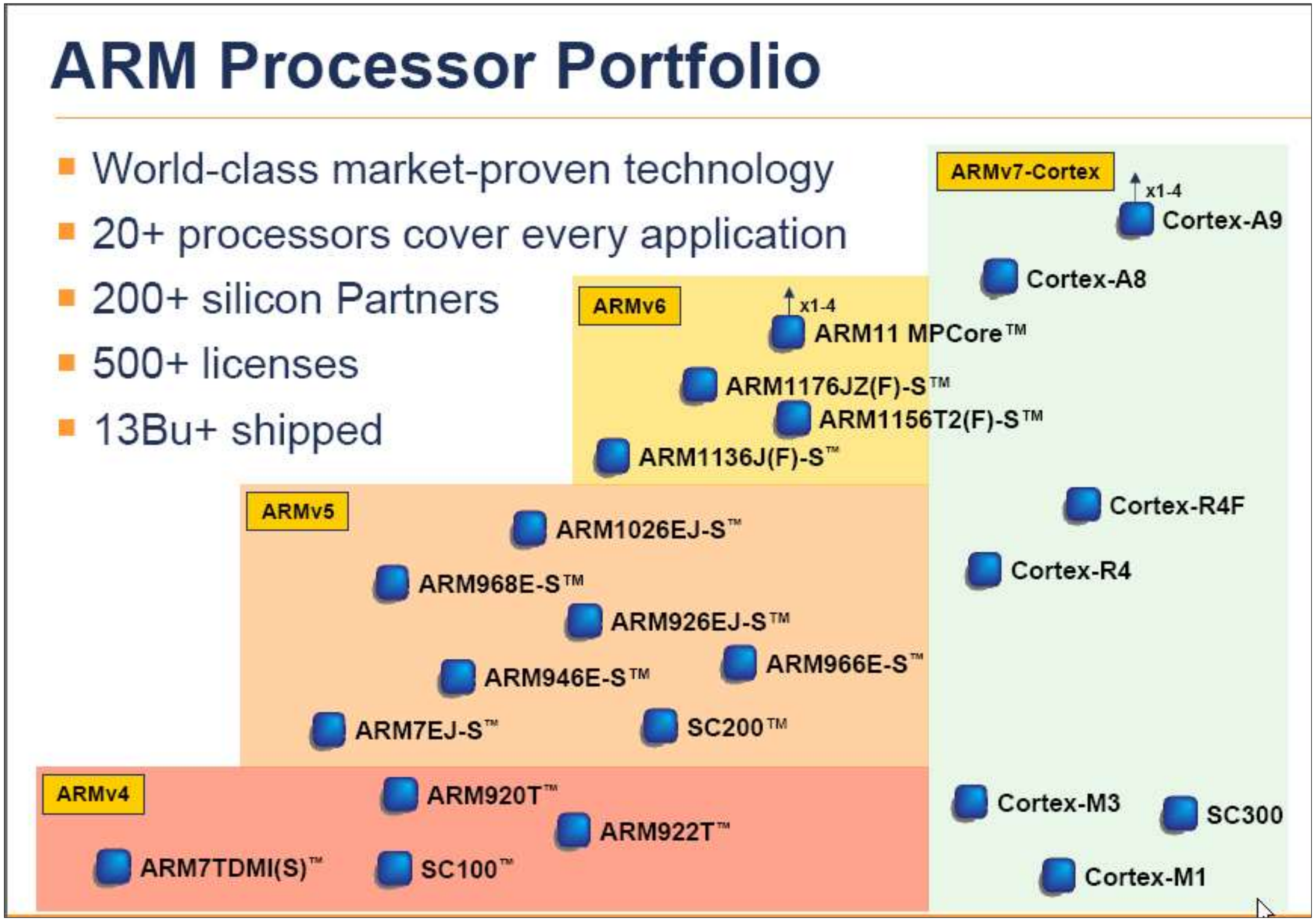
고려대학교
KOREA UNIVERSITY

COSE321 Computer Systems Design

Lecture 7. VFP & NEON in ARM

Prof. Taeweon Suh
Computer Science & Engineering
Korea University

ARM Processor Portfolio

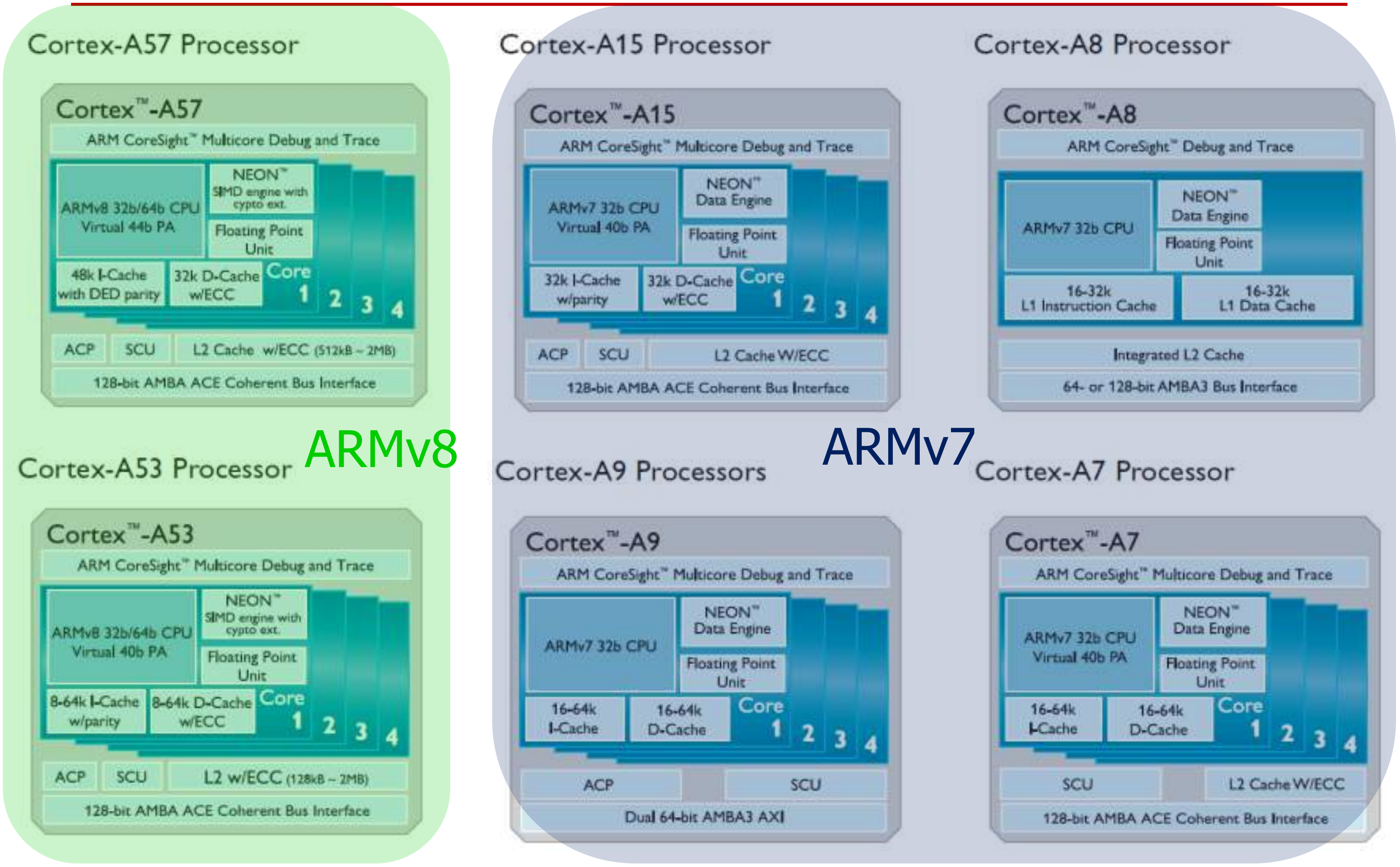


ARM Processor Portfolio

- World-class market-proven technology
- 20+ processors cover every application
- 200+ silicon Partners
- 500+ licenses
- 13Bu+ shipped

Source: 2008 Embedded SW Insight Conference

ARMv8-A & ARMv7-A

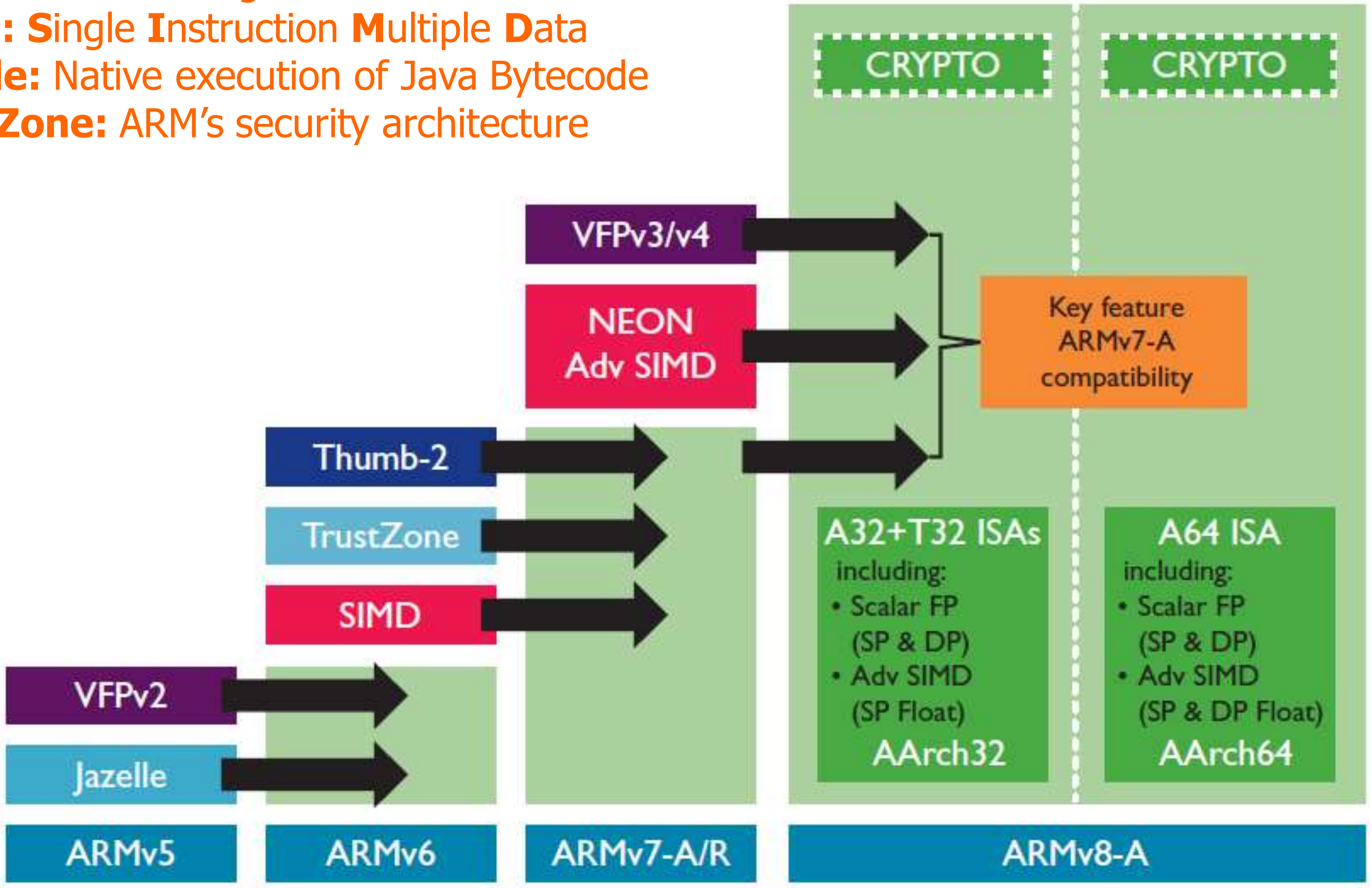


- ACP: Accelerator Coherency Port
- SCU: Snoop Control Unit

www.arm.com

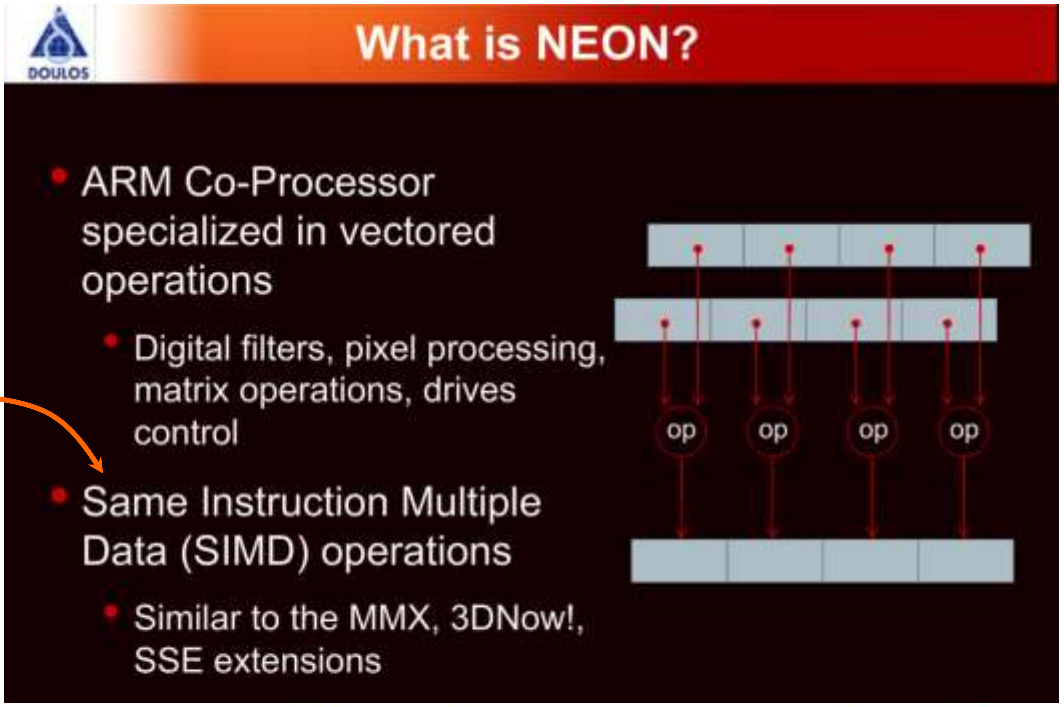
NEON & VFP

- **VFP: V**ector **F**loating **P**oint
- **SIMD: S**ingle **I**nstruction **M**ultiple **D**ata
- **Jazelle:** Native execution of Java Bytecode
- **TrustZone:** ARM's security architecture



www.arm.com

NEON


What is NEON?

- ARM Co-Processor specialized in vectored operations
  - Digital filters, pixel processing, matrix operations, drives control
- Same Instruction Multiple Data (SIMD) operations
  - Similar to the MMX, 3DNow!, SSE extensions

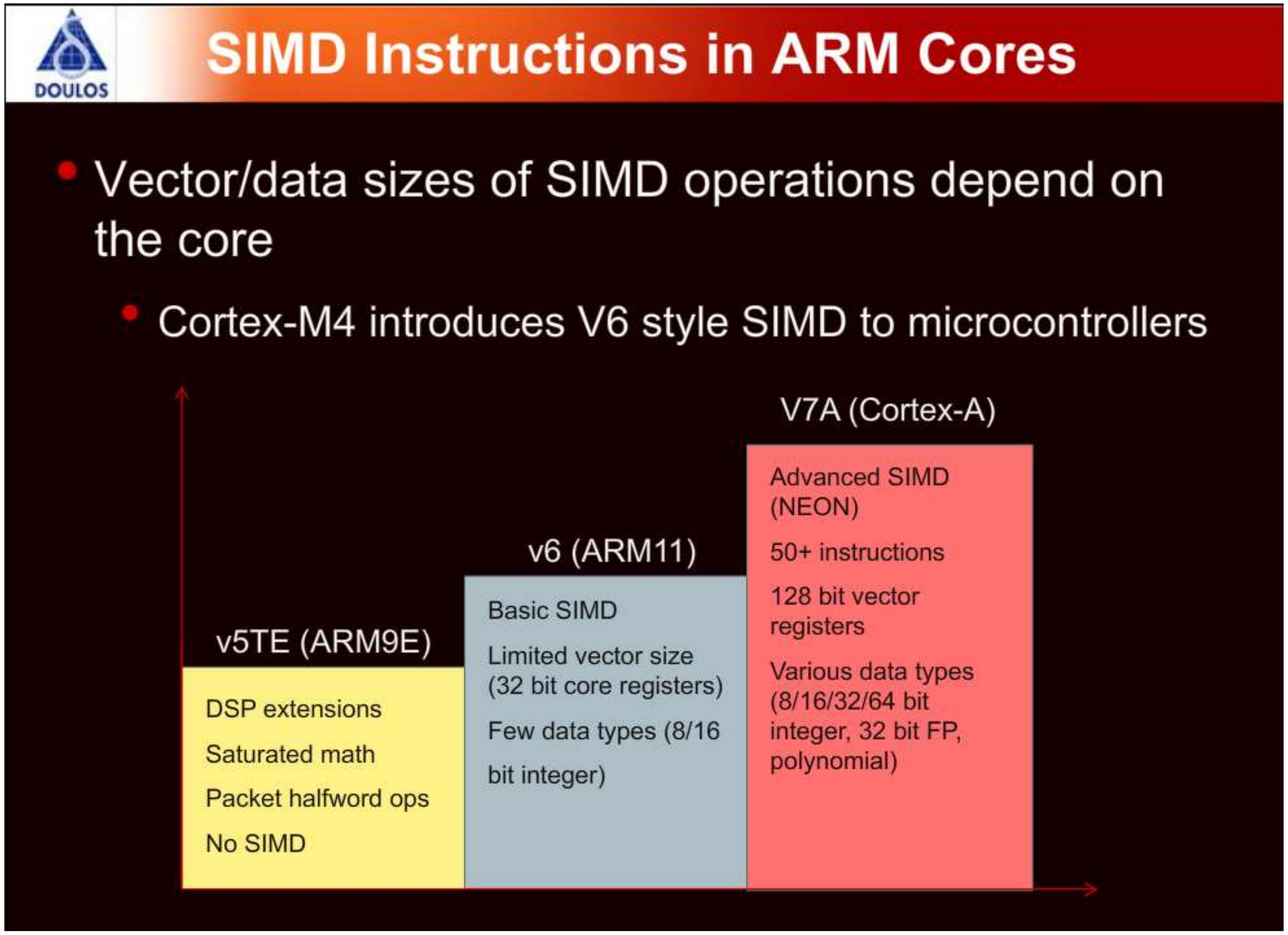
DOULOS
op
op
op
op

Single

The application processing unit (APU), located within the PS, contains two ARM Cortex A9 processors with NEON co-processors that are connected in an MP configuration sharing a 512 KB L2 cache. Each processor is a high-performance and low-power core that implements two separate 32 KB L1 caches for instruction and data. The Cortex-A9 processor implements the ARMv7-A architecture with full virtual memory support and can execute 32-bit ARM instructions, 16-bit and 32-bit Thumb instructions, and 8-bit Java™ byte codes in the Jazelle state. The NEON coprocessor's media and signal processing architecture adds instructions that target audio, video, image and speech processing, and 3D graphics. These advanced single instruction multiple data (SIMD) instructions are available in both ARM and Thumb states. A block diagram of the APU is shown in Figure 3-1.

http://www.doulos.com/knowhow/arm/using_your_c_compiler_to_exploit_neon/Resources/Presentation/Using_Neon_form_C.swf
http://esca.korea.ac.kr/teaching/cose321_CSD/labs/Zynq-7000-TRM.pdf

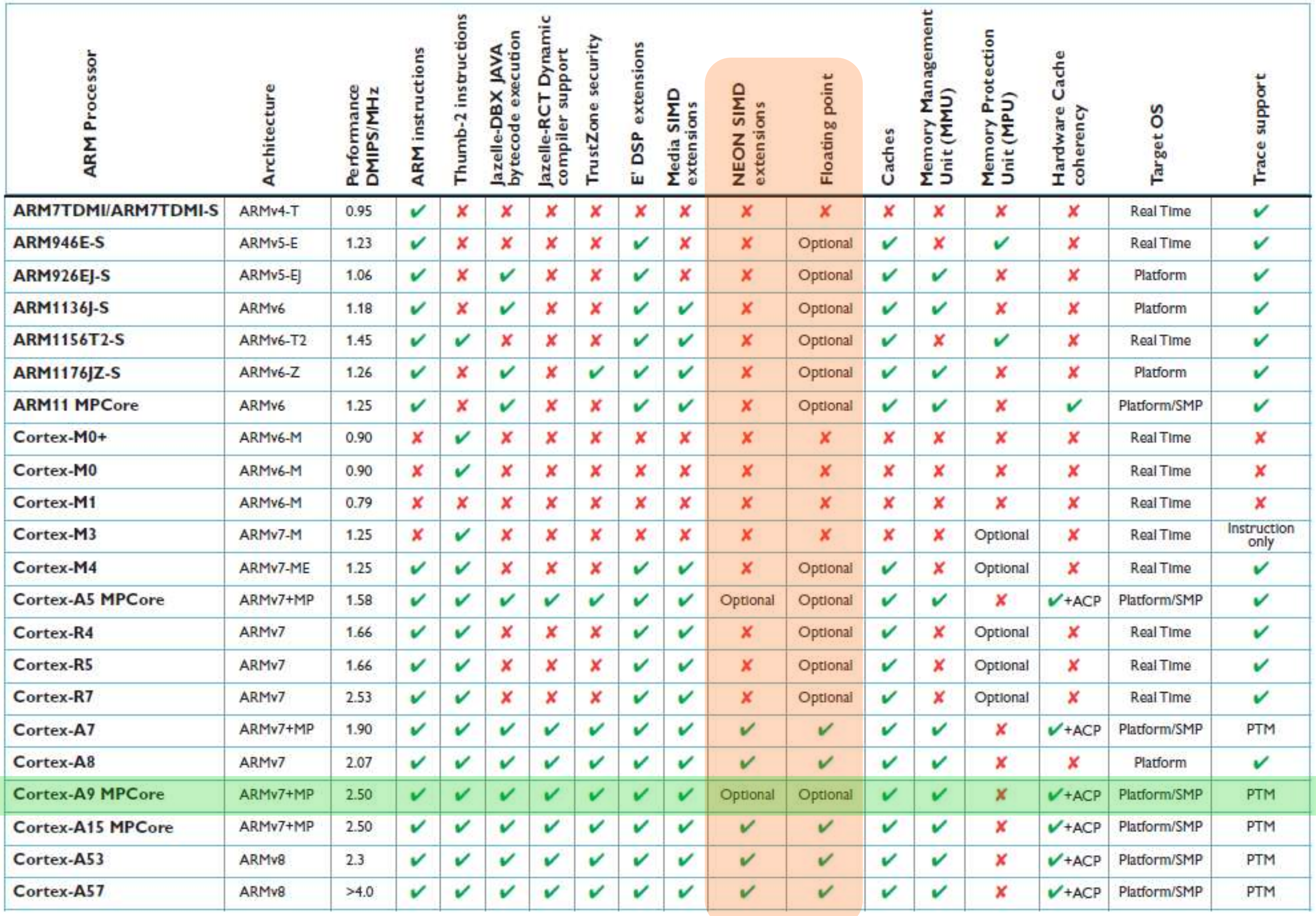
SIMD in ARM



SIMD Instructions in ARM Cores

- Vector/data sizes of SIMD operations depend on the core
  - Cortex-M4 introduces V6 style SIMD to microcontrollers

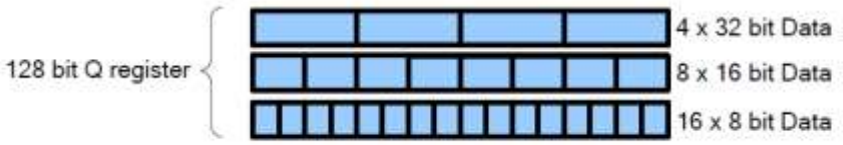
ARM Processor Selector



ARM Processor	Architecture	Performance DMIPS/MHz	ARM instructions	Thumb-2 instructions	Jazelle-DBX JAVA bytecode execution	Jazelle-RCT Dynamic compiler support	TrustZone security	E' DSP extensions	Media SIMD extensions	NEON SIMD extensions	Floating point	Caches	Memory Management Unit (MMU)	Memory Protection Unit (MPU)	Hardware Cache coherency	Target OS	Trace support
ARM7TDMI/ARM7TDMI-S	ARMv4-T	0.95	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	Real Time	✓
ARM946E-S	ARMv5-E	1.23	✓	✗	✗	✗	✗	✓	✗	✗	Optional	✓	✗	✓	✗	Real Time	✓
ARM926EJ-S	ARMv5-EJ	1.06	✓	✗	✓	✗	✗	✓	✗	✗	Optional	✓	✓	✗	✗	Platform	✓
ARM1136J-S	ARMv6	1.18	✓	✗	✓	✗	✗	✓	✓	✗	Optional	✓	✓	✗	✗	Platform	✓
ARM1156T2-S	ARMv6-T2	1.45	✓	✓	✗	✗	✗	✓	✓	✗	Optional	✓	✗	✓	✗	Real Time	✓
ARM1176JZ-S	ARMv6-Z	1.26	✓	✗	✓	✗	✓	✓	✓	✗	Optional	✓	✓	✗	✗	Platform	✓
ARM11 MPCore	ARMv6	1.25	✓	✗	✓	✗	✗	✓	✓	✗	Optional	✓	✓	✗	✓	Platform/SMP	✓
Cortex-M0+	ARMv6-M	0.90	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	Real Time	✗
Cortex-M0	ARMv6-M	0.90	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	Real Time	✗
Cortex-M1	ARMv6-M	0.79	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	Real Time	✗
Cortex-M3	ARMv7-M	1.25	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	Optional	✗	Real Time	Instruction only
Cortex-M4	ARMv7-ME	1.25	✓	✓	✗	✗	✗	✓	✓	✗	Optional	✓	✗	Optional	✗	Real Time	✓
Cortex-A5 MPCore	ARMv7+MP	1.58	✓	✓	✓	✓	✓	✓	✓	Optional	Optional	✓	✓	✗	✓+ACP	Platform/SMP	✓
Cortex-R4	ARMv7	1.66	✓	✓	✗	✗	✗	✓	✓	✗	Optional	✓	✗	Optional	✗	Real Time	✓
Cortex-R5	ARMv7	1.66	✓	✓	✗	✗	✗	✓	✓	✗	Optional	✓	✗	Optional	✗	Real Time	✓
Cortex-R7	ARMv7	2.53	✓	✓	✗	✗	✗	✓	✓	✗	Optional	✓	✗	Optional	✗	Real Time	✓
Cortex-A7	ARMv7+MP	1.90	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓+ACP	Platform/SMP	PTM
Cortex-A8	ARMv7	2.07	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	Platform	✓
Cortex-A9 MPCore	ARMv7+MP	2.50	✓	✓	✓	✓	✓	✓	✓	Optional	Optional	✓	✓	✗	✓+ACP	Platform/SMP	PTM
Cortex-A15 MPCore	ARMv7+MP	2.50	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓+ACP	Platform/SMP	PTM
Cortex-A53	ARMv8	2.3	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓+ACP	Platform/SMP	PTM
Cortex-A57	ARMv8	>4.0	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓+ACP	Platform/SMP	PTM

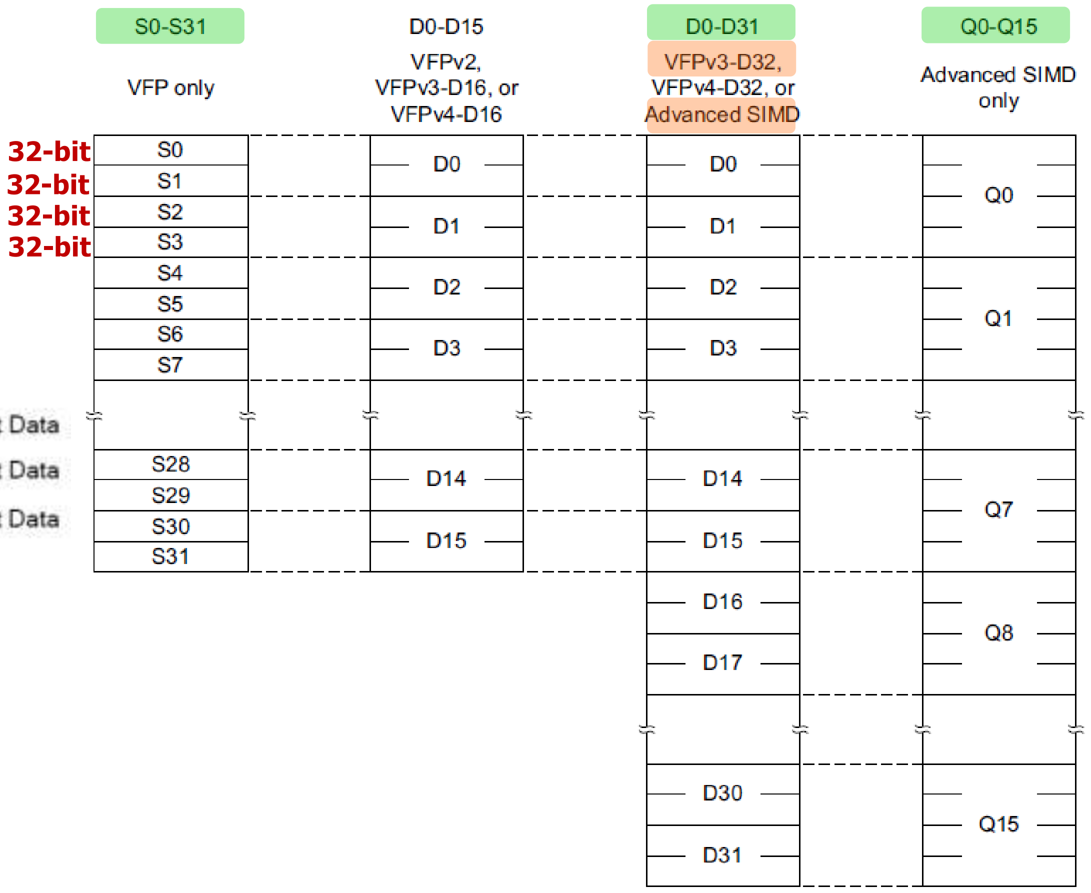
Register Mapping

- Cortex-A9 in Zynq-7000 supports ARMv7 **advanced SIMD** and **VPFv3**
- NEON Advanced SIMD and VFP use the **same** register set



Advanced SIMD and Floating-point register mapping

Figure A2-1 shows the different views of Advanced SIMD and Floating-point register banks, and the relationship between them.



S0-S31
VFP only
D0-D15
VFPv2, VFPv3-D16, or VFPv4-D16
D0-D31
VFPv3-D32, VFPv4-D32, or Advanced SIMD
Q0-Q15
Advanced SIMD only
32-bit
32-bit
32-bit
32-bit
S0 S1 S2 S3 S4 S5 S6 S7 S28 S29 S30 S31
D0 D1 D2 D3 D14 D15
D0 D1 D2 D3 D14 D15 D16 D17 D30 D31
Q0 Q1 Q7 Q8 Q15

Figure A2-1 Advanced SIMD and Floating-point Extensions register set

The mapping between the registers is as follows:

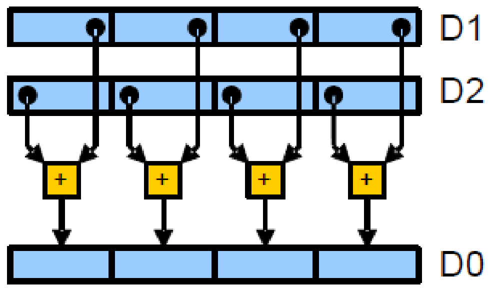
- S<2n> maps to the least significant half of D<n>
- S<2n+1> maps to the most significant half of D<n>
- D<2n> maps to the least significant half of Q<n>
- D<2n+1> maps to the most significant half of Q<n>.

For example, software can access the least significant half of the elements of a vector in Q6 by referring to D12, and the most significant half of the elements by referring to D13.

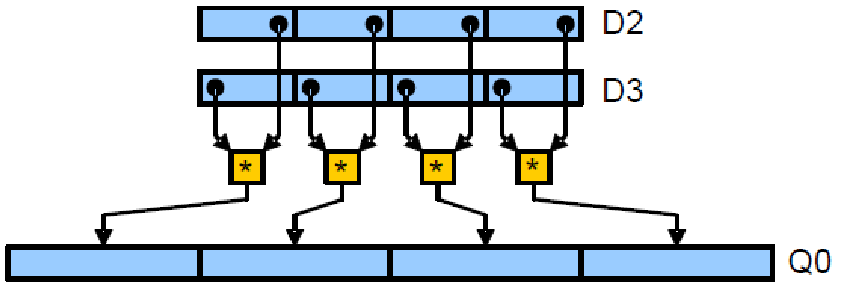
NEON

- Similar to MMX, SSE, and 3DNow! in x86
- Advanced SIMD (Single Instruction Multiple Data)
- It supports 8, 16, 32 and 64-bit integer and single-precision (32-bit) floating point data
  - NEON engine includes a dedicated floating point unit
- Up to 16 operations at the same time
  - 1B x 16 = 16B (= 1 quad word)

`VADD.I16 D0, D1, D2`



`VMULL.S16 Q0, D2, D3;` This instruction will cause four widening multiplies



VFP (Vector Floating Point)

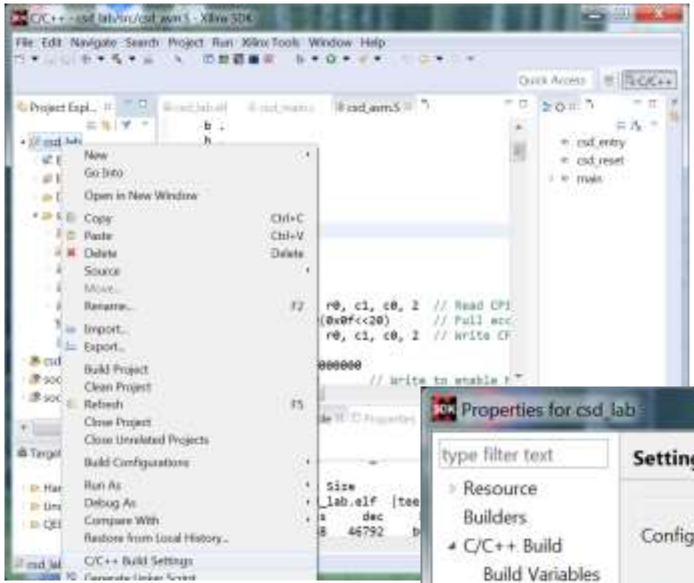
- FPU (Floating Point Unit) coprocessor extension to ARM architecture
- Single-precision and double-precision FP computation
  - Compliant with IEEE 754-1985
- Intended to support execution of short "vector mode" instructions, but operated on "each" vector element sequentially
  - Thus, did not offer the performance of true SIMD
  - This vector mode was thus removed shortly after its introduction, to be replaced with the much more powerful NEON Advanced SIMD

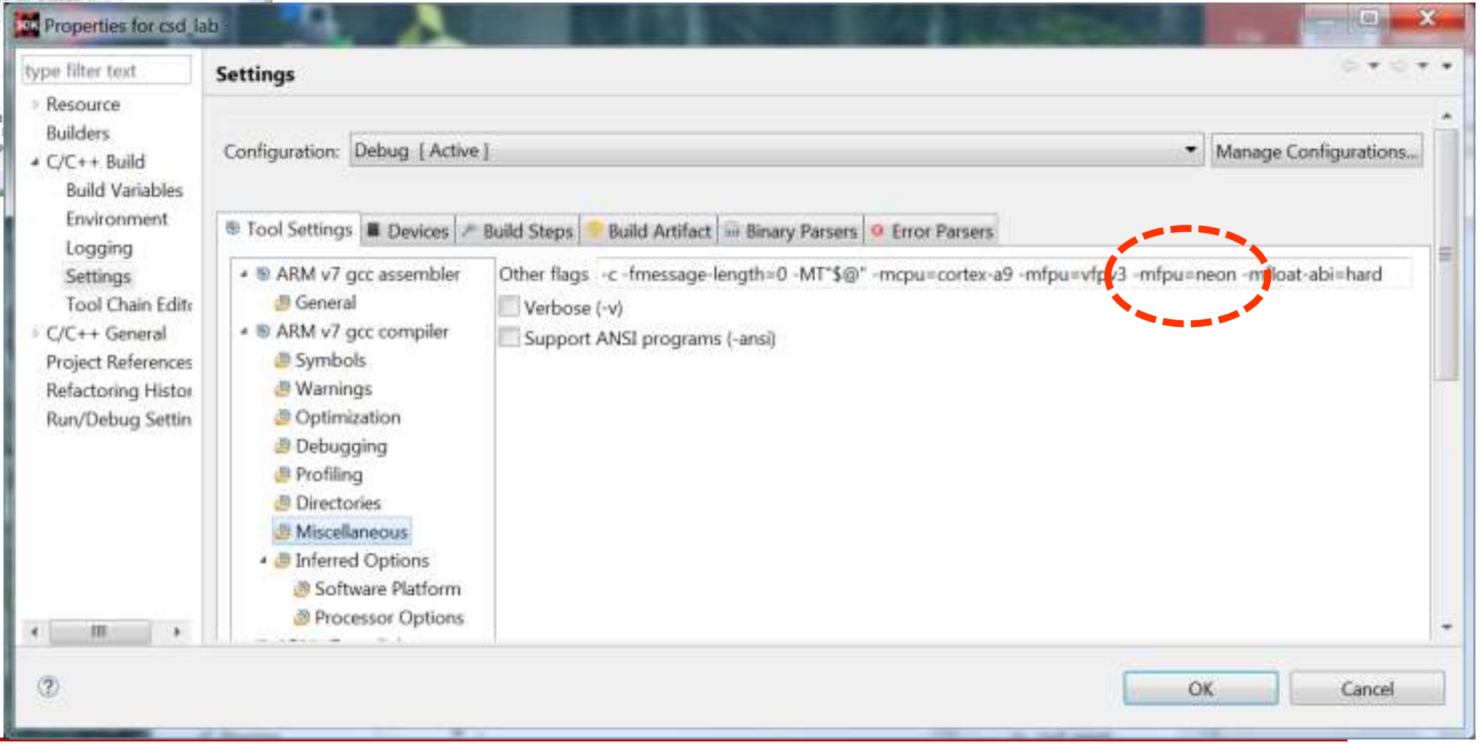
http://en.wikipedia.org/wiki/ARM_architecture

Enabling NEON

- NEON is disabled at reset
- Enabling NEON requires 2 steps
  - Enable access to coprocessors 10 and 11 and allow NEON instructions via CPACR (Coprocessor Access Control Register) – see backup slide
  - Enable NEON and VFP via FPEXC (Floating-point Exception Control Register) – see backup slide

Compiler Option (-mfpu=neon)





Settings

Configuration: Debug [Active]

Other flags: `-c -fmessage-length=0 -MT"$@" -mcpu=cortex-a9 -mfpu=vfpv3 -mfpu=neon -mfloat-abi=hard`

Verbose (-v) ☐

Support ANSI programs (-ansi) ☐

- `-mcpu=cortex-a9`. Specifies the name of the target ARM processor. GCC uses this name to determine what kind of instructions it can issue when generating assembly code.
- `-mfpu=neon`. Specifies which floating-point hardware (or hardware emulation) is available on the target. Because the Zynq-7000 device has an integrated NEON hardware unit, and because you plan to use it to accelerate software, you must specify your intention to the compiler clearly, using the name `'neon'`.

https://www.xilinx.com/support/documentation/application_notes/xapp1206-boost-sw-performance-zynq7soc-w-neon.pdf

Instruction Syntax

Syntax: V{<mod>}<op>{<shape>}{<cond>}{.<dt>}{<dest>}, src1, src2

mod	Instruction Modifiers	Note
Q	**Operation uses saturating arithmetic (ex, VQADD)**	FP: FPSCR.QC
H	**Operation halves the result (ex, VHADD)**	VHADD Q1, Q2, Q3; Q1 = (Q2 + Q3) >> 1 bit
D	**Operation doubles the result (ex, VQDMULL)**	VQDMULL Q1, D2, D3; Q1 = (D2 x D3) x 2
R	**Operation performs rounding (ex, VRHADD)**	VHADD with rounding (always round-to-nearest)

op	Instruction Operation (ex, ADD, MUL, MLA, MAX, SHR…)

shape	Operand Shapers	Typical register shape
(none)		Dd, Dn, Dm Qd, Qn, Qm
L	**Long operation: double the width of both operands (ex, VADDL.I32 Q0, D3, D4; double the width of D3 and D4 before operation)**	Qd, Dn, Dm
W	**Wide operation: double the width of the 2nd operand (ex, VADDW.I32 Q0, Q1, D5; double the width of D5 before operation)**	Qd, Qn, Dm
N	**Narrow operation: half the width of the result (ex, VADDHN.I32 D0, Q1, Q2; half the width of the result)**	Dd, Qn, Qm

- `VADDHN`: Vector Add and Narrow, returning High Half.
- There is no "`VADDN`" instruction, though

Instruction Syntax

Syntax: V{<mod>}<op>{<shape>}{<cond>}{.<dt>}{<dest>}, src1, src2

`.dt`	Data type
Unsigned integer	**.U<size> : U8, U16, U32, U64**
Signed integer	**.S <size> : S8, S16, S32, S64**
Integer of unspecified type	**.I<size> : I8, I16, I32, I64**
Floating point number	**.F<size> : F32**
Polynomial	**.P<size> : P8, P16**
	.<size> : Any element of <size> bits

- Some instructions need to know about the data type
  - Addition instruction needs to know if the source operands are integer or floating-point
  - Saturating instruction needs to know if the source operands are signed, unsigned or floating-point

Examples

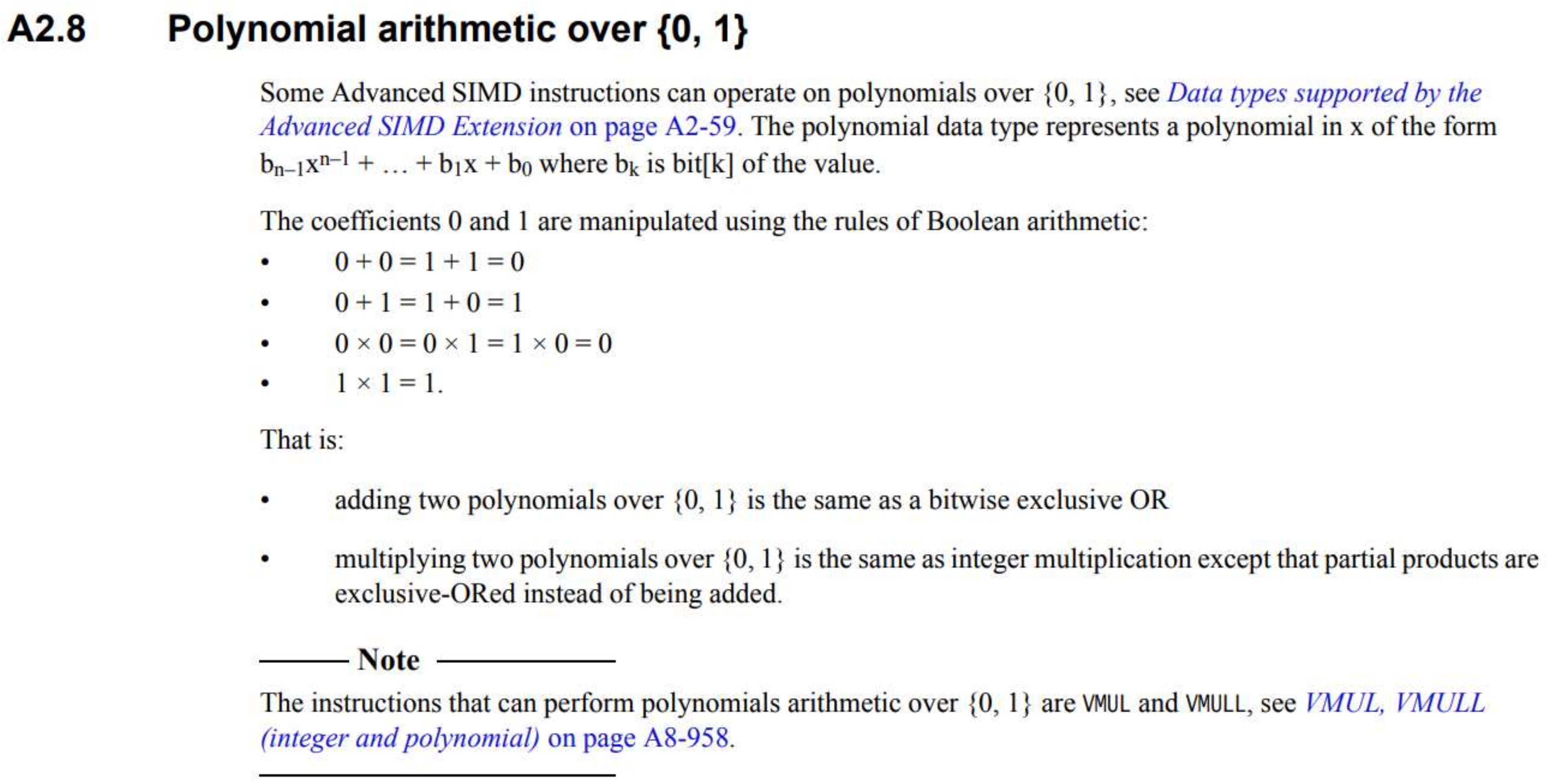
`VADD.I16  D0, D1, D2;`	adds 4 16-bit integer elements
`VADD.F32  Q4, Q7, Q8;`	adds 4 32-bit floating-point elements
`VSUBL.S32  Q8, D1, D5;`	Subtracts 2 32-bit integer elements and each result is stored in 64-bit
`VABS.S16  D0, D1, D2;`	Absolute difference of 4 16-bit integer elements
`VMUL.I8   Q0, Q1, Q2;`	Multiply 16 8-bit integer elements
`VMUL.I16 D1, D7, D4[2];`	Multiply 4 16-bit integer elements by 16-bit scalar

Polynomials (P8, P16)

- Polynomials with single-bit coefficients
  - P8: 8 single-bit coefficients
  - P16: 16 single-bit coefficients
- Cryptography algorithms such as ECC (Elliptic Curve Cryptography) require modulo arithmetic of polynomials with binary coefficients

http://esca.korea.ac.kr/teaching/cose321_CSD/labs/lab_neon/Fast-Software-Polynomial-Multiplication-Using-NEON.pdf
http://esca.korea.ac.kr/teaching/cose321_CSD/labs/lab_neon/2013hpec-SIMD-Acceleration-NEON.pdf

Polynomial Arithmetic over {0,1}



A2.8 Polynomial arithmetic over {0, 1}

Some Advanced SIMD instructions can operate on polynomials over {0, 1}, see *Data types supported by the Advanced SIMD Extension* on page A2-59. The polynomial data type represents a polynomial in x of the form $b_{n-1}x^{n-1} + \ldots + b_1x + b_0$ where b_k is bit[k] of the value.

The coefficients 0 and 1 are manipulated using the rules of Boolean arithmetic:

- $0 + 0 = 1 + 1 = 0$
- $0 + 1 = 1 + 0 = 1$
- $0 \times 0 = 0 \times 1 = 1 \times 0 = 0$
- $1 \times 1 = 1$.

That is:

- adding two polynomials over {0, 1} is the same as a bitwise exclusive OR
- multiplying two polynomials over {0, 1} is the same as integer multiplication except that partial products are exclusive-ORed instead of being added.

Note

The instructions that can perform polynomials arithmetic over {0, 1} are `VMUL` and `VMULL`, see *VMUL, VMULL (integer and polynomial)* on page A8-958.

http://esca.korea.ac.kr/teaching/cose321_CSD/labs/DDI0406C_C_armv7AR_architecture_reference_manual.pdf

Modulo Multiplication of Polynomials with Binary Coefficients

The finite field with 2 elements

The simplest **finite field** is

$$GF(2) = \mathbf{F}_2 = \{0, 1\} = \mathbf{Z}/2$$

It has addition and multiplication $+$ and $\times$ defined to be

$$\begin{array}{cccc} 0+0=0 & 0+1=1 & 1+0=1 & 1+1=0 \\ 0\times 0=0 & 0\times 1=0 & 1\times 0=0 & 1\times 1=1 \end{array}$$

Addition of polynomials with coefficients in $\mathbf{F}_2$ is to add the coefficients of corresponding powers of x, inside $\mathbf{F}_2$. For example,

$$(x^3+x^2+1)+(x^3+x+1)$$

$$=(1+1)\cdot x^3+(1+0)\cdot x^2+(0+1)x+(1+1)$$

$$=x^2+x$$

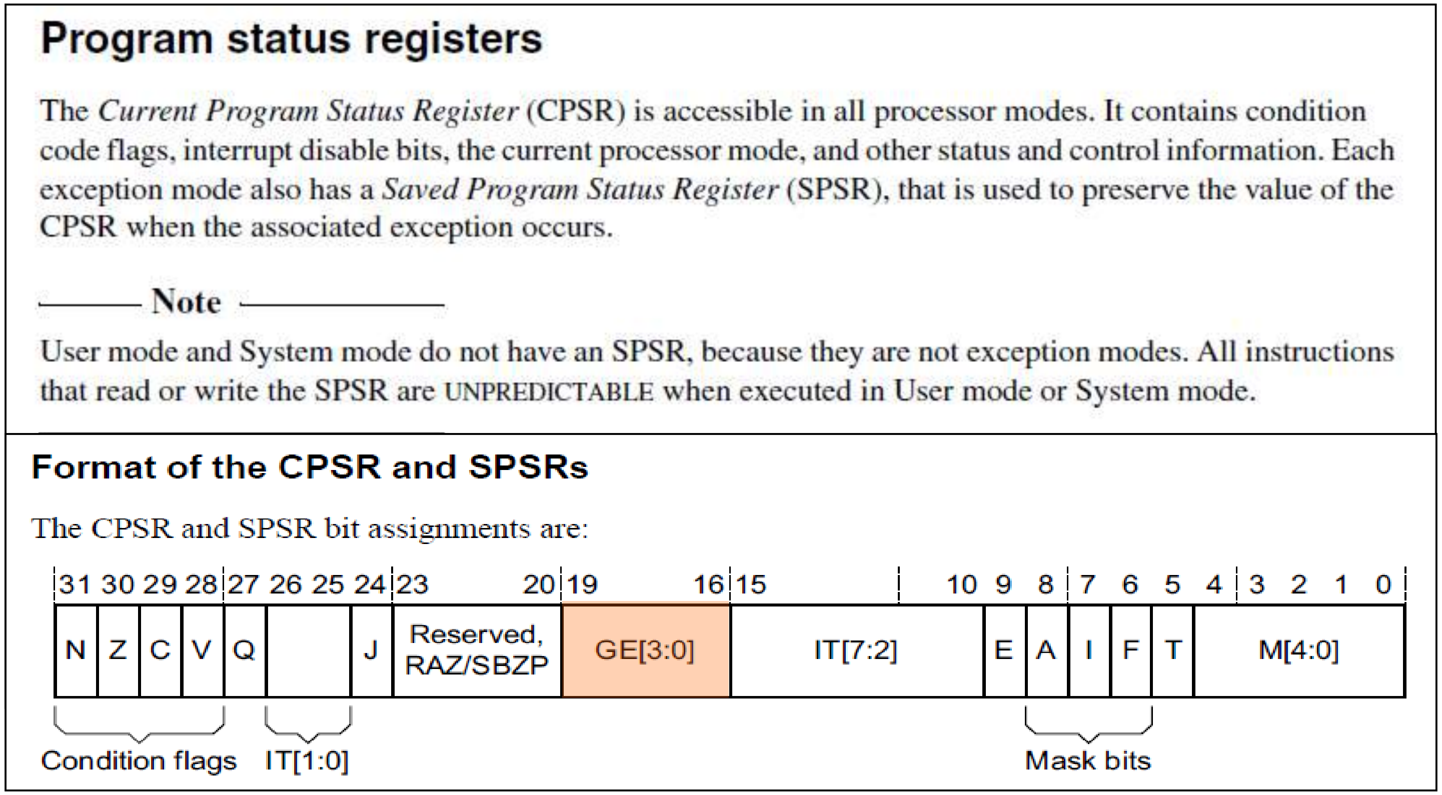
$$\begin{array}{cccccc} & & x^3 & & +x & +1 \\ \times & & & x^2 & +x & +1 \\ \hline & & +x^3 & & +x & +1 \\ & +x^4 & & +x^2 & +x & \\ x^5 & & +x^3 & +x^2 & & \\ \hline x^5 & +x^4 & & & & +1 \end{array}$$

GE Flags

- SIMD (**not** Advanced SIMD such as `vadd, vhadd`) instructions use the GE flags
  - Normal data processing instructions generates only 1 output, vs SIMD instructions generate multiple outputs
  - So, NZCV in CPSR can't be used in SIMD. Instead, GE[3:0] flags were added to CPSR
    - GE stands for greater than or equal
  - GE[3:0] is there for parallel addition and subtraction instructions
    - `add8` / `sub8` sets GE[3:0] bits in CPSR individually
    - `add16` / `sub16` sets GE[3] and GE[2] together, and GE[1] and GE[0] together

CPSR

- Current Program Status Register (CPSR) is accessible in all modes
- Contains all condition flags, interrupt disable bits, the current processor mode



Program status registers

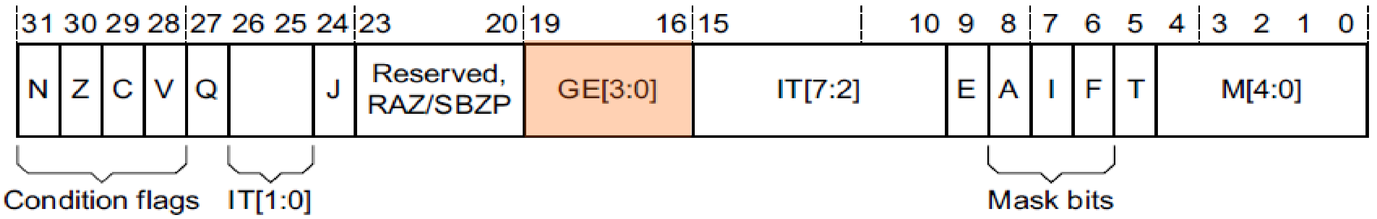
The *Current Program Status Register* (CPSR) is accessible in all processor modes. It contains condition code flags, interrupt disable bits, the current processor mode, and other status and control information. Each exception mode also has a *Saved Program Status Register* (SPSR), that is used to preserve the value of the CPSR when the associated exception occurs.

Note

User mode and System mode do not have an SPSR, because they are not exception modes. All instructions that read or write the SPSR are UNPREDICTABLE when executed in User mode or System mode.

Format of the CPSR and SPSRs

The CPSR and SPSR bit assignments are:



Korea Univ

Usage Example

- Byte Selection: selects each byte of the result from either 1st operand or 2nd operand, depending on GE[3:0]

```
usub8   r2, r1, r0    // set GE flags

sel     r2, r1, r0    //  set each byte in r2 with
                      //  the greater or equal of the
                      //  corresponding bytes in r1 and r0

sel     r3, r0, r1    // set each byte in r3 with the minimum of
                      // the corresponding bytes in r1 and r0
```

SEL Instruction

A8.8.166 SEL

Select Bytes selects each byte of its result from either its first operand or its second operand, according to the values of the GE flags.

Encoding T1 ARMv6T2, ARMv7

`SEL<c> <Rd>, <Rn>, <Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	1	0	1	0	1	0	Rn				1	1	1	1	Rd				1	0	0	0	Rm			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Encoding A1 ARMv6*, ARMv7

`SEL<c> <Rd>, <Rn>, <Rm>`

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
cond				0	1	1	0	1	0	0	0	Rn				Rd				(1)	(1)	(1)	(1)	1	0	1	1	Rm			

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

Operation

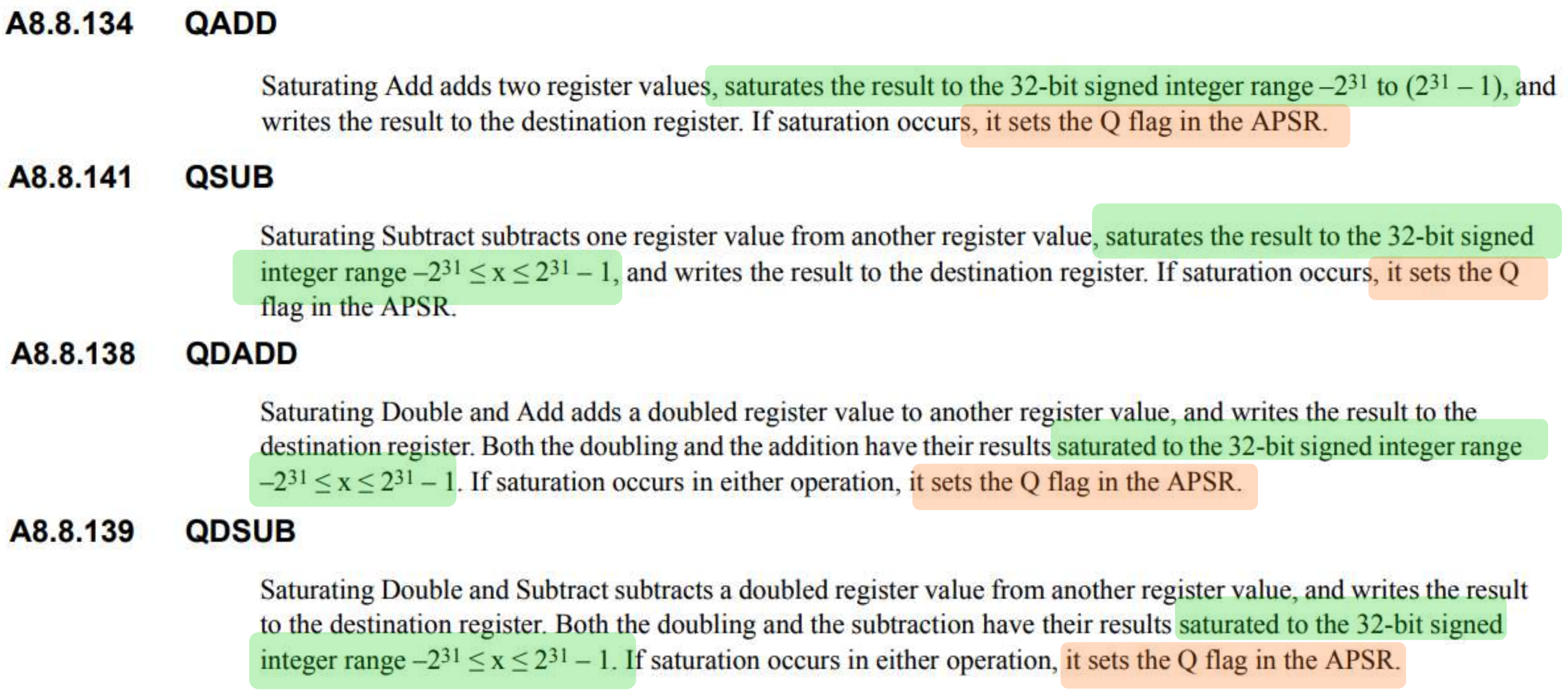
```
if ConditionPassed() then
    EncodingSpecificOperations();
    R[d]<7:0>   = if APSR.GE<0> == '1' then R[n]<7:0>   else R[m]<7:0>;
    R[d]<15:8>  = if APSR.GE<1> == '1' then R[n]<15:8>  else R[m]<15:8>;
    R[d]<23:16> = if APSR.GE<2> == '1' then R[n]<23:16> else R[m]<23:16>;
    R[d]<31:24> = if APSR.GE<3> == '1' then R[n]<31:24> else R[m]<31:24>;
```

Q Flag

- Q is for saturated math instructions
  - **Without** saturation
    - Adding 1 to 0x7FFF_FFFF causes a transition from a positive number to a negative number
    - Subtracting 1 from 0x8000_0000 causes a transition from a negative number to a positive number
  - **With** saturation
    - Adding 1 to 0x7FFF_FFFF remains to be 0x7FFF_FFFF
    - Subtracting 1 from 0x8000_0000 remains to be 0x8000_0000

- Saturation is required by DSP (Digital Signal Processing) applications

Examples



A8.8.134 QADD

Saturating Add adds two register values, saturates the result to the 32-bit signed integer range -2^{31} to $(2^{31} - 1)$, and writes the result to the destination register. If saturation occurs, it sets the Q flag in the APSR.

A8.8.141 QSUB

Saturating Subtract subtracts one register value from another register value, saturates the result to the 32-bit signed integer range $-2^{31} \leq x \leq 2^{31} - 1$, and writes the result to the destination register. If saturation occurs, it sets the Q flag in the APSR.

A8.8.138 QDADD

Saturating Double and Add adds a doubled register value to another register value, and writes the result to the destination register. Both the doubling and the addition have their results saturated to the 32-bit signed integer range $-2^{31} \leq x \leq 2^{31} - 1$. If saturation occurs in either operation, it sets the Q flag in the APSR.

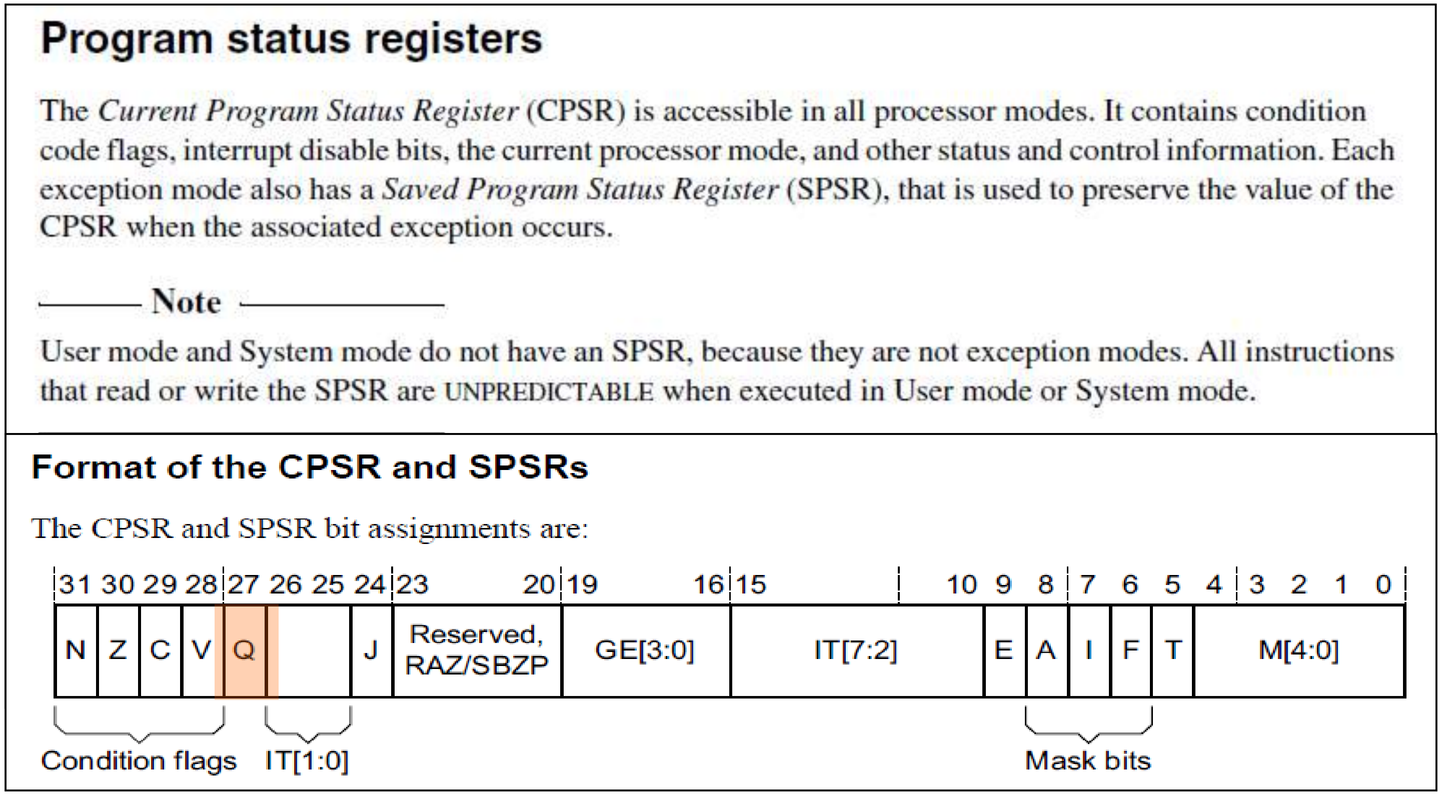
A8.8.139 QDSUB

Saturating Double and Subtract subtracts a doubled register value from another register value, and writes the result to the destination register. Both the doubling and the subtraction have their results saturated to the 32-bit signed integer range $-2^{31} \leq x \leq 2^{31} - 1$. If saturation occurs in either operation, it sets the Q flag in the APSR.

```
qadd  r2, r1, r0
qsub  r2, r1, r0
qdadd r2, r1, r0
qdsub r2, r1, r0
```

CPSR

- Current Program Status Register (CPSR) is accessible in all modes
- Contains all condition flags, interrupt disable bits, the current processor mode



Program status registers

The *Current Program Status Register* (CPSR) is accessible in all processor modes. It contains condition code flags, interrupt disable bits, the current processor mode, and other status and control information. Each exception mode also has a *Saved Program Status Register* (SPSR), that is used to preserve the value of the CPSR when the associated exception occurs.

Note

User mode and System mode do not have an SPSR, because they are not exception modes. All instructions that read or write the SPSR are UNPREDICTABLE when executed in User mode or System mode.

Format of the CPSR and SPSRs

The CPSR and SPSR bit assignments are:

NEON Types in C

```
#include <arm_neon.h>
```

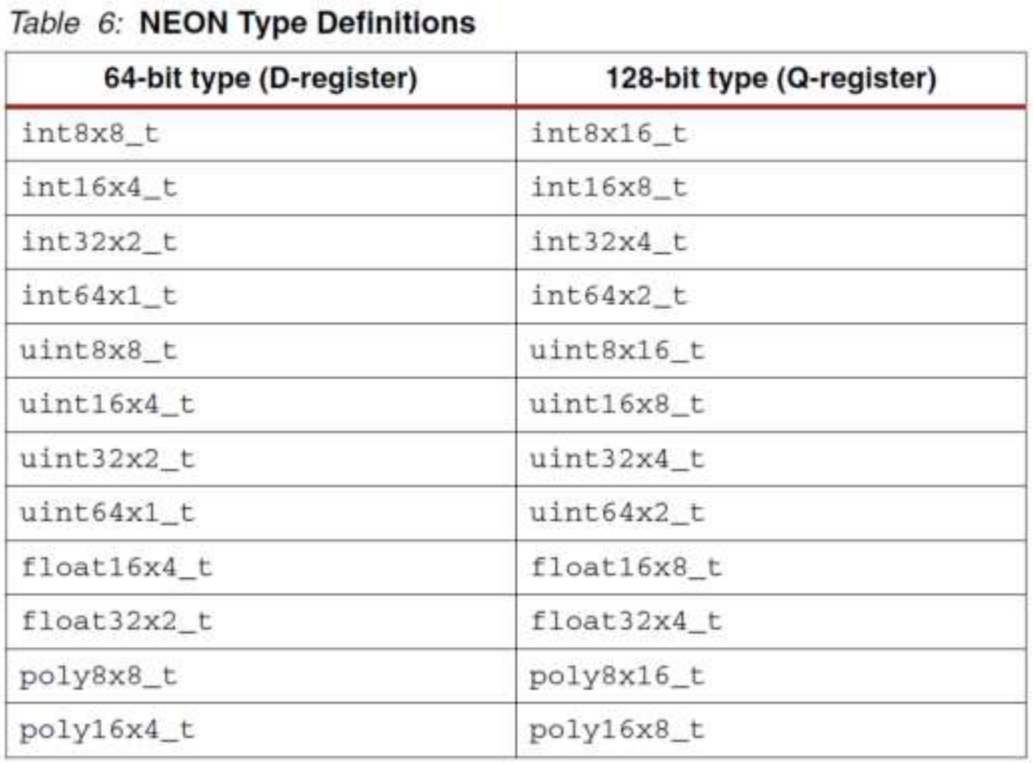


Table 6: **NEON Type Definitions**

64-bit type (D-register)	128-bit type (Q-register)
int8x8_t	int8x16_t
int16x4_t	int16x8_t
int32x2_t	int32x4_t
int64x1_t	int64x2_t
uint8x8_t	uint8x16_t
uint16x4_t	uint16x8_t
uint32x2_t	uint32x4_t
uint64x1_t	uint64x2_t
float16x4_t	float16x8_t
float32x2_t	float32x4_t
poly8x8_t	poly8x16_t
poly16x4_t	poly16x8_t

Boost Software Performance on Zynq-7000 AP SoC with NEON

Backup Slides

IEEE Standard 758: Rounding

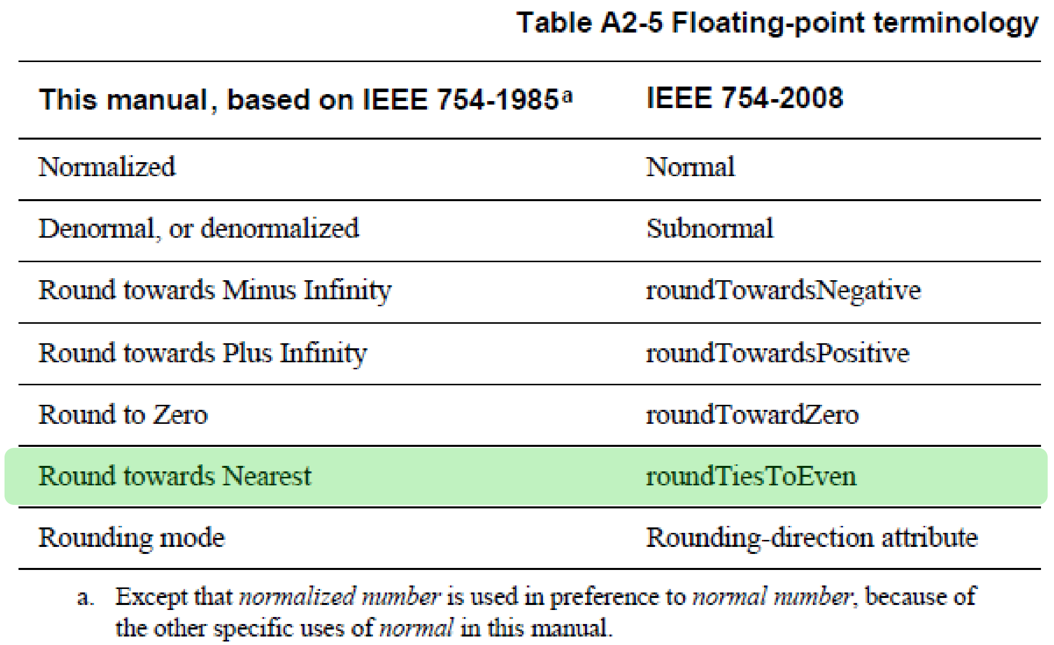
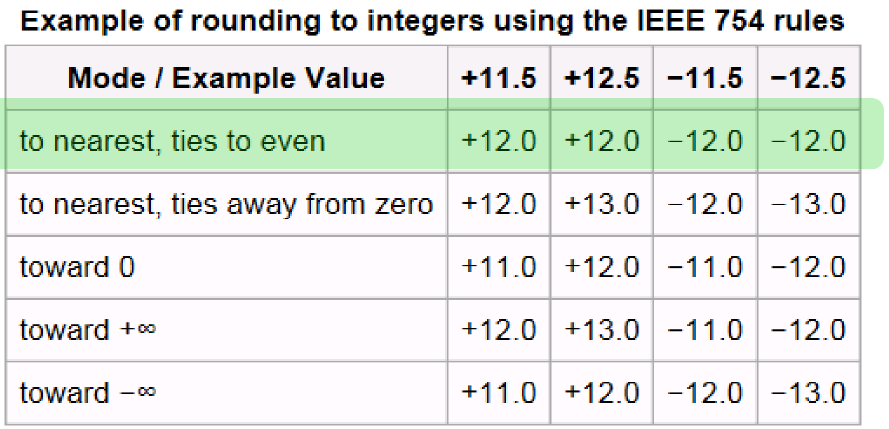


Table A2-5 Floating-point terminology

This manual, based on IEEE 754-1985[a]	IEEE 754-2008
Normalized	Normal
Denormal, or denormalized	Subnormal
Round towards Minus Infinity	roundTowardsNegative
Round towards Plus Infinity	roundTowardsPositive
Round to Zero	roundTowardZero
Round towards Nearest	roundTiesToEven
Rounding mode	Rounding-direction attribute

a. Except that *normalized number* is used in preference to *normal number*, because of the other specific uses of *normal* in this manual.

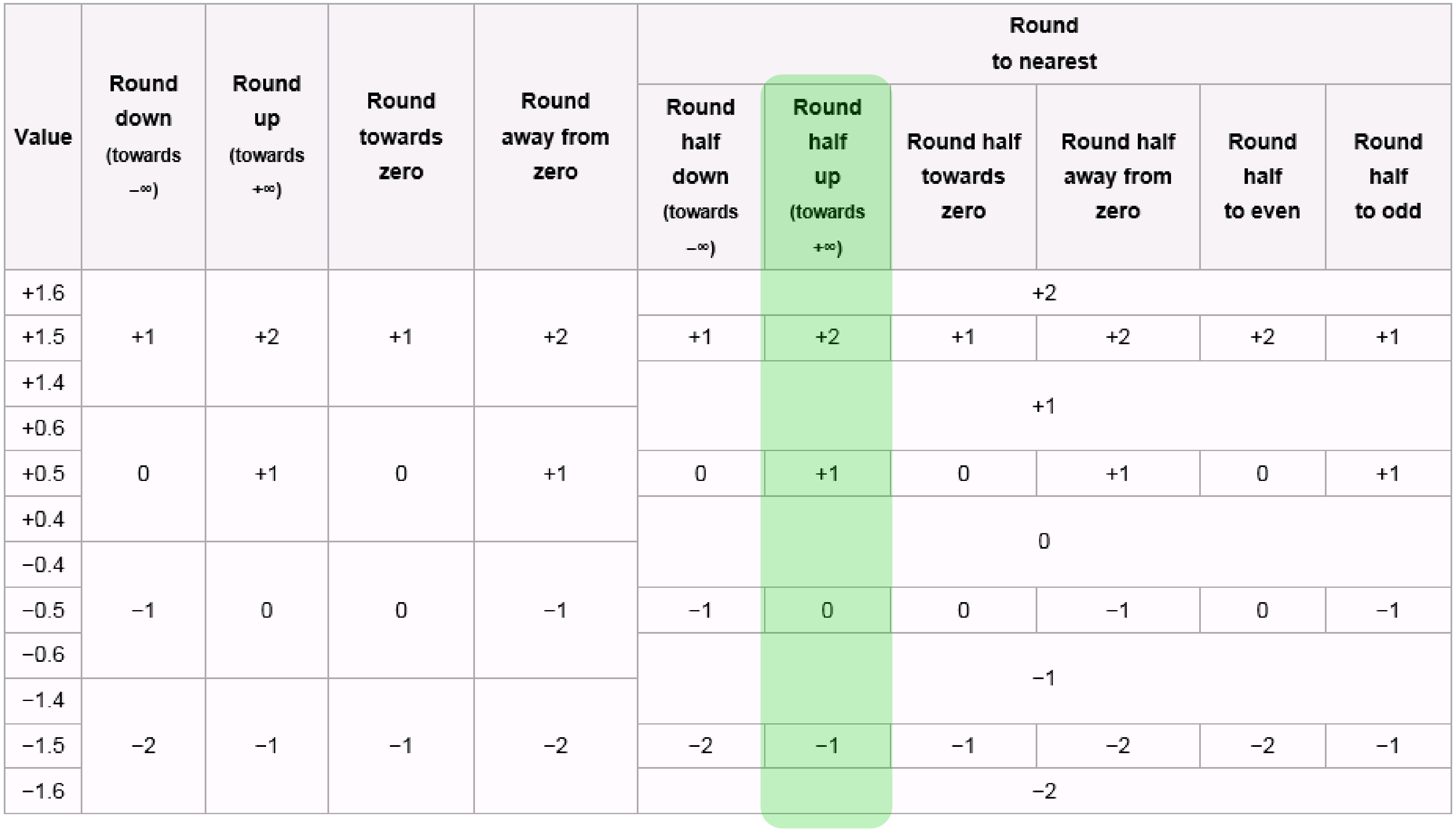


Example of rounding to integers using the IEEE 754 rules

Mode / Example Value	+11.5	+12.5	−11.5	−12.5
to nearest, ties to even	+12.0	+12.0	−12.0	−12.0
to nearest, ties away from zero	+12.0	+13.0	−12.0	−13.0
toward 0	+11.0	+12.0	−11.0	−12.0
toward +∞	+12.0	+13.0	−11.0	−12.0
toward −∞	+11.0	+12.0	−12.0	−13.0

- IEEE 754 default rounding mode: Round half to even

Rounding: Tie-Breakers



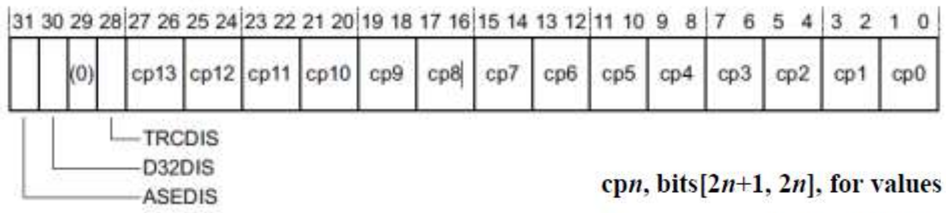
Value	Round down (towards −∞)	Round up (towards +∞)	Round towards zero	Round away from zero	Round to nearest					
					Round half down (towards −∞)	Round half up (towards +∞)	Round half towards zero	Round half away from zero	Round half to even	Round half to odd
+1.6	+1	+2	+1	+2	+2	+2	+2	+2	+2	+2
+1.5	+1	+2	+1	+2	+1	+2	+1	+2	+2	+1
+1.4	+1	+2	+1	+2	+1	+1	+1	+1	+1	+1
+0.6	0	+1	0	+1	+1	+1	+1	+1	+1	+1
+0.5	0	+1	0	+1	0	+1	0	+1	0	+1
+0.4	0	+1	0	+1	0	0	0	0	0	0
−0.4	−1	0	0	−1	0	0	0	0	0	0
−0.5	−1	0	0	−1	−1	0	0	−1	0	−1
−0.6	−1	0	0	−1	−1	−1	−1	−1	−1	−1
−1.4	−2	−1	−1	−2	−1	−1	−1	−1	−1	−1
−1.5	−2	−1	−1	−2	−2	−1	−1	−2	−2	−1
−1.6	−2	−1	−1	−2	−2	−2	−2	−2	−2	−2

I believe that Round-half-up is implemented by default in Cortex-A9

CPACR (Coprocessor Access Control Register)

- CPACR controls access to coprocessors CP0 to CP13 from PL0 and PL2

The CPACR bit assignments are:



cp*n*, bits[2*n*+1, 2*n*], for values of *n* from 0 to 13

Defines the access rights for coprocessor n, for accesses from PL1 and PL0. The possible values of the field are:

`0b00`	Access denied. Any attempt to access the coprocessor generates an Undefined Instruction exception.
`0b01`	Access at PL1 only. Any attempt to access the coprocessor from software executing at PL0 generates an Undefined Instruction exception.
`0b10`	Reserved. The effect of this value is UNPREDICTABLE.
`0b11`	Full access. The meaning of full access is defined by the appropriate coprocessor.

Accessing the CPACR

To access the CPACR, software reads or writes the CP15 registers with <opc1> set to 0, <CRn> set to c1, <CRm> set to c0, and <opc2> set to 2. For example:

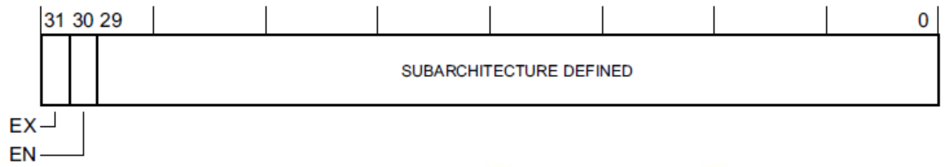
```
MRC p15, 0, <Rt>, c1, c0, 2    ; Read CPACR into Rt
MCR p15, 0, <Rt>, c1, c0, 2    ; Write Rt to CPACR
```

Normally, software uses a read, modify, write sequence to update the CPACR, to avoid unwanted changes to the access settings for other coprocessors.

FPEXC (Floating-Point Exception Control Register)

- FPEXC provides a global enable for the Advanced SIMD and Floating-point (VFP) Extensions

The FPEXC bit assignments are:



EN, bit[30]	Enable bit. A global enable for the Advanced SIMD and Floating-point Extensions:	
	0	The Advanced SIMD and Floating-point Extensions are disabled. For details of how the system operates when EN == 0 see *Enabling Advanced SIMD and floating-point support* on page B1-1228.
	1	The Advanced SIMD and Floating-point Extensions are enabled and operate normally.
	This bit is always a normal read/write bit. It has a reset value of 0.	

Accessing the FPEXC register

Software reads or writes the FPEXC register using the VMRS and VMSR instructions. For more information, see *VMRS* on page A8-954 and *VMSR* on page A8-956. For example:

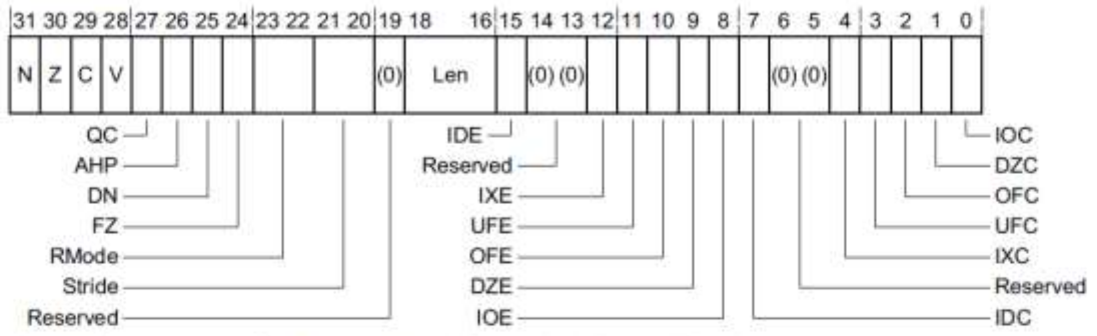
```
VMRS <Rt>, FPEXC      ; Read  Floating-point Exception Control Register
VMSR FPEXC, <Rt>      ; Write Floating-point Exception Control Register
```

Writes to the FPEXC can have side-effects on various aspects of processor operation. All of these side-effects are synchronous to the FPEXC write. This means they are guaranteed not to be visible to earlier instructions in the execution stream, and they are guaranteed to be visible to later instructions in the execution stream.

FPSCR (Floating-Point Status and Control Register)

- Provides floating-point system status information and control

The FPSCR bit assignments are:



See the field descriptions for implementation differences in different VFP versions

Bits[31:28]	Condition flags. These are updated by floating-point comparison operations, as shown in *Effect of a Floating-point comparison on the condition flags* on page A2-80.

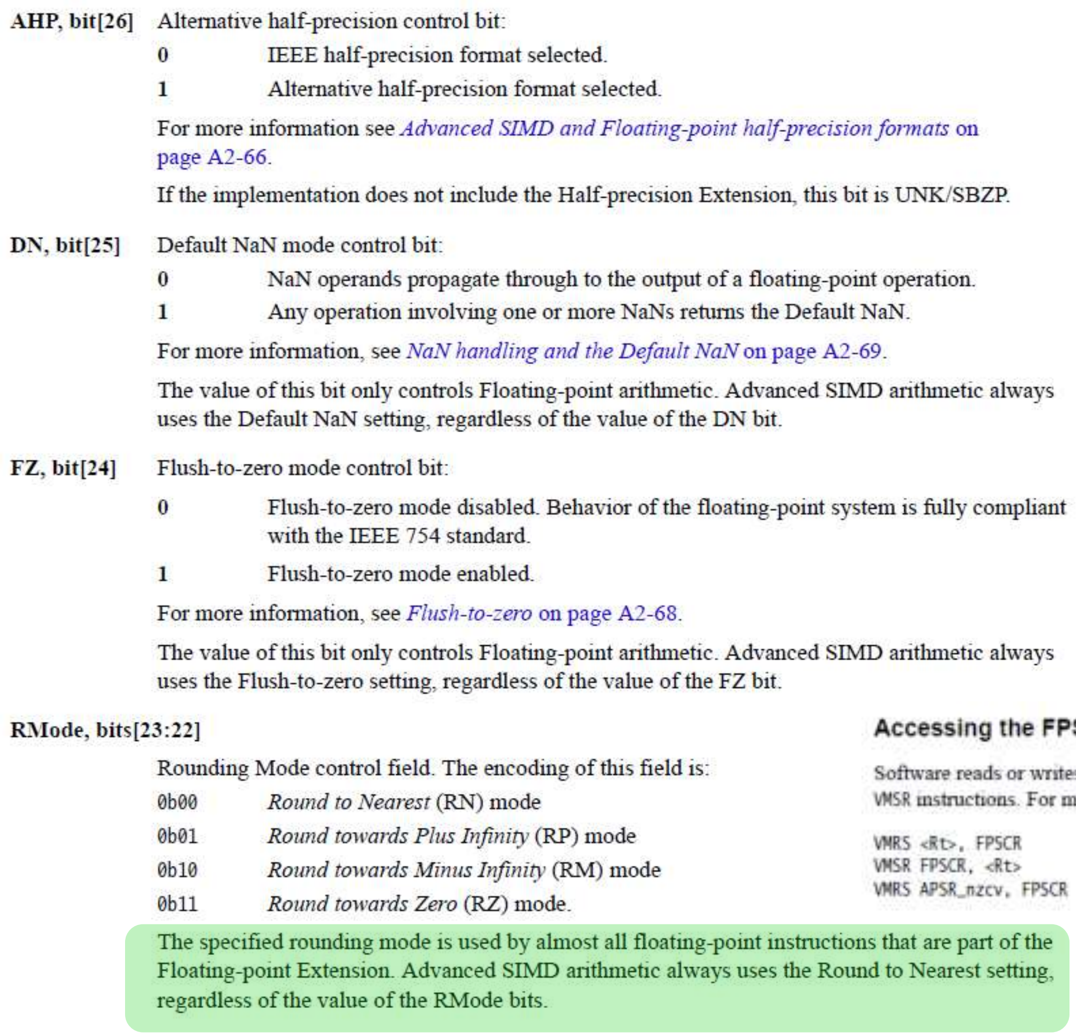
N, bit[31]	Negative condition flag.
Z, bit[30]	Zero condition flag.
C, bit[29]	Carry condition flag.
V, bit[28]	Overflow condition flag.

Note

Advanced SIMD operations never update these bits.

QC, bit[27]	Cumulative saturation bit, Advanced SIMD only. This bit is set to 1 to indicate that an Advanced SIMD integer operation has saturated since 0 was last written to this bit. For details of saturation, see *Pseudocode details of saturation* on page A2-44.

FPSCR (Floating-Point Status and Control Register)



AHP, bit[26] Alternative half-precision control bit:

0	IEEE half-precision format selected.
1	Alternative half-precision format selected.

For more information see *Advanced SIMD and Floating-point half-precision formats* on page A2-66.

If the implementation does not include the Half-precision Extension, this bit is UNK/SBZP.

DN, bit[25] Default NaN mode control bit:

0	NaN operands propagate through to the output of a floating-point operation.
1	Any operation involving one or more NaNs returns the Default NaN.

For more information, see *NaN handling and the Default NaN* on page A2-69.

The value of this bit only controls Floating-point arithmetic. Advanced SIMD arithmetic always uses the Default NaN setting, regardless of the value of the DN bit.

FZ, bit[24] Flush-to-zero mode control bit:

0	Flush-to-zero mode disabled. Behavior of the floating-point system is fully compliant with the IEEE 754 standard.
1	Flush-to-zero mode enabled.

For more information, see *Flush-to-zero* on page A2-68.

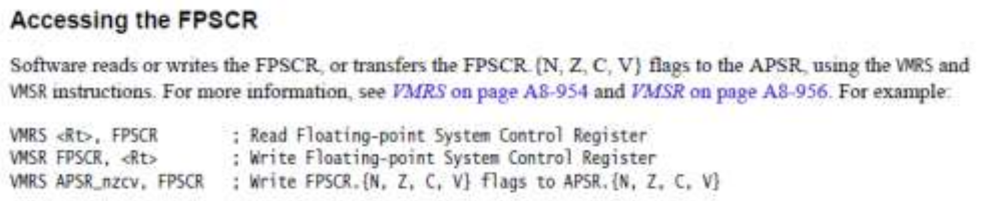
The value of this bit only controls Floating-point arithmetic. Advanced SIMD arithmetic always uses the Flush-to-zero setting, regardless of the value of the FZ bit.

RMode, bits[23:22]

Rounding Mode control field. The encoding of this field is:

`0b00`	*Round to Nearest* (RN) mode
`0b01`	*Round towards Plus Infinity* (RP) mode
`0b10`	*Round towards Minus Infinity* (RM) mode
`0b11`	*Round towards Zero* (RZ) mode.

The specified rounding mode is used by almost all floating-point instructions that are part of the Floating-point Extension. Advanced SIMD arithmetic always uses the Round to Nearest setting, regardless of the value of the RMode bits.



Accessing the FPSCR

Software reads or writes the FPSCR, or transfers the FPSCR.{N, Z, C, V} flags to the APSR, using the VMRS and VMSR instructions. For more information, see *VMRS* on page A8-954 and *VMSR* on page A8-956. For example:

```
VMRS <Rt>, FPSCR       ; Read Floating-point System Control Register
VMSR FPSCR, <Rt>       ; Write Floating-point System Control Register
VMRS APSR_nzcv, FPSCR  ; Write FPSCR.{N, Z, C, V} flags to APSR.{N, Z, C, V}
```

VMRS Instruction

B9.3.21 VMRS

Move to ARM core register from Advanced SIMD and Floating-point Extension System Register moves the value of an extension system register to an ARM core register. When the specified Floating-point Extension System Register is the FPSCR, a form of the instruction transfers the FPSCR.{N, Z, C, V} condition flags to the APSR.{N, Z, C, V} condition flags.

Depending on settings in the CPACR, NSACR, HCPTR, and FPEXC registers, and the security state and mode in which the instruction is executed, an attempt to execute a VMRS instruction might be UNDEFINED, or trapped to Hyp mode. *Summary of general controls of CP10 and CP11 functionality* on page B1-1231 and *Summary of access controls for Advanced SIMD functionality* on page B1-1233 summarize these controls.

When these settings permit the execution of floating-point and Advanced SIMD instructions, if the specified Floating-point Extension System Register is not the FPSCR, the instruction is UNDEFINED if executed in User mode.

Assembler syntax

```
VMRS{<c>}{<q>}  <Rt>, <spec_reg>
```

where:

`<c>`, `<q>` See *Standard assembler syntax fields* on page A8-287.

`<Rt>` The destination ARM core register. This register can be R0-R14.

If `<spec_reg>` is `FPSCR`, it is also permitted to be `APSR_nzcv`, encoded as Rt = '1111'. This instruction transfers the FPSCR.{N, Z, C, V} condition flags to the APSR.{N, Z, C, V} condition flags.

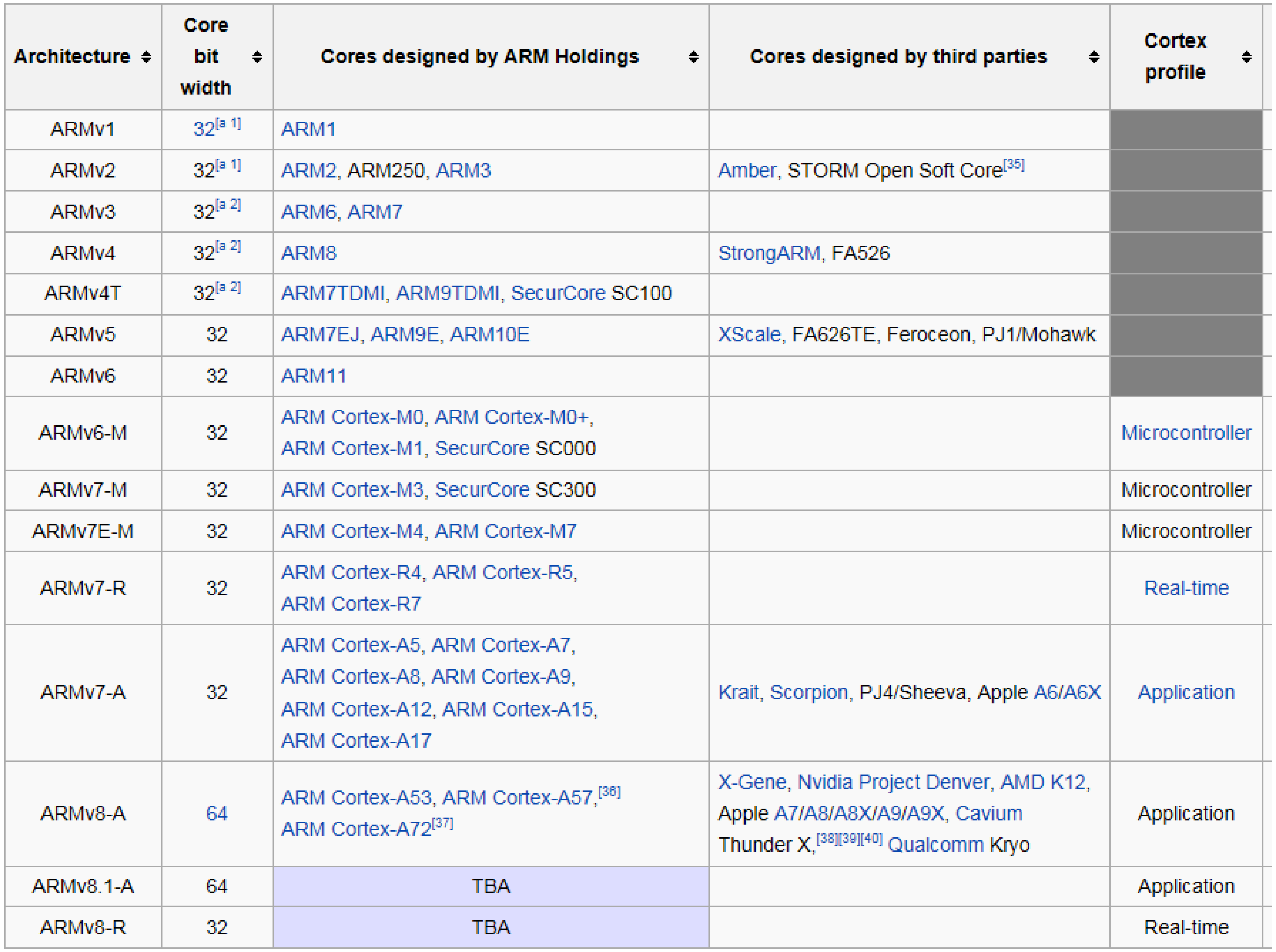
`<spec_reg>` Is one of:

FPSID	reg = '0000'
FPSCR	reg = '0001'
MVFR1	reg = '0110'
MVFR0	reg = '0111'
FPEXC	reg = '1000'.

If the Common VFP subarchitecture is implemented, see *Subarchitecture additions to the Floating-point Extension system registers* on page D6-2441 for additional values of `<spec_reg>`.

The pre-UAL instruction `FMSTAT` is equivalent to `VMRS APSR_nzcv, FPSCR`.

ARM Architecture

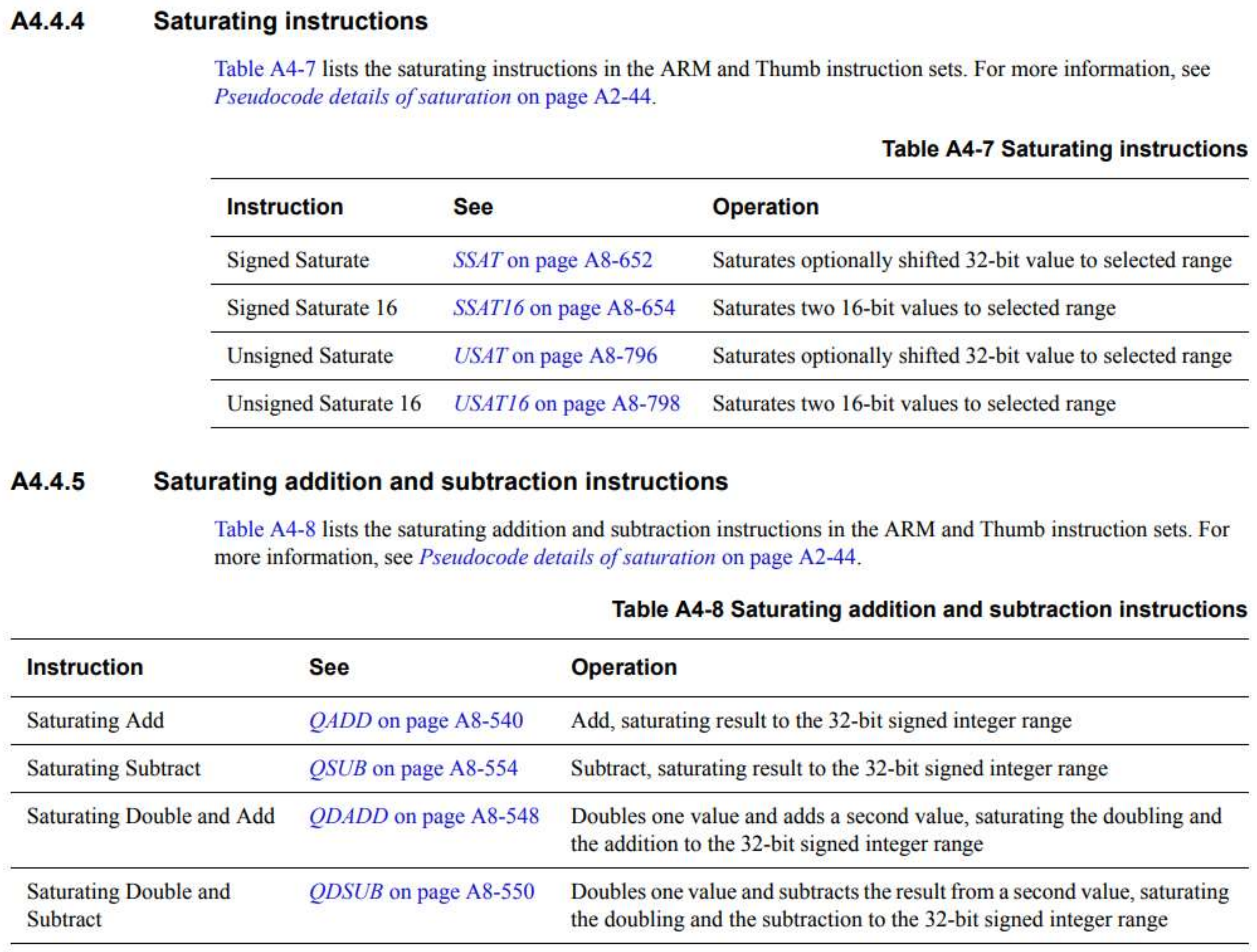


Architecture	Core bit width	Cores designed by ARM Holdings	Cores designed by third parties	Cortex profile
ARMv1	32[a 1]	ARM1		
ARMv2	32[a 1]	ARM2, ARM250, ARM3	Amber, STORM Open Soft Core[35]	
ARMv3	32[a 2]	ARM6, ARM7		
ARMv4	32[a 2]	ARM8	StrongARM, FA526	
ARMv4T	32[a 2]	ARM7TDMI, ARM9TDMI, SecurCore SC100		
ARMv5	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		Microcontroller
ARMv7-M	32	ARM Cortex-M3, SecurCore SC300		Microcontroller
ARMv7E-M	32	ARM Cortex-M4, ARM Cortex-M7		Microcontroller
ARMv7-R	32	ARM Cortex-R4, ARM Cortex-R5, ARM Cortex-R7		Real-time
ARMv7-A	32	ARM Cortex-A5, ARM Cortex-A7, ARM Cortex-A8, ARM Cortex-A9, ARM Cortex-A12, ARM Cortex-A15, ARM Cortex-A17	Krait, Scorpion, PJ4/Sheeva, Apple A6/A6X	Application
ARMv8-A	64	ARM Cortex-A53, ARM Cortex-A57,[36] ARM Cortex-A72[37]	X-Gene, Nvidia Project Denver, AMD K12, Apple A7/A8/A8X/A9/A9X, Cavium Thunder X,[38][39][40] Qualcomm Kryo	Application
ARMv8.1-A	64	TBA		Application
ARMv8-R	32	TBA		Real-time

https://en.wikipedia.org/wiki/ARM_architecture

Saturating Instructions

CPSR'Q flag is set if saturation occurs after the execution of the following inst.



A4.4.4 Saturating instructions

Table A4-7 lists the saturating instructions in the ARM and Thumb instruction sets. For more information, see *Pseudocode details of saturation* on page A2-44.

Table A4-7 Saturating instructions

Instruction	See	Operation
Signed Saturate	*SSAT* on page A8-652	Saturates optionally shifted 32-bit value to selected range
Signed Saturate 16	*SSAT16* on page A8-654	Saturates two 16-bit values to selected range
Unsigned Saturate	*USAT* on page A8-796	Saturates optionally shifted 32-bit value to selected range
Unsigned Saturate 16	*USAT16* on page A8-798	Saturates two 16-bit values to selected range

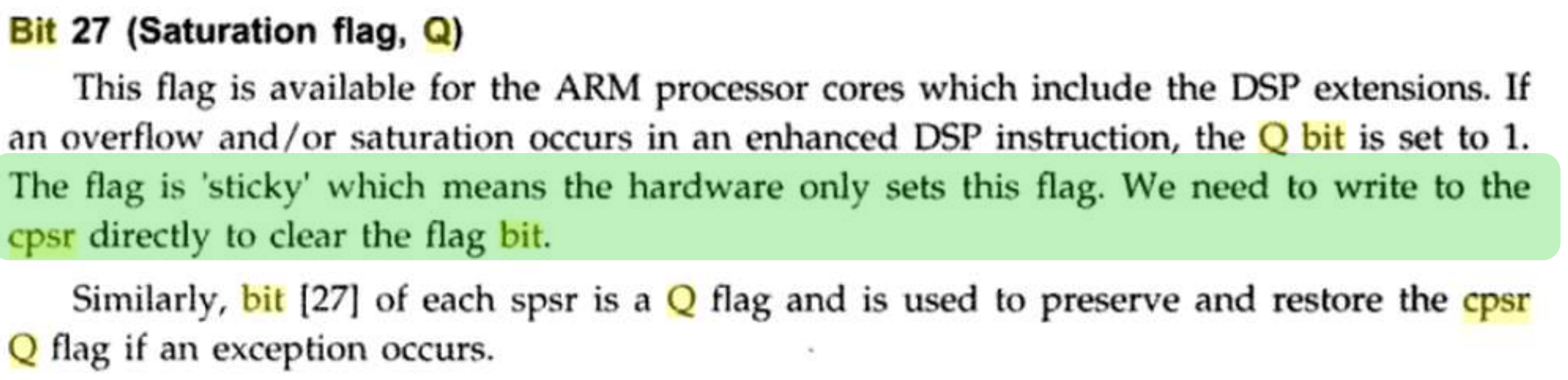
A4.4.5 Saturating addition and subtraction instructions

Table A4-8 lists the saturating addition and subtraction instructions in the ARM and Thumb instruction sets. For more information, see *Pseudocode details of saturation* on page A2-44.

Table A4-8 Saturating addition and subtraction instructions

Instruction	See	Operation
Saturating Add	*QADD* on page A8-540	Add, saturating result to the 32-bit signed integer range
Saturating Subtract	*QSUB* on page A8-554	Subtract, saturating result to the 32-bit signed integer range
Saturating Double and Add	*QDADD* on page A8-548	Doubles one value and adds a second value, saturating the doubling and the addition to the 32-bit signed integer range
Saturating Double and Subtract	*QDSUB* on page A8-550	Doubles one value and subtracts the result from a second value, saturating the doubling and the subtraction to the 32-bit signed integer range

To Clear Q bit in CPSR …



Bit 27 (Saturation flag, Q)

This flag is available for the ARM processor cores which include the DSP extensions. If an overflow and/or saturation occurs in an enhanced DSP instruction, the Q bit is set to 1. The flag is 'sticky' which means the hardware only sets this flag. We need to write to the cpsr directly to clear the flag bit.

Similarly, bit [27] of each spsr is a Q flag and is used to preserve and restore the cpsr Q flag if an exception occurs.

https://books.google.co.kr/books?id=PUZ3dtjb12kC&pg=SA17-PA22&lpg=SA17-PA22&dq=CPSR+Q+bit&source=bl&ots=PLDSzeOHL2&sig=ANiSrXXClYTHl8dBmzZ_zazbW1w&hl=ko&sa=X&ved=0ahUKEwiq2tbDra_aAhUGf7wKHcMECrcQ6AEIYTAG#v=onepage&q=CPSR%20Q%20bit&f=false

Polynomials over GF(2)

Polynomials over $GF(2)$

Basic algebra with polynomials having coefficients in the finite field $\mathbf{F}_2$ is in almost exactly the same as polynomials having real or complex coefficients.

Addition of polynomials with coefficients in $\mathbf{F}_2$ is to add the coefficients of corresponding powers of x, inside $\mathbf{F}_2$. For example,

$$(x^3 + x^2 + 1) + (x^3 + x + 1)$$

$$= (1+1) \cdot x^3 + (1+0) \cdot x^2 + (0+1)x + (1+1)$$

$$= x^2 + x$$

The finite field with 2 elements

The simplest **finite field** is

$$GF(2) = \mathbf{F}_2 = \{0, 1\} = \mathbf{Z}/2$$

It has addition and multiplication $+$ and $\times$ defined to be

$$\begin{array}{cccc} 0+0=0 & 0+1=1 & 1+0=1 & 1+1=0 \\ 0\times 0=0 & 0\times 1=0 & 1\times 0=0 & 1\times 1=1 \end{array}$$

$$\begin{array}{lcccccc}
 & & & x^3 & & +x & +1 \\
\times & & & & x^2 & +x & +1 \\
\hline
 & & & +x^3 & & +x & +1 \\
 & & +x^4 & & +x^2 & +x & \\
 & x^5 & & +x^3 & +x^2 & & \\
\hline
 & x^5 & +x^4 & & & & +1
\end{array}$$