

EE3220 System-on-Chip Design

Tutorial Note 4 – ARM Instruction Set

0. Objective

In this tutorial, we will learn to read official instruction set and get familiar with the basic functions of ARM target. From the instruction, learn the whole usage and functions and apply these to your applications. The objectives are shown as following:

- Learn basic instructions and syntax of ARM from the official
- Learn the usage of official instruction
- Learn the instructions examples in assembly code
- From this tutorial, learn to get familiar with a new target through general instructions

Content

1. Instruction
2. ARM Programmer Model
3. ARM Instruction Set
4. Calculations
5. Conclusion
6. References

I. Instruction

CPU Pipeline:

- Pipelining is an implementation technique whereby **multiple instructions** are overlapped in execution; it takes advantage of **parallelism** that exists among the actions needed to execute an instruction.
- In a computer pipeline, each step in the pipeline completes a part of an instruction. Each of these steps is called a **pipe stage** or a **pipe segment**.
- The stages are connected one to the next to form a pipe. Instructions enter at one end, progress through the stages, and exit at the other end

I. Instruction

Simple RISC pipeline

Instruction number	Clock number								
	1	2	3	4	5	6	7	8	9
Instruction i	IF	ID	EX	MEM	WB				
Instruction $i + 1$		IF	ID	EX	MEM	WB			
Instruction $i + 2$			IF	ID	EX	MEM	WB		
Instruction $i + 3$				IF	ID	EX	MEM	WB	
Instruction $i + 4$					IF	ID	EX	MEM	WB

IF = Instruction fetch
ID = Instruction decode
EX = Execution
MEM = Memory access
WB = Write-back

- On each clock cycle, another instruction is fetched and begins its **five-cycle** execution.
- If an instruction is started every clock cycle, the performance will be up to **five times** that of a processor that is not pipelined.

I. Instruction

- The original (and subsequent) Arm implementation was hardwired without microcode, like the much simpler 8-bit 6502 processor used in prior Acorn microcomputers.
- A **reduced instruction set computer**, or **RISC**, is a computer with a small, highly optimized set of instructions, rather than the more specialized set often found in other types of architecture, such as in a complex instruction set computer (CISC)
- For example, the 32-bit Arm architecture includes RISC features.

I. Instruction

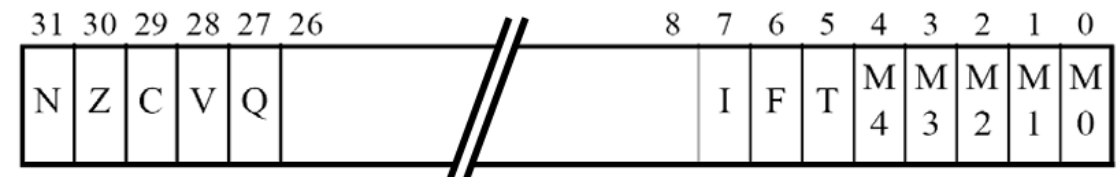
Features:

- Load/store architecture.
- No support for unaligned memory accesses in the original version of the architecture. Armv6 and later, except some microcontroller versions, support unaligned accesses for half-word and single-word load/store instructions with some limitations, such as no guaranteed atomicity.
- Uniform 16×32 -bit register file (including the program counter, stack pointer and the link register).
- Fixed instruction width of 32 bits to ease decoding and pipelining, at the cost of decreased code density. Later, the Thumb instruction set added 16-bit instructions and increased code density.
- Mostly single clock-cycle execution.
- The content we discuss is applied on assembly level

2. ARM Programmer Model

- The state of an ARM system is determined by the content of visible registers and memory.
- A user-mode program can see 15 32-bit general purpose registers (R0-R14), program counter (PC) and CPSR.
- Instruction set defines the operations that can change the state.

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



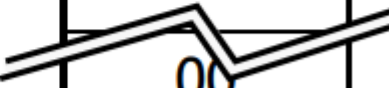
2.ARM Programmer Model

- Among these 16 registers, R13 and R14 have special functions.
- R13 is stack pointer (SP) as a pointer to active stack. A stack pointer is a small register that stores the address of the last program request in a stack. A stack is a specialized buffer which stores data from the top down. As new requests come in, they "push down" the older ones.
- R14 is link register used to store the return address from a subroutine. At other times, LR can be used for other purposes.

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

2. ARM Programmer Model

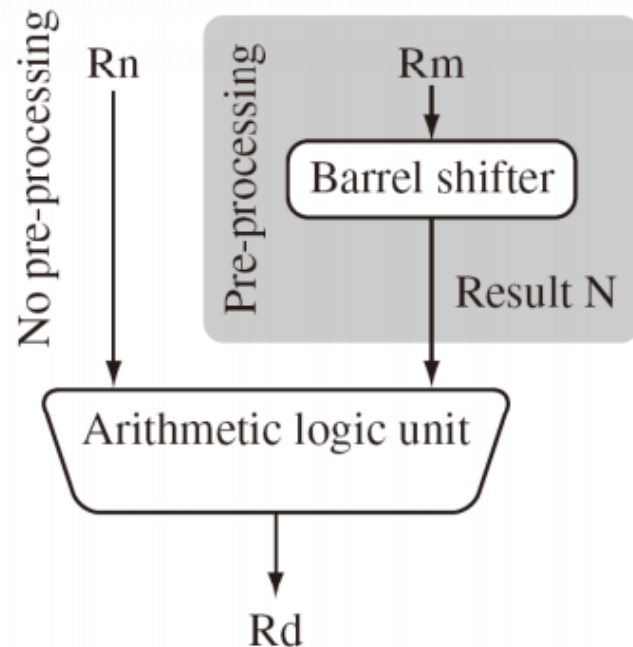
- Memory is a linear array from 0 to $(2^{32}-1)$
- The content with the format of word, half-word and byte can be saved in the address.
- The structure is shown on the right.

0x00000000	00
0x00000001	10
0x00000002	20
0x00000003	30
0x00000004	FF
0x00000005	FF
0x00000006	FF
	
0xFFFFFFF	00
0xFFFFFFF	00
0xFFFFFFF	00
0xFFFFFFF	

3. ARM Instruction Set

General Rules:

- All operands are 32-bit, coming from registers or literals.
- The result, if any, is 32-bit and placed in a register (with the exception for long multiply which produces a 64-bit result) –
- 3-address format



3	3	2	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	1	1	1	9	8	7	6	5	4	3	2	1	0		
1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0																
Cond	0	0	1	Opcode				S	Rn				Rd				Operand 2																Data Processing / PSR Transfer				
Cond	0	0	0	0	0	0	0	A	S	Rd				Rn				Rs				1	0	0	1	Rm				Multiply							
Cond	0	0	0	0	1	U	A	S	RdHi				RdLo				Rn				1	0	0	1	Rm				Multiply Long								
Cond	0	0	0	1	0	B	0	0	Rn				Rd				0	0	0	0	1	0	0	1	Rm				Single Data Swap								
Cond	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	Rn				Branch and Exchange									
Cond	0	0	0	P	U	0	W	L	Rn				Rd				0	0	0	0	1	S	H	1	Rm				Halfword Data Transfer: register offset								
Cond	0	0	0	P	U	1	W	L	Rn				Rd				Offset				1	S	H	1	Offset				Halfword Data Transfer: immediate offset								
Cond	0	1	1	P	U	B	W	L	Rn				Rd				Offset																Single Data Transfer				
Cond	0	1	1																					1					Undefined								
Cond	1	0	0	P	U	S	W	L	Rn				Register List																Block Data Transfer								
Cond	1	0	1	L	Offset																													Branch			
Cond	1	1	0	P	U	N	W	L	Rn				CRd				CP#				Offset																Coprocessor Data Transfer
Cond	1	1	1	0	CP Opc				CRn				CRd				CP#				CP	0	CRm				Coprocessor Data Operation										
Cond	1	1	1	0	CP Opc				L	CRn				Rd				CP#				CP	1	CRm				Coprocessor Register Transfer									
Cond	1	1	1	1	Ignored by processor																													Software Interrupt			
3	3	2	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	1	1	1	9	8	7	6	5	4	3	2	1	0		
1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0																

3.ARM Instruction Set

The instruction mainly contains move, arithmetic, logical, comparison and multiply instructions. All operations are shown as following:

Operation Mnemonic	Meaning	Operation Mnemonic	Meaning
ADC	Add with Carry	MVN	Logical NOT
ADD	Add	ORR	Logical OR
AND	Logical AND	RSB	Reverse Subtract
BAL	Unconditional Branch	RSC	Reverse Subtract with Carry
B<cc>	Branch on Condition	SBC	Subtract with Carry
BIC	Bit Clear	SMLAL	Mult Accum Signed Long
BLAL	Unconditional Branch and Link	SMULL	Multiply Signed Long
BL<cc>	Conditional Branch and Link	STM	Store Multiple
CMP	Compare	STR	Store Register (Word)
EOR	Exclusive OR	STRB	Store Register (Byte)
LDM	Load Multiple	SUB	Subtract
LDR	Load Register (Word)	SWI	Software Interrupt
LDRB	Load Register (Byte)	SWP	Swap Word Value
MLA	Multiply Accumulate	SWPB	Swap Byte Value
MOV	Move	TEQ	Test Equivalence
MRS	Load SPSR or CPSR	TST	Test
MSR	Store to SPSR or CPSR	UMLAL	Mult Accum Unsigned Long
MUL	Multiply	UMULL	Multiply Unsigned Long

4. Certain calculations

For general assembly instructions, the syntax is

<instruction> {<cond>} {S} Rd, N

General conditions are shown as following:

Mnemonic	Condition	Mnemonic	Condition
CS	<i>Carry Set</i>	CC	<i>Carry Clear</i>
EQ	<i>Equal (Zero Set)</i>	NE	<i>Not Equal (Zero Clear)</i>
VS	<i>Overflow Set</i>	VC	<i>Overflow Clear</i>
GT	<i>Greater Than</i>	LT	<i>Less Than</i>
GE	<i>Greater Than or Equal</i>	LE	<i>Less Than or Equal</i>
PL	<i>Plus (Positive)</i>	MI	<i>Minus (Negative)</i>
HI	<i>Higher Than</i>	LO	<i>Lower Than (aka CC)</i>
HS	<i>Higher or Same (aka CS)</i>	LS	<i>Lower or Same</i>

4.1 Register Movement

<instruction> {<cond>} {S} Rd, N

- MOV: Move a 32-bit value into a register $R_d = N$
- MVN: Move the NOT of the 32-bit value into a register $R_d = \sim N$

For example:

```
MOV    R0,  R2
```

PRE R0 = 5, R2 = 8

POST R0 = 8, R2 = 8

4.2 Operand Instructions

- During the instructions, operand is important in processing. Operand to ALU is routed through the Barrel shifter, it can be modified before it is used.
- Some of the operands are shown as following:

Mnemonic	Description	Shift	Result
LSL	logical shift left	$x\text{LSL } y$	$x \ll y$
LSR	logical shift right	$x\text{LSR } y$	$(\text{unsigned})x \gg y$
ASR	arithmetic right shift	$x\text{ASR } y$	$(\text{signed})x \gg y$
ROR	rotate right	$x\text{ROR } y$	$((\text{unsigned})x \gg y) (x \ll (32 - y))$
RRX	rotate right extended	$x\text{RRX}$	$(c \text{ flag} \ll 31) ((\text{unsigned})x \gg 1)$

4.2 Operand Instructions

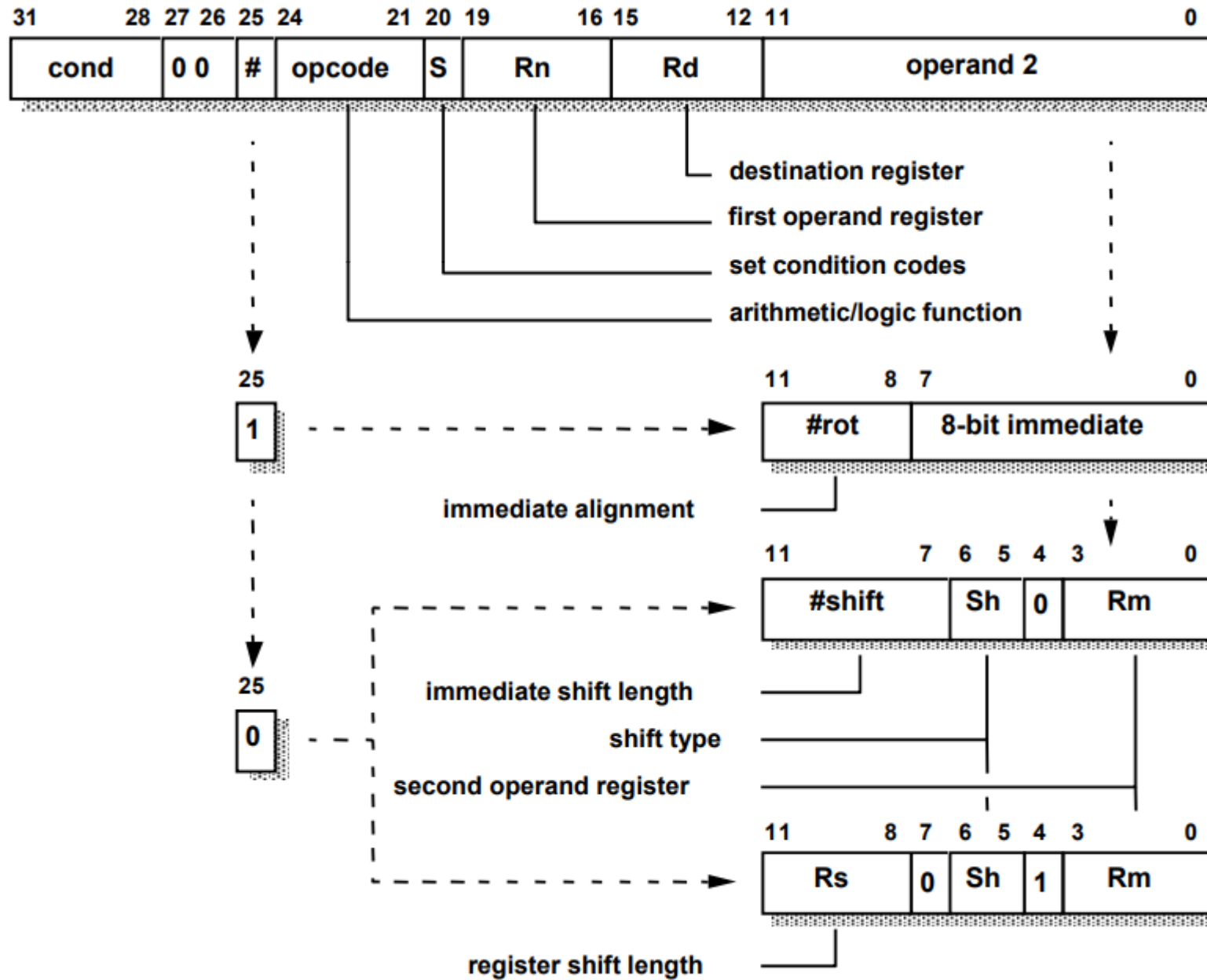
For logical shift right

MOV R0, R2, LSR #2 During this process, $R0 = R2 \gg 2$ and R2 is unchanged

For example: the value is 0...0 0011 0000

Before $R2 = 0x00000030$

After $R0 = 0x0000000C$, $R2 = 0x00000030$



4.3 Arithmetic Instructions

- Syntax: <Instruction> {<cond>} {S} Rd, Rn, N

ADC	Add two 32-bit values and carry	$Rd = Rn + N + \text{carry}$
ADD	Add two 32-bit values	$Rd = Rn + N$
RSB	Reverse subtract of two 32-bit values	$Rd = N - Rn$
RSC	Reverse subtract with carry of two 32-bit values	$Rd = N - Rn - \text{!(carry flag)}$
SBC	Subtract with carry of two 32-bit values	$Rd = Rn - N - \text{!(carry flag)}$
SUB	Subtract two 32-bit values	$Rd = Rn - N$

4.4 Logical Instructions

- Syntax: $\langle \text{instruction} \rangle \{ \langle \text{cond} \rangle \} \{ S \} R_d, R_n, N$

AND	Logical bitwise AND of two 32-bit values	$R_d = R_n \& N$
ORR	Logical bitwise OR of two 32-bit values	$R_d = R_n N$
EOR	Logical exclusive OR of two 32-bit values	$R_d = R_n \wedge N$
BIC	Logical bit clear (AND NOT)	$R_d = R_n \& \sim N$

4.5 Comparison Instructions

- These instructions do not generate a result but set condition code bits (N Z C V) in CPSR. Often, a branch operation follows to change the program flow.

Syntax: <instruction> {<cond>} Rn, N

CMN	Compare negated	Flags set as a results of $Rn + N$
CMP	compare	Flags set as a results of $Rn - N$
TEQ	Test for equality of two 32-bit values	Flags set as a results of $Rn \wedge N$
TST	Test bits of a 32-bit value	Flags set as a results of $Rn \& N$

4.6 Multiplication Instructions

Syntax: MLA {<cond>} {S} Rd, Rm, Rs, Rn

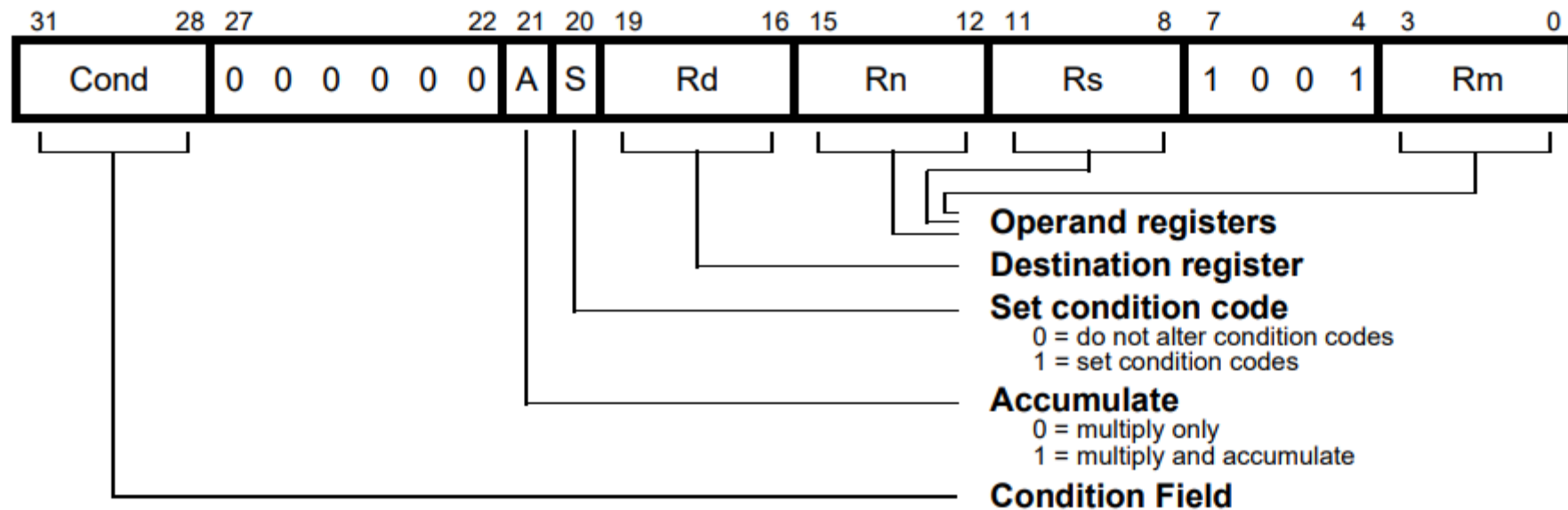
MUL {<cond>} {S} Rd, Rm, Rs

MLA	Multiply and accumulate	$Rd = (Rm * Rs) + Rn$
MUL	Multiply	$Rd = Rm * Rs$

Syntax: <instruction> {<cond>} {S} Rdlo, RdHi, Rm, Rs

SMLAL	Signed multiply and accumulate long	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm * Rs)$
SMULL	Signed multiply long	$[RdHi, RdLo] = (Rm * Rs)$
UMLAL	Unsigned multiply accumulate long	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm * Rs)$
UMULL	Unsigned multiply long	$[RdHi, RdLo] = (Rm * Rs)$

4.6 Multiplication Instructions



4.7 A Realistic Example

Consider a simple fragment of C code:

```
for (i = 10; i != 0; i--) {  
    do_something();  
}
```

A compiler may transfer the code to assembly as follows:

```
mov    r4, #10  
loop_label:  
bl     do_something  
sub    r4, r4, #1  
cmp    r4, #0  
bne    loop_label
```

Can you revise the code in a more efficient way

4.7 A Realistic Example

Revise the code in a more efficient way:

```
mov    r4, #10
loop_label:
bl     do_something
sub    r4, r4, #1
cmp    r4, #0
bne    loop_label
```

```
mov    r4, #10
loop_label:
bl     do_something
subs   r4, r4, #1
bne    loop_label
```

The **cmp** (compare) instruction compares r4 with 0, and the **bne** instruction is simply a **b** (branch) instruction that executes if the result of the **cmp** instruction was "not equal". The code works because **cmp** sets some global flags indicating various properties of the operation. The **bne** instruction — which is really just a **b** (branch) with a **ne** condition code suffix — reads these flags to determine whether or not to branch. Adding the **s** suffix to **sub** causes it to update the flags itself, based on the result of the operation.

5. Conclusion

- Besides these instructions, there are many kind of instructions when you get touch to a new device with different architecture. For most important issues, please search the instructions from official website.
- For RISC device, assembly code can help comprehend the process on computer
- Follow the syntax when you process your application or calculations.
- To realize further applications, embedded instruction sets such as Thumb instructions suffer from extremely high register pressure because they have small register sets

6.ARM vs RISC-V

	ARM	RISC-V
Origin	Acorn Computers (1980s)	University of California, Berkeley (2010)
Architecture	load-store architecture	load-store architecture
Instruction set	ARM (32-bit), Thumb (16-bit) and AArch64 (64-bit)	RV32I (32-bit), RV64I (64-bit), and RV128I (128-bit)
Processor cores	Cortex-A, Cortex-R, Cortex-M and Neoverse series	Rocket core, SiFive products including performance, automotive, intelligence and essential families
Licenses	Closed ISA, a proprietary architecture with licensing fees	Royalty-free open-source licenses
Key features	Dominant choice for embedded systems and mobile devices with a mature ecosystem	Open-source architecture with modularity and extensibility

7. Reference

- Keil official
https://www.keil.com/support/man/docs/armasm/armasm_dom1361289850039.htm
- ARM Instruction set <https://iitd-plos.github.io/col718/ref/arm-instructionset.pdf>
- LR and SP <https://developer.arm.com/documentation/dui0801/d/Writing-A32-T32-Assembly-Language/Register-usage-in-subroutine-calls>
- https://www.csie.ntu.edu.tw/~cyy/courses/assembly/10fall/lectures/handouts/lec09_ARMisa.pdf
- ARM official data processing instructions