FPGA, Git, DevOps, CI/CD

Modern yazılım geliştirme yöntemlerini neden FPGA projelerine uyarlamayalım?

Alper Yazar <ayazar@alperyazar.com>

Bu yazımda,

- FPGA projelerini Git gibi bir versiyon kontrolü ile neden takip etmemiz gerektiğinden
- Bu süreçte dikkat etmemiz gereken temel noktalardan
- Yazılım dünyasındaki DevOps, CI/CD gibi kavramlardan nasıl faydalanabileceğimizden

bahsedeceğim.

Bu konulardaki fikirlerimi serbest, biraz da karışık, formatta anlattığım bir yazı olacak.

Hadi başlayalım!

FPGA Projeleri ve Git

2018 yılının başlarında ACCLOUD, Accelerated Cloud, isimli bir AR-GE projesini başlatmıştık. Yaklaşık 3 yıl süren bu projenin birçok aşamasında baştan sona görev aldım. Bu projede yoğun bir FPGA kullanımı vardı ve bugüne kadar çalıştığım takım yapısından farklı olarak fiziksel olarak bir arada olmayan, aynı FPGA tasarımına farklı lokasyonlardan katkı sunan kişilerden oluşan bir takım oluştu. Bu durum, bana o güne kadar biraz daha deneysel takıldığım **Acaba FPGA projelerini bir yazılım projesi gibi düzgün bir şekilde Github/Gitlab gibi platformlarda nasıl tutabiliriz?** başlıklı araştırmalarımı ve denemelerimi hayata geçirmek için bir fırsat oluşturdu. FPGA projelerimizi, Gitlab üzerinde tutmaya başladık. Bu çalışmada Xilinx, şimdiki AMD, firmasının ürünleri ve araçları kullanıldı. Fakat önemli bir engeller vardı: **Xilinx gibi FPGA firmalarının araçlarının çoğu, Vivado gibi, Git ile konfigürasyon takibi yapmaya ve CI/CD süreçleri ile otomatik derleme yapmaya çok da uygun değildi. Hangi dosyalar versiyon kontrolünde olmalıydı? CI/CD süreçlerinde otomatik derleme en kolay nasıl yapılabilirdi? Hem Windows, hem Linux üzerinde çalışan kişiler için uyumlu bir sistem nasıl olabilirdi?**

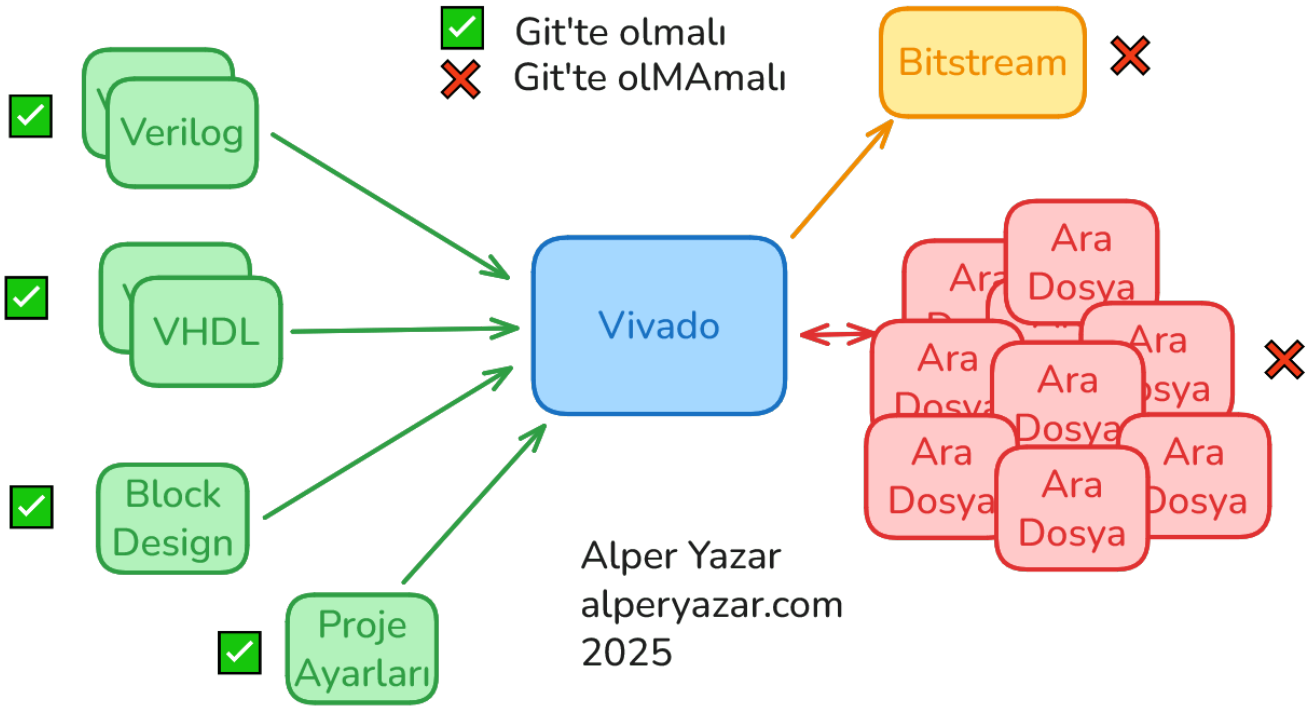
Bu yazıda bahsettiklerim o yıllardan beri biriken çalışmalara dayanmaktadır.

Kağıt üstünde baktığınız zaman, Vivado'nun 2010'lu yılların ilk versiyonlarından itibaren Git ile uyumlu olduğunu söylediğini görebilirsiniz. **Fakat bunun ayağı ne kadar yere basıyor?** Bugün bir yazılım projesini, bir framework ya da araç ile oluşturduğunuz zaman size birçoğu bir `.gitignore` dosyası sunuyor. Örneğin bilgisayarınızdaki Word dosyalarını, `.docx`, alıp bodoslama `git init`, `git add .` ve `git commit` ile Git versiyon kontrolü altına aldığınızda Word, Git uyumlu bir program mı oluyor?

Vivado gibi EDA araçları, sentez/derleme sırasında birçok ara dosya üretmekte. Günün sonunda sizin amacınız belki 5-6 adet VHDL/Verilog/Block Design dosyasından bir bitstream'e gitmek. Ama

araçlar bu hedef bitstream dosyasını üretirken onlarca, belki yüzlerce ara dosyalar üretebiliyor.

Bir projenin sağlıklı bir şekilde versiyon kontrolünün yapılabilmesi için Git gibi bir sistemde hangi dosyaların gerçekten bir kaynak dosya olduğu hangilerinin ise göz ardı edilebileceğini bilmek gerekiyor. Örneğin sentez ya da derleme sırasında üretilen ara dosyaların prensip olarak versiyon kontrol altında olmaması gerekiyor. Olmasının getireceği en önemli problemlerden biri `git diff` gibi bir komut ile iki *commit* arası farka baktığınız zaman aslında anlamlı olmayan dosyaların size bir **diff noise** yaratacak olmasıdır. Bunun dışında başka durumlar da var elbette ama yazıyı çok uzatmak istemiyorum.



İPUCU

Vivado'dan örnek verecek olursak ara çıktı dosyaları ve bitstream dosyası da, şaşırtıcı gelebilir belki, versiyon kontrol altında olmamalı. Kaynak kodlarınız ile beraber projenizin ayarları ise (`.xise` ya da `.xpr` olmak zorunda değil, başka alternatif yollar da olabilir) versiyon kontrolü altında olmalı.

ÖNEMLI

Bir projenin çalışma dizinin en azından düzgün bir `.gitignore` ile konfigüre edilmeden olduğu gibi Github/Gitlab gibi platformlara *push edilmesi* o projenin Git ile sağlıklı bir şekilde takip edildiği anlamına gelmemektedir.

Clean Build Alabilmenin Verdiği Huzur

Bir proje için gerekli olan minimum set dosyayı kullanarak hedef çıktıya yani FPGA projeleri için bitstream dosyasına sorunsuzca gidebiliyorsak burada aslında *clean build* almış oluyoruz. Tahmin edeceğiniz gibi bu terim FPGA projelerine özgü bir terim değil. **Fakat bu kavram bence FPGA projeleri için oldukça önemli.**

FPGA içeren projeler genelde uzun soluklu projeler oluyor. Örneğin savunma sanayinde FPGA sık

kullanılan bir platform. Burada bir ürünün tasarımı bitmiş, teslim edilmiş olsa da uzun yıllar ürüne yeni özellik ekleme, hata düzeltme gibi işler için destek vermek gerekiyor. Durum böyle olunca, FPGA projesinin doğru saklanması, versiyon kontrolünün yapılması, gerektiğinde geriye dönük analiz yapılabilmesi ve yıllar sonra bile derlenebileceğinden mümkün mertebe emin olunması gerekiyor.

Bunun iki ayağı var: **Projenin düzgün saklanması, ilerletilmesi** ve **geliştirme ortamının saklanması** Geliştirme ortamı konusuna başka bir yazıda değinebiliriz, bu yazının konusu ağırlıklı projenin düzgün saklanması ile ilgili ama yine de merak ettiyseniz belki EBox, Embedded Box projeme bakabilirsiniz.

Kendimize şu soruyu sorup cevabını vermemiz gerekiyor: **Ben bu projeyi 10 yıl sonra da açtığımda ve bir değişiklik yapmam gerektiğinde bunu sorunsuz bir şekilde yapabilecek miyim?**

İşte bu soruya "evet" diyebilmek için projemizin ara çıktı dosyalarını silsek de temel kaynak kodlardan tekrar derlenebildiğinden yani *clean build* alabildiğimizden emin olmamız gerekiyor.

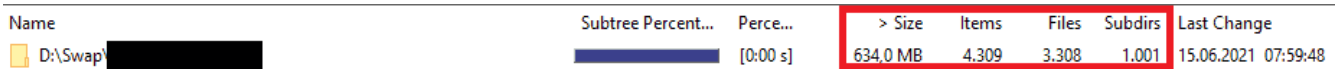
Bundan emin olmak için ise projemizin sürekli temel kaynak kodlarından sorunsuzca derleyebiliyor olmamız gerekiyor, elbette elle yapmak zorunda değiliz otomasyon kurabiliriz. Git (veya SVN fark etmez) gibi bir versiyon kontrol sisteminde de ara dosyaların olmaması gerekiyor.

Konuyu örnekler vererek derinleştirelim.

Hikaye - 1

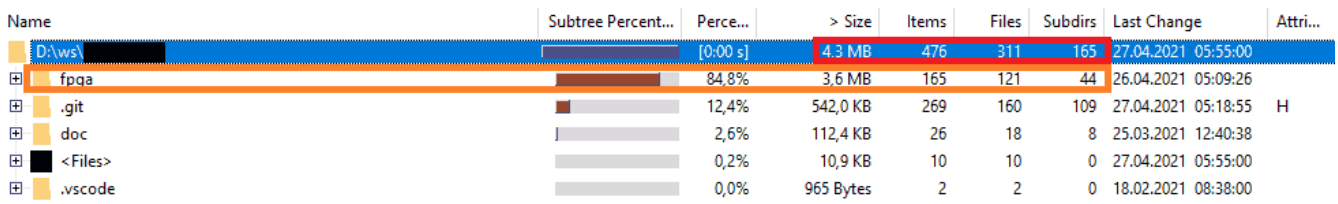
Xilinx (AMD) Vivado'dan örnek verecek olursak, bir IP core OOC, out-of-context, gibi yöntemle sentezlendiğinde Design Check Point, `.dcp`, uzantılı dosyalar oluşuyor. Bunları Vivado'nun kullanıdığı ara çıktılar ve bir nevi cache dosyaları gibi düşünebiliriz. Bu ve buna benzer dosyalar oldukça fazla sayıda olabiliyor ve yer kaplıyorlar.

Aşağıda iki adet ekran görüntüsü vereceğim. **Bu görüntüler 20-30 adet IP Core + HDL kodları içeren bir projeden alındı.**



Name	Subtree Percent...	Perce...	> Size	Items	Files	Subdirs	Last Change
D:\Swap\[redacted]		[0:00 s]	634,0 MB	4.309	3.308	1.001	15.06.2021 07:59:48

Bu şekilde saklandığı zaman projemiz 600 MB üzerinde yer kaplıyor, içerisinde 1000 adet klasör ve 3308 adet dosya barındırıyor. Peki bu projeyi saklamak için bu kadar dosyaya ihtiyaç var mı? Sizce ne kadar dosyayı atabiliriz? Gelin aşağıdaki ekran görüntüsüne bakalım.



Name	Subtree Percent...	Perce...	> Size	Items	Files	Subdirs	Last Change	Attri...
D:\ws\[redacted]		[0:00 s]	4,3 MB	476	311	165	27.04.2021 05:55:00	
fpga		84,8%	3,6 MB	165	121	44	26.04.2021 05:09:26	
.git		12,4%	542,0 KB	269	160	109	27.04.2021 05:18:55	H
doc		2,6%	112,4 KB	26	18	8	25.03.2021 12:40:38	
<Files>		0,2%	10,9 KB	10	10	0	27.04.2021 05:55:00	
.vscode		0,0%	965 Bytes	2	2	0	18.02.2021 08:38:00	

Biz bu projeyi aslında 4 MB'ın altında bir alanda ve yaklaşık 120 adet dosya ile aslında

saklayabiliriz! Geri kalanların hepsi derleme/sentez sırasında çıkan ara dosyalar ve bunları versiyon kontrolüne koymamız uygun değil.

Elbette şunu diyebilirsiniz: *Yahu üç beş megabyte'ın hesabını mı yapacağız?* Temel motivasyonumuz bu değil. Temel motivasyonumuz versiyon kontrolünü düzgün yapmak ve yukarıda da bahsettiğim *`git diff` noise* gibi problemlerden kaçınmak.

Hikaye - 2

Şimdi daha "ibretlik" bir hikayeden bahsedeceğim, anlatacaklarımı yaşadığınızı hayal edin. Bir projenin yukarıda gösterdiğim gibi tüm ara çıktıları ile, ne var ne yoksa tüm dosyaları ile saklandığını hayal edin. Bu dosyaların içerisinde çeşitli lisanslı IP core'ların çıktıları da var. Yıllar boyunca projede IP core'ların ayarları değiştirilmediği için sentez sırasında aslında IP core'lar tekrar sentezlenmiyor, araç tarafından lisansları kontrol edilmiyor ve var olan ara dosyalar, adeta cache dosyaları, kullanılıyor. Bir gün IP core'ların birinin ayarı değiştirilmek istenince de aslında **IP core'un lisansının yıllar önce bittiği ve aslında değişiklik yapılamadığı** anlaşılıyor. Neden? Çünkü proje *clean build* alınarak derlenmediği için bu tarz problemler gözden kaçıyor. Tam da yumurta kapıya dayandığı zaman bunu fark ediyorsunuz. Böyle bir durumda kalmak istemezsiniz değil mi?

Sanıyorum ki **tekralanabilir** bir şekilde bir projenin **clean build** alarak derlenebilmesinin neden önemli olduğunu ve bunun için temelde nelere dikkat etmemiz gerektiğini biraz anlatabilmişimdir.

Gelin devam edelim.

"Headless Build", "Scriptable Build" Gibi Kavramlar

Sıfırdan, tekrarlanabilir derleme konularındaki **en önemli aracımız otomasyon.** Bu konuda *Github Actions*, *Gitlab Runner*, *Jenkins* gibi sistemler yardımımıza koşuyor. Bu sayede, Git üzerinde takip edilen kodumuzda bir değişiklik olduğu zaman ya da periyodik olarak istediğimiz sıklıkta FPGA projemizi otomatik olarak hem de *clean build* şeklinde derleyebiliyoruz. FPGA'den bağımsız olarak bu konuları **DevOps**, **CI/CD** gibi anahtar kelimelerle aratabilirsiniz.

Fakat bu sistemlerin sağlıklı çalışabiliyor olması için bizim FPGA projesi sentezleme/derleme işlemini düzgün bir şekilde komut satırından yani Linux'ta BASH, Window'ta Power Shell ya da CMD üzerinden yapabilmemiz gerekiyor. GUI üzerinden sağa sola tıklanarak yapılan işlemleri hem otomatize etmek hem de bir kullanıcının düzgün *tekralanabilir* şekilde projeyi bilgisayarında oluşturması ve derlemesi çok zor. O yüzden burada da önümüze **headless build** ya da **scriptable build** gibi kavramlar ortaya çıkıyor. *Headless* kelimesi bir monitör ya da GUI olmadan yapılan anlamında kullanılıyor.

Özetle sizin tüm derleme sürecinizi, bitstream oluşturma, soft/hard işlemci varsa onun kodunu derleyip ELF dosyası oluşturma, bitstream ile ELF'i birleştirme, MCS gibi diğer formatlı çıktıları

üretme gibi scriptlenebilir hale getirmek gerekiyor. **Yani adeta komut satırından tek bir komut yazacaksınız, mesela `make`, ve Enter'a basıp çayınızı içeceksiniz.** Tüm işlemler otomatik olacak.

E peki nasıl olacak?

Peki bunları yapmak için ne yapmamız gerekiyor?

1 İlk olarak FPGA projesinin "düzgün" bir şekilde versiyon kontrolünde olması gerekiyor. Eğer var olan sisteminiz yoksa gitmeniz gereken yol Git, kesinlikle SVN ya da buna benzer eski sistemler değil. Halihazırda bir sisteminiz varsa Git'e geçiş yapmak için artı ve eksileri değerlendirmek gerekir. Burada defalarca vurguladığım gibi neyin versiyon kontrolünde olup olmayacağını iyi belirlemek gerekiyor, yani `.gitignore` içeriği gibi düşünebilirsiniz. Bu da araç ve proje bazında araştırma yaparak ve tecrübe ederek mümkün olabilir. Burada projelerimizde 4 numaralı maddede bahsedeceğim araçları kullanmak da fayda sağlayacaktır.

2 Otomasyon en büyük yardımcımız. Github, Gitlab gibi sistemlerin "otomatik olarak bir şey yaptırtma" ve bunların çıktılarını kullanma (örneğin bitstream) altyapıları oldukça iyi, bizim de bunları kullanmamız gerekiyor. Burada da kullanacağımız anahtar kelimelerin başında `DevOps`, `CI/CD` geliyor. Gitlab kullanıyorsanız Gitlab'ın, Github Actions kullanıyorsanız Github'ın otomasyon kısmını iyi anlamak gerekiyor. Bundan bağımsız olarak *Bu DevOps gibi çözümler neyi çözmeye çalışıyor ve ben bunu FPGA işlerine nasıl uyarlayabilirim?* diye düşünmek gerekiyor.

3 Geliştirme ortamımızı koruma altına almak gerekiyor. Örneğin kullandığımız Vivado versiyonunu yıllar sonra da düzgün çalıştırabilecek miyiz? Bu, otomasyon kapsamında Github Actions, Gitlab Runner, Jenkins gibi ortamlarda Docker/Podman gibi altyapılarda Vivado gibi araçları çalıştırmak için de çok anlamlı. Aynı zamanda ileriye dönük ortamı korumak için de önemli. Buna bu yazıda pek değinmedik ama container teknolojilerinin iyi bir çözüm olabileceğini düşünüyorum. Kendi repomun reklamını da yapayım:

https://github.com/alperyazar/ebox

Elbette başka çözümler de mevcut.

4 Script ya da komut satırı tabanlı derleme sistemlerinin öneminden bahsettik. **Burada kendinizin sıfırdan bir şey geliştirmenizi, "in-house" bir build sistemi kullanmanızı önermiyorum.** Özel bir sebebiniz vardır, ekbiniz çok geniştir ve bunu yıllar boyunca idame ederim diyorsanız o başka. **Eklediğiniz her bir bileşenin technical debt oluşturduğunu unutmayın.** Bir gazla kendi TCL scriptleri ile bir şeyler yapmaya çalışıp, 1-2 senede patatese dönenleri gördüğüm için (mesela ben) anlık gazla build sistemi yazmaya çalışmayın. Öğrenmek için kendi kendinize takılın, ona bir şey demem. FuseSoC, HoG gibi çözümlere bakın. Eksik buluyorsanız onlara katkıda bulunun, *bu işimi görmüyor ya* deyip kestirip atmayın.

Bu maddeleri tek tek detaylandırmak yazıyı çok uzatacağı için genel hatları ile bahsetmek istedim.

Faydalı olması dileği ile...

Alıntıla

```
@misc{yazar2025blogfpgadevopsv1,
  title = {FPGA, Git, DevOps, CI/CD},
  author = {Yazar, Alper},
  year = {2025},
  url = {https://www.alperyazar.com/dow/fpga-git-cicd-devops-v1.pdf},
}
```

NOT

Bu dokümana https://www.alperyazar.com/dow/fpga-git-cicd-devops-v1.pdf adresinden erişebilirsiniz. Fakat dokümanın daha güncel hali yayınlanmış olabilir. Bunun için lütfen https://www.alperyazar.com/dow/fpga-git-cicd-devops.pdf adresini ziyaret ediniz. Eğer bu doküman güncel ise bu dokümanı, değilse daha güncel bir sürümünü bulabilirsiniz. Bu dokümanın sürümü **v1** olup **2025** tarihinde yayınlanmıştır.