T.C.
ERCİYES ÜNİVERSİTESİ
MÜHENDİSLİK FAKÜLTESİ

İŞLEMCİ TASARIM VE MODELLEME

Hazırlayan

Ömer Can VURAL
1030516774

Danışman
Dr. Öğr. Üyesi Fehim KÖYLÜ

Bilgisayar Mühendisliği
Bitirme Ödevi

Eylül 2022
KAYSERİ

"İşlemci Tasarım ve Modelleme" adlı bu çalışma, jürimiz tarafından Erciyes Üniversitesi Bilgisayar Mühendisliği Bölümünde Bitirme Ödevi olarak kabul edilmiştir.

25/01/2018

JÜRİ :

Danışman : Dr. Öğr. Üyesi Fehim KÖYLÜ

Üye :

Üye :

ONAY :

Yukarıdaki imzaların, adı geçen öğretim elemanlarına ait olduğunu onaylarım.

.... / .... /20...
Prof. Dr. Veysel ASLANTAŞ
Bilgisayar Müh. Bölüm
Başkanı

ÖNSÖZ / TEŞEKKÜR

Bu çalışmada, literatürde yer alan, problemin çözümüne ilişkin yapılmış önceki çalışmalar hakkında ki bilgiler ve ele alınan problemin çözümünde kullanılması planlanan yöntemler konsept proje çerçevesinde bir araya getirilmiş ve gerçekleştirimiştir.

Elde ki kaynaklardan yararlanılarak VHDL donanım tanımlama dili ile basit bir işlemci modeli tasarımı yapılmış ve donanım tasarım programları ile test ve doğrulama adımları gerçekleştirilerek mikroişlemci çalışma mantığı tüm yönleri ile mercek altına alınmıştır. Elde edilen bulgular doğrultusunda bir mikroişlemci tasarımı ortaya çıkarılmıştır.

Yapılan mikroişlemci tasarımına istinaden, işlemci çıktılarını görselleştirmek üzere kullanılması planlanan bir grafik kartı, bit sayıcı, NAND ve NOT kapıları, EEPROM bellek gibi komponentler ile lehimsiz devre kartları üzerine kurularak oluşturulmuş ve fiziksel bir grafik işlemci modeli elde edilmiştir.

Çalışma sonucunda bir işlemci tasarımı ve bir grafik kartı modeli oluşturulmuştur. Çıktıların bir arada çalışarak temel bilgisayar sistemini oluşturması hedeflenmiştir.

Bu Bitirme çalışmasının hazırlanmasında katkıda bulunan, değerli bilgilerini benimle paylaşan, kıymetli zamanını ayıran Dr. Ögr. Üyesi Fehim KÖYLÜ hocama ve her türlü koşulda benden hiçbir zaman maddi ve manevi yardımlarını esirgemeyen, arkamda duran aileme sonsuz teşekkürlerimi sunarım.

İŞLEMCİ TASARIM VE MODELLEME

Ömer Can VURAL
Erciyes Üniversitesi, Bilgisayar Mühendisliği
Danışman : Dr. Öğr. Üyesi Fehim KÖYLÜ

ÖZET

Bilgisayar donanımı Kaydediciler, mantık devreleri, sonlu durum makineleri ve bellek gibi yapılardan meydana gelir. Bu donanımların belirli birtakım komutları gerçekleştirmek üzerine tasarlanması o yapının mimarisini belirler. Bu komutlar 1 ve 0 lardan oluşan ikili değerler olarak program belleğine kaydedilirler ve sıra sıra işlenerek bilgisayara işlem yaptırılmış olur. Bu komutlara yazılım denir. Her yazılım sadece donanımın izin verdiği komutlar sınırında yazılabilir.

İşlemcimiz 8-bit, dolayısıyla komutlar dahil işlenecek ve kaydedilecek her veri 8-bit uzunluğunda olabilir. Tasarladığım işlemci iki kısımdan oluşan basit bir komut yapısına sahiptir; "Opcode" ve "Operand". Opcode, komutun kimliğidir ve her komut için özel bir Opcode vardır. Operand ise komutun işlenmesi için gerekli olan veridir.

Bu çalışmanın ilk aşamasında 8 farklı altblok içeren 8-bitlik bir SYSTEM-ON Chip yapısı, Vivado programında VHDL donanım tanımlama dili ile tasarlanmış ve başarıyla simüle edilmiştir.

Tasarlanan işlemci yapısını, merkezi işlem ünitesi (CPU) ve bellek sistemi olarak iki ana kısımda inceleyebiliriz. CPU birimi, Kontrol ünitesi ve Veri Yolu altbloklarını içerir. Kontrol ünitesi, komutların kontrolünü ve uygulanmasını sağlayan durum makinasıdır. Komutun ilgili bellek adresinden alınması, hangi komut olduğunun algılanması ve ilgili işlemlerin yapılmasından sorumludur. Kontrol ünitesi, IR daki değere göre işlemlerini düzenler. MAR, bellekten okuma yazma yapmak için gerekli olan adres bilgisini içerir. PC, bellekten hangi komutun okunacağını bildiren adres bilgisidir. Kaydediciler ve ALU, CPU nun Veri Yolu (Data Path) adı verilen kısmını oluşturur. A ve B Kaydedicileri, Operandların tutulduğu veri kaydedicileridir. CCR ise ALU sonucunun hangi Bayrağı etkileyeceği bilgilerini tutan bir Kaydedicidir. Aritmetik Mantık Ünitesi (ALU), aritmetiksel ve mantıksal işlemleri yapmakla görevlidir. BUS1, Verileri belleğe iletmek ile görevlidir. BUS2, bellekten okunan verinin CPU ya iletilmesinde görevlidir.

Bellek Sistemi (Memory) bloğunda alt blok olarak program belleği, veri belleği ve INPUT/OUTPUT portları tanımlanmıştır. Program belleği (ROM) Read-Only yani sadece okumaya izin veren yapılardır. İlgili komutları çalıştırmak için bilgisayar tarafından işlenecek komutlar Program belleğinde tutulur. Veri belleği (RAM), programın çalışması sırasında ele alınan verilerin kaydedildiği ve okunduğu bellektir. INPUT, OUTPUT portları da dış dünyadan veri alabilmek ve veri iletebilmek için kullandığımız çevre birimleridir.

Anahtar Kelimeler: CPU, Kaydedici, HDL, Veri, Bellek.

PROCESSOR DESIGN AND MODELING

Ömer Can VURAL
Erciyes University, Computer Engineering
Supervisor: Asst. Prof. Fehim KÖYLÜ

ABSTRACT

Computer hardware consists of structures such as registers, logic circuits, finite state machines, and memory. The design of these hardware to perform certain commands determines the architecture of that structure. These commands are recorded in the program memory as binary values consisting of 1s and 0s, and they are processed sequentially and the computer is processed. These commands are called software. Each software can only be written within the limits of the commands allowed by the hardware.

Our processor is 8-bit, so any data to be processed and saved, including instructions, can be 8-bits long. The processor I designed has a simple instruction structure consisting of two parts; "Opcode" and "Operand". Opcode is the ID of the command and there is a specific Opcode for each command. The operand is the data required to process the instruction.

In the first stage of this study, an 8-bit SYSTEM-ON Chip structure containing 8 different subblocks was designed and successfully simulated in the Vivado program with the VHDL hardware description language.

We can examine the designed processor structure in two main parts as the central processing unit (CPU) and the memory system. It contains the CPU unit, Control unit and Bus subblocks. The control unit is the state machine that provides control and execution of commands. It is responsible for receiving the command from the relevant memory address, detecting which command it is and performing the related operations. The control unit regulates its operations according to the value in IR. The MAR contains the address information required to read and write from memory. PC is address information that tells which instruction to read from memory. The registers and the ALU make up the part of the CPU called the Data Path. Registers A and B are data registers where Operands are kept. On the other hand, CCR is a Register that keeps the information on which Flag the ALU result will affect.

The Arithmetic Logic Unit (ALU) is responsible for performing arithmetic and logical operations. BUS1 is responsible for transmitting data to memory. BUS2 is responsible for transmitting the data read from the memory to the CPU.

Program memory, data memory and INPUT/OUTPUT ports are defined as sub-blocks in the Memory System block. The commands to be processed by the computer to execute the relevant commands are kept in the Program memory (ROM). Data memory (RAM) is the memory in which data handled during program execution is stored and read. INPUT and OUTPUT ports are the peripherals we use to receive and transmit data from the outside world.

Keywords: CPU, Register, HDL, Data, Memory.

İÇİNDEKİLER

İŞLEMCİ TASARIM VE MODELLEME

TABLOLAR LİSTESİ

ŞEKİLLER LİSTESİ

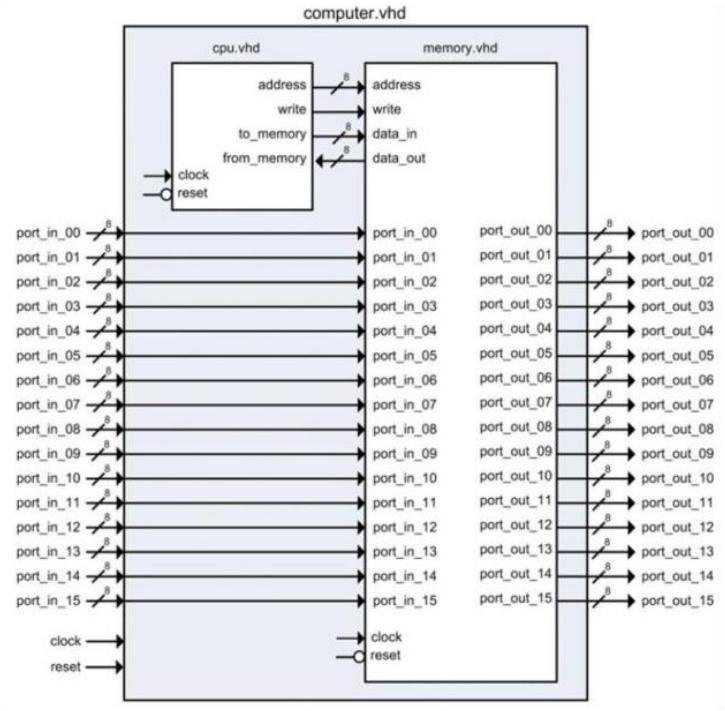
KISALTMALAR

IR	:	Instruction Register - Komut Kaydedicisi
MAR	:	Memory Address Register - Bellek Adres Kaydedicisi
ALU	:	Arithmetic Logic Unit - Aritmetik Lojik Ünitesi
CPU	:	Central Processing Unit - Merkezi İşlem Birimi
ROM	:	Read-Only Memory - Salt Okunur Bellek
RAM	:	Random Access Memory - Rastgele Erişim Belleği
CCR	:	Condition Code Register - Şartlı Kod Kaydedicisi
Reg	:	Register - Kaydedici
HDL	:	Hardware Definition Language - Donanım Tanımlama Dili

GİRİŞ

Giriş.1. İşlemci Mimarisi

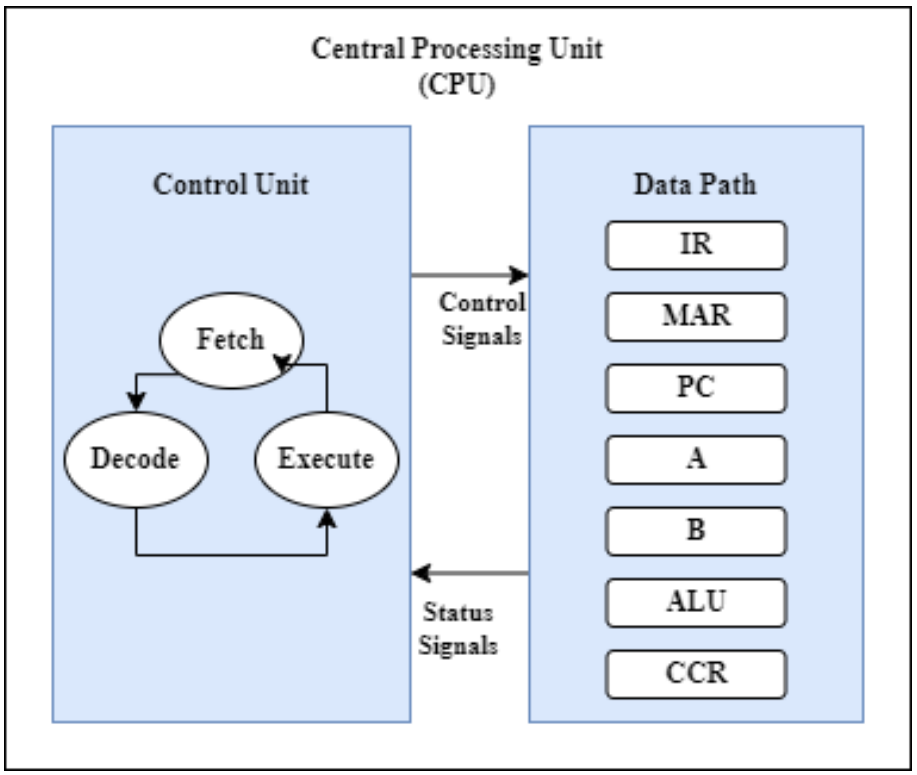
Bilgisayar donanımı Kaydediciler, mantık devreleri, sonlu durum makineleri ve bellek gibi yapılardan meydana gelir. Bu donanımların önceden belirlenmiş birtakım komutları gerçekleştirmek üzerine tasarlanması o yapının mimarisini belirler. Bu komutlar 1 ve 0 lardan oluşan değerler olarak program belleğine kaydedilirler ve sıra sıra işlenerek bilgisayarda işlem yaptırılmış olur.



Şekil G.1. Üst-Seviye VHDL Tasarım Diyagramı

Bu tasarımımızı temsil eden Üst-Seviye diyagram. Her yapının içerisine girip onu oluşturan alt bloklara ve mimariye ayrı ayrı bakmamız ve bunları doğru bir şekilde birleştirmemiz gereklidir. Temel olarak bilgisayarları, merkezi işlem ünitesi (Central Processing Unit – CPU) ve bellek sistemi larak ikiye ayırabiliriz. CPU birimini oluşturan Kontrol-Ünitesi ve Veri-Yolu alt bloklarını inceleyelim;

Giriş.2. CPU Mimarisi



Şekil G.2. CPU Mimarisi

Kontrol ünitesi, komutların kontrolünü ve uygulanmasını sağlayan durum makinasıdır. Bu durum makinesi, komutun ilgili bellekten alınması, komutun hangi komut olduğunun algılanması ve ilgili işlemin yapılmasından sorumludur. Her yeni komut durum makinesinin durumunu değiştirerek CPU nun çalışma modunu belirler.

Kaydediciler ve ALU, CPU'nun veri yolu adı verilen kısmını oluşturur. Kaydediciler veri alışverişinin ve manipülasyonlarının yapıldığı hızlı hafıza elemanlarıdır. Hızlıdır çünkü aynı saat darbesinde veri okuma yazma yapılabilir, o an kullanılması beklenen yüksek öncelikli veriler Kaydedicilerde saklanır. Öncelikli olmayan veriler ise RAM gibi yavaş ama büyük bellek yapılarında saklanır.

İşlemcimizde veri yolu, ALU haricinde tamamen özelleşmiş kaydedicilerden meydana gelmektedir:

• **Instruction Register (IR) :** (Komut Kaydedicisi) sırada ki işlenecek komutun ikili değerini tutan Kaydedicidir. Program belleğinden alınan komut, bu register da tutulur.

• **Memory Address Register (MAR) :** (Bellek Adresi Kaydedicisi) Belleğe ulaşmak için gerekli olan adres bilgisinin tutulduğu ve dışarı iletildiği Kaydedicidir. Eğer CPU bellekten bir veri okuyacaksa okuma yapılacak olan adres MAR tarafından tutulup iletilir. Komutları bellekten okuma veya RAM'e veri yazma gibi işlemler için MAR'ın belleğe ilettiği adres bilgisine ihtiyaç vardır.

• **Program Counter (PC) :** (Program Sayacı) Sırada ki işlenecek komutun program belleğinde ki adresini tutuar. İşlemci başladığında değeri 0 dır ve her komuttan sonra değeri 1 artar. Program belleğinde komutlar sıra sıra okunur ve işlenir. Bu sırayı program sayacı tutar, program sayacı hangi değeri gösteriyorsa o adresteki komut bellekten okunuyor ve komut kaydedicisine alınıyor.

• **Condition Code Register (CCR) :** (Koşul Kodu Kaydedicisi) ALU işlemleri sonucunda oluşturulan durum bayraklarını tutar. Bu durum bayrakları negatif, sıfır, overflow ve carry dir. Örneğin ALU da bir toplama işlemi yapıldı ve sonuç 8 bitten yüksek çıktı yani overflow var, öyleyse overflow bayrağı 1 olur ve bu bilgi koşul kodu kaydedicisinde saklanır. Bu bilginin saklanma nedeni ise bir komut türü olan koşullu atlama komutları içindir. Örneğin overflow varsa program sayacını sıfırlayarak programı belli bir yerden başlat şeklinde koşullu çalışma modları ve atlamalar eklenebilir. Bu bilgiler sadece koşullu atlama modları için tutuluyor.

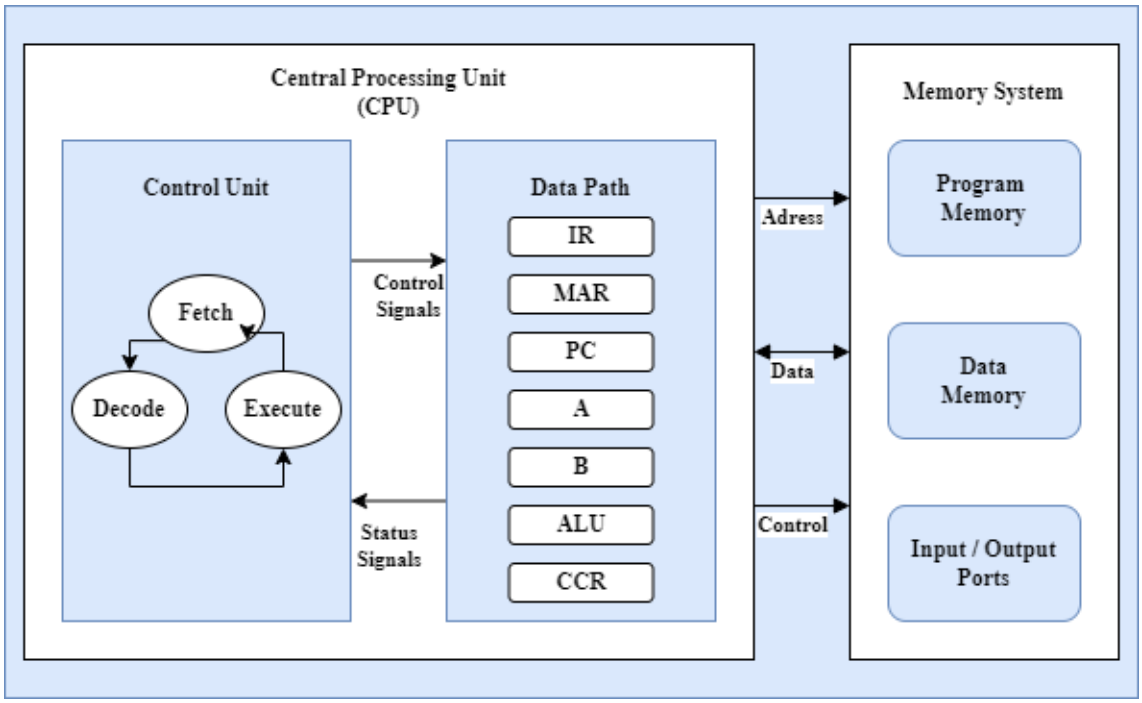
Aritmetik mantık ünitesi ise toplama, çıkarma, çarpma, bölme gibi matematiksel işlemleri AND, OR, NOT, SHIFT gibi mantık işlemlerini yapmak ile görevlidir.

Kontrol ünitesi ile veri yolu arasında ki iletişim ise şu şekilde gerçekleşir; Tüm iletişimi kontrol ünitesi yönetir. Ve durum makinasının o anki durumuna göre ihtiyacı olan işlemleri yaptırabilmek için veriyoluna kontrol sinyalleri gönderir. Örneğin ALU ya toplama işlemi yapması için emir verir veya program sayacının değerini arttır emrini verir.

Bu kontrol sinyali veri yolunda ki ilgili işlemleri aktifleştirerek oluşturulan sonucun yani verinin o anki durumunun tekrar kontrol ünitesine aktarılmasını sağlar. Bu şekilde düzenli ve kontrollü bir akış elde edilir. Veri yolu, Kontrol ünitesi ne derse onu yapar.

Giriş.3. Bellek Mimarisi

CPU dan sonra bellek yapısı Üst-Seviye tasarımımızı oluşturan iki ana altbloktan diğeridir.



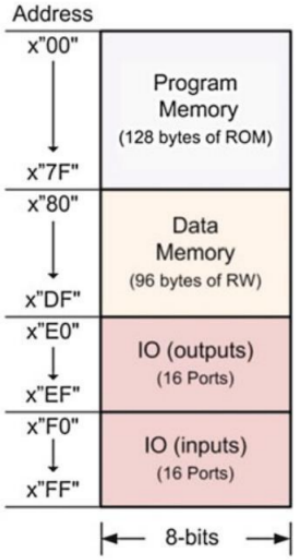
Şekil G.3. CPU-Bellek İletişim Mimarisi

CPU ile bellek arasında ki iletişim, adres, data ve kontrol sinyalleri ile sağlanıyor. Adres sinyali okuma yazma yapılacak bellek lokasyonunu, Data okunan/yazılan veriyi Kontrol sinyali ise belleğe yazma komutu olan "write-enable" sinyalini belirtmek amacı ile kullanılıyor.

Bellek sisteminde 4 ayrı yapı var. Program belleği, veri belleği, INPUT ve OUTPUT portları. Ancak CPU dan belleğe tek tek 4 farklı adres sinyali gitmiyor. Sadece 1 adet adres sinyali var.

4 Farklı bellek yapısı kullanmak yerine tek büyük bir bellek kullanılmış ve bu bellek 4 bölüme ayrılmış. Her bellek bölümünün büyüklüğüne göre program belleğinden başlanarak her birine adres aralıkları atanmış.

Burada en önemli yapılardan birisi program belleğidir, ilgili komutları çalıştırmak için bilgisayar tarafından işlenecek komutlar bu bellekte tutulur. Bu komutlar ilgili sırayla ROM da bulunur çünkü komutlar sıra sıra işlenir.



Şekil G.4. Bellek Yapısı

Program belleği Read-Only yani sadece okumaya izin veren yapılardır. İçerisine herhangi bir yazma işlemi yapılamaz. Bunun nedeni bilgisayar çalışırken bir kaza sonucu bu belleğin komut yazılı bir bölümünün üzerine başka bir verinin yazılmasını önlemektir. Bu özelliği nedeni ile program belleği genellikle ROM, EEPROM veya FLASH-ROM adı verilen yapılardan meydana gelir. ROM çiplerine üretim aşamasında istenilen veriler kazınır. Bu verilen bilgisayarın gücü kesilse bile silinmez ve üzerinde elektriksel olarak başka kod yüklenemez.

Programlamaya optimize edilmiş işlemcilerde genellikle bu programlama işlemi sorunsuz ve kolay bir şekilde yapılabilir ancak yapmış olduğum işlemci daha çok BIOS mantalitesinde çalışmaktadır. BIOS, bilgisayarın çalışması için gereken tüm komutları, ilgili yapıları iletir. Yapmış olduğum işlemcinin yapısı da bu şekildedir.

Veri belleği, programın çalışması sırasında oluşan verilerin kaydedildiği bellektir. RAM olarak düşünülebilir, bilgisayar çalıştığı sürece istenilen veriler kaydedilip okunabilir. Ancak güç kesildiğinde RAM deki veriler kaybolacaktır. Bu yapı da tam olarak aynı işlevi görmektedir.

INPUT, OUTPUT portları da dış dünyadan veri alabilmek ve veri iletebilmek için kullandığımız çevre birimlerdir. Bahsi geçen işlemci mimarisi ana hatları ile bu şekildedir.

Giriş.4. Nasıl Çalışır?

Donanımların önceden belirlenmiş birtakım komutları gerçekleştirmek üzerine tasarlanması o yapının mimarisini belirler. Bu komutlar 1 ve 0 lardan oluşan değerler olarak program belleğine kaydedilirler ve sıra sıra işlenerek bilgisayarda işlem yaptırılmış olur. Bu komutlara yazılım denir. Her yazılım sadece donanımın izin verdiği komutlar sınırında yazılabilir.

Bu komutlar görece basit komulardır. Mesela bir veriyi bir Kaydediciden başka bir Kaydediciye taşımak ya da 2 sayıyı aritmetik işleme sokup kaydetmek gibi. Bilgisayarda bu komutlar Binary sayılar yani 1 ve 0 lardan oluşan özel değerler olarak program hafızasına kaydedilirler. Ve bu komular program belleğinden sıra sıra işlenerek istenilen işlem bilgisayara yaptırılmış olur. Bu komutlara yazılım denir.

Yazılımın sıra sıra işlenerek makine komutlarına dönüştürülmesi işini ise işlemci yapar. Komutların doğru şekilde yazılması ve doğru sıra ile verilmesine yazılım geliştirme denir. Her yazılım sadece donanımın izin verdiği komutlar çerçevesinde tasarlanabilir.

İşlemcimiz 8-bit, dolayısıyla komutlar da dahil olmak üzere işlenecek ve kaydedilecek her veri ancak 8-bit uzunluğunda olabilir. Buna göre bellekte ki adresleme şu şekilde olmalı; - 0-127 adresleri program belleğini - 128-223 adrees aralığı veri belleğini - 224-239 adres aralığı OUTPUT - ve 240-255 arası adres INPUT Örnek verecek olursak CPU, veri belleğine ulaşmak isterse x”80” ile x”DF” aralığında adres göndermeli. INPUT aralığına ulaşmak isterse x”F0” ile x”FF” aralığında adres göndermelidir.

Mesela Python da kod yazıp derleyince kullandığımız derleyici bunu makine diline çevirir yani mesela Python da “a+b” ifadesi Compile edilip işlemciye girdikten sonra aslında 110101010001011.. gibi 64 bitlik bir makine kodudur.

Çünkü modern işlemcilerin çoğu 64 bit desteklemekte. Bu binary sayı bir komutu ifade eder. İşlemci, komutu ifade eden bu 64 bitlik sayıyı çeşitli bölümlere ayırır ve ilgili kısımlarda ilgili veriyi okuyarak işlemi gerçekleştirir.

64 bit içerisinde işlemin toplama olduğunu belirten kısım, a ve b sayılarının

değerlerinin yazılı olduğu kısım ve sonucun nereye kaydedileceği gibi bilgiler saklıdır. Olan şey derleyicinin bunu makine diline çevirmesi ve işlemcinin anlayacağı bir formata sokması. İşlemcinin de kendisine tanımlı olan komutları çalıştırmasıdır.

Bu çalışma sonucu ortaya konulan işlemciye ait komutlar basit bir yapıya sahiptir. "Opcode" ve "Operand" olmak üzere iki kısımdan oluşur. Opcode, komutun kimliğidir, ve her komut için özel bir Opcode vardır. Örneğin toplama işleminin Opcode kodu x"42" sayısıdır. Veya veri yükleme komutunun Opcode u x"86" sayısıdır. Sadece bu kısıma bakarak komutun ne olduğu anlaşılır.

"Operand" ise komutun işlenmesi için gerekli olan veridir, misal belleğe bir veri kaydedilecek diyelim, "Operand" kayıt için gerekli olan adres bilgisini taşıyacaktır. Veya A Kaydedicisine bir sayı yüklenecekse bu sayı Operand kısmında taşınır.

Program belleğinde ki komutlar bu şekildedir, Opcode ve ona ait Operand kısmı 8 bit şeklinde alt alta sıralanırlar, bazı komutların Operandı olmayabilir ama Opcode u olmak zorundadır, Örneğin toplama işlemi A ve B Kaydedicilerindeki sayıları toplayıp A Kaydedicisine geri yazar. Bu nedenle ekstra bir Operand bilgisine ihtiyaç yoktur. Bellekte ki her komutun bir adres değeri vardır. Bellek yapısında bu değer 0-127 arası olarak verilmişti. Bu komutlar 0. adresten başlayarak en fazla 127. adreste olmak üzere tek tek işlenecekler. Bu adres değerini de program Counter parametresi tutar.

Program Counter 0 dan başlayarak program belleğini okuyup CPU veri yolunda ki komut kaydedicisine okunan komutun yazılmasını sağlar. Böylece program sayacının gösterdiği komut CPU ya aktarılıp işlenmiş olur. Daha sonra program sayacının değeri 1 arttırılarak yeni komuta geçilir. Bu şekilde program belleğindeki bütün komutlar sırası ile işlenip yazılımın amaçladığı işlem yapılmış olur.

Program sayacının gösterdiği komutun bellekten CPU ya getirilip uygulanmasına kadar 3 adet işlem sıra ile gerçekleşir.

1- Fetch

2- Decode

3- Execute

Fetch : Gidip getirmek demektir. Komutun program belleğinden alınıp CPU veri yolunda ki komut belleğinde konulmasıdır.

Decode : Anlamını çözmek, algılamak demektir. Opcode değerine bakarak komutun ne olduğunun CPU tarafından anlaşılmasıdır.

Execute : ise uygulamak demektir. Komutun amacına göre gerekli matematiksel, mantıksal işlemlerin gerçekleştirilmesidir.

Program belleğinde 0. adreste komutun Opcode'u, 1. adreste Operand'ı bulunuyor. Program başlayınca program sayacı 0 değeri Opcode'u okur. Komut Kaydedicisine aktarır. Bu Fetch adımıdır. CPU da bulunan kontrol ünitesi bunun bir komut olduğunu anlar ve ihtiyacı olan komutu okumak için program sayacını 1 arttırır. Bu Decode adımıdır. Komutun ne olduğunu CPU tarafından algılandığı an Program sayacı bir sonraki saat darbesinde gerekli değeri okur ve işlemi gerçekleştirir. Bu ise execute adımıdır.

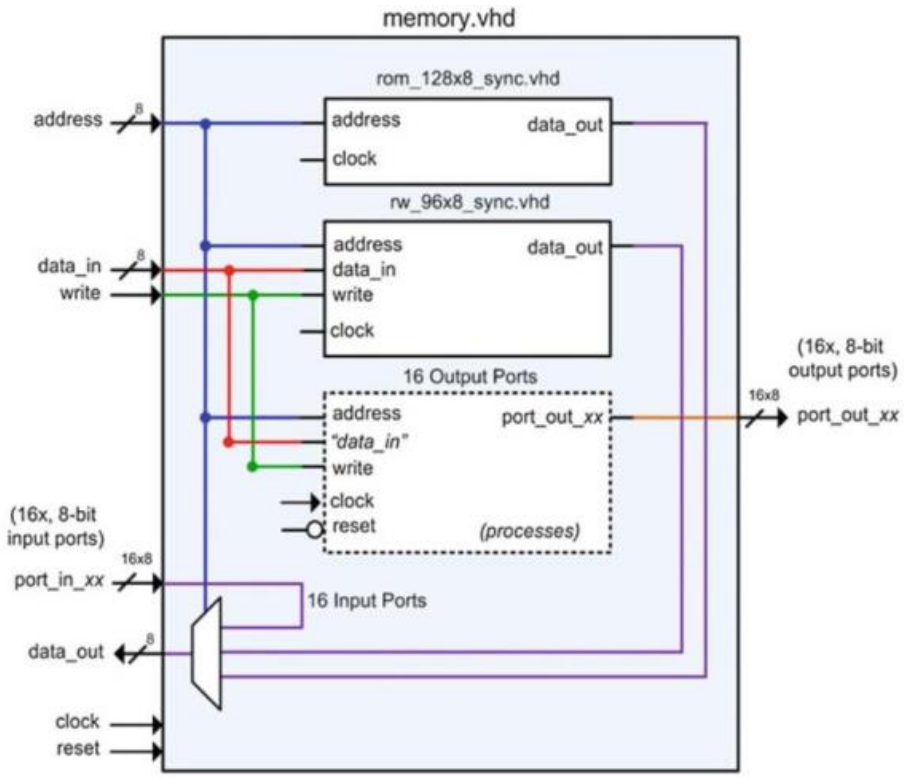
İşlemciler de basit komutlar kullanarak program belleğinin alanını küçültebiliriz. Ancak bu durumda kompleks bir algoritmayı gerçeklemek için daha fazla sayıda komuta ihtiyaç duyarız.

Çok özelleşmiş komutları olan işlemciler, bir algoritmayı belki 5-6 komutta gerçekleyebilir ancak bu karmaşık komut yapıları program belleğinin alanını ciddi boyutta arttırır. Basit ve minimal yapıda Opcode, Operand ikilisinin bulunduğu bit pozisyonları her komut için aynı ve sabit olan işlemci mimarilerine RISC (Reduced Instruction Set Computer), karmaşık komutların bulunduğu ve her komuta özel Opcode, Operand, bit pozisyonları, uzunlukları değişken olan işlemci mimarilerine ise CISC (Complex Instruction Set Computer) adı verilir. Bizim mimarimiz RISC mimarisi şeklinde.

RISC in sahip olduğu sabit, düzenli yapı ve minimal kontrol mekanizması, işlemcilerin daha optimize, daha hızlı çalışmasını sağladığı konusunda çoğu araştırma hemfikir. Modern işlemci mimarileride aynı şekilde RISC mimarisi ile tasarlanmaktadır.

Giriş.5. Sonuç

Tasarlamış olduğum İşlemci hiyerarşik bir tasarım olup en alt bloktan başlayıp yukarı doğru çıkarak, alt blokların birleştirilmesi ile tek bir Üst-Seviye tasarım elde edilmiştir.



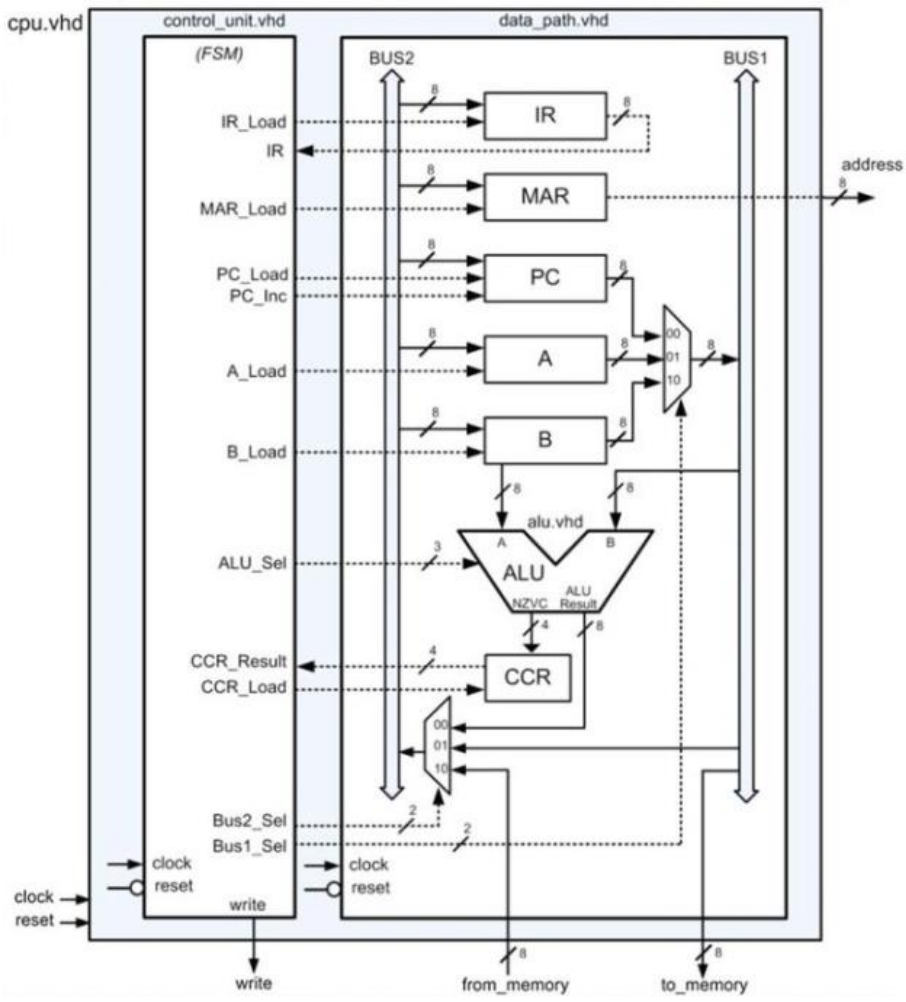
Şekil G.5. Bellek VHDL Tasarım Diyagramı

Memory bloğunda alt blok olarak program belleği, veri belleği ve OUTPUT portlarını tanımladım. Her bellek bloğuna adres bilgisi gelecek. Çünkü adres bilgisi olmadan bellekten veri okunamaz. Sadece veri belleğine ve OUTPUT portlarına "data-in" ve "write" sinyalleri giriyor. Çünkü sadece bu bellek elemanlarına veri yazılabilir.

Program belleği (ROM) sadece okumaya izin verir. Veri girişine ve yazma sinyaline ihtiyacı yoktur. Üst-Seviye tasarım da bir MUX görülüyor. MUX çıkışında ki "data-out" sinyali, bellekten okunan verinin CPU'ya gönderildiği bağlantı sinyalidir. MUX portları sırasıyla 16 adet 8 bitlik INPUT sinyali, program belleği çıkışı ve veri belleği çıkışıdır.

Bir diğer portlar; Clock ve Reset. İşlemciler tamamen senkron yapılar olduğundan

tüm tasarım senkron yapıdadır, bu sebeple bellek okuma yazmaları da tamamen clock geçişlerine duyarlı şekilde inşa edilecek. Bu sayede veriyi ne zaman bellekten çekeceğimizi tam olarak hesaplayabiliriz ve durum makinasını da bu zamanlamaya göre kurabiliriz.



Şekil G.6. CPU VHDL Tasarım Diyagramı

CPU, Kontrol birimi ve veri yolu olarak 2 alt bloktan oluşuyor. Kontrol ünitesi, işlemcinin tüm çalışma adımlarını düzenler. İçerisinde bir sonlu durum makinası vardır ve komutların bellekten alınmasını, decode edilmesini daha sonra da ilgili komutun çalışma moduna geçilmesini bu durum makinesi kontrol eder.

Yani fatch, decode ve execute dan sorumludur. Kontrol ünitesi, bellekler ile veri alışverişini veriyolu aracılığı ile yapar.

Kontrol ünitesi, komut register ındaki değere göre işlemlerini düzenler. MAR, bellekten okuma yazma yapmak için gerekli olan adres bilgisini belleğe iletir.

Program sayacı, bellekten hangi komutun okunacağını bildiren adres bilgisidir. Program başlarken 0 dan başlar ve branch komutları hariç değeri her komuttan sonra 1 atar. A ve B register ları Operandların tutulduğu veri registerlarıdır. CCR ise ALU sonucunda ortaya çıkan sayının hangi bayrağı etkileyeceği bilgilerini tutan Kaydedicidir. Bu bilgiler koşullu atlama komutları için gereklidir. Veri yolunun bir alt bloğu olarak ALU yapısı karşımıza çıkıyor. ALU nun bir girişi doğrudan B Kaydedicisine bağlı. Diğeri de BUS1 adlı yapıya bağlı. ALU içerisinde toplama çıkarma, AND, OR, NOT gibi matematiksel işlemler gerçekleştirilir. BUS1 ve BUS2, kaydedicileri birbirine bağlamak için kullanılan birer iletişim kanalıdır. Her Kaydedici birbiri ile bir şekilde haberleşmek ve veri alışverişi yapmak zorunda. Bu yapıların hepsini portlar aracılığı ile birbirine bağlarsak hem alanı çok büyütürüz. Hem de içeride çok fazla sayıda ara bağlantı kullanmış oluruz. Veri yolu içinde ki trafiği basitleştirmek için BUS1 ve BUS2 kablolarını kullanıyoruz. BUS1 e program sayacı, A ve B registerları bağlı. Hangi verinin BUS1 e sürüleceği bilgisi kontrol ünitesinden gönderilen "BUS1-select" seçim sinyali ile sağlanıyor. Ve ilgili MUX tan istenen veri BUSx'e sürülmüş oluyor.

BUS1, veriyi CPU dan çıkıp belleğe iletebilir veya BUS2 ye bağlanarak yine veri yolu Kaydedicilerine gidebilir. Yani Üst-Seviye tasarımda gördüğümüz CPU dan belleğe giden "data-in" sinyali aslında BUS1 in "to-memory" çıkışıdır.

BUS2 portları sırasıyla ALU sonucu, BUS1 ve Bellekten okunan veriyi ifade eden "from-memory" sinyalleridir. Hangi verinin veri yoluna sürüleceği yine kontrol ünitesi tarafından belirlenir. BUS1, CPU Kaydedicilerinde ki verileri belleğe iletmek ile görevlidir. BUS2 ise temel olarak bellekten okunan verinin CPU ya iletilmesinde görevlidir.

Söz konusu bağlantılar VHDL donanım tanımlama dili ile yazılarak Vivado programı üzerinde simüle edilmiştir. Burada bahsedilen hiyerarşik tasarımı aynı şekilde VHDL ile kodlama yaparken uyguladım.

Kodlamaya en alt blok olan ALU biriminden başladım. Sonrasında ALU biriminin bağlı olduğu Veri Yolu ve Kontrol Birimi tasarımlarını ve bu alt blokların hepsini barındıran CPU tasarımı kodladım.

Üst-Seviye tasarımı ifade eden iki alt blok CPU ve Bellekti. Dolayısıyla CPU tasarımından sonra RAM, ROM ve INPUT/OUTPUT portlarının tasarımını yaparak Bellek bloğunu da bitirmiş oldum.

En son bütün blokları Üst-Seviye tasarımı altında birleştirerek her bir alt bloğu bir üst blokta komponent olarak tanımladım ve Vivado programı üzerinde davranışsal olarak simüle ettim.

Simülasyon sırasında yürütülen program ve giriş çıkış değerleri dış dünyadan verilemediğinden bir test programı yazdım. Yazılan test programı içerisinde işlemciden gerçekleştirmesini istediğimiz fonksiyonlarda kullanılan giriş ve çıkış değerlerinin sanal olarak atanmasını sağlar. Gerçekleştirilmesi istenen fonksiyonlar ise program belleği (ROM) içerisine yazılır.

Her bir sinyal, her bir sinyalin geçen süre boyunca yükselen ve düşen kenarlarda aldığı değerlerden Vivado programında ki Waveform üzerinde incelenir. Bu aşama artık VHDL ile yapılan işlemci tasarımının test ve doğrulama aşamasıdır.

Gerekli test ve doğrulamalar yapıldıktan sonra tasarım fiziksel implementasyon testleri için sentez işlemine sokulur. Sentez işleminin ardından belirlediğimiz saat frekansları, geçen süre ve gerçek dünya parametrelerine göre bize işlemcinin mantık devreleri ile oluşturulmuş hali ve gerçek koşullar altında vermesi beklenen olası tepkilerin tahminleri statik zaman analizi adı altında bir rapor halinde sunulur.

8-bit işlemci tasarımı bu şekilde yapılmış olup bir FPGA mikroişlemcisine yüklenilebilir ve kullanılabilir haldedir.

Literatür Özeti.1. Donanım Tanımlama Dillerinin Temeli

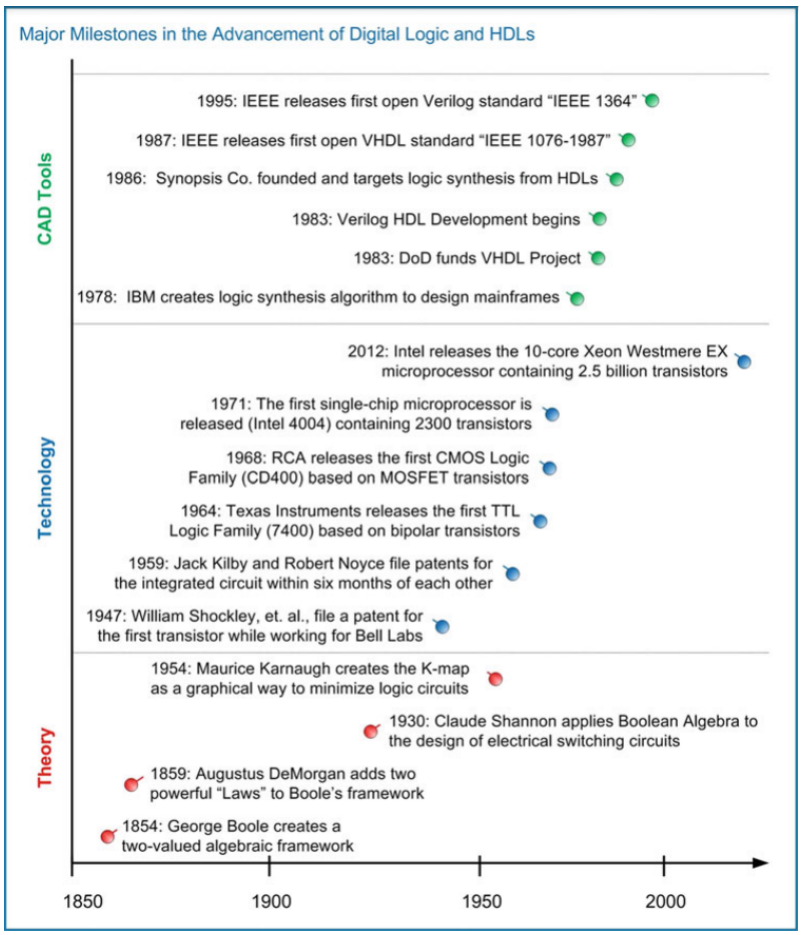
Mantık tasarımı hakkında belirgin olan birkaç gözlem var. İlk olarak, mantık devresinin boyutu, elle tasarlamanın zor olduğu noktaya hızla ölçeklenebilir. İkinci olarak, bir devrenin nasıl çalıştığına (örneğin, bir doğruluk tablosu) ilişkin üst düzey bir açıklamadan gerçek devre ile uygulanmaya hazır bir forma (örneğin, en aza indirilmiş bir mantık şeması) geçiş süreci basit ve iyi tanımlanmıştır.

Metin tabanlı bir dil kullanarak dijital devreleri tanımlamanın bir yolu olarak

donanım tanımlama dilleri (HDL'ler) kullanılır. HDL'ler, çok büyük tasarımlarda pratik olmayan şemalara ihtiyaç duymadan büyük dijital sistemleri tanımlamak için bir araç sağlar. HDL'ler, farklı soyutlama seviyelerinde mantık simülasyonunu desteklemek için gelişmiştir. Bu, tasarımcılara büyük sistemlerin işlevselliğini yüksek bir soyutlama düzeyinde tasarlamaya ve doğrulamaya başlama ve devre uygulamasının ayrıntılarını tasarım döngüsünün ilerisine erteleme yeteneği sağlar. Bu, farklı mantık aileleri arasında ölçeklenebilir bir yukarıdan aşağıya tasarım yaklaşımı sağlar.

Günümüzde kullanılan iki baskın donanım tanımlama dili vardır. Bunlar VHDL ve Verilog'dur. VHDL, çok yüksek hızlı tümleşik devre donanım tanımlama dili anlamına gelir. Verilog bir kısaltma değil, ticari bir isimdir. Bu iki HDL'nin kullanımı, dijital tasarım endüstrisinde neredeyse eşit olarak bölünmüştür.

Literatür Özeti.2. Donanım Tanımlama Dillerinin Tarihi



Şekil G.7. HDL Kronolojik Sınıflandırması

Entegre devrenin icadı en yaygın olarak 1959'da 6 ay arayla aynı temel kavramın farklı varyasyonları üzerinde patent başvurusunda bulunan iki kişiye atfedilir. Jack Kilby, 1959 yılının Şubat ayında "Minyatürize Edilmiş Devre" başlıklı ilk entegre devre patentini aldı. Elektronik Devreler", Texas Instruments için çalışırken. Robert Noyce, Temmuz 1959'da Fairchild Semiconductor adlı bir şirkette "Yarı İletken Cihaz ve Kurşun Yapısı" başlıklı entegre devre üzerine patent başvurusunda bulunan ikinci kişiydi. Kilby, icadı nedeniyle 2000 yılında Nobel Fizik Ödülü'nü kazandı, Noyce ise 1968'de Gordon Moore ile Intel Corporation'ın kurucu ortağı oldu. 1971'de Intel, entegre devre teknolojisi kullanan ilk tek çipli mikroişlemci Intel 4004'ü tanıttı. Bu mikroişlemci 2300 transistör içeriyordu. Bu buluş dizisi, Silikon Vadisi'nin büyümesinin arkasındaki itici güç olan yarı iletken endüstrisini başlattı ve modern dünyanın her yönünü etkileyen teknolojide 50 yıllık eşi görülmemiş ilerlemeye yol açtı. Intel'in kurucu ortağı Gordon Moore, 1965'te bir entegre devredeki transistör sayısının her 2 yılda bir ikiye katlanacağını tahmin etmişti. (Şimdi Moore Yasası olarak bilinir.)

Bir entegre devre üzerindeki transistör sayısı arttıkça, tasarımın boyutu ve uygulanabilecek işlevsellik de artar. İlk mikroişlemci 1971'de icat edildiğinde, CAD araçlarının kapasitesi hızla arttı ve daha büyük tasarımların gerçekleştirilmesine olanak sağladı. Daha büyük tasarımlar, CAD araçlarının daha da karmaşık hale gelmesini sağladı ve karşılığında daha da büyük tasarımlar sağladı.

Endüstrinin karşılaştığı zorluklardan biri, daha büyük sistemlerin karmaşık davranışını belgelemenin bir yoluydu. Büyük dijital tasarımları belgelemek için şemaların kullanılması çok hantal ve tasarımcı dışında herkes tarafından anlaşılması zor hale geldi. Davranışın kelime açıklamalarını anlamak daha kolaydı, ancak bu belgeleme biçimi bile ortaya çıkan tasarımların boyutu için etkili olamayacak kadar hacimli hale geldi.

1983'te ABD Savunma Bakanlığı, tüm tedarikçilerinde kullanılabilecek dijital sistemlerin davranışını belgeleyecek bir araç oluşturmak için bir programa sponsor oldu.

Savunma Bakanlığı'na sağlanan uygulamaya özel tümleşik devrelerin (ASIC)

işlevselliği için yeterli belge bulunmaması nedeniyle. ASIC yaşam döngülerinin sonuna geleceğinden ve değiştirilmeleri gerektiğinden, bu belge eksikliği kritik bir sorun haline geliyordu. Standartlaştırılmış bir dokümantasyon yaklaşımının olmaması nedeniyle, tedarikçiler, modası geçmiş olanlara eşdeğer parçaları yeniden üretmekte zorlandılar. Savunma Bakanlığı, hem arabirim (yani girdiler ve çıktılar) hem de dijital sistemlerin davranışı hakkında ayrıntılı bilgi sağlayan standartlaştırılmış bir belgeleme aracı geliştirmek üzere üç şirketle (Texas Instruments, IBM, Intermetrics) sözleşme yaptı. Yeni araç, programlama diline benzer bir biçimde uygulanacaktı. Simülasyon yeteneğinin, maksimum esneklik sağlamak için birden fazla soyutlama seviyesini kapsaması istendi. 1985 yılında, bu aracın VHDL adı verilen ilk versiyonu piyasaya sürüldü. Endüstri genelinde kullanım tutarlılığının sağlanması için VHDL, standardizasyon için Elektrik ve Elektronik Mühendisleri Enstitüsü'ne (IEEE) devredildi. IEEE, çok çeşitli açık teknoloji standartlarını tanımlayan profesyonel bir dernektir. 1987'de IEEE, VHDL'nin ilk endüstri standardı sürümünü yayınladı. İlk versiyondan gelen geri bildirimler, standardın 1993 yılında daha büyük bir revizyonuyla sonuçlandı.

1993 sürümünde birçok küçük revizyon yapılmış olsa da, 1076–1993 standardı bugün kullanımda olan VHDL işlevselliğinin büyük çoğunluğunu içermektedir. En yeni VHDL standardı IEEE 1076–2008'dir.

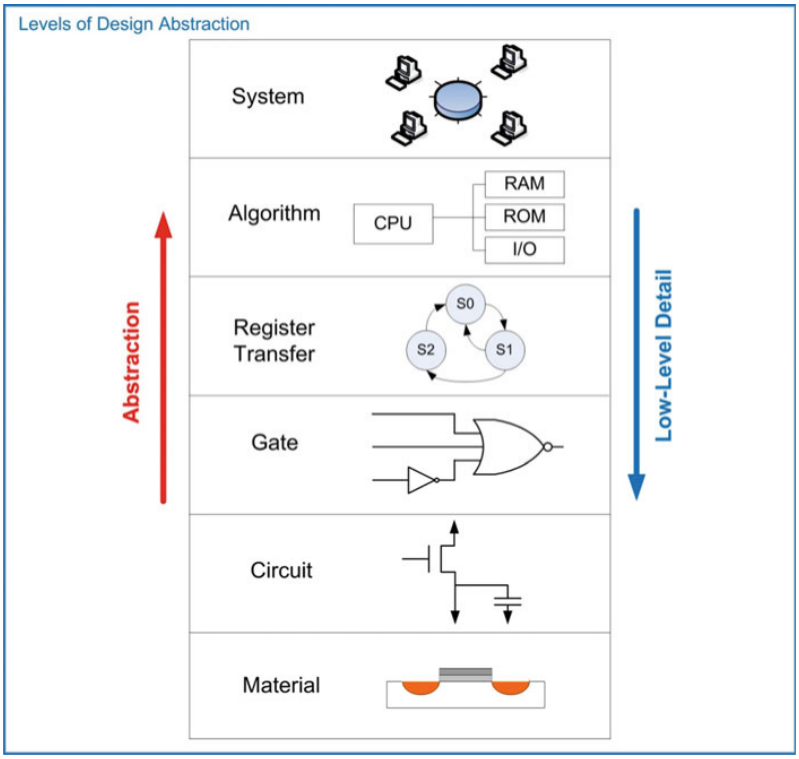
Ayrıca 1983 yılında Verilog HDL, Automated Integrated Design Systems tarafından bir mantık simülasyon dili olarak geliştirilmiştir. Verilog'un gelişimi VHDL projesinden tamamen bağımsız olarak gerçekleşti. Otomatik Entegre Tasarım Sistemleri (1985'te Ağ Geçidi Tasarım Otomasyonu olarak yeniden adlandırıldı), 1990 yılında CAD aracı satıcısı Cadence Design Systems tarafından satın alındı. Açık VHDL standardının hızla benimsenmesine yanıt olarak, Cadence, rekabetçi kalabilmek için Verilog HDL'yi halka açık hale getirdi. . IEEE bir kez daha bu HDL için açık standardı geliştirdi ve 1995'te IEEE 1364 başlıklı Verilog standardını yayınladı. Otomatik mantık sentezini gerçekleştirmek için CAD araçlarının geliştirilmesi, IBM'in bir dizi pratik sentez motoru geliştirmeye başladığı 1970'lere kadar uzanabilir. ana bilgisayarlarının tasarımında kullanılan; ancak, mantık sentezindeki ana ilerleme, 1986'da Synopsis adlı bir şirketin kurulmasıyla geldi.

Synopsis, doğrudan HDL'lerden mantık sentezine odaklanan ilk şirketti. Bu büyük bir katkıydı çünkü tasarımcılar dijital sistemlerini tanımlamak ve simüle etmek için zaten HDL'leri kullanıyorlardı ve şimdi mantık sentezi aynı tasarım akışına entegre edildi. Son derece soyut işlevsel açıklamaları sentezlemenin karmaşıklığı nedeniyle, başlangıçta yalnızca baştan sona ayrıntılı olarak hazırlanmış daha düşük soyutlama seviyeleri sentezlenebildi. CAD aracı yeteneği geliştikçe, daha yüksek soyutlama düzeylerinin sentezi mümkün hale geldi.

HDL'ler orijinal olarak doküMantasyon ve davranışsal simülasyon için tasarlanmıştır.

Literatür Özeti.3. Donanım Tanımlama Dilleri Nedir? Nasıl Çalışır?

HDL'ler başlangıçta davranışı birden fazla soyutlama düzeyinde modelleyebilmek için tanımlandı. Soyutlama, mühendislik tasarımında önemli bir kavramdır çünkü uygulama detayları ile sistemlerin nasıl çalışacağını belirlememizi sağlar. Her MOSFET için ayrıntılı modeller kullanılarak tam bir bilgisayar sistemi simüle edilirse, tamamlanması pratik olmayan bir zaman alacaktır.

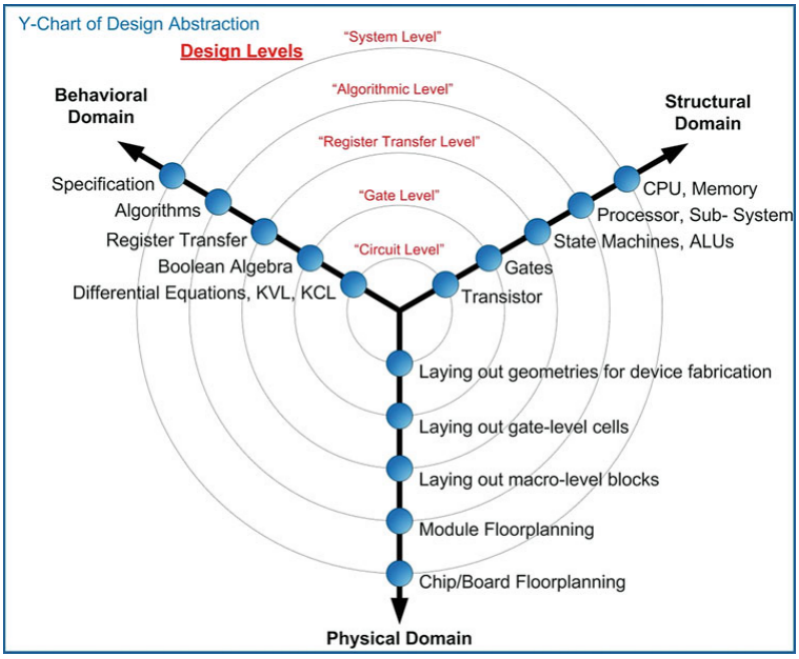


Şekil G.8. HDL Soyutlama Seviyeleri

En yüksek soyutlama seviyesi sistem seviyesidir. Bu seviyede, bir sistemin davranışı bir dizi geniş spesifikasyon belirtilerek tanımlanır. Bu düzeyde bir tasarım örneği, "bilgisayar sistemi, çift kesinlikli veriler üzerinde saniyede 10 tera kayan nokta işlemi (10 TFLOPS) gerçekleştirecek ve 100 watt'tan fazla güç tüketmeyecek" gibi bir belirtimdir. Sistem seviyesinden bir alt seviye algoritmik seviyedir. Bu seviyede, spesifikasyonlar, her biri birincil görevin bir bölümünü gerçekleştirecek ilişkili davranışlara sahip alt sistemlere bölünmeye başlar. Bu düzeyde, örnek bilgisayar özellikleri, hesaplama ve rastgele erişim belleği (RAM) gerçekleştirmek için merkezi işlem birimi (CPU) gibi alt sistemlere bölünebilir. Bu çalışmada gerçekleştirilmiş olan 8-bit işlemci modeli de bu şekilde tasarlanmıştır.

Algoritmik düzeyden bir alt düzey, kayıt aktarım düzeyidir (RTL). Bu seviyede, verilerin sistem girdilerine dayalı olarak nasıl manipüle edildiğine ek olarak, verilerin alt sistemler arasında ve içinde nasıl taşındığına ilişkin ayrıntılar açıklanır. RTL seviyesinden bir seviye aşağı kapı seviyesidir. Bu seviyede tasarım, temel kapılar ve kaydediciler kullanılarak tanımlanır. Kapı seviyesi, esasen, yukarıdaki soyutlama seviyelerinden işlevselliği uygulayacak bileşenleri ve bağlantıları içeren bir şematiktir. Kapı seviyesinden bir seviye aşağısı devre seviyesidir. Devre seviyesi, transistörler, teller ve dirençler ve kapasitörler gibi diğer elektrikli bileşenleri kullanan temel geçitlerin ve kayıtların çalışmasını tanımlar. Son olarak, tasarım soyutlamasının en düşük seviyesi malzeme seviyesidir. Bu seviye, devre seviyesinden transistörleri, cihazları ve telleri uygulamak için farklı malzemelerin nasıl birleştirildiğini ve şekillendirildiğini açıklar. HDL'ler, malzeme seviyesi hariç tüm bu seviyelerdeki davranışı modellemek için tasarlanmıştır. MOSFET'ler gibi devre düzeyinde davranışı ideal anahtarlar ve yukarı çekme/aşağı çekme dirençleri olarak modelleme yeteneği olsa da, HDL'ler tipik olarak devre düzeyinde kullanılmaz. Tasarım soyutlamasının bir başka grafiksel tasviri, Gajski ve Kuhn'un Y grafiği olarak bilinir. Bir Y grafiği, üç farklı tasarım alanında soyutlamayı gösterir: davranışsal, yapısal ve fiziksel. Bu tasarım alanlarının her biri, soyutlama seviyeleri (yani sistem, algoritma, RTL, kapı ve devre) içerir.

Örnek bir Y grafiği şurada gösterilmiştir:



Şekil G.9. Y Grafiği Örneği

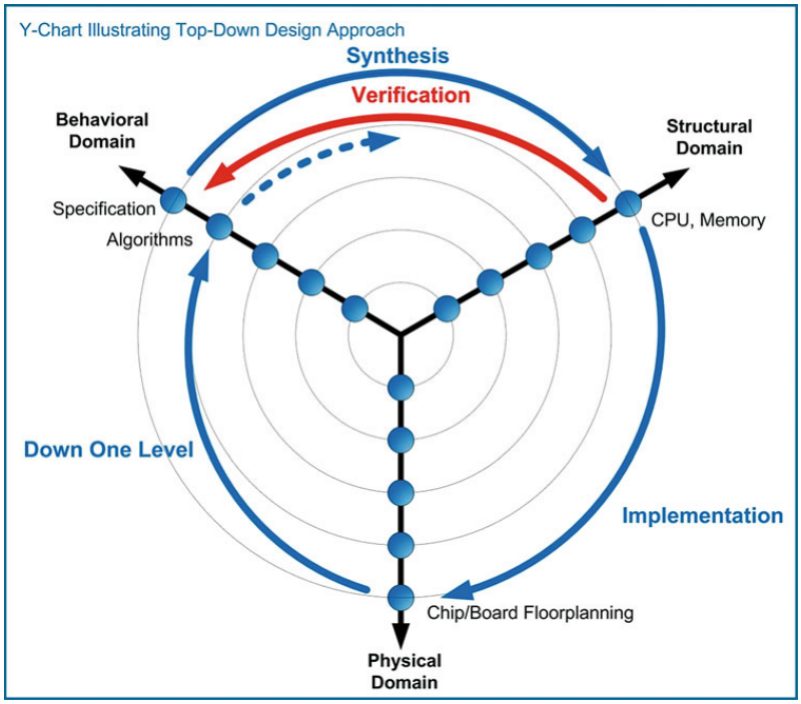
Bir Y grafiği ayrıca farklı tasarım alanlarının soyutlama düzeylerinin birbiriyle nasıl ilişkili olduğunu da gösterir. Yukarıdan aşağıya bir tasarım akışı, saat yönünde içe doğru spiral çizerek bir Y grafiğinde görselleştirilebilir. Davranışsal alandan yapısal alana geçiş sentez sürecidir. Sentez yapıldığında, ortaya çıkan sistem önceki davranışsal açıklama ile karşılaştırılmalıdır. Bu kontrole doğrulama denir. Yapısal açıklamaya karşılık gelen fiziksel devre oluşturma işlemine uygulama denir. Spiral, devre elemanlarını (transistörler, teller, vb.) temsil eden geometrilerin silikonda üretilmeye hazır olduğu bir düzeyde tasarım uygulanana kadar soyutlama seviyelerinde devam eder.

Y grafiğinde içe doğru bir spiral olarak gösterilen yukarıdan aşağıya tasarım sürecini göstermektedir.

Y grafiği, büyük dijital sistemler için resmi bir yaklaşımı temsil eder.

Mühendis ekipleri tarafından tasarlanan büyük sistemler için, uygulama daha düşük soyutlama seviyelerinde gerçekleştirildiğinden, potansiyel olarak maliyetli tasarım hatalarını ortadan kaldırmak için resmi, yukarıdan aşağıya bir tasarım sürecinin izlenmesi çok önemlidir.

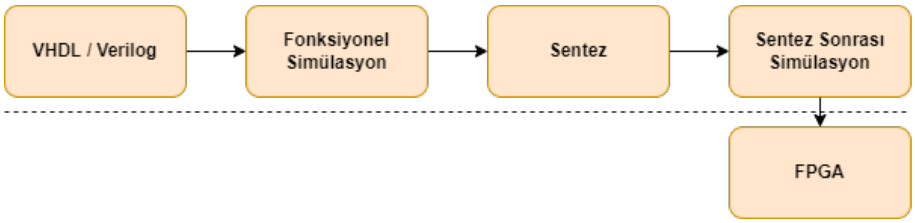
Aşağıda Y Grafiği üzerinde spiral yukarıdan aşağıya tasarım akışı görülmektedir.



Şekil G.10. Y Grafiği - Yukarıdan Aşağıya Tasarım Süreci

Literatür Özeti.4. Sonuç ve Benim İzlenimim

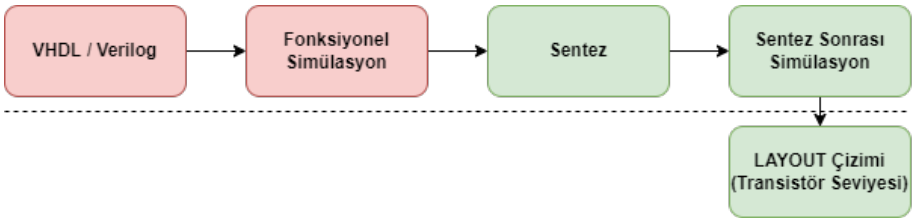
Dijital donanım tasarımı sırasında FPGA kartına ihtiyacımız yok. FPGA tasarım akışında ilk adımda VHDL / Verilog ile sentezlenebilir bir donanım mimarisi kuruyoruz, daha sonra simülasyon adımı geliyor.



Şekil G.11. FPGA Akışı

İki Tür simülasyon vardır, ilki Fonksiyonel simülasyon, bunu sentezden önce yaparız, tasarım sentezlenemeyen bir devreyi ifade etse bile algoritma davranışsal olarak simüle edilebilir. Amaç algoritmik işlemlerin doğruluğunu görmektir. İkincisi Sentez sonrası simülasyon, bu simülasyonda artık VHDL / Verilog kodu dikkate alınmaz, yazdığımız kodun sentezi sonrasında bir devre yapısı ortaya çıkartılıyor, bu aslında bizim tasarımımıza ve seçilen FPGA modeline göre Vivadonun oluşturduğu bir devre şemasıdır. Sentez sonrası simülasyon ile devre yapısı doğru kurulmuş mu

ve hala bizim istediğimiz gibi çalışıyor mu diye kontrol etmiş oluyoruz. Sonrasında tasarımı FPGA e programlarken aslında yapılan şey sentezlenmiş ve doğrulanmış bu fiziksel yapının FPGA içindeki serbest devre elemanları ile kurulmasıdır, yaptığımız sentez sonrası simülasyonlar ile FPGA de çalışan devre arasında fark yaratabilecek tek şey gerçek dünyadaki materyallerden, sıcaklıktan, basınçtan, bağlantı uzunluğu varyasyonlarından kaynaklanabilecek ufak gecikmeler, zaman farklılıkları olabilir. Bunun haricinde işlem simülasyonda ne ise FPGA de odur, zaten bizim FPGA içerisine bir müdahalemiz olamaz, müdahale edebileceğimiz tek şey VHDL / Verilog kodundaki tasarımdır. Yaptığımız tasarımı FPGA kartına programladıktan sonra FPGA de göreceğimiz işlemler tamamen sentez sonrası simülasyonda göreceğimiz işlemlerin aynısı olacaktır. Hız, alan, güç performansı da sentez analizlerinde gördüğümüz sonuçlara oldukça yakın değerler olacaktır. Diyagramda ki çizgi, donanım tasarımında front-end ve back-end taraflarına ait kısımları ayırıyor, front-end daha çok algoritmayı ve mimariyi kurmakla ilgilenirken, back-end tamamen sentezlenmiş tasarımın fiziksel yapısı ile ilgilenir, back-end grubu tasarımın ne olduğu ve ne yaptığı ile ilgilenmez.



Şekil G.12. ASIC Tasarım Akışı

ASIC, Fonksiyonel Test ve doğrulamadan hemen sonra yani Sentezden önce FPGA den ayrılır. Vivado sadece Xilinx FPGA ler için sentez yapabilir, ASIC tasarımların sentezi SYNOPSYS adı verilen başka bir program tarafından yapılır. SYNOPSYS tasarım aracı LINUX işletim sisteminde çalışır ve lisansları çok pahalıdır, bu lisanslar sadece seçili üniversiteler ve araştırma enstitülerinde bulunurlar. SYNOPSYS in yaptığı sentez de kendi kütüphanesinde bulunan özel devre elemanları ile bizim tasarımlarımızı kurar. Tasarımı direkt olarak FPGA kartına implement edip çalıştırabiliriz. Eğer tasarımımızın ASIC de çalışması hedefleniyorsa, saydığımız adımlardan sonra LAYOUT çizilmesi gerekli. Bu aslında basılacak devrenin yarı iletken ve metal düzlemindeki ifadesidir. Bu işlemi Back-end tasarımcılar yapar.

Çalışmanın Amacı

Günümüz modern teknolojilerinin fonksiyonlarını istendiği üzere yerine getirmesini sağlayan en önemli komponent mikroişlemcilerdir. Mikroişlemci, dijital bir bilgisayarın merkezi işlem biriminin işlevlerini yerine getirmek için gerekli aritmetik, mantık ve kontrol devresini içeren bir elektronik cihazdır.

Maalesef günümüz Türkiyesinde bu ürünün üretimi ve araştırma/geliştirme çalışmaları savunma sanayi haricinde yürütülmemektedir.

Bu projede hedeflenen öncelikle VHDL / Verilog gibi donanım tasarım dillerinin kullanılarak basit bir işlemci modeli ortaya konulması ve ortaya çıkan tasarımın Vivado gibi donanım tasarım programları ile test ve simülasyonlarının gerçekleştirilerek mikroişlemci çalışma mantığını tüm yönleri ile anlamaya çalışmaktır.

Sonrasında ise elde edilen sonuçlar ve bilgiler doğrultusunda ortaya basit bir mikroişlemci/grafik işlemci modelini ortaya koymaya çalışmaktır.

Bu çalışmada ele alınan, proje fikrinin gerçekleniminde kullanılması planlanan yöntemin amaç ve hedeflerinin belirtilmesi, çalışmanın kapsamının belirlenmesi ve literatürde yer alan kaynakların taranması ile projenin gerçeklenimine ilişkin yapılmış önceki çalışmalar hakkında ki bilgiler konsept proje çerçevesinde özetlenmesi ve bir araya getirilmesi üzerine yapılan çalışmalardır.

Bu çalışmanın gerçekleştirilmesi ile çözülmesi hedeflenen problemler ve çalışmanın motivasyon kaynakları şu şekilde sıralanabilir:

- Türkiye'de yarı-iletken teknolojileri şu anda üretilmemekte olup farklı amaçlar için kullanılan mikroişlemciler yurt dışından ithal edilmektedir, bunlar bilgisayarlarda kullanılan işlemciler, grafik işlemciler, savunma sanayinde bünyesinde üretilen elektronik harp sistemlerinin veya insansız araçlarının / otonom sistemlerin devre kartlarında kullanılan mikroişlemcilerdir. Ben ülkemde yarı-iletken teknolojilerinin üretilebilmesine ön ayak olarak dışa bağımlılığı azaltacak adımlar atmak istiyorum. Bu proje ile yarı-iletken üretimi

yapılamayacağına dair ön yargıları kıracağıma inanıyorum.

Aşağıda TESİD'e ait İthalatta (Gerçekleşen) Mevcut Durum ve İç Pazarda Yerli Payı Verileri Bulunmakta

Tablo 1. Elektronik Sektörünün, İmalat Sektörü İçindeki Payı

Oran Yıllar	2007	2008	2009	2010	2011	2012	2013	2014	2015
Girişim Sayısı	0,2	0,2	0,2	0,2	0,2	0,2	0,2	0,2	0,2
Çalışan Sayısı	1,2	1,0	0,9	0,9	0,9	0,8	0,8	0,9	0,9
Ciro	1,7	1,6	1,5	1,3	1,2	1,3	1,2	1,3	1,3
Üretim Değeri	1,7	1,6	1,6	1,3	1,3	1,3	1,2	1,3	1,4
Katma Değer	1,4	1,6	2,0	1,5	1,6	1,4	1,6	1,5	1,7
İhracat	2,6	1,8	2,4	2,4	2,4	1,9	2,0	1,7	1,7
İthalat	7,8	6,0	9,7	8,4	6,9	8,4	8,0	8,0	9,9
Tüketim	3,8	3,2	4,4	3,7	3,2	3,9	3,7	3,8	4,7
Kar	2,2	0,9	1,3	2,5	1,4	1,5	1,6	1,4	1,7

Bu çalışmalar ışığında belki üretilecek olan mikroişlemcilerimiz ithal ettiklerimizin yerini alabilir ve savunma sanayi, insansız hava araçları gibi üretiminde mikroişlemci kullanılan teknolojilerde tamamen yerlileşebiliriz. Kendi bilgisayar parçalarımızı en kötü ihtimalle belli fonksiyonları yerine getirmek üzere üretebiliriz. Proje, prototipleme aşaması dahil olmak üzere bu aşamaya kadar basit bir model üzerinde belli aritmatiksel işlemleri yerine getirebilen bir mikroişlemci tasarımı oluşturmayı ve bu tasarım kalıplarında bir işlemci modeli ortaya koyarak ürünün gerçeklenebilirliğini kanıtlamayı amaçlar.

Bu bağlamda bu aşamaya kadar ortaya konulan çalışmalar ve ürünler, prototiplenebilirliği kanıtlar ve çalışmaların prototiplenerek konsept niteliğinde gerçekleşmesi için belli bir altyapı sağlayarak ışık tutar. Bu çalışmalar ışığında üretilen prototip artık çalışmanın geliştirilebileceğinin, üretilebileceğinin ve kullanılabileceğinin bir göstergesidir.

- Bu çalışmada bahsi geçen proje sonucu Üretilecek grafik işlemci modeli büyük oranda konsept niteliğinde olacağından sadece mantıksal olarak ürünün yapılabilirliğini göstermeye yöneliktir.

  Çalışmaların devam etmesi halinde konsept tasarım gerekli araştırma-geliştime çalışmalarının gerçekleştirilmesi ile önce ürüne dönüştürülebilir bir prototipe ve ilerleyen safhalarda ürüne dönüştürülerek seri üretimine gidilebilir.

Tezin Organizasyonu

Çalışma süresince tasarımı yapılan 8-bit işlemci hiyerarşik yapıda tasarlanmış olup en tepede ki modülde altında farklı tasarım bloklarını ve işlevleri barındıran bir ÜST-SEVİYE tasarım bloğu bulunmaktadır. Bu üst seviye tasarım bloğu aşağıdan yukarıya doğru ve blokları oluşturan modüllerin tek tek işlenmesi ile elde edilmiştir. Bu bağlamda çalışmanın düzenlemesi de şu şekilde yapılmıştır.

1. Bölümde, ÜST-SEVİYE modülümüzü oluşturan iki ana alt bloktan biri olan Bellek bloğu "memory.vhd" dosyası adı altında VHDL dili ile tasarlanmıştır. Bu alt bloğu oluşturan RAM, ROM ve I/O portları komponent olarak bellek bloğuna tanımlanmmıştır. Tasarıma ROM komponentinden başlanıp sırası ile ROM ve I/O portları VHDL donanım tanımlama dili ile tasarlanmıştır.

2. Bölümde, ÜST-SEVİYE modülümüzü oluşturan iki ana alt bloktan ikincisi olan CPU bloğu tasarlanmıştır. Onu oluşturan kontrol ünitesi ve veri-yolu alt blokları da en alt komponentten başlamak üzere VHDL donanım tanımlama dili ile tasarlanmıştır.

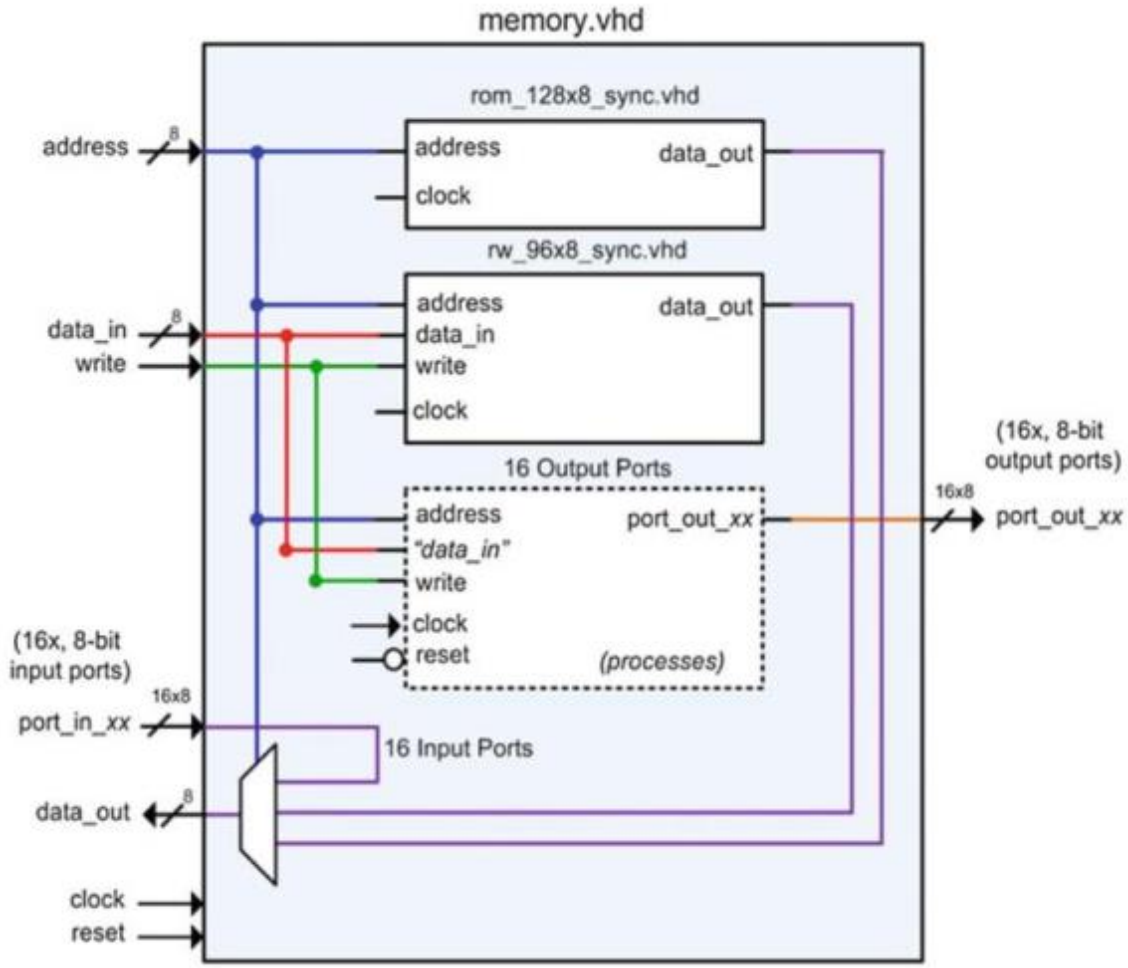
3. Bölümde tasarlanan bloklar ÜST-SEVİYE tasarımımız olan "computer.vhd" bloğu adı altında bir araya getirilerek bağlantıları yapılmıştır.

4. Bölümde tüm tasarım yazılan test dosyası ile Vivado programı üzerinde davranışsal olarak simüle edilmiş ve ROM a kayıtlı test programı üzerinden Waveform üzerinde test ve doğrulama işlemleri gerçekleştirilmiştir.

5. Bölümde, tasarımı, test ve doğrulama işlemleri biten işlemcimiz statik zaman analizi testine sokulmuş, sonuçlar üzerinden bulgu-sonuç değerlendirmeleri yapılmıştır.

1. BÖLÜM

1.BELLEK BLOĞU



Şekil 1.1. memory.vhd Mimarisi

Integer ultricies leo sem, vel tempor lacus ullamcorper in. Suspendisse potenti. Pellentesque nec augue elit. Quisque blandit vestibulum nisi, quis volutpat risus lacinia sed. Morbi odio purus, dignissim sed nisl vitae, condimentum sagittis lectus. Nulla vitae lorem leo. Morbi eget blandit metus, sit amet interdum sem. Aenean vel ligula lacus. Donec faucibus pulvinar tristique. Morbi egestas tristique justo et

tempor.

1.1. Program Belleği (Read-Only Memory - ROM)

"memory.vhd" yapımıza bakacak olursak ROM alt modülüne bağlı bir “address” portu, bir “Clock” portu ve bir de komutu dışarı verecek olan “data-out” portu var. ROM alt bloğumuzun ismini "program-memory" koyalım. Bu bilgiler ışığında yavaşça kodlamaya başlayalım ; Öncelikle kütüphanelerden yazmaya başlayalım.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;
...
```

Bu kütüphaneler bundan sonraki tüm tasarımlarımın başında da aynı şekilde yer alıyor. Bu sebeple bundan sonra ayrıca belirtmeyeceğim.

Daha sonra program belleğine bir "entity" (varlık) ekliyorum. Eklediğim "entity" çerisinde giriş portlarındaki "Clock" ve "Address" sinyallerini ve çıkış portunda ki "data-out" sinyalini ekleyerek "entity" bölümünü bitiriyorum.

```
...
entity program_memory is
   port(
         clk   : in std_logic;
         address : in std_logic_vector(7 downto 0);
         -- Output:
         data_out : out std_logic_vector(7 downto 0)
   );
end program_memory;
...
```

ROM yapısı sadece Read-Only bir yapı yani bu belleğin içine herhangi bir DATA yazamıyoruz o yüzden burada Data-In portu koymadık. Sadece çıkış veriyor. ROM yapılarında üretim aşamasında ilgili sabit veri içerisine kazınır, bu çip ve içerisinde ki veri her zaman orada kalıyor. Biz de programımızı bu çip içerisine gömeceğiz.

Şimdi mimarimizi oluşturmak için "architecture" kısmını yazıyorum, öncelikle; ROM yapısının kendisini belirtmemiz lazım. RAM-ROM gibi yapılar, Bit-Array olarak

ifade edilir. Durum makinalarında durumları belirttiğimiz STATE leri yazar gibi bu sefer de ROM için bir STATE yazacağız. Program belleği 0-127 arasında toplam 128 adet 8-bit lik veriden oluştuğu için Array imizde 128-bit bir Array olacak ve Array içerisindeki her eleman 8-bit olacak. Böylece her elemanı 8-bit olan toplam 128 elemanlı bir Array oluşturmuş oluyorum.

Daha sonra ROM yapısını "constant" olarak tanımlayacağım ve bu ROM yapısını tanımlarken içerisine gömülü olan programı yazacağım.

Örneğin bu 128 lik Array in 0. Elemanı LDA-IMMEDIATELY komutu olsun. Bu komutu yazmak için bellek yapısında komutları ne şekilde sakladığıma bakmam lazım.

Tablo 1.1. Tüm Komutlar

LDA_IMM	x"86"	ADD_AB	x"42"
LDA_DIR	x"87"	SUB_AB	x"43"
LDB_IMM	x"88"	AND_AB	x"44"
LDB_DIR	x"89"	OR_AB	x"45"
STA_DIR	x"96"	INCA	x"46"
STB_DIR	x"97"	INCB	x"47"
BRA	x"20"	DECA	x"48"
BEQ	x"23"	DECB	x"49"

OP kodunu önce yazmam lazım. Daha sonrada bir sonraki elemanın Operand ını yazmam lazım. Yani x"86" dediğim zaman LDA-IMM komutunu belirtmiş oluyorum.

```
...
architecture arch of program_memory is
    type rom_type is array (0 to 127) of std_logic_vector(7 downto 0);
      constant ROM : rom_type := (
                                0  => x"86";
                    );
end architecture;
...
```

Bunu her seferinde bu şekilde yapmamak için bu komutları sabit değerler olarak ROM içine tanımlayacağım. Bütün programları bu şekilde ROM da modelleyeceğimiz zaman yani bir program yazmak istediğimiz zaman gelip VHDL kodunun bu kısmını yazacağız. Bu davranışsal bir kodlamadır. Bunu daha kolay

yapabilmek için bütün komutları sabit olarak tanımladım. Komutlara Türkçe isimler verdim. Mesela LDA-IMM -> "Yükle A Register ı Sabit" çünkü bu komutun mantığı sabit bir sayıyı A Kaydedicisine yüklemekti. OP kodlarını sadece sabit olarak tanımlıyoruz o yüzden kodlar da 8-bit. Ve yazdığımız Constant değeri x"86".

Şimdi aynı şekilde devam edelim. Mesela "LDA-DIR" ifadesi adres vererek A register ına veri çekiyordu. Mesela buna adress demeyelim de "Yükle A" diyelim yani veri belleğinden A register ına adress vererek data yükleme. Bunun da değeri "87".

Store A ve B komutları var, bunlara da Kaydet diyeceğim çünkü Register daki veriyi belleğe kaydediyor. Bunların da değeri 96 ve 97.

```
...
-- Tum komutlar:

-- Kaydet/Yukle komutlari
constant YUKLE_A_SBT :std_logic_vector(7 downto 0) := x"86";
constant YUKLE_A   :std_logic_vector(7 downto 0) := x"87";
constant YUKLE_B_SBT :std_logic_vector(7 downto 0) := x"88";
constant YUKLE_B   :std_logic_vector(7 downto 0) := x"89";
constant KAYDET_A  :std_logic_vector(7 downto 0) := x"96";
constant KAYDET_B  :std_logic_vector(7 downto 0) := x"97";
...
```

Bu şekilde tüm komutlarımızı ROM yapısı içerisinde "constant" olarak tanımlıyorum.

```
...
-- Tum komutlar:
-- Kaydet/Yukle komutlari
constant YUKLE_A_SBT :std_logic_vector(7 downto 0) := x"86";
constant YUKLE_A   :std_logic_vector(7 downto 0) := x"87";
constant YUKLE_B_SBT :std_logic_vector(7 downto 0) := x"88";
constant YUKLE_B   :std_logic_vector(7 downto 0) := x"89";
constant KAYDET_A  :std_logic_vector(7 downto 0) := x"96";
constant KAYDET_B  :std_logic_vector(7 downto 0) := x"97";
-- ALU Komutlari
constant TOPLA_AB  :std_logic_vector(7 downto 0) := x"42";
constant CIKAR_AB  :std_logic_vector(7 downto 0) := x"43";
constant AND_AB      :std_logic_vector(7 downto 0) := x"44";
constant OR_AB     :std_logic_vector(7 downto 0) := x"45";
constant ARTTIR_A  :std_logic_vector(7 downto 0) := x"46";
constant ARTTIR_B  :std_logic_vector(7 downto 0) := x"47";
constant DUSUR_A   :std_logic_vector(7 downto 0) := x"48";
constant DUSUR_B   :std_logic_vector(7 downto 0) := x"49";
```

```
-- Atlama komutlari (Kosullu/Kosulsuz)
constant ATLA           :std_logic_vector(7 downto 0) := x"20";
constant ATLA_NEGATIFSE    :std_logic_vector(7 downto 0) := x"21";
constant ATLA_POZITIFSE    :std_logic_vector(7 downto 0) := x"22";
constant ATLA_ESITSE_SIFIR  :std_logic_vector(7 downto 0) := x"23";
constant ATLA_DEGILSE_SIFIR   :std_logic_vector(7 downto 0) := x"24";
constant ATLA_OVERFLOW_VARSA :std_logic_vector(7 downto 0) := x"25";
constant ATLA_OVERFLOW_YOKSA :std_logic_vector(7 downto 0) := x"26";
constant ATLA_ELDE_VARSA   :std_logic_vector(7 downto 0) := x"27";
constant ATLA_ELDE_YOKSA   :std_logic_vector(7 downto 0) := x"28";
...
```

Ve bu şekilde “constant” olarak tüm komutlarımızı belirledikten sonra kendi programımızı artık yazabiliriz. Yani hiç OPCODE bakmadan buraya kodlabiliriz. Örneğin ROM içerisinde (ROM Array içerisinde) 0. Eleman yani ilk komut YÜKLE-A-SBT olsun. Yani A Kaydedicisine bir sayı yükleyelim. Şimdi OPERAND ı belirteceğiz, “YUKLE-A-SBT” ifadesi OPCODE kısmıydı, bundan sonra OPERAND ın gelmesi lazım. Yüklenecek bu sayı x”0F” sayısı olsun yani 15. Bir sonraki koda geçiyorum, yine OPCODE gelecek. Bu sefer yüklediğimiz A Kaydedicisinde ki sayıyı RAM e kaydedelim. “KAYDET-A” dedik. Ve bu kaydedeceğimiz adreste x“80” olsun. Kayıt yaptıktan sonra “ATLA” diyelim, ve atlayacağımız program sayacı değeri de x“0” olsun yani program başa sarsın. Bu şekilde yazdığımız işlemi sonsuza kadar yapacaktır. Bundan sonraki elemanlar ise “0” olsun yani boş. Komutlarımız bu kadar.

```
...
    type rom_type is array (0 to 127) of std_logic_vector(7 downto 0);
       constant ROM : rom_type := (
                                0  => YUKLE_A_SBT;
                                1   => x"0F";
                                2   => KAYDET_A;
                                3   => x"80";
                                4   => ATLA;
                                5   => x"00";
                                others => x"00";
                        );
...
```

Ve işlemcimizin ilk yazılımını bu şekilde yapmış oldum. Test yaparken komutlarımızı bu şekilde ROM a yazacağız çünkü komutlar buradan okunuyordu.

ROM yapısını bu şekilde tanımladık. “Architecture” tanımına geçmeden halen sinyalleri tanımlıyoruz. Şimdi bir “Enable Signal” sinyali tanımlamalıyız. Bunun sebebi, memory yapımızda aslında şu anda program belleğini kodluyorum. Doğru adres verildi ise içeride bir “Enable Signal” yani İzin Sinyali üretmeliyim.

```
...
-- Sinyaller:
signal enable : std_logic;
end architecture;
...
```

Böylelikle yanlış bir adres program belleğine geldiği zaman herhangi bir okuma işleminin yapılmasını önlemiş olacağım. Bu bir kontrol mekanizması olacak yani adresi kontrol edip bu aralıkta ise okumaya izin ver diyecek.

Artık mimari (architecture) tanımına geçebilirim. “Begin” diyorum. Öncelikle “Enable” sinyaline aktif olacağı, adres kontrolünü yapacağımız “Process” bloğunu yazalım. Bu "Process" bloğu adrese duyarlı olacak ;

```
...
begin

process(address)
begin
   if(address >= x"00" and address <= x"7F") then -- 0 ile 127 araliginda ise
      enable <= '1';
   else
      enable <= '0';
   end if;
end process;
...
```

Eğer doğru aralıktaysak “Enable” sinyalini “1” yapıyoruz. Eğer yanlış aralıktaysak “0” oluyor. Bu "PROCESS" bloğunu kapattık. Şimdi ROM dan okuma yapılacak STATE düğümüne gelelim; Tekrar bir “PROCESS” bloğu açıyorum. Bu PROCESS, CLOCK duyarlı. Çünkü senkron bir şekilde okuma yapacağız. CLOCK atımında yükselen kenarında eğer “Enable” sinyali “1” ise ; Data-Out yani çıkışa ROM yapısının adresteki değerini okuyacağız. Eğer ENABLE izni varsa adress INPUT geldi, bu adreste yazılan değeri ROM dan okumam lazım. Örneğin PC değeri bu olacaktır. Adres 0 olduğunda ROM 0. indeksinde YÜKLE-A-SBT komutu

okunacak. Fakat, bu adres görüldüğü üzere "*std-logic-vector*" ile 8-bit lik bir data tipine sahip. Bizim burada ROM lokasyonuna erişebilmemiz için bunun "*std-logic*" değil, onun yerine bir tam sayı değeri olması lazım. Burada da VHDL in bir özelliğini kullanıyoruz; "*to-integer()*" yani tam sayıya dönüştürme komutu. Burada parantez içine ADDRESS değerini aldık. Bir de Bu değerin UNSIGNED olduğunu belirtmemiz lazım, bu pozitif bir tam sayı değeri. Bunu da "*to-integer(unsigned(adress))*" şeklinde belirttikten sonra adresin belirttiği pozisyona ulaşacağız.

```
...
process(clk)
begin
   if(rising_edge(clk)) then
      if(enable = '1') then
         data_out <= ROM(to_integer(unsigned(address)));
      end if;
   end if;
end process;

end architecture;
...
```

Şimdi tam olarak bir ROM yapısı kurduk. Tüm VHDL tasarımını inceleyecek olursak ;

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

entity program_memory is
   port(
         clk   : in std_logic;
         address : in std_logic_vector(7 downto 0);
         -- Output:
         data_out : out std_logic_vector(7 downto 0)
   );
end program_memory;

architecture arch of program_memory is

-- Tum komutlar:

-- Kaydet/Yukle komutlari
constant YUKLE_A_SBT :std_logic_vector(7 downto 0) := x"86";
```

```
constant YUKLE_A   :std_logic_vector(7 downto 0) := x"87";
constant YUKLE_B_SBT :std_logic_vector(7 downto 0) := x"88";
constant YUKLE_B   :std_logic_vector(7 downto 0) := x"89";
constant KAYDET_A  :std_logic_vector(7 downto 0) := x"96";
constant KAYDET_B  :std_logic_vector(7 downto 0) := x"97";
-- ALU Komutlari
constant TOPLA_AB  :std_logic_vector(7 downto 0) := x"42";
constant CIKAR_AB  :std_logic_vector(7 downto 0) := x"43";
constant AND_AB       :std_logic_vector(7 downto 0) := x"44";
constant OR_AB     :std_logic_vector(7 downto 0) := x"45";
constant ARTTIR_A  :std_logic_vector(7 downto 0) := x"46";
constant ARTTIR_B  :std_logic_vector(7 downto 0) := x"47";
constant DUSUR_A   :std_logic_vector(7 downto 0) := x"48";
constant DUSUR_B   :std_logic_vector(7 downto 0) := x"49";
-- Atlama komutlari (Kosullu/Kosulsuz)
constant ATLA           :std_logic_vector(7 downto 0) := x"20";
constant ATLA_NEGATIFSE    :std_logic_vector(7 downto 0) := x"21";
constant ATLA_POZITIFSE    :std_logic_vector(7 downto 0) := x"22";
constant ATLA_ESITSE_SIFIR  :std_logic_vector(7 downto 0) := x"23";
constant ATLA_DEGILSE_SIFIR   :std_logic_vector(7 downto 0) := x"24";
constant ATLA_OVERFLOW_VARSA :std_logic_vector(7 downto 0) := x"25";
constant ATLA_OVERFLOW_YOKSA :std_logic_vector(7 downto 0) := x"26";
constant ATLA_ELDE_VARSA   :std_logic_vector(7 downto 0) := x"27";
constant ATLA_ELDE_YOKSA   :std_logic_vector(7 downto 0) := x"28";

type rom_type is array (0 to 127) of std_logic_vector(7 downto 0);
   constant ROM : rom_type := (
                              0  => YUKLE_A_SBT;
                              1   => x"0F";
                              2   => KAYDET_A;
                              3   => x"80";
                              4   => ATLA;
                              5   => x"00";
                              others => x"00";
                     );

-- Sinyaller:
signal enable : std_logic;
begin

process(address)
begin
   if(address >= x"00" and address <= x"7F") then -- 0 ile 127 araliginda ise
      enable <= '1';
   else
      enable <= '0';
   end if;
end process;
----
process(clk)
```

```
begin
   if(rising_edge(clk)) then
      if(enable = '1') then
         data_out <= ROM(to_integer(unsigned(address)));
      end if;
   end if;
end process;

end architecture;
```

"Program belleği" bu kadar, bir sonraki adımda "veri Belleğini" tasarlayacağım ki zaten ROM ile çok benzer yapılar.

1.2. Veri Belleği (Random-Access Memory - RAM)

96x8 bitlik veri belleği tasarımını inceleyelim. Bu yapı "data-in" girişi ve "data-out" çıkışına sahip. CPU tarafından da bir "write" sinyali alıyor. Yapı olarak ROM yapısına çok benzese de sadece Portlarında birkaç ufak farklılık olacak.

Kütüphanelerimi tanımlayıp tasarımıma başlıyorum. ROM da "Entity" kısmını olduğu gibi alıyorum. Bu Entity bloğunun adı artık "data-memory" olsun. Adreslere geldiğimizde; "clk" ve "address" kısımları aynı. Sadece "data-in" eklememiz lazım çünkü RAM olduğun için veri kaydedebiliyoruz. Tüm verilerim 8-bit olduğundan bunu da 8-bit tanımlıyorum. Birde CPU tarafından yazmak için kullanılacak bir kontrol sinyali var; Bu sinyalde yine giriş tarafında olup ismi "write-enable" olsun. Bu CPU tarafından verilecek 1-bit lik bir INPUT sinyali.

```
...
entity data_memory is
   port(
         clk        : in std_logic;
         address    : in std_logic_vector(7 downto 0);
         data_in    : in std_logic_vector(7 downto 0);
         write_en   : in std_logic; -- CPU tarafindan gonderilen kontrol sinyali
         -- Output:
         data_out : out std_logic_vector(7 downto 0)
   );
end data_memory;
...
```

Sonrasında ROM da tanımladığımız gibi bir "data-type" oluşturacağım. Bu sefer

Array kapasitemiz 96 eleman. Architecture tanımlayarak başladım.

Burada Array imiz ve 0-127 değil 128 den 223 e olacak (x"80" – x"DF"), 128-223 arasında olduğu için indekslemeyi Array başlangıç indeksini 128, bitiş indeksini 223 dedik. Ayrıca 96x8 bit.

```
...
architecture arch of data_memory is

type ram_type is array (128 to 223) of std_logic_vector(7 downto 0); -- 96x8-bit
...
```

Şimdi RAM bloğunun kendisini tanımlıyorum. ROM yapısını constant olarak tanımlamıştım çünkü komutlarımız constant türündeydi, girişleri rahatça verebilmek için constant yaptım. RAM yapısını ise "signal" olarak tanımladım. Data tipi "ram-type". Bütün başlangıç değerleri "0", çünkü RAM dizesini boş oluşturacağız. Program süresince CPU nun istediği veriler ile dolacak.

```
...
signal RAM : ram_type := (others => x"00"); -- Baslangic verilerinin hepsi sifir
...
```

Aynı şekilde ROM yapısında kullandığımız gibi burada da bir ENABLE mantığı yürütelim. Sadece ilgili adreste ENABLE olsun ve yazmaya/işlem yapmaya izin versin. ROM yapısında olduğu gibi adrese duyarlı bir process bloğu olacak, sadece burada ki adresler artık x"80" ile x"DF" arasında olacak. Yani 128 ile 223 aralığında.

Ayrıca ROM yapısında CLOCK atımlarına duyarlı olarak kurduğumuz PROCESS bloğu da yine tamamen aynı, okuma için verilen ifade bu şekilde olacak. Eğer ENABLE sinyali "1" ise ve CPU tarafından yaz emri verildi ise (write-en=1) biz burada yazma yapacağız. Yazmada, RAM yapısının adreslenmesi yine tam sayı olmak zorunda, bu yüzden "*to-integer(unsigned(address))*" dedim, giriş sinyali olan "data-in" i RAM yapısının ilgili adres pozisyonuna yaz demek, yani "write-enable" emri geldi ise gelen datayı istediğimiz RAM adresine yazabileceğim.

Bir de okuma yapmam lazım çünkü RAM bloğu hem yazma hem okuma yapabilmelidir. Bu sefer "write-en" sinyalinin 0 olması lazım çünkü yazma değil okuma yapılacak, okumayı da "data-out" sinyalinden veriyoruz, yani belleğin çıkış

portundan. Okuduğumuz ifade de RAM bloğu adresindeki veridir.

```
...
begin

process(address)
begin
   if(address >= x"80" and address <= x"DF") then -- 128 ile 223 araliginda ise
      enable <= '1';
   else
      enable <= '0';
   end if;
end process;

--
process(clk)
begin
   if(rising_edge(clk)) then
      if(enable = '1' and write_en = '1') then -- Yazma
         RAM(to_integer(unsigned(address))) <= data_in;
      elsif(enable = '1' and write_en = '0') then -- Okuma
         data_out <= RAM(to_integer(unsigned(address)));
      end if;
   end if;
end process;

end architecture;
...
```

Veri belleği de bu şekilde tamamlanmış oldu. Veri belleğinin tüm kodları ;

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

entity data_memory is
   port(
         clk        : in std_logic;
         address    : in std_logic_vector(7 downto 0);
         data_in    : in std_logic_vector(7 downto 0);
         write_en   : in std_logic; -- CPU tarafindan gonderilen kontrol sinyali
         -- Output:
         data_out : out std_logic_vector(7 downto 0)
   );
end data_memory;

architecture arch of data_memory is

type ram_type is array (128 to 223) of std_logic_vector(7 downto 0); -- 96x8-bit
```

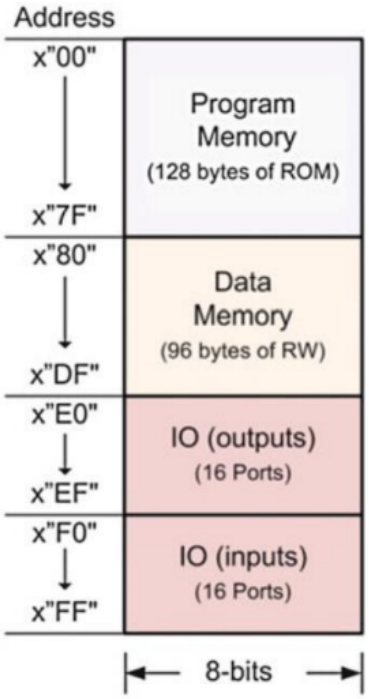
```
signal RAM : ram_type := (others => x"00"); -- Baslangic verileri hepsi sifir

signal enable : std_logic;

begin

process(address)
begin
   if(address >= x"80" and address <= x"DF") then -- 128 ile 223 araliginda ise
      enable <= '1';
   else
      enable <= '0';
   end if;
end process;

--
process(clk)
begin
   if(rising_edge(clk)) then
      if(enable = '1' and write_en = '1') then -- Yazma
         RAM(to_integer(unsigned(address))) <= data_in;
      elsif(enable = '1' and write_en = '0') then -- Okuma
         data_out <= RAM(to_integer(unsigned(address)));
      end if;
   end if;
end process;

end architecture;
```

1.3. Çıkış Portları

Program belleğini ve Veri belleğini tamamladık, sıra 16 tane OUTPUT portunun verildiği yapıyı tasarlamakta. Bellek Bloğu diyagramında görüleceği üzere burada Bellek bloğunun dışına verilen 16 tane 8-bit lik Output portu bulunuyor.

RAM ile aslında tamamen aynı INPUT portlarını alıyor, ekstra olarak RESET sinyalini alıyor çünkü RESET sinyali geldiği zaman OUTPUT portlarının yani burada ki Kaydedicilerin sıfırlanistenmiş. Port dediğimiz şey aslında Kaydedici, yani burada 8-bit lik verileri tutan 16 adet Kaydedici var ve bu Kaydediciler dışarı veriliyor. Bunların adresleri ne peki;

Veri belleğinden sonra x"E0" ile x"EF" arasında 16 adet adres var. Yani örneğin 0. OUTPUT portunun adresi x"E0", 1. Output portunun adresi x"E1", sonrası



Şekil 1.2. Bellek Adresleme Yapısı

x"E2", x"E3", .., x"EF" ye kadar gidiyor.

Entity ismimiz "output-ports". Portlarımız da CLOCK var, bir de burada RESET olacak. Çünkü Reset ile OUTPUT portlarının sıfırlanabiliyor olması lazım. "Address" var. "Write-En" var. "Data-In" var. Yani liselersek ; Data-in, adress, write, clock, reset. sinyalleri ve Inputlar. 16 adet Output portunun hepsi 8-bit olacak. Ve tam olarak 16 tane var.

```
...
entity output_ports is
   port(
         clk         : in std_logic;
         rst         : in std_logic;
         address     : in std_logic_vector(7 downto 0);
         data_in     : in std_logic_vector(7 downto 0);
         write_en    : in std_logic; -- CPU tarafindan gonderilen kontrol sinyali / yaz
         -- Output:
         port_out_00 : out std_logic_vector(7 downto 0);
         port_out_01 : out std_logic_vector(7 downto 0);
         port_out_02 : out std_logic_vector(7 downto 0);
         port_out_03 : out std_logic_vector(7 downto 0);
         port_out_04 : out std_logic_vector(7 downto 0);
         port_out_05 : out std_logic_vector(7 downto 0);
         port_out_06 : out std_logic_vector(7 downto 0);
         port_out_07 : out std_logic_vector(7 downto 0);
```

```
        port_out_08 : out std_logic_vector(7 downto 0);
        port_out_09 : out std_logic_vector(7 downto 0);
        port_out_10 : out std_logic_vector(7 downto 0);
        port_out_11 : out std_logic_vector(7 downto 0);
        port_out_12 : out std_logic_vector(7 downto 0);
        port_out_13 : out std_logic_vector(7 downto 0);
        port_out_14 : out std_logic_vector(7 downto 0);
        port_out_15 : out std_logic_vector(7 downto 0)
   );
end output_ports;
...
```

Bu şekilde 16 adet portumuzu tanımladık. Şimdi architecture tanımına geçelim ; Burada herhangi bir sinyal tanımlamama gerek yok. Bir RAM-ROM işlemi yapmayacağım için ENABLE yapmayacağım. Doğrudan PROCESS bloğuna geçiyorum.

CLOCK ve RESET sinyallerine duyarlı bir process bloğu oluşturacağım. Eğer RESET geliyor ise yani "rst" sinyalimiz "1" olduğu sürece reset aktif demek. Devamında ise RESET durumundan çıkıp CLOCK durumuna geldiğimiz durum var.

```
...
architecture arch of output_ports is
begin

   process(clk,rst)
   begin
      if(rst = '1') then

        elsif(rising_edge(clk)) then

      end if;
...
```

Öncelikle bütün portları sıfırlamam lazım.

```
...
architecture arch of output_ports is
begin

   process(clk,rst)
   begin
      if(rst = '1') then
         port_out_00 <= (others => '0');
```

```
        port_out_01 <= (others => '0');
        port_out_02 <= (others => '0');
        port_out_03 <= (others => '0');
        port_out_04 <= (others => '0');
        port_out_05 <= (others => '0');
        port_out_06 <= (others => '0');
        port_out_07 <= (others => '0');
        port_out_08 <= (others => '0');
        port_out_09 <= (others => '0');
        port_out_10 <= (others => '0');
        port_out_11 <= (others => '0');
        port_out_12 <= (others => '0');
        port_out_13 <= (others => '0');
        port_out_14 <= (others => '0');
        port_out_15 <= (others => '0');
     elsif(rising_edge(clk)) then

     end if;
...
```

Reset işlemi bu kadar. Portlara veri yazabilmesi için CPU nun gönderdiği "write-en" sinyalinin "1" olması lazım, koşulumuz bu. CLOCK duyarlı else-if de her "if" ifadesinde "else" i olmak zorunda değil, bu LATCH hatası arz etmiyor flip-flop olduğu için. Şimdi tek tek her adres için portunun durumunu belirtmemiz lazım;

- *Case "input olan adres" ...*

Şimdi her adreste tek tek hangi port işaret edilecekse yazacağız ; Mesela ilk portumuzun adresi x"E0". Bu şekilde 16 adet portuda yazdıktan sonra olası bir durum söz konusu ; eğer başka bir adres gelirse ne yapılacak? Başka bir adres ile ilglilenmiyoruz, bu yüzden bütün OUTPUT portlarını "0" değerine çekelim. Böylece sadece hedeflenen adres değerlerinde bir port ataması yapılmış olacak.

Bu şekilde OUTPUT bloğu tasarımımızı tamamlamış olduk ;

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

entity output_ports is
   port(
        clk        : in std_logic;
        rst        : in std_logic;
```

```
        address    : in std_logic_vector(7 downto 0);
        data_in    : in std_logic_vector(7 downto 0);
        write_en   : in std_logic; -- CPU tarafindan gonderilen kontrol sinyali / yaz
        -- Output:
        port_out_00 : out std_logic_vector(7 downto 0);
        port_out_01 : out std_logic_vector(7 downto 0);
        port_out_02 : out std_logic_vector(7 downto 0);
        port_out_03 : out std_logic_vector(7 downto 0);
        port_out_04 : out std_logic_vector(7 downto 0);
        port_out_05 : out std_logic_vector(7 downto 0);
        port_out_06 : out std_logic_vector(7 downto 0);
        port_out_07 : out std_logic_vector(7 downto 0);
        port_out_08 : out std_logic_vector(7 downto 0);
        port_out_09 : out std_logic_vector(7 downto 0);
        port_out_10 : out std_logic_vector(7 downto 0);
        port_out_11 : out std_logic_vector(7 downto 0);
        port_out_12 : out std_logic_vector(7 downto 0);
        port_out_13 : out std_logic_vector(7 downto 0);
        port_out_14 : out std_logic_vector(7 downto 0);
        port_out_15 : out std_logic_vector(7 downto 0)
    );
end output_ports;

architecture arch of output_ports is
begin

    process(clk,rst)
    begin
        if(rst = '1') then
            port_out_00 <= (others => '0');
            port_out_01 <= (others => '0');
            port_out_02 <= (others => '0');
            port_out_03 <= (others => '0');
            port_out_04 <= (others => '0');
            port_out_05 <= (others => '0');
            port_out_06 <= (others => '0');
            port_out_07 <= (others => '0');
            port_out_08 <= (others => '0');
            port_out_09 <= (others => '0');
            port_out_10 <= (others => '0');
            port_out_11 <= (others => '0');
            port_out_12 <= (others => '0');
            port_out_13 <= (others => '0');
            port_out_14 <= (others => '0');
            port_out_15 <= (others => '0');
        elsif(rising_edge(clk)) then
            if(write_en = '1') then
                case address is
                    when x"E0" =>
                        port_out_00 <= data_in;
```

```
            when x"E1" =>
               port_out_01 <= data_in;
            when x"E2" =>
               port_out_02 <= data_in;
            when x"E3" =>
               port_out_03 <= data_in;
            when x"E4" =>
               port_out_04 <= data_in;
            when x"E5" =>
               port_out_05 <= data_in;
            when x"E6" =>
               port_out_06 <= data_in;
            when x"E7" =>
               port_out_07 <= data_in;
            when x"E8" =>
               port_out_08 <= data_in;
            when x"E9" =>
               port_out_09 <= data_in;
            when x"EA" =>
               port_out_10 <= data_in;
            when x"EB" =>
               port_out_11 <= data_in;
            when x"EC" =>
               port_out_12 <= data_in;
            when x"ED" =>
               port_out_13 <= data_in;
            when x"EE" =>
               port_out_14 <= data_in;
            when x"EF" =>
               port_out_15 <= data_in;
            when others =>
               port_out_00 <= (others => '0');
               port_out_01 <= (others => '0');
               port_out_02 <= (others => '0');
               port_out_03 <= (others => '0');
               port_out_04 <= (others => '0');
               port_out_05 <= (others => '0');
               port_out_06 <= (others => '0');
               port_out_07 <= (others => '0');
               port_out_08 <= (others => '0');
               port_out_09 <= (others => '0');
               port_out_10 <= (others => '0');
               port_out_11 <= (others => '0');
               port_out_12 <= (others => '0');
               port_out_13 <= (others => '0');
               port_out_14 <= (others => '0');
               port_out_15 <= (others => '0');
         end case;
      end if;
   end if;
```

```
   end process;

end architecture;
```

OUTPUT portları da bu kadar, bellek elemanlarının tüm alt blok tasarımlarını yaptık. Artık Bellek Top-modülünü birleştirmeye hazırız.

1.4. Veri Belleği - Bellek Üst-Seviye Tasarım

Program belleğini, veri belleğini ve output portlarını modelledim. Bu alt blokları kullanarak "Memory.vhd" top bloğunu oluşturmam lazım. Bu tüm hafıza elemanlarını bünyesinde barındıran top-modüldür. Bu alt blokları tanımladıktan sonra bir de bellek yapısının sol altında gördüğümüz Multiplexer yapısınıda bu top-modül içerisinde modellemiş olmam lazım.

Bu multiplexer ne yapıyor? Seçim sinyali adrese bağlı, dolayısıyla hangi adres pozisyonunda isek eğer, verilen adres sinyali o bloğa ait Output sinyalini çıkışa veriyordu.

Entity ismi "memory" olsun. Şimdi INPUT portlarına bakalım. Alt bloklarda kullanılan INPUT portlarıda burada da INPUT olacak. CLOCK, RESET, ADDRESS, Data-in, Write-en sinyallerini de yazacağım. 16 tane 8-bit lik INPUT portu gerekiyor, bunların da tanımlanması lazım. Önceki kısımda OUTPUT kısmında yazdığım OUTPUT portlarını alıp isimlerini INPUT olarak değiştireceğim.

Output lara baktığım zaman sadece "16 Output Ports" bloğunun dışarı vermiş olduğu 16 adet OUTPUT portu, birde Multiplexer yapısından çıkıp CPU ya gidecek olan "data-out" portu var.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

entity memory is
   port(
        clk         : in std_logic;
        rst         : in std_logic;
        address     : in std_logic_vector(7 downto 0);
```

```
        data_in    : in std_logic_vector(7 downto 0);
        write_en   : in std_logic; -- CPU tarafindan gonderilen kontrol sinyali
        port_in_00 : in std_logic_vector(7 downto 0);
        port_in_01 : in std_logic_vector(7 downto 0);
        port_in_02 : in std_logic_vector(7 downto 0);
        port_in_03 : in std_logic_vector(7 downto 0);
        port_in_04 : in std_logic_vector(7 downto 0);
        port_in_05 : in std_logic_vector(7 downto 0);
        port_in_06 : in std_logic_vector(7 downto 0);
        port_in_07 : in std_logic_vector(7 downto 0);
        port_in_08 : in std_logic_vector(7 downto 0);
        port_in_09 : in std_logic_vector(7 downto 0);
        port_in_10 : in std_logic_vector(7 downto 0);
        port_in_11 : in std_logic_vector(7 downto 0);
        port_in_12 : in std_logic_vector(7 downto 0);
        port_in_13 : in std_logic_vector(7 downto 0);
        port_in_14 : in std_logic_vector(7 downto 0);
        port_in_15 : in std_logic_vector(7 downto 0);
        -- Output:
        data_out : out std_logic_vector(7 downto 0);
        --
        port_out_00 : out std_logic_vector(7 downto 0);
        port_out_01 : out std_logic_vector(7 downto 0);
        port_out_02 : out std_logic_vector(7 downto 0);
        port_out_03 : out std_logic_vector(7 downto 0);
        port_out_04 : out std_logic_vector(7 downto 0);
        port_out_05 : out std_logic_vector(7 downto 0);
        port_out_06 : out std_logic_vector(7 downto 0);
        port_out_07 : out std_logic_vector(7 downto 0);
        port_out_08 : out std_logic_vector(7 downto 0);
        port_out_09 : out std_logic_vector(7 downto 0);
        port_out_10 : out std_logic_vector(7 downto 0);
        port_out_11 : out std_logic_vector(7 downto 0);
        port_out_12 : out std_logic_vector(7 downto 0);
        port_out_13 : out std_logic_vector(7 downto 0);
        port_out_14 : out std_logic_vector(7 downto 0);
        port_out_15 : out std_logic_vector(7 downto 0)
    );
end memory;
...
```

Architecture kısmına geçiyorum. Burada sinyal deklerasyon kısmında alt blokların hepsini komponent olarak tanımlamam gerekir, bunu da kullanacağım blokların Entity bloğunu alıp adını komponent olarak değiştirerek yapıyorum.

```
...
architecture arch of memory is
```

```
-- Program Bellegi:
component program_memory is
   port(
        clk        : in std_logic;
        address    : in std_logic_vector(7 downto 0);
        -- Output:
        data_out : out std_logic_vector(7 downto 0)
   );
end component;

-- Veri Bellegi (RAM):
component data_memory is
   port(
        clk        : in std_logic;
        address    : in std_logic_vector(7 downto 0);
        data_in    : in std_logic_vector(7 downto 0);
        write_en   : in std_logic; -- CPU tarafindan gonderilen kontrol sinyali
        -- Output:
        data_out : out std_logic_vector(7 downto 0)
   );
end component;

-- Output Portlari:
component output_ports is
   port(
        clk        : in std_logic;
        rst        : in std_logic;
        address    : in std_logic_vector(7 downto 0);
        data_in    : in std_logic_vector(7 downto 0);
        write_en   : in std_logic; -- CPU tarafindan gonderilen kontrol sinyali
        -- Output:
        port_out_00 : out std_logic_vector(7 downto 0);
        port_out_01 : out std_logic_vector(7 downto 0);
        port_out_02 : out std_logic_vector(7 downto 0);
        port_out_03 : out std_logic_vector(7 downto 0);
        port_out_04 : out std_logic_vector(7 downto 0);
        port_out_05 : out std_logic_vector(7 downto 0);
        port_out_06 : out std_logic_vector(7 downto 0);
        port_out_07 : out std_logic_vector(7 downto 0);
        port_out_08 : out std_logic_vector(7 downto 0);
        port_out_09 : out std_logic_vector(7 downto 0);
        port_out_10 : out std_logic_vector(7 downto 0);
        port_out_11 : out std_logic_vector(7 downto 0);
        port_out_12 : out std_logic_vector(7 downto 0);
        port_out_13 : out std_logic_vector(7 downto 0);
        port_out_14 : out std_logic_vector(7 downto 0);
        port_out_15 : out std_logic_vector(7 downto 0)
   );
end component;
...
```

Komponent tanımlarım bu şekilde bitti. Sinyale ihtiyacımız var mı? Evet var, çünkü biz burada ki Multiplexer yapısınıda burada ki Top-Modüle modelleyeceğimiz için burada ki ROM ve RAM çıkışlarını sinyal olarak belirtmeliyim. Port-Map bloğunda bunları bağlarken bu sinyalleri ara bağlantı olarak kullanacağım. Buna da " MUX Sinyalleri " diyelim.

Signal Rom-Out : program belleğinin çıkışı, yani Bellek Bloğundaki "rom-128x8-sync.vhd" yapısından çıkıp multiplexer yapısına giden "data-out" kablosunu yani ROM dan multiplexer yapısına giden kabloyu bağlayacağım. Sonrasında da RAM kablosunu bağlayacağım;

```
...
-- MUX sinyalleri
signal rom_out : std_logic_vector(7 downto 0);
signal ram_out : std_logic_vector(7 downto 0);
...
```

Sonrasında Architecture içinde belirttiğimiz "komponent" değerlerini bağlamamız gerekiyor. Önce ROM bloğunu bağlayalım; Bunun için Port-Map kuracağım. Bunun Port-Map bağlantısına ROM-Unit (ROM-U) ismini verelim. Sonrasında bloğun adını yazıyorum, bloğumuzun adı "program-memory" idi.

komponente/entity sine dönüp "program*memory" de belirttiğimiz inputları alacağım. Sonra Top-modüle ait ilgili portlar ile bağlantısını yapacağım. Alt blok ve top-modül arasında ki bağlantıyı Port-Map ile sağlamış olacağım.

"Program-memory" saatine "memory" nin saati bağlı. Adresine girişteki adres bağlı. Çıkışına da bizim burada tanımladığımız multiplexer yapısına giden "data-out" kablosu (rom-out) bağlı.

```
...
-- ROM:
ROM_U: program_memory port map
              (
                 clk         => clk,
                  address    => address,
                  -- Output:
                 data_out => rom_out
              );
...
```

RAM bloğunu da aynı şekilde tanımlayacağım.

```
...
-- RAM:
RAM_U: data_memory port map
              (
                 clk        => clk,
                  address   => address,
                 data_in    => data_in,
                 write_en   => write_en, -- RAM => Memory-TOP
                  -- Output:
                 data_out => ram_out
              );
...
```

Ve son olarak OUTPUT portlarının da Port-Map ile bağlantılarını kuruyorum

```
...
-- OUTPUT PORTLARI:
OUT_U: output_ports port map
              (
                 clk         => clk ,
                 rst         => rst ,
                 address     => address,
                 data_in     => data_in ,
                 write_en    => write_en ,
                 -- Output:  => -- Output: ,
                 port_out_00 => port_out_00 ,
                 port_out_01 => port_out_01 ,
                 port_out_02 => port_out_02 ,
                 port_out_03 => port_out_03 ,
                 port_out_04 => port_out_04 ,
                 port_out_05 => port_out_05 ,
                 port_out_06 => port_out_06 ,
                 port_out_07 => port_out_07 ,
                 port_out_08 => port_out_08 ,
                 port_out_09 => port_out_09 ,
                 port_out_10 => port_out_10 ,
                 port_out_11 => port_out_11 ,
                 port_out_12 => port_out_12 ,
                 port_out_13 => port_out_13 ,
                 port_out_14 => port_out_14 ,
                 port_out_15 => port_out_15

              );
...
```

Multiplexer yapısını ise bir PROCESS bloğu ile tanımlayacağım. Bu blok içerisinde

kullanacağımız sinyalleri “sensivity list” içerisine yazmamız lazım. Kesinlikle adres olacak, zaten seçim sinyali. Inputlar da ROM-OUT ve RAM-OUT, ayrıca input portlarını da yazmamız lazım onlarda input olarak gireceği için;

```
...
process(address, rom_out, ram_out,
        port_in_00, port_in_01, port_in_02, port_in_03,
        port_in_04, port_in_05, port_in_06, port_in_07,
        port_in_07, port_in_08, port_in_09, port_in_10,
        port_in_11, port_in_12, port_in_13, port_in_14, port_in_15)
  begin
     if(address >= x"00" and address <= x"7F") then
        data_out <= rom_out;
     elsif(address >= x"80" and address <= x"DF") then
        data_out <= ram_out;
     -- Input Routing
     elsif(address = x"F0") then
        data_out <= port_in_00;
     elsif(address = x"F1") then
        data_out <= port_in_01;
     elsif(address = x"F2") then
        data_out <= port_in_02;
     elsif(address = x"F3") then
        data_out <= port_in_03;
     elsif(address = x"F4") then
        data_out <= port_in_04;
     elsif(address = x"F5") then
        data_out <= port_in_05;
     elsif(address = x"F6") then
        data_out <= port_in_06;
     elsif(address = x"F7") then
        data_out <= port_in_07;
     elsif(address = x"F8") then
        data_out <= port_in_08;
     elsif(address = x"F9") then
        data_out <= port_in_09;
     elsif(address = x"FA") then
        data_out <= port_in_10;
     elsif(address = x"FB") then
        data_out <= port_in_11;
     elsif(address = x"FC") then
        data_out <= port_in_12;
     elsif(address = x"FD") then
        data_out <= port_in_13;
     elsif(address = x"FE") then
        data_out <= port_in_14;
     elsif(address = x"FF") then
        data_out <= port_in_15;
     else
        data_out <= x"00";
```

```
      end if;
   end process;
...
```

inputları da yazdıktan sonra parantezi kapattım, begin dedim. Multiplexer yapısını modellemeye başladım. Eğer adres x"00" ile adres x"7F" aralığında ise multiplexer in çıkışı olan data-out, ROM-out olacak. Program belleğinin çıkışı olacak. Adres 80 ile DF aralığında ise multiplexer in çıkışı RAM-out olacak. Sırada inputlar var. Buna Input-routing dedim. Yani multiplexer deki "16 input ports" yazan kısımdayım. x"F0" dan x"FF" e kadar yazdım.

Eğer bunlardan hiç biri değilse MUX çıkışına "0" bastırdım.

Böylelikle Memory bloğunun da TOP bloğunu tasarlamış oldum. Artık bir üst modüle çıkıp CPU tasarımına geçebilirim. Memory bu kadar.

Memory.vhd kodları ;

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

entity memory is
   port(
         clk        : in std_logic;
         rst        : in std_logic;
         address    : in std_logic_vector(7 downto 0);
         data_in    : in std_logic_vector(7 downto 0);
         write_en   : in std_logic; -- CPU tarafindan gonderilen kontrol sinyali / yaz
         port_in_00 : in std_logic_vector(7 downto 0);
         port_in_01 : in std_logic_vector(7 downto 0);
         port_in_02 : in std_logic_vector(7 downto 0);
         port_in_03 : in std_logic_vector(7 downto 0);
         port_in_04 : in std_logic_vector(7 downto 0);
         port_in_05 : in std_logic_vector(7 downto 0);
         port_in_06 : in std_logic_vector(7 downto 0);
         port_in_07 : in std_logic_vector(7 downto 0);
         port_in_08 : in std_logic_vector(7 downto 0);
         port_in_09 : in std_logic_vector(7 downto 0);
         port_in_10 : in std_logic_vector(7 downto 0);
         port_in_11 : in std_logic_vector(7 downto 0);
         port_in_12 : in std_logic_vector(7 downto 0);
         port_in_13 : in std_logic_vector(7 downto 0);
         port_in_14 : in std_logic_vector(7 downto 0);
```

```
        port_in_15 : in std_logic_vector(7 downto 0);
        -- Output:
        data_out : out std_logic_vector(7 downto 0);
        --
        port_out_00 : out std_logic_vector(7 downto 0);
        port_out_01 : out std_logic_vector(7 downto 0);
        port_out_02 : out std_logic_vector(7 downto 0);
        port_out_03 : out std_logic_vector(7 downto 0);
        port_out_04 : out std_logic_vector(7 downto 0);
        port_out_05 : out std_logic_vector(7 downto 0);
        port_out_06 : out std_logic_vector(7 downto 0);
        port_out_07 : out std_logic_vector(7 downto 0);
        port_out_08 : out std_logic_vector(7 downto 0);
        port_out_09 : out std_logic_vector(7 downto 0);
        port_out_10 : out std_logic_vector(7 downto 0);
        port_out_11 : out std_logic_vector(7 downto 0);
        port_out_12 : out std_logic_vector(7 downto 0);
        port_out_13 : out std_logic_vector(7 downto 0);
        port_out_14 : out std_logic_vector(7 downto 0);
        port_out_15 : out std_logic_vector(7 downto 0)
    );
end memory;


architecture arch of memory is

-- Program Bellegi:
component program_memory is
    port(
        clk         : in std_logic;
        address     : in std_logic_vector(7 downto 0);
        -- Output:
        data_out : out std_logic_vector(7 downto 0)
    );
end component;

-- Veri Bellegi (RAM):
component data_memory is
    port(
        clk         : in std_logic;
        address     : in std_logic_vector(7 downto 0);
        data_in     : in std_logic_vector(7 downto 0);
        write_en    : in std_logic; -- CPU tarafindan gonderilen kontrol sinyali / yaz
        -- Output:
        data_out : out std_logic_vector(7 downto 0)
    );
end component;

-- Output Portlari:
component output_ports is
```

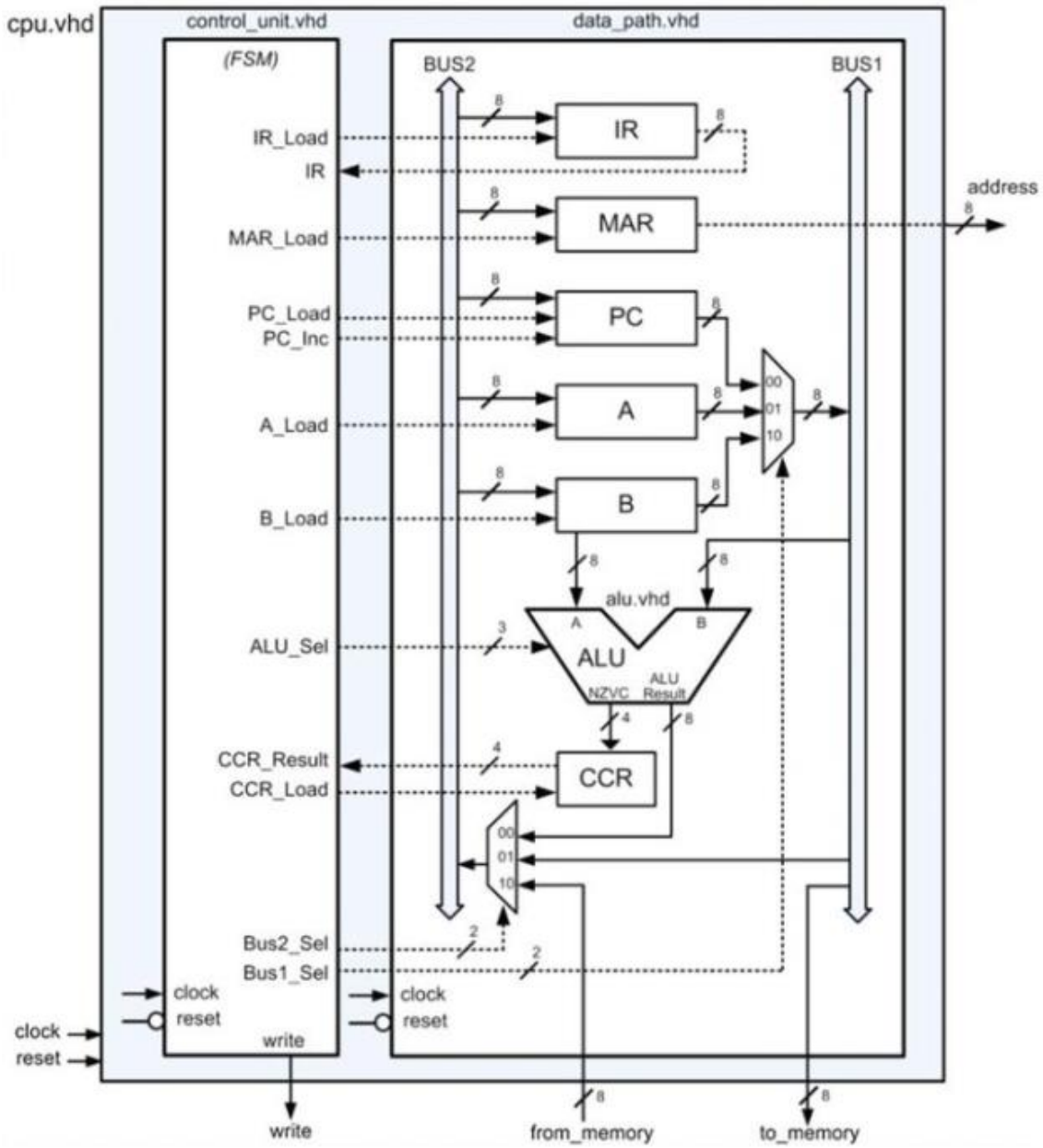
```
   port(
        clk        : in std_logic;
        rst        : in std_logic;
        address    : in std_logic_vector(7 downto 0);
        data_in    : in std_logic_vector(7 downto 0);
        write_en   : in std_logic; -- CPU tarafindan gonderilen kontrol sinyali / yaz
        -- Output:
        port_out_00 : out std_logic_vector(7 downto 0);
        port_out_01 : out std_logic_vector(7 downto 0);
        port_out_02 : out std_logic_vector(7 downto 0);
        port_out_03 : out std_logic_vector(7 downto 0);
        port_out_04 : out std_logic_vector(7 downto 0);
        port_out_05 : out std_logic_vector(7 downto 0);
        port_out_06 : out std_logic_vector(7 downto 0);
        port_out_07 : out std_logic_vector(7 downto 0);
        port_out_08 : out std_logic_vector(7 downto 0);
        port_out_09 : out std_logic_vector(7 downto 0);
        port_out_10 : out std_logic_vector(7 downto 0);
        port_out_11 : out std_logic_vector(7 downto 0);
        port_out_12 : out std_logic_vector(7 downto 0);
        port_out_13 : out std_logic_vector(7 downto 0);
        port_out_14 : out std_logic_vector(7 downto 0);
        port_out_15 : out std_logic_vector(7 downto 0)
   );
end component;

-- MUX sinyalleri
signal rom_out : std_logic_vector(7 downto 0);
signal ram_out : std_logic_vector(7 downto 0);

begin

-- ROM:
ROM_U: program_memory port map
             (
                clk        => clk,
                 address   => address,
                 -- Output:
                data_out => rom_out
             );

-- RAM:
RAM_U: data_memory port map
             (
                clk        => clk,
                 address   => address,
                data_in    => data_in,
                write_en   => write_en,
                 -- Output:
                data_out => ram_out
```

```
            );
-- OUTPUT PORTLARI:
OUT_U: output_ports port map
            (
               clk         => clk ,
               rst         => rst ,
               address     => address,
               data_in     => data_in ,
               write_en    => write_en ,
               -- Output:  => -- Output: ,
               port_out_00 => port_out_00 ,
               port_out_01 => port_out_01 ,
               port_out_02 => port_out_02 ,
               port_out_03 => port_out_03 ,
               port_out_04 => port_out_04 ,
               port_out_05 => port_out_05 ,
               port_out_06 => port_out_06 ,
               port_out_07 => port_out_07 ,
               port_out_08 => port_out_08 ,
               port_out_09 => port_out_09 ,
               port_out_10 => port_out_10 ,
               port_out_11 => port_out_11 ,
               port_out_12 => port_out_12 ,
               port_out_13 => port_out_13 ,
               port_out_14 => port_out_14 ,
               port_out_15 => port_out_15

            );

-----------------------------------------

   process(address, rom_out, ram_out,
         port_in_00, port_in_01, port_in_02, port_in_03,
         port_in_04, port_in_05, port_in_06, port_in_07,
         port_in_07, port_in_08, port_in_09, port_in_10,
         port_in_11, port_in_12, port_in_13, port_in_14, port_in_15)
   begin
      if(address >= x"00" and address <= x"7F") then
         data_out <= rom_out;
      elsif(address >= x"80" and address <= x"DF") then
         data_out <= ram_out;
      -- Input Routing
      elsif(address = x"F0") then
         data_out <= port_in_00;
      elsif(address = x"F1") then
         data_out <= port_in_01;
      elsif(address = x"F2") then
         data_out <= port_in_02;
      elsif(address = x"F3") then
         data_out <= port_in_03;
```

```
      elsif(address = x"F4") then
         data_out <= port_in_04;
      elsif(address = x"F5") then
         data_out <= port_in_05;
      elsif(address = x"F6") then
         data_out <= port_in_06;
      elsif(address = x"F7") then
         data_out <= port_in_07;
      elsif(address = x"F8") then
         data_out <= port_in_08;
      elsif(address = x"F9") then
         data_out <= port_in_09;
      elsif(address = x"FA") then
         data_out <= port_in_10;
      elsif(address = x"FB") then
         data_out <= port_in_11;
      elsif(address = x"FC") then
         data_out <= port_in_12;
      elsif(address = x"FD") then
         data_out <= port_in_13;
      elsif(address = x"FE") then
         data_out <= port_in_14;
      elsif(address = x"FF") then
         data_out <= port_in_15;
      else
         data_out <= x"00";
      end if;
   end process;

end architecture;
```

2. BÖLÜM

MERKEZİ İŞLEM BİRİMİ (CPU) BLOĞU



Şekil 2.1. CPU.vhd Mimarisi

Belleğin top modülünü tamamlamıştık. İçerisine 3 adet alt bloğuda yazıp top modülü oluşturduk ve sırada CPU ya geçmek vardı. CPU nun içine tekrar gelelim;

CPU bloğu, data-path (veri yolu) ve control-unit olarak iki bölümden oluşuyor. Control-unit birimine sonra bakacağım. Burada yaptığımız bütün blokları yöneten bir durum makinası var. Veri yolu bloğu içerisinde 5-6 adet farklı register vardı. Bunların hepsini tek tek açıklamıştım. İki adet BUS yapısı vardı. Bu BUS lar MUX aracılığı ile kontrol ediliyordu ve bu MUX ların seçimleri kontrol ünitesi tarafından sağlanıyordu. Ayrıca veri yolunda aritmetik ve matematiksel işlemlerin yapıldığı bir Aritmetic-Logic-Unit vardı (ALU), bu aslında veri yolunun alt bloğu.

2.1. Veri Yolu - Data Path

2.1.1. Aritmetik Mantık Ünitesi - Arichmetic Logic Unit (ALU)

Tasarımım da en aşağıdan yukarıya doğru ilerliyorum. Önce küçük parçaları tamamlayacağım. Bu yüzden ilk olarak veri yolu (Data-Path) bloğu altında Aritmetic-Logic-Unit yapısını inşa edeceğim.

ALU, CLOCK ve RESET sinyallerini içermiyor. Girişleri verdiğimiz anda çıkışları alıyoruz. Yani aynı saat darbesinde sonucu alıyoruz. Diyagrama bakarsak ALU nun 2 tane portu var ; A ve B

A portuna doğrudan B Kaydedicisi bağlı. B portuna ise BUS1 den gelecek veri bağlı. ALU sonucunun verildiği 8-bit lik bir Output portu var (ALU Result), aynı zamanda koşulları belirttiğimiz koşullu atlamalarda kullandığımız negatif, sıfır, overflow ve carry bilgilerinin yazıldığı 4-bit lik bir bayrak sinyali var (NZVC), burada belirtilen sinyal aslında 4-bit lik bir bayrak sinyali ; std-logic-vector ve bunun

- 3. Biti negatif bilgilisini
- 2. biti 0 mı değil mi bilgisini
- 1. biti overflow var mı yok mu bilgisini
- 0. biti en düşük anlamlı biti ise carry biti var mı bilgisini içeriyor.

Bunun ALU içerisinde ayarlanması gerekiyor. Ve bu bilgi daha sonra koşul kaydedicisine gidecek. Buna göre portlarımızı ve işlemlerimizi tamamlayalım. Ayrıca ALU yapısına CPU nun kontrol ünitesi tarafından görüldüğü üzere 3-bitlik bir seçim sinyali veriliyor. Bu ALU içindeki yapılacak işlemin türünü belirliyor.

ALU da yapılacak işlemlerin hepsi burada ;

Tablo 2.1. Tüm ALU Komutları

ADD_AB	x"42"
SUB_AB	x"43"
AND_AB	x"44"
OR_AB	x"45"
INCA	x"46"
INCB	x"47"
DECA	x"48"
DECB	x"49"

Burada gördüğümüz komutlara göre ALU içerisine işlemleri implemente edeceğim. A ve B portunu tanımlayarak başladım. Daha sonra kontrol ünitesi tarafından verilecek "selection" seçim sinyalini yazdım; Bu işlemin türünü belirtecekti ve 3 bit, o yüzden "2 down to 0" diyoruz.

Output sinyallerine geçtiğimde iki tane output sinyali vardı, NZVC bilgi sinyalleri (bayraklar), her biri birer bit olduğu için bunların her birinin ayrı bir anlamı var, o yüzden her bir bit için bunun vektörü tanımlı olduğundan 4-bit lik bir sinyal ("3 down to 0"). Aynı zamanda bizim sonucumuzu vereceğimiz ALU-result çıkışımız var, bu da tabii ki 8-bit lik.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

entity ALU is
   port(
         A       : in std_logic_vector(7 downto 0); -- Signed
         B       : in std_logic_vector(7 downto 0); -- Signed
         ALU_Sel    : in std_logic_vector(2 downto 0); -- Islem turu
         -- Output:
         NZVC    : out std_logic_vector(3 downto 0);
         ALU_result : out std_logic_vector(7 downto 0)
```

```
   );
end ALU;
...
```

Mimariye geçiyorum. Burada birkaç sinyal tanımlayacağız. Bunun sebebi NZVC bayraklarını bulurken OUTPUT değil de içeride kullanacağımız işlem sinyallerinden bu işlemleri halletmek istiyorum. Mesela öncelikle sonuçta carry olduğunu anlamak için 9-bit lik bit sonuç sinyali tanımlayacağım ki carry varsa normalde 8-bit olması gereken işlem sonucu 9-bit olmuşsa ben burada carry var diyeceğim. O yüzden şöyle diyelim ;

- *Signal sum-unsigned : std-logic-vector(8 downto 0);*

Daha sonra ALU sonucunu içeride tutacağım bir kaydedici tanımladım. Yine aynı şekilde 8-bit olacak.

Daha sonra hem toplama hem çıkarma için overflow ları tespit edeceğim birer sinyal tanımlıyorum.

Ve sinyal tanımlamalarım bu kadar.

```
...
architecture arch of ALU is

signal sum_unsigned : std_logic_vector(8 downto 0); -- Carry var mi gormek icin
signal alu_signal : std_logic_vector(7 downto 0);
signal toplama_overflow : std_logic; -- Overflow var mi gormek icin (Toplamada)
signal cikarma_overflow : std_logic; -- Carry var mi gormek icin (Cikarmada)

begin
...
```

Şimdi ALU tanımına geçebilirim. Process, "combinational" da olduğu için carry de sadece kullanılacak sinyalleri tanımlayacağım sensivity listte. ALU-Sel sinyali işlemin türünü belirticek.

İçeride kullanacağımız sinyalleri sıfırlayacağım. Daha sonra seçime göre işlemleri tek tek tanımlayacağım;

Eğer ALU-sel 0 ise, (artık komutlardan takip edelim -> ADD-AB den başlıyorum

(A = A + B)) bu toplama işlemi olarak tanımlansın.

Sonucu "alu-signal" sinyalinde tutucaz, OUTPUT daha sonradan atanacak. Bir de bunun carry olup olmadığını anlamamız için 9-bit lik bir toplamasını yapalım. Başına 9-bit olması için 0 padleyerek bu sayıları topluyorum. Yine aynı şekilde hiçbir farkı yok, tek farkı başına "0" padledim. Eğer overflow varsa 9-bitlik sonucun en yüksek anlamlı bitin overflow bitini taşıyacak ve bende anlamış olacağım. Yani burada "C" değerini set etmek için bu sinyal var.

Çıkarma işlemi için 001 dedik.

Sonraki işlemimiz AND operasyonu ki bu 010. Burada overflow tehlikesi yok, çünkü mantık operasyonlarında bit sayısı korunuyor. AND işlemi bit bazında yapılıyor, o pozisyondaki bit ile AND leniyor, o yüzden asla değişmez, overflow olmaz.

Sonrasında OR işlemi var sırada, 011,

Burada bir increase operasyonu var "100". Yani sayıyı 1 arttırıyor. Bu da sadece A da tanımlı. Burada 8-bit uzun uzun yazmamak için sadece Hex olarak 01 yazdım yani A register ına 1 eklemiş olduk böylece.

Bir de decrease diyelim ; yani sayıyı 1 azalt "101",

Bunu tekrar "B" Kaydedicisi için yapmama gerek yok çünkü zaten A portuna B bağlanarak "100" selection sinyaline bu işlem yaptırılabilir, o yüzden gereksiz yere tekrar yapmamıza gerek yok.

"when others" diyoruz ve burada da işlem sinyallerimizi sıfırlayalım.

```
...
process(ALU_Sel, A, B)
begin
   sum_unsigned <= (others => '0'); -- reset parameter

   case ALU_Sel is
      when "000" => -- Toplama
         alu_signal <= A + B;
         sum_unsigned <= ('0' & A) + ('0' + B);

      when "001" => -- Cikarma
         alu_signal <= A - B;
         sum_unsigned <= ('0' & A) - ('0' + B);
```

```
      when "010" => -- AND
         alu_signal <= A and B;

      when "011" => -- OR
         alu_signal <= A or B;

      when "100" => -- +1 Arttir
         alu_signal <= A + x"01";

      when "101" => -- -1 Azalt
         alu_signal <= A - x"01";

      when others =>
         alu_signal <= (others => '0');
         sum_unsigned <= (others => '0');
   end case;

end process;
...
```

Evet, bu kadar. Yani bu şekilde ALU nun "result" kısmını yapmış olduk. Şimdi sırada NZVC kısmını set etmek var. Bundan önce, burada ki sinyalde tuttuğumuz veriyi output a verelim. Process içerisinde ki hesaplanan sinyal doğrudan çıkışa veriliyor. Ve ben içeride bu sinyali kullanabilirim. Yani sonucu önce bir ara sinyalde tutup daha sonra ayrı şekilde Output a atarım ve bu sonucu ileri ki işlemlerimde kullanabilirim.

NZVC : (Negative, Zero, OverFlow, Carry)

• N : Öncelikle negatif flag ini set edelim. Şimdi öncelikle "N" deyiz. Negatif olup olmadığı nasıl anlaşılır? Sayılar signed olduğu için sonucumuzun en yüksek anlamlı biti "1" ise bu sayı negatiftir.

Negatifi set etmek için NZVC 4-bit lik sinyalinin 3. Bitine erişmem lazım. ALU-signal yani ALU nun sonucunun en yüksek anlamlı bitini. Eğer sayı en yüksek anlamlı biti "1" ise N=1 olacak, "0" ise N=0 olacak.

• Z : Bu da çok bariz. Bu sefer NZVC sinyalinin 2. Biti yani Z seçili. Eğer ALU sinyali "0" a eşit ise Z=1, değilse Z=0.

• V : Şimdi overflow a bakalım. Overflow iki durumda olabilir, bir toplamada

bir çıkartmada. Örneğin toplama işleminin overflow durumunu kontrol ediyoruz. Bir sayının toplamada overflow olabilmesi için toplamaya giren sayıların işareti ile sonucun işaretinin birbirine zıt olması lazım. Bunu da şöyle belirtiriz : A parametresinin işareti "0" ise ve B parametresinin işareti "0" ise ve sonucumuzun işareti "1" ise burada overflow var. Ya da Tam tersi ; A nın işareti "1", B nin işareti "1", sonucun işareti "0" ise yine overflow var demektir.

toplama-overflow <= (not(A(7)) and not(B(7)) and alu-signal(7)) or (A(7) and B(7) and not(alu-signal(7)));

cikartma-overflow <= (not(A(7)) and B(7) and alu-signal(7)) or (A(7) and not(B(7)) and not(alu-signal(7)));

Şimdi bunun çıkartma için olanını yapalım; Burada A-B dediğimiz için B nin "-" işaretinden dolayı işareti değişecek, o yüzden değişen tek şey burada B nin ifadesi olacak. Bu yüzden A "0", B "1" iken sonuç "1" ise overflow var demektir. Ya Da A "1", B "0" iken sonuç "0" ise overflow var demektir. Yani mantık tamamen toplama ile aynı, sadece şu "-" işaretinden dolayı B nin önündeki, işaretin tam zıttını yapmamız lazım. Bunları belirttikten sonra artık V yi set edebiliriz.

Yani işlemimiz toplama ise overflow bilgisi toplama-overflow da. Eğer işlemimiz çıkartma işlemi ise ve ALU-Sel 001 ise overflow bilgisi cikarma-overflow da. Değilse overflow = 0 dır. Bu şekilde.

• C : Carry en küçük anlamlı 0. Bite eşitleyeceğiz. Çünkü C orada. "sun-unsigned" sinyalinin en yüksek anlamlı bitine bakacağım yani 8. Bit pozisyonuna, ve bit pozisyonunun değeri ne ise toplama ve çıkartma işlemine göre bu ifadeyi C ye eşitleyeceğim.

Yani aslında toplama ve çıkartma burada farketmiyor, ve eğer hiç biri değilse de "C" değerine "0" verelim. Böylelikle hem ALU result değeri "alu-signal" aracılığı ile bulunmuş oldu. ALU nun yapması gereken bütün işlemleri burada tanımladık. Bu yapılacak işlem CPU tarafından verilecek select sinyali ile belirleniyor dedik. Ve daha sonra bizim bu sonuca göre NZVC bilgi sinyallerini tek tek burada implemente ettik. ALU muz hazır, artık veri yolunu kodlayabiliriz.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

entity ALU is
   port(
         A        : in std_logic_vector(7 downto 0); -- Signed
         B        : in std_logic_vector(7 downto 0); -- Signed
         ALU_Sel    : in std_logic_vector(2 downto 0); -- Islem turu
         -- Output:
         NZVC     : out std_logic_vector(3 downto 0);
         ALU_result : out std_logic_vector(7 downto 0)
   );
end ALU;

architecture arch of ALU is

signal sum_unsigned : std_logic_vector(8 downto 0); -- Carry var mi gormek icin
signal alu_signal : std_logic_vector(7 downto 0);
signal toplama_overflow : std_logic; -- Overflow var mi gormek icin (Toplamada)
signal cikarma_overflow : std_logic; -- Carry var mi gormek icin (Cikarmada)

begin

process(ALU_Sel, A, B)
begin
   sum_unsigned <= (others => '0'); -- reset parameter

   case ALU_Sel is
      when "000" => -- Toplama
         alu_signal <= A + B;
         sum_unsigned <= ('0' & A) + ('0' + B);

      when "001" => -- Cikarma
         alu_signal <= A - B;
         sum_unsigned <= ('0' & A) - ('0' + B);

      when "010" => -- AND
         alu_signal <= A and B;

      when "011" => -- OR
         alu_signal <= A or B;

      when "100" => -- +1 Arttir
         alu_signal <= A + x"01";

      when "101" => -- -1 Azalt
         alu_signal <= A - x"01";
```

```
      when others =>
         alu_signal <= (others => '0');
         sum_unsigned <= (others => '0');
   end case;

end process;

ALU_result <= alu_signal;

--- NZVC (Negatif, Sifir, Overflow, Carry)

--N:
NZVC(3) <= alu_signal(7);

--Z:
NZVC(2) <= '1' when (alu_signal = x"00") else '0';

--V:
toplama_overflow <= (not(A(7)) and not(B(7)) and alu_signal(7)) or (A(7) and B(7) and
cikarma_overflow <= (not(A(7)) and B(7) and alu_signal(7)) or (A(7) and not(B(7)) and

NZVC(1) <= toplama_overflow when (ALU_Sel = "000") else
        cikarma_overflow when (ALU_Sel = "001") else '0';

--C:
NZVC(0) <= sum_unsigned(8) when (ALU_Sel = "000") else
        sum_unsigned(8) when (ALU_Sel = "001") else '0';

end architecture;
```

2.1.2. Veri Yolu ve Kaydediciler

ALU nun tasarımını tamamladık, artık CPU.vhd diyagramında görmüş olduğumuz "data-path" bloğunu, BUS ları da dahil olmak üzere tanımlayabiliriz. ALU birimini burada komponent olarak tanımlayacağız, burada ki farklı registerları tek tek tanımlayacağız ve bunların nasıl çalıştığını modelleyeceğiz. Buradaki BUS yapılarını tanımlayacağız. Ve ilgili giriş çıkışları vereceğiz, örneğin alttaki "to-memory" ve "from-memory" sinyalleri aslında veri yolunun portları. Ve aynı zamanda kontrol ünitesi ile haberleştiği bağlantılar var. Bunların hepsini bu data-path modülünde belirtmemiz lazım.

Entity adına "data-path" dedim. Şimdi Input sinyallerimiz neler? Öncelikle "CLOCK" ve "RESET" sinyallerimiz var. Daha sonra yukarıdan başlayıp

diyagramdaki okları takip ederek bütün input sinyallerini tek tek yazacağız. IR-Load (Instruction Register Load) yani komut kaydedicisine yükleme yap emri. Bu bir kontrol sinyali 1-bit lik. Data-path.vhd diyagramında gördüğümüz IR aktifleştirecek.

IR-Load dan sonra MAR-Load (Memory Access Register Load) var. Daha sonra PC-Load (Program Counter Load) ve PC-inc (Program Counter Increase) var. PC-Load, program sayacını yüklemek için kullanılıyor. PC-inc ise program sayacının değerini "1" arttırmak için kullanılıyor.

Daha sonra A-Load ve B-Load var. Bunlarda A ve B registerlarına yükleme için kullanılıyor.

Daha sonra, burada ALU-Sel var. Önceki çalışmada kodladık. ALU da yapılacak işlemi seçen sinyaldi. Ve bu sinyal inputlarla uğraştığımızdan ve bu sinyal de data-path bloğuna input olarak girdiğinden bunu da burada alıyoruz.

CCR-Load, yine data-path e gelen bir input sinyali.

Bus1-Sel ve Bus2-Sel var, bunlar BUS yapılarına hangi sinyalin sürüleceğini kontrol eden kontrol sinyalleri ve görüldüğü üzere bunlar 2-bit sinyaller. Sebebi ise 3 farklı inputtan birini seçecek olmaları.

Ve burada çok önemli bir input sinyalimiz bellekten gelen verinin iletildiği "from-memory" sinyali. Bu bellek ile CPU arasındaki bağlantı. Yani "from-memory". TOP-LEVEL Diagram a bakacak olursak "memory" den "CPU" ya giden "data-out" sinyali görülüyor. Bunun CPU daki karşılığı "from-memory". Ve "from-memory" sinyali "data-path" modülünde 8-bitlik bir sinyal.

Input sinyallerimiz bu kadar. OUTPUT sinyallerine gelecek olursam.

Komut register ının çıkışı (IR). Daha sonra "address" var, bu da dışarı verilen bir sinyal. Biraz diyagramın aşağı taraflarında gördüğümüz Condition Code Register (CCR) ın sonucu var. Bu 4-bit lik bir sinyal çünkü NZVC den oluşuyor bu sadece. ALU.vhd ye bakarsak ta 4-bit olduğunu görüyoruz zaten.

Belleğe giden "to-memory" sinyali var. "address" sinyali belleğe giden adres bilgisi, "to-memory" ise belleğe giden veri. Yine en TOP modüle bakacak olursak bellekte

yani “memory.vhd” deki “data-in” sinyaline denktir.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

entity data_path is
   port(
         clk   : in std_logic;
         rst   : in std_logic;
         IR_Load : in std_logic; -- Komut register'i yukle kontrol
         MAR_Load : in std_logic;
         PC_Load : in std_logic;
         PC_Inc : in std_logic;
         A_Load : in std_logic;
         B_Load : in std_logic;
         ALU_Sel : in std_logic_vector(2 downto 0);
         CCR_Load : in std_logic;
         BUS1_Sel : in std_logic_vector(1 downto 0);
         BUS2_Sel : in std_logic_vector(1 downto 0);
         from_memory : in std_logic_vector(7 downto 0);
         -- Outputs:
         IR       : out std_logic_vector(7 downto 0);
         address    : out std_logic_vector(7 downto 0); -- bellege giden adres bilgisi
         CCR_Result : out std_logic_vector(3 downto 0); -- NZVC
         to_memory  : out std_logic_vector(7 downto 0) -- bellege giden veri

   );
end data_path;
...
```

Ve bu şekilde entity kısmını halletmiş oldum. Architecture tanımında ise her şeyden önce ALU yapısını component olarak tanımlamam lazım veri yolunun bir alt bloğu olduğu için.

ALU tanımladıktan sonra data path de kullanılacak sinyalleri tanımlamalıyım. “Veri yolu ic sinyalleri” diyelim. Her şeyden önce BUS yapılarımız var. Bu BUS lar 8-bit lik birer kablo aslında, verilerin iletilmesinde görevliler. Bunların sinyal olarak tanımlanması lazım.

Signal BUS1 : std-logic-vector(7 downto 0);

Aynı şekilde BUS2 yi de tanımlayacağım. Sonrasında Diyagrama tekrar geldiğimde burada ALU da sonuç sinyali var ve bu MUX a bağlı. Bu kabloyu da “result” portuna

bağlamamız lazım. Buna da ;

Signal ALU-result : std-logic-vector(7 downto 0);

Aynı ismi veriyorum ki bağlanacakları belli olsun. Bu da 8-bit lik bir sinyal. Tekrar diyagramdan devam edecek olursam burada kaydediciler görülüyor, Bu kaydediciler 8-bit, her birini sinyal olarak tanımlamam lazım. Öncelikle komut register ından başlayalım;

Signal IR-Reg : std-logic-vector(7 downto 0); ...

Sonrasında CCR kaydedicisini de tanımlamam lazım. Buna önce ALU dan NZVC nin bağlanacağı kabloyu tanımlayalım yani CCR a giriş yapacak olan 4-bit lik kablo. Bunun adı "CCR-in". Bir de CCR ın blok olarak kendisini tanımlamam lazım, buna da CCR diyeceğim ve bu da 4-bit.

```
...
architecture arch of data_path is

-- ALU:
component ALU is
   port(
         A       : in std_logic_vector(7 downto 0); -- Signed
         B       : in std_logic_vector(7 downto 0); -- Signed
         ALU_Sel    : in std_logic_vector(2 downto 0); -- Islem turu
         -- Output:
         NZVC    : out std_logic_vector(3 downto 0);
         ALU_result : out std_logic_vector(7 downto 0)
   );
end component;

-- Veri yolu ic sinyalleri:
signal BUS1       : std_logic_vector(7 downto 0);
signal BUS2       : std_logic_vector(7 downto 0);
signal ALU_result : std_logic_vector(7 downto 0);
signal IR_reg     : std_logic_vector(7 downto 0);
signal MAR        : std_logic_vector(7 downto 0);
signal PC         : std_logic_vector(7 downto 0);
signal A_reg      : std_logic_vector(7 downto 0);
signal B_reg      : std_logic_vector(7 downto 0);
signal CCR_in     : std_logic_vector(3 downto 0);
signal CCR          : std_logic_vector(3 downto 0);
begin
...
```

Mimari tanımına geçiyorum. Her şeyden önce diyagramda görülen BUS MUX yapılarını modelleyelim ki (zaten görünen 2 tane var) hangi durumda BUS veri yoluna ne sürülecek en başta belli olsun.

Sağ üstteki MUX yapısındayım, eğer CPU dan, kontrol ünitesinden gelen BUS1 seçimi ;

• "00" sa BUS1 e Program Counter (PC),

• "01" se A Kaydedicisi

• "10" ise B Kaydedicisi sürülüyor.

• Hiç biri değilse "0" sürelim.

Burada "IF" üzerinden koştuğum koşullarda "A-Reg" "B-Reg" diyelim ki ALU ile karışmasın.

Ve aynı şekilde BUS2 yi de tanımlamamız lazım. Sol alttaki MUX tayız. BUS2-Sel Eğer

• "00" sa ALU-Result

• "01" se BUS1 sürülüyor, yani BUS1 ile BUS2 arasında ki bağlantı sağlanmış oluyor.

• "10" sa "from-memory", bellekten gelen veri sürülüyor.

```
...
-- BUS1_Mux:
   BUS1 <= PC   when BUS1_Sel <= "00" else
         A_reg when BUS1_Sel <= "01" else
         B_reg when BUS1_Sel <= "10" else (others => '0');

-- BUS2_Mux:
   BUS2 <= ALU_result when BUS2_Sel <= "00" else
         BUS1     when BUS2_Sel <= "01" else
         from_memory when BUS2_Sel <= "10" else (others => '0');
...
```

BUS ları tanımladık. Şimdi burada ki veri yoluna ait kaydedicileri tek tek modelleyeceğiz. Komut register ından başlıyorum (IR).

Kaydedici olduğu için “Clock” ve “Reset” sinyalleri olmak zorunda, “Reset” anında kaydediciyi sıfırlayacağım, “Clock” ta kontrol kaydedicisi tarafından IR-Load emri verilirse BUS2 yapısından kaydediciye bir input giriliyor, BUS2 verisi komut kaydedicisine atanacak. Yani eğer kontrol ünitesi, yükle emri verdi ise IR kaydedicimiz BUS2 deki veriyi alacak. Bu sayede kaydedicinin içeriğini güncellemiş olacağım.

```
...
-- Komut Register (IR)
   process(clk, rst)
   begin
      if(rst = '1') then
         IR_reg <= (others => '0');
      elsif(rising_edge(clk)) then
         if(IR_Load = '1') then
            IR_reg <= BUS2;
         end if;
      end if;
   end process;
   IR <= IR_reg;
...
```

Aynı şekilde “memory access register” (MAR) kaydedicisini yapalım. MAR kaydedicisine BUS2 nin değeri atanacak, aynı zamanda bu MAR kaydedicisine belleğe bir adres Outputu veriyordu ve bu Outputu da zaten entity kısmında tanımlamıştık veri yolunda. (bu kodun içindeki veriyolu entitysi) Bunu da burada eşitleyelim. Process içerisinde güncellenen MAR değeri anında (çünkü process dışında bu) adrese verilmiş/sürülmüş olacak.

```
...
-- Memory Access Register (MAR)
   process(clk, rst)
   begin
      if(rst = '1') then
         MAR <= (others => '0');
      elsif(rising_edge(clk)) then
         if(MAR_Load = '1') then
            MAR <= BUS2;
         end if;
      end if;
   end process;
   address <= MAR;
...
```

Şimdi program counter dayız (PC), Reset "1" ise PC yi sıfırlayalım, eğer clock var ise PC-Load da "1" ise PC ye BUS2 deki veriyi vereceğiz. Bakın birde Program Counter Increase yani program sayacını arttır komutu vardı, onuda buraya yazmam lazım.

```
...
-- Program Counter (PC)
   process(clk, rst)
   begin
      if(rst = '1') then
         PC <= (others => '0');
      elsif(rising_edge(clk)) then
         if(PC_Load = '1') then
            PC <= BUS2;
         elsif(PC_Inc = '1') then
            PC <= PC + x"01";
         end if;
      end if;
   end process;
...
```

PC (Program Counter) da bitti, A ve B kaydedicilerini yapalım hemen. Bunların adını X-Reg olarak verdik! Eğer A-Load gelirse A Kaydedici BUS2 deki veriyi koy diyoruz. B tamamen aynı şekilde sadece isimleri değişecek.

```
...
-- A Register (A_reg)
   process(clk, rst)
   begin
      if(rst = '1') then
         A_reg <= (others => '0');
      elsif(rising_edge(clk)) then
         if(A_Load = '1') then
            A_reg <= BUS2;
         end if;
      end if;
   end process;

-- B Register (B_reg)
   process(clk, rst)
   begin
      if(rst = '1') then
         B_reg <= (others => '0');
      elsif(rising_edge(clk)) then
         if(B_Load = '1') then
            B_reg <= BUS2;
         end if;
```

```
    end if;
  end process;
...
```

CCR kaldı, ondan önce ALU nun Port-Map ini yapalım. Komponent olarak tanımladık ama portlarını bağlamadık. Port-Map yapacağız.

Bloğun ismi ALU idi, port map dedik. Komponentteki portları aldık. Şimdi bu portlara bağlantıları yapacağız. A portuna B kaydedicisinin kendisi bağlı, o yüzden A ya B kaydedicisini bağlıyoruz. Hiçbir şeyi ezbere yapmıyoruz yani diyagrama bakarak bunu söylüyoruz. B kaydedicisine BUS1 bağlı, BUS1 deki veriyi anında alacak demek. ALU-Sel sinyaline normal ALU-Sel bağlı. Bunu input olarak zaten veri yoluna tanımlamıştık port olarak.

Çıkış portlarına gelecek olursak; NZVC bir çıkış portu, bunlar outputlar. CCR a input olarak gidecek CCR-in sinyalini verelim ve burada tutulsun bu değer.

```
...
-- ALU
ALU_U: ALU port map
        (
          A       => B_reg,
          B       => BUS1,
          ALU_Sel    => ALU_Sel,
          -- Output:
          NZVC    => CCR_in,
          ALU_result => ALU_result
        );
```

ALU-Result için ise zaten öyle bir sinyal tanımlamıştık. İlgili output değerleri bu 2 sinyalde tutulacak ve biz bunları kullanabiliriz. CCR artık kalan son bloğumuz. Veri yolunu tamamladık. B-Reg den kaydedici process bloğunu kopyalayalım process aynı process. Reset gelise CCR sıfırlanacak.

Eğer CCR-Load gelirse kaydediciye NZVC den gelen değer gelecek. Bizim bloğumuzda bu CCR-in de tutuluyordu. Böylelikle ALU deki Flagleri almış olduk. Aynı zamanda bu kaydedicinin bir çıkışı var, yani burada ki değer kontrol ünitesine geri veriliyor. Bu sinyalin adı da CCR-Result. Onu da Output larda tanımlamıştık zaten.

Aynı şekilde burada MAR da "adres" i tanımladığımız, çıkışa verdiğimiz gibi bunu da çıkışa verelim. CCR-Result, CCR değerini çıkışa vermiş olacak.

```
...
-- CCR Register
   process(clk, rst)
   begin
      if(rst = '1') then
         CCR <= (others => '0');
      elsif(rising_edge(clk)) then
         if(CCR_Load = '1') then
            CCR <= CCR_in; -- NZVC flag bilgisi
         end if;
      end if;
   end process;
   CCR_Result <= CCR;
...
```

Tek bir işlemimiz kaldı: "Data-path den belleğe gidecek sinyal ataması":

```
...
-- Veri yolundan bellege gidecek sinyal atamasi:
   to_memory <= BUS1;
...
```

"to-memory" BUS1 den gidecek sinyal. Her zaman BUS1 veri yoluna bağlı. Diyagramda/sistemde herhangi bir kontrol mekanizması yok. BUS1 değeri ne olursa olsun, hangi değer sürülmüşse sürülsün bu her zaman dışarıda bir "to-memory" portuna bağlı. Yani ben bu veri yolundaki output olarak tanımlanmış "to-memory" sinyaline her zaman BUS1 değerini süreceğim. Ve böylece belleğe gidecek veri yolunuda tanımlamış oldum.

Böylece Data-Path bloğumuz tamamlanmış oldu;

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

entity data_path is
   port(
         clk   : in std_logic;
         rst   : in std_logic;
         IR_Load : in std_logic; -- Komut register'i yukle kontrol
         MAR_Load : in std_logic;
```

```
        PC_Load : in std_logic;
        PC_Inc : in std_logic;
        A_Load : in std_logic;
        B_Load : in std_logic;
        ALU_Sel : in std_logic_vector(2 downto 0);
        CCR_Load : in std_logic;
        BUS1_Sel : in std_logic_vector(1 downto 0);
        BUS2_Sel : in std_logic_vector(1 downto 0);
        from_memory : in std_logic_vector(7 downto 0);
        -- Outputs:
        IR      : out std_logic_vector(7 downto 0);
        address    : out std_logic_vector(7 downto 0); -- bellege giden adres bilgisi
        CCR_Result : out std_logic_vector(3 downto 0); -- NZVC
        to_memory  : out std_logic_vector(7 downto 0) -- bellege giden veri

   );
end data_path;


architecture arch of data_path is

-- ALU:
component ALU is
   port(
        A       : in std_logic_vector(7 downto 0); -- Signed
        B       : in std_logic_vector(7 downto 0); -- Signed
        ALU_Sel    : in std_logic_vector(2 downto 0); -- Islem turu
        -- Output:
        NZVC    : out std_logic_vector(3 downto 0);
        ALU_result : out std_logic_vector(7 downto 0)
   );
end component;

-- Veri yolu ic sinyalleri:
signal BUS1       : std_logic_vector(7 downto 0);
signal BUS2       : std_logic_vector(7 downto 0);
signal ALU_result : std_logic_vector(7 downto 0);
signal IR_reg     : std_logic_vector(7 downto 0);
signal MAR        : std_logic_vector(7 downto 0);
signal PC         : std_logic_vector(7 downto 0);
signal A_reg      : std_logic_vector(7 downto 0);
signal B_reg      : std_logic_vector(7 downto 0);
signal CCR_in     : std_logic_vector(3 downto 0);
signal CCR          : std_logic_vector(3 downto 0);
begin

-- BUS1_Mux:
   BUS1 <= PC   when BUS1_Sel <= "00" else
        A_reg when BUS1_Sel <= "01" else
        B_reg when BUS1_Sel <= "10" else (others => '0');
```

```
-- BUS2_Mux:
   BUS2 <= ALU_result when BUS2_Sel <= "00" else
         BUS1     when BUS2_Sel <= "01" else
         from_memory when BUS2_Sel <= "10" else (others => '0');

-- Komut Register (IR)
   process(clk, rst)
   begin
      if(rst = '1') then
         IR_reg <= (others => '0');
      elsif(rising_edge(clk)) then
         if(IR_Load = '1') then
            IR_reg <= BUS2;
         end if;
      end if;
   end process;
   IR <= IR_reg;

-- Memory Access Register (MAR)
   process(clk, rst)
   begin
      if(rst = '1') then
         MAR <= (others => '0');
      elsif(rising_edge(clk)) then
         if(MAR_Load = '1') then
            MAR <= BUS2;
         end if;
      end if;
   end process;
   address <= MAR;

-- Program Counter (PC)
   process(clk, rst)
   begin
      if(rst = '1') then
         PC <= (others => '0');
      elsif(rising_edge(clk)) then
         if(PC_Load = '1') then
            PC <= BUS2;
         elsif(PC_Inc = '1') then
            PC <= PC + x"01";
         end if;
      end if;
   end process;

-- A Register (A_reg)
   process(clk, rst)
   begin
      if(rst = '1') then
```

```
            A_reg <= (others => '0');
         elsif(rising_edge(clk)) then
            if(A_Load = '1') then
               A_reg <= BUS2;
            end if;
         end if;
      end process;

   -- B Register (B_reg)
      process(clk, rst)
      begin
         if(rst = '1') then
            B_reg <= (others => '0');
         elsif(rising_edge(clk)) then
            if(B_Load = '1') then
               B_reg <= BUS2;
            end if;
         end if;
      end process;

   -- ALU
   ALU_U: ALU port map
            (
               A        => B_reg,
               B        => BUS1,
               ALU_Sel    => ALU_Sel,
               -- Output:
               NZVC     => CCR_in,
               ALU_result => ALU_result
            );

   -- CCR Register
      process(clk, rst)
      begin
         if(rst = '1') then
            CCR <= (others => '0');
         elsif(rising_edge(clk)) then
            if(CCR_Load = '1') then
               CCR <= CCR_in; -- NZVC flag bilgisi
            end if;
         end if;
      end process;
      CCR_Result <= CCR;

   -- Veri yolundan bellege gidecek sinyal atamasi:
      to_memory <= BUS1;



   end architecture;
```

Veri yolu artık tamamlandı. Her şeyi belirttik. Diyagramda görülen tüm yapıları tek tek tanımladım. Kontrol ünitesinin verdiği sinyallere göre bu kaydedicilerin davranışlarını belirledim. Geriye kontrol ünitesinde ki sonlu durum makinasını yazmak ve daha sonra TOP-MODÜL de tüm işlemciyi birleştirmek kaldı.

2.2. Kontrol Ünitesi

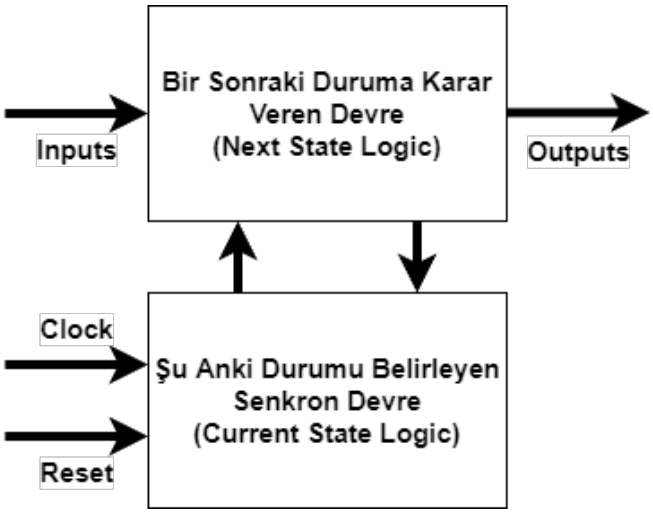
2.2.1. Kontrol Ünitesi - Durum Makinesi Mimarisi

Kontrol ünitesi içerisinde

- Fetch
- Decode
- Execute

adımlarını yürüten sonlu durum makinesi bulunur, işlemcinin tüm yönetimini bu kısım yapıyor.

Clock ve Reset e duyarlı bir process bloğu içerisinde Reset ile tüm kaydedicileri sıfırlayıp State durumu bilinen bir başlangıç noktasına getiriliyor. Reset, inaktif olduğunda ise, saatin her yükselen kenarında her bir state senaryosunda ilgili işlemleri tanımlayıp bir sonraki Clock için state değerlerini güncelleniyor.



Şekil 2.2. Next State - Current State Logic

Burada bahsi geçen sonlu durum makinesi ise farklı bir yapıda. Burada next state ve current state yani sonraki ve şimdiki durum olarak 2 adet durum sinyali görüyoruz. Şimdiki durum devresi clock lu ve senkron bir yapıda, sonraki durum devresi ise INPUT lara ve şimdiki duruma bağlı olarak çalışan Combinational bir devre.

Ayrıca burada ayrı bir block olarak belirtmesekte Next state e bağlı bir OUTPUT logic var. Bu da ilgili state deki OUTPUT ların üretilmesi için gerekli işlemlerden sorumlu. İşlemcimizde ki durum makinesi tam olarak bu yapıda. Sonraki durum ve şimdiki durum olarak 2 adet durum sinyali kullanılacak. Ayrıca 3 adet ayrı process blğu olacak.

Clock ve Reset duyarlı bir process bloğunda şimdiki durumu güncelleyeceğiz yani Current State i güncelleyeceğim. Böylece her yeni state darbesinde gerekli durum geçişleri aynı şekilde normal akışta sağlanmış.

Ancak, sonraki durum devresine ve komutun uygulandığı OUTPUT logic devresine herhangi bir clock bağlı olmayacak. INPUT lara ve şimdiki duruma bağlı process bloklarında gerekli atamaları yapacağım. Böyle bir sistem kurulmasının sebebi kontrol ünitesinin veri yoluna ve belleğe gönderdiği kontrol sinyali asenkron olmak zorunda. Yani gerekli kontrol sinyali, bir sonraki saat darbesini beklemeden anlık olarak gerekli hedefe iletilmeli.

Örneğin ALU, tamamen Combinational bir devre, INPUT ları ve seçim sinyalini verdğimiz gibi OUTPUT ları çıkarıyor. Biz bunu saat kontrolünde yönetmeye kalkarsak işlemleri düzene sokmamız, beklediğimzi zaman aralıklarında doğru dataları çekmemiz, efektif bir şekilde yapılamayacaktır. Bu nedenle gerekli sinyallerin OUTPUT logic tarafından asenkron olarak atandığı ancak STATE in bildiğimiz gibi saat darbelerine göre düzenli bir şekilde güncellendiği bir yapı kurmak bu sistem için daha uygundur. H

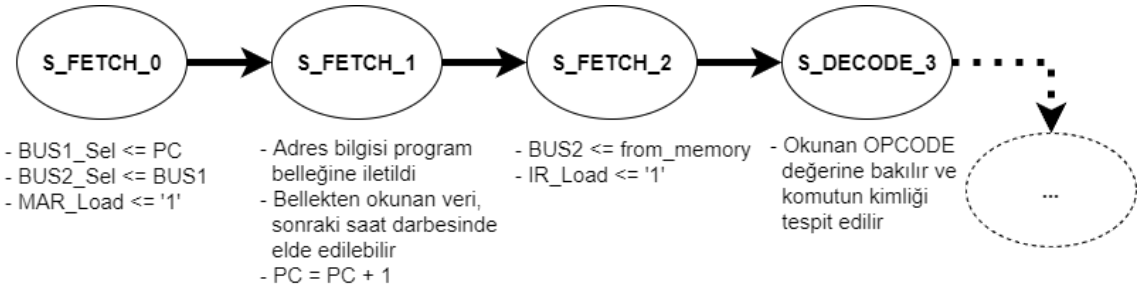
BStateleri 3 ana kategoride inceleyebiliriz;

- Fetch,
- Decode

• ve Execute

Fetch, program sayacının değerine göre program belleğinden komut okunup bu komutun CPU içerisinde ki komut kaydedicisine alınması işlemi idi. **Decode**, bu komutun kimliğinin tespit edilmesi idi, **Execute** ise tespit edilen komutun uygulanması idi. Fetch ve Decode adımlarından state leri incelemeye başlayalım.

Her ne kadar 3 adımın işlevi çok net olsa da bu adımların tamamlanması için birden fazla saat darbesi gerekiyor çünkü işin içerisinde bellekten okuma var. Bellekten okuma yapmak için adres sinyalini iletmeliyiz, bu sinyalleri belleğe ilettikten ancak 1 saat darbesi sonra veri okunabilir. Senkron bellek yapılarının hepsi için bu durum böyledir. Okuma için bir saat darbesi harcanır. Buna göre State lerimizi Fetch den başlayarak yerleştirelim.



Şekil 2.3. İşlemci Durum Makinesi - Başlangıç

S-FETCH-0 (SF0) durumunun amacı , program sayacının değerini bir şekilde belleğe iletmek. Adres iletimini ancak MAR yapabiliyordu. Bu nedenle program sayacı 0 olan mevcut başlangıç değerini MAR a iletmeli. Bunu da BUS1 aracılığı ile yapacak. Hiçbir kaydedicinin portları doğrudan birbirine bağlı değildi, çünkü tüm veri alışverişi BUS üzerinden yapılıyordu. Kısacası bu state de program sayacının değerini BUS1 e yükledik. BUS2 ye de BUS1 i sürdük ve MAR-Load değerini 1 yaptık. Böylece MAR, BUS2 deki program sayacı verisini bir sonraki saat darbesinde alabilir.

S-FETCH-1 (SF1) in başlangıcında ki saat darbesinde, memory access register ın içerisine program sayacının değeri güncellenmiş olur. Ve adres çıkışından belleğe anında adres bilgisi iletilir. Okuma ancak 1 saat darbesi sonrasında yapılacak. O nedenle bu state de boş durmamak için program sayacının değerini 1 arttırabiliriz.

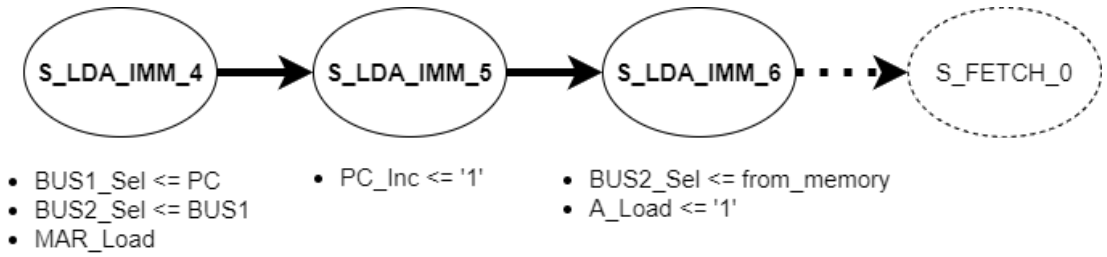
S-FETCH-2 (SF2) state inde komut belleğinden ilgili komut okundu ve from memory portundan BUS2 ye sürüldü, BUS2, bu değeri komut kaydedicisine iletmeli, bunun için IR-Load, kontrol sinyali ni 1 e eşitliyoruz. Bir sonraki saat darbesinde IR a okuduğumuz komut yazılacaktır. Tabii ki IR kaydedicisine Opcode değeri yazılıyor.

S-DECODE-3 (SD3) state inde bu Opcode okunur, komutun ne olduğuna karar verilir. Böylece komutun kimliği kesinleşmiş, kod decode edilmiş olur. Bu saydığımız adımlar tüm komutlar için ortak çünkü komuta ait Opcode un program belleğinden okunup decode edilmesi gerekli.

Bu state den sonra Execute adımını uygulamak üzere o komuta ait state lere geçiş yapılır. Durum makinesi buradan sonra içerisinde tanımlı olan komutlar oranında dallanır.

Şimdi, Implement edeceğimiz bazı komutların Decode state inden sonra Execute adımını tamamlamak üzere izleyiceği state leri inceleyelim;

2.2.1.1. YUKLE_SBT_A



Şekil 2.4. İşlemci Durum Makinesi - YUKLE_SBT_A

Yükle sabit A komutu, A register ına bir sabit sayı yüklüyordu ve bu sayı komutun Operand bilgisine taşınıyordu. Bu nedenle yine komut belleğinden operand değerinin okunması lazım. Bunun için;

S_LDA_IMM_4 : yani Load Immediate 4 state inde program sayacının değeri MAR a BUS1 ve BUS2 aracılığı ile yüklenmeli. Bunun için BUS1 e program sayacı (PC), BUS2 ye de BUS1 yükleniyor ve MAR-Load 1 yapılarak MAR a BUS2 deki veriyi al emri veriliyor. Tabii ki de bu geçiş bir sonraki saat darbesinde gerçekleşecek.

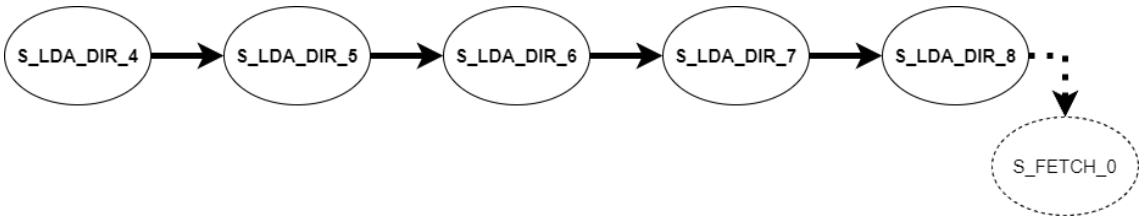
S_LDA_IMM_5 : yani Load Immediate 5 state inde MAR a program sayacının

değeri alındı, adres OUTPUT portundan bu değer komut belleğine iletilecek. Verinin okunması 1 saat darbesi sonra gerçekleşeceğinden beklemek zorundayız. Bu sırada boş durmamak için program sayacının değerini 1 arttırarak yeni bir komut okumaya hazırlayabiliriz sistemi.

S_LDA_IMM_6 : state inde bellekten veri geldi ve from memory portundan BUS2 ye sürüldü. Artık yapmamız gereken tek şey BUS2 deki veriyi A register ına koymak. Bunun için de kontrol ünitesi A-LOAD emrini veriyor ve böylece komutun amacı tamamlanmış oluyor.

Artık bir sonraki saat darbesinde A register ında Operand da yazan değer atanmış olacak. Komut tamamlandığına göre Fetch-0 başlangıç state ine geri dönüyoruz.

2.2.1.2. YUKLE_A



Şekil 2.5. İşlemci Durum Makinesi - YUKLE_A

Bu bir önceki komuttan farklı oalrak, A register ına Operand daki sayıyı değil, Operand da verilen veri belleği adresinde ki datayı yazıyordu. Operand da adres bilgisi bulunuyordu yani, bu adrese göre veri belleğinden bir okuma yapılıp okunan datayı A register ına yazmamız gerekli, bunun için;

S_LDA_DIR_4 yani Load Direct 4 adı verilen state de tıpkı bir önceki komutta yaptığımız gibi önce komut belleğinden operandı okuyarak adres bilgisini elde etmemiz lazım. Bunun için program sayacını MAR a yüklüyoruz.

S_LDA_DIR_5 te MAR a değer atanıyor ve adres portu belleğe sürülüyor, bu sırada program sayacını +1 arttırıyoruz,

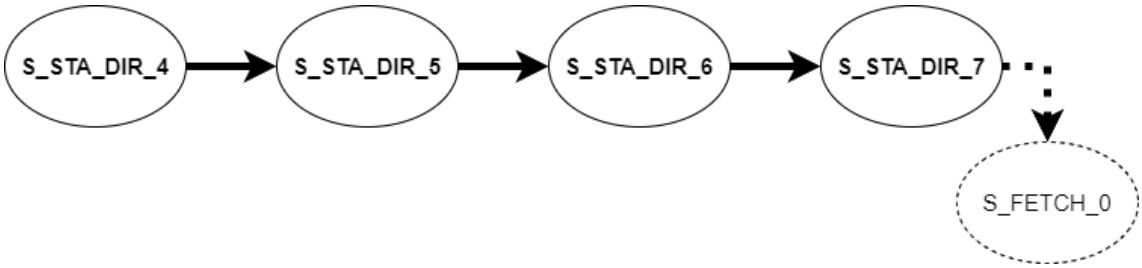
S_LDA_DIR_6 da bellekten veri geldi ve BUS2 ye sürüldü, bu veri adres bilgisi olduğundan tekrar MAR a sürmemiz lazım, bu yüzden MAR-Load tekrar aktif oldu. Bu sefer adres komut belleğini yani ROM u değil, veri belleğine RAM i işaret edicek

çünkü bellek organizasyonu hatırlarsak x"80" ile x"DF" arasında olursa veri belleğini işaret ediyordu, burada da Operand da böyle bir değer gelmek zorunda.

S_LDA_DIR_7 state inde MAR da ki adres değeri anca güncellendi ve adres OUTPUT portu bellek yapısına sürüldü, burada yapacak bir işimiz yok, program sayacında okumadan sonra zaten arttırmıştık, bu nedenle hiçbir şey yapmadan boş bekliyoruz.

S_LDA_DIR_8 bir sonraki saat darbesinde geldiğimiz state oluyor. Veri belleği üzerine sürülen adreste bulunan veriyi BUS2 üzerinden kontrol ünitesine CPU ya iletti, bu veriyi A register ına koyabilmek için A-LOAD register ını 1 yaptık ve işlem tamamlandı.

2.2.1.3. KAYDET_A



Şekil 2.6. İşlemci Durum Makinesi - KAYDET_A

A register ındaki veriyi, veri belleğine yazar. Bunun için komutun operandından belleğe kayıt yapılacak adresin bilgisi alınmalıdır.

S_STA_DIR_4 ten (state direct 4) S_STA_DIR_6 ya kadar bir önceki komut ile tamamen aynı şeyleri yapıyoruz. Komut belleğinden operand ı okuyoruz ve buradad yazan adres bilgisini

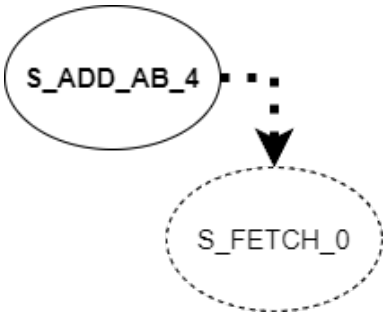
S_STA_DIR_6 da tekrar MAR a yüklüyoruz.

S_STA_DIR_7 de (Store A Direct 7) de önemli, bu state in başında MAR değeri anca güncellendi ve adres portu yazım yapılacak veri belleğine sürüldü, bununla beraber BUS1 e A register ı sürülüyor çünkü bu kaydedilecek datanın ta kendisi.

Ayrıca kontrol ünitesi tarafından Write-Enable sinyali 1 yapılarak yaz emri veriliyor.

Böylece adres, veri ve yaz emri, aynı anda belleğe iletilmiş oluyor. Bir sonraki saat darbesinde işlemi tamamlanacaktır. Komut tamamlandığı için S_FETCH_0 başlangıç state ine geri dönüyoruz.

2.2.1.4. TOPLA_A



Şekil 2.7. İşlemci Durum Makinesi - TOPLA_A

Topla AB komutu, ALU da gerçekleşiyordu ve A register ı ile B register ında ki sayıları toplayıp sonucu tekrar A register ına yazıyordu. Bu işlemi tek state ile halledebiliyoruz çünkü bu işlem için Operand bilgisine ihtiyaç yok, tüm veriler zaten registerlarda,

Ayrıca ALU, full kombinasyonel bir devre, portlarında sinyalleri sürüldüğü anda, aynı saat darbesi içerisinde tüm sonuçları elde edebiliriz. Bizim sadece toplama işlemi için gerekli olan sinyalleri portlara iletmemiz ve aritmetik lojik ünite operasyonun bilgisini iletmemiz lazım.

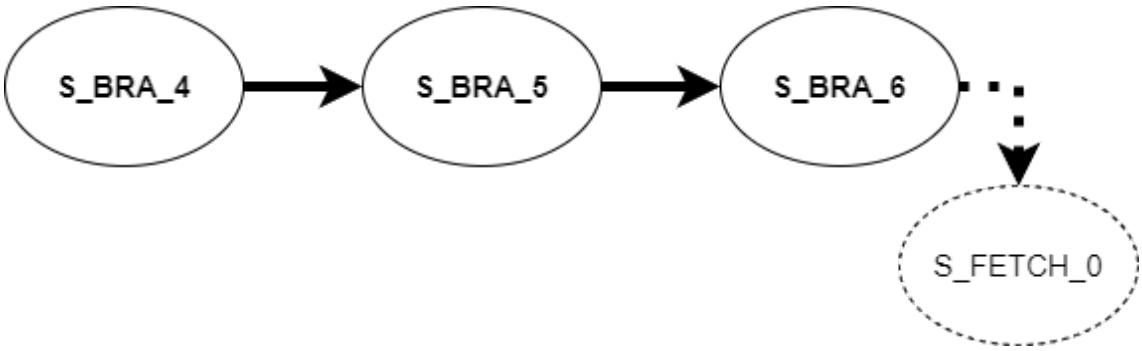
ALU nun A portu B register ına bağlıydı, B portu ise BUS1 e bağlıydı, bunu mimari şemasında görebiliriz. Yani B register ı zaten ALU ya bağlı, bizim BUS1 üzeriinden A register ını ALU ya iletmemiz lazım.

S_ADD_AB_4, state inde bunun için BUS1 e A register ını yüklüyoruz, BUS2 ye ise ALU sonucunu sürüyoruz ki sonuç ilgili yere BUS üzerinden iletilebilsin.

ALU select seçim sinyalini toplama işlemine ayarlıyoruz ki bu değer de 3 bit 000 değeriydi. A-Load sinyali 1 yapılıyor çünkü BUS2 de duran ALU sonucu tekrar A register ına alınaacak. CCR-LOAD sinyalini de aktifleştirerek negatif, zero ve carry flaglarini CCR register ının korunmasını sağlıyoruz böylece tüm gerekli işlemleri 1

adımda tamamlamış olduk.

2.2.1.5. ATLA



Şekil 2.8. İşlemci Durum Makinesi - ATLA

komutu geldiğinde hiçbir koşula bakılmaksızın program sayacı Operand da belirtilen değere atlıyordu. Bunun için Branch 4 state inde;

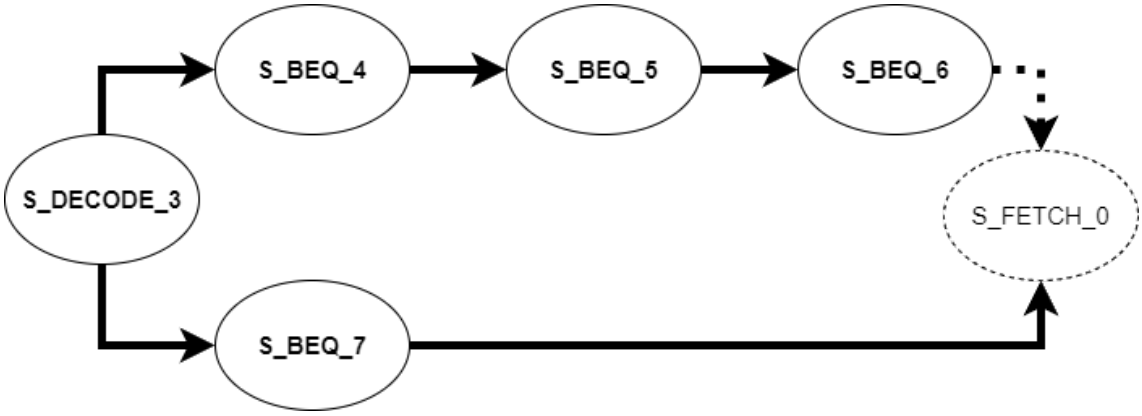
S_BRA_4, stateinde yani, Operandı okumak için gerekli lan komut belleği adres bilgisini MAR a yüklüyoruz. Diğer komutlar ile tamamen aynı şekilde.

S_BRA_5 state inde normalde program sayacının değerini 1 arttırıyorduk adresin yüklenmesini beklerken, ancak burada program sayacı operandın değerine göre zaten güncelleneceği için bunu yapmamıza gerek yok. Bu state de adres register ının adresi iletmesini bekliyoruz sadece.

S_BRA_6 state inde komut belleğinde operand okundu ve BUS2 ye sürüldü. Bu operand da atlanacak program sayacı değeri yazıyordu. Yapmamız gereken tek şey program sayacı load yani PC-load sinyalini 1 e çekerek BUS2 de bulunan değeri PC register ına almak. Böylece komut tamamlanmış oldu.

2.2.1.6. ATLA_ESITSE_SIFIR

son olarak komutuna bakalım. Bu komut aritmetik lojik sonucu 0 ise atlama yapıyordu. değilse normal şekilde sıralı çalışmasına devam ediyor ve atlama komutunu yok sayıyordu. ALU sonucunda belirlenen bilgi sinyallerinden zero yani 0 bilgisine bakılıyor. Bu da decode adımında yapılan bir şey.



Şekil 2.9. İşlemci Durum Makinesi - ATLA_ESITSE_SIFIR

S_DECODE_3 eğer sıfır bilgi sinyali 1 ise brach işleme konulacak demektir. yani atlama yapılacak demektir. önce zero 1 durumuna bakalım;

S_BEQ_4, gördüğünüz gibi bu kısım bir önceki atla komutunun state diyagramı ile birebir aynı. Operand bellekten okunuyor ve program sayacı register ına yükleniyor. Sonra başlangıç state ine dönülüyor hiçbir fark yok.

Eğer Z bayrağı 0 ise branch işleme konmaz demiştik. Yani program sayacı normal seyrinde ilerler ve değeri basitçe +1 artar.

2.2.2. Kontrol Ünitesi - VHDL Tasarımı

Entity yapısını oluşturalım; control-unit. Portları tanımlayacağız. Portlara "cpu.vhd" diyagramından bakabiliriz. Kontrol ünitesine giren inputlar çok belli. IR dan ve CCR registerlarından giren bilgiler var. Clock ve Reset var. Görüldüğü üzere sadece 4 adet bilgi giriliyor kontrol ünitesine. Ki onları da veri yolundan kopyalayabiliriz. Zaten output olarak yazmışız. O zaman IR, CCR, clock ve reset i yazalım.

Outputlara geçelim ; Pek çok output umuz var. Diyagrama baktığımız zaman kontrol ünitesinden çıkıp dışarıya giden tüm sinyaller output.

- IR-Load,
- MAR-Load,

- PC-Load,
- PC-Inc,
- A-Load,
- B-Load,
- ALU-Sel,
- CCR-Load,
- BUS2-Sel,
- BUS1-Sel,
- Write

Bunları veriyolunda yazmıştık zaten, yine oradan kopyalayacağız. Veri yolu kodunda ki "port" kısmından inputlardan IR-Load dan itibaren (dahil) "from-memory" e kadar (dahil değil) seçiyor olmamız lazım. Bu portları Output a çevireceğiz tabii ki (in -> out). Ve bir de kontrol ünitesinden çıkan "write" outputunu yazmamız lazım, bunu da "write-enable" olarak ifade etmiştik öncesinde. Ve Entity i bu şekilde tamamladık.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

entity control_unit is
   port(
        clk        : in std_logic;
        rst        : in std_logic;
        CCR_Result : in std_logic_vector(3 downto 0);
        IR      : in std_logic_vector(7 downto 0);
        -- Outputlar:
        IR_Load    : out std_logic; -- Komut register'i yukle kontrol
        MAR_Load   : out std_logic;
        PC_Load  : out std_logic;
        PC_Inc     : out std_logic;
        A_Load     : out std_logic;
        B_Load     : out std_logic;
        ALU_Sel  : out std_logic_vector(2 downto 0);
        CCR_Load   : out std_logic;
        BUS1_Sel : out std_logic_vector(1 downto 0);
```

```
        BUS2_Sel : out std_logic_vector(1 downto 0);
        write_en : out std_logic

   );
end control_unit;
...
```

Architecture a geçelim. Burada state lerimizin type ını yazacağız. "state-type". Şimdi burada kullanacağımız tüm state leri tek tek yazmamız lazım.

S_FETCH_0 , S_FETCH_1 , S_FETCH_2, S_DECODE_3,

Önce Fetch adımının state lerini yazıyorum. 3 adet vardı. Ve daha sonra S_Decode adımı geliyordu. Buna da S_DECODE_3 diyelim. Bunlar tüm komutlar için ortaktı.

Şimdi hangilerini implemente edeceksek onların state lerini yazalım. Mesela YÜKLE_A_SBT komutunu implemente edecektik. Bu şekilde sırası ile tüm komutları implemente ediyorum.

```
...
rchitecture arch of control_unit is

type state_type is (
                S_FETCH_0, S_FETCH_1, S_FETCH_2, S_DECODE_3,
                S_LDA_IMM_4, S_LDA_IMM_5, S_LDA_IMM_6,
                S_LDA_DIR_4, S_LDA_DIR_5, S_LDA_DIR_6, S_LDA_DIR_7, S_LDA_DIR_8,
                S_LDB_IMM_4, S_LDB_IMM_5, S_LDB_IMM_6,
                S_LDB_DIR_4, S_LDB_DIR_5, S_LDB_DIR_6, S_LDB_DIR_7, S_LDB_DIR_8,
                S_STA_DIR_4, S_STA_DIR_5, S_STA_DIR_6, S_STA_DIR_7,
                S_ADD_AB_4,
                S_BRA_4, S_BRA_5, S_BRA_6,
                S_BEQ_4, S_BEQ_5, S_BEQ_6, S_BEQ_7
             );
...
```

Typelarımız bu kadar Şimdi artık state sinyalimizi tanımlayalım. İki tane state imiz olacak demiştik ; Current-State ve Next-State. Bunların type ı tabii ki state-type. "program belleği" nde yazmış olduğumuz constantlar, tek tek bu OPCODE ları yazmamak için bütün komutlarımızı tanıtmıştık. Onu yine alıp buraya tanıtmam lazım.

```
...
signal current_state, next_state : state_type;

-- Tum komutlar:

-- Kaydet/Yukle komutlari
constant YUKLE_A_SBT :std_logic_vector(7 downto 0) := x"86";
constant YUKLE_A   :std_logic_vector(7 downto 0) := x"87";
constant YUKLE_B_SBT :std_logic_vector(7 downto 0) := x"88";
constant YUKLE_B   :std_logic_vector(7 downto 0) := x"89";
constant KAYDET_A  :std_logic_vector(7 downto 0) := x"96";
constant KAYDET_B  :std_logic_vector(7 downto 0) := x"97";
-- ALU Komutlari
constant TOPLA_AB  :std_logic_vector(7 downto 0) := x"42";
constant CIKAR_AB  :std_logic_vector(7 downto 0) := x"43";
constant AND_AB       :std_logic_vector(7 downto 0) := x"44";
constant OR_AB     :std_logic_vector(7 downto 0) := x"45";
constant ARTTIR_A  :std_logic_vector(7 downto 0) := x"46";
constant ARTTIR_B  :std_logic_vector(7 downto 0) := x"47";
constant DUSUR_A   :std_logic_vector(7 downto 0) := x"48";
constant DUSUR_B   :std_logic_vector(7 downto 0) := x"49";
-- Atlama komutlari (Kosullu/Kosulsuz)
constant ATLA           :std_logic_vector(7 downto 0) := x"20";
constant ATLA_NEGATIFSE    :std_logic_vector(7 downto 0) := x"21";
constant ATLA_POZITIFSE    :std_logic_vector(7 downto 0) := x"22";
constant ATLA_ESITSE_SIFIR  :std_logic_vector(7 downto 0) := x"23";
constant ATLA_DEGILSE_SIFIR   :std_logic_vector(7 downto 0) := x"24";
constant ATLA_OVERFLOW_VARSA :std_logic_vector(7 downto 0) := x"25";
constant ATLA_OVERFLOW_YOKSA :std_logic_vector(7 downto 0) := x"26";
constant ATLA_ELDE_VARSA   :std_logic_vector(7 downto 0) := x"27";
constant ATLA_ELDE_YOKSA   :std_logic_vector(7 downto 0) := x"28";
...
```

"Begin" diyip mimari tanımımıza başlıyorum. Öncelikle şimdiki durum makinesini yazalım. Current-state ile. Bu şimdiki durumu güncelleyen process bloğu idi. Ve tabii ki de clock lu idi. (ve resetli) Eğer reset = 1 ise o anki durum S_FETCH_0

Bu processte sadece Current-State i güncelliyoruz, başka hiçbir data ile ilgili işimiz yok. Current state i bilinen bir başlangıç durumuna alalım bu da S_FETCH_0 state yani FETCH adımına, yeni komut okumaya başlamadığında geri dönsün. Böylelikle şimdiki durum devresini saate bağlı düzenli bir şekilde yazmış olduk.

```
...
begin

-- Current State Logic
```

```
   process (clk, rst)
   begin
      if(rst = '1') then
         current_state <= S_FETCH_0;
      elsif(rising_edge(clk)) then
         current_state <= next_state;
      end if;
   end process;
...
```

Şimdi Next-State bloğuna geçiyorum. Bu process bloğunda saat yoktu, bu process bloğu INPUT lara ve Current-State e bağlıydı. O yüzden sensivity list içinde "current state" olacak, aynı zamanda DECODE işleminde komutlar kullanılacağı için IR değeri burada kullanılacak. Ve birde atlama komutları için CCR değerleri gerekiyor. Bu bilgiler sayesinde Next-State güncellenecek.

Sonlu durum makinelerinde anlattığımız şekilde olası tüm komut durumlarını burada işliyorum ;

```
...
-- Next State Logic
   process(current_state, IR, CCR_Result)
   begin
      case current_state is
         when S_FETCH_0 =>
            next_state <= S_FETCH_1;
         when S_FETCH_1 =>
            next_state <= S_FETCH_2;
         when S_FETCH_2 =>
            next_state <= S_DECODE_3;
         when S_DECODE_3 =>
            if(IR = YUKLE_A_SBT) then
               next_state <= S_LDA_IMM_4;
            elsif(IR = YUKLE_A) then
               next_state <= S_LDA_DIR_4;
            elsif(IR = YUKLE_B_SBT) then
               next_state <= S_LDB_IMM_4;
            elsif(IR = YUKLE_B) then
               next_state <= S_LDB_DIR_4;
            elsif(IR = KAYDET_A) then
               next_state <= S_STA_DIR_4;
            elsif(IR = TOPLA_AB) then
               next_state <= S_ADD_AB_4;
            elsif(IR = ATLA) then
               next_state <= S_BRA_4;
            elsif(IR = ATLA_ESITSE_SIFIR) then
```

```
            if(CCR_Result(2) = '1') then --NZVC, Zero bilgisi 2. bitte
               next_state <= S_BEQ_4;
            else  -- Z = '0'
               next_state <= S_BEQ_7;
            end if;
         else
            next_state <= S_FETCH_0;
         end if;
-------------------------------------------------------------
      when S_LDA_IMM_4 =>
         next_state <= S_LDA_IMM_5;
      when S_LDA_IMM_5 =>
         next_state <= S_LDA_IMM_6;
      when S_LDA_IMM_6 =>
         next_state <= S_FETCH_0;
-------------------------------------------------------------

      when S_LDA_DIR_4 =>
         next_state <= S_LDA_DIR_5;
      when S_LDA_DIR_5 =>
         next_state <= S_LDA_DIR_6;
      when S_LDA_DIR_6 =>
         next_state <= S_LDA_DIR_7;
      when S_LDA_DIR_7 =>
         next_state <= S_LDA_DIR_8;
      when S_LDA_DIR_8 =>
         next_state <= S_FETCH_0;

-------------------------------------------------------------
      when S_LDB_IMM_4 =>
         next_state <= S_LDB_IMM_5;
      when S_LDB_IMM_5 =>
         next_state <= S_LDB_IMM_6;
      when S_LDB_IMM_6 =>
         next_state <= S_FETCH_0;
-------------------------------------------------------------

      when S_LDB_DIR_4 =>
         next_state <= S_LDB_DIR_5;
      when S_LDB_DIR_5 =>
         next_state <= S_LDB_DIR_6;
      when S_LDB_DIR_6 =>
         next_state <= S_LDB_DIR_7;
      when S_LDB_DIR_7 =>
         next_state <= S_LDB_DIR_8;
      when S_LDB_DIR_8 =>
         next_state <= S_FETCH_0;
-------------------------------------------------------------

      when S_STA_DIR_4 =>
```

```
            next_state <= S_STA_DIR_5;
         when S_STA_DIR_5 =>
            next_state <= S_STA_DIR_6;
         when S_STA_DIR_6 =>
            next_state <= S_STA_DIR_7;
         when S_STA_DIR_7 =>
            next_state <= S_FETCH_0;
-------------------------------------------------------------

         when S_ADD_AB_4 =>
            next_state <= S_FETCH_0;
-------------------------------------------------------------

         when S_BRA_4 =>
            next_state <= S_BRA_5;
         when S_BRA_5 =>
            next_state <= S_BRA_6;
         when S_BRA_6 =>
            next_state <= S_FETCH_0;
-------------------------------------------------------------

         when S_BEQ_4 =>
            next_state <= S_BEQ_5;
         when S_BEQ_5 =>
            next_state <= S_BEQ_6;
         when S_BEQ_6 =>
            next_state <= S_FETCH_0;
         when S_BEQ_7 => -- Z = '0' durumu, komut bypass
            next_state <= S_FETCH_0;
-------------------------------------------------------------

         when others =>
            next_state <= S_FETCH_0;

      end case;
   end process;
...
```

Next State process bloğunu da kapattık, current-state ve next-state bitti, şimdi sırada komutların gerçekten işlemlerin yapılacağı blok var. Bu blok tamamen current-state e bağlı yani şu anki durum ne ise ona göre ilgili RAM bloğuna geçiş yapılacak. Inputlar ile ilgili herhangi bir işimiz yok.

Bu blok OUTPUT Logic bloğu. Burada şöyle bir mantık var. Case current-state diyerek bütün state lerimizi yine yazıcaz aynı şekilde. State lerimizi komple "current-state" bloğundan çekelim.

```
...
-- Output Logic--

   process(current_state)
   begin
      IR_Load <= '0';
      MAR_Load <= '0';
      PC_Load <= '0';
      PC_Inc <= '0';
      A_Load <= '0';
       B_Load <= '0';
      ALU_Sel <= (others => '0');
      CCR_Load <= '0';
      BUS1_Sel <= (others => '0');
      BUS2_Sel <= (others => '0');
      write_en <= '0';

      case current_state is
         when S_FETCH_0 =>
            BUS1_Sel <= "00"; -- PC
            BUS2_Sel <= "01"; -- BUS1
            MAR_Load <= '1'; -- BUS2 daki program sayaci degeri MAR'a alindi
         when S_FETCH_1 =>
            PC_Inc <= '1';
         when S_FETCH_2 =>
            BUS2_Sel <= "10";  -- from memory
            IR_Load <= '1';
         when S_DECODE_3 =>
            -- next state zaten guncellendi, ve ilgili dallanma saglandi
-----------------------------------------------------------------
         when S_LDA_IMM_4 =>
            BUS1_Sel <= "00"; -- PC
            BUS2_Sel <= "01"; -- BUS1
            MAR_Load <= '1'; -- BUS2 daki program sayaci degeri MAR'a alindi
         when S_LDA_IMM_5 =>
            PC_Inc <= '1';
         when S_LDA_IMM_6 =>
            BUS2_Sel <= "10";  -- from memory
            A_Load <= '1';
-----------------------------------------------------------------

         when S_LDA_DIR_4 =>
            BUS1_Sel <= "00"; -- PC
            BUS2_Sel <= "01"; -- BUS1
            MAR_Load <= '1'; -- BUS2 daki program sayaci degeri MAR'a alindi
         when S_LDA_DIR_5 =>
            PC_Inc <= '1';
         when S_LDA_DIR_6 =>
            BUS2_Sel <= "10";  -- from memory
            MAR_Load <= '1'; -- BUS2 daki program sayaci degeri MAR'a alindi
```

```
        when S_LDA_DIR_7 =>
           -- BOS : Bellekten okuma yapilmasi bekleniyor.
        when S_LDA_DIR_8 =>
           BUS2_Sel <= "10";  -- from memory
           A_Load <= '1';
-------------------------------------------------------------
        when S_LDB_IMM_4 =>
           BUS1_Sel <= "00"; -- PC
           BUS2_Sel <= "01"; -- BUS1
           MAR_Load <= '1'; -- BUS2 daki program sayaci degeri MAR'a alindi
        when S_LDB_IMM_5 =>
           PC_Inc <= '1';
        when S_LDB_IMM_6 =>
           BUS2_Sel <= "10";  -- from memory
           B_Load <= '1';
-------------------------------------------------------------

        when S_LDB_DIR_4 =>
           BUS1_Sel <= "00"; -- PC
           BUS2_Sel <= "01"; -- BUS1
           MAR_Load <= '1'; -- BUS2 daki program sayaci degeri MAR'a alindi
        when S_LDB_DIR_5 =>
           PC_Inc <= '1';
        when S_LDB_DIR_6 =>
           BUS2_Sel <= "10";  -- from memory
           MAR_Load <= '1'; -- BUS2 daki program sayaci degeri MAR'a alindi
        when S_LDB_DIR_7 =>
           -- BOS : Bellekten okuma yapilmasi bekleniyor.
        when S_LDB_DIR_8 =>
           BUS2_Sel <= "10";  -- from memory
           B_Load <= '1';
-------------------------------------------------------------

        when S_STA_DIR_4 =>
           BUS1_Sel <= "00"; -- PC
           BUS2_Sel <= "01"; -- BUS1
           MAR_Load <= '1'; -- BUS2 daki program sayaci degeri MAR'a alindi
        when S_STA_DIR_5 =>
           PC_Inc <= '1';
        when S_STA_DIR_6 =>
           BUS2_Sel <= "10";  -- from memory
           MAR_Load <= '1'; -- BUS2 daki program sayaci degeri MAR'a alindi
        when S_STA_DIR_7 =>
           BUS1_Sel <= "01"; -- A_reg
           write_en <= '1';
-------------------------------------------------------------

        when S_ADD_AB_4 =>
           BUS1_Sel <= "01"; -- A_reg
           BUS2_Sel <= "00"; -- ALU result
```

```
            ALU_Sel <= "000"; -- Toplama kodu ALU'daki
            A_Load <= '1';
            CCR_Load <= '1';
-------------------------------------------------------------

         when S_BRA_4 =>
            BUS1_Sel <= "00"; -- PC
            BUS2_Sel <= "01"; -- BUS1
            MAR_Load <= '1'; -- BUS2 daki program sayaci degeri MAR'a alindi
         when S_BRA_5 =>
            -- BOS
         when S_BRA_6 =>
            BUS2_Sel <= "10";  -- from memory
            PC_Load <= '1';       -- Program sayaci registerina BUS2 verisini al
-------------------------------------------------------------

         when S_BEQ_4 =>
            BUS1_Sel <= "00"; -- PC
            BUS2_Sel <= "01"; -- BUS1
            MAR_Load <= '1'; -- BUS2 daki program sayaci degeri MAR'a alindi
         when S_BEQ_5 =>

         when S_BEQ_6 =>
            BUS2_Sel <= "10";  -- from memory
            PC_Load <= '1';       -- Program sayaci registerina BUS2 verisini al
         when S_BEQ_7 => -- Z = '0' durumu, komut bypass
            PC_Inc <= '1';
-------------------------------------------------------------

         when others =>
            IR_Load <= '0';
            MAR_Load <= '0';
            PC_Load <= '0';
            A_Load <= '0';
             B_Load <= '0';
            ALU_Sel <= (others => '0');
            CCR_Load <= '0';
            BUS1_Sel <= (others => '0');
            BUS2_Sel <= (others => '0');
            write_en <= '0';

      end case;
   end process;


end architecture;
...
```

Böylece kontrol ünitesi tasarımını da bitmiş oldu. Şimdi artık TOP-MODÜL

tasarımlarını kuracağım. CPU da veri yolunu ve kontrol ünitesini birleştireceğim. Daha sonra "computer.vhd" de tüm bu alt blokları birleştirerek CPU TOP-LEVEL tasarımımızı oluşturacağım.

2.3. Merkezi İşlem Birimi Bloğu - CPU

Kontrol ünitesi ve veriyolunu tamamladık. Sadece alt blokları kullanarak TOP Modülü yapmak kaldı. Şimdi CPU Top Modülünü inşaa edeceğiz. Yapacağımız tek şey aslında, komponent tanımı ve ilgili port-map yapıları olacak.

CPU entity sini oluşturalım. Clock, reset ve "from-memory" sinyallerini input olarak görüyoruz. Çıkışlarında da "to-memory", write ve adres olarak görünüyor.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

entity CPU is
   port(
        clk   : in std_logic;
        rst   : in std_logic;
        from_memory : in std_logic_vector(7 downto 0);
        -- Outputs:
        to_memory  : out std_logic_vector(7 downto 0);
        write_en : out std_logic;
        address    : out std_logic_vector(7 downto 0)
   );
end CPU;
...
```

Architecture a geçiyorum. Komponentleri tanımlamamız lazım. "Control-Unit" ve "Data-Path" olarak iki tane komponent imiz var. Sonrasında bağlantıyı kurmak için ihtiyacımız olan bağlantı sinyallerini tanımlayalım. Sonuçta bunları portmap ile birbirine bağlayacağız. Bunları bağlamak için porttan porta bağlama yapmıyoruz. Ara sinyalleri kullanıyoruz.

```
...
architecture arch of CPU is

-- Control Unit:
component control_unit is
```

```
   port(
        clk        : in std_logic;
        rst        : in std_logic;
        CCR_Result : in std_logic_vector(3 downto 0);
        IR         : in std_logic_vector(7 downto 0);
        -- Outputlar:
        IR_Load    : out std_logic; -- Komut register'i yukle kontrol
        MAR_Load   : out std_logic;
        PC_Load    : out std_logic;
        PC_Inc     : out std_logic;
        A_Load     : out std_logic;
        B_Load     : out std_logic;
        ALU_Sel    : out std_logic_vector(2 downto 0);
        CCR_Load   : out std_logic;
        BUS1_Sel   : out std_logic_vector(1 downto 0);
        BUS2_Sel   : out std_logic_vector(1 downto 0);
        write_en   : out std_logic

   );
end component;

...
-- Data Path:
component data_path is
   port(
        clk        : in std_logic;
        rst        : in std_logic;
        IR_Load    : in std_logic; -- Komut register'i yukle kontrol
        MAR_Load   : in std_logic;
        PC_Load    : in std_logic;
        PC_Inc     : in std_logic;
        A_Load     : in std_logic;
        B_Load     : in std_logic;
        ALU_Sel    : in std_logic_vector(2 downto 0);
        CCR_Load   : in std_logic;
        BUS1_Sel   : in std_logic_vector(1 downto 0);
        BUS2_Sel   : in std_logic_vector(1 downto 0);
        from_memory : in std_logic_vector(7 downto 0);
        -- Outputs:
        IR         : out std_logic_vector(7 downto 0);
        address    : out std_logic_vector(7 downto 0); -- bellege giden adres bilgisi
        CCR_Result : out std_logic_vector(3 downto 0); -- NZVC
        to_memory  : out std_logic_vector(7 downto 0) -- bellege giden veri

   );
end component;
...
```

Bu bağlantıyı sağladık. Şimdi sırada Data-Path var. Burada IR karşısına yine "IR,"

yazacağız.

```
-- Baglanti Sinyalleri

signal IR_Load  : std_logic;  -- Komut register'i yukle kontrol
signal IR       : std_logic_vector(7 downto 0);
signal MAR_Load  : std_logic;
signal PC_Load     : std_logic;
signal PC_Inc   : std_logic;
signal A_Load   : std_logic;
signal B_Load   : std_logic;
signal ALU_Sel     : std_logic_vector(2 downto 0);
signal CCR_Load  : std_logic;
signal CCR_Result : std_logic_vector(3 downto 0);
signal BUS1_Sel    : std_logic_vector(1 downto 0);
signal BUS2_Sel    : std_logic_vector(1 downto 0);

begin

-- Control Unit:
control_unit_module: control_unit port map
                        (
                           clk          =>    clk,
                           rst           =>  rst,
                           CCR_Result    =>  CCR_Result,
                           IR          =>  IR,
                           -- Outputlar
                           IR_Load       => IR_Load,
                           MAR_Load      => MAR_Load,
                           PC_Load     => PC_Load,
                           PC_Inc        => PC_Inc,
                           A_Load        => A_Load,
                           B_Load        => B_Load,
                           ALU_Sel     => ALU_Sel,
                           CCR_Load      => CCR_Load,
                           BUS1_Sel    => BUS1_Sel,
                           BUS2_Sel    => BUS2_Sel,
                           write_en    => write_en
                        );


-- Data Path:
data_path_module: data_path port map
                     (
                        clk          => clk,
                        rst           => rst,
                        IR_Load       => IR_Load,
                        MAR_Load      => MAR_Load,
                        PC_Load     => PC_Load,
                        PC_Inc        => PC_Inc,
                        A_Load        => A_Load,
```

```
            B_Load          => B_Load,
            ALU_Sel     => ALU_Sel,
            CCR_Load        => CCR_Load,
            BUS1_Sel    => BUS1_Sel,
            BUS2_Sel    => BUS2_Sel,
            from_memory     => from_memory,
            -- Outputs:     => -- Outputs:
            IR          => IR,
            address         => address,
            CCR_Result      => CCR_Result,
            to_memory       => to_memory
        );


end architecture;
...
```

"to memory" ve "from-memory" CPU ya ait input-output portlar, o yüzden bunlar burada bir yere bağlı değil. Doğrudan giriş ve çıkış portlarına bağlı.

Entity deki CPU input-output portlarını ilgili veriyolu veya kontrol ünitesi portlarına bağladıktan sonra gerisi tamamen iç bağlantılar.

Kontrol ünitesinin "IR" portu, Data Path in çıkışına bağlı. Bu şekilde bağlantıyı sağladık.

Tüm CPU.vhd kodu ;

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

entity CPU is
   port(
         clk   : in std_logic;
         rst   : in std_logic;
         from_memory : in std_logic_vector(7 downto 0);
         -- Outputs:
         to_memory  : out std_logic_vector(7 downto 0);
         write_en : out std_logic;
         address    : out std_logic_vector(7 downto 0)
   );
end CPU;

architecture arch of CPU is
```

```
-- Control Unit:
component control_unit is
   port(
        clk        : in std_logic;
        rst        : in std_logic;
        CCR_Result : in std_logic_vector(3 downto 0);
        IR      : in std_logic_vector(7 downto 0);
        -- Outputlar:
        IR_Load    : out std_logic; -- Komut register'i yukle kontrol
        MAR_Load   : out std_logic;
        PC_Load  : out std_logic;
        PC_Inc     : out std_logic;
        A_Load     : out std_logic;
        B_Load     : out std_logic;
        ALU_Sel  : out std_logic_vector(2 downto 0);
        CCR_Load   : out std_logic;
        BUS1_Sel : out std_logic_vector(1 downto 0);
        BUS2_Sel : out std_logic_vector(1 downto 0);
        write_en : out std_logic

   );
end component;

-- Data Path:
component data_path is
   port(
        clk        : in std_logic;
        rst        : in std_logic;
        IR_Load    : in std_logic; -- Komut register'i yukle kontrol
        MAR_Load   : in std_logic;
        PC_Load  : in std_logic;
        PC_Inc     : in std_logic;
        A_Load     : in std_logic;
        B_Load     : in std_logic;
        ALU_Sel  : in std_logic_vector(2 downto 0);
        CCR_Load   : in std_logic;
        BUS1_Sel : in std_logic_vector(1 downto 0);
        BUS2_Sel : in std_logic_vector(1 downto 0);
        from_memory : in std_logic_vector(7 downto 0);
        -- Outputs:
        IR      : out std_logic_vector(7 downto 0);
        address    : out std_logic_vector(7 downto 0); -- bellege giden adres bilgisi
        CCR_Result : out std_logic_vector(3 downto 0); -- NZVC
        to_memory  : out std_logic_vector(7 downto 0) -- bellege giden veri

   );
end component;

-- Baglanti Sinyalleri
```

```
signal IR_Load   : std_logic;  -- Komut register'i yukle kontrol
signal IR        : std_logic_vector(7 downto 0);
signal MAR_Load  : std_logic;
signal PC_Load     : std_logic;
signal PC_Inc    : std_logic;
signal A_Load    : std_logic;
signal B_Load    : std_logic;
signal ALU_Sel     : std_logic_vector(2 downto 0);
signal CCR_Load  : std_logic;
signal CCR_Result : std_logic_vector(3 downto 0);
signal BUS1_Sel    : std_logic_vector(1 downto 0);
signal BUS2_Sel    : std_logic_vector(1 downto 0);

begin

-- Control Unit:
control_unit_module: control_unit port map
                        (
                           clk           =>    clk,
                           rst            =>  rst,
                           CCR_Result     =>  CCR_Result,
                           IR          =>  IR,
                           -- Outputlar
                           IR_Load        => IR_Load,
                           MAR_Load       => MAR_Load,
                           PC_Load     => PC_Load,
                           PC_Inc         => PC_Inc,
                           A_Load         => A_Load,
                           B_Load         => B_Load,
                           ALU_Sel     => ALU_Sel,
                           CCR_Load       => CCR_Load,
                           BUS1_Sel    => BUS1_Sel,
                           BUS2_Sel    => BUS2_Sel,
                           write_en    => write_en
                        );


-- Data Path:
data_path_module: data_path port map
                     (
                        clk           => clk,
                        rst            => rst,
                        IR_Load        => IR_Load,
                        MAR_Load       => MAR_Load,
                        PC_Load     => PC_Load,
                        PC_Inc         => PC_Inc,
                        A_Load         => A_Load,
                        B_Load         => B_Load,
                        ALU_Sel     => ALU_Sel,
                        CCR_Load       => CCR_Load,
```

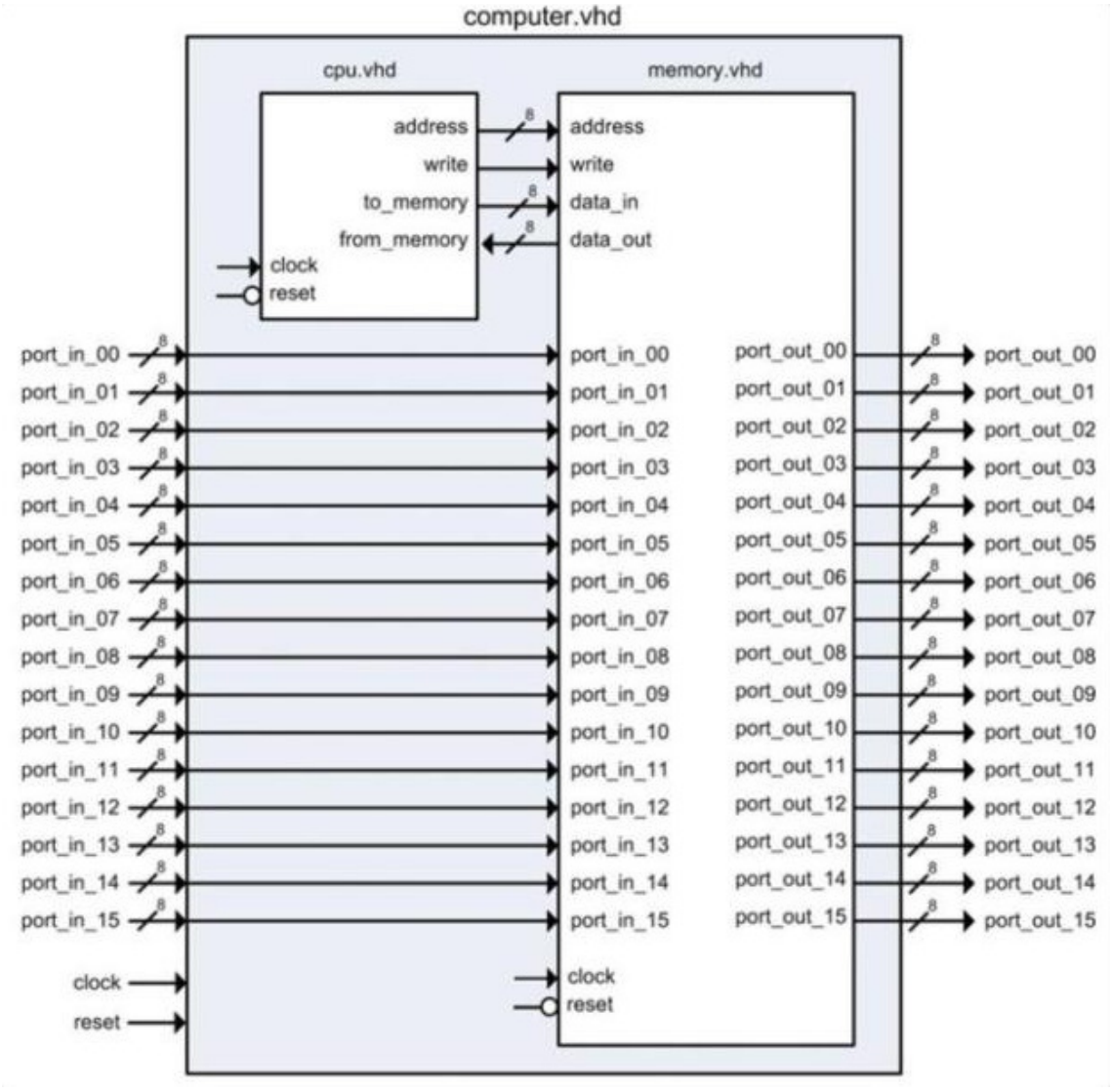
```
        BUS1_Sel    => BUS1_Sel,
        BUS2_Sel    => BUS2_Sel,
        from_memory    => from_memory,
        -- Outputs:    => -- Outputs:
        IR          => IR,
        address        => address,
        CCR_Result     => CCR_Result,
        to_memory      => to_memory
    );

end architecture;
```

Artık “computer.vhd” tanımına geçebiliriz.

3. BÖLÜM

İşlemci ÜST-SEVİYE BLOĞU

3.0.1. İşlemci ÜST-SEVİYE Bloğu



Şekil 3.1. Computer.vhd ÜST-SEVİYE Tasarım Diyagramı

"computer.vhd" top modülüne ulaştık, "memory" ile başlamıştık. 3 Farklı bellek türü kodladık, daha sonra CPU içerisinde hem veriyolu hem de kontrol ünitesini ayrıntılı bir şekilde işlevlerini açıklayıp tasarladık, kodladık. Şimdi geriye yaptığımız bütün yapıyı içerecek olan computer yapısını kodlamak kalıyor. Yani bu bizim tasarımımızın TOP MODÜL ü olmuş oluyor. Bakarsak "computer.vhd" nin portlarında 16 tane 8-bit lik INPUT portu, 16 tane 8-bit lik output portu, clock ve reset var. İçeride CPU ve Memory komponentleri var, ki bunların ikisini de TOP MODÜL olarak belirttim. İçeride de kullanacağımız bağlantı sinyalleri ikisi arasında görünmektedir ; address, write, "to-memory", "from-memory", bu iki bloğu birbirine bağlayan sinyaller.

Şimdi bunları tanımlayarak "computer.vhd" top modülünü tasarımına başlıyorum. Portları 16 adet 8-bit Input portu ve 16 tane 8-bit lik output portları ve clock ve reset olacak.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

entity computer is
   port(
         clk        : in std_logic;
         rst        : in std_logic;
         port_in_00 : in std_logic_vector(7 downto 0);
         port_in_01 : in std_logic_vector(7 downto 0);
         port_in_02 : in std_logic_vector(7 downto 0);
         port_in_03 : in std_logic_vector(7 downto 0);
         port_in_04 : in std_logic_vector(7 downto 0);
         port_in_05 : in std_logic_vector(7 downto 0);
         port_in_06 : in std_logic_vector(7 downto 0);
         port_in_07 : in std_logic_vector(7 downto 0);
         port_in_08 : in std_logic_vector(7 downto 0);
         port_in_09 : in std_logic_vector(7 downto 0);
         port_in_10 : in std_logic_vector(7 downto 0);
         port_in_11 : in std_logic_vector(7 downto 0);
         port_in_12 : in std_logic_vector(7 downto 0);
         port_in_13 : in std_logic_vector(7 downto 0);
         port_in_14 : in std_logic_vector(7 downto 0);
         port_in_15 : in std_logic_vector(7 downto 0);
         -- Output:
         port_out_00 : out std_logic_vector(7 downto 0);
         port_out_01 : out std_logic_vector(7 downto 0);
         port_out_02 : out std_logic_vector(7 downto 0);
```

```
        port_out_03 : out std_logic_vector(7 downto 0);
        port_out_04 : out std_logic_vector(7 downto 0);
        port_out_05 : out std_logic_vector(7 downto 0);
        port_out_06 : out std_logic_vector(7 downto 0);
        port_out_07 : out std_logic_vector(7 downto 0);
        port_out_08 : out std_logic_vector(7 downto 0);
        port_out_09 : out std_logic_vector(7 downto 0);
        port_out_10 : out std_logic_vector(7 downto 0);
        port_out_11 : out std_logic_vector(7 downto 0);
        port_out_12 : out std_logic_vector(7 downto 0);
        port_out_13 : out std_logic_vector(7 downto 0);
        port_out_14 : out std_logic_vector(7 downto 0);
        port_out_15 : out std_logic_vector(7 downto 0)

   );
end entity;
...
```

Portlar bu kadar, architecture a geçelim ve komponentlerimizi tanımlayalım.

CPU komponentlerimizin biri, diğeri de Memory. CPU dan başlayalım.

```
...
architecture arch of computer is

-- CPU:
component CPU is
   port(
        clk        : in std_logic;
        rst        : in std_logic;
        from_memory : in std_logic_vector(7 downto 0);
        -- Outputs:
        to_memory  : out std_logic_vector(7 downto 0);
        write_en : out std_logic;
        address    : out std_logic_vector(7 downto 0)
   );
end component;
...
```

memory e geçelim.

```
...
-- memory:
component memory is
   port(
        clk        : in std_logic;
        rst        : in std_logic;
        address    : in std_logic_vector(7 downto 0);
```

```
        data_in    : in std_logic_vector(7 downto 0);
        write_en   : in std_logic; -- CPU tarafindan gonderilen kontrol sinyali / yaz
        port_in_00 : in std_logic_vector(7 downto 0);
        port_in_01 : in std_logic_vector(7 downto 0);
        port_in_02 : in std_logic_vector(7 downto 0);
        port_in_03 : in std_logic_vector(7 downto 0);
        port_in_04 : in std_logic_vector(7 downto 0);
        port_in_05 : in std_logic_vector(7 downto 0);
        port_in_06 : in std_logic_vector(7 downto 0);
        port_in_07 : in std_logic_vector(7 downto 0);
        port_in_08 : in std_logic_vector(7 downto 0);
        port_in_09 : in std_logic_vector(7 downto 0);
        port_in_10 : in std_logic_vector(7 downto 0);
        port_in_11 : in std_logic_vector(7 downto 0);
        port_in_12 : in std_logic_vector(7 downto 0);
        port_in_13 : in std_logic_vector(7 downto 0);
        port_in_14 : in std_logic_vector(7 downto 0);
        port_in_15 : in std_logic_vector(7 downto 0);
        -- Output:
        data_out : out std_logic_vector(7 downto 0);
        --
        port_out_00 : out std_logic_vector(7 downto 0);
        port_out_01 : out std_logic_vector(7 downto 0);
        port_out_02 : out std_logic_vector(7 downto 0);
        port_out_03 : out std_logic_vector(7 downto 0);
        port_out_04 : out std_logic_vector(7 downto 0);
        port_out_05 : out std_logic_vector(7 downto 0);
        port_out_06 : out std_logic_vector(7 downto 0);
        port_out_07 : out std_logic_vector(7 downto 0);
        port_out_08 : out std_logic_vector(7 downto 0);
        port_out_09 : out std_logic_vector(7 downto 0);
        port_out_10 : out std_logic_vector(7 downto 0);
        port_out_11 : out std_logic_vector(7 downto 0);
        port_out_12 : out std_logic_vector(7 downto 0);
        port_out_13 : out std_logic_vector(7 downto 0);
        port_out_14 : out std_logic_vector(7 downto 0);
        port_out_15 : out std_logic_vector(7 downto 0)
    );
end component;
...
```

Ve içeride kullanacağımız sinyaller vardı, onlarda “cpu.vhd” ve “memory.vhd” arasında ki sinyallerdi.

```
...
-- Computer top module sinyalleri:
signal address : std_logic_vector(7 downto 0);
signal data_in : std_logic_vector(7 downto 0);
signal data_out : std_logic_vector(7 downto 0);
```

```
signal write_en : std_logic;
...
```

portmaplere geçebiliriz. CPU ve memory bloklarında aynı sinyallerin isimleri birbirinden farklı (örn. "from-memory" cpu da data-out, o yüzden dikkat etmek lazım.

```
...
begin

-- CPU port map
cpu_module: CPU port map
          (
             clk          => clk,
             rst           =>  rst,
             from_memory    =>   data_out,
             -- Outputs:    =>
             to_memory      =>   data_in,
             write_en    =>   write_en,
             address        =>   address
          );
...
```

Memory e geçiyoruz. Memory port map;

```
...
-- Memory port map:
memory_module: memory port map
        (
           clk          => clk   ,
           rst           =>  rst,
           address       =>  address   ,
           data_in       =>  data_in   ,
           write_en      =>  write_en  ,
           port_in_00    =>  port_in_00,
           port_in_01    =>  port_in_01,
           port_in_02    =>  port_in_02,
           port_in_03    =>  port_in_03,
           port_in_04    =>  port_in_04,
           port_in_05    =>  port_in_05,
           port_in_06    =>  port_in_06,
           port_in_07    =>  port_in_07,
           port_in_08    =>  port_in_08,
           port_in_09    =>  port_in_09,
           port_in_10    =>  port_in_10,
           port_in_11    =>  port_in_11,
           port_in_12    =>  port_in_12,
```

```
            port_in_13    =>  port_in_13,
            port_in_14    =>  port_in_14,
            port_in_15    =>  port_in_15,
            -- Output:
            data_out    =>  data_out,
            --
            port_out_00   =>  port_out_00 ,
            port_out_01   =>  port_out_01 ,
            port_out_02   =>  port_out_02 ,
            port_out_03   =>  port_out_03 ,
            port_out_04   =>  port_out_04 ,
            port_out_05   =>  port_out_05 ,
            port_out_06   =>  port_out_06 ,
            port_out_07   =>  port_out_07 ,
            port_out_08   =>  port_out_08 ,
            port_out_09   =>  port_out_09 ,
            port_out_10   =>  port_out_10 ,
            port_out_11   =>  port_out_11 ,
            port_out_12   =>  port_out_12 ,
            port_out_13   =>  port_out_13 ,
            port_out_14   =>  port_out_14 ,
            port_out_15   =>  port_out_15
        );


end architecture;
...
```

Computer Top modülünü de tamamlamış olduk. Artık sistemimiz bitti, tasarımı sonlardırdık. Şimdi yapmamız geren şey proje oluşturup test, doğrulama ve sentez adımlarına geçmek.

Tüm Computer.vhd kodu;

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

entity computer is
   port(
        clk        : in std_logic;
        rst        : in std_logic;
        port_in_00 : in std_logic_vector(7 downto 0);
        port_in_01 : in std_logic_vector(7 downto 0);
        port_in_02 : in std_logic_vector(7 downto 0);
        port_in_03 : in std_logic_vector(7 downto 0);
        port_in_04 : in std_logic_vector(7 downto 0);
```

```
        port_in_05 : in std_logic_vector(7 downto 0);
        port_in_06 : in std_logic_vector(7 downto 0);
        port_in_07 : in std_logic_vector(7 downto 0);
        port_in_08 : in std_logic_vector(7 downto 0);
        port_in_09 : in std_logic_vector(7 downto 0);
        port_in_10 : in std_logic_vector(7 downto 0);
        port_in_11 : in std_logic_vector(7 downto 0);
        port_in_12 : in std_logic_vector(7 downto 0);
        port_in_13 : in std_logic_vector(7 downto 0);
        port_in_14 : in std_logic_vector(7 downto 0);
        port_in_15 : in std_logic_vector(7 downto 0);
        -- Output:
        port_out_00 : out std_logic_vector(7 downto 0);
        port_out_01 : out std_logic_vector(7 downto 0);
        port_out_02 : out std_logic_vector(7 downto 0);
        port_out_03 : out std_logic_vector(7 downto 0);
        port_out_04 : out std_logic_vector(7 downto 0);
        port_out_05 : out std_logic_vector(7 downto 0);
        port_out_06 : out std_logic_vector(7 downto 0);
        port_out_07 : out std_logic_vector(7 downto 0);
        port_out_08 : out std_logic_vector(7 downto 0);
        port_out_09 : out std_logic_vector(7 downto 0);
        port_out_10 : out std_logic_vector(7 downto 0);
        port_out_11 : out std_logic_vector(7 downto 0);
        port_out_12 : out std_logic_vector(7 downto 0);
        port_out_13 : out std_logic_vector(7 downto 0);
        port_out_14 : out std_logic_vector(7 downto 0);
        port_out_15 : out std_logic_vector(7 downto 0)

    );
end entity;

architecture arch of computer is

-- CPU:
component CPU is
    port(
        clk        : in std_logic;
        rst        : in std_logic;
        from_memory : in std_logic_vector(7 downto 0);
        -- Outputs:
        to_memory  : out std_logic_vector(7 downto 0);
        write_en : out std_logic;
        address    : out std_logic_vector(7 downto 0)
    );
end component;

-- memory:
component memory is
    port(
```

```
        clk        : in std_logic;
        rst        : in std_logic;
        address    : in std_logic_vector(7 downto 0);
        data_in    : in std_logic_vector(7 downto 0);
        write_en   : in std_logic; -- CPU tarafindan gonderilen kontrol sinyali / yaz
        port_in_00 : in std_logic_vector(7 downto 0);
        port_in_01 : in std_logic_vector(7 downto 0);
        port_in_02 : in std_logic_vector(7 downto 0);
        port_in_03 : in std_logic_vector(7 downto 0);
        port_in_04 : in std_logic_vector(7 downto 0);
        port_in_05 : in std_logic_vector(7 downto 0);
        port_in_06 : in std_logic_vector(7 downto 0);
        port_in_07 : in std_logic_vector(7 downto 0);
        port_in_08 : in std_logic_vector(7 downto 0);
        port_in_09 : in std_logic_vector(7 downto 0);
        port_in_10 : in std_logic_vector(7 downto 0);
        port_in_11 : in std_logic_vector(7 downto 0);
        port_in_12 : in std_logic_vector(7 downto 0);
        port_in_13 : in std_logic_vector(7 downto 0);
        port_in_14 : in std_logic_vector(7 downto 0);
        port_in_15 : in std_logic_vector(7 downto 0);
        -- Output:
        data_out : out std_logic_vector(7 downto 0);
        --
        port_out_00 : out std_logic_vector(7 downto 0);
        port_out_01 : out std_logic_vector(7 downto 0);
        port_out_02 : out std_logic_vector(7 downto 0);
        port_out_03 : out std_logic_vector(7 downto 0);
        port_out_04 : out std_logic_vector(7 downto 0);
        port_out_05 : out std_logic_vector(7 downto 0);
        port_out_06 : out std_logic_vector(7 downto 0);
        port_out_07 : out std_logic_vector(7 downto 0);
        port_out_08 : out std_logic_vector(7 downto 0);
        port_out_09 : out std_logic_vector(7 downto 0);
        port_out_10 : out std_logic_vector(7 downto 0);
        port_out_11 : out std_logic_vector(7 downto 0);
        port_out_12 : out std_logic_vector(7 downto 0);
        port_out_13 : out std_logic_vector(7 downto 0);
        port_out_14 : out std_logic_vector(7 downto 0);
        port_out_15 : out std_logic_vector(7 downto 0)
    );
end component;

-- Computer top module sinyalleri:
signal address : std_logic_vector(7 downto 0);
signal data_in : std_logic_vector(7 downto 0);
signal data_out : std_logic_vector(7 downto 0);
signal write_en : std_logic;

begin
```

```
-- CPU port map
cpu_module: CPU port map
          (
             clk           => clk,
             rst            =>  rst,
             from_memory    =>   data_out,
             -- Outputs:    =>
             to_memory      =>   data_in,
             write_en    =>   write_en,
             address        =>   address
          );

-- Memory port map:
memory_module: memory port map
        (
          clk           => clk   ,
          rst            =>  rst,
          address        =>  address   ,
          data_in        =>  data_in   ,
          write_en       =>  write_en ,
          port_in_00     =>  port_in_00,
          port_in_01     =>  port_in_01,
          port_in_02     =>  port_in_02,
          port_in_03     =>  port_in_03,
          port_in_04     =>  port_in_04,
          port_in_05     =>  port_in_05,
          port_in_06     =>  port_in_06,
          port_in_07     =>  port_in_07,
          port_in_08     =>  port_in_08,
          port_in_09     =>  port_in_09,
          port_in_10     =>  port_in_10,
          port_in_11     =>  port_in_11,
          port_in_12     =>  port_in_12,
          port_in_13     =>  port_in_13,
          port_in_14     =>  port_in_14,
          port_in_15     =>  port_in_15,
          -- Output:
          data_out    =>  data_out,
          --
          port_out_00    =>  port_out_00 ,
          port_out_01    =>  port_out_01 ,
          port_out_02    =>  port_out_02 ,
          port_out_03    =>  port_out_03 ,
          port_out_04    =>  port_out_04 ,
          port_out_05    =>  port_out_05 ,
          port_out_06    =>  port_out_06 ,
          port_out_07    =>  port_out_07 ,
          port_out_08    =>  port_out_08 ,
          port_out_09    =>  port_out_09 ,
```

```
            port_out_10    =>  port_out_10 ,
            port_out_11    =>  port_out_11 ,
            port_out_12    =>  port_out_12 ,
            port_out_13    =>  port_out_13 ,
            port_out_14    =>  port_out_14 ,
            port_out_15    =>  port_out_15
        );


end architecture;
```

4. BÖLÜM

TEST VE DOĞRULAMA

4.0.1. Vivado Programı Üzerinde Test ve Doğrulama

Computer.vhd ile üst-seviye tasarımıda tamamlayarak 8-bit işlemci tasarımını bitirdik. Şimdi Vivado programı üzerinde bu tasarımın test ve doğrulama işlemlerini yapacağım. İşlemcimizin BIOS mantığında çalışan bir yapı olduğundan bahsetmiştim. Bu sebeple ROM yapısında giriş ve çıkış portları üzerinden değer alan, bu değerleri kaydedicilere yazan ve işlem yapan örnek bir program yazmam lazım.

Bu programı "program-memory.vhd" kodumda ki ROM-TYPE kısmına yazıyorum.

```
...
type rom_type is array (0 to 127) of std_logic_vector(7 downto 0);
   constant ROM : rom_type := (
                                0  => YUKLE_A,
                           1  => x"F0",  -- input port-00
                           2  => YUKLE_B,
                           3  => x"F1",  -- input port-01
                           4   => TOPLA_AB,
                           5   => ATLA_ESITSE_SIFIR,
                           6   => x"0B",
                           7   => KAYDET_A,
                           8   => X"80",
                           9  => ATLA,
                           10 => x"20",
                           11 => YUKLE_A,
                           12 => x"F2",  -- input port-02
                           13  => ATLA,
                           14  => x"04",
                           others  => x"00"
                       );
...
```

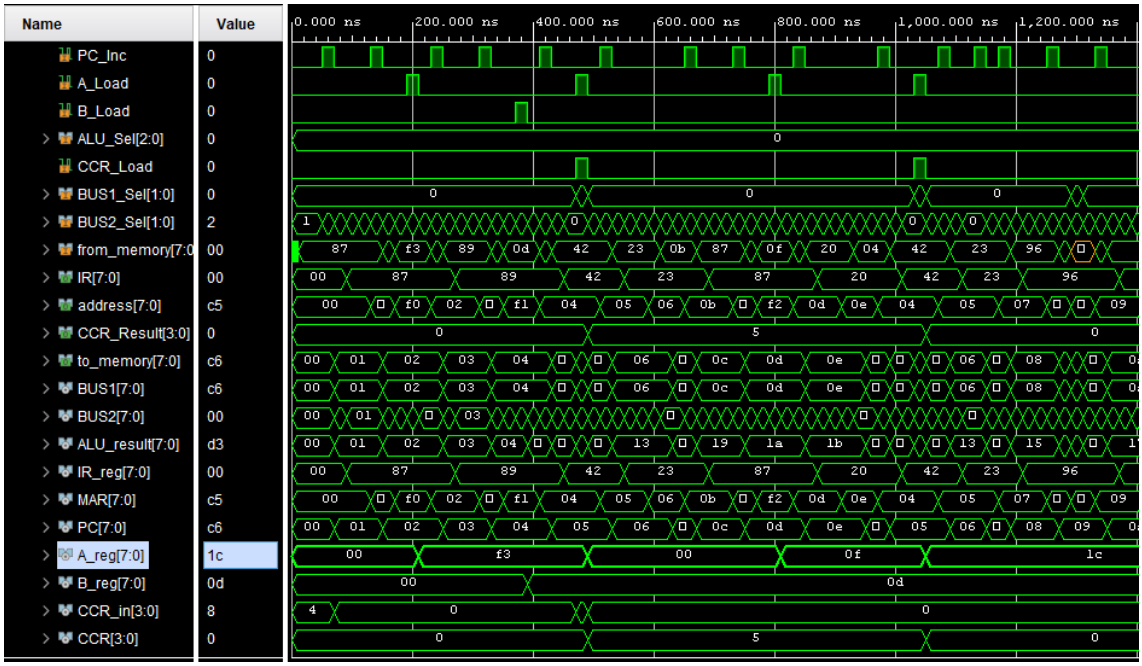
Sonrasında işlemcimizi sanal bir ortamda test edeceğim için A ve B INPUT portlarına bir test-bench yazıyorum;

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity tb_computer is
--  Port ( );
end tb_computer;

architecture Behavioral of tb_computer is
component computer is
   port(
        clk        : in std_logic;
        rst        : in std_logic;
        port_in_00 : in std_logic_vector(7 downto 0);
        port_in_01 : in std_logic_vector(7 downto 0);
        port_in_02 : in std_logic_vector(7 downto 0);
        port_in_03 : in std_logic_vector(7 downto 0);
        port_in_04 : in std_logic_vector(7 downto 0);
        port_in_05 : in std_logic_vector(7 downto 0);
        port_in_06 : in std_logic_vector(7 downto 0);
        port_in_07 : in std_logic_vector(7 downto 0);
        port_in_08 : in std_logic_vector(7 downto 0);
        port_in_09 : in std_logic_vector(7 downto 0);
        port_in_10 : in std_logic_vector(7 downto 0);
        port_in_11 : in std_logic_vector(7 downto 0);
        port_in_12 : in std_logic_vector(7 downto 0);
        port_in_13 : in std_logic_vector(7 downto 0);
        port_in_14 : in std_logic_vector(7 downto 0);
        port_in_15 : in std_logic_vector(7 downto 0);
        -- Output:
        port_out_00 : out std_logic_vector(7 downto 0);
        port_out_01 : out std_logic_vector(7 downto 0);
        port_out_02 : out std_logic_vector(7 downto 0);
        port_out_03 : out std_logic_vector(7 downto 0);
        port_out_04 : out std_logic_vector(7 downto 0);
        port_out_05 : out std_logic_vector(7 downto 0);
        port_out_06 : out std_logic_vector(7 downto 0);
        port_out_07 : out std_logic_vector(7 downto 0);
        port_out_08 : out std_logic_vector(7 downto 0);
        port_out_09 : out std_logic_vector(7 downto 0);
        port_out_10 : out std_logic_vector(7 downto 0);
        port_out_11 : out std_logic_vector(7 downto 0);
        port_out_12 : out std_logic_vector(7 downto 0);
        port_out_13 : out std_logic_vector(7 downto 0);
        port_out_14 : out std_logic_vector(7 downto 0);
        port_out_15 : out std_logic_vector(7 downto 0)

   );
```

```
end component;

--
constant clock_period : time := 20 ns;
--
signal clk      : std_logic;
signal rst      : std_logic;
signal port_in_00 : std_logic_vector(7 downto 0);
signal port_in_01 : std_logic_vector(7 downto 0);
signal port_in_02 : std_logic_vector(7 downto 0);
signal port_in_03 : std_logic_vector(7 downto 0);
signal port_in_04 : std_logic_vector(7 downto 0);
signal port_in_05 : std_logic_vector(7 downto 0);
signal port_in_06 : std_logic_vector(7 downto 0);
signal port_in_07 : std_logic_vector(7 downto 0);
signal port_in_08 : std_logic_vector(7 downto 0);
signal port_in_09 : std_logic_vector(7 downto 0);
signal port_in_10 : std_logic_vector(7 downto 0);
signal port_in_11 : std_logic_vector(7 downto 0);
signal port_in_12 : std_logic_vector(7 downto 0);
signal port_in_13 : std_logic_vector(7 downto 0);
signal port_in_14 : std_logic_vector(7 downto 0);
signal port_in_15 : std_logic_vector(7 downto 0);

signal port_out_00 : std_logic_vector(7 downto 0);
signal port_out_01 : std_logic_vector(7 downto 0);
signal port_out_02 : std_logic_vector(7 downto 0);
signal port_out_03 : std_logic_vector(7 downto 0);
signal port_out_04 : std_logic_vector(7 downto 0);
signal port_out_05 : std_logic_vector(7 downto 0);
signal port_out_06 : std_logic_vector(7 downto 0);
signal port_out_07 : std_logic_vector(7 downto 0);
signal port_out_08 : std_logic_vector(7 downto 0);
signal port_out_09 : std_logic_vector(7 downto 0);
signal port_out_10 : std_logic_vector(7 downto 0);
signal port_out_11 : std_logic_vector(7 downto 0);
signal port_out_12 : std_logic_vector(7 downto 0);
signal port_out_13 : std_logic_vector(7 downto 0);
signal port_out_14 : std_logic_vector(7 downto 0);
signal port_out_15 : std_logic_vector(7 downto 0);

begin

clock_process: process
               begin
                    clk <= '0';
                    wait for clock_period/2;
                    clk <= '1';
                    wait for clock_period/2;
               end process;
```

```
uut: computer port map
(
    clk       => clk       ,
    rst       => rst       ,
    port_in_00 => port_in_00 ,
    port_in_01 => port_in_01 ,
    port_in_02 => port_in_02 ,
    port_in_03 => port_in_03 ,
    port_in_04 => port_in_04 ,
    port_in_05 => port_in_05 ,
    port_in_06 => port_in_06 ,
    port_in_07 => port_in_07 ,
    port_in_08 => port_in_08 ,
    port_in_09 => port_in_09 ,
    port_in_10 => port_in_10 ,
    port_in_11 => port_in_11 ,
    port_in_12 => port_in_12 ,
    port_in_13 => port_in_13 ,
    port_in_14 => port_in_14 ,
    port_in_15 => port_in_15 ,

    port_out_00 => port_out_00 ,
    port_out_01 => port_out_01 ,
    port_out_02 => port_out_02 ,
    port_out_03 => port_out_03 ,
    port_out_04 => port_out_04 ,
    port_out_05 => port_out_05 ,
    port_out_06  => port_out_06 ,
    port_out_07  => port_out_07 ,
    port_out_08  => port_out_08 ,
    port_out_09  => port_out_09 ,
    port_out_10  => port_out_10 ,
    port_out_11  => port_out_11 ,
    port_out_12  => port_out_12 ,
    port_out_13  => port_out_13 ,
    port_out_14  => port_out_14 ,
    port_out_15  => port_out_15
);

process
begin
rst <= '1';
wait for 40ns;
rst <= '0';
wait for clock_period*2;

-- PROGRAM-3 ICIN ASAGIDAKI YORUMLARI KALDIRINIZ:
port_in_00 <= "11110011"; -- -13 demek
port_in_01 <= x"0D";
```

```
port_in_02 <= x"0F";

wait for clock_period*200;
wait;

end process;

end Behavioral;
```



Şekil 4.1. WaveForm Görüntüsü - DataPath

Şimdi Yazdığım programı Vivado üzerinde davranışsal olarak simüle edeceğim ve işlemcimizin WaveForm üzerinde verdiği tepkiye göre test ve doğrulama adımlarını izleyeceğim.

WaveForm a baktığımız zaman, A register ını Data-Path den bulalım.

- A register ına F0 da bulunan değeri yükle (Input 00 portunu A register ına yükleyecek)

F0 aslında bizim inputumuzu ifade ediyor. Nereden anladık? Memory yapımıza (Bellek yapısı) baktığımız zaman x"F0" ve x"FF" arası IO Input Portlarını adresliyordu. Yani bu adresteki bir değeri almak istediğim zaman Input lara ulaşıyorum demekti. Ve bunu da implemente etmiştik zaten.

- B register ına F1 de bulunan değeri yükle (Input 01 portunu B register ına

yükleyecek)

Aynı şekilde F1 adresi de Input Port 01 i ifade eder, bunu da B register ına yükleyeceğiz.

• A ve B register larında ki değerleri toplayıp sonucu A register ına yaz

Daha sonra A ve B register ını toplayacağız.

• Sonuç 0 a eşitse 11. Satıra atla

Eğer bu sonuç 0 ise x"0B" yani 11. Program adımına atlayacağız demektir (B=11).

• YUKLE_A – x"F2" – ATLA – x"04"

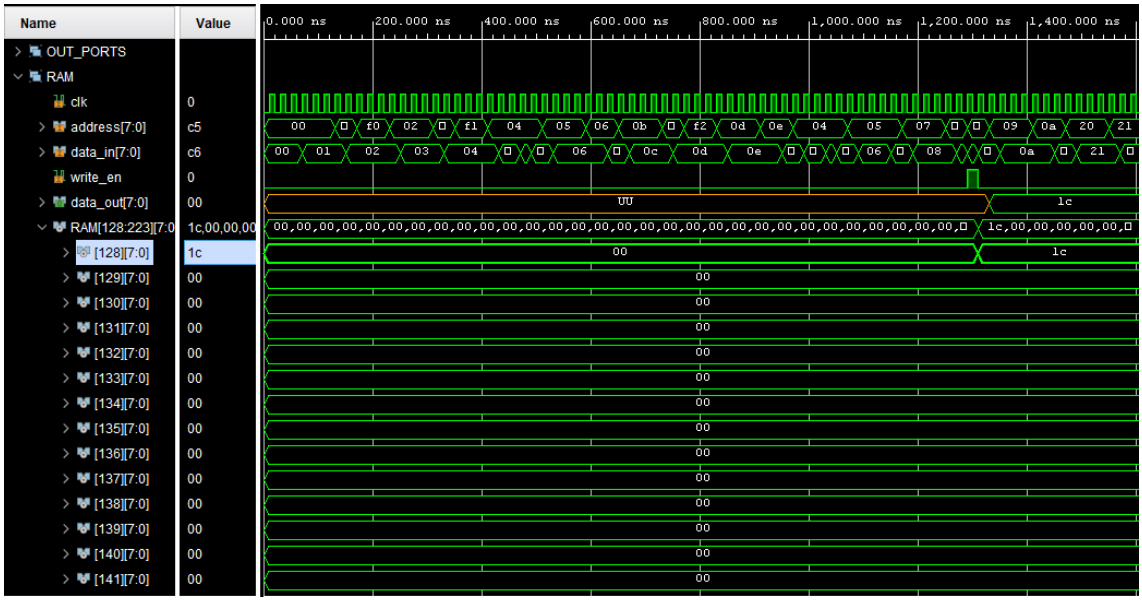
Bu sefer A register ına F2 yani Input port 02 değerini yükleyeceğiz ve daha sonra tekrar bir ATLA komutu ile 04 satırına döneceğiz. Yani A ya yeni bir Input portundan değer verip tekrar topluyoruz ve yine bir 0 kontrolü yapacağız. Yine 0 sa tekrar buraya atlar, değilse devam eder. Çünkü sıra sıra işleniyor Program Counter ın geldikten sonra ki değeri.

• Eğer 0 değilse atlamıyoruz ve Branch kod alınmıyor. KAYDET_A ya geliyor ve x"80" yani RAM ilk adresi olan 128. Adrese A register ındaki bulunan toplam sonucu değeri yazılıyor.

• Daha sonra ATLA ile 32.Komut belleği adresine atlıyoruz, bu aslında boş yani OTHERS a denk gelen "00" olan bir adres. Bunu neden yaptım? Programı kitlemek için yani artık 0 oku, hiçbir şey yapma, bitir programı der gibi implement ettiğimiz bütün komutlar burada hem Input portlarını kullanabiliyor muyuz? Data çekiyor muyuz bunu göreceğiz. Tabii bunun için sisteme Input vermemiz lazım.

Ben de örnek olması açısından Inputları tb_computer.vhd yani test-bench koduna yazdım.

A ya yüklenecek port-in-00 portuna -13 sayısını yerleştirdim, -13 tools complement ifadesidir. Scientific bir sayı ile çalıştığımız için zaten. B ye de 13 yerleştirdik, 0D. Yani bunların toplamları 0 edicek ve ilk branch i almış olucaz. İlk branch i aldığımız için A ya tekrar bir sayı yüklememiz gerekecek. Bu da port 02 idi. Buna



Şekil 4.2. WaveForm Görüntüsü - RAM

da 15 sayısını yükleyeceğiz ve böylelikle bütün komutun bize vermiş olduğu döngüyü sağlamış olacağız.

Şimdi waveform dan görüleceği üzere ; A ya -13 yüklemiştik zaten. B ye de 13 yüklemiştik ki bunların toplamları 0 etsin. Ve 2. A-Load da gördüğünüz gibi toplamları 0 etmiş. A register ında 0 ı elde ettikten sonra tekrar Branch e girip Input tan 2.Input portunda ki veriyi A ya yüklüyorduk. Yeni değerimiz 15, burada belli bir süre işlemlerini yaptıktan sonra porta erişmiş, gerekli state geçişleri yapılmış ve A register ına 15 yüklenmiş.

Ve şimdi tekrar toplamaya gideceğiz neden? A ya yeni Input u yükledik, ATLA diyip 04 e gelip tekrar topla-AB ye gelmemiz gerekiyor. TOPLA_AB ye geldik, A register ında bu sefer yeni değer olan 15 var, B register ında da hala 13 var ve toplamları 28 yapacağından bu da A register ına tekrar yazılmış. Yani satır 6 e kadar (ATLA_ESITSE_SIFIR dahil) kesinlikle doğru.

Geriye 7-8-9-10. Satırlar kaldı. A yı kaydedebilecek miyiz RAM e ve daha sonra kitleyebilecek miyiz programı 32. Komut pozisyonuna atlayarak. RAM e bakarsak 28 (x"1c") kaydedilmiş.

Write-en yani yazma emri geldikten sonra 0. Eleman olarak 28 i görürüz. RAM e doğru bir şekilde yazılmış.

Sistemi tekrar koşturduğumda sistemde ki bütün değerlerin korunduğunu yani sistemin kitlenmiş olduğunu görüyorum. Sadece boş bir komut okuyoruz yani 00 pozisyonunu. Örnek program konuştuğumuz her şeyi içeriyordu, ve bizim işlemcimiz bu işlemi de hatasız bir şekilde yaptı.

5. BÖLÜM

TARTIŞMA, SONUÇ ve ÖNERİLER

5.1. Tartışma, Sonuç ve Öneriler

CPU-module ve memory-module ü tam olarak dökümanda belirttiğimiz gibi görebiliyoruz. Yapının tamamı da tamamen computer TOP MODUL ünü oluşturuyor.

bunlar INPUT BUFFER larımız. OUTPUT BUFFER larımız da aynı şekildeler. (Şekil 5.2)

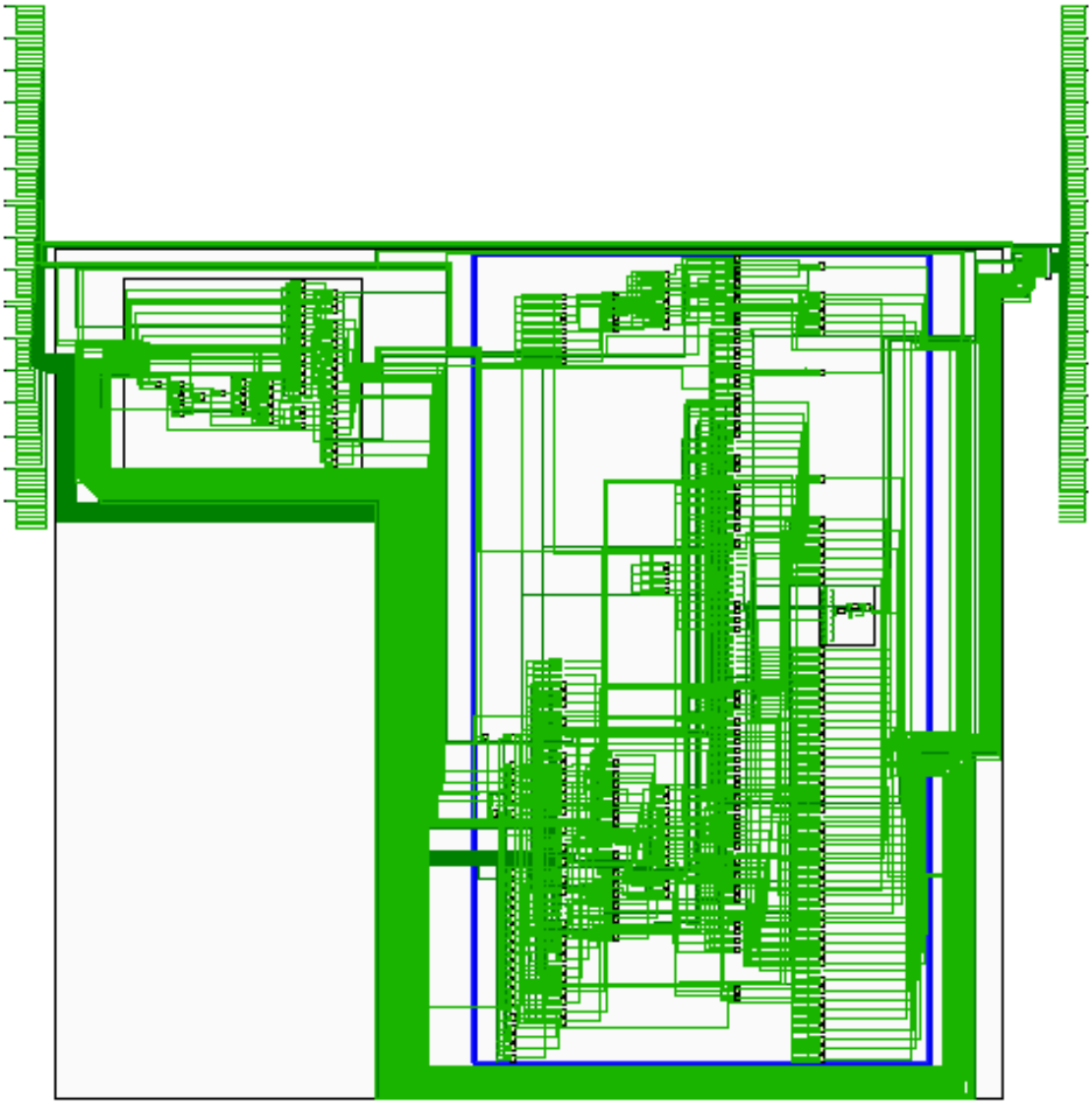
Alt bloklar CPU-module ve memory-module içinde, (Şekil 5.3 ve Şekil 5.4)

Statik zaman analizinde ise görmüş olduğunuz gibi Setup Time ımız 12,428 pozitif slack vermiş yani 50 MHz lik clock frekansını 20 ns lik periyodunu fazlası ile karşılayabiliyoruz.(Şekil 5.5)

İşlemciyi fiziksel olarak çıkaracak olursak genel yapısını Şekil 5.6. da görebiliriz

Bu çalışma sonucunda BIOS mantalitesinde ROM yapısı içerisine kazınan programı başarılı bir şekilde çalıştırabilen ve gerçek dünya koşulları altında statik zaman analizinden elde ettiğimiz sonuçlar doğrultusunda problemsiz istenileni yerine getirebilecek 8-bit bir işlemci tasarımı yapıldı.

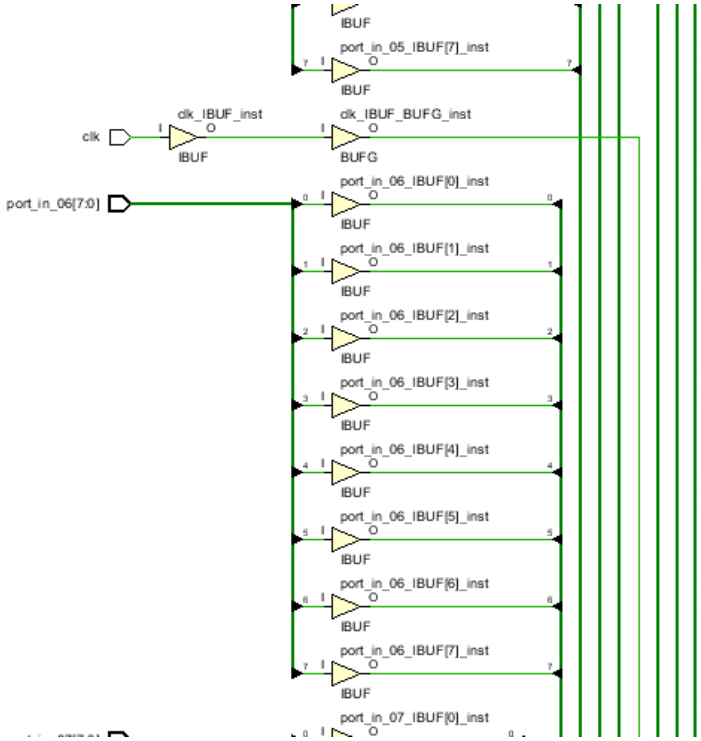
Söz konusu tasarım daha geniş çerçevede çok miktarda veriyi işlemek üzere tekrar dizayn edilerek işlemciye 16-bit veri işleme kapasitesi kazandırılabilir ve ROM yapısında yapılan değişiklikler ile dinamik olarak programlanabilir hale getirilerek günümüzde kullanılan PIC, Arduino ve hatta STM32, ESP32 gibi mikroişlemci



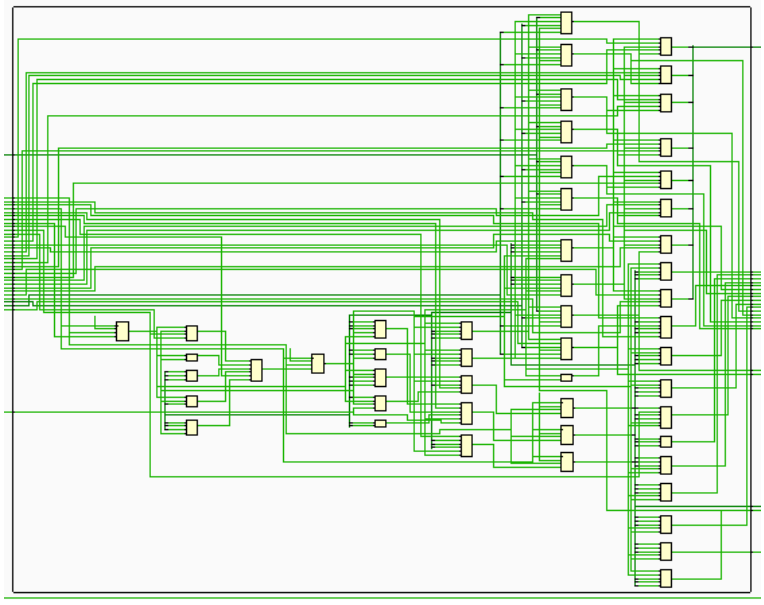
Şekil 5.1. İşlemcimizin Lojik Şeması

seviyelerine getirilebilir ve Türkiye endüstrisinde gömülü sistemlerin olduğu her yerde kullanılabilir.

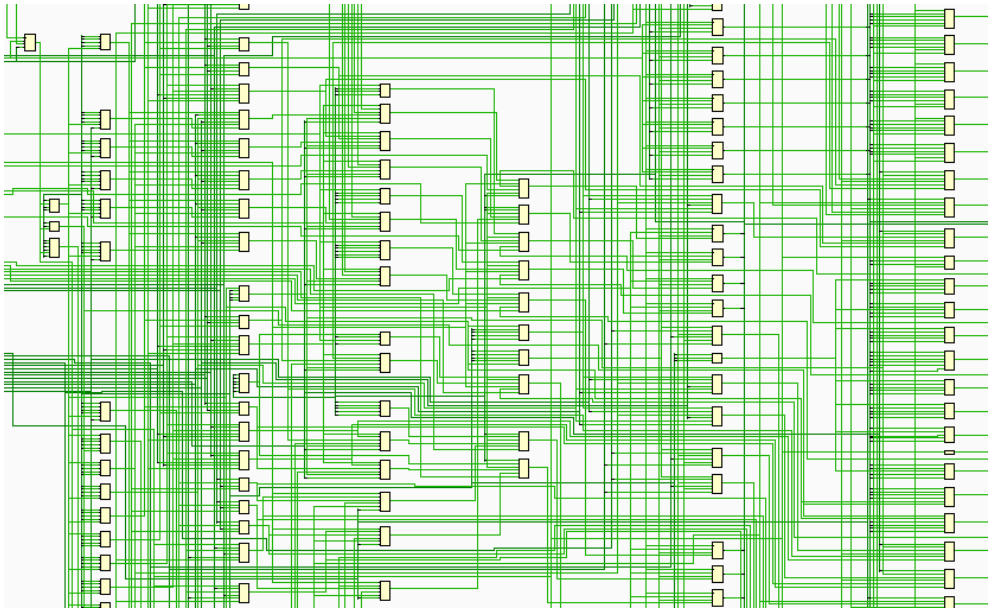
Bir VGA/HDMI arayüzü eklenerek veriler görselleştirilebilir ve ticari amaçlı olmasa da savunma sanayii ya da devletin belli kurumlarında sadece ülke içi amaçlara hizmet etmek üzere kullanılabilir. Bu durum bizim dışa bağımlılığımızı büyük oranda azaltabilir.



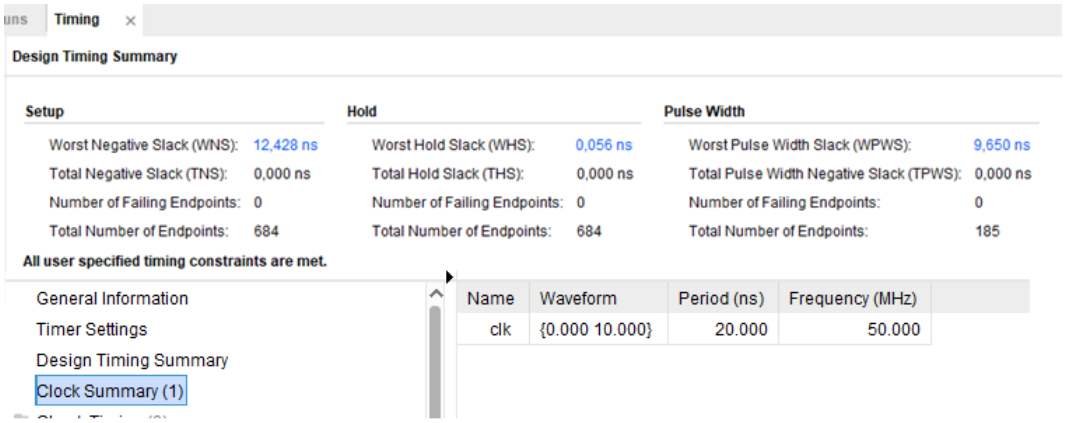
Şekil 5.2. INPUT BUFFERS



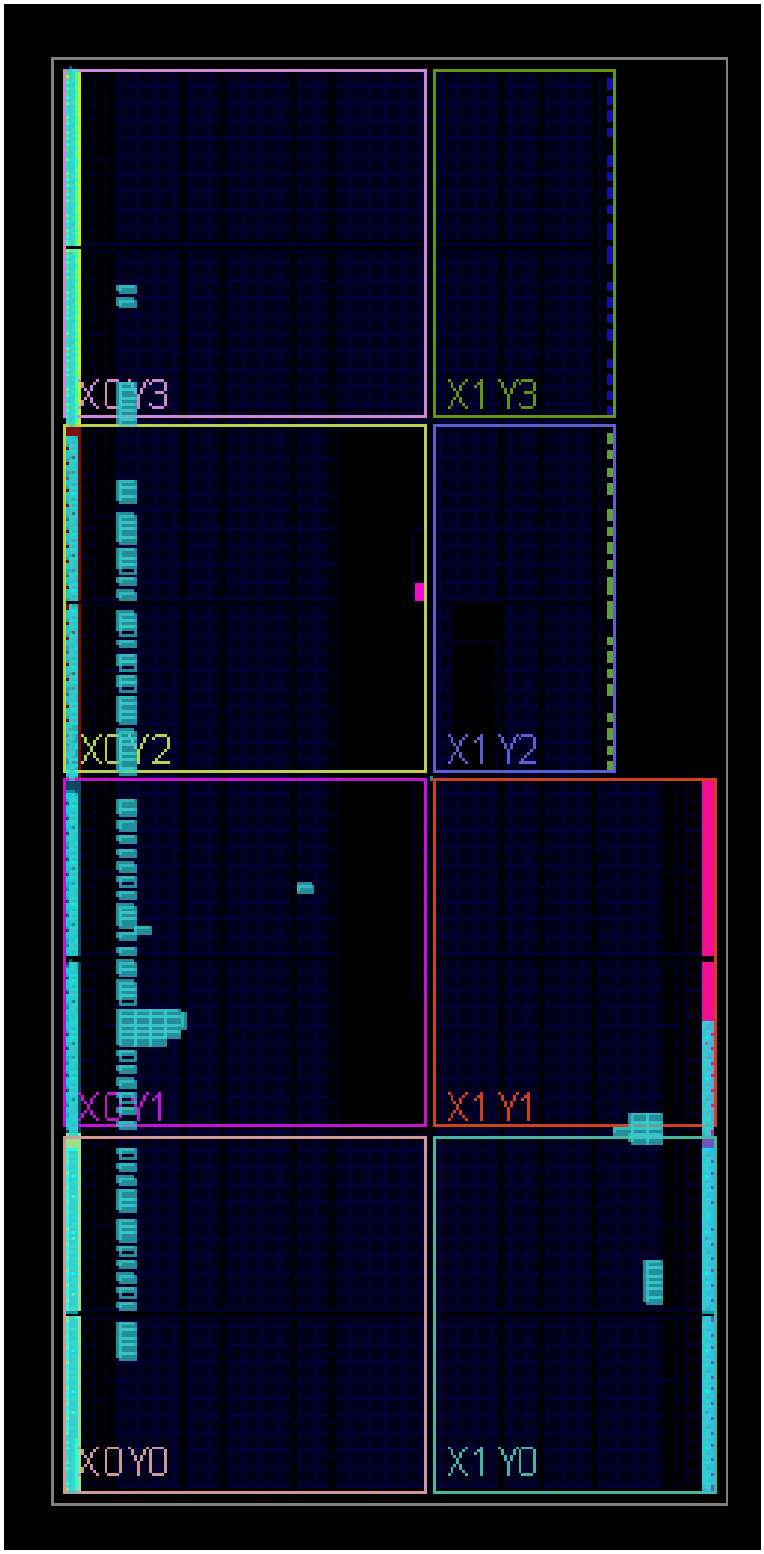
Şekil 5.3. CPU Devresi



Şekil 5.4. Memory Devresi



Şekil 5.5. Statik Zaman Analizi Sonunçları



Şekil 5.6. Donanım Şeması

KAYNAKLAR

ÖZGEÇMİŞ

KİŞİSEL BİLGİLER

Adı, Soyadı	:	Ömer Can VURAL
Uyruğu	:	Türkiye (T.C.)
Doğum Tarihi ve Yeri	:	XX.XX.1998 - XXXXX
Telefon	:	X XXX XXX XX XX
Belgegeçer	:	
E-posta	:	1030516774@erciyes.edu.tr
Adres	:	XXXXX XXXXX XXXXX Ankara TÜRKİYE

EĞİTİM

Derece	**Kurum**	**Mez.Yılı**
Lise	Ankara Sanayi Odası Mesleki ve Teknik Anadolu Lisesi, ANKARA	2016
Ortaokul	XXXXX	2012