Lecture 5: ARM Cortex-M4 ISA & Assembly

Seyed-Hosein Attarzadeh-Niaki

Some slides due to ARM and Mazidi

Review

- Cortex M4 core and special registers
- Operating states and modes
- Memory system
- Debug and trace
- Reset and the reset sequence

Outline

- ARM Cortex-M4 ISA
  - ARM, Thumb, and Thumb 2 instructions
  - Load and store instructions
  - Stack operations
  - Arithmetic and logic instructions
  - Branch and call instructions
  - Other instructions
- ARM assembly language syntax
- Examples

ARM CORTEX-M4 ISA

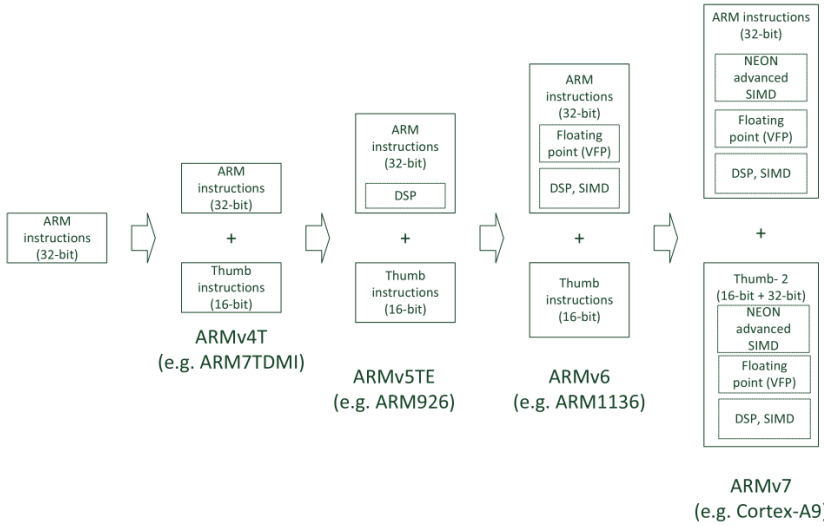
ARM, Thumb and Thumb-2 Instructions

- ARM instructions optimized for resource-rich high-performance computing systems
  - Deeply pipelined processor, high clock rate, wide (e.g. 32-bit) memory bus
- Low-end embedded computing systems are different
  - Slower clock rates, shallow pipelines
  - Different cost factors – e.g. code size matters much more, bit and byte operations critical
- Modifications to ARM ISA to fit low-end embedded computing
  - 1995: Thumb instruction set
    - 16-bit instructions
    - Reduces memory requirements but also performance
  - 2003: Thumb-2 instruction set
    - Adds some 32-bit instructions
    - Improves speed with little memory overhead
  - CPU decodes instructions based on whether in Thumb state or ARM state
    - Controlled by the T bit

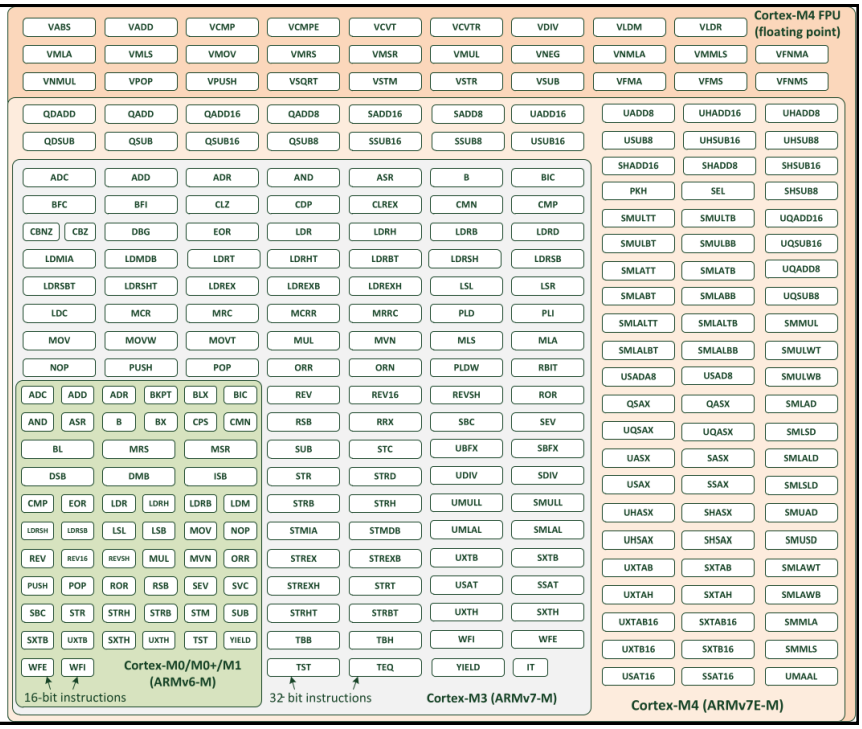
Instruction Set

- Cortex-M4 core implements ARMv7E-M Thumb instructions
- Only uses Thumb instructions, *always in Thumb state* (no switching)
  - Most instructions are 16 bits long, some are 32 bits
  - Most 16-bit instructions can only access low registers (R0-R7), but some can access high registers (R8-R15)
- Thumb state indicated by program counter being odd (LSB = 1)
  - Branching to an even address will cause an exception, since switching back to ARM state is not allowed
- Conditional execution supported for both 16-bit and 32-bit (B.W) branch
- 32 bit address space
- Half-word aligned instructions
- Upward compatible
- Refer to **ARMv7M Architecture Reference Manual** for specific instructions

Evolution of the ARM Instruction Set Architecture



Instruction set of the Cortex-M processors



Range of Instructions in Different Cortex-M Processors

Instruction Groups	Cortex-M0, M1	Cortex-M3	Cortex-M4	Cortex-M4 with FPU
16-bit ARMv6-M instructions	●	●	●	●
32-bit Branch with Link instruction	●	●	●	●
32-bit system instructions	●	●	●	●
16-bit ARMv7-M instructions		●	●	●
32-bit ARMv7-M instructions		●	●	●
DSP extensions			●	●
Floating point instructions				●

Assembler Instruction Format

- <operation> <operand1> <operand2> <operand3>
  - There may be fewer operands
  - First operand is typically destination (<Rd>) (Exception: memory write)
  - Other operands are sources (<Rn>, <Rm>)
- Examples
  - ADDS <Rd>, <Rn>, <Rm>
    - Add registers: <Rd> = <Rn> + <Rm>
  - AND <Rdn>, <Rm>
    - Bitwise and: <Rdn> = <Rdn> & <Rm>
  - CMP <Rn>, <Rm>
    - Compare: Set condition flags based on result of computing <Rn> - <Rm>

Where Can the Operands Be Located?

- In a general-purpose register R
  - Destination: Rd
  - Source: Rm, Rn
  - Both source and destination: Rdn
  - Target: Rt
  - Source for shift amount: Rs
- An immediate value encoded in instruction word
- In a condition code flag
- In memory
  - Only for load, store, push and pop instructions

Update Condition Codes in APSR?

31	30	29	28	27	26 … 20	19 … 16	15 … 0
N	Z	C	V	Q	Reserved	GE[3:0]	Reserved

- "S" suffix indicates the instruction updates APSR
  - ADD vs. ADDS
  - ADC vs. ADCS
  - SUB vs. SUBS
  - MOV vs. MOVS
- There are some instructions that update the APSR without explicitly adding S to them since their basic functions are to update the APSR
  - CMP
  - TST

Instruction Set Summary

Instruction Type	Instructions
Move	MOV
Load/Store	LDR, LDRB, LDRH, LDRSH, LDRSB, LDM, STR, STRB, STRH, STM
Add, Subtract, Multiply	ADD, ADDS, ADCS, ADR, SUB, SUBS, SBCS, RSBS, MULS
Compare	CMP, CMN
Logical	ANDS, EORS, ORRS, BICS, MVNS, TST
Shift and Rotate	LSLS, LSRS, ASRS, RORS
Stack	PUSH, POP
Conditional branch	IT, B, BL, B{cond}, BX, BLX
Extend	SXTH, SXTB, UXTH, UXTB
Reverse	REV, REV16, REVSH
Processor State	SVC, CPSID, CPSIE, BKPT
No Operation	NOP
Hint	SEV, WFE, WFI

Load/Store Register

- ARM is a load/store architecture, so must process data in registers, not memory
- **`LDR`**: load register from memory (32-bit)
  - LDR <Rt>, source address
- **`STR`**: store register to memory (32-bit)
  - STR <Rt>, destination address

Addressing Memory

- Offset Addressing mode: [<Rn>, <offset>] accesses address <Rn>+<offset>
  - Base Register <Rn>
  - <offset> is added or subtracted from base register to create effective address
    - Can be an immediate constant, e.g. #0x02
    - Can be another register, used as index <Rm>
- Auto-update (write back): Can write effective address back to base register- with an exclamation mark(!) at the back
- Pre-indexing: use effective address to access memory, then update base register with that effective address
- Post-indexing: use base register to access memory, then update base register with effective address

Addressing Modes

- Register addressing mode
  `ADD R1,R1,R3`
- Immediate addressing mode
  `ADD R6,R6,#0x40`
- Register Indirect Addressing Mode (Indexed addressing mode)
  `STR R5,[R6]`

Addressing Mode	Syntax	Effective Address of Memory	Rm Value After Execution
Pre-index	LDR Rd, [Rm, #k]	Rm + #k	Rm
Pre-index with WB*	LDR Rd, [Rm, #k]!	Rm + #k	Rm + #k
Post-index	LDR Rd, [Rm], #k	Rm	Rm + #k

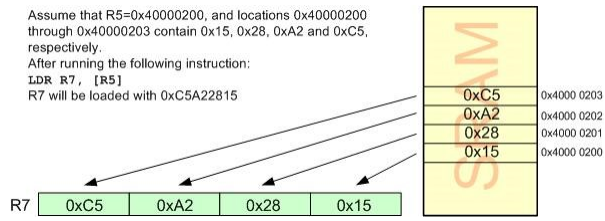
**WB means Writeback*
*** Rd and Rm are any of registers and #k is a signed 12-bit immediate value between -4095 and +4095*

Offset	Syntax	Pointing Location
Fixed value	LDR Rd, [Rm, #k]	Rm + #k
Shifted register	LDR Rd, [Rm, Rn, <shift>]	Rm + (Rn shifted <shift>)

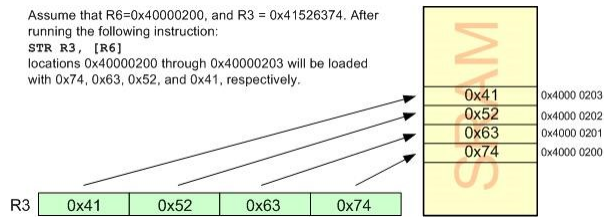
** Rn and Rm are any registers and #k is a signed 12-bit immediate value between -4095 and +4095*
*** <shift> is any of the shift operations studied in Chapter3 like LSL #2*

Load and Store Instructions in ARM

Assume that R5=0x40000200, and locations 0x40000200 through 0x40000203 contain 0x15, 0x28, 0xA2 and 0xC5, respectively.
After running the following instruction:
LDR R7, [R5]
R7 will be loaded with 0xC5A22815

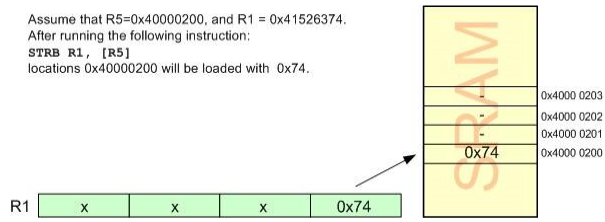


Assume that R6=0x40000200, and R3 = 0x41526374. After running the following instruction:
STR R3, [R6]
locations 0x40000200 through 0x40000203 will be loaded with 0x74, 0x63, 0x52, and 0x41, respectively.

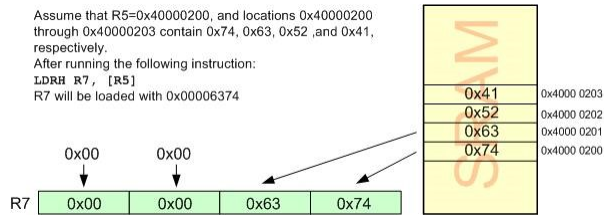


Load and Store Instructions in ARM

Assume that R5=0x40000200, and R1 = 0x41526374.
After running the following instruction:
STRB R1, [R5]
locations 0x40000200 will be loaded with 0x74.



Assume that R5=0x40000200, and locations 0x40000200 through 0x40000203 contain 0x74, 0x63, 0x52 ,and 0x41, respectively.
After running the following instruction:
LDRH R7, [R5]
R7 will be loaded with 0x00006374



Other Data Sizes

- Load and store instructions can also handle double-word (64 bits), half-word (16 bits), byte (8 bits), and even multiple word (n*32 bits)
- Store just writes to double-word half-word or byte without considering sign or unsigned.
  - STRH, STRB, STRD, STM
- Load a byte or half-word or double-word: What do we put in the upper bits?
- How do we extend 0x80 into a full word?
  - Unsigned? Then 0x80 = 128, so zero-pad to extend to word 0x0000_0080 = 128
  - Signed? Then 0x80 = -128, so sign-extend to word 0xFFFF_FF80 = -128

	Signed	Unsigned
Byte	LDRSB	LDRB
Half-word	LDRSH	LDRH

Data Size Extension

- Can extend byte or half-word already in a register
  - Signed or unsigned (zero-pad)
- How do we extend 0x80 into a full word?
  - Unsigned? Then 0x80 = 128, so zero-pad to extend to word 0x0000_0080 = 128
  - Signed? Then 0x80 = -128, so sign-extend to word 0xFFFF_FF80 = -128

	Signed	Unsigned
Byte	SXTB	UXTB
Half-word	SXTH	UXTH

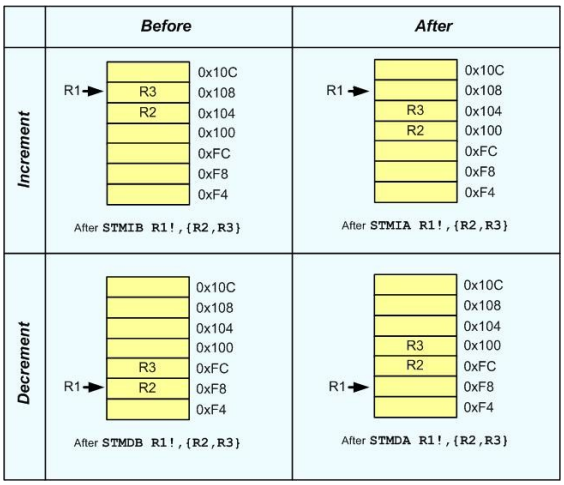
Data Size	Bits	2^n	Decimal	Hexadecimal
Byte	8	-2^7 to $+2^7-1$	-128 to +127	0x80–0x7F
Half-word	16	-2^{15} to $+2^{15}-1$	-32,768 to +32,767	0x8000–0x7FFF
Word	32	-2^{31} to $+2^{31}-1$	-2,147,483,648 to +2,147,483,647	0x80000000–0x7FFFFFFF

Load/Store Multiple

- LDM/LDMIA: load multiple registers starting from [base register], update base register afterwards
  - LDM <Rn>!,<registers>
  - LDM <Rn>,<registers>
- STM/STMIA: store multiple registers starting at [base register], update base register after
  - STM <Rn>!, <registers>
- LDMIA and STMIA are pseudo-instructions, translated by assembler
- Also, there are two counterparts LDMDB and STMDB: decrement before

Load/Store Multiple Registers

Option	Description
IA	Increment After
IB	Increment Before
DA	Decrement After
DB	Decrement Before



Load Literal Value into Register

- Assembly instruction: LDR <rd>, =value
  - Assembler generates code to load <rd> with value
- Assembler selects best approach depending on value
  - Load immediate
    - MOV instruction provides 8-bit unsigned immediate operand (0-255)
  - Load and shift immediate values
    - Can use MOV, shift, rotate, sign extend instructions
  - Load from literal pool
    1. Place value as a 32-bit literal in the program's literal pool (table of literal values to be loaded into registers)
    2. Use instruction LDR <rd>, [pc,#offset] where offset indicates position of literal relative to program counter value

```
LDR R0, =0x12345678 ; Set R0 to 0x12345678
```

```
LDR R0, [PC, #offset]
....
DCD 0x12345678
```

- Example formats for literal values (depends on compiler and toolchain)
  - Decimal: 3909
  - Hexadecimal: 0xa7ee
  - Character: 'A'
  - String: "44??"

Move (Pseudo-)Instructions

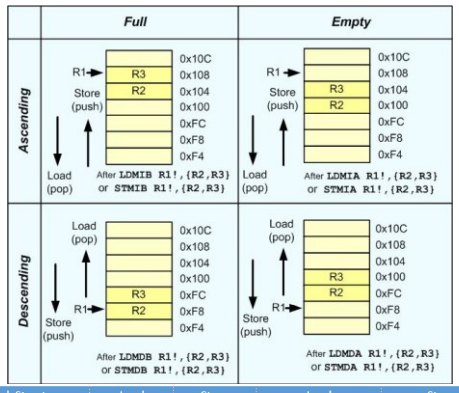
- Copy data from one register to another without updating condition flags
  - MOV <Rd>, <Rm>
- Assembler translates pseudo-instructions into equivalent instructions (shifts, rotates)
  - Copy data from one register to another and update condition flags
    - MOVS <Rd>, <Rm>
  - Copy immediate literal value (0-255) into register and update condition flags
    - MOVS <Rd>, #<imm8>

MOV instruction	Canonical form
MOVS <Rd>,<Rm>,ASR #<n>	ASRS <Rd>,<Rm>,#<n>
MOVS <Rd>,<Rm>,LSL #<n>	LSLS <Rd>,<Rm>,#<n>
MOVS <Rd>,<Rm>,LSR #<n>	LSRS <Rd>,<Rm>,#<n>
MOVS <Rd>,<Rm>,ASR <Rs>	ASRS <Rd>,<Rm>,<Rs>
MOVS <Rd>,<Rm>,LSL <Rs>	LSLS <Rd>,<Rm>,<Rs>
MOVS <Rd>,<Rm>,LSR <Rs>	LSRS <Rd>,<Rm>,<Rs>
MOVS <Rd>,<Rm>,ROR <Rs>	RORS <Rd>,<Rm>,<Rs>

Stack Operations

- Push some or all of registers to stack
  - PUSH {<registers>}
  - Decrements SP by 4 bytes for each register saved
  - Pushing LR saves return address
  - PUSH {r1, r2, LR}
- Pop some or all of registers from stack
  - POP {<registers>}
  - Increments SP by 4 bytes for each register restored
  - If PC is popped, then execution will branch to new PC value after this POP instruction (e.g. return address)
  - POP {r5, r6, r7}

Four General Stack Structures



Stack Structure	Load	Store	Load (alternate Names)	Store (alternate Names)
Full Ascending	LDMDA	STMIB	LDMFA	STMFA
Full Descending	LDMIA	STMDB	LDMFD	STMFD
Empty Ascending	LDMDB	STMIA	LDMEA	STMEA
Empty Descending	LDMIB	STMDA	LDMED	STMED

Add Instructions

- Add registers, update condition flags
  - **ADDS <Rd>,<Rn>,<Rm>**
- Add registers and carry bit, update condition flags
  - **ADCS <Rdn>,<Rm>**
- Add registers
  - **ADD <Rdn>,<Rm>**
- Add immediate value to register, update condition flags
  - **ADDS <Rd>,<Rn>,#<imm3>**
  - **ADDS <Rdn>,#<imm8>**

Add Instructions with Stack Pointer

- Add SP and immediate value
  - ADD <Rd>,SP,#<imm8>
  - ADD SP,SP,#<imm7>
- Add SP value to register
  - ADD <Rdm>, SP, <Rdm>
  - ADD SP,<Rm>

Address to Register Pseudo-Instruction

- Generate a PC-relative address in register
  - **ADR <Rd>,<label>**
  - Used in look-up tables, etc.

```
 ADR R0, DataTable
 …
 ALIGN
DataTable
 DCD 0, 245, 132, …
```

- How is this used?
  - ADR always assembles to one instruction.
  - The assembler attempts to produce a single ADD or SUB instruction to load the address.
  - If the address cannot be constructed in a single instruction, an error is generated and the assembly fails.
  - Use the ADRL pseudo-instruction to assemble a wider range of effective addresses.

Subtract

- Subtract immediate from register, update condition flags
  - SUBS <Rd>,<Rn>,#<imm3>
  - SUBS <Rdn>,#<imm8>
- Subtract registers, update condition flags
  - SUBS <Rd>,<Rn>,<Rm>
- Subtract registers with carry, update condition flags
  - SBCS <Rdn>,<Rm>
- Subtract immediate from SP
  - SUB SP,SP,#<imm7>

Multiply

- Multiply source registers, save lower word of result in destination register, update condition flags
  - **`MULS <Rd>, <Rn>, <Rm>`**
  - <Rd> = <Rn> * <Rm>
- Note: upper word of result is truncated
  - **`UMULL`** for 64-bit result

Multiplication	Operand 1	Operand 2	Result
word×word	Rm	Rs	RdHi= upper 32-bit,RdLo=lower 32-bit

Note: *Using SMULL (signed multiply long) for word × words multiplication provides the 64-bit result in RdLo and RdHi register. This is used for 32-bit × 32-bit numbers in which result can go beyond 0xFFFFFFFF.*

Logical Operations

- Bitwise AND registers, update condition flags
  - **`ANDS <Rdn>,<Rm>`**
- Bitwise OR registers, update condition flags
  - **`ORRS <Rdn>,<Rm>`**
- Bitwise Exclusive OR registers, update condition flags
  - **`EORS <Rdn>,<Rm>`**
- Bitwise AND register and complement of second register, update condition flags
  - **`BICS <Rdn>,<Rm>`**
- Move inverse of register value to destination, update condition flags
  - **`MVNS <Rd>,<Rm>`**
- Update condition flags by ANDing two registers, discarding result
  - **`TST <Rn>, <Rm>`**

Compare

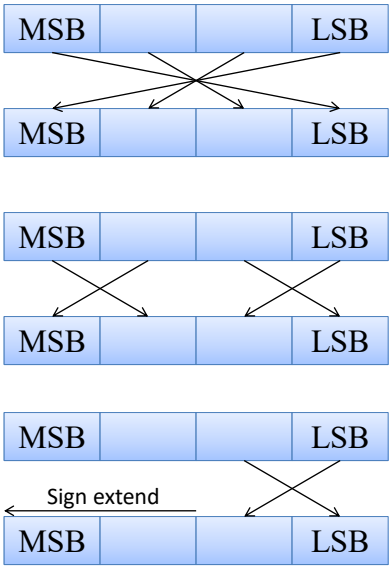
- Compare - subtracts second value from first, discards result, updates APSR
  - **CMP <Rn>,#<imm8>**
  - **CMP <Rn>,<Rm>**
- Compare negative - adds two values, updates APSR, discards result
  - **CMN <Rn>,<Rm>**

Shift and Rotate

- Common features
  - All of these instructions update APSR condition flags
  - Shift/rotate amount (in number of bits) specified by last operand
- Logical shift left - shifts in zeroes on right
  - **LSLS <Rd>,<Rm>,#<imm5>**
  - **LSLS <Rdn>,<Rm>**
- Logical shift right - shifts in zeroes on left
  - **LSRS <Rd>,<Rm>,#<imm5>**
  - **LSRS <Rdn>,<Rm>**
- Arithmetic shift right - shifts in copies of sign bit on left (to maintain arithmetic sign)
  - **ASRS <Rd>,<Rm>,#<imm5>**
- Rotate right
  - **RORS <Rdn>,<Rm>**

Reversing Bytes

- REV - reverse all bytes in word
  - **`REV <Rd>,<Rm>`**
- REV16 - reverse bytes in both half-words
  - **`REV16 <Rd>,<Rm>`**
- REVSH - reverse bytes in low half-word (signed) and sign-extend
  - **`REVSH <Rd>,<Rm>`**



Bit-Field Processing Instructions

Instruction	Operation
BFC Rd, #<lsb>, #<width>	Clear bit field within a register
BFI Rd, Rn, #<lsb>, #<width>	Insert bit field to a register
CLZ Rd, Rm	Count leading zero
RBIT Rd, Rn	Reverse bit order in register
SBFX Rd, Rn, #<lsb>, #<width>	Copy bit field from source and sign extend it
UBFX Rd, Rn, #<lsb>, #<width>	Copy bit field from source register

- Example:

```
LDR R0,=0x1234FFFF
BFC R0, #4, #8        ; R0=0x1234F00F
```

- Example:

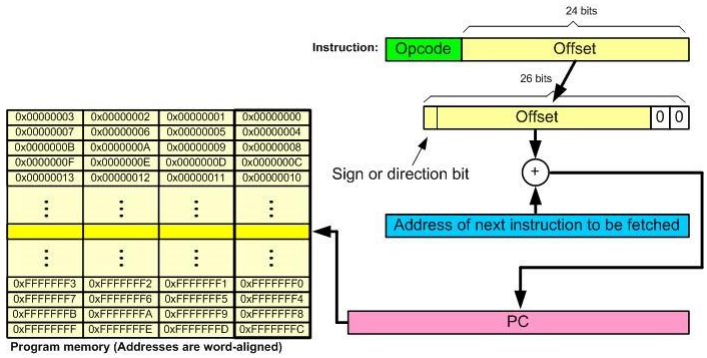
```
LDR R0,=0x5678ABCD
UBFX R1, R0, #4, #8 ; R1=0x000000BC
```

Changing Program Flow Branches

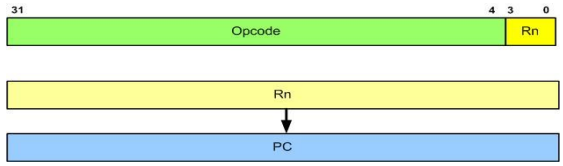
- Unconditional Branches
  - B <label>
  - Target address must be within 2 KB of branch instruction (-2048 B to +2046 B)
- Conditional Branches
  - **B<cond> <label>**
    - <cond> is condition - see next pages
  - B<cond> target address must be within 256 B of branch instruction (-256 B to +256 B)
  - Alternatively, can use the B.W as 32-bit version of branch instruction for wider range.

Branch Instructions (wide version)

B (Branch)



BX (Branch and exchange)



Condition Codes

- Append to branch instruction (B) to make a conditional branch
- Full ARM instructions (not Thumb or Thumb-2) support conditional execution of arbitrary instructions
- Note: Carry bit = not-borrow for compares and subtractions

Mnemonic extension	Meaning	Condition flags
EQ	Equal	Z = 1
NE	Not equal	Z = 0
CS [a]	Carry set	C = 1
CC [b]	Carry clear	C = 0
MI	Minus, negative	N = 1
PL	Plus, positive or zero	N = 0
VS	Overflow	V = 1
VC	No overflow	V = 0
HI	Unsigned higher	C = 1 and Z = 0
LS	Unsigned lower or same	C = 0 or Z = 1
GE	Signed greater than or equal	N == V
LT	Signed less than	N != V
GT	Signed greater than	Z = 0 and N == V
LE	Signed less than or equal	Z = 1 or N != V
None (AL) [d]	Always (unconditional)	Any

ARM Conditional Branch (Jump) Instructions for Signed Data

Instruction		Action
BEQ	Branch equal	Branch if Z = 1
BNE	Branch not equal	Branch if Z = 0
BMI	Branch minus (branch negative)	Branch if N = 1
BPL	Branch plus (branch positive)	Branch if N = 0
BVS	Branch if V set (branch overflow)	Branch if V = 1
BVC	Branch if V clear (branch if no overflow)	Branch if V = 0
BGE	Branch greater than or equal	Branch if N = V
BLT	Branch less than	Branch if N ≠ V
BGT	Branch greater than	Branch if Z = 0 and N = V
BLE	Branch less than or equal	Branch if Z = 1 or N ≠ V

Changing Program Flow - Subroutines

- Call
  - **`BL <label>`** - branch with link
    - Call subroutine at <label>
      - PC-relative, range limited to PC+/-16MB
    - Save return address in LR
  - **`BLX <Rd>`** - branch with link and exchange
    - Call subroutine at address in register Rd
      - Supports full 4GB address range
    - Save return address in LR
- Return
  - **`BX <Rd>`** branch and exchange
    - Branch to address specified by <Rd>
    - Supports full 4 GB address space
    - **`BX LR`** - Return from subroutine

Branch Instructions (wide version)

BL (Branch and Link)

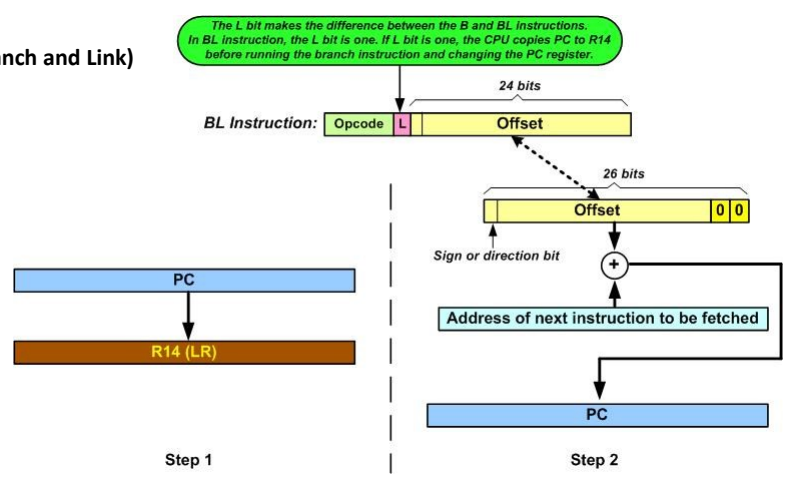
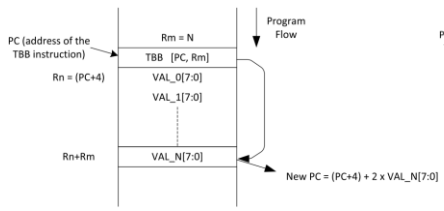
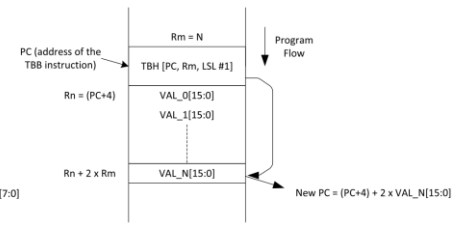


Table Branches

- Used to implement switch statements in C
  - Table Branch Byte
    - **`TBB [Rn, Rm]`**
  - Table Branch Half-word
    - **`TBH [Rn, Rm, LSL #1]`**





Special Register Instructions

- Move to Register from Special Register
  - **`MSR <Rd>, <spec_reg>`**
- Move to Special Register from Register
  - **`MRS <spec_reg>, <Rd>`**
- Change Processor State - Modify PRIMASK register
  - **`CPSIE`** - Interrupt enable
  - **`CPSID`** - Interrupt disable

Special register	Contents
APSR	The flags from previous instructions.
IAPSR	A composite of IPSR and APSR.
EAPSR	A composite of EPSR and APSR.
XPSR	A composite of all three PSR registers.
IPSR	The Interrupt status register.
EPSR	The execution status register.[b]
IEPSR	A composite of IPSR and EPSR.
MSP	The Main Stack pointer.
PSP	The Process Stack pointer.
PRIMASK	Register to mask out configurable exceptions.[c]
CONTROL	The CONTROL register, see *The special-purpose CONTROL register* on page B1-215.

More Instructions In The Reference Manual

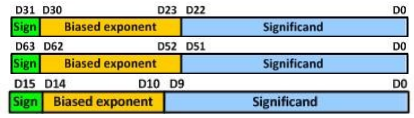
- Saturation operations
- Sleep mode-related instructions
- Memory barrier instructions
- Enhanced DSP extension instructions
  - SIMD, MAC, (un)packing, floating-point

Other Instructions

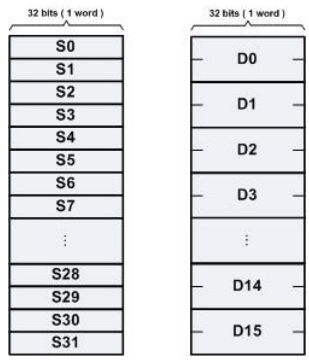
- No Operation - does nothing!
  - **NOP**
- Breakpoint - causes hard fault or debug halt - used to implement software breakpoints
  - **BKPT #<imm8>**
- Wait for interrupt - Pause program, enter low-power state until a WFI wake-up event occurs (e.g. an interrupt)
  - **WFI**
- Supervisor call generates SVC exception (#11), same as software interrupt
  - **SVC #<imm>**

ARM Floating Point

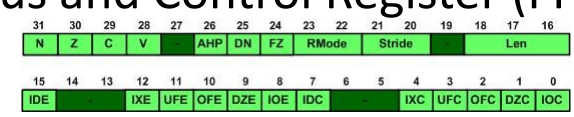
IEEE 754 Single-precision Floating-point Numbers
IEEE 754 Double-precision Floating-point Numbers
IEEE 754 Half-precision Floating-point Numbers



ARM Floating-point data Register Bank



ARM Floating-point Status and Control Register (FPSCR)



Bits	Name	Function
31-28	N, Z, C, V	Negative, Zero, Carry, Overflow flags
25	DN	Default NaN mode control
24	FZ	Flush-to-zero mode control
23-22	RMode	Rounding Mode control
21-20	Stride	Step size in vector
18-16	Len	Length of the vector
15, 12-8		Exception trap enable bits IDE — Input subnormal exception enable IXE — Inexact exception enable UFE — Underflow exception enable OFE — Overflow exception enable DZE — Division by zero exception enable IOE — Invalid operation exception enable
7, 4-0		Cumulative exception bits IDC — Input subnormal exception flag IXC — Inexact exception flag UFC — Underflow exception flag OFC — Overflow exception flag DZC — Division by zero exception flag IOC — Invalid operation exception flag

Floating-point Data Processing Instructions

Mnemonic	Function	Description
VABS	Absolute	Obtain the absolute value of the operand
VNEG	Negate	Negate the value of the operand
VSQRT	square root	Obtain the square root of the operand
VADD	Add	Add the operands
VSUB	Subtract	Subtract the second operand from the first operand
VDIV	Divide	Divide the first operand by the second operand
VMUL	Multiply	Multiply the two operands
VNMUL	multiply negate	Multiply the two operands then negate the result
VMLA	multiply and accumulate	Multiply the two operands then add the result to the destination register and store the final result in the destination register
VNMLA	multiply and accumulate negate	Multiply the two operands then add the result to the destination register, negate the final result and store it in the destination register
VMLS	multiply and subtract	Multiply the two operands then subtract the result from the destination register and store the final result in the destination register
VNMLS	multiply and subtract negate	Multiply the two operands then subtract the result from the destination register, negate the final result and store it in the destination register
VFMA	fused multiply and accumulate	Same as VMLA except using fused operation (single rounding at the final result)
VFMS	fused multiply and subtract	Same as VMLS except using fused operation
VFNMA	fused multiply and accumulate negate	Same as VNMLA except using fused operation
VFNMS	fused multiply and subtract negate	Same as VNMLS except using fused operation
VCMP	Compare	Subtract the second operand from the first operand and set the NZCV bits of FPSCR

ARM ASSEMBLY LANGUAGE SYNTAX

Programming Languages

- Machine language
  - Binary code, for CPU but not human beings
- Assembly language
  - Mnemonics for machine code instructions
  - *Low-level language:* deals with the internal structure of a CPU
  - Hard to program, poor portability but very efficient
- BASIC, Pascal, C, Fortran, Perl, TCL, Python, …
  - *High-level languages:* do not have to be concerned with the internal details of a CPU
  - Easy to program, good portability but less efficient

How to convert your program in low/high-level languages into machine language?

Understanding Different Assembly Language Syntaxes

ARM (armasm)

```
NVIC_IRQ_SETEN   EQU 0xE 000E100
NVIC_IRQ0_ENABLE EQU 0x1
 …
 LDR R0,=NVIC_IRQ_SETEN     ; Put 0xE000E100 into R0
     ; LDR here is a pseudo instruction that will be converted
     ; to a PC relative literal data load by the assembler
 MOVS R1, #NVIC_IRQ0_ENABLE ; Put immediate data (0x1) into
                            ; register R1
 STR R1, [R0] ; Store 0x1 to 0xE000E100, this enable external
              ; interrupt IRQ#0
```

GNU

```
 .equ NVIC_IRQ_SETEN,   0xE000E100
 .equ NVIC_IRQ0_ENABLE, 0x1
 …
 LDR R0,=NVIC_IRQ_SETEN /* Put 0xE000E100 into R0
     LDR here is a pseudo instruction that will be
     converted to a PC relative load by the assembler */
 MOVS R1, #NVIC_IRQ0_ENABLE /* Put immediate data (0x1) into
                                 register R1 */
 STR R1, [R0] /* Store 0x1 to 0xE000E100, this enable
                 external interrupt IRQ#0 */
```

```
 LDR R3,=MY_NUMBER ; Get the memory location of MY_NUMBER
 LDR R4, [R3]      ; Read the value 0x12345678 into R4
 …
 LDR R0,=HELLO_TEXT ; Get the starting address of HELLO_TEXT
 BL  PrintText      ; Call a function called PrintText to
                    ; display string
 …
 ALIGN 4
MY_NUMBER  DCD 0x12345678
HELLO_TEXT DCB "Hello\n", 0 ; Null terminated string
```

```
 LDR R3,=MY_NUMBER /* Get the memory location of MY_NUMBER */
 LDR R4, [R3]      /* Read the value 0x12345678 into R4 */
 …
 LDR R0,=HELLO_TEXT /* Get the starting address of
                       HELLO_TEXT */
 BL  PrintText      /* Call a function called PrintText to
                       display string */
 …
 .align 4
MY_NUMBER:
 .word 0x12345678
HELLO_TEXT:
 .asciz "Hello\n"  /* Null terminated string */
```

Assembler Instruction Format

- How to make sense of the reference manual?

mnemonic

Conditional Execution in "IT block"

Destination and Source Register

Encoding T3 ARMv7-M

ADD{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}

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	0	1	0	1	1	0	0	0	S	Rn				(0)	imm3			Rd				imm2		type		Rm			

```
if Rd == '1111' && S == '1' then SEE CMN (register);
if Rn == '1101' then SEE ADD (SP plus register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d IN {13,15} || n == 15 || m IN {13,15} then UNPREDICTABLE;
```

"S" updates APSR condition flags

if ".W" (wide), assembler must select a 32-bit

Constant Shifts, uses "type"

Commonly Used Directives for Inserting Data Into a Program

Type of Data to Insert	ARM Assembler (e.g., Keil MDK-ARM)	GNU Assembler
Byte	DCB E.g., DCB 0x12	.byte E.g., .byte 0x012
Half-word	DCW E.g., DCW 0x1234	.hword / .2byte E.g., .hword 0x01234
Word	DCD E.g., DCD 0x01234567	.word / .4byte E.g., .word 0x01234567
Double-word	DCQ E.g., DCQ 0x12345678FF0055AA	.quad/.octa E.g., .quad 0x12345678FF0055AA
Floating point (single precision)	DCFS E.g., DCFS 1E3	.float E.g., .float 1E3
Floating point (double precision)	DCFD E.g., DCFD 3.14159	.double E.g., .double 3f14159
String	DCB E.g., DCB "Hello\n" 0,	.ascii / .asciz (with NULL termination) E.g., .ascii "Hello\n" .byte 0 /* add NULL character */ E.g., .asciz "Hello\n"
Instruction	DCI E.g., DCI 0xBE00 ; Breakpoint (BKPT 0)	.word / .hword E.g., .hword 0xBE00 /* Breakpoint (BKPT 0) */

Commonly Used Directives

Directive (GNU assembler equivalent)	ARM Assembler
THUMB (.thumb)	Specify assembly code as Thumb instruction in Unified Assembly Language (UAL) format.
CODE16 (.code 16)	Specify assembly code as Thumb instruction in legacy pre-UAL syntax.
AREA <section_name>{,<attr>}{,attr}... (.section <section_name>)	Instructs the assembler to assemble a new code or data section. Sections are independent, named, indivisible chunks of code or data that are manipulated by the linker.
SPACE <*num of bytes*> (.zero <*num of bytes*>)	Reserves a block of memory and fills it with zeros.
FILL <*num of bytes*>{, <*value*>{, <*value_sizes*>}} (.fill <*num of bytes*>{, <*value*>{, <*value_sizes*>}})	Reserves a block of memory and fills it with the specified value. The size of the value can be byte, half-word, or word, specified by value_sizes (1/2/4).
ALIGN {<*expr*>{,<*offset*>{,<*pad*>{,<*padsize*>}}}} (.align <alignment>{,<fill>{,<max}}})	Aligns the current location to a specified boundary by padding with zeros or NOP instructions. E.g., ALIGN 8 ; make sure the next instruction or ; data is aligned to 8 byte boundary
EXPORT <*symbol*> (.global <symbol>)	Declare a symbol that can be used by the linker to resolve symbol references in separate object or library files.
IMPORT <*symbol*>	Declare a symbol reference in separate object or library files that is to be resolved by linker.
LTORG (.pool)	Instructs the assembler to assemble the current literal pool immediately. Literal pool contains data such as constant values for LDR pseudo instruction.

Unified Assembly Language (UAL)

- Before Thumb 2, Thumb instructions were more relaxed
  - e.g., all instructions update APSR and the "S" suffix was not necessary
  - If destination and source registers are the same, one is omitted
- In UAL this has changed
  - You can also choose between 16-bit and 32-bit versions

```
ADDS R0, #1   ; Use 16-bit Thumb instruction by default
              ; for smaller size
ADDS.N R0, #1 ; Use 16-bit Thumb instruction (N=Narrow)
ADDS.W R0, #1 ; Use 32-bit Thumb-2 instruction (W=wide)
```

- 32-bit Thumb-2 instructions can be half-word aligned.

Suffixes for Cortex-M Assembly Language

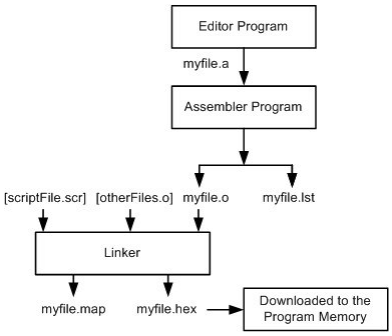
Suffixes	Descriptions
S	Update APSR (Application Program Status Register, such as Carry, Overflow, Zero and Negative flags); for example: ADDS R0, R1 ; this ADD operation will update APSR
EQ, NE, CS, CC, MI, PL, VS, VC, HI, LS, GE, LT, GT, LE	Conditional execution. EQ = Equal, NE = Not Equal, LT = Less Than, GT = Greater Than, etc. On the Cortex-M processors these conditions can be applied to conditional branches; for example: `BEQ label` ; Branch to label if previous operation result in ; equal status or conditionally executed instructions (see IF-THEN instruction in section 5.6.9); for example: `ADDEQ R0, R1, R2` ; Carry out the add operation if the previous ; operation results in equal status
.N, .W	Specify the use of 16-bit (narrow) instruction or 32-bit (wide) instruction.
.32, .F32	Specify the operation is for 32-bit single-precision data. In most toolchains, the .32 suffix is optional.
.64, F64	Specify the operation is for 64-bit double-precision data. In most toolchains, the .64 suffix is optional.

Data Format Representation

- Hex numbers
  `MOV R1,#0x99`
- Decimal numbers
  `MOV R7,#12`
- Binary numbers
  `MOV R6,#2_10011001`
- Numbers in any base between 2 and 9
  `MOV R7,#8_33`
- ASCII characters
  `LDR R3,#'2'`

Assembling an ARM Program

- The map file shows the labels defined in the program together with their values.
- The lst (list) file shows the binary and source code and the amount of memory the program uses.

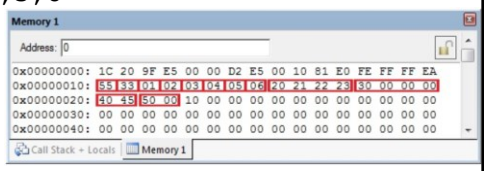


A FEW ARM ASSEMBLY EXAMPLES

```
;ARM Assembly Language Program To Add
Some Data and Store the SUM in R3.
   AREA PROG_2_1, CODE, READONLY
   ENTRY
   MOV  R1, #0x25   ;R1 = 0x25
   MOV  R2, #0x34   ;R2 = 0x34
   ADD  R3, R2, R1  ;R3 = R2 + R1
HERE
   B    HERE        ;stay here forever
END
```


Storing Constants in Memory

```
   ;storing data in program memory.
   AREA  LOOKUP_EXAMPLE,READONLY,CODE
   ENTRY
   LDR   R2,=OUR_FIXED_DATA ;point to OUR_FIXED_DATA
   LDRB  R0,[R2]      ;load R0 with the contents
                      ;of memory pointed to by R2
   ADD   R1, R1, R0   ;add R0 to R1
HERE
   B  HERE            ;stay here forever
OUR_FIXED_DATA
   DCB   0x55,0x33,1,2,3,4,5,6
   DCD   0x23222120,0x30
   DCW   0x4540,0x50
END
```



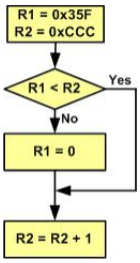
If Example

Pseudo code:

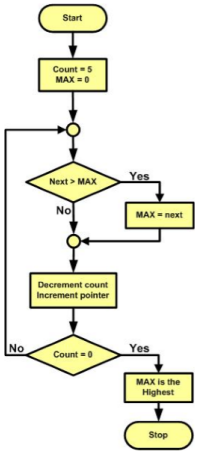
```
R1 = 0x35F
R2 = 0xCCC

IF (R1 >= R2) THEN
        R1 = 0
ENDIF

R2 = R2 + 1
```

In C:

```
R1 = 0x35F;
R2 = 0xCCC;

if (R1 >= R2)
{
        R1 = 0;
}

R2 = R2 + 1;
```



Loop Example



Pseudo code:

```
Count = 5
Highest = 0

REPEAT
   IF (Next > MAX)
   THEN
         MAX = Next
   ENDIF
   Increment pointer
   Decrement Count

UNTIL Count = 0

; now MAX is the Highest
```

In C:

```
//In Keil, int is 32-bit wide

unsigned int myData[5]= {69, 87, 96, 45, 75};
unsigned int count = 5;
unsigned int max = 0;
unsigned int next;
unsigned int *pointer = myData;

do
{
  next = *pointer;

  if (max < next)
    max = next;

  pointer ++;
  count --;
} while(count != 0);
```

Loop Example

```
;searching for highest value
COUNT     RN   R0   ;COUNT is the new name of R0
MAX       RN   R1   ;MAX is the new name of R1
     ;(MAX has the highest value)
POINTER   RN   R2   ;POINTER is the new name of R2
NEXT      RN   R3   ;NEXT is the new name of R3

     AREA PROG_4_1D, DATA, READONLY
MYDATA    DCD  69,87,96,45,75
     AREA PROG_4_1, CODE, READONLY
     ENTRY
          MOV  COUNT,#5  ;COUNT = 5
          MOV  MAX,#0    ;MAX = 0
          LDR  POINTER,=MYDATA    ;POINTER = MYDATA ( address of first data )
AGAIN     LDR  NEXT,[POINTER]     ;load contents of POINTER location to NEXT
          CMP  MAX,NEXT           ;compare MAX and NEXT
          BHS  CTNU               ;if MAX > NEXT branch to CTNU
          MOV  MAX,NEXT           ;MAX = NEXT
CTNU      ADD  POINTER,POINTER,#4 ;POINTER=POINTER+4 to point to the next
          SUBS COUNT,COUNT,#1     ;decrement counter
BNE       AGAIN                   ;branch AGAIN if counter is not zero
HERE      B    HERE
END
```


ARM Assembly Main Program that Calls Subroutines

```
      ; MAIN program calling subroutines
      AREA  PogramName, CODE, READONLY

MAIN  BL    SUBR_1            ; Call Subroutine 1
      BL    SUBR_2            ; Call Subroutine 1
      BL    SUBR_3            ; Call Subroutine 1
HERE  BAL   HERE        ; stay here. BAL is the same as B
      ; -------end of MAIN

      ; --------------------SUBROUTINE 1
SUBR_1            ....
            ....
      BX    LR    ; return to main
      ; ------    end of subroutine 1

      ; --------------------SUBROUTINE 2
SUBR_2      ....
      ....
      BX    LR    ; return to main
      ; ------    end of subroutine 2

      ; --------------------SUBROUTINE 3
SUBR_3      ....
      ....
      BX    LR    ; return to main
      ; ------    end of subroutine 3
      END         ; notice the END of file
```

Bit-addressable (bit-band) SRAM

SRAM Byte addresses	SRAM Bit addresses (We use these addresses to access the individual bits)							
	D7	D6	D5	D4	D3	D2	D1	D0
200FFFFF	23FFFFFC	F8	F4	F0	EC	E8	E4	23FFFFE0
200FFFFE	23FFFFDC	D8	D4	D0	CC	C8	C4	23FFFFC0
200FFFFD	23FFFFBC	B8	B4	B0	AC	A8	A4	23FFFFA0
200FFFFC	23FFFF9C	98	94	90	8C	88	84	23FFFF80
200FFFFB	23FFFF7C	78	74	70	6C	68	64	23FFFF60
200FFFFA	23FFFF5C	x8	x4	x0	xC	x8	x4	23FFFF40
200XXXXX	2XXXXXXC	X8	X4	X0	XC	X8	X4	2XXXXXX0
20000008	2200011C	118	114	...		104		22000100
20000007	220000FC	F8	F4	F0	EC	E8	E4	220000E0
20000006	220000DC	D8	D4	D0	CC	C8	C4	220000C0
20000005	220000BC	B8	B4	B0	AC	A8	A4	220000A0
20000004	2200009C	98	94	90	8C	88	84	22000080
20000003	2200007C	78	74	70	6C	68	64	22000060
20000002	2200005C	58	54	50	4C	48	44	20000040
20000001	2200003C	38	34	30	2C	28	24	22000020
20000000	2200001C	18	14	10	0C	08	04	22000000

Peripherals bit-addressable region and their alias addresses

Peripherals Byte addresses	Peripherals Bit addresses. (We use these addresses to access the individual bits)							
	D7	D6	D5	D4	D3	D2	D1	D0
400FFFFF	43FFFFFC	F8	F4	F0	EC	E8	E4	43FFFFE0
400FFFFE	43FFFFDC	D8	D4	D0	CC	C8	C4	43FFFFC0
400FFFFD	43FFFFBC	B8	B4	B0	AC	A8	A4	43FFFFA0
400FFFFC	43FFFF9C	98	94	90	8C	88	84	43FFFF80
400FFFFB	43FFFF7C	78	74	70	6C	68	64	43FFFF60
400FFFFA	43FFFF5C	x8	x4	x0	xC	x8	x4	43FFFF40
400XXXXX	4XXXXXXC	X8	X4	X0	XC	X8	X4	4XXXXXX0
40000008	4200011C	118	114	...		104		42000100
40000007	420000FC	F8	F4	F0	EC	E8	E4	420000E0
40000006	420000DC	D8	D4	D0	CC	C8	C4	420000C0
40000005	420000BC	B8	B4	B0	AC	A8	A4	420000A0
40000004	4200009C	98	94	90	8C	88	84	42000080
40000003	4200007C	78	74	70	6C	68	64	42000060
40000002	4200005C	58	54	50	4C	48	44	42000040
40000001	4200003C	38	34	30	2C	28	24	42000020
40000000	4200001C	18	14	10	0C	08	04	42000000

Example: set HIGH the D6 of the SRAM location 0x20000001

- Byte address

```
LDR     R1,=0x20000001          ;load the address of the byte
LDRB    R2,[R1]                 ;get the byte
ORR     R2,R2,#2_01000000       ;make D6 bit high
;(binary representation in Keil for 0b01000000)
STRB    R2,[R1]                 ;write it back
```

- Bit alias address

```
LDR     R1,=0x22000038          ;load the alias address of the bit
MOV     R2,#1                   ;R2 = 1
STRB    R2,[R1]                 ;Write one to D6
```