U N I V E R S I D A D E DE S A˜ O P A U L O

**Escola de Artes, Ciˆencias e Humanidades**

Relato´rio Te´cnico PPgSI-000/2019

*Arquitetura ARMv8-A*

Bruno Lourenc¸o da Cunha Kevin Gabriel Gonc¸alves de Oliveira

Renan Ernesto Silva Pinto Vinicius Alves Matias

Novembro - 2019

O conteu´do do presente relato´rio e´ de u´nica responsabilidade dos autores.

Se´rie de Relato´rios Te´cnicos PPgSI-EACH-USP

Rua Arlindo Be´ttio, 1000 – Ermelino Matarazzo 03828-000 – Sa˜o Paulo, SP.

TEL: (11) 3091-8197 <http://www.each.usp.br/ppgsi>

**Arquitetura ARMv8-A**

## Bruno Lourenc¸o da Cunha1 , Kevin Gabriel Gonc¸alves de Oliveira2 , Renan Ernesto Silva Pinto3 , Vinicius Alves Matias4

1Escola de Artes, Cieˆncias e Humanidades – Universidade de Sa˜o Paulo Sa˜o Paulo – SP, Brazil

bruno [cunha@usp.br](mailto:cunha@usp.br)

2Escola de Artes, Cieˆncias e Humanidades – Universidade de Sa˜o Paulo Sa˜o Paulo – SP, Brazil

[autor2@email.com](mailto:autor2@email.com)

3Escola de Artes, Cieˆncias e Humanidades – Universidade de Sa˜o Paulo Sa˜o Paulo – SP, Brazil

[renan.ernesto@usp.br](mailto:renan.ernesto@usp.br)

4Escola de Artes, Cieˆncias e Humanidades – Universidade de Sa˜o Paulo Sa˜o Paulo – SP, Brazil

[viniciusmatias@usp.br](mailto:viniciusmatias@usp.br)

***Resumo.*** *Relato´rio te´cnico referente a` arquitetura ARMv8-A, desenvolvida pela ARM Holdings e presente em diversos chips fabricados para dispositivos mo´veis.*

# Histo´rico

A arquitetura ARMv8-A foi anunciada em outubro de 2011 pela empresa britaˆnica ARM (Advanced RISC Machine) Holdings como a oitava versa˜o da arquitetura ARM, tendo seu perfil definido para aplicac¸o˜es e portanto, sendo otimizada para sistemas ope- racionais de alto n´ıvel. A arquitetura pertencente a` linha de arquitetura RISC (Reduced Instruction Set Computer) contendo as seguintes caracter´ısticas:

Um grande arquivo uniforme de registro.

•

Uma arquitetura de *load/store*, onde as operac¸o˜es de processamento de dados operam somente no conteu´do do registrador e na˜o diretamente no conteu´do da memo´ria.

•

Modo simples de enderec¸amento, com todos os enderec¸os de *load/store* determi- nados somente do conteu´do do registrador e campos da intruc¸a˜o.

•

A arquitetura suporta tanto enderec¸amento quanto aritme´tica de 64 bits e instruc¸o˜es de tamanho fixo de 32 bits, ale´m de um estado de execuc¸a˜o de 64 bits (AArch64) e outro

de 32 bits (AArch32), que e´ completamente compat´ıvel com as verso˜es anteriores da

arquitetura ARM. Atualmente, a arquitetura vem sendo aprimorada e esta´ em constante evoluc¸a˜o. A ARMv8.6-A fornece um ambiente prop´ıcio para o desenvolvimento de Redes Neurais (NN) para Machine Learning (ML) atrave´s de General Matrix Multiply (GEMM) e BFloat 16. Ale´m de todas evoluc¸o˜es presentes nas verso˜es ARMv8.1-A, ARMv8.2-A, ARMv8.3-A, ARMv8.4-A e ARMv8.5-A.

# Uso atual

A arquitetura esta´ presente em diversos chips que visam uma boa eficieˆncia energe´tica aliada a um alto desempenho. Um exemplo de chip com tais caracter´ısticas e´ o Snap- dragon 855, presente em celulares como Asus Zenfone 5Z1, Xiaomi Mi 82 e va´rios ou- tros3 que usam as microarquiteturas Cortex-A76 e Cortex-A55. Com o encerramento da fabricac¸a˜o dos processadores Atom da Intel (maio de 2016), a arquitetura ARMv8-A tornou-se o padra˜o da indu´stria para todos os dispositivos mo´veis.

# Desempenho

Para medir o desempenho da arquitetura, o dispositivo usado como refereˆncia foi o OnePlus 7 Pro, equipado com o chip Snapdragon 855 (anunciado em 5 de Dezembro de 2018 pela Qualcomm Technologies, baseado na arquitetura ARMv8-A). Os testes fo- ram realizados a partir da versa˜o 5.0.3 do Geekbench para Android AArch64. Como refereˆncia, o Geekbench estabelece 1000 pontos como sendo o resultado da pontuac¸a˜o de um i3-8100. Os resultados foram os seguintes:

* + Single-Core Score: 763 Pontos
  + Single-Core Crypto Score: 1027 Pontos
  + Single-Core Integer Score: 734 Pontos
  + Single-Core Floating Point Score: 781 Pontos
  + Multi-Core Score: 2778 Pontos
  + Multi-Core Crypto Score: 3974 Pontos
  + Multi-Core Integer Score: 2707 Pontos
  + Multi-Core Floating Point Score: 2731 Pontos

# Instruction Set

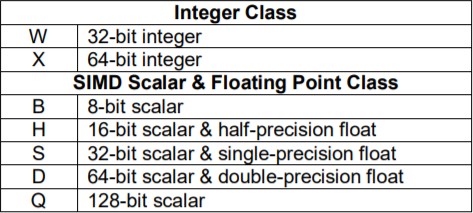
## Estrutura ba´sica

O assembler da arquitetura ARMv8-A reconhece instruc¸o˜es tanto em caixa alta como em caixa baixa. As instruc¸o˜es sa˜o linhas compostas por um ou mais ro´tulos(labels) segui- dos do nome da operac¸a˜o, um registrador de destino e um ou mais registradores, separados por v´ırgula, utilizados na operac¸a˜o. Sendo assim, a estrutura das instruc¸o˜es segue o se- guinte padra˜o:

label:\* opcode dest , source1 , source2 , source3

{ } { { { { { }}}}}

A ordem do registrador de destino e dos registradores fonte sa˜o trocadas apenas na instruc¸a˜o store.

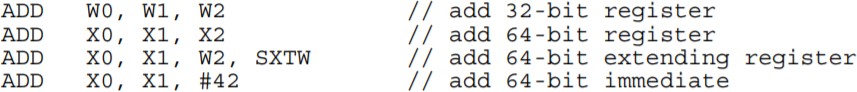
Na instruc¸a˜o assembly os registradores por sua vez podem assumir diferentes formatos. A tabela abaixo elenca isso:

1[https://www.asus.com/Phone/ZenFone-](http://www.asus.com/Phone/ZenFone-5Z-ZS620KL/Tech-Specs/)5Z-[ZS620KL/Tech-Specs/](http://www.asus.com/Phone/ZenFone-5Z-ZS620KL/Tech-Specs/) .

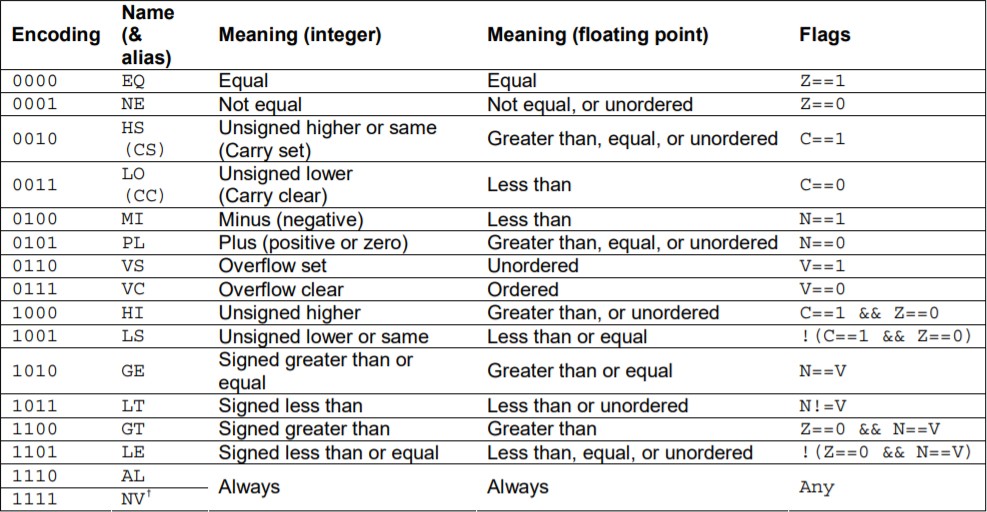
2[https://www.mi.com/global/mi8/specs](http://www.mi.com/global/mi8/specs) .

3[https://www.techwalls.com/qualcomm-snapdragon-](http://www.techwalls.com/qualcomm-snapdragon-855-smartphones/)855-[smartphones/](http://www.techwalls.com/qualcomm-snapdragon-855-smartphones/) .

Assim, as instruc¸o˜es podem tomar as seguintes formas:



## Condicionais



* 1. **Usados frequentemente**

As seguintes sintaxes sa˜o frequentemente usadas:

**Xn -** O operador Xn ou Wn interpreta o registrador 31 como um registrador zero, re- presentado pelos nomes XZR ou WZR respectivamente.

**Xn SP -** O operador Xn SP ou Wn WSP interpreta o registrador 31 como o ponteiro de pilha representado pelos nomes SP ou WSP respectivamente.

| | |

**cond -** Uma condic¸a˜o padra˜o ARM como EQ, NE, CS HS, CC LO, MI, PL, VS, VC, HI, LS, GE, LT, GT, LE, AL ou NV com os mesmos significados da arquitetura AArch32.

| |

**invert(cond) -** O inverso de cond, por exemplo, o inverso de GT e´ LE.

**uimmn -** Um n-bit valor imediato sem sinal.

**simmn -** Um n-bit valor imediato com sinal em forma de complemento de 2 (onde n inclui o bit de sinal).

**label -** Representa uma refereˆncia para uma parte do co´digo ou localizac¸a˜o.

**addr -** Representa um modo de enderec¸amento.

**lshift -** Representa um operador de deslocamento realizado ao final de operadores lo´gicos. Engloba instruc¸o˜es como LSL, LSR, ASR ou ROR, que sa˜o seguidas de uma quantidade constante de deslocamento.

**ashift -** Representa um operador de deslocamento realizado ao final de operadores ari- time´ticos. Engloba instruc¸o˜es como LSL, LSR, ou ASR, que sa˜o seguidas de uma quan- tidade constante e deslocamento.

## Controle de Fluxo

* + 1. **Branch condicional**

Operadores utilizados em branch’s condicionais:

**B.cond label -** Branch: Condicionalmente salta para label se cond e´ verdadeiro.

**CBNZ Wn, label -** Compare and Branch Not Zero: Condicionalmente salta para label se Wn na˜o e´ igual a zero.

**CBNZ Xn, label -** Compare and Branch Not Zero (extended): Condicionalmente salta para label se Xn na˜o e´ igual a zero.

**CBZ Wn, label -** Compare and Branch Zero: Condicionalmente salta para label se Wn e´ igual a zero.

**CBZ Xn, label -** Compare and Branch Zero (extended): Condicionalmente salta para label se Xn e´ igual a zero.

**TBNZ Xn Wn #uimm6 label -** Test and Branch Not Zero: condicionalmente salta para label se o nu´mero de bits uimn6 no registrador Xn na˜o e´ igual a zero.

|

**TBZ Xn Wn, #uimm6, label -** Test and Branch Zero: condicionalmente salta para la- bel se o nu´mero de bits uimn6 no registrador Xn e´ igual a zero.

|

## Branch na˜o condicional imediato

Operadores utilizados em branch’s na˜o condicionais imediatos:

**B label -** Branch: incondicionalmente salta para label.

**BL label -** Branch and Link: incondicionalmente salta para label e escreve o enderec¸o da pro´xima instruc¸a˜o sequencial no registrador X30.

## Branch na˜o condicional com registrador

Operadores utilizados em branch’s na˜o condicionais utilizando o enderec¸o de me´moria em registradores:

**BLR Xm -** Branch and Link Register: incondicionalmente salta para o enderec¸o em Xm e escreve o enderec¸o da pro´xima instruc¸a˜o sequencial no registrador X30.

**BR Xm -** Branch Register: salta para o enderec¸o em Xm com um lembrete a CPU que isso na˜o e´ um retorno de subrotina.

**RET Xm -** Return: salta para o enderec¸o em Xm com um lembrete a CPU que isso e´ um retorno de subrotina.

## Memory Access

* + 1. **Load-Store com registrador u´nico**

A forma mais geral de load-store, ela suporta uma variedade de modos de enderec¸amento ale´m dos registradores base Xn ou SP.

**LDR Wt, addr -** Load Register: carrega em Wt uma palavra da memo´ria enderec¸ada por addr.

**LDR Xt, addr -** Load Register (extended): carrega em Xt uma dupla-palavra (dou- bleword) da memo´ria enderec¸ada por addr.

**LDRB Wt, addr -** Load Byte: carrega em Wt um byte da memo´ria enderec¸ada por addr e preenche os bits restantes com zero.

**LDRSB Wt, addr -** Load Signed Byte: carrega em Wt um byte da memo´ria enderec¸ada por addr e preenche os bits restantes de acordo com o sinal.

**LDRSB Xt, addr -** Load Signed Byte (extended): carrega em Xt um byte da memo´ria enderec¸ada por addr e preenche os bits restantes de acordo com o sinal do byte.

**LDRH Wt, addr -** Load Halfword: carrega em Wt metade de uma palavra (halfword) da memo´ria enderec¸ada por addr e preenche os bits restantes com zero.

**LDRSH Wt, addr -** Load Signed Halfword: carrega em Wt metade de uma palavra (halfword) da memo´ria enderec¸ada por addr e preenche os bits restantes de acordo com o sinal.

**LDRSH Xt, addr -** Load Signed Halfword (extended): carrega em Xt metade de uma palavra (halfword) da memo´ria enderec¸ada por addr e preenche os bits restantes de acordo com o sinal.

**LDRSW Xt, addr -** Load Signed Word (extended): carrega em Xt uma palavra da memo´ria enderec¸ada por addr e preenche os bits restantes de acordo com o sinal.

**STR Wt, addr -** Store Register: armazena uma palavra de Wt no enderec¸o de memo´ria addr.

**STR Xt, addr -** Store Register (extended): armazena uma palavra dupla (doubleword) de Xt no enderec¸o de memo´ria addr.

**STRB Wt, addr -** Store Byte: armazena um byte contido em Wt no enderec¸o de memo´ria addr.

**STRH Wt, addr -** Store Halfword: armazena metade de uma palavra (halfword) con- tida em Wt no enderec¸o de memo´ria addr.

## Data Processing (immediate)

Os seguintes grupos de instruc¸a˜o sa˜o suportados:

* Aritme´ticas (immediate)
* Lo´gicas (immediate)
* Move (immediate)
* Bitfield (operations)
* Deslocamento (immediate)
* Extensa˜o de sinal/zero

## Arithmetic (immediate)

Operac¸o˜es aritme´ticas que aceitam valores imediatos:

**ADD Wd**|**WSP, Wn**|**WSP, #aimm -** Add (immediate): Wd|WSP = Wn|WSP + aimm.

**ADD Xd**|**SP, Xn**|**SP, #aimm -** Add (extended immediate): Xd|SP = Xn|SP + aimm.

**ADDS Wd, Wn WSP, #aimm -** Add and set flags (immediate): Wd = Wn WSP + aimm, configurando as flags de condic¸a˜o.

| |

**ADDS Xd, Xn SP, #aimm -** Add and set flags (extended immediate): Xd = Xn SP + aimm, configurando as flags de condic¸a˜o.

| |

**SUB Wd**|**WSP, Wn**|**WSP, #aimm -** Subtract (immediate): Wd|WSP = Wn|WSP - aimm.

**SUB Xd**|**SP, Xn**|**SP, #aimm -** Subtract (extended immediate): Xd|SP = Xn|SP - aimm.

**SUBS Wd, Wn WSP, #aimm -** Subtract and set flags (immediate): Wd = Wn WSP - aimm, configurando as flags de condic¸a˜o.

| |

**SUBS Xd, Xn SP, #aimm -** Subtract and set flags (extended immediate): Xd = Xn SP

| |

- aimm, configurando as flags de condic¸a˜o.

**CMP Wn**|**WSP, #aimm -** Compare (immediate): pseudoˆmino para SUBS WZR,Wn|WSP,#aimm.

**CMP Xn, #aimm -** Compare (extended immediate): pseudoˆmino para SUBS XZR,Xn|SP,#aimm.

**CMN Wn**|**WSP, #aimm -** Compare negative (immediate): pseudoˆmino para ADDS WZR,Wn|WSP,#aimm.

**CMN Xn**|**SP, #aimm -** Compare negative (extended immediate): pseudoˆmino para ADDS XZR,Xn|SP,#aimm.

**MOV Wd WSP, Wn WSP -** Move (register): pseudoˆmino para ADD Wd WSP,Wn WSP,#0, mas somente quando algum dos registradores e´ o WSP. Em outros casos a instruc¸a˜o ORR Wd,WZR,Wn e´ usada.

| | | |

**MOV Xd SP, Xn SP -** Move (extended register): pseudoˆmino para ADD Xd SP,Xn SP,#0, mas somente quando algum dos registradores e´ o SP. Em outros casos a instruc¸a˜o ORR Xd,XZR,Xn e´ usada.

| | | |

## Logical (immediate)

Operac¸o˜es lo´gicas que aceitam valores imediatos:

**AND Wd WSP, Wn, #bimm32 -** Bitwise AND (immediate): Wd WSP = Wn AND bimm32.

| |

**AND Xd SP, Xn, #bimm64 -** Bitwise AND (extended immediate): Xd SP = Xn AND bimm64.

| |

**ANDS Wd, Wn, #bimm32 -** Bitwise AND and Set Flags (immediate): Wd = Wn AND bimm32, atribuindo as flags de condic¸a˜o N e Z baseado no resultado, ale´m de limpar as flags C e V.

**ANDS Xd, Xn, #bimm64 -** Bitwise AND and Set Flags (extended immediate): Xd = Xn AND bimm64, atribuindo as flags de condic¸a˜o N e Z baseado no resultado, ale´m de limpar as flags C e V.

**EOR Wd WSP, Wn, #bimm32 -** Bitwise exclusive OR (immediate): Wd WSP = Wn EOR bimm32.

| |

**EOR Xd SP, Xn, #bimm64 -** Bitwise exclusive OR (extended immediate): Xd SP = Xn EOR bimm64.

| |

**ORR Wd WSP, Wn, #bimm32 -** Bitwise inclusive OR (immediate): Wd WSP = Wn OR bimm32.

| |

**ORR Xd SP, Xn, #bimm64 -** Bitwise inclusive OR (extended immediate): Xd SP = Xn OR bimm64.

| |

**MOVI Wd, #bimm32 -** Move bitmask (immediate): pseudoˆmino para ORR Wd,WZR,#bimm32.

**MOVI Xd, #bimm64 -** Move bitmask (extended immediate): pseudoˆmino para ORR

Xd,XZR,#bimm64.

**TST Wn, #bimm32 -** Bitwise test (immediate): pseudoˆmino para ANDS WZR,Wn,#bimm32.

**TST Xn, #bimm64 -** Bitwise test (extended immediate): pseudoˆmino para ANDS XZR,Xn,#bimm64

## Move (wide immediate)

As seguintes instruc¸o˜es inserem um valor imediato em um registrador destino

**MOVZ Wt, #uimm16, LSL #pos -** Move with Zero (immediate): Wt = LSL(uimm16, pos).

**MOVZ Xt, #uimm16, LSL #pos -** Move with Zero (extended immediate): Xt = LSL(uimm16, pos).

**MOVN Wt, #uimm16, LSL #pos -** Move with NOT (immediate): Wt = NOT(LSL(uimm16, pos)).

**MOVN Xt, #uimm16, LSL #pos -** Move with NOT (extended immediate): Xt = NOT(LSL(uimm16, pos)).

**MOVK Wt, #uimm16, LSL #pos -** Move with Keep (immediate): Wt¡pos+15:pos¿ = uimm16.

**MOVK Xt, #uimm16, LSL #pos -** Move with Keep (extended immediate): Xt¡pos+15:pos¿

= uimm16.

**MOV Wd, #simm32 -** Uma instruc¸a˜o sinte´tica que gera tanto MOVZ, MOVN, como MOVI.

**MOV Xd, #simm64 -** Funciona assim como o MOV pore´m para carregar registradores Xd com valores de 64-bit.

## Bitfield Operations

Operac¸o˜es de bits:

**BFM Wd, Wn, #r, #s -** Bitfield Move: se s¿=r enta˜o Wd¡s-r:0¿ = Wn¡s:r¿, caso contra´rio Wd¡32+s-r,32-r¿ = Wn¡s:0¿. Deixando os outros bits em Wd inalterados.

**BFM Xd, Xn, #r, #s -** Bitfield Move: se s¿=r enta˜o Xd¡s-r:0¿ = Xn¡s:r¿, caso contra´rio Xd¡64+s-r,64-r¿ = Xn¡s:0¿. Deixando os outros bits em Xd inalterados.

**SBFM Wd, Wn, #r, #s -** Signed Bitfield Move: se s¿=r enta˜o Wd¡s-r:0¿ = Wn¡s:r¿, caso contra´rio Wd¡32+s-r,32-r¿ = Wn¡s:0¿. atribuindo os bits a` esquerda com o valor do bit mais a esquerda e os bits da direita do bitfield de destino com zero.

**SBFM Xd, Xn, #r, #s -** Signed Bitfield Move: se s¿=r enta˜o Xd¡s-r:0¿ = Xn¡s:r¿, caso contra´rio Xd¡64+s-r,64-r¿ = Xn¡s:0¿. atribuindo os bits a` esquerda com o valor do bit mais a esquerda e os bits da direita do bitfield de destino com zero.

**UBFM Wd, Wn, #r, #s -** Unsigned Bitfield Move: se s¿=r enta˜o Wd¡s-r:0¿ = Wn¡s:r¿, caso contra´rio Wd¡32+s-r,32-r¿ = Wn¡s:0¿. Atribuindo zero aos bits a esquerda e a direita do bitfield de destino.

**UBFM Xd, Xn, #r, #s -** Unsigned Bitfield Move: se s¿=r enta˜o Xd¡s-r:0¿ = Xn¡s:r¿, caso contra´rio Xd¡32+s-r,32-r¿ = Xn¡s:0¿. Atribuindo zero aos bits a esquerda e a direita do bitfield de destino.

As seguintes instruc¸o˜es sa˜o pseudoˆnimos mais amiga´veis para operac¸o˜es de bit:

**BFI Wd, Wn, #lsb, #width -** Bitfield Insert: pseudoˆnimo para BFM Wd,Wn,#((32- lsb)&31),#(width-1).

**BFI Xd, Xn, #lsb, #width -** Bitfield Insert (extended): pseudoˆnimo para BFM Xd,Xn,#((64- lsb)&63),#(width-1).

**BFXIL Wd, Wn, #lsb, #width -** Bitfield Extract and Insert Low: pseudoˆnimo para BFM Wd,Wn,#lsb,#(lsb+width-1).

**BFXIL Xd, Xn, #lsb, #width -** Bitfield Extract and Insert Low (extended): pseudoˆnimo para BFM Xd,Xn,#lsb,#(lsb+width-1).

**SBFIZ Wd, Wn, #lsb, #width -** Signed Bitfield Insert in Zero: pseudoˆnimo para SBFM Wd,Wn,#((32-lsb)&31),#(width-1).

**SBFIZ Xd, Xn, #lsb, #width -** Signed Bitfield Insert in Zero (extended): pseudoˆnimo para SBFM Xd,Xn,#((64-lsb)&63),#(width-1).

**SBFX Wd, Wn, #lsb, #width -** Signed Bitfield Extract: pseudoˆnimo para SBFM Wd,Wn,#lsb,#(lsb+width- 1).

**SBFX Xd, Xn, #lsb, #width -** Signed Bitfield Extract (extended): pseudoˆnimo para SBFM Xd,Xn,#lsb,#(lsb+width-1).

**UBFIZ Wd, Wn, #lsb, #width -** Unsigned Bitfield Insert in Zero: pseudoˆnimo para UBFM Wd,Wn,#((32-lsb)&31),#(width-1).

**UBFIZ Xd, Xn, #lsb, #width -** Unsigned Bitfield Insert in Zero (extended): pseudoˆnimo para UBFM Xd,Xn,#((64-lsb)&63),#(width-1).

**UBFX Wd, Wn, #lsb, #width -** Unsigned Bitfield Extract: pseudoˆnimo para UBFM

Wd,Wn,#lsb,#(lsb+width-1).

**UBFX Xd, Xn, #lsb, #width -** Unsigned Bitfield Extract (extended): pseudoˆnimo para UBFM Xd,Xn,#lsb,#(lsb+width-1).

## Extract (immediate)

Operac¸o˜es de extrac¸a˜o de bit de valores imediatos:

**EXTR Wd, Wn, Wm, #lsb -** Extract: Wd = Wn:Wm¡lsb+31,lsb¿. O bit na posic¸a˜o lsb deve estar entre 0 e 31.

**EXTR Xd, Xn, Xm, #lsb -** Extract (extended): Xd = Xn:Xm¡lsb+63,lsb¿. O bit na posic¸a˜o lsb deve estar entre 0 e 63.

## Shift (immediate)

a

**ASR Wd, Wn, uimm -** Arithmetic Shift Right (immediate): alias for SBFM Wd,Wn,uimm,31.

**ASR Xd, Xn, uimm -** Arithmetic Shift Right (extended immediate): alias for SBFM Xd,Xn,uimm,63.

**LSL Wd, Wn, uimm -** Logical Shift Left (immediate): alias for UBFM Wd,Wn,((32- uimm)&31),(31-uimm).

**LSL Xd, Xn, uimm -** Logical Shift Left (extended immediate): alias for UBFM Xd,Xn,((64- uimm)&63),(63-uimm)

**LSR Wd, Wn, uimm -** Logical Shift Left (immediate): alias for UBFM Wd,Wn,uimm,31.

**LSR Xd, Xn, uimm -** Logical Shift Left (extended immediate): alias for UBFM Xd,Xn,uimm,31.

**ROR Wd, Wm, uimm -** Rotate Right (immediate): alias for EXTR Wd,Wm,Wm,uimm.

**ROR Xd, Xm, uimm -** Rotate Right (extended immediate): alias for EXTR Xd,Xm,Xm,uimm.

## Sign/Zero Extend

**SXT[BH] Wd, Wn -** Signed Extend Byte—Halfword: alias for SBFM Wd,Wn,0,7—15.

**SXT[BHW] Xd, Wn -** Signed Extend Byte—Halfword—Word (extended): alias for SBFM Xd,Xn,0,7—15—31.

**UXT[BH] Wd, Wn -** Unsigned Extend Byte—Halfword: alias for UBFM Wd,Wn,0,7—15.

**UXT[BHW] Xd, Wn -** Unsigned Extend Byte—Halfword—Word (extended): alias for UBFM Xd,Xn,0,7—15—31.

## Data Processing (register)

a

## Arithmetic (shifted register)

**ADD Wd, Wn, Wm, ashift imm -** Add (register): Wd = Wn + ashift(Wm, imm).

**ADD Xd, Xn, Xm, ashift imm -** Add (extended register): Xd = Xn + ashift(Xm, imm).

**ADDS Wd, Wn, Wm, ashift imm -** Add and Set Flags (register): Wd = Wn + ashift(Wm, imm), setting condition flags.

**ADDS Xd, Xn, Xm, ashift imm -** Add and Set Flags (extended register): Xd = Xn + ashift(Xm, imm), setting condition flags.

**SUB Wd, Wn, Wm, ashift imm -** Subtract (register): Wd = Wn - ashift(Wm, imm).

**SUB Xd, Xn, Xm, ashift imm -** Subtract (extended register): Xd = Xn - ashift(Xm, imm).

**SUBS Wd, Wn, Wm, ashift imm -** Subtract and Set Flags (register): Wd = Wn - ashift(Wm, imm), setting condition flags.

**SUBS Xd, Xn, Xm, ashift imm -** Subtract and Set Flags (extended register): Xd = Xn - ashift(Xm, imm), setting condition flags.

**CMN Wn, Wm, ashift imm -** Compare Negative (register): alias for ADDS WZR, Wn, Wm, ashift imm.

**CMN Xn, Xm, ashift imm -** Compare Negative (extended register): alias for ADDS XZR, Xn, Xm, ashift imm.

**CMP Wn, Wm, ashift imm -** Compare (register): alias for SUBS WZR, Wn, Wm,ashift imm.

**CMP Xn, Xm, ashift imm -** Compare (extended register): alias for SUBS XZR, Xn, Xm, ashift imm.

**NEG Wd, Wm, ashift imm -** Negate: alias for SUB Wd, WZR, Wm, ashift imm.

**NEG Xd, Xm, ashift imm -** Negate (extended): alias for SUB Xd, XZR, Xm, ashift imm.

**NEGS Wd, Wm, ashift imm -** Negate and Set Flags: alias for SUBS Wd, WZR,

Wm, ashift imm.

**NEGS Xd, Xm, ashift imm -** Negate and Set Flags (extended): alias for SUBS Xd, XZR, Xm, ashift imm.

## Arithmetic (extending register)

a

**ADD Wd—WSP, Wn—WSP, Wm, extend imm -** Add (register, extending): Wd—WSP

= Wn—WSP + LSL(extend(Wm),imm).

**ADD Xd—SP, Xn—SP, Wm, extend imm -** Add (extended register, extending): Xd—SP = Xn—SP + LSL(extend(Wm),imm).

**ADD Xd—SP, Xn—SP, Xm, UXTX—LSL imm -** Add (extended register, exten- ding): Xd—SP = Xn—SP + LSL(Xm,imm).

**ADDS Wd, Wn—WSP, Wm, extend imm -** Add and Set Flags (register, extending): Wd = Wn—WSP + LSL(extend(Wm),imm), setting the condition flags.

**ADDS Xd, Xn—SP, Wm, extend imm -** Add and Set Flags (extended register, exten- ding): Xd = Xn—SP + LSL(extend(Wm),imm), setting the condition flags.

**ADDS Xd, Xn—SP, Xm, UXTX—LSL imm -** Add and Set Flags (extended register, extending): Xd = Xn—SP + LSL(Xm,imm), setting the condition flags.

**SUB Wd—WSP, Wn—WSP, Wm, extend imm -** Subtract (register, extending): Wd—WSP

= Wn—WSP - LSL(extend(Wm),imm).

**SUB Xd—SP, Xn—SP, Wm, extend imm -** Subtract (extended register, extending): Xd—SP = Xn—SP - LSL(extend(Wm),imm).

**SUB Xd—SP, Xn—SP, Xm, UXTX—LSL imm -** Subtract (extended register, exten- ding): Xd—SP = Xn—SP - LSL(Xm,imm).

**SUBS Wd, Wn—WSP, Wm, extend imm -** Subtract and Set Flags (register, exten- ding): Wd = Wn—WSP - LSL(extend(Wm),imm), setting the condition flags.

**SUBS Xd, Xn—SP, Wm, extend imm -** Subtract and Set Flags (extended register, ex- tending): Xd = Xn—SP - LSL(extend(Wm),imm), setting the condition flags.

**SUBS Xd, Xn—SP, Xm, UXTX—LSL imm -** Subtract and Set Flags (extended register, extending): Xd = Xn—SP - LSL(Xm,imm), setting the condition flags.

**CMN Wn—WSP, Wm, extend imm -** Compare Negative (register, extending): alias for ADDS WZR,Wn,Wm,extend imm.

**CMN Xn—SP, Wm, extend imm -** Compare Negative (extended register, extending): alias for ADDS XZR,Xn,Wm,extend imm.

**CMN Xn—SP, Xm, UXTX—LSL imm -** Compare Negative (extended register, ex-

tending): alias for ADDS XZR,Xn,Xm,UXTX—LSL imm.

**CMP Wn—WSP, Wm, extend imm -** Compare (register, extending): alias for SUBS WZR,Wn,Wm,extend imm.

**CMP Xn—SP, Wm, extend imm -** Compare (extended register, extending): alias for SUBS XZR,Xn,Wm,extend imm.

**CMP Xn—SP, Xm, UXTX—LSL imm -** Compare (extended register, extending): alias for SUBS XZR,Xn,Xm,UXTX—LSL imm.

## Logical (shifted register)

a

**AND Wd, Wn, Wm, lshift imm -** Bitwise AND (register): Wd = Wn AND lshift(Wm, imm).

**AND Xd, Xn, Xm, lshift imm -** Bitwise AND (extended register): Xd = Xn AND lshift(Xm, imm).

**ANDS Wd, Wn, Wm, lshift imm -** Bitwise AND and Set Flags (register): Wd = Wn AND lshift(Wm, imm), setting N & Z condition flags based on the result and clearing the C & V flags.

**ANDS Xd, Xn, Xm, lshift imm -** Bitwise AND and Set Flags (extended register): Xd = Xn AND lshift(Xm, imm), setting N Z condition flags based on the result and clearing the C V flags.

**BIC Wd, Wn, Wm, lshift imm -** Bit Clear (register): Wd = Wn AND NOT(lshift(Wm, imm)).

**BIC Xd, Xn, Xm, lshift imm -** Bit Clear (extended register): Xd = Xn AND NOT(lshift(Xm, imm)).

**BICS Wd, Wn, Wm, lshift imm -** Bit Clear and Set Flags (register): Wd = Wn AND NOT(lshift(Wm, imm)), setting N Z condition flags based on the result and clearing the C V flags.

**BICS Xd, Xn, Xm, lshift imm -** Bit Clear and Set Flags (extended register): Xd = Xn AND NOT(lshift(Xm, imm)), setting N Z condition flags based on the result and clearing the C V flags.

**EON Wd, Wn, Wm, lshift imm -** Bitwise exclusive OR NOT (register): Wd = Wn EOR NOT(lshift(Wm, imm)).

**EON Xd, Xn, Xm, lshift imm -** Bitwise exclusive OR NOT (extended register): Xd

= Xn EOR NOT(lshift(Xm, imm)).

**EOR Wd, Wn, Wm, lshift imm -** Bitwise exclusive OR (register): Wd = Wn EOR lshift(Wm, imm).

**EOR Xd, Xn, Xm, lshift imm -** Bitwise exclusive OR (extended register): Xd = Xn EOR lshift(Xm, imm).

**ORR Wd, Wn, Wm, lshift imm -** Bitwise inclusive OR (register): Wd = Wn OR lshift(Wm, imm).

**ORR Xd, Xn, Xm, lshift imm -** Bitwise inclusive OR (extended register): Xd = Xn OR lshift(Xm, imm).

**ORN Wd, Wn, Wm, lshift imm -** Bitwise inclusive OR NOT (register): Wd = Wn OR NOT(lshift(Wm, imm)).

**ORN Xd, Xn, Xm, lshift imm -** Bitwise inclusive OR NOT (extended register): Xd

= Xn OR NOT(lshift(Xm, imm)).

**MOV Wd, Wm -** Move (register): alias for ORR Wd,WZR,Wm.

**MOV Xd, Xm -** Move (extended register): alias for ORR Xd,XZR,Xm.

**MVN Wd, Wm, lshift imm -** Move NOT (register): alias for ORN Wd,WZR,Wm,lshift imm.

**MVN Xd, Xm, lshift imm -** Move NOT (extended register): alias for ORN Xd,XZR,Xm,lshift imm.

**TST Wn, Wm, lshift imm -** Bitwise Test (register): alias for ANDS WZR,Wn,Wm,lshift imm.

**TST Xn, Xm, lshift imm -** Bitwise Test (extended register): alias for ANDS XZR,Xn,Xm,lshift imm.

## Variable Shift

a

## -

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

**-**

Cortex-A57 Software Optimization Guide

Date of Issue: January 28, 2016

Copyright ARM Limited 2016. All rights reserved.

## Cortex®-A57

## Software Optimization Guide

Copyright © **2016** ARM. All rights reserved.

## Release Information

The following changes have been made to this Software Optimization Guide.

## Change History

## Date Issue Confidentiality Change

19 January 2015 A Non-confidential First release

28 January 2016 B Non-confidential Second release Added 4.7 Non-Temporal

Loads/Stores

## Proprietary Notice

Words and logos marked with ™ or ® are registered trademarks or trademarks of ARM® in the EU and other countries except as otherwise stated below in this proprietary notice. Other brands and names mentioned herein may be the trademarks of their respective owners.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. ARM shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

Where the term ARM is used it means “ARM or any of its subsidiaries as appropriate”.

## Confidentiality Status

This document is Non-Confidential. This document has no restriction on distribution.

## Product Status

The information in this document is final, that is for a developed product.

## Web Address

[http://www.arm.com](http://www.arm.com/)

**Contents**

1. [ABOUT THIS DOCUMENT 5](#_TOC_250043)
   1. [References 5](#_TOC_250042)
   2. [Terms and abbreviations 5](#_TOC_250041)
   3. [Document Scope 5](#_TOC_250040)
2. [INTRODUCTION 6](#_TOC_250039)
   1. [Pipeline Overview 6](#_TOC_250038)
3. [INSTRUCTION CHARACTERISTICS 7](#_TOC_250037)
   1. [Instruction Tables 7](#_TOC_250036)
   2. [Branch Instructions 7](#_TOC_250035)
   3. [Arithmetic and Logical Instructions 7](#_TOC_250034)
   4. [Move and Shift Instructions 8](#_TOC_250033)
   5. [Divide and Multiply Instructions 9](#_TOC_250032)
   6. [Saturating and Parallel Arithmetic Instructions 10](#_TOC_250031)
   7. [Miscellaneous Data-Processing Instructions 11](#_TOC_250030)
   8. [Load Instructions 12](#_TOC_250029)
   9. [Store Instructions 15](#_TOC_250028)
   10. [FP Data Processing Instructions 16](#_TOC_250027)
   11. [FP Miscellaneous Instructions 18](#_TOC_250026)
   12. [FP Load Instructions 19](#_TOC_250025)
   13. [FP Store Instructions 20](#_TOC_250024)
   14. [ASIMD Integer Instructions 22](#_TOC_250023)
   15. [ASIMD Floating-Point Instructions 26](#_TOC_250022)
   16. [ASIMD Miscellaneous Instructions 29](#_TOC_250021)
   17. [ASIMD Load Instructions 31](#_TOC_250020)
   18. [ASIMD Store Instructions 33](#_TOC_250019)
   19. [Cryptography Extensions 35](#_TOC_250018)
   20. [CRC 35](#_TOC_250017)
4. [SPECIAL CONSIDERATIONS 37](#_TOC_250016)
   1. [Dispatch Constraints 37](#_TOC_250015)
   2. [Conditional Execution 37](#_TOC_250014)
   3. [Conditional ASIMD 38](#_TOC_250013)
   4. [Register Forwarding Hazards 38](#_TOC_250012)
   5. [Load/Store Throughput 39](#_TOC_250011)
   6. [Load/Store Alignment 39](#_TOC_250010)
   7. [Non-Temporal Loads/Stores 40](#_TOC_250009)
   8. [Branch Alignment 40](#_TOC_250008)
   9. [Setting Condition Flags 40](#_TOC_250007)
   10. [Accelerated Accumulator Forwarding in the Floating-PointPipelines 40](#_TOC_250006)
   11. [Load Balancing in the Floating-PointPipelines 41](#_TOC_250005)
   12. [Special Register Access 41](#_TOC_250004)
   13. [AES Encryption/Decryption 42](#_TOC_250003)
   14. [Fast literal generation 42](#_TOC_250002)
   15. [PC-relative address calculation 43](#_TOC_250001)
   16. [FPCR self-synchronization 43](#_TOC_250000)

# ABOUT THIS DOCUMENT

* 1. **References**

This document refers to the following documents.

|  |  |
| --- | --- |
| **Title** | **Location** |
| ARM Cortex-A57 MPCore Processor Technical Reference Manual | Infocenter.arm.com |

* 1. **Terms and abbreviations**

This document uses the following terms and abbreviations.

## Term Meaning

ALU ASIMD

µop VFP

Arithmetic/Logical Unit Advanced SIMD

Micro-Operation Vector Floating Point

* 1. **Document Scope**

This document provides high-level information about the Cortex-A57 pipeline, instruction performance characteristics, and special performance considerations. This information is intended to aid those who are optimizing software and compilers for Cortex-A57. For a more complete description of the Cortex-A57 processor, please refer to the *ARM Cortex-A57 MPCore Processor Technical Reference Manual*, available at infocenter.arm.com.

# INTRODUCTION

* 1. **Pipeline Overview**

The following diagram describes the high-level Cortex-A57 instruction processing pipeline. Instructions are first fetched, then decoded into internal micro-operations (µops). From there, the µops proceed through register renaming and dispatch stages. Once dispatched, µops wait for their operands and issue out-of-order to one of eight execution pipelines. Each execution pipeline can accept and complete one µop per cycle.

Store

Load

FP/ASIMD 1

FP/ASIMD 0

Integer Multi-Cycle

Integer 1

Integer 0

Branch

Issue

Decode, Rename, Dispatch

Fetch

IN ORDER

OUT OF ORDER

The execution pipelines support different types of operations, as follows.

|  |  |
| --- | --- |
| **Pipeline (mnemonic)** | **Supported functionality** |
| Branch (B) | Branch µops |
| Integer 0/1 (I) | Integer ALU µops |
| Multi-cycle (M) | Integer shift-ALU, multiply, divide, CRC and sum-of-absolute-differences  µopµops |
| Load (L) | Load and register transfer µops |
| Store (S) | Store and special memory µops |
| FP/ASIMD-0 (F0) | ASIMD ALU, ASIMD misc, FP misc, FP add, FP multiply, ASIMD integer multiply, FP divide µops, crypto µops |

|  |  |
| --- | --- |
| FP/ASIMD-1 (F1) | ASIMD ALU, ASIMD misc,FP misc, FP add,FP multiply,ASIMD shift µops |

# INSTRUCTION CHARACTERISTICS

* 1. **Instruction Tables**

This chapter describes high-level performance characteristics for most ARMv8 A32, T32 and A64 instructions. A series of tables summarize the effective execution latency and throughput, pipelines utilized, and special behaviors associated with each group of instructions. Utilized pipelines correspond to the execution pipelines described in chapter 2.

In the tables below, Exec Latency is defined as the minimum latency seen by an operation dependent on an instruction in the described group.

In the tables below, Execution Throughput is defined as the maximum throughput (in instructions / cycle) of the specified instruction group that can be achieved in the entirety of the Cortex-A57 microarchitecture .

* 1. **Branch Instructions**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch32 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| Branch, immed | B | 1 | 1 | B |  |
| Branch, register | BX | 1 | 1 | B |  |
| Branch and link, immed | BL, BLX | 1 | 1 | I0/I1, B |  |
| Branch and link, register != LR | BLX | 2 | 1 | I0/I1, B |  |
| Branch and link, register = LR | BLX | 3 | 1 | I0/I1, B |  |
| Compare and branch | CBZ, CBNZ | 1 | 1 | B |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch64 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| Branch, immed | B | 1 | 1 | B |  |
| Branch, register | BR, RET | 1 | 1 | B |  |
| Branch and link, immed | BL | 1 | 1 | I0/I1, B |  |
| Branch and link, register !=  LR | BLR | 2 | 1 | I0/I1, B |  |
| Branch and link, register =  LR | BLR | 3 | 1 | I0/I1, B |  |
| Compare and branch | CBZ, CBNZ, TBZ, TBNZ | 1 | 1 | B |  |

* 1. **Arithmetic and Logical Instructions**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch32 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| ALU, basic | ADD{S}, ADC{S}, ADR, AND{S},  BIC{S}, CMN, CMP, EOR{S},  ORN{S}, ORR{S}, RSB{S},  RSC{S}, SUB{S}, SBC{S}, TEQ, TST | 1 | 2 | I0/I1 |  |
| ALU, shift by immed | (same as above) | 2 | 1 | M |  |
| ALU, shift by register, unconditional | (same as above) | 2 | 1 | M |  |
| ALU, shift by register,  conditional | (same as above) | 2 | 1 | I0/I1 |  |
| ALU, branch forms |  | +2 | 1 | +B | 1 |

Notes:

* + 1. Branch forms are possible when the instruction destination register is the PC. For those cases, an additional branch µop is required. This adds 2 cycles to the latency.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch64 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| ALU, basic | ADD{S}, ADC{S}, AND{S},  BIC{S}, EON, EOR, ORN, ORR,  SUB{S}, SBC{S} | 1 | 2 | I0/I1 |  |
| ALU, extend and/or shift | ADD{S}, AND{S}, BIC{S}, EON,  EOR, ORN, ORR, SUB{S} | 2 | 1 | M |  |
| Conditional compare | CCMN, CCMP | 1 | 2 | I0/I1 |  |
| Conditional select | CSEL, CSINC, CSINV, CSNEG | 1 | 2 | I0/I1 |  |

* 1. **Move and Shift Instructions**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch32 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| Move, basic | MOV{S}, MOVW, MVN{S} | 1 | 2 | I0/I1 | 1 |
| Move, shift by immed, no  setflags | ASR, LSL, LSR, ROR, RRX, MVN | 1 | 2 | I0/I1 |  |
| Move, shift by immed,  setflags | ASRS, LSLS, LSRS, RORS, RRXS,  MVNS | 2 | 1 | M |  |
| Move, shift by register, no  setflags, unconditional | ASR, LSL, LSR, ROR, RRX, MVN | 1 | 2 | I0/I1 |  |
| Move, shift by register, no  setflags, conditional | ASR, LSL, LSR, ROR, RRX, MVN | 2 | 1 | I0/I1 |  |
| Move, shift by register, | ASRS, LSLS, LSRS, RORS, RRXS, | 2 | 1 | M |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| setflags, unconditional | MVNS |  |  |  |  |
| Move, shift by register,  setflags, conditional | ASRS, LSLS, LSRS, RORS, RRXS,  MVNS | 2 | 1 | I0/I1 |  |
| Move, top | MOVT | 2/1 | 1 on r0px, 2 on  r1px | M/I | 2 |
| (Move, branch forms) |  | +2 | 1 | +B | 3 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch64 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| Address generation | ADR, ADRP | 1 | 2 | I0/I1 | 4 |
| Move immed | MOVN, MOVK, MOVZ | 1 | 2 | I0/I1 | 1 |
| Variable shift | ASRV, LSLV, LSRV, RORV | 1 | 2 | I0/I1 |  |

Note:

* + 1. In Cortex-A57 r1p0 and later revisions, sequential MOVW/MOVT (AArch32) instruction pairs and certain MOVZ/MOVK, MOVK/MOVK (AArch64) instruction pairs can be executed with one-cycle execute latency and four-instruction/cycle execution throughput in I0/I1. See Section 4.14 for more details on the instruction pairs that can be merged.
    2. MOVT is implemented as a single-cycle µop in Cortex-A57 r1p0 and later. A latency given as “N/M” implies a latency of N cycles in r0pX revisions and M cycles in r1p0 and later revisions. A similar notation applies to utilized pipelines.
    3. Branch forms are possible when the instruction destination register is the PC. For those cases, an additional branch µop is required. This adds 2 cycles to the latency.
    4. In Cortex-A57 r1p0 and later revisions, sequential ADRP/ADD instruction pairs can be executed with one- cycle execute latency and four-instruction/cycle execution throughput in I0/I1. See Section 4.15 for more information.
  1. **Divide and Multiply Instructions**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch32 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| Divide | SDIV, UDIV | 4 -­‐ 20 | 1/20 -­‐ 1/4 | M | 1 |
| Multiply | MUL, SMULBB, SMULBT, SMULTB, SMULTT, SMULWB, SMULWT, SMMUL{R},  SMUAD{X}, SMUSD{X} | 3 | 1 | M |  |
| Multiply accumulate | MLA, MLS, SMLABB, SMLABT, SMLATB, SMLATT, SMLAWB, SMLAWT, SMLAD{X}, SMLSD{X}, SMMLA{R},  SMMLS{R} | 3 (1) | 1 | M | 2 |
| Multiply accumulate long | SMLAL, SMLALBB, SMLALBT,  SMLALTB, SMLALTT, | 4 (2) | 1/2 | M | 2, 3 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | SMLALD{X}, SMLSLD{X},  UMAAL, UMLAL |  |  |  |  |
| Multiply long | SMULL, UMULL | 4 | 1/2 | M | 3 |
| (Multiply, setflags forms) |  | +1 | (Same as  above) | +I0/I1 | 4 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch64 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized pipelines** | **Notes** |
| Divide, W-­‐form | SDIV, UDIV | 4-­‐20 | 1/20 – 1/4 | M | 1 |
| Divide, X-­‐form | SDIV, UDIV | 4-­‐36 | 1/36 -­‐ 1/4 | M | 1 |
| Multiply accumulate, W-­‐ form | MADD, MSUB | 3 (1) | 1 | M | 2 |
| Multiply accumulate, X-­‐ form | MADD, MSUB | 5 (3) | 1/3 | M | 2,5 |
| Multiply accumulate long | SMADDL, SMSUBL, UMADDL, UMSUBL | 3 (1) | 1 | M | 2 |
| Multiply high | SMULH, UMULH | 6 [3] | 1/4 | M | 6 |

Note:

* + 1. Integer divides are performed using a iterative algorithm and block any subsequent divide operations until complete. Early termination is possible, depending upon the data values.
    2. Multiply-accumulate pipelines support late-forwarding of accumulate operands from similar µops, allowing a typical sequence of multiply-accumulate µops to issue one every N cycles (accumulate latency N shown in parentheses).
    3. Long-form multiplies (which produce two result registers) stall the multiplier pipeline for one extra cycle.
    4. Multiplies that set the condition flags require an additional integer µop .
    5. X-form multiply accumulates stall the multiplier pipeline for 2 extra cycles.
    6. Multiply high operations stall the multiplier pipeline for N extra cycles before any other type M µop can be issued to that pipeline, with N shown in parentheses.
  1. **Saturating and Parallel Arithmetic Instructions**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch32 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized pipelines** | **Notes** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch32 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized pipelines** | **Notes** |
| Parallel arith, unconditional | SADD16, SADD8, SSUB16, SSUB8, UADD16, UADD8,  USUB16, USUB8 | 2 | 1 | M |  |
| Parallel arith, conditional | SADD16, SADD8, SSUB16, SSUB8, UADD16, UADD8,  USUB16, USUB8 | 2 (4) | 1/2 | M, I0/I1 | 1 |
| Parallel arith with  exchange, unconditional | SASX, SSAX, UASX, USAX | 3 | 1 | I0/I1, M |  |
| Parallel arith with  exchange, conditional | SASX, SSAX, UASX, USAX | 3 (5) | 1/2 | I0/I1, M | 1 |
| Parallel halving arith | SHADD16, SHADD8, SHSUB16, SHSUB8, UHADD16, UHADD8,  UHSUB16, UHSUB8 | 2 | 1 | M |  |
| Parallel halving arith with  exchange | SHASX, SHSAX, UHASX,  UHSAX | 3 | 1 | I0/I1, M |  |
| Parallel saturating arith | QADD16, QADD8, QSUB16,  QSUB8, UQADD16, UQADD8, UQSUB16, UQSUB8 | 2 | 1 | M |  |
| Parallel saturating arith  with exchange | QASX, QSAX, UQASX, UQSAX | 3 | 1 | I0/I1, M |  |
| Saturate | SSAT, SSAT16, USAT, USAT16 | 2 | 1 | M |  |
| Saturating arith | QADD, QSUB | 2 | 1 | M |  |
| Saturating doubling arith | QDADD, QDSUB | 3 | 1 | I0/I1, M |  |

NOTE 1 – Conditional GE-setting instructions require three extra µops and two additional cycles to conditionally update the GE field (GE latency shown in parentheses).

* 1. **Miscellaneous Data-Processing Instructions**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch32 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| Bit field extract | SBFX, UBFX | 1 | 2 | I0/I1 |  |
| Bit field insert/clear | BFI, BFC | 2 | 1 | M |  |
| Count leading zeros | CLZ | 1 | 2 | I0/I1 |  |
| Pack halfword | PKH | 2 | 1 | M |  |
| Reverse bits/bytes | RBIT, REV, REV16, REVSH | 1 | 2 | I0/I1 |  |
| Select bytes, unconditional | SEL | 1 | 2 | I0/I1 |  |
| Select bytes, conditional | SEL | 2 | 1 | I0/I1 |  |
| Sign/zero extend, normal | SXTB, SXTH, UXTB, UXTH | 1 | 2 | I0/I1 |  |
| Sign/zero extend, parallel | SXTB16, UXTB16 | 2 | 1 | M |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch32 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| Sign/zero extend and add,  normal | SXTAB, SXTAH,UXTAB, UXTAH | 2 | 1 | M |  |
| Sign/zero extend and add,  parallel | SXTAB16, UXTAB16 | 4 | 1/2 | M |  |
| Sum of absolute  differences | USAD8, USADA8 | 3 | 1 | M |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch64 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| Bitfield extract, one reg | EXTR | 1 | 2 | I0/I1 |  |
| Bitfield extract, two regs | EXTR | 3 | 1 | I0/I1, M |  |
| Bitfield move, basic | SBFM, UBFM | 1 | 2 | I0/I1 |  |
| Bitfield move, insert | BFM | 2 | 1 | M |  |
| Count leading | CLS, CLZ | 1 | 2 | I0/I1 |  |
| Reverse bits/bytes | RBIT, REV, REV16, REV32 | 1 | 2 | I0/I1 |  |

NOP instructions are resolved at dispatch and have a maximum execution throughput of 1 per cycle. Only 1 NOP can be dispatched per cycle.

* 1. **Load Instructions**

The latencies shown assume the memory access hits in the Level 1 Data Cache.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch32 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| Load, immed offset | LDR{T}, LDRB{T}, LDRD,  LDRH{T}, LDRSB{T}, LDRSH{T} | 4 | 1 | L |  |
| Load, register offset, plus | LDR, LDRB, LDRD, LDRH, LDRSB, LDRSH | 4 | 1 | L |  |
| Load, register offset, minus | LDR, LDRB, LDRD, LDRH, LDRSB, LDRSH | 5 | 1 | I0/I1, L |  |
| Load, scaled register  offset, plus LSL2 | LDR, LDRB | 4 | 1 | L |  |
| Load, scaled register offset, other | LDR, LDRB, LDRH, LDRSB, LDRSH | 5 | 1 | I0/I1, L |  |
| Load, immed pre-­‐indexed | LDR, LDRB, LDRD, LDRH, LDRSB, LDRSH | 4 (1) | 1 | L, I0/I1 | 1 |
| Load, register pre-­‐indexed | LDR, LDRB, LDRH, LDRSB, LDRSH | 4 (2) | 1 | I0/I1, L | 1 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch32 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| Load, register pre-­‐indexed | LDRD | 5 (2) | 1 | I0/I1, L | 1 |
| Load, scaled register pre-­‐  indexed, plus LSL2 | LDR, LDRB | 4 (2) | 1 | I0/I1, L | 1 |
| Load, scaled register pre-­‐  indexed, other | LDR, LDRB | 5 (2) | 1 | I0/I1, L | 1 |
| Load, immed post-­‐indexed | LDR{T}, LDRB{T}, LDRD,  LDRH{T}, LDRSB{T}, LDRSH{T} | 4 (1) | 1 | L, I0/I1 | 1 |
| Load, register post-­‐ indexed | LDR{T}, LDRB{T}, LDRD,  LDRH{T}, LDRSB{T}, LDRSH{T} | 4 (2) | 1 | I0/I1, L | 1 |
| Load, scaled register post-­‐  indexed | LDR, LDRB | 4 (2) | 1 | I0/I1, L | 1 |
| Load, scaled register post-­‐  indexed | LDRT, LDRBT | 4 (3) | 1 | I0/I1, L, M | 1 |
| Preload, immed offset | PLD, PLDW | 4 | 1 | L |  |
| Preload, register offset,  plus | PLD, PLDW | 4 | 1 | L |  |
| Preload, register offset,  minus | PLD, PLDW | 5 | 1 | I0/I1, L |  |
| Preload, scaled register  offset, plus LSL2 | PLD, PLDW | 4 | 1 | L |  |
| Preload, scaled register offset, other | PLD, PLDW | 5 | 1 | I0/I1, L |  |
| Load multiple, no writeback, base reg not in  list | LDMIA, LDMIB, LDMDA, LDMDB | 3 + N | 1/N | L | 2 |
| Load multiple, no  writeback, base reg in list | LDMIA, LDMIB, LDMDA,  LDMDB | 4 + N | 1/N | I0/I1, L | 2 |
| Load multiple, writeback | LDMIA, LDMIB, LDMDA,  LDMDB, POP | 3 + N | 1/N | L, I0/I1 | 1, 2 |
| Load, branch forms with addressing mode as register post-­‐indexed (scaled or unscaled) or scaled, register pre-­‐  indexed, plus, LSL2 | LDR | 4(2) | 1 | L, M | 1 |
| Load, branch forms with addressing mode register  pre-­‐indexed | LDR | 4(1) | 1 | L, I0/I1 | 1 |
| (Load, branch forms) |  | +2 |  | +B | 3 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch64 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch64 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| Load register, literal | LDR, LDRSW, PRFM | 4 | 1 | L |  |
| Load register, unscaled immed | LDUR, LDURB, LDURH, LDURSB, LDURSH, LDURSW, PRFUM | 4 | 1 | L |  |
| Load register, immed post-­‐ index | LDR, LDRB, LDRH, LDRSB, LDRSH, LDRSW | 4 (1) | 1 | L, I0/I1 | 1 |
| Load register, immed pre-­‐ index | LDR, LDRB, LDRH, LDRSB, LDRSH, LDRSW | 4 (1) | 1 | L, I0/I1 | 1 |
| Load register, immed unprivileged | LDTR, LDTRB, LDTRH, LDTRSB, LDTRSH, LDTRSW | 4 | 1 | L |  |
| Load register, unsigned immed | LDR, LDRB, LDRH, LDRSB, LDRSH, LDRSW, PRFM | 4 | 1 | L |  |
| Load register, register offset, basic | LDR, LDRB, LDRH, LDRSB, LDRSH, LDRSW, PRFM | 4 | 1 | L |  |
| Load register, register  offset, scale by 4/8 | LDR, LDRSW, PRFM | 4 | 1 | L |  |
| Load register, register  offset, scale by 2 | LDRH, LDRSH | 5 | 1 | I0/I1, L |  |
| Load register, register offset, extend | LDR, LDRB, LDRH, LDRSB, LDRSH, LDRSW, PRFM | 4 | 1 | L |  |
| Load register, register offset, extend, scale by  4/8 | LDR, LDRSW, PRFM | 4 | 1 | L |  |
| Load register, register  offset, extend, scale by 2 | LDRH, LDRSH | 5 | 1 | I0/I1, L |  |
| Load pair, immed offset,  normal | LDP, LDNP | 4 | 1 | L |  |
| Load pair, immed offset,  signed words, base != SP | LDPSW | 5 | 1/2 | I0/I1, L |  |
| Load pair, immed offset,  signed words, base = SP | LDPSW | 5 | 1/2 | L |  |
| Load pair, immed post-­‐  index, normal | LDP | 4 (1) | 1 | L, I0/I1 | 1 |
| Load pair, immed post-­‐  index, signed words | LDPSW | 5 (1) | 1/2 | L, I0/I1 | 1 |
| Load pair, immed pre-­‐  index, normal | LDP | 4 (1) | 1 | L, I0/I1 | 1 |
| Load pair, immed pre-­‐  index, signed words | LDPSW | 5 (1) | 1/2 | L, I0/I1 | 1 |

NOTE 1 – Base register updates are typically completed in parallel with the load operation and with shorter latency (update latency shown in parentheses).

NOTE 2 – For load multiple instructions, N=floor((num\_regs+1)/2).

NOTE 3 – Branch forms are possible when the instruction destination register is the PC. For those cases, an additional branch µop is required. This adds 2 cycles to the latency.

* 1. **Store Instructions**

The following table describes performance characteristics for standard store instructions. Stores µops may issue once their address operands are available and do not need to wait for data operands. Once executed, stores are buffered and committed in the background.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch32 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| Store, immed offset | STR{T}, STRB{T}, STRD, STRH{T} | 1 | 1 | S |  |
| Store, register offset, plus | STR, STRB, STRD, STRH | 1 | 1 | S |  |
| Store, register offset,  minus | STR, STRB, STRD, STRH | 3 | 1 | I0/I1, S |  |
| Store, scaled register  offset, plus LSL2 | STR, STRB | 1 | 1 | S |  |
| Store, scaled register  offset, other | STR, STRB | 3 | 1 | I0/I1, S |  |
| Store, immed pre-­‐indexed | STR, STRB, STRD, STRH | 1 (1) | 1 | S, I0/I1 | 1 |
| Store, register pre-­‐  indexed, plus | STR, STRB, STRD, STRH | 1 (1) | 1 | S, I0/I1 | 1 |
| Store, register pre-­‐  indexed, minus | STR, STRB, STRD, STRH | 3 (2) | 1 | I0/I1, S | 1 |
| Store, scaled register pre-­‐  indexed, plus LSL2 | STR, STRB | 1 (2) | 1 | S, M | 1 |
| Store, scaled register pre-­‐  indexed, other | STR, STRB | 3 (2) | 1 | I0/I1, S | 1 |
| Store, immed post-­‐ indexed | STR{T}, STRB{T}, STRD, STRH{T} | 1 (1) | 1 | S, I0/I1 | 1 |
| Store, register post-­‐  indexed | STRH{T}, STRD | 1 (1) | 1 | S, I0/I1 | 1 |
| Store, register post-­‐  indexed | STR{T}, STRB{T} | 1 (2) | 1 | S, M | 1 |
| Store, scaled register post-­‐  indexed | STR{T}, STRB{T} | 1 (2) | 1 | S, M | 1 |
| Store multiple, no  writeback | STMIA, STMIB, STMDA,  STMDB | N | 1/N | S | 1, 2 |
| Store multiple, writeback | STMIA, STMIB, STMDA,  STMDB, PUSH | N | 1/N | S, I0/I1 | 1, 2 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch64 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| Store register, unscaled  immed | STUR, STURB, STURH | 1 | 1 | S |  |
| Store register, immed  post-­‐index | STR, STRB, STRH | 1 (1) | 1 | S, I0/I1 | 1 |
| Store register, immed pre-­‐  index | STR, STRB, STRH | 1 (1) | 1 | S, I0/I1 | 1 |
| Store register, immed  unprivileged | STTR, STTRB, STTRH | 1 | 1 | S |  |
| Store register, unsigned  immed | STR, STRB, STRH | 1 | 1 | S |  |
| Store register, register  offset, basic | STR, STRB, STRH | 1 | 1 | S |  |
| Store register, register  offset, scaled by 4/8 | STR | 1 | 1 | S |  |
| Store register, register  offset, scaled by 2 | STRH | 3 | 1 | I0/I1, S |  |
| Store register, register  offset, extend | STR, STRB, STRH | 1 | 1 | S |  |
| Store register, register offset, extend, scale by  4/8 | STR | 1 | 1 | S |  |
| Store register, register  offset, extend, scale by 1 | STRH | 3 | 1 | I0/I1, S |  |
| Store pair, immed offset,  W-­‐form | STP, STNP | 1 | 1 | S |  |
| Store pair, immed offset,  X-­‐form | STP, STNP | 2 | 1/2 | S |  |
| Store pair, immed post-­‐  index, W-­‐form | STP | 1 (1) | 1 | S, I0/I1 | 1 |
| Store pair, immed post-­‐  index, X-­‐form | STP | 2 (1) | 1/2 | S, I0/I1 | 1 |
| Store pair, immed pre-­‐  index, W-­‐form | STP | 1 (1) | 1 | S, I0/I1 | 1 |
| Store pair, immed pre-­‐  index, X-­‐form | STP | 2 (1) | 1/2 | S, I0/I1 | 1 |

NOTE 1 – Base register updates are typically completed in parallel with the store operation and with shorter latency (update latency shown in parentheses).

NOTE 2 – For store multiple instructions, N=floor((num\_regs+1)/2).

* 1. **FP Data Processing Instructions**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch32 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| FP absolute value | VABS | 3 | 2 | F0/F1 |  |
| FP arith | VADD, VSUB | 5 | 2 | F0/F1 |  |
| FP compare, unconditional | VCMP, VCMPE | 3 | 1 | F1 |  |
| FP compare, conditional | VCMP, VCMPE | 6 | 1/6 | F0/F1, F1 |  |
| FP convert | VCVT{R}, VCVTB, VCVTT, VCVTA, VCVTM, VCVTN,  VCVTP | 5 | 2 | F0/F1 |  |
| FP round to integral | VRINTA, VRINTM, VRINTN, VRINTP, VRINTR, VRINTX,  VRINTZ | 5 | 2 | F0/F1 |  |
| FP divide, S-­‐form | VDIV | 7-­‐17 | 2/15-­‐2/5 | F0 | 1 |
| FP divide, D-­‐form | VDIV | 7-­‐32 | 1/30-­‐1/5 | F0 | 1 |
| FP max/min | VMAXNM, VMINNM | 5 | 2 | F0/F1 |  |
| FP multiply, FZ | VMUL, VNMUL | 5 | 2 | F0/F1 | 2 |
| FP multiply, no FZ | VMUL, VNMUL | 6 | 2 | F0/F1 | 2 |
| FP multiply accumulate, FZ | VFMA, VFMS, VFNMA, VFNMS, VMLA, VMLS,  VNMLA, VNMLS | 9 (4) | 2 | F0/F1 | 3 |
| FP multiply accumulate, no FZ | VFMA, VFMS, VFNMA, VFNMS, VMLA, VMLS,  VNMLA, VNMLS | 10 (4) | 2 | F0/F1 | 3 |
| FP negate | VNEG | 3 | 2 | F0/F1 |  |
| FP select | VSELEQ, VSELGE, VSELGT,  VSELVS | 3 | 2 | F0/F1 |  |
| FP square root, S-­‐form | VSQRT | 7-­‐17 | 2/15-­‐2/5 | F0 | 1 |
| FP square root, D-­‐form | VSQRT | 7-­‐32 | 1/30-­‐1/5 | F0 | 1 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch64 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| FP absolute value | FABS | 3 | 2 | F0/F1 |  |
| FP arithmetic | FADD, FSUB | 5 | 2 | F0/F1 |  |
| FP compare | FCCMP{E}, FCMP{E} | 3 | 1 | F1 |  |
| FP divide, S-­‐form | FDIV | 7-­‐17 | 2/15-­‐2/5 | F0 | 1 |
| FP divide, D-­‐form | FDIV | 7-­‐32 | 1/30-­‐1/5 | F0 | 1 |
| FP min/max | FMIN, FMINNM, FMAX,  FMAXNM | 5 | 2 | F0/F1 |  |
| FP multiply, FZ | FMUL, FNMUL | 5 | 2 | F0/F1 | 2 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch64 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| FP multiply, no FZ | FMUL, FNMUL | 6 | 2 | F0/F1 | 2 |
| FP multiply accumulate, FZ | FMADD, FMSUB, FNMADD,  FNMSUB | 9 (4) | 2 | F0/F1 | 3 |
| FP multiply accumulate,  no FZ | FMADD, FMSUB, FNMADD,  FNMSUB | 10 (4) | 2 | F0/F1 | 3 |
| FP negate | FNEG | 3 | 2 | F0/F1 |  |
| FP round to integral | FRINTA, FRINTI, FRINTM, FRINTN, FRINTP, FRINTX,  FRINTZ | 5 | 2 | F0/F1 |  |
| FP select | FCSEL | 3 | 2 | F0/F1 |  |
| FP square root, S-­‐form | FSQRT | 7-­‐17 | 2/15-­‐2/5 | F0 | 1 |
| FP square root, D-­‐form | FSQRT | 7-­‐32 | 1/30-­‐1/5 | F0 | 1 |

NOTE 1 – FP divide and square root operations are performed using an iterative algorithm and block subsequent similar operations to the same pipeline until complete.

NOTE 2 – FP multiply-accumulate pipelines support late forwarding of the result from FP multiply µops to the accumulate operands of an FP multiply-accumulate µop . The latter can potentially be issued 1 cycle after the FP multiply µop has been issued.

NOTE 3 – FP multiply-accumulate pipelines support late-forwarding of accumulate operands from similar µops, allowing a typical sequence of multiply-accumulate µops to issue one every N cycles (accumulate latency N shown in parentheses).

* 1. **FP Miscellaneous Instructions**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch32 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| FP move, immed | VMOV | 3 | 2 | F0/F1 |  |
| FP move, register | VMOV | 3 | 2 | F0/F1 |  |
| FP transfer, vfp to core reg | VMOV | 5 | 1 | L |  |
| FP transfer, core reg to upper or lower half of vfp  D-­‐reg | VMOV | 8 | 1 | L, F0/F1 |  |
| FP transfer, core reg to vfp | VMOV | 5 | 1 | L |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch64 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| FP convert, from vec to  vec reg | FCVT, FCVTXN | 5 | 1 | F0/F1 |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch64 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| FP convert, from gen to  vec reg | SCVTF, UCVTF | 10 | 1 | L, F0/F1 |  |
| FP convert, from vec to gen reg | FCVTAS, FCVTAU, FCVTMS, FCVTMU, FCVTNS, FCVTNU, FCVTPS, FCVTPU, FCVTZS,  FCVTZU | 10 | 1 | L, F0/F1 |  |
| FP move, immed | FMOV | 3 | 2 | F0/F1 |  |
| FP move, register | FMOV | 3 | 2 | F0/F1 |  |
| FP transfer, from gen to  vec reg | FMOV | 5 | 1 | L |  |
| FP transfer, from vec to  gen reg | FMOV | 5 | 1 | L |  |

* 1. **FP Load Instructions**

The latencies shown assume the memory access hits in the Level 1 Data Cache. Compared to standard loads, an extra cycle is required to forward results to FP/ASIMD pipelines.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch32 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| FP load, register | VLDR | 5 | 1 | L |  |
| FP load multiple, unconditional | VLDMIA, VLDMDB,  VPOP | 4 + N | 1/N | L | 1 |
| FP load multiple, conditional | VLDMIA, VLDMDB,  VPOP | 4 + N | 1/N | L | 2 |
| (FP load, writeback forms) |  | (1) | Same as before | +I0/I1 | 3 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch64 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| Load vector reg, literal | LDR | 5 | 1 | L |  |
| Load vector reg, unscaled immed | LDUR | 5 | 1 | L |  |
| Load vector reg, immed post-­‐index | LDR | 5 (1) | 1 | L, I0/I1 | 3 |
| Load vector reg, immed pre-­‐index | LDR | 5 (1) | 1 | L, I0/I1 | 3 |
| Load vector reg, unsigned immed | LDR | 5 | 1 | L |  |
| Load vector reg, register offset, basic | LDR | 5 | 1 | L |  |
| Load vector reg, register offset, scale,  S/D-­‐form | LDR | 5 | 1 | L |  |
| Load vector reg, register offset, scale,  H/Q-­‐form | LDR | 6 | 1 | I0/I1, L |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch64 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| Load vector reg, register offset,  extend | LDR | 5 | 1 | L |  |
| Load vector reg, register offset,  extend, scale, S/D-­‐form | LDR | 5 | 1 | L |  |
| Load vector reg, register offset,  extend, scale, H/Q-­‐form | LDR | 6 | 1 | I0/I1, L |  |
| Load vector pair, immed offset, S/D-­‐  form | LDP, LDNP | 5 | 1 | L |  |
| Load vector pair, immed offset, Q-­‐  form | LDP, LDNP | 6 | 1/2 | L |  |
| Load vector pair, immed post-­‐index,  S/D-­‐form | LDP | 5 (1) | 1 | L, I0/I1 | 3 |
| Load vector pair, immed post-­‐index,  Q-­‐form | LDP | 6 (1) | 1/2 | L, I0/I1 | 3 |
| Load vector pair, immed pre-­‐index,  S/D-­‐form | LDP | 5 (1) | 1 | L, I0/I1 | 3 |
| Load vector pair, immed pre-­‐index, Q-­‐  form | LDP | 6 (1) | 1/2 | L, I0/I1 | 3 |

NOTE 1 – For FP load multiple instructions, N=floor((num\_regs+1)/2) for unconditional forms only. NOTE 2 – For conditional FP load multiple instructions, N = num\_regs for conditional forms only.

NOTE 3 – Writeback forms of load instructions require an extra µop to update the base address. This update is typically performed in parallel with or prior to the load µop (update latency shown in parentheses).

* 1. **FP Store Instructions**

Stores µops may issue once their address operands are available and do not need to wait for data operands. Once executed, stores are buffered and committed in the background.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **Aarch32 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| FP store, immed offset | VSTR | 1 | 1 | S |  |
| FP store multiple, S-­‐form | VSTMIA, VSTMDB,  VPUSH | N | 1/N | S | 1 |
| FP store multiple, D-­‐form | VSTMIA, VSTMDB,  VPUSH | N | 1/N | S | 1 |
| (FP store, writeback forms) |  | (1) | Same as before | +I0/I1 | 2 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch64 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch64 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| Store vector reg, unscaled immed,  B/H/S/D-­‐form | STUR | 1 | 1 | S |  |
| Store vector reg, unscaled immed, Q-­‐form | STUR | 2 | 1/2 | S |  |
| Store vector reg, immed post-­‐index,  B/H/S/D-­‐form | STR | 1 (1) | 1 | S, I0/I1 | 2 |
| Store vector reg, immed post-­‐index, Q-­‐  form | STR | 2 (1) | 1/2 | S, I0/I1 | 2 |
| Store vector reg, immed pre-­‐index,  B/H/S/D-­‐form | STR | 1 (1) | 1 | S, I0/I1 | 2 |
| Store vector reg, immed pre-­‐index, Q-­‐form | STR | 2 (1) | 1/2 | I0/I1, S | 2 |
| Store vector reg, unsigned immed,  B/H/S/D-­‐form | STR | 1 | 1 | S |  |
| Store vector reg, unsigned immed, Q-­‐form | STR | 2 | 1/2 | I0/I1, S |  |
| Store vector reg, register offset, basic,  B/H/S/D-­‐form | STR | 1 | 1 | S |  |
| Store vector reg, register offset, basic, Q-­‐  form | STR | 2 | 1/2 | I0/I1, S |  |
| Store vector reg, register offset, scale, H-­‐  form | STR | 3 | 1 | I0/I1, S |  |
| Store vector reg, register offset, scale, S/D-­‐  form | STR | 1 | 1 | S |  |
| Store vector reg, register offset, scale, Q-­‐  form | STR | 4 | 1/2 | I0/I1, S |  |
| Store vector reg, register offset, extend,  B/H/S/D-­‐form | STR | 1 | 1 | S |  |
| Store vector reg, register offset, extend, Q-­‐  form | STR | 4 | 1/2 | M, S |  |
| Store vector reg, register offset, extend,  scale, H-­‐form | STR | 3 | 1 | I0/I1, S |  |
| Store vector reg, register offset, extend,  scale, S/D-­‐form | STR | 1 | 1 | S |  |
| Store vector reg, register offset, extend,  scale, Q-­‐form | STR | 4 | 1/2 | I0/I1, S |  |
| Store vector pair, immed offset, S-­‐form | STP | 1 | 1 | S |  |
| Store vector pair, immed offset, D-­‐form | STP | 2 | 1/2 | S S |  |
| Store vector pair, immed offset, Q-­‐form | STP | 4 | 1/4 | I0/I1, S |  |
| Store vector pair, immed post-­‐index, S-­‐  form | STP | 1 (1) | 1 | S, I0/I1 | 2 |
| Store vector pair, immed post-­‐index, D-­‐  form | STP | 2 (1) | 1/2 | S, I0/I1 | 2 |
| Store vector pair, immed post-­‐index, Q-­‐  form | STP | 4 (1) | 1/4 | S, I0/I1 | 2 |
| Store vector pair, immed pre-­‐index, S-­‐form | STP | 1 (1) | 1 | S, I0/I1 | 2 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch64 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| Store vector pair, immed pre-­‐index, D-­‐  form | STP | 2 (1) | 1/2 | S, I0/I1 | 2 |
| Store vector pair, immed pre-­‐index, Q-­‐  form | STP | 4 (1) | 1/4 | I0/I1, S | 2 |

NOTE 1 – For single precision store multiple instructions, N=floor((num\_regs+1)/2). For double-precision stores, N=(num\_regs).

NOTE 2 – Writeback forms of store instructions require an extra µop to update the base address. This update is typically performed in parallel with or prior to the store µop µop (address update latency shown in parentheses).

* 1. **ASIMD Integer Instructions**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch32 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| ASIMD absolute diff, D-­‐form | VABD | 3 | 2 | F0/F1 |  |
| ASIMD absolute diff, Q-­‐form | VABD | 3 | 1 | F0/ F1 |  |
| ASIMD absolute diff accum, D-­‐  form | VABA | 4 (1) | 1 | F1 | 2 |
| ASIMD absolute diff accum, Q-­‐  form | VABA | 5 (2) | 1/2 | F1 | 2 |
| ASIMD absolute diff accum long | VABAL | 4 (1) | 1 | F1 | 2 |
| ASIMD absolute diff long | VABDL | 3 | 2 | F0/F1 |  |
| ASIMD arith, basic | VADD, VADDL, VADDW, VNEG, VPADD, VPADDL,  VSUB, VSUBL, VSUBW | 3 | 2 | F0/F1 |  |
| ASIMD arith, complex | VABS, VADDHN, VHADD, VHSUB, VQABS, VQADD, VQNEG, VQSUB, VRADDHN, VRHADD,  VRSUBHN, VSUBHN | 3 | 2 | F0/F1 |  |
| ASIMD compare | VCEQ, VCGE, VCGT,  VCLE, VTST | 3 | 2 | F0/F1 |  |
| ASIMD logical | VAND, VBIC, VMVN, VORR, VORN, VEOR | 3 | 2 | F0/F1 |  |
| ASIMD max/min | VMAX, VMIN, VPMAX,  VPMIN | 3 | 2 | F0/F1 |  |
| ASIMD multiply, D-­‐form | VMUL, VQDMULH,  VQRDMULH | 5/4 | 1 | F0 | 4 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch32 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| ASIMD multiply, Q-­‐form | VMUL, VQDMULH,  VQRDMULH | 6/5 | 1/2 | F0 | 4 |
| ASIMD multiply accumulate, D-­‐  form | VMLA, VMLS | 5/4 (1) | 1 | F0 | 1, 4 |
| ASIMD multiply accumulate, Q-­‐  form | VMLA, VMLS | 6/5 (2) | 1/2 | F0 | 1, 4 |
| ASIMD multiply accumulate long | VMLAL, VMLSL | 5/4 (1) | 1 | F0 | 1, 4 |
| ASIMD multiply accumulate  saturating long | VQDMLAL, VQDMLSL | 5/4 (2) | 1 | F0 | 1, 4 |
| ASIMD multiply long | VMULL.S, VMULL.I,  VMULL.P8, VQDMULL | 5/4 | 1 | F0 | 4 |
| ASIMD pairwise add and  accumulate | VPADAL | 4 (1) | 1 | F1 | 2 |
| ASIMD shift accumulate | VSRA, VRSRA | 4 (1) | 1 | F1 | 2 |
| ASIMD shift by immed, basic | VMOVL, VSHL, VSHLL,  VSHR, VSHRN | 3 | 1 | F1 |  |
| ASIMD shift by immed, complex | VQRSHRN, VQRSHRUN, VQSHL{U}, VQSHRN, VQSHRUN, VRSHR,  VRSHRN | 4 | 1 | F1 |  |
| ASIMD shift by immed and  insert, basic, D-­‐form | VSLI, VSRI | 3 | 1 | F1 |  |
| ASIMD shift by immed and  insert, basic, Q-­‐form | VSLI, VSRI | 4 | 1/2 | F1 |  |
| ASIMD shift by register, basic, D-­‐  form | VSHL | 3 | 1 | F1 |  |
| ASIMD shift by register, basic, Q-­‐  form | VSHL | 4 | 1/2 | F1 |  |
| ASIMD shift by register,  complex, D-­‐form | VQRSHL, VQSHL, VRSHL | 4 | 1 | F1 |  |
| ASIMD shift by register,  complex, Q-­‐form | VQRSHL, VQSHL, VRSHL | 5 | 1/2 | F1 |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch64 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| ASIMD absolute diff, D-­‐form | SABD, UABD | 3 | 2 | F0/F1 |  |
| ASIMD absolute diff, Q-­‐form | SABD, UABD | 3 | 1 | F0/F1 |  |
| ASIMD absolute diff accum, D-­‐  form | SABA, UABA | 4 (1) | 1 | F1 | 2 |
| ASIMD absolute diff accum, Q-­‐  form | SABA, UABA | 5 (2) | 1/2 | F1 | 2 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch64 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| ASIMD absolute diff accum long | SABAL(2), UABAL(2) | 4 (1) | 1 | F1 | 2 |
| ASIMD absolute diff long | SABDL, UABDL | 3 | 2 | F0/F1 |  |
| ASIMD arith, basic | ABS, ADD, ADDP, NEG, SADDL(2), SADDLP, SADDW(2), SHADD, SHSUB, SSUBL(2), SSUBW(2), SUB, UADDL(2), UADDLP, UADDW(2), UHADD,  UHSUB, USUBW(2) | 3 | 2 | F0/F1 |  |
| ASIMD arith, complex | ADDHN(2), RADDHN(2), RSUBHN(2), SQABS, SQADD, SQNEG, SQSUB, SRHADD, SUBHN(2), SUQADD, UQADD, UQSUB,  URHADD, USQADD | 3 | 2 | F0/F1 |  |
| ASIMD arith, reduce, 4H/4S | ADDV, SADDLV,  UADDLV | 4 | 1 | F1 |  |
| ASIMD arith, reduce, 8B/8H | ADDV, SADDLV, UADDLV | 7 | 1 | F1, F0/F1 |  |
| ASIMD arith, reduce, 16B | ADDV, SADDLV,  UADDLV | 8 | 1/2 | F1 |  |
| ASIMD compare | CMEQ, CMGE, CMGT, CMHI, CMHS, CMLE,  CMLT, CMTST | 3 | 2 | F0/F1 |  |
| ASIMD logical | AND, BIC, EOR, MOV,  MVN, ORN, ORR | 3 | 2 | F0/F1 |  |
| ASIMD max/min, basic | SMAX, SMAXP, SMIN, SMINP, UMAX,  UMAXP, UMIN, UMINP | 3 | 2 | F0/F1 |  |
| ASIMD max/min, reduce, 4H/4S | SMAXV, SMINV,  UMAXV, UMINV | 4 | 1 | F1 |  |
| ASIMD max/min, reduce, 8B/8H | SMAXV, SMINV,  UMAXV, UMINV | 7 | 1 | F1, F0/F1 |  |
| ASIMD max/min, reduce, 16B | SMAXV, SMINV,  UMAXV, UMINV | 8 | 1/2 | F1 |  |
| ASIMD multiply, D-­‐form | MUL, PMUL,  SQDMULH, SQRDMULH | 5/4 | 1 | F0 | 4 |
| ASIMD multiply, Q-­‐form | MUL, PMUL,  SQDMULH, SQRDMULH | 6/5 | 1/2 | F0 | 4 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch64 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| ASIMD multiply accumulate, D-­‐  form | MLA, MLS | 5/4 (1) | 1 | F0 | 4 |
| ASIMD multiply accumulate, Q-­‐  form | MLA, MLS | 6/5 (2) | 1/2 | F0 | 4 |
| ASIMD multiply accumulate long | SMLAL(2), SMLSL(2), UMLAL(2), UMLSL(2) | 5/4 (1) | 1 | F0 | 1, 4 |
| ASIMD multiply accumulate  saturating long | SQDMLAL(2),  SQDMLSL(2) | 5/4 (2) | 1 | F0 | 1, 4 |
| ASIMD multiply long | SMULL(2), UMULL(2),  SQDMULL(2) | 5/4 | 1 | F0 | 4 |
| ASIMD polynomial (8x8) multiply  long | PMULL.8B,  PMULL2.16B | 5/4 | 1 | F0 | 3, 4 |
| ASIMD pairwise add and  accumulate | SADALP, UADALP | 4 (1) | 1 | F1 | 2 |
| ASIMD shift accumulate | SRA, SRSRA, USRA,  URSRA | 4 (1) | 1 | F1 | 2 |
| ASIMD shift by immed, basic | SHL, SHLL(2), SHRN(2), SLI, SRI, SSHLL(2), SSHR, SXTL(2), USHLL(2), USHR,  UXTL(2) | 3 | 1 | F1 |  |
| ASIMD shift by immed and  insert, basic, D-­‐form | SLI, SRI | 3 | 1 | F1 |  |
| ASIMD shift by immed and  insert, basic, Q-­‐form | SLI, SRI | 4 | 1/2 | F1 |  |
| ASIMD shift by immed, complex | RSHRN(2), SRSHR, SQSHL{U}, SQRSHRN(2), SQRSHRUN(2), SQSHRN(2), SQSHRUN(2), URSHR, UQSHL, UQRSHRN(2),  UQSHRN(2) | 4 | 1 | F1 |  |
| ASIMD shift by register, basic, D-­‐ form | SSHL, USHL | 3 | 1 | F1 |  |
| ASIMD shift by register, basic, Q-­‐  form | SSHL, USHL | 4 | 1/2 | F1 |  |
| ASIMD shift by register, complex, D-­‐form | SRSHL, SQRSHL, SQSHL, URSHL, UQRSHL,  UQSHL | 4 | 1 | F1 |  |
| ASIMD shift by register, complex, Q-­‐form | SRSHL, SQRSHL, SQSHL, URSHL, UQRSHL,  UQSHL | 5 | 1/2 | F1 |  |

NOTE 1 – Multiply-accumulate pipelines support late-forwarding of accumulate operands from similar µops, allowing a typical sequence of integer multiply-accumulate µops to issue one every cycle or one every other cycle(accumulate latency shown in parentheses).

NOTE 2 – Other accumulate pipelines also support late-forwarding of accumulate operands from similar µops, allowing a typical sequence of such µops to issue one every cycle (accumulate latency shown in parentheses).

NOTE 3 – This category includes instructions of the form “PMULL Vd.8H, Vn.8B, Vm.8B” and “PMULL2 Vd.8H, Vn.16B, Vm.16B”

NOTE 4 – Cortex-A57 r1p0 and later reduce the latency of ASIMD multiply and multiply-with-accumulate instructions relative to r0pX. Latencies listed as ‘N/M’ imply a latency of N on r0pX and M on r1p0.

* 1. **ASIMD Floating-Point Instructions**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch32 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| ASIMD FP absolute value | VABS | 3 | 2 | F0/F1 |  |
| ASIMD FP arith, D-­‐form | VABD, VADD, VPADD,  VSUB | 5 | 2 | F0/F1 |  |
| ASIMD FP arith, Q-­‐form | VABD, VADD, VSUB | 5 | 1 | F0/F1 |  |
| ASIMD FP compare, D-­‐form | VACGE, VACGT, VACLE, VACLT, VCEQ, VCGE,  VCGT, VCLE | 5 | 2 | F0/F1 |  |
| ASIMD FP compare, Q-­‐form | VACGE, VACGT, VACLE,  VACLT, VCEQ, VCGE, VCGT, VCLE | 5 | 1 | F0/F1 |  |
| ASIMD FP convert, integer, D-­‐  form | VCVT, VCVTA, VCVTM,  VCVTN, VCVTP | 5 | 2 | F0/F1 |  |
| ASIMD FP convert, integer, Q-­‐  form | VCVT, VCVTA, VCVTM,  VCVTN, VCVTP | 5 | 1 | F0/F1 |  |
| ASIMD FP convert, fixed, D-­‐form | VCVT | 5 | 2 | F0/F1 |  |
| ASIMD FP convert, fixed, Q-­‐form | VCVT | 5 | 1 | F0/F1 |  |
| ASIMD FP convert, half-­‐precision | VCVT | 8 | 2/3 | F0/F1 |  |
| ASIMD FP max/min, D-­‐form | VMAX, VMIN, VPMAX, VPMIN, VMAXNM,  VMINNM | 5 | 2 | F0/F1 |  |
| ASIMD FP max/min, Q-­‐form | VMAX, VMIN,  VMAXNM, VMINNM | 5 | 1 | F0/F1 |  |
| ASIMD FP multiply, D-­‐form | VMUL | 5 | 2 | F0/F1 | 2 |
| ASIMD FP multiply, Q-­‐form | VMUL | 5 | 1 | F0/F1 | 2 |
| ASIMD FP multiply accumulate,  D-­‐form | VMLA, VMLS, VFMA,  VFMS | 9 (4) | 2 | F0/F1 | 1 |
| ASIMD FP multiply accumulate, | VMLA, VMLS, VFMA, | 9 (4) | 1 | F0/F1 | 1 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch32 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| Q-­‐form | VFMS |  |  |  |  |
| ASIMD FP negate | VNEG | 3 | 2 | F0/F1 |  |
| ASIMD FP round to integral, D-­‐ form | VRINTA, VRINTM, VRINTN, VRINTP, VRINTX, VRINTZ | 5 | 2 | F0/F1 |  |
| ASIMD FP round to integral, Q-­‐ form | VRINTA, VRINTM, VRINTN, VRINTP, VRINTX, VRINTZ | 5 | 1 | F0/F1 |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch64 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| ASIMD FP absolute value | FABS | 3 | 2 | F0/F1 |  |
| ASIMD FP arith, normal, D-­‐form | FABD, FADD, FSUB | 5 | 2 | F0/F1 |  |
| ASIMD FP arith, normal, Q-­‐form | FABD, FADD, FSUB | 5 | 1 | F0/F1 |  |
| ASIMD FP arith, pairwise, D-­‐form | FADDP | 5 | 2 | F0/F1 |  |
| ASIMD FP arith, pairwise, Q-­‐ form | FADDP | 8 | 2/3 | F0/F1 |  |
| ASIMD FP compare, D-­‐form | FACGE, FACGT, FCMEQ, FCMGE, FCMGT,  FCMLE, FCMLT | 5 | 2 | F0/F1 |  |
| ASIMD FP compare, Q-­‐form | FACGE, FACGT, FCMEQ,  FCMGE, FCMGT, FCMLE, FCMLT | 5 | 1 | F0/F1 |  |
| ASIMD FP convert, long | FCVTL(2) | 8 | 2/3 | F0/F1 |  |
| ASIMD FP convert, narrow | FCVTN(2), FCVTXN(2) | 8 | 2/3 | F0/F1 |  |
| ASIMD FP convert, other, D-­‐form | FCVTAS, VCVTAU, FCVTMS, FCVTMU, FCVTNS, FCVTNU, FCVTPS, FCVTPU, FCVTZS, FCVTZU,  SCVTF, UCVTF | 5 | 2 | F0/F1 |  |
| ASIMD FP convert, other, Q-­‐ form | FCVTAS, VCVTAU, FCVTMS, FCVTMU, FCVTNS, FCVTNU, FCVTPS, FCVTPU, FCVTZS, FCVTZU,  SCVTF, UCVTF | 5 | 1 | F0/F1 |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch64 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| ASIMD FP divide, D-­‐form, F32 | FDIV | 7-­‐17 | 1/15-­‐1/5 | F0 | 3 |
| ASIMD FP divide, Q-­‐form, F32 | FDIV | 14-­‐34 | 1/30-­‐1/10 | F0 | 3 |
| ASIMD FP divide, Q-­‐form, F64 | FDIV | 14-­‐64 | 1/60-­‐1/10 | F0 | 3 |
| ASIMD FP max/min, normal, D-­‐  form | FMAX, FMAXNM,  FMIN, FMINNM | 5 | 2 | F0/F1 |  |
| ASIMD FP max/min, normal, Q-­‐  form | FMAX, FMAXNM,  FMIN, FMINNM | 5 | 1 | F0/F1 |  |
| ASIMD FP max/min, pairwise, D-­‐  form | FMAXP, FMAXNMP,  FMINP, FMINNMP | 5 | 2 | F0/F1 |  |
| ASIMD FP max/min, pairwise, Q-­‐  form | FMAXP, FMAXNMP,  FMINP, FMINNMP | 9 | 2/3 | F0/F1 |  |
| ASIMD FP max/min, reduce | FMAXV, FMAXNMV,  FMINV, FMINNMV | 10 | 1 | F0/F1 |  |
| ASIMD FP multiply, D-­‐form, FZ | FMUL, FMULX | 5 | 2 | F0/F1 | 2 |
| ASIMD FP multiply, D-­‐form, no  FZ | FMUL, FMULX | 6 | 2 | F0/F1 | 2 |
| ASIMD FP multiply, Q-­‐form, FZ | FMUL, FMULX | 5 | 1 | F0/F1 | 2 |
| ASIMD FP multiply, Q-­‐form, no  FZ | FMUL, FMULX | 6 | 1 | F0/F1 | 2 |
| ASIMD FP multiply accumulate,  D-­‐form, FZ | FMLA, FMLS | 9 (4) | 2 | F0/F1 | 1 |
| ASIMD FP multiply accumulate,  D-­‐form, no FZ | FMLA, FMLS | 9 (4) | 2 | F0/F1 | 1 |
| ASIMD FP multiply accumulate,  Q-­‐form, FZ | FMLA, FMLS | 10 (4) | 1 | F0/F1 | 1 |
| ASIMD FP multiply accumulate,  Q-­‐form, no FZ | FMLA, FMLS | 10 (4) | 1 | F0/F1 | 1 |
| ASIMD FP negate | FNEG | 3 | 2 | F0/F1 |  |
| ASIMD FP round, D-­‐form | FRINTA, FRINTI, FRINTM, FRINTN, FRINTP, FRINTX,  FRINTZ | 5 | 2 | F0/F1 |  |
| ASIMD FP round, Q-­‐form | FRINTA, FRINTI, FRINTM, FRINTN, FRINTP, FRINTX,  FRINTZ | 5 | 1 | F0/F1 |  |

NOTE 1 – ASIMD multiply-accumulate pipelines support late-forwarding of accumulate operands from similar

µops, allowing a typical sequence of floating-point multiply-accumulate µops to issue one every four cycles (accumulate latency shown in parentheses).

NOTE 2 - ASIMD multiply-accumulate pipelines support late forwarding of the result from ASIMD FP multiply µops to the accumulate operands of an ASIMD FP multiply-accumulate µop . The latter can potentially be issued 1 cycle after the ASIMD FP multiply µop has been issued.

NOTE 3 – FP divide operations are performed using an iterative algorithm and block subsequent similar operations to the same pipeline until complete.

* 1. **ASIMD Miscellaneous Instructions**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch32 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| ASIMD bitwise insert, D-­‐form | VBIF, VBIT, VBSL | 3 | 2 | F0/F1 |  |
| ASIMD bitwise insert, Q-­‐form | VBIF, VBIT, VBSL | 3 | 1 | F0/F1 |  |
| ASIMD count, D-­‐form | VCLS, VCLZ, VCNT | 3 | 2 | F0/F1 |  |
| ASIMD count, Q-­‐form | VCLS, VCLZ, VCNT | 3 | 1 | F0/F1 |  |
| ASIMD duplicate, core reg, D-­‐  form | VDUP | 8 | 1 | L, F0/F1 |  |
| ASIMD duplicate, core reg, Q-­‐  form | VDUP | 8 | 1 | L, F0/F1 |  |
| ASIMD duplicate, scalar | VDUP | 3 | 2 | F0/F1 |  |
| ASIMD extract | VEXT | 3 | 2 | F0/F1 |  |
| ASIMD move, immed | VMOV | 3 | 2 | F0/F1 |  |
| ASIMD move, register | VMOV | 3 | 2 | F0/F1 |  |
| ASIMD move, narrowing | VMOVN | 3 | 2 | F0/F1 |  |
| ASIMD move, saturating | VQMOVN, VQMOVUN | 4 | 1 | F1 |  |
| ASIMD reciprocal estimate, D-­‐  form | VRECPE, VRSQRTE | 5 | 2 | F0/F1 |  |
| ASIMD reciprocal estimate, Q-­‐  form | VRECPE, VRSQRTE | 5 | 1 | F0/F1 |  |
| ASIMD reciprocal step, D-­‐form,  FZ | VRECPS, VRSQRTS | 9 | 2 | F0/F1 |  |
| ASIMD reciprocal step, D-­‐form,  no FZ | VRECPS, VRSQRTS | 10 | 2 | F0/F1 |  |
| ASIMD reciprocal step, Q-­‐form,  FZ | VRECPS, VRSQRTS | 9 | 1 | F0/F1 |  |
| ASIMD reciprocal step, Q-­‐form,  no FZ | VRECPS, VRSQRTS | 10 | 1 | F0/F1 |  |
| ASIMD reverse | VREV16, VREV32,  VREV64 | 3 | 2 | F0/F1 |  |
| ASIMD swap, D-­‐form | VSWP | 3 | 2 | F0/F1 |  |
| ASIMD swap, Q-­‐form | VSWP | 3 | 1 | F0/F1 |  |
| ASIMD table lookup, 1 reg | VTBL, VTBX | 3 | 2 | F0/F1 |  |
| ASIMD table lookup, 2 reg | VTBL, VTBX | 3 | 2 | F0/F1 |  |
| ASIMD table lookup, 3 reg | VTBL, VTBX | 6 | 2 | F0/F1 |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch32 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| ASIMD table lookup, 4 reg | VTBL, VTBX | 6 | 2 | F0/F1 |  |
| ASIMD transfer, scalar to core  reg | VMOV | 6 | 1 | L, I0/I1 |  |
| ASIMD transfer, core reg to  scalar | VMOV | 8 | 1 | L, F0/F1 |  |
| ASIMD transpose, D-­‐form | VTRN | 3 | 2 | F0/F1 |  |
| ASIMD transpose, Q-­‐form | VTRN | 3 | 1 | F0/F1 |  |
| ASIMD unzip/zip, D-­‐form | VUZP, VZIP | 3 | 2 | F0/F1 |  |
| ASIMD unzip/zip, Q-­‐form | VUZP, VZIP | 6 | 2/3 | F0/F1 |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch64 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| ASIMD bit reverse | RBIT | 3 | 2 | F0/F1 |  |
| ASIMD bitwise insert, D-­‐form | BIF, BIT, BSL | 3 | 2 | F0/F1 |  |
| ASIMD bitwise insert, Q-­‐form | BIF, BIT, BSL | 3 | 1 | F0/F1 |  |
| ASIMD count, D-­‐form | CLS, CLZ, CNT | 3 | 2 | F0/F1 |  |
| ASIMD count, Q-­‐form | CLS, CLZ, CNT | 3 | 1 | F0/F1 |  |
| ASIMD duplicate, gen reg | DUP | 8 | 1 | L, F0/F1 |  |
| ASIMD duplicate, element | DUP | 3 | 2 | F0/F1 |  |
| ASIMD extract | EXT | 3 | 2 | F0/F1 |  |
| ASIMD extract narrow | XTN | 3 | 2 | F0/F1 |  |
| ASIMD extract narrow, saturating | SQXTN(2), SQXTUN(2),  UQXTN(2) | 4 | 1 | F1 |  |
| ASIMD insert, element to  element | INS | 3 | 2 | F0/F1 |  |
| ASIMD move, integer immed | MOVI | 3 | 2 | F0/F1 |  |
| ASIMD move, FP immed | FMOV | 3 | 2 | F0/F1 |  |
| ASIMD reciprocal estimate, D-­‐ form | FRECPE, FRECPX, FRSQRTE, URECPE,  URSQRTE | 5 | 2 | F0/F1 |  |
| ASIMD reciprocal estimate, Q-­‐ form | FRECPE, FRECPX, FRSQRTE, URECPE,  URSQRTE | 5 | 1 | F0/F1 |  |
| ASIMD reciprocal step, D-­‐form,  FZ | FRECPS, FRSQRTS | 9 | 2 | F0/F1 |  |
| ASIMD reciprocal step, D-­‐form,  no FZ | FRECPS, FRSQRTS | 10 | 2 | F0/F1 |  |
| ASIMD reciprocal step, Q-­‐form,  FZ | FRECPS, FRSQRTS | 9 | 1 | F0/F1 |  |
| ASIMD reciprocal step, Q-­‐form,  no FZ | FRECPS, FRSQRTS | 10 | 1 | F0/F1 |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch64 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| ASIMD reverse | REV16, REV32, REV64 | 3 | 2 | F0/F1 |  |
| ASIMD table lookup, D-­‐form | TBL, TBX | 3xN |  | F0/F1 | 1 |
| ASIMD table lookup, Q-­‐form | TBL, TBX | 3xN + 3 |  | F0/F1 | 1 |
| ASIMD transfer, element to gen  reg, word or dword | UMOV | 5 | 1 | L |  |
| ASIMD transfer, element to gen  reg | SMOV, UMOV | 6 | 1 | L, I0/I1 |  |
| ASIMD transfer, gen reg to  element | INS | 8 | 1 | L, F0/F1 |  |
| ASIMD transpose | TRN1, TRN2 | 3 | 2 | F0/F1 |  |
| ASIMD unzip/zip | UZP1, UZP2, ZIP1, ZIP2 | 3 | 2 | F0/F1 |  |

NOTE 1 – For table branches (TBL and TBX), N denotes the number of registers in the table.

* 1. **ASIMD Load Instructions**

The latencies shown assume the memory access hits in the Level 1 Data Cache. Compared to standard loads, an extra cycle is required to forward results to FP/ASIMD pipelines.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch32 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| ASIMD load, 1 element, multiple, 1 reg | VLD1 | 5 | 1 | L |  |
| ASIMD load, 1 element, multiple, 2 reg | VLD1 | 5 | 1 | L |  |
| ASIMD load, 1 element, multiple, 3 reg | VLD1 | 6 | 1/2 | L |  |
| ASIMD load, 1 element, multiple, 4 reg | VLD1 | 6 | 1/2 | L |  |
| ASIMD load, 1 element, one lane | VLD1 | 8 | 1 | L, F0/F1 |  |
| ASIMD load, 1 element, all lanes | VLD1 | 8 | 1 | L, F0/F1 |  |
| ASIMD load, 2 element, multiple, 2 reg | VLD2 | 8 | 1 | L, F0/F1 |  |
| ASIMD load, 2 element, multiple, 4 reg | VLD2 | 9 | 1/2 | L, F0/F1 |  |
| ASIMD load, 2 element, one lane, size 32 | VLD2 | 8 | 1 | L, F0/F1 |  |
| ASIMD load, 2 element, one lane, size  8/16 | VLD2 | 8 | 1 | L, F0/F1 |  |
| ASIMD load, 2 element, all lanes | VLD2 | 8 | 1 | L, F0/F1 |  |
| ASIMD load, 3 element, multiple, 3 reg | VLD3 | 9 | 1/2 | L, F0/F1 |  |
| ASIMD load, 3 element, one lane, size 32 | VLD3 | 8 | 1 | L, F0/F1 |  |
| ASIMD load, 3 element, one lane, size  8/16 | VLD3 | 9 | 2/3 | L, F0/F1 |  |
| ASIMD load, 3 element, all lanes | VLD3 | 8 | 1 | L, F0/F1 |  |
| ASIMD load, 4 element, multiple, 4 reg | VLD4 | 9 | 1/2 | L, F0/F1 |  |
| ASIMD load, 4 element, one lane, size 32 | VLD4 | 8 | 1 | L, F0/F1 |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch32 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| ASIMD load, 4 element, one lane, size  8/16 | VLD4 | 9 | 1/2 | L, F0/F1 |  |
| ASIMD load, 4 element, all lanes | VLD4 | 8 | 1 | L, F0/F1 |  |
| (ASIMD load, writeback form) |  | (1) | Same as  before | +I0/I1 | 1 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch64 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| ASIMD load, 1 element, multiple, 1 reg, D-­‐form | LD1 | 5 | 1 | L |  |
| ASIMD load, 1 element, multiple, 1 reg, Q-­‐form | LD1 | 5 | 1 | L |  |
| ASIMD load, 1 element, multiple, 2 reg, D-­‐form | LD1 | 5 | 1 | L |  |
| ASIMD load, 1 element, multiple, 2 reg, Q-­‐form | LD1 | 6 | 1/2 | L |  |
| ASIMD load, 1 element, multiple, 3 reg, D-­‐form | LD1 | 6 | 1/2 | L |  |
| ASIMD load, 1 element, multiple, 3 reg, Q-­‐form | LD1 | 7 | 1/3 | L |  |
| ASIMD load, 1 element, multiple, 4 reg, D-­‐form | LD1 | 6 | 1/2 | L |  |
| ASIMD load, 1 element, multiple, 4 reg, Q-­‐form | LD1 | 8 | 1/4 | L |  |
| ASIMD load, 1 element, one lane, B/H/S | LD1 | 8 | 1 | L, F0/F1 |  |
| ASIMD load, 1 element, one lane, D | LD1 | 5 | 1 | L |  |
| ASIMD load, 1 element, all lanes, D-­‐form, B/H/S | LD1R | 8 | 1 | L, F0/F1 |  |
| ASIMD load, 1 element, all lanes, D-­‐form, D | LD1R | 5 | 1 | L |  |
| ASIMD load, 1 element, all lanes, Q-­‐form | LD1R | 8 | 1 | L, F0/F1 |  |
| ASIMD load, 2 element, multiple, D-­‐form, B/H/S | LD2 | 8 | 1 | L, F0/F1 |  |
| ASIMD load, 2 element, multiple, Q-­‐form,  B/H/S | LD2 | 9 | 1/2 | L, F0/F1 |  |
| ASIMD load, 2 element, multiple, Q-­‐form, D | LD2 | 6 | 1/2 | L |  |
| ASIMD load, 2 element, one lane, B/H | LD2 | 8 | 1 | L, F0/F1 |  |
| ASIMD load, 2 element, one lane, S | LD2 | 8 | 1 | L, F0/F1 |  |
| ASIMD load, 2 element, one lane, D | LD2 | 6 | 1 | L |  |
| ASIMD load, 2 element, all lanes, D-­‐form, B/H/S | LD2R | 8 | 1 | L, F0/F1 |  |
| ASIMD load, 2 element, all lanes, D-­‐form, D | LD2R | 5 | 1 | L |  |
| ASIMD load, 2 element, all lanes, Q-­‐form | LD2R | 8 | 1 | L, F0/F1 |  |
| ASIMD load, 3 element, multiple, D-­‐form, B/H/S | LD3 | 9 | 1/2 | L, F0/F1 |  |
| ASIMD load, 3 element, multiple, Q-­‐form,  B/H/S | LD3 | 10 | 1/3 | L, F0/F1 |  |
| ASIMD load, 3 element, multiple, Q-­‐form, D | LD3 | 8 | 1/4 | L |  |
| ASIMD load, 3 element, one lane, B/H | LD3 | 9 | 2/3 | L, F0/F1 |  |
| ASIMD load, 3 element, one lane, S | LD3 | 8 | 1 | L, F0/F1 |  |
| ASIMD load, 3 element, one lane, D | LD3 | 6 | 1/2 | L |  |
| ASIMD load, 3 element, all lanes, D-­‐form, B/H/S | LD3R | 8 | 1 | L, F0/F1 |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch64 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| ASIMD load, 3 element, all lanes, D-­‐form, D | LD3R | 6 | 1/2 | L |  |
| ASIMD load, 3 element, all lanes, Q-­‐form, B/H/S | LD3R | 9 | 2/3 | L, F0/F1 |  |
| ASIMD load, 3 element, all lanes, Q-­‐form, D | LD3R | 9 | 1/2 | L, F0/F1 |  |
| ASIMD load, 4 element, multiple, D-­‐form, B/H/S | LD4 | 9 | 1/2 | L, F0/F1 |  |
| ASIMD load, 4 element, multiple, Q-­‐form,  B/H/S | LD4 | 11 | 1/4 | L, F0/F1 |  |
| ASIMD load, 4 element, multiple, Q-­‐form, D | LD4 | 8 | 1/4 | L |  |
| ASIMD load, 4 element, one lane, B/H | LD4 | 9 | 1/2 | L, F0/F1 |  |
| ASIMD load, 4 element, one lane, S | LD4 | 8 | 1 | L, F0/F1 |  |
| ASIMD load, 4 element, one lane, D | LD4 | 6 | 1/2 | L |  |
| ASIMD load, 4 element, all lanes, D-­‐form, B/H/S | LD4R | 8 | 1 | L, F0/F1 |  |
| ASIMD load, 4 element, all lanes, D-­‐form, D | LD4R | 6 | 1 | L |  |
| ASIMD load, 4 element, all lanes, Q-­‐form, B/H/S | LD4R | 9 | 1/2 | L, F0/F1 |  |
| ASIMD load, 4 element, all lanes, Q-­‐form, D | LD4R | 9 | 2/5 | L, F0/F1 |  |
| (ASIMD load, writeback form) |  | (1) | Same as  before | +I0/I1 | 1 |

NOTE 1 – Writeback forms of load instructions require an extra µop to update the base address. This update is typically performed in parallel with the load µop (update latency shown in parentheses).

* 1. **ASIMD Store Instructions**

Stores µops may issue once their address operands are available and do not need to wait for data operands. Once executed, stores are buffered and committed in the background.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch32 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| ASIMD store, 1 element, multiple, 1 reg | VST1 | 1 | 1 | S |  |
| ASIMD store, 1 element, multiple, 2 reg | VST1 | 2 | 1/2 | S |  |
| ASIMD store, 1 element, multiple, 3 reg | VST1 | 3 | 1/3 | S |  |
| ASIMD store, 1 element, multiple, 4 reg | VST1 | 4 | 1/4 | S |  |
| ASIMD store, 1 element, one lane | VST1 | 3 | 1 | F0/F1, S |  |
| ASIMD store, 2 element, multiple, 2 reg | VST2 | 3 | 1/2 | F0/F1, S |  |
| ASIMD store, 2 element, multiple, 4 reg | VST2 | 4 | 1/4 | F0/F1, S |  |
| ASIMD store, 2 element, one lane | VST2 | 3 | 1 | F0/F1, S |  |
| ASIMD store, 3 element, multiple, 3 reg | VST3 | 3 | 1/3 | F0/F1, S |  |
| ASIMD store, 3 element, one lane, size 32 | VST3 | 3 | 1/2 | F0/F1, S |  |
| ASIMD store, 3 element, one lane, size 8/16 | VST3 | 3 | 1 | F0/F1, S |  |
| ASIMD store, 4 element, multiple, 4 reg | VST4 | 4 | 1/4 | F0/F1, S |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch32 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| ASIMD store, 4 element, one lane, size 32 | VST4 | 3 | 1/2 | F0/F1, S |  |
| ASIMD store, 4 element, one lane, size 8/16 | VST4 | 3 | 1 | F0/F1, S |  |
| (ASIMD store, writeback form) |  |  | +1 | +I0/I1 | 1 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch64 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| ASIMD store, 1 element, multiple, 1 reg, D-­‐form | ST1 | 1 | 1 | S |  |
| ASIMD store, 1 element, multiple, 1 reg, Q-­‐form | ST1 | 2 | 1/2 | S |  |
| ASIMD store, 1 element, multiple, 2 reg, D-­‐form | ST1 | 2 | 1/2 | S |  |
| ASIMD store, 1 element, multiple, 2 reg, Q-­‐form | ST1 | 4 | 1/4 | S |  |
| ASIMD store, 1 element, multiple, 3 reg, D-­‐form | ST1 | 3 | 1/3 | S |  |
| ASIMD store, 1 element, multiple, 3 reg, Q-­‐form | ST1 | 6 | 1/6 | S |  |
| ASIMD store, 1 element, multiple, 4 reg, D-­‐form | ST1 | 4 | 1/4 | S |  |
| ASIMD store, 1 element, multiple, 4 reg, Q-­‐form | ST1 | 8 | 1/8 | S |  |
| ASIMD store, 1 element, one lane, B/H/S | ST1 | 3 | 1 | F0/F1, S |  |
| ASIMD store, 1 element, one lane, D | ST1 | 1 | 1 | S |  |
| ASIMD store, 2 element, multiple, D-­‐form, B/H/S | ST2 | 3 | 1/2 | F0/F1, S |  |
| ASIMD store, 2 element, multiple, Q-­‐form, B/H/S | ST2 | 4 | 1/4 | F0/F1, S |  |
| ASIMD store, 2 element, multiple, Q-­‐form, D | ST2 | 4 | 1/4 | S |  |
| ASIMD store, 2 element, one lane, B/H/S | ST2 | 3 | 1 | F0/F1, S |  |
| ASIMD store, 2 element, one lane, D | ST2 | 2 | 1/2 | S |  |
| ASIMD store, 3 element, multiple, D-­‐form, B/H/S | ST3 | 3 | 1/3 | F0/F1, S |  |
| ASIMD store, 3 element, multiple, Q-­‐form, B/H/S | ST3 | 6 | 1/6 | F0/F1, S |  |
| ASIMD store, 3 element, multiple, Q-­‐form, D | ST3 | 6 | 1/6 | S |  |
| ASIMD store, 3 element, one lane, B/H | ST3 | 3 | 1 | F0/F1, S |  |
| ASIMD store, 3 element, one lane, S | ST3 | 3 | 1/2 | F0/F1, S |  |
| ASIMD store, 3 element, one lane, D | ST3 | 3 | 1/3 | S |  |
| ASIMD store, 4 element, multiple, D-­‐form, B/H/S | ST4 | 4 | 1/4 | F0/F1, S |  |
| ASIMD store, 4 element, multiple, Q-­‐form, B/H/S | ST4 | 8 | 1/8 | F0/F1, S |  |
| ASIMD store, 4 element, multiple, Q-­‐form, D | ST4 | 8 | 1/8 | S |  |
| ASIMD store, 4 element, one lane, B/H | ST4 | 3 | 1 | F0/F1, S |  |
| ASIMD store, 4 element, one lane, S | ST4 | 3 | 1/2 | F0/F1, S |  |
| ASIMD store, 4 element, one lane, D | ST4 | 4 | 1/4 | S |  |
| (ASIMD store, writeback form) |  | (1) | Same as  before | +I0/I1 | 1 |

NOTE 1 – Writeback forms of store instructions require an extra µop to update the base address. This update is typically performed in parallel with the store µop (update latency shown in parentheses).

* 1. **Cryptography Extensions**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch32 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| Crypto AES ops | AESD, AESE, AESIMC, AESMC | 3 | 1 | F0 | 1 |
| Crypto polynomial (64x64)  multiply long | VMULL.P64 | 3 | 1 | F0 |  |
| Crypto SHA1 xor ops | SHA1SU0 | 6 | 2 | F0/F1 |  |
| Crypto SHA1 fast ops | SHA1H, SHA1SU1 | 3 | 1 | F0 |  |
| Crypto SHA1 slow ops | SHA1C, SHA1M, SHA1P | 6 | 1/2 | F0 |  |
| Crypto SHA256 fast ops | SHA256SU0 | 3 | 1 | F0 |  |
| Crypto SHA256 slow ops | SHA256H, SHA256H2,  SHA256SU1 | 6 | 1/2 | F0 |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch64 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| Crypto AES ops | AESD, AESE, AESIMC, AESMC | 3 | 1 | F0 | 1 |
| Crypto AES ops | AESE/AESMC pair,  AESD/AESIMC pair | 3 | 1 | F0 | 2 |
| Crypto polynomial (64x64)  multiply long | PMULL(2) | 3 | 1 | F0 |  |
| Crypto SHA1 xor ops | SHA1SU0 | 6 | 2 | F0/F1 |  |
| Crypto SHA1 schedule acceleration ops | SHA1H, SHA1SU1 | 3 | 1 | F0 |  |
| Crypto SHA1 hash acceleration ops | SHA1C, SHA1M, SHA1P | 6 | 1/2 | F0 |  |
| Crypto SHA256 schedule  acceleration op (1 µop ) | SHA256SU0 | 3 | 1 | F0 |  |
| Crypto SHA256 schedule  acceleration op (2 µops) | SHA256SU1 | 6 | 1/2 | F0 |  |
| Crypto SHA256 hash  acceleration ops | SHA256H, SHA256H2 | 6 | 1/2 | F0 |  |

NOTE 1 – In Cortex-A57 r0p0, each AESE/AESMC/AESD/AESIMC will exhibit the described performance characteristics.

NOTE 2 – In Cortex-A57 r0p1 and later revisions, adjacent AESE/AESMC instruction pairs and adjacent AESD/AESIMC instruction pairs will exhibit the described performance characteristics. See Section 4.13 for additional details.

* 1. **CRC**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch32 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| CRC checksum ops | CRC32, CRC32C | 3 | 1 | M |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction Group** | **AArch64 Instructions** | **Exec Latency** | **Execution Throughput** | **Utilized Pipelines** | **Notes** |
| CRC checksum ops | CRC32, CRC32C | 3 | 1 | M |  |

# SPECIAL CONSIDERATIONS

* 1. **Dispatch Constraints**

Dispatch of µops from the in-order portion to the out-of-order portion of the microarchitecture includes a number of constraints. It is important to consider these constraints during code generation in order to maximize the effective dispatch bandwidth and subsequent execution bandwidth of the Cortex-A57.

The dispatch stage can process up to 3 µops per cycle, with the following limitations on the number of µops of each type that may be simultaneously dispatched.

* 1 µop utilizing B pipeline
* Up to 2 µops utilizing I pipelines
* Up to 2 µops utilizing M pipeline
* 1 µop utilizing F0 pipeline
* 1 µop utilizing F1 pipeline
* Up to 2 µops utilizing L or S pipeline

In the event there are more µops available to be dispatched in a given cycle than can be supported by the constraints above, µops will be dispatched in oldest to youngest age-order to the extent allowed by the above.

* 1. **Conditional Execution**

The ARMv8 architecture allows many types of A32 instructions to be conditionally executed based upon condition flags (N, Z, C, V). If the condition flags satisfy a condition specified in the instruction encoding, an instruction has its normal effect. If the flags do not satisfy this condition, the instruction acts as a NOP.

This leads to conditional register writes for most types of conditional instructions. In an out-of-order processor such as Cortex-A57, this has two side-effects.

The first side-effect is that the conditional instruction requires the old value of its destination register as an input operand. The second side-effect is that all subsequent consumers of the destination register of the conditional instruction are dependent upon this operation, regardless of the state of the condition flags (i.e. even if the destination register is unchanged in the event the condition is not met.).

These effects should be taken into account when considering using conditional execution for long-latency operations. The overheads of conditional execution may begin to outweigh the benefits. Consider the following example.

MULEQ R1, R2, R3 MULNE R1, R2, R4

For this pair of instructions, the second multiply is dependent upon the result of the first multiply, not through one of its normal input operands (R2 and R4), but through the destination register R1. The combined latency for these instructions is six cycles, rather than the four cycles that would be required if these instructions were not conditional (3 cycles latency for the first, one additional cycle for the second which is fully pipelined behind the first). So if the condition is easily predictable (by the branch predictor), conditional execution can lead to a performance loss. But if the condition is not easily predictable, conditional execution may lead to a performance gain since the latency of a branch mispredict is generally higher than the execution latency of conditional

instructions. In general, it is recommended that conditional instruction forms be considered only for integer instructions with latency less than or equal to 2 cycles, loads, and stores.

* 1. **Conditional ASIMD**

Conditional execution is architecturally possible for ASIMD instructions in Thumb state using IT blocks. However, this type of encoding is considered abnormal and is not recommended for Cortex-A57. It is expected to perform worse than the equivalent unconditional encodings.

* 1. **Register Forwarding Hazards**

The ARMv8-A architecture allows FP instructions to read and write 32-bit S-registers. In AArch32, Each S-register corresponds to one half (upper or lower) of an overlayed 64-bit D-register. Register forwarding hazards may occur when one µop reads a D-register or Q-register operand that has recently been written with one or more S-register result. Consider the following abnormal scenario.

VMOV S0,R0 VMOV S1,R1 VADD D2, D1, D0

The first two instructions write S0 and S1, which correspond to the bottom and top halves of D0. The third instruction then requires D0 as an input operand. In this scenario, Cortex-A57 detects that at least one of the upper or lower S0/S1 registers overlayed on D0 were previously written, at which point the VADD instruction is serialized until the prior S-register writes are guaranteed to have been architecturally committed, likely incurring significant additional latency. Note that once the D0 register has been written as a D-register or Q-register destination, subsequent consumers of that register will no longer encounter this register-hazard condition, until the next S-register write, if any.

Cortex-A57 is able to avoid this register-hazard condition for certain cases. The following rules describe the conditions under which a register-hazard can occur.

* The producer writes an S-register (not a D[x] scalar)
* The consumer reads an overlapping D-register (not as a D[x] scalar, nor as an implicit operand due to conditional execution)
* The consumer is a FP/ASIMD µop (not a store µop )

To avoid unnecessary hazards, it is recommended that the programmer use D[x] scalar writes when populating registers prior to ASIMD operations. For example, either of the following instruction forms would safely prevent a subsequent hazard.

VLD1.32 Dd[x], [address]

VMOV.32 Dd[x], Rt

The Performance Monitor Unit (PMU) in Cortex-A57 may be used to determine when register forwarding hazards are actually occuring. The implementation defined PMU event number 0x12C (DISP\_SWDW\_STALL) has been assigned to count the number of cycles spent stalling due to these hazards.

* 1. **Load/Store Throughput**

The Cortex-A57 processor includes separate load and store pipelines, which allow it to execute one load µop and one store µop every cycle. .

To achieve maximum throughput for memory copy (or similar loops), one should do the following.

* Unroll the loop to include multiple load and store operations per iteration, minimizing the overheads of looping.
* Use discrete, non-writeback forms of load and store instructions (such as LDRD and STRD), interleaving them so that one load and one store operation may be performed each cycle. Avoid load-/store-multiple instruction encodings (such as LDM and STM), which lead to separated bursts of load and store µops which may not allow concurrent utilization of both the load and store pipelines.

The following example shows a recommended instruction sequence for a long memory copy in AArch32 state:

Loop\_start:

SUBS r2,r2,#64 LDRD r3,r4,[r1,#0] STRD r3,r4,[r0,#0] LDRD r3,r4,[r1,#8] STRD r3,r4,[r0,#8]

LDRD r3,r4,[r1,#16] STRD r3,r4,[r0,#16] LDRD r3,r4,[r1,#24] STRD r3,r4,[r0,#24] LDRD r3,r4,[r1,#32] STRD r3,r4,[r0,#32] LDRD r3,r4,[r1,#40] STRD r3,r4,[r0,#40] LDRD r3,r4,[r1,#48] STRD r3,r4,[r0,#48] LDRD r3,r4,[r1,#56] STRD r3,r4,[r0,#56] ADD r1,r1,#64

ADD r0,r0,#64

BGT Loop\_start

A recommended copy routine for AArch64 would look similar to the sequence above, but would use LDP/STP instructions.

* 1. **Load/Store Alignment**

The ARMv8-A architecture allows many types of load and store accesses to be arbitrarily aligned. The Cortex- A57 processor handles most unaligned accesses without performance penalties. However, there are cases which reduce bandwidth or incur additional latency, as described below.

* Load operations that cross a cache-line (64-byte) boundary
* Store operations that cross a 16-byte boundary
  1. **Non-Temporal Loads/Stores**

The ARM v8-A architecture provides load/store non-temporal pair instructions (LDNP/STNP) that provide a hint to the memory system that an access is non-temporal or streaming, and unlikely to be repeated in the near future.

Versions of the Cortex-A57 processor prior to r1p3 do not prefetch or cache non-temporal data and hence, there could be a performance degradation when using non-temporal loads/stores instead of normal loads/stores.

Ignoring the non-temporal hint will avoid this degradation. To this effect, it is recommended that bit 52 of the CPU auxiliary control register (CPUACTLR\_EL1[52]) be always enabled.

* 1. **Branch Alignment**

Branch instruction and branch target instruction alignment can affect performance. For best-case performance, consider the following guidelines:

* Try not to include more than two taken branches within the same quadword-aligned quadword of instruction memory.
* Consider aligning subroutine entry points and branch targets to quadword boundaries, within the bounds of the code-density requirements of the program. This will ensure that the subsequent fetch can retrieve four (or a full quadword’s worth of) instructions, maximizing fetch bandwidth following the taken branch.
  1. **Setting Condition Flags**

The ARM instruction set includes instruction forms that set the condition flags. In addition to compares, many types of data processing operations set the condition flags as a side-effect. Excessive use of flag-setting instruction forms may result in performance degradation, thus it is recommended that, where possible, non-flag- setting instructions and instruction-forms be used except where the condition-flag result is explicitly required for subsequent branches or conditional instructions.

When using the Thumb instruction set, special attention should be given to the use of 16-bit instruction forms. Many of those (moves, adds, shifts, etc) automatically set the condition flags. For best performance, consider using the 32-bit encodings which include forms that do not set the condition flags, within the bounds of the code- density requirements of the program.

* 1. **Accelerated Accumulator Forwarding in the Floating-PointPipelines**

As described in chapter 2 of this document, the Cortex-A57 processor implements two floating point execution pipelines (F0/F1). For versions of Cortex-A57 prior to r1p3, µops are steered to one pipeline or the other based upon a load-balancing hardware mechanism. For these versions of A57, forwarding from a multiply or multiply- accumulate into the accumulator operand of a subsequent multiply-accumulate can be accelerated if all instructions are contained within the same pipeline. Thus, it is recommended that when performing a critical sequence of dependent, non-quadword FP/ASIMD floating-point multiply or multiply-accumulate operations, the programmer should ensure the accumulator source and destination registers should both be even or odd in order to facilitate accelerated accumulator forwarding.

For Cortex-A57 r1p3 and later revisions, this constrained register usage model is not required.

* 1. **Load Balancing in the Floating-PointPipelines**

As described in chapter 2 of this document, the Cortex-A57 processor implements two floating point execution pipelines (F0/F1). For versions of Cortex-A57 prior to r1p3, FP/ASIMD floating-point multiply or multiply- accumulate µops are steered to one pipeline or the other based upon a load-balancing hardware mechanism. For these versions of A57, it is recommended that, subject to the recommendation described in 4.10 above, the programmer use a balanced mix of odd and even destination D-registers for FP/ASIMD floating-point multiply or multiply-accumulate instructions, to ensure those instructions are evenly distributed across the two floating-point pipelines.

For Cortex-A57 r1p3 and later revisions, this constrained register usage model is not required.

* 1. **Special Register Access**

The Cortex-A57 processor performs register renaming for general purpose registers to enable speculative and out-of-order instruction execution. But most special-purpose registers are not renamed. Instructions that read or write non-renamed registers are subjected to one or more of the following additional execution constraints.

* Non-Speculative Execution – Instructions may only execute non-speculatively.
* In-Order Execution – Instructions must execute in-order with respect to other similar instructions or in some cases all instructions.
* Flush Side-Effects – Instructions trigger a flush side-effect after executing for synchronization.

The table below summarizes various special instructions and the associated execution constraints or side-effects.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instructions** | **Forms** | **Non- Speculative** | **In- Order** | **Flush Side- Effect** | **Notes** |
| ISB |  | Yes | Yes | Yes | 1 |
| CPS |  | Yes | Yes | Yes | 1 |
| SETEND |  | Yes | Yes | Yes | 1 |
| MRS (read) | APSR, CPSR | Yes | Yes | No | 1 |
| MRS (read) | SPSR | No | Yes | No | 1 |
| MSR (write) | ASPR\_nzcvq, CPSR\_f | No | No | No | 1, 2, 3 |
| MSR (write) | APSR, CPSR other | Yes | Yes | Yes | 1 |
| MSR (write) | SPSR | Yes | Yes | No | 1 |
| VMRS (read) | FPSCR to APSR\_nzcv | No | No | No | 1, 2 |
| VMRS (read) | Other | Yes | Yes | No | 1 |
| VMSR (write) | FPSCR, changing only NZCV | Yes | Yes | No | 1 |
| VMSR (write) | Other | Yes | Yes | Yes | 1 |
| MRC (read) |  | Some | Yes | No | 1, 2, 4 |
| MCR (write) |  | Yes | Yes | Some | 1, 4 |

NOTE 1 – Conditional forms of these instructions for which the condition is not satisfied will not access special registers or trigger flush side-effects.

NOTE 2 – Conditional forms of these instructions are always executed non-speculatively and in-order to properly resolve the condition.

NOTE 3 – MSR instructions that write APSR\_nzcvq generate a separate µop to write the Q bit. That µop executes non-speculatively and in-order. But the main µop , which writes the NZCV bits, executes as shown in the table above.

NOTE 4 – A subset of MCR instructions must be executed non-speculatively. A subset of MRC instructions trigger flush side-effects for synchronization. Those subsets are not documented here.

* 1. **AES Encryption/Decryption**

Cortex-A57 r0p0 can issue one AESE/AESMC/AESD/AESIMC instruction every cycle (fully pipelined) with an execution latency of three cycles (see Section 3.19). This means encryption or decryption for at least three data chunks should be interleaved for maximum performance:

AESE data0, key0 AESMC data0, data0 AESE data1, key0 AESMC data1, data1 AESE data2, key0 AESMC data2, data2 AESE data0, key1 AESMC data0, data0

...

In Cortex-A57 r0p1 and later revisions, pairs of dependent AESE/AESMC and AESD/AESIMC instructions are higher performance when adjacent, and in in the described order, in the program code. Therefore it is important to ensure that these instructions come in pairs in AES encryption/decryption loops, as shown in the code segment above.

* 1. **Fast literal generation**

Cortex-A57 r1p0 and later revisions support optimized literal generation for 32- and 64-bit code. A typical literal generation sequence in 32-bit code:

MOV rX, #bottom\_16\_bits MOVT rX, #top\_16\_bits

And in 64-bit code, generating a 32-bit immediate:

MOV wX, #bottom\_16\_bits

MOVK wX, #top\_16\_bits, lsl #16

In 64-bit code, generating the bottom half of a 64-bit immediate:

MOV xX, #bottom\_16\_bits

MOVK xX, #top\_16\_bits, lsl #16

In 64-bit code, generating the top half of a 64-bit immediate:

MOVK xX, #bits\_47\_to\_32, lsl #32 MOVK xX, #bits\_63\_to\_48, lsl #48

If any of these sequences appear sequentially and in the described order in program code, the two instructions can be executed at lower latency and higher bandwidth than if they do not appear sequentially in the program code, enabling 32-bit literals to be generated in a single cycle and 64-bit literals to be generated in two cycles.

Thus it is advantageous to ensure that compilers or programmers writing assembly code schedule these instruction pairs sequentially.

* 1. **PC-relative address calculation**

Cortex-A57 r1p3 and later revisions support optimized PC-relative address calculation using the following instruction sequence:

ADRP xX, #label ADD xY, xX, #imm

If this sequence appears sequentially and in the described order in program code, the two instructions can be executed at lower latency and higher bandwidth than if they do not appear sequentially in the program code.

Thus it is advantageous to ensure that compilers or programmers writing assembly code schedule these instruction pairs sequentially.

* 1. **FPCR self-synchronization**

Programmers and compiler writers should note that writes to the FPCR register are self-synchronizing, i.e. its effect on subsequent instructions can be relied upon without an intervening context synchronizing operation.

1. **Detalhes da Arquitetura**
   1. **Tipo de Arquitetura**

A arquitetura ARMv8-A extende a arquitetura ARMv7, tendo agora dois estados de execuc¸a˜o: 32 bits e 64 bits, mantendo a compatibilidade com a arquitetura predeces- sora e, portanto, seus fundamentos. A arquitetura ARMv8-A segue uma pipeline, com a quantidade de esta´gios variando de acordo com o processador em que a arquitetura foi implementada. A ordem de execuc¸a˜o tambe´m varia de acordo com o processador, como exemplo:

* + - Cortex-A53: Pipeline executa em ordem.
    - Cortex-A57: Superescalar, na˜o segue uma ordem espec´ıfica (Speculative Issue).

## Arquitetura do Processador

Exemplificaremos a implementac¸a˜o da arquitetura ARMv8-A em um processador Cortex-A57, lanc¸ado em janeiro de 2015, com uma quantidade de cores que varia de 1 a` 4.

* + - Esta´gios por Pipeline: 15+.
    - Velocidade do Clock: 1.5 a` 2.5 GHz em 20nm.
    - Tamanho do Cache L1 (Instruc¸a˜o): 48 KB (associativo de 3 vias).
    - Tamanho do Cache L1 (Dados): 32 KB (associativo bidirecional).
    - Tamanho do Cache L2: 512 KB a` 2 MB (associativo de 16 vias).
    - Taxa de Tranfereˆncia ma´xima de Inteiro: 4.1 a` 4.76MIPS/MHz

# Refereˆncias

ARM (2011). Arm discloses technical details of the next version of the arm archi- tecture. <http://www.arm.com/about/newsroom/arm-discloses-technical-details-of-the-> next-version-of-the-arm-architecture.php. Acesso: 09 Junho 2019.

ARM (2013). *Arm*ⓍR *Architecture Reference Manual Armv8, for Armv8-A architecture*

*profile Documentation*.

Geekbench (2019). Oneplus 7 pro. https://browser.geekbench.com/v5/cpu/592464.

Acesso: 14 Novembro 2019.

Qualcomm (2018). Qualcomm announces new flagship snapdra- gon 855 mobile platform - a new decade of 5g, ai, and xr. [https://www.qualcomm.com/news/releases/2018/12/05/qualcomm-announces-ne](http://www.qualcomm.com/news/releases/2018/12/05/qualcomm-announces-new-)w- flagship-snapdragon-855-mobile-platform-new-decade. Acesso: 14 Novembro 2019.

Shah, A. (2016). Intel’s atom architecture lives on despite mobile chip cancella- tions. [https://www](http://www.pcworld.com/article/3064716/intels-atom-architecture-to-live-on-).pcw[orld.com/article/3064716/intels-atom-architecture-to-li](http://www.pcworld.com/article/3064716/intels-atom-architecture-to-live-on-)v[e-on-](http://www.pcworld.com/article/3064716/intels-atom-architecture-to-live-on-) despite-smartphone-chip-cancellations.html. Acesso: 16 Novembro 2019.

Stephens, N. (2019). Developments in the arm a-profile architecture: Armv8.6-

a. https://community.arm.com/developer/ip-products/processors/b/processors-ip- blog/posts/arm-architecture-developments-armv8-6-a. Acesso: 09 Novembro 2019.