

**ÇİP TASARIM YARIŞMASI**

**2024**

**MİKRODENETLEYİCİ TASARIM KATEGORİSİ**

**DETAY TASARIM RAPORU**

**ArchOzU**

**……………………………….**

**2007098**

**………………………………**

İçindekiler

[1. SİSTEM TANIMI VE TEMEL TASARIM ÖZETİ 3](#_Toc171769706)

[2. PROJE DETAY TASARIMI 4](#_Toc171769707)

[2.1. Sistem Mimarisi 4](#_Toc171769708)

[2.2. Tasarım Detayı 5](#_Toc171769709)

[2.2.1. İşlemci ve Bus Yapısının Tasarımı 5](#_Toc171769710)

[2.2.2. Peripheral Tasarım Detayları 6](#_Toc171769711)

[2.2.2.1. Çevre Birimi Belleği 7](#_Toc171769712)

[2.2.2.2. UART 8](#_Toc171769713)

[2.2.2.3. I2C Master 9](#_Toc171769714)

[2.2.2.4. QSPI Master 11](#_Toc171769715)

[2.2.3. Yazılım Ortamı 14](#_Toc171769716)

[2.2.3.1. peripherals.h 14](#_Toc171769717)

[2.2.3.2. main.c 15](#_Toc171769718)

[2.2.3.3. linker\_script.ld 15](#_Toc171769719)

[2.2.3.4. mem\_file\_gen.sh 16](#_Toc171769720)

[2.2.3.5. hex2mem.py 17](#_Toc171769721)

[2.2.4. FPGA Prototipleme Detayları 17](#_Toc171769722)

[3. ÇİP TASARIM AKIŞI 19](#_Toc171769723)

[4. TEST 19](#_Toc171769724)

[4.1. Simülasyon Testleri 20](#_Toc171769725)

[4.1.1. UART [UVM] 20](#_Toc171769726)

[4.2. İmplementasyon Testleri 21](#_Toc171769727)

[5. TAKIM ORGANİZASYONU 22](#_Toc171769728)

[5.1. Takım Organizasyonu 22](#_Toc171769729)

[5.2. Görev Dağılımı 22](#_Toc171769730)

[6. İŞ PLANI ve RİSK PLANLAMASI 23](#_Toc171769731)

[7. KAYNAKÇA 24](#_Toc171769732)

# SİSTEM TANIMI VE TEMEL TASARIM ÖZETİ

{Bu kısımda sistemin basitçe genel tanımı ve proje kapsamında yürütülen faaliyetlerle ilgili olarak özet bilgiler sunulur. Örneğin RTL aşaması, simülasyon aşaması, çip fiziksel tasarım aşaması vs. tamamlanma oranları nelerdir?}

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

* UART [RTL] [UVM] [İT]
* I2C Master [RTL] [ST] [İT]
* QSPI Master [RTL] [ST] [İT]
* Timer [RTL] [ST] [İT]
* GPIO [RTL] [ST] [İT]
* JTAG [RTL] [ST] [İT]
* USB [RTL] [ST] [İT]

ST: Simülasyon Testi

İT: İmplementasyon Testi

Yeşil renk tamamlandığını, kırmızı renk başlanmadığını/yarım kaldığını göstermektedir.

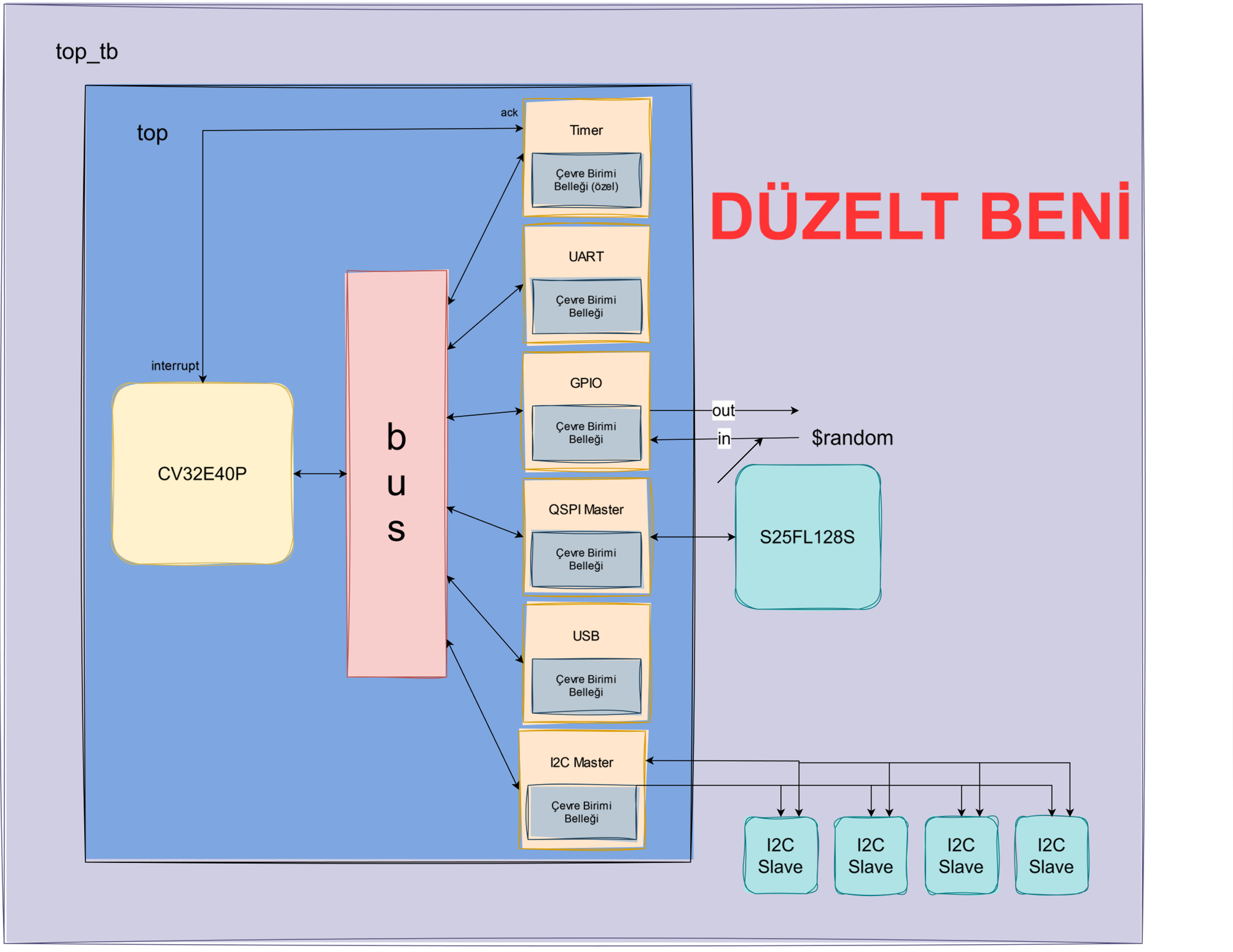
Bu çevre birimlerinin çekirdek ile bağlantısının kurulabilmesi için bus ve çevre birimi belleği modülleri kullanılmıştır. USB ve JTAG, DTR sonrasına bırakılmıştır. Sistemin şu anki halinin fiziksel tasarımı tamamlanmıştır.

# PROJE DETAY TASARIMI

## Sistem Mimarisi

{Bu kısımda projenin nihai tasarımının blok şeması verilir ve yarışma şartnamesi ile tasarlanan sistemin uyumluluklarına değinilir.

Projenin tamamlanan kısmının blok şeması aşağıdaki şekildedir:



Alt bloklardan ve görevlerinden bahsedilir.

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 kesme 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.

Bus yapısı,

Kendi yazdığımız bus yapısını kullandık. Az sayıda çevre birimi olduğundan bu bus yapısında okuma/yazma gecikmesi 1 cycle’dır.

boot sekansı,

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.

peripheralların bağlantı detayları, yazılım ile ilgili yapılan çalışmalar, linker script,

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

projenin FPGA ve ASIC implementasyon aşamalarındaki farklar vs}

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 kullanılıyoruz.

Ayrıntılı bilgilere Tasarım Detayı bölümünde ulaşabilirsiniz.

## Tasarım Detayı

### İşlemci ve Bus Yapısının Tasarımı

{CV32E40P işlemcisinin portları mikrodenetleyici içerisinde nasıl bağlanıyor?

Top modülde işlemci ve bus birbirlerine bağlanıyor.

İşlemciden bus’a:

Hangi bus yapısı seçildi, işlemci ve peripherallar bu bus’a nasıl bağlandırlar?

Önce APB + AHB kombinasyonunu deneyecektik lakin karmaşık geldiği ve slave’lerin her zaman yanıt vereceğini gördüğümüz için bundan vazgeçtik. 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 top modülden 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 gerektikçe kullanılmaktan çekinilmemiştir. İşlemci ile bus arasında veri belleği, komut belleği ve kesme bağlantıları bulunmaktadır.

Seçilen yöntem ve metotların diğerlerine göre artı eksileri? Karşılaşılan zorluklar ve çözümleri?}

Bus yapısını biz hazırladığımız için kontrol tamamen bizde, rahatça değişiklik yapabildik. 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.

### Peripheral Tasarım Detayları

{Şartnamede tanımlanan peripheralların tasarım detayları hakkında bilgi verilmesi.

Bizim yazdığımız modül(ler):

UART

I2C

QSPI  
Timer

GPIO

Çevre Birimi Belleği

Bus

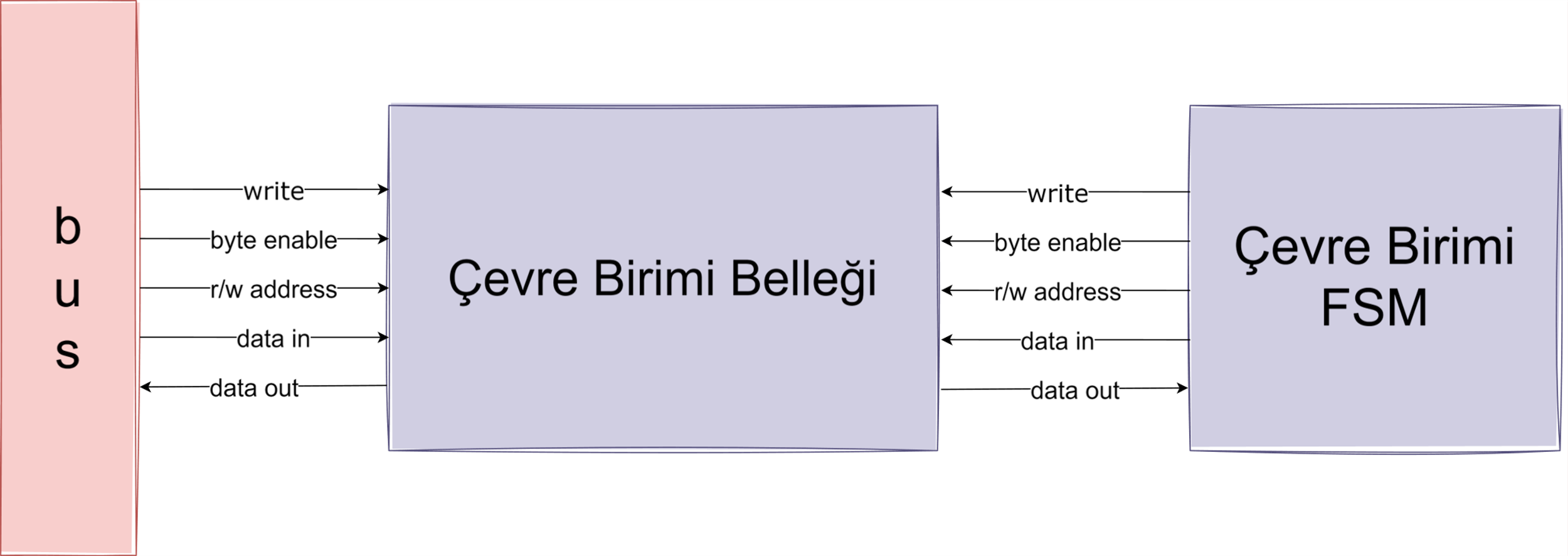
Hazır aldığımız/alacağımız modül(ler):

JTAG

USB

### Çevre Birimi Belleği

Bus’ı tasarlarken önce veri belleğini 8kB’tan daha büyük yapıp tüm çevre birimlerine erişim hakkı vermeyi 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 şartnamede belirtilen yazmaçları tutar. Hem çekirdek hem de çevre birimi yazma/okuma işlemleri yapabilir. Modülün blok şeması aşağıdaki gibidir:



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. Şöyle 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.

### UART

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.

Alıcı 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 alınsın. 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 yazmaçtaki değeri, asıl yazmaca (UART\_RDR) yaz.

Alıcı FSM’de direk asıl yazmaca yazmak yerine geçici bir yazmaca yazıp. Daha sonrasında bu geçici yazmacı asıl yazmaca yazmamızın bir sebebi var: Varsayalım 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 sayılara denk gelebilir. Her ne kadar peripherals.h header’ında buna karşın önlem alınmış olsa da kullanılması zorunlu olmadığı için donanım seviyesinde bu önlem alınmıştır.

### I2C Master

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’lık 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 yapılabilir). 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.

### QSPI Master

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 yazılabilsin.
2. CMD: QSPI\_STA yazmacına meşgul olduğunu belirten biti yaz. X0 modunda isek 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: Sayaç sıfırlanınca IDLE state’ine dön.
   1. Okuma modu:
      1. x1: 4 bitlik io bağlantısının 2. telinden okuduğun veriyi sayacı göz önünde bulundurarak ilgili bölüme yaz. Sayacı 1 azalt.
      2. x2: 4 bitlik io bağlantısının ilk 2 telinden okuduğun veriyi sayacı göz önünde bulundurarak ilgili bölüme yaz. Sayacı 2 azalt.
      3. x4: 4 bitlik io bağlantısının tüm tellerinden okuduğun veriyi sayacı göz önünde bulundurarak ilgili bölüme yaz. Sayacı 4 azalt.
   2. Yazma modu:
      1. x1: 4 bitlik io bağlantısının 1. teline QSPI\_DRx yazmacından okuduğun veriyi sayacı göz önünde bulundurarak yaz. Sayacı 1 azalt.
      2. x2: 4 bitlik io bağlantısının ilk 2 teline QSPI\_DRx yazmacından okuduğun veriyi sayacı göz önünde bulundurarak yaz. Sayacı 2 azalt.
      3. 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’ın 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 içerisine komutları gömmediğimiz için yalnızca şartnamede belirtilen dasdasdadadas[BENİ DÜZELT] 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 içeride yazan sayı kadar cycle diye baz aldık ama sonra fark ettik ki bu mümkün değil. Çünkü 5 bitlik yazılabilecek en yüksek sayı 31 ve dasdasdadadas[BENİ DÜZELT] 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:

A screenshot of a computer screen

Description automatically generated

### Yazılım Ortamı

C driver’ları (en azından header dosyaları) hazırlandı mı?

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

### 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 (baud rate, stop bit)
  + Yaz (veri)
  + Oku
* I2C:
  + Konfigüre et (adres)
  + Yaz (veri, byte sayısı)
  + Oku (byte sayısı)
* 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 (prescaler değeri, auto reload değeri, mod)
  + Sayacı oku
  + Event’i oku
  + Sayacı sıfırla
  + Event’i sıfırla
  + Çalıştır
  + Durdur
* GPIO:
  + Yaz (veri)
  + Oku
* Veri Belleği
  + Yaz (adres, veri)

USB’nin fonksiyonları henüz tasarlanmadı.

### main.c

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

Kesme yapılmak isteniyorsa kesme fonksiyonunun adı interrupt olmak zorundadır. Yazılımcının yükünü hafifletmek için kesme fonksiyonuna interrupt(“machine”) attribute’u verilmesini tavsiye ediyoruz. Çünkü kesme fonksiyonlarında sözde ret komutu ra yazmacındaki adrese gitmenizi sağlar. Halbuki kesme 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) kesme 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.

### linker\_script.ld

Linker script dosyasıdır. Oldukça küçük bir dosyadır. Linker’a nasıl davranması gerektiğini söyler. GCC’nin linker’ı yalnızca Von Neumann mimarisini destekliyor. Dolayısıyla burada ufak bir hileye başvurmamız gerekiyor. MCU ve Linker gözünden iki farklı bakış açımız var:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Harvard (MCU) | | Von Neumann (Linker) |
| Bus | Komut Belleği |
| 0x0000 | Veri Belleği | bootloader | Veri Belleği |
| 0x2000 | *kullanılmıyor* | 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 |

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 şekle çevirdik ve bu şekilde kaydettik. Veri belleğinin başlangıç adresinin 0x0000 olduğunu görüyorsunuz. Bitişi 0x2000’e denk geliyor. GCC verileri sp yazmacını eksilterek yazıyor. Bootloaderın komut belleğine dallanmadan önce sp yazmacına 0x2000 yazmasının sebebi işte bu. Kesme fonksiyonunun adresini sabit 0x2200 olarak belirledik. Eğer main fonksiyonu 0x2000 ile 0x2200 arasına sığmazsa GCC hata veriyor. Bu yüzden main fonksiyonunun gerekirse parçalanmasını tavsiye ederiz. Diğer fonksiyonlar kesme 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 gerekir.

### mem\_file\_gen.sh

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

### 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 gerekilen dosyadır. İçerisinde .text bölümleri ve .data bölümleri 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.

### FPGA Prototipleme Detayları

{FPGA prototipleme detayları. FPGA’da işlemci üzerinde yazılım koşturuldu mu?

Elimizde S25FL128S model Flash bellek olmadığı için FPGA’e gömeriz diye düşündük. Fakat Vivado, boyutu çok büyük olduğu için sentezde hata verdi. Bu yüzden “defines.vh” headerına “`define NO\_FLASH” diye bir satır ekledik.

Bu satır FPGA implementasyonunda Flash belleğe önceden yazılmış olacağını varsaydığımız 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:

A screenshot of a computer code

Description automatically generated

Yazılımlarla peripherallar test edildi mi?

Timer’ı konfigüre eden ve sonrasında periyodik olarak GPIO’nun giriş portlarından veri okuyup bunu işleyerek (toplama, çıkarma, 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 kesme isteklerine yanıt verip vermediğini, GPIO modülünü, linker scripti, program header’ı ve yazdığımız C kodunu doğrulamış olduk.

I2C modülünü Arduino UNO’yu slave olarak bağladıktan 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. İmplementasyon testlerini 50MHz saat frekansı ile yaptık.

C kodları için github hesabınızdaki ilgili dosya pathi link verilebilir ama FPGA üzerinde veya simülasyon ortamındaki çalışmalar DTR’de detaylandırılmalıdır. Alınan sonuçlar görsellerle belirtilmelidir. Karşılaşılan zorluklar ve çözümleri?}

# ÇİP TASARIM AKIŞI

{Bütün bir sistemin olmasa da peripheralların biri veya birkaçı ya da işlemci çekirdeği için çip fiziksel tasarım akışının tamamlanması beklenmektedir. Mümkünse tasarımın hepsi, tasarımın hepsi tamamlanmadıysa da tamamlanmış alt modüller üzerinde synthesis, placement ve routing gibi aşamalar sonrasında oluşan zamanlama analizi (timing analysis), kaynak kullanım (utilization) gibi dosyaların çıktılarından yararlanıp yorumlarda bulunulmalıdır. GDSII layout resimleri eklenmelidir. Güç tüketimi, performans ve alan tüketimini iyileştirmek adına akışta yapılan özelleştirmelere değinilmelidir. Tasarım sürecinde gerek yazılımsal olarak gerek tasarımsal olarak karşılaşılan sorunlara ve bu sorunların nasıl çözüldüğüne değinilmelidir (Örneğin P & R programı tasarımı route edemediği şeklinde bir hata vermiştir, yarışmacılar da bu durumu bir yöntemle çözmüştür veya akış sırasında bir hata alınmıştır, bunu çözmek için başka bir commit’teki versiyona geçilmiştir şeklinde). Bu çözümler esnasında kullanılan teorik-genelgeçer bilgilere yer verilmelidir (Örneğin bu kaynaktan edinilen bilgiye göre utilization değeri bu değerin altında tutulmuştur). Açık kaynak programlar kullanıldıysa bahsedilen genelgeçer bilgilerle elde edilen sonuçlar uyumsuzluklar görüldüyse bunların sebebi açıklanmalıdır. Çip tasarım akışı boyunca faydalanılan topluluklar, Slack kanalları gibi yerler varsa bunların hangilerinden ne yönde yararlanıldığından bahsedilmelidir. Çip akışında geçirilen aşamaların hangilerinin daha kolay hangilerinin daha zor bulunduğuna, hangilerinde ne kadar zaman geçirildiğine kısaca değinilmelidir.}

# TEST

# Simülasyon Testleri

# UART [UVM]

UART protokolünün doğrulanması için UVM metodu izlenmiştir. Aşağıdaki figürde UVM ortamının yukarıdan görünümü bulunabilir. Oluşturulan test ortamı bu hiyerarşik yapıyı izlemektedir ve bu ortam uart\_core modülünün doğrulanması için oluşturulmuştur. Figürden sonra oluşturulan UVM ortamının sınıf ve bileşen bazında detaylı açıklamasını bulabilirsiniz.

A screenshot of a computer

Description automatically generated

Şekil 1: UART UVM İnfografiği

**uart\_core (DUT):**

Doğrulama altındaki gerçek UART mantığını içerir.

**uart\_interface:**

uart\_core modülü için saat, reset ve veri hatlarını içeren iletişim sinyallerini tanımlar. Doğru senkronizasyon için sinyaller burada ‘’clocking’’ yapısı kullanarak sistem saatine göre senkronize edilmiştir.

**uart\_sequence\_item:**

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 fonksiyonelliğini test etmek için reset, 1 stoplı rx testi, tamamen rastgele gibi farklı türde sekanslar oluşturulmuştur ve hepsi sırayla DUT’u uyarmak için kullanılmaktadır.

**uart\_sequencer:**

Sıra öğelerinin sürücüye gönderilme sırasını ve zamanlamasını kontrol etmektedir.

**uart\_driver:**

Sıra öğeleri işlemlerini DUT arayüzünde pin seviyesi aktivitelerine dönüştürmektedir.

**uart\_monitor:**

Pin seviyesi aktiviteleri gözlemler ve analiz için bunları tekrar sıra öğelerine dönüştürür. Oluşturduğu sıra öğelerini aynı anda regresyon testlerini yapmak için scoreboard sınıfına hem de ‘’functional coverage’’ raporu oluşturmak için subscriber sınıfına gönderir.

**uart\_scoreboard:**

Sekanslardan beklenen sonuçları monitor tarafından gözlemlenen gerçek sonuçlarla karşılaştırır. Gelen sıra öğelerini kaybetmemek için içinde tlm\_fifo kullanılmıştır. Uart’ın baud rate’ine göre veriyi doğru bir şekilde gönderip göndermediğini ve alıp almadığını kontrol edip raporlamaktadır.

**uart\_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. Monitor ve scoreboard bağlantısı burada yapılmaktadır.

**uart\_test:**

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

**tb\_top:**

DUT'u, arayüzü ve çevreyi örnekleyen, her şeyi birbirine bağlayan ve simülasyonu başlatan üst düzey testbench modülüdü olarak kullanılmaktadır.

# İmplementasyon Testleri

Simülasyon Testleri

Simülasyon Testleri

{Bu kısımda tasarımın RTL seviyesinde fonksiyonel simülasyonlar, CV32E40P işlemcinin doğrulanması, UART peripheral için UVM/SV ile yapılan doğrulama çalışmaları ve static timing analysis ile max frekans hesaplamaları detaylandırılmalıdır. Tüm tasarım tamamlanmasa bile hangi alt modüllerde simülasyonların nasıl gerçekleştirildiği, verification ortamı (FPGA ve çip tasarım doğrulaması için hangi verification araçları kullanıldıysa) ve testbench kodları ile bilgiler detaylıca verilmelidir.}

# TAKIM ORGANİZASYONU

## Takım Organizasyonu

{Bu kısımda takım üyeleri ve varsa danışman hakkında bilgi verilir. (İsim, soyisim, okul, bölüm, sınıf)}

Cengiz Emre Dedeağaç: Takımın danışmanıdır. Özyeğin Üniversitesi’nde Bilgisayar Mühendisliği bölümünde doktora yapmaktadır.

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

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

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

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

## Görev Dağılımı

{Bu kısımda görev dağılımı ve ekip organizasyonu hakkında bilgi verilir. Kim hangi kısımlardan sorumlu?}

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.

# İŞ PLANI ve RİSK PLANLAMASI

{Bu kısımda projenin FPGA ve çip aşamaları için tasarım, doğrulama, sentez, fiziksel gerçekleme ve test süreçlerini içeren bir zaman planlaması ve risk planlaması yapılır. Zaman akış çizelgesi üzerinde iş paketlerinin ne kadarının tamamlandığı ne kadarının henüz tamamlanmadığı, takvimde gecikme olup olmadığı açık bir şekilde gösterilmelidir.}

# KAYNAKÇA

{Bu bölümde raporda kullanılan kaynaklar yer almalıdır. Kaynaklar rapor içerisinde refere edilmelidir.}

ChipVerify. (2024). [İnfografik]. https://www.chipverify.com/images/uvm/tb\_top.svg

|  |
| --- |
| **RAPOR TASLAKLARI İLE İLGİLİ NOT:** |
| **Bu belge Çip Tasarım Yarışması Mikrodenetleyici Kategorisi Detay Tasarım Raporu (DTR) şablonu olarak hazırlanmıştır.**  **Rapor, Çip Tasarım Yarışması Şartnamesi Bölüm 3.2.2’de belirtilen kurallara uygun olarak hazırlanmalıdır.**  **Rapor dili anlaşılırlığına, yazım düzenine, rapor içeriğine ve rapor bütünlüğüne dikkat edilmelidir.**  **Grafikler, resimler, devre şemaları vb. okunaklı olmalıdır. Kullanılan resimler mümkün olduğunca yüksek kaliteli olarak eklenmelidir.**  **Raporlar, A4 formatında, 11 punto, Calibri fontunda, satır aralığı 1.15 ve alt-üst ve yan kenarları 2.5 cm olacak şekilde hazırlanmalıdır. Detay tasarım raporu en fazla 30 sayfa olmalıdır (kapak sayfası, resimler, tablolar, referanslar dâhil).**  **Toplam sayfa sayısı 30’u geçmemelidir. Sayfa sınırı aşıldığı takdirde ilk 30 sayfa dikkate alınacaktır.** |