Chapter

3

Cortex-M4 Architecture and ASM Programming

Introduction

In this chapter programming the Cortex-M4 in assembly and C will be introduced. Preference will be given to explaining code development for the Cypress FM4 S6E2CC, STM32F4 Discovery, and LPC4088 Quick Start. The basis for the material presented in this chapter is the course notes from the ARM LiB program[1].

Overview

- Cortex-M4 Memory Map
  - Cortex-M4 Memory Map
  - Bit-band Operations
  - Cortex-M4 Program Image and Endianness
- ARM Cortex-M4 Processor Instruction Set
  - ARM and Thumb Instruction Set
  - Cortex-M4 Instruction Set

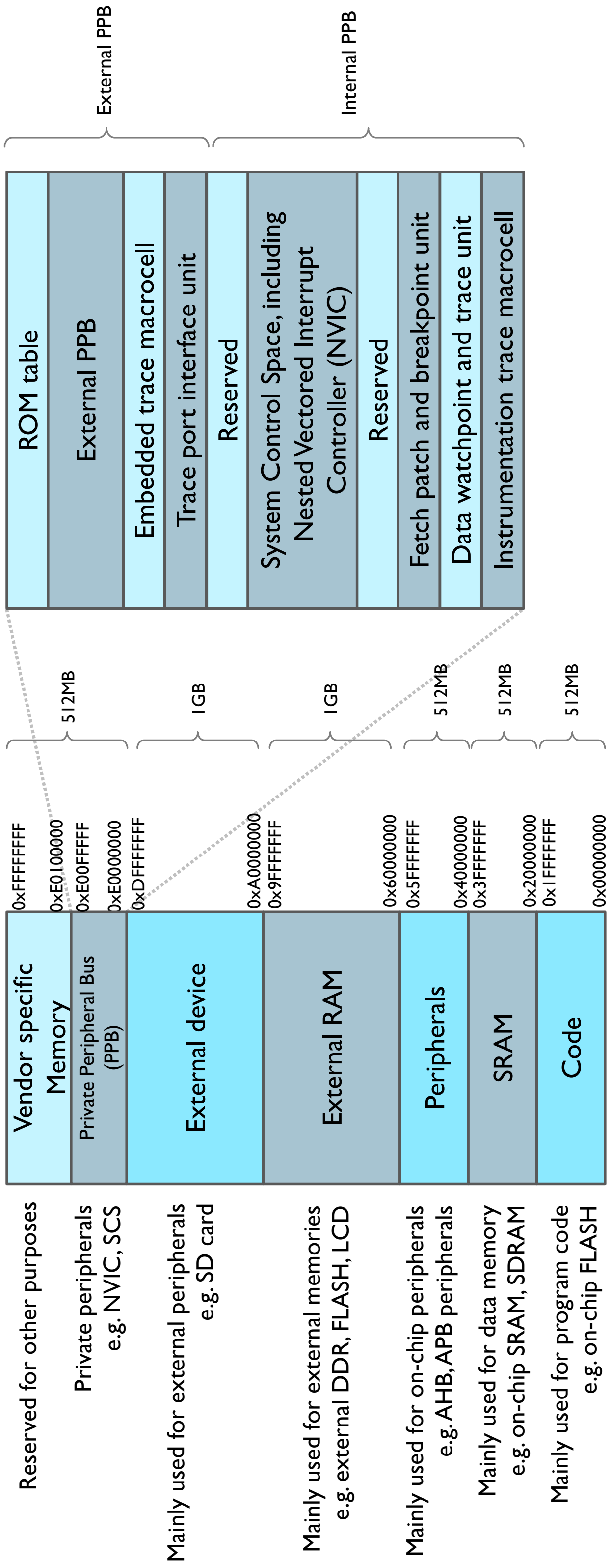
---

[1] LiB Low-level Embedded NXP LPC4088 Quick Start

Cortex-M4 Memory Map

- The Cortex-M4 processor has 4 GB of memory address space
  - Support for bit-band operation (detailed later)
- The 4GB memory space is architecturally defined as a number of regions
  - Each region is given for recommended usage
  - Easy for software programmer to port between different devices
- Nevertheless, despite of the default memory map, the actual usage of the memory map can also be flexibly defined by the user, except some fixed memory addresses, such as internal private peripheral bus

M4 Memory Map (cont.)

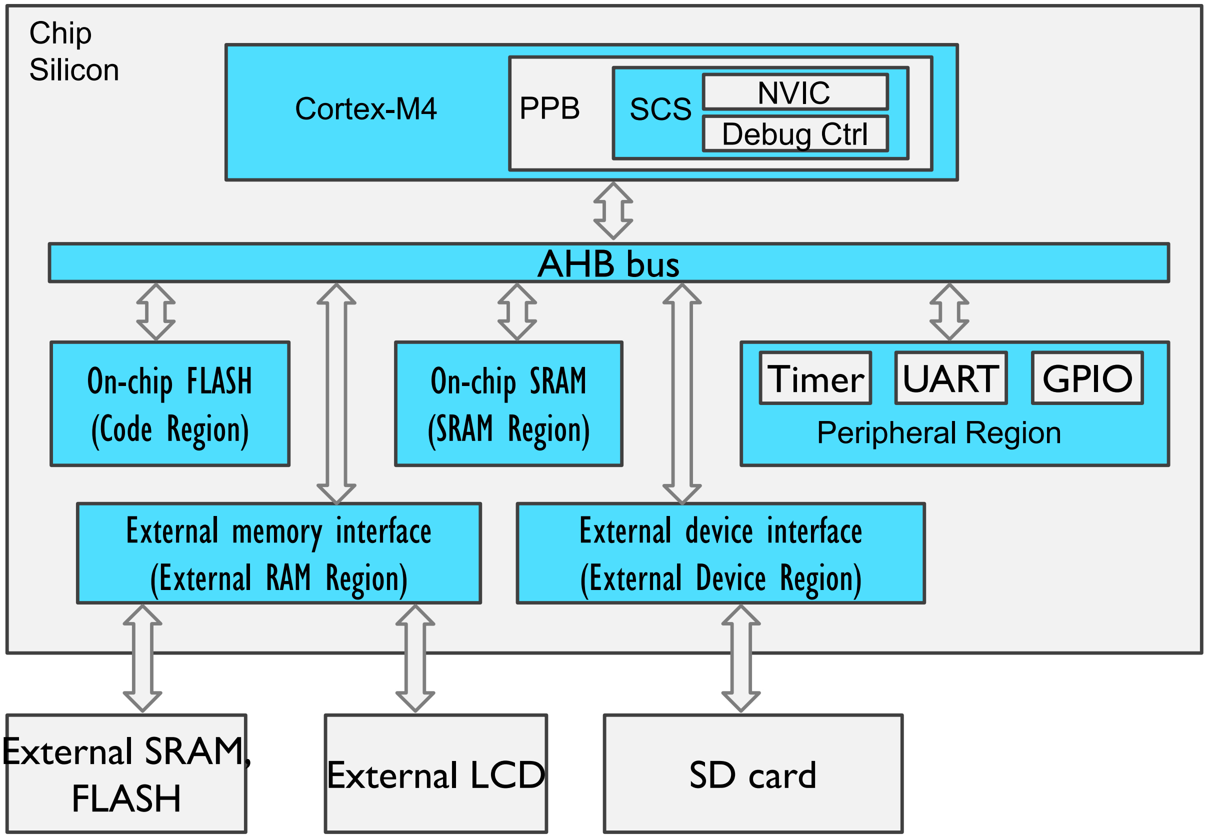


M4 Memory Map (cont.)

- Code Region
  - Primarily used to store program code
  - Can also be used for data memory
  - On-chip memory, such as on-chip FLASH
- SRAM Region
  - Primarily used to store data, such as heaps and stacks
  - Can also be used for program code
  - On-chip memory; despite its name “SRAM”, the actual device could be SRAM, SDRAM or other types
- Peripheral Region
  - Primarily used for peripherals, such as Advanced High-performance Bus (AHB) or Advanced Peripheral Bus (APB) peripherals
- External RAM Region
  - Primarily used to store large data blocks, or memory caches
  - Off-chip memory, slower than on-chip SRAM region
- External Device Region
  - Primarily used to map to external devices
  - Off-chip devices, such as SD card
- Internal Private Peripheral Bus (PPB)

- Used inside the processor core for internal control
- Within PPB, a special range of memory is defined as System Control Space (SCS)
- The Nested Vectored Interrupt Controller (NVIC) is part of SCS

Cortex-M4 Memory Map Example



Bit-band Operations

- Bit-band operation allows a single load/store operation to access a single bit in the memory, for example, to change a single bit of one 32-bit data:
  - Normal operation without bit-band (read-modify-write)
  - Read the value of 32-bit data
  - Modify a single bit of the 32-bit value (keep other bits unchanged)
  - Write the value back to the address
  - Bit-band operation
  - Directly write a single bit (0 or 1) to the "bit-band alias address" of the data
- Bit-band alias address
  - Each bit-band alias address is mapped to a real data address
  - When writing to the bit-band alias address, only a single bit of the data will be changed

Bit-band Operation Example

- For example, in order to set bit[3] in word data in address 0x20000000:

```
;Read-Modify-Write Operation

LDR      R1, =0x20000000   ;Setup address
LDR      R0, [R1]          ;Read
ORR.W    R0, #0x8          ;Modify bit
STR      R0, [R1]          ;Write back
```

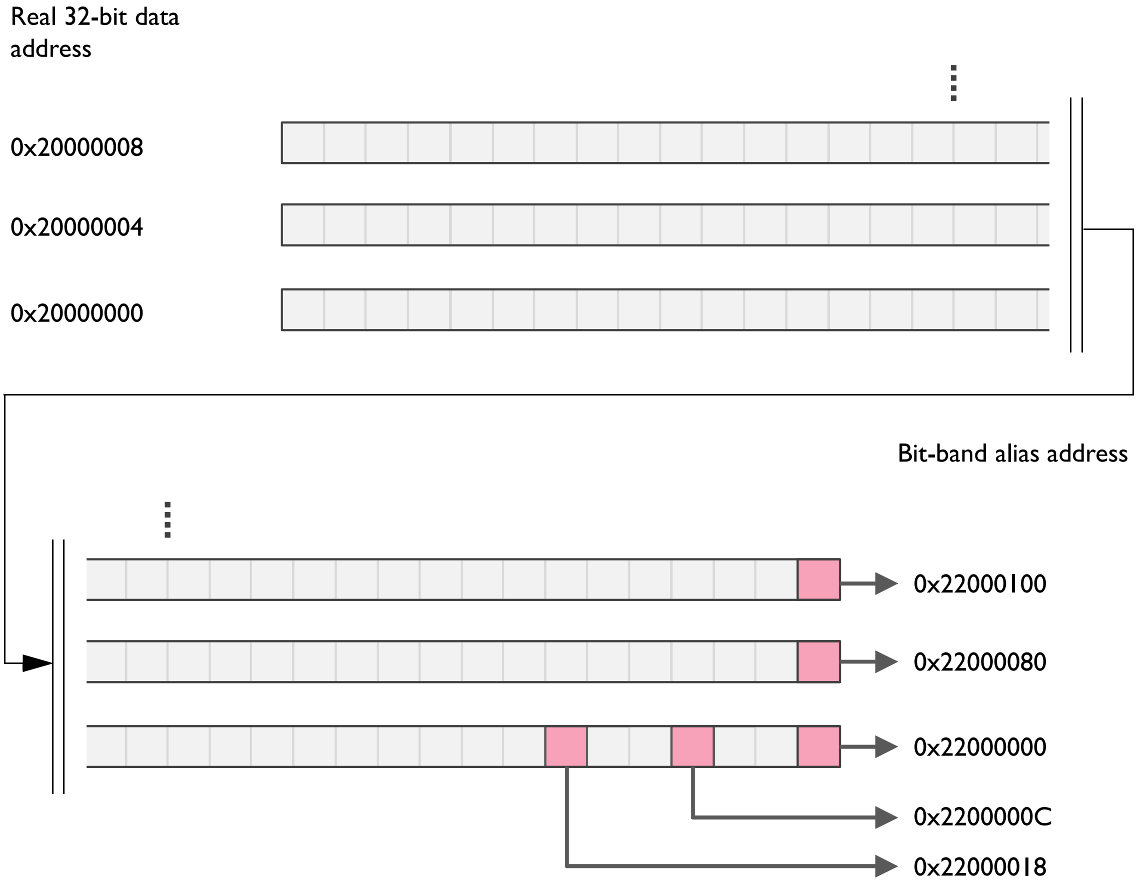
```
;Bit-band Operation

LDR      R1, =0x2200000C   ;Setup address
MOV      R0, #1            ;Load data
STR      R0, [R1]          ;Write
```

- Read-Modify-Write operation
  - Read the real data address (0x20000000)
  - Modify the desired bit (retain other bits unchanged)
  - Write the modified data back
- Bit-band operation
  - Directly set the bit by writing ‘1’ to address 0x2200000C, which is the alias address of the fourth bit of the 32-bit data at 0x20000000
  - In effect, this single instruction is mapped to 2 bus transfers: read data from 0x20000000 to the buffer, and then write to 0x20000000 from the buffer with bit [3] set

Bit-band Alias Address

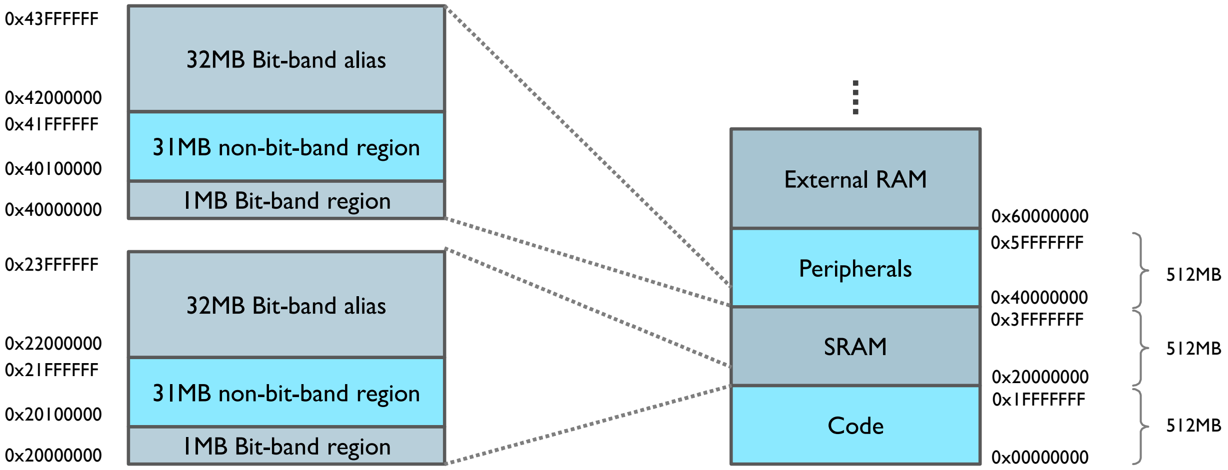
Each bit of the 32-bit data is one-to-one mapped to the bit-band alias address

- For example, the fourth bit (bit [3]) of the data at 0x20000000 is mapped to the bit-band alias address at 0x2200000C
- Hence, to set bit [3] of the data at 0x20000000, we only need to write ‘1’ to address 0x2200000C
- In Cortex-M4, there are two pre-defined bit-band alias regions: one for SRAM region, and one for peripherals region



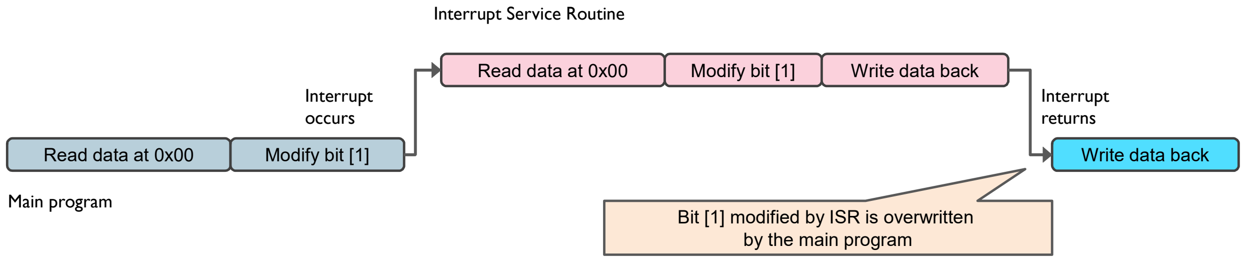
Bit-band Alias Address (cont.)

- SRAM region
  - 32MB memory space (`0x22000000` – `0x23FFFFFF`) is used as the bit-band alias region for 1MB data (`0x20000000` – `0x200FFFFF`)
- Peripherals region
  - 32MB memory space (`0x42000000` – `0x43FFFFFF`) is used as the bit-band alias region for 1MB data (`0x40000000` – `0x400FFFFF`)



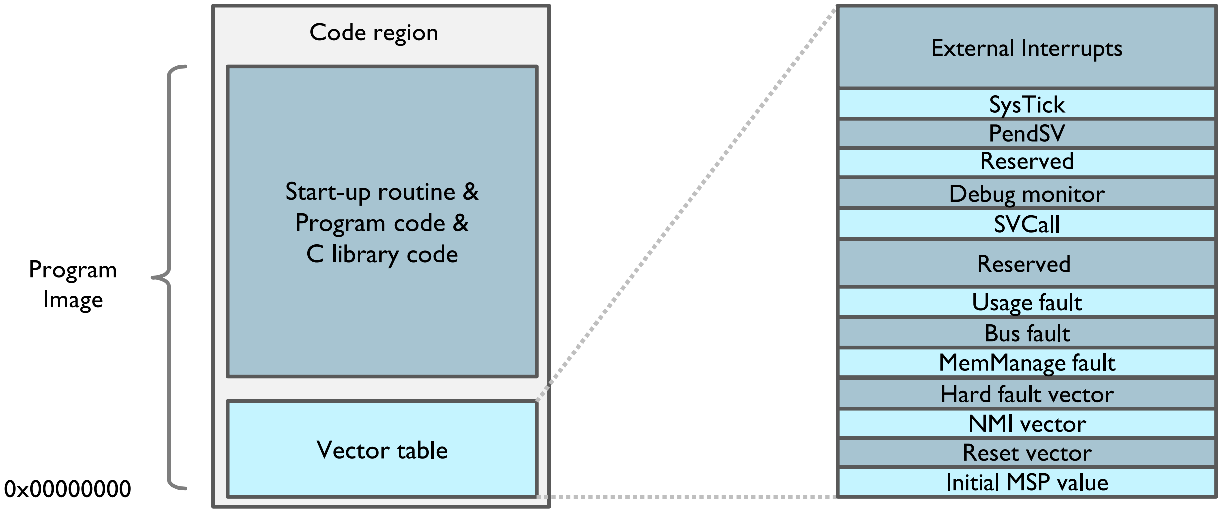
Benefits of Bit-Band Operations

- Faster bit operations
- Fewer instructions
- Atomic operation, avoid hazards
  - For example, if an interrupt is triggered and served during the Read-Modify-Write operations, and the interrupt service routine modifies the same data, a data conflict will occur



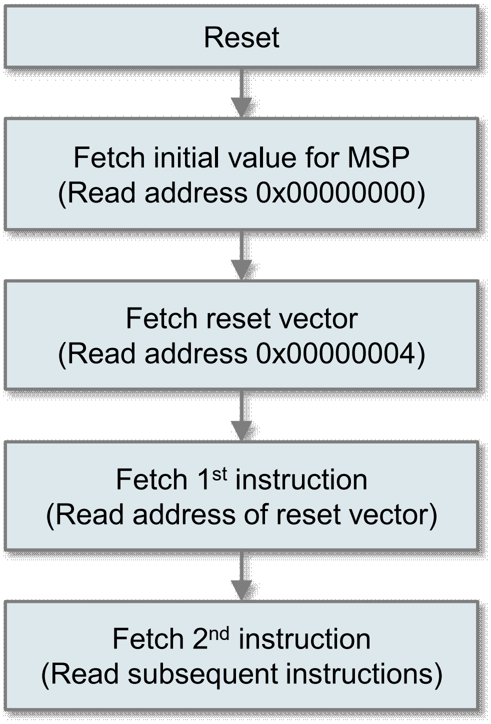
Cortex-M4 Program Image

- The program image in Cortex-M4 contains
  - Vector table -- includes the starting addresses of exceptions (vectors) and the value of the main stack point (MSP);
  - C start-up routine;
  - Program code – application code and data;
  - C library code – program codes for C library functions



Cortex-M4 Program Image (cont)

- After Reset, the processor:
  - First reads the initial MSP value;
  - Then reads the reset vector;
  - Branches to the start of the programme execution address (reset handler);
  - Subsequently executes program instructions



Reset

Fetch initial value for MSP
(Read address 0x00000000)

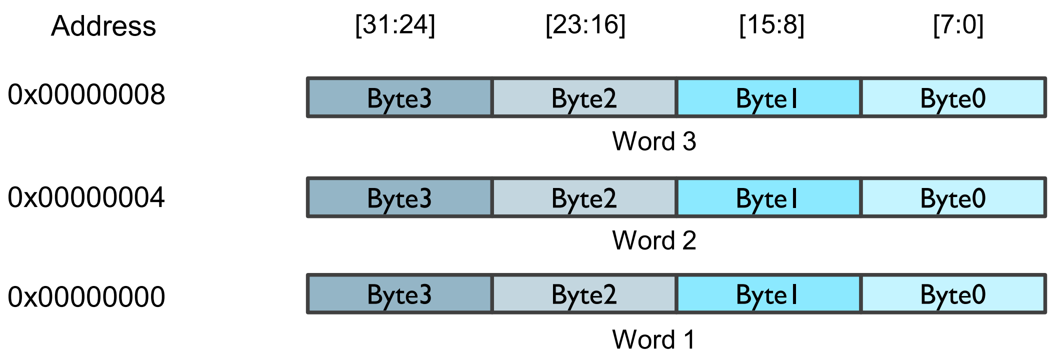
Fetch reset vector
(Read address 0x00000004)

Fetch 1st instruction
(Read address of reset vector)

Fetch 2nd instruction
(Read subsequent instructions)

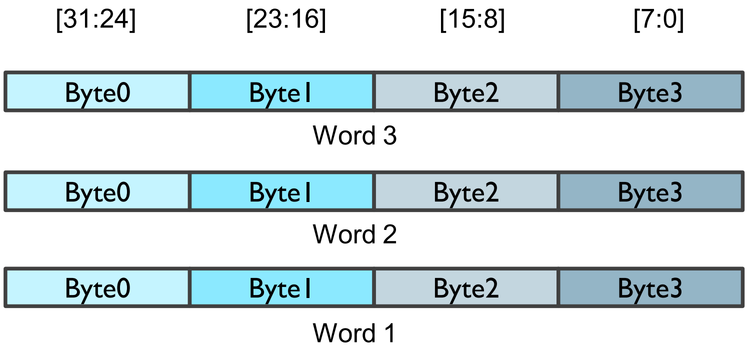
Cortex-M4 Endianness

- Endian refers to the order of bytes stored in memory
  - Little endian: lowest byte of a word-size data is stored in bit 0 to bit 7
  - Big endian: lowest byte of a word-size data is stored in bit 24 to bit 31
- Cortex-M4 supports both little endian and big endian
- However, Endianness only exists in the hardware level



Address	[31:24]	[23:16]	[15:8]	[7:0]
0x00000008	Byte3	Byte2	Byte1	Byte0
		Word 3		
0x00000004	Byte3	Byte2	Byte1	Byte0
		Word 2		
0x00000000	Byte3	Byte2	Byte1	Byte0
		Word 1		

Little endian 32-bit memory



[31:24]	[23:16]	[15:8]	[7:0]
Byte0	Byte1	Byte2	Byte3
	Word 3		
Byte0	Byte1	Byte2	Byte3
	Word 2		
Byte0	Byte1	Byte2	Byte3
	Word 1		

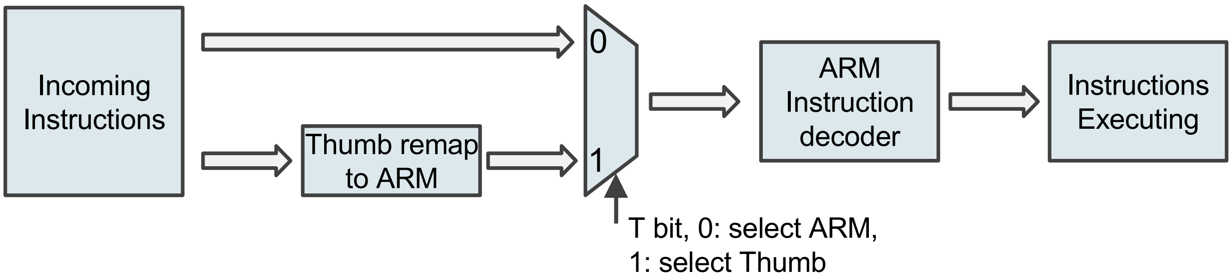
Big endian 32-bit memory

ARM and Thumb® Instruction Set

- Early ARM instruction set
  - 32-bit instruction set, called the ARM instructions
  - Powerful and good performance
  - Larger program memory compared to 8-bit and 16-bit processors
  - Larger power consumption
- Thumb-1 instruction set
  - 16-bit instruction set, first used in ARM7TDMI processor in 1995
  - Provides a subset of the ARM instructions, giving better code density compared to 32-bit RISC architecture
  - Code size is reduced by ~30%, but performance is also reduced by ~20%

ARM and Thumb Instruction Set (cont.)

- Mix of ARM and Thumb-1 Instruction sets
  - Benefit from both 32-bit ARM (high performance) and 16-bit Thumb-1 (high code density)
  - A multiplexer is used to switch between two states: ARM state (32-bit) and Thumb state (16-bit), which requires a switching overhead



- Thumb-2 instruction set
- Consists of both 32-bit Thumb instructions and original 16-bit Thumb-1 instruction sets
- Compared to 32-bit ARM instructions set, code size is reduced by ~26%, while keeping a similar performance
- Capable of handling all processing requirements in one operation state

Cortex-M4 Instruction Set

- Cortex-M4 processor
  - ARMv7-M architecture
  - Supports 32-bit Thumb-2 instructions
  - Possible to handle all processing requirements in one operation state (Thumb state)
  - Compared with traditional ARM processors (use state switching), advantages include:
    - * No state switching overhead – both execution time and instruction space are saved
    - * No need to separate ARM code and Thumb code source files, which makes the development and maintenance of software easier
    - * Easier to get optimized efficiency and performance

Cortex-M4 Instruction Set (cont.)

- ARM assembly syntax:

```
label
    mnemonic operand1,operand2, …; Comments
```

  - Label is used as a reference to an address location;
  - Mnemonic is the name of the instruction;
  - Operand1 is the destination of the operation;
  - Operand2 is normally the source of the operation;
  - Comments are written after " ; ", which does not affect the program;
  - For example

```
MOVS R3, #0x11;Set register R3 to 0x11
```

  - Note that the assembly code can be assembled by either ARM assembler (armasm) or assembly tools from a variety of vendors (e.g. GNU tool chain). When using GNU tool chain, the syntax for labels and comments is slightly different

Cortex-M4 Instruction Set Tables

Mnemonic	Operands	Brief description	Flags
ADC, ADCS	{Rd,} Rn, Op2	Add with Carry	N,Z,C,V
ADD, ADDS	{Rd,} Rn, Op2	Add	N,Z,C,V
ADD, ADDW	{Rd,} Rn, #imm12	Add	N,Z,C,V
ADR	Rd, label	Load PC-relative Address	
AND, ANDS	{Rd,} Rn, Op2	Logical AND	N,Z,C
ASR, ASRS	Rd, Rm, <Rs|#n>	Arithmetic Shift Right	N,Z,C
B	label	Branch	
BFC	Rd, #lsb, #width	Bit Field Clear	
BFI	Rd, Rn, #lsb, #width	Bit Field Insert	
BIC, BICS	{Rd,} Rn, Op2	Bit Clear	N,Z,C
BKPT	#imm	Breakpoint	
BL	label	Branch with Link	
BLX	Rm	Branch indirect with Link	
BX	Rm	Branch indirect	

Cortex-M4 Instruction Set Tables (cont.)

Mnemonic	Operands	Brief description	Flags
CBNZ	Rn, label	Compare and Branch if Non Zero	
CBZ	Rn, label	Compare and Branch if Zero	
CLREX		Clear Exclusive	
CLZ	Rd, Rm	Count Leading Zeros	
CMN	Rn, Op2	Compare Negative	N,Z,C,V
CMP	Rn, Op2	Compare	N,Z,C,V
CPSID	i	Change Processor State, Disable Interrupts	
CPSIE	i	Change Processor State, Enable Interrupts	
DMB		Data Memory Barrier	
DSB		Data Synchronization Barrier	
EOR, EORS	{Rd,} Rn, Op2	Exclusive OR	N,Z,C
ISB	-	Instruction Synchronization Barrier	

Cortex-M4 Instruction Set Tables (cont.)

Mnemonic	Operands	Brief description	Flags
IT		If-Then condition block	
LDM	Rn{!}, reglist	Load Multiple registers, increment after	
LDMDB, LDMEA	Rn{!}, reglist	Load Multiple registers, decrement before	
LDMFD, LDMIA	Rn{!}, reglist	Load Multiple registers, increment after	
LDR	Rt, [Rn, #offset]	Load Register with word	
LDRB, LDRBT	Rt, [Rn, #offset]	Load Register with byte	
LDRD	Rt, Rt2, [Rn, #offset]	Load Register with two bytes	
LDREX	Rt, [Rn, #offset]	Load Register Exclusive	
LDREXB	Rt, [Rn]	Load Register Exclusive with Byte	
LDREXH	Rt, [Rn]	Load Register Exclusive with Halfword	
LDRH, LDRHT	Rt, [Rn, #offset]	Load Register with Halfword	

Cortex-M4 Instruction Set Tables (cont.)

Mnemonic	Operands	Brief description	Flags
LDRSB, LDRSBT	Rt, [Rn, #offset]	Load Register with Signed Byte	
LDRSH, LDRSHT	Rt, [Rn, #offset]	Load Register with Signed Halfword	
LDRT	Rt, [Rn, #offset]	Load Register with word	
LSL, LSLS	Rd, Rm, <Rs|#n>	Logical Shift Left	N,Z,C
LSR, LSRS	Rd, Rm, <Rs|#n>	Logical Shift Right	N,Z,C
MLA	Rd, Rn, Rm, Ra	Multiply with Accumulate, 32-bit result	
MLS	Rd, Rn, Rm, Ra	Multiply and Subtract, 32-bit result	
MOV, MOVS	Rd, Op2	Move	N,Z,C
MOVT	Rd, #imm16	Move Top	
MOVW, MOV	Rd, #imm16	Move 16-bit constant	N,Z,C
MRS	Rd, spec_reg	Move from Special Register to general register	
MSR	spec_reg, Rm	Move from general register to Special Register	N,Z,C,V

Cortex-M4 Instruction Set Tables (cont.)

Mnemonic	Operands	Brief description	Flags
MUL, MULS	{Rd,} Rn, Rm	Multiply, 32-bit result	N,Z
MVN, MVNS	Rd, Op2	Move NOT	N,Z,C
NOP		No Operation	
ORN, ORNS	{Rd,} Rn, Op2	Logical OR NOT	N,Z,C
ORR, ORRS	{Rd,} Rn, Op2	Logical OR	N,Z,C
PKHTB, PKHBT	{Rd, } Rn, Rm, Op2	Pack Halfword	
POP	reglist	Pop registers from stack	
PUSH	reglist	Push registers onto stack	
QADD	{Rd, } Rn, Rm	Saturating double and Add	Q
QADD16	{Rd, } Rn, Rm	Saturating Add 16	
QADD8	{Rd, } Rn, Rm	Saturating Add 8	

Cortex-M4 Instruction Set Tables (cont.)

Mnemonic	Operands	Brief description	Flags
QASX	{Rd,} Rn, Rm	Saturating Add and Subtract with Exchange	
QDADD	{Rd,} Rn, Rm	Saturating Add	Q
QDSUB	{Rd,} Rn, Rm	Saturating double and Subtract	Q
QSAX	{Rd,} Rn, Rm	Saturating Subtract and Add with Exchange	
QSUB	{Rd,} Rn, Rm	Saturating Subtract	Q
QSUB16	{Rd,} Rn, Rm	Saturating Subtract 16	
QSUB8	{Rd,} Rn, Rm	Saturating Subtract 8	
RBIT	Rd, Rn	Reverse Bits	
REV	Rd, Rn	Reverse byte order in a word	
REV16	Rd, Rn	Reverse byte order in each halfword	
REVSH	Rd, Rn	Reverse byte order in bottom halfword and sign extend	
ROR, RORS	Rd, Rm, <Rs|#n>	Rotate Right	N,Z,C

Cortex-M4 Instruction Set Tables (cont.)

Mnemonic	Operands	Brief description	Flags
RRX, RRXS	Rd, Rm	Rotate Right with Extend	N,Z,C
RSB, RSBS	{Rd,} Rn, Op2	Reverse Subtract	N,Z,C,V
SADD16	{Rd, } Rn, Rm	Signed Add 16	GE
SADD8	{Rd, } Rn, Rm	Signed Add 8	GE
SASX	{Rd, } Rn, Rm	Signed Add and Subtract with Exchange	GE
SBC, SBCS	{Rd,} Rn, Op2	Subtract with Carry	N,Z,C,V
SBFX	Rd, Rn, #lsb, #width	Signed Bit Field Extract	
SDIV	{Rd,} Rn, Rm	Signed Divide	
SEV		Send Event	
SHADD16	{Rd,} Rn, Rm	Signed Halving Add 16	
SHADD8	{Rd,} Rn, Rm	Signed Halving Add 8	
SHASX	{Rd,} Rn, Rm	Signed Halving Add and Subtract with Exchange	

Cortex-M4 Instruction Set Tables (cont.)

Mnemonic	Operands	Brief description	Flags
SHSAX	{Rd,} Rn, Rm	Signed Halving Subtract and Add with Exchange	
SHSUB16	{Rd,} Rn, Rm	Signed Halving Subtract 16	
SHSUB8	{Rd,} Rn, Rm	Signed Halving Subtract 8	
SMLABB, SMLABT, SMLATB, SMLATT	Rd, Rn, Rm, Ra	Signed Multiply Accumulate Long (halfwords)	Q
SMLAD, SMLADX	Rd, Rn, Rm, Ra	Signed Multiply Accumulate Dual	Q
SMLAL	RdLo, RdHi, Rn, Rm	Signed Multiply with Accumulate (32 x 32 + 64), 64-bit result	
SMLALBB, SMLALBT, SMLALTB, SMLALTT	RdLo, RdHi, Rn, Rm	Signed Multiply Accumulate Long, halfwords	
SMLALD, SMLALDX	RdLo, RdHi, Rn, Rm	Signed Multiply Accumulate Long Dual	
SMLAWB, SMLAWT	Rd, Rn, Rm, Ra	Signed Multiply Accumulate, word by halfword	Q
SMLSD	Rd, Rn, Rm, Ra	Signed Multiply Subtract Dual	Q
SMLSLD	RdLo, RdHi, Rn, Rm	Signed Multiply Subtract Long Dual	
SMMLA	Rd, Rn, Rm, Ra	Signed Most significant word Multiply Accumulate	

Cortex-M4 Instruction Set Tables (cont.)

Mnemonic	Operands	Brief description	Flags
SMMLS, SMMLR	Rd, Rn, Rm, Ra	Signed Most significant word Multiply Subtract	
SMMUL, SMMULR	{Rd,} Rn, Rm	Signed Most significant word Multiply	
SMUAD	{Rd,} Rn, Rm	Signed dual Multiply Add	Q
SMULBB, SMULBT SMULTB, SMULTT	{Rd,} Rn, Rm	Signed Multiply (halfwords)	
SMULL	RdLo, RdHi, Rn, Rm	Signed Multiply (32 x 32), 64-bit result	
SMULWB, SMULWT	{Rd,} Rn, Rm	Signed Multiply word by halfword	
SMUSD, SMUSDX	{Rd,} Rn, Rm	Signed dual Multiply Subtract	
SSAT	Rd, #n, Rm {,shift #s}	Signed Saturate	Q
SSAT16	Rd, #n, Rm	Signed Saturate 16	Q
SSAX	{Rd,} Rn, Rm	Signed Subtract and Add with Exchange	GE
SSUB16	{Rd,} Rn, Rm	Signed Subtract 16	
SSUB8	{Rd,} Rn, Rm	Signed Subtract 8	

Cortex-M4 Instruction Set Tables (cont.)

Mnemonic	Operands	Brief description	Flags
STM	Rn{!}, reglist	Store Multiple registers, increment after	
STMDB, STMEA	Rn{!}, reglist	Store Multiple registers, decrement before	
STMFD, STMIA	Rn{!}, reglist	Store Multiple registers, increment after	
STR	Rt, [Rn, #offset]	Store Register word	
STRB, STRBT	Rt, [Rn, #offset]	Store Register byte	
STRD	Rt, Rt2, [Rn, #offset]	Store Register two words	
STREX	Rd, Rt, [Rn, #offset]	Store Register Exclusive	
STREXB	Rd, Rt, [Rn]	Store Register Exclusive Byte	
STREXH	Rd, Rt, [Rn]	Store Register Exclusive Halfword	
STRH, STRHT	Rt, [Rn, #offset]	Store Register Halfword	
STRT	Rt, [Rn, #offset]	Store Register word	
SUB, SUBS	{Rd,} Rn, Op2	Subtract	N,Z,C,V

Cortex-M4 Instruction Set Tables (cont.)

Mnemonic	Operands	Brief description	Flags
SUB, SUBW	{Rd,} Rn, #imm12	Subtract	N,Z,C,V
SVC	#imm	Supervisor Call	
SXTAB	{Rd,} Rn, Rm,{,ROR #}	Extend 8 bits to 32 and add	
SXTAB16	{Rd,} Rn, Rm,{,ROR #}	Dual extend 8 bits to 16 and add	
SXTAH	{Rd,} Rn, Rm,{,ROR #}	Extend 16 bits to 32 and add	
SXTB16	{Rd,} Rm {,ROR #n}	Signed Extend Byte 16	
SXTB	{Rd,} Rm {,ROR #n}	Sign extend a byte	
SXTH	{Rd,} Rm {,ROR #n}	Sign extend a halfword	
TBB	[Rn, Rm]	Table Branch Byte	
TBH	[Rn, Rm, LSL #1]	Table Branch Halfword	
TEQ	Rn, Op2	Test Equivalence	N,Z,C
TST	Rn, Op2	Test	N,Z,C

Cortex-M4 Instruction Set Tables (cont.)

Mnemonic	Operands	Brief description	Flags
UADD16	{Rd,} Rn, Rm	Unsigned Add 16	GE
UADD8	{Rd,} Rn, Rm	Unsigned Add 8	GE
USAX	{Rd,} Rn, Rm	Unsigned Subtract and Add with Exchange	GE
UHADD16	{Rd,} Rn, Rm	Unsigned Halving Add 16	
UHADD8	{Rd,} Rn, Rm	Unsigned Halving Add 8	
UHASX	{Rd,} Rn, Rm	Unsigned Halving Add and Subtract with Exchange	
UHSAX	{Rd,} Rn, Rm	Unsigned Halving Subtract and Add with Exchange	
UHSUB16	{Rd,} Rn, Rm	Unsigned Halving Subtract 16	
UHSUB8	{Rd,} Rn, Rm	Unsigned Halving Subtract 8	
UBFX	Rd, Rn, #lsb, #width	Unsigned Bit Field Extract	
UDIV	{Rd,} Rn, Rm	Unsigned Divide	
UMAAL	RdLo, RdHi, Rn, Rm	Unsigned Multiply Accumulate Accumulate Long (32 x 32 + 32 +32), 64-bit result	

Cortex-M4 Instruction Set Tables (cont.)

Mnemonic	Operands	Brief description	Flags
UMLAL	RdLo, RdHi, Rn, Rm	Unsigned Multiply with Accumulate (32 x 32 + 64), 64-bit result	
UMULL	RdLo, RdHi, Rn, Rm	Unsigned Multiply (32 x 32), 64-bit result	
UQADD16	{Rd,} Rn, Rm	Unsigned Saturating Add 16	
UQADD8	{Rd,} Rn, Rm	Unsigned Saturating Add 8	
UQASX	{Rd,} Rn, Rm	Unsigned Saturating Add and Subtract with Exchange	
UQSAX	{Rd,} Rn, Rm	Unsigned Saturating Subtract and Add with Exchange	
UQSUB16	{Rd,} Rn, Rm	Unsigned Saturating Subtract 16	
UQSUB8	{Rd,} Rn, Rm	Unsigned Saturating Subtract 8	
USAD8	{Rd,} Rn, Rm	Unsigned Sum of Absolute Differences	
USADA8	{Rd,} Rn, Rm, Ra	Unsigned Sum of Absolute Differences and Accumulate	
USAT	Rd, #n, Rm {,shift #s}	Unsigned Saturate	Q
USAT16	Rd, #n, Rm	Unsigned Saturate 16	Q

Cortex-M4 Instruction Set Tables (cont.)

Mnemonic	Operands	Brief description	Flags
UASX	{Rd,} Rn, Rm	Unsigned Add and Subtract with Exchange	GE
USUB16	{Rd,} Rn, Rm	Unsigned Subtract 16	GE
USUB8	{Rd,} Rn, Rm	Unsigned Subtract 8	GE
UXTAB	{Rd,} Rn, Rm,{,ROR #}	Rotate, extend 8 bits to 32 and Add	
UXTAB16	{Rd,} Rn, Rm,{,ROR #}	Rotate, dual extend 8 bits to 16 and Add	
UXTAH	{Rd,} Rn, Rm,{,ROR #}	Rotate, unsigned extend and Add Halfword	
UXTB	{Rd,} Rm {,ROR #n}	Zero extend a Byte	
UXTB16	{Rd,} Rm {,ROR #n}	Unsigned Extend Byte 16	
UXTH	{Rd,} Rm {,ROR #n}	Zero extend a Halfword	
VABS.F32	Sd, Sm	Floating-point Absolute	
VADD.F32	{Sd,} Sn, Sm	Floating-point Add	
VCMP.F32	Sd, <Sm | #0.0>	Compare two floating-point registers, or one floating-point register and zero	FPSCR

Cortex-M4 Instruction Set Tables (cont.)

Mnemonic	Operands	Brief description	Flags
VCMPE.F32	Sd, <Sm | #0.0>	Compare two floating-point registers, or one floating-point register and zero with Invalid Operation check	FPSCR
VCVT.S32.F32	Sd, Sm	Convert between floating-point and integer	
VCVT.S16.F32	Sd, Sd, #fbits	Convert between floating-point and fixed point	
VCVTR.S32.F32	Sd, Sm	Convert between floating-point and integer with rounding	
VCVT<B|H>.F32.F16	Sd, Sm	Converts half-precision value to single-precision	
VCVTT<B|T>.F32.F16	Sd, Sm	Converts single-precision register to half-precision	
VDIV.F32	{Sd,} Sn, Sm	Floating-point Divide	
VFMA.F32	{Sd,} Sn, Sm	Floating-point Fused Multiply Accumulate	
VFNMA.F32	{Sd,} Sn, Sm	Floating-point Fused Negate Multiply Accumulate	
VFMS.F32	{Sd,} Sn, Sm	Floating-point Fused Multiply Subtract	
VFNMS.F32	{Sd,} Sn, Sm	Floating-point Fused Negate Multiply Subtract	
VLDM.F<32|64>	Rn{!}, list	Load Multiple extension registers	

Cortex-M4 Instruction Set Tables (cont.)

Mnemonic	Operands	Brief description	Flags
VLDR.F<32|64>	<Dd|Sd>, [Rn]	Load an extension register from memory	
VLMA.F32	{Sd,} Sn, Sm	Floating-point Multiply Accumulate	
VLMS.F32	{Sd,} Sn, Sm	Floating-point Multiply Subtract	
VMOV.F32	Sd, #imm	Floating-point Move immediate	
VMOV	Sd, Sm	Floating-point Move register	
VMOV	Sn, Rt	Copy ARM core register to single precision	
VMOV	Sm, Sm1, Rt, Rt2	Copy 2 ARM core registers to 2 single precision	
VMOV	Dd[x], Rt	Copy ARM core register to scalar	
VMOV	Rt, Dn[x]	Copy scalar to ARM core register	
VMRS	Rt, FPSCR	Move FPSCR to ARM core register or APSR	N,Z,C,V
VMSR	FPSCR, Rt	Move to FPSCR from ARM Core register	FPSCR
VMUL.F32	{Sd,} Sn, Sm	Floating-point Multiply	

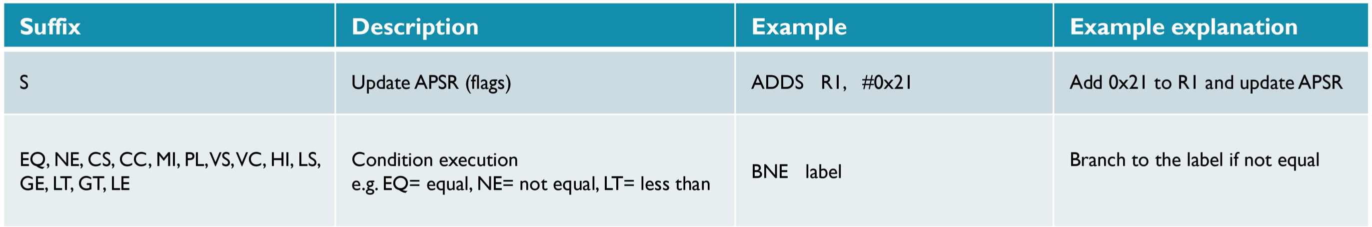
Cortex-M4 Instruction Set Tables (cont.)

Mnemonic	Operands	Brief description	Flags
VNEG.F32	Sd, Sm	Floating-point Negate	
VNMLA.F32	Sd, Sn, Sm	Floating-point Multiply and Add	
VNMLS.F32	Sd, Sn, Sm	Floating-point Multiply and Subtract	
VNMUL	{Sd,} Sn, Sm	Floating-point Multiply	
VPOP	list	Pop extension registers	
VPUSH	list	Push extension registers	
VSQRT.F32	Sd, Sm	Calculates floating-point Square Root	
VSTM	Rn{!}, list	Floating-point register Store Multiple	
VSTR.F<32|64>	Sd, [Rn]	Stores an extension register to memory	
VSUB.F<32|64>	{Sd,} Sn, Sm	Floating-point Subtract	
WFE		Wait For Event	
WFI		Wait For Interrupt	

Note: full explanation of each instruction can be found in Cortex-M4 Devices' Generic User Guide (Ref-4)

Cortex-M4 Instruction Set Tables (cont.)

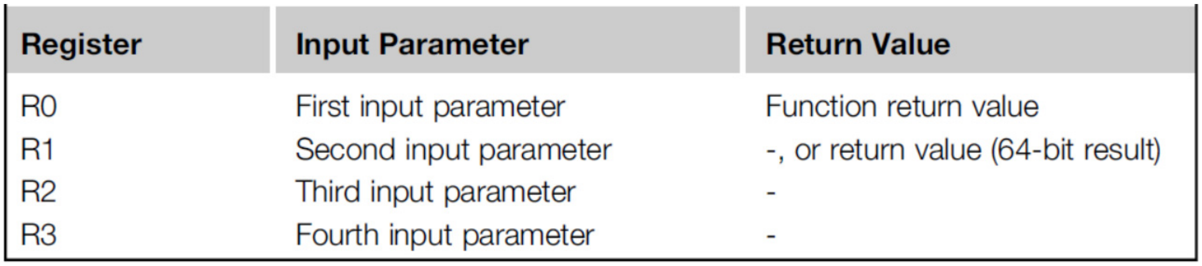
- Cortex-M4 Suffix
  - Some instructions can be followed by suffixes to update processor flags or execute the instruction on a certain condition



Suffix	Description	Example	Example explanation
S	Update APSR (flags)	ADDS R1, #0x21	Add 0x21 to R1 and update APSR
EQ, NE, CS, CC, MI, PL, VS, VC, HI, LS, GE, LT, GT, LE	Condition execution e.g. EQ= equal, NE= not equal, LT= less than	BNE label	Branch to the label if not equal

C Calling Assembly

For real-time DSP applications the most common scenario involving assembly code writing, if needed at all, will be C calling assembly. In simple terms the rules are:



Register	Input Parameter	Return Value
R0	First input parameter	Function return value
R1	Second input parameter	-, or return value (64-bit result)
R2	Third input parameter	-
R3	Fourth input parameter	-

- Formally, the *ARM Architecture Procedure Call Standard* (AAPCS) defines:
  - Which registers must be saved and restored
  - How to call procedures
  - How to return from procedures

AAPCS Register Use Conventions

- Make it easier to create modular, isolated and integrated code
- Scratch registers are not expected to be preserved upon returning from a called subroutine
  - This applies to `r0–r3`
- Preserved ("variable") registers are expected to have their original values upon returning from a called subroutine
  - This applies to `r4–r8`, `r10–r11`
  - Use `PUSH {r4,..}` and `POP {r4,...}`

AAPCS Core Register Use

egister	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6,SB,TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

Must be saved, restored by callee-procedure it may modify them.

Must be saved, restored by callee-procedure it may modify them.
Calling subroutine expects these to retain their value.

Don’t need to be saved. May be used for arguments, results, or temporary values.

Example: Vector Norm Squared

In this example we will be computing the squared length of a vector using 16-bit (int16_t) signed numbers. In mathematical terms we are finding

$$\|\mathbf{A}\|^2 = \sum_{n=1}^{N} A_n^2 \tag{3.1}$$

where

$$\mathbf{A} = \begin{bmatrix} A_1 & \cdots & A_N \end{bmatrix} \tag{3.2}$$

is an N-dimensional vector (column or row vector).

- The solution will be obtained in two different ways:
  - Conventional C programming
  - Cortex-M assembly
- Optimization is not a concern at this point
- The focus here is to see by way of a simple example, how to call a C routine from C (obvious), and how to call an assembly routine from C

C Version

- We implement this simple routine in C using a declared vector length `N` and vector contents in the array `v`
- The C source, which includes the called function `norm_sq_c` is given below:

```
/*****************************************************

      Vector norm-squared routine in C and Assembly

*****************************************************/

#include "fm4_wm8731_init.h"
#include "FM4_slider_interface.h"

// Macros from fm4_wm8731_init.c need to configure GPIO without starting
// codec ISRs. PFR is port function setting register - 0 for GPIO, 1 for
// peripheral function
#define GET_PFR(pin_ofs)  ((volatile unsigned char*) (PFR_BASE + pin_ofs))
// PCR is port pull-up setting register - 0 for pull-up, 1 for no pull-up
#define GET_PCR(pin_ofs)  ((volatile unsigned char*) (PCR_BASE + pin_ofs))
// PDDR is port direction setting register - 0 for GPIO in, 1 for GPIO out
#define GET_DDR(pin_ofs)  ((volatile unsigned char*) (DDR_BASE + pin_ofs))

// Create (instantiate) GUI slider data structure
struct FM4_slider_struct FM4_GUI;

// Norm squared in C prototype
int16_t norm_sq_c(int16_t* v, int16_t n);
// ASM function prototypes
extern uint32_t simple_sqrt(uint32_t x);
extern int16_t norm_sq_asm(int16_t *x, int16_t n);

/*----------------------------------------------------------------------------
 MAIN function
 *----------------------------------------------------------------------------
*/
int main(void){
      int16_t x = 0;
      int16_t v[5] = {1,2,3,6,7};
      uint32_t zInt = 99;
      uint32_t zInt_sq;
      char message[50];
```

```
      // Set up DIAGNOSTIC_PIN GPIO for timing of function calls
      // Follow approach used in fm4_wm8731_init(), but without starting ISR
  bFM4_GPIO_ADE_AN00 = 0x00; // P10   DIAGNOSTIC_PIN
  *GET_PFR(DIAGNOSTIC_PIN) &= ~0u; // set pin function as GPIO
  *GET_DDR(DIAGNOSTIC_PIN) = 1u  ; // set pin direction as output
  *GET_PCR(DIAGNOSTIC_PIN) &= ~0u; // set pin to have pull-up


      // Initialize the slider interface by setting the baud rate (460800 or
      // 921600) and initial float values for each of the 6 slider parameters
      init_slider_interface(&FM4_GUI,460800, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0);

      // Norm squared experiment
      gpio_set(DIAGNOSTIC_PIN,HIGH); //pin = P10/A3
      x = norm_sq_c (v, 5);// call c function
      gpio_set(DIAGNOSTIC_PIN,LOW);
      sprintf(message, "Norm squared C: The answer is %d\n", x);
      write_uart0(message);
      gpio_set(DIAGNOSTIC_PIN,HIGH); //pin = P10/A3
      x = norm_sq_asm (v, 5);// call assembly function
      gpio_set(DIAGNOSTIC_PIN,LOW);
      sprintf(message, "Norm squared ASM: The answer is %d\n", x);
      write_uart0(message);

      // uint16_t Square root experiment
      gpio_set( DIAGNOSTIC_PIN,HIGH); //pin = P10/A3
      zInt_sq = simple_sqrt(zInt);
      gpio_set(DIAGNOSTIC_PIN,LOW);
      sprintf(message, "uint16_t SQRT of %d is %d\n", zInt, zInt_sq);
      write_uart0(message);

  while(1)
      {
       // Update slider parameters
       //update_slider_parameters(&FM4_GUI);
      }
}

int16_t norm_sq_c(int16_t* v, int16_t n)
{
      int16_t i;
      int16_t out = 0;
      for(i=0; i<n; i++)
      {
       out += v[i]*v[i];
      }
      return out;
```

```
}
```

- Notice in this code we have configured three additional output pins for physical code timing
  - `P1B` = `D0` is used to time `norm_sq_c`
  - `P1C` = `D1` is used to time `norm_sq_asm`
- The expected answer is 1 + 4 + 9 + 36 + 49 = 99

```
Norm squared C: The answer is 99
```

From Terminal

- Physical code time and cycle count timing comparison with the assembly version, is come up next

Assembly Version

- The assembly routine is the following:

```
  ; File demo_asm.s
   PRESERVE8 ; Preserve 8 byte stack alignment
   THUMB     ; indicate THUMB code is used
  AREA |.text|, CODE, READONLY;Start of the CODE area
  EXPORT norm_sq_asm
norm_sq_asm FUNCTION
  ; Input array address: R0
  ; Number of elements: R1
  MOVS R2, R0        ; move the address in R0 to R2
  MOVS R0, #0        ; initialize the result
sum_loop
  LDRSH R3, [R2],#0x2; load int16_t value pointed to
                      ; by R2 into R3, then increment
  MLA R0, R3, R3, R0; sq & accum in one step (faster)
  SUBS R1, R1, #1; R1 = R1 - 1, decrement the count
  CMP R1, #0         ; compare to 0 and set Z register
  BNE sum_loop; branch if compare not zero
  BX LR                     ; return R0
  ENDFUNC
  END   ; End of file
```

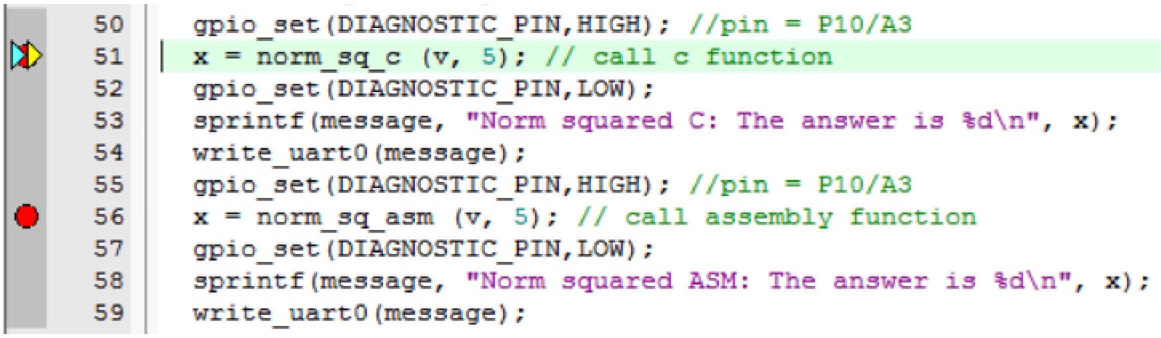
- From just the C source it is not obvious that the function prototype for `norm_asm` is actually an assembly routine
- The answer is again 99

```
Norm squared ASM: The answer is 99
```

From Terminal

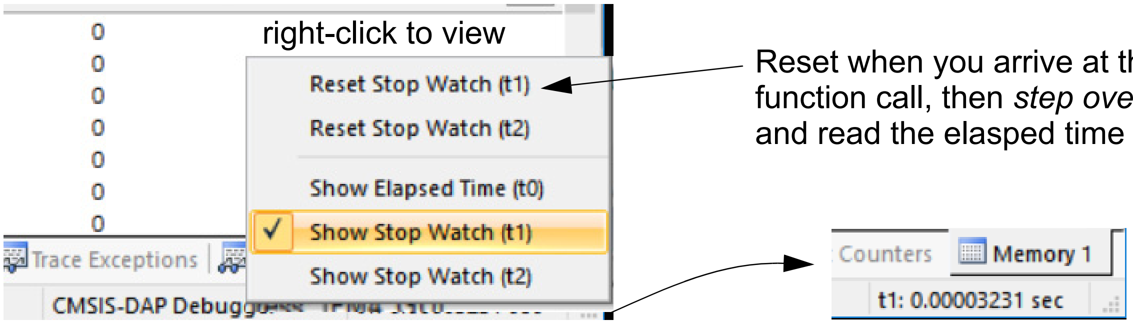
Performance Comparison

- In the Keil IDE debugger we set break points around the function to be timed:



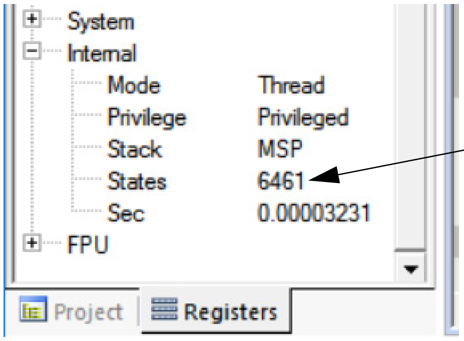
```
gpio_set(DIAGNOSTIC_PIN,HIGH); //pin = P10/A3
x = norm_sq_c (v, 5); // call c function
gpio_set(DIAGNOSTIC_PIN,LOW);
sprintf(message, "Norm squared C: The answer is %d\n", x);
write_uart0(message);
gpio_set(DIAGNOSTIC_PIN,HIGH); //pin = P10/A3
x = norm_sq_asm (v, 5); // call assembly function
gpio_set(DIAGNOSTIC_PIN,LOW);
sprintf(message, "Norm squared ASM: The answer is %d\n", x);
write_uart0(message);
```

- We take measurements using the timers available in the lower right status bar; alternatively count cycles using the states register:



t1: 0.00003231 sec

Reset when you arrive at the function call, then *step over* and read the elasped time



Mode Thread
Privilege Privileged
Stack MSP
States 6461
Sec 0.00003231

When debugging observe the registers and in particular note the change in states before and after the function call is made

norm_sq_c with -O0

t1: 0.00000038 sec

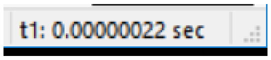
time = 380 ns

norm_sq_c with -O3

t1: 0.00000034 sec

time = 340 ns

norm_sq_asm



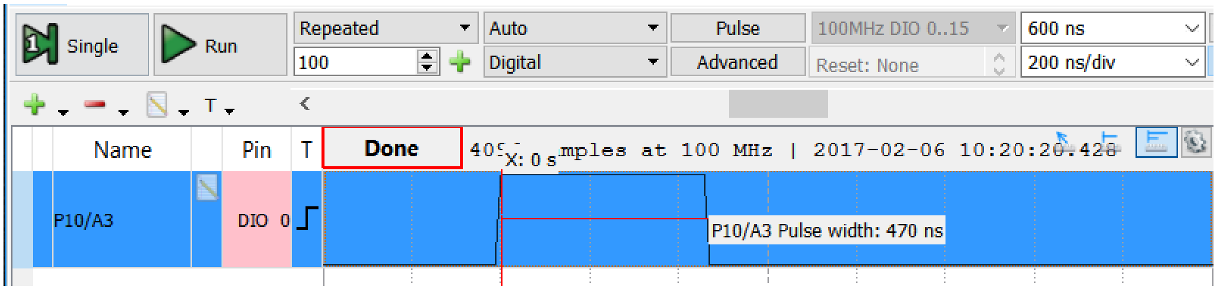
t1: 0.00000022 sec

time = 220 ns

- To interpret the cycle count as real time, consider the FM4 running with a 200 MHz clock frequency or 5 ns clock period
- As a cross check physical timing is explored using the Ana-

log Discovery logic analyzer:

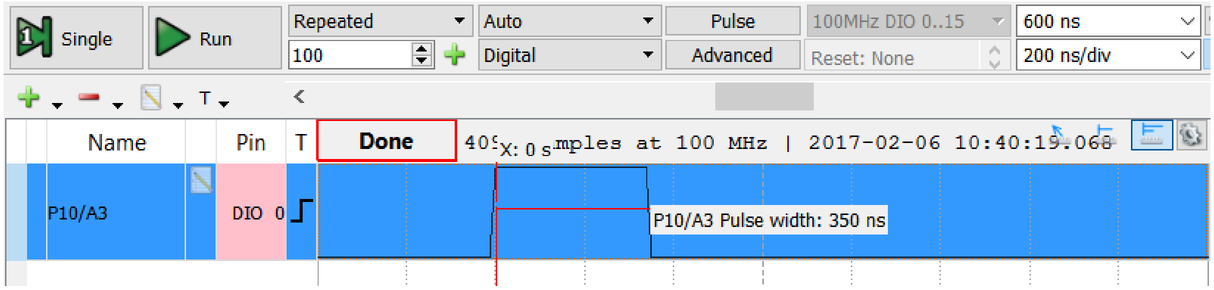
C version (-O3)



40⁵ samples at 100 MHz | 2017-02-06 10:20:20.428

P10/A3 Pulse width: 470 ns

C calling ASM version

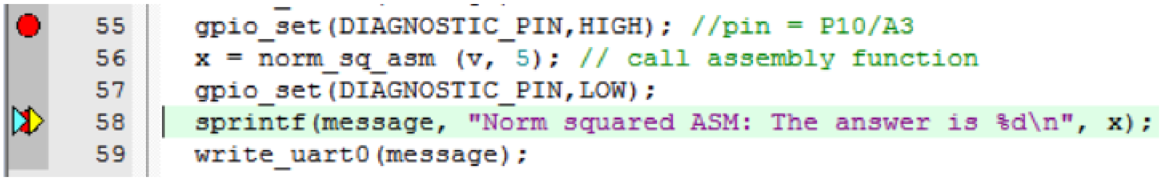


40⁵ samples at 100 MHz | 2017-02-06 10:40:19.068

P10/A3 Pulse width: 350 ns

470 ns vs 350 ns implies a speedup of ~25.5%

- The physical time results are more consistent, but a bias is introduced since time is required to set and reset the GPIO pin
- The ASM code is timed from Keil by setting the break points around the GPIO pin set functions:



```
gpio_set(DIAGNOSTIC_PIN,HIGH); //pin = P10/A3
x = norm_sq_asm (v, 5); // call assembly function
gpio_set(DIAGNOSTIC_PIN,LOW);
sprintf(message, "Norm squared ASM: The answer is %d\n", x);
write_uart0(message);
```

t1: 0.00000042 sec

Time is 420 ns including pin set and reset, vs 220 ns without —> ~200 ns due to pins

- The above result is much longer than the physically measured result; conclude that removing the GPIO bias is not obvious

Example: Unsigned Integer Square Root[1]

- Added to `main`

```
  // uint16_t Square root experiment
  gpio_set(PF7, HIGH); //Pin PF7 = D2
  zInt_sq = simple_sqrt(zInt);
  gpio_set(PF7, LOW); //Pin PF7 = D2
  sprintf(message, "uint16_t SQRT of %d is %d\n", zInt, zInt_sq);
  write_uart0(message);
...
```

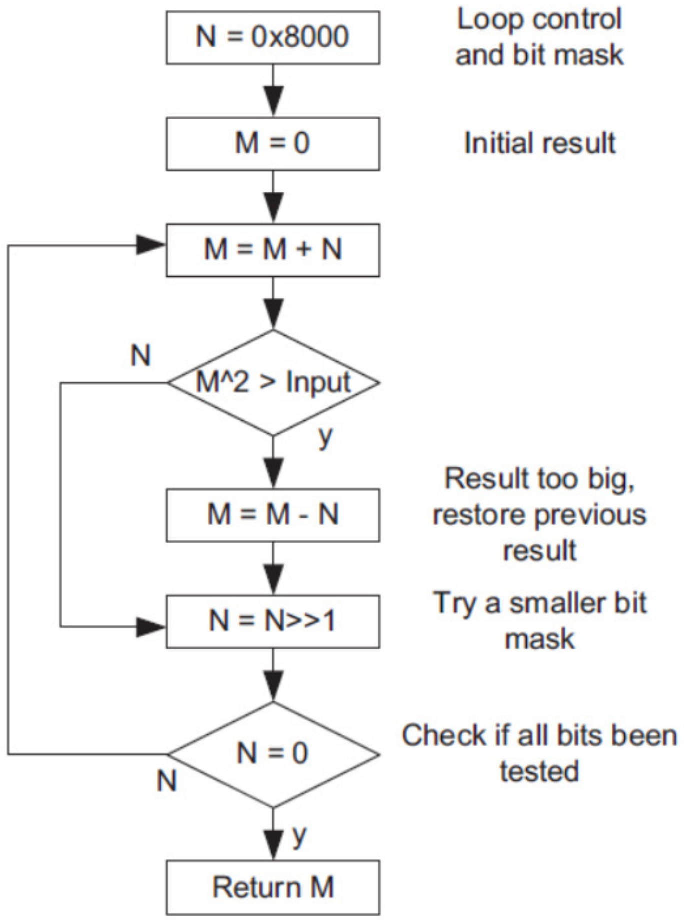
- Added to `asm`

```
    ; in demo_asm.s
     PRESERVE8 ; Preserve 8 byte stack alignment
    THUMB     ; indicate THUMB code is used
     AREA |.text|, CODE, READONLY; Start of the CODE area
     EXPORT simple_sqrt
simple_sqrt FUNCTION
     ; Input : R0
     ; Output : R0 (square root result)
     MOVW R1, #0x8000 ; R1 = 0x00008000
     MOVS R2, #0 ; Initialize result
simple_sqrt_loop
     ADDS R2, R2, R1 ; M = (M + N)
     MUL R3, R2, R2 ; R3 = M^2
     CMP R3, R0 ; If M^2 > Input
     IT HI     ; Greater Than
     SUBHI R2, R2, R1 ; M = (M - N)
     LSRS R1, R1, #1 ; N = N >> 1
     BNE simple_sqrt_loop
     MOV R0, R2 ; Copy to R0 and return
     BX LR     ; Return
     ENDFUNC
```

---

[1] Yiu Chapter 20, p. 664.

Function Flow Chart

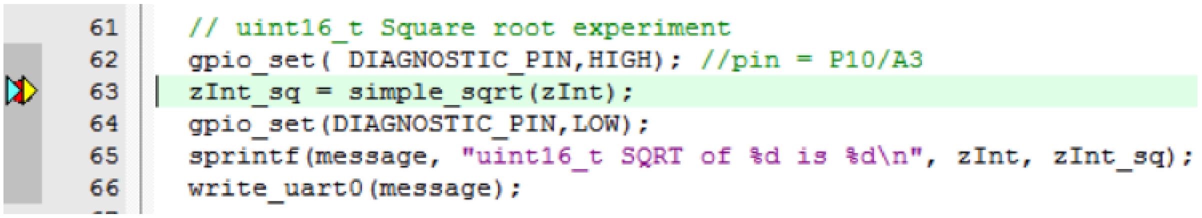


Sample Results

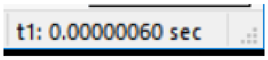
- For an input of 64 the output is 8, as expected

```
uint16_t SQRT of 64 is 8
```

From terminal



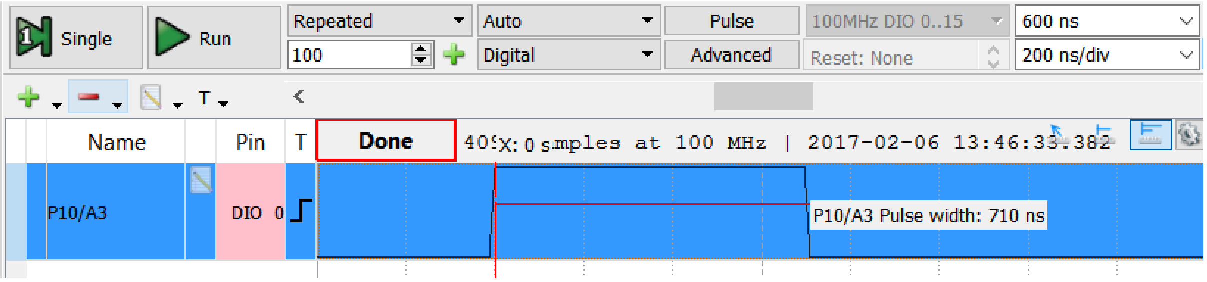
```
// uint16_t Square root experiment
gpio_set( DIAGNOSTIC_PIN,HIGH); //pin = P10/A3
zInt_sq = simple_sqrt(zInt);
gpio_set(DIAGNOSTIC_PIN,LOW);
sprintf(message, "uint16_t SQRT of %d is %d\n", zInt, zInt_sq);
write_uart0(message);
```



```
t1: 0.00000060 sec
```

Execution time = 0.60us (~600ns)

- Physical is comparable at 710ns:



- For an input of 99 the output is 9 (81 is closest to 99), as expected

```
uint16_t SQRT of 99 is 9
```

From terminal

Useful Resources

- Architecture Reference Manual:

http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0403c/index.html

- Cortex-M4 Technical Reference Manual:

http://infocenter.arm.com/help/topic/com.arm.doc.ddi0439d/DDI0439D_cortex_m4_processor_r0p1_trm.pdf

- Cortex-M4 Devices Generic User Guide:

http://infocenter.arm.com/help/topic/com.arm.doc.dui0553a/DUI0553A_cortex_m4_dgug.pdf