



İçindekiler

1.	SİSTEM TANIMI VE TEMEL TASARIM ÖZETİ.....	3
2.	PROJE DETAY TASARIMI	4
2.1.	Sistem Mimarisi.....	4
2.2.	Tasarım Detayı	5
2.2.1.	İşlemci ve Bus Yapısının Tasarımı.....	5
2.2.2.	Peripheral Tasarım Detayları.....	6
2.2.2.1.	Çevre Birimi Belleği	6
2.2.2.2.	UART.....	8
2.2.2.3.	I2C Master	9
2.2.2.4.	QSPI Master.....	10
2.2.2.5.	Timer	12
2.2.2.6.	GPIO.....	13
2.2.3.	Yazılım Ortamı	13
2.2.3.1.	peripherals.h	13
2.2.3.2.	main.c.....	14
2.2.3.3.	linker_script.ld.....	15
2.2.3.4.	mem_file_gen.sh.....	16
2.2.3.5.	hex2mem.py.....	16
2.2.4.	FPGA Prototipleme Detayları	16
3.	ÇİP TASARIM AKIŞI.....	18
4.	TEST	20
4.1.	Simülasyon Testleri	20
4.1.1.	UART [UVM]	20
4.1.2.	GPIO	26
4.1.3.	Timer	26
4.1.4.	I2C Master	26
4.1.5.	QSPI Master	26
4.2.	Implementasyon Testleri.....	27
4.2.1.	GPIO	27
4.2.2.	Timer	27
4.2.3.	I2C Master	27
5.	TAKIM ORGANİZASYONU	28
5.1.	Takım Organizasyonu.....	28
5.2.	Görev Dağılımı.....	28
6.	İŞ PLANI ve RİSK PLANLAMASI.....	28
7.	KAYNAKÇA	29

1. SİSTEM TANIMI VE TEMEL TASARIM ÖZETİ

Bu proje, OpenHW tarafından geliştirilen 4 aşamalı CV32E40P RISC-V çekirdeğine çeşitli çevre birimlerini ekleyerek mikrokontrolcü yapmayı hedeflemektedir. Bu çevre birimleri şu şekildedir:

	RTL Tasarımı	UVM/Simülasyon Testi	Implementasyon Testi
UART	✓	✓	✗
I2C Master	✓	✓	✗
QSPI Master	✓	✓	✗
Timer	✓	✓	✓
GPIO	✓	✓	✓
JTAG	✗	✗	✗
USB	✗	✗	✗

Tablo 1: Çevre birimlerinin tamamlanma oranı

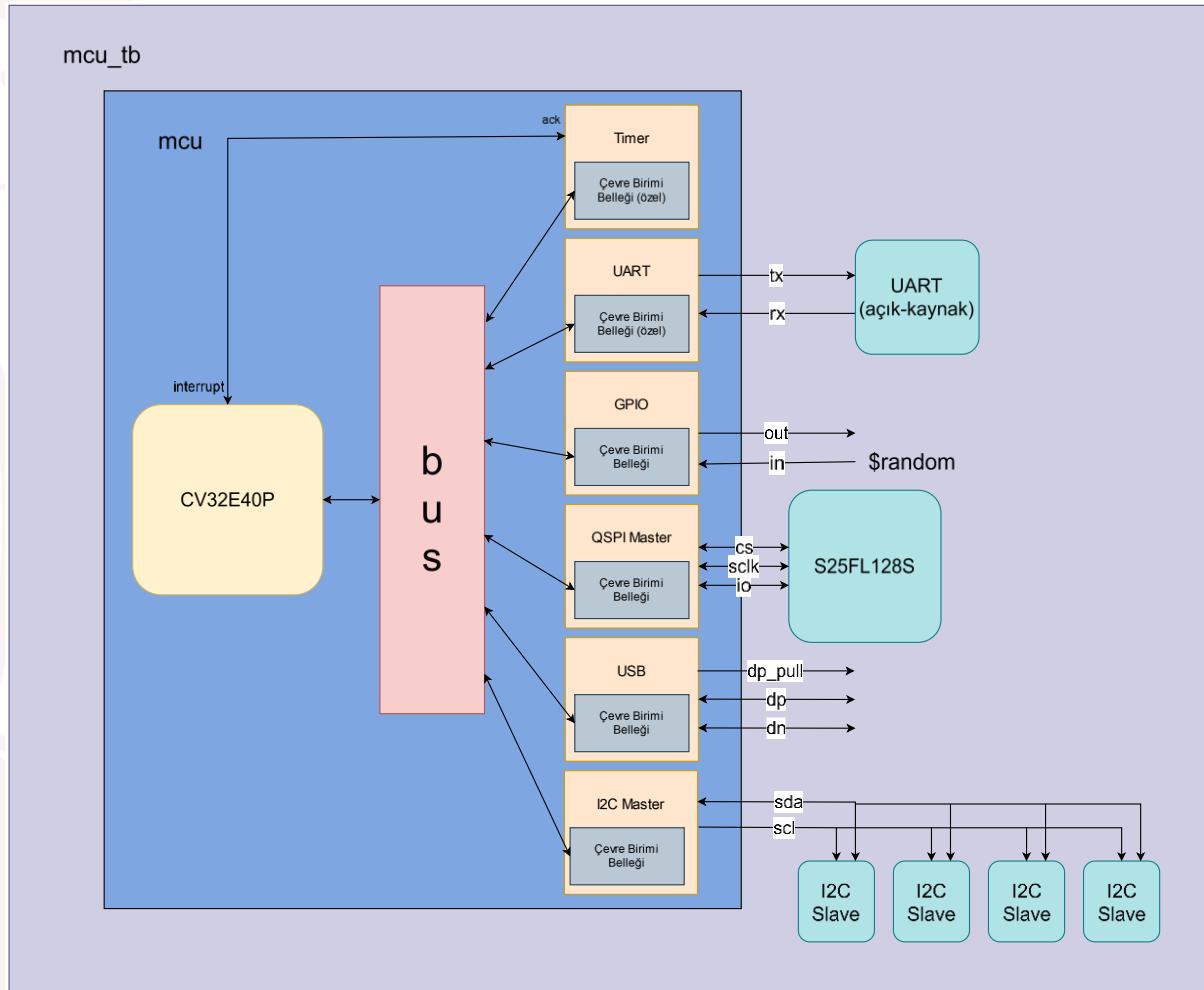
Yeşil renk tamamlandılığını, sarı renk devam ettiğini ve kırmızı renk başlanmadığını göstermektedir.

Bu çevre birimlerinin çekirdek ile bağlantısının kurulabilmesi için bus ve çevre birimi belleği modülleri kullanılmıştır. USB ve JTAG, DTR sonrasında bırakılmıştır. Sistemin şu anki halinin fiziksel tasarımları tamamlanmıştır.

2. PROJE DETAY TASARIMI

2.1. Sistem Mimarisi

İki farklı top modülüden bahsedebiliriz bir tanesi testbench için (mcu_tb). Diğerini, sentezlenebilir (mcu). Projenin tamamlanan kısmının blok şeması aşağıdaki şekildedir:



Şekil 1: Sistemin blok şeması

CV32E40P: Sistemin RISCV çekirdeği buradadır. Tüm işlemler burada yapılır.

Bus: Çevre birimleri ile çekirdek arasında aracılık yapar. Çekirdeğin tüm çevre birimlerine önceliği vardır yani tüm çevre birimleri çekirdek gözünden slave diyebiliriz.

Timer: Çekirdeğin interrupt yapabilmesini sağlar. Bu çekirdek ile gerçek zaman arasında bağlantı kurmak için önemlidir.

UART: UART protokolünü kullanarak veri aktarımını sağlar.

I2C Master: I2C protokolünü kullanarak veri aktarımını sağlar.

QSPI Master: QSPI protokolünü kullanarak Flash bellek ile (şart değil) iletişim kurulabilmesini sağlar.

USB: USB aygıtlarıyla iletişim kurabilmesini sağlar. Henüz bitmedi.

JTAG: Çekirdeğin debug edilebilmesini sağlar. Henüz başlanmadı, o yüzden şematikte yok.

Kendi yazdığımız bus yapısını kullandık. Az sayıda çevre birimi olduğundan bu bus yapısında okuma/yazma gecikmesi 1 cycle'dır. Çekirdek önce 0x2000 adresinden okur. Bu komut belleğindeki bootloader koduna denk gelmektedir. Bootloader kodu QSPI modülüne gerekli emirleri vererek komut ve veri belleklerini doldurur. x2 (ya da sp) yazmacını ayarlar ve 0x0000 adresine (komut belleğindeki ilk adrese) dallanır.

C programlarının derlenmesi için GCC kullanılmıştır. Yazılımcının çevre birimlerini rahat bir şekilde kullanabilmesi için yazmış olduğumuz header'ı eklemesi gereklidir. Sisteme uygun bir linker script de hazırlanmıştır.

Aslında FPGA ve ASIC arasındaki temel fark STA ve alandan geçiyor. Özetlemek gerekirse FPGA'de komut ve veri belleği Xilinx'in XPM modülleri kullanılarak oluşturulmuş oluyor yoksa sistem ciddi şekilde (%400->%40) yer kaplıyor. Aynı şekilde clock gate modülü için Xilinx'in BUFGCE modülünü kullanıyoruz.

Ayrıntılı bilgilere Tasarım Detayı bölümünde ulaşabilirsiniz.

2.2. Tasarım Detayı

2.2.1. İşlemci ve Bus Yapısının Tasarımı

mcu modülünde işlemci ve bus birbirlerine bağlıyor. Bu bağlantıların ne olduğunu detaylandırmak gerekirse:

- Komut belleği arayüzü portları
- Veri belleği arayüzü portları
- Interrupt portları

Önce APB + AHB kombinasyonunu denedik lakin karmaşık geldiği ve slave'lerin her zaman yanıt vereceğini gördüğümüz için bundan vazgeçtiğimiz. Aynı zamanda çok fazla çevre birimi olmadığı için bus yapısını kendimiz hazırladık. Hem bus'a hem de işlemciye saat ve reset sinyalleri gitmektedir. FPGA implementasyonunda saat sinyali FPGA'de halihazırda bulunan osilatörden, reset sinyali ise butondan gelmektedir (Elle etkinleştirilir). Bus'tan çıkan çevre birimi telleri mcu modülünden çıkış portu olarak verilmektedir. Bu portların karşılık geldiği pinler xdc dosyasında belirtilmiştir.

FPGA'de flip-flopların baştaki değerleri belirlenebilmektedir. Dolayısıyla reset sinyaline pek ihtiyaç yoktur, lakin ASIC'te aynı durum geçerli olmadığından yine de reset sinyali gereklükçe kullanılmaktan çekinilmemiştir.

Bus yapısını biz hazırladığımız için rahatça değişiklik yapabildik. APB protokolünü seçseydik çok değişiklik yapamazdık. Şartnamedeki çevre birimleri de APB'yi zorunlu tutacak cinsten değildi. Ayrıca bu tercih, bus'ın nasıl çalıştığını çok daha iyi anlamamızı sağladı. Dahası, APB+AHB kombinasyonuna göre çok daha az yer kaplıyor. Sıfırdan bus yapısı yazmak çok fazla hata yapmamıza sebep oldu. Hataları çevre birimlerini test ederken gördükçe düzelttik.

2.2.2. Peripheral Tasarım Detayları

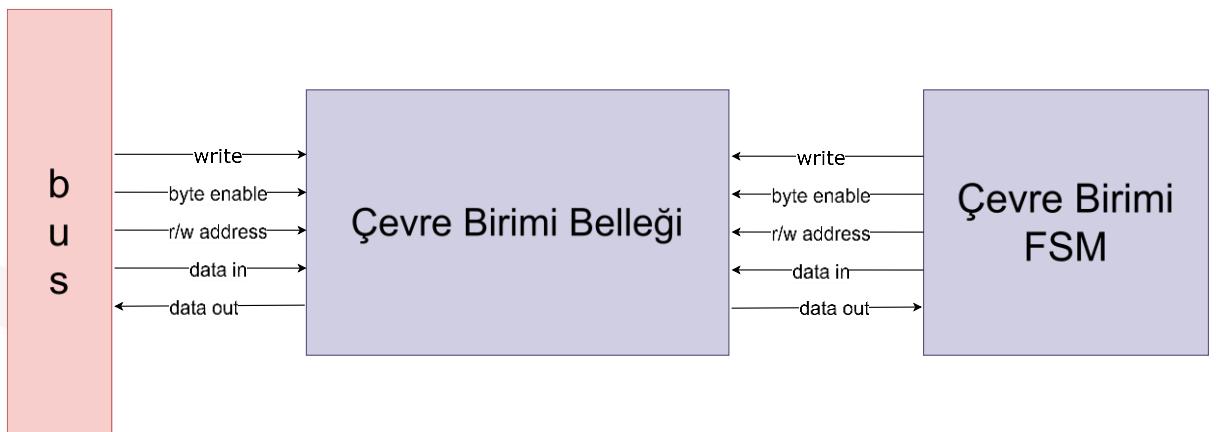
Hangi modülleri bizim yazdığını ve hangi modülleri hazır alacağımızı gösteren tablo aşağıdadır:

Bizim yazdığımız modüller	Hazır alacağımız modüller
UART	JTAG
I2C Master	USB
QSPI Master	
Timer	
GPIO	
Çevre Birimi Belleği	
Bus	

Tablo 2: Modüllerin listesi

2.2.2.1. Çevre Birimi Belleği

Bus'ı tasarlarken önce veri belleğini 8kB'tan daha büyük yapıp tüm çevre birimlerine erişim hakkı vermemi düşündük lakin bu sistemi inanılmaz karmaşık yapıyordu. Daha sonra çevre birimlerinin içerisinde yazmaç olarak tutmayı denedik. Bu sistem çalışıyordu fakat hata yapmaya çok müsaitti, debug etmesi çok zordu ve aynı anda tüm yazmaçları okumuyorduk. Dolayısıyla bir verimsizlik vardı. Ek olarak tüm çevre birimleri için benzer adımları tekrar etmek gerekiyordu. Tüm bunları çözen bir modül yazmaya karar verdik. Bu modüle "Çevre Birimi Belleği" dedik. Çevre Birimi Belleği, çekirdek ile çevre birimi arasında aracılık yapmaktadır. İçerisinde şartnameye belirtilen yazmaçları tutar. Hem çekirdek hem de çevre birimi yazma/okuma işlemleri yapabilir. Modülün blok şeması aşağıdaki gibidir:



Şekil 2: Çevre Birimi Belleği

Çevre birimlerinde sık sık rastlanan bir örüntü gördük: Okunmak istenen ile yazmak istenen yazmaçların her zaman aynı olmaması. Bu yüzden çevre birimlerine çekirdeğin bağlantılarına ek olarak okuma adresi koyduk. Çevre birimi belleğinin büyük bir dezavantajı olabileceğini fark ettik: Aynı adrese her iki taraftan da yazma isteği gelirse ne olacak?

Böyle bir durum söz konusu değil çünkü tüm çevre birimlerindeki yazmaçların fonksiyonları o kadar iyi belirlenmiş ki kullanım talimatlarına uyan bir yazılımcı çekirdekten çevre birimiyle çakışacak yazma isteği gönderemez. Şöyledir ki, çevre birimlerinde işlemin tamamlandığını söyleyen bir yazmaç var. Eğer işlem tamamlanmışsa/başlamamışsa, söz konusu çevre biriminin hiçbir yazmaca yazma emri vermeyeceğinin garantisini veriyoruz. Öteki taraftan, eğer işlem devam etmekte ise yazılımcı bize bunun garantisini vermelidir. Yazılımcıdan işlemi başlattıktan sonra işlemin devam etmekte olduğunu gösteren yazmacı okumadan diğer yazmaçlara yazma emri vermemesini istiyoruz. Bu anlaşma sayesinde, aradaki uyumsuzluk giderilir. Yazdığımız header'ı kullanan birisinin bunları hesaba katmasına gerek yoktur. Header bu husus göz önünde bulundurularak yazılmıştır.

Bazı yazmaçlar çekirdek tarafından yalnızca okunabilir. Hangi yazmaçların yazılabileceği ve yazmaçların toplamının büyüklüğü çevre birimi belleğine parametre olarak verilir.

2.2.2.2. UART



Şekil 3: UART waveform

Tek yönlü tel(ler) kullanır. TX teli karşı tarafa veri iletmek için, RX teli ise karşı taraftan veri almak için kullanılır. UART, oldukça basit bir protokoldür. Özetlemek gerekirse:

- Start biti gönderilir.
- Veri gönderilir.
- Parity bit gönderilir. [Opsiyonel]
- Stop biti gönderilir.

RX ve TX tellerinin çalışma prensibi tamamen aynıdır ve birbirinden bağımsız olarak çalışır. Dolayısıyla iki farklı FSM bulunmaktadır. Verici FSM şu şekilde çalışır:

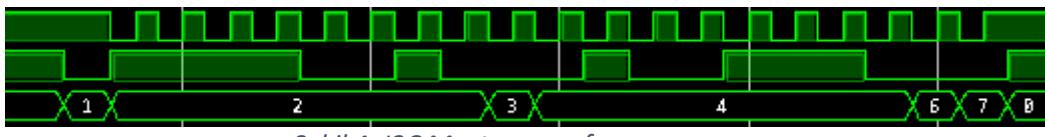
1. IDLE: TX teline 1 yaz. Geçerli yazma emri gelirse START state'ine geç.
2. START: Start biti oluşturmak için TX teline 0 yaz. Sayacı sıfırla. DATA state'ine geç.
3. DATA: TX teline bus'tan gelen 8-bitlik veriyi sayacı göz önünde bulundurarak LSB en başta olmak üzere sırasıyla yaz. Sayaç bitince STOP state'ine geç.
4. STOP: Stop biti oluşturmak için TX teline 1 yaz. Bus'tan gelen stop bit uzunluğu isteği kadar bekle. Seçenekler arasında 1.5 cycle da bulunduğu için STOP state'ine geldiğinde frekans 2 katına çıkar ve 1-1.5-2 cycle yerine sırasıyla 2-3-4 cycle bekler. TX teli açısından değişen bir durum olmaz.

Alicı FSM çok benzer şekilde çalışır:

1. IDLE: RX teli 0 olunca START state'ine geç.
2. START: Sayac sıfırla. Yarım cycle bekle ki okuduğun veriler tam ortadan alınsin. Metastabilite riskini minimuma indir.
3. DATA: RX telinden gelen 8-bitlik veriyi sayacı göz önünde bulundurarak LSB en başta olmak üzere sırasıyla geçici bir yazmaca yaz. Sayaç bitince STOP state'ine geç.
4. STOP: Geçici yazmactaki değeri, asıl yazmaca (UART_RDR) yaz.

Alicı FSM'de direkt asıl yazmaca yazmak yerine geçici bir yazmaca yazıp. Daha sonrasında bu geçici yazmacı asıl yazmaca yazmamızın bir sebebi var: Varsayılmış UART üzerinden bir sensöre sürekli veri okuma emri gönderiyorsunuz ve hedefiniz en güncel veriyi almak olduğu için sürekli UART_RDR yazmacını okuyorsunuz. Eğer alıcı FSM DATA state'indeyken okursanız okuduğunuz veri hem eski bitleri hem de güncel bitleri size verir. Bu da çok farklı sayılarla denk gelebilir. Her ne kadar peripherals.h header'ında buna karşın önlem alınmış olsa da kullanılması zorunlu olmadığı için donanım seviyesinde bu önlem alınmıştır.

2.2.2.3. I2C Master



Şekil 4: I2C Master waveform

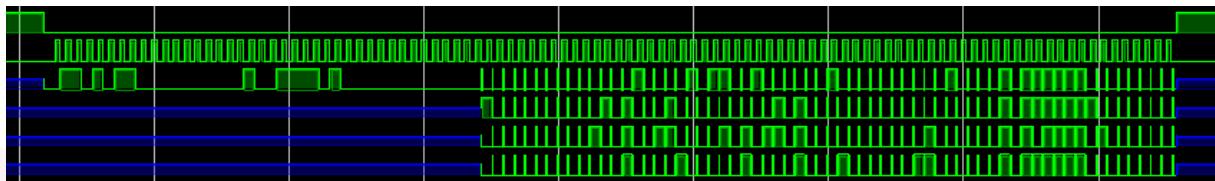
SDA ve SCL adında iki tel kullanılır. SDA çift yönlüdür. SCL, tek yönlüdür. Normalde SCL de çift yönlüdür fakat sistemde tek bir master olduğu için SCL teline yazan sadece bir tane modül vardır. I2C protokolünü özetlemek gerekirse:

- SCL 1 iken SDA 0 olursa başlangıç anlamına gelir.
- SCL 0 iken SDA 1 olursa bitiş anlamına gelir.
- Başlangıç ve bitiş arasında SCL 1 iken SDA değişemez.
- Adresler 7 bittir.
- Veriler 8 bittir.
- Sırasıyla “başlangıç + adres + okuma + ack + veri + ack/nack + bitiş” şeklindedir. 8 bitten daha fazla veri göndermek istenirse “veri + ack/nack” tekrarlanır.

Bu çevre biriminin içerisinde 8 state'den oluşan FSM bulunmaktadır. Stateler sırasıyla şu şekildedir:

1. IDLE: I2C_CFG yazmacında tamamlanmamış yazma ya da okuma emri varsa START state'ine geç. Yalnızca okuma emri geldiği zaman oku (Yazma öncelikli).
2. START: 3 bitlik sayacı sıfırla. SCL teline master clock'tan oluşturulmuş senkron frekansı düşük clock sinyalini ver. Start condition oluşması için SDA teline 0 yaz. ADDR state'ine geç.
3. ADDR: I2C_ADR yazmacındaki adresi SDA teli üzerinden gönder. Sayaç tamamlandığında ACK0 state'ine geç.
4. ACK0: Sayacı sıfırla. SDA teline yazmayı bırak. SDA teli 0'lanmışsa (ACK) bir sonraki state'e geç. Bir sonraki state yazma yapılacaksa WDATA yoksa RDATA'dır.
5. WDATA: I2C_TDR yazmacından bir byte'lık veri gönder. Sayaç tamamlandığında ACK1 state'ine geç.
6. RDATA: SDA telinden I2C_RDR yazmacına bir byte'lük veri yaz. Sayaç tamamlandığında ACK1 state'ine geç.
7. ACK1: Sayacı sıfırla. I2C_NBY yazmacındaki sayı kadar (1-4 aralığına göre düzenlenmiş şekilde) önceki state'i tekrarla. Yazma modundaysak ve SDA telinden ACK sinyali gelmemişse tekrar göndermeyi dene (işlemin komple iptal edilemesine tercih edilmiştir, ileride değişiklik yapılabılır). Tekrarlar tamamlandıysa STOP state'ine geç. Okuma işlemi yapıyorsak son byte hariç ACK gönder. Son byte'ta slave'in daha fazla istemediğimizi anlaması için NACK gönder. Yazma işlemi yapıyorsak SDA teline yazmayı bırak.
8. STOP: STOP condition oluşması için SDA teline 1 yaz. SCL teline de gecikmeli 1 yaz. İşlemin tamamlandığını belirtmek için I2C_CFG yazmacını düzelt. IDLE state'ine dön.

2.2.2.4. QSPI Master



Şekil 5: QSPI Master waveform

QSPI'da SCLK telinden saat sinyali, CSn telinden select sinyali ve io tellerinden (4 tane) veri aktarılır. Şartname açısından QSPI protokolünü özetlemek gerekirse io tellerine yazılanlar sırasıyla şu şekildedir:

1. 8 bitlik komut gönderilir.
2. 24 bitlik adres gönderilir. [Opsiyonel]
3. Dummy cycle('lar) gönderilir. [Opsiyonel]
4. Veri gönderilir/alınır. [Opsiyonel]

QSPI Master modülünün yükünü hafifletmek için oldukça esnek bir şekilde yazdık. QSPI Master modülünün konfigürasyonunu tamamen yazılıma bıraktık. İçerisinde 4 state'den oluşan FSM bulunmaktadır. State'ler sırasıyla şu şekildedir:

1. IDLE: Sayacı sıfırla. QSPI_STA yazmacına işlemin bittiğini belirten biti yaz. io tellerinden sadece 1.sine (boş) veri yazılabilir.
2. CMD: QSPI_STA yazmacına meşgul olduğunu belirten biti yaz. X0 modunda ısek IDLE state'ine geç(bitir). Dummy cycle istendiyse DUMMY state'ine geç, yoksa EXECUTE state'ine geç.
3. DUMMY: QSPI_ADR yazmacındaki adresi oku. Okunan bitleri sırasıyla io bağlantısındaki 1. tele yaz. Eğer istenilen dummy cycle adresin uzunluğundan fazlaysa başa dön ve tekrar oku. İstenilen cycle kadar beklendiğinde EXECUTE state'ine geç.
4. EXECUTE: Sayacı sıfırlanınca IDLE state'ine dön.
 - a. Okuma modu:
 - i. x1: 4 bitlik io bağlantısının 2. telinden okuduğun veriyi sayacı göz önünde bulundurarak ilgili bölüme yaz. Sayacı 1 azalt.
 - ii. x2: 4 bitlik io bağlantısının ilk 2 telinden okuduğun veriyi sayacı göz önünde bulundurarak ilgili bölüme yaz. Sayacı 2 azalt.
 - iii. x4: 4 bitlik io bağlantısının tüm tellerinden okuduğun veriyi sayacı göz önünde bulundurarak ilgili bölüme yaz. Sayacı 4 azalt.
 - b. Yazma modu:
 - i. x1: 4 bitlik io bağlantısının 1. teline QSPI_DRx yazmacından okuduğun veriyi sayacı göz önünde bulundurarak yaz. Sayacı 1 azalt.
 - ii. x2: 4 bitlik io bağlantısının ilk 2 teline QSPI_DRx yazmacından okuduğun veriyi sayacı göz önünde bulundurarak yaz. Sayacı 2 azalt.
 - iii. x4: 4 bitlik io bağlantısının tüm tellerine QSPI_DRx yazmacından okuduğun veriyi sayacı göz önünde bulundurarak yaz. Sayacı 4 azalt.

QSPI Master'da adresin yazımıyla dummy cycle'in yazımını aynı state'de yapıyoruz. Bunun sebebi, dummy cycle'da io bağlantısının 1. teline yazılan verinin önemsiz olmasıdır. Böylece

hem adresi göndermeyi hem de dummy cycle yazmayı aynı devre ile yaparak alandan tasarruf ediyoruz. Yazdığımız bu modülün içerisinde komutları gömmedigimiz için yalnızca şartnamede belirtilen S25FL128S flash belleği ile değil, birçok bellekle uyumlu bir şekilde çalışabiliyor. Bu anlatılanlar göz önünde bulundurulduğunda QSPI_CCR yazmacına atamalar şu şekilde yapılıyor:

1. QSPI_CCR[9:8] bölümüne veri modu yazılıyor. Veri modu EXECUTE state’inde kaç tane tel kullanılacağına göre belirtiliyor. Eğer sadece komut gönderilecekse 0 yazılması gerekiyor.
2. QSPI_CCR[15:11] bölümünü önce şartnamedeki gibi içerisinde yazan sayı kadar cycle diye baz aldık ama sonra fark ettik ki bu mümkün değil. Çünkü 5 bitlik yazılabilecek en yüksek sayı 31 ve S25FL128S flash belleğinde 24b adres + 8 dummy cycle = 32 cycle DUMMY(bizim FSM cinsinden) gerekiyor. Adres 24b ya da 32b. İki sayı da 8’e bölünüyor. Kullanıcının 8’in katı olan bir sayı kadar dummy cycle isteyeceğini baz alıp yazılan sayının 8 katı kadar DUMMY state’inde kalıyoruz. Dolayısıyla CMD ve EXECUTE state’leri arasında minimum 0, maksimum 256 cycle bekleyebilirsiniz.
3. QSPI_CCR yazmacının geri kalan kısmına ek bir yorumumuz yok.

Bu açıklamalar göz önünde bulundurulduğunda şöyle bir tablo çıkıyor karşımıza:

// WREN	8'h06 + 8b cmd	133Mhz x0 xxxx xxxx xxxx
// WRDI	8'h04 + 8b cmd	133Mhz x0 xxxx xxxx xxxx
// CLSR	8'h30 + 8b cmd	133Mhz x0 xxxx xxxx xxxx
// RESET	8'hF0 + 8b cmd	133Mhz x0 xxxx xxxx xxxx
// RDID	8'h9F + 8b cmd	- 648b data 133Mhz x1 read 0dummy 0-31
// RDSR1	8'h05 + 8b cmd	- 8b data 133Mhz x1 read 0dummy 0
// RDSR2	8'h07 + 8b cmd	- 8b data 133Mhz x1 read 0dummy 0
// RDCR	8'h35 + 8b cmd	- 8b data 133Mhz x1 read 0dummy 0
// READ_ID	8'h90 + 8b cmd + 24b addr	- 16b data 133Mhz x1 read 3dummy 1
// RES	8'hAB + 8b cmd + 24b dummy	- 8b data 50Mhz x1 read 3dummy 0
// WRR	8'h01 + 8b cmd	+ 16b data 133Mhz x1 write 0dummy 1
// READ	8'h03 + 8b cmd + 24b addr	- inf 50Mhz x1 read 3dummy 0-31
// PP	8'h02 + 8b cmd + 24b addr + 256B data	133Mhz x1 write 3dummy 0-31
// SE	8'hD8 + 8b cmd	+ 24b data 133Mhz x1 write 0dummy 2
// DOR	8'h3B + 8b cmd + 24b addr + 8b dummy	-inf 104Mhz x2 read 4dummy 0-31 -inf
//		
// QOR	8'h6B + 8b cmd + 24b addr + 8b dummy	-inf 104Mhz x4 read 4dummy 0-31 -inf -inf -inf
//		
//		
// QPP	8'h32 + 8b cmd + 24b addr + 64B data	80Mhz x4 write 3dummy 0-31 + 64B data + 64B data + 64B data
//		
//		
//		

Denklem 1: QSPI Master S25FL128S analizi

2.2.2.5. Timer

Çekirdek interrupt işlerini Machine CSR'lar ile halleder. Machine CSR'ların başka görevleri olanları da var ama biz doğrudan sadece 3 tanesiyle ilgileniyoruz. Dolaylı olarak ilgilendiğimiz birçok CSR yazmacı da var ama burada bundan bahsetmedik. Değişiklik yaptığımız CSR yazmaçları ve görevleri şu şekildedir:

Machine CSR'lar	Görevleri
mtvec	Interrupt olduğunda burada tutulan adrese dallanılır.
mstatus	Mie CSR'ını aktif etmek de dahil birkaç ayarı mevcuttur.
mie	Machine interruptları (timer bunlardan biri) burada aktif edilir.

Tablo 3: Machine CSR'lar ve görevleri

Yazılımcının Timer modülünü kullanması için peripherals.h headerında yazmış olduğumuz timer_conf(...) fonksiyonunu kullanmış olması gereklidir. Bu fonksiyonun görevlerinden biri CSR yazmaçlarını ayarlamasıdır. mtvec yazmacına 0x2200 (ayrintılı bilgi linker bölümünde) yazılır. mstatus yazmacının 3. pini aktif edilir. Bu pin machine interruptlarının değerlendirilmesini sağlar. Machine interruptlarından da ihtiyacımız olanı aktif etmemiz gerekiyor. Bu ayarı da mie yazmacından yapıyoruz. mie yazmacından 7. pini aktif ediyoruz. Bu pin Timer interruptlarına denk geliyor. Böylece çekirdek Timer interruptlarını almaya hazır oluyor. Interrupt fonksiyonuna girince mie yazmacının 7. biti mret gelinceye kadar (fonksiyondan çıkışına kadar) 0 oluyor. Bu interrupt içinde interrupt olmasını engellemek içindir. OpenHW, nested interruptın yapılmış yapılamayacağı kullanıcıyı bırakmıştır. Şartnamede bu durumdan bahsedilmemiği için biz varsayılan ayar olan interrupt içinde interrupt isteğinin umursanmaması durumunu değiştirmedik.

Timer modülü, çekirdeğe interrupt sinyali verir. Çekirdek, Timer'a interrupt sinyalının kabul edilip edilmediğini ve kabul edildiyse kabul edilen interruptın id'sini verir. Timer, kabul edilen interrupt'ın id'si 7 mi diye kontrol ediyor sürekli. Henüz 7 dışında bir interrupt geldiğini görmedik ve olacağını düşünmüyorum ama yine de bu kontrol mekanizmasını kaldırmadık. Çekirdek interrupt sinyalının kabul edildiğini söyleyene kadar Timer interrupt sinyali vermektedir. Timer'ın prescaler değerinin düşük olması interrupt fonksiyonu bitmediği için interrupt sinyallerinin gözden kaçırılmasına yol açabilir. Bunu göz önünde bulundurmak yazılımcının sorumluluğundadır. Gerekirse Timer interrupt'ın içine girildiğinde devre dışı bırakılmalıdır.

TIM_CLR yazmacına 1 yazılırsa TIM_CNT yazmacı ve sayaç sıfırlanır. TIM_EVC yazmacına 1 yazılırsa TIM_EVN sıfırlanır. TIM_CLR ve TIM_EVC yazmaçları 1 cycle sonra 0'a çekilir.

Timer'ın içerisinde 32 bitlik bir sayaç bulunur. Bu sayaç TIM_PRE sayacına yazılıan değere ulaşana kadar artar. TIM_PRE yazmacına -1 yazılmışsa (saat frekansı – 1) değerine ulaşıcaya kadar artar.

TIM_ENA yazmacı 1 ise ve sayaç TIM_PRE'ye eşitse:

- Sayaç sıfırlanır.
- TIM_MOD 1 ise TIM_CNT 1 arttırılır. Değilse, 1 azaltılır.
- TIM_ARE, TIM_CNT'ye eşitse:
 - TIM_CNT sıfırlanır.
 - TIM_EVN 1 arttırılır.
 - Interrupt sinyali verilir.

2.2.2.6. GPIO

Oldukça basit bir çevre birimidir. İçerisindeki çevre birimi belleğinde 2 tane yazmaç bulunur. in tellerinden gelen veriyi GPIO_IDR yazmacına yazar. GPIO_ODR yazmacına yazılan veriyi out tellerine verir. Bahsi geçen tellerin yönü sabittir.

2.2.3. Yazılım Ortamı

Hazırlamış olduğumuz sistemi daha rahat programlayabilmek adına bir yazılım ortamı oluşturduk. Bu yazılım ortamı şu dosyalardan oluşmaktadır:

- peripherals.h
- main.c
- linker_script.ld
- mem_file_gen.sh
- hex2mem.py

2.2.3.1. peripherals.h

C dilinde yazılmış header dosyasıdır. Bu header dosyası çevre birimlerinin kolayca ve güvenli bir şekilde kullanılması içindir. İçerdiği fonksiyonlar ve kullanım şekilleri:

- UART:
 - Konfigüre et
 - Yaz
 - Oku
- I2C:
 - Konfigüre et
 - Yaz
 - Oku
- QSPI:
 - S25FL128S WREN
 - S25FL128S WRDI
 - S25FL128S CLSR
 - S25FL128S RESET
 - S25FL128S RDID
 - S25FL128S RDSR1

- S25FL128S RDSR2
 - S25FL128S RDCR
 - S25FL128S READ_ID
 - S25FL128S RES
 - S25FL128S WRR
 - S25FL128S READ
 - S25FL128S PP
 - S25FL128S SE
 - S25FL128S DOR
 - S25FL128S QOR
 - S25FL128S QPP
 - Yaz [S25FL128S olmak zorunda değil]
 - Oku [S25FL128S olmak zorunda değil]
- Timer:
 - Konfigüre et
 - Sayacı oku
 - Event'i oku
 - Sayacı sıfırla
 - Event'i sıfırla
 - Çalıştır
 - Durdur
 - GPIO:
 - Yaz
 - Oku
 - Veri Belleği
 - Yaz

USB'nin fonksiyonları henüz tasarılanmadı.

2.2.3.2. main.c

Yazılımcının bu dosyada main fonksiyonunu tutması beklenir. Main fonksiyonundan çıkış olmamalıdır. Aksi takdirde olacak durumdan yazılımcı sorumludur. Main fonksiyonundan çıkış olmayacağı bile bazen GCC bunu anlayamaz. Gereksiz yükleme ve depolama komutları yazılmaması için naked attribute'u verilebilir.

Interrupt yapılmak isteniyorsa interrupt fonksiyonunun adı interrupt olmak zorundadır. Yazılımcının yükünü hafifletmek için interrupt fonksiyonuna interrupt("machine") attribute'u verilmesini tavsiye ediyoruz. Çünkü interrupt fonksiyonlarında sözde ret komutu ra yazmacındaki adrese gitmenizi sağlar. Halbuki interrupt fonksiyonlarında bu anlamsızdır çünkü dönülecek adres ra yazmacında tutulmaz, MEPC yazmacında tutulur. Kullanılması gereken komut mret idir. Ayrıca değiştirilen her yazmaç (klasik fonksiyonların aksine istisnasız bir şekilde) interrupt gerçekleştiği andaki haline döndürülmelidir. Bu attribute, bütün bunları sizin için yapar.

Çevre birimlerini kullanmak istiyorsanız peripherals.h'yi programınıza eklemenizi tavsiye ederiz. Çevre birimlerinin nasıl çalıştığını anlamadan yalnızca istenilen değerleri girerek çevre birimlerini kullanabilirsiniz.

2.2.3.3. linker_script.ld

Linker script dosyasıdır. Oldukça küçük bir dosyadır. Linker'a nasıl davranış gereğini söyler. Bildiğimiz kadarıyla, GCC'nin linker'ı yalnızca Von Neumann mimarisini destekliyor. Dolayısıyla burada ekstra bir işlem yapmamız gerekiyor. MCU ve Linker gözünden iki farklı bakışımız var:

	Harvard (MCU)		Von Neumann (Linker)
	Bus	Komut Belleği	
0x0000	Veri Belleği	bootloader	Veri Belleği
0x2000	<i>kullanılmıyor</i>	main()	main()
0x2200		interrupt() + diğer()	interrupt() + diğer()
0x4000	Komut Belleği	----->	Komut Belleği
0x6000	UART	----->	UART
0x8000	I2C	----->	I2C
0xA000	QSPI	----->	QSPI
0xC000	Timer	----->	Timer
0xE000	USB	----->	USB
0x10000	GPIO	----->	GPIO

Tablo 4: Linker script analizi

Linker'a çevre birimlerinin adresi olduğu gibi verildi. İşlem yetkisi yok, sadece okuma ve yazma var. Dolayısıyla bu birimlere erişim sadece veri belleği adresleri üzerinden yapılıyor (Memory-mapped). Bootloaderı GCC'de normal yazdık. Daha sonra adresleri elle 0x0000'a hizalı bir şekilde çevirdik ve bu şekilde kaydettik. Veri belleğinin başlangıç adresinin 0x0000 olduğunu görüyorsunuz. Bitisi 0x2000'e denk geliyor. GCC verileri sp yazmacını eksilterek yazıyor. Bootloaderın komut belleğine dallanmadan önce sp yazmacına 0x2000 yazmasının sebebi işte bu. Interrupt fonksiyonunun adresini sabit 0x2200 olarak belirledik. Eğer main fonksiyonu 0x2000 ile 0x2200 arasına sızmazsa GCC hata veriyor. Bu yüzden main fonksiyonunun gereklirse parçalanmasını tavsiye ederiz. Diğer fonksiyonlar interrupt fonksiyonunun hemen ardından başlıyor (0x4000'e kadar). 0x4000'deki komut belleği sizi şaşırtmasın. Burası yalnızca komut belleğine yazmak için ayrılan adres. 0x4004 ve sonrası umursanmıyor.

Lakin veri belleğinde bir sorunumuz var. GCC, yazdığınız programın veri kısmını 0x0000'dan başlayarak hex dosyasına yazıyor (.dynamic, .sdata vb. şekillerde). Bootloader, bu verileri veri belleğine yazıyor. Lakin veri belleğine aynı zamanda 0x2000'den aşağı yönde giden veriler de yazılıyor program çalışırken. Linker, sp yazmacının ilk baştaki değerini bilmiyor. Dolayısıyla bu iki veri kümesi çakışırsa yazılımcıya veri belleği taşması hatası verilmiyor. Açıkçası biz veri belleğini sonuna kadar kullanamadık ama yazılımcının yine de bunu göz önünde bulundurması gereklidir.

2.2.3.4. mem_file_gen.sh

main.c yazıldıktan sonra çalıştırılması gereken dosya budur. GCC'yi uygun parametrelerle çalıştırır. hex2mem.py dosyasını çalıştırır.

2.2.3.5. hex2mem.py

mem_file_gen.sh dosyasının ürettiği hex uzantılı dosyadan mem dosyaları üretir. Bu dosyalar şunlardır:

- s25fl128s.mem
- instr_mem_no_flash.mem
- data_mem_no_flash.mem

s25fl128s.mem flash belleğe yüklenmesi gereken dosyadır. İçerisinde .text bölgümleri ve .data bölgümleri bulunur. .data olanlar 0x0000 ile 0x2000 arasına yazılır. .text olanlar 0x2000 ile 0x4000 arasına yazılır. *no_flash.mem olanlar ise implementasyon için flash bellek devre dışı bırakıldığında ilgili belleğin ilk baştaki değerlerini tutar.

2.2.4. FPGA Prototipleme Detayları

Elimizde S25FL128S model Flash bellek olmadığı için FPGA'e gömemezdim diye düşündük. Fakat Vivado, boyutu çok büyük olduğu için sentezde hata verdi. Bu yüzden "defines.vh" headerine "define NO_FLASH" diye bir satır ekledik. Bu satır FPGA implementasyonunda Flash belleğe önceden yazılmış olacağını varsayıdığımız programı komut belleğine yazıyor. Aslında bir nevi QSPI protokolünü baypas ediyoruz. Bootloader kodu da sadece komut belleğine dallanmak için programlanıyor.

FPGA olarak Digilent Basys3 (A7-35T) kartını seçtik. Bağlantıları şu şekilde yaptık:

- Osilatör (pin W5): Saat
- Anahtarlar: GPIO input
- LEDler: GPIO output
- Buton (pin U18): Reset
- JA: I2C portları
- JB: QSPI portları
- JC: UART portları

USB tamamlanmadığı için port ayırmadık. FPGA kaynaklarının kullanım miktarları şu şekilde:

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs	8673	0	0	20800	41.70
LUT as Logic	6241	0	0	20800	30.00
LUT as Memory	2432	0	0	9600	25.33
LUT as Distributed RAM	2432	0			
LUT as Shift Register	0	0			
Slice Registers	3288	0	0	41600	7.90
Register as Flip Flop	3288	0	0	41600	7.90
Register as Latch	0	0	0	41600	0.00
F7 Muxes	1030	0	0	16300	6.32
F8 Muxes	265	0	0	8150	3.25

Tablo 5: FPGA kaynaklarının kullanım miktarları

Timer'ı konfigüre eden ve sonrasında periyodik olarak GPIO'nun giriş portlarından veri okuyup bunu işleyerek (toplama, çıkışma, kaydırma komutları vb. şekilde yorumlayarak) GPIO'nun çıkış portlarına yazan bir program yazdık. Programı derledik ve sentez sırasında çıkan kritik uyarıları çözdük. Sonrasında implementasyon ve bitstream üretim aşamalarını başarılı bir şekilde tamamladık. Bu test sayesinde çekirdek, bus, bootloader kodunu (dallanma yapıp yapmamasını), timer modülünü, işlemcinin interrupt isteklerine yanıt verip vermediğini, GPIO modülünü, linker scripti, program header'ı ve yazdığımız C kodunu doğrulamış olduk.

I2C modülünü Arduino UNO'yu slave olarak bağladıkten sonra test ederken ACK sinyali ile ilgili bir sorun yaşadık. Sorunun FPGA – Arduino uyumsuzluğu olduğunu anladık. DTR'den sonraya bıraktık.

UART ve USB'nin implementasyon testlerini DTR'den sonraya bıraktık.

WNS, -15ns civarında çıkıyor. Bu da maksimum saat frekansı yaklaşık 67MHz demek. En kötü 10 WNS'ye baktık ve birçoğu çekirdeğin içerisindeydi. O yüzden sistemi optimize etme yoluna gitmedik. Implementasyon testlerini 60MHz saat frekansı ile yaptık. USB için de 60MHz olması gerekiyordu. Ek bir devreye ihtiyaç duymadık.

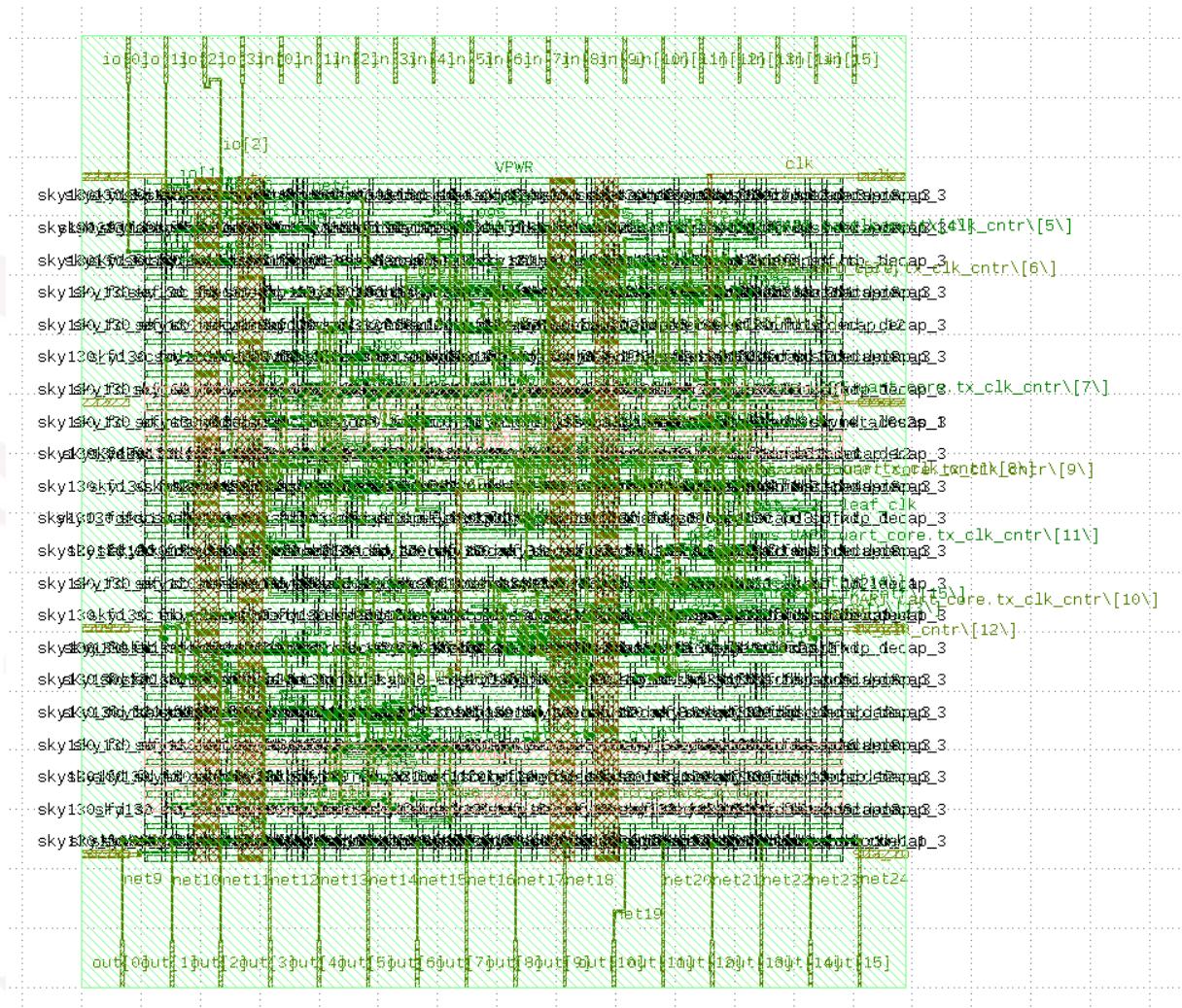
3. CİP TASARIM AKIŞI

Projeden bir tasarım çıktı dosyası alabilmek için OpenLane adlı açık-kaynak program kullandık. Bu program kullanırken karşılaştığımız bir sürü problemler, hatalar oldu. Özellikle açık-kaynak bir program olmasından kaynaklı olmasının etkilerivardı bu durumda. Öncelikli olarak Yosys programının SystemVerilog dosyalarını desteklemiyor olmasından kaynaklı olarak bize verilen çekirdek dosyaları sentez aşamasını geçemiyordu. Bu durumla ilgili olarak birkaç farklı çözüm önerisi ile geldik.

Öncelikli olarak tasarım çıktı dosyasının içine çekirdek programlarını dahil etmeden bir tasarım çıktı dosyası oluşturmak denenen çözümlerden biriydi. Bu durum yazdığımız çevre birimlerinin ne kadar alan kapladığını ve çalışıp çalışmadığını görmemizi sağlayan bir test olmasını sağladı. Rapor içerisinde gösterilen resimler, çekirdek kullanılmadan oluşturulan tasarım çıktı dosyaları kullanılarak alınmıştır.

Diğer problemi çözmek için kullanılan yöntemlerden biri SystemVerilog dosyalarını Yosys ile sentezleyip tasarım çıktı dosyalarını elde etmektı. Ancak Yosys'in kendi içinde desteklediği SystemVerilog özelliklerinden daha fazlasını kullanılarak oluşturulmuş olan SystemVerilog dosyaları yüzünden Yosys bu dosyaları sentezleyemedi. Onun yerine Yosys'in üzerine çalışılarak oluşturulmuş olan başka bir program olan synlig [1] programını kullanmayı denedik. Bu repository Yosys'e SystemVerilog desteği eklemek için yapılmış bir programdı. Bu programı docker konteynörünün içerisinde kurup çalıştırmayı denedik. Bu kısımla alakalı olarak OpenLane'in iş zincirinin içerisinde entegre etmekle alakalı bir problem yaşadık. Belki bu problem üzerine daha fazla vakit harcansa çözülebilirdi.

Son olarak bir diğer çözüm ise SystemVerilog dosyalarını Verilog dosyalarına çevirmekti. Bunu el ile çevirmek pek mümkün olmadığı için bazı araçlar kullanmayı denedik. Bunu çevirmeyi sağlayan bir program olarak sv2v [2] vardı. Ancak bu program bazı SystemVerilog parametelerini desteklemediği için işimize yarayan yeterli bir çözüm değildi.



Şekil 6: Tasarım çıktısı

4. TEST

Yaptığımız testler simülasyon testi ve implementasyon testi olarak ikiye ayrılmaktadır. Simülasyon testinden başarıyla geçen modüller genelde implementasyon testinden de başarıyla geçmektedir fakat gerek FPGA'in yapısal farklılıklarını gerek bazı tellerin harici komponentlere ihtiyacı olmaları vb. durumlar başarısızlıkla sonuçlanmasına yol açmaktadır.

Simülasyon ve implementasyon testleri, daha önce de belirtildiği şekilde 60MHz saat frekansında yapılmıştır.

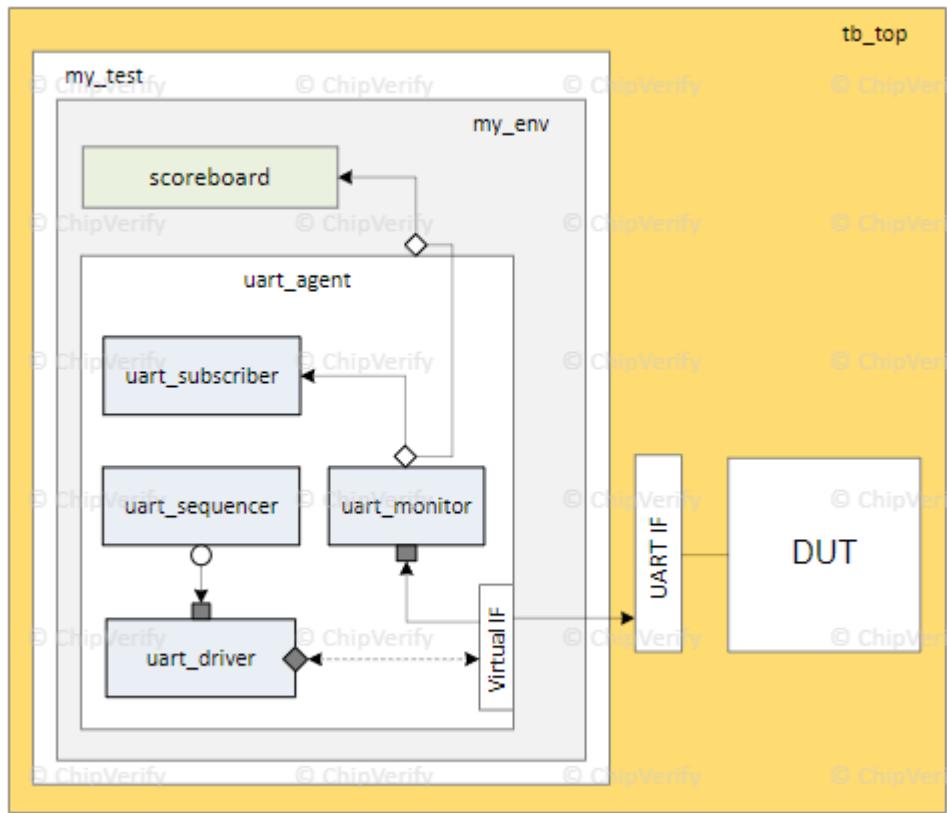
4.1. Simülasyon Testleri

Simülasyon testleri, Sistem Mimarisi bölümündeki blok şemasında gösterildiği üzere çevre birimlerinin bağlanacağı cihazların da simülasyona dahil edilmesiyle yapılmıştır. Simülasyon testleri, Vivado 2023.2'nin simülasyon aracı ile yapılmıştır. Simülasyon testinin başarılı olup olmadığı waveform incelenerek anlaşılmıştır. Hataların da bu şekilde farkına varılmıştır.

İşlemcinin doğrulanması için ek bir test yapılmamıştır. Çevre birimleri test edilirken işlemci hep olması gerektiği gibi davranışmıştır. İşlemcide bir hata çıksayıda da düzeltmemiz pek mümkün olmazdı.

4.1.1. UART [UVM]

UART protokolünün doğrulanması için UVM metodu izlenmiştir. Aşağıdaki figürde benzeri bir UVM ortamının yukarıdan görünümü bulunabilir. Oluşturulan test ortamı scoreboard olmaması haricinde bu hiyerarşik yapıyı izlemektedir ve bu ortam uart_top modülünün doğrulanması için oluşturulmuştur. Figürden sonra oluşturulan UVM ortamının sınıf ve bileşen bazında detaylı açıklamasını bulabilirsiniz.



Şekil 7: Kuşbaşı UART UVM [3]

uart_top (DUT):

Doğrulama altındaki gerçek UART mantığını içerir.

uart_interface:

uart_core modülü için saat, reset ve veri hatlarını içeren iletişim sinyallerini tanımlar. DUT ile iletişim bu arayüzden kurulmaktadır.

uart_trans:

İlgili veri ve meta verilerle birlikte bireysel UART işlemlerini kapsar. Sinyaller üzerinde bazı kısıtlamalar ve modifikasyonlar haricinde arayüzdeki sinyaller oluşturulmuştur.

uart_sequence:

DUT'u uyarmak için sıra öğeleri işlemlerinin akışını yönetir. Sistemin fonksiyonellliğini test etmek için gönderilen verilerden bazıları burada randomize edilmek için ayarlanmaktadır.

uart_sequencer:

Sıra öğelerinin sürücüye gönderilme sırasını ve zamanlamasını kontrol etmektedir. Sequencer, sequence sınıfının içinde tanımlanmıştır.

uart_driver:

Sıra öğeleri işlemlerini DUT arayüzünde pin seviyesi aktivitelerine dönüştürmektedir. Aynı zamanda da scoreboard sınıfı ayrı implemente edilmek yerine burada implemente edilmiştir. Her test için cpb sayısı parametrelerle ayarlanabilmektedir. Regresyon testleri rx ve tx için her testte ayrı ayrı yapılmaktadır. İleri bölümlerde buna daha detaylı değinilecektir.

uart_monitor:

Pin seviyesi aktiviteleri gözlemler ve analiz için bunları tekrar sıra öğelerine dönüştürür. Oluşturduğu sıra öğelerini “functional coverage” raporu oluşturmak için subscriber sınıfına gönderir.

uart_cov (subscriber):

Monitor tarafından gözlemlenen sekansları alır ve bu sekansların tasarımın ne kadarını kullandığı bilgisi depolar. En son bütün testler bittiğinde de bu bilgileri “functional coverage” olarak raporlar.

uart_agent:

Driver, subscriber, sequencer ve monitorü kapsar. Driver ve sequencer bağlantısı ile monitor ve subscriber bağlantısı burada yapılmaktadır.

uart_env:

Ajanlar ve diğer bileşenleri içeren testbench altyapısını kuran üst düzey ortamdır.

uart_test:

Ortamı yapılandırmak, test senaryolarını ve başlangıç koşullarını tanımlamak için kullanılmaktadır.

tb_uart_top:

DUT'u, arayüzü ve çevreyi önekleyen, her şeyi birbirine bağlayan ve simülasyonu başlatan üst düzey testbench modülüdür olarak kullanılmaktadır. DUT'a reset sinyalleri buradan verildikten sonra uart testi buradan koşturulmaktadır. Aşağıdaki figürde UVM ortamının nasıl çalıştığı görülebilir:

Name	Type	Size	Value
<hr/>			
uvm_test_top	uart_test	-	0343
env	uart_env	-	0356
agent	uart_agent	-	0365
cov	uart_cov	-	0549
analysis_imp	uvm_analysis_imp	-	0558
driv	uart_driver	-	0511
rsp_port	uvm_analysis_port	-	0530
seq_item_port	uvm_seq_item_pull_port	-	0520
mon	uart_mon	-	0540
ap_port	uvm_analysis_port	-	0569
seqr	uvm_sequencer	-	0374
rsp_export	uvm_analysis_export	-	0383
seq_item_export	uvm_seq_item_pull_imp	-	0501
arbitration_queue	array	0	-
lock_queue	array	0	-
num_last_reqs	integral	32	'd1
num_last_rsp	integral	32	'd1

Denklem 2: Koşturulan UVM ortamı

REGRESYON TESTLERİ

Arayüzle daha kolay iletişim kurabilmek için driver sınıfının içinde yapılmaktadır. Her test hem UART'ın hem rx hem de tx fonksiyonellliğini test etmektedir. Kaç tane test sinyali gönderileceği tb_uart_top modülünden ayarlanabilmektedir. Veri göndermeyi test etmek için, cfg[0]'ı 1 yapma gibi gerekli başlama sinyalleri ve gönderilecek randomize edilmiş veri DUT'a gönderilir. Sonra veri gönderiminin tamamlandığını söyleyen sinyal beklenir ve bu sinyalin değerine göre testin geçilip geçilmediği raporlanır. Veri alımını test etmek için ise DUT'a rx portundan 0 gönderilerek veri alımı başlatılır sonra rx randomize edilerek DUT'a sinyal gönderilir. Start, data ve stop bitleri üzere her verinin gelmesi cpb saat sayısı kadar beklenir ve gönderilen rx verilerinin en son toplanan rx_data_o sinyali ile karşılaştırılması yapılarak testin geçip geçilmediğine bakılır. Özellikle veri gönderimi testleri olmak üzere, regresyon testlerinin daha modüler ve parametrize bir şekilde yapılması amaçlanmaktadır. Örneğin, cpb sinyalini de randomize etmek veya veri gönderiminde gönderilen her veriyi ayrı ayrı test etmek gibi. Aşağıdaki figürde testlerin bir kısmı gözlenebilir.

```

-----
[ ] Transaction No. = 0
[ ]   cfg_i = xx1,      tx_data_i = 88,      done_tx_o = 1
[ ] [TRANSACTION]::TX PASS
[ ]   Expected data = 24,      Obtained data = 24
[ ] [TRANSACTION]::RX PASS
[ ] -----
[ ] -----
[ ] Transaction No. = 1
[ ]   cfg_i = 1,      tx_data_i = 65,      done_tx_o = 1
[ ] [TRANSACTION]::TX PASS
[ ]   Expected data = 81,      Obtained data = 81
[ ] [TRANSACTION]::RX PASS
[ ] -----
[ ] -----
[ ] Transaction No. = 2
[ ]   cfg_i = 1,      tx_data_i = 32,      done_tx_o = 1
[ ] [TRANSACTION]::TX PASS

```

Şekil 8: Regresyon testleri

--- UVM Report Summary ---

```

** Report counts by severity
UVM_INFO :    75
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
** Report counts by id
[ ]    71
[RNTST]     1
[TEST_DONE]    1
[UVM/COMP/NAMECHECK]    1
[UVM/RELNOTES]   1

```

Şekil 9: UVM rapor özeti

FUNCTIONAL COVERAGE RAPORU

Functional coverage'in raporlanması için DUT'a giren ve çıkan port sinyalleri için ayrı ayrı binler oluşturulmuştur. Veri alımını ve veri gönderimini ayrı ayrı test etmek için ise bunların fonksiyonellliğini kapsayan cross binler oluşturulmuştur. Örneğin, tx_i, tx_data_o ve tx_done sinyallerinin birleştirilmesi gibi. Bu yöntemi izleyen coverage sonuçları aşağıda gözlemlenebilir. Henüz %100 coverage'a ulaşılmamıştır fakat daha kapsamlı testler ile bu sonuca ulaşılması hedeflenmektedir.

Instance : obj.cov_inst

Instance Info : obj.cov_inst

Score	Weight	Goal	At Least	Auto Bin Max	Print Missing	Is Instantized	Comment
82.5	1	100	1	64	0	1	

Instance Variable[s] Summary : obj.cov_inst

Cover Points	Category	Expected	Uncovered	Covered	Avg Percent
7	Variables	21	6	15	89.2857
3	Crosses	24	8	16	66.6667

Tablo 6: Genel Coverage

Instance Cover Point[s] Details : obj.cov_inst

Name ↴	Expected	Uncovered	Covered	Percent ↴	Goal	Weight	At Least	Auto Bin Max	Comment
RX	1	0	1	100	100	1	1	1	
TX_DIN	8	2	6	75	100	1	1	8	
TX_STAR	1	0	1	100	100	1	1	1	
TX	1	0	1	100	100	1	1	1	
RX_DOUT	8	4	4	50	100	1	1	8	
TX_DONE	1	0	1	100	100	1	1	1	
RX_DONE	1	0	1	100	100	1	1	1	

Tablo 7: Instance Coverage

Instance Cross[es] Details : obj.cov_inst

Name ↴	Expected	Uncovered	Covered	Percent ↴	Goal	Weight	At Least	Comment
RXxRX_D ...	8	4	4	50	100	1	1	
TXxTX_D ...	8	2	6	75	100	1	1	
TX_STAR ...	8	2	6	75	100	1	1	

Tablo 8: Cross Coverage

4.1.2. GPIO

in'e sürekli rastgele sayılar yazılıyor. out'a in'den okuduğu sayıyı bir arttırıp yazan bir program yazdık.

4.1.3. Timer

Timer'ı her 1 saniyede interrupt yapacak şekilde konfigüre ettik. Interrupt fonksiyonuna daha önceden tanımlamış olduğumuz değişkeni bir artıran bir bölüm yazdık. Bu değişkeni sürekli GPIO out'a yazdık. Özetlemek gerekirse, saniyede 1 kere artan sayaç yaptıktı diyebiliriz. Timer'ı diğer konfigürasyonlarla da denedik çalışıyordu. Simülasyon 1 saniye boyunca çalıştırınmak epey zaman aldı.

4.1.4. I2C Master

İnternetten açık-kaynak I2C Slave kodu bulunmuştur. 4 tane I2C Slave bağlanılmıştır. Yalnızca adresleri elle ayarlanmıştır. Geri kalan koda bilerek dokunulmamıştır ki olabildiğince uyumluluğu sağlamak için değişen kısım sadece bizim yazdığımız I2C Master olsun.

Birçok küçük boyutta testler denedik ama en kapsamlı şu testti:

Önce I2C Master'ı, adresini slave'lerden birinin adresiyle aynı olacak şekilde konfigüre ettik. Bir int tanımladık ve bu int'e rastgele bir sayı yazdık. Daha sonra program şöyle bir döngüye giriyor:

- I2C Master'a bu tanımladığımız int'i yazmasını söyleyen fonksiyonu çağır.
- I2C Master'a okumasını söyleyen fonksiyonu çağır. Okuduğu değeri 1 arttır ve bu int'e yaz.

Ayrıca repeated start'ı da denedik. Donanımsal olarak implement etmedik ama yazılımsal olarak art arda iki kere yazma fonksiyonunu çağırarak yapılabildiğini gördük.

4.1.5. QSPI Master

QSPI Master'a, internetten açık-kaynak olarak bulduğumuz S25FL128S modülünü bağladık. QSPI Master için özel bir test yapmadık açıkçası. Bootloader kodu düzgün çalışıyor mu diye baktık. Waveform'u inceledik. Her şey olması gerektiği gibi olana kadar hataları giderdik. S25FL128S komutlarının yaklaşık 3'te 1'ini test ettik diyebiliriz. Bunun içinde her türden yazma/okuma modu var.

4.2. İmplementasyon Testleri

GPIO, Timer ve I2C Master'ın implementasyon testlerini yaptık. UART, USB, JTAG ve QSPI'ın implementasyon testlerini DTR'den sonra bıraktık.

4.2.1. GPIO

Anahtarlardan alınan veriyi okuyan ve okuduğu bu veriyle LED'leri yazan bir program yazdık. İlk denemede başarılı oldu.

4.2.2. Timer

Timer'ı her 1 saniyede interrupt yapacak şekilde konfigüre ettik. Interrupt fonksiyonuna daha önceden tanımlamış olduğumuz değişkeni anahtarlardan okuduğu veri kadar artırmasını sağladık. Bu değişkeni sürekli LED'lere yazdık. Özetlemek gerekirse, LED'lerin gösterdiği sayı her saniyede bir anahtarların gösterdiği sayı kadar artıyor. Timer modülünde bazı hatalarla karşılaştık. Bu hataları giderdikten sonra başarılı bir şekilde bu testi tamamladık.

4.2.3. I2C Master

I2C Slave olarak Arduino UNO'yú kullandık. 12. ve 13. pini output olarak tanımladık. Wire kütüphanesini kullandık. Bir adres belirledik. Bir değişkende tanımladık. Yazma emri geldiğinde okuduğu değeri bu değişkene yazıyor. Okuma emri geldiğinde bu değişkenin değerini yazıyor.

Bu test yarı kaldi. İlk başta Arduino'yu doğrudan FPGA'e bağlamanın yeterli olacağını düşünmüştük ama sandığımız gibi olmadı. FPGA, Arduino'dan ack sinyalini bazen almıyor. Arduino da gönderilen veriyi yanlış kaydediyor. Bunun gibi anlam veremediğimiz birçok sorun çıktı. İlk başta sorunun yazılımsal olabileceğini ya da RTL kodunda bir sorun olabileceğini düşündük ama sorun donanımsal. Harici direnç gerekiyormuş. Osiloskop ile detaylı bir analizinin yapılması gerekiyor. Tahmin edebileceğiniz üzere ILA pek yeterli bir bilgi vermiyor.

5. TAKIM ORGANİZASYONU

5.1. Takım Organizasyonu

Cengiz Emre Dedeağacı: Takımın danışmanıdır. Özyegin Üniversitesi’nde Bilgisayar Mühendisliği bölümünde doktora yapmaktadır.

Kutay Bulun: Özyegin Üniversitesi’nde Elektrik-Elektronik Mühendisliği bölümünde lisans öğrencisidir.

Metin Arda Köker: Özyegin Üniversitesi’nde Bilgisayar Mühendisliği bölümünde yüksek lisans yapmaktadır.

Taha Gemici: Özyegin Üniversitesi’nde Elektrik-Elektronik Mühendisliği bölümünde lisans öğrencisidir.

Ahmet Utku Erşahin: Özyegin Üniversitesi’nde Elektrik-Elektronik Mühendisliği bölümünde yüksek lisans yapmaktadır.

5.2. Görev Dağılımı

Kutay Bulun, UART’ın UVM standardında test edilmesinden sorumludur.

Taha Gemici, sistem mimarisinden ve UART UVM dışındaki testlerden sorumludur.

Metin Arda Köker, çip tasarım akışından sorumludur.

Ahmet Utku Erşahin, çip tasarım akışından sorumludur.

6. İŞ PLANI ve RİSK PLANLAMASI

Açıkçası projenin büyük bir çoğunluğu bitti. Sırasıyla geriye kalan görevler:

1. USB RTL kodunun uyumlu hale getirilmesi
2. JTAG bağlantılarının yapılması
3. Geriye kalan testlerin yapılması
4. Sistemin tamamının GDSII çıktısının alınması

İlk 3 görevi Taha Gemici yapacaktır. 4. görevi Metin Arda Köker yapacaktır. 4. görev ilk 2 görev bittikten sonra yapılacaktır. Hata bulundukça düzeltilecektir.

7. KAYNAKÇA

- [1] Chipsalliance. "Chipsalliance/Synlig: Systemverilog Support for Yosys." GitHub, github.com/chipsalliance/synlig. Accessed 19 July 2024.
- [2] Zachjs. "Zachjs/SV2V: Systemverilog to Verilog Conversion." GitHub, github.com/zachjs/sv2v. Accessed 19 July 2024.
- [3] Admin. "UVM Testbench Top." ChipVerify, www.chipverify.com/uvm/uvm-testbench-top. Accessed 19 July 2024.