Samstag, den 21.01.2017	**Prof. Dr.-Ing. Tim Tiedemann**

Klausur RMP

Name	**Matrikel-Nummer**

Hinweise:

1.) Tragen Sie in obige Felder Ihren Namen und Ihre Matrikelnummer ein.

2.) Zusätzliche Lösungsblätter versehen Sie bitte (vor der Verwendung) mit **Namen und Matrikelnummer**.
Nehmen Sie zur Bearbeitung einer Aufgabe jeweils ein neues Blatt.

3.) Vermerken Sie in den vorgesehenen Lösungsfeldern der Aufgabenblätter, dass ein Zusatzblatt existiert.

4.) Zur Bearbeitung stehen **90 Minuten** zur Verfügung.

5.) **Erlaubte Hilfsmittel**:
Tabellen/Cheat-Sheets/Auszüge im Anhang.
Sonst keine weiteren Hilfsmittel (Taschenrechner, Notebooks, Handys).

Übersicht zur Bewertung der Aufgaben.

Aufgabe	Punkte	
01	**30**	
02	**18**	
03	**12**	
Summe ≅	**60**	

Aufgabe 1 (Grundlagenwissen) [30 Punkte]

Wie lautet der Dezimalwert der 8-Bit-Binärzahl `1011 0111`$_B$

a) als vorzeichenlose Zahl?
b) als vorzeichenbehaftete Zahl?

Geben Sie die Dezimalzahl 300$_D$ als Binärzahl an.

Verwenden Sie die Modulo-Division.

Wie lautet die Binärdarstellung (8-Bit) der vorzeichenbehafteten Zahl –92$_D$?

Berechnen Sie (8-bit-Zahl) :

Wird das Carry-Flag gesetzt?
Wird das Overflow-Flag gesetzt?
Ist das Ergebnis richtig oder falsch:
a) bei Vz.-loser b) Vz.-behafteter Interpretation?

```
 01010011     C=
+10101111     V=
----------    a)
              b)
```

Berechnen Sie durch Addition des Zweierkomplements (8-bit-Zahl):

Wird das Carry-Flag gesetzt?
Wird das Overflow-Flag gesetzt?

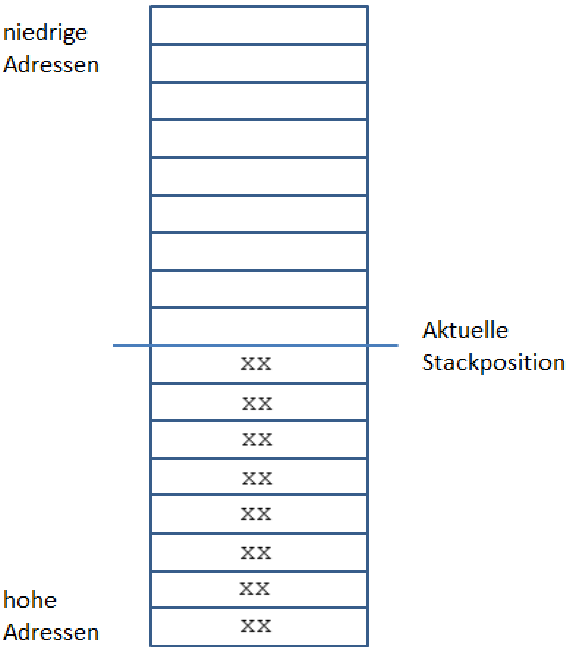
Ist das Ergebnis richtig oder falsch:
a) bei Vz.-loser b) Vz.-behafteter Interpretation?

```
 11011010     C=
-10100011     V=
              a)
              b)
```

Geben Sie die Zahl 24.9 als binäre Fixkommazahl (nicht Floatingpoint) mit 8 Vorkomma- und 8 Nachkommastellen an.	_ _ _ _ _ _ _ _ , _ _ _ _ _ _ _ _
Geben Sie den Wertebereich (den kleinsten und den größten Wert (Hex.)) in r0 an, damit ein Sprung nach LAB erfolgt! cmp r0, #0x8 bhi LAB ; if higher	Wertebereich (einschließlich): von **0x** _ _ _ _ _ _ _ _ bis **0x** _ _ _ _ _ _ _ _
Geben Sie einen Assembler-Befehl an, um die Bits 0, 1 und 2 des Registers r0 zu invertieren, ohne die anderen Bits zu verändern!	
Geben Sie eine Assembler-Befehlssequenz an, mit der zum Label LAB gesprungen wird, wenn die Bits 0-3 in den Registern r3 und r4 nicht gleich sind! Anm.: max. 4 Befehle	

Was steht auf dem Stack, wenn das Programm beim Label LOOK angekommen ist?   Wo stehen **sp** und **fp** (einzeichnen)?

```
----------------------------------------------------
main   …
        …
        mov     r3, #0x23
        mov     r4, #0x42
        push    {r3, r4}
        bl      myProg
        …
        …
myProg
        push    {lr, fp}
        mov     fp,  sp
        sub     sp,  #8
        push    {r1-r2}
LOOK    …
        …
        bx      lr
----------------------------------------------------
```



Anm.: Schreiben Sie für unbekannte Registerinhalte [sp], [fp], [lr], [r1] usw. und für unbekannte Werte `xx`.

Hinw.: fp = r11, lr = r14, push beginnt mit höchsten Registernummern

Anm.: Ein Feld sind 32 bit.

Was ist an der folgenden Anweisung fehlerhaft?

```
push {r3, r2, r1}
```

Wozu dient in der Regel in Unterprogrammen die Kombination von push-Anweisungen zu Beginn und pop-Anweisungen am Ende?

Aufgabe 2 (Assemblerbefehle, Adressierungsarten) [18 Punkte]

Nachfolgend ist ein Programm angegeben. Der Datenblock beginne bei Adresse `0x50000030`.

a) Geben Sie die Speicherbelegung des Datenblocks in hexadezimaler Darstellung an.

Adresse	**+0**	**+1**	**+2**	**+3**
0x50000030				
0x50000034				
0x50000038				
0x5000003C				

Anm. : unbekannte Inhalte mit ‘xx’ markieren

Welche Werte haben die Label (d.h. was steht in der Symboltabelle)?

w0 = b0 =
w1 = E0 =

b) Geben Sie die Werte der geänderten Register bzw. Speicherzellen sowie den Zustand des Statusregisters (Flags) an. Verwenden Sie die neben dem Programm stehende Tabelle.

```
            AREA MyData, DATA, align = 2 ; 4-Byte-Alignment

w0          DCD   0x12AB34CD
b0          DCB   0x10, 10, 2_10, "10", 0
            ALIGN 4
w1          DCD   0xABCD12
E0          EQU   w0 + 4
```

alle Werte im Hexadezimalformat

						Byte			
	N	**Z**	**V**	**C**		**3**	**2**	**1**	**0**
	0	0	0	0	**r0=r1=r2 =r3=r4=**	**AA**	**77**	**AA**	**77**
`ldr   r0, =w0`					**r0 =**				
`ldr   r1, [r0]`					**r1 =**				
`ldrb  r2, [r0, #4]`					**r2 =**				
`mov   r3, #0x8787`					**r3 =**				
`eors  r4, r3, #0x78`					**r4 =**				
`ands  r5, r3, #0x78`					**r5 =**				
`subs  r3, r3, LSR #8`					**r3 =**				
`ldrh  r7, [r0, #2]!`					**r7 =**				
`ldrh  r8, [r0], #2`					**r8 =**				
`ldrh  r9, [r0]`					**r9 =**				

Hinweis: 'A' = 0x41, '0'=0x30

Aufgabe 3 (Fragen zur Programmiersprache C) [12 Punkte]

Welchen Wert hat `i` nach folgender Sequenz?
`int  i;`
`i = 2+3*4;`	

Welchen Wert hat f nach folgender Programmsequenz:
 	
`double f;`
`f = 3/4 + 2.25;`	

Welchen Wert hat `x` nach folgender Sequenz?
`int             a=64;`
`signed char     x;`
`x = 2*a;`	

Welchen Wert hat a nach folgender Programmsequenz?
`#define MLA(A,B,C)  A * B + C`
`int      a, x=3, y=4, z=5;`
`a = MLA(x+1, y-1, z);`
 	
Was wollte der Programmierer vermutlich eigentlich erreichen? Helfen Sie ihm und verbessern Sie das Programm.	

Schreiben Sie ein äquivalentes Programm, welches statt der while-Schleife eine for-Schleife nutzt.
 	
`int   x=20;`
`while(x  < 30 ){`
`       printf("x=%d\n", ++x);`
`}`	

Welchen Wert (dezimal) hat c nach der Addition?
`char a=’1’, b=’2’, c;`
`c = a+b;`	

Initialisieren Sie den Zeiger `pA` so, dass er auf das `‘A‘` zeigt.
`char  str[] = “HAW Hamburg“;`
`char *pA;`
	`pA =`

Geben Sie die Definition (also den Programmcode) einer Funktion "`addFirst`" an, der ein int-Vektor übergeben wird und welche die Summe der ersten beiden Vektorelemente zurückgibt (als int).	
Gegeben sei der Vektor:: **`int`** `Vek[] = {1,2,3,4,5,6};`	
Wie wird die Funktion "`addFirst`" (s. Aufgabe darüber) mit dem Vektor `Vek` aufgerufen?	`Erg = addFirst (                    );`
Wie lautet der Funktionsaufruf, wenn das 3. und 4. Element des Vektors addiert werden soll?	`Erg = addFirst (                    );`

Opcode	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LSL Rd, Rm, #	0	0	0	0	0	#					Rm			Rd		
LSR Rd, Rm, #	0	0	0	0	1	#					Rm			Rd		
ASR Rd, Rm, #	0	0	0	1	0	#					Rm			Rd		
ADD Rd, Rn, Rm	0	0	0	1	1	0	0	Rm			Rn			Rd		
SUB Rd, Rn, Rm	0	0	0	1	1	0	1	Rm			Rn			Rd		
ADD Rd, Rn, #	0	0	0	1	1	1	0	#			Rn			Rd		
SUB Rd, Rn, #	0	0	0	1	1	1	1	#			Rn			Rd		
MOV Rd, #	0	0	1	0	0	Rd			#							
CMP Rn, #	0	0	1	0	1	Rn			#							
ADD Rd, #	0	0	1	1	0	Rd			#							
SUB Rd, #	0	0	1	1	1	Rd			#							
AND Rd, Rm	0	1	0	0	0	0	0	0	0	0	Rm			Rd		
EOR Rd, Rm	0	1	0	0	0	0	0	0	0	1	Rm			Rd		
LSL Rd, Rs	0	1	0	0	0	0	0	0	1	0	Rs			Rd		
LSR Rd, Rs	0	1	0	0	0	0	0	0	1	1	Rs			Rd		
ASR Rd, Rs	0	1	0	0	0	0	0	1	0	0	Rs			Rd		
ADC Rd, Rm	0	1	0	0	0	0	0	1	0	1	Rm			Rd		
SBC Rd, Rm	0	1	0	0	0	0	0	1	1	0	Rm			Rd		
ROR Rd, Rs	0	1	0	0	0	0	0	1	1	1	Rs			Rd		
TST Rm, Rn	0	1	0	0	0	0	1	0	0	0	Rn			Rm		
NEG Rd, Rm	0	1	0	0	0	0	1	0	0	1	Rm			Rd		
CMP Rm, Rn	0	1	0	0	0	0	1	0	1	0	Rn			Rm		
CMN Rm, Rn	0	1	0	0	0	0	1	0	1	1	Rn			Rm		
ORR Rd, Rm	0	1	0	0	0	0	1	1	0	0	Rm			Rd		
MUL Rd, Rm	0	1	0	0	0	0	1	1	0	1	Rm			Rd		
BIC Rm, Rd	0	1	0	0	0	0	1	1	1	0	Rn			Rm		
MVN Rd, Rm	0	1	0	0	0	0	1	1	1	1	Rm			Rd		
Unpredictable	0	1	0	0	0	1	0	0	0	0	x	x	x	x	x	x
ADD Rd, Rm	0	1	0	0	0	1	0	0	H1	H2	Rm			Rd		
Unpredictable	0	1	0	0	0	1	0	1	0	0	x	x	x	x	x	x
CMP Rm, Rn	0	1	0	0	0	1	0	1	H1	H2	Rn			Rm		
Unpredictable	0	1	0	0	0	1	1	0	0	0	x	x	x	x	x	x
MOV Rd, Rm	0	1	0	0	0	1	1	0	H1	H2	Rm			Rd		
BX Rm	0	1	0	0	0	1	1	1	0	H2	Rm			0	0	0
BLX Rm	0	1	0	0	0	1	1	1	1	H2	Rm			0	0	0
Unpredictable	0	1	0	0	0	1	1	1	x	x	x	x	x	1	x	x
LDR Rd, [PC, #]	0	1	0	0	1	Rd			PC Relative Offset							
STR Rd, [Rn, Rm]	0	1	0	1	0	0	0	Rm			Rn			Rd		
STRH Rd, [Rn, Rm]	0	1	0	1	0	0	1	Rm			Rn			Rd		
STRB Rd, [Rn, Rm]	0	1	0	1	0	1	0	Rm			Rn			Rd		
LDRSB Rd, [Rn, Rm]	0	1	0	1	0	1	1	Rm			Rn			Rd		
LDR Rd, [Rn, Rm]	0	1	0	1	1	0	0	Rm			Rn			Rd		
LDRH Rd, [Rn, Rm]	0	1	0	1	1	0	1	Rm			Rn			Rd		
LDRB Rd, [Rn, Rm]	0	1	0	1	1	1	0	Rm			Rn			Rd		
LDRSH Rd, [Rn, Rm]	0	1	0	1	1	1	1	Rm			Rn			Rd		
STR Rd, [Rn, #OFF]	0	1	1	0	0	# Offset					Rn			Rd		
LDR Rd, [Rn, #OFF]	0	1	1	0	1	# Offset					Rn			Rd		
STRB Rd, [Rn, #OFF]	0	1	1	1	0	# Offset					Rn			Rd		
LDRB Rd, [Rn, #OFF]	0	1	1	1	1	# Offset					Rn			Rd		
STRH Rd, [Rn, #OFF]	1	0	0	0	0	# Offset					Rn			Rd		
LDRH Rd, [Rn, #OFF]	1	0	0	0	1	# Offset					Rn			Rd		
STR Rd, [SP, #OFF]	1	0	0	1	0	Rd			SP Relative Offset							
LDR Rd, [SP, #OFF]	1	0	0	1	1	Rd			SP Relative Offset							
ADD Rd, PC, #OFF	1	0	1	0	0	Rd			PC Relative Offset							
ADD Rd, SP, #OFF	1	0	1	0	1	Rd			SP Relative Offset							
SUB SP, SP, #OFF	1	0	1	1	0	0	0	0	1	SP Relative Offset						
Unpredictable	1	0	1	1	0	0	0	1	x	x	x	x	x	x	x	x
PUSH {<reg list>, <LR>}	1	0	1	1	0	1	0	LR	Register List							
POP {<reg list>, <PC>}	1	0	1	1	1	1	0	PC	Register List							
Unpredictable	1	0	1	1	0	x	1	x	x	x	x	x	x	x	x	x
Unpredictable	1	0	1	1	1	0	x	x	x	x	x	x	x	x	x	x
BKPT #	1	0	1	1	1	1	1	0	#							
Unpredictable	1	0	1	1	1	1	1	1	x	x	x	x	x	x	x	x
STMIA Rn!, {<reg list>}	1	1	0	0	0	Rn			Register List							
LDMIA Rn!, {<reg list>}	1	1	0	0	1	Rn			Register List							
B{<cond>} <Target Addr>	1	1	0	1	cond				# Offset							
Unused Opcode	1	1	0	1	1	1	1	0	x	x	x	x	x	x	x	x
SWI #	1	1	0	1	1	1	1	1	#							
B <Target Addr>	1	1	1	0	0	# Offset										
BLX <Target Addr>	1	1	1	0	1	# Offset (lower half)										
BL{X} <Target Addr> (+)	1	1	1	1	0	# Offset (upper half)										
BL <Target Addr>	1	1	1	1	1	# Offset (lower half)										

rE Ejected

re-eject.gbadev.org

ARM Thumb Reference V2

rev	Mnemonic	Definition	Alternate Description
	ADC	Add with Carry	ADD numbers and Carry bit
	ADD	Add	ADD numbers
	AND	Logical AND	AND together numbers
	ASR	Arithmetic Shift Right	Signed Right Shift (>>)
	B	Branch	Jump to an address
	BIC	Bit Clear	AND's the compliment of a number
v5	BKPT	Breakpoint	Software Breakpoint
	BL	Branch with Link	Jump to an address, and set LR to return address
v5	BLX	Branch with Link and Exchange	Jump to an address, set LR to return address, switch operating modes
	BX	Branch and Exchange	Jump to an address, and switch operating modes
	CMN	Compare Negative	Compare numbers by addition
	CMP	Compare	Compare numbers by subtraction
	EOR	Logical Exclusive OR (XOR)	Exclusive OR together numbers
	LDMIA	Load Multiple, Increment After	Load Multiple registers at once
	LDR	Load Register (word)	Load an unsigned 32bit number into a register
	LDRB	Load Register (byte)	Load an unsigned 8bit number into a register
	LDRH	Load Register (halfword)	Load an unsigned 16bit number into a register
	LDRSB	Load Register (byte)	Load a signed 8bit number into a register
	LDRSH	Load Register (halfword)	Load a signed 16bit number into a register
	LSL	Logical Shift Left	Unsigned Left Shift (<<)
	LSR	Logical Shift Right	Unsigned Right Shift (>>)
	MOV	Move	Move a number
	MUL	Multiply	Multiply numbers
	MVN	Move Not	Compliment a number
	NEG	Negate	Negate a number
	ORR	Logical OR	OR together numbers
	POP	Pop multiple registers	Takes numbers off the stack
	PUSH	Push multiple registers	Puts numbers on to the stack
	ROR	Rotate Right Register	Shifts right (>>), and numbers shifted off are appended to top
	SBC	Subtract with Carry	Subtract numbers and ADD Carry bit
	STMIA	Store Multiple, Increment After	Store Multiple registers at once
	STR	Store Register (word)	Store a 32bit number into an address
	STRB	Store Register (byte)	Store an 8bit number into an address
	STRH	Store Register (halfword)	Store a 16bit number into an address
	SUB	Subtract	Subtract numbers
	SWI	Software Interrupt	Execute code/"bios" calls
	TST	Test	Checks if one of more bits are set
	Unused	Unused Opcode	Future revisions of the Architecture will not use this space

Opcode	Work	Notes	Z	C	N	V
ADC Rd, Rm	Rd = Rd + Rm + C	-	x	x	x	x
ADD Rd, #	Rd = Rd + #	-	x	x	x	x
ADD Rd, PC, #OFF	Rd = Rd + (PC + (#OFF << 2))	-				
ADD Rd, Rm	Rd = Rd + Rm	Rd or Rm must be a *high register*				
ADD Rd, Rn, #	Rd = Rn + #	-	x	x	x	x
ADD Rd, Rn, Rm	Rd = Rm + Rn	-	x	x	x	x
ADD Rd, SP, #OFF	Rd = SP + (#OFF << 2)	-				
AND Rd, Rm	Rd = Rd & Rm	-	x	x	x	x
ASR Rd, Rm, #	Rd = Rm >> #	signed	x	x	x	
ASR Rd, Rs	Rd = Rm >> Rs	signed	x	x	x	
B <Target Addr>	PC = PC + (#OFF << 1)	-				
B{<cond>} <Target Addr>	PC = PC + (#OFF << 1)	If <cond> is true				
BIC Rm, Rd	Rd = Rd & !(Rm)	-	x	x	x	
BKPT #	CALL Breakpoint with #	v5 only. v4 it does nothing				
BL <Target Addr>	See Branching Description	-				
BLX <Target Addr>	See Branching Description	-				
BLX Rm	LR = (PC + 2) | 1; PC = Rm[31..1] << 1; T=Rm[0]	-				
BX Rm	PC = Rm[31..1] << 1; T = Rm[0]	-				
CMN Rm, Rn	<flags> = Rm + Rn	-	x	x	x	x
CMP Rm, Rn	<flags> = Rm - Rn	-	x	x	x	x
CMP Rm, Rn	<flags> = Rm - Rn	Rm or Rn must be a *high register*	x	x	x	x
CMP Rn, #	<flags> = Rm - #	-	x	x	x	x
EOR Rd, Rm	Rd = Rd ^ Rm	-	x	x	x	x
LDMIA Rn!, {<reg list>}	for each in <reg list> = [Rn+=4]	-				
LDR Rd, [PC, #OFF]	Rd = [PC + (#OFF << 2)]	Word				
LDR Rd, [Rn, #OFF]	Rd = [Rn + (#OFF << 2)]	Word				
LDR Rd, [Rn, Rm]	Rd = [Rn + Rm]	Word				
LDR Rd, [SP, #OFF]	Rd = [SP + (#OFF << 2)]	Word				
LDRB Rd, [Rn, #OFF]	Rd = [Rn + (#OFF << 2)]	Unsigned Byte				
LDRB Rd, [Rn, Rm]	Rd = [Rn + Rm]	Unsigned Byte				
LDRH Rd, [Rn, #OFF]	Rd = [Rn + (#OFF << 2)]	Unsigned Halfword				
LDRH Rd, [Rn, Rm]	Rd = [Rn + Rm]	Unsigned Halfword				
LDRSB Rd, [Rn, Rm]	Rd = [Rn + Rm]	Signed Byte				
LDRSH Rd, [Rn, Rm]	Rd = [Rn + Rm]	Signed Halfword				
LSL Rd, Rm, #	Rd = Rm << #	Unsigned/Signed	x	x	x	
LSL Rd, Rs	Rd = Rm << Rs	Unsigned/Signed	x	x	x	
LSR Rd, Rm, #	Rd = Rm >> #	Unsigned	x	x	x	
LSR Rd, Rs	Rd = Rm >> Rs	Unsigned	x	x	x	
MOV Rd, #	Rd = #	-	x		x	
MOV Rd, Rm	Rd = Rm	Rd or Rm must be a *high register*				
MUL Rd, Rm	Rd = Rd * Rm	-	x	x	x	
MVN Rd, Rm	Rd = !(Rm)	-	x		x	
NEG Rd, Rm	Rd = -(Rm)	-	x	x	x	x
ORR Rd, Rm	Rd = Rd | Rm	-	x	x	x	
POP {<reg list>, <PC>}	get <reg list> and/or <PC> from stack	-				
PUSH {<reg list>, <LR>}	put <reg list> and/or <LR> on stack	-				
ROR Rd, Rs	Rd = Rd >|> Rs	-	x	x	x	
SBC Rd, Rm	Rd = (Rd - Rm) + C	-	x	x	x	x
STMIA Rn!, {<reg list>}	[Rn+=4] = for each in <reg list>	-				
STR Rd, [Rn, #OFF]	[Rn + (#OFF << 2)] = Rd	word				
STR Rd, [Rn, Rm]	[Rn + Rm] = Rd	word				
STR Rd, [SP, #OFF]	[SP + (#OFF << 2)] = Rd	word				
STRB Rd, [Rn, #OFF]	[Rn + (#OFF << 2)] = Rd	byte				
STRB Rd, [Rn, Rm]	[Rn + Rm] = Rd	byte				
STRH Rd, [Rn, #OFF]	[Rn + (#OFF << 2)] = Rd	halfword				
STRH Rd, [Rn, Rm]	[Rn + Rm] = Rd	halfword				
SUB Rd, #	Rd = Rd - #	-	x	x	x	x
SUB Rd, Rn, #	Rd = Rn - #	-	x	x	x	x
SUB Rd, Rn, Rm	Rd = Rn - Rm	-	x	x	x	x
SUB SP, #OFF	SP = SP - (#OFF << 2)	-				
SWI #	Run "bios" function	-				
TST Rm, Rn	<flags> = Rn & Rm	-	x		x	
Unused Opcode	none	Free for software use. Minimal risk of future CPU revisions turning this into an opcode.				

Meaning	Mnemonic	Opcode				Status Flags
Equal	EQ	0	0	0	0	z = 1
Not Equal	NE	0	0	0	1	z = 0
Carry Set	CS	0	0	1	0	c = 1
Carry Clear	CC	0	0	1	1	c = 0
Unsigned Higher or Same	HS	0	0	1	0	c = 1
Unsigned Lower	LO	0	0	1	1	c = 0
Minus/Negative	MI	0	1	0	0	n = 1
Plus/Positive or Zero	PL	0	1	0	1	n = 0
Overflow	VS	0	1	1	0	v = 1
No Overflow	VC	0	1	1	1	v = 0
Unsigned Higher	HI	1	0	0	0	c = 1; z = 0
Unsigned Lower or Same	LS	1	0	0	1	c = 0; z = 1
Signed Greater Than or Equal	GE	1	0	1	0	n = v
Signed Less Than	LT	1	0	1	1	n != v
Signed Greater Than	GT	1	1	0	0	z = 0; n = v
Signed Less Than or Equal	LE	1	1	0	1	z = 1; n != v
Always	AL	1	1	1	0	-
Never	NE	1	1	1	1	-

Opcode	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADC Rd, Rm	0	1	0	0	0	0	0	1	0	1	Rm			Rd		
ADD Rd, #	0	0	1	1	0	Rd			#							
ADD Rd, PC, #OFF	1	0	1	0	0	Rd			PC Relative Offset							
ADD Rd, Rm	0	1	0	0	0	1	0	0	H1	H2	Rm			Rd		
ADD Rd, Rn, #	0	0	0	1	1	1	0	#			Rn			Rd		
ADD Rd, Rn, Rm	0	0	0	1	1	0	0	Rm			Rn			Rd		
ADD Rd, SP, #OFF	1	0	1	0	1	Rd			SP Relative Offset							
AND Rd, Rm	0	1	0	0	0	0	0	0	0	0	Rm			Rd		
ASR Rd, Rm, #	0	0	0	1	0	#					Rm			Rd		
ASR Rd, Rs	0	1	0	0	0	0	0	1	0	0	Rs			Rd		
B <Target Addr>	1	1	1	0	0	# Offset										
B{<cond>} <Target Addr>	1	1	0	1	cond				# Offset							
BIC Rm, Rd	0	1	0	0	0	0	1	1	1	0	Rn			Rm		
BKPT #	1	0	1	1	1	1	1	0	#							
BL <Target Addr>	1	1	1	1	1	# Offset (lower half)										
BL{X} <Target Addr> (+)	1	1	1	1	0	# Offset (upper half)										
BLX <Target Addr>	1	1	1	0	1	# Offset (lower half)										
BLX Rm	0	1	0	0	0	1	1	1	1	H2	Rm			0	0	0
BX Rm	0	1	0	0	0	1	1	1	0	H2	Rm			0	0	0
CMN Rm, Rn	0	1	0	0	0	0	1	0	1	1	Rn			Rm		
CMP Rm, Rn	0	1	0	0	0	0	1	0	1	0	Rn			Rm		
CMP Rm, Rn	0	1	0	0	0	1	0	1	H1	H2	Rn			Rm		
CMP Rn, #	0	0	1	0	1	Rn			#							
EOR Rd, Rm	0	1	0	0	0	0	0	0	0	1	Rm			Rd		
LDMIA Rn!, {<reg list>}	1	1	0	0	1	Rn			Register List							
LDR Rd, [PC, #]	0	1	0	0	1	Rd			PC Relative Offset							
LDR Rd, [Rn, #OFF]	0	1	1	0	1	# Offset					Rn			Rd		
LDR Rd, [Rn, Rm]	0	1	0	1	1	0	0	Rm			Rn			Rd		
LDR Rd, [SP, #OFF]	1	0	0	1	1	Rd			SP Relative Offset							
LDRB Rd, [Rn, #OFF]	0	1	1	1	1	# Offset					Rn			Rd		
LDRB Rd, [Rn, Rm]	0	1	0	1	1	1	0	Rm			Rn			Rd		
LDRH Rd, [Rn, #OFF]	1	0	0	0	1	# Offset					Rn			Rd		
LDRH Rd, [Rn, Rm]	0	1	0	1	1	0	1	Rm			Rn			Rd		
LDRSB Rd, [Rn, Rm]	0	1	0	1	0	1	1	Rm			Rn			Rd		
LDRSH Rd, [Rn, Rm]	0	1	0	1	1	1	1	Rm			Rn			Rd		
LSL Rd, Rm, #	0	0	0	0	0	#					Rm			Rd		
LSL Rd, Rs	0	1	0	0	0	0	0	0	1	0	Rs			Rd		
LSR Rd, Rm, #	0	0	0	0	1	#					Rm			Rd		
LSR Rd, Rs	0	1	0	0	0	0	0	0	1	1	Rs			Rd		
MOV Rd, #	0	0	1	0	0	Rd			#							
MOV Rd, Rm	0	1	0	0	0	1	1	0	H1	H2	Rm			Rd		
MUL Rd, Rm	0	1	0	0	0	0	1	1	0	1	Rm			Rd		
MVN Rd, Rm	0	1	0	0	0	0	1	1	1	1	Rm			Rd		
NEG Rd, Rm	0	1	0	0	0	0	1	0	0	1	Rm			Rd		
ORR Rd, Rm	0	1	0	0	0	0	1	1	0	0	Rm			Rd		
POP {<reg list>, <PC>}	1	0	1	1	1	1	0	PC	Register List							
PUSH {<reg list>, <LR>}	1	0	1	1	0	1	0	LR	Register List							
ROR Rd, Rs	0	1	0	0	0	0	0	1	1	1	Rs			Rd		
SBC Rd, Rm	0	1	0	0	0	0	0	1	1	0	Rm			Rd		
STMIA Rn!, {<reg list>}	1	1	0	0	0	Rn			Register List							
STR Rd, [Rn, #OFF]	0	1	1	0	0	# Offset					Rn			Rd		
STR Rd, [Rn, Rm]	0	1	0	1	0	0	0	Rm			Rn			Rd		
STR Rd, [SP, #OFF]	1	0	0	1	0	Rd			SP Relative Offset							
STRB Rd, [Rn, #OFF]	0	1	1	1	0	# Offset					Rn			Rd		
STRB Rd, [Rn, Rm]	0	1	0	1	0	1	0	Rm			Rn			Rd		
STRH Rd, [Rn, #OFF]	1	0	0	0	0	# Offset					Rn			Rd		
STRH Rd, [Rn, Rm]	0	1	0	1	0	0	1	Rm			Rn			Rd		
SUB Rd, #	0	0	1	1	1	Rd			#							
SUB Rd, Rn, #	0	0	0	1	1	1	1	#			Rn			Rd		
SUB Rd, Rn, Rm	0	0	0	1	1	0	1	Rm			Rn			Rd		
SUB SP, SP, #OFF	1	0	1	1	0	0	0	0	1	SP Relative Offset						
SWI #	1	1	0	1	1	1	1	1	#							
TST Rm, Rn	0	1	0	0	0	0	1	0	0	0	Rn			Rm		
Unpredictable	0	1	0	0	0	1	0	0	0	0	x	x	x	x	x	x
Unpredictable	0	1	0	0	0	1	0	1	0	0	x	x	x	x	x	x
Unpredictable	0	1	0	0	0	1	1	0	0	0	x	x	x	x	x	x
Unpredictable	0	1	0	0	0	1	1	1	x	x	x	x	x	1	x	x
Unpredictable	1	0	1	1	0	0	0	1	x	x	x	x	x	x	x	x
Unpredictable	1	0	1	1	0	x	1	x	x	x	x	x	x	x	x	x
Unpredictable	1	0	1	1	1	0	x	x	x	x	x	x	x	x	x	x
Unpredictable	1	0	1	1	1	1	1	1	x	x	x	x	x	x	x	x
Unused Opcode	1	1	0	1	1	1	1	0	x	x	x	x	x	x	x	x

The ASCII Table

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
00	00	NUL	32	20	SP	64	40	@	96	60	‘
01	01	SOH	33	21	!	65	41	A	97	61	a
02	02	STX	34	22	"	66	42	B	98	62	b
03	03	ETX	35	23	#	67	43	C	99	63	c
04	04	EOT	36	24	$	68	44	D	100	64	d
05	05	ENQ	37	25	%	69	45	E	101	65	e
06	06	ACK	38	26	&	70	46	F	102	66	f
07	07	BEL	39	27	’	71	47	G	103	67	g
08	08	BS	40	28	(	72	48	H	104	68	h
09	09	HT	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	ˆ	126	7E	˜
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

3.5.1 ADD, ADC, SUB, SBC, and RSB

Add, Add with carry, Subtract, Subtract with carry, and Reverse Subtract.

Syntax

```
op{S}{cond} {Rd,} Rn, Operand2

op{cond} {Rd,} Rn, #imm12              ; ADD and SUB only
```

where:

op Is one of:

ADD	Add.
ADC	Add with Carry.
SUB	Subtract.
SBC	Subtract with Carry.
RSB	Reverse Subtract.

S Is an optional suffix. If S is specified, the condition code flags are updated on the result of the operation, see *Conditional execution* on page 3-18.

cond Is an optional condition code, see *Conditional execution* on page 3-18.

Rd Specifies the destination register. If *Rd* is omitted, the destination register is *Rn*.

Rn Specifies the register holding the first operand.

Operand2 Is a flexible second operand. See *Flexible second operand* on page 3-12 for details of the options.

imm12 Is any value in the range 0-4095.

Operation

The ADD instruction adds the value of *Operand2* or *imm12* to the value in *Rn*.

The ADC instruction adds the values in *Rn* and *Operand2*, together with the carry flag.

The SUB instruction subtracts the value of *Operand2* or *imm12* from the value in *Rn*.

The SBC instruction subtracts the value of *Operand2* from the value in *Rn*. If the carry flag is clear, the result is reduced by one.

The RSB instruction subtracts the value in *Rn* from the value of *Operand2*. This is useful because of the wide range of options for *Operand2*.

Use ADC and SBC to synthesize multiword arithmetic, see *Multiword arithmetic examples* on page 3-42.

See also *ADR* on page 3-23.

Note

ADDW is equivalent to the ADD syntax that uses the *imm12* operand. SUBW is equivalent to the SUB syntax that uses the *imm12* operand.

3.3.1 Operands

An instruction operand can be an ARM register, a constant, or another instruction-specific parameter. Instructions act on the operands and often store the result in a destination register. When there is a destination register in the instruction, it is usually specified before the operands.

Operands in some instructions are flexible in that they can either be a register or a constant. See *Flexible second operand*.

3.3.2 Restrictions when using PC or SP

Many instructions have restrictions on whether you can use the *Program Counter* (PC) or *Stack Pointer* (SP) for the operands or destination register. See instruction descriptions for more information.

Note

Bit[0] of any address you write to the PC with a `BX`, `BLX`, `LDM`, `LDR`, or `POP` instruction must be 1 for correct execution, because this bit indicates the required instruction set, and the Cortex-M4 processor only supports Thumb instructions.

3.3.3 Flexible second operand

Many general data processing instructions have a flexible second operand. This is shown as *`Operand2`* in the descriptions of the syntax of each instruction.

`Operand2` can be a:

- *Constant*
- *Register with optional shift* on page 3-13

Constant

You specify an Operand2 constant in the form:

`#constant`

where *`constant`* can be:

- any constant that can be produced by shifting an 8-bit value left by any number of bits within a 32-bit word
- any constant of the form `0x00XY00XY`
- any constant of the form `0xXY00XY00`
- any constant of the form `0xXYXYXYXY`.

Note

In the constants shown above, `X` and `Y` are hexadecimal digits.

In addition, in a small number of instructions, *`constant`* can take a wider range of values. These are described in the individual instruction descriptions.

When an Operand2 constant is used with the instructions `MOVS`, `MVNS`, `ANDS`, `ORRS`, `ORNS`, `EORS`, `BICS`, `TEQ` or `TST`, the carry flag is updated to bit[31] of the constant, if the constant is greater than 255 and can be produced by shifting an 8-bit value. These instructions do not affect the carry flag if Operand2 is any other constant.

Instruction substitution

Your assembler might be able to produce an equivalent instruction in cases where you specify a constant that is not permitted. For example, an assembler might assemble the instruction `CMP` *Rd*, `#0xFFFFFFFE` as the equivalent instruction `CMN` *Rd*, `#0x2`.

Register with optional shift

You specify an Operand2 register in the form:

Rm {, *shift*}

where:

Rm	The register holding the data for the second operand.
shift	An optional shift to be applied to *Rm*. It can be one of:

`ASR #`*n*	Arithmetic shift right *n* bits, 1 ≤ *n* ≤ 32.
`LSL #`*n*	Logical shift left *n* bits, 1 ≤ *n* ≤ 31.
`LSR #`*n*	Logical shift right *n* bits, 1 ≤ *n* ≤ 32.
`ROR #`*n*	Rotate right *n* bits, 1 ≤ *n* ≤ 31.
`RRX`	Rotate right one bit, with extend.
-	If omitted, no shift occurs, equivalent to `LSL #0`.

If you omit the shift, or specify `LSL #0`, the instruction uses the value in *Rm*.

If you specify a shift, the shift is applied to the value in *Rm*, and the resulting 32-bit value is used by the instruction. However, the contents in the register *Rm* remains unchanged. Specifying a register with shift also updates the carry flag when used with certain instructions. For information on the shift operations and how they affect the carry flag, see *Shift Operations*.

3.3.4 Shift Operations

Register shift operations move the bits in a register left or right by a specified number of bits, the *shift length*. Register shift can be performed:

- directly by the instructions `ASR`, `LSR`, `LSL`, `ROR`, and `RRX`, and the result is written to a destination register
- during the calculation of *Operand2* by the instructions that specify the second operand as a register with shift, see *Flexible second operand* on page 3-12. The result is used by the instruction.

The permitted shift lengths depend on the shift type and the instruction, see the individual instruction description or *Flexible second operand* on page 3-12. If the shift length is 0, no shift occurs. Register shift operations update the carry flag except when the specified shift length is 0. The following sub-sections describe the various shift operations and how they affect the carry flag. In these descriptions, *Rm* is the register containing the value to be shifted, and *n* is the shift length.

ASR

Arithmetic shift right by *n* bits moves the left-hand `32-`*n* bits of the register *Rm*, to the right by *n* places, into the right-hand `32-`*n* bits of the result. And it copies the original bit[31] of the register into the left-hand *n* bits of the result. See Figure 3-1 on page 3-14.