Chapter 2
HCS12 Assembly Language
ECE 3120

Dr. Mohamed Mahmoud

http://iweb.tntech.edu/mmahmoud/
mahmoud@tntech.edu
Outline

2.1 Assembly language program structure
2.2 Data transfer instructions
2.3 Arithmetic instructions
2.4 Branch and loop instructions
2.5 Shift and rotate instructions
2.6 Boolean logic instructions
2.7 Bit test and manipulate instructions
2.8 Stack
2.9 Subroutines
# 1- Assembler directives

- Commands to the assembler
- Are not executed by the microcontroller – are not converted to machine codes
- Define program constants and reserve space for variable

## Sample Program

<table>
<thead>
<tr>
<th>Label</th>
<th>Opcode</th>
<th>Operand</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>main:</td>
<td>LDA A</td>
<td>$800</td>
<td>; A = m[$800]</td>
</tr>
<tr>
<td></td>
<td>ADD A</td>
<td>$801</td>
<td>; A = A + m[$801]</td>
</tr>
<tr>
<td></td>
<td>ADD A</td>
<td>$802</td>
<td>; A = A + m[$802]</td>
</tr>
<tr>
<td></td>
<td>STA A</td>
<td>$805</td>
<td>; m[$805] = A</td>
</tr>
</tbody>
</table>

Directive: Tells loader where to put program

Directive: Tells assembler where program finished

END
1. Org (origin)
- Tells the assembler where to place the next instruction/data in memory
- Example:

```
org $1000
ldab #$FF ;this instruction will be stored in memory starting from location $1000.
```

2. dc.b (define constant byte), db (define byte), fcb (form constant byte)
- Define the value of a byte or bytes that will be placed at a given location.
- Example:

```
org $800
array dc.b $11,$22,$33,$44
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>800</td>
<td>$11</td>
</tr>
<tr>
<td>801</td>
<td>$22</td>
</tr>
<tr>
<td>802</td>
<td>$33</td>
</tr>
<tr>
<td>803</td>
<td>$44</td>
</tr>
</tbody>
</table>
3. dc.w (define constant word), dw (define word), fdb (form double bytes)

- Define the value of a word or words that will be placed at a given location.

- For example:

  org $800
  array dc.w $AC11,$F122,$33,$F44

4. fcc (form constant character)

- Tells the assembler to store a string of characters (a message) in memory.
- The last character must be the same as the first character.
- The first and last characters are used as delimiters.
- Each character is represented by its ASCII code.
- For example: `msg fcc “Please enter your name:”`

```
<table>
<thead>
<tr>
<th>1000</th>
<th>$64</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>$65</td>
</tr>
<tr>
<td>1002</td>
<td>$66</td>
</tr>
</tbody>
</table>
```

- Assembler will convert to Ascii

5. fill

- Example:

```asm
Org $800
fill $20, 40
```

; fill 40 bytes with $20 starting from the memory location $800.

6- ds (define storage), rmb (reserve memory byte), ds.b (define storage bytes)

- Reserves a number of bytes for later use.

- Example: `buffer ds 100`

; reserves 100 bytes starting from the location represented by buffer - none of these locations is initialized
<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>Null</td>
<td>32</td>
<td>20</td>
<td>Space</td>
<td>64</td>
<td>40</td>
<td>@</td>
<td>96</td>
<td>60</td>
<td>'</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>Start of heading</td>
<td>33</td>
<td>21</td>
<td>!</td>
<td>65</td>
<td>41</td>
<td>A</td>
<td>97</td>
<td>61</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>Start of text</td>
<td>34</td>
<td>22</td>
<td>&quot;</td>
<td>66</td>
<td>42</td>
<td>B</td>
<td>98</td>
<td>62</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>End of text</td>
<td>35</td>
<td>23</td>
<td>#</td>
<td>67</td>
<td>43</td>
<td>C</td>
<td>99</td>
<td>63</td>
<td>c</td>
</tr>
<tr>
<td>4</td>
<td>04</td>
<td>End of transmit</td>
<td>36</td>
<td>24</td>
<td>$</td>
<td>68</td>
<td>44</td>
<td>D</td>
<td>100</td>
<td>64</td>
<td>d</td>
</tr>
<tr>
<td>5</td>
<td>05</td>
<td>Enquiry</td>
<td>37</td>
<td>25</td>
<td>%</td>
<td>69</td>
<td>45</td>
<td>E</td>
<td>101</td>
<td>65</td>
<td>e</td>
</tr>
<tr>
<td>6</td>
<td>06</td>
<td>Acknowledge</td>
<td>38</td>
<td>26</td>
<td>&amp;</td>
<td>70</td>
<td>46</td>
<td>F</td>
<td>102</td>
<td>66</td>
<td>f</td>
</tr>
<tr>
<td>7</td>
<td>07</td>
<td>Audible bell</td>
<td>39</td>
<td>27</td>
<td>'</td>
<td>71</td>
<td>47</td>
<td>G</td>
<td>103</td>
<td>67</td>
<td>g</td>
</tr>
<tr>
<td>8</td>
<td>08</td>
<td>Backspace</td>
<td>40</td>
<td>28</td>
<td>(</td>
<td>72</td>
<td>48</td>
<td>H</td>
<td>104</td>
<td>68</td>
<td>h</td>
</tr>
<tr>
<td>9</td>
<td>09</td>
<td>Horizontal tab</td>
<td>41</td>
<td>29</td>
<td>)</td>
<td>73</td>
<td>49</td>
<td>I</td>
<td>105</td>
<td>69</td>
<td>i</td>
</tr>
<tr>
<td>10</td>
<td>0A</td>
<td>Line feed</td>
<td>42</td>
<td>2A</td>
<td>*</td>
<td>74</td>
<td>4A</td>
<td>J</td>
<td>106</td>
<td>6A</td>
<td>j</td>
</tr>
<tr>
<td>11</td>
<td>0B</td>
<td>Vertical tab</td>
<td>43</td>
<td>2B</td>
<td>+</td>
<td>75</td>
<td>4B</td>
<td>K</td>
<td>107</td>
<td>6B</td>
<td>k</td>
</tr>
<tr>
<td>12</td>
<td>0C</td>
<td>Form feed</td>
<td>44</td>
<td>2C</td>
<td>,</td>
<td>76</td>
<td>4C</td>
<td>L</td>
<td>108</td>
<td>6C</td>
<td>l</td>
</tr>
<tr>
<td>13</td>
<td>0D</td>
<td>Carriage return</td>
<td>45</td>
<td>2D</td>
<td>-</td>
<td>77</td>
<td>4D</td>
<td>M</td>
<td>109</td>
<td>6D</td>
<td>m</td>
</tr>
<tr>
<td>14</td>
<td>0E</td>
<td>Shift out</td>
<td>46</td>
<td>2E</td>
<td>.</td>
<td>78</td>
<td>4E</td>
<td>N</td>
<td>110</td>
<td>6E</td>
<td>n</td>
</tr>
<tr>
<td>15</td>
<td>0F</td>
<td>Shift in</td>
<td>47</td>
<td>2F</td>
<td>/</td>
<td>79</td>
<td>4F</td>
<td>O</td>
<td>111</td>
<td>6F</td>
<td>o</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>Data link escape</td>
<td>48</td>
<td>30</td>
<td>0</td>
<td>80</td>
<td>50</td>
<td>P</td>
<td>112</td>
<td>70</td>
<td>p</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>Device control 1</td>
<td>49</td>
<td>31</td>
<td>1</td>
<td>81</td>
<td>51</td>
<td>Q</td>
<td>113</td>
<td>71</td>
<td>q</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>Device control 2</td>
<td>50</td>
<td>32</td>
<td>2</td>
<td>82</td>
<td>52</td>
<td>R</td>
<td>114</td>
<td>72</td>
<td>r</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>Device control 3</td>
<td>51</td>
<td>33</td>
<td>3</td>
<td>83</td>
<td>53</td>
<td>S</td>
<td>115</td>
<td>73</td>
<td>s</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>Device control 4</td>
<td>52</td>
<td>34</td>
<td>4</td>
<td>84</td>
<td>54</td>
<td>T</td>
<td>116</td>
<td>74</td>
<td>t</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>Neg. acknowledge</td>
<td>53</td>
<td>35</td>
<td>5</td>
<td>85</td>
<td>55</td>
<td>U</td>
<td>117</td>
<td>75</td>
<td>u</td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>Synchronous idle</td>
<td>54</td>
<td>36</td>
<td>6</td>
<td>86</td>
<td>56</td>
<td>V</td>
<td>118</td>
<td>76</td>
<td>v</td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>End trans. block</td>
<td>55</td>
<td>37</td>
<td>7</td>
<td>87</td>
<td>57</td>
<td>W</td>
<td>119</td>
<td>77</td>
<td>w</td>
</tr>
<tr>
<td>24</td>
<td>18</td>
<td>Cancel</td>
<td>56</td>
<td>38</td>
<td>8</td>
<td>88</td>
<td>58</td>
<td>X</td>
<td>120</td>
<td>78</td>
<td>x</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>End of medium</td>
<td>57</td>
<td>39</td>
<td>9</td>
<td>89</td>
<td>59</td>
<td>Y</td>
<td>121</td>
<td>79</td>
<td>y</td>
</tr>
<tr>
<td>26</td>
<td>1A</td>
<td>Substitution</td>
<td>58</td>
<td>3A</td>
<td>;</td>
<td>90</td>
<td>5A</td>
<td>Z</td>
<td>122</td>
<td>7A</td>
<td>z</td>
</tr>
<tr>
<td>27</td>
<td>1B</td>
<td>Escape</td>
<td>59</td>
<td>3B</td>
<td>;</td>
<td>91</td>
<td>5B</td>
<td>[</td>
<td>123</td>
<td>7B</td>
<td>(</td>
</tr>
<tr>
<td>28</td>
<td>1C</td>
<td>File separator</td>
<td>60</td>
<td>3C</td>
<td>&lt;</td>
<td>92</td>
<td>5C</td>
<td>\</td>
<td>124</td>
<td>7C</td>
<td>I</td>
</tr>
<tr>
<td>29</td>
<td>1D</td>
<td>Group separator</td>
<td>61</td>
<td>3D</td>
<td>=</td>
<td>93</td>
<td>5D</td>
<td>]</td>
<td>125</td>
<td>7D</td>
<td>)</td>
</tr>
<tr>
<td>30</td>
<td>1E</td>
<td>Record separator</td>
<td>62</td>
<td>3E</td>
<td>&gt;</td>
<td>94</td>
<td>5E</td>
<td>^</td>
<td>126</td>
<td>7E</td>
<td>~</td>
</tr>
<tr>
<td>31</td>
<td>1F</td>
<td>Unit separator</td>
<td>63</td>
<td>3F</td>
<td>?</td>
<td>95</td>
<td>5F</td>
<td>_</td>
<td>127</td>
<td>7F</td>
<td>_</td>
</tr>
</tbody>
</table>
7. **ds.w (define storage word), rmw (reserve memory word)**

- Reserve a number of words

```assembly
Dbuf ds.w 20 ;Reserves 20 words (or 40 bytes) starting from the current location counter.
```

8. **equ (equate)**

- Assigns a value to a label.
- Makes programs more readable.
- **Examples:**

```assembly
motor_speed equ 50
```

The assembler will replace motor_speed with the value 50 in the whole program
Example 1: Array of bytes

```assembly
org $800
a1 db $11, $22, $33, $44
a2 dc.b $01
   dc.b $02
da3 rmb 2
```

```
a1 = $800
a2 = $804
a3 = $807
```

Example 2: Array of words

```assembly
org $800
a1 dw $11, $22, $33, $44
a2 dc.w $01
da3 rmw 2
```

```
a1 = 00
a2 = 11
a3 = 22
```

```
Memory Map

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$800</td>
<td>00</td>
</tr>
<tr>
<td>$801</td>
<td>11</td>
</tr>
<tr>
<td>$802</td>
<td>22</td>
</tr>
<tr>
<td>$803</td>
<td>33</td>
</tr>
<tr>
<td>$804</td>
<td>44</td>
</tr>
<tr>
<td>$805</td>
<td>01</td>
</tr>
<tr>
<td>$806</td>
<td>02</td>
</tr>
<tr>
<td>$807</td>
<td>03</td>
</tr>
<tr>
<td>$808</td>
<td>?</td>
</tr>
<tr>
<td>$809</td>
<td>?</td>
</tr>
<tr>
<td>$80A</td>
<td>?</td>
</tr>
<tr>
<td>$80B</td>
<td>?</td>
</tr>
<tr>
<td>$80C</td>
<td>?</td>
</tr>
<tr>
<td>$80D</td>
<td>?</td>
</tr>
<tr>
<td>$80E</td>
<td>?</td>
</tr>
</tbody>
</table>
```
Sample Program

```
ORG $4000

main:
  LDAA $800
  ADDA $801
  ADDA $802
  STAA $805

; A = m[$800]
; A = A + m[$801]
; A = A + m[$802]
; m[$805] = A

END
```

**Label field**

- Labels are used to identify memory locations in the programs and data areas.
- Optional
- Must starts with a letter (A-Z or a-z) and can be followed by letters, digits, or special symbols (_ or .)
Label field

- Can start from any column if ended with “:”
- Must start from column 1 if it is not ended with “:”
- Example:

```
Begin: ldaa #10 ; Begin is a valid label
Print jsr hexout ; jump to hexout subroutine, Print is a valid label
jmp begin ; jump to begin label, do not put “:” when referring to a label
```
### Sample Program

```assembly
ORG $4000

<table>
<thead>
<tr>
<th>Label</th>
<th>Opcode</th>
<th>Operand</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>main:</td>
<td>LDAA</td>
<td>$800</td>
<td>; A = m[$800]</td>
</tr>
<tr>
<td></td>
<td>ADDA</td>
<td>$801</td>
<td>; A = A + m[$801]</td>
</tr>
<tr>
<td></td>
<td>ADDA</td>
<td>$802</td>
<td>; A = A + m[$802]</td>
</tr>
<tr>
<td></td>
<td>STAA</td>
<td>$805</td>
<td>; m[$805] = A</td>
</tr>
</tbody>
</table>

END
```

### Comment field

- **Optional**
- **Explain the function of a single or a group of instructions**
- **For programmer – not for assembler or processor.**
- **Ignored by assembler and are not converted to machine code.**
- **Can improve a program readability - very important in assembly**
- Any line starts with an * or ; is a comment
- Separated from the operand for at least one space
### Sample Program

```plaintext
ORG $4000

<table>
<thead>
<tr>
<th>Label</th>
<th>Opcode</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>main:</td>
<td>LDAA</td>
<td>$800</td>
</tr>
<tr>
<td></td>
<td>ADDA</td>
<td>$801</td>
</tr>
<tr>
<td></td>
<td>ADDA</td>
<td>$802</td>
</tr>
<tr>
<td></td>
<td>STAA</td>
<td>$805</td>
</tr>
</tbody>
</table>

; A = m[$800]
; A = A + m[$801]
; A = A + m[$802]
; m[$805] = A

END
```

### Instructions

- Instruct the processor to do a sequence of operations
- Converted to machine code
- Operands follow the opcode and is separated from the opcode by at least one space
- Operands are separated by commas (if there is more than one operand)
- Opcode is the operation and separated from the label by at least one space
- Assembler instructions or directives are not case sensitive
- **Must not** start at column 1
- Addressing modes specify the operand to be operated on.

- The addressing mode may specify an immediate value, a register, or a memory location to be used as an operand.

The Basic Addressing Modes

1. Inherent
2. Immediate
3. Direct
4. Extended
5. Relative
6. Indexed
7. Indexed-Indirect
1- Inherent Mode

- Either do not need operands or all operands are CPU registers. The instruction has only an opcode.

- Operands can be detected from the opcode.

- Examples:

  INX  ; Increment X
  CLRA  ; clear A
  ABA  ; A = A + B

2- Immediate Mode

- Operands’ values are included in the instruction. The values are fetched from the machine code in the memory.

- An immediate value is preceded by # character
- Example:

```
LDAA #$55          ; A ← $55
LDX #$1000         ; X ← $1000
movw #$10, $100    ; m[$100] ← $00 and m[$101] ← $10
                 ; Store the hex values $00 and $10 in
                 ; the memory locations at $100 and $101
```

3- Direct Mode

- The operand is a memory location in the range of $00 to $FF.

Examples:

```
LDAA $20          ; A ← [$20] A = the value at memory location
                 ; $0020
LDAB $40          ; B ← [$40]
LDX $20           ; X_H ← [$20] X_L ← [$21]
```
What is the difference between:

```
ldaa $45    ; A = the content of memory location 45
ldaa #$45 ; A = 45
```

4- Extended Mode

- Same as Direct mode but with using 16-bit memory address.
- Used to access any location in the 64 kB memory from 0000 to FFFF

```
LDAA $4000 ; A ← [$4000]
LDX $FE60 ; X ← [$FE60]:[$FE61] places the first byte in the high-order byte
```
5- Relative Mode

- Used only by branch instructions.
- If a branch is taken \(\rightarrow\) PC = PC + offset
- A short branch instruction: offset is a signed 8-bit \(\rightarrow\) can specify a range of \(-128 \sim +127\).
- A long branch instruction: offset is a signed 16-bit \(\rightarrow\) can specify a range of \(-32768 \sim +32767\).
- A programmer uses a label to specify the branch target and the assembler will figure out the offset and add it to the machine instruction.

- Example:

  ```
  minus  ... 
  ... 
  bmi    minus  ; go to minus if N flag in CCR register= 1
  ```
6- Indexed Mode

- The operand is a memory location.

6.1 Indexed with constant offset

- The address of the operand = a base register + a constant offset
- The base register can be (X, Y, PC, or SP)

Examples

\[
\text{ldaa 4,X ; } A \leftarrow [4+[X]]
\]

Load A with the content of the memory location at address X+4

\[
\text{ldaa 0,X ; } A = [0 + [X]]
\]

\[
\text{stab -8,X ; Store B at memory location X – 8}
\]

\[
\text{ldd 100,Y ; A = [100+[Y]], B = [101+[Y]]}
\]
6.2 Indexed with an accumulator register offset
- The operand address = accumulator + base index register.
- The accumulator can be A, B, or D
- The base index register can be X, Y, SP, or PC.
- Examples:
  \[
  \text{ldda B,X} \quad ; \text{load A with the content of memory location X+} B \\
  \text{ldy D,X} \quad ; \text{Y = memory locations D + X and D + X +1}
  \]

6.3 Indexed with auto pre-/post-increment/decrement of index register
- The base register r can be X, Y, or SP. (No PC)
- New r = old r + (or -) n
- n is the amount of decrement or increment. It is in the ranges -8 thru -1 or 1 thru 8.
- Post-decrement/increment: Operand address = old r
- Pre-decrement/increment: Operand address = new r
### Examples: Assume $X$ has the value $\$1000$

<table>
<thead>
<tr>
<th>Pre-decrement (n,-r)</th>
<th>Post-decrement (n,r-)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{ldaa 2,-X}</td>
<td>\texttt{ldaa 2,X-}</td>
</tr>
<tr>
<td>$X = $1000 - 2 = $FFE$</td>
<td>$A = [$1000]$</td>
</tr>
<tr>
<td>$A = [$FFE]$</td>
<td>$X = 1000 - 2 = $FFE$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pre-increment (n,+r)</th>
<th>Post-increment (n,r+)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{ldaa 2,+X}</td>
<td>\texttt{ldaa 2,X+}</td>
</tr>
<tr>
<td>$X = 1000 + 2 = $1002$</td>
<td>$A = [$1000]$</td>
</tr>
<tr>
<td>$A = [$1002]$</td>
<td>$X = 1000 + 2 = $1002$</td>
</tr>
</tbody>
</table>

Can be used to read an array.

\texttt{ldaa 1,X+ ; A = [1000]}
\texttt{ldaa 1,X+ ; A = [1001]}
\texttt{ldaa 1,X+ ; A = [1002]}
### 6.4 Indexed-Indirect

- The sum of the base register and offset does **not** point at the operand address but to the address of memory location where the operand address can be found (address of address)

### 6.4.1 16-bit Offset Indirect Indexed Addressing

- Syntax of the addressing mode is \([n,r]\)
- \(n\) is 16 bit offset
- \(r\) is base register X, Y, SP, PC
- The operand address = the content of the memory location at \(n + r\)
- The square brackets distinguish this addressing mode from the 16-bit constant offset indexing.

**ldaa [10, X]**

- If \(X = 1000\), then \(X + 10 = 100A\)
- It reads 16 bits in the locations $100A$ and next one $100B$. Assume this value is $2000$
- \(A\) = the value stored in location $2000$
Example

\[
\text{ldaa \#}$10 \rightarrow A = 10
\]

\[
\text{ldaa 10,X} \rightarrow A = [X+10] = 20
\]

\[
\text{ldy 10,X} \rightarrow Y = [X+10]: [X+11] = 2000
\]

\[
\text{ldaa [10,X]} \rightarrow A = [[X+10]] = [2000] = \$F0
\]

\[
\text{ldy [10,X]} \rightarrow y = [[X+10]]: [[X+11]] = [2000]: [2001] = \$F03A
\]
6.4.2 Accumulator D Indirect Indexed Addressing

- The syntax of this addressing mode is \([D,r]\)
- \(r\) is base register, \(X, Y, SP,\) or \(PC\)
- The operand address is stored in the memory location \(D + r\)

The possible addressing modes of each instruction are given in the instruction set file.

<table>
<thead>
<tr>
<th>Source Form</th>
<th>Operation</th>
<th>Addr. Mode</th>
<th>Machine Coding (hex)</th>
<th>Access Detail</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDAA #opr8i</td>
<td>(M) (\Rightarrow) A</td>
<td>IMM</td>
<td>86 ii</td>
<td>p</td>
</tr>
<tr>
<td>LDAA opr8a</td>
<td>Load Accumulator A</td>
<td>IMM</td>
<td>96 dd</td>
<td>rPf</td>
</tr>
<tr>
<td>LDAA opr16a</td>
<td></td>
<td>IMM</td>
<td>B6 hh 1l</td>
<td>rPO</td>
</tr>
<tr>
<td>LDAA oprx0, yxsp</td>
<td></td>
<td>IMM</td>
<td>A6 xb</td>
<td>rPf</td>
</tr>
<tr>
<td>LDAA oprx9, yxsp</td>
<td></td>
<td>IMM</td>
<td>A6 xb ff</td>
<td>rPO</td>
</tr>
<tr>
<td>LDAA oprx16, yxsp</td>
<td></td>
<td>IMM</td>
<td>A6 xb ee ff</td>
<td>frPP</td>
</tr>
<tr>
<td>LDAA [D, yxsp]</td>
<td></td>
<td>IMM</td>
<td>A6 xb</td>
<td>fIfPf</td>
</tr>
<tr>
<td>LDAA [opr16, yxsp]</td>
<td></td>
<td>IMM</td>
<td>A6 xb ee ff</td>
<td>fIPrPf</td>
</tr>
</tbody>
</table>
Summary of important addressing modes

<table>
<thead>
<tr>
<th>Name</th>
<th>Example</th>
<th>Machine code</th>
<th>Effective Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>INH Inherent</td>
<td>ABA</td>
<td>87</td>
<td>None</td>
</tr>
<tr>
<td>IMM Immediate</td>
<td>LDAA #$35</td>
<td>86 35</td>
<td>PC + 1</td>
</tr>
<tr>
<td>DIR Direct</td>
<td>LDAA $35</td>
<td>96 35</td>
<td>0x0035</td>
</tr>
<tr>
<td>EXT Extended</td>
<td>LDAA $0935</td>
<td>B6 09 35</td>
<td>0x0935</td>
</tr>
<tr>
<td>IDX Indexed</td>
<td>LDAA 3,X</td>
<td>A6 03</td>
<td>X + 3</td>
</tr>
<tr>
<td>IDX Indexed</td>
<td>LDAA 3,X+</td>
<td>A6 32</td>
<td>X (X+3 -&gt; X)</td>
</tr>
<tr>
<td>IDX Indexed</td>
<td>LDAA 3,+X</td>
<td>A6 22</td>
<td>X+3 (X+3 -&gt; X)</td>
</tr>
</tbody>
</table>
1. **Problem definition**: Identify what should be done

2. Identify the inputs and outputs

3. **Develop the algorithm** (or a flowchart):
   - The overall plan for solving the problem at hand.
   - A sequence of operations that transform inputs to output expressed in the following format (pseudo code):

   ```
   Step 1: read a value and store in variable X
   ... ... ...
   Step i: N = X + 5
   ... ... ...
   ```

4. **Programming**: Convert the algorithm into programs.

5. **Program Testing**:
   - Testing for anomalies.
   - Test for the max. and min. values of inputs
   - Enter values that can test all branches
2.1 Assembly language program structure

2.2 Data transfer instructions

2.3 Arithmetic instructions

2.4 Branch and loop instructions

2.5 Shift and rotate instructions

2.6 Boolean logic instructions

2.7 Bit test and manipulate instructions

2.8 Stack

2.9 Subroutines
**Load**: Copies the contents of a memory location (or immediate value) into a register. The memory location does not change but the register changes.

**Store**: Copies the contents of a register into a memory location. The register does not change but the memory location changes.

**Move**: Copies the content of a memory location into other memory location.

**Transfer**: Copies the content of a register into another register.
What to keep in mind to learn how the instructions work

- How does the instruction affect registers and/or memory?
- How does the instruction affect the flags?
- Is it clear where the input numbers are and where the results (destination) should go? 8 or 16 bits?
- The instruction is for signed or unsigned numbers?
- What kind of addressing modes are available?

1- The LOAD and STORE Instructions

- The LOAD instruction copies the content of a memory location or an immediate value to an accumulator or a CPU register.

- Memory contents are not changed.

- STORE instructions copies a CPU register into a memory location. The register contents are not changed.

- N and Z flags of the CCR register are automatically updated, the V flag is cleared, and C does not change.
# Table 1.4 Load and store instructions

## Load Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldaa &lt;opr&gt;</td>
<td>Load A</td>
<td>A ← [opr]</td>
</tr>
<tr>
<td>ldab &lt;opr&gt;</td>
<td>Load B</td>
<td>B ← [opr]</td>
</tr>
<tr>
<td>ldd &lt;opr&gt;</td>
<td>Load D</td>
<td>A:B ← [opr]:[opr+1]</td>
</tr>
<tr>
<td>lds &lt;opr&gt;</td>
<td>Load SP</td>
<td>SP ← [opr]:[opr+1]</td>
</tr>
<tr>
<td>idx &lt;opr&gt;</td>
<td>Load index register X</td>
<td>X ← [opr]:[opr+1]</td>
</tr>
<tr>
<td>idy &lt;opr&gt;</td>
<td>Load index register Y</td>
<td>Y ← [opr]:[opr+1]</td>
</tr>
<tr>
<td>leas &lt;opr&gt;</td>
<td>Load effective address into SP</td>
<td>SP ← effective address</td>
</tr>
<tr>
<td>leax &lt;opr&gt;</td>
<td>Load effective address into X</td>
<td>X ← effective address</td>
</tr>
<tr>
<td>leay &lt;opr&gt;</td>
<td>Load effective address into Y</td>
<td>Y ← effective address</td>
</tr>
</tbody>
</table>

## Store Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>staa &lt;opr&gt;</td>
<td>Store A in a memory location</td>
<td>m[opr] ← [A]</td>
</tr>
<tr>
<td>stab &lt;opr&gt;</td>
<td>Store B in a memory location</td>
<td>m[opr] ← [B]</td>
</tr>
<tr>
<td>std &lt;opr&gt;</td>
<td>Store D in a memory location</td>
<td>m[opr]:m[opr+1] ← [A]:[B]</td>
</tr>
<tr>
<td>sts &lt;opr&gt;</td>
<td>Store SP in a memory location</td>
<td>m[opr]:m[opr+1] ← [SP]</td>
</tr>
<tr>
<td>stx &lt;opr&gt;</td>
<td>Store X in a memory location</td>
<td>m[opr]:m[opr+1] ← [X]</td>
</tr>
<tr>
<td>sty &lt;opr&gt;</td>
<td>Store Y in a memory location</td>
<td>m[opr]:m[opr+1] ← [Y]</td>
</tr>
</tbody>
</table>
- All except for the relative mode can be used to select the memory location.

- Examples:

  ld aa 0,X ; A = the content of the memory location pointed by X + 0
  staa $20 ; Store the content of A in the memory location $20
  stx $8000 ; Store the content of register X in memory location at $8000 and $8001

  ldd #100 ; d = $0100
  ldab $1004 ; B = the content of $1004
  ldx #$6000 ; X = $6000 this can be the beginning address of an array
  ldd 0,X ; read the first two bytes in the array
  ldd 2,X ; read the second two bytes in the array
  leax 2,X ; X = the address (not the content) = X + 2
  leay d,y ; Y = D + Y
2- Transfer Instructions

- Copy the contents of a register into another register. Source register is not changed.

- TAB copies A to B and TBA copies B to A.

- TAB and TBA change N and Z, V = 0, and does not change C.

- The TFR instruction does not affect any condition code bits.

- For example:

  TFR D,X ; X = D
  TFR A,B ; B = A
  TFR X,A ; X[7:0] ⇒ A, lower 8 bits of X are copied to A
  TFR A,X ; A is extended to an unsigned 16-bit number and stored in X. X = 00: contents of A

- When transferring 8-bit register to 16-bit register, the content of the 8-bit register is extended as unsigned 16-bit number by adding zeroes on left.
The exchange instruction

- The **EXG** instruction **swaps** the contents of a pair of registers. The contents of the two registers change.

- For example:
  
  ```
  exg A,B ; if A = 1 and B = 2, after executing the instruction
  A = 2 and B = 1
  exg D,X
  exg A,X ; A ← X[7:0], X ← $00:[A]
  exg X,B ; X ← $00:[B], B ← X[7:0]
  Un Signed Exchange 8-bits register with 16 bits one
  ```

- **Signed Exchange:** **SEX** A,X
  
  - A = X[0:7] the lowest 8 bits in X
  - X = $00:[A] if the number in A is positive
  - X = $FF:[A] if the number in A is negative
  
  - To extend 8-bit positive number to 16 bits, add zeroes on the left
  - To extend 8-bit negative number to 16 bits, add ones on the left
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAB</td>
<td>Transfer A to B</td>
<td>((A) \Rightarrow B)</td>
</tr>
<tr>
<td>TAP</td>
<td>Transfer A to CCR</td>
<td>((A) \Rightarrow CCR)</td>
</tr>
<tr>
<td>TBA</td>
<td>Transfer B to A</td>
<td>((B) \Rightarrow A)</td>
</tr>
<tr>
<td>TFR</td>
<td>Transfer register to register</td>
<td>((A, B, CCR, D, X, Y, or SP) \Rightarrow A, B, CCR, D, X, Y, or SP)</td>
</tr>
<tr>
<td>TPA</td>
<td>Transfer CCR to A</td>
<td>((CCR) \Rightarrow A)</td>
</tr>
<tr>
<td>TSX</td>
<td>Transfer SP to X</td>
<td>((SP) \Rightarrow X)</td>
</tr>
<tr>
<td>TSY</td>
<td>Transfer SP to Y</td>
<td>((SP) \Rightarrow Y)</td>
</tr>
<tr>
<td>TXS</td>
<td>Transfer X to SP</td>
<td>((X) \Rightarrow SP)</td>
</tr>
<tr>
<td>TYS</td>
<td>Transfer Y to SP</td>
<td>((Y) \Rightarrow SP)</td>
</tr>
</tbody>
</table>

**Exchange Instructions**

| EXG      | Exchange register to register | \((A, B, CCR, D, X, Y, or SP) \leftrightarrow (A, B, CCR, D, X, Y, or SP)\) |
| XGDX     | Exchange D with X            | \((D) \leftrightarrow (X)\)    |
| XGDY     | Exchange D with Y            | \((D) \leftrightarrow (Y)\)    |

**Sign Extension Instruction**

<table>
<thead>
<tr>
<th>SEX</th>
<th>Sign extend 8-Bit operand</th>
<th>Sign-extended ((A, B, or CCR) \Rightarrow D, X, Y, or SP)</th>
</tr>
</thead>
</table>
3- Move Instructions

- Copy a byte or a word from a memory location (or immediate value) to other memory location
- Do not change all the flags.
- Example:

  movb $1000,$2000 ; Copies the byte at memory location $1000 to the memory location at $2000
  movw 0,X,0,Y     ; Copy 16 bit word pointed by X+0 to the location pointed by Y+0
  movb #3A,$0F     ; the memory location 0F = 3A

<table>
<thead>
<tr>
<th>Move instructions</th>
<th>Function</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>movb &lt;src&gt;, &lt;dest&gt;</td>
<td>Move byte (8-bit)</td>
<td>dest ← [src]</td>
</tr>
<tr>
<td>movw &lt;src&gt;, &lt;dest&gt;</td>
<td>Move word (16-bit)</td>
<td>dest ← [src]</td>
</tr>
</tbody>
</table>
Outline

2.1 Assembly language program structure
2.2 Data transfer instructions

**2.3 Arithmetic instructions**

2.4 Branch and loop instructions
2.5 Shift and rotate instructions
2.6 Boolean logic instructions
2.7 Bit test and manipulate instructions
2.8 Stack
2.9 Subroutines
1- Add and Subtract Instructions

- The destinations are always a CPU register.
- Update N, Z, N, and C flags.
- Examples:

  adda $1000 ; A ← [A] + [$1000]
  adca $1000 ; A ← [A] + [$1000] + C
  suba $1002 ; A ← [A] - [$1002]
  sbca $1000 ; A ← [A] - [$1000] – C

(A - B) is done by adding A to the two’s complement of B
### Add Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>aba</td>
<td>Add B to A</td>
<td>A ← [A] + [B]</td>
</tr>
<tr>
<td>abx</td>
<td>Add B to X</td>
<td>X ← [X] + [B]</td>
</tr>
<tr>
<td>aby</td>
<td>Add B to Y</td>
<td>Y ← [Y] + [B]</td>
</tr>
<tr>
<td>adca &lt;opr&gt;</td>
<td>Add with carry to A</td>
<td>A ← [A] + [opr] + C</td>
</tr>
<tr>
<td>adcb &lt;opr&gt;</td>
<td>Add with carry to B</td>
<td>B ← [B] + [opr] + C</td>
</tr>
<tr>
<td>adda &lt;opr&gt;</td>
<td>Add without carry to A</td>
<td>A ← [A] + [opr]</td>
</tr>
<tr>
<td>addb &lt;opr&gt;</td>
<td>Add without carry to B</td>
<td>B ← [B] + [opr]</td>
</tr>
<tr>
<td>addd &lt;opr&gt;</td>
<td>Add without carry to D</td>
<td>D ← [D] + [opr]</td>
</tr>
</tbody>
</table>

### Subtract Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>sba</td>
<td>Subtract B from A</td>
<td>A ← [A] - [B]</td>
</tr>
<tr>
<td>sbca &lt;opr&gt;</td>
<td>Subtract with borrow from A</td>
<td>A ← [A] - [opr] - C</td>
</tr>
<tr>
<td>sbcb &lt;opr&gt;</td>
<td>Subtract with borrow from B</td>
<td>B ← [B] - [opr] - C</td>
</tr>
<tr>
<td>suba &lt;opr&gt;</td>
<td>Subtract memory from A</td>
<td>A ← [A] - [opr]</td>
</tr>
<tr>
<td>subb &lt;opr&gt;</td>
<td>Subtract memory from B</td>
<td>B ← [B] - [opr]</td>
</tr>
<tr>
<td>subd &lt;opr&gt;</td>
<td>Subtract memory from D</td>
<td>D ← [D] - [opr]</td>
</tr>
</tbody>
</table>

<opr> field is specified using one of the addressing modes. All addressing modes except inherent and relative can be used.
- **Zero flag (Z):** set when the result is zero

- **Negative flag (N):** set whenever the result is negative, i.e., most significant bit of the result is 1.

- **Carry/borrow flag (C):** set when addition/subtraction generates a carry/borrow.

- **Overflow flag (V):** Set when:

  - the addition of two positive numbers results in a negative number

  or

  - the addition of two negative numbers results in a positive number.
Overflow

**Problem:** fixed width registers have limited range

Overflow occurs when two numbers are added or subtracted and the correct result is outside the range that can a register hold → the given result is not correct

**Addition:**

\[ C = 1 \rightarrow \text{there is an overflow if the numbers are unsigned.} \]

\[ V = 1 \rightarrow \text{there is an overflow if the numbers are signed.} \]

Overflow cannot occur when adding numbers of opposite sign why?

**Subtraction:** \( A - B \)

There is no unsigned overflow but there is signed overflow

\[ C = 1, \text{ when there is a borrow or } B > A, \textbf{C is called borrow flag} \]

\[ V = 1, \text{ when} \]

\(-\text{ve}) - (+\text{ve}) = (+\text{ve}) \text{ this is equivalent to } (-\text{ve}) + (-\text{ve}) = (+\text{ve})\]

\(+\text{ve}) - (-\text{ve}) = (-\text{ve}) \text{ this is equivalent to } (+\text{ve}) + (+\text{ve}) = (-\text{ve})\]
Signed numbers: \(-1 + 1 = 0\), no overflow and the result is correct.

Unsigned numbers: \(255 + 1 = 0\), incorrect, the correct result is 256, overflow because the max. unsigned number for 8 bit number is 255.

\[
\begin{array}{cccc}
 1111 & 1111 \\
 0000 & 0001 \\
\hline
 0000 & 0000 \\
\end{array}
\]

C = 1, V = 0, Z = 1, N = 0

By using more bits (9 bits or more) instead of 8 bits, the result is correct, no overflow.

\[
\begin{array}{cccc}
 0 & 1111 & 1111 \\
 0000 & 0001 \\
\hline
 10000 & 0000 \\
\end{array}
\]

Signed numbers: \(-86 + 85 = -1\), no overflow and the result is correct.

Unsigned numbers: \(170 + 85 = 255\), no overflow and the result is correct.
Unsigned numbers: $127 + 1 = 128$, no overflow and the result is **correct**

Signed numbers: $127 + 1 = -128$, there is overflow, the result is **incorrect**, the max. positive number in 8 bits is 127 that is less than the correct answer 128. If we use 9 bit addition, the result will be correct because 128 can be represented in 9 bits.

Unsigned numbers: $172 + 138 = 54$ should be 310, overflow, 255 is the max. number for 8 bit number

Signed numbers: $-84 + (-118) = 54$, should be -202, overflow, the max. negative number in 8 bits is -128 that is less than the correct answer
If we use 9 bit addition, the result will be correct.

\[
\begin{array}{c}
\text{Unsigned numbers:} \\
0 \ 1010 \ 1100 \\
+ \ 0 \ 1000 \ 1010 \\
\hline
1 \ 0011 \ 0110 \\
\end{array}
\]

\[
\begin{array}{c}
\text{Signed numbers:} \\
1 \ 1010 \ 1100 \\
+ \ 1 \ 1000 \ 1010 \\
\hline
1 \ 0011 \ 0110 \\
\end{array}
\]

\[C = 0\]

\[V = 0\]

**Subtraction:**  \(A - B = A + \text{the two’s complement of } B\)

\[
\begin{array}{c}
0111 \ 1010 \\
- \ 0101 \ 1100 \\
\hline
0111 \ 1010 \\
+ \ 1010 \ 0100 \\
\hline
1 \ 0001 \ 1110 \\
\end{array}
\]

\[V = 0, \ C = 0, \ N = 0, \ Z = 0\]

\[C\] is called borrow flag and it is set when we need to borrow from the most significant byte

**Unsigned numbers:**  \(122 - 92 = 30\) correct

**Signed numbers:**  \(122 - 92 = 30\) correct

\[V = 0\] because \((+ve) - (+ve)\) no overflow
Unsigned numbers:
- There is borrow, $92 - 138 = 210$
- What happened is $(92+256) - 138 = 210$, where 256 is the borrow.
- If we do multi-byte subtraction, the result (210) is right and we should subtract one from the next byte.
- If you want to get the absolute difference 46, subtract the small number from the bigger one or $1000\ 1010 - 0101\ 1100 = 46$

Signed numbers:
$92 - (-118) = -46$ should be 210, $V = 1$ means the numbers should be represented in more bits.

\[
\begin{array}{c}
0 & 0101 & 1100 \\
- & 1 & 1000 & 1010 \\
0 & 0101 & 1100 \\
+ & 0 & 0111 & 0110 \\
0 & 1101 & 0010
\end{array}
\]
Write a code to subtract the contents of the memory location at $1005 from the sum of the memory locations at $1000 and $1002, and store the difference at $1100.

```assembly
ldaa $1000 ; A = [$1000]
adda $1002 ; A = A + [$1002]
suba $1005 ; A = A - [$1005]
staa $1100 ; [$1100] = A
```

Write a code to swap the 2 bytes at $100 and $200.

```assembly
ldaa 0,X ; store the byte pointed by X in A
adda 1,X ; add the following byte to A
staa 0,Y ; store the sum at location pointed by Y
```

Write a code to add the byte pointed by register X and the following byte and store the sum at the memory location pointed by register Y.

```assembly
ldaa 0,X ; store the byte pointed by X in A
adda 1,X ; add the following byte to A
staa 0,Y ; store the sum at location pointed by Y
```
ldaa $100 ; A = [$100]
movb $200,$100 ; store [$200] in memory location $100
staa $200 ; store A in the memory location $200

Write a code to add 3 to the memory locations at $10 and $15.

A memory location cannot be the destination in ADD instructions. We need to copy the memory content into register A or B, add 3 to it, and then return the sum back to the same memory location.

ldaa $10 ; copy the contents of memory location $10 to A
adda #3 ; add 3 to A
staa $10 ; store the sum to memory location at $10

ldaa $15 ; copy the contents of memory location $15 to A
adda #3 ; add 3 to A
staa $15 ; store the sum to memory location at $15
Multi-precision arithmetic

- HCS12 can add/sub at most 16-bit numbers using one instruction

- To add/sub numbers that are larger than 16 bits, we need to consider the carry or borrow resulted from 16-bit operation.

How to add (or subtract) two 32-bit numbers

- Carry flag is set to 1 when the addition operation produces a carry. This carry should be added to the next addition operation

- Carry flag is set to 1 when the subtraction operation produces a borrow. This borrow should be subtracted from the next subtraction operation
Write a program to add two 4-byte numbers that are stored at $1000-$1003 and $1004-$1007, and store the sum at $1010-$1013.

The addition starts from the lease significant byte and proceeds toward the most significant number.

Notice there is no instruction for addition with carry for 16 bits.
; Add and save the least significant two bytes
ldd $1002 ; D ← [$1002]:[$1003]
add $1006 ; D ← [D] + [$1006]:[$1007]
std $1012 ; [$1012]:[$1013] ← [D]

; Add and save the second most significant bytes
ldaa $1001 ; A ← [$1001]
adca $1005 ; A ← [A] + [$1005] + C
staa $1011 ; $1011 ← [A]

; Add and save the most significant bytes
ldaa $1000 ; A ← [$1000]
adca $1004 ; A ← [A] + [$1004] + C
staa $1010 ; $1010 ← [A]

For subtraction: The same code can be used but use
subd instead of addd
sbca instead of adca
These instructions are faster than using Add/sub instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC</td>
<td>Decrement memory by 1</td>
<td>$M \leftarrow (M) - 1$</td>
</tr>
<tr>
<td>DECA</td>
<td>Decrement A by 1</td>
<td>$A \leftarrow (A) - 1$</td>
</tr>
<tr>
<td>DECB</td>
<td>Decrement B by 1</td>
<td>$B \leftarrow (B) - 1$</td>
</tr>
<tr>
<td>DES</td>
<td>Decrement SP by 1</td>
<td>$SP \leftarrow (SP) - 1$</td>
</tr>
<tr>
<td>DEX</td>
<td>Decrement X by 1</td>
<td>$X \leftarrow (X) - 1$</td>
</tr>
<tr>
<td>DEY</td>
<td>Decrement Y by 1</td>
<td>$Y \leftarrow (Y) - 1$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>INC</td>
<td>Increment memory by 1</td>
<td>$M \leftarrow (M) + 1$</td>
</tr>
<tr>
<td>INCA</td>
<td>Increment A by 1</td>
<td>$A \leftarrow (A) + 1$</td>
</tr>
<tr>
<td>INCB</td>
<td>Increment B by 1</td>
<td>$B \leftarrow (B) + 1$</td>
</tr>
<tr>
<td>INS</td>
<td>Increment SP by 1</td>
<td>$SP \leftarrow (SP) + 1$</td>
</tr>
<tr>
<td>INX</td>
<td>Increment X by 1</td>
<td>$X \leftarrow (X) + 1$</td>
</tr>
<tr>
<td>INY</td>
<td>Increment Y by 1</td>
<td>$Y \leftarrow (Y) + 1$</td>
</tr>
</tbody>
</table>

<opr> can be direct, extended, or indexed addressing modes.
### Clear, Complement and Negate instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Operation</th>
<th>Updated flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>clc</td>
<td>Clear C bit in CCR</td>
<td>$0 \Rightarrow C$</td>
<td>$C = 0$</td>
</tr>
<tr>
<td>cli</td>
<td>Clear I bit in CCR</td>
<td>$0 \Rightarrow I$</td>
<td>$I = 0$</td>
</tr>
<tr>
<td>clr &lt;opr&gt;</td>
<td>Clear memory</td>
<td>$M \Rightarrow M$</td>
<td>$NZVC = 0100$</td>
</tr>
<tr>
<td>clra</td>
<td>Clear A</td>
<td>$A \Rightarrow A$</td>
<td>$V = 0$</td>
</tr>
<tr>
<td>clrb</td>
<td>Clear B</td>
<td>$B \Rightarrow B$</td>
<td>$NZVC = YY01$</td>
</tr>
<tr>
<td>clv</td>
<td>Clear V bit in CCR</td>
<td>$V \Rightarrow V$</td>
<td>$NZVC = YYYY$</td>
</tr>
<tr>
<td>com &lt;opr&gt;</td>
<td>One’s complement memory</td>
<td>$FF - (M) \Rightarrow M$ or $(\overline{M}) \Rightarrow M$</td>
<td></td>
</tr>
<tr>
<td>coma</td>
<td>One’s complement A</td>
<td>$FF - (A) \Rightarrow A$ or $(\overline{A}) \Rightarrow A$</td>
<td></td>
</tr>
<tr>
<td>comb</td>
<td>One’s complement B</td>
<td>$FF - (B) \Rightarrow B$ or $(\overline{B}) \Rightarrow B$</td>
<td></td>
</tr>
<tr>
<td>neg &lt;opr&gt;</td>
<td>Two’s complement memory</td>
<td>$00 - (M) \Rightarrow M$ or $(\overline{M}) + 1 \Rightarrow M$</td>
<td></td>
</tr>
<tr>
<td>nega</td>
<td>Two’s complement A</td>
<td>$00 - (A) \Rightarrow A$ or $(\overline{A}) + 1 \Rightarrow A$</td>
<td></td>
</tr>
<tr>
<td>negb</td>
<td>Two’s complement B</td>
<td>$00 - (B) \Rightarrow B$ or $(\overline{B}) + 1 \Rightarrow B$</td>
<td></td>
</tr>
</tbody>
</table>

- Clear command stores 0 in registers/memory locations. Used for initialization.
- Complement command computes the one’s complement.
- Negate command computes the two’s complement.
### 4- Multiplication and Division instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Operation</th>
<th>NZVC</th>
</tr>
</thead>
<tbody>
<tr>
<td>emul</td>
<td>unsigned 16 by 16 multiply</td>
<td>((D) \times (Y) \rightarrow Y:D)</td>
<td>YYNY</td>
</tr>
<tr>
<td>emuls</td>
<td>signed 16 by 16 multiply</td>
<td>((D) \times (Y) \rightarrow Y:D)</td>
<td>NNNY</td>
</tr>
<tr>
<td>mul</td>
<td>unsigned 8 by 8 multiply</td>
<td>((A) \times (B) \rightarrow A:B)</td>
<td>YYYY</td>
</tr>
<tr>
<td>ediv</td>
<td>unsigned 32 by 16 divide</td>
<td>((Y:D) \div (X)) quotient (\rightarrow Y) remainder (\rightarrow D)</td>
<td>YYYY</td>
</tr>
<tr>
<td>edivs</td>
<td>signed 32 by 16 divide</td>
<td>((Y:D) \div (X)) quotient (\rightarrow Y) remainder (\rightarrow D)</td>
<td>YYYY</td>
</tr>
<tr>
<td>fdiv</td>
<td>16 by 16 fractional divide</td>
<td>((D) \div (X) \rightarrow X) remainder (\rightarrow D)</td>
<td>NYYY</td>
</tr>
<tr>
<td>idiv</td>
<td>unsigned 16 by 16 integer divide</td>
<td>((D) \div (X) \rightarrow X) remainder (\rightarrow D)</td>
<td>NYYY</td>
</tr>
<tr>
<td>idivs</td>
<td>signed 16 by 16 integer divide</td>
<td>((D) \div (X) \rightarrow X) remainder (\rightarrow D)</td>
<td>YYYY</td>
</tr>
</tbody>
</table>

The upper 16 bits in Y and the lower ones in D.
Write an instruction sequence to multiply the 16-bit numbers stored at $1000$-$1001$ and $1002$-$1003$ and store the product at $1100$-$1103$.

```
ldd $1000 ; load first word
ldy $1002 ; load second word
emul ; [D] x [Y] \rightarrow Y:D use emuls if the numbers are signed
sty $1100 ; store most significant 16 bits in 1100 and 1101
std $1102 ; store least significant 16 bits in 1102 and 1103
```

Write an instruction sequence to divide the signed 16-bit number stored at $1020$-$1021$ by the signed 16-bit number stored at $1005$-$1006$ and store the quotient and remainder at $1100$ and $1102$, respectively.

```
ldd $1005
ldx $1020
idivs ; D/X X = quotient, D = remainder, use idiv if numbers are unsigned
stx $1100 ; store the quotient (16 bits) at $1100$ and $1101$
std $1102 ; store the remainder (16 bits)
```

; To compute the squared value of A
tab ; B = A
mul ; A:B = A \times B
Converting binary number to decimal

Input: Binary number

11000000111001 (= 12345 in decimal)

Binary to decimal conversion

Output: equivalent decimal number

Can be sent to the LCD
- Using repeated division by 10.
- The largest 16-bit number is 65,535 which has five decimal digits.
- The first division by 10 generates the least significant digit (in the remainder).

<table>
<thead>
<tr>
<th>12345</th>
<th>Quotient</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>= 1234</td>
<td>5</td>
</tr>
<tr>
<td>123</td>
<td>= 123</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>= 12</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>= 1</td>
<td>2</td>
</tr>
</tbody>
</table>

The least significant digit is 5, and the most significant digit is 1.
Write a program to convert the 16-bit number stored at register D to decimal store the result at memory locations $1010$ to $1014$.

\begin{verbatim}
ldy #$1010 ; Y points at the first address of the decimal result

ldx #10 ; X = 10
idiv ; D/X Quotient \rightarrow X, Remainder \rightarrow D
stab 4,Y ; save the least significant digit (5)
xgdx ; D = the quotient required for the next division

ldx #10 ; X = 10
idiv ; D/10 Quotient \rightarrow X, Remainder \rightarrow D
stab 3,Y ; save the second number (4)
xgdx ; D = the quotient required for the next division
\end{verbatim}

Assume: D = 12345

\begin{align*}
Y & \rightarrow $1010 \\
   & \rightarrow $1011 \\
   & \rightarrow $1012 \\
   & \rightarrow $1013 \\
   & \rightarrow $1014 \\
Y + 4 & \rightarrow $1014
\end{align*}
ldx #10 ; X = 10
idiv ; D/10 Quotient → X, Remainder → D
stab 2,Y ; save the third number (3)
xgdx ; D = the quotient

ldx #10 ; X = 10
idiv ; D/10 Quotient → X, Remainder → D
stab 1,Y ; save the second number (2)
xgdx ; D = the quotient

addb #$30
stab 0,Y ; save the most significant digit (1)

Y → $1010 1
    $1011 2
    $1012 3
    $1013 4
Y+4 → $1014 5
Converting decimal number to binary

Input: decimal number

Decimal to binary conversion

Output: equivalent binary number

Most significant number

N1 = 1
N2 = 2
N3 = 3
N4 = 4
N5 = 5

Can come from the keypad

11000000111001 (= 12345 in decimal)
Binary = (((((N1 \times 10) + N2) \times 10 + N3) \times 10 + N4) \times 10 + N5

= N1 \times 10000 + N2 \times 1000 + N3 \times 100 + N4 \times 10 + N5

Write a program to convert a 5-digit number stored at memory locations $1010$ to $1014$ into its 16-bit equivalent binary number. Store the result in memory locations $2000$ and $2001$.

; processing N1
ldaa $1010  ; A = N1 \text{ (the most significant digit)}
ldab #10    ; B = 10
mul          ; \textbf{operation 1} D = N1 \times 10
std $2000    ; \text{store the result in }$2000 \text{ and } $2001

; processing N2
ldab $1011  ; B = N2
clra         ; A = 0 so D = A:B = 00: N2
add $2000    ; \textbf{operation 2} d = [$2000]+D = (N1 \times 10) + N2
; process N3
ldy #10    ; Y = 10
emul       ; Y:D = D x Y operation 3
std $2000   ; d = ((N1 x 10) + N2) x 10
ldab $1012  ; B = N3
chra        ; A = 0 so D = A:B = 00: N3
add $2000   ; operation 4 d = ((N1 x 10) + N2) x 10 + N3

; process N4
ldy #10    ; Y = 10
emul       ; Y:D = D x Y operation 5
std $2000   ; d = (((N1 x 10) + N2) x 10 + N3) x 10
ldab $1013  ; B = N3
chra        ; A = 0 so D = A:B = 00: N4
add $2000   ; operation 6 d = (((N1 x 10) + N2) x 10 + N3) x 10) + N4
; process N5

ldy #10    ; Y = 10

emul       ; Y:D = D x Y **operation 7**

std $2000   ; d = d = (((((N1 x 10) + N2) x 10 + N3) x 10) + N4)x 10

ldab $1014  ; B = N5

clra       ; A = 0 so D = A:B = 00: N5

add $2000   ; **operation 8**  d = (((((N1 x 10) + N2) x 10 + N3) x 10) + N4)x 10 ) + N5
2.1 Assembly language program structure
2.2 Data transfer instructions
2.3 Arithmetic instructions
2.4 Branch and loop instructions
2.5 Shift and rotate instructions
2.6 Boolean logic instructions
2.7 Bit test and manipulate instructions
2.8 Stack
2.9 Subroutines
1. Branch instructions

Conditional or unconditional

Short or long

Signed or unsigned

Unconditional branches
- Branches are always taken

Conditional branches
- A branch is taken if a condition is satisfied.
- A condition is satisfied if certain flags are set.
- Usually we use a comparison or arithmetic command to set up the flags before the branch instruction.

CBA ; compare A to B - used to set the flags
LBHI next ; branch to next if A > B – LBHI tests the flags
1. Branch instructions

- Conditional or unconditional
- Short or long
- Signed or unsigned

**Short branches**
- The range of the branch is -128 and +127 bytes.

**Long branches**
- Can branch to anywhere in the memory

For peace of mind, always use long branches
1. Branch instructions

Conditional or unconditional

Short or long

Signed or unsigned

Unsigned branches
- The numbers of the condition are unsigned
- Use instructions: branch if higher (LBHI), branch if higher or same (LBHS), branch if lower (LBLO), and branch if lower and same (LBLS).

Signed branches
- The numbers of the condition are Signed
- Use instructions: branch if greater (LBGT), branch if greater or equal (LBGE), branch if less (LBLE), and branch if less and equal (LBLE).

; A = 1111 1111    B = 0000 0001
CPA   ; compare A and B . Used to set the flags
LBHI next ; unsigned the branch is taken because A = 225 > B =1
LBGT next ; Signed the branch is not taken because A = -1 is not greater than B =1
### Unary Branches

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Equation or Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>bra rel8  or ibra rel16</td>
<td>Branch always</td>
<td>1 = 1</td>
</tr>
<tr>
<td>brn rel8  or ibrn rel16</td>
<td>Branch never</td>
<td>1 = 0</td>
</tr>
</tbody>
</table>

### Simple Branches

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Equation or Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>bcc rel8   or ibcc rel16</td>
<td>Branch if carry clear</td>
<td>C = 0</td>
</tr>
<tr>
<td>bcs rel8   or ibcs rel16</td>
<td>Branch if carry set</td>
<td>C = 1</td>
</tr>
<tr>
<td>beq rel8   or ibeq rel16</td>
<td>Branch if equal</td>
<td>Z = 1</td>
</tr>
<tr>
<td>bmi rel8   or ibmi rel16</td>
<td>Branch if minus</td>
<td>N = 1</td>
</tr>
<tr>
<td>bne rel8   or ibne rel16</td>
<td>Branch if not equal</td>
<td>Z = 0</td>
</tr>
<tr>
<td>bpl rel8   or ibpl rel16</td>
<td>Branch if plus</td>
<td>N = 0</td>
</tr>
<tr>
<td>bvc rel8   or ibvc rel16</td>
<td>Branch if overflow clear</td>
<td>V = 0</td>
</tr>
<tr>
<td>bus rel8   or ibus rel16</td>
<td>Branch if overflow set</td>
<td>V = 1</td>
</tr>
</tbody>
</table>

### Unsigned Branches

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Equation or Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>bhi rel8   or ibhi rel16</td>
<td>Branch if higher</td>
<td>C + Z = 0</td>
</tr>
<tr>
<td>bhs rel8   or ibhs rel16</td>
<td>Branch if higher or same</td>
<td>C = 0</td>
</tr>
<tr>
<td>blo rel8   or iblo rel16</td>
<td>Branch if lower</td>
<td>C = 1</td>
</tr>
<tr>
<td>bles rel8  or ibles rel16</td>
<td>Branch if lower or same</td>
<td>C + Z = 1</td>
</tr>
</tbody>
</table>

### Signed Branches

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Equation or Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>bge rel8   or ibge rel16</td>
<td>Branch if greater than or equal</td>
<td>N ⊕ V = 0</td>
</tr>
<tr>
<td>bgt rel8   or ibgt rel16</td>
<td>Branch if greater than</td>
<td>Z + (N ⊕ V) = 0</td>
</tr>
<tr>
<td>ble rel8   or ible rel16</td>
<td>Branch if less than or equal</td>
<td>Z + (N ⊕ V) = 1</td>
</tr>
<tr>
<td>blt rel8   or iblt rel16</td>
<td>Branch if less than</td>
<td>N ⊕ V = 1</td>
</tr>
</tbody>
</table>

**Note.**
1. Each row contains two branch instructions that are separated by the word ‘or’
2. The instruction to the left of ‘or’ is a short branch with 8-bit offset.
3. The instruction to the right of ‘or’ is a long branch with 16-bit offset.
2. Compare and Test instructions

- Flags should be set up before using conditional branch instructions.
- The compare and test instructions perform subtraction, set the flags based on the result, and does not store the result. ONLY change flags.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>cba</td>
<td>Compare A to B</td>
<td>(A) - (B)</td>
</tr>
<tr>
<td>cmpa &lt;opr&gt;</td>
<td>Compare A to memory</td>
<td>(A) - (M)</td>
</tr>
<tr>
<td>cmpb &lt;opr&gt;</td>
<td>Compare B to memory</td>
<td>(B) - (M)</td>
</tr>
<tr>
<td>cpdx &lt;opr&gt;</td>
<td>Compare D to memory</td>
<td>(D) - (M:M+1)</td>
</tr>
<tr>
<td>cps &lt;opr&gt;</td>
<td>Compare SP to memory</td>
<td>(SP) - (M:M+1)</td>
</tr>
<tr>
<td>cpx &lt;opr&gt;</td>
<td>Compare X to memory</td>
<td>(X) - (M:M+1)</td>
</tr>
<tr>
<td>cpy &lt;opr&gt;</td>
<td>Compare Y to memory</td>
<td>(Y) - (M:M+1)</td>
</tr>
</tbody>
</table>

Table 2.3 Summary of compare and test instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>tst &lt;opr&gt;</td>
<td>Test memory for zero or minus</td>
<td>(M) - $00</td>
</tr>
<tr>
<td>tsta</td>
<td>Test A for zero or minus</td>
<td>(A) - $00</td>
</tr>
<tr>
<td>tstb</td>
<td>Test B for zero or minus</td>
<td>(B) - $00</td>
</tr>
</tbody>
</table>

The memory and register does not change

<opr> can be an immediate value or a memory location
3. Loop instructions

- Repeat a sequence of instructions several times.
- Either decrement or increment a count to determine if the loop should continue.
- The range of the branch is from -128 to +127.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Equation or Operation</th>
</tr>
</thead>
</table>
| dbeq cntr, rel | Decrement counter and branch if = 0  
(cntr = A, B, D, X, Y, or SP) | cntr ← (cntr) - 1  
If (cntr) = 0, then branch  
else continue to next instruction |
| dbne cntr, rel | Decrement counter and branch if ≠ 0  
(cntr = A, B, D, X, Y, or SP) | cntr ← (cntr) - 1  
If (cntr) ≠ 0, then branch  
else continue to next instruction |
| ibeq cntr, rel | Increment counter and branch if = 0  
(cntr = A, B, D, X, Y, or SP) | cntr ← (cntr) + 1  
If (cntr) = 0, then branch  
else continue to next instruction |
| ibne cntr, rel | Increment counter and branch if ≠ 0  
(cntr = A, B, D, X, Y, or SP) | cntr ← (cntr) + 1  
If (cntr) ≠ 0, then branch  
else continue to next instruction |
| tbeq cntr, rel | Test counter and branch if = 0  
(cntr = A, B, D, X, Y, or SP) | If (cntr) = 0, then branch  
else continue to next instruction |
| tbne cntr, rel | Test counter and branch if ≠ 0  
(cntr = A, B, D, X, Y, or SP) | If (cntr) ≠ 0, then branch  
else continue to next instruction |

Note. 1. \textbf{cntr} is the loop counter and can be accumulator A, B, or D and register X, Y, or SP.  
Note: rel is the relative branch offset and usually a label
4. Bit condition branch instructions

- Make branch decision based on the value of few bits in a memory location.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>brclr &lt;opr&gt;,msk,rel</td>
<td>Branch is taken when the tested bits are zeroes</td>
</tr>
<tr>
<td>brset &lt;opr&gt;,msk,rel</td>
<td>Branch is taken when the tested bits are ones</td>
</tr>
</tbody>
</table>

<opr>: The memory location to be checked.

msk: 8 bits that specifies the bits of to be checked. The bits to be checked correspond to those that are 1s in msk.

rel: if a branch is taken, it branches to the label rel

```
loop: ................
        ................
brset $66,$E0,loop
```

The branch is taken if the last three bits at memory location $66$ are all ones.

Notice: $E0 = \%1110 0000$

```
brclr $66,$80,here
        ................
here:  ................
```

The branch is taken if the most significant bit at the memory location $66$ is zero.

Notice: $80 = \%1000 0000$
How brclr and brset work?

1 and Bi = Bi $\rightarrow$ put 1 at the bits you test
0 and Bi = 0 $\rightarrow$ put 0 at the bits you do not test

I wanna test These bits

\[
\begin{array}{cccccccc}
\text{B7} & \text{B6} & \text{B5} & \text{B4} & \text{B3} & \text{B2} & \text{B1} & \text{B0} \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
\end{array}
\]

AND

\[
\begin{array}{cccccccc}
\text{B7} & \text{B6} & \text{B5} & \text{B4} & \text{B3} & \text{B2} & \text{B1} & \text{B0} \\
0 & 0 & 0 & 0 & 0 & 0 & \text{B1} & \text{B0} \\
\end{array}
\]

This number is zero if B0 and B1 are zeros, otherwise it is not zero
1. Endless loop

do a sequence of instructions (S) forever.

```
Loop: ------  ------  ------
       LBRA Loop
```

Repeat these instructions forever

![Figure 2.4 An infinite loop](image)

2. For loops

```
For (i = n1, i <= n2, i++)
{a sequence of instructions (S) }
```

- i is loop counter that can be incremented in each iteration.
- Sequence S is repeated n2-n1+1 times

**Steps:**
1- Initialize loop counter
2- Compare the loop counter with n2. If it is not equal, do the loop otherwise exit
3- increment the loop and go to step 2
Implementation of  for (i = 1, i <= 20, i++)  {S}

i ds.b 1 ; i is the loop counter

movb #1,i ;initialize i to 1

Loop: ldaa i ; A = [i]
cmpa #2 ; check index i
LBHI Next ; if i > n2, exit the loop

........... ; performs S
........... ; "

inc i ;increment loop index
lbra Loop ;go back to the loop body

Next: ...

Since i is a byte, the max. number of iterations is 256.
For more iterations, use two loops (outer and inner loops)
For loop using dbeq

up to 65,535 iterations

```
ldx #6000     ; number of iterations

Loopf: dbeq x,next
       ........ ; performs S
       ........
       lbra Loopf

next: ...
```

May be a good idea to use a memory location as a loop counter because you may need to use X to perform the sequence S
While Loop

While (condition) { Sequence S; }

- S is executed as long as the condition is true
- Unlike for loop, the number of iterations may not be known beforehand

Figure 2.6 The While ... Do looping construct

While (A ≠ 0) {Sequence S;}

Wloop: cmpa #0
       lbeqNext
       ........ ; perform S
       ........
       lbra Wloop

Next: ...

A is updated in the instruction sequence S
If (I == 1)
{Sequence S;}

I ds.b 1
ldaa I
cmpa #1
lbne end_if

........ ; perform S
........ ; ”

end_if:

If I does not equal 1, skip the Sequence S

If (I == 1) {Sequence S1;}
else {Sequence S2;}

I ds.b 1
........
ldaa I ; A = I
cmpa #1
lbne else
........ ; perform S1
........ ; ”
lbra end_if
else:
........ ; perform S2
........ ; ”
end_if:
If ( A == 1 and B > 8)
{Sequence S;}

```assembly
cmpa #1
lbne end_if
; the first condition is satisfied test the second one
cmpb #8
lbls end_if

.........; perform S
.........; ”
end_if:
```

Sequence S is executed only when the two conditions are satisfied, i.e., if one condition is not satisfied, do not execute S.

If ( A == 1 or B > 8)
{Sequence S;}

```assembly
cmpa #1
lbeq perform_S
; the first condition is not satisfied. Try the second one
cmpb 8
lbhi perform_S
; the two conditions are not satisfied, go to end_if
lbra end_if

perform_S:

.........; perform S
.........; ”
end_if:
```

Sequence S should be executed when at least one condition is satisfied, i.e., S is not executed when the two conditions are not satisfied.
If (A == 1)
{Sequence S1;}
Else If (A == 2)
{Sequence S2;}
Else If (A == 3)
{Sequence S3;}
Else
{Sequence Se;}

end_if:
Switch Case

Switch (variable)
Case 1:
    Sequence S1;
    Break;
Case 2:
    Sequence S2;
    Break;
Case 3:
    Sequence S3;
    Break;
Else:
    Sequence Se;

I rmb 1
    ldaa I ; A = I
    cmpa #1
    lbeq Case1
    cmpa #2
    lbeq Case2
    cmpa #3
    lbeq Case3
    lbra else
Case1:
    .......... ; perform S1
    lbra end_case
Case2:
    .......... ; perform S2
    lbra end_case
Case3:
    .......... ; perform S3
    lbra end_case
else:
    .......... ; perform Se
end_case:
Write a code to calculate the absolute value of the memory location $1000$. Store the result in $1000$

```
lda $1000
cmpa #00
lbge done ; do nothing if [$1000] >=0
; the number is negative
nega
staa $1000
done:
```
Write a program to find the maximum element in an array of 20 elements and each element is byte. The array starts from location $2000$

1- max_value = Array[0]

2- Scan the array from Array[2] to Array[20]

3- In each iteration:
   
   if Array[i] > max_value then max_value = Array[i]

4- After scanning all the array elements, max_value = the max. element
org $1000    ; starting address of data
max_val ds.b 1    ; max. value is hold here

org $1500    ; starting address of program
ldaa $2000    ; A = the first element
staa max_val    ; max_val = the first element
ldx #$2001    ; X = the address of the second element
ldab #19    ; b is the loop count = 19

Loop:  ldaa max_val    ; A = max_val
cmpa 0,x    ; compare A and the element at 0,X
lbge chk_end    ; do not change max_value if it is greater

; an element greater than max_val is found
movb 0,x,max_val    ; update the array’s max value
chk_end:  inx    ; move to the next array element
dbne b,Loop    ; loop for 19 times

Can you modify this code to find the minimum value?
Write a program to compute the number of elements that are divisible by 4 in an array of 5 elements. Each element is a byte. A number is divisible by 4 when the least significant two bits equal 0s.

<table>
<thead>
<tr>
<th>org $1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>total ds.b 1 ; to store the number of elements divisible by 4</td>
</tr>
<tr>
<td>array dc.b 1,2,3,4,5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>org $1500</th>
</tr>
</thead>
<tbody>
<tr>
<td>clr total ; initialize total to 0</td>
</tr>
<tr>
<td>ldx #array ; X = the starting address of the array, use X as the array pointer</td>
</tr>
<tr>
<td>ldab #5 ; use b as the loop count</td>
</tr>
</tbody>
</table>

```
loop: brclr 0,X,$03,yes ; check bits number 0 and 1
    bra chkend
yes:  inc total

chkend: inx ; point at the next element in the array
dbneb,loop
```
2.1 Assembly language program structure
2.2 Data transfer instructions
2.3 Arithmetic instructions
2.4 Branch and loop instructions
2.5 Shift and rotate instructions
2.6 Boolean logic instructions
2.7 Bit test and manipulate instructions
2.8 Stack
2.9 Subroutines
1. Logical shift instructions

1.1 Logical shift left

- **lsl <opr>**; Memory location opr is shifted left by one bit
- **lsla**; Accumulator A is shifted left by one bit
- **lslb**; Accumulator B is shifted left by one bit

After shifting A eight times, what’s the value of A?

- **lsld**; 16-bit logical shift left instruction for D

Shifting one byte data
1.2 Logical shift right

- \texttt{lsr <opr>} \quad ; \text{Memory location opr is shifted right one place}
- \texttt{lsra} \quad ; \text{Accumulator A is shifted right one place}
- \texttt{lsrb} \quad ; \text{Accumulator B is shifted right one place}

\texttt{lsrd} \quad ; 16\text{-}bit logical shift right instruction for D
2. Arithmetic shift instructions

2.1 Arithmetic shift left

- Shift left is equivalent to multiply by 2.
- For example, %0000 0100 = 4  After one shift left: %0000 1000 = 8
- Faster than multiply instructions

```
asl <opr> ; Memory location opr is shifted left one place
asla ; Accumulator A is shifted left one place
aslb ; Accumulator B is shifted left one place
```

```
asld ; 16-bit arithmetic shift left instruction logical shift left D
```
2.2 Arithmetic shift right

- Arithmetic shift right is equivalent to divide by 2.
- For example, %0000 1000 = 8 After one shift right : %0000 0100 = 8
- Faster than divide instructions

Asr shifts by the last bit instead of 0 to keep the number’s sign.

No 16 bit arithmetic shift right
3. Rotate instructions

3.1 Rotate left

rol <opr> ; Memory location opr is rotated left one place
rola ; Accumulator A is rotated left one place
rolb ; Accumulator B is rotated left one place

No 16 bit rotate left instruction

After rotating A 9 times, what’s the value of A?

ror <opr> ; Memory location opr is rotated right one place
rora ; Accumulator A is rotated right one place
rorb ; Accumulator B is rotated right one place

No 16 bit rotate right instruction
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical Shifts</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSL&lt;opr&gt;</td>
<td>Logic shift left memory</td>
<td><img src="image1" alt="Diagram" /></td>
</tr>
<tr>
<td>LSLA</td>
<td>Logic shift left A</td>
<td></td>
</tr>
<tr>
<td>LSLB</td>
<td>Logic shift left B</td>
<td></td>
</tr>
<tr>
<td>LSLD</td>
<td>Logic shift left D</td>
<td><img src="image2" alt="Diagram" /></td>
</tr>
<tr>
<td>LSR&lt;opr&gt;</td>
<td>Logic shift right memory</td>
<td><img src="image3" alt="Diagram" /></td>
</tr>
<tr>
<td>LSRA</td>
<td>Logic shift right A</td>
<td></td>
</tr>
<tr>
<td>LSRB</td>
<td>Logic shift right B</td>
<td></td>
</tr>
<tr>
<td>LSRD</td>
<td>Logic shift right D</td>
<td><img src="image4" alt="Diagram" /></td>
</tr>
<tr>
<td>Arithmetic Shifts</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASL&lt;opr&gt;</td>
<td>Arithmetic shift left memory</td>
<td><img src="image5" alt="Diagram" /></td>
</tr>
<tr>
<td>ASLA</td>
<td>Arithmetic shift left A</td>
<td></td>
</tr>
<tr>
<td>ASLB</td>
<td>Arithmetic shift left B</td>
<td></td>
</tr>
<tr>
<td>ASLD</td>
<td>Arithmetic shift left D</td>
<td><img src="image6" alt="Diagram" /></td>
</tr>
<tr>
<td>ASR&lt;opr&gt;</td>
<td>Arithmetic shift right memory</td>
<td><img src="image7" alt="Diagram" /></td>
</tr>
<tr>
<td>ASRA</td>
<td>Arithmetic shift right A</td>
<td></td>
</tr>
<tr>
<td>ASRB</td>
<td>Arithmetic shift right B</td>
<td></td>
</tr>
<tr>
<td>Rotates</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROL&lt;opr&gt;</td>
<td>Rotate left memory through carry</td>
<td><img src="image8" alt="Diagram" /></td>
</tr>
<tr>
<td>ROLA</td>
<td>Rotate left A through carry</td>
<td></td>
</tr>
<tr>
<td>ROLB</td>
<td>Rotate left B through carry</td>
<td></td>
</tr>
<tr>
<td>ROR&lt;opr&gt;</td>
<td>Rotate right memory through carry</td>
<td><img src="image9" alt="Diagram" /></td>
</tr>
<tr>
<td>RORA</td>
<td>Rotate right A through carry</td>
<td></td>
</tr>
<tr>
<td>RORB</td>
<td>Rotate right B through carry</td>
<td></td>
</tr>
</tbody>
</table>
**Example:** Suppose that $[A] = 95$ and $C = 1$. Compute the new values of $A$ and $C$ after the execution of the instruction asla.

![Figure 2.11a Operation of the ASLA instruction](image)

<table>
<thead>
<tr>
<th>Original value</th>
<th>New value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[A] = 10010101$</td>
<td>$[A] = 00101010$</td>
</tr>
<tr>
<td>$C = 1$</td>
<td>$C = 1$</td>
</tr>
</tbody>
</table>

Figure 2.11b Execution result of the ASLA instruction

**Example:** Suppose that $m[\$800] = ED$ and $C = 0$. Compute the new values of $m[\$800]$ and $C$ after the execution of `asr $1000$`.

![Figure 2.12a Operation of the ASR $1000$ instruction](image)

<table>
<thead>
<tr>
<th>Original value</th>
<th>New value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[$1000] = 11101101$</td>
<td>$[$1000] = 11110110$</td>
</tr>
<tr>
<td>$C = 0$</td>
<td>$C = 1$</td>
</tr>
</tbody>
</table>

Figure 2.12b Result of the `asr $1000$` instruction
**Example:** Suppose that \( m[\$800] = \$E7 \) and \( C = 1 \). Compute the new contents of \( m[\$800] \) and \( C \) after the execution of \texttt{lsr } \$800.  

![Figure 2.13a Operation of the LSR \$800 instruction](image)

<table>
<thead>
<tr>
<th>Original value</th>
<th>New value</th>
</tr>
</thead>
<tbody>
<tr>
<td>([$800] = 11100111 )</td>
<td>([$800] = 01110011 )</td>
</tr>
<tr>
<td>( C = 1 )</td>
<td>( C = 1 )</td>
</tr>
</tbody>
</table>

![Figure 2.13b Execution result of LSR \$800](image)

**Example:** Suppose that \([B] = \$BD \) and \( C = 1 \). Compute the new values of \( B \) and the \( C \) flag after the execution of \texttt{rolb}. 

![Figure 2.14a Operation of the instruction ROLB](image)

<table>
<thead>
<tr>
<th>Original value</th>
<th>New value</th>
</tr>
</thead>
<tbody>
<tr>
<td>([B] = 10111101 )</td>
<td>([B] = 01111011 )</td>
</tr>
<tr>
<td>( C = 1 )</td>
<td>( C = 1 )</td>
</tr>
</tbody>
</table>

![Figure 14b. Execution result of ROLB](image)
Example: Suppose that $[A] = $BE and $C = 1$. Compute the new values of $A$ and $C$ after the execution of the instruction `rora`.

![Diagram showing the operation of the `rora` instruction]

<table>
<thead>
<tr>
<th>Original value</th>
<th>New value</th>
</tr>
</thead>
</table>
| $[A] = 10111110$  
$C = 1$ | $[A] = 11011111$  
$C = 0$ |

Figure 2.15a Operation of the instruction `rora`

Figure 2.15b Execution result of `rora`
**Example:** Write a program to count the number of 0s in the 16-bit D register and store the result in memory location $1005.$

- The 16-bit number is shifted to the right
- If the bit shifted out is a 0 then increment the 0s count by 1.
- Loop for 16 iterations
Can you modify the program to count the number of ones?

An application: In voting system, each bit can reflect a switch condition (connected or disconnected). We use this program to count the number of approvals (ones) and the number of disapprovals (zeros).
HCS12 instructions can shift 8 or 16 bit numbers

Rotate right can be used a shift right with carry

Use lsl and rol for shifting left
Example Write a program to shift the 32-bit number stored at $820$-$823$ to the right four places.

```assembly
ldab#4 ; set up the loop count = the number of shifts
ldx #$820 ; use X as the pointer to the left most byte

Again: lsr 0,X
ror 1,X
ror 2,X
ror 3,X

dbne b,Again ; decrement b and loops if it is not 0
```

Can you change the code to shift the 32-bit number to the left?
2.1 Assembly language program structure
2.2 Data transfer instructions
2.3 Arithmetic instructions
2.4 Branch and loop instructions
2.5 Shift and rotate instructions
2.6 Boolean logic instructions
2.7 Bit test and manipulate instructions
2.8 Stack
2.9 Subroutines
- Logic instructions perform a logic operation between an 8-bit accumulator or the CCR and a memory or immediate value.

Table 2.7. Boolean Logic Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>anda &lt;opr&gt;</td>
<td>AND A with memory</td>
<td>(A) \cdot (M) \Rightarrow A</td>
</tr>
<tr>
<td>andb &lt;opr&gt;</td>
<td>AND B with memory</td>
<td>(B) \cdot (M) \Rightarrow B</td>
</tr>
<tr>
<td>andcc &lt;opr&gt;</td>
<td>AND CCR with memory (clear CCR bits)</td>
<td>(CCR) \cdot (M) \Rightarrow CCR</td>
</tr>
<tr>
<td>eora &lt;opr&gt;</td>
<td>Exclusive OR A with memory</td>
<td>(A) \oplus (M) \Rightarrow A</td>
</tr>
<tr>
<td>eorb &lt;opr&gt;</td>
<td>Exclusive OR B with memory</td>
<td>(B) \oplus (M) \Rightarrow B</td>
</tr>
<tr>
<td>oraa &lt;opr&gt;</td>
<td>OR A with memory</td>
<td>(A) + (M) \Rightarrow A</td>
</tr>
<tr>
<td>orab &lt;opr&gt;</td>
<td>OR B with memory</td>
<td>(B) + (M) \Rightarrow B</td>
</tr>
<tr>
<td>orcc &lt;opr&gt;</td>
<td>OR CCR with memory (set CCR bits)</td>
<td>(CCR) + (M) \Rightarrow CCR</td>
</tr>
</tbody>
</table>

<opr> can be specified using all except the relative addressing modes
“AND” is used to reset one or more bits

Ex. Clear the first 4 bits in register B

\[
\begin{array}{cccccccc}
B7 & B6 & B5 & B4 & B3 & B2 & B1 & B0 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
\hline
B7 & B6 & B5 & B4 & 0 & 0 & 0 & 0 \\
\end{array}
\]

Thanks to: \( B_i AND 0 = 0 \)
\( B_i AND 1 = B_i \)

“OR” is used to set one or few bits

Ex. Set the first 4 bits in register B

\[
\begin{array}{cccccccc}
B7 & B6 & B5 & B4 & B3 & B2 & B1 & B0 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
\hline
B7 & B6 & B5 & B4 & 1 & 1 & 1 & 1 \\
\end{array}
\]

Thanks to: \( B_i OR 0 = B_i \)
\( B_i OR 1 = 1 \)
"XOR" is used to flip (change 0 to 1 and 1 to 0) one or more bits.

Ex. Flip the first 4 bits in register B

Thanks to: Bi XOR 0 = Bi
Bi XOR 1 = Bi'
Bi' = the inversion of Bi

I wanna set these bits

<table>
<thead>
<tr>
<th>B7</th>
<th>B6</th>
<th>B5</th>
<th>B4</th>
<th>B3</th>
<th>B2</th>
<th>B1</th>
<th>B0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

B XOR mask

```
1010 1101 (data)
0000 1111 (mask)
```

result

```
1010 0010
```

bits preserved
bits inverted

```
1010 1101 (data)
0000 1111 (mask)
```

result

```
0000 1101
```

cleared bits

```
1010 1101 (data)
1111 0000 (mask)
```

result

```
1111 1101
```

bits set
bits preserved
ldaa $56
anda #$0F
staa $56

Clear the upper 4 pins of the I/O port located at $56

ldaa $56
oraa #$01
staa $56

Set the bit 0 of the I/O port at $56

ldaa $56
eora #$0F
staa $56

Toggle (or flip) the lower 4 bits of the I/O port at $56

ldaa $10
oraa #$00011000
staa $10

Force bits 3,4 of [$10] to be 1’s

ldaa $10
anda #%11100111
staa $10

Force bits 3,4 of [$10] to be 0’s

ldaa $56
oraa #$0F
staa $56

Test if both bits 5 and 6 of A are zeroes

ldaa $10
oraa #%01100000
lbeq bothzeros ; branch if Z flag = 1

Test if both bits 5 and 6 of A are zeroes
Outline

2.1 Assembly language program structure
2.2 Data transfer instructions
2.3 Arithmetic instructions
2.4 Branch and loop instructions
2.5 Shift and rotate instructions
2.6 Boolean logic instructions

2.7 Bit test and manipulate instructions

2.8 Stack
2.9 Subroutines
<opr>: memory location.

msk8: 8-bit mask value. Used to test or change the value of individual bits.

bita and bitb are used to test bits without changing the value of the operand. They do AND operation and update flags but do not store the result. Used to set the flags before a branch instruction.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>bclr &lt;opr&gt;²,msk8</td>
<td>Clear bits in memory</td>
<td>(M) • (mm) ⇒ M</td>
</tr>
<tr>
<td>bita &lt;opr&gt;¹</td>
<td>Bit test A</td>
<td>(A) • (M)</td>
</tr>
<tr>
<td>bitb &lt;opr&gt;¹</td>
<td>Bit test B</td>
<td>(B) • (M)</td>
</tr>
<tr>
<td>bset &lt;opr&gt;²,msk8</td>
<td>Set bits in memory</td>
<td>(M) + (mm) ⇒ M</td>
</tr>
</tbody>
</table>

ldaa $10
bita #%01100000
bne eitherones

Bita does [$10] AND %01100000. This masks off (zeroes) all bits except bits 5 and 6. Branch is taken if either bit 5 or 6 is one.
bclr 0,X,$81 ; clear the most significant and least significant bits of the memory location pointed by register X ($81=%10000001)

bclr $812,$81 ; Clear bits 0 and 7 in location $812. It does not change the other bits.

bset 0,y,$33 ; Sets bits five, four, one, and zero of memory location pointed to by register Y

bset $810,$4 ; Set bit 2 in memory location $810. It does not change the other bits.

What is the difference between bclr and brclr (slide 2-64)