CSUN | CALIFORNIA STATE UNIVERSITY NORTHRIDGE

COMP122

COMP 122

DR JEFF SOFTWARE
INDIE APP DEVELOPER
© Jeff Drobman
2016-2022

Rev 2-22-22

ASSEMBLY Programming/ISA

ARM

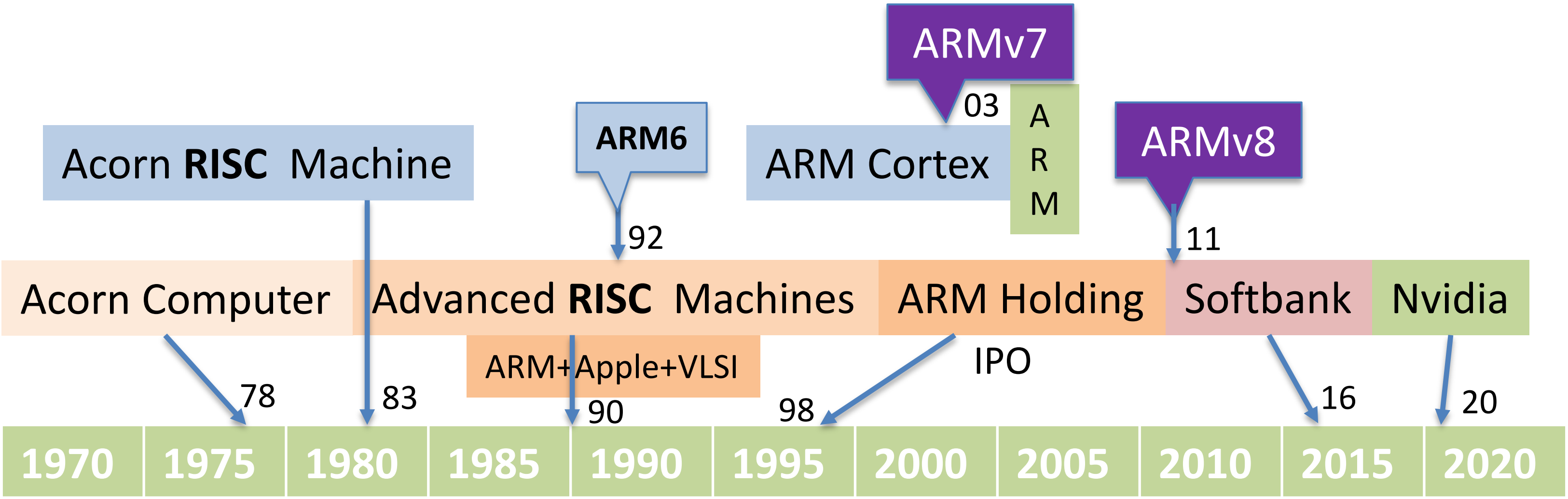
Dr Jeff Drobman

website → drjeffsoftware.com/classroom.html

email → jeffrey.drobman@csun.edu

ARM Index

ARM History



ARMv7

03

ARM

ARMv8

ARM6

Acorn **RISC** Machine

92

ARM Cortex

11

Acorn Computer | Advanced **RISC** Machines | ARM Holding | Softbank | Nvidia

ARM+Apple+VLSI

IPO

78

83

90

98

16

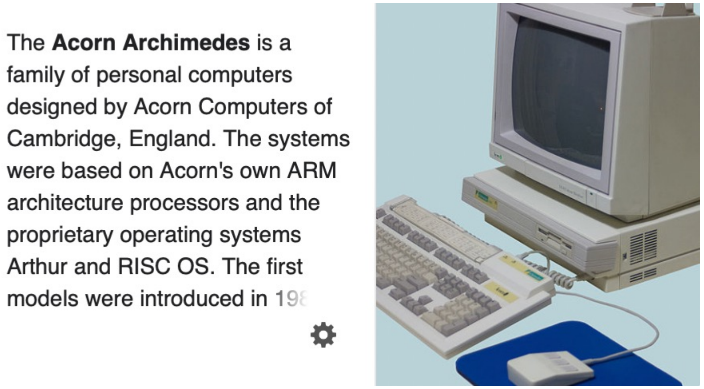
20

1970 | 1975 | 1980 | 1985 | 1990 | 1995 | 2000 | 2005 | 2010 | 2015 | 2020

Legend

CPU design

Company



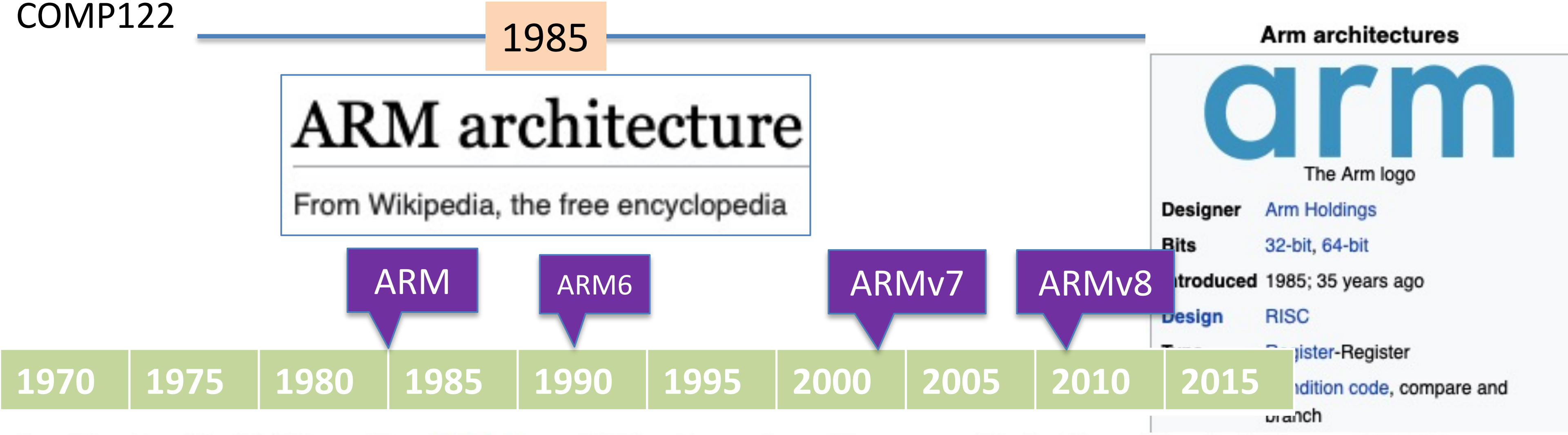
The **Acorn Archimedes** is a family of personal computers designed by Acorn Computers of Cambridge, England. The systems were based on Acorn's own ARM architecture processors and the proprietary operating systems Arthur and RISC OS. The first models were introduced in 19

CSUN
CALIFORNIA
STATE UNIVERSITY
NORTHRIDGE

COMP122

ARM

DR JEFF SOFTWARE
INDIE APP DEVELOPER
© Jeff Drobman
2016-2022



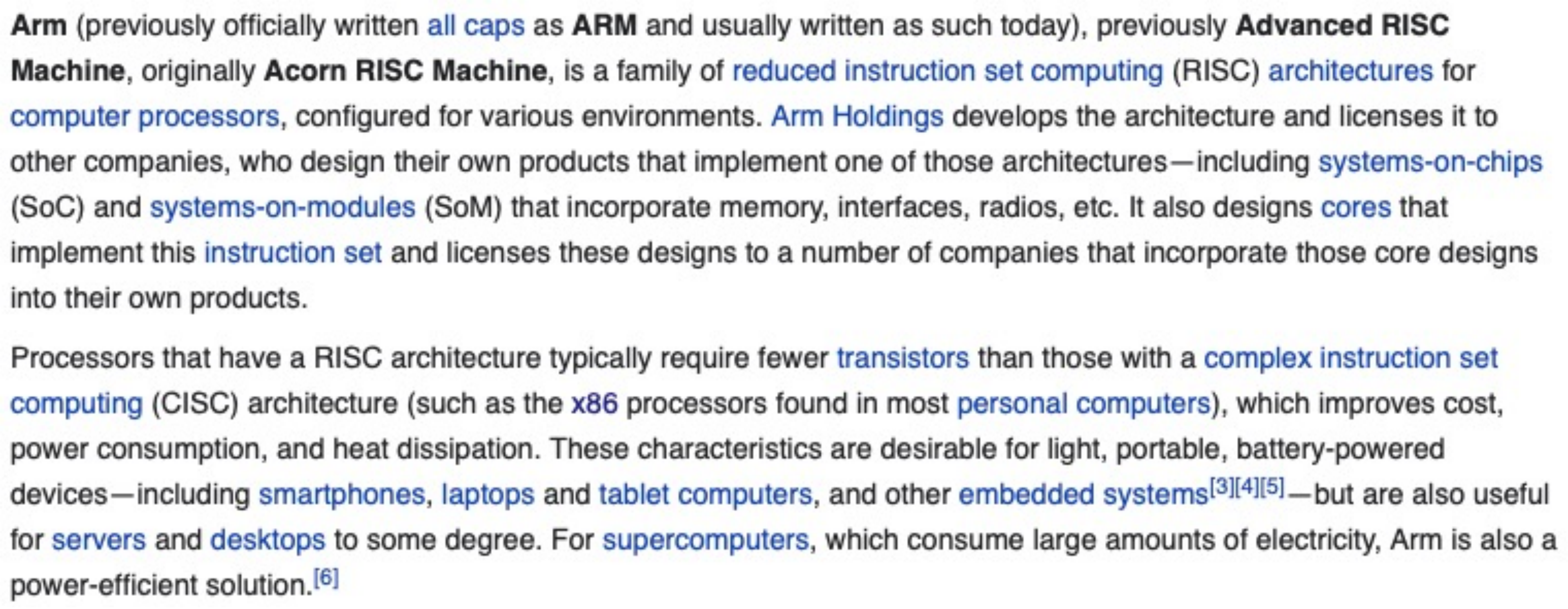
ARM architecture

From Wikipedia, the free encyclopedia

Arm architectures

The Arm logo

Designer	Arm Holdings
Bits	32-bit, 64-bit
troduced	1985; 35 years ago
esign	RISC
	gister-Register
	ndition code, compare and branch

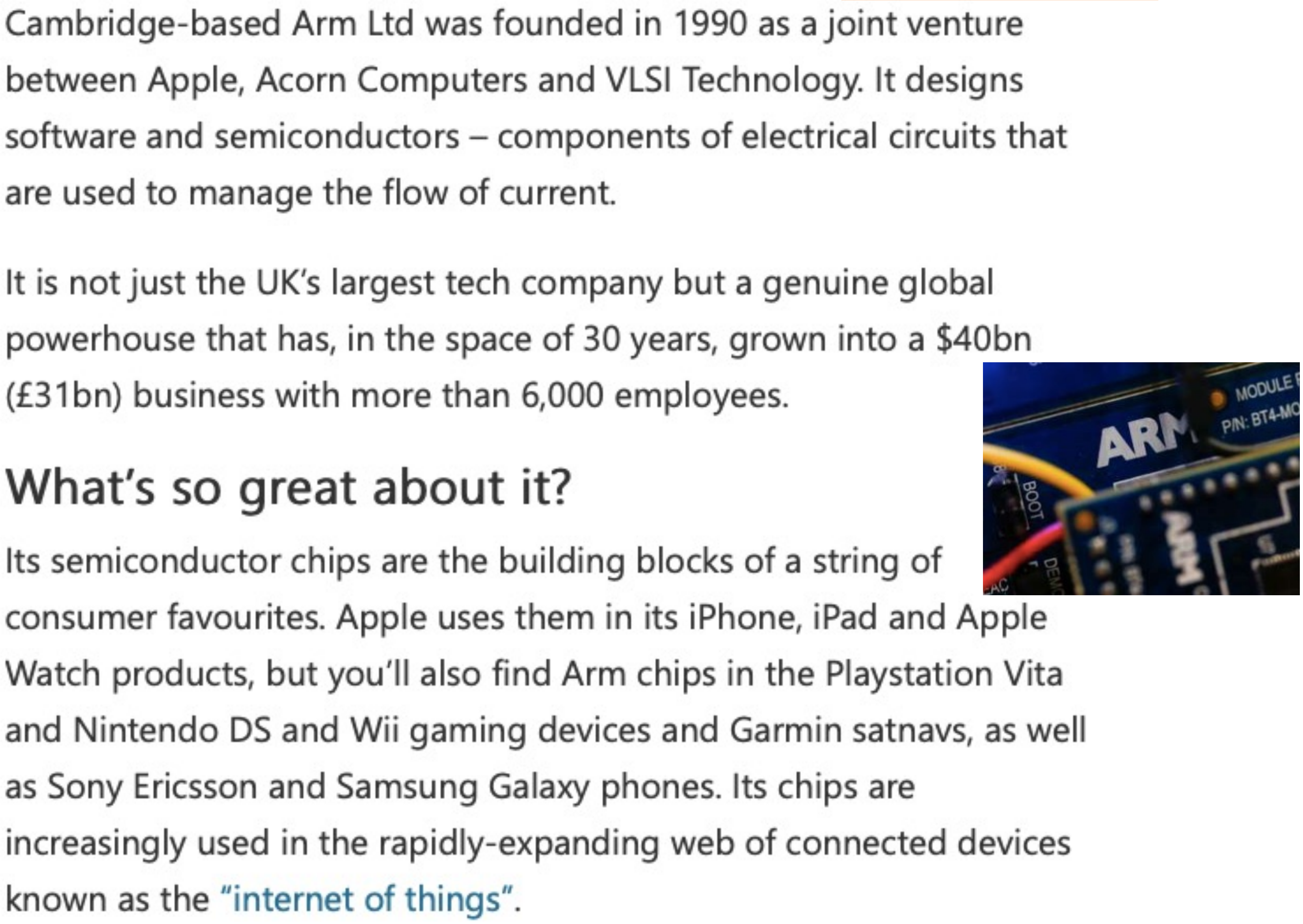


Arm (previously officially written all caps as **ARM** and usually written as such today), previously **Advanced RISC Machine**, originally **Acorn RISC Machine**, is a family of reduced instruction set computing (RISC) architectures for computer processors, configured for various environments. Arm Holdings develops the architecture and licenses it to other companies, who design their own products that implement one of those architectures—including systems-on-chips (SoC) and systems-on-modules (SoM) that incorporate memory, interfaces, radios, etc. It also designs cores that implement this instruction set and licenses these designs to a number of companies that incorporate those core designs into their own products.

Processors that have a RISC architecture typically require fewer transistors than those with a complex instruction set computing (CISC) architecture (such as the x86 processors found in most personal computers), which improves cost, power consumption, and heat dissipation. These characteristics are desirable for light, portable, battery-powered devices—including smartphones, laptops and tablet computers, and other embedded systems[3][4][5]—but are also useful for servers and desktops to some degree. For supercomputers, which consume large amounts of electricity, Arm is also a power-efficient solution.[6]

ARM History

1990 joint venture



Cambridge-based Arm Ltd was founded in 1990 as a joint venture between Apple, Acorn Computers and VLSI Technology. It designs software and semiconductors – components of electrical circuits that are used to manage the flow of current.

It is not just the UK's largest tech company but a genuine global powerhouse that has, in the space of 30 years, grown into a $40bn (£31bn) business with more than 6,000 employees.

What's so great about it?

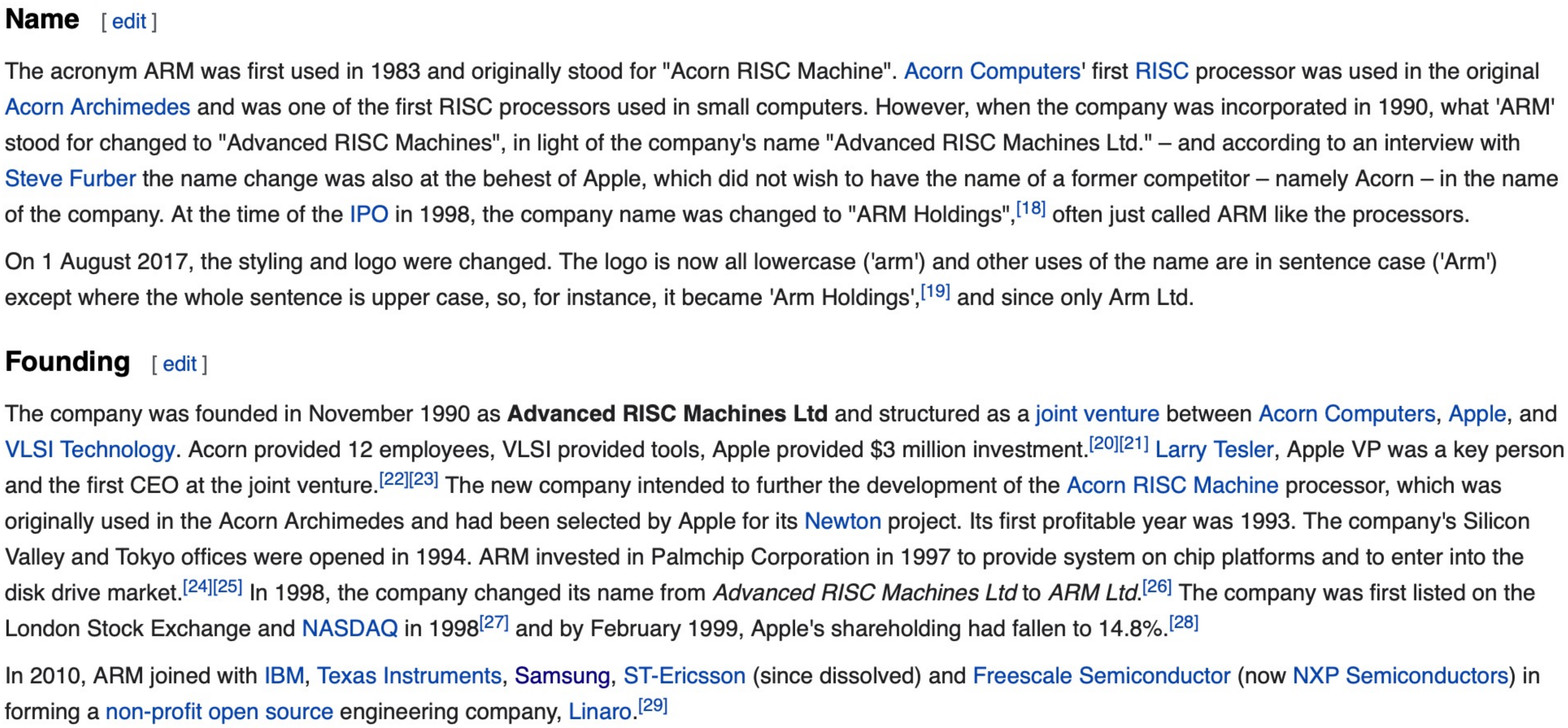
Its semiconductor chips are the building blocks of a string of consumer favourites. Apple uses them in its iPhone, iPad and Apple Watch products, but you'll also find Arm chips in the Playstation Vita and Nintendo DS and Wii gaming devices and Garmin satnavs, as well as Sony Ericsson and Samsung Galaxy phones. Its chips are increasingly used in the rapidly-expanding web of connected devices known as the "internet of things".

ARM History

Acorn RISC Machine

ARM architecture

From Wikipedia, the free encyclopedia



Name [edit]

The acronym ARM was first used in 1983 and originally stood for "Acorn RISC Machine". Acorn Computers' first RISC processor was used in the original Acorn Archimedes and was one of the first RISC processors used in small computers. However, when the company was incorporated in 1990, what 'ARM' stood for changed to "Advanced RISC Machines", in light of the company's name "Advanced RISC Machines Ltd." – and according to an interview with Steve Furber the name change was also at the behest of Apple, which did not wish to have the name of a former competitor – namely Acorn – in the name of the company. At the time of the IPO in 1998, the company name was changed to "ARM Holdings",[18] often just called ARM like the processors.

On 1 August 2017, the styling and logo were changed. The logo is now all lowercase ('arm') and other uses of the name are in sentence case ('Arm') except where the whole sentence is upper case, so, for instance, it became 'Arm Holdings',[19] and since only Arm Ltd.

Founding [edit]

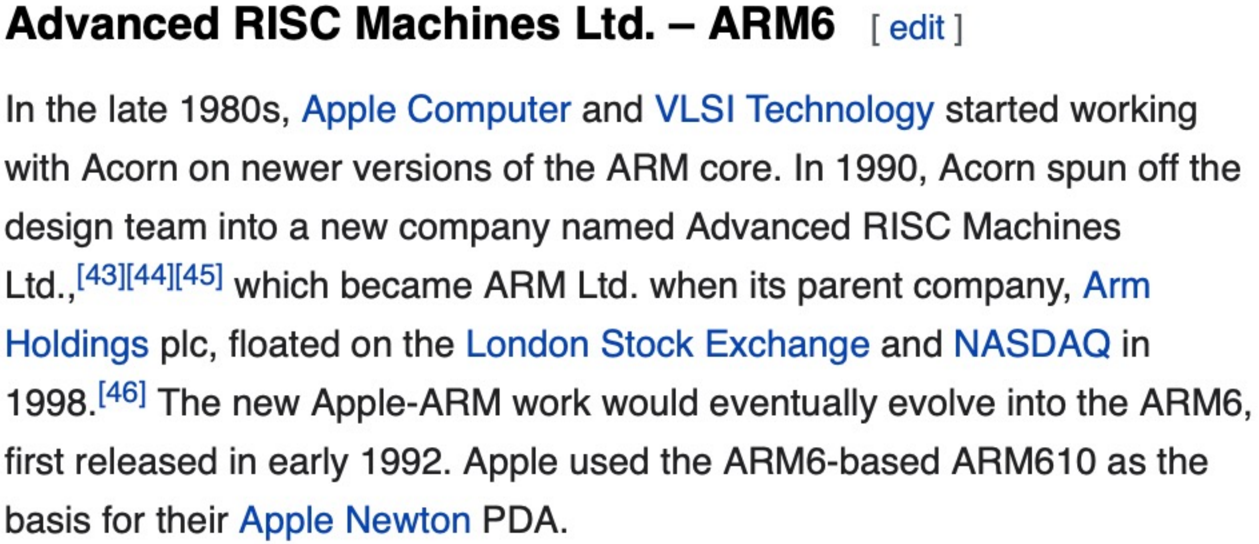
The company was founded in November 1990 as **Advanced RISC Machines Ltd** and structured as a joint venture between Acorn Computers, Apple, and VLSI Technology. Acorn provided 12 employees, VLSI provided tools, Apple provided $3 million investment.[20][21] Larry Tesler, Apple VP was a key person and the first CEO at the joint venture.[22][23] The new company intended to further the development of the Acorn RISC Machine processor, which was originally used in the Acorn Archimedes and had been selected by Apple for its Newton project. Its first profitable year was 1993. The company's Silicon Valley and Tokyo offices were opened in 1994. ARM invested in Palmchip Corporation in 1997 to provide system on chip platforms and to enter into the disk drive market.[24][25] In 1998, the company changed its name from *Advanced RISC Machines Ltd* to *ARM Ltd*.[26] The company was first listed on the London Stock Exchange and NASDAQ in 1998[27] and by February 1999, Apple's shareholding had fallen to 14.8%.[28]

In 2010, ARM joined with IBM, Texas Instruments, Samsung, ST-Ericsson (since dissolved) and Freescale Semiconductor (now NXP Semiconductors) in forming a non-profit open source engineering company, Linaro.[29]

ARM History

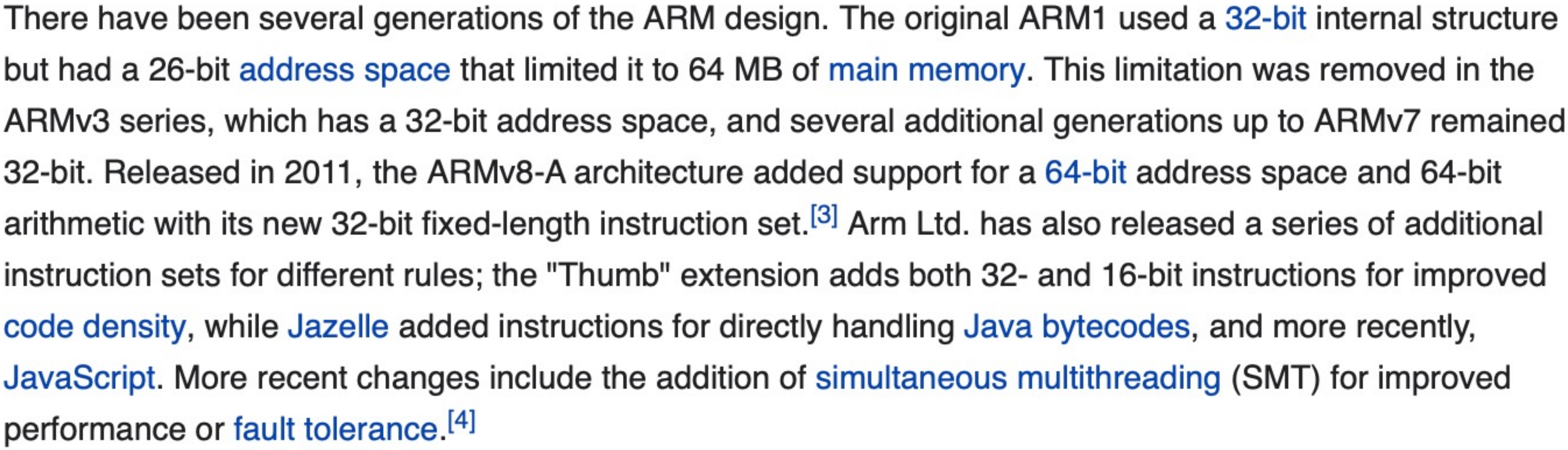
ARM architecture

From Wikipedia, the free encyclopedia



Advanced RISC Machines Ltd. – ARM6 [edit]

In the late 1980s, Apple Computer and VLSI Technology started working with Acorn on newer versions of the ARM core. In 1990, Acorn spun off the design team into a new company named Advanced RISC Machines Ltd.,[43][44][45] which became ARM Ltd. when its parent company, Arm Holdings plc, floated on the London Stock Exchange and NASDAQ in 1998.[46] The new Apple-ARM work would eventually evolve into the ARM6, first released in early 1992. Apple used the ARM6-based ARM610 as the basis for their Apple Newton PDA.

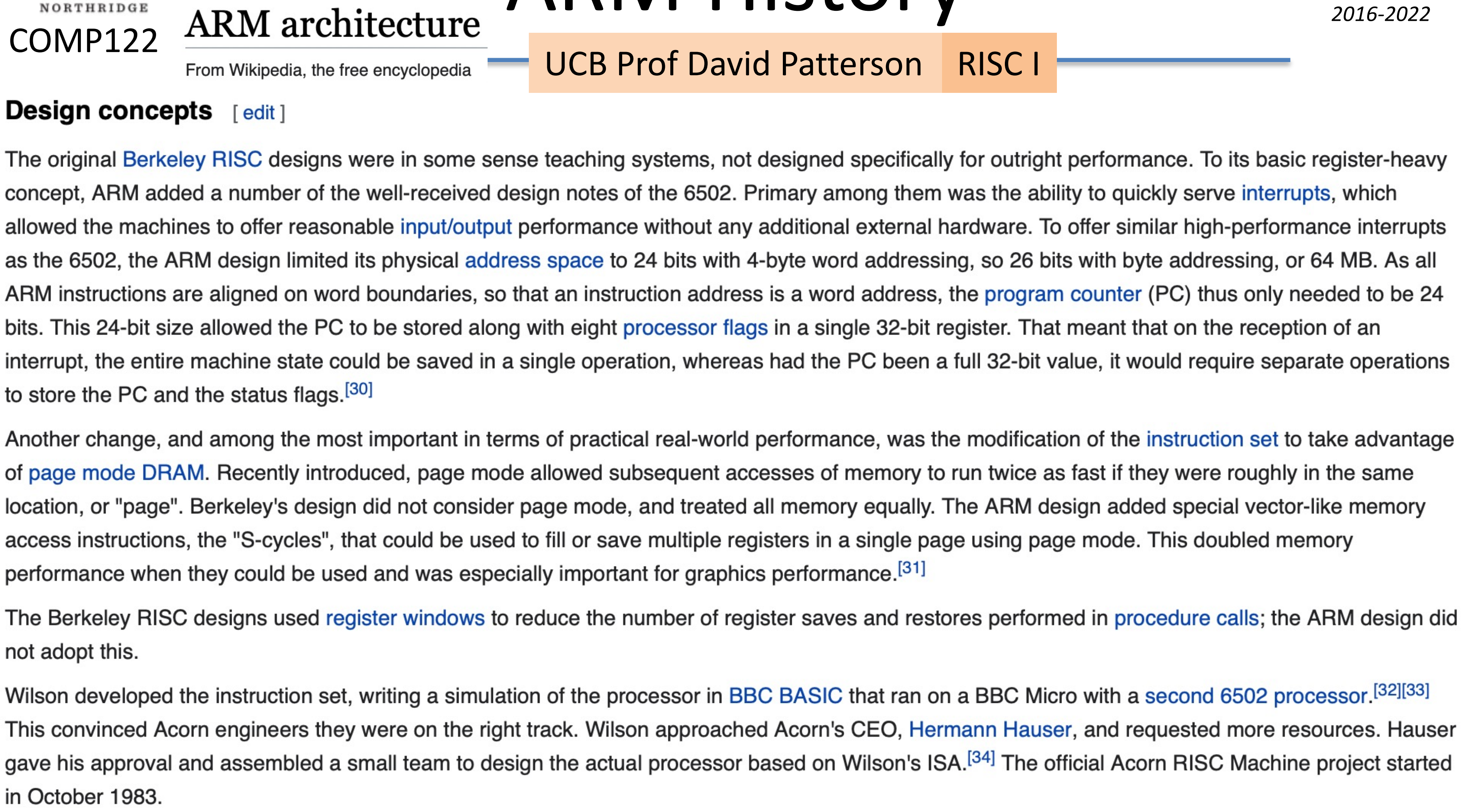


There have been several generations of the ARM design. The original ARM1 used a 32-bit internal structure but had a 26-bit address space that limited it to 64 MB of main memory. This limitation was removed in the ARMv3 series, which has a 32-bit address space, and several additional generations up to ARMv7 remained 32-bit. Released in 2011, the ARMv8-A architecture added support for a 64-bit address space and 64-bit arithmetic with its new 32-bit fixed-length instruction set.[3] Arm Ltd. has also released a series of additional instruction sets for different rules; the "Thumb" extension adds both 32- and 16-bit instructions for improved code density, while Jazelle added instructions for directly handling Java bytecodes, and more recently, JavaScript. More recent changes include the addition of simultaneous multithreading (SMT) for improved performance or fault tolerance.[4]

ARM History

UCB Prof David Patterson

RISC I



ARM architecture

From Wikipedia, the free encyclopedia

Design concepts [edit]

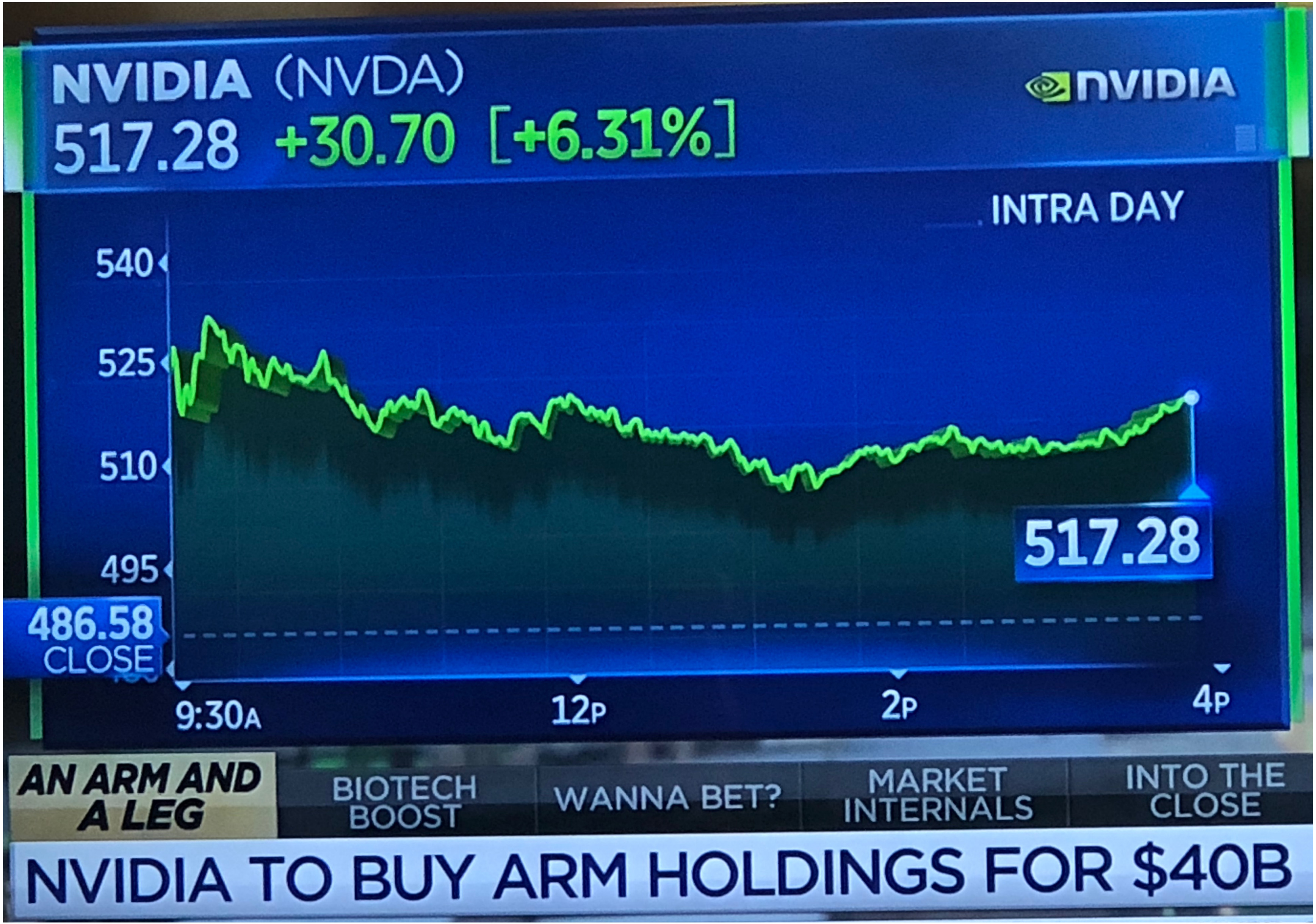
The original Berkeley RISC designs were in some sense teaching systems, not designed specifically for outright performance. To its basic register-heavy concept, ARM added a number of the well-received design notes of the 6502. Primary among them was the ability to quickly serve interrupts, which allowed the machines to offer reasonable input/output performance without any additional external hardware. To offer similar high-performance interrupts as the 6502, the ARM design limited its physical address space to 24 bits with 4-byte word addressing, so 26 bits with byte addressing, or 64 MB. As all ARM instructions are aligned on word boundaries, so that an instruction address is a word address, the program counter (PC) thus only needed to be 24 bits. This 24-bit size allowed the PC to be stored along with eight processor flags in a single 32-bit register. That meant that on the reception of an interrupt, the entire machine state could be saved in a single operation, whereas had the PC been a full 32-bit value, it would require separate operations to store the PC and the status flags.[30]

Another change, and among the most important in terms of practical real-world performance, was the modification of the instruction set to take advantage of page mode DRAM. Recently introduced, page mode allowed subsequent accesses of memory to run twice as fast if they were roughly in the same location, or "page". Berkeley's design did not consider page mode, and treated all memory equally. The ARM design added special vector-like memory access instructions, the "S-cycles", that could be used to fill or save multiple registers in a single page using page mode. This doubled memory performance when they could be used and was especially important for graphics performance.[31]

The Berkeley RISC designs used register windows to reduce the number of register saves and restores performed in procedure calls; the ARM design did not adopt this.

Wilson developed the instruction set, writing a simulation of the processor in BBC BASIC that ran on a BBC Micro with a second 6502 processor.[32][33] This convinced Acorn engineers they were on the right track. Wilson approached Acorn's CEO, Hermann Hauser, and requested more resources. Hauser gave his approval and assembled a small team to design the actual processor based on Wilson's ISA.[34] The official Acorn RISC Machine project started in October 1983.

Nvidia Buys ARM

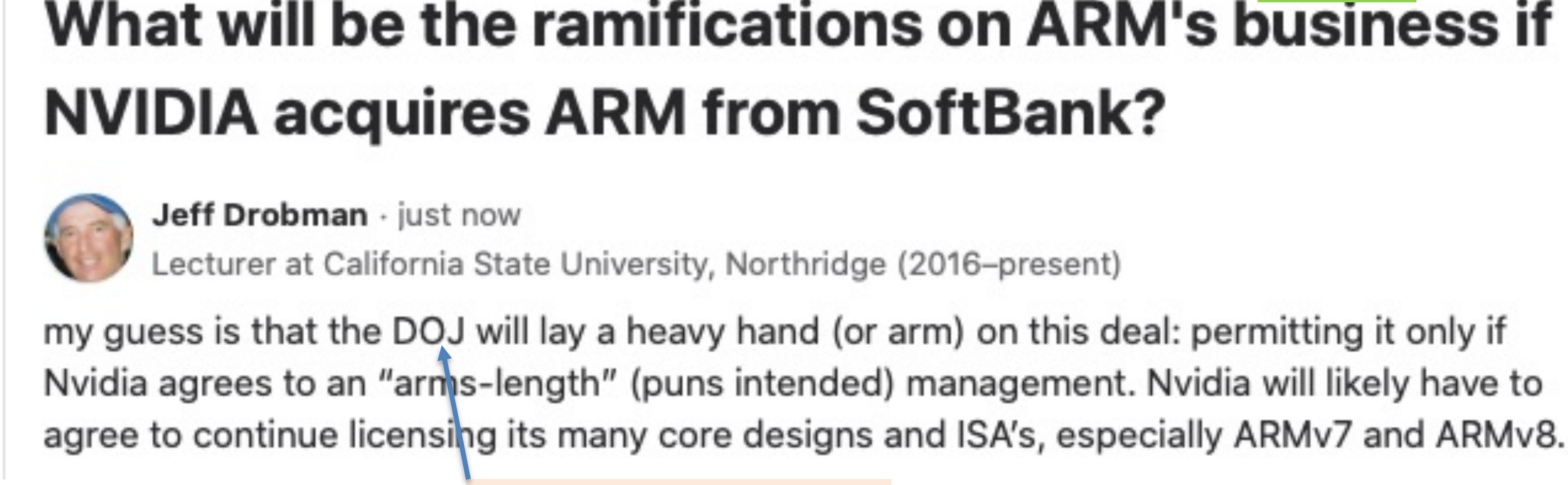


NVIDIA (NVDA)
517.28 +30.70 [+6.31%]

NVIDIA TO BUY ARM HOLDINGS FOR $40B

Nvidia Buys ARM

$40B



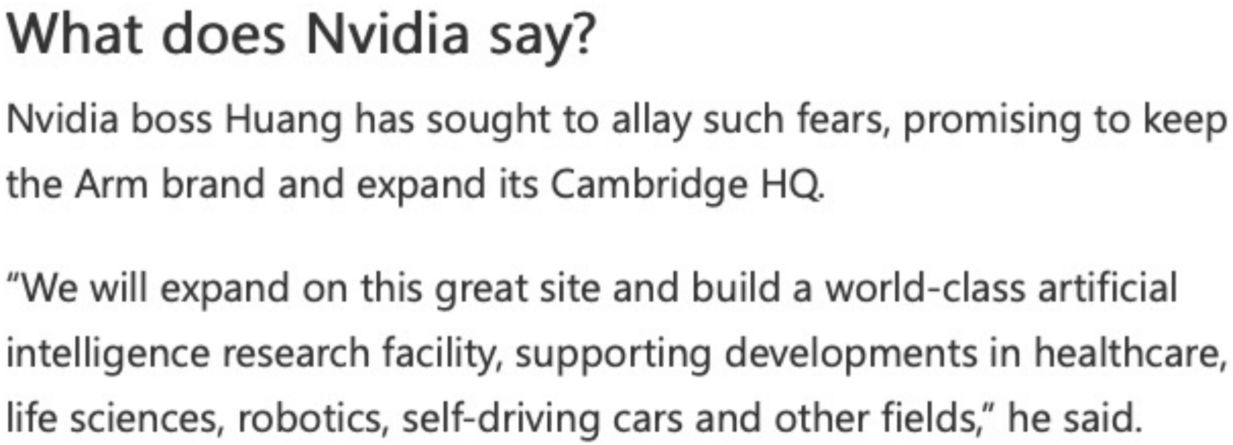
What will be the ramifications on ARM's business if NVIDIA acquires ARM from SoftBank?

Jeff Drobman · just now
Lecturer at California State University, Northridge (2016–present)

my guess is that the DOJ will lay a heavy hand (or arm) on this deal: permitting it only if Nvidia agrees to an "arms-length" (puns intended) management. Nvidia will likely have to agree to continue licensing its many core designs and ISA's, especially ARMv7 and ARMv8.

Actually the UK equiv

- Japanese conglomerate Softbank bought ARM Holdings in 2016 for $32B



What does Nvidia say?

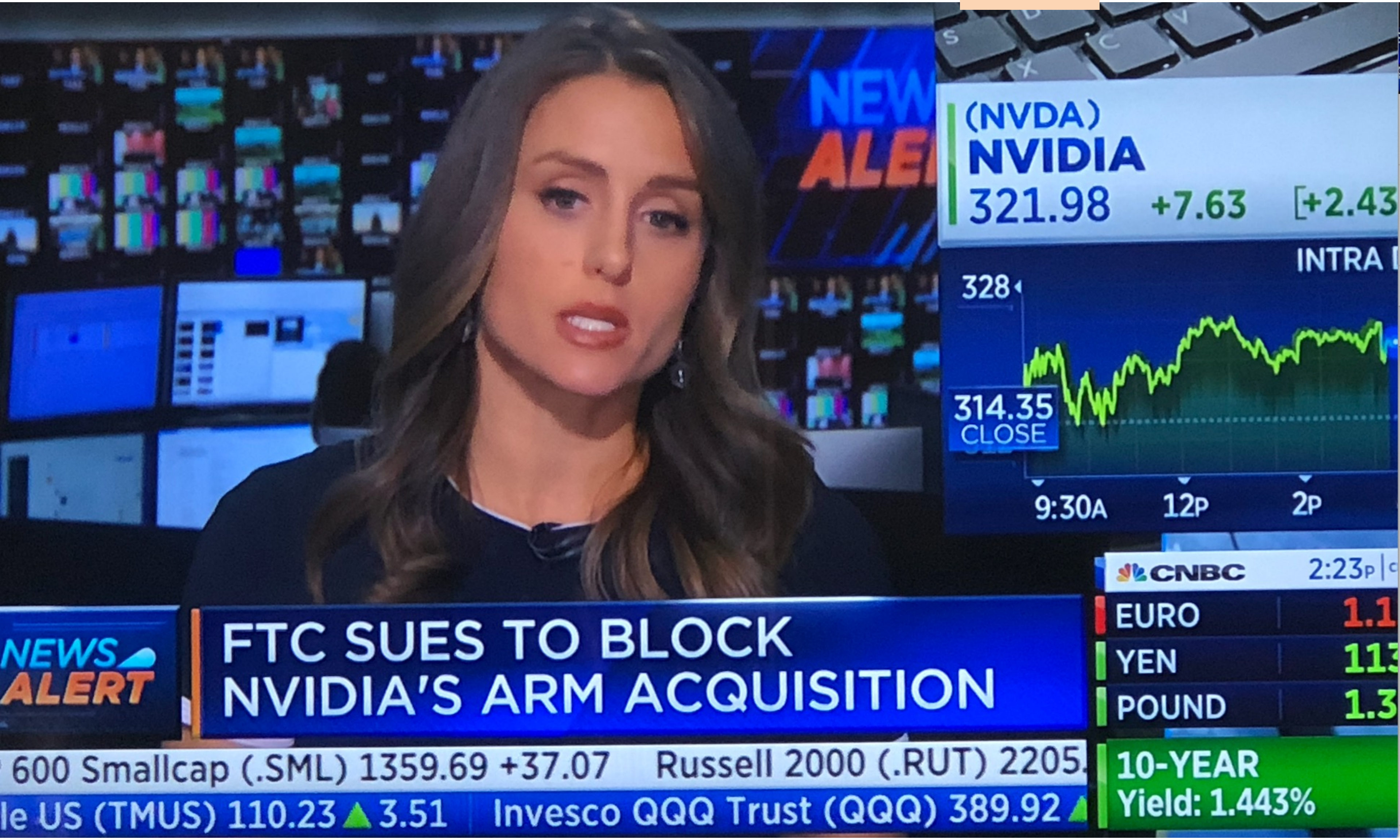
Nvidia boss Huang has sought to allay such fears, promising to keep the Arm brand and expand its Cambridge HQ.

"We will expand on this great site and build a world-class artificial intelligence research facility, supporting developments in healthcare, life sciences, robotics, self-driving cars and other fields," he said.

AI

Nvidia + ARM

12-2-21



NEWS ALERT

FTC SUES TO BLOCK NVIDIA'S ARM ACQUISITION

600 Smallcap (.SML) 1359.69 +37.07 Russell 2000 (.RUT) 2205

le US (TMUS) 110.23 ▲3.51 Invesco QQQ Trust (QQQ) 389.92 ▲

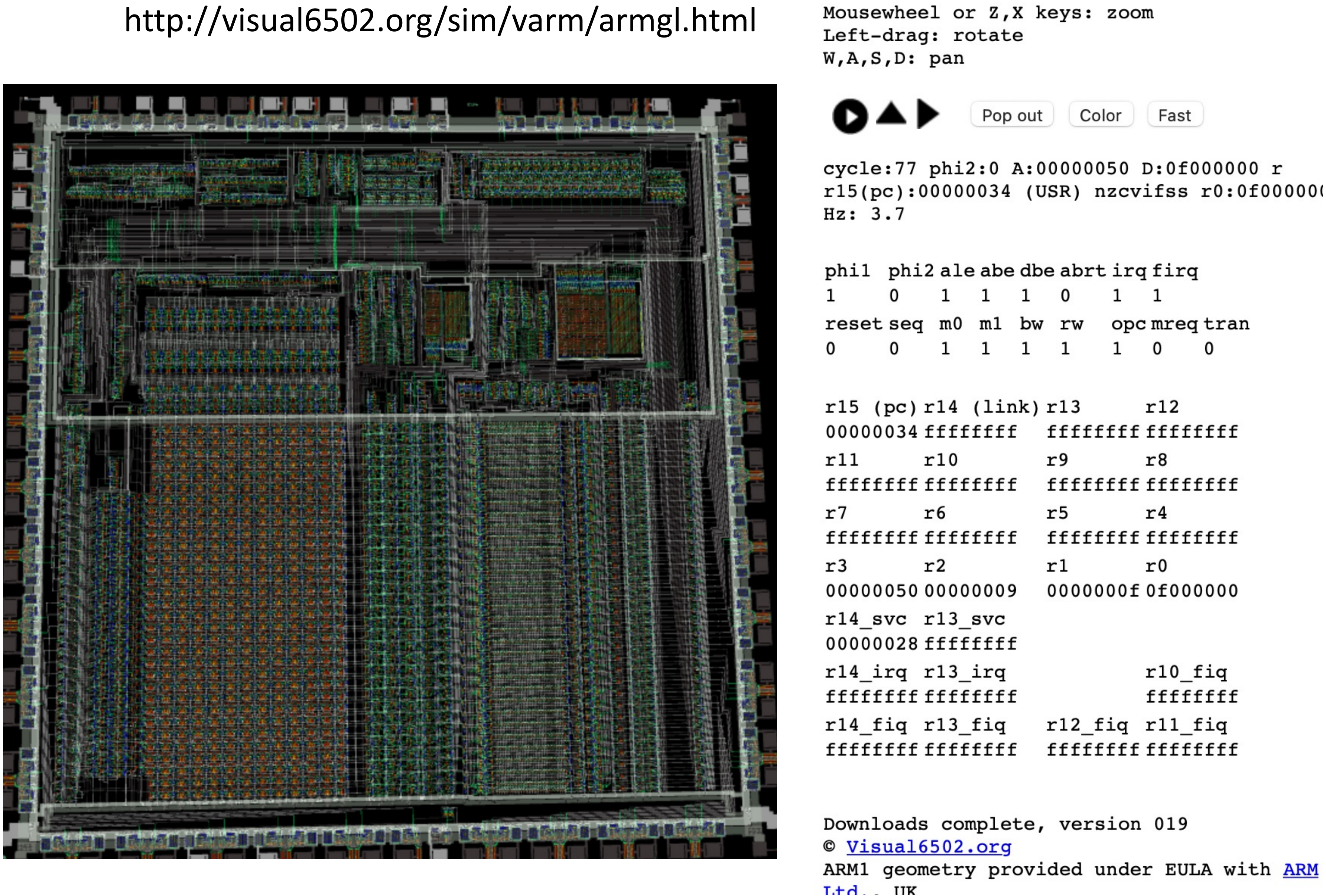
ARM

ARM CPU Models

- ❖ Timeline of models
- ❖ Apple A series (A12-15)
- ❖ New models
  - ❑ CPU: Cortex A-78
  - ❑ GPU: Mali
  - ❑ NPU: Ethos

ARM 1 Die Sim

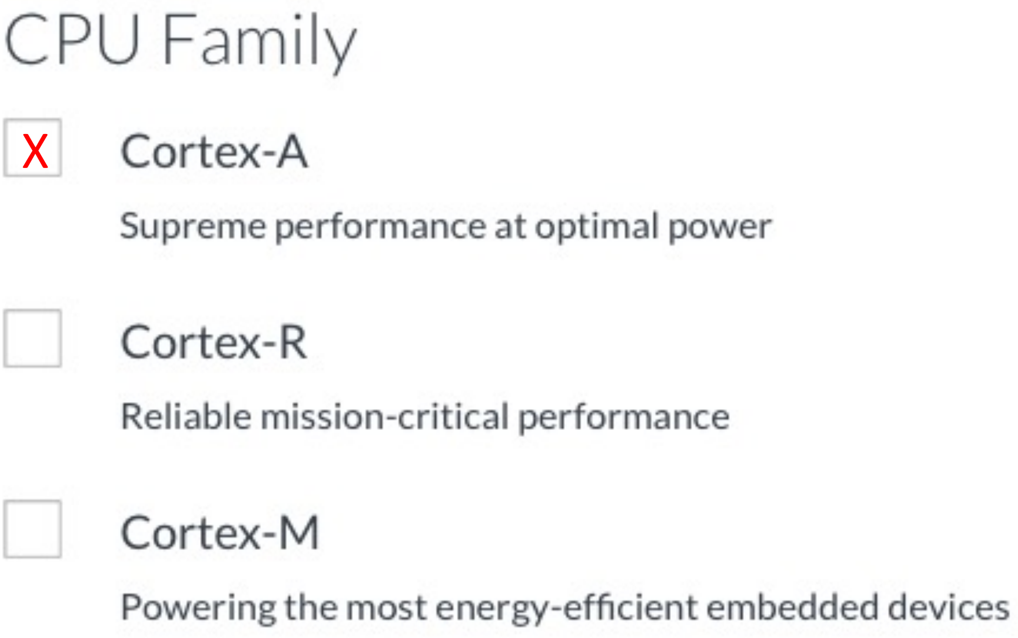
http://visual6502.org/sim/varm/armgl.html



```
Mousewheel or Z,X keys: zoom
Left-drag: rotate
W,A,S,D: pan

cycle:77 phi2:0 A:00000050 D:0f000000 r
r15(pc):00000034 (USR) nzcvifss r0:0f00000
Hz: 3.7

phi1  phi2 ale abe dbe abrt irq firq
1     0    1   1   1   0    1   1
reset seq  m0  m1  bw  rw   opc mreq tran
0     0    1   1   1   1    1   0    0

r15 (pc) r14 (link) r13      r12
00000034 ffffffff   ffffffff ffffffff
r11      r10        r9       r8
ffffffff ffffffff   ffffffff ffffffff
r7       r6         r5       r4
ffffffff ffffffff   ffffffff ffffffff
r3       r2         r1       r0
00000050 00000009   0000000f 0f000000
r14_svc  r13_svc
00000028 ffffffff
r14_irq  r13_irq             r10_fiq
ffffffff ffffffff            ffffffff
r14_fiq  r13_fiq    r12_fiq  r11_fiq
ffffffff ffffffff   ffffffff ffffffff

Downloads complete, version 019
© Visual6502.org
ARM1 geometry provided under EULA with ARM
Ltd., UK
```

ARM

ARM CPUs



CPU Family

☒ Cortex-A

Supreme performance at optimal power

□ Cortex-R

Reliable mission-critical performance

□ Cortex-M

Powering the most energy-efficient embedded devices

8/16-bit **MCU**

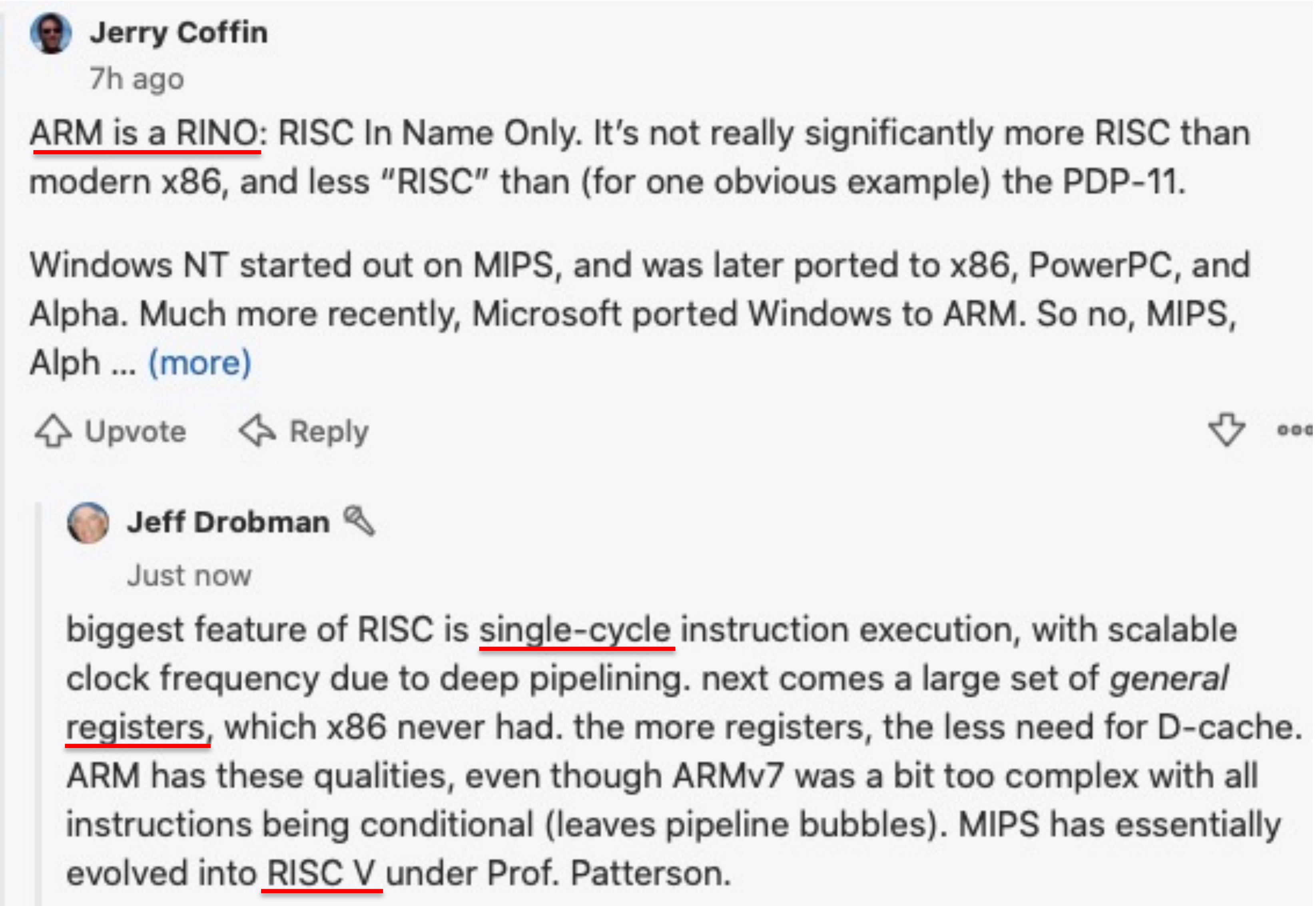
Cortex-A:

- A5
- A7
- A9
- A15
- A17
- A32
- A34
- A35
- A53
- A55
- A57
- A65
- A65AE
- A72
- A73
- A75
- A76AE
- A76
- A77

ARM64/ARMv8

AArch64 or **ARM64** is the 64-bit extension of the ARM architecture.

ARM vs x86 re RISC



Jerry Coffin

7h ago

ARM is a RINO: RISC In Name Only. It's not really significantly more RISC than modern x86, and less "RISC" than (for one obvious example) the PDP-11.

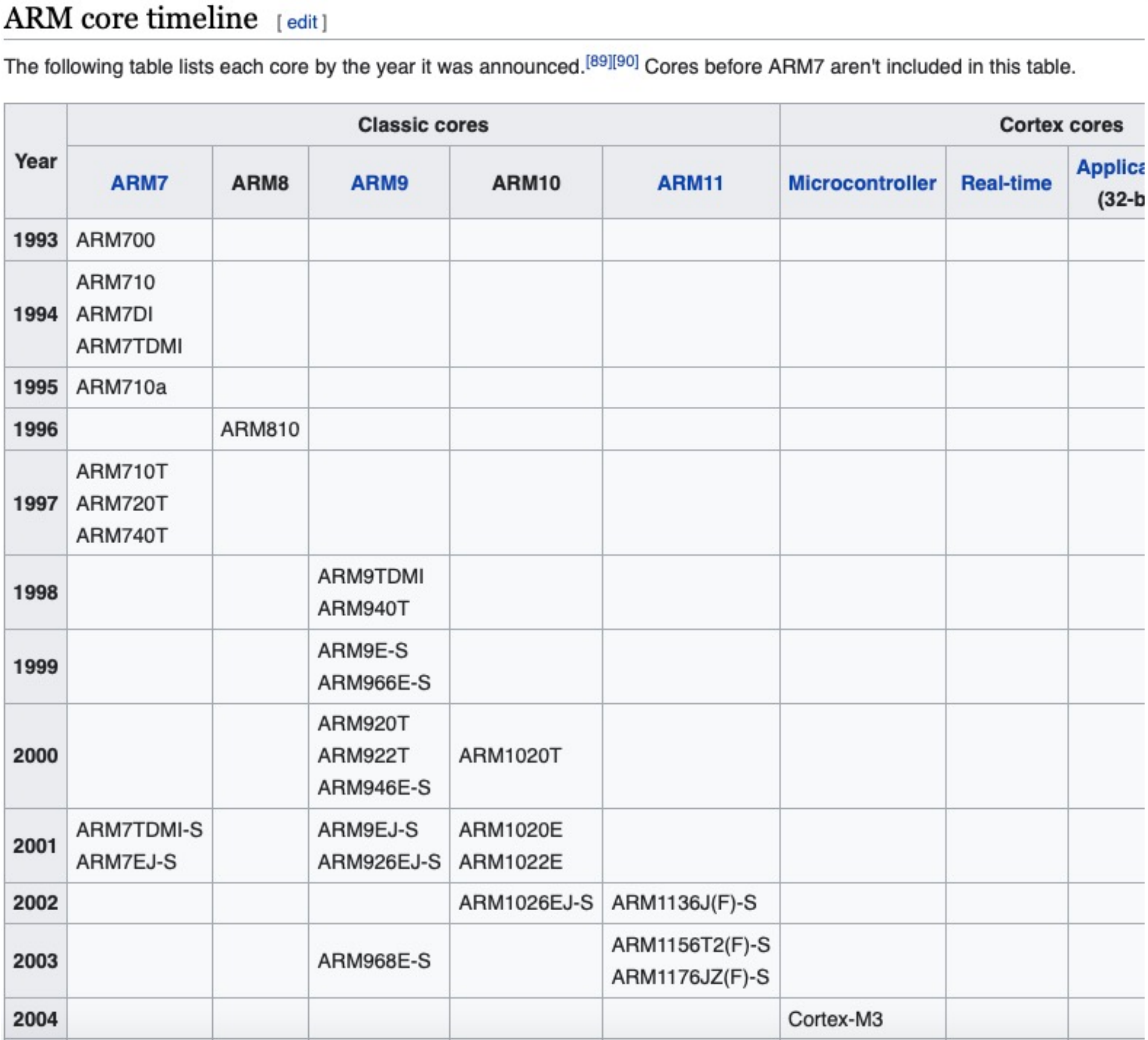
Windows NT started out on MIPS, and was later ported to x86, PowerPC, and Alpha. Much more recently, Microsoft ported Windows to ARM. So no, MIPS, Alph ... (more)

Jeff Drobman

Just now

biggest feature of RISC is single-cycle instruction execution, with scalable clock frequency due to deep pipelining. next comes a large set of *general* registers, which x86 never had. the more registers, the less need for D-cache. ARM has these qualities, even though ARMv7 was a bit too complex with all instructions being conditional (leaves pipeline bubbles). MIPS has essentially evolved into RISC V under Prof. Patterson.

ARM Cores 7-11 Timeline

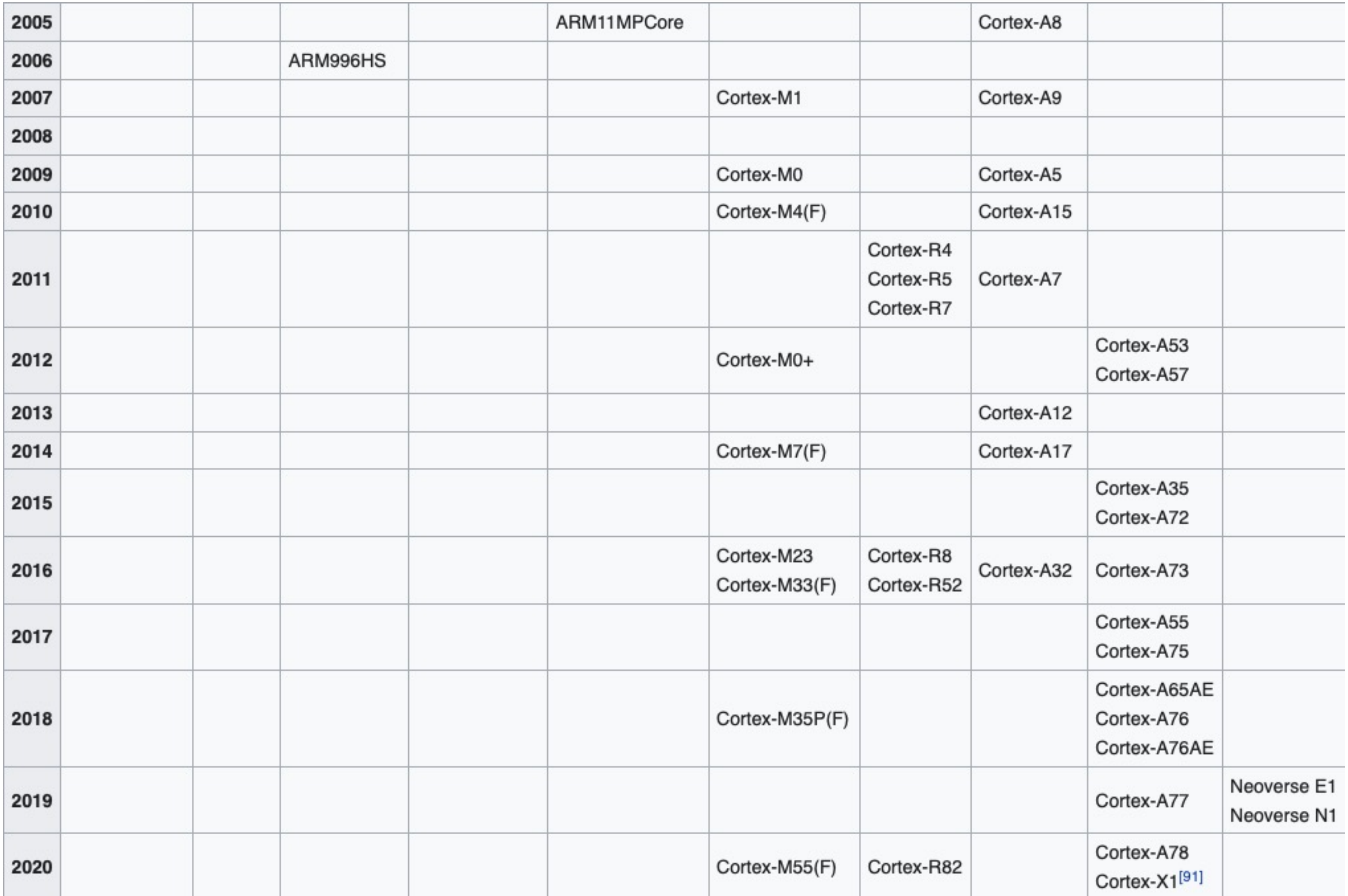


ARM core timeline [edit]

The following table lists each core by the year it was announced.[89][90] Cores before ARM7 aren't included in this table.

Year	Classic cores					Cortex cores		
	ARM7	ARM8	ARM9	ARM10	ARM11	Microcontroller	Real-time	Applica (32-b
1993	ARM700							
1994	ARM710 ARM7DI ARM7TDMI							
1995	ARM710a							
1996		ARM810						
1997	ARM710T ARM720T ARM740T							
1998			ARM9TDMI ARM940T					
1999			ARM9E-S ARM966E-S					
2000			ARM920T ARM922T ARM946E-S	ARM1020T				
2001	ARM7TDMI-S ARM7EJ-S		ARM9EJ-S ARM926EJ-S	ARM1020E ARM1022E				
2002				ARM1026EJ-S	ARM1136J(F)-S			
2003			ARM968E-S		ARM1156T2(F)-S ARM1176JZ(F)-S			
2004						Cortex-M3		

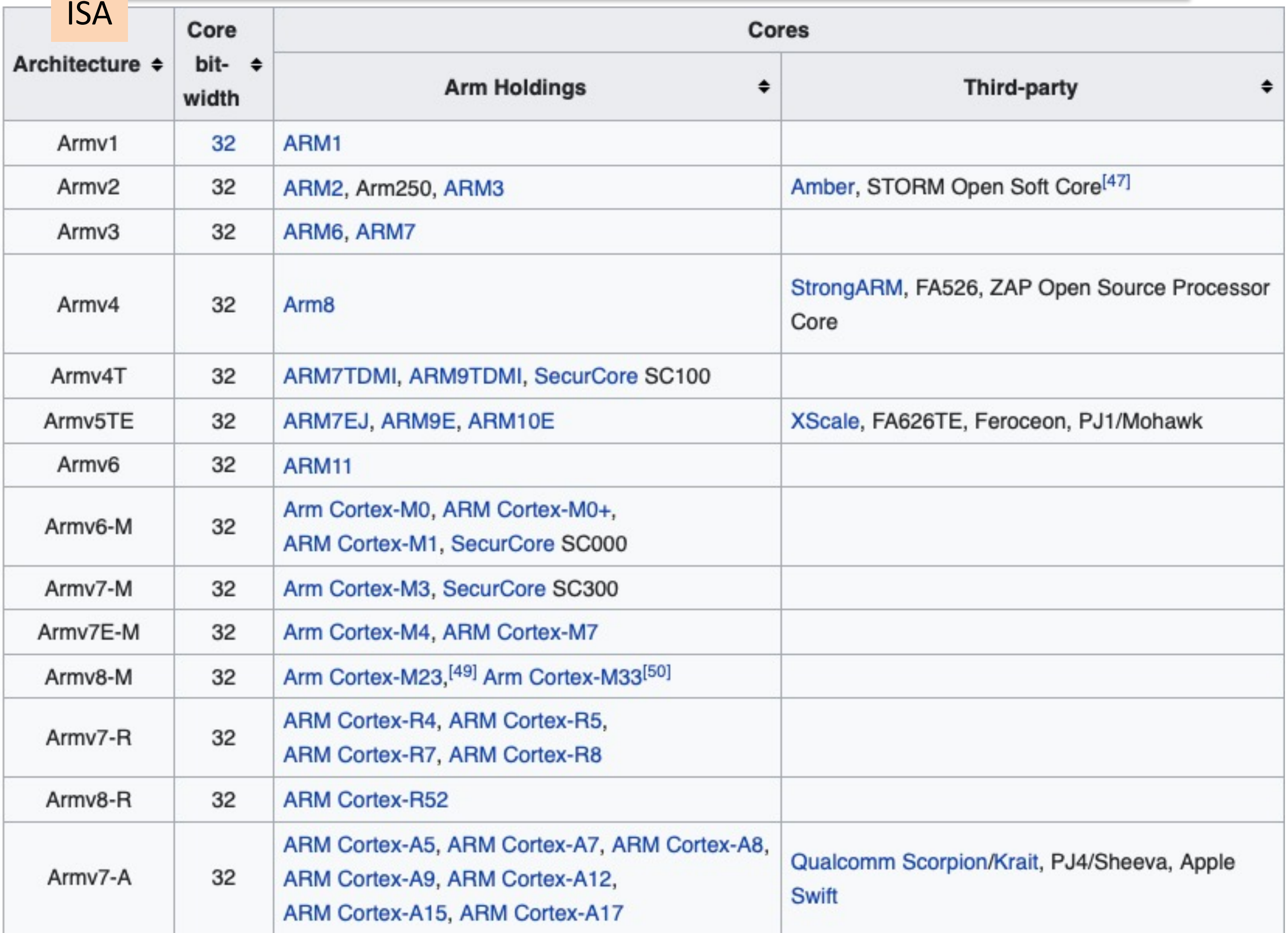
ARM Cores 7-11 Timeline



2005					ARM11MPCore			Cortex-A8		
2006			ARM996HS							
2007						Cortex-M1		Cortex-A9		
2008										
2009						Cortex-M0		Cortex-A5		
2010						Cortex-M4(F)		Cortex-A15		
2011							Cortex-R4 Cortex-R5 Cortex-R7	Cortex-A7		
2012						Cortex-M0+			Cortex-A53 Cortex-A57	
2013								Cortex-A12		
2014						Cortex-M7(F)		Cortex-A17		
2015									Cortex-A35 Cortex-A72	
2016						Cortex-M23 Cortex-M33(F)	Cortex-R8 Cortex-R52	Cortex-A32	Cortex-A73	
2017									Cortex-A55 Cortex-A75	
2018						Cortex-M35P(F)			Cortex-A65AE Cortex-A76 Cortex-A76AE	
2019									Cortex-A77	Neoverse E1 Neoverse N1
2020						Cortex-M55(F)	Cortex-R82		Cortex-A78 Cortex-X1[91]	

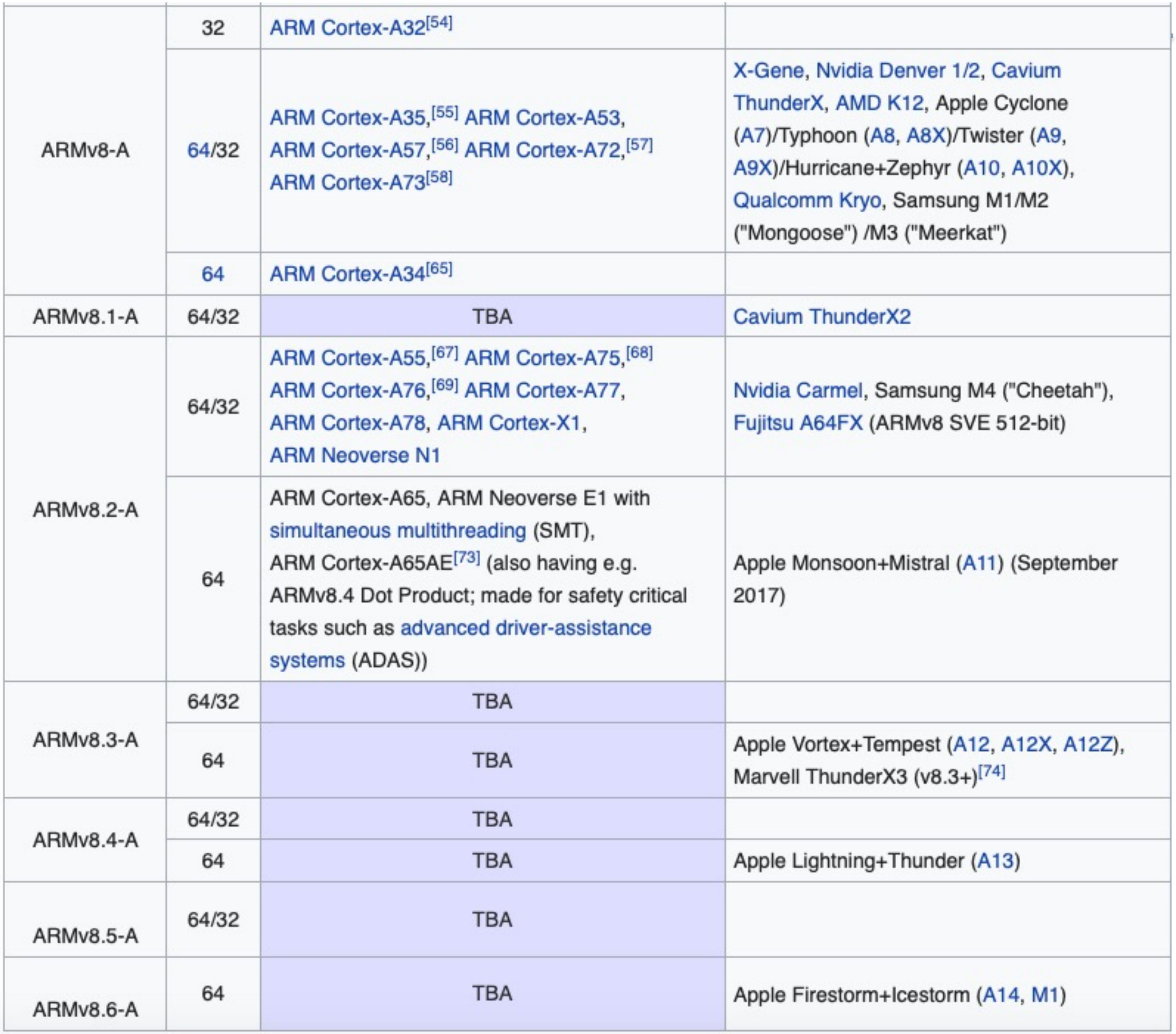
ARM Timeline (v1-7)

ISA



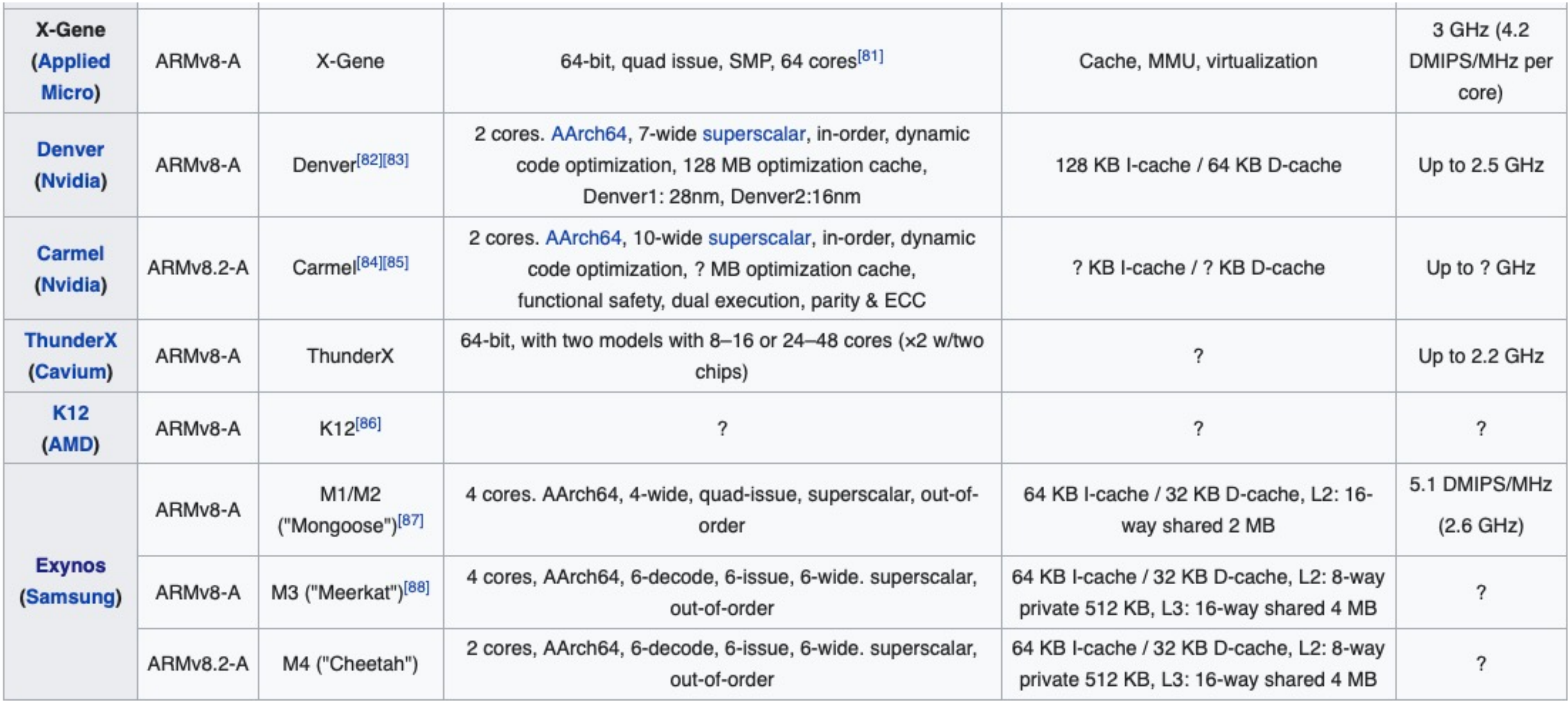
Architecture	Core bit-width	Cores	
		Arm Holdings	Third-party
Armv1	32	ARM1	
Armv2	32	ARM2, Arm250, ARM3	Amber, STORM Open Soft Core[47]
Armv3	32	ARM6, ARM7	
Armv4	32	Arm8	StrongARM, FA526, ZAP Open Source Processor Core
Armv4T	32	ARM7TDMI, ARM9TDMI, SecurCore SC100	
Armv5TE	32	ARM7EJ, ARM9E, ARM10E	XScale, FA626TE, Feroceon, PJ1/Mohawk
Armv6	32	ARM11	
Armv6-M	32	Arm Cortex-M0, ARM Cortex-M0+, ARM Cortex-M1, SecurCore SC000	
Armv7-M	32	Arm Cortex-M3, SecurCore SC300	
Armv7E-M	32	Arm Cortex-M4, ARM Cortex-M7	
Armv8-M	32	Arm Cortex-M23,[49] Arm Cortex-M33[50]	
Armv7-R	32	ARM Cortex-R4, ARM Cortex-R5, ARM Cortex-R7, ARM Cortex-R8	
Armv8-R	32	ARM Cortex-R52	
Armv7-A	32	ARM Cortex-A5, ARM Cortex-A7, ARM Cortex-A8, ARM Cortex-A9, ARM Cortex-A12, ARM Cortex-A15, ARM Cortex-A17	Qualcomm Scorpion/Krait, PJ4/Sheeva, Apple Swift

ARM Timeline (v8)



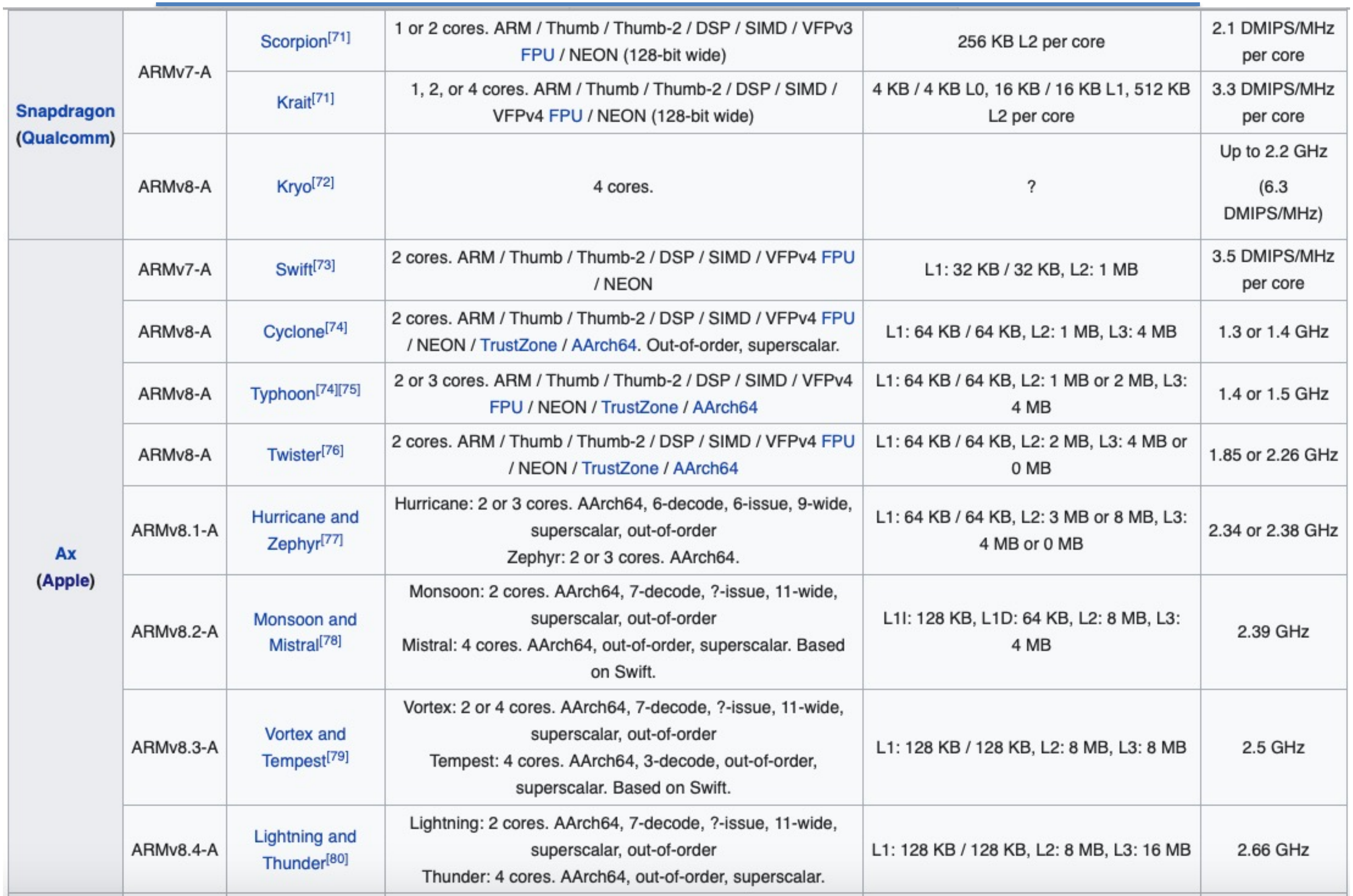
ARMv8-A	32	ARM Cortex-A32[54]	
	64/32	ARM Cortex-A35,[55] ARM Cortex-A53, ARM Cortex-A57,[56] ARM Cortex-A72,[57] ARM Cortex-A73[58]	X-Gene, Nvidia Denver 1/2, Cavium ThunderX, AMD K12, Apple Cyclone (A7)/Typhoon (A8, A8X)/Twister (A9, A9X)/Hurricane+Zephyr (A10, A10X), Qualcomm Kryo, Samsung M1/M2 ("Mongoose") /M3 ("Meerkat")
	64	ARM Cortex-A34[65]	
ARMv8.1-A	64/32	TBA	Cavium ThunderX2
ARMv8.2-A	64/32	ARM Cortex-A55,[67] ARM Cortex-A75,[68] ARM Cortex-A76,[69] ARM Cortex-A77, ARM Cortex-A78, ARM Cortex-X1, ARM Neoverse N1	Nvidia Carmel, Samsung M4 ("Cheetah"), Fujitsu A64FX (ARMv8 SVE 512-bit)
	64	ARM Cortex-A65, ARM Neoverse E1 with simultaneous multithreading (SMT), ARM Cortex-A65AE[73] (also having e.g. ARMv8.4 Dot Product; made for safety critical tasks such as advanced driver-assistance systems (ADAS))	Apple Monsoon+Mistral (A11) (September 2017)
ARMv8.3-A	64/32	TBA	
	64	TBA	Apple Vortex+Tempest (A12, A12X, A12Z), Marvell ThunderX3 (v8.3+)[74]
ARMv8.4-A	64/32	TBA	
	64	TBA	Apple Lightning+Thunder (A13)
ARMv8.5-A	64/32	TBA	
ARMv8.6-A	64	TBA	Apple Firestorm+Icestorm (A14, M1)

ARM 3rd Party SoC



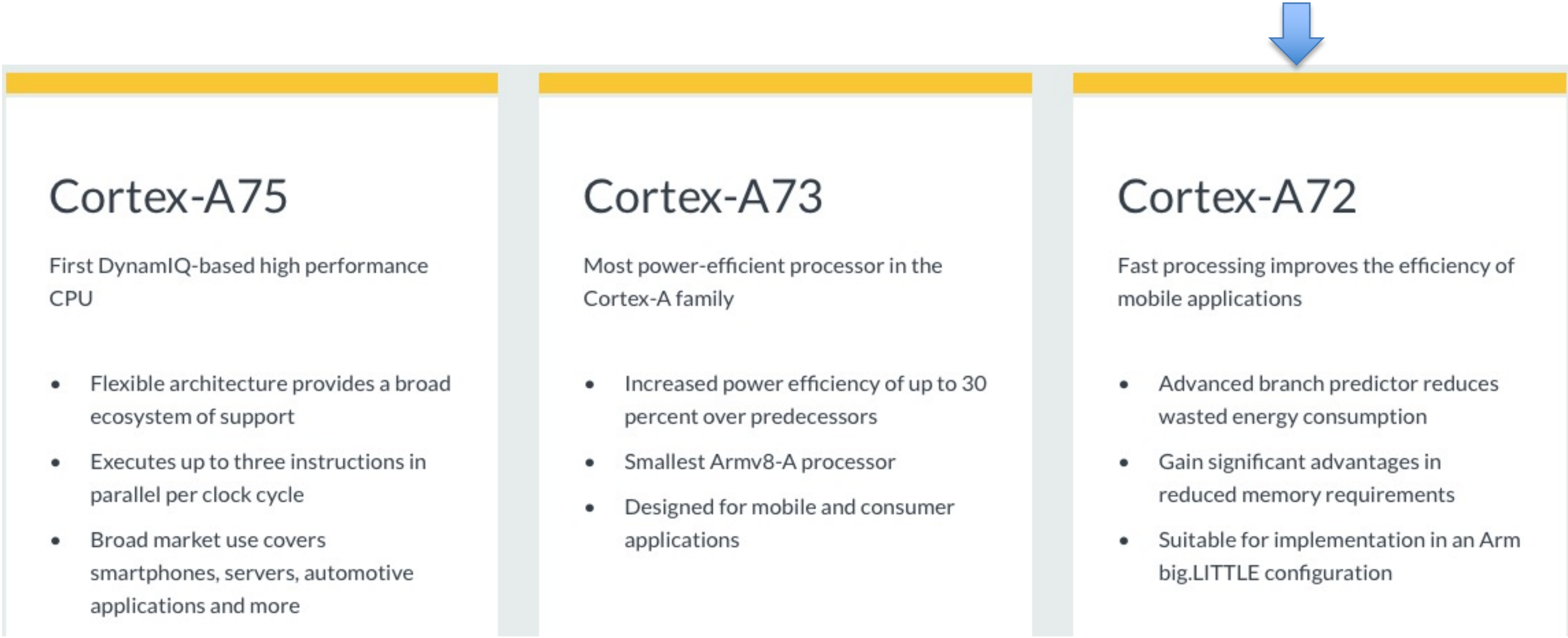
X-Gene (Applied Micro)	ARMv8-A	X-Gene	64-bit, quad issue, SMP, 64 cores[81]	Cache, MMU, virtualization	3 GHz (4.2 DMIPS/MHz per core)
Denver (Nvidia)	ARMv8-A	Denver[82][83]	2 cores. AArch64, 7-wide superscalar, in-order, dynamic code optimization, 128 MB optimization cache, Denver1: 28nm, Denver2:16nm	128 KB I-cache / 64 KB D-cache	Up to 2.5 GHz
Carmel (Nvidia)	ARMv8.2-A	Carmel[84][85]	2 cores. AArch64, 10-wide superscalar, in-order, dynamic code optimization, ? MB optimization cache, functional safety, dual execution, parity & ECC	? KB I-cache / ? KB D-cache	Up to ? GHz
ThunderX (Cavium)	ARMv8-A	ThunderX	64-bit, with two models with 8–16 or 24–48 cores (×2 w/two chips)	?	Up to 2.2 GHz
K12 (AMD)	ARMv8-A	K12[86]	?	?	?
Exynos (Samsung)	ARMv8-A	M1/M2 ("Mongoose")[87]	4 cores. AArch64, 4-wide, quad-issue, superscalar, out-of-order	64 KB I-cache / 32 KB D-cache, L2: 16-way shared 2 MB	5.1 DMIPS/MHz (2.6 GHz)
	ARMv8-A	M3 ("Meerkat")[88]	4 cores, AArch64, 6-decode, 6-issue, 6-wide. superscalar, out-of-order	64 KB I-cache / 32 KB D-cache, L2: 8-way private 512 KB, L3: 16-way shared 4 MB	?
	ARMv8.2-A	M4 ("Cheetah")	2 cores, AArch64, 6-decode, 6-issue, 6-wide. superscalar, out-of-order	64 KB I-cache / 32 KB D-cache, L2: 8-way private 512 KB, L3: 16-way shared 4 MB	?

ARM SoC – Apple, Qualcomm



Snapdragon (Qualcomm)	ARMv7-A	Scorpion[71]	1 or 2 cores. ARM / Thumb / Thumb-2 / DSP / SIMD / VFPv3 FPU / NEON (128-bit wide)	256 KB L2 per core	2.1 DMIPS/MHz per core
		Krait[71]	1, 2, or 4 cores. ARM / Thumb / Thumb-2 / DSP / SIMD / VFPv4 FPU / NEON (128-bit wide)	4 KB / 4 KB L0, 16 KB / 16 KB L1, 512 KB L2 per core	3.3 DMIPS/MHz per core
	ARMv8-A	Kryo[72]	4 cores.	?	Up to 2.2 GHz (6.3 DMIPS/MHz)
Ax (Apple)	ARMv7-A	Swift[73]	2 cores. ARM / Thumb / Thumb-2 / DSP / SIMD / VFPv4 FPU / NEON	L1: 32 KB / 32 KB, L2: 1 MB	3.5 DMIPS/MHz per core
	ARMv8-A	Cyclone[74]	2 cores. ARM / Thumb / Thumb-2 / DSP / SIMD / VFPv4 FPU / NEON / TrustZone / AArch64. Out-of-order, superscalar.	L1: 64 KB / 64 KB, L2: 1 MB, L3: 4 MB	1.3 or 1.4 GHz
	ARMv8-A	Typhoon[74][75]	2 or 3 cores. ARM / Thumb / Thumb-2 / DSP / SIMD / VFPv4 FPU / NEON / TrustZone / AArch64	L1: 64 KB / 64 KB, L2: 1 MB or 2 MB, L3: 4 MB	1.4 or 1.5 GHz
	ARMv8-A	Twister[76]	2 cores. ARM / Thumb / Thumb-2 / DSP / SIMD / VFPv4 FPU / NEON / TrustZone / AArch64	L1: 64 KB / 64 KB, L2: 2 MB, L3: 4 MB or 0 MB	1.85 or 2.26 GHz
	ARMv8.1-A	Hurricane and Zephyr[77]	Hurricane: 2 or 3 cores. AArch64, 6-decode, 6-issue, 9-wide, superscalar, out-of-order Zephyr: 2 or 3 cores. AArch64.	L1: 64 KB / 64 KB, L2: 3 MB or 8 MB, L3: 4 MB or 0 MB	2.34 or 2.38 GHz
	ARMv8.2-A	Monsoon and Mistral[78]	Monsoon: 2 cores. AArch64, 7-decode, ?-issue, 11-wide, superscalar, out-of-order Mistral: 4 cores. AArch64, out-of-order, superscalar. Based on Swift.	L1I: 128 KB, L1D: 64 KB, L2: 8 MB, L3: 4 MB	2.39 GHz
	ARMv8.3-A	Vortex and Tempest[79]	Vortex: 2 or 4 cores. AArch64, 7-decode, ?-issue, 11-wide, superscalar, out-of-order Tempest: 4 cores. AArch64, 3-decode, out-of-order, superscalar. Based on Swift.	L1: 128 KB / 128 KB, L2: 8 MB, L3: 8 MB	2.5 GHz
	ARMv8.4-A	Lightning and Thunder[80]	Lightning: 2 cores. AArch64, 7-decode, ?-issue, 11-wide, superscalar, out-of-order Thunder: 4 cores. AArch64, out-of-order, superscalar.	L1: 128 KB / 128 KB, L2: 8 MB, L3: 16 MB	2.66 GHz

ARM



Cortex-A75

First DynamIQ-based high performance CPU

- Flexible architecture provides a broad ecosystem of support
- Executes up to three instructions in parallel per clock cycle
- Broad market use covers smartphones, servers, automotive applications and more

Cortex-A73

Most power-efficient processor in the Cortex-A family

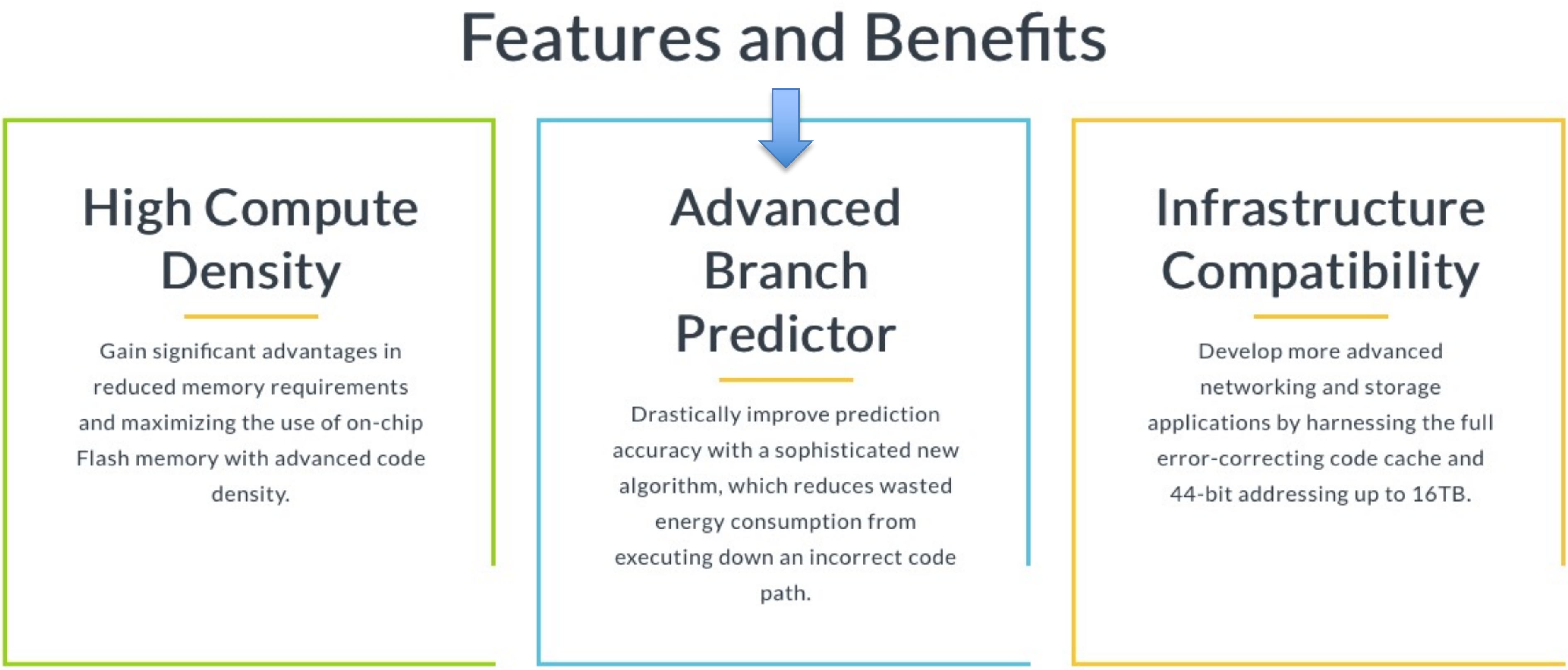
- Increased power efficiency of up to 30 percent over predecessors
- Smallest Armv8-A processor
- Designed for mobile and consumer applications

Cortex-A72

Fast processing improves the efficiency of mobile applications

- Advanced branch predictor reduces wasted energy consumption
- Gain significant advantages in reduced memory requirements
- Suitable for implementation in an Arm big.LITTLE configuration

ARM



Features and Benefits

High Compute Density

Gain significant advantages in reduced memory requirements and maximizing the use of on-chip Flash memory with advanced code density.

Advanced Branch Predictor

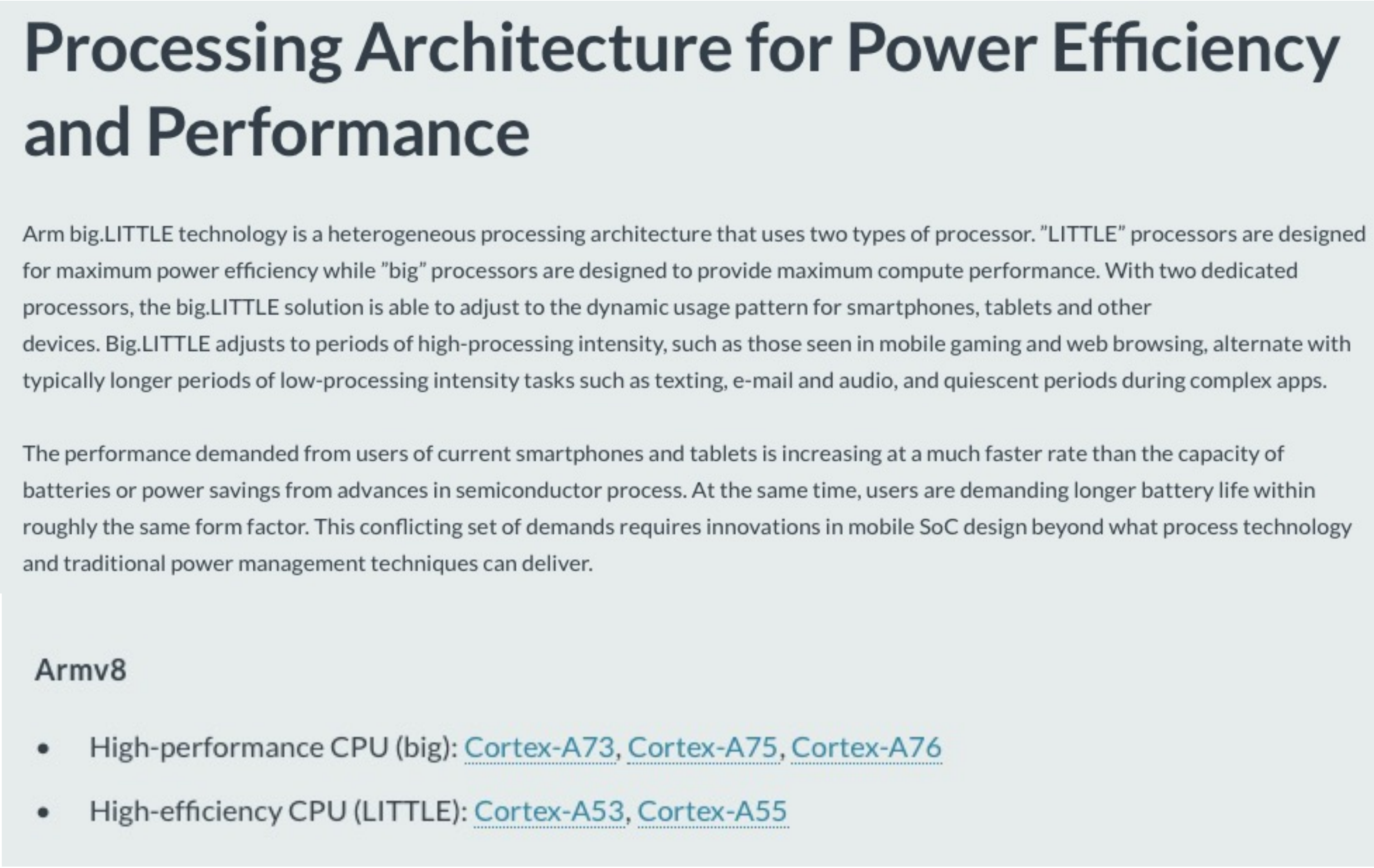
Drastically improve prediction accuracy with a sophisticated new algorithm, which reduces wasted energy consumption from executing down an incorrect code path.

Infrastructure Compatibility

Develop more advanced networking and storage applications by harnessing the full error-correcting code cache and 44-bit addressing up to 16TB.

ARM

ARM



Processing Architecture for Power Efficiency and Performance

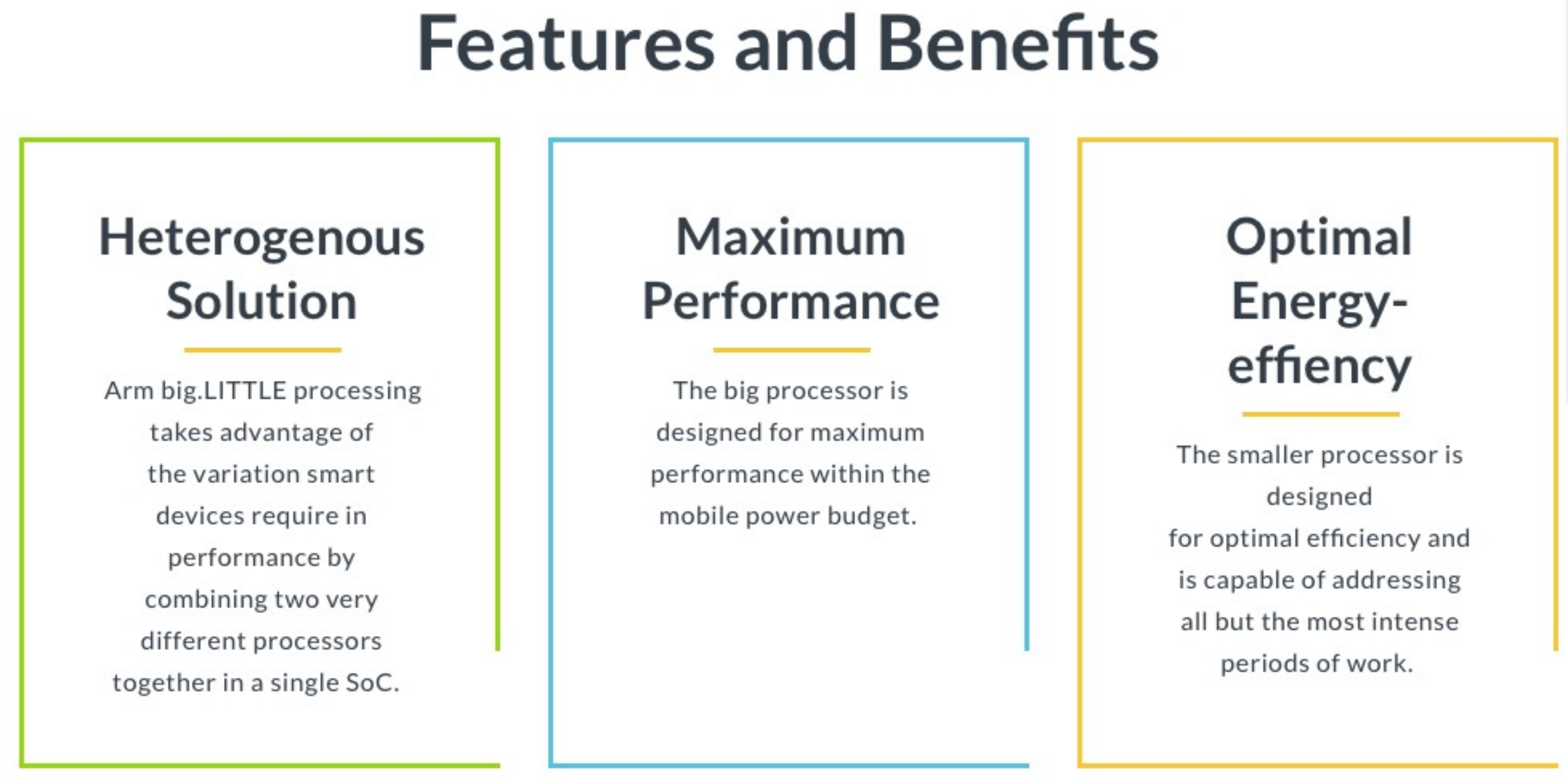
Arm big.LITTLE technology is a heterogeneous processing architecture that uses two types of processor. ”LITTLE” processors are designed for maximum power efficiency while ”big” processors are designed to provide maximum compute performance. With two dedicated processors, the big.LITTLE solution is able to adjust to the dynamic usage pattern for smartphones, tablets and other devices. Big.LITTLE adjusts to periods of high-processing intensity, such as those seen in mobile gaming and web browsing, alternate with typically longer periods of low-processing intensity tasks such as texting, e-mail and audio, and quiescent periods during complex apps.

The performance demanded from users of current smartphones and tablets is increasing at a much faster rate than the capacity of batteries or power savings from advances in semiconductor process. At the same time, users are demanding longer battery life within roughly the same form factor. This conflicting set of demands requires innovations in mobile SoC design beyond what process technology and traditional power management techniques can deliver.

Armv8

- High-performance CPU (big): Cortex-A73, Cortex-A75, Cortex-A76
- High-efficiency CPU (LITTLE): Cortex-A53, Cortex-A55

ARM



Features and Benefits

Heterogenous Solution

Arm big.LITTLE processing takes advantage of the variation smart devices require in performance by combining two very different processors together in a single SoC.

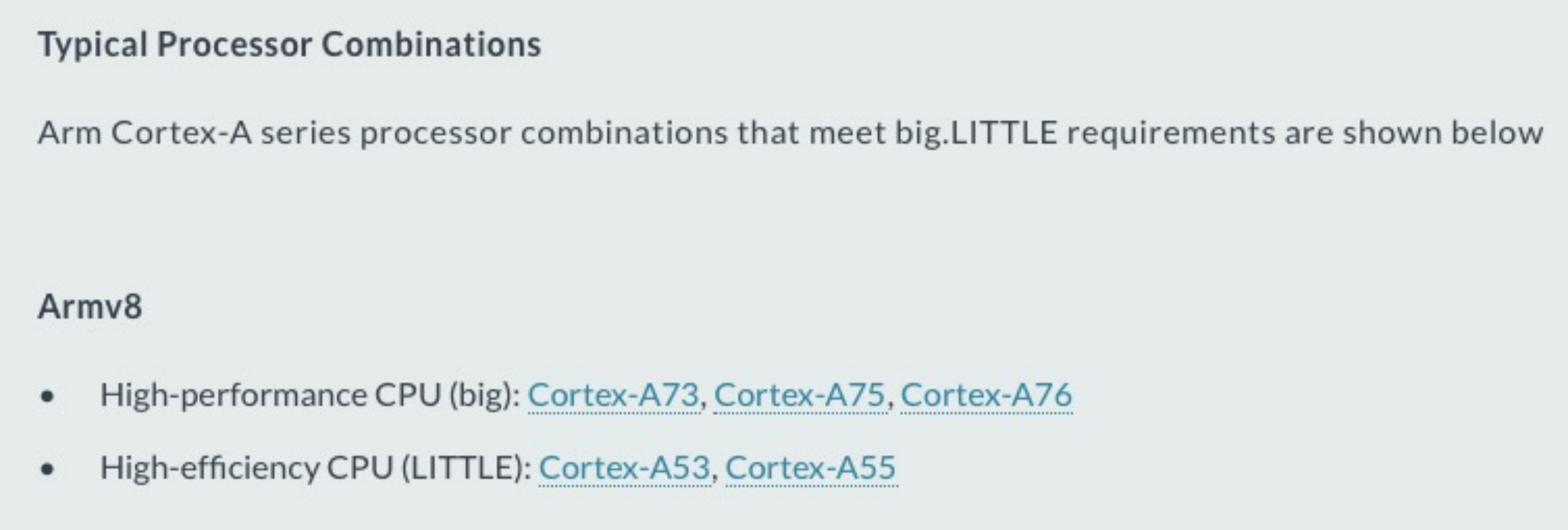
Maximum Performance

The big processor is designed for maximum performance within the mobile power budget.

Optimal Energy-effiency

The smaller processor is designed for optimal efficiency and is capable of addressing all but the most intense periods of work.

ARM



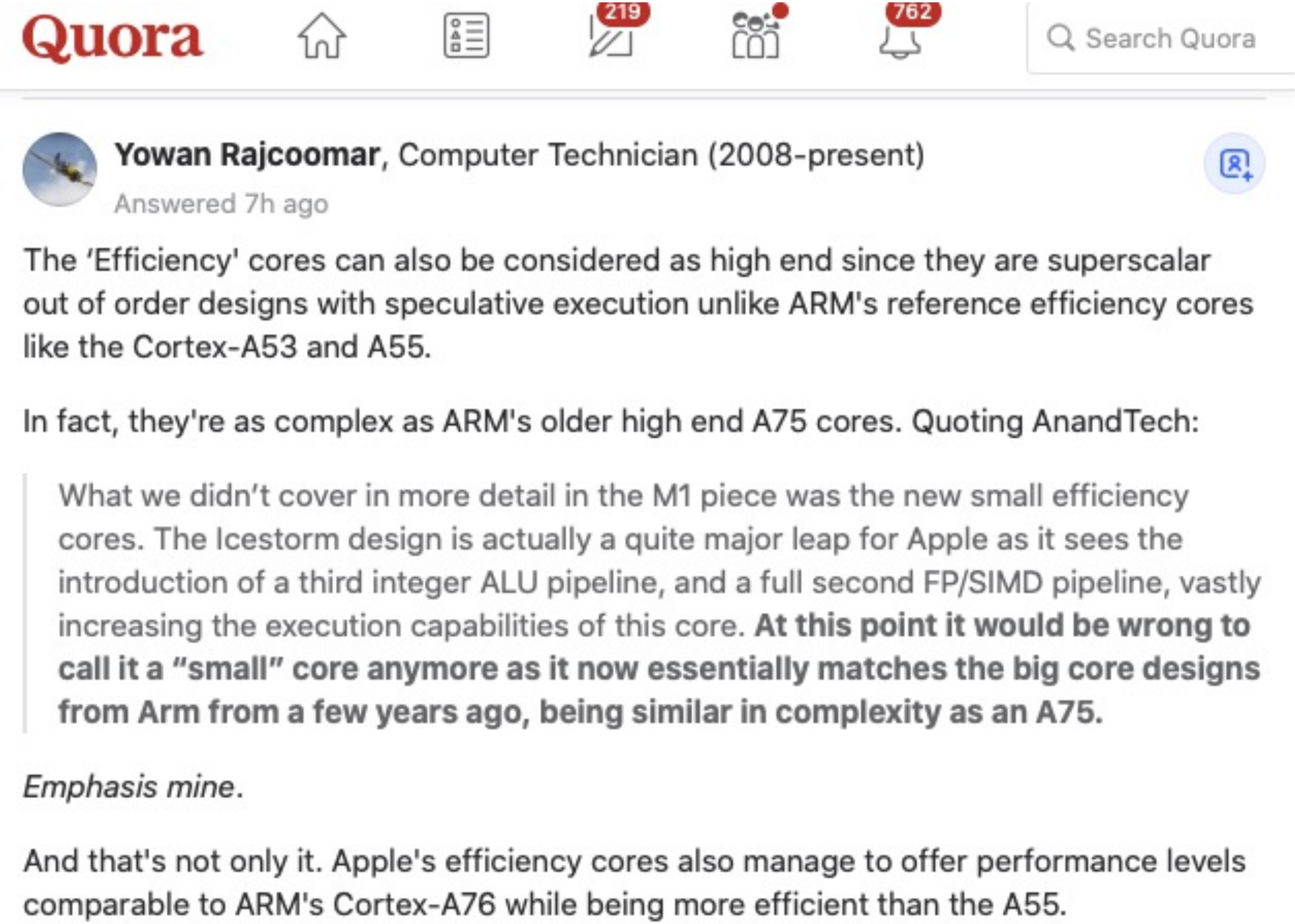
Typical Processor Combinations

Arm Cortex-A series processor combinations that meet big.LITTLE requirements are shown below

Armv8

- High-performance CPU (big): Cortex-A73, Cortex-A75, Cortex-A76
- High-efficiency CPU (LITTLE): Cortex-A53, Cortex-A55

ARM



Yowan Rajcoomar, Computer Technician (2008-present)
Answered 7h ago

The 'Efficiency' cores can also be considered as high end since they are superscalar out of order designs with speculative execution unlike ARM's reference efficiency cores like the Cortex-A53 and A55.

In fact, they're as complex as ARM's older high end A75 cores. Quoting AnandTech:

> What we didn't cover in more detail in the M1 piece was the new small efficiency cores. The Icestorm design is actually a quite major leap for Apple as it sees the introduction of a third integer ALU pipeline, and a full second FP/SIMD pipeline, vastly increasing the execution capabilities of this core. **At this point it would be wrong to call it a "small" core anymore as it now essentially matches the big core designs from Arm from a few years ago, being similar in complexity as an A75.**

Emphasis mine.

And that's not only it. Apple's efficiency cores also manage to offer performance levels comparable to ARM's Cortex-A76 while being more efficient than the A55.

Architecture

Hennessy & Patterson

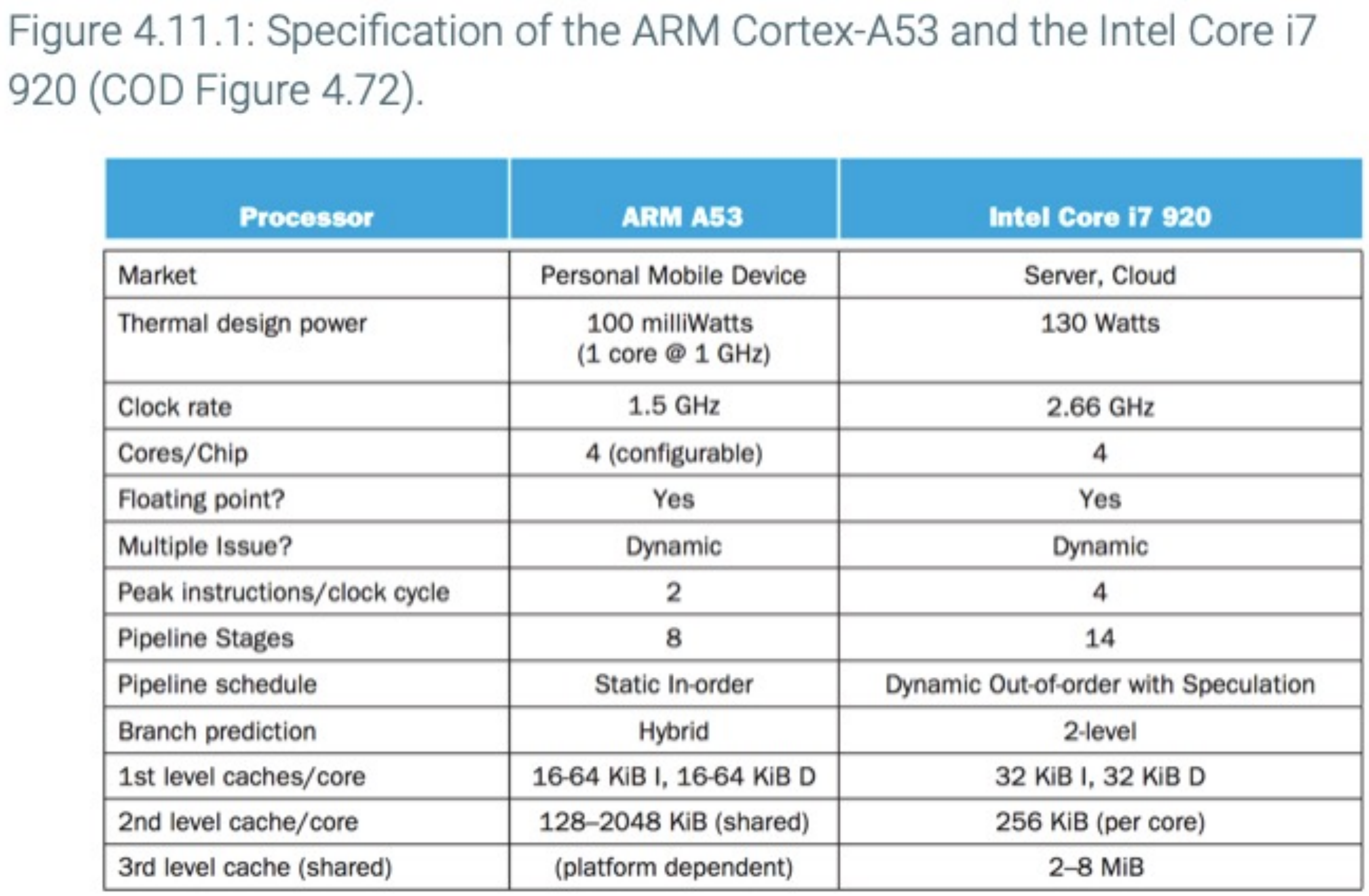
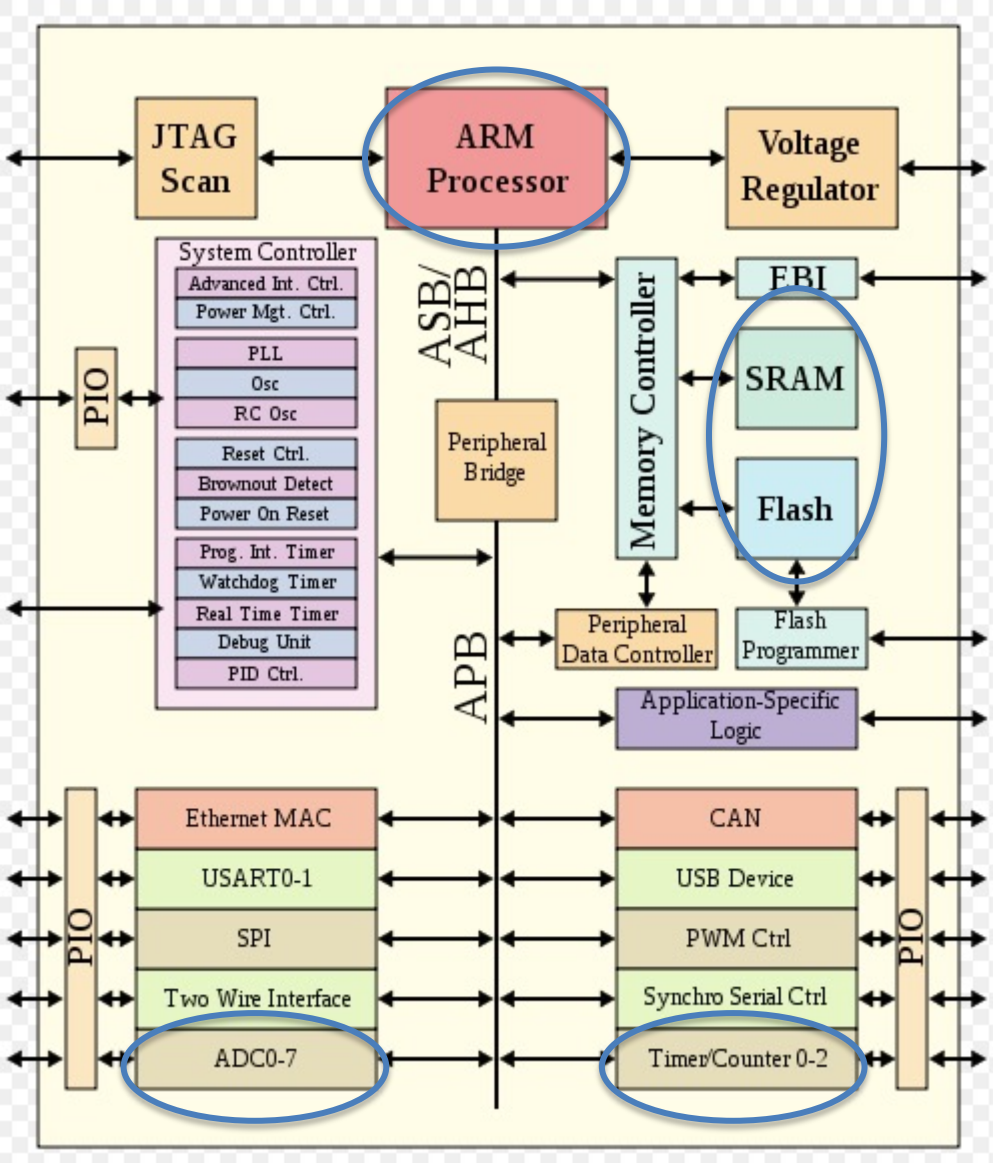


Figure 4.11.1: Specification of the ARM Cortex-A53 and the Intel Core i7 920 (COD Figure 4.72).

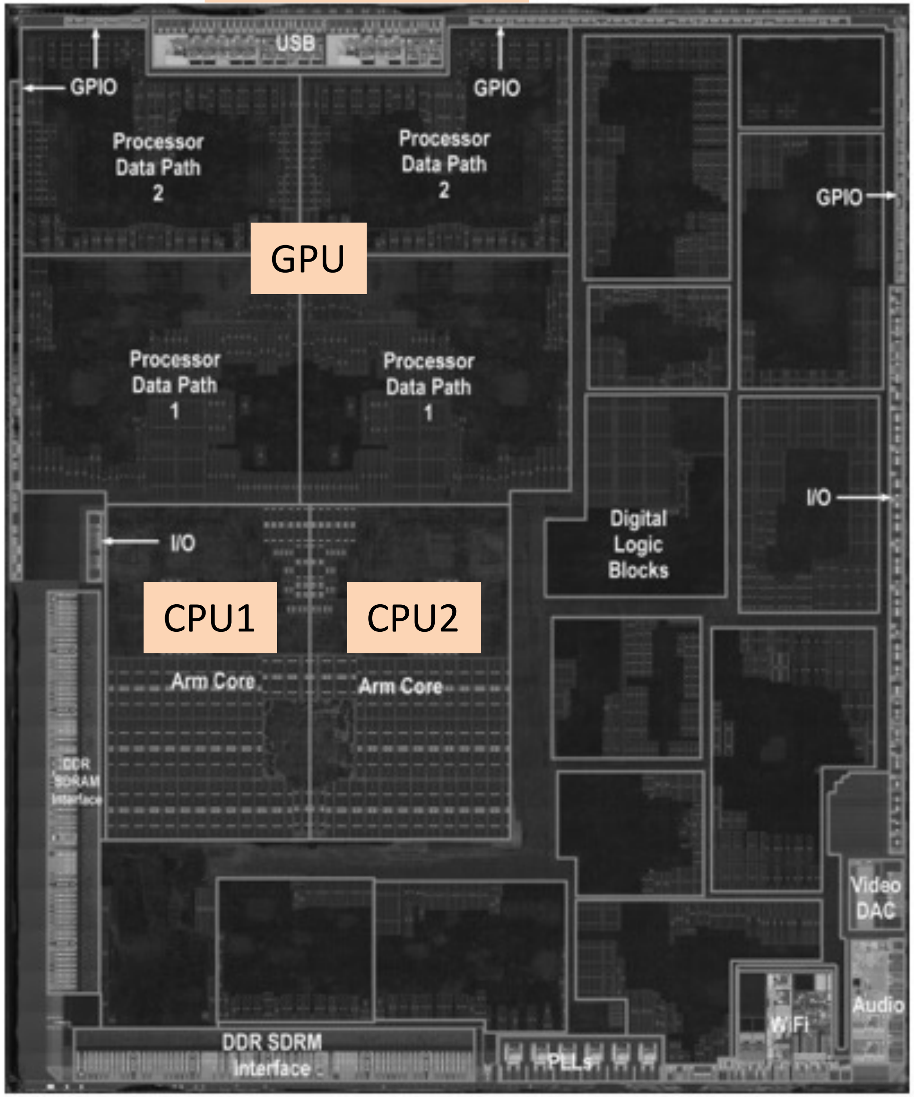
Processor	ARM A53	Intel Core i7 920
Market	Personal Mobile Device	Server, Cloud
Thermal design power	100 milliWatts (1 core @ 1 GHz)	130 Watts
Clock rate	1.5 GHz	2.66 GHz
Cores/Chip	4 (configurable)	4
Floating point?	Yes	Yes
Multiple Issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline Stages	8	14
Pipeline schedule	Static In-order	Dynamic Out-of-order with Speculation
Branch prediction	Hybrid	2-level
1st level caches/core	16-64 KiB I, 16-64 KiB D	32 KiB I, 32 KiB D
2nd level cache/core	128–2048 KiB (shared)	256 KiB (per core)
3rd level cache (shared)	(platform dependent)	2–8 MiB

Hardware-ARM SoC

Hennessy & Patterson



Apple ARM **A5** 12.1x10.2mm



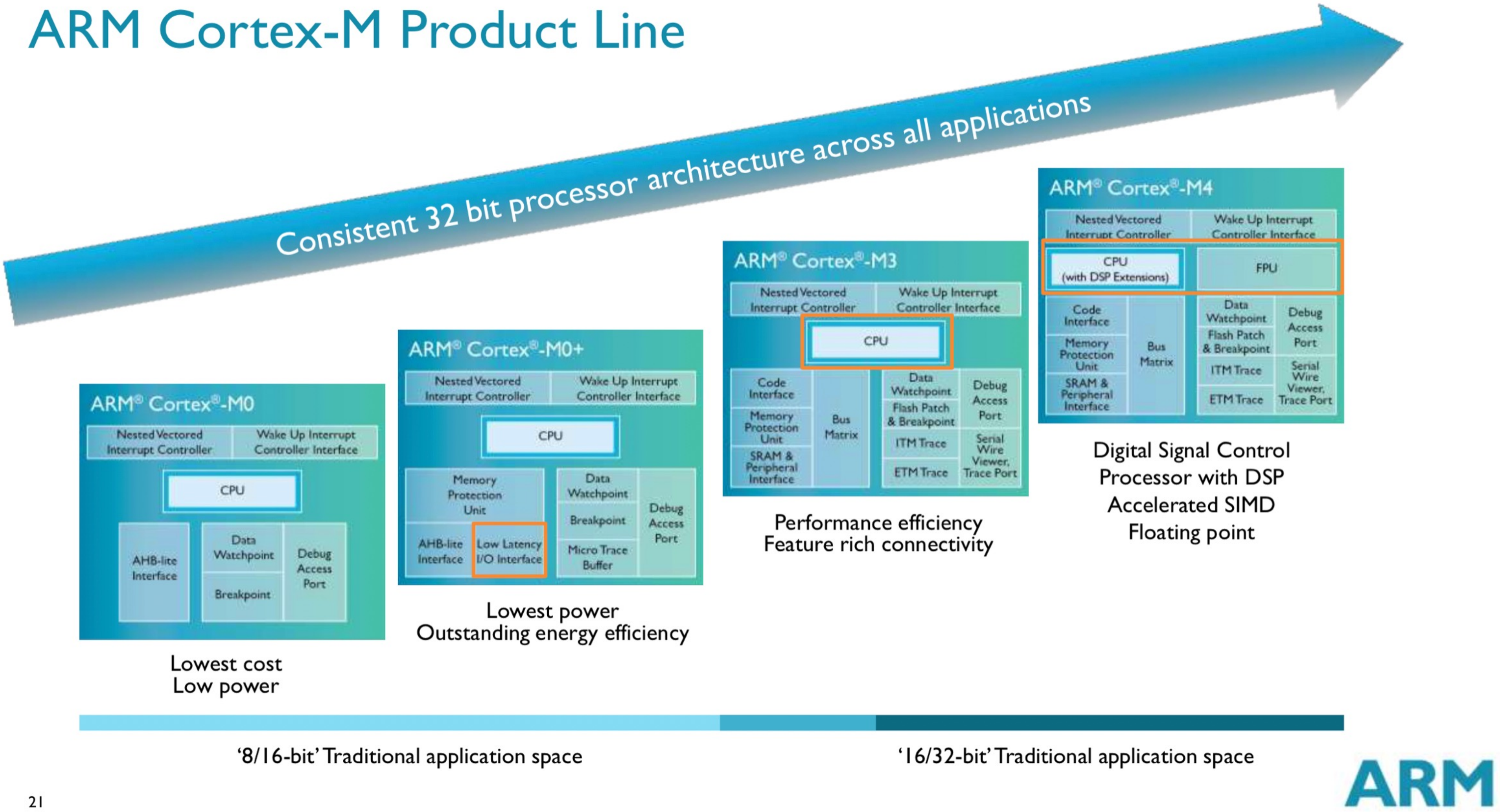
CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP122

ARM M Series

DR JEFF SOFTWARE
INDIE APP DEVELOPER
© Jeff Drobman 2016-2022

MCU

ARM Cortex-M Product Line


Consistent 32 bit processor architecture across all applications
ARM® Cortex®-M0
Nested Vectored Interrupt Controller
Wake Up Interrupt Controller Interface
CPU
AHB-lite Interface
Data Watchpoint
Breakpoint
Debug Access Port
ARM® Cortex®-M0+
Nested Vectored Interrupt Controller
Wake Up Interrupt Controller Interface
CPU
Memory Protection Unit
AHB-lite Interface
Low Latency I/O Interface
Data Watchpoint
Breakpoint
Micro Trace Buffer
Debug Access Port
ARM® Cortex®-M3
Nested Vectored Interrupt Controller
Wake Up Interrupt Controller Interface
CPU
Code Interface
Memory Protection Unit
SRAM & Peripheral Interface
Bus Matrix
Data Watchpoint
Flash Patch & Breakpoint
ITM Trace
ETM Trace
Debug Access Port
Serial Wire Viewer, Trace Port
ARM® Cortex®-M4
Nested Vectored Interrupt Controller
Wake Up Interrupt Controller Interface
CPU (with DSP Extensions)
FPU
Code Interface
Memory Protection Unit
SRAM & Peripheral Interface
Bus Matrix
Data Watchpoint
Flash Patch & Breakpoint
ITM Trace
ETM Trace
Debug Access Port
Serial Wire Viewer, Trace Port
ARM

Lowest cost
Low power

Lowest power
Outstanding energy efficiency

Performance efficiency
Feature rich connectivity

Digital Signal Control
Processor with DSP
Accelerated SIMD
Floating point

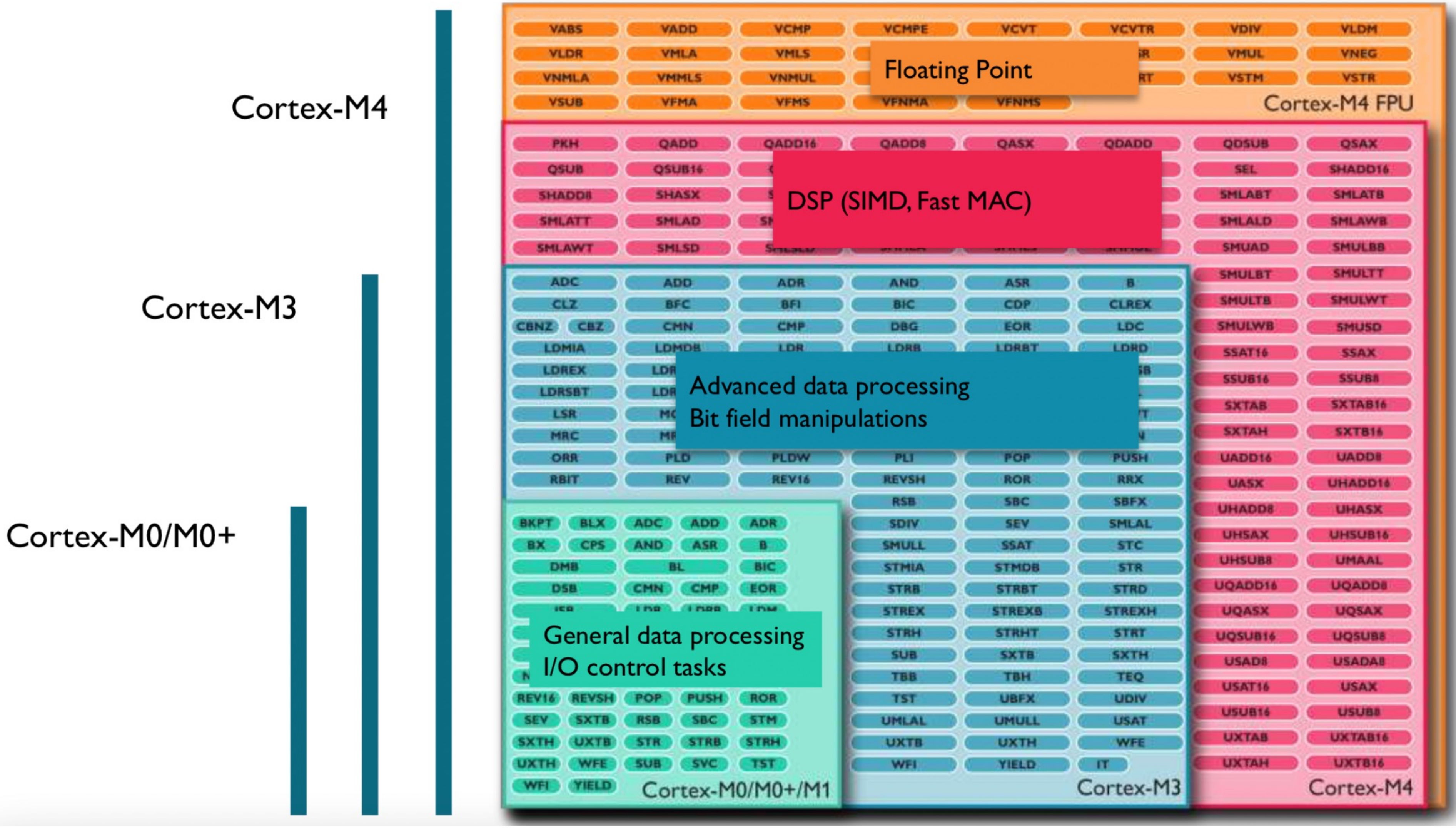
'8/16-bit' Traditional application space

'16/32-bit' Traditional application space

ARM M Series

MCU

ARM Cortex-M Instruction Set Architecture



Cortex-M4

Cortex-M3

Cortex-M0/M0+

VABS VADD VCMP VCMPE VCVT VCVTR VDIV VLDM
VLDR VMLA VMLS VMUL VNEG
VNMLA VMMLS VNMUL VSTM VSTR
VSUB VFMA VFMS VFNMA VFNMS

Floating Point

Cortex-M4 FPU

PKH QADD QADD16 QADD8 QASX QDADD QDSUB QSAX
QSUB QSUB16 SEL SHADD16
SHADD8 SHASX SMLABT SMLATB
SMLATT SMLAD SMLALD SMLAWB
SMLAWT SMLSD SMUAD SMULBB
SMULBT SMULTT
SMULTB SMULWT
SMULWB SMUSD
SSAT16 SSAX
SSUB16 SSUB8
SXTAB SXTAB16
SXTAH SXTB16
UADD16 UADD8
UASX UHADD16
UHADD8 UHASX
UHSAX UHSUB16
UHSUB8 UMAAL
UQADD16 UQADD8
UQASX UQSAX
UQSUB16 UQSUB8
USAD8 USADA8
USAT16 USAX
USUB16 USUB8
UXTAB UXTAB16
UXTAH UXTB16

DSP (SIMD, Fast MAC)

Cortex-M4

ADC ADD ADR AND ASR B
CLZ BFC BFI BIC CDP CLREX
CBNZ CBZ CMN CMP DBG EOR LDC
LDMIA LDMDB LDR LDRB LDRBT LDRD
LDREX
LDRSBT
LSR
MRC
ORR PLD PLDW PLI POP PUSH
RBIT REV REV16 REVSH ROR RRX
RSB SBC SBFX
SDIV SEV SMLAL
SMULL SSAT STC
STMIA STMDB STR
STRB STRBT STRD
STREX STREXB STREXH
STRH STRHT STRT
SUB SXTB SXTH
TBB TBH TEQ
TST UBFX UDIV
UMLAL UMULL USAT
UXTB UXTH WFE
WFI YIELD IT

Advanced data processing
Bit field manipulations

Cortex-M3

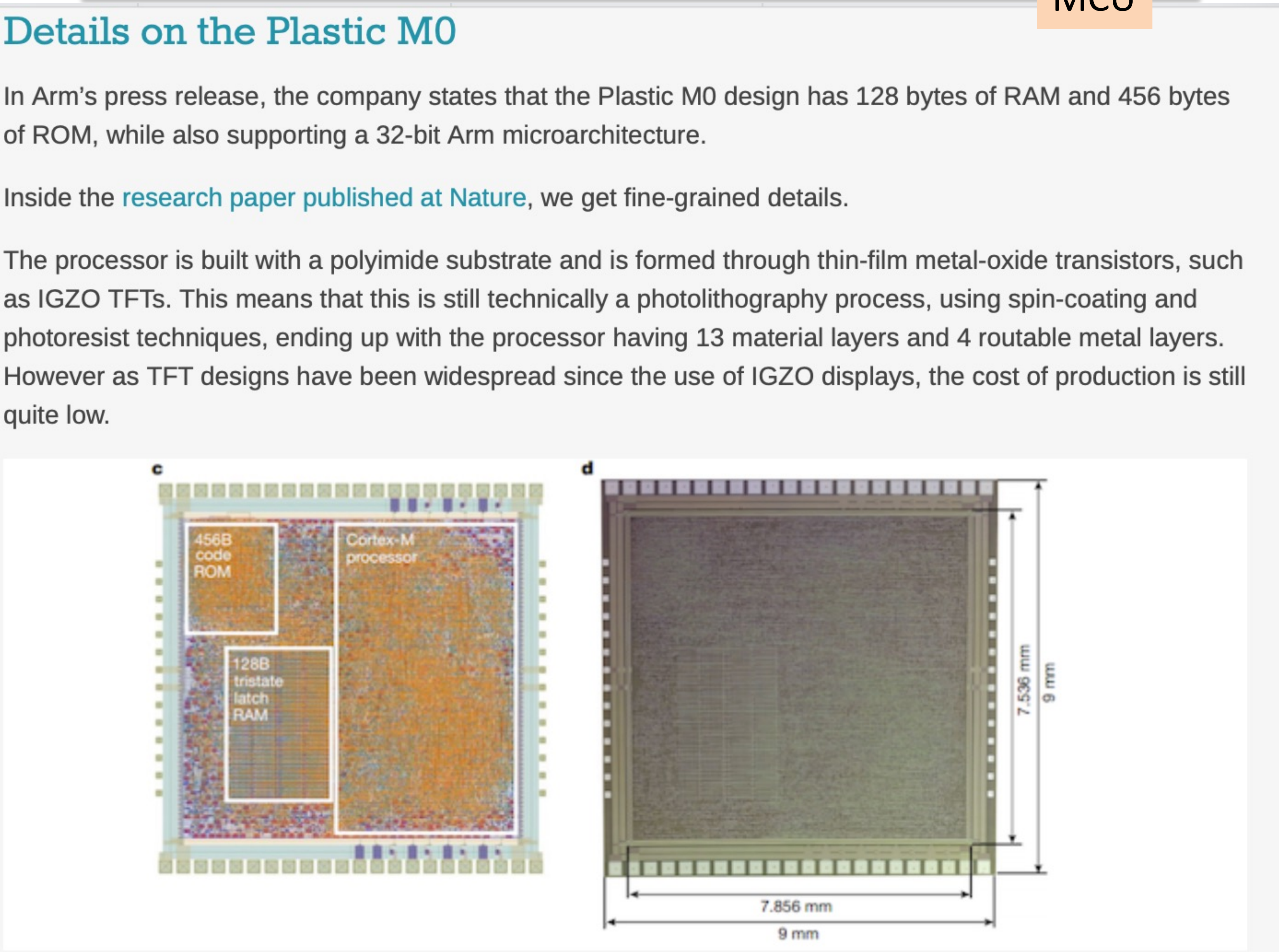
BKPT BLX ADC ADD ADR
BX CPS AND ASR B
DMB BL BIC
DSB CMN CMP EOR
REV16 REVSH POP PUSH ROR
SEV SXTB RSB SBC STM
SXTH UXTB STR STRB STRH
UXTH WFE SUB SVC TST
WFI YIELD

General data processing
I/O control tasks

Cortex-M0/M0+/M1

ARM M Series – M0

MCU



Details on the Plastic M0

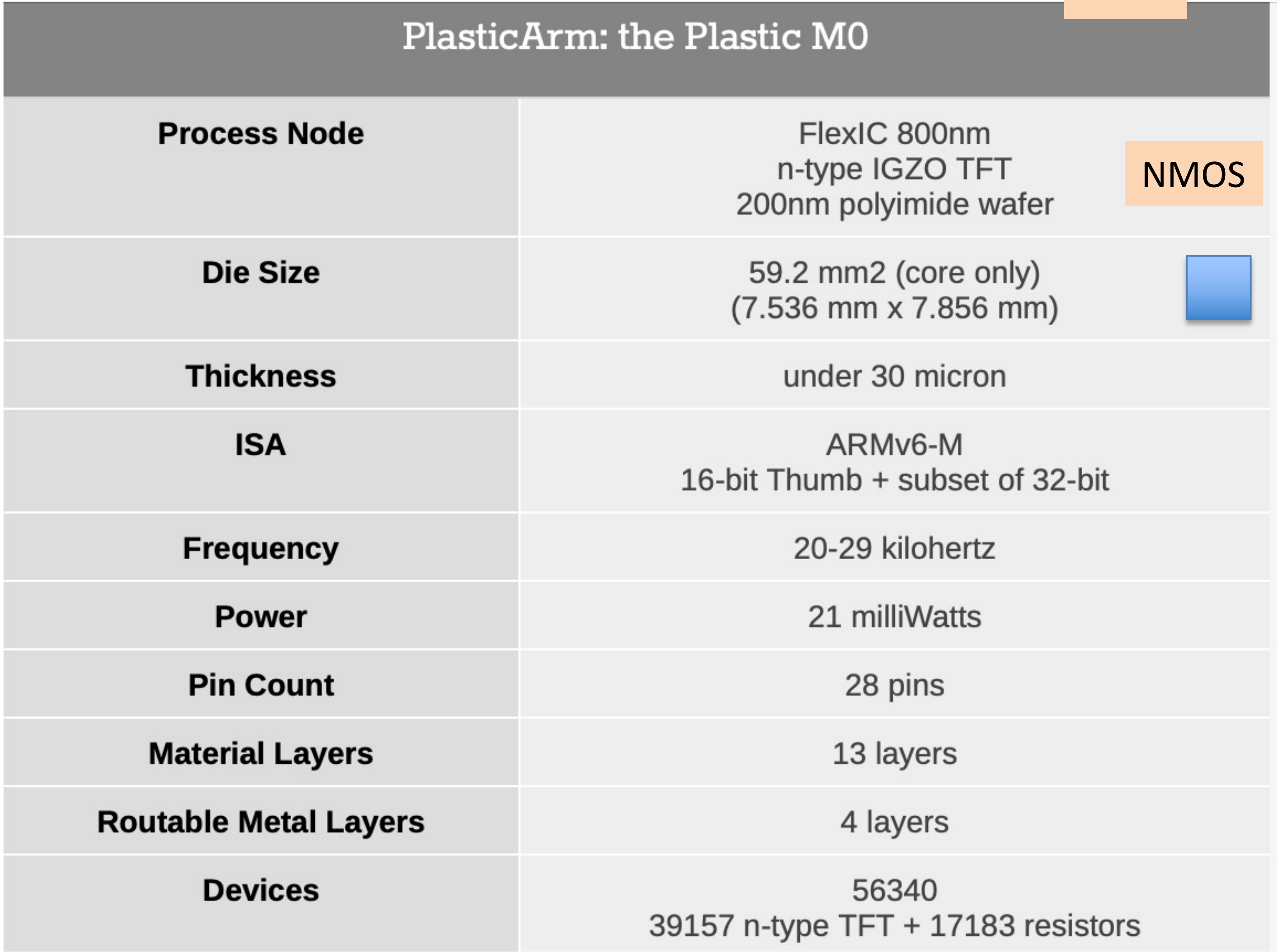
In Arm's press release, the company states that the Plastic M0 design has 128 bytes of RAM and 456 bytes of ROM, while also supporting a 32-bit Arm microarchitecture.

Inside the research paper published at Nature, we get fine-grained details.

The processor is built with a polyimide substrate and is formed through thin-film metal-oxide transistors, such as IGZO TFTs. This means that this is still technically a photolithography process, using spin-coating and photoresist techniques, ending up with the processor having 13 material layers and 4 routable metal layers. However as TFT designs have been widespread since the use of IGZO displays, the cost of production is still quite low.

ARM M Series

MCU

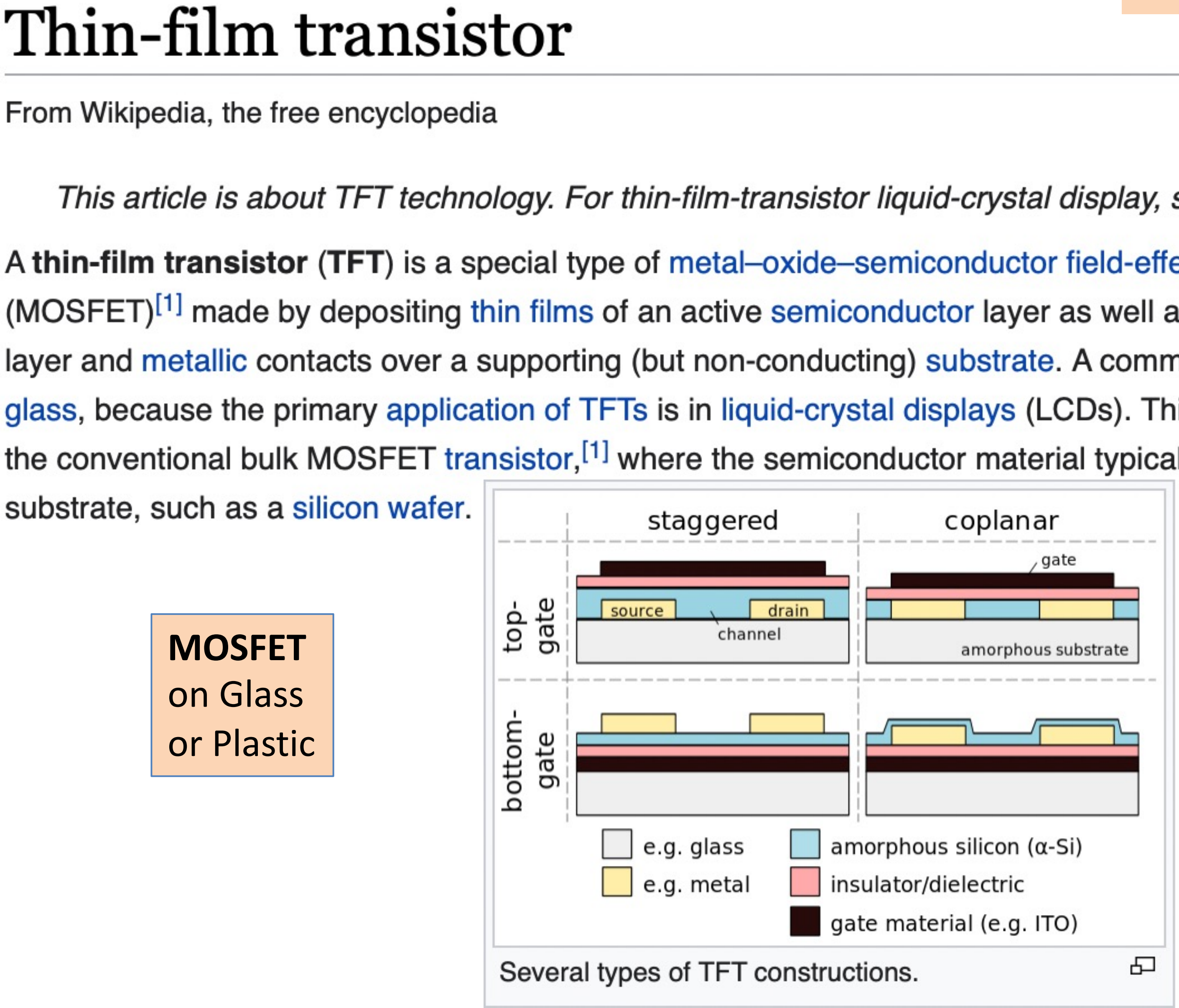


PlasticArm: the Plastic M0	
Process Node	FlexIC 800nm n-type IGZO TFT 200nm polyimide wafer
Die Size	59.2 mm2 (core only) (7.536 mm x 7.856 mm)
Thickness	under 30 micron
ISA	ARMv6-M 16-bit Thumb + subset of 32-bit
Frequency	20-29 kilohertz
Power	21 milliWatts
Pin Count	28 pins
Material Layers	13 layers
Routable Metal Layers	4 layers
Devices	56340 39157 n-type TFT + 17183 resistors

NMOS

ARM M Series

MCU



Thin-film transistor

From Wikipedia, the free encyclopedia

This article is about TFT technology. For thin-film-transistor liquid-crystal display, see TFT LCD.

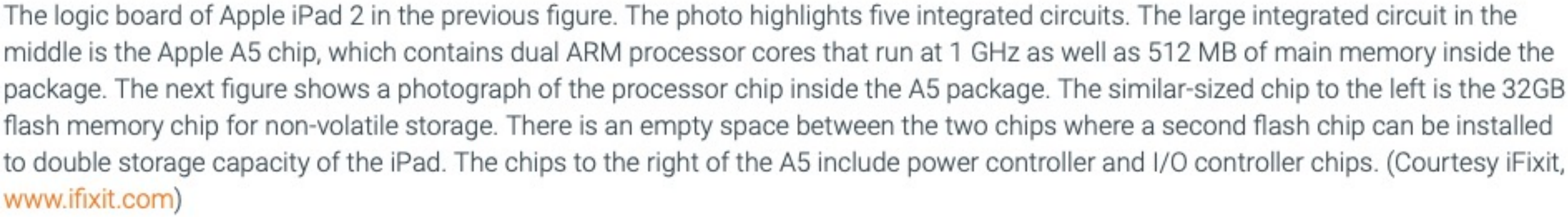
A **thin-film transistor** (**TFT**) is a special type of metal–oxide–semiconductor field-effect transistor (MOSFET)[1] made by depositing thin films of an active semiconductor layer as well as the dielectric layer and metallic contacts over a supporting (but non-conducting) substrate. A common substrate is glass, because the primary application of TFTs is in liquid-crystal displays (LCDs). This differs from the conventional bulk MOSFET transistor,[1] where the semiconductor material typically *is* the substrate, such as a silicon wafer.

Several types of TFT constructions.

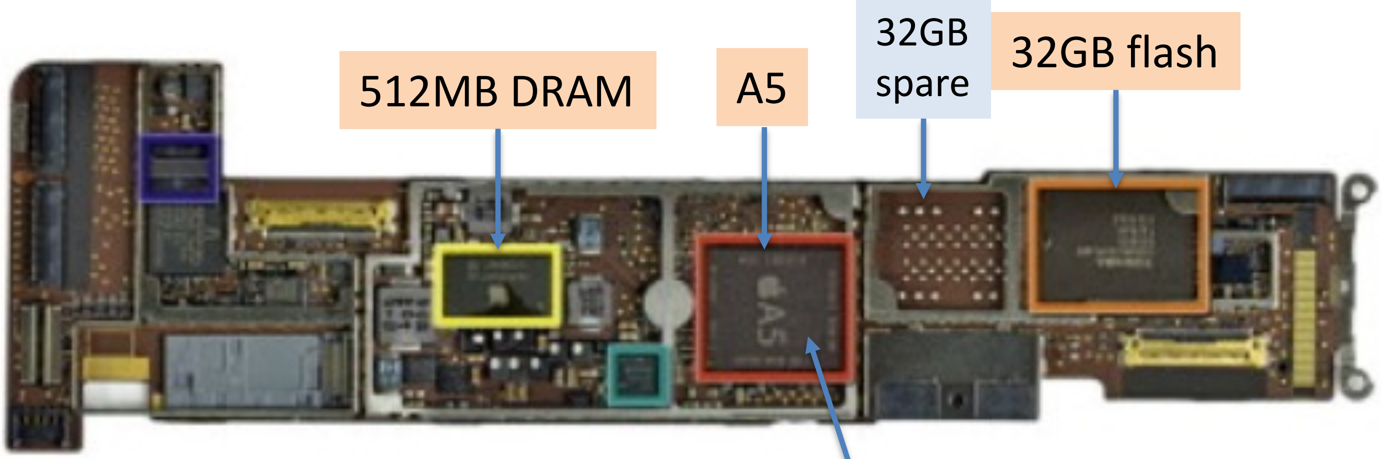
MOSFET on Glass or Plastic

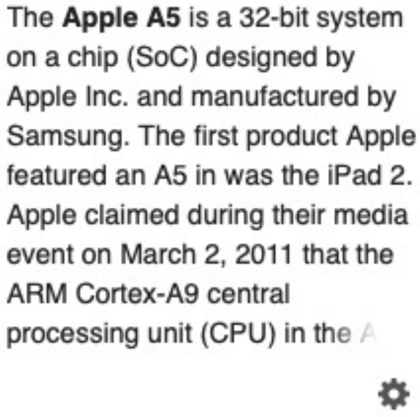
A5 CPU on iPad2

Hennessy & Patterson



The logic board of Apple iPad 2 in the previous figure. The photo highlights five integrated circuits. The large integrated circuit in the middle is the Apple A5 chip, which contains dual ARM processor cores that run at 1 GHz as well as 512 MB of main memory inside the package. The next figure shows a photograph of the processor chip inside the A5 package. The similar-sized chip to the left is the 32GB flash memory chip for non-volatile storage. There is an empty space between the two chips where a second flash chip can be installed to double storage capacity of the iPad. The chips to the right of the A5 include power controller and I/O controller chips. (Courtesy iFixit, www.ifixit.com)

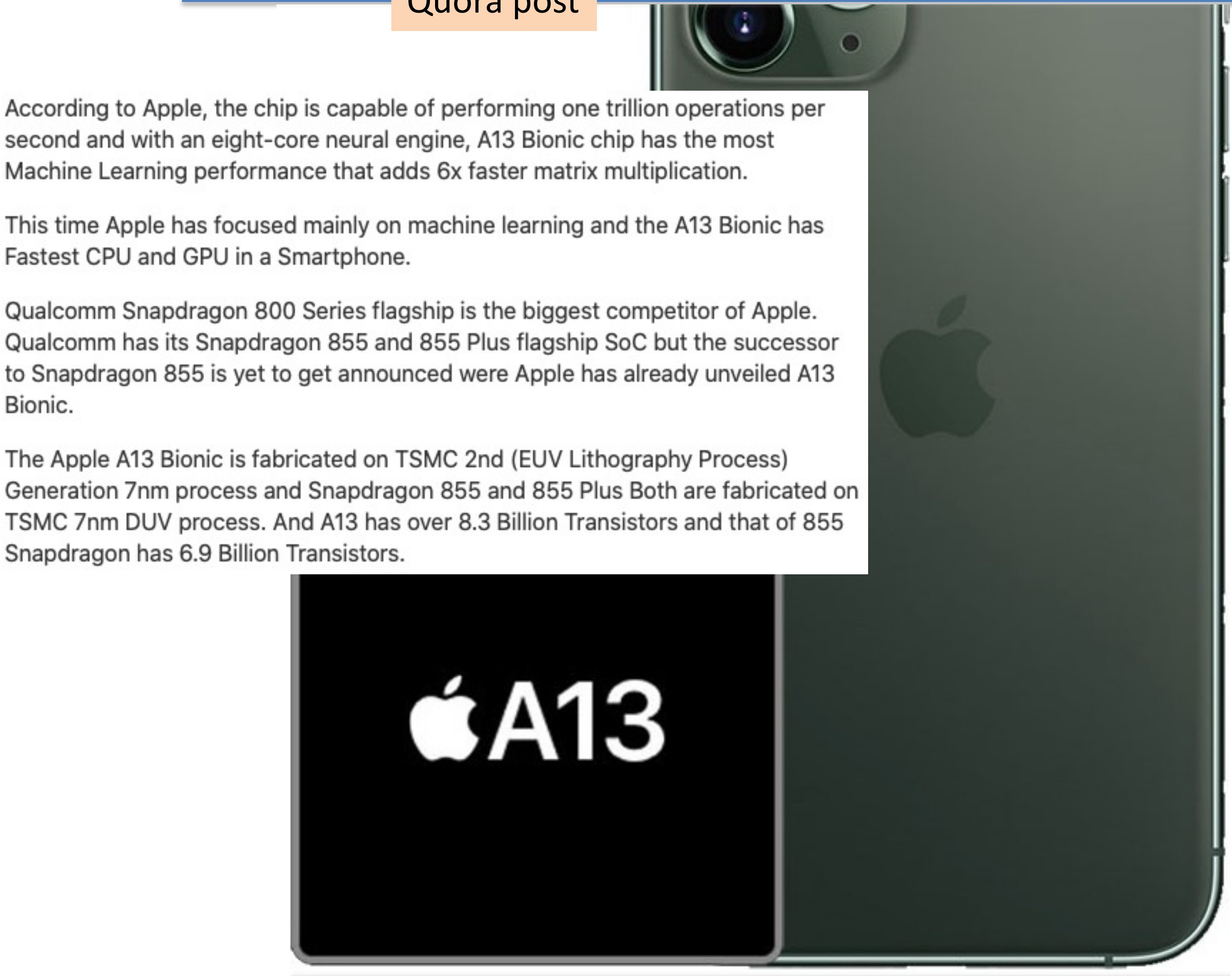




The **Apple A5** is a 32-bit system on a chip (SoC) designed by Apple Inc. and manufactured by Samsung. The first product Apple featured an A5 in was the iPad 2. Apple claimed during their media event on March 2, 2011 that the ARM Cortex-A9 central processing unit (CPU) in the A

Apple ARM A Series

Quora post



According to Apple, the chip is capable of performing one trillion operations per second and with an eight-core neural engine, A13 Bionic chip has the most Machine Learning performance that adds 6x faster matrix multiplication.

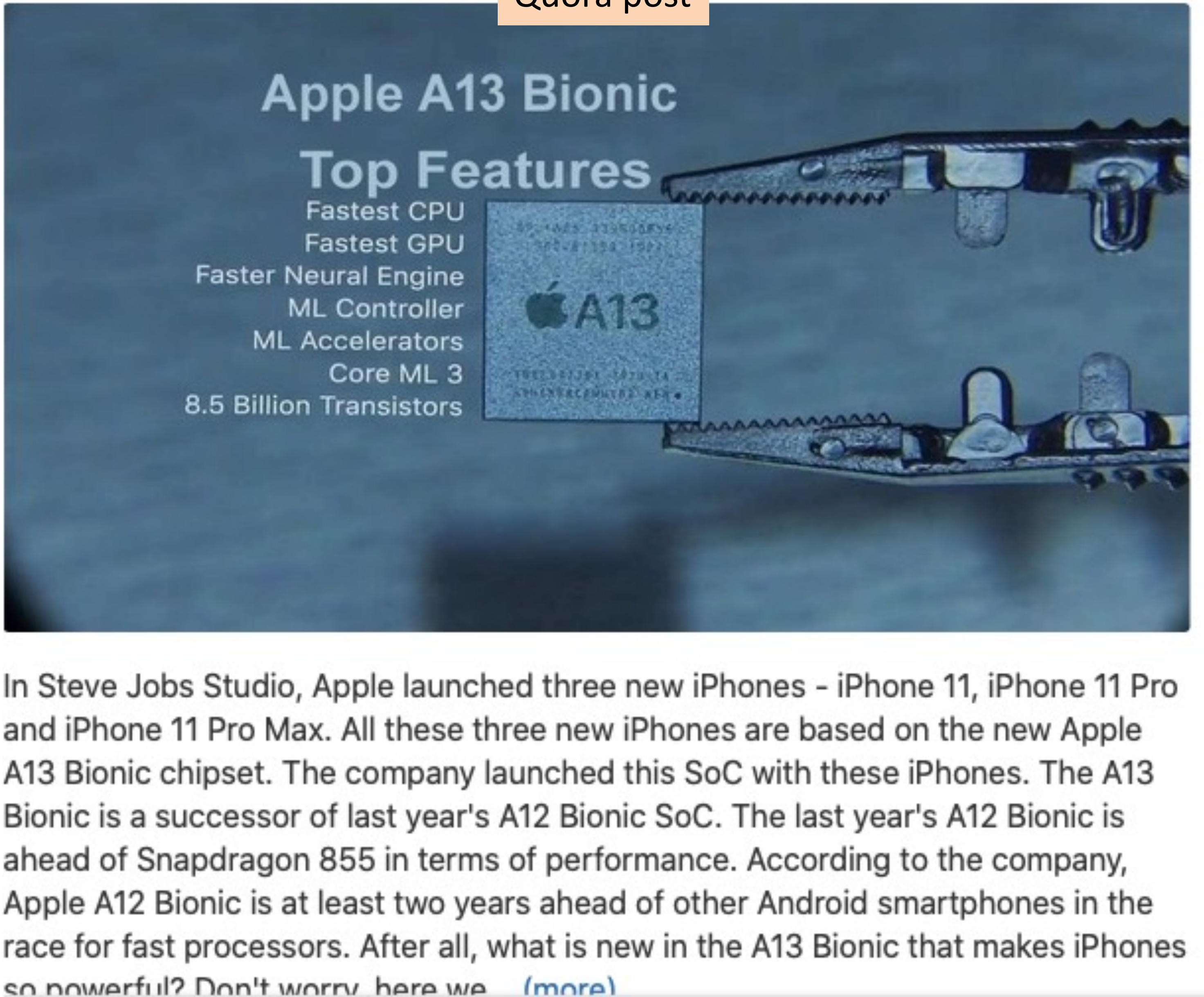
This time Apple has focused mainly on machine learning and the A13 Bionic has Fastest CPU and GPU in a Smartphone.

Qualcomm Snapdragon 800 Series flagship is the biggest competitor of Apple. Qualcomm has its Snapdragon 855 and 855 Plus flagship SoC but the successor to Snapdragon 855 is yet to get announced were Apple has already unveiled A13 Bionic.

The Apple A13 Bionic is fabricated on TSMC 2nd (EUV Lithography Process) Generation 7nm process and Snapdragon 855 and 855 Plus Both are fabricated on TSMC 7nm DUV process. And A13 has over 8.3 Billion Transistors and that of 855 Snapdragon has 6.9 Billion Transistors.

Apple A13

Quora post



Apple A13 Bionic
Top Features
Fastest CPU
Fastest GPU
Faster Neural Engine
ML Controller
ML Accelerators
Core ML 3
8.5 Billion Transistors

In Steve Jobs Studio, Apple launched three new iPhones - iPhone 11, iPhone 11 Pro and iPhone 11 Pro Max. All these three new iPhones are based on the new Apple A13 Bionic chipset. The company launched this SoC with these iPhones. The A13 Bionic is a successor of last year's A12 Bionic SoC. The last year's A12 Bionic is ahead of Snapdragon 855 in terms of performance. According to the company, Apple A12 Bionic is at least two years ahead of other Android smartphones in the race for fast processors. After all, what is new in the A13 Bionic that makes iPhones so powerful? Don't worry, here we… (more)

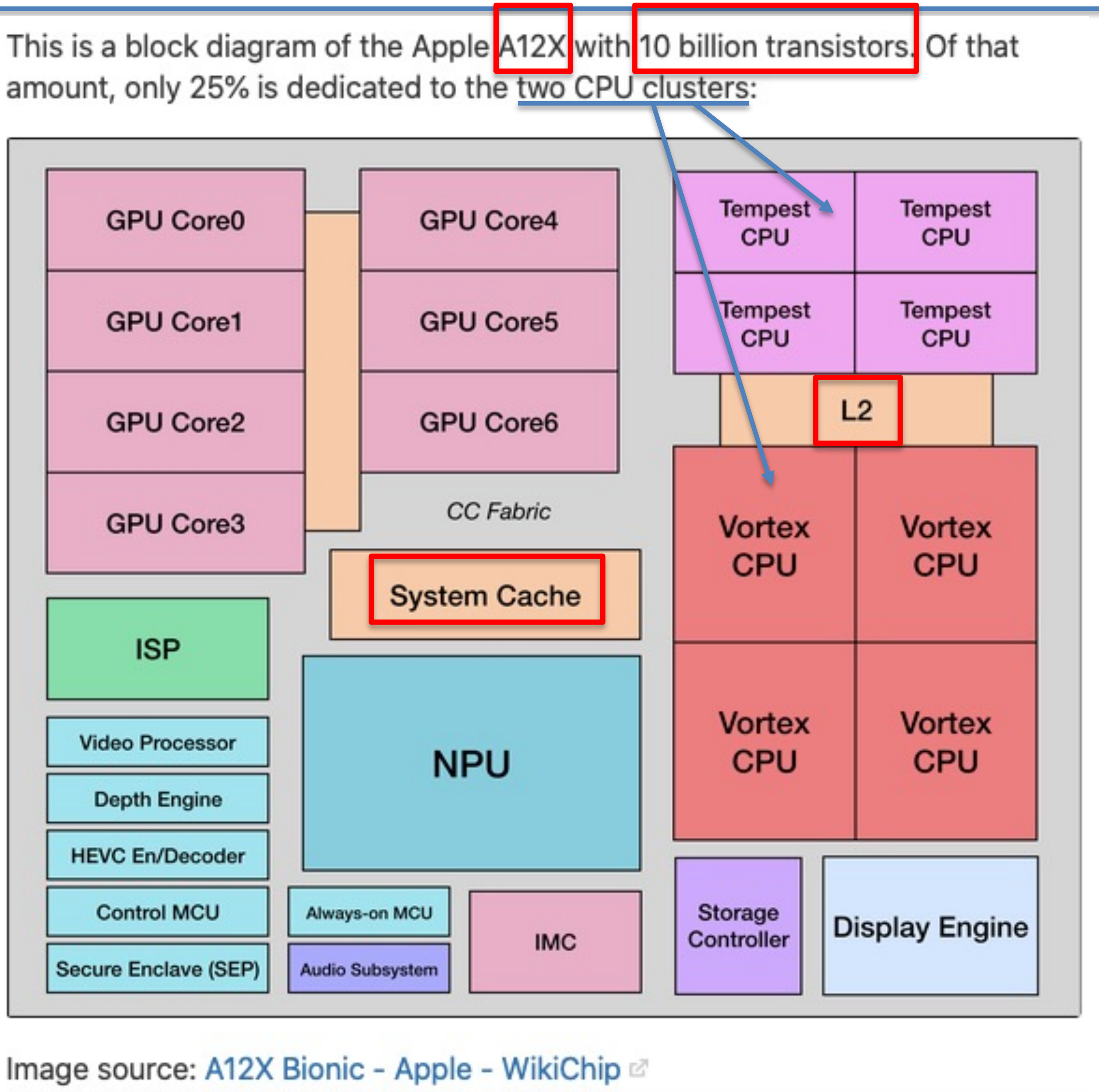
Apple A13

The A13 is a new multi-core architecture designed by Apple with **8.5B** transistors manufactured at TSMC (**7nm** EUV) -- extremely state-of-the-art. It contains a large number of **ARM** ISA cores:
8 CPU + 7 GPU + 8 NPU + 2 MCU. All cores are Apple designed (ARM 64-bit v8 ISA is licensed).

It includes a ***Neural engine*** (8x NPU) with machine learning (core ML 3 at 6x faster matrix multiply) – which sets it apart from Intel chips without GPU's or an NPU. The GPU's can perform 1 trillion operations per second (1 Tflops=1000 Gflops), and the NPU may hit 5 Tflops. It has extreme power management as well (so good for portables and mobile).

Apple ARM A12/13 SoC

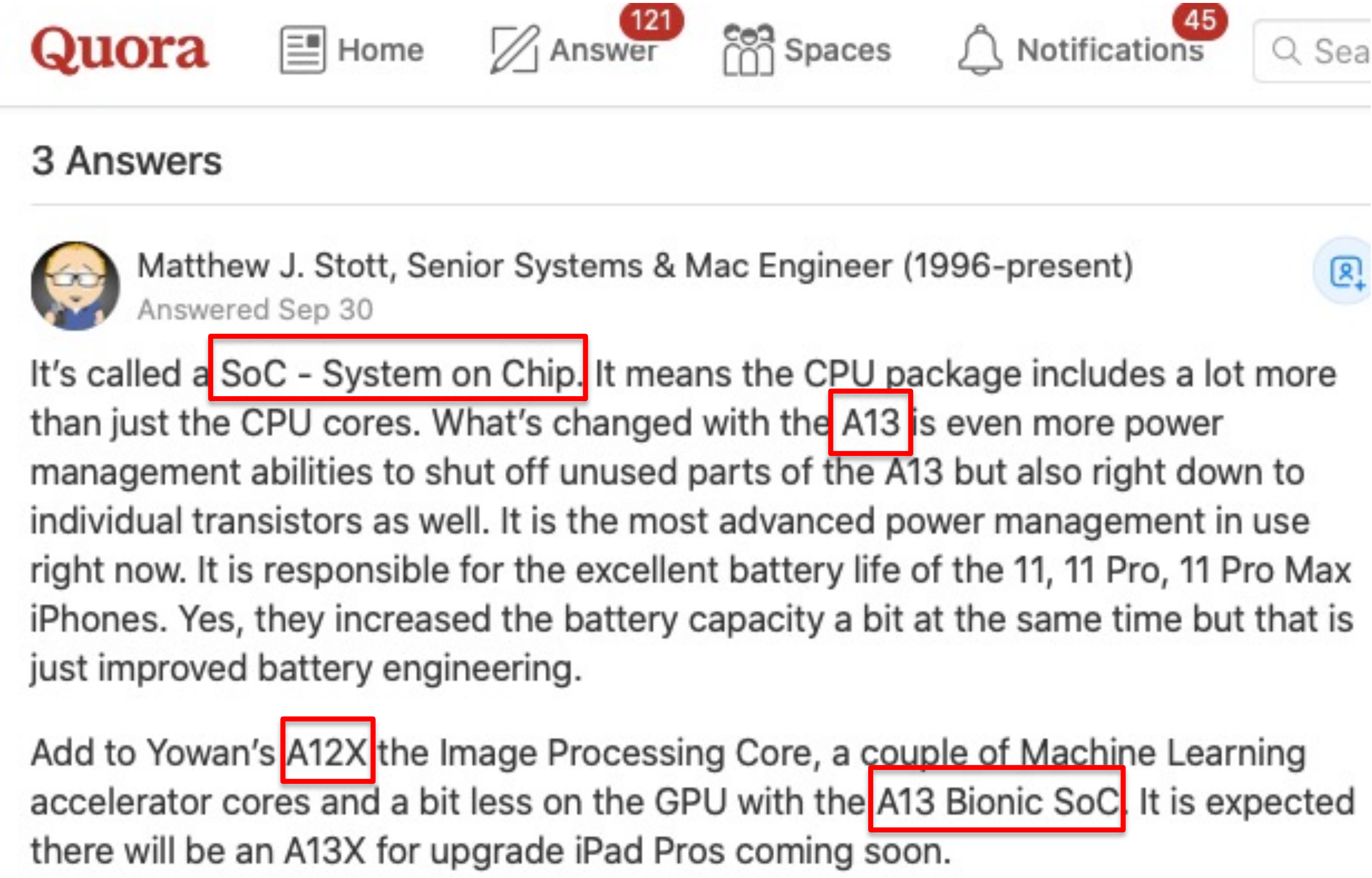
- 8 CPU
- 7 GPU
- 1 NPU
- 2 MCU



This is a block diagram of the Apple A12X with 10 billion transistors. Of that amount, only 25% is dedicated to the two CPU clusters:

Image source: A12X Bionic - Apple - WikiChip

Apple ARM A12/13 SoC



3 Answers

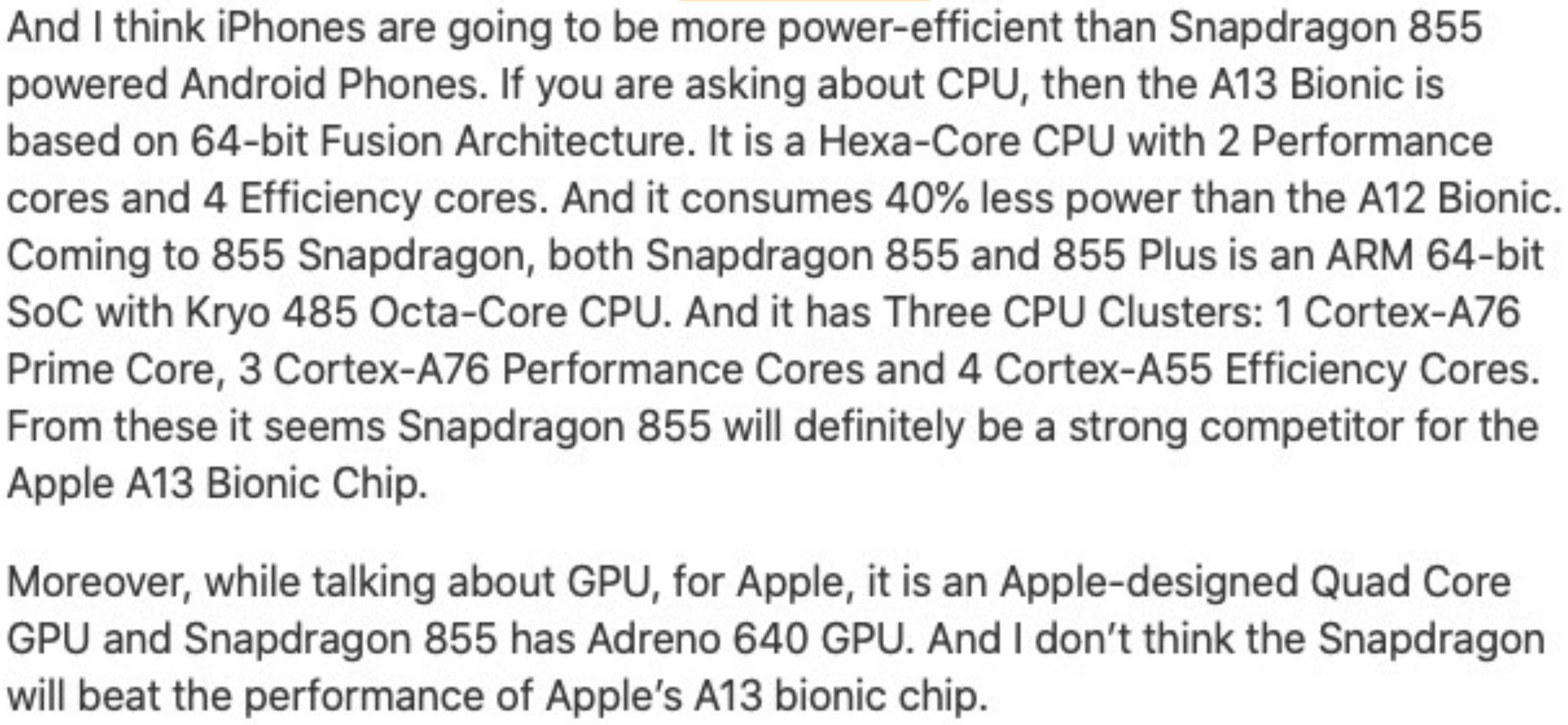
Matthew J. Stott, Senior Systems & Mac Engineer (1996-present)
Answered Sep 30

It's called a SoC - System on Chip. It means the CPU package includes a lot more than just the CPU cores. What's changed with the A13 is even more power management abilities to shut off unused parts of the A13 but also right down to individual transistors as well. It is the most advanced power management in use right now. It is responsible for the excellent battery life of the 11, 11 Pro, 11 Pro Max iPhones. Yes, they increased the battery capacity a bit at the same time but that is just improved battery engineering.

Add to Yowan's A12X the Image Processing Core, a couple of Machine Learning accelerator cores and a bit less on the GPU with the A13 Bionic SoC. It is expected there will be an A13X for upgrade iPad Pros coming soon.

Apple A13 v Snapdragon 855

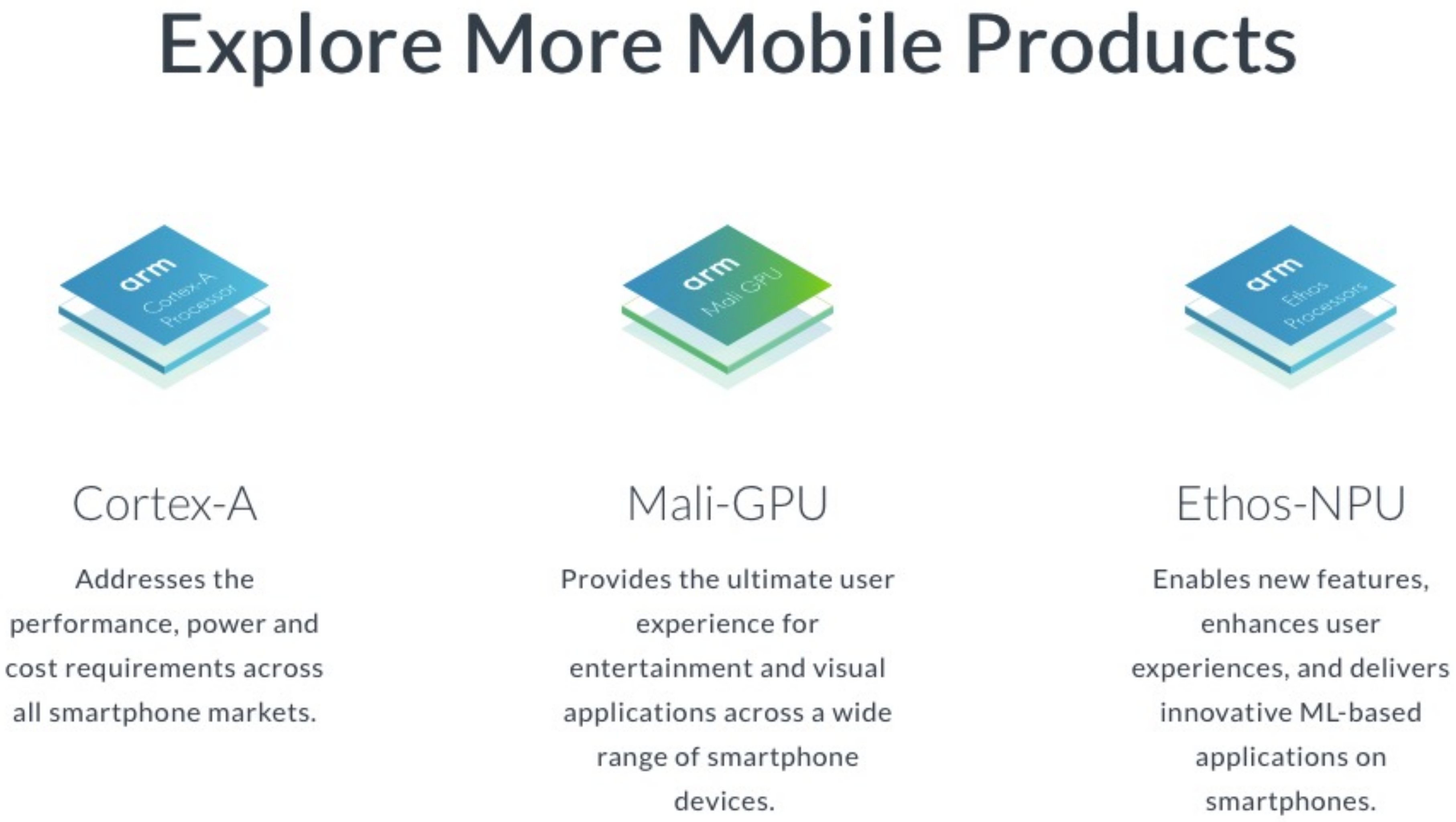
Quora post



And I think iPhones are going to be more power-efficient than Snapdragon 855 powered Android Phones. If you are asking about CPU, then the A13 Bionic is based on 64-bit Fusion Architecture. It is a Hexa-Core CPU with 2 Performance cores and 4 Efficiency cores. And it consumes 40% less power than the A12 Bionic. Coming to 855 Snapdragon, both Snapdragon 855 and 855 Plus is an ARM 64-bit SoC with Kryo 485 Octa-Core CPU. And it has Three CPU Clusters: 1 Cortex-A76 Prime Core, 3 Cortex-A76 Performance Cores and 4 Cortex-A55 Efficiency Cores. From these it seems Snapdragon 855 will definitely be a strong competitor for the Apple A13 Bionic Chip.

Moreover, while talking about GPU, for Apple, it is an Apple-designed Quad Core GPU and Snapdragon 855 has Adreno 640 GPU. And I don't think the Snapdragon will beat the performance of Apple's A13 bionic chip.

New ARM Models



Explore More Mobile Products

Cortex-A

Addresses the performance, power and cost requirements across all smartphone markets.

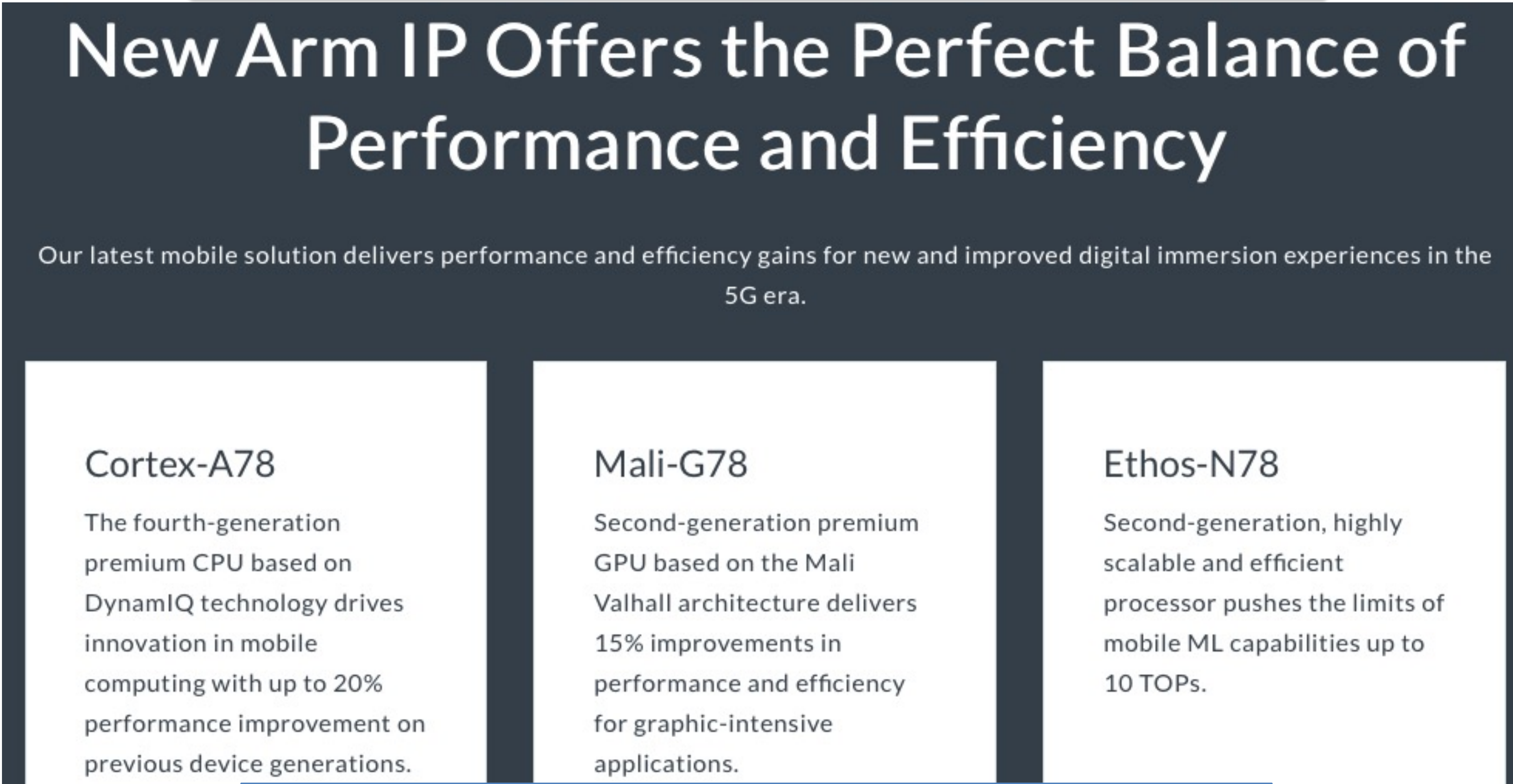
Mali-GPU

Provides the ultimate user experience for entertainment and visual applications across a wide range of smartphone devices.

Ethos-NPU

Enables new features, enhances user experiences, and delivers innovative ML-based applications on smartphones.

New ARM Models



New Arm IP Offers the Perfect Balance of Performance and Efficiency

Our latest mobile solution delivers performance and efficiency gains for new and improved digital immersion experiences in the 5G era.

Cortex-A78

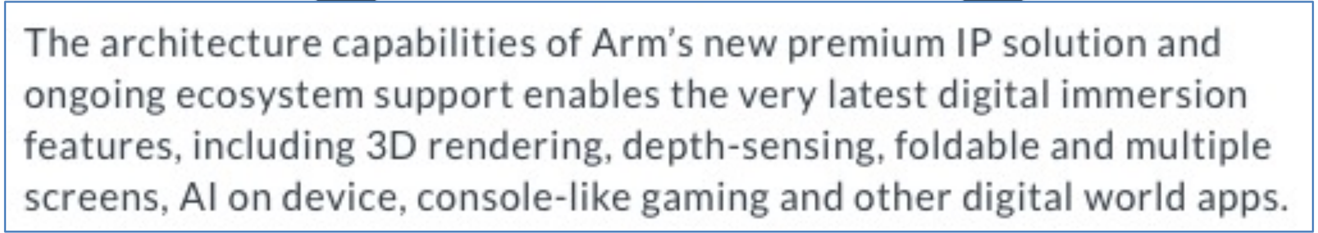
The fourth-generation premium CPU based on DynamIQ technology drives innovation in mobile computing with up to 20% performance improvement on previous device generations.

Mali-G78

Second-generation premium GPU based on the Mali Valhall architecture delivers 15% improvements in performance and efficiency for graphic-intensive applications.

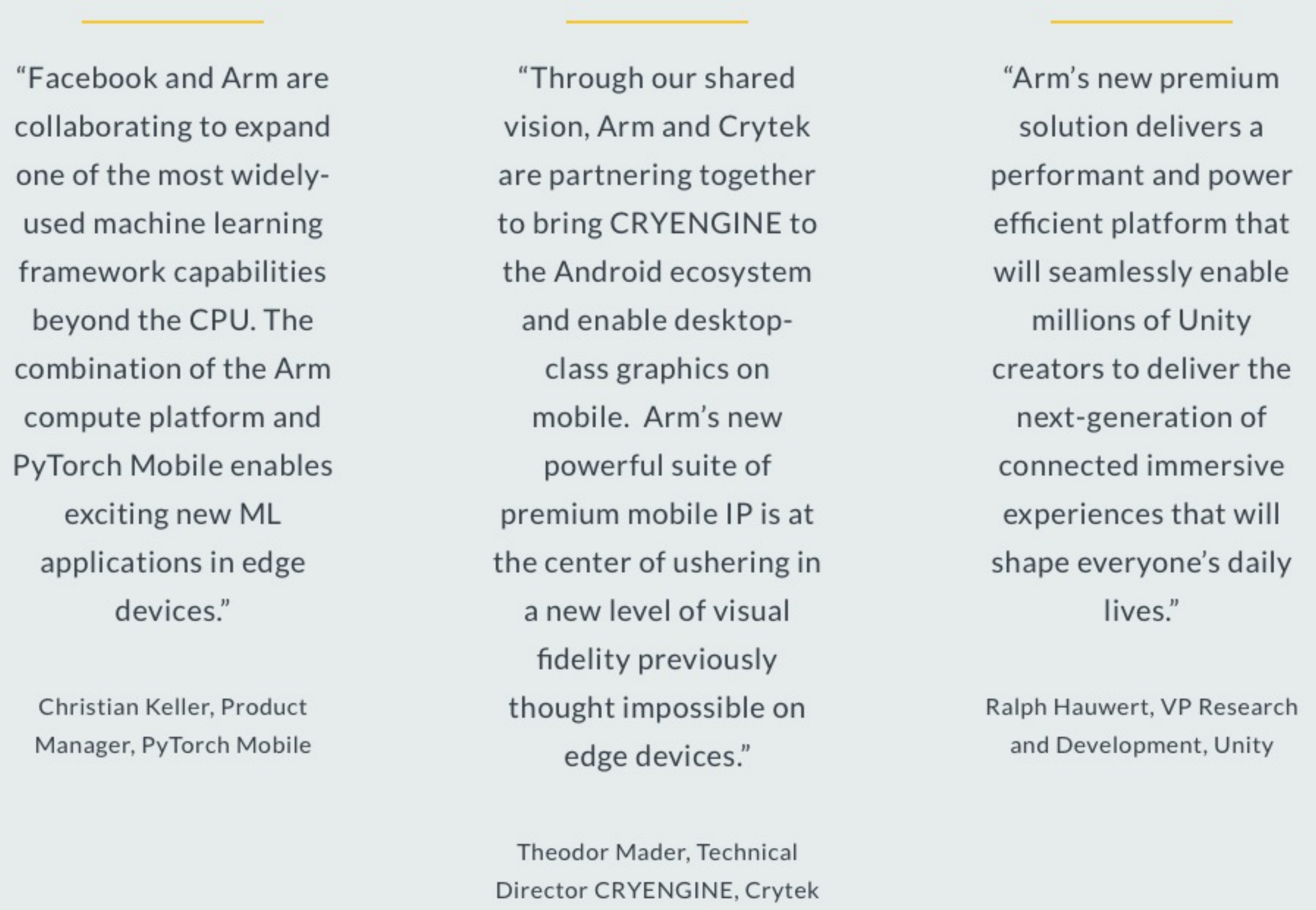
Ethos-N78

Second-generation, highly scalable and efficient processor pushes the limits of mobile ML capabilities up to 10 TOPs.



The architecture capabilities of Arm's new premium IP solution and ongoing ecosystem support enables the very latest digital immersion features, including 3D rendering, depth-sensing, foldable and multiple screens, AI on device, console-like gaming and other digital world apps.

New ARM Models



"Facebook and Arm are collaborating to expand one of the most widely-used machine learning framework capabilities beyond the CPU. The combination of the Arm compute platform and PyTorch Mobile enables exciting new ML applications in edge devices."

Christian Keller, Product Manager, PyTorch Mobile

"Through our shared vision, Arm and Crytek are partnering together to bring CRYENGINE to the Android ecosystem and enable desktop-class graphics on mobile. Arm's new powerful suite of premium mobile IP is at the center of ushering in a new level of visual fidelity previously thought impossible on edge devices."

Theodor Mader, Technical Director CRYENGINE, Crytek

"Arm's new premium solution delivers a performant and power efficient platform that will seamlessly enable millions of Unity creators to deliver the next-generation of connected immersive experiences that will shape everyone's daily lives."

Ralph Hauwert, VP Research and Development, Unity

ARM u-Arch

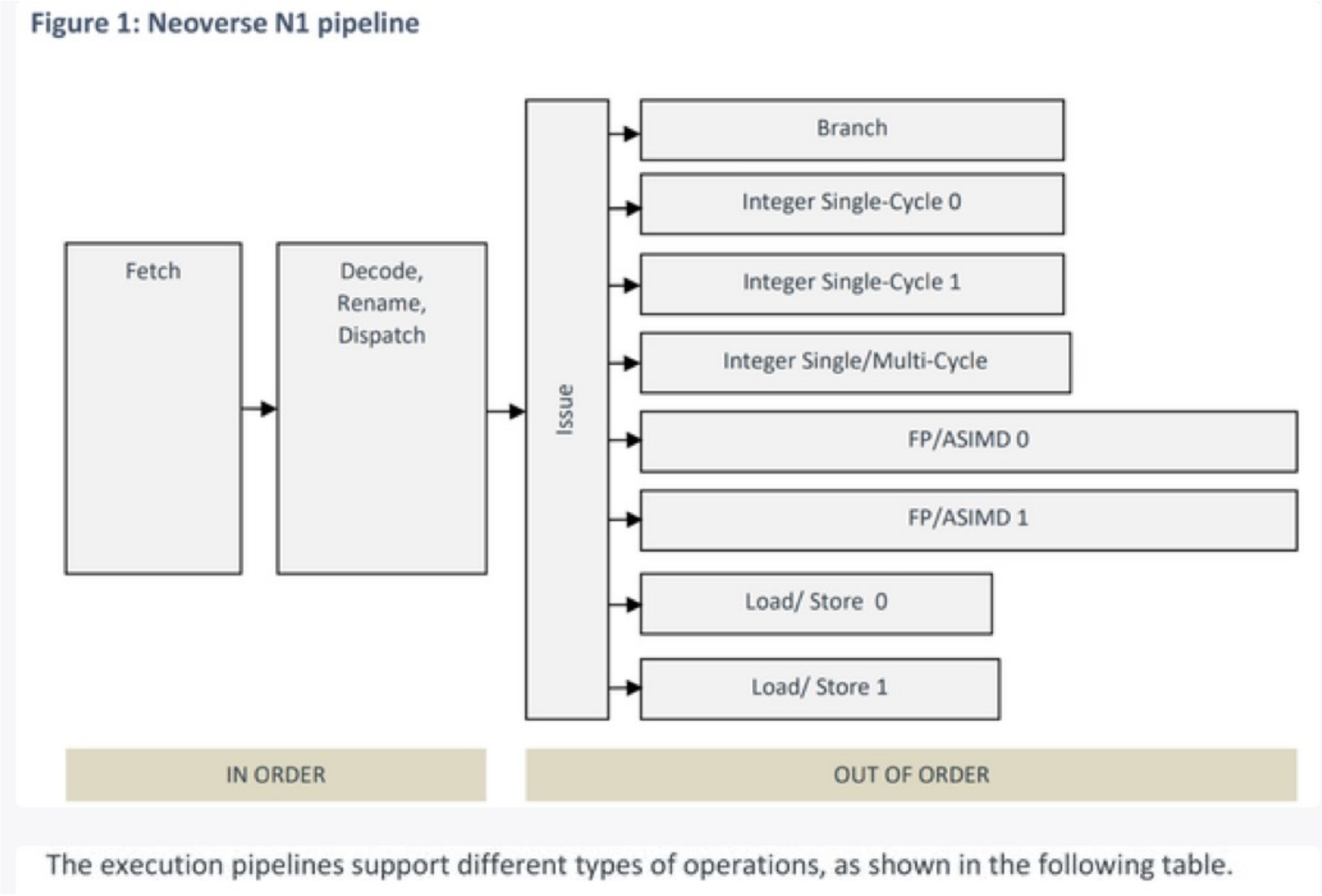


Figure 1: Neoverse N1 pipeline

The execution pipelines support different types of operations, as shown in the following table.

ARM u-Arch

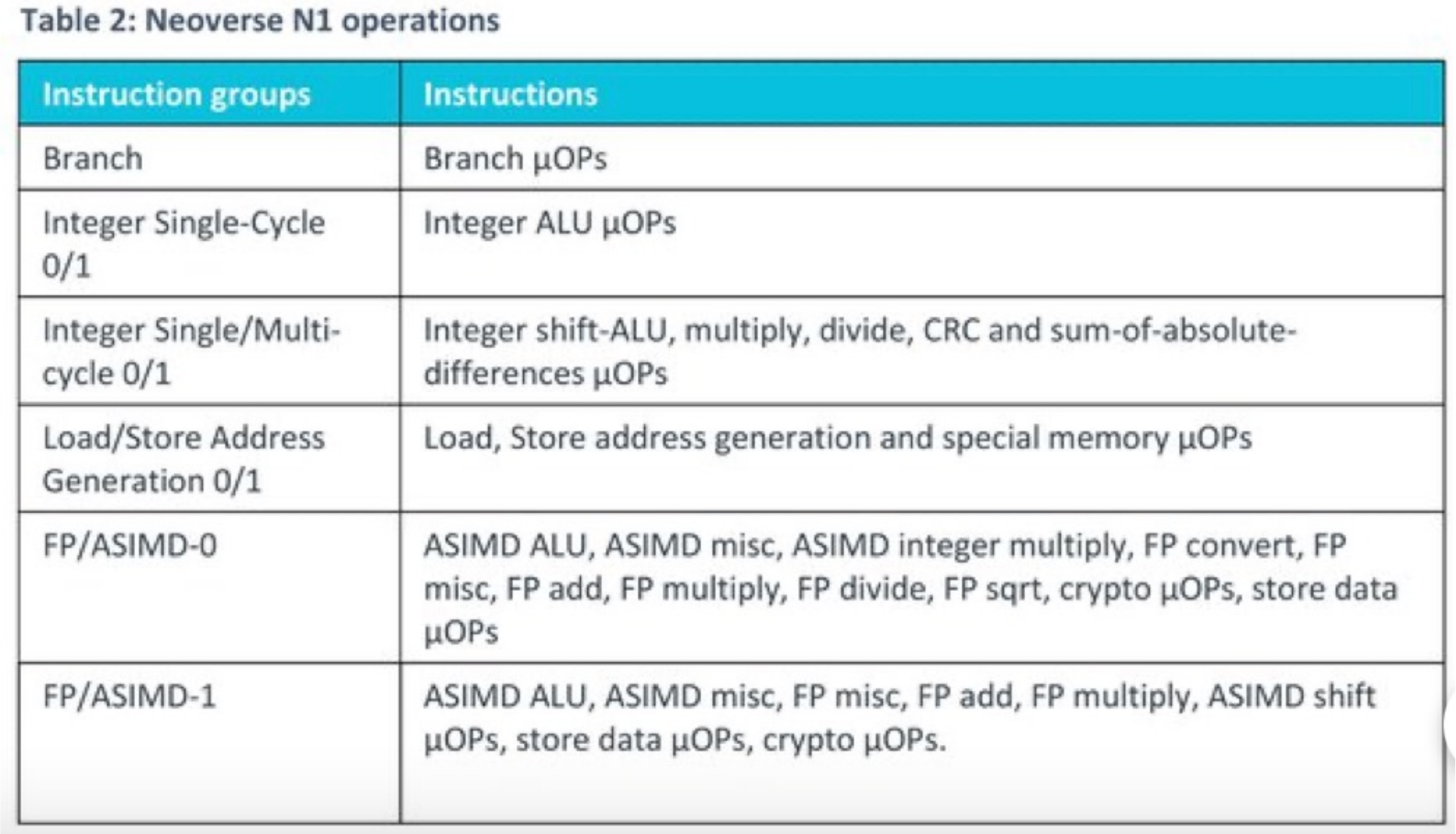
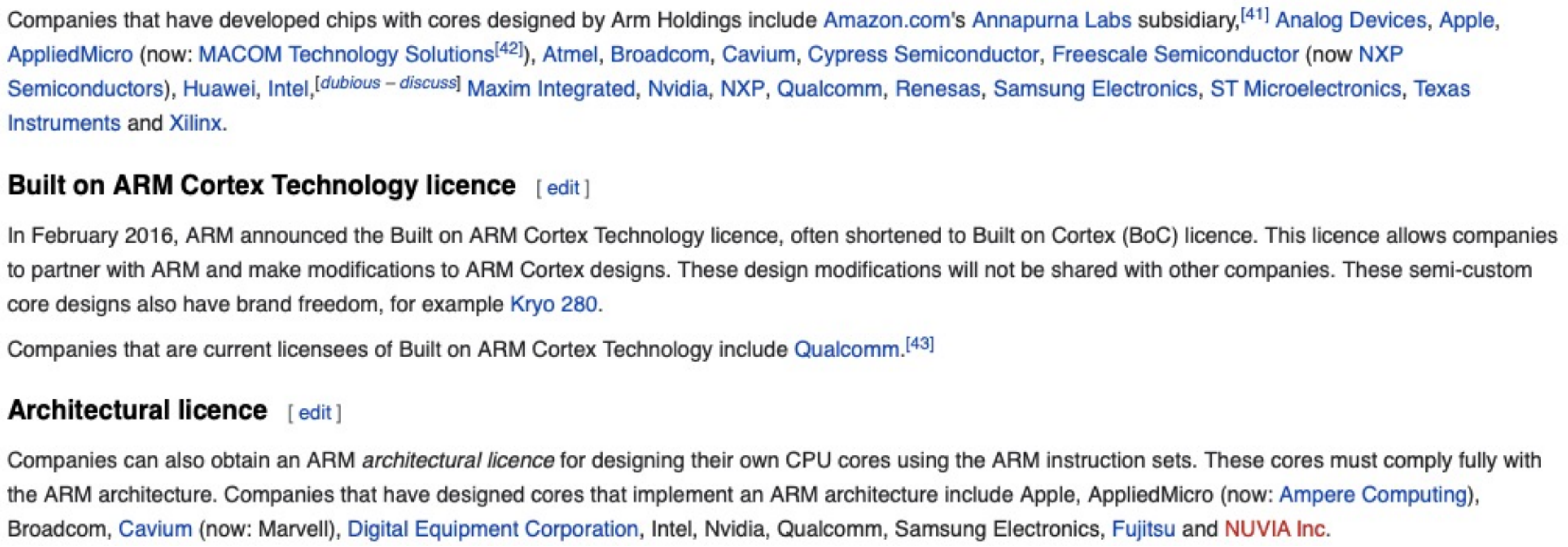


Table 2: Neoverse N1 operations

Instruction groups	Instructions
Branch	Branch µOPs
Integer Single-Cycle 0/1	Integer ALU µOPs
Integer Single/Multi-cycle 0/1	Integer shift-ALU, multiply, divide, CRC and sum-of-absolute-differences µOPs
Load/Store Address Generation 0/1	Load, Store address generation and special memory µOPs
FP/ASIMD-0	ASIMD ALU, ASIMD misc, ASIMD integer multiply, FP convert, FP misc, FP add, FP multiply, FP divide, FP sqrt, crypto µOPs, store data µOPs
FP/ASIMD-1	ASIMD ALU, ASIMD misc, FP misc, FP add, FP multiply, ASIMD shift µOPs, store data µOPs, crypto µOPs.

ARM Licensing



Companies that have developed chips with cores designed by Arm Holdings include Amazon.com's Annapurna Labs subsidiary,[41] Analog Devices, Apple, AppliedMicro (now: MACOM Technology Solutions[42]), Atmel, Broadcom, Cavium, Cypress Semiconductor, Freescale Semiconductor (now NXP Semiconductors), Huawei, Intel,[dubious – discuss] Maxim Integrated, Nvidia, NXP, Qualcomm, Renesas, Samsung Electronics, ST Microelectronics, Texas Instruments and Xilinx.

Built on ARM Cortex Technology licence [edit]

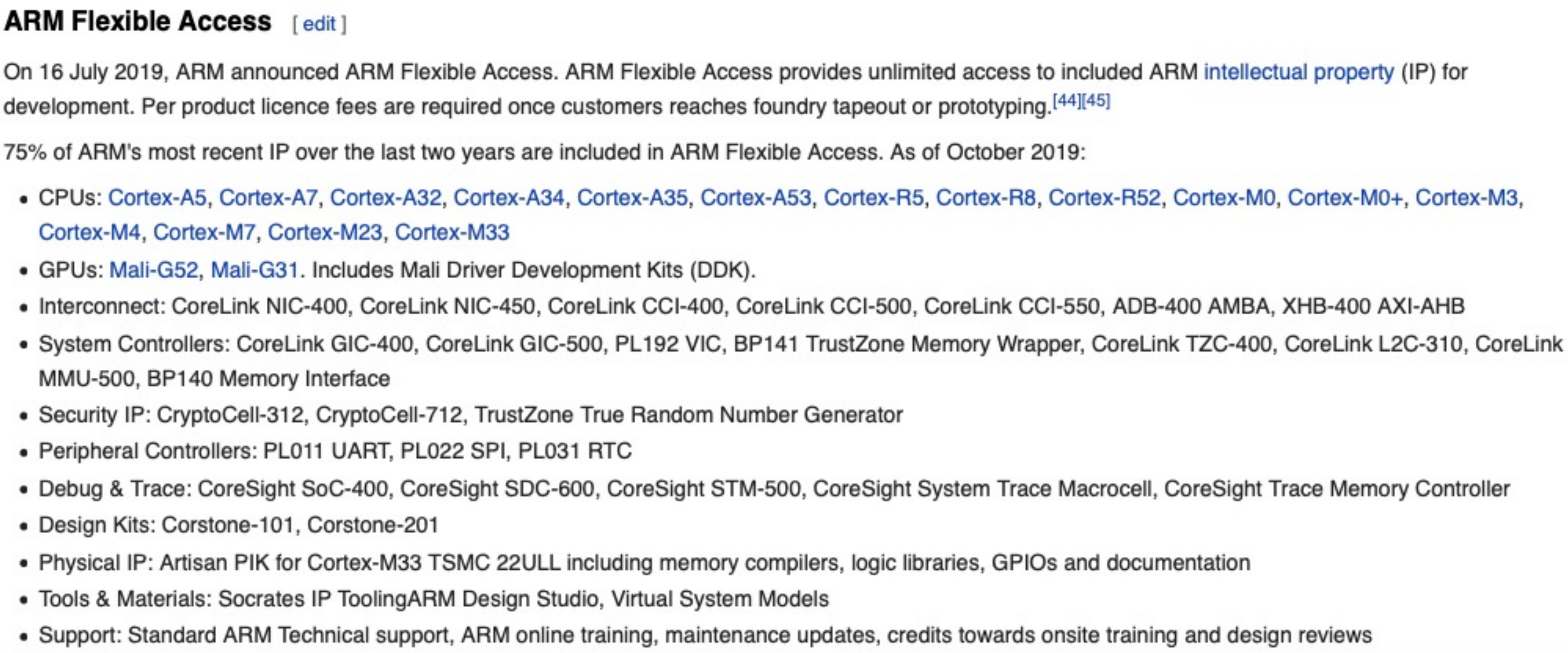
In February 2016, ARM announced the Built on ARM Cortex Technology licence, often shortened to Built on Cortex (BoC) licence. This licence allows companies to partner with ARM and make modifications to ARM Cortex designs. These design modifications will not be shared with other companies. These semi-custom core designs also have brand freedom, for example Kryo 280.

Companies that are current licensees of Built on ARM Cortex Technology include Qualcomm.[43]

Architectural licence [edit]

Companies can also obtain an ARM *architectural licence* for designing their own CPU cores using the ARM instruction sets. These cores must comply fully with the ARM architecture. Companies that have designed cores that implement an ARM architecture include Apple, AppliedMicro (now: Ampere Computing), Broadcom, Cavium (now: Marvell), Digital Equipment Corporation, Intel, Nvidia, Qualcomm, Samsung Electronics, Fujitsu and NUVIA Inc.

ARM Licensing



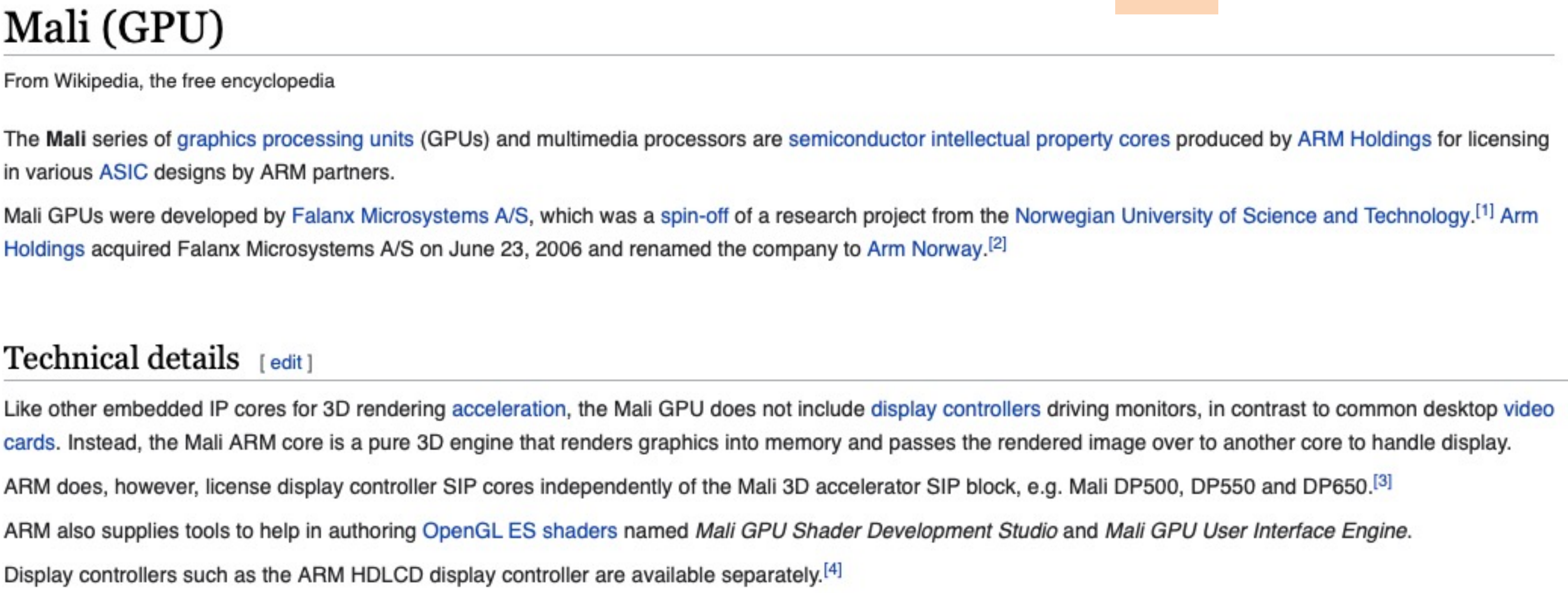
ARM Flexible Access [edit]

On 16 July 2019, ARM announced ARM Flexible Access. ARM Flexible Access provides unlimited access to included ARM intellectual property (IP) for development. Per product licence fees are required once customers reaches foundry tapeout or prototyping.[44][45]

75% of ARM's most recent IP over the last two years are included in ARM Flexible Access. As of October 2019:

- CPUs: Cortex-A5, Cortex-A7, Cortex-A32, Cortex-A34, Cortex-A35, Cortex-A53, Cortex-R5, Cortex-R8, Cortex-R52, Cortex-M0, Cortex-M0+, Cortex-M3, Cortex-M4, Cortex-M7, Cortex-M23, Cortex-M33
- GPUs: Mali-G52, Mali-G31. Includes Mali Driver Development Kits (DDK).
- Interconnect: CoreLink NIC-400, CoreLink NIC-450, CoreLink CCI-400, CoreLink CCI-500, CoreLink CCI-550, ADB-400 AMBA, XHB-400 AXI-AHB
- System Controllers: CoreLink GIC-400, CoreLink GIC-500, PL192 VIC, BP141 TrustZone Memory Wrapper, CoreLink TZC-400, CoreLink L2C-310, CoreLink MMU-500, BP140 Memory Interface
- Security IP: CryptoCell-312, CryptoCell-712, TrustZone True Random Number Generator
- Peripheral Controllers: PL011 UART, PL022 SPI, PL031 RTC
- Debug & Trace: CoreSight SoC-400, CoreSight SDC-600, CoreSight STM-500, CoreSight System Trace Macrocell, CoreSight Trace Memory Controller
- Design Kits: Corstone-101, Corstone-201
- Physical IP: Artisan PIK for Cortex-M33 TSMC 22ULL including memory compilers, logic libraries, GPIOs and documentation
- Tools & Materials: Socrates IP ToolingARM Design Studio, Virtual System Models
- Support: Standard ARM Technical support, ARM online training, maintenance updates, credits towards onsite training and design reviews

ARM Mali GPU



Mali (GPU)

From Wikipedia, the free encyclopedia

The **Mali** series of graphics processing units (GPUs) and multimedia processors are semiconductor intellectual property cores produced by ARM Holdings for licensing in various ASIC designs by ARM partners.

Mali GPUs were developed by Falanx Microsystems A/S, which was a spin-off of a research project from the Norwegian University of Science and Technology.[1] Arm Holdings acquired Falanx Microsystems A/S on June 23, 2006 and renamed the company to Arm Norway.[2]

Technical details [edit]

Like other embedded IP cores for 3D rendering acceleration, the Mali GPU does not include display controllers driving monitors, in contrast to common desktop video cards. Instead, the Mali ARM core is a pure 3D engine that renders graphics into memory and passes the rendered image over to another core to handle display.

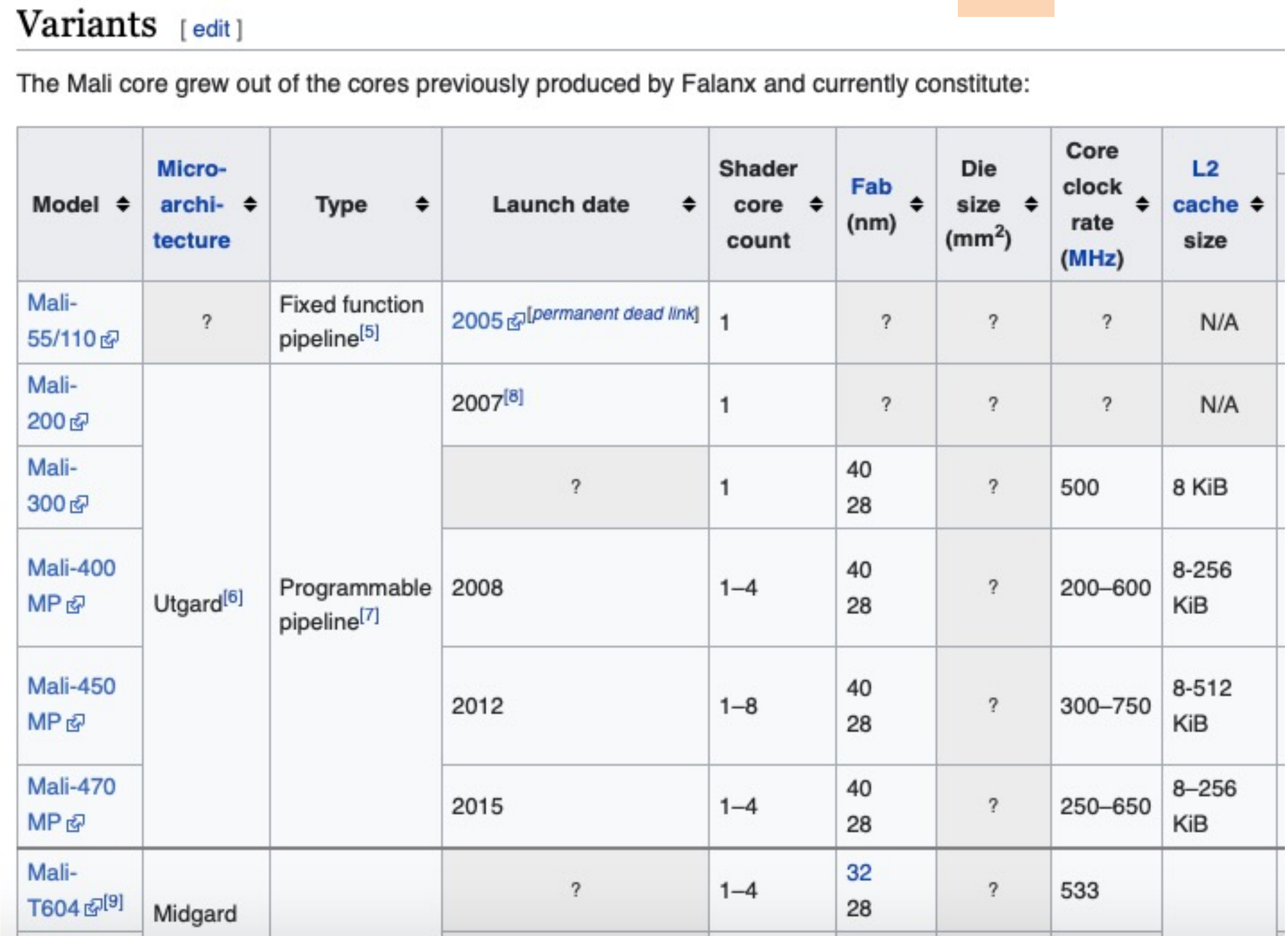
ARM does, however, license display controller SIP cores independently of the Mali 3D accelerator SIP block, e.g. Mali DP500, DP550 and DP650.[3]

ARM also supplies tools to help in authoring OpenGL ES shaders named *Mali GPU Shader Development Studio* and *Mali GPU User Interface Engine*.

Display controllers such as the ARM HDLCD display controller are available separately.[4]

Mali GPU Timeline

GPU



Variants [edit]

The Mali core grew out of the cores previously produced by Falanx and currently constitute:

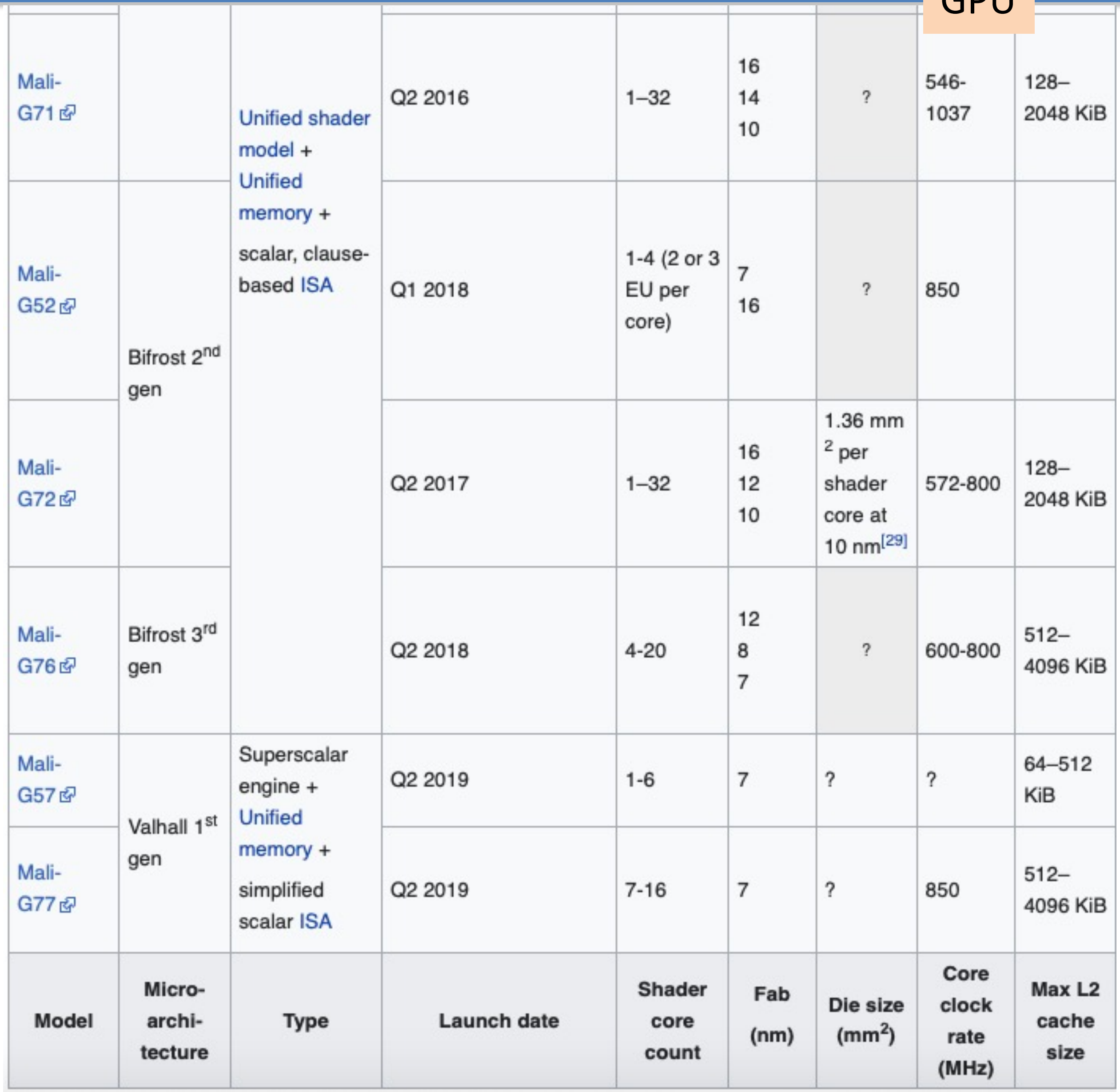
Model	Micro-architecture	Type	Launch date	Shader core count	Fab (nm)	Die size (mm^2)	Core clock rate (MHz)	L2 cache size
Mali-55/110	?	Fixed function pipeline[5]	2005 [permanent dead link]	1	?	?	?	N/A
Mali-200	Utgard[6]	Programmable pipeline[7]	2007[8]	1	?	?	?	N/A
Mali-300			?	1	40 28	?	500	8 KiB
Mali-400 MP			2008	1–4	40 28	?	200–600	8-256 KiB
Mali-450 MP			2012	1–8	40 28	?	300–750	8-512 KiB
Mali-470 MP			2015	1–4	40 28	?	250–650	8–256 KiB
Mali-T604[9]	Midgard		?	1–4	32 28	?	533	

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP122

Mali GPU Timeline

DR JEFF SOFTWARE
INDIE APP DEVELOPER
© Jeff Drobman
2016-2022

GPU



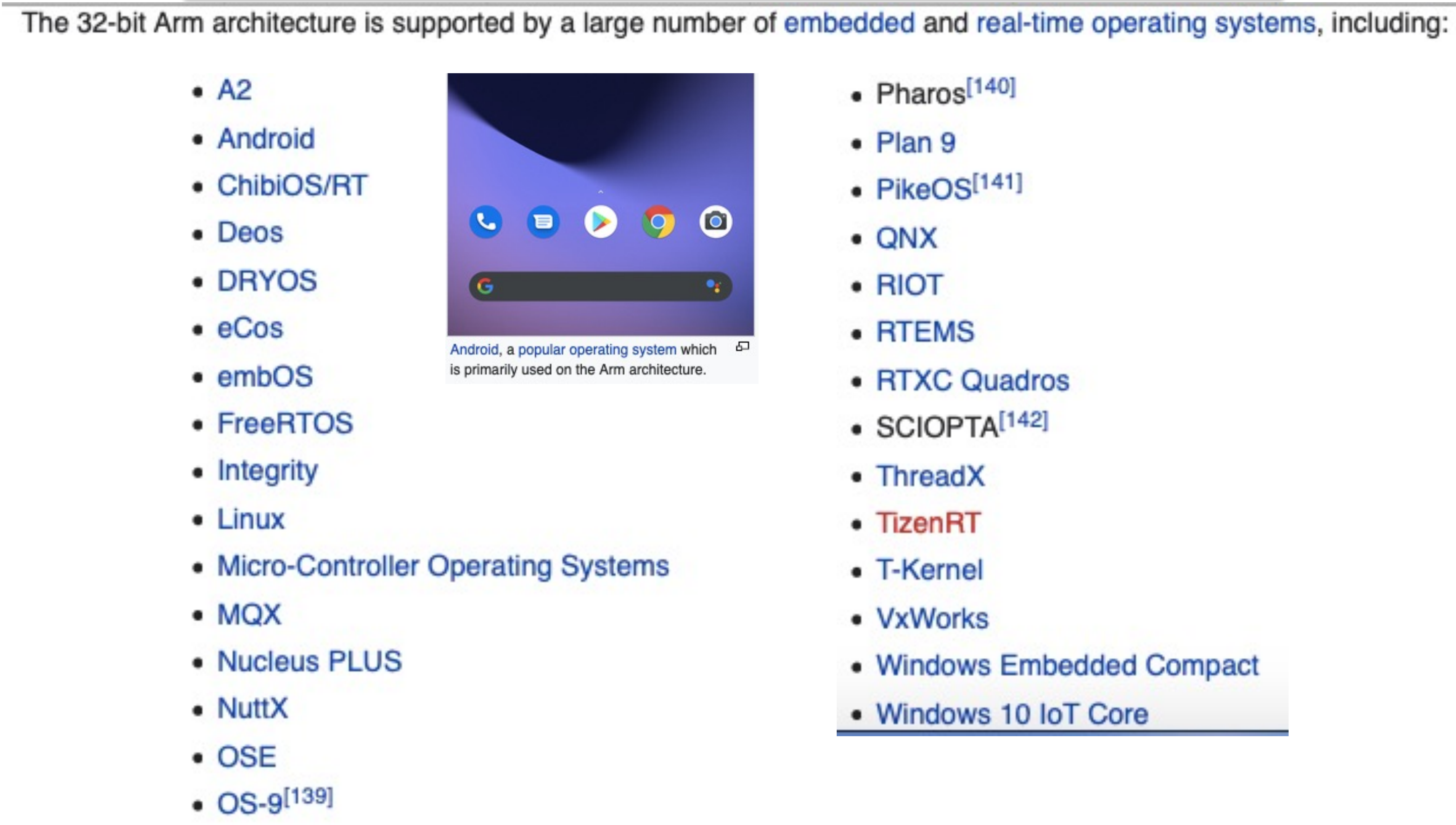
Model	Micro-archi-tecture	Type	Launch date	Shader core count	Fab (nm)	Die size (mm^2)	Core clock rate (MHz)	Max L2 cache size
Mali-G71		Unified shader model + Unified memory + scalar, clause-based ISA	Q2 2016	1–32	16 14 10	?	546-1037	128–2048 KiB
Mali-G52	Bifrost 2nd gen		Q1 2018	1-4 (2 or 3 EU per core)	7 16	?	850	
Mali-G72			Q2 2017	1–32	16 12 10	1.36 mm^2 per shader core at 10 nm[29]	572-800	128–2048 KiB
Mali-G76	Bifrost 3rd gen		Q2 2018	4-20	12 8 7	?	600-800	512–4096 KiB
Mali-G57	Valhall 1st gen	Superscalar engine + Unified memory + simplified scalar ISA	Q2 2019	1-6	7	?	?	64–512 KiB
Mali-G77			Q2 2019	7-16	7	?	850	512–4096 KiB

ARM

ARM OS

- Embedded (RT)
- Desktop

OS – RT



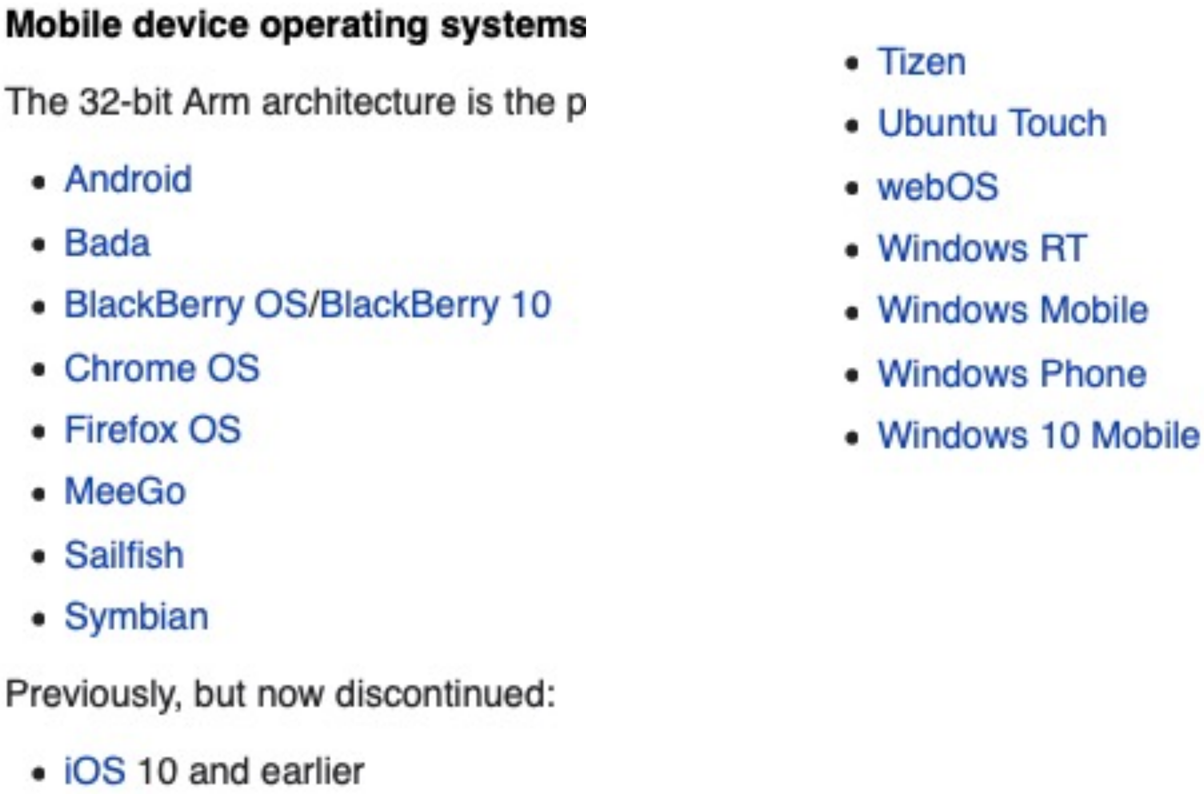
The 32-bit Arm architecture is supported by a large number of embedded and real-time operating systems, including:

- A2
- Android
- ChibiOS/RT
- Deos
- DRYOS
- eCos
- embOS
- FreeRTOS
- Integrity
- Linux
- Micro-Controller Operating Systems
- MQX
- Nucleus PLUS
- NuttX
- OSE
- OS-9[139]

Android, a popular operating system which is primarily used on the Arm architecture.

- Pharos[140]
- Plan 9
- PikeOS[141]
- QNX
- RIOT
- RTEMS
- RTXC Quadros
- SCIOPTA[142]
- ThreadX
- TizenRT
- T-Kernel
- VxWorks
- Windows Embedded Compact
- Windows 10 IoT Core

OS – Mobile



Mobile device operating systems

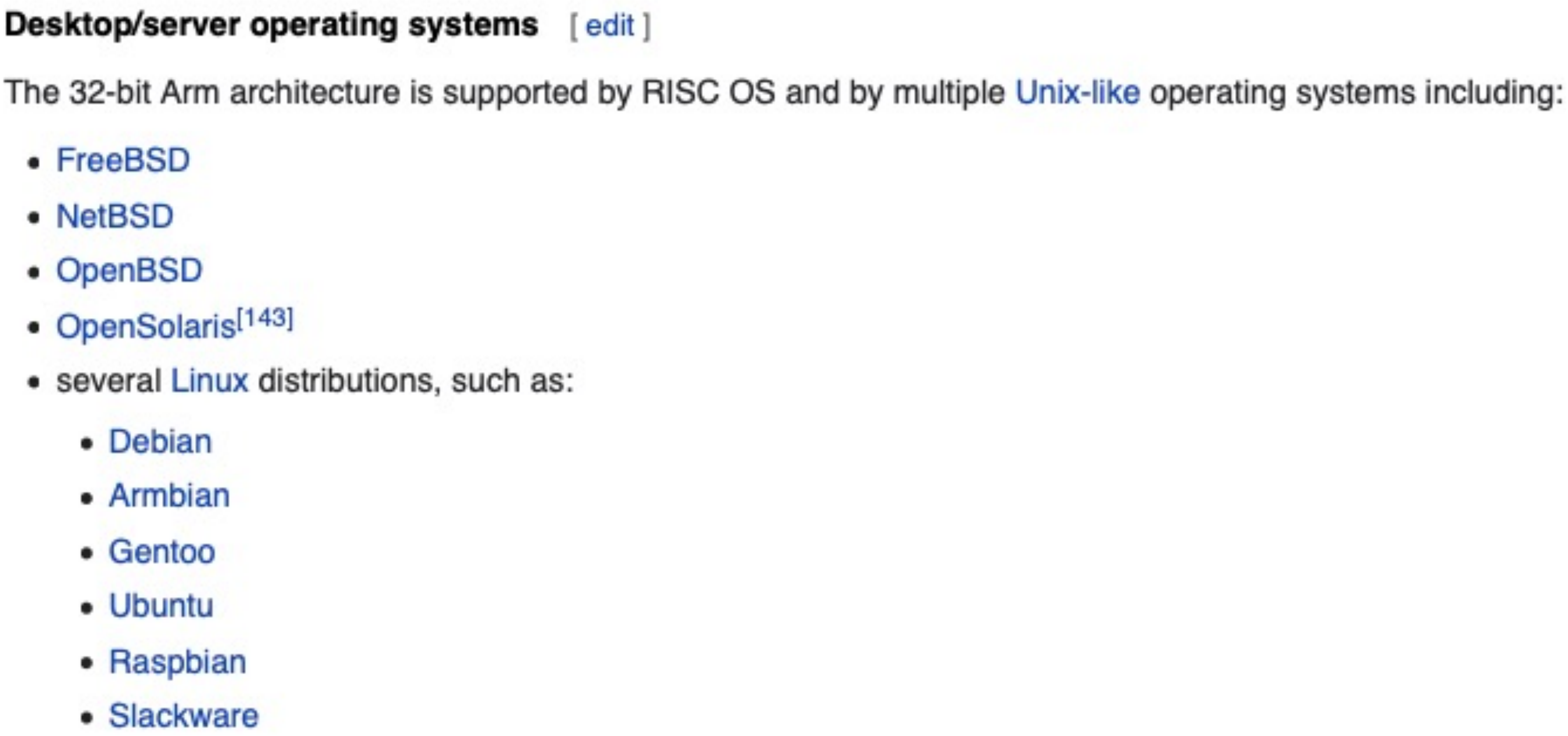
The 32-bit Arm architecture is the p

- Android
- Bada
- BlackBerry OS/BlackBerry 10
- Chrome OS
- Firefox OS
- MeeGo
- Sailfish
- Symbian
- Tizen
- Ubuntu Touch
- webOS
- Windows RT
- Windows Mobile
- Windows Phone
- Windows 10 Mobile

Previously, but now discontinued:

- iOS 10 and earlier

OS – Desktop



Desktop/server operating systems [edit]

The 32-bit Arm architecture is supported by RISC OS and by multiple Unix-like operating systems including:

- FreeBSD
- NetBSD
- OpenBSD
- OpenSolaris[143]
- several Linux distributions, such as:
  - Debian
  - Armbian
  - Gentoo
  - Ubuntu
  - Raspbian
  - Slackware

ARM

ISA

- ❖vs MIPS
- ❖Registers
- ❖Memory
- ❖Instructions

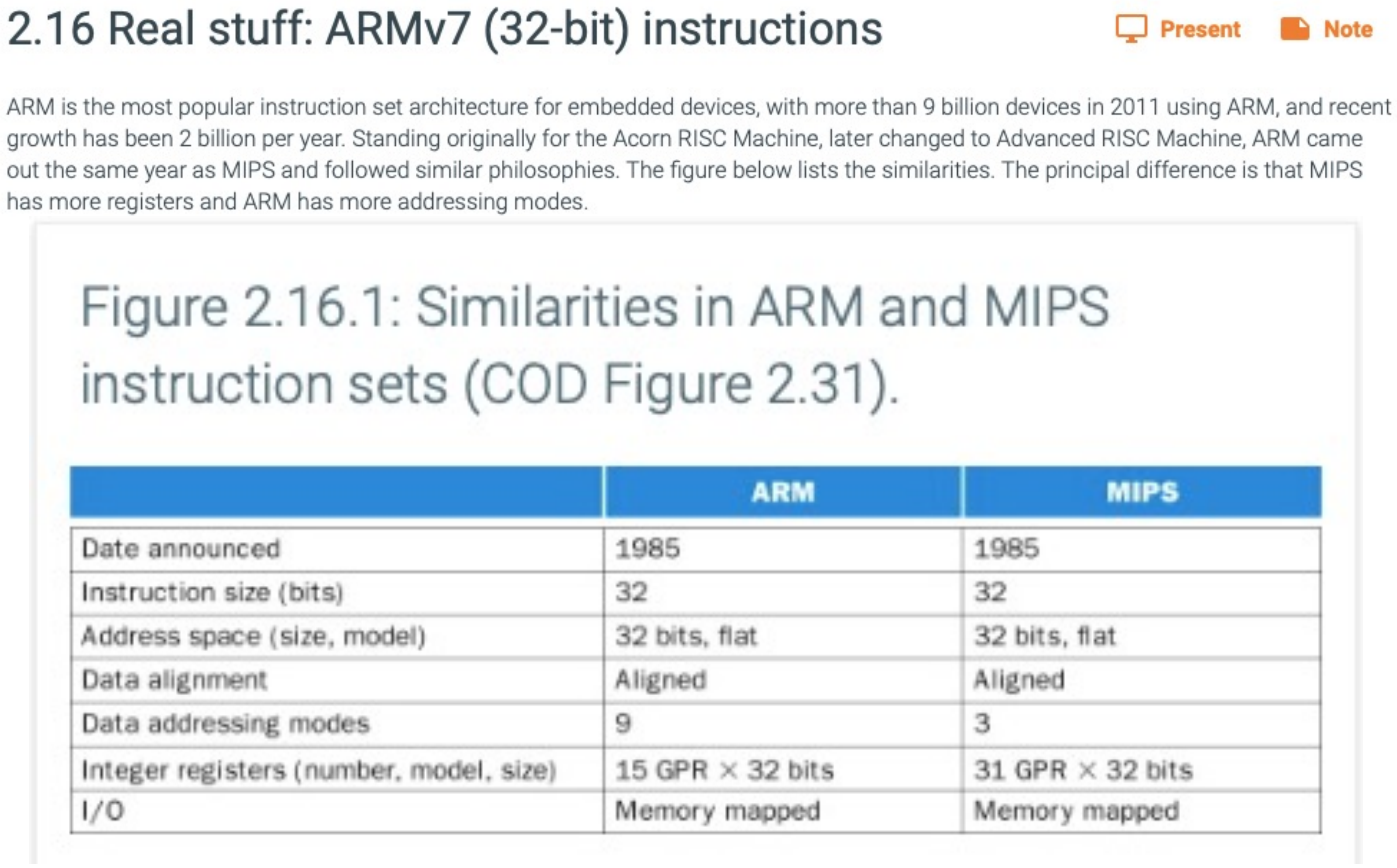
CSUN
CALIFORNIA
STATE UNIVERSITY
NORTHRIDGE

COMP122

ARM vs MIPS

DR JEFF SOFTWARE
INDIE APP DEVELOPER
© Jeff Drobman
2016-2022

Hennessy & Patterson



2.16 Real stuff: ARMv7 (32-bit) instructions

ARM is the most popular instruction set architecture for embedded devices, with more than 9 billion devices in 2011 using ARM, and recent growth has been 2 billion per year. Standing originally for the Acorn RISC Machine, later changed to Advanced RISC Machine, ARM came out the same year as MIPS and followed similar philosophies. The figure below lists the similarities. The principal difference is that MIPS has more registers and ARM has more addressing modes.

Figure 2.16.1: Similarities in ARM and MIPS instruction sets (COD Figure 2.31).

	ARM	MIPS
Date announced	1985	1985
Instruction size (bits)	32	32
Address space (size, model)	32 bits, flat	32 bits, flat
Data alignment	Aligned	Aligned
Data addressing modes	9	3
Integer registers (number, model, size)	15 GPR × 32 bits	31 GPR × 32 bits
I/O	Memory mapped	Memory mapped

ARM vs MIPS

Hennessy & Patterson

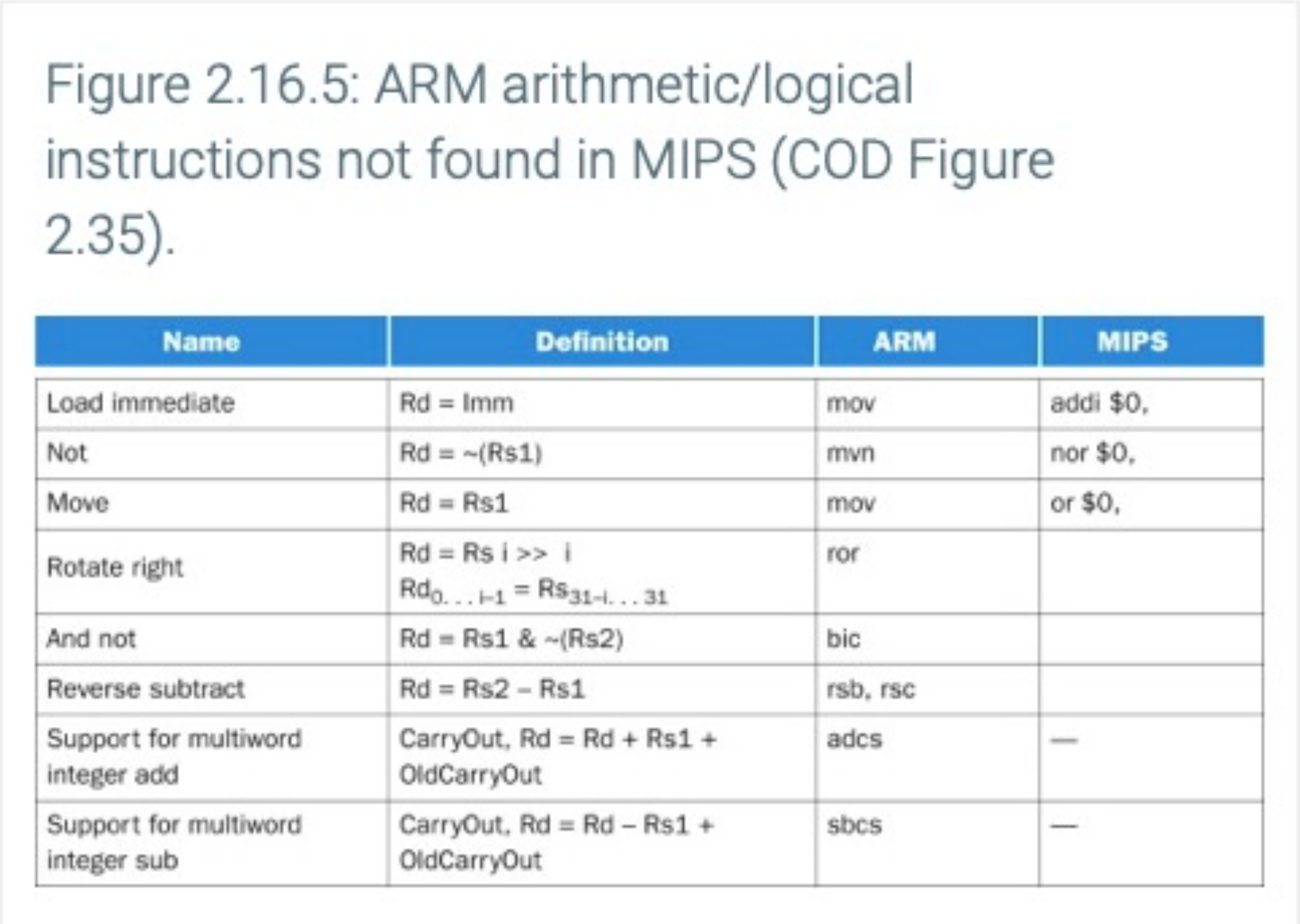
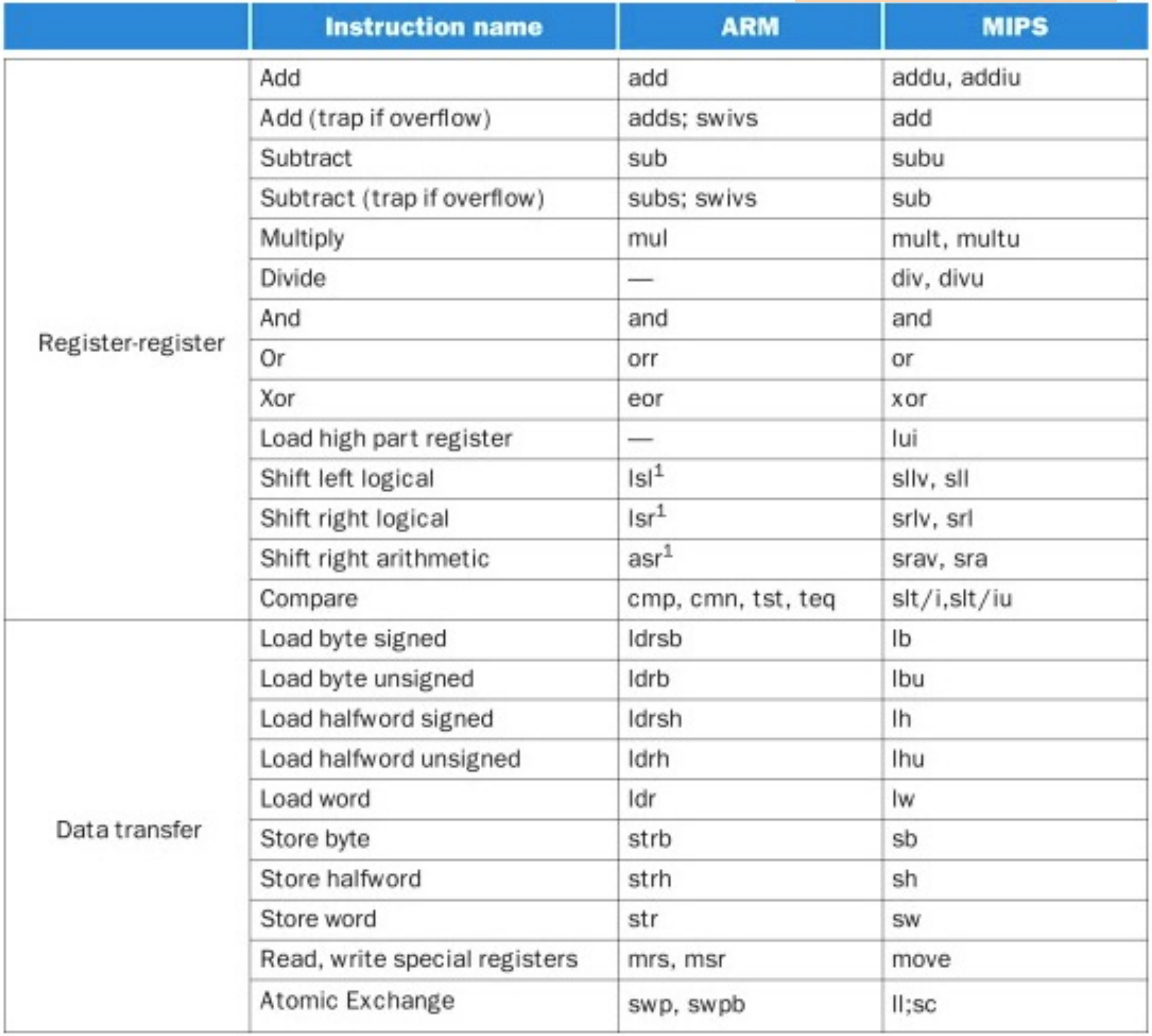


Figure 2.16.5: ARM arithmetic/logical instructions not found in MIPS (COD Figure 2.35).

Name	Definition	ARM	MIPS
Load immediate	Rd = Imm	mov	addi $0,
Not	Rd = ~(Rs1)	mvn	nor $0,
Move	Rd = Rs1	mov	or $0,
Rotate right	Rd = Rs i >> i $Rd_{0...i-1} = Rs_{31-i...31}$	ror	
And not	Rd = Rs1 & ~(Rs2)	bic	
Reverse subtract	Rd = Rs2 – Rs1	rsb, rsc	
Support for multiword integer add	CarryOut, Rd = Rd + Rs1 + OldCarryOut	adcs	—
Support for multiword integer sub	CarryOut, Rd = Rd – Rs1 + OldCarryOut	sbcs	—

ARM vs MIPS

Hennessy & Patterson



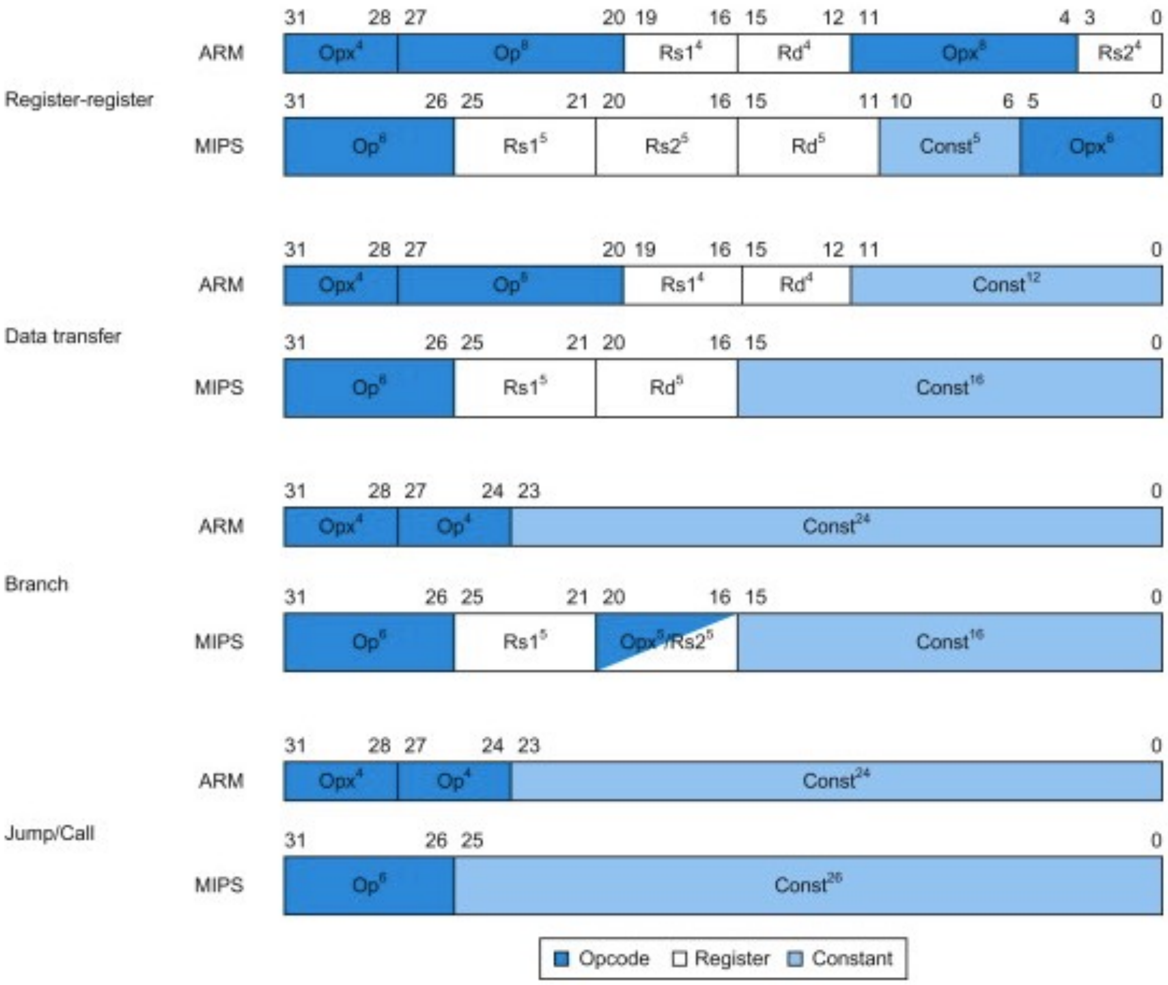
	Instruction name	ARM	MIPS
Register-register	Add	add	addu, addiu
	Add (trap if overflow)	adds; swivs	add
	Subtract	sub	subu
	Subtract (trap if overflow)	subs; swivs	sub
	Multiply	mul	mult, multu
	Divide	—	div, divu
	And	and	and
	Or	orr	or
	Xor	eor	xor
	Load high part register	—	lui
	Shift left logical	lsl[1]	sllv, sll
	Shift right logical	lsr[1]	srlv, srl
	Shift right arithmetic	asr[1]	srav, sra
	Compare	cmp, cmn, tst, teq	slt/i,slt/iu
Data transfer	Load byte signed	ldrsb	lb
	Load byte unsigned	ldrb	lbu
	Load halfword signed	ldrsh	lh
	Load halfword unsigned	ldrh	lhu
	Load word	ldr	lw
	Store byte	strb	sb
	Store halfword	strh	sh
	Store word	str	sw
	Read, write special registers	mrs, msr	move
	Atomic Exchange	swp, swpb	ll;sc

ARM vs MIPS

Hennessy & Patterson

Figure 2.16.4: Instruction formats, ARM and MIPS (COD Figure 2.34).

The differences result from whether the architecture has 16 or 32 registers.



Register-register

	31 28	27 20	19 16	15 12	11 4	3 0
ARM	Opx^4	Op^8	$Rs1^4$	Rd^4	Opx^8	$Rs2^4$

	31 26	25 21	20 16	15 11	10 6	5 0
MIPS	Op^6	$Rs1^5$	$Rs2^5$	Rd^5	$Const^5$	Opx^6

Data transfer

	31 28	27 20	19 16	15 12	11 0
ARM	Opx^4	Op^8	$Rs1^4$	Rd^4	$Const^{12}$

	31 26	25 21	20 16	15 0
MIPS	Op^6	$Rs1^5$	Rd^5	$Const^{16}$

Branch

	31 28	27 24	23 0
ARM	Opx^4	Op^4	$Const^{24}$

	31 26	25 21	20 16	15 0
MIPS	Op^6	$Rs1^5$	$Opx^5/Rs2^5$	$Const^{16}$

Jump/Call

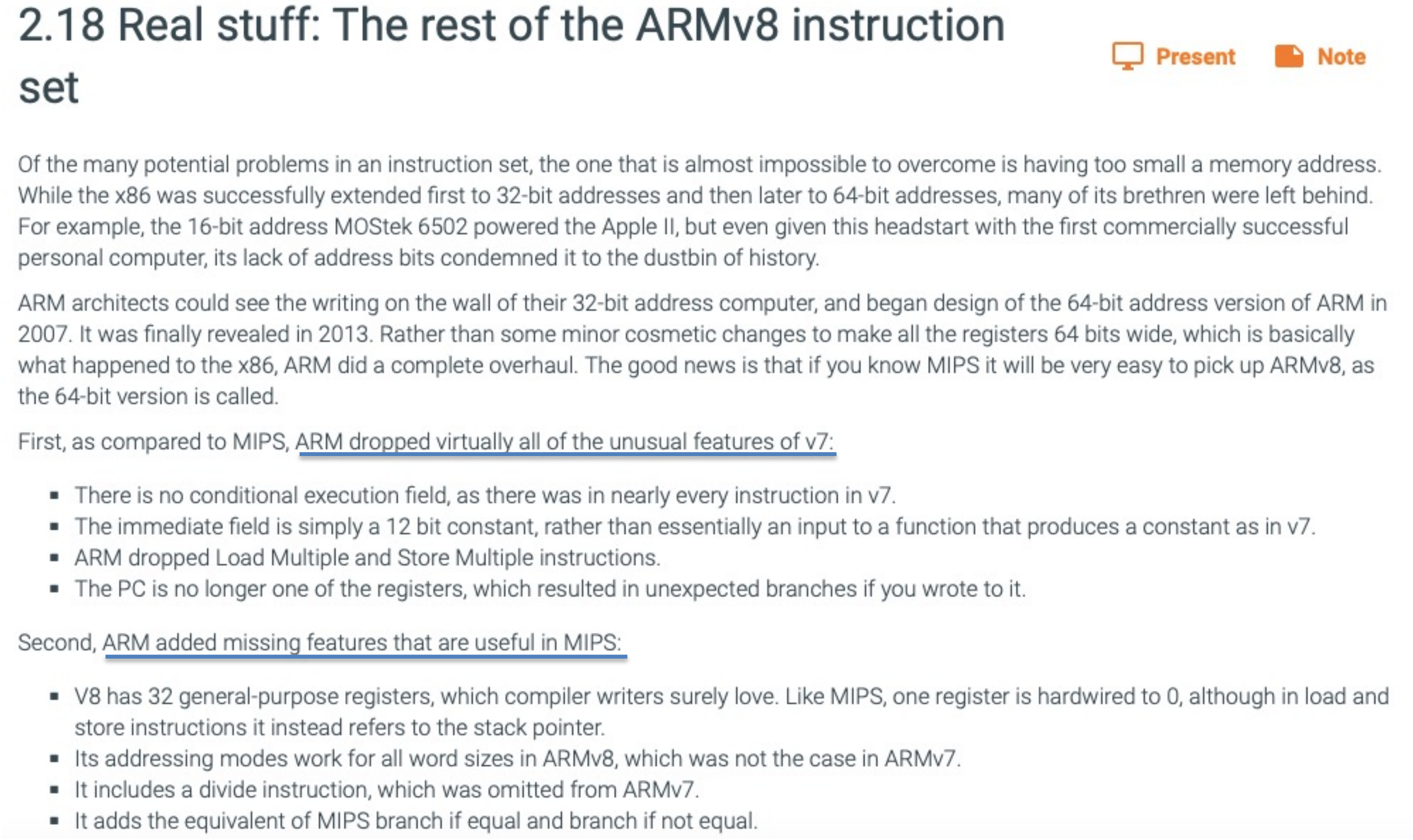
	31 28	27 24	23 0
ARM	Opx^4	Op^4	$Const^{24}$

	31 26	25 0
MIPS	Op^6	$Const^{26}$

■ Opcode □ Register ▨ Constant

ARMv8

Hennessy & Patterson



2.18 Real stuff: The rest of the ARMv8 instruction set

Of the many potential problems in an instruction set, the one that is almost impossible to overcome is having too small a memory address. While the x86 was successfully extended first to 32-bit addresses and then later to 64-bit addresses, many of its brethren were left behind. For example, the 16-bit address MOStek 6502 powered the Apple II, but even given this headstart with the first commercially successful personal computer, its lack of address bits condemned it to the dustbin of history.

ARM architects could see the writing on the wall of their 32-bit address computer, and began design of the 64-bit address version of ARM in 2007. It was finally revealed in 2013. Rather than some minor cosmetic changes to make all the registers 64 bits wide, which is basically what happened to the x86, ARM did a complete overhaul. The good news is that if you know MIPS it will be very easy to pick up ARMv8, as the 64-bit version is called.

First, as compared to MIPS, ARM dropped virtually all of the unusual features of v7:

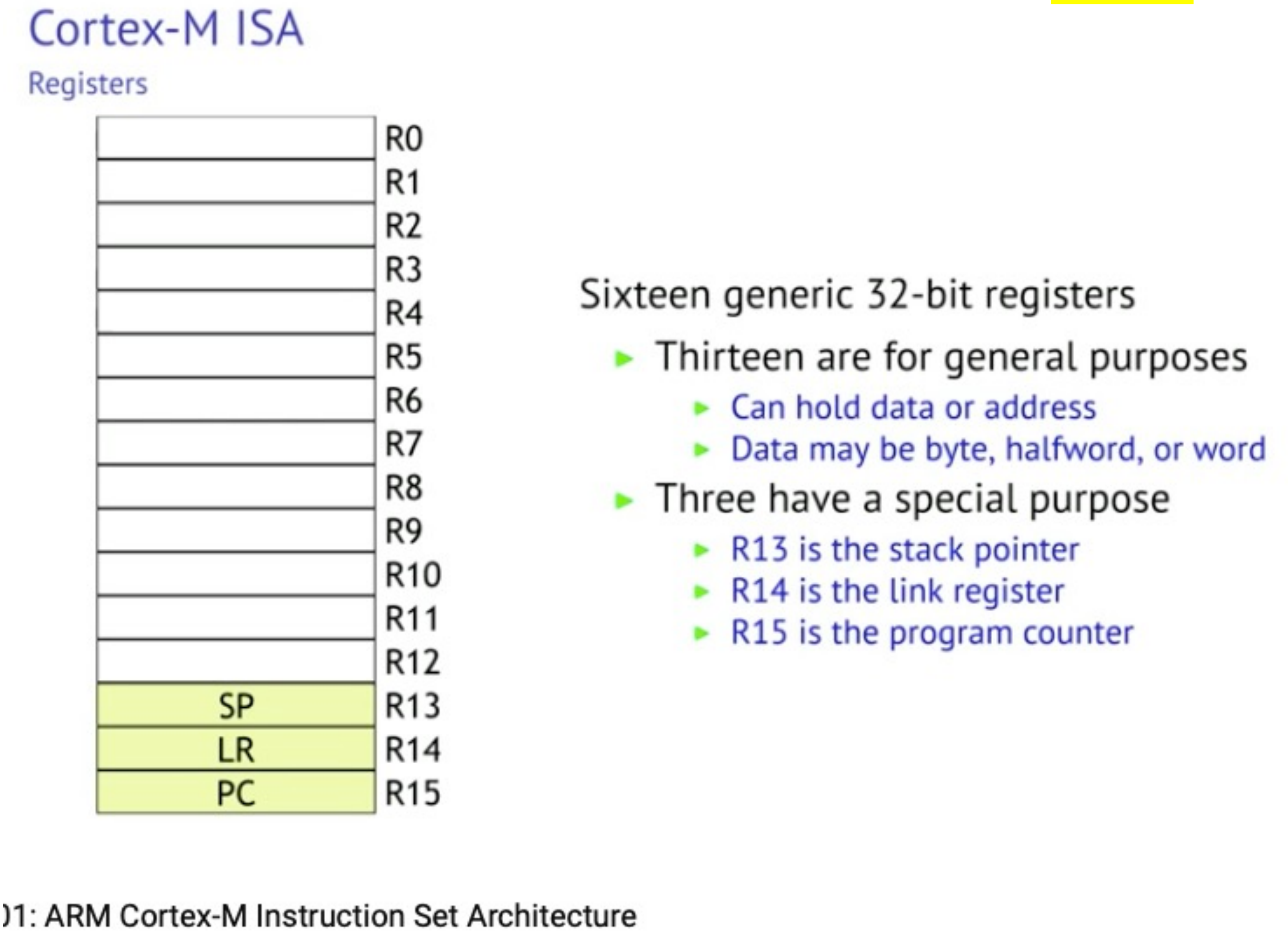
- There is no conditional execution field, as there was in nearly every instruction in v7.
- The immediate field is simply a 12 bit constant, rather than essentially an input to a function that produces a constant as in v7.
- ARM dropped Load Multiple and Store Multiple instructions.
- The PC is no longer one of the registers, which resulted in unexpected branches if you wrote to it.

Second, ARM added missing features that are useful in MIPS:

- V8 has 32 general-purpose registers, which compiler writers surely love. Like MIPS, one register is hardwired to 0, although in load and store instructions it instead refers to the stack pointer.
- Its addressing modes work for all word sizes in ARMv8, which was not the case in ARMv7.
- It includes a divide instruction, which was omitted from ARMv7.
- It adds the equivalent of MIPS branch if equal and branch if not equal.

ARM

ARMv**7**



Cortex-M ISA

Registers

	R0
	R1
	R2
	R3
	R4
	R5
	R6
	R7
	R8
	R9
	R10
	R11
	R12
SP	R13
LR	R14
PC	R15

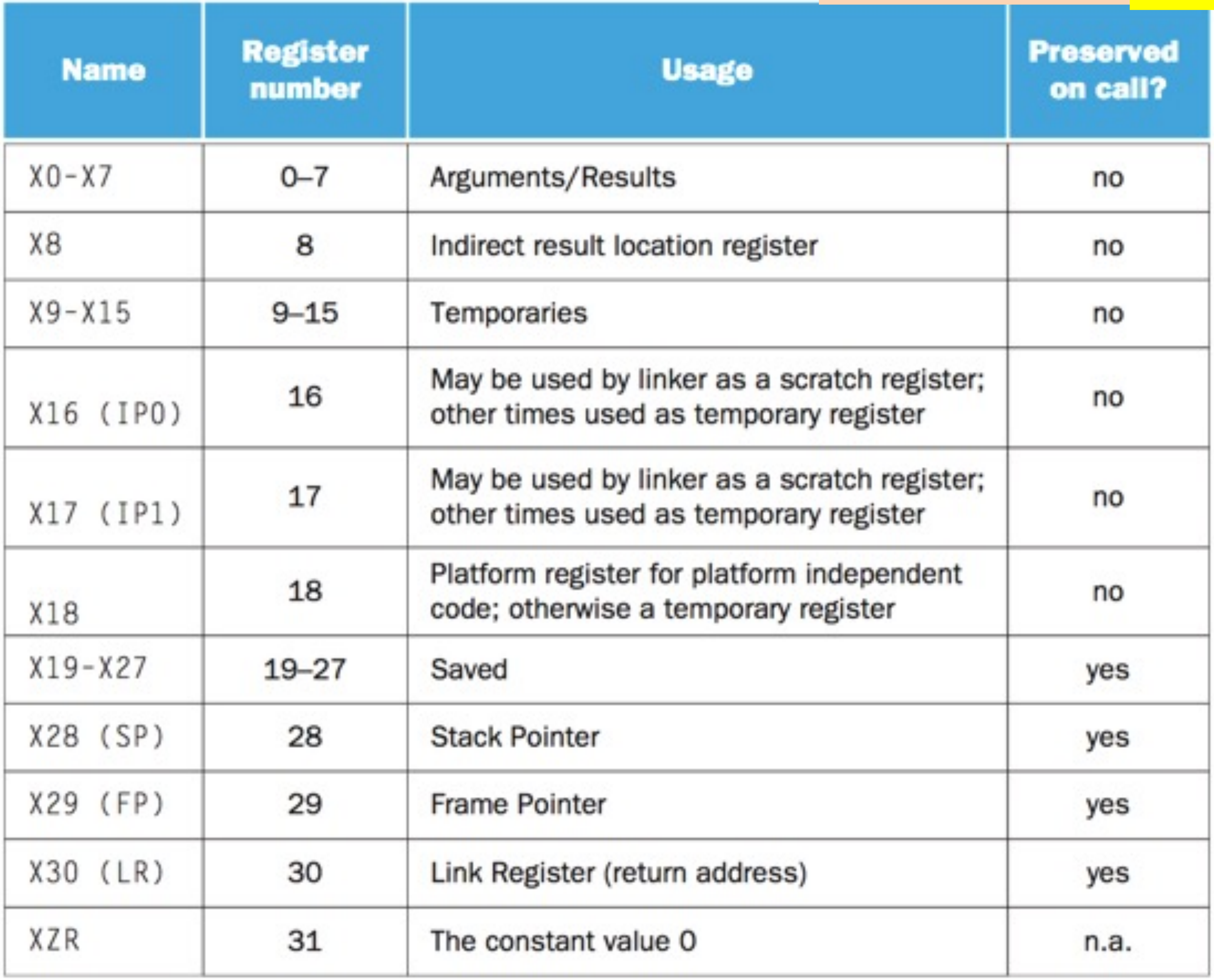
Sixteen generic 32-bit registers

- Thirteen are for general purposes
  - Can hold data or address
  - Data may be byte, halfword, or word
- Three have a special purpose
  - R13 is the stack pointer
  - R14 is the link register
  - R15 is the program counter

)1: ARM Cortex-M Instruction Set Architecture

Registers

Hennessy & Patterson ARMv8



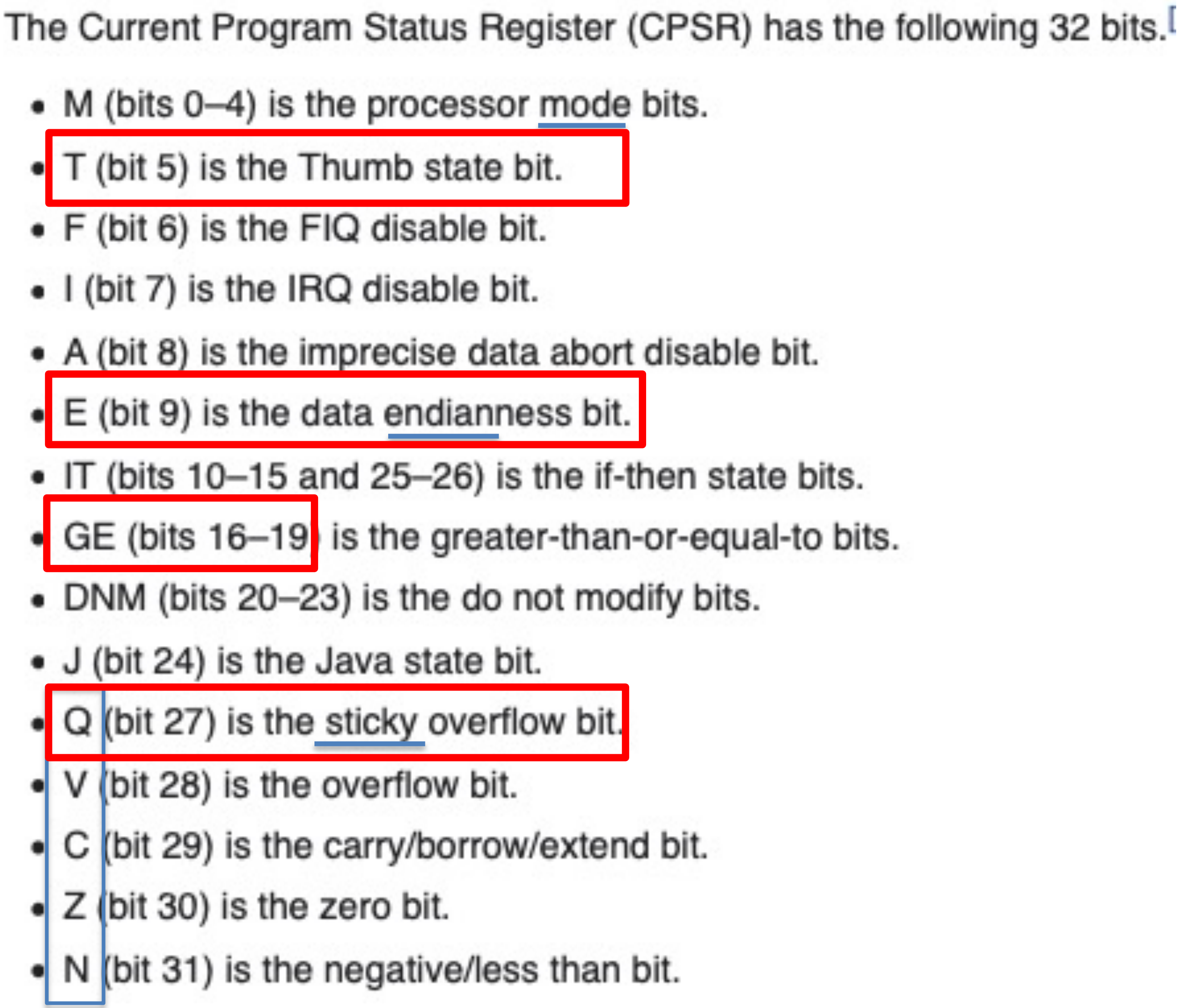
Name	Register number	Usage	Preserved on call?
X0-X7	0–7	Arguments/Results	no
X8	8	Indirect result location register	no
X9-X15	9–15	Temporaries	no
X16 (IP0)	16	May be used by linker as a scratch register; other times used as temporary register	no
X17 (IP1)	17	May be used by linker as a scratch register; other times used as temporary register	no
X18	18	Platform register for platform independent code; otherwise a temporary register	no
X19-X27	19–27	Saved	yes
X28 (SP)	28	Stack Pointer	yes
X29 (FP)	29	Frame Pointer	yes
X30 (LR)	30	Link Register (return address)	yes
XZR	31	The constant value 0	n.a.

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP122

DR JEFF SOFTWARE
INDIE APP DEVELOPER
© Jeff Drobman
2016-2022

Status & Control

CPSR



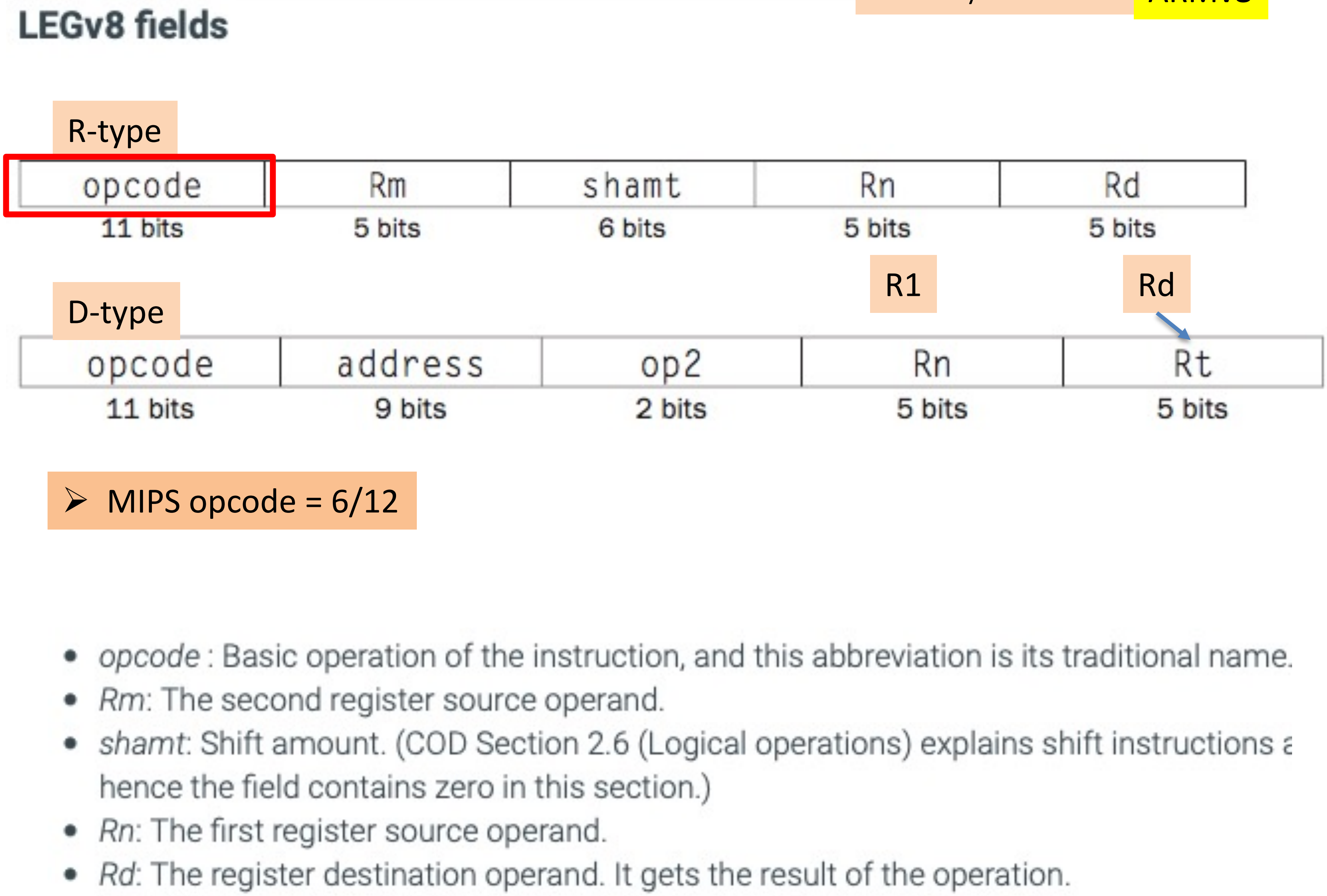
The Current Program Status Register (CPSR) has the following 32 bits.[

- M (bits 0–4) is the processor mode bits.
- T (bit 5) is the Thumb state bit.
- F (bit 6) is the FIQ disable bit.
- I (bit 7) is the IRQ disable bit.
- A (bit 8) is the imprecise data abort disable bit.
- E (bit 9) is the data endianness bit.
- IT (bits 10–15 and 25–26) is the if-then state bits.
- GE (bits 16–19) is the greater-than-or-equal-to bits.
- DNM (bits 20–23) is the do not modify bits.
- J (bit 24) is the Java state bit.
- Q (bit 27) is the sticky overflow bit.
- V (bit 28) is the overflow bit.
- C (bit 29) is the carry/borrow/extend bit.
- Z (bit 30) is the zero bit.
- N (bit 31) is the negative/less than bit.

Flags

Instruction Formats

Hennessy & Patterson **ARMv8**



LEGv8 fields

R-type

opcode	Rm	shamt	Rn	Rd
11 bits	5 bits	6 bits	5 bits	5 bits

R1 Rd

D-type

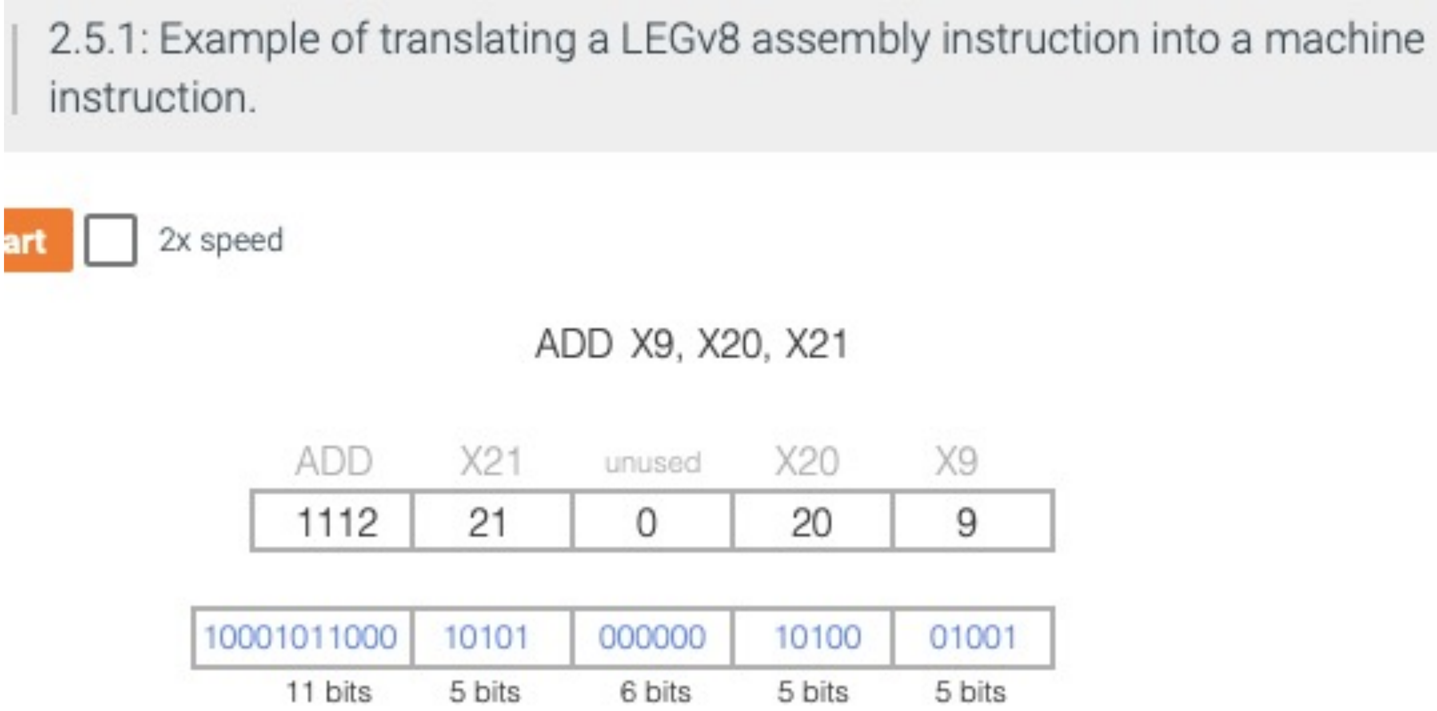
opcode	address	op2	Rn	Rt
11 bits	9 bits	2 bits	5 bits	5 bits

- MIPS opcode = 6/12

- *opcode* : Basic operation of the instruction, and this abbreviation is its traditional name.
- *Rm*: The second register source operand.
- *shamt*: Shift amount. (COD Section 2.6 (Logical operations) explains shift instructions a hence the field contains zero in this section.)
- *Rn*: The first register source operand.
- *Rd*: The register destination operand. It gets the result of the operation.

Instruction Encoding

Hennessy & Patterson ARMv8



2.5.1: Example of translating a LEGv8 assembly instruction into a machine instruction.

art □ 2x speed

ADD X9, X20, X21

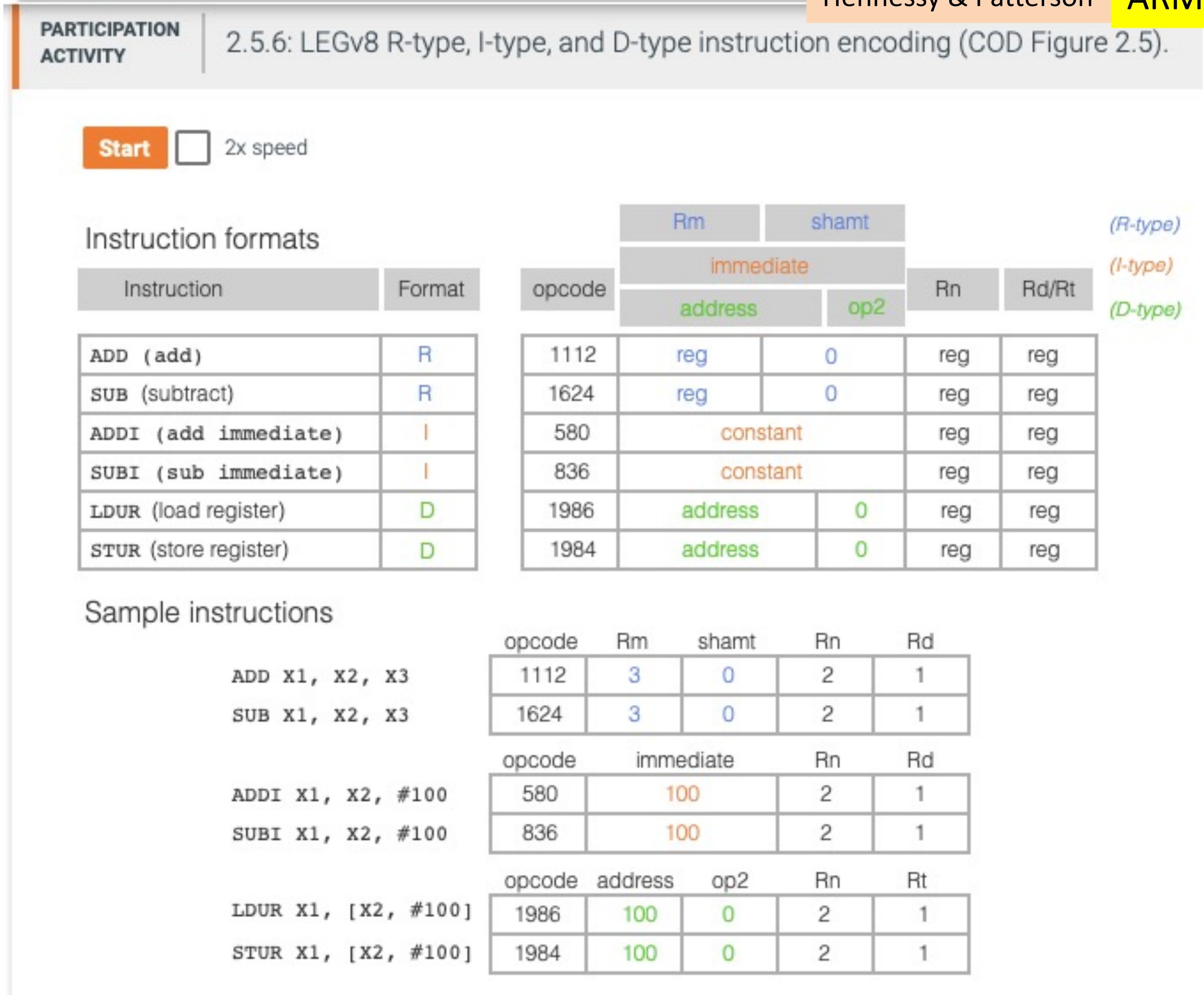
ADD	X21	unused	X20	X9
1112	21	0	20	9
10001011000	10101	000000	10100	01001
11 bits	5 bits	6 bits	5 bits	5 bits

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP122

Instruction Encoding

DR JEFF SOFTWARE
INDIE APP DEVELOPER

Hennessy & Patterson ARMv8



PARTICIPATION ACTIVITY 2.5.6: LEGv8 R-type, I-type, and D-type instruction encoding (COD Figure 2.5).

Instruction formats

			Rm	shamt			(R-type)
			immediate				(I-type)
Instruction	Format	opcode	address	op2	Rn	Rd/Rt	(D-type)
ADD (add)	R	1112	reg	0	reg	reg	
SUB (subtract)	R	1624	reg	0	reg	reg	
ADDI (add immediate)	I	580	constant		reg	reg	
SUBI (sub immediate)	I	836	constant		reg	reg	
LDUR (load register)	D	1986	address	0	reg	reg	
STUR (store register)	D	1984	address	0	reg	reg	

Sample instructions

	opcode	Rm	shamt	Rn	Rd
ADD X1, X2, X3	1112	3	0	2	1
SUB X1, X2, X3	1624	3	0	2	1

	opcode	immediate	Rn	Rd
ADDI X1, X2, #100	580	100	2	1
SUBI X1, X2, #100	836	100	2	1

	opcode	address	op2	Rn	Rt
LDUR X1, [X2, #100]	1986	100	0	2	1
STUR X1, [X2, #100]	1984	100	0	2	1

Addressing Modes

Hennessy & Patterson

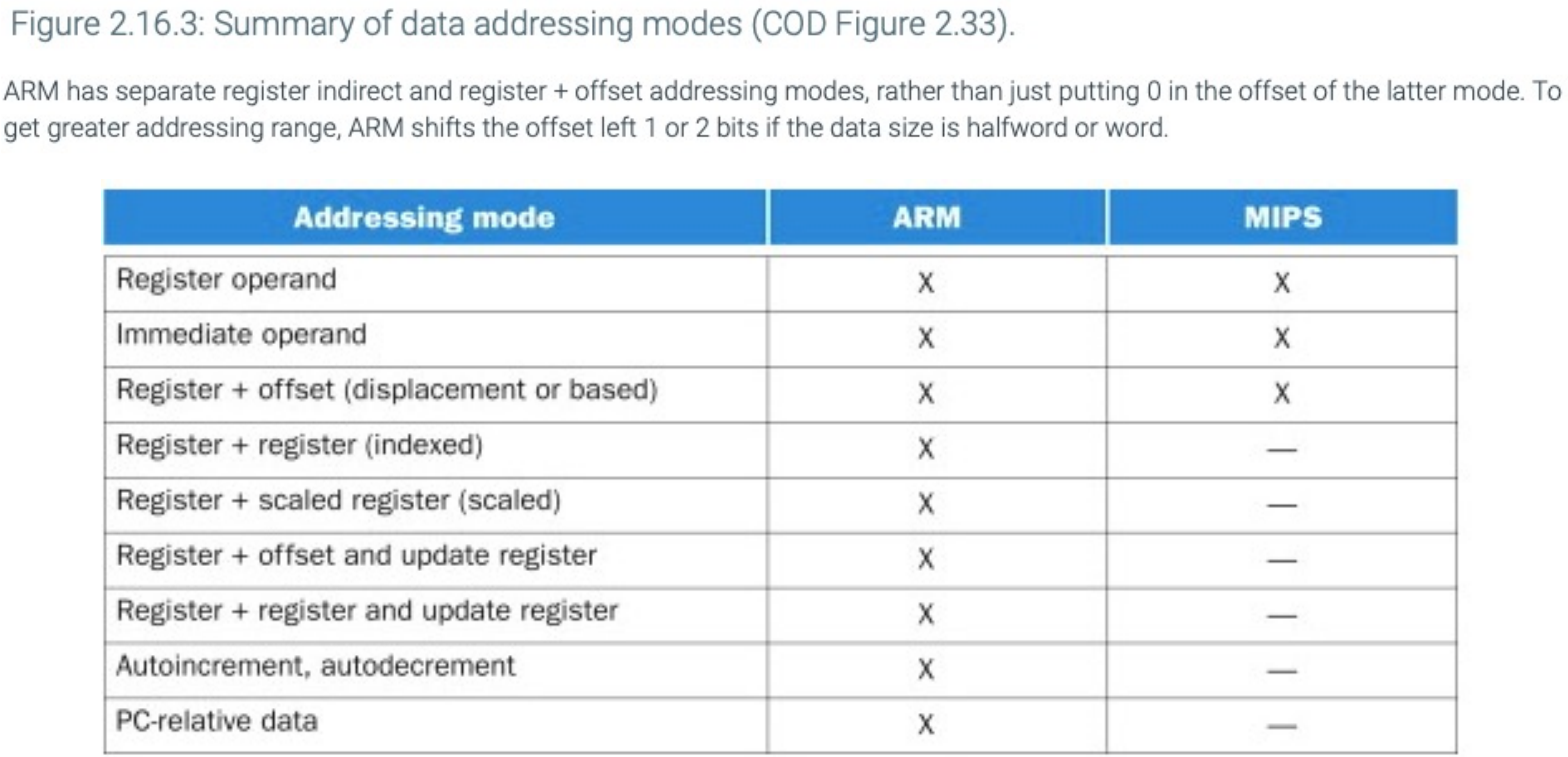
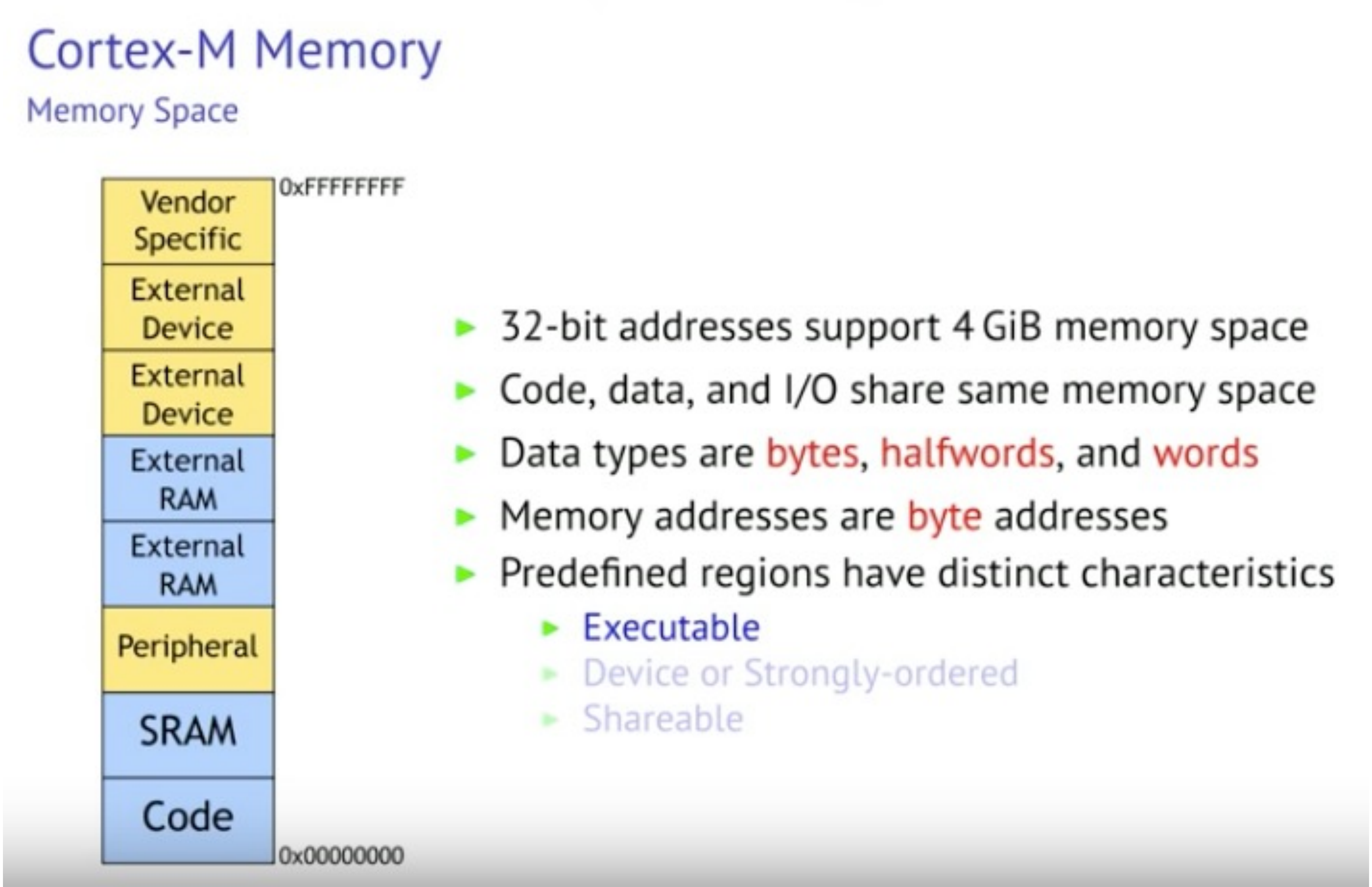


Figure 2.16.3: Summary of data addressing modes (COD Figure 2.33).

ARM has separate register indirect and register + offset addressing modes, rather than just putting 0 in the offset of the latter mode. To get greater addressing range, ARM shifts the offset left 1 or 2 bits if the data size is halfword or word.

Addressing mode	ARM	MIPS
Register operand	X	X
Immediate operand	X	X
Register + offset (displacement or based)	X	X
Register + register (indexed)	X	—
Register + scaled register (scaled)	X	—
Register + offset and update register	X	—
Register + register and update register	X	—
Autoincrement, autodecrement	X	—
PC-relative data	X	—

ARM



Cortex-M Memory

Memory Space

0xFFFFFFFF

Vendor Specific

External Device

External Device

External RAM

External RAM

Peripheral

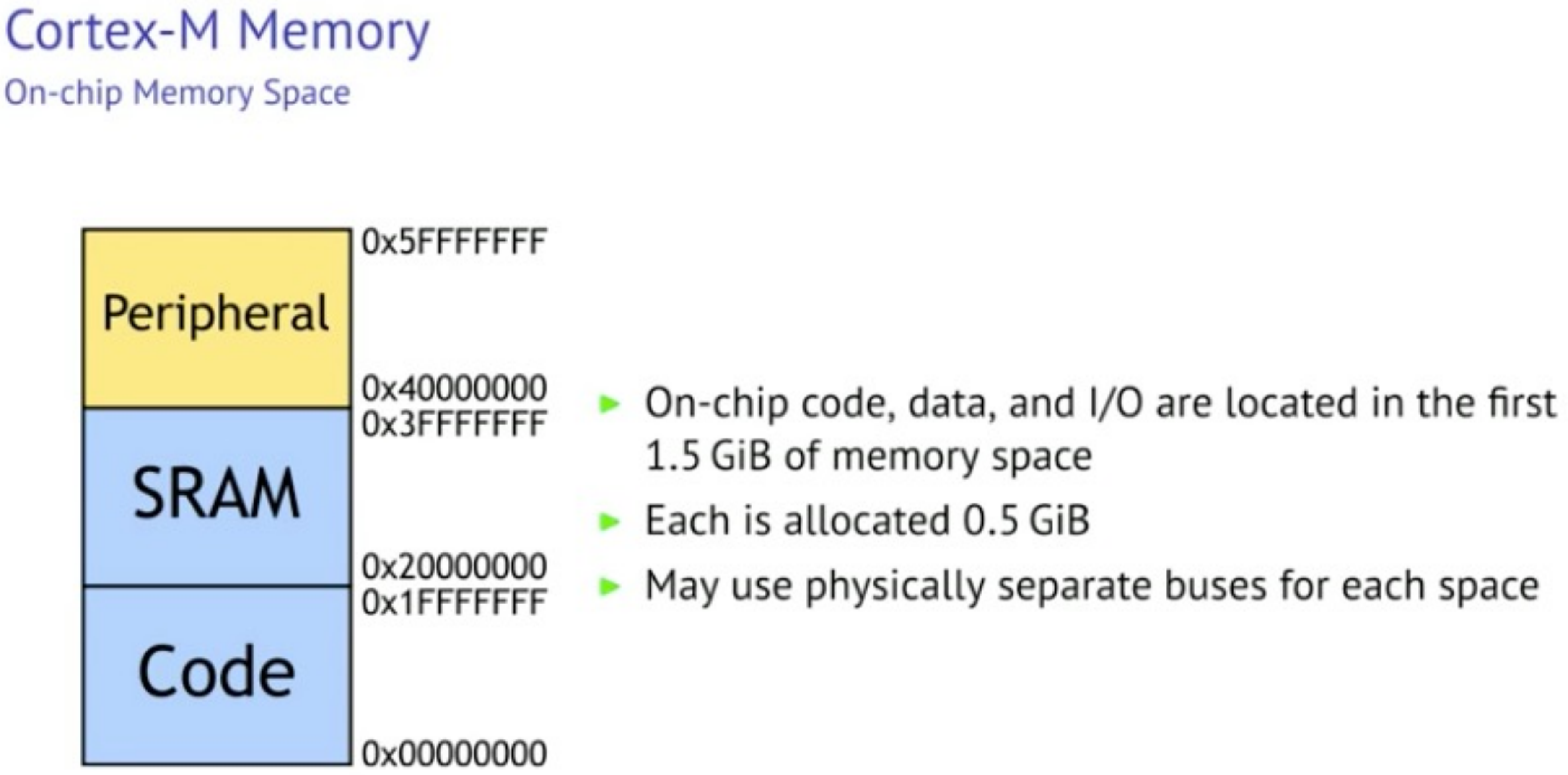
SRAM

Code

0x00000000

- 32-bit addresses support 4 GiB memory space
- Code, data, and I/O share same memory space
- Data types are bytes, halfwords, and words
- Memory addresses are byte addresses
- Predefined regions have distinct characteristics
  - Executable
  - Device or Strongly-ordered
  - Shareable

ARM



Cortex-M Memory

On-chip Memory Space

0x5FFFFFFF

Peripheral

0x40000000
0x3FFFFFFF

SRAM

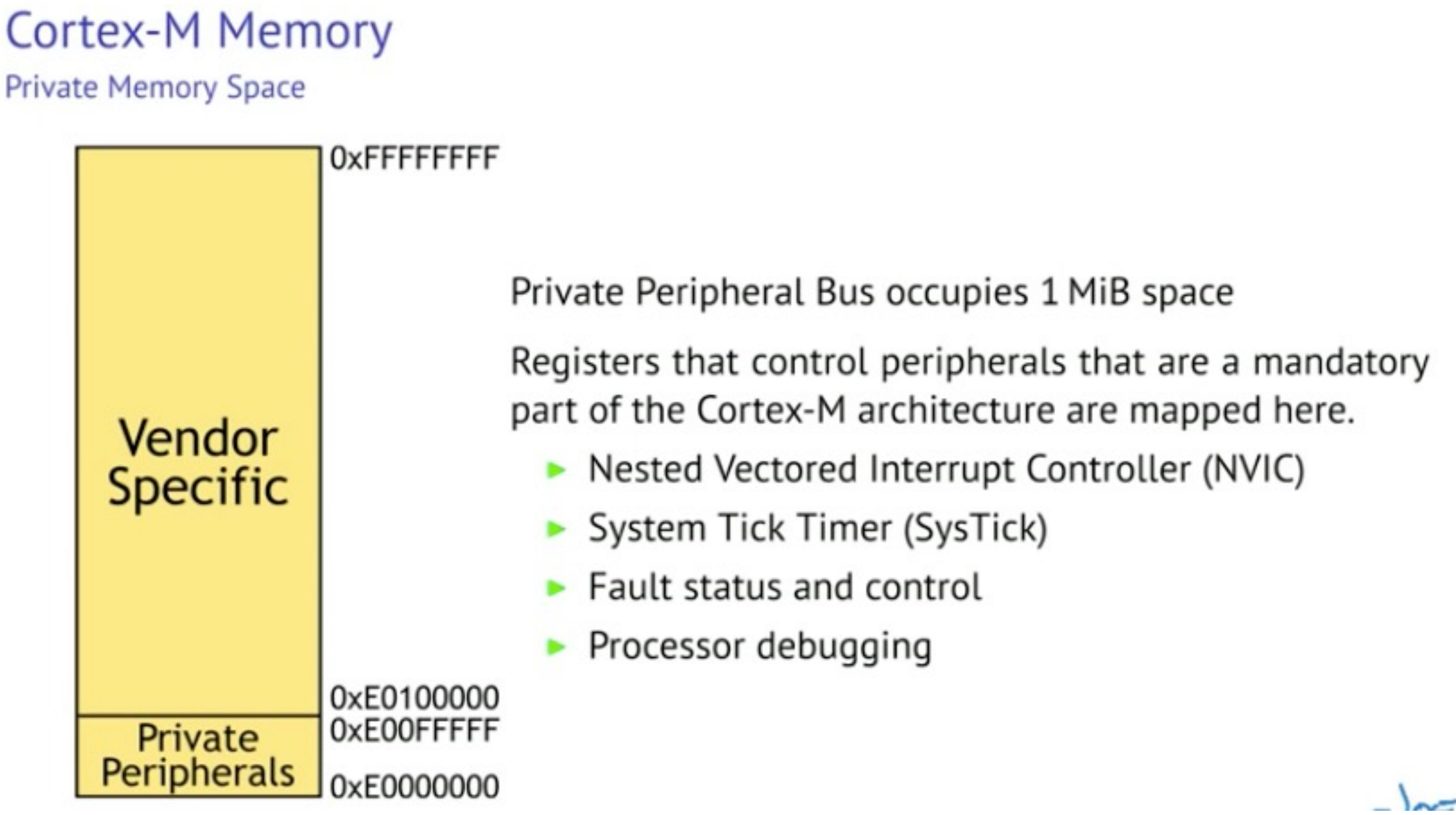
0x20000000
0x1FFFFFFF

Code

0x00000000

- On-chip code, data, and I/O are located in the first 1.5 GiB of memory space
- Each is allocated 0.5 GiB
- May use physically separate buses for each space

ARM



Cortex-M Memory

Private Memory Space

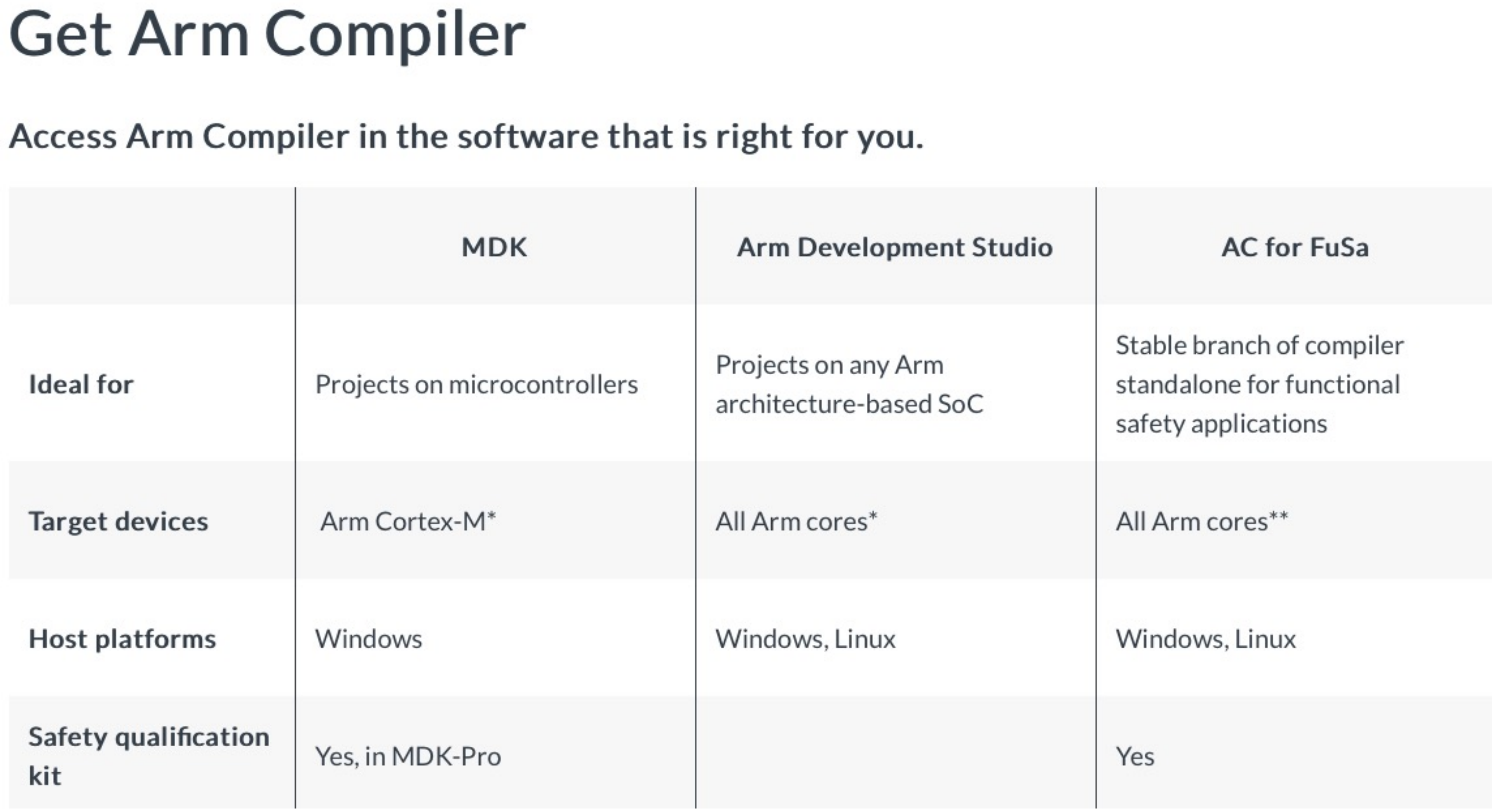
Private Peripheral Bus occupies 1 MiB space

Registers that control peripherals that are a mandatory part of the Cortex-M architecture are mapped here.

- Nested Vectored Interrupt Controller (NVIC)
- System Tick Timer (SysTick)
- Fault status and control
- Processor debugging

ARM

ARM

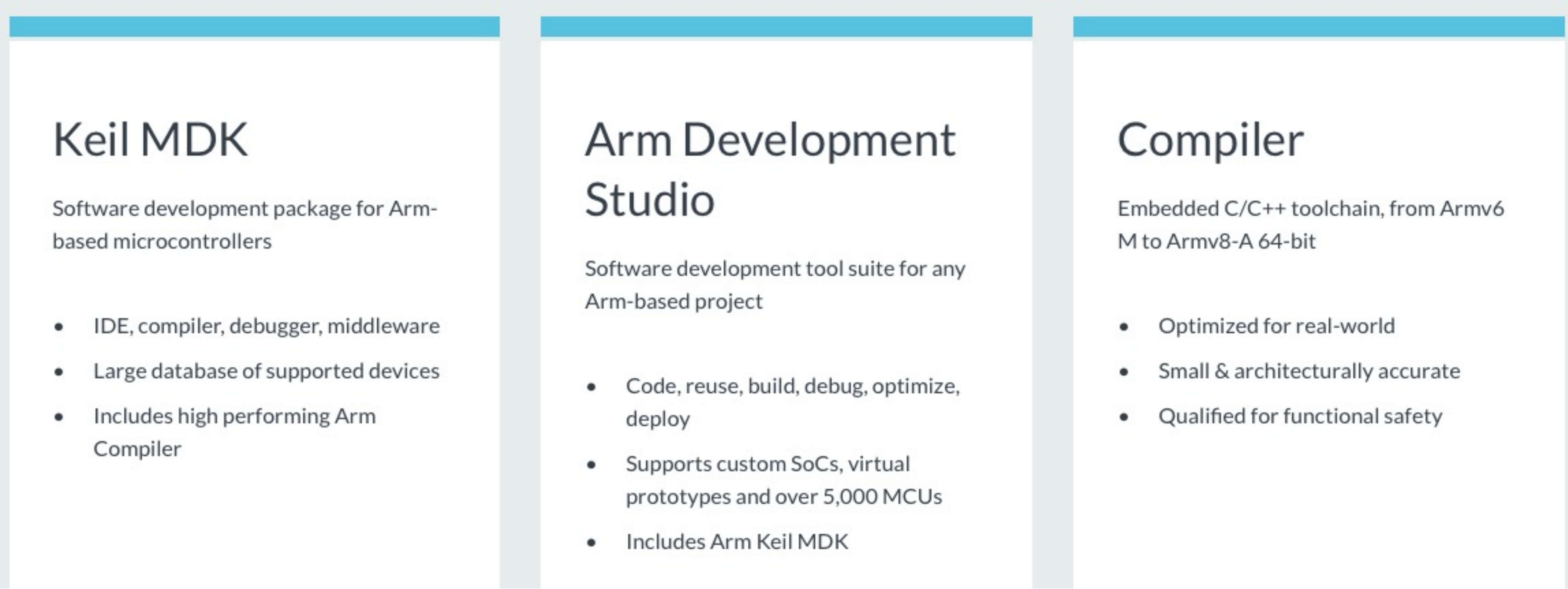


Get Arm Compiler

Access Arm Compiler in the software that is right for you.

	MDK	Arm Development Studio	AC for FuSa
Ideal for	Projects on microcontrollers	Projects on any Arm architecture-based SoC	Stable branch of compiler standalone for functional safety applications
Target devices	Arm Cortex-M*	All Arm cores*	All Arm cores**
Host platforms	Windows	Windows, Linux	Windows, Linux
Safety qualification kit	Yes, in MDK-Pro		Yes

ARM



Keil MDK

Software development package for Arm-based microcontrollers

- IDE, compiler, debugger, middleware
- Large database of supported devices
- Includes high performing Arm Compiler

Arm Development Studio

Software development tool suite for any Arm-based project

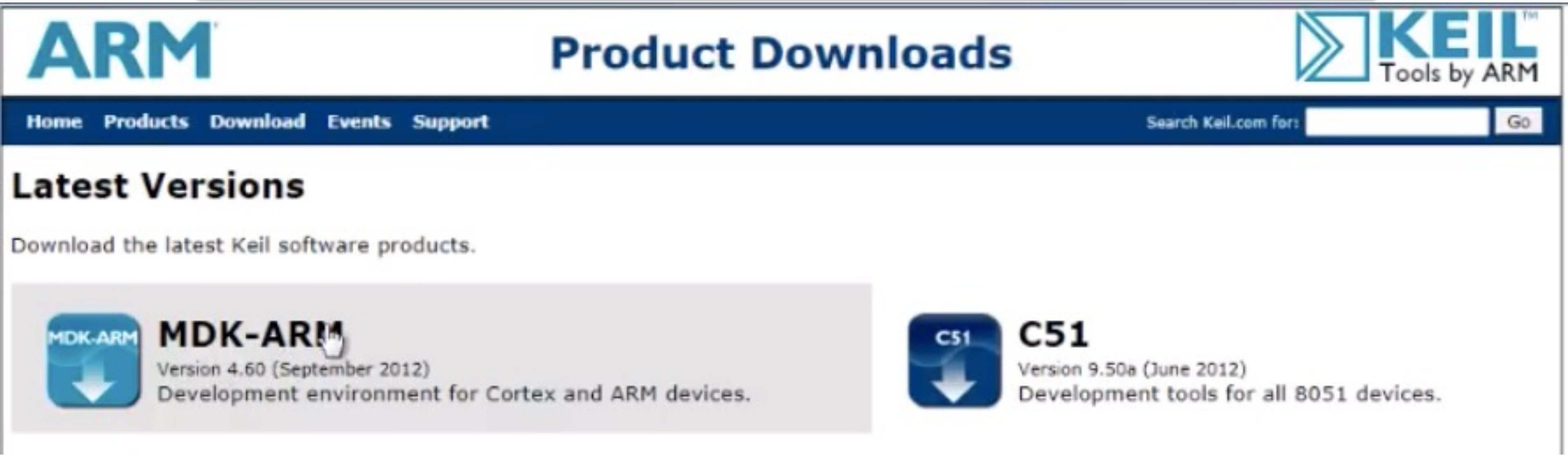
- Code, reuse, build, debug, optimize, deploy
- Supports custom SoCs, virtual prototypes and over 5,000 MCUs
- Includes Arm Keil MDK

Compiler

Embedded C/C++ toolchain, from Armv6 M to Armv8-A 64-bit

- Optimized for real-world
- Small & architecturally accurate
- Qualified for functional safety

ARM



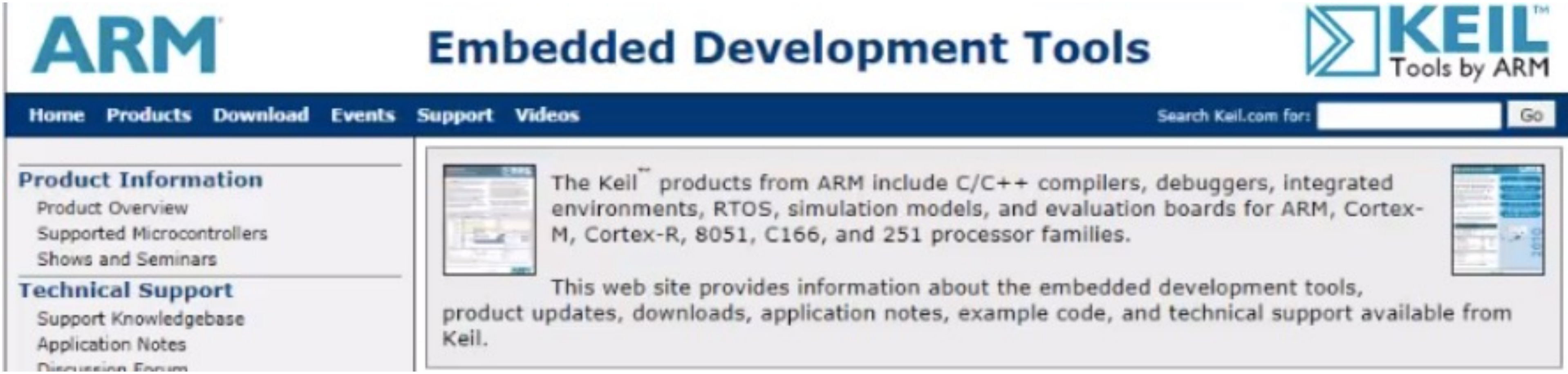
Product Downloads

Latest Versions

Download the latest Keil software products.

MDK-ARM
Version 4.60 (September 2012)
Development environment for Cortex and ARM devices.

C51
Version 9.50a (June 2012)
Development tools for all 8051 devices.



Embedded Development Tools

The Keil™ products from ARM include C/C++ compilers, debuggers, integrated environments, RTOS, simulation models, and evaluation boards for ARM, Cortex-M, Cortex-R, 8051, C166, and 251 processor families.

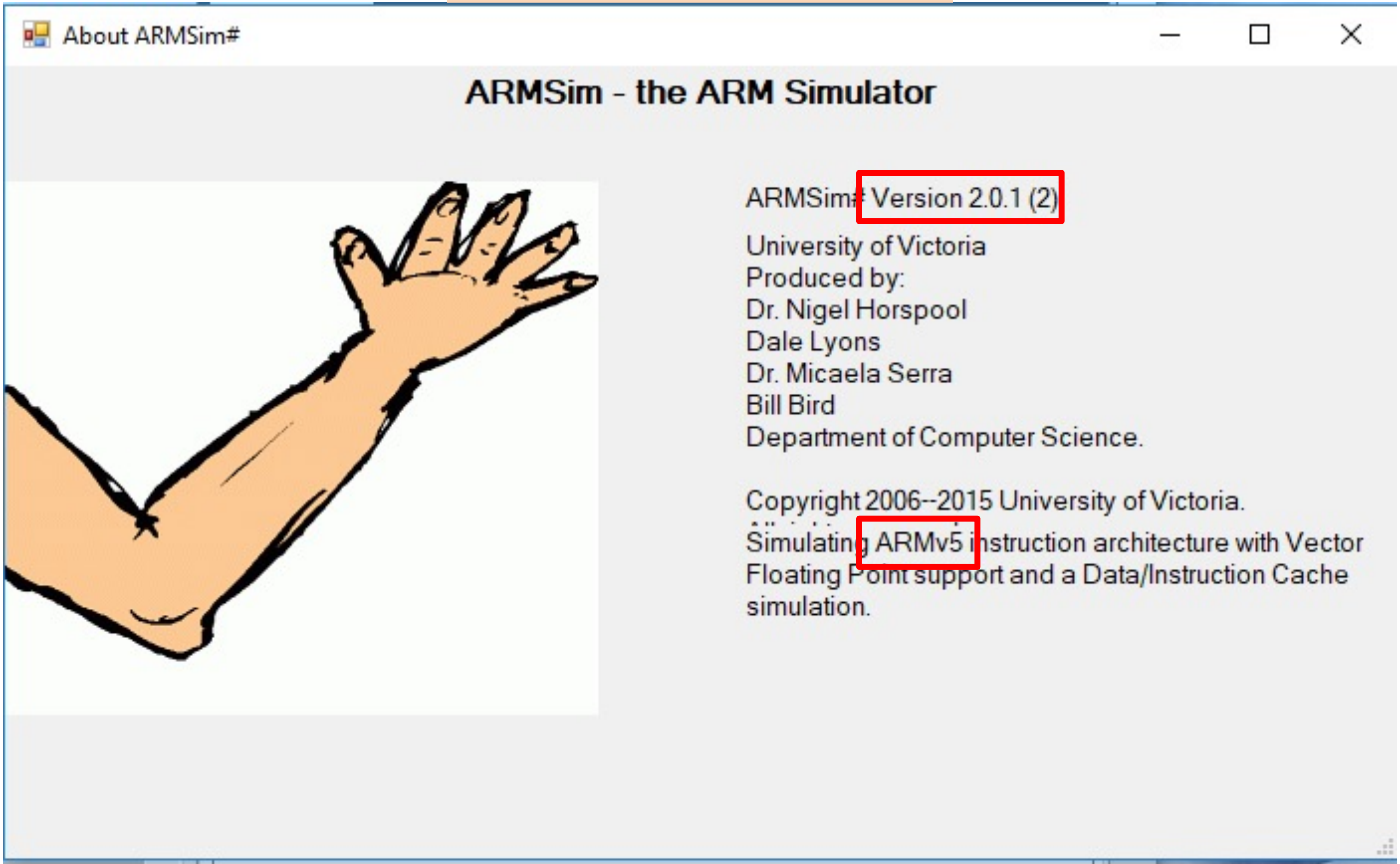
This web site provides information about the embedded development tools, product updates, downloads, application notes, example code, and technical support available from Keil.

Lab

ARMsim

ARM Sim

tinyurl.com/armsimcsun



ARMSim - the ARM Simulator

ARMSim# Version 2.0.1 (2)

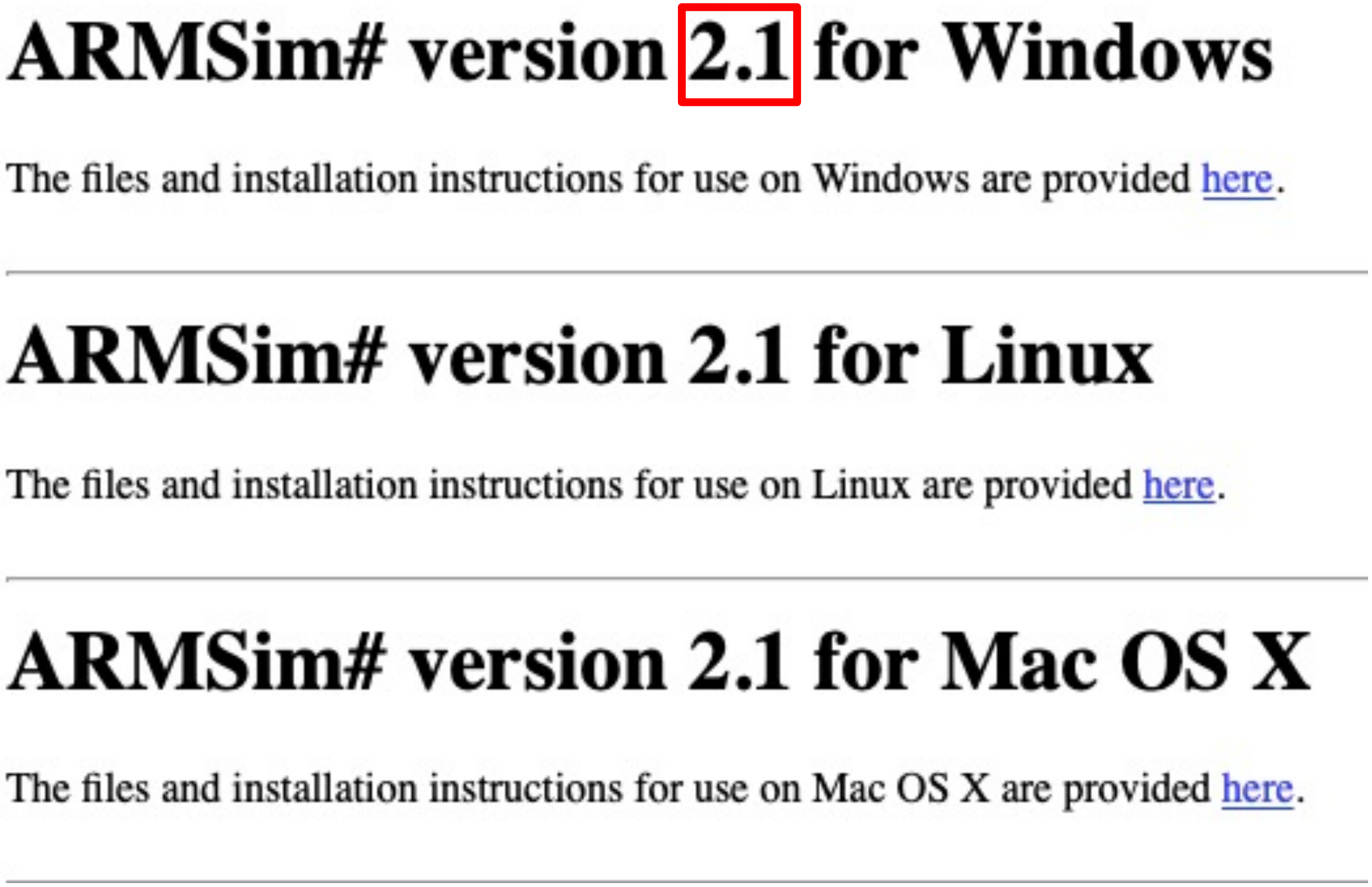
University of Victoria
Produced by:
Dr. Nigel Horspool
Dale Lyons
Dr. Micaela Serra
Bill Bird
Department of Computer Science.

Copyright 2006--2015 University of Victoria.

Simulating ARMv5 instruction architecture with Vector Floating Point support and a Data/Instruction Cache simulation.

ARM Sim

tinyurl.com/armsimcsun



ARMSim# version 2.1 for Windows

The files and installation instructions for use on Windows are provided here.

ARMSim# version 2.1 for Linux

The files and installation instructions for use on Linux are provided here.

ARMSim# version 2.1 for Mac OS X

The files and installation instructions for use on Mac OS X are provided here.

ARM Sim

Assembly Manual

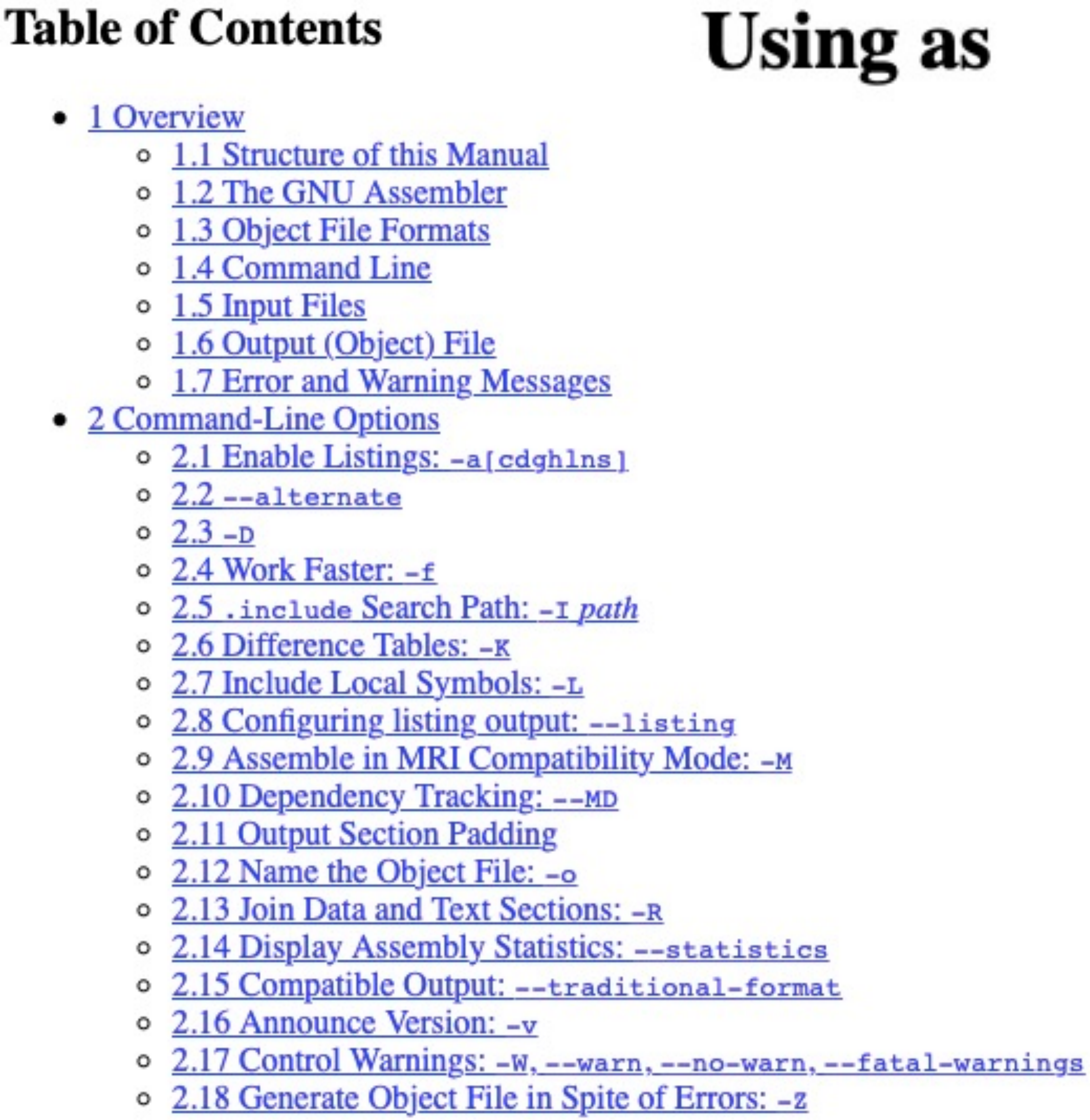
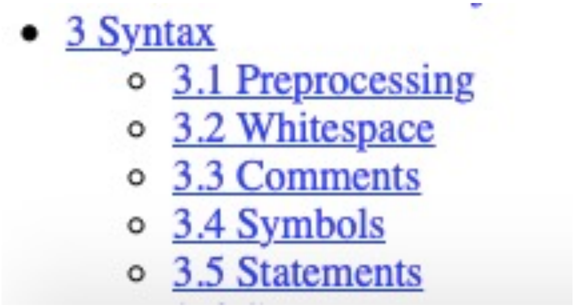
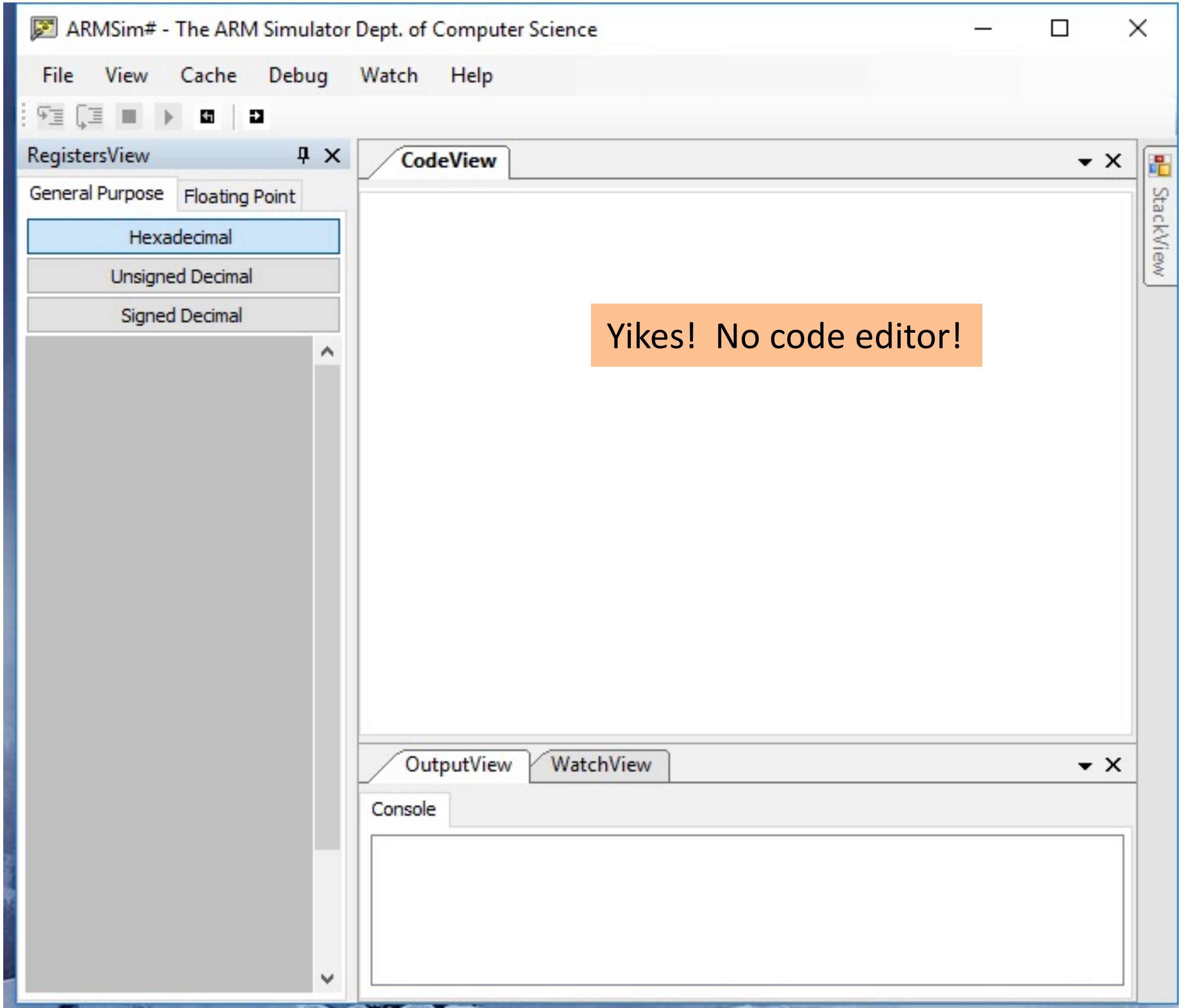


Table of Contents

Using as



ARM Sim

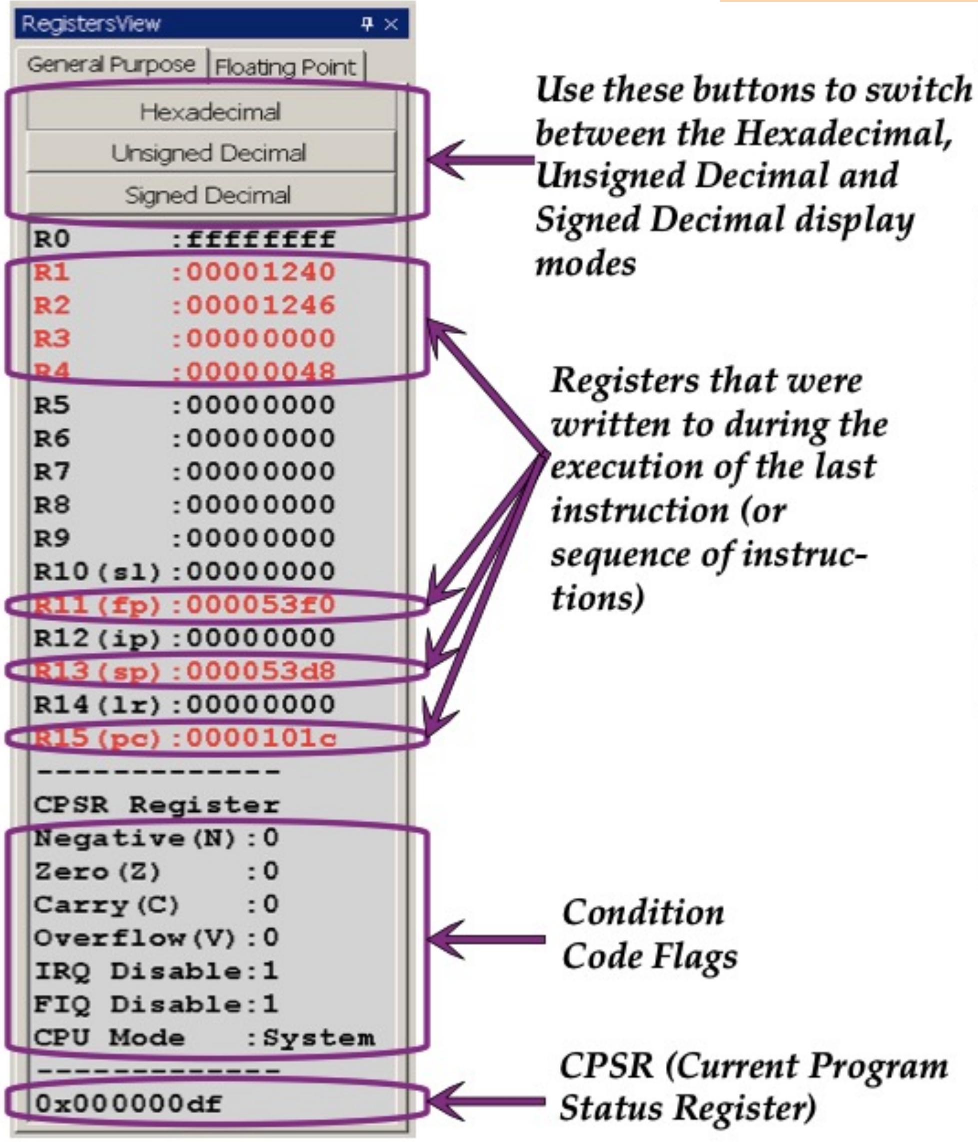


Yikes! No code editor!

ARM Sim

Register View

ARMSim# User Guide



```
R0       :ffffffff
R1       :00001240
R2       :00001246
R3       :00000000
R4       :00000048
R5       :00000000
R6       :00000000
R7       :00000000
R8       :00000000
R9       :00000000
R10(sl) :00000000
R11(fp) :000053f0
R12(ip) :00000000
R13(sp) :000053d8
R14(lr) :00000000
R15(pc) :0000101c
-------------
CPSR Register
Negative(N):0
Zero(Z)     :0
Carry(C)    :0
Overflow(V):0
IRQ Disable:1
FIQ Disable:1
CPU Mode    :System
-------------
0x000000df
```

Use these buttons to switch between the Hexadecimal, Unsigned Decimal and Signed Decimal display modes

Registers that were written to during the execution of the last instruction (or sequence of instructions)

Condition Code Flags

CPSR (Current Program Status Register)

Registers R10-R15 are also labelled:

Table 3.

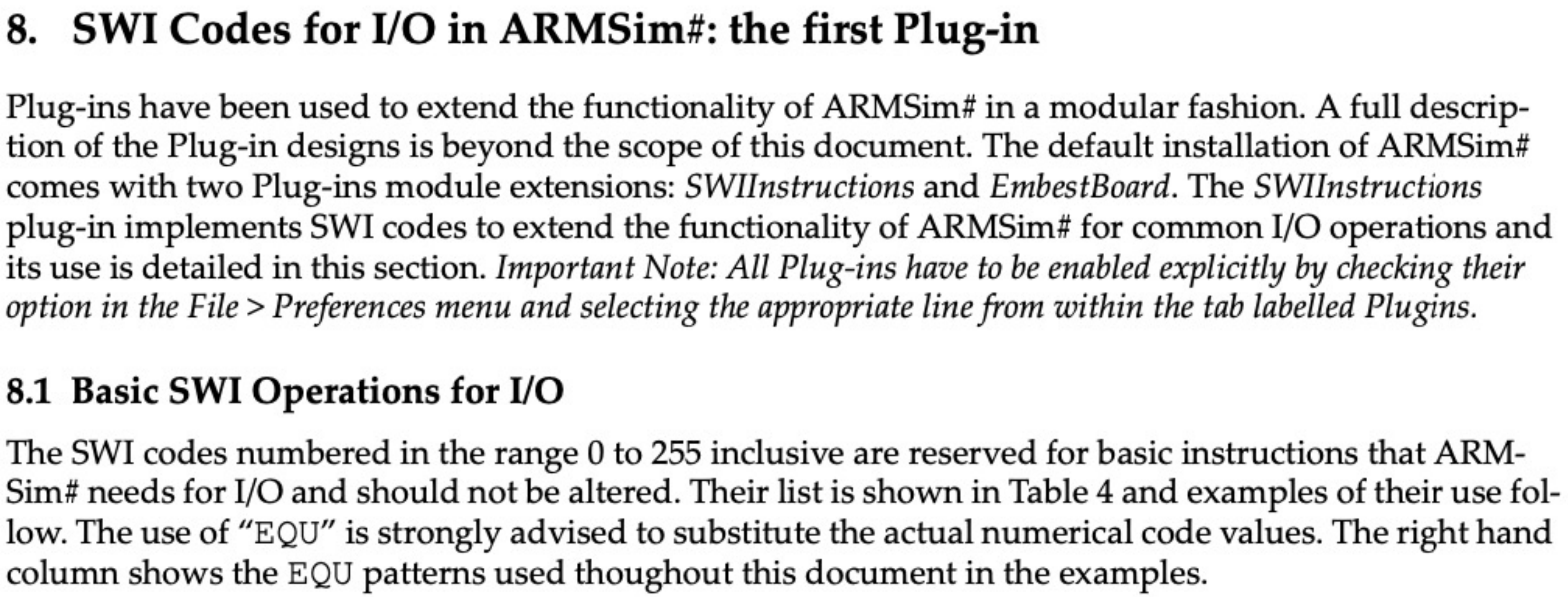
R10	sl	stack limit
R11	fp	frame pointer
R12	ip	intra-procedure-call scratch register
R13	sp	stack pointer
R14	lr	link register
R15	pc	program counter

Figure 4. General Purpose Registers View.

ARM Sim

Syscall <-> **SWI**

ARMSim# User Guide



8. SWI Codes for I/O in ARMSim#: the first Plug-in

Plug-ins have been used to extend the functionality of ARMSim# in a modular fashion. A full description of the Plug-in designs is beyond the scope of this document. The default installation of ARMSim# comes with two Plug-ins module extensions: *SWIInstructions* and *EmbestBoard*. The *SWIInstructions* plug-in implements SWI codes to extend the functionality of ARMSim# for common I/O operations and its use is detailed in this section. *Important Note: All Plug-ins have to be enabled explicitly by checking their option in the File > Preferences menu and selecting the appropriate line from within the tab labelled Plugins.*

8.1 Basic SWI Operations for I/O

The SWI codes numbered in the range 0 to 255 inclusive are reserved for basic instructions that ARMSim# needs for I/O and should not be altered. Their list is shown in Table 4 and examples of their use follow. The use of "`EQU`" is strongly advised to substitute the actual numerical code values. The right hand column shows the `EQU` patterns used thoughout this document in the examples.

ARM Sim

Syscall <-> SWI

ARMSim# User Guide

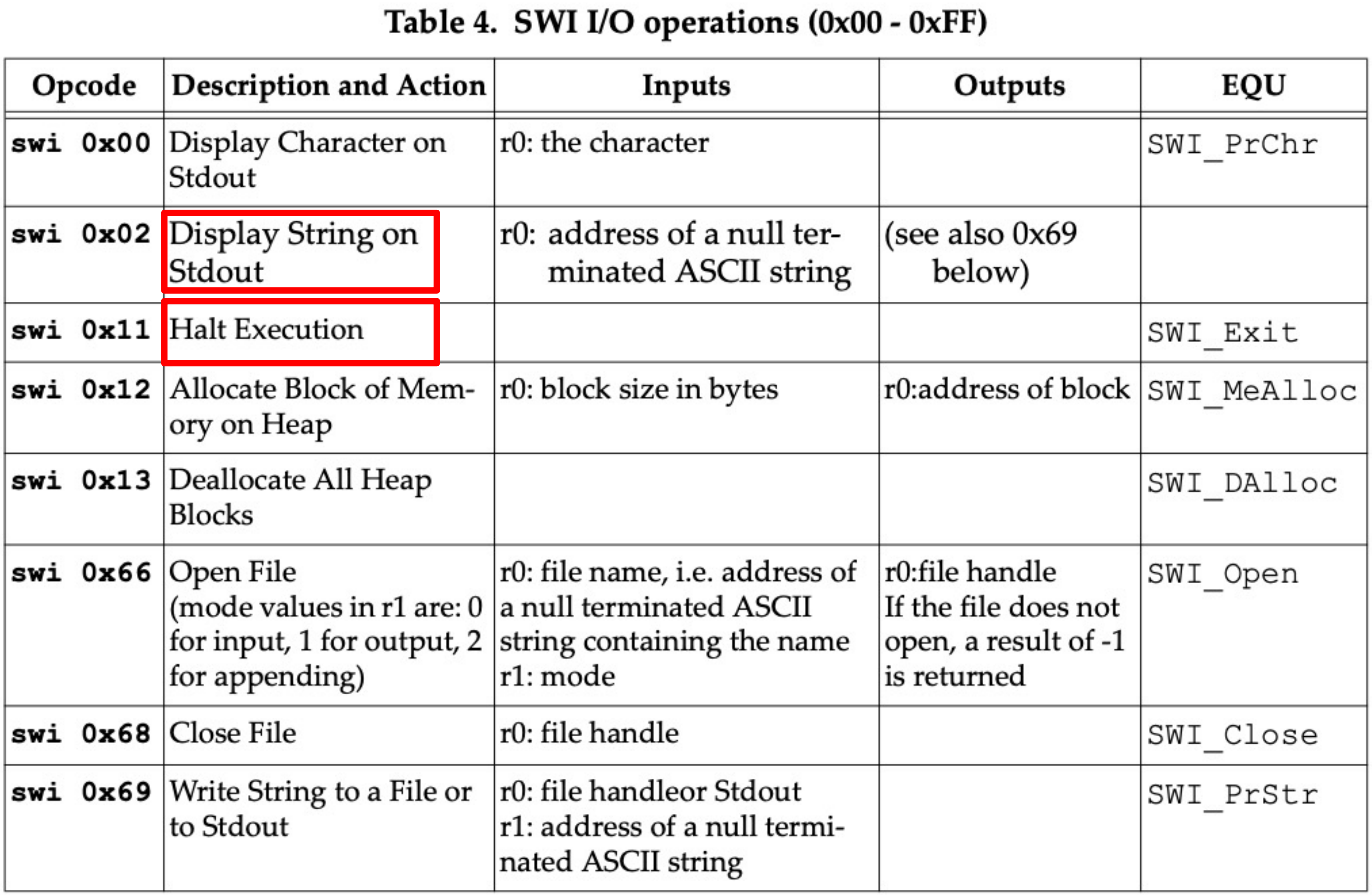


Table 4. SWI I/O operations (0x00 - 0xFF)

Opcode	Description and Action	Inputs	Outputs	EQU
`swi 0x00`	Display Character on Stdout	r0: the character		`SWI_PrChr`
`swi 0x02`	Display String on Stdout	r0: address of a null terminated ASCII string	(see also 0x69 below)	
`swi 0x11`	Halt Execution			`SWI_Exit`
`swi 0x12`	Allocate Block of Memory on Heap	r0: block size in bytes	r0:address of block	`SWI_MeAlloc`
`swi 0x13`	Deallocate All Heap Blocks			`SWI_DAlloc`
`swi 0x66`	Open File (mode values in r1 are: 0 for input, 1 for output, 2 for appending)	r0: file name, i.e. address of a null terminated ASCII string containing the name r1: mode	r0:file handle If the file does not open, a result of -1 is returned	`SWI_Open`
`swi 0x68`	Close File	r0: file handle		`SWI_Close`
`swi 0x69`	Write String to a File or to Stdout	r0: file handleor Stdout r1: address of a null terminated ASCII string		`SWI_PrStr`

ARM Sim

Breakpoint

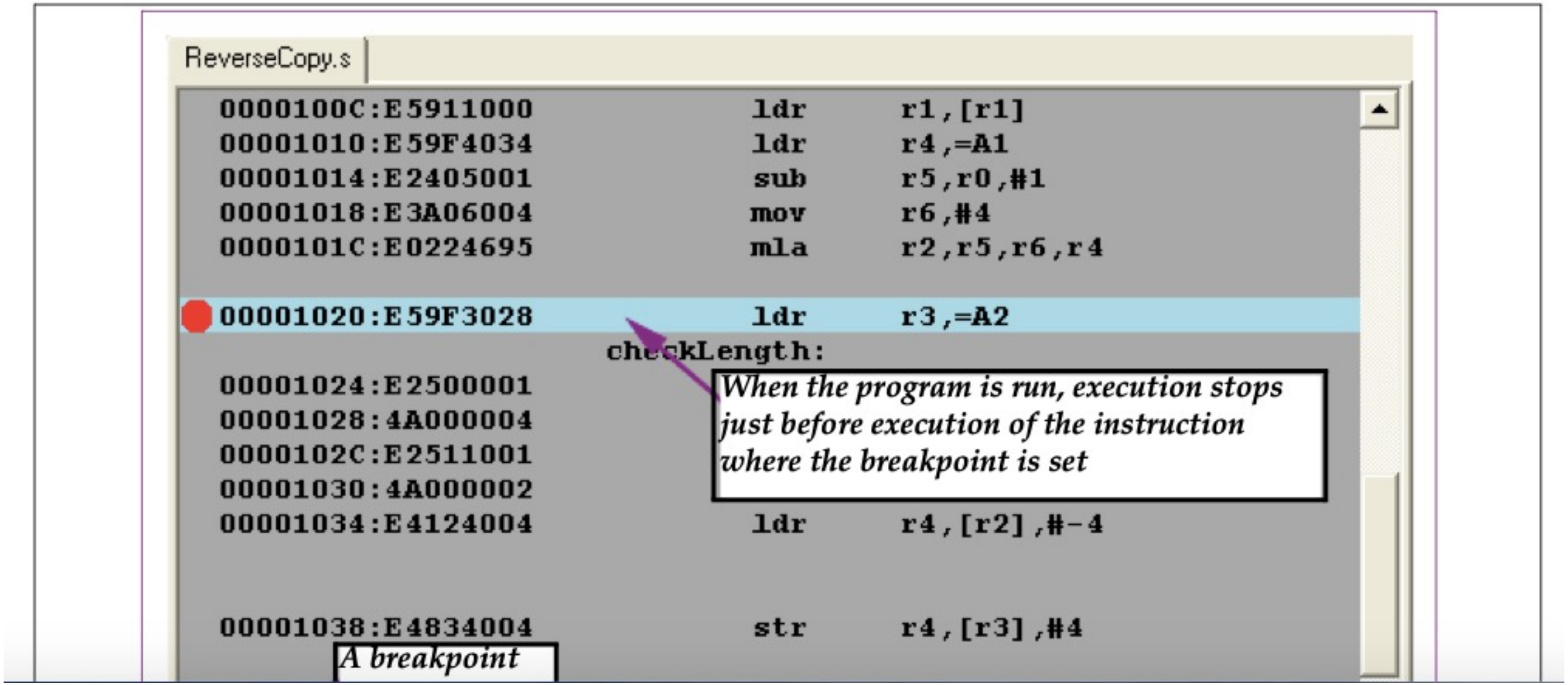
ARMSim# User Guide

To set a breakpoint, double-click the line of code, at which the bre ively, step through the code to the line, at which the breakpoint should be set, and then select **Debug > Toggle Breakpoint**. When the breakpoint is set, a large red dot appears in the **Code View** next to the address of the instruction at which the breakpoint was set.

To clear a breakpoint, double-click the line of code, at which the breakpoint is set. Alternatively, step through the code to the line, at which the breakpoint is set, and then select **Debug > Toggle Breakpoint**. To clear all of the breakpoints in a program, select **Debug > Clear All Breakpoints**.

Note:

- **Clear All Breakpoints** clears the breakpoints in *all* files that are currently open.

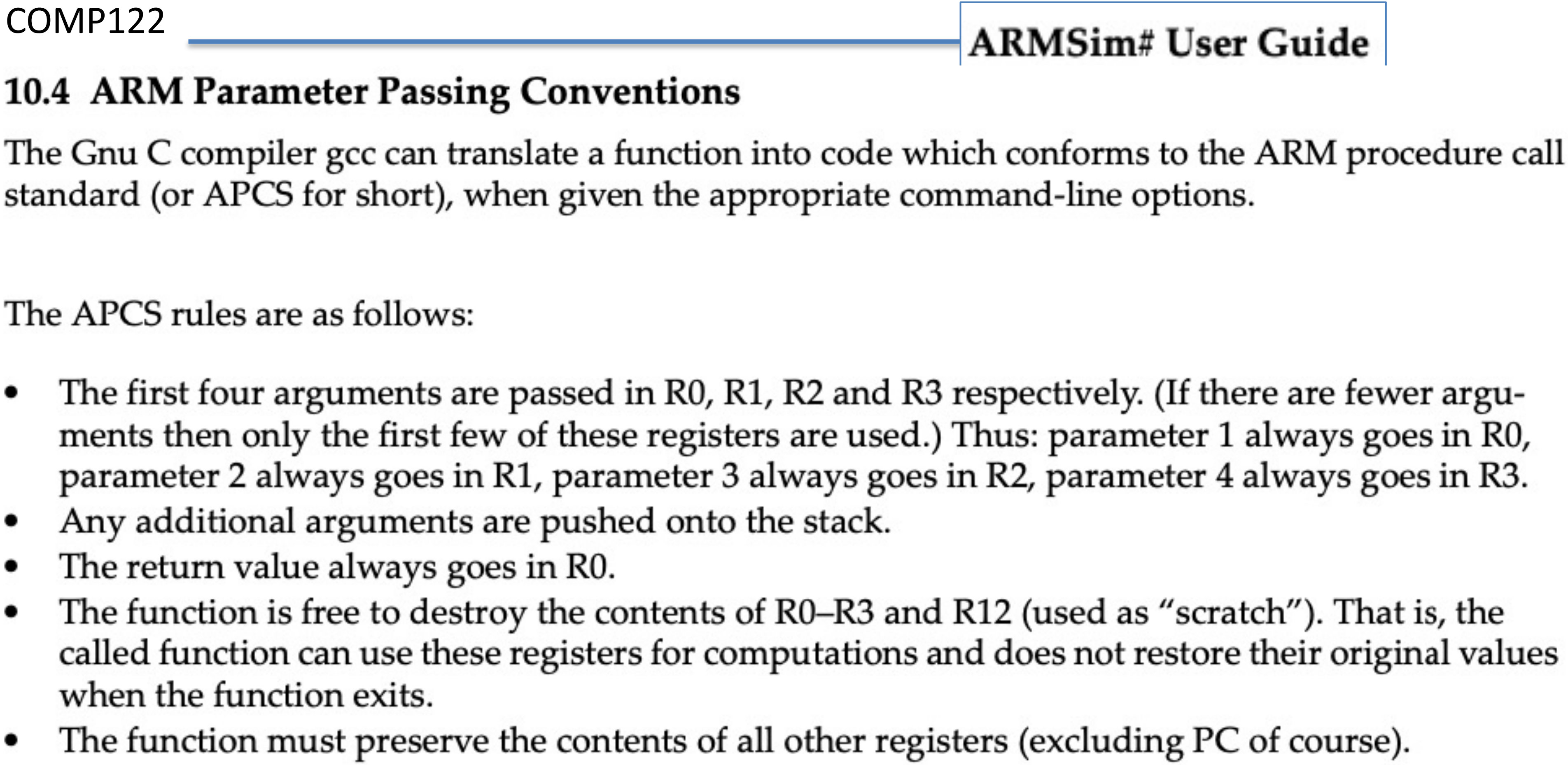


```
  0000100C:E5911000            ldr     r1,[r1]
  00001010:E59F4034            ldr     r4,=A1
  00001014:E2405001            sub     r5,r0,#1
  00001018:E3A06004            mov     r6,#4
  0000101C:E0224695            mla     r2,r5,r6,r4

● 00001020:E59F3028            ldr     r3,=A2
                    checkLength:
  00001024:E2500001
  00001028:4A000004
  0000102C:E2511001
  00001030:4A000002
  00001034:E4124004            ldr     r4,[r2],#-4


  00001038:E4834004            str     r4,[r3],#4
```

When the program is run, execution stops just before execution of the instruction where the breakpoint is set

A breakpoint

ARM Sim



ARMSim# User Guide

10.4 ARM Parameter Passing Conventions

The Gnu C compiler gcc can translate a function into code which conforms to the ARM procedure call standard (or APCS for short), when given the appropriate command-line options.

The APCS rules are as follows:

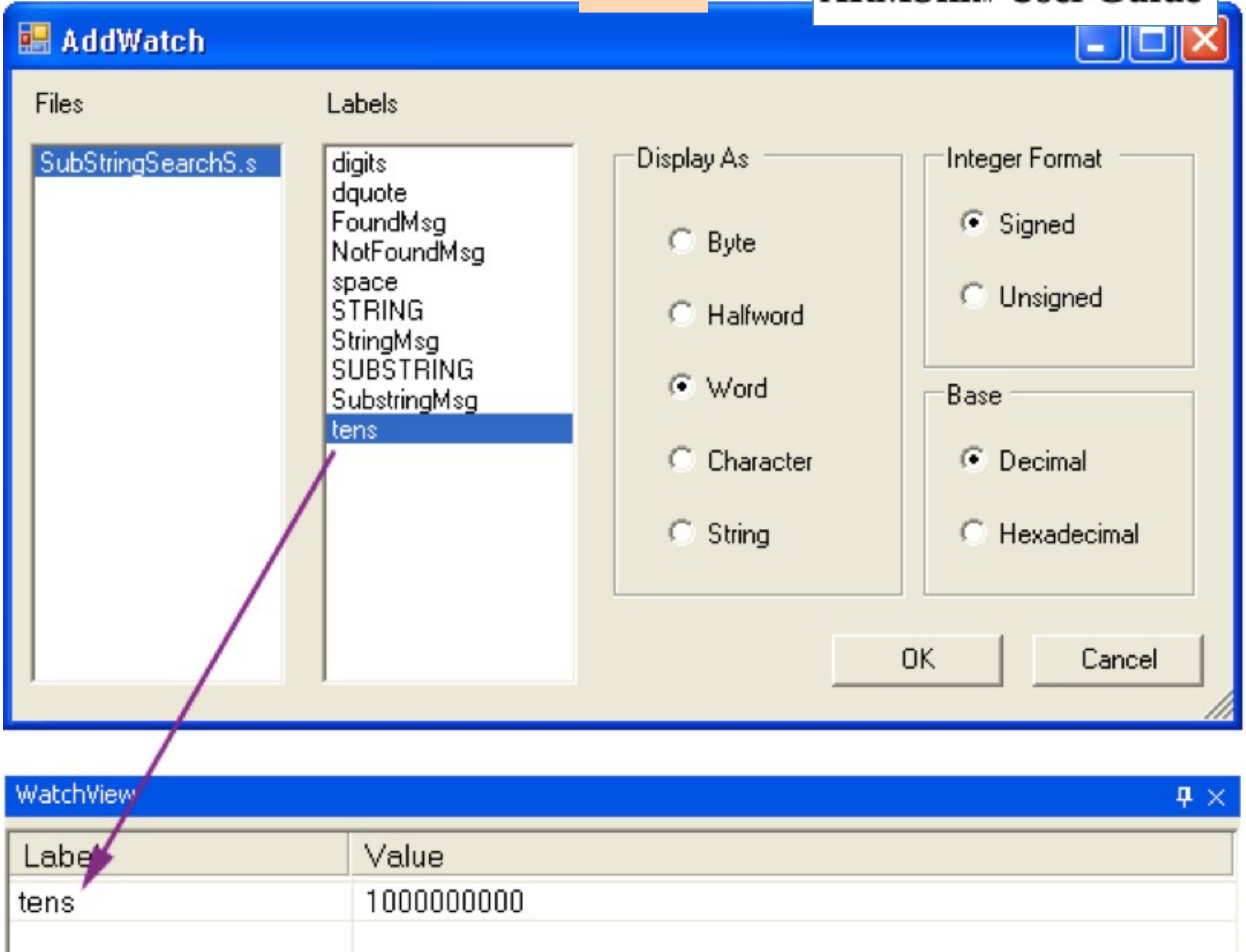
- The first four arguments are passed in R0, R1, R2 and R3 respectively. (If there are fewer arguments then only the first few of these registers are used.) Thus: parameter 1 always goes in R0, parameter 2 always goes in R1, parameter 3 always goes in R2, parameter 4 always goes in R3.
- Any additional arguments are pushed onto the stack.
- The return value always goes in R0.
- The function is free to destroy the contents of R0–R3 and R12 (used as "scratch"). That is, the called function can use these registers for computations and does not restore their original values when the function exits.
- The function must preserve the contents of all other registers (excluding PC of course).

- ❖ **MIPS** = $a0-3 → $v0-1
- ❖ **ARM** = R0-3 → R0

ARM Sim

Watch

ARMSim# User Guide



AddWatch

Files
SubStringSearchS.s

Labels
digits
dquote
FoundMsg
NotFoundMsg
space
STRING
StringMsg
SUBSTRING
SubstringMsg
tens

Display As
○ Byte
○ Halfword
● Word
○ Character
○ String

Integer Format
● Signed
○ Unsigned

Base
● Decimal
○ Hexadecimal

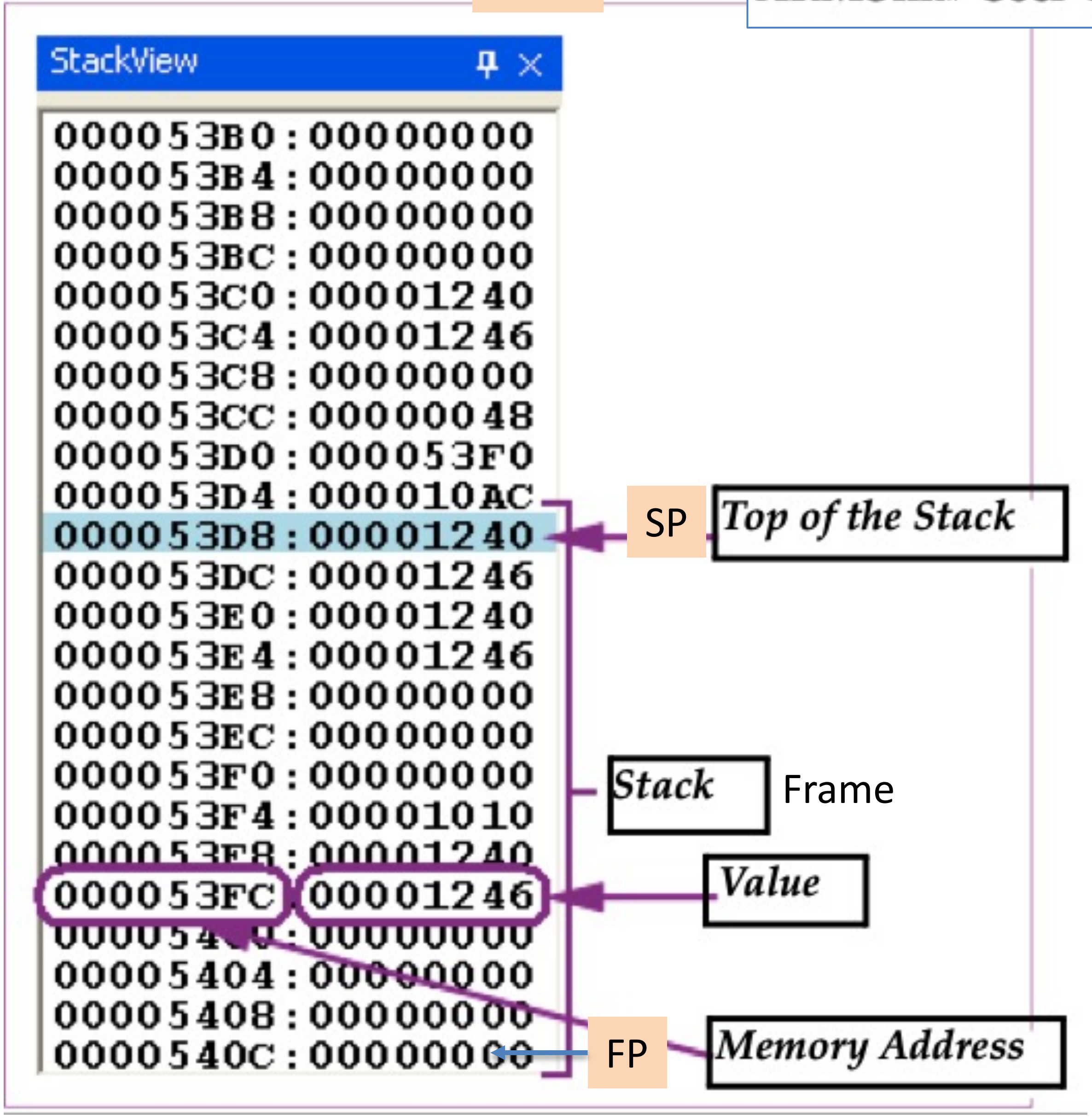
WatchView

Label	Value
tens	1000000000

ARM Sim

Stack

ARMSim# User Guide



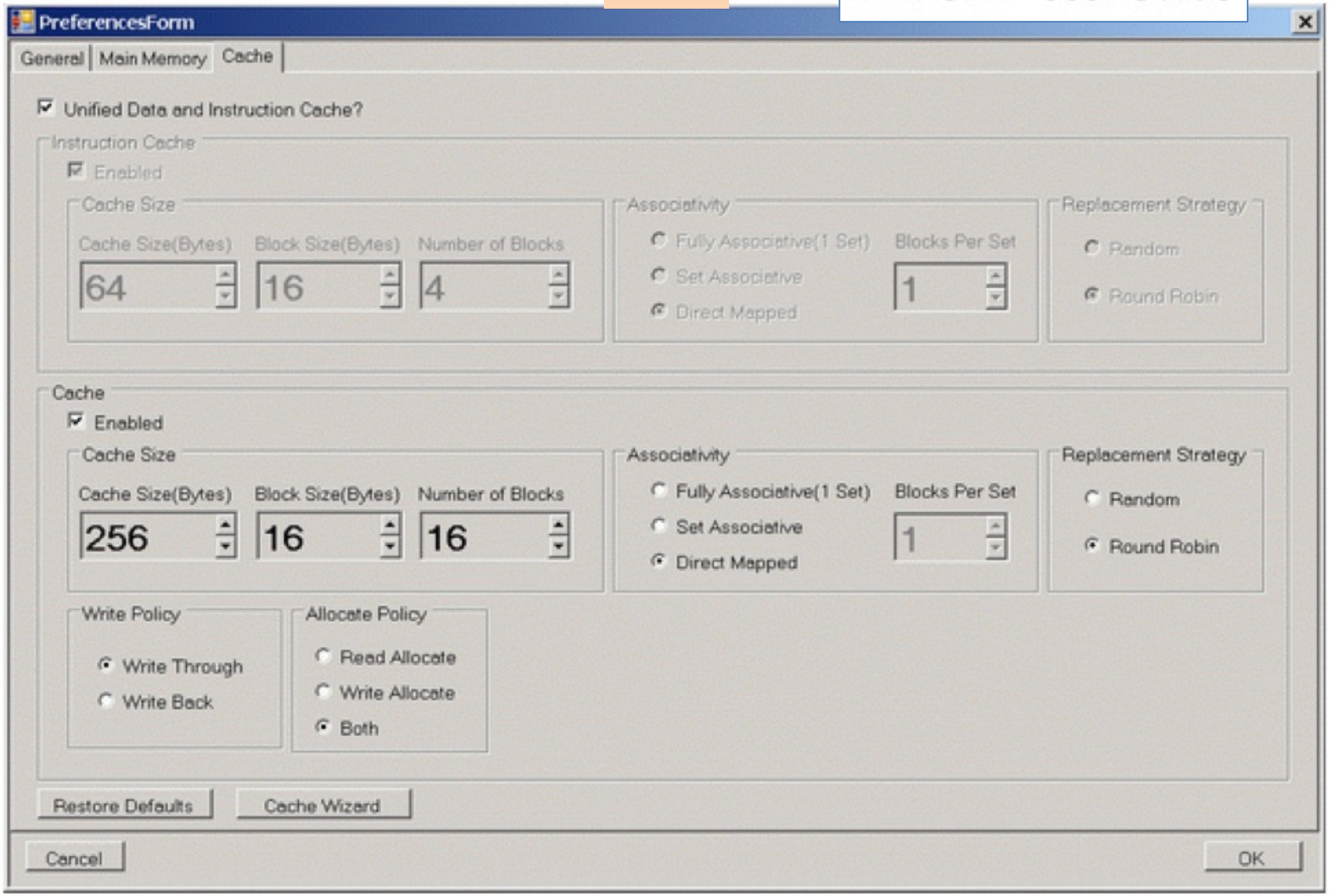
StackView

```
000053B0:00000000
000053B4:00000000
000053B8:00000000
000053BC:00000000
000053C0:00001240
000053C4:00001246
000053C8:00000000
000053CC:00000048
000053D0:000053F0
000053D4:000010AC
000053D8:00001240
000053DC:00001246
000053E0:00001240
000053E4:00001246
000053E8:00000000
000053EC:00000000
000053F0:00000000
000053F4:00001010
000053F8:00001240
000053FC:00001246
00005400:00000000
00005404:00000000
00005408:00000000
0000540C:00000000
```

ARM Sim

Cache

ARMSim# User Guide



PreferencesForm

General | Main Memory | Cache

☑ Unified Data and Instruction Cache?

Instruction Cache

☑ Enabled

Cache Size

Cache Size(Bytes)	Block Size(Bytes)	Number of Blocks
64	16	4

Associativity

○ Fully Associative(1 Set)
○ Set Associative
● Direct Mapped

Blocks Per Set: 1

Replacement Strategy

○ Random
● Round Robin

Cache

☑ Enabled

Cache Size

Cache Size(Bytes)	Block Size(Bytes)	Number of Blocks
256	16	16

Associativity

○ Fully Associative(1 Set)
○ Set Associative
● Direct Mapped

Blocks Per Set: 1

Replacement Strategy

○ Random
● Round Robin

Write Policy

● Write Through
○ Write Back

Allocate Policy

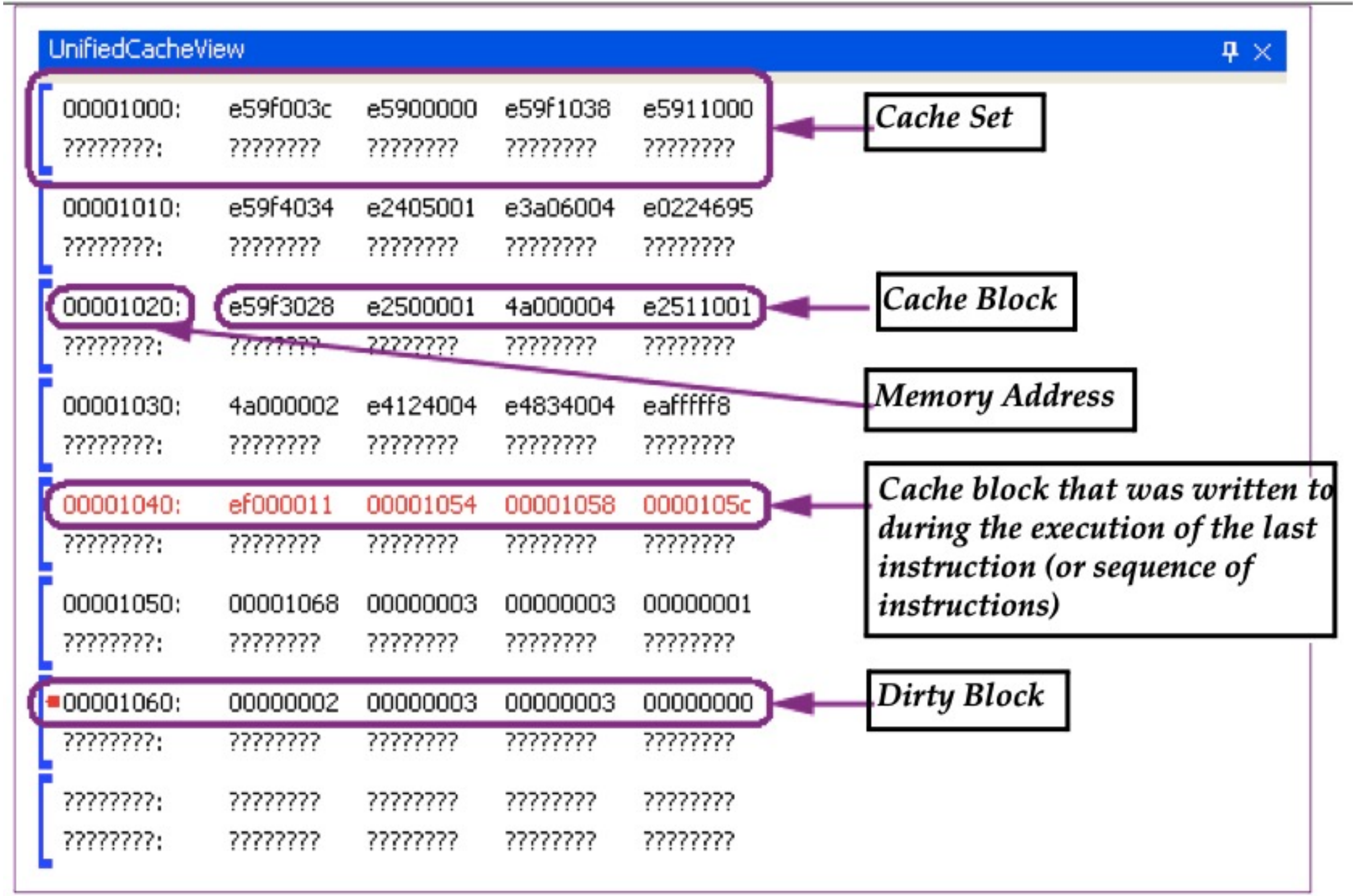
○ Read Allocate
○ Write Allocate
● Both

Figure 12. Cache Preferences Form.

ARM Sim

Cache

ARMSim# User Guide



UnifiedCacheView

```
00001000:   e59f003c   e5900000   e59f1038   e5911000
????????:   ????????   ????????   ????????   ????????
00001010:   e59f4034   e2405001   e3a06004   e0224695
????????:   ????????   ????????   ????????   ????????
00001020:   e59f3028   e2500001   4a000004   e2511001
????????:   ????????   ????????   ????????   ????????
00001030:   4a000002   e4124004   e4834004   eafffff8
????????:   ????????   ????????   ????????   ????????
00001040:   ef000011   00001054   00001058   0000105c
????????:   ????????   ????????   ????????   ????????
00001050:   00001068   00000003   00000003   00000001
????????:   ????????   ????????   ????????   ????????
00001060:   00000002   00000003   00000003   00000000
????????:   ????????   ????????   ????????   ????????
????????:   ????????   ????????   ????????   ????????
????????:   ????????   ????????   ????????   ????????
```

Cache Set

Cache Block

Memory Address

Cache block that was written to during the execution of the last instruction (or sequence of instructions)

Dirty Block

Figure 13. Cache View.

ARM

ARM Assembly

ARM Assembly

ALU

ARM Book

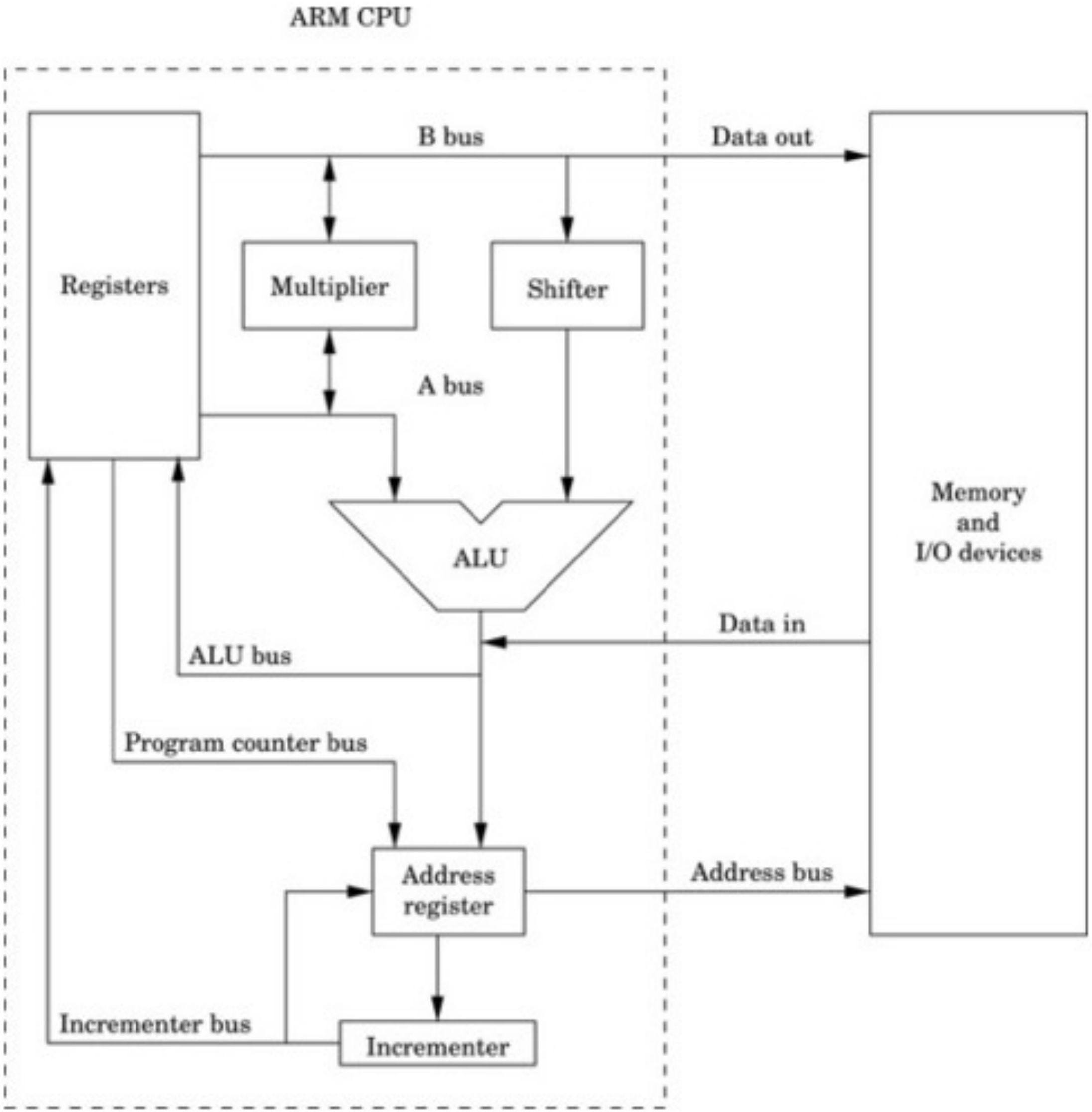


FIGURE 3.1 The ARM processor architecture.

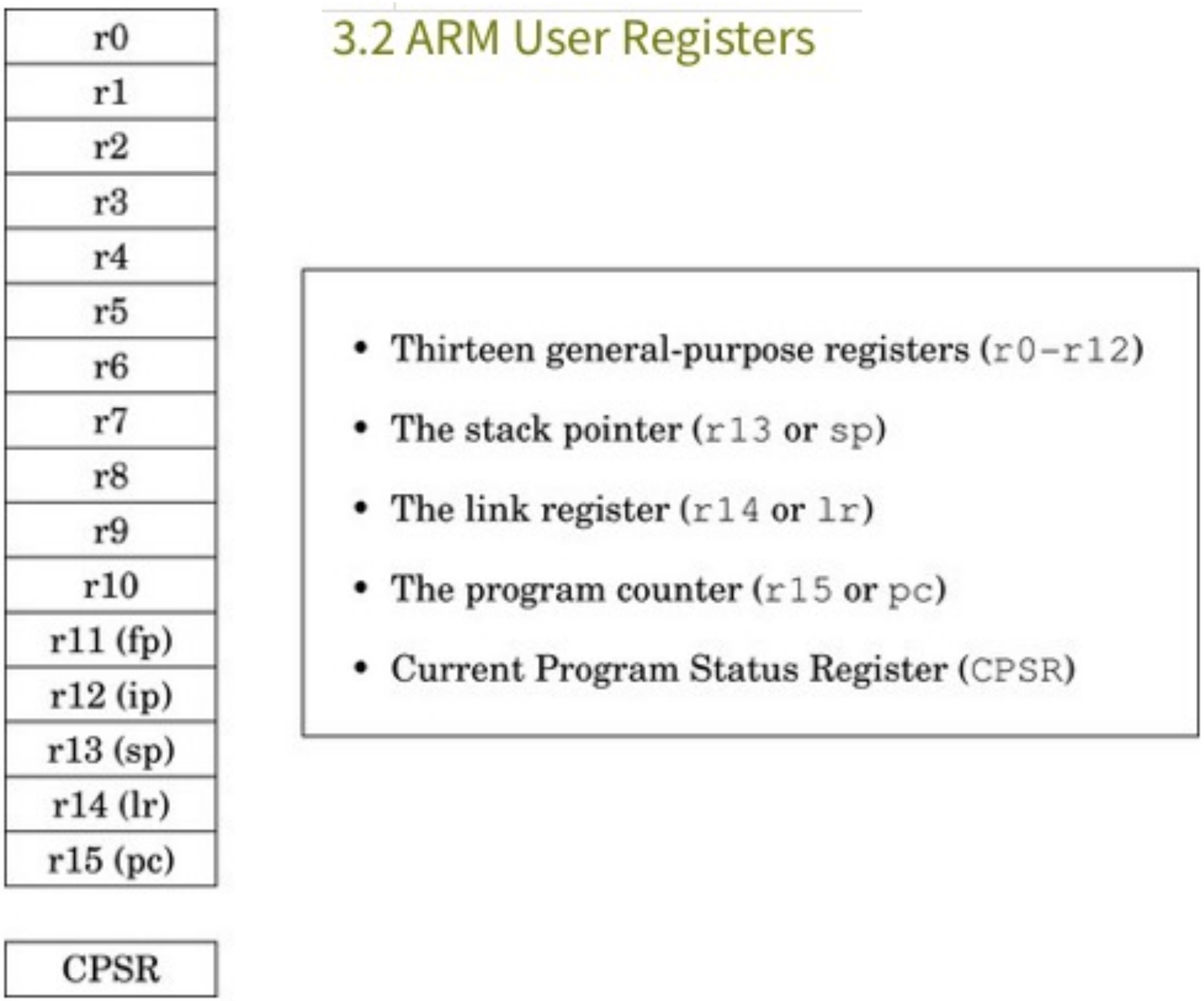
CSUN
CALIFORNIA
STATE UNIVERSITY
NORTHRIDGE
COMP122

ARM Assembly

Registers

ARM Book

3.2 ARM User Registers



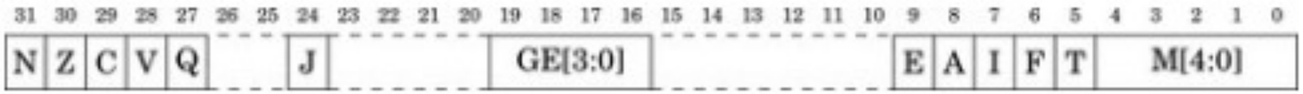
- Thirteen general-purpose registers (r0–r12)
- The stack pointer (r13 or sp)
- The link register (r14 or lr)
- The program counter (r15 or pc)
- Current Program Status Register (CPSR)

FIGURE 3.2 The ARM user program registers.

ARM Assembly

PSR: flags

ARM Book



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
N	Z	C	V	Q			J					GE[3:0]										E	A	I	F	T	M[4:0]				

FIGURE 3.3 The ARM process status register.

Negative: This bit is set to one if the signed result of an operation is negative, and set to zero if the result is positive or zero.

Zero: This bit is set to one if the result of an operation is zero, and set to zero if the result is non-zero.

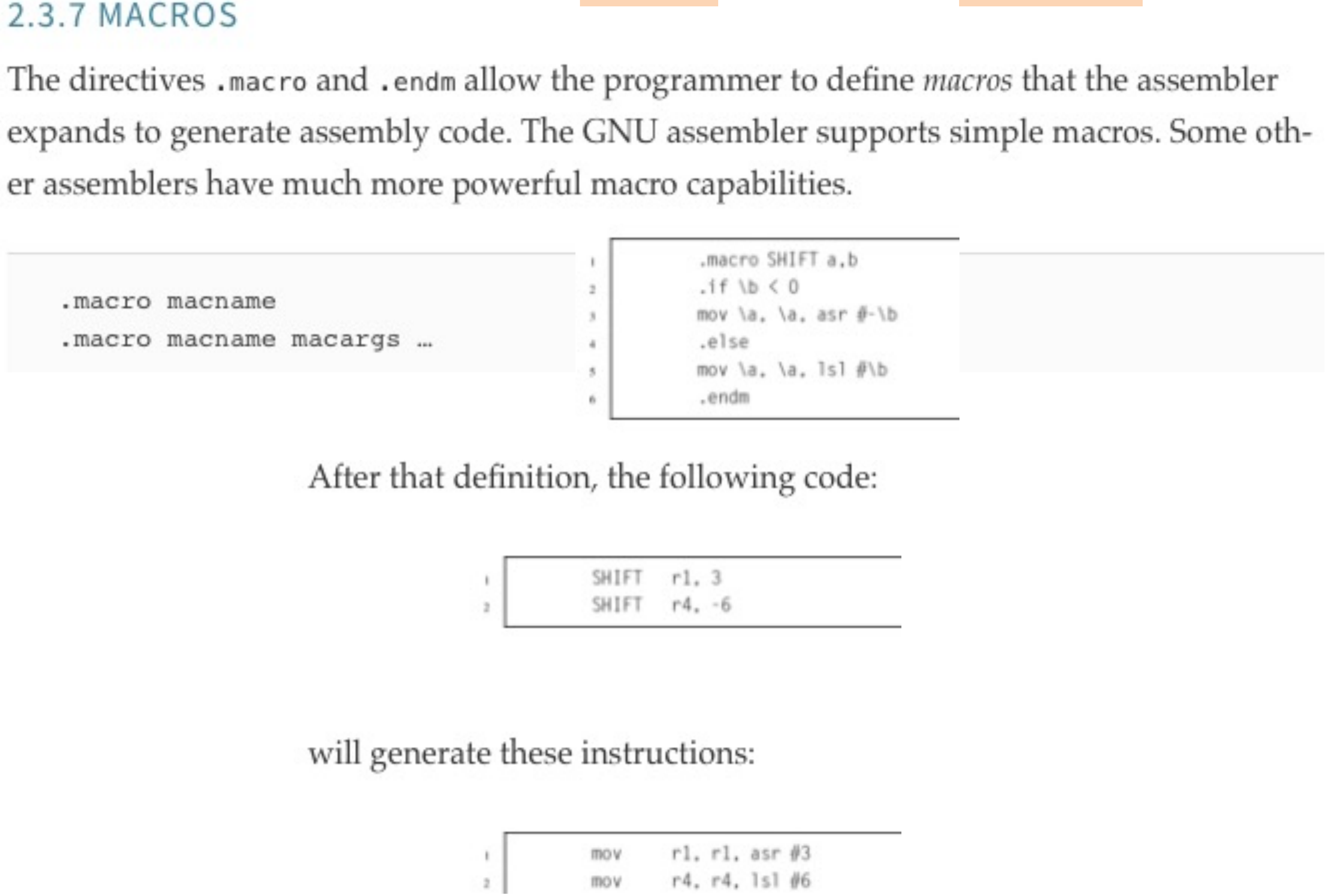
Carry: This bit is set to one if an add operation results in a carry out of the most significant bit, or if a subtract operation results in a borrow. For shift operations, this flag is set to the last bit shifted out by the shifter.

oVerflow: For addition and subtraction, this flag is set if a signed overflow occurred.

ARM Assembly

Macros

ARM Book



2.3.7 MACROS

The directives `.macro` and `.endm` allow the programmer to define *macros* that the assembler expands to generate assembly code. The GNU assembler supports simple macros. Some other assemblers have much more powerful macro capabilities.

```
.macro macname
.macro macname macargs …
```

```
        .macro SHIFT a,b
        .if \b < 0
        mov \a, \a, asr #-\b
        .else
        mov \a, \a, lsl #\b
        .endm
```

After that definition, the following code:

```
        SHIFT   r1, 3
        SHIFT   r4, -6
```

will generate these instructions:

```
        mov     r1, r1, asr #3
        mov     r4, r4, lsl #6
```

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE

DR JEFF SOFTWARE
INDIE APP DEVELOPER
© Jeff Drobman
2016-2022

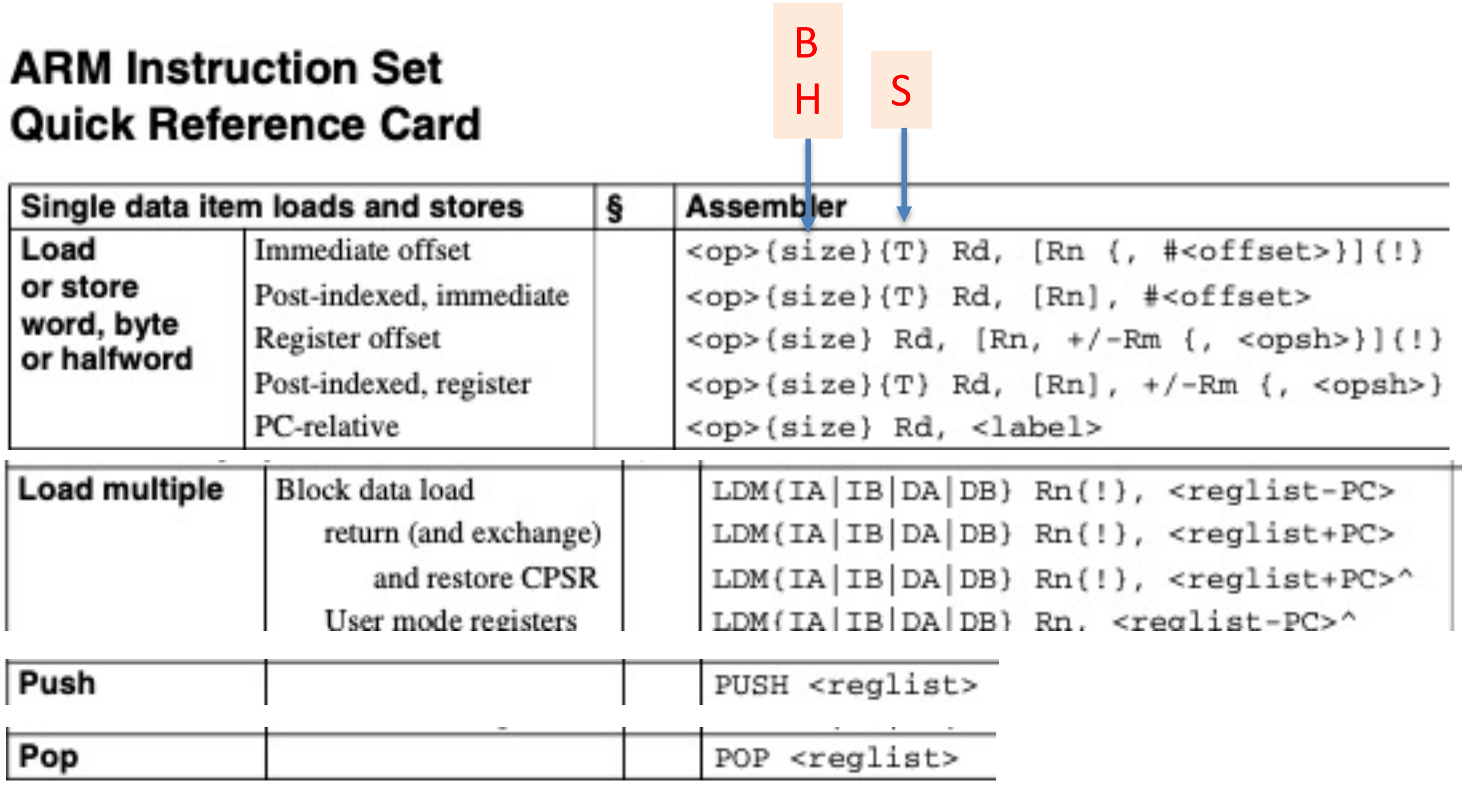
ARM Assembly

COMP122 ARM v7 ISA

ARM Ref

Load/store:

LDR (b, h, w)
STR (b, h, w)
LDM{IA} [load multiple]
STM [store multiple]
SWP ((b, w) [swap]
PUSH/POP



ARM Instruction Set
Quick Reference Card

Single data item loads and stores		§	Assembler
Load or store word, byte or halfword	Immediate offset		`<op>{size}{T} Rd, [Rn {, #<offset>}]{!}`
	Post-indexed, immediate		`<op>{size}{T} Rd, [Rn], #<offset>`
	Register offset		`<op>{size} Rd, [Rn, +/-Rm {, <opsh>}]{!}`
	Post-indexed, register		`<op>{size}{T} Rd, [Rn], +/-Rm {, <opsh>}`
	PC-relative		`<op>{size} Rd, <label>`

Load multiple	Block data load		`LDM{IA|IB|DA|DB} Rn{!}, <reglist-PC>`
	return (and exchange)		`LDM{IA|IB|DA|DB} Rn{!}, <reglist+PC>`
	and restore CPSR		`LDM{IA|IB|DA|DB} Rn{!}, <reglist+PC>^`
	User mode registers		`LDM{IA|IB|DA|DB} Rn, <reglist-PC>^`

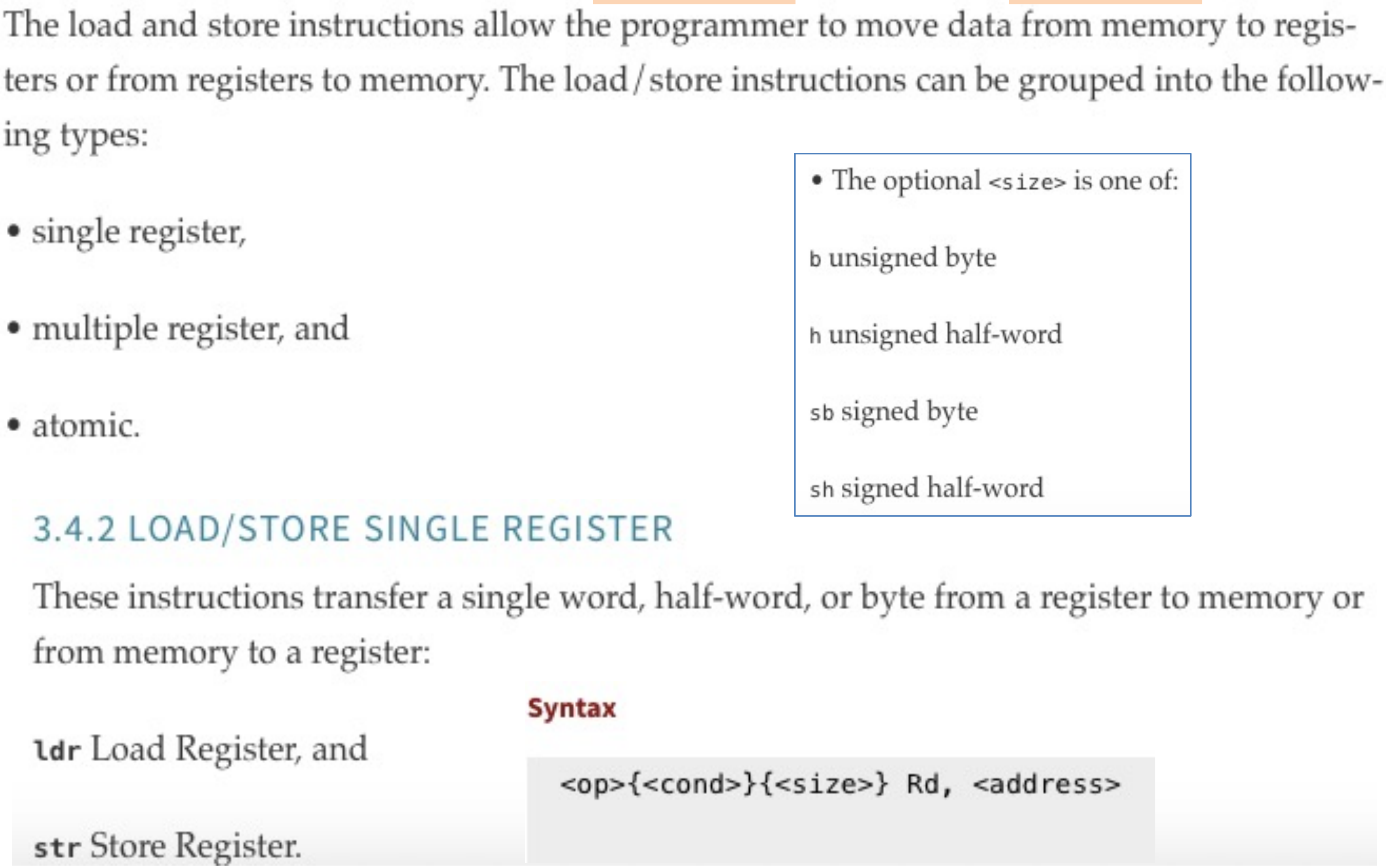
Push			`PUSH <reglist>`

Pop			`POP <reglist>`

ARM Assembly

Load/Store

ARM Book



The load and store instructions allow the programmer to move data from memory to registers or from registers to memory. The load / store instructions can be grouped into the following types:

- single register,
- multiple register, and
- atomic.

- The optional `<size>` is one of:

`b` unsigned byte

`h` unsigned half-word

`sb` signed byte

`sh` signed half-word

3.4.2 LOAD/STORE SINGLE REGISTER

These instructions transfer a single word, half-word, or byte from a register to memory or from memory to a register:

`ldr` Load Register, and

`str` Store Register.

Syntax

```
<op>{<cond>}{<size>} Rd, <address>
```

COMP122

ARM Assembly

Load/Store

ARM Book

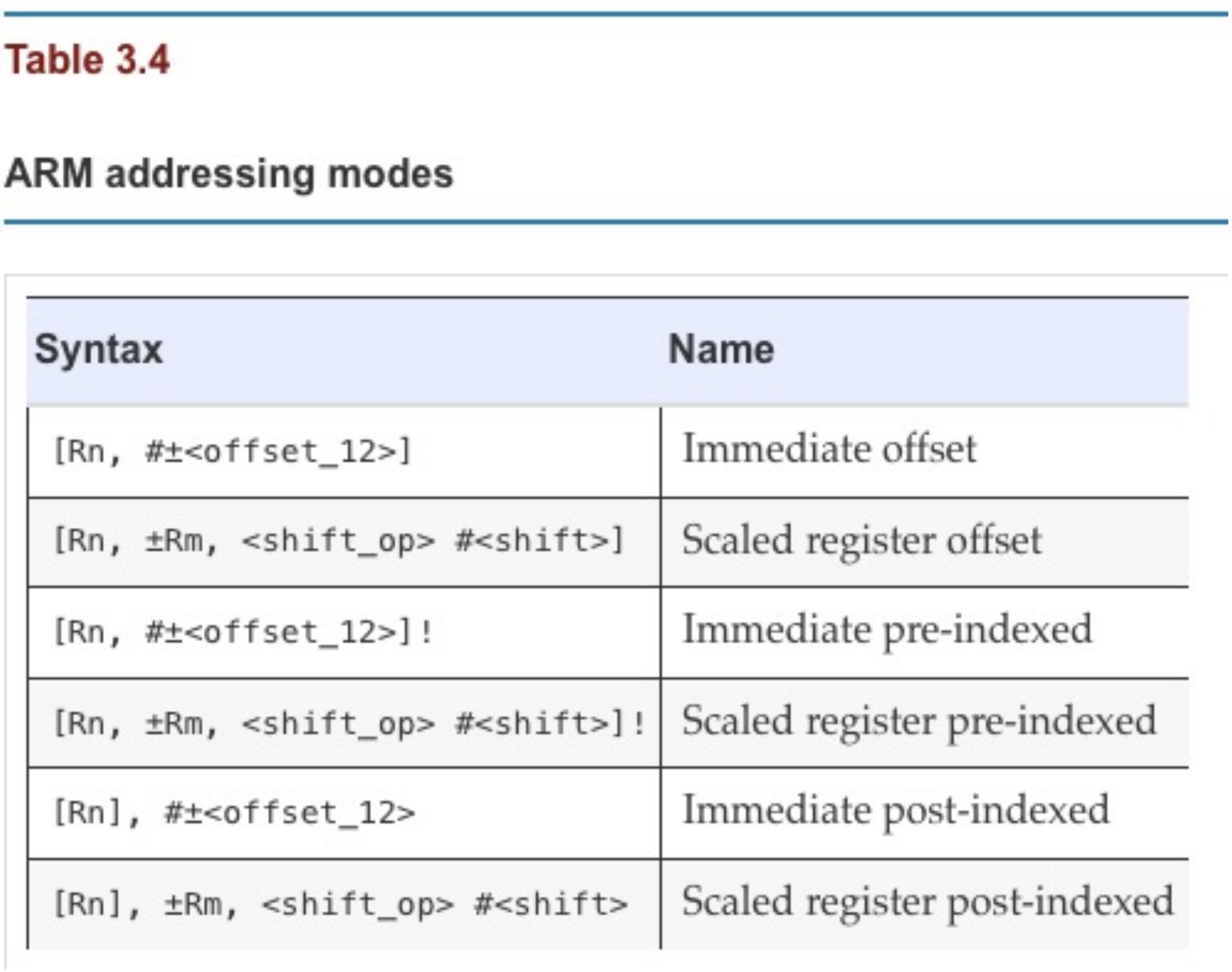


Table 3.4

ARM addressing modes

Syntax	Name
`[Rn, #±<offset_12>]`	Immediate offset
`[Rn, ±Rm, <shift_op> #<shift>]`	Scaled register offset
`[Rn, #±<offset_12>]!`	Immediate pre-indexed
`[Rn, ±Rm, <shift_op> #<shift>]!`	Scaled register pre-indexed
`[Rn], #±<offset_12>`	Immediate post-indexed
`[Rn], ±Rm, <shift_op> #<shift>`	Scaled register post-indexed

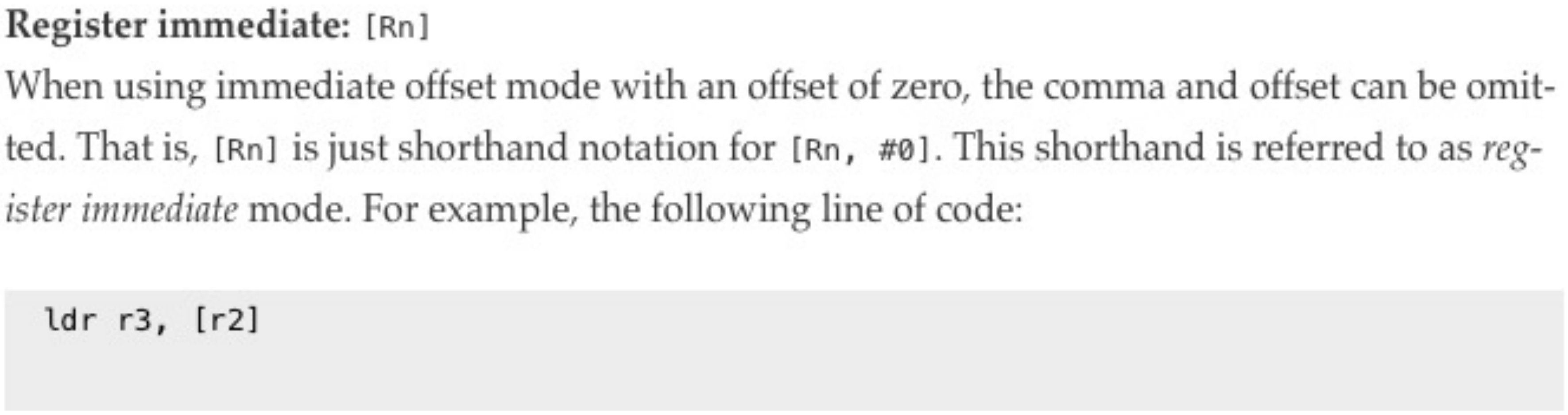
CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP122

ARM Assembly

DR JEFF SOFTWARE
INDIE APP DEVELOPER

Load/Store

ARM Book

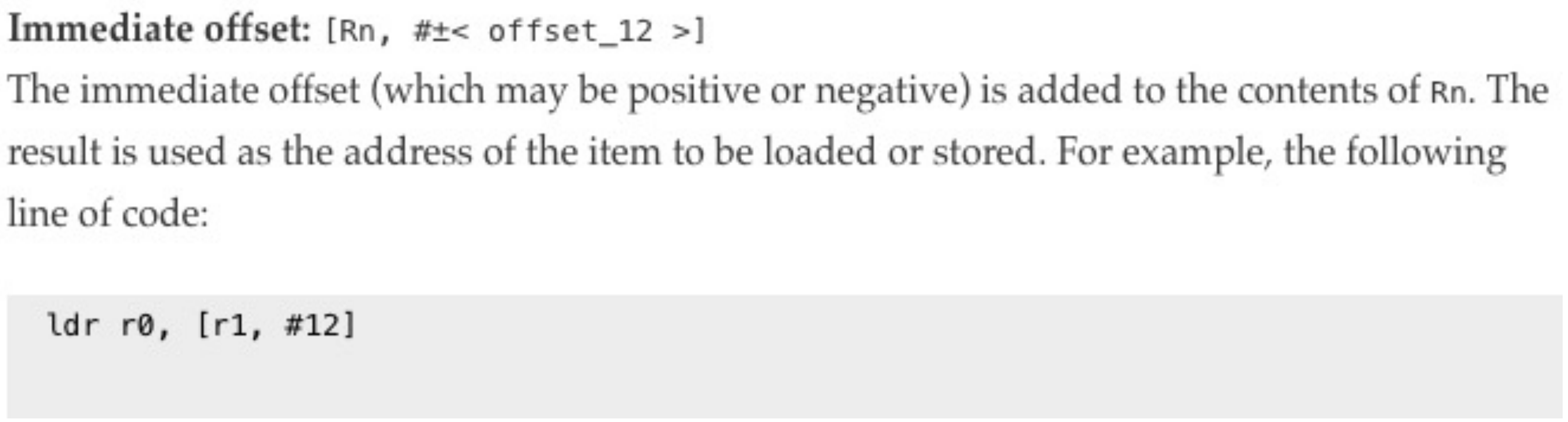


Register immediate: [Rn]

When using immediate offset mode with an offset of zero, the comma and offset can be omitted. That is, [Rn] is just shorthand notation for [Rn, #0]. This shorthand is referred to as *register immediate* mode. For example, the following line of code:

```
ldr r3, [r2]
```

ldr



Immediate offset: [Rn, #±< offset_12 >]

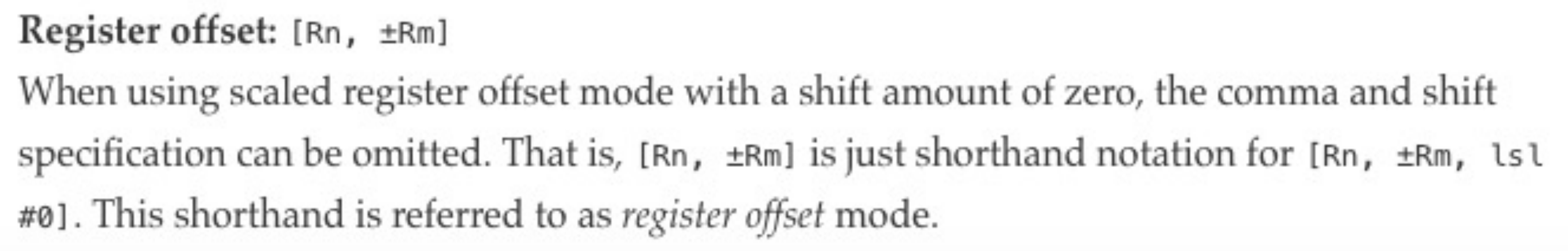
The immediate offset (which may be positive or negative) is added to the contents of Rn. The result is used as the address of the item to be loaded or stored. For example, the following line of code:

```
ldr r0, [r1, #12]
```

ARM Assembly

Load/Store

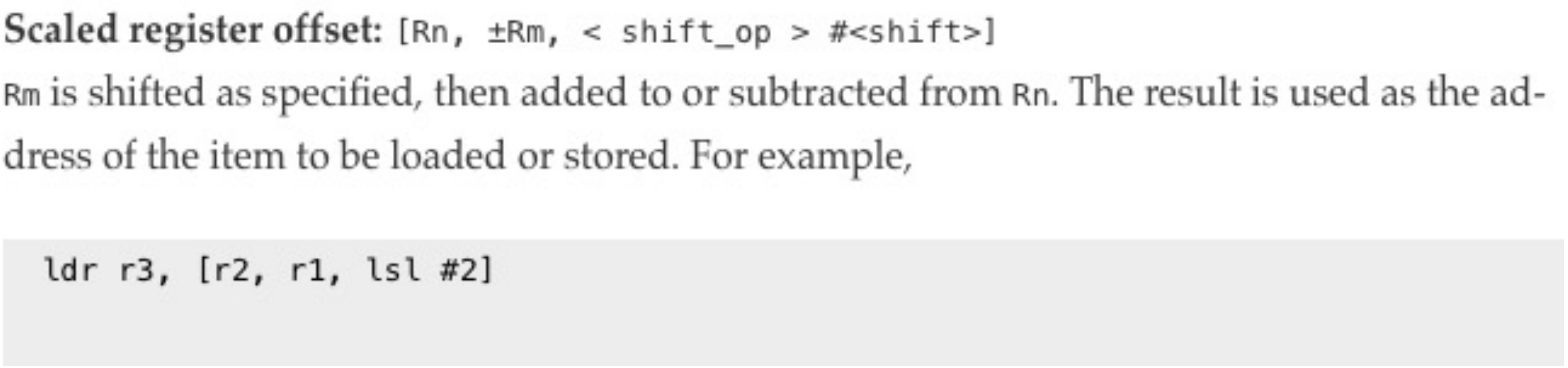
ARM Book



Register offset: `[Rn, ±Rm]`

When using scaled register offset mode with a shift amount of zero, the comma and shift specification can be omitted. That is, `[Rn, ±Rm]` is just shorthand notation for `[Rn, ±Rm, lsl #0]`. This shorthand is referred to as *register offset* mode.

Offsets to Eff. Address



Scaled register offset: `[Rn, ±Rm, < shift_op > #<shift>]`

`Rm` is shifted as specified, then added to or subtracted from `Rn`. The result is used as the address of the item to be loaded or stored. For example,

```
ldr r3, [r2, r1, lsl #2]
```

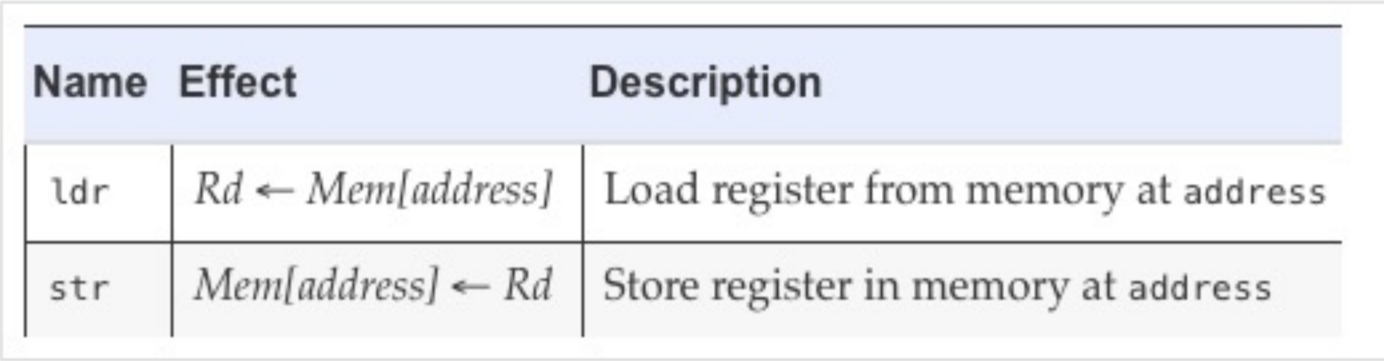
May be shifted (scaled)

ARM Assembly

Load/Store

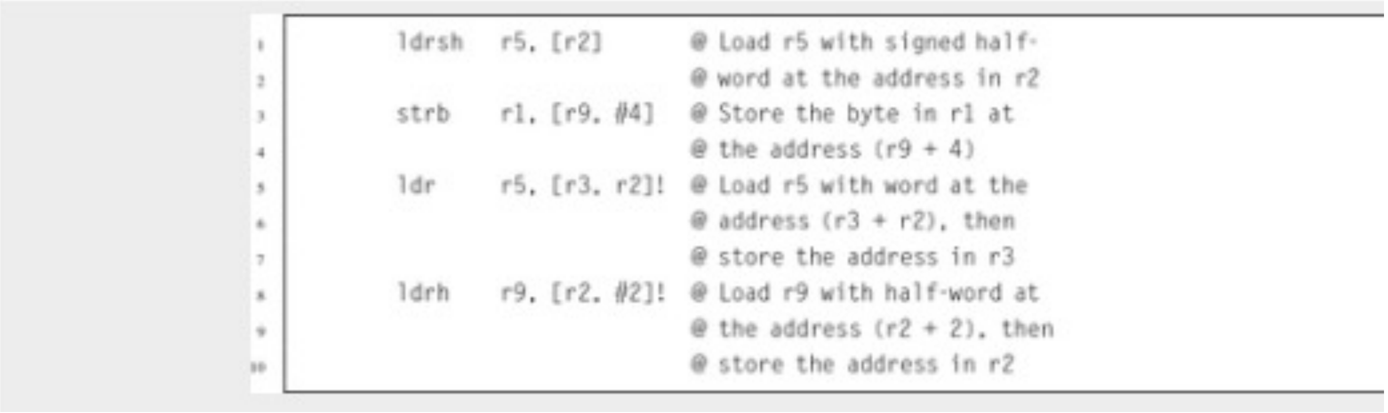
ARM Book

Operations



Name	Effect	Description
ldr	*Rd* ← *Mem[address]*	Load register from memory at address
str	*Mem[address]* ← *Rd*	Store register in memory at address

Examples



```
ldrsh   r5, [r2]        @ Load r5 with signed half-
                        @ word at the address in r2
strb    r1, [r9, #4]    @ Store the byte in r1 at
                        @ the address (r9 + 4)
ldr     r5, [r3, r2]!   @ Load r5 with word at the
                        @ address (r3 + r2), then
                        @ store the address in r3
ldrh    r9, [r2, #2]!   @ Load r9 with half-word at
                        @ the address (r2 + 2), then
                        @ store the address in r2
```

ARM Assembly

Load/Store

ARM Book

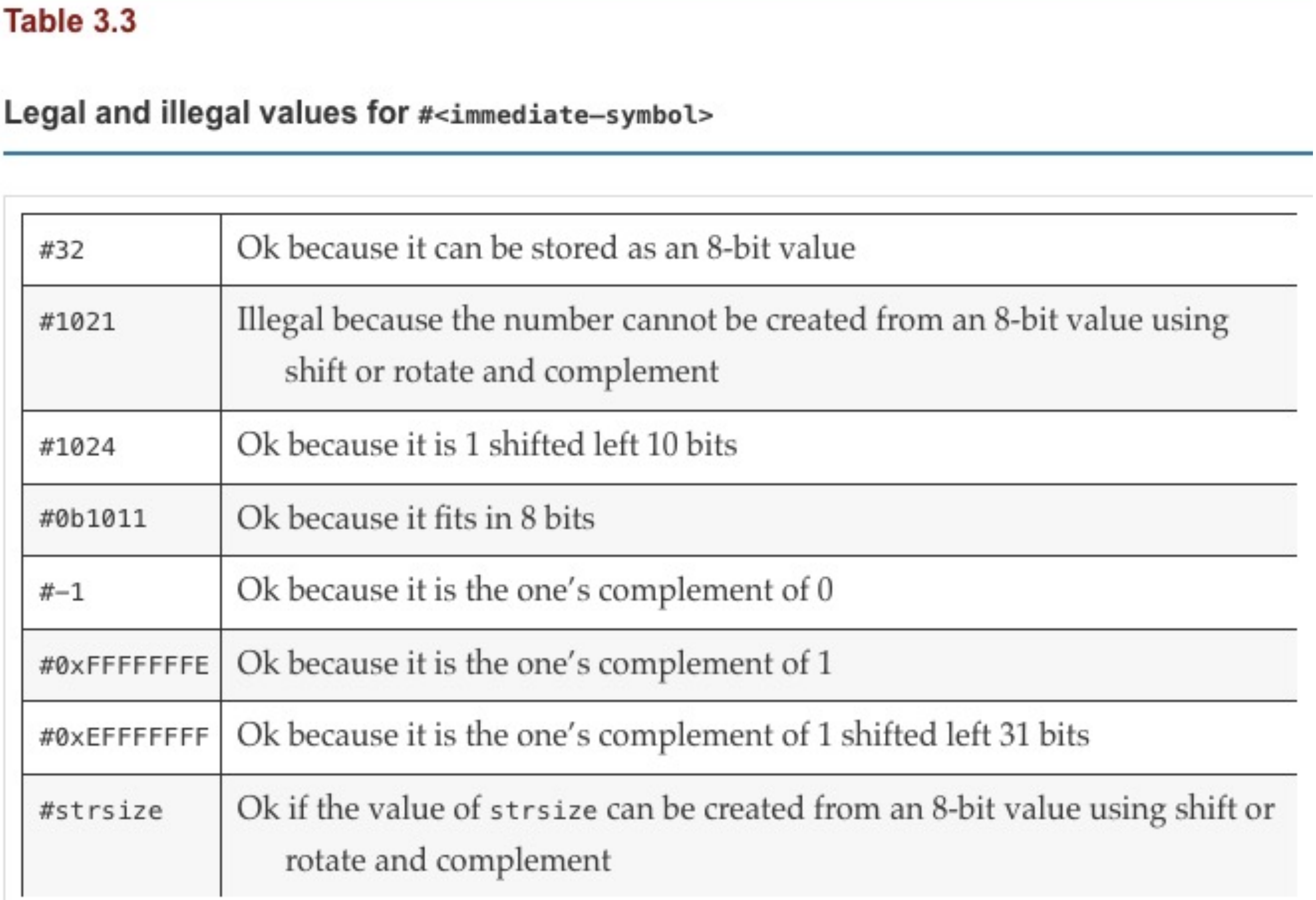


Table 3.3

Legal and illegal values for #<immediate-symbol>

#32	Ok because it can be stored as an 8-bit value
#1021	Illegal because the number cannot be created from an 8-bit value using shift or rotate and complement
#1024	Ok because it is 1 shifted left 10 bits
#0b1011	Ok because it fits in 8 bits
#-1	Ok because it is the one's complement of 0
#0xFFFFFFFE	Ok because it is the one's complement of 1
#0xEFFFFFFF	Ok because it is the one's complement of 1 shifted left 31 bits
#strsize	Ok if the value of strsize can be created from an 8-bit value using shift or rotate and complement

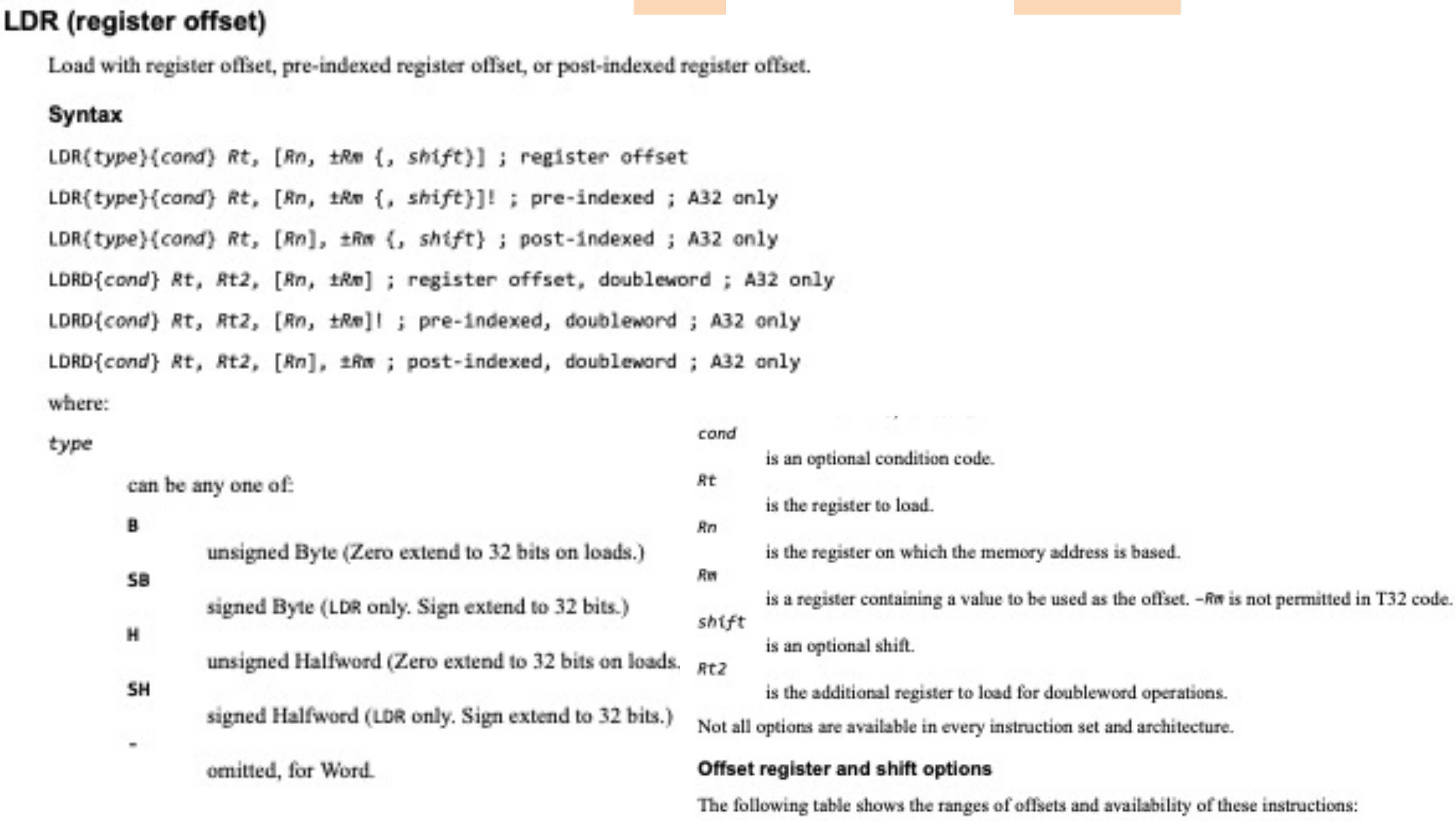
CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP122

ARM Assembly

DR JEFF SOFTWARE
INDIE APP DEVELOPER
© Jeff Drobman
2016-2022

LDR

ARM Ref



LDR (register offset)

Load with register offset, pre-indexed register offset, or post-indexed register offset.

Syntax

```
LDR{type}{cond} Rt, [Rn, ±Rm {, shift}] ; register offset
LDR{type}{cond} Rt, [Rn, ±Rm {, shift}]! ; pre-indexed ; A32 only
LDR{type}{cond} Rt, [Rn], ±Rm {, shift} ; post-indexed ; A32 only
LDRD{cond} Rt, Rt2, [Rn, ±Rm] ; register offset, doubleword ; A32 only
LDRD{cond} Rt, Rt2, [Rn, ±Rm]! ; pre-indexed, doubleword ; A32 only
LDRD{cond} Rt, Rt2, [Rn], ±Rm ; post-indexed, doubleword ; A32 only
```

where:

type

can be any one of:

B	unsigned Byte (Zero extend to 32 bits on loads.)
SB	signed Byte (LDR only. Sign extend to 32 bits.)
H	unsigned Halfword (Zero extend to 32 bits on loads.
SH	signed Halfword (LDR only. Sign extend to 32 bits.)
-	omitted, for Word.

cond

is an optional condition code.

Rt

is the register to load.

Rn

is the register on which the memory address is based.

Rm

is a register containing a value to be used as the offset. –*Rm* is not permitted in T32 code.

shift

is an optional shift.

Rt2

is the additional register to load for doubleword operations.

Not all options are available in every instruction set and architecture.

Offset register and shift options

The following table shows the ranges of offsets and availability of these instructions:

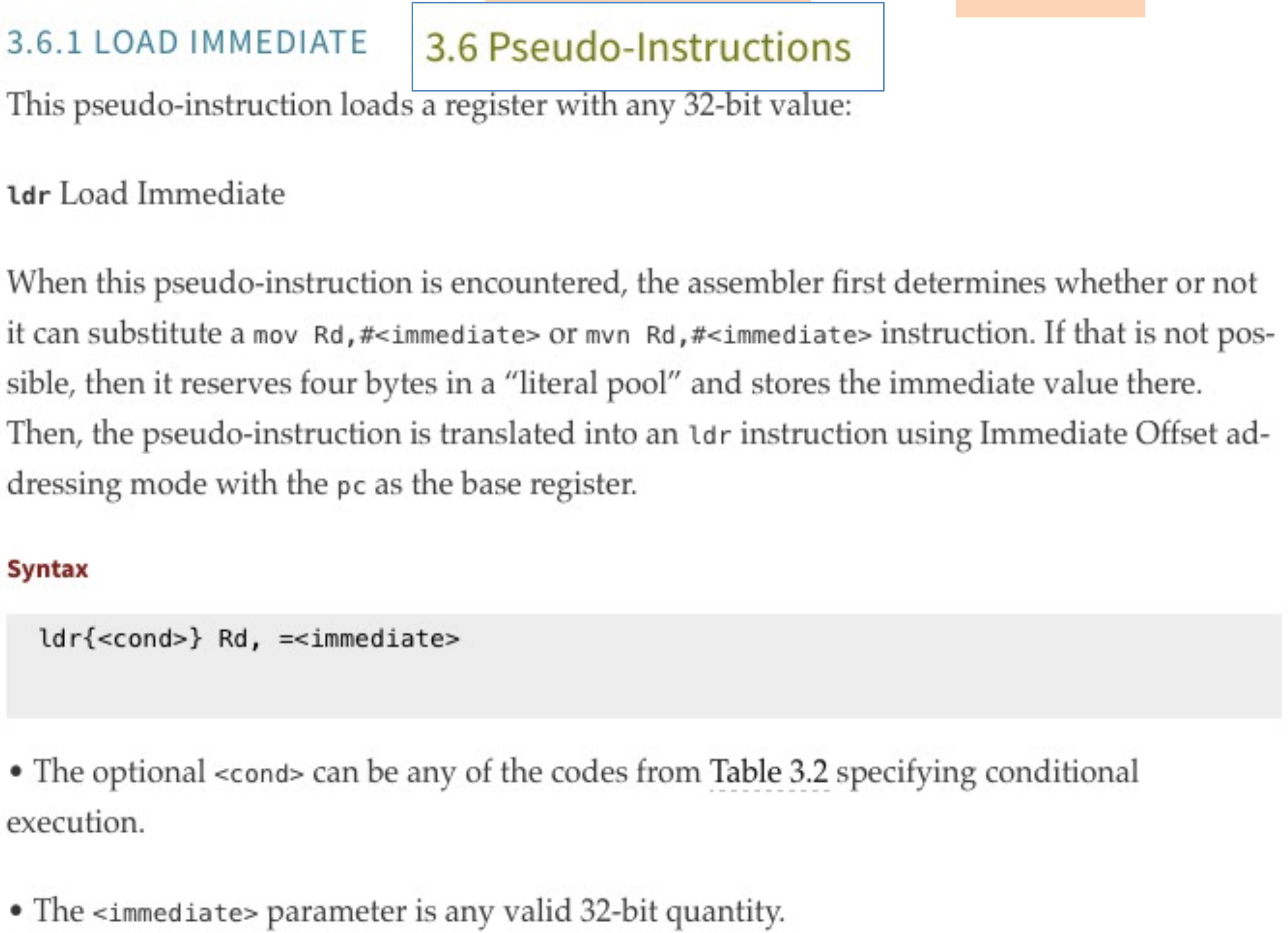
COMP122

ARM Assembly

© Jeff Drobman
2016-2022

Load Immediate (li)

ARM Book



3.6 Pseudo-Instructions

3.6.1 LOAD IMMEDIATE

This pseudo-instruction loads a register with any 32-bit value:

`ldr` Load Immediate

When this pseudo-instruction is encountered, the assembler first determines whether or not it can substitute a `mov Rd,#<immediate>` or `mvn Rd,#<immediate>` instruction. If that is not possible, then it reserves four bytes in a "literal pool" and stores the immediate value there. Then, the pseudo-instruction is translated into an `ldr` instruction using Immediate Offset addressing mode with the `pc` as the base register.

Syntax

```
ldr{<cond>} Rd, =<immediate>
```

- The optional `<cond>` can be any of the codes from Table 3.2 specifying conditional execution.
- The `<immediate>` parameter is any valid 32-bit quantity.

CSUN
CALIFORNIA
STATE UNIVERSITY
NORTHRIDGE

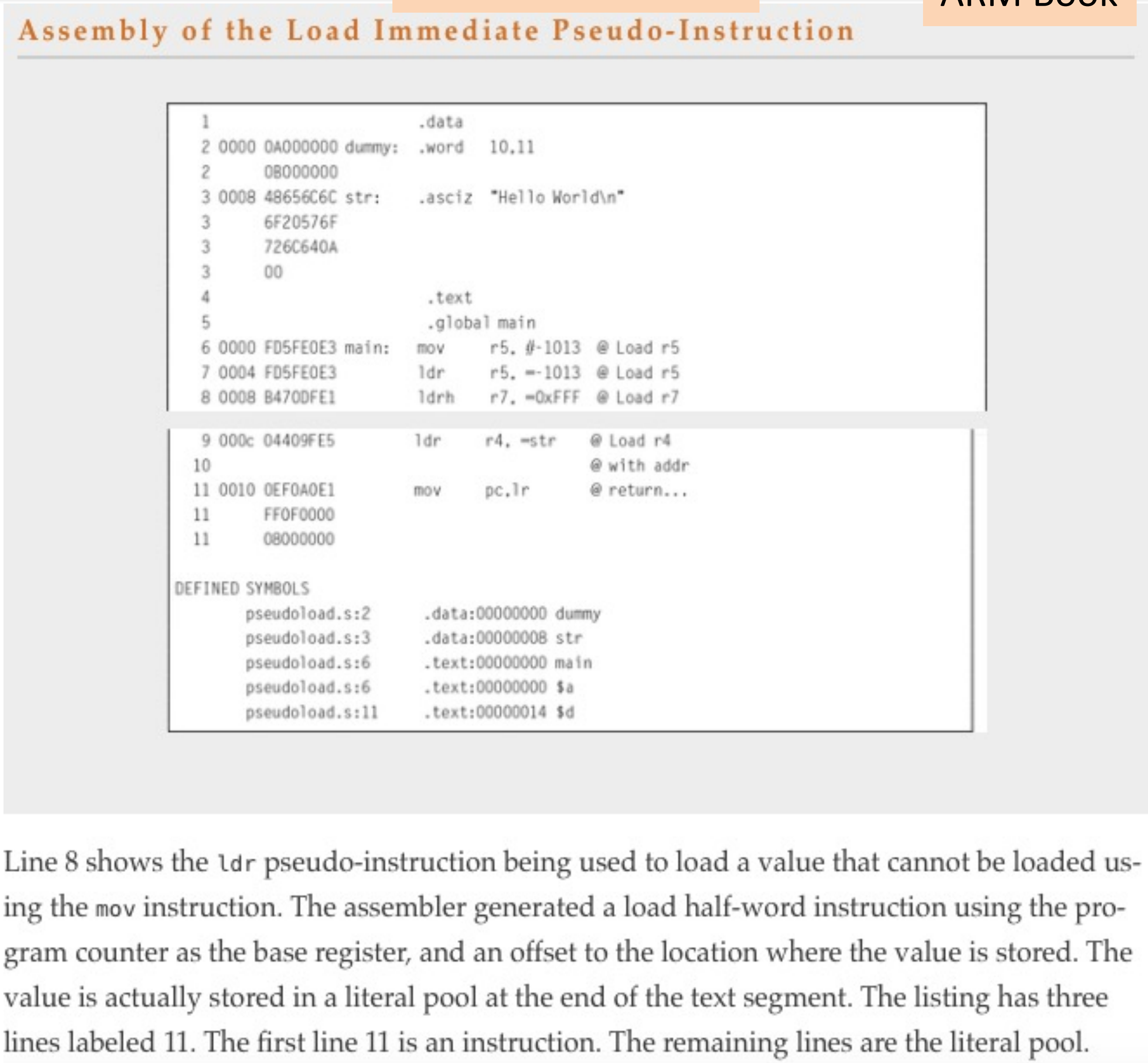
COMP122

ARM Assembly

DR JEFF SOFTWARE
INDIE APP DEVELOPER
© Jeff Drobman
2016-2022

Load Immediate (li)

ARM Book



Assembly of the Load Immediate Pseudo-Instruction

```
 1                     .data
 2 0000 0A000000 dummy: .word   10,11
 2      0B000000
 3 0008 48656C6C str:   .asciz  "Hello World\n"
 3      6F20576F
 3      726C640A
 3      00
 4                      .text
 5                      .global main
 6 0000 FD5FE0E3 main:  mov     r5, #-1013  @ Load r5
 7 0004 FD5FE0E3        ldr     r5, =-1013  @ Load r5
 8 0008 B470DFE1        ldrh    r7, =0xFFF  @ Load r7
 9 000c 04409FE5        ldr     r4, =str    @ Load r4
10                                          @ with addr
11 0010 0EF0A0E1        mov     pc,lr       @ return...
11      FF0F0000
11      08000000

DEFINED SYMBOLS
        pseudoload.s:2      .data:00000000 dummy
        pseudoload.s:3      .data:00000008 str
        pseudoload.s:6      .text:00000000 main
        pseudoload.s:6      .text:00000000 $a
        pseudoload.s:11     .text:00000014 $d
```

Line 8 shows the `ldr` pseudo-instruction being used to load a value that cannot be loaded using the `mov` instruction. The assembler generated a load half-word instruction using the program counter as the base register, and an offset to the location where the value is stored. The value is actually stored in a literal pool at the end of the text segment. The listing has three lines labeled 11. The first line 11 is an instruction. The remaining lines are the literal pool.

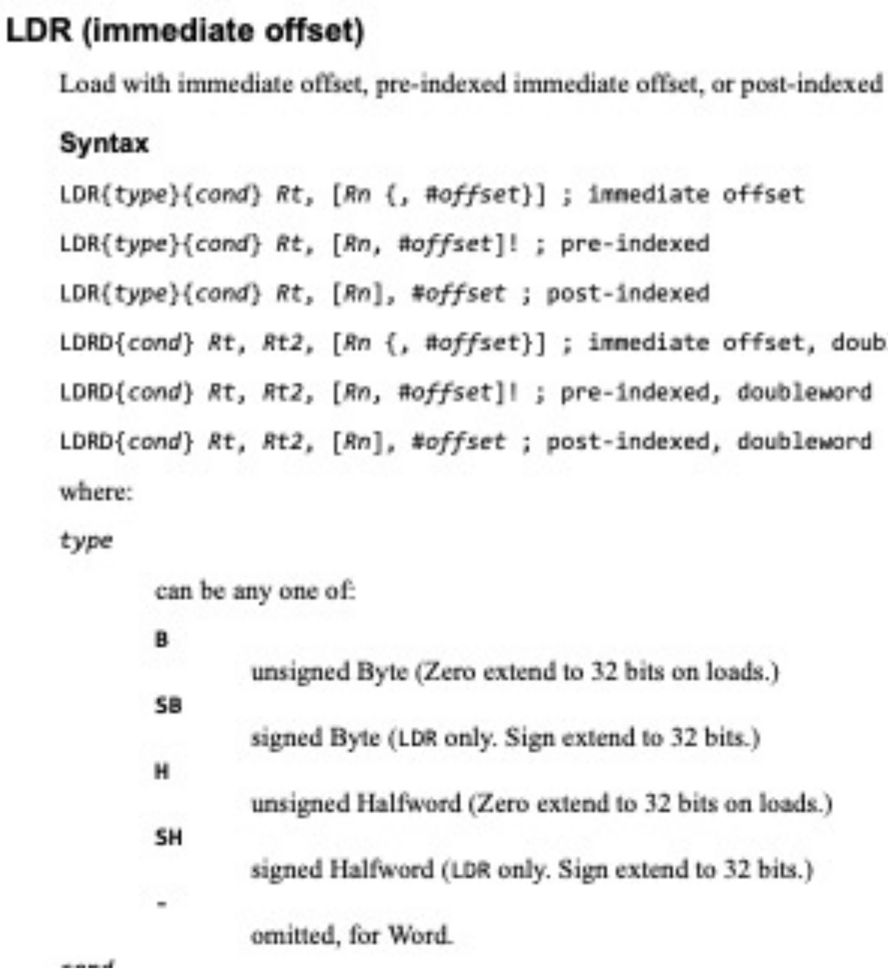
CSUN
CALIFORNIA
STATE UNIVERSITY
NORTHRIDGE
COMP122

ARM Assembly

DR JEFF SOFTWARE
INDIE APP DEVELOPER
© Jeff Drobman
2016-2022

LDR

ARM Ref



LDR (immediate offset)

Load with immediate offset, pre-indexed immediate offset, or post-indexed immediate offset.

Syntax

```
LDR{type}{cond} Rt, [Rn {, #offset}] ; immediate offset
LDR{type}{cond} Rt, [Rn, #offset]! ; pre-indexed
LDR{type}{cond} Rt, [Rn], #offset ; post-indexed
LDRD{cond} Rt, Rt2, [Rn {, #offset}] ; immediate offset, doubleword
LDRD{cond} Rt, Rt2, [Rn, #offset]! ; pre-indexed, doubleword
LDRD{cond} Rt, Rt2, [Rn], #offset ; post-indexed, doubleword
```

where:

type

can be any one of:

B

unsigned Byte (Zero extend to 32 bits on loads.)

SB

signed Byte (LDR only. Sign extend to 32 bits.)

H

unsigned Halfword (Zero extend to 32 bits on loads.)

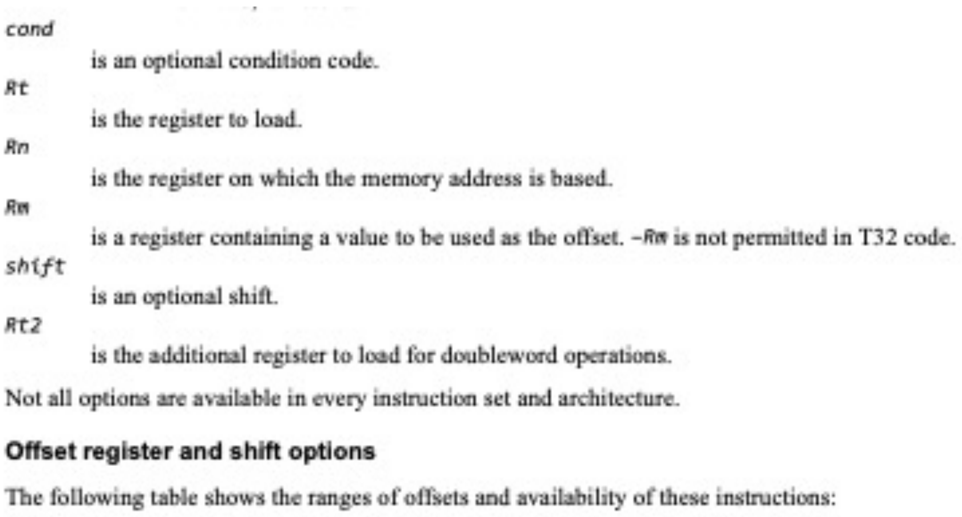
SH

signed Halfword (LDR only. Sign extend to 32 bits.)

-

omitted, for Word.

cond



cond

is an optional condition code.

Rt

is the register to load.

Rn

is the register on which the memory address is based.

Rm

is a register containing a value to be used as the offset. –*Rm* is not permitted in T32 code.

shift

is an optional shift.

Rt2

is the additional register to load for doubleword operations.

Not all options are available in every instruction set and architecture.

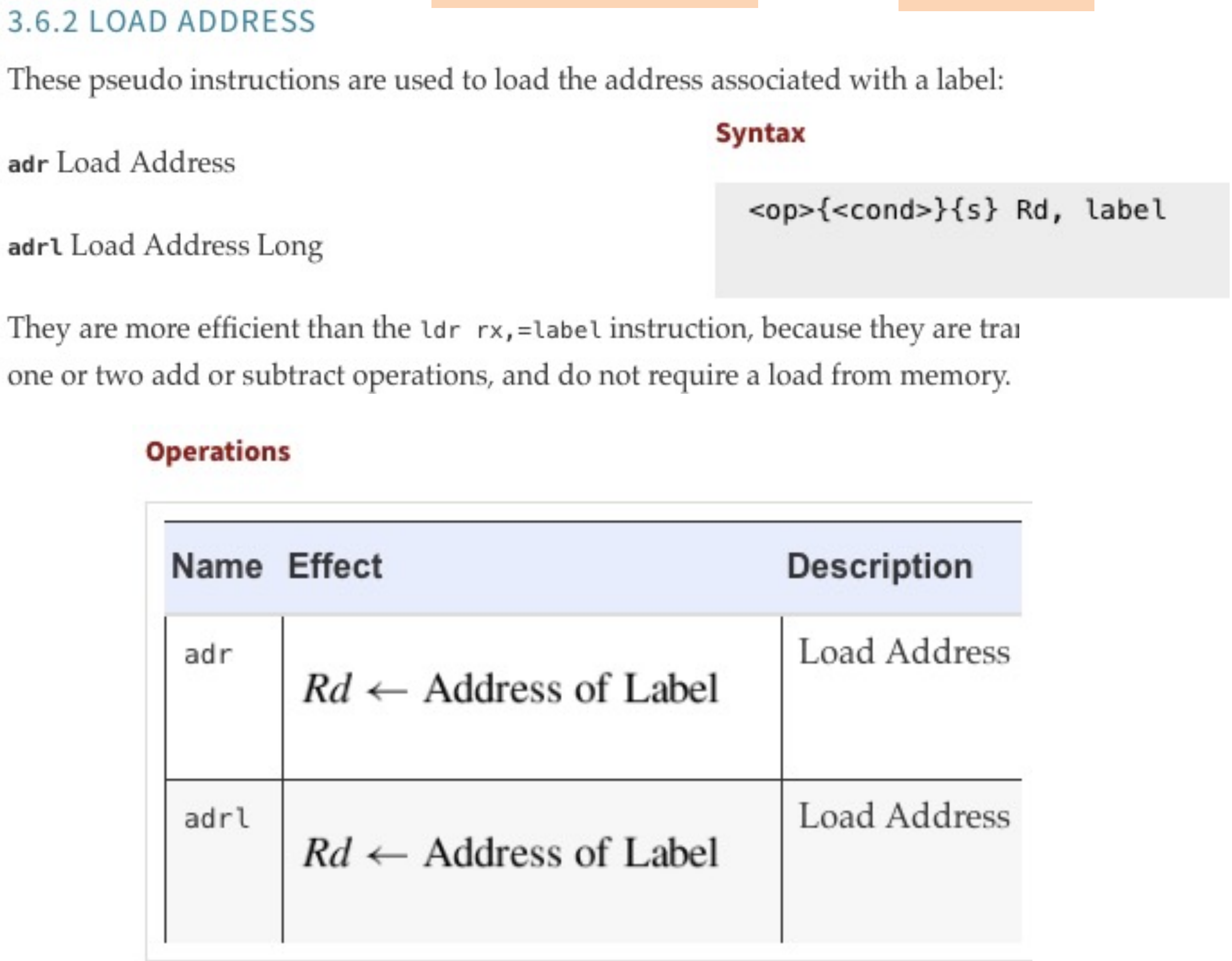
Offset register and shift options

The following table shows the ranges of offsets and availability of these instructions:

ARM Assembly

Load Address (adr)

ARM Book



3.6.2 LOAD ADDRESS

These pseudo instructions are used to load the address associated with a label:

adr Load Address

adrl Load Address Long

Syntax

```
<op>{<cond>}{s} Rd, label
```

They are more efficient than the `ldr rx,=label` instruction, because they are trai one or two add or subtract operations, and do not require a load from memory.

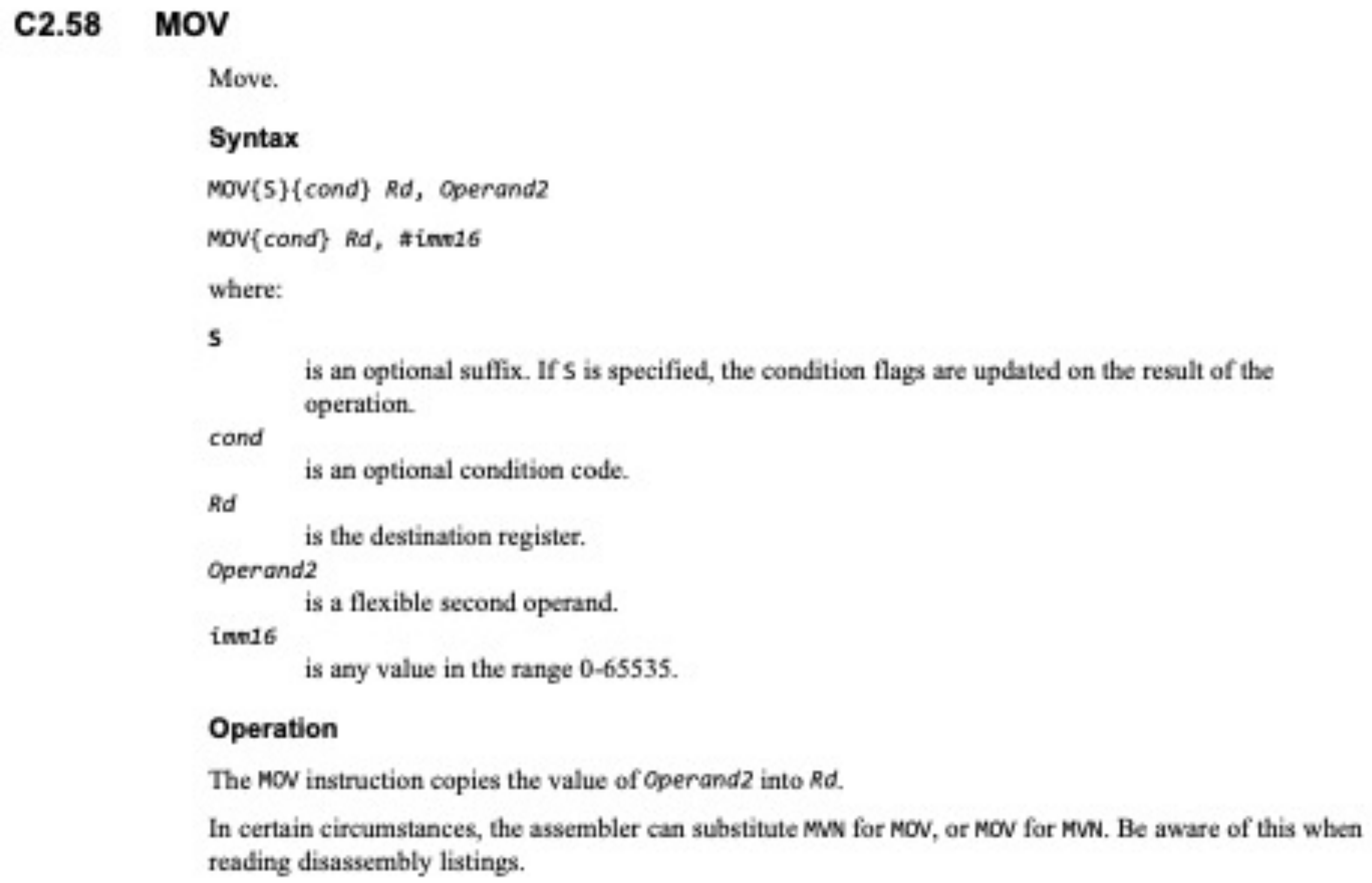
Operations

Name	Effect	Description
`adr`	$Rd \leftarrow$ Address of Label	Load Address
`adrl`	$Rd \leftarrow$ Address of Label	Load Address

ARM Assembly

MOV

ARM Ref



C2.58 MOV

Move.

Syntax

```
MOV{S}{cond} Rd, Operand2
MOV{cond} Rd, #imm16
```

where:

`S`
is an optional suffix. If `S` is specified, the condition flags are updated on the result of the operation.

cond
is an optional condition code.

Rd
is the destination register.

Operand2
is a flexible second operand.

imm16
is any value in the range 0-65535.

Operation

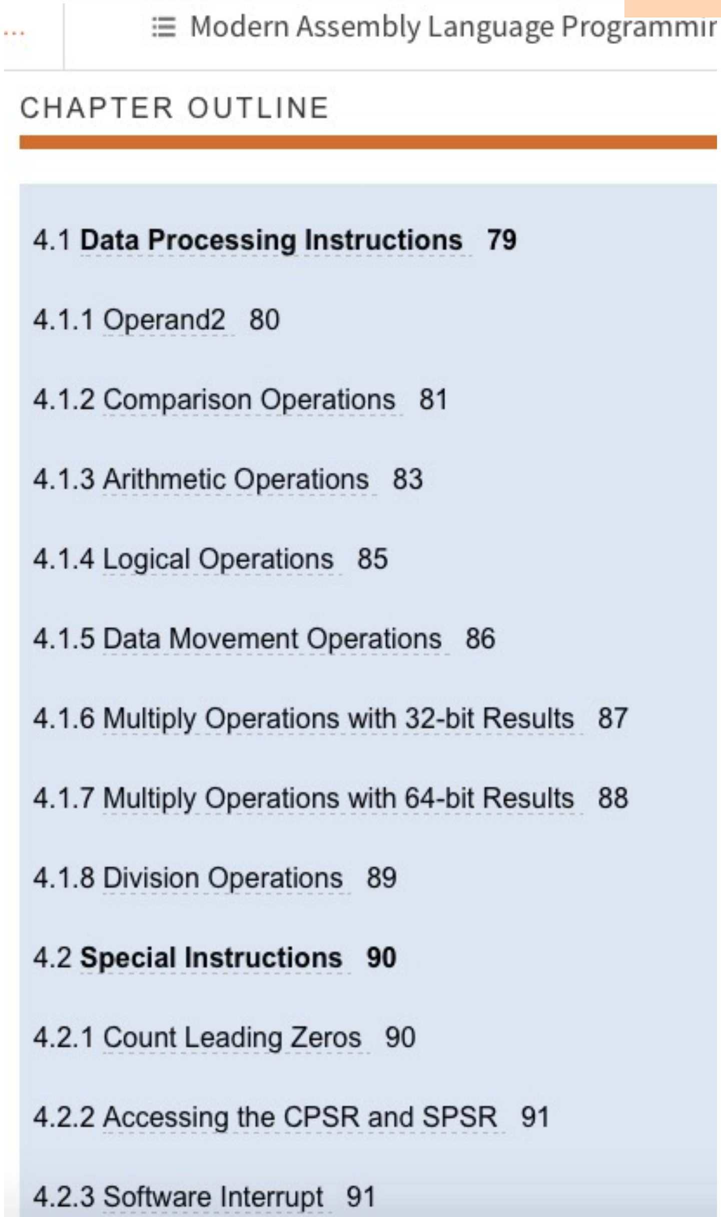
The `MOV` instruction copies the value of *Operand2* into *Rd*.

In certain circumstances, the assembler can substitute `MVN` for `MOV`, or `MOV` for `MVN`. Be aware of this when reading disassembly listings.

ARM Assembly

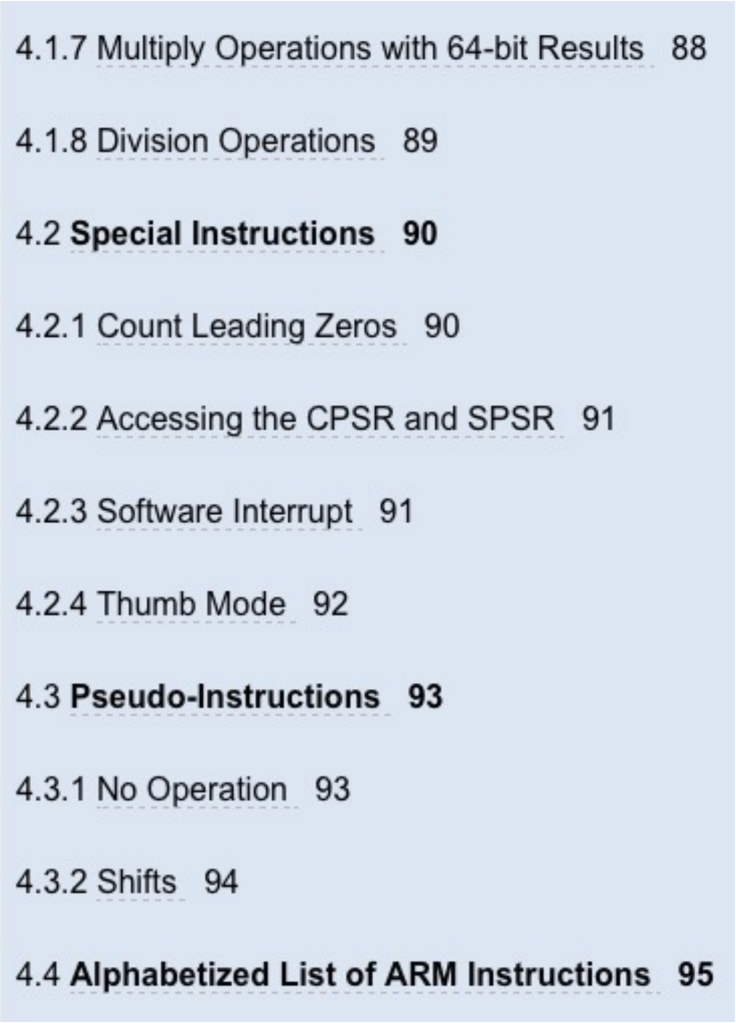
Ch 4: ALU Ops

ARM Book



Modern Assembly Language Programmir

CHAPTER OUTLINE

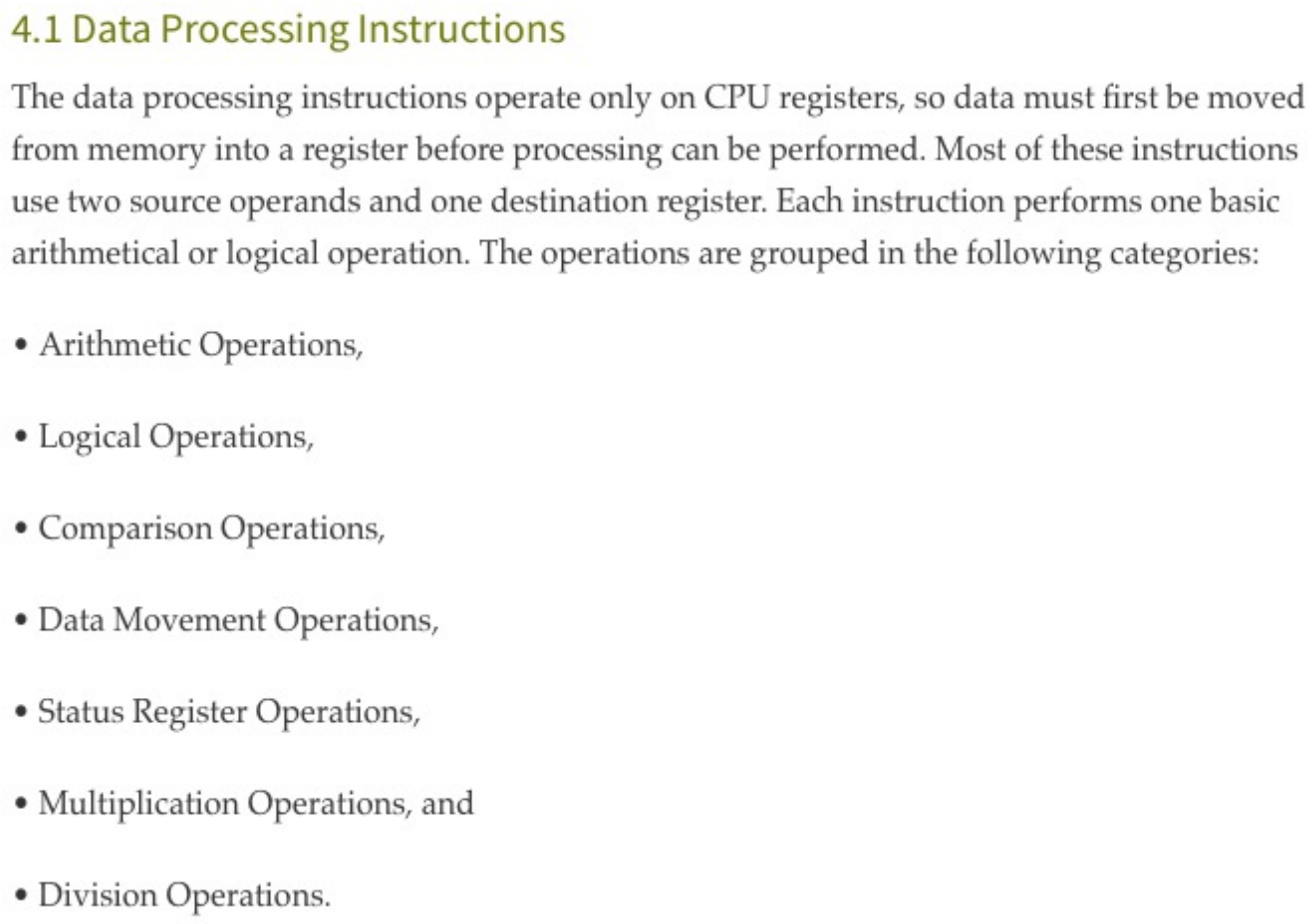


4.1.7 Multiply Operations with 64-bit Results 88
4.1.8 Division Operations 89
4.2 **Special Instructions** 90
4.2.1 Count Leading Zeros 90
4.2.2 Accessing the CPSR and SPSR 91
4.2.3 Software Interrupt 91

ARM Assembly

Ch 4: ALU Ops

ARM Book



4.1 Data Processing Instructions

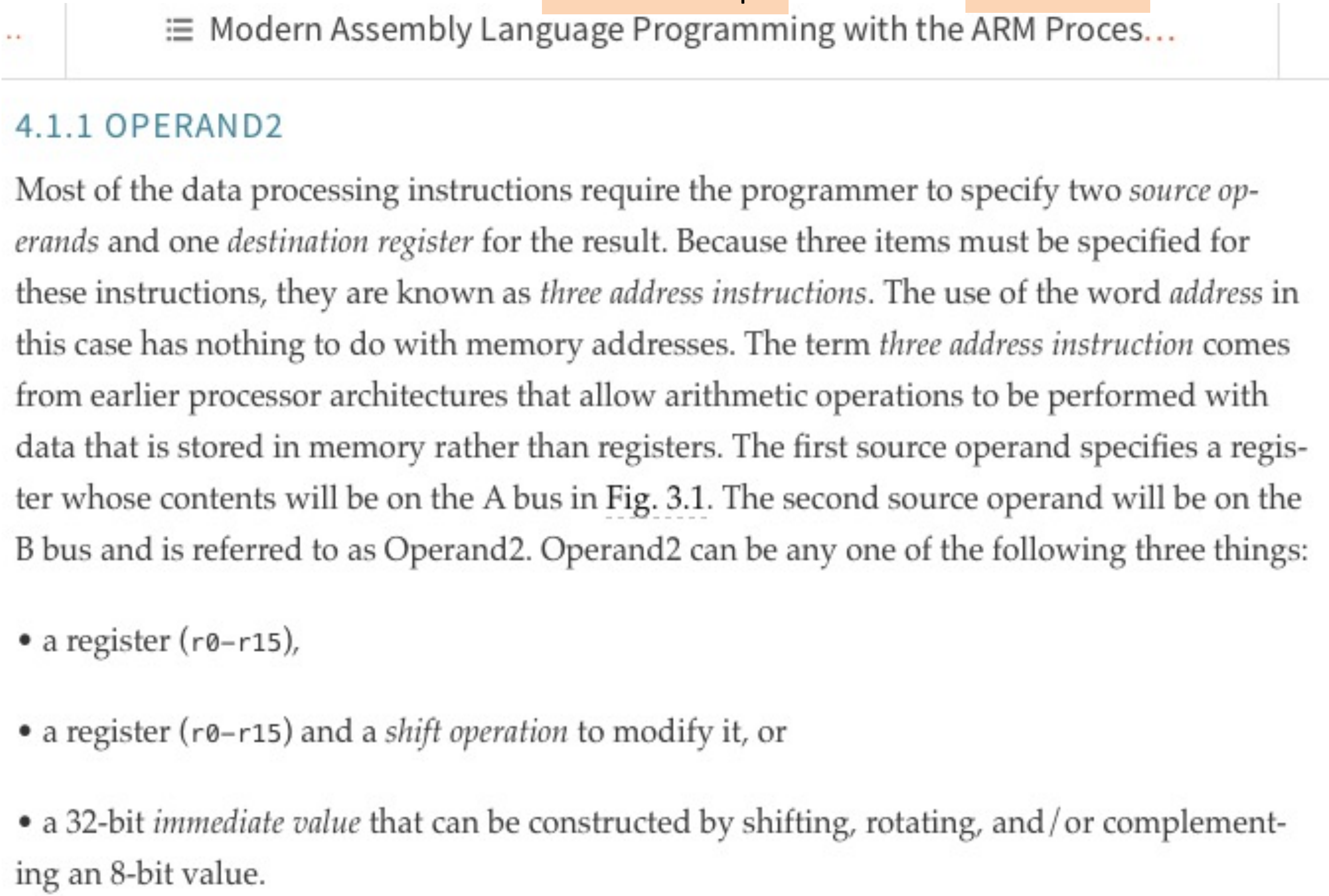
The data processing instructions operate only on CPU registers, so data must first be moved from memory into a register before processing can be performed. Most of these instructions use two source operands and one destination register. Each instruction performs one basic arithmetical or logical operation. The operations are grouped in the following categories:

- Arithmetic Operations,
- Logical Operations,
- Comparison Operations,
- Data Movement Operations,
- Status Register Operations,
- Multiplication Operations, and
- Division Operations.

ARM Assembly

Ch 4: ALU Ops

ARM Book



☰ Modern Assembly Language Programming with the ARM Proces...

4.1.1 OPERAND2

Most of the data processing instructions require the programmer to specify two *source operands* and one *destination register* for the result. Because three items must be specified for these instructions, they are known as *three address instructions*. The use of the word *address* in this case has nothing to do with memory addresses. The term *three address instruction* comes from earlier processor architectures that allow arithmetic operations to be performed with data that is stored in memory rather than registers. The first source operand specifies a register whose contents will be on the A bus in Fig. 3.1. The second source operand will be on the B bus and is referred to as Operand2. Operand2 can be any one of the following three things:

- a register (`r0–r15`),
- a register (`r0–r15`) and a *shift operation* to modify it, or
- a 32-bit *immediate value* that can be constructed by shifting, rotating, and/or complementing an 8-bit value.

ARM Assembly

Ch 4: ALU Ops

ARM Book

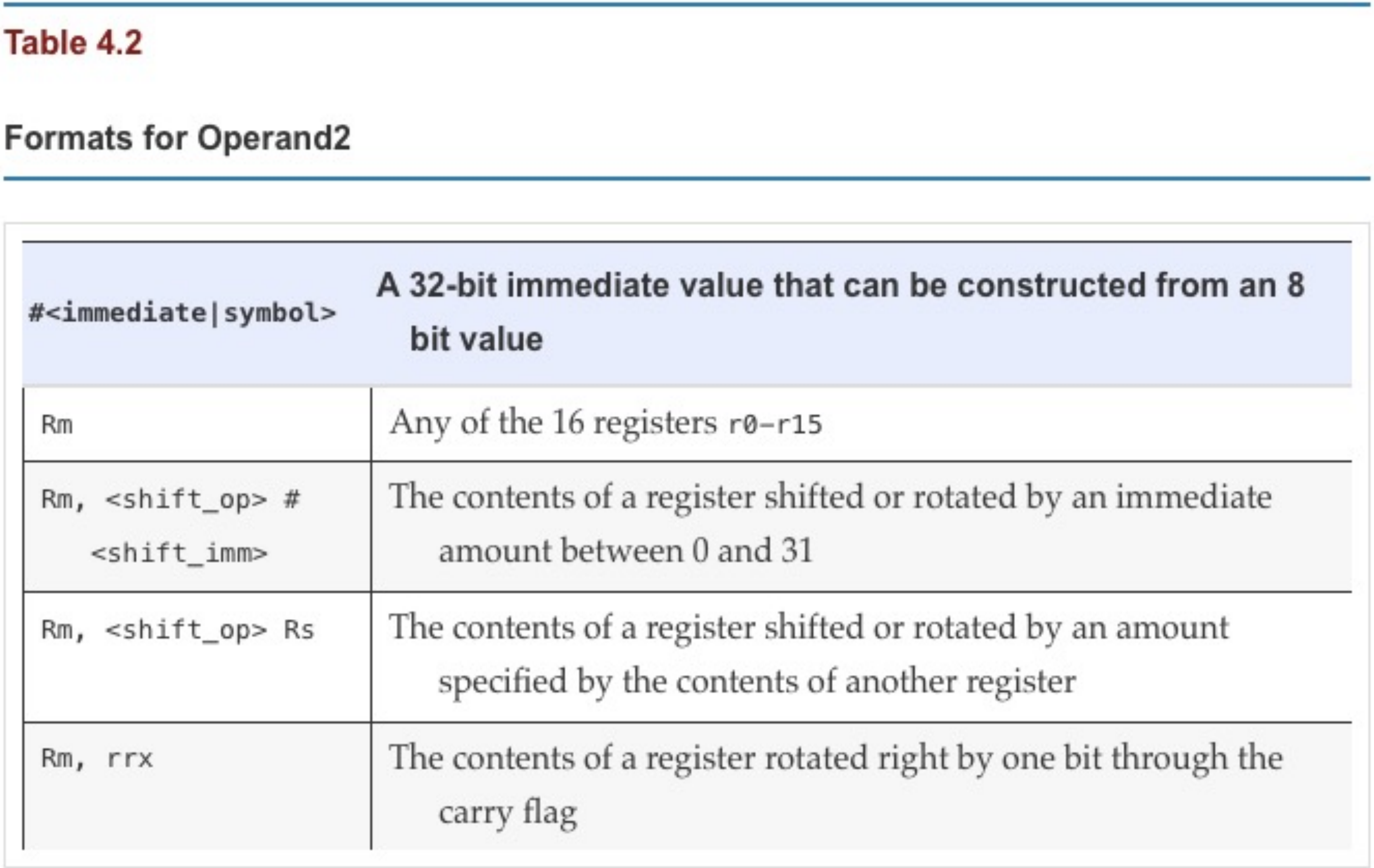


Table 4.2

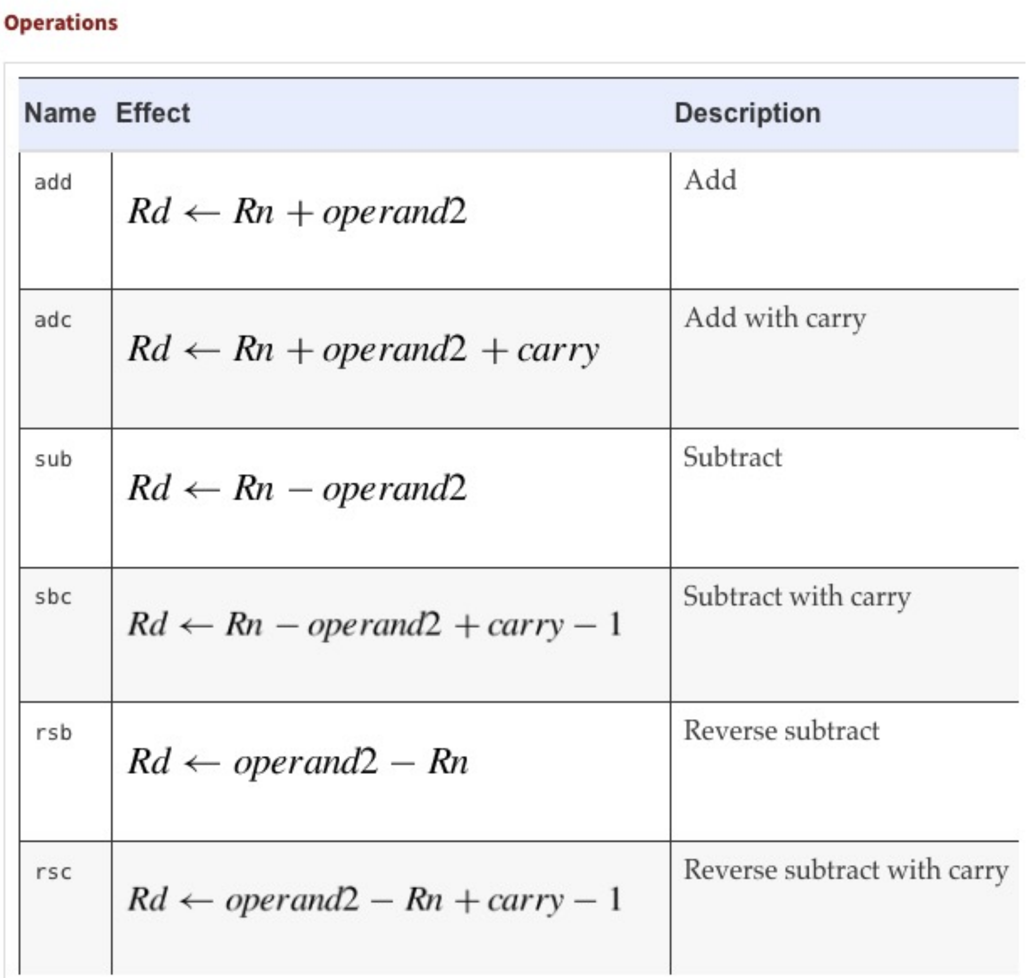
Formats for Operand2

#<immediate|symbol>	A 32-bit immediate value that can be constructed from an 8 bit value
Rm	Any of the 16 registers r0–r15
Rm, <shift_op> #<shift_imm>	The contents of a register shifted or rotated by an immediate amount between 0 and 31
Rm, <shift_op> Rs	The contents of a register shifted or rotated by an amount specified by the contents of another register
Rm, rrx	The contents of a register rotated right by one bit through the carry flag

ARM Assembly

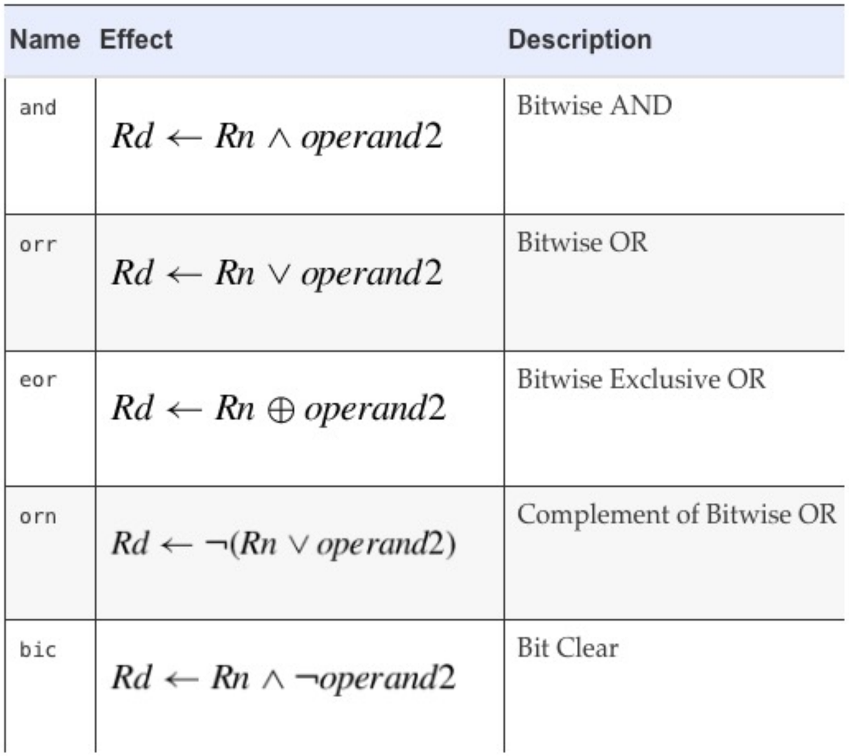
Ch 4: ALU Ops

ARM Book



Operations

Name	Effect	Description
add	$Rd \leftarrow Rn + operand2$	Add
adc	$Rd \leftarrow Rn + operand2 + carry$	Add with carry
sub	$Rd \leftarrow Rn - operand2$	Subtract
sbc	$Rd \leftarrow Rn - operand2 + carry - 1$	Subtract with carry
rsb	$Rd \leftarrow operand2 - Rn$	Reverse subtract
rsc	$Rd \leftarrow operand2 - Rn + carry - 1$	Reverse subtract with carry

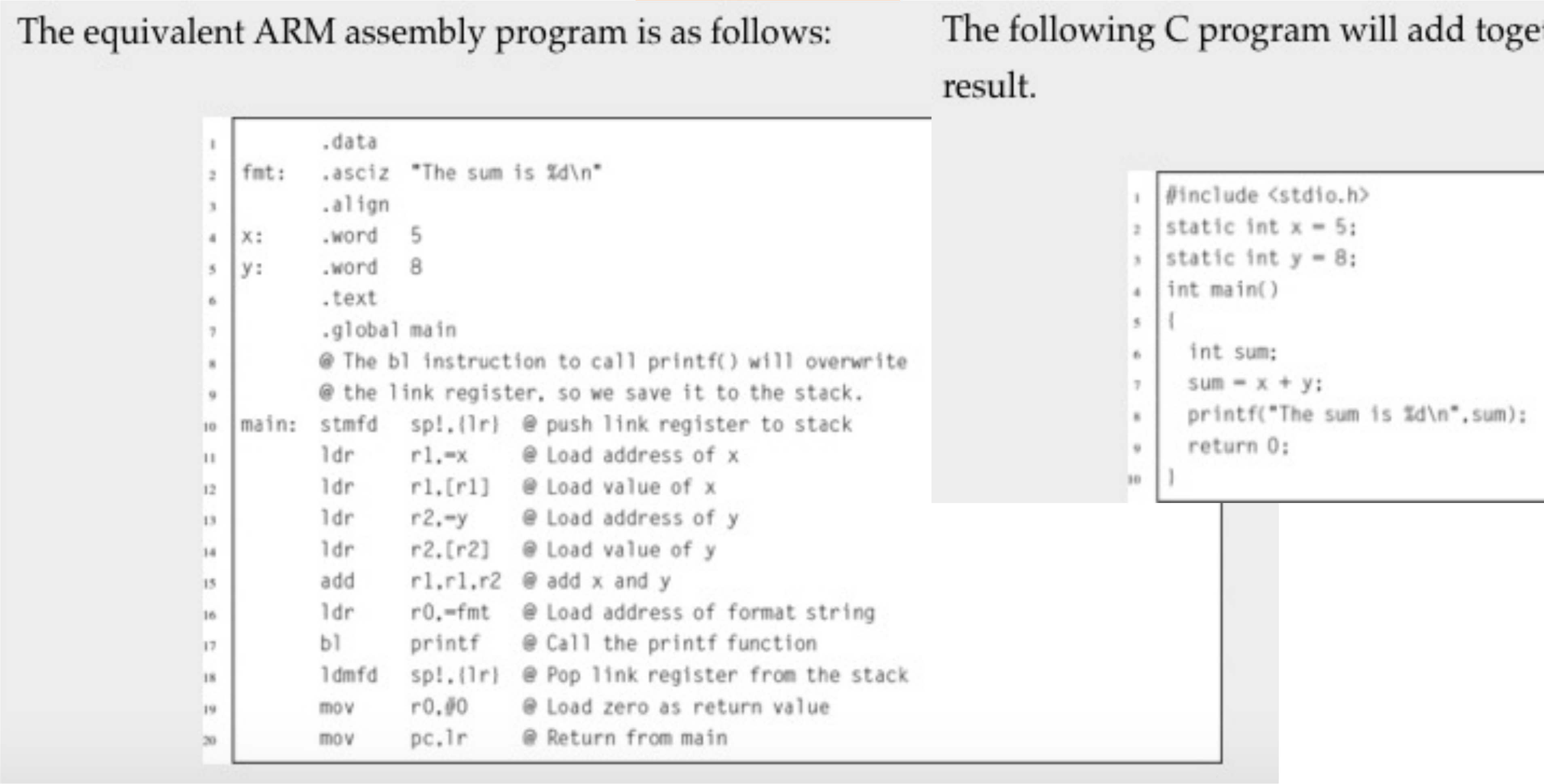


Name	Effect	Description
and	$Rd \leftarrow Rn \wedge operand2$	Bitwise AND
orr	$Rd \leftarrow Rn \vee operand2$	Bitwise OR
eor	$Rd \leftarrow Rn \oplus operand2$	Bitwise Exclusive OR
orn	$Rd \leftarrow \neg(Rn \vee operand2)$	Complement of Bitwise OR
bic	$Rd \leftarrow Rn \wedge \neg operand2$	Bit Clear

ARM Assembly

Ch 4: ALU Ops

ARM Book



The equivalent ARM assembly program is as follows:

```
        .data
fmt:    .asciz  "The sum is %d\n"
        .align
x:      .word   5
y:      .word   8
        .text
        .global main
        @ The bl instruction to call printf() will overwrite
        @ the link register, so we save it to the stack.
main:   stmfd   sp!,{lr}   @ push link register to stack
        ldr     r1,=x      @ Load address of x
        ldr     r1,[r1]    @ Load value of x
        ldr     r2,=y      @ Load address of y
        ldr     r2,[r2]    @ Load value of y
        add     r1,r1,r2   @ add x and y
        ldr     r0,=fmt    @ Load address of format string
        bl      printf     @ Call the printf function
        ldmfd   sp!,{lr}   @ Pop link register from the stack
        mov     r0,#0      @ Load zero as return value
        mov     pc,lr      @ Return from main
```

The following C program will add toge

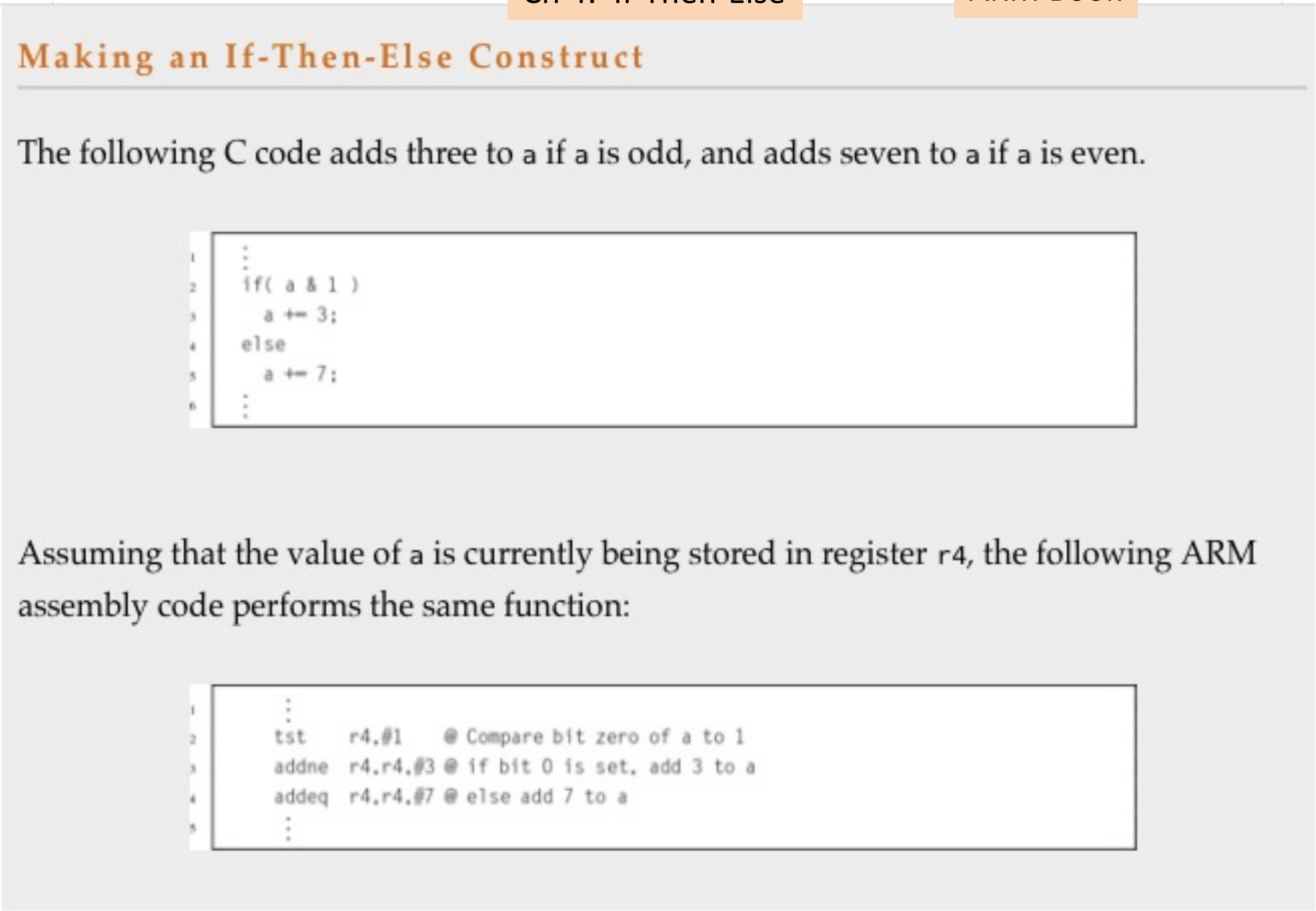
result.

```
#include <stdio.h>
static int x = 5;
static int y = 8;
int main()
{
  int sum;
  sum = x + y;
  printf("The sum is %d\n",sum);
  return 0;
}
```

ARM Assembly

Ch 4: If-Then-Else

ARM Book



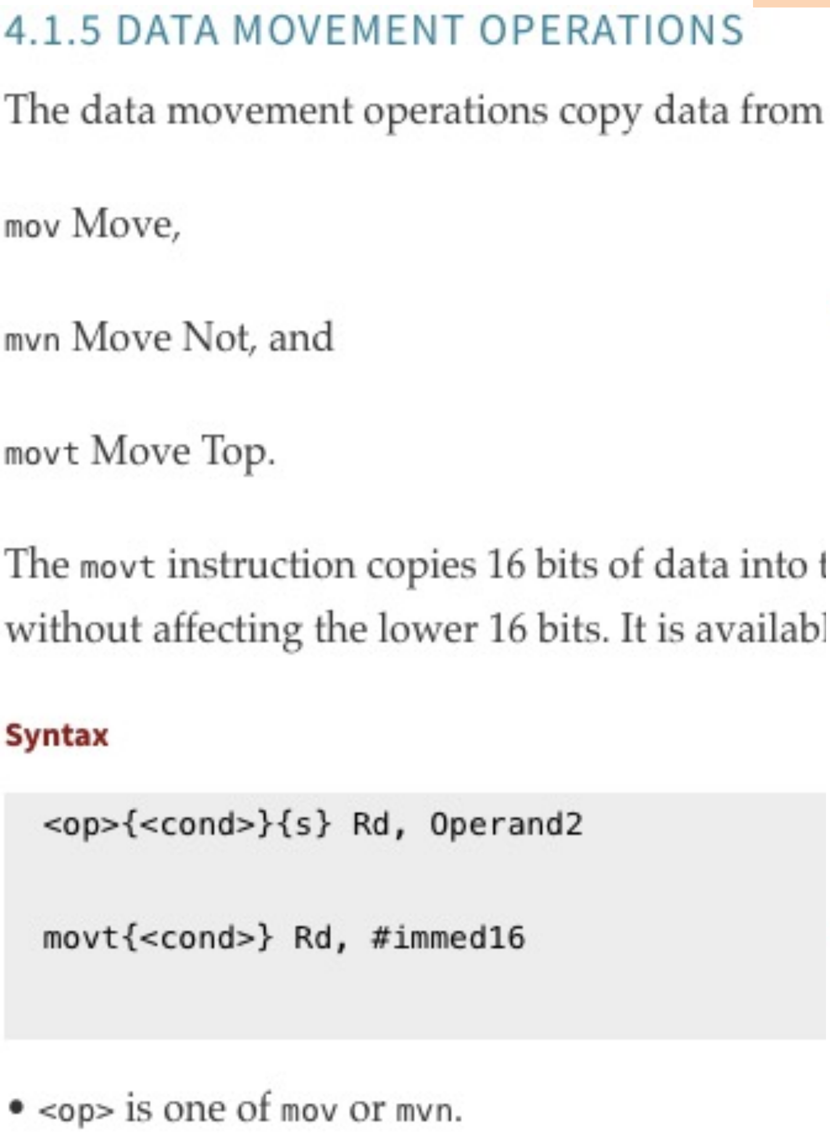
Making an If-Then-Else Construct

The following C code adds three to a if a is odd, and adds seven to a if a is even.

```
⋮
if( a & 1 )
  a += 3;
else
  a += 7;
⋮
```

Assuming that the value of a is currently being stored in register r4, the following ARM assembly code performs the same function:

```
   ⋮
tst     r4,#1     @ Compare bit zero of a to 1
addne   r4,r4,#3 @ if bit 0 is set, add 3 to a
addeq   r4,r4,#7 @ else add 7 to a
   ⋮
```



4.1.5 DATA MOVEMENT OPERATIONS

The data movement operations copy data from one register to another:

mov Move,

mvn Move Not, and

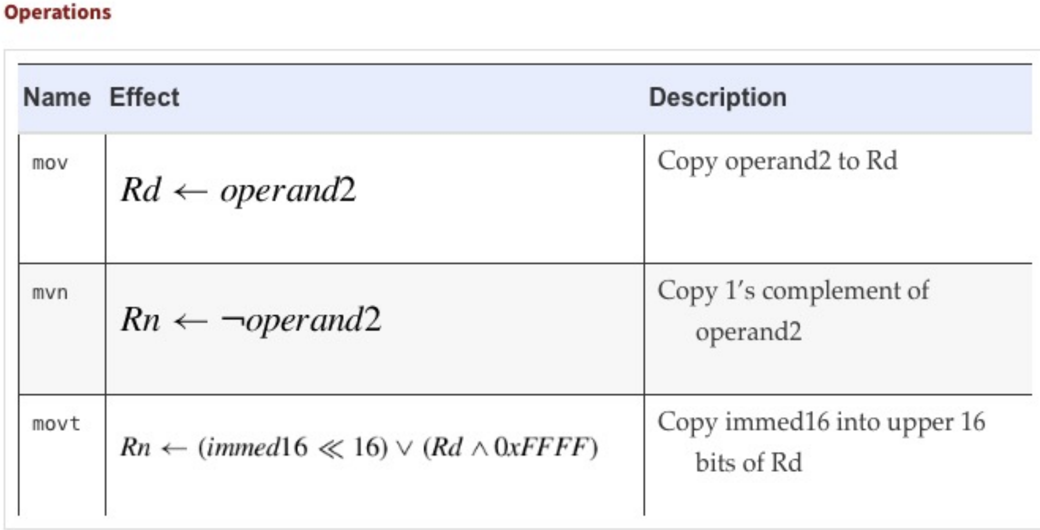
movt Move Top.

The movt instruction copies 16 bits of data into t without affecting the lower 16 bits. It is availabl

Syntax

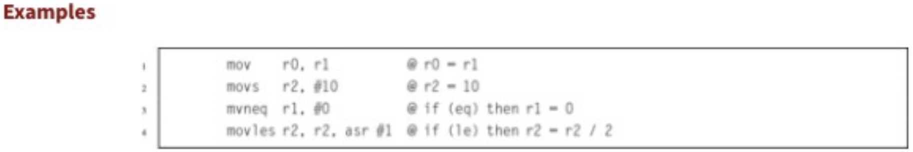
```
<op>{<cond>}{s} Rd, Operand2

movt{<cond>} Rd, #immed16
```

- <op> is one of mov or mvn.



Operations

Name	Effect	Description
mov	$Rd \leftarrow operand2$	Copy operand2 to Rd
mvn	$Rn \leftarrow \neg operand2$	Copy 1's complement of operand2
movt	$Rn \leftarrow (immed16 \ll 16) \vee (Rd \wedge 0xFFFF)$	Copy immed16 into upper 16 bits of Rd



Examples

```
        mov    r0, r1           @ r0 = r1
        movs   r2, #10          @ r2 = 10
        mvneq  r1, #0           @ if (eq) then r1 = 0
        movles r2, r2, asr #1   @ if (le) then r2 = r2 / 2
```

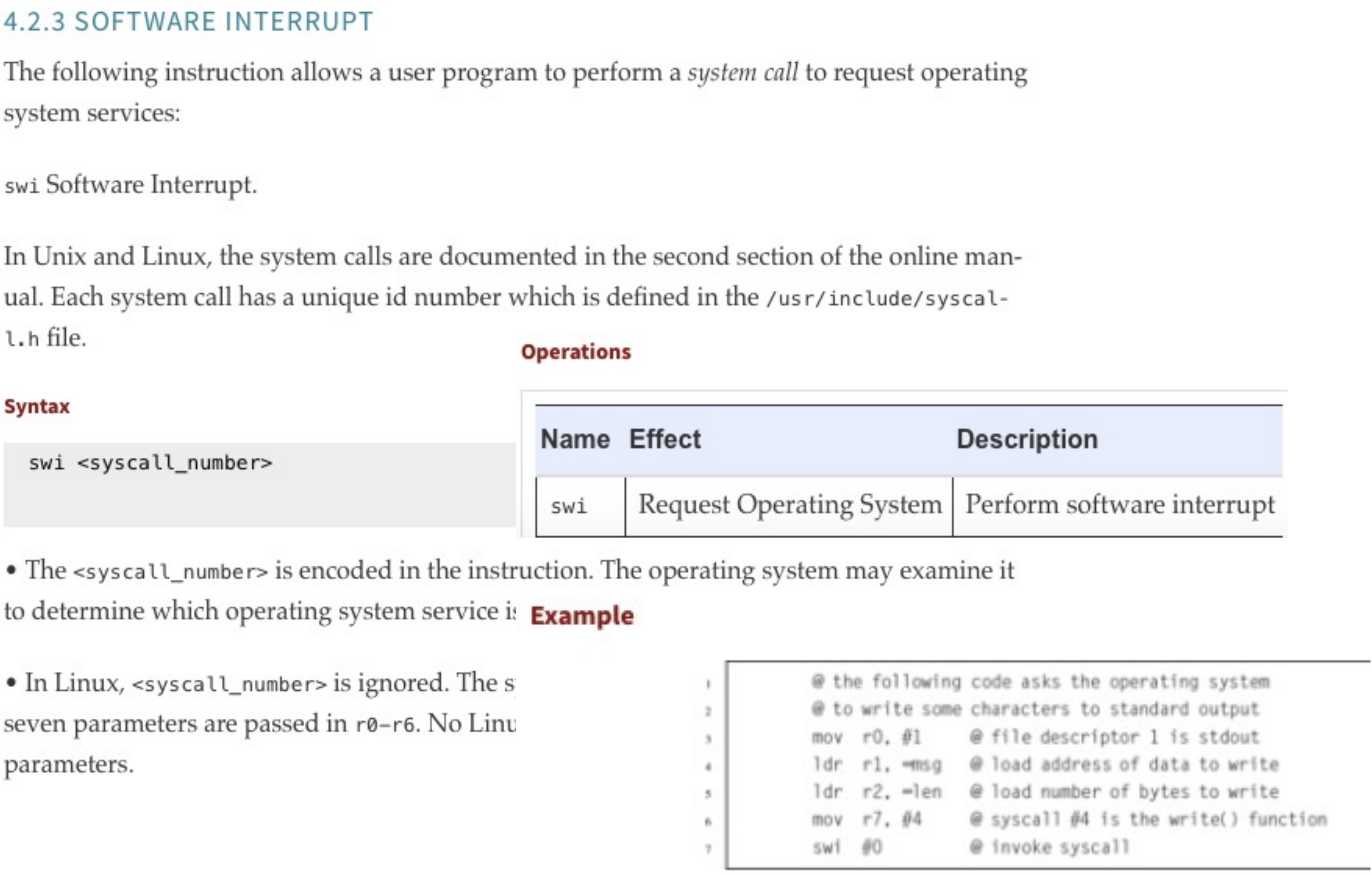
CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP122

ARM Assembly

Ch 4: ALU Ops

ARM Book

DR JEFF SOFTWARE
INDIE APP DEVELOPER



4.2.3 SOFTWARE INTERRUPT

The following instruction allows a user program to perform a *system call* to request operating system services:

`swi` Software Interrupt.

In Unix and Linux, the system calls are documented in the second section of the online manual. Each system call has a unique id number which is defined in the `/usr/include/syscall.h` file.

Syntax

```
swi <syscall_number>
```

Operations

Name	Effect	Description
`swi`	Request Operating System	Perform software interrupt

- The `<syscall_number>` is encoded in the instruction. The operating system may examine it to determine which operating system service i

- In Linux, `<syscall_number>` is ignored. The s seven parameters are passed in `r0`–`r6`. No Linu parameters.

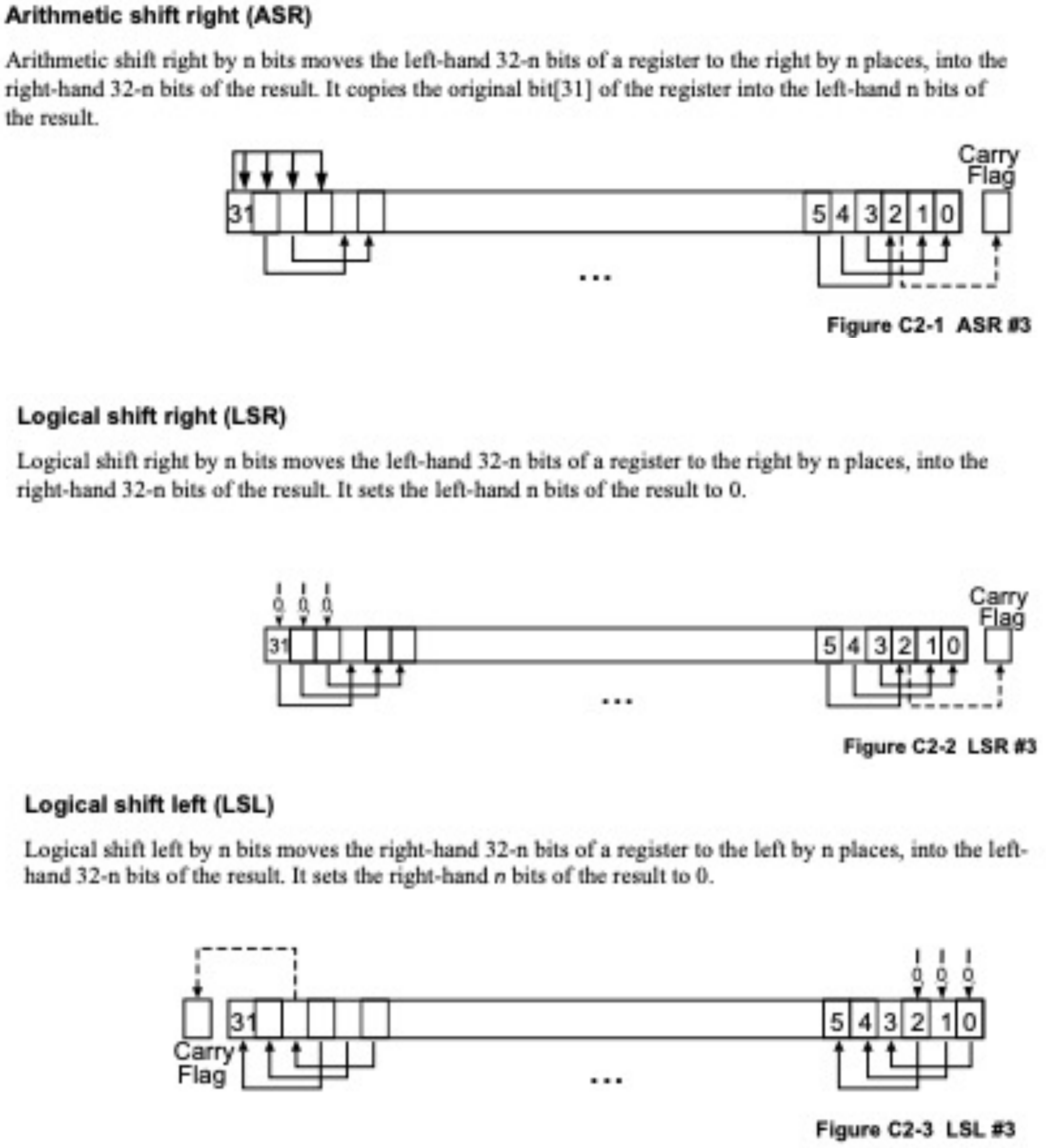
Example

```
@ the following code asks the operating system
@ to write some characters to standard output
mov  r0, #1       @ file descriptor 1 is stdout
ldr  r1, =msg     @ load address of data to write
ldr  r2, =len     @ load number of bytes to write
mov  r7, #4       @ syscall #4 is the write() function
swi  #0           @ invoke syscall
```

ARM Assembly

Shift

ARM Ref



Arithmetic shift right (ASR)

Arithmetic shift right by n bits moves the left-hand 32-n bits of a register to the right by n places, into the right-hand 32-n bits of the result. It copies the original bit[31] of the register into the left-hand n bits of the result.

Figure C2-1 ASR #3

Logical shift right (LSR)

Logical shift right by n bits moves the left-hand 32-n bits of a register to the right by n places, into the right-hand 32-n bits of the result. It sets the left-hand n bits of the result to 0.

Figure C2-2 LSR #3

Logical shift left (LSL)

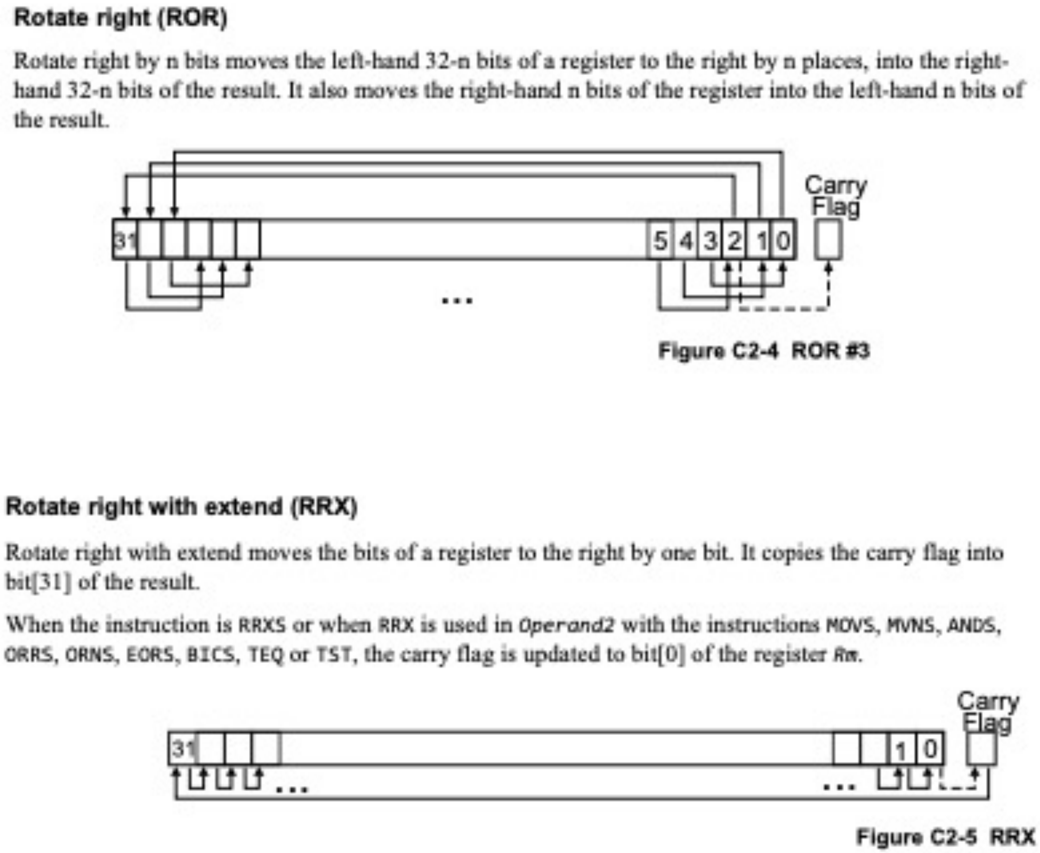
Logical shift left by n bits moves the right-hand 32-n bits of a register to the left by n places, into the left-hand 32-n bits of the result. It sets the right-hand *n* bits of the result to 0.

Figure C2-3 LSL #3

ARM Assembly

Rotate

ARM Ref



Rotate right (ROR)

Rotate right by n bits moves the left-hand 32-n bits of a register to the right by n places, into the right-hand 32-n bits of the result. It also moves the right-hand n bits of the register into the left-hand n bits of the result.

Carry Flag
31
5
4
3
2
1
0
...

Figure C2-4 ROR #3

Rotate right with extend (RRX)

Rotate right with extend moves the bits of a register to the right by one bit. It copies the carry flag into bit[31] of the result.

When the instruction is RRXS or when RRX is used in *Operand2* with the instructions MOVS, MVNS, ANDS, ORRS, ORNS, EORS, BICS, TEQ or TST, the carry flag is updated to bit[0] of the register *Rm*.

Figure C2-5 RRX

ARM Assembly

{cond}

ARM Book

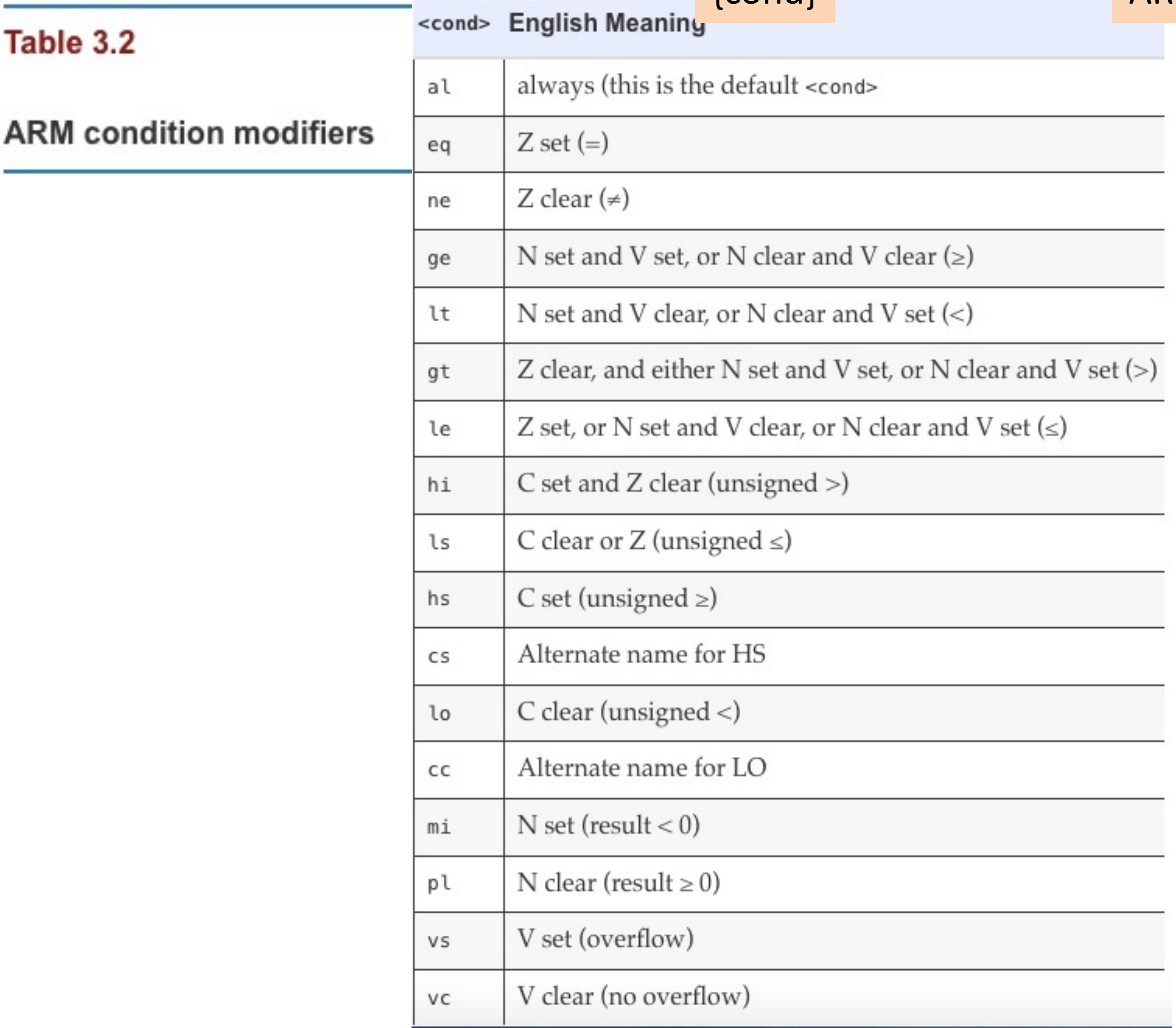


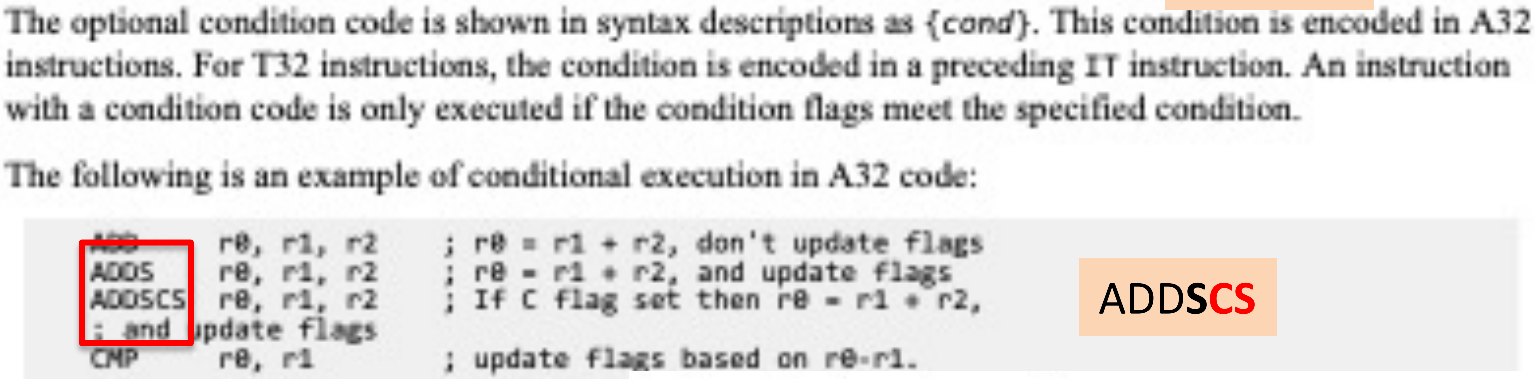
Table 3.2

ARM condition modifiers

<cond>	English Meaning
al	always (this is the default <cond>
eq	Z set (=)
ne	Z clear (≠)
ge	N set and V set, or N clear and V clear (≥)
lt	N set and V clear, or N clear and V set (<)
gt	Z clear, and either N set and V set, or N clear and V set (>)
le	Z set, or N set and V clear, or N clear and V set (≤)
hi	C set and Z clear (unsigned >)
ls	C clear or Z (unsigned ≤)
hs	C set (unsigned ≥)
cs	Alternate name for HS
lo	C clear (unsigned <)
cc	Alternate name for LO
mi	N set (result < 0)
pl	N clear (result ≥ 0)
vs	V set (overflow)
vc	V clear (no overflow)

ARM Conditionals

ARM Ref

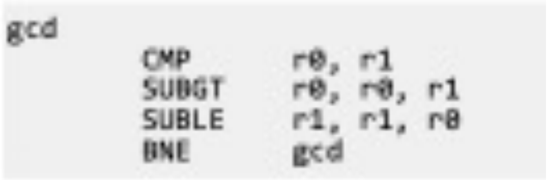


The optional condition code is shown in syntax descriptions as {*cond*}. This condition is encoded in A32 instructions. For T32 instructions, the condition is encoded in a preceding IT instruction. An instruction with a condition code is only executed if the condition flags meet the specified condition.

The following is an example of conditional execution in A32 code:

```
ADD     r0, r1, r2      ; r0 = r1 + r2, don't update flags
ADDS    r0, r1, r2      ; r0 = r1 + r2, and update flags
ADDSCS  r0, r1, r2      ; If C flag set then r0 = r1 + r2,
; and update flags
CMP     r0, r1          ; update flags based on r0-r1.
```

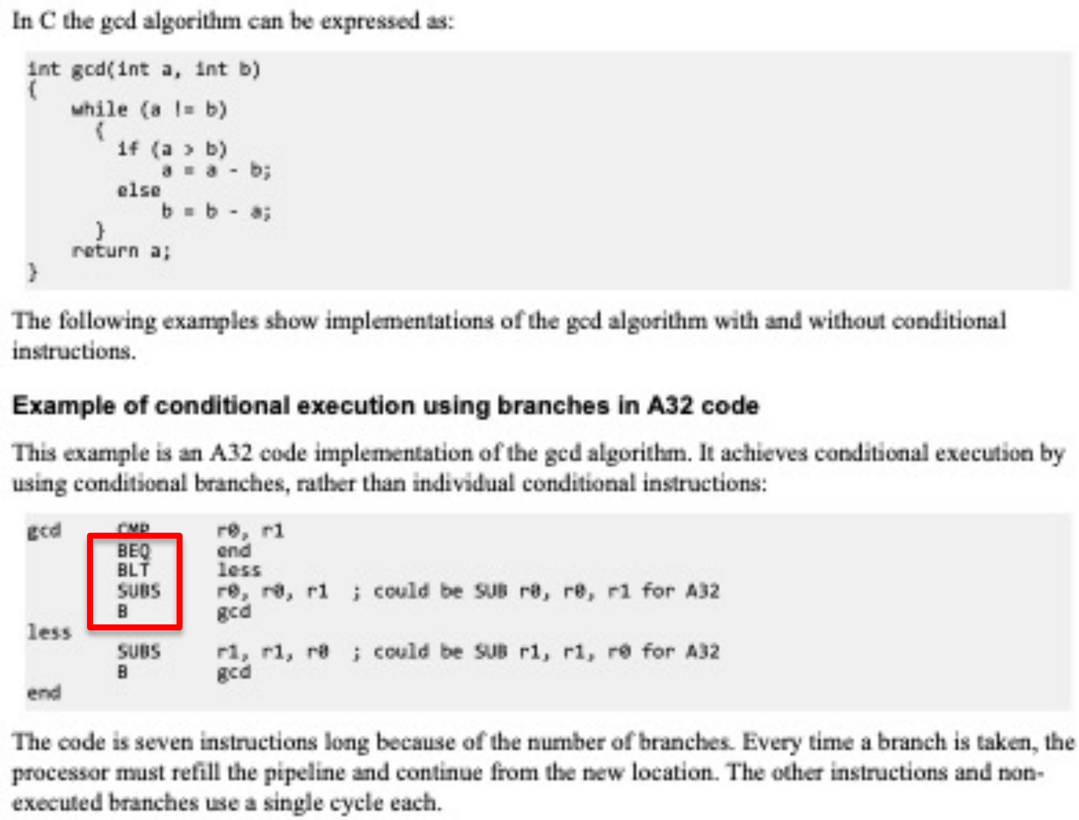
ADDSCS



```
gcd
        CMP     r0, r1
        SUBGT   r0, r0, r1
        SUBLE   r1, r1, r0
        BNE     gcd
```

SUBGT
SUBLE

CMP



In C the gcd algorithm can be expressed as:

```
int gcd(int a, int b)
{
    while (a != b)
      {
        if (a > b)
            a = a - b;
        else
            b = b - a;
      }
    return a;
}
```

The following examples show implementations of the gcd algorithm with and without conditional instructions.

Example of conditional execution using branches in A32 code

This example is an A32 code implementation of the gcd algorithm. It achieves conditional execution by using conditional branches, rather than individual conditional instructions:

```
gcd     CMP     r0, r1
        BEQ     end
        BLT     less
        SUBS    r0, r0, r1  ; could be SUB r0, r0, r1 for A32
        B       gcd
less
        SUBS    r1, r1, r0  ; could be SUB r1, r1, r0 for A32
        B       gcd
end
```

The code is seven instructions long because of the number of branches. Every time a branch is taken, the processor must refill the pipeline and continue from the new location. The other instructions and non-executed branches use a single cycle each.

CSUN
CALIFORNIA
STATE UNIVERSITY
NORTHRIDGE

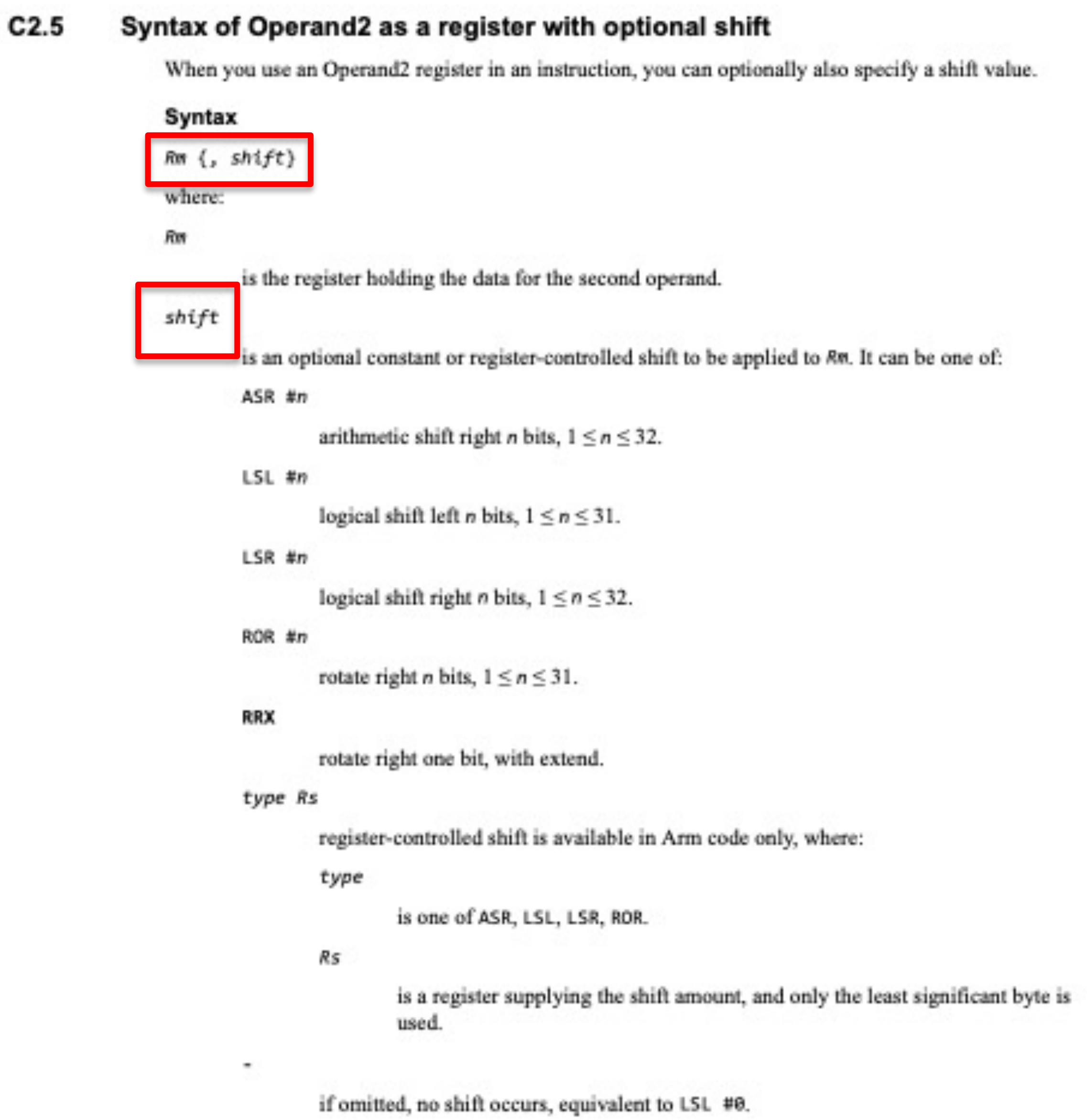
COMP122

ARM Assembly

DR JEFF
SOFTWARE
INDIE APP DEVELOPER
© Jeff Drobman
2016-2022

Operand2+*shift*

ARM Ref



C2.5 Syntax of Operand2 as a register with optional shift

When you use an Operand2 register in an instruction, you can optionally also specify a shift value.

Syntax

`Rm {, shift}`

where:

Rm

is the register holding the data for the second operand.

shift

is an optional constant or register-controlled shift to be applied to *Rm*. It can be one of:

`ASR #n`

arithmetic shift right *n* bits, 1 ≤ *n* ≤ 32.

`LSL #n`

logical shift left *n* bits, 1 ≤ *n* ≤ 31.

`LSR #n`

logical shift right *n* bits, 1 ≤ *n* ≤ 32.

`ROR #n`

rotate right *n* bits, 1 ≤ *n* ≤ 31.

RRX

rotate right one bit, with extend.

type Rs

register-controlled shift is available in Arm code only, where:

type

is one of ASR, LSL, LSR, ROR.

Rs

is a register supplying the shift amount, and only the least significant byte is used.

-

if omitted, no shift occurs, equivalent to `LSL #0`.

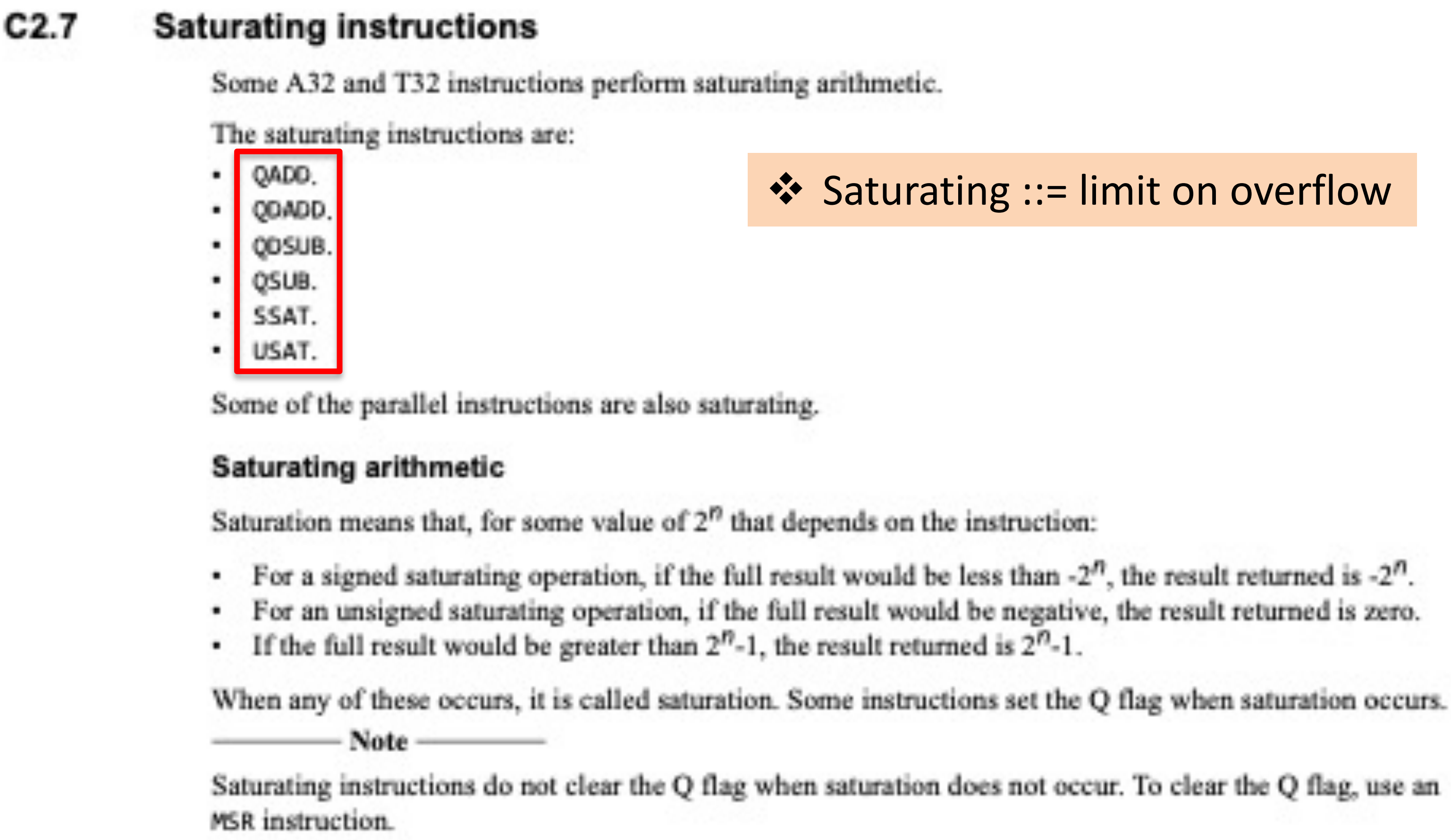
CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP122

ARM Assembly

DR JEFF SOFTWARE
INDIE APP DEVELOPER
© Jeff Drobman
2016-2022

Q

ARM Ref



C2.7 Saturating instructions

Some A32 and T32 instructions perform saturating arithmetic.

The saturating instructions are:

- QADD.
- QDADD.
- QDSUB.
- QSUB.
- SSAT.
- USAT.

Some of the parallel instructions are also saturating.

Saturating arithmetic

Saturation means that, for some value of 2^n that depends on the instruction:

- For a signed saturating operation, if the full result would be less than -2^n, the result returned is -2^n.
- For an unsigned saturating operation, if the full result would be negative, the result returned is zero.
- If the full result would be greater than 2^n-1, the result returned is 2^n-1.

When any of these occurs, it is called saturation. Some instructions set the Q flag when saturation occurs.

——— Note ———

Saturating instructions do not clear the Q flag when saturation does not occur. To clear the Q flag, use an MSR instruction.

❖ Saturating ::= limit on overflow

ARM Assembly

ARM Ref

Branch/jump:
B{cond}
BL{cond}
<no J>

BNE
BEQ
BLNE
BLEQ

returns:
ERET

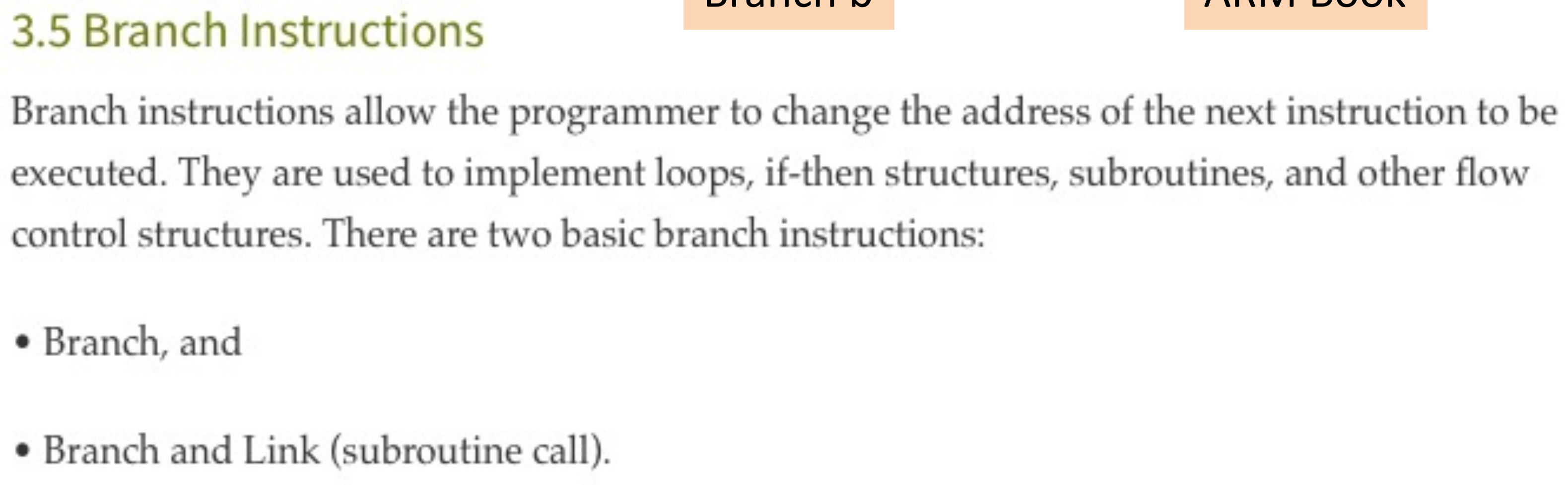
conditionals:
IT (if-then)

debug:
BKPT
DBG (debug)
HLT (halt)

ARM Assembly

Branch b

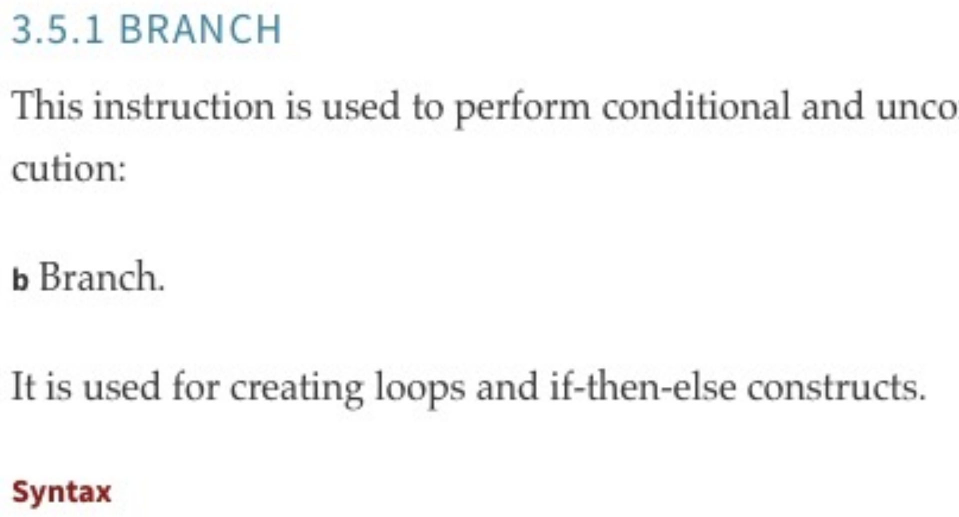
ARM Book



3.5 Branch Instructions

Branch instructions allow the programmer to change the address of the next instruction to be executed. They are used to implement loops, if-then structures, subroutines, and other flow control structures. There are two basic branch instructions:

- Branch, and
- Branch and Link (subroutine call).



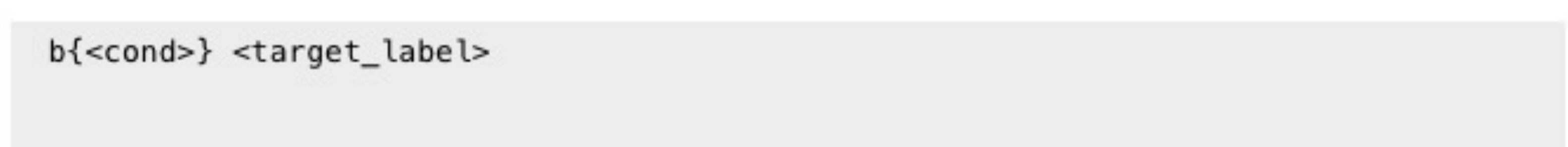
3.5.1 BRANCH

This instruction is used to perform conditional and unconditional branches in program execution:

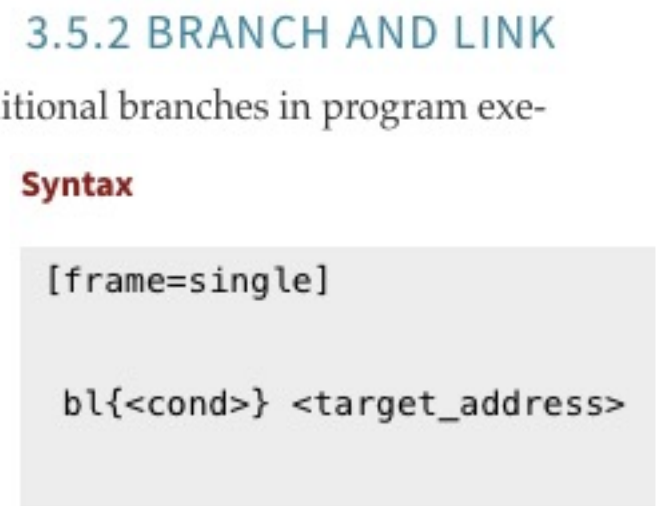
b Branch.

It is used for creating loops and if-then-else constructs.

Syntax



```
b{<cond>} <target_label>
```



3.5.2 BRANCH AND LINK

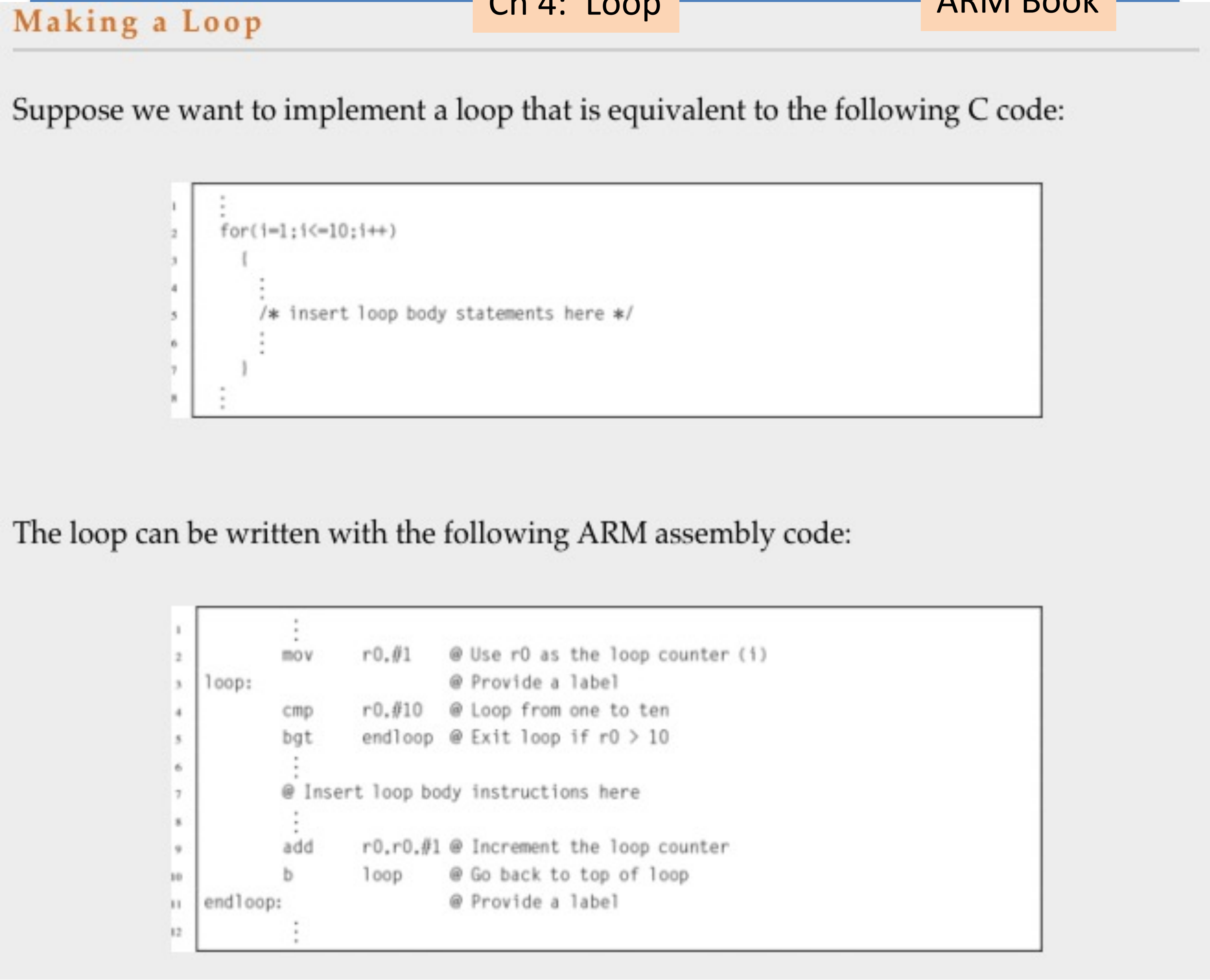
Syntax

```
[frame=single]

 bl{<cond>} <target_address>
```

ARM Assembly

Ch 4: Loop

ARM Book



Making a Loop

Suppose we want to implement a loop that is equivalent to the following C code:

```
 ⋮
 for(i=1;i<=10;i++)
   {
     ⋮
     /* insert loop body statements here */
     ⋮
   }
 ⋮
```

The loop can be written with the following ARM assembly code:

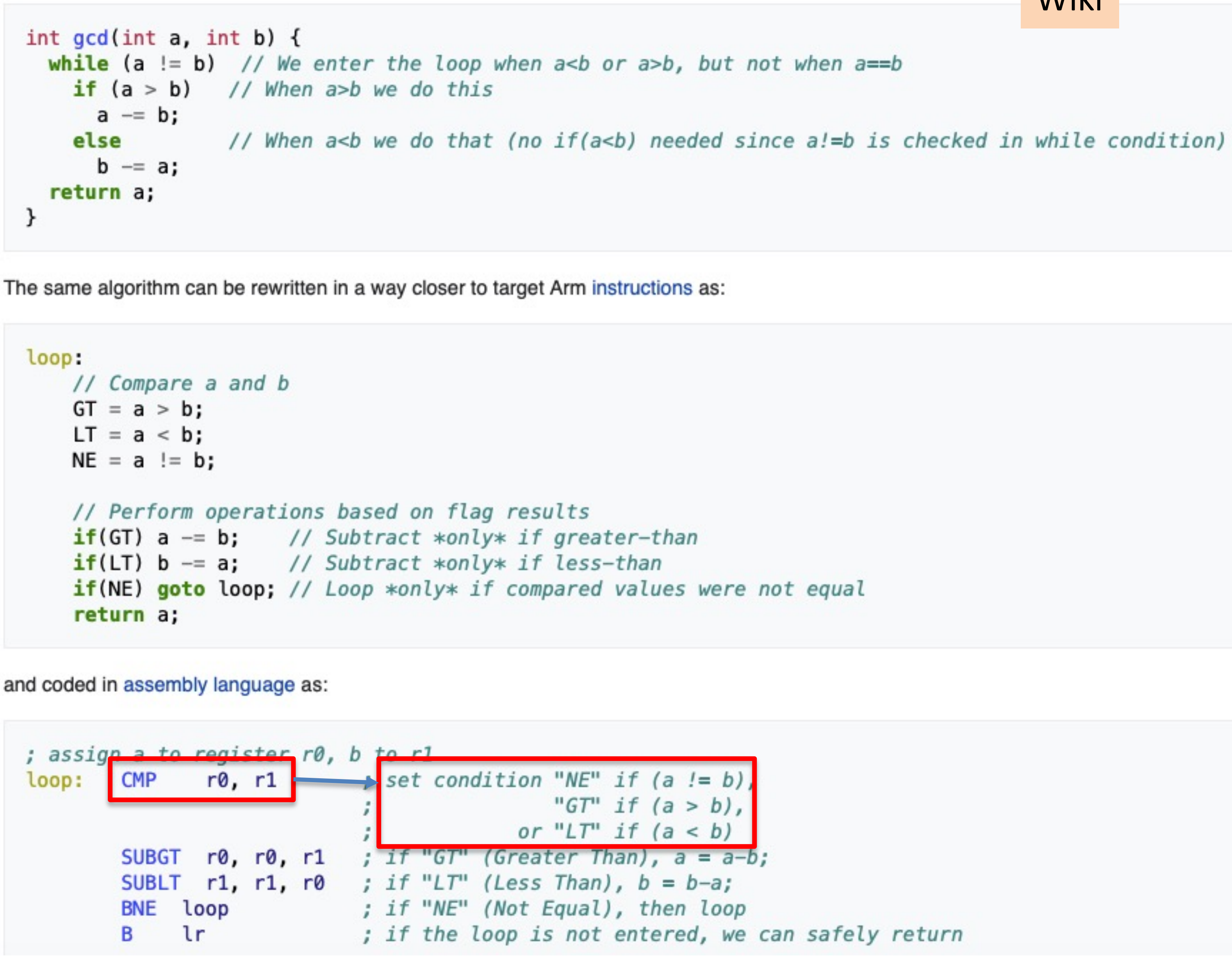
```
          ⋮
          mov     r0,#1      @ Use r0 as the loop counter (i)
loop:                        @ Provide a label
          cmp     r0,#10     @ Loop from one to ten
          bgt     endloop    @ Exit loop if r0 > 10
          ⋮
          @ Insert loop body instructions here
          ⋮
          add     r0,r0,#1   @ Increment the loop counter
          b       loop       @ Go back to top of loop
endloop:                     @ Provide a label
          ⋮
```

CSUN
CALIFORNIA
STATE UNIVERSITY
NORTHRIDGE
COMP122

ARM Assembly

DR JEFF
SOFTWARE
INDIE APP DEVELOPER
© Jeff Drobman
2016-2022

Wiki



```
int gcd(int a, int b) {
  while (a != b)  // We enter the loop when a<b or a>b, but not when a==b
    if (a > b)   // When a>b we do this
      a -= b;
    else         // When a<b we do that (no if(a<b) needed since a!=b is checked in while condition)
      b -= a;
  return a;
}
```

The same algorithm can be rewritten in a way closer to target Arm instructions as:

```
loop:
    // Compare a and b
    GT = a > b;
    LT = a < b;
    NE = a != b;

    // Perform operations based on flag results
    if(GT) a -= b;    // Subtract *only* if greater-than
    if(LT) b -= a;    // Subtract *only* if less-than
    if(NE) goto loop; // Loop *only* if compared values were not equal
    return a;
```

and coded in assembly language as:

```
; assign a to register r0, b to r1
loop:   CMP    r0, r1       ; set condition "NE" if (a != b),
                            ;                "GT" if (a > b),
                            ;             or "LT" if (a < b)
        SUBGT  r0, r0, r1   ; if "GT" (Greater Than), a = a-b;
        SUBLT  r1, r1, r0   ; if "LT" (Less Than), b = b-a;
        BNE  loop           ; if "NE" (Not Equal), then loop
        B    lr             ; if the loop is not entered, we can safely return
```

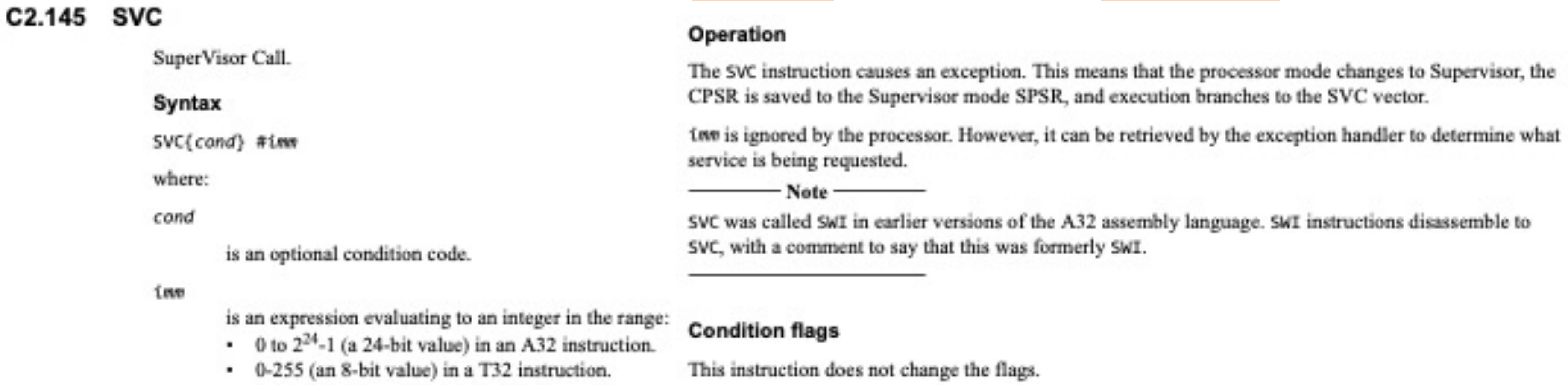
CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP122

ARM Assembly

DR JEFF SOFTWARE
INDIE APP DEVELOPER
© Jeff Drobman
2016-2022

Syscall

ARM Ref



C2.145 SVC

SuperVisor Call.

Syntax

`SVC{cond} #imm`

where:

cond

is an optional condition code.

imm

is an expression evaluating to an integer in the range:

- 0 to 2^{24}-1 (a 24-bit value) in an A32 instruction.
- 0-255 (an 8-bit value) in a T32 instruction.

Operation

The SVC instruction causes an exception. This means that the processor mode changes to Supervisor, the CPSR is saved to the Supervisor mode SPSR, and execution branches to the SVC vector.

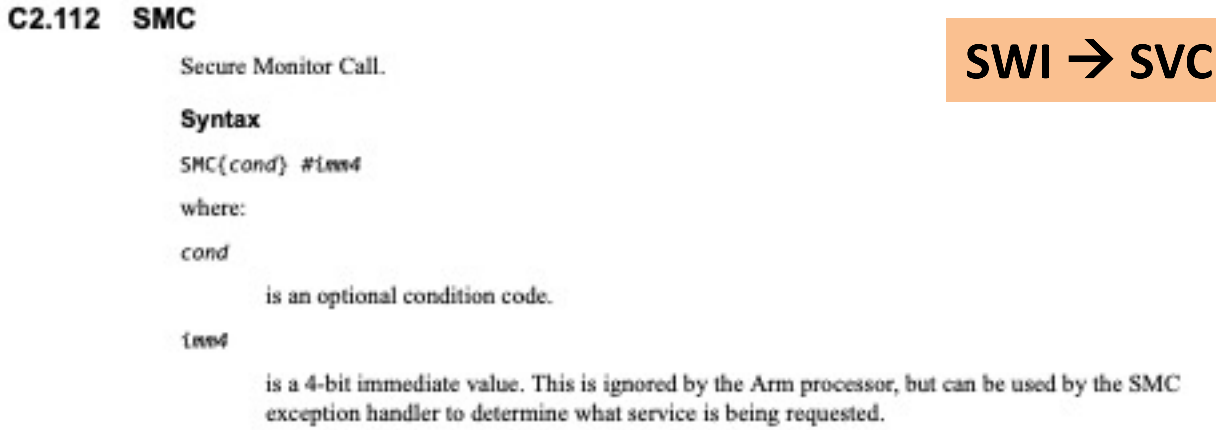
imm is ignored by the processor. However, it can be retrieved by the exception handler to determine what service is being requested.

Note

SVC was called SWI in earlier versions of the A32 assembly language. SWI instructions disassemble to SVC, with a comment to say that this was formerly SWI.

Condition flags

This instruction does not change the flags.



C2.112 SMC

Secure Monitor Call.

Syntax

`SMC{cond} #imm4`

where:

cond

is an optional condition code.

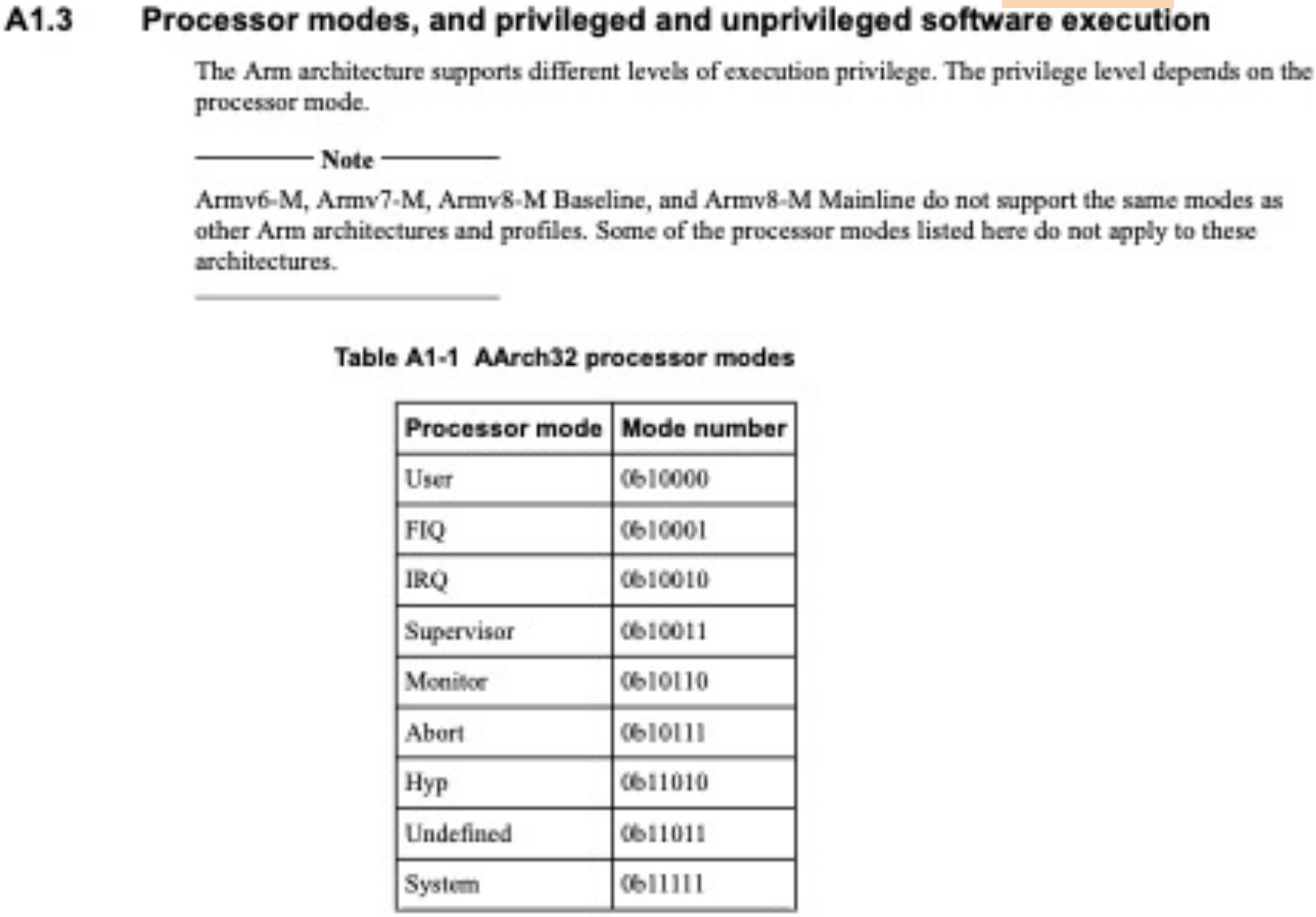
imm4

is a 4-bit immediate value. This is ignored by the Arm processor, but can be used by the SMC exception handler to determine what service is being requested.

SWI → SVC

ARM Assembly

ARM Ref



A1.3 Processor modes, and privileged and unprivileged software execution

The Arm architecture supports different levels of execution privilege. The privilege level depends on the processor mode.

Note

Armv6-M, Armv7-M, Armv8-M Baseline, and Armv8-M Mainline do not support the same modes as other Arm architectures and profiles. Some of the processor modes listed here do not apply to these architectures.

Table A1-1 AArch32 processor modes

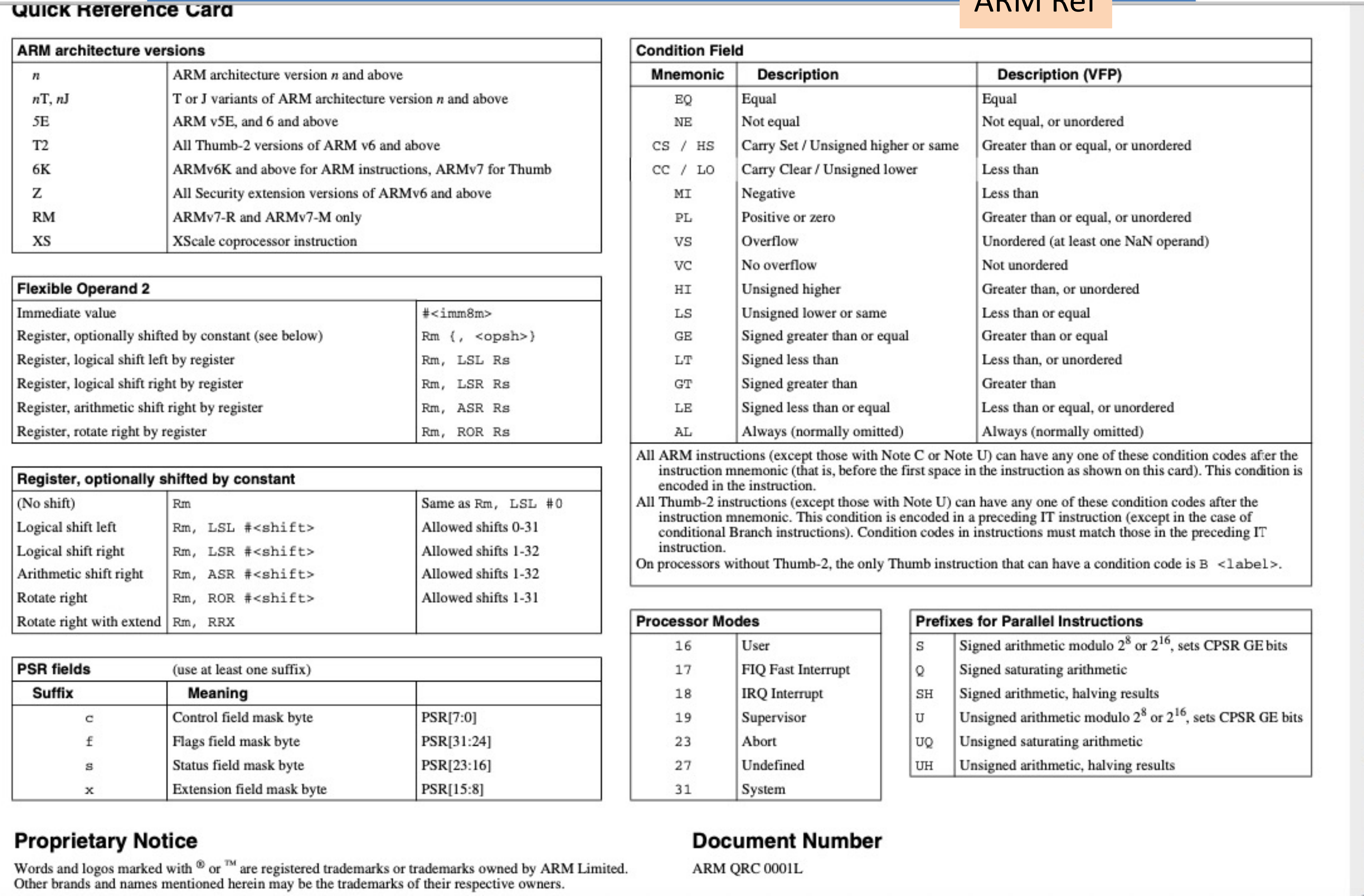
Processor mode	Mode number
User	0b10000
FIQ	0b10001
IRQ	0b10010
Supervisor	0b10011
Monitor	0b10110
Abort	0b10111
Hyp	0b11010
Undefined	0b11011
System	0b11111

ARM ISA

ARM
Instruction Set

ARM Assembly

ARM Ref



Quick Reference Card

ARM architecture versions	
n	ARM architecture version *n* and above
*n*T, *n*J	T or J variants of ARM architecture version *n* and above
5E	ARM v5E, and 6 and above
T2	All Thumb-2 versions of ARM v6 and above
6K	ARMv6K and above for ARM instructions, ARMv7 for Thumb
Z	All Security extension versions of ARMv6 and above
RM	ARMv7-R and ARMv7-M only
XS	XScale coprocessor instruction

Flexible Operand 2	
Immediate value	`#<imm8m>`
Register, optionally shifted by constant (see below)	`Rm {, <opsh>}`
Register, logical shift left by register	`Rm, LSL Rs`
Register, logical shift right by register	`Rm, LSR Rs`
Register, arithmetic shift right by register	`Rm, ASR Rs`
Register, rotate right by register	`Rm, ROR Rs`

Register, optionally shifted by constant		
(No shift)	`Rm`	Same as `Rm, LSL #0`
Logical shift left	`Rm, LSL #<shift>`	Allowed shifts 0-31
Logical shift right	`Rm, LSR #<shift>`	Allowed shifts 1-32
Arithmetic shift right	`Rm, ASR #<shift>`	Allowed shifts 1-32
Rotate right	`Rm, ROR #<shift>`	Allowed shifts 1-31
Rotate right with extend	`Rm, RRX`	

PSR fields	(use at least one suffix)	
Suffix	**Meaning**	
c	Control field mask byte	PSR[7:0]
f	Flags field mask byte	PSR[31:24]
s	Status field mask byte	PSR[23:16]
x	Extension field mask byte	PSR[15:8]

Condition Field

Mnemonic	Description	Description (VFP)
EQ	Equal	Equal
NE	Not equal	Not equal, or unordered
CS / HS	Carry Set / Unsigned higher or same	Greater than or equal, or unordered
CC / LO	Carry Clear / Unsigned lower	Less than
MI	Negative	Less than
PL	Positive or zero	Greater than or equal, or unordered
VS	Overflow	Unordered (at least one NaN operand)
VC	No overflow	Not unordered
HI	Unsigned higher	Greater than, or unordered
LS	Unsigned lower or same	Less than or equal
GE	Signed greater than or equal	Greater than or equal
LT	Signed less than	Less than, or unordered
GT	Signed greater than	Greater than
LE	Signed less than or equal	Less than or equal, or unordered
AL	Always (normally omitted)	Always (normally omitted)

All ARM instructions (except those with Note C or Note U) can have any one of these condition codes after the instruction mnemonic (that is, before the first space in the instruction as shown on this card). This condition is encoded in the instruction.

All Thumb-2 instructions (except those with Note U) can have any one of these condition codes after the instruction mnemonic. This condition is encoded in a preceding IT instruction (except in the case of conditional Branch instructions). Condition codes in instructions must match those in the preceding IT instruction.

On processors without Thumb-2, the only Thumb instruction that can have a condition code is `B <label>`.

Processor Modes	
16	User
17	FIQ Fast Interrupt
18	IRQ Interrupt
19	Supervisor
23	Abort
27	Undefined
31	System

Prefixes for Parallel Instructions	
S	Signed arithmetic modulo 2^8 or 2^{16}, sets CPSR GE bits
Q	Signed saturating arithmetic
SH	Signed arithmetic, halving results
U	Unsigned arithmetic modulo 2^8 or 2^{16}, sets CPSR GE bits
UQ	Unsigned saturating arithmetic
UH	Unsigned arithmetic, halving results

Proprietary Notice

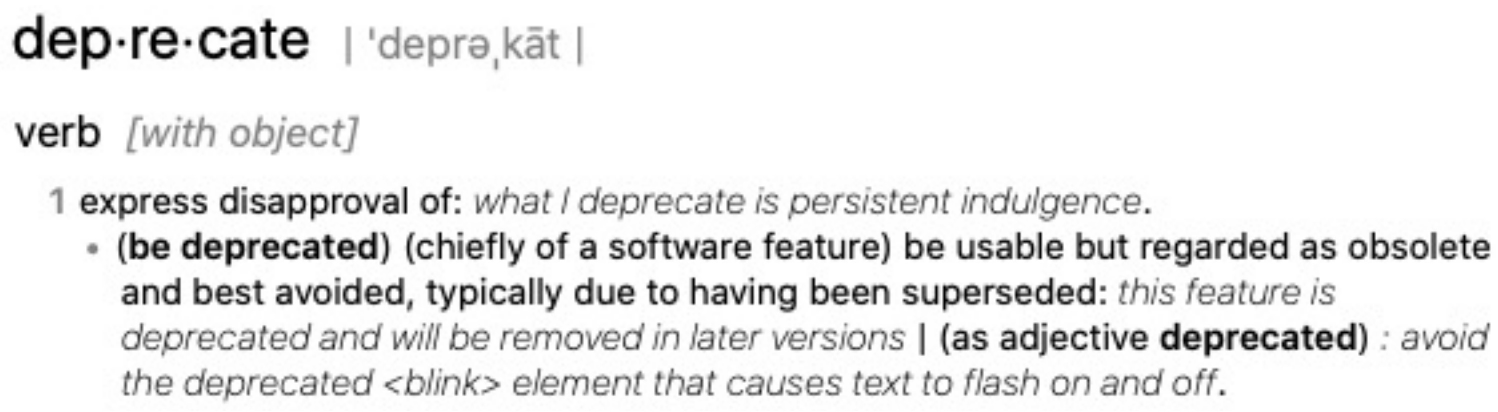
Words and logos marked with ® or ™ are registered trademarks or trademarks owned by ARM Limited. Other brands and names mentioned herein may be the trademarks of their respective owners.

Document Number

ARM QRC 0001L

ARM Assembly

ARM Ref



dep·re·cate | ˈdeprəˌkāt |

verb *[with object]*

1 **express disapproval of:** *what I deprecate is persistent indulgence.*

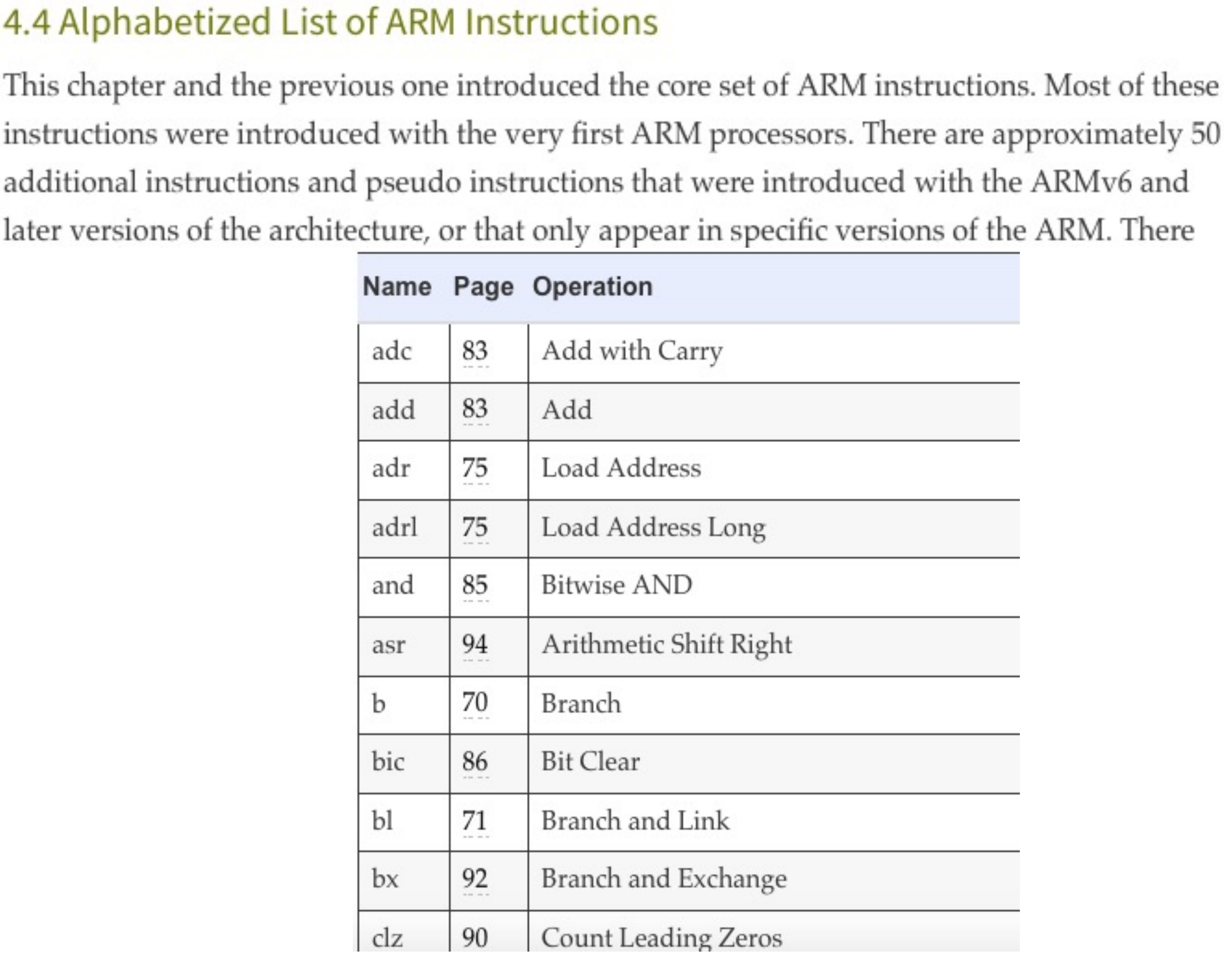
- (**be deprecated**) (chiefly of a software feature) be usable but regarded as obsolete and best avoided, typically due to having been superseded: *this feature is deprecated and will be removed in later versions* | (as adjective **deprecated**) : *avoid the deprecated <blink> element that causes text to flash on and off.*

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP122

ARM Assembly

Ch 4: I-Set

ARM Book



4.4 Alphabetized List of ARM Instructions

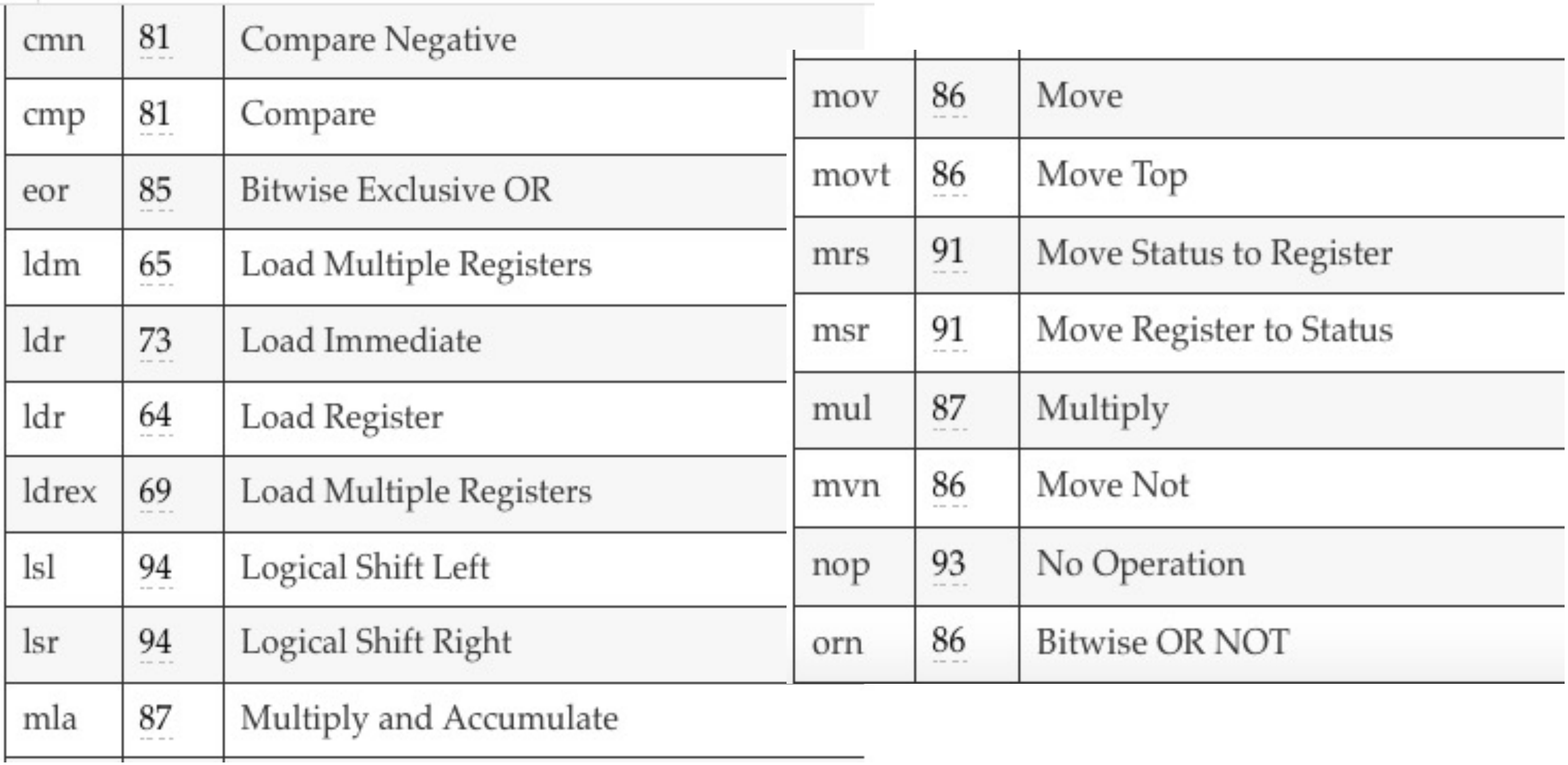
This chapter and the previous one introduced the core set of ARM instructions. Most of these instructions were introduced with the very first ARM processors. There are approximately 50 additional instructions and pseudo instructions that were introduced with the ARMv6 and later versions of the architecture, or that only appear in specific versions of the ARM. There

Name	Page	Operation
adc	83	Add with Carry
add	83	Add
adr	75	Load Address
adrl	75	Load Address Long
and	85	Bitwise AND
asr	94	Arithmetic Shift Right
b	70	Branch
bic	86	Bit Clear
bl	71	Branch and Link
bx	92	Branch and Exchange
clz	90	Count Leading Zeros

ARM Assembly

Ch 4: I-Set

ARM Book

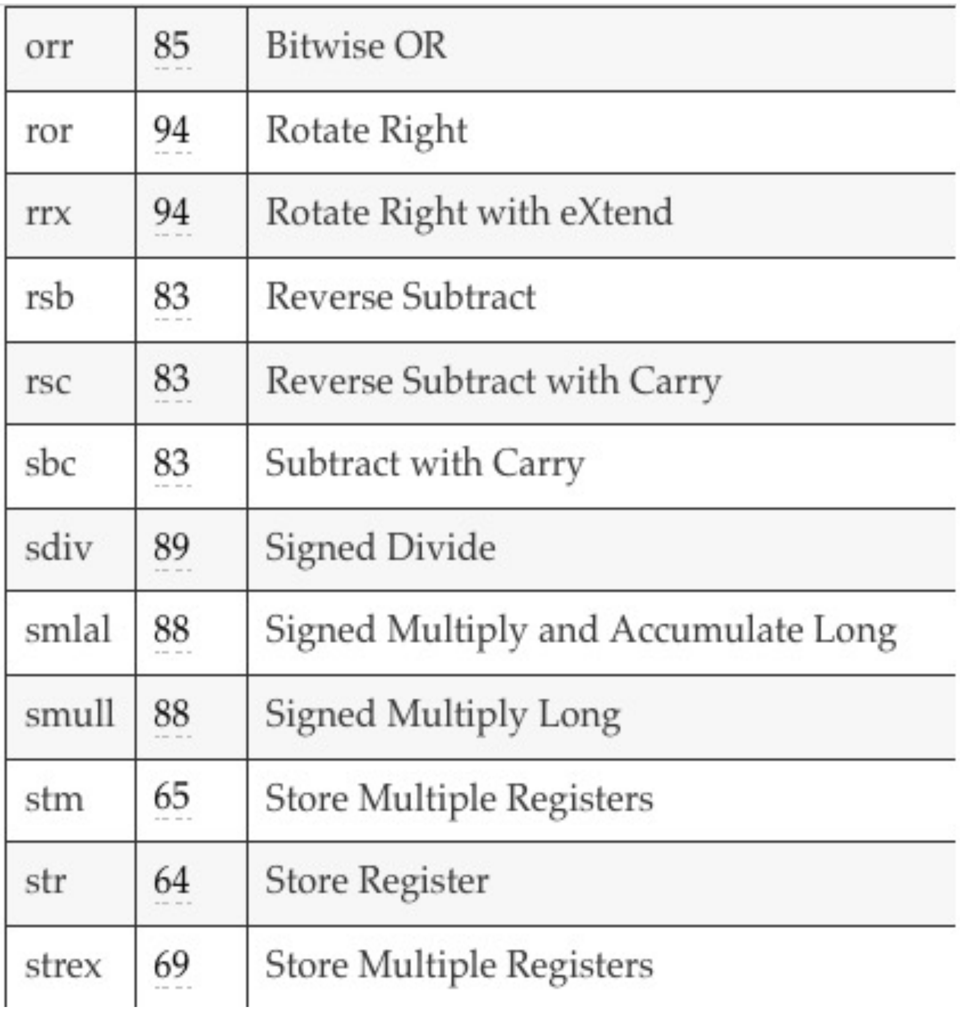


cmn	81	Compare Negative
cmp	81	Compare
eor	85	Bitwise Exclusive OR
ldm	65	Load Multiple Registers
ldr	73	Load Immediate
ldr	64	Load Register
ldrex	69	Load Multiple Registers
lsl	94	Logical Shift Left
lsr	94	Logical Shift Right
mla	87	Multiply and Accumulate
mov	86	Move
movt	86	Move Top
mrs	91	Move Status to Register
msr	91	Move Register to Status
mul	87	Multiply
mvn	86	Move Not
nop	93	No Operation
orn	86	Bitwise OR NOT

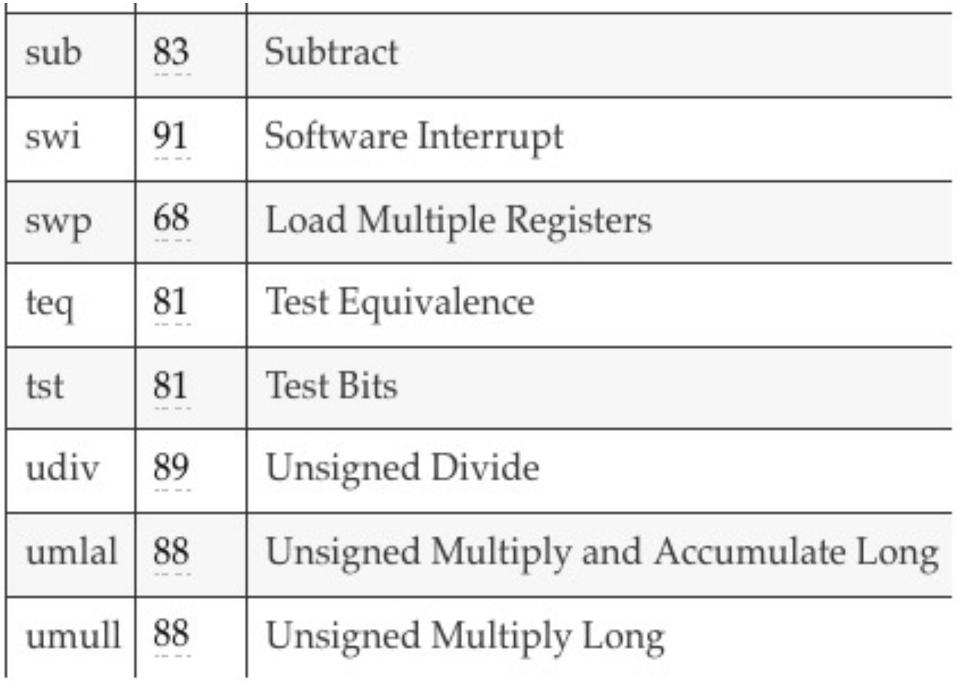
ARM Assembly

Ch 4: I-Set

ARM Book



orr	85	Bitwise OR
ror	94	Rotate Right
rrx	94	Rotate Right with eXtend
rsb	83	Reverse Subtract
rsc	83	Reverse Subtract with Carry
sbc	83	Subtract with Carry
sdiv	89	Signed Divide
smlal	88	Signed Multiply and Accumulate Long
smull	88	Signed Multiply Long
stm	65	Store Multiple Registers
str	64	Store Register
strex	69	Store Multiple Registers



sub	83	Subtract
swi	91	Software Interrupt
swp	68	Load Multiple Registers
teq	81	Test Equivalence
tst	81	Test Bits
udiv	89	Unsigned Divide
umlal	88	Unsigned Multiply and Accumulate Long
umull	88	Unsigned Multiply Long

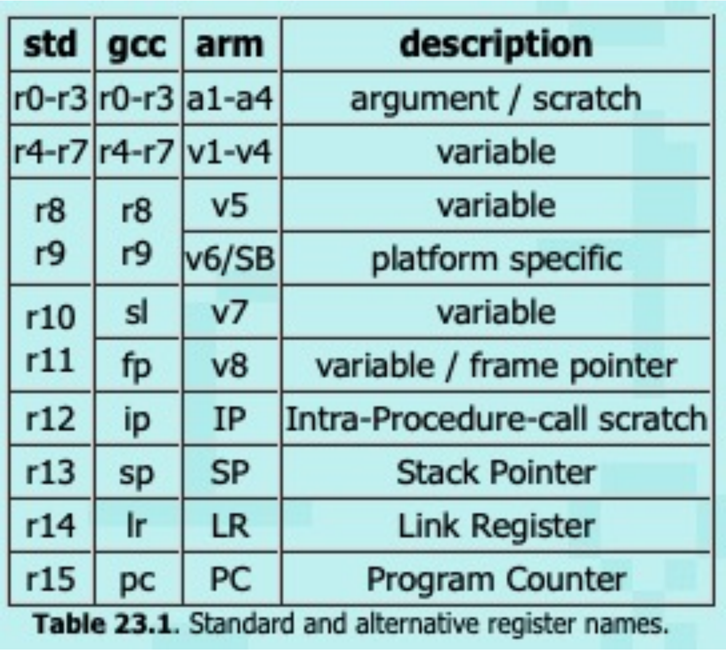
ARM Assembly

coranac.com

Website: coranac.com

Registers

coranac.com



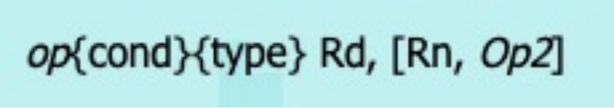
std	gcc	arm	description
r0-r3	r0-r3	a1-a4	argument / scratch
r4-r7	r4-r7	v1-v4	variable
r8	r8	v5	variable
r9	r9	v6/SB	platform specific
r10	sl	v7	variable
r11	fp	v8	variable / frame pointer
r12	ip	IP	Intra-Procedure-call scratch
r13	sp	SP	Stack Pointer
r14	lr	LR	Link Register
r15	pc	PC	Program Counter

Table 23.1. Standard and alternative register names.

Load/Store

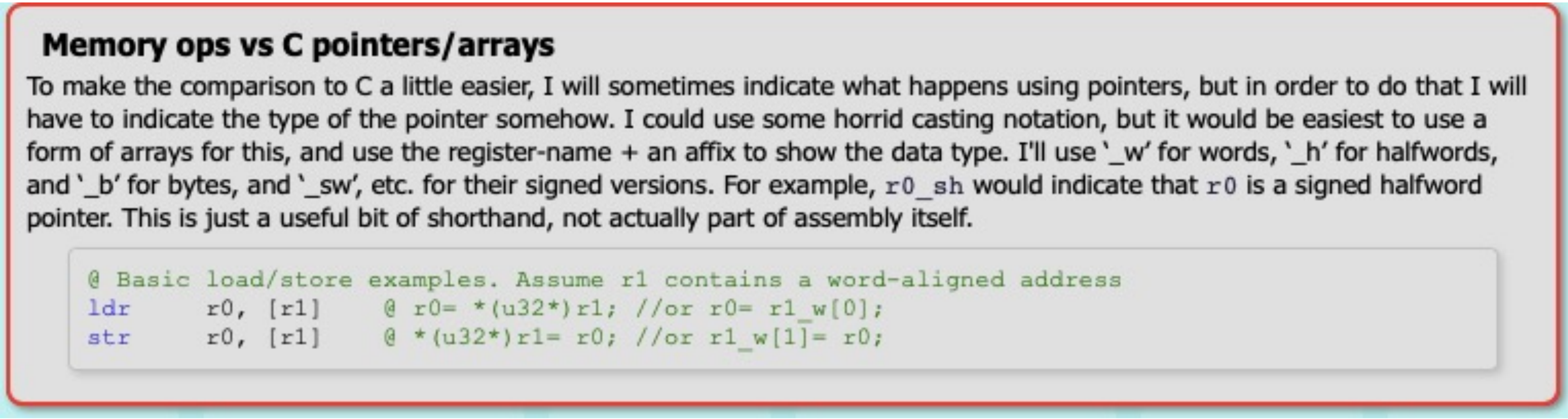
coranac.com

23.3.3. Memory instructions: load and store


op{cond}{type} Rd, [Rn, *Op2*]

OP2 ::= {Rs, Rs+offset, <const/immed>}

{cond} ::= {EQ, NE, GE, GT, LE, LT}
{type} ::= {B, SB, H, SH, W, SW}


Memory ops vs C pointers/arrays

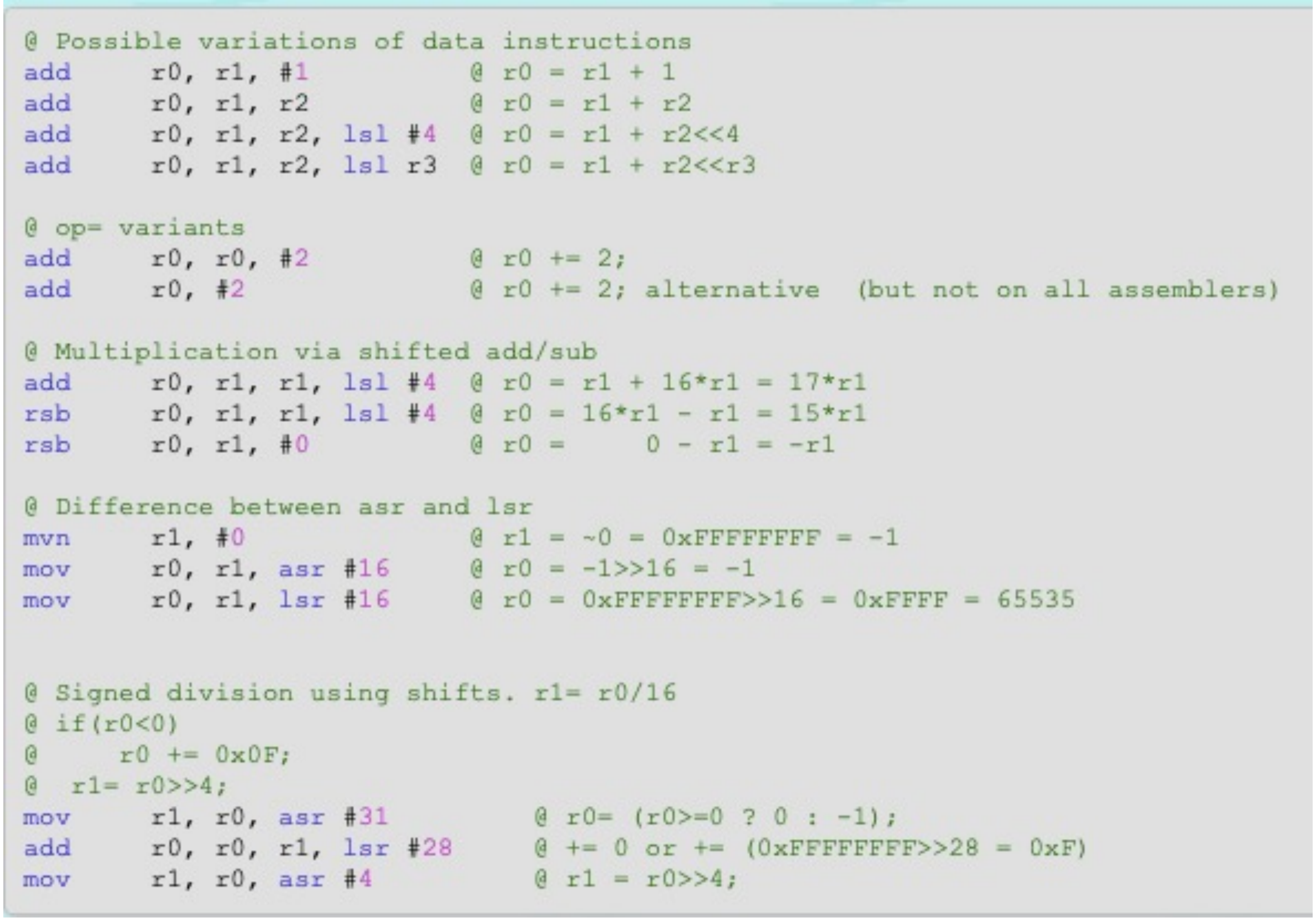
To make the comparison to C a little easier, I will sometimes indicate what happens using pointers, but in order to do that I will have to indicate the type of the pointer somehow. I could use some horrid casting notation, but it would be easiest to use a form of arrays for this, and use the register-name + an affix to show the data type. I'll use '_w' for words, '_h' for halfwords, and '_b' for bytes, and '_sw', etc. for their signed versions. For example, `r0_sh` would indicate that `r0` is a signed halfword pointer. This is just a useful bit of shorthand, not actually part of assembly itself.

```
@ Basic load/store examples. Assume r1 contains a word-aligned address
ldr     r0, [r1]     @ r0= *(u32*)r1; //or r0= r1_w[0];
str     r0, [r1]     @ *(u32*)r1= r0; //or r1_w[1]= r0;
```

push and pop are not universal ARM instructions

Add

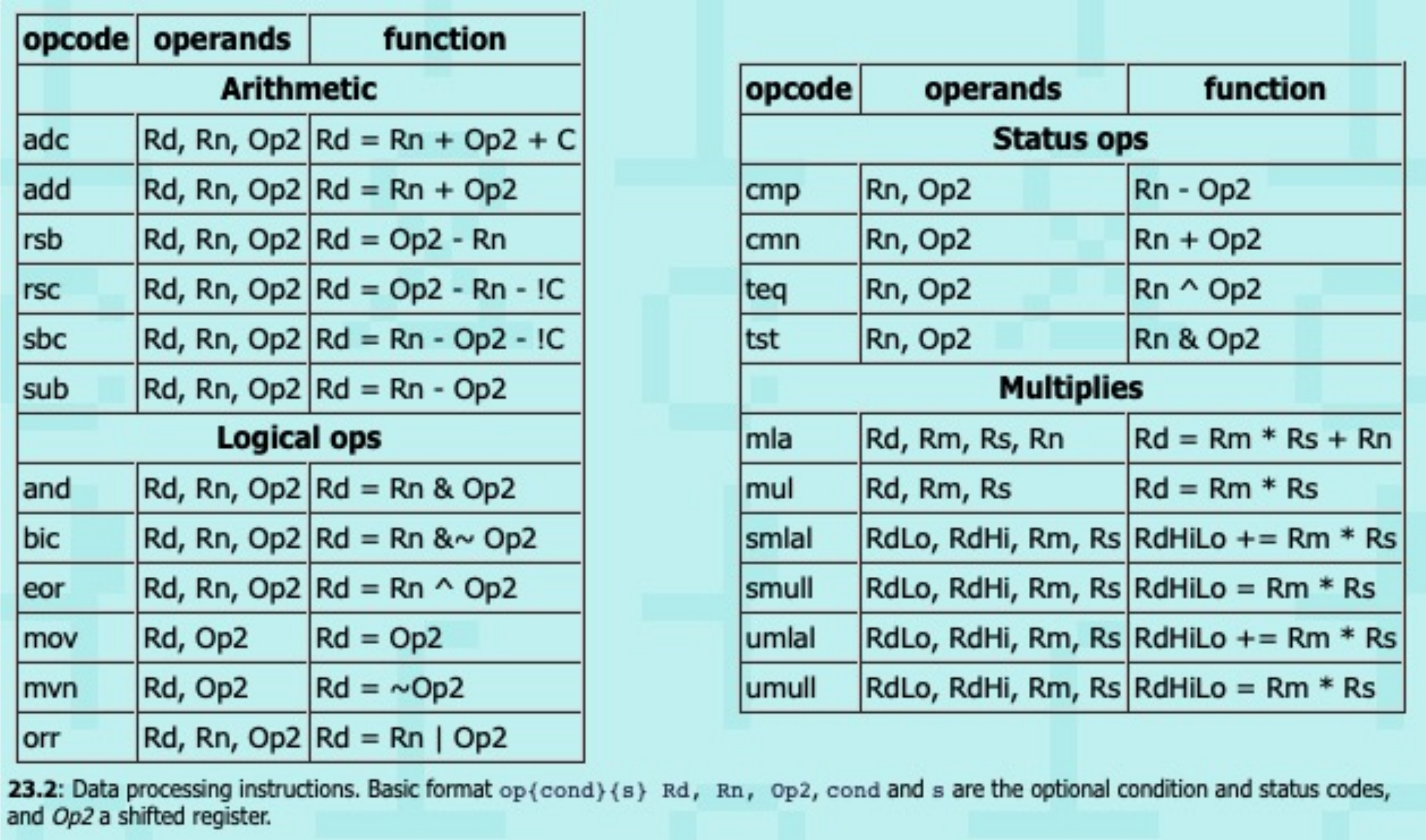
coranac.com



```
@ Possible variations of data instructions
add     r0, r1, #1              @ r0 = r1 + 1
add     r0, r1, r2              @ r0 = r1 + r2
add     r0, r1, r2, lsl #4      @ r0 = r1 + r2<<4
add     r0, r1, r2, lsl r3      @ r0 = r1 + r2<<r3

@ op= variants
add     r0, r0, #2              @ r0 += 2;
add     r0, #2                  @ r0 += 2; alternative  (but not on all assemblers)

@ Multiplication via shifted add/sub
add     r0, r1, r1, lsl #4      @ r0 = r1 + 16*r1 = 17*r1
rsb     r0, r1, r1, lsl #4      @ r0 = 16*r1 - r1 = 15*r1
rsb     r0, r1, #0              @ r0 =     0 - r1 = -r1

@ Difference between asr and lsr
mvn     r1, #0                  @ r1 = ~0 = 0xFFFFFFFF = -1
mov     r0, r1, asr #16         @ r0 = -1>>16 = -1
mov     r0, r1, lsr #16         @ r0 = 0xFFFFFFFF>>16 = 0xFFFF = 65535


@ Signed division using shifts. r1= r0/16
@ if(r0<0)
@     r0 += 0x0F;
@  r1= r0>>4;
mov     r1, r0, asr #31             @ r0= (r0>=0 ? 0 : -1);
add     r0, r0, r1, lsr #28         @ += 0 or += (0xFFFFFFFF>>28 = 0xF)
mov     r1, r0, asr #4              @ r1 = r0>>4;
```

ALU Ops

coranac.com



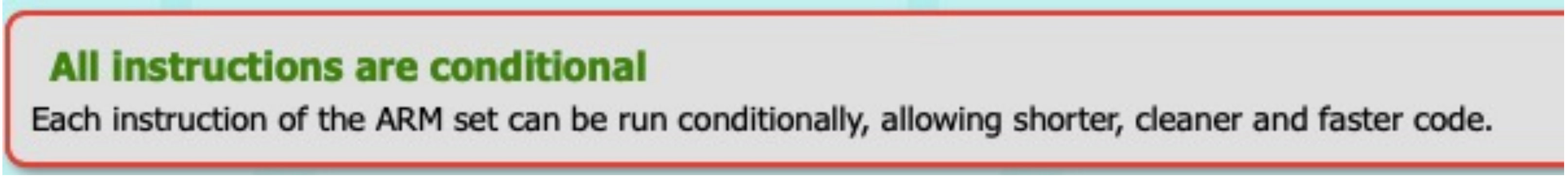
opcode	operands	function
Arithmetic		
adc	Rd, Rn, Op2	Rd = Rn + Op2 + C
add	Rd, Rn, Op2	Rd = Rn + Op2
rsb	Rd, Rn, Op2	Rd = Op2 - Rn
rsc	Rd, Rn, Op2	Rd = Op2 - Rn - !C
sbc	Rd, Rn, Op2	Rd = Rn - Op2 - !C
sub	Rd, Rn, Op2	Rd = Rn - Op2
Logical ops		
and	Rd, Rn, Op2	Rd = Rn & Op2
bic	Rd, Rn, Op2	Rd = Rn &~ Op2
eor	Rd, Rn, Op2	Rd = Rn ^ Op2
mov	Rd, Op2	Rd = Op2
mvn	Rd, Op2	Rd = ~Op2
orr	Rd, Rn, Op2	Rd = Rn | Op2

opcode	operands	function
Status ops		
cmp	Rn, Op2	Rn - Op2
cmn	Rn, Op2	Rn + Op2
teq	Rn, Op2	Rn ^ Op2
tst	Rn, Op2	Rn & Op2
Multiplies		
mla	Rd, Rm, Rs, Rn	Rd = Rm * Rs + Rn
mul	Rd, Rm, Rs	Rd = Rm * Rs
smlal	RdLo, RdHi, Rm, Rs	RdHiLo += Rm * Rs
smull	RdLo, RdHi, Rm, Rs	RdHiLo = Rm * Rs
umlal	RdLo, RdHi, Rm, Rs	RdHiLo += Rm * Rs
umull	RdLo, RdHi, Rm, Rs	RdHiLo = Rm * Rs

23.2: Data processing instructions. Basic format `op{cond}{s} Rd, Rn, Op2`, `cond` and `s` are the optional condition and status codes, and *Op2* a shifted register.

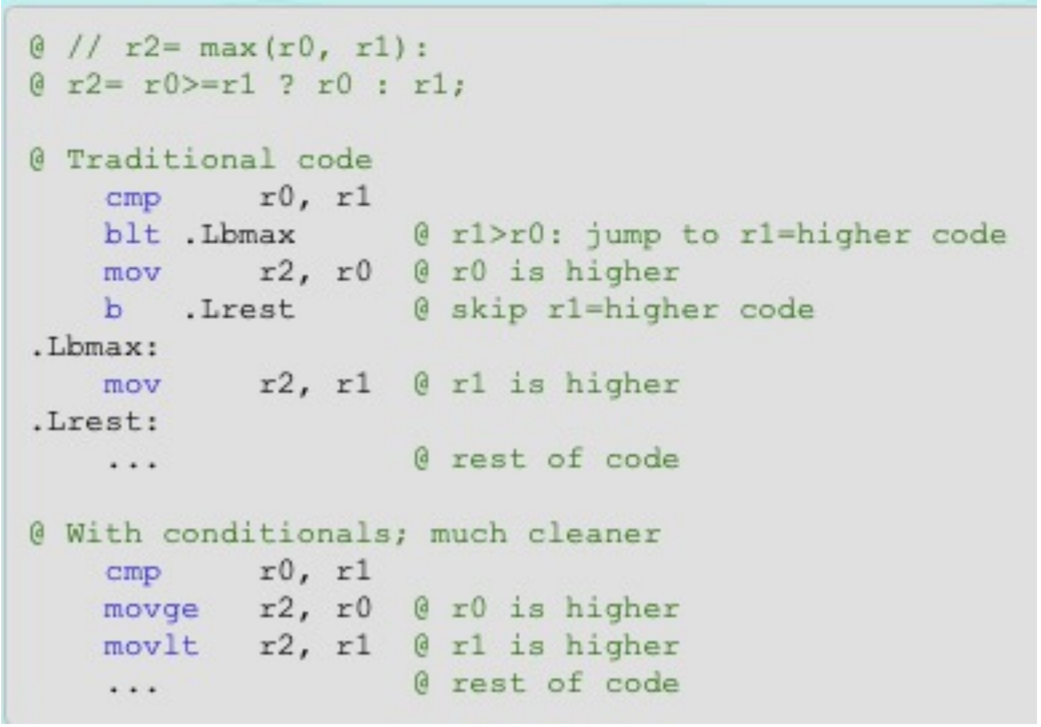
Conditional

coranac.com

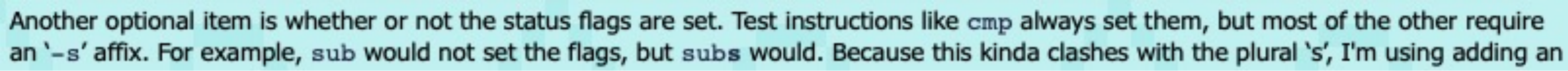


All instructions are conditional

Each instruction of the ARM set can be run conditionally, allowing shorter, cleaner and faster code.



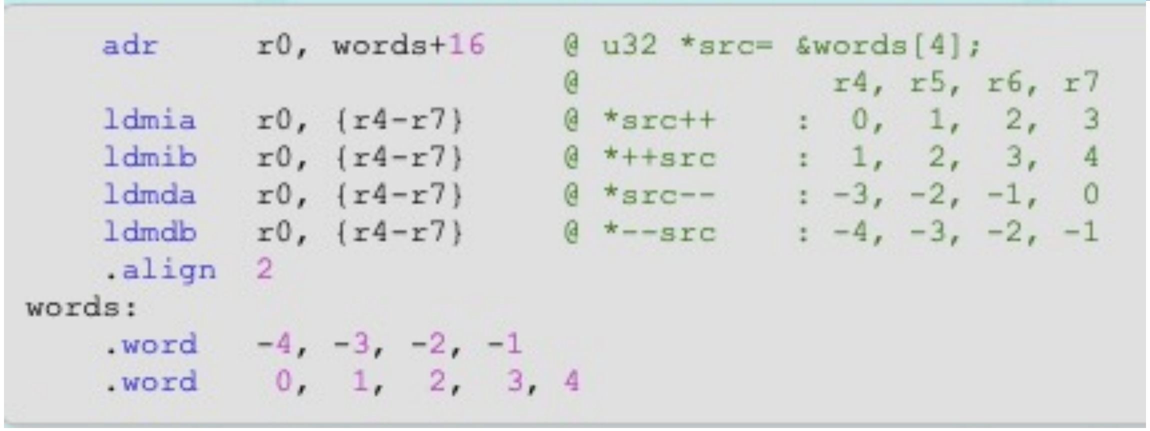
```
@ // r2= max(r0, r1):
@ r2= r0>=r1 ? r0 : r1;

@ Traditional code
    cmp     r0, r1
    blt .Lbmax      @ r1>r0: jump to r1=higher code
    mov     r2, r0  @ r0 is higher
    b   .Lrest      @ skip r1=higher code
.Lbmax:
    mov     r2, r1  @ r1 is higher
.Lrest:
    ...             @ rest of code

@ With conditionals; much cleaner
    cmp     r0, r1
    movge   r2, r0  @ r0 is higher
    movlt   r2, r1  @ r1 is higher
    ...             @ rest of code
```



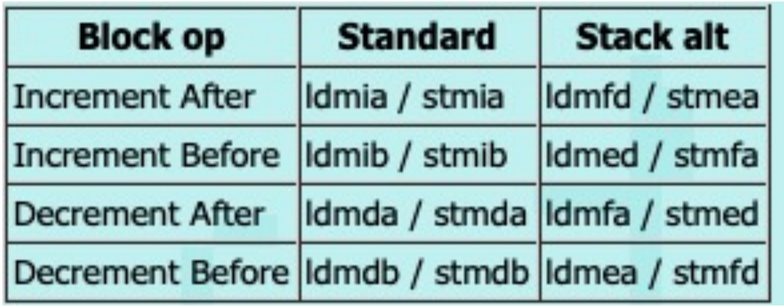
Another optional item is whether or not the status flags are set. Test instructions like `cmp` always set them, but most of the other require an '`-s`' affix. For example, `sub` would not set the flags, but `subs` would. Because this kinda clashes with the plural 's', I'm using adding an

Load Multiple

coranac.com



```
    adr     r0, words+16    @ u32 *src= &words[4];
                            @            r4, r5, r6, r7
    ldmia   r0, {r4-r7}     @ *src++    :  0,  1,  2,  3
    ldmib   r0, {r4-r7}     @ *++src    :  1,  2,  3,  4
    ldmda   r0, {r4-r7}     @ *src--    : -3, -2, -1,  0
    ldmdb   r0, {r4-r7}     @ *--src    : -4, -3, -2, -1
    .align  2
words:
    .word   -4, -3, -2, -1
    .word    0,  1,  2,  3, 4
```



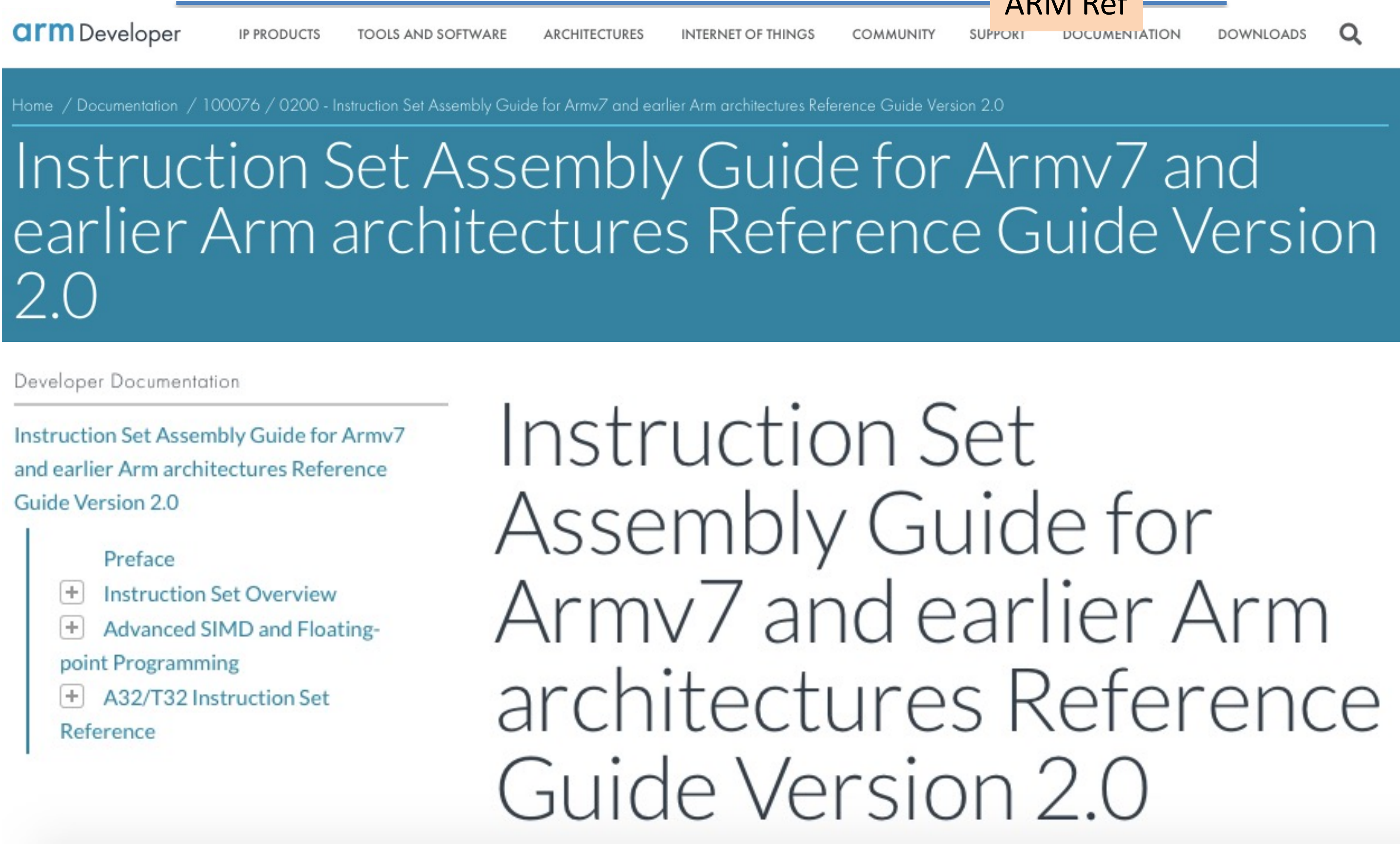
Block op	Standard	Stack alt
Increment After	ldmia / stmia	ldmfd / stmea
Increment Before	ldmib / stmib	ldmed / stmfa
Decrement After	ldmda / stmda	ldmfa / stmed
Decrement Before	ldmdb / stmdb	ldmea / stmfd

ARM Assembly

ARM Ref Manual

ARM Assembly

ARM Ref



arm Developer | IP PRODUCTS | TOOLS AND SOFTWARE | ARCHITECTURES | INTERNET OF THINGS | COMMUNITY | SUPPORT | DOCUMENTATION | DOWNLOADS

Home / Documentation / 100076 / 0200 - Instruction Set Assembly Guide for Armv7 and earlier Arm architectures Reference Guide Version 2.0

Instruction Set Assembly Guide for Armv7 and earlier Arm architectures Reference Guide Version 2.0

Developer Documentation

Instruction Set Assembly Guide for Armv7 and earlier Arm architectures Reference Guide Version 2.0

- Preface
- ⊞ Instruction Set Overview
- ⊞ Advanced SIMD and Floating-point Programming
- ⊞ A32/T32 Instruction Set Reference

Instruction Set Assembly Guide for Armv7 and earlier Arm architectures Reference Guide Version 2.0

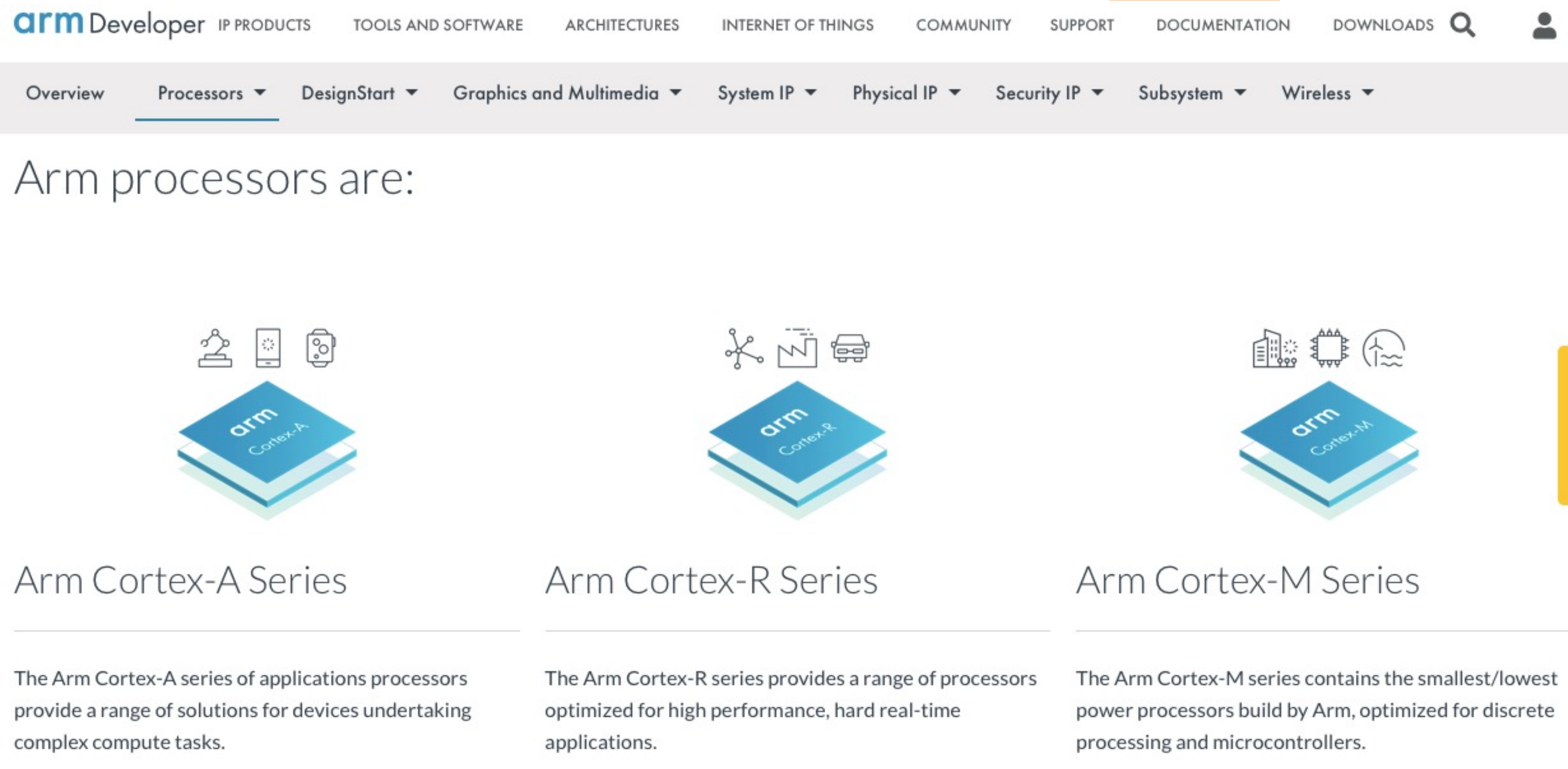
CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE

COMP122

ARM Assembly

DR JEFF SOFTWARE
INDIE APP DEVELOPER
© Jeff Drobman
2016-2022

ARM Ref



Arm processors are:

Arm Cortex-A Series

The Arm Cortex-A series of applications processors provide a range of solutions for devices undertaking complex compute tasks.

Arm Cortex-R Series

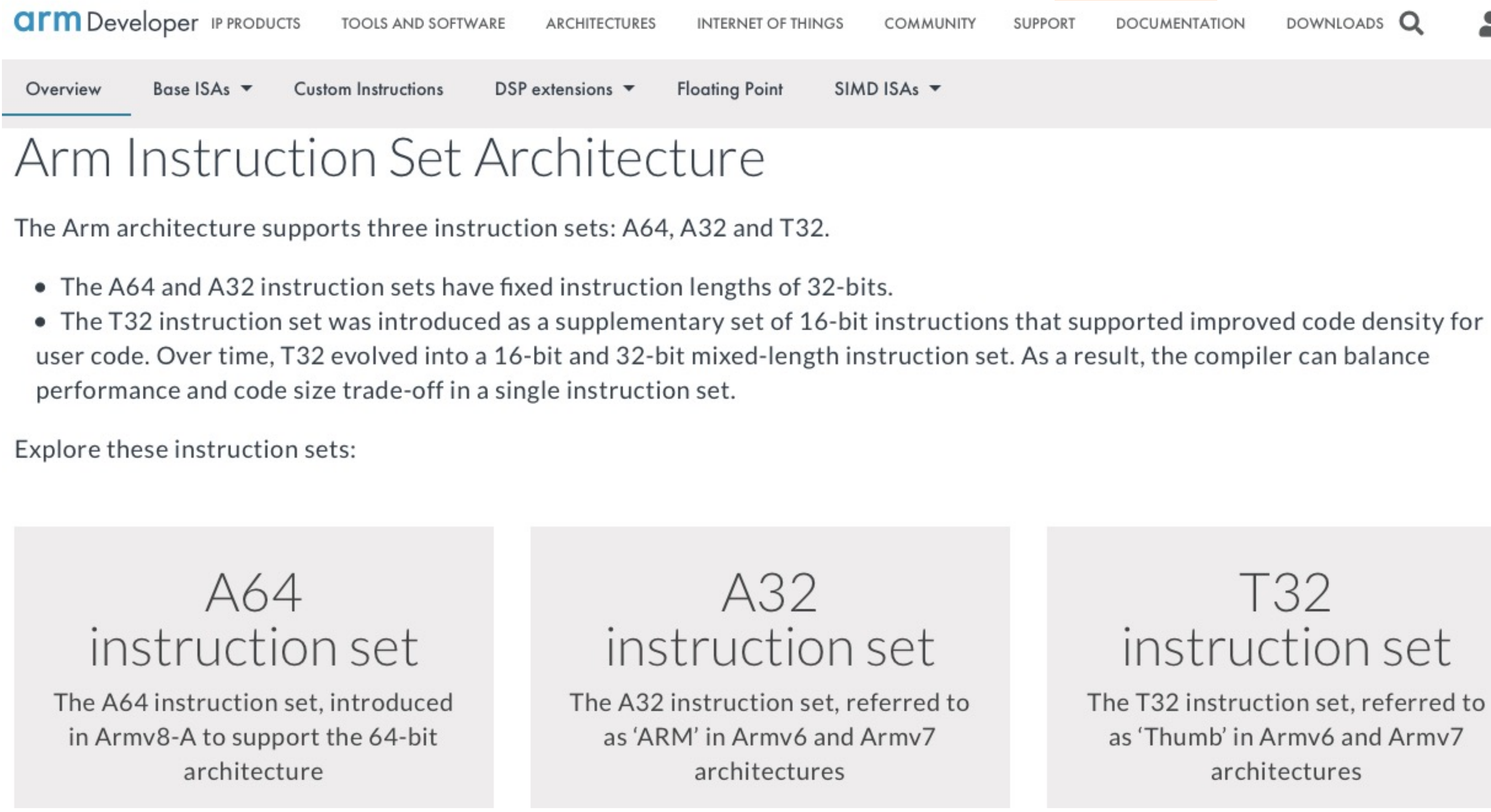
The Arm Cortex-R series provides a range of processors optimized for high performance, hard real-time applications.

Arm Cortex-M Series

The Arm Cortex-M series contains the smallest/lowest power processors build by Arm, optimized for discrete processing and microcontrollers.

ARM Assembly

ARM Ref



Arm Instruction Set Architecture

The Arm architecture supports three instruction sets: A64, A32 and T32.

- The A64 and A32 instruction sets have fixed instruction lengths of 32-bits.
- The T32 instruction set was introduced as a supplementary set of 16-bit instructions that supported improved code density for user code. Over time, T32 evolved into a 16-bit and 32-bit mixed-length instruction set. As a result, the compiler can balance performance and code size trade-off in a single instruction set.

Explore these instruction sets:

A64 instruction set

The A64 instruction set, introduced in Armv8-A to support the 64-bit architecture

A32 instruction set

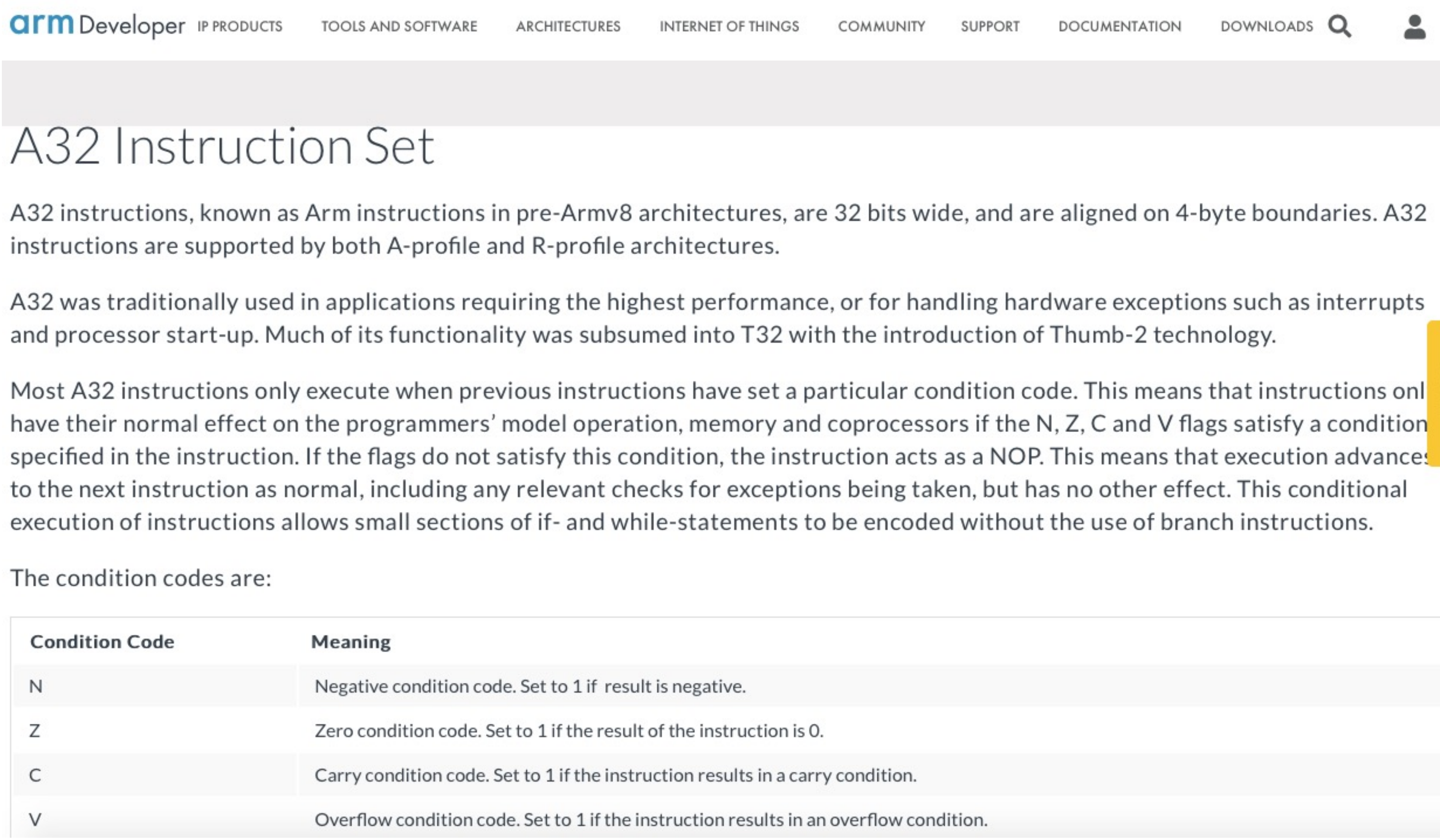
The A32 instruction set, referred to as 'ARM' in Armv6 and Armv7 architectures

T32 instruction set

The T32 instruction set, referred to as 'Thumb' in Armv6 and Armv7 architectures

ARM Assembly

ARM Ref



A32 Instruction Set

A32 instructions, known as Arm instructions in pre-Armv8 architectures, are 32 bits wide, and are aligned on 4-byte boundaries. A32 instructions are supported by both A-profile and R-profile architectures.

A32 was traditionally used in applications requiring the highest performance, or for handling hardware exceptions such as interrupts and processor start-up. Much of its functionality was subsumed into T32 with the introduction of Thumb-2 technology.

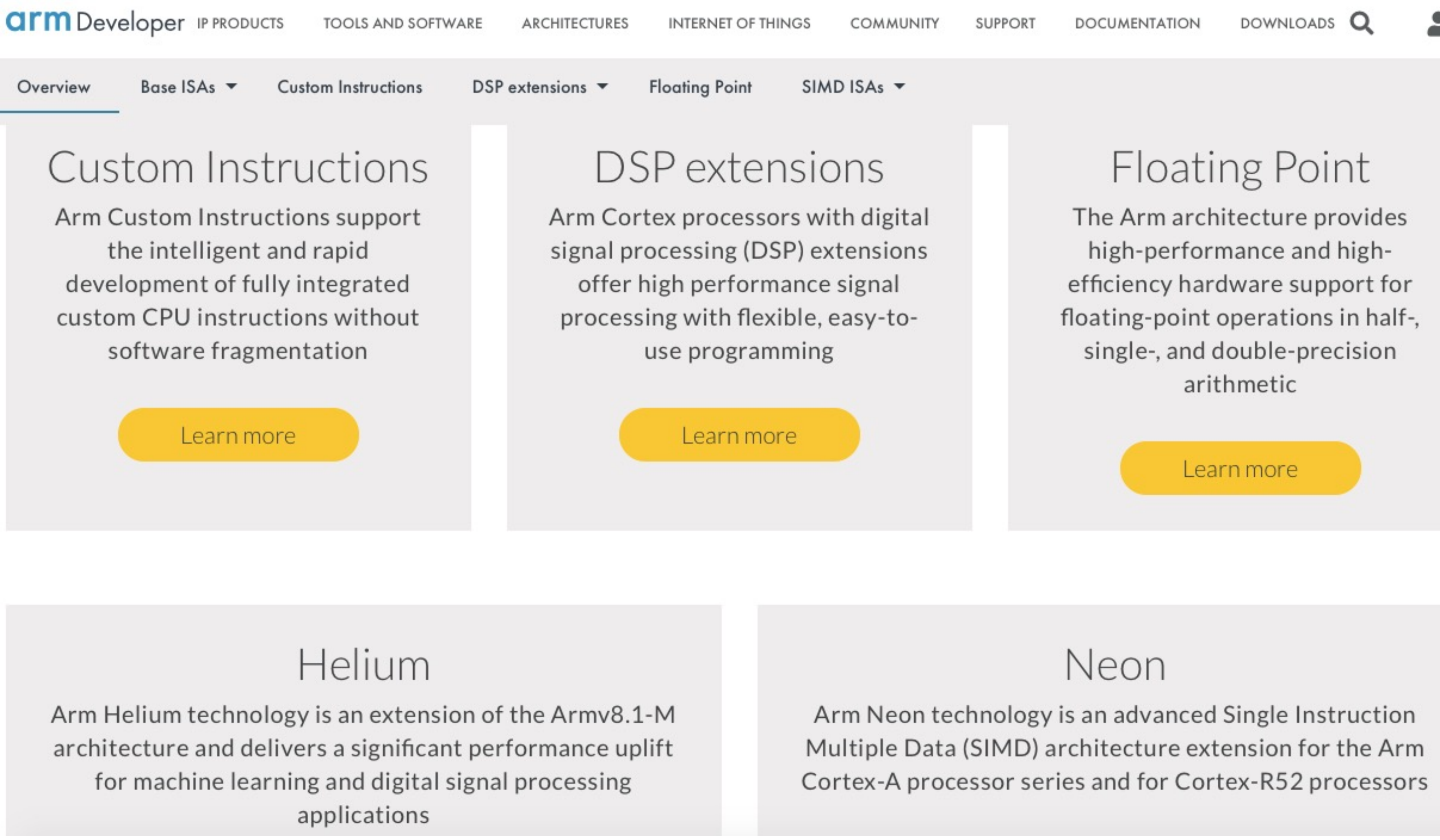
Most A32 instructions only execute when previous instructions have set a particular condition code. This means that instructions onl have their normal effect on the programmers' model operation, memory and coprocessors if the N, Z, C and V flags satisfy a condition specified in the instruction. If the flags do not satisfy this condition, the instruction acts as a NOP. This means that execution advance to the next instruction as normal, including any relevant checks for exceptions being taken, but has no other effect. This conditional execution of instructions allows small sections of if- and while-statements to be encoded without the use of branch instructions.

The condition codes are:

Condition Code	Meaning
N	Negative condition code. Set to 1 if result is negative.
Z	Zero condition code. Set to 1 if the result of the instruction is 0.
C	Carry condition code. Set to 1 if the instruction results in a carry condition.
V	Overflow condition code. Set to 1 if the instruction results in an overflow condition.

ARM Assembly

ARM Ref



Custom Instructions

Arm Custom Instructions support the intelligent and rapid development of fully integrated custom CPU instructions without software fragmentation

DSP extensions

Arm Cortex processors with digital signal processing (DSP) extensions offer high performance signal processing with flexible, easy-to-use programming

Floating Point

The Arm architecture provides high-performance and high-efficiency hardware support for floating-point operations in half-, single-, and double-precision arithmetic

Helium

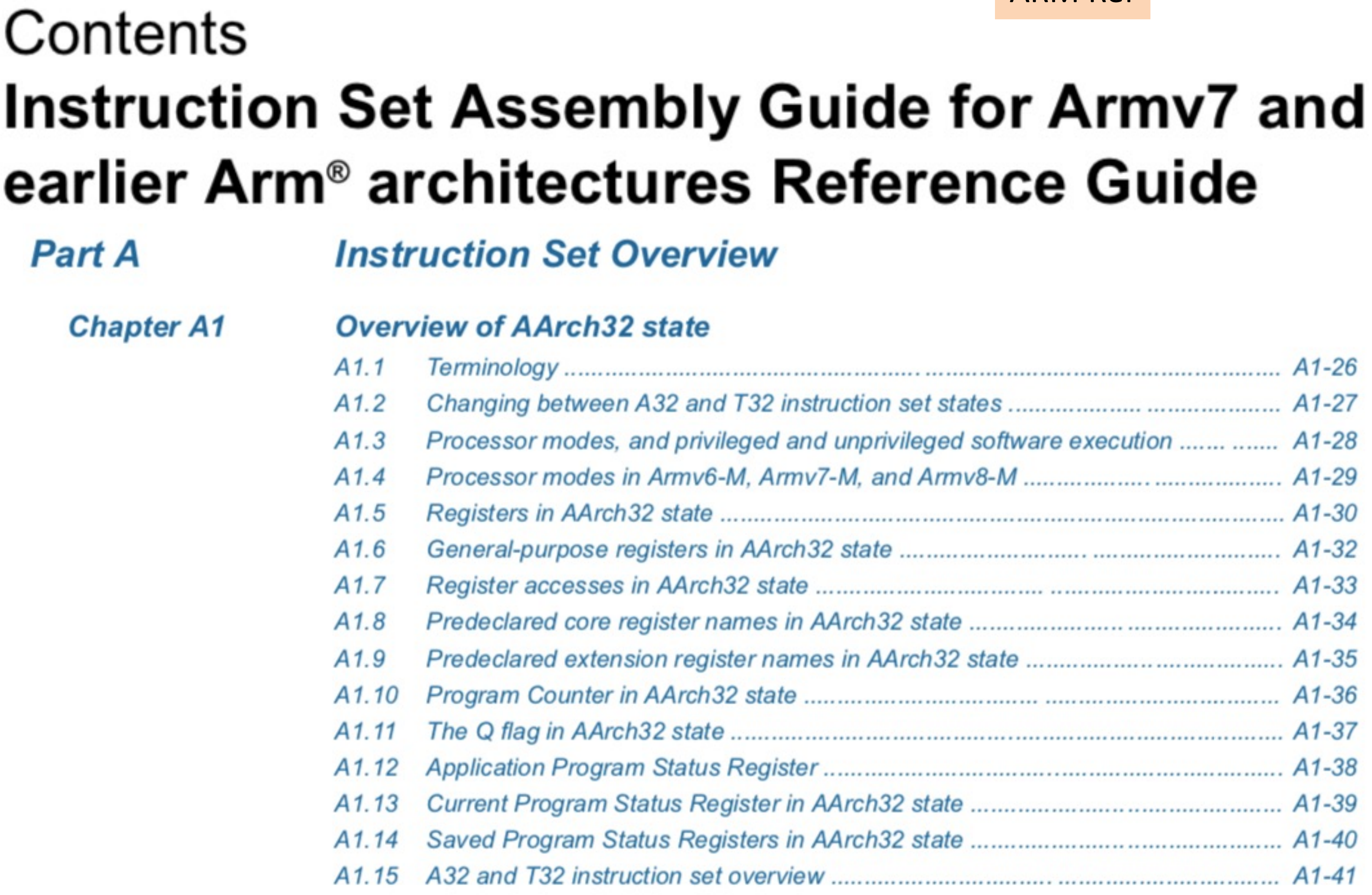
Arm Helium technology is an extension of the Armv8.1-M architecture and delivers a significant performance uplift for machine learning and digital signal processing applications

Neon

Arm Neon technology is an advanced Single Instruction Multiple Data (SIMD) architecture extension for the Arm Cortex-A processor series and for Cortex-R52 processors

ARM Assembly

ARM Ref



Contents

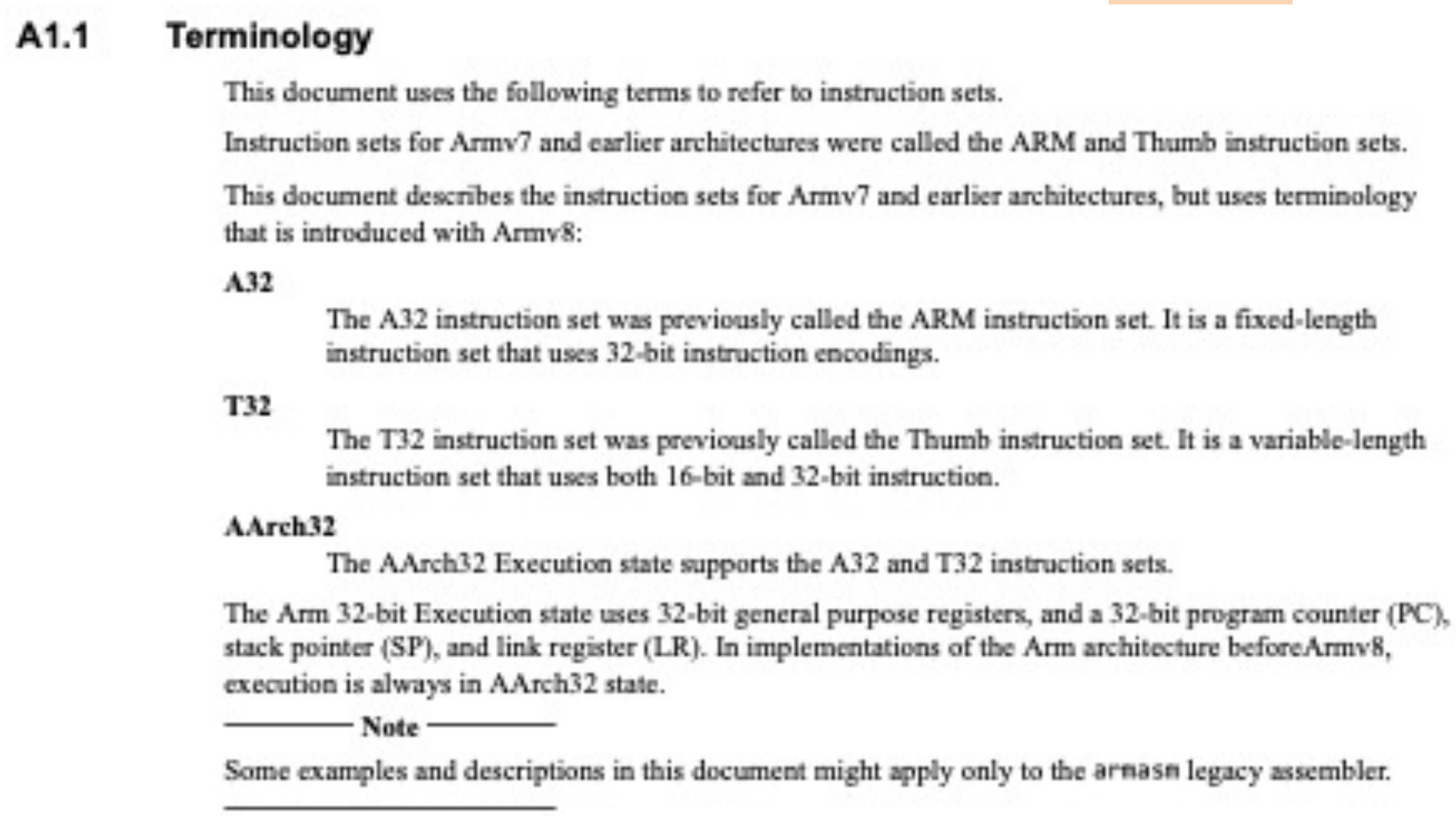
Instruction Set Assembly Guide for Armv7 and earlier Arm® architectures Reference Guide

Part A Instruction Set Overview

Chapter A1 Overview of AArch32 state

ARM Assembly

ARM Ref



A1.1 Terminology

This document uses the following terms to refer to instruction sets.

Instruction sets for Armv7 and earlier architectures were called the ARM and Thumb instruction sets.

This document describes the instruction sets for Armv7 and earlier architectures, but uses terminology that is introduced with Armv8:

A32

The A32 instruction set was previously called the ARM instruction set. It is a fixed-length instruction set that uses 32-bit instruction encodings.

T32

The T32 instruction set was previously called the Thumb instruction set. It is a variable-length instruction set that uses both 16-bit and 32-bit instruction.

AArch32

The AArch32 Execution state supports the A32 and T32 instruction sets.

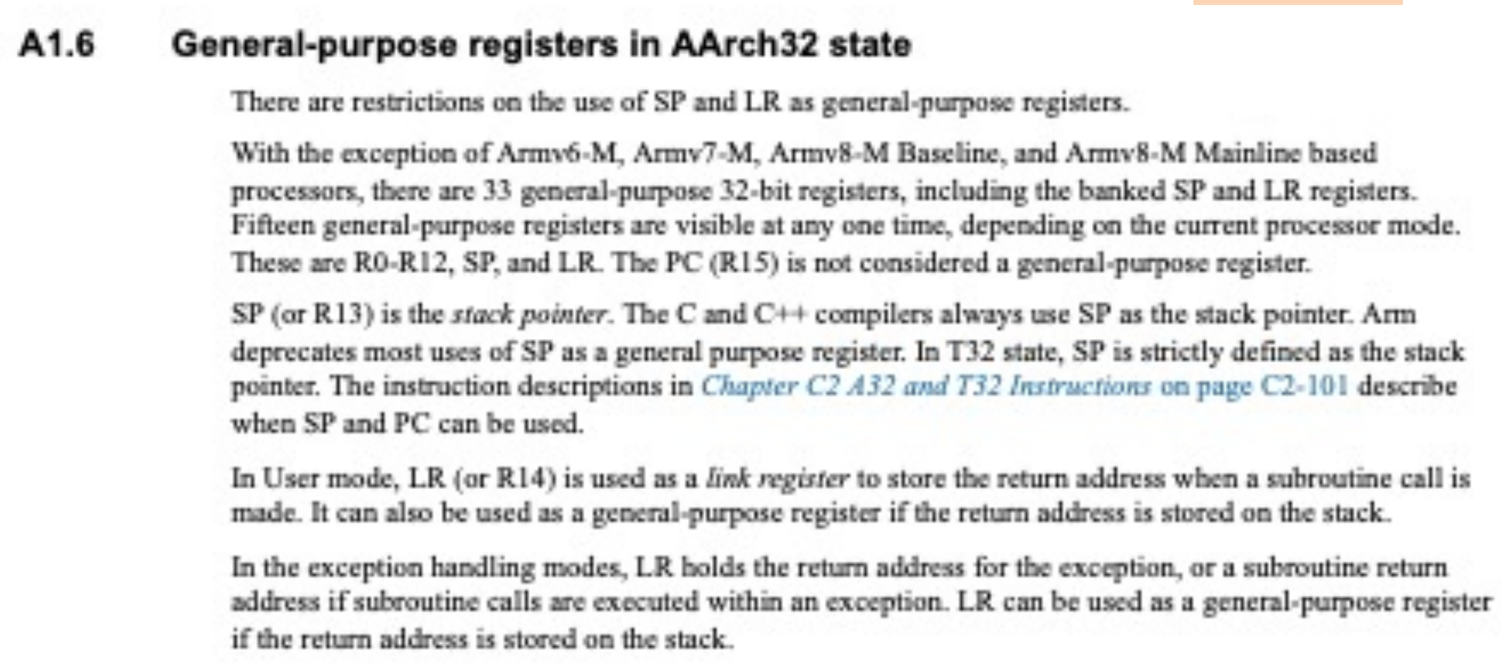
The Arm 32-bit Execution state uses 32-bit general purpose registers, and a 32-bit program counter (PC), stack pointer (SP), and link register (LR). In implementations of the Arm architecture beforeArmv8, execution is always in AArch32 state.

Note

Some examples and descriptions in this document might apply only to the `armasm` legacy assembler.

ARM Assembly

ARM Ref



A1.6 General-purpose registers in AArch32 state

There are restrictions on the use of SP and LR as general-purpose registers.

With the exception of Armv6-M, Armv7-M, Armv8-M Baseline, and Armv8-M Mainline based processors, there are 33 general-purpose 32-bit registers, including the banked SP and LR registers. Fifteen general-purpose registers are visible at any one time, depending on the current processor mode. These are R0-R12, SP, and LR. The PC (R15) is not considered a general-purpose register.

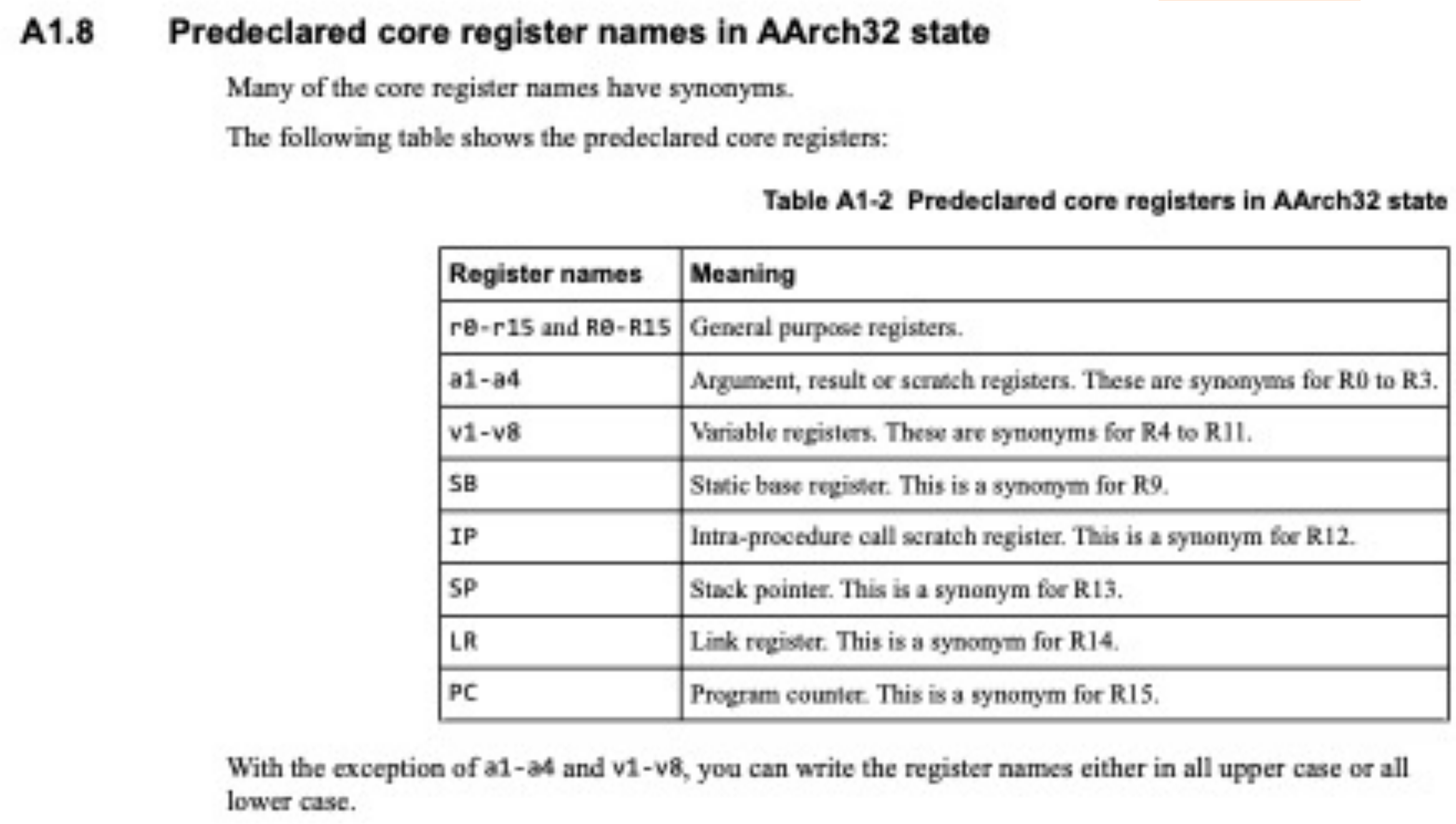
SP (or R13) is the *stack pointer*. The C and C++ compilers always use SP as the stack pointer. Arm deprecates most uses of SP as a general purpose register. In T32 state, SP is strictly defined as the stack pointer. The instruction descriptions in *Chapter C2 A32 and T32 Instructions* on page C2-101 describe when SP and PC can be used.

In User mode, LR (or R14) is used as a *link register* to store the return address when a subroutine call is made. It can also be used as a general-purpose register if the return address is stored on the stack.

In the exception handling modes, LR holds the return address for the exception, or a subroutine return address if subroutine calls are executed within an exception. LR can be used as a general-purpose register if the return address is stored on the stack.

ARM Assembly

ARM Ref



A1.8 Predeclared core register names in AArch32 state

Many of the core register names have synonyms.

The following table shows the predeclared core registers:

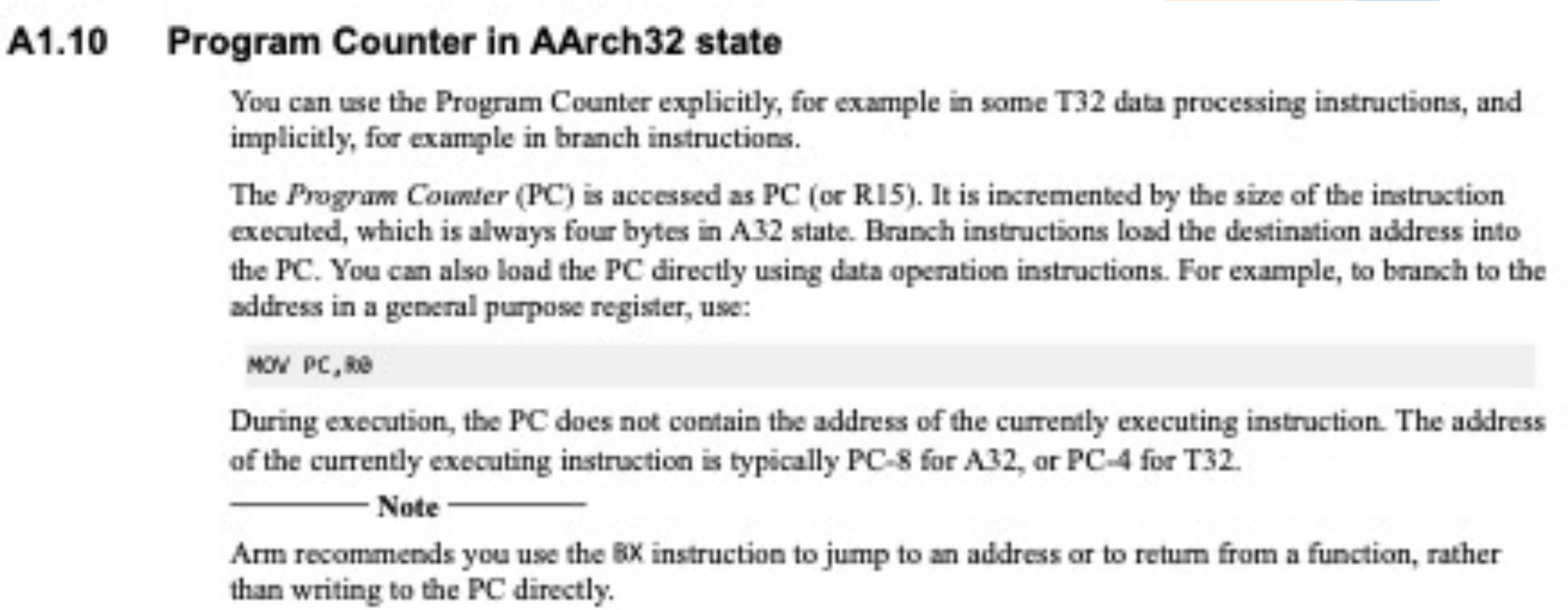
Table A1-2 Predeclared core registers in AArch32 state

Register names	Meaning
`r0-r15` and `R0-R15`	General purpose registers.
`a1-a4`	Argument, result or scratch registers. These are synonyms for R0 to R3.
`v1-v8`	Variable registers. These are synonyms for R4 to R11.
`SB`	Static base register. This is a synonym for R9.
`IP`	Intra-procedure call scratch register. This is a synonym for R12.
`SP`	Stack pointer. This is a synonym for R13.
`LR`	Link register. This is a synonym for R14.
`PC`	Program counter. This is a synonym for R15.

With the exception of `a1-a4` and `v1-v8`, you can write the register names either in all upper case or all lower case.

ARM Assembly

ARM Ref | **PC**



A1.10 Program Counter in AArch32 state

You can use the Program Counter explicitly, for example in some T32 data processing instructions, and implicitly, for example in branch instructions.

The *Program Counter* (PC) is accessed as PC (or R15). It is incremented by the size of the instruction executed, which is always four bytes in A32 state. Branch instructions load the destination address into the PC. You can also load the PC directly using data operation instructions. For example, to branch to the address in a general purpose register, use:

```
MOV PC,R0
```

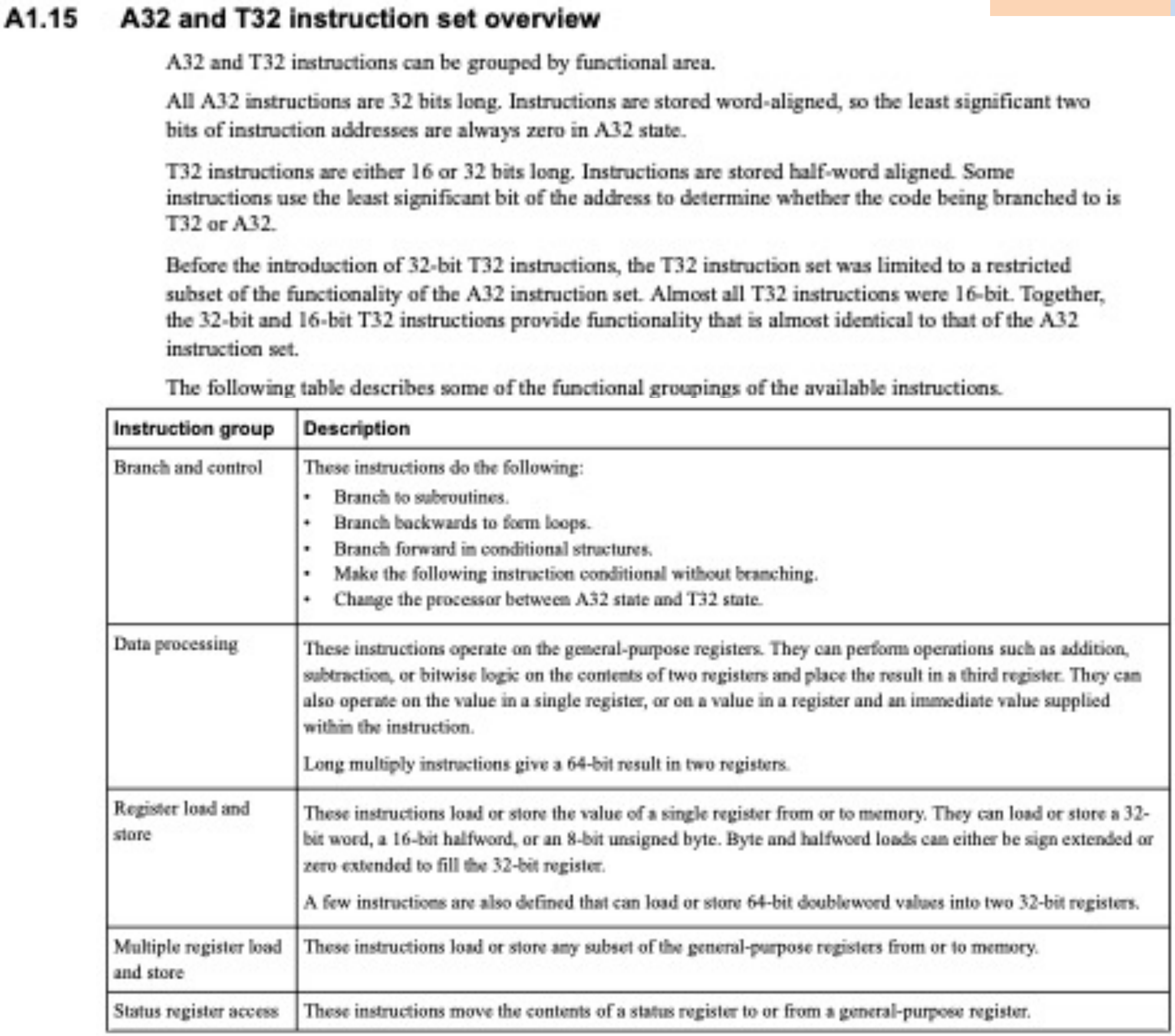
During execution, the PC does not contain the address of the currently executing instruction. The address of the currently executing instruction is typically PC-8 for A32, or PC-4 for T32.

Note

Arm recommends you use the BX instruction to jump to an address or to return from a function, rather than writing to the PC directly.

ARM Assembly

ARM Ref | **PSW**



A1.15 A32 and T32 instruction set overview

A32 and T32 instructions can be grouped by functional area.

All A32 instructions are 32 bits long. Instructions are stored word-aligned, so the least significant two bits of instruction addresses are always zero in A32 state.

T32 instructions are either 16 or 32 bits long. Instructions are stored half-word aligned. Some instructions use the least significant bit of the address to determine whether the code being branched to is T32 or A32.

Before the introduction of 32-bit T32 instructions, the T32 instruction set was limited to a restricted subset of the functionality of the A32 instruction set. Almost all T32 instructions were 16-bit. Together, the 32-bit and 16-bit T32 instructions provide functionality that is almost identical to that of the A32 instruction set.

The following table describes some of the functional groupings of the available instructions.

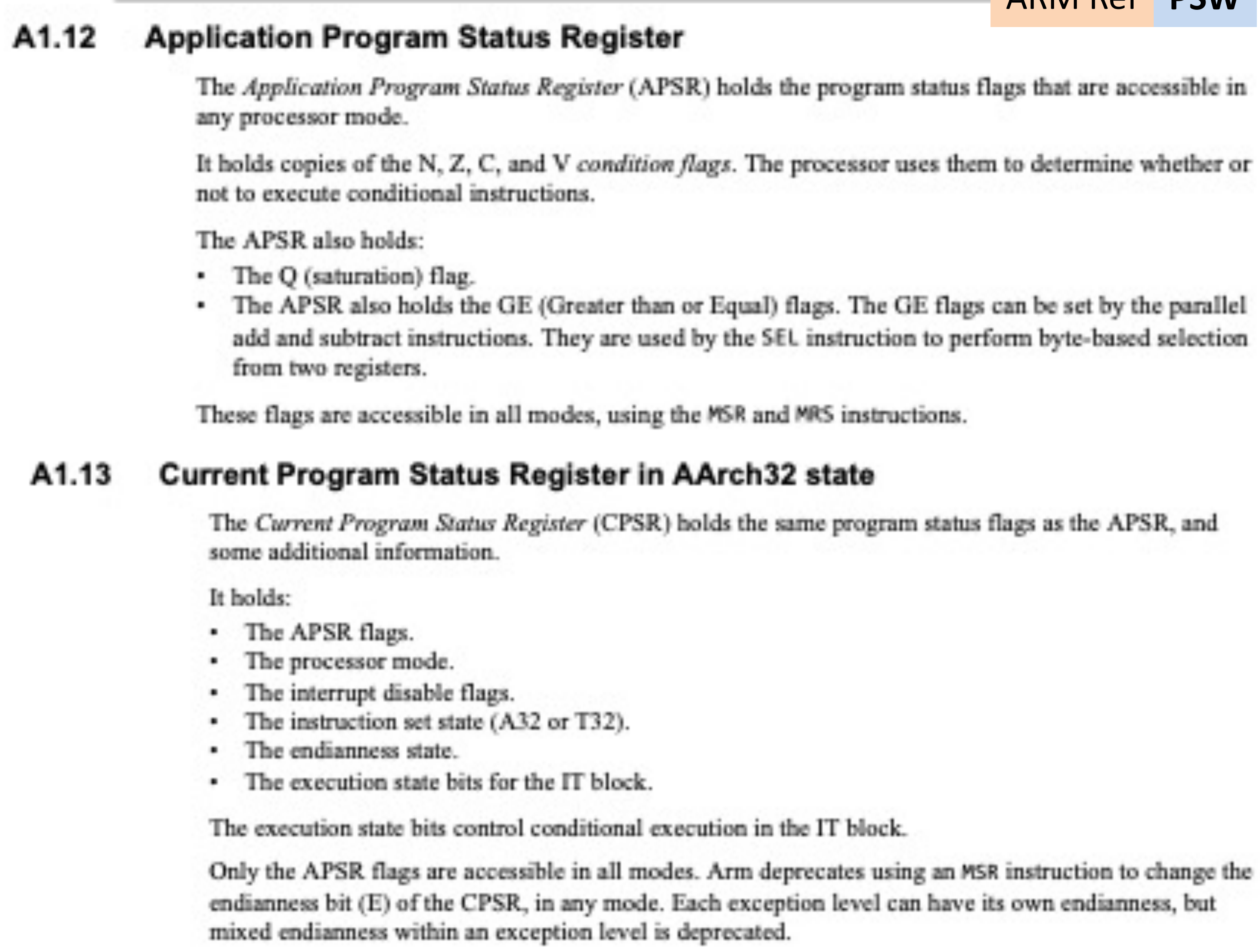
Instruction group	Description
Branch and control	These instructions do the following: • Branch to subroutines. • Branch backwards to form loops. • Branch forward in conditional structures. • Make the following instruction conditional without branching. • Change the processor between A32 state and T32 state.
Data processing	These instructions operate on the general-purpose registers. They can perform operations such as addition, subtraction, or bitwise logic on the contents of two registers and place the result in a third register. They can also operate on the value in a single register, or on a value in a register and an immediate value supplied within the instruction. Long multiply instructions give a 64-bit result in two registers.
Register load and store	These instructions load or store the value of a single register from or to memory. They can load or store a 32-bit word, a 16-bit halfword, or an 8-bit unsigned byte. Byte and halfword loads can either be sign extended or zero extended to fill the 32-bit register. A few instructions are also defined that can load or store 64-bit doubleword values into two 32-bit registers.
Multiple register load and store	These instructions load or store any subset of the general-purpose registers from or to memory.
Status register access	These instructions move the contents of a status register to or from a general-purpose register.

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP122

ARM Assembly

DR JEFF SOFTWARE
INDIE APP DEVELOPER
© Jeff Drobman
2016-2022

ARM Ref | **PSW**



A1.12 Application Program Status Register

The *Application Program Status Register* (APSR) holds the program status flags that are accessible in any processor mode.

It holds copies of the N, Z, C, and V *condition flags*. The processor uses them to determine whether or not to execute conditional instructions.

The APSR also holds:

- The Q (saturation) flag.
- The APSR also holds the GE (Greater than or Equal) flags. The GE flags can be set by the parallel add and subtract instructions. They are used by the SEL instruction to perform byte-based selection from two registers.

These flags are accessible in all modes, using the MSR and MRS instructions.

A1.13 Current Program Status Register in AArch32 state

The *Current Program Status Register* (CPSR) holds the same program status flags as the APSR, and some additional information.

It holds:

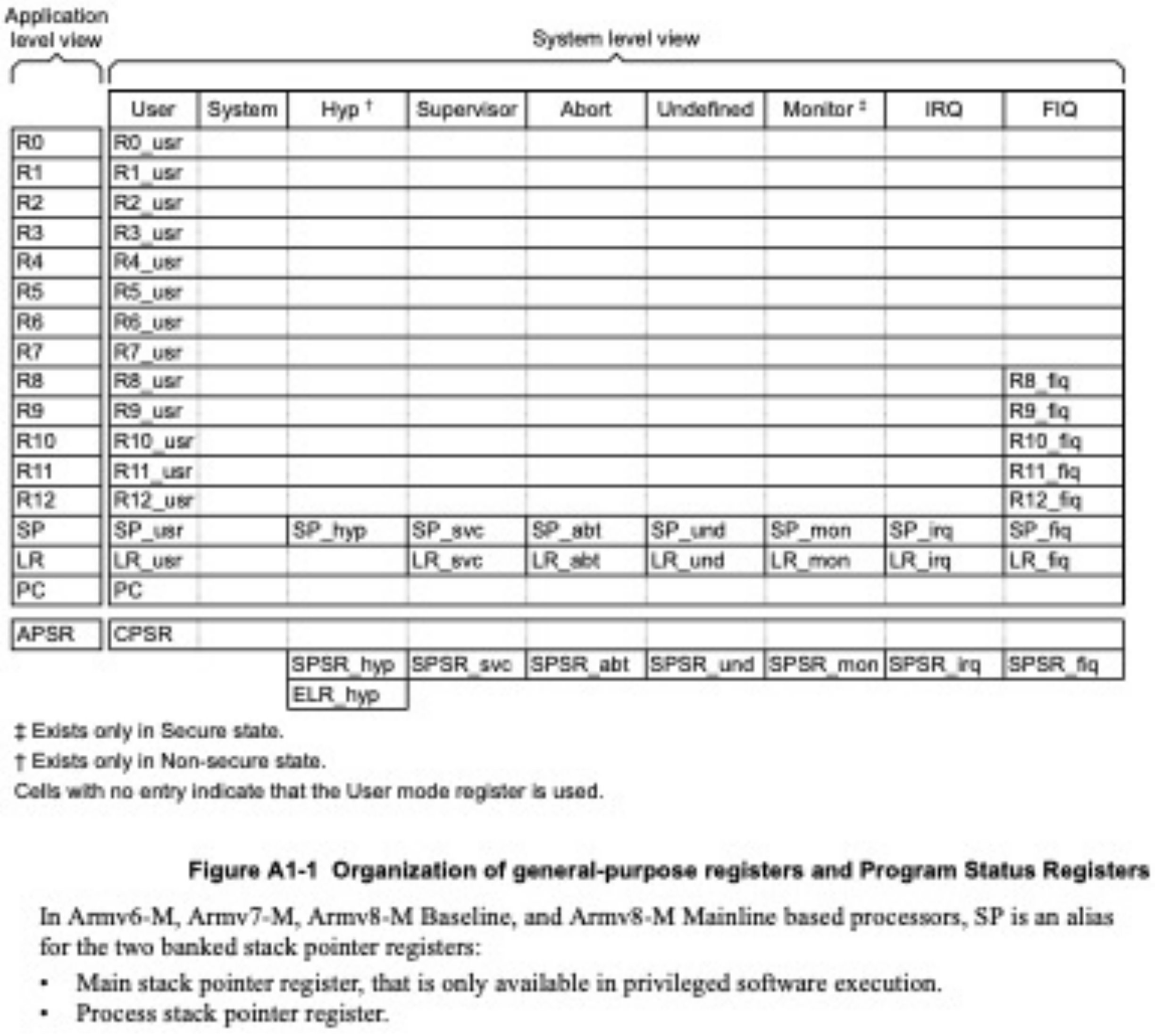
- The APSR flags.
- The processor mode.
- The interrupt disable flags.
- The instruction set state (A32 or T32).
- The endianness state.
- The execution state bits for the IT block.

The execution state bits control conditional execution in the IT block.

Only the APSR flags are accessible in all modes. Arm deprecates using an MSR instruction to change the endianness bit (E) of the CPSR, in any mode. Each exception level can have its own endianness, but mixed endianness within an exception level is deprecated.

ARM Assembly

ARM Ref



Application level view	System level view								
	User	System	Hyp †	Supervisor	Abort	Undefined	Monitor ‡	IRQ	FIQ
R0	R0_usr								
R1	R1_usr								
R2	R2_usr								
R3	R3_usr								
R4	R4_usr								
R5	R5_usr								
R6	R6_usr								
R7	R7_usr								
R8	R8_usr								R8_fiq
R9	R9_usr								R9_fiq
R10	R10_usr								R10_fiq
R11	R11_usr								R11_fiq
R12	R12_usr								R12_fiq
SP	SP_usr		SP_hyp	SP_svc	SP_abt	SP_und	SP_mon	SP_irq	SP_fiq
LR	LR_usr			LR_svc	LR_abt	LR_und	LR_mon	LR_irq	LR_fiq
PC	PC								
APSR	CPSR								
			SPSR_hyp	SPSR_svc	SPSR_abt	SPSR_und	SPSR_mon	SPSR_irq	SPSR_fiq
			ELR_hyp						

‡ Exists only in Secure state.
† Exists only in Non-secure state.
Cells with no entry indicate that the User mode register is used.

Figure A1-1 Organization of general-purpose registers and Program Status Registers

In Armv6-M, Armv7-M, Armv8-M Baseline, and Armv8-M Mainline based processors, SP is an alias for the two banked stack pointer registers:

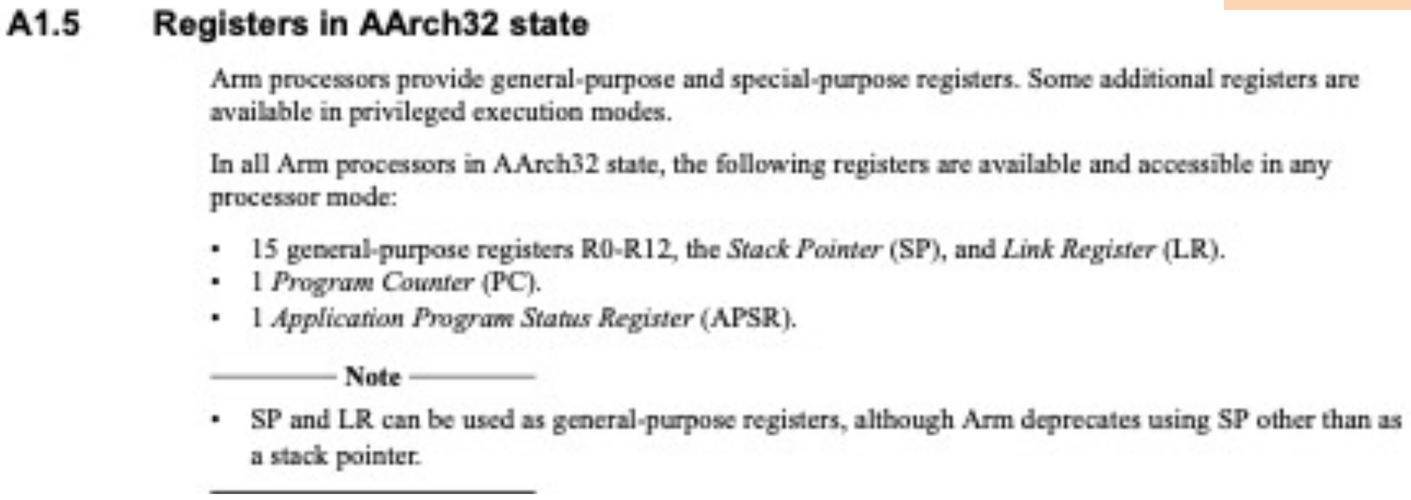
- Main stack pointer register, that is only available in privileged software execution.
- Process stack pointer register.

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP122

ARM Assembly

DR JEFF SOFTWARE
INDIE APP DEVELOPER
© Jeff Drobman
2016-2022

ARM Ref



A1.5 Registers in AArch32 state

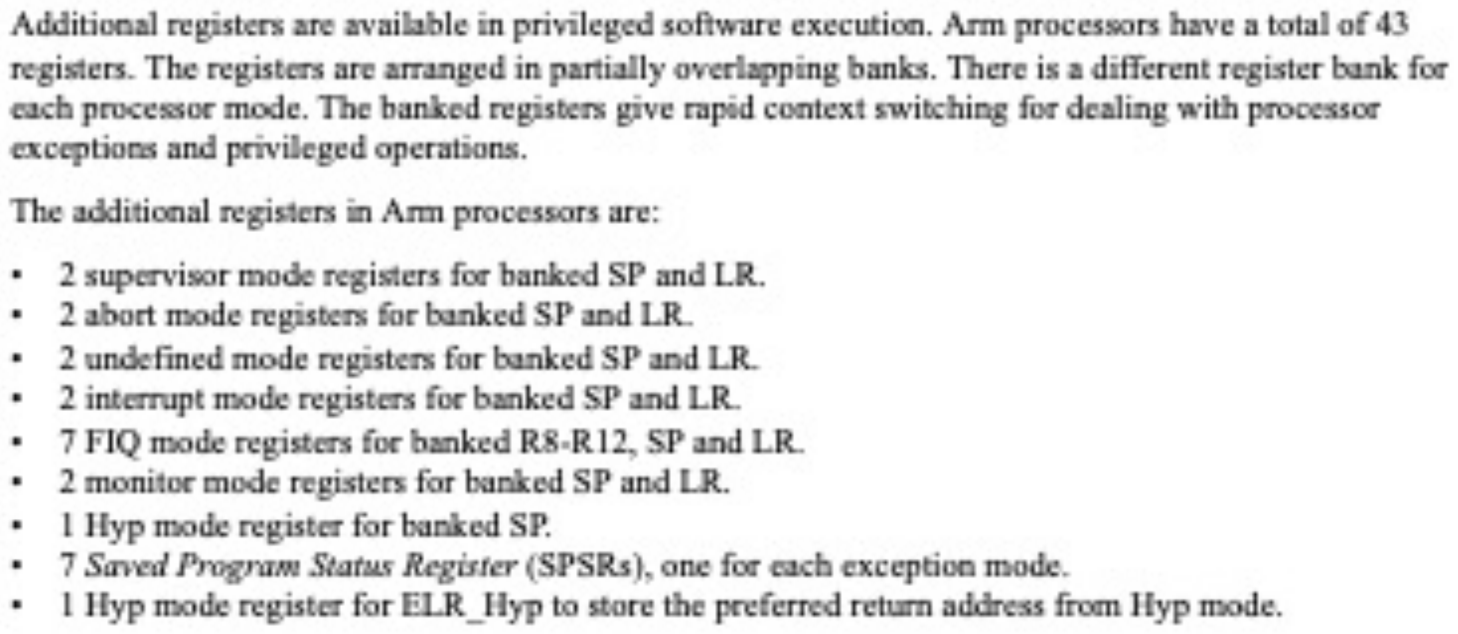
Arm processors provide general-purpose and special-purpose registers. Some additional registers are available in privileged execution modes.

In all Arm processors in AArch32 state, the following registers are available and accessible in any processor mode:

- 15 general-purpose registers R0-R12, the *Stack Pointer* (SP), and *Link Register* (LR).
- 1 *Program Counter* (PC).
- 1 *Application Program Status Register* (APSR).

Note

- SP and LR can be used as general-purpose registers, although Arm deprecates using SP other than as a stack pointer.



Additional registers are available in privileged software execution. Arm processors have a total of 43 registers. The registers are arranged in partially overlapping banks. There is a different register bank for each processor mode. The banked registers give rapid context switching for dealing with processor exceptions and privileged operations.

The additional registers in Arm processors are:

- 2 supervisor mode registers for banked SP and LR.
- 2 abort mode registers for banked SP and LR.
- 2 undefined mode registers for banked SP and LR.
- 2 interrupt mode registers for banked SP and LR.
- 7 FIQ mode registers for banked R8-R12, SP and LR.
- 2 monitor mode registers for banked SP and LR.
- 1 Hyp mode register for banked SP.
- 7 *Saved Program Status Register* (SPSRs), one for each exception mode.
- 1 Hyp mode register for ELR_Hyp to store the preferred return address from Hyp mode.

ARM Assembly

ARM Instruction Set

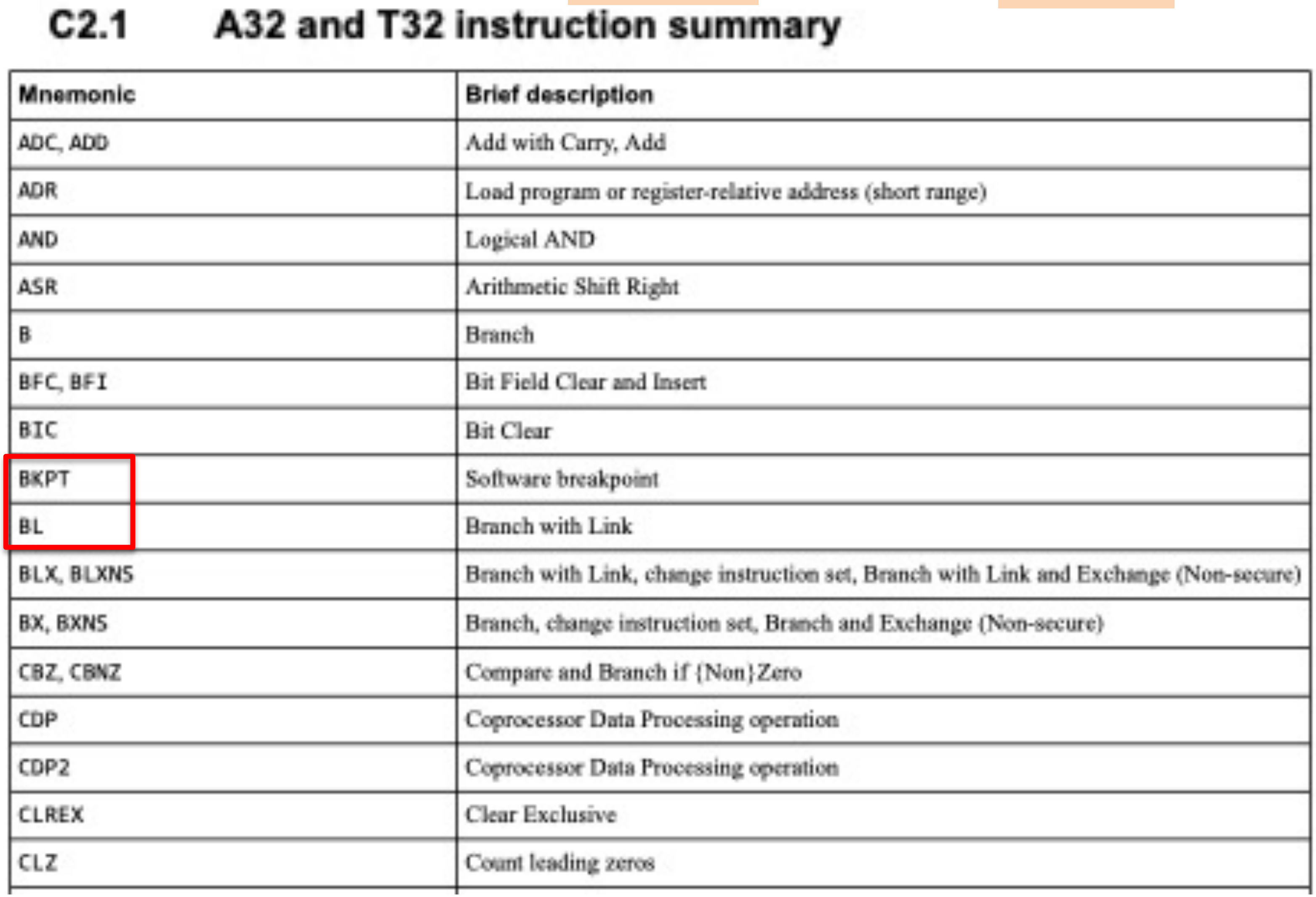
CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP122

ARM Instruction Set

DR JEFF SOFTWARE
INDIE APP DEVELOPER
© Jeff Drobman
2016-2022

Summary

ARM Ref

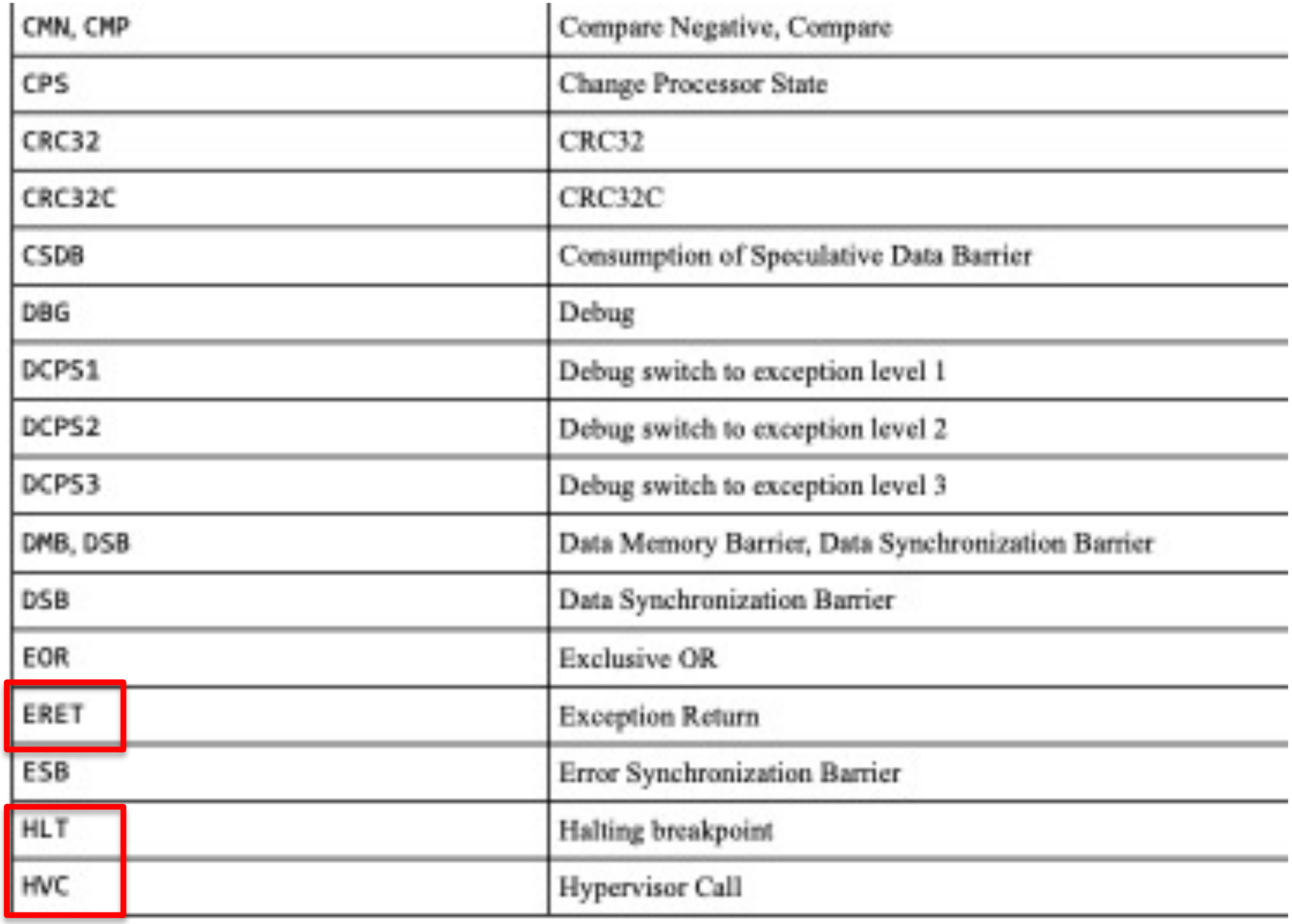


C2.1 A32 and T32 instruction summary

Mnemonic	Brief description
ADC, ADD	Add with Carry, Add
ADR	Load program or register-relative address (short range)
AND	Logical AND
ASR	Arithmetic Shift Right
B	Branch
BFC, BFI	Bit Field Clear and Insert
BIC	Bit Clear
BKPT	Software breakpoint
BL	Branch with Link
BLX, BLXNS	Branch with Link, change instruction set, Branch with Link and Exchange (Non-secure)
BX, BXNS	Branch, change instruction set, Branch and Exchange (Non-secure)
CBZ, CBNZ	Compare and Branch if {Non}Zero
CDP	Coprocessor Data Processing operation
CDP2	Coprocessor Data Processing operation
CLREX	Clear Exclusive
CLZ	Count leading zeros

ARM Instruction Set

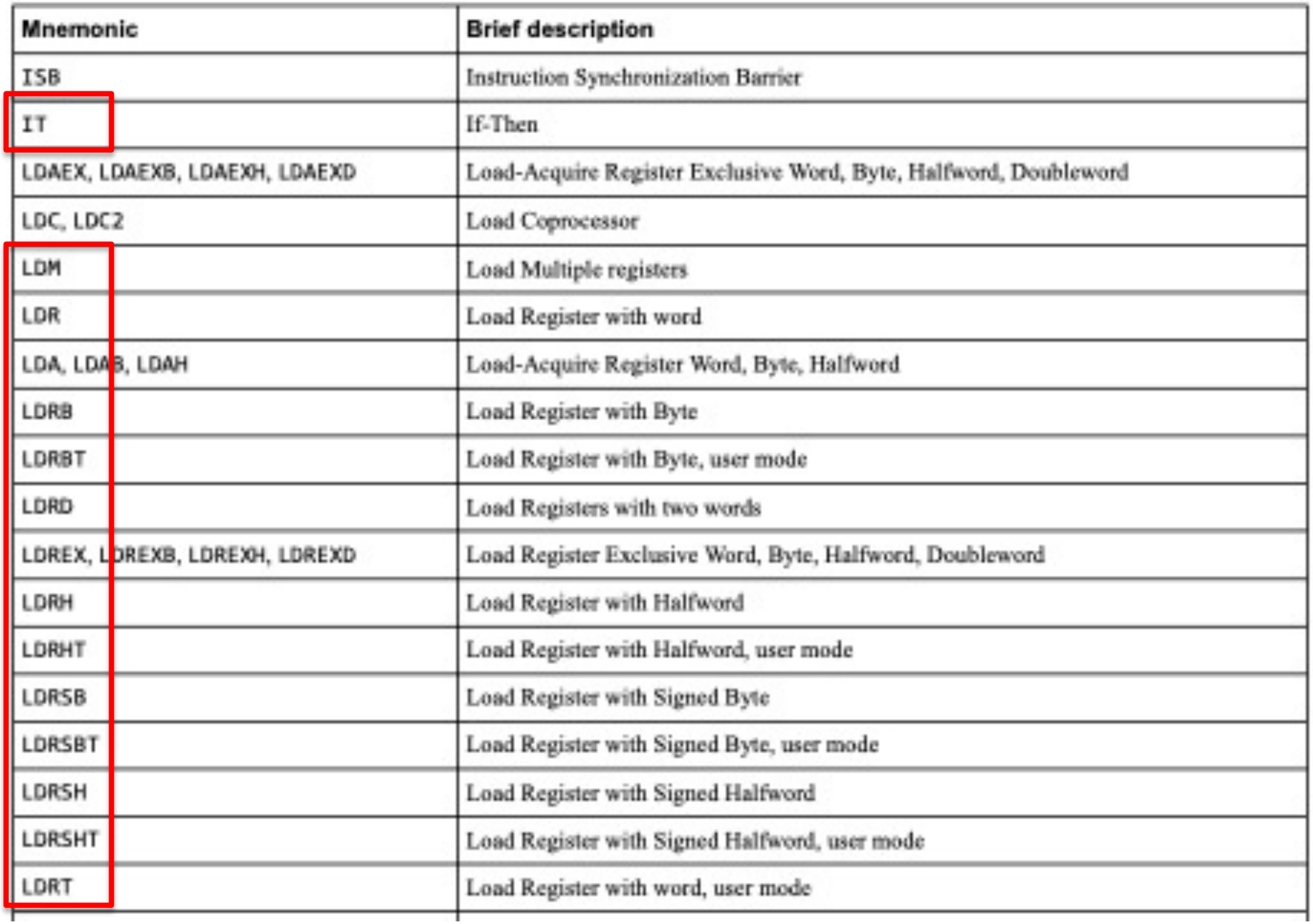
ARM Ref



CMN, CMP	Compare Negative, Compare
CPS	Change Processor State
CRC32	CRC32
CRC32C	CRC32C
CSDB	Consumption of Speculative Data Barrier
DBG	Debug
DCPS1	Debug switch to exception level 1
DCPS2	Debug switch to exception level 2
DCPS3	Debug switch to exception level 3
DMB, DSB	Data Memory Barrier, Data Synchronization Barrier
DSB	Data Synchronization Barrier
EOR	Exclusive OR
ERET	Exception Return
ESB	Error Synchronization Barrier
HLT	Halting breakpoint
HVC	Hypervisor Call

ARM Instruction Set

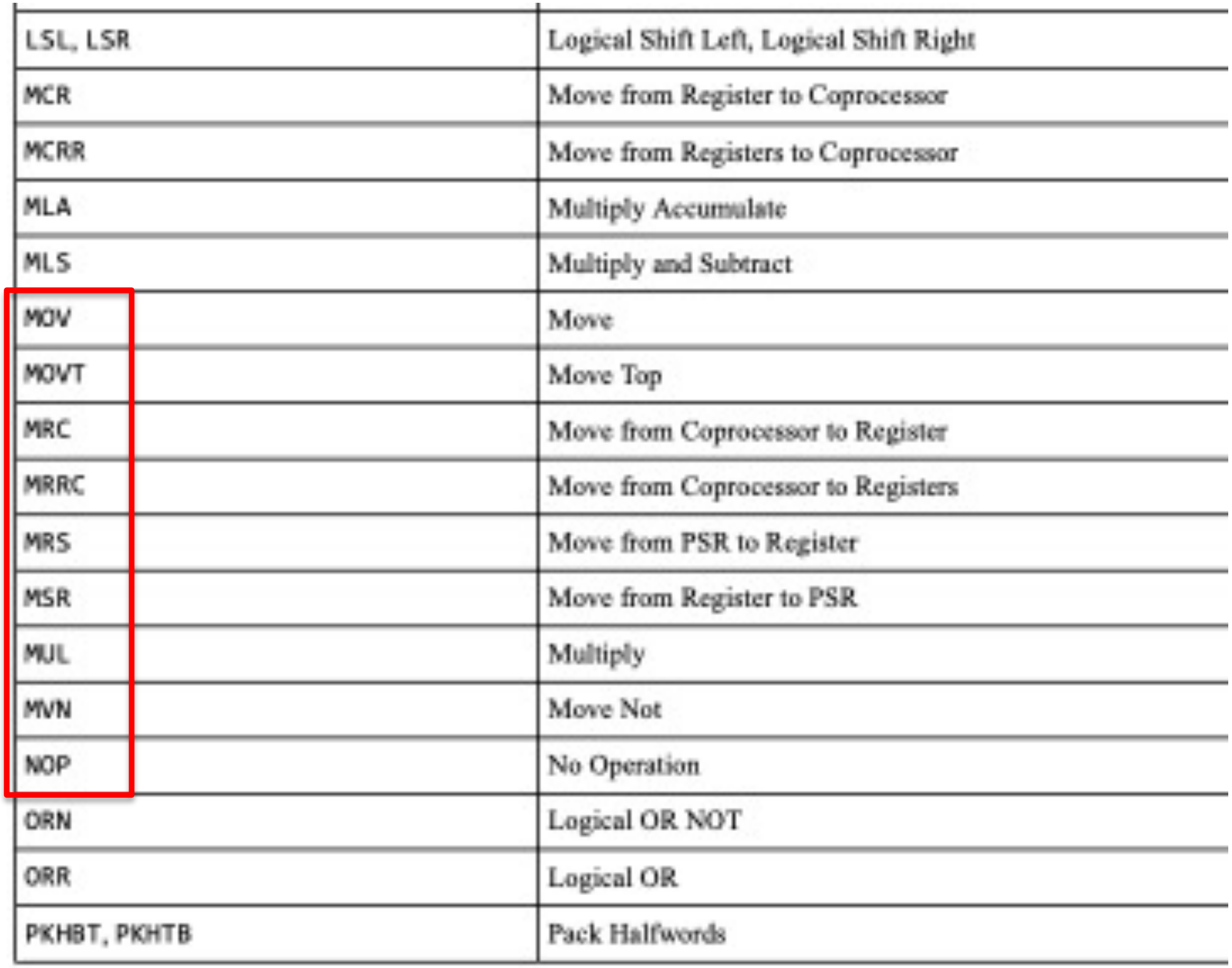
ARM Ref



Mnemonic	Brief description
ISB	Instruction Synchronization Barrier
IT	If-Then
LDAEX, LDAEXB, LDAEXH, LDAEXD	Load-Acquire Register Exclusive Word, Byte, Halfword, Doubleword
LDC, LDC2	Load Coprocessor
LDM	Load Multiple registers
LDR	Load Register with word
LDA, LDAB, LDAH	Load-Acquire Register Word, Byte, Halfword
LDRB	Load Register with Byte
LDRBT	Load Register with Byte, user mode
LDRD	Load Registers with two words
LDREX, LDREXB, LDREXH, LDREXD	Load Register Exclusive Word, Byte, Halfword, Doubleword
LDRH	Load Register with Halfword
LDRHT	Load Register with Halfword, user mode
LDRSB	Load Register with Signed Byte
LDRSBT	Load Register with Signed Byte, user mode
LDRSH	Load Register with Signed Halfword
LDRSHT	Load Register with Signed Halfword, user mode
LDRT	Load Register with word, user mode

ARM Instruction Set

ARM Ref



LSL, LSR	Logical Shift Left, Logical Shift Right
MCR	Move from Register to Coprocessor
MCRR	Move from Registers to Coprocessor
MLA	Multiply Accumulate
MLS	Multiply and Subtract
MOV	Move
MOVT	Move Top
MRC	Move from Coprocessor to Register
MRRC	Move from Coprocessor to Registers
MRS	Move from PSR to Register
MSR	Move from Register to PSR
MUL	Multiply
MVN	Move Not
NOP	No Operation
ORN	Logical OR NOT
ORR	Logical OR
PKHBT, PKHTB	Pack Halfwords

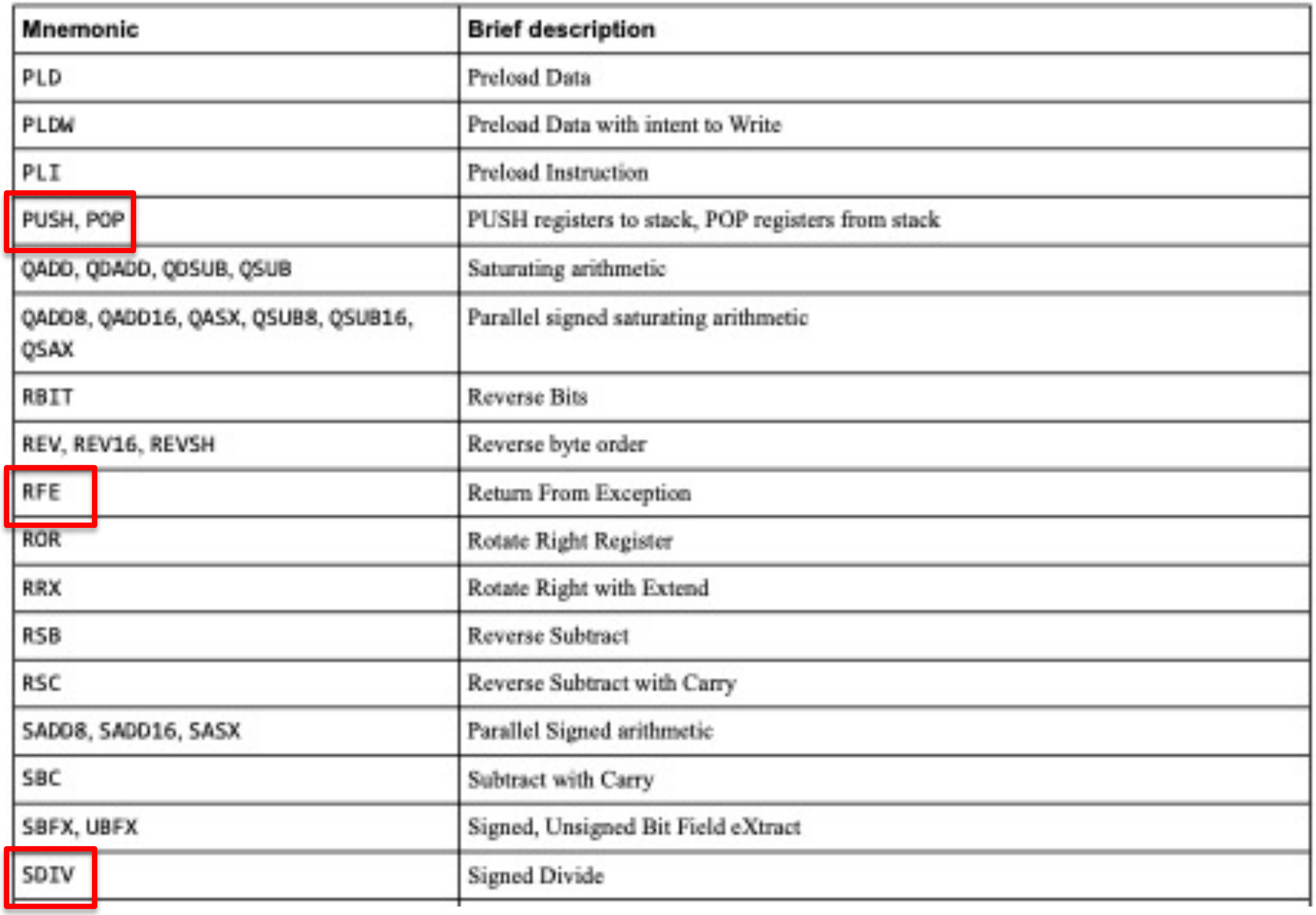
CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE

COMP122

ARM Instruction Set

DR JEFF SOFTWARE
INDIE APP DEVELOPER
© Jeff Drobman
2016-2022

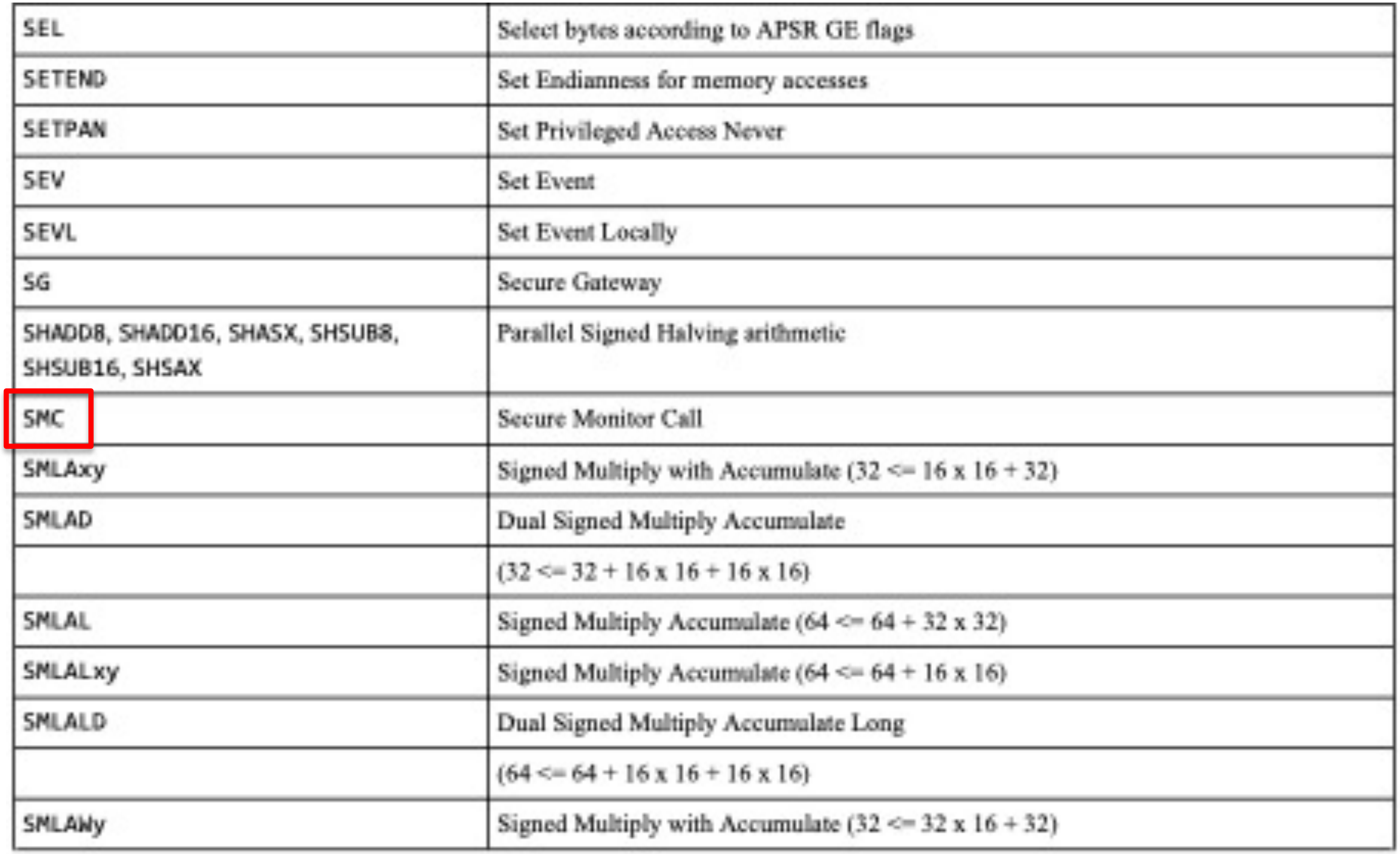
ARM Ref



Mnemonic	Brief description
PLD	Preload Data
PLDW	Preload Data with intent to Write
PLI	Preload Instruction
PUSH, POP	PUSH registers to stack, POP registers from stack
QADD, QDADD, QDSUB, QSUB	Saturating arithmetic
QADD8, QADD16, QASX, QSUB8, QSUB16, QSAX	Parallel signed saturating arithmetic
RBIT	Reverse Bits
REV, REV16, REVSH	Reverse byte order
RFE	Return From Exception
ROR	Rotate Right Register
RRX	Rotate Right with Extend
RSB	Reverse Subtract
RSC	Reverse Subtract with Carry
SADD8, SADD16, SASX	Parallel Signed arithmetic
SBC	Subtract with Carry
SBFX, UBFX	Signed, Unsigned Bit Field eXtract
SDIV	Signed Divide

ARM Instruction Set

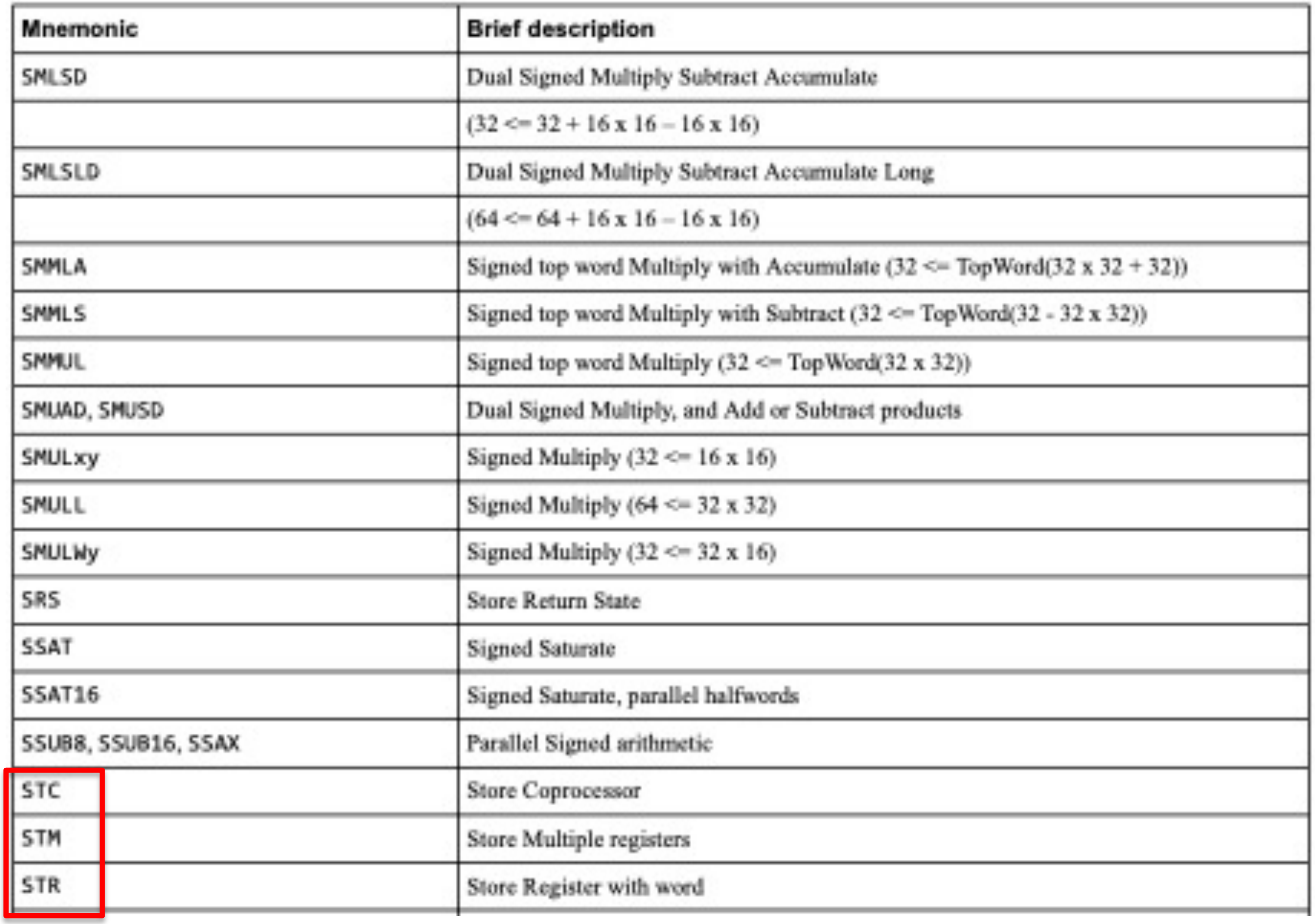
ARM Ref



SEL	Select bytes according to APSR GE flags
SETEND	Set Endianness for memory accesses
SETPAN	Set Privileged Access Never
SEV	Set Event
SEVL	Set Event Locally
SG	Secure Gateway
SHADD8, SHADD16, SHASX, SHSUB8, SHSUB16, SHSAX	Parallel Signed Halving arithmetic
SMC	Secure Monitor Call
SMLAxy	Signed Multiply with Accumulate (32 <= 16 x 16 + 32)
SMLAD	Dual Signed Multiply Accumulate
	(32 <= 32 + 16 x 16 + 16 x 16)
SMLAL	Signed Multiply Accumulate (64 <= 64 + 32 x 32)
SMLALxy	Signed Multiply Accumulate (64 <= 64 + 16 x 16)
SMLALD	Dual Signed Multiply Accumulate Long
	(64 <= 64 + 16 x 16 + 16 x 16)
SMLAWy	Signed Multiply with Accumulate (32 <= 32 x 16 + 32)

ARM Instruction Set

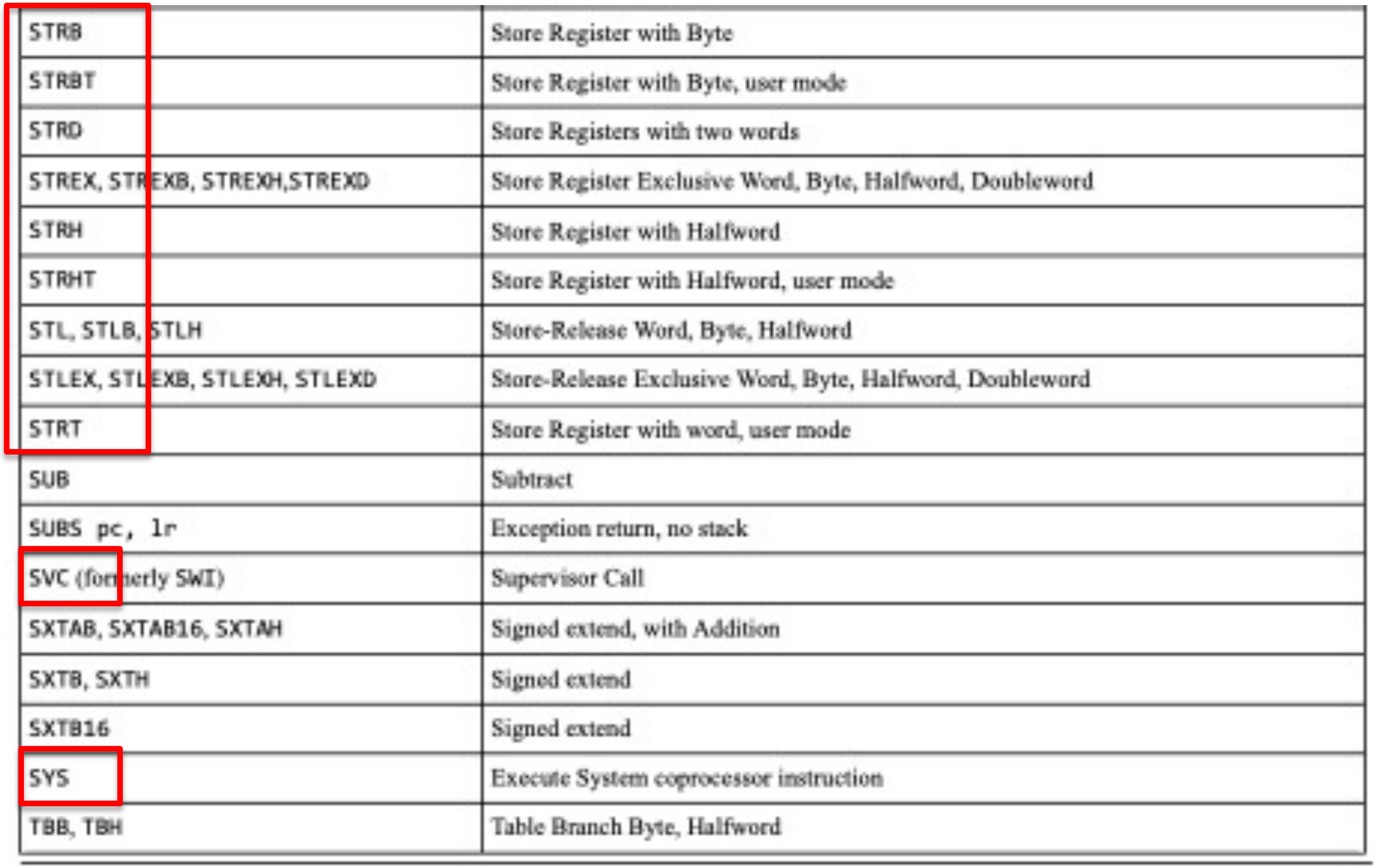
ARM Ref



Mnemonic	Brief description
SMLSD	Dual Signed Multiply Subtract Accumulate
	(32 <= 32 + 16 x 16 – 16 x 16)
SMLSLD	Dual Signed Multiply Subtract Accumulate Long
	(64 <= 64 + 16 x 16 – 16 x 16)
SMMLA	Signed top word Multiply with Accumulate (32 <= TopWord(32 x 32 + 32))
SMMLS	Signed top word Multiply with Subtract (32 <= TopWord(32 - 32 x 32))
SMMUL	Signed top word Multiply (32 <= TopWord(32 x 32))
SMUAD, SMUSD	Dual Signed Multiply, and Add or Subtract products
SMULxy	Signed Multiply (32 <= 16 x 16)
SMULL	Signed Multiply (64 <= 32 x 32)
SMULWy	Signed Multiply (32 <= 32 x 16)
SRS	Store Return State
SSAT	Signed Saturate
SSAT16	Signed Saturate, parallel halfwords
SSUB8, SSUB16, SSAX	Parallel Signed arithmetic
STC	Store Coprocessor
STM	Store Multiple registers
STR	Store Register with word

ARM Instruction Set

ARM Ref



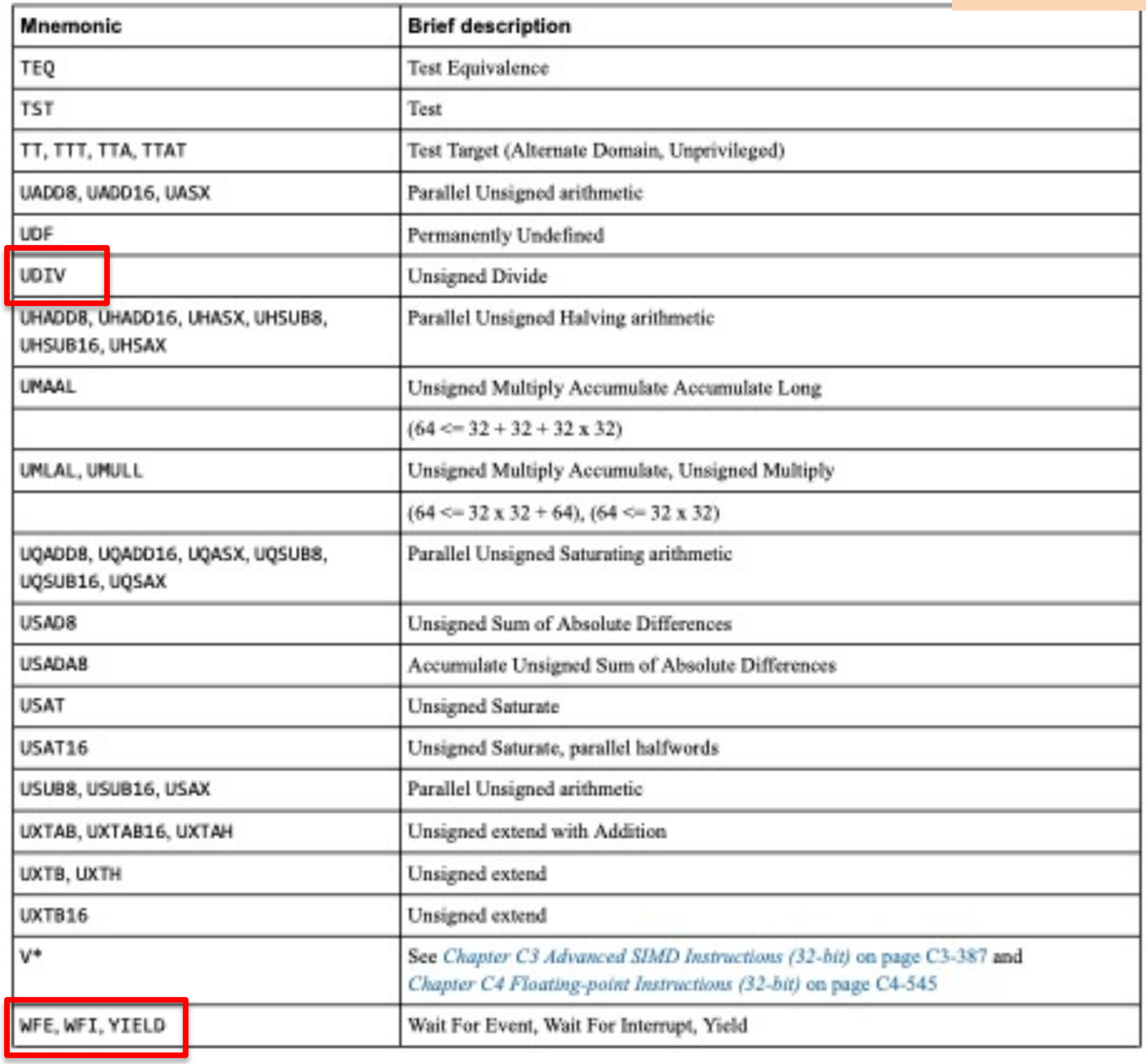
STRB	Store Register with Byte
STRBT	Store Register with Byte, user mode
STRD	Store Registers with two words
STREX, STREXB, STREXH,STREXD	Store Register Exclusive Word, Byte, Halfword, Doubleword
STRH	Store Register with Halfword
STRHT	Store Register with Halfword, user mode
STL, STLB, STLH	Store-Release Word, Byte, Halfword
STLEX, STLEXB, STLEXH, STLEXD	Store-Release Exclusive Word, Byte, Halfword, Doubleword
STRT	Store Register with word, user mode
SUB	Subtract
SUBS pc, lr	Exception return, no stack
SVC (formerly SWI)	Supervisor Call
SXTAB, SXTAB16, SXTAH	Signed extend, with Addition
SXTB, SXTH	Signed extend
SXTB16	Signed extend
SYS	Execute System coprocessor instruction
TBB, TBH	Table Branch Byte, Halfword

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP122

ARM Instruction Set

DR JEFF SOFTWARE
INDIE APP DEVELOPER
© Jeff Drobman
2016-2022

ARM Ref



Mnemonic	Brief description
TEQ	Test Equivalence
TST	Test
TT, TTT, TTA, TTAT	Test Target (Alternate Domain, Unprivileged)
UADD8, UADD16, UASX	Parallel Unsigned arithmetic
UDF	Permanently Undefined
UDIV	Unsigned Divide
UHADD8, UHADD16, UHASX, UHSUB8, UHSUB16, UHSAX	Parallel Unsigned Halving arithmetic
UMAAL	Unsigned Multiply Accumulate Accumulate Long
	(64 <= 32 + 32 + 32 x 32)
UMLAL, UMULL	Unsigned Multiply Accumulate, Unsigned Multiply
	(64 <= 32 x 32 + 64), (64 <= 32 x 32)
UQADD8, UQADD16, UQASX, UQSUB8, UQSUB16, UQSAX	Parallel Unsigned Saturating arithmetic
USAD8	Unsigned Sum of Absolute Differences
USADA8	Accumulate Unsigned Sum of Absolute Differences
USAT	Unsigned Saturate
USAT16	Unsigned Saturate, parallel halfwords
USUB8, USUB16, USAX	Parallel Unsigned arithmetic
UXTAB, UXTAB16, UXTAH	Unsigned extend with Addition
UXTB, UXTH	Unsigned extend
UXTB16	Unsigned extend
V*	See *Chapter C3 Advanced SIMD Instructions (32-bit)* on page C3-387 and *Chapter C4 Floating-point Instructions (32-bit)* on page C4-545
WFE, WFI, YIELD	Wait For Event, Wait For Interrupt, Yield

CSUN
CALIFORNIA
STATE UNIVERSITY
NORTHRIDGE

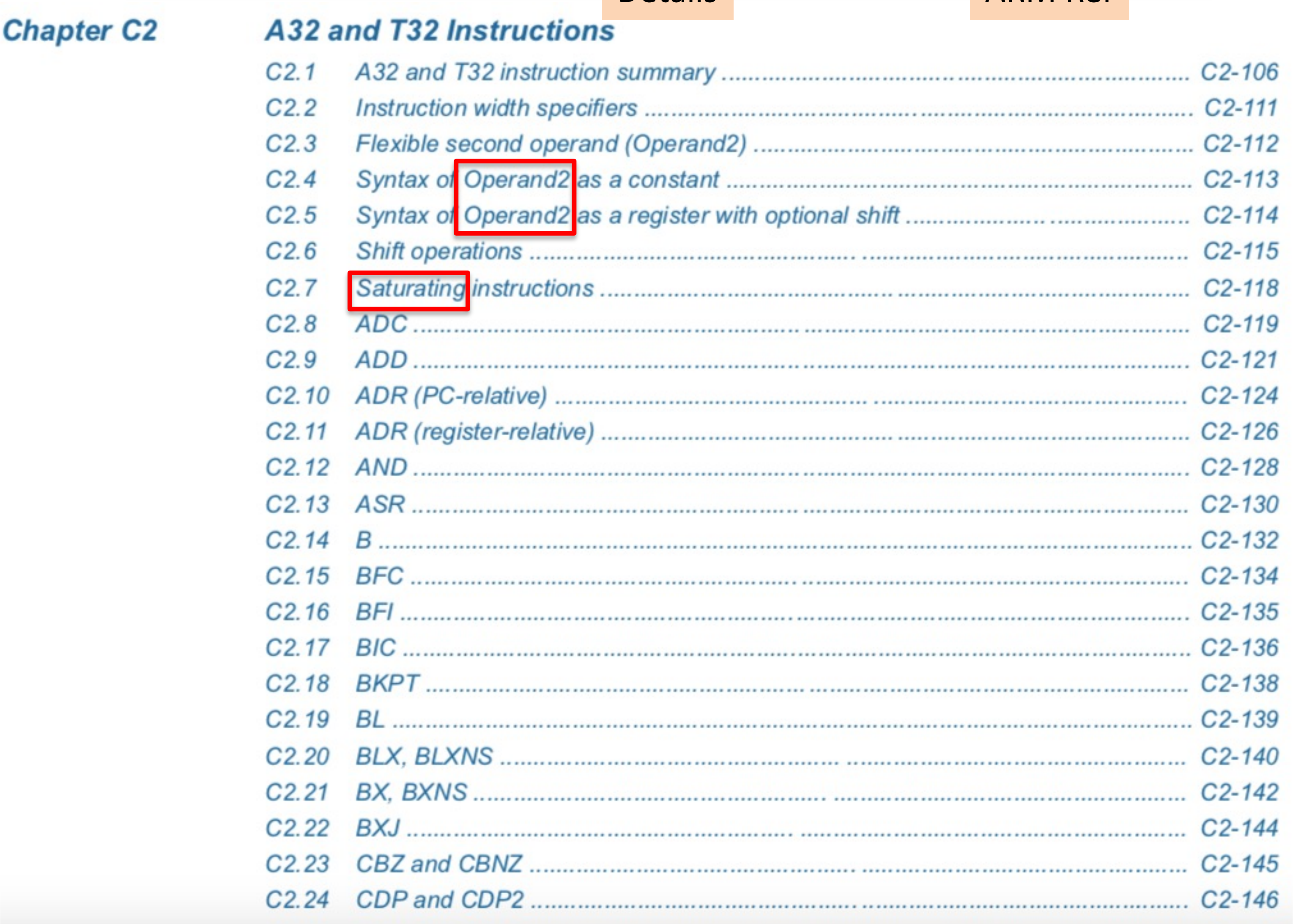
COMP122

ARM Assembly

DR JEFF SOFTWARE
INDIE APP DEVELOPER
© Jeff Drobman
2016-2022

Details

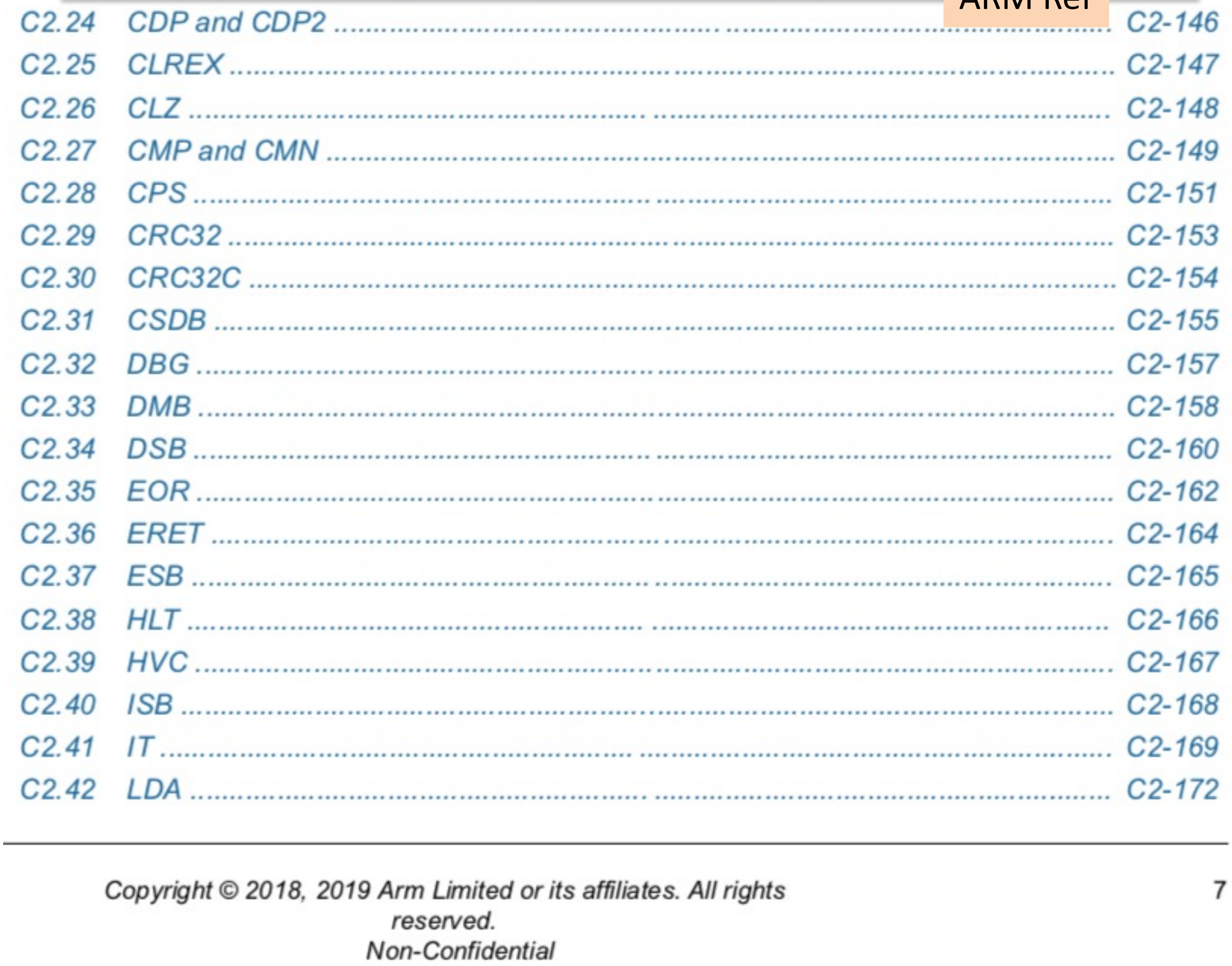
ARM Ref



Chapter C2 A32 and T32 Instructions

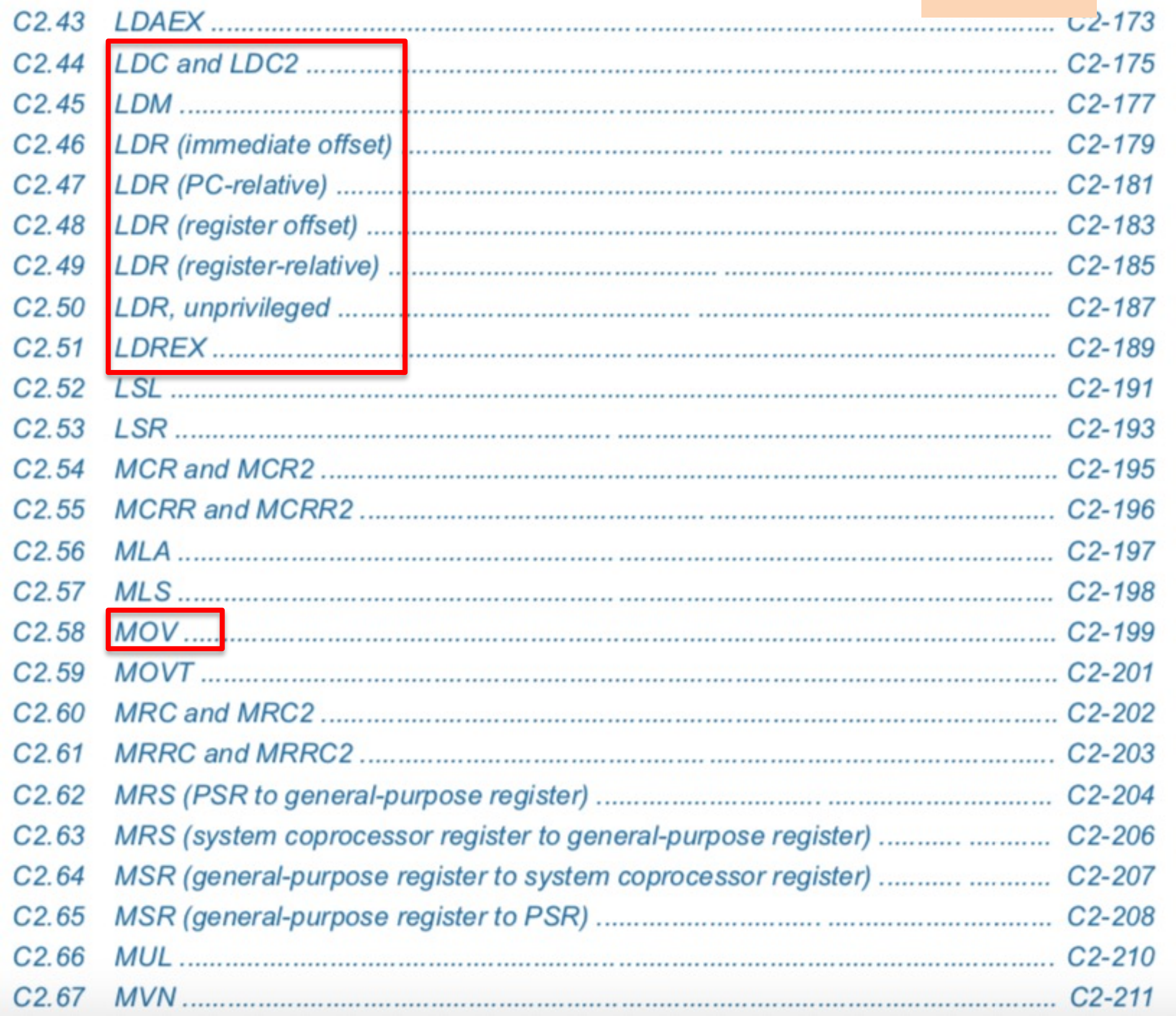
ARM Assembly

ARM Ref



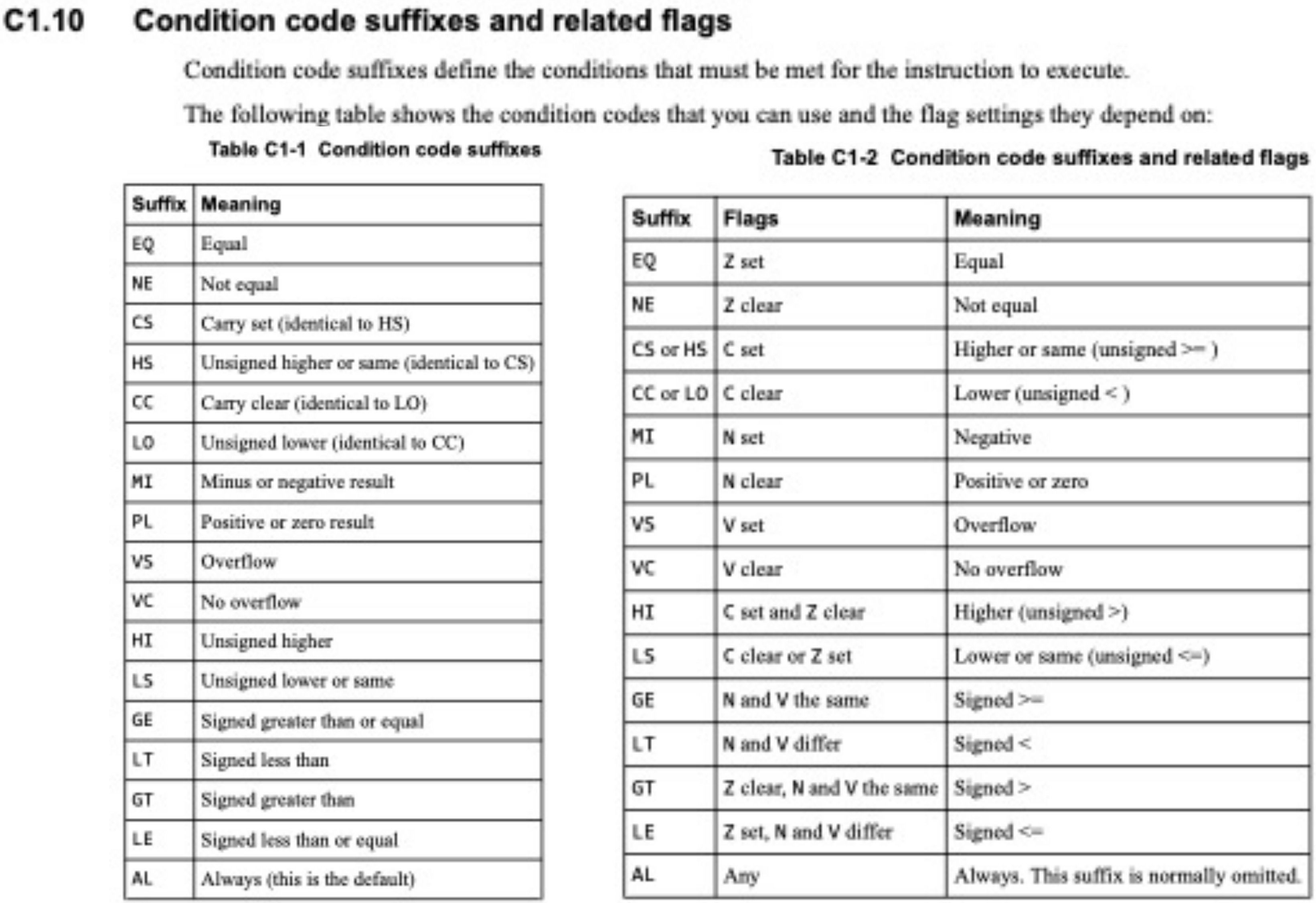
ARM Assembly

ARM Ref



ARM Conditionals

ARM Ref



C1.10 Condition code suffixes and related flags

Condition code suffixes define the conditions that must be met for the instruction to execute.

The following table shows the condition codes that you can use and the flag settings they depend on:

Table C1-1 Condition code suffixes

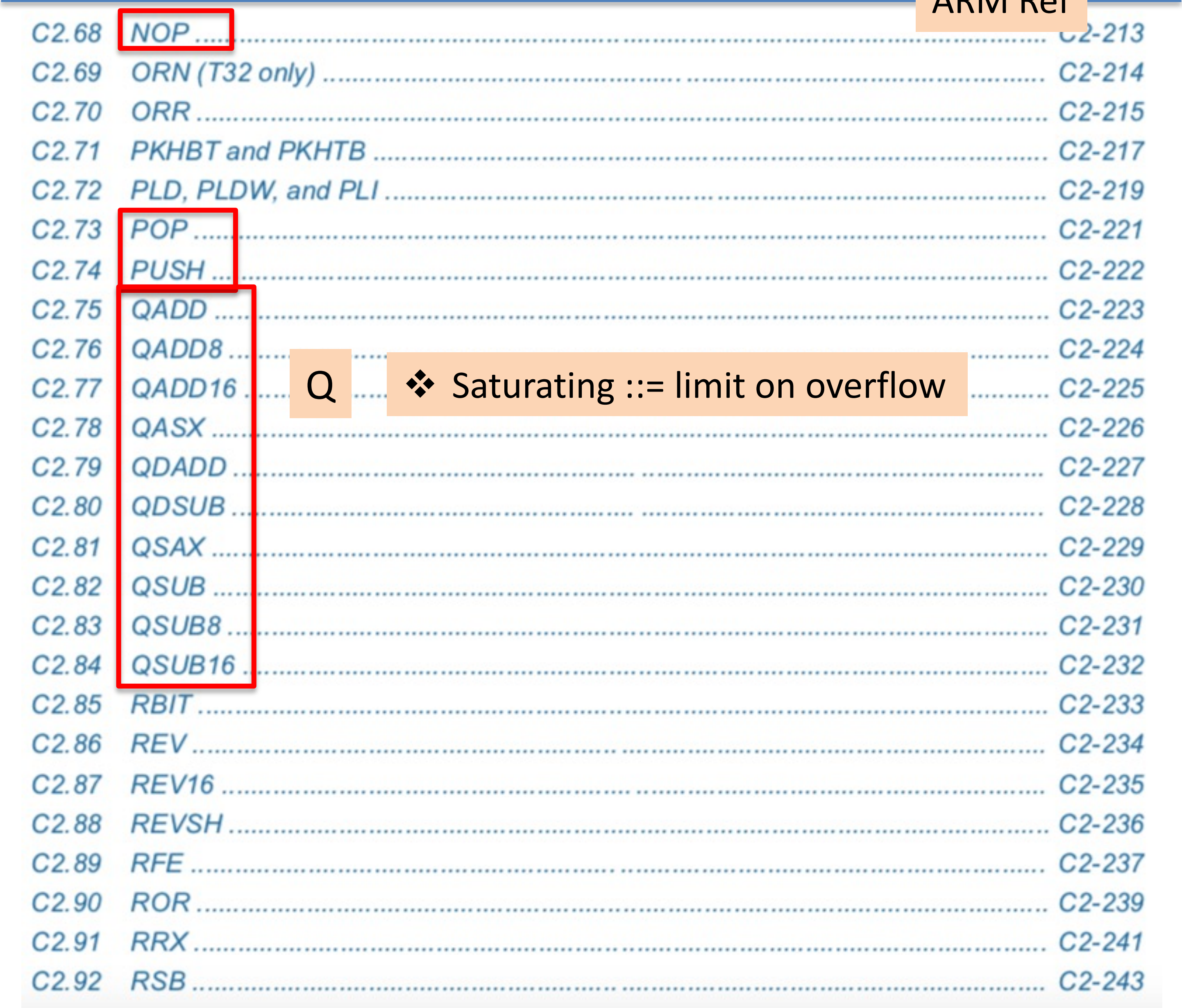
Suffix	Meaning
EQ	Equal
NE	Not equal
CS	Carry set (identical to HS)
HS	Unsigned higher or same (identical to CS)
CC	Carry clear (identical to LO)
LO	Unsigned lower (identical to CC)
MI	Minus or negative result
PL	Positive or zero result
VS	Overflow
VC	No overflow
HI	Unsigned higher
LS	Unsigned lower or same
GE	Signed greater than or equal
LT	Signed less than
GT	Signed greater than
LE	Signed less than or equal
AL	Always (this is the default)

Table C1-2 Condition code suffixes and related flags

Suffix	Flags	Meaning
EQ	Z set	Equal
NE	Z clear	Not equal
CS or HS	C set	Higher or same (unsigned >=)
CC or LO	C clear	Lower (unsigned <)
MI	N set	Negative
PL	N clear	Positive or zero
VS	V set	Overflow
VC	V clear	No overflow
HI	C set and Z clear	Higher (unsigned >)
LS	C clear or Z set	Lower or same (unsigned <=)
GE	N and V the same	Signed >=
LT	N and V differ	Signed <
GT	Z clear, N and V the same	Signed >
LE	Z set, N and V differ	Signed <=
AL	Any	Always. This suffix is normally omitted.

ARM Assembly

ARM Ref



Q

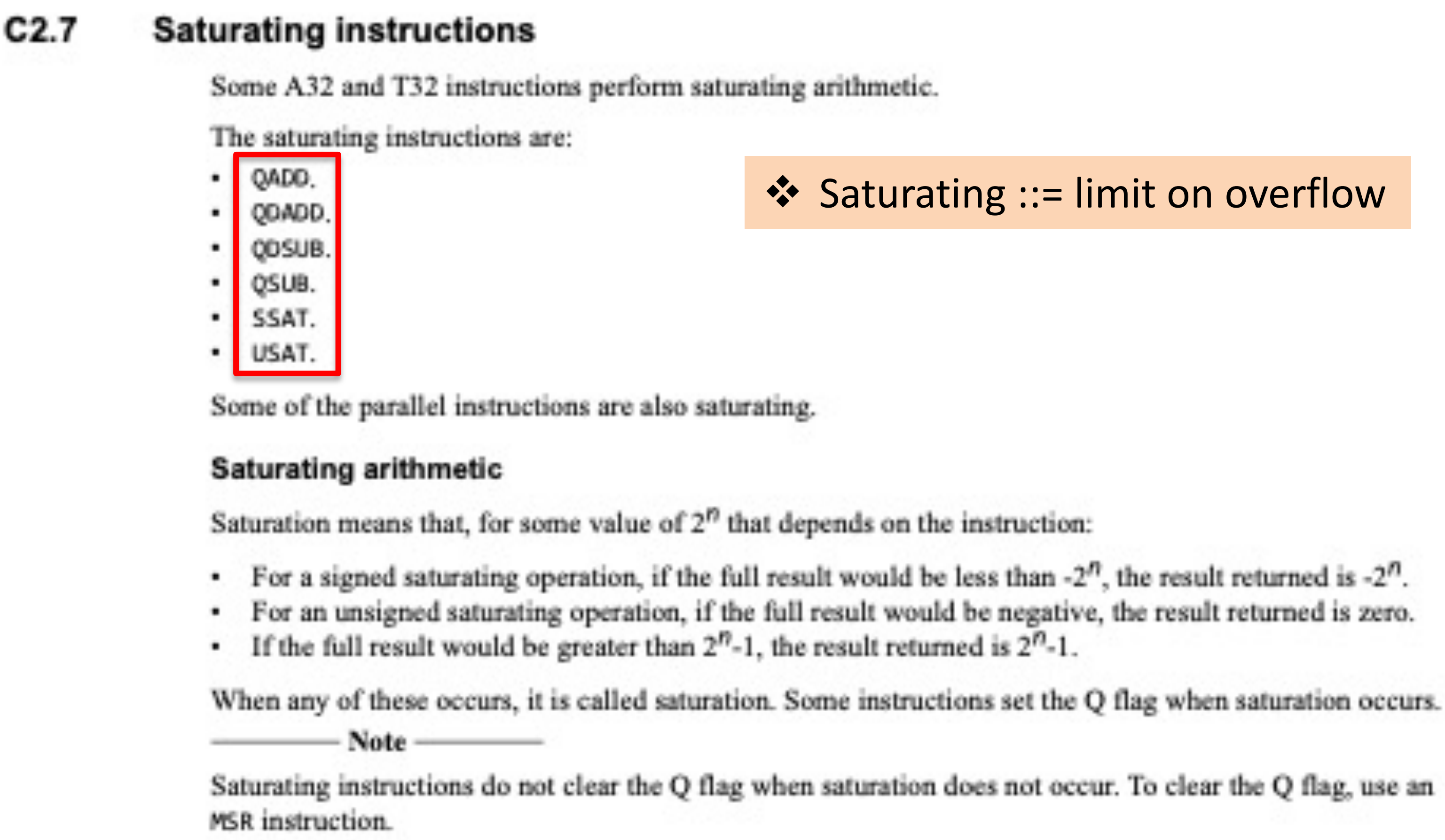
- ❖ Saturating ::= limit on overflow

ARM Assembly

Q

ARM Ref

❖ Saturating ::= limit on overflow



C2.7 Saturating instructions

Some A32 and T32 instructions perform saturating arithmetic.

The saturating instructions are:

- QADD.
- QDADD.
- QDSUB.
- QSUB.
- SSAT.
- USAT.

Some of the parallel instructions are also saturating.

Saturating arithmetic

Saturation means that, for some value of 2^n that depends on the instruction:

- For a signed saturating operation, if the full result would be less than -2^n, the result returned is -2^n.
- For an unsigned saturating operation, if the full result would be negative, the result returned is zero.
- If the full result would be greater than 2^n-1, the result returned is 2^n-1.

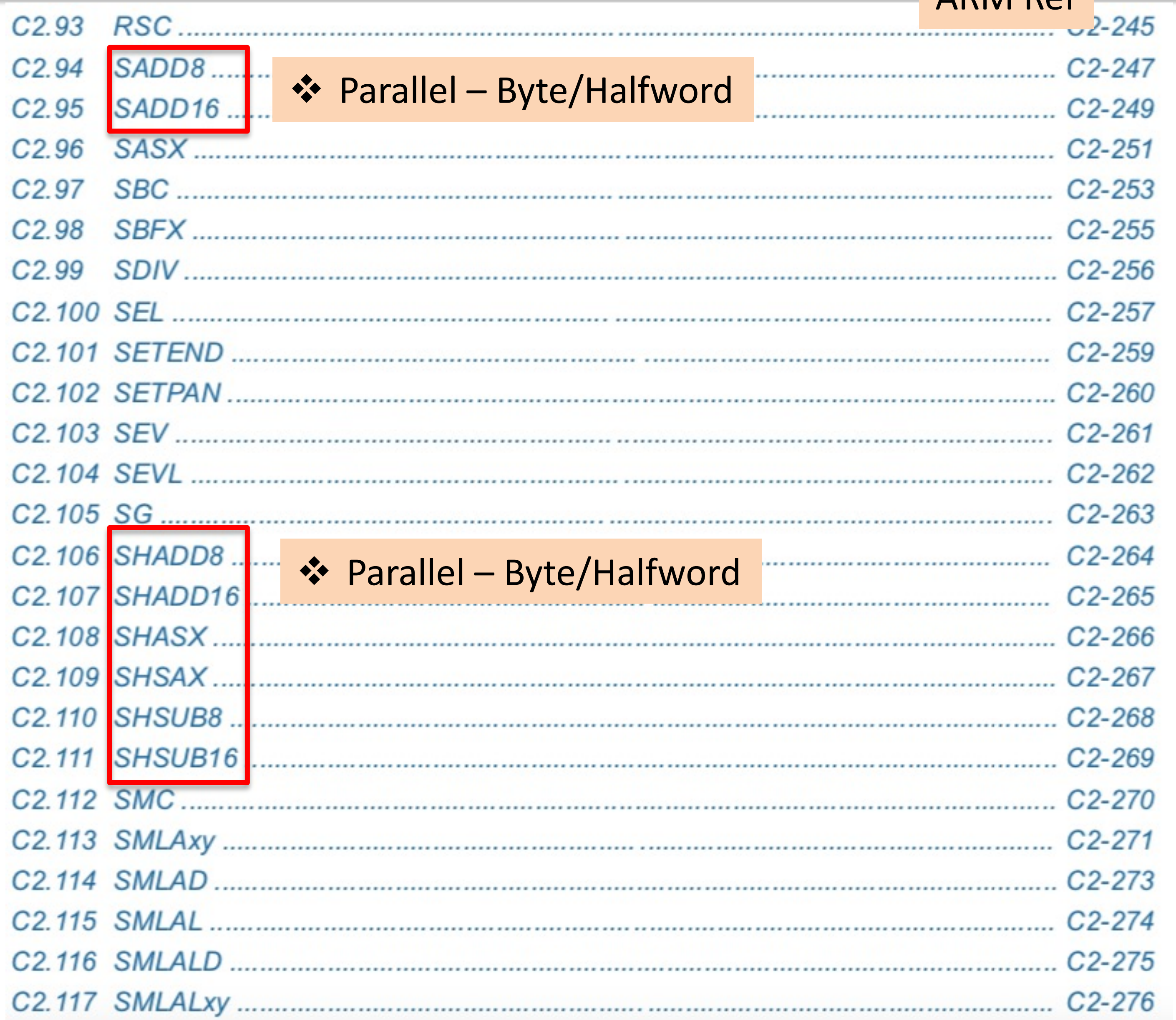
When any of these occurs, it is called saturation. Some instructions set the Q flag when saturation occurs.

Note

Saturating instructions do not clear the Q flag when saturation does not occur. To clear the Q flag, use an MSR instruction.

ARM Assembly

ARM Ref

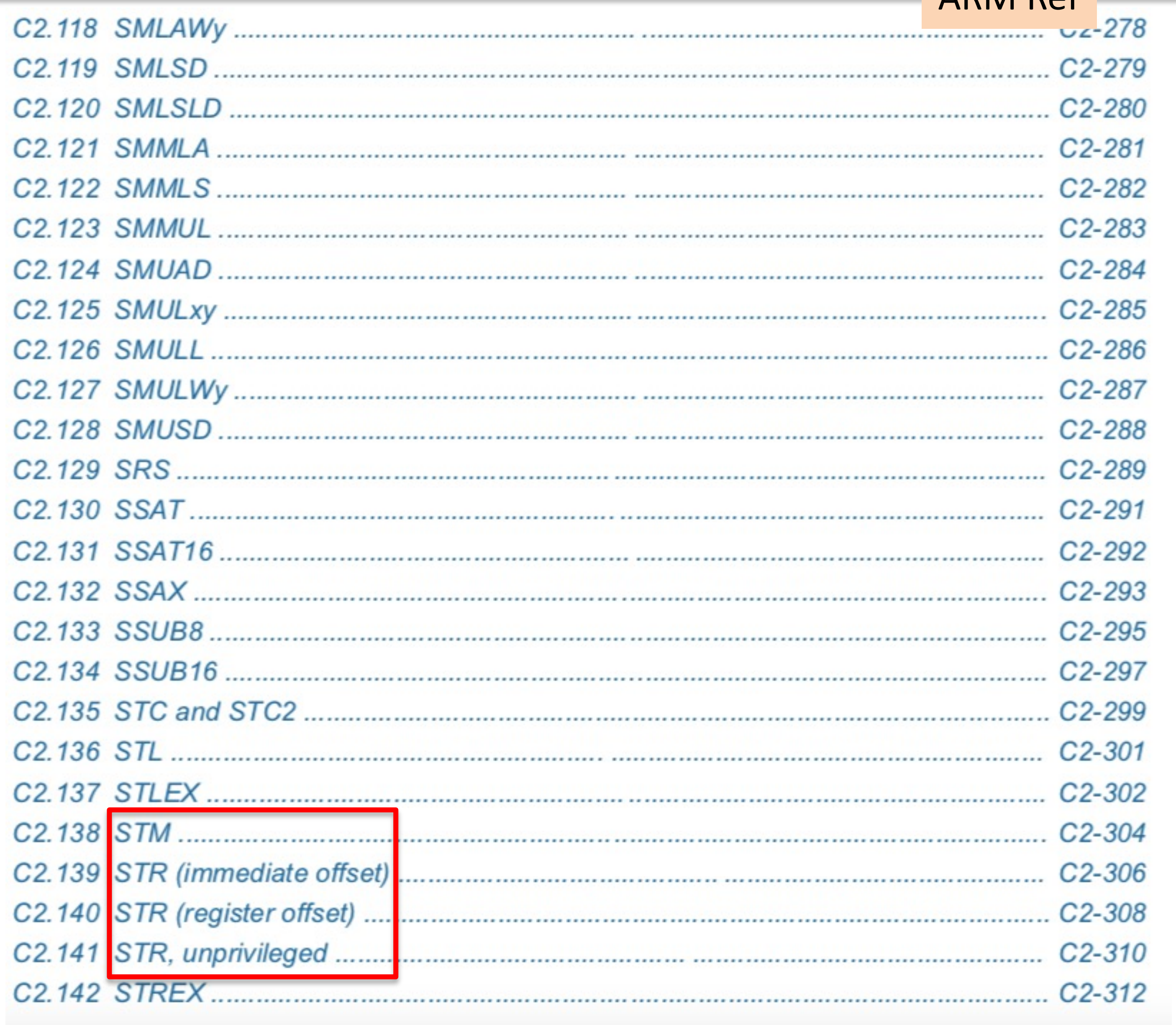


❖ Parallel – Byte/Halfword

❖ Parallel – Byte/Halfword

ARM Assembly

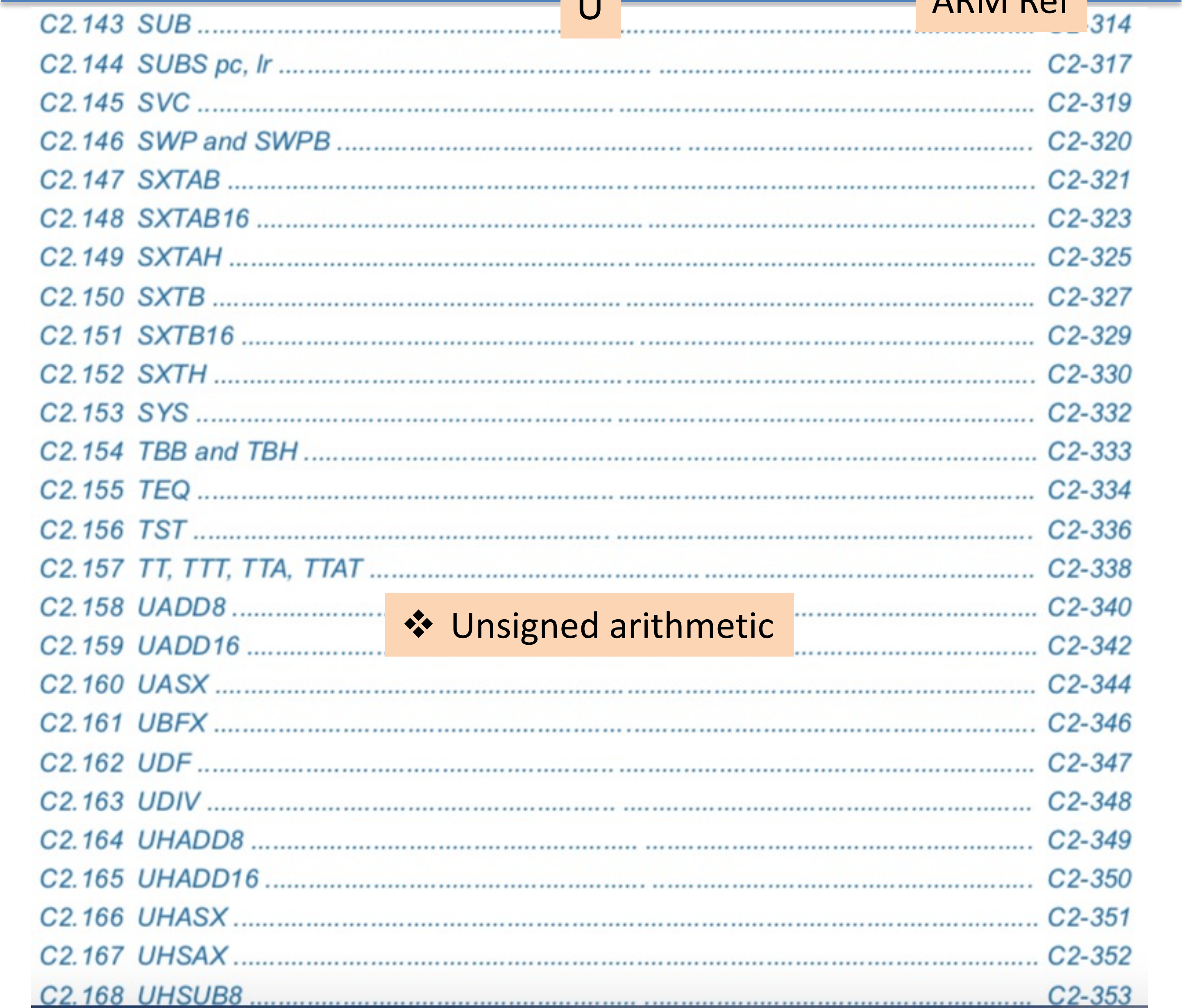
ARM Ref



ARM Assembly

U

ARM Ref

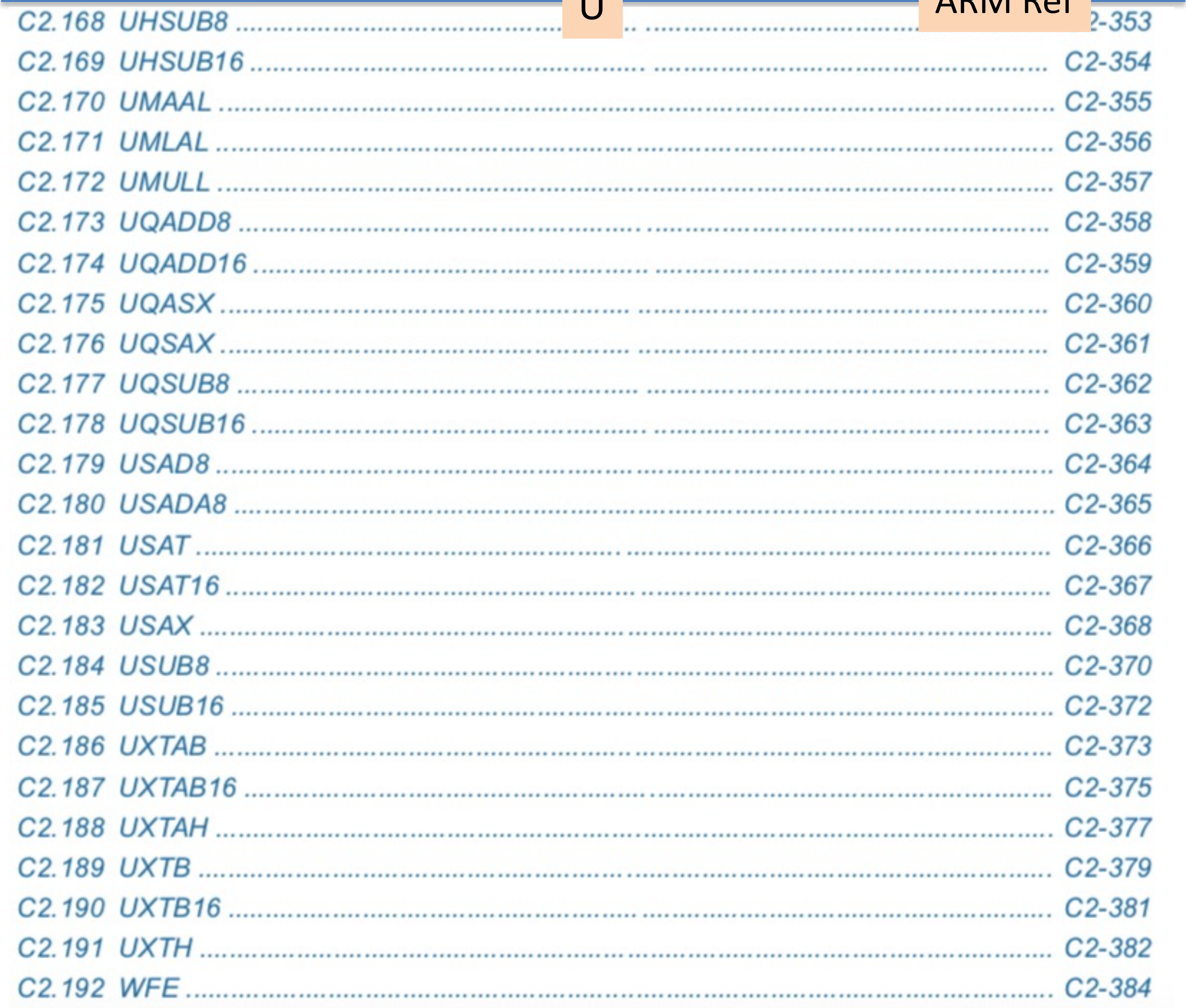


❖ Unsigned arithmetic

ARM Assembly

U

ARM Ref

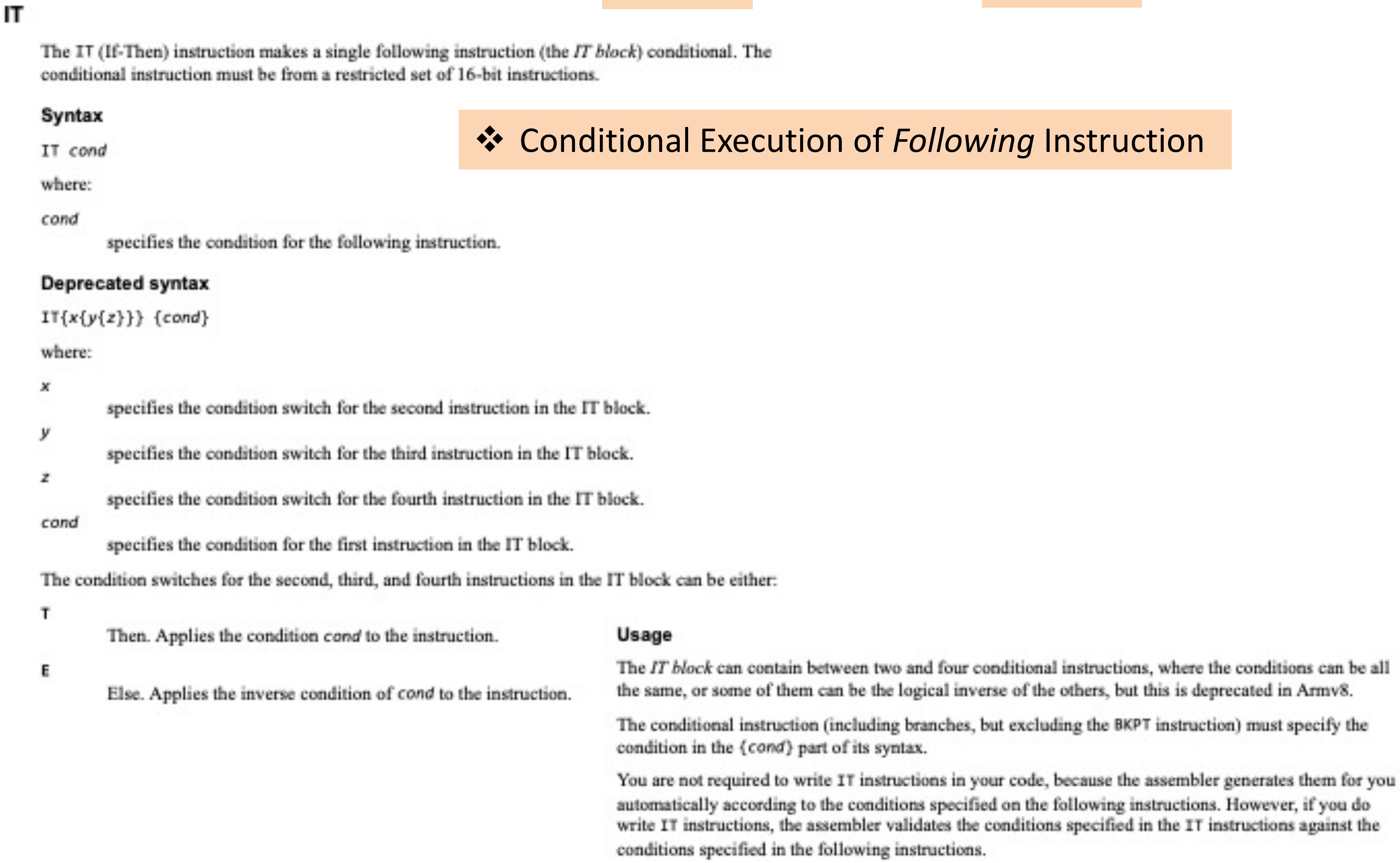


ARM Assembly

IF-THEN

ARM Ref

- ❖ Conditional Execution of *Following* Instruction



IT

The IT (If-Then) instruction makes a single following instruction (the *IT block*) conditional. The conditional instruction must be from a restricted set of 16-bit instructions.

Syntax

IT *cond*

where:

cond
specifies the condition for the following instruction.

Deprecated syntax

IT{*x*{*y*{*z*}}} {*cond*}

where:

x
specifies the condition switch for the second instruction in the IT block.

y
specifies the condition switch for the third instruction in the IT block.

z
specifies the condition switch for the fourth instruction in the IT block.

cond
specifies the condition for the first instruction in the IT block.

The condition switches for the second, third, and fourth instructions in the IT block can be either:

T
Then. Applies the condition *cond* to the instruction.

E
Else. Applies the inverse condition of *cond* to the instruction.

Usage

The *IT block* can contain between two and four conditional instructions, where the conditions can be all the same, or some of them can be the logical inverse of the others, but this is deprecated in Armv8.

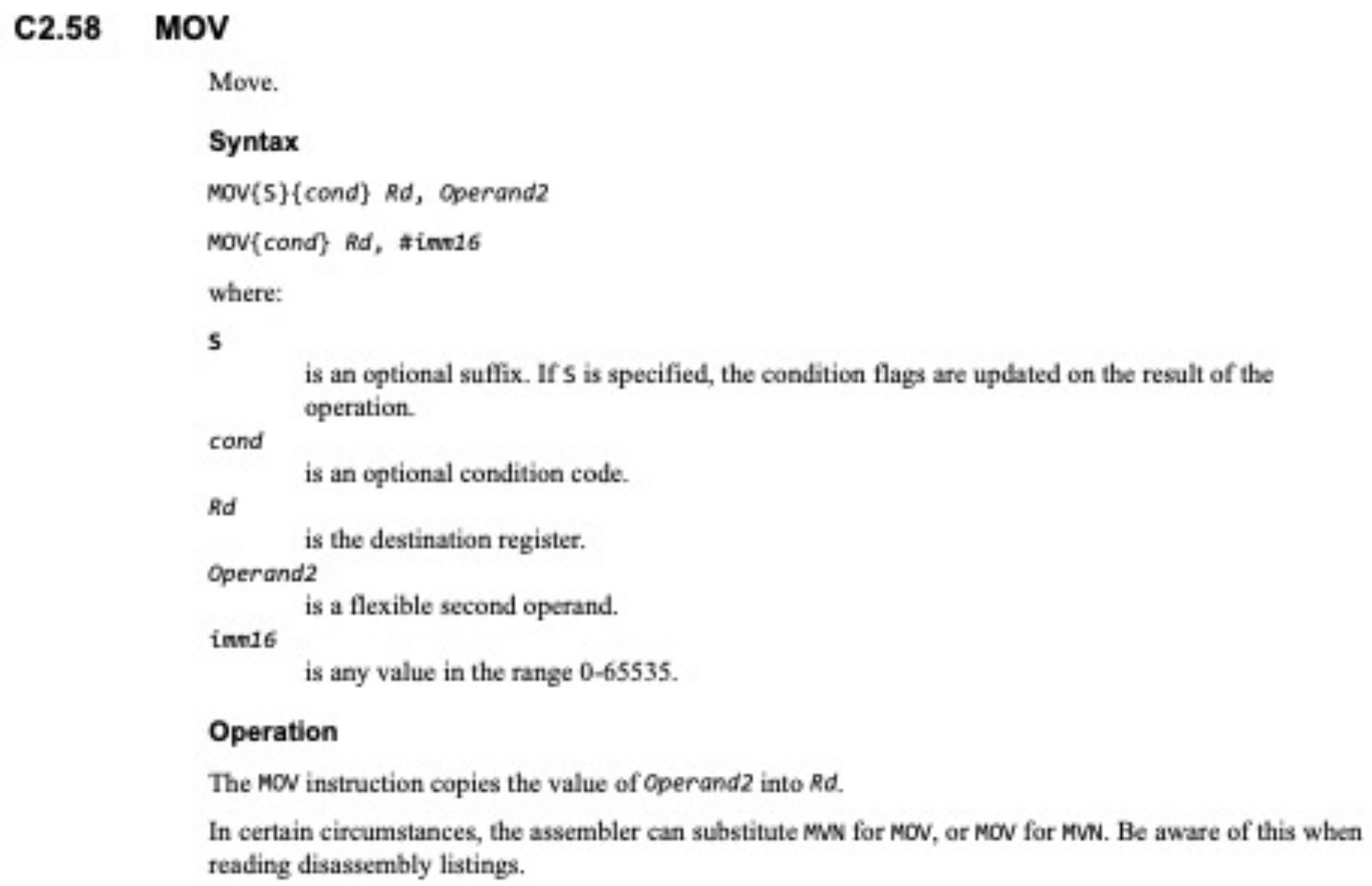
The conditional instruction (including branches, but excluding the BKPT instruction) must specify the condition in the {*cond*} part of its syntax.

You are not required to write IT instructions in your code, because the assembler generates them for you automatically according to the conditions specified on the following instructions. However, if you do write IT instructions, the assembler validates the conditions specified in the IT instructions against the conditions specified in the following instructions.

ARM Assembly

MOV

ARM Ref



C2.58 MOV

Move.

Syntax

```
MOV{S}{cond} Rd, Operand2
MOV{cond} Rd, #imm16
```

where:

`S`
: is an optional suffix. If `S` is specified, the condition flags are updated on the result of the operation.

cond
: is an optional condition code.

Rd
: is the destination register.

Operand2
: is a flexible second operand.

imm16
: is any value in the range 0-65535.

Operation

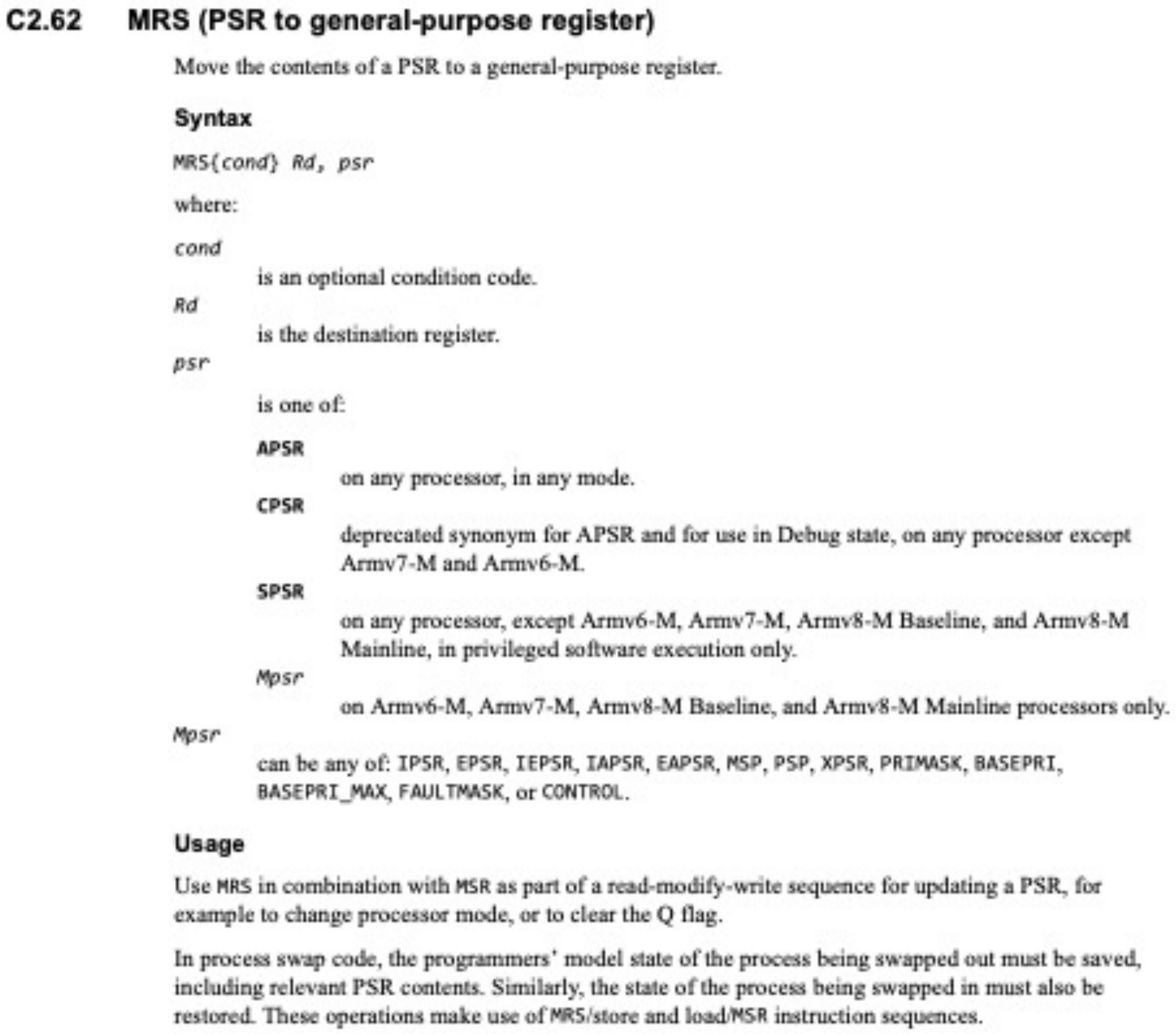
The `MOV` instruction copies the value of *Operand2* into *Rd*.

In certain circumstances, the assembler can substitute `MVN` for `MOV`, or `MOV` for `MVN`. Be aware of this when reading disassembly listings.

ARM Assembly

PSR

ARM Ref



C2.62 MRS (PSR to general-purpose register)

Move the contents of a PSR to a general-purpose register.

Syntax

`MRS{cond} Rd, psr`

where:

cond
is an optional condition code.

Rd
is the destination register.

psr
is one of:

APSR
on any processor, in any mode.

CPSR
deprecated synonym for APSR and for use in Debug state, on any processor except Armv7-M and Armv6-M.

SPSR
on any processor, except Armv6-M, Armv7-M, Armv8-M Baseline, and Armv8-M Mainline, in privileged software execution only.

Mpsr
on Armv6-M, Armv7-M, Armv8-M Baseline, and Armv8-M Mainline processors only.

Mpsr
can be any of: `IPSR`, `EPSR`, `IEPSR`, `IAPSR`, `EAPSR`, `MSP`, `PSP`, `XPSR`, `PRIMASK`, `BASEPRI`, `BASEPRI_MAX`, `FAULTMASK`, or `CONTROL`.

Usage

Use `MRS` in combination with `MSR` as part of a read-modify-write sequence for updating a PSR, for example to change processor mode, or to clear the Q flag.

In process swap code, the programmers' model state of the process being swapped out must be saved, including relevant PSR contents. Similarly, the state of the process being swapped in must also be restored. These operations make use of `MRS`/store and load/`MSR` instruction sequences.

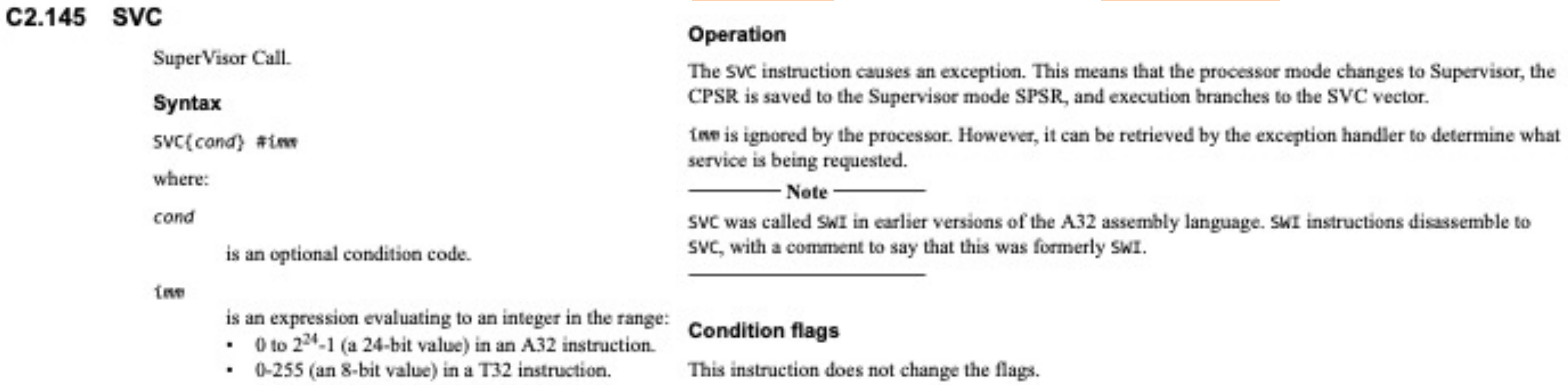
CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP122

ARM Assembly

DR JEFF SOFTWARE
INDIE APP DEVELOPER
© Jeff Drobman
2016-2022

Syscall

ARM Ref



C2.145 SVC

SuperVisor Call.

Syntax

`SVC{cond} #imm`

where:

cond

is an optional condition code.

imm

is an expression evaluating to an integer in the range:

- 0 to 2^{24}-1 (a 24-bit value) in an A32 instruction.
- 0-255 (an 8-bit value) in a T32 instruction.

Operation

The SVC instruction causes an exception. This means that the processor mode changes to Supervisor, the CPSR is saved to the Supervisor mode SPSR, and execution branches to the SVC vector.

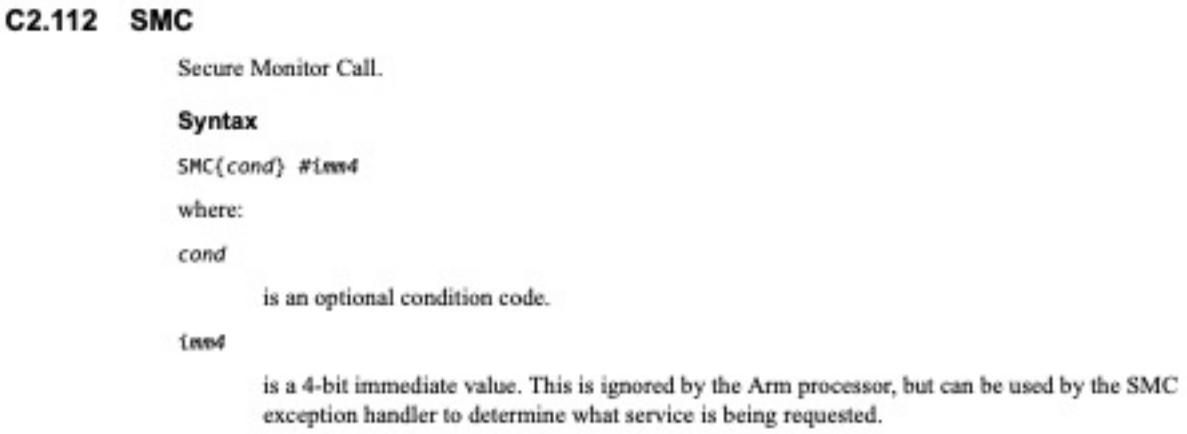
imm is ignored by the processor. However, it can be retrieved by the exception handler to determine what service is being requested.

Note

SVC was called SWI in earlier versions of the A32 assembly language. SWI instructions disassemble to SVC, with a comment to say that this was formerly SWI.

Condition flags

This instruction does not change the flags.



C2.112 SMC

Secure Monitor Call.

Syntax

`SMC{cond} #imm4`

where:

cond

is an optional condition code.

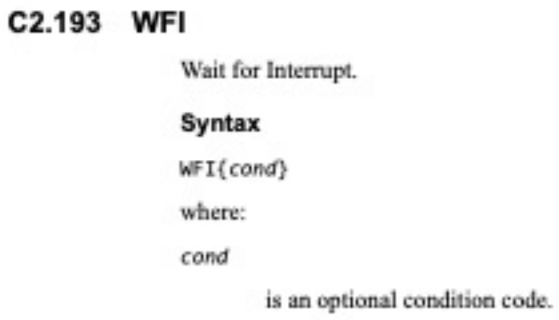
imm4

is a 4-bit immediate value. This is ignored by the Arm processor, but can be used by the SMC exception handler to determine what service is being requested.

ARM Assembly

Wait

ARM Ref



C2.193 WFI

Wait for Interrupt.

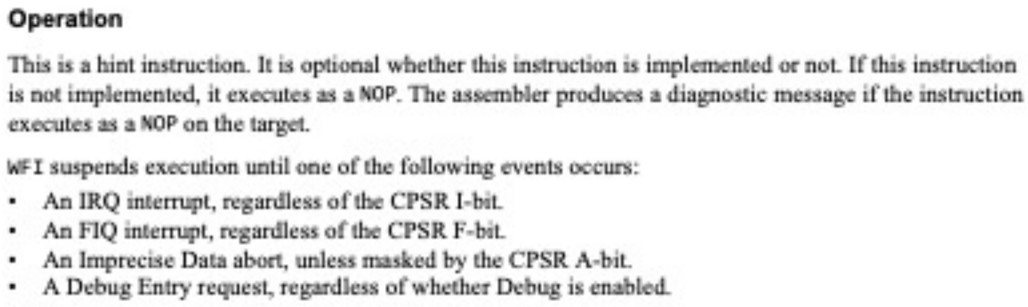
Syntax

`WFI{cond}`

where:

cond

is an optional condition code.

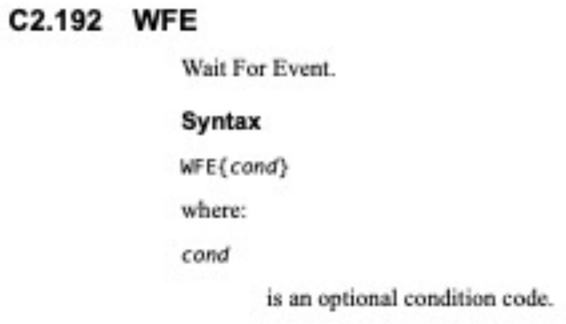


Operation

This is a hint instruction. It is optional whether this instruction is implemented or not. If this instruction is not implemented, it executes as a `NOP`. The assembler produces a diagnostic message if the instruction executes as a `NOP` on the target.

`WFI` suspends execution until one of the following events occurs:

- An IRQ interrupt, regardless of the CPSR I-bit.
- An FIQ interrupt, regardless of the CPSR F-bit.
- An Imprecise Data abort, unless masked by the CPSR A-bit.
- A Debug Entry request, regardless of whether Debug is enabled.



C2.192 WFE

Wait For Event.

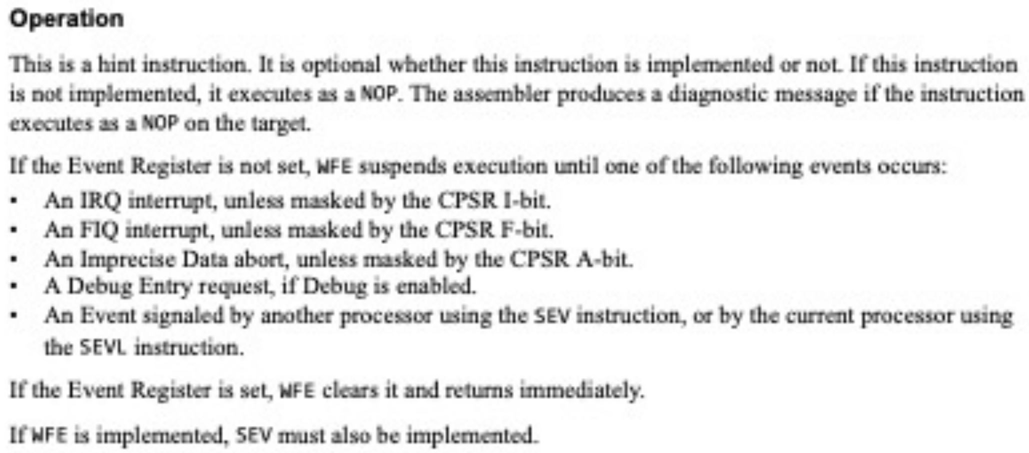
Syntax

`WFE{cond}`

where:

cond

is an optional condition code.



Operation

This is a hint instruction. It is optional whether this instruction is implemented or not. If this instruction is not implemented, it executes as a `NOP`. The assembler produces a diagnostic message if the instruction executes as a `NOP` on the target.

If the Event Register is not set, `WFE` suspends execution until one of the following events occurs:

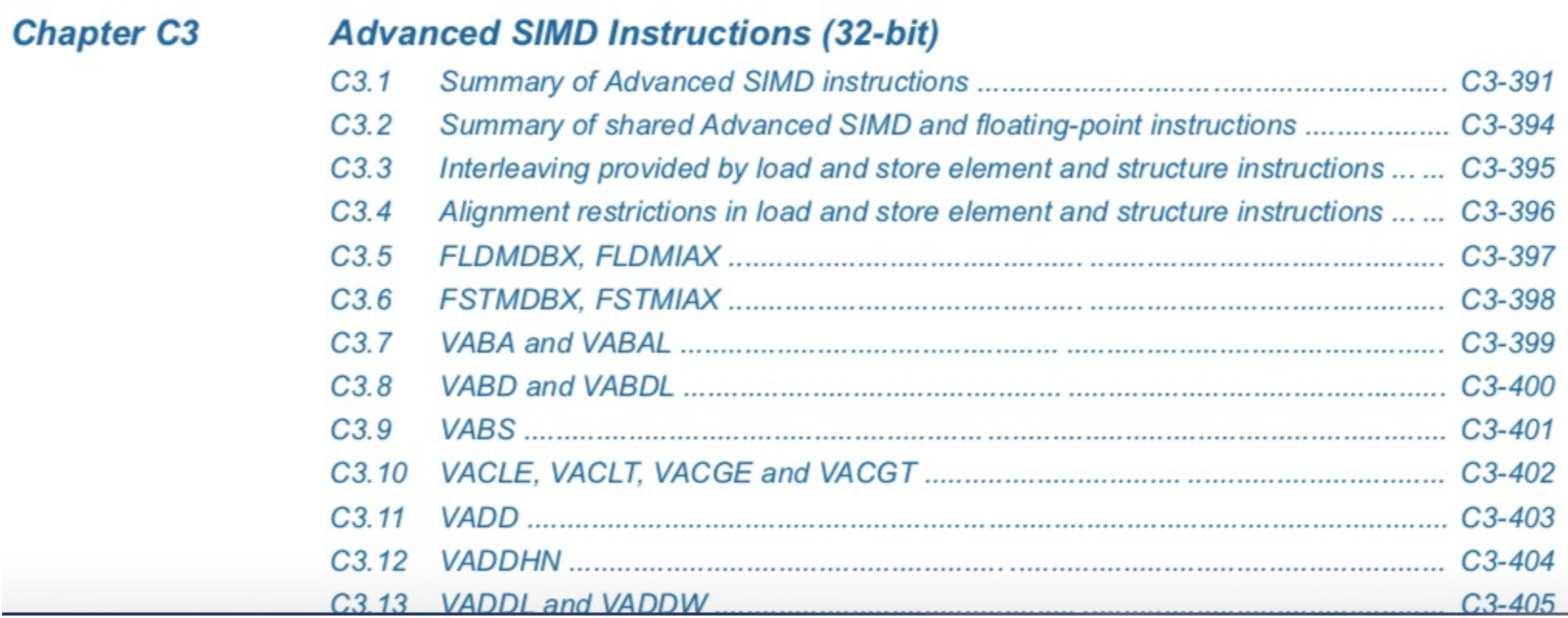
- An IRQ interrupt, unless masked by the CPSR I-bit.
- An FIQ interrupt, unless masked by the CPSR F-bit.
- An Imprecise Data abort, unless masked by the CPSR A-bit.
- A Debug Entry request, if Debug is enabled.
- An Event signaled by another processor using the `SEV` instruction, or by the current processor using the `SEVL` instruction.

If the Event Register is set, `WFE` clears it and returns immediately.

If `WFE` is implemented, `SEV` must also be implemented.

ARM Assembly

ARM Ref



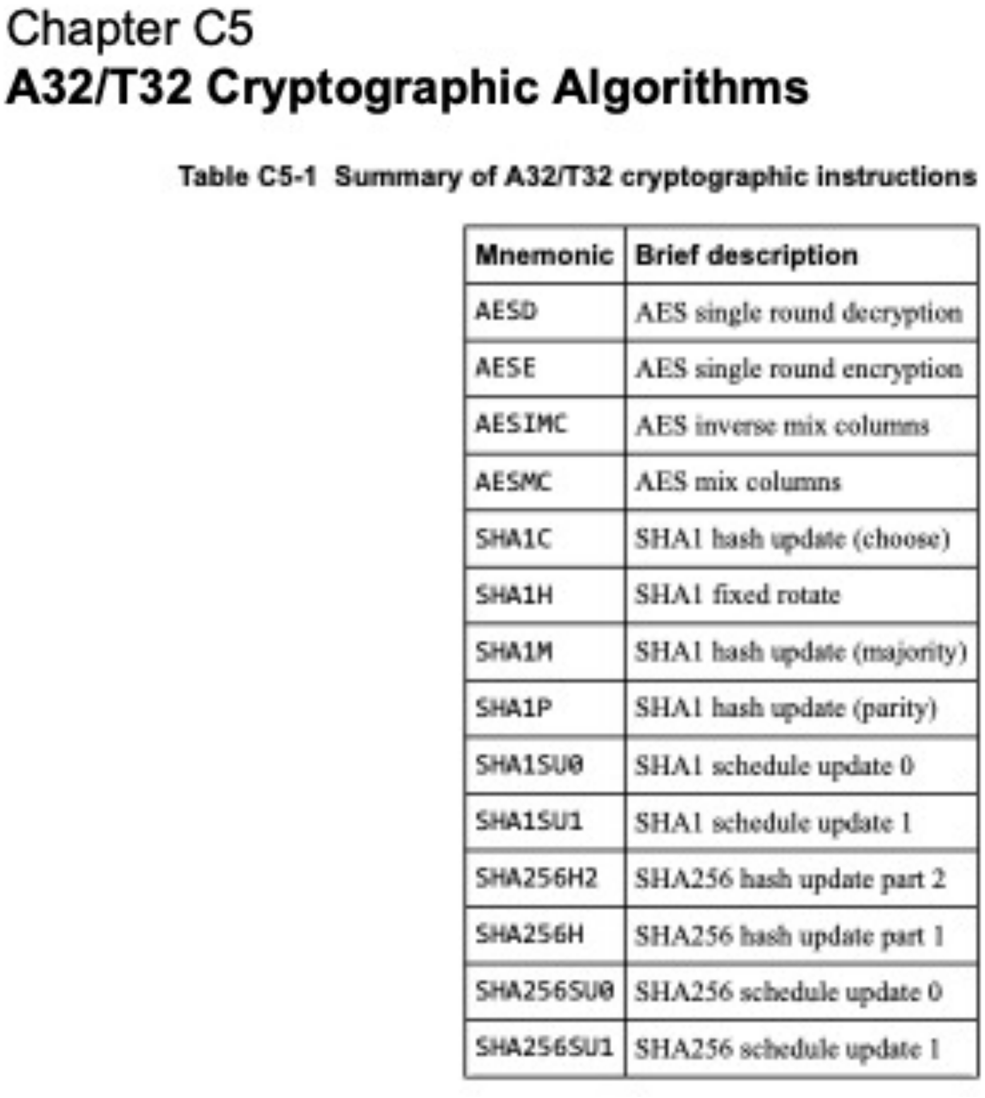
Chapter C3

Advanced SIMD Instructions (32-bit)

❖ SIMD ::= vector operations

Crypto Helpers

ARM Ref



Chapter C5
A32/T32 Cryptographic Algorithms

Table C5-1 Summary of A32/T32 cryptographic instructions

Mnemonic	Brief description
AESD	AES single round decryption
AESE	AES single round encryption
AESIMC	AES inverse mix columns
AESMC	AES mix columns
SHA1C	SHA1 hash update (choose)
SHA1H	SHA1 fixed rotate
SHA1M	SHA1 hash update (majority)
SHA1P	SHA1 hash update (parity)
SHA1SU0	SHA1 schedule update 0
SHA1SU1	SHA1 schedule update 1
SHA256H2	SHA256 hash update part 2
SHA256H	SHA256 hash update part 1
SHA256SU0	SHA256 schedule update 0
SHA256SU1	SHA256 schedule update 1