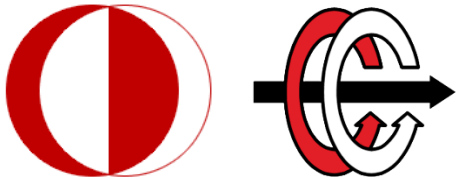
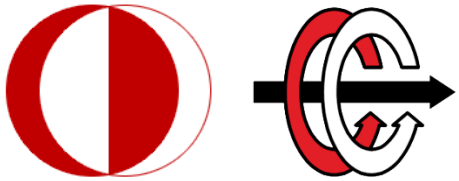


EE447 Introduction to Microprocessors

Fall 2023/24



Assembly Programming Details

Week 3

Cortex-M4: Overview

Cortex-M4

Revision r0p0

Technical Reference Manual

- Chapter 2 Functional Description
- Chapter 3 Programmers Model
- Chapter 4 System Control
- Chapter 6 Nested Vectored Interrupt Controller

Thumb2 Instructions: Details

ARM®v7-M Architecture Reference Manual

- Chapter A4 The ARMv7-M Instruction Set
- Chapter A5 The Thumb Instruction Set Encoding
- Chapter A7 Instruction Details

Thumb2 Instructions: Overview

- Data Movement Operations
  - Memory-to-register and register-to-memory
    - Includes different memory “addressing” options
    - “memory” includes peripheral function registers
  - Register-to-register
  - Constant-to-register (or to memory in some CPUs)
- Arithmetic operations
  - Add/subtract/multiply/divide
  - Multi-precision operations (more than 32 bits)
- Logical operations
  - And/or/exclusive-or/complement (between operand bits)
  - Shift/rotate
  - Bit test/set/reset

Thumb2 Instructions: Overview

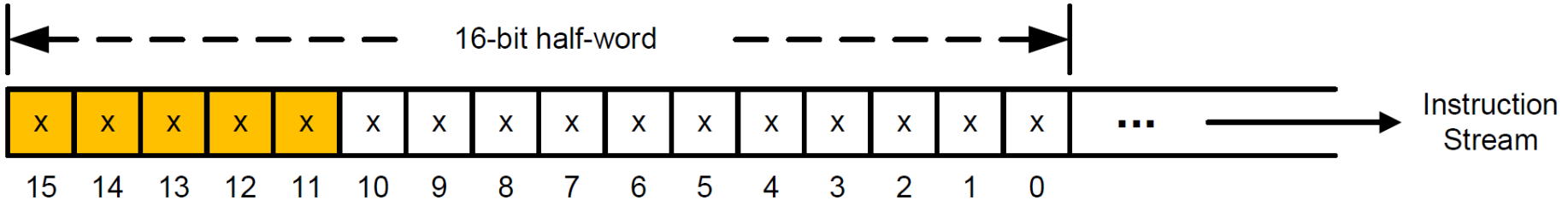
- Flow control operations
  - Branch to a location (conditionally or unconditionally)
  - Branch to a subroutine/function
  - Return from a subroutine/function
- Miscellaneous
  - Wait for events
  - Interrupts
  - Others

Thumb2 Instructions: Overview

- Some Facts
  - Most instructions are 16 bits long, some are 32 bits (Thumb2)
  - Half-word aligned instructions
  - Most 16-bit instructions can only access low registers (R0-R7)
  - Some 16-bit instructions can access high registers (R8-R15)
  - Some instructions can be followed by suffixes
  - There can be multiple encodings for various instructions
  - Instructions are unconditional but can be executed conditionally
    - Branching
    - IF-THEN (IT) blocks

Thumb2 Instructions: Instruction Length

- Instruction Stream



- 32-bit Instructions
  - bit[15-11] = 11101, 11110, or 11111
    - → This half-word is the first half-word of a 32-bit instruction
- Otherwise, this half-word is a 16-bit instruction

Thumb2 Instructions: Syntax Fields

Standard assembler syntax fields

The following assembler syntax fields are standard across all or most instructions:

<c> Is an optional field. It specifies the condition under which the instruction is executed. If <c> is omitted, it defaults to *always* (AL). For details see *Conditional instructions* on page A4-102.

<q> Specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

- .N Meaning narrow, specifies that the assembler must select a 16-bit encoding for the instruction. If this is not possible, an assembler error is produced.
- .W Meaning wide, specifies that the assembler must select a 32-bit encoding for the instruction. If this is not possible, an assembler error is produced.

If neither .W nor .N is specified, the assembler can select either 16-bit or 32-bit encodings. If both are available, it must select a 16-bit encoding. In a few cases, more than one encoding of the same length can be available for an instruction. The rules for selecting between such encodings are instruction-specific and are part of the instruction description.

- Examples

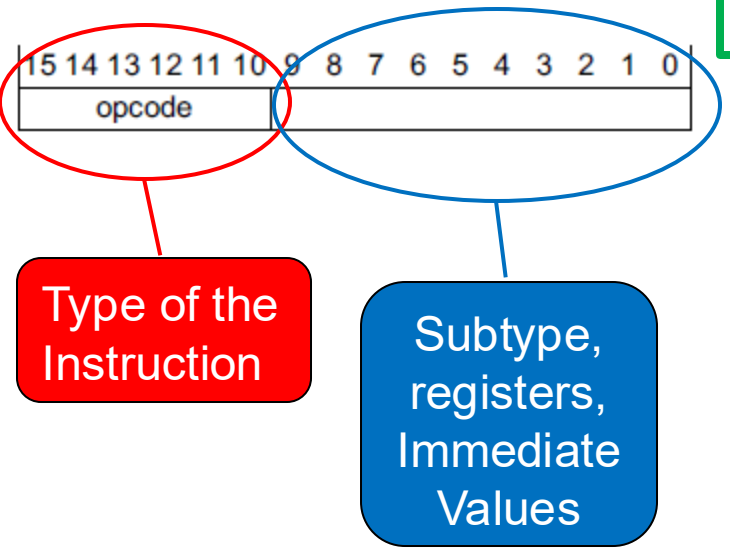
```
STR<c> <Rt>,[<Rn>,<Rm>]
STR<c>.W <Rt>,[<Rn>,<Rm>{,LSL #<imm2>}]
```

Thumb2 Instructions: Condition Flags

cond	Mnemonic extension	Meaning, integer arithmetic	Meaning, floating-point arithmetic[a]	Condition flags
0000	EQ	Equal	Equal	Z == 1
0001	NE	Not equal	Not equal, or unordered	Z == 0
0010	CS[b]	Carry set	Greater than, equal, or unordered	C == 1
0011	CC[c]	Carry clear	Less than	C == 0
0100	MI	Minus, negative	Less than	N == 1
0101	PL	Plus, positive or zero	Greater than, equal, or unordered	N == 0
0110	VS	Overflow	Unordered	V == 1
0111	VC	No overflow	Not unordered	V == 0
1000	HI	Unsigned higher	Greater than, or unordered	C == 1 and Z == 0
1001	LS	Unsigned lower or same	Less than or equal	C == 0 or Z == 1
1010	GE	Signed greater than or equal	Greater than or equal	N == V
1011	LT	Signed less than	Less than, or unordered	N != V
1100	GT	Signed greater than	Greater than	Z == 0 and N == V
1101	LE	Signed less than or equal	Less than, equal, or unordered	Z == 1 or N != V
1110	None (AL)[d]	Always (unconditional)	Always (unconditional)	Any

Encoding 16-bit Thumb Instructions

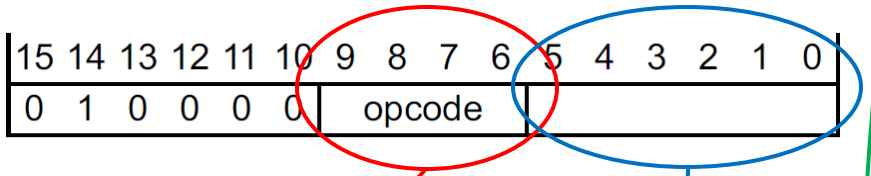
- General (A5.2)



opcode	Instruction or instruction class
00xxxx	*Shift (immediate), add, subtract, move, and compare* on page A5-128
010000	*Data processing* on page A5-129
010001	*Special data instructions and branch and exchange* on page A5-130
01001x	Load from Literal Pool, see *LDR (literal)* on page A7-254
0101xx 011xxx 100xxx	*Load/store single data item* on page A5-131
10100x	Generate PC-relative address, see *ADR* on page A7-197
10101x	Generate SP-relative address, see *ADD (SP plus immediate)* on page A7-193
1011xx	*Miscellaneous 16-bit instructions* on page A5-132
11000x	Store multiple registers, see *STM, STMIA, STMEA* on page A7-422
11001x	Load multiple registers, see *LDM, LDMIA, LDMFD* on page A7-248
1101xx	*Conditional branch, and supervisor call* on page A5-134
11100x	Unconditional Branch, see *B* on page A7-207

Encoding 16-bit Thumb Instructions: Examples

- Data Processing (A.5.2.2)



Type of the Instruction

Registers, Immediate Values

16 = 2^4 Instructions

opcode	Instruction	See
0000	Bitwise AND	*AND (register)* on page A7-201
0001	Exclusive OR	*EOR (register)* on page A7-239
0010	Logical Shift Left	*LSL (register)* on page A7-300
0011	Logical Shift Right	*LSR (register)* on page A7-304
0100	Arithmetic Shift Right	*ASR (register)* on page A7-205
0101	Add with Carry	*ADC (register)* on page A7-187
0110	Subtract with Carry	*SBC (register)* on page A7-380
0111	Rotate Right	*ROR (register)* on page A7-368
1000	Set flags on bitwise AND	*TST (register)* on page A7-466
1001	Reverse Subtract from 0	*RSB (immediate)* on page A7-372
1010	Compare Registers	*CMP (register)* on page A7-231
1011	Compare Negative	*CMN (register)* on page A7-227
1100	Logical OR	*ORR (register)* on page A7-336
1101	Multiply Two Registers	*MUL* on page A7-324
1110	Bit Clear	*BIC (register)* on page A7-213
1111	Bitwise NOT	*MVN (register)* on page A7-328

Encoding 16-bit Thumb Instructions: AND (A.7.7.9)

AND (register)

AND (register) performs a bitwise AND of a register value and an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 All versions of the Thumb instruction set.

```
ANDS    <Rdn>,<Rm>      Outside IT block.
AND<c>  <Rdn>,<Rm>      Inside IT block.
```

1. source register

2. source and destination register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	0	0	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding for 16 bit instruction

Encoding T2 ARMv7-M

```
AND{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	0	0	0	0	S	Rn				(0)	imm3			Rd				imm2		type		Rm			

```
if Rd == '1111' && S == '1' then SEE TST (register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d == 13 || (d == 15 && S == '0') || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Encoding for 32 bit instruction (see later)

Encoding 16-bit Thumb Instructions: AND

Assembler syntax

`AND{S}<c><q>  {<Rd>,} <Rn>, <Rm> {,<shift>}`

where:

S	If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.
<c><q>	See *Standard assembler syntax fields* on page A7-175.
<Rd>	Specifies the destination register. If <Rd> is omitted, this register is the same as <Rn>.
<Rn>	Specifies the register that contains the first operand.
<Rm>	Specifies the register that is optionally shifted and used as the second operand.
<shift>	Specifies the shift to apply to the value read from <Rm>. If <shift> is omitted, no shift is applied and both encodings are permitted. If <shift> is specified, only encoding T2 is permitted. The possible shifts and how they are encoded are described in *Shifts applied to a register* on page A7-180.

A special case is that if `AND<c> <Rd>,<Rn>,<Rd>` is written with <Rd> and <Rn> both in the range R0-R7, it will be assembled using encoding T2 as though `AND<c> <Rd>,<Rn>` had been written. To prevent this happening, use the .W qualifier.

The pre-UAL syntax `AND<c>S` is equivalent to `ANDS<c>`.

Encoding 16-bit Thumb Instructions: AND

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] AND shifted;
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A.7.4.2

```
// Shift_C()
// =========

(bits(N), bit) Shift_C(bits(N) value, SRType type, integer amount, bit carry_in)
    assert !(type == SRType_RRX && amount != 1);

    if amount == 0 then
        (result, carry_out) = (value, carry_in);
    else
        case type of
            when SRType_LSL
                (result, carry_out) = LSL_C(value, amount);
            when SRType_LSR
                (result, carry_out) = LSR_C(value, amount);
            when SRType_ASR
                (result, carry_out) = ASR_C(value, amount);
            when SRType_ROR
                (result, carry_out) = ROR_C(value, amount);
            when SRType_RRX
                (result, carry_out) = RRX_C(value, carry_in);

    return (result, carry_out);
```

Encoding 16-bit Thumb Instructions: AND

- Example ANDS <Rdn>,<Rm>


```
ANDS   R2, R5
```

Machine Code

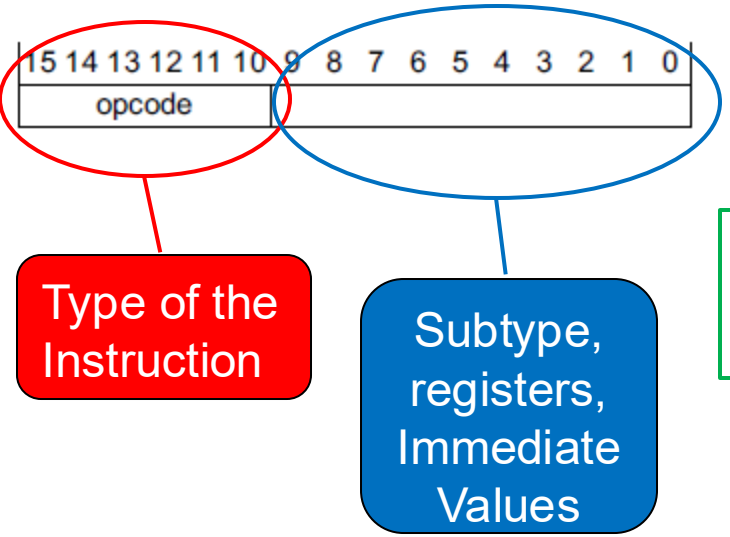
0	1	0	0	0	0	0	0	0	0	1	0	1	0	1	0

- Computation

```
R2 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
R5 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 1
-------------------------------------------------------------------
R2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
```

Encoding 16-bit Thumb Instructions

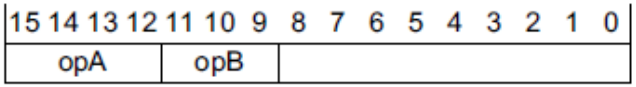
- General



opcode	Instruction or instruction class
00xxxx	*Shift (immediate), add, subtract, move, and compare* on page A5-128
010000	*Data processing* on page A5-129
010001	*Special data instructions and branch and exchange* on page A5-130
01001x	Load from Literal Pool, see *LDR (literal)* on page A7-254
0101xx 011xxx 100xxx	*Load/store single data item* on page A5-131
10100x	Generate PC-relative address, see *ADR* on page A7-197
10101x	Generate SP-relative address, see *ADD (SP plus immediate)* on page A7-193
1011xx	*Miscellaneous 16-bit instructions* on page A5-132
11000x	Store multiple registers, see *STM, STMIA, STMEA* on page A7-422
11001x	Load multiple registers, see *LDM, LDMIA, LDMFD* on page A7-248
1101xx	*Conditional branch, and supervisor call* on page A5-134
11100x	Unconditional Branch, see *B* on page A7-207

Encoding 16-bit Thumb Instructions: Examples

- Load/store Single Data Item



These instructions have one of the following values in opA:

- `0b0101`.
- `0b011x`.
- `0b100x`.

opcode	Instruction or instruction class
0101xx 011xxx 100xxx	*Load/store single data item* on page A5-131

opA	opB	Instruction	See
0101	000	Store Register	*STR (register)* on page A7-428
0101	001	Store Register Halfword	*STRH (register)* on page A7-444
0101	010	Store Register Byte	*STRB (register)* on page A7-432
0101	011	Load Register Signed Byte	*LDRSB (register)* on page A7-286
0101	100	Load Register	*LDR (register)* on page A7-256
0101	101	Load Register Halfword	*LDRH (register)* on page A7-278
0101	110	Load Register Byte	*LDRB (register)* on page A7-262
0101	111	Load Register Signed Halfword	*LDRSH (register)* on page A7-294
0110	0xx	Store Register	*STR (immediate)* on page A7-426
0110	1xx	Load Register	*LDR (immediate)* on page A7-252
0111	0xx	Store Register Byte	*STRB (immediate)* on page A7-430
0111	1xx	Load Register Byte	*LDRB (immediate)* on page A7-258
1000	0xx	Store Register Halfword	*STRH (immediate)* on page A7-442
1000	1xx	Load Register Halfword	*LDRH (immediate)* on page A7-274
1001	0xx	Store Register SP relative	*STR (immediate)* on page A7-426
1001	1xx	Load Register SP relative	*LDR (immediate)* on page A7-252

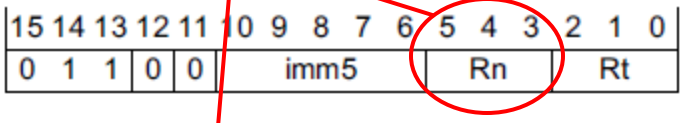
Encoding 16-bit Thumb Instructions: Store

STR (immediate)

Store Register (immediate) calculates an address from a base register value and an immediate offset, and stores a word from a register to memory. It can use offset, post-indexed, or pre-indexed addressing. See *Memory accesses* on page A7-182 for information about memory accesses.

Encoding T1 All versions of the Thumb instruction set.

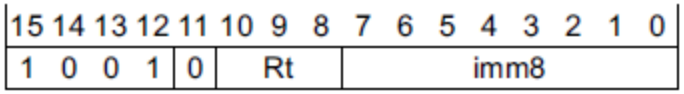
```
STR<c> <Rt>, [<Rn>{,#<imm5>}]
```




```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

Encoding T2 All versions of the Thumb instruction set.

```
STR<c> <Rt>,[SP,#<imm8>]
```




```
t = UInt(Rt);  n = 13;  imm32 = ZeroExtend(imm8:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

Encoding 16-bit Thumb Instructions: Examples

Assembler syntax

`STR<c><q> <Rt>, [<Rn> {, #+/-<imm>}]`	Offset: `index==TRUE, wback==FALSE`
`STR<c><q> <Rt>, [<Rn>, #+/-<imm>]!`	Pre-indexed: `index==TRUE, wback==TRUE`
`STR<c><q> <Rt>, [<Rn>], #+/-<imm>`	Post-indexed: `index==FALSE, wback==TRUE`

where:

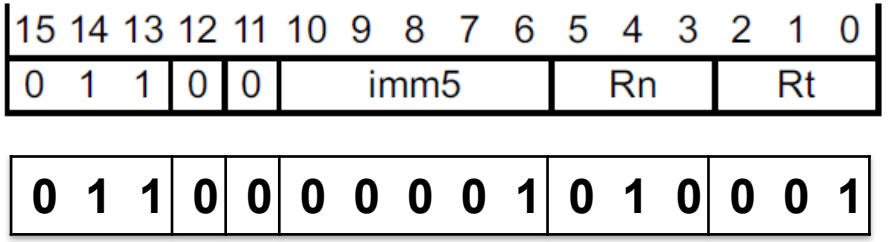
`<c><q>`	See *Standard assembler syntax fields* on page A7-175.
`<Rt>`	Specifies the source register. This register is permitted to be the SP.
`<Rn>`	Specifies the base register. This register is permitted to be the SP.
`+/-`	Is + or omitted to indicate that the immediate offset is added to the base register value (`add == TRUE`), or – to indicate that the offset is to be subtracted (`add == FALSE`). Different instructions are generated for `#0` and `#-0`.
`<imm>`	Specifies the immediate offset added to or subtracted from the value of `<Rn>` to form the address. Permitted values are multiples of 4 in the range 0-124 for encoding T1, multiples of 4 in the range 0-1020 for encoding T2, any value in the range 0-4095 for encoding T3, and any value in the range 0-255 for encoding T4. For the offset addressing syntax, `<imm>` can be omitted, meaning an offset of 0.

Encoding 16-bit Thumb Instructions: Store

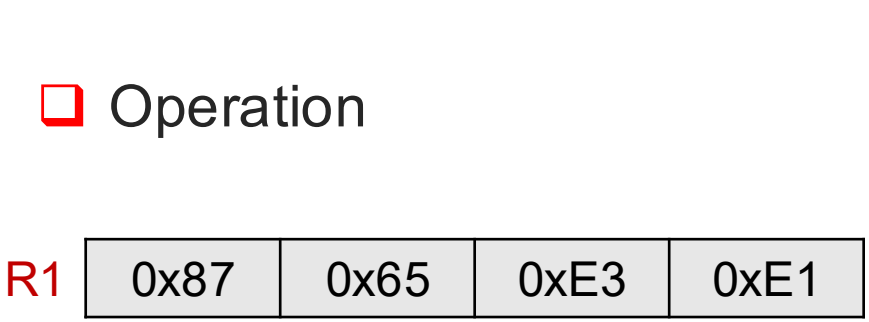
- Example

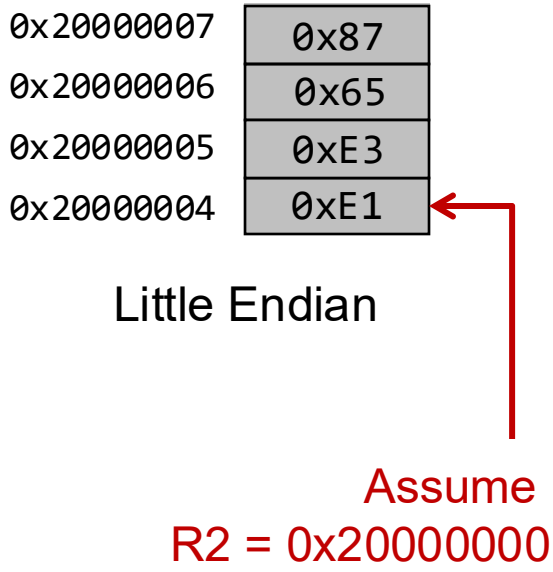
```
STR<c> <Rt>, [<Rn>{,#<imm5>}]

STR R1, [R2,#0x4]
```



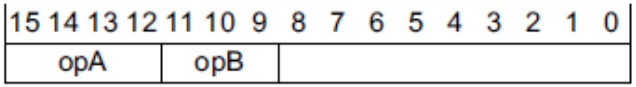
- Operation





Encoding 16-bit Thumb Instructions: Examples

- Load/store single data item



These instructions have one of the following values in opA:

- `0b0101`.
- `0b011x`.
- `0b100x`.

opcode	Instruction or instruction class
0101xx 011xxx 100xxx	*Load/store single data item* on page A5-131

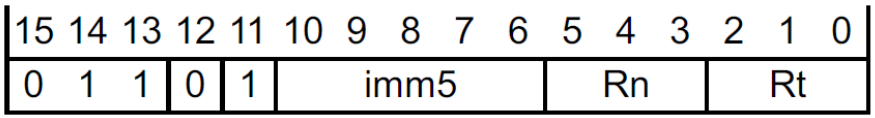
opA	opB	Instruction	See
0101	000	Store Register	*STR (register)* on page A7-428
0101	001	Store Register Halfword	*STRH (register)* on page A7-444
0101	010	Store Register Byte	*STRB (register)* on page A7-432
0101	011	Load Register Signed Byte	*LDRSB (register)* on page A7-286
0101	100	Load Register	*LDR (register)* on page A7-256
0101	101	Load Register Halfword	*LDRH (register)* on page A7-278
0101	110	Load Register Byte	*LDRB (register)* on page A7-262
0101	111	Load Register Signed Halfword	*LDRSH (register)* on page A7-294
0110	0xx	Store Register	*STR (immediate)* on page A7-426
0110	1xx	Load Register	*LDR (immediate)* on page A7-252
0111	0xx	Store Register Byte	*STRB (immediate)* on page A7-430
0111	1xx	Load Register Byte	*LDRB (immediate)* on page A7-258
1000	0xx	Store Register Halfword	*STRH (immediate)* on page A7-442
1000	1xx	Load Register Halfword	*LDRH (immediate)* on page A7-274
1001	0xx	Store Register SP relative	*STR (immediate)* on page A7-426
1001	1xx	Load Register SP relative	*LDR (immediate)* on page A7-252

Encoding 16-bit Thumb Instructions: Load

- LDR (immediate)
  - Load a word to Rt
  - Address in Rn
  - Offset imm5:00

Encoding T1 All versions of the Thumb instruction set.

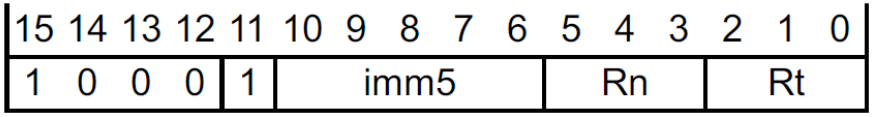
`LDR<c> <Rt>, [<Rn>{,#<imm5>}]`



- LDRH (immediate)
  - Load a Halfword to Rt
  - Address in Rn
  - Offset imm5:0

Encoding T1 All versions of the Thumb instruction set.

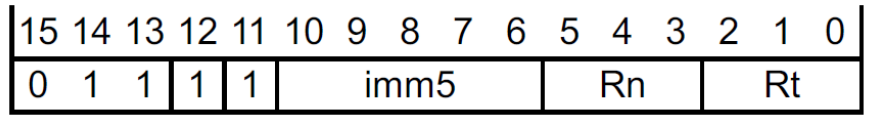
`LDRH<c> <Rt>,[<Rn>{,#<imm5>}]`



- LDRB (immediate)
  - Load a Byte to Rt
  - Address in Rn
  - Offset imm5

Encoding T1 All versions of the Thumb instruction set.

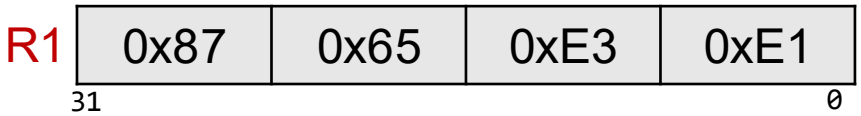
`LDRB<c> <Rt>,[<Rn>{,#<imm5>}]`



Encoding 16-bit Thumb Instructions: Load

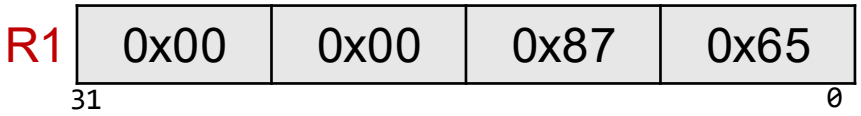
- Load a Word

```
LDR R1, [R2,#0x4]
```



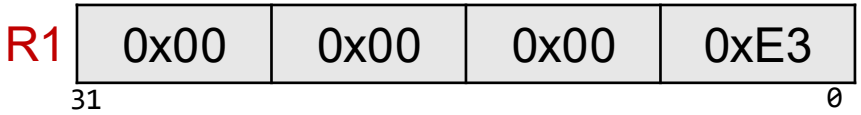
- Load a Halfword

```
LDRH R1, [R2,#0x6]
```

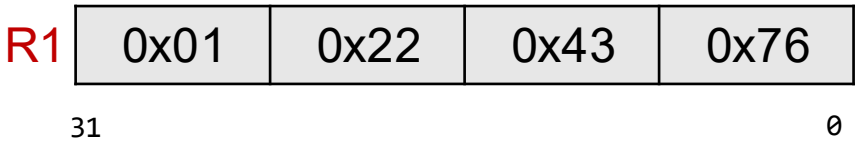


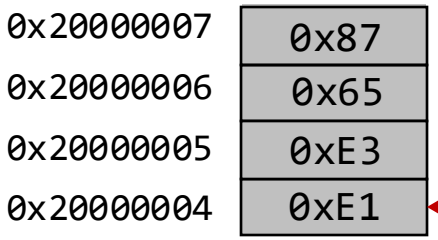
- Load a Byte

```
LDRB R1, [R2,#0x5]
```



R1 before load



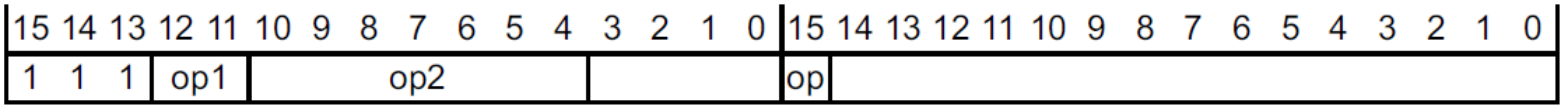


Little Endian

Assume
R2 = 0x20000000

Encoding 32-bit Thumb Instructions

- General (A5.3)



op1 != 0b00. If op1 == 0b00, a 16-bit instruction is encoded.

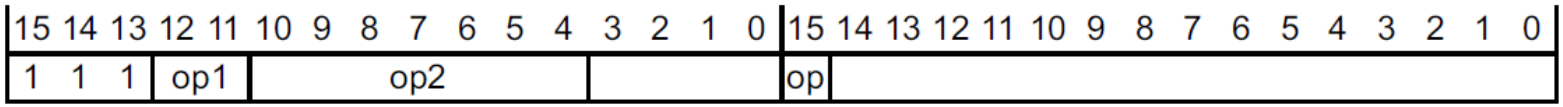
Encoding 32-bit Thumb Instructions

- 32-bit Thumb

op1	op2	op	Instruction class
01	00xx0xx	x	*Load Multiple and Store Multiple* on page A5-142
01	00xx1xx	x	*Load/store dual or exclusive, table branch* on page A5-143
01	01xxxxx	x	*Data processing (shifted register)* on page A5-148
01	1xxxxxx	x	*Coprocessor instructions* on page A5-156
10	x0xxxxx	0	*Data processing (modified immediate)* on page A5-136
10	x1xxxxx	0	*Data processing (plain binary immediate)* on page A5-139
10	xxxxxxx	1	*Branches and miscellaneous control* on page A5-140
11	000xxx0	x	*Store single data item* on page A5-147
11	00xx001	x	*Load byte, memory hints* on page A5-146
11	00xx011	x	*Load halfword, memory hints* on page A5-145
11	00xx101	x	*Load word* on page A5-144
11	00xx111	x	UNDEFINED
11	010xxxx	x	*Data processing (register)* on page A5-150
11	0110xxx	x	*Multiply, multiply accumulate, and absolute difference* on page A5-154
11	0111xxx	x	*Long multiply, long multiply accumulate, and divide* on page A5-154
11	1xxxxxx	x	*Coprocessor instructions* on page A5-156

Encoding 32-bit Thumb Instructions: Examples

- Data Processing (A.5.3.11)



op	Rn	Rd	S	Instruction	See	Variant
0000	-	not 1111	x	Bitwise AND	*AND (register)* on page A7-201	All
		1111	0	UNPREDICTABLE	-	-
			1	Test	*TST (register)* on page A7-466	All
0001	-	-	-	Bitwise Bit Clear	*BIC (register)* on page A7-213	All
0010	not 1111	-	-	Bitwise OR	*ORR (register)* on page A7-336	All
	1111	-	-	-	*Move register and immediate shifts*	-
0011	not 1111	-	-	Bitwise OR NOT	*ORN (register)* on page A7-333	All
	1111	-	-	Bitwise NOT	*MVN (register)* on page A7-328	All
0100	-	not 1111	-	Bitwise Exclusive OR	*EOR (register)* on page A7-239	All
		1111	0	UNPREDICTABLE	-	-
			1	Test Equivalence	*TEQ (register)* on page A7-464	All
0110	-	-	-	Pack Halfword	*PKHBT, PKHTB* on page A7-338	v7E-M

Encoding 32-bit Thumb Instructions: AND (A.7.7.9)

AND (register)

AND (register) performs a bitwise AND of a register value and an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 All versions of the Thumb instruction set.

`ANDS <Rdn>,<Rm>` Outside IT block.

`AND<c> <Rdn>,<Rm>` Inside IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	0	0	Rm			Rdn		

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 ARMv7-M

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

1. source register

target register

2. source

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

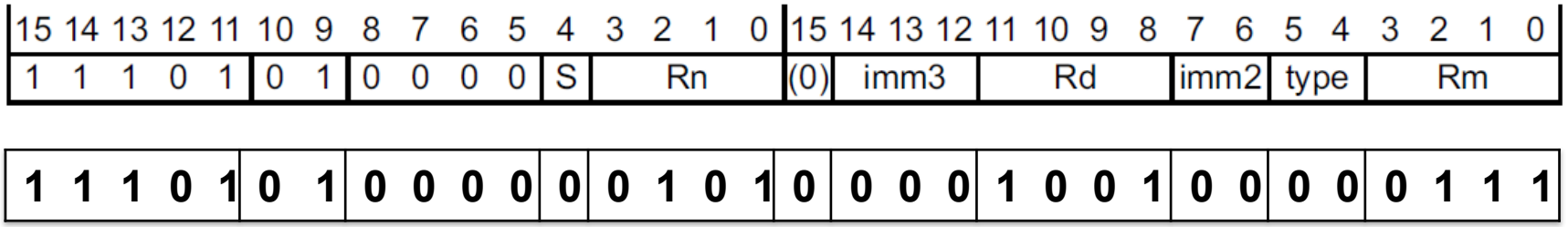
```
if Rd == '1111' && S == '1' then SEE TST (register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d == 13 || (d == 15 && S == '0') || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Encoding for 32 bit instruction (see later)

Encoding 32-bit Thumb Instructions: AND

- Example `AND{S}<c>.W <Rd>,<Rn>,<Rm>{,<shift>}`

```
AND.W  R9, R5, R7
```



- Computation

```
R5 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
R7 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 1
-------------------------------------------------------------------
R9 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
```

Load/store Double Registers

- Store Rt to address in Rn
- Store Rt2 to a word 4 bytes above the address in Rn

Encoding T1 ARMv7-M

```
STRD<c> <Rt>,<Rt2>,[<Rn>{,#+/-<imm8>}]
STRD<c> <Rt>,<Rt2>,[<Rn>],#+/-<imm8>
STRD<c> <Rt>,<Rt2>,[<Rn>,#+/-<imm8>]!
```

- Example

```
STRD R1,R2, [R0]
; R0 = 0x20008000
; R1 = 0x76543210
; R2 = 0xABCDEF10
```

Memory Address	Memory Data
0x20008007	0xAB
0x20008006	0xCD
0x20008005	0xEF
0x20008004	0x10
0x20008003	0x76
0x20008002	0x54
0x20008001	0x32
0x20008000	0x10

Load/Store Multiple

- General (Pseudo)-Instructions

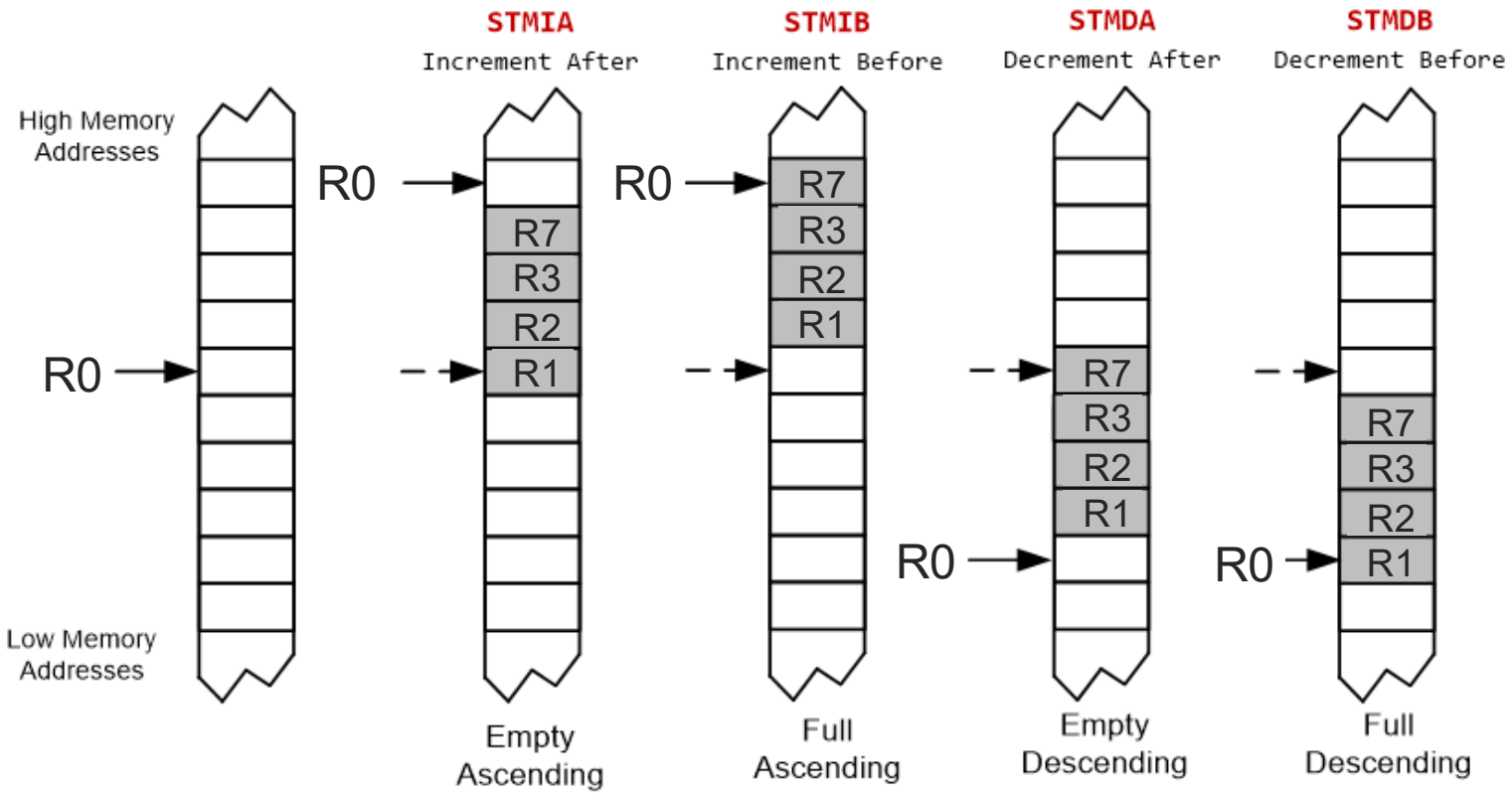
```
STMxx <Rn>!,<registers>    STMxx.W <Rn>{!},<registers>
LDMxx <Rn>!,<registers>    LDMxx.W <Rn>{!},<registers>
```

- Explanation: xx = IA, IB, DA, DB

Addressing Modes	Description	Instructions
IA	Address is incremented by 4 after a word is loaded or stored	STMIA, LDMIA
IB	Address is incremented by 4 before a word is loaded or stored	STMIB, LDMIB
DA	Address is decremented by 4 after a word is loaded or stored	STMDA, LDMDA
DB	Address is decremented by 4 before a word is loaded or stored	STMDB, LDMDB

Store Multiple Registers: Example

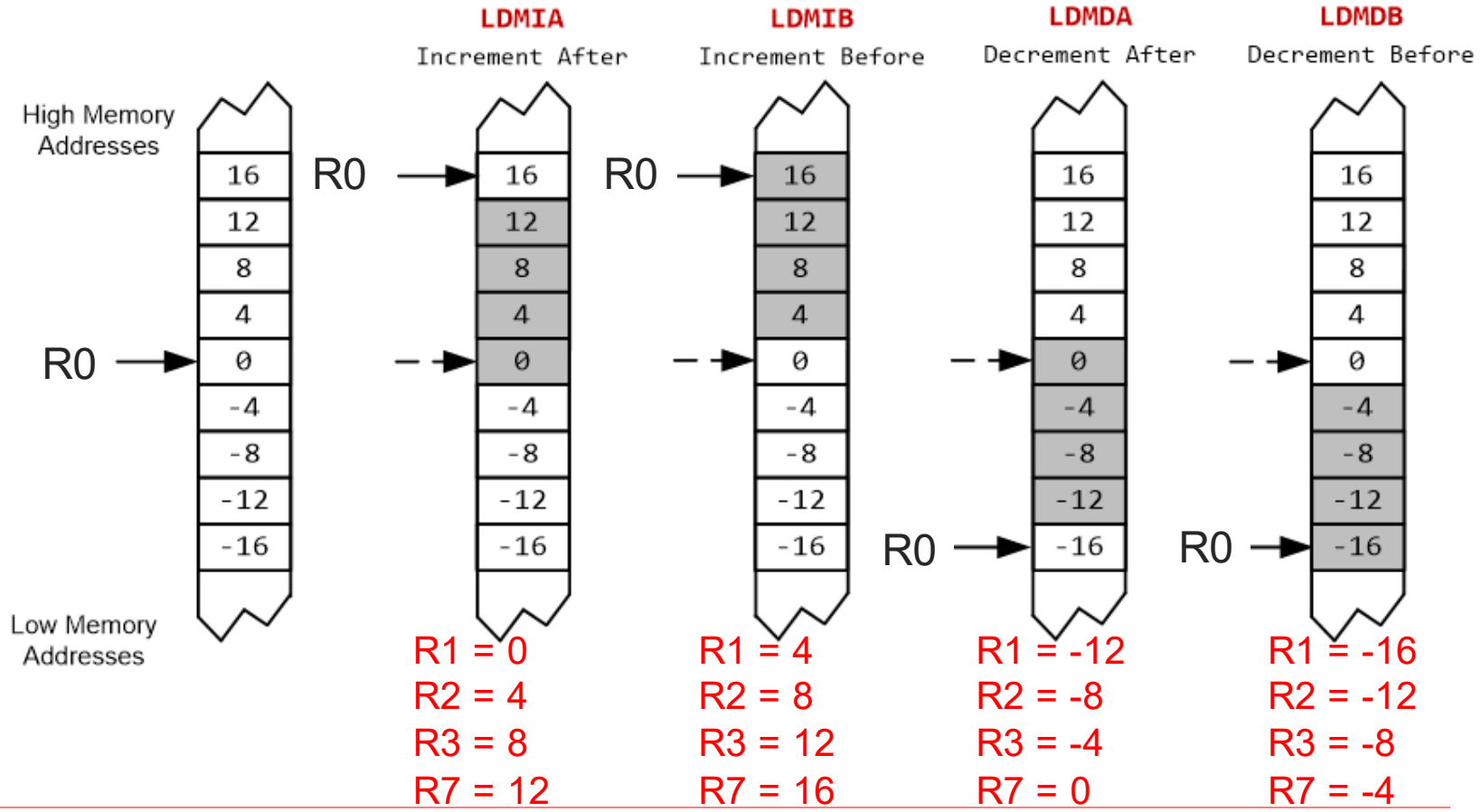
```
STMxx R0!, {R3, R1, R7, R2}
```



Note: lowest register is stored at lowest memory address

Load Multiple Registers: Example

```
LDMxx R0!, {R3, R1, R7, R2}
```



PC-relative Addressing Mode: ADR

- Properties
  - Indexed addressing using PC as the pointer
  - Represented in the instruction as the PC value plus/minus a numeric offset
  - The assembler calculates the required offset from the label and the address of the current instruction
  - If the offset is too big, the assembler produces an error
- Usage of this addressing mode
  - Branching, calling functions, etc.
- Example

```
0x00001000 ADR R5,label  ;R5=PC+$offset where offset
                         ;is from PC to label=$0C
                .
                .
0x0000100C      label LDR   r0,[r1]
```

Arithmetic and Logic Instructions

- Shift
  - LSL (logic shift left), LSR (logic shift right), ASR (arithmetic shift right), ROR (rotate right), RRX (rotate right with extend)
- Logic
  - AND (bitwise and), ORR (bitwise or), EOR (bitwise exclusive or), ORN (bitwise or not), MVN (move not)
- Bit set/clear
  - BFC (bit field clear), BFI (bit field insert), BIC (bit clear), CLZ (count leading zeroes)
- Bit/byte reordering
  - RBIT (reverse bit order in a word), REV (reverse byte order in a word), REV16 (reverse byte order in each half-word independently), REVSH (reverse byte order in each half-word independently)
- Addition
  - ADD, ADC (add with carry)

Arithmetic and Logic Instructions

- Subtraction
  - SUB, RSB (reverse subtract), SBC (subtract with carry)
- Multiplication
  - MUL (multiply), MLA (multiply-accumulate), MLS (multiply-subtract), SMULL (signed long multiply-accumulate), SMLAL (signed long multiply-accumulate), UMULL (unsigned long multiply-subtract), UMLAL (unsigned long multiply-subtract)
- Division
  - SDIV (signed), UDIV (unsigned)
- Sign extension
  - SXTB (signed), SXTH, UXTB, UXTH
- Bit field extract
  - SBFX (signed), UBFX (unsigned)

Commonly Used Arithmetic Operations

ADD {Rd,} Rn, Op2	Add. Rd ← Rn + Op2
ADC {Rd,} Rn, Op2	Add with carry. Rd ← Rn + Op2 + Carry
SUB {Rd,} Rn, Op2	Subtract. Rd ← Rn - Op2
SBC {Rd,} Rn, Op2	Subtract with carry. Rd ← Rn - Op2 + Carry - 1
RSB {Rd,} Rn, Op2	Reverse subtract. Rd ← Op2 - Rn
MUL {Rd,} Rn, Rm	Multiply. Rd ← (Rn × Rm)[31:0]
MLA Rd, Rn, Rm, Ra	Multiply with accumulate. Rd ← (Ra + (Rn × Rm))[31:0]
MLS Rd, Rn, Rm, Ra	Multiply and subtract. Rd ← (Ra – (Rn × Rm))[31:0]
SDIV {Rd,} Rn, Rm	Signed divide. Rd ← Rn / Rm
UDIV {Rd,} Rn, Rm	Unsigned divide. Rd ← Rn / Rm

Example: ADD and SUBS

- Basic Format

```
ADD R1, R2, #4     ; R1 = R2 + 4 (immediate)
ADD R1, R2, R3     ; R1 = R2 + R3 (register)
```

- Example: Assume R3=1000, R2=4000, R4=3000

```
ADD R3,#250     ; R3=1250, APSR flags don't change
ADDS R3,#250    ; R3=1250, N=Z=C=V=0
ADDS R1,R2,R4   ; results in R1=7000, N=Z=C=V=0
```

- Example: Assume R3=1000, R2=250, R4=3000

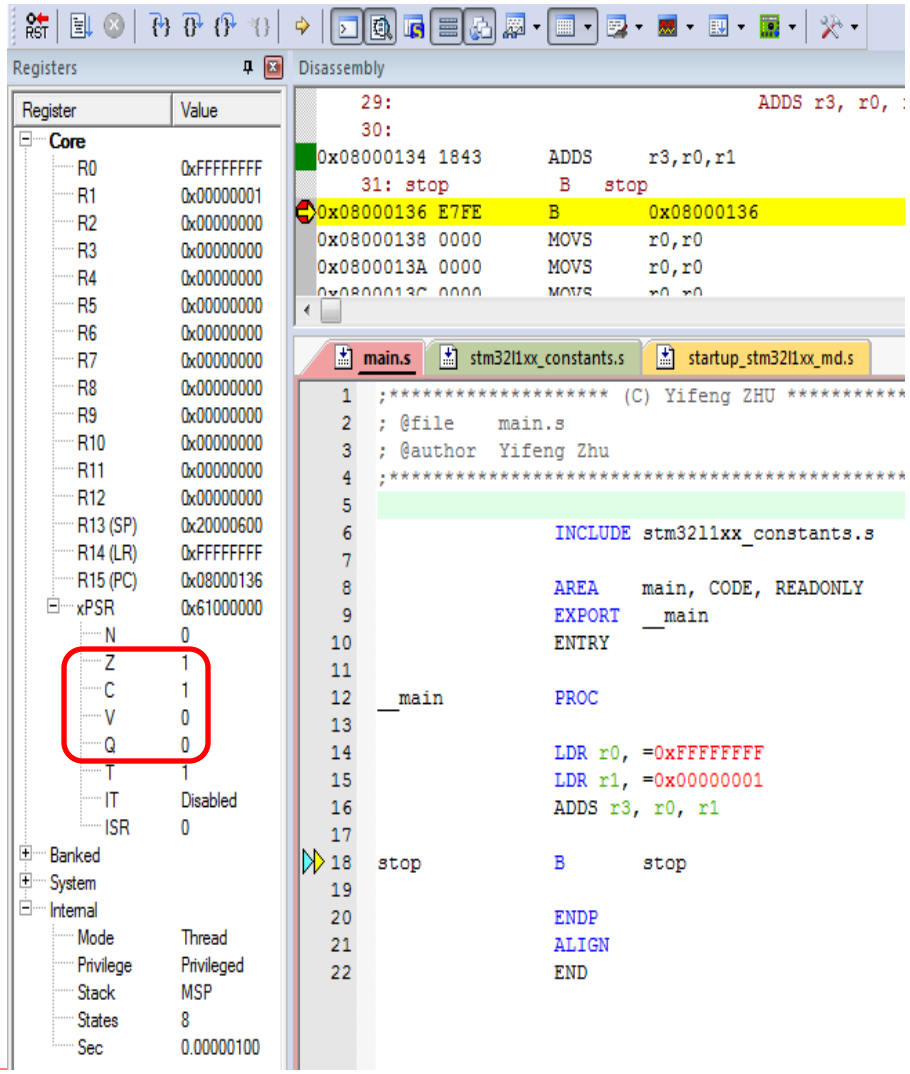
```
SUBS R3, R2      ; R3=750, N=Z=V=0, C=1
SUBS R1, R4, R2  ; R1=2750, N=Z=V=0, C=1
```

Example: Condition Flags

```
start
        LDR     r0, =0xFFFFFFFF
        LDR     r1, =0x00000001
        ADDS    r0, r0, r1
stop    B       stop
```

Set Condition Flags

→ In this example, the Z and C bits are set



Example: Short Multiplication

- Signed Multiplication: MUL

```
MUL   R6, R4, R2          ; R6 = LSB32( R4 × R2 )
```

- Unsigned Multiplication: UMUL

```
UMUL R6, R4, R2           ; R6 = LSB32( R4 × R2 )
```

- Multiplication with Accumulation: MLA

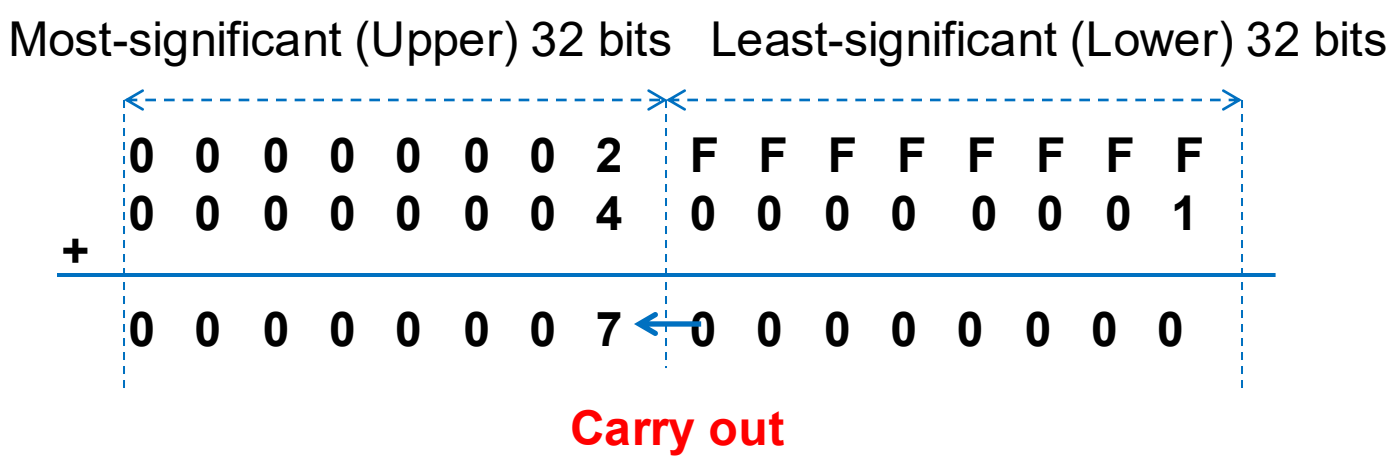
```
MLA   R6, R4, R1, R0      ; R6 = LSB32( R4 × R1 + R0)
```

- Multiplication with Subtract: MLS

```
MLS   R6, R4, R1, R0      ; R6 = LSB32( R0 - R4 × R1 )
```

Example: 64-bit Addition

- Problem
  - A register can only store 32 bits
  - A 64-bit integer needs two registers
  - Split 64-bit addition into two 32-bit additions
- Realization



Example: 64-bit Addition

```
start
  ; C = A + B
  ; Two 64-bit integers A (R1,R0) and B (R3, r2).
  ; Result C (R5, R4)
  ; A = 00000002FFFFFFFF
  ; B = 0000000400000001
  LDR  R0, =0xFFFFFFFF  ;  A's lower 32 bits
  LDR  R1, =0x00000002  ;  A's upper 32 bits
  LDR  R2, =0x00000001  ;  B's lower 32 bits
  LDR  R3, =0x00000004  ;  B's upper 32 bits

  ;  Add A and B
  ADDS R4, R2, R0  ; C[31..0] = A[31..0] +  B[31..0],  update Carry
  ADC  R5, R3, R1  ; C[64..32] = A[64..32] + B[64..32]  +  Carry

stop  B  stop
```

Example: 64-bit Subtraction

```
start
  ; C = A - B
  ; Two 64-bit integers A (R1, R0) and B (R3, R2).
  ; Result C (R5, R4)
  ; A = 00000002FFFFFFFF
  ; B = 0000000400000001
  LDR  R0, =0xFFFFFFFF  ;  A's lower 32 bits
  LDR  R1, =0x00000002  ;  A's upper 32 bits
  LDR  R2, =0x00000001  ;  B's lower 32 bits
  LDR  R3, =0x00000004  ;  B's upper 32 bits

  ;  Subtract B from A
  SUBS R4, R0, R2  ; C[31..0]= A[31..0] - B[31..0],  update Carry
  SBC  R5, R1, R3  ; C[64..32]= A[64..32] - B[64..32] -  Carry

stop  B  stop
```

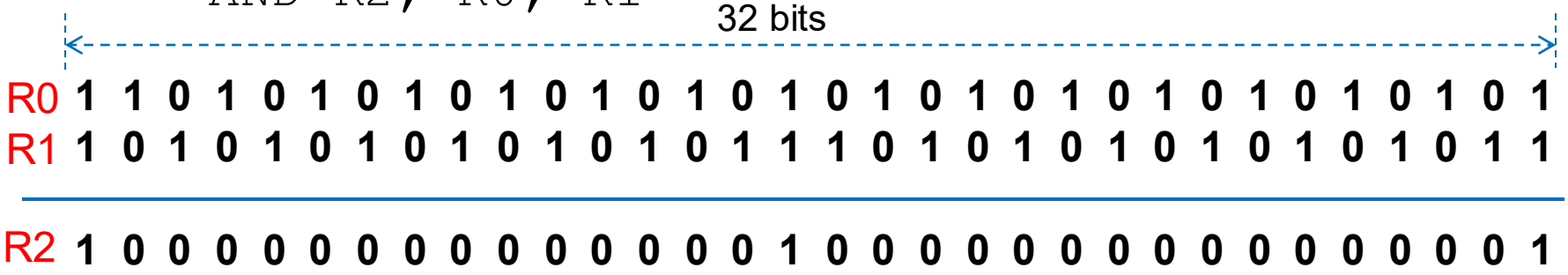
Bitwise Logic

AND {Rd,} Rn, Op2	Bitwise logic AND. Rd ← Rn & operand2
ORR {Rd,} Rn, Op2	Bitwise logic OR. Rd ← Rn | operand2
EOR {Rd,} Rn, Op2	Bitwise logic exclusive OR. Rd ← Rn ^ operand2
ORN {Rd,} Rn, Op2	Bitwise logic NOT OR. Rd ← Rn | (NOT operand2)
BIC {Rd,} Rn, Op2	Bit clear. Rd ← Rn & NOT operand2
BFC Rd, #lsb, #width	Bit field clear. Rd[(width+lsb–1):lsb] ← 0
BFI Rd, Rn, #lsb, #width	Bit field insert. Rd[(width+lsb–1):lsb] ← Rn[(width-1):0]
MVN Rd, Op2	Move NOT, logically negate all bits. Rd ← 0xFFFFFFFF EOR Op2

Examples: AND and OR

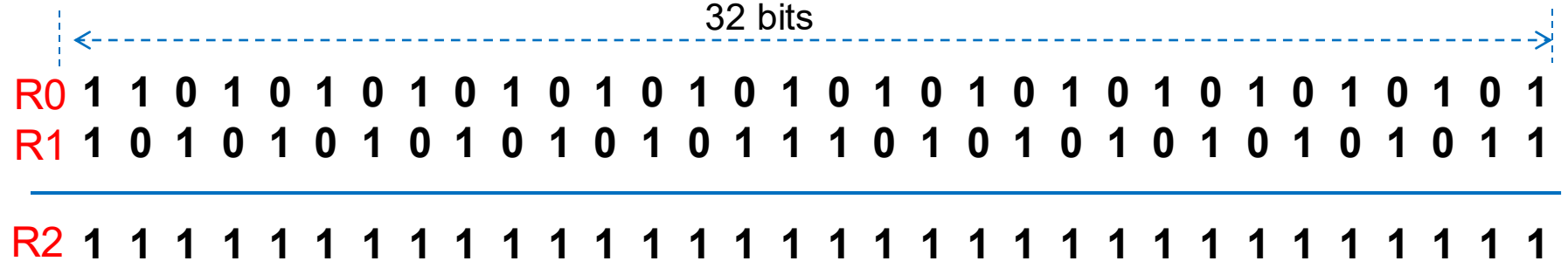
- Bitwise Logic AND

```
AND R2, R0, R1
```



- Bitwise Logic OR

```
ORR R2, R0, R1
```



Examples: Bit Clear

- Instruction

```
BIC R2, R0, R1 ; R2= R0 & NOT R1
```

Step 1: R1 = NOT R1

R1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
NOT R1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0

Step 2: R2 = R0 & NOT R1

R0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
NOT R1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
R2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0

Examples: Bit Field Clear and Bit Field Insert

- Instruction

```
BFC Rd, #lsb, #width
BFI Rd, Rn, #lsb, #width
```

- Examples

```
BFC R4, #8, #12
;Clear bit 8 to bit 19 (12 bits) of R4 to 0


BFI R9, R2, #8, #12
; Replace bit 8 to bit 19 (12 bits) of R9
; with bit 0 to bit 11 from R2
```

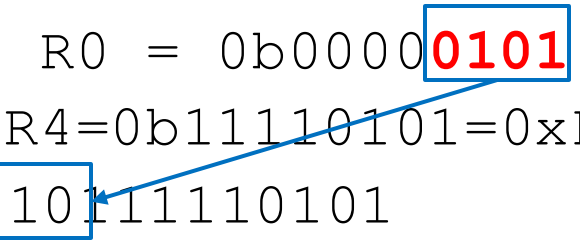
Examples: Different Instructions

- Assume

```
[R0] = 0x0F = 0b00001111 = 15
[R4] = 0xF0 = 0b11110000
[R1] = 0xAD = 0b10101101
```

- Code

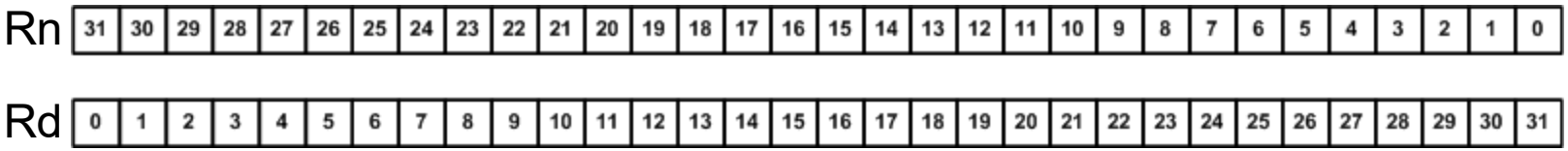
```
AND R0,R0,#5    ; perform AND; R0 = 0b00000101
ORR R4,R0,R4    ; perform OR; R4=0b11110101=0xF5
BFI R4,R0,#8,#4 ;  R4=0b000001011110101
BFC R4,#1,#5    ;  R4=0b000001011100001=0x05C1
ORN R4,R0,R1    ; NOT R1 = 0xFFFFFF52
                ;      R0 = 0x00000005
                ;      R4 = 0xFFFFFF57
```



Reverse Order

RBIT Rd, Rn	Reverse bit order in a word. for (i = 0; i < 32; i++) Rd[i] ← RN[31– i]
REV Rd, Rn	Reverse byte order in a word. Rd[31:24] ← Rn[7:0], Rd[23:16] ← Rn[15:8], Rd[15:8] ← Rn[23:16], Rd[7:0] ← Rn[31:24]
REV16 Rd, Rn	Reverse byte order in each half-word. Rd[15:8] ← Rn[7:0], Rd[7:0] ← Rn[15:8], Rd[31:24] ← Rn[23:16], Rd[23:16] ← Rn[31:24]
REVSH Rd, Rn	Reverse byte order in bottom half-word and sign extend. Rd[15:8] ← Rn[7:0], Rd[7:0] ← Rn[15:8], Rd[31:16] ← Rn[7] & 0xFFFF

RBIT Rd, Rn



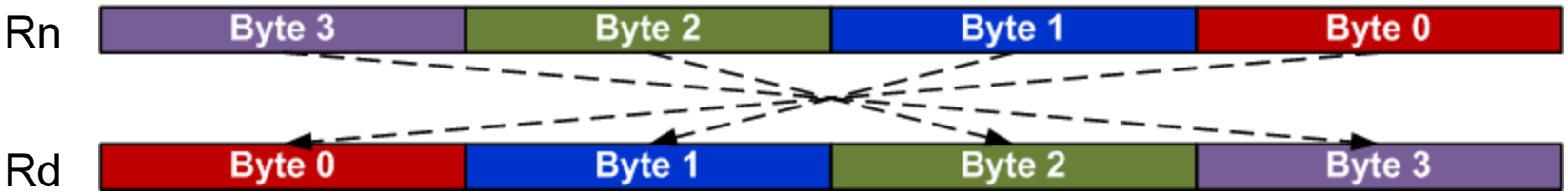
Example:

```
LDR  R0, =0x12345678  ; R0 = 0x12345678
RBIT R1, R0           ; Reverse bits, R1 = 0x1E6A2C48
```

Reverse Order

RBIT Rd, Rn	Reverse bit order in a word. for (i = 0; i < 32; i++) Rd[i] ← RN[31– i]
REV Rd, Rn	Reverse byte order in a word. Rd[31:24] ← Rn[7:0], Rd[23:16] ← Rn[15:8], Rd[15:8] ← Rn[23:16], Rd[7:0] ← Rn[31:24]
REV16 Rd, Rn	Reverse byte order in each half-word. Rd[15:8] ← Rn[7:0], Rd[7:0] ← Rn[15:8], Rd[31:24] ← Rn[23:16], Rd[23:16] ← Rn[31:24]
REVSH Rd, Rn	Reverse byte order in bottom half-word and sign extend. Rd[15:8] ← Rn[7:0], Rd[7:0] ← Rn[15:8], Rd[31:16] ← Rn[7] & 0xFFFF

REV Rd, Rn



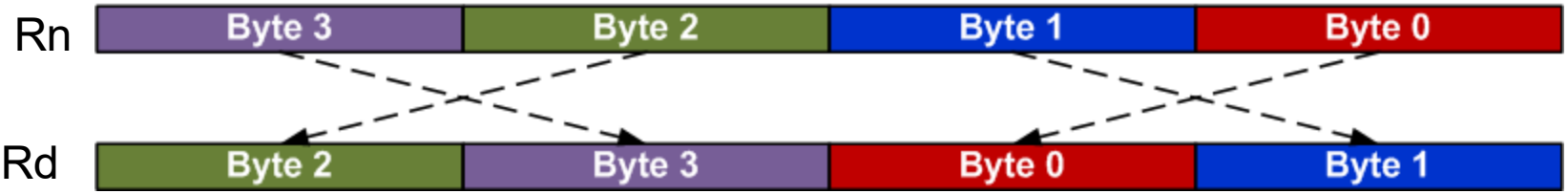
Example:

```
LDR R0, =0x12345678    ; R0 = 0x12345678
REV R1, R0             ; R1 = 0x78563412
```

Reverse Order

RBIT Rd, Rn	Reverse bit order in a word. for (i = 0; i < 32; i++) Rd[i] ← RN[31– i]
REV Rd, Rn	Reverse byte order in a word. Rd[31:24] ← Rn[7:0], Rd[23:16] ← Rn[15:8], Rd[15:8] ← Rn[23:16], Rd[7:0] ← Rn[31:24]
REV16 Rd, Rn	Reverse byte order in each half-word. Rd[15:8] ← Rn[7:0], Rd[7:0] ← Rn[15:8], Rd[31:24] ← Rn[23:16], Rd[23:16] ← Rn[31:24]
REVSH Rd, Rn	Reverse byte order in bottom half-word and sign extend. Rd[15:8] ← Rn[7:0], Rd[7:0] ← Rn[15:8], Rd[31:16] ← Rn[7] & 0xFFFF

REV16 Rd, Rn



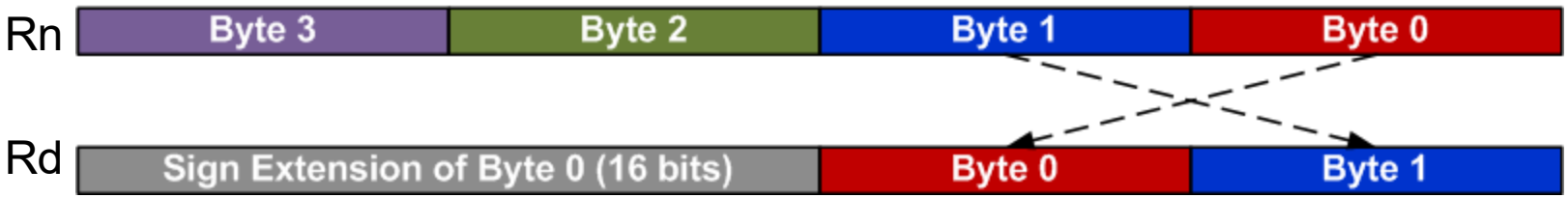
Example:

```
LDR R0, =0x12345678    ; R0 = 0x12345678
REV16 R2, R0           ; R2 = 0x34127856
```

Reverse Order

RBIT Rd, Rn	Reverse bit order in a word. for (i = 0; i < 32; i++) Rd[i] ← RN[31– i]
REV Rd, Rn	Reverse byte order in a word. Rd[31:24] ← Rn[7:0], Rd[23:16] ← Rn[15:8], Rd[15:8] ← Rn[23:16], Rd[7:0] ← Rn[31:24]
REV16 Rd, Rn	Reverse byte order in each half-word. Rd[15:8] ← Rn[7:0], Rd[7:0] ← Rn[15:8], Rd[31:24] ← Rn[23:16], Rd[23:16] ← Rn[31:24]
REVSH Rd, Rn	Reverse byte order in bottom half-word and sign extend. Rd[15:8] ← Rn[7:0], Rd[7:0] ← Rn[15:8], Rd[31:16] ← Rn[7] & 0xFFFF

REVSH Rd, Rn



Example:

```
LDR R0, =0x33448899   ; R0 = 0x33448899
REVSH R1, R0          ; R1 = 0xFFFF9988
```

Sign and Zero Extension

- Motivation

```
signed int_8  a = -5;      // a signed 8-bit integer,  a = 0xFB
signed int_16 b = -120;    // a signed 16-bit integer, b = 0xFF88
signed int_32 c;           // a signed 32-bit integer

c = a;                   // sign extension required, c = 0xFFFFFFFB
c = b;                   // sign extension required, c = 0xFFFFFF88
```

- 2s Complement Computation

```
5   → 00000101          120   → 0000000001111000
-5  → 11111011          -120  → 1111111110001000
```

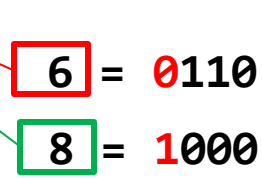
Sign and Zero Extension

- Instructions

SXTB {Rd,} Rm {,ROR #n}	Sign extend a byte. Rd[31:0] ← Sign Extend((Rm ROR (8 × n))[7:0])
SXTH {Rd,} Rm {,ROR #n}	Sign extend a half-word. Rd[31:0] ← Sign Extend((Rm ROR (8 × n))[15:0])
UXTB {Rd,} Rm {,ROR #n}	Zero extend a byte. Rd[31:0] ← Zero Extend((Rm ROR (8 × n))[7:0])
UXTH {Rd,} Rm {,ROR #n}	Zero extend a half-word. Rd[31:0] ← Zero Extend((Rm ROR (8 × n))[15:0])

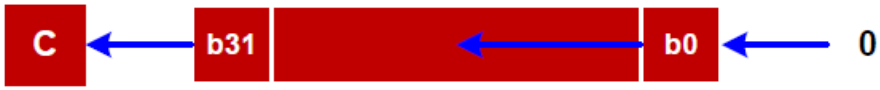
- Example

```
LDR R0, =0x55AA8765    ; R0 = 0x55AA8765
SXTB R1, R0            ; R1 = 0x00000065
SXTH R1, R0            ; R1 = 0xFFFF8765
UXTB R1, R0            ; R1 = 0x00000065
UXTH R1, R0            ; R1 = 0x00008765
```



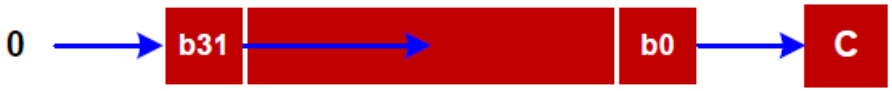
Shift and Rotate Instructions: Overview

- Logical Shift Left (LSL)



- Arithmetic Shift Right (ASR)

- Logical Shift Right (LSR)



- Rotate Right (ROR)

- Rotate Right Extended (RRX)

Question: Why is there rotate right but no rotate left?

Rotate left can be replaced by a rotate right with a different rotate offset

Shift and Rotate Instructions: Example

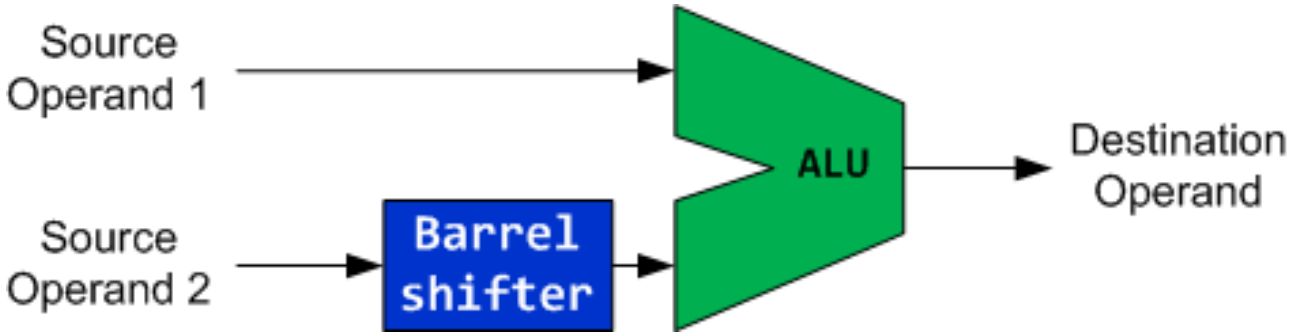
```
LSL R1, R0, #3  ; Before R0=0b00001111=0x0F=15
                ; After  R1=0b01111000=0x78=120=15*2^3

LSR R3, R4, #2  ; Before R4=0b11110000=0xF0=240
                ; After  R3=0b00111100=0x3C=60=240/2^2

ASR R3, R4, #2  ; Before R4=0b11110000=0xF0=240
                ; After  R3=0b00111100=0x3C=60=240/2^2
```

Using Barrel Shifter in Arithmetic

- Barrel Shifter
  - Special hardware at the second operand of ALU
  - Shift a data word by a specified number of bits
  - Does not use any sequential logic, it is a pure combinational logic circuit



- Example

```
ADD R1,R0,R0, LSL #3  ; R1 = R0+R0<<3 = 9 × R0
```

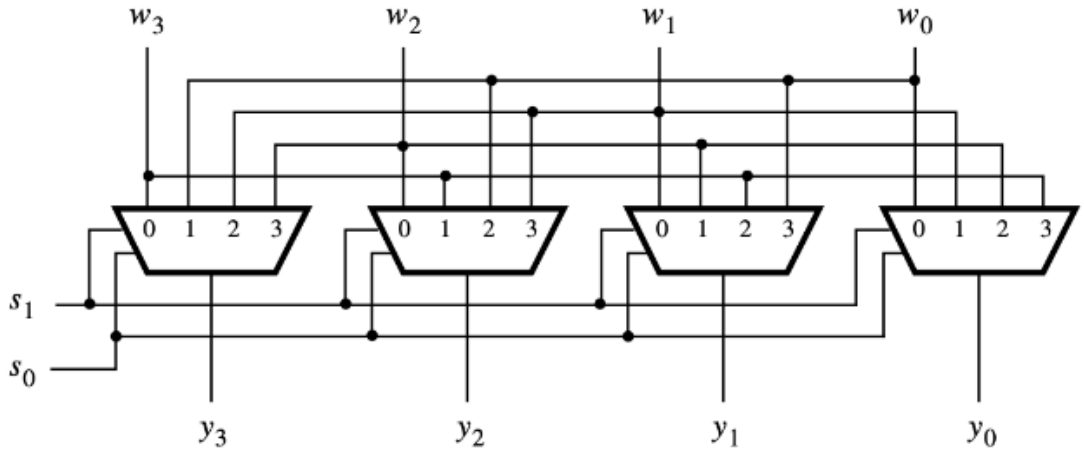
Shift left

Using Barrel Shifter in Arithmetic

- Barrel Shifter Operation

s_1	s_0	y_3	y_2	y_1	y_0
0	0	w_3	w_2	w_1	w_0
0	1	w_0	w_3	w_2	w_1
1	0	w_1	w_0	w_3	w_2
1	1	w_2	w_1	w_0	w_3

(a) Truth table



(b) Circuit

Using Barrel Shifter in Arithmetic: Examples

- Different Shift Operations

```
ADD R1, R0, R0, LSL #3
  ; R1 = R0 + R0 << 3 = R0 + 8 × R0
ADD R1, R0, R0, LSR #3
  ; R1 = R0 + R0 >> 3 = r0 + r0/8 (unsigned)
ADD R1, R0, R0, ASR #3
  ; R1 = R0 + R0 >> 3 = r0 + r0/8 (signed)
```

- Use Barrel shifter to speed up the application

```
ADD R1, R0, R0, LSL #3
          <=> MOV R2, #9        ; R2 = 9
              MUL R1, R0, R2 ; R1 = r0 * 9
```

Comparison Instructions

- Only Effect of Comparisons
  - Update the condition flags
  - No need to set S bit and no need to specify Rd (destination register)
- Operations
  - `CMP` operand1 - operand2, but result not written
  - `CMN` operand1 + operand2, but result not written
  - `TST` operand1 & operand2 (bitwise AND), but result not written
  - `TEQ` operand1 ^ operand2 (bitwise XOR), but result not written
- Examples
  - `CMP R0, R1`
  - `TST R2, #5`

Instruction	Operands	Brief description	Flags
CMP	Rn, Op2	Compare	N,Z,C,V
CMN	Rn, Op2	Compare Negative	N,Z,C,V
TEQ	Rn, Op2	Test Equivalence	N,Z,C
TST	Rn, Op2	Test	N,Z,C

CMP and CMN: Explanation

- **CMP{cond} Rn, Operand2**
  - Subtracts the value of Operand2 from the value in Rn
  - Same as a SUBS instruction, except that the result is discarded
- **CMN{cond} Rn, Operand2**
  - Adds the value of Operand2 to the value in Rn
  - Same as an ADDS instruction, except that the result is discarded
- Both instructions update the N, Z, C, V flags according to the result

TST and TEQ: Explanation

- **TST{cond} Rn, Operand2 ; Bitwise AND**
  - Performs a bitwise AND on the value in Rn and the value of Operand2
  - Same as a ANDS instruction, except that the result is discarded
- **TEQ{cond} Rn, Operand2 ; Bitwise Exclusive OR**
  - Performs a bitwise XOR on the value in Rn and the value of Operand2
  - Same as a EORS instruction, except that the result is discarded
- Both instructions update the N, Z, flags according to the result
- Both instructions can update the C flag during the calculation of Operand2 (see ARM®v7-M Architecture Reference Manual)
- Both instructions do not affect the V flag

Overflow: Reminder

- Compute 0xF2 – 0x74 for an 8 bit Register

```
  11110010          11110010
- 01110100        + 10001100
----------        ----------
                    01111110
```

Signed Greater or Equal (N == V)

```
CMP R0,R1; subtraction R0-R1, without saving the result
```

	N = 0	N = 1
V = 0	• No overflow, the result is correct. • The result is non-negative, • Thus r0 – r1 ≥ 0, i.e., r0 ≥ r1	• No overflow, the result is correct. • The result is negative. • Thus r0 – r1 < 0, i.e., r0 < r1
V = 1	• Overflow occurs, the result is incorrect. • The result is mistakenly reported as non-negative but it should be negative. • Thus r0 – r1 < 0 in reality, i.e., r0 < r1	• Overflow occurs, the result is incorrect. • The result is mistakenly reported as negative but it should be non-negative. • Thus r0 – r1 ≥ 0 in reality., i.e. r0 ≥ r1

- Conclusions
  - If N == V, then it is signed greater or equal (GE)
  - Oherwise, it is signed less than (LT)

Number Interpretation

- Example Comparison

  0xFFFFFFFF versus **0x00000001**

  Which is greater?

- Assume the numbers are signed numbers

  → 0xFFFFFFFF = #-1 < 0x00000001 = #1

- Assume the numbers are unsigned numbers

  → 0xFFFFFFFF = #4294967295 > 0x00000001 = #1

Number Interpretation: Software Reasonability

- Tell the Computer how to Interpret Data
  - If written in C, declare the signed vs unsigned variable
  - If written in Assembly, use signed vs unsigned branch instructions

```
signed int x, y ;
x = -1;
y = 1;
if (x > y)
   ...
```

```
    MOVS r6, #0xFFFFFFFF
    MOVS r5, #0x00000001
    CMP  r5, r6
    BLE  Then_Clause
    ...
```

BLE: Branch if less than or equal, signed ≤

```
unsigned int x, y ;
x = 4294967295;
y = 1;
if (x > y)
   ...
```

```
    MOVS r6, #0xFFFFFFFF
    MOVS r5, #0x00000001
    CMP  r5, r6
    BLS  Then_Clause
    ...
```

BLS: Branch if lower or same, unsigned ≤

Branch Instructions

- General Information
  - Instruction that causes the processor to branch to another instruction
  - Branch instructions change the sequence of instruction execution
- Overview

Instruction	Operands	Brief description	Flags
B	label	Cause a branch to label	-
BL	label	Copy the address of the next instruction into R14 (LR, the link register), and causes a branch to label	-
BX	Rm	Branch to the address held in Rm	-
BLX	Rm	Copy the address of the next instruction into R14 and branch to the address held in Rm	-

Branch Instructions: Branch With Link (BL)

- Usage of BL Instruction for Implementing a Subroutine
  - Write PC+4 into the LR
    - → This is the address of the next instruction following the branch with link
  - Return from subroutine by restoring the PC from the LR
    - `MOV PC, LR`
    - Again, pipeline has to refill before execution continues.
- Additional Facts
  - Branch instruction takes 3 cycles because of refilling the pipeline
  - Similar on return from branch instruction
  - The "Branch" instruction itself does not affect LR

Branch Instructions: Conditional

	Instruction	Description	Flags tested
Unconditional Branch	B label	Branch to label	
Conditional Branch	BEQ label	Branch if EQual	Z = 1
	BNE label	Branch if Not Equal	Z = 0
	BCS/BHS label	Branch if unsigned Higher or Same	C = 1
	BCC/BLO label	Branch if unsigned LOwer	C = 0
	BMI label	Branch if MInus (Negative)	N = 1
	BPL label	Branch if PLus (Positive or Zero)	N = 0
	BVS label	Branch if oVerflow Set	V = 1
	BVC label	Branch if oVerflow Clear	V = 0
	BHI label	Branch if unsigned HIgher	C = 1 & Z = 0
	BLS label	Branch if unsigned Lower or Same	C = 0 or Z = 1
	BGE label	Branch if signed Greater or Equal	N = V
	BLT label	Branch if signed Less Than	N != V
	BGT label	Branch if signed Greater Than	Z = 0 & N = V
	BLE label	Branch if signed Less than or Equal	Z = 1 or N = !V

Condition Codes

Suffix	Description	Flags tested
EQ	EQual	Z=1
NE	Not Equal	Z=0
CS/HS	Unsigned Higher or Same	C=1
CC/LO	Unsigned LOwer	C=0
MI	MInus (Negative)	N=1
PL	PLus (Positive or Zero)	N=0
VS	oVerflow Set	V=1
VC	oVerflow Clear	V=0
HI	Unsigned HIgher	C=1 & Z=0
LS	Unsigned Lower or Same	C=0 or Z=1
GE	Signed Greater or Equal	N=V
LT	Signed Less Than	N!=V
GT	Signed Greater Than	Z=0 & N=V
LE	Signed Less than or Equal	Z=1 or N!=V
AL	ALways	

Conditional Branch Instructions

- Conditional Codes Applied to Branch Instructions

Compare	Signed	Unsigned
==	EQ	EQ
≠	NE	NE
>	GT	HI
≥	GE	HS
<	LT	LO
≤	LE	LS

Compare	Signed	Unsigned
==	BEQ	BEQ
!=	BNE	BNE
>	BGT	BHI
>=	BGE	BHS
<	BLT	BLO
<=	BLE	BLS

Conditional Execution

Add instruction	Condition	Flag tested
ADDEQ r3, r2, r1	Add if EQual	Add if Z = 1
ADDNE r3, r2, r1	Add if Not Equal	Add if Z = 0
ADDHS r3, r2, r1	Add if Unsigned Higher or Same	Add if C = 1
ADDLO r3, r2, r1	Add if Unsigned LOwer	Add if C = 0
ADDMI r3, r2, r1	Add if MInus (Negative)	Add if N = 1
ADDPL r3, r2, r1	Add if PLus (Positive or Zero)	Add if N = 0
ADDVS r3, r2, r1	Add if oVerflow Set	Add if V = 1
ADDVC r3, r2, r1	Add if oVerflow Clear	Add if V = 0
ADDHI r3, r2, r1	Add if Unsigned HIgher	Add if C = 1 & Z = 0
ADDLS r3, r2, r1	Add if Unsigned Lower or Same	Add if C = 0 or Z = 1
ADDGE r3, r2, r1	Add if Signed Greater or Equal	Add if N = V
ADDLT r3, r2, r1	Add if Signed Less Than	Add if N != V
ADDGT r3, r2, r1	Add if Signed Greater Than	Add if Z = 0 & N = V
ADDLE r3, r2, r1	Add if Signed Less than or Equal	Add if Z = 1 or N = !V

Example of Conditional Execution

```
if (a <= 0)
  y = -1;
else
  y = 1;
```

```
a ⟶ r0
y ⟶ r1
```

```
CMP   r0, #0
MOVLE r1, #-1
MOVGT r1, #1
```

LE: Signed Less than or Equal
GT: Signed Greater Than

Conditional Execution in Thumb2

- **IT** (If-then) Block
  - IT makes up to four following instructions conditional
  - Conditions can all be the same or some can be the logical inverse of others
- Syntax: `IT{x{y{z}}} {cond}`
  - cond is a condition code
  - x , y and z specify the condition switch for the 2nd, 3rd and 4th instruction
  - The condition switch can be
    - T (Then), which applies the condition cond to the instruction
    - E (Else), which applies the inverse condition of cond to the instruction
- Example

```
ITTE NE                 ; Next 3 instructions are conditional
ANDNE R0, R0, R1        ; ANDNE does not update condition flags
ADDSNE R2, R2, #1       ; ADDSNE updates condition flags
MOVEQ R2, R3            ; Conditional move
```

Conditional Execution in Thumb2

- ARM Instruction Set
  - A large part of each instruction (4 leading bits) is dedicated to conditional execution
- Thumb Instruction Set
  - Does not support conditional execution
- Thumb2 Instruction Set
  - Variation on conditional execution
  - Instead of compiling a condition in each instruction, there is an IT instruction which checks 8 bits in the condition register

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
APSR	N	Z	C	V	Q				GE							
IPSR												Exception Number				
EPSR						ICI/IT	T			ICI/IT						

Example of Conditional Execution

```
a ⟶ r0
y ⟶ r1
```

```
if (a <= 0)
  y = -1;
else
  y = 1;
```

```
CMP   r0, #0
MOVLE r1, #-1
MOVGT r1, #1
```

LE: Signed Less than or Equal
GT: Signed Greater Than

If-then-else block

```
CMP   r0, #0
ITE     LE
MOVLE r1, #-1
MOVGT r1, #1
```

Example 1: Simple Addition

- Write a program which adds the contents of memory locations 0x20000040 and 0x20000044 and stores the sum to memory location 0x20000048

```
Label   mnemonic                 comments
        LDR R0, =0x20000040  ;Set the operand's address to R0/ set R0 as add. pointer
        LDR R1, [R0], #4     ;Load the first operand to R1 and (post)-increment pointer
        LDR R2, [R0], #4     ;Load the second operand to R2 and increment pointer
        ADD R1, R2           ;R1=R1+R2
        STR R1, [R0]         ;Store R1 to address 0x20000048
Done    B Done               ;Loop at this instruction
```

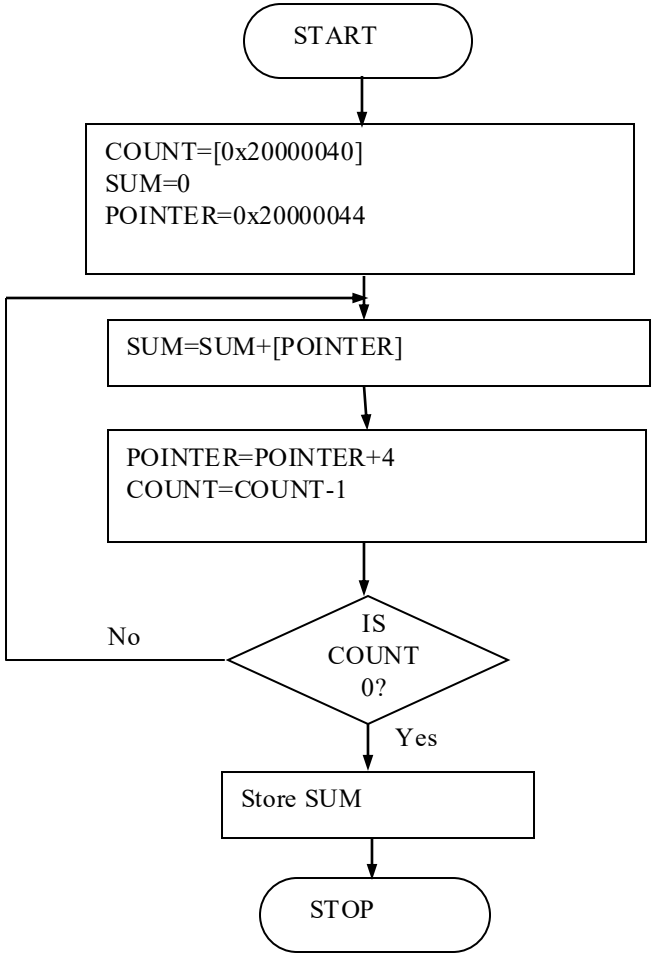
Example 2: Nibble Separation

- Write a program which takes the 32 bits stored starting at the location 0x20000040 and stores the four least significant bits (LSBs) of the least significant byte in location 0x20000044, i.e., separate these bits, and stores the four most significant bits (MSB) of the least significant byte in location 0x20000045 as the least significant bits.

Label	mnemonic	comments
	LDR R0, =0x20000040	;Set the operand's address to R0
	LDRB R1, [R0]	;Load the byte at the address R0
	BFC R1, #4, #4	;Clear bits from 4 to 7 of R1
	STRB R1, [R0, #4]	;Store the byte to R0+4
	LDRB R1, [R0]	;Load the byte at the address R0
	LSR R1, R1, #4	;Shift the four MSB to LSB position
	STRB R1, [R0, #5]	;Store the byte to R0+5
Done	B Done	;Loop at this instruction

Example 3: Array Addition

- Memory location 0x20000040 contains the length (≠0) of a set of numbers. The set starts at memory location 0x20000044. Write a program that stores the sum of numbers (<264) starting from memory location 0x20001000.



Label	mnemonic	comments
	LDR R0, =0x20000040	; Set the address of the length to R0
	LDR R1, [R0], #4	; R1 in COUNT, RO is POINTER
	BFC R2, #0, #32	; R2 and R3 to store SUM, R2 is least sig.
	BFC R3, #0, #32	
	LDR R2, [R0], #4	; Load the first number and increment
POINTER		
	SUBS R1, #1	; Reduce COUNT by 1
	BEQ End	; If COUNT is zero, branch to End
Loop	LDR R4, [R0], #4	; Load the following # and increment POINTER
	ADDS R2, R4	; Add the number to SUM's least sig. word
	BCC Cont	; If there is no carry, branch to Cont
	ADD R3, #1	; else increment R3
Cont	SUBS R1, #1	; Reduce COUNT by 1
	BEQ End	; If COUNT is zero, branch to End
	B Loop	
End	LDR R0, =0x20001000	; Set the address of storage to R0
	STRD R2, R3, [R0]	; Save R2 and then R3 starting from add. R0
Done	B Done	; Loop at this instruction

Example 4: Condition Code Flags in the Program Status Register (PSR)

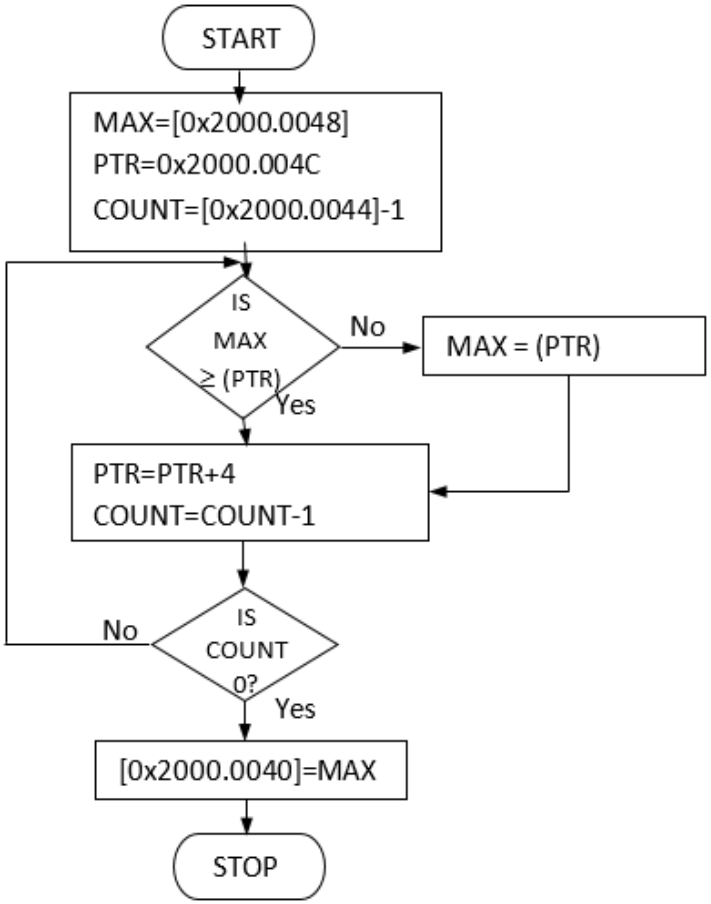
- Assume register R1 contains the following data and register R2 contains 0x1000.0000 before the execution of "SUBS R1, R1, R2" instruction. What is the result in R1 and the N, Z, C, and V bits?

R1: 0x2000.0000;
0x0000.1000;
0x1000.0000

Before R1	After R1	N Z C V
0x2000.0000	0x1000.0000	0 0 1 0
0x0000.1000	0xF000.1000	1 0 0 0
0x1000.0000	0x0000.0000	0 1 1 0

Example 5: Maximum Value

- Length (≠0) of a memory array, which contains unsigned numbers, is in 0x2000.0044 and the array starts at 0x2000.0048. Write a program that places the maximum value in the array in 0x2000.0040.



Example 5: Solution

```
label           mnemonic                    comment
                LDR R0, =0x20000044         ; R0 is the pointer to the length
                LDR R4, =0x20000040         ; R4 is the pointer to the max storage
                LDR R1, [R0], #4            ; R1 holds the length/count
information
                LDR R2, [R0]                ; R2 holds the maximum
                SUBS R1, #1                 ; Decrement counter
                BEQ Final                   ; If it is the end of the array, finish
Loop            LDR R3, [R0, #4]!       ; R3 holds the next data
                CMP R2, R3                  ; R2 - R3
                BGE Cont                    ; If R2>R3, go to Cont
                LDR R2, [R0]                ; Else load the new data to max (R2)
Cont            SUBS R1, #1             ; Decrement counter
                BEQ Final                   ; If it is the end of the array, finish
                B Loop                      ;Else go to Loop
Final           STR R2, [R4]            ; Store max
Forever         B Forever
```

Example 5: Initial Values

- Initial Values of the Locations 0x2000.0040:0x2000.0053

Location	Content
0x2000.0040	0x0000.0000
0x2000.0044	0x0000.0003
0x2000.0048	0x0000.0005
0x2000.004C	0x0000.0002
0x2000.0050	0x0000.0006

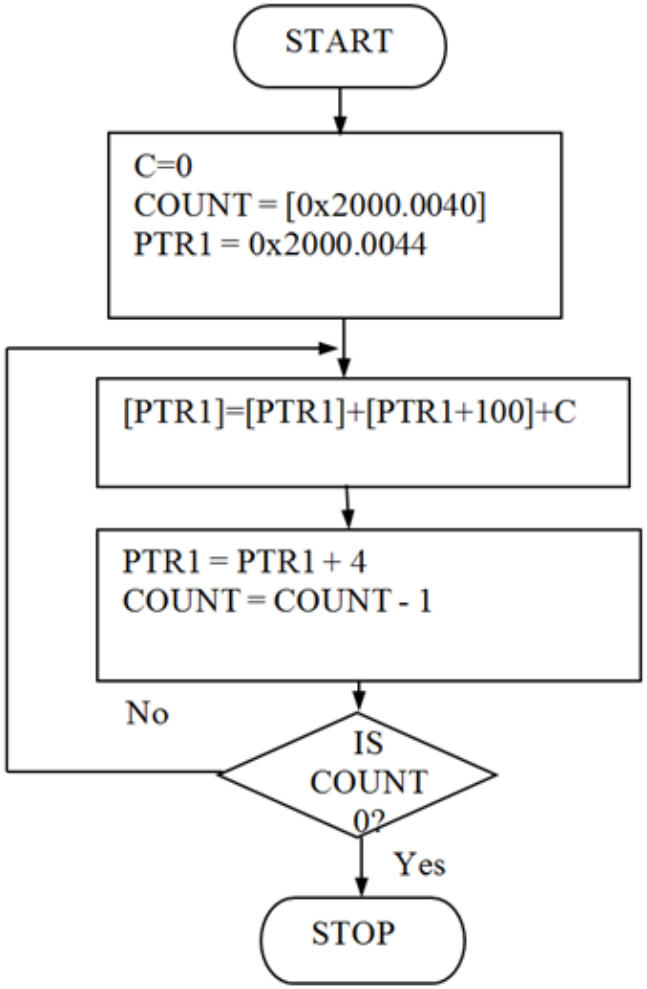
Example 5: Example Trace

Instruction	R1	R2	R3	R0	CZ	R4	0x 20000040
LDR R0, =0x20000044				0x 20000044			
LDR R4, =0x20000040						0x20000040	
LDR R1, [R0], #4	3			0x 20000048			
LDR R2, [R0]		5					
SUBS R1, #1	2				10		
BEQ Final							
LDR R3, [R0, #4]!			2	0x 2000004C			
CMP R2, R3					10		
BGE Cont							
SUBS R1, #1	1				10		
BEQ Final							
B Loop							
LDR R3, [R0, #4]!			6	0x 20000050			
CMP R2, R3					00		
BGE Cont							
LDR R2, [R0]		6					
SUBS R1, #1	0				01		
BEQ Final							
STR R2, [R4]							6
B Forever							
B Forever							
….							

Example 6: Multiple Precision Arithmetic

- Two n-word (32 bit) numbers will be added. The number of words (n, which is greater than 0 and less than 256) in the numbers is given in location 0x2000.0040. First number starts (least significant word first) at 0x2000.0044 and the second number at 0x2000.0144. The sum is to replace the first number.

Location	Before	After
0x20000040	0x00001DFE	0x00103E2E
0x20000044	0xAB11FFAA	0x29BCBB77
0x20000048	0x01A2B3C4	0x02446688
0x2000004C	0x00000102	
...		
0x20000144	0x00102030	
0x20000148	0x7EAABBCC	
0x2000014C	0x00A1B2C3	



Example 6: Multiple Precision Arithmetic

```
label                   mnemonic                comment

                        LDR R0, =0x20000040     ; Pointer to address 0x20000040

                        LDR R2, [R0], #4        ; R2 is Length counter, R0 is
incremented
                        ADDS R2, #0             ; Dummy instruction to clear C bit
Loop                    LDR R3, [R0]            ; First number is read to R3
                        LDR R4, [R0, #0x0100]   ; Second number to R4
                        ADCS R3, R3, R4         ; Add second number to first
                        STR R3, [R0], #4        ; Store R3, increment pointer
                        SUBS R2, #1     ; Decrement counter without changing PSR
                        BNE Loop        ; If counter is not equal to zero go to Loop

Forever         B Forever
```

Example 7: Square From a Lookup Table

- Write a program to find the square of a 16-bit binary number from a lookup table. The table starts at SQTAB, location NUM contains the number whose square is required and the result will be saved in location NUM.

  Assume that SQTAB contains 0, 1, 4, 9, 16, 25, 36, 49, ….

```
label       mnemonic                comment
            LDR R0, =SQTAB          ; R0 is set as the pointer to the table
            LDR R2, =NUM            ; R1 is set as the pointer to the number
            LDR R1, [R2]            ; Load number in R1
            ADD R0, R0, R1, LSL #2  ; R0 is moved to the location of square
            LDR R1, [R0]            ; Square of the number is loaded to R1
            STR R1, [R2]            ; It is stored to NUM

Forever     B Forever
```

Example 8: Character Manipulation

- Determine the length of a string of ASCII characters (one ASCII character is coded by one byte) starting at ARRAY and ending with period ".". Store the length of the string to location LENGTH. ASCII equivalent of "." is 0x2E.
- Example

Location	Content	
ARRAY	54	T
ARRAY+1	4F	O
ARRAY+2	4F	O
ARRAY+3	20	
ARRAY+4	4C	L
ARRAY+5	41	A
ARRAY+6	54	T
ARRAY+7	45	E
ARRAY+8	2E	.

Example 8: Character Manipulation

- Determine the length of a string of ASCII characters (one ASCII character is coded by one byte) starting at ARRAY and ending with period ".". Store the length of the string to location LENGTH. ASCII equivalent of "." is 0x2E.

```
label       mnemonic                comment
            LDR R0, =ARRAY          ; R0 is pointer to ARRAY
            MOV R1, #0x00           ; R1 holds length counter, initialized to 0
Loop        LDRB R2, [R0], #1       ; Read the data, increment pointer by one
            CMP R2, #0x2E           ; Compare the byte with '.'
            BEQ End                 ; If equal go to End
            ADD R1, R1, #1          ; else increment counter and go to Loop
            B Loop                  ; Branch to Loop
End         LDR R0, =LENGTH         ; Store the length information
            STR R1, [R0]
Forever     B Forever
```

Example 9: Pattern Comparison

- Two strings of 32 bit words start in memory locations STRING1 and STRING2, respectively. Memory location LENGTH contains the length ($\neq 0$) of the strings. Write a program that compares these two strings. If they are equal, the program will place zeros to location LENGTH, otherwise ones will be placed.

Example 9: Pattern Comparison

```
Label       mnemonic                comment
            LDR R0, =STRING1        ; Pointer to STRING1
            LDR R1, =STRING2        ; Pointer to STRING2
            LDR R2, =LENGTH         ; Length and final decision
            LDR R3, [R2]            ; R3 keeps the length
            MOV R7, #0xFFFFFFFF     ; not equal = Set R7 bits to 1's
Loop        LDR R5, [R0], #4        ; R5 keeps the element from STRING1
            LDR R6, [R1], #4        ; R6 keeps the element from STRING2
            CMP R5, R6              ; Compare R5 and R6
            BNE End                 ; If they are not equal go to End
            SUBS R3, #1             ; else continue checking and decrement counter
            BNE Loop
            BFC R7, #0, #32         ; equal = Set R7 bits to 0's
End         STR R7, [R2]            ; store the decision
Forever     B Forever
```

Example 10: Finite Impulse Response Filtering

- For a causal discrete-time FIR filter of order N, each value of the output sequence is a weighted sum of the most recent input values:

$$y[n] = c_0 x[n] + c_1 x[n-1] + \cdots + c_N x[n-N] = \sum_{i=0}^{N} c_i x[n-i]$$

Write a program which computes output $\boldsymbol{y[n]}$ when the coefficients (c's and x's) are stored starting at memory locations COEFF and INPUT, respectively. The order N of the filter is stored at FORDER. Store the computed value of $\boldsymbol{y[n]}$ to memory location OUTPUT.

Example 10: Finite Impulse Response Filtering

```
Label       mnemonic              comment
            MOV R0,#0             ; use R0 for i
            MOV R8,#0             ; use separate index for arrays
            LDR R2,=FORDER        ; get address for N
            LDR R1,[R2]           ; get value of N
            MOV R2,#0             ; use R2 for y[n]
            LDR R3, =COEFF        ; load R3 with base of c
            LDR R5,=INPUT         ; load R5 with base of x
Loop        LDR R4,[R3,R8]        ; get c[i]
            LDR R6,[R5,R8]        ; get x[n-i]
            MUL R4,R4,R6          ; compute c[i]*x[n-i]
            ADD R2,R2,R4          ; add into running sum y[n]
            ADD R8,R8,#4          ; add word offset to array index
            ADD R0,R0,#1          ; add 1 to i
            CMP R0,R1             ; exit?
            BLT loop              ; if i < N, continue
            LDR R3,=OUTPUT        ; else get address for y[n]
            STR R2, [R3]          ; store y[n]
Forever     B Forever
```

Example 11: C vs Assembly

C:

```
int p, k;
int w[10];
int *ps;
p = k;
p = w[k];
ps = &k;
p = *ps;
```

```
LDR     R0,=K       ;r0 = address of variable k; R0 points to K
LDR     R1,[R0]     ;read value at  k from memory and put to r1
LDR     R0,=P               ;r0 = address of p
STR     R1,[R0]     ;write value in r1 to memory address p ; (P)=K
LDR     R0,=K               ;address of variable k
LDR     R1,[R0]     ;load value at k
LDR     R0,=W               ;base address of array W
LDR     R2,[R0,R1,LSL #2] ;value of w[k] (scale index k×4 for 32-bit words)
LDR     R0,=P               ;address of variable p
STR     R2,[R0]     ;write to variable p
LDR     R0,=K               ;address of k -> r0
LDR     R1,=PS      ;address of ps -> r1
STR     R0,[R1]     ;store address of k into pointer variable ps
 LDR    R2,=P               ;address of p -> r2
LDR     R3,[R0]     ;value of variable k -> r3 (r0 = address of k)
STR     R3,[R2]     ;store value of variable k -> variable p

K       DCD 333             ;dummy values
P       DCD 444
W       DCD 555
PS      DCD 777
```

Example 11: Aligning Data

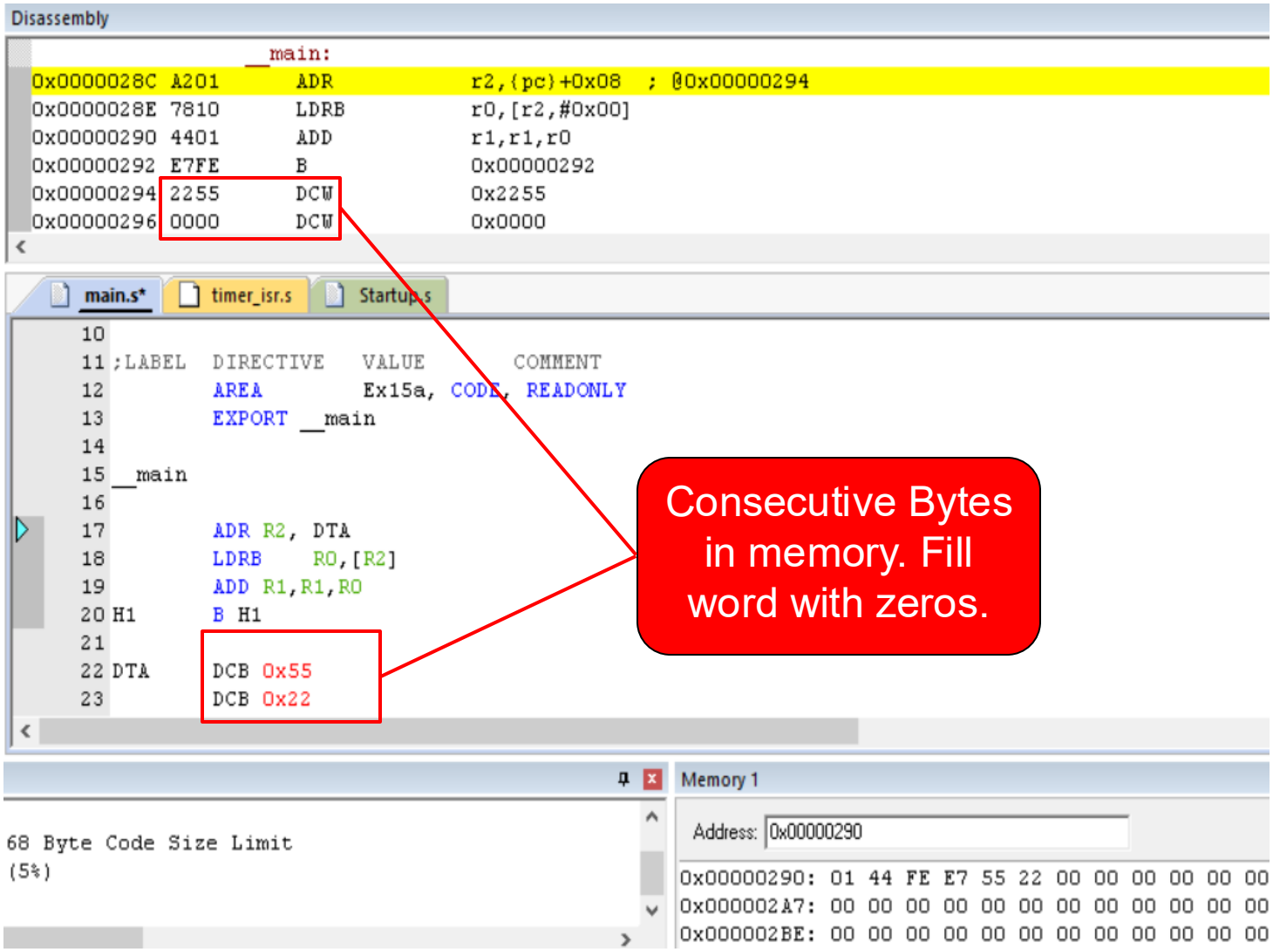
- Example Code

```
AREA |.text|, READONLY,CODE
_main
    ADR   R2, DTA
    LDRB  R0,[R2]
    ADD R1,R1,R0
H1  B H1

DTA DCB   0x55
    DCB   0x22
    END
```

Data Section

Example 11: Aligning Data



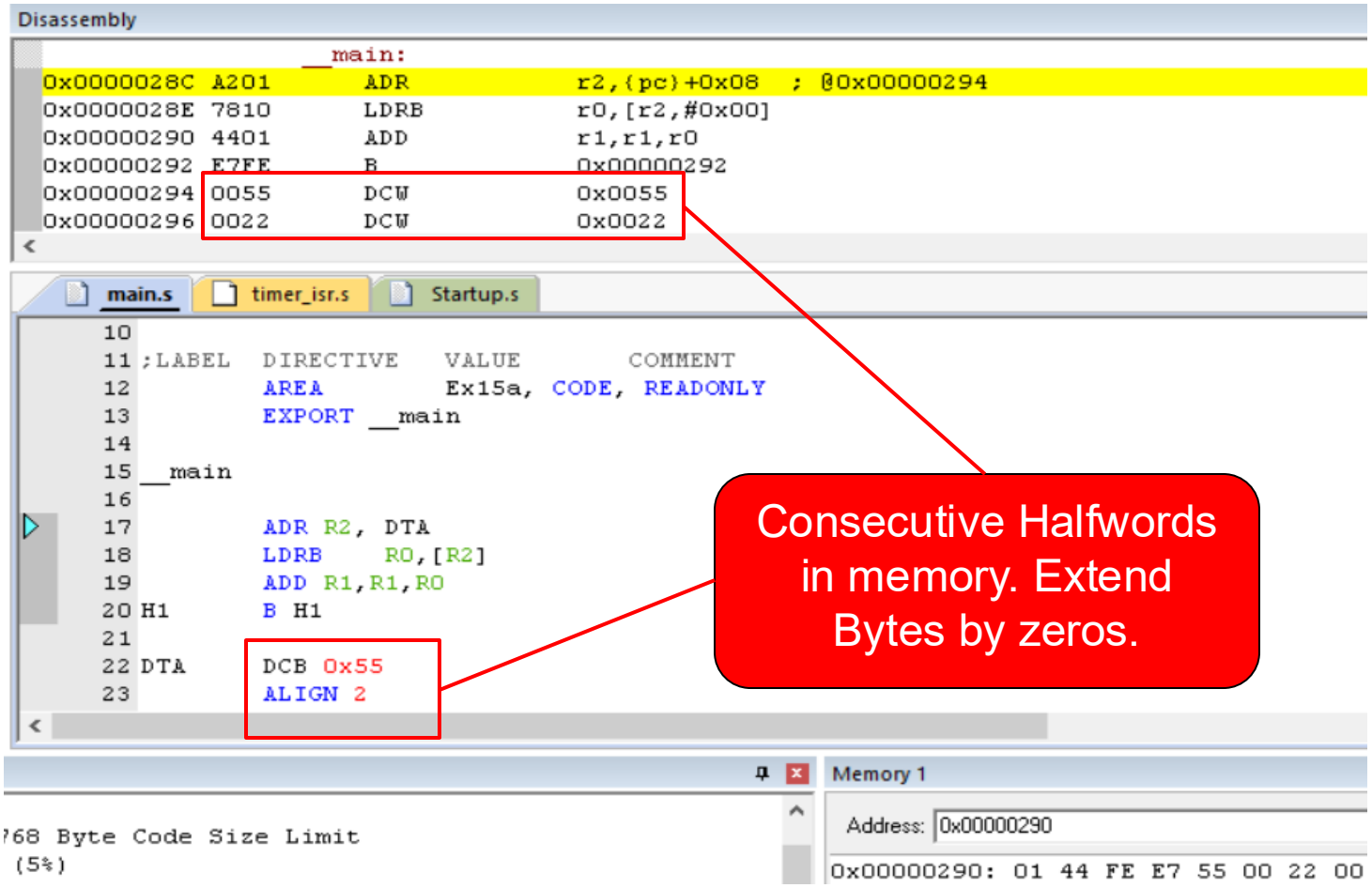
Example 11: Aligning Data

- Modified Example Code with ALIGN

```
AREA |.text|, READONLY,CODE
_main
    ADR   R2, DTA
    LDRB  R0,[R2]
    ADD R1,R1,R0
H1  B H1

DTA DCB   0x55
    ALIGN 2
    DCB   0x22
    END
```

Example 11: Aligning Data



Example 12: Accessing Non-aligned Data

- Show the data transfer of the following cases and indicate the number of memory cycle times it takes for data transfer.
  Assume R2=0x4598F31E

```
LDR R1, = 0x40000000  ; R1=0x40000000
LDR R2, =0x4598F31E   ;R2=0x4598F31E
STR R2, [R1]     ; store R2 to location  0x400000000
ADD R1,R1,#1     ; R1=R1+1=0x40000001
STR R2, [R1]     ; Store R2 to location 0x40000001
ADD R1,R1,#1     ; R1=R1+1=0x40000002
STR R2, [R1]     ; Store R2 to location 0x40000002
```

```
STR R2, [R1]     ; store R2 to location  0x400000000
```

Memory address	Memory data after cycle1
0x40000003	0x45
0x40000002	0x98
0x40000001	0xF3
0x40000000	0x1E

Aligned: 1 cycle

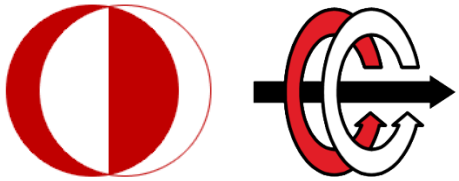
```
STR R2, [R1]     ; Store R2 to location 0x40000001
```

Memory address	Memory data after cycle1	Memory data after cycle1
0x40000005		
0x40000004		0x45
0x40000003	0x98	0x98
0x40000002	0xF3	0xF3
0x40000001	0x1E	0x1E
0x40000000		

Unaligned: 2 cycles

```
STR R2, [R1]     ; Store R2 to location 0x40000002
```

Memory address	Memory data after cycle1	Memory data after cycle1
0x40000005		0x45
0x40000004		0x98
0x40000003	0xF3	0xF3
0x40000002	0x1E	0x1E
0x40000001		
0x40000000		



Assembly Programming Details

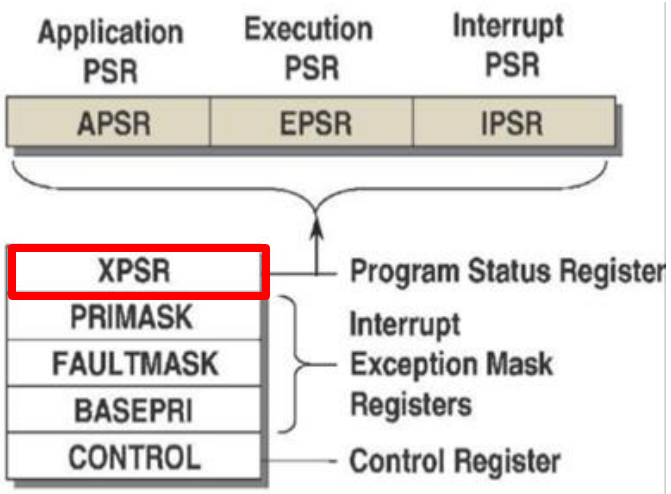
Week 3

ARM Cortex-M3/4 Registers: PSR

Back

- Program Status Register (PSR) is three views of same register
- Application PSR (APSR)
  - Condition code flag bits Negative, Zero, Carry, Overflow, DSP overflow and saturation, Great-Than or Equal
- Interrupt PSR (IPSR)
  - Holds exception number of currently executing ISR
- Execution PSR (EPSR)
  - ICI/IT, Interrupt-Continuable Instruction, IF-THEN instruction
  - Thumb state, always 1

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
APSR	N	Z	C	V	Q				GE*							
IPSR												Exception Number				
EPSR						ICI/IT	T			ICI/IT						



Instruction Set: Comparison

Back

- ARM now called AArch32

32-bit	32-bit	32-bit	32-bit	32-bit

- Thumb (includes all ARM 32 bit instructions)

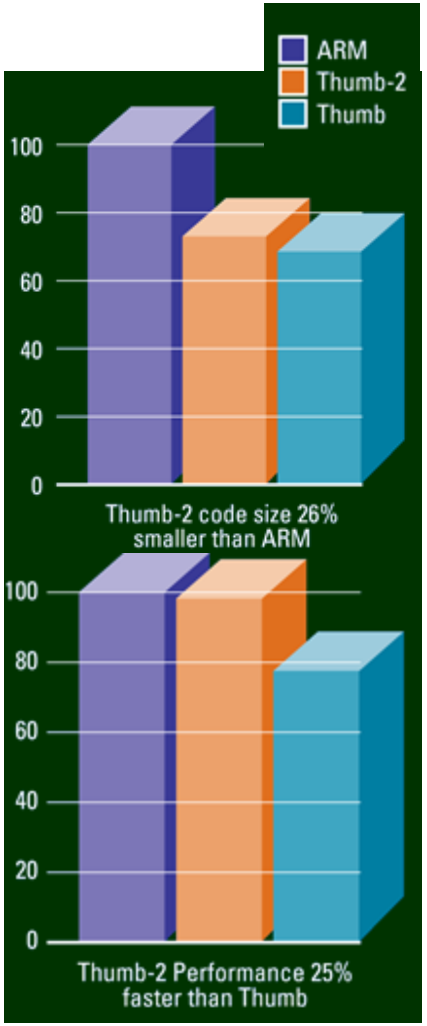
16-bit	16-bit	16-bit	16-bit	16-bit	16-bit	16-bit	16-bit	16-bit	16-bit

- Thumb-2

32-bit	32-bit	16-bit	16-bit	16-bit	32-bit	16-bit

- Range of Instructions

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				●



ARM Cortex-M3/4 Registers: PSR Details

APSR

Bits	Name	Function
[31]	N	Negative flag
[30]	Z	Zero flag
[29]	C	Carry or borrow flag
[28]	V	Overflow flag
[27]	Q	DSP overflow and saturation flag
[26:20]	-	Reserved
[19:16]	GE[3:0]	Greater than or Equal flags.
[15:0]	-	Reserved

IPSR

Bits	Name	Function
[31:9]	-	Reserved
[8:0]	ISR_NUMBER	This is the number of the current exception: 0 = Thread mode 1 = Reserved 2 = NMI 3 = HardFault 4 = MemManage 5 = BusFault 6 = UsageFault 7-10 = Reserved 11 = SVCall 12 = Reserved for Debug 13 = Reserved 14 = PendSV 15 = SysTick 16 = IRQ0. .

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
APSR	N	Z	C	V	Q				GE*							
IPSR												Exception Number				
EPSR						ICI/IT	T			ICI/IT						

Note: GE flags are only available on Cortex-M4 and M7

Back

ADD (immediate)

ADD (immediate)

This instruction adds an immediate value to a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 All versions of the Thumb instruction set.

```
ADDS <Rd>,<Rn>,#<imm3>                    Outside IT block.
ADD<c> <Rd>,<Rn>,#<imm3>                  Inside IT block.
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	imm3			Rn			Rd		

```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm3, 32);
```

Encoding T2 All versions of the Thumb instruction set.

```
ADDS <Rdn>,#<imm8>                        Outside IT block.
ADD<c> <Rdn>,#<imm8>                      Inside IT block.
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	Rdn							imm8			

```
d = UInt(Rdn);  n = UInt(Rdn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);
```

Encoding T3 ARMv7-M

```
ADD{S}<c>.W <Rd>,<Rn>,#<const>
```

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	1	0	i	0	1	0	0	0	S	Rn				0	imm3			Rd				imm8							

```
if Rd == '1111' && S == '1' then SEE CMN (immediate);
if Rn == '1101' then SEE ADD (SP plus immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ThumbExpandImm(i:imm3:imm8);
if d == 13 || (d == 15 && S == '0') || n == 15 then UNPREDICTABLE;
```

Encoding T4 ARMv7-M

```
ADDW<c> <Rd>,<Rn>,#<imm12>
```

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	1	0	i	1	0	0	0	0	0	Rn				0	imm3			Rd				imm8							

Back

ADD (register)

ADD (register)

ADD (register) adds a register value and an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding T1 All versions of the Thumb instruction set.

`ADDS <Rd>,<Rn>,<Rm>` Outside IT block.

`ADD<c> <Rd>,<Rn>,<Rm>` Inside IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	Rm			Rn			Rd		

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 All versions of the Thumb instruction set.

`ADD<c> <Rdn>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0		Rm				Rdn		
							DN								

```
if (DN:Rdn) == '1101' || Rm == '1101' then SEE ADD (SP plus register);
d = UInt(DN:Rdn);  n = UInt(DN:Rdn);  m = UInt(Rm);  setflags = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
if d == 15 && m == 15 then UNPREDICTABLE;
```

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			

Back

Assembly Directives

- Description
  - Directives are used to provide key information for assembly
  - Important: Directives are NOT instruction
  - Examples

AREA	Make a new block of data or code
ENTRY	Declare an entry point where the program execution starts
ALIGN	Align data or code to a particular memory boundary
DCB	Allocate one or more bytes (8 bits) of data
DCW	Allocate one or more half-words (16 bits) of data
DCD	Allocate one or more words (32 bits) of data
SPACE	Allocate a zeroed block of memory with a particular size
FILL	Allocate a block of memory and fill with a given value.
EQU	Give a symbol name to a numeric constant
RN	Give a symbol name to a register
EXPORT	Declare a symbol and make it referable by other source files
IMPORT	Provide a symbol defined outside the current source file
INCLUDE/GET	Include a separate source file within the current source file
PROC	Declare the start of a procedure
ENDP	Designate the end of a procedure
END	Designate the end of a source file

Back