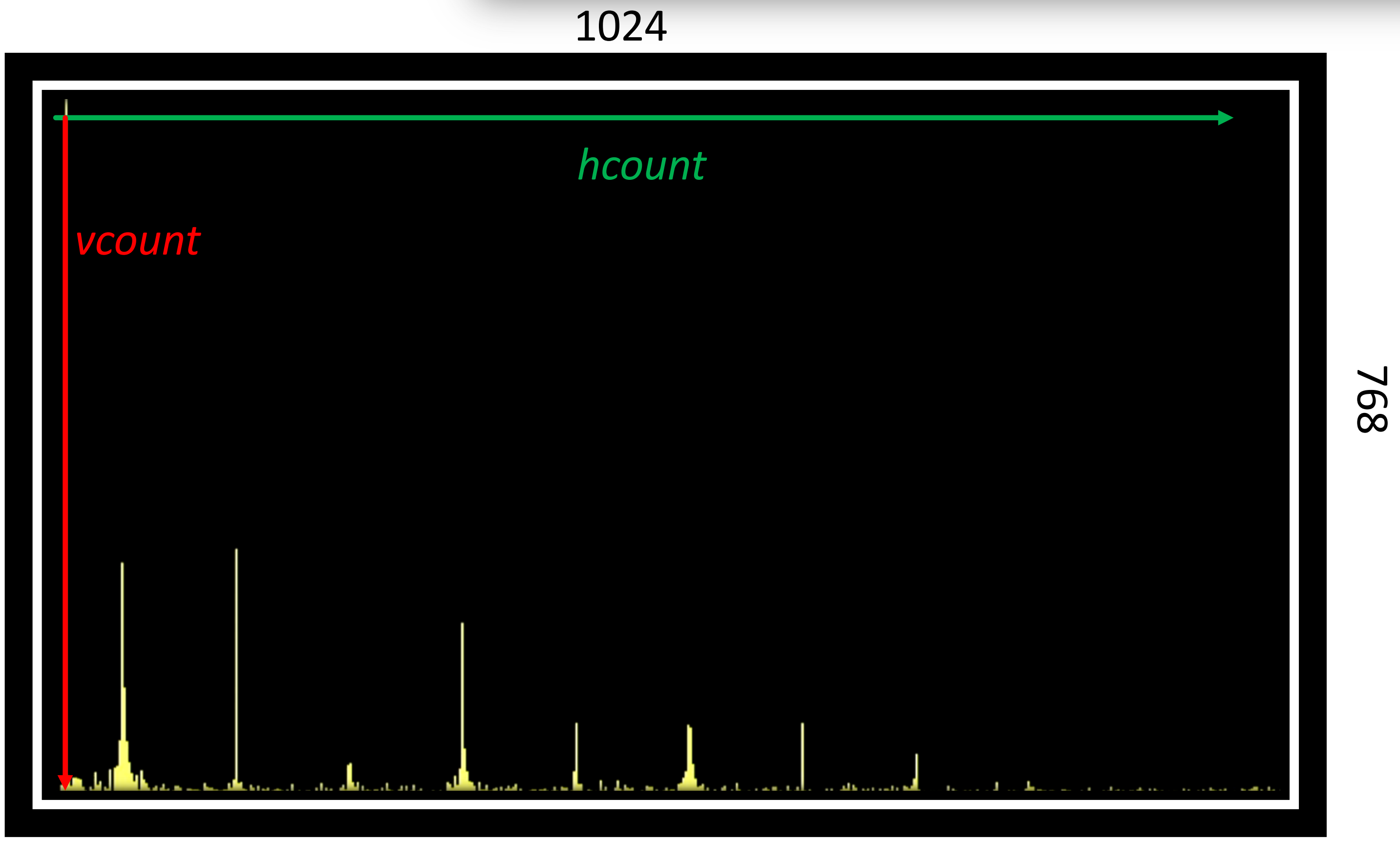


```
always_ff @(posedge pixel_clk)begin
    draw_addr <= hcount/2; //draw lower 512 samples (top redundant)
    //draw bargraphs:
    //height based on amplitude scaled,
    //color based on switch settings
    rgb <= ((amp_out>>sw[3:0])>='d768-vcount)?sw[15:4]:12'b0000_0000_0000;
end
```

Display Output

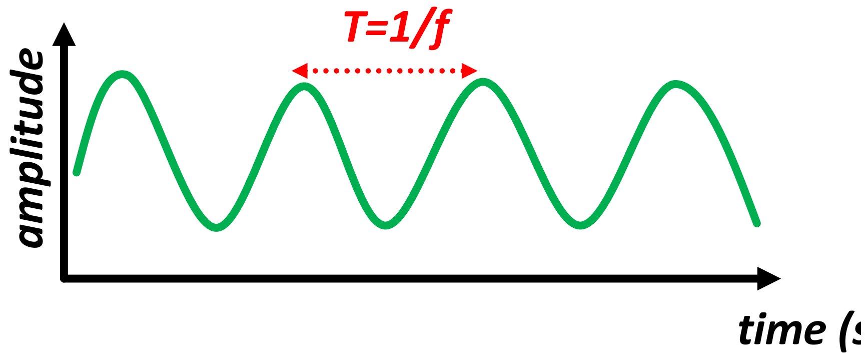


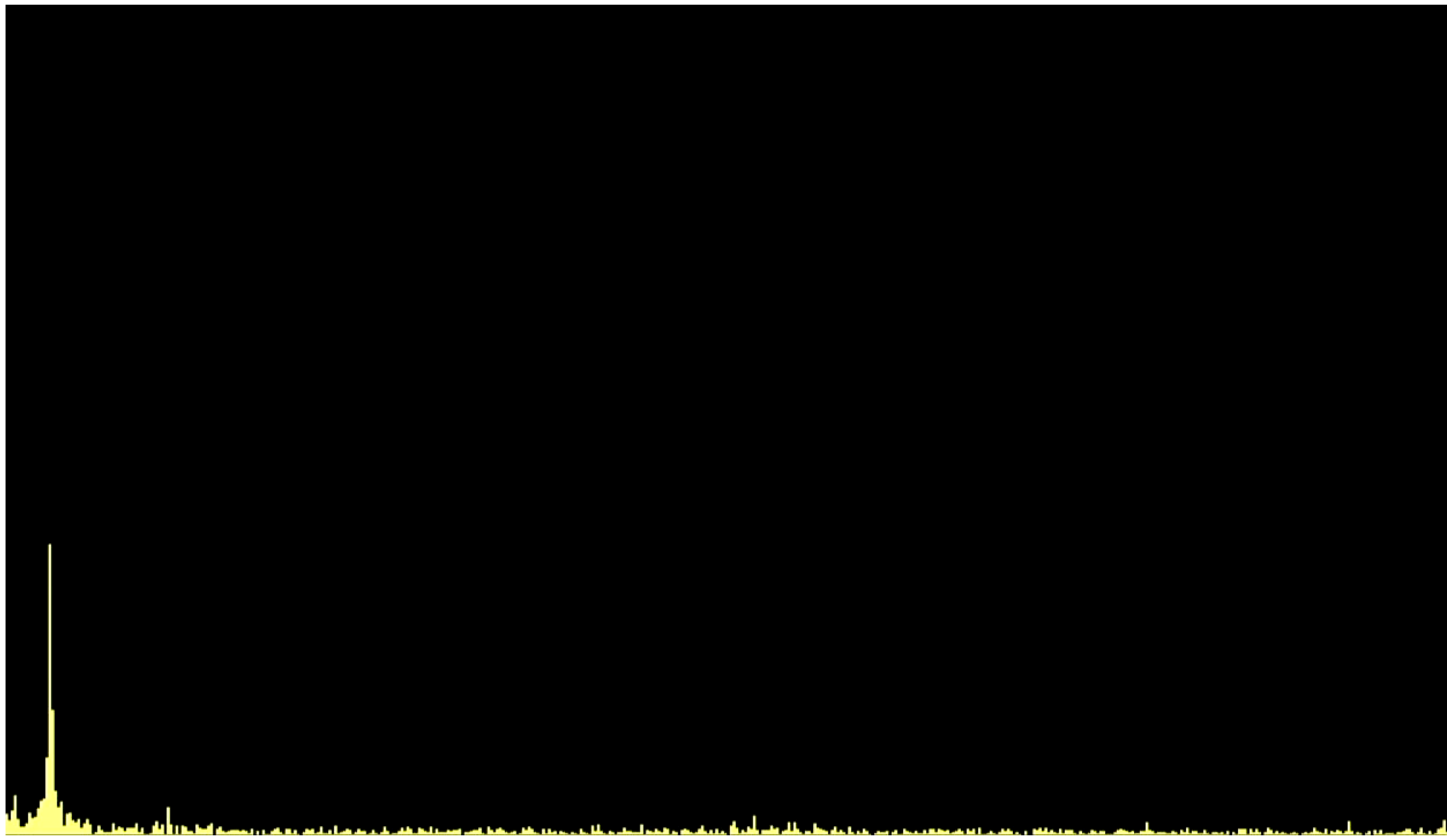
```
always_ff @(posedge pixel_clk)begin
    draw_addr <= hcount/2; //draw lower 512 samples (top redundant)
    //draw bargraphs:
    //height based on amplitude scaled,
    //color based on switch settings
    rgb <= ((amp_out>>sw[3:0])>='d768-vcount)?sw[15:4]:12'b0000_0000_0000;
end
```



Sine Waves In

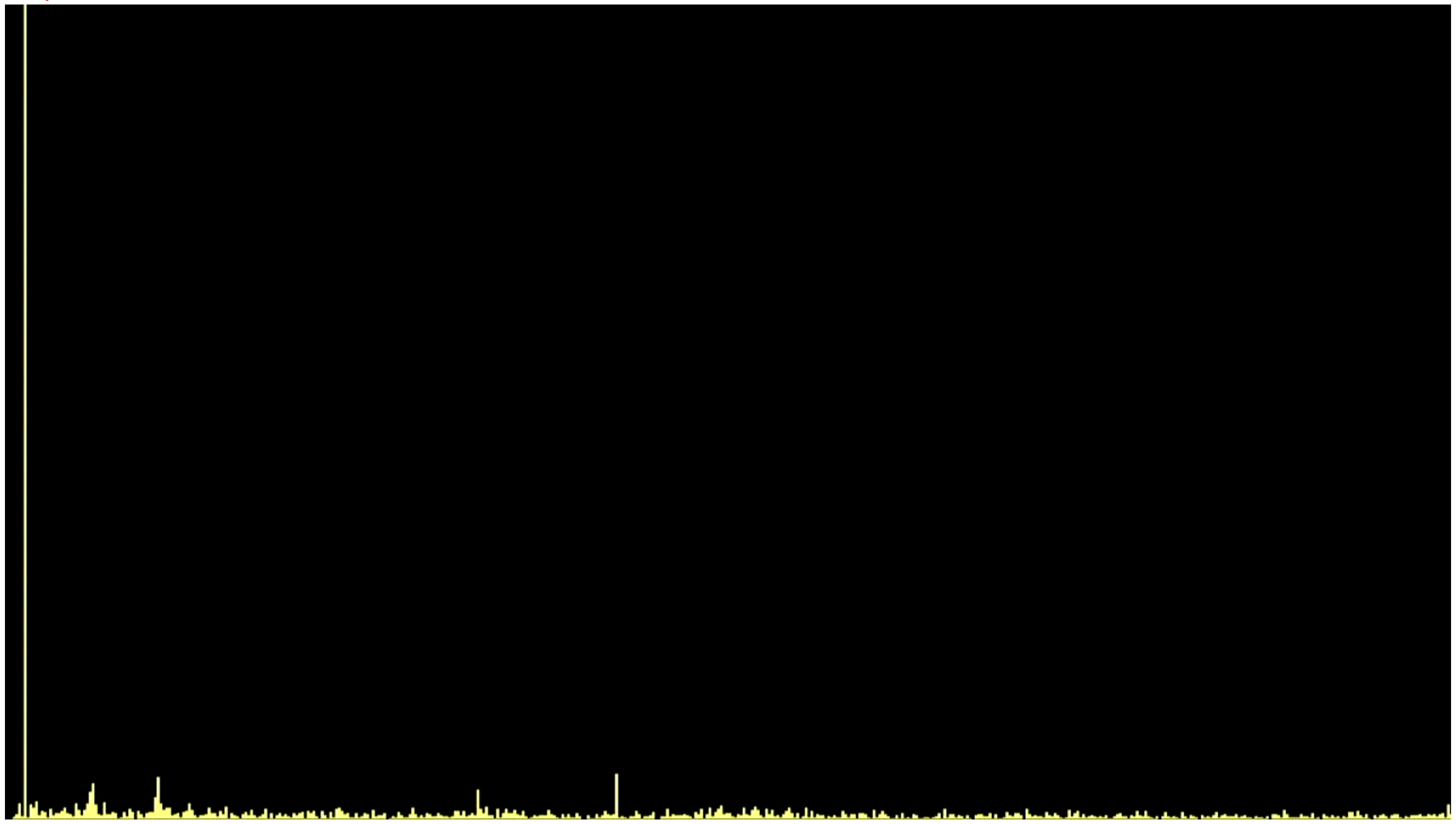
*The square waves in later





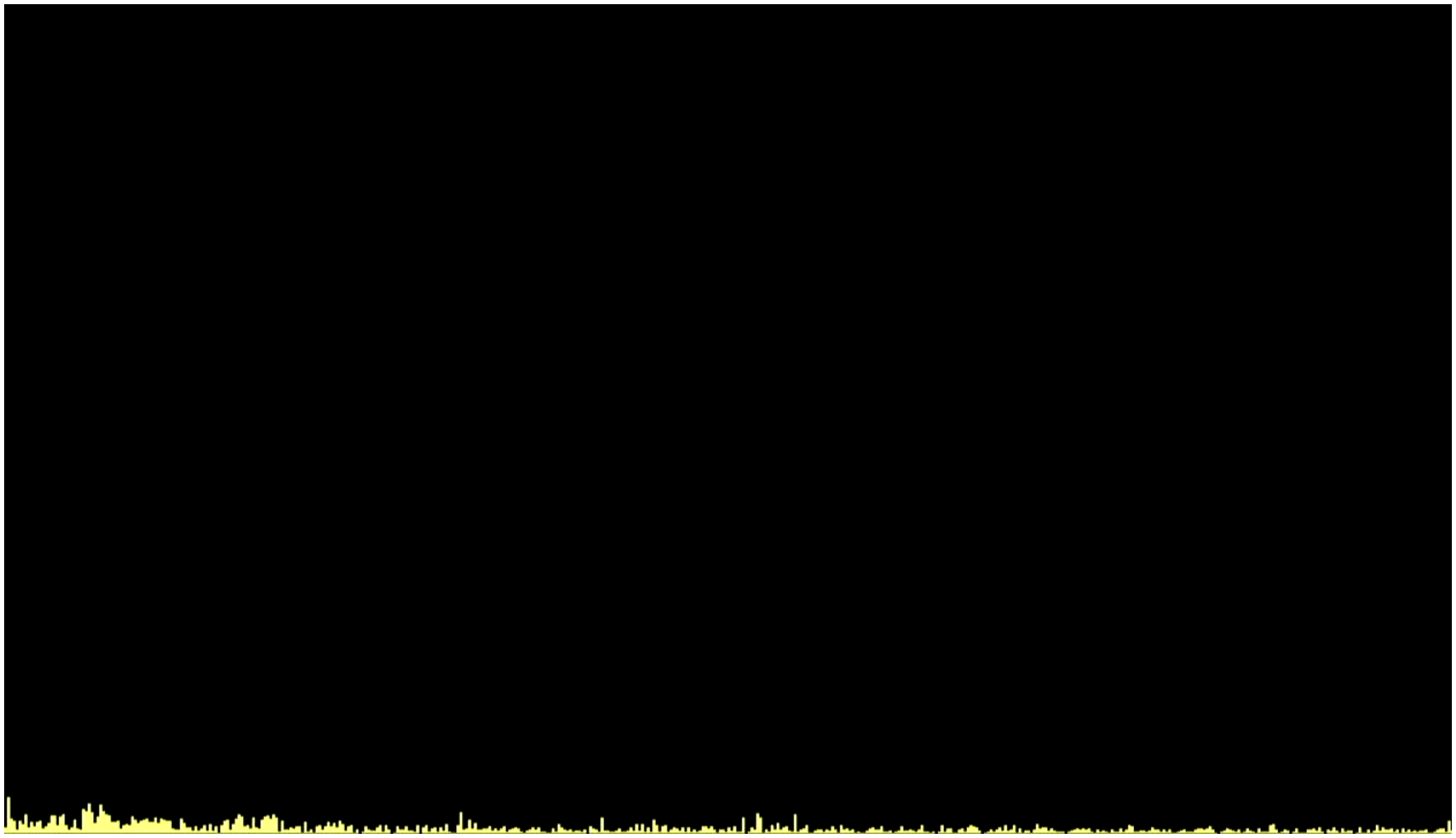
Cat

Ignore that line…I had a pipelining issue



Me

Beyoncé



20th Century Fox

Celine Dion

Are we good on timing?

From post_route_timing.rpt

- Report say, “yes”

```
Timing Report

Slack (MET) :              1.005ns  (required time - arrival time)
```

Resource Usage?

From post_place_util.rpt

- Quite a bit

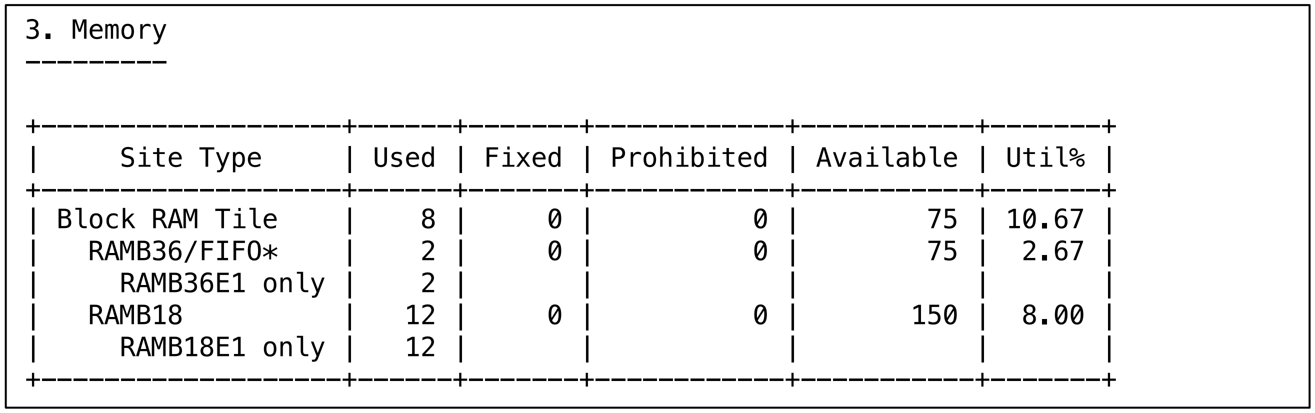
```
2. Slice Logic Distribution
---------------------------


+-------------------------------------------+------+-------+------------+-----------+-------+
|                 Site Type                 | Used | Fixed | Prohibited | Available | Util% |
+-------------------------------------------+------+-------+------------+-----------+-------+
| Slice                                     | 1304 |     0 |          0 |      8150 | 16.00 |
|   SLICEL                                  |  828 |     0 |            |           |       |
|   SLICEM                                  |  476 |     0 |            |           |       |
| LUT as Logic                              | 2524 |     0 |          0 |     32600 |  7.74 |
|   using O5 output only                    |    7 |       |            |           |       |
|   using O6 output only                    | 1719 |       |            |           |       |
|   using O5 and O6                         |  798 |       |            |           |       |
| LUT as Memory                             |  584 |     0 |          0 |      9600 |  6.08 |
|   LUT as Distributed RAM                  |    0 |     0 |            |           |       |
|   LUT as Shift Register                   |  584 |     0 |            |           |       |
|     using O5 output only                  |   29 |       |            |           |       |
|     using O6 output only                  |  199 |       |            |           |       |
|     using O5 and O6                       |  356 |       |            |           |       |
| Slice Registers                           | 5356 |     0 |          0 |     65200 |  8.21 |
|   Register driven from within the Slice   | 3574 |       |            |           |       |
|   Register driven from outside the Slice  | 1782 |       |            |           |       |
|     LUT in front of the register is unused | 1128 |       |            |           |       |
|     LUT in front of the register is used  |  654 |       |            |           |       |
| Unique Control Sets                       |   51 |       |          0 |      8150 |  0.63 |
+-------------------------------------------+------+-------+------------+-----------+-------+
* * Note: Available Control Sets calculated as Slice * 1, Review the Control Sets Report for more
information regarding control sets.
```

Resource Usage?

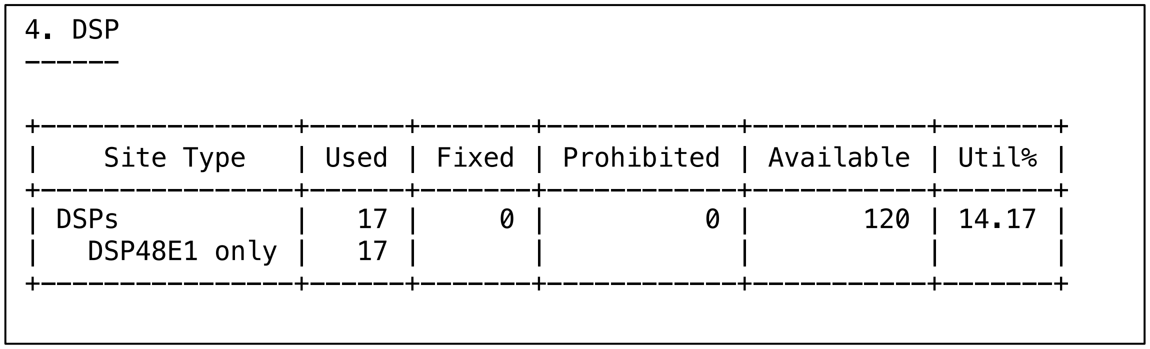
From post_place_util.rpt

- Not much!



```
3. Memory
---------

+-------------------+------+-------+------------+-----------+-------+
|     Site Type     | Used | Fixed | Prohibited | Available | Util% |
+-------------------+------+-------+------------+-----------+-------+
| Block RAM Tile    |    8 |     0 |          0 |        75 | 10.67 |
|   RAMB36/FIFO*    |    2 |     0 |          0 |        75 |  2.67 |
|     RAMB36E1 only |    2 |       |            |           |       |
|   RAMB18          |   12 |     0 |          0 |       150 |  8.00 |
|     RAMB18E1 only |   12 |       |            |           |       |
+-------------------+------+-------+------------+-----------+-------+
```



```
4. DSP
------

+----------------+------+-------+------------+-----------+-------+
|    Site Type   | Used | Fixed | Prohibited | Available | Util% |
+----------------+------+-------+------------+-----------+-------+
| DSPs           |   17 |     0 |          0 |       120 | 14.17 |
|   DSP48E1 only |   17 |       |            |           |       |
+----------------+------+-------+------------+-----------+-------+
```

Make it much better

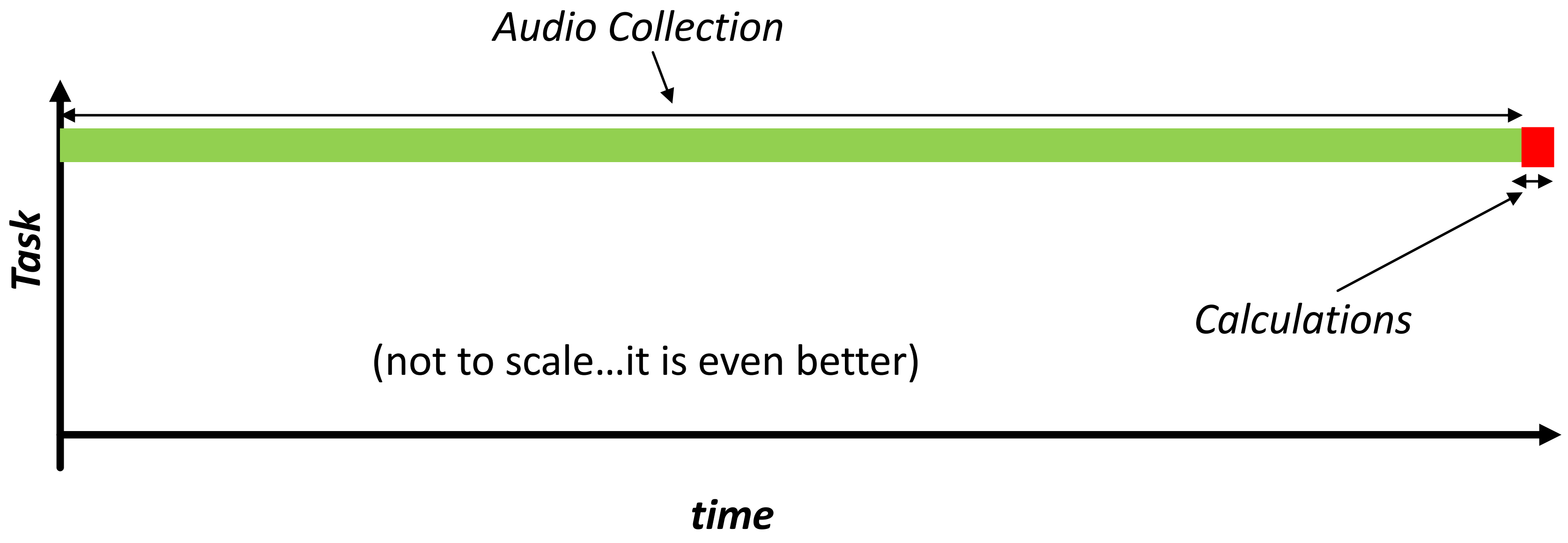
- This was a 2048 point FFT at 19 kHz
- It is a very poorly designed pipeline
    - There’s a FIFO for no reason.
    - We use lots of extra bits because I was lazy
    - The FFT is so ridiculously over-performant that it isn’t even funny
- We could likely get same or better performance out of system that uses far fewer resources on almost all fronts.

How Quick to calculate FFT?

- Collect 2048 audio measurements :
  - @~19 KHz. Every 52 microseconds (so ~107 milliseconds total)
- Compute 2048 point FFT:
  - 6273 clock cycles @ 148.5MHz (42.25 μs)
- Square and Sum:
  - 2 cycles @ 148.5MHz (13.48 ns)
- FIFO:
  - 3 cycles @ 148.5MHz overhead latency (20 ns)
- Root:
  - 26 cycles @ 148.5 MHz (175 ns)

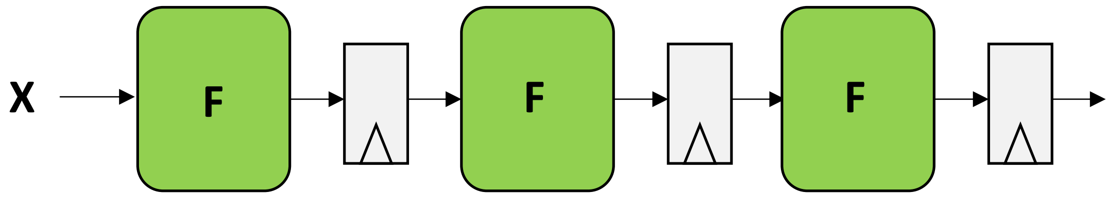
How Quick?...Uselessly Quick

- After audio clip captured, FFT generated and ready to render in 42.5 µs
- Our audio samples are measured *every* 52 µs and and a full frame of samples is captured every 100 milliseconds.
- This is a differential of like *2000x*
- We can calculate our entire FFT in between individual audio samples,



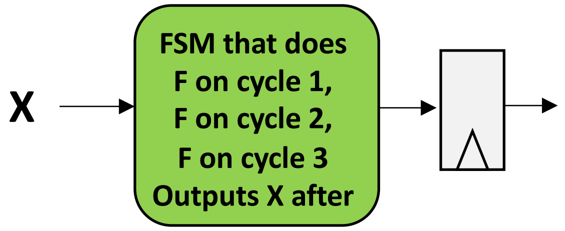
No need to have fully-pipelined FFT for this application

- 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:?

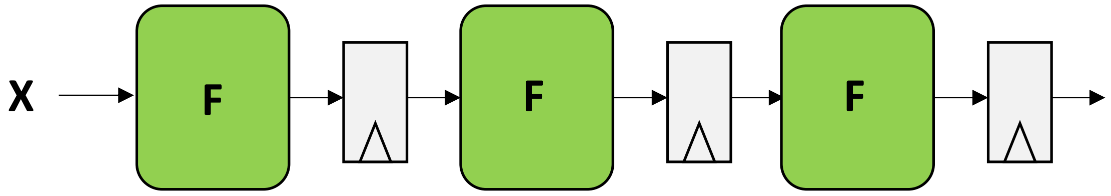


Latency: 3*T_{clk}
Throughput: 1/ (3*T_{clk})
Likely uses fewer resources

Where Could We Go From Here?

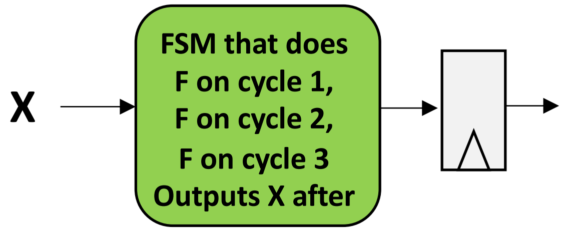
- Cut the FIFO (I put it in just for fun)
- Size the IP for the actual data we’re handling:
  - a lot of the systems are set at 16 bits but our audio samples are only 7 bits originally
  - The CORDIC is uselessly large
- Pick a better FFT:
  - Meaning…

This is the Great Tradeoff!



More resources,
Better Throughput
Same Latency

OR

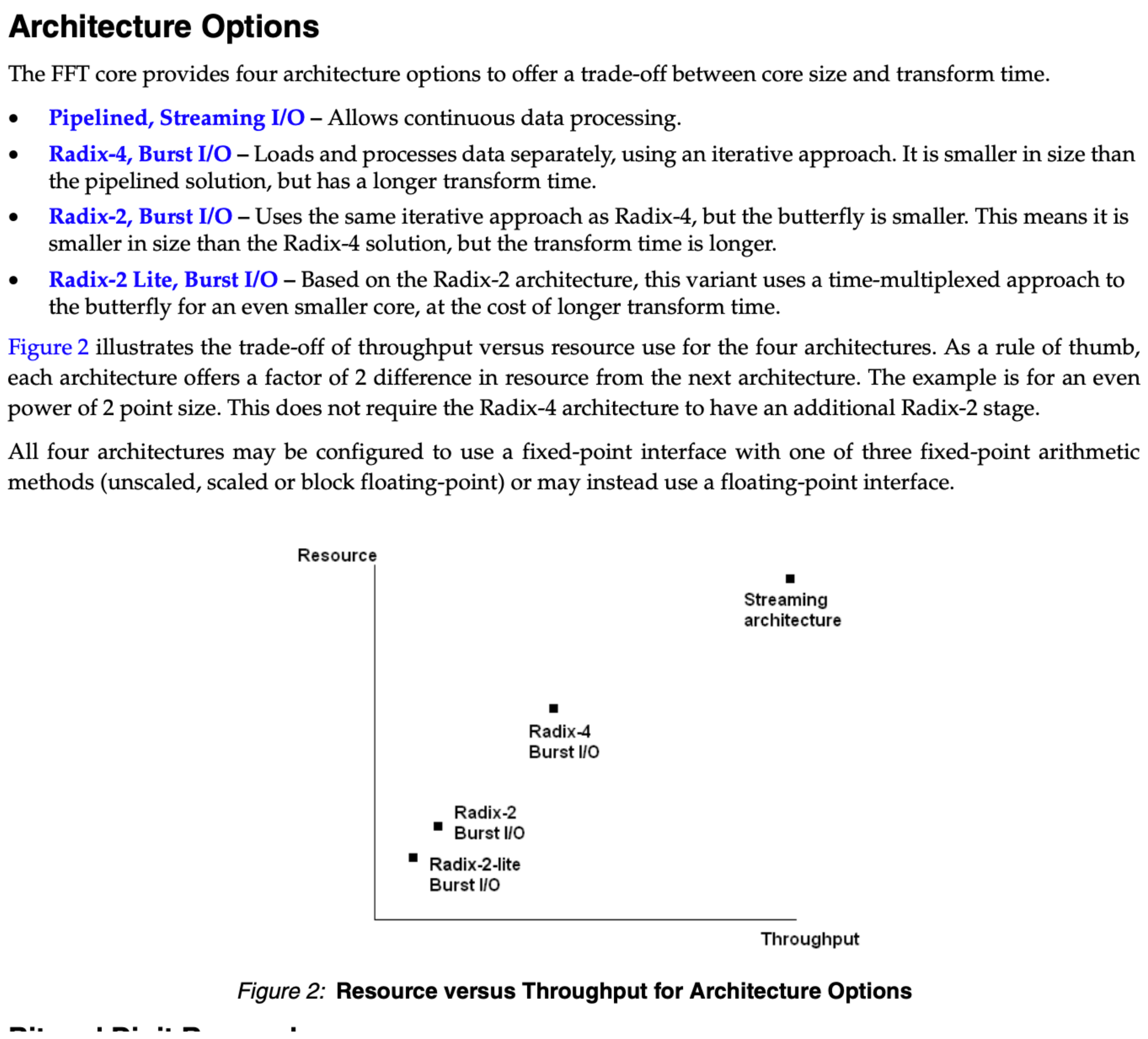


Fewer resources,
Worse Throughput
Same Latency

- Base on what you need for the design!

Pick Better FFT Implementation

- We can get 16 times the frequency resolution
- For the same resource usage if we modify things to take advantage of slow data production



Architecture Options

The FFT core provides four architecture options to offer a trade-off between core size and transform time.

- **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.

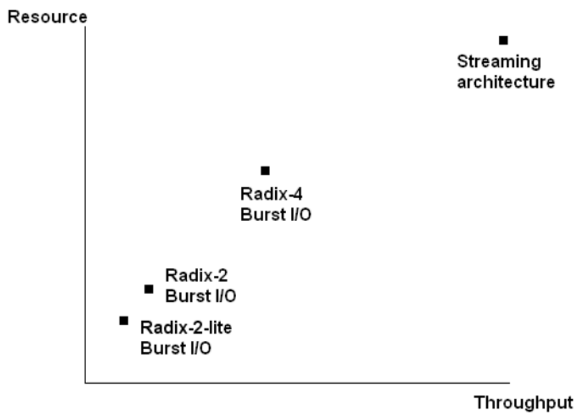
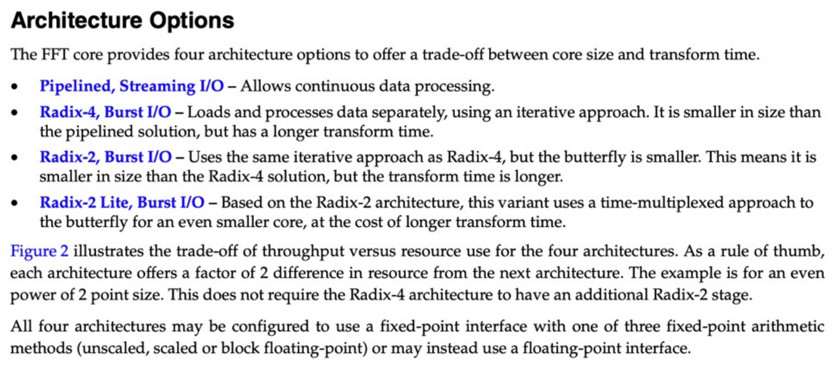


Figure 2: **Resource versus Throughput for Architecture Options**

Pick Better FFT Implementation

- We can get 16 times the frequency resolution and use ¼ the DSP blocks at the expense of:
  - Using 3X the BRAM, (still fine)
  - Having a latency of 3.764 ms (still totally fine)

And tons of other optimizations!!!



Architecture Options

The FFT core provides four architecture options to offer a trade-off between core size and transform time.

- **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.

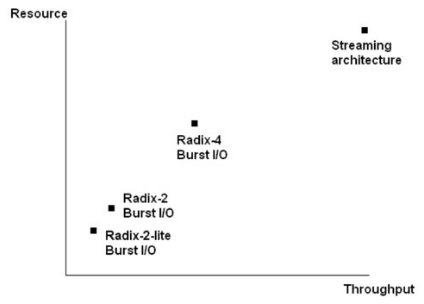
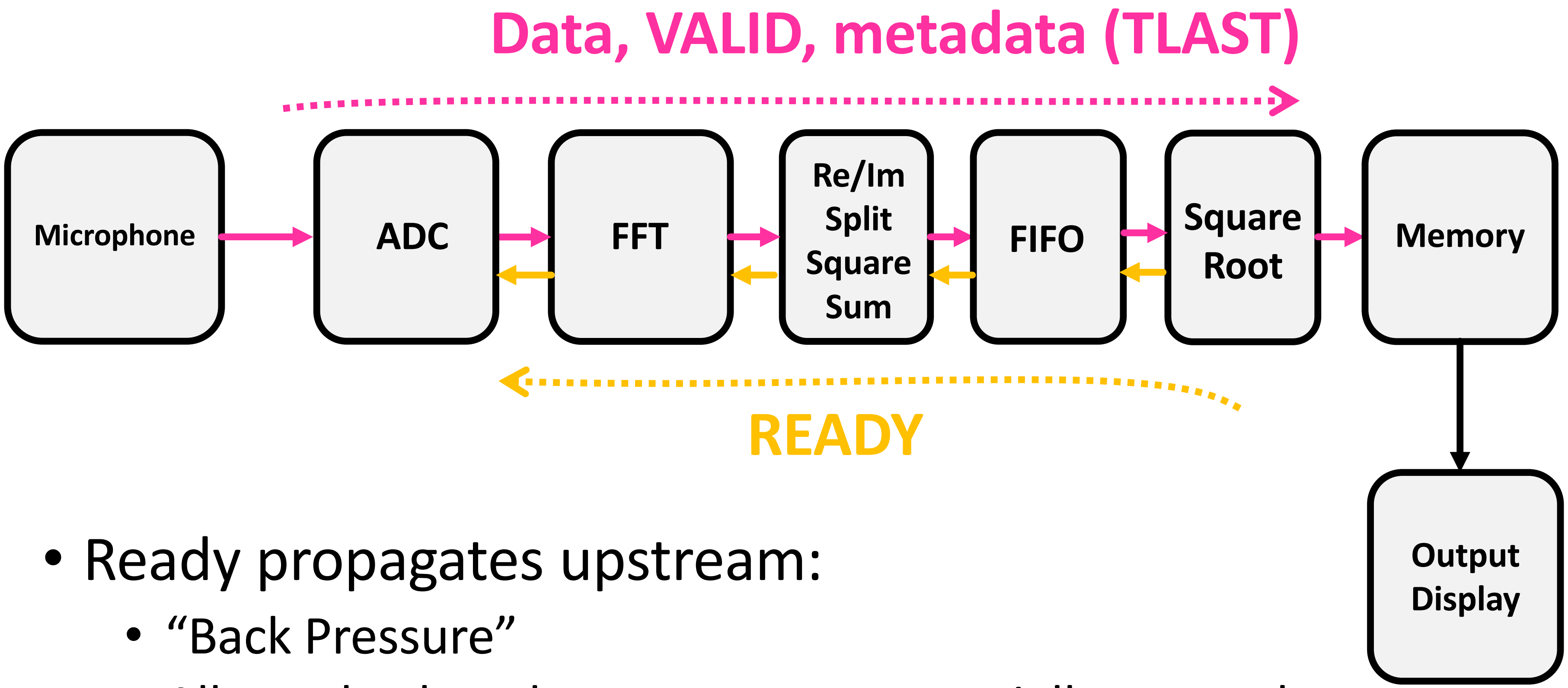


Figure 2: **Resource versus Throughput for Architecture Options**

Different Directions

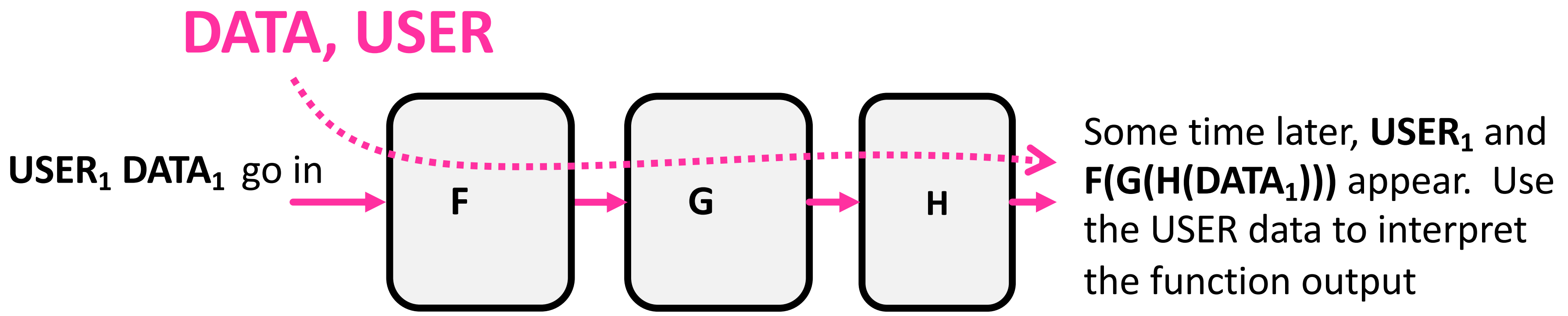
- Data Propagates downstream:



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

Usefulness of Metadata or markers

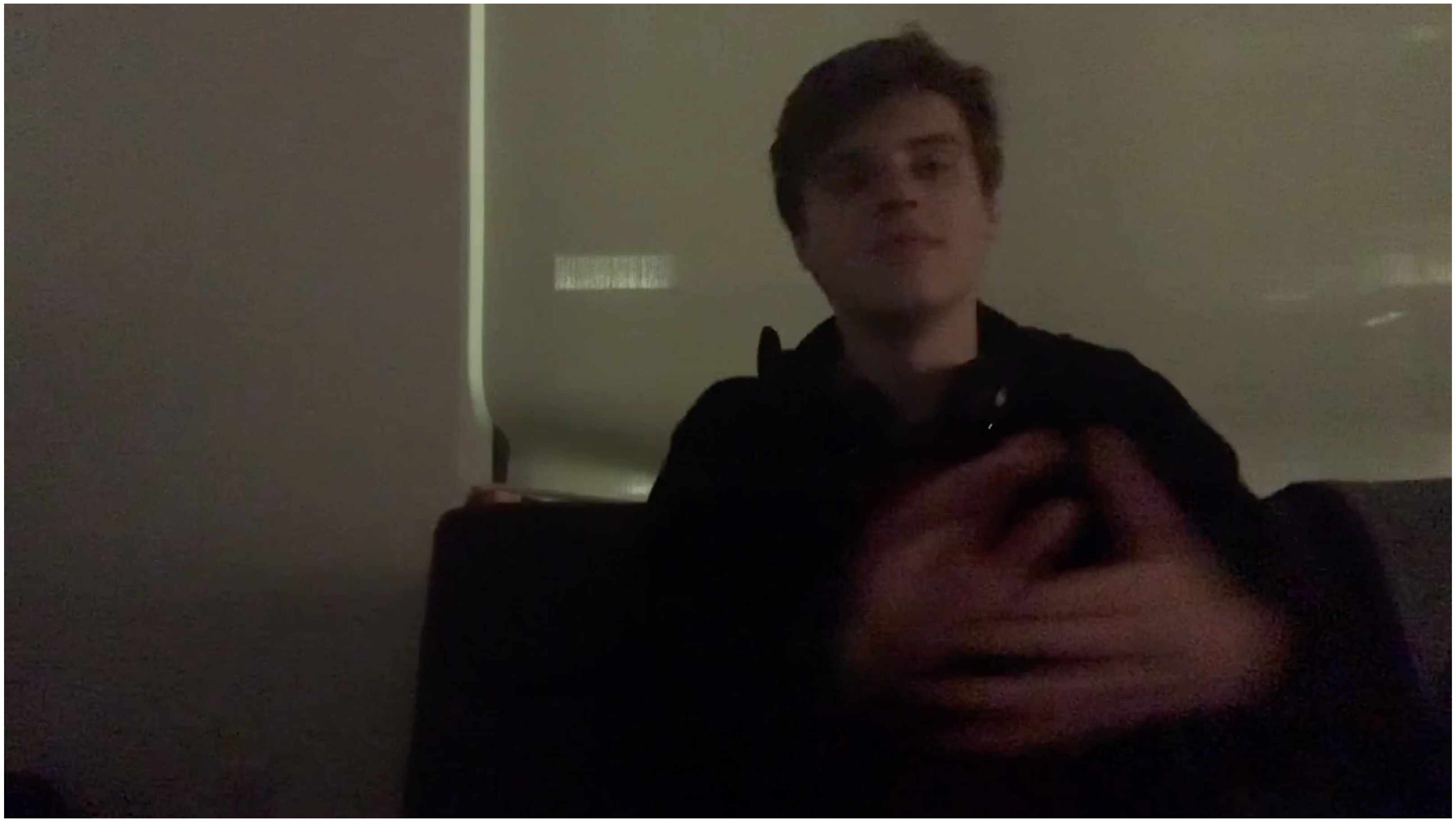
- If data takes a really long time you can also activate a USER field to send along with DATA
- USER values will be unchanged but will get pipelined properly along with the corresponding data they're sent in with


DATA, USER
USER$_1$ DATA$_1$ go in
F
G
H
Some time later, **USER$_1$** and **F(G(H(DATA$_1$)))** appear. Use the USER data to interpret the function output

Next Week

- No class on Tuesday (holiday):
- Thursday we’ll do some signal processing concepts and that will likely bleed into the following Tuesday
- Then one or two more lectures and we’re done.

Final Project TAs from 2022



Sources

This is the thing right here…the spec sheet/manual is surprisingly good!!

- **"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.