CUKUROVA UNIVERSITY

INSTITUTE OF NATURAL AND APPLIED SCIENCES

MSc THESIS

RISC Processor Design on FPGA and BMS Application

Polatcan POLAT

Electrical and Electronics Engineering Department

January, 2025

CUKUROVA UNIVERSITY

INSTITUTE OF NATURAL AND APPLIED SCIENCES

MSc THESIS APPROVAL

RISC Processor Design on FPGA and BMS Application

Polatcan POLAT

Electrical and Electronics Engineering Department

This Master's Thesis was evaluated by the following Jury Members on 06/01/2025 and was approved by majority of votes.

Jury : Prof. Dr. Mustafa GÖK (Advisor) ……

: Assoc. Prof. Dr. Ahmet AYDIN ……

: Assoc. Prof. Dr. Zehan KESİLMİŞ ……

This Thesis was written in the Department of Electrical and Electronics Engineering, Institute of Natural and Applied Sciences.

Thesis Number:

Prof. Dr. Sadık DİNÇER
Director
Institute of Natural and Applied Sciences

CONTENTS

CUKUROVA UNIVERSITY
INSTITUTE OF NATURAL AND APPLIED SCIENCES

MSc THESIS

RISC Processor Design on FPGA and BMS Application

Polatcan POLAT

Advisor: Prof. Dr. Mustafa GÖK

Department of Electrical and Electronics Engineering

ABSTRACT

Fast calculations within Battery Management Systems (BMS) with low power use is important for the safety of humans and the environment. To accomplish this goal, BMS requires a fast processor which does the calculations. RV32I base integer instruction set, which is part of RISC-V instruction set architecture (ISA), is one of the candidates with its robust RISC architecture.

The issue of storing electric energy for future use became crucial with the increasing demand for portable electronic devices, electric vehicles and temporary power storage. Li-ion batteries are one of the most reliable choices in today's technology due to their energy density, efficiency, and cycle life. However, they should be monitored and managed for safety and reliability reasons.

BMS monitors and manages the batteries. There are many parameters to measure from Li-ion battery and to calculate from the measurement. For example, the voltage of the battery, charge-discharge current, temperature, state of charge (SOC), state of health (SOH), etc. Some of the calculations even include the time domain, and these parameters are so essential to be known they indicate the remaining time to charge fully or the remaining time to discharge fully, the charges left in the battery, the condition of the battery, and many more.

This study aims to design, implement and test the designed RISC processor. In this thesis, a 7-stage pipelined RV32I core has been described in Verilog HDL and implemented on a Basys-3 FPGA board with 100MHz clock frequency using the Vivado Design Suit. When the core is separated, it can reach 150 MHz frequency on the Basys-3 board and 270 MHz frequency on the Kintex-7 board. The core executes instructions as described in ISA. To communicate with other devices, such as a personal computer, a UART module is also described in Verilog HDL and PuTTy software is used as a serial communication terminal application for the personal computer. For analog data, LogiCORE XADC IP is used, and a compatible module is also described in Verilog HDL to read external voltage levels from the sensors. This study contributes a 7-staged pipelined RV32I core to literature, and it can be improved in the future to more generalized or more specific versions.

Keywords: Processor, RISC, RISC-V, RV32I, FPGA, BMS.

ÇUKUROVA ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

YÜKSEK LİSANS TEZİ

FPGA Üzerinde Gerçekleştirilen RISC işlemci ve BYS Uygulaması

Polatcan POLAT

Danışman: Prof. Dr. Mustafa GÖK

Elektrik Elektronik Mühendisliği Anabilim Dalı

ÖZ

Batarya yönetim sistemlerinde yüksek hızlarda ve düşük güç tüketerek yapılan işlemler hem insan hem de çevre sağlığı için önemlidir. Bu gereksinimi karşılamak için sistem hızlı bir işlemciye ihtiyaç duyar. RISC-V ISA'sı içerisinde yer alan RV32I komut seti hızlı ve etkin yapısı ile ön plana çıkmaktadır.

Özellikle taşınabilir elektronik aletler, elektrikli arabalar ve yenilenebilir enerji gibi alanların gelişmesiyle ortaya çıkan elektrik enerjisini depolama sorunu günümüzde önem kazanmıştır. Li-ion bataryalar enerji yoğunluğu, verimi ve kullanım süresi gibi pozitif özellikleri ile elektrik enerjisini depolamak için en çok tercih edilir seçeneklerden biri haline gelmiştir. Bu bataryaların güvenilir ve verimli olabilmesi için parametrelerinin izlenmesi, radikal durumlarda müdahale edilmesi gerekir.

Batarya yönetim sistemleri Li-ion bataryaları izler ve yönetir. Li-ion bataryalarda ölçülmesi ve hesaplanması gereken pek çok parametre vardır. Bunlardan bazıları voltaj, akım ve sıcaklık olup hesaplanması gereken diğer bazı parametreler ise şarj durumu, sağlık durumu gibidir. Bazı hesaplamalar zaman içerisindeki sensör verilerinin değişimleri neticesinde yapılması gerekir. Bu parametreler çok temel ve gereklidir. Bazı parametreler neticesinde bataryanın ne zaman boşalacağı, ne kadar şarjı kaldığı, gibi pek çok hesaplama yapılabilir.

Bu çalışma çalışan bir RISC işlemcinin tasarlanmasını, gerçeklenmesini ve gerçeklenen tasarımın test edilmesini hedefler. Bu tezde Verilog HDL ile RV32I komut setli 7 boru-hattı aşamalı çekirdek tasarlanmıştır ve Vivado Design Suit programı ile Basys-3 FPGA kartında 100 MHz saat çevriminde gerçeklenmiştir. Çekirdek ISA tarifine göre komutları çalıştırabilmektedir. Çekirdek tek başına Basys-3 üzerinde 150 MHz ve Kintex-7 üzerinde 273 MHz saat çevrim hızlarına çıkabilmektedir. Sistem Basys-3 üzerinde 130 MHz ve Kintex-7 üzerinde 212 MHz saat çevrim hızlarına çıkabilmektedir. Sistem içeriği UART, XADC, Bellek ve veri yolu conttrol mantığıdır. Bu çalışma literatüre 7 boru-hattı aşamalı bir RV32I çekirdeği katkısında bulunur ve bu çekirdek gelecekte genel işlev üzerine veya özgül işlev üzerine daha da geliştirilebilir.

Anahtar Kelimeler: İşlemci, RISC, RISC-V, RV32I, FPGA, BMS.

GENİŞLETİLMİŞ ÖZET

Teknolojinin gelişmesi ile elektronik aletlere ve elektrikli araçlara artan talepler yenilenebilir enerji kayakları üzerine yapılan yatırımlar, vb. beraberinde elektrik enerjisinin depolanabilme durumunun ve depolanmış enerjinin taşınma durumunu öne çıkarttı. Bu ihtiyacın karşılanmasında en çok tercih edilen ikincil pil kategorisine, yani tekrar şarj edilebilen piller arasına giren Li-ion bataryalar oldu.

Batarya, elektrik enerjisini uzunca bir süre depolayabilen depolama birimlerine verilen bir isimdir. Li-ion bataryaların tercih olarak öne çıkmasındaki temel faktörler birim ağırlık başına depolayabildiği enerjinin ve birim hacim başına depolayabildiği enerjinin yüksek olması, yüksek verimliliğe sahip olması ve kullanım ömrü ile raf ömrünün uzun olması olarak gösterilebilir.

Zaman içerisinde bir Li-ion batarya hücresi kendini öne çıkartan bu özellikleri kaybedecek olsa da istenmeyen kullanım koşulları Li-ion bataryaların kullanım sürelerini önemli ölçüde azaltabilir. Ekonomik kaybın da ötesinde ekstrem durumlarda olan hücreler yanma ve patlama tehlikesi arz edebilir. Cep telefonları, diz üstü bilgisayarlar, gibi pek çok günlük kullanımda olan modern hayatın vazgeçilmezi haline gelmiş elektronik aletlerin içerisinde kullanılan Li-ion bataryaların kullanım şartları, kullanılması için tasarlanan şartların dışına çıkartılması durumunda kullanım ömrünün kısalmasından başlayıp, yanma ve patlama gibi durumlara sebep olabilecek raddelere gelebilir. Li-ion bataryanın durumunun bu noktalara gelmesine engel olmak için, bataryanın durumu izlenmeli ve gerekiyorsa müdahale edilmelidir.

Bataryayı izleyip gerektiğinde müdahale eden sistemlere batarya yönetim sistemleri (BYS) denir. Bu sistemlerin pek çok türü ve işlevi olup, bu işlevleri yerine getirebilmesi için ilk olarak bataryadan alınması gereken sıcaklık, voltaj, akım gibi değerlerin bilinmesi gerekir. Bu veriler neticesinde bataryanın diğer parametreleri hesaplanabilir.

Basite indirgendiğinde, bir BYS'nin iki ana işlevi olduğu söylenebilir. Birinci işlevi bataryayı izlemek, ikinci işlevi ise bataryayı korumaktır. Kritik değerlerin bilinmesi, bataryanın çalıştırdığı cihazı kullanan kişi açısından büyük öneme sahiptir. Örneğin, bir telefon kullanıcısı telefonunun içerisindeki bataryada kalan şarj durumunu bilmek ister. Eğer BYS bataryanın parametrelerini düzgün okuyamaz veya düzgün hesaplayamazsa bataryanın şarjının ne durumda olduğunu bilemez. Her ne kadar bu durum acil bir duruma tekabül etmediği zaman zararsız olsa da elektrikli arabalarda yolda kalma gibi durumlara sebep olabilir. İkincisi ise bataryanın istenmeyen şartlarda çalışmaya zorlandığında koruma yapmak. Anlaşılacağı üzere izleme aşaması hem dışardan yapılması gereken müdahaleler için bataryanın durumunu kullanıcıya aktarır, hem de BYS içerisindeki elektronik devrenin davranışını düzenleyerek bataryanın ömrünün uzatılmasını sağlar. Dolayısıyla bataryanın tehlike arz eden bir duruma geçmemesini önleyerek hem insan sağlığına hem de çevre sağlığına zarar vermesinin önüne geçer. Ek olarak sistem hücreler arasındaki dengesizliği

aktif veya pasif olarak dengeleyebilir. Bu dengeleme bataryanın tek bir hücre yüzünden işlevsiz hale gelmesine mani olabilir.

Batarya koruma sisteminin bataryayı korumaya geçtiği durumlar; aşırı şarj, aşırı şarj boşaltımı, şarj esnasında aşırı akım, şarj boşaltımı sırasında aşrı akım, yüksek voltajda şarj, şarj boşaltımı sırasında düşük voltaj, yüksek sıcaklık, düşük sıcaklık gibi durumlardır. Batarya koruma sistemleri bu tezin kapsamına alınmamıştır.

Batarya yönetim sistemlerinde hesaplamalar genellikle sıcaklık, voltaj ve akım verileri üzerinden yapıldığı belirtilmişti, bu veriler kullanılarak yapılan hesaplamalar şarj durumu, sağlık durumu, deşarj derinliği, güç durumu gibi değerlerdir. Örneğin şarj durumunun hesaplanması için geliştirilmiş metotlarından bazıları voltaj metodu ve coulomb sayma (coulomb counting) metotlarıdır. Voltaj metodu, bataryanın voltajına göre daha önceden bilinen ve bataryanın üretimine göre belirlenmiş voltaj – şarj durumu eğrisine göre belirlenmesi yöntemidir. Bu yöntem anlık olarak bilgilendirme almamızı sağlayabiliyor olsa da batarya şarj veya deşarj durumundayken bataryanın değişken empedansından dolayı batarya voltajının değişmesi bataryanın kullanım durumlarında bu yöntemi hatalı kılabilmektedir. Bu yüzden bu metot Li-ion bataryaları üzerinden akım geçmediği durumlarda kullanılması tercih edilmektedir. Diğer yöntemlerden biri olan coulomb sayma yöntemi ise batarya üzerinden geçen akımı sürekli ölçerek zaman ile bataryaya giren veya çıkan enerjiyi izleyip şarj durumunun hesaplanması yöntemidir. Ancak hiçbir ölçüm mükemmel olamayacağı için zaman içerisinde sensör hatalarından kaynaklanan kaymalar olabilir. Bu bahsedilen iki yöntemin birleştirilmesi açıklarını birbiri üzerinden kapatabilen kombine bir yöntem oluşturur.

Bataryanın sağlık durumun hesaplanması için geliştirilmiş metotlardan bazılarına ise Ah (amper saat) sayımı, direnç testi, şarj döngüsü sayımı gibi pek çok metot vardır. Ah sayım metodu bataryanın üretildiği zamanki depolama kapasitesinin, boşken tam doluma gelinceye kadar bataryanın depoladığı enerjinin veya tam doluyken boşalıncaya kadar verdiği toplam enerjinin oranıyla hesaplanır. Bu yöntemin uygulanabilmesi için bataryanın tamamen boşaltılıp tamamen doldurulması ya da tersi olması gerektiğinden bu yöntemin daha çok deneysel olarak kullanıldığı söylenebilir. Direnç değişim testi yöntemi ise bataryanın zaman içerisinde ve kullanılarak kaybettiği sağlığının bataryanın iç direncini arttırması durumundan yola çıkarak yapılan hesaplama üzerine dayanır. Li-ion bataryanın yaşlanması iç direncini arttırır, bu durumda bataryanın üzerinden geçen akıma bağlı olarak bataryanın açık devre voltajından farklı bir voltajın iki terminalden okunmasına sebep olur. Şarj döngüsü sayımı yöntemi ise bataryanın sağlığının döngü sayısı yani bataryanın kaç kez şarj edilip deşarj edildiğinin sayısını tutarak hesaplanır. Bu yöntem için döngü sayısına göre bataryanın sağlığının ne olacağı bilinmesi gerekmektedir.

Li-ion bataryaların bunlar gibi pek çok parametreleri olup bu parametrelerin de karmaşıklıkları ve hassasiyetleri değişiklik gösteren hesaplama metotları vardır. Verilmiş olan örnekler üzerinden basit bir kullanım örneği verilmesi gerekirse. Bir Li-ion bataryanın üretildiği kapasitesi 4000mAh olduğu göz önüne alındığında, eğer %40 şarj durumundaysa ve sağlık durumu

da %75'e düşmüş ise bu bataryanın içerisinde kalan dolu ve kullanıma müsait kapasitenin hesabı yapılabilir. Bataryanın sağlığının %75 olduğu andaki kapasitesi 3000 mAh olup, şarj durumunun da %40 olduğu anda 1200 mAh kapasitelik kullanımı kaldığı çıkarımı yapabilir ve bataryanın kullanımına göre ne zaman biteceği veya dolumuna göre ne zaman dolacağı hesaplanabilir. Batarya yönetim sistemlerinde yapılan zamana bağlı hesaplamalar, veri iletimleri, veri depolamaları gibi pek çok işlemler, işlemcilerin kullanımını gerekli hale getirirler.

İşlemci tümleşik bir devredir ve komut setindeki komutları çalıştırıp, komutlara göre veri işleyen birimler olarak nitelendirilebilir. İşlemler genel olarak aritmetik işlemler, mantık işlemleri, veri manipülasyonları olarak kategorize edilebilir. Programlar bir dizi komuttan oluşur. Programlar insanların anlayabileceği dillerden başlayarak (yüksek seviyeli programlama dilleri) işlemcinin anlayabileceği dile yani makine diline kadar iner. Programlar ne kadar üst seviye bir dille yazılmış olursa olsun işlemcinin çalıştırması için en nihayetinde makine diline çevrilmesi gereklidir. Makine dili, ikili sayı sistemiyle ifade edilebilen ve işlemci mimarisine göre değişebilen bir dildir. Assembly dili ise bunun bir üstü olarak kabul edilen bir işlemci ailesine özgü kısa komutlardan oluşur.

Komut setlerinin karmaşıklığı baz alınarak işlemcileri iki ana kategoriye ayırmak mümkündür; indirgenmiş komut setli bilgisayarlar (RISC) ve karmaşık komut setli bilgisayarlar (CISC). İlk başlarda bilgisayar programları, doğrudan fiziksel yer kaplayan delikli kağıtlar üzerinde depolanıyordu. Bir programın kısaltılabilmesi doğrudan fiziksel bir hacme karşılık geldiği için, işlemcilerin işledikleri komutların çok işlevsel olması durum için çok avantajlıydı. Bu sebeple CISC mimarisi içerisindeki komutlar, tek bir komut ile doğrudan yüksek dile karşılık gelebilecek komutlara sahipti. Bu durum hem programı yoğunlaştırıyordu hem de program yazımını kolaylaştırıyordu. Ancak işlemci karmaşık bir işlemi tek bir komut halinde aldığı zaman işlemin tamamlanma süresi uzar komutlar arasındaki iş yapma süreleri değişkenlik gösterir ve boru-hattı mimarisi yapmayı zorlaştırır. Bu da işlemcinin yonga büyüklüğünü arttırıyordu. İşlemcinin hızını yavaşlıyorken harcanan güç ve fiyatı artırıyordu. 1970'lerde geliştirilmeye başlanan RISC mimarisi, veri depolama teknolojilerinin ilerlemesiyle ortaya çıkma imkanı buldu.

RISC işlemcileri için yazılan programlar, CISC işlemcileri için yazılan programlardan daha uzun olsa da RISC işlemci mimarileri, her saat çevriminde bir komut işleme felsefesi doğrultusunda çevrim saat hızı bakımından CISC işlemci mimarilerini geride bırakan bir performans sergiledi. RISC işlemci mimarilerinin komutları sadeleştirilmiş olması, komutların işlem sürelerini düşürmüş ve komutların işlem süresinin birbirine yakın olmasını sağlamıştır. Bu durum komutları kolayca bölüp boru-hattı mimarisi yapmayı kolaylaştırmıştır. İşlemci içerisine boru-hattı yapılması, işlemcinin komutları işlediği ünitelerin farklı kompartmanlara yerleştirilmesi anlamına gelir. Her bir saat çevriminde boru-hattı içerisinde ilerleyen komutlar sonraki kompartmana geçer. İşlemcinin saat çevrim hızı en yavaş kompartmana göre belirlenir. RISC-V, RISC felsefesi ile hazırlanmış bir komut seti mimarisi (ISA) standardıdır. RISC-V açık kaynaklı olması, esnekliği, özelleştirilebilirliği gibi yönler ile ön plana çıkmaktadır. İşlemciler tasarlanırken sadece uygulamaya özel entegre devre

(ASIC) olarak tasarlanmak zorunda olmayıp, aynı zamanda prototipleme, eğitim, test gibi amaçlar için kullanılmak üzere alanda programlanabilir kapı dizisi (FPGA) benzeri yapılarda tasarlanabilmektedir. Bu işlemci tasarımlarına soft işlemci denir.

FPGA'ler üzerinde tasarım modellemek amacıyla Verilog ve VHDL gibi donanım tanımlama dilleri (HDL) kullanılır. FPGA'lerin programlanması sentez programları aracılığıyla gerçekleştirilir. FPGA içerisindeki donanım ünitelerin HDL modeline uygun devre elemanları haline dönüştürülmesini sağlar.

Bu tezin yazımındaki motivasyon batarya yönetim sistemlerinin çalıştırdığı karmaşıklıkta bir programı çalıştırabilecek bir çekirdek tasarlamaktır. Buna ek ek motivasyon ise literatürdeki işlemci ve BYS arasındaki boşluktur. Bu tezde karmaşık görevleri yerine getirebilen bir işlemci tasarlanması hedeflenmiştir. Bu hedef 7 boru-hattı aşamalı bir soft işlemci tasarlayarak gerçekleştirilmiştir. Bu tasarım AMD'nin Vivado Design Suit (versiyon 2023.2) adlı programında Verilog HDL ile tarif edilmiştir. Tasarlanmış olan işlemci RISC-V ISA'sı içerisinde yer alan RV32I komut seti ile tasarlanmıştır. Yapılan testler neticesinde işlemcinin çalışması doğrulanmış olup, 100 MHz saat çevrim hızında AMD'nin Basys-3 FPGA kartına gerçeklenip test edilmiştir. İşlemcinin kritik yolu incelendiğinde tarlanmış olan işlemcinin 100 MHz saat çevrim hızının üstüne çıkabileceği tespit edilmiştir. Bu tespit üzerine Basys-3 ve Kintex-7 kartı üzerinde gerçeklenme durumlarında tüm sistemin ve yalnızca çekirdeğin çıkabileceği en yüksek saat çevrimleri bulunmuş ve paylaşılmıştır. İşlemcinin çalıştırması için assembly dilinde programlar yazılmış olup bu programlar makine diline C++ dilinde yazılmış olan tez için özel yapım bir program aracılığıyla dönüştürülmüştür ve bu C++ programı AtoM olarak isimlendirilmiştir. C++ kodunu yazmak için Visual Studio Community programının 2022 versiyonu kullanılmıştır. Assembly dilinden dönüştürülen makine dili doğrudan işlemci belleği içerisine gömülmüştür. Bellek, Basys-3 içerisindeki RAM bloklarından oluşturulmuştur. Testlerin doğruluğunu kontrol edebilmek için bir UART modülü tasarlanmıştır. Sensörler üzerindeki analog voltaj verilerini okuması için LogiCORE IP ile XADC modülü tasarlanmış ve işlemci ile uyumlu çalışabilecek hale getirilmiştir. Çekirdek ile çevre birimlerin ve belleğin uyumlu çalışabilmesi için yertutucu olarak bir veri yolu kontrol mantığı tasarlanmıştır.

Tasarlanmış olan işlemcinin, BYS işlemlerini yapabildiğinin denetlenmesi için bir devre tasarlanmıştır. Bu devre içerisinde kullanılan analog çıkış veren akım sensörü ACS712 ve sıcaklık sensörü LM35 kullanılmıştır. Voltaj ölçümü için voltaj bölücü devre kullanılmıştır. Kişisel bilgisayar ile işlemci arsında terminal üzerinden UART protokolü ile veri aktarımını ve veri alımını sağlayabilmesi için kişisel bilgisayar üzerinde PuTTy programı kullanılmıştır.

Tasarlanan sistem test edilmek için 100 Mhz saat çevrim hızında Basys-3 FPGA kartında gerçeklenmiştir. Teorik olarak, tüm sistem 130 MHz saat çevrim hızına, çekirdek ise tek başına 150 MHz saat çevrim hızına Basys-3 kartı üzerinde çıkabilmektedir. Kintex-7 kartı kullanıldığında, sistemin 212 MHz saat çevrim hızına, çekirdeğin tek başına 273 MHz saat çevrim hızına çıkabildiği saptanmıştır. Eğer varsayımsal bir program %45 ALU, %15 load, %10 store, %25 branch (hepsi

dallanıyor) ve %5 jump komutlarından oluşsaydı, programın işlemcide çalıştığında alınan ortama CPI 2.5 olurdu.

Batarya yönetim sistemleri gibi sistemlerde hesaplama yapması amaçlanarak tasarlanmış soft işlemci, gelecekte geliştirilebilecek şekilde tasarlanmış olup hesaplama gerekliliklerini sağlamıştır. Bu tez için tasarlanmış olan soft işlemci ve diğer birçok modül, LogiCORE IP blokları ve bazı hafıza kesitleri dışında, orijinal birer tasarımdır ve bu tez için yapılmıştır.

ACKNOWLEDGEMENTS

I want to express my deepest gratitude to everyone who supported me during the most challenging times of my master's thesis journey. Your support was not just helpful, but it was crucial. I wouldn't have made it without you.

First and foremost, I am sincerely indebted to my supervisor, Prof. Dr. Mustafa GÖK. With his deep knowledge, comprehensive experience and unshakeable patience, he pointed out the direction I should venture.

I would also like to extend my gratitude to the jury members, Assoc. Prof. Dr. Ahmet AYDIN, and Assoc. Prof. Dr. Zehan KESİLMİŞ Their objective comments and feedback were invaluable to me.

I am grateful to my colleagues who helped me at work and my superiors who supported me while I was writing my thesis.

Last but not least, I want to thank my mother Fatma POLAT, my father Namık POLAT and my sister Bircan A. POLAT. Their unwavering support and understanding, especially during the most challenging times of my thesis, mean the world to me.

LIST OF TABLES

LIST OF FIGURES

SYMBOLS AND ABBREVIATIONS

ALU : Arithmetic Logic Unit
AMD : Advance Micro Devices
ASIC : Application-Specific Integrated Circuit
AtoM : Assembly to Machine
BMS : Battery Management Systems
BYS : Batarya Yönetim Sistemi
CISC : Complex Instruction Set Computer
CPI : Cycles Per Instruction
CPU : Central Processing Unit
DC : Direct Current
DR : Data Register
DRFP : Decode Register File Pipeline
EEI : Execution Environment Interface
EF : Execution Forwarding
EWP : Execution Write-Back Pipeline
FCU : Fetch Control Unit
FDP : Fetch Decode Pipeline
FP : Fetch Pipeline
FPGA : Field Programmable Gate Arrays
HDL : Hardware Description Language
I/O : Input-Output
IP : Intellectual Property
ISA : Instruction Set Architecture
Li-Ion : Lithium-Ion
LS : Load Separator
LSB : Least Significant Bit
MIPS : Million Instructions Per Second
MSB : Most Significant Bit
MWBF : Memory Write-Back Forwarding
NOP : No-Operation
PC : Program Counter
PC : Program Counter

PCM : Protection Circuit Modules

R&F : Register and Forwarding

RAM : Random-Access Memory

RFEP : Register File Execution Pipeline

RFU : Register File Unit

RISC : Reduced Instruction Set Computer

RV32I : 32-Bit RISC-V Base Integer Instruction Set

SOC : State Of Charge

SOH : State Of Health

TR : Thermal Runaway

UART : Universal Asynchronous Receiver Transmitter

VHDL : VHSIC hardware description language

VHSIC: Very High Speed Integrated Circuit

VMS : Voltage Management Systems

WBLF : Write-Back Latency Forwarding

1. INTRODUCTION

Advancements in technology increased demand for electronic devices, electric vehicles, renewable energy investments, and so on. These demands exposed a gap in portable electrical storage advancements. To satisfy the need in this area, Li-ion batteries, which belong to the secondary battery category, was a great candidate (Matsumura, 2023).

Batteries

Batteries can store electric energy for a good period of time, and secondary batteries can be recharged. Li-ion batteries are one of the first picks for the job due to their properties of efficiency, specific energy, energy density, cycle life and calendar life. Even though a Li-ion battery cell will lose its properties over time, using the batteries under undesirable conditions may decrease the lifetime of the battery over the hill. Above the economic aspect, in radical cases, if a cell reaches critical conditions, it may catch fire or even explode. Even cell phones, notebooks, and many more daily used electronic devices can include Li-ion batteries (Crompton, 2000; Guo et al., 2022).

To prevent Li-ion batteries from reaching critical conditions, they should be monitored. The systems that monitor and manage batteries are called battery management systems (BMS). There are many types and functionalities of battery management systems. To achieve its functionalities, BMS needs to monitor the battery's temperature, voltage, current, and similar values. With the knowledge of these values, the other parameters can be calculated (Plett, 2015).

Battery Management System

BMS is simply responsible for two main duties: monitoring and protecting the battery. Knowing the condition of the battery is crucial for the user of the device. For example, a mobile phone user always wants to know the remaining battery charge. If the information is not available, the user cannot know when to charge it. The second duty of the BMS is to protect the battery when the operating conditions pass the advised operating conditions. To protect the battery, its current condition needs to be monitored. It is important to inform the user about the condition of the battery and to protect the battery from undesired conditions by changing the behavior of the electronic circuit that is also connected to the battery. This act increases the lifespan of the battery, protects human health and protects environmental health. In addition, BMS can also do active or passive balancing. This prevents the battery from becoming useless due to unbalanced charge distribution among cells (Warner, 2015; Hannan et al., 2017).

BMS protects the battery under the conditions of overcharge, over-discharge, over-current while charging, over current while discharging, over voltage while charging, under voltage while discharging, overheating, under temperature, and such conditions dealt with electronically and were

not included in this thesis (Gabbar et al., 2021). The only part of BMS that is included is the monitoring stage, and electronic parts are not included in this thesis.

BMS Parameters

It was stated that, in a BMS, the main calculations can be made from three main data. Using these data, state of charge (SOC), state of health (SOH), and many more parameters can be calculated. As an example, some of the methods to calculate SOC include the voltage method and the coulomb counting method. The voltage method calculates SOC using the voltage of the battery. In this method, the voltage of the battery can be calculated through an already known voltage-SOC curve. Although this method can be calculated almost instantly, if the battery has a current flow, this current changes the battery's terminal voltage due to its impedance. In this regard, this method is very useful when the battery is not charging or discharging. The other method, which is called the coulomb counting method, can be used while a current flows through the battery. In this method, SOC can be calculated by counting the energy that leaves or enters the battery over time. However, no measurement can be perfect, so by only using this method over time, the calculation can be shifted. By mixing the two methods that are mentioned, their weaknesses can be overcome by each other (Lelie et al., 2018; Cheng et al., 2010; Wang et al., 2020).

To calculate the SOH, there are many methods such as capacitance test, impedance test, charge cycle counting, and so on. The SOH calculation made by the capacitance test method can be calculated by getting the ratio of the maximum real (current) capacity of the battery over the initial (produced) capacity. The real capacity in this method can be calculated by fully recharging the empty battery or fully discharging the fully charged battery while calculating the energy stored or energy released with respect to charging or discharging the battery. For this method to be used, it is required to fully charge a fully discharged battery, or the opposite, for a correct calculation. For this reason, this method is more likely to be used for experimental purposes due to the expectation that the user of the device can't be expected to act as required. When it comes to the impedance test method, the health loss due to time passing or usage increases the impedance of the battery. So, the health of the battery can be estimated by calculating the internal impedance of the battery. Knowing the aging of the battery is associated with the SOH parameter. When the current flows through the battery, it changes the terminal voltage of the cell(s). Knowing the open circuit voltage and applying Ohm's law can reveal the required data. For the charge cycle counting method, the SOH can be calculated by counting how many times the battery is charged. To do so, it's required to know what the corresponding cycle's SOH is (Pradhan and Chakraborty, 2022; Tan et al., 2023).

Li-ion batteries have many parameters like the ones mentioned above, and those have changing precision and complexity according to methods that have been used to calculate. To give a simple use example of the parameters that have been mentioned before, considering a Li-ion battery which's capacity was 4000 mAh. If its SOC is 40% and its SOH dropped to 75%, its remaining

charged capacity can be calculated. With the knowledge of SOH being 75%, the current maximum capacity can be estimated as 3000 mAh, and considering the SOC is 40%, the remaining charged capacity can be calculated as 1200 mAh. With this knowledge we can estimate the remaining time until the battery is fully charged or remaining time until the battery is fully discharged according to current flowing through the battery (Pradhan and Chakraborty, 2022; Meng and Li, 2019; Wang et al., 2020; Tan et al., 2023). In the BMS, calculations, being able to understand changes in time, communication, data storing, and similar reasons forces BMS to run with a processor.

Processor

Processors are integrated circuits, and they execute instructions that are fetched from outside of the processor. Executing an instruction basically means processing the data according to the instruction's specification, which can be arithmetic operations, logic operations, data manipulation, and alike. Programs are set of instructions, and they can command the processor to do complicated processes. Programs can be expressed in languages, starting with high languages, which can be understood by humans, and they can be expressed in low languages, such as assembly mnemonics or machine language. Machine language is written down in a binary number system, and it is defined by the processor's architecture. It is the lowest level of programming language and the only meaningful language to the processor. On the other hand, assembly could be considered as one level higher than the machine language and can be explained as the direct translation of machine language. Even though it is still more primitive than high-level language, it can be more understandable than machine language. A processor's instruction set architecture (ISA) defines its own assembly language, so assembly language is specific to its processor's architecture. There are many processor architectures. However, two of them that will be mentioned are reduced instruction set computer (RISC) and complex instruction set computer (CISC) (Patterson and Hennessy, 2021).

In the early days of CISC processors, they used physical space for storage purposes. The programs were stored in punched cards. The bigger the program is, the bigger the space it occupies in a physical manner. So, the more functional a command means, the less storage space a program occupies. Due to this occurrence, the instructions were doing multiple actions in one go. In its era, when storage space was a nuance, doing multiple actions was a great way, and CISC architecture was doing its job on this. However, CISC processors had fatal problems. Having complex and many instructions makes the processor slower. Due to complexity, each instruction would have different time requirements to complete its job. Doing so creates problems in dividing them to build a pipeline. Also, a complex processor means more hardware, and more hardware means an expensive product. However, when the technology of the storage space improved, the requirement for physical storage space was abolished. With the improvement in data storage technology in the 1970s, the development of RISC architecture has begun (Feldman and Retter, 1994; Hennessy and Patterson, 2003).

Even though the programs written for RISC processors occupy more storage space than CISC processor's equivalent programs, RISC architecture applies one instruction per clock cycle philosophy to the design. When compared, RISC processors out scaled CISC processors in terms of clock frequency and price. RISC architecture had an advantage over performance due to its philosophy. While simplifying the instructions, the time required to execute them decreased, which made each instruction closer to the average execution time. Due to approximate timing requirements, dividing them to build a pipeline became easier. Building a pipeline in a processor means dividing the hardware into portions for each instruction. While the instructions go through stages in the pipeline, one per clock cycle, they get processed. Dividing the process also divides process time. This gives the ability to increase the frequency of the clock cycle and to do parallel processing (Feldman and Retter, 1994; Patterson and Hennessy, 2021). RISC-V is an instruction set architecture (ISA) designed with RISC philosophy. RISC-V has the properties of being open source, flexible, customizable, and so on. While processors are being designed, they are not obliged to be designed as application-specific integrated circuit (ASIC). They can also be designed for prototyping, testing, educating and similar reasons. For these reasons, they can be designed in field programmable gate arrays (FPGA). This type of processor designs are called soft processors (Patterson and Hennessy, 2021; Harris and Harris, 2022).

FPGAs can be programmed in Verilog hardware description language (HDL), very high speed integrated circuit (VHSIC) hardware description language (VHDL). In FPGA, the synthesization of described hardware can process data in real-time. Hardware defining in FPGA can be done by using programs like Vivado Design Suit, which is a product of Advance Micro Devices (AMD). The work principle of FPGA rests on the shoulders of switching. The corresponding circuit of the hardware description connects through switches and creates the circuit (Ünsalan and Tar, 2017).

Conclusion

The motivations of this thesis are to design a core that can execute programs to satisfy the needs of a BMS, and the late gap between processors and BMS processors in the literature. In this thesis, it’s aimed to design a processor which can do various complex calculations. The aim is achieved by designing a soft core by using AMD’s Vivado Design Suit in Verilog HDL. The ISA of the designed processor is a 32-bit RISC-V base integer instruction set (RV32I). The processor is designed with 7 pipeline stages. According to tests that have been done on the designed soft processor, the outcome has satisfied the desired. The designed soft processor has been implemented on AMD’s Basys-3 board. In the 2023.2 version of Vivado Design Suit, the designed processor can run at 100 MHz clock frequency. Considering the delay on the critical path, the processor can even reach a higher clock frequency. The results of the system and core implementations with maximum clock frequencies without timing violations for Basys-3 and Kintex-7 boards are given. The test

programs that run in the designed processor have been coded in the processor's assembly variant and then translated into machine language by a custom coded C++ program. The C++ program is coded in Visual Studio Community 2022 and named Assembly to Machine (AtoM). The converted machine language is implemented into the processor's memory module. The memory module is designed by using the random-access memory (RAM) blocks within Basys-3. Another module is defined to communicate with the processor via universal asynchronous receiver transmitter (UART). One of the features of the designed UART module is an adjustable baud rate. So, the processor can adjust the baud rate of the UART module any time. To read the voltage rating at the sensors, which gives analog output, the XADC pins of Basys-3 and XADC wizard intellectual property (IP) from LogiCORE within Vivado Design Suit are used. For harmony between the processor and the IP block, an XADC control logic is designed. To include compatibility and reachability between the core, peripherals and the memory, Bus Control Logic is designed as a placeholder.

To test the processor to see if it can be used as a BMS, a circuit has been designed. This circuit includes AC712, a current sensor, LM35, a temperature sensor, a DC-DC boost converter and voltage dividers for the voltage readings. PuTTy program is used as a terminal application on the personal computer to communicate with the processor by using the UART protocol.

The system is tested with 100 MHz clock frequency in Basys-3 FPGA board. Theoretically, the system can reach 130 MHz clock frequency, and if the core gets implemented alone, it can reach 150 MHz clock frequency within Basys-3 board. In Kintex-7 board, the maximum system frequency is 212 MHz clock frequency and maximum core frequency is 273 MHz clock frequency. If an imaginary program consists of 45% ALU, 15% load, 10% store, 25% branch (all branches) and 5% jump instructions, the CPI of the program is 2.5.

The implemented soft processor is designed to execute tasks at the degree of BMS calculations within a relatively small size. Also, this processor is designed to be available for improvements in the future. The soft processor design and the other modules used in this thesis, apart from the LogiCORE IP block and apart from some parts of the memory, were originally created for this thesis work.

2. PRELIMINARY WORK

Without batteries, most of the mobile devices we use in our daily lives are useless, and we can be at risk of immobilization in the near future. Portable electronic devices increased our quality of life with the scale of addiction. Moreover, with the increase in the electric car industry, batteries are also crucial in the sector. Suppose that the requirement of storage units in the renewable energy sector is included. The importance of batteries and their stability due to environmental and economic reasons becomes more evident (Matsumura, 2023).

2.1. Batteries

Batteries can convert chemical energy to electric energy and vice versa within the same device. They can be categorized as primary and secondary batteries according to their rechargeability. While primary batteries are non-rechargeable types of batteries, secondary batteries are rechargeable types of batteries (Warner, 2015; Crompton, 2000). For portable devices, it is more convenient to use rechargeable batteries. Since in the long run, there is no need to change the batteries constantly, which reduces the cost. In Figure 2.1, battery types and battery examples are given.

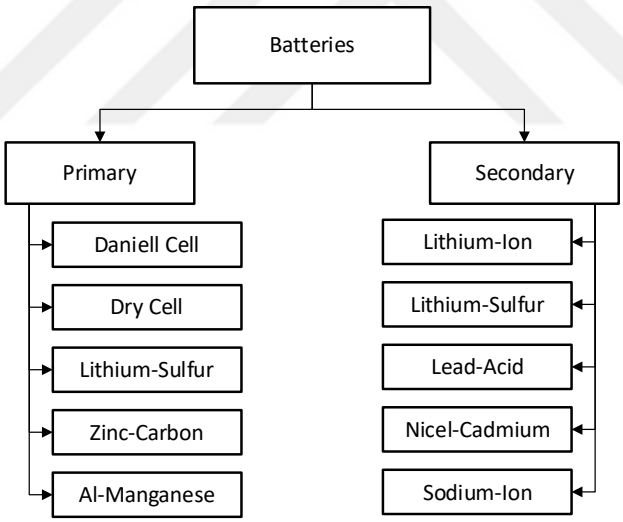

Figure 2.1. Battery Types and Battery Examples (Dutta et al., 2023)

Within the given examples in Figure 2.1, Li-ion batteries are preferred due to their advantage over the ones within the same category. The advantages of Li-ion batteries are as follows; high energy density, lack of memory effect, stability, low-self discharge rate, cycle life, reduced cost, and many more (Chombo and Laoonual, 2020)

2.1.1. Lithium-Ion Batteries

The structure of a Li-ion battery consists of casing, separator, electrolyte, and electrodes. Commonly used cathode materials include LiMn2O4, LiFePO4, and NCM. When it comes to the

anode silicon, carbon, and lithium titanite are examples of commonly used base materials. When Li-ion batteries are compared with other similar batteries like Ni-Cd, Ni-MH, and lead-acid, the main reason for their commercialization becomes clear. Li-ion has a longer cycle life, higher energy density, and higher C rate when compared to others (Yang et al., 2023; Matsumura, 2023).

Over its advantages, Li-ion batteries have disadvantages like aging and thermal runaway (TR). While aging can occur over time, it can also happen under undesirable operating conditions. Undesirable operating conditions also cause TR. TR can cause the ignition of batteries and even lead to explosions. To prevent the TR from occurring, the battery needs to be protected physically and electronically. While TR can occur due to physical damage, overtemperature, and under temperature, it can also happen electronically caused by overvoltage, undervoltage, overcurrent, and short circuit. *Overvoltage* can damage the battery internally by changing the cell structure. The excess lithium deposited on the anode can cause an internal short circuit. On the other hand, *under voltage* is the condition of discharging a battery below its cutoff voltage. In the end, under voltage causes the battery to change structurally and ages it drastically while increasing safety risk. Overcurrent causes overheating due to the impedance of the battery, also causing a higher transfer rate over lithium more than diffusion, which results in local overcharge or discharge. When the temperature of a battery is too high, reactions such as electrolyte decomposition are accelerated. Moreover, if it is under temperature, ionic mobility slows down and lithium ions may be plated on the surface of the anode during the occurrence of discharge. *Short circuits* can happen within the cell or outside of it. If it occurs outside and doesn't get treated, the cell or the pack can suffer from permanent damage by leading the voltage to 0 volts and considering the overtemperature, it can lead to the TR. If it occurs within the cell, it may get treated by its localized melting acting as a fuse, and the cell will have a high discharge rate, if it survives. If it doesn't, it may trigger TR (Matsumura, 2023; Chombo and Laoonual, 2020; Barsukov and Qian, 2013; Weicker, 2013; Plett, 2015).

While some manufacturing errors can lead to TR due to defective production of cell structure, it can be minimized, but it can be an impossible task to reduce its occurrence probability to zero. Mechanical damages like penetration, crushing, and similar situations can cause cells to leak and even short circuit internally or externally. Leading to TR (Plett, 2015; Weicker, 2013).

Aging increases the occurrence probability of most failures in Li-ion batteries. While aging cannot be prevented, it can be slowed down. While high temperature can accelerate aging, aging causes increased internal impedance, reduced capacity and increased self-discharge rate. Internal impedance increases over aging, and it leads to power fade by decreasing the power that a cell can provide. Aging decreases the capacity due to unrecoverable chemical reactions, leading to capacity fade. Also, due to permanent changes in the cell's structure over time, aging leads to an increase in self-discharge rate (Plett, 2015; Weicker, 2013).

While explosion and ignition of the battery must be prevented at all costs, aging should be slowed down. Battery management systems (BMS) can help the users of the battery by monitoring and managing the cell(s) while providing rather safer and longer usage of the battery.

2.2. Battery Management Systems

In general, there are two types of Battery management systems (BMS). One being the centralized BMS, and the other being the distributed BMS. As the name suggests, the distributed BMS is separated on a hardware level and controlled with a master unit. The centralized BMS is just a central unit that controls the battery. This being the case, the distributed BMS can work more accurately and with higher precision due to its separated slave units, which can monitor and manage individual sections of the battery. However centralized BMS is more economical (Warner, 2015). In Figure 2.2, the generalized structure of centralized BMS and separated BMS is given.

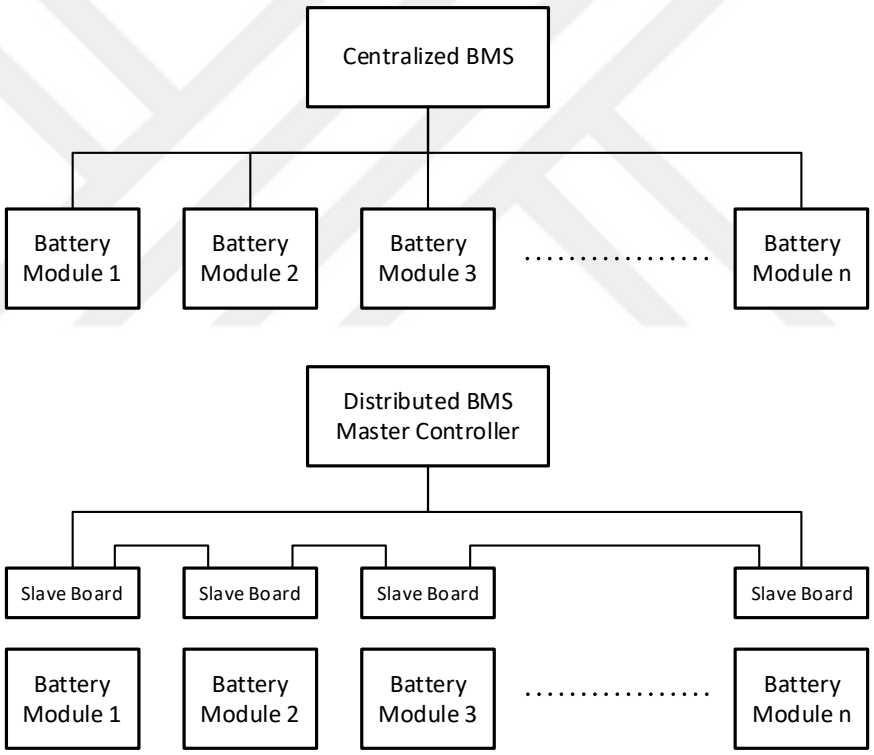

Figure 2.2. Overview of Centralized BMS and Separated BMS (Warner, 2015)

There are also other terms that are used to indicate the BMS. Such as voltage management systems (VMS) or protection circuit modules (PCM). While the unique definition of BMS can't be mentioned, its operational roles are as followings, monitoring the battery, protecting the battery, estimating the state of the battery, maximizing the performance of the battery and reporting the battery to users or external devices. It is not an obligation for a BMS to perform all the given functions (Andrea, 2010).

2.2.1. Monitoring the Battery

In the simplest form, batteries need to be observed to be managed. In this manner, it's essential to monitor the battery for other steps to know what to do. Knowing the state of the battery creates the opportunity to prevent damage in the battery or worse. There are many parameters to be observed in the battery. However, mainly voltage, current, and temperature of the cells are important in Li-ion batteries (Matsumura, 2023; Weicker, 2013; Pradhan and Chakraborty, 2022; Lu at al., 2023).

2.2.2. Protecting the Battery

Protection of the battery is very important in Li-ion batteries due to its unforgiven nature. BMS protects the battery with an electronic circuit under undesirable conditions. Undesirable condition examples are as follows: exceeding the maximum charging or discharging battery current, exceeding the maximum charging voltage or minimum discharge voltage and high temperature (the parameters can change depending on the battery). To protect the battery, BMS can limit the current flow into or out of the battery, and it can disconnect the battery from the circuitry. Cooling down the battery is also another option for high temperatures (Andrea, 2010; Hannan et al., 2017).

2.2.3. Maximizing the Performance of Battery

To increase the performance of the battery, every bit of delay in the aging counts towards the goal due to delaying the capacity fade, delaying the power fade and delaying the increase of self-discharge rate. Also, balancing the battery plays a crucial role. While using a battery pack, a difference in parameters like battery capacity can cause significant problems. Considering a pack with 2 cells serial to each other and with different capacities. When the battery starts to get used, the difference between the state of charge (SoC) parameters of the cells will increase. Over time, the cell with low capacity will reach the cut-off voltage while the other cell having a remaining charge. If the battery still gets used even though one cell reaches cut-off, the battery can cause safety risks. The opposite is also correct, trying to charge an imbalanced cell can cause an overcharge, if not noticed (Matsumura, 2023; Rahimi-Eichi et al., 2013). Imbalance in a battery pack can cause some cells to expose overvoltage due to impedance and some cells can be overcharged, early charged and early discharged due to capacity fade (Barsukov and Qian, 2013). Imbalance in a cell can happen due to several reasons like the difference in the production of the cells, undistributed temperature in the battery, using cells that have different initial capacities, using cells that have different self-discharge rates and difference of state of charge (SOC) in between cells (Lelie et al., 2018; Rahimi-Eichi et al., 2013). In Figure 2.3, visual examples to imbalanced cells can be observed.

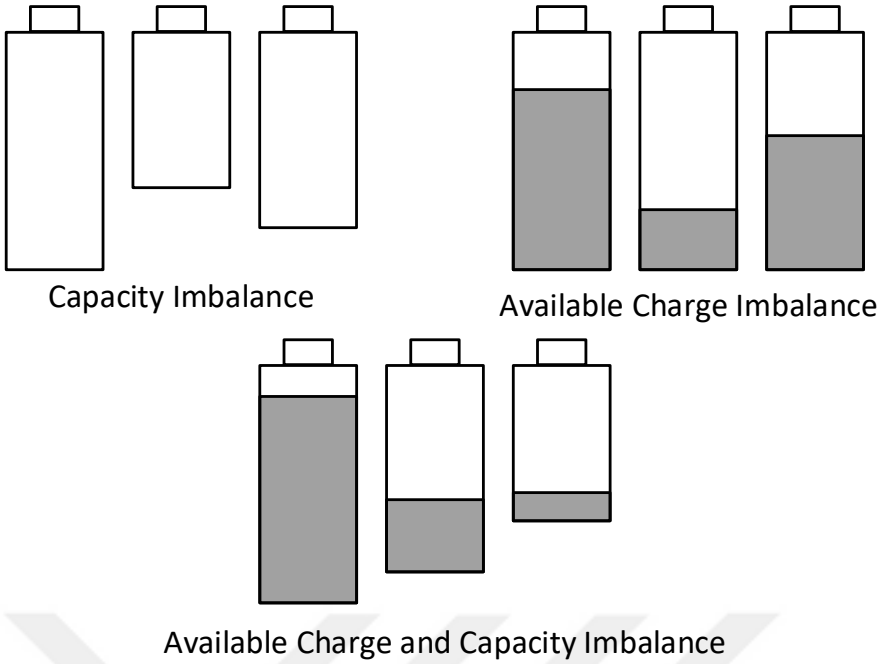

Figure 2.3. Some Variations of Imbalanced Cells (Weicker, 2013; Yang et al., 2023)

There are two types of balancing. One being the passive balancing the other being the active balancing. In the passive balancing, contained energy in the cell(s) which have higher SOC than the others, get converted to heat energy until they reach the same SOC. This method is used in serially connected cells to ensure the initial condition of the battery pack starts within a balanced state. As the name suggests, this method is passive. While one resistor is connected to one cell, the switching occurs if the cell is imbalanced. Switching is controlled by BMS. However, this method loses energy rather than preserving it in another cell, lacks the efficiency and it's not practical to be used in a large battery, like the ones in an electric vehicle. On the other hand, the active balancing method can preserve the excess energy of a cell into another cell. While being the downsides of active balancing such as difficult control strategy, higher cost of the circuit and more complexity in the design, the upsides can be mentioned as higher efficiency, the ability to relocate energy within a cell to another cell, active balancing method can balance the cell while not resting and it can be used on large battery packs like the batteries of electric vehicles (Warner, 2015; Tan et al., 2023).

2.2.4. Estimating States of Battery

The monitored values can be used directly in the state estimation. As an example of frequently used state estimations can be given as state of charge (SOC) and state of health (SOH). However, to be capable of estimating states, the equivalent circuit of the battery may be required (Shen and Gao, 2019; Hannan et al., 2017). In Figure 2.4, some simple equivalent circuit models of batteries are given.

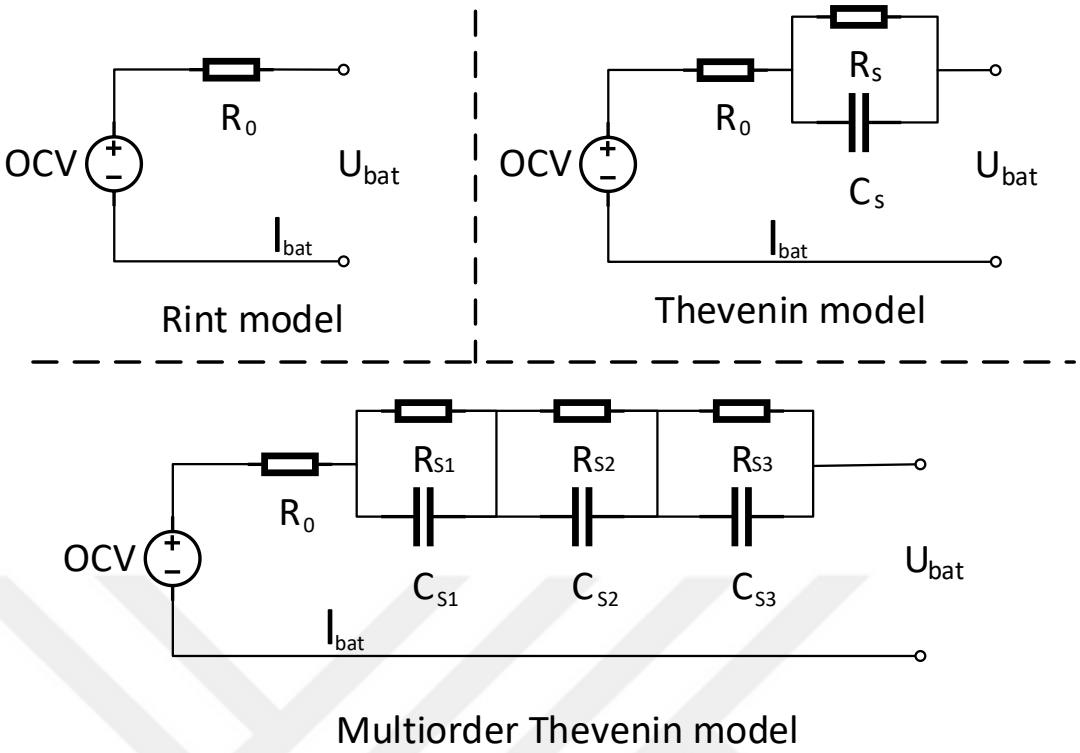

Figure 2.4. Equivalent Circuit Models of Batteries (Shen and Gao, 2019; Andrea, 2010)

SOC parameter indicates the remaining charge in the battery at a percentage. This indication is about the battery's current charged capacity over its current maximum capacity. However, this parameter does not indicate the maximum capacity of the battery. There are many methods to calculate SOC parameter. Some of the methods are, open circuit voltage, coulomb counting and Kalman filter methods. According to the open *circuit voltage method*, the SOC can be associated with battery voltage using the data specified by the producer of the cell. However, while the method is accurate, it lacks the capability to estimate SOC while current flows through the battery due to its changing internal resistance according to aging. *Coulomb counting method* is about the current passing through the battery, and it also referred to as the Ah counting method. In coulomb counting, the charging or discharging state of the battery is observed according to the charge the battery receives or provides. However, this method can be flawed if it is used long enough without taking reference of SOC from time to time. This situation is caused by the imperfection of the measuring and the time constant. If two methods (open circuit voltage method and coulomb counting method) are combined as a mixed method, the drawbacks of the two can cover each other. While the open circuit voltage method makes an accurate estimation of SOC while the battery is resting, the estimation can be used as a reference point for coulomb counting. On the other hand, coulomb counting can be used to estimate SOC while the battery is not resting, and the combination of the two makes an accurate estimation method (Hannan et al., 2017; Rahimi-Eichi et al., 2013; Ali et al., 2019; Espedal et al., 2021; Yang et al., 2023; Andrea, 2010; Weicker, 2013). The *Kalman filter*

method does not just estimate SOC of a battery, it also is an algorithm to estimate the state of a dynamic system. It can be used in many fields like trajectory and position calculation, signal processing, control systems, and alike. Kalman filter is important in state estimation areas, especially with noisy inputs (Tan et al., 2023; Weicker, 2013; Yang et al., 2023; Espedal et al., 2021; Ali et al., 2019).

The SOH parameter indicates the health of a battery at a percentage rate. The health of the battery is important in safety. The health can be calculated from the aging of the battery, which is directly associated with the current maximum capacity of the battery, impedance of the battery, ability to accept charge, ability to discharge, cycle count, and so on. The techniques to estimate SOH include capacity test, ohmic resistance test, destructive technique, cycle number counting, voltage curve and many more. The *capacity test technique* is about the ratio between the current capacity of the battery over the capacity when it was produced. Although this technique is simple, it requires accurate measurement of the battery. The battery needs to be emptied and charged fully, or it needs to be discharged fully, while it is fully charged. While the charge or discharge process is happening, the battery needs to be accurately monitored to calculate its total capacity. The *ohmic resistance test technique* is about calculating the resistance of the battery. The less healthy cells will have, the higher their internal resistance will be. This affects the maximum power deliverance of the battery. The open circuit voltage of the battery drops when the current battery passes through the battery. Although the battery has a capacitive equivalent circuit, if direct current (DC) is used and if the cell rests in the same state for a while, the resistance can be calculated according to voltage change in the voltage measurement of the battery. If open circuit voltage, current and voltage change are known, the resistance can be calculated from Ohm's law. The *destructive* technique focuses on the integrity of the cell. It requires the battery to be destroyed. After that, X-ray diffraction, scanning transmission electron microscope and other scanning techniques can be used to monitor the cell. Due to the destruction of the battery, this technique is more suitable for the laboratory. The *cycle counting technique* is about aging which is relative to usage. The charge-discharge count can be used on a look-up table to define the health of the cell. However, the relationship between the cycle count and health needs to be defined by the producer of the cell. The *voltage curve technique* uses the voltage curve of the cell under high current. Rather than discharge current, charge current is a better sample stage due to its rather more stable nature. So, when the large current passes through the battery, sampling starts. The calculated voltage curve is determined and compared with known curves to estimate SOH (Pradhan and Chakraborty, 2022; Xiong et al., 2018; Yang et al., 2023; Weicker, 2013; Tan et al., 2023).

2.2.5. Reporting the Battery to Users or External Devices

A BMS is required to communicate with internal devices and external devices to keep its operations in an optimal state. The communication should be high speed, and the protocol should be

compatible with multiple devices while the interface being bidirectional. Serial communication is common in BMS to use minimum dedicated lines. To give an example of internal connection, a master-slave can be given. As an example to external devices, charge controller, power controller, etc., can be given (Tan et al., 2023; Weicker, 2013; Andrea, 2010).

2.3. Processor

A processor is an integrated circuit. It is also referred to as central processing unit (CPU) and microprocessor. At its core, processors process instruction stream or tread, which is called a program. These programs that run-in processors consist of 1's (ones) and 0's (zeros), which is also referred to as a binary number system. However, for a processor, it is no different from a language, and it's called machine language. Machine language can be symbolized with mnemonics to be understood by a human. The mnemonic form of the machine language is called assembly language. Assembly language is formed by the instruction set architecture (ISA), and it is specific to that architecture. while there are many processor architectures, two of them are reduced instruction set architecture computer (RISC) and complex instruction set architecture computer (CISC) (Winans, 2022; Tanenbaum, 2013; Ledin and Farley, 2022).

RISC architecture is made by the philosophy of one simplified instruction per cycle. However, CISC architecture designs were shouldering the burden of programmers to write less lines for coding and the lack of storage space to make compact instructions. Due to the design architecture, the integrated circuit was complicated and big. When the first computers were made, they didn't even have digital storage space like the regular computers we have today. They were using punched cards to store programs. Due to punched cards using physical space to store data, bigger programs meant more punched cards and more physical space. Even after the technological advances in storage space, the general allocated space for a computer wasn't big enough to be wasted. The reasoning from storage space and many technological withdraws were making the CISC architecture optimal design since the storage space technology improves to provide enough space for RISC architecture to be implemented (Feldman and Retter, 1994).

RISC architecture requires larger storage space than CISC architecture because of the philosophy of RISC. The philosophy makes the instruction simpler. While CISC can achieve many operations with just one instruction, RISC may need several instructions to do the same operations. However, this raises the problem of storage space in RISC designs, but also, the design philosophy makes up with the advantages of an increase in speed, less complex designs and the cost of the processor. The cost of the processor decreases simply because of less complex designs leads to less hardware. When it comes to speed, although having complex instructions can slow down processors simply by existing, the programmers were not mastering the use of ginormous amounts of instructions CISC processors had which makes the design less effective. While simplifying the design with RISC philosophy makes the instruction process time closer to each other, it also

decreases the total time requirement of an instruction to execute. A simple design combined with relatively close process times significantly helps to implement a pipeline (Feldman and Retter, 1994; Parhami, 2005).

2.3.1. Pipeline

The change in the processor, if it was designed by RISC philosophy, significantly affected its data path by simplifying it. This effect results in an easier way to implement a pipeline by having less numbered and relatively the same process-timed instructions. To explain in a simple manner, pipelining is dividing the main process into sub-processes. So, as an example, a product in an assembly line of a factory can be given. If the product requires T second of process time and if this process is divided into N levels of stages, while the stages have similar time to be completed and feed the assembly line, the output period of the assembly line will be T/N second, but only after the first output. In conclusion, the pipeline in this example provides the product with less time than the total process time of a product. This can be achievable due to the simultaneous process at each stage of the pipeline. While the process time is not shrinking, the processed product at a given time (excluding the initial state of the pipeline) is equal to N. While pipelining can be designed for a processor to increase its output rate by increasing the clock cycle frequency, it comes with various disadvantages (Feldman and Retter, 1994; Patterson and Hennessy, 2021; Tanenbaum, 2013).

As mentioned in section 2.3, the processor executes treads, which are called programs. This creates the least wanted thing in the pipeline, variations. While the ISA of a RISC processor is rather simple, it still has variations of instructions, and these instructions can be conditional, which leads to branches. Branch instructions indicate a possibility of changing the special register called program counter (PC). So, if a branch occurs, all the pipeline needs to be flushed to progress in a different path in the program. This is required due to the pipeline is filled with unfinished instructions which need to be disposed of. If an unwanted instruction gets executed, can change the program calculations or even worse. Flushing also includes reverting any changes in the pipeline flushed instructions done. To get the full potential from a pipeline, the pipeline needs to be filled again (Feldman and Retter, 1994; Patterson and Hennessy, 2021). The pipeline stages in RISC designs include fetch, decode, register file read, execution, address generation, data memory access, and write back. In Figure 2.5, classic pipeline of a RISC processor is given

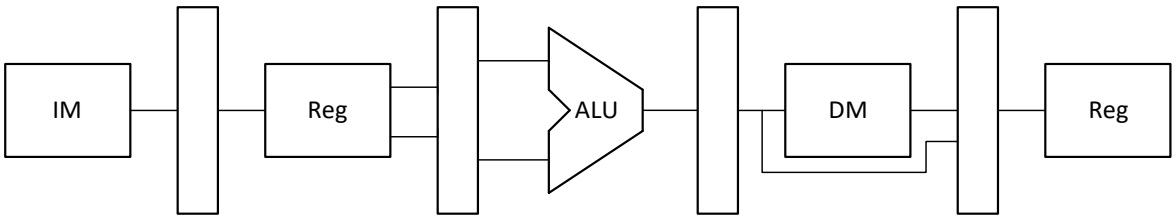

Figure 2.5. Classic Pipeline of a RISC Processor (Patterson and Hennessy, 2021)

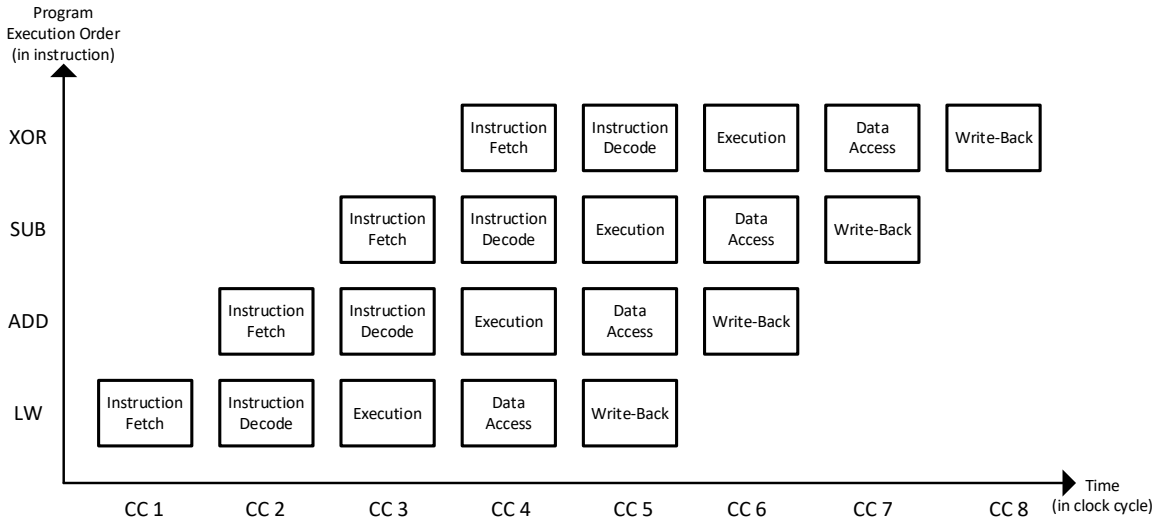

Figure 2.6. Simultaneous Processing of Instructions Within Pipeline (Patterson and Hennessy, 2021; Feldman and Retter, 1994)

In Figure 2.6, simultaneous processing of instructions within pipeline can be observed. In the given examples of pipeline (Figure 2.5 and Figure 2.6), the structure of the pipeline is an example, and it is not fixed. The structural design of the pipeline in RISC designs could be changed according to design requirements. As an example, the heavy weight of the pipeline is the arithmetic logic unit, and it could be divided into more stages rather than just one. However, the stages also remain the same without a pipeline. This means the instructions need to be fetched, decoded, executed, and may be accessed and written back depending on the instruction (Feldman and Retter, 1994).

2.3.2. Some Stages of a RISC Processor

While all stages are important and can't be thrown away, the fetch stage has the importance of many. To start the execution of instructions, it must be fetched from the memory, which is typically a ram or cache. To store the location, there is a dedicated register called the program counter. PC register stores the address of the fetched instruction. As the name suggests, the value in the PC is increased if there is no request of it to change from the instructions that have been executed or from an exception (Patterson and Hennessy, 2021; Harris and Harris, 2022; Parhami, 2005).

The instructions fetched from memory goes to instruction decode. In instruction decode, the fetched instructions get decoded according to ISA. As an example of RISC-V ISA starting from the least significant bit (LSB) of an instruction, the first 7 bits indicate the opcode. The opcode indicates the operation code, such as load and store. This section deciphers ISA into more usable indicative forms or separations like the index of a register, which will be accessed to prepare the data path for execution. The register file, which stores the current value of the registers, is also located here. The decoded indexes of the registers lead to the register file, and the value of the corresponding register is given to the output (Patterson and Hennessy, 2021; Harris and Harris, 2022; Parhami, 2005).

After the instruction is decoded, the more meaningful data is passed to the execution stage. The operations are made in the execution unit. The operations can include arithmetic, logic, address calculations, and branch decision-making. The main unit in this section is arithmetic logic unit (ALU). While it receives input, the decoded data plays a crucial role in multiplexers and operation executions. Depending on the design requirements like speed and power consumption, the operation can be executed after the mux decision according to the decoder for less power consumption or all the operations can be executed, and the output can be picked from the desired operation (Patterson and Hennessy, 2021; Harris and Harris, 2022; Parhami, 2005).

The next stage, which named as data access, is about instructions that use memory, which are load and store instructions. If the instruction is neither of them, the stage is irrelevant. However, this stage is crucial for memory access. If a store is executed, the value is sent to the address, or if an address is loaded, the data is delivered here to write back into register file in the next step. As an example, some designs have separate bus for memory instructions. Specially for load instructions, if the instruction bus and memory bus is not separated, this may lead to the stall of the pipeline till the bus is free (Patterson and Hennessy, 2021; Harris and Harris, 2022; Parhami, 2005).

At the final destination of the pipeline, the data, which is the result of an instruction execution, is written to an indexed register in the register file. The write-back stage is an optional stage, which means it depends on instruction. As an example, store instruction stores the data from an indicated register in the register file to a calculated address in the memory and store instruction in RISC-V ISA doesn't require the write-back stage. In processors, some structures or units, such as pipeline stall, forwarding logic, control unit, and many more, may be required for more functional designs. (Patterson and Hennessy, 2021; Harris and Harris, 2022; Parhami, 2005).

2.3.3. Pipeline Stall

Under some circumstances, the pipeline cannot proceed and requires stalling. The stalling can be made by freezing the pipeline or inserting No-Operation (NOP) instructions. If a NOP is used, the pipeline works as usual, but it is prevented from executing meaningful results. If the pipeline is frozen, the pipeline stops until the condition is resolved. Pipeline stalls are important to resolve issues like data hazards, control hazards, structural hazards and memory access delays. However, stalling the pipeline reduces its efficiency. There are other ways to solve these problems rather than stalling the pipeline some example solutions can be given as forwarding logic and branch prediction (Patterson and Hennessy, 2021; Feldman and Retter, 1994; Parhami, 2005).

2.3.4. Pipeline Hazards

Hazards are referred as incorrectness or inefficiency in the pipeline. Examples can be given as data hazards, control hazards and structural hazards. The *data hazards* happen within the program flow. When the instruction operand is overwritten just before it is executed, the executed instruction

is required to reach the write-back stage. Waiting for the operand to reach write-back stage causes delay. If not waited operand's old value gets used in the execution stage unless forwarding logic is used. The *control hazard* is about branches. When a branch instruction is issued, there are two conditions: either the branch isn't taken, and program flow continues, or else the branch is taken, and PC is changed. So, when the branch is taken, the instructions that are already taken and inside the pipeline becomes a problem. Because the program flow shouldn't execute the instructions that come after the branch, the instructions in the pipeline after performing branch may cause unpredictable changes in the program flow. So, if a branch is taken, the instructions that are already in the pipeline and instructions that are already fetched according to program flow must be flushed. This causes pipeline inefficiency. To decrease the inefficiency, branch prediction can be used. The *structural hazards* are related to the hardware of the processor. When the request from an instruction forces the pipeline to be stalled, this is called structural hazard. This hazard could have been caused by two instructions that require the same hardware resource or an instruction that requires processor's normal flow to be disturbed. As an example, if a processor has only a word length bus that leads to the memory, this may cause stalling in the pipeline when a load instruction is executed due to fetch unit won't be able to request data at that moment. About the memory access delay, the memory instructions may require a stall due to the delay to reach and read from the memory, and if a unit like catch doesn't contain the address, it fetches from other memory units until it finds the data (Patterson and Hennessy, 2021; Parhami, 2005; Horie, 2014).

2.3.5. Forwarding Logic

In a pipeline, if one instruction uses the result of the instruction that just came before it as an operand, the forwarding logic can bypass in between stages to prevent pipeline stalling. The example situation for forwarding logic is given in Figure 2.7 as assembly code. As it can be seen in Figure 2.7 the result of the first line is used in the next instruction as an operand, which requires a pipeline stall for execution to be written in the register file so it can be read as operand (Patterson and Hennessy, 2021; Parhami, 2005).

```
add x1, x1, x2
add x1, x1, x2
add x1, x1, x2
```

Figure 2.7. Example of Forwarding Logic Working Condition (Patterson and Hennessy, 2021)

However, when indexes of the operands and index of the executed instruction destination register are compared and decided if they are equal, the output of the executed instruction can be directed back to ALU by bypassing the register file. This structure is called forwarding logic. Even

though it is bypassed, the execution output is written to the register file at the writeback stage (Patterson and Hennessy, 2021). In Figure 2.8, simplified forwarding logic data path is given.

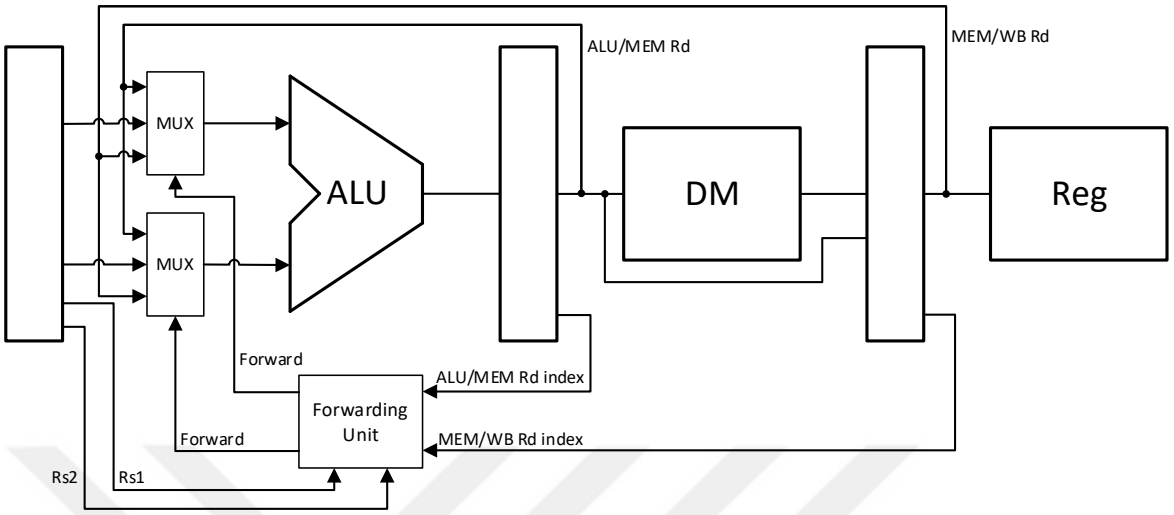

Figure 2.8. Forwarding Logic Data Path (Patterson and Hennessy, 2021)

2.3.6. Branch Prediction

In a pipelined processor branch instructions create high inefficiency when the branch instruction forces the pipeline to flush due to change in control flow. To lessen the impact of branch instructions, the branch prediction unit predicts the branch outcome. While it may depend on the architecture of the processor, the instruction type is revealed in the decode stage. Until the decode stage, the program flow is fetched as regular, and if the branch instruction is revealed in the decode section, the prediction hardware decides on if the branch instruction may lead to a branch or not. While there are more advanced techniques to do branch prediction, simpler techniques can be given as static branch prediction and dynamic branch prediction. Static branch prediction is the simplest of all prediction techniques. The *static branch prediction* method relies on the instruction information. This means that when the instruction is revealed as a branch instruction, the fetch process may change according to the type it uses. Some types of static branches are as follows: always predict the instruction doesn't branch, and always predict the instruction branches and predict the instruction branches if the branch is backward. The static branch prediction has the advantage of having minimal hardware requirements. However, in complex programs, the pattern of branch instructions may lead to a miss in the static prediction method. When it comes to *dynamic branch prediction*, the method uses more hardware but has a higher correct prediction ratio. The dynamic branch prediction method uses index to store the history of branch instructions. The index is the portion of the PC taken from the LSB side. When a branch is revealed, the index comparison is made, and the behavior of the instruction is checked from the record. If the instruction has not been recorded before, it starts to record. The prediction can be decided from the prediction buffer technique that is used to record the

earlier behavior of the instruction. The techniques include the 1-bit branch prediction buffer and the 2-bit branch prediction buffer. In a 1-bit branch prediction buffer, the old prediction is recorded as taken or not taken. When the prediction is wrong, the bit is reverted. However, recording only one outcome of the instruction may create frequent flushes in frequently alternating branches. On the other hand, the 2-bit branch prediction buffer requires two repeated failed predictions to revert, and the states can be classified as strongly taken, weakly taken, weakly not taken and strongly not taken. So, in the dynamic prediction method, the index is compared and if there is a recorded history, the prediction is decided according to past behavior. The branch prediction methods increase the efficiency of the pipeline by decreasing the flushes (Tanenbaum, 2013; Patterson and Hennessy, 2021; Parhami, 2005).

2.3.7. Pipeline Flushing

If there are instructions in the pipeline that are not valid anymore, the non-valid instructions need to be discarded before they get executed. Flushing also includes reverting any effects those non-valid instructions had. This requirement can be caused by a change in control flow, and this can happen because of instructions like jump and branch, and it can also happen because of exceptions (Feldman and Retter, 1994; Patterson and Hennessy, 2021).

2.3.8. Control Unit

Within a single-cycle processor, handling the instruction control is rather easier than in pipelined processors. In pipelined processors, the required control signals must be generated while the instruction is processed through the pipeline. Having control signals through different stages of the pipeline to be processed correctly creates complexity. While there can be a main control unit, the control unit could be separated within the pipeline structure. Some designs may even have a hybrid approach. Using smaller control units may reduce the latency of the control unit. The control unit controls the data flow, timing, fetching, branching and some other control-related operations (Patterson and Hennessy, 2021; Tanenbaum, 2013; Ledin and Farley, 2022; Harris and Harris, 2022; Parhami, 2005).

2.3.9. Exceptions

Exceptions occur for situations when something doesn’t go as planned. When a change in the control flow of instructions other than branches are referred to as exceptions. While there are hardware exceptions within processors, there are also other types of exceptions that are caused by programs. External hardware and asynchronous signals can be given as examples. When an exception occurs. It's handled by the exception handler. Address alignment fault, undefined instruction fault, divide by zero fault, and page fault can be given as examples. In pipelined processors, some

exceptions are treated as control hazards, and they should be handled (Waterman and Asanović; Patterson and Hennessy, 2021; Ledin and Farley, 2022).

2.4. RISC-V ISA

RISC-V is an open-source computer instruction set architecture which designed to create efficient, robust and flexible processor architectures. RISC-V includes a variety of base instruction sets and extensions. A RISC-V processor must at least include a base instruction set. In RISC-V architecture, a word is defined as 32-bit (Patterson and Hennessy, 2021). The behavior of a RISC-V program can change depending on the execution environment. The RISC-V execution environment interface (EEI) defines the behavior of the processor (Waterman and Asanović, 2019). In Table 2.1 some extensions of RISC-V are given.

Table 2.1. Some Extensions of RISC-V (Waterman and Asanović, 2019)

Subset	Name	Implies
Integer Multiplication and Division	M	Zicsr
Atomics	A	F
Single-Precision Floating-Point	F	IMADZifencei
Double-Precision Floating-Point	D	D
General	G	
Quad-Precision Floating-Point	Q	
Decimal Floating-Point	L	
16-bit Compressed Instructions	C	
Bit Manipulation	B	
Dynamic Languages	J	
Transactional Memory	T	
Packed-SIMD Extensions	P	
Vector Extensions	V	
User-Level Interrupts	N	
Control and Status Register Access	Zicsr	
Instruction-Fetch Fence	Zifencei	
Misaligned Atomics	Zam	A
Total Store Ordering	Ztso	

2.4.1. RISC-V Base Instruction Set

RISC-V 32-bit base integer instruction set (RV32I) can emulate most of the other ISA extensions. RV32I is one of the simplest forms of the ISA. However, it's also sufficient to support modern operating system environments. While the developers were designing RV32I, they included reduced hardware requirements as a high priority. The RV32I contains 40 unique instructions. The instruction set uses 32 general-purpose registers and one register for PC, with each one having a 32-bit length. Other than one of the general-purpose registers, all of them can be read and can be overwritten by instructions. The special register that can't be changed is register x0. The x0 register

always has a fixed value of 0, and it cannot be changed by any instructions. The fact that the x0 register can't be overwritten allows instructions to dump unnecessary writebacks to it without overwriting a register. Always being able to read zero values can be used to move the registers, useful for reference, and many more. The rest of the 31 general purpose registers (x1 to x31) are used to read and store Boolean values, or as two's complement signed binary integers or unsigned binary integers within the processor. They are included due to fast reachability purposes. PC contains the address of the current instruction (Waterman and Asanović, 2019).

2.4.2. RV32I Base Instruction Set Formats

In the RV32I base integer instruction set, there are mainly four instruction format types, and all the instructions are 32-bit wide. The format types are R-type, I-type, S-type and U-type. The formats are used to determine the decoding of the instruction. The address of the instructions must be aligned with a 4-byte boundary in the memory (Waterman and Asanović, 2019). In Figure 2.9, the representation of R-type, I-type, S-type and U-type, and extended types B-type and S-type due to immediate generation is given.

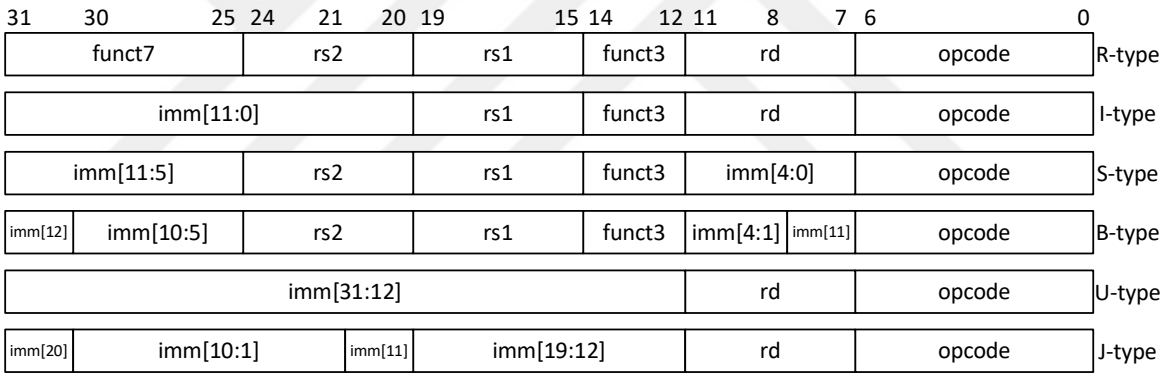

Figure 2.9. Representation of Base Instruction Format Types (Waterman and Asanović, 2019)

The formats shown in Figure 2.9 also represent the essential parts of the instructions. The first section is opcode, which is the abbreviation of operation code. As the name suggests, it represents the operation code of the instruction in a general way. While some instructions may not include fields for indexes of rd, rs2, etc., opcode defines the instruction type, and it's a must to have in the instruction. The other identifiers of instructions are funct3 and funct7. They identify the instructions further if they use the same opcodes. However, unlike opcode, funct3 and func7 are not obliged to exist, as can be seen in Figure 2.9. For the indexes of registers, rd, rs1 and rs2 fields are used and orderly represented as; destination register, first source register and second source register. They represent one of the 32 registers. (Waterman and Asanović, 2019; Winans, 2022).

2.4.3. Integer Register-Immediate Instructions

In this section, the computational instructions for immediate and register use are given. The immediate is referred to the value given within instruction. Although it is given, it is required to be at least extended or reordered to be useful (Waterman and Asanović, 2019). In Figure 2.10, instance of instructions which use I-type format are given. Given examples to instructions include ADDI, SLT, SLTU, ANDI, ORI and XORI.

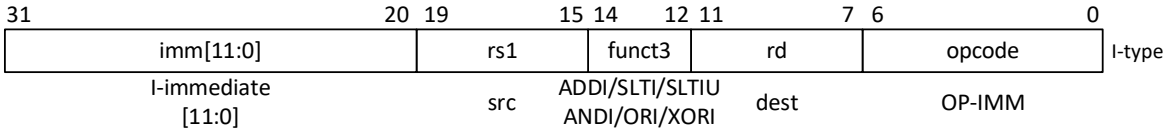

Figure 2.10. Instruction Format of ADDI, SLT, SLTUI, ANDI, ORI and XORI Instructions (Waterman and Asanović, 2019)

In ADDI instruction, register rs1 is added to sign-extended 32-bit immediate. When overflow occurs, it is ignored. The result is written in register rd. ADDI instruction also operates two's compliment. This allows the instruction to subtract according to sign-bit, which is the bit 31 of the instruction (Waterman and Asanović, 2019).

In SLTI (set less than immediate) instruction, the signed comparison between register rs1 and sign-extended immediate is made. If the value in register rs1 is bigger than the sign-extended value immediate represents, value 1 is written in rd. But if the register rs1 is smaller than the sign-extended immediate, the value 0 is written in rd (Waterman and Asanović, 2019).

In SLTIU (set less than immediate unsigned) instruction, the unsigned comparison between register rs1 and sign-extended immediate is made. Even though the comparison is unsigned, the immediate is sign-extended till 32-bit with. If the value in register rs1 is bigger than the sign-extended value immediate represents, value 1 is written in rd. But if the register rs1 is smaller than the sign-extended immediate, the value 0 is written in rd (Waterman and Asanović, 2019).

For ANDI, ORI and XORI instructions, they perform logical bitwise operations. Bitwise AND, OR, XOR operations are executed between the sign-extended immediate and register rs1 and the result is written to register rd (Waterman and Asanović, 2019).

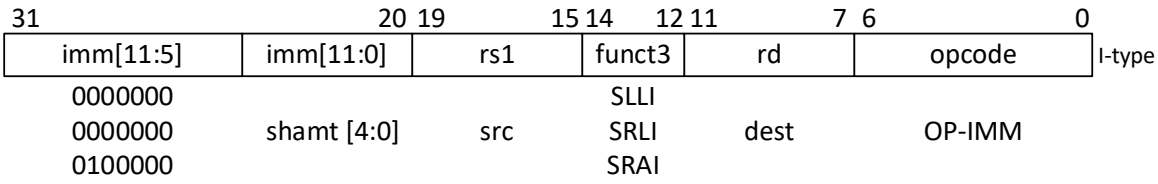

Figure 2.11. Instruction Format of SLL, SRLI and SRAI Instructions (Waterman and Asanović, 2019)

In Figure 2.11, instruction format for SLL, SRLI and SRAI instructions is given. The main difference between the format in Figure 2.10 and Figure 2.11 is the immediate part. SLL, SRLI and SRAI instructions use specialized versions of the I-type format. The shamt is the 5-bit portion of immediate value. For SLL, SRLI and SRAI instructions, the shamt determines how many bits will be shifted. The rest of the immediate is left as logic 0, except for the SRAI instruction. In SRAI instruction, bit 30 determines if the instruction is SRAI or SRLI. This situation is caused by SRAI and SRLI using the same opcode and func3. By setting the bit 30, the instruction becomes SRAI. If not set, it is SRLI (Waterman and Asanović, 2019).

In SLLI (shift left logic immediate) instruction, the value in register rs1 is logically left shifted by the value of shamt. The shift is made by adding zero to the right of the value, basically multiplying by 2 per zero added. The extended side at the left is terminated to maintain a 32-bit width in a register. The result of the operation is written in register rd (Waterman and Asanović, 2019).

In SRLI (shift right logic immediate) instruction, the value in register rs1 is logically right shifted by the value of shamt. The shift is made by adding zero to the left of the value, basically dividing by 2 per zero added. The extended side at the right is terminated to maintain a 32-bit width in a register. The result of the operation is written in register rd (Waterman and Asanović, 2019).

In SRAI (shift right arithmetic immediate) instruction, the value in register rs1 is arithmetically right shifted by the value of shamt. The shift is made by adding the most significant bit (MSB) to the left of the value, which is basically sign extending. The extended side at the right is terminated to maintain a 32-bit width in a register. The result of the operation is written in register rd (Waterman and Asanović, 2019).

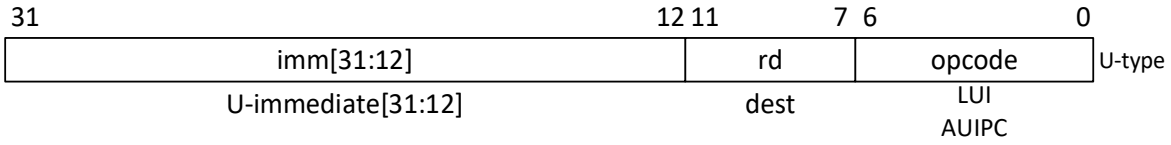

Figure 2.12. Instruction Format of LUI and AUIPC Instructions (Waterman and Asanović, 2019)

In Figure 2.12, the instruction format of LUI and AUIPC instructions is given. The instructions shown in Figure 2.12 use U-type instruction format, which gives them 20-bit immediate to use. These instructions don't need funct3 and funct7 because their opcodes directly indicate them (Waterman and Asanović, 2019).

In LUI (load upper immediate) instruction, the 20-bit immediate of the instruction is placed top 20 bits of the register rd. and then the lowest 12 bits are filled with zeros. The result of the operation is written in register rd (Waterman and Asanović, 2019).

In AUIPC (add upper immediate to program counter) instruction, a 32-bit value is created using the 20-bit immediate of the instruction. As in LUI, the lowest 12 bits of the created value is filled with zeros, and upper 20 bits of the created value is directly taken from the instruction. Then,

the value created from the instruction is added to the PC. The result of the operation is written in register rd. However, AUIPC instruction doesn't change the PC other than increasing it as a computation instruction. Only the result of the addition is written to register rd (Waterman and Asanović, 2019).

2.4.4. Integer Register-Register Instructions

In this section, the computational instructions that use rs1 register and rs2 register are given. Computational register-register instructions use R-type instruction format and don't use values directly from the instruction. The instructions indicate; operation, source registers and destination register (Waterman and Asanović, 2019). In Figure 2.13 instruction format of integer register-register instructions is given.

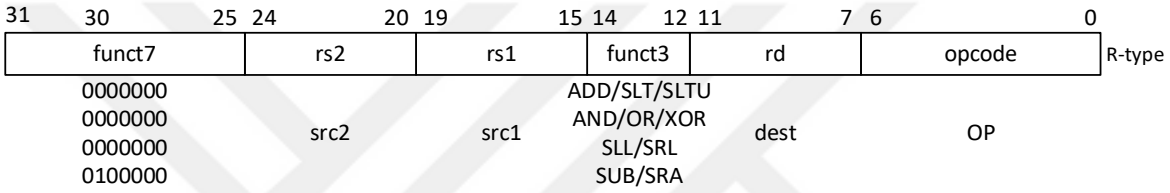

Figure 2.13. Instruction Format of Integer Register-Register Instructions (Waterman and Asanović, 2019)

While the ADD instruction adds rs1 and rs2, then writes to rd, the SUB instruction subtracts rs2 from rs1, then writes to rd. The overflow is ignored. Bit 30 makes the separation between ADD instruction and SUB instruction. If bit 30 is set, the instruction indicates the operation is subtraction. If bit 30 is not set, the instruction indicates the operation is addition (Waterman and Asanović, 2019).

The SLT (set less than) and SLTU (set less than unsigned) instructions compare the values of register rs1 and register rs2. If register rs1 is smaller than register rs2, the value 1 is written in register rd. If rs1 is not smaller than rs2, the value 0 is written in rd. The difference between SLT and SLTU is that SLTU instruction compares the values, considering they are unsigned. SLT compares them, considering they are signed (Waterman and Asanović, 2019).

SLL (shift left logic), SRL (shift right logic) and SRA (shift right arithmetic) instructions shifts the value in register rs1 according to the value at the lower 5 bits of register rs2. While SLL does left shift, SRL and SRA instructions do right shift with the difference of logic right shift or arithmetic right shift. SRL and SRA use the same opcode and the same funct3. If the bit 30 in the instruction is set, this indicates the instruction is SRA. If bit 30 is not set, this indicates the instruction is SRL (Waterman and Asanović, 2019).

2.4.5. Unconditional Jump Instructions

RV32I base integer instruction set offers two unconditional jump instructions, JAL and JALR. While JAL instruction uses J-type instruction format, JALR instruction uses I-type instruction format. Unconditional jump instructions change the PC when executed according to immediate and register if JALR is used. When they are executed, PC+4 is stored to destination register (Waterman and Asanović, 2019). In Figure 2.14, JAL instruction format is given.

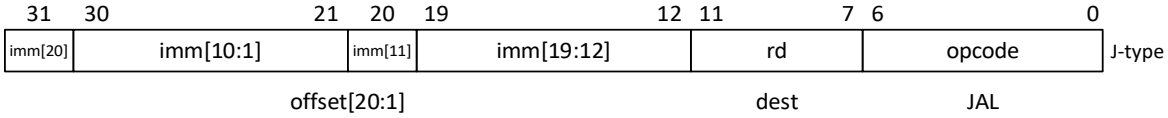

Figure 2.14. Instruction Format of JAL instruction (Waterman and Asanović, 2019)

When the JAL (jump and link) instruction is executed, the reordered and sign-extended immediate is added to PC to form the jump target address. After that, the PC value is changed with the address which calculated, and the address of the next instruction (PC+4) is stored in register rd. If the calculated address is not aligned with four-byte boundary, the instruction-address-misaligned exception is generated (Waterman and Asanović, 2019).

In Figure 2.15, JALR instruction format is given. When the JALR (jump and link register) instruction is executed, sign-extended immediate is added to rs1 and then LSB is set to zero, this forms the jump target address. After that, the PC value is changed with the address which calculated, and the address of the next instruction (PC+4) is stored in register rd. If the calculated address is not aligned with four-byte boundary, the instruction-address-misaligned exception is generated (Waterman and Asanović, 2019).

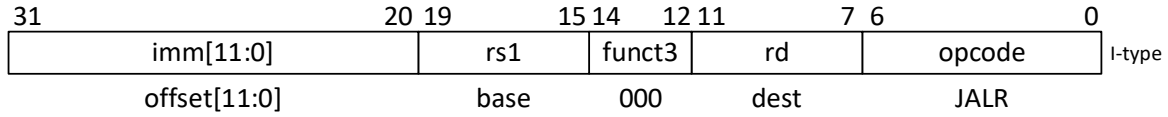

Figure 2.15. Instruction Format of JALR instruction (Waterman and Asanović, 2019)

2.4.6. Conditional Branch Instructions

All conditional branch instructions use B-type instruction format. The 12-bit immediate is reordered and sign-extended. The created value is required to be multiples of two. Then the value is added to its address to calculate the target address. If the comparison returns to requiring a branch, the processor continues to execute instructions from the targeted address. If the comparison doesn't return to requiring branch, the processor continues to execute instructions regularly (Waterman and Asanović, 2019). In Figure 2.16 instruction format of conditional branch instructions is given.

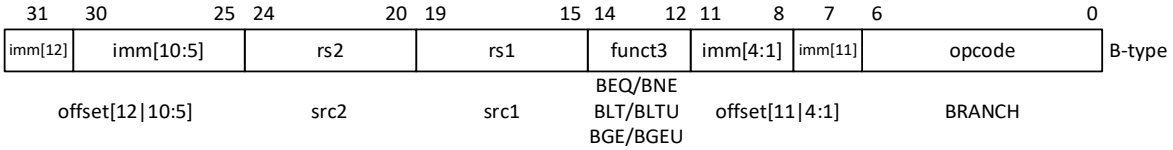

Figure 2.16. Instruction Format of Conditional Branch Instructions (Waterman and Asanović, 2019)

BEQ and BNE instructions check for equality. While BEQ checks if the two registers, register rs1 and register rs2, are equal, BNE checks if the two registers are not equal. So, for BEQ, if they are equal, branch occurs, and the register PC is changed with the target address and if they are not equal, the register PC remains the same. For BNE, if they are not equal branch occurs and register PC is changed with the target address. If they are equal, the register PC remains the same (Waterman and Asanović, 2019).

BLT and BLTU instructions check if the register rs1 is less than the register rs2. If the comparison is true, branch occurs and register PC is changed with the target address. If the comparison is not true, register PC remains the same. The difference between BLT and BLTU is that BLT checks the registers considering register values are signed and BLTU checks the registers considering register values are not signed (Waterman and Asanović, 2019).

BGE and BGEU instructions check if register rs1 is greater or equal to register rs2. If the comparison is true, branch occurs, and the register PC is changed with the target address. If the comparison is not true, the register PC remains the same. The difference between BGE and BGEU is that BGE checks the registers considering register values are signed, and BLTU checks the registers considering register values are not signed (Waterman and Asanović, 2019).

2.4.7. Load Instructions

Load Instructions are used to load data from memory to register, and they use I-type instruction format. With the immediate and register rs1, an address is calculated by adding sign-extended immediate to register rs1. The result of the calculation is a 32-bit wide address, and it is byte-addressed, meaning that the address specifies which portion of the value will be loaded. The value in the specified address is loaded to the register rd. If the alignment of the address doesn't match, the exception is generated depending on EEI. In Figure 2.17, instruction format of load instructions is given. There are a total of 5 different load instructions (Waterman and Asanović, 2019).

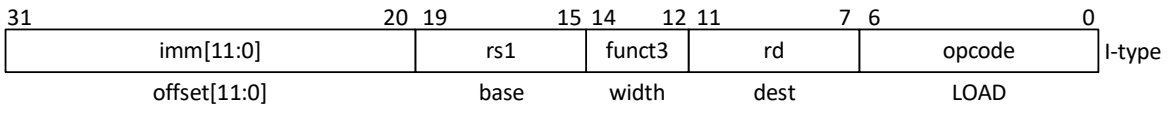

Figure 2.17. Instruction Format of Load Instructions (Waterman and Asanović, 2019)

The LW (load word) instruction loads the value of the address into register rd. LH (load half) instruction loads half of the value, which is 16-bit, then sign-extends the value to 32-bit before writing to register rd. LHU (load half unsigned) instruction loads half of the value, which is 16 bits, and then zero-extends the value to 32 bits before writing to register rd. LB (load byte) instruction loads 8-bit values, then sign-extends the value to 32-bit before writing to register rd. LBU (load byte unsigned) instruction loads 8-bit values, then zero-extends the value to 32-bit before writing to register rd. The loaded portion of the value is decided on the address calculated in the instructions LH, LHU, LB and LBU. EEI defines the type of the endianness configuration, whether little-endian or big-endian (Waterman and Asanović, 2019).

2.4.8. Store Instructions

Store instructions are used to store data from registers to memory, and they use S-type instruction format. Just like the load instructions, with the immediate and register rs1, an address is calculated by adding reordered and sign-extended immediate to register rs1. The result of the calculation is a 32-bit wide address and its byte-addressed. The value in the register rs2 is stored at the calculated address of the memory. If the alignment of the address doesn't match, the exception is generated depending on EEI. In Figure 2.18, instruction format of store instructions is given. There are a total of 3 different store instructions (Waterman and Asanović, 2019).

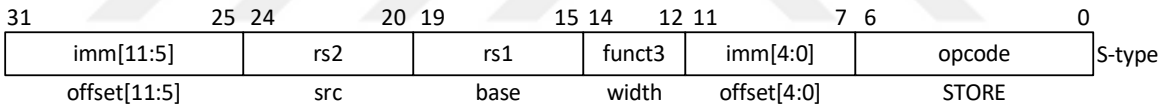

Figure 2.18. Instruction Format of Store Instructions (Waterman and Asanović, 2019)

The SW (store word) instruction stores the word in rs2 to calculate the address in memory. SH (store half) instruction stores the 16 lowest bits of the register rs2 into the calculated address in memory. SB (store byte) instruction stores the 8 lowest bits of the register rs2 into the calculated address in memory (Waterman and Asanović, 2019).

2.4.9. FENCE and SYSTEM Instructions

In Figure 2.19, instruction format of FENCE and SYSTEM instructions are given. FENCE instructions are used for memory ordering, and SYSTEM instructions consisting of ECALL and EBREAK instructions (Waterman and Asanović, 2019).

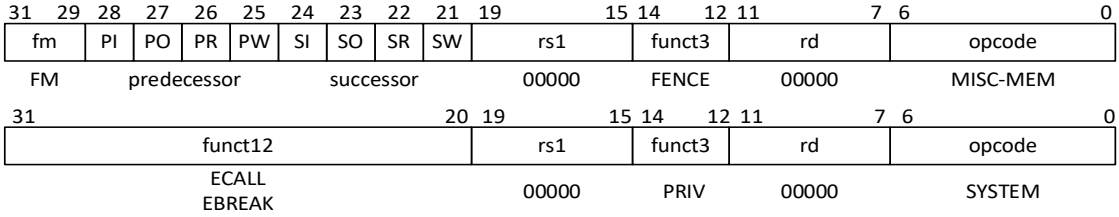

Figure 2.19. Instruction Format of FENCE And SYSTEM Instructions (Waterman and Asanović, 2019)

2.4.10. RV32I Machine Language Representation and Their Assembly Translation

In Table 2.2, assembly equivalents of RV32I base integer instruction set instructions can be observed. In Table 2.3, machine language representation of RV32I base integer instruction set instructions are given.

Table 2.2. Assembly Equivalents of RV32I Base Integer Instruction Set Instructions (Patterson and Waterman, 2017)

Category	Name	Format	Notation
Shifts	Shift Left Logical	R	SLL rd,rs1,rs2
	Shift Left Log. Imm	I	SLLI rd,rs1,shamt
	Shift Right Logical	R	SRL rd,rs1,rs2
	Shift Right Log. Imm.	I	SRLI rd,rs1,shamt
	Shift Right Arithmetic	R	SRA rd,rs1,rs2
	Shift Right Arith. Imm.	I	SRAI rd,rs1,shamt
Arithmetic	ADD	R	ADD rd,rs1,rs2
	ADD Immediate	I	ADDI rd,rs1,imm
	SUBtract	R	SUB rd,rs1,rs2
	Load Upper Imm	U	LUI rd,imm
	Add Upper Imm to PC	U	AUIPC rd,imm
Logical	XOR	R	XOR rd,rs1,rs2
	XOR Immediate	I	XORI rd,rs1,imm
	OR	R	OR rd,rs1,rs2
	OR Immediate	I	ORI rd,rs1,imm
	AND	R	AND rd,rs1,rs2
	AND Immediate	I	ANDI rd,rs1,imm
Compare	Set <	R	SLT rd,rs1,rs2
	Set < Immediate	I	SLTI rd,rs1,imm
	Set < Unsigned	R	SLTU rd,rs1,rs2
	Set < Imm Unsigned	I	SLTIU rd,rs1,imm
Branches	Branch =	B	BEQ rs1,rs2,imm
	Branch ≠	B	BNE rs1,rs2,imm
	Branch <	B	BLT rs1,rs2,imm
	Branch ≥	B	BGE rs1,rs2,imm
	Branch < Unsigned	B	BLTU rs1,rs2,imm
	Branch ≥ Unsigned	B	BGEU rs1,rs2,imm
Jump & Link	J&L	J	JAL rd,imm
	Jump & Link Register	I	JALR rd,rs1,imm
Synch	Synch thread	I	FENCE
	Synch Instr & Data	I	FENCE.I
Environment	CALL	I	ECALL
	BREAK	I	EBREAK

Table 2.3. Machine Language Representation of RV32I Base Integer Instruction Set Instructions (Waterman and Asanović, 2019)

RV32I Base Instruction Set						
imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm | pred | succ		rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

2.5. UART

UART (universal asynchronous receiver transmitter) is a serial communication that is used in the communication of devices. UART works on asynchronous clock domains. The key principle of UART is to use asynchronous clocks, but also a pre-determined format of communication. The baud rate, start condition, width of the data, parity bits, stop bits and idle state are predetermined. Some example modes of UART are half duplex and full duplex. If UART communication is done on

only one communication line, this is called half-duplex. In half duplex devices, it takes turns to communicate. If UART communication is done on two communication lines, one for transmission and one to receive, this is called full duplex (Ünsalan and Tar, 2017; Axelson, 2007; Frenzel, 2016).

Considering there are two units communicating through UART with one line for transmission and the other for receiver. While transmitting data under the limitations of the predetermined format, the idling state requires the line to be on logic high. This indicates that there is no data to transmit until the line goes logic low. When the line goes logic low, this indicates the transmission is beginning, which is also called the start bit. Because the transmission is done within asynchronous clock domains, at least the time between the bits needs to be known. The baud rate indicates bits per second. So, the next bit is issued according to the baud rate. After the start bit, the data comes bit by bit, starting from the LSB of the data, and data length is also predetermined. After the MSB of the data is sent, parity bits are sent optionally. As an example, even parity and odd parity can be given. With even parity bit, the total number of logic high in the data bits and parity bits required to be even. For the odd parity bit, the total number of logic high in the data bits and parity bits required to be odd. If the parity bit exists, it is determined whether it is logic high or logic low according to its type and data. After the parity bit, the stop bit is set. Stop bit is always logic high, and it can be more than 1 bit. After that, the stop bit is sent. The transmitter either starts to idle for new data to transmit or it starts right away (Ünsalan and Tar, 2017; Axelson, 2007; Frenzel, 2016).

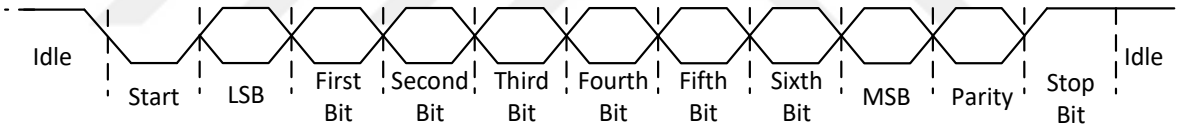

Figure 2.20. Example UART Frame (Ünsalan and Tar, 2017)

When receiving the data, the line gets sampled by the receiver till the logic low is seen. Logic low indicates that the data is about to be received. According to the predetermined baud rate, the time between bits is known. So, the data bit will be received after the time it takes to send 1 bit. However, due to asynchronous clocks, the sample of the start bit might be shifted according to sample time. To prevent possible shifts of read data bits, the sampling count should be at least 16 times in a period of 1 bit, and the sampling location should be the middle of the data for accurate readings. So, after reading the start bit the first time, the second sampling for the LSB should be made after a 1.5-bit period. After the LSB is sampled, the time required to wait for the next middle point is a 1-bit period, and this situation keeps its state until the stop bit is reached. In Figure 2.20, an example of a UART frame with 8-bit data is given (Ünsalan and Tar, 2017; Axelson, 2007; Frenzel, 2016).

3. THE SYSTEM DESIGN

In Figure 3.1, the system block diagram is presented.

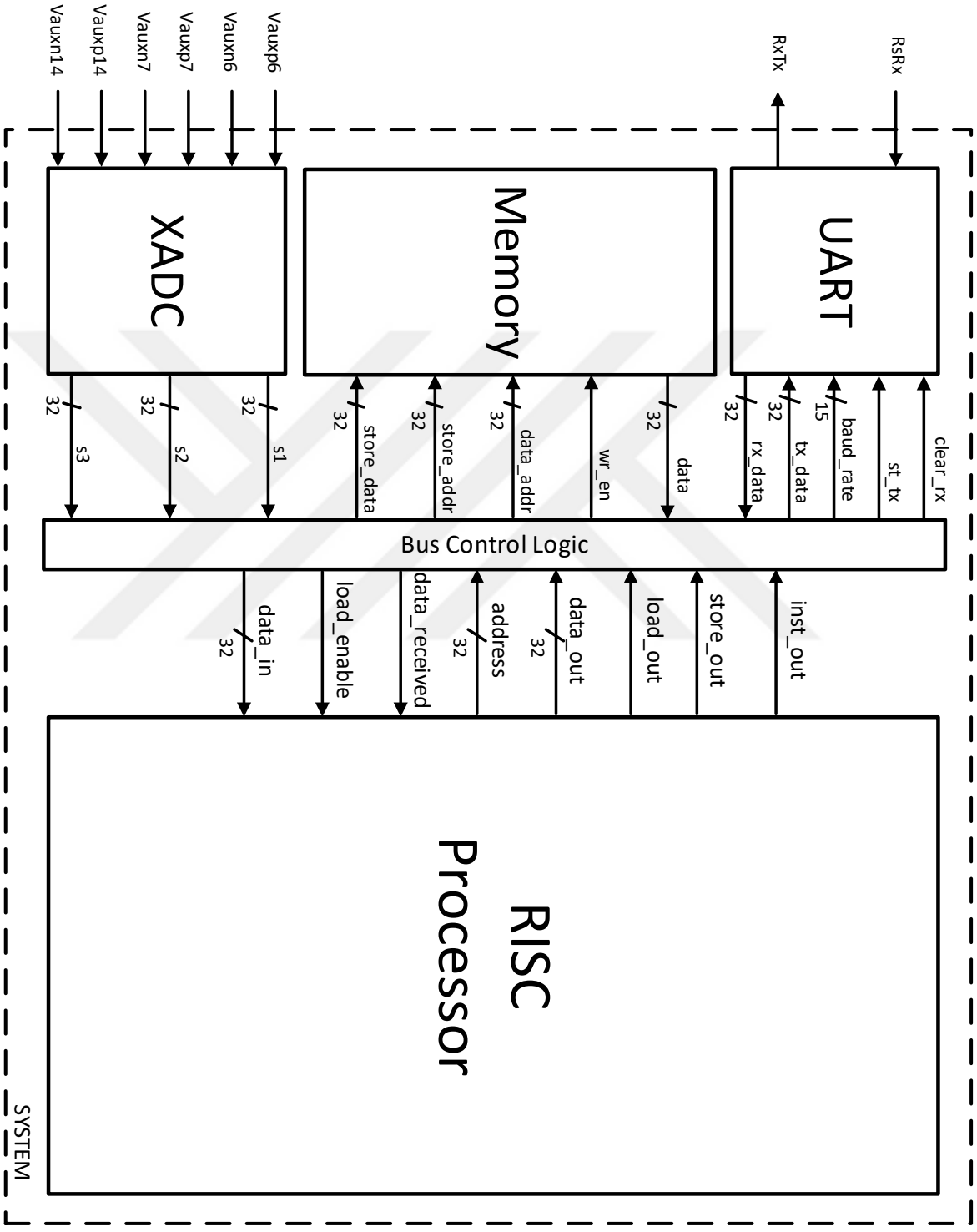

Figure 3.1. The Block Diagram of The System

The block diagram shows the RISC core, Memory, Bus Control Logic, UART and XADC peripherals and their connections. The Bus Control Logic orchestrates the data traffic among the modules, manages the peripheral units according to the core's commands and gives feedback to the core if required. Within it, a custom bus protocol is used.

The design of the Bus Control Logic and Memory Unit is made according to the RISC core's needs as a placeholder. The pipelined RISC core is designed to work on a cache. However, the bus protocol is unstandardized. In the representation of the system, the Memory Unit does not directly correspond to RAM or cache.

3.1. The RISC Processor Design

There are seven pipeline stages for ALU type operations. The stages include the Fetch, Decoder, Register and Forwarding, Execution and Write-back. Fetch and Write-back are divided into two stages. The pipeline registers in between the stages are named as follows; fetch pipeline (FP), fetch-decode, decode-register-file, register-file-execution and execution-writeback. Depending on the instructions, the stages change due to stalls and flushes. The pipeline registers store data and control signals while the instruction progresses through the pipeline.

The processor sends the address to memory via the 32-bit *address* bus for all memory accesses. It receives data via the 32-bit *data_in* bus. This data could be either an instruction or data which is requested by load instructions. For store instructions, the data is sent to the memory via the 32-bit *data_out* bus. There are five interface signals to control communication between the core and the bus control logic. Two input signals specify whether the instruction or data is ready to be received by the core. Three output signals specify if the core is requesting an instruction, performing a load, or performing a store. In Figure 3.2, the I/O of the core is given.

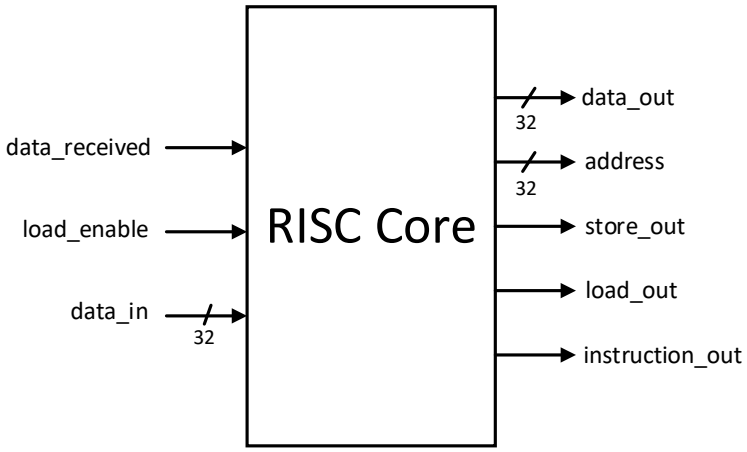

Figure 3.2. I/O of the Core

Although the core can handle the store instructions without a stall in the pipeline due to extended bus *data_out*, it is necessary to stall the pipeline when a load instruction is performed due

to limitations in the data bus. In Figure 3.3, the layout of the core according to the main stages is given. The control is distributed within the pipeline stages. However, the majority of the control unit signals are generated by the Decoder Stage and the Fetch Stage.

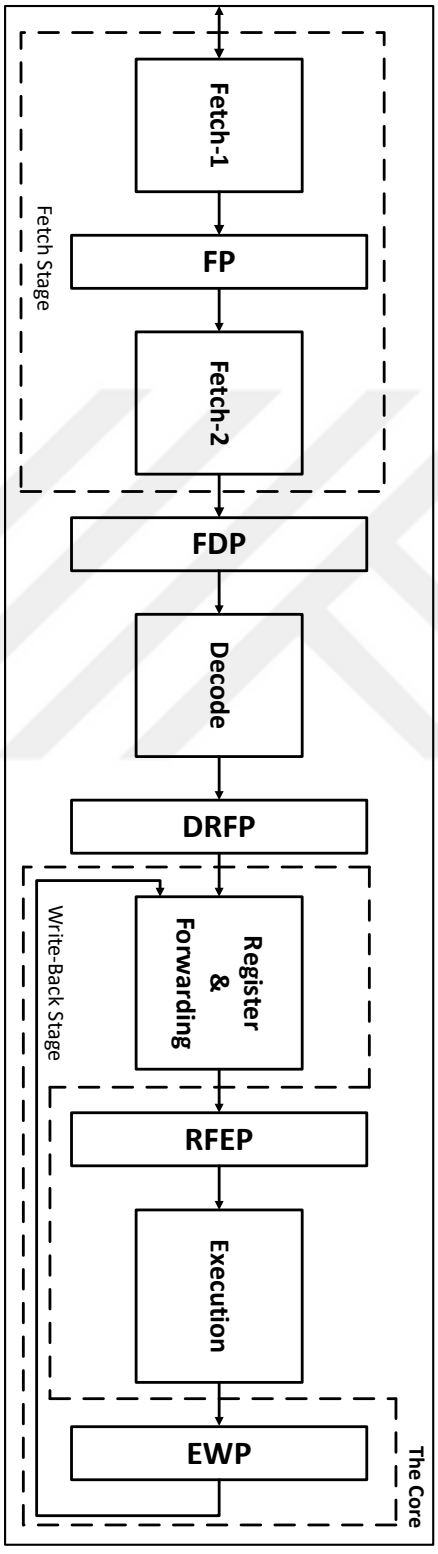

Figure 3.3. The Core Layout Based on Main Stages

3.1.1. Overview of The Core

In this section, the execution of the core based on instruction types is explained. Specifically, ALU, store, load, jump and branch types are discussed with their stall and flush cycles. Also, the execution of core under special cases such as initialization, undefined instructions and no data received are included.

Initialization

The core starts fetching from 0x00000000. The core fetches four instructions to fill the pipeline. While 4 instructions are getting fetched, the pipeline is flushed, and pipeline states are ignored while flushing. The pipeline resumes after the fourth fetch. No write-back is made. Has four flush stages.

ALU Type Execution

An ALU operation takes 7 cycles to complete. Write-back is made. In Figure 3.4, the ALU operation states are given.

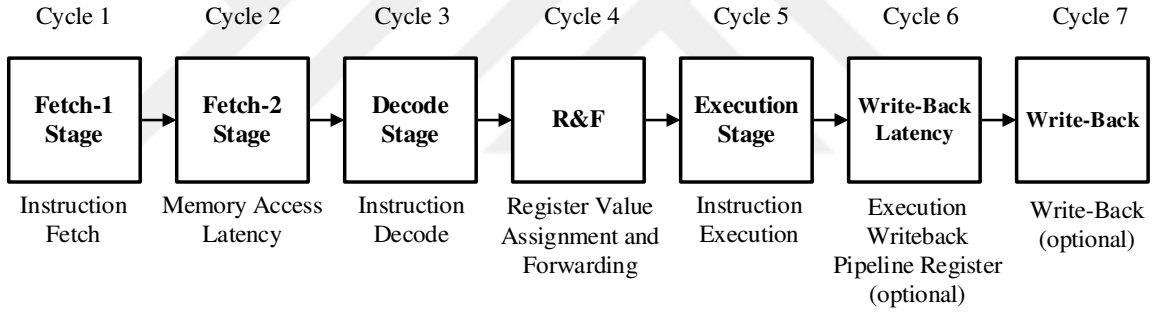

Figure 3.4. States of ALU Operations

Load Type Execution

In Figure 3.5, the load operation states are given.

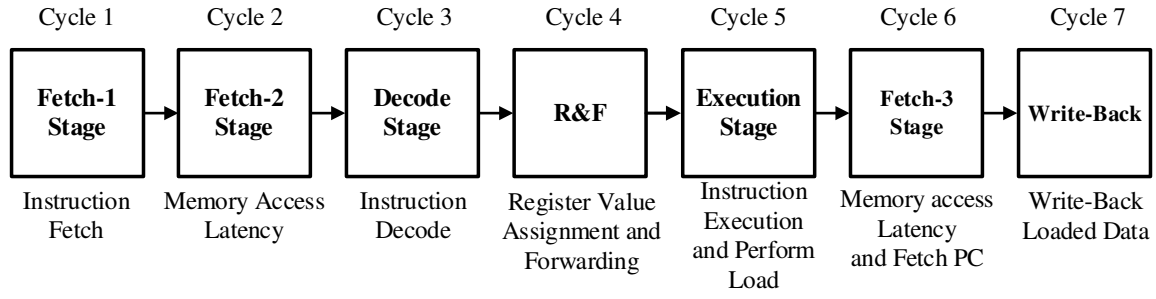

Figure 3.5. States of Load Operations

The calculated address of the load is sent at the fifth cycle, and the pipeline is stalled (pipeline state is ignored at stall). At the sixth cycle, the same PC value is fetched. At the third cycle, the core receives the load data and pipeline resumes. Write-back is made. Takes 7 cycles to complete.

Store Type Execution

The address of the data and the data is sent at the fifth cycle and next instruction is prefetched. No write-back is made. Takes 5 cycles to complete. In Figure 3.6, the store operation states are given.

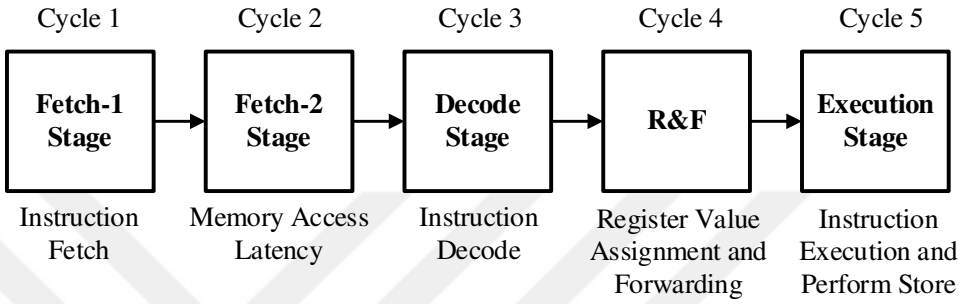

Figure 3.6. States of Store Operations

Jump Type Execution

At the fifth cycle, the PC address is changed with the jump address, and the pipeline is flushed (pipeline state is ignored while flushing) while fetching 4 instructions to re-fill the pipeline. Write-back is concluded at the seventh cycle. Takes 9 cycles to complete. In Figure 3.7, the jump operation states are given.

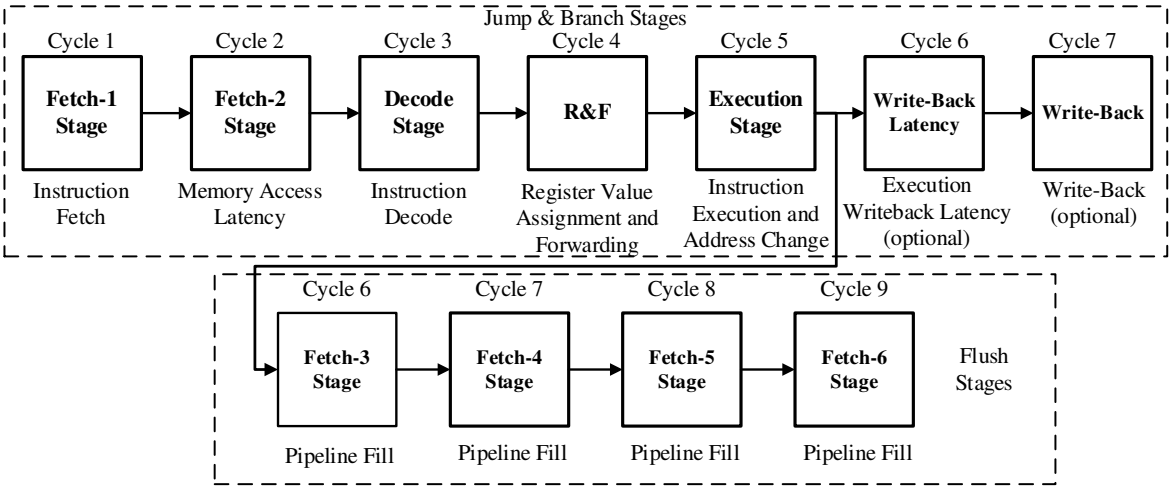

Figure 3.7. States of Jump and Branch True Operations

Branch Type Execution

If the branch condition returns false, the next instruction is fetched and takes 5 cycles to complete. However, If the branch condition returns true, the PC address is changed with the branch

address, and the pipeline is flushed at the fifth cycle. Flush is made while fetching 4 instructions to re-fill the pipeline. No write-back is made. Takes 5 or 9 cycles. If it branches, the stages are no more different than jump operation states (Figure 3.7). In Figure 3.8, failed branch states are given.

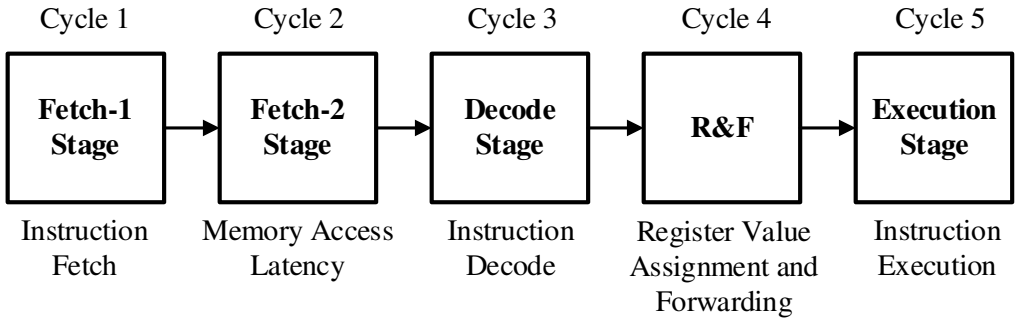

Figure 3.8. States of Failed Branch Operation

Undefined Instructions

Considered as ALU operations. No write-back is made. Takes 5 cycles. The instruction is ignored.

No Data Received

If the core doesn't receive data, the pipeline gets stalled until the data is received. Stalls are made by freezing the pipeline.

Conclusion

To sum up, the processor is designed as a 7-stage pipelined RV32I single core. However, while indicating the stages, ALU operations are taken as a reference. The other operations may take more or less than seven stages to complete. The store instructions have six stages, the load instructions have seven stages, the branch instructions can have six stages, or nine stages, and the jump instructions have nine stages.

3.1.2. Fetch Stages

The Fetch stage includes mainly two parts. The first part is where the instruction is fetched, and the same address is forwarded to the Fetch Pipeline (FP). The second part is where the instruction is delivered from the memory to the Decode stage pipeline with the corresponding address of that instruction from FP. Within the stage, the instruction and data transfer for the core is handled. But also, the flush and stall control of the pipeline is managed. This results in several extra states within the Fetch stage other than instruction type execution variants: initialization, flushes and stalls. The fetch changes its state according to pipeline control signals and the signals received from outside of the core. However, under some conditions, pipeline control signals are ignored and operations are

performed according to signals received outside of the core. In Figure 3.9, the block diagram of the fetch control unit is given. In Figure 3.10, layout of the fetch control unit is given.

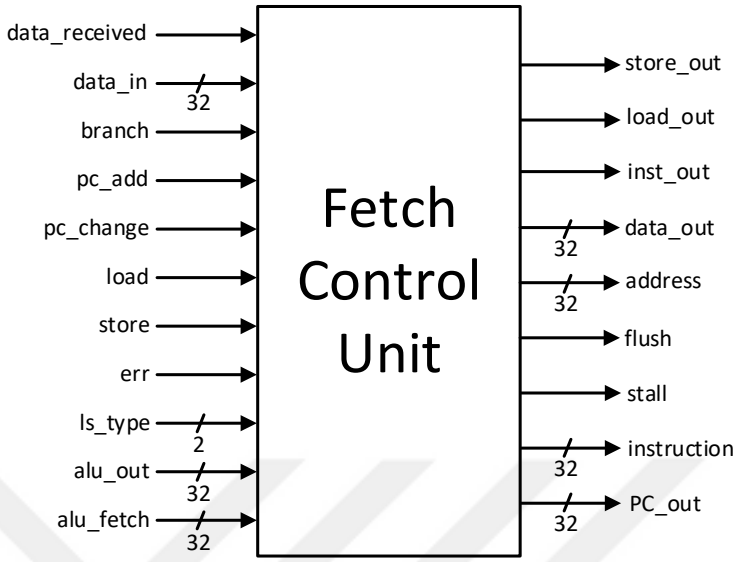

Figure 3.9. Block Diagram of the Fetch Control Unit

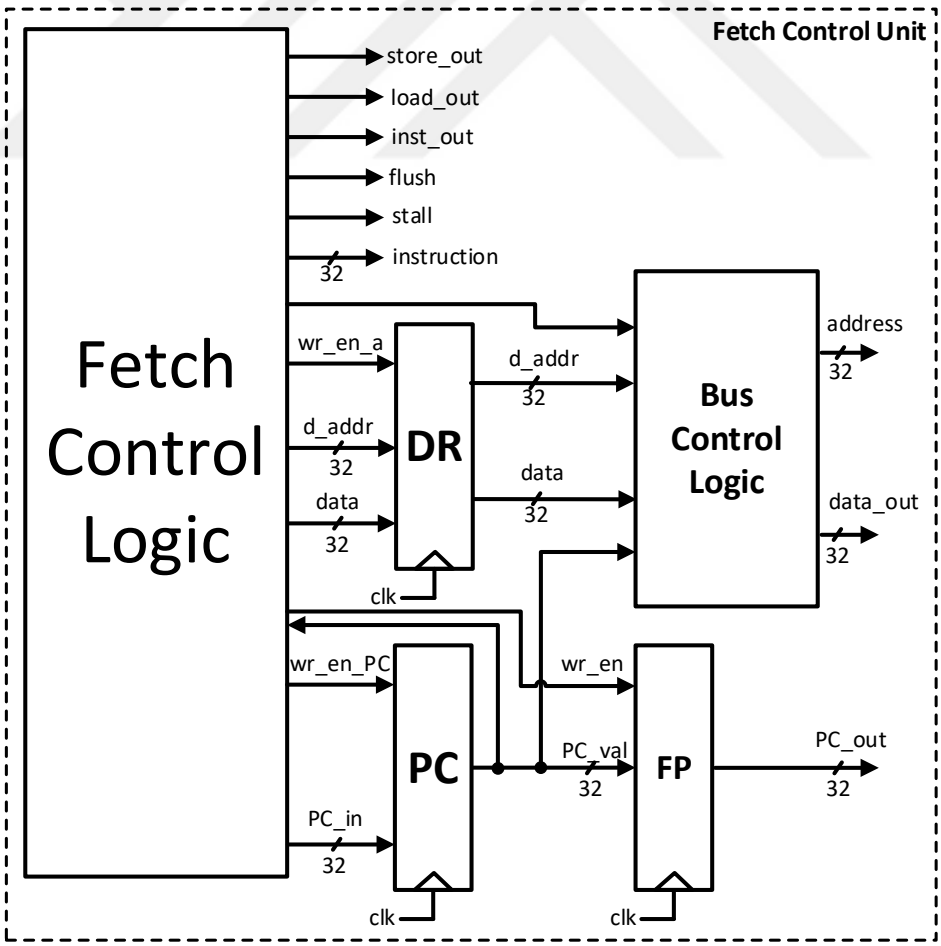

Figure 3.10. The Layout of The Fetch Control Unit

The inputs taken from the outside of the core include *data_received* signal and *data_in* bus. The outputs to outside of the core from the fetch control unit (FCU) include *store_out*, *load_out* and *inst_out* signals, *data*, and *address* buses.

The inputs from the pipeline to the FCU include *branch*, *pc_add*, *pc_change*, *load*, *store*, *err*, *ls_type* signals, *alu_out* and *alu_fetch* values. The outputs to the pipeline include *flush*, *stall* signals, *ins_out* and *pc_out* signals.

As can be observed in the figure, there is a pipeline register within the FUC. Also, within the figure, the Data Register (DR) is used to forward data addresses of load and store, as well as data of the store. It is a temporary storage register.

PC register and FP Register

The special PC register is located within the FCU. The <1:0> bits of the PC are hardwired zeros. This naturally aligns the instruction address and prevents miscalculations in address calculation of branch, jump and AUIPC instructions.

The Fetch Pipeline Register (FP) register stores the older value of the PC register. This register is used for the memory access latency. Since the instruction address sent by the fetch will not be received on the next cycle, the content of the PC is shifted to the FP register for the second cycle, where the data will presumably be delivered. If the data is not sent, the core keeps up with the change and stalls the pipeline. So, extra measures are not required other than regular memory latency. In Figure 3.11, a simplified version of the FP register is shown. Although the *wr_en* behavior is the same as given, the *load_received* signal is not used in the FCU. So, the approximate signal is received from the Fetch Control Logic.

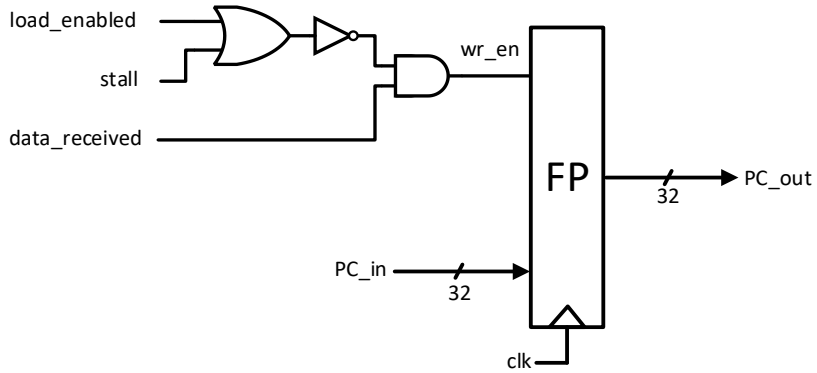

Figure 3.11. Block Diagram of FP Register

Control Signals

The FCU informs the outside of the core with *store_out*, *load_out* and *inst_out* signals. *store_out* signal indicates that the FCU is performing a store. *load_out* signal indicates that the FCU expects a load from the memory. *inst_out* signal indicates that the FCU is expecting an instruction.

The *store_out* signal also indicates that the core is expecting a pre-fetch from the memory. Pre-fetch is made within the Bus Control Logic.

The FU uses *branch*, *pc_add*, *load*, *store* and *error* signals to monitor the state of the pipeline. All the signals mentioned are received from the register_file_execution pipeline, with the exception of the branch signal. It is received from the ALU. The FCU uses *flush* and *stall* signals to force the pipeline to flush instruction(s) or to stall for incoming data. The *stall* signal freezes the pipeline when set. The *flush* signal stops the execution_writeback pipeline register. *ls_type* signal is overwritten to <1:0> bits of the store address. So, 2 bits of the address is used to indicate the type of store to inform the memory. In Figure 3.12, the variations of store data and their address layout are shown.

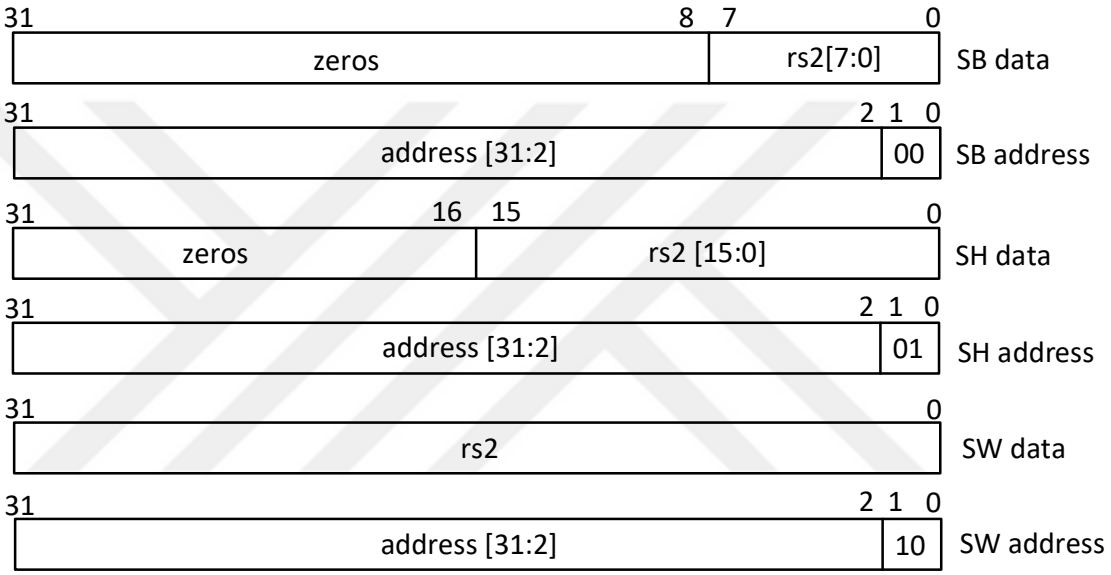

Figure 3.12. Variations of Store Data and Their Address Layout

FCU Data Signals

Data signals are used as follows:

- *data_in* bus is used to take instructions from memory. Although this bus is also used to load data from peripherals, the pipeline doesn't take the load data as instruction.
- *data_out* bus is used to send store data to peripherals.
- *address* bus is used to indicate the addresses of an instruction or a load or a store to peripherals.
- *alu_out* value used to receive address of the branch, store and load instructions from ALU.
- *alu_fetch* is used to receive the address of jump instructions or data to be stored from store instructions. Forwarded from the ALU.
- *ins_forward* output is used to forward instructions to fetch_decoder pipeline register.
- *pc_forward* signal is used to send the FP register data for the incoming instructions to the fetch_decoder pipeline register.

The bus control logic (Figure 3.10) within the FCU is made from muxes, and it chooses the *address* and *data_out* busses output values according to fetch control logic. The fetch control logic adjusts the control signals according to input signals. Its main states are signal check, flush1, flush2, flush3, flush4, load1 and load2. Within signal control state, the FCU checks the pipeline signals and adjusts the pipeline while performing data transfer operations. Within the other states, the pipeline is ignored.

FCU Signal Check

The FCU responds according to the control signals received from the pipeline stages and the peripherals. The control signals from the pipeline indicate how to react, and *data_recieved* signal determines when to react. The *data_recieved* signal must be set for any state to progress to the next phase. If it is not set, the pipeline freezes, with the exception being the start of the initialization. In Table 3.1, sent and received control signals for instruction types are given.

Table 3.1. Received and Sent Control Signals for Instruction Types

Types	**Input Signals**				**Output Signals**		
	load	**store**	**pc_add**	**pc_change**	**store_out**	**load_out**	**inst_out**
Initial	0	0	0	0	0	0	1
Jump	0	0	0	1	0	0	1
ALU	0	0	1	0	0	0	1
Branch	0	0	1	1	0	0	1
Store	0	1	1	0	1	0	0
Load	1	0	1	0	0	1	0

load_out signal can't be set for more than one cycle (exception being non-set data signal, in this condition outputs remain same) due to an instruction that is required to be executed for the next load. However, *store_out* can remain set if stores are executed repeatedly.

FCU States

In Table 3.2, the names of the FCU states and corresponding signals are given. Within the table, the signal check state is expanded for further information. The division between signal check and other states is made by 1-bit *r_state* register content. If it is set, the FCU will ignore the pipeline signals. The table is divided into three categories: *input signals*, *internal state* and *output signals*. Input signals indicate the signals received. Output signals indicate what the output will be after the state concludes. The internal state covers the current state, registers content, next state and next registers content. The current statement indicates that the content starts the data, and the next statement indicates what the contents will be when the state ends.

Table 3.2. Details of FCU States

Name	Input Signals							Internal					Output					
	data received	load	store	pc add	pc chang	err	branch	Current State	r_state	Next PC	Next State	Next r_state	address	inst out	store out	load out	stall	flush
Initial	0	0					x	Signal Check	0	zeros	flush1	1	zeros	1	0	0	0	1
Idle		x					x			PC	Signal Check	0	untouched				untouched	
Undefined	1	x	x	x	x	1	x			PC+4	Signal Check	0	PC+4	1	0	0	0	0
Jump		0	0	0	1	0	x			alu_fetc	flush1	1	alu_fetch	1	0	0	0	1
ALU		0	0	1	0	0	x			PC+4	Signal Check	0	PC+4	1	0	0	0	0
Branch		0	0	1	1	0	0			PC+4	Signal Check	0	PC+4	1	0	0	0	0
							1			alu_out	flush1	1	alu_out	1	0	0	0	1
Store		0	1	1	0	0	x			PC+4	Signal Check	0	alu_fetch	0	1	0	0	0
Load		1	0	1	0	0	x			PC	load1	1	alu_out	0	0	1	1	0
Idle	0	X						untouched					untouched				untouched	
Flush1	1							flush1	1	PC+4	flush2	1	PC+4	1	0	0	0	1
Flush2								flush2		PC+4	flush3	1	PC+4	1	0	0	0	1
Flush3								flush3		PC+4	flush4	1	PC+4	1	0	0	0	1
Flush4								flush4		PC+4	Signal Check	0	PC+4	1	0	0	0	0
Load1								load1		PC	load2	1	PC	1	0	0	1	0
Load2								load2		PC+4	Signal Check	0	PC+4	1	0	0	0	0

- *Initialization*: Initialization of the pipeline is made. The FU starts fetching from the 0x00000000 address. The *inst_out* is set, and the PC is cleared. The *state* register is set. Initialization leads to the Flush1 state.
- *ALU operation*: PC is incremented, and *inst_out* is set.
- *Store operation:* PC is incremented, but address bus is changed to *alu_out<31:2> ls_type<1:0>*. With the extra bus (*data_out*), it is possible to store data in memory and receive instructions from the memory at the same time. So, *data_out* is set to the *alu_fetch* value which holds stored data. The next instruction is pre-fetched by the bus control logic. This strategy is used to prevent delays and improve CPI. *store_out* is set.
- *Load operation:* The core requests load from the bus control logic. The load data can be received from the memory or the peripherals. PC remains the same to reacquire the last instruction. After the load address (received from *alu_out*) is sent, because of the stall, the last instruction received by the bus control logic is terminated. *load_out* is set. The pipeline is stalled. the *r_state* register is set. Load leads to *Load1* state.
- *Branch operation: branch* signal is checked. If the branch fails, PC is incremented. If branch doesn't fail; *flush* is set, PC is overwritten with *alu_out* and sent to request instruction, r*_state* register is set, and this condition leads to *Flush1* state. *inst_out* is set for both conditions.
- *Jump operation:* The *flush* is set. PC is overwritten with *alu_fetch* and sent to request instruction. The *inst_out* is set. Jump leads to *Flush1* state.
- *Undefined: err* signal indicates that the instruction is undefined. PC is incremented. Due to the result being already in the execution_writeback pipeline register, the FU can't undo the result (although there will be no result which is prevented by decoder) the pipeline register doesn't take input if it reads *err* signal.
- *Flush states:* PC is incremented. The *inst_out* is set. The *flush* is set. The r*_state* is set. Leads to next flush state until the *Flush4* state.
- *Flush4*: PC is incremented. The *inst_out* is set. The *flush* is cleared. The r*_state* is cleared. Leads back to *signal check*. When it is led to the signal check, the pipeline will be filled.
- *Load1*: PC is recent. *inst_out* is set. The *stall* is set. The r*_state* is set. Leads to *Laod2* state.
- *Load2*: PC is incremented. The *inst_out* is set, the *stall* is cleared. the r*_state* is cleared. Leads back to *signal check*. Load is received in this state and the instruction will be received in the next state.
- *Idle*: Whether the *r_state* is set or not if *data_received* is not set the fetch control logic preserves its state (except in initialization). Notting changes.

3.1.3. Decode Stage

In the Decode stage, operands, immediate value and control signals are extracted. In Figure 3.13, the Instruction Decoder Unit ports are given.

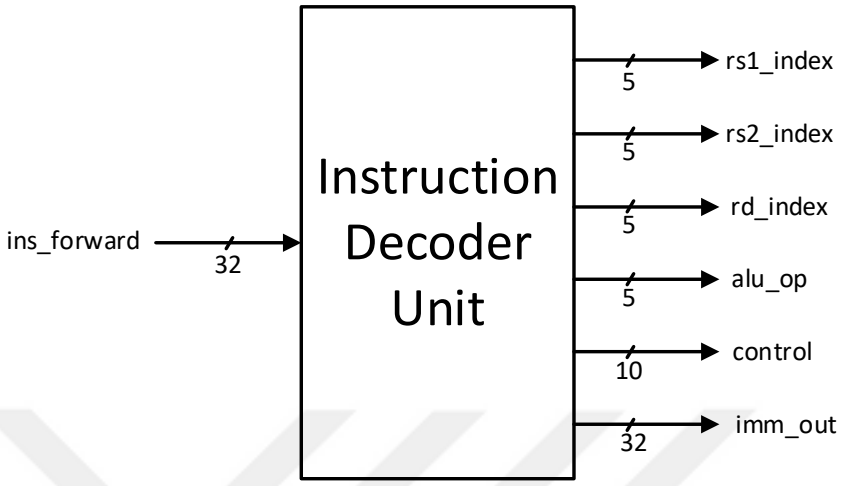

Figure 3.13. Ports of the Instruction Decoder Unit

Control Signals

The control signals are separated into two types. ALU operation control signals and pipeline control signals. The *alu_op* signal controls the ALU operation. The operation of the execution unit is determined by using the opcode, funct3 and funct7 fields of the instructions. Some instructions have multiple versions which share the same opcode, to decode them funct3 and funct7 fields are used. The specifications of the *alu_op* signal will be given under the execution stage. In Table 3.3, the main control signals for all instructions are given. The *control* signals generated by opcode, funct3 and funct7 fields.

Table 3.3. 10-bit Control Signal Generation Per Instruction

Instructions	err	imm en	pc add	pc change	rd write	load	load sign	store	load store type
Immediate Instructions	0	1	1	0	1	0	x	0	xx
Register-Register Instructions	0	0	1	0	1	0	x	0	xx
Branch Instructions	0	1	1	1	0	0	x	0	xx
JAL, JALR	0	1	0	1	1	0	x	0	xx
LB	0	1	1	0	0	1	1	0	00
LH	0	1	1	0	0	1	1	0	01
LW	0	1	1	0	0	1	x	0	10
LBU	0	1	1	0	0	1	0	0	00
LHU	0	1	1	0	0	1	0	0	01
SB	0	1	1	0	0	0	x	1	00
BH	0	1	1	0	0	0	x	1	01
SW	0	1	1	0	0	0	x	1	10
Undefined Instructons	1	0	0	0	0	0	0	0	00

err signals that current instruction is undefined.

imm_en corresponds to the immediate enable signal. If the instruction is a register-register instruction, this bit is cleared, otherwise it is set.

pc_add is set for all instructions except for jump instructions to increment PC by four.

pc_change is set for jump and branch instructions.

rd_write indicates that the instruction requires write-back operation.

load is set for load instructions.

load_sign is set when the load is zero extended and reset when the load is sign extended.

store is set for store instructions.

ls_type is used to determine the length of the data (byte, half-word, word) to be transferred or received.

Immediate Value Generation

There are five different immediate classes to decode: U-immediate, I-immediate, B-immediate, S-immediate and J-immediate. Figure 3.14 presents immediate value formats explained in the following list.

- U-immediate type instructions are LUI and AUIPC. U-immediate is generated by putting the top part <31:12> of the instruction to the top <31:12> of the immediate and putting 0's to the lower <11:0> portion of the immediate.
- I-immediate type instructions are ADDI, SLTI, SLTIU, XORI, ORI, ANDI, JALR, LB, LH, LW, LBU and LHU. I-immediate is generated by using <31:20> bits of the instruction and sign extending to 32-bit.
- B-immediate type instructions are BEQ, BNE, BLT, BGE, BLTU, and BGEU. B-immediate is generated by placing the instruction bits in the following order <31|7|30:25|11:8>. Then, LSB is cleared, and the immediate is sign-extended to 32 bits.
- S-immediate type instructions are SB, SH and SW. S-immediate value is generated by placing the instruction bits in the following order <31:25|11:7> and then sign extending it to 32 bits.
- J-immediate type instruction is JAL. J-immediate is generated by placing the instruction bits in the following order <31|19:12|20|30:21> and then putting a 0 to LSB and sign extending it to 32 bits.
- Instructions which do not fit in the standard immediate formats are the shift instructions SLLI, SRLI and SRAI. Although these can be considered as a special version of I-immediate, funct7 field is also tested for instruction type. So, the immediate for the three instructions is derived by using five bits of shamt field (<4:0>), zero extended to 32 bits. Figure 3.15 illustrates the generation of this immediate value.

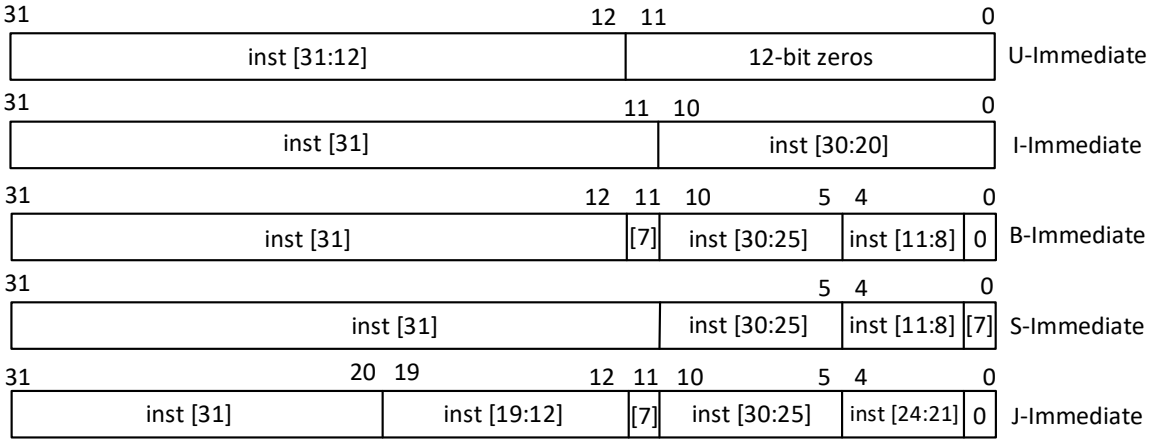

Figure 3.14. Format Generations for Immediates

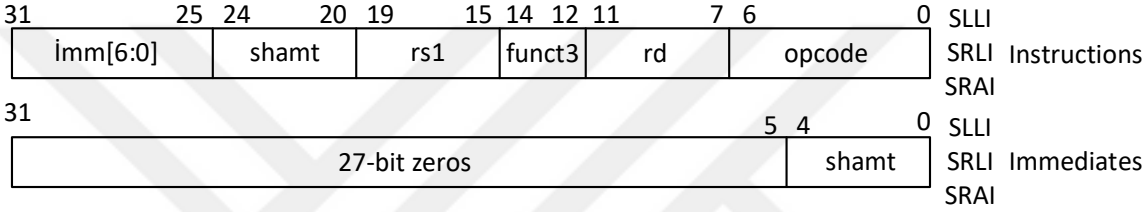

Figure 3.15. Immediate Format SLLI, SRLI and SRAI Instructions

The rest of the instructions that do not use immediate format include ADD, SUB, SLL, SLT, SLTU, XOR, SRL, SRA, OR and AND instructions. FENCE and SYSTEM instructions are not implemented in the design. The core can execute 37 unique instructions. An error is generated if a non-instruction word is fetched.

Register Index Generation

The index of each register is five bits. There are two source register indexes denoted as rs1 and rs2 and one destination register index denoted as rd. Thus, those 10 bits for the rs indexes are directly sent to the Register File Unit in case they are required by the instruction. This strategy saves time. If those registers are not used by the instruction, they are not used by the execution unit. The rd index is forwarded till the write-back stage.

3.1.4. Register and Forwarding Stage

The Register and Forwarding (R&F) stage contains Register File Unit (RFU), Forwarding Logic Unit, Load Separator logic and Register Mux Logic. In R&F stage rs1, rs2 and imm_rs2 values are forwarded to the third pipeline register. If a change happens to an operand (rs1 and rs2), the forwarding logic unit detects and forwards the changed version of the value to the third pipeline register. Also, the Write-Back stage concludes within this unit with the overwrite of the

corresponding register data received from the execution_writeback pipeline register or from the memory. In Figure 3.16, the block diagram of R&F stage units and logic is given.

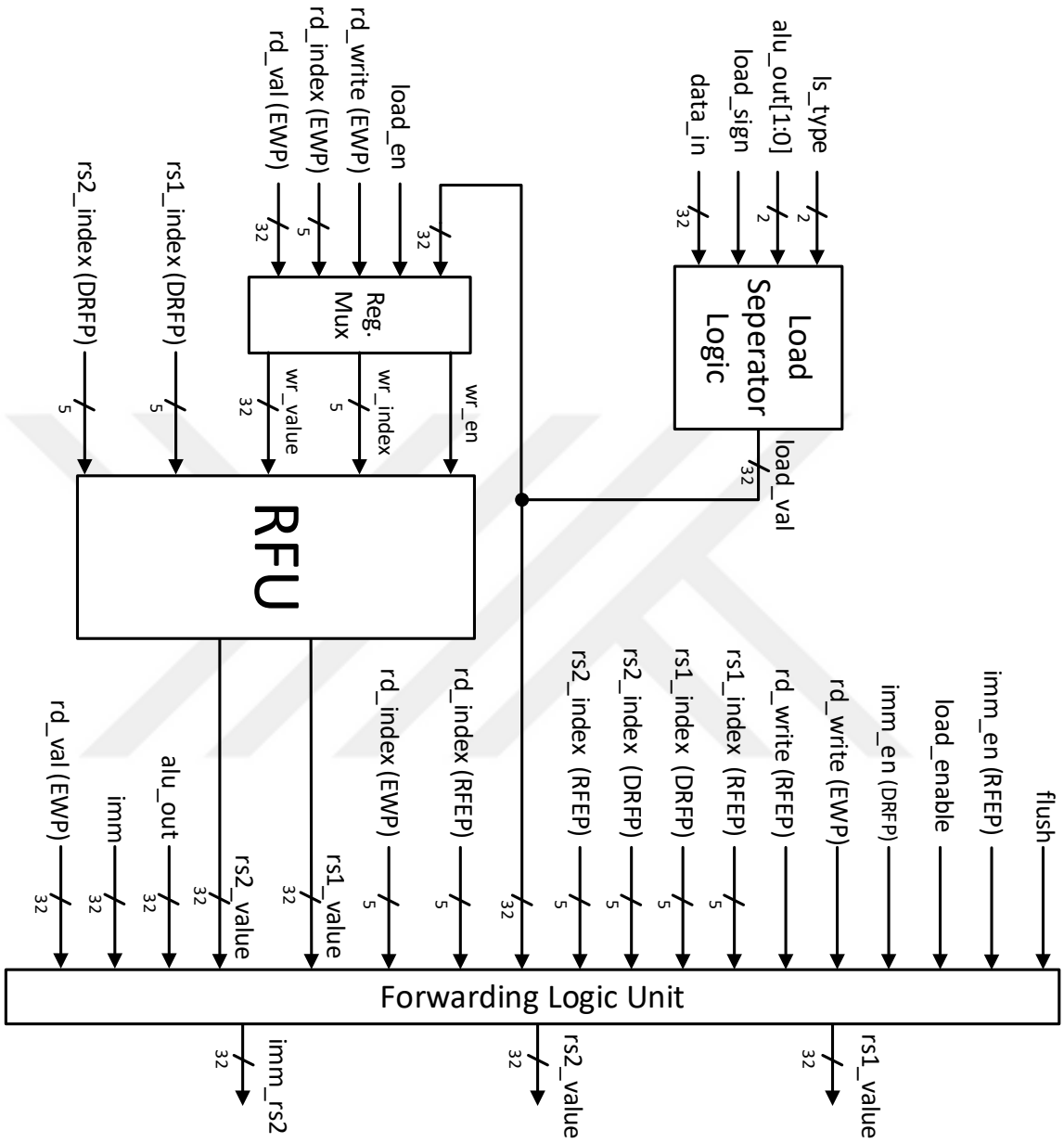

Figure 3.16. Block Diagram of the R&F Stage Units and Logics

Register File Unit

The RFU contains thirty-two general-purpose registers. Each register is 32 bits. The unit has two read ports, and one write port where each port receives a 5-bit index value. This unit stores the destination register value and load value by concluding Write-Back stage while providing the source register values.

RFU works as follows. The source register indexes, sent by the instruction decode unit are used to select source register value outputs. The destination register index received from the

execution-writeback pipeline register. the *wr_en*, *wr_index* and *wr_value* is determined by register mux logic. In Figure 3.17, the ports of the RFU are given.

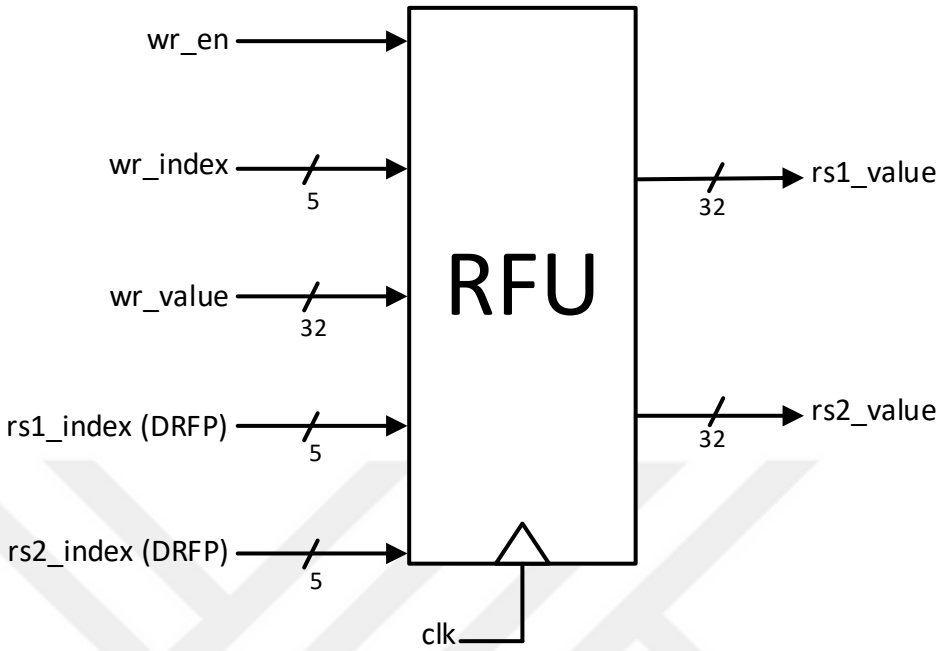

Figure 3.17. Ports of the Register File Unit

Register Mux

In Figure 3.18, the logic diagram of the Register Mux is given.

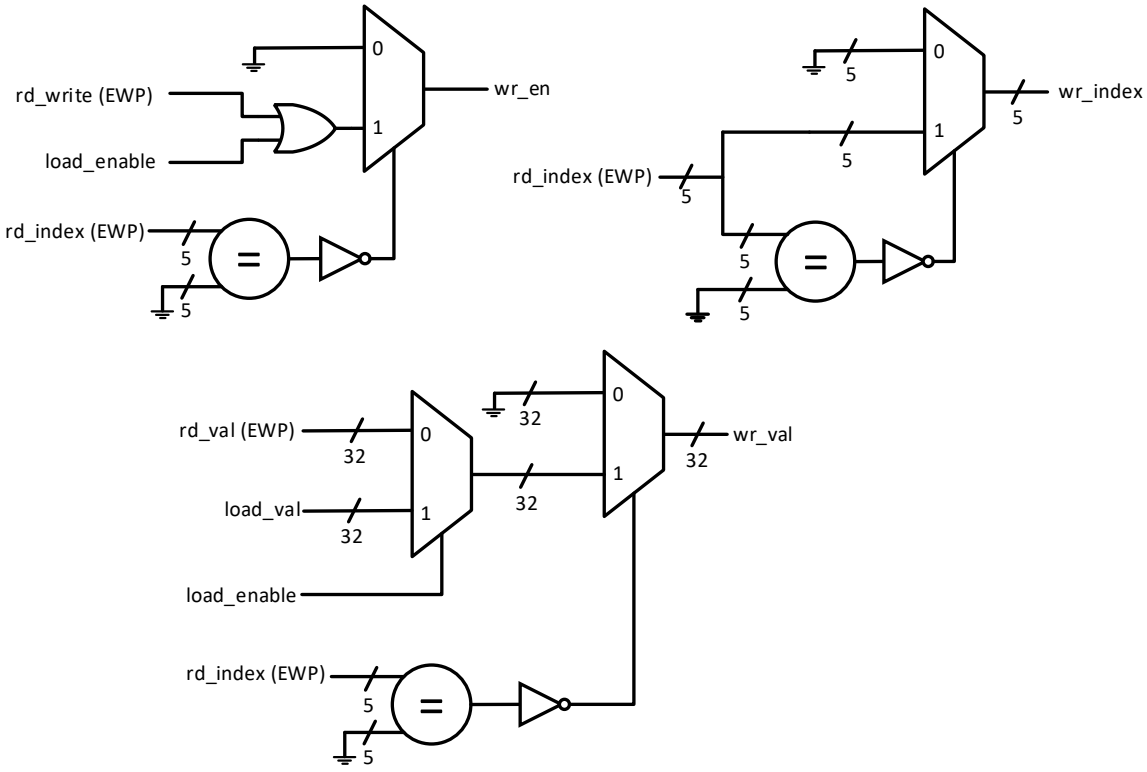

Figure 3.18. Logic Diagram of the Register Mux

To determine if the RFU needs to overwrite a register, what it needs to overwrite with and to where it needs to overwrite, a logic circuit is designed and referred to as Register Mux Logic.

To determine if overwrite is required, the rd index is tested for zero since the zero register is read only. Then, if either *load_ received* or *rd_write* signal is set, the *wr_en* signal for RFU is set.

To determine where to overwrite, only rd is tested for zero. If the test passes, *rd_index* is forwarded to *r_index* which indicates the write index. Otherwise, 5-bit zeros are sent.

To determine which data will be overwritten to RFU, the *rd_index* is tested for zero. After that, the *load_received* signal is checked. In the overwrite phase, the loaded value has priority. So, if the *load_ received* signal is set, the output is *load_val*. Otherwise, the output is *rd_val*.

Due to the architecture of the designed processor, if a load is received, the pipeline can't acquire a new instruction to process. This causes the pipeline stall. Within the stall, the loaded data can be stored in the RFU for the next instructions.

Forwarding Logic Unit

The core is designed to support forwarding operations. Forwarding logic is used to update the rs values of the instructions when necessary. The unit also selects the *imm_rs2* content regarding the *imm_en* signal.

The forwarding is made for three stages of the pipeline: *Execution Stage forwarding* (EF), *Write-Back Latency Stage Forwarding* (WBLF) and *Memory Write-Back Stage (load data) Forwarding* (MWBF). Every forwarding part has different priorities. The priority of the stage forwarding is made according to instruction conclude time. Meaning that *MWBF* has priority over the *EF*, and *EF* has priority over *WBLF*. Forwarding is made according to the rs index and the corresponding rd index of the stage, which the forwarding is made from. If WBLF is made, the rd index and rd value is also received within the execution_writeback pipeline register. If EF is made, the rd index received from the register_file_execution pipeline register and rd value is received from the alu_out signal. If MWBF is made, the rd index received from execution_writeback pipeline register and rd value is received from load separator logic.

Forward activation depends on the signals received from various sources including fetch control logic, register_file_execution pipeline register, ALU and execution_writeback pipeline register. In Figure 3.19, the logic diagram of the forwarding logic is given.

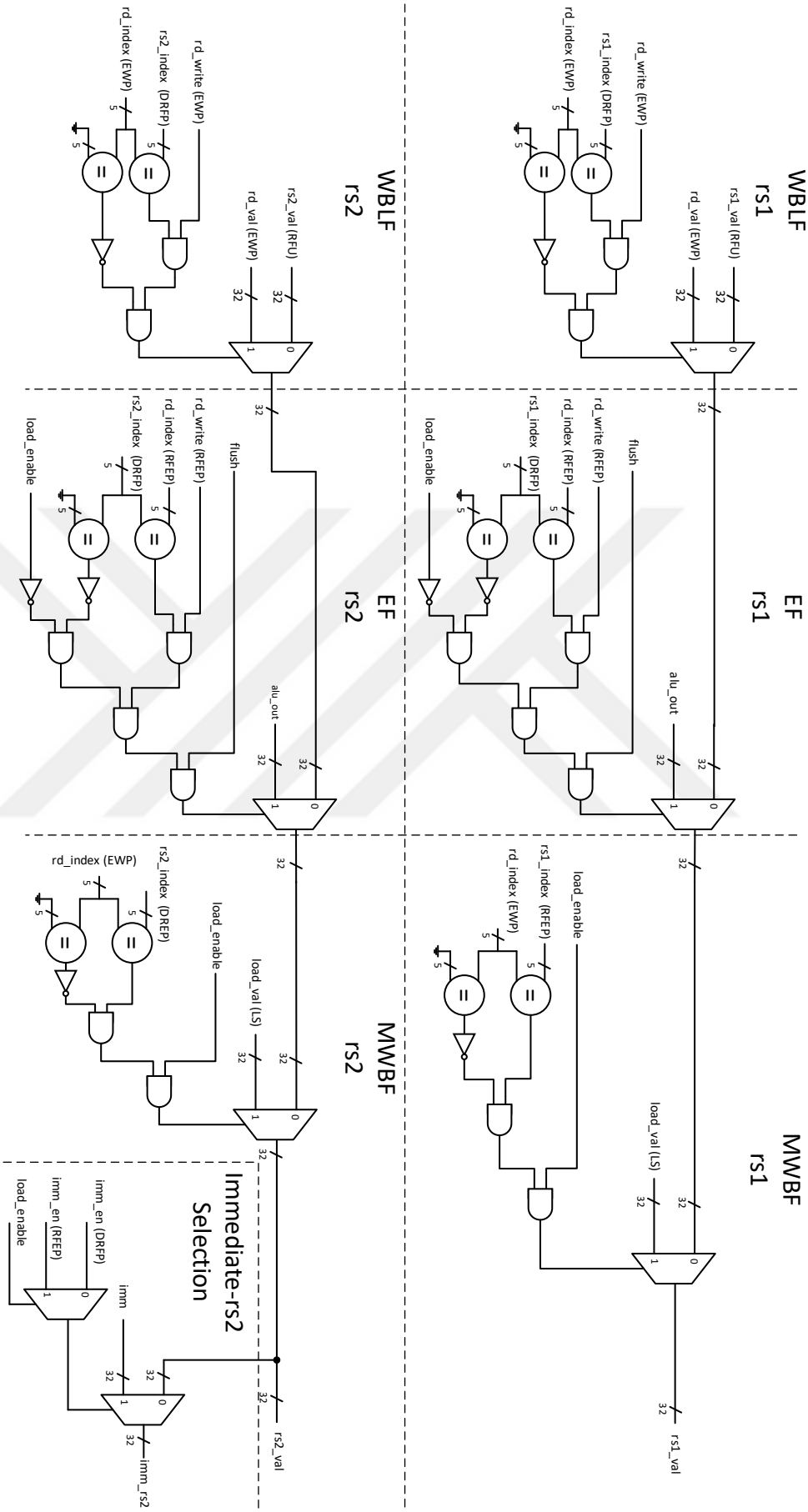

Figure 3.19. Logic Diagram of the Forwarding Logic

Load Separator Logic

The Load Separator Logic consists of multiplexers. The separation is made according to *load_sign*, *ls_type* and *alu_out <1:0>* signals. These signals are received from the EWP register. Word load, half word load and byte load separation are made according to *ls_type* signal. In table 3.4, variations are given.

Table 3.4. Load Type Separation According to ls_type Signal

Name	**Byte Load**	**Half Word Load**	**Word Load**
alu_out <1:0>	00	01	10 or 11

Although, '11' signal from *ls_type* signal will never be achieved, considering this unit designed as concurrent, word type load is connected to that selection as well.

load_sign signal determines if the extension of the loaded data will be the MSB of the separated portion or will be zero extension. This signal does not contribute to word load separation.

If the load type is not word load, *alu_out<1:0>* signal determines the selection of the load, and it is made according to little-endian format. In table 3.5, the variations are given.

Table 3.5. Corresponding Variations of alu_out<1:0> Signal According to Load Type

alu_out <1:0>	**LB/LBU**	**LH/LHU**	**LW**
00	data<7:0>	data<15:0>	data<31:0>
01	data<15:8>		
10	data<23:16>	data<31:16>	
11	data<31:24>		

The output of the load separation unit is forwarded to the register mux unit and forwarding logic unit to be overwritten to corresponding register and to be used in load forwarding. In Figure 3.20, the variations of the Load Separator Unit outputs are given. In Figure 3.21, the logic diagram of the Load Separator Unit is given.

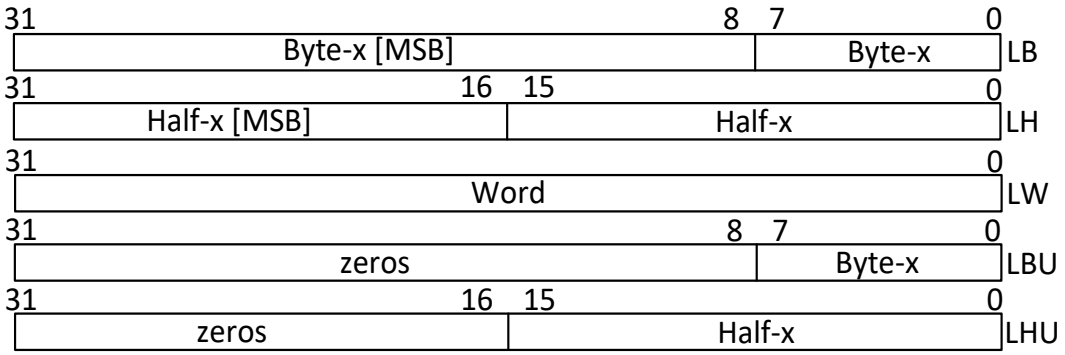

Figure 3.20. Variations of Load Separator Unit Outputs

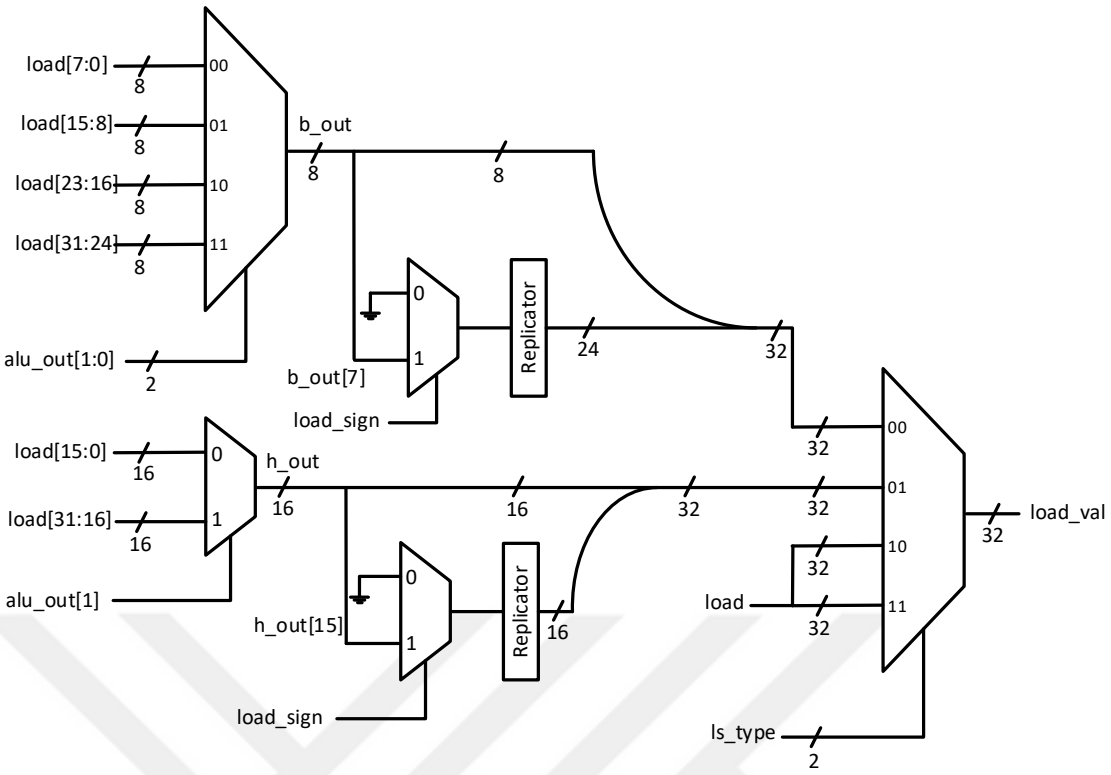

Figure 3.21. Logic Diagram of Load Separator Unit

3.1.5. Execution Stage

In Figure 3.22, the simplified block diagram of Execution Unit is shown.

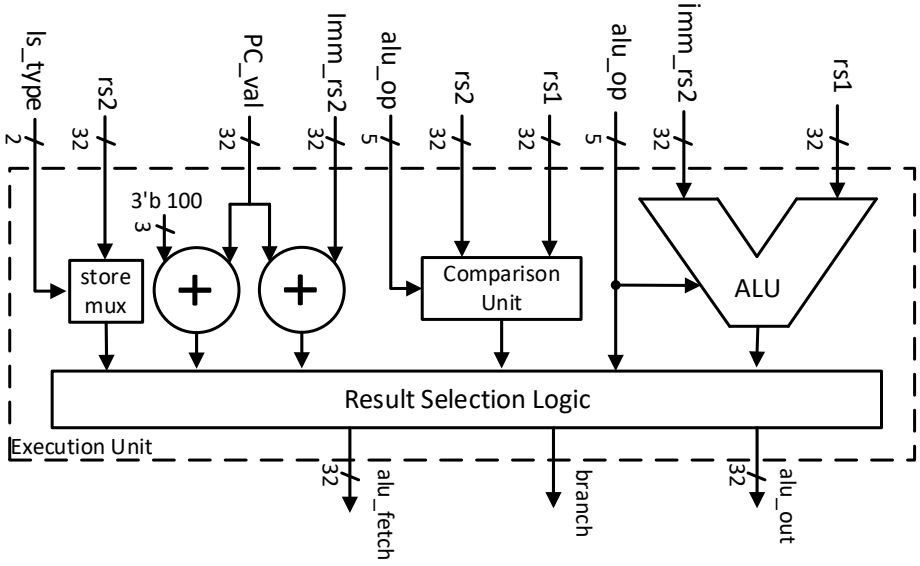

Figure 3.22. The Simplified Block Diagram of The Execution Unit

The Execution stage consists of ALU, a comparison unit, extra addition logics and store mux. The output values are selected according to *alu_op* signal. This selection is made within the result selection logic. The register-register and the register-immediate calculations are made in ALU. Branch comparison is made in comparison unit. The jump destination address and branch and JAL

address calculations are made in extra addition logics. The store mux separates the rs2 value. This separation is made due to memory doesn't accept half word or byte as input, but it takes the whole word. If the memory design would fulfill the requirement, the rs2 value should have been sent to fetch without process to perform the store, in addition to calculated address. The calculations within the Execution Unit are made at the same time as the selection.

Operands and Outputs of the Execution Unit

To reduce resources in the ALU, immediate versions are combined with register-immediate. This is achieved using *imm_rs2* operand. Such that *imm_rs2* is either equal to the immediate or the rs2 value. If the immediate is not used in the instruction, this signal sends rs2 value. The selection of *imm_rs2* value is made in the Forwarding Logic Unit. This approach reduces the delay in conclusion of the *alu_out* output port to reach Fetch Control Unit, which has the longest delay. *rs2_val* is used as the rs2 value only if the immediate and the rs2 value is required at the same time which are the branch instructions. *PC_val* is used in the AUIPC, branch and jump instructions.

The *branch* output value shows whether the branch is taken or not. It is set when the branch comparison is true. The *alu_out* is used to deliver the result of the operation for write-back or to send the result to the fetch stage. When the *alu_out* is used, *alu_fetch* is used to extend the *alu_out* port if another result is required in the fetch stage

Operation Selection of the Execution Unit According to Instruction Type

In Table 3.6, the encodings of the *alu_op* bits for all instructions are given.

Table 3.6. Operations of the Execution Unit According to alu_op Signal

alu op		Instruction
00	000	ADDI
	001	SLLI/SLL
	010	SLTI/SLT
	011	SLTIU/SLTU
	100	XORI/XOR
	101	SRLI/SRL
	110	ORI/OR
	111	ANDI/AND

alu op		Instruction
01	000	BEQ
	001	BNE
	010	ADD
	011	SUB
	100	BLT
	101	BGE
	110	BLTU
	111	BGEU

alu op		Instruction
10	000	LB
	001	LH
	010	LW
	011	-
	100	LBU
	101	LHU
	110	SRAI/SRA
	111	-

alu op		Instruction
11	000	SB
	001	SH
	010	SW
	011	-
	100	LUI
	101	AUIPC
	110	JAL
	111	JALR

As stated, the ALU operation selection is made according to the 5-bit *alu_op* signal, and it encodes 29 different operations. The combined register-register operations are SLL, SLT SLTU, XOR, OR, AND, SRL and SRA. They are combined with SLLI, SLTI SLTIU, XORI, ORI, ANDI, SRLI and SRAI immediate-register operations, respectively. In the given table, instructions in the same category have the same two most significant bits, and the shared instructions are designated by slash symbol. Regarding the undefined *alu_op* signal combinations, although it is impossible, when the received *alu_op* signal value is undefined, all outputs are reset.

Outputs Based on The Instruction Type

Depending on the instruction type, the result calculations are forwarded from different ports. In the following, if the output value is not explained, it is reset.

- *Branch instructions:* When the comparison of registers is made, the *branch* signal is set to the result of the comparison. The address is calculated regardless of the comparison and given as output via alu_out port.
- *Store instructions:* the *rs2* operand is reordered according to the type of the store instruction and forwarded via *alu_fetch* signal. The forwarded value is sent in little-endian format. Address calculation is forwarded via *alu_out* port.
- *Load instructions:* Address calculation is forwarded via *alu_out* port.
- *Jump instructions:* Address calculation is forwarded via *alu_fetch* port. The link value is forwarded via *alu_out* bus.
- *Register-immediate instructions:* The result is forwarded via *alu_out* port.
- *Register-register instructions:* The result is forwarded via *alu_out* port.

3.1.6. The Pipeline Registers

The core is pipelined by separating the main stages with registers that store the processed or forwarded data. The core design consists of five pipeline registers. However, the number of pipeline stages is seven due to extra stages required for fetch and write-back.

The FP register is considered as the initial pipeline register located in between fetch-1 stage and fetc-2 stage. The first pipeline register is located between the Fetch-2 stage and the Decode stage. It is named as *fetch_decode* pipeline register. The second pipeline register is located between the Decoder stage and the R&F stage. It is named as *decode_register_file* pipeline register. The third pipeline register is located between the R&F stage and the Execution stage. It is named as *register_file_execution* pipeline register. The fourth and the last pipeline register is located between Execution stage and Write-Back stage. It is named as the *execution_writeback* pipeline register. The last pipeline register is also connected to the register file unit for the write-back stage.

The write condition for the initial, first and second pipeline registers is as follows: When the stall and *load_received* signals are cleared and the *data_received* signal is set. The write condition of the fourth pipeline register is as follows: When the *stall*, *flush*, *err* and *load_received* signals are cleared and the *data_received* signal is set. The third pipeline register is special due to MWBF. While the conditions for initial, first and second pipeline registers apply, the third pipeline register also works if the operands (rs1, rs2 and imm_rs2) need to change with performed load. If load is performed and the rd index of the execution writeback pipeline register and operand index at the register file execution pipeline register is equal, the loaded value is written to corresponding operand register. Operand value is forwarded by forwarding logic. So, the third pipeline has four different write signals for four different registers. It is a stacked pipeline register.

Fetch pipeline register

Fetch pipeline (FP) register consists of 2 input and 1 output ports. This register is used for instruction memory latency. The received instruction from the memory is forwarded with the PC content of this register to the fetch decode pipeline register. In Figure 3.11, the block diagram of the FP is given.

Fetch Decode Pipeline Register

The Fetch Decode Pipeline (FDP) register consists of 3 input and 2 output ports. It stores the most recently fetched instruction and its PC value, and forwards to decode state. In Figure 3.23, the block diagram of the FDP register is given.

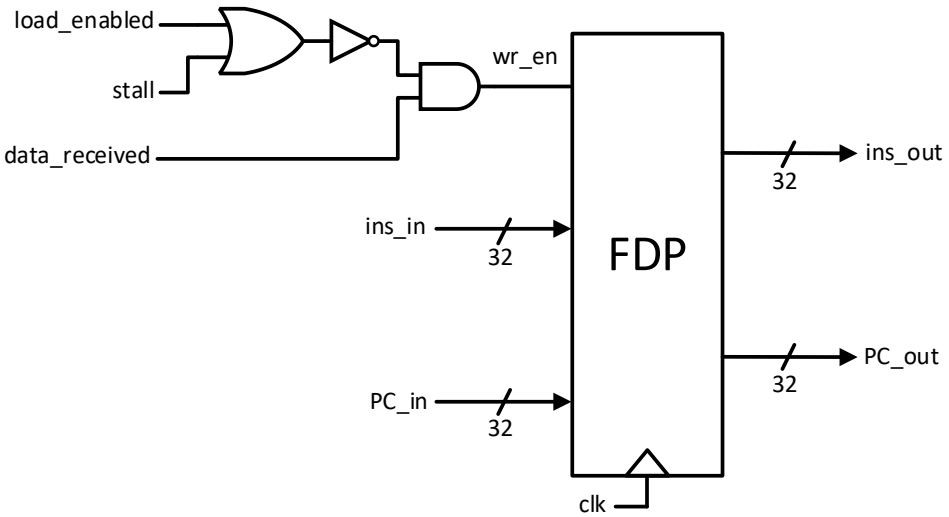

Figure 3.23. The Block Diagram of the FDP Register

Decode Register File Pipeline Register

Decode Register File Pipeline (DRFP) Register gets decoded data from the decode stage and PC from the first pipeline register as input. Then, the pipeline register forwards them to R&F stage and the third pipeline register. In Figure 3.24, the block diagram of the DRFP register is given.

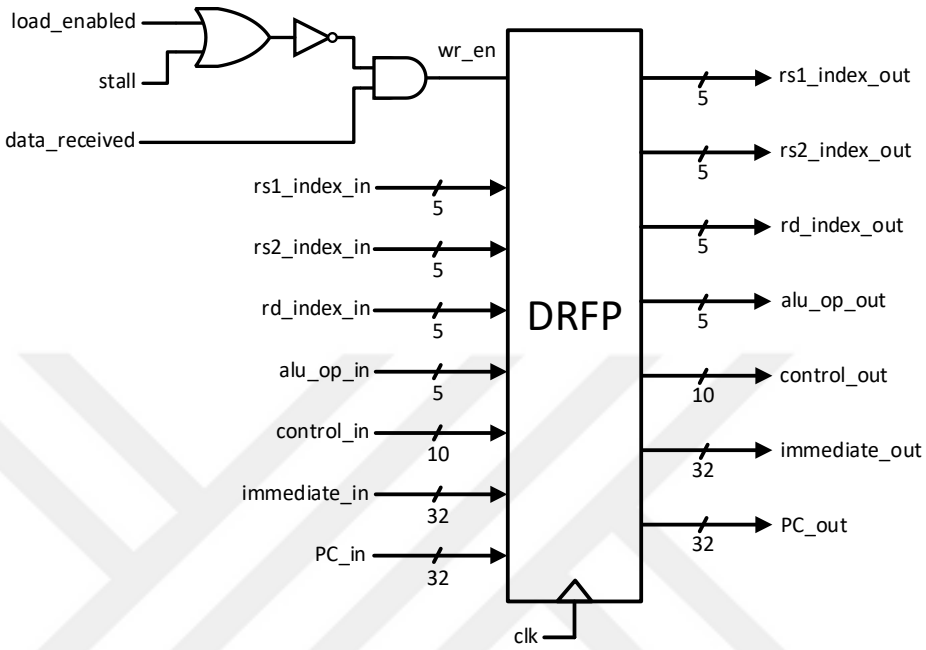

Figure 3.24. The Block Diagram of the DRFP Register

Register File Execution Pipeline Register

In Figure 3.25, the block diagram of the RFEP register is given.

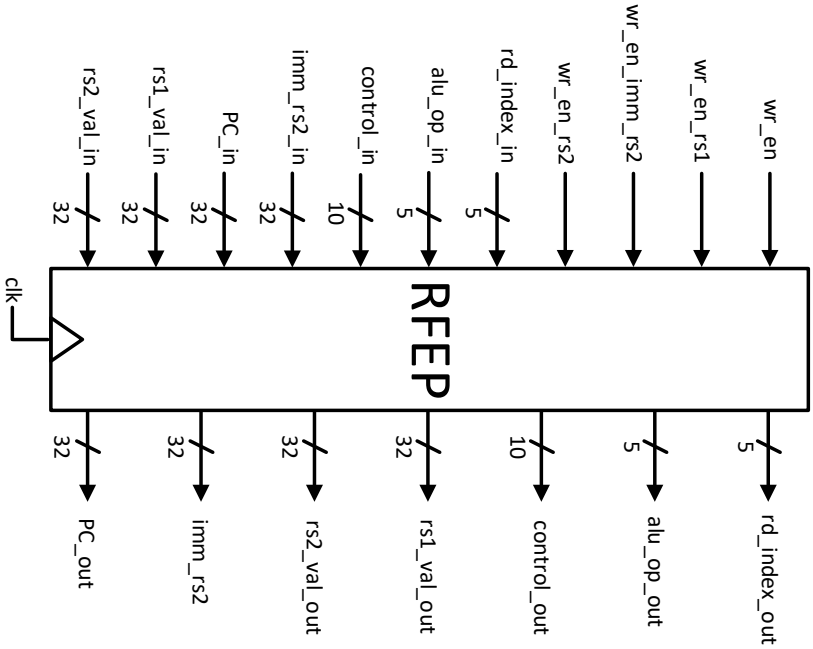

Figure 3.25. The Block Diagram of the RFEP Register

The Register File Execution Pipeline (RFEP) register gets its inputs from the R&F stage and DRFP register. Then delivers the operands to the Execution stage and some of the signals are passed to the Execution Writeback Pipeline (EWP) register. This register provides most of the control signals to the Fetch state. So, the timing of Fetch control logic depends on this pipeline register. This pipeline register has multiple write signals, and it is required to be different due to MWBF. In Figure 3.26, the logic diagrams of the write signals are given.

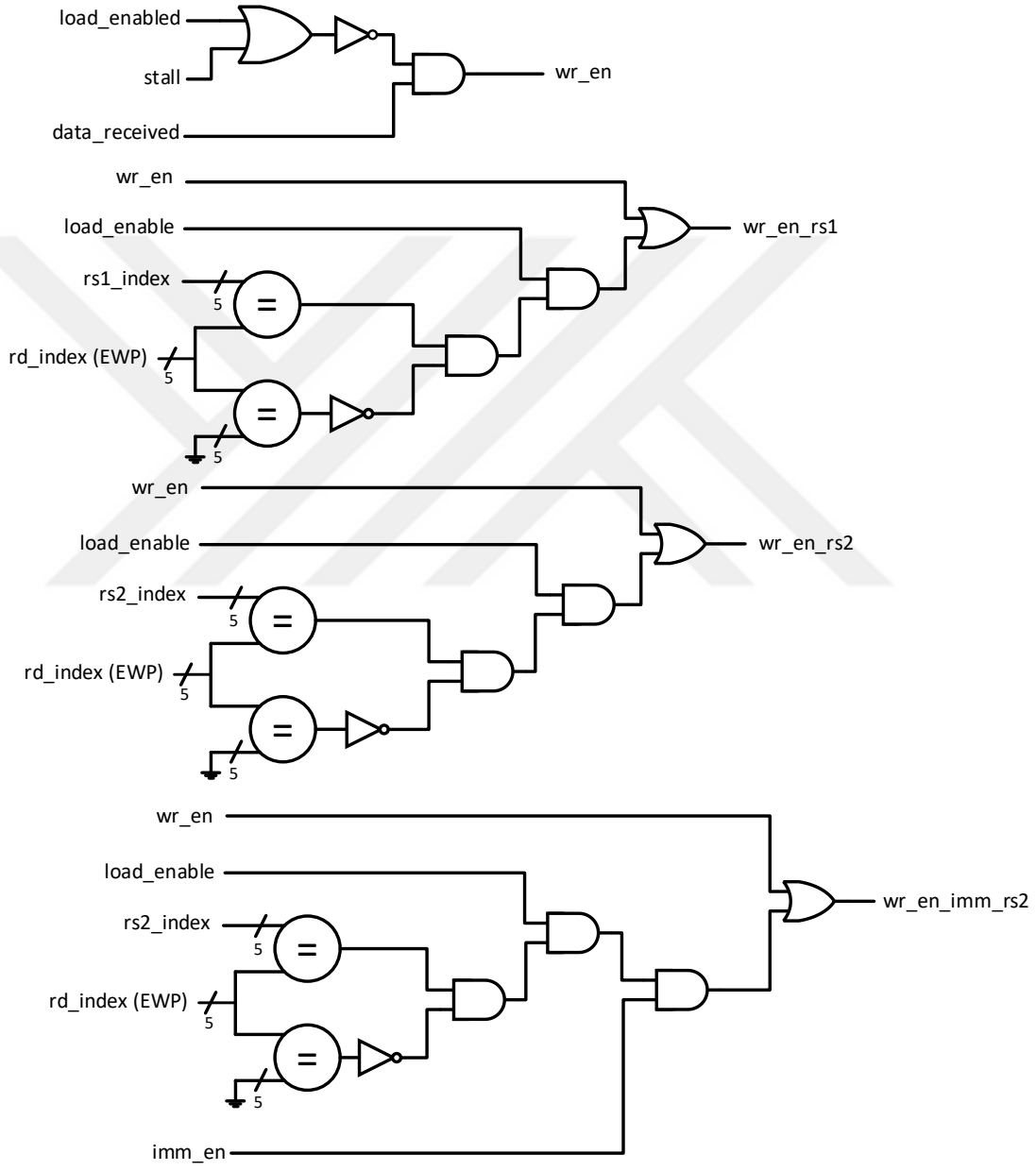

Figure 3.26. the Logic Diagrams of the Write Signals for RFEP

When a load is performed, the data needs to be written to the corresponding operand register and the other registers cannot be changed. The *wr_en_rs1* controls the rs1 operand write signal. The *wr_en_rs2* controls rs2 operand write signal. The *wr_en_imm_rs2* controls the imm_rs2 operand

write signal. The *wr_en* signal controls the rest of the registers. All the operand registers contain the *wr_en* signal to be functional, but addition is made to forward load value.

Execution Writeback Pipeline Register

WP Register is the last pipeline register. It gets input from the Execution stage and the forwarded data from the third pipeline register. This pipeline only takes data if the instruction is valid to process. Meaning that, if data enters this pipeline register, data is executed. So, other than just *data_received* is set and the *load_enable* and stall signals are clear, the condition also includes zero on flush and error signals. In Figure 3.27, the block diagram of the EWP register is given.

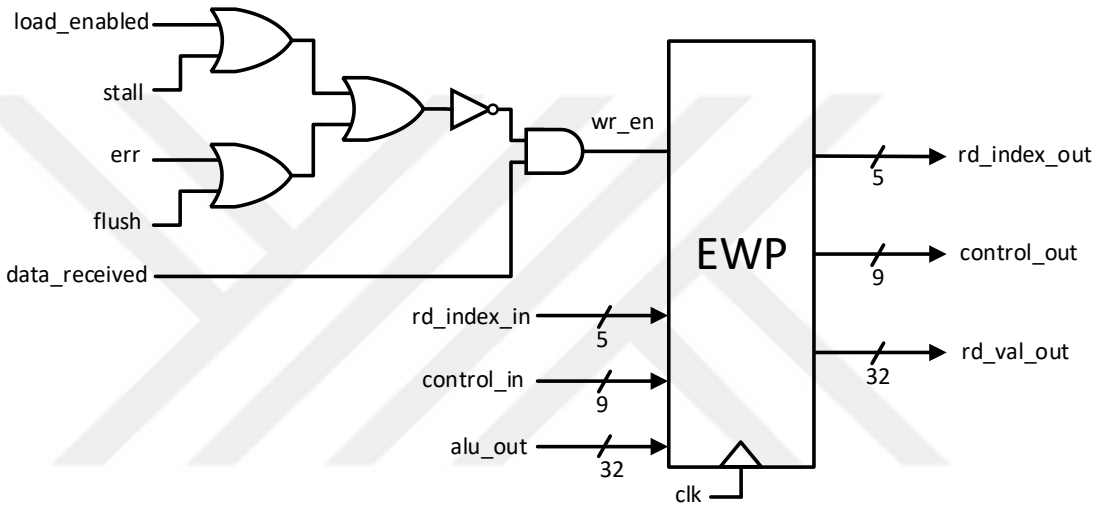

Figure 3.27. The Block Diagram of the EWP Register

3.2. Bus Control Logic

Replaces the bus protocol. The Bus Control Logic changes its behavior depending on *load_out*, *store_out* and *inst_out* signals (this order is used in explanation below). When an address is received, the placeholder sends the data at the address to the core, or from the core to a unit (depending on the operation), while setting the data received signal each time. The last fetched instruction address is stored and 4 is added. If a store instruction is performed the next instruction is pre-fetched by *store_out* signal selecting the memory address.

When an instruction fetch (‘001’) is performed, the data at the corresponding address is sent to the core at the next cycle. When a store (‘010’) is performed, the data is sent to the corresponding address with the *overwrite* signal set, and the next instruction is pre-fetched from the memory. When a load (‘100’) is performed, the data at the address is sent to the core while setting *load_recied* signal. If the core doesn’t request an enabled action, the action data gets deleted, or the core receives zeros.

The memory is selected when the MSB of the address is clear. If set;

- 0x80000000 corresponds to the UART baud rate counter. Load and store are enabled.
- 0x80000004 corresponds to UART rx data. Only loads are enabled. When the data is loaded, the content clear signal is sent to the UART unit.
- 0x80000008 corresponds to UART tx data. Only stores are enabled. Repetitive stores are ignored. When a store is issued the sent signal is set.
- 0x8000000C corresponds to XADC first data. Only load is enabled.
- 0x80000010 corresponds to XADC second data. Only load is enabled.
- 0x80000014 corresponds to XADC third data. Only load is enabled.
- 0x80000018 corresponds to UART rx data. Only loads are enabled. When performed, the content clear signal is not sent.

3.3. Memory

The memory is designed to forward the stored instructions and stored values to the core according to received addresses. The memory design uses the block RAM tiles within the FPGA to reduce the index comparison delay. The only way to program the memory is to bury the machine language codes into the module at the initialization of the FPGA. No bootloader is included. This unit can read and overwrite at the same time to different addresses.

The Memory Unit is designed to act as a placeholder of cache. To use RAM tiles of FPGA, the unit design works with 4 Byte data. Of FPGA, Although the core can make SB and SH instructions, the memory can't handle Bytes and treats them as words. This concludes with overwriting the other portions of the word. The designed memory can store 32768 words, which is 131072 Bytes. So, considering it is word aligned only 15-bit address bus is required. In Figure 3.28, I/O overview of the memory module is given.

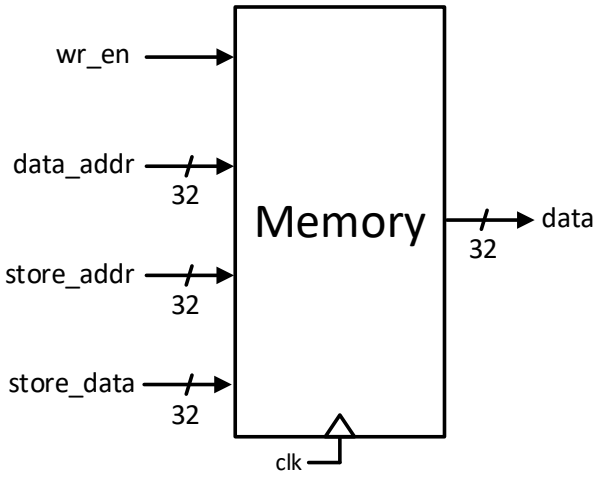

Figure 3.28. I/O Overview of the Memory

3.4. UART

UART communication is implemented due to the need for communication with external devices. UART design is full duplex. The frame is selected to contain 8 data bits and 2 stop bits. In Figure 3.29, the UART frame is given.

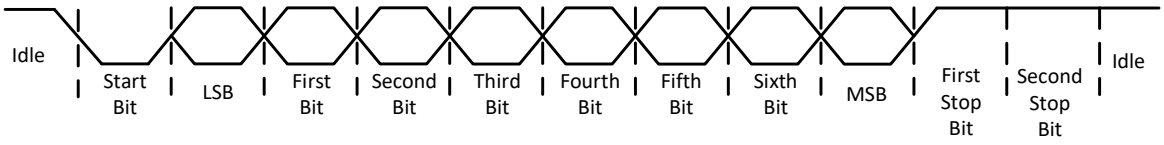

Figure 3.29. UART Frame of the Design

The baud rate of the UART is adjusted by changing the baud rate counter limit. This limit indicates what counter needs to reach for the next bit. Leading to calculation of baud rate by dividing the clock frequency with baud rate counter data. The design requires constant 15-bit baud rate counter data from the bus control logic. This data is stored within the bus control logic and adjusted by the core. The lower fifteen bits of the word are stored in the register to be forwarded to UART. In Figure 3.30, I/O Overview of the UART is given. In Figure 3.31, the layout of UART is given.

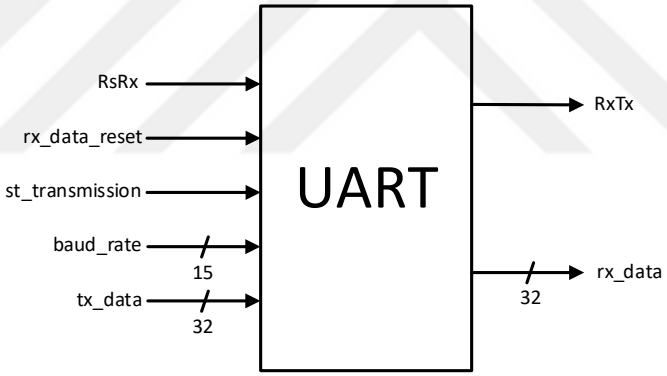

Figure 3.30. I/O Overview of the UART

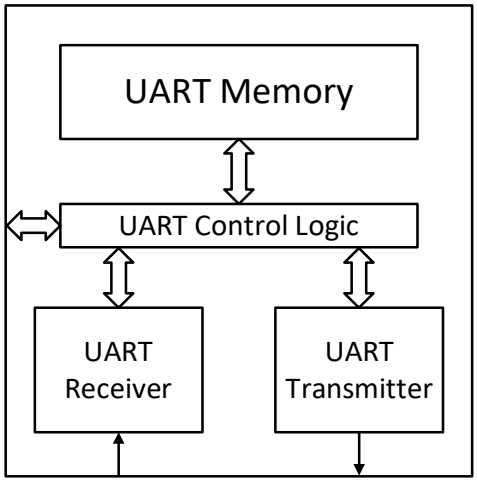

Figure 3.31. Layout of the UART

The baud rate value adjusts both bit sent, and bit received. The design doesn't have pre-set baud rate counter data. If the design is forced to work without adjusting the baud rate counter, the baud rate will be the same as clock frequency (100 Mbps in this case). If a change occurs in the baud rate value while the peripheral is sending or receiving the data, the change in baud rate counter is applied after the byte is sent or received.

The control mechanism of the UART transmission works as follows; When a 32-bit data is received from the core, it is written to one of the 32-bit registers. There are eight dedicated registers to store transmission data. The UART starts sending from LSB of the word to MSB of it by dividing the word to byte. When a data is stored, the data is treated as a word. However, when the transmission process begins, its byte is taken per transmission. Every byte in the UART transmission memory has an address. The control logic knows the address of the byte that changed most recently (MSB side) and the address that transmitted, or getting transmitted, most recently. When two addresses do not match, the control logic starts the process of transmission of the next address until the addresses match. When the addresses reach the maximum point, they get reset by ignoring the overflow in the address calculation. This forces the processor to assume the current state of UART transmission to not overwrite the data that is in transmission. If the data is overwritten to transmitting data, the sending byte is unchanged, but the next byte will be taken from recently written value and the older values that were waiting to be transmitted are unreachable after the transmission data overwrite. When a byte is selected to be transmitted, it is sent to the UART transmission. UART transmission sends the bytes received from UART control logic according to baud rate counter. When the transmission starts by setting the start signal, the *tx_byte* and *baud_rate* signals are sampled once then *busy_tx* signal is set. When the data is transmitted, *busy_tx* is cleared. This signal is tracked by control logic and next data is forwarded to UART transmission. In Figure 3.32, ports of the UART Transmitter is given.

The UART control logic doesn’t accept repetitive stores from Bus Control Logic. The *st_transmission* signal is used to notify UART if transmission data is in present. However, data is taken once when the signal is set for the first time while it was clear. Then it needs to be cleared again. If consequently stored, only the first data is taken.

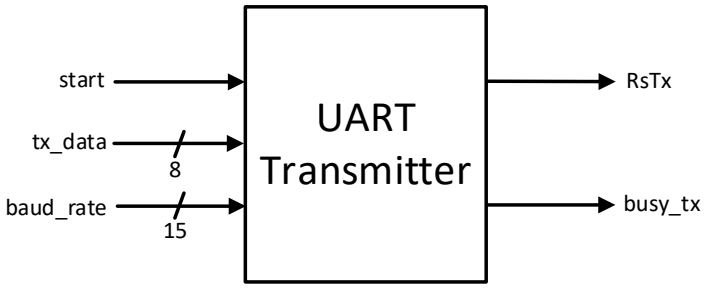

Figure 3.32. Ports of UART Transmitter

The control of the UART data received is as follows; When the data receive process starts by receiving start bit, the baud rate is sampled for counter and the *busy_rx* signal is set. When the signal gets cleared, the received byte is provided by *rx_data* signal. When the *busy_rx* signal is cleared the UART control logic stores the data to a 32-bit register. Each time a data is received the content of this register is left shifted by eight and new data is written to LSB side. The content of this register is always sent to the Bus Control Logic. The register content can be cleared by setting the *rx_data_reset* signal. If data is received at the same time with *rx_data_reset* signal, the register is cleared, and the data is written to LSB byte. In Figure 3.33, ports of the UART Receiver is given.

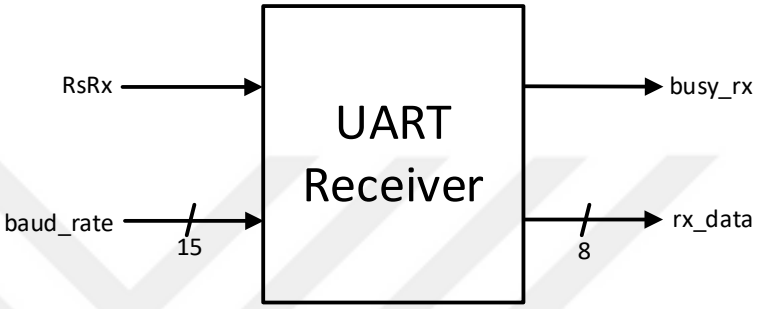

Figure 3.33. Ports of UART Receiver

3.5. XADC Unit

XADC Wizard (3.3), provided by AMD LogiCORE IP is implemented to convert analog inputs from XADC Pins. In Table 3.7, the summary of the used XADC Wizard IP block is given.

Table 3.7. Summary of Used XADC Wizard IP Block

Summary of XADC Wizard (3.3)	
Interface Selected	DRP
XADC operatingmode	channel_sequencer
AXI4Stream Interface	false
Timing Mode	Continuous
DCLK Freq(MHz)	100
Sequencer Mode	Continuous
Channel Averaging	None
Enable External Mux	false

The IP can only convert one analog input at a time, and this input is selected by analog mux. A control logic is designed to cycle through the addresses of analog mux selection. When the IP sends the ready signal ten times, the output is saved to the corresponding register for the processor to read the data. The selection is sampled ten times due to capacitor discharge failure. So, more time is provided for conversion to increase stability. The cycling is made according to *drdy_out* signal received from the IP. The longest data update per input requires 300 clock cycles. In Figure 3.34, I/O overview of the XADC is given.

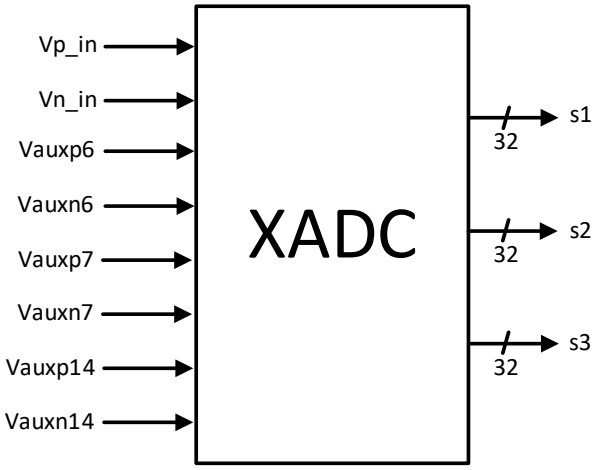

Figure 3.34. I/O of XADC

To store the received data three 32-bit registers are dedicated. However, the data received from the IP block is 16 bits, and 4 bits are from the LSB is noise. So, 12 bits of the data is stored in LSB side of the registers, and the rest are hardwired zeros. In Figure 3.35, layout of the XADC is given.

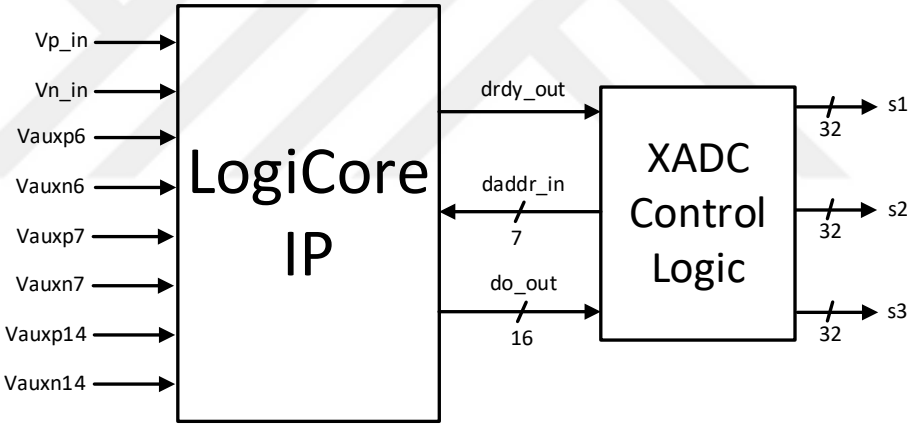

Figure 3.35. Layout of the XADC

4. RESULTS AND DISCUSSIONS

The single core 7-stage pipelined processor that designed for this thesis under the specifications of RISC-V ISA that uses RV32I base integer instruction set architecture, excluding FENCE and SYSTEM instructions, implemented on Basys-3 FPGA board with XC7A35T-1CPG236C integrates circuit via Vivado Design Suit version 2023.2. To test the core, a prototype circuit is designed. The required program for the core is written in RV32I assembly and then translated to machine language with custom made C++ program in Visual Studio Community version 2022. The core output is tested with the data received by UART. In personal computer PuTTy application is used for UART data transmissions.

4.1. Implementation of The Design

The design's hardware is described in Verilog HDL. Visual Studio Code application is used as editor, and Vivado Design Suit application is used as editor, simulator, synthesis tool, implementation tool, and bitstream generator for programming the Basys-3 board. The implementation and test of the designed system is made with 100 MHz clock frequency. The designed system and the core are simulated in Basys-3 and Kintex-7 boards for their maximum clock frequencies. To observe the impact of FPGA architecture, implementation results of the system and the core, implementations are made, and results are provided. By the maximum clock frequency, it meant that the limit of the clock frequency that cannot be raised while not making timing violation. The maximum clock frequencies found by manually adjusting the clock frequency of the implemented design. The separated core implementations are given to present information about the core, isolated from the peripherals and the memory. Information about the sliced logic for the Basys-3 maximum clock frequency is not given because the table had maximum of 1% increase in used sliced logics (not utilization percentage) when it is compared to the Basys-3 100 MHz sliced logic use. The total on-chip powers are not mentioned, but for the tested system, it is 0.181 W. The colors of the given implementation schematics under sections 4.1.1 and 4.1.2 are inverted and given as black and white.

The critical path mentioned in this section consists of logic delay and route delay. Logic delays are caused by the logical operations made within FPGA while describing the design. The route delays are caused by the structure of FPGA. If the route the signal takes increases, the route delay increases. The given critical paths are the ones which have the most delay in total. Hold time is referred to minimum time the signal is required to stay unchanged to be written in a register.

The separated core implementations are given to present information about the core, isolated from the influences of peripherals and the Memory. Although the core implementations cannot work on their own, the system implementations could work with some changes in the design (system implementation with 100 MHz clock frequency is already implemented and tested on Basys-3).

The memory utilization is given in Table 4.1, and it is the same for all the Basys-3 system implementations. For the Kintex-7, it has the same block tile use, but the total available resources increase to 445. The part number used in the Kintex-7 board is xc7k325tffg900-2.

Table 4.1. Used Memory Tiles by the System

Site Type			Used	Available	
Block RAM Tile	RAMB36/FIFO	RAMB36E1	32	50	50
	RAMB18		0	100	

4.1.1. Basys-3 Implementations

Implementing with 100 MHz Clock Frequency

For the system implementation at 100 MHz, the critical path is in between the memory and RFU. This path corresponds to the data access stage for load operations. The data is received from the RAM tile that is far away from the destination increasing the routing delay (66.871% of the total delay) and with the logic delay (33.129%). The load separator logic increases logic delay even more. The other lowest slacks are the same, the assumption for critical path of the system is between the memory and RFU can be made. The path of the worst hold time is between FDP and DRFP registers, which corresponds to decode stage. The next lowest hold slack is 0.037ns and within the UART.

For the core implementation, the critical path is between the RFEP register and FCU, and execution stage in between. The other lowest slacks are in between either RFEP register and FCU or RFEP and RFEP register (forwarding). This result created the assumption of execution stage has the most delay within the core. The lowest slack of the hold is located between FCU and DRFP register.

In Figure 4.1 and in Figure 4.2, the critical path of the system and core implementation to Basys-3 board with 100 MHz clock frequency is given, respectively. In Table 4.2, slice logic use, timing and power consumption results of the system and core implementations to Basys-3 boardwith 100 MHz clock frequency are given.

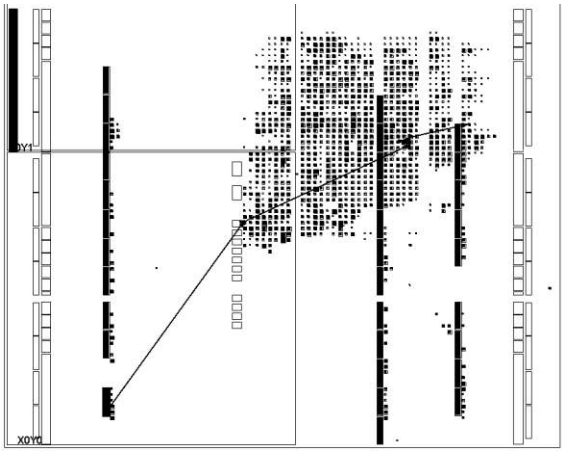

Figure 4.1.Critical Path of the System in Basys-3 Board with the Clock Frequency of 100 MHz

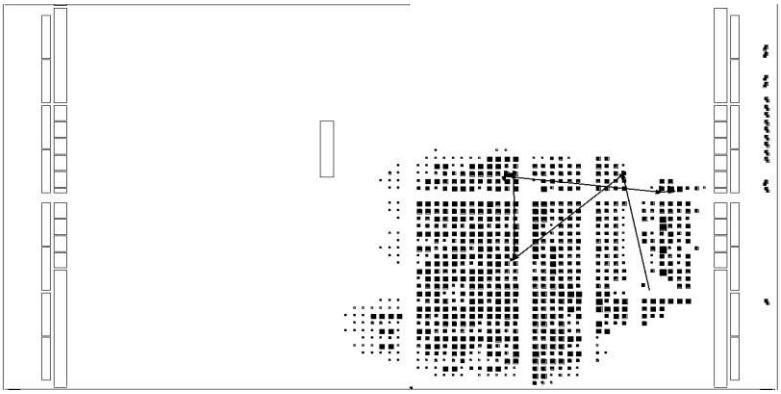

Figure 4.2. Critical Path of the Core in Basys-3 Board with the Clock Frequency of 100 MHz

Table 4.2. Basys-3 Implementation Results of System and Core with 100 MHz Clock Frequency

THE SYSTEM - Basys3 - 100 MHz					
SLICE LOGIC USE					
Site Type		**Used**		**Available**	**Utilization (%)**
Slice LUTs		2402		20800	11.55
LUT as Logic		2364		20800	11.37
LUT as Memory	**LUT as Distributed RAM**	22	38	9600	0.40
LUT as Shift Register		16			
Slice Registers		1785		41600	4.29
Register as Flip Flop		1785		41600	4.29
Register as Latch		0		41600	0.00
F7 Muxes		257		16300	1.58
F8 Muxes		128		8150	1.57

TIMING				
Timing	**Logic Delay**	**Route Delay**	**Total Delay**	**Remaining Time**
Setup	2.950ns	5.955ns	8.905ns	0.898ns
Hold	0.128ns	0.176ns	0.304ns	0.025ns

POWER CONSUMPTION						
Name	**The Core**	**Memory**	**Bus Control Logic**	**UART**	**XADC**	**Total**
Power (W)	0.032	0.072	0.001	0.001	0.002	0.108

THE CORE - Basys3 - 100 MHz					
SLICE LOGIC USE					
Site Type		**Used**		**Available**	**Utilization (%)**
Slice LUTs		1848		20800	8.88
LUT as Logic		1832		20800	8.81
LUT as Memory	**LUT as Distributed RAM**	0	16	9600	0.17
LUT as Shift Register		16			
Slice Registers		1462		41600	3.51
Register as Flip Flop		1462		41600	3.51
Register as Latch		0		41600	0.00
F7 Muxes		256		16300	1.57
F8 Muxes		128		8150	1.57

TIMING				
Timing	**Logic Delay**	**Route Delay**	**Total Delay**	**Remaining Time**
Setup	1.192ns	6.659ns	7.851ns	1.777ns
Hold	0.141ns	0.119ns	0.260ns	0.044ns

POWER CONSUMPTION (W)
0.025

Implementing with Maximum Clock Frequency

Implementation of the system is made with 130 MHz (clock period was 7.7ns) clock frequency. The critical path of the system is within the data access stage of the load operations. The data path is between Memory and RFU. This result concludes the assumption made for the critical path of the system. When the critical paths given for the system implementations are compared, the used design sources are tilted towards the center of RAM tiles to reduce routing delay of the data access stage for the load operations. The connection with the lowest slack of hold time is between the Bus Control Logic and the UART.

Implementation of the core is made with 150 MHz (clock period was 6.66ns) clock frequency. The critical path of the core is in between the RFEP register and FCU, the execution stage in between. The other lowest slacks are being in between either RFEP register and FCU or RFEP and RFEP register. Concludes the assumption made for the critical path of the core being true and, in the execution stage.

In Figure 4.3 and in Figure 4.4, the critical path of the system and core implementation to Basys-3 board with maximum clock frequency is given, respectively. In Table 4.3, timing and power consumption results of the system and core implementations to Basys-3 board with maximum clock frequencies are given.

Table 4.3. Basys-3 Implementation Results of System and Core with Maximum Clock Frequencies

THE SYSTEM - Basys3 - 130 MHz						
TIMING						
Timing		**Logic Delay**		**Route Delay**	**Total Delay**	**Remaining Time**
Setup		2.950ns		4.422ns	7.372ns	0.898ns
Hold		0.141ns		0.056ns	0.197ns	0.037ns
POWER CONSUMPTION						
Name	**The Core**	**Memory**	**Bus Control Logic**	**UART**	**XADC**	**Total**
Power (W)	0.042	0.093	0.001	0.001	0.003	0.140
THE CORE - Basys3 - 150 MHz						
TIMING						
Timing		**Logic Delay**		**Route Delay**	**Total Delay**	**Remaining Time**
Setup		1.588ns		4.998ns	6.586ns	0.036ns
Hold		0.164ns		0.115ns	0.279ns	0.064ns
POWER CONSUMPTION (W)						
0.039						

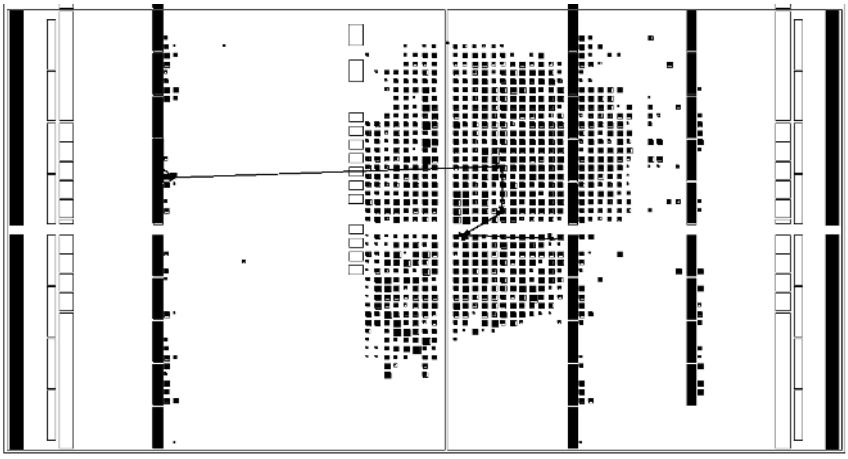

Figure 4.3. Critical Path of the System in Basys-3 Board with the Clock Frequency of 130 MHz

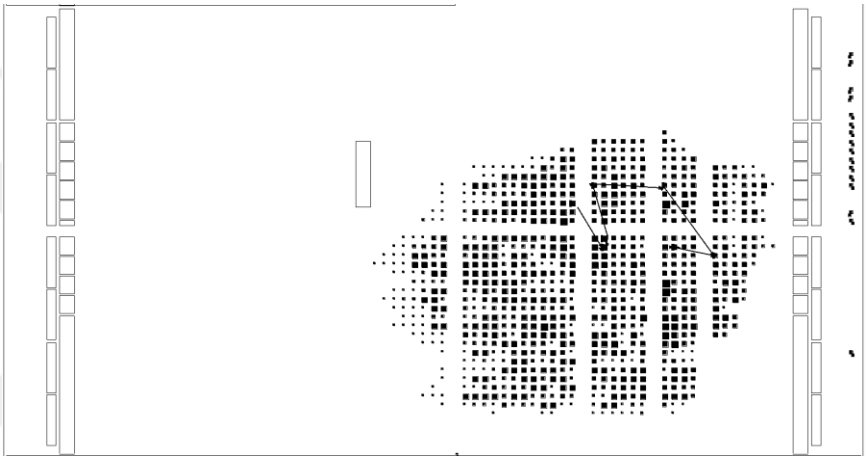

Figure 4.4. Critical Path of the Core in Basys-3 Board with the Clock Frequency of 150 MHz

4.1.2. Kintex-7 Implementation with Maximum Clock Frequency

Implementation of the system to Kintex-7 board is made with 212 MHz (clock period was 4.70ns) frequency. The critical path of the system in Kintex-7 shows similarity to Basys-3 system maximum frequency implementation with the exception in the 63% clock frequency increase and reduced resources. The path is between the memory and RFU. The lowest slack of hold timing connection is being the same and between Memory and UART.

Implementation of the core to Kintex-7 board is made with 273 MHz (clock period is assigned to 3.66ns) clock frequency. The critical path of the core in Kintex-7 shows similarity to Basys-3 core maximum frequency implementation with the exception in the 82% clock frequency increase. The used resources decreased minimally. The lowest slack of hold timing is between FCU and DRFP register (control signal).

In Figure 4.5 and in Figure 4.6, the critical path of the system and core implementation to Kintex-7 board with maximum clock frequency is given, respectively. In Table 4.4, timing and power consumption results of the system and core implementations to Kintex-7 board with maximum clock frequencies are given.

Table 4.4. Kintex-7 Implementation Results of System and Core with Maximum Clock Frequencies

THE SYSTEM – Kintex-7 - 212 MHz					
SLICE LOGIC USE					
Site Type		**Used**		**Available**	**Utilization (%)**
Slice LUTs		2228		203800	1.09
LUT as Logic		2190		203800	1.07
LUT as Memory	**LUT as Distributed RAM**	22	38	64000	0.06
LUT as Shift Register		16			
Slice Registers		1785		407600	0.44
Register as Flip Flop		1785		407600	0.44
Register as Latch		0		407600	0.00
F7 Muxes		257		101900	0.25
F8 Muxes		128		50950	0.25

TIMING				
Timing	**Logic Delay**	**Route Delay**	**Total Delay**	**Remaining Time**
Setup	2.071ns	2.356ns	4.427ns	0.146ns
Hold	0.118ns	0.101ns	0.219ns	0.073ns

POWER CONSUMPTION						
Name	**The Core**	**Memory**	**Bus Control Logic**	**UART**	**XADC**	**Total**
Power (W)	0.066	0.153	0.001	0.003	0.005	0.228

THE CORE - Basys3 - 273 MHz					
SLICE LOGIC USE					
Site Type		**Used**		**Available**	**Utilization (%)**
Slice LUTs		1881		203800	0.92
LUT as Logic		1865		203800	0.92
LUT as Memory	**LUT as Distributed RAM**	0	16	64000	0.03
LUT as Shift Register		16			
Slice Registers		1474		407600	0.36
Register as Flip Flop		1474		407600	0.36
Register as Latch		0		407600	0.00
F7 Muxes		256		101900	0.25
F8 Muxes		128		50950	0.25

TIMING				
Timing	**Logic Delay**	**Route Delay**	**Total Delay**	**Remaining Time**
Setup	0.954ns	2.516ns	3.470ns	0.092ns
Hold	0.100ns	0.102ns	0.202ns	0.075ns

POWER CONSUMPTION (W)
0.074

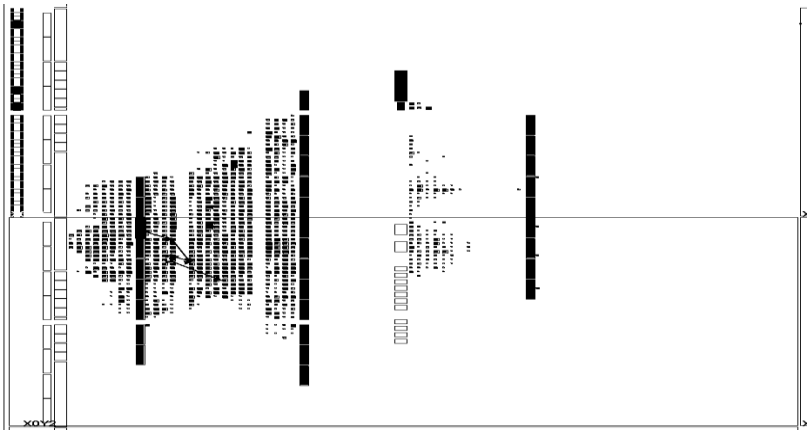

Figure 4.5. Critical Path of the System in Kintex-7 Board with the Clock Frequency of 212 MHz

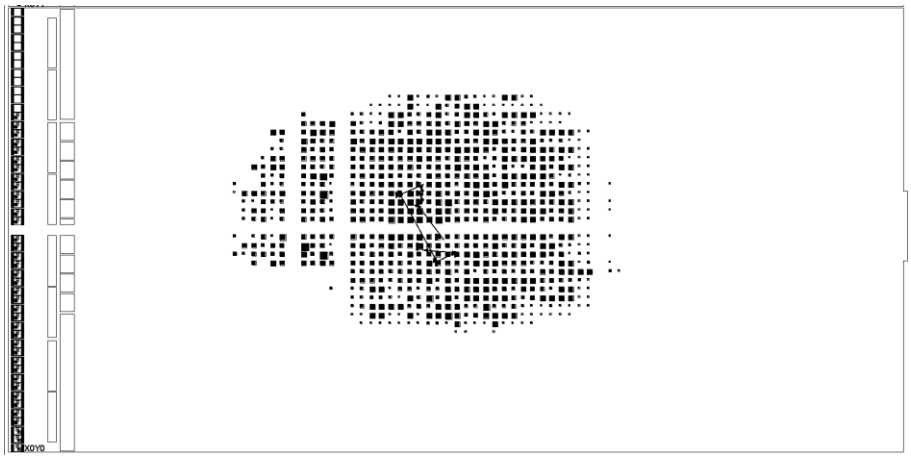

Figure 4.6. Critical Path of the Core in Kintex-7 Board with the Clock Frequency of 273MHz

4.2. Performance of The Core

A program is assumed for the processor to execute. If the executed parts of the program consist of 2000 instructions with 45% ALU instructions, 15% load instructions, 10% store instructions, 25% branch instructions (all of them branches, which is the worst-case scenario for the design) and 5% jump instructions. The performance of the core is determined by cycles per instruction (CPI) and million instructions per second (MIPS). In Table 4.5, general CPI per instruction type is given (considering the pipeline is filled). In Table 4.6, number of required clock cycle for the performance test program is given. In table 4.7, MIPS and runtime of the program is given with implemented frequencies. Initilization takes 5 clock cycles. So the total required clock for this program is 5005. In Table 4.7, initialization is included in the calculations.

Table 4.5. General CPI Per Instruction Type

NAME	ALU	Load	Store	Branch	Jump
CPI	1	3	1	1/5	5

Table 4.6. Number of Required Clock Cycle for the Performance Test Program

NAME	Instruction Execution	CPI	Required Clock Cycle To Execute
ALU	900	1	900
Load	300	3	900
Store	200	1	200
Branch	500	1/5	2500
Jump	100	5	500
Total	2000	2.5	5000

Table 4.7, MIPS and Runtime of the Program with Implemented Frequencies

	With 100MHz Clock	With 130MHz Clock	With 150MHz Clock	With 212MHz Clock	With 273MHz Clock
Program Run Time (µs)	50.05	38.5	33.37	23.61	18.33
MIPS	39.96	51.95	59.93	84.71	109.11

4.3. Testing the Core

Although the system and core is tested, including the modules within the core, in simulation. It is also tested in the Basys-3 board with 100 MHz clock frequency implementation. A program is written in assembly language to test the core. To translate the assembly language to machine language a C++ program is written in Visual Studio Community 2022. Simply, the written C++ program opens "assembly.txt" file located in the path of the executable file and then translates the lines one by one, writing the translated version to the "machine.txt" file. The program checks for errors. If the written assembly doesn't match with any of the defined instructions or if the boundary of used fields exceeds the defined limits, the program notifies the user through the console including the number of errors and writes error on corresponding line to translated machine language code. This program is named Assembly to Machine (AtoM). Another C++ which does the opposite (Machine to Assembly) is written to test if AtoM works properly.

The test code is written in assembly tests, x0 hardwire, operand forwarding, pipeline control at load, store, branch and jump instructions, repetitive execution of instructions and every extreme of instructions.

To start from somewhere (due to the only way to test the core is relative to the core itself), branch instructions used as reference and considered designed well and correctly. To test an instruction, branches were used as PC locking mechanism. If the execution result doesn't match with RISC-V ISA, used branch instruction branches itself, locking the processor to the same PC in the process. For example, (the given registers are given as an example and the code is simplified to explain including given illustrations) ADDI instruction is used, and x1 is changed to ten. Then, its equality check is made with x0 with the help of BEQ instruction. If equal (meaning that ADDI didn't change a thing in register), BEQ branches itself and locks the processor. If not equal, the next test is initiated and minus five is added with ADDI to x1. Another constant five is created with ADDI to register x2 to check the equality of x1 and x2 with the help of BNE. If not equal, it branches to itself. If an instruction is working differently or behaving differently with signed values, different conditions of instruction are tested accordingly.

For every ALU instruction a similar pattern is applied. A value is generated with the instruction which is getting tested. Then the desired outcome is created at another register with ADDI, LUI, and SLLI instructions or their combination. After that, the instruction outcome is tested with the constant, that is created with already tested instructions. The test is made either with BNE or BEQ (mostly BNE).

For forwarding tests, x1 is changed with ADDI. After that instruction, the changed destination register is used as operand several times with ADDI instructions to copy the operand value to other registers x2, x3 and x4 (ADDI x2, x1 (0)). Then their equality test with the x1 is made. The same applies for load forwarding. But before load, data within the address is changed with store instruction

to load the data. In addition to forwarding test, x0 not forwarding test is also made and x0 writing test is made at RFU.

For store and load instructions, they performed firstly with aligned addresses, then unaligned addresses with every combination. Both are tested with positive and negative immediate. When the load is tested, it is performed from the address that is changed with store instructions. To test the store load is performed after the store (the memory doesn't have byte addresses, it works with word aligned addresses. So, when a partial store is performed the other bits are zeros).

Before testing the branch instructions, they intentionally failed with every non-branching condition (if they branch, they branch to themselves). The branch (branching) and jump instruction test are made with every condition test (for branch) and positive and negative immediate with aligned and unaligned variants for each instruction. To test the branch and jump instructions two registers are chosen as counters. One (x1) to always flushed, and one (x2) to always executed. This is to test if the flushes and precision of address changes works correctly. When a branch or jump is executed in this test, it jumps over ADDI x1, x1, (1) and it changes to address to the ADDI x2, x2, (1) then proceeds to execute another branch or jump instruction. In Figure 4.7, an example to path of address change is given.

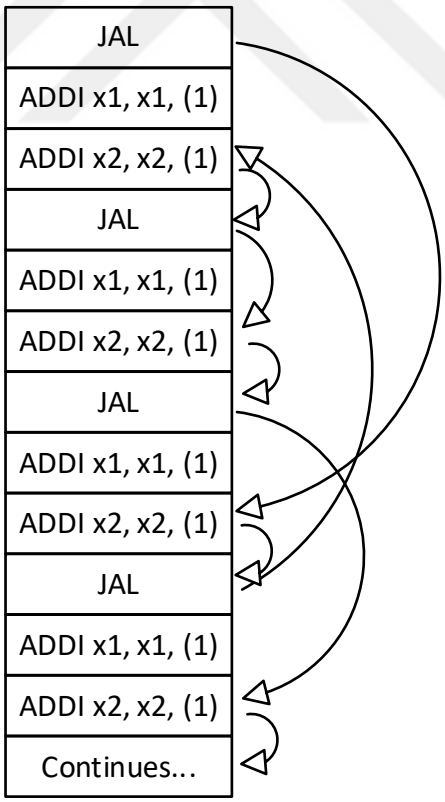

Figure 4.7. Test Illustration of JAL

To test the execution order, a JALR instruction is created with ALU instructions ordered by changing the address. Then, the created instruction (JALR x0, x4, (4)) at the register is stored in the

0x00000000. After that, JALR x4, x0, (0) is performed to reach that address, and the jump located that address jumps back to Link+4. In Figure 4.8, an illustration of instruction creation while changing the address is shown. When the test concludes the expected value at x1 is zero and expected value at x2 is forty-three. Repetitive jumps and branches are performed. But the x1 and x2 registers are untouched due to the test purpose to check if it is capable of repetitive execution. At the end of the code forty-eight is added to x1 and five is added to x2 to equalize them with ASCII '0'. Baud rate is adjusted, and x1 and x2 are sent via UART to console and '00' is read from the console. The addition is made with ADDI which proves the correctness of branch, jump and ADDI instructions. This concludes that, if an instruction didn't execute properly the processor would lock at the same PC address and never gets the chance to send the ASCII zeros. Test code consists of 979 lines of assembly code but some of them newer gets executed. In Table 4.8, statistics of the test code according to instructions are given.

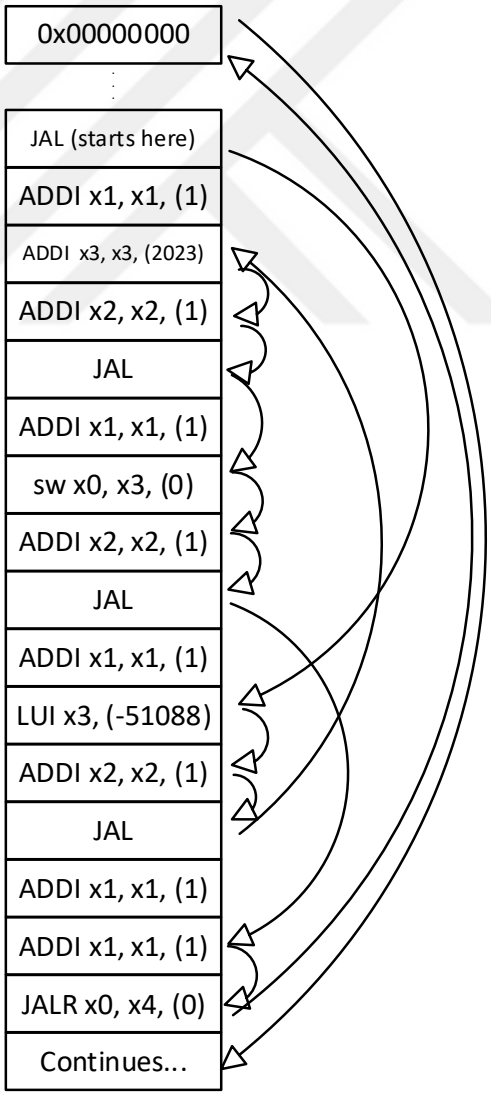

Figure 4.8. Orderly Changing PC to Create JALR Instruction

In Table 4.9, final statistics of the test code are given.

Table 4.8. Statistics of the Test Code According to Instructions

Type of Inst.	Name of Inst.	Code Includes	Inst. Executed	Not Executed	Created and Executed	Branch Taken	CPI	Total Clock Cycle
Arithmetic	**ADDI**	353	310	43	0	-	1	310
	ADD	21	21	0	0	-	1	21
	SUB	1	1	0	0	-	1	1
	LUI	16	16	0	0	-	1	16
	AUIPC	5	5	0	0	-	1	5
	Local Total	**396**	**353**	**43**	**0**	**-**	**-**	**353**
Logical	**XORI**	1	1	0	0	-	1	1
	ORI	1	1	0	0	-	1	1
	ANDI	1	1	0	0	-	1	1
	XOR	1	1	0	0	-	1	1
	OR	1	1	0	0	-	1	1
	AND	34	34	0	0	-	1	34
	Local Total	**39**	**39**	**0**	**0**	**-**	**-**	**39**
Shift	**SLLI**	22	22	0	0	-	1	22
	SRLI	23	23	0	0	-	1	23
	SRAI	23	23	0	0	-	1	23
	SLL	4	4	0	0	-	1	4
	SRL	4	4	0	0	-	1	4
	SRA	4	4	0	0	-	1	4
	Local Total	**80**	**80**	**0**	**0**	**-**	**-**	**80**
Compare	**SLTI**	6	6	0	0	-	1	6
	SLTIU	6	6	0	0	-	1	6
	SLT	6	6	0	0	-	1	6
	SLTU	6	6	0	0	-	1	6
	Local Total	**24**	**24**	**0**	**0**	**-**	**-**	**24**
Load	**LW**	47	47	0	0	-	3	141
	LH	16	16	0	0	-	3	48
	LB	18	18	0	0	-	3	54
	LHU	14	14	0	0	-	3	42
	LBU	14	14	0	0	-	3	42
	Local Total	**109**	**109**	**0**	**0**	**-**	**-**	**327**
Store	**SW**	21	21	0	0	-	1	21
	SH	12	12	0	0	-	1	12
	SB	12	12	0	0	-	1	12
	Local Total	**45**	**45**	**0**	**0**	**-**	**-**	**45**
Branch	**BEQ**	29	26	3	0	21	1/5	110
	BNE	201	201	0	0	4	1/5	217
	BLT	8	8	0	0	3	1/5	20
	BGE	9	9	0	0	4	1/5	25
	BLTU	8	8	0	0	3	1/5	20
	BGEU	8	8	0	0	4	1/5	24
	Local Total	**263**	**260**	**3**	**0**	**39**	**-**	**416**
Jump	**JAL**	15	15	0	1	-	5	75
	JALR	8	9	0	0	-	5	45
	Local Total	**23**	**24**	**0**	**1**	**-**	**-**	**120**

Table 4.9. Final Statistics of the Test Code

Types	Code Includes	Inst. Executed	Not Executed	Created and Executed	Total Clock Cycle	With Initilization of Pipeline (5 cycle)	CPI
ALU op. Total	539	496	43	0	496	-	-
Other op. Total	440	438	3	1	908	-	-
Main Total	979	934	46	1	1404	1409	1,508

4.4. The Design as BMS Monitoring

An assembly code is written for the processor to take converted data from XADC, then process them with Kalman filter and sends the results after approximately 80000 iterations via UART. The core is not stalled while the UART is transmitting the data, its potential is used with maximum. Due to averaging option not being implemented within the XADC wizard, data provided by XADC includes ripples and noise. This choice has been intentionally deselected to show the capabilities of the core. The parameters read from the test circuit are temperature, voltage and current. Although temperature and voltage readings from the sensors are relatively stable, the reading from current sensor is noisy. The voltage output of the current sensor changes according to its voltage inputs, this causes it to ripple with input voltage. When its low resolution (185mV/A) is included, it gets worse. The algorithm written for the test circuit consists of initial field, main field and functions, and they are explained briefly.

The initial section consists of the codes that will be run one time to set address data and constants. The addresses are used to jump to functions or to generate memory addresses. Constants include baud rate counter of UART (set to 99 which corresponds to 1M bps), the coefficients of Kalman filter and initial estimations of Kalman filter. Then the constants are stored in the memory until it is needed. Lastly before preceding to main field, within the initial field there is a counter that stalls the core for XADC to sample all channels.

Main section consists of two parts. The first part is where the Kalman filter equations are made for three sensor data. The second part is where the conversion of calculated data to meaningful version, ASCII conversion and UART transmission is made. The crossing between two parts is made with a counter. Counter counts the amount Kalman filter equations are made. When three of the equations for each sensor data are concluded, the counter is increased by one. When it reaches 8192, one portion of the second part is executed, and counter is reset. The portion of the second part is determined by links. When it is executed the first time, it executes the part where the data conversions are applied. Then other portions where data is loaded and sent via UART. The division of the portions are made as follows; Firstly, the conversions are separated, and then the separations are made for data load and UART transmission of data according to UART transmission buffer length. Register x6 is used to store the link of the next portion according to division. When portion is executed, JALR x6, x0, (400) is executed at the end of the portion and Kalman part starts again while

storing the portion link. This process repeats until it reaches the last portion where the x6 register is set manually to restart the address. Although UART transmission portions require minimal processing power, the convert portion requires mentionable processing power. This causes the processor to lose contact from the sensors for a moment. So, the algorithm does approximately 106000 Kalman iterations per sensor data while transmitting the pre converted data, then converts the 106000 Kalman iteration result per sensor to ASCII and repeats the process. The only exception is the first time the algorithm runs. When the algorithm runs the first time, it converts the 8192 iteration per sensor output via convert portion. Then it is 106000 iterations per sensor for the rest of the run time. For the first part of the main, where the Kalman filter equations run, there are three Scalar Kalman filter equations, one for each sensor. The Kalman filters of voltage and temperature sensors trust the readings. However, due to low resolution and ripple, the current readings are not trusted. For the first portion of the second part, the conversion results are stored in the memory for the transmission process. The conversions at the second part of the main include temperature, voltage and current data conversion and their ASCII values. The Kalman filter result of temperature is first multiplied with two hundred to convert sensor output to corresponding Celsius equivalent than it is divided with 4095 (resolution of the XADC) to convert XADC reading to meaningful number. The operations are made with fractional multiplication and division operations and first with multiplication to reduce the data loss. The result is then converted to ASCII and sent to memory. The same goes for voltage calculation as well. However, the temperature sensor voltage divider ratio is 1/2, and its output is 10mV per Celsius, but for the voltage sensor the voltage divider ratio is 1/5, and the read voltage directly corresponds to the value. So, data is multiplied by five and divided by 4095. Then stored in the memory after the result is converted to ASCII. The current sensor voltage divider ratio is 1/3. For the current sensor calculation, it has an offset which is half of the input voltage (Vcc), and it can show the current for both directions by reducing or increasing the offset voltage. So, firstly offset which is 2.5 and corresponds to 3412.5 is subtracted. After that the current calculation is made by multiplying the result by 3.96 (4 mA approximately corresponds to 1 bit). Then the result is converted to unsigned then its ASCII corresponding is made and stored. The sign of the current is stored. If the result is above 80mA and positive “discharging” ASCII corresponding data is taken from the memory and stored back to memory to be transmitted at UART transmission portion of the code. If the result is above 80mA and negative, the same is applied with “charging”. If neither, the same is applied with “N/A”. The other portions of the second part just include loading data and sending it back to UART. However, it must be mentioned that when a newline is made in PuTTy, it goes one below line with the same spacing that transmission is made. So, the spacing must be deleted and it is made for every line.

Functions include integer multiplication and division, ASCII converter, fractional addition and substruction, fractional multiplication and division, fractional multiplication division sign converter, fractional add subtract sign converter and integer multiplication division sign converter. The

fractional operations are made compatible with signed values by using the fractional sign converter function within the function by default. The integer sign converter function is not applied by default for optimization purposes. Optimizations like adding the smaller number (checking is made with unsigned branch) times bigger number for integer multiplication and for other operations are made. The integer multiplication and division functions are made with adding and subtracting, respectively. They are used with the other functions to reduce the number of codes. The ASCII converter takes a value than converts it to its corresponding ASCII decimal number. It reverses the order of numbers to be compatible with UART. The limit of conversion value is 9999. If it is bigger (unsigned) than this value it is set to 9999 ASCII. The result is stored in a word. Fractional operations are made with two-word data per value. Two of them include the integer part of different values and two of them include the fractional part of different values. Two digits (meant in decimal system, in binary system corresponds to 7 bits) are taken for the fractional part. To make multiplication and division computation, first reverting every number to unsigned by taking their two's complement if negative. The sign conversion function stores the sign of the result in the memory, and forwards converted values within registers. Then the data is multiplied with hundred (first left shifted by two then multiplied with 25) to add fractional part with its integer part to make the calculation same time for both parts. After that multiplication or division is made. After the operation the result is divided by hundred or thousand, depending on the operation and integer and fractional part is revealed with quotient and remaining of the division, respectably. The fractional addition or substruction is made within integers and fractions separately, then the integer is adjusted according to the fractional part. The fractional addition and subtraction have a separate function for sign adjusting. It takes the value as negative if any part is negative. The calculation is designed to be made for any given numbers including values with below one, zero and negative. If any of the value parts are negative the whole value is taken as negative. This is implemented considering the integer part could be zero, but the fraction could be negative. Finally, the sign conversion of the result is made according to stored sign data.

This code proves that the core can make approximately 320000 Kalman iterations and send them by UART after converting the data. The mentioned part has the runtime of the algorithm can change according to variable values read from sensors. To give a perspective about runtime it has been measured as 5.3 seconds. The test is made with 260 mV (130 mV at port) from temperature sensor, 3.3 V (0.66 V at port) from voltage reading and zero current (0.83 V at port) is received from current sensor.

4.5. Test Circuit

To run the BMS algorithm, a prototype circuit is made with a current sensor, a temperature sensor and voltage dividers. The module with ACS712 is used as current sensor. LM35 is used as a temperature sensor. A boost converter module is used to boost the 3.3V Vcc output of the FPGA to

5V, and the 5V output is supplied to the current sensor and to the temperature sensor. Three trimer potentiometers are used to divide the voltages for each sensor and one to divide the voltage of the battery. In Figure 4.9, the circuit scheme is given.

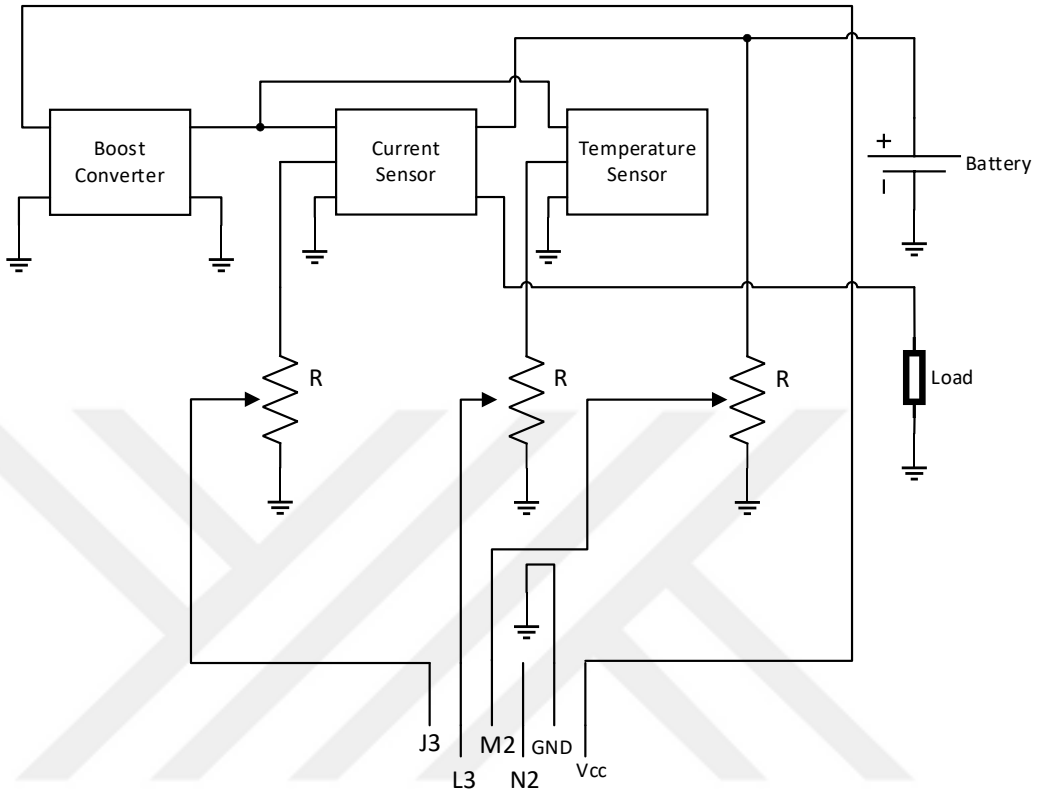

Figure 4.9. Test Circuit for the BMS Algorithm

4.6. Comments About the Design

The immediate generation for SLLI, SRLI and SRAI could have been used as I immediate rather than modifying it. This gives the opportunity to combine them with same alu_op code with their register-register versions.

RFU doesn't need comparison to give source register value, so it could have been designed in the decode stage. However, forwarding logic unit does need the conclusion of decode stage for comparison.

Received FCU pipeline control signals left separately for other RISC-V extensions.

Some forwarding logic and RFU comparisons be included in decode stage, like index zero comparison. For example, if rd index of the instruction is zeros, rd_write could be cleared if it is supposed to be set. This would reduce the logic and delay in general at R&F stage.

Load and store instructions use execution unit to calculate address. So, they could have been combined under the same alu_op code. The rs2 reordering at execution stage for the store operations is unnecessary. But is it required due to the non-existence of well-designed cache.

In FCU and Decode Unit the generation of their control signals are mentioned. Although control signals could have been designed to be more compact and effective, the design has been made with the intention of adding cache and additional extension from RISC-V.

Performing a load issues stall for two reasons. The first is in sufficient bus, and the second is load data forwarding. The instructions that will be executed after performing load could be checked at decode stage for needed stalls to inform the FCU to make the stall. This gives the possibility to reduce load instruction time to two if bus extension is made it could potentially reach to one cycle (considering pipeline is filled and no forwarding is required).

The stall and flush signals could be expanded for every pipeline register having one bit control signal received from FCU. This would increase the flexibility of the design.

In FCU there is a flip-flop that stores the flush counter. If a branch prediction is designed, it should be integrated with the flush counter which is updated to adjustable version (otherwise FCU flushes four times while fetching four instructions).

The custom UART design is made for general use than customized for the core. UART can’t store repetitive data to transmit due to possible stall at the pipeline while performing store (this is an intentional future).

Although the LogiCORE IP used for XADC can perform better, due to high resistance in the test circuit, the capacitor charge discharge rate can’t keep up to make proper conversion.

The main needs of the core are better forwarding logic handling, branch prediction, pipeline extension at execution stage, cache and bus interface.

5. CONCLUSIONS

This thesis sets out to explore and design a processor. After the literature review the need for fast, efficient, cheap and small sized processor requirement in BMS drew the attention. In regard to requirements, the RV32I base integer instruction set, part of RISC-V ISA, was meeting the requirements.

In the exploration of designing a processor, a simple and relatively small-sized processor design was proposed for BMS calculations. The designed processor can execute the instructions of RV32I base integer instruction set instructions excluding FENCE and SYSTEM instructions. The 7-staged pipelined structure of the designed processor allows the processor to reach 100 MHz clock frequency and above within Basys-3 FPGA. Within Basys-3 core can reach 150 MHz and within Kintex-7 it can reach 273 MHz clock frequencies. The programs that are executed in the processor are coded in assembly and translated to machine language with a custom C++ program written for this thesis. Then the translated machine language code is implemented in Memory of the processor. The output of the program is received via UART.

The soft processor was designed and implemented with Vivado Design Suit version 2023.2 software to the Basys-3 board. The design takes out 2402 LUT space in Basys-3, including the core, UART, XADC, Memory and the Bus Control Logic. If the core is separated and implemented, it takes out 1848 LUT space in Basys-3.

According to data gathered in the result section, the core can perform the instructions accurately. In section 4.3, this is proven with test algorithm that is written in assembly which sends output according to program execution. If an executed program is assumed to have 2000 instructions and if the instructions are as follows; 45% ALU, 15% load, 10% store, 25% branch (all branches) and 5% jump instructions, the CPI of the program is 2.5. To test the processor's BMS monitoring capability, a test circuit was designed. The test circuit consists of a current sensor AC712, a temperature sensor LM35, a DC-DC boost converter and voltage dividers. In section 4.4, monitoring algorithm is written in assembly to prove the core can process the received data. The monitoring algorithm consists of fraction operations within Scalar Kalman filter. Although some problems have been encountered in the behavior of the circuit due to ripple and practical imperfections, the result showed the processing power of the soft processor which can process more than 55000 Scaler Kalman filter iterations per second. The results of the monitoring algorithm received via UART. The UART data is gathered from PuTTy software within the personal computer.

This thesis presents an original design of a 7-stage pipelined RV32I base integer instruction set core to the literature. The designed processor is open to future improvements. Extensions like the M extension within RISC-V ISA can be added with some changes in the decoder, ALU and FCU. The most important potential improvements for the processor include the cache memory, bus interface and branch prediction. According to additions, the pipeline might be extended.

REFERENCES

Ali, M. U., Zafar, A., Nengroo, S. H., Hussain, S., Junaid Alvi, M., and Kim, H. J., 2019. Towards a smarter battery management system for electric vehicle applications: A critical review of lithium-ion battery state of charge estimation. Energies, 12(3): 446.

Andrea, D., 2010. Battery management systems for large lithium-ion battery packs. Artech house, Norwood, 290s.

Axelson, J., 2007. Serial port complete: COM ports, USB virtual COM ports, and ports for embedded systems. Lakeview Research, Madison, 379s.

Barsukov, Y., and Qian, J., 2013. Battery power management for portable devices. Artech house, Norwood,241s.

Cheng, K. W. E., Divakar, B. P., Wu, H., Ding, K., and Ho, H. F., 2010. Battery-management system (BMS) and SOC development for electrical vehicles. IEEE transactions on vehicular technology, 60(1): 76-88.

Chombo, P. V., and Laoonual, Y., 2020. A review of safety strategies of a Li-ion battery. Journal of Power Sources, 478(1): 228649.

Crompton, T. R., 2000. Battery reference book. Newnes, Woburn, 760s.

Dutta, A., Mitra, S., Basak, M., and Banerjee, T., 2023. A comprehensive review on batteries and supercapacitors: Development and challenges since their inception. Energy Storage, 5(1): e339.

Espedal, I. B., Jinasena, A., Burheim, O. S., and Lamb, J. J., 2021. Current trends for state-of-charge (SoC) estimation in lithium-ion battery electric vehicles. Energies, 14(11): 3284.

Feldman, J. M., and Retter, C. T., 1994. Computer Architecture a Designer's Text Bases On a Generic RISC. McGraw-Hill, Singapore, 617s.

Frenzel, L. E., 2016. Handbook of serial communications interfaces: a comprehensive compendium of serial digital input/output (I/O) standards., Newnes, Waltham, 325s.

Guo, Y., Wu, S., He, Y. B., Kang, F., Chen, L., Li, H., and Yang, Q. H., 2022. Solid-state lithium batteries: Safety and prospects. EScience, 2(2): 138-163.

Hannan, M. A., Lipu, M. H., Hussain, A., and Mohamed, A., 2017. A review of lithium-ion battery state of charge estimation and management system in electric vehicle applications: Challenges and recommendations. Renewable and Sustainable Energy Reviews, 78(1): 834-854.

Harris, S. L., Harris, D., 2022. Digital Design and Computer Architecture RISC-V Edition. Morgan Kaufmann, Cambridge, 564s.

Hennessy, J. L., and Patterson, D. A., 2003. Computer Architecture: A Quantitative Approach. Morgan Kaufmann, San Francisco, 883s.

Horie, T., 2014. How to make RISC-V Microcomputer using FPGA for programmer. The Regents

of the University of California, 366s.

Ledin, J., and Farley, D., 2022. Modern Computer Architecture and Organization: Learn x86, ARM, and RISC-V architectures and the design of smartphones, PCs, and cloud servers. Packt Publishing Ltd., Birmingham, 628s.

Lelie, M., Braun, T., Knips, M., Nordmann, H., Ringbeck, F., Zappen, H., and Sauer, D. U., 2018. Battery management system hardware concepts: An overview. Applied Sciences, 8(4): 534.

Lu, L., Han, X., Li, J., Hua, J., and Ouyang, M., 2013. A review on the key issues for lithium-ion battery management in electric vehicles. Journal of power sources, 226(1): 272-288.

Matsumura, N., 2023. Practical Battery Design and Control. Artech House, Norwood, 279s.

Meng, H., and Li, Y. F., 2019. A review on prognostics and health management (PHM) methods of lithium-ion batteries. Renewable and Sustainable Energy Reviews, 116(1): 109405.

Parhami, B., 2005. Computer Architecture From Microprocessors To Supercomputers. Oxford University Press, New York, 556s.

Patterson, D. A., and Hennessy, J. L., 2021. Computer Organization and Design RISC-V Edition The Hardware Software Interface. Morgan Kaufmann, Cambridge, 602s.

Patterson, D., and Waterman A., 2017. The RISC-V Reader: An Open Architecture Atlas Beta Edition 0.0.1. Strawberry Canyon LLC, California, 172s.

Plett, G. L., 2015. Battery management systems, Volume I: Battery modeling. Artech House, Norwood, 327s.

Pradhan, S. K., and Chakraborty, B., 2022. Battery management strategies: An essential review for battery state of health monitoring techniques. Journal of energy storage, 51(1): 104427.

Rahimi-Eichi, H., Ojha, U., Baronti, F., and Chow, M. Y., 2013. Battery management system: An overview of its application in the smart grid and electric vehicles. IEEE industrial electronics magazine, 7(2): 4-16.

Shen, M., and Gao, Q., 2019. A review on battery management system from the modeling efforts to its multiapplication and integration. International Journal of Energy Research, 43(10): 5042-5075.

Tan, X., Vezzini, A., Fan, Y., Khare, N., Xu, Y., and Wei, L., 2023. Battery Management System and Its Applications. John Wiley & Sons, Singapore, 392s.

Tanenbaum, A. S., and Austin T., 2013. Structured Computer Organization. Pearson Education, United States of America, 775s.

Ünsalan, C., and Tar, B., 2017. Digital system design with FPGA: Implementation using Verilog and VHDL. McGraw-Hill, 574s.

Wang, Y., Tian, J., Sun, Z., Wang, L., Xu, R., Li, M., and Chen, Z., 2020. A comprehensive review of battery modeling and state estimation approaches for advanced battery management systems. Renewable and Sustainable Energy Reviews, 131(1): 110015.

Warner, J., 2015. The handbook of lithium-ion battery pack design: Chemistry, components, types,

and terminology. Elsevier, Waltham, 239s.

Waterman, A., and Asanović, K., 2019. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191214-draft. RISC-V Foundation, 650s.

Weicker, P., 2013. A systems approach to lithium-ion battery management. Artech house, Norwood, 299s.

Winans, J., 2022, rvalp: RISC-V Assembly Language Programming https://github.com/johnwinans/rvalp [accessed: 26.08.2024].

Xiong, R., Li, L., and Tian, J., 2018. Towards a smarter battery management system: A critical review on battery state of health monitoring methods. Journal of Power Sources, 405(1): 18-29.

Yang, S., Liu, X., Shen, L., and Zhang, C., 2023. Advanced Battery Management System for Electric Vehicles. Springer, Singapore, 313s.

CURRICULUM VITAE

Polatcan POLAT graduated from Çukurova University, Faculty of Engineering, Department of Electrical and Electronics Engineering in 2021. Currently enrolled in the master's program in the same department from 2021. Working as a Research Assistant in the Department of Electrical and Electronics Engineering at Çukurova University since March 2024.