

ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL

**DESIGN AND IMPLEMENTATION OF A FAULT-TOLERANT
SUPERSCALAR OUT-OF-ORDER RISC-V PROCESSOR**



M.Sc. THESIS

Mustafa Ensar IŞKIN

Department of Electronics and Communication Engineering

Electronics Engineering Programme

FEBRUARY 2026

ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL

**DESIGN AND IMPLEMENTATION OF A FAULT-TOLERANT
SUPERSCALAR OUT-OF-ORDER RISC-V PROCESSOR**



M.Sc. THESIS

**Mustafa Ensar IŞKIN
(504221237)**

Department of Electronics and Communication Engineering

Electronics Engineering Programme

Thesis Advisor: Prof. Dr. Sıddıka Berna ÖRS YALÇIN

FEBRUARY 2026

**HATA TOLERANSLI SÜPERÖLÇEKLİ SIRA DIŞI YÜRÜTME
ÖZELLİKLİ RISC-V İŞLEMCİ TASARIMI VE GERÇEKLEMESİ**

YÜKSEK LİSANS TEZİ

**Mustafa Ensar IŞKIN
(504221237)**

Elektronik ve Haberleşme Mühendisliği Anabilim Dalı

Elektronik Mühendisliği Programı

Tez Danışmanı: Prof. Dr. Sıddıka Berna ÖRS YALÇIN

ŞUBAT 2026

Mustafa Ensar IŞKIN, an M.Sc. student of ITU Graduate School student ID 504221237 successfully defended the thesis entitled “DESIGN AND IMPLEMENTATION OF A FAULT-TOLERANT SUPERSCALAR OUT-OF-ORDER RISC-V PROCESSOR”, which he prepared after fulfilling the requirements specified in the associated legislations, before the jury whose signatures are below.

Thesis Advisor : **Prof. Dr. Sıddıka Berna ÖRS YALÇIN**
Istanbul Technical University

Jury Members : **Dr. Öğr. Üyesi Ayşe YILMAZER METİN**
Istanbul Technical University

Doç. Dr. Umut Engin AYTEN
Yildiz Technical University

Date of Submission : **1 January 2026**

Date of Defense : **11 February 2026**





To my family,



FOREWORD

I would like to express my sincere thanks to my family for their unwavering trust and unlimited support at every stage of this thesis study.

I would like to offer my gratitude to my esteemed thesis supervisor Prof. Dr. Sıddıka Berna Örs Yalçın, who always encouraged me to do better and guided me with her academic and technical knowledge.

I would like to thank my manager Mr. Refik Sever and Analog Devices for providing access to electronic design automation tools enabling me to perform synthesis and physical design stages and allowing me to take my thesis to the next level.

February 2026

Mustafa Ensar IŞKIN
(Engineer & M.Sc. Student)

TABLE OF CONTENTS

	Page
FOREWORD	ix
TABLE OF CONTENTS	xi
ABBREVIATIONS	xv
SYMBOLS	xvii
LIST OF TABLES	xix
LIST OF FIGURES	xxi
SUMMARY	xxiii
ÖZET	xxv
1. INTRODUCTION	1
1.1 Thesis Objective	2
1.2 Thesis Contributions	3
1.3 Thesis Scope	4
1.4 Thesis Organization	4
2. BACKGROUND AND RELATED WORKS	7
2.1 RISC-V Architecture	7
2.2 Processor Microarchitecture Concepts	8
2.2.1 Pipeline and hazards	8
2.2.2 Superscalar execution	8
2.2.3 Out-of-order execution	9
2.2.4 Branch prediction and recovery	9
2.3 Fault Tolerance Concepts	10
2.3.1 Fault types	10
2.3.2 Spatial redundancy	10
2.3.3 Temporal redundancy	11
2.3.4 Information redundancy	11
2.3.5 On-demand modular redundancy	11
2.4 Related Works	12
2.4.1 Superscalar RISC-V processors	12
2.4.2 Fault-tolerant RISC-V processors	12
2.4.3 Thesis positioning	13
3. SUPERSCALAR PROCESSOR ARCHITECTURE	15
3.1 Overview	15
3.1.1 Architectural goals	15
3.1.2 Pipeline stages overview	16
3.2 Instruction Fetch Stage	17
3.2.1 Multi-fetch unit	17
3.2.2 Program counter controller	21
3.2.3 Branch prediction system	22
3.2.3.1 Branch prediction	23
3.2.3.2 JALR prediction	24
3.2.4 Instruction buffer	25
3.3 Decode and Rename Stage	26
3.3.1 Instruction decoding	27
3.3.2 Register renaming	27
3.3.2.1 Register alias table	27
3.3.2.2 Resource allocation lists	29
3.3.3 Branch speculation support	31
3.3.3.1 Snapshot mechanism	31
3.3.3.2 Misprediction recovery	32
3.4 Data Control and Issue Stage	33
3.4.1 Reservation station	35

3.4.1.1 Tag system and common data bus monitoring	36
3.4.1.2 Instruction issue	37
3.4.1.3 Speculative flush	38
3.4.2 Reorder buffer	39
3.4.2.1 Allocation and result update	39
3.4.2.2 Store permission mechanism	40
3.4.3 Common data bus	41
3.4.4 Register file	42
3.5 Execute Stage	43
3.5.1 Functional unit structure	43
3.5.2 Branch resolution	45
3.5.3 Result selection	47
3.6 Memory Stage	47
3.6.1 Load store queue structure	48
3.6.2 LSQ operations	49
3.6.3 Store-to-load forwarding	50
3.6.4 Eager misprediction flush	51
3.6.5 Memory interface	52
3.7 Writeback Stage	53
3.7.1 Commit operation	53
3.7.2 Register alias table update	54
3.7.3 Free list update	55
3.7.4 Memory operations commit	55
3.7.5 Pipeline flush	56
3.7.6 Commit signals	57
4. ON-DEMAND REDUNDANCY TECHNIQUE	59
4.1 Motivation	59
4.2 Limitations of Existing Methods	59
4.2.1 Disadvantages of dual and triple-core lockstep	59
4.2.2 Advantages of pipeline-level protection	60
4.2.3 Need for on-demand modular redundancy	60
4.3 Triple Modular Redundancy Suitability of Three-Way Superscalar Design	61
4.4 Contributions of the Design	62
4.4.1 Combination of pipeline-level triple modular redundancy with on-demand modular redundancy	62
4.4.2 Low-latency mode transition	62
4.4.3 Early fault detection	62
4.4.4 Adaptive mode transition infrastructure	63
4.5 Triple Modular Redundancy Implementation	63
4.5.1 Majority voter	63
4.5.2 Protected structures	64
4.6 Error Correction Code Assumption for Large Memory Structures	64
4.7 Critical Fault Recovery with Register Alias Table Checkpoint	65
4.7.1 Dual use of checkpoints	65
4.7.2 Secure mode checkpoint strategy	65
4.7.3 Critical fault detection mechanism and proposed recovery flow	66
4.8 Summary	67
5. VERIFICATION AND RESULTS	69
5.1 Verification Methodology	69
5.1.1 Simulation environment and tools	70
5.1.2 Test programs and coverage	71
5.1.2.1 Constraint-based random test programs	71
5.1.2.2 Algorithm-based deterministic test programs	71
5.2 Functional Verification Results	73
5.3 Performance Analysis	74
5.3.1 Instructions per cycle comparison and speedup ratios	74
5.3.2 Pipeline stall and bottleneck analysis	75
5.3.3 Correlation analysis and performance prediction	79
5.4 Fault Injection and Reliability Verification	79
5.4.1 Fault injection mechanism	80

5.4.2 Fault injection test results	80
5.5 Logic Synthesis Results	81
5.6 Physical Design Results	82
5.6.1 Redundant register placement strategy	84
5.7 Critical Path Analysis and Evaluation.....	87
5.8 Discussion	87
6. CONCLUSIONS AND RECOMMENDATIONS.....	89
6.1 Summary of the Work.....	89
6.2 Obtained Results	89
6.2.1 Functional verification	89
6.2.2 Performance analysis	90
6.2.3 Fault tolerance	90
6.2.4 Physical implementation	90
6.3 Contributions of the Thesis	91
6.4 Limitations	91
6.5 Future Works	92
REFERENCES	93
CURRICULUM VITAE.....	97





ABBREVIATIONS

ALU	: Arithmetic Logic Unit
CDB	: Common Data Bus
DCLS	: Dual-Core Lockstep
ECC	: Error Correction Code
ILP	: Instruction Level Parallelism
IPC	: Instructions Per Cycle
ISA	: Instruction Set Architecture
LSQ	: Load/Store Queue
ODMR	: On-Demand Modular Redundancy
RAT	: Register Alias Table
RAS	: Return Address Stack
ROB	: Reorder Buffer
RS	: Reservation Station
SECDED	: Single Error Correction Double Error Detection
SEU	: Single Event Upset
TCLS	: Triple-Core Lockstep
TMR	: Triple Modular Redundancy
WAR	: Write After Read
WAW	: Write After Write
WNS	: Worst Negative Slack



SYMBOLS

A, B, C	: Triple Modular Redundancy Input Signals
B	: Branch Instruction Ratio
M	: Misprediction Rate
P	: Misprediction Penalty (cycles)
W	: Processor Width (issue width)
η	: Efficiency Ratio





LIST OF TABLES

	<u>Page</u>
Table 3.1: Processor pipeline stages and their main tasks.	16
Table 3.2: Technical specifications of the designed processor.	17
Table 3.3: Same-cycle forwarding example: source register resolution for three parallel instructions.	29
Table 3.4: Register renaming operation for an example instruction sequence. ...	30
Table 3.5: Physical register space regions and their properties.	34
Table 3.6: Operand tag values and their meanings.	36
Table 3.7: Speculative flush example scenario (<i>head_ptr</i> = 20, branch index = 5). ...	39
Table 3.8: Reorder buffer entry fields.	39
Table 3.9: Common data bus channel signals.	41
Table 3.10: ALU function selection signals and supported operations.	44
Table 3.11: RV32I instruction functional unit usage.	45
Table 3.12: Branch selection signals and branch decisions.	46
Table 3.13: Load store queue entry fields.	49
Table 3.14: Store-to-load forwarding and stall rules.	51
Table 3.15: Commit output signals.	57
Table 4.1: Processor operating modes.	61
Table 5.1: Constraint-based random test programs.	72
Table 5.2: Algorithm-based deterministic test programs.	73
Table 5.3: IPC comparison under random test scenarios.	75
Table 5.4: Comparison of 1-way and 3-way configurations under deterministic tests.	75
Table 5.5: Logic synthesis results.	82
Table 5.6: Synthesis stage power distribution.	82
Table 5.7: Physical design results.	84
Table 5.8: Physical design stage power distribution (1 GHz, 0.8V).	84
Table 5.9: Critical path comparison.	87



LIST OF FIGURES

	<u>Page</u>
Figure 3.1: Top-level block diagram of the designed superscalar processor.	18
Figure 3.2: Example of invalidation of subsequent instructions	20
Figure 3.3: Decode and rename stage general flow.	26
Figure 3.4: Register alias table (RAT) structure.....	28
Figure 3.5: Circular buffer-based free list structure.	29
Figure 3.6: RAT checkpoint and snapshot structure.	31
Figure 3.7: Source selection based on the most significant bit of physical register address.....	34
Figure 3.8: Data control and issue stage components and data flow between them.	35
Figure 3.9: Reservation station entry structure and stored information.	36
Figure 3.10: Common data bus broadcast network and component connections. .	42
Figure 3.11: Execute stage general structure and component connections.....	44
Figure 3.12: Misprediction detection mechanism.	47
Figure 3.13: Load store queue circular buffer structure.	48
Figure 3.14: Store-to-load forwarding logic.	52
Figure 3.15: LSQ and memory interface.....	53
Figure 3.16: In-order commit process.....	54
Figure 3.17: Pipeline flush mechanism.	56
Figure 4.1: TMR register recovery mechanism: comparison and correction with majority voting.....	64
Figure 5.1: Reservation station stall analysis.....	77
Figure 5.2: Previous stage bottleneck details.....	77
Figure 5.3: Instructions completed, mispredictions, and memory operation counts per 100 cycles.	77
Figure 5.4: Cumulative wait cycles for each reservation station.	78
Figure 5.5: Branch misprediction penalty percentage for reservation station idle time.	78
Figure 5.6: Physical layout view of processor core.	83
Figure 5.7: Connection density and metal layers after signal routing.....	83
Figure 5.8: TMR-protected register placement: (left) before optimization, (right) after optimization.	85
Figure 5.9: Distance distribution between TMR register replicas.	86



DESIGN AND IMPLEMENTATION OF A FAULT-TOLERANT SUPERSCALAR OUT-OF-ORDER RISC-V PROCESSOR

SUMMARY

Miniaturization in semiconductor technology has significantly increased the susceptibility of processors to environmental factors. Protection of systems used especially in space, aviation, and automotive applications against radiation-induced transient faults is critically important. Traditional fault tolerance methods, triple modular redundancy (TMR) and lockstep approaches, provide high reliability but cause significant performance and area costs due to their continuously active state.

In this thesis, a fault-tolerant superscalar out-of-order execution RISC-V processor was designed and implemented. The designed processor supports the RV32I base instruction set and offers high performance through its three-way superscalar structure. Using the on-demand modular redundancy (ODMR) approach, the system can switch between high-performance mode and secure mode through external control signals based on mission criticality.

The processor architecture has a six-stage pipeline structure: instruction fetch, decode and rename, issue, execute, memory access, and writeback. Out-of-order execution is implemented as a modern adaptation of the Tomasulo algorithm. The register alias table (RAT) is used for register renaming, and the reorder buffer (ROB) is used for speculative and out-of-order execution management. The RAT checkpoint mechanism provides single-cycle state restoration in case of branch misprediction.

In secure mode, the three parallel execution channels form the TMR structure, masking single-bit faults. Redundancy applied at the pipeline level offers faster fault detection and recovery compared to core-level lockstep methods. Pipeline flushing or state synchronization is not required for mode transition.

Verification work was performed using the Google RISC-V DV framework and Berkeley Spike reference model. All test results showed full compliance with the reference model. In performance analysis, an average instruction per cycle (IPC) value of 1.71 (on deterministic algorithm tests) and a maximum IPC value of 2.90 (on arithmetic-intensive workloads) were obtained. The three-way structure provided a 2.04x average speedup compared to the single-way structure. In fault injection tests, all single-bit faults injected into TMR-protected registers were successfully masked, and a 100% detection rate was achieved for multi-bit faults when correlated faults were prevented through physical design measures.

Physical design was performed in TSMC 16nm FinFET technology. Logic synthesis results show that approximately 142,000 cells are used and 1 GHz frequency is achievable. After TMR register placement optimization for improved fault tolerance, the achievable frequency is approximately 870 MHz with 140 mW power consumption.

This work presents a novel RISC-V processor architecture that combines superscalar out-of-order execution capacity with intra-core fault tolerance using the ODMR method. The obtained results demonstrate that a flexible and efficient solution is provided for applications requiring high reliability.



HATA TOLERANSLI SÜPERÖLÇEKLİ SIRA DIŞI YÜRÜTME ÖZELLİKLİ RISC-V İŞLEMCI TASARIMI VE GERÇEKLEMESİ

ÖZET

Yarı iletken teknolojisindeki sürekli minyatürleşme, entegre devrelerin çevresel faktörlere karşı duyarlılığını önemli ölçüde artırmıştır. Kozmik ışınlar ve yüksek enerjili parçacıklar yarı iletken malzemelerle etkileşime girerek, bellek hücreleri ve yazmaçlarda bit dönüşlerine neden olan tek olay bozulmalarına yol açabilmektedir. Teknoloji daha küçük düğümlere ölçeklendikçe, tek bir parçacığın bitişik birden fazla hücreyi etkilediği çoklu bit bozulmalarının olasılığı da artmaktadır. Bu olgular, güvenilirliğin büyük önem taşıdığı uzay, havacılık ve otomotiv uygulamalarında konuşlandırılan sistemler için kritik zorluklar oluşturmaktadır.

Üçlü modüler yedeklilik ve kilitli adım yürütme gibi geleneksel hata toleransı teknikleri, geçici hatalara karşı sağlam koruma sağlamaktadır. Üçlü modüler yedeklilik, tek bit hatalarını maskeleyerek için çoğunluk oylama ile üç yedekli donanım kopyası kullanırken, kilitli adım yöntemleri tutarsızlıkları tespit etmek için birden fazla işlemci çekirdeğini senkronize etmektedir. Ancak bu yaklaşımlar önemli alan ve güç ek yükü getirmekte ve daha kritik olarak, hata korumasının gerekmediği durumlarda bile aktif kalmakta, bu da normal çalışma sırasında performans düşüşüne neden olmaktadır.

İsteğe bağlı modüler yedeklilik, bu sınırlamaları ele alan esnek bir alternatif olarak ortaya çıkmıştır. İsteğe bağlı modüler yedeklilik sistemlerinde, yedeklilik görev kritikliğine göre dinamik olarak etkinleştirilebilir veya devre dışı bırakılabilmektedir. Kritik işlemler sırasında sistem tam yedeklilik ile korumalı modda çalışırken, normal işlemler sırasında yedekli kaynaklar performansı maksimize etmek için bağımsız olarak çalışabilmektedir. Bu tez, isteğe bağlı modüler yedeklilik kavramını süperölçekli sıra dışı işlemcilerde uygulamaktadır; bu minvalde yapılan bir çalışma ile karşılaşılmamış olup bu tez özgün bir fikir sunmaktadır.

Bu tezin temel amacı, isteğe bağlı yedekliliği destekleyen hata toleranslı süperölçekli sıra dışı yürütme özellikli bir RISC-V işlemci tasarlamak ve gerçeklemektir. Spesifik hedefler şunlardır:

- İsteğe bağlı modüler yedeklilik çalışmasına uygun üç yollu süperölçekli mimari tasarlamak; burada üç yürütme kanalı yüksek performans için bağımsız olarak veya hata toleransı için toplu olarak üçlü modüler yedeklilik birimi olarak çalışabilmektedir.
- Çekirdek seviyesi yerine boruhattı seviyesinde çekirdek içi uzaysal yedeklilik gerçeklemek, geleneksel kilitli adım yöntemlerine kıyasla daha hızlı hata tespiti ve toparlanma sağlamak.

- Boruhattı boşaltma veya durum senkronizasyonu gerektirmeyen düşük gecikmeli mod geçiş mekanizması geliştirmek.
- Hem dal yanlış tahmini toparlanması hem de radyasyon kaynaklı hata toparlanması için altyapı olarak hizmet eden çift amaçlı yazmaç takma ad tablosu anlık görüntüsü yapısı oluşturmak.

İşlemci RV32I taban tamsayı komut setini desteklemekte ve yalnızca makine modunda çalışmaktadır. Tasarım, gelecek çalışmalar olarak belirlenen çarpma/bölme (M) ve kayan nokta (F) uzantılarını, önbellek hiyerarşisini ve çok çekirdekli yapılandırmaları hariç tutmaktadır.

Tasarlanan işlemci altı aşamalı boruhattı mimarisi kullanmaktadır: getirme, kod çözme/yeniden adlandırma, yayın, yürütme, bellek ve geri yazma. Bu yapı, süperölçekli çalışma ve hata toleransı için ek geliştirmeler içeren Tomasulo algoritmasının modern bir gerçekleştirmesini temsil etmektedir.

Getirme aşaması, üç yollu süperölçekli çalışma için yeterli bant genişliği sağlayarak, hizalanmış komut tamponundan çevrim başına beş adede kadar komut almaktadır. İki bitlik doyurucu sayaç tabanlı dal tahmincisi, gerçekçi iş yüklerinde yüzde seksenden fazla tahmin doğruluğu elde etmektedir. Dönüş adresi yığını, fonksiyon dönüşleri için doğru tahminler sağlamaktadır.

Kod çözme/yeniden adlandırma aşaması aynı anda üç adede kadar komutu işlemektedir. Her komut, kontrol sinyalleri ve operand bilgilerini çıkarmak için anlık kod çözmeye tabi tutulmaktadır. Yazmaç takma ad tablosu, sahte bağımlılıkları (yazma sonrası yazma ve okuma sonrası yazma tehlikelerini) ortadan kaldırmak için yazmaç yeniden adlandırma gerçekleştirmektedir. Mimari yazmaçlar, 64 giriş içeren daha büyük bir fiziksel yazmaç dosyasına dinamik olarak eşlenmektedir.

Yayın aşaması, her kod çözülen komut için kaynak tahsis etmektedir. Üç paralel kanal, komutları rezervasyon istasyonlarına, yeniden sıralama arabelleğine ve yükleme/saklama kuyruğuna dağıtmaktadır. Yeniden sıralama arabelleği, sıralı kesinleştirme ve hassas istisna işleme için program sırasını korumaktadır. Kaynak tahsisi, kilitlenmeleri önlemek için hem yeniden sıralama arabelleği hem de yükleme/saklama kuyruğu kullanılabilirliğini dikkate almaktadır.

Yürütme aşaması, her biri tamsayı aritmetik ve mantıksal işlemleri yürütebilen üç aritmetik mantık birimi içermektedir. Komutlar, tüm kaynak operandlar kullanılabilir hale geldiğinde rezervasyon istasyonlarından yayınlanmaktadır. Ortak veri yolu, Tomasulo algoritmasının sonuç iletme mekanizmasını gerçekleyerek sonuçları bekleyen komutlara yayınlamaktadır.

Bellek aşaması, yükleme/saklama kuyruğu aracılığıyla yükleme ve saklama işlemlerini yönetmektedir. Bellek erişim gecikmesini azaltmak için saklamadan yüklemeye iletme gerçekleşmiştir. Doğru program yürütmesini sağlamak için bellek sıralama kısıtlamaları korunmaktadır.

Geri yazma aşaması, tamamlanan komutları program sırasına göre kesinleştirmektedir. Kesinleştirme üzerine sonuçlar mimari yazmaç dosyasına yazılmakta ve fiziksel yazmaçlar yeniden kullanım için serbest bırakılmaktadır. Yanlış dal tahmini tespiti,

yazma takma ad tablosu anlık grnts mekanizmasını kullanarak boruhattı toparlanmasını tetiklemektedir.

Yazma takma ad tablosu anlık grnts mekanizması, tek evrim gecikme ile erken yanlış tahmin toparlanması saėlamaktadır. Yanlış tahmin edilen dalın yeniden sıralama arabelleėi en st sırasına ulařmasını bekleyen toparlanma yntemlerinden farklı olarak, erken toparlanma kaydedilmiş bir kayıt noktasından doėru iřlemci durumunu anında geri yklemeaktadır. 16 giriřlik dairesel tampon, her uuřtaki dal iin dnř adresi yıėını tepe iřaretileri ve program sayacı deėerleri ile birlikte yazma takma ad tablosu anlık grntlerini saklamaktadır.

Bir dal yanlış tahmini tespit edildiėinde, ilgili kayıt noktası alınmakta ve yazma takma ad tablosu tek evrimde geri yklenmektedir. Bu mekanizma, ardışık durum yeniden yapılandırması gerektiren geleneksel yaklařımlara kıyasla yanlış tahmin cezasını nemli lde azaltmaktadır.

Gvenli modda,  sperlekli kanal aynı komut akışını eř zamanlı olarak yrterek l modler yedeklilik yapılandırması oluřturmaktadır. oėunluk oylayıcıları, her  kanaldan gelen ıkışları karřılařtırmakta ve bir kanal hata yařasa bile doėru sonucu retmektedir. Kritik kontrol sinyalleri ve yazma deėerleri bu yaklařım kullanılarak korunmaktadır.

l modler yedeklilik gereklemesi, radyasyon testi alıřmalarında belirlenen en savunmasız yapıları korumaya odaklanmaktadır. Arařtırmalar, radyasyon kaynaklı hataların yaklařık yzde yetmiř sekizinin belleklerde, yzde on beřinin ise yazma dosyalarında oluřtuėunu gstermiřtir. Buna gre, kritik boruhattı yazmaları l modler yedeklilik oylayıcıları ile korunmaktadır.

Yazma dosyası ve komut tamponu gibi byk bellek yapılarının, tek bit hatalarını dzeltebilen ve ift bit hatalarını tespit edebilen hata dzeltme kodları ile korunduėu varsayılmaktadır. Hata dzeltme kodları tasarımı ve entegrasyonu bu alıřmanın kapsamı dıřındadır.

Yedekliliėi ekirdek seviyesi yerine boruhattı seviyesinde gereklemenin temel avantajı, basitleřtirilmiş mod geiřidir. Geleneksel kilitli adım sistemleri, yedekli moda girmeden nce birden fazla ekirdeėi senkronize etmeyi gerektirir; bu durum boruhatlarını bořaltmayı ve yazma durumlarını kopyalamayı ierebilmektedir. Buna karřılık, nerilen mimari, yalnızca l modler yedeklilik oylayıcılarını etkinleřtirerek ve komut daėıtım mantıėını deėiřtirerek mod geiřini saėlamaktadır. Boruhattı bořaltma veya durum senkronizasyonu gerekmemektedir.

Mod geiři, zel bir kontrol yazmacı aracılıėıyla yazılım tarafından ayarlanabilen gvenli mod etkinleřtirme sinyali ile kontrol edilmektedir. Bu, iřletim sistemi veya uygulamanın grev ařaması veya evresel kořullara gre koruma seviyesini dinamik olarak ayarlamasına olanak tanımaktadır.

İřlemci, hem kısıtlama tabanlı rastgele hem de deterministik testleri ieren kapsamlı bir test paketi kullanılarak doėrulanmıřtır. Google RISC-V DV erevesi, eřitli kře durumları ve stres kořullarını uygulayan rastgele komut dizileri retmiřtir. Deterministik testler, eřitli hesaplama desenlerini kapsayan algoritma gereklemelerinden oluřmuřtur.

Tüm test sonuçları, Berkeley Spike referans simülatörü ile çevrim bazında karşılaştırılmıştır. Tüm testlerde tam uyum sağlanmış, RV32I komut setinin ve süperölçekli sıra dışı yürütme mekanizmalarının doğru gerçekleştiği teyit edilmiştir.

Performans ölçümleri, süperölçekli mimarinin etkinliğini ortaya koymuştur. Aritmetik yoğun rastgele testlerde, işlemci teorik maksimumun yüzde doksan yedisini temsil eden maksimum çevrim başına 2,90 komut değerine ulaşmıştır. Deterministik algoritma testleri, tek yayınlı skaler işlemciye kıyasla 2,04 kat hızlanmaya karşılık gelen ortalama çevrim başına 1,71 komut değeri vermiştir.

Dal tahmincisi, gerçekçi iş yüklerini temsil eden deterministik testlerde yüzde seksenden fazla doğruluk elde etmiştir. Rastgele testler, rastgele üretilen komut dizilerinde istatistiksel desenlerin bulunmaması nedeniyle daha düşük tahmin doğruluğu (yüzde 32-43) sergilemiştir.

Hata enjeksiyonu testleri, üçlü modüler yedeklilik korumasının etkinliğini doğrulamıştır. Üçlü modüler yedeklilik korumalı yazmaçlara enjekte edilen tek bit hataları çoğunluk oylayıcıları tarafından başarıyla maskelenmiş, mimari duruma bozulma yayılmamıştır. Çoklu bit hataları, fiziksel tasarım önlemleriyle eşlenik hatalar engellendiğinde yüzde yüz doğrulukla tespit edilmiştir.

İşlemci, TSMC 16nm FinFET teknolojisi kullanılarak sentezlenmiş ve yerleştirilmiştir. Lojik sentez yaklaşık 142.000 hücre üretmiş ve 1 GHz çalışmanın ulaşılabilir olduğunu göstermiştir. Yerleştirme sonrası zamanlama analizi, kritik yolda 106 ps'lik zamanlama ihlali ortaya koymuş, ulaşılabilir frekansı yaklaşık 900 MHz'e düşürmüştür. Üçlü modüler yedeklilik yazmaç yerleşim optimizasyonu sonrası, zamanlama ihlali 146 ps'ye çıkmış ve ulaşılabilir frekans yaklaşık 870 MHz'e düşmüştür.

Kritik yol, yazmaç takma ad tablosu anlık görüntüsü yapısından, özellikle yanlış tahmin toparlanması sırasında kayıt noktaları arasından seçim yapan çoğullayıcıdan geçmektedir. Gelecek çalışmalar bunu ek boruhattı aşamaları veya daha agresif sentez kısıtlamaları aracılığıyla ele alabilir.

Yüzde 20 değiştirme aktivitesi varsayımı altında güç analizi, yerleştirme sonrası aşamada toplam 131,6 mW güç tüketimi vermiştir. Bu, hem dinamik hem de kaçak güç bileşenlerini içermektedir.

Bu tez aşağıdaki özgün katkıları sunmaktadır:

1. İsteğe bağlı modüler yedeklilik uyumluluğu için tasarlanmış üç yollu süperölçekli mimari; burada aynı üç kanal, çalışma moduna bağlı olarak ya paralel komut yürütme ya da üçlü modüler yedeklilik koruması sağlamaktadır.
2. Çekirdek seviyesi kilitli adım yöntemlerine kıyasla daha hızlı hata tespiti ve toparlanma sağlayan, oylayıcı devreleriyle sınırlı minimal alan ek yükü ile boruhattı seviyesi uzaysal yedeklilik gerçekleştirilmesi.
3. Boruhattı boşaltma veya durum senkronizasyonu olmadan yalnızca oylayıcıların etkinleştirilmesini ve dağıtım mantığının değiştirilmesini gerektiren düşük gecikmeli mod geçiş mekanizması.

4. Tek çevrimli dal yanlış tahmini toparlanması sağlayan ve radyasyon kaynaklı hata toparlanması için altyapı olarak hizmet eden çift amaçlı yazmaç takma ad tablosu anlık görüntüsü yapısı.

Bu tez, isteğe bağlı modüler yedeklilik yaklaşımını kullanarak süperölçekli sıra dışı yürütmenin çekirdek içi hata toleransı ile birleştirilmesinin uygulanabilirliğini göstermiştir. Tasarlanan RISC-V işlemci, normal modda üç yollu süperölçekli yürütme ile yüksek performans elde ederken, güvenli modda üçlü modüler yedeklilik oylama ile tek olay bozulmasına karşı koruma sağlamaktadır.

Deneyisel sonuçlar, önerilen mimarinin hem fonksiyonel doğruluğunu hem de fiziksel gerçekleştirilebilirliğini doğrulamaktadır. Performans ölçümleri, komut seviyesi paralelliğinin etkin kullanımını teyit ederken, hata enjeksiyonu testleri üçlü modüler yedeklilik ile korumalı yapılarda tek olay bozulmasının maskelendiğini göstermektedir.

Gelecek çalışma yönleri şunları içermektedir:

1. Bellek yapıları için hata düzeltme kodları ile koruma eklenmesi.
2. RV32M ve RV32F komut seti uzantılarına desteğin genişletilmesi.
3. Gerçekçi bellek modelleri ile önbellek hiyerarşisinin entegrasyonu.
4. Kritik yol optimizasyonu yoluyla zamanlama ihlalinin çözülmesi.
5. Çekirdekler arası isteğe bağlı modüler yedeklilik gruplandırması ile çok çekirdekli yapılandırmaların araştırılması.



1. INTRODUCTION

Rapid advancements in semiconductor technology have enabled transistor dimensions to shrink below 10 nanometers. While this miniaturization provides significant gains in terms of performance and energy efficiency, it also brings various reliability challenges. Shrinking feature sizes increase the susceptibility of circuits to environmental factors and raise the probability of transient faults known as soft errors [1].

Particularly in critical application domains such as space, aviation, and automotive, cosmic rays and high-energy particles can interact with semiconductor materials, causing bit-flips known as single event upsets (SEU). When multiple bits are affected simultaneously, multiple bit upsets (MBU) occur, and the adverse effects on the system increase significantly [2,3].

Various fault tolerance methods have been developed against such errors. Different approaches exist, including spatial redundancy, temporal redundancy, and information redundancy, and these methods must be used together in reliable systems. Each method has different cost, performance, and protection characteristics, and selecting the most appropriate combination according to application requirements is critically important.

Fault-tolerant designs can be examined in two main categories according to their implementation level. The first category comprises core-level solutions where multiple processor cores are operated together using the lockstep method. The second category consists of intra-core solutions that provide protection at the pipeline level within a single core. Intra-core solutions enable faults to be detected at the stage where they occur and prevent their propagation [4]. Furthermore, this approach minimizes detection latency, offering faster recovery.

A significant disadvantage of traditional redundancy methods is the requirement for the system to continuously operate in redundant mode. This situation causes resource waste even during periods when fault risk is low. The on-demand modular redundancy

(ODMR) approach provides a solution to this problem [5]. In ODMR, the system can switch between redundant mode and independent operation mode as needed. Thanks to this flexibility, fault protection is provided during critical tasks while full performance is achieved under normal conditions.

The RISC-V instruction set architecture (ISA), with its open-source and royalty-free nature, provides a suitable platform for fault tolerance research. Its modular structure allows different protection mechanisms to be flexibly integrated.

1.1 Thesis Objective

The primary objective of this thesis is to design a RISC-V processor that combines the advantages of intra-core fault detection with the flexibility of on-demand redundancy.

Intra-core solutions enable faults to be detected at the pipeline level and allow early intervention. The ODMR approach allows the system to dynamically adjust its redundancy level. This thesis aims to combine the advantages of these two approaches in a single architecture.

The design has been planned to provide an infrastructure suitable for ODMR. Triple modular redundancy (TMR), which is widely used in space, aviation, and automotive sectors and has fault masking capability, has been selected as the redundancy level [6]. TMR's fault masking capability enables single-bit errors to be corrected without affecting the system output. In accordance with this choice, the processor has been designed as three-way superscalar with out-of-order execution capability.

The designed processor can operate in three different modes. In superscalar mode, three independent instruction streams are executed in parallel, providing high performance. In secure mode, the same instruction is processed simultaneously across three channels, achieving fault tolerance. Additionally, when required, the processor can operate in low-power mode using a single lane. Mode transitions can be performed with minimum latency thanks to the pipeline-level implementation.

This thesis constitutes the implementation of the research direction proposed in our previously published work as a preliminary study for this thesis. In that work, it was proposed that by implementing the ODMR method on-chip, more effective

fault correction could be achieved in secure mode, while more applications could be supported in performance mode [7].

1.2 Thesis Contributions

This thesis presents a novel processor architecture that combines the advantages of intra-core fault detection with the flexibility of on-demand redundancy (ODMR). The main contributions of this work can be summarized as follows:

1. Three-way superscalar architecture suitable for ODMR: The designed processor has been implemented with a superscalar structure featuring three parallel execution channels. This design decision offers a critical advantage for ODMR: during normal operation, three channels can theoretically reach an IPC value of three by processing different instructions in parallel, while in secure mode, the same three channels process a single instruction simultaneously, forming the TMR structure. The reason for choosing TMR as the redundancy level is its fault masking capability, which is critically important in space, aviation, and automotive sectors.
2. Pipeline-level redundancy with minimal overhead: Unlike core-level lockstep methods, spatial redundancy has been implemented within the pipeline. Since the processor has been designed as three-way superscalar, many structures already exist in triplicate, keeping the additional cost required for security low. The additional area cost required for TMR is limited only to voter circuits and is kept at a minimum level.
3. Low-latency mode transition: Implementing redundancy at the pipeline level minimizes mode transition latency. Switching from superscalar mode to secure mode does not require pipeline flushing or core state synchronization. Only the activation of voter circuits and modification of instruction dispatch logic are sufficient.
4. Dual-purpose recovery mechanism with RAT checkpoint: RAT checkpoint has been designed to enable single-cycle restoration of processor state in case of branch misprediction. This sixteen-entry structure has been developed to provide infrastructure that can also be used for recovery from radiation-induced critical faults

in secure mode. Thus, it provides infrastructure for a fast rollback mechanism to prevent multi-bit errors that cannot be masked by TMR from affecting the workflow.

1.3 Thesis Scope

This thesis has been conducted within the following limitations:

Instruction set: The RV32I base instruction set is supported. Multiply, divide, and floating-point extensions are out of scope.

Microarchitecture: The processor has been designed with a three-way superscalar structure and provides out-of-order execution support. A total of 64 physical registers, comprising 32 architectural registers plus 32 rename registers, and a 16-entry RAT checkpoint are used.

Fault tolerance: Critical control flow elements are protected with TMR. Error correction code (ECC) protection is assumed for large memory structures, but the implementation of ECC is out of scope.

Verification: Functional verification has been performed using the Google RISC-V DV framework, which is also used by many academic and industrial works, and validated against the Berkeley Spike reference model [8,9]. The effectiveness of the TMR mechanism has been measured through fault injection tests.

Physical implementation: The design has been synthesized using TSMC 16nm FinFET technology and has undergone place-and-route stages. A specialized placement strategy was developed to ensure adequate spacing between TMR register replicas, mitigating the risk of correlated faults from single particle strikes.

1.4 Thesis Organization

This thesis consists of six chapters.

The first chapter presents the motivation, objective, contributions, scope, and organization of the thesis.

The second chapter contains the fundamental concepts and literature review. Fault tolerance methods, RISC-V architecture, and related works are explained in detail in this chapter.

The third chapter describes the microarchitecture of the designed superscalar processor. Pipeline stages are addressed in separate subsections.

The fourth chapter presents the on-demand redundancy implementation. Operating modes and the critical fault recovery mechanism are explained in this chapter.

The fifth chapter reports the verification methodology, performance analysis, fault injection tests, and physical design results.

The sixth chapter summarizes the conclusions and presents recommendations for future work.



2. BACKGROUND AND RELATED WORKS

This chapter presents the fundamental concepts that form the basis of this thesis work and the related literature review. First, the RISC-V instruction set architecture is introduced, followed by an explanation of the fundamental concepts of superscalar processor design. The chapter continues with fault tolerance methods and finally examines related works.

2.1 RISC-V Architecture

RISC-V is an open-source instruction set architecture developed at the University of California, Berkeley in 2010 [10]. Unlike traditional commercial architectures, RISC-V does not require royalty fees and offers designers complete customization freedom. These features have led to widespread adoption in research and educational applications as well as industrial applications.

The fundamental design philosophy of RISC-V is modularity. The architecture consists of a mandatory base instruction set and optional extensions. The base instruction set, called RV32I, supports 32-bit integer operations.

The base instruction set uses six different instruction formats: R-type (register-register operations), I-type (immediate value operations), S-type (store), B-type (branch), U-type (upper immediate), and J-type (jump). This regular format structure allows simplification of instruction decoding hardware [11].

The RISC-V architecture defines three privilege levels: machine mode (M-mode), supervisor mode (S-mode), and user mode (U-mode) [12]. While only machine mode is typically used in embedded systems, all levels can be active in systems running an operating system. The processor designed in this thesis supports only machine mode.

2.2 Processor Microarchitecture Concepts

Modern processors increase performance by exploiting instruction level parallelism (ILP). This section explains the fundamental concepts of superscalar and out-of-order execution processors.

2.2.1 Pipeline and hazards

Pipelined processors can start a new instruction every clock cycle by dividing the instruction execution process into independent stages. The classic five-stage pipeline consists of instruction fetch, decode, execute, memory access, and writeback stages [11].

Three fundamental hazard types exist in pipeline design. Data hazards occur when an instruction depends on a result that has not yet been computed. Control hazards arise from the need to determine program flow before the outcome of branch instructions is known. Structural hazards occur when multiple instructions want to access the same hardware resource simultaneously.

Forwarding and stalling techniques are used to resolve data hazards. Branch prediction mechanisms have been developed to reduce the impact of control hazards [13]. Two-bit saturating counter-based prediction is widely used as a simple and effective method.

2.2.2 Superscalar execution

Superscalar processors have the capacity to fetch, decode, and execute multiple instructions per clock cycle. An N-way superscalar processor can theoretically complete N instructions per cycle. However, actual performance falls below this theoretical maximum due to data dependencies and resource conflicts [14].

The effectiveness of superscalar execution depends on the instruction-level parallelism in the program. The ILP value in typical programs ranges between 2 and 5 [15]. Speculative execution and dynamic scheduling techniques are required to achieve higher parallelism.

2.2.3 Out-of-order execution

Out-of-order execution enables instructions to be executed as soon as their operands are ready, independent of program order. This approach increases ILP utilization by reducing stalls caused by data dependencies.

The Tomasulo algorithm is the dynamic scheduling method that forms the basis of out-of-order execution [16]. The algorithm implements operand waiting and result broadcasting mechanisms through reservation stations (RS).

Register renaming plays a critical role in out-of-order execution. This technique eliminates false dependencies that do not exist at the software level. Architectural registers are dynamically mapped to a larger physical register pool. The register alias table (RAT) manages this mapping.

The reorder buffer (ROB) ensures that out-of-order executed instructions are committed in program order. This structure is necessary for correct handling of interrupts and exceptions. Management of speculative execution is also performed through the ROB.

2.2.4 Branch prediction and recovery

Speculative execution enables instruction fetching and execution to continue before branch outcomes are determined. The branch predictor predicts the branch decision and target address in advance. If the prediction is correct, a performance gain is achieved; if incorrect, speculative instructions are cancelled.

Two fundamental approaches exist for restoring processor state in case of misprediction. In the lazy recovery method, the processor waits for the mispredicted branch to reach ROB head, and the state is corrected by scanning backward. This approach causes high latency.

In the eager recovery method, snapshots of the RAT state are saved while branch instructions are being processed. When a misprediction is detected, the state is immediately restored from the relevant snapshot. This approach requires additional

hardware called RAT checkpoint or shadow RAT but significantly reduces recovery latency [14].

2.3 Fault Tolerance Concepts

Miniaturization in semiconductor technology has increased the susceptibility of circuits to environmental factors. This section explains fault types and protection methods developed against them.

2.3.1 Fault types

Hardware faults are classified into two main categories according to their permanence. Permanent faults arise from manufacturing defects or aging and persist even if the system is restarted. Transient faults generally originate from environmental factors and correct themselves [1].

Cosmic rays and high-energy particles can interact with semiconductor materials, causing bit-flips known as single event upsets (SEU). As technology scales, the probability of a single particle affecting multiple cells increases, and multiple bit upsets (MBU) occur [2,3].

In radiation tests conducted by Santos et al., it was observed that 78% of faults occurred in memories and 15% in register files [17]. These results clearly indicate that large memory structures and register files need to be protected.

2.3.2 Spatial redundancy

Spatial redundancy involves operating multiple copies of hardware in parallel. Double modular redundancy (DMR) provides fault detection by comparing the results of two copies. Triple modular redundancy (TMR) can both detect and mask faults through majority voting of three copies [6].

The lockstep method applied at the core level operates multiple processor cores synchronously. Dual-core lockstep (DCLS) provides fault detection, while triple-core lockstep (TCLS) provides fault masking. However, these approaches allow faults to propagate to the core output and increase recovery latency.

Spatial redundancy applied at the pipeline level enables faults to be detected at the stage where they occur. Intra-core protection prevents fault propagation, offering faster recovery [4].

2.3.3 Temporal redundancy

Temporal redundancy involves repeating the same operation at different times and comparing results. Additionally, rollback mechanisms that restore processor state to a known-good checkpoint upon error detection also fall under temporal redundancy; this thesis utilizes such an approach through the RAT checkpoint structure. Using only temporal redundancy needs minimal additional hardware but reduces performance. Tedeschi et al. developed a design that provides low-cost fault tolerance using temporal redundancy [18]. This method can be preferred when there are power and area constraints in embedded systems.

2.3.4 Information redundancy

Information redundancy provides fault detection and correction by adding additional bits to data [19]. In the examined literature, single error correction, double error detection (SECDED) is the most commonly used ECC implementation, which can correct single-bit errors and detect double-bit errors.

According to the work by Annink et al., ECC implementation can increase memory area by 33.5% and power consumption by 14%. However, with optimization techniques, this increase can be reduced to 3% [3].

Information redundancy applied at the pipeline level enables faults to be detected at the stage where they occur. DuckCore, developed by Li et al., is an example of this approach [20].

2.3.5 On-demand modular redundancy

A significant disadvantage of traditional redundancy methods is the requirement for the system to continuously operate in redundant mode. This situation causes resource waste even during periods when fault risk is low.

On-demand modular redundancy (ODMR) is a flexible approach proposed by Rogenmoser et al. [5]. In ODMR, the system can switch between redundant mode and independent operation mode according to task criticality. Fault protection is provided during critical tasks, while full performance is achieved under normal conditions.

The Trikarenos project presents a concrete implementation of the ODMR approach [21]. This design, implemented in 28nm technology, provides flexible fault tolerance for CubeSats.

2.4 Related Works

This section examines superscalar RISC-V processor designs and fault-tolerant processor works.

2.4.1 Superscalar RISC-V processors

Berkeley Out-of-Order Machine (BOOM) is the most recognized open-source superscalar RISC-V processor [22]. The third-generation design, known as SonicBOOM, demonstrated 6.2 CoreMark performance per MHz [23].

The Xiangshan project is a high-performance RISC-V processor developed by the Chinese Academy of Sciences [24]. This design, with six-way superscalar and out-of-order execution capabilities, targets industrial-level performance.

2.4.2 Fault-tolerant RISC-V processors

DuckCore, developed by Li et al., is among the first RISC-V designs providing pipeline-level fault protection [20]. The design presents a low-cost intra-core protection architecture covering program counter (PC), arithmetic logic unit (ALU), and register file units by optimizing standard error correction codes.

Dörflinger et al. proposed a comprehensive fault tolerance framework for RISC-V [4]. This framework allows different protection levels to be applied modularly.

Rogenmoser's Trikarenos project optimized the ODMR approach for small satellite applications [21]. It uses Ibex cores with ECC-protected SRAM and achieves up to 2.96x speedup in parallel mode.

The approach developed by Mach and Kohútka proposes the pipeline splitting technique as an alternative to the traditional lockstep method [25]. This approach provides fault tolerance with only 8% area overhead and saves resources by excluding large structures such as the branch predictor from protection. Faults in predictors are naturally corrected by the existing misprediction recovery mechanism.

2.4.3 Thesis positioning

When existing works are examined, fault-tolerant superscalar out-of-order execution RISC-V processor designs are not directly encountered. Works such as DuckCore provide pipeline-level protection but do not support superscalar out-of-order execution. High-performance designs such as BOOM do not include fault tolerance.

This thesis presents a novel RISC-V processor design that combines superscalar out-of-order execution capacity with intra-core TMR protection using the ODMR method. Thanks to the ODMR approach, a dynamic balance between performance and reliability can be established.



3. SUPERSCALAR PROCESSOR ARCHITECTURE

3.1 Overview

This section introduces the overall architecture of the designed three-way superscalar out-of-order execution RISC-V processor. The processor aims to achieve high performance by exploiting instruction-level parallelism and has been designed as a modern adaptation of the Tomasulo algorithm [16]. Similar RISC-V superscalar designs are actively being researched in the literature [24,26].

3.1.1 Architectural goals

The designed processor has been developed in accordance with four fundamental goals:

- The superscalar execution goal enables multiple instructions to be processed in parallel per clock cycle.
- The out-of-order execution goal enables ready instructions to be executed independent of program order.
- The speculative execution goal allows instruction fetching and execution to proceed speculatively without waiting for branch outcomes.
- The fast misprediction recovery goal enables the processor to return to the correct state in a single cycle when branch misprediction is detected.

In accordance with these goals, the processor eliminates false dependencies with register renaming technique, dynamically resolves data dependencies with tag-based operand waiting, and performs in-order commitment with the reorder buffer.

3.1.2 Pipeline stages overview

The processor pipeline consists of six main stages. The instruction fetch stage fetches five instructions per clock cycle, but some instructions may become invalid due to branch and jump instructions. Fetched instructions are forwarded to subsequent stages three at a time through a buffer. This asymmetric design compensates for branch losses, ensuring continuous feeding of the pipeline. Table 3.1 summarizes the main tasks of the pipeline stages.

Table 3.1: Processor pipeline stages and their main tasks.

Stage	Main Tasks
Instruction Fetch	This stage fetches five instructions from memory in parallel, predicts control flow with the branch prediction system, and buffers valid instructions.
Decode and Re-name	This stage decodes three instructions in parallel, maps architectural registers to physical registers, and saves snapshots for branch speculation.
Data Control and Issue	This stage allocates instructions to reservation stations and the reorder buffer, reads operand values, and broadcasts results over the common data bus (CDB).
Execute	This stage performs arithmetic and logical operations with three parallel functional units, evaluates branch conditions, and detects mispredictions.
Memory	This stage manages load and store operations with three parallel ports, performs store-to-load forwarding, and ensures memory consistency.
Writeback	This stage writes results to architectural state, commits up to three instructions in order per cycle, and frees physical registers.

Detailed explanations of each stage are provided in the following sections of this chapter: Instruction Fetch (Section 3.2), Decode and Rename (Section 3.3), Data Control and Issue (Section 3.4), Execute (Section 3.5), Memory (Section 3.6), and Writeback (Section 3.7).

To provide an overview, the fundamental structural features of the designed processor are summarized in Table 3.2. This table collectively presents the numerical parameters of components that will be explained in detail in subsequent subsections.

Table 3.2: Technical specifications of the designed processor.

Feature	Value
Architecture	RISC-V (RV32I)
Pipeline Depth	6 Stages
Instruction Fetch Width	5 Instructions/Cycle
Issue/Commit Width	3 Instructions/Cycle
Physical Register Count	64 (32 Architectural + 32 ROB)
Reservation Stations	3 Units
Load Store Queue (LSQ)	32 Entries
Branch Predictor	256-Entry 2-bit Counter, RAS, JALR Cache

Each of these features is explained in detail within the scope of the relevant pipeline stage in the following subsections. The top-level block diagram of the designed processor is shown in Figure 3.1.

3.2 Instruction Fetch Stage

The instruction fetch stage is the first stage in the superscalar processor's pipeline and plays a critical role in determining program flow. This stage performs the tasks of fetching instructions from memory in parallel, predicting branch behavior in advance, and buffering fetched instructions for subsequent stages. In the designed processor, the instruction fetch stage consists of three fundamental components: the multi-fetch unit, the branch prediction system, and the instruction buffer.

3.2.1 Multi-fetch unit

The multi-fetch unit is the central coordination unit designed to enable the superscalar processor to exploit instruction-level parallelism. This module has the capacity to fetch five instructions from memory per clock cycle and dynamically determines the validity of fetched instructions according to branch prediction results.

Although the designed processor has a three-way issue structure, the instruction fetch width has been set to five. An important rationale exists behind this asymmetric design decision.

All components of the processor attempt to bring the number of instructions per cycle close to the theoretical limit of three. In accordance with this goal, the instruction fetch

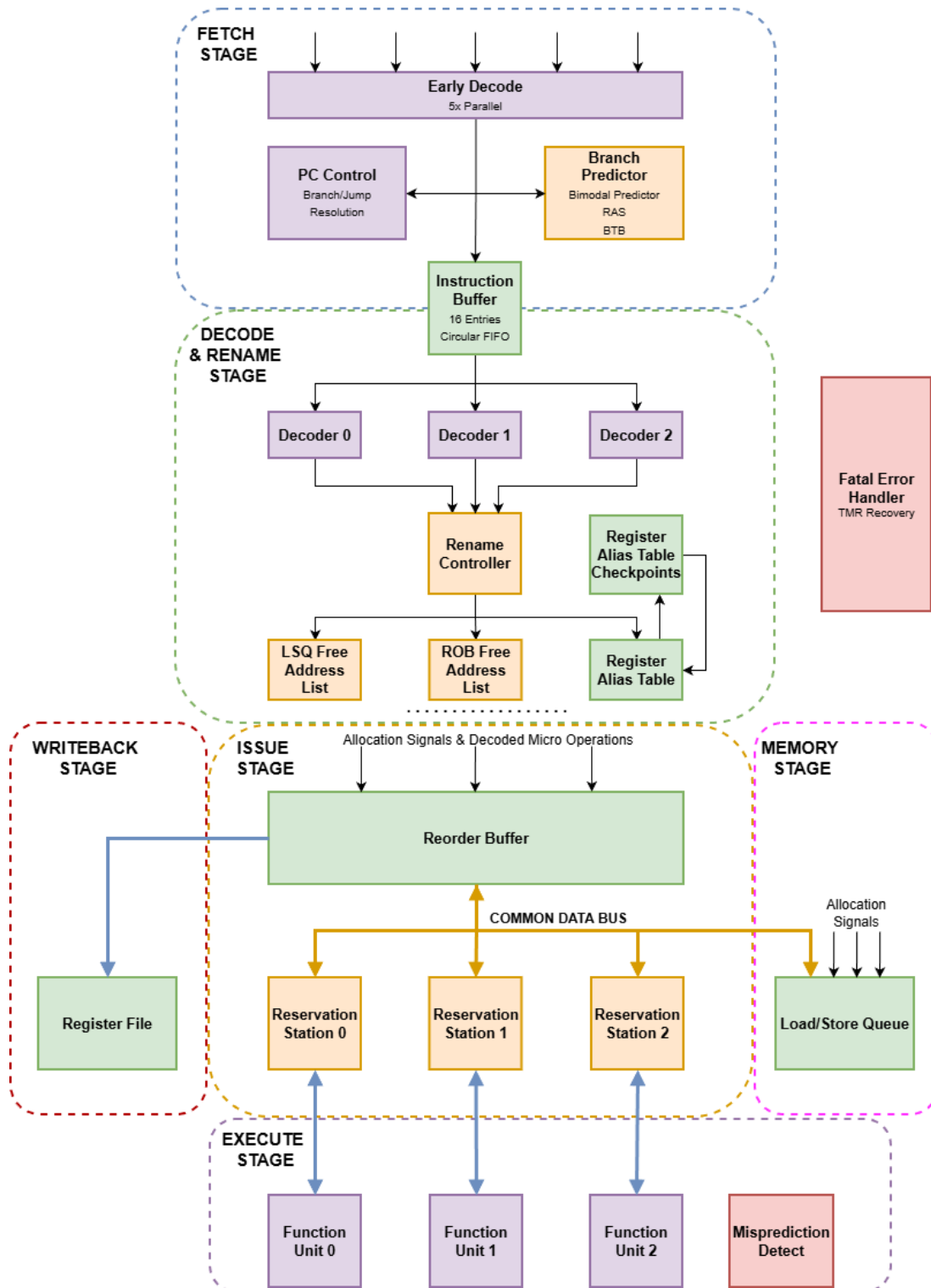


Figure 3.1: Top-level block diagram of the designed superscalar processor.

stage must also feed an average of three instructions per clock cycle to the forward portions of the pipeline. However, due to branch instructions predicted as “taken” and direct and indirect jump instructions, not every fetched instruction can be forwarded to the forward portions of the pipeline. In this case, instructions following the predicted branch or jump instruction are considered invalid and are not written to the instruction buffer.

The way to compensate for these losses is to fetch more than three instructions in subsequent cycles. Thanks to the five-wide fetch design, the goal is that even if some instructions become invalid due to branch and jump instructions, the average number of instructions written to the instruction buffer remains three. This approach prevents the instruction fetch stage from becoming a bottleneck for the pipeline.

The parallel instruction fetch mechanism enables high throughput by providing simultaneous access to five consecutive memory addresses. Separate address outputs and instruction inputs exist for each instruction. Instruction addresses are calculated starting from the program counter value at 4-byte intervals; this is because of the fixed 32-bit instruction length in the RISC-V architecture.

The validity of fetched instructions is determined according to branch prediction results. When any instruction’s branch or jump prediction yields a “taken” result, all instructions following this instruction are automatically marked as invalid. The reason for not forwarding these instructions to subsequent stages is that these instructions are predicted to not be executed; if these instructions were forwarded to subsequent stages, the processor would operate incorrectly by following the wrong program flow.

The invalidation logic operates in a cascaded manner: if the first instruction branches or jumps, the second, third, fourth, and fifth instructions become invalid; if the second instruction branches or jumps, the third, fourth, and fifth instructions become invalid, and this logic continues through the fifth instruction. This situation is illustrated in Figure 3.2.

An early-stage immediate decoder module has been added to each instruction to enable early calculation of branch target addresses. These modules extract immediate values

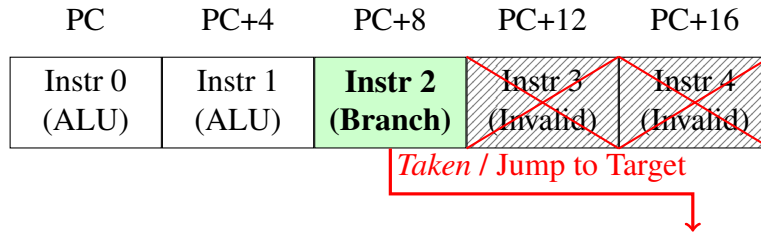


Figure 3.2: Example of invalidation of subsequent instructions

from within instructions without waiting for full instruction decoding. The regular structure of RISC-V instruction formats facilitates this early decoding operation.

The target address of branch instructions is calculated by adding the current program counter value and the immediate value within the instruction.

Performing this calculation in the instruction fetch stage allows branch prediction to determine the correct target address in a single clock cycle. The immediate decoder extracts values from different bit fields for B-type branch instructions and J-type jump instructions and applies sign extension.

When a branch or jump instruction is mispredicted, all speculative state in the pipeline must be cleared and the correct program flow must be resumed. Since up to three instructions can be processed simultaneously in the designed processor, multiple misprediction results can occur in the same cycle. Even if these results are calculated out of order, they are quickly prioritized to begin the clearing operation.

The oldest among the three possible mispredictions must be considered. The reason for this is that younger branches may be on the speculative path of the older branch. When the older misprediction is corrected, younger branches will already be cleared; therefore, separately processing younger mispredictions means performing unnecessary computation. This prioritization is performed by the misprediction handling logic in the multi-fetch unit: a misprediction from channel zero has priority over channels one and two, and a misprediction from channel one has priority over channel two.

When the clear signal is activated, the program counter is redirected to the correct target address and the instruction buffer is flushed.

3.2.2 Program counter controller

The program counter controller is a critical component responsible for determining the next fetch address. This unit decides how the program counter value should be updated by evaluating information from various sources.

Program counter update is performed according to 4 different scenarios, and these scenarios are evaluated according to a specific priority order.

The highest priority scenario is misprediction correction. When a misprediction signal comes from the execution stage, the program counter is redirected to the correct address. This situation overrides all other updates because re-establishing the correct program flow has the highest priority.

The second priority scenario includes indirect jump predictions. When a prediction is available for a JALR instruction, the program counter is updated to this predicted target address.

The third priority scenario covers direct branch and jump instructions. When a branch or JAL instruction is predicted as “taken,” the program counter is updated to the target address calculated by adding the immediate value to the current value.

The fourth and final scenario is normal sequential progression. When no branch or jump instruction is predicted, the program counter is advanced by 20 bytes for five instructions.

Program counter values for five instructions are calculated in parallel. Consecutive addresses are obtained by adding the current program counter value with fixed offsets of 4, 8, 12, and 16 bytes for each instruction. This parallel calculation minimizes instruction fetch latency.

Additionally, return address values are also calculated for jump instructions. Instructions classified as CALL (using x1 or x5 as destination) push their return addresses to the return address stack (RAS). This value corresponds to 4 bytes beyond the program counter.

3.2.3 Branch prediction system

The branch prediction system enables early detection and high-accuracy prediction of control flow changes. In typical programs, branch and jump instructions have a ratio between 15 and 25% [11]. Without effective branch prediction, each branch instruction causes a delay of several clock cycles in the pipeline; this situation significantly reduces processor performance.

The impact of misprediction penalty on superscalar processors is much more devastating compared to scalar processors. The fundamental reason for this is that more speculative instructions are being processed simultaneously in the wide pipeline, and all these instructions are wasted in case of a misprediction. The effective instructions per cycle value ($IPC_{effective}$) is expressed in terms of ideal value (W), branch ratio (B), misprediction rate (M), and penalty cycles (P) as follows:

$$IPC_{effective} = \frac{W}{1 + W \cdot B \cdot M \cdot P}. \quad (3.1)$$

In this equation, W represents processor width (1 for scalar, 3 for three-way superscalar), B represents branch instruction ratio, $M = (1 - \text{Accuracy})$ represents misprediction rate, and P represents misprediction penalty cycle count. With assumptions of 20% branch ratio ($B = 0.20$), 80% prediction accuracy ($M = 0.20$), and three-cycle penalty ($P = 3$):

$$IPC_{scalar} = \frac{1}{1 + 1 \times 0.20 \times 0.20 \times 3} = \frac{1}{1.12} = 0.89, \quad (3.2)$$

$$IPC_{super} = \frac{3}{1 + 3 \times 0.20 \times 0.20 \times 3} = \frac{3}{1.36} = 2.21. \quad (3.3)$$

Evaluated in terms of efficiency, the scalar processor maintains 89% of its ideal value ($\eta_{scalar} = 0.89/1 = 89\%$), while the superscalar processor can only maintain 74% ($\eta_{super} = 2.21/3 = 74\%$). This analysis clearly demonstrates why branch prediction accuracy is critically important in superscalar architectures.

Two main prediction mechanisms exist in the designed system: a two-bit counter predictor for conditional branch instructions and a JALR predictor for indirect jump instructions.

3.2.3.1 Branch prediction

A 256-entry two-bit saturating counter predictor is used for the direction prediction of conditional branch instructions. This predictor accesses the relevant counter value using the lower bits of the program counter as an index and generates a prediction based on the most significant bit of the counter. Each counter can be in one of 4 states: strongly not taken (00), weakly not taken (01), weakly taken (10), and strongly taken (11).

The fundamental rationale for using a two-bit counter is the weak performance that single-bit counters exhibit at loop exits. When a loop repeats 100 times and then exits, a single-bit counter immediately changes its direction and makes an incorrect prediction at the next loop entry. A two-bit counter provides resilience against such anomalous situations because it requires two consecutive wrong outcomes [13].

During the design process, a GShare-based tournament predictor was also developed and tested. The GShare predictor is indexed by XORing the program counter with global branch history and offers higher accuracy rates [27]. However, two critical limitations were identified in the practical application of this predictor. The first limitation is that access to RAM synthesis tools was not available during development, causing large tables to be synthesized as flip-flops; this situation causes unacceptable area consumption and critical path extension. The second limitation is that if access to large tables cannot be completed in a single clock cycle, pipeline additions are required; this addition increases misprediction penalty.

To evaluate the impact of these limitations, two predictor scenarios were compared. In the simulations performed, the two-bit counter predictor achieved an average 80% accuracy rate across different test programs, while the GShare predictor exhibited an average 83% accuracy rate. These values may vary depending on the branching characteristics of the application being run. In the first scenario, a 256-entry two-bit counter predictor is used with a 3-cycle misprediction penalty. In the second scenario, a 4096-entry GShare predictor is used, but a 4-cycle misprediction penalty is applied due to the pipeline. With assumptions of branch ratio $B = 0.20$ and processor width $W = 3$, IPC values were calculated as follows:

$$IPC_{simple} = \frac{3}{1 + 3 \times 0.20 \times 0.20 \times 3} = \frac{3}{1.36} = 2.21, \quad (3.4)$$

$$IPC_{gshare} = \frac{3}{1 + 3 \times 0.20 \times 0.17 \times 4} = \frac{3}{1.41} = 2.13. \quad (3.5)$$

The calculations show that the 3% accuracy improvement provided by the GShare predictor cannot compensate for the increased misprediction penalty. In accordance with this result, the two-bit counter predictor offering smaller area cost and lower misprediction penalty was selected.

3.2.3.2 JALR prediction

Indirect jump instructions read their target address from a register, so it is not known at fetch time. Therefore, a separate prediction mechanism is required for JALR instructions. The JALR predictor consists of two fundamental components: the JALR target buffer and the return address stack.

The JALR target buffer is a cache structure that stores the target addresses of previously executed JALR instructions. This structure is indexed by the program counter value and returns the most recently used target address. Since many indirect jumps, especially virtual function calls and switch statements, repeatedly jump to the same target address, this approach provides high accuracy.

In the designed system, the JALR target buffer has a capacity of 32 entries. Each entry contains a validity bit, tag value, and target address. Tag comparison is used to distinguish different JALR instructions.

The return address stack is a specialized structure for predicting function call and return patterns. The majority of function calls occur as call-return pairs; this situation naturally matches a last-in-first-out structure.

In the RISC-V architecture, call and return instructions are implemented with JAL and JALR instructions that use registers x1 or x5 as destination or source. When a call instruction is detected, the return address is pushed onto the stack; when a return instruction is detected, it is popped from the stack and used as the target address.

In the designed system, the return address stack has a depth of 8 entries. To support speculative execution, the stack top pointer is saved to the RAT checkpoint during each branch prediction and restored in case of misprediction.

3.2.4 Instruction buffer

The instruction buffer is a first-in-first-out structure that provides decoupling between the instruction fetch and decode stages. This decoupling allows both stages to operate at different speeds and increases pipeline efficiency.

The instruction fetch stage can operate at different speeds due to branch and jump instructions, and the forward stages of the pipeline can operate at different speeds due to reasons such as data dependencies and memory operations. Therefore, instructions obtained from instruction memory must be collected in a buffer and served to the forward stages of the pipeline from there. This approach reduces the complexity of the processor to a manageable level while also providing performance improvement.

Additionally, when the issue stage is stalled, the instruction fetch stage can continue operating until the buffer is full. This isolation prevents short-term stalls from affecting the entire pipeline.

The instruction buffer has been implemented as a circular buffer managed with head and tail pointers. The default buffer depth is 16 entries, and this value must be a power of two; this requirement simplifies modular arithmetic operations.

Each buffer entry contains various information: 32-bit instruction data, program counter value at prediction time, predicted next program counter value, branch prediction result, and return address stack top pointer value. This information is necessary for correct processing of the instruction in subsequent stages.

The backpressure mechanism slows down the instruction fetch stage when the buffer is full, preventing data loss. When at least five empty slots exist in the buffer, the fetch ready signal is activated. The five-slot requirement was determined to accommodate the maximum number of instructions that can arrive in a single cycle.

Performing instruction insertion when fewer than five empty slots exist would complicate the calculation of the next instruction address. In the current design, this complexity is avoided and address calculation logic is kept simple.

3.3 Decode and Rename Stage

The decode and rename stage is responsible for processing instructions received from the instruction buffer and performs two fundamental tasks: instruction decoding and register renaming. Instruction decoding analyzes RV32I instruction formats to generate control signals. Register renaming eliminates false dependencies that hinder out-of-order execution [16,28].

Three fundamental structures are used in this stage: the register alias table (RAT) that maps architectural registers to physical registers, free list structures that manage available physical register and queue addresses, and RAT checkpoint that stores snapshots for branch speculation. The general flow of this stage is illustrated in Figure 3.3.

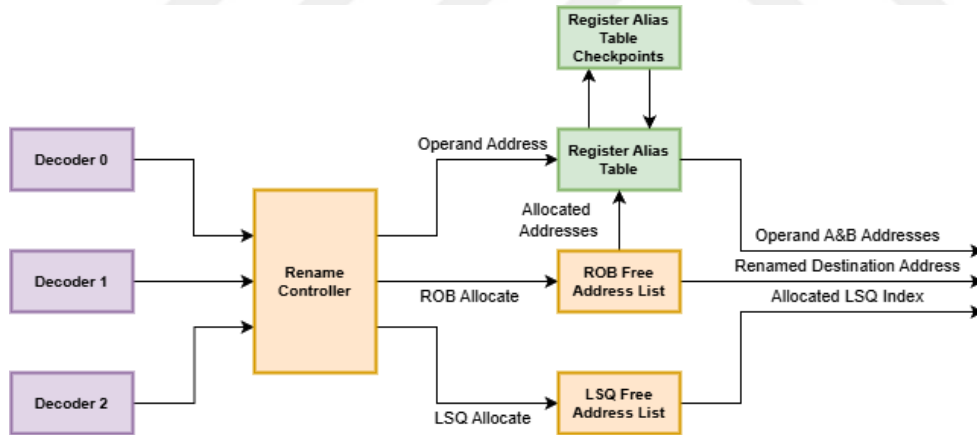


Figure 3.3: Decode and rename stage general flow.

This stage receives three instructions from the instruction buffer and performs the following operations in parallel for each:

1. Instruction decoding operation (RV32I format parsing)
2. Register renaming (RAT query and allocation)
3. Reorder buffer and load store queue resource allocation

4. If branch instruction, save snapshot to RAT checkpoint

All of these operations are completed in a single clock cycle. In cycle N , instruction reception, decoding, and renaming are completed combinatorially. In cycle $(N + 1)$, all information is forwarded to the next stage and the RAT is updated.

3.3.1 Instruction decoding

The instruction decoding unit is implemented with three separate RV32I decoder modules operating in parallel. Three instructions are processed simultaneously per clock cycle and control signals are generated. The fundamental reason for preferring this parallel structure is that instruction decoding is an entirely combinational process and there is no dependency between instructions at this stage.

Each decoder analyzes the 32-bit RISC-V instruction to generate architectural register addresses, control signals, and instruction type information. Architectural register addresses are five bits wide each for the first source register, second source register, and destination register. Control signals determine the arithmetic logic unit operation, memory access type, branch condition, and register write permission.

3.3.2 Register renaming

Register renaming is a critical mechanism that forms the foundation of out-of-order execution. This technique eliminates write-after-write (WAW) and write-after-read (WAR) type false dependencies by mapping architectural registers to a larger physical register file [29]. Thanks to this elimination, only the true data dependency read-after-write (RAW) condition remains.

The RISC-V architecture defines 32 architectural registers, but the designed processor has 64 physical registers. The structure and management of the physical register space is explained in detail in Section 3.4.

3.3.2.1 Register alias table

The register alias table (RAT) is a structure that tracks which physical register each of the 32 architectural registers is mapped to. This table, shown in Figure 3.4, consists of

32 entries that are 6-bits wide. Each entry holds the current physical register address of the relevant architectural register. The most significant bit indicates whether the physical register is located in the register file or in the reorder buffer.

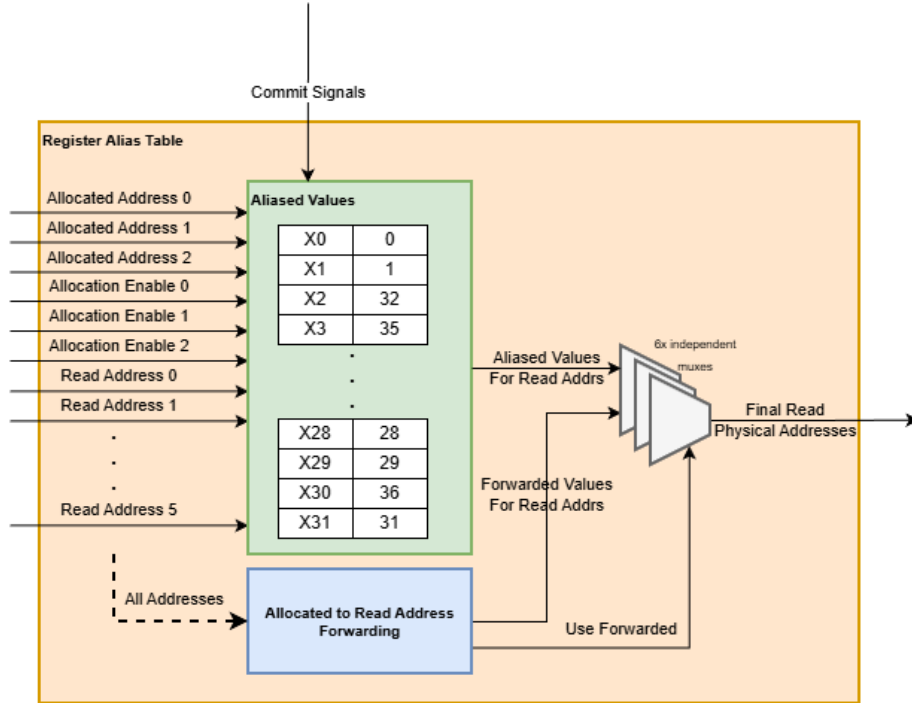


Figure 3.4: Register alias table (RAT) structure.

The multi-port structure of the table is designed to support three-way superscalar execution. Six read operations can be performed per clock cycle; this number corresponds to two source register queries for each of three instructions. Additionally, three write operations can also be performed in parallel.

As an important design decision, the zeroth architectural register is always mapped to the zeroth physical register. In the RISC-V architecture, the zeroth register contains a fixed zero value, and this special case is handled in hardware.

During parallel renaming of three instructions, dependencies between instructions must be handled correctly. For example, if the first instruction writes to a register and the second instruction uses the same register as a source, the second instruction should receive the first instruction's new physical register address, not the old value in the table. This requirement is met with same-cycle forwarding logic.

When source register addresses are queried, comparison is made with destination registers of previous instructions. If there is a match, the newly allocated physical register address is used instead of the table value. Forwarding priority is given to the most recent instruction; this prioritization is mandatory to preserve program order. This process is shown in 3.3.

Table 3.3: Same-cycle forwarding example: source register resolution for three parallel instructions.

Instruction	Source 1	Source 2	Dest	S1 Fwd	S2 Fwd
ADD x1, x2, x3	x2	x3	p33	–	–
SUB x4, x1, x5	p33	x5	p34	Instr 0	–
XOR x6, x1, x4	p33	p34	p35	Instr 0	Instr 1

3.3.2.2 Resource allocation lists

In out-of-order execution architecture, physical register and queue entry allocation for memory operations must be performed for each new instruction. These allocation operations are managed through circular buffer-based free lists.

The free register list is a critical structure that manages available physical register addresses. In a three-way superscalar architecture, up to three instructions are processed per cycle and each instruction potentially requires a physical register allocation.

In the traditional free list approach, finding three independent free resources requires complex priority encoder logic. In this design, the index-as-value structure has been preferred to solve this problem. Each entry in the circular buffer has a value equal to its own index: entry 0 contains value 0, entry 1 contains value 1. In this approach, the buffer does not store actual data but only manages which indices are available.

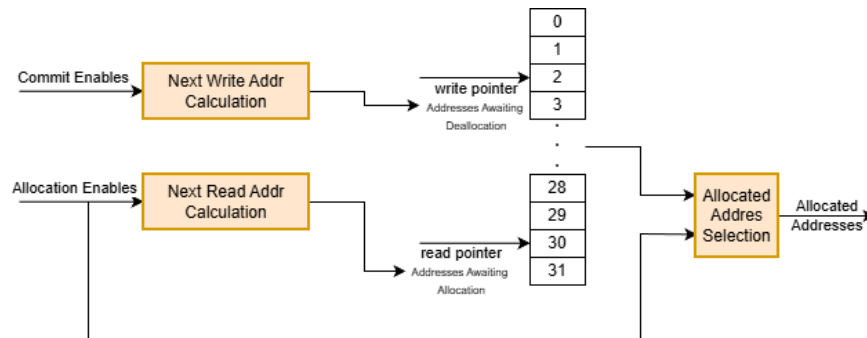


Figure 3.5: Circular buffer-based free list structure.

In this structure shown in Figure 3.5, allocation is performed simply by reading from the read pointer. The first allocated ID is the value of the read pointer; second and third allocations are one and two greater, respectively. The superiority of this approach is that complex logic operations are not required to find free addresses; only the pointer value is read. Additionally, three allocations can be performed simultaneously and all allocations can be reverted in a single cycle by resetting the pointer in case of misprediction.

Deallocation is performed through the write pointer as the reverse of allocation. When an instruction is committed, the old physical register previously used for that instruction's destination register is returned to the free list.

A separate address allocation mechanism exists for memory operations. When load and store instructions are detected, an entry is allocated in the load store queue. This allocation operation occurs in parallel with physical register allocation.

The load store queue address list is also implemented in the same circular buffer structure. This design provides the same advantages as the free register list: all speculative allocations can be instantly cancelled by returning the read pointer in case of misprediction. Up to three queue entries can be allocated per cycle for memory operations, and queue fullness is reflected back to the decode stage as backpressure.

As shown in Table 3.4, the register renaming mechanism eliminates two types of false dependencies. In Write-After-Write (WAW) dependency, two instructions writing to the same architectural register write to different physical registers and can operate independently of each other. In Write-After-Read (WAR) dependency, the read operation is from the old physical register while the write operation is to the new physical register.

Table 3.4: Register renaming operation for an example instruction sequence.

Original Instruction	Operation	Renamed Instruction
ADD x1, x2, x3	$x2 \rightarrow p2, x3 \rightarrow p3, x1 \leftarrow p33$	ADD p33, p2, p3
SUB x4, x1, x5	$x1 \rightarrow p33, x5 \rightarrow p5, x4 \leftarrow p34$	SUB p34, p33, p5
XOR x1, x6, x7	$x6 \rightarrow p6, x7 \rightarrow p7, x1 \leftarrow p35$	XOR p35, p6, p7

3.3.3 Branch speculation support

During out-of-order execution, branch instructions are executed speculatively before their outcomes are determined. This approach allows subsequent instructions to be processed without waiting for branch instruction outcomes. However, special mechanisms are required for the processor to return to a consistent state in case of misprediction.

Several different methods exist for returning the processor to a consistent state when misprediction is detected. In traditional Tomasulo-based processors, waiting until the branch instruction reaches the head of the reorder buffer is required; however, this method causes serious performance losses. Snapshot-based methods store a copy of processor state for each speculative branch and instantly restore this copy in case of misprediction [30].

3.3.3.1 Snapshot mechanism

RAT checkpoint is implemented as a 16-entry circular buffer. When each branch or jump instruction is processed, a snapshot of the current state is saved to this buffer.

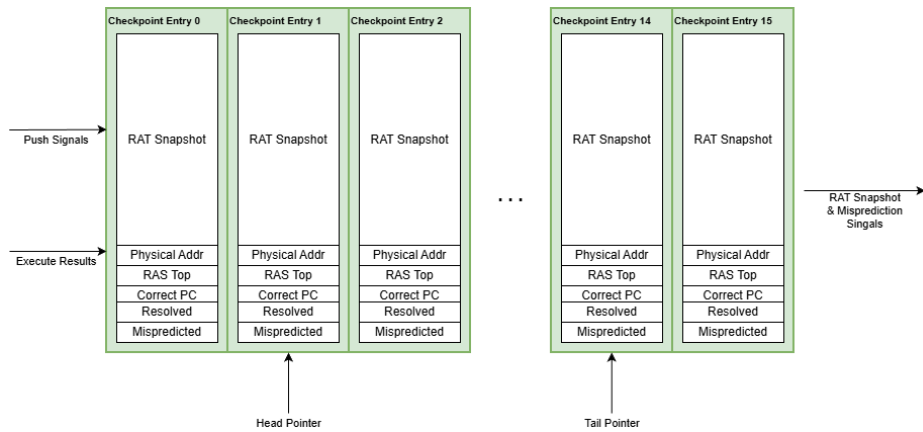


Figure 3.6: RAT checkpoint and snapshot structure.

The information saved in this structure shown in Figure 3.6 includes the physical mapping of 32 architectural registers, the branch instruction's physical register address, the program counter value at prediction time, the indirect jump flag, and the return address stack top pointer.

It has been observed that a smaller buffer may be sufficient during normal operation. However, in secure mode, snapshots are saved not only for branch instructions but for every instruction in order to return to the nearest safe point during unrecoverable errors resulting from multi-bit errors. Therefore, a larger capacity is required. With three parallel push interfaces, up to three snapshots can be taken per clock cycle.

After snapshots are taken, commit operations modify the register alias table. These changes must also be reflected in the snapshots. During each commit operation, the mapping of the relevant architectural register is checked in all active snapshots and updated if necessary.

Even if branches are executed out of order, resolution outputs must be forwarded to other modules in order. RAT checkpoint sets the resolved or mispredicted flags of the relevant entry according to out-of-order resolution results from the execution stage. When the entry at the head pointer is resolved, its result is forwarded to other modules and the pointer is advanced. This ensures priority is always given to the older instruction.

3.3.3.2 Misprediction recovery

When misprediction is detected, the processor is returned to a consistent state using information from the snapshot. The recovery operation consists of several steps:

1. The register alias table is restored from the snapshot.
2. The free register list pointer is set to one greater than the mispredicted branch's physical register address.
3. The load store queue pointer is set according to the value from the LSQ.
4. The correct program counter value is forwarded to the instruction fetch stage.
5. The return address stack top pointer is restored.
6. If GShare is used, information such as the global history table, program counter value, and prediction made is also sent for predictor update. This information is not needed when using the two-bit predictor.

These operations are completed in a single clock cycle and correct-path instructions can be fetched in the next cycle.

For example, consider the following instruction sequence:

- BEQ x1, x2, target ; Branch instruction (mispredicted: taken)
- ADD x3, x4, x5 ; Speculative instruction
- LW x9, 0(x10) ; Speculative load

Suppose the branch instruction was predicted as “taken” but is actually “not taken.” As speculative instructions are executed, new physical registers have been allocated for registers x3 and x9, and a queue entry has been allocated for the load instruction.

When the branch is detected as mispredicted, the register alias table is restored to the state at the branch point, free list pointers are reverted, and the program counter is set to the correct address.

3.4 Data Control and Issue Stage

The data control and issue stage has been designed as a modern adaptation of the Tomasulo algorithm [16]. This stage obtains operand values for instructions coming from the decode and rename stage, provides a dynamic waiting mechanism for operands that are not yet ready, and issues ready instructions to functional units. Additionally, broadcasting execution results to all waiting units simultaneously is the responsibility of this stage.

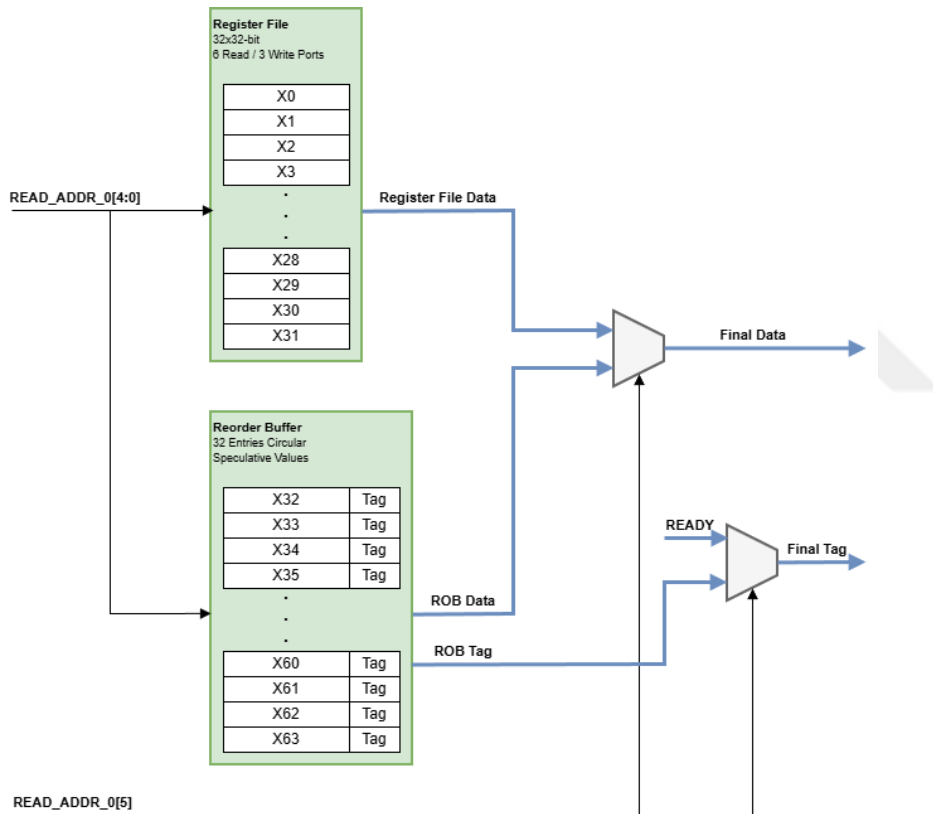
The designed processor has 64 physical registers, and this space is divided into two different regions. Physical registers from 0 to 31 are located in the architectural register file and hold committed values; these values are results whose correctness is guaranteed architecturally. Physical registers from 32 to 63 are located in the reorder buffer and store speculative values that have not yet been committed. The properties of these regions are summarized in Table 3.5.

The 6-bit physical register addresses coming from the decode and rename stage are used to obtain operand values in this stage. The most significant bit of the physical

Table 3.5: Physical register space regions and their properties.

Region	Address Range	MSB	Property
Register File	0–31	0	Committed, always ready
Reorder Buffer	32–63	1	Speculative, tag check required

address determines from which source the operand value will be read. When this bit is zero, the value is read from the register file and is always considered ready. When the bit is one, the value is requested from the reorder buffer and the readiness status of the value is determined according to the tag field of the relevant entry. This dual structure, illustrated in Figure 3.7, enables consistent management of speculative and committed values under a single addressing scheme.

**Figure 3.7:** Source selection based on the most significant bit of physical register address.

The data control and issue stage consists of three fundamental components: reservation stations that handle operand waiting and instruction dispatch functions, the reorder buffer that stores speculative results and provides in-order commit infrastructure, and the register file that holds committed values. These components communicate with each other through the common data bus. In addition to the components in this stage,

the communication with the load-store queue (LSQ) and functional units (FU) is also illustrated in Figure 3.8.

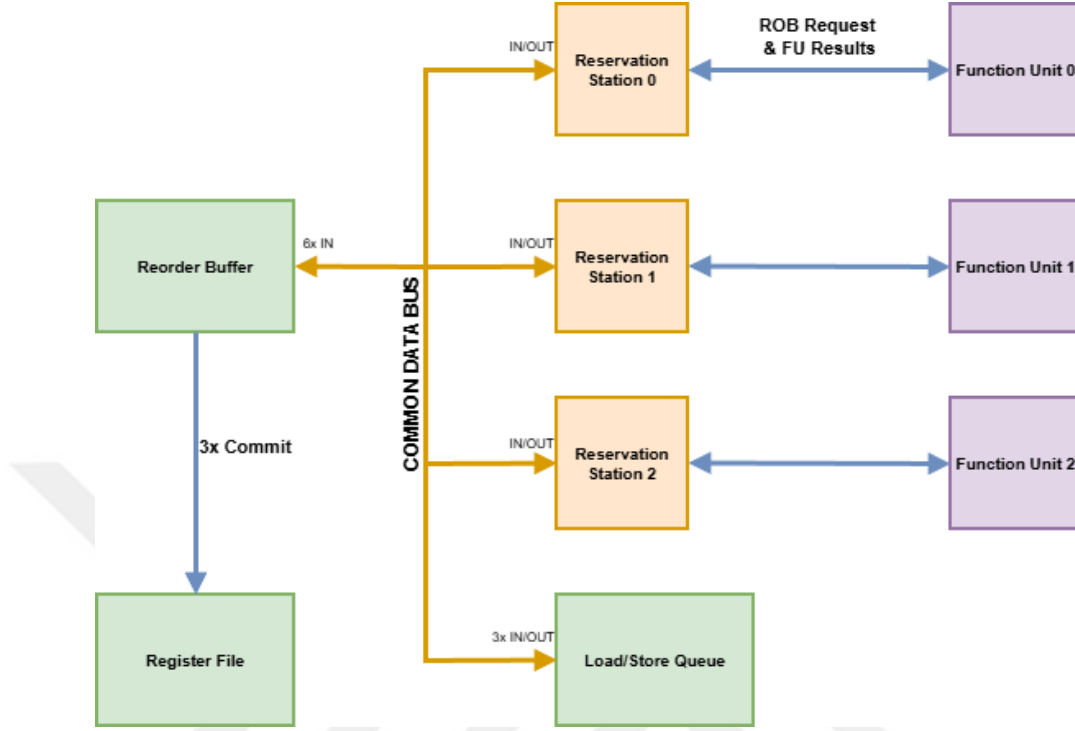


Figure 3.8: Data control and issue stage components and data flow between them.

3.4.1 Reservation station

Reservation stations are one of the critical components of out-of-order execution architecture. These structures dynamically resolve instruction operand dependencies, allowing consumer instructions to progress in the pipeline even if producer instructions have not yet completed [16]. In the designed processor, each reservation station stores the instruction assigned to it and this instruction's operand information, monitors the common data bus until operands are ready, and dispatches the instruction to the functional unit it is connected to when all operands are ready.

Three separate reservation stations exist in the designed system, each connected to a single functional unit. Each reservation station has been designed with single-instruction capacity, as shown in Figure 3.9. The rationale behind this design decision is the reliability requirement: since the primary goal of the designed processor is reliability, critical structures are protected with TMR technique. Since reservation stations contain

wide data fields such as operand data and control information, multi-entry capacity increases area cost to an unacceptable level.

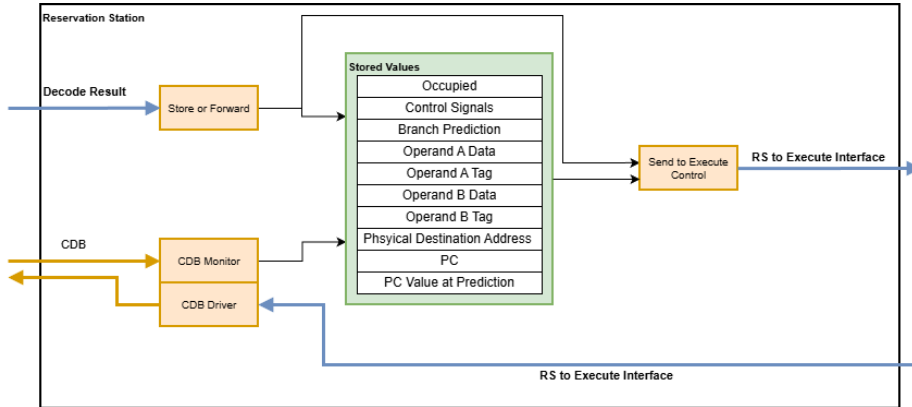


Figure 3.9: Reservation station entry structure and stored information.

3.4.1.1 Tag system and common data bus monitoring

The fundamental innovation of the Tomasulo algorithm is tracking producer-consumer relationships through tags. In the classical approach, a dependent instruction must wait for the instruction that produces its source operand to complete. In the tag-based system, a dependent instruction can be placed in the reservation station even if the producer instruction has not yet executed. This approach can be considered the equivalent of pipeline forwarding in out-of-order execution architectures.

Three-bit tag values are used in the designed system. When the tag value is zero, one, or two, it is understood that the operand is being awaited from the zeroth, first, or second ALU, respectively. When the tag value is three, the operand is awaited from the load store queue. When the tag value is seven, the operand is ready and its value can be used directly. These values are summarized in Table 3.6.

Table 3.6: Operand tag values and their meanings.

Tag Value	Meaning
0 (3'b000)	Awaiting from zeroth ALU
1 (3'b001)	Awaiting from first ALU
2 (3'b010)	Awaiting from second ALU
3 (3'b011)	Awaiting from load store queue
7 (3'b111)	Operand ready

The reservation station continuously monitors the common data bus to resolve its pending operands. The tag value stored for each operand is compared with validity

signals on the common data bus. For example, if an operand's tag is zero and a valid result has been broadcast on the zeroth ALU channel, this result is written to the relevant operand field and the tag is updated to seven.

A special matching mechanism is required for results coming from the load store queue. ALU results are fixed to specific channels; each ALU broadcasts from its own channel. However, the load store queue can produce results from different entries each cycle and can dynamically route these results to one of three channels (for details, see Section 3.6). Therefore, for an operand with tag value three, channel checking alone is insufficient; additionally, the match between the broadcast result's target physical register address and the address the operand is waiting for must also be verified.

3.4.1.2 Instruction issue

For an instruction to be dispatched to a functional unit, both of its operands must be in ready state. For an operand to be considered ready, one of two conditions is sufficient: the operand came from the register file (physical address MSB=0) or it came from the reorder buffer and its tag value is seven. The reservation station supports two different dispatch scenarios.

In the direct dispatch scenario, the instruction's operands coming from the decode stage are already ready. In this case, the instruction is directed to the functional unit in the next clock cycle without being held in the reservation station. This scenario applies to committed values read from the register file or speculative values marked as ready in the reorder buffer.

In the stalled dispatch scenario, at least one of the operands is not yet ready. The instruction is placed in the reservation station and the common data bus monitoring mechanism is activated. As pending operands are resolved from the common data bus, the instruction is dispatched to the functional unit in the clock cycle after both operands become ready.

3.4.1.3 Speculative flush

When branch misprediction is detected in the execution stage, speculative instructions that came after the mispredicted branch must be flushed from the pipeline. In the traditional approach, this flush operation waits for the mispredicted branch to reach the head of the reorder buffer. However, this waiting period causes significant performance losses, especially in deep-pipeline processors.

An eager flush mechanism is used in the designed system. At the moment misprediction is detected, speculative instructions are flushed in a single cycle using information obtained from the RAT checkpoint. The flush decision is based on distance comparison on the circular buffer.

Since the reorder buffer operates as a circular buffer, simple index comparison is insufficient. Each instruction's age is determined by its distance from the reorder buffer's head pointer. Distance calculation is performed with the following equation:

$$distance = \begin{cases} idx - head_ptr & \text{if } idx \geq head_ptr \\ 32 - head_ptr + idx & \text{if } idx < head_ptr \end{cases} \quad (3.6)$$

In this equation, idx represents the instruction's index in the reorder buffer, and $head_ptr$ represents the head pointer. The distance of the mispredicted branch is compared with the distance of the instruction in the reservation station; if the instruction's distance in the reservation station is greater, this instruction came after the mispredicted branch and must be flushed.

An example scenario can be considered to illustrate this mechanism. Assume three reservation stations hold instructions with indices 3, 25, and 8 respectively, and the head pointer is 20. Also assume that the branch instruction at index 5 has been detected as mispredicted. Distance calculations are shown in Table 3.7.

In this example, only the instruction in RS_2 is considered speculative and is flushed. Thanks to this mechanism, misprediction recovery is significantly accelerated compared to traditional methods.

Table 3.7: Speculative flush example scenario ($head_ptr = 20$, branch index = 5).

Unit	Index	Calculation	Distance	Decision
Branch	5	$32 - 20 + 5$	17	(reference)
RS ₀	3	$32 - 20 + 3$	15	Not flushed ($15 < 17$)
RS ₁	25	$25 - 20$	5	Not flushed ($5 < 17$)
RS ₂	8	$32 - 20 + 8$	20	Flushed ($20 > 17$)

3.4.2 Reorder buffer

The reorder buffer is a critical structure that acts as a bridge between out-of-order execution and in-order commit [29]. This structure temporarily stores speculative results from functional units and coordinates the commitment of instructions in program order. The commit operation itself is explained in the writeback stage; this section addresses the roles of the reorder buffer in the data control stage.

The reorder buffer is implemented as a 32 entry circular buffer. The fields contained in each entry are summarized in Table 3.8. The circular buffer structure is managed through head and tail pointers, with allocation operations performed by the tail and commit operations performed by the head.

Table 3.8: Reorder buffer entry fields.

Field Name	Size (bits)	Description
Result Value	32	Execution result
Producer Tag	3	Source from which result is expected
Architectural Register Address	5	Destination register
Executed Flag	1	Has instruction completed
Exception Flag	1	Misprediction/exception
Branch Flag	1	Is branch instruction
Store Flag	1	Is store instruction

3.4.2.1 Allocation and result update

For each instruction coming from the decode and rename stage, an entry is allocated in the reorder buffer. In the three-way superscalar structure, up to three allocation operations can be performed in parallel per clock cycle. During allocation, the entry's producer tag is set, the executed flag is reset, and the destination architectural register address is recorded.

When functional units complete execution, results are conveyed to the reorder buffer over the common data bus. Common data bus channels carry the target physical register address in addition to result data. Since this address directly indicates the location of the entry in the reorder buffer, the update operation is performed with position-based addressing. The content-based addressing approach used in traditional designs requires parallel comparison across all entries for tag matching, and this situation means 32 comparator circuits. By directly using the physical address in the designed system, this complex structure is avoided and area cost is significantly reduced. When the matching entry is found, the result value is written to the relevant field, the tag is updated to seven, and the executed flag is activated.

While operand reading is being performed for newly arriving instructions, the reorder buffer may simultaneously be receiving updates from the common data bus. A forwarding mechanism is used to handle this situation. If an update is coming from the common data bus for the address being read in the same cycle, the current value from the common data bus is returned instead of the old value in the buffer. This mechanism prevents data inconsistency and eliminates unnecessary wait cycles.

Recovery of the reorder buffer in case of misprediction is performed with a quite simple mechanism. The tail pointer is moved to the position after the mispredicted branch instruction. This operation logically invalidates all speculatively allocated entries. There is no need to separately clear the contents of entries because these positions are overwritten when new instructions are allocated.

3.4.2.2 Store permission mechanism

Management of store operations in a speculative execution environment presents a more complex problem compared to load operations. When a speculative load operation is cancelled due to misprediction, the read value can simply be ignored; the problem is resolved by not writing this value to the register file at the commit stage. However, when a speculative store operation writes to memory, reversing this write operation requires much more complex mechanisms.

To avoid this complexity, a store permission mechanism is used in the designed system. Store instructions cannot perform memory write operations until they reach one of the head three positions of the reorder buffer. For a store instruction to be written to memory, three conditions must be satisfied: the instruction must be in one of the head three positions, address calculation must be completed, and there must be no unresolved branch instruction ahead of it.

Thanks to this mechanism, only store operations guaranteed to commit are written to memory. In case of misprediction, speculative store operations that have not yet received permission are cleared from the reorder buffer and memory consistency is preserved.

3.4.3 Common data bus

The common data bus is the communication infrastructure that implements the result broadcasting mechanism of the Tomasulo algorithm. This structure distributes results from functional units to all waiting units simultaneously.

6 common data bus channels exist in the designed system. The first three channels are dedicated to each of the three ALUs; the zeroth ALU broadcasts its results from channel zero, the first ALU from channel one, and the second ALU from channel two. The next three channels are used by the load store queue. The reason for allocating three channels to the load store queue is that three separate memory ports exist in the designed system; each memory port can independently produce results and these results are broadcast from separate channels.

Each common data bus channel carries a standard set of signals. These signals are listed in Table 3.9.

Table 3.9: Common data bus channel signals.

Signal Name	Size (bits)	Description
Valid	1	Indicates result is valid
Result Data	32	Execution result value
Target Physical Address	6	Target physical register address
Misprediction Flag	1	Branch misprediction detected
Branch Flag	1	Result belongs to branch instruction

All signals are conveyed in parallel to reservation stations, the reorder buffer, and the load store queue, as illustrated in Figure 3.10.

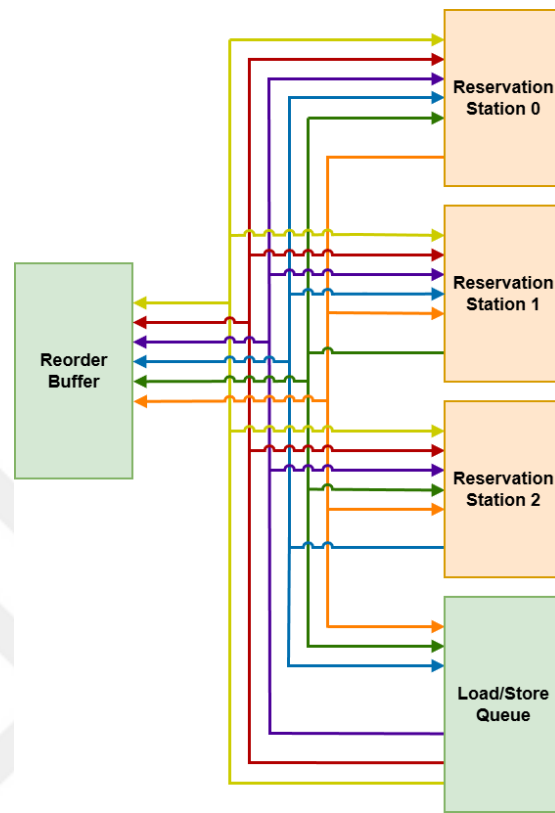


Figure 3.10: Common data bus broadcast network and component connections.

The broadcast mechanism ensures that results reach all relevant units in a single cycle. Reservation stations resolve their pending operands, the reorder buffer updates relevant entries, and the load store queue receives address calculation results and store data. This simultaneous distribution forms the foundation of the Tomasulo algorithm's dynamic resolution of data dependencies.

3.4.4 Register file

The register file is the structure that permanently stores committed instruction results. This structure, corresponding to the 32 architectural registers defined by the RISC-V architecture, contains 32 32-bit registers.

To support the three-way superscalar architecture, the register file has a multi-port structure. Six read ports enable parallel reading of 2 source operands for each of 3

instructions per clock cycle. Three write ports allow simultaneous write operations at the commit stage.

Read operations are performed combinationaly and data is returned in the same clock cycle. Write operations are performed on the rising edge of the clock signal. When both reading and writing to an address occur in the same cycle, bypass logic is activated and enables the new value being written to be read. This mechanism is applied for all three write ports, with highest priority given to the value of the instruction that came last in program order.

In the RISC-V architecture, the zeroth register has a special position and always carries the value zero. To ensure this behavior, the zeroth physical register is forced to zero every clock cycle. Even if any instruction attempts to write to this register, the value does not change and the architectural requirement is guaranteed at the hardware level.

3.5 Execute Stage

The execute stage is the pipeline stage where actual computational operations of instructions are performed. This stage receives ready instructions from reservation stations, performs arithmetic and logical operations, evaluates branch conditions, and performs misprediction detection. Computed results are sent to the issuing reservation station, which then broadcasts them through the common data bus.

In the designed three-way superscalar structure, three independent functional units operate in parallel, as shown in Figure 3.11. Each functional unit has full arithmetic logic unit (ALU), shifter, and branch controller capacity and can execute one instruction per clock cycle. Since all operations for the RV32I base instruction set are completed in a single clock cycle, functional units can accept new instructions every cycle.

3.5.1 Functional unit structure

Each functional unit consists of an ALU, a shifter, and a branch controller. These three subunits operate in parallel and their results are selected through a multiplexer according to the function selection signal. This design shortens the critical path and makes all RV32I arithmetic, logical, and control flow operations completable in a single cycle.

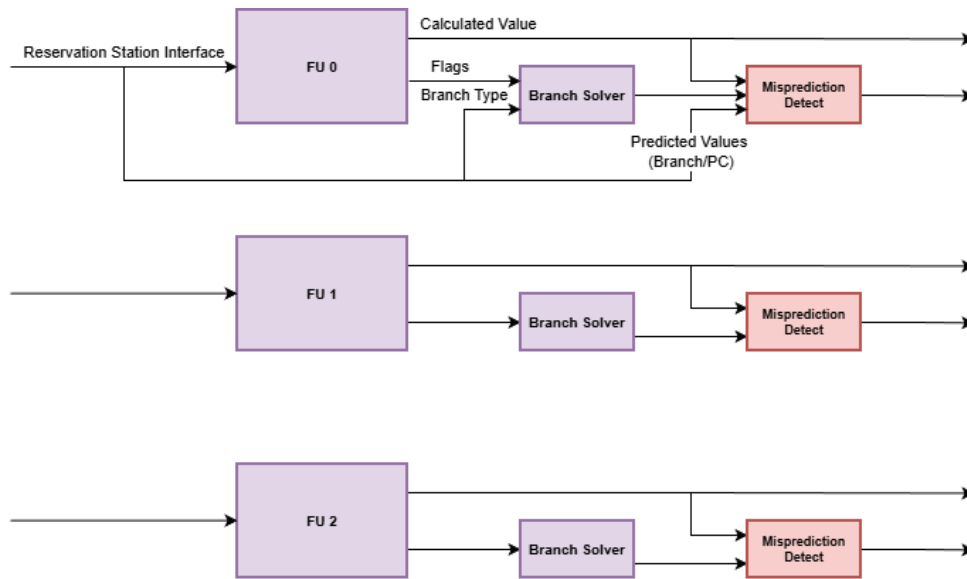


Figure 3.11: Execute stage general structure and component connections.

The ALU consists of two subunits: arithmetic and logical. The arithmetic unit supports addition, subtraction, signed comparison, and unsigned comparison operations. The logical unit performs AND, OR, and XOR operations. Results of both subunits are calculated in parallel and selection is made according to the most significant bit of the function selection signal. Supported operations and selection values are shown in Table 3.10.

Table 3.10: ALU function selection signals and supported operations.

Selection Value	Operation	Description
0 (3'b000)	ADD	Addition
1 (3'b001)	SUB	Subtraction
2 (3'b010)	SLT	Signed less-than comparison
3 (3'b011)	SLTU	Unsigned less-than comparison
4 (3'b100)	XOR	Exclusive OR
5 (3'b101)	OR	OR
6 (3'b110)	AND	AND

In addition to computation results, the ALU also produces status flags used in branch condition evaluation. The zero flag is active when the result is zero, the negative flag is active when the most significant bit of the result is one, the carry flag is active when a carry occurs in addition or subtraction operations, and the overflow flag is active in signed arithmetic overflow conditions.

The shifter is designed in barrel shifter architecture and performs variable-amount shift operations in a single clock cycle. The logical left shift operation shifts the first operand left by the specified amount and fills with zeros from the right. The logical right shift operation shifts right and fills with zeros from the left. The arithmetic right shift operation preserves the sign bit while shifting right and fills with the sign bit from the left. The shift amount is processed as five bits and shifting from zero to thirty-one bits can be performed.

The branch controller evaluates the branch condition using status flags from the ALU and the branch selection signal. The operating principle of this component is detailed in the next section.

How all instructions in the RV32I base instruction set are processed in the functional unit is shown in Table 3.11. This table summarizes which subunit each instruction uses and the operation performed.

Table 3.11: RV32I instruction functional unit usage.

Instruction	ALU Operation	Function	Block
ADD/ADDI/LUI	$G = A + B$	ADD	ALU
LB/LH/LW/LBU/LHU	$G = A + B$	ADD	ALU
SB/SH/SW	$G = A + B$	ADD	ALU
JAL/JALR/AUIPC	$G = A + B$	ADD	ALU
SUB	$G = A - B$	SUB	ALU
BEQ/BNE/BLT/BGE	$G = A - B$	SUB	ALU
SLT/SLTI	$G = \text{sign}(A - B)$	SLT	ALU
SLTU/SLTIU	$G = \text{usign}(A) < \text{usign}(B)$	SLTU	ALU
BLTU/BGEU	$G = \text{usign}(A) < \text{usign}(B)$	SLTU	ALU
XOR/XORI	$G = A \oplus B$	XOR	ALU
OR/ORI	$G = A B$	OR	ALU
AND/ANDI	$G = A \& B$	AND	ALU
SLL/SLLI	$G = A \ll B[4:0]$	SLL	Shifter
SRL/SRLI	$G = A \gg B[4:0]$	SRL	Shifter
SRA/SRAI	$G = A \ggg B[4:0]$	SRA	Shifter

3.5.2 Branch resolution

Branch resolution encompasses the calculation of actual results of branch instructions and comparison with predictions. This process is the critical point where the correctness of speculative execution is verified.

The branch controller determines the branch decision based on a three-bit selection signal and ALU status flags. Equality branch (BEQ) branches when the zero flag is active, while inequality branch (BNE) branches when the zero flag is inactive. Less-than branches (BLT/BLTU) branch when the negative flag is active, and greater-than-or-equal branches (BGE/BGEU) branch when the negative flag is inactive. For unconditional jump (JAL) and indirect jump (JALR) instructions, the branch condition is always evaluated as taken. These conditions are summarized in Table 3.12.

Table 3.12: Branch selection signals and branch decisions.

Instruction	sel[2]	sel[1]	sel[0]	Decision
No Branch	0	0	X	0
BEQ	0	1	0	Z
BNE	0	1	1	$\sim Z$
BLT/BLTU	1	0	0	N
BGE/BGEU	1	0	1	$\sim N$
JAL	1	1	0	1
JALR	1	1	1	1

It is noteworthy that signed (BLT/BGE) and unsigned (BLTU/BGEU) branch instructions share the same selection values. The reason for this is that the ALU already produces the negative and zero flags considering unsigned arithmetic in unsigned comparison operations (SLTU). Therefore, the branch controller does not need to separately check carry or overflow flags.

Misprediction detection is performed by comparing the actual branch result with the predicted result. For conditional branch instructions, the actual direction (taken or not taken) is compared with the predicted direction. For indirect jump instructions (JALR), the calculated target address is compared with the predicted target address. The reason for special handling of JALR instructions is that these instructions require not only direction but also target address prediction. The calculated target address is the four-byte aligned sum of the first source operand and the immediate value.

When misprediction is detected, as illustrated in Figure 3.12, the correct program counter value must be calculated. For indirect jump instructions, the correct program counter is the calculated target address. For conditional branch instructions, the situation is handled in two ways: if the prediction was taken but actually not taken, the correct

address is the current program counter plus four; if the prediction was not taken but actually taken, the correct address is the current program counter plus the branch offset.

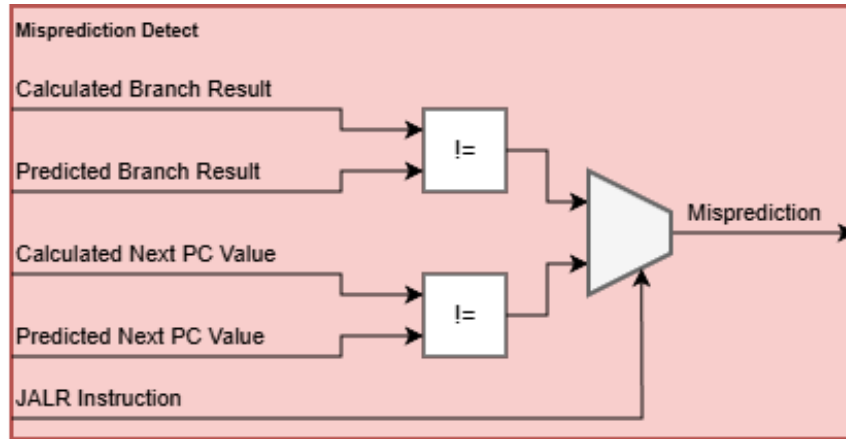


Figure 3.12: Misprediction detection mechanism.

3.5.3 Result selection

The execute stage produces different results for different instruction types, and these results are broadcast over the common data bus.

For arithmetic and logical instructions, the result is the ALU computation result. For load and store instructions, the result is the memory address calculation; this value is forwarded to the load store queue. For unconditional jump (JAL) and indirect jump (JALR) instructions, the result is the return address and is calculated as the current program counter plus four. For conditional branch instructions, the result always returns as zero, but in case of misprediction detection, the misprediction flag is activated and the correct address is forwarded to previous stages.

Result selection is performed through a multiplexer based on control signals. The selected result is written to the common data bus along with the target physical register address and other control information. The structure of the common data bus and the broadcast mechanism are detailed in Section 3.4.3.

3.6 Memory Stage

The memory stage is the pipeline stage where load and store operations are managed. Maintaining memory operation consistency and correctly managing dependencies in

an out-of-order execution environment is critically important. Store operations must appear in program order, a load may depend on a previous store, and speculative store operations must be able to be rolled back in case of misprediction.

In the performance analyses presented in Chapter 5, memory operations were observed to be the mechanism that most reduces IPC value after misprediction. For this reason, unnecessary complexity was avoided when designing the memory stage, but performance optimizations were not forgone. In the designed system, the Load Store Queue (LSQ) enables out-of-order execution of memory operations while maintaining consistency.

3.6.1 Load store queue structure

The LSQ is implemented as a 32 entry circular buffer. In a three-way superscalar processor, three memory instructions can be allocated per cycle. For memory operations of these instructions to be performed at the same throughput, the LSQ operates with three independent memory ports. Otherwise, the number of memory lanes becomes a bottleneck limiting IPC.

A head pointer exists corresponding to each memory port. These three head pointers monitor different entries in the LSQ simultaneously and each determines which entry will be processed by its memory port. The structure of the LSQ is shown in Figure 3.13.

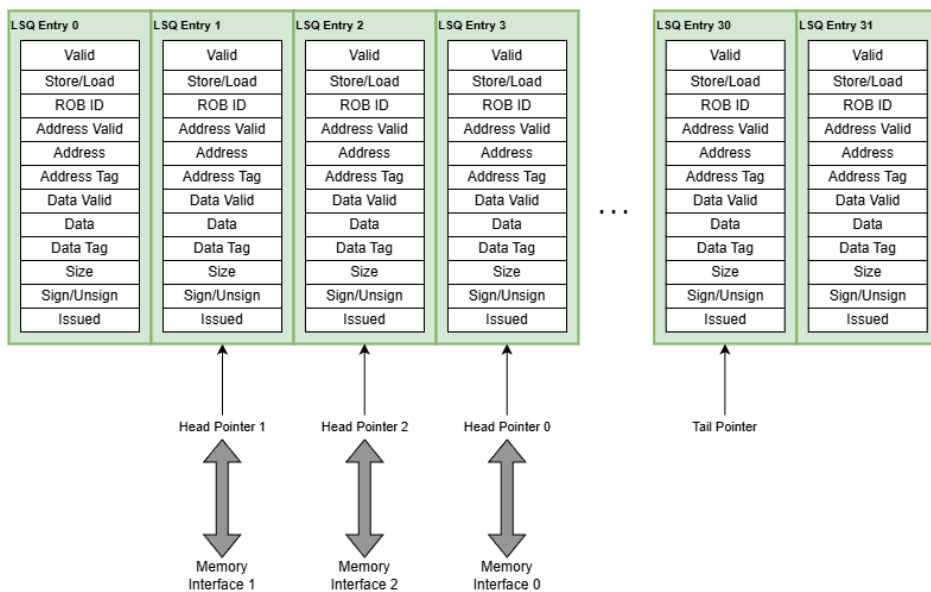


Figure 3.13: Load store queue circular buffer structure.

Each LSQ entry contains all information required for execution of the memory operation. These fields are listed in Table 3.13.

Table 3.13: Load store queue entry fields.

Field Name	Size (bits)	Description
Valid	1	Is entry valid
Type	1	Store (1) or Load (0)
Physical Register	6	Target ROB ID
Address Valid	1	Is address calculated
Address	32	Calculated memory address
Address Tag	3	Tag of address producer
Data Valid	1	Is store data ready
Data	32	Store data
Data Tag	3	Tag of data producer
Size	2	Byte (00), Halfword (01), Word (10)
Signed Load	1	Signed (1) or unsigned (0)
Sent to Memory	1	Has memory request been made
Memory Completed	1	Has memory response been received

When a memory operation completes, the relevant head pointer is updated. Each head pointer is shifted to the oldest incomplete entry, also considering the positions of the other two head pointers. For example, if three head pointers are pointing to indices 5, 7, and 9 respectively, when the operation in entry 5 completes, this head pointer is moved to entry 10 (one after the newest pointer). This mechanism guarantees that head pointers always point to different entries and will give priority to all entries according to allocation order. However, if an operation with priority does not block other operations, operations can run out of order despite being allocated later; this situation accelerates memory operations.

3.6.2 LSQ operations

LSQ entries are allocated for load and store instructions coming from the decode stage. During allocation, the validity flag is activated, operation type and physical register address are recorded, address validity is reset, and size and signed load information are stored. For store instructions, data may or may not be ready; if not ready, the data tag is recorded and set to wait from the common data bus. Since address information will be

calculated at the earliest in the next clock cycle, address information is also expected to come from the common data bus.

When the execute stage calculates the address, the result is broadcast over the common data bus. The LSQ monitors all 6 common data bus channels. Address update is performed in parallel across all LSQ entries: for each entry, the target register address on the common data bus is compared with the entry's physical register address; in case of a match, address validity is activated and the calculated address is stored.

For store operations, data update is also performed in the same parallel structure. Data tags of all store entries are compared in parallel with common data bus channels; in case of a match, data validity is activated and data is stored.

When an operation pointed to by any head pointer is ready and there is no incomplete store operation ahead that would block it, the operation can be sent to the relevant memory port. For load operations, the entry must be valid and address must be calculated. For store operations, additionally data must be ready and permission must have been received from the reorder buffer.

The store permission mechanism is explained in Section 3.4.2.2. Briefly, the ROB indicates that a store operation with a particular physical register address is safe. The LSQ compares this address with its own entries and can send the matching store operation to memory. This mechanism allows only store operations guaranteed to commit to modify memory.

3.6.3 Store-to-load forwarding

The LSQ can forward store data to a load without going to memory. This mechanism significantly improves performance by bypassing memory latency.

The forwarding decision is made based on the relationship between operations at the three head pointers. A load operation can receive data from a store operation that comes before it in program order and writes to the same address. For forwarding, four conditions must be satisfied: the load must come after the store (checked with age comparison), the store's address and data must be ready, addresses must match, and the store size must be equal to or greater than the load size.

The necessity of size checking arises from the requirement that the store must contain all bytes that the load needs. For example, forwarding from a 4-byte store to a one-byte load is possible. However, forwarding from a 1-byte store to a 4-byte load is not possible because the store does not contain all bytes that the load needs.

If forwarding is not possible but a potential dependency exists, the load is stalled. Stall conditions and forwarding rules are summarized in Table 3.14.

Table 3.14: Store-to-load forwarding and stall rules.

Address Match	Size Sufficient	Data Ready	Decision
None	-	-	Read from memory
Yes	Yes	Yes	Forward
Yes	Yes	No	Stall
Yes	No	-	Stall

Performing forwarding checks across all LSQ entries would incur high area cost because it would require many comparators. For this reason, forwarding checks are performed only between entries pointed to by the three head pointers, as shown in Figure 3.14. Each load operation at a head checks the store operations at the other two heads. Age comparison is done with distance values calculated from LSQ pointers; the newest store with a matching address is prioritized and selected as the forwarding source in case of address match. This approach keeps the complexity of forwarding logic at an acceptable level.

3.6.4 Eager misprediction flush

In case of misprediction, speculative load and store entries are flushed. The flush mechanism in the LSQ works similarly to the mechanism in reservation stations but involves additional complexity.

Each LSQ entry's physical register field also carries the corresponding reorder buffer index. Using this index, the distance from the reorder buffer's head pointer is calculated for each entry. Distance calculation is performed considering the circular buffer structure.

The misprediction signal and the distance of the mispredicted instruction are received from the reorder buffer. The distance of each LSQ entry is compared with the distance of

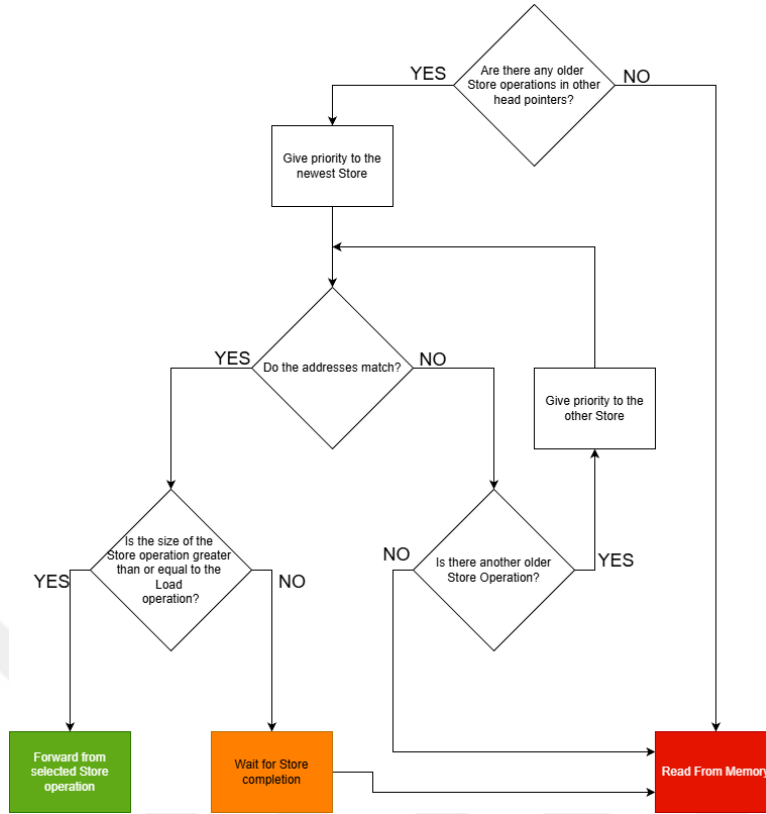


Figure 3.14: Store-to-load forwarding logic.

the mispredicted instruction. Entries with greater distance came after the mispredicted branch and must be flushed.

The flush operation is performed in parallel across all entries in the same cycle. A comparator circuit exists for each entry and the result is fed to a priority encoder. The priority encoder finds the index of the first (oldest) entry that must be flushed. The LSQ tail pointer is updated to this index, logically invalidating speculative entries. The synthesis results presented in Chapter 5 show that this parallel comparison structure forms the critical path.

All three head pointers must also be updated. If a head pointer remains in the flushed region, this head pointer is also moved to the new LSQ tail position. This check is performed considering the wrap-around situation of the circular buffer.

3.6.5 Memory interface

The LSQ supports three independent memory ports, as illustrated in Figure 3.15. Each port can independently send memory requests and receive responses.

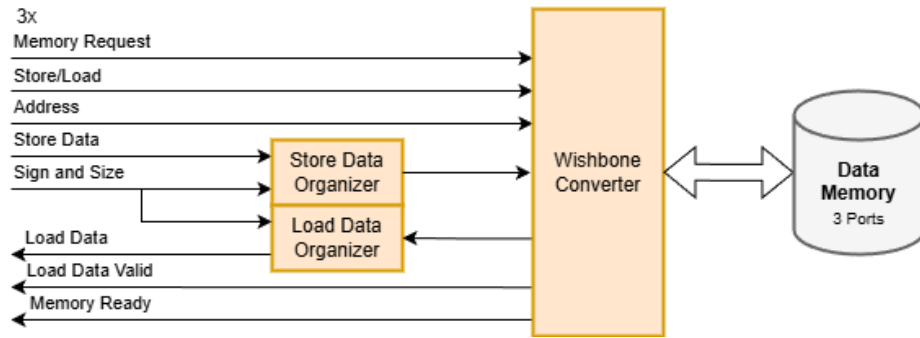


Figure 3.15: LSQ and memory interface.

Request signals for each port include validity signal, write/read selection, memory address, write data, and byte enable signals. Response signals include ready signal, response validity, and read data.

The byte enable signal specifies which bytes will be affected for different-sized operations. For byte operations, a single bit positioned according to address, for halfword operations 2 bits, and for word operations 4 bits are enabled.

Load data is processed according to size and signed load information from the memory response. For byte operations, the relevant byte is extracted from the 4-byte data coming from memory and filling with the seventh bit is performed for signed loads. For halfword operations, two bytes are extracted and filling with the fifteenth bit is applied. For word operations, data is used directly.

3.7 Writeback Stage

The writeback stage is the final stage of out-of-order execution and ensures that instructions are committed in program order. During the commit operation, the results of instructions at the head of the ROB are reflected to architectural state and relevant resources are made available for reuse.

3.7.1 Commit operation

Commit is the processing of instructions at the head of the ROB in program order. Each clock cycle, up to 3 instructions at the head of the ROB are evaluated for commit.

For an instruction to be committed, it must have been executed. This condition is verified by checking the executed flag in the ROB entry. For load operations, the data must have been received from memory; for store operations, dispatch permission to the LSQ must have been granted.

In 3-way commit, there is a chained dependency, as illustrated in Figure 3.16. For the first instruction to be committed, it is sufficient that it has been executed. The second instruction is evaluated only if the first instruction will be committed. The third instruction is evaluated only if both the first and second instructions will be committed. This chained structure guarantees preservation of program order.

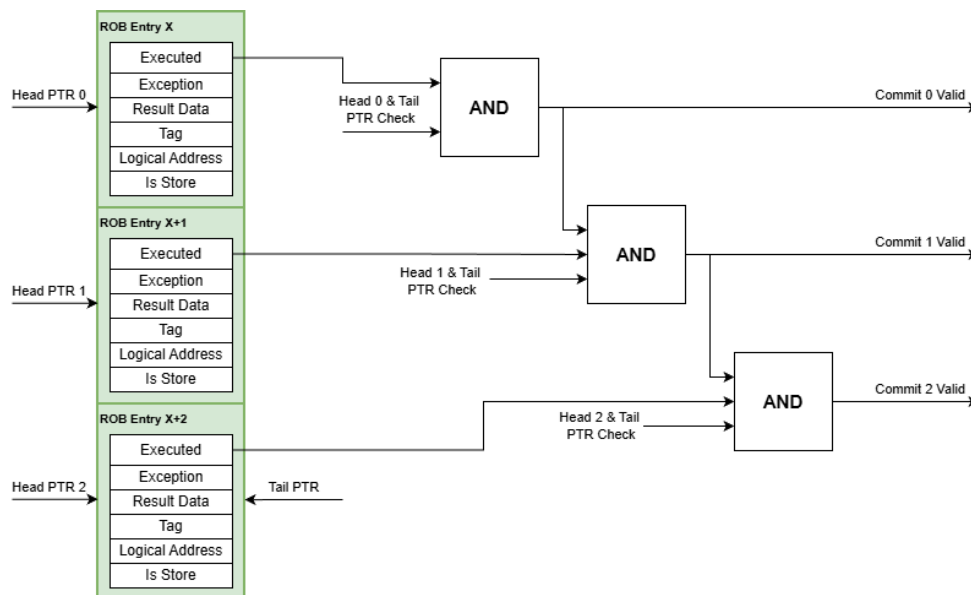


Figure 3.16: In-order commit process.

Each cycle, the number of committed instructions is calculated and the ROB head pointer is advanced by this amount. For example, if all 3 instructions are committed, the head pointer is incremented by 3.

3.7.2 Register alias table update

During commit, the RAT is updated. This update ensures that the mapping is updated to point to the register file instead of the reorder buffer.

The update condition is important: the update is performed if the current mapping of the architectural register in the RAT points to the ROB index of the committed instruction.

This check ensures that if there is a newer instruction writing to the same architectural register, the older instruction does not perform the update.

For example, consider 2 instructions writing to architectural register x5: the first instruction is allocated to ROB[3], and the second instruction to ROB[7]. When the first instruction is committed, the RAT[x5] value is checked. If RAT[x5] still points to 3, x5 is updated to its original position in the register file (0,5). However, if RAT[x5] now points to 7 (since the second instruction was allocated), the update is not performed.

This mechanism ensures correct management of successive writes to the same architectural register.

3.7.3 Free list update

During commit, used ROB entry addresses are returned to the free list. The free list is maintained in a circular buffer structure and provides available ROB addresses for new instructions.

Each commit signal is connected to a write port of the free list. When commit is valid, the relevant ROB index is added to the free list. With 3 parallel write ports, 3 addresses can be freed each cycle. There is no actual write operation here; as a result of this operation, only the pointer value of the circular buffer holding the free list is incremented.

In case of misprediction, the free list pointer is also updated. It is set to the mispredicted branch's ROB index plus one, automatically reclaiming speculatively allocated addresses.

3.7.4 Memory operations commit

For memory operations, commit includes store permission and LSQ address release.

The store permission mechanism is detailed in Section 3.6.2. The ROB produces a permission signal and the permitted store address for the store operation at its head. The LSQ uses this information to write only store operations guaranteed to commit to memory. Before granting store permission, the branch status of previous instructions

is checked; if there is a branch instruction before the store operation, this branch instruction must have been executed and not mispredicted.

LSQ address release is performed during commit. Although the LSQ works out of order, release is done in order. The reason for this is that LSQ address allocation is done with a circular buffer; since allocation is in order, release must also be in order. The LSQ quickly resolves data dependencies by working out of order, while the release operation is left to the ROB to ensure ordering. When a memory instruction is committed, the ROB produces an LSQ commit signal. This signal is conveyed both to the LSQ and to the LSQ free address circular buffer within the RAT.

3.7.5 Pipeline flush

The in-order commit structure allows for future addition of exception and interrupt handling. In the current design, only branch misprediction detection and pipeline flush operations are implemented.

Pipeline flush is performed when misprediction is detected, as shown in Figure 3.17. Flush signals are sent simultaneously to all pipeline stages. Speculative entries are invalidated by setting the ROB's tail pointer to one after the mispredicted branch. All speculative entries in reservation stations and LSQ are flushed.

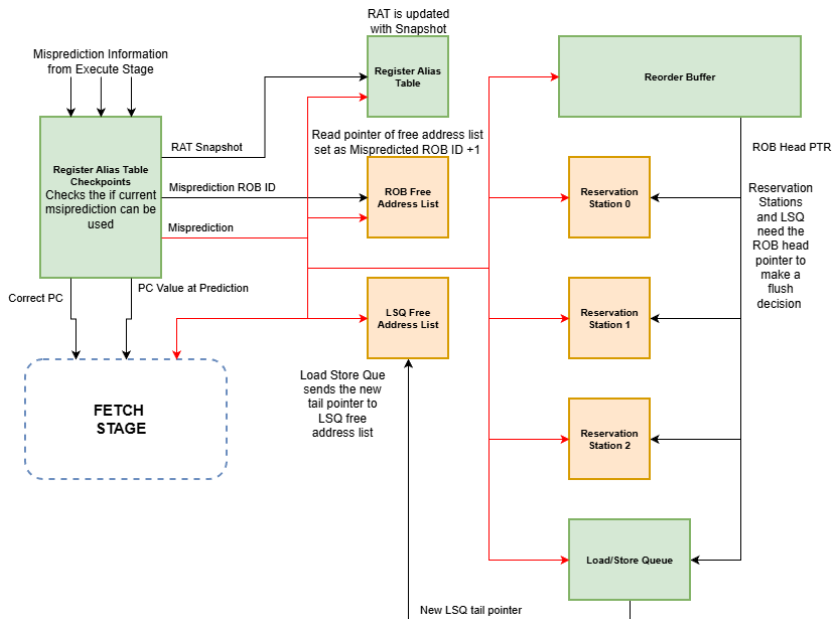


Figure 3.17: Pipeline flush mechanism.

RAT state is restored by loading the correct register mapping snapshot from RAT checkpoint. This mechanism is explained in Section 3.3.3. The program counter is set to the correct address and a redirect signal is sent to the instruction fetch stage.

3.7.6 Commit signals

ROB commit output signals provide state updates to other modules. Signals produced in parallel for 3 instructions each cycle are listed in Table 3.15.

Table 3.15: Commit output signals.

Signal	Target	Description
commit_valid	RAT	Instruction committed
commit_addr	RAT	Architectural register address
commit_rob_idx	RAT	Committed ROB index
lsq_commit_valid	LSQ, RAT	LSQ commit signal
commit_is_branch	Branch predictor	Prediction update



4. ON-DEMAND REDUNDANCY TECHNIQUE

This chapter discusses in detail the on-demand redundancy technique applied to the designed superscalar RISC-V processor. First, the importance of fault tolerance and limitations of existing methods are explained, followed by the contributions of the designed processor in this field. This work constitutes the implementation of the research direction proposed in a previously published literature survey [7].

4.1 Motivation

The reliability of modern electronic systems, especially in critical applications such as space, aviation, and automotive, is directly related to minimizing the effects of faults. In these fields, faults originating from environmental factors can threaten system stability. In such applications, fault tolerance has become an indispensable requirement for ensuring both system reliability and user safety [2].

As explained in Section 2.3, transient faults (SEU, MBU) and permanent faults require different protection methods. Spatial redundancy (TMR, DCLS/TCLS), temporal redundancy, and information redundancy (ECC) are fundamental approaches developed for this purpose. In this chapter, the limitations of existing methods are analyzed and how the designed processor provides solutions to these limitations is explained.

4.2 Limitations of Existing Methods

4.2.1 Disadvantages of dual and triple-core lockstep

Core-level lockstep (DCLS/TCLS) methods have various limitations in practical applications:

- Late fault detection: DCLS and TCLS can only detect faults at the core output. This situation can cause the fault to propagate within the pipeline and affect multiple

stages before detection. As mentioned in Dörflinger's work, detectors located at deeper levels of the pipeline can catch faults much earlier [4].

- High recovery latency and fault penalty: Core-level comparison waits for all pipeline stages to complete. This situation increases recovery latency and fault penalty.
- Performance loss: The TCLS method requires 3 cores to execute the same instruction, so only one-third of the potential processing power can be utilized. This resource waste continues even under normal conditions.
- Fixed mode operation: Traditional DCLS/TCLS implementations cannot switch between performance and security.

4.2.2 Advantages of pipeline-level protection

As demonstrated in the DuckCore architecture developed by Li et al., pipeline-level fault detection offers significant advantages [20]:

- Early detection: Faults can be detected at the stage where they occur and their propagation can be prevented.
- Low latency: Checks performed at each stage do not require waiting for core output.
- Targeted correction: Since which stage the fault occurred in is known, the correction mechanism can work more effectively.

In Dörflinger's work, it was shown that Deep Level Detectors reduced the average detection latency of transient logic faults by up to 79% [4]. This result proves the practical effectiveness of pipeline-level protection.

4.2.3 Need for on-demand modular redundancy

The on-demand modular redundancy (ODMR) approach developed by Rogenmoser et al. provides a solution to the above limitations [5]. In this approach, the system can operate:

- In full redundancy (TMR) mode during critical tasks,

- In independent operation mode under normal conditions.

The Trikarenos project is a concrete implementation of this flexible structure. This microcontroller, manufactured with TSMC 28nm technology, achieves energy savings or performance gains by switching to full redundancy mode only during critical tasks [21].

4.3 Triple Modular Redundancy Suitability of Three-Way Superscalar Design

One of the most fundamental features of the designed processor is its superscalar architecture with 3 parallel execution channels. This design decision was made considering both performance and fault tolerance goals.

Reasons for selecting three-way design:

1. Natural compatibility with TMR: Three channels directly correspond to the TMR structure. When each channel is considered as a module, the existing hardware infrastructure can be reused for redundancy purposes.
2. Performance and cost balance: In normal operation, 3 instructions can be processed in parallel (IPC=3); in secure mode, the same hardware is used for fault masking.
3. Minimal overhead: No additional hardware is required except for voter circuits needed for TMR. The existing superscalar infrastructure is used for both performance and security purposes.

The designed processor supports 3 different operating modes as shown in Table 4.1.

Table 4.1: Processor operating modes.

Mode	Channels	TMR	IPC	Description
Superscalar	3 independent	Off	3	Three channels process different instructions in parallel
Secure	3 locked	On	1	Three channels process the same instruction, results are voted
Low Power	1 active	Off	1	Other channels are disabled for energy savings

4.4 Contributions of the Design

In the existing literature, designs implementing intra-core protection at pipeline level like DuckCore [20], designs implementing ODMR at core level like Trikarenos [21], and high-performance open-source RISC-V cores like BOOM [23] exist separately. However, combining these approaches has not been previously realized.

The processor designed in this thesis presents the following novel contributions:

4.4.1 Combination of pipeline-level triple modular redundancy with on-demand modular redundancy

The designed processor combines both pipeline-level fault detection and ODMR support in a single architecture. This combination provides intra-core protection with mode-switching capability in a single design.

4.4.2 Low-latency mode transition

Pipeline-level TMR implementation enables minimization of inter-mode transition latency. For transition:

- Pipeline flushing is not required.
- Core state synchronization is not required.
- Only activation of voter circuits is sufficient.

This feature enables fast mode changes according to application needs.

4.4.3 Early fault detection

Unlike core-level DCLS/TCLS, the designed processor can detect faults at every stage of the pipeline. This enables:

- Prevention of fault propagation.

- Minimization of detection latency.
- Shortening of recovery time.

4.4.4 Adaptive mode transition infrastructure

The designed architecture is planned to support automatic mode transition in the future. Although this feature has not yet been implemented, the infrastructure is designed to support the following scenario:

1. While the processor is operating in superscalar mode, some voters monitor mismatches.
2. When mismatches above a certain threshold are detected, the system can automatically switch to secure mode.
3. When radiation risk decreases, the system can return to superscalar mode.

This approach has the potential to provide adaptive fault tolerance without requiring external intervention.

4.5 Triple Modular Redundancy Implementation

4.5.1 Majority voter

The majority voter, the fundamental component of TMR, selects the value that at least 2 of 3 inputs produce as output [6]. Mathematically:

$$Output = (A \cdot B) + (B \cdot C) + (A \cdot C). \quad (4.1)$$

In this equation, A , B , and C represent the outputs of 3 modules, the multiplication operation represents logical AND, and the addition operation represents logical OR.

The voter module used in the design has the following features:

- Parametric data width: Can be used with different widths from 1 bit to 32 bits.

- Bypass mode: When secure mode is off, the primary input is directly transferred to the output.
- Error reporting: Which input is in the minority is detected and reported.
- Critical error flag: A warning is generated when all 3 inputs are different.

4.5.2 Protected structures

In the design, all pipeline registers are protected with TMR, excluding large memory structures (ROB, LSQ, RAT, and register file data fields). These large structures are assumed to be ECC protected as discussed in Section 4.6.

Three copies are maintained for each protected structure and majority voting is performed for every read operation, as shown in Figure 4.1. In write operations, all 3 copies are updated with the same value.

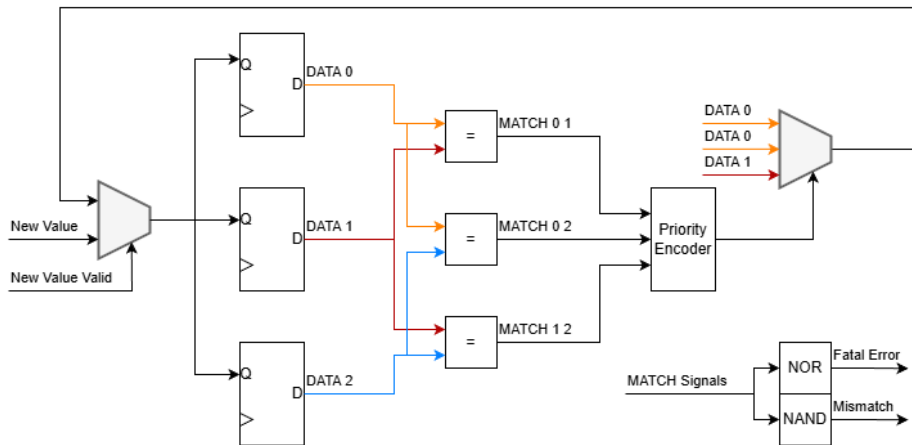


Figure 4.1: TMR register recovery mechanism: comparison and correction with majority voting.

4.6 Error Correction Code Assumption for Large Memory Structures

Applying TMR to large memory structures incurs 200% area cost. ECC provides a much more efficient alternative for these structures.

As Dörflinger pointed out, using ECC instead of full data replicas for memories significantly reduces resource usage [4]. In Annink's work, it was stated that adding ECC reduced achievable frequency by 20% and area overhead was 33.7% [3]. These values are acceptable compared to TMR's 200% area overhead.

In the design, the following large memory structures are assumed to be ECC protected:

- Reorder buffer (ROB) data fields
- Reservation stations (RS) data fields
- Load-store queue (LSQ) data fields
- Register file

It was decided to keep ECC implementation outside the scope of this thesis and is recommended as an important research topic for future work.

4.7 Critical Fault Recovery with Register Alias Table Checkpoint

The RAT checkpoint explained in Section 3.3.3 is designed for fast recovery in case of branch misprediction. However, this structure can also be used for recovery from critical faults in secure mode. This approach constitutes a concrete example of reusing existing hardware infrastructure for fault tolerance purposes.

4.7.1 Dual use of checkpoints

The RAT checkpoint takes a checkpoint for every branch instruction in normal mode. However, in secure mode, faults can originate not only from mispredictions but also from radiation effects such as SEU and MBU.

The RAT checkpoint requires more area compared to other eager misprediction recovery methods. However, the ability to use the same structure for recovery from radiation-induced faults justifies this additional area cost. Both branch speculation and fault tolerance are provided with a single hardware structure.

4.7.2 Secure mode checkpoint strategy

In normal mode, snapshots are taken only for branch and jump instructions. In secure mode, the proposed approach is to take a snapshot for every instruction. Thanks to this strategy:

- When a critical fault is detected in any instruction, it is possible to return to the appropriate state stored in RAT checkpoint.
- Recovery latency is minimized because the most recent snapshot is always available.
- The same mechanism used for misprediction recovery is used.

4.7.3 Critical fault detection mechanism and proposed recovery flow

While the processor is operating in secure mode, the TMR mechanism can mask single-bit faults. However, faults affecting multiple bits or corrupting related signals can cause all 3 copies to produce different values. This situation is classified as a “fatal error” that cannot be corrected by TMR.

In the design, TMR modules distributed across different stages of the pipeline generate a signal when they detect a fatal error. In the current design, this signal enables the error to be reported and serves as a trigger for the proposed recovery mechanism. The proposed recovery flow is designed as follows:

1. Fatal error detection: A TMR module at any stage detects that all 3 copies are different and activates the fatal error signal.
2. Identification of relevant instruction: The physical register address assigned to the faulty instruction is used to access the relevant snapshot in the RAT checkpoint.
3. Restoration from snapshot: The register alias table snapshot and program counter value are retrieved from the RAT checkpoint.
4. Pipeline flush: All speculative instructions processed after the faulty instruction are flushed.
5. Restart: The program counter is set to the faulty instruction’s address and the instruction is re-fetched from memory.

The central controller for autonomous operation of this mechanism has been implemented. The fatal error handler module detects critical faults and initiates automatic recovery using the RAT checkpoint mechanism.

4.8 Summary

This chapter presented the on-demand redundancy technique applied to the designed superscalar RISC-V processor. The main contributions of the design are:

1. Reuse of 3-way superscalar architecture for TMR: The existing hardware infrastructure can be used for both performance and security purposes with minimal overhead.
2. Combination of pipeline-level protection with ODMR: Early fault detection advantages and flexible mode-switching capabilities are combined in a single architecture.
3. Low-latency mode transition: Pipeline-level implementation enables inter-mode transitions to be performed with minimum latency.
4. Critical fault recovery with RAT checkpoint: The RAT checkpoint structure designed for misprediction recovery is also used for recovery from radiation-induced critical faults. Dual functionality is provided with a single hardware structure.
5. Dynamic mode transition infrastructure: Architectural infrastructure supporting automatic fault detection and mode transition in the future has been prepared.

This approach implements the idea of “combining a superscalar core with pipeline-level ODMR” proposed in the literature. As a result, a flexible processor architecture has been achieved that provides fault tolerance in critical tasks while offering full superscalar performance under normal conditions.



5. VERIFICATION AND RESULTS

This chapter presents the comprehensive verification process, performance analysis, fault injection tests, and physical implementation results of the designed 3-way superscalar out-of-order execution RISC-V processor.

Within the scope of functional verification, extensive random tests were performed to stress different parts of the processor, consisting of 9 different random test scenarios generated with the Google RISC-V DV framework and a deterministic test program containing 8 different algorithms. All tests were verified by cycle-by-cycle comparison with the Berkeley Spike reference model.

In performance evaluation, comparative IPC analyses were performed with the single-way (scalar) version of the processor; the effects of pipeline bottlenecks, branch prediction performance, and memory access latencies on performance were examined using statistical methods.

Within the scope of reliability verification, fault injection tests were applied to critical registers protected by TMR, and a 100% masking rate against single-bit faults was achieved.

Finally, the design was synthesized in TSMC 16nm FinFET technology using Cadence tools and physical design results were reported.

5.1 Verification Methodology

Verification of superscalar processor designs requires a significantly more complex process compared to single-way designs. Many critical functions must be tested together, such as fetching, decoding, and executing multiple instructions simultaneously; dynamically managing data dependencies; and the correctness of speculative execution and recovery mechanisms in case of misprediction. In this work, a comprehensive

verification strategy was followed, using both random test programs and deterministic algorithm-based test programs.

5.1.1 Simulation environment and tools

A commercial simulation tool was used for functional verification of the design. A specialized tracing infrastructure was developed to monitor and record processor behavior during simulation.

In the verification environment, the processor core is tested within a specially designed test environment. The test environment includes a multi-port memory subsystem designed to meet the parallel access requirements of the superscalar architecture. A 5-port SRAM model is used for instruction memory, and this structure supports 5-way instruction fetch operations. On the data memory side, a 3-port structure was preferred, enabling 3 independent load or store operations to be performed in the same cycle. Memory access operations are managed through the Wishbone protocol. Special adapter modules are positioned between the processor core and memory modules, and these adapters convert the core's simplified memory interface to Wishbone signals.

The developed tracing infrastructure records every instruction exiting from the processor's 3-way commit stage. For each completed instruction, the program counter value, instruction code, destination register address with the written value, and in case of memory write operations, address and data information are recorded. These records are used for comparison with the reference model. The tracing module listens to 3 separate commit channels from the processor core and records completed instructions in program order.

The memory models used exhibit ideal behavior and provide single-cycle access latency. This approach enables the functional correctness of the processor microarchitecture to be tested independently of cache or DRAM latencies.

The Spike instruction set simulator, developed by UC Berkeley and accepted as the official reference implementation of the RISC-V instruction set architecture, was used as the reference model [9]. Spike is a golden model that fully defines the behavior of

RISC-V instructions. The designed processor and Spike execute the same test programs, and the trace records produced by both sides are compared line by line.

5.1.2 Test programs and coverage

Two different categories of test programs were used in the verification process. The first category is random instruction sequences generated using a constraint-based random test generation framework. The second category is deterministic C programs specially developed to represent real algorithmic workloads.

5.1.2.1 Constraint-based random test programs

Random test programs were generated using the RISC-V DV verification framework developed by Google and provided as open source [8]. This framework has the capability to create directed instruction streams targeting specific microarchitectural scenarios. Nine different test configurations were prepared to stress different processor subsystems. Table 5.1 summarizes the random test programs used, the microarchitectural components each targets, and the number of instructions generated.

Random test programs cover the majority of the processor's basic functional blocks. Complex scenarios critical for a superscalar architecture, such as data dependencies (RAW, WAW, WAR) and back-to-back branching, have been stressed through these random flows. Test coverage has been brought to a sufficient level to demonstrate the behavior of pipeline resources (ROB, RS, LSQ) at boundary values and the correctness of recovery mechanisms after speculative execution.

5.1.2.2 Algorithm-based deterministic test programs

In addition to random test programs, complex algorithmic C programs modeling real application scenarios were also developed. These programs were compiled using the RISC-V GCC compiler and executed on the processor.

The most comprehensive test program generates a large number of instructions. This program contains a collection of fundamental algorithms from computer science and

Table 5.1: Constraint-based random test programs.

Test Program	Instr.	Targeted Scenario and Description
Basic Arithmetic	30,000	Contains only arithmetic and logical instructions with branch and memory access instructions disabled. ALU performance and data dependency resolution are tested.
Complex Arithmetic	10,000	Special instruction streams containing boundary value cases of integer arithmetic (overflow, underflow, maximum and minimum values) are used.
Basic Branching	30,000	Random instruction sequences containing conditional branch and unconditional jump instructions are generated. Basic correctness of the branch prediction mechanism is tested.
Loop and Branching	10,000	Instruction streams containing loop structures with boundary value arithmetic are generated. Loop-based branch prediction patterns are tested.
Complex Branching	30,000	Instruction streams containing intensive loop structures, boundary value arithmetic, and nested branch patterns together are used. The branch prediction update mechanism is stressed.
Basic Memory	10,000	With branch instructions disabled, random memory access patterns containing only load and store instructions are generated. The LSQ is tested.
Complex Memory	10,000	Intensive memory operations including page boundary crossings, memory region stress tests, and multi-page access patterns are performed.
Simple Mixed Test	30,000	A basic combination test containing loop structures, jump instructions, and random memory accesses together is applied.
Mixed Test	30,000	A comprehensive stress test covering boundary value arithmetic, loops, jumps, random memory accesses, and multi-page operations is performed.

tests the processor's behavior under different computation patterns, memory access patterns, and control flow structures.

The algorithms included in the program and the processor characteristics they target at the microarchitectural level are shown in Table 5.2:

Table 5.2: Algorithm-based deterministic test programs.

Algorithm Group	Targeted Processor Characteristic
Dijkstra and Topological Sort	Intensive memory access and irregular branch patterns with LSQ and branch predictor loading.
0/1 Knapsack and Edit Distance	Matrix-based data access and sequential data dependency chains with RS and CDB usage.
Heap Sort	Intensive repetition of comparison and swap operations with speculative execution performance.
KMP String Matching	Loop-based character comparisons and frequent branch operations.
Sieve of Eratosthenes and Prime Factors	Intensive arithmetic operation capacity and ALU unit stress testing.
Matrix Exponentiation (Fibonacci)	Recursive structures and intensive register usage with Rename/Physical Register File management.
Bit Manipulation and XOR Linear Basis	Complex logical operation sequences and data path integrity.
Minimax (Tic-Tac-Toe)	Deep recursive function calls (call/return) with RAS performance.

These algorithms exhibit different computational characteristics. Graph algorithms contain intensive memory access and irregular branch patterns, while dynamic programming algorithms create matrix-based access patterns and intensive dependency chains. Sorting algorithms require intensive repetition of comparison and swap operations, while text processing algorithms contain loop-based character comparisons.

5.2 Functional Verification Results

All test programs were run comparatively with the Spike reference model and results were analyzed. Extensive constraint-based random tests and algorithm-based deterministic tests were executed. Since random tests generate different instruction sequences on each run, the effective coverage is significantly higher than any fixed

instruction count. In both test categories, the designed out-of-order execution processor achieved full cycle-by-cycle compliance with the Spike reference model.

Verification results demonstrate that the designed processor implements the RV32I instruction set architecture functionally completely and correctly. The completion order of each test instruction, calculated result values, and memory write operations exactly match the reference model.

Ensuring functional correctness demonstrates that the fundamental mechanisms of the out-of-order execution structure are working correctly. These mechanisms are: resolving data dependencies through register renaming, preserving architectural state during speculative execution, returning to the correct recovery point in case of branch misprediction, and completing load-store operations without violating memory ordering.

5.3 Performance Analysis

To evaluate the efficiency of the processor's superscalar architecture, the average number of instructions executed per cycle (IPC) and performance bottlenecks in the pipeline were analyzed under different workloads. The developed test environment has the capability to simultaneously report processor performance parameters during simulation.

5.3.1 Instructions per cycle comparison and speedup ratios

To quantify the performance improvement provided by the designed 3-way superscalar processor, comparative tests were conducted with the single-way (scalar) version of the same architecture. To observe the behavior of the architecture under synthetic stress tests, the results of random tests generated with the RISC-V DV framework are summarized in Table 5.3.

When random test results are examined, it can be seen that the processor achieves a quite high theoretical efficiency of 97% by reaching 2.90 IPC in arithmetic-heavy workloads. However, decreases in IPC values were observed in branch and memory-intensive scenarios. The main reason for this decrease is the difference in branch predictor performance. The absence of any statistical pattern or cyclic structure in randomly

Table 5.3: IPC comparison under random test scenarios.

Test Scenario	1-Way IPC	3-Way IPC	Speedup Factor
Basic Arithmetic	1.00	2.90	2.90x
Complex Arithmetic	1.00	2.80	2.80x
Basic Branching	0.80	1.59	1.99x
Loop and Branching	0.93	2.23	2.40x
Complex Branching	0.94	2.33	2.48x
Basic Memory	0.95	1.87	1.97x
Complex Memory	0.76	1.47	1.93x
Simple Mixed Test	0.83	1.54	1.85x
Mixed Test	0.85	1.65	1.94x

generated instruction sequences has led to the branch predictor exhibiting low success in the 32-43% band.

The average performance metrics obtained on the deterministic algorithm set defined in Section 5.1.2 are presented in Table 5.4.

Table 5.4: Comparison of 1-way and 3-way configurations under deterministic tests.

Architecture	Total Cycles	Average IPC
1-Way (Scalar)	70,550	0.84
3-Way (Superscalar)	34,580	1.71

According to the results obtained in deterministic tests, approximately 2.04x speedup was achieved compared to the scalar core thanks to the 3-way parallel execution structure. The increase in average IPC value from 0.84 to 1.71 proves that the superscalar architecture efficiently utilizes instruction-level parallelism (ILP). In deterministic algorithm tests (representing real-world workloads), the branch predictor achieved prediction accuracy above 80%, demonstrating that the branch predictor provides high efficiency in real applications.

5.3.2 Pipeline stall and bottleneck analysis

To identify the factors limiting processor performance (bottlenecks) at the microarchitectural level, stalls occurring at each stage of the pipeline were examined in detail. Within this analysis, the reasons why reservation stations (RS) cannot feed execution units are addressed in two main categories:

- Not occupied: Covers cases where the reservation station is empty. This situation can arise from 3 different scenarios:
 - Branch misprediction penalty: The pipeline has been flushed after misprediction and correct instructions have not yet reached the reservation station.
 - Instruction buffer empty: New instruction flow has been interrupted due to the instruction buffer being emptied without branch misprediction.
 - Previous stage bottleneck: Although there are instructions waiting in the instruction buffer, new instructions cannot be dispatched due to fill rates of structures such as reorder buffer (ROB) or load-store queue (LSQ).
- Operands not ready: Although the reservation station has an instruction, one or more of the instruction's source operands have not yet been calculated, so it cannot be sent to the execution unit.

A specialized graphical user interface (GUI) was developed to visualize these bottleneck analyses. This interface interprets performance records generated during simulation and presents critical data such as IPC change graph, stall distribution percentages, and branch prediction performance for live or post-simulation analysis.

Figure 5.1 shows the hierarchical stall analysis screen of the developed performance monitoring interface. In this screen, stall reasons for each reservation station (RS0, RS1, RS2) are presented according to the categories explained above. Thanks to the hierarchical structure, sub-reasons of each category can also be examined in detail.

Figure 5.2 presents the detailing of previous stage bottlenecks called “Previous Stage Bottleneck.” This analysis reveals why the dispatch stage is stalling. The ROB fill rate remaining at a low level of 2.6% indicates that the reorder buffer has sufficient capacity for current workloads. LSQ fill rates also run at similarly low levels.

Figure 5.3 shows the change in instruction mix over time. The number of instructions completed (commit), misprediction count, and memory operation count per 100 cycles are monitored. Sudden drops in commit rate indicate periods of intensive branch

Metric	Cycles	Percentage	Visual
▼ RS0 (Pipeline 0) Stall	11,534	33.4%	
▼ Not Occupied	7,588	22.4%	
Misprediction Penalty	6,385	18.5%	
▶ Previous Stage Bottleneck	1,195	3.5%	
▼ Operands Not Ready	3,954	11.5%	
Waiting for CD81	777	2.3%	
Waiting for CD82	717	2.1%	
Waiting for CD83	3,188	9.2%	
▶ RS1 (Pipeline 1) Stall	12,725	36.9%	
▼ Not Occupied	7,727	22.4%	
Misprediction Penalty	6,415	18.6%	
▶ Previous Stage Bottleneck	1,312	3.8%	
▼ Operands Not Ready	4,998	14.5%	
Waiting for CD80	2,853	6.4%	
Waiting for CD82	331	1.0%	
Waiting for CD83	3,267	9.5%	
▼ RS2 (Pipeline 2) Stall	15,282	44.3%	
▼ Not Occupied	7,633	22.1%	
Misprediction Penalty	6,388	18.5%	
▶ Previous Stage Bottleneck	1,245	3.6%	
▼ Operands Not Ready	7,649	22.2%	
Waiting for CD80	3,281	9.5%	
Waiting for CD81	1,936	5.6%	
Waiting for CD83	2,825	8.2%	

Figure 5.1: Reservation station stall analysis.

▼ Previous Stage Bottleneck	1,195	3.5%	
▼ Decode Not Ready	987	2.9%	
RFB Full	892	2.6%	
LSQ Full	94	0.3%	
Instruction Buffer Empty	288	0.6%	

Figure 5.2: Previous stage bottleneck details.

misprediction or memory latency. This graph is critically important for understanding how the processor responds to different workload characteristics.

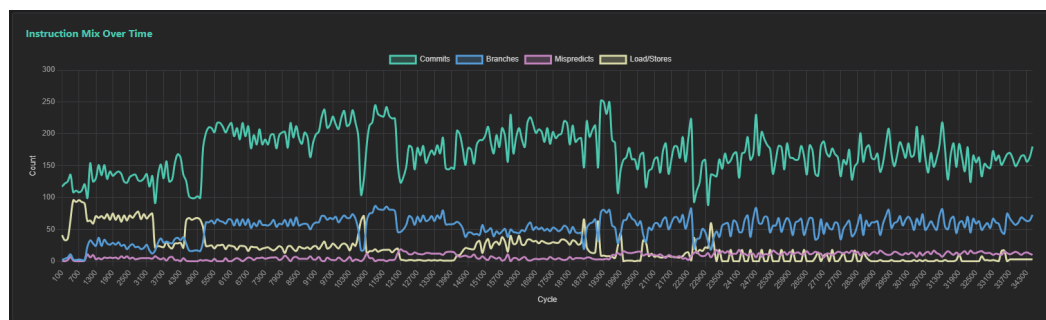


Figure 5.3: Instructions completed, mispredictions, and memory operation counts per 100 cycles.

Figure 5.4 shows how operand dependency stalls accumulate cumulatively for each reservation station. RS2 having the highest wait time reveals that instructions assigned to this station contain longer data dependency chains. This situation indicates that the types of instructions assigned to RS2 (e.g., memory operations or complex arithmetic operations) inherently contain more dependencies.

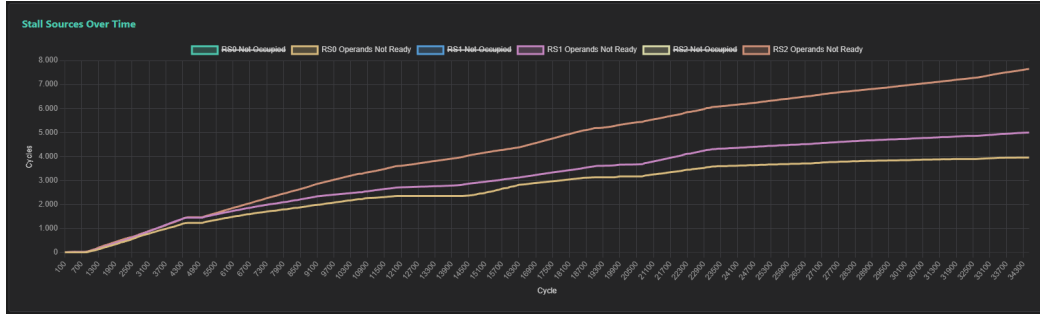


Figure 5.4: Cumulative wait cycles for each reservation station.

Figure 5.5 reveals one of the most critical performance findings: the vast majority of reservation station idle time (Not Occupied) originates from branch misprediction penalty. As simulation progresses, this ratio rises above 80%. This result proves that under current workloads, processor performance is predominantly determined by branch predictor accuracy. Branch predictor improvements stand out as the most effective optimization area that will directly translate to IPC increase.

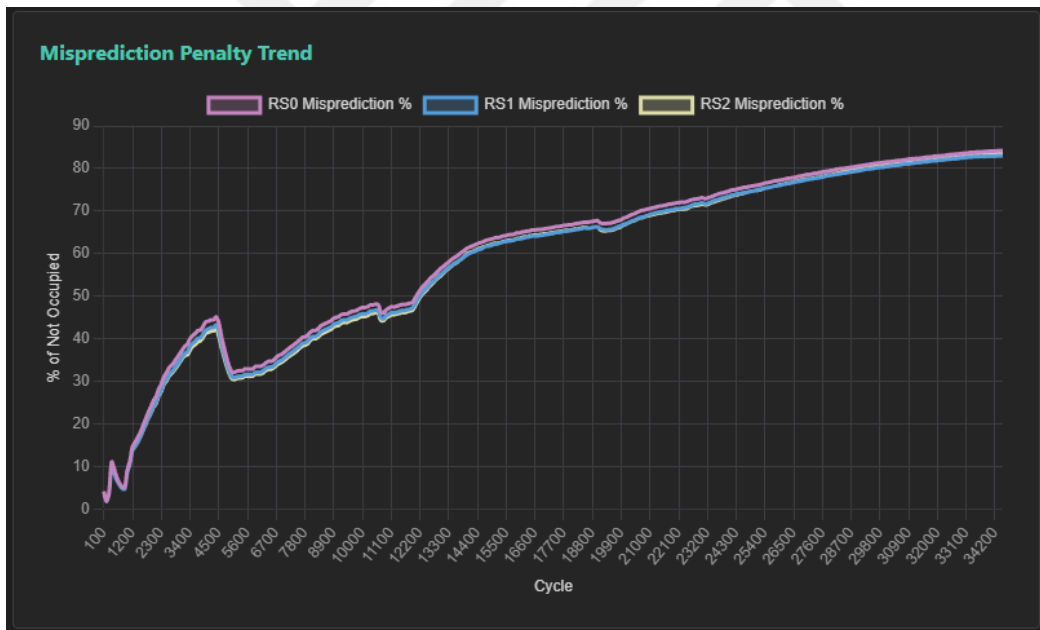


Figure 5.5: Branch misprediction penalty percentage for reservation station idle time.

This developed tool has enabled rapid identification of bottlenecks in optimization work on the architecture and direct observation of the effects of changes made on the system.

5.3.3 Correlation analysis and performance prediction

A statistical analysis method was followed to understand the relationships between factors affecting processor performance. Using the Pearson correlation coefficient on data obtained from simulations, the correlation between IPC value and branch misprediction and memory operations was calculated.

In performance analysis, correlation values are obtained with the following approach:

- Pearson correlation: Measures the direct linear relationship between IPC and a specific metric (e.g., memory access).
- Partial correlation: Isolates the pure effect of another factor (e.g., memory latency) on performance by holding constant the effect of one factor (e.g., branch prediction error).

As a result of the statistical analyses performed, it was confirmed that the most fundamental performance constraint of the superscalar architecture is branch prediction errors. Pearson correlation revealed a strong negative relationship at the level of -0.74 between branch misprediction rate and IPC in the 3-way structure.

A more notable finding was obtained from partial correlation analysis; the pure performance effect of memory operations, when branch prediction errors are removed, is -0.21 in the scalar core while rising to -0.80 level in the superscalar core. This situation shows that as the architecture expands, the effect of the memory system on performance (memory wall problem) becomes much more prominent and that after branch predictor improvements, memory latencies will become the main bottleneck.

5.4 Fault Injection and Reliability Verification

A specialized reliability analysis process was executed to verify the effectiveness of the hardware redundancy mechanisms explained in Chapter 4. The resistance of the design to physical faults was measured with fault injection tests performed at the microarchitectural level.

5.4.1 Fault injection mechanism

A specialized fault injection module was integrated into the design to verify the effectiveness of the hardware redundancy infrastructure. Preferred over commercial safety simulation tools due to license restrictions, this module has the capability to simulate transient hardware faults at the simulation level.

The developed fault injection mechanism has the following features:

- **Target scope:** Except for memory structures assumed to be protected by ECC, all control and data registers in the design can be subjected to fault injection.
- **Fault model:** The injected fault is instantaneously placed in the target register but is designed not to prevent normal write operations to this register at the next clock edge. This behavior reflects real transient fault characteristics.
- **Fault types:** The module has the capacity to generate both single-bit errors and multi-bit errors. Additionally, the maximum number of multi-bit errors that can occur in one cycle is also configurable.
- **Fault rate control:** The number of faults to be injected every thousand clock cycles can be set as a simulation parameter.

A fault injected into an unprotected register will remain in the register until the next write operation, and if a fault also occurs in one of the other replicas in the same TMR group during this period, the voting mechanism will fail, leading to a fatal error. All protected registers in the designed processor are configured to eliminate injected faults with TMR voting results within one clock cycle.

5.4.2 Fault injection test results

Fault injection tests were performed to observe the effects of both single-bit (SEU) and multi-bit (MBU) faults on the system.

In single-bit fault tests, simulations were performed with varying fault densities, including scenarios where SEU events occur at every clock cycle. In both scenarios,

the processor successfully masked all injected single-bit faults through the TMR voting mechanism while maintaining functional correctness. No incompatibility was detected in comparisons made with the Spike reference model, and program flow was completed without interruption. A particularly notable point is that even under intensive fault injection, the processor maintained the IPC value of the single-way (1-way) configuration without experiencing performance loss.

In tests for multi-bit faults (MBU), both detection and recovery capabilities were verified. It should be noted that TMR cannot detect all multi-bit faults: if the same bit is corrupted in two of the 3 replicas, the TMR voter will incorrectly accept the faulty value as correct. The mitigation strategy for this scenario is addressed through physical design measures discussed in Section 5.6.1.

In the test environment, such correlated faults were prevented, and under these conditions, all multi-bit faults were successfully detected. Faults that could not be resolved by TMR voting were rapidly recovered using the checkpoint and rollback mechanism through the RAT checkpoint structure.

5.5 Logic Synthesis Results

The design was synthesized in industrial production technology using the Cadence Genus 23.11 logic synthesis tool. The low-threshold voltage (LVT) cell library of the TSMC 16nm FinFET process was selected as the target technology.

The synthesis process was performed for typical operating conditions (TT corner) at 0.8V supply voltage and 25°C temperature values. A 1 ns clock period was specified as the timing constraint, corresponding to a 1 GHz operating frequency.

Timing results obtained from logic synthesis show that the specified 1 GHz target frequency is met. Table 5.5 summarizes the basic metrics obtained at the synthesis stage.

The critical path identified at the synthesis stage occurs on the signal path extending from the load-store queue (LSQ) head pointer in the dispatch stage to the RAT snapshot memory within the RAT checkpoint. The total combinational delay of this path was

Table 5.5: Logic synthesis results.

Metric	Value
Target Technology	TSMC 16nm FinFET LVT
Operating Conditions	TT, 0.8V, 25°C
Target Clock Period	1 ns (1 GHz)
Total Cell Count	141,868
Worst-Case Slack Value	0 ps (timing met)
Total Power Consumption	150.4 mW

measured as 937 ps, with the remaining 63 ps allocated for setup time and timing uncertainty.

Power analysis performed at the synthesis stage reveals total power consumption under random test activity assumption. Table 5.6 shows the distribution of power components.

Table 5.6: Synthesis stage power distribution.

Power Category	Value (mW)	Ratio
Internal Power	87.3	58.0%
Switching Power	63.0	41.9%
Leakage Power	0.2	0.1%
Total Power Consumption	150.4	100%

When power distribution is analyzed, it can be seen that internal power is the dominant component. This situation reflects the density of sequential elements (flip-flops) within the processor. In the superscalar structure, structures such as the register file, reorder buffer, branch prediction history table, and load-store queue contain significant amounts of sequential elements.

5.6 Physical Design Results

The logic synthesis output was processed through the Place and Route stage using the Cadence Innovus 22.31 physical design tool. Clock tree synthesis, placement optimization, and signal routing were performed during the physical design process.

Figure 5.6 presents the physical view of the design after the placement stage. On the chip area of approximately $400\ \mu m \times 400\ \mu m$, the placement of basic functional units (fetch, issue, RAT checkpoint, etc.) and the areas they occupy can be seen.

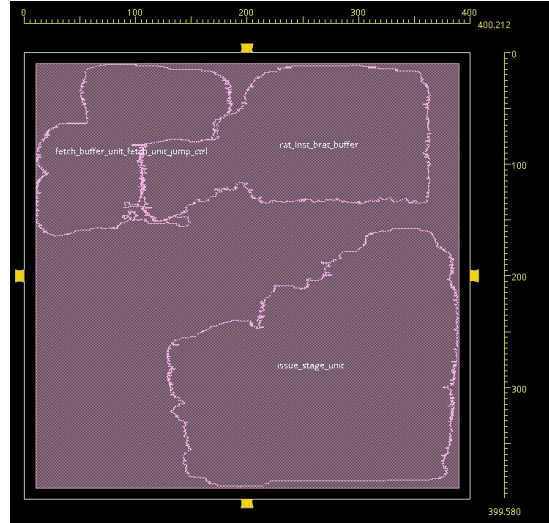


Figure 5.6: Physical layout view of processor core.

Figure 5.7 shows the complexity after signal routing is completed and the density of metal layers. At this stage, all signal nets and connections of the design were implemented in accordance with the specified geometric design rules (DRC).

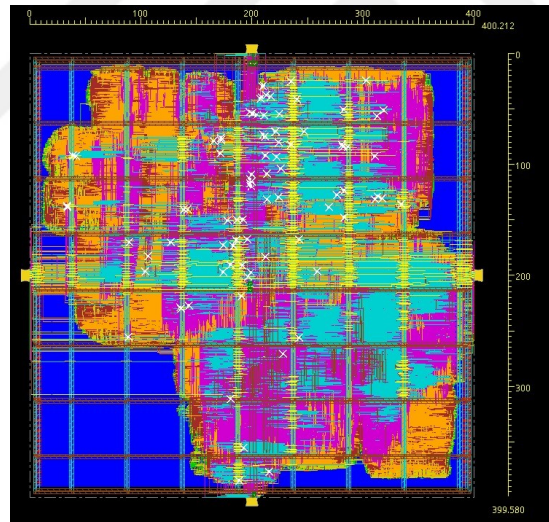


Figure 5.7: Connection density and metal layers after signal routing.

Post-physical design timing analysis reveals different results compared to logic synthesis due to calculation of actual wire delays. Table 5.7 summarizes the physical design results.

The main reason for the increase in cell count compared to logic synthesis at the physical design stage is buffer cells added for timing optimization and clock buffers placed during clock tree synthesis.

Table 5.7: Physical design results.

Metric	Value
Total Cell Count	146,500
Worst-Case Slack Value	-106 ps
Achievable Operating Frequency	~900 MHz
Total Power Consumption	131.6 mW

The critical path after physical design was identified on the signal path extending from the head pointer within the RAT checkpoint to the LSQ head pointer in the dispatch stage. As a result of physical design, the 1 GHz target frequency was not achieved, and a 106 ps timing violation occurred on the critical path.

This result shows that the design cannot fully meet the 1 GHz target frequency but can safely operate at approximately 900 MHz. To eliminate the timing violation, restructuring the combinational logic on the critical path or applying more aggressive synthesis constraints can be considered. Post-physical design power analysis shows lower total power consumption compared to the logic synthesis stage. This difference arises from placement optimization providing shorter wire lengths and consequently reduced capacitive load. Table 5.8 shows post-physical design power distribution.

Table 5.8: Physical design stage power distribution (1 GHz, 0.8V).

Power Category	Value (mW)	Ratio
Internal Power	78.3	59.5%
Switching Power	53.1	40.3%
Leakage Power	0.24	0.2%
Total Power Consumption	131.6	100%

These power analysis results are based on the design tool's default 20% switching activity assumption. For more accurate power estimates, switching activity information obtained from real workloads value change dump (VCD) file needs to be fed to the power analysis tool.

5.6.1 Redundant register placement strategy

As noted in the multi-bit fault testing section, TMR cannot detect faults when the same bit is corrupted in two or more replicas. Research has shown that a single charged particle striking silicon can affect multiple circuits, and this effect area is significantly larger for particles arriving at oblique angles [31]. Therefore, replica registers must be

placed far apart from each other; otherwise, all replicas can be corrupted identically by a single particle strike despite being separate registers.

The designed processor contains 342 separate TMR groups. Each group consists of 3 replica elements that must be placed at a sufficient distance from each other. A target separation distance of $15\text{ }\mu\text{m}$ was established.

Three different approaches were attempted to achieve this placement:

1. Pre-PNR triangular placement: TMR elements were placed in a triangular pattern before the PNR process began. However, this approach failed to achieve the target frequency.
2. Post-PNR displacement: After complete PNR, replica registers were shifted from their original positions by specified amounts. This method resulted in routing failures.
3. Post-placement pre-routing displacement: Replica registers were displaced immediately after the placement phase but before routing began. This approach produced successful results.

Figure 5.8 shows the TMR-protected register placements before and after this optimization. The reduction in placement density and increased separation between replicas is clearly visible.

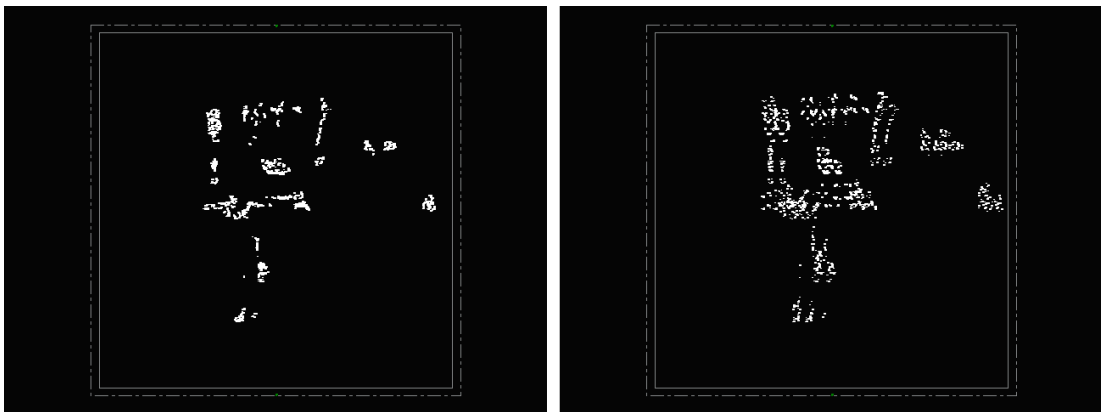


Figure 5.8: TMR-protected register placement: (left) before optimization, (right) after optimization.

Figure 5.9 presents the distance distribution between TMR replicas before and after optimization. The inter-replica distance increased from the 1-2 μm range to approximately 15 μm .

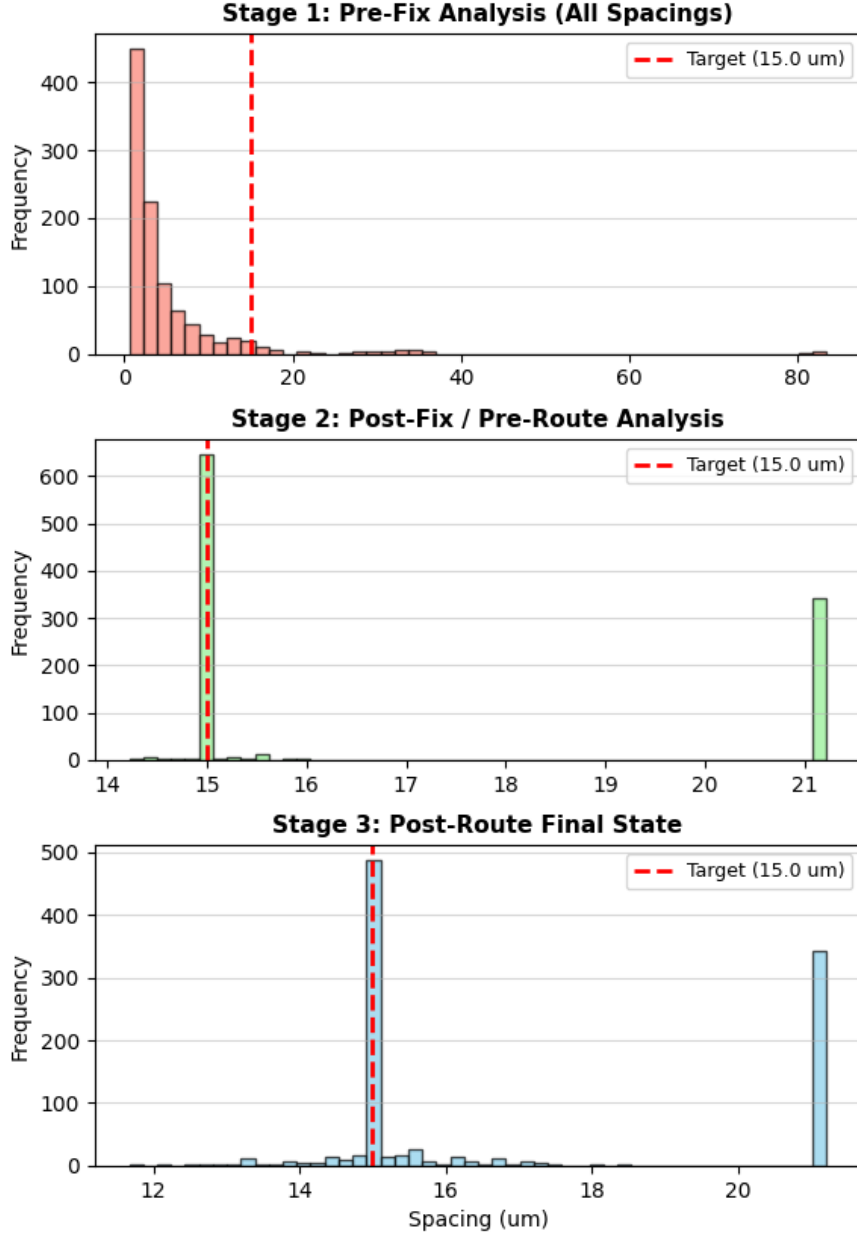


Figure 5.9: Distance distribution between TMR register replicas.

This optimization significantly reduces the probability of correlated faults affecting multiple replicas simultaneously. However, the optimization comes with costs in area, performance, and power:

- Area: Core dimensions increased from $400 \times 400 \mu\text{m}$ to $450 \times 450 \mu\text{m}$.

- Performance: Worst negative slack (WNS) increased from -106 ps to -146 ps, reducing the achievable frequency to approximately 870 MHz.
- Power: Total power consumption increased from 131 mW to 140 mW.

5.7 Critical Path Analysis and Evaluation

In both design stages, the critical path occurs in signal transmission between fundamental components of the speculative execution infrastructure. Table 5.9 compares the critical paths in synthesis and physical design stages.

Table 5.9: Critical path comparison.

Stage	Critical Path Description	Delay	Slack
Logic Synthesis	From LSQ head pointer to RAT checkpoint snapshot	937 ps	0 ps
Physical Design	From RAT checkpoint head pointer to LSQ head pointer	1054 ps	-106 ps
Physical Design with TMR Placement	Same path with increased wire lengths	1146 ps	-146 ps

It is not coincidental that the critical path occurs within the speculative execution mechanism in both stages. In out-of-order execution architectures, all critical pointers must be updated in a single cycle for rapid restoration of processor state in case of branch misprediction. This requirement creates extensive combinational connections between the load-store queue, reorder buffer, and branch recovery structure.

5.8 Discussion

The verification and implementation results presented in this work enable evaluation of the designed 3-way superscalar RISC-V processor in terms of both functional correctness and physical implementability.

From a functional verification perspective, comprehensive test coverage including both random and deterministic test programs was achieved. Full compliance of all tests with the Spike reference model verifies that the processor implements the RV32I instruction set architecture completely. Especially the successful completion of complex algorithm

tests demonstrates that speculative execution and recovery mechanisms work correctly under different computation patterns.

Logic synthesis results reveal that 1 GHz frequency is achievable using approximately 142,000 cells in 16nm FinFET technology. However, at the physical design stage, with actual wire delays taken into account, a 106 ps timing violation occurs, bringing the achievable frequency to approximately 900 MHz level. After applying the TMR register placement optimization for improved fault tolerance, the timing violation increases to 146 ps, reducing the achievable frequency to approximately 870 MHz.

In terms of power consumption, the 131.6 mW value obtained after physical design should be evaluated. With the TMR register placement optimization, power consumption increases to 140 mW. For comparison purposes, Berkeley's single-way in-order Rocket core shows 0.034 mW/MHz dynamic power consumption in TSMC 40nm technology [32]. While direct comparison cannot be made considering the technology difference, the designed processor's 3-way parallel execution capacity, speculative execution infrastructure, and reordering mechanisms can be evaluated as the main sources of power increase.

6. CONCLUSIONS AND RECOMMENDATIONS

This chapter summarizes the results of the thesis, evaluates the findings obtained, and presents recommendations for future work.

6.1 Summary of the Work

In this thesis, a fault-tolerant superscalar out-of-order execution RISC-V processor was designed and implemented. The designed processor offers high performance through its three-way superscalar structure while providing flexible fault tolerance with the on-demand modular redundancy (ODMR) approach.

The processor supports the RV32I base instruction set and is designed as a modern adaptation of the Tomasulo algorithm. With its six-stage pipeline structure, speculative execution, and dynamic scheduling features, it efficiently utilizes instruction-level parallelism. In secure mode, the three parallel execution channels form a TMR structure, masking single-bit faults.

6.2 Obtained Results

The comprehensive verification and physical design work performed demonstrates that the processor produces successful results in terms of both functional correctness and physical implementability.

6.2.1 Functional verification

The processor was comprehensively verified using the Google RISC-V DV framework and Berkeley Spike reference model. Both random and deterministic test programs were successfully completed, and all test results showed full compliance with the reference model. These results verify that the RV32I instruction set architecture is implemented completely, while also demonstrating that all targeted architectural decisions such as

superscalar out-of-order execution structure, eager misprediction recovery, dynamic scheduling, and speculative execution have been successfully applied.

6.2.2 Performance analysis

The superscalar architecture provides the expected performance gains. A maximum IPC value of 2.90 was achieved in random test scenarios, corresponding to 97% of the theoretical maximum. In deterministic algorithm tests, an average IPC of 1.71 was obtained.

When speedup ratios are examined, a maximum speedup of 2.90x was observed in arithmetic-heavy workloads, and a minimum speedup of 1.85x in mixed workloads. In deterministic tests, the three-way structure provided an average speedup of 2.04x compared to the single-way structure.

Although the branch predictor was kept in a simple structure, it achieved prediction accuracy above 80% in deterministic tests representing real-world workloads.

6.2.3 Fault tolerance

Fault injection tests verified the effectiveness of TMR protection. Except for large memory structures assumed to be protected by ECC, all single-bit faults (SEU) injected into registers protected by TMR were successfully masked. For multi-bit faults (MBU), when correlated faults affecting multiple replicas identically were prevented through physical design measures, a 100% detection rate was achieved, preventing silent data corruption.

6.2.4 Physical implementation

Logic synthesis results show that approximately 142,000 cells are used in TSMC 16nm FinFET technology and 1 GHz frequency is achievable. At the physical design stage, when actual wire delays were taken into account, a timing violation of 106 ps occurred, bringing the achievable frequency to approximately 900 MHz. After applying TMR register placement optimization for improved fault tolerance, the timing violation increased to 146 ps, reducing the achievable frequency to approximately 870 MHz.

Under the assumption of 20% switching activity, total power consumption increased from 131.6 mW to 140 mW.

6.3 Contributions of the Thesis

This thesis presents the following novel contributions:

1. A three-way superscalar architecture design suitable for ODMR was implemented. In normal operation, three channels can reach high IPC values by processing independent instructions in parallel; in secure mode, the same three channels form the TMR structure.
2. Spatial redundancy was applied at the pipeline level. Unlike core-level lockstep methods, TMR protection was implemented inside the pipeline, and additional area cost remained limited only to voter circuits.
3. Low-latency mode transition was provided. The application of redundancy at the pipeline level does not require pipeline flushing or state synchronization for inter-mode transitions.
4. Dual-purpose use of the RAT checkpoint structure was proposed. This structure, designed for branch misprediction recovery, provides the infrastructure that can also be used for recovery from radiation-induced critical faults.

6.4 Limitations

This work has some limitations:

1. The 1 GHz target frequency was not fully achieved after physical design. A timing violation of 106 ps occurred on the critical path, which increased to 146 ps after TMR register placement optimization.
2. Memory integration was performed using ideal models. Integration of real SRAM macros will bring additional delay and area cost.
3. ECC protection is assumed for large memory structures, but the physical implementation of ECC is outside the scope of this work.

4. Only the RV32I base instruction set is supported. However, the current microarchitecture is designed to support multi-cycle operations, and integration of multiplication, division, and floating-point extensions can be performed with relatively low complexity.

6.5 Future Works

This work can be extended in the following directions:

1. Implementation of ECC protection and integration into the register file and reorder buffer.
2. Implementation of automatic mode switching from performance mode to secure mode upon fault detection.
3. Extension of application scope by adding RV32M (multiplication/division) and RV32F (floating-point) extensions.
4. Design of cache hierarchy and realistic modeling of memory access latencies.
5. Critical path optimization or addition of additional pipeline stages to eliminate timing violation.
6. Expansion to multi-core structure and inter-core ODMR grouping.
7. Real-time verification and performance measurement on FPGA prototype.

In conclusion, this thesis presents a novel RISC-V processor architecture that combines superscalar out-of-order execution capacity with intra-core fault tolerance. Thanks to the ODMR approach, a dynamic balance can be established between performance and reliability, providing a flexible solution for applications requiring high reliability such as space, aviation, and automotive.

REFERENCES

- [1] **Baumann, R.** (2005). Radiation-induced soft errors in advanced semiconductor technologies, *IEEE Transactions on Device and Materials Reliability*, 5(3), 305–316.
- [2] **Rogenmoser, M., Tortorella, Y., Rossi, D., Conti, F. and Benini, L.** (2025). Hybrid Modular Redundancy: Exploring Modular Redundancy Approaches in RISC-V Multi-core Computing Clusters for Reliable Processing in Space, *ACM Trans. Cyber-Phys. Syst.*, 9(1), <https://doi.org/10.1145/3635161>.
- [3] **Annink, E.B., Geelen, B. and Goossens, S.** (2022). Preventing Soft Errors and Hardware Trojans in RISC-V Cores, *2022 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pp.1–6.
- [4] **Dörflinger, A., Kleinbeck, B., Albers, M., Michalik, H. and Moya, M.** (2022). A Framework for Fault Tolerance in RISC-V, *2022 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCCom/CyberSciTech)*, pp.1–8.
- [5] **Rogenmoser, M., Wistoff, N., Vogel, P., Gürkaynak, F. and Benini, L.** (2022). On-Demand Redundancy Grouping: Selectable Soft-Error Tolerance for a Multicore Cluster, *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp.398–401.
- [6] **Lyons, R.E. and Vanderkulk, W.** (1962). The Use of Triple-Modular Redundancy to Improve Computer Reliability, *IBM Journal of Research and Development*, 6(2), 200–209.
- [7] **Iskin, M.E., Yalcin, B.O. and Yilmazer, A.** (2025). Exploring Fault-Tolerance in RISC-V Architectures, *2025 12th International Conference on Electrical and Electronics Engineering (ICEEE)*, pp.19–23.
- [8] **CHIPS Alliance** (2024). *RISC-V DV: Random Instruction Generator for RISC-V Processor Verification*, <https://github.com/chipsalliance/riscv-dv>.
- [9] **RISC-V Software** (2024). *Spike, a RISC-V ISA Simulator*, <https://github.com/riscv-software-src/riscv-isa-sim>.

- [10] **RISC-V Foundation** (2019). *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2*, <https://riscv.org/specifications/>.
- [11] **Patterson, D.A. and Hennessy, J.L.** (2014). *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann, 5th edition.
- [12] **Waterman, A. and Asanović, K.** (2021). *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.12*, <https://riscv.org/specifications/>.
- [13] **Smith, J.E.** (1981). A Study of Branch Prediction Strategies, *Proceedings of the 8th Annual Symposium on Computer Architecture*, ISCA '81, IEEE Computer Society Press, Washington, DC, USA, p.135–148.
- [14] **Shen, J.P. and Lipasti, M.H.** (2005). *Modern Processor Design: Fundamentals of Superscalar Processors*, McGraw-Hill.
- [15] **Wall, D.W.** (1991). Limits of instruction-level parallelism, *SIGOPS Oper. Syst. Rev.*, 25(Special Issue), 176–188, <https://doi.org/10.1145/106974.106991>.
- [16] **Tomasulo, R.M.** (1967). An Efficient Algorithm for Exploiting Multiple Arithmetic Units, *IBM Journal of Research and Development*, 11(1), 25–33.
- [17] **Santos, D.A., Mattos, A.M.P., Melo, D.R. and Dillo, L.** (2023). Enhancing Fault Awareness and Reliability of a Fault-Tolerant RISC-V System-on-Chip, *Electronics*, 12(12), <https://www.mdpi.com/2079-9292/12/12/2557>.
- [18] **Tedeschi, R., Nadalini, A., Grillotti, F., De Ambroggi, F., Guidetti, E., Benini, L. and Rossi, D.** (2025). A Low-Cost Fault Tolerance Technique for Microcontroller-Class RISC-V Processors, *M. Valle, P. Gastaldo and E. Limiti, editors, Proceedings of SIE 2024*, Springer Nature Switzerland, Cham, pp.21–28.
- [19] **Hamming, R.W.** (1950). Error detecting and error correcting codes, *The Bell System Technical Journal*, 29(2), 147–160.
- [20] **Li, J., Zhang, S. and Bao, C.** (2021). DuckCore: A Fault-Tolerant Processor Core Architecture Based on the RISC-V ISA, *Electronics*, 11(1).
- [21] **Rogenmoser, M. and Benini, L.** (2023). Trikarenos: A Fault-Tolerant RISC-V-based Microcontroller for CubeSats in 28nm, *2023 30th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pp.1–4.
- [22] **Celio, C., Patterson, D.A. and Asanović, K.** (2015). The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor, **Technical Report UCB/EECS-2015-167**, EECS Department, University of

California, Berkeley, <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.html>.

- [23] **Zhao, J., Korpan, B., Gonzalez, A. and Asanović, K.** (2020). SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine, *Proceedings of the Fourth Workshop on Computer Architecture Research with RISC-V*, CARRV 2020, p. 1, https://carrv.github.io/2020/papers/CARRV2020_paper_15_Zhao.pdf.
- [24] **Xu, Y., Yu, Z., Tang, D., Chen, G., Chen, L., Gou, L., Jin, Y., Li, Q., Li, X., Li, Z., Lin, J., Liu, T., Liu, Z., Tan, J., Wang, H., Wang, H., Wang, K., Zhang, C., Zhang, F., Zhang, L., Zhang, Z., Zhao, Y., Zhou, Y., Zhou, Y., Zou, J., Cai, Y., Huan, D., Li, Z., Zhao, J., Chen, Z., He, W., Quan, Q., Liu, X., Wang, S., Shi, K., Sun, N. and Bao, Y.** (2022). Towards Developing High Performance RISC-V Processors Using Agile Methodology, *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp.1178–1199.
- [25] **Mach, J. and Kohútka, L.** (2025). Lockstep Replacement: Fault-Tolerant Design, *IEEE Access*, 13, 94302–94318.
- [26] **Antonacci, M.** (2023). *Design and Implementation of a Superscalar RISC-V Processor*, Politecnico di Milano, <https://www.politesi.polimi.it/handle/10589/205634>.
- [27] **McFarling, S.** (1993). Combining Branch Predictors, **Technical Report TN-36**, Western Research Laboratory, Digital Equipment Corporation, <https://www.hpl.hp.com/techreports/Compaq-DEC/WRL-TN-36.pdf>.
- [28] **Keller, R.M.** (1975). Look-Ahead Processors, *ACM Comput. Surv.*, 7(4), 177–195, <https://doi.org/10.1145/356654.356657>.
- [29] **Johnson, M.** (1991). *Superscalar Microprocessor Design*, Prentice-Hall, Inc., USA.
- [30] **Akkary, H., Rajwar, R. and Srinivasan, S.** (2003). Checkpoint processing and recovery: towards scalable large instruction window processors, *36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36. Proceedings.*, pp.423–434.
- [31] **Zhang, H., Jiang, H., Assis, T.R., Ball, D.R., Narasimham, B., Anvar, A., Massengill, L.W. and Bhuva, B.L.** (2017). Angular Effects of Heavy-Ion Strikes on Single-Event Upset Response of Flip-Flop Designs in 16-nm Bulk FinFET Technology, *IEEE Transactions on Nuclear Science*, 64(1), 491–496.
- [32] **Lee, Y., Waterman, A., Cook, H., Zimmer, B., Keller, B., Puggelli, A., Kwak, J., Jevtic, R., Bailey, S., Blagojevic, M., Chiu, P.F., Avizienis, R., Richards, B., Bachrach, J., Patterson, D., Alon, E., Nikolic, B. and Asanovic, K.** (2016). An Agile Approach to Building RISC-V Microprocessors, *IEEE Micro*, 36(2), 8–20.



CURRICULUM VITAE

Name Surname: Mustafa Ensar IŞKIN

EDUCATION:

- **B.Sc.:** 2022, Istanbul Technical University, Faculty of Electrical and Electronics, Electronics and Communication Engineering Department
- **M.Sc.:** 2026, Istanbul Technical University, Graduate School, Electronics and Communication Engineering Department

PROFESSIONAL EXPERIENCE AND REWARDS:

- 2022-2023, TÜBİTAK BİLGEM, Digital Design Engineer
- 2023-Present, Analog Devices, Digital Design Engineer

PUBLICATIONS, PRESENTATIONS AND PATENTS ON THE THESIS:

- **Iskin, M. E., Yalcin, B. O., & Yilmazer, A.** 2025. Exploring Fault-Tolerance in RISC-V Architectures. *2025 12th International Conference on Electrical and Electronics Engineering (ICEEE)* (pp. 19-23). doi: 10.1109/ICEEE67194.2025.11261941.

OTHER PUBLICATIONS, PRESENTATIONS AND PATENTS:

- **Iskin, M. E., Sevgili, S. O., & Yeniceri, R.** 2021. Auto Relay Handover Method for Extended Star Topology UAV Networks. *2021 12th International Conference on Mechanical and Aerospace Engineering (ICMAE)* (pp. 378-383). doi: 10.1109/ICMAE52228.2021.9522498.