<table>
<thead>
<tr>
<th>Opcode</th>
<th>Page</th>
<th>Opcode</th>
<th>Page</th>
<th>Opcode</th>
<th>Page</th>
<th>Opcode</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>4</td>
<td>&lt;lit&gt;</td>
<td>20</td>
<td>Cl INC</td>
<td>37</td>
<td>DROP</td>
<td>54</td>
</tr>
<tr>
<td>!_INC</td>
<td>4</td>
<td>&lt;loop&gt;</td>
<td>21</td>
<td>C+!</td>
<td>38</td>
<td>DROP_LIT</td>
<td>55</td>
</tr>
<tr>
<td>+</td>
<td>5</td>
<td>&lt;next&gt;</td>
<td>22</td>
<td>C@</td>
<td>38</td>
<td>DROT</td>
<td>55</td>
</tr>
<tr>
<td>+!</td>
<td>5</td>
<td>&lt;short&gt;</td>
<td>23</td>
<td>C@_INC</td>
<td>39</td>
<td>DS!</td>
<td>56</td>
</tr>
<tr>
<td>+!_@</td>
<td>6</td>
<td>=</td>
<td>24</td>
<td>CMOVE</td>
<td>39</td>
<td>DS@</td>
<td>57</td>
</tr>
<tr>
<td>+!_C!</td>
<td>7</td>
<td>=branch</td>
<td>24</td>
<td>CONFIG!</td>
<td>40</td>
<td>DSWAP</td>
<td>57</td>
</tr>
<tr>
<td>+!_C@</td>
<td>7</td>
<td>&gt;</td>
<td>25</td>
<td>CONFIG@</td>
<td>41</td>
<td>DUP</td>
<td>58</td>
</tr>
<tr>
<td>+!_FLAGS</td>
<td>8</td>
<td>&gt;R</td>
<td>25</td>
<td>COUNT</td>
<td>41</td>
<td>DUP_0&lt;</td>
<td>58</td>
</tr>
<tr>
<td>–</td>
<td>8</td>
<td>?branch</td>
<td>26</td>
<td>C-_OR!</td>
<td>42</td>
<td>DUP_@</td>
<td>59</td>
</tr>
<tr>
<td>/STRING</td>
<td>10</td>
<td>?EXIT</td>
<td>27</td>
<td>D-</td>
<td>43</td>
<td>EXECUTE</td>
<td>60</td>
</tr>
<tr>
<td>0&lt;</td>
<td>10</td>
<td>?NEGATE</td>
<td>28</td>
<td>D=R</td>
<td>44</td>
<td>FALSE</td>
<td>60</td>
</tr>
<tr>
<td>0=</td>
<td>11</td>
<td>@</td>
<td>28</td>
<td>D@</td>
<td>45</td>
<td>FP!</td>
<td>62</td>
</tr>
<tr>
<td>0!=NOT</td>
<td>11</td>
<td>@!</td>
<td>29</td>
<td>DASR</td>
<td>46</td>
<td>FP+!</td>
<td>63</td>
</tr>
<tr>
<td>0&gt;</td>
<td>12</td>
<td>@+</td>
<td>29</td>
<td>DBASE!</td>
<td>46</td>
<td>FP@</td>
<td>63</td>
</tr>
<tr>
<td>1+</td>
<td>12</td>
<td>@@</td>
<td>30</td>
<td>DBASE+!</td>
<td>47</td>
<td>FRAME_POP</td>
<td>64</td>
</tr>
<tr>
<td>1-</td>
<td>13</td>
<td>@_INC</td>
<td>30</td>
<td>DBASE+!</td>
<td>47</td>
<td>FRAME_PUSH</td>
<td>64</td>
</tr>
<tr>
<td>2*</td>
<td>13</td>
<td>@_LIT+</td>
<td>31</td>
<td>DBASE+_@</td>
<td>48</td>
<td>HI</td>
<td>65</td>
</tr>
<tr>
<td>2/5</td>
<td>14</td>
<td>ABS</td>
<td>31</td>
<td>DBASE@</td>
<td>48</td>
<td>H@</td>
<td>65</td>
</tr>
<tr>
<td>2&gt;PICK</td>
<td>14</td>
<td>ADC</td>
<td>32</td>
<td>DDROP</td>
<td>49</td>
<td>HALT</td>
<td>66</td>
</tr>
<tr>
<td>3.Roll</td>
<td>15</td>
<td>AND</td>
<td>32</td>
<td>DDUP</td>
<td>49</td>
<td>I’</td>
<td>66</td>
</tr>
<tr>
<td>4+</td>
<td>15</td>
<td>AND!</td>
<td>33</td>
<td>DISABLE</td>
<td>50</td>
<td>I’!</td>
<td>67</td>
</tr>
<tr>
<td>4-!</td>
<td>16</td>
<td>ASIC!</td>
<td>34</td>
<td>DLSL</td>
<td>50</td>
<td>I’-4-!</td>
<td>67</td>
</tr>
<tr>
<td>4-@</td>
<td>16</td>
<td>ASIC@</td>
<td>34</td>
<td>DLSR</td>
<td>51</td>
<td>I’-_@</td>
<td>68</td>
</tr>
<tr>
<td>&lt;+loop&gt;</td>
<td>17</td>
<td>ASR</td>
<td>35</td>
<td>DNEGATE</td>
<td>51</td>
<td>I’+</td>
<td>68</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>18</td>
<td>B@</td>
<td>35</td>
<td>docon</td>
<td>52</td>
<td>J</td>
<td>69</td>
</tr>
<tr>
<td>&lt;cold&gt;</td>
<td>18</td>
<td>BIT-CLEAR</td>
<td>36</td>
<td>dovar</td>
<td>52</td>
<td>LEAVE</td>
<td>70</td>
</tr>
<tr>
<td>&lt;do&gt;</td>
<td>19</td>
<td>branch</td>
<td>36</td>
<td>DOVER</td>
<td>53</td>
<td>LOAD_DS</td>
<td>70</td>
</tr>
<tr>
<td>&lt;Interrupt&gt;</td>
<td>19</td>
<td>Cl</td>
<td>37</td>
<td>DR&gt;</td>
<td>53</td>
<td>LOAD_RS</td>
<td>71</td>
</tr>
<tr>
<td>Opcode</td>
<td>Page</td>
<td>Opcode</td>
<td>Page</td>
<td>Opcode</td>
<td>Page</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>------</td>
<td>------------</td>
<td>------</td>
<td>------------</td>
<td>------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOC_!</td>
<td>72</td>
<td>R&gt;</td>
<td>89</td>
<td>SWAP_!</td>
<td>106</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOC_+!</td>
<td>72</td>
<td>R&gt;!</td>
<td>90</td>
<td>SWAP_@</td>
<td>106</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOC_@</td>
<td>73</td>
<td>R&gt; @</td>
<td>90</td>
<td>SWAP_@!</td>
<td>107</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOC_@!</td>
<td>73</td>
<td>R@</td>
<td>91</td>
<td>SWAP_@C</td>
<td>107</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOC_@_+</td>
<td>74</td>
<td>R@!</td>
<td>91</td>
<td>ROT_+</td>
<td>108</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOC_@@</td>
<td>74</td>
<td>R@@<em>4-</em>!</td>
<td>92</td>
<td>SWAP_OVER_!</td>
<td>108</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOC_B@</td>
<td>75</td>
<td>R@ @</td>
<td>92</td>
<td>TEST_AND_SET</td>
<td>109</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOC_C!</td>
<td>75</td>
<td>RLC</td>
<td>93</td>
<td>TEST_UNDER_MASK</td>
<td>110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOC_C@</td>
<td>76</td>
<td>ROLL</td>
<td>93</td>
<td>TRUE</td>
<td>110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSLN</td>
<td>76</td>
<td>ROT</td>
<td>94</td>
<td>TUCK</td>
<td>111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSRN</td>
<td>77</td>
<td>RP!</td>
<td>95</td>
<td>U&gt;</td>
<td>111</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSR</td>
<td>78</td>
<td>RPLIM!</td>
<td>96</td>
<td>UDNORMAIZE</td>
<td>112</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M+</td>
<td>78</td>
<td>RPLIM@</td>
<td>96</td>
<td>UM*</td>
<td>113</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVE</td>
<td>79</td>
<td>RPLIM</td>
<td>97</td>
<td>UM/MOD</td>
<td>115</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MRAM!</td>
<td>80</td>
<td>RRC</td>
<td>97</td>
<td>UNORMALIZE</td>
<td>116</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MRAM@</td>
<td>80</td>
<td>RTI</td>
<td>97</td>
<td>WAIT</td>
<td>117</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MROM@</td>
<td>81</td>
<td>S&gt;D</td>
<td>98</td>
<td>WFILL</td>
<td>118</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEGATE</td>
<td>81</td>
<td>SBASE!</td>
<td>98</td>
<td>XOR</td>
<td>118</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NIP</td>
<td>82</td>
<td>SBASE+!</td>
<td>99</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOP</td>
<td>82</td>
<td>SBASE+!</td>
<td>99</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOT</td>
<td>83</td>
<td>SBASE+ @</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>not?branch</td>
<td>83</td>
<td>SBASE@</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ONE</td>
<td>84</td>
<td>SPI</td>
<td>101</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR</td>
<td>84</td>
<td>SP!</td>
<td>101</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR!</td>
<td>85</td>
<td>SPLIM!</td>
<td>102</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OVER</td>
<td>85</td>
<td>SPLIM@</td>
<td>102</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OVER_!</td>
<td>86</td>
<td>STORE_DS</td>
<td>103</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OVER_+</td>
<td>86</td>
<td>STORE_RS</td>
<td>103</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OVER_@</td>
<td>87</td>
<td>SWAP</td>
<td>104</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R+!</td>
<td>87</td>
<td>SWAP_!</td>
<td>105</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PICK</td>
<td>88</td>
<td>SWAP_</td>
<td>105</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Value</td>
<td>Opcode</td>
<td>Value</td>
<td>Opcode</td>
<td>Value</td>
<td>Opcode</td>
<td>Value</td>
<td>Opcode</td>
</tr>
<tr>
<td>--------</td>
<td>---------</td>
<td>--------</td>
<td>---------</td>
<td>--------</td>
<td>---------</td>
<td>--------</td>
<td>---------</td>
</tr>
<tr>
<td>0x00</td>
<td>NOP</td>
<td>0x20</td>
<td>?branch</td>
<td>0x3F</td>
<td>OVER</td>
<td>0x6A</td>
<td>CONFIG!</td>
</tr>
<tr>
<td>0x01</td>
<td>&lt;interrupt&gt;</td>
<td>0x21</td>
<td>@</td>
<td>0x40</td>
<td>OVER_QUICK</td>
<td>0x6B</td>
<td>ENABLE</td>
</tr>
<tr>
<td>0x03</td>
<td>&lt;trap&gt;</td>
<td>0x22</td>
<td>AND</td>
<td>0x41</td>
<td>R&gt;</td>
<td>0x6C</td>
<td>DISABLE</td>
</tr>
<tr>
<td>0x04</td>
<td>&lt;cold&gt;</td>
<td>0x23</td>
<td>AND_QUICK</td>
<td>0x42</td>
<td>R&gt;_QUICK</td>
<td>0x6D</td>
<td>1</td>
</tr>
<tr>
<td>0x05</td>
<td>HALT</td>
<td>0x24</td>
<td>ASR</td>
<td>0x43</td>
<td>RTI</td>
<td>0x6E</td>
<td>1+</td>
</tr>
<tr>
<td>0x06</td>
<td>NOP_QUICK</td>
<td>0x25</td>
<td>branch</td>
<td>0x44</td>
<td>.UM</td>
<td>0x6F</td>
<td>1-</td>
</tr>
<tr>
<td>0x07</td>
<td>.0&lt;</td>
<td>0x26</td>
<td>.C!</td>
<td>0x45</td>
<td>SWAP</td>
<td>0x70</td>
<td>.2*</td>
</tr>
<tr>
<td>0x08</td>
<td>.0&lt;</td>
<td>0x27</td>
<td>.C@</td>
<td>0x46</td>
<td>SWAP_QUICK</td>
<td>0x71</td>
<td>.2*_QUICK</td>
</tr>
<tr>
<td>0x09</td>
<td>NOT</td>
<td>0x28</td>
<td>B@</td>
<td>0x47</td>
<td>.U&gt;</td>
<td>0x72</td>
<td>.4+</td>
</tr>
<tr>
<td>0x0A</td>
<td>.0=</td>
<td>0x29</td>
<td>D!</td>
<td>0x48</td>
<td>.UM*</td>
<td>0x73</td>
<td>.+!</td>
</tr>
<tr>
<td>0x0B</td>
<td>2/</td>
<td>0x2A</td>
<td>D+</td>
<td>0x51</td>
<td>.UM/MOD</td>
<td>0x74</td>
<td>&lt;do&gt;</td>
</tr>
<tr>
<td>0x0C</td>
<td>.1</td>
<td>0x2B</td>
<td>D&gt;R</td>
<td>0x56</td>
<td>XOR</td>
<td>0x75</td>
<td>DDROP</td>
</tr>
<tr>
<td>0x0D</td>
<td>SPI!</td>
<td>0x2C</td>
<td>D@</td>
<td>0x57</td>
<td>XOR_QUICK</td>
<td>0x76</td>
<td>DDUP</td>
</tr>
<tr>
<td>0x0E</td>
<td>SP@</td>
<td>0x2D</td>
<td>DASR</td>
<td>0x58</td>
<td>ASIC!</td>
<td>0x77</td>
<td>NIP</td>
</tr>
<tr>
<td>0x0F</td>
<td>RP!</td>
<td>0x2E</td>
<td>docen</td>
<td>0x59</td>
<td>LITASIC!</td>
<td>0x78</td>
<td>NIP_QUICK</td>
</tr>
<tr>
<td>0x10</td>
<td>RP@</td>
<td>0x2F</td>
<td>dovar</td>
<td>0x5A</td>
<td>.ASIC@</td>
<td>0x79</td>
<td>TUCK</td>
</tr>
<tr>
<td>0x11</td>
<td>.+</td>
<td>0x30</td>
<td>DR&gt;</td>
<td>0x5B</td>
<td>.LITASIC@</td>
<td>0x7A</td>
<td>.=</td>
</tr>
<tr>
<td>0x12</td>
<td>+_QUICK</td>
<td>0x31</td>
<td>DROP</td>
<td>0x5C</td>
<td>DBASE!</td>
<td>0x7B</td>
<td>&lt;&gt;</td>
</tr>
<tr>
<td>0x13</td>
<td>-1</td>
<td>0x32</td>
<td>DROP_QUICK</td>
<td>0x5D</td>
<td>DBASE@</td>
<td>0x7C</td>
<td>?DUP</td>
</tr>
<tr>
<td>0x14</td>
<td>TRUE</td>
<td>0x33</td>
<td>DUP</td>
<td>0x5E</td>
<td>.SPLIM!</td>
<td>0x7D</td>
<td>.ABS</td>
</tr>
<tr>
<td>0x15</td>
<td>&lt;lit&gt;</td>
<td>0x34</td>
<td>DUP_QUICK</td>
<td>0x5F</td>
<td>.SPLIM@</td>
<td>0x7E</td>
<td>DSL</td>
</tr>
<tr>
<td>0x16</td>
<td>&lt;short&gt;</td>
<td>0x35</td>
<td>.EXECUTE</td>
<td>0x60</td>
<td>.FP!</td>
<td>0x7F</td>
<td>DOVER</td>
</tr>
<tr>
<td>0x17</td>
<td>LIT+</td>
<td>0x36</td>
<td>.HI!</td>
<td>0x61</td>
<td>.FP@</td>
<td>0x80</td>
<td>DSWAP</td>
</tr>
<tr>
<td>0x18</td>
<td>&lt;short&gt;_QUICK</td>
<td>0x37</td>
<td>.H@</td>
<td>0x62</td>
<td>.MRAM!</td>
<td>0x81</td>
<td>.DROT</td>
</tr>
<tr>
<td>0x19</td>
<td>LIT+_QUICK</td>
<td>0x38</td>
<td>.R@</td>
<td>0x63</td>
<td>.MRAM@</td>
<td>0x83</td>
<td>.S&gt;D</td>
</tr>
<tr>
<td>0x1A</td>
<td>&lt;loop&gt;</td>
<td>0x39</td>
<td>.R@_QUICK</td>
<td>0x64</td>
<td>.RPLIM!</td>
<td>0x84</td>
<td>NEGATE</td>
</tr>
<tr>
<td>0x1C</td>
<td>.PICK</td>
<td>0x3A</td>
<td>.I+</td>
<td>0x65</td>
<td>.RPLIM@</td>
<td>0x85</td>
<td>DNEGATE</td>
</tr>
<tr>
<td>0x1D</td>
<td>ROLL</td>
<td>0x3B</td>
<td>.I'</td>
<td>0x66</td>
<td>SBASE!</td>
<td>0x86</td>
<td>.D-</td>
</tr>
<tr>
<td>0x1E</td>
<td>.&gt;</td>
<td>0x3C</td>
<td>.J</td>
<td>0x67</td>
<td>SBASE@</td>
<td>0x87</td>
<td>/STRING</td>
</tr>
<tr>
<td>0x1F</td>
<td>.&gt;R</td>
<td>0x3D</td>
<td>OR</td>
<td>0x68</td>
<td>.WAIT</td>
<td>0x88</td>
<td>.CMOVE</td>
</tr>
</tbody>
</table>

Harris Semiconductor Proprietary
Opcode | Value  
---|---
0x8C  | FILL  
0x8D  | WFILL 
0x8E  | COUNT 
0x8F  | M+    
0x90  | 0= NOT 
0x91  | >     
0x92  | 2 PICK 
0x93  | 3 ROLL 
0x94  | .ROT 
0x95  | .-!   
0x96  | .-@   
0x97  | .NEGATE 
0x98  | .not?branch 
0x99  | =branch 
0x9A  | @@   
0x9B  | @!   
0x9C  | .@ +  
0x9D  | @_LIT+ 
0x9E  | .ADC  
0x9F  | .AND! 
0xA0  | OR! 
0xA1  | C+I 
0xA2  | .C_OR! 
0xA3  | ?DNEGATE 
0xA4  | .D 
0xA5  | .D= 
0xA6  | DLSR 
0xA7  | .LIT_NIP 
0xA8  | .DUP_0< 
0xA9  | .DUP_@ 
0xAA  | .SWAP_0< 
0xAD  | .SWAP_@ 

Opcode | Value  
---|---
0xAE  | LIT_AND_QUICK 
0xB0  | LIT_OR_QUICK 
0xB1  | LIT_! 
0xB2  | LIT_@ 
0xB3  | LSLN 
0xB4  | LIT_LSLN 
0xB5  | LSR 
0xB6  | LSRN 
0xB7  | LIT_LSRN 
0xB8  | OVER_! 
0xB9  | OVER_@ 
0xBA  | OVER_+ 
0xBB  | RLC 
0xBC  | .RLC 
0xBE  | .RAC 
0xBF  | .R@_! 
0xC0  | R>@ 
0xC1  | LIT_R+! 
0xC2  | .LIT_ROT_+ 
0xC3  | .BIT-CLEAR 
0xC4  | .+_FLAGS 
0xC5  | .DS@ 
0xC6  | .DSI 
0xC7  | TEST_UNDER 
0xC8  | .FETCH_AND 
0xC9  | .LIT_FETCH 
0xCA  | .TEST_AND_SET 
0xCB  | .LIT_LOC_@ 
0xCD  | .LIT_LOC_C@ 
0xCE  | .LIT_LOC_B@ 
0xCF  | .LIT_LOC_@ + 
0xD0  | .LIT_LOC_! 
0xD1  | .LIT_LOC_@ + 
0xD2  | .LIT_LOC_@ + 
0xD3  | .LIT_LOC_@ + 
0xD4  | .LIT_LOC_@ +! 
0xD5  | .LIT_LOC_@ +! 
0xD6  | .LIT_LOC_@ +! 
0xD7  | .LIT_LOC_@ +! 
0xD8  | .LIT_LOC_@ +! 
0xD9  | .LIT_LOC_@ +! 
0xDA  | .LIT_LOC_@ +! 
0xDB  | .LOAD_DS 
0xDC  | .LOAD_RS 
0xDD  | .STORE_DS 
0xDE  | .STORE_RS 
0xDF  | .SWAP_! 
0xE0  | .?EXIT 
0xE1  | .<next> 
0xE2  | .I'! 
0xE3  | .I'! @ 
0xE4  | .R@ 4! 
0xE5  | .R@ 4! 
0xE6  | .SWAP_! 
0xE7  | .SWAP_! 
0xE8  | .FP_+! 
0xE9  | .LIT_FP_+! 
0xEA  | .SBASE_+! 
0xEB  | .DBASE_+! 
0xEC  | .FRAME_PUSH 
0xED  | .FRAME_POP 
0xEE  | .LIT_@ inc 
0xEF  | .LIT_C@ inc 
0xF0  | .LIT_! inc 
0xF1  | .LIT_C! inc 
0xF2  | .LEAVE 
0xF3  | .+ @ 
0xF4  | .LIT_+ @ 
0xF5  | .+ C@ 
0xF6  | .LIT_+ C@ 
0xF7  | .SWAP_! 
0xF8  | .SWAP_ C@ 
0xF9  | .+ ! 
0xFA  | .+ C! 
0xFB  | .LIT_+ ! 
0xFC  | .LIT_+ C! 
0xFD  | .SWAP_! 
0xFE  | .SWAP_ C! 
0xFF  | .-3_ROLL 

Harris Semiconductor Proprietary
This is a preliminary document. While reasonable effort has been made to ensure its accuracy, it probably has minor errors, and is subject to change.

Please report any errors or omissions so that they may be corrected in the final version.

This document lists the BINAR instruction set as used with the target-resident Forth compiler. The opcodes are designed to represent the operation of the machine conveniently for Forth programmers. The "assembly language" instruction set for use with other language will use different naming conventions and syntax.

Notation:
Addresses are represented by "addr".
Signed 32-bit integers are represented by "n". 32-bit integers treated as unsigned quantities are represented by "un".
Signed 64-bit integers are represented by "d". They are placed on the stack as a pair of 32-bit values, with the more significant 32 bits on the top of the stack with respect to the least significant 32 bits. 64-bit integers treated as unsigned quantities are represented by "ud".
Signed 16-bit integers are represented by "h". 16-bit integers treated as unsigned quantities are represented by "uh".
Signed 8-bit integers are represented by "b". 8-bit integers treated as unsigned quantities (e.g. characters) are represented by "c".
Logical flags are represented by "flag". Flags input to an opcode are true if non-zero. Flags output by opcodes are 0 if false, -1 if true.

All quantities are 32-bit signed integers unless otherwise specified. Bits are numbered from lowest to highest, with bit 0 being the lowest order bit, and bit 31 being the highest order bit of a 32-bit quantity (the sign bit).

Each opcode starts at the top of a new page. The mnemonic is the Forth notation for the opcode. Below the header is a stack picture which shows the inputs and outputs of the opcode. As an example, the word with the following stack picture:

( nl b2 addr3 n4 un5 )

takes three inputs on the data stack: a signed 32-bit integer n1, a signed 8-bit integer b2, and an address addr3, with addr3 on the top of the stack, b2 as the second element on the stack, and n1 as the third element on the stack. The word returns two quantities on the data stack: a signed 32-bit integer n4 as the second element on the stack, and an unsigned 32-bit integer un5 on the top of the stack. In words which affect the return stack, the notation "RS(→)") is used in a similar manner. The notation "Immediate" to the right of a stack notation means that that stack picture applies to the immediate operand mode of the instruction, using the literal field to supply a constant value.

Some opcodes require address or literal fields for proper operation. When the encoding notation includes "/lit", then the short or long literal instruction fields must be used with the opcode, mandating use of the JNEXT, EXIT, or 2OPS instruction formats. Literal fields are used as sign-extended integers. When the encoding notation includes "/call", then the CALL instruction format must be used, and bits 2-22 of the word aligned address of the call target (typically a conditional branch target) must be placed in bits 2-22 of the CALL instruction. Encodings that are annotated "(2OPS format)" may only be used in the
2OPS format, but provide single-cycle instruction execution.

Some opcodes have variants that allow the use of either a literal field or a data stack value for operation. In these cases, the description is written for the variant that uses the data stack, with appropriate annotation for the behavior of the literal field.

Words that have an effect on the Return Stack should only be used in JNEXT or 2OPS instruction formats unless otherwise noted. Return Stack manipulation words can interfere with the operation of subroutine calls and returns in subtle ways that are not obvious to the casual user.

Alignment:

All word addresses are forced to be word aligned by ignoring the bottom two bits (i.e., bits 0 and 1) and substituting zeros.

All half-word addresses are forced to be half-word aligned by ignoring the bottom bit (i.e., bit 0) and substituting a zero.

Cautions:

The instruction set and its implementation can and will be changed somewhat on the next version of the silicon. Therefore, code that exploits the particular numeric value of an opcode or any unusual properties of its interpretation must be avoided. Code should be written so that replacing an opcode with an equivalent subroutine call will have no adverse effects. This is extremely easy to do on a stack machine, and so should present no real problem.

The “conventional” mnemonics are very preliminary, and are subject to change without notice during the implementation of the RTX assembler. The “Forth” mnemonics are reasonably stable, and track words available in the target-resident Forth compiler being developed by Harris.
**BINAR QUICK REFERENCE**

<table>
<thead>
<tr>
<th>DC</th>
<th>MP</th>
<th>MAD</th>
<th>COND</th>
<th>DLO</th>
<th>DHI</th>
<th>CIN</th>
<th>ALU</th>
<th>SHIFT</th>
<th>RP</th>
<th>DP</th>
<th>DESTINATION</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DECODE CONTROL (30):</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: nop (default)</td>
</tr>
<tr>
<td>1: DECODE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INCREMENT MPC (29):</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: nop (default)</td>
</tr>
<tr>
<td>1: INC (MPC)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MICRO-ADDRESS (27-28):</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: JMP = 00x</td>
</tr>
<tr>
<td>1: JMP = 01x</td>
</tr>
<tr>
<td>2: JMP = 10x</td>
</tr>
<tr>
<td>3: JMP = 11x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CONDITION CODE (24-26):</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: JMP = xx0 always 0</td>
</tr>
<tr>
<td>1: JMP = xxC Carry out</td>
</tr>
<tr>
<td>2: JMP = xxZ Zero</td>
</tr>
<tr>
<td>3: JMP = xxS Sign</td>
</tr>
<tr>
<td>4: JMP = xxL Lowest bit of DLO</td>
</tr>
<tr>
<td>5: JMP = xxV overflow</td>
</tr>
<tr>
<td>6: JMP = xxP Pending Interrupt</td>
</tr>
<tr>
<td>7: JMP = xx1 always 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DLO CONTROL (22-23):</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: nop (DLO) (default)</td>
</tr>
<tr>
<td>1: SR(DLO)</td>
</tr>
<tr>
<td>2: SL(DLO)</td>
</tr>
<tr>
<td>3: DEST = DLO</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DHI (21):</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: DHI[0] (default)</td>
</tr>
<tr>
<td>1: DHI[1]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CIN (20):</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: CIN = 0 (default)</td>
</tr>
<tr>
<td>1: CIN = 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ALU FUNCTION (16-19):</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIN=0 CIN=1</td>
</tr>
<tr>
<td>0: A (default)</td>
</tr>
<tr>
<td>1: A or B</td>
</tr>
<tr>
<td>2: A + A</td>
</tr>
<tr>
<td>3: A + B</td>
</tr>
<tr>
<td>4: A and B</td>
</tr>
<tr>
<td>5: B</td>
</tr>
<tr>
<td>6: A + B</td>
</tr>
<tr>
<td>7: A + 0</td>
</tr>
<tr>
<td>8: A - 1</td>
</tr>
<tr>
<td>9: A xor B</td>
</tr>
<tr>
<td>10: not B</td>
</tr>
<tr>
<td>11: A and B</td>
</tr>
<tr>
<td>12: 0</td>
</tr>
<tr>
<td>13: A - B - 1</td>
</tr>
<tr>
<td>14: A nor B</td>
</tr>
<tr>
<td>15: not A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ALU SHIFT (14-15):</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: PASS(ALU) (default)</td>
</tr>
<tr>
<td>1: SR(ALU)</td>
</tr>
<tr>
<td>2: SL(ALU)</td>
</tr>
<tr>
<td>3: FP (pass FPU output)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RP CONTROL (12-13):</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: DEC[R]</td>
</tr>
<tr>
<td>1: INC[R]</td>
</tr>
<tr>
<td>2: nop[R] (default)</td>
</tr>
<tr>
<td>3: DEST = RP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DP CONTROL (10-11):</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: DEC[DP]</td>
</tr>
<tr>
<td>1: INC[DP]</td>
</tr>
<tr>
<td>2: nop[DP] (default)</td>
</tr>
<tr>
<td>3: DEST = DP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DESTINATION (5-9):</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: none (default)</td>
</tr>
<tr>
<td>1: DEST = HOST</td>
</tr>
<tr>
<td>2: DEST = DS</td>
</tr>
<tr>
<td>3: DEST = DP-LIMIT</td>
</tr>
<tr>
<td>4: DEST = RS</td>
</tr>
<tr>
<td>5: DEST = RP-LIMIT</td>
</tr>
<tr>
<td>6: DEST = DHI[0]</td>
</tr>
<tr>
<td>7: DEST = DHI[1]</td>
</tr>
<tr>
<td>8: DEST = MRAM</td>
</tr>
<tr>
<td>9: DEST = MICRO-ADR</td>
</tr>
<tr>
<td>10: DEST = DS-FROM-DHI</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SOURCE (0-4):</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: SOURCE = HOST</td>
</tr>
<tr>
<td>1: SOURCE = LIT</td>
</tr>
<tr>
<td>2: SOURCE = PAGE/NAR/CTL</td>
</tr>
<tr>
<td>3: SOURCE = ADDR</td>
</tr>
<tr>
<td>4: SOURCE = MRAM</td>
</tr>
<tr>
<td>5: SOURCE = MRAM</td>
</tr>
<tr>
<td>6: SOURCE = SBASE</td>
</tr>
<tr>
<td>7: SOURCE = DBASE</td>
</tr>
<tr>
<td>8: SOURCE = FP</td>
</tr>
<tr>
<td>9: SOURCE = MIS</td>
</tr>
<tr>
<td>10: SOURCE = RP</td>
</tr>
<tr>
<td>11: SOURCE = DP</td>
</tr>
<tr>
<td>12: SOURCE = RS</td>
</tr>
<tr>
<td>13: SOURCE = RETURN-SAVE</td>
</tr>
<tr>
<td>14: SOURCE = CONFIG</td>
</tr>
<tr>
<td>15: SOURCE = I-LATCH</td>
</tr>
<tr>
<td>16: SOURCE = DHI[0]</td>
</tr>
<tr>
<td>17: SOURCE = DHI[1]</td>
</tr>
<tr>
<td>18: SOURCE = RP-LIMIT</td>
</tr>
<tr>
<td>19: SOURCE = DP-LIMIT</td>
</tr>
<tr>
<td>20: SOURCE = DLO</td>
</tr>
<tr>
<td>21: SOURCE = -1</td>
</tr>
<tr>
<td>22: SOURCE = 0</td>
</tr>
<tr>
<td>23: SOURCE = 4</td>
</tr>
<tr>
<td>24: SOURCE = unused-24</td>
</tr>
<tr>
<td>25: SOURCE = DS</td>
</tr>
<tr>
<td>26: MULTIPLE-STEP</td>
</tr>
<tr>
<td>27: DIVIDE</td>
</tr>
<tr>
<td>28: SOURCE = RD</td>
</tr>
<tr>
<td>29: SOURCE = RD-SIGNED</td>
</tr>
<tr>
<td>30: SOURCE = unused-30</td>
</tr>
<tr>
<td>31: SOURCE = unused-31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CALL</th>
<th>00</th>
<th>CALL ADDRESS</th>
<th>00</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXIT</td>
<td>01</td>
<td>LIT</td>
<td>01</td>
</tr>
<tr>
<td>2OPS</td>
<td>10</td>
<td>LITB</td>
<td>10</td>
</tr>
<tr>
<td>JNEXT</td>
<td>11</td>
<td>LIT</td>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MISCELLANEOUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>MISC</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CONFIG</th>
<th>DSO</th>
<th>RSO</th>
<th>DSU</th>
<th>RSU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00</td>
<td>00</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MASK</th>
</tr>
</thead>
<tbody>
<tr>
<td>00...00</td>
</tr>
</tbody>
</table>

---

**Harris Semiconductor Proprietary Information**
Store
(n1 addr2 → )
(n1 → ) Immediate

Encoding:
0x0C 4 cycles
0xB1 / lit 4 cycles

Operation:
Store n1 at address addr2. The store immediate variant uses the literal field to provide the value of addr2.

Implementation:
opcode: I
0: SOURCE=DHI
   ADDR=BUS+0(CYCLE) ;
1: SOURCE=DS INC[DP] DEST=RAM-1 ;
2: ;
3: SOURCE=DS INC[DP] ALU=B DECODE ;

opcode: LIT_1
0: SOURCE=LIT ADDR=BUS+0(CYCLE) ;
1: SOURCE=DHI DEST=RAM-1 ;
2: ;
3: SOURCE=DS INC[DP] ALU=B DECODE ;

Incrementing Store
(n1 addr2 → addr3 ) Immediate

Encoding: 0xF0 / lit 4 cycles

Operation:
Store value n1 at location addr2, then add the compiled instruction literal field to addr2, leaving addr3.

Implementation:
opcode: _INC
0: SOURCE=DHI
   ADDR=BUS+0(CYCLE) ;
1: SOURCE=DS INC[DP] DEST=RAM-1 ;
2: ;
3: SOURCE=LIT ALU=A+B DECODE ;
Add

\[(n_1 \ n_2 \rightarrow n_3) \]
\[(n_1 \rightarrow n_3)\] Immediate

Encoding:

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Immediate Encoding</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x11</td>
<td>2 cycles</td>
<td></td>
</tr>
<tr>
<td>0x12</td>
<td>1 cycle (2OPS format)</td>
<td></td>
</tr>
<tr>
<td>0x17 / lit</td>
<td>2 cycles</td>
<td></td>
</tr>
<tr>
<td>0x19 / lit</td>
<td>1 cycle (2OPS format)</td>
<td></td>
</tr>
</tbody>
</table>

Operation:
Add \(n_1\) and \(n_2\), giving \(n_3 = n_1 + n_2\). In the immediate variant, the value \(n_2\) is provided by the literal field.

Implementation:

```
opcode: +
  0: SOURCE=DS INC[DP] ALU=A+B ;
  1: DECODE ;

opcode: +_QUICK
  0: SOURCE=DS INC[DP] ALU=A+B DECODE ;

opcode: LIT_+
  0: SOURCE=LIT ALU=A+B ;
  1: DECODE ;

opcode: LIT+_QUICK
  0: SOURCE=LIT ALU=A+B DECODE ;
```

Add To Memory

\[(n_1 \ addr2 \rightarrow)\]

Encoding:

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Immediate Encoding</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x73</td>
<td>7 cycles</td>
<td></td>
</tr>
</tbody>
</table>

Operation:
Performs an atomic increment of the word at addr2 by value \(n_1\).

Implementation:

```
opcode: +
  0: SOURCE=DHI ADDR=BUS+0(CYCLE) ;
  1: RAM-RMW ;
  2: SOURCE=DS INC[DP] ALU=B ;
  3: SOURCE=RD ALU=A+B CYCLE-RAM ;
  4: SOURCE=DHI DEST=RAM-1 ;
  5: SOURCE=DS INC[DP] ALU=B ;
  6: DECODE ;
```

Harris Semiconductor Proprietary
Indexed Store
(n1 n2 addr3 \rightarrow )
(n1 addr3 \rightarrow ) Immediate

Encoding:
0xF9  5 cycles
0xFB / lit  5 cycles

Operation:
Add the offset n2 to addr3, then store the value n1 in that same memory location. In the immediate variant, the value n2 is taken from the literal field.

Implementation:
opcode: \_!_1
0: SOURCE=DS INC[DP] ALU=A+B ;;
1: SOURCE=DHI ADDR=BUS+0(CYCLE) ;;
2: SOURCE=DS INC[DP] DEST=RAM-1 ;;
3: ; ;
4: SOURCE=DS INC[DP] ALU=B DECODE ;;

opcode: LIT+_1
0: SOURCE=LIT ALU=A+B ;;
1: SOURCE=DHI ADDR=BUS+0(CYCLE) ;;
2: SOURCE=DS INC[DP] DEST=RAM-1 ;;
3: ; ;
4: SOURCE=DS INC[DP] ALU=B DECODE ;;

Indexed Load
(n1 addr2 \rightarrow n3 )
(addr2 \rightarrow n3 ) Immediate

Encoding:
0xF3  5 cycles
0xF4 / lit  5 cycles

Operation:
Add the offset n1 to addr2, then fetch the value n3 from that memory location. In the immediate variant, the value n1 is provided by the literal field.

Implementation:
opcode: \_!_@
0: SOURCE=DS INC[DP] ALU=A+B ;;
1: SOURCE=DHI ADDR=BUS+0(CYCLE) ;;
2: ; ;
3: ; ;
4: SOURCE=RD ALU=B DECODE ;;

opcode: LIT+_@
0: SOURCE=LIT ALU=A+B ;;
1: SOURCE=DHI ADDR=BUS+0(CYCLE) ;;
2: ; ;
3: ; ;
4: SOURCE=RD ALU=B DECODE ;;
**+_C!**

Indexed Store Character

( b1 n2 addr3 → )

( b1 addr3 → ) Immediate

Encoding:

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFA</td>
<td>5</td>
</tr>
<tr>
<td>0xFC / lit</td>
<td>5</td>
</tr>
</tbody>
</table>

Operation:

Add the offset n2 to addr3, then store the value b1 in that memory location. The highest 24 bits of the stack value for b1 are ignored. In the immediate variant, the value n2 is provided by the literal field.

Implementation:

- **opcode**: +_C!
  - 0: SOURCE=DS INC[DP] ALU=A+B ;
  - 1: SOURCE=DHI ADDR=BUS+0(CYCLE) ;
  - 2: SOURCE=DS INC[DP] DEST=RAM-C! ;
  - 3: ;
  - 4: SOURCE=D INC[DP] ALU=B DECODE ;

- **opcode**: LIT+_C!
  - 0: SOURCE=LIT ALU=A+B ;
  - 1: SOURCE=DHI ADDR=BUS+0(CYCLE) ;
  - 2: SOURCE=DS INC[DP] DEST=RAM-C! ;
  - 3: ;
  - 4: SOURCE=DS INC[DP] ALU=B DECODE ;

**+_C@**

Indexed Load Character

( n1 addr2 → c3 )

( addr2 → c3 ) Immediate

Encoding:

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xF5</td>
<td>5</td>
</tr>
<tr>
<td>0xF6 / lit</td>
<td>5</td>
</tr>
</tbody>
</table>

Operation:

Add the offset n1 to addr2, then fetch the value c3 from that memory location. c3 is zero-extended to form a 32-bit stack word. In the immediate variant, the value n1 is provided by the literal field.

Implementation:

- **opcode**: +_C@
  - 0: SOURCE=DS INC[DP] ALU=A+B ;
  - 1: SOURCE=DHI ADDR=BUS+0(CYCLE) ;
  - 2: RAM-C@ ;
  - 3: ;
  - 4: SOURCE=RD ALU=B DECODE ;

- **opcode**: LIT+_C@
  - 0: SOURCE=LIT ALU=A+B ;
  - 1: SOURCE=DHI ADDR=BUS+0(CYCLE) ;
  - 2: RAM-C@ ;
  - 3: ;
  - 4: SOURCE=RD ALU=B DECODE ;

Harris Semiconductor Proprietary
**+ _FLAGS**

**Add With Flags**

\( n1 \, n2 \rightarrow n3 \, \text{MISC} \)

**Encoding:**

\[ 0xC4 \quad \text{2 cycles} \]

**Operation:**

Perform an addition of \( n1 \) and \( n2 \) giving \( n3=n1+n2 \). Additionally, return the value of the MISC bus source, the lowest bits of which contain the ALU flags resulting from the addition (bit 0 = C, bit 1 = Z, bit 2 = S, bit 4 = V).

**Implementation:**

\[
\text{opcode}: +\_\text{FLAGS} \\
0: \text{SOURCE}=\text{DS} \quad \text{ALU}=A+B \\
1: \text{DS}=\text{FROM}=\text{DHI} \\
\text{SOURCE}=\text{MISC} \quad \text{ALU}=B \quad \text{DECODE} \\
\]

---

**-**

**Subtract**

\( n1 \, n2 \rightarrow n3 \)

**Encoding:**

\[ 0x14 \quad \text{2 cycles} \]

**Operation:**

Subtract \( n2 \) from \( n1 \), giving \( n3=n1-n2 \).

**Implementation:**

\[
\text{opcode}: - \\
0: \text{SOURCE}=\text{DS} \quad \text{ALU}=B \quad \text{DS}=\text{FROM}=\text{DHI} \\
1: \text{SOURCE}=\text{DS} \quad \text{INC}[\text{DP}] \quad \text{ALU}=A-B \quad \text{DECODE} \\
\]
**-3_ROLL**

Put Top As Fourth

\[(n_1 \ n_2 \ n_3 \ n_4 \rightarrow n_4 \ n_1 \ n_2 \ n_3)\]

Encoding:

```
0xFF 5 cycles
```

Operation:

Perform a "backwards ROLL" of the data stack involving 4 elements, moving the top stack element into the fourth-from-top position. This is an inverse operation to 3_ROLL.

Implementation:

```
opcode: -3_ROLL
0: SOURCE=DS INC[DP] DEST=DLO ;;
2: DHI[0] DS-FROM-DHI
   SOURCE=DS ALU=B ;;
3: DHI[0] DEC[DP] DS-FROM-DHI
   SOURCE=DLO ALU=B ;;
   DECODE ;;
```

**-ROT**

Put Top as Third

\[(n_1 \ n_2 \ n_3 \rightarrow n_3 \ n_1 \ n_2)\]

Encoding:

```
0x94 3 cycles
```

Operation:

Perform a "backwards ROLL" of the data stack involving 3 elements, moving the top stack element into the third-from-top position. This is an inverse operation to ROT.

Implementation:

```
opcode: -ROT
1: DHI[0] DS-FROM-DHI
   SOURCE=DS ALU=B ;;
2: DEC[DP] DHI[0] DS-FROM-DHI
   SOURCE=DHI[1] ALU=B DECODE ;;
```

Harris Semiconductor Proprietary
/STRING

Chop String
( addr1 cnt2 n3 → addr4 cnt5 )

Encoding:
0x87 4 cycles

Operation:
Zen-Forth support word for truncating the left-most n3 characters of a string represented by addr1 and cnt2. Addr4 is computed as addr1+n3. Cnt5 is computed as cnt2-n3.

Implementation:
opcode: /STRING
0: DHI[0] ALU=notA DEST=DHI[1] ;;
1: SOURCE=DS INC[DP]
   DHI[1] ALU=A+B+1 ;;
2: SOURCE=DS ALU=A+B ;;
3: DHI[0] DS-FROM-DHI
   SOURCE=DHI[1] ALU=B DECODE ;;

0<

Test For Less Than Zero
( n1 → flag2 )

Encoding:
0x08 2 cycles

Operation:
Flag2 is true if n1 is less than zero (i.e., if sign bit is set).

Implementation:
opcode: 0<
0: JMP=01S ;;
2: ALU=0 DECODE ;;
3: ALU=-1 DECODE ;;
0 = Test For Equal To Zero
(n1 → flag2)

Encoding:
0x0A  2 cycles

Operation:
Flag2 is true if n1 equals zero.

Implementation:
opcode: 0=
0: JMP=OlZ ; ;
2: ALU=0  DECODE ; ;
3: ALU=-1  DECODE ; ;

0 = _NOT Test For Not Equal To Zero
(n1 → flag2)

Encoding:
0x90  2 cycles

Operation:
Flag2 is true if n1 is not equal to zero. Useful for forcing a clean flag value of -1 or 0 if n1 is a “dirty” flag value.

Implementation:
opcode: 0 = _NOT
0: JMP=OlZ ; ;
2: ALU=-1  DECODE ; ;
3: ALU=0  DECODE ; ;
0>

Test For Greater Than Zero
(n1 → flag2)

Encoding:
0x91 3 cycles

Operation:
Flag2 is true if n1 is greater than zero. This involves both a check of the sign bit and a test for exactly equal to zero.

Implementation:
opcode: 0>
0: JMP=01z ;;
2: (<>0) JMP=10s ;;
3: (=0) JMP=101 ;;
4: (>0) ALU=-1 DECODE ;;
5: (<=0) ALU=0 DECODE ;;

1+

Increment
(n1 → n2)

Encoding:
0x6E 2 cycles

Operation:
Increments n1, producing n2 = n1 + 1

Implementation:
opcode: 1+
0: ALU=A+1 ;;
1: DECODE ;;
1-

**Decrement**

* (n1 \(\rightarrow\) n2 )

**Encoding:**

0x6F  
2 cycles

**Operation:**

Decrements n1, producing n2 = n1 - 1

**Implementation:**

opcode: 1-

0:  ALU=A-1 ;;
1:  DECODE ;;

2*

**Logical Shift Left**

* (n1 \(\rightarrow\) n2 )

**Encoding:**

0x70  
2 cycles
0x71  
1 cycle (2OPS format)

**Operation:**

Shifts n1 left by 1 bit, producing n2. This is equivalent to multiplication by 2.

**Implementation:**

opcode: 2*

0:  ALU=A+A ;;
1:  DECODE ;;

opcode: 2*_QUICK

0:  ALU=A+A  DECODE ;;
2/

Divide by 2
(n1 \rightarrow n2)

Encoding:

0xB            3 cycles

Operation:
Performs truncated signed division by two. This is true division, not simply a shift right operation.

Implementation:

opcode: 2/
0: ALU=notA DEST=DHI[1] JMP=01S ;;
2: ALU=A CIN=0 SR[ALU] JMP=110 ;;
6: DECODE ;;
( Input non-negative )

2: ALU=A CIN=0 SR[ALU] JMP=110 ;;
6: DECODE ;;
( Input negative )
3: JMP=10Z ;; ( test for -1 )

4: ALU=A+1 SR[ALU] DECODE ;;
( Input was not -1 )
5: ALU=0 DECODE ;;
( Input was -1 )

2_PICK

Push 3rd Element On Stack
(n1 n2 n3 \rightarrow n1 n2 n3 n1)

Encoding:

0x92            3 cycles

Operation:
Copies the third element on the stack to the top of stack.

Implementation:

opcode: 2_PICK
0: INC[DP] ;;
3ROLL
Get Fourth Element On Stack
( n1 n2 n3 n4 → n2 n3 n4 n1 )

Encoding:
0x93 5 cycles

Operation:
Moves the fourth element on the stack to the top of stack.

Implementation:
opcode: 3_ROLL
0: SOURCE=DS INC[DP] DEST=DLO ;
2: DHI[1] DS-FROM-DHI
   SOURCE=DS ALU=B ;;
3: SOURCE=DLO DEC[DP] DEST=DS ;;
4: DEC[DP] DHI[0] DS-FROM-DHI
   SOURCE=DHI[1] ALU=B DECODE ;;

4+
Increment By 4
( n1 → n2 )

Encoding:
0x72 2 cycles

Operation:
Adds 4 to n1, giving n2 = n1 + 4.

Implementation:
opcode: 4+
0: SOURCE=4 ALU=A+B ;;
1: DECODE ;;
4-_1

Store At Address-4
(n1 addr2 →)

Encoding:
0x95 4 cycles

Operation:
Stores n1 at address “addr2-4”.

Implementation:
opcode: 4-_1
0: SOURCE=DHI ADDR=BUS-4 (CYCLE) ;
1: SOURCE=DS INC[DP] DEST=RAM-1 ;
2: ;
3: SOURCE=DS INC[DP] ALU=B DECODE ;

4-_@

Load From Address-4
(addr1 → n2)

Encoding:
0x96 4 cycles

Operation:
Fetches value n1 from address “addr1-4”.

Implementation:
opcode: 4-_@
0: SOURCE=DHI ADDR=BUS-4 (CYCLE) ;
1: ;
2: ;
3: SOURCE=RD ALU=B DECODE ;
< + loop >

Loop, Count By N
(n1 → )
RS( limit1 index2 → limit1 index3 ...

Encoding:
0x1F0 / call 7/10 cycles

Operation:
Adds n1 to the loop index index2, then checks to see if the new index value index3 exceeds the loop limit limit1. If n1 is non-negative, then “exceeds” is defined as index3 > limit1. If n1 is negative, then “exceeds” is defined as index3 < limit1. If index3 exceeds limit1, the index and limit are popped off the return stack and execution continues in-line. If index3 does not exceed limit1, the new index value is stored on the return stack, and a branch is taken to the target address. This opcode is always compiled with a CALL instruction type, where the CALL address field supplies the branch address. The opcode takes 7 clock cycles if the branch is taken, 10 clock cycles if not taken. This opcode assumes that index3 and limit1 will never differ in value more than 0x80000000. In other words, it does not check for overflow when doing the comparison.

Notes:
This opcode is broken in the BINAR ROM. The Implementation given below is correct, and is loaded as an MRAM opcode during Forth cold-start.

Implementation:
opcode: <+loop>
0: SOURCE=RETURN-SAVE DEST=DLO ;;
1: DEC[DP] SOURCE=DHI DEST=DS
   INC[RP] ;;
2: SOURCE=RS INC[RP] ALU=A+B ;;
   ( If result sign same as increment, done )
5: SOURCE=DHI DEC[RP] DEST=RS
   JMP=11S ;;
   ( Signs different, continue loop )
7: SOURCE=DS INC[DP] ALU=B DECODE ;;
   ( Signs match, abort loop )
6: SOURCE=DLO ADDR=BUS+0(CYCLE)
   INC[MPC] JMP=101 ;;
NEXT opcode
5: INC[RP] ;;
6: INC[RP] LATCH-INSTRUCTION ;;
7: SOURCE=DS INC[DP] ALU=B DECODE ;;
< >

Test For Not Equal
( n1 n2 → flag3 )

Encoding:
0xB 3 cycles

Operation:
Flag3 is true if n1 is not equal to n2.

Implementation:
opcode: <>
0: SOURCE=DS INC[DP] ALU=A-B ;;
1: JMP=01Z ;;
2: ALU=-1 DECODE ;; ( Result not0, <> )
3: ALU=0 DECODE ;; ( Result 0, = )

< cold >

Cold Start
( → )

Encoding:
0x04 8 cycles

Operation:
Opcode executed when the RESET pin is released. Use at any other time is not recommended.

Implementation:
opcode: <cold>
0: ;;
1: ;;
2: ;;
3: ;;
4: ;;
5: ;;
6: ;;
7: ALU=0 DECODE ;;
**<do>**

**Start Loop**

( limit1 start2 → )
RS( → limit1 index2 )

**Encoding:**
0x74 3 cycles

**Operation:**
The limit1 and start2 value for a loop are moved from the data stack to the return stack (where the start value becomes the initial index value). This opcode is compatible with all looping instructions except NEXT.

**Notes:**
Do not combine this opcode with a CALL or EXIT instruction.

**Implementation:**
opcode: <do>
0: SOURCE=DS INC[DP] DEC[RP] DEST=RS ;
1: SOURCE=DHI DEC[RP] DEST=RS ;
2: SOURCE=DS INC[DP] ALU=B DECODE ;

**<interrupt>**

**Interrupt**

( → n1 )
RS( → addr2 )

**Encoding:**
0x01 2 cycles

**Operation:**
This is the opcode executed when an interrupt is recognized. The n1 value returned is the configuration register value before the interrupt was processed. This value contains the interrupt mask bit in bit 0 (before the interrupt mask bit was set by processing the interrupt), as well as the four bits indicating data or return stack interrupts. The interrupt restart address plus 4 is pushed as value addr2 onto the return stack (in order to restart after an interrupt, the value addr2-4 is used as the restart address).

**Notes:**
This opcode is automatically invoked by the hardware when processing an interrupt. Its definition is subject to change in future chip revisions, so using it for other purposes would be dim-witted.

**Implementation:**
opcode: <interrupt>
0: DEC[RP]
( Required to catch return address! )
<lit>

Push Long Immediate

(→ n₁)

Encoding:
0x15 / call 4 cycles

Operation:
Returns an in-line 32-bit literal value n₁. The <LIT> opcode is compiled with a CALL instruction whose address field points to the next instruction to be executed, typically (but not necessarily) at the address of the <LIT> instruction plus 8. The memory word after the <LIT> instruction (at <LIT> instruction plus 4 in memory) contains the 32-bit value returned as n.

Implementation:
opcode: <lit>
0: SOURCE=RETURN-SAVE
   ADDR=BUS+0(CYCLE) ;;
2: ;;
3: SOURCE=RD ALU=B DECODE ;;
**<loop>**

**Loop**

\( RS( \text{limit1 index2} \rightarrow \text{limit1 index3} \rightarrow ) \)

**Encoding:**

| 0x1A / call | 6/9 clocks |

**Operation:**

Adds 1 to the loop index index2, then checks to see if the new index value index3 exceeds (is greater than or equal to) the loop limit limit1. If index3 exceeds limit1, the index and limit are popped off the return stack and execution continues in-line. If index3 does not exceed limit1, the new index value is stored on the return stack, and a branch is taken to the target address. This opcode is always compiled with a CALL instruction type, where the CALL address field supplies the branch address. The opcode takes 6 clock cycles if the branch is taken, 9 clock cycles if not taken. This opcode assumes that index3 and limit1 will never differ in value more than 0x80000000. In other words, it does not check for overflow when doing the comparison.

**Implementation:**

**opcode**: <loop>

| 0: | DEC[DP] DS-FROM-DHI | DEST=DLO ; ; |
| 1: | INC[RP] ALU=0 ; ; |
| 2: | SOURCE=RS INC[RP] ALU=A+B+1 ; ; |
| 3: | SOURCE=RS ALU=A-B DEST=DHI[1] ; ; |
| 4: | SOURCE=DHI DEC[RF] DEST=RS |
| | JMP=11S ; ; |
| ( Negative result, continue loop ) |
| 5: | INC[RP] ; ; |
| 6: | INC[RP] LATCH-INSTRUCTION ; ; |
| 7: | SOURCE=DS INC[DP] ALU=B DECODE ; ; |
| ( Non-Negative comparison, abort loop ) |
| 6: | SOURCE=DLO ADDR=BUS+0(CYCLE) |
| | INC[MPC] JMP=101 ; ; |
| NEXT opcode |

Harris Semiconductor Proprietary
**< next >**

**Count Down Loop**

\[
\begin{align*}
( \rightarrow ) \\
\text{RS( count1 } \rightarrow \text{ count2} \\
\quad \ldots \rightarrow )
\end{align*}
\]

**Encoding:**

0xE1 4 cycles

**Operation:**

Decrement loop counter count1 (at top of return stack) by 1, then checks for new count2 value equal to zero. If count2 is zero, then it is popped off the return stack and execution continues in-line. If the count2 is non-zero, it is placed on the return stack, and a branch is taken to the target address. This opcode is always compiled with a CALL instruction type, where the CALL address field supplies the branch address. Note that an initial count value of -1 performs 4 giga-iterations.

**Implementation:**

opcode: `<next>`

( Remember that 1st cycle will get pre-call RS value )

0: \( \text{SOURCE=}\text{RS DHI}[1] \text{ ALU=}B ;; \)

1: \( \text{SOURCE=}\text{RS INC[RP]} \)
   \[
   \begin{align*}
   \text{ADDR=}&\text{BUS+0(CYCLE)} \\
   \text{DHI}[1] &\text{ ALU=}A-1 \quad \text{JMP=}01Z ;;
   \end{align*}
\]

( Zero flag, fall through )

3: \( \text{INC[RP]} ;; \)

4: \( \text{LATCH- INSTRUCTION} ;; \)

5: \( \text{DECODE} ;; \)

( Non-zero flag, take the branch )

2: \( \text{SOURCE=}DHI[1] \text{ DEST=}\text{RS} \text{ JMP=}110 ;; \)

6: \( ;; \)

7: \( \text{DECODE} ;; \)
### <short>

**Push Short Immediate**

(→ n1 ) Immediate

**Encoding:**

- 0x16 / lit  
  2 cycles
- 0x18 / lit  
  1 cycle (2OPS format)

**Operation:**

Pushes the compiled instruction lit field onto the stack as n1.

**Notes:**

Use the `<LIT>` opcode if a full 32-bit literal value is required.

**Implementation:**

```
opcode: <short>
0: SOURCE=LIT ALU=B
   DEC[DP] DS-FROM-DHI ;;
1: DECODE ;;

opcode: <short> QUICK
0: SOURCE=LIT ALU=B
   DEC[DP] DS-FROM-DHI DECODE ;;
```

### <trap>

**Illegal Opcode Trap**

(→ addr1 n2 )

**Encoding:**

- 0x03  
  5 cycles

**Operation:**

Perform a subroutine call to address $FFFFFFFC (regardless of instruction type). The address just beyond the address of the instruction containing this opcode (i.e., the instruction's address plus 4) is returned as addr1, and the value of the MISC register (which, among other things, contains the micro-address field indicating the value of the opcode causing the fault) is returned as n2. This opcode is invoked by hardware whenever an illegal opcode value is interpreted at execution time. It may also be used as a software interrupt if desired with appropriate trap handling software.

**Implementation:**

```
opcode: <trap>
0: DEC[DP) DS-FROM-DHI
   SOURCE=RETURN-SAVE ALU=B
1: SOURCE=O ADDR=sus-4(CYCLE)
2: DEC[DP] DS-FROM-DHI
   SOURCE=MISC ALU=B ;;
3: LATCH-INSTRUCTION ;;
4: DECODE ;;
```
=

**Test For Equal**

\[(n1 \, n2 \rightarrow \text{flag3})\]

**Encoding:**

\[0x7A \quad 3 \text{ cycles}\]

**Operation:**

Flag3 is true if \(n1\) and \(n2\) are equal.

**Implementation:**

opcode: \(=\)

0: \(\text{SOURCE}=\text{DS} \, \text{INC}[\text{DP}] \, \text{ALU}=\text{A-B} \, ;;\)
1: \(\text{JMP}=01Z \, ;;\)
2: \(\text{ALU}=0 \, \text{DECODE} \, ;;\) (Result not0, <>)
3: \(\text{ALU}=-1 \, \text{DECODE} \, ;;\) (Result 0, =)

= branch

**Jump If Equal**

\[(n1 \, n2 \rightarrow )\]

**Encoding:**

\[0x99 / \text{call} \quad 5 \text{ cycles}\]

**Operation:**

Branch if \(n1\) and \(n2\) are equal. The \(=\)branch opcode is compiled with a CALL instruction having an address field pointing to the branch target.

**Implementation:**

opcode: \(=\)branch

0: \(\text{SOURCE}=\text{DS} \, \text{INC}[\text{DP}] \, \text{ALU}=\text{A-B} \, ;;\)
1: \(\text{SOURCE}=\text{RS} \, \text{ADDR}=\text{BUS}+0(\text{CYCLE}) \, ;;\)
2: \(\text{INC}[\text{RP}] \quad \text{JMP}=10Z \, ;;\)

(Non-Zero flag, fall through)

4: \(\text{LATCH-INSTRUCTION} \quad \text{JMP}=111 \, ;;\)
7: \(\text{SOURCE}=\text{DS} \, \text{INC}[\text{DP}] \, \text{ALU}=\text{B} \, \text{DECODE} \, ;;\)

(Zero flag, take the branch)

5: ;;
6: \(\text{SOURCE}=\text{DS} \, \text{INC}[\text{DP}] \, \text{ALU}=\text{B} \, \text{DECODE} \, ;;\)
Test For Greater Than
(n1 n2 → flag3)

Encoding:
0x1E 4 cycles

Operation:
Flag3 is true if n1 is greater than n2.

Implementation:
opcode: >
0: SOURCE=DS INC[DP] ALU=A-B ;;
1: JMP=01V ;;
2: JMP=10S ;;
3: JMP=11S ;;
4: ALU=0 DECODE ;; (not S not V)
5: ALU=-1 DECODE ;; (S not V)
6: ALU=-1 DECODE ;; (not S V)
7: ALU=0 DECODE ;; (S V)

> R
Transfer Dstack to Rstack
(n1 → )
RS( → n1 )

Encoding:
0x1F 2 cycles

Operation:
Pop n1 from the data stack, placing it on the return stack. This opcode also accomplishes the <for> function for use with the <next> looping opcode in the "FOR ... NEXT" construct.

Notes:
Do not combine this opcode with a CALL or EXIT instruction.

Do not execute this opcode as the opcode immediately preceding a subroutine return instruction (will be corrected on later versions of the chip). Workaround: form a 2OPS instruction with >R as the first opcode and NOP_QUICK as the second opcode, then place the subroutine return as the following instruction.

Implementation:
opcode: >R
0: DS-FROM-DHI SOURCE=DS ALU=B ;;
1: SOURCE=DS INC[DP]
DEC[RP] DEST=RS DECODE ;;
???branch

Jump If Zero
( flag1 → )

Encoding:
0x20  4 cycles

Operation:
Perform a branch if flag1 is zero. ?BRANCH must be compiled as the opcode in a CALL instruction that has the branch target as its next address field.

Implementation:
opcode: ?branch
0: SOURCE=RETURN-SAVE
   ADDR=BUS+0(CYCLE) ;;
1: INC[RP]   JMP=OlZ ;;

( Non-zero flag, fall through )
2: LATCH-INSTRUCTION   JMP=111 ;;
7: SOURCE=DS INC[DP]   ALU=B   DECODE ;;
( Zero flag, take the branch )
3: ;;
4: SOURCE=DS INC[DP]   ALU=B   DECODE ;;

?DNEGATE

Double Precision Conditional Negate
( d1 n2 → d3 )

Encoding:
0xA3  5 cycles

Operation:
If n2 is non-negative, return d3 = d1. If n2 is negative, return d3 as the two's complement of d1.

Implementation:
opcode: ?DNEGATE
0: SOURCE=DS INC[DP]
   DHI[1] ALU=notB   JMP=OlS ;;

( Negative )
3: SOURCE=DS   ALU=notB ;;
4: ALU=A+1 ;;
5: DHI[0] DS-FROM-DHI
   SOURCE=DHI[1]   ALU=B   JMP=11C ;;
6: ALU=A   DECODE ;;
7: ALU=A+1   DECODE ;;

( Non-Negative )
2: SOURCE=DHI[1] ALU=notB
   INC[MPC]   JMP=101 ;;
next opcode
( Nop padding for consistent timing )
5: ;;
6: ;;
7: DECODE ;;
**?DUP**

**Conditional Push**

(n1 → n1 n1
... → n1)

**Encoding:**

0x7C 2 cycles

**Operation:**

Push a second copy of n1 onto the stack if it is non-zero, else leave it alone.

**Implementation:**

opcode: ?DUP
0: JMP=01Z ;;
( Push )
2: DEC[DP] DS-FROM-DHI DECODE ;;
( Don't Push )
3: DECODE ;;

**?EXIT**

**Conditional Exit**

(flag1 → )

**Encoding:**

0xE0 4 cycles

**Operation:**

Perform a subroutine exit if flag1 is non-zero.

**Notes:**

This opcode should only be used as a JNEXT opcode or as the second opcode of a 2OPS instruction.

Do not use this as the very first opcode of a subroutine, or improper operation will result (a 5-clock version could be written that does not have this limitation).

**Implementation:**

opcode: ?EXIT
0: SOURCE=RS ADDR=BUS+0(CYCLE) ;;
1: JMP=01Z ;;
( don't exit )
2: JMP=100 ;;
( exit )
3: LATCH-INSTRUCTION INC[RP] JMP=100 ;;
4: SOURCE=DS INC[DP] ALU=B DECODE ;;
**?NEGATE**

**Conditional Negation**

\[( n1 \ n2 \rightarrow n3 )\]

**Encoding:**

0x97 2 cycles

**Operation:**

If \( n2 \) is less than zero, perform a two's complement negation on integer \( n1 \), returning integer \( n3 \). Otherwise, return \( n1 \) as \( n3 \).

**Implementation:**

opcode: +-
0: SOURCE=DS INC[DP] DHI[1]
    ALU=notB JMP=01S ;;

(non-negative)
2: DHI[1] ALU=notA DEST=DHI[0] DECODE ;;
(negative)

---

**@**

**Load**

\[( addr1 \rightarrow n2 )\]

\(( \rightarrow n2 )\) Immediate

**Encoding:**

0x21 4 cycles
0xB2 /lit 4 cycles

**Operation:**

Fetches the 32-bit value at \( addr1 \), returning \( n2 \). In the immediate variant, the address \( addr1 \) is supplied by the literal field. Note that in this case the address is a byte-addressed literal, not the address field found in a CALL instruction.

**Implementation:**

opcode: @
0: SOURCE=DHI ADDR=BUS+O(CYCLE) ;;
1: ;;
2: ;;
3: SOURCE=RD ALU=B DECODE ;;

opcode: LIT @
0: SOURCE=LIT ADDR=BUS+O(CYCLE) ;;
1: SOURCE=DHI DEC[DP] DEST=DS ;;
2: ;;
3: SOURCE=RD ALU=B DECODE ;;
@_!

Store Indirect
(n1 addr2 →)

Encoding:
0x9B 7 cycles

Operation:
Performs an indirect store of n1 through addr2. In other words, the value n1 is stored at the address contained in the memory word addressed by addr2.

Implementation:
opcode: @_!
0: SOURCE=DHI ADDR=BUS+0(CYCLE) ;;
1: ;;
2: ;;
3: SOURCE=RD ADDR=BUS+0(CYCLE) ;;
4: SOURCE=DS INC[DP] DEST=RAM-1 ;;
5: ;;
6: SOURCE=DS INC[DP] ALU=B DECODE ;;

@_+

Fetch Then Add
(n1 addr2 → n3)

Encoding:
0x9C 4 cycles

Operation:
Add the value contained in memory location addr2 to n1, giving result n3.

Implementation:
opcode: @_+
0: SOURCE=DHI ADDR=BUS+0(CYCLE) ;;
1: ;;
2: SOURCE=DS INC[DP] ALU=B ;;
3: SOURCE=RD ALU=A+B DECODE ;;
@_@

Load Indirect
( addr1 → n2 )

Encoding:
0x9A 7 cycles

Operation:
Performs an indirect fetch through addr1. In other words, the value n2 is fetched from the address contained in the memory word addressed by addr1.

Implementation:
opcode: @_@
0: SOURCE=DHI ADDR=BUS+0(CYCLE) ;;
1: ;;
2: ;;
3: SOURCE=RD ADDR=BUS+0(CYCLE) ;;
4: ;;
5: ;;
6: SOURCE=RD ALU=B DECODE ;;

@_INC

Incrementing Fetch
( addr1 → addr2 n3 ) Immediate

Encoding:
0xEE/lit 4 cycles

Operation:
Perform a fetch operation from addr1, returning the value n3. Also, add the compiled literal value to addr1 after the fetch is performed to return addr2.

Implementation:
opcode: @_INC
0: SOURCE=DHI ADDR=BUS+0(CYCLE) ;;
1: SOURCE=LIT ALU=A+B ;;
2: SOURCE=DHI DEC[DP] DEST=DS ;;
3: SOURCE=RD ALU=B DECODE ;;
@_LIT+

Fetch Then Add Immediate

( addr2 → n2 ) Immediate

Encoding:

0x9D / lit 4 cycles

Operation:

Add the value contained in memory at the address given by the compiled instruction literal field to n1, giving result n2.

Implementation:

opcode: @_LIT+
0: SOURCE=DHI ADDR=BUS+0(CYCLE) ;;
1: ;
2: SOURCE=LIT ALU=B ;;
3: SOURCE=RD ALU=A+B DECODE ;;

ABS

Absolute Value

( n1 → n2 )

Encoding:

0x7D 2 cycles

Operation:

Take the absolute value of n1, returning n2. n2 equals n1 if n1 is greater than or equal to 0. n2 is the two's complement of n1 if n1 is less than zero.

Implementation:

opcode: ABS
0: ALU=notA JMP=01S ;;
( Non-negative, restore to original )
2: ALU=notA DECODE ;;
( Two's complement )
3: ALU=A+l DECODE ;;
ADC

Add With Carry
(n1 n2 cflagin → n3 cflagout)

Encoding:
0x9E  4 cycles

Operation:
Perform an add with carry. cflagin is a truth flag that represents the carry into the addition, and cflagout is a truth flag that represents the carry out of the addition, with a true value indicating carry set. n3 is the sum of n1 and n2 plus a 0 or 1 (depending on cflagin). cflagout is the carry out of the addition. This is a useful primitive for synthesizing extended precision arithmetic.

Implementation:
opcode: ADC
0: SOURCE=DS INC[DP] ALU=B JMP=01Z ;;
   ( cin <>0 )
2: SOURCE=DS ALU=A+B+1 JMP=100 ;;
   ( cin = 0 )
3: SOURCE=DS ALU=A+B JMP=100 ;;
4: SOURCE=DHI DEST=DS JMP=11C ;;
   ( cout = 0 )
6: ALU=0 DECODE ;;
   ( cout <>0 )
7: ALU=-1 DECODE ;;

AND

Bitwise AND
(n1 n2 → n3)
(n1 → n3) Immediate

Encoding:
0x22  2 cycles
0x23  1 cycles (2OPS format)
0xAD/lit  2 cycles
0xAE/lit  1 cycle (2OPS format)

Operation:
Perform a bitwise logical AND of n1 and n2, returning n3. In the immediate variant, the value n2 is supplied by the literal field. Note that since the literal field is signed, the highest bits of the literal value may be either 0 (to mask n1 bits), or 1 (to allow n1 bits to propagate to n2).

Implementation:
opcode: AND
0: SOURCE=DS INC[DP] ALU=AandB ;;
1: DECODE ;;
opcode: AND_QUICK
0: SOURCE=DS INC[DP] ALU=AandB
   DECODE ;;
opcode: LIT_AND
0: SOURCE=LIT ALU=AandB ;;
1: DECODE ;;
opcode: LIT_AND_QUICK
0: SOURCE=LIT ALU=AandB DECODE ;;

AND!

And To Memory

(nl addr2 →)

Encoding:
0x9F 7 cycles

Operation:
Perform an atomic bitwise logical AND of nl with the word at memory addr2, returning the result to addr2 (similar to +!, but with an AND operation instead of a + operation).

Implementation:
opcode: AND!
0: SOURCE=DHI ADDR=BUS+0(CYCLE) ;;
1: RAM-RMW ;;
2: SOURCE=DS INC[DP] ALU=B ;;
3: SOURCE=RD ALU=AandB CYCLE-RAM ;;
4: SOURCE=DHI DEST=RAM-1 ;;
5: ;;
6: SOURCE=DS INC[DP] ALU=B DECODE ;;
**ASIC!**

**I/O Output**
( n1 portaddr2 → )
( n1 → ) Immediate

**Encoding:**
- 0x58  4 cycles
- 0x59 / lit  4 cycles

**Operation:**
Write value n1 to output device using address portaddr2. In the immediate variant, the port address portaddr2 is provided by the literal field.

**Implementation:**
- **opcode:** ASIC!
  0: SOURCE=DHI ADDR=BUS+0(CYCLE) ;;
  1: SOURCE=DS INC[DP] DEST=ASIC-1 ;;
  2: ;;
  3: SOURCE=DS INC[DP] ALU=B DECODE ;;

- **opcode:** LIT ASIC!
  0: SOURCE=LIT ADDR=BUS+0(CYCLE) ;;
  1: SOURCE=DHI DEST=ASIC-1 ;;
  2: ;;
  3: SOURCE=DS INC[DP] ALU=B DECODE ;;

**ASIC@**

**I/O Input**
( portaddr1 → n2 )
( → n2 ) Immediate

**Encoding:**
- 0x5A  4 cycles
- 0x5B / lit  4 cycles

**Operation:**
Read value n2 from input device using address portaddr1. In the immediate variant, portaddr1 is supplied by the literal field.

**Implementation:**
- **opcode:** ASIC@
  0: SOURCE=DHI ADDR=BUS+0(CYCLE) ;;
  1: ASIC-@ ;;
  2: ;;
  3: SOURCE=RD ALU=B DECODE ;;

- **opcode:** LIT ASIC@
  0: SOURCE=LIT ADDR=BUS+0(CYCLE) ;;
  1: ASIC-@ ;;
  2: SOURCE=DHI DEC[DP] DEST=DS ;;
  3: SOURCE=RD ALU=B DECODE ;;

Harris Semiconductor Proprietary
**ASR**

Arithmetic Shift Right

( n1 → n2 )

**Encoding:**

0x24 2 cycles

**Operation:**

Perform an arithmetic shift right of n1 by one bit, giving n2.

**Notes:**

This is equivalent to division by two for non-negative integers, and equivalent to floored division by two for all integers.

**Implementation:**

opcode: ASR

0: JMP=01S ;;

( non-negative: shift in a 0 )

2: ALU=A CIN=0 SR[ALU] DECODE ;;
( negative: shift in a 1 )

3: ALU=A CIN=1 SR[ALU] DECODE ;;

---

**B@**

Load Signed Byte

( addr1 → b2 )

**Encoding:**

0x28 4 cycles

**Operation:**

Fetch byte value b1 from address addr2. The value b2 is sign extended to form a 32-bit signed integer.

**Implementation:**

opcode: B@

0: SOURCE=DHI ADDR=BUS+0(CYCLE) ;;
1: RAM-C@ ;;
2: ;;
3: SOURCE=RD-SIGNED ALU=B DECODE ;;
**BIT-CLEAR**

**Bit Clear**

( n1 n2 → n3 )

**Encoding:**

0xC3 2 cycles

**Operation:**

Every bit set in n2 is cleared in value n1, giving result n3. This is a bit clear operation that uses n2 as an enabling mask for clearing bits in n1.

**Implementation:**

opcode: BIT-CLEAR
0: ALU=notA ;;
1: SOURCE=DS INC[DP]
   ALU=AandB DECODE ;;

---

**branch**

**Unconditional Branch**

( → )

**Encoding:**

0x25 /call 2 cycles

**Operation:**

Perform an unconditional branch. The BRANCH opcode is compiled with a CALL instruction having an address field pointing to the branch target.

**Notes:**

Do not execute this opcode as the opcode immediately preceding a subroutine return instruction (will be corrected on later versions of the chip).

**Implementation:**

opcode: branch
0: ;;
1: INC[RP] DECODE ;;
C!

**Store Character**
(c1 addr2 → )

**Encoding:**
0x26 4 cycles

**Operation:**
Store character value c1 at address addr2. The lowest 8 bits of value c1 are stored.

**Implementation:**
opcode: C1
0: SOURCE=DHI ADDR=BUS+0(CYCLE) ;
1: SOURCE=DS INC[DP] DEST=RAM-C1 ;
2: ;
3: SOURCE=DS INC[DP] ALU=B DECODE ;

C!_INC

**Incrementing Store Character**
(c1 addr2 → addr3) Immediate

**Encoding:**
0xF1 /lit 4 cycles

**Operation:**
Store value c1 at location addr2, then add the compiled instruction literal field to addr2, leaving addr3.

**Implementation:**
opcode: C1_INC
0: SOURCE=DHI ADDR=BUS+0(CYCLE) ;
1: SOURCE=DS INC[DP] DEST=RAM-C1 ;
2: ;
3: SOURCE=LIT ALU=A+B DECODE ;
**C+!**

**Indirect Increment Byte**

(b1 addr2 → )

**Encoding:**

0xA1 7 cycles

**Operation:**

Perform an 8-bit addition of b1 to the byte at memory addr2, returning the result to addr2 (similar to +!, but with an 8-bit operation).

**Implementation:**

```
opcode: C+!
0: SOURCE=DHI ADDR=BUS+0(CYCLE) ;;
1: RAM-C@ ;;
2: SOURCE=DS INC[DP] ALU=B ;;
3: SOURCE=RD ALU=A+B CYCLE-RAM ;;
4: SOURCE=DHI DEST=RAM-C@ ;;
5: SOURCE=DS INC[DP] ALU=B ;;
6: DECODE ;;
```

**C@**

**Load Character**

(addr1 → c2)

**Encoding:**

0x27 4 cycles

**Operation:**

Fetch character value c2 from address addr1. The value c2 is placed in the lowest 8 bits of the stack word and padded with leading zeros.

**Implementation:**

```
opcode: C@
0: SOURCE=DHI ADDR=BUS+0(CYCLE) ;;
1: RAM-C@ ;;
2: ;;
3: SOURCE=RD ALU=B DECODE ;;
```
C@_INC

Incrementing Fetch Character
( addr1 → addr2 c3 ) Immediate

Encoding: 0xEF / lit 4 cycles

Operation: Perform a character fetch operation from addr1, returning the value n3. Also, add the compiled literal value to addr1 after the fetch is performed to return addr2.

Implementation:
opcode: C@_INC
0: SOURCE=DHI ADDR=BUS+0(CYCLE) ;;
1: SOURCE=LIT ALU=A+B RAM-C@ ;;
2: SOURCE=DHI DEC[DP] DEST=DS ;;
3: SOURCE=RD ALU=B DECODE ;;

CMOVE

Move Bytes
( src1 dest2 cnt3 → )

Encoding: 0x8A 4 + 4*cnt3 cycles

Operation: Move a block of cnt3 bytes of data starting at address src1 to a block of memory starting at address dest2.

Notes:
A cnt3 value of 0 will attempt to move 4G bytes.
CMOVE destroys the value of registers SBASE and DBASE.
This is a non-interruptible instruction. The commercial chip will have an interruptible version of this opcode.

Implementation:
opcode: CMOVE
0: SOURCE=DS INC[DP] DEST=DBASE ;;
1: SOURCE=DS INC[DP] DEST=SBASE
   DHI[1] ALU=0 ( offset ) ;;
   ( Byte move loop )
2: SOURCE=DHI[1] ADDR=BUS+SBASE(CYCLE)
   JMP=101 ;;
5: RAM-C@ ;;
( Bump & Test count)
6: SOURCE=DHI[1] ADDR=BUS+DBASE(CYCLE)
   DHI[0] ALU=A-1 ;;
7: SOURCE=RD DEST=RAM-C!
   DHI[1] ALU=A+1 JMP=01Z ;;
( End loop )
3: ;; (wait for RAM cycle to complete)
4: SOURCE=DS INC[DP] ALU=B DECODE ;;

CONFIG!

Set Configuration Register
(n1 →)

Encoding:
0x6A    2 cycles

Operation:
Store n1 into the on-chip configuration register, setting stack over/underflow interrupt bits and the interrupt mask.

*** picture of config reg could go here ***

Notes:
This implementation should have a nop microinstruction in it to allow the config register to properly mask/unmask interrupts before executing the DECODE.

Implementation:
opcode: CONFIG!
0: SOURCE=DHI DEST=CONFIG ;;
1: SOURCE=DS INC[DP] ALU=B DECODE ;;
CONFIG@

Get Configuration Register
( → n1 )

Encoding:
0x69 2 cycles

Operation:
Fetch n from the on-chip configuration register, reading stack over/underflow interrupt bits and the interrupt mask.

*** picture of config reg could go here ***

Implementation:
opcode: CONFIG@
0: DEC[DP] DS-FROM-DHI
   SOURCE=CONFIG ALU=B ;
1: DECODE ;

COUNT

Load Count
( addr1 → addr2 un3 )

Encoding:
0x8E 4 cycles

Operation:
Fetch the unsigned byte count value at addr1, returning the start of a string address addr2 (equal to addr1 plus 1) and the count (the unsigned byte at addr1). This primitive is used in conjunction with Forth counted strings.

Implementation:
opcode: COUNT
0: SOURCE=DHI
   ADDR=BUS+0(CYCLE) ALU=A+1 ;
1: RAM-C@ ;
2: ;
3: DEC[DP] DS-FROM-DHI
   SOURCE=RD ALU=B DECODE ;
C_OR!

Or Character To Memory 
( c1 addr2 → )

Encoding:  
0xA2 7 cycles

Operation:  
Perform an 8-bit bitwise logical OR of c1 to 
the byte at memory addr2, returning the 
result to addr2 (similar to C+!, but with an 
OR operation instead of an addition).

Implementation:  
opcode: C_ORI  
0: SOURCE=DHI ADDR=BUS+0(CYCLE) ; ;  
1: RAM=C0 ; ;  
2: SOURCE=DS INC[DP] ALU=B ; ;  
3: SOURCE=RD ALU=AorB CYCLE=RAM ; ;  
4: SOURCE=DHI DEST=RAM-C1 ; ;  
5: SOURCE=DS INC[DP] ALU=B ; ;  
6: DECODE ; ;

D!

Double Precision Store 
( d1 addr2 → )

Encoding:  
0x29 7 cycles

Operation:  
Store 64-bit value d1 at address addr2, placing 
the lower order word at address addr2+0, and 
the higher order word at address addr2+4.

Implementation:  
opcode: D!  
0: SOURCE=DHI ADDR=BUS+0(CYCLE) 
INC[DP] ; ;  
1: SOURCE=DS DEST=RAM-1 ; ;  
2: SOURCE=4 ALU=A+B ; ;  
3: SOURCE=DS ADDR=BUS+0(CYCLE) 
DEC[DP] ; ;  
4: SOURCE=DS DEST=RAM-1 INC[DP] ; ;  
5: INC[DP] ; ;  
6: SOURCE=DS ALU=B INC[DP] 
DECODE ; ;
**D+**

Double Precision Add

\((d_1 \ d_2 \rightarrow d_3)\)

**Encoding:**

0x2A  \  \ 5 cycles

**Operation:**

Add 64-bit values \(d_1\) and \(d_2\), resulting in \(d_3\).

**Implementation:**

```
opcode: D+
1: SOURCE=DS INC[DP] DEST=DLO ;
3: SOURCE=DHI[1] DEST=DS JMP=10C ;

( Carry false )
4: SOURCE=DLO ALU=A+B DECODE ;
( Carry true )
5: SOURCE=DLO ALU=A+B+1 DECODE ;
```

---

**D−**

Double Precision Subtract

\((d_1 \ d_2 \rightarrow d_3)\)

**Encoding:**

0x86  \  \ 5 cycles

**Operation:**

Subtract 64-bit value \(d_2\) from \(d_1\), giving \(d_3\).

**Implementation:**

```
opcode: D−
1: SOURCE=DS INC[DP] ALU=notA DEST=DLO ;
3: SOURCE=DHI[1] DEST=DS JMP=10C ;

( Carry false )
4: SOURCE=DLO ALU=A+B DECODE ;
( Carry true )
5: SOURCE=DLO ALU=A+B+1 DECODE ;
```
**D =**

Test for Double Precision Equal

( d1 d2 → flag )

**Encoding:**

0xA5 5 cycles

**Operation:**

Compare d1 to d2, returning a true flag if they are equal.

**Implementation:**

opcode: D=

0: SOURCE=DS INC(DP) DHI[1] ALU=B ; ;
1: SOURCE=DS INC(DP) DHI[0] ALU=A-B ; ;
2: SOURCE=DS INC(DP)
   DHI[1] ALU=A-B JMP=10Z ; ;
4: ALU=0 JMP=110 ; ; (low words not equal)
5: JMP=11Z ; ; (low words equal )
6: ALU=0 DECODE ; ; (not equal )
7: ALU=-1 DECODE ; ; (both words equal )

---

**D > R**

Double Precision Transfer Dstack To Rstack

( n1 n2 → )

RS( → n2 n1 )

**Encoding:**

0x2B 3 cycles

**Operation:**

Transfer the pair n1 and n2 from the data stack to the return stack. Note that the order of n1 and n2 on the stack are reversed. D>R is identical in operation to execution of the pair “>R >R”. However, as long as DR> is used to transfer the results back, D>R may be thought of as a word that transfers a double precision number to the return stack as well.

**Notes:**

Do not combine this opcode with a CALL or EXIT instruction.

Do not execute this opcode as the opcode immediately preceding a subroutine return instruction (will be corrected on later versions of the chip). Workaround: form a 2OPS instruction with D>R as the first opcode and NOP_FAST as the second opcode, then place the subroutine return as the following instruction.
Implementation:

opcode: D>R
0: SOURCE=DHI  DEC[R] DEST=RS ;
1: SOURCE=DS INC[DP] DEC[R] DEST=RS ;
2: SOURCE=DS INC[DP] ALU=B DECODE ;

D@

Double Precision Load
( addr1 → d2 )

Encoding:

0x2C  6 cycles

Operation:

Fetch 64-bit value d1 from address addr1, reading the low order word from address addr1+0, and the high order word from address addr1+4.

Implementation:

opcode: D@
0: SOURCE=DHI  ADDR=BUS+0(CYCLE) ;
1: SOURCE=4  ALU=A+B ;
2: SOURCE=DHI  ADDR=BUS+0(CYCLE) ;
3: SOURCE=RD  DEC[DP] DEST=DS ;
4: ;
5: SOURCE=RD  ALU=B DECODE ;
DASR

Double Precision Arithmetic Shift Right
( d1 → d2 )

Encoding:
0x2D 3 cycles

Operation:
Perform a 64-bit arithmetic shift right of d1, giving d2.

Implementation:
opcode: DASR
0: SOURCE=DS DEST=DLO JMP=01s ;;
( non-negative: shift in a 0 )
2: ALU=A CIN=0 SR[ALU]
  SR[DLO] JMP=100 ;;
( negative: shift in a 1 )
3: ALU=A CIN=1 SR[ALU] SR[DLO] ;;
4: SOURCE=DLO DEST=DS
  DECODE ;;

DBASE!

Set DBASE Register
( n1 → )

Encoding:
0x5C 2 cycles

Operation:
Store value n1 in register DBASE.

Implementation:
opcode: DBASE!
0: SOURCE=DHI DEST=DBASE ;;
1: SOURCE=DS INC[DP] ALU=B DECODE ;;
DBASE+!

Add To DBASE
(n1 →)

Encoding:
0xEB 3 cycles

Operation:
Add the value n1 to the contents of DBASE, and place the sum back in the DBASE register.

Implementation:
opcode: DBASE+!
0: SOURCE=DBASE ALU=A+B ;;
1: SOURCE=DHI DEST=DBASE ;;
2: SOURCE=DS INC[DP] ALU=B DECODE ;;

DBASE+_!

Store Indexed With DBASE
(n1 offset2 →)

Encoding:
0xD8 4 cycles

Operation:
Store value n1 at address computed by adding offset2 to the value of DBASE.

Implementation:
opcode: DBASE+_!
0: SOURCE=DHI ADDR=BUS+DBASE(CYCLE) ;;
1: SOURCE=DS INC[DP] DEST=RAM-1 ;;
2: ;;
3: SOURCE=DS INC[DP] ALU=B DECODE ;;
DBASE+_

Load Indexed With DBASE
( offset1 → n2 )

Encoding: 0xD7 4 cycles

Operation: Fetch value n2 from address computed by adding offset1 to the value of DBASE.

Implementation:
opcode: DBASE+_@
0: SOURCE=DHI ADDR=BUS+DBASE(CYCLE) ;
1: ;
2: ;
3: SOURCE=RD ALU=B DECODE ;

DBASE@

Get From DBASE
( → n1 )

Encoding: 0x5D 2 cycles

Operation: Fetch the value n1 from register DBASE.

Implementation:
opcode: DBASE@
0: DEC[DP] DS-FROM-DHI
   SOURCE=DBASE ALU=B ;
1: DECODE ;
**DDROP**

Double Precision Drop

( d1 → )

**Encoding:**

0x75 2 cycles

**Operation:**

Drop d1 from the data stack.

**Implementation:**

opcode: DDROP

0: INC[DP] ;;
1: SOURCE=DS INC[DP] ALU=B DECODE ;;

---

**DDUP**

Double Precision Duplicate

( d1 → d1 d1 )

**Encoding:**

0x76 2 cycles

**Operation:**

Duplicate d1 on the data stack.

**Implementation:**

opcode: DDUP

0: SOURCE=DS ALU=B DEC[DP] DS-FROM-DHI ;;
1: SOURCE=DS ALU=B DEC[DP] DS-FROM-DHI DECODE ;;
DISABLE

Disable Interrupts

(→)

Encoding:

0x6C 

3 cycles

Operation:

Disable maskable interrupts by setting the mask flag in CONFIG register.

Notes:

This implementation may allow an interrupt to be accepted just after the opcode executes. Future chip versions may include a NOP in the microcode to change this behavior.

Implementation:

opcode: DISABLE
0: DHI[1] SOURCE=CONFIG
    ALU=B SR[ALU] ;;
1: DHI[1] ALU=A+A+1 ;;
    ( Sets lowest bit )
2: SOURCE=DHI[1] DEST=CONFIG DECODE ;;

DLSL

Double Precision Logical Shift Left

(d1 → d2)

Encoding:

0x7E 

3 cycles

Operation:

Shift d1 left one bit, shifting in a zero bit, resulting in d2 (same as multiplication by 2).

Implementation:

opcode: DLSL
0: SOURCE=DS DEST=DLO ;
1: SL[ALU] SL[DLO] CIN=0 ;;
2: SOURCE=DLO DEST=DS DECODE ;;
DLSR

Double Precision Logical Shift Right

( d1 → d2 )

**Encoding:**

0xA6  
3 cycles

**Operation:**

Logical shift d1 right 1 bit, shifting a 0 into the highest bit, resulting in d2.

**Implementation:**

opcode: DLSR

0: SOURCE=DS DEST=DLO ;;
1: CIN=0 SR[ALU] SR[DLO] ;;
2: SOURCE=DLO DEST=DS DECODE ;;

DNEGATE

Double Precision Negate

( d1 → d2 )

**Encoding:**

0x85  
4 cycles

**Operation:**

Take the two's complement of d1, giving d2.

**Implementation:**

opcode: DNEGATE

0: SOURCE=DS DHI[1] ALU=notB ;;
1: DHI[1] ALU=A+1 ;;
2: SOURCE=DHI[1] DEST=DS
   ALU=notA JMP=10C ;;
4: ALU=A DECODE ;;
5: ALU=A+1 DECODE ;;
docon

Constant Value

( → n1 )
RS( addr2 → )

Encoding:
0x2E 4 cycles

Operation:
Fetch the in-line 32-bit value n1 and perform a subroutine return to address addr2

Notes:
MUST be compiled as an EXIT instruction for proper operation.

Implementation:
opcode: docon
0: SOURCE=RETURN-SAVE
   ADDR=BUS+0 (CYCLE) ;;
1: ;;
2: ;
3: DEC[DP] DS-FROM-DHI
   SOURCE=RD ALU=B DECODE ;;

dovar

Variable Address

( → addr1 )
RS( addr2 → )

Encoding:
0x2F 2 cycles

Operation:
Return the address of an in-line 32-bit variable as addr1 and perform a subroutine return to addr2 (MUST be compiled as an EXIT instruction for proper operation).

Implementation:
opcode: dovav
0: DEC[DP] DS-FROM-DHI
   SOURCE=RETURN-SAVE ALU=B ;;
1: DECODE ;;
DOVER

Double Precision Get Second Element

( d1 d2 → d1 d2 d1 )

Encoding:
0x7F 6 cycles

Operation:
Perform a double-precision OVER operation, copying d1 to the top of the stack.

Implementation:
opcode: DOVER
0: INC[DP] ALU=A DEST=DHI[1];
1: INC[DP];
2: SOURCE=DS DEC[DP] DEST=DLO;
3: SOURCE=DS DEC[DP] ALU=B;
5: SOURCE=DLO DEC[DP] DEST=DS DECODE;
Implementation:
opcode: DR>
0: DEC[DP] DS-FROM-DHI
    SOURCE=RS INC[RP] ALU=B
1: DEC[DP] DS-FROM-DHI
    SOURCE=RS INC[RP] ALU=B DECODE

DROP

Drop
(n1 →)

Encoding:
0x31  2 cycles
0x32  1 cycle (2OPS format)

Operation:
Pop the top stack element n1, discarding it.

Implementation:
opcode: DROP
0: SOURCE=DS INC[DP] ALU=B
1: DECODE

opcode: DROP_FAST
0: SOURCE=DS INC[DP] ALU=B DECODE
### DROP_LIT

**Drop Then Push Immediate**

( n1 → n2 )

**Encoding:**

0xA7 / lit  
2 cycles

**Operation:**

Drop n1 from the top stack element, then replace it by pushing on n2 from the compiled instruction literal field.

**Implementation:**

opcode: DROP_LIT
0: SOURCE=LIT ALU=B ;;
1: DECODE ;;

### DROT

**Double Precision Rotate**

( d1 d2 d3 → d2 d3 d1 )

**Encoding:**

0x81  
9 cycles

**Operation:**

Move d1 from the third element on the data stack to the top element.

**Notes:**

Do not combine this opcode with a CALL or EXIT instruction.

Do not execute this opcode as the opcode immediately preceding a subroutine return instruction (will be corrected on later versions of the chip). Workaround: form a 2OPS instruction with DROT as the first opcode and NOP_FAST as the second opcode, then place the subroutine return as the following instruction.

**Implementation:**

opcode: DROT
0: SOURCE=DS INC[DP] DEST=DLO ;;
1: SOURCE=DS INC[DP] DEC[RP] DEST=RS ;;
   SOURCE=DS ALU=B INC[RP] ;;
DS!

Store Into Data Stack
(n1 addr2 → )

Encoding:
0xC6 7 cycles

Operation:
Using the Data Stack as a separately addressed memory, store value n1 at address addr2. The highest bits of addr2 (outside the range of the stack memory space) are ignored. No checking is performed to detect interference with stack operation, and this opcode will generate a stack overflow/underflow interrupt if addr2 is outside the permissible stack operating range.

Implementation:
opcode: DS!
0: SOURCE=DP DHI[1] ALU=B
1: SOURCE=DHI[0] DEST=DP
   DHI[1] ALU=A+1
2: SOURCE=DS DEST=DLO
3: SOURCE=DLO DEST=DS
4: SOURCE=DHI[1] DEST=DP
5: ;
6: SOURCE=DS INC[DP] ALU=B DECODE
DS@

Load From Data Stack
( addr1 \rightarrow n2 )

Encoding:
0xC5 5 cycles

Operation:
Using the Data Stack as a separately addressed memory, fetch the value n2 from address addr1. The highest bits of addr1 (outside the range of the stack memory space) are ignored. No checking is performed to detect interference with stack operation, and this opcode will generate a stack overflow/underflow interrupt if addr1 is outside the permissible stack operating range.

Implementation:
opcode: DS@
0: SOURCE=DP DEST=DLO ;;
1: SOURCE=DHI DEST=DP ;;
2: ;;
3: SOURCE=DLO DEST=DP ;;
   ( Exploits fact that DSREG maintains value
    while waiting for the new DP to access
    the DS RAM. )
4: SOURCE=DS ALU=B DECODE ;;

DSWAP

Double Precision Swap
( d1 d2 \rightarrow d2 d1 )

Encoding:
0x80 5 cycles

Operation:
Exchange the top two double precision stack elements, moving d1 from the second stack position to the top of the stack.

Implementation:
opcode: DSWAP
1: SOURCE=DS INC[DP] DEST=DLO ;;
2: DHI[1] DS-FROM-DHI
   SOURCE=DS ALU=B ;;
3: DHI[0] DEC[DP] DS-FROM-DHI
   SOURCE=DLO ALU=B ;;
4: SOURCE=DHI[1]
   DEC[DP] DEST=DS DECODE ;;
**DUP**

**Duplicate**

\[( n1 \rightarrow n1 n1 )\]

**Encoding:**

- 0x33: 2 cycles
- 0x34: 1 cycle (2OPS format)

**Operation:**

Duplicate the top element \( n1 \) on the stack.

**Implementation:**

- **opcode:** DUP
  - 0: \( \text{SOURCE=DHI} \) \( \text{DEC[DP]} \) \( \text{DEST=DS} \) ;;
  - 1: \text{DECODE} ;;

- **opcode:** DUP\_FAST
  - 0: \( \text{SOURCE=DHI} \) \( \text{DEC[DP]} \) \( \text{DEST=DS} \) \( \text{DECODE} \) ;;
  - 2: \text{ALU=0} \text{ DECODE} ;;
  - 3: \text{ALU=-1} \text{ DECODE} ;;

---

**DUP\_0<**

**Test For Less Than Zero (Non-Destructive)**

\[( n1 \rightarrow n1 \text{flag2} )\]

**Encoding:**

- 0xA8: 2 cycles

**Operation:**

Nondestructively test the top stack element \( n1 \), returning a true \( \text{flag2} \) if \( n1 \) is less than zero, else returning a false \( \text{flag2} \).

**Implementation:**

- **opcode:** DUP\_0<
  - 0: \( \text{SOURCE=DHI} \) \( \text{DEC[DP]} \) \( \text{DEST=DS} \) \( \text{JMP=01S} \) ;;
  - 2: \text{ALU=0} \text{ DECODE} ;;
  - 3: \text{ALU=-1} \text{ DECODE} ;;
**DUP_@**

Load (Non-Destructive)

( addr1 \(\rightarrow\) addr1 n2 )

**Encoding:**

0xA9 4 cycles

**Operation:**

Perform a nondestructive load operation, leaving addr1 on the stack unchanged, and fetching value n2 from memory at addr1.

**Implementation:**

opcode: DUP_@

0: SOURCE=DHI ADDR=BUS+0(CYCLE) ;;
1: SOURCE=DHI DEC[DP] DEST=DS ;;
2: ;;
3: SOURCE=RD ALU=B DECODE ;;

---

**ENABLE**

Enable Interrupts

( \(\rightarrow\) )

**Encoding:**

0x6B 3 cycles

**Operation:**

Enable maskable interrupts by clearing the mask flag in CONFIG register.

**Notes:**

This implementation may not allow an interrupt to be accepted just after the opcode executes. Future chip versions may include a NOP in the microcode to change this behavior.

**Implementation:**

opcode: INT_ENABLE

0: DHI[1] SOURCE=CONFIG
   ALU=B SR[ALU] ;;
1: DHI[1] ALU=A+A ;; (lears lowest bit)
2: SOURCE=DHI[1] DEST=CONFIG DECODE ;;
EXECUTE

Call Data Stack Address
( addr1 → )

Encoding:
0x35 /call  4 cycles

Operation:
Perform a subroutine call or jump to addr1.

Notes:
This opcode must be compiled with a CALL instruction to any address (e.g. address 0) to accomplish a subroutine call. This dummy address will not be called, but rather the addr1 value from the stack will be substituted as the subroutine call target.

If a JNEXT instruction instead of a CALL instruction is used, this opcode performs an unconditional branch to the value addr1 on the data stack.

A 2OPS instruction or EXIT instruction may also be used, but the user is cautioned to beware of the effects on program control flow.

Implementation:
opcode: EXECUTE
0: SOURCE=DHI ADDR=BUS+0(CYCLE) ;;
1: SOURCE=DS INC[DP] ALU=B ;;
2: LATCH-INSTRUCTION ;;
3: DECODE ;;

FALSE

Push False Flag
( → 0 )

Encoding:
0x07  2 clocks

Operation:
Pushes constant 0. 0 is the “FALSE” flag value for the machine.

Implementation:
opcode: FALSE
0: DEC[DP] DS-FROM-DHI ALU=0 ;
1: DECODE ;
**FETCH_AND_ADD**

*Add To Memory*

\[( n1 \text{ addr2} \rightarrow n3 )\]

\[( \text{ addr2} \rightarrow n3 ) \text{ Immediate}\]

**Encoding:**

- 0xC8 7 cycles
- 0xC9 / lit 7 cycles

**Operation:**

Fetch the value from word addr2, and add it to n1 to produce n3. Store the value n3 back at the effective address, all as an atomic read/modify/write bus operation. Useful as a synchronization instruction. The immediate variant provides the value n1 from the literal field.

**Implementation:**

- **opcode**: LIT_FETCH_AND_ADD
  0: SOURCE=DHI ADDR=BUS+0(CYCLE) ;
  1: RAM-RMW ;
  2: SOURCE=LIT ALU=B ;
  3: SOURCE=RD ALU=A+B CYCLE-RAM ;
  4: SOURCE=DHI DEST=RAM-1 ;
  5: ;
  6: DECODE ;

- **opcode**: FETCH_AND_ADD
  0: SOURCE=DHI ADDR=BUS+0(CYCLE) ;
  1: RAM-RMW ;
  2: SOURCE=DS INC[DP] ALU=B ;
  3: SOURCE=RD ALU=A+B CYCLE-RAM ;
**FILL**

**Fill Memory With Characters**

( addr1 count2 c3 \rightarrow )

**Encoding:**

0x8C 4 + 2*count cycles

**Operation:**

Fill count2 bytes of memory with value c3, starting at address addr1 and counting up.

**Notes:**

Destroys value in the DBASE register.

**Implementation:**

opcode: FILL
( Keep count/offset in DHI[1] )
0: SOURCE=DS INC[DP] DHI[1] ALU=B
1: SOURCE=DS INC[DP] DEST=DBASE
   DHI[1] ALU=A-1 ;;
   ( Store bytes in reverse order, using count as offset )
2: SOURCE=DHI[1] ADDR=BUS+DBASE(CYCLE)
   DHI[1] ALU=A-1 JMP=110 ;;
6: SOURCE=DHI[0] DEST=RAM-C! JMP=01S ;;
3: ;;
   ( Wait for RAM cycle to complete)
4: SOURCE=DS INC[DP] ALU=B DECODE ;;

---

**FP!**

**Put Frame Pointer Value**

( n1 \rightarrow )

**Encoding:**

0x60 2 cycles

**Operation:**

Store the value n1 into the FP register.

**Implementation:**

opcode: FP!
0: SOURCE=DHI DEST=FP ;;
1: SOURCE=DS INC[DP] ALU=B DECODE ;;
**FP + !**

*Add To Frame Pointer Value*

\((n_1 \rightarrow)\)
\((\rightarrow)\) Immediate

**Encoding:**
0xE8 \hspace{1em} 3 cycles
0xE9/lit \hspace{1em} 3 cycles

**Operation:**
Add the value \(n_1\) to the contents of FP, and place the sum back in the FP register. In the immediate variant the value \(n_1\) is provided by the literal field.

**Implementation:**

```
opcode: FP+!
0: SOURCE=FP ALU=A+B ;;
1: SOURCE=DHI DEST=FP ;;
2: SOURCE=DS INC[DP] ALU=B DECODE ;;
```

```
opcode: LIT_FP+!
0: SOURCE=FP DHI[1] ALU=B ;;
1: SOURCE=LIT DHI[1] ALU=A+B ;;
2: SOURCE=DHI[1] DEST=FP DECODE ;;
```

---

**FP @**

*Get FP Register*

\((\rightarrow n_1)\)

**Encoding:**
0x61 \hspace{1em} 2 cycles

**Operation:**
Fetch the value of the FP register, returning \(n_1\).

**Implementation:**

```
opcode: FP@
0: DEC[DP] DS-FROM-DHI
    SOURCE=FP ALU=B ;;
1: DECODE ;;
```
**FRAME_POP**  
Frame Deallocate Immediate  
(→)

**Encoding:**  
0xED  
4 cycles

**Operation:**  
Using the FP register, deallocate a memory-resident stack frame. FP is loaded with the value contained in the memory word address by adding the old FP contents and the compiled instruction literal field. The compiled instruction literal field must be -4 to work properly with FRAME_PUSH.

**Implementation:**  
opcode: FRAME_POP  
0: SOURCE=LIT ADDR=BUS+FP(CYCLE) ;;  
1: ;;  
2: ;;  
3: SOURCE=RD DEST=FP DECODE ;;  

---

**FRAME_PUSH**  
Frame Allocate Immediate  
(→)

**Encoding:**  
0xEC  
6 cycles

**Operation:**  
Using the FP register, allocate a number of bytes (determined by the compiled instruction literal field) to the memory-resident stack frame. The new value of FP is calculated by adding the old value of FP to the compiled instruction literal field. The old value of the FP register is saved at the address new FP-4.

**Implementation:**  
opcode: FRAME_PUSH  
0: DHI[1] SOURCE=FP ALU=B DEST=DLO ;;  
1: DHI[1] SOURCE=LIT ALU=A-B ;;  
2: SOURCE=DHI[1] ADDR=BUS-4(CYCLE) ;;  
3: SOURCE=DLO DEST=RAM-1 ;;  
4: SOURCE=DHI[1] DEST=FP ;;  
5: DECODE ;;
**H!**

**Store Halfword**

\((h1 \rightarrow addr2)\)

**Encoding:**

\(0x36\)

4 cycles

**Operation:**

Store half-word value \(h1\) at address \(addr2\).
The lowest 16 bits of value \(h1\) are stored.

**Implementation:**

- **opcode:** H!
- **0:** SOURCE=DHI ADDR=BUS+0(CYCLE) ;;
- **1:** SOURCE=DS DEST=RAM-W I INC[DP] ;;
- **2:** ;;
- **3:** SOURCE=DS ALU=B INC[DP] DECODE ;;

**H@**

**Load Halfword**

\((addr1 \rightarrow h2)\)

**Encoding:**

\(0x37\)

4 cycles

**Operation:**

Fetch halfword value \(h1\) from address \(addr\).
The 16-bit value is sign-extended to 32 bits.

**Implementation:**

- **opcode:** H@
- **0:** SOURCE=DHI ADDR=BUS+0(CYCLE) ;;
- **1:** RAM-W@ ;;
- **2:** ;;
- **3:** SOURCE=RD-SIGNED ALU=B DECODE ;;
HALT

**Halt Processor**

(→)

**Encoding:**

0x05 ∞ cycles

**Operation:**

Enter an infinite microcode loop that is uninterruptible.

**Implementation:**

opcode: HALT
0: ;;
1: ;;
2: ;;
3: ;;
4: ;;
5: ;;
6: ;;
7: JMP=000 ;;

I'

**Copy Second On Rstack To Dstack**

(→ n1)

RS(n1 n2 → n1 n2)

**Encoding:**

0x3B 2 cycles

**Operation:**

Copy the second return stack element onto the data stack. In Looping constructs, this returns the loop limit value.

**Notes:**

Do not combine this opcode with a CALL or EXIT instruction.

Do not execute this opcode as the opcode immediately preceding a subroutine return instruction (will be corrected on later versions of the chip). Workaround: form a 2OPS instruction with I' as the first opcode and NOP_FAST as the second opcode, then place the subroutine return as the following instruction.

**Implementation:**

opcode: I'
0: SOURCE=DHI INC[RP] DEC[DP] DEST=DS ;;
1: SOURCE=RS DEC[RP] ALU=B DECODE ;;
I'_!

**Store With Rstack2**
(n1 →)
RS( addr2 n3 → addr2 n3)

**Encoding:**
0xE2 5 cycles

**Operation:**
Store n1 at location addr2.

**Notes:**
Do not combine this opcode with a CALL or EXIT instruction.

**Implementation:**
opcode: I'_!
0: INC[RP] ;
1: SOURCE=RS DEC[RP]
   ADDR=BUS+0(CYCLE) ;
2: SOURCE=DHI DEST=RAM-1 ;
3: ;
4: SOURCE=DS INC[DP] ALU=B
   DECODE ;

I'_4--_

**Store With Rstack2 Minus 4**
(n1 →)
RS( addr2 n3 → addr2 n3)

**Encoding:**
0xE5 5 cycles

**Operation:**
Store n1 at location addr2.

**Notes:**
Do not combine this opcode with a CALL or EXIT instruction.

**Implementation:**
opcode: I'_4--_
0: INC[RP] ;
1: SOURCE=RS DEC[RP]
   ADDR=BUS-4(CYCLE) ;
2: SOURCE=DHI DEST=RAM-1 ;
3: ;
4: SOURCE=DS INC[DP] ALU=B
   DECODE ;

Harris Semiconductor Proprietary 67
I'\_@

Load With Rstack2

\((\rightarrow n1)\)
RS( addr2 n3 \rightarrow addr2 n3)

Encoding:
0xE3 5 cycles

Operation:
Fetch n1 from address addr2.

Notes:
Do not combine this opcode with a CALL or EXIT instruction.

Implementation:
opcode: I'\_@
0: INC [RP];
1: SOURCE=RS DEC [RP]
   ADDR=BUS+0 (CYCLE);;
2: ; ;
3: SOURCE=DHI DEC [DP] DEST=DS ;;
4: SOURCE=RD ALU=B DECODE ;;

I_+

Add Return Stack (Non-Destructive)

\((n1 \rightarrow n2)\)
RS( n3 \rightarrow n3)

Encoding:
0x3A 2 cycles

Operation:
Add the top return stack element n3 to the top data stack element n1, giving n2=n1+n3. Note that the return stack element is not popped.

Notes:
Do not combine this opcode with a CALL or EXIT instruction.

Implementation:
opcode: I_+
0: ; ;
1: SOURCE=RS ALU=A+B DECODE ;;
Copy Third On Return Stack

( \rightarrow n1 \)

RS( n1 n2 n3 \rightarrow n1 n2 n3 )

Encoding:

0x3C 4 cycles

Operation:

Copy the third return stack element n1 onto the data stack. In a doubly-nested DO loop, this returns the value of the index of the outer loop.

Notes:

Do not combine this opcode with a CALL or EXIT instruction.

Do not execute this opcode as the opcode immediately preceding a subroutine return instruction (will be corrected on later versions of the chip). Workaround: form a 2OPS instruction with J as the first opcode and NOP_FAST as the second opcode, then place the subroutine return as the following instruction.

Implementation:

opcode: J
0: SOURCE=DHI INC[RP]
   DECODE
3: DEC[RP] DECODE
1: INC[RP] ;
2: SOURCE=RS DEC[RP] ALU=B ;
LEAVE

Terminate Loop
(→)
RS(n1 n2 → n2 n2)

Encoding:
0xF2 3 cycles

Operation:
Set the loop limit n1 equal to the current index n2, terminating a DO..LOOP at the end of the current iteration. Works with <loop> and <+loop>.

Notes:
Do not combine this opcode with a CALL or EXIT instruction.

Implementation:
opcode: LEAVE
0: SOURCE=RS DEST=DLO ;;
1: SOURCE=DLO INC[RP] DEST=RS ;;
2: DEC[RP] DECODE ;;

LOAD_DS

Streamed Read Data Stack
(addr1 n2 → n.1 n.2 n.3 ... n.n2)

Encoding:
0xDB 3 + 3*count cycles

Operation:
Read n2 consecutive memory words, starting at memory location addr1, onto the data stack. This is a streamed data stack load with auto-increment. The first word read will be the deepest on the stack.

Implementation:
opcode: LOAD_DS
0: SOURCE=DS INC[DP] DEST=SBASE
   DHI[1] ALU=0 ;;
1: SOURCE=DHI[1] ADDR=BUS+SBASE(CYCLE)
   DHI[0] ALU=A-1 JMP=011 ;;
( Loop for fetching)
2: SOURCE=RD DEC[DP] DEST=DS
   DHI[0] ALU=A-1 ;;
4: SOURCE=DHI[1] ADDR=BUS+SBASE(CYCLE)
   JMP=010 ;;
( Finish up)
5: ;;
6: SOURCE=RD ALU=B DECODE ;;
LOAD_RS

Streamed Read Return Stack

( addr1 n2 –> )
RS( → n.1 n.2 n.3 ... n.n2 )

Encoding:
0xDC  4 + 3*count cycles

Operation:
Read n2 consecutive words, starting at
memory location addr1, onto the return stack.
This is a streamed data stack load with auto-
increment. The first word read will be deepest
on the return stack.

Notes:
Do not combine this opcode with a CALL or
EXIT instruction.

Implementation:
opcode: LOAD_RS

0: SOURCE=DS INC[DP] DEST=SBASE
   DHI[1] ALU=0 ;;
1: SOURCE=DHI[1] ADDR=BUS+SBASE(CYCLE)
   DHI[0] ALU=A-1 JMP=011 ;;
( Loop for fetching )
2: SOURCE=RD DEC[RP] DEST=RS
   DHI[0] ALU=A-1 ;;
4: SOURCE=DHI[1] ADDR=BUS+SBASE(CYCLE)
   JMP=010 ;;

( Finish up )
5: ;;
6: SOURCE=RD DEC[RP] DEST=RS ;;
7: SOURCE=DS INC[DP] ALU=B DECODE ;;
LOC_!

Local Store
(n1 →) Immediate

Encoding:
0xD0 / lit 4 cycles

Operation:
Store the value n1 at the effective address computed by adding the FP value to the compiled instruction literal.

Implementation:
opcode: LOC_I
0: SOURCE=LIT ADDR=BUS+FP(CYCLE) ;
1: SOURCE=DHI DEST=RAM-1 ;
2: ;
3: SOURCE=DS INC[DP] ALU=B
DECODE ;

LOC_+!

Local Add To Memory
(n1 →) Immediate

Encoding:
0xD4 / lit 7 cycles

Operation:
Performs an atomic addition to the word at the address (FP+lit) with value n1, where FP is the FP register value and lit is the compiled instruction literal value.

Implementation:
opcode: LOC_+I
0: SOURCE=LIT ADDR=BUS+FP(CYCLE) ;
1: ;
2: ;
3: SOURCE=RD ALU=A+B CYCLE-RAM ;
4: SOURCE=DHI DEST=RAM-1 ;
5: ;
6: SOURCE=DS INC[DP] ALU=B DECODE ;
**LOC_@**

Local Load
(→ n1) Immediate

**Encoding:**
0xCC / lit 4 cycles

**Operation:**
Fetch the value n1 from the effective address computed by adding the FP value to the compiled instruction literal.

**Implementation:**
opcode: LOC_@
0: SOURCE=LIT ADDR=BUS+FP(CYCLE) ;;
1: ;;
2: SOURCE=DHI DEC[DP] DEST=DS ;;
3: SOURCE=RD ALU=B DECODE ;;

**LOC_@_!**

Local Indirect Store
(n1 →) Immediate

**Encoding:**
0xD3 7 cycles

**Operation:**
Perform an indirect store of the value n1 at the effective address computed by adding the FP value to the compiled instruction literal.

**Implementation:**
opcode: LOC_@_!
0: SOURCE=LIT ADDR=BUS+FP(CYCLE) ;;
1: ;;
2: ;;
3: SOURCE=RD ADDR=BUS+0(CYCLE) ;;
4: SOURCE=DHI DEST=RAM-1 ;;
5: ;;
6: SOURCE=DS INC[DP] ALU=B DECODE ;;
**LOC@+**

Local Load and Add

( n1 → n2 ) Immediate

**Encoding:**

0xCF / lit  

4 cycles

**Operation:**

Fetch the value from the effective address computed by adding the FP value to the compiled instruction literal, then add it to n1, producing n2.

**Implementation:**

opcode: FP+LIT_@_+

0: SOURCE=LIT  ADDR=BUS+FP(CYCLE)  ;
1: ;
2: ;
3: SOURCE=RD  ALU=A+B  DECODE  ;

**LOC@@**

Local Load Indirect

( → n1 ) Immediate

**Encoding:**

0xD2 / lit  

7 cycles

**Operation:**

Perform an indirect fetch of the value n1 from the effective address computed by adding the FP value to the compiled instruction literal.

**Implementation:**

opcode: FP+LIT_@_@

0: SOURCE=LIT  ADDR=BUS+FP(CYCLE)  ;;
1: ;;
2: SOURCE=DHI  DEC[DP]  DEST=DS  ;;
3: SOURCE=RD  ADDR=BUS+0(CYCLE)  ;;
4: ;;
5: ;;
6: SOURCE=RD  ALU=B  DECODE  ;;
**LOC_B@**

**Local Load Byte**

(→ b1) Immediate

**Encoding:**

```
0xCE  4 cycles
```

**Operation:**

Fetch the sign-extended value b1 from the effective address computed by adding the FP value to the compiled instruction literal.

**Implementation:**

```
opcode: LOC_B@
0: SOURCE=LIT ADDR=BUS+FP(CYCLE) ;;
1: RAM-C@ ;;
2: SOURCE=DHI DEC[DP] DEST=DS ;;
3: SOURCE=RD-SIGNED ALU=B DECODE ;;
```

---

**LOC_C!**

**Local Store Character**

(→ c1) Immediate

**Encoding:**

```
0xD1/lit  4 cycles
```

**Operation:**

Store the value c1 at the effective address computed by adding the FP value to the compiled instruction literal.

**Implementation:**

```
opcode: LOC_C!
0: SOURCE=LIT ADDR=BUS+FP(CYCLE) ;;
1: SOURCE=DHI DEST=RAM-C1 ;;
2: ;;
3: SOURCE=DS INC[DP] ALU=B DECODE ;;
```
**LOC_C@**

**Local Load Character**

(→ c1) Immediate

**Encoding:**

0xCD / lit 4 cycles

**Operation:**

Fetch the value c1 from the effective address computed by adding the FP value to the compiled instruction literal. c1 is placed in the lowest 8 bits of the stack word and padded with 0s in the high bits.

**Implementation:**

opcode: LOC_C@

0: SOURCE=LIT ADDR=BUS+FP(CYCLE) ;
1: RAM-C@ ;
2: SOURCE=DHI DEC[DP] DEST=DS ;
3: SOURCE=RD ALU=B DECODE ;

---

**LSLN**

**Logical Shift Left By N**

( n1 count2 → n3 )

( n1 → n3 ) Immediate

**Encoding:**

0xB3 2 + count cycles
0xB4 / lit 3 + count cycles

**Operation:**

Shift n1 left by count2 bits, shifting a 0 into the lowest bit, and discarding the highest bit on each shift. In the immediate variant, the count count2 is supplied by the literal field.

**Implementation:**

opcode: LSLN

0: SOURCE=DS INC[DP] DEST=DLO
   ALU=A-1 JMP=OlZ ;
2: SL[DLO] ALU=A-1 JMP=OlZ ;
3: SOURCE=DLO ALU=B DECODE ;

opcode: LIT_LSLN

0: SOURCE=LIT DHI[1] ALU=B ;
1: SOURCE=DHI[0] DEST=DL0
   DHI[1] ALU=A-1 JMP=OlZ ;
3: SOURCE=DLO ALU=B DECODE ;

Harris Semiconductor Proprietary
**LSRN**

**Logical Shift Right By N**

\(( n1 \text{ count2} \rightarrow n3 )\)

\(( n1 \rightarrow n3 ) \text{ Immediate}\)

**Encoding:**

- 0xB6 \(2 + 2\times\text{count cycles}\)
- 0xB7/\text{lit} \(2 + 2\times\text{count cycles}\)

**Operation:**

Shift \(n1\) right by \(\text{count2}\) bits, shifting a 0 into the highest bit, and discarding the lowest bit on each shift (logical shift right), producing the result \(n3\). The immediate variant uses the literal field to supply the value \(\text{count2}\).

**Implementation:**

```
opcode: LSRN
0: SOURCE=DS INC[DP]  
   DHI[1] ALU=B JMP=01Z ;
2: DHI[0] ALU=A-1 JMP=100 ;
4: DHI[1] CIN=0 ALU=A SR[ALU]  
   JMP=01Z ;
3: SOURCE=DHI[1] ALU=B DECODE ;
```

```
opcode: LIT_LSRN
0: SOURCE=LIT ALU=B DHI[1] ;
1: DHI[1] ALU=A-1 JMP=10Z ;
   (count of 0 is 3 cycles)
5:      DECODE ;;
```
**LSR**

**Logical Shift Right**

\((n1 \rightarrow n2)\)

**Encoding:**
0xB5 2 cycles

**Operation:**
Shift n1 right one bit, shifting a 0 bit into the highest order bit position (logical shift right).

**Implementation:**

```
opcode: LSR
0: CIN=0 SR[ALU] ;;
1: DECODE ;;
```

---

**M+**

**Add Double To Single**

\((d1 n2 \rightarrow d3)\)

**Encoding:**
0x8F 4 cycles

**Operation:**
Mixed precision addition, where n2 is sign-extended to 64 bits, then added to d1, giving d3.

**Implementation:**

```
opcode: M+
1: SOURCE=DS ALU=A+B ;;
2: SOURCE=DHI DEST=DS JMP=10C ;;
4: DHI[1] ALU=A DEST=DHI[0] DECODE ;;
```
MOVE

Move Words
( addr1 addr2 n3 → )

Encoding:
0xB 3 + 5*cnt cycles

Operation:
Move n3 words of memory from address addr1 to address addr2.

Notes:
An n3 value of 0 will attempt to move 4G words.

MOVE destroys the value of registers SBASE and DBASE.

This is a non-interruptible instruction. The commercial chip will have an interruptible version of this opcode.

Implementation:
opcode: MOVE
0: SOURCE=DS INC[DP] DEST=DBASE
1: SOURCE=DS INC[DP] DEST=SBASE
   DHI[1] ALU=A-1
( Word move loop )
2: SOURCE=DHI[1] ADDR=BUS+SBASE(CYCLE)
   JMP=100
4: 
5: SOURCE=DHI[1] ADDR=BUS+DBASE(CYCLE)
**MRAM!**

**Microcode RAM Store**

\[( n1 \text{ addr2 } \rightarrow )\]

**Encoding:**

0x62 \hspace{1cm} 3 cycles

**Operation:**

Store value n1 at address addr2 within the MRAM memory. The highest order bits of addr2 are ignored when addressing the memory.

**Implementation:**

\[
\text{opcode: MRAM1} \\
0: \text{SOURCE=DHI \hspace{0.5cm} DEST=MICRO-ADR \hspace{0.5cm} ; ;} \\
1: \text{SOURCE=DS \hspace{0.5cm} INC[DP] \hspace{0.5cm} DEST=MRAM \hspace{0.5cm} ; ;} \\
2: \text{SOURCE=DS \hspace{0.5cm} INC[DP] \hspace{0.5cm} ALU=B \hspace{0.5cm} DECODE \hspace{0.5cm} ; ;}
\]

---

**MRAM@**

**Microcode RAM Load**

\[( \text{ addr1 } \rightarrow n2)\]

**Encoding:**

0x63 \hspace{1cm} 3 cycles

**Operation:**

Fetch value n2 from address addr1 within the MRAM memory. The highest order bits of addr1 are ignored when addressing the memory.

**Implementation:**

\[
\text{opcode: MRAM@} \\
0: \text{SOURCE=DHI \hspace{0.5cm} DEST=MICRO-ADR \hspace{0.5cm} ; ;} \\
1: \text{SOURCE=MRAM \hspace{0.5cm} ALU=B \hspace{0.5cm} ; ;} \\
2: \text{SOURCE=MRAM \hspace{0.5cm} ALU=B \hspace{0.5cm} DECODE \hspace{0.5cm} ; ;}
\]

Harris Semiconductor Proprietary
**MROM@**

**Microcode ROM Load**

( addr1 → n2 )

**Encoding:**

n/a

3 cycles

**Operation:**

Fetch value n2 from address addr1 within the MROM memory. The highest order bits of addr1 are ignored when addressing the memory.

**Notes:**

This opcode must be executed from MRAM to work properly, since a program may not both read MROM and execute from it at the same time. The implementation given below may be used by application programs if desired.

**Implementation:**

opcode: MROM@

0: SOURCE=DHI DEST=MICRO-ADR ;;
   ( Can't put DECODE in next microinstruction in case of MROM opcode )
1: SOURCE=MROM ALU=B ;;
2: SOURCE=MROM ALU=B DECODE ;;

**NEGATE**

**Two’s Complement Negation**

( n1 → n2 )

**Encoding:**

0x84

2 cycles

**Operation:**

Take the two's complement of n1, returning it as n2.

**Implementation:**

opcode: NEGATE

0: ALU=notA ;;
1: ALU=A+1 DECODE ;;
### NIP

**Drop Second On Stack**

(n1 \(\rightarrow\) n2)

**Encoding:**

- \(0x77\) 2 cycles
- \(0x78\) 1 cycle (2OPS format)

**Operation:**

Drop the second element on the data stack n1, putting n2 in its place and decreasing the stack size by 1 element.

**Implementation:**

- opcode: NIP
  - 0: INC [DP] ;
  - 1: DECODE ;

- opcode: NIP_FAST
  - 0: INC [DP] DECODE ;

### NOP

**No Operation**

(\(\rightarrow\))

**Encoding:**

- \(0x00\) 2 cycles
- \(0x06\) 1 cycle (2OPS format)

**Operation:**

Do nothing, except wasting 1 or 2 clock cycles. NOP should be used to fill the opcode field for CALL, EXIT, and JNEXT instructions for which there is no desired opcode. NOP_FAST may be used as the first or second opcode of a 2OPS instruction which, for some reason, does not have two useful opcodes.

**Implementation:**

- opcode: NOP
  - 0: ;
  - 1: DECODE ;
  - 2: JMP=000 ;
  - 3: JMP=000 ;
  - 4: JMP=000 ;
  - 5: JMP=000 ;
  - 6: JMP=000 ;
  - 7: JMP=000 ;

- opcode: NOP_FAST
  - 0: DECODE ;
**NOT**

One's Complement Negation

\(( n_1 \rightarrow n_2 )\)

**Encoding:**

0x09 2 cycles

**Operation:**

Take the one's complement of \( n_1 \), returning \( n_2 \). This is a bitwise logical complement.

**Implementation:**

opcode: NOT

0: ALU=notA ;;
1: DECODE ;;

---

**not?branch**

Jump If Not Zero

\(( \text{flag1} \rightarrow )\)

**Encoding:**

0x98 / call 4 cycles

**Operation:**

Perform branch if flag1 is non-zero, otherwise continue executing in-line. ?BRANCH must be compiled as the opcode in a CALL instruction that has the branch target as its next address field.

**Implementation:**

opcode: NOT?BRANCH

0: SOURCE=RETURN-SAVE ADDR=BUS+0(CYCLE) ;;
1: INC[RP] JMP=01Z ;;

( Zero flag, fall through )

3: LATCH-INSTRUCTION ;;
4: SOURCE=DS INC[DP] ALU=B DECODE ;;

( Non-zero flag, take the branch )

2: JMP=111 ;;
7: SOURCE=DS INC[DP] ALU=B DECODE ;;
**ONE**

**One**

\[ \rightarrow 1 \]

**Encoding:**

0x6D 2 cycles

**Operation:**

Pushes constant 1.

**Implementation:**

```c
opcode: ONE
0: DEC[DP] DS-FROM-DHI ALU=0 ;;
1: ALU=A+1 DECODE ;;
```

---

**OR**

**Logical Or**

\[ (n1 \land n2) \rightarrow n3 \]

\[ (n1 \rightarrow n3) \text{ Immediate} \]

**Encoding:**

- 0x3D 2 cycles
- 0x3E 1 cycle (2OPS format)
- 0xAF / lit 2 cycles
- 0xB0 / lit 1 cycle (2OPS format)

**Operation:**

Perform a bitwise logical OR operation of \( n1 \) and \( n2 \), giving \( n3 \). For the immediate variant, \( n2 \) is supplied by the literal field.

**Implementation:**

```c
opcode: OR
0: SOURCE=DS INC[DP] ALU=AorB ;;
1: DECODE ;;

opcode: OR_FAST
0: SOURCE=DS INC[DP] ALU=AorB DECODE ;;

opcode: LIT_OR
0: SOURCE=LIT ALU=AorB ;;
1: DECODE ;;

opcode: LIT_OR_FAST
0: SOURCE=LIT ALU=AorB DECODE ;;
```
**OR!**

Logical Or To Memory

( n1 addr2 → )

**Encoding:**

0xA0  7 cycles

**Operation:**

Perform an atomic bitwise logical OR of n1 to the word at memory addr2, returning the result to addr2 (similar to +!, but with an OR operation instead of a + operation).

**Implementation:**

opcode: ORI

0: SOURCE=DHI  ADDR=BUS+0 (CYCLE) ;
1: RAM-RMW ;
2: SOURCE=DS INC[DP]  ALU=B ;
3: SOURCE=RD  ALU=AorB  CYCLE-RAM ;
4: SOURCE=DHI  DEST=RAM-1 ;
5: ;
6: SOURCE=DS INC[DP]  ALU=B  DECODE ;

**OVER**

Copy Second On Stack

( n1 n2 → n1 n2 n1 )

**Encoding:**

0x3F  2 cycles
0x40  1 cycle (2OPS format)

**Operation:**

Copy the second element on the data stack n1 to the top.

**Implementation:**

opcode: OVER

0: SOURCE=DS  ALU=B  DEC[DP]  DS-FROM-DHI ;
1: DECODE ;

opcode: OVER_FAST

0: SOURCE=DS  ALU=B  DEC[DP]  DS-FROM-DHI  DECODE ;
OVER_!

Reversed Store (Non-Destructive)
( addr1 n2 → addr1 )

Encoding:
0xB8 4 cycles

Operation:
Store the value n2 at address addr1, without popping addr1.

Implementation:
opcode: OVER_
0: SOURCE=DS ADDR=BUS+0(CYCLE) ;
1: SOURCE=DHI DEST=RAM-1 ;
2: ;;
3: SOURCE=DS INC[DP] ALU=B DECODE ;;

OVER_+

Add (Non-Destructive to Second Element)
( n1 n2 → n1 n3 )

Encoding:
0xBA 2 cycles

Operation:
Add n1 and n2, giving n3, but without destroying n1.

Implementation:
opcode: OVER_
0: SOURCE=DS ALU=A+B ;
1: DECODE ;
**OVER_@**

Fetch Using Second On Stack (Non-Destructive)

\( \text{addr1 n2} \rightarrow \text{addr1 n2 n3} \)

Encoding:

0xB9 4 cycles

Operation:

Fetch value n3 from addr1, pushing it onto the stack.

Implementation:

opcode: OVER_@

0: SOURCE=DS ADDR=BUS+0(CYCLE) ;;
1: ;;
2: SOURCE=DHI DEC[DP] DEST=DS ;;
3: SOURCE=RD ALU=B DECODE ;;

---

**R+!**

Add To Rstack Immediate

\( (\rightarrow) \)

RS( n1 \rightarrow n2 )

Encoding:

0xC1 4 cycles

Operation:

Pop n1 from the return stack, add to it the compiled instruction literal field, and place the result back onto the return stack as n2.

Notes:

Do not combine this opcode with a CALL or EXIT instruction.

Implementation:

opcode: R+!

0: SOURCE=LIT DHI[1] ALU=B ;;
1: SOURCE=RS DHI[1] ALU=A+B ;;
2: SOURCE=DHI[1] DEST=RS ;;
3: DECODE ;;
**PICK**

**Push Nth On Stack**

\[(n.m \ldots n.1 m \rightarrow n.m \ldots n.1 n.m)\]

**Encoding:**

0x1C \quad 5 cycles

**Operation:**

Copies the \(m\)th value from the data stack to the top of stack. <PICK> is a primitive that does not check for stack underflow, nor for negative input numbers.

**Notes:**

"0 <PICK>" is a no-op. "1 <PICK>" is equivalent to "DUP". "2 <PICK>" is equivalent to "OVER".

This primitive is dangerous in an environment where stack memory is being spilled into program memory. It will not check to see if the input \(m\) is too deep in the stack, nor will it generate a stack underflow interrupt if \(m\) is big enough to cause a wrap-around back into the active stack space.

**Implementation:**

```
opcode: PICK
0: SOURCE=DP DEST=DLO ALU=A+B ;;
1: SOURCE=DHI DEST=DP ;;
2: ;;
3: SOURCE=DLO DEST=DP ;;
```

(Exploits fact that DSREG maintains value while waiting for the new DP to access the DS RAM.)
R >

Transfer From Rstack to Dstack

\( \rightarrow n1 \)
RS( n1 \rightarrow )

Encoding:

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x41</td>
<td>2 cycles</td>
</tr>
<tr>
<td>0x42</td>
<td>1 cycle (2OPS format)</td>
</tr>
</tbody>
</table>

Operation:

Pop n1 from the return stack, placing it on the data stack.

Notes:

Do not combine this opcode with a CALL or EXIT instruction.

Do not execute this opcode as the opcode immediately preceding a subroutine return instruction (will be corrected on later versions of the chip). Workaround: form a 2OPS instruction with R> as the first opcode and NOP_FAST as the second opcode, then place the subroutine return as the following instruction.

Implementation:

opcode: R>
0: SOURCE=DHI DEC[DP] DEST=DS ;;
1: SOURCE=RS INC[RP] ALU=B DECODE ;;

opcode: R>_FAST
0: DEC[DP] DS-FROM-DHI

SOURCE=RS INC[RP] ALU=B DECODE ;;
**R > !_**

**Store With Return Stack**

\[ ( n1 \rightarrow ) \]
\[ RS( \text{addr2} \rightarrow ) \]

**Encoding:**
0xBF  
4 cycles

**Operation:**
Store value n1 at location \( \text{addr2} \), where \( \text{addr2} \) resides on the return stack.

**Notes:**
Do not combine this opcode with a CALL or EXIT instruction.

**Implementation:**
```
opcode: R>_!
0: SOURCE=RS
   ADDR=BUS+0(CYCLE) ; ;
1: SOURCE=DHI
   DEST=RAM-1 ; ;
2: INC[RP] ; ;
3: SOURCE=DS INC[DP] ALU=B
   DECODE ; ;
```

---

**R > _@**

**Load With Return Stack**

\[ ( \rightarrow n1 ) \]
\[ RS( \text{addr2} \rightarrow ) \]

**Encoding:**
0xC0  
4 cycles

**Operation:**
Fetch value n1 from location \( \text{addr2} \), where \( \text{addr2} \) resides on the return stack.

**Notes:**
Do not combine this opcode with a CALL or EXIT instruction.

**Implementation:**
```
opcode: R>_@
0: SOURCE=RS
   ADDR=BUS+0(CYCLE) ; ;
1: INC[RP] ; ;
2: SOURCE=DHI
   DEC[DP] DEST=DS ; ;
3: SOURCE=RD ALU=B DECODE ; ;
```
R@  

Copy Top Of Rstack To Dstack  
( → n1 )  
RS( n1 → n1 )  

Encoding:  
0x38  2 cycles  
0x39  1 cycle (2OPS format)  

Operation:  
Copy the top return stack element onto the data stack.  

Notes:  
This is the same opcode used to access the inner loop index (the I word in Forth).  
Do not combine this opcode with a CALL or EXIT instruction.  

Implementation:  
opcode: I  
0: SOURCE=DHI  DEC[DP]  DEST=RS  
1: SOURCE=RS  ALU=B  DECODE  

opcode: I_FAST  
0: DEC[DP]  DS-FROM-DHI  SOURCE=RS  ALU=B  DECODE  

R@_1  

Store With Return Stack (Non-Destructive)  
( n1 → )  
RS( addr2 → addr2 )  

Encoding:  
0xBD  4 cycles  

Operation:  
Store n1 at location addr2, where addr2 is copied from the return stack.  

Notes:  
Do not combine this opcode with a CALL or EXIT instruction.  

Implementation:  
opcode: R@_1  
0: SOURCE=RS  
   ADDR=BUS+0(CYCLE)  
1: SOURCE=DHI  DEST=RAM-1  
2: ;  
3: SOURCE=DS  INC[DP]  ALU=B  
   DECODE  

Harris Semiconductor Proprietary
**R@_4-_!**

**Store With Return Stack Minus 4 (Non-Destructive)**

\[( n1 \rightarrow )\]

\[RS( addr2 \rightarrow addr2 )\]

**Encoding:**

0xE4 4 cycles

**Operation:**

Store \(n1\) at location \(addr2-4\), where \(addr2\) is copied from the return stack.

**Notes:**

Do not combine this opcode with a CALL or EXIT instruction.

**Implementation:**

opcode: R@_4-_!

0: SOURCE=RS ADDR=BUS-4(CYCLE) ;;
1: SOURCE=DHI DEST=RAM-1 ;;
2: ;;
3: SOURCE=DS INC(DP) ALU=B DECODE ;;

---

**R@_@**

**Load With Return Stack (Non-Destructive)**

\[( \rightarrow n1 )\]

\[RS( addr2 \rightarrow addr2 )\]

**Encoding:**

0xBE 4 cycles

**Operation:**

Fetch \(n1\) from location \(addr2\), where \(addr2\) is copied from the return stack.

**Notes:**

Do not combine this opcode with a CALL or EXIT instruction.

**Implementation:**

opcode: R@_

0: SOURCE=RS ADDR=BUS+0(CYCLE) ;;
1: ;;
2: SOURCE=DHI DEC[DP] DEST=DS ;;
3: SOURCE=RD ALU=B DECODE ;;
RLC

Rotate Left Through Carry

\[( n1 \text{ flag2 } \rightarrow n3 \text{ flag4} )\]

Encoding:

0xBB 4 cycles

Operation:
Perform a one-bit rotate left, using flag2 as the carry-in value (true or false), and n1 as the input number. If flag2 is true, a 1 is rotated into n1, otherwise a 0 is rotated into n1. flag4 is set to true if the bit rotated out of n1 (i.e. bit 31 of n1) is 1, otherwise flag4 is set to zero. The carry flag participates as a 33rd bit in the rotation.

Implementation:

opcode: RLC
0: SOURCE=DS ALU=B JMP=01Z ; ;
2: \( ( \text{ cin } \leftrightarrow 0 ) \) ALU=A+A+1 JMP=001 ; ;
3: \( ( \text{ cin } = 0 ) \) ALU=A+A JMP=001 ; ;
1: SOURCE=DHI DEST=DS JMP=10C ; ;
4: \( ( \text{ cout } = 0 ) \) ALU=0 DECODE ; ;
5: \( ( \text{ cout } \leftrightarrow 0 ) \) ALU=-1 DECODE ; ;

ROLL

Get Nth On Stack

\[( n.m \text{ n.x } \ldots n.1 \text{ m } \rightarrow \text{n.x } \ldots \text{n.1 n.m} )\]

Operation:

0x1D 5+2\(\cdot\)m cycles

Operation:
Moves the mth value from the data stack to the top of stack. <PICK> is a primitive that does not check for stack underflow, nor for negative input numbers.

Notes:
“0 <ROLL>” and “1 <ROLL>” are no-ops. “1 <ROLL>” is equivalent to “SWAP”. “2 <ROLL>” is equivalent to “ROT”.

This primitive is dangerous in an environment where stack memory is being spilled into program memory. It will not check to see if the input m is too deep in the stack, nor will it generate a stack underflow interrupt if m is big enough to cause a wrap-around back into the active stack space.

Implementation:

opcode: <ROLL>
0: SOURCE=DP DEST=DLO ; ;
\{ Roll loop \}
1: DS-FROM-DHI
\hspace{1cm} SOURCE=DS DHI[1] ALU=B JMP=01Z ; ;
2: INC[DP] DHI[0] ALU=A-1 JMP=001 ; ;

Harris Semiconductor Proprietary
( Done with loop )
3: SOURCE=DLO DEST=DP ;;
4: ;;
5: SOURCE=DHI[1] DHI[0] ALU=B
   INC[DP] DECODE ;;

**ROT**

Get Third Stack Element
( n1 n2 n3 → n2 n3 n1 )

**Encoding:**

0x43 3 cycles

**Operation:**

Rotate the third stack element n1 to the top.

**Implementation:**

opcode: ROT
1: DHI[1] DS-FROM-DHI
   SOURCE=DS ALU=B ;;
2: DEC[DP] DHI[0] DS-FROM-DHI
   SOURCE=DHI[1] ALU=B DECODE ;;
**RP!**

**Put Return Stack Pointer**

\((n1 \rightarrow)\)

**Encoding:**

\(0x0F\) \hspace{1cm} 3 cycles

**Operation:**

Store the lowest six bits of \(n\) in the hardware return stack pointer.

**Notes:**

Do not combine this opcode with a CALL or EXIT instruction.

**Implementation:**

opcode: \(\text{RP!}\)

0: \(\text{SOURCE}=\text{DHI} \hspace{0.5cm} \text{DEST}=\text{RP} \) ;;

1: ;;

2: \(\text{SOURCE}=\text{DS} \hspace{0.5cm} \text{INC}[\text{DP}] \hspace{0.5cm} \text{ALU}=\text{B} \hspace{0.5cm} \text{DECODE} ;;\)

---

**RP@**

**Get Return Stack Pointer**

\((\rightarrow n1)\)

**Encoding:**

\(0x10\) \hspace{1cm} 2 cycles

**Operation:**

Get \(n\), the six-bit return stack pointer value.

**Notes:**

Do not combine this opcode with a CALL or EXIT instruction.

**Implementation:**

opcode: \(\text{RP@}\)

0: \(\text{SOURCE}=\text{RP} \hspace{0.5cm} \text{DEST}=\text{DLO} ;;\)

1: \(\text{DEC}[\text{DP}] \hspace{0.5cm} \text{DS}-\text{FROM}-\text{DHI} \hspace{0.5cm} \text{SOURCE}=\text{DLO} \hspace{0.5cm} \text{ALU}=\text{B} \hspace{0.5cm} \text{DECODE} ;;\)
RPLIM!

Put RP Limit
( n1 → )

Encoding:
0x64 2 cycles

Operation:
Store n1 in the return stack pointer limit register. Bits 0-5 of n1 are stored in the lower limit register, while bits 16-20 are stored in the upper limit register. If the value of RP ever exceeds the upper limit register value, or is less than the lower limit register value, a stack overflow/underflow interrupt is generated.

*** Picture here ***

Implementation:
opcode: RPLIM!
0: SOURCE=DHI DEST=RP-LIMIT ;
1: SOURCE=DS INC[DP] ALU=B DECODE ;

RPLIM@

Get RP Limit Register
( → n1 )

Encoding:
0x65 2 cycles

Operation:
Fetch n1 from the return stack pointer limit register. Bits 0-5 of n1 are from the lower limit register, while bits 16-20 are from the upper limit register.

*** Picture here ***

Implementation:
opcode: RPLIM@
0: DEC[DP] DS-FROM-DHI
   SOURCE=RP-LIMIT ALU=B ;
1: DECODE ;
**RRC**

**Rotate Right Carry**

( n1 flag2 → n3 flag4 )

**Encoding:**

0xBC 4 cycles

**Operation:**

Perform a one-bit rotate right, using flag2 as the carry-in value (true or false), n1 as the input number, and n3 as the output number. If flag2 is true, a 1 is rotated into n1, otherwise a 0 is rotated into n1. flag4 is set to true if the bit rotated out of n1 (i.e. bit 0 of n1) is 1, otherwise flag4 is set to zero. In this opcode the carry flag acts as a 33rd bit for the rotation.

**Implementation:**

```
opcode: RRC
0: SOURCE=DS ALU=B DEST=DLO JMP=01Z ;
2: ( cin <>0 ) CIN=1 SR[ALU] JMP=001 ;
3: ( cin = 0 ) CIN=0 SR[ALU] JMP=001 ;
1: SOURCE=DHI DEST=DS JMP=10L ;
4: ( cout = 0 ) ALU=0 DECODE ;
5: ( cout <>0 ) ALU=-1 DECODE ;
```

**RTI**

**Return From Interrupt**

( n1 → )

**Encoding:**

0x44 4 cycles

**Operation:**

Return from interrupt. n1 is written to the config register (it is the user’s responsibility to ensure that this clears the desired stack underflow/overflow interrupt bits and clears the interrupt mask if desired). The return stack has a value addr2 that points to the restart address plus 4.

**Implementation:**

```
opcode: RTI
0: SOURCE=RS INC[RP] ADDR=BUS-4(CYCLE) ;
1: SOURCE=DHI DEST=CONFIG ;
2: LATCH-INSTRUCTION ;
3: SOURCE=DS INC[DP] ALU=B DECODE ;
```
S > D

Extend From Single To Double
(n1 → d2)

Encoding:
0x83 2 cycles

Operation:
Sign extend n1, creating d2.

Implementation:
opcode: S>D
0: JMP=01S ;;
2: DEC[DP] DS-FROM-DHI ALU=0 DECODE ;;
3: DEC[DP] DS-FROM-DHI ALU=-1 DECODE ;;

SBASE!

Set SBASE Register
(n1 → )

Encoding:
0x66 2 cycles

Operation:
Store value n1 in register SBASE.

Implementation:
opcode: SBASE!
0: SOURCE=DHI DEST=SBASE ;;
1: SOURCE=DS INC[DP] ALU=B DECODE ;;
**SBASE+!**

Add To SBASE
( n → )

**Encoding:**
0xEA 3 cycles

**Operation:**
Add the value n1 to the contents of SBASE, and place the sum back in the SBASE register.

**Implementation:**
opcode: SBASE+!
0: SOURCE=SBASE ALU=A+B ;;
1: SOURCE=DHI DEST=SBASE ;;
2: SOURCE=DS INC[DP] ALU=B DECODE ;;

---

**SBASE+_!**

Store Indexed With SBASE
( n offset2 → )

**Encoding:**
0xD6 4 cycles

**Operation:**
Store value n1 at address computed by adding offset2 to the value of SBASE.

**Implementation:**
opcode: SBASE+_!
0: SOURCE=DHI ADDR=BUS+SBASE(CYCLE) ;;
1: SOURCE=DS INC[DP] DEST=RAM-1 ;;
2: ;;
3: SOURCE=DS INC[DP] ALU=B DECODE ;;

---

Harris Semiconductor Proprietary

99

Harris Semiconductor Proprietary
**SBASE+ @**

Load Indexed With SBASE

( offset1 → n2 )

**Encoding:**

```
0xD5
```

**4 cycles**

**Operation:**

Fetch value n2 from address computed by adding offset1 to the value of SBASE.

**Implementation:**

opcode: SBASE+ @

- 0: SOURCE=DHI ADDR=BUS+SBASE(CYCLE) ;
- 1: ;
- 2: ;
- 3: SOURCE=RD ALU=B DECODE ;

---

**SBASE@**

Get From SBASE

( → n1 )

**Encoding:**

```
0x67
```

**2 cycles**

**Operation:**

Fetch the value n1 from register SBASE.

**Implementation:**

opcode: SBASE@

- 0: DEC[DP] DS-FROM-DHI
- SOURCE=SBASE ALU=B ;
- 1: DECODE ;
**SP!**

**Put Data Stack Pointer**

( n1 → )

**Encoding:**

0x0D 3 cycles

**Operation:**

Store the lowest six bits of n1 in the hardware data stack pointer.

**Implementation:**

opcode: SP!

0: SOURCE=DHI DEST=DP ;;
1: ;;
2: SOURCE=DS INC[DP] ALU=B DECODE ;;

**SP@**

**Get Data Stack Pointer**

( → n1 )

**Encoding:**

0x0E 2 cycles

**Operation:**

Get n1, the six-bit data stack pointer value.

**Implementation:**

opcode: SP@

0: SOURCE=DP DEST=DLO ;;
1: DEC[DP] DS-FROM-DHI SOURCE=DLO ALU=B DECODE ;;
**SPLIM!**

Get Data Stack Pointer Limits  
( n1 → )

Encoding:  
0x5E  2 cycles

Operation:  
Store n1 in the data stack pointer limit register. Bits 0-5 of n1 are stored in the lower limit register, while bits 16-20 are stored in the upper limit register. If the value of DP ever exceeds the upper limit register value, or is less than the lower limit register value, a stack overflow/underflow interrupt is generated.

**PICTURE**

Implementation:

opcode: SPLIM!
0: SOURCE=DHI DEST=DP-LIMIT ;
1: SOURCE=DS INC[DP] ALU=B DECODE ;

---

**SPLIM@**

Put Data Stack Pointer Limit  
( → n1 )

Encoding:  
0x5F  2 cycles

Operation:  
Fetch n1 from the data stack pointer limit register. Bits 0-5 of n1 are from the lower limit register, while bits 16-20 are from the upper limit register.

**PICTURE**

Implementation:

opcode: SPLIM@
0: DEC[DP] DS-FROM-DHI SOURCE=DP-LIMIT ALU=B ;
1: DECODE ;
STORE_DS

Streamed Write Data Stack
( n.1 n.2 ... n.count addr1 ncount2 → )

Encoding:
0xDD 3 + 3*count cycles

Operation:
Write ncount2 consecutive words, starting *backwards* at memory location addr1, from the data stack. This is a streamed data stack store with auto-decrement.

Implementation:
opcode: STORE_DS
0: SOURCE=DS DEC[DP] DEST=DBASE
   DHI[1] ALU=0 ;
   ( Count in DHI[0], offset in DHI[1] )
1: SOURCE=DHI[1]
   ADDR=BUS+DBASE(CYCLE) ;;

   ( Loop for fetching )
2: SOURCE=DS INC[DP] DEST=RAM-1
   DHI[0] ALU=A-1 ;;
4: SOURCE=DHI[1] ADDR=BUS+DBASE(CYCLE)
   JMP=010 ;;

   ( Finish up )
5: SOURCE=DS INC[DP] ALU=B DECODE ;;

STORE_RS

Streamed Write Return Stack
( addr1 ncount2 → )
( RS: n.1 n.2 ... n.count → )

Encoding:
0xDE 3 + 3*count cycles

Operation:
Write ncount2 consecutive words, starting *backwards* at memory location addr1, from the return stack. This is a streamed return stack store with auto-decrement.

Notes:
Do not combine this opcode with a CALL or EXIT instruction.

Implementation:
opcode: STORE_RS
0: SOURCE=DS INC[DP] DEST=DBASE
   DHI[1] ALU=0 ;
   ( Count in DHI[0], offset in DHI[1] )
1: SOURCE=DHI[1]
   ADDR=BUS+DBASE(CYCLE) ;;

   ( Loop for fetching )
2: SOURCE=RS INC[RP] DEST=RAM-1
   DHI[0] ALU=A-1 ;;
4: SOURCE=DHI[1] ADDR=BUS+DBASE(CYCLE)
   JMP=010 ;;

Harris Semiconductor Proprietary
SWAP

Get Second On Stack

( n1 n2 → n2 n1 )

Encoding:

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Cycles</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x45</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>0x46</td>
<td>1</td>
<td>2OPS</td>
</tr>
</tbody>
</table>

Operation:
Swap the top two stack elements.

Implementation:

opcode: SWAP
0: SOURCE=DS ALU=B DS-FROM-DHI
data
1: DECODE
data

opcode: SWAP_FAST
0: SOURCE=DS ALU=B DS-FROM-DHI
1: DECODE data
**SWAP_!**

Reversed Store
( addr1 n2 → )

Encoding:
0xE6 4 cycles

Operation:
Store n2 at address addr1.

Implementation:
opcode: SWAP_!
0: SOURCE=DS INC[DP]
   ADDR=BUS+0(CYCLE) ;;
1: SOURCE=DHI DEST=RAM-1 ;;
2: ;;
3: SOURCE=DS INC[DP] ALU=B DECODE ;;

---

**SWAP_-**

Reverse Subtract
( n1 n2 → n3 )

Encoding:
0xE7 2 cycles

Operation:
Subtract n1 from n2, giving n3.

Implementation:
opcode: SWAP_-;
0: SOURCE=DS INC[DP] ALU=A-B ;;
1: DECODE ;;
SWAP_-_!

Store With Subtracted Index
(n1 n2 addr3 →)

Encoding:
0xFD 5 cycles

Operation:
Subtract offset n2 from addr3, then store n1 at the resulting address.

Implementation:
opcode: SWAP_-_!
0: SOURCE=DS INC[DP] ALU=A-B ;
1: SOURCE=DHI
    ADDR=BUS+0(CYCLE) ;
2: SOURCE=DS INC[DP] DEST=RAM-1 ;
3: ;
4: SOURCE=DS INC[DP] ALU=B
    DECODE ;

SWAP_-_@

Load With Subtracted Index
(n1 addr2 → n3)

Encoding:
0xF7 5 cycles

Operation:
Subtract offset n1 from addr2, then fetch n3 from the resulting address.

Implementation:
opcode: SWAP_-_@
0: SOURCE=DS INC[DP] ALU=A-B ;
1: SOURCE=DHI ADDR=BUS+0(CYCLE) ;
2: ;
3: ;
4: SOURCE=RD ALU=B DECODE ;
**SWAP-_C!**

Store Character With Subtracted Index

( c1 n2 addr3 → )

**Encoding:**

0xFE 5 cycles

**Operation:**

Subtract offset n2 from addr3, then store c1 at the resulting address.

**Implementation:**

Opcode: SWAP-_C!

0: SOURCE=DS INC[DP] ALU=A-B ;;
1: SOURCE=DHI
  ADDR=BUS+0(CYCLE) ;;
2: SOURCE=DS INC[DP] DEST=RAM-C1 ;;
3: ;;
4: SOURCE=DS INC[DP] ALU=B
  DECODE ;;

**SWAP-_C@**

Load Character With Subtracted Index

( n1 addr2 → c3 )

**Encoding:**

0xF8 5 cycles

**Operation:**

Subtract offset n1 from addr2, then fetch c3 from the resulting address.

**Implementation:**

Opcode: SWAP-_C@

0: SOURCE=DS INC[DP] ALU=A-B ;;
1: SOURCE=DHI ADDR=BUS+0(CYCLE) ;;
2: RAM-C@ ;;
3: ;;
4: SOURCE=RD ALU=B DECODE ;;
**ROT_+**

Get Second On Stack, Add Immediate

( n1 n2 → n2 n3) Immediate

**Encoding:**

0xC2 / lit 2 cycles

**Operation:**

Add the compiled instruction literal field to n1, leaving n3. Note that n2 moves from the top of the stack to the second element on the stack.

**Implementation:**

```
opcode: LIT_ROT_+
0: DS-FROM-DHI
    SOURCE=DS ALU=B ;;
1: SOURCE=LIT ALU=A+B DECODE ;;
```

---

**SWAP_OVER_!**

Store (Non-Destructive)

( n1 addr2 → addr2 )

**Encoding:**

0xAA 4 cycles

**Operation:**

Store the value n1 at location addr2, without destroying addr2.

**Implementation:**

```
opcode: SWAP_OVER_!
0: SOURCE=DHI
    ADDR=BUS+0(CYCLE) ;;
1: SOURCE=DS INC[DP] DEST=RAM-1 ;;
2: ;;
3: DECODE ;;
```
TEST_AND_SET

Test And Set
(n1 addr2 → n3 tflag)
... → fflag)

Encoding:
0xCA 7 cycles

Operation:
Fetch the value n3 from word addr2, then perform a logical OR with n1, and write the result back to addr2. If any of the bits set in n1 were already set at location addr2, return n3 and a true flag. If none of the bits set in n1 were set at location addr2 before the OR operation, return a false flag. The operation is atomic, using the read/modify/write bus protocol. Useful as a synchronization instruction.

Implementation:
opcode: TEST_AND_SET
0: SOURCE=DHI ADDR=BUS+0(CYCLE) ;;
1: RAM-RMW SOURCE=DS DHI[1] ALU=B ;;
2: SOURCE=DS INC[DP] ALU=B ;;
3: SOURCE=RD DEST=DLO ALU=AorB
   CYCLE=RAM ;;
4: SOURCE=DHI DEST=RAM-1 ;;
5: SOURCE=DLO DHI[1] ALU=AandB ;;
6: SOURCE=DLO ALU=B INC[MPC] JMP=11Z ;;
next opcode
6: SOURCE=DHI DEC[DP] DEST=DS
   ALU=-1 DECODE ;;
**TEST_UNDER_MASK**

Test Under Mask

\[( n1 \land n2 \rightarrow \text{flag3} )\]

**Encoding:**

0xC7 3 cycles

**Operation:**

Test value n1 under mask n2. Returns a true flag if n1 AND n2 is non-zero, otherwise returns a false flag. In other words, a true flag is returned if any of the bits set in n1 correspond to set bits of n2.

**Implementation:**

opcode: TEST_UNDER_MASK

0: SOURCE=DS INC[DP] ALU=AandB ;
1: JMP=01Z ;;
2: ALU=-1 DECODE ;;
3: ALU=0 DECODE ;;

---

**TRUE**

Push True Flag

\[( \rightarrow -1 \)\]

**Encoding:**

0x13 2 cycles

**Operation:**

Push the value -1 on the stack. -1 is the “true” flag value for the machine.

**Implementation:**

opcode: TRUE

0: DEC[DP] DS-FROM-DHI ALU=-1 ;;
1: DECODE ;;
**TUCK**

**Put Top As Second (Non-Destructive)**

( n1 n2 → n2 n1 n2 )

**Encoding:**

0x79 2 cycles

**Operation:**
Copy the top stack element n2 under the second stack element n1.

**Implementation:**

opcode: TUCK
0: SOURCE=DS ALU=B DS-FROM-DHI ;
1: SOURCE=DS ALU=B DEC[DP] DS-FROM-DHI DECODE ;

---

**U >**

**Test For Unsigned Greater Than**

( u1 u2 → flag3 )

**Encoding:**

0x47 3 cycles

**Operation:**
Perform an unsigned comparison of u1 to u2, returning a true flag3 only if u1 > u2 is satisfied.

**Implementation:**

opcode: U>
0: SOURCE=DS INC[DP] ALU=A-B ;
( if A-B results in borrow, then true )
1: JMP=01C ;
2: ALU=0 DECODE ;
3: ALU=-1 DECODE ;
UDNORMALIZE

Unsigned Double Precision Normalize
(dmant1 exp1 → dmant2 exp2)

Encoding:  
0xDA  5 + 2*x cycles

Operation:
Normalize dmant1 so that its most significant 1 bit is in bit position 63, giving dmant2. Add one to exp1 for every right shift required for normalization, subtract one from exp1 for every left shift required for normalization, giving exp2.

Notes:
Does not check for zero input, and will infinitely loop if given one.

Implementation:
opcode: UDNORMALIZE
(shifts left until high bit set, then shifts right one bit,)
(to normalize top bit in bit 30 position.)
 DOES *NOT* check for zero input
( Exponent in DHI0, Mantissa in DHI1 )
( Perform zero check & pre-adjust exponent )
1: SOURCE=DS DEST=DLO DHI[0] ALU=A+1
  JMP=10S ;;
( Normalization loop )

2: DHI[0] ALU=A-1 JMP=10S ;;
   JMP=010 ;;

( Done, unshift )
6: SOURCE=DLO DEST=DS ;;
   DECODE ;;

Harris Semiconductor Proprietary
**UM**

*Unsigned Multiply (u1 u2 → ud3)*

**Encoding:**
0x48 36 cycles

**Operation:**
Perform a 32x32 bit unsigned multiply, giving a 64 bit unsigned product.

**Implementation:**

opcode: UM*
(Multiplier in DHI, Multiplicand in DLO)
(DHI initialized to 0)
0: SOURCE=DHI DEST=DLO ALU=0
(Initial bit shifts)
1: SR[DLO] ; ;
2: SR[DLO] JMP=10L ; ;

(Every pair of microinstructions does one bit)
4: MULTIPLY-STEP ALU=A+0
   SR[ALU] SR[DLO] JMP=11L ; ;
5: MULTIPLY-STEP ALU=A+B
   SR[ALU] SR[DLO] JMP=11L ; ;
6: MULTIPLY-STEP ALU=A+0
   SR[ALU] SR[DLO] JMP=00L INC[MPC] ; ;
7: MULTIPLY-STEP ALU=A+B
   SR[ALU] SR[DLO] JMP=00L INC[MPC] ; ;
next opcode: (bits 2-5)
0: MULTIPLY-STEP ALU=A+0
   SR[ALU] SR[DLO] JMP=01L ; ;
1: MULTIPLY-STEP ALU=A+B
   SR[ALU] SR[DLO] JMP=01L ; ;
2: MULTIPLY-STEP ALU=A+0
   SR[ALU] SR[DLO] JMP=10L ; ;
3: MULTIPLY-STEP ALU=A+B
   SR[ALU] SR[DLO] JMP=10L ; ;
4: MULTIPLY-STEP ALU=A+0
   SR[ALU] SR[DLO] JMP=11L ; ;
5: MULTIPLY-STEP ALU=A+B
   SR[ALU] SR[DLO] JMP=11L ; ;
6: MULTIPLY-STEP ALU=A+0
   SR[ALU] SR[DLO] JMP=00L INC[MPC] ; ;
7: MULTIPLY-STEP ALU=A+B
   SR[ALU] SR[DLO] JMP=00L INC[MPC] ; ;
next opcode: (bits 6-9)
0: MULTIPLY-STEP ALU=A+0
   SR[ALU] SR[DLO] JMP=01L ; ;
1: MULTIPLY-STEP ALU=A+B
   SR[ALU] SR[DLO] JMP=01L ; ;
2: MULTIPLY-STEP ALU=A+0
   SR[ALU] SR[DLO] JMP=10L ; ;
3: MULTIPLY-STEP ALU=A+B
   SR[ALU] SR[DLO] JMP=10L ; ;
4: MULTIPLY-STEP ALU=A+0
   SR[ALU] SR[DLO] JMP=11L ; ;
5: MULTIPLY-STEP ALU=A+B
   SR[ALU] SR[DLO] JMP=11L ; ;
6: MULTIPLY-STEP ALU=A+0
   SR[ALU] SR[DLO] JMP=00L INC[MPC] ; ;
7: MULTIPLY-STEP ALU=A+B
   SR[ALU] SR[DLO] JMP=00L INC[MPC] ; ;
next opcode: (bits 10-13)
0: MULTIPLY-STEP ALU=A+0
   SR[ALU] SR[DLO] JMP=01L ; ;
1: MULTIPLY-STEP ALU=A+B
   SR[ALU] SR[DLO] JMP=01L ; ;
2: MULTIPLY-STEP ALU=A+0
   SR[ALU] SR[DLO] JMP=10L ; ;
3: MULTIPLY-STEP ALU=A+B
   SR[ALU] SR[DLO] JMP=10L ; ;
SR[ALU] SR[DLO] JMP=10L 
4: MULTIPLY-STEP ALU=A+O
   SR[ALU] SR[DLO] JMP=11L 
5: MULTIPLY-STEP ALU=A+B
   SR[ALU] SR[DLO] JMP=11L 
6: MULTIPLY-STEP ALU=A+O
   SR[ALU] SR[DLO] JMP=00L INC[MPC] 
7: MULTIPLY-STEP ALU=A+B
   SR[ALU] SR[DLO] JMP=00L INC[MPC] 
next opcode: (bits 14-17)
0: MULTIPLY-STEP ALU=A+O
   SR[ALU] SR[DLO] JMP=01L 
1: MULTIPLY-STEP ALU=A+B
   SR[ALU] SR[DLO] JMP=01L 
2: MULTIPLY-STEP ALU=A+O
   SR[ALU] SR[DLO] JMP=10L 
3: MULTIPLY-STEP ALU=A+B
   SR[ALU] SR[DLO] JMP=10L 
4: MULTIPLY-STEP ALU=A+O
   SR[ALU] SR[DLO] JMP=11L 
5: MULTIPLY-STEP ALU=A+B
   SR[ALU] SR[DLO] JMP=11L 
6: MULTIPLY-STEP ALU=A+O
   SR[ALU] SR[DLO] JMP=00L INC[MPC] 
7: MULTIPLY-STEP ALU=A+B
   SR[ALU] SR[DLO] JMP=00L INC[MPC] 
next opcode: (bits 22-25)
0: MULTIPLY-STEP ALU=A+O
   SR[ALU] SR[DLO] JMP=01L 
1: MULTIPLY-STEP ALU=A+B
   SR[ALU] SR[DLO] JMP=01L 
2: MULTIPLY-STEP ALU=A+O
   SR[ALU] SR[DLO] JMP=10L 
3: MULTIPLY-STEP ALU=A+B
   SR[ALU] SR[DLO] JMP=10L 
4: MULTIPLY-STEP ALU=A+O
   SR[ALU] SR[DLO] JMP=11L 
5: MULTIPLY-STEP ALU=A+B
   SR[ALU] SR[DLO] JMP=11L 
6: MULTIPLY-STEP ALU=A+O
SR[ALU] SR[DLO] JMP=00L INC[MPC] ;;  
0: MULTIPLY-STEP ALU=A+O  
  SR[ALU] SR[DLO] JMP=01L ;;  
1: MULTIPLY-STEP ALU=A+B  
  SR[ALU] SR[DLO] JMP=01L ;;  
2: MULTIPLY-STEP ALU=A+O  
  SR[ALU] SR[DLO] JMP=100 ;;  
3: MULTIPLY-STEP ALU=A+B  
  SR[ALU] SR[DLO] JMP=100 ;;  
4:  
    SOURCE=DLO  
    DEST=DS  
    DECODE ;;  

UM/MOD  

Unsigned Division  
(udivdnd undivsr → unrem unquot)

Encoding:
0x51 37 cycles

Operation:
Perform a 64/32 bit unsigned division, giving a 32 bit unsigned remainder and a 32 bit unsigned quotient.

Implementation:
opcode: UM/MOD  
(DHI:DLO is dividend, DS is divisor)
1: DS-FROM-DHI DHI[0]  
    SOURCE=DS DEST=DLO ;;  
(Initial subtraction)

(Iterated step-divisions bits 0-4)
7: DIVIDE SL[DLO] SL[ALU]  
    INC[MPC] JMP=000 ;;  

next opcode (bits 5-12)
0: DIVIDE SL[DLO] SL[ALU] ;;  
**UNNORMALIZE**

**Unsigned Normalize**

(mant1 exp1 → mant2 exp2)

**Encoding:**

0xD9 3/5+2*N cycles

**Operation:**

Normalize mant1 so that its most significant 1 bit is in bit position 30, giving mant2. Add one to expl for every right shift required for normalization, subtract one from expl for every left shift required for normalization, giving exp2. Takes 3 clock cycles if the input is zero.

**Implementation:**

opcode: **UNNORMALIZE**

(shifts left until high bit set, then shifts right one bit to normalize top bit in bit 30 position.)

(Exponent in DHI0, Mantissa in DHI1)

0: SOURCE=DS DHI[1] ALU=B ;;

(Perform zero check & pre-adjust exponent)

1: DHI[1] JMP=01Z ;;

2: DHI[0] ALU=A+1 JMP=10S ;;

(Normalization loop)

7: DHI[0] ALU=A-1 JMP=10S ;;

4: DHI[1] ALU=A+A JMP=111 ;;

(Done, unshift)
WAIT

Wait For Interrupt

(→)

Encoding:

0x68  2 or more cycles

Operation:

Halt until an unmasked interrupt is recognized, then continue.

Implementation:

opcode: WAIT
0: JMP=00P
1: DECODE ;;

5: DHI[1] CIN=0 SR[ALU] ;;
6: SOURCE=DHI[1] DEST=DS DECODE ;;

( zero input – force clean zero )
3: ALU=0 DECODE ;;
**WFILL**

**Fill Memory**

( addr1 count2 n3 → )

**Encoding:**

0x8D  \( 4 + 3 \times \text{count cycles} \)

**Operation:**

Fill count2 words of memory with value n3, starting at address addr1 and counting up.

**Notes:**

Destroys value in the DBASE register.

**Implementation:**

opcode: WFILL

( DHI[1] = OFFSET & COUNT*4 )

0: SOURCE=0 DEST=DLO DHI[1] ALU=-1 ;;
1: SOURCE=DS INC[DP]
7: SOURCE=DS INC[DP] DEST=DBASE
   DHI[1] ALU=A+A JMP=010 ;;

( Store words )

2: SOURCE=DHI[1] ADDR=BUS+DBASE (CYCLE)
   JMP=100 ;;
4: SOURCE=DHI[0] DEST=RAM-1 DHI[1] ;;
   JMP=01Z ;;
3: SOURCE=DS INC[DP] ALU=B DECODE ;;

---

**XOR**

**Exclusive Or**

( n1 n2 → n3 )

**Encoding:**

0x56  2 cycles
0x57  1 cycle (2OPS format)

**Operation:**

Perform a bitwise logical eXclusive OR function on n1 and n2, returning n3.

**Implementation:**

opcode: XOR

0: SOURCE=DS INC[DP] ALU=A\text{\_or\_}B ;;
1: DECODE ;;

opcode: XOR\_FAST

0: SOURCE=DS INC[DP]
   ALU=A\text{\_or\_}B DECODE ;;