CENG318

MICROPROCESSORS VIZE

Ayben GÜLNAR-191180041

MAYIS 2023

AYBEN GÜLNAR

191180041

CENG318

MICROPROCESSORS

İÇİNDEKİLER

Sayfa

ÇİZELGELERİN LİSTESİ

Çizelge **Sayfa**

ŞEKİLLERİN LİSTESİ

Şekil **Sayfa**

SİMGELER VE KISALTMALAR

Kısaltmalar	Açıklamalar
AArch32:	ARMv8-A 32-bit Architecture
AArch64:	ARMv8-A 64-bit Architecture
ARM:	Advanced RISC Machines
CISC:	Complex Instruction Set Computing
CPU:	Central Processing Unit
FTSE:	Financial Times Stock Exchange
ID:	Identification
IEEE:	Institute of Electrical and Electronics Engineers
IoT:	Internet of Things
LLC:	Limited Liability Company
Ltd:	Limited
MMX:	MultiMedia eXtension
PC:	Program Counter
PSTATE:	Processor State
RISC:	Reduced Instruction Set Computing
SIMD:	Single Instruction Multiple Data
SVC:	Supervisor Call
UCI:	User-Configurable Interrupt

1. GİRİŞ

ARMv8 yani ARM64, 2011 yılında tanıtılan en son ARM CPU mimarisidir. Bu mimari, 64-bit bilgi işlem desteği de dahil olmak üzere birçok yeni özellik ve geliştirmeyi tanıtmasıyla ARM mimarisinin büyük bir evrimini temsil eder.

ARMv8'in en önemli özelliklerinden biri, daha büyük ve daha karmaşık talimat-instructrlar ve veri yapıları yürütmeye olanak tanıyan yeni 64-bit talimat-instructr kümesinin bulunmasıdır. Bu, mimarinin daha fazla bellek ve işlem gücü desteği sağlamasını ve sunucular, masaüstü bilgisayarlar ve mobil cihazlar gibi yüksek performanslı uygulamalar için uygun hale getirmesini sağlar.

Yeni 64-bit talimat-instructr kümesine ek olarak, ARMv8 ayrıca daha geniş bir veri türü yelpazesini destekler. Bu, kayan nokta ve SIMD (Tek Talimat-instructrla Çoklu Veri) işlemleri de dahil olmak üzere daha verimli ve esnek bir veri işleme imkanı sağlar ve uygulamaların daha karmaşık işlemler ve algoritmalar yapmasına olanak tanır.

ARMv8'in bir diğer önemli yönü, enerji verimliliğine odaklanmasıdır. Mimari, iş yüküne bağlı olarak saat hızlarını ve voltajları dinamik olarak ayarlayabilme özelliği gibi güç tüketimini azaltmak için tasarlanmış bir dizi özellik içerir. Ayrıca güç yönetimi teknikleri için destek de sağlar ve böylece ARMv8 tabanlı sistemler, düşük güç tüketimini korurken yüksek performans sunabilir, bu da bataryalı cihazlar için ideal hale getirir.

ARMv8'in en dikkate değer yönlerinden biri, sanal-virtuallaştırmayı desteklemesidir. Bu, birden çok işletim sistemi ve uygulamanın tek bir fiziksel işlemci üzerinde aynı anda çalışmasına olanak tanır ve kaynakların verimli kullanımını ve daha iyi bir güvenlik sağlar. Özellikle, tek bir fiziksel sunucuda birden çok sanal-virtual makinenin çalıştırılması gereken bulut bilişim ve diğer uygulamalar için son derece faydalıdır [1].

Bu araştırmamda, ARM işlemci mimarisinin temel özellikleri ve AArch64 uygulama seviyesi mimarisi hakkında bilgi vermeyi amaçlamaktadır. İlk olarak, ARM işlemci

tarihçesi ve x86 mimarisinden farklılıkları ele alınmaktadır. Daha sonra, ARMv8 mimarisinin desteklediği veri tipleri ve bellek modeli gibi temel konular ele alınmaktadır. Ardından, AArch64 uygulama seviyesi programcı modeli, Registerlar, sistem Registerları, özel durumlar ve bellek modeli gibi konular incelenmektedir. **Özellikle Arm64 Registerları üzerinde detaylıca durulmuştur.**

2. ARM İŞLEMCİ

2.1 Tarihçe

Acorn Computers Ltd, bir İngiliz şirketi, 1970'lerin sonlarında BBC Micro adlı ürünüyle mikrobilgisayar endüstrisinde lider konumdaydı. Bu ürün, 8-bit 6502 işlemciye sahipti. Ancak, daha güçlü işlemcilere ihtiyaç duyduklarında, mevcut 16-bit ve 32-bit işlemcilerin gereksinimlerini karşılamayacağına karar verdiler. Bu nedenle, kendi işlemcilerini tasarlamak ve üretmek için BBC Micro'yu kullanarak tasarım ve simülasyon çalışmalarına başladılar. 1987'de Acorn, Archimedes adlı ürününü tanıttı ve o dönem ev bilgisayarları arasında en güçlüsü olarak kabul edildi. Yeni işlemcilerinde, daha az transistör kullanarak daha yüksek performans sağlayan Reduced Instruction Set Computing (RISC) mimarisini kullanmaya karar verdiler. Bu işlemci daha sonra ARM işlemcisi olarak bilinir hale geldi ve şu anda ARM Holdings, LLC tarafından yönetiliyor. ARM işlemcisi, taşınabilir cihaz pazarında hakimiyet kurmuş durumda ve çoğu akıllı telefonda kullanılıyor. ARM Holdings, 2012 yılında yüksek uçlu masaüstü ve sunucu pazarında rekabet etmek için tasarlanmış yeni bir 64-bit komut seti olan ARMv8-A mimarisini tanıttı. AArch64 olarak da bilinen bu mimari, ARM32'den farklı bir montaj dili kullanır ve daha kolay öğrenilir.

1990'ların sonu, teknoloji alanında patlama yaşanan bir dönemdi ve Arm, bu patlamanın zirvesinde yer alıyordu. Ancak bir sorun vardı: birçok teknoloji şirketi, borca dayalı olarak kurulmuş ve çoğunlukla somut gelir tahminleri olmadan şişirilmiş değerlemelerle değer kazanmıştı. Arm, teknoloji şirketlerinin yüksek hisse değerlemeleriyle yükselmekteydi ve hisse fiyatı, 1999 yılında £10'un üzerine çıkarak, şirketin gerçek kazançlarının 300 katından daha fazla değerlendirildi (çok büyük bir rakam, biliyorum). Bu, şirketin boyutu için olağanüstü bir değerlemeydi ve FTSE 100 endeksinde daha değerli şirketler arasında 30. sıraya yerleşti. Dot-com balonundan beri, bir şirketin boyutu için yapılan bu büyük değerleme hiç görülmemişti. Ancak 2000'lerin başında, kaçınılmaz bir şey oldu ve kriz geldi. Teknoloji sektörü çöktü ve hisse piyasaları genel olarak %80-90 oranında değer kaybetti. Kârlı veya kârsız bir şirket olsa da, keskin bir düşüş yaşandı. Bu, o yıllar için küresel yarıiletken gelirlerinin en kötü dönemi olarak görüldü. Arm, hedeflerine

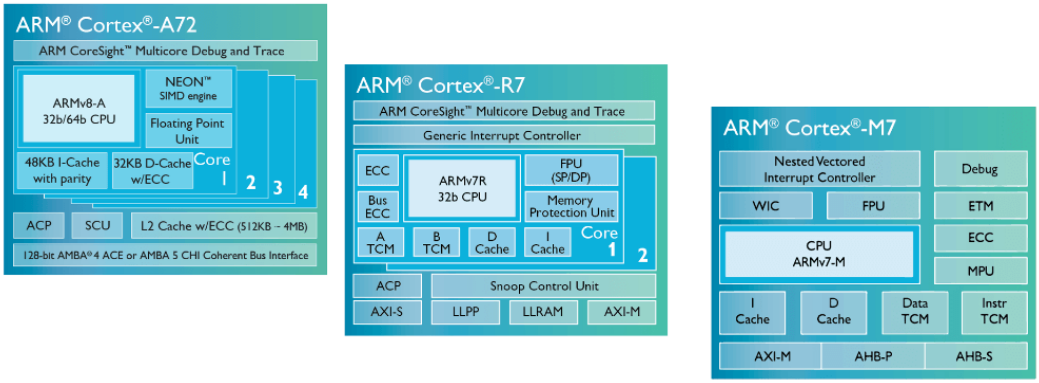
ulaştı ve borç veya finansal hayal kırıklığı yaşamadı, ancak resesyondan da etkilendi. Bu dönemde çalışanlar, "mutluluk" dönemi olarak nitelendirilen ve "olması gereken yer" olarak kabul edilmeyen bir dönem yaşadı. Endüstri çapında işten çıkarmalar yaşandı, ancak Arm'daki işgücü şükür ki çok kötü etkilenmedi. Arm, çeyrekten çeyreğe hayatta kalma çağına girdi. İşletme, 5 yıl içinde nereye varacağına dair özenle hazırlanmış yol haritaları oluşturdu ve 2001 yılında Warren East, Arm'ın CEO'su olarak atandı ve Robin Saxby, Arm'ın Başkanı olarak görev yaptı. Standart işlemci mimarisi olma vizyonları gerçekleşiyordu.

Mikroişlemcilerin boyutları o kadar küçülmüştü ki, sadece çipteki küçük bir bölümü kaplamaktaydılar. Bu nedenle, tek bir çipte veya sistemde yazılım tabanlı sistemlerin nasıl oluşturulacağı sorunu ortaya çıkmıştı. Ayrıca, özel bir işlemci mimarisini sürdürmenin yüksek yatırım ve sahiplik maliyetleri de mevcuttu. Ancak, çoğu şirketin kendi mikroişlemcisini oluşturma yeteneğine veya bunları kullanılabilir hale getirmek için gerekli araçlara sahip olmaması nedeniyle, mikroişlemcilerin IP lisans modelini kullanmaları en önemli nedenlerden biriydi. Bu nedenle, özellikle patlayıcı şekilde büyüyen cep telefonu pazarında Arm, giderek daha fazla SoC'ye tasarlandı ve de facto standart haline geldi.

Ancak, Arm çekirdeği "hard IP" idi ve farklı teknolojilere uygulanması gerçek bir sorundu. Bu nedenle, Arm, teknolojiye özgü bir çekirdek portuna ihtiyaç duymadan herkese lisanslanabilecek bir sentezlenebilir çekirdek üretmek zorundaydı. 2001'de ARM926EJ-S duyuruldu. 5 aşamalı bir pipeline'a ve bir MMU'ya sahipti ve Java hızlandırması ve bazı DSP işlemleri için donanım desteği de sağladı. Dünya çapında 100'den fazla silikon satıcısı tarafından lisanslandı ve milyarlarca ünite sevkiyat yapmaya devam etti.

Dot com çöküşünden sonra endüstri hala toparlanıyordu, Arm da öyleydi. Ancak, istikrarlı bir büyüme ve ARM9, ARM7'nin yerini aldı, daha sonra ARM9E ve ardından ARM10 oldu. ARM10 ve ARM11 teknolojisi, düşük güç tüketimi, yüksek performanslı işleme konusunda gerçekten yeni bir alan açtı. Bu noktada, Arm'ın müşterilerinin başarısının önemi, yaklaşık bir yıl önce Arm'a katıldığımda gerçekten dikkatimi çeken şeylerden biriydi. Önceki blog yazımda da belirttiğim gibi,

ortakların başarısı, Arm için başarı demektir. Bu, Arm'ın iş yaptığı her şirketin "birlikte daha iyi" sembiyotik bir ilişkiye sahip olduğu nadir işletme modellerinden biridir. Bu yaklaşım, Arm'ın İngiliz olması ve İngiltere merkezli olması (yarı iletken endüstrisi için nispeten tarafsız bir bölge) ile kolaylaştırılmaktadır. Arm hiçbir zaman sadece belirli bir sayıda ortağı hedeflememiştir ve bugüne kadar dünya çapında yüzlerce ortağı olmuştur. Dot com çöküşünden sonra endüstri hala toparlanıyordu ve Arm da öyleydi. Ancak istikrarlı bir büyüme ve ARM9, ARM7'nin yerini aldı ve ARM9E ve ardından ARM10 oldu. ARM10 ve ARM11 teknolojisi düşük güç tüketimi ve yüksek performanslı işlemeye yönelik yeni bir dönem başlattı. Arm, sadece 3 yılda 400 kişiden 1.300 kişiye kadar olan bir iş gücü artışı gerçekleştirdi! Ancak, Arm bu noktada daha olgun ve akıllı bir şirket haline gelmişti ve mevcut ürünlerinin yukarı doğru trendini sürdüremeyeceğinin farkındaydı. Endüstrinin tüm ihtiyaçlarını karşılamak için ürün yelpazesini çeşitlendirmeye karar verdiler. Arm, ürün yelpazesini çeşitlendirerek endüstrinin tüm ihtiyaçlarını karşılamaya karar verdi. Bu amaçla geliştirilen Cortex ailesi, farklı sektörlerin ihtiyaçlarına özel olarak tasarlanmış işlemciler sunuyor. Cortex-A, önde gelen mobil uygulamalar için yüksek performanslı işlemciler sağlayarak bu sektöre yönelik çözümler sunuyor. Cortex-R, gerçek zamanlı işlemciler ile özel gereksinimleri olan sektörlere hitap ediyor. Cortex-M, mikro denetleyici endüstrisi için düşük güçlü ve düşük maliyetli çekirdekler sağlıyor. Arm, akıllı telefon kullanımının artması ve daha uzun pil ömrü talebi ile birlikte farklı kullanıcı ihtiyaçlarına yönelik işlem dinamik aralığını karşılayabilen çözümler sunuyor. Örneğin Cortex-A9 MPCore, farklı ihtiyaçlara uygun çoklu çekirdekli işlemciler sunuyor. 2011 yılında tanıtılan "big.LITTLE" yaklaşımı ise yüksek performans gerektiğinde güçlü bir çekirdek sunarak ve performans gerektirilmediğinde daha düşük güçlü bir çekirdeğe geçerek bu konuda daha da gelişme sağladı.



Şekil 2.1.1 ARM Cortex

Arm tabanlı ünitelerin sayısı her geçen yıl büyük ölçüde artıyor ve şu anda yıllık olarak 12 milyar adede ulaşmış durumda. Toplamda ise 60 milyar adet sevkiyat gerçekleştirilmiştir. Bu hızlı artış ile Arm, 2020 yılına kadar yılda 20 milyar sevkiyat hedefine ulaşmayı planlıyor. Bu muazzam bir sayı ve Arm'ın partnerleriyle olan iş birliğinin önemini vurguluyor [2].

2.2 ARM ve x86 Farklılıkları

Tablo 1 ARM vs X86

TABLE I
ARM ARCHITECTURE VS. X86 ARCHITECTURE

	ARM	X86
Size of instructions	The size of the instructions are unified and execution is done within a single clock cycle	Instruction sizes vary according to different complexities
Energy consumption	Control unit design is simplified with an efficiency better energy	Thanks to the complex design, the control unit has a level higher consumption of energy
Program size	Specific function needs more simple instructions to create programs larger size.	Similar function needs a smaller number of instructions, adding more efficiency space.
The pickup and decoding	Pickup process is instructions is much more simple. In addition, access operational codes and operands is possible in the way simultaneously	The pickup process is complex, including different formats and several ways of addressing.

Günümüzdeki bilgisayarlarda, 100-200'den fazla komut kümesi bulunduğunu söylyebiliriz. Komut sayısındaki artışa neden olan birçok faktör sayabiliriz. Bu faktörler arasında, yüksek seviye dillerinin yürütülebilir kod programlarına dönüştürülmesi için yeni komutların eklenmesi veya en iyi hız performansı için yazılım bileşenleri için özel donanım oluşturulması yer almaktadır. Donanım ve yazılım arasındaki ilişkide önemli farklılıkların olduğu iki ana komut kümesi mimarisi türü vardır: Karmaşık Komut Kümeleri (CISC) [4] ve Azaltılmış Komut Kümeleri (RISC) [4]. CISC (Intel x86), yüksek seviye dil işlemleri için donanım desteği sağlar ve kompakt programlara sahiptir. Öte yandan, RISC (ARM) basit komutlara ve esnekliğe odaklanır [4]. RISC mimarisinin ana hedefi, çıktıyı arttırmak ve daha hızlı yürütme sağlamaktır. Bu amaçla, komutların atlanması ve bellek erişimlerinin tamamen önlenmesi gibi teknikler kullanılır. CISC mimarisinin amacı

ise, programlama dilinde kullanılan işlemleri daha iyi eşleştirmek ve kompakt ama aynı zamanda bellek verimli programlar oluşturmak için uygun komutları sağlamaktır.

RISC mimarisi ile başa çıkmak için, Intel 80x86 işlemcisini içsel olarak RISC'e çevirerek komutları genişletir ve bu sayede birçok yeniliği ilk kez tasarımlarında benimseyebilir. Bu yeni zorluklara uyum sağlamak için, Intel işlemcileri değişen pazar taleplerine cevap verebilmek için sürekli olarak geliştirilmektedir.

İntel, sürekli olarak geliştirilerek pazar taleplerine cevap vermektedir. RISC mimarisi ile başa çıkmak ve yeni zorluklara uyum sağlamak için 80x86 işlemcisini dahili olarak RISC'e çevirerek bir genişleme yapmıştır [5]. Tablo 1, ARM ve x86 ISA'ları arasındaki karşılaştırmayı göstermektedir. İntel x86'da MMX, AVX ve SSE ve ARM'da NEON, SIMD talimat-instructrları uzantıları olarak kullanılır. SIMD, büyük miktarda veriyi hesaplamak için tek bir talimat-instructr kullanır ve çoklu döngüler gerektiren görevleri tek bir çevrimde tamamlar. Bu performans avantajına ek olarak, yeniden örnekleme ve çözme talimat-instructrlarını atlayarak güç tüketimini azaltır. Çoklu ortam içeriği işleme gibi birçok yerleşik uygulama, SIMD algoritmalarından faydalanabilir. NEON, birden fazla veri (SIMD) uygulamasını tek bir talimat-instructr ile yapabilen bir teknolojidir. ARM NEON, hatta düşük güçlü mobil cihazlarda bile kullanılmaktadır ve geliştiriciler mümkün olduğunca bu teknolojiden yararlanmalıdır. Bazı matematiksel işlemleri hızlandırmak için kullanılır ve özellikle DSP, görüntü işleme, bilgisayar grafikleri ve bilgisayar görüşünde çok yararlıdır. Bir geliştiricinin bakış açısından, bu teknoloji, Intel'in SIMD SSE talimat-instructr ailesine oldukça benzer. ARM işlemciler RISC mimarisi olarak tasarlanmıştır ve yerel derleyicileri Intel'in tescilli SSE işlemci ailesine kıyasla daha basit bir yapıdadır, ancak bir dizi yerel veya entegre Intel özelliklerine dönüştürülebilir. Bu, C / C ++ kodunun bir ARM işlemcide ortak SIMD ve Intel için optimize edilmesine olanak tanır [3].

3. ARM64

3.1 Armv8 Mimarisi Hakkında

ARMv8, iki yürütme durumu sunar: AArch64 ve AArch32. AArch64 yürütme durumu, 31 adet 64-bit genel amaçlı Register, 64-bit program sayacı (PC), yığın işaretçileri (SP) ve exception bağlantı Registerları (ELR) sağlayan 64-bit yürütme durumlarından oluşur. AArch64 yürütme durumu, 32 adet 128-bit Register desteği ile SIMD vektör ve skalerlere de destek sağlamaktadır. ARMv8 Exception modeli ile dört exception seviyesi (EL0-EL3) arasında yürütme ayrıcalığı hiyerarşisi sağlar. Ayrıca, 64-bit sanal-virtual adresleme desteği sunar ve PE durumunu tutan bir dizi PSTATE elemanı tanımlar. A64 talimat-instructr kümesi, çeşitli PSTATE öğeleri üzerinde doğrudan işlem yapan talimat-instructrlar içerir.

AArch32 yürütme durumu, 32-bit PC, SP ve bağlantı kaydı (LR) dahil olmak üzere 13 adet 32-bit genel amaçlı Register sağlar. Bu Registerların bazıları, farklı PE modlarında kullanılmak üzere çoklu bankalı örnekler içerir. Hyp modundan özel durum dönüşleri için tek bir ELR sağlar. AArch32 yürütme durumu, gelişmiş SIMD vektör ve skaleri için 32 adet 64-bit Register desteği sağlar. PE modlarına dayanan ARMv7-A exception modelini destekler ve bunu exception seviyelerine dayanan ARMv8 Exception modeline uyumlu hale getirir.

Her yürütme durumu, kendi talimat-instructr kümesine sahiptir: AArch64 için A64 ve AArch32 için A32 ve T32. ARM talimat-instructr kümesi, farklı Registerlar ve bellek üzerinde farklı işlemleri gerçekleştirmek için kullanılabilen çeşitli talimat-instructrları içerir.

Sistem Registerları, işlemcinin mevcut durumunu korumak ve yazılımın işlemcinin özelliklerine erişmesini ve davranışını kontrol etmesini sağlamak için kullanılır. Her sistem kaydı, kaydın erişilebileceği en düşük exception seviyesini belirleyen bir ek kullanılarak adlandırılır.

ARMv8 Debug, ARMv8 işlemcilerinde yazılımın hata ayıklama işlemleri için yeni özellikler ve yetenekler sunar. Yeni bir hata ayıklama mimarisi, yeni bir hata

ayıklama iletişim kanalı ve yeni hata ayıklama Registerları içerir. ARMv8 Debug ayrıca, sistem Registerları aracılığıyla hata ayıklama işlevselliğine erişim sağlar.

ARMv8'de kullanılabilecek komut setleri yürütme durumuna bağlıdır: AArch64 yürütme durumu yalnızca A64 olarak adlandırılan tek bir komut setini destekler. Bu, 32 bitlik komut kodlamaları kullanan sabit uzunluklu bir komut setidir. A64 komut seti hakkında daha fazla bilgi için C2 Bölümünde A64 Komut Seti Genel Bakışı'na bakabilirsiniz. AArch32 yürütme durumu ise A32 ve T32 komut setlerini destekler. A32, 32 bitlik sabit uzunluklu bir komut seti olarak ARMv7 ARM komut setiyle uyumludur. T32 ise, değişken uzunluklu bir komut seti olup hem 16 bitlik hem de 32 bitlik kodlamalar kullanır. ARMv7 Thumb® komut setiyle uyumludur. Önceki belgelerde, bu komut setleri ARM ve Thumb komut setleri olarak adlandırılırdı. Her bir komut seti, PE yürütme durumunun hangi komut setini kullanacağını belirler.

ARMv8 işlemcinin özellikleri hakkında kontrol ve durum bilgileri tutan özel Registerlara Sistem Registerları denir. Yazılım, işlemcinin davranışını kontrol etmek ve durumunu erişmek için bu Registerlarını kullanır. Sistem Registerları, bir nokta (.) ve bit alanı adını içeren standart bir formatta adlandırılır. Bu adlandırma yöntemi, Registerların belirli bit alanlarını ve kontrol bitlerini tanımlamak için kullanılır. Örneğin, "PC" Registerı, işlemcinin mevcut program sayacını tutar ve "MODE" biti, işlemcinin belirli bir modda olup olmadığını gösterir. Bu nedenle, PC kaydındaki mod biti için bit alanı "PC.MODE" olarak adlandırılır. Ayrıca, Register içindeki bitlerin sayısal konumları, "<Register_adı>[x:y]" gösterimi kullanılarak belirtilebilir. Burada x ve y Register içindeki bit konumlarının aralığını belirtir. Örneğin, PC kaydındaki 16 ila 23 bit aralığı "PC[23:16]" olarak adlandırılabilir. AArch64 durumunda, Register adları, kaydın erişilebileceği en düşük Exception seviyesini adın sonuna ekleyerek adlandırılır. Örneğin, EL0'dan erişilebilen bir sistem kaydı "<Register_adı>_EL0" olarak adlandırılırken, yalnızca EL3'ten erişilebilen bir Register "<Register_adı>_EL3" olarak adlandırılır. Bu adlandırma yöntemi, ARMv8 Exception modeli tarafından tanımlanan uygun Exception seviyesine göre Registerlara erişimi kontrol eder.

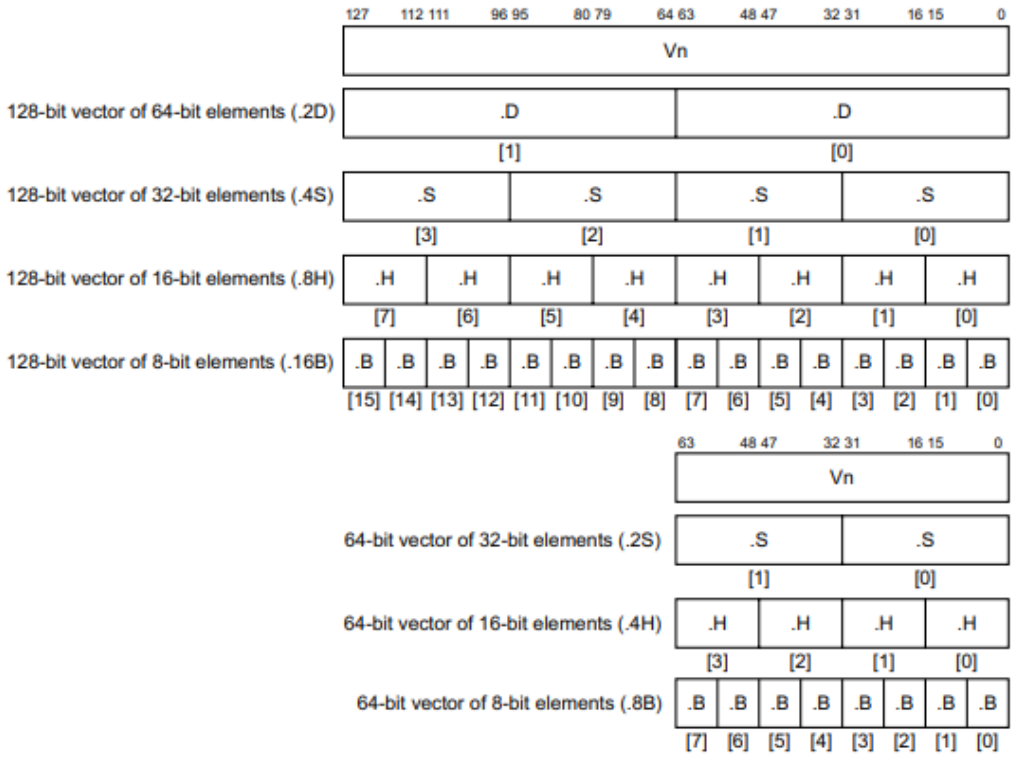
Armv8 mimarisi için çeşitli versiyon isimleri kullanılmaktadır. İşte bazı örnekler:

- Cortex-A53
- Cortex-A57
- Cortex-A72
- Cortex-A73
- Cortex-A75
- Cortex-A76
- Cortex-A77
- Cortex-A78
- Cortex-A79
- Neoverse N1
- Neoverse E1
- Neoverse V1
- Neoverse V2
- Cortex-X1
- Cortex-A510
- Cortex-A710

ARMv8, 64-bit işlemci ailesidir ve ARMv8-A ve ARMv8-R profillerini içerir. ARMv8-M ise, mikrodenetleyiciler için optimize edilmiş bir işlemci ailesidir ve özellikle güvenlik, enerji verimliliği ve performans açısından farklı özelliklere sahiptir. ARMv8-M, ARMv8'in sanal-virtual bellek desteğine sahip değildir ve daha küçük bellek boyutlarına sahip cihazlar için optimize edilmiştir. Ayrıca, ARMv8-M işlemcileri donanım tabanlı güvenlik özelliklerine sahiptir, bu da onları özellikle IoT cihazları ve diğer güvenlik hassasiyeti olan uygulamalar için uygun hale getirir. Ancak, ARMv8 tabanlı birçok farklı çip ve işlemci mevcuttur ve örnekler sadece bunlardan bazılarıdır [6,7].

3.2 Armv8 Desteklenen Veri Tipleri

Desteklenen veri tipleri, registerların ne tür verileri depolayabileceğini ve işleyebileceğini belirler. Bu yüzden araştırmamda buna da yer verdim.



Şekil 3.2.1 SIMD Vectors in AArch64 State

ARMv8 mimarisi çeşitli tamsayı ve floating point veri tiplerini destekler. Sabit noktalı yorumlama ve vektörler de desteklenir. Tamsayı veri tipleri, farklı bit boyutları olan byte, halfword, word, doubleword ve quadword içerir. Floating point veri tipleri ise half precision, tek hassasiyetli ve çift hassasiyetli olmak üzere üç türdür.

ARMv8 mimarisi, genel amaçlı bir Register dosyası ve SIMD ve floating point bir Register file sağlar. AArch64 durumunda, genel amaçlı Register dosyası 64 bitlik Registerlar içerirken, SIMD ve floating point Register file 128 bitlik Registerlar içerir. Ancak, SIMD ve floating point Register dosyasının etkin vektör u<zunluğu, kullanılan A64 komut kodlama türüne göre 64 bit veya 128 bit olabilir. AArch32 durumunda, genel amaçlı Register dosyası 32 bitlik Registerlar içerirken, SIMD ve floating point Register dosyası 64 bitlik Registerlar içerir.

SIMD komutları içeren bir uygulamada, bir Register aynı boyutta ve tipte birden fazla paketlenmiş öğe tutabilir. Bir Register ve veri tipi kombinasyonu, öğelerin

vektörünü tanımlar ve belirtilen veri tipinin bir dizi öğesi olarak kabul edilir. Vektör, veri öğelerinin boyutu ve Register boyutuyla ima edilir.

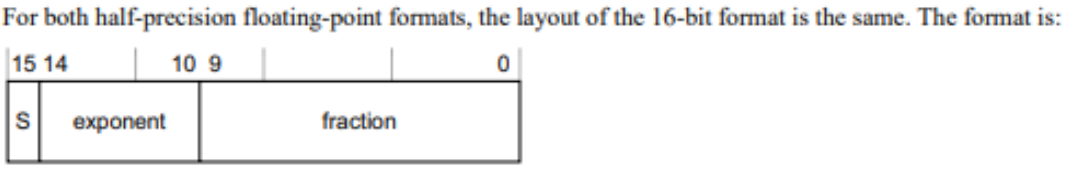
AArch64 durumundaki SIMD ve floating point Registerlar, genellikle Vn olarak adlandırılır ve 0 ile 31 arasındaki bir değere sahiptir. Bu Registerlar, yüklemeler, depolamalar ve veri işleme işlemleri için üç farklı veri formatını destekler. Bu veri formatları, tek bir skaler öğe, 64 bitlik bir vektörde bulunan byte, halfword veya word öğeleri ve 128 bitlik bir vektörde bulunan byte, halfword, word veya doubleword öğeleridir. Tablo A1-1'de belirtilen öğe boyutlarına ve 128 bitlik bir vektör için Vn{.2D, .4S, .8H, .16B} ve 64 bitlik bir vektör için Vn{.1D, .2S, .4H, .8B} şeklinde belirtilen vektör formatlarına sahiptirler [10].

3.2.1 Half Precision Floating-Point

ARMv8 mimarisi, half precision floating point formatlarını destekler ve bu iki formattan biri IEEE half precision, diğeri ise alternatif half precisiondir. Her iki formatta da 16-bit düzeni bulunur ve formatın yorumlanması üst bitlere [14:10] bağlıdır. Eğer 0 < üst < 0x1F ise, değer normalize bir sayıdır ve (–1)S × 2^(üst-15) × (1.fraction) şeklinde ifade edilir. Burada S işaret bitini, fraction ise kesir bitlerini temsil eder. Minimum pozitif normalize sayı 2^-14 veya yaklaşık olarak 6.104 × 10^-5'tir ve max pozitif normalize sayı (2 – 2^-10)×2^15 veya 65504'tür. Üst 0 ise, değer kesir bitlerine bağlı olarak sıfır veya denormalize bir sayıdır. Kesir bitleri 0 ise, değer sıfırdır ve iki farklı sıfır vardır: S==0 ise +0 ve S==1 ise –0. Kesir bitleri sıfır değilse, değer denormalize bir sayıdır ve (–1)S × 2^-14 × (0.fraction) şeklinde ifade edilir. Minimum pozitif denormalize sayı 2^-24 veya yaklaşık olarak 5.960 × 10^-8'dir.

Eğer üst 0x1F ise, değer hangi half precision formatın kullanıldığına bağlıdır. IEEE half precision için, eğer kesir bitleri 0 ise, değer sonsuzdur. S==0 ise tüm normalize sayıları doğru bir şekilde temsil edemeyecek kadar büyük olan tüm pozitif sayıları temsil ederken, S==1 ise tüm normalize sayıları doğru bir şekilde temsil edemeyecek kadar büyük olan tüm negatif sayıları temsil eder. Kesir bitleri sıfır değilse, değer NaN'dir, yani sessiz NaN veya sinyal veren NaN'dir. İki tür NaN, en önemli kesir biti olan bit[9] tarafından ayrılır.

Alternatif half precision format için, eğer üst 0x1F ise, değer normalize bir sayıdır ve -1^S × 2^16 × (1.fraction) şeklinde ifade edilir. Max pozitif normalize sayı (2-2^-10) × 2^16 veya 131008'dir [10].



Şekil 3.2.1.1 Half Precision Floating Point Format

3.2.2 Single-Precision Floating-Point Format

Single precision floating point formatı, bilgisayarlarda sınırlı hassasiyetle gerçek sayıları temsil etmek için bir standardizasyon yöntemidir. Bu format, 32 bitlik bir kelimenin içindeki bitlerin düzenini ve gerçek sayı olarak yorumlanması için belirlenen kuralları içeren IEEE 754 standardı tarafından tanımlanmıştır.

Bu formatın üç ana bileşeni vardır: işaret biti, üstel alan ve kesir alanı. İşaret biti sayının pozitif veya negatif olduğunu belirtirken, üstel alan ve kesir alanı birlikte sayının büyüklüğünü ve hassasiyetini belirler.

Eğer üstel alan 1 ile 254 arasındaysa (0 < üstel < 0xFF), sayı normalleştirilmiş bir sayıdır. Bu sonuçta, değer (-1)^S x. 2^(üstel-127.) x (1.kesir) şeklinde olur. Burada, S işaret bitini, "1." kesirin önünde ikilik olan kısmı gösterir. Normalleştirilmiş kısımda, en büyük hassasiyet ve aralıktatdırlar. Minimum pozitif değer 2^(-126), max pozitif değer ise (2-2^(-23))x2^127' olarak okunur.

Üstel alan 0 ise, sayı sıfır veya normalleştirilmemiş bir sayı olabilir. Bu durumda, kesir alanının da 0 olması durumunda, sayı +0 veya -0 olur ve işaret bitine bağlı olarak aynı şekilde davranır. Ancak, tamsayı karşılaştırmasında ayırt edilebilirler. Kesir alanı sıfır olmayan durumlarda, sayı normalleştirilmemiş bir sayıdır ve normalleştirilmiş sayılardan daha küçük bir aralığa ve hassasiyete sahiptir. Minimum pozitif normalleştirilmemiş değer 2^(-149). 'dir.

Üstel fonksiyon, gerçel sayılarda tanımlanan fonksiyonlardır ve x reel sayısı için en küçük tam sayıyı vermelidir. Sonuçta, x'ten küçük olmayan en küçük tamsayı değeri elde edilir.

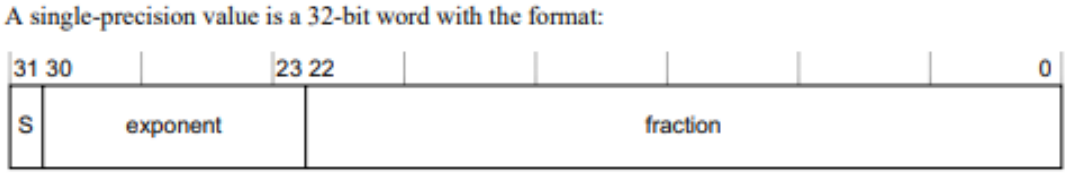
Üstel fonksiyonun sonucu, "⌈x⌉" tir ve aşağıdaki gibi tanımlanır:

$\lceil x \rceil = \min \{n \in Z \mid n \geq x\}$

Burada, Z tamsayılar kümesini gösterir.

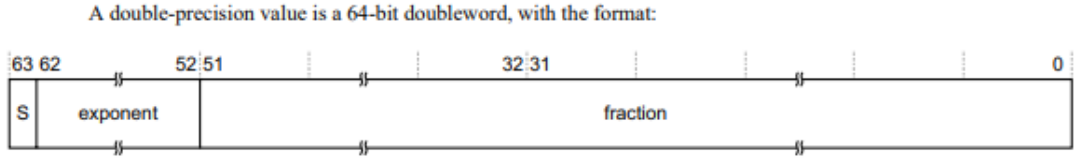
Örneğin, ⌈3.2⌉=4, ⌈-2.5⌉=-2 ve ⌈7⌉=7.

Üstel fonksiyon, matematiksel analizlerde ve algoritmik sorunlarda sıklıkla kullanılan fonksiyonlar arasında örnek verilebilir. Açmak gerekirse, bir dizinin uzunluğunu yukarıya doğru yakınsamak veya bir işlemci veya bellek boyutunu hesaplayark bulmak gibi birçok uygulama alanı vardır [10].



Şekil 3.2.2.1 Single Precision Value Format

3.2.3 Double-Precision Floating-Point Format



Şekil 3.2.3.1 Double Precision Value Format

Double-precision floating point format, bilgisayarlar tarafından sayıların 64 bitlik bir doubleword formatında gösterilmesi için kullanılan bir standart haline gelmiştir ve IEEE 754 tarafından tanımlanmıştır. Bu format, sonsuzluklar, NaN'ler (Not a Number) gibi farklı değerleri temsil edebilir. AArch64 durumunda hem floating point hem de SIMD işlemleri tarafından desteklenirken, AArch32 durumunda sadece

floating point işlemler tarafından desteklenir. Formatın yorumu, üssün değerine bağlıdır. Üs alanı 0 < üs < 0x7FF aralığında bir değere denk geliyrsa, değer normalize edilmiş bir sayıdır ve eşitlik (–1)S × 2^(üs–1023) × (1.fraction) şeklindedir. S işaret bitini gösterirken, üs üstel alanı gösterir, fraction ise önemli rakam kısmını belirtir. Minimum pozitif normalize edilmiş sayı 2^–1022'dir ve max pozitif normalize edilmiş sayı ise (2 – 2^–52) × 2^1023'tür. Üs alanı 0 ise, değer sıfır veya normalize edilmemiş bir sayı olabilir ve bu durum önemli rakam alanının değerine bağlıdır. Eğer önemli rakam biti 0 ise, değer sıfırdır. +0, S==0 ise pozititir, –0 ise S==1 ise negatitir. Eğer önemli rakam alanı 0 değilse, değer normalize edilmemiş bir sayı olark tanımlanır ve eşitlik (-1)S × 2^–1022 × bu şekildedir. Minimum pozitif normalize edilmemiş sayı 2^–1074'tür.

Üs alanı 0x7FF ise, değer sonsuzluk olabilir ya da NaN olarak geçebilirs ve bu durum önemli rakam alanının değerine bağlı olarak değişir. Eğer önemli rakam alanı 0 ise, değer sonsuzluktur. +sonsuzluk, S==0 ise pozitif sonsuzluk, –sonsuzluk ise S==1 ise negatif sonsuzluktur. Eğer önemli rakam alanı sıfır tersi ise, değer NaN'dir. İki tür NaN vardır, sinyalli NaN ve sessiz NaN. Sinyalli NaN, en önemli kelimenin bit[19] alanı 0 olan NaN'dir ve diğer önemli rakam alanları tüm sıfırlar hariç herhangi bir değer alabilir. Sessiz NaN, en önemli kelimenin bit[19] alanı 1 olan NaN'dir ve işaret biti ve diğer önemli rakam alanları herhangi bir değer alabilir [10].

3.2.4 Fixed-Point Format

Sabit noktalı formatlar, yalnızca floating point ve sabit noktalı değerler arasında dönüştürmelerde kullanılır ve genel amaçlı kaydedicilerde kullanılır. Sabit noktalı değerler, işaretli veya işaretsiz olabilir ve 16 bit veya 32 bit olabilir. Dönüştürme talimat-instructrları, sabit noktalı sayıdaki kesirli bit sayısını belirleyen bir argüman alır. Bu argüman, ikili noktanın konumunu belirtir [10].

3.2.5 Bellek Modeli ARM ve Arch64 Durumu

ARM bellek modeli, ARM işlemcinin memory erişimini ve yönetimini nasıl ele aldığına atfeder. ARM bellek modeli, hatalı bellek erişimlerinde exception oluşturma, belirli bellek alanlarına erişimi sınırlama, sanal-virtual adresleri fiziksel

adreslere dönüştürme gibi, çoklu baytlı veriler büyük uçtan küçük uca yorumlanmasını kontrol eder, bellek erişim sırasını kontrol eder, cacheleri ve adres dönüşüm yapılarını kontrol etme gibi ve çoklu işlem birimleri paylaşır belleğe erişimi senkronize olur ve buna benzer özellikler sunar.

Özellikle, ARM bellek modeli, belleğe erişim için işlemci tarafından fiziksel adreslere çevrilen sanal-virtual adresleri kullanarak belleğe erişimi destekler. AArch64 durumunda, 64 bit sanal-virtual adresleri destekler ve Çeviri Kontrol Kaydı, desteklenen sanal-virtual adres aralığını belirler. ARM işlemcide farklı ayrıcalık seviyelerini ifade eden EL1 ve EL0'da her biri kendi çeviri kontrolleri olan iki bağımsız sanal-virtual adres aralığı destekler.

Genel olarak, ARM bellek modeli, bellek erişimini yönetmek ve veri bütünlüğünü sağlamak için güçlü bir özellik seti sağlarken, belleğe verimli erişim sağlamayı ve veri transfer sürelerini en aza indirmeyi destekler [10].

3.3 Armv8-A Aarch64 Uygulama Seviyesi Mimarisi'ndeki Farklılıklar

Armv8-R AArch64, Armv8-A AArch64 profiline göre aşağıdaki şekillerde farklılık gösteren bir uygulama seviyesi programcı modeli sunar: • Armv8-R AArch64 yalnızca tek bir Güvenlik durumunu, Secure'ı destekler.

• EL2 zorunludur.

• EL3 desteklenmez.

• Armv8-R AArch64, A64 ISA talimat-instructr setini bazı değişikliklerle destekler [11].

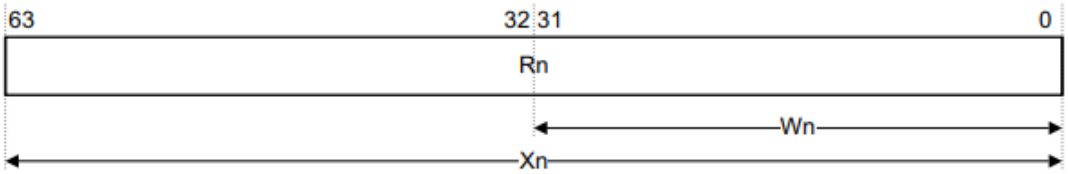
4. AARCH64 UYGULAMA DÜZEYİ PROGRAMCI MODELİ

4.1 Aarch64 Durumunda Registerlar

AArch64 uygulama düzeyinde, bir ARM İşlem birimi 31 adet genel amaçlı register içerir. Bu registerlar R0 ila R30 arasında numaralandırılır ve her biri şu şekilde erişilebilir:

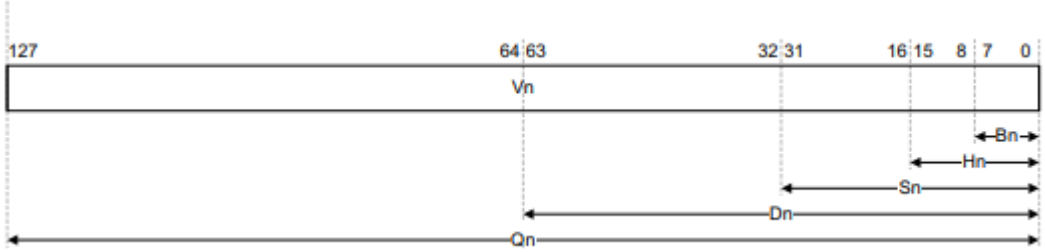
- 64-bit genel amaçlı bir belirtilir X0 ila X30 olarak adlandırılır.
- 32-bit genel amaçlı bir belirtilir W0 ila W30 olarak adlandırılır. 64-bit ve 32-bit register isimlendirmesi arasındaki ilişki Şekil B1-1'de gösterilmiştir. Örneğin, X0, W0'ın alt 32 bitine karşılık gelir ve X1, W0'ın üst 32 bitine karşılık gelir.

See the register name mapping in Figure B1-1.



Şekil 0.1.1 General Purpose Register

AArch64 mimarisinde, X30 kaydı işlem çağrıları için bağlantı kaydı olarak kullanılır. Alt bir program çağrısı yapıldığında, çağıran kodun dönüş adresini X30 kaydında saklar. Bu sayede, alt program işlemini tamamladığında, doğru konuma geri dönebilir. Talimat-instructr kodlamalarında, ZR (sıfır kaydırgıcısı) olarak adlandırılan kaydın değeri 0b11111 (31) olarak temsil edilir. Bu değer, talimat-instructrlar için bir operand olarak kullanılarak, argümanın sıfır değerini alması gerektiğini belirtmek için kullanılır. Ancak, ZR kaydının fiziksel bir Belirtilir uygulanması gerekli değildir. Bazı uygulamalar, sıfır değerlerini oluşturmak için farklı bir mekanizma kullanabilirler.



Şekil 0.1.2 SIMD and Floating Point Register

AArch64 mimarisinde, SP (Stack Pointer - Yığın İşaretçisi) 64 bitlik bir Belirtilir kullanılır ve yığın işaretçisi olarak ayrılmıştır. SP kaydının en az anlamlı 32 biti,

WSP Register adı kullanılarak erişilebilir. SP kaydı bir talimat-instructrta bir işlemci operandı olarak kullanıldığında, mevcut yığın işaretçisinin kullanımını belirtir. Yığın işaretçisi, Exception Seviyesi 1'de (EL1) yapılandırılabilecek 16 bayt sınırına hizalanabilir.

PC (Program Counter - Program Sayacı), AArch64 mimarisinde yürütülen komutun adresini tutan 64 bitlik bir Registertır. Ancak, yazılım doğrudan PC kaydına yazamaz. PC yalnızca bir dallanma talimat-instructrı yürütüldüğünde veya bir exception meydana geldiğinde ve işlemci bir exception girişi veya exception dönüşü gerçekleştirdiğinde güncellenebilir. Başka bir deyişle, PC işlemci tarafından kontrol edilir ve yazılım yalnızca belirli talimat-instructrları yürüterek veya exceptionları tetikleyerek değerini dolaylı olarak değiştirebilir.

Triple ve Floating-Point İşlemcisi için 32 adet V0-V31 SIMD ve kayan nokta registeri mevcuttur. Her register aşağıdaki şekillerde erişilebilir:

• Q0'dan Q31'e kadar 128 bitlik bir register belirtilir.

• D0'dan D31'e kadar 64 bitlik bir register belirtilir.

• S0'dan S31'e kadar 32 bitlik bir register belirtilir.

• H0'dan H31'e kadar 16 bitlik bir belirtilir.

• B0'dan B31'e kadar 8 bitlik bir belirtilir.

• 128 bitlik bir dizi elemanlar vektörü.

• 64 bitlik bir dizi elemanlar vektörü.

Eğer bir SIMD ve kayan nokta registeri tamamını işgal etmeyen bir register ismi belirtiliyorsa, o register için belirtilen bit sayısı en az anlamlı bitleri ifade eder [10, 12].

4.2 Aarch64 Durumundaki Registerların Sözde Kod Ayrıntıları

Burada zaten kod olarak verdiğim için şekillerde ayrıca belirtmedim.

```
// X[] - assignment form
// ====================
// Write to general-purpose register from either a 32-bit and 64-bit value.
X[integer n] = bits(width) value
 assert n >= 0 && n <= 31;
 assert width IN {32,64};
 if n != 31 then
 _R[n] = ZeroExtend(value);
 return;
// X[] - non-assignment form
// ========================
// Read from general-purpose register with implicit slice of 8, 16, 32 or 64 bits.
bits(width) X[integer n]
 assert n >= 0 && n <= 31;
 assert width IN {8,16,32,64};
 if n != 31 then
 return _R[n]<width-1:0>;
 else
 return Zeros(width);
The _R[] function provides a view of the physical array of the physical general-purpose registers.
array bits(64) _R[0..30];
The use of the SP[] function is reading or writing the current SP. This function has prototypes:
SP[] = bits(width) value;
bits(width) SP[];
The use of the PC[] function is reading the PC. This function has prototype:
bits(64) PC[];
The _V[] function provides a view of the physical array of the physical SIMD and floating-point registers.
array bits(128) _V[0..31];
The use of the V[] function is reading or writing V0-V31, using n to index the required register.
// V[] - assignment form
// ====================
// Write to SIMD&FP register with implicit extension from
// 8, 16, 32, 64 or 128 bits.
V[integer n] = bits(width) value
 assert n >= 0 && n <= 31;
 assert width IN {8,16,32,64,128};
 _V[n] = ZeroExtend(value);
 return;
```

```
 return;
// V[] - non-assignment form
// =========================
// Read from SIMD&FP register with implicit slice of 8, 16
// 32, 64 or 128 bits.
bits(width) V[integer n]
 assert n >= 0 && n <= 31;
 assert width IN {8,16,32,64,128};
 return _V[n]<width-1:0>;
The use of the Vpart[] function is reading or writing the lower or upper half of V0-V31, using n to index the required
register, and part to indicate the required half.
// Vpart[] - non-assignment form
// ============================
// Read lower half of a SIMD&FP register with implicit slice
// of 8, 16, 32 or 64 bits, or read upper half as 64 bits.
bits(width) Vpart[integer n, integer part]
 assert n >= 0 && n <= 31;
 assert part IN {0, 1};
 if part == 0 then
 assert width IN {8,16,32,64};
 return _V[n]<width-1:0>;
 else
 assert width == 64;
 return _V[n]<127:64>;
// Vpart[] - assignment form
// =========================
// Write lower half of a SIMD&FP register with implicit extension
// from 8, 16, 32, or 64 bits, or write upper half from 64 bits.
Vpart[integer n, integer part] = bits(width) value
 assert n >= 0 && n <= 31;
 assert part IN {0, 1};
 if part == 0 then
 assert width IN {8,16,32,64};
 _V[n] = ZeroExtend(value);
 else
```

[10]

4.3 İşlemci Durumları, PSTATE

AArch64 için, PSTATE işlem durumu ile ilgili bilgileri tutmaya yarar. Aşağıdaki PSTATE bilgileri EL0'da erişilebilir:

Veri İşleme Bayrakları:

N: Negatif durum bayrağı (N), bir işlemin sonucunun işaretli bir tam sayı olarak kabul edilmesi durumunda, sonucun negatif olup olmadığını belirlemek için kullanılan bir bayraktır. Eğer sonuç negatif ise N bayrağı 1 olarak ayarlanır, aksi takdirde 0 olarak ayarlanır. Örneğin, -5 ve +5 işaretli tam sayılarına birer bitlik imza eklenerek binary sisteme dönüştürüldüğünde, -5'in binary karşılığı 11111011, +5'in binary karşılığı ise 00000101 olur. Bir işlem sonucu negatif olduğunda, sonuçta en sol bit 1 olacağı için, N bayrağı 1 olarak ayarlanır.

Z: Sıfır durum bayrağı (Z) bir komutun sonucunun sıfır olduğunu gösterir. Bu genellikle bir karşılaştırma işleminin eşit olduğunu ifade eder. Ancak, bazı

komutlarda sıfır sonucu farklı anlamlar taşıyabilir, bu nedenle Z bayrağı sadece karşılaştırma işlemleri için değil, diğer işlemler için de kullanılabilir.

C: Taşıma durumu (carry) bayrağı, bir işlemin sonucunda bir taşıma işlemi oluşup oluşmadığını gösteren bir bayraktır. Örnek vermek gerekirse, bir eklemenin sonucunda sayıların toplamı, işlemci tarafından tutulan veri genişliğini aşarsa, bir taşıma işlemi gerçekleşir. Bu durumda, taşıma durumu bayrağı (C) 1 olarak ayarlanır. Ancak, toplam işlemi sınırları aşmazsa C bayrağı 0 olarak ayarlanır.

V: Taşma durum bayrağı (V), bir işlemin sonucu veri boyutunu aşan bir değer üretirse (overflow), yani sonuç için ayrılan bit sayısı yetersiz kalırsa, V bayrağı 1 olarak ayarlanır. Örneğin, 8 bitlik bir işlemde 11111111 (255) sayısına 1 eklediğimizi düşünelim. Sonuç olarak 100000000 (256) elde edilir. Ancak 8 bitlik bir sayı sadece 11111111'e kadar olan sayıları temsil edebilir, dolayısıyla sonuç 8 bitlik bir alanda sığmazsa taşma durumunda kalır ve V bayrağı 1 olarak set edilir. Taşma durumu oluşmazsa V bayrağı 0 olarak set edilir.

Exception Maskeleme Bitleri

D: Hata ayıklama exceptionsı maskesi biti. EL0, maske bitlerini değiştirmek için etkinleştirildiğinde, bu bit görünür ve değiştirilebilir. Ancak bu bit EL0'da mimari olarak görmezden gelinir.

A: Sistem hatası maskesi biti, önceki mimari sürümlerinde dış asenkron kesme biti olarak adlandırılır.

I: IRQ maskeleme biti.

F: FIQ maskeleme biti.

Her bitin olası değerleri şunlardır:

0: Exception maskelenmedi

1: Exception maskelendi [10,15]

4.4 Sistem Registerları

Sistem kayıtları, bilgisayar donanımı ve yazılımı arasındaki interfacete önemli bir rol oynadığı açıktır. Bu kayıtlar, yürütme kontrolünü sağlamak, sistem durumu ile ilgili bilgi sağlamak, sistem yapılandırması için gereken parametreleri tutmak ve işletim sistemi gibi high-level yazılımların çalışmasına yardımcı olmak için kullanılır. Ancak, çoğu sistem kaydı EL0 (Kullanıcı Seviyesi 0) düzeyinde erişilemez. EL0, normal kullanıcı programları tarafından çalıştırılır ve erişim hakları kısıtlıdır. Bu yüzden, birçok sistem kaydı EL0 seviyesinde erişime kapalıdır ve yalnızca private işlemler veya düşük seviye sistem yazılımları tarafından erişilebilir durumdadırlar. Bununla birlikte, bazı sistem kayıtları EL0 seviyesinde çalışan yazılımlar tarafından erişilebilir hale getirilebilir.

Bu kayıtlar, önbellek yönetimi desteği sağlayan Cache ID, CTR_EL0 ve DCZID_EL0 kayıtları, hata ayıklama işlemleri için MDCCSR_EL0, DBGDTR_EL0, DBGDTRRX_EL0 ve DBGDTRTX_EL0 gibi kayıtlar, iş parçacığı kimlikleri için TPIDR_EL0 ve TPIDRRO_EL0 kayıtları ve zamanlayıcı işlemleri için kullanılan CNTPCT_EL0, CNTVCT_EL., CNTP_CVAL_EL0, CNTP_TVAL_EL0, CNTP_CTL_EL0, CNTV_CVAL_EL0, CNTV_TVAL_EL0 ve CNTV_CTL_EL0 gibi registerlar arasındadır.

Ancak, EL0 seviyesinde erişim izni verilmeyen bir sistem kaydına erişilmeye denenirse, bu talimat-instructr “UNDEFINED” olarak kabullenilir ve işlemci bu talimat-instructrı çalıştırmaz. Bu nedenle, EL0 seviyesinde erişim izni verilen kayıtların yanı sıra, izin verilmeyen kayıtların hangileri olduğunu da bilmek önem taşımaktadır [12].

ACCDATA_EL1: Accelerator Data

ACTLR_EL1: Auxiliary Control Register (EL1)

ACTLR_EL2: Auxiliary Control Register (EL2)

ACTLR_EL3: Auxiliary Control Register (EL3)

AFSR0_EL1: Auxiliary Fault Status Register 0 (EL1)

AFSR0_EL2: Auxiliary Fault Status Register 0 (EL2)

AFSR0_EL3: Auxiliary Fault Status Register 0 (EL3)

AFSR1_EL1: Auxiliary Fault Status Register 1 (EL1)

AFSR1_EL2: Auxiliary Fault Status Register 1 (EL2)

AFSR1_EL3: Auxiliary Fault Status Register 1 (EL3)

AIDR_EL1: Auxiliary ID Register

ALLINT: All Interrupt Mask Bit

AMAIR2_EL1: Extended Auxiliary Memory Attribute Indirection Register (EL1)

AMAIR2_EL2: Extended Auxiliary Memory Attribute Indirection Register (EL2)

AMAIR2_EL3: Extended Auxiliary Memory Attribute Indirection Register (EL3)

AMAIR_EL1: Auxiliary Memory Attribute Indirection Register (EL1)

AMAIR_EL2: Auxiliary Memory Attribute Indirection Register (EL2)

AMAIR_EL3: Auxiliary Memory Attribute Indirection Register (EL3)

AMCFGR_EL0: Activity Monitors Configuration Register

AMCG1IDR_EL0: Activity Monitors Counter Group 1 Identification Register

AMCGCR_EL0: Activity Monitors Counter Group Configuration Register

AMCNTENCLR0_EL0: Activity Monitors Count Enable Clear Register 0

AMCNTENCLR1_EL0: Activity Monitors Count Enable Clear Register 1

AMCNTENSET0_EL0: Activity Monitors Count Enable Set Register 0

Şekil 4.1.3 Sistem Registerları

Yukarıdaki şekilde Sistem registerlarının bir kısmı verilmiştir. Normalde 3 sayfa olduğu için bir kısmını ekledim [16].

4.4.1 Sistem Kontrol Registerları

AArch64 yürütme durumu, her seviyede ve her zaman 31 adet 64 bit genel amaçlı register sağlar. Bu registerlar 64 bit genişliğinde olup X0-X30 olarak adlandırılır.

EL1 seviyesinden sonrasında btün bitler kullanılamaz drumdadır. Bazı bitlerin anlamları şöyledir:

- UCI: Bu bit ayarlandığında, AArch64'te EL0 erişimini kullanarak DC CVAU, DC CIVAC, DC CVAC ve IC IVAU komutlarını çalıştırarak önbellek bakımı yapılabilir.

- EE: Exception işlemlerinde kullanılan bayt sırasını belirler. 0, küçük endian'ı; 1, büyük endian'ı gösterir.
- EOE: EL0'da açık veri erişiminin bayt sırasını belirler. 0, küçük endian'ı; 1, büyük endian anlamındadır.
- WXN: Yazma izni XN (eXecute Never) anlamına gelir. 0, yazma izni olan bölgelerin XN'ye zorlanmadığı anlamına gelir; 1, yazma izni olan bölgelerin XN'ye zorlandığı anlamına gelir.
- nTWE: Tuzağa düşürmeyin WFE. 1, WFE komutlarının normal olarak yürütüleceğini belirtir.
- nTWI: Tuzağa düşürmeyin WFI. 1, WFI komutlarının normal olarak yürütüleceğini belirtir.
- UCT: Bu bit ayarlandığında, CTR_EL0 registerına AArch64'te EL0'dan erişim yapılabilir.
- DZE: DC ZVA komutuna EL0'dan erişim sağlanıp sağlanamayacağını belirler. 0, yasaklandığı anlamına gelir; 1, izin verildiği anlamına gelir.
- I: Komut önbelleğini etkinleştirir. Bu bit, EL0 ve EL1'deki komut önbellekleri için bir etkinleştirme bitidir. Normal bellekteki komut erişimleri önbelleğe alınır.
- UMA: EL0, AArch64 kullanırken, kesme maskelerine EL0'dan erişim kontrol eder.
- SED: EL0'da AArch32 kullanarak SETEND komutlarını devre dışı bırakır. 0, SETEND etkin konumda oldpunu; 1 ise devre dışı bırakıldıını gösterir.
- ITD: IT komutunun kullanılıp kullanılamayacağını belirler. 0, IT komutunun kullanılabilir olduğunu; 1, 16 bitlik bir komut olarak kabul edildiğini belirtir.

"CP15BEN", AArch32 CP15 DMB, DSB ve ISB engelleyici işlemleri için bir etkinleştirme bitidir. SA0, EL0 için Yığın Hizalaması Kontrol Etkinleştirme, SA,

Yığın Hizalaması Kontrol Etkinleştirme, C, EL0 ve EL1'deki veri önbellekleri için bir etkinleştirme bitidir. Ayrıca, M, MMU'yu etkinleştirir. Bu ifadeler, ARM tabanlı işlemcilerdeki bellek yönetimi ve performans optimizasyonu gibi konuları ele alır [12].

4.4.2 Endianness

Endianness, bilgisayar belleğindeki çok byte'lı verilerin saklanma sırasını ifade eder. Büyük endian ve küçük endian olmak üzere iki farklı türü vardır. Büyük endian'da, en yüksek anlamlı byte (MSB) belleğin en düşük adresinde saklanırken, küçük endian'da en düşük anlamlı byte (LSB) belleğin en düşük adresinde saklanır.

Verilerin bellekte saklanma şekli iki şekilde olabilir: Küçük uçlu (Little-Endian - LE) veya Büyük uçlu (Big-Endian - BE) olarak adlandırılır. Büyük uçlu makinelerde, bir veri bellekteki en anlamlı bayttan başlayarak sırayla depolanır. Küçük uçlu makinelerde ise veri bellekteki en anlamsız bayttan başlayarak sırayla depolanır. Bu farklılık byte-sırası terimiyle de ifade edilir [15].

4.5 EL0 Düzeyinde Performans Monitörlerine Erişim

Performans Monitörleri, sistemdeki farklı bileşenlerin performansını ölçmek ve analiz etmek için kullanılan önemli bir araçtır. Ancak, yazılımın Performans Monitörleri kullanabilmesi için daha yüksek bir Exception seviyesinde çalışması gerekmektedir.

Bu seviyede, PMUSERENR_EL0 sistem kaydındaki EN, ER, CR ve SW bitlerinin ayarlanması gerekmektedir. EN biti, Performans Monitörleri Registerlarına EL0'da erişime izin verirken diğer erişimleri engeller. ER biti, EL0'da etkinlik sayacı okuma erişimine izin verir. CR biti, EL0'da PMCCNTR_EL0'nin okuma erişimine izin verir. SW biti ise, EL0'da PMSWINC_EL0'in yazma erişimine izin verir.

Performans Monitörleri, sistemdeki performans sorunlarını tespit etmek ve gidermek için önemli bir araçtır. Ancak, Performans Monitörleri'nin kullanımı sistem kaynaklarını da kullanır, bu nedenle Performans Monitörleri'nin etkinleştirilmesi,

gereksiz yere kullanılmaması ve sadece performans sorunlarının tespit edilmesi ve giderilmesi için kullanılması önerilir [12].

4.6 Exception Handling (Exception İşleme)

Exceptionlar işlemci çalışması sırasında beklenmedik olayları temsil eder ve önceden belirlenmiş bir exception seviyesiyle ilişkilidir. Exceptionnın meydana gelmesi durumunda, işlemci kaydedilen işlem durumunu, program sayacını ve diğer ilgili bilgileri exception işleyicisi için ayrılmış bir yığına kaydeder. Bu kayıtlar, exception işleyicisi tarafından kullanılacak ve işlemciye exception ile başa çıkma talimat-instructrları verecektir.

Exception işleyicisi, işlemcinin bir hata durumundan kurtulmasına ve exception nedeniyle kesintiye uğrayan programın yeniden başlatılmasına yardımcı olur. İşlemci, sabit bir adresteki vektör tablosundan ilgili exception işleyicisinin adresini yükler ve işlemci çalışmasını bu işleyici ile devam ettirir. Exception işleyicisi, exceptionnın kaynağına bağlı olarak özelleştirilmiş bir işlevdir ve exception durumunu ele almak için uygun adımları atar. Bu adımlar, exception kaynağına bağlı olarak farklılık gösterebilir ve çoğunlukla exception durumunun etkilerini ortadan kaldırmak veya minimize etmek için tasarlanmıştır.

Bir işlemcinin çalışması sırasında, sisteme bağlı cihazlar tarafından tetiklenen asenkron olaylar, kesintiler (interrupts) olarak adlandırılır. Kesintiler, zamanlayıcılar, harici G/Ç cihazları veya diğer sistem bileşenleri tarafından tetiklenebilir ve işlemcinin anında dikkat gerektiren olaylarla ilgilenmesini gerektirirler. Kesinti meydana geldiğinde, işlemci mevcut çalışma durumunu kaydederek kesinti işleyicisine kontrolü aktarır. Kesinti işleyicisi, kesintinin kaynağına göre özelleştirilmiş bir işlevdir ve işlemci tarafından yürütülür. İşlemci, kesinti işleyicisi tarafından işlendikten sonra, kesinti öncesindeki işlemlerine devam eder.

Bellek sistem hataları, bellek erişim hataları veya izin hataları olarak adlandırılan hatalar, işlemcinin bellek adreslerini çözümleyemediği veya yeterli izinlere sahip olmadığı durumlarda ortaya çıkar. Tanımlanmayan talimat-instructrlar ise işlemcinin tanıyamadığı veya yürütemediği talimat-instructrlardır ve genellikle mevcut çalışma

durumuna uygun olmayan veya kullanılmayan bir işlem yapmak istediği durumlarda ortaya çıkarlar. Sistem çağrıları ise "supervisor çağrıları" veya "SVC" olarak da bilinirler ve ayrıcalıklı yazılım tarafından kullanılarak işletim sisteminden hizmetler istenir. Bu çağrılar işletim sistemi tarafından işlenir ve istenen hizmetler gerçekleştirilir.

Exceptionnın Exception seviyesi, exceptionnın kaynağına ve işlemcinin mevcut Exception seviyesine bağlıdır. Genellikle, daha yüksek öncelikli exceptionlar daha yüksek Exception seviyelerinde meydana gelir ve işlemci bir exception meydana geldiğinde daha yüksek bir Exception seviyesine geçer. ARM mimarisi, dört Exception seviyesi tanımlar: EL0, EL1, EL2 ve EL3. EL0 en az ayrıcalıklı seviye iken EL3 en çok ayrıcalıklı seviyedir. EL0'da çalışan yazılım, yalnızca sınırlı sayıda ayrıcalıklı talimat-instructrı yürütebilir. İşletim sistemi hizmetleri gibi ayrıcalıklı işlemler için, SVC talimat-instructrı kullanılır. Bu talimat-instructr, EL0'da yürütüldüğünde, Supervisor Call (SVC) exceptionsı oluşturur ve kontrol, işletim sistemiye aktarılır. İşletim sistemi, istenen hizmeti gerçekleştirebilir ve kontrolü çağıran programa geri döndürebilir.

SVC talimat-instructrı, işletim sistemi tarafından sağlanan ayrıcalıklı hizmetlere erişmek için kullanılan bir talimat-instructrtır. EL0'da çalışan bir program, yalnızca sınırlı sayıda ayrıcalıklı talimat-instructr yürütebilir ve ayrıcalıklı işlemler için SVC talimat-instructrı kullanır. Bir SVC talimat-instructrı yürütüldüğünde, işlemci Supervisor

Call (SVC) exceptionsı oluşturur ve kontrol, ilgili Exception seviyesine geçerek işletim sistemiye aktarılır. İşletim sistemi, istenen hizmeti gerçekleştirir ve kontrolü çağıran programa geri döndürür. SVC exceptionları, işletim sistemi tarafından işlenir ve istenen hizmeti gerçekleştirmek için özelleştirilmiş işlevler kullanılır. SVC talimat-instructrı, ayrıcalıklı bir talimat-instructr olduğu için yalnızca daha yüksek Exception seviyelerinde (EL1, EL2 ve EL3) kullanılabilir. EL0'da yürütülen bir SVC talimat-instructrı, Supervisor Call (SVC) exceptionsı oluşturarak, işlemcinin Exception seviyesini yükseltmesini sağlar ve kontrolü işletim sistemiye aktarır [10].

5. AARCH64 UYGULAMA SEVİYESİ BELLEK MODELİ

5.1 Adres Uzayı

ARM mimarisi yazılım tarafından kullanılan sanal-virtual adresleri bellekteki fiziksel adreslere eşleştirmek için sanal-virtual bellek sistemi kullanır. Bu eşleştirme Bellek Yönetim Birimi (MMU) tarafından gerçekleştirilir ve yazılımın bellek sistemi üzerinde daha fazla kontrol sahibi olmasını sağlar. MMU, bellekteki sanal-virtual adresleri bellekteki fiziksel adreslere dönüştürür, böylece bellek erişimleri doğru şekilde gerçekleştirilebilir. Adres etiketleme kullanımı, yazılımın bellek sistemi için ek bir koruma seviyesi eklemesine olanak tanır. Yazılım, hafıza erişimlerinin belirli bir adres aralığıyla daraltığından emin olmak için üst sekiz adres bitini bir tag olarak kullanır. Bu, kritik sistem verilerinin korunması veya kötü amaçlı yazılımın belirli bellek alanlarına erişmesinin engellenmesi gibi amaçlar için kullanılabilir.

Bellek erişimleri, ARM mimarisinde Mem[] fonksiyonu kullanılarak gerçekleştirilir. Bu fonksiyon, erişilecek bellek konumunun sanal-virtual adresi, erişim boyutu ve erişim türü gibi parametreleri alır. MMU, sanal-virtual adresi fiziksel adrese çevirir ve bellek erişimini gerçekleştirir. Ancak, adres hesaplamalarının 2^64'e göre mod alınması, adres aralığının taşması veya eksilmesi durumunda bilinmeyen bir adres elde edilmesi anlamına gelir. Bu durum, hesaplanan adres geçersiz olduğunda bellek erişim hatası oluşabilir. Bellek erişim hatası, işlemci tarafından algılanır ve işlemci, ilgili kesinti işleyicisine kontrolü aktararak hatayı işler veya işletim sistemi tarafından yönetilen bir hata durumu olabilir.

Bellek erişim hataları, yazılımın doğru şekilde çalışmasını engelleyebilir veya sistemde istenmeyen davranışlara neden olabilir. Bu nedenle, bellek erişimi işlemlerinin doğru şekilde gerçekleştirilmesi ve adres hesaplamalarının uygun şekilde yapılması büyük önem taşır [10].

```
bits(size*8) Mem[bits(64) address, integer size, AccType acctype]
    assert size IN {1, 2, 4, 8, 16};

Mem[bits(64) address, integer size, AccType acctype] = bits(size*8) value;
The AccType enumeration defines the different access types:
enumeration AccType {AccType_NORMAL, AccType_VEC,       // Normal loads and stores
                     AccType_STREAM, AccType_VECSTREAM, // Streaming loads and stores
                     AccType_ATOMIC,                    // Atomic loads and stores
                     AccType_ORDERED,                   // Load-Acquire and Store-Release
                     AccType_UNPRIV,                    // Load and store unprivileged
                     AccType_IFETCH,                    // Instruction fetch
                     AccType_PTW,                       // Page table walk
                     // Other operations
                     AccType_DC,                        // Data cache maintenance
                     AccType_IC,                        // Instruction cache maintenance
                     AccType_AT};                       // Address translation
```

Şekil 0.1.1 Pseudocode

5.2 Önbellekler ve Bellek Hiyerarşisi

Önbellek, yüksek hızlı bir bellek bloğudur ve birden fazla girdiden oluşur. Her girdi, ana bellek adresi (etiket) ve ilgili veriden oluşur. Önbellekleme, bellek erişimlerinin hızını artırır. Bu işlem, mekansal ve zamansal yerellik prensiplerini kullanarak gerçekleştirilir.

Mekansal yerellik, bir bellek konumuna erişimin ardından bitişik konumlara erişimlerin takip edilme olasılığının yüksek olması prensibidir. Sıralı talimat-instructr yürütme ve veri yapısına erişim gibi durumlar buna örnek verilebilir.

Zamansal yerellik ise, bir bellek konumuna erişimin kısa bir süre içinde tekrarlanma olasılığının yüksek olması prensibidir. Yazılım döngüsü yürütülmesi buna bir örnek verilebilir.

Mekansal yerellik prensibi, aynı etiket altında birden fazla konumu gruplandırır. Bu gruplanmış bellek blokları önbellek satırı olarak adlandırılır. Veri önbelleğe yüklendiğinde, sonraki bellek erişimleri daha hızlı gerçekleşir ve performans artar.

Önbellek kontrol edilir ve önbellek isabeti olup olmadığı kontrol edilir. Eğer erişim önbellek isabeti ise, bellek erişimi önbellekte gerçekleşir. Aksi takdirde, bellek erişimi ana bellekten yapılır. Önbellek konumu tahsis edilir ve önbellek satırı bellekten yüklenir.

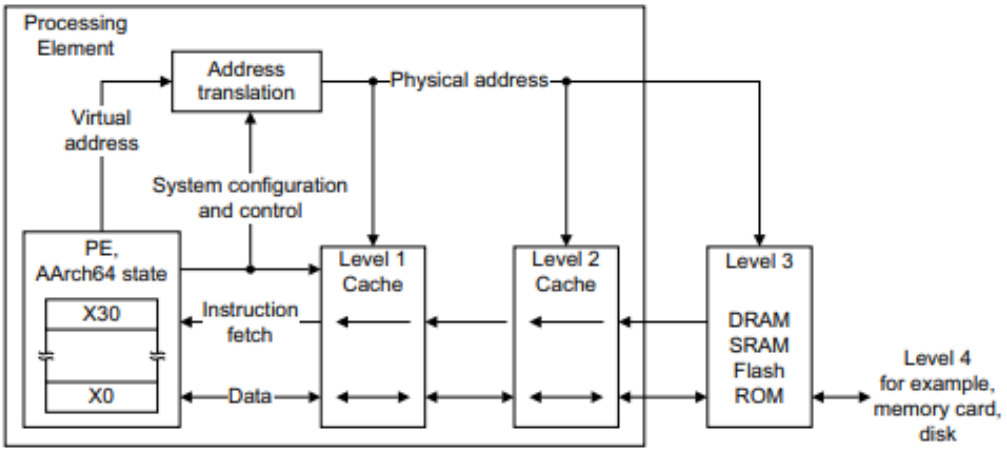
Farklı önbellek topolojileri ve erişim politikaları, ARMv8 bellek tutarlılık modeline uydukları sürece kullanılabilir. Önbellekler genellikle kendini yöneten yapıdadır ve güncellemeler otomatik olarak gerçekleştirilir.

Önbellekler, aşağıdaki nedenlerden dolayı birçok potansiyel sorunu beraberinde getirir:

•Programcının beklediği zamanlardan farklı zamanlarda bellek erişimleri gerçekleşebilir.

• Bir veri öğesi birden fazla fiziksel konumda tutulabilir.

Genellikle PE'ye yakın olan belleklerin gecikmesi çok düşüktür, ancak boyutları sınırlıdır ve maliyetli bir şekilde uygulanır. PE'den daha uzakta, daha büyük bellek blokları uygulamak yaygındır, ancak bunların gecikmesi artar. Genel performansı optimize etmek için, bir ARMv8 bellek sistemi, boyut ve gecikme arasındaki bu dengeyi kullanarak hiyerarşik bir bellek sistemine çoklu önbellek seviyeleri ekleyebilir. 5.2.1 Şekli, sanal-virtual adreslemeyi destekleyen bir ARMv8-A sisteminde böyle bir sistemin örneğini gösterir [10].



Şekil 0.2.1 Multiple Level Of Cache in Memory Hierarchy

6. THE AARCH64 INSTRUCTİON SET

AArch64 yürütme durumunda desteklenen komut kümesine A64 denir. Tüm A64 komutları 32 bit genişliğindedir. A64 kodlama yapısı aşağıdaki işlevsel gruplara ayrılır:

- Birleşik bir grup olarak, exception oluşturan komutlar, sistem komutları ve dallanma komutları içeren çeşitli komutlar bulunur.
- Genel amaçlı Registerlarla ilişkili veri işleme komutları. Bu komutlar, işlemlerin tümünün Registerlarda olup olmadığına veya bir sabit anlık değerle bir işlemin yapılıp yapılmayacağına bağlı olarak iki işlevsel gruba sahiptir.

- Genel amaçlı Register dosyası ile SIMD ve kayan nokta Register dosyası ile ilişkili yükleme ve depolama komutları.
- SIMD ve kayan nokta Registerları üzerinde işlem yapan SIMD ve skaler kayan nokta veri işleme komutları.

Bir fonksiyonel grup, ilgili bir dizi talimat-instructr sınıfından oluşur. A64 talimat-instructrları, sayfa C3-172'deki kodlama endeksinde, talimat-instructr sınıflarının işlevsel grupları içinde bir liste olarak talimat-instructr kodlamalarının genel bir bakışını sağlar.

• Bir talimat-instructr sınıfı, ilgili talimat-instructr formlarının bir kümesinden oluşur. Talimat-instructr formları, iki alfabetik listeden birinde belgelenir:

- Genel amaçlı Registerlarla ilişkili yükleme, depolama ve veri işleme talimat-instructrları ile diğer talimat-instructr sınıflarındakiler. A64 Temel Talimat-instructr Açıklamaları bölümüne bakın.
- SIMD ve kayan nokta destekli yükleme, depolama ve veri işleme talimat-instructrları. A64 SIMD ve Kayan Nokta Talimat-instructr Açıklamaları bölümüne bakın.

• Bir talimat-instructr formu, tek bir talimat-instructr sözdizimini destekleyebilir. Bir talimat-instructrın birden fazla sözdizimini desteklediği durumlarda, her sözdizimi bir talimat-instructr varyantıdır. Talimat-instructr varyantları, işleçlerin boyutu veya biçimi, işleçler için kullanılan Register dosyası, bellek işlemcileri için kullanılan adresleme modu gibi farklılıklar nedeniyle oluşabilir. Talimat-instructr varyantları, bireysel talimat-instructrların açıklamasında belirtilir.

A64 talimat-instructrları düzenli bir bit kodlama yapısına sahiptir: • 5 bitlik Register işlemci alanları, talimat-instructrta sabit pozisyonlarda bulunur. Genel amaçlı Register işlemcileri için, 0-30 değerleri, 31 Registertan birini seçer. 31 değeri, yalnızca bir yığın işaretçisi kaydedici olarak veya sınırlı bir veri işleme talimat-instructrında kullanıldığında bir özel durum olarak kullanılır. Kaynak Register işlemcisi olarak kullanıldığında sıfır değerini, hedef Register işlemcisi olarak

kullanıldığında sonucun atılmasını belirtir. SIMD ve kayan nokta Register erişimi için kullanılan değer, 32 Registertan birini seçer. • Sabit veri işleme değerleri veya adres ofsetleri sağlayan anlık bitler, ardışık bit alanlarına yerleştirilir. Talimat-instructr varyantlarındaki bazı hesaplanmış değerler, ikincil kodlama bit alanları ile birlikte bir veya daha fazla anlık bit alanını kullanır.

Tam olarak tanımlanmayan tüm kodlamalar "TANIMSIZ" olarak tanımlanır. TANIMSIZ bir talimat-instructrın yürütülmesi, Exception modelinde aksi belirtilmedikçe, Tanımsız Talimat-instructr exceptionsına neden olur. A64 komutları, düzenli bir bit kodlama yapısına sahiptir: • 5-bitlik Register operand alanları, komut içinde sabit pozisyonlarda bulunur. Genel amaçlı Register operandları için, 0-30 arasındaki değerler 31 Registertan birini seçer. Değer 31, aşağıdaki durumlarda özel bir durum olarak kullanılabilir: — Bir yükleme/depolama taban kaydırıcısını tanımlarken veya sınırlı sayıda veri işleme komutunda kullanıldığında, geçerli yığın işaretçisini kullanımını gösterir. D1-1416 sayfasındaki yığın işaretçi Registerlarına bakınız. — Bir kaynak Register operandı olarak kullanıldığında sıfır değerini gösterir. — Bir hedef Register operandı olarak kullanıldığında sonucu atmayı gösterir. SIMD ve kayan nokta Register erişimi için, kullanılan değer 32 Registertan birini seçer. • Sabit veri işleme değerleri veya adres ofsetlerini sağlayan anlık bitler, bir arada bit alanlarında yer alır. Komut varyasyonlarında bazı hesaplanmış değerler, ikincil kodlama bit alanlarıyla birlikte bir veya daha fazla anlık bit alanı kullanır. Tam olarak tanımlanmayan tüm kodlamalar UNALLOCATED olarak tanımlanır. UNALLOCATED bir komutun çalıştırılması, İndefined Instruction (Tanımsız Komut) exceptionsı verir, aksi belirtilmedikçe [12,15].

6.1 A64 Assembler Dilinin Yapısı

A64 assembler, hem talimat-instructr mnemoniklerinin hem de Register isimlerinin büyük harfli ve küçük harfli varyantlarını tanır, ancak karışık harfli varyantları tanımaz. A64 bir disassembler, büyük harfli veya küçük harfli mnemonikler ve Register isimleri çıktısı verebilir. Program ve veri etiketleri harf büyüklüğüne duyarlıdır. A64 montaj dili, sabit anlık işleçlerini tanıtmak için # karakterini gerektirmez, ancak bir montajcı, # karakteriyle veya olmadan tanıtılan anlık değerleri

kabul etmelidir. ARM, bir A64 disassembler'ının anlık bir işleçten önce # karakteri çıkarmasını önerir [9].

6.1.1 Ortak Sözdizimi Terimleri

UPPER	Text in upper-case letters is fixed. Text in lower-case letters is variable. This means that register name Xn indicates that the X is required, followed by a variable register number, for example X29.
< >	Any text enclosed by angle braces, < >, is a value that the user supplies. Subsequent text might supply additional information.
{ }	Any item enclosed by curly brackets, { }, is optional. A description of the item and how its presence or absence affects the instruction is normally supplied by subsequent text. In some cases curly braces are actual symbols in the syntax, for example when they surround a register list. These cases are called out in the surrounding text.
[ ]	Any items enclosed by square brackets, [ ], constitute a list of alternative characters. A single one of the characters can be used in that position and the subsequent text describes the meaning of the alternatives. In some case the square brackets are part of the syntax itself, such as addressing modes or vector elements. These cases are called out in the surrounding text.
a|b	Alternative words are separated by a vertical bar, |, and can be surrounded by parentheses to delimit them. For example, U(ADD|SUB)W represents UADDW or USUBW.
±	This indicates an optional + or - sign. If neither is used then + is assumed.
uimm*n*	An *n*-bit unsigned, positive, immediate value.
simm*n*	An *n*-bit two's complement, signed immediate value, where *n* includes the sign bit.
SP	See *Register names* on page C1-114.
Wn	See *Register names* on page C1-114.
WSP	See *Register names* on page C1-114.
WZR	See *Register names* on page C1-114.
Xn	See *Register names* on page C1-114.
XZR	See *Register names* on page C1-114

Şekil 6.2 Syntax Terms

6.1.2 Instruction Mnemonics

A64 assembly dili, talimat-instructr mnemoniklerini operand tiplerine göre ayırt eden işlem kodları arasındaki farklı biçimleri aşırı yükler. Örneğin, aşağıdaki ADD talimat-instructrları farklı işlem kodlarına sahiptir:

ADD X0, X1, X2 ; X1 ve X2'yi topla demektir ve sonucu X0'a yaz anlamını taşır
ADD X0, X1, #42 ; X1'e 42 değerini ekler ve sonucu X0'a yazar

Ancak programcı, assembler otomatik olarak operandlara bağlı olarak doğru işlem kodunu seçtiği için yalnızca bir mnemoniği hatırlamalıdır. Bu, programcının kod yazarken daha az işlem kodu hakkında düşünmesine ve kodun daha okunaklı olmasına olanak tanır.

Tersine, disassembler, makine kodunu açıklayıcı bir A64 koduna dönüştürür. Bu işlemde, disassembler, işlem kodunun operandlarını ve tiplerini analiz ederek doğru talimat-instructr mnemoniğini belirler. Böylece, disassembler, makine kodunu daha okunaklı bir A64 assembly koduna dönüştürür [12].

6.1.3 Condition Kodu

cond	Mnemonic	Meaning (integer)	Meaning (floating-point)[a]	Condition flags
0000	EQ	Equal	Equal	Z == 1
0001	NE	Not equal	Not equal or unordered	Z == 0
0010	CS or HS	Carry set	Greater than, equal, or unordered	C == 1
0011	CC or LO	Carry clear	Less than	C == 0
0100	MI	Minus, negative	Less than	N == 1
0101	PL	Plus, positive or zero	Greater than, equal, or unordered	N == 0
0110	VS	Overflow	Unordered	V == 1
0111	VC	No overflow	Ordered	V == 0
1000	HI	Unsigned higher	Greater than, or unordered	C ==1 && Z == 0
1001	LS	Unsigned lower or same	Less than or equal	!(C ==1 && Z ==0)
1010	GE	Signed greater than or equal	Greater than or equal	N == V
1011	LT	Signed less than	Less than, or unordered	N! = V
1100	GT	Signed greater than	Greater than	Z == 0 && N == V
1101	LE	Signed less than or equal	Less than, equal, or unordered	!(Z == 0 && N == V)
1110	AL	Always	Always	Any
1111	NV[b]	Always	Always	Any

a. Unordered means at least one NaN operand.
b. The condition code NV exists only to provide a valid disassembly of the 0b1111 encoding, otherwise its behavior is identical to AL.

Şekil 6.1.3.1 Conditions

6.1.4 Genel Amaçlı Register File ve Stack Pointer

A64 işlemcinin genel amaçlı Register dosyasında R0-R30 olarak adlandırılan 31 genel amaçlı Register yer alır ve bu Registerların değerleri 0-30 arasındaki sayılarla kodlanır. Genel amaçlı Register alanı değeri 31 olan bir Register, komut ve işlem konumuna bağlı olarak geçerli yığın göstericisini veya sıfır kaydını temsil eder.

Registerlar belirli bir komut varyantında kullanıldığında, operan veri boyutunu, 32 bit veya 64 bit olarak nitelendirilmelidir. Veri boyutu 32 bit olduğunda, kaydın alt 32 biti kullanılır. Okuma işlemi sırasında üst 32 bit yok sayılır ve yazma işlemi sırasında sıfıra ayarlanır. Bu işlem, kaydın tam değerini değiştirmediğinden önemlidir ve verilerin tutarlı kalmasını sağlar [12].

Tablo 2 General Register Name

Table C1-2 General-purpose register names

Name	Size	Encoding	Description
Wn	32 bits	0-30	General-purpose register 0-30
Xn	64 bits	0-30	General-purpose register 0-30
WZR	32 bits	31	Zero register
XZR	64 bits	31	Zero register
WSP	32 bits	31	Current stack pointer
SP	64 bits	31	Current stack pointer

Aşağıdaki liste Tablo C1-2 ile ilgili daha fazla detay sağlar:

• Xn ve Wn isimleri her ikisi de genel amaçlı Register Rn'yi ifade eder.

• W31 veya X31 adında bir Register yoktur.

• SP adı, karşılık gelen Register alanındaki değer 31'in bir kodlaması olarak yorumlandığı durumlarda 64 bitlik işlemler için yığın işaretçisini temsil eder. Bu işlem, bu operand kodlamasını yığın işaretçisi olarak yorumlamayan talimat-instructrlar için SP adının kullanımı bir hatadır.

• WSP adı, 32 bitlik bir bağlamda mevcut yığın işaretçisini temsil eder.

• XZR adı, karşılık gelen Register alanındaki değer 31'in bir kodlaması olarak yorumlandığı 64 bitlik işlemler için sıfır kaydını temsil eder. Bu işlem, bu operand kodlamasını sıfır kaydı olarak yorumlamayan talimat-instructrlar için XZR adının kullanımı bir hatadır.

• WZR adı, 32 bitlik bir bağlamda sıfır kaydını temsil eder.

• Mimari, prosedür çağrıları sırasında link kaydırıcısı olarak özel bir isimlendirme sağlamaz. Bir A64 assembler her zaman W30 ve X30'u kullanmalıdır. İşlem Çağrısı Standartı'nın bir parçası olarak ek yazılım adları tanımlanabilir, bkz. ARM 64-bit Mimarisi için İşlem Çağrısı Standartı [12].

6.1.5 SIMD ve Floating-Point Scalar Register İsimleri

Skaler veri üzerinde çalışan SIMD ve kayan nokta komutları, yalnızca bir SIMD veya kayan nokta kaydının daha düşük bitlerine erişirler. Kullanılmayan üst bitler bir okumada yoksayılır ve bir yazımda sıfırlanır. C1-3 Tablosu, skaler SIMD ve kayan nokta Registerlarına erişmek için nitelikli isimleri gösterir. Harf n, 0 ile 31 arasında bir Register numarasını gösterir [8,12].

Size	Name
8 bits	Bn
16 bits	Hn
32 bits	Sn
64 bits	Dn
128 bits	Qn

Şekil 6.1.5.1 SIMD and Floating Point Scalar Register Names

6.1.6 SIMD Vector Register İsimleri

Tablo 3 SIMD Vector Registers

Table C1-4 SIMD vector register names

Shape	Name
8 bits × 8 lanes	Vn.8B
8 bits × 16 lanes	Vn.16B
16 bits × 4 lanes	Vn.4H
16 bits × 8 lanes	Vn.8H
32 bits × 2 lanes	Vn.2S
32 bits × 4 lanes	Vn.4S
64 bits × 1 lane	Vn.1D
64 bits × 2 lanes	Vn.2D

Bir Register, paralel SIMD şeklinde işlem yapılan birden fazla veri öğesi içeriyorsa, bir nitelik vektör şeklini tanımlar. Vektör şekli, öğe boyutu ve öğe sayısı veya şerit sayısıdır. Öğe boyutu bit cinsinden çarpılan şerit sayısı 128'e eşit olmazsa, kaydın üstteki 64 biti okumada dikkate alınmaz ve yazarken sıfırlanır [8,12].

6.2 A64 Komut Kümesi Genel Bakışı

Koşullu atlamalar, işlem durum bayraklarının veya genel amaçlı bir Registertaki değerin mevcut durumuna bağlı olarak yürütmenin akışını değiştirir.

Tablo 4 Conditional Branch

Table C2-1 Conditional branch instructions

Mnemonic	Instruction	Branch offset range from the PC	See
B.cond	Branch conditionally	±1MB	*B.cond* on page C5-420
CBNZ	Compare and branch if nonzero	±1MB	*CBNZ* on page C5-434
CBZ	Compare and branch if zero	±1MB	*CBZ* on page C5-435
TBNZ	Test bit and branch if nonzero	±32KB	*TBNZ* on page C5-754
TBZ	Test bit and branch if zero	±32KB	*TBZ* on page C5-755

Koşulsuz dallanma (anlık) komutları, işlemcinin okuduğu komut sayacının değerine ±128MB aralığında bir hemen etki değeri ekleyerek koşulsuz olarak işlem akışını

değiştirir. BL komutu ayrıca ardışık olarak takip eden komutun adresini genel amaçlı X30 kaydedicisine yazar.

Tablo 5 Unconditional Branch

Table C2-2 Unconditional branch instructions (immediate)

Mnemonic	Instruction	Immediate branch offset range from the PC	See
B	Branch unconditionally	±128MB	*B* on page C5-421
BL	Branch with link	±128MB	*BL* on page C5-430

Registerlı bir genel amaçlı kaydediciye atanan değerle program sayaçını değiştirerek, koşulsuz dallanma (register) komutları, yürütmenin akışını koşulsuz bir şekilde değiştirirler. BLR komutu, ardışık izleyen komutun adresini genel amaçlı kaydedici X30'a yazarak çalışır. RET komutu BR ile aynı şekilde davranır, ancak PE'ye bu komutun bir alt rutineden dönüş olduğu konusunda ek bir ipucu sağlar. C2-3 tablosu, genel amaçlı bir kaydedicide tutulan adrese doğrudan atlama yapan koşulsuz dallanma komutlarını göstermektedir.

Tablo 6 Unconditional Branch Instructions

Table C2-3 Unconditional branch instructions (register)

Mnemonic	Instruction	See
BLR	Branch with link to register	*BLR* on page C5-431
BR	Branch to register	*BR* on page C5-432
RET	Return from subroutine	*RET* on page C5-642

Tablo 7 Exception generating instructions

Table C2-4 Exception generating instructions

Mnemonic	Instruction	See
BRK	Software breakpoint instruction	*BRK* on page C5-433
HLT	Halting software breakpoint instruction	*HLT* on page C5-484
HVC	Generate exception targeting Exception level 2	*HVC* on page C5-485
SMC	Generate exception targeting Exception level 3	*SMC* on page C5-663
SVC	Generate exception targeting Exception level 1	*SVC* on page C5-748

Tablo 8 Debug state instructions

Table C2-6 Debug state instructions

Mnemonic	Instruction	See
DCPS1	Debug switch to Exception level 1	*DCPS1* on page C5-466
DCPS2	Debug switch to Exception level 2	*DCPS2* on page C5-467
DCPS3	Debug switch to Exception level 3	*DCPS3* on page C5-468
DRPS	Debug restore PE state	*DRPS* on page C5-471

[12,15]

6.3 AArch64 Sistem Talimat-instructr Sınıfı

Register açıklamalarında, bazı bitlerin anlamı PE durumuna bağlıdır. Sabit değerlere sahip bitleri veya alanları tanımlamak için şu terimler kullanılır: RAZ Herhangi bir uygulamada:

Alanı için bir "SBZP" uygulanabilir. Şemalarda, RAZ biti 0 olarak gösterilebilir. RES0 Şemalarda ve bazen diğer tanımlarda, RES0 biti (0) olarak gösterilebilir. Bu gösterim, bit alanları için genişletilebilir, böylece üç bitlik bir RES0 alanı, ya (0)(0)(0) ya da (000). olarak gösterilebilir. Mimari içinde, bir Register biti veya bit alanı için:

• Belirli bir mimari bağlamında RES0'dır.

• Farklı bir mimari bağlamda farklı tanımlanmış davranışı vardır. RES0 tanımı bu bitler için değiştirilir. Bu, RES0 tanımının şu şekilde olduğu anlamına gelir: Eğer bir bit tüm bağlamlarda RES0 ise: bitler sıfır okunur ve alana yazılanlar görülmez.

Eğer bir Register biti yalnızca bazı bağlamlarda RES0 ise, o bit RES0 olarak tanımlanırken:

• Bir bitin okunması, yazıldığı değeri, bitin yazıldığı sırada kaydettiği değer dikkate alınarak return eder. Eğer bit resetten bu yana düzgün yazılmamışsa, o zaman bitin okunması sıfırlama değerini return ederse, aksi takdirde bilinmeyen bir değer gösterir.

• Bir bitin yazılması, bitle ilişkili bir bellek konumunu güncellemelidir.

• Kaydın kullanımı, bit RES0 olarak tanımlanacak şekilde olduğunda, bitin değeri, bitin okunduğu değeri belirlemek dışında, PE'nin işleyişini etkilememelidir.

• Yazılım kısmı: Alanı yazmak için SBZP kullanılır ve alanın sıfır olacağı kesin görülmemelidir.

RES0 tanımı salt okunur olan bitler veya sadece yazılabilir olan bit alanları için uygulanabilir:

• Salt okunur bir bit için, RES0 bitin 0 olarak okunduğunu gösterir, ancak yazılımın biti BİLİNMEYEN olarak işlemesi gerektiğini belirtir.

• Yalnızca yazılabilen bir bit için, RES0 yazılımın biti SBZ olarak işlemesi gerektiğini belirtir.

RAO herhangi bir uygulamada:

• Alanda tüm 1'ler olarak görülür.

• Alana yapılan yazılımlar yok sayılmalıdır.

• Yazılım kısmında: Alana için SBOP kullanmalıdır. Şemalarda, bir RAO biti 1 olarak gösterilmelidir. RES1 tanımı, salt okunur olan bitler veya sadece yazılabilir olan bit alanları için uygulanabilir:

• Salt okunur bir bit için, RES1 bitin 1 olarak okunduğunu gösterir, ancak yazılımın biti BİLİNMEYEN olarak işlemesi gerektiğini belirtir.

• Yalnızca yazılabilen bir bit için, RES1 yazılımın biti SBO olarak işlemesi gerektiğini söyler.

Bitin tüm bağlamlarda RES1 olarak belirtildiği kısımlarda

• Bit 1 olarak görülür.

Eğer bir Register biti sadece bazı bağlamlarda RES1 ise, o zaman bu bit RES1 olarak tanımlandığında

• Bitin okunması, bitin yazıldığı sırada Register kullanımından bağımsız olarak son başarılı yazma değerini return eder. Eğer bit sıfırlama sonrası başarılı bir şekilde yazılmamışsa, o zaman bitin okunması sıfırlama değerini döndürür, aksi takdirde BİLİNMEYEN bir değer return eder.

• Bir bitin yazılması, bitle ilgili bir bellek konumunu güncellenir.

• Register kullanımı, bitin RES1 olarak tanımlanması durumunda, bitin değerinin, o bitin okunduğu değeri belirlemek dışında, PE işlemine etkisi olmamalıdır [12,15].

6.4 Prosedür Çağrısı Standartları

Arm mimarisi, genel amaçlı Registerların kullanımı için belirli kısıtlamalar getirmez. Tamsayı Registerları ve kayan nokta Registerları genel amaçlı Registerlar arasında yer alır. Ancak, kodunuzun başka bir kaynak koduyla veya derleyicinin ürettiği kodla etkileşimde bulunması ihityaçda, Register kullanımı için belli kurallar lisresli belirlenir. Arm mimarisinde bunlar Procedure Call Standard (Prosedür Çağrı Standartları) olarak geçer. Prosedür Çağrı Standartları şunları belirtir:

1. Hangi Registerların argümanları işleve aktarmak için kullanıldığı.
2. Hangi Registerların işlevi çağıran işlevin geri dönüş değerini almak için kullandığı, bu işlev çağıran olarak bilinir.
3. Hangi Registerların işlevin çağrısı yapılana, yani callee olarak bilinen işleve zarar verebileceği.
4. Hangi Registerların callee tarafından zarar verilemeyeceği.

Örneğin, main() işlevinden çağrılan foo() işlevi düşünüldüğünde:

Procedure Call Standard (PCS), argümanların fonksiyona nasıl geçirileceğini belirler. Arm mimarisinde, ilk 8 argüman X0'dan X7'ye kadar sırasıyla kaydedilir. Daha fazla argüman, yığın üzerinden geçirilir. Bu nedenle, foo() fonksiyonu 2 argüman alır (b ve c) ve b W0'da c W1'de olacak [14].

6.5 Sistem Çağrıları

A64 işlemcide, özel komutlar kullanılarak systwm çağrıları yapılabilir. Bu komutlar exception oluşturarak daha ayrıcalıklı bir Exception seviyesine kontrollü bir giriş sağlar. SVC (Supervisor Call) komutu, EL1 hedef Exception seviyesine yönelik bir exception oluşturur ve bir uygulamanın işletim sistemi çağırmak için kullanılır. HVC (Hypervisor Call) komutu, EL2 hedef Exception seviyesine yönelik bir exception oluşturur ve bir işletim sisteminde hipervizörü çağırır. Ancak EL0 seviyesinde değildir. SMC (Secure Monitor Call) komutu ise EL3 hedef Exception seviyesine yönelik bir exception oluşturur ve bir işletim sistemi veya hipervizör, EL3 yazılımını çağırmak için kullanır. EL0 seviyesinde kullanılamaz. Bir Exception, hedef Exception seviyesinden daha yüksek bir Exception seviyesinde yürütülürse, Exception geçerli Exception seviyesine alınır. Örneğin, bir SVC EL2'de exception girişine neden olurken, bir HVC EL3'te exception girişine neden olur. İşlemcinin ayrıcalığını kaybetmesine neden olamayacağı kuralıyla tutarlıdır [14].

7. ARMV8-M MİMARİSİ KURALLARI

7.1 Resets, Cold reset, and Warm reset

RBDPL Armv8.0-M mimarisi veya sonrasındaki bir uygulama için geçerli olan iki sıfırlama türü vardır:

• Cold Reset (Soğuk sıfırlama)

• Warm Reset (Ilık sıfırlama)

Armv8.0-M mimarisi veya sonrasındaki uygulamalar için geçerli olan bazı bellek işlemleri ve sıfırlama yöntemleriyle ilgilidir.

- "Warm sıfırlama olmadan soğuk sıfırlama yapmak mümkün değildir" ifadesi, soğuk sıfırlamanın ilik sıfırlamayla birlikte kullanılması gerektiğini belirtiyor.
- "Cold sıfırlamada, tanımlanmış bir sıfırlama değerine sahip olan Registerlar o değeri içerirler" ifadesi, soğuk sıfırlama işleminin belirli bir sıfırlama değeri kullanarak gerçekleştirildiğini söylüyor.
- "Warm sıfırlamada, tanımlanmış bir sıfırlama değeri olan bazı hata ayıklama kaydı kontrol alanları değiştirilmez ancak diğer tüm Registerlar tanımlanmış sıfırlama değerini içerirler" ifadesi, ilik sıfırlama işleminin de belirli bir sıfırlama değeri kullanarak gerçekleştirildiğini ancak bazı özel bellek kayıtlarının bu işlemden etkilenmediğini söylüyor.
- "Warm sıfırlamada, PE TakeReset() sözdiziminde tanımlandığı şekilde hareket eder" ifadesi, ilik sıfırlamanın belirli bir programlama sözdizimi kullanılarak gerçekleştirildiğini söylüyor.
- "AIRCR.SYSRESETREQ, bir warm sıfırlama isteği göndermek için kullanılır" ifadesi, ilik sıfırlama işleminin gerçekleştirilmesi için AIRCR.SYSRESETREQ adlı bir bellek kaydının kullanıldığını belirtiyor.

RCTPC Ilık sıfırlama olmadan cold sıfırlama yapmak mümkün değildir. Armv8.0-M mimarisi veya sonrasındaki bir uygulama için geçerlidir.

RFNNX cold sıfırlamada, tanımlanmış bir sıfırlama değerine sahip olan Registerlar o değeri içerirler. Armv8.0-M mimarisinde ya da sonrasındaki uygulamalarda için söylenebilir.

RGTXW warm sıfırlamada, tanımlanmış bir sıfırlama değeri olan bazı hata ayıklama kaydı kontrol alanları değiştirilmez ancak diğer tüm Registerlar tanımlanmış sıfırlama değerini içerirler. Armv8.0-M mimarisinde ya da sonrasındaki uygulamalarda için söylenebilir.

RYMHN warm sıfırlamada, PE TakeReset() sözdiziminde tanımlandığı şekilde hareket eder. Armv8.0-M mimarisi veya sonrasındaki bir uygulama için geçerlidir.

RWSZN AIRCR.SYSRESETREQ, bir warms sıfırlama isteği göndermek için kullanılır. Armv8.0-M mimarisi veya sonrasındaki bir uygulama için geçerlidir.

RHFRS AIRCR.SYSRESETREQ için, mimari sıfırlamanın hemen gerçekleşeceğini garanti etmez. Armv8.0-M mimarisi veya sonrasındaki bir uygulama için geçerlidir [13].

8. AARCH64 REGİSTER FONKSİYONLARI

Arm developer sitesine göre birçok pseudocode örneği bulunmaktadır. Çok fazla olduğu için ben birkaç tanesini örnek vereceğim:

Bu kod, ARMv8-A mimarisindeki bir A64 assembly programı için yazılmış bir C fonksiyonunu temsil etmektedir. Fonksiyon, verilen bellek adresinden başlayarak belirtilen boyut kadar bellek bölgesini sıfırlar.

Fonksiyonun adı "**AArch64.DataMemZero"** ve C dilindeki imzası aşağıdaki gibidir:

Fonksiyonun giriş parametreleri şunlardır:

- **regval**: İşlem yürütülürken kullanılan bir kayıt değeridir.
- **vaddress**: Sıfırlanacak bellek bölgesinin sanal-virtual adresidir.
- **accdesc_in**: Bellek erişimini açıklayan bir yapıdır.

- **size**: Sıfırlanacak bellek bölgesinin boyutudur.

Fonksiyonun işlevi, verilen bellek adresinden başlayarak belirtilen boyutta bir bellek bölgesi sıfırlamaktır. İşlemin gerçekleştirilmesi için, öncelikle "**AArch64.TranslateAddress"** fonksiyonu kullanılarak sanal-virtual bellek adresi fiziksel bellek adresine dönüştürülür. Daha sonra, bellek bölgesi boyutuna göre bir döngü oluşturulur ve her döngü adımında bellek bölgesinin bir byte'ı sıfırlanır. Bellek erişimi sırasında, etkinleştirilmiş olan bellek özelliklerine ve sistem yapılandırmasına bağlı olarak bellek erişimleri kontrol edilir. Ayrıca, bellek özellikleri doğrulanır ve etiket denetimi yapılır. Herhangi bir hata durumunda, işlem abort edilir. Son olarak, sıfırlama işlemi tamamlandığında fonksiyon sonlanır.

"AArch64.TagMemZero()"

Bu fonksiyon, AArch64 mimarisinde "tag" olarak adlandırılan bellek bölgesine sıfır yazmak için kullanılır. AArch64 mimarisinde bellek bölgesi iki ayrı parçaya ayrılır: veri belleği ve etiket belleği. Veri belleği, gerçek verilerin depolandığı bölgedir. Etiket belleği, her bir veri belleği bloğuna atanan bir etiketi içerir. Etiketler, bir veri bloğunun hangi amaçla kullanıldığını belirlemeye yardımcı olur ve bu blokların hangi işlemci kaynağı tarafından kullanıldığını izlemek için kullanılır.

Bu fonksiyon, AArch64 mimarisinde bir bellek bölgesine sıfır etiket yazmak için kullanılır. Fonksiyon, verilen bellek adresini alır ve bellek adresindeki etiket bölgesine sıfır yazar. Fonksiyon, işlemci kaynağı ve bellek adresindeki etiketin uygun olup olmadığını doğrular ve ardından etiket bölgesine sıfır yazar.

CompareOp, vektör işlemlerini destekleyen bilgisayar mimarilerinde bulunan farklı türdeki vektör karşılaştırma işlemlerini temsil eden bir numaralandırma türüdür. Beş farklı değer alabilir: CompareOp_GT (büyüktür), CompareOp_GE (büyük veya eşittir), CompareOp_EQ (eşittir), CompareOp_LE (küçük veya eşittir), CompareOp_LT (küçüktür).

CountOp, sayma (bit counting) işlemlerini destekleyen bilgisayar mimarilerinde bulunan farklı türdeki bit sayma işlemlerini temsil eden bir numaralandırma türüdür. Üç farklı değer alabilir: **CountOp_CLZ** (sol taraftaki sıfır bitlerin sayısını verir),

CountOp_CLS (sol taraftaki bir bitin ardından gelen sıfır bitlerin sayısını verir) ve **CountOp_CNT** (birlerin sayısını verir).

AArch64 (ARMv8) işlemci mimarisinde kullanılan bir işlevi tanımlar. Fonksiyonun adı **IsD128Enabled()** ve 128-bit sayfa tanımlayıcısının (page descriptor) etkin olup olmadığını kontrol eder. Bu, bellek yönetimi için önemlidir çünkü 128-bit sayfa tanımlayıcıları, daha büyük bir sanal-virtual adres uzayı sağlar ve daha yüksek performanslı ve daha etkili bellek yönetimi için gereklidir.

AArch64.DC() adında bir fonksiyon tanımlar. Bu fonksiyon bir Veri Önbelleği işlemi gerçekleştirir. Fonksiyon dört parametre alır: 64 bitlik bir kayıt değeri, bir CacheType numaralandırma değeri, bir CacheOp numaralandırma değeri ve bir CacheOpScope numaralandırma değeri.

Fonksiyon, önbellek parametreleri ile bir CacheRecord yapısını başlatır ve işlem kapsamının CacheOpScope_SetWay olarak ayarlanıp ayarlanmadığını kontrol eder. Eğer öyleyse, kayıt değerini çözümler, mevcut özel durum seviyesindeki güvenlik durumunu belirler ve gerekiyorsa önbellek işlemini CacheOp_CleanInvalidate olarak ayarlar. Daha sonra önbellek işlemini gerçekleştirir ve döner [17].

Bunun gibi sayfalarca örnek bulunmaktadır.

9.SONUÇ

Bu araştırmamda ele alınan konular, ARM işlemcilerin evrimi, ARM64 temelleri, ARMv8 mimarisinde desteklenen veri türleri, AArch64 uygulama düzeyi programcı modeli, bellek modeli, komut seti ve ARMv8-M mimarisindeki bazı temel kurallar gibi konuları içermektedir. Mikro işlemci teknolojileri, özellikle ARM mimarisi, günümüzde kullanılan cihazların çoğunda yaygın olarak kullanılmaktadır. ARM işlemciler, düşük güç tüketimi, yüksek performans ve yüksek verimlilik özellikleri nedeniyle özellikle mobil cihazlarda, akıllı ev aletlerinde, akıllı saatlerde, arabalarda ve daha birçok cihazda tercih edilmektedir.

ARM işlemcilerin en önemli özelliklerinden biri, geniş bir Register setine sahip olmalarıdır. ARM64 registerları, 64 bitlik bir mimariye sahiptir ve programcıların daha fazla veri işlemesi yapmalarını sağlar. Ayrıca, ARM64 registerları, daha yüksek sayıda Register kullanımı sayesinde daha az bellek erişimine ihtiyaç duyar, bu da performans açısından büyük bir avantajdır.

Sonuç olarak, ARM mimarisi, mikro işlemci teknolojileri arasında en önemli seçeneklerden biridir ve ARM64 registerları, yüksek performans, düşük güç tüketimi ve yüksek verimlilik özellikleri nedeniyle çok yönlü kullanım alanları sunar. Bu araştırmada ele alınan konuların anlaşılması, ARM işlemcilerin tasarım ve geliştirme sürecinde önemli bir rol oynamaktadır.

KAYNAKLAR

1. C. Bellew, "The ARMv8 CPU architecture in a nutshell," Medium, 14-Jul-2019. [Online]. Available: https://cbellew.medium.com/the-armv8-cpu-architecture-in-a-nutshell-82010ce6fbb6. [Accessed: 07-May-2023].

2. D. Brash, "A Brief History of ARM: Part 2," Arm Community Blogs, Mar. 18, 2019. [Online]. Available: https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/a-brief-history-of-arm-part-2. [Accessed: May. 7, 2023].

3. N. Mohammed, N. Al-Ajlan and A. Hussain, "Intel x86 and ARM Processors: A Survey on Architectural Differences," 2014 12th International Conference on Innovations in Information Technology (IIT), Al Ain, United Arab Emirates, 2014, pp. 1-6. doi: 10.1109/Innovations.2014.6970423.

4. D. A. Patterson and D. R. Ditzel, "The case for the reduced instruction set computer," ACM SIGARCH Computer Architecture News, vol. 8, no. 6, pp. 25-33, 1980. doi: 10.1145/641914.641917.

5. J. L. Hennessy and D. A. Patterson, "Computer Architecture: A Quantitative Approach," Elsevier, 2011

6. D. Vostokov, "Foundations of ARM64 Linux Debugging, Disassembling, and Reversing: Analyze Code, Understand Stack Memory Usage, and Reconstruct Original C/C++ Code with ARM64," 1st ed., OpenTask, 2021.

7. "List of ARM Processors," Wikipedia, accessed May 7, 2023. [Online]. Available: https://en.wikipedia.org/wiki/List_of_ARM_processors#References.

8. L. D. Pyeatt and W. Ughetta, "ARM 64-Bit Assembly Language," 1st ed., CRC Press, 2017.

9. ARM Holdings, "ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile Beta," ARM Holdings, 2014.

10. ARM Holdings, "Arm Architecture Reference Manual Supplement Armv8, for R-profile AArch64 architecture," ARM Holdings, 2021.

11. ARM Holdings, "Arm Architecture Reference Manual Supplement ARMv8.1, for ARMv8-A architecture profile," ARM Holdings, 2017.

12. ARM Holdings, "Arm Architecture Reference Manual Supplement, Custom Datapath Extension for Armv8-M," ARM Holdings, 2020

13. ARM Holdings, "Arm v8-M Architecture Reference Manual," Document number DDI0553B.v, Document version ID16122022, Document confidentiality Non-confidential.

14. Arm Limited, "Procedure Call Standard for the Arm® 64-bit Architecture (AArch64)," Document number DEN0028A, Document version D, Arm Limited, 2021.

15.ARM Developer Documentation [Online]: https://developer.arm.com/documentation [Accessed: May. 7, 2023].

16. Arm Limited. (2023, March). AArch64 Registers. Retrieved from https://developer.arm.com/documentation/ddi0601/2023-03/AArch64-Registers?lang=en

17. ARM. "AArch64 Functions - eret." Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile, Arm Limited, March 2023, https://developer.arm.com/documentation/ddi0597/2023-03/Shared-seudocode/aarch64-functions-eret?lang=en.

EKLER

EK-1. ARMv8 A64 Quick Reference

ARMv8 A64 Quick Reference

Arithmetic Instructions			
ADC{S}	rd, rn, rm	rd = rn + rm + C	
ADD{S}	rd, rn, op2	rd = rn + op2	S
ADR	Xd, ±rel_{21}	Xd = PC + $rel^{\pm}$	
ADRP	Xd, ±rel_{33}	Xd = $PC_{63:12}$:0_{12} + $rel^{\pm}_{33:12}$:0_{12}	
CMN	rd, op2	rd + op2	S
CMP	rd, op2	rd − op2	S
MADD	rd, rn, rm, ra	rd = ra + rn × rm	
MNEG	rd, rn, rm	rd = − rn × rm	
MSUB	rd, rn, rm, ra	rd = ra − rn × rm	
MUL	rd, rn, rm	rd = rn × rm	
NEG{S}	rd, op2	rd = −op2	
NGC{S}	rd, rm	rd = −rm − ∼C	
SBC{S}	rd, rn, rm	rd = rn − rm − ∼C	
SDIV	rd, rn, rm	rd = rn $\tilde{\div}$ rm	
SMADDL	Xd, Wn, Wm, Xa	Xd = Xa + Wn $\tilde{\times}$ Wm	
SMNEGL	Xd, Wn, Wm	Xd = − Wn $\tilde{\times}$ Wm	
SMSUBL	Xd, Wn, Wm, Xa	Xd = Xa − Wn $\tilde{\times}$ Wm	
SMULH	Xd, Xn, Xm	Xd = (Xn $\tilde{\times}$ Xm)$_{127:64}$	
SMULL	Xd, Wn, Wm	Xd = Wn $\tilde{\times}$ Wm	
SUB{S}	rd, rn, op2	rd = rn - op2	S
UDIV	rd, rn, rm	rd = rn ÷ rm	
UMADDL	Xd, Wn, Wm, Xa	Xd = Xa + Wn × Wm	
UMNEGL	Xd, Wn, Wm	Xd = − Wn × Wm	
UMSUBL	Xd, Wn, Wm, Xa	Xd = Xa − Wn × Wm	
UMULH	Xd, Xn, Xm	Xd = (Xn × Xm)$_{127:64}$	
UMULL	Xd, Wn, Wm	Xd = Wn × Wm	

Bit Manipulation Instructions			
BFI	rd, rn, #p, #n	$rd_{p+n-1:p}$ = $rn_{n-1:0}$	
BFXIL	rd, rn, #p, #n	$rd_{n-1:0}$ = $rn_{p+n-1:p}$	
CLS	rd, rn	rd = CountLeadingOnes(rn)	
CLZ	rd, rn	rd = CountLeadingZeros(rn)	
EXTR	rd, rn, rm, #p	rd = $rn_{p-1:0}$:rm_{N0}	
RBIT	rd, rn	rd = ReverseBits(rn)	
REV	rd, rn	rd = BSwap(rn)	
REV16	rd, rn	for(n=0..1|3) rd_{Hn}=BSwap(rn_{Hn})	
REV32	Xd, Xn	Xd=BSwap($Xn_{63:32}$):BSwap($Xn_{31:0}$)	
{S,U}BFIZ	rd, rn, #p, #n	rd = $rn^{?}_{n-1:0}$ ≪ p	
{S,U}BFX	rd, rn, #p, #n	rd = $rn^{?}_{p+n-1:p}$	
{S,U}XT{B,H}	rd, Wn	rd = $Wn^{?}_{N0}$	
SXTW	Xd, Wn	Xd = $Wn^{\pm}$	

Logical and Move Instructions			
AND{S}	rd, rn, op2	rd = rn & op2	
ASR	rd, rn, rm	rd = rn $\tilde{\gg}$ rm	
ASR	rd, rn, #i_6	rd = rn $\tilde{\gg}$ i	
BIC{S}	rd, rn, op2	rd = rn & ∼op2	
EON	rd, rn, op2	rd = rn ⊕ ∼op2	
EOR	rd, rn, op2	rd = rn ⊕ op2	
LSL	rd, rn, rm	rd = rn ≪ rm	
LSL	rd, rn, #i_6	rd = rn ≪ i	
LSR	rd, rn, rm	rd = rn ≫ rm	
LSR	rd, rn, #i_6	rd = rn ≫ i	
MOV	rd, rn	rd = rn	S
MOV	rd, #i	rd = i	
MOVK	rd,#i_{16}{, sh}	$rd_{sh+15:sh}$ = i	
MOVN	rd,#i_{16}{, sh}	rd = ∼($i^{\emptyset}$ ≪ sh)	
MOVZ	rd,#i_{16}{, sh}	rd = $i^{\emptyset}$ ≪ sh	
MVN	rd, op2	rd = ∼op2	
ORN	rd, rn, op2	rd = rn | ∼op2	
ORR	rd, rn, op2	rd = rn | op2	
ROR	rd, rn, #i_6	rd = rn ⋙ i	
ROR	rd, rn, rm	rd = rn ⋙ rm	
TST	rn, op2	rn & op2	

Branch Instructions			
B	rel_{28}	PC = PC + $rel^{\pm}_{27:2}$:0_2	
Bcc	rel_{21}	if(cc) PC = PC + $rel^{\pm}_{20:2}$:0_2	
BL	rel_{28}	X30 = PC + 4; PC += $rel^{\pm}_{27:2}$:0_2	
BLR	Xn	X30 = PC + 4; PC = Xn	
BR	Xn	PC = Xn	
CBNZ	rn, rel_{21}	if(rn ≠ 0) PC += $rel^{\emptyset}_{21:2}$:0_2	
CBZ	rn, rel_{21}	if(rn = 0) PC += $rel^{\emptyset}_{21:2}$:0_2	
RET	{Xn}	PC = Xn	
TBNZ	rn, #i, rel_{16}	if(rn_i ≠ 0) PC += $rel^{\pm}_{15:2}$:0_2	
TBZ	rn, #i, rel_{16}	if(rn_i = 0) PC += $rel^{\pm}_{15:2}$:0_2	

Atomic Instructions			
CAS{A}{L}	rs, rt, [Xn]	if (rs = $[Xn]_N$) $[Xn]_N$ = rt	1
CAS{A}{L}{B,H}	Ws, Wt, [Xn]	if (Ws_{N0} = $[Xn]_N$) $[Xn]_N$ = Wt_{N0}	1
CAS{A}{L}P	rs,rs2,rt,rt2,[Xn]	if (rs2:rs = $[Xn]_{2N}$) $[Xn]_{2N}$ = rt2:rt	1
LDao{A}{L}{B,H}	Ws, Wt, [Xn]	Wt=$[Xn]^{\emptyset}_N$; $[Xn]_N$=ao($[Xn]_N$,Ws_{N0})	1
LDao{A}{L}	rs, rt, [Xn]	rt = $[Xn]_N$; $[Xn]_N$ = ao($[Xn]_N$, rs)	1
STao{A}{L}{B,H}	Ws, [Xn]	$[Xn]_N$ = ao($[Xn]_N$, Ws_{N0})	1
STao{A}{L}	rs, [Xn]	$[Xn]_N$ = ao($[Xn]_N$, rs)	1
SWP{A}{L}{B,H}	Ws, Wt, [Xn]	Wt = $[Xn]^{\emptyset}_N$; $[Xn]_N$ = Ws_{N0}	1
SWP{A}{L}	rs, rt, [Xn]	rt = $[Xn]_N$; $[Xn]_N$ = rs	1

Conditional Instructions		
CCMN	rn, #i_5, #f_4, cc	if(cc) rn + i; else N:Z:C:V = f
CCMN	rn, rm, #f_4, cc	if(cc) rn + rm; else N:Z:C:V = f
CCMP	rn, #i_5, #f_4, cc	if(cc) rn − i; else N:Z:C:V = f
CCMP	rn, rm, #f_4, cc	if(cc) rn − rm; else N:Z:C:V = f
CINC	rd, rn, cc	if(cc) rd = rn + 1; else rd = rn
CINV	rd, rn, cc	if(cc) rd = ∼rn; else rd = rn
CNEG	rd, rn, cc	if(cc) rd = −rn; else rd = rn
CSEL	rd, rn, rm, cc	if(cc) rd = rn; else rd = rm
CSET	rd, cc	if(cc) rd = 1; else rd = 0
CSETM	rd, cc	if(cc) rd = ∼0; else rd = 0
CSINC	rd, rn, rm, cc	if(cc) rd = rn; else rd = rm + 1
CSINV	rd, rn, rm, cc	if(cc) rd = rn; else rd = ∼rm
CSNEG	rd, rn, rm, cc	if(cc) rd = rn; else rd = −rm

Load and Store Instructions		
LDP	rt, rt2, [addr]	rt2:rt = $[addr]_{2N}$
LDPSW	Xt, Xt2, [addr]	Xt = $[addr]^{\pm}_{32}$; Xt2 = $[addr+4]^{\pm}_{32}$
LD{U}R	rt, [addr]	rt = $[addr]_N$
LD{U}R{B,H}	Wt, [addr]	Wt = $[addr]^{\emptyset}_N$
LD{U}RS{B,H}	rt, [addr]	rt = $[addr]^{\pm}_N$
LD{U}RSW	Xt, [addr]	Xt = $[addr]^{\pm}_{32}$
PRFM	prfop, addr	Prefetch(addr, prfop)
STP	rt, rt2, [addr]	$[addr]_{2N}$ = rt2:rt
ST{U}R	rt, [addr]	$[addr]_N$ = rt
ST{U}R{B,H}	Wt, [addr]	$[addr]_N$ = Wt_{N0}

Addressing Modes (addr)		
xxP,LDPSW	[Xn{, #i_{7*s}}]	addr = Xn + $i^{\pm}_{6+s:s}$:0_s
xxP,LDPSW	[Xn], #i_{7*s}	addr=Xn; Xn+=$i^{\pm}_{6+s:s}$:0_s
xxP,LDPSW	[Xn, #i_{7*s}]!	Xn+=$i^{\pm}_{6+s:s}$:0_s; addr=Xn
xxR*,PRFM	[Xn{, #i_{12+s}}]	addr = Xn + $i^{\emptyset}_{11+s:s}$:0_s
xxR*	[Xn], #i_9	addr = Xn; Xn += $i^{\pm}$
xxR*	[Xn, #i_9]!	Xn += $i^{\pm}$; addr = Xn
xxR*,PRFM	[Xn,Xm{, LSL #0|s}]	addr = Xn + Xm ≪ s
xxR*,PRFM	[Xn,Wm,{S,U}XTW{ #0|s}]	addr = Xn + $Wm^{?}$ ≪ s
xxR*,PRFM	[Xn,Xm,SXTX{ #0|s}]	addr = Xn + $Xm^{\pm}$ ≪ s
xxUR*,PRFM	[Xn{, #i_9}]	addr = Xn += $i^{\pm}$
LDR{SW},PRFM	±rel_{21}	addr = PC + $rel^{\pm}_{20:2}$:0_2

Atomic Operations (ao)			
ADD	[Xn] + rs	SMAX	[Xn] $\tilde{>}$ rs ? [Xn] : rs
CLR	[Xn] & ∼rs	SMIN	[Xn] $\tilde{<}$ rs ? [Xn] : rs
EOR	[Xn] ⊕ rs	UMAX	[Xn] > rs ? [Xn] : rs
SET	[Xn] | rs	UMIN	[Xn] < rs ? [Xn] : rs

Operand 2 (op2)		
all	*rm*	rm
all	rm, LSL #i_6	rm $\ll$ i
all	rm, LSR #i_6	rm $\gg$ i
all	rm, ASR #i_6	rm $\overset{>}{\gg}$ i
logical	rm, ROR #i_6	rm $\gg\!\!\gg$ i
arithmetic	Wm, {S,U}XTB{ #i_3}	$Wm^{?}_{B0} \ll i$
arithmetic	Wm, {S,U}XTH{ #i_3}	$Wm^{?}_{H0} \ll i$
arithmetic	Wm, {S,U}XTW{ #i_3}	$Wm^{?} \ll i$
arithmetic	Xm, {S,U}XTX{ #i_3}	$Xm^{?} \ll i$
arithmetic	#i_{12}	$i^{\emptyset}$
arithmetic	#i_{24}	$i^{\emptyset}_{23..12}{:}0_{12}$
AND,EOR,ORR,TST	#mask	mask

Registers	
X0-X7	Arguments and return values
X8	Indirect result
X9-X15	Temporary
X16-X17	Intra-procedure-call temporary
X18	Platform defined use
X19-X28	Temporary (must be preserved)
X29	Frame pointer (must be preserved)
X30	Return address
SP	Stack pointer
XZR	Zero
PC	Program counter

Special Purpose Registers		
SPSR_EL{1..3}	Process state on exception entry to EL{1..3}	64
ELR_EL{1..3}	Exception return address from EL{1..3}	
SP_EL{0..2}	Stack pointer for EL{0..2}	64
SPSel	SP selection (0: SP=SP_EL0, 1: SP=SP_ELn)	
CurrentEL	Current Exception level (at bits 3..2)	RO
DAIF	Current interrupt mask bits (at bits 9..6)	
NZCV	Condition flags (at bits 31..28)	
FPCR	Floating-point operation control	
FPSR	Floating-point status	

Keys	
N	Operand bit size (8, 16, 32 or 64)
s	Operand log byte size (0=byte,1=hword,2=word,3=dword)
rd, rn, rm, rt	General register of either size (Wn or Xn)
prfop	P{LD,LI,ST}L{1..3}{KEEP,STRM}
{,sh}	Optional halfword left shift (LSL #{16,32,48})
$val^{\pm}$, $val^{\emptyset}$, $val^{?}$	Value is sign/zero extended (? depends on instruction)
$\overset{>}{\times}$ $\overset{>}{+}$ $\overset{>}{\gg}$ $\overset{>}{-}$ $\overset{>}{\ll}$	Operation is signed

Checksum Instructions		
CRC32{B,H}	Wd, Wn, Wm	Wd=CRC32(Wn,0x04c11db7,Wm_{N0})
CRC32W	Wd, Wn, Wm	Wd = CRC32(Wn,0x04c11db7,Wm)
CRC32X	Wd, Wn, Xm	Wd = CRC32(Wn,0x04c11db7,Xm)
CRC32C{B,H}	Wd, Wn, Wm	Wd=CRC32(Wn,0x1edc6f41,Wm_{N0})
CRC32CW	Wd, Wn, Wm	Wd = CRC32(Wn,0x1edc6f41,Wm)
CRC32CX	Wd, Wn, Xm	Wd = CRC32(Wn,0x1edc6f41,Xm)

Load and Store Instructions with Attribute		
LD{A}XP	rt, rt2, [Xn]	rt:rt2 = [Xn, <SetExclMonitor>]$_{2N}$
LD{A}{X}R	rt, [Xn]	rt = [Xn, <SetExclMonitor>]$_{N}$
LD{A}{X}R{B,H}	Wt, [Xn]	Wt = [Xn, <SetExclMonitor>]$^{\emptyset}_{N}$
LDNP	rt,rt2,[Xn{,#i_{7*s}}]	rt2:rt = [Xn + $i^{\pm}_{6+s}{:}0_s$, <Temp>]$_{2N}$
LDTR	rt, [Xn{, #i_9}]	rt = [Xn += $i^{\pm}$, <Unpriv>]$_{N}$
LDTR{B,H}	Wt, [Xn{, #i_9}]	Wt = [Xn += $i^{\pm}$, <Unpriv>]$^{\emptyset}_{N}$
LDTRS{B,H}	rt, [Xn{, #i_9}]	rt = [Xn += $i^{\pm}$, <Unpriv>]$^{\pm}_{N}$
LDTRSW	Xt, [Xn{, #i_9}]	Xt = [Xn += $i^{\pm}$, <Unpriv>]$^{\pm}_{32}$
STLR	rt, [Xn]	[Xn, <Release>]$_{N}$ = rt
STLR{B,H}	Wt, [Xn]	[Xn, <Release>]$_{N}$ = Wt_{N0}
ST{L}XP	Wd, rt, rt2, [Xn]	[Xn, <Excl>]$_{2N}$=rt:rt2; Wd=fail?1:0
ST{L}XR	Wd, rt, [Xn]	[Xn, <Excl>]$_{N}$=rt; Wd=fail?1:0
ST{L}XR{B,H}	Wd, Wt, [Xn]	[Xn, <Excl>]$_{N}$=Wt_{N0}; Wd=fail?1:0
STNP	rt,rt2,[Xn{,#i_{7*s}}]	[Xn + $i^{\pm}_{6+s}{:}0_s$, <Temp>]$_{2N}$ = rt2:rt
STTR	rt, [Xn{, #i_9}]	[Xn += $i^{\pm}$, <Unpriv>]$_{N}$ = rt
STTR{B,H}	Wt, [Xn{, #i_9}]	[Xn += $i^{\pm}$, <Unpriv>]$_{N}$ = Wt_{N0}

Condition Codes (cc)		
EQ	Equal	Z
NE	Not equal	!Z
CS/HS	Carry set, Unsigned higher or same	C
CC/LO	Carry clear, Unsigned lower	!C
MI	Minus, Negative	N
PL	Plus, Positive or zero	!N
VS	Overflow	V
VC	No overflow	!V
HI	Unsigned higher	C & !Z
LS	Unsigned lower or same	!C | Z
GE	Signed greater than or equal	N = V
LT	Signed less than	N $\neq$ V
GT	Signed greater than	!Z & N = V
LE	Signed less than or equal	Z | N $\neq$ V
AL	Always (default)	1

Notes for Instruction Set	
S	SP/WSP may be used as operand(s) instead of XZR/WZR
1	Introduced in ARMv8.1

System Instructions		
AT	S1{2}E{0..3}{R,W}, Xn	PAR_EL1 = AddrTrans(Xn)
BRK	#i_{16}	SoftwareBreakpoint(i)
CLREX	{#i_4}	ClearExclusiveLocal()
DMB	barrierop	DataMemoryBarrier(barrierop)
DSB	barrierop	DataSyncBarrier(barrierop)
ERET		PC=ELR_ELn;PSTATE=SPSR_ELn
HVC	#$_{16}$	CallHypervisor(i)
ISB	{SY}	InstructionSyncBarrier(SY)
MRS	Xd, sysreg	Xd = sysreg
MSR	sysreg, Xn	sysreg = Xn
MSR	SPSel, #i_1	PSTATE.SP = i
MSR	DAIFSet, #i_4	PSTATE.DAIF |= i
MSR	DAIFClr, #i_4	PSTATE.DAIF &= ~i
NOP		
SEV		SendEvent()
SEVL		EventRegisterSet()
SMC	#i_{16}	CallSecureMonitor(i)
SVC	#i_{16}	CallSupervisor(i)
WFE		WaitForEvent()
WFI		WaitForInterrupt()
YIELD		

Cache and TLB Maintenance Instructions		
DC	{C,CI,I}SW, Xx	DC clean and/or inv by Set/Way
DC	{C,CI,I}VAC, Xx	DC clean and/or inv by VA to PoC
DC	CVAU, Xx	DC clean by VA to PoU
DC	ZVA, Xx	DC zero by VA (len in DCZID_EL0)
IC	IALLU{IS}	IC inv all to PoU
IC	IVAU, Xx	IC inv VA to PoU
TLBI	ALLE{1..3}{IS}	TLB inv all
TLBI	ASIDE1{IS}, Xx	TLB inv by ASID
TLBI	IPAS2{L}E1{IS}, Xx	TLB inv by IPA {last level}
TLBI	VAA{L}E1{IS}, Xx	TLB inv by VA, all ASID {last level}
TLBI	VA{L}E{1..3}{IS}, Xx	TLB inv by VA {last level}
TLBI	VMALL{S12}E1{IS}	TLB inv by VMID, all, at stage 1{&2}

DMB and DSB Options	
OSH{,LD,ST}	Outer shareable, {all,load,store}
NSH{,LD,ST}	Non-shareable, {all,load,store}
ISH{,LD,ST}	Inner shareable, {all,load,store}
LD	Full system, load
ST	Full system, store
SY	Full system, all