

TEKNOFEST

HAVACILIK, UZAY VE TEKNOLOJİ FESTİVALİ

ÇİP TASARIM YARIŞMASI

SAYISAL TASARIM KATEGORİSİ

DETAY TASARIM RAPORU ŞABLONU

TAKIM ADI

MYST-IC

PROJE ADI

RISC-V İŞLEMCİLİ ÇİP TASARIMI

BAŞVURU ID

463899

İçindekiler

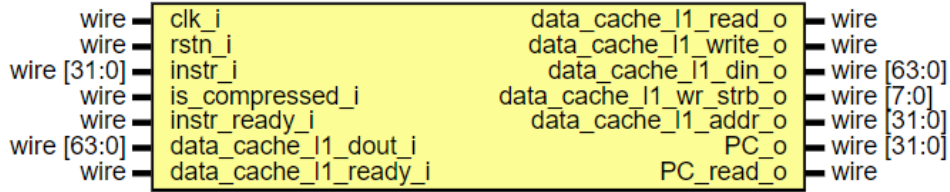
| | |
|---------------------------------------|----|
| 1. TEMEL TASARIM ÖZETİ | 3 |
| 2. PROJE MEVCUT DURUM DEĞERLENDİRMESİ | 3 |
| 3. PROJE DETAY TASARIMI | 3 |
| 3.1. Sistem Mimarisi | 3 |
| 3.1.1. Çekirdek tasarımı | 4 |
| 3.1.2. Bellek Tasarımları | 9 |
| 3.1.3. Çevre Birimleri Tasarımları | 13 |
| 4. ÇİP TASARIM AKIŞI | 17 |
| 5. TEST | 17 |
| 6. TAKIM ORGANİZASYONU | 18 |
| 6.1. Takım Organizasyonu | 18 |
| 6.2. Görev Dağılımı | 18 |
| 7. İŞ PLANI ve RİSK PLANLAMASI | 19 |
| 8. KAYNAKÇA | 19 |



Çekirdek mystic_riscv64 olarak adlandırılmıştır. Çekirdekten çıkan adres master_cntrl kısmında bulunan mystic_address_controller modülüne girerek adresin UART, SPI arayüzlerine veya ana hafızaya ait olup olmadığına karar verilmektedir. Ana hafızaya yazılacak veriler sbbox transform yöntemi kullanılarak karıştırılmaktadır. Ana hafızadan veri çekildiği zaman bu veri ters sbbox dönüşümü kullanılarak çözülmektedir.

3.1.1. Çekirdek tasarımı

Nihai çekirdek çok çevrimli (multicycle) bir yapıya sahiptir. Şekil 2’de nihai çekirdeğin girişleri ve çıkışları görülmektedir.

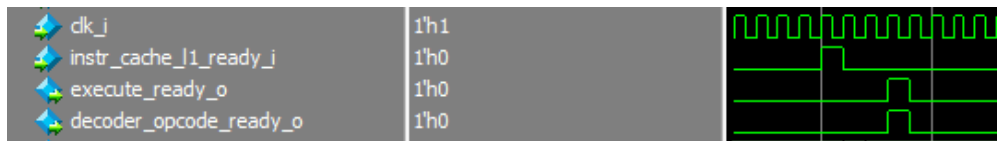


Şekil 2. Mystic_RV64 giriş çıkış portları

İşlemci basit bir arayüze sahiptir. Tasarlanan işlemcinin sıkıştırılmış buyrukları desteklemesi gerekmektedir. Buyruğun sıkıştırılmış veya normal olduğuna çekirdek dışında karar verilmektedir. is_compressed_i girişi gelen buyruğun sıkıştırılmış olup olmadığının göstergesidir. Gelen buyruğun sıkıştırılmış olup olmadığı buyruk seviye 1 önbelleğinde buyruk okunurken belirlenmektedir. Böylece buyruğun sıkıştırılmış olup olmadığı işlemci çekirdeği dışında veri hafızadan okunurken belirlenmektedir. Bu hem işlemcinin yükünü azaltmakta hem de okunduğu anda kontrol yapıldığı için işlemci içerisinde bu işlem için ayrı bir süre harcanmamaktadır.

Buyruk okuma PC_read_o çıkışı 1 olduğu zaman PC_o adresinden gerçekleştirilmektedir. Gelen buyruk eğer sıkıştırılmış ise o zaman instr_ready_i girişi ile beraber is_compressed_i girişi de 1 olmaktadır. data_cache_l1_dout_i arayüzlerden veya seviye 1 önbellek bloğundan okunan değeri ifade etmektedir. Seviye 1 önbellekten veya arayüzlerden veri okumak için data_cache_l1_read_o, veri yazmak için ise data_cache_l1_write_o sinyalleri kullanılmaktadır. İşlemci 64 bit olarak tasarlandığından yazılacak ve okunacak sinyaller 64 bit uzunluğundadır. İşlemci instr_ready_i 1 olduğunda gelen buyruğu işlemeye başlamaktadır ve işlem tamamlandığında PC_o 1 olmaktadır, böylece işlemci yeni buyruk talep etmektedir. Buyruğun işleme süresi buyruktan buyruğa farklılık göstermektedir. Çarpma ve bölme buyruklarında bu süre uzunken diğer buyruklarda daha kısa sürmektedir.

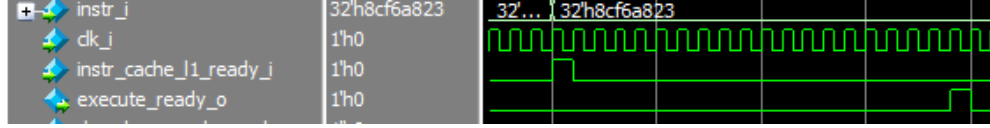
R(egister) tipindeki örnek bir buyruğun önbellekten okunduktan sonra işlem tamamlanincaya kadar geçen süresi benzetim ortamında gösterilmektedir. Bu süre 3 saat döngüsü kadardır. Gelen buyruğun çözülmesi 1 saat döngüsü, talep edilen işlemin gerçekleştirilmesi 1 saat döngüsü ve register file denilen işlemci içerisinde bulunan çok hızlı ve geçici hafızaya yazılımı 1 saat döngüsü sürmektedir.



Şekil 3. Decode, execute, write back

Bu süre LW(Load word), LH(Load halfword), LB(Load byte), LD(Load doubleword), SB(Store byte), SH(Store halfword), SW(Store word), SD(Store doubleword) gibi önbellek veya arayüz

birimlerine erişerek işlem yapmamızı gerektiren buyruklarda daha uzun sürmektedir. Bu süre şekil 4'te görüldüğü üzere 19 saat döngüdür. Direct-mapped önbellek yönetiminde ana hafızaya yazılacak veri önce seviye 1 önbelleğe, daha sonra seviye 2 önbelleğe ve en son ana hafızaya yazılmaktadır. Önbellek başlığında bu işlem ayrıntılı açıklanacaktır. Burada bahsedilmesinin sebebi S(tore) tipindeki buyruklarda buyruk işlem süresinin neden bu kadar uzun sürdüğünü açıklamaktır.

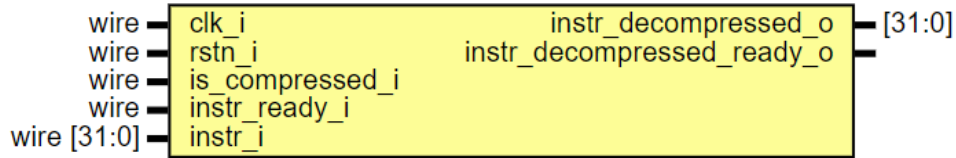


Şekil 4. S tip buyruk gecikmesi

Buyruk önbelleğinden işlemciye buyruk ilk olarak mystic_decompressed modülünden geçerek normal 32 bitlik buyruğa dönüştürülmektedir.

Decompression:

Mystic_decompressed modülünün giriş çıkışları şekil 5'te verilmektedir. Okunan buyruğun sıkıştırılıp olup olmadığına en anlamsız 2 bitine bakılarak karar verilmektedir. Aşağıda örnek buyruklar verilmiştir. En anlamsız bitin 11b olduğu durumda bu buyruğun 32 bitlik normal bir buyruk olduğu anlaşılmaktadır. Eğer en anlamsız 2 bit 11b'den farklı ise o zaman bu buyruk 16 bitlik sıkıştırılmış bir buyruktur ve 32 bitlik karşılığına dönüştürülmesi gerekmektedir. Buyruğun sıkıştırılmış olup olmadığına işlemci dışarısında karar verildiğinden seviye 1 buyruk önbelleği anlatılırken bu konu daha detaylı olarak anlatılacaktır.



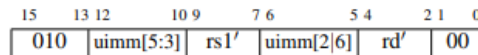
Şekil 5. Mystic_decompressed modül giriş çıkışları

Yukarıdaki şekilde verilen modülde instr_decompressed_o çıkışı is_compressed_i girişinin 0 olduğu durumda instr_i girişine bağlanmaktadır. 1 olduğu durumda ise yapılan örnek bir işlem aşağıda verilmiştir.

Örneğin c.lw buyruğunun geldiği durumda is_compressed_i girişi 1 olmaktadır. 16 bitlik buyrukların 32 bitlik buyruklara dönüşümünün nasıl yapılacağı David Patterson ve Andrew Waterman yazarlarının yazmış olduğu The RISC-V Reader: An Open Atlas kitabında aşağıda verilmiştir [1].

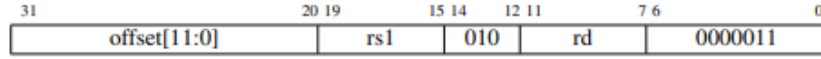
Aşağıda sıkıştırılmış olarak gelen bir buyruğun 32 bitlik normal bir buyruğa nasıl dönüştürüleceği gösterilmiştir. Öncelikle bu buyruğun hangi işlemi yapmak istediğine karar verilir. Her sıkıştırılmış buyruğun 32 bitlik normal buyruk karşılığı olmaktadır. c.lw buyruğu lw buyruğuna karşılık gelmektedir. Dönüşüm Şekil 6'da verilen formüller kullanılarak yapılır.

c.lw rd', uimm(rs1') $x[8+rd'] = sext(M[x[8+rs1'] + uimm][31:0])$
 Load Word. RV32IC and RV64IC.
 Expands to **lw** rd, uimm(rs1), where rd=8+rd' and rs1=8+rs1'.



Şekil 6. c.lw sıkıştırılmış buyruk

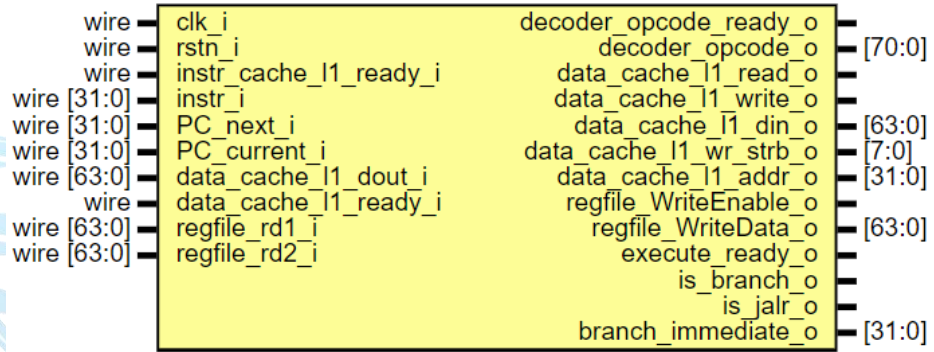
lw rd, offset(rs1) $x[rd] = \text{sext}(M[x[rs1] + \text{sext}(\text{offset})])[31:0]$
Load Word. I-type, RV32I and RV64I.
 Loads four bytes from memory at address $x[rs1] + \text{sign-extend}(\text{offset})$ and writes them to $x[rd]$. For RV64I, the result is sign-extended.
Compressed forms: c.lwsp rd, offset; c.lw rd, offset(rs1)



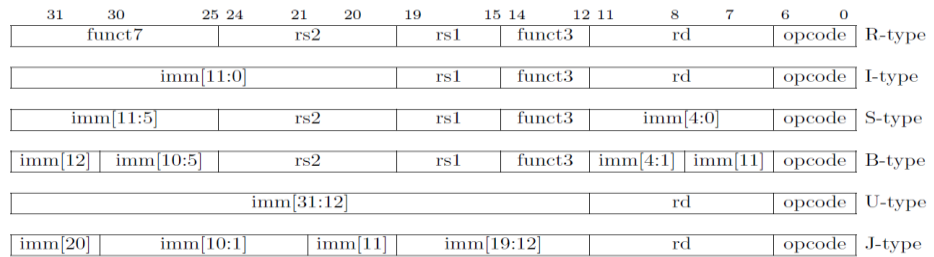
Şekil 7. lw normal(32 bit) buyruk

Decode:

Buyruk mystic_compressed_decoder modülünden geçtikten sonra çıkışa 32 bit olarak gitmektedir. Bu 32 bitlik buyruk mystic_main_decoder modülüne girerek hangi işlemlerin yapılacağına karar verilecektir. Mytsic_main_decoder modülünün içerisinde şartnamade belirtilen tüm buyruklar bulunmaktadır. Bu modülün giriş ve çıkışları aşağıdaki şekilde görülmektedir.



Buyruk geldiği zaman instr_cache_l1_ready_i girişi 1 olmaktadır ve buyruk geldiği anlaşılmaktadır. Buyruk her zaman 32 bitlik olarak instr_i içerisinde bulunmaktadır. Verinin gelmesiyle çözüm işlemi yapılmakta ve buyruk işletilme işlemine başlanmaktadır. (R)egister, (I)mmediate, (S)tore, (B)ranch, Long Immediates(U), (J)ump olmak üzere toplamda 6 farklı buyruk tipi bulunmaktadır.



R tipindeki buyruk geldiğinde register file içerisinde rs2 ve rs1 adresindeki veriler ile işlem yapıp yine register file içerisinde bulunan rd adresine yazım gerçekleştirilmektedir. I tipindeki buyruğun özelliği R tipinden farklı olarak işlem yapılacak rd2 ve rd1 adreslerindeki bilgilerden yalnızca rs1 adresindeki okunmasıdır. Rs2 adresindeki değer yerine buyruk içerisinde bulunan immediate değeri kullanılmaktadır. Register file içerisinde yazım işlemi R tipi ile benzerlik göstermektedir. S tipindeki buyruk hafızaya veya arayüz birimlerine veri yazmak için kullanılmaktadır. Rs1 adresindeki değer hafıza adresi, immediate hafıza adresinden ne kadar

uzakta olacağını belirlemektedir. Rs2 ise yazılacak olan verinin register file içerisinde hangi adreste olacağı bilgisini taşımaktadır. B tipindeki buyruklar tipik olarak if-else, while, for döngülerü için kullanılmaktadır. U tipindeki buyruk adından da anlaşılacağı üzere büyük değerlerde bir immediate oluşturmak için kullanılmaktadır. I tipindeki buyruk ile en çok 12 bitlik bir immediate oluşturmak mümkün iken U tipindeki ile 32 bitlik immediate oluşturmak mümkündür. Oluşturulan değer register file içerisindeki rd adresine yazılmaktadır. J tipindeki buyruk ile hafızada herhangi bir yere atlamak mümkün olmaktadır [2].

Farklı buyruk tiplerinde çarpma, bölme, çıkarma, kaydırma, jump, branch gibi işlemleri yapabileceğimiz bir çok buyruk işlemci tarafından desteklenmektedir. Bu buyruklar şartnamede belirtilen buyrukların tümünü kapsamaktadır.

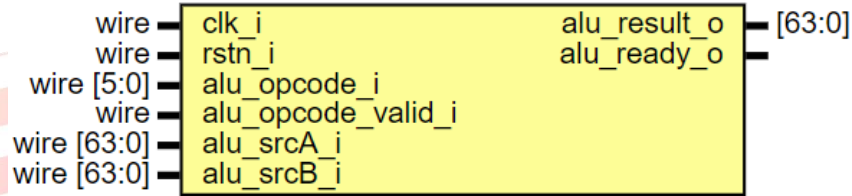
Tasarlanan işlemi bazı özel buyrukları da desteklemektedir. Bu buyruklar hmdst, pkg, rsrs, sladd, cntz ve cntp buyruklarıdır. Bu buyrukların özelliklerine şartnamede verilmiştir.

Execute (İşletim):

Bu adımda çözülen buyrukların amaçladığı işlemler yapılmaktadır. Örneğin buyruk 2 register adresinden okunan değerleri toplayıp diğer register adresine yazma işlemini gerçekleştirmeyi amaçlıyor ise o zaman bu işlem yapılmaktadır. Bu başlıkta desteklenen buyruklardan çarpma, bölme, toplama, çıkarma, kaydırma gibi işlemlerinin nasıl yapıldığı anlatılacaktır. Çarpma ve bölme için ayrı birimler oluşturulmuştur. Bunlar dışındaki aritmetik ve mantıksal işlemleri için ALU tasarlanmıştır.

ALU:

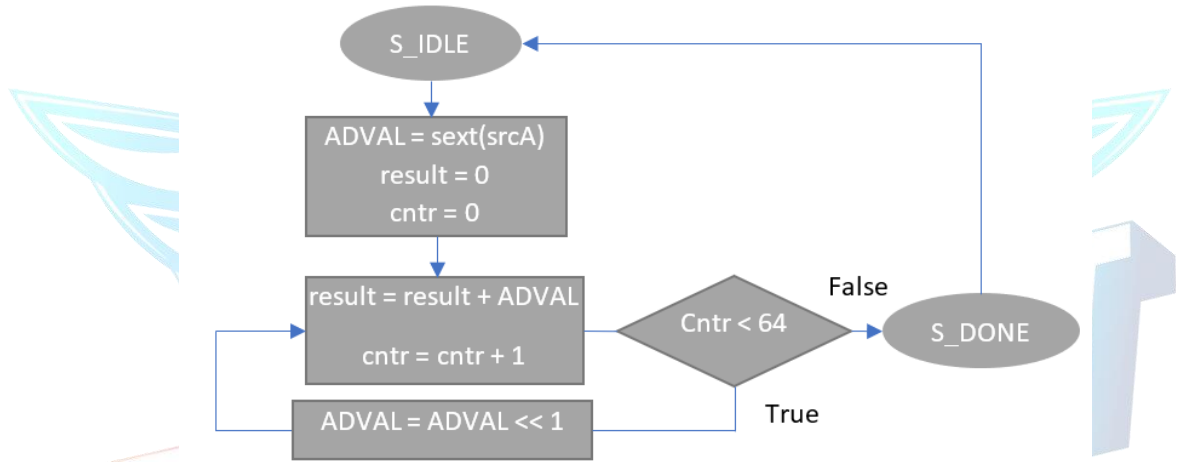
Arithmetic Logic Unit(ALU) içerisinde toplama, AND, OR, SLL, SRA, SRL, XOR, SUB işlemleri olmak üzere toplamda 8 farklı işlem gerçekleştirilebilmektedir. Hangi işlem yapılacağı opcode seçilerek belirlenmektedir. İşleme alu_opcode_valid_i sinyali 1 olduğunda başlanmaktadır. Giriş çıkış sinyalleri aşağıda verilmiştir.



Bu birim içerisindeki toplama işlemini yapmak için şartnamede istendiği üzere toplama sembolü kullanılmamıştır. Toplama işlemini yapan ayrı bir birim eklenmiştir. Toplama işlemi carry-lookahead-adder algoritması kullanılarak gerçekleştirilmiştir. Carry lookahead adder algoritmasının kullanılmasının sebebi oldukça bilinen bir algoritma oluşu ve açık kaynak çok sayıda kod örneğinin bulunmasıdır [3]. Carry lookahead adder toplama yöntemi kombinasyonel olarak gerçekleştirilmektedir. ALU birimi içerisinde talep edilen işlem tamamlandığında alu_ready_o 1 olmaktadır ve işlemci sonraki adıma geçmektedir. Toplama işleminde farklı bir algoritma kullanıldığı ve kodun açık kaynaklı bir siteden alındığı için işlemcide herhangi bir bug olmağını garantilemek için doğrulamasının yapılması gerekmektedir. Doğrulama işlemi kendi kendini kontrol eden testbench yazılarak 2**30 farklı kombinasyon için yapılmıştır. Tüm bu durumlar için verilog dilinde gömülü olan toplama işlemi ile carry lookahead adder işleminin sonuçları karşılaştırılmıştır. Böylece açık kaynaklı olarak alınan toplama işlemini gerçekleştiren modülün doğru çalıştığından emin olunmuştur.

Çarpma:

Tasarlanan çarpıcı devresi sıralı bir devre olup aritmetik ve mantıksal işlemlerden farklı olarak birden fazla saat vuruşunda gerçekleşmektedir. İşlemcinin çarpma işlemini gerçekleştiren kullanılan sayılar 64 bitlik olduğundan çarpma işlemi 64 saat vuruşunda gerçekleştirilmektedir. Kullandığımız algoritma tek toplayıcı devresi kullanılmıştır. Toplamda 64 toplama işlemi gerçekleştirilmektedir ancak bu işlem sıralı yapıldığından aynı donanımı kullanarak toplama işlemi sırasıyla gerçekleştirilmekte ve kaynak tasarrufu sağlanmaktadır. Tek toplayıcı devresi kullanılmasıdaki amaç daha az kaynak kullanımıdır. 2 farklı toplama devresi kullanılarak çarpma işlemi 32 saat döngüsünde gerçekleştirilmektedir. Toplayıcı devresi arttıkça ve toplama işlemi kombinasyonel gerçekleştirilirse bu süre kısalmaktadır ancak gecikmelerden dolayı işlemcinin maksimum frekansı düşmektedir. İdeal olanı tasarlanan işlemcinin çıkarılmak istenen saat frekansına göre veya kaynak kullanımının ne kadar olması gerektiğine göre değişiklik göstermektedir. Tasarlanan sistemde amaç en yüksek saat frekansına en az kaynak kullanarak çıkabilmektir. Bu amaçla tasarlanan çarpıcı devresi işlemi 64 saat vuruşunda gerçekleştirecek ve bunu tek toplama devresi kullanacak şekilde yapılmıştır. Çarpma işleminin alış tasarımı aşağıda verilmiştir.



Çarpılacak olan 2 veriden birinin işareti genişletilerek ADVAL değerine atanmaktadır. Sonrasında çarpılacak olan değer $cntr$ adresindeki biti 1 ise sonuca ADVAL eklenmektedir. Blok tasarımı yer kazanmak amacı ile bu koşul gösterilmemiştir. $srcB[cntr]$ 0 olduğu durumda ise ADVAL değeri eklenmemekte yalnızca $cntr$ değeri artırılmaktadır. Bu işlem 64 defa tekrar edildiğinde ise çarpma işlemi gerçekleşmiş olmaktadır. Çarpma birimi işaretli ve işaretli sayıların çarpımını desteklemektedir. Sayı eğer işaretli değilse o zaman işaret genişletme yerine üst 64 bit 0 ile doldurulmaktadır.

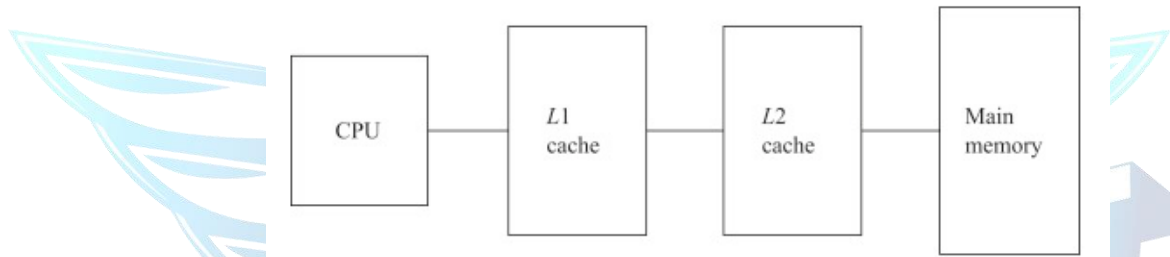
Bölme:

Bölme işlemi için çarpma işleminde olduğu gibi az kaynak kullanımı amaçlanmıştır. İşlemin kaç saat vuruşu sürece çok önemsenmemiştir. Tasarladığımız işlemcinin olabildiğinde az kaynak kullanması bir çok durumda öncelik olarak değerlendirilmiştir. Tasarlanan işlemcideki bölme biriminde non-restoring bölme algoritması kullanılmıştır [4]. Bu bölme işlemi bazı durumlarda sonucu çok fazla saat vuruşu sonra vermektedir. Bu yönden işlemin ne zaman biteceği tam olarak bilinmemektedir ve bu da kullanılan algoritmanın dezavantajı olarak görülebilir. Non-restoring bölme algoritması aşağıdaki gibi çalışmaktadır.

Bölme algoritmasında kalan bölenden küçük oluncaya kadar devam etmektedir. Bölen eğer kalandan büyük ise bölünün sayıdan bölen eksiltilemektedir. Sayıcı devresi ile her adımda 1 artırımı yapılmaktadır. Kalan bölenden küçük olduğu zaman sayıcı devresinde her defada 1 artan değer bölüm olarak belirlenmektedir. Bölme devresinde hem bölüm hem de kalan hesabı böylece yapılmaktadır. Tasarlanan işlemcinin desteklediği buyruklar arasında rem ve div buyrukları bulunmaktadır. Böylece 2 çeşit buyruk tek modül kullanılarak desteklenebilmektedir.

3.1.2. Bellek Tasarımları

Tasarlanan sistem toplamda 4 tane önbellek içermektedir. Bunlardan 2 tanesi uyruk önbelleği ve diğer 2 tanesi veri önbelleğidir. Seviye 1 önbellekler 2 KB boyutunda ve seviye 2 önbellekler seviye 8 KB boyutundadır. Önbellek kontrolü için direct-mapped algoritması tercih edilmiştir. Bu algoritmaya öncelik verilmesinin sebebi uygulamasının diğer önbelleklere göre daha kolay oluşudur. Ayrıca direct-mapped önbellek yöntemi her satırda tek tek verinin olup olmadığını aramak yerine adres ve etiket ile doğrudan verinin olduğu noktaya gidebilmektedir. Bu da enerji tasarrufu sağlamaktadır. Bellek hiyerarşisi şekil 8’de görülmektedir.



Şekil 8. Önbellek hiyerarşisi

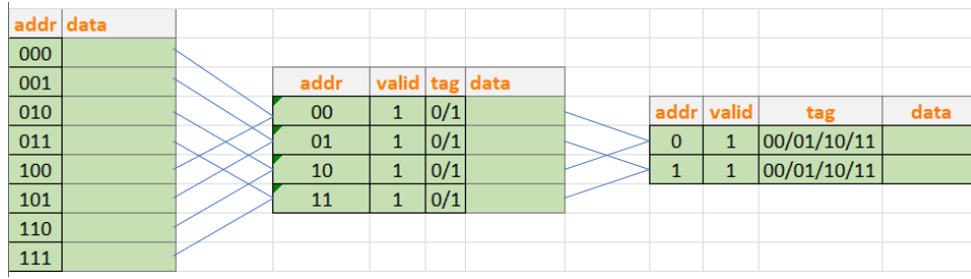
Direct-Mapped Önbellek

Direct-mapped önbellek yönetim algoritması uygulaması diğer önbellek algoritmalarına göre daha kolaydır. Direct-mapped algoritmasında veriyi bulma oranı daha düşüktür. Veri ile birlikte verinin hangi adrese ait olduğunu belirlemek için etiket ve o bölgede veri olduğunu belirlemek için valid biti bulunmaktadır. Örnek bir dizilim Şekil 9’da verilmiştir.

| Adres | data tag | valid | buyruk |
|--------|----------|-------|--------|
| 10 bit | 22 bit | 1 bit | 16 bit |

Şekil 9. Direct-mapped önbellek dizilimi

Direct mapped Şekil 10’da verildiği bir yapıya sahiptir. Örneğin 001 adresinden veri okuma yapmak istediğimizde önce seviye 1 önbelleği kontrol eder, seviye 1 önbellekte okuduğu adres 001 adresinin en anlamsız biti 1 olduğu için 1’dir. Etiket değeri 00 ise veri bulunmuştur ancak 00’dan farklıysa seviye 2 önbelleğe gider. Seviye 2 önbellekte baktığı adres 01’dir. 01 adresindeki etiket(tag) 0 ise veri bulunmuştur. Bu veriyi alıp seviye 1 önbelleğe yazar ve sonra bu veriyi işlemciye gönderir. Eğer etiket seviye 2 önbellekte farklıysa o zaman ana hafızaya gider ve 001 adresindeki veriyi alır, bu veriyi sırasıyla seviye 2 ve seviye 1 önbelleğe yazar ve daha sonra işlemciye gönderir [5]. Yazma işleminde ise seviye 1 önbellekteki verinin 1. adresine 00 etiketi ile beraber veri yazılmaktadır. Daha sonra seviye 2 önbelleğe gitmektedir ve 01 adresine 0 etiketi ile beraber veri yazılmaktadır. Son olarak ana hafızaya gidilmekte ve 001 adresine veri yazılmaktadır.



Şekil 10. Direct-mapped önbellek yapısı

Seviye 1 Buyruk Önbelleği

Direct-mapped önbellek yönetiminde farklı yöntemler olabilmektedir. Örneğin her satırda kaç tane Word tutulacağı uygulamadan uygulamaya farklılık gösterebilmektedir. Bizim izlediğimiz yöntemde her satırda 16 bitlik (half word) buyruk tutulmuştur. Her bir adreste 16 bitlik (2 bayt) buyruk olduğundan ve toplamda 2 kB (2048 bayt) buyruk olması gerektiğinden hafıza derinliği 1024 olmalıdır.

$$2048/2 = 1024$$

1024 derinliği adresleyebilmek için 10 bitlik adrese ihtiyaç duyulmaktadır.

$$2^{10} = 1024$$

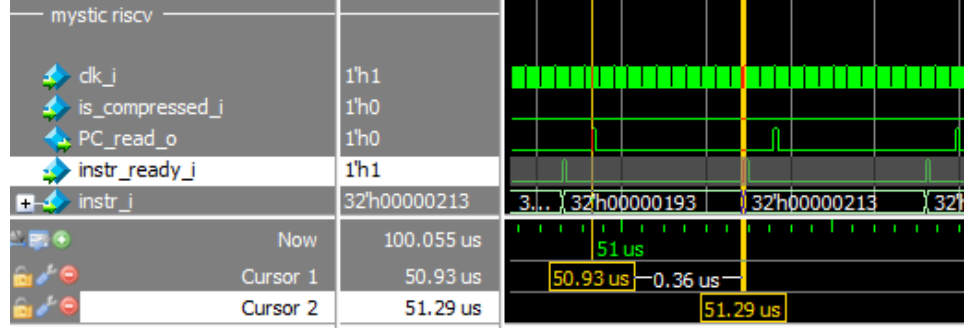
Ana hafızaya 32 bit adres ile ulaşıldığından 10 bitlik verinin dışında 22 bitlik kısımda önem taşımaktadır. O yüzden ön belleğin her adresinde 22 bitlik bir bölüm daha tutulmakta ve bu kısım 10 bitlik adresin ana hafızada gerçekten istenilen kısmın adresi olduğundan emin olunmaktadır.

"0_0000000000" ile "1_0000000000" önbellekte aynı adresi işaret etmektedir. 10 bitten sonrası önbellek hafızasında buyruk verisinin yanına yazılmakta ve adres ile birlikte bu kısım da eşleşirse o zaman o adresteki buyruk işlemciye gönderilmektedir. Her bir satırdaki tüm veriler Şekil 11'de verilmiştir. Her bir adreste toplamda 39 bit veri bulunmalıdır.

| Adres | data tag | valid | buyruk |
|--------|----------|-------|--------|
| 10 bit | 22 bit | 1 bit | 16 bit |

Şekil 11. Buyruk önbelleği dizilim

16 bitlik buyruk tutulmasının sebebi işlemcinin sıkıştırılmış buyrukları da desteklemesi ve sıkıştırılmış buyrukların 16 bit uzunluğunda oluşudur. Diğer amacı ana hafızaya giden fiziksel adres uzunluğunun 32 bit oluşudur. RISC-V Instruction Set Architecture (ISA) fiziksel adres uzunluğunda bir kısıt koymamaktadır. Ana hafızaya giden fiziksel adres uzunluğu daha uzun olsaydı her satırda 16 bitlik buyruk tutmak dezavantaj olabilirdi. Örneğin 128 bitlik adres genişliğinin olduğu bir durumda çekilen verinin 16 bit olacak şekilde önbellek buyruğuna yazımı toplamda 8 saat vuruşu sürmesi beklenirdi ($16 \times 8 = 128$). Ancak bizim uygulamamızda çekilen veri 32 bit olduğundan yalnızca verinin 2 farklı adrese yazılması gerekmektedir ve buyruk 2 saat vuruşunda seviye 1 önbelleğe yazılabilmektedir. 2 saat vuruşu makul bir süredir. Her adrese 16 bit yazılması sıkıştırılmış buyrukları okumada kolaylık sağlamaktadır. İlgili adresteki en anlamsız 2 bite bakılıp buyruğun sıkıştırılmış veya normal olduğuna karar verildiğinden eğer sıkıştırılmış ise program sayacı 2 arttırılarak diğer adrese ulaşmaktadır. Normal buyruk olması durumunda ise 16 bitlik kısım önce okunmakta ve sonrasında 32 bitlik kısmı tamamlamak için adres 2 arttırılarak diğer adres okunup işlemciye buyruk o şekilde gitmektedir. Önbelleklerde buyruk olmadığı durumda ana hafızadan çekilen buyruk önce seviye 2 önbelleğe yazılmakta, sonrasında seviye 1 önbelleğe yazılmakta ve sonrasında işlemciye iletilmektedir. Bu süre işlemci 100 MHz hızında çalıştırıldığında 0.36 us (36 saat döngüsü) sürmektedir.



Şekil 12. Buyruğun ana hafızadan okunması

Bu süre veri önbellekte bulunursa buyruğun sıkıştırılmış olup olmamasına göre değişmektedir. Eğer buyruk sıkıştırılmış ise her durumda 1 saat vuruşunda veri hazır olmaktadır. Block Ram içerisinde örnek buyruk dizilimi şekil 13'te verilmiştir. Block Ram içinde toplamda 3 farklı buyruk bulunmaktadır. Adres 0'dan veri okuma tek saat vuruşunda gerçekleşmektedir ve buyruk sıkıştırılmış olduğu için bu buyruğun tamamıdır. Adres 1 ve 2'de ise sıkıştırılmamış buyruk bulunmaktadır. 32 bitlik buyruğun en anlamsız 15 biti adres 1'de iken en anlamlı 15 biti adres 2'de konumlanmıştır. 2 farklı adresten okuma işlemi yapıldığından bu süre 2 saat vuruşunda gerçekleşmektedir.

| Adres | data tag | valid | buyruk | compressed |
|--------|----------|-------|---------|------------|
| 10 bit | 22 bit | 1 bit | 16 bit | evet/hayır |
| 3 | | 1 | buyruk3 | evet |
| 2 | | 1 | buyruk2 | hayır |
| 1 | | 1 | buyruk2 | hayır |
| 0 | | 1 | buyruk1 | evet |

Şekil 13. Önbellekte örnek buyruk dizilimi

Seviye 1 buyruk önbelleğinin bulunduğu modülün diğer görevi okunan buyruğun sıkıştırılmış olup olmadığını anlaması ve işlemciye buyruk ile birlikte bildirmesidir. Adres 0'da ve 1'de sıkıştırılmış, 2'de ve 3'te normal buyruklar bulunmaktadır. İşlemci adres 0'dan buyruk okumaya başladığında adres 0'daki bilginin en anlamsız 2 bitini kontrol etmektedir. Bu değer b11'den farklı olduğu durumda sıkıştırılmış olduğu bilgisi ise okunan değeri doğrudan işlemciye göndermektedir. İşlemcide program sayacı bu durumda 2 arttırılmaktadır. Eğer işlemci adres 2'den okuma işlemi yapıyorsa bu adresteki veri sıkıştırılmış olmadığı için en anlamsız 2 biti b11 olarak okunmaktadır. b11 okunduğunda ise bir sonraki adrese gidecek şekilde adres arttırılır ve o adresteki bilgi de okunduktan sonra art arda adresteki bilgiler birleştirilerek işlemciye 32 bitlik bir buyruk gönderilir. İşlemci bu durumda program sayacını 4 arttırmaktadır.

| Adres | Buyruk |
|-------|---------------|
| 3 | Normal |
| 2 | Normal |
| 1 | Sıkıştırılmış |
| 0 | Sıkıştırılmış |

Seviye 2 Buyruk Önbelleği

Seviye 2 buyruk önbelleği 8 kB boyutundadır ve ana hafızadan okunan 32 bitlik verinin tek saat vuruşunda önbellek adresine yazılabilecek ve seviye 1 önbellekten buyruk talep edildiğinde tek saat vuruşunda cevap verebilecek şekilde tasarlanmıştır. Seviye 2 buyruk önbelleğinde her

satırda 32 bit buyruk tutulmaktadır. Seviye 1 önbellekteğinden farklı olarak 16 bit tutulmayıp 32 bit tutulmasının sebebi verinin sıkıştırılmış veya normal oluşunun burada önemli olmayışdır. Buyruğun sıkıştırılıp olup olmadığını karar verirken kod kısmında 16 bitlik olması bizlere kolaylık sağlamıştır. 2. Seviye buyruk önbelleğinin giriş çıkış sinyalleri aşağıda verilmektedir. Seviye 1 önbellekte buyruk yok ise buraya buyruk oku talebi yapılmaktadır. Burada da eğer yoksa ana hafızaya gidip buyruk oradan çekilmektedir.

Seviye 2 önbellekte kodlama kısmında yine benzer bir mantık kurulmuştur. Seviye 2 önbellekte her bir adreste 32 bit (4 byte) buyruk tutulmaktadır ve toplamda 8kB (8192 byte) veri olması şartnamede istenmiştir. Adres derinliği hesabı aşağıdaki formül ile hesaplanmıştır.

$$8192/4 = 2048$$

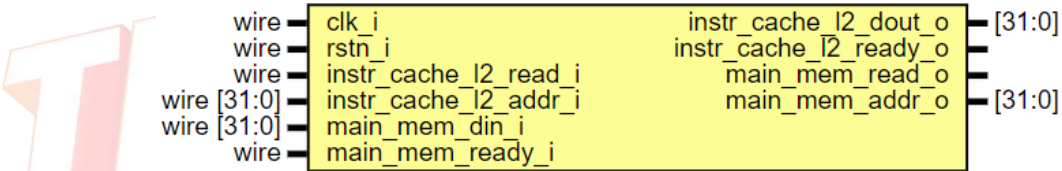
Seviye 2 buyruk önbelleği derinliği 2048 olarak belirlenmiştir. 2048 derinlik en az 11 bir ile ifade edilmelidir. Seviye 2 önbelleği her bir satır hafızada Şekil 14'teki gibi tutulmaktadır.

| Adres | data tag | valid | buyruk |
|--------|----------|-------|--------|
| 11 bit | 21 bit | 1 bit | 32 bit |

Şekil 14. Seviye 2 önbellek dizilimi

21 bit adres tag, 1 bit valid bit, 32 bit buyruk verisi olmak üzere toplamda her satırda 54 bit bulunmaktadır.

Örneğin adres 0'da herhangi bir veri olsun. Bu verinin çözümü şu şekilde olmaktadır. İlk olarak 32.bite bakılmaktadır. 32. bit 1 olduğunda bu adresteki verinin geçerli olduğu anlaşılmaktadır. Sonrasında adres etiketine bakılmaktadır. Adres tag ve data tag birbirine eşit olduğu durumda talep edilen adresteki verinin önbellekte olduğu anlaşılmaktadır. Bu durumda veri bulunmuştur ve istenilen buyruk önbellekten çekilebilmiştir. Şartların karşılanmaması durumunda ana hafızadan okuma yapılmaktadır. Buyruk önbellekte bulunduğunda tek saat vuruşunda hızlı bir şekilde seviye 2 önbellekten çekilerek seviye 1 önbelleğe aktarılmaktadır. Şekil 15'te seviye 2 önbelleğin ana hafıza ve seviye 1 önbellek ile bağlantılarını sağlayan giriş çıkış sinyalleri görülmektedir.



Şekil 15. Seviye 2 buyruk önbelleği

Veri Önbelleği:

Veri önbelleğinde buyruk önbelleğinde olduğu gibi direct-mapped önbellek yöntemi tercih edilmiştir. Buyruk önbelleğinden farklı olarak işlemci tarafından yalnızca okuma talebi değil yazma talebi de gelmektedir. Veri Hafızasına erişim LB, LH, LW, LD, SB, SH, SW ve SD komutları ile olmaktadır. Bu komutlar ile hafızaya byte (8 bit) half Word(16 bit), Word(32 bit) ve double Word(64 bit) şeklinde erişmem mümkündür. Hafızadan veri okunacağı zaman ilk olarak seviye 1 önbellek kontrol edilmektedir. Seviye 1 önbellekte olmayan veriler için seviye 2 önbelleğe bakılmaktadır. Seviye 2 önbellekte olmayan veriler için ise ana hafızaya gidilmekte ve veri oradan okunmaktadır. Veri yazılacağı zaman ise önce seviye 1 hafızadaki işaret ettiği yere, sonra seviye 2 önbellekteki işaret edilen yere ve son olarak ana hafızadaki adrese veri yazılmaktadır. Görüldüğü üzere yazma işlemi direct-mapped metodu kullanıldığı zaman oldukça maliyetlidir. Normalde yalnızca ana hafızaya yazılsaydı daha önbelleklere yazma işlemi yapılmayacağı için daha hızlı olacağı görülmektedir. Burada avantajı okuma kısmındadır. Veri

önbelleğe yazıldıktan sonra işlemci tarafından okunması muhtemeldir. Bu noktada okuma işleminde ana hafızaya gitmeye gerek duyulmadan aynı veri önbellekte de olduğundan okuma işlemi hızlıca gerçekleştirilmektedir.

Seviye 1 Veri Önbelleği:

Seviye 1 önbelleği şartnamede 2 kB olarak istendiğinden boyutu 2kB olarak seçilmiştir. 64 bitlik işlemci tasarlandığından her satırda 64 bit uzunluğunda veri tutulması tercih edilmiştir. İşlemci yaz veya oku komutundan sonra ilgili adreste yazma veya okuma işlemi yapılmaktadır. 2kB veri için toplamda gereken derinlik formülü aşağıda verilmiştir.

$$2kB (2048 \text{ byte}) / 64 \text{ bit (8 byte)} = 256 = 2^{**8}$$

| Adres | data tag | valid | veri |
|-------|----------|-------|--------|
| 8 bit | 24 bit | 1 bit | 64 bit |

Şekil 16. Seviye 1 veri Önbelleği dizilimi

Görüldüğü üzere seviye 1 önbelleği adreslemek için en az 8 bite ihtiyaç duyulmaktadır. Buradan yola çıkarak her satırdaki veri uzunluğu aşağıdaki gibi hesaplanmaktadır.

24 bit data tag + 1 bit valid + 64 bit veri = 89 bit veriye ihtiyaç duyulmuştur. 32 bitlik adresin 8 biti block ram adreslemesinde kullanılırken kalan 24 bit tag(etiket) bölgesine yazılmaktadır.

Seviye 2 Veri Önbelleği:

Seviye 2 veri önbelleğinde seviye 1 önbelleğe benzer yapı kullanılmıştır. Veri boyutu 8 KB olduğundan hafıza derinliği değişmiştir. 2 KB veri 8 bir ile adreslenebilirken 8KB veri 10 bit ile adreslenmiştir. Her satırda 64 bitlik veri tutulmaktadır. Her satırda tag ve valid biti ile toplamda 87 bit bulunmaktadır.

| Adres | data tag | valid | veri |
|--------|----------|-------|--------|
| 10 bit | 22 bit | 1 bit | 64 bit |

Şekil 17. Seviye 2 veri önbelleği dizilimi

Ana hafızadan veri okunup seviye 2 önbelleğine yazım işlemi 2 aşamada gerçekleşmektedir. Ana hafızaya giden fiziksel adres ve veri 32'şer bit olduğundan 64 bit veri okuma için ana hafızaya 2 defa gitmek gerekmektedir. Benzer şekilde ana hafızaya 64 bitlik veri yazılacağı zaman bu işlem 2 aşamada gerçekleşmektedir. Önce 0'dan 31'e kadar tutulan 32 bitlik veri yazılmaktadır. Sonrasında 32'den 63'e kadar olan veri ana hafızaya yazılmaktadır. Burada fiziksel adresin daha geniş olması tek seferde daha fazla verinin yazılabilmesine olanak sağlamaktadır. Ancak teknofest_wrapper da verilen adres 32 bit olarak verilmiştir.

3.1.3. Çevre Birimleri Tasarımları

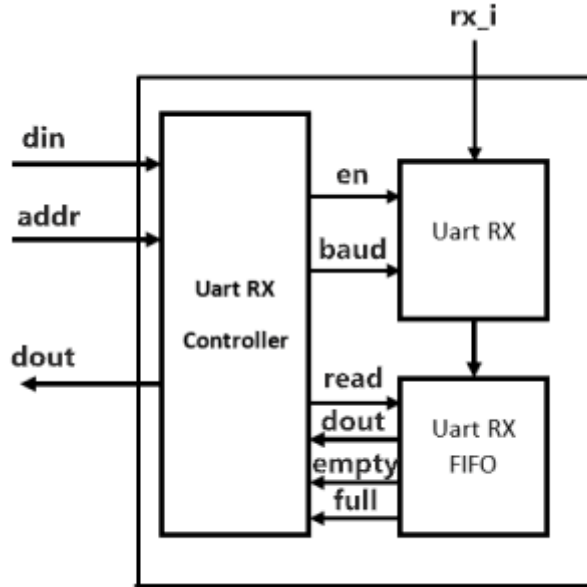
Nihai tasarım Serial Peripheral Interface (SPI) ve Universal asynchronous receiver-transmitter (UART) çevre birimlerini içermektedir. SPI ve UART arayüzleri dış dünya ile seri bir şekilde haberşeme yapmak için kullanılmaktadır.

UART RX

UART asenkron olarak haberleşme yapmayı sağlayan seri haberleşme türüdür. Bu yarışma kapsamında işlemcinin dış dünya ile haberleşmesini sağlamak amacı ile tasarıma dahil edilmiştir. Tasarlanan UART arayüzü şartnamede istenilen özelliklerin hepsine sahiptir. Bu özellikler şu şekildedir:

1 start bit, 8 veri bit, 1 stop bit, programlanabilir baud hızının en az 1 Mbps'ye kadar olması, 32x8 rx buffer, 32x8 tx buffer.

Tasarlanan UART arayüzünün dış dünyadan veri alıcı kısmının blok tasarımı Şekil 18'de verilmiştir.



Şekil 18. UART alıcı arayüzü

UART arayüzünde din ve addr girişleri görülmektedir. UART arayüzünün toplamda 4 farklı adresi vardır. İşlemciden bu 4 adresten biri gönderildiğinde UART arayüzü işlem yapmaya başlamaktadır. Bu adresler aşağıda verilmiştir.

| Adres | İsim | Açıklama |
|------------|-------------|--------------------|
| 0x20000000 | uart_ctrl | Kontrol yazmacı |
| 0x20000004 | uart_status | Durum yazmacı |
| 0x20000008 | uart_rdata | Veri okuma yazmacı |
| 0x2000000c | uart_wdata | Veri yazma yazmacı |

Şekil 19. UART arayüzü adres bilgileri

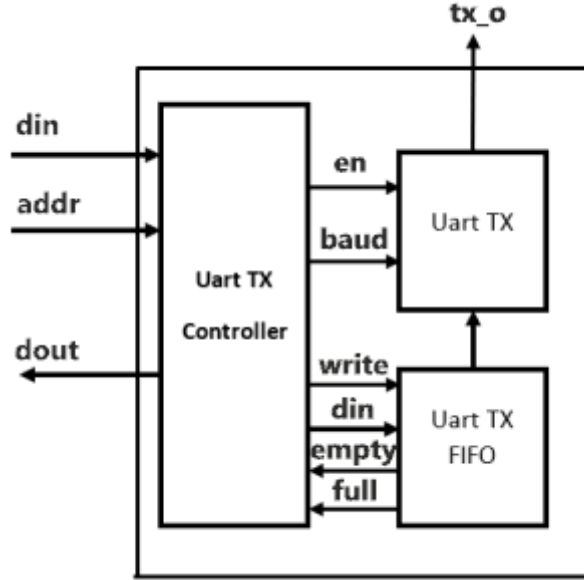
İşlemci UART alıcı devresinin baud hızını ve aktiflik durumunu bu yazmaç bu adres ile gelen 32 bitlik veri ile belirlemektedir.

UART arayüzü aktif olması durumunda dış dünyadan veri gönderildiğinde alabilmesi gerekmektedir. Seri olarak alınan verilerin alımı tamamlandıkça UART RX Buffer içinde 8 bit olarak depolanmaktadır. Depolanan veriler işlemcinin 0x20000008 adresinde bulunan yazmaç ile okunmaktadır. İşlemciden bu adres geldiğinde eğer FIFO içerisinde veri varsa 8 bit olacak şekilde işlemciye gönderilmektedir. İşlemciden 0x20000004 adresi geldiğinde işlemciye cevap olarak FIFO dolu ve boş bilgileri gönderilmektedir.

UART TX

UART arayüzünün dış dünyaya veri gönderdiği kısmın blok tasarımı Şekil 20'de verilmiştir. Gönderici devresi toplamda 32 tane 8 bitlik verini saklanabileceği kadar alana sahip FIFO'ya sahiptir. Veri gönderimine işlemciden gelen aktivasyon UART_TX bloğunu aktif eden 0x20000000 adresinden veri geldikten sonra başlamaktadır. Veri gönderimi eğer FIFO TX FIFO

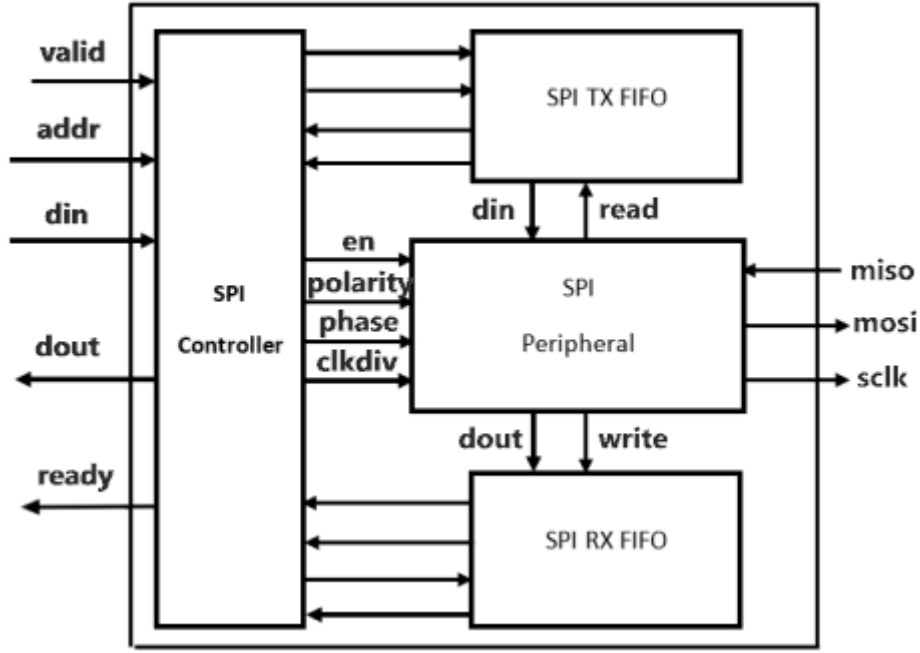
boş değil ise yapılmaktadır. FIFO boşalana kadar gönderim devam etmektedir. Baud hızlı alıcı devresinde olduğu gibi işlemci tarafından ayarlanabilir olacaktır. UART TX FIFO'suna veri yazımı işlemi tarafından gelen SB, SH, SW, SD komutları ile gerçekleştirilebilecektir. Bu komutlar ana hafıza veri yazımında da kullanılan komutlardır. İşlemci tarafından gelen adres UART arayüzü adresini işaret ediyor ise UART arayüzü işlem yapabilecektir. UART gönderici devresinde ait FIFO'nun boş veya dolu olduğu işlemciden gelen adresin 0x20000004 olduğu durumda işlemciye bildirilebilecektir.



Şekil 20. UART gönderici arayüzü

SPI

SPI çevre birimi seri arayüzü ile, flash hafıza, ADC, DAC, sıcaklık sensörü, imze sensörü gibi entegreler ile haberleşme amaçlanmaktadır. Tasarlanan arayüz tüm polarite ve faz modlarını desteklemektedir. SPI saat hızı işlemci tarafından programlanabilmektedir. Alıcı ve verici tarafından 8'er tane 32 bitlik veri depolanabilecek kadar FIFO buffer bulunmaktadır. Tasarlanan arayüzün blok tasarımı Şekil 21'de verilmiştir. Blok tasarımı haberleşmede arayüzde kullanılan giriş çıkış sinyalleri de görülmektedir. Valid 1 olduğu zaman işlem başlamaktadır ve ready 1 olana kadar sürmektedir. Ready sinyali 1 olduğu zaman işlemciye haber gitmektedir ve işlemci diğer işlemlerine devam etmektedir.



Şekil 21. SPI arayüzü blok tasarımı

SPI arayüzü işlemci tarafından gönderilen adresin Şekil 22’de verilen adreslerden biri olduğunda aktif olmaktadır. Bu adresler SPI arayüzünün kontrolünde farklı amaçlar için kullanılmaktadır.

| Adres | İsim | Açıklama |
|------------|------------|---------------------|
| 0x20010000 | spi_ctrl | Kontrol yazmacı |
| 0x20010004 | spi_status | Durum yazmacı |
| 0x20010008 | spi_rdata | Seri alma yazmacı |
| 0x2001000c | spi_wdata | Seri iletim yazmacı |
| 0x20010010 | spi_cmd | Komut yazmacı |

Şekil 22. SPI yazmaç bilgileri

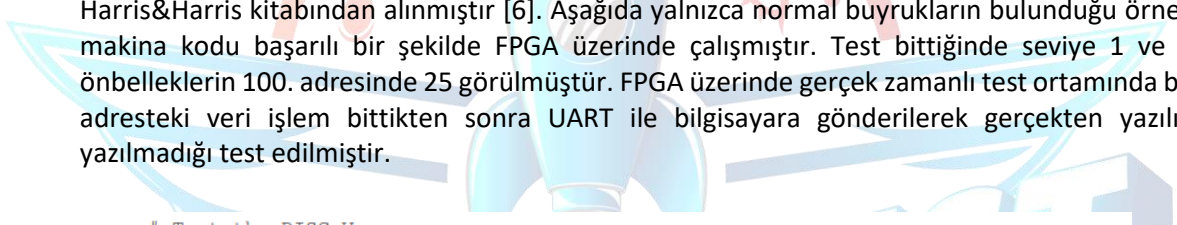
Kontrol yazmacı adresi geldiğinde işlemci tarafından alınan 32 bitlik veri ile polarity, phase bilgileri, spi saat hızı, SPI arayüzü aktivasyon ve sıfırlama işlemleri belirlenmektedir. Durum yazmacı geldiğinde SPI alıcı ve verici için kullanılan FIFO depolama alanlarının boş ve dolu bilgileri işlemciye gönderilir. Komut yazmacı ile SPI transferinde seri verilerin dış dünyadan alınacağı veya dış gönyaya gönderileceği belirlenir. Bu transferin kaç byte olacağı bilgisi yine bu yazmaçta bulunan veri ile belirlenmektedir. SPI gönderici FIFO içerisine yazılacak olan veriler işlemci tarafından gelmektedir. Spi_wdata isimli yazmacın adresindeki veri işlemci talep ettiğinde gönderici FIFO içerisine yazılmaktadır. Yazılan veriler 32 bir uzunluğundadır. İşlemci SB, SH, SW, SD komutları ile yazma işlemini gerçekleştirebilmektedir. Veri 32 bit olduğundan SW(store word) komutu tercih etmek mantıklı olacaktır. Çünkü bu buyruk ile 32 bitlik veri yazma işlemi yapılmaktadır. SPI alıcı FIFO içerisinde ise SPI seri arayüzünde ile dış dünyadan gelen veriler alınan veriler depolanmaktadır. FIFO dolana kadar bu işlem alım işlemi gerçekleşebilmektedir. Dolu olduğu durumda veri gelmesi durumunda bu veriler gözardı edilmektedir. Dış dünyadan alınacak olan verilerin uzunluğu komut yazmacı ile belirlenmektedir.

4. ÇİP TASARIM AKIŞI

Çip tasarım akışında Ubuntu ve Openlane bilgisayarımıza kurulmuştur. Tasarımımızın openlane ortamında sentezlenebilmesi için gereken Open SRAM modellerinin sistemimize entegrasyonu üzerinde çalışılmaktadır. Tasarımımızın Block RAM kullanılmayan alt modülleri başarılı bir şekilde sentezlebilmiştir ve Timing analizi yapılmıştır. Open SRAM kullanmadan sentezleme yapmaya çalışılmıştır ancak bu boyutlar büyük olduğundan mümkün olmamıştır. 32x64 boyutundaki register file dahi sentezlenememiştir. Bu birim, önbellek blokları Open SRAM modeli ile değiştirilecektir. Sentezleme başarılı bir şekilde gerçekleştirildikten sonra openlane akışında gerekli olan diğer testler yapılacaktır.

5. TEST

FPGA ortamında Block ve Distributed RAM kullanarak ana hafıza ve önbellek blokları oluşturulmuştur. İşlemcinin FPGA testi için ilk olarak ana hafızada SBOX dönüşümü kullanılarak karıştırılmış, sıkıştırılmış ve normal buyruklar konulmuştur. Sonra bu buyrukların önbelleklere doğru bir şekilde çekilip çekilmediği kontrol edilmiştir. Her bir birim ayrı ayrı doğrulanmıştır. Bütün olarak doğrulama işleminde ise önce toplama, XOR, atlama, Store, Load gibi buyrukların bulunduğu kısa ve basit testler yapılmıştır. Daha uzun testler TÜBİTAK tarafından sağlanan makina kodları ile yapılacaktır. Örnek bir test kodu Şekil 23'te verilmiştir. Test kodu Harris&Harris kitabından alınmıştır [6]. Aşağıda yalnızca normal buyrukların bulunduğu örnek makina kodu başarılı bir şekilde FPGA üzerinde çalışmıştır. Test bittiğinde seviye 1 ve 2 önbelleklerin 100. adresinde 25 görülmüştür. FPGA üzerinde gerçek zamanlı test ortamında bu adresteki veri işlem bittikten sonra UART ile bilgisayara gönderilerek gerçekten yazılıp yazılmadığı test edilmiştir.



| # Test the RISC-V processor: | | | | |
|--|--------------------|--------------------------|---------|--------------|
| # add, sub, and, or, slt, addi, lw, sw, beq, jal | | | | |
| # If successful, it should write the value 25 to address 100 | | | | |
| | RISC-V Assembly | Description | Address | Machine Code |
| main: | addi x2, x0, 5 | # x2 = 5 | 0 | 00500113 |
| | addi x3, x0, 12 | # x3 = 12 | 4 | 00C00193 |
| | addi x7, x3, -9 | # x7 = (12 - 9) = 3 | 8 | FF718393 |
| | or x4, x7, x2 | # x4 = (3 OR 5) = 7 | C | 0023E233 |
| | and x5, x3, x4 | # x5 = (12 AND 7) = 4 | 10 | 0041F2B3 |
| | add x5, x5, x4 | # x5 = 4 + 7 = 11 | 14 | 004282B3 |
| | beq x5, x7, end | # shouldn't be taken | 18 | 02728863 |
| | slt x4, x3, x4 | # x4 = (12 < 7) = 0 | 1C | 0041A233 |
| | beq x4, x0, around | # should be taken | 20 | 00020463 |
| | addi x5, x0, 0 | # shouldn't execute | 24 | 00000293 |
| around: | slt x4, x7, x2 | # x4 = (3 < 5) = 1 | 28 | 0023A233 |
| | add x7, x4, x5 | # x7 = (1 + 11) = 12 | 2C | 005203B3 |
| | sub x7, x7, x2 | # x7 = (12 - 5) = 7 | 30 | 402383B3 |
| | sw x7, 84(x3) | # [96] = 7 | 34 | 0471AA23 |
| | lw x2, 96(x0) | # x2 = [96] = 7 | 38 | 06002103 |
| | add x9, x2, x5 | # x9 = (7 + 11) = 18 | 3C | 005104B3 |
| | jal x3, end | # jump to end, x3 = 0x44 | 40 | 008001EF |
| | addi x2, x0, 1 | # shouldn't execute | 44 | 00100113 |
| | add x2, x2, x9 | # x2 = (7 + 18) = 25 | 48 | 00910133 |
| | sw x2, 0x20(x3) | # [100] = 25 | 4C | 0221A023 |
| done: | beq x2, x2, done | # infinite loop | 50 | 00210063 |

Şekil 23. Örnek assembly ve makina kodu

6. TAKIM ORGANİZASYONU

6.1. Takım Organizasyonu



6.2. Görev Dağılımı

| İSİM | KATKILARI |
|-------------------------------|--|
| Prof. Dr. Özcan Öztürk | Bu projedeki rolü çoğunlukla ekip üyelerine mikro mimari, önbellek hiyerarşisi ve ALU'da cebirsel işlemlerin nasıl ele alınacağı konusunda yardımcı olmaya odaklanmaktadır. |
| Muhammed Kocaoğlu | Tasarım gereksinimlerini değerlendirmekten ve üst düzey mimari tasarım oluşturmaktan sorumludur. Ayrıca dijital tasarım modülleri parametresi ve port tanımlarının oluşturulmasından sorumludur. Modüller için Verilog kodları yazma görevleri, kendisi ve ekip üyesi arasında dağıtılacaktır. Mikromimari, boru hattı ve kontrol ünitelerini içeren modüller tasarlayacaktır. OpenLANE akışını kullanarak SoC tasarımını ASIC'e taşımak da onun görevi olacaktır. |
| Muhammed Yıldırım | Çevresel veri yolu(wishbone), UART ve SPI çevre birimleri tasarımlarından sorumlu olacaktır. Ayrıca önbellek hiyerarşisi, önbellek adresi kod çözme, önbellekler ve önbellek kayıplarını ele alacak. Tasarımın FPGA panosunda doğrulanması ve uygulanması için RISC-V talimatları ile test vektörleri oluşturmak ana sorumlulukları olacaktır. Her iki katılımcı da RTL modüllerini doğrulayacaktır. |

7. İŞ PLANI ve RİSK PLANLAMASI

| Tamamlanma Durum | İş Paketleri | Başlangıç | Bitiş | Mart | | | Nisan | | | Mayıs | | | Haziran | | | Temmuz | | | Ağustos | | |
|------------------|---|------------|------------|------|---|---|-------|---|---|-------|---|---|---------|---|---|--------|---|---|---------|---|---|
| | | | | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 1 | 2 | 3 |
| TRUE | Literatür Araştırması | 14.03.2022 | 01.07.2022 | | | | | | | | | | | | | | | | | | |
| TRUE | Genel Tasarımın Oluşturulması | 14.03.2022 | 11.04.2022 | | | | | | | | | | | | | | | | | | |
| TRUE | Ulanılacak Tasarım Tekniklerinin Belirlenmesi | 14.03.2022 | 11.04.2022 | | | | | | | | | | | | | | | | | | |
| TRUE | Hedef Performansın Belirlenmesi | 22.03.2022 | 11.04.2022 | | | | | | | | | | | | | | | | | | |
| TRUE | Şematik Seviye Tasarım | 18.03.2022 | 13.06.2022 | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | |
| TRUE | UART ve SPI Çevre Birimlerinin Tasarımı | 11.04.2022 | 25.04.2022 | | | | | | | | | | | | | | | | | | |
| TRUE | Çevresel Veri Yolu Tasarımı | 11.04.2022 | 24.05.2022 | | | | | | | | | | | | | | | | | | |
| TRUE | Temel Buyruk Seti Kodlanması | 18.04.2022 | 09.06.2022 | | | | | | | | | | | | | | | | | | |
| TRUE | Özel Buyruk Seti Kodlanması | 18.04.2022 | 24.05.2022 | | | | | | | | | | | | | | | | | | |
| TRUE | S-box Tasarımı | 02.06.2022 | 13.06.2022 | | | | | | | | | | | | | | | | | | |
| TRUE | Bellek Tasarımı | 24.05.2022 | 21.06.2022 | | | | | | | | | | | | | | | | | | |
| TRUE | Önbellek Tasarımı | 24.05.2022 | 24.06.2022 | | | | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | | | | | | | |
| FALSE | OpenLane Akışı | 09.06.2022 | 07.07.2022 | | | | | | | | | | | | | | | | | | |
| TRUE | Test / Analiz | 16.06.2022 | 09.07.2022 | | | | | | | | | | | | | | | | | | |

Tasarlanan işlemcinin RTL kodlaması tamamlanmış olup testler devam etmektedir. Openlane kullanarak çip tasarım akışı üzerinde çalışılmış ancak DTR'ye yetişmemiştir.

8. KAYNAKÇA

- [1] David Patterson ve Andrew Waterman (2017) The RISC-V Reader: An Open Source Architecture Atlas, Beta Edition, 0.0.1
- [2] David A. Patterson and John L. Hennessy (2018), Computer Organization and Design The Hardware/Software Interface: RISC-V Edition
- [3] Nandland, Carry Lookahead Adder in VHDL and Verilog, Erişim: 12.05.2022, <https://nandland.com/carry-lookahead-adder/>.
- [4] GeeksforGeeks, Non-restoring Division for Unsigned Integer, Erişim: 25.05.2022, <https://www.geeksforgeeks.org/non-restoring-division-unsigned-integer/>.
- [5] Sarah L. Harris, David Money Harris, Digital Design and Computer Architecture RISC-V Edition, p.507
- [6] Sarah L. Harris, David Money Harris, Digital Design and Computer Architecture RISC-V Edition, p.464