Table of Contents

1 Introduction ................................................................................................................................. 12
  1.1 Instruction Set Summary ......................................................................................................... 12
  1.2 About The Instruction Descriptions .................................................................................... 15
  1.2.1 Operands .......................................................................................................................... 15
  1.2.2 Restrictions When Using the PC or SP ............................................................................. 15
  1.2.3 Flexible Second Operand .................................................................................................. 15
  1.2.4 Shift Operations ............................................................................................................... 17
  1.2.5 Address Alignment ......................................................................................................... 20
  1.2.6 PC-Relative Expressions ................................................................................................. 20
  1.2.7 Conditional Execution ................................................................................................... 20
  1.2.8 Instruction Width Selection ............................................................................................ 22

2 Memory Access Instructions ........................................................................................................ 24
  2.1 ADR .................................................................................................................................. 25
    2.1.1 Syntax .......................................................................................................................... 25
    2.1.2 Operation ..................................................................................................................... 25
    2.1.3 Restrictions ................................................................................................................... 25
    2.1.4 Condition Flags ............................................................................................................ 25
    2.1.5 Examples ...................................................................................................................... 25
  2.2 LDR and STR (Immediate Offset) .......................................................................................... 26
    2.2.1 Syntax .......................................................................................................................... 26
    2.2.2 Operation ..................................................................................................................... 27
    2.2.3 Restrictions ................................................................................................................... 27
    2.2.4 Condition Flags ............................................................................................................ 28
    2.2.5 Examples ...................................................................................................................... 28
  2.3 LDR and STR (Register Offset) ............................................................................................. 29
    2.3.1 Syntax .......................................................................................................................... 29
    2.3.2 Operation ..................................................................................................................... 29
    2.3.3 Restrictions ................................................................................................................... 30
    2.3.4 Condition Flags ............................................................................................................ 30
    2.3.5 Examples ...................................................................................................................... 30
  2.4 LDR and STR (Unprivileged Access) ..................................................................................... 31
    2.4.1 Syntax .......................................................................................................................... 31
    2.4.2 Operation ..................................................................................................................... 31
    2.4.3 Restrictions ................................................................................................................... 32
    2.4.4 Condition Flags ............................................................................................................ 32
    2.4.5 Examples ...................................................................................................................... 32
  2.5 LDR (PC-Relative) .................................................................................................................. 33
    2.5.1 Syntax .......................................................................................................................... 33
    2.5.2 Operation ..................................................................................................................... 33
    2.5.3 Restrictions ................................................................................................................... 34
    2.5.4 Condition Flags ............................................................................................................ 34
    2.5.5 Examples ...................................................................................................................... 34
  2.6 LDM and STM ...................................................................................................................... 35
    2.6.1 Syntax .......................................................................................................................... 35
    2.6.2 Operation ..................................................................................................................... 36
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6.3</td>
<td>Restrictions</td>
<td>36</td>
</tr>
<tr>
<td>2.6.4</td>
<td>Condition Flags</td>
<td>36</td>
</tr>
<tr>
<td>2.6.5</td>
<td>Examples</td>
<td>36</td>
</tr>
<tr>
<td>2.6.6</td>
<td>Incorrect Examples</td>
<td>36</td>
</tr>
<tr>
<td>2.7</td>
<td>PUSH and POP</td>
<td>37</td>
</tr>
<tr>
<td>2.7.1</td>
<td>Syntax</td>
<td>37</td>
</tr>
<tr>
<td>2.7.2</td>
<td>Operation</td>
<td>37</td>
</tr>
<tr>
<td>2.7.3</td>
<td>Restrictions</td>
<td>37</td>
</tr>
<tr>
<td>2.7.4</td>
<td>Condition Flags</td>
<td>37</td>
</tr>
<tr>
<td>2.7.5</td>
<td>Examples</td>
<td>37</td>
</tr>
<tr>
<td>2.8</td>
<td>LDREX and STREX</td>
<td>39</td>
</tr>
<tr>
<td>2.8.1</td>
<td>Syntax</td>
<td>39</td>
</tr>
<tr>
<td>2.8.2</td>
<td>Operation</td>
<td>39</td>
</tr>
<tr>
<td>2.8.3</td>
<td>Restrictions</td>
<td>40</td>
</tr>
<tr>
<td>2.8.4</td>
<td>Condition Flags</td>
<td>40</td>
</tr>
<tr>
<td>2.8.5</td>
<td>Examples</td>
<td>40</td>
</tr>
<tr>
<td>2.9</td>
<td>CLREX</td>
<td>41</td>
</tr>
<tr>
<td>2.9.1</td>
<td>Syntax</td>
<td>41</td>
</tr>
<tr>
<td>2.9.2</td>
<td>Operation</td>
<td>41</td>
</tr>
<tr>
<td>2.9.3</td>
<td>Condition Flags</td>
<td>41</td>
</tr>
<tr>
<td>2.9.4</td>
<td>Examples</td>
<td>41</td>
</tr>
<tr>
<td>3</td>
<td>General Data Processing Instructions</td>
<td>42</td>
</tr>
<tr>
<td>3.1</td>
<td>ADD, ADC, SUB, SBC, and RSB</td>
<td>43</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Syntax</td>
<td>43</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Operation</td>
<td>43</td>
</tr>
<tr>
<td>3.1.3</td>
<td>Restrictions</td>
<td>44</td>
</tr>
<tr>
<td>3.1.4</td>
<td>Condition Flags</td>
<td>44</td>
</tr>
<tr>
<td>3.1.5</td>
<td>Examples</td>
<td>45</td>
</tr>
<tr>
<td>3.1.6</td>
<td>Multiword Arithmetic Examples</td>
<td>45</td>
</tr>
<tr>
<td>3.2</td>
<td>AND, ORR, EOR, BIC, and ORN</td>
<td>46</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Syntax</td>
<td>46</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Operation</td>
<td>46</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Restrictions</td>
<td>47</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Condition Flags</td>
<td>47</td>
</tr>
<tr>
<td>3.2.5</td>
<td>Examples</td>
<td>47</td>
</tr>
<tr>
<td>3.3</td>
<td>ASR, LSL, LSR, ROR, and RRX</td>
<td>48</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Syntax</td>
<td>48</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Operation</td>
<td>49</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Restrictions</td>
<td>49</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Condition Flags</td>
<td>49</td>
</tr>
<tr>
<td>3.3.5</td>
<td>Examples</td>
<td>49</td>
</tr>
<tr>
<td>3.4</td>
<td>CLZ</td>
<td>50</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Syntax</td>
<td>50</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Operation</td>
<td>50</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Restrictions</td>
<td>50</td>
</tr>
<tr>
<td>3.4.4</td>
<td>Condition Flags</td>
<td>50</td>
</tr>
<tr>
<td>3.4.5</td>
<td>Examples</td>
<td>50</td>
</tr>
<tr>
<td>3.5</td>
<td>CMP and CMN</td>
<td>51</td>
</tr>
</tbody>
</table>
### Multiply and Divide Instructions

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>60</td>
</tr>
<tr>
<td>4.1.1</td>
<td>60</td>
</tr>
<tr>
<td>4.1.2</td>
<td>60</td>
</tr>
<tr>
<td>4.1.3</td>
<td>60</td>
</tr>
<tr>
<td>4.1.4</td>
<td>61</td>
</tr>
<tr>
<td>4.1.5</td>
<td>61</td>
</tr>
<tr>
<td>4.2</td>
<td>62</td>
</tr>
<tr>
<td>4.2.1</td>
<td>62</td>
</tr>
<tr>
<td>4.2.2</td>
<td>62</td>
</tr>
<tr>
<td>4.2.3</td>
<td>62</td>
</tr>
<tr>
<td>4.2.4</td>
<td>63</td>
</tr>
<tr>
<td>4.2.5</td>
<td>63</td>
</tr>
<tr>
<td>4.3</td>
<td>64</td>
</tr>
<tr>
<td>4.3.1</td>
<td>64</td>
</tr>
<tr>
<td>4.3.2</td>
<td>64</td>
</tr>
<tr>
<td>4.3.3</td>
<td>64</td>
</tr>
<tr>
<td>4.3.4</td>
<td>64</td>
</tr>
<tr>
<td>4.3.5</td>
<td>64</td>
</tr>
</tbody>
</table>
# Table of Contents

## 5 Saturating Instructions
5.1 SSAT and USAT
5.1.1 Syntax
5.1.2 Operation
5.1.3 Restrictions
5.1.4 Condition Flags
5.1.5 Examples

## 6 Bitfield Instructions
6.1 BFC and BFI
6.1.1 Syntax
6.1.2 Operation
6.1.3 Restrictions
6.1.4 Condition Flags
6.1.5 Examples
6.2 SBFX and UBFX
6.2.1 Syntax
6.2.2 Operation
6.2.3 Restrictions
6.2.4 Condition Flags
6.2.5 Examples
6.3 SXT and UXT
6.3.1 Syntax
6.3.2 Operation
6.3.3 Restrictions
6.3.4 Condition Flags
6.3.5 Examples

## 7 Branch and Control Instructions
7.1 B, BL, BX, and BLX
7.1.1 Syntax
7.1.2 Operation
7.1.3 Restrictions
7.1.4 Condition Flags
7.1.5 Examples
7.2 CBZ and CBNZ
7.2.1 Syntax
7.2.2 Operation
7.2.3 Restrictions
7.2.4 Condition Flags
7.2.5 Examples
7.3 IT
7.3.1 Syntax
7.3.2 Operation
7.3.3 Restrictions
7.3.4 Condition Flags
7.3.5 Example
7.4 TBB and TBH
7.4.1 Syntax
7.4.2 Operation
## List of Figures

<table>
<thead>
<tr>
<th>Figure 1-1.</th>
<th>ASR #3</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1-2.</td>
<td>LSR #3</td>
<td>18</td>
</tr>
<tr>
<td>Figure 1-3.</td>
<td>LSL #3</td>
<td>19</td>
</tr>
<tr>
<td>Figure 1-4.</td>
<td>ROR #3</td>
<td>19</td>
</tr>
<tr>
<td>Figure 1-5.</td>
<td>RRX</td>
<td>19</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1-1.</td>
<td>Cortex-M3 Instructions</td>
<td>12</td>
</tr>
<tr>
<td>Table 1-2.</td>
<td>Condition Code Suffixes</td>
<td>22</td>
</tr>
<tr>
<td>Table 2-1.</td>
<td>Memory Access Instructions</td>
<td>24</td>
</tr>
<tr>
<td>Table 2-2.</td>
<td>Offset Ranges</td>
<td>27</td>
</tr>
<tr>
<td>Table 2-3.</td>
<td>Offset Ranges</td>
<td>34</td>
</tr>
<tr>
<td>Table 3-1.</td>
<td>General Data Processing Instructions</td>
<td>42</td>
</tr>
<tr>
<td>Table 4-1.</td>
<td>Multiply and Divide Instructions</td>
<td>59</td>
</tr>
<tr>
<td>Table 5-1.</td>
<td>Saturating Instructions</td>
<td>65</td>
</tr>
<tr>
<td>Table 6-1.</td>
<td>Bitfield Instructions</td>
<td>68</td>
</tr>
<tr>
<td>Table 7-1.</td>
<td>Branch and Control Instructions</td>
<td>73</td>
</tr>
<tr>
<td>Table 7-2.</td>
<td>Branch Ranges</td>
<td>75</td>
</tr>
<tr>
<td>Table 8-1.</td>
<td>Miscellaneous Instructions</td>
<td>82</td>
</tr>
</tbody>
</table>
List of Examples

Example 1-1. Absolute Value ................................................................. 22
Example 1-2. Compare and Update Value ........................................... 22
Example 1-3. Instruction Width Selection ........................................... 23
Example 3-1. 64-Bit Addition ............................................................... 45
Example 3-2. 96-Bit Subtraction ......................................................... 45
1 Introduction

Each of the following chapters describes a functional group of Cortex-M3 instructions. Together they describe all the instructions supported by the Cortex-M3 processor:

- “Memory Access Instructions” on page 24
- “General Data Processing Instructions” on page 42
- “Multiply and Divide Instructions” on page 59
- “Saturating Instructions” on page 65
- “Bitfield Instructions” on page 68
- “Branch and Control Instructions” on page 73
- “Miscellaneous Instructions” on page 82

1.1 Instruction Set Summary

The processor implements a version of the Thumb instruction set. Table 1-1 on page 12 lists the supported instructions.

In Table 1-1 on page 12:

- Angle brackets, <>, enclose alternative forms of the operand.
- Braces, {}, enclose optional operands.
- The Operands column is not exhaustive.
- Op2 is a flexible second operand that can be either a register or a constant.
- Most instructions can use an optional condition code suffix.

For more information on the instructions and operands, see the instruction descriptions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operands</th>
<th>Brief Description</th>
<th>Flags</th>
<th>See Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC, ADCS</td>
<td>(Rd,) Rn, Op2</td>
<td>Add with carry</td>
<td>N,Z,C,V</td>
<td>43</td>
</tr>
<tr>
<td>ADD, ADDS</td>
<td>(Rd,) Rn, Op2</td>
<td>Add</td>
<td>N,Z,C,V</td>
<td>43</td>
</tr>
<tr>
<td>ADD, ADDW</td>
<td>(Rd,) Rn, #imm12</td>
<td>Add</td>
<td>N,Z,C,V</td>
<td>43</td>
</tr>
<tr>
<td>ADR</td>
<td>Rd, label</td>
<td>Load PC-relative address</td>
<td>-</td>
<td>25</td>
</tr>
<tr>
<td>AND, ANDS</td>
<td>(Rd,) Rn, Op2</td>
<td>Logical AND</td>
<td>N,Z,C</td>
<td>46</td>
</tr>
<tr>
<td>ASR, ASRS</td>
<td>Rd, Rm, &lt;Rs</td>
<td>#n&gt;</td>
<td>Arithmetic shift right</td>
<td>N,Z,C</td>
</tr>
<tr>
<td>B</td>
<td>label</td>
<td>Branch</td>
<td>-</td>
<td>74</td>
</tr>
<tr>
<td>BFC</td>
<td>Rd, #lsb, #width</td>
<td>Bit field clear</td>
<td>-</td>
<td>69</td>
</tr>
<tr>
<td>BFI</td>
<td>Rd, Rn, #lsb, #width</td>
<td>Bit field insert</td>
<td>-</td>
<td>69</td>
</tr>
<tr>
<td>BIC, BICS</td>
<td>(Rd,) Rn, Op2</td>
<td>Bit clear</td>
<td>N,Z,C</td>
<td>43</td>
</tr>
<tr>
<td>BKPT</td>
<td>#imm</td>
<td>Breakpoint</td>
<td>-</td>
<td>83</td>
</tr>
<tr>
<td>BL</td>
<td>label</td>
<td>Branch with link</td>
<td>-</td>
<td>74</td>
</tr>
<tr>
<td>BLX</td>
<td>Rm</td>
<td>Branch indirect with link</td>
<td>-</td>
<td>74</td>
</tr>
<tr>
<td>BX</td>
<td>Rm</td>
<td>Branch indirect</td>
<td>-</td>
<td>74</td>
</tr>
<tr>
<td>CBNZ</td>
<td>Rn, label</td>
<td>Compare and branch if non-zero</td>
<td>-</td>
<td>76</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Operands</td>
<td>Brief Description</td>
<td>Flags</td>
<td>See Page</td>
</tr>
<tr>
<td>-----------</td>
<td>---------------------------</td>
<td>------------------------------------</td>
<td>-------</td>
<td>----------</td>
</tr>
<tr>
<td>CBZ</td>
<td>Rn, label</td>
<td>Compare and branch if zero</td>
<td>-</td>
<td>76</td>
</tr>
<tr>
<td>CLREX</td>
<td>-</td>
<td>Clear exclusive</td>
<td>-</td>
<td>41</td>
</tr>
<tr>
<td>CLZ</td>
<td>Rd, Rm</td>
<td>Count leading zeros</td>
<td>-</td>
<td>50</td>
</tr>
<tr>
<td>CMN</td>
<td>Rn, Op2</td>
<td>Compare negative</td>
<td>N,Z,C,V</td>
<td>51</td>
</tr>
<tr>
<td>CMP</td>
<td>Rn, Op2</td>
<td>Compare</td>
<td>N,Z,C,V</td>
<td>51</td>
</tr>
<tr>
<td>CPSID</td>
<td>iflags</td>
<td>Change processor state, disable interrupts</td>
<td>-</td>
<td>84</td>
</tr>
<tr>
<td>CPSIE</td>
<td>iflags</td>
<td>Change processor state, enable interrupts</td>
<td>-</td>
<td>84</td>
</tr>
<tr>
<td>DMB</td>
<td>-</td>
<td>Data memory barrier</td>
<td>-</td>
<td>85</td>
</tr>
<tr>
<td>DSB</td>
<td>-</td>
<td>Data synchronization barrier</td>
<td>N,Z,C</td>
<td>43</td>
</tr>
<tr>
<td>EOR, EORS</td>
<td>[Rd,] Rn, Op2</td>
<td>Exclusive OR</td>
<td>N,Z,C</td>
<td>43</td>
</tr>
<tr>
<td>ISB</td>
<td>-</td>
<td>Instruction synchronization barrier</td>
<td>-</td>
<td>87</td>
</tr>
<tr>
<td>IT</td>
<td>-</td>
<td>If-Then condition block</td>
<td>-</td>
<td>77</td>
</tr>
<tr>
<td>LDM</td>
<td>Rn(!), reglist</td>
<td>Load multiple registers, increment after</td>
<td>-</td>
<td>35</td>
</tr>
<tr>
<td>LDMDB, LDMEA</td>
<td>Rn(!), reglist</td>
<td>Load multiple registers, decrement before</td>
<td>-</td>
<td>35</td>
</tr>
<tr>
<td>LDMFD, LDMIA</td>
<td>Rn(!), reglist</td>
<td>Load multiple registers, increment after</td>
<td>-</td>
<td>35</td>
</tr>
<tr>
<td>LDR</td>
<td>Rt, [Rn{, #offset}]</td>
<td>Load register with word</td>
<td>-</td>
<td>26</td>
</tr>
<tr>
<td>LDRB, LDRBT</td>
<td>Rt, [Rn{, #offset}]</td>
<td>Load register with byte</td>
<td>-</td>
<td>26</td>
</tr>
<tr>
<td>LDRD</td>
<td>Rt, Rt2, [Rn{, #offset}]</td>
<td>Load register with two words</td>
<td>-</td>
<td>26</td>
</tr>
<tr>
<td>LDREX</td>
<td>Rt, [Rn, #offset]</td>
<td>Load register exclusive</td>
<td>-</td>
<td>39</td>
</tr>
<tr>
<td>LDREXB</td>
<td>Rt, [Rn]</td>
<td>Load register exclusive with byte</td>
<td>-</td>
<td>39</td>
</tr>
<tr>
<td>LDREXH</td>
<td>Rt, [Rn{, #offset}]</td>
<td>Load register exclusive with halfword</td>
<td>-</td>
<td>39</td>
</tr>
<tr>
<td>LDRH, LDRHT</td>
<td>Rt, [Rn{, #offset}]</td>
<td>Load register with halfword</td>
<td>-</td>
<td>26</td>
</tr>
<tr>
<td>LDRSB, LDRSBT</td>
<td>Rt, [Rn{, #offset}]</td>
<td>Load register with signed byte</td>
<td>-</td>
<td>26</td>
</tr>
<tr>
<td>LDRSH, LDRSHT</td>
<td>Rt, [Rn{, #offset}]</td>
<td>Load register with signed halfword</td>
<td>-</td>
<td>26</td>
</tr>
<tr>
<td>LDRT</td>
<td>Rt, [Rn{, #offset}]</td>
<td>Load register with word</td>
<td>-</td>
<td>31</td>
</tr>
<tr>
<td>LSL, LSLS</td>
<td>Rd, Rm, &lt;Rs</td>
<td>#n&gt;</td>
<td>Logical shift left</td>
<td>N,Z,C</td>
</tr>
<tr>
<td>LSR, LSRS</td>
<td>Rd, Rm, &lt;Rs</td>
<td>#n&gt;</td>
<td>Logical shift right</td>
<td>N,Z,C</td>
</tr>
<tr>
<td>MLA</td>
<td>Rd, Rn, Rm, Ra</td>
<td>Multiply with accumulate, 32-bit result</td>
<td>-</td>
<td>60</td>
</tr>
<tr>
<td>MLS</td>
<td>Rd, Rn, Rm, Ra</td>
<td>Multiply and subtract, 32-bit result</td>
<td>-</td>
<td>60</td>
</tr>
<tr>
<td>MOV, MOVS</td>
<td>Rd, Op2</td>
<td>Move</td>
<td>N,Z,C</td>
<td>52</td>
</tr>
<tr>
<td>MOV, MOVW</td>
<td>Rd, #imm16</td>
<td>Move 16-bit constant</td>
<td>N,Z,C</td>
<td>52</td>
</tr>
<tr>
<td>MOVT</td>
<td>Rd, #imm16</td>
<td>Move top</td>
<td>-</td>
<td>54</td>
</tr>
<tr>
<td>MRS</td>
<td>Rd, spec_reg</td>
<td>Move from special register to general register</td>
<td>-</td>
<td>88</td>
</tr>
<tr>
<td>MSR</td>
<td>spec_reg, Rn</td>
<td>Move from general register to special register</td>
<td>N,Z,C,V</td>
<td>89</td>
</tr>
<tr>
<td>MUL, MULS</td>
<td>[Rd,] Rn, Rm</td>
<td>Multiply, 32-bit result</td>
<td>N,Z</td>
<td>60</td>
</tr>
<tr>
<td>MVN, MVNS</td>
<td>Rd, Op2</td>
<td>Move NOT</td>
<td>N,Z,C</td>
<td>52</td>
</tr>
<tr>
<td>NOP</td>
<td>-</td>
<td>No operation</td>
<td>-</td>
<td>90</td>
</tr>
<tr>
<td>ORN, ORNS</td>
<td>[Rd,] Rn, Op2</td>
<td>Logical OR NOT</td>
<td>N,Z,C</td>
<td>43</td>
</tr>
</tbody>
</table>
Table 1-1. Cortex-M3 Instructions (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operands</th>
<th>Brief Description</th>
<th>Flags</th>
<th>See Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORR, ORRS</td>
<td>{Rd}, Rn, Op2</td>
<td>Logical OR</td>
<td>N,Z,C</td>
<td>43</td>
</tr>
<tr>
<td>POP</td>
<td>reglist</td>
<td>Pop registers from stack</td>
<td>-</td>
<td>37</td>
</tr>
<tr>
<td>PUSH</td>
<td>reglist</td>
<td>Push registers onto stack</td>
<td>-</td>
<td>37</td>
</tr>
<tr>
<td>RBIT</td>
<td>Rd, Rn</td>
<td>Reverse bits</td>
<td>-</td>
<td>55</td>
</tr>
<tr>
<td>REV</td>
<td>Rd, Rn</td>
<td>Reverse byte order in a word</td>
<td>-</td>
<td>55</td>
</tr>
<tr>
<td>REV16</td>
<td>Rd, Rn</td>
<td>Reverse byte order in each halfword</td>
<td>-</td>
<td>55</td>
</tr>
<tr>
<td>REVSH</td>
<td>Rd, Rn</td>
<td>Reverse byte order in bottom halfword and sign extend</td>
<td>-</td>
<td>55</td>
</tr>
<tr>
<td>ROR, RORS</td>
<td>Rd, Rm, &lt;Rs</td>
<td>#n&gt;</td>
<td>Rotate right</td>
<td>N,Z,C</td>
</tr>
<tr>
<td>RXX, RRXS</td>
<td>Rd, Rm</td>
<td>Rotate right with extend</td>
<td>N,Z,C</td>
<td>48</td>
</tr>
<tr>
<td>RSB, RSBS</td>
<td>{Rd}, Rn, Op2</td>
<td>Reverse subtract</td>
<td>N,Z,C,V</td>
<td>43</td>
</tr>
<tr>
<td>SBC, SBCS</td>
<td>{Rd}, Rn, Op2</td>
<td>Subtract with carry</td>
<td>N,Z,C,V</td>
<td>43</td>
</tr>
<tr>
<td>SBFX</td>
<td>Rd, Rn, #lsb, #width</td>
<td>Signed bit field extract</td>
<td>-</td>
<td>70</td>
</tr>
<tr>
<td>SDIV</td>
<td>{Rd}, Rn, Rm</td>
<td>Signed divide</td>
<td>-</td>
<td>64</td>
</tr>
<tr>
<td>SEV</td>
<td>-</td>
<td>Send event</td>
<td>-</td>
<td>91</td>
</tr>
<tr>
<td>SMLAL</td>
<td>RdLo, RdHi, Rn, Rm</td>
<td>Signed multiply with accumulate (32x32+64), 64-bit result</td>
<td>-</td>
<td>62</td>
</tr>
<tr>
<td>SMLULL</td>
<td>RdLo, RdHi, Rn, Rm</td>
<td>Signed multiply (32x32), 64-bit result</td>
<td>-</td>
<td>62</td>
</tr>
<tr>
<td>SSAT</td>
<td>Rd, #n, Rm, (#shift #s)</td>
<td>Signed saturate</td>
<td>Q</td>
<td>66</td>
</tr>
<tr>
<td>STM</td>
<td>Rn!!, reglist</td>
<td>Store multiple registers, increment after</td>
<td>-</td>
<td>35</td>
</tr>
<tr>
<td>STMDB, STMIA</td>
<td>Rn!!, reglist</td>
<td>Store multiple registers, decrement before</td>
<td>-</td>
<td>35</td>
</tr>
<tr>
<td>STMFD, STMIA</td>
<td>Rn!!, reglist</td>
<td>Store multiple registers, increment after</td>
<td>-</td>
<td>35</td>
</tr>
<tr>
<td>STR</td>
<td>Rt, [Rn{, #offset}]</td>
<td>Store register word</td>
<td>-</td>
<td>26</td>
</tr>
<tr>
<td>STRB, STRBT</td>
<td>Rt, [Rn{, #offset}]</td>
<td>Store register byte</td>
<td>-</td>
<td>26</td>
</tr>
<tr>
<td>STRD</td>
<td>Rt, Rt2, [Rn{, #offset}]</td>
<td>Store register two words</td>
<td>-</td>
<td>26</td>
</tr>
<tr>
<td>STREX</td>
<td>Rd, Rt, [Rn{, #offset}]</td>
<td>Store register exclusive</td>
<td>-</td>
<td>39</td>
</tr>
<tr>
<td>STREXB</td>
<td>Rd, Rt, [Rn{, #offset}]</td>
<td>Store register exclusive byte</td>
<td>-</td>
<td>39</td>
</tr>
<tr>
<td>STREXH</td>
<td>Rd, Rt, [Rn{, #offset}]</td>
<td>Store register exclusive halfword</td>
<td>-</td>
<td>39</td>
</tr>
<tr>
<td>STRH, STRHT</td>
<td>Rt, [Rn{, #offset}]</td>
<td>Store register halfword</td>
<td>-</td>
<td>26</td>
</tr>
<tr>
<td>STRSB, STRSBT</td>
<td>Rt, [Rn{, #offset}]</td>
<td>Store register signed byte</td>
<td>-</td>
<td>26</td>
</tr>
<tr>
<td>STRSH, STRSHT</td>
<td>Rt, [Rn{, #offset}]</td>
<td>Store register signed halfword</td>
<td>-</td>
<td>26</td>
</tr>
<tr>
<td>STRT</td>
<td>Rt, [Rn{, #offset}]</td>
<td>Store register word</td>
<td>-</td>
<td>31</td>
</tr>
<tr>
<td>SUB, SUBS</td>
<td>{Rd}, Rn, Op2</td>
<td>Subtract</td>
<td>N,Z,C,V</td>
<td>43</td>
</tr>
<tr>
<td>SUB, SUBW</td>
<td>{Rd}, Rn, #imm12</td>
<td>Subtract 12-bit constant</td>
<td>N,Z,C,V</td>
<td>43</td>
</tr>
<tr>
<td>SVC</td>
<td>#imm</td>
<td>Supervisor call</td>
<td>-</td>
<td>92</td>
</tr>
<tr>
<td>SXTB</td>
<td>{Rd}, Rm ((, ROR #n)</td>
<td>Sign extend a byte</td>
<td>-</td>
<td>71</td>
</tr>
<tr>
<td>SXTH</td>
<td>{Rd}, Rm ((, ROR #n)</td>
<td>Sign extend a halfword</td>
<td>-</td>
<td>71</td>
</tr>
<tr>
<td>TBB</td>
<td>[Rn, Rm]</td>
<td>Table branch byte</td>
<td>-</td>
<td>80</td>
</tr>
<tr>
<td>TBH</td>
<td>[Rn, Rm, LSL #1]</td>
<td>Table branch halfword</td>
<td>-</td>
<td>80</td>
</tr>
<tr>
<td>TEQ</td>
<td>Rn, Op2</td>
<td>Test equivalence</td>
<td>N,Z,C</td>
<td>57</td>
</tr>
<tr>
<td>TST</td>
<td>Rn, Op2</td>
<td>Test</td>
<td>N,Z,C</td>
<td>57</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Operands</td>
<td>Brief Description</td>
<td>Flags</td>
<td>See Page</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
<td>-------------------------------------------------------</td>
<td>-------</td>
<td>----------</td>
</tr>
<tr>
<td>UBFX</td>
<td>Rd, Rn, #lsb, #width</td>
<td>Unsigned bit field extract</td>
<td>-</td>
<td>70</td>
</tr>
<tr>
<td>UDIV</td>
<td>{Rd,} Rn, Rm</td>
<td>Unsigned divide</td>
<td>-</td>
<td>64</td>
</tr>
<tr>
<td>UMLAL</td>
<td>RdLo, RdHi, Rn, Rm</td>
<td>Unsigned multiply with accumulate (32x32+64), 64-bit result</td>
<td>-</td>
<td>62</td>
</tr>
<tr>
<td>UMULL</td>
<td>RdLo, RdHi, Rn, Rm</td>
<td>Unsigned multiply (32x2), 64-bit result</td>
<td>-</td>
<td>62</td>
</tr>
<tr>
<td>USAT</td>
<td>Rd, #n, Rm (,shift #s)</td>
<td>Unsigned saturate</td>
<td>Q</td>
<td>66</td>
</tr>
<tr>
<td>UXTB</td>
<td>{Rd,} Rm (,ROR #n)</td>
<td>Zero extend a byte</td>
<td>-</td>
<td>71</td>
</tr>
<tr>
<td>UXTH</td>
<td>{Rd,} Rm (,ROR #n)</td>
<td>Zero extend a halfword</td>
<td>-</td>
<td>71</td>
</tr>
<tr>
<td>WFE</td>
<td>-</td>
<td>Wait for event</td>
<td>-</td>
<td>93</td>
</tr>
<tr>
<td>WFI</td>
<td>-</td>
<td>Wait for interrupt</td>
<td>-</td>
<td>94</td>
</tr>
</tbody>
</table>

### 1.2 About The Instruction Descriptions

The following sections give more information about using the instructions:

- “Operands” on page 15
- “Restrictions When Using the PC or SP” on page 15
- “Flexible Second Operand” on page 15
- “Shift Operations” on page 17
- “Address Alignment” on page 20
- “PC-Relative Expressions” on page 20
- “Conditional Execution” on page 20
- “Instruction Width Selection” on page 22

#### 1.2.1 Operands

An instruction operand can be an ARM Cortex-M3 register, a constant, or another instruction-specific parameter. Instructions act on the operands and often store the result in a destination register. When there is a destination register in the instruction, it is usually specified before the operands.

Operands in some instructions are flexible in that they can either be a register or a constant. See “Flexible Second Operand” on page 15.

See the Stellaris® Data Sheet for more information on the ARM Cortex-M3 registers.

#### 1.2.2 Restrictions When Using the PC or SP

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

**Important:** Bit[0] of any address you write to the PC with a BX, BLX, LDM, LDR, or POP instruction must be 1 for correct execution, because this bit indicates the required instruction set, and the Cortex-M3 processor only supports Thumb instructions.

#### 1.2.3 Flexible Second Operand

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

*Operand2* can be a constant or a register with optional shift.
1.2.3.1 Constant
You specify an Operand2 constant in the form:

\[ \#\text{constant} \]

where \textit{constant} can be (X and Y are hexadecimal digits):

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

In addition, in a small number of instructions, \textit{constant} can take a wider range of values. These are described in the individual instruction descriptions.

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

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

1.2.3.2 Register With Optional Shift
You specify an Operand2 register in the form:

\[ Rm \{, \text{shift} \} \]

where:

- \textit{Rm} is the register holding the data for the second operand.
- \textit{shift} is an optional shift to be applied to \textit{Rm}. It can be one of:
  - \textit{ASR} \#n
    Arithmetic shift right \( n \) bits, \( 1 \leq n \leq 32 \).
  - \textit{LSL} \#n
    Logical shift left \( n \) bits, \( 1 \leq n \leq 31 \).
  - \textit{LSR} \#n
    Logical shift right \( n \) bits, \( 1 \leq n \leq 32 \).
  - \textit{ROR} \#n
    Rotate right \( n \) bits, \( 1 \leq n \leq 31 \).
  - \textit{RRX}
    Rotate right one bit, with extend.
If omitted, no shift occurs; equivalent to \textit{LSL} \#0.

If you omit the shift, or specify \textit{LSL} \#0, the instruction uses the value in \textit{Rm}.

If you specify a shift, the shift is applied to the value in \textit{Rm}, and the resulting 32-bit value is used by the instruction. However, the contents in the register \textit{Rm} remain unchanged. Specifying a register with shift also updates the carry flag when used with certain instructions. For information on the shift operations and how they affect the carry flag, see “Shift Operations” on page 17.

1.2.4 Shift Operations

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

- Directly by the instructions \textit{ASR}, \textit{LSR}, \textit{LSL}, \textit{ROR}, and \textit{RRX}, and the result is written to a destination register.

- During the calculation of \textit{Operand2} by the instructions that specify the second operand as a register with shift (see “Flexible Second Operand” on page 15). The result is used by the instruction.

The permitted shift lengths depend on the shift type and the instruction (see the individual instruction description or see “Flexible Second Operand” on page 15). If the shift length is 0, no shift occurs. Register shift operations update the carry flag except when the specified shift length is 0. The following sub-sections describe the various shift operations and how they affect the carry flag. In these descriptions, \textit{Rm} is the register containing the value to be shifted, and \textit{n} is the shift length.

1.2.4.1 ASR

An arithmetic shift right (\textit{ASR}) by \textit{n} bits moves the left-hand \(32-n\) bits of the register \textit{Rm}, to the right by \textit{n} places, into the right-hand \(32-n\) bits of the result. And it copies the original bit[31] of the register into the left-hand \textit{n} bits of the result. See Figure 1-1 on page 17.

You can use the \textit{ASR} \#n operation to divide the value in the register \textit{Rm} by \(2^n\), with the result being rounded towards negative-infinity.

When the instruction is \textit{ASRS} or when \textit{ASR} \#n is used in \textit{Operand2} with the instructions \textit{MOVX}, \textit{MVNS}, \textit{ANDS}, \textit{ORRS}, \textit{ORNS}, \textit{EORS}, \textit{BICS}, \textit{TEQ} or \textit{TST}, the carry flag is updated to the last bit shifted out, bit[\textit{n-1}], of the register \textit{Rm}.

\textbf{Note:}
- If \textit{n} is 32 or more, then all the bits in the result are set to the value of bit[31] of \textit{Rm}.
- If \textit{n} is 32 or more and the carry flag is updated, it is updated to the value of bit[31] of \textit{Rm}.

\textbf{Figure 1-1. ASR #3}

\[\text{Diagram of ASR #3}\]
1.2.4.2 LSR

A logical shift right (LSR) by n bits moves the left-hand 32\(\cdot n\) bits of the register \(Rm\), to the right by \(n\) places, into the right-hand 32\(\cdot n\) bits of the result. And it sets the left-hand \(n\) bits of the result to 0. See Figure 1-2 on page 18.

You can use the \(LSR \ #n\) operation to divide the value in the register \(Rm\) by \(2^n\), if the value is regarded as an unsigned integer.

When the instruction is \(LSRS\) or when \(LSR \ #n\) is used in \(Operand2\) with the instructions \(MOV\), \(MVNS\), \(ANDS\), \(ORRS\), \(ORNS\), \(EORS\), \(BICS\), \(TEQ\) or \(TST\), the carry flag is updated to the last bit shifted out, bit\([n-1]\), of the register \(Rm\).

**Note:**
- If \(n\) is 32 or more, then all the bits in the result are cleared to 0.
- If \(n\) is 33 or more and the carry flag is updated, it is updated to 0.

![Figure 1-2. LSR #3](image)

1.2.4.3 LSL

A logical shift left (LSL) by \(n\) bits moves the right-hand 32\(\cdot n\) bits of the register \(Rm\), to the left by \(n\) places, into the left-hand 32\(\cdot n\) bits of the result. And it sets the right-hand \(n\) bits of the result to 0. See Figure 1-3 on page 19.

You can use the \(LSL \ #n\) operation to multiply the value in the register \(Rm\) by \(2^n\), if the value is regarded as an unsigned integer or a two's complement signed integer. Overflow can occur without warning.

When the instruction is \(LSLS\) or when \(LSL \ #n\), with non-zero \(n\), is used in \(Operand2\) with the instructions \(MOV\), \(MVNS\), \(ANDS\), \(ORRS\), \(ORNS\), \(EORS\), \(BICS\), \(TEQ\) or \(TST\), the carry flag is updated to the last bit shifted out, bit\([32\cdot n]\), of the register \(Rm\). These instructions do not affect the carry flag when used with \(LSL \ #0\).

**Note:**
- If \(n\) is 32 or more, then all the bits in the result are cleared to 0.
- If \(n\) is 33 or more and the carry flag is updated, it is updated to 0.
1.2.4.4 ROR

A rotate right (ROR) by "n" bits moves the left-hand 32\(\times\)n bits of the register \(R_m\) to the right by "n" places, into the right-hand 32\(\times\)n bits of the result. And it moves the right-hand "n" bits of the register into the left-hand "n" bits of the result. See Figure 1-4 on page 19.

When the instruction is RORS or when ROR \#n is used in Operand2 with the instructions MOVs, MVNS, ANDs, ORRs, ORNs, EORS, BICS, TEQ or TST, the carry flag is updated to the last bit rotation, bit[\(n-1\)], of the register \(R_m\).

Note:  
- If "n" is 32, then the value of the result is the same as the value in \(R_m\), and if the carry flag is updated, it is updated to bit[31] of \(R_m\).
- ROR with shift length, "n", more than 32 is the same as ROR with shift length \(n\leq32\).

1.2.4.5 RRX

A rotate right with extend (RRX) moves the bits of the register \(R_m\) to the right by one bit. And it copies the carry flag into bit[31] of the result. See Figure 1-5 on page 19.

When the instruction is RRXS or when RRX is used in Operand2 with the instructions MOVs, MVNS, ANDs, ORRs, ORNs, EORS, BICS, TEQ or TST, the carry flag is updated to bit[0] of the register \(R_m\).
1.2.5 Address Alignment

An aligned access is an operation where a word-aligned address is used for a word, dual word, or multiple word access, or where a halfword-aligned address is used for a halfword access. Byte accesses are always aligned.

The Cortex-M3 processor supports unaligned access only for the following instructions:

- LDR, LDRT
- LDRH, LDRHT
- LDRSH, LDRSHT
- STR, STRT
- STRH, STRHT

All other load and store instructions generate a usage fault exception if they perform an unaligned access, and therefore their accesses must be address aligned. For more information about usage faults, see "Fault Handling" in the Stellaris® Data Sheet.

Unaligned accesses are usually slower than aligned accesses. In addition, some memory regions might not support unaligned accesses. Therefore, ARM recommends that programmers ensure that accesses are aligned. To avoid accidental generation of unaligned accesses, use the UNALIGNED bit in the Configuration and Control (CFGCTRL) register to trap all unaligned accesses (see CFGCTRL in the Stellaris® Data Sheet).

1.2.6 PC-Relative Expressions

A PC-relative expression or label is a symbol that represents the address of an instruction or literal data. It is represented in the instruction as the PC value plus or 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.

Note: For B, BL, CBNZ, and CBZ instructions, the value of the PC is the address of the current instruction plus 4 bytes.

- For all other instructions that use labels, the value of the PC is the address of the current instruction plus 4 bytes, with bit[1] of the result cleared to 0 to make it word-aligned.

- Your assembler might permit other syntaxes for PC-relative expressions, such as a label plus or minus a number, or an expression of the form [PC, #number].

1.2.7 Conditional Execution

Most data processing instructions can optionally update the condition flags in the Application Program Status Register (APSR) register according to the result of the operation (see APSR in the Stellaris® Data Sheet). Some instructions update all flags, and some only update a subset. If a flag is not updated, the original value is preserved. See the instruction descriptions for the flags they affect.

You can execute an instruction conditionally, based on the condition flags set in another instruction, either immediately after the instruction that updated the flags, or after any number of intervening instructions that have not updated the flags.

Conditional execution is available by using conditional branches or by adding condition code suffixes to instructions. See Table 1-2 on page 22 for a list of the suffixes to add to instructions to make them conditional instructions. The condition code suffix enables the processor to test a condition based on the flags. If the condition test of a conditional instruction fails, the instruction:
Does not execute

Does not write any value to its destination register

Does not affect any of the flags

Does not generate any exception

Conditional instructions, except for conditional branches, must be inside an If-Then instruction block. See "IT" on page 77 for more information and restrictions when using the IT instruction. Depending on the vendor, the assembler might automatically insert an IT instruction if you have conditional instructions outside the IT block. See “IT” on page 77 for more on the IT block.

Use the CBZ and CBNZ instructions to compare the value of a register against zero and branch on the result.

1.2.7.1 Condition Flags

The Application Program Status Register (APSR) contains the following condition flags:

- **N.** Set to 1 when the result of the operation was negative; cleared to 0 otherwise.
- **Z.** Set to 1 when the result of the operation was zero; cleared to 0 otherwise.
- **C.** Set to 1 when the operation resulted in a carry; cleared to 0 otherwise.
- **V.** Set to 1 when the operation caused overflow; cleared to 0 otherwise.

For more information about APSR, see the Stellaris® Data Sheet.

A carry occurs:

- If the result of an addition is greater than or equal to \(2^{32}\)
- If the result of a subtraction is positive or zero
- As the result of an inline barrel shifter operation in a move or logical instruction

Overflow occurs if the result of an add, subtract, or compare is greater than or equal to \(2^{31}\), or less than \(-2^{31}\).

**Note:** Most instructions update the status flags only if the S suffix is specified. See the instruction descriptions for more information.

1.2.7.2 Condition Code Suffixes

The instructions that can be conditional have an optional condition code, shown in syntax descriptions as \(\{\text{cond}\}\). Conditional execution requires a preceding IT instruction. An instruction with a condition code is only executed if the condition code flags in APSR meet the specified condition. Table 1-2 on page 22 shows the condition codes to use.

You can use conditional execution with the IT instruction to reduce the number of branch instructions in code.

Table 1-2 on page 22 also shows the relationship between condition code suffixes and the N, Z, C, and V flags.
### Table 1-2. Condition Code Suffixes

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Flags</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>$Z = 1$</td>
<td>Equal</td>
</tr>
<tr>
<td>NE</td>
<td>$Z = 0$</td>
<td>Not equal</td>
</tr>
<tr>
<td>CS or HS</td>
<td>$C = 1$</td>
<td>Higher or same, unsigned $\geq$</td>
</tr>
<tr>
<td>CC or LO</td>
<td>$C = 0$</td>
<td>Lower, unsigned $&lt;$</td>
</tr>
<tr>
<td>MI</td>
<td>$N = 1$</td>
<td>Negative</td>
</tr>
<tr>
<td>PL</td>
<td>$N = 0$</td>
<td>Positive or zero</td>
</tr>
<tr>
<td>VS</td>
<td>$V = 1$</td>
<td>Overflow</td>
</tr>
<tr>
<td>VC</td>
<td>$V = 0$</td>
<td>No overflow</td>
</tr>
<tr>
<td>HI</td>
<td>$C = 1$ and $Z = 0$</td>
<td>Higher, unsigned $&gt;$</td>
</tr>
<tr>
<td>LS</td>
<td>$C = 0$ or $Z = 1$</td>
<td>Lower or same, unsigned $\leq$</td>
</tr>
<tr>
<td>GE</td>
<td>$N = V$</td>
<td>Greater than or equal, signed $\geq$</td>
</tr>
<tr>
<td>LT</td>
<td>$N ! = V$</td>
<td>Less than, signed $&lt;$</td>
</tr>
<tr>
<td>GT</td>
<td>$Z = 0$ and $N = V$</td>
<td>Greater than, signed $&gt;$</td>
</tr>
<tr>
<td>LE</td>
<td>$Z = 1$ and $N ! = V$</td>
<td>Less than or equal, signed $\leq$</td>
</tr>
<tr>
<td>AL</td>
<td>Can have any value</td>
<td>Always. This is the default when no suffix is specified.</td>
</tr>
</tbody>
</table>

Example 1-1, “Absolute Value” on page 22 shows the use of a conditional instruction to find the absolute value of a number. $R0 = \text{ABS}(R1)$.

**Example 1-1. Absolute Value**

```
MOVS R0, R1 ; R0 = R1, setting flags.
IT MI ; IT instruction for the negative condition.
RSBMI R0, R1, #0 ; If negative, R0 = -R1.
```

Example 1-2, “Compare and Update Value” on page 22 shows the use of conditional instructions to update the value of $R4$ if the signed value $R0$ is greater than $R1$ and $R2$ is greater than $R3$.

**Example 1-2. Compare and Update Value**

```
CMP R0, R1 ; Compare R0 and R1, setting flags
ITT GT ; IT instruction for the two GT conditions
CMPPGT R2, R3 ; If 'greater than', compare R2 and R3, setting flags
MOVGT R4, R5 ; If still 'greater than', do R4 = R5
```

### 1.2.8 Instruction Width Selection

There are many instructions that can generate either a 16-bit encoding or a 32-bit encoding depending on the operands and destination register specified. For some of these instructions, you can force a specific instruction size by using an instruction width suffix. The .W suffix forces a 32-bit instruction encoding. The .N suffix forces a 16-bit instruction encoding.

If you specify an instruction width suffix and the assembler cannot generate an instruction encoding of the requested width, it generates an error.

**Note:** In some cases it might be necessary to specify the .W suffix, for example if the operand is the label of an instruction or literal data, as in the case of branch instructions. This is because the assembler might not automatically generate the right size encoding.
To use an instruction width suffix, place it immediately after the instruction mnemonic and condition code, if any. Example 1-3, “Instruction Width Selection” on page 23 shows instructions with the instruction width suffix.

Example 1-3. Instruction Width Selection

BCS.W label ; creates a 32-bit instruction even for a short branch

ADDS.W R0, R0, R1 ; creates a 32-bit instruction even though the same ; operation can be done by a 16-bit instruction
2 Memory Access Instructions

Table 2-1 on page 24 shows the memory access instructions:

Table 2-1. Memory Access Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Brief Description</th>
<th>See Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADR</td>
<td>Load PC-relative address</td>
<td>25</td>
</tr>
<tr>
<td>CLR EX</td>
<td>Clear exclusive</td>
<td>41</td>
</tr>
<tr>
<td>LDM{mode}</td>
<td>Load multiple registers</td>
<td>35</td>
</tr>
<tr>
<td>LDR{type}</td>
<td>Load register using immediate offset</td>
<td>26</td>
</tr>
<tr>
<td>LDR{type}</td>
<td>Load register using register offset</td>
<td>29</td>
</tr>
<tr>
<td>LDR{type}T</td>
<td>Load register with unprivileged access</td>
<td>31</td>
</tr>
<tr>
<td>LDR{type}</td>
<td>Load register using PC-relative address</td>
<td>33</td>
</tr>
<tr>
<td>LDRD</td>
<td>Load register using PC-relative address (two words)</td>
<td>33</td>
</tr>
<tr>
<td>LDREX{type}</td>
<td>Load register exclusive</td>
<td>39</td>
</tr>
<tr>
<td>POP</td>
<td>Pop registers from stack</td>
<td>37</td>
</tr>
<tr>
<td>PUSH</td>
<td>Push registers onto stack</td>
<td>37</td>
</tr>
<tr>
<td>STM{mode}</td>
<td>Store multiple registers</td>
<td>35</td>
</tr>
<tr>
<td>STR{type}</td>
<td>Store register using immediate offset</td>
<td>26</td>
</tr>
<tr>
<td>STR{type}</td>
<td>Store register using register offset</td>
<td>35</td>
</tr>
<tr>
<td>STR{type}T</td>
<td>Store register with unprivileged access</td>
<td>31</td>
</tr>
<tr>
<td>STREX{type}</td>
<td>Store register exclusive</td>
<td>39</td>
</tr>
</tbody>
</table>
2.1 ADR
Load PC-relative address.

2.1.1 Syntax
ADR(cond) Rd, label

where:
cond
Is an optional condition code. See Table 1-2 on page 22.

Rd
Is the destination register.

label
Is a PC-relative expression. See “PC-Relative Expressions” on page 20.

2.1.2 Operation
ADR determines the address by adding an immediate value to the PC, and writes the result to the
destination register.
ADR produces position-independent code, because the address is PC-relative.

If you use ADR to generate a target address for a BX or BLX instruction, you must ensure that bit[0]
of the address you generate is set to 1 for correct execution.

Values of label must be within the range of −4095 to +4095 from the address in the PC.

Note: You might have to use the .W suffix to get the maximum offset range or to generate
addresses that are not word-aligned. See “Instruction Width Selection” on page 22.

2.1.3 Restrictions
Rd must not be SP and must not be PC.

2.1.4 Condition Flags
This instruction does not change the flags.

2.1.5 Examples
ADR R1, TextMessage ; Write address value of a location labeled as
                     ; TextMessage to R1.
2.2 LDR and STR (Immediate Offset)

Load and Store with immediate offset, pre-indexed immediate offset, or post-indexed immediate offset.

2.2.1 Syntax

\[ \text{op} \{ \text{type} \} \{ \text{cond} \} \text{ Rt}, [\text{Rn }, \#\text{offset}] \] ; immediate offset

\[ \text{op} \{ \text{type} \} \{ \text{cond} \} \text{ Rt}, [\text{Rn}, \#\text{offset}]! \] ; pre-indexed

\[ \text{op} \{ \text{type} \} \{ \text{cond} \} \text{ Rt}, [\text{Rn}], \#\text{offset} \] ; post-indexed

\[ \text{opD} \{ \text{cond} \} \text{ Rt}, \text{Rt2}, [\text{Rn }, \#\text{offset}] \] ; immediate offset, two words

\[ \text{opD} \{ \text{cond} \} \text{ Rt}, \text{Rt2}, [\text{Rn}, \#\text{offset}]! \] ; pre-indexed, two words

\[ \text{opD} \{ \text{cond} \} \text{ Rt}, \text{Rt2}, [\text{Rn}], \#\text{offset} \] ; post-indexed, two words

where:

\text{op} \text{ is one of:}

- \text{LDR} \hspace{1cm} \text{Load Register.}
- \text{STR} \hspace{1cm} \text{Store Register.}

\text{type} \text{ is one of:}

- \text{B} \hspace{1cm} \text{Unsigned byte, zero extend to 32 bits on loads.}
- \text{SB} \hspace{1cm} \text{Signed byte, sign extend to 32 bits (LDR only).}
- \text{H} \hspace{1cm} \text{Unsigned halfword, zero extend to 32 bits on loads.}
- \text{SH} \hspace{1cm} \text{Signed halfword, sign extend to 32 bits (LDR only).}
- \text{—} \hspace{1cm} \text{Omit, for word.}

\text{cond} \text{ is an optional condition code. See Table 1-2 on page 22.}

\text{Rt} \text{ is the register to load or store.}
Rn
Is the register on which the memory address is based.

offset
Is an offset from Rn. If offset is omitted, the address is the contents of Rn.

Rt2
Is the additional register to load or store for two-word operations.

2.2.2 Operation

LDR instructions load one or two registers with a value from memory.

STR instructions store one or two register values to memory.

Load and store instructions with immediate offset can use the following addressing modes:

Offset addressing
The offset value is added to or subtracted from the address obtained from the register Rn. The result is used as the address for the memory access. The register Rn is unaltered. The assembly language syntax for this mode is:

[Rn, #offset]

Pre-indexed addressing
The offset value is added to or subtracted from the address obtained from the register Rn. The result is used as the address for the memory access and written back into the register Rn. The assembly language syntax for this mode is:

[Rn, #offset]!

Post-indexed addressing
The address obtained from the register Rn is used as the address for the memory access. The offset value is added to or subtracted from the address, and written back into the register Rn. The assembly language syntax for this mode is:

[Rn], #offset

The value to load or store can be a byte, halfword, word, or two words. Bytes and halfwords can either be signed or unsigned. See “Address Alignment” on page 20.

Table 2-2 on page 27 shows the ranges of offset for immediate, pre-indexed and post-indexed forms.

Table 2-2. Offset Ranges

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Immediate Offset</th>
<th>Pre-Indexed</th>
<th>Post-Indexed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word, halfword, signed halfword, byte, or signed byte</td>
<td>-255 to 4095</td>
<td>-255 to 255</td>
<td>-255 to 255</td>
</tr>
<tr>
<td>Two words</td>
<td>Multiple of 4 in the range -1020 to 1020</td>
<td>Multiple of 4 in the range -1020 to 1020</td>
<td>Multiple of 4 in the range -1020 to 1020</td>
</tr>
</tbody>
</table>

2.2.3 Restrictions

For load instructions:

- Rt can be SP or PC for word loads only.
Memory Access Instructions

- **Rt** must be different from **Rt2** for two-word loads.
- **Rn** must be different from **Rt** and **Rt2** in the pre-indexed or post-indexed forms.

When **Rt** is **PC** in a word load instruction:
- Bit[0] of the loaded value must be 1 for correct execution.
- A branch occurs to the address created by changing bit[0] of the loaded value to 0.
- If the instruction is conditional, it must be the last instruction in the IT block.

For store instructions:
- **Rt** can be **SP** for word stores only.
- **Rt** must not be **PC**.
- **Rn** must not be **PC**.
- **Rn** must be different from **Rt** and **Rt2** in the pre-indexed or post-indexed forms.

### 2.2.4 Condition Flags
These instructions do not change the flags.

### 2.2.5 Examples

LDR R8, [R10] ; Loads R8 from the address in R10.
LDRNE R2, [R5, #960]! ; Loads (conditionally) R2 from a word 960 bytes above the address in R5, and increments R5 by 960.
STR R2, [R9, #const-struc] ; `const-struc` is an expression evaluating to a constant in the range 0-4095.
STRH R3, [R4], #4 ; Store R3 as halfword data into address in R4, then increment R4 by 4.
LDRD R8, R9, [R3, #0x20] ; Load R8 from a word 32 bytes above the address in R3, and load R9 from a word 36 bytes above the address in R3.
STRD R0, R1, [R8], #-16 ; Store R0 to address in R8, and store R1 to a word 4 bytes above the address in R8, and then decrement R8 by 16.
2.3 LDR and STR (Register Offset)

Load and Store with register offset.

2.3.1 Syntax

\[ \text{op\{type\}}\{\text{cond}\} \ Rt, [Rn, Rm \{, \text{LSL \#n}\}] \]

where:

- \text{op} is one of:
  - \text{LDR} Load Register.
  - \text{STR} Store Register.

- \text{type} is one of:
  - \text{B} Unsigned byte, zero extend to 32 bits on loads.
  - \text{SB} Signed byte, sign extend to 32 bits (\text{LDR} only).
  - \text{H} Unsigned halfword, zero extend to 32 bits on loads.
  - \text{SH} Signed halfword, sign extend to 32 bits (\text{LDR} only).
  - \text{—} Omit, for word.

- \text{cond} is an optional condition code. See Table 1-2 on page 22.

- \text{Rt} is the register to load or store.

- \text{Rn} is the register on which the memory address is based.

- \text{Rm} is a register containing a value to be used as the offset.

- \text{LSL \#n} is an optional shift, with \text{n} in the range 0 to 3.

2.3.2 Operation

\text{LDR} instructions load a register with a value from memory.
**Memory Access Instructions**

`STR` instructions store a register value into memory.

The memory address to load from or store to is at an offset from the register `Rn`. The offset is specified by the register `Rm` and can be shifted left by up to 3 bits using `LSL`.

The value to load or store can be a byte, halfword, or word. For load instructions, bytes and halfwords can either be signed or unsigned. See "Address Alignment" on page 20.

### 2.3.3 Restrictions

In these instructions:

- `Rn` must not be `PC`.
- `Rm` must not be `SP` and must not be `PC`.
- `Rt` can be `SP` only for word loads and word stores.
- `Rt` can be `PC` only for word loads.

When `Rt` is `PC` in a word load instruction:

- Bit[0] of the loaded value must be 1 for correct execution, and a branch occurs to this halfword-aligned address.
- If the instruction is conditional, it must be the last instruction in the IT block.

### 2.3.4 Condition Flags

These instructions do not change the flags.

### 2.3.5 Examples

```
STR R0, [R5, R1] ; Store value of R0 into an address equal to sum of R5 and R1.
LDRSB R0, [R5, R1, LSL #1] ; Read byte value from an address equal to sum of R5 and two times R1, sign extend it to a word value and put it in R0.
STR R0, [R1, R2, LSL #2] ; Store R0 to an address equal to sum of R1 and four times R2.
```
2.4 LDR and STR (Unprivileged Access)

Load and Store with unprivileged access.

2.4.1 Syntax

\[ \text{op[type]} \{ \text{cond} \} \text{ Rt, [Rn \{, #offset\}]} \] ; immediate offset

where:

- \( \text{op} \) is one of:
  - \( \text{LDR} \) Load Register.
  - \( \text{STR} \) Store Register.

- \( \text{type} \) is one of:
  - \( \text{B} \) Unsigned byte, zero extend to 32 bits on loads.
  - \( \text{SB} \) Signed byte, sign extend to 32 bits (\( \text{LDR} \) only).
  - \( \text{H} \) Unsigned halfword, zero extend to 32 bits on loads.
  - \( \text{SH} \) Signed halfword, sign extend to 32 bits (\( \text{LDR} \) only).
  - \( \_ \) Omit, for word.

- \( \text{cond} \) is an optional condition code. See Table 1-2 on page 22.

- \( \text{Rt} \) is the register to load or store.

- \( \text{Rn} \) is the register on which the memory address is based.

- \( \text{offset} \) is an offset from \( \text{Rn} \) and can be 0 to 255. If \( \text{offset} \) is omitted, the address is the value in \( \text{Rn} \).

2.4.2 Operation

These load and store instructions perform the same function as the memory access instructions with immediate offset (see "LDR and STR (Immediate Offset)" on page 26). The difference is that these instructions have only unprivileged access even when used in privileged software.
When used in unprivileged software, these instructions behave in exactly the same way as normal memory access instructions with immediate offset.

### 2.4.3 Restrictions

In these instructions:

- $R_n$ must not be $PC$.
- $R_t$ must not be $SP$ and must not be $PC$.

### 2.4.4 Condition Flags

These instructions do not change the flags.

### 2.4.5 Examples

```assembly
STRBTEQ R4, [R7] ; Conditionally store least significant byte in R4 to an address in R7, with unprivileged access.
LDRHT R2, [R2, #8] ; Load halfword value from an address equal to sum of R2 and 8 into R2, with unprivileged access.
```
2.5 LDR (PC-Relative)

Load register from memory.

2.5.1 Syntax

\[
\text{LDR}\{\text{type}\}\{\text{cond}\} \ Rt, \ label \\
\text{LDRD}\{\text{cond}\} \ Rt, \ Rt2, \ label \quad ; \text{Load two words}
\]

where:

- \text{type} \\
  Is one of:
  - \text{B} \\
    Unsigned byte, zero extend to 32 bits.
  - \text{SB} \\
    Signed byte, sign extend to 32 bits.
  - \text{H} \\
    Unsigned halfword, zero extend to 32 bits.
  - \text{SH} \\
    Signed halfword, sign extend to 32 bits.
  - - \\
    Omit, for word.

- \text{cond} \\
  Is an optional condition code. See Table 1-2 on page 22.

- \text{Rt} \\
  Is the register to load or store.

- \text{Rt2} \\
  Is the second register to load or store.

- \text{label} \\
  Is a PC-relative expression. See “PC-Relative Expressions” on page 20.

2.5.2 Operation

\text{LDR} \text{ loads a register with a value from a PC-relative memory address. The memory address is specified by a label or by an offset from the PC.}

The value to load or store can be a byte, halfword, or word. For load instructions, bytes and halfwords can either be signed or unsigned. See “Address Alignment” on page 20.

\text{label} \text{ must be within a limited range of the current instruction. Table 2-3 on page 34 shows the possible offsets between label and the PC.}
Table 2-3. Offset Ranges

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Offset Rangea</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word, halfword, signed halfword, byte, signed byte</td>
<td>−4095 to 4095</td>
</tr>
<tr>
<td>Two words</td>
<td>−1020 to 1020</td>
</tr>
</tbody>
</table>

a. You might have to use the .W suffix to get the maximum offset range. See “Instruction Width Selection” on page 22.

2.5.3 Restrictions

In these instructions:

■ $R_t$ can be SP or PC only for word loads.

■ $R_t2$ must not be SP and must not be PC.

■ $R_t$ must be different from $R_t2$.

When $R_t$ is PC in a word load instruction:

■ Bit[0] of the loaded value must be 1 for correct execution, and a branch occurs to this halfword-aligned address.

■ If the instruction is conditional, it must be the last instruction in the IT block.

2.5.4 Condition Flags

These instructions do not change the flags.

2.5.5 Examples

LDR R0, LookUpTable ; Load R0 with a word of data from an address labeled as LookUpTable.

LDRSB R7, localdata ; Load a byte value from an address labeled as localdata, sign extend it to a word value, and put it in R7.
2.6 LDM and STM

Load and Store Multiple registers.

2.6.1 Syntax

\[ op \{ addr\_mode \} \{ cond \} \ Rn\{!\}, \ reglist \]

where:

- \( op \) Is one of:
  - \( LDM \) Load Multiple registers.
  - \( STM \) Store Multiple registers.

- \( addr\_mode \) Is any one of the following:
  - \( IA \) Increment address After each access. This is the default.
  - \( DB \) Decrement address Before each access.

- \( cond \) Is an optional condition code. See Table 1-2 on page 22.

- \( Rn \) Is the register on which the memory addresses are based.

- \( ! \) Is an optional writeback suffix. If \( ! \) is present then the final address, that is loaded from or stored to, is written back into \( Rn \).

- \( reglist \) Is a list of one or more registers to be loaded or stored, enclosed in braces. It can contain register ranges. It must be comma separated if it contains more than one register or register range. See "Examples" on page 36.

\( LDM \) and \( LDMFD \) are synonyms for \( LDMIA \). \( LDMFD \) refers to its use for popping data from Full Descending stacks.

\( LDMEA \) is a synonym for \( LDMDB \), and refers to its use for popping data from Empty Ascending stacks.

\( STM \) and \( STMEA \) are synonyms for \( STMIA \). \( STMEA \) refers to its use for pushing data onto Empty Ascending stacks.

\( STMFD \) is a synonym for \( STMDB \), and refers to its use for pushing data onto Full Descending stacks.
2.6.2 Operation

LDM instructions load the registers in `reglist` with word values from memory addresses based on `Rn`.

STM instructions store the word values in the registers in `reglist` to memory addresses based on `Rn`.

For LDM, LDMIA, LDMFD, STM, STMIA, and STMEA, the memory addresses used for the accesses are at 4-byte intervals ranging from `Rn` to `Rn + 4 * (n-1)`, where `n` is the number of registers in `reglist`. The accesses happen in order of increasing register numbers, with the lowest numbered register using the lowest memory address and the highest number register using the highest memory address. If the writeback suffix is specified, the value of `Rn + 4 * (n-1)` is written back to `Rn`.

For LDMDB, LDMEA, STMDB, and STMFD, the memory addresses used for the accesses are at 4-byte intervals ranging from `Rn` to `Rn - 4 * (n-1)`, where `n` is the number of registers in `reglist`. The accesses happen in order of decreasing register numbers, with the highest numbered register using the highest memory address and the lowest number register using the lowest memory address. If the writeback suffix is specified, the value of `Rn - 4 * (n-1)` is written back to `Rn`.

The PUSH and POP instructions can be expressed in this form. See “PUSH and POP” on page 37 for details.

2.6.3 Restrictions

In these instructions:

- `Rn` must not be PC.
- `reglist` must not contain SP.
- In any STM instruction, `reglist` must not contain PC.
- In any LDM instruction, `reglist` must not contain PC if it contains LR.
- `reglist` must not contain `Rn` if you specify the writeback suffix.

When PC is in `reglist` in an LDM instruction:

- Bit[0] of the value loaded to the PC must be 1 for correct execution, and a branch occurs to this halfword-aligned address.
- If the instruction is conditional, it must be the last instruction in the IT block.

2.6.4 Condition Flags

These instructions do not change the flags.

2.6.5 Examples

```
LDM   R8,{R0,R2,R9} ; LDMIA is a synonym for LDM.
STMDB R1!,{R3-R6,R11,R12}
```

2.6.6 Incorrect Examples

```
STM   R5!,{R5,R4,R9} ; Value stored for R5 is unpredictable.
LDM   R2, {} ; There must be at least one register in the list.
```
2.7 PUSH and POP

Push registers on and pop registers off a full-descending stack.

2.7.1 Syntax

PUSH\(cond\) reglist

POP\(cond\) reglist

where:

\(cond\)

Is an optional condition code. See Table 1-2 on page 22.

\(reglist\)

Is a non-empty list of registers, enclosed in braces. It can contain register ranges. It must be comma separated if it contains more than one register or register range.

PUSH and POP are synonyms for STMDB and LDM (or LDMIA) with the memory addresses for the access based on SP, and with the final address for the access written back to the SP. PUSH and POP are the preferred mnemonics in these cases.

2.7.2 Operation

PUSH stores registers on the stack in order of decreasing register numbers, with the highest numbered register using the highest memory address and the lowest numbered register using the lowest memory address.

POP loads registers from the stack in order of increasing register numbers, with the lowest numbered register using the lowest memory address and the highest numbered register using the highest memory address.

See “LDM and STM” on page 35 for more information.

2.7.3 Restrictions

In these instructions:

- \(reglist\) must not contain SP.
- For the \textbf{PUSH} instruction, \(reglist\) must not contain PC.
- For the \textbf{POP} instruction, \(reglist\) must not contain PC if it contains LR.

When PC is in \(reglist\) in a POP instruction:

- Bit[0] of the value loaded to the PC must be 1 for correct execution, and a branch occurs to this halfword-aligned address.
- If the instruction is conditional, it must be the last instruction in the IT block.

2.7.4 Condition Flags

These instructions do not change the flags.
2.7.5 Examples

PUSH   {R0,R4-R7}
PUSH   {R2,LR}
POP    {R0,R10,PC}
2.8 LDREX and STREX

Load and Store Register Exclusive.

2.8.1 Syntax

LDREX{cond} Rt, [Rn {, #offset}]

STREX{cond} Rd, Rt, [Rn {, #offset}]

LDREXB{cond} Rt, [Rn]

STREXB{cond} Rd, Rt, [Rn]

LDREXH{cond} Rt, [Rn]

STREXH{cond} Rd, Rt, [Rn]

where:

cond
Is an optional condition code. See Table 1-2 on page 22.

Rd
Is the destination register for the returned status.

Rt
Is the register to load or store.

Rn
Is the register on which the memory address is based.

offset
Is an optional offset applied to the value in Rn. If offset is omitted, the address is the value in Rn.

2.8.2 Operation

LDREX, LDREXB, and LDREXH load a word, byte, and halfword respectively from a memory address.

STREX, STREXB, and STREXH attempt to store a word, byte, and halfword respectively to a memory address. The address used in any Store-Exclusive instruction must be the same as the address in the most recently executed Load-exclusive instruction. The value stored by the Store-Exclusive instruction must also have the same data size as the value loaded by the preceding Load-exclusive instruction. This means software must always use a Load-exclusive instruction and a matching Store-Exclusive instruction to perform a synchronization operation (see "Synchronization Primitives" in the Stellaris® Data Sheet).

If a Store-Exclusive instruction performs the store, it writes 0 to its destination register. If it does not perform the store, it writes 1 to its destination register. If the Store-Exclusive instruction writes 0 to the destination register, it is guaranteed that no other process in the system has accessed the memory location between the Load-exclusive and Store-Exclusive instructions.

For reasons of performance, keep the number of instructions between corresponding Load-Exclusive and Store-Exclusive instruction to a minimum.
**Important:** The result of executing a Store-Exclusive instruction to an address that is different from that used in the preceding Load-Exclusive instruction is unpredictable.

### 2.8.3 Restrictions

In these instructions:

- Do not use **PC**.
- Do not use **SP** for **Rd** and **Rt**.
- For **STREX**, **Rd** must be different from both **Rt** and **Rn**.
- The value of **offset** must be a multiple of four in the range 0-1020.

### 2.8.4 Condition Flags

These instructions do not change the flags.

### 2.8.5 Examples

```assembly
MOV  R1, #0x1 ; Initialize the 'lock taken' value.
try
LDREX R0, [LockAddr] ; Load the lock value.
CMP  R0, #0 ; Is the lock free?
ITT  EQ ; IT instruction for STREXEQ and CMPEQ.
STREXEQ R0, R1, [LockAddr] ; Try and claim the lock.
CMPEQ R0, #0 ; Did this succeed?
BNE  try ; No - try again.
.... ; Yes - we have the lock.
```
2.9 **CLREX**
Clear Exclusive.

2.9.1 **Syntax**

\[
\text{CLREX} \{ \text{cond} \}
\]

where:

\[
\text{cond}
\]

is an optional condition code. See Table 1-2 on page 22.

2.9.2 **Operation**

Use \text{CLREX} to make the next \text{STREX}, \text{STREXB}, or \text{STREXH} instruction write 1 to its destination register and fail to perform the store. It is useful in exception handler code to force the failure of the store exclusive if the exception occurs between a load exclusive instruction and the matching store exclusive instruction in a synchronization operation (see "Synchronization Primitives" in the \textit{Stellaris® Data Sheet}).

2.9.3 **Condition Flags**

These instructions do not change the flags.

2.9.4 **Examples**

\[
\text{CLREX}
\]
3 General Data Processing Instructions

Table 3-1 on page 42 shows the data processing instructions:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Brief Description</th>
<th>See Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>Add with carry</td>
<td>43</td>
</tr>
<tr>
<td>ADD</td>
<td>Add</td>
<td>43</td>
</tr>
<tr>
<td>ADDW</td>
<td>Add</td>
<td>43</td>
</tr>
<tr>
<td>AND</td>
<td>Logical AND</td>
<td>46</td>
</tr>
<tr>
<td>ASR</td>
<td>Arithmetic shift right</td>
<td>48</td>
</tr>
<tr>
<td>BIC</td>
<td>Bit clear</td>
<td>46</td>
</tr>
<tr>
<td>CLZ</td>
<td>Count leading zeros</td>
<td>50</td>
</tr>
<tr>
<td>CMN</td>
<td>Compare negative</td>
<td>51</td>
</tr>
<tr>
<td>CMP</td>
<td>Compare</td>
<td>51</td>
</tr>
<tr>
<td>EOR</td>
<td>Exclusive OR</td>
<td>46</td>
</tr>
<tr>
<td>LSL</td>
<td>Logical shift left</td>
<td>48</td>
</tr>
<tr>
<td>LSR</td>
<td>Logical shift right</td>
<td>48</td>
</tr>
<tr>
<td>MOV</td>
<td>Move</td>
<td>52</td>
</tr>
<tr>
<td>MOVT</td>
<td>Move top</td>
<td>54</td>
</tr>
<tr>
<td>MOVW</td>
<td>Move 16-bit constant</td>
<td>52</td>
</tr>
<tr>
<td>MVN</td>
<td>Move NOT</td>
<td>52</td>
</tr>
<tr>
<td>ORN</td>
<td>Logical OR NOT</td>
<td>46</td>
</tr>
<tr>
<td>ORR</td>
<td>Logical OR</td>
<td>46</td>
</tr>
<tr>
<td>RBIT</td>
<td>Reverse bits</td>
<td>55</td>
</tr>
<tr>
<td>REV</td>
<td>Reverse byte order in a word</td>
<td>55</td>
</tr>
<tr>
<td>REV16</td>
<td>Reverse byte order in each halfword</td>
<td>55</td>
</tr>
<tr>
<td>REVSH</td>
<td>Reverse byte order in bottom halfword and sign extend</td>
<td>55</td>
</tr>
<tr>
<td>ROR</td>
<td>Rotate right</td>
<td>48</td>
</tr>
<tr>
<td>RRX</td>
<td>Rotate right with extend</td>
<td>48</td>
</tr>
<tr>
<td>RSB</td>
<td>Reverse subtract</td>
<td>43</td>
</tr>
<tr>
<td>SBC</td>
<td>Subtract with carry</td>
<td>43</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtract</td>
<td>43</td>
</tr>
<tr>
<td>SUBW</td>
<td>Subtract</td>
<td>43</td>
</tr>
<tr>
<td>TEQ</td>
<td>Test equivalence</td>
<td>57</td>
</tr>
<tr>
<td>TST</td>
<td>Test</td>
<td>57</td>
</tr>
</tbody>
</table>
3.1 ADD, ADC, SUB, SBC, and RSB

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

3.1.1 Syntax

\[
\text{op}(S)\{cond\} \{Rd,\} \text{ } Rn, \text{ } \text{Operand2} \\
\text{op}\{cond\} \{Rd,\} \text{ } Rn, \text{ } #\text{imm12} \\
\text{where:}
\]

- **op**
  - Is one of:
    - **ADD**
      - Add.
    - **ADC**
      - Add with Carry.
    - **SUB**
      - Subtract.
    - **SBC**
      - Subtract with Carry.
    - **RSB**
      - Reverse Subtract.

- **S**
  - Is an optional suffix. If *S* is specified, the condition code flags are updated on the result of the operation. See “Conditional Execution” on page 20.

- **cond**
  - Is an optional condition code. See Table 1-2 on page 22.

- **Rd**
  - Is the destination register. If *Rd* is omitted, the destination register is *Rn*.

- **Rn**
  - Is the register holding the first operand.

- **Operand2**
  - Is a flexible second operand. See “Flexible Second Operand” on page 15 for details of the options.

- **imm12**
  - Is any value in the range 0-4095.

3.1.2 Operation

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

The **ADC** instruction adds the values in **Rn** and **Operand2**, together with the carry flag.
The **SUB** instruction subtracts the value of *Operand2* or *imm12* from the value in *Rn*.

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

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

Use **ADC** and **SBC** to synthesize multiword arithmetic. See "Multiword Arithmetic Examples" on page 45.

See also 25.

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

### 3.1.3 Restrictions

In these instructions:

- **Operand2** must not be **SP** and must not be **PC**.

- **Rd** can be **SP** only in **ADD** and **SUB**, and only with the additional restrictions:
  - *Rn* must also be **SP**.
  - any shift in *Operand2* must be limited to a maximum of 3 bits using **LSL**.

- **Rn** can be **SP** only in **ADD** and **SUB**.

- **Rd** can be **PC** only in the **ADD{cond} PC, PC, Rm** instruction where:
  - You must not specify the S suffix.
  - **Rm** must not be **PC** and must not be **SP**.
  - If the instruction is conditional, it must be the last instruction in the **IT** block.

- With the exception of the **ADD{cond} PC, PC, Rm** instruction, **Rn** can be **PC** only in **ADD** and **SUB**, and only with the additional restrictions:
  - You must not specify the S suffix.
  - The second operand must be a constant in the range 0 to 4095.

**Note:** When using the **PC** for an addition or a subtraction, bits[1:0] of the **PC** are rounded to **00** before performing the calculation, making the base address for the calculation word-aligned.

- If you want to generate the address of an instruction, you have to adjust the constant based on the value of the **PC**. **ARM** recommends that you use the **ADR** instruction instead of **ADD** or **SUB** with **Rn** equal to the **PC**, because your assembler automatically calculates the correct constant for the **ADR** instruction.

When **Rd** is **PC** in the **ADD{cond} PC, PC, Rm** instruction:

- **Bit[0]** of the value written to the **PC** is ignored

- A branch occurs to the address created by forcing **bit[0]** of that value to **0**.

### 3.1.4 Condition Flags

If **S** is specified, these instructions update the **N, Z, C** and **V** flags according to the result.
3.1.5 Examples

ADD R2, R1, R3
SUBS R8, R6, #240 ; Sets the flags on the result.
RSB R4, R4, #1280 ; Subtracts contents of R4 from 1280.
ADCHI R11, R0, R3 ; Only executed if C flag set and Z flag clear.

3.1.6 Multiword Arithmetic Examples

Example 3-1, “64-Bit Addition” on page 45 shows two instructions that add a 64-bit integer contained in R2 and R3 to another 64-bit integer contained in R0 and R1, and place the result in R4 and R5.

Example 3-1. 64-Bit Addition

ADDS R4, R0, R2 ; Add the least significant words.
ADC R5, R1, R3 ; Add the most significant words with carry.

Multiword values do not have to use consecutive registers. Example 3-2, “96-Bit Subtraction” on page 45 shows instructions that subtract a 96-bit integer contained in R9, R1, and R11 from another contained in R6, R2, and R8. The example stores the result in R6, R9, and R2.

Example 3-2. 96-Bit Subtraction

SUBS R6, R6, R9 ; Subtract the least significant words.
SBCS R9, R2, R1 ; Subtract the middle words with carry.
SBC R2, R8, R11 ; Subtract the most significant words with carry.
3.2 AND, ORR, EOR, BIC, and ORN

Logical AND, OR, Exclusive OR, Bit Clear, and OR NOT.

3.2.1 Syntax

\[ op\{S\}\{cond\}\{Rd,\}\ Rn,\ Operand2 \]

where:

\( op \)

Is one of:

AND

Logical AND.

ORR

Logical OR, or bit set.

EOR

Logical Exclusive OR.

BIC

Logical AND NOT, or bit clear.

ORN

Logical OR NOT.

\( S \)

Is an optional suffix. If \( S \) is specified, the condition code flags are updated on the result of the operation. See “Conditional Execution” on page 20.

\( cond \)

Is an optional condition code. See Table 1-2 on page 22.

\( Rd \)

Is the destination register.

\( Rn \)

Is the register holding the first operand.

\( Operand2 \)

Is a flexible second operand. See “Flexible Second Operand” on page 15 for details of the options.

3.2.2 Operation

The AND, EOR, and ORR instructions perform bitwise AND, Exclusive OR, and OR operations on the values in \( Rn \) and \( Operand2 \).

The BIC instruction performs an AND operation on the bits in \( Rn \) with the complements of the corresponding bits in the value of \( Operand2 \).

The ORN instruction performs an OR operation on the bits in \( Rn \) with the complements of the corresponding bits in the value of \( Operand2 \).
3.2.3 Restrictions

Do not use SP and do not use PC.

3.2.4 Condition Flags

If S is specified, these instructions:

- Update the N and Z flags according to the result.
- Can update the C flag during the calculation of Operand2. See “Flexible Second Operand” on page 15.
- Do not affect the V flag.

3.2.5 Examples

\[
\begin{align*}
\text{AND} & \quad R9, R2, \#0xFF00 \\
\text{ORREQ} & \quad R2, R0, R5 \\
\text{ANDS} & \quad R9, R8, \#0x19 \\
\text{EORS} & \quad R7, R11, \#0x18181818 \\
\text{BIC} & \quad R0, R1, \#0xab \\
\text{ORN} & \quad R7, R11, R14, \text{ROR} \#4 \\
\text{ORNS} & \quad R7, R11, R14, \text{ASR} \#32
\end{align*}
\]
3.3 **ASR, LSL, LSR, ROR, and RRX**

Arithmetic Shift Right, Logical Shift Left, Logical Shift Right, Rotate Right, and Rotate Right with Extend.

3.3.1 **Syntax**

\[ op(S)\{cond\} \quad Rd, \, Rm, \, Rs \]
\[ op(S)\{cond\} \quad Rd, \, Rm, \, \#n \]
\[ RRX(S)\{cond\} \quad Rd, \, Rm \]

where:

\( op \)

Is one of:

- **ASR**
  - Arithmetic Shift Right.

- **LSL**
  - Logical Shift Left.

- **LSR**
  - Logical Shift Right.

- **ROR**
  - Rotate Right.

\( S \)

Is an optional suffix. If \( S \) is specified, the condition code flags are updated on the result of the operation. See "Conditional Execution" on page 20.

\( Rd \)

Is the destination register.

\( Rm \)

Is the register holding the value to be shifted.

\( Rs \)

Is the register holding the shift length to apply to the value in \( Rm \). Only the least significant byte is used and can be in the range 0 to 255.

\( n \)

Is the shift length. The range of shift length depends on the instruction:

- **ASR**
  - Shift length from 1 to 32.

- **LSL**
  - Shift length from 0 to 31.

- **LSR**
  - Shift length from 1 to 32.
ROR
Shift length from 1 to 31.

Note: MOV(S){cond} Rd, Rm is the preferred syntax for LSL(S){cond} Rd, Rm, #0.

3.3.2 Operation
ASR, LSL, LSR, and ROR move the bits in the register Rm to the left or right by the number of places specified by constant n or register Rs.
RRX moves the bits in register Rm to the right by 1.
In all these instructions, the result is written to Rd, but the value in register Rm remains unchanged. For details on what result is generated by the different instructions, see “Shift Operations” on page 17.

3.3.3 Restrictions
Do not use SP and do not use PC.

3.3.4 Condition Flags
If S is specified:
- These instructions update the N and Z flags according to the result.
- The C flag is updated to the last bit shifted out, except when the shift length is 0. See “Shift Operations” on page 17.

3.3.5 Examples
ASR R7, R8, #9 ; Arithmetic shift right by 9 bits.
LSLS R1, R2, #3 ; Logical shift left by 3 bits with flag update.
LSR R4, R5, #6 ; Logical shift right by 6 bits.
ROR R4, R5, R6 ; Rotate right by the value in the bottom byte of R6.
RRX R4, R5 ; Rotate right with extend.
3.4 CLZ

Count Leading Zeros.

3.4.1 Syntax

\[ \text{CLZ(} \text{cond} \text{)} \text{ Rd, Rm} \]

where:

- \text{cond} is an optional condition code. See Table 1-2 on page 22.
- \text{Rd} is the destination register.
- \text{Rm} is the operand register.

3.4.2 Operation

The CLZ instruction counts the number of leading zeros in the value in \text{Rm} and returns the result in \text{Rd}. The result value is 32 if no bits are set in the source register, and zero if bit[31] is set.

3.4.3 Restrictions

Do not use SP and do not use PC.

3.4.4 Condition Flags

This instruction does not change the flags.

3.4.5 Examples

- \text{CLZ} \text{ R4, R9}
- \text{CLZNE} \text{ R2, R3}
3.5 CMP and CMN

Compare and Compare Negative.

3.5.1 Syntax

CMP {cond} Rn, Operand2
CMN {cond} Rn, Operand2

where:

cond
  Is an optional condition code. See Table 1-2 on page 22.

Rn
  Is the register holding the first operand.

Operand2
  Is a flexible second operand. See "Flexible Second Operand" on page 15 for details of the options.

3.5.2 Operation

These instructions compare the value in a register with Operand2. They update the condition flags on the result, but do not write the result to a register.

The CMP instruction subtracts the value of Operand2 from the value in Rn. This is the same as a SUBS instruction, except that the result is discarded.

The CMN instruction adds the value of Operand2 to the value in Rn. This is the same as an ADDS instruction, except that the result is discarded.

3.5.3 Restrictions

In these instructions:

- Do not use PC.
- Operand2 must not be SP.

3.5.4 Condition Flags

These instructions update the N, Z, C and V flags according to the result.

3.5.5 Examples

CMP R2, R9
CMN R0, #6400
CMPGT SP, R7, LSL #2
3.6 **MOV and MVN**

Move and Move NOT.

### 3.6.1 Syntax

\[
\text{MOV} \{S\} \{\text{cond}\} \text{Rd, Operand2} \\
\text{MOV} \{\text{cond}\} \text{Rd, } \#\text{imm16} \\
\text{MVN} \{S\} \{\text{cond}\} \text{Rd, Operand2}
\]

where:

- \( S \)
  
  Is an optional suffix. If \( S \) is specified, the condition code flags are updated on the result of the operation. See “Conditional Execution” on page 20.

- \( \text{cond} \)
  
  Is an optional condition code. See Table 1-2 on page 22.

- \( \text{Rd} \)
  
  Is the destination register.

- \( \text{Operand2} \)
  
  Is a flexible second operand. See “Flexible Second Operand” on page 15 for details of the options.

- \( \text{imm16} \)
  
  Is any value in the range 0-65535.

### 3.6.2 Operation

The \text{MOV} instruction copies the value of \text{Operand2} into \text{Rd}.

When \text{Operand2} in a \text{MOV} instruction is a register with a shift other than \text{LSL } \#0, the preferred syntax is the corresponding shift instruction:

<table>
<thead>
<tr>
<th>MOV Instruction</th>
<th>Preferred Syntax using Shift Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{MOV} {S} {\text{cond}} \text{Rd, Rm, ASR } #n</td>
<td>\text{ASR} {S} {\text{cond}} \text{Rd, Rm, } #n</td>
</tr>
<tr>
<td>\text{MOV} {S} {\text{cond}} \text{Rd, Rm, LSL } #n (if } n \neq 0)</td>
<td>\text{LSL} {S} {\text{cond}} \text{Rd, Rm, } #n</td>
</tr>
<tr>
<td>\text{MOV} {S} {\text{cond}} \text{Rd, Rm, LSR } #n</td>
<td>\text{LSR} {S} {\text{cond}} \text{Rd, Rm, } #n</td>
</tr>
<tr>
<td>\text{MOV} {S} {\text{cond}} \text{Rd, Rm, ROR } #n</td>
<td>\text{ROR} {S} {\text{cond}} \text{Rd, Rm, } #n</td>
</tr>
<tr>
<td>\text{MOV} {S} {\text{cond}} \text{Rd, Rm, RRX}</td>
<td>\text{RRX} {S} {\text{cond}} \text{Rd, Rm}</td>
</tr>
</tbody>
</table>

Also, the \text{MOV} instruction permits additional forms of \text{Operand2} as synonyms for shift instructions. See “ASR, LSL, LSR, ROR, and RRX” on page 48.

<table>
<thead>
<tr>
<th>Shift Instruction</th>
<th>MOV Instruction Synonym</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{ASR} {S} {\text{cond}} \text{Rd, Rm, Rs}</td>
<td>\text{MOV} {S} {\text{cond}} \text{Rd, Rm, ASR } \text{Rs}</td>
</tr>
<tr>
<td>\text{LSL} {S} {\text{cond}} \text{Rd, Rm, Rs}</td>
<td>\text{MOV} {S} {\text{cond}} \text{Rd, Rm, LSL } \text{Rs}</td>
</tr>
<tr>
<td>\text{LSR} {S} {\text{cond}} \text{Rd, Rm, Rs}</td>
<td>\text{MOV} {S} {\text{cond}} \text{Rd, Rm, LSR } \text{Rs}</td>
</tr>
<tr>
<td>\text{ROR} {S} {\text{cond}} \text{Rd, Rm, Rs}</td>
<td>\text{MOV} {S} {\text{cond}} \text{Rd, Rm, ROR } \text{Rs}</td>
</tr>
</tbody>
</table>
The `MVN` instruction takes the value of `Operand2`, performs a bitwise logical NOT operation on the value, and places the result into `Rd`.

**Note:** The `MOVW` instruction provides the same function as `MOV`, but is restricted to using the `imm16` operand.

### 3.6.3 Restrictions
You can use `SP` and `PC` only in the `MOV` instruction, with the following restrictions:

- The second operand must be a register without shift
- You must not specify the `S` suffix.

When `Rd` is `PC` in a `MOV` instruction:

- Bit[0] of the value written to the `PC` is ignored
- A branch occurs to the address created by forcing bit[0] of that value to 0.

**Note:** Though it is possible to use `MOV` as a branch instruction, Texas Instruments strongly recommends the use of a `BX` or `BLX` instruction to branch for software portability to the ARM Cortex-M3 instruction set.

### 3.6.4 Condition Flags
If `S` is specified, these instructions:

- Update the `N` and `Z` flags according to the result.
- Can update the `C` flag during the calculation of `Operand2`. See “Flexible Second Operand” on page 15.
- Do not affect the `V` flag.

### 3.6.5 Example

```
MOVS R11, #0x000B ; Write value of 0x000B to R11, flags get updated.
MOV R1, #0xFA05 ; Write value of 0xFA05 to R1, flags are not updated.
MOVS R10, R12 ; Write value in R12 to R10, flags get updated.
MOV R3, #23 ; Write value of 23 to R3.
MOV R8, SP ; Write value of stack pointer to R8.
MVNS R2, #0xF ; Write value of 0xFFFFFFFF (bitwise inverse of 0xF) to R2 and update flags.
```
3.7  **MOVT**

Move Top.

### 3.7.1 Syntax

\[ \text{MOVT}\{\text{cond}\} \ R_d, \ #\text{imm16} \]

where:

- \( \text{cond} \) is an optional condition code. See Table 1-2 on page 22.
- \( \text{R}_d \) is the destination register.
- \( \text{imm16} \) is a 16-bit immediate constant.

### 3.7.2 Operation

\text{MOVT} writes a 16-bit immediate value, \text{imm16}, to the top halfword, \text{R}_d[31:16], of its destination register. The write does not affect \text{R}_d[15:0].

The \text{MOV, MOVT} instruction pair enables you to generate any 32-bit constant.

### 3.7.3 Restrictions

\( \text{R}_d \) must not be \text{SP} and must not be \text{PC}.

### 3.7.4 Condition Flags

This instruction does not change the flags.

### 3.7.5 Examples

\text{MOVT} \ R_3, \ #0xF123 ; \text{Write 0xF123 to upper halfword of R3, lower halfword ; and APSR are unchanged.}
3.8 REV, REV16, REVSH, and RBIT
Reverse bytes and Reverse bits.

3.8.1 Syntax

\[ op \{cond\} \ Rd, \ Rn \]

where:

\( op \)
Is any of:

REV
Reverse byte order in a word.

REV16
Reverse byte order in each halfword independently.

REVSH
Reverse byte order in the bottom halfword, and sign extend to 32 bits.

RBIT
Reverse the bit order in a 32-bit word.

\( cond \)
Is an optional condition code. See Table 1-2 on page 22.

\( Rd \)
Is the destination register.

\( Rn \)
Is the register holding the operand.

3.8.2 Operation
Use these instructions to change endianness of data:

REV
Converts 32-bit big-endian data into little-endian data or 32-bit little-endian data into big-endian data.

REV16
Converts 16-bit big-endian data into little-endian data or 16-bit little-endian data into big-endian data.

REVSH
Converts either:
- 16-bit signed big-endian data into 32-bit signed little-endian data.
- 16-bit signed little-endian data into 32-bit signed big-endian data.

3.8.3 Restrictions
Do not use SP and do not use PC.
3.8.4 Condition Flags

These instructions do not change the flags.

3.8.5 Examples

REV R3, R7 ; Reverse byte order of value in R7 and write it to R3.
REV16 R0, R0 ; Reverse byte order of each 16-bit halfword in R0.
REVSH R0, R5 ; Reverse Signed Halfword.
REVHS R3, R7 ; Reverse with Higher or Same condition.
RBIT R7, R8 ; Reverse bit order of value in R8 and write the result to R7.
3.9 TST and TEQ
Test bits and Test Equivalence.

3.9.1 Syntax

\[
\text{TST}\{\text{cond}\} \ Rn, \ \text{Operand2} \\
\text{TEQ}\{\text{cond}\} \ Rn, \ \text{Operand2}
\]

where:

\text{cond}

Is an optional condition code. See Table 1-2 on page 22.

\text{Rn}

Is the register holding the first operand.

\text{Operand2}

Is a flexible second operand. See "Flexible Second Operand" on page 15 for details of the options.

3.9.2 Operation

These instructions test the value in a register against \text{Operand2}. They update the condition flags based on the result, but do not write the result to a register.

The \text{TST} instruction performs a bitwise AND operation on the value in \text{Rn} and the value of \text{Operand2}. This is the same as the \text{ANDS} instruction, except that it discards the result.

To test whether a bit of \text{Rn} is 0 or 1, use the \text{TST} instruction with an \text{Operand2} constant that has that bit set to 1 and all other bits cleared to 0.

The \text{TEQ} instruction performs a bitwise Exclusive OR operation on the value in \text{Rn} and the value of \text{Operand2}. This is the same as the \text{EORS} instruction, except that it discards the result.

Use the \text{TEQ} instruction to test if two values are equal without affecting the V or C flags.

\text{TEQ} is also useful for testing the sign of a value. After the comparison, the \text{N} flag is the logical Exclusive OR of the sign bits of the two operands.

3.9.3 Restrictions

Do not use \text{SP} and do not use \text{PC}.

3.9.4 Condition Flags

These instructions:

- Update the \text{N} and \text{Z} flags according to the result.

- Can update the \text{C} flag during the calculation of \text{Operand2}. See "Flexible Second Operand" on page 15.

- Do not affect the \text{V} flag.
3.9.5 Examples

TST     R0, #0x3F8 ; Perform bitwise AND of R0 value to 0x3F8;
         ; APSR is updated but result is discarded.
TEQEQ   R10, R9  ; Conditionally test if value in R10 is equal to
                 ; value in R9; APSR is updated but result is discarded.
4 Multiply and Divide Instructions

Table 4-1 on page 59 shows the multiply and divide instructions:

Table 4-1. Multiply and Divide Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Brief Description</th>
<th>See Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLA</td>
<td>Multiply with accumulate, 32-bit result</td>
<td>60</td>
</tr>
<tr>
<td>MLS</td>
<td>Multiply and subtract, 32-bit result</td>
<td>60</td>
</tr>
<tr>
<td>MUL</td>
<td>Multiply, 32-bit result</td>
<td>60</td>
</tr>
<tr>
<td>SDIV</td>
<td>Signed divide</td>
<td>64</td>
</tr>
<tr>
<td>SMLAL</td>
<td>Signed multiply with accumulate (32x32+64), 64-bit result</td>
<td>62</td>
</tr>
<tr>
<td>SMULL</td>
<td>Signed multiply (32x32), 64-bit result</td>
<td>62</td>
</tr>
<tr>
<td>UDIV</td>
<td>Unsigned divide</td>
<td>64</td>
</tr>
<tr>
<td>UMLAL</td>
<td>Unsigned multiply with accumulate (32x32+64), 64-bit result</td>
<td>62</td>
</tr>
<tr>
<td>UMULL</td>
<td>Unsigned multiply (32x32), 64-bit result</td>
<td>62</td>
</tr>
</tbody>
</table>
4.1 MUL, MLA, and MLS

Multiply, Multiply with Accumulate, and Multiply with Subtract, using 32-bit operands, and producing a 32-bit result.

4.1.1 Syntax

\[
\text{MUL} (S) (\text{cond}) \{\text{Rd},\} \text{ Rn, Rm} \ ; \text{ Multiply}
\]

\[
\text{MLA} (\text{cond}) \text{ Rd, Rn, Rm, Ra} \ ; \text{ Multiply with accumulate}
\]

\[
\text{MLS} (\text{cond}) \text{ Rd, Rn, Rm, Ra} \ ; \text{ Multiply with subtract}
\]

where:

- \text{cond} is an optional condition code. See Table 1-2 on page 22.
- \text{S} is an optional suffix. If \text{S} is specified, the condition code flags are updated on the result of the operation. See "Conditional Execution" on page 20.
- \text{Rd} is the destination register. If \text{Rd} is omitted, the destination register is \text{Rn}.
- \text{Rn, Rm} are registers holding the values to be multiplied.
- \text{Ra} is a register holding the value to be added or subtracted from.

4.1.2 Operation

The MUL instruction multiplies the values from \text{Rn} and \text{Rm}, and places the least-significant 32 bits of the result in \text{Rd}.

The MLA instruction multiplies the values from \text{Rn} and \text{Rm}, adds the value from \text{Ra}, and places the least-significant 32 bits of the result in \text{Rd}.

The MLS instruction multiplies the values from \text{Rn} and \text{Rm}, subtracts the product from \text{Ra}, and places the least-significant 32 bits of the result in \text{Rd}.

The results of these instructions do not depend on whether the operands are signed or unsigned.

4.1.3 Restrictions

In these instructions, do not use SP and do not use PC.

If you use the S suffix with the MUL instruction:

- \text{Rd, Rn, and Rm} must all be in the range R0 to R7.
- \text{Rd} must be the same as \text{Rm}.
- You must not use the cond suffix.
4.1.4 Condition Flags

If \( S \) is specified, the MUL instruction:

- Updates the \( N \) and \( Z \) flags according to the result.
- Does not affect the \( C \) and \( V \) flags.

4.1.5 Examples

```assembly
MUL    R10, R2, R5 ; Multiply, \( R10 = R2 \times R5 \).
MLA    R10, R2, R1, R5 ; Multiply with accumulate, \( R10 = (R2 \times R1) + R5 \).
MULS   R0, R2, R2 ; Multiply with flag update, \( R0 = R2 \times R2 \).
MULLT  R2, R3, R2 ; Conditionally multiply, \( R2 = R3 \times R2 \).
MLS    R4, R5, R6, R7 ; Multiply with subtract, \( R4 = R7 - (R5 \times R6) \).
```
4.2 UMULL, UMLAL, SMULL, and SMLAL

Signed and Unsigned Long Multiply, with optional Accumulate, using 32-bit operands and producing a 64-bit result.

4.2.1 Syntax

\[ \text{op}(\text{cond}) \text{ RdLo, RdHi, Rn, Rm} \]

where:

\[ \text{op} \]

Is one of:

- UMULL: Unsigned Long Multiply.
- UMLAL: Unsigned Long Multiply, with Accumulate.
- SMULL: Signed Long Multiply.
- SMLAL: Signed Long Multiply, with Accumulate.

\[ \text{cond} \]

Is an optional condition code. See Table 1-2 on page 22.

\[ \text{RdHi, RdLo} \]

Are the destination registers. For UMLAL and SMLAL they also hold the accumulating value.

\[ \text{Rn, Rm} \]

Are registers holding the operands.

4.2.2 Operation

The UMULL instruction interprets the values from Rn and Rm as unsigned integers. It multiplies these integers and places the least-significant 32 bits of the result in RdLo, and the most-significant 32 bits of the result in RdHi.

The UMLAL instruction interprets the values from Rn and Rm as unsigned integers. It multiplies these integers, adds the 64-bit result to the 64-bit unsigned integer contained in RdHi and RdLo, and writes the result back to RdHi and RdLo.

The SMULL instruction interprets the values from Rn and Rm as two’s complement signed integers. It multiplies these integers and places the least-significant 32 bits of the result in RdLo, and the most-significant 32 bits of the result in RdHi.

The SMLAL instruction interprets the values from Rn and Rm as two’s complement signed integers. It multiplies these integers, adds the 64-bit result to the 64-bit signed integer contained in RdHi and RdLo, and writes the result back to RdHi and RdLo.

4.2.3 Restrictions

In these instructions:
Do not use SP and do not use PC.

RdHi and RdLo must be different registers.

4.2.4 Condition Flags
These instructions do not affect the flags.

4.2.5 Examples

UMULL R0, R4, R5, R6 ; Unsigned (R4,R0) = R5 x R6.
SMLAL R4, R5, R3, R8 ; Signed (R5,R4) = (R5,R4) + R3 x R8.
4.3 SDIV and UDIV
Signed Divide and Unsigned Divide.

4.3.1 Syntax

SDIV{cond} {Rd,} Rn, Rm
UDIV{cond} {Rd,} Rn, Rm

where:

cond
Is an optional condition code. See Table 1-2 on page 22.

Rd
Is the destination register. If Rd is omitted, the destination register is Rn.

Rn
Is the register holding the value to be divided.

Rm
Is a register holding the divisor.

4.3.2 Operation

SDIV performs a signed integer division of the value in Rn by the value in Rm.
UDIV performs an unsigned integer division of the value in Rn by the value in Rm.

For both instructions, if the value in Rn is not divisible by the value in Rm, the result is rounded towards zero.

4.3.3 Restrictions
Do not use SP and do not use PC.

4.3.4 Condition Flags
These instructions do not change the flags.

4.3.5 Examples

SDIV R0, R2, R4 ; Signed divide, R0 = R2/R4.
UDIV R8, R8, R1 ; Unsigned divide, R8 = R8/R1.
5 Saturating Instructions

Table 5-1 on page 65 shows the saturating instructions:

Table 5-1. Saturating Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Brief Description</th>
<th>See Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSAT</td>
<td>Signed saturate</td>
<td>66</td>
</tr>
<tr>
<td>USAT</td>
<td>Unsigned saturate</td>
<td>66</td>
</tr>
</tbody>
</table>
5.1 SSAT and USAT
Signed Saturate and Unsigned Saturate to any bit position, with optional shift before saturating.

5.1.1 Syntax

\[ \text{op} \{ \text{cond} \} \ R_d, \ #_n, \ R_m \{, \ \text{shift} \ #_s \} \]

where:

\( \text{op} \) is one of:

- **SSAT**
  Saturates a signed value to a signed range.

- **USAT**
  Saturates a signed value to an unsigned range.

\( \text{cond} \) is an optional condition code. See Table 1-2 on page 22.

\( R_d \)
Is the destination register.

\( n \)
Specifies the bit position to saturate to:
- \( n \) ranges from 1 to 32 for **SSAT**
- \( n \) ranges from 0 to 31 for **USAT**

\( R_m \)
Is the register containing the value to saturate.

\( \text{shift} \ #_s \)
Is an optional shift applied to \( R_m \) before saturating. It must be one of the following:

- **ASR** \( #_s \)
  Where \( s \) is in the range 1 to 31.

- **LSL** \( #_s \)
  Where \( s \) is in the range 0 to 31.

5.1.2 Operation
These instructions saturate to a signed or unsigned \( n \)-bit value.
The **SSAT** instruction applies the specified shift, then saturates to the signed range \( -2^{n-1} \leq x \leq 2^{n-1}-1 \).
The **USAT** instruction applies the specified shift, then saturates to the unsigned range \( 0 \leq x \leq 2^{n}-1 \).
For signed \( n \)-bit saturation using **SSAT**, this means that:
- If the value to be saturated is less than \( -2^{n-1} \), the result returned is \( -2^{n-1} \).
- If the value to be saturated is greater than \( 2^{n-1}-1 \), the result returned is \( 2^{n-1}-1 \).
- Otherwise, the result returned is the same as the value to be saturated.

For unsigned \( n \)-bit saturation using \text{USAT} \), this means that:

- If the value to be saturated is less than 0, the result returned is 0.
- If the value to be saturated is greater than \( 2^{n-1} \), the result returned is \( 2^{n-1} \).
- Otherwise, the result returned is the same as the value to be saturated.

If the returned result is different from the value to be saturated, it is called saturation. If saturation occurs, the instruction sets the \( Q \) flag to 1 in the \text{APSR}. Otherwise, it leaves the \( Q \) flag unchanged.

To clear the \( Q \) flag to 0, you must use the \text{MSR} instruction. See “MSR” on page 89.

To read the state of the \( Q \) flag, use the \text{MRS} instruction. See “MRS” on page 88.

### 5.1.3 Restrictions

Do not use \text{SP} and do not use \text{PC}.

### 5.1.4 Condition Flags

These instructions do not affect the condition code flags.

If saturation occurs, these instructions set the \( Q \) flag to 1.

### 5.1.5 Examples

\text{SSAT} \ R7, \ #16, \ R7, \ LSL \ #4 \ ; \ Logical \ shift \ left \ value \ in \ R7 \ by \ 4, \ then
\ ; \ saturate \ it \ as \ a \ signed \ 16\text{-}bit \ value \ and
\ ; \ write \ it \ back \ to \ R7.

\text{USATNE} \ R0, \ #7, \ R5 \ ; \ Conditionally \ saturate \ value \ in \ R5 \ as \ an
\ ; \ unsigned \ 7 \bit \ value \ and \ write \ it \ to \ R0.
6 Bitfield Instructions

Table 6-1 on page 68 shows the instructions that operate on adjacent sets of bits in registers or bitfields:

Table 6-1. Bitfield Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Brief Description</th>
<th>See Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFC</td>
<td>Bit field clear</td>
<td>69</td>
</tr>
<tr>
<td>BFI</td>
<td>Bit field insert</td>
<td>69</td>
</tr>
<tr>
<td>SBFX</td>
<td>Signed bit field extract</td>
<td>70</td>
</tr>
<tr>
<td>SXTB</td>
<td>Sign extend a byte</td>
<td>71</td>
</tr>
<tr>
<td>SXTH</td>
<td>Sign extend a halfword</td>
<td>71</td>
</tr>
<tr>
<td>UBFX</td>
<td>Unsigned bit field extract</td>
<td>70</td>
</tr>
<tr>
<td>UXTB</td>
<td>Zero extend a byte</td>
<td>71</td>
</tr>
<tr>
<td>UXTH</td>
<td>Zero extend a halfword</td>
<td>71</td>
</tr>
</tbody>
</table>
6.1 **BFC and BFI**

Bit Field Clear and Bit Field Insert.

### 6.1.1 Syntax

\[
\text{BFC}\{\text{cond}\} \text{ Rd, } \#\text{lsb}, \#\text{width} \\
\text{BFI}\{\text{cond}\} \text{ Rd, Rn, } \#\text{lsb}, \#\text{width}
\]

where:

- **cond**
  Is an optional condition code. See Table 1-2 on page 22.

- **Rd**
  Is the destination register.

- **Rn**
  Is the source register.

- **lsb**
  Is the position of the least-significant bit of the bitfield. \(\text{lsb}\) must be in the range 0 to 31.

- **width**
  Is the width of the bitfield and must be in the range 1 to \(32 - \text{lsb}\).

### 6.1.2 Operation

**BFC** clears a bitfield in a register. It clears \(\text{width}\) bits in \(\text{Rd}\), starting at the low bit position \(\text{lsb}\). Other bits in \(\text{Rd}\) are unchanged.

**BFI** copies a bitfield into one register from another register. It replaces \(\text{width}\) bits in \(\text{Rd}\) starting at the low bit position \(\text{lsb}\), with \(\text{width}\) bits from \(\text{Rn}\) starting at bit[0]. Other bits in \(\text{Rd}\) are unchanged.

### 6.1.3 Restrictions

Do not use **SP** and do not use **PC**.

### 6.1.4 Condition Flags

These instructions do not affect the flags.

### 6.1.5 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.
```
6.2  **SBFX and UBFX**

Signed Bit Field Extract and Unsigned Bit Field Extract.

### 6.2.1 Syntax

```c
SBFX{cond} Rd, Rn, #lsb, #width
UBFX{cond} Rd, Rn, #lsb, #width
```

where:

- `cond` is an optional condition code. See Table 1-2 on page 22.

- `Rd` is the destination register.

- `Rn` is the source register.

- `lsb` is the position of the least-significant bit of the bitfield. `lsb` must be in the range 0 to 31.

- `width` is the width of the bitfield and must be in the range 1 to `32−lsb`.

### 6.2.2 Operation

**SBFX** extracts a bitfield from one register, sign extends it to 32 bits, and writes the result to the destination register.

**UBFX** extracts a bitfield from one register, zero extends it to 32 bits, and writes the result to the destination register.

### 6.2.3 Restrictions

Do not use **SP** and do not use **PC**.

### 6.2.4 Condition Flags

These instructions do not affect the flags.

### 6.2.5 Examples

```assembly
SBFX  R0, R1, #20, #4  ; Extract bit 20 to bit 23 (4 bits) from R1 and sign extend to 32 bits and then write the result to R0.
UBFX  R8, R11, #9, #10 ; Extract bit 9 to bit 18 (10 bits) from R11 and zero extend to 32 bits and then write the result to R8.
```
6.3 **SXT and UXT**

Sign extend and Zero extend.

### 6.3.1 Syntax

\[
\begin{align*}
SXT & \text{extend} \{cond\} \{Rd,\} \{Rm,\ \text{ROR} \ #n\} \\
UXT & \text{extend} \{cond\} \{Rd,\} \{Rm,\ \text{ROR} \ #n\}
\end{align*}
\]

where:

**extend**

Is one of:

- **B**
  
  Extends an 8-bit value to a 32-bit value.

- **H**
  
  Extends a 16-bit value to a 32-bit value.

**cond**

Is an optional condition code. See Table 1-2 on page 22.

**Rd**

Is the destination register.

**Rm**

Is the register holding the value to extend.

**ROR \ #n**

Is one of:

- **ROR \ #8**
  
  Value from \(Rm\) is rotated right 8 bits.

- **ROR \ #16**
  
  Value from \(Rm\) is rotated right 16 bits.

- **ROR \ #24**
  
  Value from \(Rm\) is rotated right 24 bits.

If **ROR \ #n** is omitted, no rotation is performed.

### 6.3.2 Operation

These instructions do the following:

1. Rotate the value from \(Rm\) right by 0, 8, 16 or 24 bits.

2. Extract bits from the resulting value:
   - **SXTB** extracts bits[7:0] and sign extends to 32 bits.
   - **UXTB** extracts bits[7:0] and zero extends to 32 bits.
   - **SXTH** extracts bits[15:0] and sign extends to 32 bits.
   - **UXTH** extracts bits[15:0] and zero extends to 32 bits.
6.3.3  Restrictions
Do not use SP and do not use PC.

6.3.4  Condition Flags
These instructions do not affect the flags.

6.3.5  Examples

SXTH R4, R6, ROR #16 ; Rotate R6 right by 16 bits, then obtain the lower halfword of the result and then sign extend to 32 bits and write the result to R4.

UXTB R3, R10 ; Extract lowest byte of the value in R10 and zero extend it, and write the result to R3.
7 Branch and Control Instructions

Table 7-1 on page 73 shows the branch and control instructions:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Brief Description</th>
<th>See Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Branch</td>
<td>74</td>
</tr>
<tr>
<td>BL</td>
<td>Branch with link</td>
<td>74</td>
</tr>
<tr>
<td>BLX</td>
<td>Branch indirect with link</td>
<td>74</td>
</tr>
<tr>
<td>BX</td>
<td>Branch indirect</td>
<td>74</td>
</tr>
<tr>
<td>CBNZ</td>
<td>Compare and branch if non-zero</td>
<td>76</td>
</tr>
<tr>
<td>CBZ</td>
<td>Compare and branch if zero</td>
<td>76</td>
</tr>
<tr>
<td>IT</td>
<td>If-Then</td>
<td>77</td>
</tr>
<tr>
<td>TBB</td>
<td>Table branch byte</td>
<td>80</td>
</tr>
<tr>
<td>TBH</td>
<td>Table branch halfword</td>
<td>80</td>
</tr>
</tbody>
</table>
7.1 B, BL, BX, and BLX

Branch instructions.

7.1.1 Syntax

B\{cond\} label
BL\{cond\} label
BX\{cond\} Rm
BLX\{cond\} Rm

where:

B
Is branch (immediate).

BL
Is branch with link (immediate).

BX
Is branch indirect (register).

BLX
Is branch indirect with link (register).

cond
Is an optional condition code. See Table 1-2 on page 22.

label
Is a PC-relative expression. See “PC-Relative Expressions” on page 20.

Rm
Is a register that indicates an address to branch to. Bit[0] of the value in Rm must be 1, but the address to branch to is created by changing bit[0] to 0.

7.1.2 Operation

All these instructions cause a branch to label, or to the address indicated in Rm. In addition:

- The BL and BLX instructions write the address of the next instruction to the Link Register (LR), register R14. See the Stellaris® Data Sheet for more on LR.

- The BX and BLX instructions cause a UsageFault exception if bit[0] of Rm is 0.

B cond label is the only conditional instruction that can be either inside or outside an IT block. All other branch instructions must be conditional inside an IT block, and must be unconditional outside the IT block. See “IT” on page 77.

Table 7-2 on page 75 shows the ranges for the various branch instructions.
Table 7-2. Branch Ranges

| Instruction                | Branch Range  
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>B label</td>
<td>−16 MB to +16 MB</td>
</tr>
<tr>
<td>B&lt;cond&gt; label (outside IT block)</td>
<td>−1 MB to +1 MB</td>
</tr>
<tr>
<td>B&lt;cond&gt; label (inside IT block)</td>
<td>−16 MB to +16 MB</td>
</tr>
<tr>
<td>BL&lt;cond&gt; label</td>
<td>−16 MB to +16 MB</td>
</tr>
<tr>
<td>BX&lt;cond&gt; Rm</td>
<td>Any value in register</td>
</tr>
<tr>
<td>BLX&lt;cond&gt; Rm</td>
<td>Any value in register</td>
</tr>
</tbody>
</table>

a. You might have to use the .W suffix to get the maximum branch range. See “Instruction Width Selection” on page 22.

7.1.3 Restrictions

The restrictions are:

- Do not use PC in the BLX instruction.
- For BX and BLX, bit[0] of Rm must be 1 for correct execution but a branch occurs to the target address created by changing bit[0] to 0.
- When any of these instructions is inside an IT block, it must be the last instruction of the IT block.

Note: B<cond> is the only conditional instruction that is not required to be inside an IT block. However, it has a longer branch range when it is inside an IT block.

7.1.4 Condition Flags

These instructions do not change the flags.

7.1.5 Examples

B loopA ; Branch to loopA.
BLE ng ; Conditionally branch to label ng.
B.W target ; Branch to target within 16MB range.
BEQ target ; Conditionally branch to target.
BEQ.W target ; Conditionally branch to target within 1MB.
BL funC ; Branch with link (Call) to function funC, return address stored in LR.
BX LR ; Return from function call.
BXNE R0 ; Conditionally branch to address stored in R0.
BLX R0 ; Branch with link and exchange (Call) to a address stored in R0.
### 7.2 CBZ and CBNZ

Compare and Branch if Zero, Compare and Branch if Non-Zero.

#### 7.2.1 Syntax

CBZ \( Rn, \ label \)

CBNZ \( Rn, \ label \)

where:

\( Rn \)

Is the register holding the operand.

\( label \)

Is the branch destination.

#### 7.2.2 Operation

Use the CBZ or CBNZ instructions to avoid changing the condition code flags and to reduce the number of instructions.

CBZ \( Rn, \ label \) does not change condition flags but is otherwise equivalent to:

\[
\begin{align*}
  \text{CMP} & \quad Rn, \ #0 \\
  \text{BEQ} & \quad \text{label}
\end{align*}
\]

CBNZ \( Rn, \ label \) does not change condition flags but is otherwise equivalent to:

\[
\begin{align*}
  \text{CMP} & \quad Rn, \ #0 \\
  \text{BNE} & \quad \text{label}
\end{align*}
\]

#### 7.2.3 Restrictions

The restrictions are:

- \( Rn \) must be in the range of \( R0 \) to \( R7 \).
- The branch destination must be within 4 to 130 bytes after the instruction.
- These instructions must not be used inside an IT block.

#### 7.2.4 Condition Flags

These instructions do not change the flags.

#### 7.2.5 Examples

CBZ \( R5, \ target \) ; Forward branch if R5 is zero.

CBNZ \( R0, \ target \) ; Forward branch if R0 is not zero.
7.3 IT

If-Then

7.3.1 Syntax

\[ IT(x\{y\{z\}\}) \quad cond \]

where:

\[ x \]

Specifies the condition switch for the second instruction in the IT block.

\[ y \]

Specifies the condition switch for the third instruction in the IT block.

\[ z \]

Specifies the condition switch for the fourth instruction in the IT block.

\[ cond \]

Specifies the condition for the first instruction in the IT block.

The condition switch for the second, third and fourth instruction in the IT block can be either:

\[ T \]

Then. Applies the condition \( cond \) to the instruction.

\[ E \]

Else. Applies the inverse condition of \( cond \) to the instruction.

Note: It is possible to use \( AL \) (the always condition) for \( cond \) in an IT instruction. If this is done, all of the instructions in the IT block must be unconditional, and each of \( x \), \( y \), and \( z \) must be \( T \) or omitted but not \( E \).

7.3.2 Operation

The IT instruction makes up to four following instructions conditional. The conditions can be all the same, or some of them can be the logical inverse of the others. The conditional instructions following the IT instruction form the IT block.

The instructions in the IT block, including any branches, must specify the condition in the \( \{ cond \} \) part of their syntax.

Note: Your assembler might be able to generate the required IT instructions for conditional instructions automatically, so that you do not need to write them yourself. See your assembler documentation for details.

A BKPT instruction in an IT block is always executed, even if its condition fails.

Exceptions can be taken between an IT instruction and the corresponding IT block, or within an IT block. Such an exception results in entry to the appropriate exception handler, with suitable return information in LR and stacked PSR. See the PSR register in the Stellaris® Data Sheet for more information.

Instructions designed for use for exception returns can be used as normal to return from the exception, and execution of the IT block resumes correctly. This is the only way that a PC-modifying instruction is permitted to branch to an instruction in an IT block.
7.3.3 Restrictions

The following instructions are not permitted in an IT block:

- IT
- CBZ and CBNZ
- CPSID and CPSIE

Other restrictions when using an IT block are:

- A branch or any instruction that modifies the PC must either be outside an IT block or must be the last instruction inside the IT block. These are:
  - ADD PC, PC, Rm
  - MOV PC, Rm
  - B, BL, BX, BLX
  - any LDM, LDR, or POP instruction that writes to the PC
  - TBB and TBH

- Do not branch to any instruction inside an IT block, except when returning from an exception handler.

- All conditional instructions except Bcond must be inside an IT block. Bcond can be either outside or inside an IT block but has a larger branch range if it is inside one.

- Each instruction inside the IT block must specify a condition code suffix that is either the same or the logical inverse.

Note: Your assembler might place extra restrictions on the use of IT blocks, such as prohibiting the use of assembler directives within them.

7.3.4 Condition Flags

This instruction does not change the flags.

7.3.5 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.

CMP R0, #9 ; Convert R0 hex value (0 to 15) into ASCII
            ; ('0'-'9', 'A'-'F').
ITE GT ; Next 2 instructions are conditional.
ADDDT R1, R0, #55 ; Convert 0xA -> 'A'.
ADDLDE R1, R0, #48 ; Convert 0x0 -> '0'.

IT GT ; IT block with only one conditional instruction.
ADDDT R1, R1, #1 ; Increment R1 conditionally.
ITTEE  EQ        ; Next 4 instructions are conditional.
MOVEQ  R0, R1   ; Conditional move.
ADDEQ  R2, R2, #10 ; Conditional add.
ANDNE  R3, R3, #1 ; Conditional AND.
BNE.W  dloop    ; Branch instruction can only be used in the last
                ; instruction of an IT block.

IT    NE         ; Next instruction is conditional.
ADD   R0, R0, R1 ; Syntax error: no condition code used in IT block.
7.4 TBB and TBH

Table Branch Byte and Table Branch Halfword.

7.4.1 Syntax

TBB [Rn, Rm]
TBH [Rn, Rm, LSL #1]

where:

Rn
Is the register containing the address of the table of branch lengths.
If Rn is the Program Counter (PC) register, R15, then the address of the table is the address of the byte immediately following the TBB or TBH instruction.

Rm
Is the index register. This contains an index into the table. For halfword tables, LSL #1 doubles the value in Rm to form the right offset into the table.

7.4.2 Operation

These instructions cause a PC-relative forward branch using a table of single byte offsets for TBB, or halfword offsets for TBH. Rn provides a pointer to the table, and Rm supplies an index into the table. For TBB the branch offset is twice the unsigned value of the byte returned from the table. For TBH, the branch offset is twice the unsigned value of the halfword returned from the table. The branch occurs to the address at that offset from the address of the byte immediately after the TBB or TBH instruction.

7.4.3 Restrictions

The restrictions are:

■ Rn must not be SP.
■ Rn must not be SP and must not be PC.
■ When any of these instructions is used inside an IT block, it must be the last instruction of the IT block.

7.4.4 Condition Flags

These instructions do not change the flags.

7.4.5 Examples

ADR.W R0, BranchTable_Bit
TBB [R0, R1] ; R1 is the index, R0 is the base address of the ; branch table.

Case1
; an instruction sequence follows
Case2
; an instruction sequence follows
Case3
; an instruction sequence follows
BranchTable.Byte

DCB 0 ; Case1 offset calculation.
DCB ((Case2-Case1)/2) ; Case2 offset calculation.
DCB ((Case3-Case1)/2) ; Case3 offset calculation.

TBH [PC, R1, LSL #1] ; R1 is the index, PC is used as base of the
; branch table.

BranchTable.H

DCI ((CaseA - BranchTable_H)/2) ; CaseA offset calculation.
DCI ((CaseB - BranchTable_H)/2) ; CaseB offset calculation.
DCI ((CaseC - BranchTable_H)/2) ; CaseC offset calculation.

CaseA
; an instruction sequence follows
CaseB
; an instruction sequence follows
CaseC
; an instruction sequence follows
8 Miscellaneous Instructions

Table 8-1 on page 82 shows the remaining Cortex-M3 instructions:

Table 8-1. Miscellaneous Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Brief Description</th>
<th>See Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>BKPT</td>
<td>Breakpoint</td>
<td>83</td>
</tr>
<tr>
<td>CPSID</td>
<td>Change processor state, disable interrupts</td>
<td>84</td>
</tr>
<tr>
<td>CPSIE</td>
<td>Change processor state, enable interrupts</td>
<td>84</td>
</tr>
<tr>
<td>DMB</td>
<td>Data memory barrier</td>
<td>85</td>
</tr>
<tr>
<td>DSB</td>
<td>Data synchronization barrier</td>
<td>86</td>
</tr>
<tr>
<td>ISB</td>
<td>Instruction synchronization barrier</td>
<td>87</td>
</tr>
<tr>
<td>MRS</td>
<td>Move from special register to register</td>
<td>88</td>
</tr>
<tr>
<td>MSR</td>
<td>Move from register to special register</td>
<td>89</td>
</tr>
<tr>
<td>NOP</td>
<td>No operation</td>
<td>90</td>
</tr>
<tr>
<td>SEV</td>
<td>Send event</td>
<td>91</td>
</tr>
<tr>
<td>SVC</td>
<td>Supervisor call</td>
<td>92</td>
</tr>
<tr>
<td>WFE</td>
<td>Wait for event</td>
<td>93</td>
</tr>
<tr>
<td>WFI</td>
<td>Wait for interrupt</td>
<td>94</td>
</tr>
</tbody>
</table>
8.1 **BKPT**
Breakpoint.

8.1.1 **Syntax**

BKPT  #imm

where:

    *imm*  
    Is an expression evaluating to an integer in the range 0-255 (8-bit value).

8.1.2 **Operation**

The BKPT instruction causes the processor to enter Debug state. Debug tools can use this to investigate system state when the instruction at a particular address is reached.

*imm* is ignored by the processor. If required, a debugger can use it to store additional information about the breakpoint.

The BKPT instruction can be placed inside an IT block, but it executes unconditionally, unaffected by the condition specified by the IT instruction.

8.1.3 **Condition Flags**

This instruction does not change the flags.

8.1.4 **Examples**

    BKPT 0xAB   ; Breakpoint with immediate value set to 0xAB (debugger can; extract the immediate value by locating it using the PC).
8.2 CPS
Change Processor State.

8.2.1 Syntax

\[
\text{CPS} \text{effect iflags}
\]

where:

- **effect**
  - Is one of:
    - **IE**
      - Clears the special-purpose register.
    - **ID**
      - Sets the special-purpose register.

- **iflags**
  - Is a sequence of one or more flags:
    - **i**
      - Set or clear the Priority Mask Register (PRIMASK).
    - **f**
      - Set or clear the Fault Mask Register (FAULTMASK).

8.2.2 Operation

CPS changes the PRIMASK and FAULTMASK special register values. See the Stellars® Data Sheet for more information about these registers.

8.2.3 Restrictions

The restrictions are:

- Use CPS only from privileged software; it has no effect if used in unprivileged software.
- CPS cannot be conditional and so must not be used inside an IT block.

8.2.4 Condition Flags

This instruction does not change the flags.

8.2.5 Examples

- CPSID i ; Disable interrupts and configurable fault handlers (set PRIMASK).
- CPSID f ; Disable interrupts and all fault handlers (set FAULTMASK).
- CPSIE i ; Enable interrupts and configurable fault handlers (clear PRIMASK).
- CPSIE f ; Enable interrupts and fault handlers (clear FAULTMASK).
8.3 DMB

Data Memory Barrier.

8.3.1 Syntax

\texttt{DMB(\textit{cond})}

where:

\textit{cond} is an optional condition code. See Table 1-2 on page 22.

8.3.2 Operation

DMB acts as a data memory barrier. It ensures that all explicit memory accesses that appear before the \texttt{DMB} instruction (in program order) are completed before any explicit memory accesses that appear after the \texttt{DMB} instruction (in program order). \texttt{DMB} does not affect the ordering or execution of instructions that do not access memory.

8.3.3 Condition Flags

This instruction does not change the flags.

8.3.4 Examples

\texttt{DMB ; Data Memory Barrier}
8.4 DSB
Data Synchronization Barrier.

8.4.1 Syntax

\[
\text{DSB}(\ cond)\]

where:

\( cond \)

Is an optional condition code. See Table 1-2 on page 22.

8.4.2 Operation

\text{DSB} acts as a special data synchronization memory barrier. Instructions that come after \text{DSB} (in program order) do not execute until the \text{DSB} instruction completes. The \text{DSB} instruction completes when all explicit memory accesses before it complete.

8.4.3 Condition Flags

This instruction does not change the flags.

8.4.4 Examples

\[
\text{DSB} ; \ Data \ Synchronization \ Barrier
\]
8.5 **ISB**

Instruction Synchronization Barrier.

8.5.1 **Syntax**

\[
\text{ISB}(\text{cond})
\]

where:

\[
\text{cond}
\]

Is an optional condition code. See Table 1-2 on page 22.

8.5.2 **Operation**

ISB acts as an instruction synchronization barrier. It flushes the pipeline of the processor, so that all instructions following the ISB are fetched from cache or memory again, after the ISB instruction has been completed.

8.5.3 **Condition Flags**

This instruction does not change the flags.

8.5.4 **Examples**

\[
\text{ISB} \; ; \; \text{Instruction Synchronization Barrier}
\]
8.6  **MRS**
Move the contents of a special register to a general-purpose register.

### 8.6.1 Syntax

\[
\text{MRS}\{\text{cond}\} \ Rd, \ spec\_reg
\]

where:

- **cond**
  Is an optional condition code. See Table 1-2 on page 22.

- **Rd**
  Is the destination register.

- **spec\_reg**
  Can be any of the following special registers: APSR, IPSR, EPSR, IEPSR, IAPSR, EAPSR, PSR, MSP, PSP, PRIMASK, BASEPRI, BASEPRI\_MAX, FAULTMASK, or CONTROL.

### 8.6.2 Operation

Use **MRS** in combination with **MSR** as part of a read-modify-write sequence for updating a **PSR**, for example to clear the Q flag.

In process swap code, the programmers model state of the process being swapped out must be saved, including relevant **PSR** contents. Similarly, the state of the process being swapped in must also be restored. These operations use **MRS** in the state-saving instruction sequence and **MSR** in the state-restoring instruction sequence.

**Note:** **BASEPRI\_MAX** is an alias of **BASEPRI** when used with the **MRS** instruction.

See also “**MSR**” on page 89.

### 8.6.3 Restrictions

**Rd** must not be **SP** and must not be **PC**.

### 8.6.4 Condition Flags

This instruction does not change the flags.

### 8.6.5 Examples

\[
\text{MRS} \ R0, \ PRIMASK \ ; \ Read \ PRIMASK \ value \ and \ write \ it \ to \ R0.
\]
8.7 MSR
Move the contents of a general-purpose register to a special register.

8.7.1 Syntax

MSR\{cond\} spec_reg, Rn

where:

cond
Is an optional condition code. See Table 1-2 on page 22.

Rn
Is the source register.

spec_reg
Can be any of: APSR, IPSR, EPSR, IEPSR, IAPSR, EAPSR, PSR, MSP, PSP, PRIMASK, BASEPRI, BASEPRI_MAX, FAULTMASK, or CONTROL.

8.7.2 Operation

The register access operation in MSR depends on the privilege level. Unprivileged software can only access the Application Program Status Register (APSR) (see APSR in the Stellaris® Data Sheet). Privileged software can access all special registers.

In unprivileged software writes to unallocated or execution state bits in the PSR are ignored.

Note: When you write to BASEPRI_MAX, the instruction writes to BASEPRI only if either:

- Rn is non-zero and the current BASEPRI value is 0.
- Rn is non-zero and less than the current BASEPRI value.

See also “MRS” on page 88.

8.7.3 Restrictions

Rn must not be SP and must not be PC.

8.7.4 Condition Flags

This instruction updates the flags explicitly based on the value in Rn.

8.7.5 Examples

MSR CONTROL, R1 ; Read R1 value and write it to the CONTROL register.
8.8 **NOP**
No Operation.

8.8.1 **Syntax**

NOP\(\text{cond}\)

where:

\(\text{cond}\)
Is an optional condition code. See Table 1-2 on page 22.

8.8.2 **Operation**

NOP does nothing. NOP is not necessarily a time-consuming NOP. The processor might remove it from the pipeline before it reaches the execution stage.

Use NOP for padding, for example to place the following instruction on a 64-bit boundary.

8.8.3 **Condition Flags**

This instruction does not change the flags.

8.8.4 **Examples**

NOP ; No Operation
8.9 SEV

Send Event.

8.9.1 Syntax

$$SEV \{ \text{cond} \}$$

where:

$$\text{cond}$$

Is an optional condition code. See Table 1-2 on page 22.

8.9.2 Operation

SEV is a hint instruction that causes an event to be signaled to all processors within a multiprocessor system. It also sets the one-bit event register to 1. See "Power Management" in the Stellaris® Data Sheet.

8.9.3 Condition Flags

This instruction does not change the flags.

8.9.4 Examples

SEV ; Send Event
8.10 SVC
Supervisor Call.

8.10.1 Syntax

\[ \text{SVC}\{\text{cond}\} \ # \text{imm} \]

where:

*cond*

Is an optional condition code. See Table 1-2 on page 22.

*imm*

Is an expression evaluating to an integer in the range 0-255 (8-bit value).

8.10.2 Operation

The *SVC* instruction causes the *SVC* exception.

*imm* is ignored by the processor. If required, it can be retrieved by the exception handler to determine what service is being requested.

8.10.3 Condition Flags

This instruction does not change the flags.

8.10.4 Examples

\[ \text{SVC} \ 0x32 \ ; \text{ Supervisor Call (SVC handler can extract the immediate value by locating it via the stacked PC).} \]
8.11 WFE

Wait For Event.

8.11.1 Syntax

\[ \text{WFE} \{ \text{cond} \} \]

where:

\( \text{cond} \)

Is an optional condition code. See Table 1-2 on page 22.

8.11.2 Operation

\( \text{WFE} \) is a hint instruction.

If the one-bit event register is 0, \( \text{WFE} \) suspends execution until one of the following events occurs:

- An exception, unless masked by the exception mask registers (\text{PRIMASK}, \text{FAULTMASK}, and \text{BASEPRI}) or the current priority level.
- An exception enters the Pending state, if \text{SEVONPEND} in the \text{System Control Register (SCR)} is set.
- A Debug Entry request, if Debug is enabled.
- An event signaled by a peripheral or another processor in a multiprocessor system using the \text{SEV} instruction.

If the event register is 1, \( \text{WFE} \) clears it to 0 and returns immediately.

For more information, see "Power Management" in the \textit{Stellaris® Data Sheet}.

8.11.3 Condition Flags

This instruction does not change the flags.

8.11.4 Examples

\[ \text{WFE} \ ; \ \text{Wait for Event} \]
8.12 WFI
Wait for Interrupt.

8.12.1 Syntax

\[ \text{WFI}(\text{cond}) \]

where:

\[ \text{cond} \]

Is an optional condition code. See Table 1-2 on page 22.

8.12.2 Operation

WFI is a hint instruction that suspends execution until one of the following events occurs:

- An exception.
- A Debug Entry request, regardless of whether Debug is enabled.

8.12.3 Condition Flags

This instruction does not change the flags.

8.12.4 Examples

WFI ; Wait for Interrupt
**IMPORTANT NOTICE**

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI’s terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of TI. Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are neither designed nor intended for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and use of TI products. Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products which TI has not designated as military-grade, unless specified by the Buyer. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or “enhanced plastic.” Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use. TI products are neither designed nor intended for use in automotive applications or environments unless the TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

<table>
<thead>
<tr>
<th>Products</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amplifiers</td>
<td>Audio</td>
</tr>
<tr>
<td>Data Converters</td>
<td>Automotive</td>
</tr>
<tr>
<td>DLP® Products</td>
<td>Communications and Telecom</td>
</tr>
<tr>
<td>DSP</td>
<td>Computers and Peripherals</td>
</tr>
<tr>
<td>Clocks and Timers</td>
<td>Consumer Electronics</td>
</tr>
<tr>
<td>Interface</td>
<td>Energy</td>
</tr>
<tr>
<td>Logic</td>
<td>Industrial</td>
</tr>
<tr>
<td>Power Mgmt</td>
<td>Medical</td>
</tr>
<tr>
<td>Microcontrollers</td>
<td>Security</td>
</tr>
<tr>
<td>RFID</td>
<td>Space, Avionics &amp; Defense</td>
</tr>
<tr>
<td>RF/IF and ZigBee® Solutions</td>
<td>Video and Imaging</td>
</tr>
<tr>
<td></td>
<td>Wireless</td>
</tr>
</tbody>
</table>

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2010, Texas Instruments Incorporated