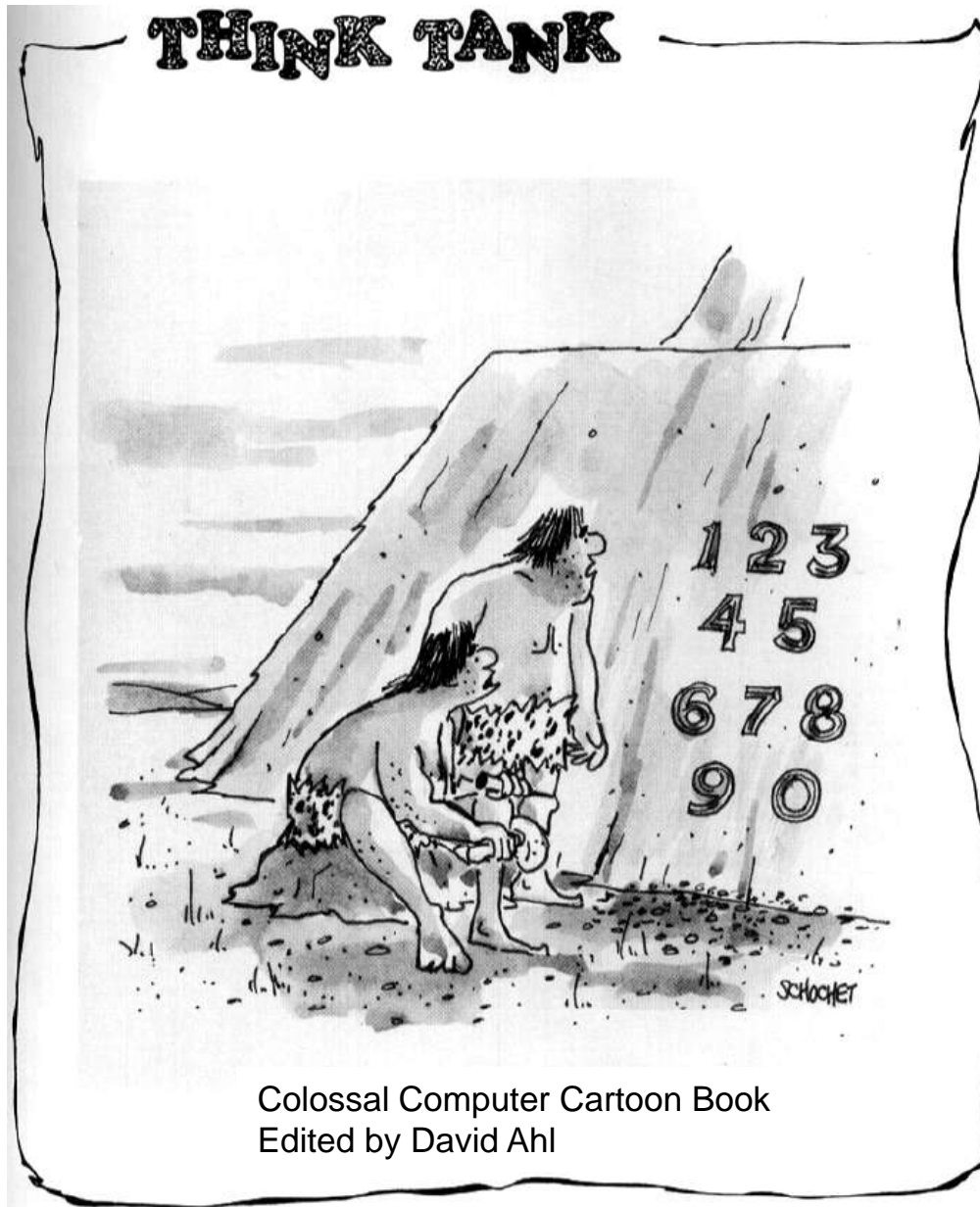


Arithmetic Instructions

Computer Engineering 1

**CT Team: A. Gieriet, J. Gruber, R. Gübeli, M. Meli, M. Rosenthal,
A. Rüst, J. Scheier, M. Thaler**

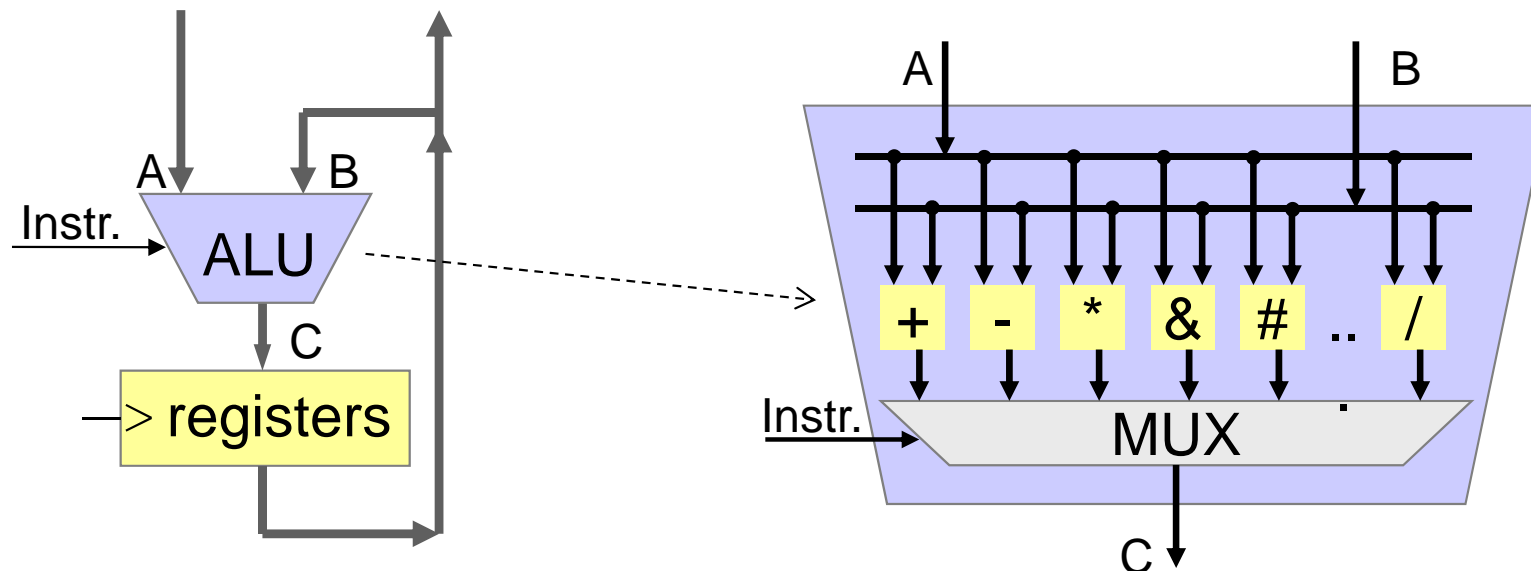
Motivation



"I call them numbers, you can add them, subtract them, multiply them, divide them ... find their square root ..."

■ Instructions to process data in the ALU

- **arithmetic** Addition, Subtraction, Multiplication, Division
- **logic** NOT, AND, OR, XOR
- **shift** Shift left/right. Fill with 0 or MSB
- **rotate** Cyclic shift left/right: What drops out enters on the other side



■ How often is the for loop executed?

- For `#define INCREMENT 1`
- For `#define INCREMENT 10`

```
...  
int main(void) {  
    uint8_t uc;  
    uint32_t count = 0;  
  
    for (uc = 0; uc < 255; uc += INCREMENT) {  
        printf("hello again %d \n",uc);  
        count++;  
    }  
    printf("Loop executed %d times\n",count);  
}
```

- **Cortex-M0**
 - Data flow
 - Flags
 - Overview of arithmetic instructions
- **Add Instructions Cortex-M0**
- **Negative Numbers**
- **Addition**
- **Subtraction**
- **Subtract Instructions Cortex-M0**
- **Multi-Word Arithmetic**
- **Multiplication**

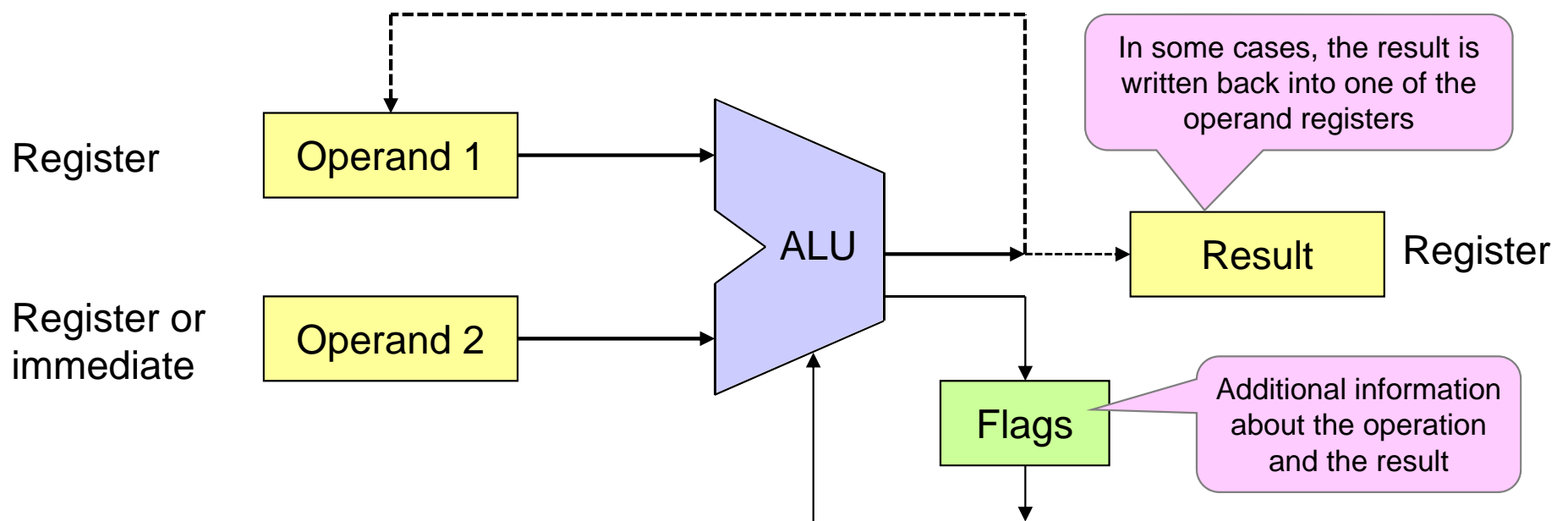
Learning Objectives

At the end of this lesson you will be able

- to enumerate and apply the Cortex-M0 arithmetic instructions
- to interpret Cortex-M0 assembly programs with arithmetic instructions
- to enumerate and explain the meaning of the Cortex-M0 Flags (N, Z, C, V)
- to carry out additions and subtractions of signed and unsigned integers and to explain the operations with the circle of numbers
- to calculate and interpret carry/borrow and overflow/underflow
- to determine (with the help of documents) the state of Cortex-M0 Flags (N, Z, C, V) after an arithmetic instruction
- to describe how addition and subtraction are done in hardware (in the ALU)
- to program integer calculations with operands that exceed the number of bits available in the ALU
- to explain how numbers in two's complements representation are multiplied

■ Operands and results stored in registers

- Exception: Immediate operations with data in OP-code
- Load/store architecture

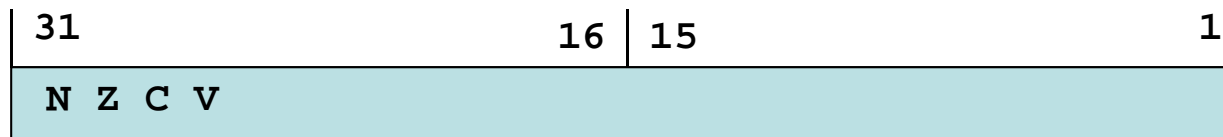


Examples

ADDS	R0,R0,R1	; R0 = R0 + R1	result back in R0
ADDS	R0,#0x34	; R0 = R0 + 0x34	
SUBS	R3,R4,#5	; R3 = R4 - 0x05	result in other reg

■ Cortex-M0

- APSR: Application Program Status Register



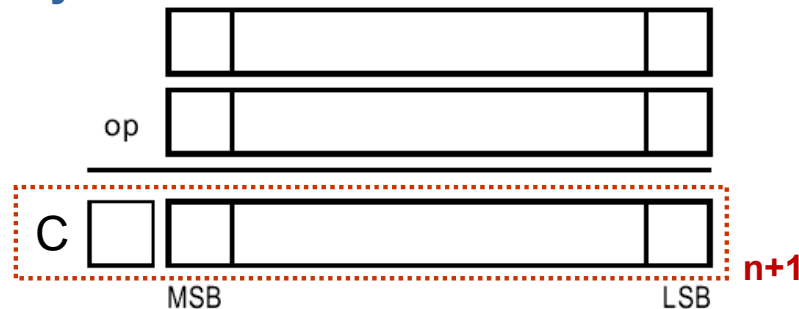
Flag	Meaning	Action	Operands
Negative	MSB = 1	N = 1	signed
Zero	Result = 0	Z = 1	signed , unsigned
Carry	Carry	C = 1	unsigned
Overflow	Overflow	V = 1	signed

- Processor does not know whether the user is working with **unsigned (C)** or with **signed (V)** numbers
→ Processor therefore always calculates C and V!

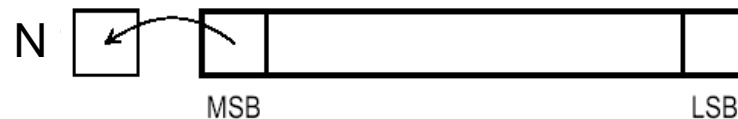
■ Cortex-M0

- Instructions ending with ,S' allow modification of flags
- Examples: **MOVS**, **ADDS**, **SUBS**, ...

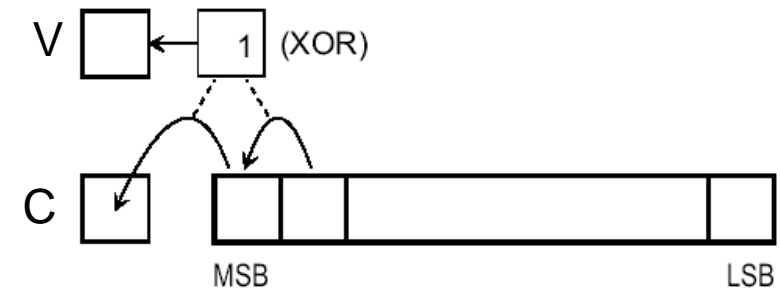
Carry



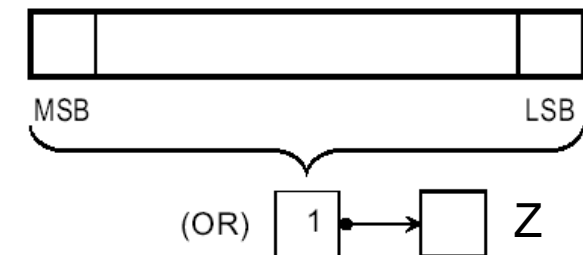
Negative



Overflow



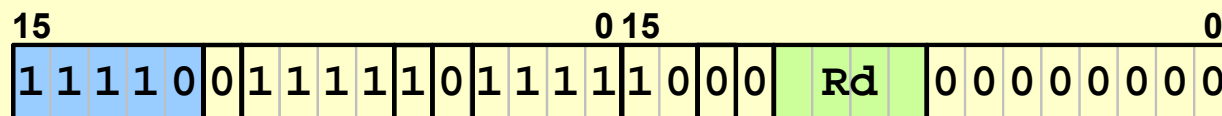
Zero



■ Instructions to access APSR

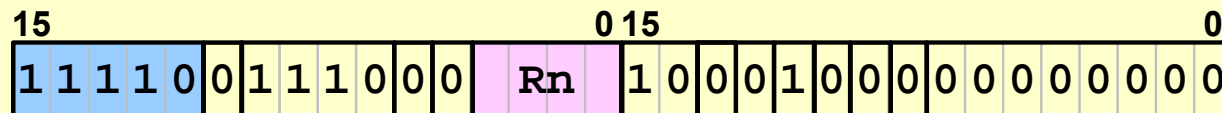
- **MRS** Move to register from special register (APSR)
- **MSR** Move to special register (APSR) from register
- 32-bit opcode

MRS <Rd>, APSR



Rd = APSR

MSR APSR, <Rn>



APSR = Rn

■ Overview Cortex-M0

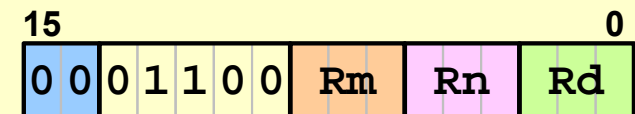
Mnemonic	Instruction	Function
ADD / ADDS	Addition	$A + B$
ADCS	Addition with carry	$A + B + c$
ADR	Address to Register	$PC + A$
SUB / SUBS	Subtraction	$A - B$
SBCS	Subtraction with carry (borrow)	$A - B - \text{NOT}(c)$ ¹⁾
RSBS	Reverse Subtract (negative)	$-1 \cdot A$
MULS	Multiplication	$A \cdot B$

¹⁾ borrow = NOT(c)

■ ADDS (register)

- Update of flags
- Result and two operands
- Only low register

ADDS <Rd> , <Rn> , <Rm>



$Rd = Rn + Rm$

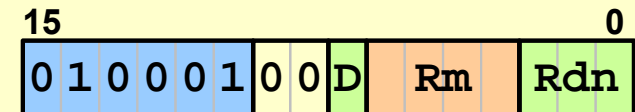
00000002	18D1	ADDS	R1,R2,R3	
00000004	1889	ADDS	R1,R1,R2	
00000006	1889	ADDS	R1,R2	; the same (dest = R1)
00000008		;ADDS	R9,R2	; not possible (high reg)
00000008		;ADDS	R1,R10	; not possible (high reg)

Add Instructions Cortex M0

■ ADD (register)

- No update of Flags
- High or low register
- $\langle Rdn \rangle \rightarrow$ result and operand
 - I.e. same register for operand and result!

ADD $\langle Rdn \rangle, \langle Rm \rangle$



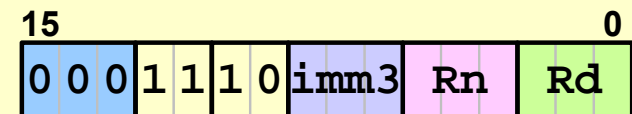
$Rdn = Rdn + Rm$

00000008	4411	ADD	R1,R1,R2	; low regs
0000000A	44D1	ADD	R9,R9,R10	; high regs
0000000C	44D1	ADD	R9,R10	; the same (dest = R9)
0000000E	4411	ADD	R1,R1,R2	
0000000E		;ADD	R1,R2,R3	; not possible

■ ADDS (immediate) – T1

- Update of flags
- Two different¹⁾ low registers and immediate value 0 - 7

ADDS <Rd>,<Rn>,<#imm3>



$Rd = Rn + \text{imm3}$

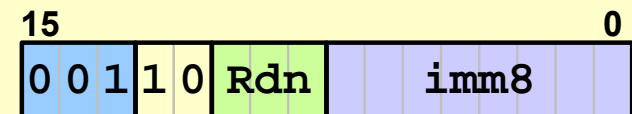
00000010 1D63	ADDS	R3,R4,#5	
00000012	;ADDS	R3,R4,#8	; out of range immediate
00000012	;ADDS	R10,R11,#5	; not possible (high reg)

¹⁾ If the same register is used for Rd and Rn, the assembler will choose the encoding T2 on the next slide

■ ADDS (immediate) – T2

- Update of flags
- Low register with immediate value 0 - 255d
- <Rdn> → Result and operand in same register

ADDS <Rdn>, #<imm8>



Rdn = Rdn + <imm8>

```
00000012 33F0  ADDS  R3,R3,#240
00000014 33F0  ADDS  R3,#240      ; the same (dest = R3)
00000016      ;ADDS  R8,R8,#240 ; not possible (high reg)
00000016      ;ADDS  R3,#260  ; out of range immediate
```

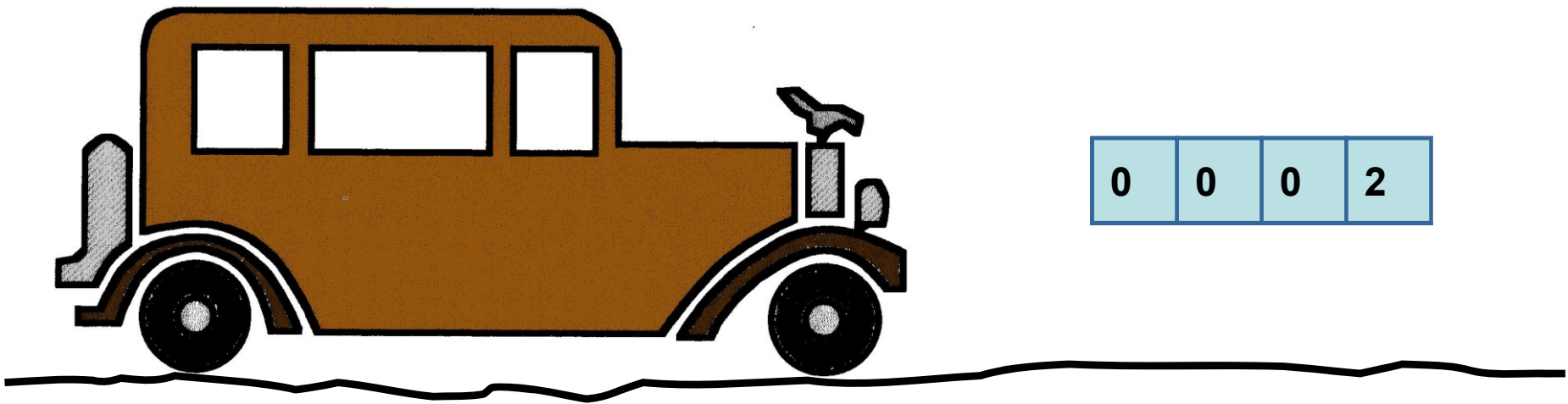
■ ADD / ADDS (Summary)

- ADD Flags not changed
- ADDS Flags changed according to Operation Result

Instr	Rd	Rn	Rm	imm	Restrictions
ADD	R0-R15	R0-R15	R0-R15	-	Rd and Rn must specify the same register. Rn and Rm must not both specify the PC (R15)
ADDS	R0-R7	R0-R7	-	0 - 7	-
ADDS	R0-R7	R0-R7	-	0 - 255	Rd and Rn must specify the same register
ADDS	R0-R7	R0-R7	R0-R7	-	-

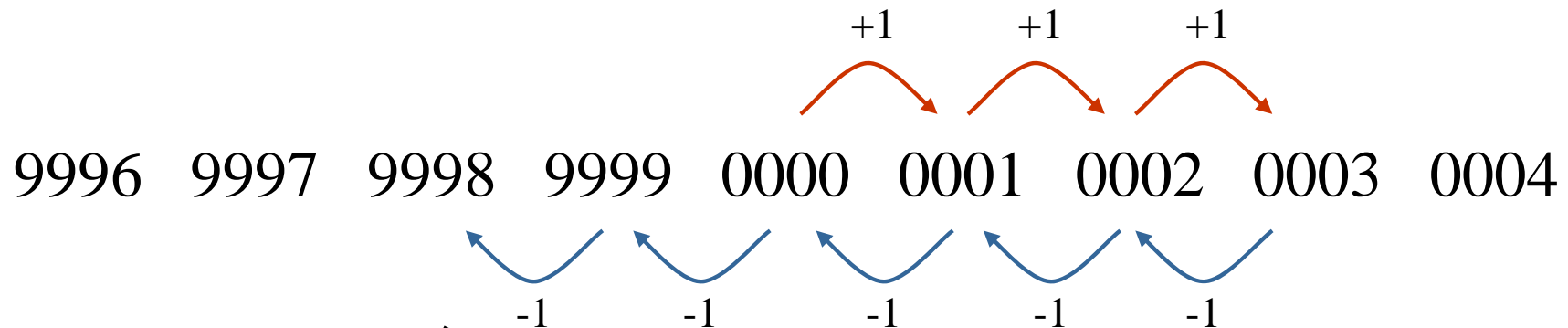
Negative Numbers


■ Lord Ardry's new Rolls Royce



Negative Numbers

■ What happened?




$$0000 - 1 - 1 \triangleq -2 \triangleq 9998$$

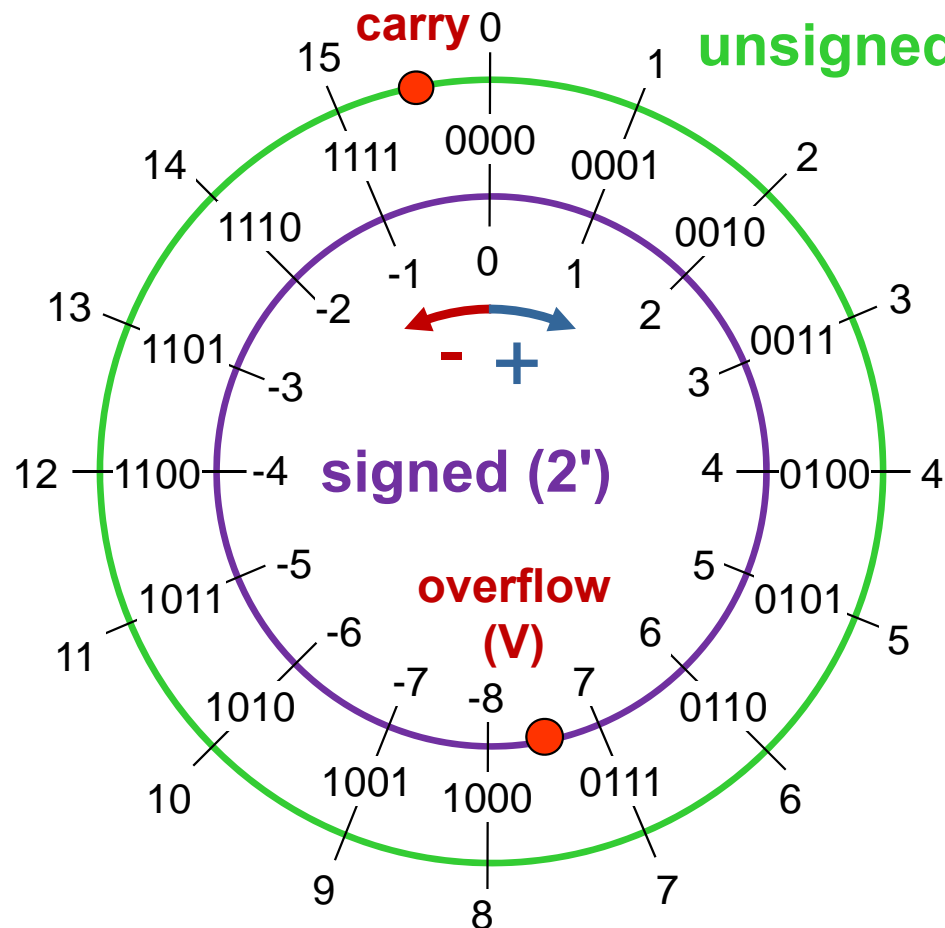
possible because

- number of digits is finite
- respectively given by the word size

Negative Numbers

signed $-b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0$

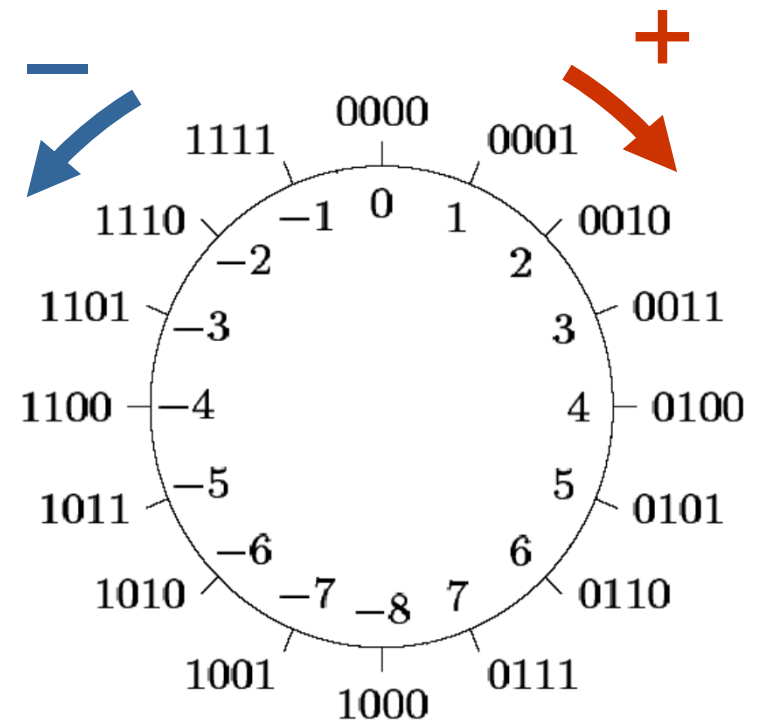
unsigned $+b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0$



binary	unsigned	signed 2'-Compl.
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

■ Numbers with finite number of digits

- can be represented on a circle
- Addition
 - Clockwise
- Subtraction
 - Counter-clockwise

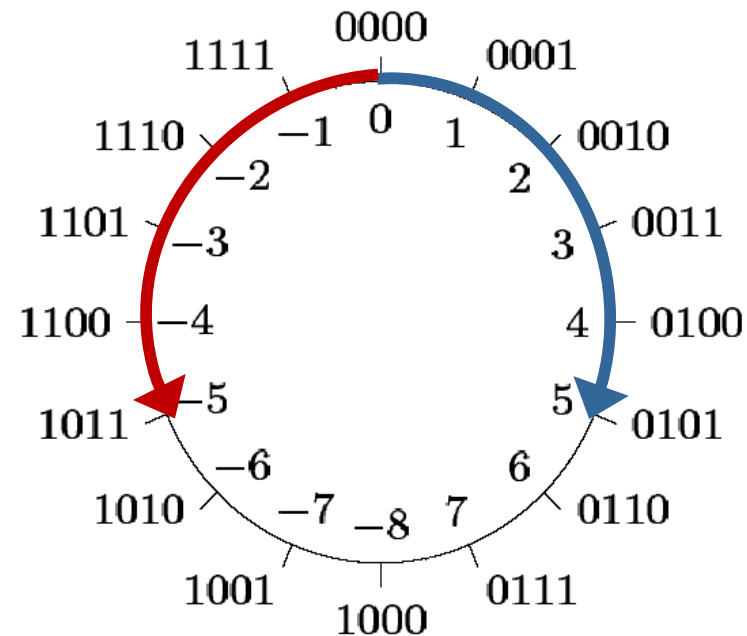


■ Negation of a number

- **Idea:** $-a = 0 - a$
- Starting at 0000 enter the absolute value of **a** counter-clockwise
- Example: $5d = 0101 \rightarrow -5d = 1011$

written calculation

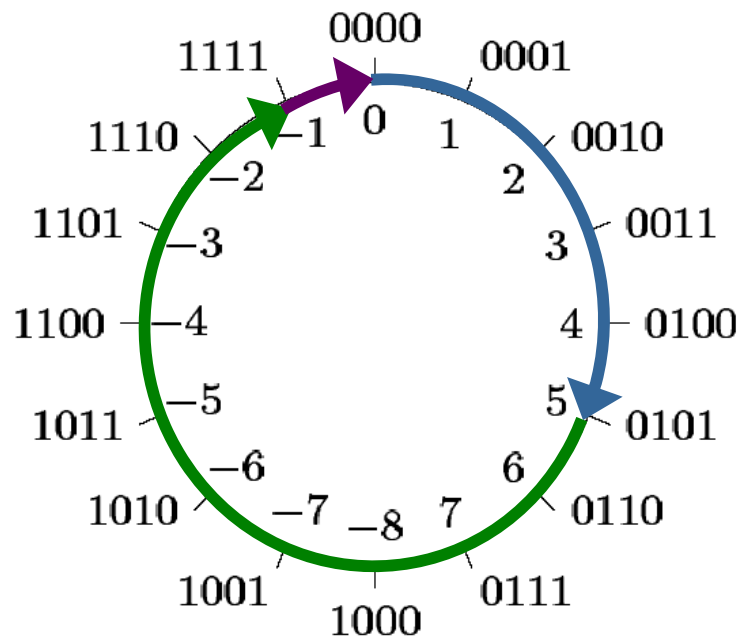
$$\begin{array}{r} 0000 \\ - 0101 \\ \hline 1\ 1\ 1\ 1 \\ \hline 1011 \\ \hline \hline \end{array}$$



Negative Numbers

■ 2' Complement

$$a - a = 0 \leftrightarrow a + \text{OC}(a) + 1 = 0$$

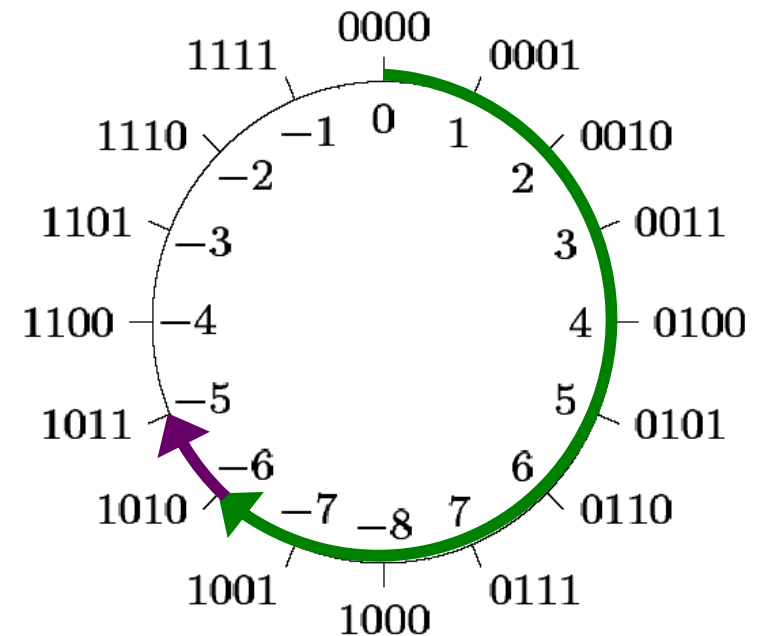


OC(a) is the bit-wise inverse of a

1111
- 0101
0000
1010

OC(0101) = 1010

$$-a = \text{OC}(a) + 1 = \text{TC}(a)$$



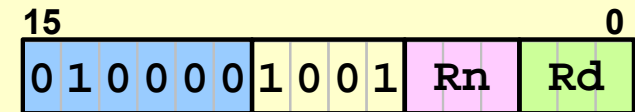
$$-5_{10} = \text{TC}(0101) = 1010 + 1 = 1011$$

OC: 1' complement
TC: 2' complement

■ Instruction RSBS

- *Reverse Subtract*
- Generates 2' complement
- Updates flags
- Only low register possible

RSBS <Rd>, <Rn>, #0



$Rd = 0 - Rn$

00000022	4251	RSBS	R1,R2,#0	
00000024	427F	RSBS	R7,R7,#0	
00000026	427F	RSBS	R7,#0	; the same (dest = R7)
00000028		;RSBS	R8,R1,#0	;not possible (high reg)
00000028		;RSBS	R1,R8,#0	;not possible (high reg)

■ Example: Instruction RSBS

C-Code

```
int32_t intA = 3;
int32_t intB;

int32_t invertSign(void)
{
    ...
    intB = -intA;
    ...
}
```

Assembler-code

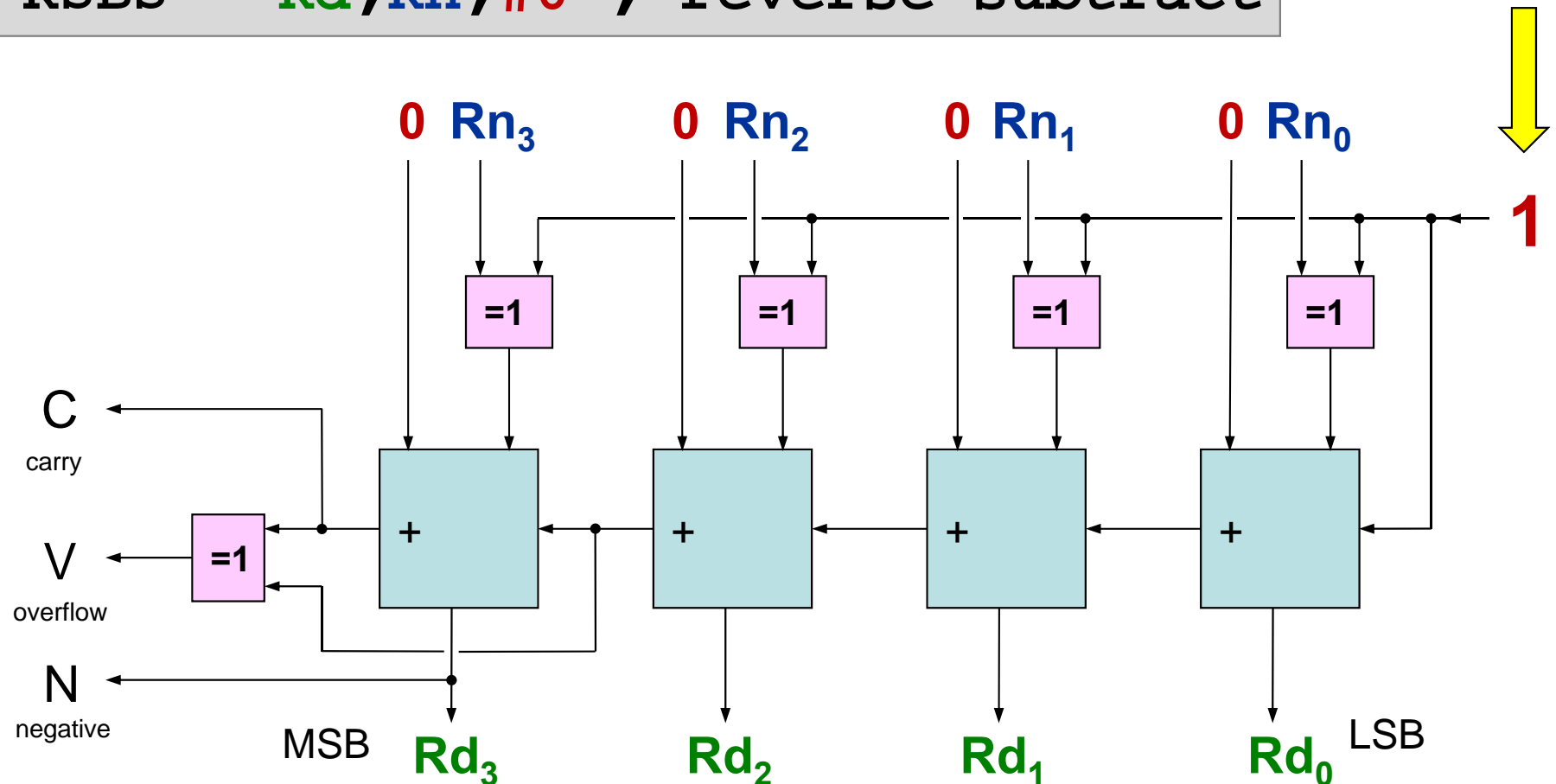
```
AREA ||.text||, CODE, READONLY
...
LDR    R0,=intA ❶
LDR    R0,[R0,#0]
RSBS   R0,R0,#0
LDR    R1,=intB ❶
STR    R0,[R1,#0]
...

AREA ||.data||, DATA
intA   DCD 0x00000003
intB   DCD 0x00000000
```


Negative Numbers

■ 2' complement in Hardware (ALU)

RSBS $Rd, Rn, \#0$; reverse subtract



Programmer interprets register content
either as signed or as unsigned.
CPU does not care!

```
ADDS R1,R2,R3
```

■ Unsigned

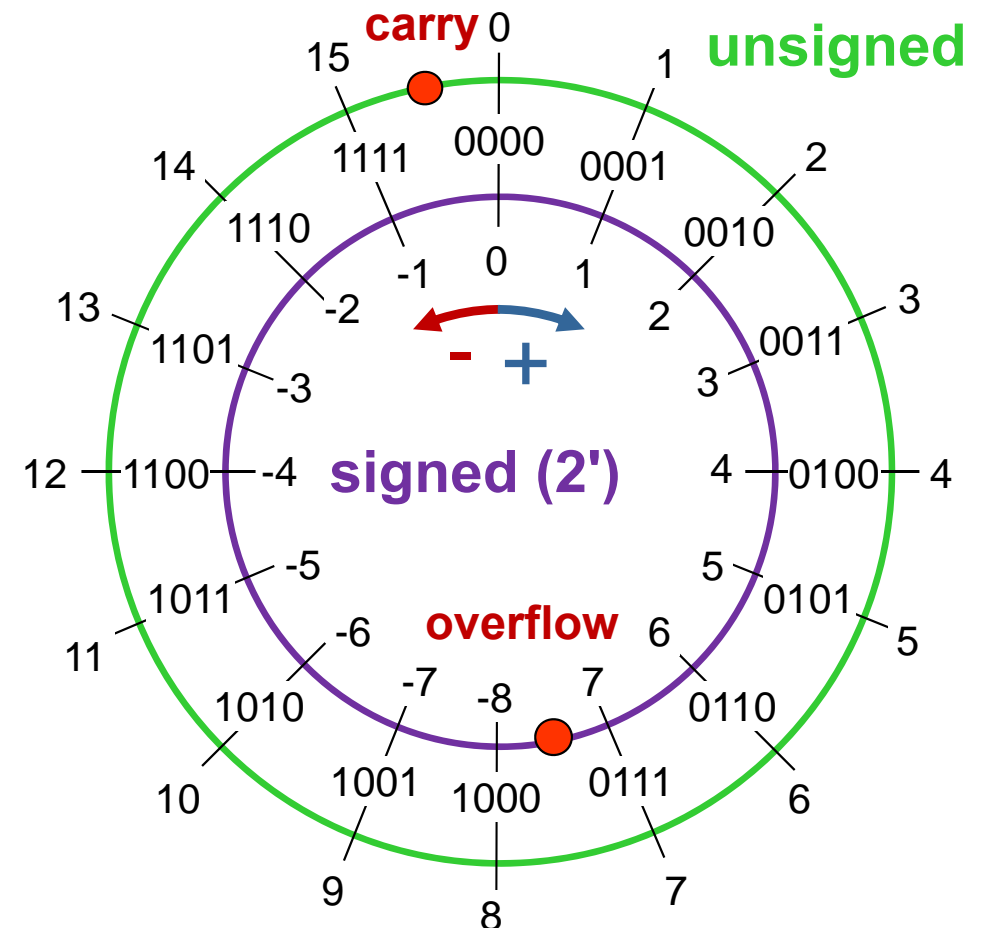
- C = 1 indicates carry
- V irrelevant
- Addition of 2 big numbers
→ can yield a small result

■ Signed

- V = 1 indicates overflow
 - possible with same „sign“



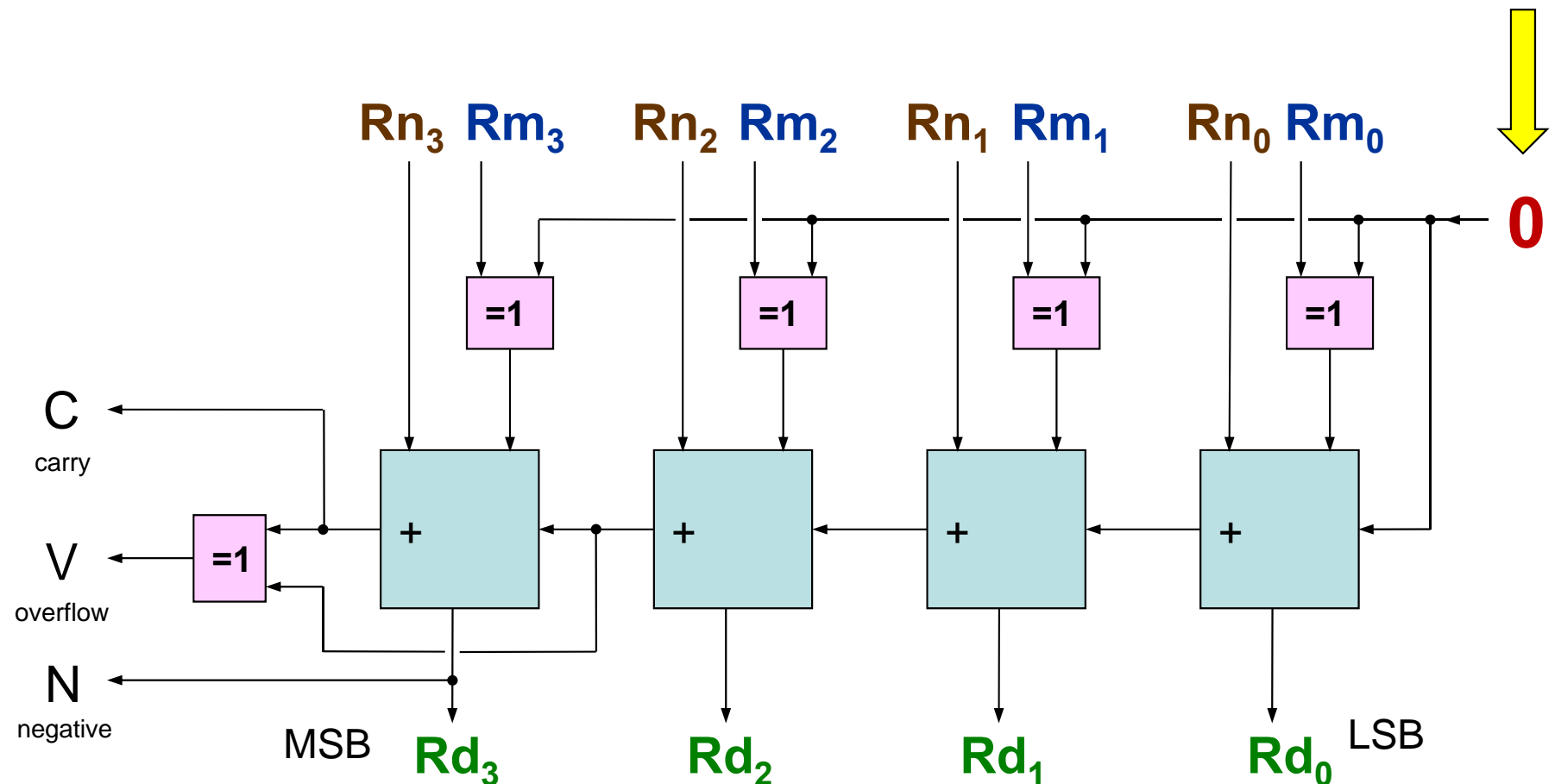
- C irrelevant



Addition

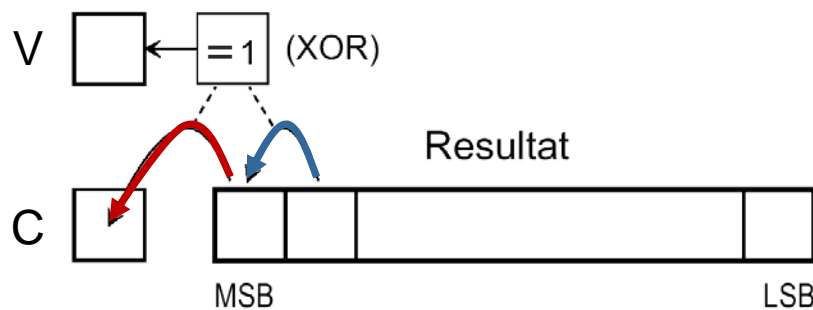
■ Addition in Hardware (ALU)

ADDS Rd, Rn, Rm



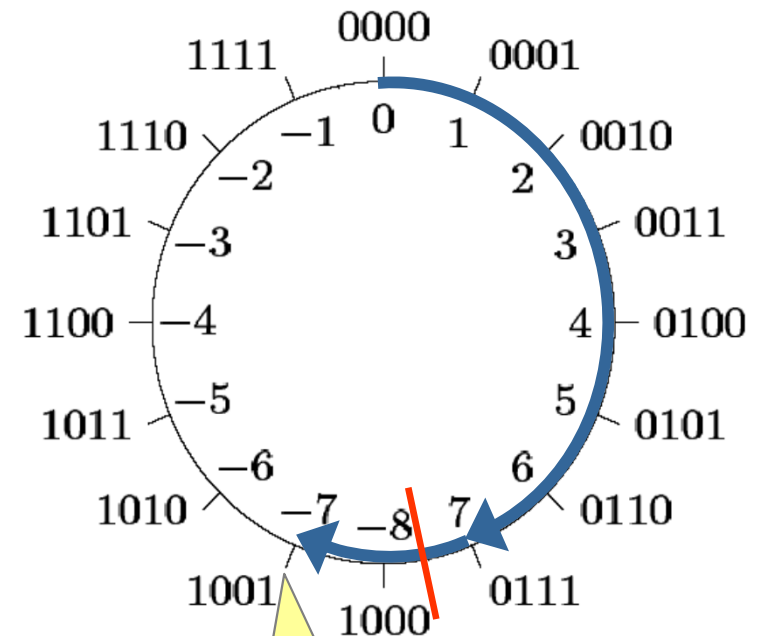
■ Signed

- Overflow: carry to the MSB has a different value than the carry from the MSB



- Examples for 4 cases

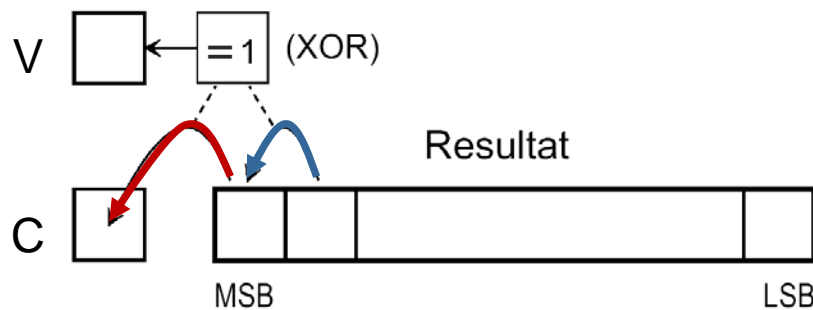
0 1	1 0	0 0	1 1
$\begin{array}{r} 0111 \\ +0010 \\ \hline 01110 \\ \hline 1001 \end{array}$	$\begin{array}{r} 1010 \\ +1101 \\ \hline 1000 \\ \hline 0111 \end{array}$	$\begin{array}{r} 0001 \\ +1110 \\ \hline 0000 \\ \hline 1111 \end{array}$	$\begin{array}{r} 0011 \\ +1111 \\ \hline 1111 \\ \hline 0010 \end{array}$



Number 9 cannot be represented

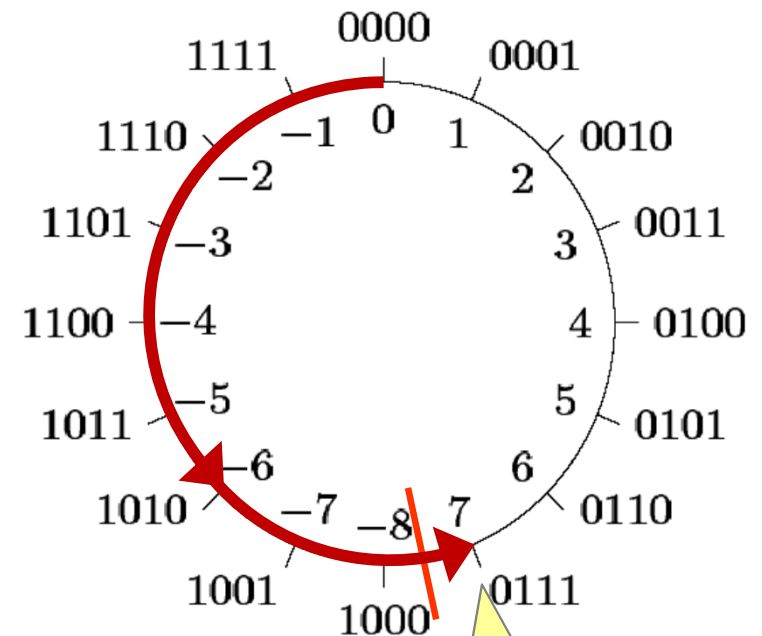
■ Signed

- Overflow: carry to the MSB has a different value than the carry from the MSB



- Examples for 4 cases

0 1	1 0	0 0	1 1
$\begin{array}{r} 0111 \\ +0010 \\ \hline 0110 \\ \hline 1001 \end{array}$	$\begin{array}{r} 1010 \\ +1101 \\ \hline 1000 \\ \hline 0111 \end{array}$	$\begin{array}{r} 0001 \\ +1110 \\ \hline 0000 \\ \hline 1111 \end{array}$	$\begin{array}{r} 0011 \\ +1111 \\ \hline 1111 \\ \hline 0010 \end{array}$



Number -9 cannot be represented

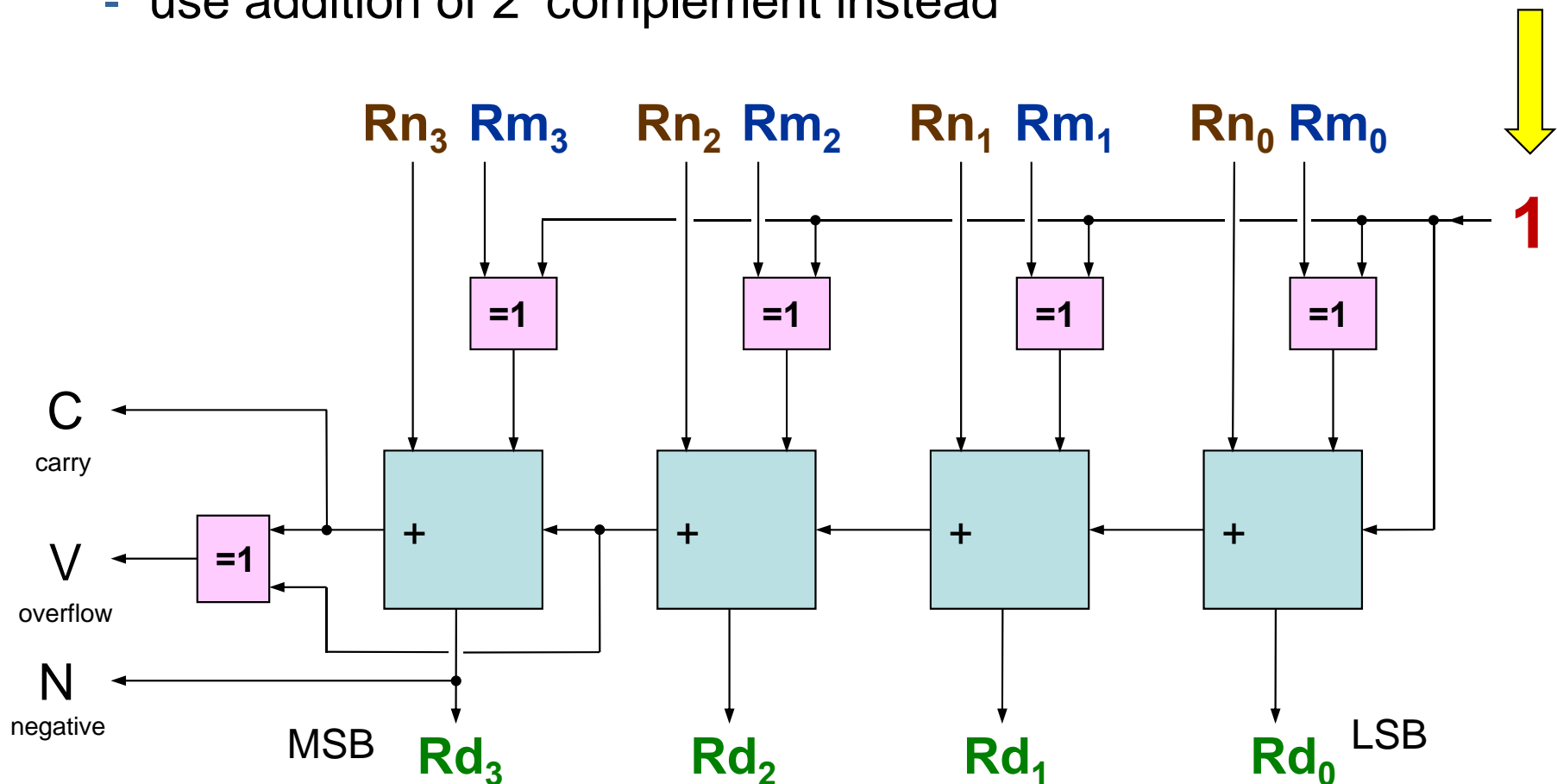
Subtraction

■ Subtraction in Hardware (ALU)

SUBS

Rd, Rn, Rm

- There is no subtraction!
 - use addition of 2's complement instead

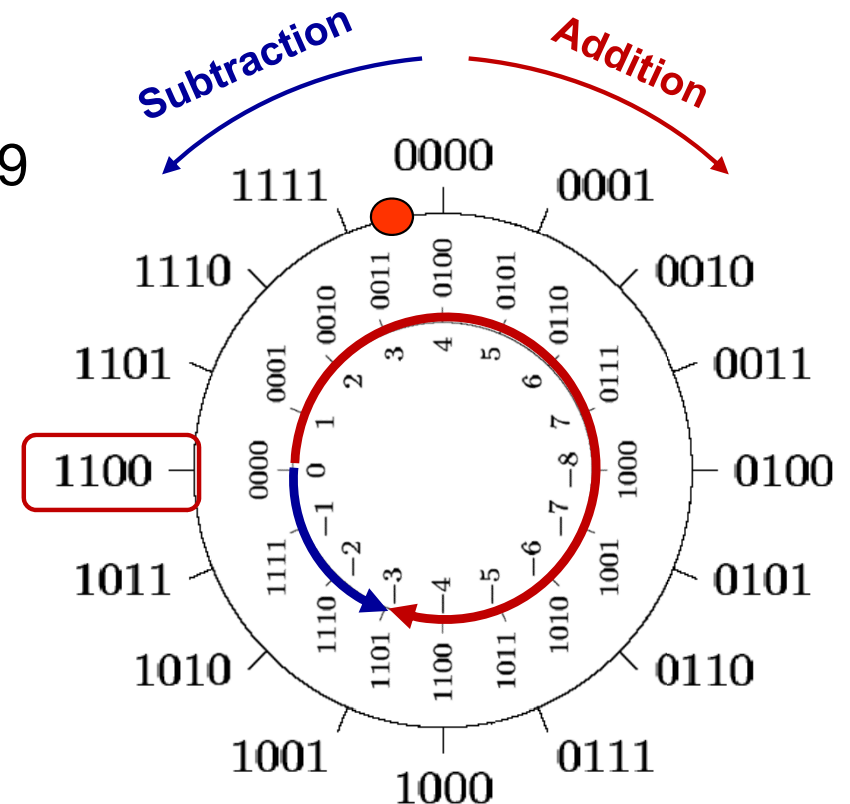


■ unsigned

- Use 2' complement as well
- Example 4-Bit unsigned: $12 - 3 = 9$

12	1100	1100
- 3	- 0011	+ 1101
		1 1
9	1001	1 1001
	human	computer

- Attention
 - $C = 1 \rightarrow \underline{\text{NO}}$ borrow
 - $C = 0 \rightarrow$ borrow
- Borrow
 - operation yields negative result \rightarrow cannot be represented in unsigned
 - in multi-word operations missing digits are borrowed from more significant word



Programmer interprets register content
either as signed or as unsigned.
CPU does not care!

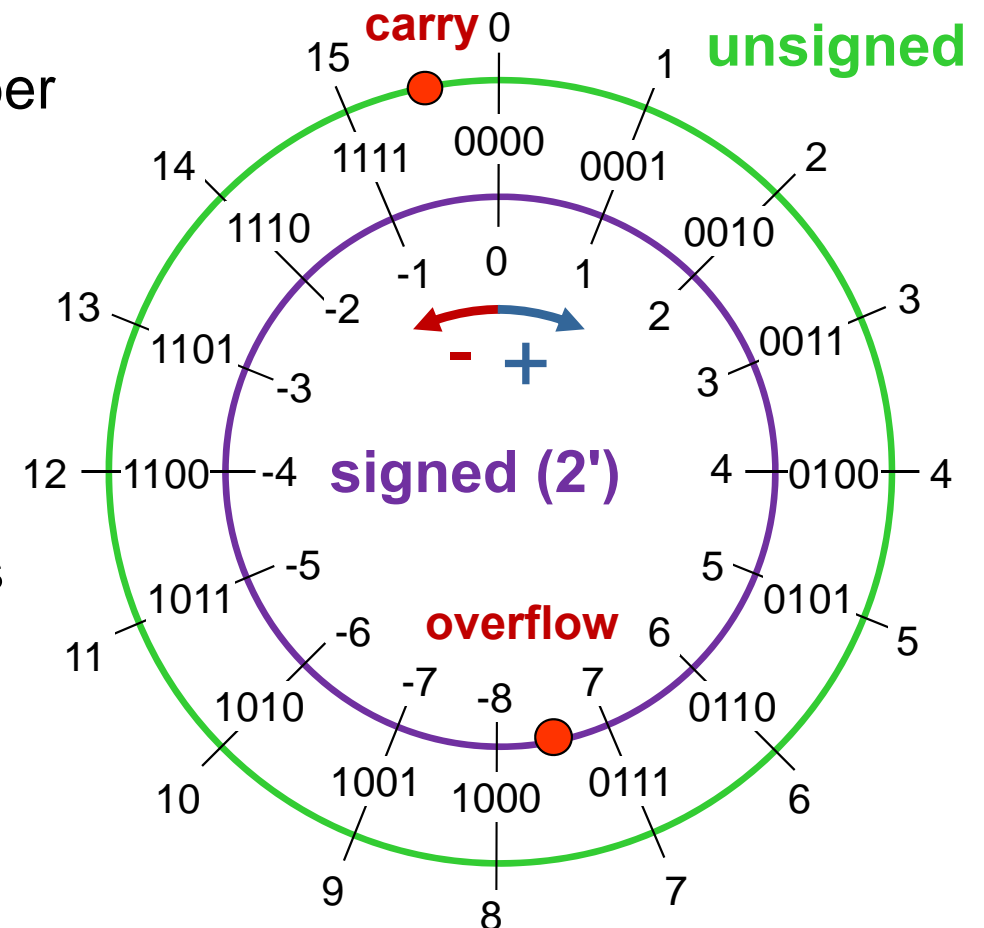
```
SUBS R1,R2,R3
```

■ Unsigned

- C = 0 indicates borrow
- V irrelevant
- Subtraction from a small number
→ can yield a big result

■ Signed

- V = 1 indicates overflow
or underflow
 - Possible with opposite signs
 - NOT possible when operands
have same sign
- C irrelevant

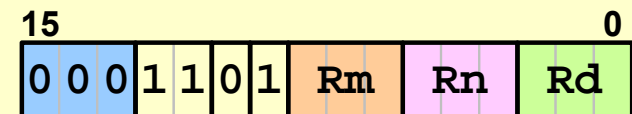


Subtraction Operations Cortex M0

■ SUBS (register)

- Updates flags
- Result and 2 operands
- Only low register

SUBS <Rd> , <Rn> , <Rm>



$$\begin{aligned} R_d &= R_n - R_m \\ &= R_n + \text{NOT}(R_m) + 1 \end{aligned}$$

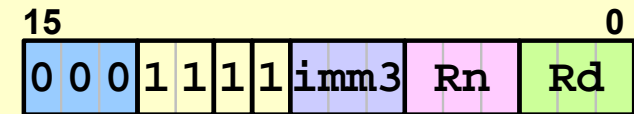
00000016	1AD1	SUBS	R1,R2,R3	
00000018	1B64	SUBS	R4,R4,R5	
0000001A	1B64	SUBS	R4,R5	; the same (dest = R4)
0000001C				
0000001C		;SUBS	R8,R4,R5	; not possible (high reg)
0000001C		;SUBS	R4,R8,R5	; not possible (high reg)
0000001C		;SUBS	R4,R5,R8	; not possible (high reg)

Subtraction Operations Cortex M0

■ SUBS (immediate) – T1

- Updates flags
- 2 different low registers and immediate value 0 - 7d

SUBS <Rd>, <Rn>, #<imm3>



$Rd = Rn - \text{<imm3>}$

$= Rn + \text{NOT<imm3>} + 1$

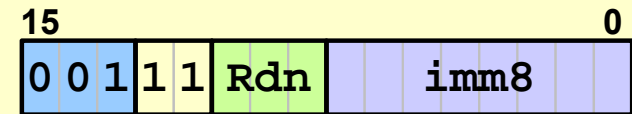
0000001C 1F63	SUBS	R3,R4,#5	
0000001E	;SUBS	R3,R4,#8	; out of range immediate
0000001E	;SUBS	R10,R11,#5	; not possible (high reg)

Subtraction Operations Cortex M0

■ SUBS (immediate) – T2

- Updates flags
- One low register and immediate value 0 - 255d
- <Rdn> → Result and operand
→ same register for result and operand

SUBS <Rdn>, #<imm8>



$Rdn = Rdn - \langle imm8 \rangle$

$= Rdn + \text{NOT}\langle imm8 \rangle + 1$

0000001E 3BF0	SUBS	R3,R3,#240	
00000020 3BF0	SUBS	R3,#240	; the same (dest = R3)
00000022	;SUBS	R8,R8,#240	; not possible (high reg)
00000022	;SUBS	R3,#260	; out of range immediate

Addition / Subtraction

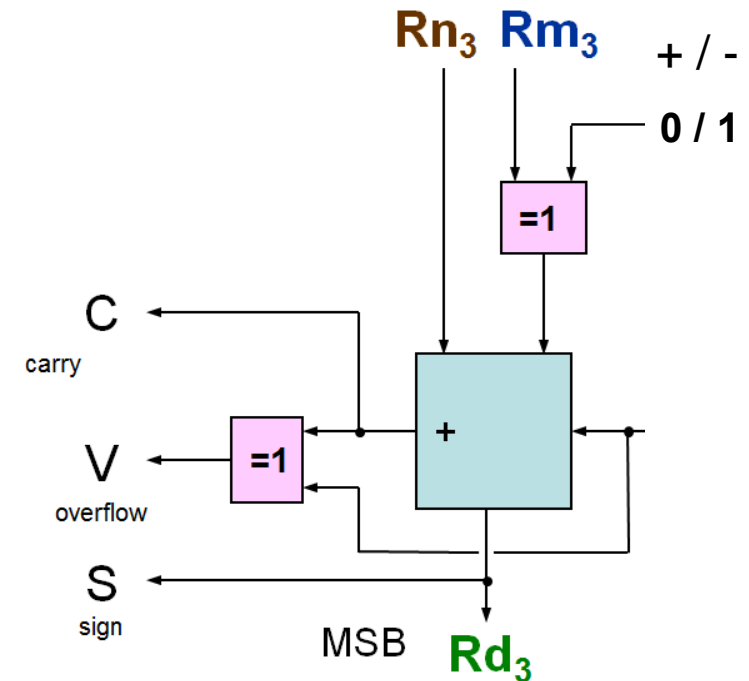
■ Interpretation of carry flag

- Addition $C = 1 \rightarrow \text{Carry}$

1	1	0	1	13d
0	1	1	1	7d
1	1	1	1	
1	0	1	0	0
				20d \rightarrow 16d + 4d

- Subtraction $C = 0 \rightarrow \text{Borrow}$

6d - 14d = 0110b - 1110b = 0110b + 0010b				
0	1	1	0	6d
0	0	1	0	2d = ZK(14d)
0	1	1	0	
0	1	0	0	0
				8d \rightarrow - 16d + 8d



■ **unsigned** Interpretation

- Program must check **carry flag (C)** after operation
- **C = 1 for Addition** **C = 0 for Subtraction**
 - Result cannot be represented (not enough digits / no negative numbers)
 - Full turn on number circle must be added or subtracted
→ odometer effect
- Overflow flag (V) irrelevant

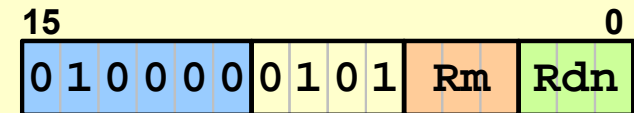
■ **signed** Interpretation

- Program must check **overflow flag (V)** after operation
- **V = 1** means
 - Not enough digits available to represent the result
 - Full turn on number circle must be added or subtracted
→ odometer effect
- Carry flag (C) irrelevant

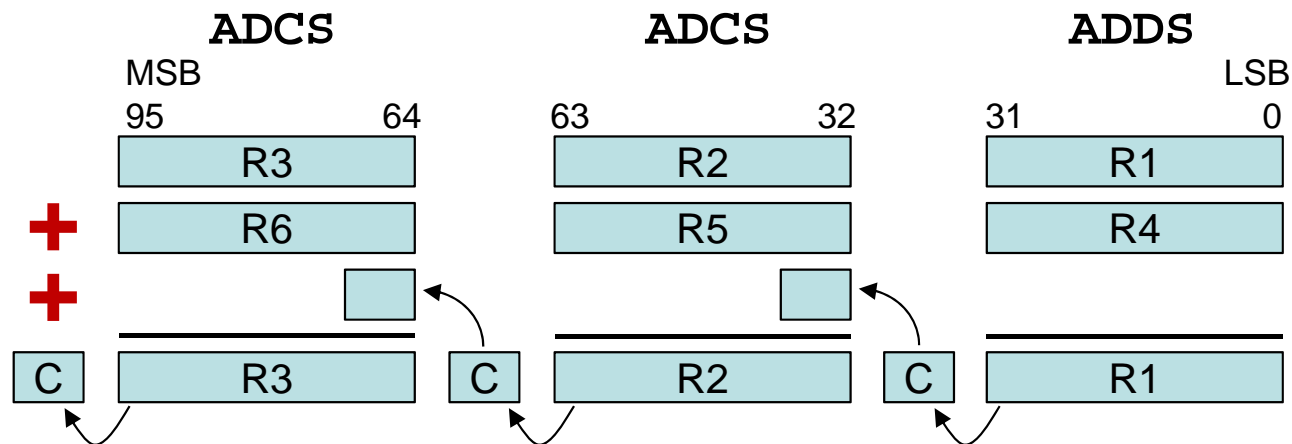
■ Multi-Word Addition ADCS

- Example: Addition of two 96-bit Operands

ADCS <Rdn>, <Rm>



$$Rdn = Rdn + Rm + C$$

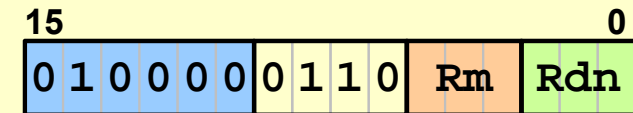


```
; Operand A:      96-bit in R3(MSW),R2,R1(LSW)
; Operand B:      96-bit in R6(MSW),R5,R4(LSW)
; Result = A + B:  96-bit in R3(MSW),R2,R1(LSW)
00000028 1909      ADDS      R1,R1,R4
0000002A 416A      ADCS      R2,R2,R5
0000002C 4173      ADCS      R3,R3,R6
```

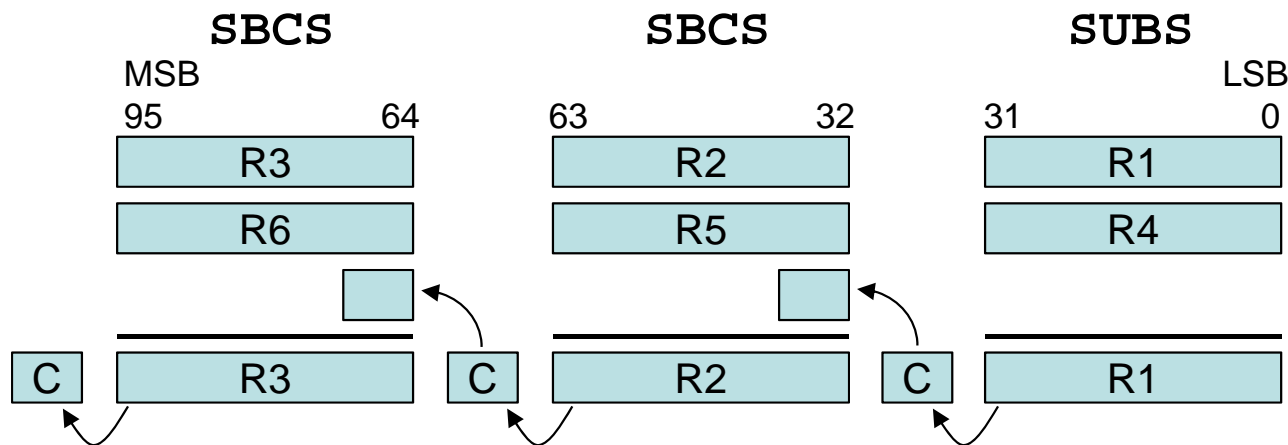
■ Multi-Word Subtraction SBCS

- Example: Subtraction of two 96-bit Operands

SBCS <Rdn>, <Rm>



$$\begin{aligned} Rdn &= Rdn - Rm - NOT(C) \\ &= Rdn + NOT(Rm) + C \quad 1) \end{aligned}$$



```
; Operand A: 96-bit in R3(MSW),R2,R1(LSW)
; Operand B: 96-bit in R6(MSW),R5,R4(LSW)
; Result A - B: 96-bit in R3(MSW),R2,R1(LSW)
0000002E 1B09 SUBS R1, R1, R4
00000030 41AA SBCS R2, R2, R5
00000032 41B3 SBCS R3, R3, R6
```

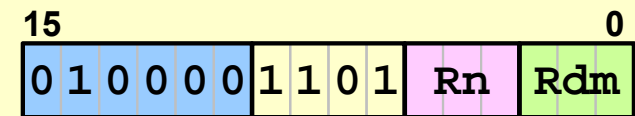
1) $-Rm - NOT(C)$
 $= (NOT(Rm) + 1) - NOT(C)$
 $= NOT(Rm) + C$

The CPU actually calculates
 $Rdn + NOT(Rm) + C$

■ MULS (register)

- Flags
 - N and Z updated
 - C and V unchanged
- Operands
 - Rn multiplicand
 - Rdm multiplier
 - only low registers
- Result
 - **Rdm contains only lowest 32 bits of product**

MULS <Rdm> , <Rn> , <Rdm>



$Rdm = Rn * Rdm$

0000002E 4351 MULS R1,R2,R1

00000030 ;MULS R1,R1,R2 ; not possible: destination and
00000030 ; 2nd source must be same
00000030 ;MULS R1,R8,R1 ; not possible: high reg

- **Result requires twice as many binary digits**
 - Example 4-bit * 4-bit → 8-bit result
- **Signed and unsigned multiplication are different**

```

0101 * 0011
-----
      0011
     0000
    0011
   0000
  -----
00001111
    
```

Interpretation **unsigned**
5d * 3d = 15d → correct

Interpretation **signed**
5d * 3d = 15d → correct

unsigned

```

0101 * 1101
-----
 00001101
00000000
 00001101
00000000
  -----
0000100001
    
```

zero extension
of multiplier

Interpretation **unsigned**
5d * 13d = 65d → correct

Interpretation **signed**
5d * -3d = -15d → wrong

signed

```

0101 * 1101
-----
 11111101
00000000
 11111101
00000000
  -----
1001111001
    
```

sign extension
of multiplier

Interpretation **unsigned**
5d * 13d = 241d → wrong

Interpretation **signed**
5d * -3d = -15d → correct

■ MULS on Cortex-M0

- R_n (32-bit) * R_{dm} (32-bit) \rightarrow R_{dm} (32-bit)
 - Upper 32-bit of result are lost!
 - Lower 32-bit are the same for unsigned and signed
- unsigned \rightarrow watch out if result requires more than 32 bits
- signed \rightarrow part with sign bit is missing

■ Processor Arithmetic

- Odometer effect because of finite word length

■ Processors do **not** distinguish **signed** and **unsigned**

- User (resp. compiler) has to know, whether he is working with signed or unsigned numbers
- Processor always calculates flags for both cases
 - carry (C) unsigned
 - overflow (V) signed
negative (N)

■ unsigned

- Addition → $C = 1 \rightarrow$ carry
result too large for available bits
- Subtraction → $C = 0 \rightarrow$ borrow
result less than zero → no negative numbers

■ signed

- Addition → potential overflow in case of operands with equal signs
- Subtraction → potential overflow in case of operands with opposite signs

■ Arithmetic Instructions

- | | | |
|--------------|------|------|
| • ADD / ADDS | ADCS | ADR |
| SUB / SUBS | SBCS | RSBS |
| MULS | | |