Voltage|Divide

FPGA

GROWING AS AN FPGA DEVELOPER

APRIL 3, 2023 | ALEXLAO512 | 4 COMMENTS

Table of Contents

March 19, 2023 – Initial

I get asked pretty often about how to get into FPGA design or how to become a good or even great FPGA developer. Since I get asked quite often, I decided to put my thoughts down here instead of typing it out again and again forever. I hope what I write here will help you on your journey. I think some of the content might be useful to non-FPGA developers even though I don't really plan for it to be as such.

This article is meant for readers who have already made it through their first FPGA course, maybe they are in a masters program or they might be in the first few years of their new job as an FPGA developer. It is not reall meant for beginners but might be useful for them none the less.

The stuff here may be overwhelming for beginners or even people a few years into their careers. I want to mention for the readers that the knowledge and skills I talk about here were all developed slowly in parallel o an as needed basis, over many years. Most importantly, I want to mention a lot of the things I list are a work in progress for me, I am not a master in any of the things and in some cases only know the existence of certain things I want to look into later. This page is not meant to be something you follow and become an FPGA devel oper on completion, it is meant to be a resource added to the plethora of resources available to you online, in books, or at school and work.

I also want to preface what I write with where most of my experience comes from, I am an electronics hobbyis and I work as an FPGA developer within a small, very multi-disciplinary team. The path an FPGA developer would take when working in a large company would probably be very different. People in large companies tend to be more focused on fewer types of tasks day to day. I generally prefer being multi-disciplinary myself, starting initially in embedded software development, moving through circuit board design, before settling on FPGA development as a focus for myself. I have a lot of respect for people that focus on and become extremel skilled in a single field but I favour a broad range of knowledge and skill with a focus on FPGA development fo myself. My writing will reflect that.

This article was first written when I have had about 5 years of professional FPGA development experience including the internships I have held. It has been 8 years since my first FPGA course and I do hold an FPGA related masters degree however the masters are not entirely FPGA focused.

I don't claim to know what I am doing, I am sure there are things I say below that I won't agree with in a few years time. I intend on updating this post as I learn over time. At the top of the article is the date this was last edited with a short change log.

FPGA and RTL Development

Some of what I say might apply to ASIC design as well, however I have never worked on ASICs so that is as much as I will mention about them.

Assuming you already have the fundamentals often taught in second or third year undergraduate FPGA courses, where do you go? There are many paths to take on your journey towards being able to construct more complex logic and systems. Most of these skills are best learned in the context of a project either at work or o your own time, on an as needed basis.

- General RTL Development
  - Traditional Hardware Description Languages
    - You should aim to know how to write one of SystemVerilog, Verilog or VHDL, and if you know SystemVerilog, know where the line is drawn between Verilog and SystemVerilog's feature set
      - Regardless which traditional, commonly used HDLs you are comfortable writing in, know enough in the other languages to be able to read and make small modifications to the code
  - New Hardware Description Languages
    - There are many HDLs being developed that are worth looking at such as SpinalHDL, Chisel, Amaranth, Bluespec, MyHDL, RustHDL and others. I don't think people serious about entering the industry should spend too much time learning them at this time, however you should be aware of why they exist, their benefits, their pitfalls, and you should generally keep up with their developments
  - External Interfaces
    - Every FPGA developer should understand and know how to create bus masters and slaves for all the common low speed embedded interfaces such as SPI, UART, I2C, CAN, etc.
    - Understanding of external memory interfaces for SDRAM and DDR is pretty important, many designs need a place to buffer larger amounts of data then can be held in registers or embedded SRAM
  - Memory Busses
    - No matter what RTL designs you do, its very likely you will need to have control and status registers. It is very much helpful to follow a standard so things are compatible with third party IP later down th road. Overall you should know the existence of things like AXI, AXI-Lite, APB, Wishbone, Avalon. You don't need to have experience with all of these but you should know one well, know the features of a of them and just in general know all the major ones that exist. Knowing how to adapt between them without too much overhead is quite important as well.

    - No matter what standard memory bus you use, you should understand the concepts of bus muxing, demuxing, crossbar switch, width adaptation, pipelining, bursting.
  - Data Streams
    - Generally there is one streaming protocol used by everyone and that is AXI-Stream. Making your designs AXI-Stream compliant when you need to move data with bytes or words back to back is critical Many protocols such as Ethernet, UART, PCIe, CAN, Aurora, SpaceWire and many many more can be carried internally in an FPGA as an AXI-Stream. If you follow the AXI-Stream standard for everything it is suitable for then you can have FIFOs, pipeline stages, muxes, switches, hubs and more that are re usable no matter what the protocol is. You can also do fancy things like carrier Ethernet frames over UART for debugging as long as you keep in mind the bandwidth limits
  - Clock Domain Crossing
    - Know when you need to do a clock domain crossing, understand the concepts well. Nothing worse then a faulty multi-clock design. Especially when it randomly works when you test it, then randomly fails some other time
    - Some CDC techniques that you should know include single bit synchronizers, handshake logic, grey code counters, mux hold logic, asynchronous FIFOs to name a few
    - Learn to write IO and timing constraints, you must know the basics constraints such as set_input_delay, set_output_delay, set_false_path, set_clock_group
  - Self-Checking Code
    - Nothing worse then a typo in your code causing things to break. Learn to write checks in your code that will throw an error at simulation or synthesis time if you parameterize code into a configuration you never tested or is incompatible
  - Verification
    - Learn to simulate more, regardless of tool. Larger teams might have separate verification teams but even then you need to run simulations during development. Finding bugs in simulation is generally a lot less time consuming and painful then debugging on hardware.
  - Design Visibility
    - When verification fails or you have external devices not behaving the way you expect then you do sometimes end up needing to debug in hardware. Design visibility, design for test, whatever you wan to call it, its important.
      - Control and Status Registers
      - Performance Counters
      - Integrated Logic Analyzers
- Devices
  - You should generally know what is available on the market on the low end, mid range and high end in terms of size
    - Tiny FPGAs include those from Lattice and Tang
    - Small and medium sized FPGAs are available from Lattice, AMD/Xilinx, Intel/Altera
    - Large FPGAs are generally available only from AMD/Xilinx, Intel/Altera
  - You should eventually gain a feel for the relative performance of the different devices and their generations, maximum usable frequency is something that should slowly become second nature for you to

guess at for all common device families
  - Application Specific
    - Your needs might be unique and you might need stuff like instant on capability (don't need to load from external configuration memory like QSPI flash or similar), or you might need radiation tolerance You should know about the different parts available for these unique use cases from companies like Microsemi, AMD/Xilinx, Intel/Altera
- Device Specific
  - No matter what device you end up gaining experience with, everyone should at least skim all the documentation and available free IP for the chip provided by the part vendor
  - Know what low level primitives are available such as IO drivers, IO serializers, IO delay lines, embedded RAM, clock generators, transceivers and more
  - If you choose to use device specific features, think about how to keep your code portable between devices and different manufacturers, vendor lock in might be ok but it has to be a choice you make
- Tooling
  - It is important to know what tools are available to you, both commercial and open source ones. There are incredible tools available to you for free that can really increase your productivity. When it comes to commercial tools, many are quite expensive but the time savings can be worth more then their cost especially on a larger team. Regardless you should be aware of what exists in the industry.
  - Source Control
  - Every once and a while I run into a developer that does not use source control. It is really a shock when I hear this because source control is one of the most important and productivity improving tool that can be added to a workflow. Learn git at a minimum, spend a day or two following tutorials on how to use a git managed code base. You don't have to be an expert, you can learn over time, but it is something you should start using if you are not.
  - Editors, Editor Plugins, IDEs and Language Servers
    - I think most people find its a terrible idea to edit code in Vivado, ISE, Quartus or ModelSim… spend some time setting up your favorite text editor for HDL and related software development, this is probably the single highest productivity increase you can gain through a day or two of effort.
    - Sublime-Text is one of my favorite editors for SystemVerilog development, its SV plugin is fast and has the ability to jump into modules by clicking on the headers and also has the ability to quickly search for usages of the module you are in. A feature I have found to be hard to find in other places. With that said, I have switched to VSCode myself because I also spend a lot of time writing Python and C++ and I find the plugins for VSCode are better.
    - VSCode is my go to these days, more information is in the Software section.
    - Vim and Emacs are great options, I just feel I don't have the time to set them up or learn them properly, if you do then more power to you, lots of people tell me it is worth the time.
  - Verification
    - Most FPGA developers quickly discovers that debugging in simulation is much faster then debugging in hardware, learning to simulate more, learning to catch more issues in simulation is very important
    - Learn about all the different simulators available to you, commercially and open source. Synopsys VCS, Cadence Incisive, Siemens QuestaSim, Icarus Verilog, GHDL, Verilator. It is especially important

to consider what is available to you. Verilator is a particularly interesting one to me, it is open source and the approach they take, converting Verilog and SystemVerilog code to C++, results in some limitations but brings with it massive simulation performance increase and ease of integration with software you might want to test alongside your RTL that might benefit your work flow
    - Simulators can be difficult to work with, tools that can help you reduce the overhead include things like VUnit and CocoTB
    - There are many popular methodologies in verification, UVM and OSVVM are popular in large companies, if you use CocoTB then PyUVM might interest you. In a smaller team, UVM might not be appropriate and unit tests in VUnit or CocoTB might be less overhead
- Code Generators
    - You should learn to appreciate the benefits that commercial HLS tools like Vivado HLS, Catapult C, Simulink HDL Coder, Pipeline C, and others can bring to your code base. However don't believe all the things the marketing materials say, when you deviate away from what the tool it good at then you can end up spending a lot of time fighting the tools. By using HLS tools, you might be signing up for additional tool incompatibility and if the HLS tool has a bug then you might not be able to debug the output easily
    - Another important thing to consider are system generation tools like Quartus System Builder and Vivado Block Designer. I feel its not good to depend on these tools but they can be very effective when assembling an embedded system within an FPGA design. I think its best to contain the output of these tools in a wrapper that is instanced within a manually written HDL design rather then having the tool generate the top level. This way, higher flexibility and control is maintained.
    - Never underestimate the power of custom, hand written code generators with restricted scope written by your own team. They could be fully custom built in any software language like Python and can be a massive time saver when you need to parameterize a system beyond what something like preprocessor statements and SystemVerilog generate blocks can take you. A common use case for custom code generators is management of register maps and keeping software and RTL in sync. One interesting tool to help with this is the systemrdl-compiler library for Python.
- Static Code Analysis
    - Style Linters
        - Coding style is more important then people give it credit for. On a team, when code is written consistently, so consistently that when people on a team opens code and it looks like they wrote the code it saves a lot of time, things are where they expect it to be and they don't have to look out for oddities on how things are done. They can focus on understanding the logic or getting in and making the change they need to make. Since I work in SystemVerilog, my favorite is SVLint but there are linters for basically every language out there. Try what is available, choose and configure the tools to keep your code base consistent.
    - Design Linters
        - One of the most invaluable things in development are tools that will yell at you when you do something silly. You should be aware that simulators like QuestaSim and Verilator among others have a linting mode that can catch things that can break designs even without simulating or implementing anything. Beyond this there are tools like Vivado's CDC report generator, Bluepearl CDC and

Questa CDC that can find clock domain crossing issues that can lurk in your design, causing failures across temperature, voltage and silicon variation. There are similar tools for reset verification. It is important to explore what tools are available and evaluate pricing to see if you should consider them. Even tools with seemingly high price tags can save you a lot of money, salary for your staff is not cheap.

Equipment

Something I find is a real shame is when I see developers being paid from $20 an hour at the start of their career to upwards of $100 an hour further in their career skimping too hard on their computer setup. This is something that should scale with you as you grow. You are not only doing less but you are learning less per da when you are forced to stare at progress bars.

Early on, sure, a student can make do with whatever laptop they already have for HDL development but even tually spend the money on 2-3 good monitors and get a workstation that is suitable for the load you put on it. FPGA compiles start at 10 minutes and can go up to half a day for large designs. It is not unreasonable to spend a few thousand dollars on a better setup when your time is worth hundreds of dollars per day to your company.

Spend the time looking at your setup and consider how you can improve it once a year. You don't have to go nuts but make the case for an upgrade when it makes sense to. The situation is of course quite a bit different for a hobbiest so use your judgement there, I did all my work, whether it be FPGA development, PCB design, or mechanical CAD on a $400 laptop in undergrad. There is a price point that makes sense for each situation, just make sure you are not unintentionally spending more by trying to spend less. I think this goes not only for a computer setup but for lab equipment and server equipment as well.

Software

There are two major interfaces that surround most FPGAs. One of them is software, at development time as part of tooling scripts, and code generation, and at runtime where software is actively interacting with the FPGA for control, status, debugging or when software is tightly coupled to an FPGA design with interrupts and shared memory. In addition, you might have a analysis environment for logging and analyzing data either from an experiment in the lab or data captured from devices in the field. There is no denying that software skills are super important when it comes to the day to day of an FPGA developer.

In this section, I lump together software tools and software development. I find it hard to seperate the two be cause very quickly many developers find they have needs beyond what their tools provide so they begin to configure and modify them according to their needs.

Development Environment

Most developers take some time before they embrace text based shells and terminals but it is something that is important to get used to. It is easier to make text based tools so that is why many of the most powerful tools are text based. Whether you are on Windows or Linux, this holds true. When on Windows, PowerShell is likely your friend, else an MSYS2 install with many Linux tools ported to Windows might be a good option. If you are on Linux then Bash is probably the default but I prefer ZSH much more.

Whatever your choice of a shell is, spend some time customizing it and writing custom scripts to reduce the impact that repetitive actions have on your workflow.

My ZSH setup is something I only spent 2-3 days working on over the past year or so but has really been worth it.

- Oh-My-ZSH
  - OMZ is a package manager for ZSH and is really a must for keeping things up to date
- Theme
  - I decided to use Powerlevel10k because it shows a lot of useful information at a glance. It can be configured to show which Git branch you are in and if there are uncommitted files so you need to run `git status` less often. Since I work in Python a lot, I also have P10k show me what Python venv I am in.
- Plugins
  - Zsh-autosuggestions
    - You have autocomplete in most text editors so you should have autocomplete in a shell. This plugin suggests commands as you type using your command history. I find it frees me from remembering how to do things in the terminal, saving me from having to look things up or ask people as often.
  - Zsh-syntax-highlighting
    - I find syntax highlighting really helps you see typos and errors just as much as it help with reading a command.
  - Zsh-autoswitch-virtualenv
    - Since I use Python venv a lot, I often have a seperate one setup for each project or repository. Switching venvs is kindof annoying but this plugin does it for you.
  - Zsh-z
    - The `z` command added by this plugin is a real game changer. It makes navigating between folders you have been in a breeze. It does fuzzy search for folder paths you been in. Say you were in `/home/alex/Desktop/projects/littleriscy/` some time in the past, no matter where you are in your file system you can type `z Littleriscy` and it will jump all the way there. Tab completion is also supported if there are multiple matches.

Whatever shell you use, the terminal emulator you use is something that you might want to spend some time configuring. Do some research on this front, its a tool you will use often. Another thing worth looking into is if you want to just depend on your terminal emulators window splitting and tab functions or if you want to use something like `tmux` on top of it. I find `tmux` a must when using SSH to connect to a remote machine.

Your text editor is probably your next most used tool so spend a bit of time sprucing it up. Opinions are high o this front, whatever you choose, spend enough time that you grab all the low hanging fruit in regards to the plugins and configuration that affects your productivity the most. Every once and a while I find someone that writes code without any syntax highlighting, an insanely costly mistake.

My choice is VSCode these days even though I like the SystemVerilog plugin on Sublime-Text a lot more because I find VSCode's library of plugins for other software languages, source control, merging, remote device work is better.

- Language Plugins
  - My favourite plugin for SystemVerilog development is by Eirik. Unfortunately it is quite resource intensive on a large codebase so I have had to turn off some of its indexing options.
  - Another good option for HDL developers is TerosHDL
  - Install language support for TCL, Python, C, C++, as you see fit depending on what you need to do
- The VSCode git plugin makes it much easier to merge changes and navigate through a files history
- Remote-SSH might be a plugin that you consider if you work on remote machines a lot and want to edit code as if the files were local

At some point in your adventures in writing code, I hope you run into source control. One of the most popular is git used in conjunction with GitHub or GitLab. If you don't use source control, consider starting, it might save you one day. It also lets you work on multiple parallel unfinished tasks at the same time, on the same code base. Do some research into how to use the tool effectively, I would at a minimum understand how to create a repository, how to fork, branch, merge, view diff's, push, pull and work with remotes. `git` is a command line tool, graphical interfaces exist if that makes you more comfortable, there will generally be a plugin available for your text editor or IDE of choice. My personal preference is to use git from the command line and use a tool call `tig` which is a text user interface for git that resides in your shell when doing more repetitive operations.

Firmware

If your FPGA or board contains firmware then its really important to understand the challenges of firmware development, performance characteristics and development effort related to doing various things in RTL or in firmware. The divide is a fuzzy and extremely consequential one. Some problems are more easily solved in firmware without performance penalty while some problems must be solved in RTL due to performance limits of a firmware solution. How fine grained the interactions between the firmware and FPGA is also a huge concern for performance and how tightly coupled the code bases will be going forward. Whether the FPGA team does some or all of the firmware development or if a separate software team handles the firmware development does not change the fact that keeping things like register maps in sync or testing and releasing known firmware versions with know FPGA image versions is extremely important. This is something that needs to be planned and possibly automated.

Runtime Environment

Your runtime environment for FPGA and software development is of course highly dependant on what you d Most FPGA developers will generally do a lot of their work in a simulation environment but usually at some point they need to work with real hardware.

In my experience, many teams gravitate towards Python because it can be interactively executed in somethin like IPython with ease. It is easy to learn and even non-software developers know enough Python to be functional in it. With that said, a lot of embedded software is written in C or C++. This means you might be duplicating a lot of work if you develop hardware support in Python as well. One option is to look into tools like Cling which allows you to execute C or C++ in an interpreted environment.

Whatever your software language of choice. Something important to put together is an abstraction layer for the interface you attach the FPGA to the host system with. A well written abstraction layer allows you to use the same software no matter if you use JTAG,SPI, UART, CAN, Ethernet, PCIe or whatever protocol. Usually a a minimum you will want to put together enough software to read or write any register in your attached FPG at a minimum.

With some effort, using either something like SystemVerilog DPI, Verilator, CocoTB or another co-simulation framework you can attach your runtime software to RTL running in simulation. The same code base can hope fuly also be repurposed for hardware in loop testing.

Performance and Benchmarking

They say if you don't measure performance then you don't care about performance. It might make sense for you to consider how you can measure performance of software in various ways. Some tools to consider if you write in C++ would be tools like KCacheGrind, and Tracy. Tracy is a tool popular in the game development industry however I have found it extremely adept and useful at profiling high performance digital signal process ing code, especially when multithreaded.

Data Analysis Environment

Something worth considering is a data collection and analysis environment. If boards are deployed in the lab or field and you have connectivity, it might be worth considering tools like influxDB and Grafana to store and visualise telemetry. This is something very situational but I wanted to mention this to get the ideas rolling.

System Architectural Design and Integration

Architectural is possibly the most important part of any system design. Any significant achievements as part c any individual sub-system in a larger system is immediately negated if the total system performance, reliability, or extensibility and flexibility is sub-par. An FPGA developer should ensure any effort they put into RTL de

velopment is of the correct performance level, feature level and reliability such that it matches the reset of th system and is not over engineered or under performing.

I think the most important thing to consider while designing with or for other stakeholders in a design is that more often then not, the other people you work with don't know what an FPGA can do, what is difficult to do on an FPGA or outright impossible. It is your job to help everyone figure out how an FPGA fits into the bigger picture.

High Level Design

While this is something that varies between jobs, things to think about when performing architectural design with the rest of the team include…

- What do the other teams need the FPGA to do for them to do their job effectively?
  - Do electrical teams need toggles for IOs or status reports so they don't have to dig into the system to know if something is working?
  - Do RF teams need to capture some data to look at to evaluate performance of a receiver? Do they need internally generated signal to evaluate the performance of a transmitter?
  - If there is a dedicated systems integration team discuss what they need to be able to do in a fully assem bled system in production or in the field.
  - In production does the FPGA need to support some kind of built in self test?
  - Does each unit need some way to be individually calibrated?
  - If implementing peripherals for a software team, what features do they actually need? Is high performance a must? Determinism?
- If the FPGA is integrated into a larger system, has everyone thought about situations where a device or pe ripheral attached downstream from an FPGA has to be accessed by an upstream system?
- Is the FPGA capable of operating with a partially assembled system? What if some components are swapped out for alternatives?
- Do we need the ability to do in field upgrades? Does the system need to support in field debug and diagnos tics? If there is some sort of field service team, what additional functionality do they need?
- If determinism, high throughput or low latency is required, is everyone involved in the critical path aware c what needs to be done in each of their domains to achieve the target goals?
  - If determinism is required, down to what time scale is synchronisation needed? Do multiple systems need reference clocks, synchronisation pulses?
- Does the system have to somehow operate in a degraded state in the case of partial failure?
- Does the system need provisions for security? Cyber based or physical/tamper resistance? If so, the entire system must be designed with this in mind.

If the designs are very complicated, block diagrams are a minimum to make sure everyone is on the same page Shared document that are reviewed and controlled might be a must. For larger designs, a requirement management tool might be needed. Formalised description of the system and how it functions, the concept of op-

erations might be very important to put together. When changes need to be made, a formalised engineering change order system might be needed to make sure everyone affected is aware what is changing.

System Integration

Systems integration is one of the most difficult things to get right. Anyone who has plugged multiple things in together physically or figuratively has probably observed show stopping bugs, incorrect behaviour or something else unexpected. Prepare to be available, maybe even prepare documentation on how to observe and/o operate a sub-system written in an understandable way for people without the same knowledge and background as you.

Delays in delivery of sub-systems or feature sets is also something that happens a lot when multiple teams de liverables converge, the team often needs to have scheduling and those that need to know should be informe of schedule changes.

Electrical

The other major interface that surround most FPGAs is the peripherals attached to it on the circuit board. Often, it makes sense to have a board custom designed for your application either by your team or through contracted designers. One of the most important things is to recognise what the challenges are in circuit board design, bring up and post-manufacturing debug and modification. My past experience involves a lot of circuit board design and wiring harness design. Experience doing this kind of work is generally not required for an FPGA developer to have, however, it very valuable to have the skills needed to support the team desigr ing the circuit boards with the FPGAs.

It is important to recognise that the typical board designer is likely not use to designing a board with an FPGA on it. There are some unique aspects about FPGAs that might be worth keeping in mind or mentioning to you board designer.

- Pins
  - Pins are mostly re-mappable and pin swapping can be used to ease routing
    - Keep in mind when selecting pins if you need to use specific IO standards, speeds or voltage ranges, FPGA IO is highly configurable, to the point where it is super confusing, the IO standard support list i massive and hard to follow, you have to follow up on every IO pin used because this is not typical of general purpose ICs that board designers are use to working with
    - Care must be taken with clock inputs, not all pins are capable of feeding the clock network
      - Which clock capable pin being used is also potentially a performance concern if a specific PLL or clock management block must be used, the routing distance can introduce skew, it is application dependant if this matters at all
    - When differential inputs are used single ended, often FPGAs prefer or require certain that the P or N input is used

    - Differential clock or data polarity can usually be swapped but not always
    - Pin selection for related or unrelated logic has impact on routing congestion and final achievable FPGA performance
  - On Die Termination
    - Some FPGAs support on die termination for some standards, ODT is more effective and often advised for high speed interfaces. Less components on the board. I've met a lot of board designers that think resistors on the board is better but this is really not the case. If ODT is supported for the specific standard you are using, it should be used
  - On Die Pulls
    - On chip pull ups and pull downs are a different story, they might not be sufficient pull strength and some attention should be given when deciding to use them or not
  - Drive Strength Control
    - Drive strength control is something that can have quite a significant impact on impedance match of a driver to a trace or transmission line. FPGAs often do not need series resistors on lines for matching impedance if everything is designed properly. Overall lower power consumption and lower part cour can be achieved.
  - Byte Lanes
    - When pins are used in parallel, there are additional concerns with how clocking for source synchronous interfaces works. Clocks have to be either paired with specific bits of the interface or clocks car only travel up 1 or 2 banks depending on FPGA architecture
- Source Synchronous Interfaces
  - Source synchronous means data transmitted with a clock that must stay aligned to data. There are various source synchronous interfaces, SPI, MII, RGMII, "CMOS", DDR memory interfaces and others. It is important to note what capture technique is used on the FPGA side. Just because an interface appears source synchronous does not really mean it is captured the same way. SPI is often but not always oversampled while MII, RGMII and DDR are all usually captured synchronously directly using the clock.
    - Alignment of clock and data is generally always required for source synchronous interfaces however things can get quite complicated. Often things like RGMII PHYs have ways to introduce skew througl configuration registers accessible over MDIO (or they might have built in non-configurable delays) which could make it possible to delay data or the clock. Usually the multiple data lanes are skewed to gether which means those need to be length matched together.
- High Speed Serial or Channel Bonded (Parallel) Serial
  - It is important to know if the protocol is Single Data Rate (SDR) or Double Data Rate (DDR), the nyquist frequency of 1 Gbit/s SDR is 1 GHz while the nyquist frequency of 1 Gbit/s DDR is 500 MHz.
  - LVDS
    - LVDS even at low data rates (10-100 Mbit/s) can be incredibly challenging to close the link on,
    - Length Matching
      - Intra-pair length matching is always important to reduce received and transmitted interference
      - The various types of LVDS interfaces makes for vastly different requirements when it comes to length matching. When it comes to high speed (0.5-1.5Gbit/s), length matching between clock anc data pairs, or between data pairs might be pointless, it depends on if the interface is statically cap

tured or dynamically captured. If dynamic capture is used on an interface that has more then 2 data pairs then there is an extra detail important, is the entire parallel bus statically captured together or are each bit captured separately and an alignment pattern is used to do lane alignment?
    - Static capture
        - In statically captured interfaces, length matching between all pairs is incredibly important
    - Dynamic capture but multiple lanes statically captured together
        - Length matching between clock and data pairs is not important but length matching between data pairs is incredibly important
    - Fully dynamically captured
        - Length matching between all pairs is not that important, within limits of what the FPGA can compensate for using either internal analog delay lines or bit slipping if used
- Transceivers (Multi-Gigabit Transceivers)
    - It is important to know what transceivers do for you when it comes to data reception in challenging environments, it is very common for a 10 Gbit/s transceiver link to be significantly less challenging then a 1 Gbit/s LVDS link just due to what a transceiver is able to handle in terms of signal integrity impairments. Which techniques are used is dependant on both the transceiver itself and the selectec protocol.
        - Transmit pre-emphasis can be applied which boosts high frequencies of the transmitted signal to make up for some of the low pass response of the transmission line
        - Receive continuous time linear equalization (CTLE) is similar to transmit pre-emphasis, it boosts the high frequencies of the received signal to make up for some of the low pass response of the transmission line
        - Receive decision feedback equalization (DFE) is a significantly more capable equalization technique when compared to CTLE, it has the ability to cancel reflections in the transmission line within the DFE's symbol length which is dependant on the length of the transmission line and symbol rate of the link
    - Length Matching
        - It is important to understand the different types of length matching of PCB traces
            - Intra-pair length matching is always important to reduce received and transmitted interferenc
            - Inter-pair length matching might not be so important, it depends on the protocol. Often for something like PCIe, when multiple lanes are used, lane to lane mismatch is handled by elastic FIFOs in the transceivers operating in receiver channel bonding modes
            - Reference clocks pairs often do not need to be length matched to data pairs because they are only used as a frequency reference, the clock and data recovery (CDR) circuit in each transceiver's receiver recovers the transmitting sides actual clock frequency and aligns the phase
- Power up delay
    - Output pins will not be put into the correct reset state on power up because the FPGA programming takes time to load unless the FPGA is an instant on device. This can have significant complications when it comes to external devices, power up circular dependencies and unexpected behaviour on power up
- Power sequencing

  - Input pins might have to be biased to the correct voltage before power on of the FPGA else the FPGA might enter an unknown state
  - Power sequencing done incorrectly can result in improper operation or excessive power draw after power on
- Power and cooling
  - Average and peak power requirements is something very tricky when it comes to FPGAs because it is highly dependant on what is deployed on the FPGA. It is not uncommon for one FPGA chip to draw 10s of watts in one application but the exact same FPGA chip draws upwards of 100s of watts in a different one. Rough modelling is usually provided by the FPGA manufacturer in the form of data sheets with equations of Excel worksheets. More accurate modelling is provided by the FPGA software tools upon successful full design compilation however the model depends on toggle rates supplied by your team which has significant impact on the actual power draw. The toggle rates could be estimated from logic simulations however this might be too much effort for most FPGA teams to do.
  - It should be noted that peak power requirements and transient currents might far exceed the average and margin must be included. Generally this also means power supply noise, ripple and stability measurements only matter when an FPGA is under dynamic loading conditions.

Design Scope

Design scoping is a massively important step that is very multidisciplinary and an important step as a boards design goals are developed and prioritised. There are competing design goals that you should keep in mind when helping architect a custom PCB as a solution.

1. The designer responsible for the board has immediate concerns with component size and availability, they likely want to design using components already used in other boards designed in the past, they also have concerns about ease of manufacturing, bring up and production testing. They possibly have to work with a mechanical designer for thermal, form, fit and function reasons.
2. If software or firmware is involved in the board then its likely the software, firmware and FPGA team have long term ownership of the board in terms of maintenance and upgrades over time. Planning for future features that are desired to be implemented on the board is important. At this time, planning how in field FPGA or firmware updates will occur might also be important.
3. All teams should be concerned about future revisions of the board being similar enough that future revisions can run the same firmware, software and FPGA images. Techniques to make this possible might include identifiers that can be read by the FPGA or by firmware that identifies the revision of the board so any differences can be handled at runtime.
4. All teams should be thinking about the signals on the board that need to be easy to be probed, these should be brought to test points.
5. Everyone needs to note what debug interfaces they need on the board when its in bring up state and also when the board is fully assembled in enclosures or whatever it might be.

It is critically important to discuss what FPGA, software and firmware functionality should be available for board bring up. It is imperative that the entire team is prepared to control and read status of specific things on the board so bring up can be quick, issues identified and characterised easily.

Production and longer term maintenance of a system involving a custom board is a challenge. Have a plan for logging faults along with a way to identify FPGA, software versions, board and assembly unique identifiers and other information relevant to discovering the real issue when one board starts acting up in a customers hands. Have a plan for retrieving these logs and also possibly have a plan on how remote diagnostics will be performed in the field.

Schematic Review

Schematic review is something that should be done by at least one person of each discipline who needs to touch a board. Divide responsibilities in a smart way depending on who has the highest likelihood of actually finding an issue. Ask the electrical designer to leave notes about things they are unsure about on the schematic. Find a way to verify that the expected change requested was made so at the end of layout review, there is proper followup without too much overhead.

Something I find most board designers don't know is that FPGA tools will throw compilation errors if pins are connected in ways that are not supported. Creating a sufficiently accurate periphery but stubbed out FPGA design and compiling the design to see if your tools accept the IO configuration is one of the most valuable tasks during the board design process. Ensure that before a board is released for manufacturing, any pins moved are also moved in the FPGA design and another compile is performed to validate the changes. Doing this early also allows you to start some of the FPGA design while layout is happening, often it is this point where I find some errors in the architecture and it allows me to ask for changes before the board is manufactured.

Layout Review

Layout review is the perfect time for the FPGA, software and firmware developers can look at the board and ensure signals they thing they might want probed during bring up are easy to probe. Asking for test points or handful of signals routed to a debug header at this point is usually not too bad.

Bring Up

Bring up is one of the most painful parts of the board design process. Arrive prepared with minimal FPGA images, firmware images and software support. Prepare to be available, preferably in the lab with the electrical designer. It is highly likely that multiple issues will be encountered during bring up and many issues across multiple disciplines will occur at the same time, possibly masking the real issues. It is important that everyone that needs to be involved are available and have a full picture of what the current state of the system is, if

problems are troubleshooted by only one or two people with only a sub-set of the disciplines required to find the real problem then bring up will really be a slog.

When issues are found, everyone involved needs to understand the long term impact of electrical, FPGA, software or firmware workarounds. It should be understood that some workarounds are more costly then proper fixes. Put in the effort to understand when it makes sense to ask for a board revision or if it makes sense to pu an ugly hack into a code base.

Automation

Something very important for FPGA development and software development is automation. While there are many specifics about FPGA development where automation is different, a lot of useful tools and techniques can and should be borrowed from the larger software industry.

One of my colleagues once said, if you don't automate testing then you will lose features over time. I have found this to be the case time and time again, especially in a code base where code is reused across a lot of projects. Tools like Github Actions, Gitlab CI, Jenkins, and Docker can be a real help in constructing a framework for automated simulations, unit tests and hardware tests.

Another area where tool automation is important is anywhere where a designer has to do manual steps such as clicking through a GUI to create a project or configure components. Most tools have the ability to be scripted. It is important to script project creation, IP generation and builds to ensure consistency.

When it comes to supporting other teams, designing systems such that continuous testing of what you are de livering, adding self test logic and more might be very important. If the team decides to do software in the loo testing or hardware in the loop testing and the setup is available for continuous use then the test should be automated so any issues created through continuous evolution of software or FPGA code can be tested at the system level.

Code Generation

It is pretty common that a code generation tool can help reduce repetitive work on an FPGA team. Developin your own custom code generator can be a big help. They do not have to be complicated or too generalized. Working around the limits of some RTL languages using a custom, more advanced pre-processor is a common thing to do. Writing the same bus slave over and over again for each module could possibly be automated. For more advanced code generation (bordering on high level synthesis (HLS) territory), you might want to look into compiler theory and graph theory if you are going for something custom.

FPGA Applications and Related Skills and Knowledge

At the end of the day, an FPGA is an implementation platform. It is meant to do something. At some point the logic you implement will involve concepts and algorithms from other domains of expertise. Whether you choose to also become an expert in these other fields is up to you. However, basic knowledge in the fields you colleagues are skilled in can be very important to ensure you are able to design a system with good overall performance.

Something I will also hammer hard is that you should consider learning to bridge knowledge between fields that are traditionally taught separately. I suggest that you don't fall into the trap of "I don't use most of what I learned in school", I don't think there is any course I have taken that I did not eventually apply to my work or hobbies eventually. The various areas of engineering is often more similar then different. My favourite example of this are analogical models, a system of springs, masses and dampers can be converted into a resistor, capacitor and inductor circuit and analysed with the equations commonly used to analyse electrical circuits. There are many fields with sometimes non-obvious overlap. Learning to see the overlap lets you apply seemingly unrelated knowledge and skills to a new situation, letting you become omnipotent. Seemingly being able to understand or solve problems you have no obvious experience solving previously. I love working in multidisciplinary teams so this has become invaluable to me.

Working with other Disciplines

When it comes to the logic and algorithms implemented on the FPGA. One of the most important things an FPGA developer needs to do is to evaluate how difficult something another team or person of another discipline is asking you is to do. A few things an FPGA developer has to worry about is how long something will take to implement, how big of an FPGA will be needed to implement, what throughput is required, is the system latency sensitive, or if a system requires determinism. This is something where people of different expertise will ask you for and you are expected to be able to understand enough about the problem they are trying to solve to make a decent estimate. Usually only the FPGA developers know if a problem is hard or easy to do on an FPGA and if its even suitable for FPGA implementation at all… sometimes the answer is to use a micro-controller or digital signal processor.

Another thing to keep in mind is the scale adjectives mean to people working in different disciplines, something fast in an FPGA is a few tens of nanoseconds while something fast to someone working on a controls system of a mechanical system might consider tens of milliseconds to be fast. It is important to properly understand peoples asks and wants and translate them into things you either have to worry about in your FPGA design or not. It is not uncommon for a designer of a system to be extremely worried about throughput or latency in a system, have you over optimise something in an FPGA only for you to realise much later that the difference is negligible in practice when looked at with a big picture perspective.

Digital Signal Processing and Software Defined Radios

FPGAs are often used to perform digital signal processing or implement software defined radios. Being able to estimate the size and generation of FPGA required to achieve a certain required throughput can be importan

It can be quite important for an FPGA developer implementing DSP algorithms or making an SDR to understand the following concepts at least at a basic level.

- Sample rates and what it means for bandwidth
- Sample rate conversion and how to accomplish it in an FPGA
- Fixed-point math and how to effectively convert algebraic equations or floating-point software models of signal processing algorithms to fixed-point
  - Sample word-length (how many bits per sample) and its impact on dynamic range, quantisation noise and FPGA area utilisation
  - Understand the impact of quantisation noise on total system performance and if it matters
- IQ sampling and complex numbers as it applies to signals
- Concept of digital up-conversion and down-conversion and implementation of numerically controller oscillators
- Time domain and frequency domain representation of signals and when they should be used. Implementation of Fast Fourier Transforms and its inverse
- Finite impulse response filter and cascaded integrator comb filter implementation techniques in an FPGA

These are just the basics that would be handy when discussing DSP with a colleague more familiar with the subject. If your job involves more intensive DSP then the following topics could be stuff you should know about.

- Phase shift keying and other forms of signal modulation
- Implementation of numerical frequency and phased locked loops and Costas loops
- Clock and data recovery algorithms such as Mueller-Muller (MM) or Gardner timing recovery
- Forward error correction algorithms, constitutional encoders, Viterbi decoders to mention a few
- Blind and adaptive equalization techniques, why they are needed and how they are implemented in the digital domain

Something I want to bring to your attention is that concepts such as adaptive signal processing and cognitive radio share a lot with AI and machine learning. Adaptive filtering is actually quite similar to something like a multi-layer perceptron. If you work in this field, do some reading into where these fields overlap.

Radio Frequency

If your job involves interacting with radio frequency hardware or a team that designs it with you then often you will be asked to interface your FPGA with things like ADCs, DACs, frequency synthesizers, power amplifiers, low noise amplifiers, RF switches, RF power detectors and more. You will also often be asked to collect data from these devices directly or capture data in your FPGA for your RF team to perform measurements or generate calibrations.

Here are some things an FPGA developer might need to know when interacting with RF hardware or an RF team.

- Understanding of dynamic range and where analog frontend automatic gain control might be needed, understand where digital automatic gain control might be needed so logic internal to an FPGA behaves as expected
- Understand the concept of noise figure, signal energy, signal noise, signal to noise ratio and how to measur these things within an FPGA
- If working with ADCs and/or DACs, understand the meaning of spurious free dynamic range (SFDR), sample jitter and their impacts to maximum achievable signal to noise ratio (SNR)
- Be able to work and discuss design metrics in decibels, understand the limits of channel isolation and leakage between channels that you might observe on the FPGA which might result in odd behaviour or confusing data
- Pre-emphasis and digital pre-distortion techniques and what demands these techniques place on the FPG design

RF designers often have a lot of fancy equipment. I think just learning to use the equipment can teach you a lot. If there is a particular instrument used often in work related to what you do on an FPGA, have a dig through the manuals and white papers by the manufactures of RF test equipment. Instrument makers educat their customers a lot, often the RF knowledge is explained in very practical ways by these equipment manufac turers. In many cases you will learn how to re-create the digital portion of a receiver or transmitter that is in an instrument, in your FPGA from the user guides of test gear.

Networking

Pretty quickly when working with FPGAs and host machines, things like UART and SPI are not able to provide the bandwidth required between the systems. Often, the next best choice is Ethernet. It is reasonable easy to integrate an Ethernet subsystem in an FPGA using something like Verilog-Ethernet from Alex Forencich but pretty quickly you will need more background and tooling. Note that what I discuss about Ethernet actually applies to other protocols such as CAN bus, PCIe, SATA, SpaceWire, USB, wireless packets, forward error cor rection frames and others which are also framed and streamed as bytes or words.

I think every FPGA developer should be able to write a UDP and TCP client/server in a software programmin language of their choosing. Past that, they should be familiar with the lower level framing, MAC addresses, IP and other details in the TCP/IP and UDP/IP protocols. An FPGA developer should also know what ICMP and ARP is. On top of that they should probably be familiar with tools like Wireshark and tcpdump. In addition, at some point if you work with any protocol that involves error checking then you will probably encounter cyclic redundancy checks (CRCs) which you should quickly discover can be computed efficiently using linear feedback shift registers (LFSRs). Note that Wireshark can be used for non-Ethernet protocols like USB, PCIe, SAT and more through plugins, the pcap file format is simple and you can dump frames from simulations or off an FPGA for analysis in Wireshark.

In most cases, if you work with streaming data with or without framing you should quickly notice that in an FPGA, all of these protocols can be carried on AXI-Streams. Modules such as FIFOs, multiplexers, hubs, switches, header/footer extractors, header/footer inserters, frame capture buffers, frame replay buffers, DMAs and more are actually all often reusable between protocols.

Control Systems

I sometimes argue that control systems is the same field as digital signal processing. The difference is while feedback loops in signal processing stay within the signal domain, the feedback loops in control systems ofter traverse between the signal domain and the physical world. If you are at all focused in either field, try to learn from both fields simultaneously, you will find a lot of overlap and begin to see instances where prevalent techniques in one field is unheard of in the other, a true breeding ground for innovation. A phase locked loop in communications systems is a control system and can be analysed as such even if it is not always done the sam way.

As an example, Kalman filtering is something very common in control systems. It is not often applied to DSP but can definitely be applied.

Image Processing

Something you should realise quickly is that image processing is very much 2D digital signal processing. As such, there is a lot of conceptual overlap of image processing and DSP. The implementation challenges are vastly different but because of the conceptual overlap, novel research and ideas in one field should not be hel in isolation. I don't do much image processing so I won't go into too much detail here but just realise that 2D convolution is quite similar to 1D convolution and auto-correlation you might be use to doing in DSP apply th same way in image processing.

AI and Machine Learning

It is important to realise in what ways AI is affecting the industry (all industries) and how it can help you. It is also important to recognise when its used as a buzzword for new software or hardware but is really just an evolution of things you are already use to working with. An important point to notice in AI and machine learning is that a lot of it boils down to matrix multiplication accelerated in custom hardware. This means that a lot of AI accelerators can also be repurposed for image processing or digital signal processing. Hardware architectures such as systolic arrays common in AI acceleration is applicable to many different applications. In some cases, AI accelerators are also suitable for high speed network packet processing. It all just depends on the actual hardware architecture and software tooling supporting it.

A lot of money and research effort is spent on AI, therefore there is a plethora of good ideas hidden in papers. Have a look at the hardware architecture and software support put into AI accelerators and explore how they acheive incredible throughput. Due to the similarity between AI, machine learning and things like adaptive fil

tering and cognitive radio, skim some papers in this field from time to time to see what can be adapted over to another field.

Compilers and Graph Theory

You might think that a compiler course is only for software developers or that graph theory has little real applications in FPGA design. However, many FPGA tools opperate on graphs just like software compilers do. Many HLS tools such as Vivado HLS are built on top of traditionally software compiler frameworks such as LLVM.

If you are developing a code generation tool, modules, connections and nets can be represented as nodes and edges of a graph. Even if you do not use much mathematics related to graph theory, you still have to traverse this graph.

If you look at a tool like systemrdl-compiler, the register map gets transformed from the SystemRDL input into a graph that you traverse for code and document generation. With code generation tools, there is no reason to stop with just generating the FPGA side stuff, C headers, Python libraries, and more can also be generated from the same data structure.

Within an FPGA synthesis or place and route tool, the circuitry is represented as graphs and transformations are performed on it for optimizations and fitting logic circuits into lookup tables.

Learning, Teaching and Discovery

Team Driven

To be team driven, as a quality, is often said without an explanation. I think being team driven to various amounts is one of the true characteristics of various levels within an organisation (junior, senior, principal, staff, director, vice president, chief executive, whatever it might be…). Moving up is about multiplicative impact. Think about how you can multiply the impact of others on the team. Think about how you can help prevent the same mistake from occurring again in the future, in a project, on a team, or across the organisation. Often, getting more done does not necessarily mean you have to do more work but help everyone around you get more done with less effort. Tooling, infrastructure, automation, retrospectives, documentation, process, mentoring, and innovation are all part of scaling a team.

I think a persons role on a team changes very significantly over time as they become more senior. More junior developers should try to bring ideas and feedback to a team without feeling like they will judged, I think they should not be afraid of saying something wrong. More senior developers need to foster a welcoming environment for ideas and feedback while using their experience to guide discussion. The senior developers should directly solicit feedback from junior staff and help them articulate feelings, ideas or feedback that are not fully

formed in their minds yet. Most important of all, senior staff should be responsible for making a decision whe discussions become circular or start to feel pointless, they should get the team to commit and ask the team to execute with confidence. If a decision ends up facing challenges later, senior members of the team are respon sible for owning the original decision, plot a path to correct and guide the entire team to a satisfactory delivery.

Being Observant, Critical and Analytical

Working in a team or organization for a long time can sometimes be like a lobster in a pot of slowly heating water, issues and problems arise but no one notices, over time they become the norm and no one realizes changes need to be made. Avoiding this requires that team members are encouraged to speak up. All feedbac should be taken seriously and any reason to take or not take action should be explained to who gave the feedback.

They say hindsight is 20/20, therefore retrospectives should be held on a regular basis. Not every complaint or idea needs to be executed on but every complaint or idea should be considered, ranked and actions the nex time around prioritized.

I should note that not everything that seems like a poor decision in the past was actually a poor decision, the situation change. It is important to understand why something was done a certain way in the past, and why it might not be sustainable for the future. Being critical about the past is not about negativity or blame. It is simply because what worked in the past doesn't necessarily dictate what will work in the future.

Being Mentored and Mentoring

To be taught and to teach. Both things that will never end in your career. Learning and teaching well are skills that you have to actively learn.

Early on in your career, the expectation is that there will be a lot of things you do not know. A lot of people say that you will have a lot of questions. I find that this is actually not always the case. Formation of questions when you don't know is actually something that you actively have to do. While working, think of questions you can ask people who know more then you, prepare questions for your 1:1's, for meetings, make notes and follow up if the answers don't make sense to you even after doing some research on your own. You can and should ask a lot of questions, as long as you don't ask the same ones multiple times. When I am mentoring someone, the most anxiety inducing thing for me is when they don't ask me enough things, when this happens I tend to have to actively plan to check in on someone I am responsible for.

Growth as an FPGA developer often means getting more work done in less time then it did earlier in your career. One way of doing this is to improve the output of your other team members. This can be done through mentoring them. A side effect of mentoring that you will quickly discovery is that the people you mentor will often teach you just as much as you teach them.

One of the most important things to do when mentoring is knowing when to tell your mentee how to do some thing and when to let them struggle and figure things out on their own. Some types of struggle such as with troubleshooting and debugging is necessary to some degree. However, if you see your mentee having trouble with stuff that does not teach a valuable lesson like simply not knowing where internal documentation is, how something should be designed to reuse work done previously at the company, or any organization specific thing that someone new to the environment just won't know you should stay on top of telling these too your mentee so they don't walk an unnecessarily painful path. Something I find is very helpful is showing a mentee how to use infrastructure and tools so they are not fighting with nonsense when trying to learn to develop or debug better. Another helpful thing is providing coding style guides with explanations on why specific rules ex ist and what problems and bugs they prevent. A great example of this is putting `default_nettype none` at the top of every SystemVerilog or Verilog file to avoid implicit nets being declared when a typo is made.

Another thing to do as a mentor is to check in often so if something is done in a way you did not expect, the mentee does not have to throw away as much work and do too much rewriting.

Something to look out for as a mentor are tasks that are great vehicles for learning for your more junior staff. By this, I do not mean grunt work. I mean work where junior staff can go through the steps of bringing up an entire system in a way similar to how more senior staff had to do when the team just started. Things like writing a board support package from scratch, writing simple bus slaves, putting together a memory bus architecture, writing an SPI or I2C controller or other things you expect a senior developer to do quickly in an afternoon might be good to give a junior developer as a task even if it takes an extra few days. What you can do to speed up the process and get the result you want is to have senior and junior developers pair code.

If pair coding is selected as a teaching avenue then it should be done carefully. The mentor should not write too much code. In my experience it is better to only leave some pseudo-code in comments or just TODO comments while explaining what I expect the mentee to put together. Sometimes the comments just list the locations or file names of code where something similar was done in the past and serves as a good example. Sometimes pair coded sessions can be very short, if some debugging needs to happen then they can be an hou or two long as needed but in many cases, a 10-15 minute check in is all that is needed to really increase the rate someone learns. Overall, I think is important to give less instruction sometime to allow your mentee to be creative with their solutions.

Finally, learning to mentor also requires some mentoring. Be sure to let your mentees know they can ask othe people on your team for help. When you have more staff, assign different people to mentor eachother. Everyone needs an opportunity to try and learn.

Giving and Receiving Code Review

Code review is one of those things that has a lot of value on a team. It is not just about maintaining code quality, it is one form of mentorship, spreads knowledge, encourages discussion and more. Teams should be conscious about how they do code review.

Choosing to expose more developers to different portions of the code is important however if someone reviewing has absolutely no idea what the purpose of the code is or have context of the overall architecture the they might not be able to perform a good review in reasonable time and effort. In some cases, if you need mor people to get acclimated to a certain part of the code base, multiple reviewers can be assigned, the person newer to the code base should review first, then someone more familiar can review before everyone goes through the comments again.

It is not necessary that someone more senior to who wrote a piece of code perform a code review. I don't thin in many cases it matters. Everyone makes mistakes and anyone can catch them. Multiple reviewers can be assigned if there is any worry that issues can slip through. Performing some reviews is required to learn how to review well so give people the opportunity to try.

Experiments

Running small experiments to try new techniques or tools is an important thing to do from time to time. If you have been working on something for a while or you are on a large team, chances are you have a large code base that makes it hard to make major changes to. What can be valuable is making small temporary code base to experiment with new tools or a different way of doing something. After trying something out, you can better decide if it is worth the effort of integrating the new ideas or tools into your main code base. Another major benefit is you now better understand what you have to do to the large code base to properly integrate the new thing. Integrating something new to a code base can be very time consuming and challenging. Often, you have to convince other people to see the value in what you are looking to add. The experiment can serve as a demo for this purpose.

Over the last few years, I've spent a handful of days and weeks trying out new tools like CocoTB, VUnit, Verilator, SVLint, systemrdl-compiler or new versions of Vivado or HDL simulators where updating is often not so simple due to tool bugs and the need for new workarounds. Every time, it was a good idea to try the too before suggesting it to the team at work. Some of what I discovered during these experiments has resulted in better productivity and lower chance of introducing a bug into our code base. Other times, I discovered that a tool is not compatible with the way we work, not as useful as I expected, or the integration too painful. In either case, only experimentation allowed for this insight.

Leading

Leading is often about guiding your staff towards a solution that is just good enough to get the job done. It is sometimes about cutting circular discussions that just need an executive decision. Other times it is about making sure that open conversations happen between the right people when decisions or changes need to be made.

Its often required that a leader have a greater understanding of the wider reaching impacts of the work individual contributors do then the individual contributors themselves. This is so effective task planning can be

performed and the correct person can be assigned to a task. Far reaching architectural consequences need to be understood by the leads. The lead should have the experience to evaluate time estimates and potential timing and schedule conflicts by gut feel in some cases. Overall, the lead should be responsible for ensuring there is enough due diligence performed and deliverables are delivered on time and In budget.

It is often about choosing the corners to cut, feature requests to push back on, bugs that need to be tagged as won't fix. Sometimes having the backbone to say no to a request is very important. Other times it is about being flexible and finding ways to satisfy requests in a way that does not make it feel like the ground is moving below the feet of your individual contributors.

Leading is also about making sure people are happy working on the things they work on. Keeping in mind the longer term career goals they have in mind.

Additional Education

Masters Programs

I wanted to touch briefly on masters programs. Specifically about masters programs that involve writing a thesis since that was what I decided to do at some point. First and foremost, In my opinion, I don't think anyone should go get a masters with an expectation that with the degree there will come big change, if you don't like the work you do or you can't find a job, or if there is any other problem, then I think you need to look at everything from a different point of view. There are likely other issues you need to figure out or work on, getting a masters is possibly part of the solution but a graduate degree on its own is not an answer. I think what you could expect from a graduate degree is a bunch of skills and experience that will gradually shape your career over the coming years after graduation.

I wanted to talk about what I learned during my time doing a masters, which was not at all what I thought I would learn.

One of the things I realized I learned during my graduate degree years after completing it is that I gained the skill and confidence to push forward on a problem even when all the things I am reading does not make sense to me or my experiments are all failing and I feel like I am spinning my wheels. I gained the ability to take on challenges where there are more unknowns then knowns.

Another thing I learned is that discussing problems with people even when they don't have much expertise in your field is incredibly helpful. Insights adapted from outside your field or industry is one of the most important things to me. Ever since, I have tried to build an environment around myself that is not an echo chamber. Disagreement and discussion is one of the paths to innovation and avoidance of overly strong biases.

Resources

Everyone should have their own collection of resources they learn best from, I happen to not like using textbooks much but that is because they don't work well for me. In anycase, whatever your favourite resources might be, I encourage anyone wanting to advance to curate and sort their resources so you can refer to them when you need to.

As with any resource, including the one you are reading now, I think you have to actively pick and choose wha you learn from each one. You should be opinionated about the things you consume, no expert has all the answers, no person is 100% correct. You should decide on what advice to follow and what advice to ignore.

I think it is very important to learn from many places, not just FPGA focused resources. It is important to learn about and recognize the challenges associated with the design, development and maintenance of everything around you. You should get used to adapting knowledge to FPGA development especially given how scarce good FPGA resources are in general when compared to the larger software industry for instance.

Some of my favourite resources are listed below with some explanation on how they helped me.

- Chips Alliance: https://www.youtube.com/@chipsalliance8321/
  - Part of The Linux Foundation, lots of good presentations about various open source projects and tooling relevant to FPGA development. Keep your eye on this, open source tooling is getting a lot better in the RTL development space and you don't want to miss out when free tools overtake commercial ones in certain ways. Traditionally software companies like Google, Facebook, Amazon, etc. are becoming incredible interested in ASIC and FPGA development, as such they have brought money and other resources into open source projects.
- EEVBlog: https://www.youtube.com/@EEVblog/
  - Over the years, I have learned so much from Dave Jones, learning about general electronics is somethin I think is very important for any FPGA developer however something that I realized many years later was how useful it was to me to see FPGAs in commercial products like oscilloscopes, spectrum analyzers, live broadcast equipment and more.
- The Signal Path: https://www.youtube.com/@Thesignalpath
  - Dr. Shahriar Shahramian has some of the best videos on RF design and experimentation. For me, his videos have really opened my eyes to the challenges my colleagues face when designing their portion of the system involving an FPGA from the perspectives of RF or electrical design.
- The Cherno: https://youtube.com/@TheCherno
  - My favourite channel for Video Game engine programming. Yan Chernikov showcases a lot of high performance C++ and graphical development. While focused on C++ and video game engines, the stuff you learn here is applicable to any software language and the focus on performance has a lot of spillover into hardware design when you think about memory locality.
- Chips and Cheese: https://chipsandcheese.com/

  - Extremely deep dives into various ASIC designs out there. Often detailed microarchitectural measurements and discussions that tell you what is going on in the ASIC industries most interesting designs. Much of what you learn here at a high level apply to FPGA designs as well.
- Greg Davill: https://gregdavill.com/
  - Fellow PCB designer and FPGA developer, always an absolute treat when Greg posts something on his website or on Twitter, definitely worth a follow.
- Alex Forencich: http://www.alexforencich.com/
  - I feel like there are few FPGA developers that do not use or at least have come across Alex Forencich's open source repos. Whoever I talk to, the reusable components in Verilog-AXI, Verilog-AXIS and more are probably useful to you even if you use commercial or internally developed IP. No matter what FPGA team you join, there is a pretty decent chance you will find at least something pulled from Alex's open source efforts.
- Adam Taylor: https://www.hackster.io/adam-taylor
  - If you need to do something on an FPGA, one of the first things you should do is to check if Adam Taylor has done it before and written about it. I feel like Adam Taylor has become the FPGA FAE of the world. I it involves an FPGA then you can be sure Adam has something to say about it.
- Project Nayuki: https://www.nayuki.io/
  - I have learned so much from Nayuki over the years, tons of software and math snippets that have been helpful to me more then a few times over the last 10 years.
- CPPCon: https://www.youtube.com/@CppCon/
  - My language of choice these days when execution performance matters is C++ so CPPCon's archive is one I browse quite often.
- Chaos Communication Congress: https://www.youtube.com/@mediacccde/
  - Massive conference on many topics related to cybersecurity, computer architecture, computer networking, encryption. Topics are very far reaching. The top talks every year are an absolute treat, I have learned so much from the presenters here.
- Game Developer Conference: https://www.youtube.com/@Gdconf/
  - The amount of stuff I have learned from GDC that can be applied to FPGA development is too numerou to mention. The scalability of game engines and the challenge of managing a very diverse team is something not just unique to game development but common across engineering.

DSP FPGA HDL MICROCONTROLLERS SIGNALS

4 THOUGHTS ON "GROWING AS AN FPGA DEVELOPER"

K

APRIL 3, 2023 AT 12:20 PM

Nice info. very informative

Prashant

DECEMBER 26, 2023 AT 1:56 PM

Very thoughtful and informative. Thanks for putting it online.

Hakeem S.

MARCH 25, 2024 AT 6:16 PM

Thank you for the time taken to share this. Valuable beyond belief. Greatly appreciated. I already know I will come back time and time again!

kolman

JUNE 17, 2024 AT 9:58 PM

thank you so much! definitely learning a lot from the article