Accelerating Time-to-Market with Continuous Integration for FPGA Design

The unique adaptability of FPGAs allows innovators to be at the forefront of technology. They provide crucial first-mover advantage over Application Specific Integrated Circuits (ASICs) by requiring no manufacturing lead time. Recent advances in scale and System on Chip (SoC) designs are also now allowing FPGAs to compete against CPU and GPU technology. As FPGA architecture has increased in resources and complexity, so too has the required hardware development effort.

FPGA designs coded at the Register Transfer Level (RTL) have increased from thousands of lines to hundreds of thousands of lines, requiring more development time to get a product to market. In response, high-level tools that make designing for FPGAs accessible to many developers, such as software developers, have been released. Adopting high-level tools is not enough to overcome this development challenge alone – a framework for automating the development process is also needed to ensure high quality testing with fast releases.

By adopting Continuous Integration (CI)/ Continuous Deployment (CD), FPGA developers can accelerate their time to market, even as the complexity of their systems grow. This whitepaper will explore how adopting CI/CD leads to higher quality, more predictable releases. It will also show how CI/CD reduces communication overhead and keeps teams integrated and efficient. CI/CD also provides benefits to the individual developer by reducing the need for them to maintain their own developer environments and instead, focus on exploring the design space. As a case study, a computer-vision based colour detection system is provided, which shows how the project may be fully automated into a single workflow. We believe that through the automation that CI/CD provides, the FPGA design process can be accelerated and the final design to be delivered to market rapidly.

The need for design process productivity

In the early days of FPGAs, they were often limited to glue logic or implementing different IO standards for embedded markets. FPGAs then began increasing in resources, complexity and speed, reducing power consumption, thereby opening them up to new applications. They began becoming a viable alternative to ASICs, as they were able to provide the required performance, whilst reducing the number of verification steps and having no manufacturing lead times. This makes FPGAs appealing in time-to-market critical applications, where any time lost leads to significant loss of sales.

More recently FPGAs are being used in preference over CPU and GPU technology for high speed applications as modern FPGAs can provide tenfold performance over CPUs and offer low performance per Watt compared to GPUs. However, many developers in these markets are still hesitant to use FPGAs, due to the large development times compared to CPUs and GPUs.

For FPGA applications to be competitive, the time taken for development must become comparable to CPUs and GPUs. Yet, as FPGAs become more advanced, development times are at risk of rising. When FPGA designs were small in scope, design and verification hardware engineers could efficiently design novel solutions at the Register Transfer Level (RTL), using languages such as Verilog or VHDL. Bit and cycle accurate hardware simulation speeds have always been slow to capture, but designers were able to effectively debug by tracing a few critical signals. As FPGA designs increase from thousands of lines of RTL code to hundreds of thousands, far more development time must be spent on implementation details. Systems may now consist of tens of modules, making it unrealistic for small teams of hardware engineers to rapidly design and debug every individual module. These modules must all be integrated into the system once they are complete. This integration requires large amounts of time devoted to debugging interfacing and communication.

The effect of this increasing complexity is that designers spend less time exploring the design space and novel solutions. Instead, they are forced to implement the fine details of a single design. If this trend continues, developers will need to either lengthen development time and sacrifice time-to-market advantage, or settle for less optimal designs, potentially harming the performance benefits of using FPGAs.

This has created a push for FPGA design process to adopt more modern, high-level productive methods that allow designers more time to work on value-added solutions. We have seen innovation in FPGA toolsets that are focused on making FPGA designs far more accessible to all types of developers, not only those skilled in RTL. For instance, new High Level Synthesis (HLS) tools allow software developers to program FPGA designs in C/C++. These new tools allow for a far more diverse and modern development force that can effectively explore the design space of complex systems.

New toolsets alone are not enough to overcome these development challenges. Diverse teams need to be able to coordinate with each other to build a system. Moreover, a system must be tested from the RTL level to the final software application, to guarantee correct functionality. To solve the problems faced by adopting high-level tools, FPGA designs must also be developed using modern software development practises. We believe that Continuous Integration and Continuous Development practices present the solution to reducing development time, whilst ensuring novel, innovative designs are created on schedule.

Decreasing time-to-market through CI/CD practises

Higher quality, more predictable releases

CI is the practise of teams making continuous, incremental changes to a FPGA design's code base, thereby providing higher quality, more predictable releases. Code is automatically and frequently built and tested to ensure high quality, bug-free systems. To achieve this, the code base is worked on simultaneously and regularly updated by all members of the team, sometimes as often as multiple times a day. Version controls systems, such as Git are mandatory for managing these changes. The entire system then undergoes a series of builds and tests known as a workflow. Since tests are a crucial part of CI, developers must keep tests up-to-date and constantly add tests as an application progresses. This provides higher quality releases than when testing is left till the end of development.

CD is the natural extension of CI that ensures that at the end of the workflow, the product is ready to be deployed to customers. CD prevents backloading system integration to the end of development, where system-level bugs can become time consuming to fix. Instead, CD ensures that risk is spread more evenly as system progress is far easier to track.

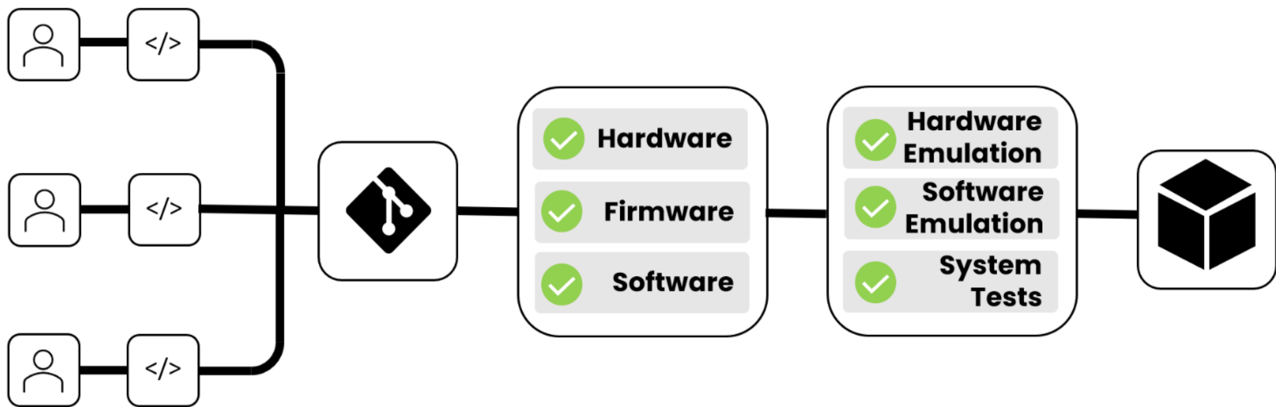

Figure 1

An overview of how a CI system functions. Developers contribute to a version control system such as GitHub. When a change to the code is pushed, the system is built. The system is tested through emulation and hardware tests, alerting the user to test and performance results. Afterwards, files are made ready for download to deploy in production.

Figure 1 provides an overview of a simple CI system for FPGA design. In this overview, developers contribute code to a single codebase held in Git. An automated system will then pull code from Git and build the system, beginning with the hardware and ending at the software application. Once the system has been built, it is tested through hardware emulation that runs RTL at a cycle-accurate level. We also test through software emulation that provides fast results for our software application. Finally, we test the entire system through placing the design on hardware itself. If all our tests succeed, we then package our build, ready for deployment.

Automating these workflows, such as those in Figure 1, requires a CI/CD ecosystem made of one or more tools. Figure 2 demonstrates the typical infrastructure needed for CI/CD, split into four distinct modules:

- **Repository:** The code repository is a version control system that all developers submit their code to. The repository will track any changes made to the source code and submit any significant events to the orchestration software. Typical events include any committed changes to the code. Most used is the open-source Git, which has several hosts, such as Github, Gitlab or Bitbucket.
- **Orchestration software:** The orchestration software organises the CI/CD workflows. They setup when, where and in what order builds and tests must occur in through user-created configuration files. Orchestration software will also manage sending tasks to the execution server to run the builds and tests. It will also typically manage where the builds are stored after they have been run. Popular examples of Orchestration Software include CircleCI, TeamCity and Jenkins.
- **Execution server:** The execution server is responsible for running the builds and tests. These run the code in a clean environment that contains only the tools needed to run the software. They receive instruction from the orchestration software and constantly update it with the status of the build. Once the build is finished, the files are transferred over to the orchestration software for long-term storage. Most orchestration software will come pre-packaged with cloud-based execution servers, such as CircleCI, but many will also allow users to set up their on-premise servers as execution servers.
- **Deploy:** The resulting output of each build is known as an artifact, and if that artifact passes tests, it is ready to be deployed. The method of deployment varies heavily, depending on a range of factors, such as if we are deploying to servers or to embedded devices. Server deployments may take advantage of popular container orchestration servers such as Kubernetes, whereas embedded services can utilise over-the-air updates.

There are numerous tools and methods for creating a CI/CD infrastructure from completely on-premise, open-source solutions such as Jenkins with Docker to more cloud-based solutions such as Github Actions or Amazon

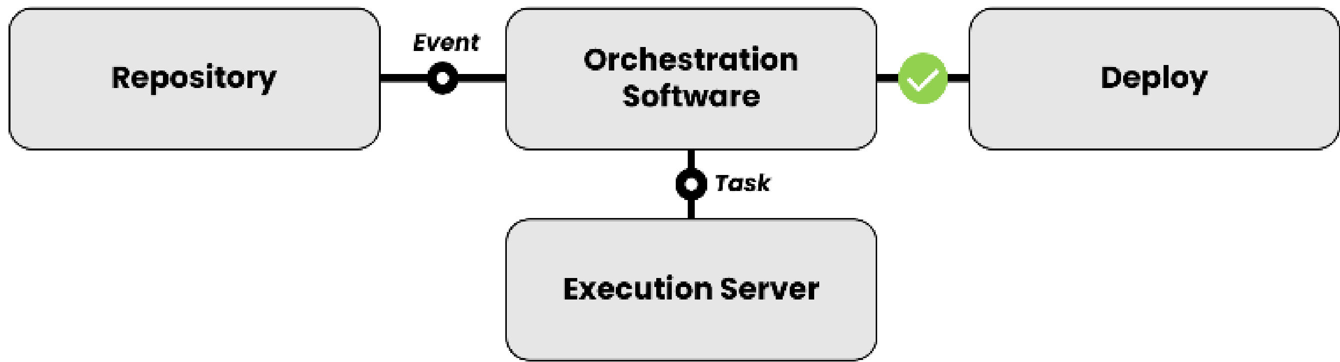

Figure 2

An overview of CI/CD infrastructure. Developer code is held in a repository and when a significant event occurs, such as new code being committed, an event signal is sent to Orchestration Software. Orchestration Software is responsible for the coordination of the workflows and will typically schedule tasks such as build and tests to run on an execution server. Once tests have been passed, the design is ready for deployment. The deployment mechanism will vary depending on environment. For servers, this will typically involve uploading bitstreams, whilst embedded environments will be more focused on flashing images.

Web Services. The right CI/CD for an organisation will depend on both practical and cultural considerations. For instance, more security-minded teams may need a completely on-premise solution that is functional with no connection to the internet, whereas more cost-conscious teams may chose a hybrid solution, where they have a baseline amount of on-premise servers, but use the cloud during the small periods of time when they require more computational speeds.

By choosing the right infrastructure, a team can automate their workflows for any application they work on. Workflows allow developers to regularly commit their code and have it built and tested to debug errors fast. Through CI/CD, developers take a more holistic view of the entire system. Instead of waiting till the last stage to put together an entire system, the system should always be ready for deployment. This helps spread risk evenly through a project and keeps tests regular and up to date.

Teams are better co-ordinated and more effective

CI means fostering a team culture where everyone contributes to single effort. Without CI, developers will naturally work in a silo where they focus on their own modules and only coordinate with one another when integrating their modules. This creates a communication overhead as project management must manually track each developer's progress and estimate how that fits into the overall system.

As FPGA teams diversify, this coordination becomes more difficult as FPGAs are no longer limited to hardware engineers. New FPGA tools allow for a far more diverse and modern development force that can effectively explore the design space of complex systems. Examples of these new developers include:

- **Hardware design and verification engineers:** In modern FPGA development, hardware engineers are responsible for creating a hardware shell to allow other developers to build the system within. These shells tend to focus on the design interfaces, IO and connectivity. By being able to provide this shell, hardware engineers can offload much of the work they previously had to do onto the system itself.
- **Software developers:** Much of the hardware developers' work can now be handled by software developers using High Level Synthesis (HLS) tools. Software developers can develop hardware-accelerated code at much faster rates, using the languages they are familiar with such as Python, C and OpenCL. This allows for better design space exploration, reaching more novel solutions within a limited development period.
- **Firmware developers:** Modern FPGAs will often be combined with an on-chip CPU to form a System on Chip (SoC). These SoCs can run Operating Systems (OS), such as RTOS and Linux. Using OS is a requirement in most modern systems. Firmware developers handle the installation of these OS as well as any needed programs or drivers.
- **Data Scientists:** Most recently, FPGA designs can accelerate AI models developed using popular

Open-Source tools, such as TensorFlow and PyTorch. Data Scientists can convert their pre-existing models onto FPGAs and enjoy high-performance inference of their models.

Team Role	Level Name	Examples
Data Scientist	Software API	Tensorflow, PyTorch
Software Developer	Software Kernels	HLS tools, OpenCL, C/C++
Firmware Engineers	Firmware	Yocto, Linux, Petalinux
Hardware Engineers	Hardware	Vivado

Figure 3

The full stack of an FPGA design from hardware to software API. Each level also includes the team member responsible and examples of tools that are used at these levels.

Figure 3 shows the entire software stack for a solution. Keeping organised at all levels is a difficult task. For instance, data scientists are reliant on AI modules that provide an API that can run their models. This hardware needs to be integrated as soon as possible. This requires a co-ordinated effort for the hardware team to move the latest build through approval. Firmware and software developers can automate their approval process before the design is usable to the data scientists. CI makes managing these relationships far easier through automation.

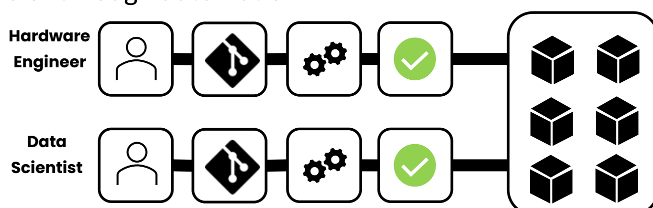

Figure 4

An example of how developers contribute to an artifact store to form a single point of truth for all built and tested binaries. A hardware engineer will commit changes to repository that is then built and tested and stored. Data scientists may then use the latest build as a base for their own development, which they then subsequently will commit, test and store.

Figure 4 demonstrates how CI can synchronise developers together. In this example, the hardware engineer commits their code to a Git repository that is then built and tested. If the tests sucessfully pass, the resulting platform becomes an artifact which is then stored in the aritifact store. A data scientist in the team may then use this platform as the basis of their applications. The advantage of this system is that it removes out-of-sync communication between the data scientist or any member of the team. The builds and tests can be set up to ensure that the platform automatically applies the firmware and software kernels, and ensures correct functionality. In essence, the artifact store forms a single source of truth for all succesful builds, packages or binaries. Teams are able to quickly coordinate, monitor tests and download any needed files from a single source. Other advantages of the artifact store include:

- **Scalability:** FPGA designs produce large files that can fill developers' local environments, forcing the deletion of data. Artifact stores present a long-term storage solution that can scale up as more data is needed to be stored. Team-wide data policy is enforcable, such as only deleting data if it is more than a month old.
- **Security:** By a having a single source of data that is immediately relevant to a solution, it is simple to back-up and restore data.

The artifact store presents a way of keeping the team synchronised, providing files as they are needed. This is required in modern FPGA designs as teams are becoming more diverse, embracing a larger stack that starts from hardware developers and continues on to data scientists making use of accelerated software kernels to develop applications. Through CI, FPGA design teams can keep efficient, whilst also fostering a co-operative, shared environment.

Individual developers are far more efficient

One of the greatest bottlenecks for FPGA developers is managing their builds and tests, especially if they wish to automate tasks or run multiple tests in parallel. CI/CD is an effective method to help individual developers manage this workload. FPGA tools processing- and memory-intensive, meaning controlling their computing resources effectively is challenging. Minor changes to the code base are often accompanied by long wait times for builds and tests.

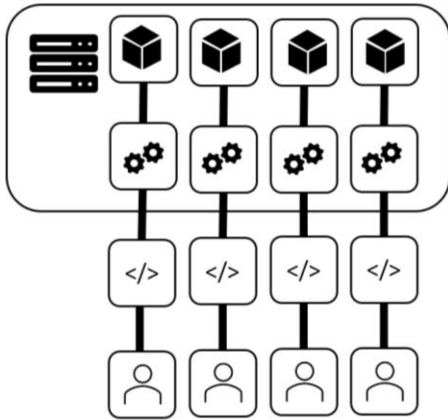

Figure 5

A typical-build server infrastructure that is managed by developers themselves. Developers commit code and jobs that run in their own local environment and is isolated from other environments.

Providing developers dedicated-build servers can allow them to run as shown in Figure 5, which may either be on-premises or in the cloud. In this model, individual developers will locally work on code that is then submitted to a server. The server may then provide a separate environment for the developer through virtual machines or containers. This separate environment will contain the tools and computing resources needed for the developer to run their submitted code.

Developers are given full responsibility of managing their personal environment. They must effectively set up, monitor, and maintain these environments through scripts. The problem with this system is that developers must spend time managing their resources. Automating their personal workflow requires writing scripts. If a developer wishes to run multiple builds and tests in parallel, they must manage their provided resources to do so. All of this distracts from innovation and performing design space exploration.

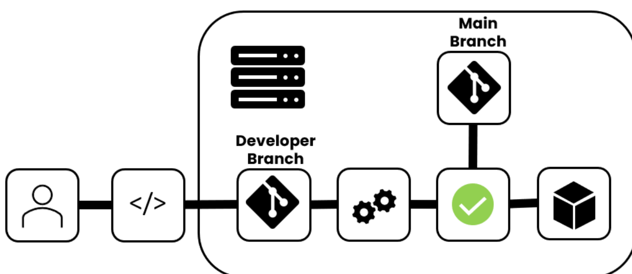

Figure 6

CI/CD workflow from the perspective of an individual developer.

Figure 6 shows CI/CD from the perspective of a developer. Before code is added to the main code base, it is first built and tested using a local copy of the code base that the developer has changed. This local copy is known as the developer branch of the code. If the code is successfully built and tested, it is merged into the main branch of code, which is the shared code base.

The significant change of CI/CD over dedicated build servers is that the infrastructure for builds and tests has shifted from an individual developer managing their own environment on to a team-wise responsibility. Automation and the criteria of acceptance into the main branch is handled by the group. At a minimum, these tests are used to ensure that the system remains stable and working as before, which is referred to as regression tests. Teams may also decide to use techniques such as test-driven development, where tests are written before code is developed and code is only accepted upon passing of the test.

This lowers the workload of an individual developer because ensuring that their changes work properly is now a group responsibility. The infrastructure is maintained as a group rather than individuals having to support their own environment, as was the case with the build server infrastructure. Using CI/CD, developers no longer need to be concerned about managing their local computers' resources, and may instead focus on the development of the system.

A practical example of CI/CD system

Now that we have established why adopting CI/CD can benefit FPGA designs, let us look at a practical example of how we can develop a colour detection system using CI/CD. Colour detection is a common application within computer vision and robotics. In this example, we will assume a team of engineers consists of a hardware design engineer, a hardware verification engineer, a firmware engineer and a software developer specialised in computer vision.

Our colour detection system

The specification for our colour detection system is as follows:

- **Development board:** Ultra96-V2 Arm-based, Xilinx Zynq Ultrascale+ MPSoC development board.
- **OS:** Linux
- **Input:** HDMI
- **Output:** HDMI

- **Acceleration:** The image-processing pipeline shown in Figure 7 is needed to accelerate the colour detection system. Once a frame from the HDMI input has been received, it is converted into the correct colour format. A thresholding filter is then applied to the image to isolate the colour that is needed. Finally, the image is dilated to remove noise.

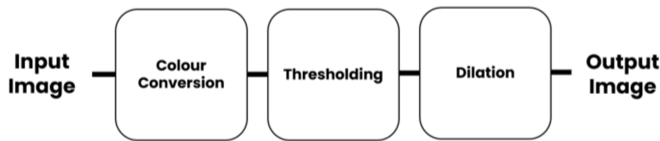

Figure 7

Colour detection image-processing pipeline.

The team must then break down the system specification into the high-level features needed at each level. Figure 8 provides the full system stack and what each level must include for proper functionality:

- **Test architecture:** For the first level, the verification and hardware design engineer must design a testing architecture that ensures proper functionality of the hardware shell. The test architecture may include an IO emulator to mimic HDMI inputs and outputs, as well pattern generation to feed in video data. To check the correct output, the test architecture will also need to provide a signal check for the outputs to ensure no signal is malformed. This test architecture can be developed using System Verilog with the Vivado Design Suite.
- **Hardware shell:** The hardware design engineer must then provide a hardware shell for the other engineers to develop a system within. This involves deciding on the necessary IO and interfaces, such as HDMI, as well as setting up the CPU and reconfiguration environment on the SoC. A first version of this may involve using the pre-provided images for the board. This hardware shell is developed in Vivado using Verilog or VHDL.
- **Firmware:** An embedded Linux OS system must be placed on the CPU. To do so, firmware engineers may use tools such as Petalinux that create a custom OS. These can be pre-loaded with necessary libraries such as GStreamer or OpenCV.
- **Software API:** To implement the colour detection itself, we can build an image-processing stream through Vitis Unified Development environment. It provides HLS tools for software developers to develop hardware accelerated systems. This image-processing stream can then be made into a kernel through OpenCL that can be exposed to the application through an API. The colour detection requires filter modules, a colour threshold module and a dilation module. Software developers can make use of pre-provided hardware acceleration libraries for these modules, such as the Vitis Vision library,

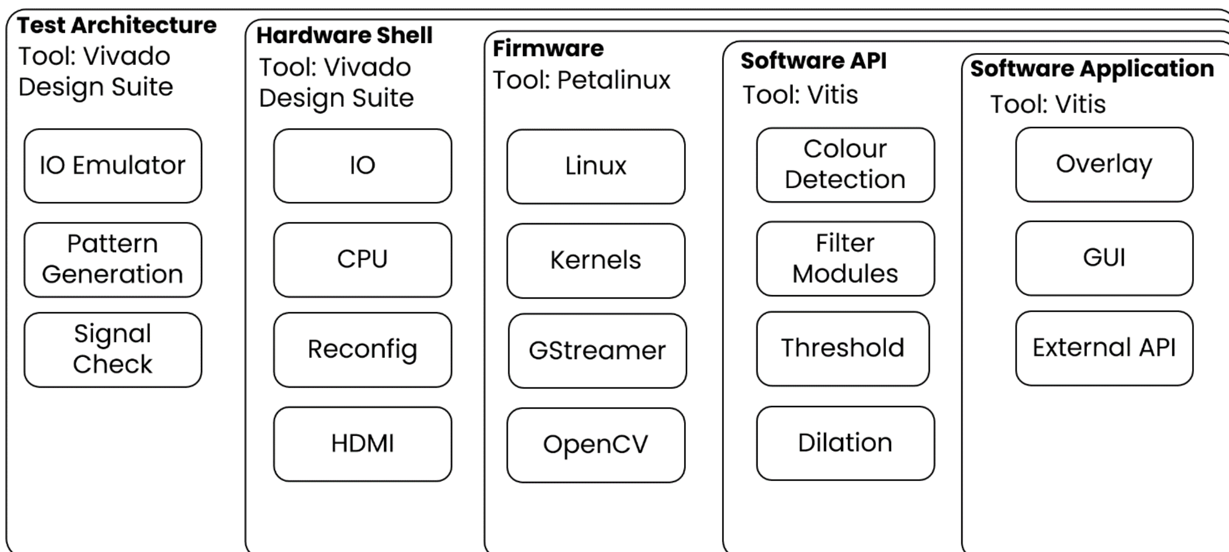

Figure 8

The full system stack for colour detection. Each level indicates the minimum requirements to perform the colour detection, as well as the tools used.

which can be linked together to form an image-processing stream.

- **Software application:** The colour detection application can be developed in a similar manner to other software applications, even if a developer has limited experience with HLS tools or OpenCL. The software API will expose the hardware accelerated colour detection system through an overlay. This allows the software application development to instead focus on user experience, through a Graphics User Interface (GUI) or an external API that can allow users to call commands from terminal.

Managing development through CI

For our colour detection system development to stay on target, we will show how CI/CD can be used to manage it. The CI/CD infrastructure must first be established. We propose using GitHub as our code repository. For our orchestration software, we will use open-source Jenkins, which will allow us to integrate GitHub as well as set up servers remotely or use on-premises servers for execution of our builds and tests. For deployment, we will store the images on our on-premises servers, to be manually installed on our embedded boards through SD cards.

How CI aids individual developers

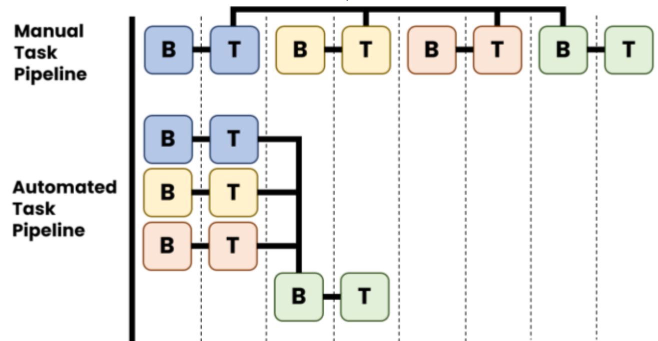

Figure 9

*Two CI workflows that build the image-processing stream. Top: Manually running each task in the pipeline. Bot: Automating each task in the pipeline through CI. Each block represents a task: **B** blocks represent build tasks, whilst **T** blocks represent test tasks. Connected blocks have a dependency on each other. Each colour represents closely related tasks. Blue: colour conversion tasks. Yellow: threshold tasks. Orange: dilation tasks. Green: kernel integration tasks.*

Figure 9 shows the workflow that the software developer will use to develop the image processing stream. The top workflow shows the task pipeline if the software developer runs each task manually. The developer must perform setup, monitoring and packaging of artifacts manually. This makes running different tasks at the same time difficult. The bottom workflow represents how the tasks are run in an automated pipeline. Crucially, setup, monitoring and packaging are automated by the CI/CD system, making it simpler for the programmer to run tasks in parallel. Tasks with dependencies on each other, such as a build and test of a module, cannot be run at the same time. Even with these dependencies, automation can reduce the build and test time that the developer will need to undergo, making them more efficient and able to explore the design space more effectively.

How CI helps manage the team

The continuous nature of CI means teams become proactive in debugging and meeting system requirements. Small iterative changes to the codebase, that are constantly checked, have two major benefits. The first is that bugs become simpler to track as the exact version that introduced the bug can be found. Secondly, performance and other metrics can be monitored on a fine-grain level. Tests can identify the impact of a change on the codebase, allowing developers to react to poorly performing iterations or to continue pursuing positive changes.

In Figure 10, we show the workflow setup and at the end, the entire system will be ready for deployment. Each stage creates a discrete artifact that must be used on the next level of our stack, creating a chain of dependencies on which we can base our workflow. Each stage consists of the same iterative processes:

- The developer creates a feature for the next iteration, and their code is committed to the Git repository.
- The orchestration software will then build and use pre-determined tests to see if the build is successful. Tests may include proper functionality or hitting performance criteria.
- On successfully passing tests, the new artifact is then made available in the artifact store. Developers may then use this artifact as part of their testing process.
- The tests are also used to inform the developer on metrics such as performance, which can guide the developer on their next iteration.

In this workflow, we will create five artifacts: a testing architecture; a hardware shell; a Linux OS distribution;

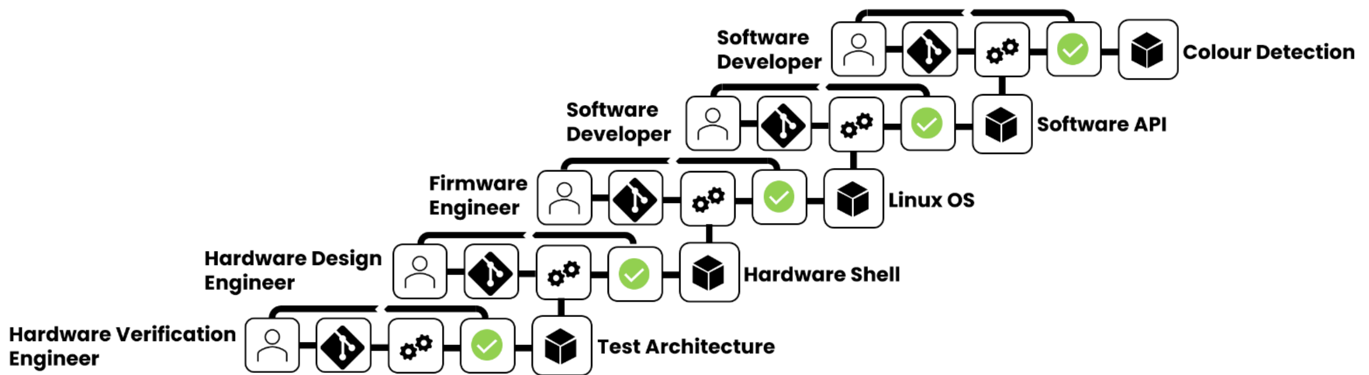

Figure 10

Our workflow for the colour detection system. Each level represents the process needed to produce an artifact for the next level. Starting from the bottom: the hardware verification engineer will iterate on the code base and commit that code to the Git repository. The CI/CD system will then build and test this system, and if it passes, will produce the Test Architecture artifact. The test will be used to inform the developer and provide feedback for their next iteration. This artifact is needed by the hardware design engineer for their tests. This forms a chain of dependencies until the entire system is complete.

and the hardware accelerated colour detection application. Each artifact has a separate testing procedure to ensure high-quality code:

- **Test architecture:** The testing architecture will consist of a series of modules that can be imported to place a design under test. Each of these modules must also be tested to ensure that they are error free. Common tests will include correct interfacing and accurate capture of a design-under-test's output. Tests at this stage will be cycle and signal accurate hardware tests, using assertion-based and coverage tests.
- **Hardware shell:** The hardware shell will be reliant on the test architecture modules to build a testing system. The hardware shell is placed under test by connecting the pattern generation module to the input, and the signal check on the output. Using this, the system can then be tested, using standard procedure such as constrained random tests or assertion-based tests. We may also choose to track both code and functional coverage, and have a certain threshold that must be passed. If these tests pass, the hardware platform is generated.
- **Firmware:** The firmware requires a description of the hardware to create a Linux OS image to use. The OS can be generated through Petalinux, which also allows us to set up kernels for the IO and preinstall any required libraries. Testing may involve correct installation and booting.
- **Software API:** Vitis requires the OS image to create acceleration kernels. Individual modules may be tested for both functionality and performance data. These modules can then be integrated into a larger kernel test. At this point, we can derive the acceleration that is possible for the colour detection system and gain important performance measurements such as throughput and latency.
- **Software Application:** Finally, the software application may be tested using conventional software techniques, such as unit tests and system tests. The artifacts at this point can then be combined to form a complete system, ready for distribution.

Through CI/CD we can establish a workflow that efficiently builds and tests each level of the colour detection stack. We also highlight how developers may integrate standard testing practises at each one of their levels. Using CI/CD encourages these standard tests to be integrated into the system as fast as possible, so that the developers using the latest released artifacts have confidence in their stability. The work of developers in different fields can be integrated into a single workflow. We encourage teams to see how they can adapt their own processes into an automated process to see how CI/CD can help improve their time-to-market.

Summary

Through our colour detection system, we provide an example of how CI/CD can benefit FPGA development. CI/CD helps individual developers by providing an infrastructure to automate their builds and tests on, instead of them individually managing their environment. The advantages also extend to the entire team by making it possible to manage large workflows across multiple levels in the software stack, reducing the communication overhead and providing a single source of truth for all artifacts. The use of CI/CD provides regular automated tests, which can identify system-level bugs early within the design process and evenly distribute risk across the project. CI/CD ensures that FPGA design releases are higher quality and more predictable. Through these benefits, CI/CD can accelerate products to market, even as sophistication of the system increases.

Beetlebox Limited is a company registered in England & Wales with Company Number 11215854 and VAT no. GB328268288