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 Arithmetic instructions

2.3 Branch and loop instructions

2.4 Shift and rotate instructions

2.5 Boolean logic instructions

2.6 Bit test and manipulate instructions
Sample Program

Directive: Tells loader where to put program

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>

Directive: Tells assembler where program finished

Assembler directives
- Commands to the assembler
- Not executable by the microprocessor – are not converted to machine codes
- Define program constants and reserve space for dynamic variable
- Specifies the end of a program.
1. end  
   - Ends a program to be processed by an assembler  
   - Any statement following the end directive is ignored

2. 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.

3. 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  

| 800 | $11 |
| 801 | $22 |
| 802 | $33 |
| 803 | $44 |
4. 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
```

5. fcc (form constant character)
- Tells the assembler to store a string of characters (a message) in memory.
- The first character (and the last character) is used as the delimiter.
- The last character must be the same as the first character.
- The delimiter must not appear in the string.
- The space character cannot be used as the delimiter.
- Each character is represented by its ASCII code.
- For example: msg fcc “Please enter your name:”

<table>
<thead>
<tr>
<th>Org $1000</th>
<th>1000</th>
<th>$64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha fcc “def”</td>
<td>1001</td>
<td>$65</td>
</tr>
<tr>
<td></td>
<td>1002</td>
<td>$66</td>
</tr>
</tbody>
</table>

- Assembler will convert to Ascii

6. fill
- Fill a certain number of memory locations with a given value.
- Syntax: fill value, count
- Example:
  space_line fill $20, 40
  ; fill 40 bytes with $20 starting from the memory location referred to by the label space_line

7- 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>]</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>J</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>
8. ds.w (define storage word), rmw (reserve memory word)
   - Reserve a number of words
     
     Dbuf ds.w 20 ;Reserves 20 words (or 40 bytes) starting from the current location counter.

9. equ (equate)
   - Assigns a value to a label.
   - Makes programs more readable.
   - Examples:
     
     loop_count equ 50

     Informs the assembler whenever the symbol loop_count is encountered, it should be replaced with the value 50
Example 1: Array of bytes

```
org $800
a1 db $11, $22, $33, $44
a2 dc.b $01
dc.b $02
dc.b $03
a3 rmb 2
```

Example 2: Array of words

```
org $800
a1 dw $11, $22, $33, $44
a2 dc.w $01
dc.w $02
dc.w $03
a3 rmw 2
```

Memory Map:

```
<table>
<thead>
<tr>
<th>800</th>
<th>801</th>
<th>802</th>
<th>803</th>
<th>804</th>
<th>805</th>
<th>806</th>
<th>807</th>
<th>808</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>22</td>
<td>33</td>
<td>44</td>
<td>01</td>
<td>02</td>
<td>03</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

...
Example 3: Two dimensional arrays

```
org $800
a1 db $11, $22, $33, $44
a2 dc.b $01, $05
  dc.b $02, $06
  dc.b $03, $07
a3 rmw 2*2
```

This computation is done by assembler

Later, we will write a code to read one and two dimensional arrays
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>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

Label field
- Labels are symbols defined by the user 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 not ended with “:”
- Example:

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

```
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
```

## 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 and operation field for at least one space
Sample Program

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
- 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
1. **Problem definition:** Identify what should be done

2. **Identify the inputs and outputs**

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

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

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

5. **Program Testing:**
   - Testing for anomalies.
   - Test for the max. and min. values of inputs
   - Enter values that can test all branches
Outline

2.1 Assembly language program structure

2.2 Arithmetic instructions

2.3 Branch and loop instructions

2.4 Shift and rotate instructions

2.5 Boolean logic instructions

2.6 Bit test and manipulate instructions
- **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.

- **Half carry flag (H):** set when there is a carry from the lower four bits to the upper four bits.

- **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. i.e. whenever the carry from the most significant bit and the second most significant bit differs

\[
\begin{align*}
1010 & \ 1010 \\
+ & \ 0101 \ 0101 \\
\hline
1111 & \ 1111
\end{align*}
\]

C = 0, V = 0, Z = 0, N = 1
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7A</td>
<td>+52</td>
<td>2A</td>
</tr>
<tr>
<td>+52</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CC</td>
<td>7C</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>C: 0</td>
<td>C: 0</td>
</tr>
<tr>
<td></td>
<td>V: 1</td>
<td>V: 0</td>
</tr>
<tr>
<td></td>
<td>N: 1</td>
<td>N: 0</td>
</tr>
<tr>
<td></td>
<td>Z: 0</td>
<td>Z: 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AC</th>
<th>+8A</th>
<th>AC</th>
</tr>
</thead>
<tbody>
<tr>
<td>+72</td>
<td></td>
<td>1E</td>
</tr>
<tr>
<td></td>
<td>36</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C: 1</td>
<td>C: 1</td>
</tr>
<tr>
<td></td>
<td>V: 1</td>
<td>V: 0</td>
</tr>
<tr>
<td></td>
<td>N: 0</td>
<td>N: 1</td>
</tr>
<tr>
<td></td>
<td>Z: 0</td>
<td>Z: 0</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7A</td>
<td>8A</td>
<td>5C</td>
</tr>
<tr>
<td>-5C</td>
<td>-5C</td>
<td>-8A</td>
</tr>
<tr>
<td>1E</td>
<td>2E</td>
<td>D2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BA</td>
</tr>
<tr>
<td></td>
<td>C: 0</td>
<td>C: 0</td>
</tr>
<tr>
<td></td>
<td>V: 0</td>
<td>V: 1</td>
</tr>
<tr>
<td></td>
<td>N: 0</td>
<td>N: 0</td>
</tr>
<tr>
<td></td>
<td>Z: 0</td>
<td>Z: 0</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C: 1</td>
<td>C: 1</td>
</tr>
<tr>
<td></td>
<td>V: 1</td>
<td>V: 0</td>
</tr>
<tr>
<td></td>
<td>N: 1</td>
<td>N: 1</td>
</tr>
<tr>
<td></td>
<td>Z: 0</td>
<td>Z: 0</td>
</tr>
</tbody>
</table>
**Overflow**

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

Overflow occurs when two numbers are added or subtracted and the correct result is a number outside the range that can a register hold

---

**Overflow Detection**

1. **Unsigned numbers:** Overflow occurs when \( C = 1 \), C flag can be considered as a bit of the result.
2. **Signed numbers:** Overflow occurs when \( V = 1 \)

Overflow cannot occur when adding numbers of opposite sign why?

If there is an overflow, then the given result is not correct

---

**Signed numbers:** \(-1 + 1 = 0\), no overflow and the result is correct

**Unsigned numbers:** \(255 + 1 = 256\), overflow, 256 needs 9 bits instead of 8, the result is incorrect
**Addition:**

C = 1, the result needs more space than the register width

V = 1, (+ve) + (+ve) = (-ve) or (–ve) + (-ve) = (+ve)

**Subtraction:** A - B

There is no unsigned overflow but there is signed overflow

C = 1, when there is a borrow or B > A

V =1, when

(-ve) - (+ve) = (+ve) this is equivalent to (–ve) + (-ve) = (+ve)
(+ve) - (-ve) = (-ve) this is equivalent to (+ve) + (+ve) = (-ve)

<table>
<thead>
<tr>
<th>Unsigned</th>
<th>Signed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0111 1111</td>
<td>127</td>
</tr>
<tr>
<td>+ 0000 0001</td>
<td>+ 1</td>
</tr>
<tr>
<td>0000 0000</td>
<td>128</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Unsigned</th>
<th>Signed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0110 1011</td>
<td>107</td>
</tr>
<tr>
<td>- 1101 1011</td>
<td>- 219</td>
</tr>
<tr>
<td>1001 0010</td>
<td>- 112</td>
</tr>
</tbody>
</table>
**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 48-bit numbers

- Carry flag is set to 1 when the subtraction operation produces a borrow 1. This borrow should be subtracted from the next subtraction operation.

- Add with carry and sub with borrow instructions enable users to implement multi-precision arithmetic.
**Example:** 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 LSB and proceeds toward MSB.

```
org $1500

; Add and save the least significant two bytes
ldd $1002 ; D ← [$1002, $1003]
add $1006 ; D ← [D] + [$1006, $1007]
std $1012 ; m[$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]
```

Notice there is no instruction for addition with carry for 16 bits.
Example: Write a program to subtract the 4-byte number stored at $1004$-$1007$ from the number stored at $1000$-$1003$ and save the result at $1010$-$1013$.

The subtraction Addition starts from the LSB and proceeds toward MSB.

```
org $1500
; Subtract and save the least significant two bytes
ldd $1002 ; D ← [$1002, $1003]
subd $1006 ; D ← [D] - [$1006, $1007]
std $1012 ; m[$1012, $1013] ← [D]

; Subtract and save the second most significant bytes
ldaa $1001 ; A ← [$1001]
sbca $1005 ; A ← [A] - [$1005] - C
staa $1011 ; $1001 ← [A]

; Add and save the most significant bytes
ldaa $1000 ; A ← [$1000]
sbca $1004 ; A ← [A] - [$1004] - C
staa $1010 ; $1010 ← [A]
```

There is no instruction for subtraction with borrow for 16 bits.

Can loop instruction make the program shorter? Will see later.
Binary-Coded-Decimal (BCD)

- Although computers work internally with binary numbers, the input and output equipment usually uses decimal numbers.

How are decimal values processed?

- **Option 1**
  - Convert decimal to binary on input
  - Operate in binary
  - Convert binary to decimal before output

- **Option 2 (simplifies I/O conversion)**
  - Save the decimal inputs in binary-coded-decimal (BCD) code
  - Operate in binary with adjusting the result of BCD arithmetic after every operation using daa instruction

- One way to convert decimal to binary is called a binary coded decimal (BCD).

- Each digit is encoded by 4 bits that can take values from 0000 to 1001 (9 in decimal)

**Example:**

\[ 25 = 0010 0101 \text{ in BCD} \]

\[ = 0001 1001 \text{ in binary} \]
- Since addition instructions do binary addition, 4 bits can hold a value more than 9 (from 10 to 15).
- To keep the format of BCD after addition, adjustment is needed to the results to insure that each 4 bits can only have at most 9.
- This adjustment can be done by daa (decimal adjustment accumulator A) instruction.

**How daa works:**

1. If one digit of the sum > 9, then subtract 10 from the digit and add 1 to the next 4 bit group this is equivalent to adding $6$.
2. If there is a carry from 4 bits, then add $6$.
   
   why? This carry moves 16 from a 4-bit group and adds 1 to the next 4 bits, but in decimal we should move only 10.

- daa is used immediately after one of the three instructions that leaves their sum in accumulator A (adda, adca, aba)
- It can be used only for BCD add but not subtraction
- Numbers added must be legal BCD numbers to begin with
- H flag can capture the carry from lower nibble to the larger nibble and C flag can capture the carry from the higher nibble.
**Example:** write an instruction sequence to add the BCD numbers stored at memory locations $1000$ and $1001$ and store the sum at $1002$

```
ldaa $1000
adda $1001
daadaa $1002
```

**Conclusion**

```
adda $1001 ;binary addition
; BCD addition
adda $1001
daadaa
```
Numbers at most 9 and no carry from any nibbles, DAA does not do anything.

\[
\begin{align*}
\text{Adjustment: add 66 because a half carry and carry are generated.} \\
\text{Adjustment: add 66 because there is a half carry, and the higher nibble is > 9.}
\end{align*}
\]
Decrementing and incrementing instructions

Add/sub instructions can be used to do increment and decrement, but it is less efficient.

```
ldaa i
adda #1
staa i
```

\[
= \]

```
inc i
```

\(i\) is a memory location

<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) - $01)</td>
</tr>
<tr>
<td>DECA</td>
<td>Decrement A by 1</td>
<td>(A \leftarrow (A) - $01)</td>
</tr>
<tr>
<td>DECB</td>
<td>Decrement B by 1</td>
<td>(B \leftarrow (B) - $01)</td>
</tr>
<tr>
<td>DES</td>
<td>Decrement SP by 1</td>
<td>(SP \leftarrow (SP) - $01)</td>
</tr>
<tr>
<td>DEX</td>
<td>Decrement X by 1</td>
<td>(X \leftarrow (X) - $01)</td>
</tr>
<tr>
<td>DEY</td>
<td>Decrement Y by 1</td>
<td>(Y \leftarrow (Y) - $01)</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) + $01)</td>
</tr>
<tr>
<td>INCA</td>
<td>Increment A by 1</td>
<td>(A \leftarrow (A) + $01)</td>
</tr>
<tr>
<td>INCB</td>
<td>Increment B by 1</td>
<td>(B \leftarrow (B) + $01)</td>
</tr>
<tr>
<td>INS</td>
<td>Increment SP by 1</td>
<td>(SP \leftarrow (SP) + $01)</td>
</tr>
<tr>
<td>INX</td>
<td>Increment X by 1</td>
<td>(X \leftarrow (X) + $01)</td>
</tr>
<tr>
<td>INY</td>
<td>Increment Y by 1</td>
<td>(Y \leftarrow (Y) + $01)</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>
</tr>
</thead>
<tbody>
<tr>
<td>clc</td>
<td>Clear C bit in CCR</td>
<td>$0 \Rightarrow C$</td>
</tr>
<tr>
<td>cli</td>
<td>Clear I bit in CCR</td>
<td>$0 \Rightarrow I$</td>
</tr>
<tr>
<td>clr &lt;opr&gt;</td>
<td>Clear memory</td>
<td>$00 \Rightarrow M$</td>
</tr>
<tr>
<td>clra</td>
<td>Clear A</td>
<td>$00 \Rightarrow A$</td>
</tr>
<tr>
<td>clrB</td>
<td>Clear B</td>
<td>$00 \Rightarrow B$</td>
</tr>
<tr>
<td>clv</td>
<td>Clear V bit in CCR</td>
<td>$0 \Rightarrow V$</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>
</tr>
<tr>
<td>coma</td>
<td>One’s complement A</td>
<td>$FF - (A) \Rightarrow A$ or $(\overline{A}) \Rightarrow A$</td>
</tr>
<tr>
<td>comb</td>
<td>One’s complement B</td>
<td>$FF - (B) \Rightarrow B$ or $(\overline{B}) \Rightarrow B$</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>
</tr>
<tr>
<td>nega</td>
<td>Two’s complement A</td>
<td>$00 - (A) \Rightarrow A$ or $(\overline{A}) + 1 \Rightarrow A$</td>
</tr>
<tr>
<td>negb</td>
<td>Two’s complement B</td>
<td>$00 - (B) \Rightarrow B$ or $(\overline{B}) + 1 \Rightarrow B$</td>
</tr>
</tbody>
</table>

- Clear operation clears the value to 0, used for variable initialization.
- <opr> is a memory location specified using the extended or index (direct or indirect) addressing modes.
- Complement operation replaces the value with its one’s complement.
- Negate operations replace the value with its two’s complement.
## Multiplication and Division instructions

Table 2.1 Summary of HCS12 multiply and divide instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>emul</td>
<td>unsigned 16 by 16 multiply</td>
<td>(D) × (Y) → Y:D</td>
</tr>
<tr>
<td>emuls</td>
<td>signed 16 by 16 multiply</td>
<td>(D) × (Y) → Y:D</td>
</tr>
<tr>
<td>mul</td>
<td>unsigned 8 by 8 multiply</td>
<td>(A) × (B) → A:B</td>
</tr>
<tr>
<td>ediv</td>
<td>unsigned 32 by 16 divide</td>
<td>(Y:D) ÷ (X) quotient → Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>remainder → D</td>
</tr>
<tr>
<td>edivs</td>
<td>signed 32 by 16 divide</td>
<td>(Y:D) ÷ (X) quotient → Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>remainder → D</td>
</tr>
<tr>
<td>fdiv</td>
<td>16 by 16 fractional divide</td>
<td>(D) ÷ (X) → X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>remainder → D</td>
</tr>
<tr>
<td>idiv</td>
<td>unsigned 16 by 16 integer divide</td>
<td>(D) ÷ (X) → X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>remainder → D</td>
</tr>
<tr>
<td>idivs</td>
<td>signed 16 by 16 integer divide</td>
<td>(D) ÷ (X) → X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>remainder → D</td>
</tr>
</tbody>
</table>

- **fdiv**: D should be less than X. The radix point of the quotient is to the left of bit 15.
- **fdiv** assumes the operands are unsigned binary fractions 0.2⁻¹₂⁻²⁻²⁻³.....
**Example:** 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
std $1102 ; store least significant 16 bits
```

**Example:** 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)
```
Conversion of Binary to BCD to ASCII

- A binary number can be converted to BCD format by using repeated division by 10.
- The largest 16-bit binary number is 65,535 which has five decimal digits.
- The first division by 10 generates the least significant digit (in the remainder).
- The ASCII code of a digit can be obtained by adding $30$ to it.
- The ASCII code of 0 is “$30$”, the ASCII code of 1 is 31 and so on.

<table>
<thead>
<tr>
<th>Quotient</th>
<th>Remainder</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345</td>
<td>1234</td>
</tr>
<tr>
<td>1234</td>
<td>123</td>
</tr>
<tr>
<td>123</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

5  Least significant
4
3
2
1  Most significant
Example: Write a program to convert the 16-bit number stored at $1000-$1001 to BCD format and store the result at $1010-$1014. Convert each BCD digit into its ASCII code and store it in one byte.

```plaintext
org $1000
data dc.w 12345   ;data to be tested
org $1010
result ds.b 5     ; reserve bytes to store the result
org $1500
ldd data         ;D = the number to be converted
ldy #result      ;Y = the first address of result
ldx #10           ;X =10
idiv              ;D/X \rightarrow X, R\rightarrow D
addb #$30         ;convert the digit into ASCII code
stab 4,Y           ;save the least significant digit
xgdx
ldx #10
idiv
idiv
addb #$30
```
- If the number is less than 5 digits, we get zeros at left and do unnecessary operations for example: 345 will be 5, 4, 3, 0, 0, 0

- Two improvements: (1) loop can be used to reduce the program and (2) a condition to exit the loop when the quotient = 0
Outline

2.1 Assembly language program structure

2.2 Arithmetic instructions

2.3 Branch and loop instructions

2.4 Shift and rotate instructions

2.5 Boolean logic instructions

2.6 Bit test and manipulate instructions
2.3.1 Branch instructions

1- Unconditional and conditional branches
   - **Unconditional**: Always branch takes place.
   - **Conditional**: branch if a condition is satisfied. A condition is satisfied if certain flags are set. Usually there is a comparison or arithmetic operation to set up the flags before the branch instruction.

2- Short and long branches
   - **Short Branches**: in the range of $80(-128) \sim $7F(+127) bytes. A signed 8 bit offset is added to PC when a condition is met
   - **Long Branches**: in the range of 64KB ($8000(-32,768) to $7FFF(+32,767)). A signed 16-bit offset is added to PC when a condition is met

3- Unsigned and Signed branches
   - **Unsigned branches**: treat the numbers compared previously as unsigned numbers. Use instructions: higher (bhi), higher or same (bhs), lower (blo), and lower and same (bls).
   - **Signed branches**: treat the numbers compared previously as signed numbers. Use instructions: greater (bgt), greater or equal (bge), less (blt), and less and same (ble).
Table 2.2 Summary of short and long branch instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Equation or Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unary Branches</strong></td>
<td></td>
<td></td>
</tr>
<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>
<tr>
<td><strong>Simple Branches</strong></td>
<td></td>
<td></td>
</tr>
<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>bvs rel8 or Ibvs rel16</td>
<td>Branch if overflow set</td>
<td>V = 1</td>
</tr>
<tr>
<td><strong>Unsigned Branches</strong></td>
<td></td>
<td></td>
</tr>
<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>bls rel8 or Ibls rel16</td>
<td>Branch if lower or same</td>
<td>C + Z = 1</td>
</tr>
<tr>
<td><strong>Signed Branches</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bge rel8 or Idge rel16</td>
<td>Branch if greater than or equal</td>
<td>N ⊕ V = 0</td>
</tr>
<tr>
<td>bgt rel8 or Iggt 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.
2.3.2 Compare and Test Instructions

- Condition flags need to be set up before conditional branch instruction are executed.

- The compare and test instructions perform subtraction, set the flags based on the result, and does not store the result. ONLY flags changes.

- Most instructions update the flags automatically so sometimes compare or test instructions are not needed

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>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>cpd &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>cpix &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>

The memory and register does not change

<opr> can be an immediate value, or a memory location that can be specified using immediate, direct, extended, indexed addressing modes

<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>
### 2.3.3 Loop Primitive Instructions

- HCS12 provides a group of instructions that either decrement or increment a loop count to determine if the looping should be continued.

- The range of the branch is from $80$ (-128) to $7F$ (+127).

#### Table 2.5 Summary of loop primitive instructions

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

Note. 1. **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
2.3.4 Bit Condition Branch Instructions

- In some applications, one needs to make branch decisions on the basis of the value of few bits in a memory location.

```
brclr <opr>,msk,rel ;jump takes place when the tested bits are zeros
brset <opr>,msk,rel ;jump takes place when the tested bits are ones
```

**<opr>**: The memory location to be checked and must be specified using either the direct, extended, or index addressing mode.

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

**rel**: The branch offset and is specified in the 8-bit relative mode.

Brclr: does logic AND operation between <opr> and msk and branches if Z = 1, this means the tested bits are zeros

Brset: does logic AND operation between the one’s complement of <opr> and msk and branches if Z =1, this means the tested bits are ones.
How can we check if some bits are zeros?

1 and B = B → put 1 at the bits you test
0 and B = 0 → put 0 at the bits you do not test

The bits I wanna test

<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>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<opr>

AND

This number is zero if B0 and B1 are zeros, otherwise it is not zero

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

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

The branch is taken if the most significant three bits at memory location $66$ are all ones. Notice: $E0 = %1110 0000$

The branch is taken if the most significant bit at the memory location $66$ is zero. Notice: $80 = %1000 0000$
- Loops are used to repeat a sequence of instructions several times.

1. **Endless loop**
   - do a sequence of instructions (S) forever.
   
   Loop: ldaa 1, x+
   adda #$12
   bra Loop

2. **For loops**
   - i is loop counter that can be incremented (or decremented) in each iteration.
   - Sequence S is repeated n2-n1+1 times
   - n2 > n1

   Steps:
   1. Initialize loop counter
   2. Compare the loop counter with the limit n2 (or n1) if it is not equal do the loop otherwise exit
   3. increment (or decrement) the loop and go to step 2
Implementation of $\text{for (i = n1, i <= n2, i++) \{S\}}$

$n1 \text{ equ 1 }$ ; starting index
$n2 \text{ equ 20 }$ ; ending index
$i \text{ ds.b 1 }$ ; $i$ is the loop counter

$\text{movb } \#n1, i$ ; initialize $i$ to $n1$

Loopf: $\text{ldaa } i$ ; check index $i$
$cmpa \ #n2$
$bhi \ \text{Next}$ ; if $i > n2$, exit the loop

... ; performs $S$

... ;

$\text{inc } i$ ; increment loop index
$\text{bra Loopf}$ ; go back to the loop body

Next: ...
Implementation of $\text{for (i = n2, i \geq n1, i--) \{S\}}$

\[\text{n1 equ 1 ; starting index} \]
\[\text{n2 equ 20 ; ending index} \]
\[\text{i ds.b 1 ; i is the loop counter} \]

\[\text{movb \#n2,i ; initialize i to n2} \]

\[\text{Loopf: ldaa i ; check index i} \]
\[\text{cmpa \#n1} \]
\[\text{blo Next ; if i < n1, exit the loop} \]

\[\text{... ; performs S} \]

\[\text{... ;} \]

\[\text{dec i ; decrement loop index} \]
\[\text{bra Loopf ; go back to the loop body} \]

\[\text{Next: ...} \]

Since i is a byte, the max. number of iterations is 256. For more iterations:-

1- use nested loops - outer and inner For loops See next slide.
Or 2- i can be a word. See next slide.
n1 equ 1
n2 equ 6000
i rmb 2

movw #n2,i

ldd i
Loopf: cpd #n1
   blo Next
   ...
   ...
   ldd i
   subd #1
   std i
   bra Loopf

Next: ...

n11 equ 1
n12 equ 20
n21 equ 1
n22 equ 20
i1 ds.b 1
i2 ds.b 1

movb #n12,i1
Loop1: ldaa i1
       cmpa #n11
       blo next1

movb #n22,i2
Loop2: ldaa i2
       cmpa #n21
       blo next2
       ...... ; performs S
       ......
       ......
       dec i2
       bra Loop2

next2: dec i1
       bra Loop1

next1:
For loop using dbeq

up to 65,535 iterations

\[
\begin{align*}
n & \equiv \ 6000 \quad ; \text{number of iterations} \\
\text{ldx} & \ #n+1 \\
\text{Loopf: dbeq} & \ x, \text{next} \\
\text{.........} & \quad ; \text{performs S} \\
\text{.........} & \\
\text{bra} & \ \text{Loopf} \\
\text{next:} & \ \ldots
\end{align*}
\]

up to 256 iterations

\[
\begin{align*}
n & \equiv \ 60 \quad ; \text{number of iterations} \\
\text{ldab} & \ #n+1 \\
\text{Loopf: dbeq} & \ b, \text{next} \\
\text{.........} & \quad ; \text{performs S} \\
\text{.........} & \\
\text{bra} & \ \text{Loopf} \\
\text{next:} & \ \ldots
\end{align*}
\]
**While Loop**

\[
\text{While (condition) } \{ \text{Sequence } S; \} \]

- The condition is evaluated first, if it is false, S will not be executed
- Unlike for loop, the number of iterations may not be known beforehand
- It will repeat until an event happens, e.g., user enter escape character

The update of icount is done by an interrupt service routine (not shown)

\[N \text{ equ 10}\]
\[icount \text{ ds.b } 1\]
\[\text{movb } \#N,icount ; \text{initial value}\]

**Wloop:**
\[\text{ldaa } \#0\]
\[\text{cmpaicount}\]
\[\text{beq Next}\]
\[\ldots \ldots \ldots \ldots \ldots \ldots \ldots ; \text{perform S}\]
\[\ldots \ldots \ldots \ldots \ldots \ldots \ldots \]
\[\text{bra Wloop}\]

**Next:** ...
**Do - While loop**

Do \{ Sequence S; \} While (condition)

- The main difference between while and do-while loops is that do-while loop can execute S at least once because it is executed first and then the condition is evaluated.

---

**Do \{Sequence S;\} While (icount ≠ 0)**

N equ 10
icount ds.b 1
movb #N,icount

Wloop:

\[ \ldots \ldots ; \text{perform S} \]
\[ \ldots \ldots ; \text{"} \]

ldaa #0
cmpa icount
bne Wloop

\[ \ldots \ldots \]

Figure 2.7 The Repeat ... Until looping construct
Other examples:-

Do {Sequence S;} While (m1 == m2)

m1 ds.b 1
m2 ds.b 1
movb #5,m1 ; initial value
movb #5,m2 ; initial value
Wloop:
........ ; perform S
........ ; "
ldaa m1
cmpa m2
beq Wloop
........

I = 1;
Do { Sequence S;
    I++;}
While (I<= 10)

I rmb 2
movw #1,I
Wloop:
        .......... ; perform S
        ldd I
        addd #1
        std I
        cpd #10
        bls Wloop

ldy #10 ;Y is loop counter (I) = 10
Wloop: ...
...
     ...
dbne Y,Wloop ;Y = Y-1
             ;loop if Y ≠ 0
If (I == 1) {Sequence S;}

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

........... ; perform S
........... ; "

end_if:

I ds.b 1
ldaa I ; A = I
cmpa #1
bne else

........... ; perform S1
........... ; "
bra end_if

else:
........... ; perform S2
........... ; "

end_if:
If (I == 1)  
{Sequence S1;}  
Else If (I == 2)  
{Sequence S2;}  
Else If (I == 3)  
{Sequence S3;}  
Else  
{Sequence Se;}
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
  beq Case1
cmpa #2
  beq Case2
cmpa #3
  beq Case3
bra else
Case1:
  .......... ; perform S1
  bra end_case
Case2:
  .......... ; perform S2
  bra end_case
Case3:
  .......... ; perform S3
  bra end_case
else:
  .......... ; perform Se
end_case:
### Signed vs. Unsigned Comparison

**Signed**

- `lda #$01` ;interpret as +1
- `cmpa #$FF` ;interpret as -1
- `bgt label` ;branch taken

**Unsigned**

- `lda #$01` ;interpret as 1
- `cmpa #$FF` ;interpret as 255
- `bhi label` ;branch not taken

- `if (A > 32) go to L1`
- `cmpa #32`
- `bhi L1`

- `if (B < -2) go to L2`
- `cmpb #-2`
- `blt L2`

- `if (MAX <= A) go to L3 (Unsigned)`
- `cmpa MAX`
- `bhs L3`
Example: Write a program to add an array of N 8-bit numbers and store the sum at memory locations $1000$-$1001$.

\[
\begin{align*}
N &\text{ equ 10 } &;\text{Array count} \\
\text{org} &\text{ $1000$ } &;\text{Starting address of the data} \\
s\text{um} &\text{ ds.b 2 } &;\text{2 byte variable to hold result} \\
i &\text{ ds.b 1 } &;\text{i is the array index} \\
\text{array} &\text{ dc.b 1,2,3,4,5} \\
&\text{ dc.b 6,7,8,9,10} \\
\text{org} &\text{ $1500$ } &;\text{starting address of the program} \\
\text{movb} &\text{ #0,i } &;\text{i }\leftarrow\text{ 0} \\
\text{movw} &\text{ #0,sum } &;\text{sum }\leftarrow\text{ 0} \\
\text{Loop:} &\text{ ldab i } &;\text{b }\leftarrow\text{i} \\
&\text{ cmpb }\#N &;\text{is i = N?} \\
&\text{ beq Done} &\text{is i = N?} \\
&\text{ ldx }\#\text{array} &;\text{X = the beginning address of} \\
&\text{ abx} &;\text{the array, X is a pointer} \\
&\text{ ldab }0,X &;\text{B+X}\rightarrow\text{X = the address of the current byte} \\
&\text{ ldy sum } &;\text{y }\leftarrow\text{sum} \\
&\text{ aby sum} &;\text{B+Y }\rightarrow\text{Y (= sum }\leftarrow\text{sum + array[i])} \\
&\text{ sty sum} &;\text{sum }\leftarrow\text{y (= sum + array[i])} \\
&\text{ inc i} & \\
&\text{ bra Loop} &
\end{align*}
\]

Done:

Figure 2.9 Logic flow of example 2.14
A shorter solution

Modify the code to capture signed/unsigned overflow. If there is no overflow, a variable `overflow = 0`, otherwise it is FF

**Hint:** for unsigned overflow, after `aby` instruction, add:

```
bcc no_overflow ; (or “bvc no_overflow” in case of signed overflow)
movb #$FF,overflow
bra done
no_overflow:
```
Example: Write a program to find the maximum element from an array of N 8-bit elements

1- max_value = Array[0]
2- Scan the array from Array[1] to Array [N-1]
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
N equ 20

org $1000 ; starting address of on-chip SRAM
max_val ds.b 1 ; max. value is hold here

org $1500 ; starting address of program
lda array ; a = array[0]
sta max_val ; max_val = a = array[0]
ldx #array+N-1 ; start from the end of the array
ldab #N-1 ; b is loop count i = N - 1

Loop: ldaa max_val ; a = max_val
cmpa 0,x ; compare a and array[i]
bge chk_end ; do not change max_value if it is greater

; an element greater than max_val is found
lda 0,x
sta max_val ; update the array max

chk_end: dex ; move to the next array element
dbne b,Loop ; finish all the comparison yet?

Can you modify this code to find the minimum value?
Can you modify this code to find the minimum and the maximum values?
Example Write a program to compute the number of elements that are divisible by 4 in an array of N 8-bit elements. Notice: a number is divisible by 4 when the least significant two bits equal 0s.

```
N equ 5
org $1000
total ds.b 1
array dc.b 1,2,3,4,5

org $1500
clr total ; initialize total to 0
ldx #array ; use X as the array pointer
ldab #N ; use b as the loop count
loop: brclr 0,x,$03,yes ; check bits 1 and 0
   bra chkend
yes: inc total
chkend: inx
   dbneb,loop
```

Think: Can we reduce the code if we test if the number is not divisible by 4 instead of divisible by 4? Sometimes testing the opposite is helpful.
Example: Write a code fragment showing how to implement the following pseudocode

```c
boolean done = FALSE;
while ( ! done ) {
  ...
  ; at some point
  done = TRUE;
}
```

Solution:

```assembly
TRUE equ 1
FALSE equ 0

org $4000
ldab #FALSE ; b = done
notDone: TBNE b, end
; ...
; At some point,
ldab #TRUE
bra notDone
end:
```
Write a code to read element \((i, j)\) in a two dimensional arrays

Given \(i\), \(j\), \(W\), and \(\text{array}\) (the starting address), how can the address of location \((i, j)\) be computed?

Address of element \((i, j)\) = \(\text{array} + i \times W + j\)

Starting address of row \(i\)
Address of element \((i, j)\) = \(\text{array} + i \times W + j\)

```
Array dc.b 10, 12, 6
dc.b 19, 23, 9
dc.b 1, 21, 60
i dc.b 2
j dc.b 1
W dc.b 3

ldaa i
ldab W
mul ; D = A \times B = i \times w
addd j ; d = i \times w + j
addd #Array ; d = d = i \times w + j
tfr d,X ; X = \text{the address of the element} \ (i, j)
ldaa 0,x ; a = \text{array}[i][j]
```
Write a code to calculate the absolute value of the memory location $1000. Store the result in $1000

```
ldaa $1000
cmpa #00
bge done ; do nothing if [$1000] >=0
; the number is negative
nega
staa $1000
```

Done:
Multi-precision binary addition

```
nbytes equ 5
augend  dc.b 1,2,3,4,5
addend dc.b 1,2,3,4,5
result ds.b #nbytes
result_index ds.b 1

movb  #nbytes,a   ; a is a counter
ldx  #augend+nbytes-1 ; X points at last element in augend
ldy  #addend+nbytes-1 ; Y points at last element in addend
movb #result+nbytes-1,result_index  ; a pointer to result array
clc                                           ; clear carry initially

Loop: ldab 0,x   ; b = current augend byte
      adc b 1,y- ; b = augend element+ addend element, and Y points to the next byte
      stab result_index
      dex              ; Y points to the next byte
dec result_index    ; result_index points to the next byte

dbeq a,Loop

The same code can be used for subtraction but replace “adc b 1,y-” with “sbc b 1,y-”
```
Outline

2.1 Assembly language program structure

2.2 Arithmetic instructions

2.3 Branch and loop instructions

2.4 Shift and rotate instructions

2.5 Boolean logic instructions

2.6 Bit test and manipulate instructions
- Shift and rotate instructions apply to a memory location, accumulators A, B and D.
- A memory operand must be specified using the extended or index (direct and indirect) addressing modes.

1. Logical shift instructions

1.1 Logical shift left

\[
\begin{array}{c}
\text{lsl } <\text{opr}> \quad ; \text{Memory location opr is shifted left one place} \\
\text{lsla} \quad ; \text{Accumulator A is shifted left one place} \\
\text{lslb} \quad ; \text{Accumulator B is shifted left one place}
\end{array}
\]

\[
\begin{array}{c}
\text{lsld} \quad ; 16\text{-bit logical shift left instruction for D}
\end{array}
\]
1.1 Logical shift right

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

\texttt{lsrd}; 16-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
C ← b7 -------------- b0 ← 0

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

C ← b7 -------------- b0 ← 0
A b7 -------------- b0 B

asld ; 16-bit arithmetic shift left instruction logical shift left D

2.2 Arithmetic shift right

C ← b7 -------------- b0 ← 0

asr <opr> ; Memory location opr is shifted right one place
asra ; Accumulator A is shifted right one place
asrb ; Accumulator B is shifted right one place

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

3.2 Rotate right

- **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></td>
<td><strong>Logical Shifts</strong></td>
<td></td>
</tr>
<tr>
<td>LSL&lt;opr&gt;</td>
<td>Logic shift left memory</td>
<td></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></td>
</tr>
<tr>
<td>LSR&lt;opr&gt;</td>
<td>Logic shift right memory</td>
<td></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></td>
</tr>
<tr>
<td></td>
<td><strong>Arithmetic Shifts</strong></td>
<td></td>
</tr>
<tr>
<td>ASL&lt;opr&gt;</td>
<td>Arithmetic shift left memory</td>
<td></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></td>
</tr>
<tr>
<td>ASR&lt;opr&gt;</td>
<td>Arithmetic shift right memory</td>
<td></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></td>
<td><strong>Rotates</strong></td>
<td></td>
</tr>
<tr>
<td>ROL&lt;opr&gt;</td>
<td>Rotate left memory through carry</td>
<td></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></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.

<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.

<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>

**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>
**Example:** Suppose that \([A] = $BE\) and \(C = 1\). Compute the new values of \(A\) and \(C\) after the execution of the instruction \(\text{rora}\).

Figure 2.15a Operation of the instruction \(\text{rora}\)

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

Figure 2.15b Execution result of \(\text{rora}\)
**Example:** Write a program to count the number of 0s in the 16-bit number stored at $1000 -$1001 and save the result in $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
org $1000
 db  $23,$55 ; test data

org $1005
 zero_cnt  rmb  1
 lp_cnt    rmb  1

org $1500
 clr zero_cnt ; initialize the 0s count to 0
 ldaa #16 ; A = 16
 staa lp_cnt ; lp_cnt = A = 16
 ldd $1000 ; place the number in D

Loop:  lsr d ; shift the lsb of D to the C flag
 bcs chkend ; branch if the C flag (= LSB bits) is 1
 inc zero_cnt ; increment 0s count if the lsb is a 0

chkend: dec lp_cnt ; check to see if D is already 0
 bne loop

An application: each bit can be an input from a switch. For example in voting system we need to know the number of approvals (ones) and the number of disapprovals (zeros)
Shift a multi-byte number

- Sometimes we need to shift a number larger than 16 bits, but HCS12 does not have such an instruction.
- Suppose a number of K bytes that are located at locations loc, loc+1,.., loc+k-1, where the most significant byte is stored at loc

<table>
<thead>
<tr>
<th>loc</th>
<th>loc+1</th>
<th>.......</th>
<th>Loc+k-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>msb</td>
<td></td>
<td>.......</td>
<td>lsb</td>
</tr>
</tbody>
</table>

**For shifting right**
1. The bit 7 of each byte will receive the bit 0 of its immediate left byte with the exception of the most significant byte which will receive a 0.
2. Each byte will be shifted to the right by 1 bit. The bit 0 of the least significant byte will be stored in C.

**For shifting left**
1. The bit 0 of each byte will receive the bit 7 of its immediate right byte with the exception of the least significant byte which will receive a 0.
2. Each byte will be shifted to the left by 1 bit. The bit 7 of the most significant byte will be stored in C.
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.

```
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

dbneb,Again
```

Can you change the code to shift the 32-bit number stored at $820-$823 to the left four places?
Multiplication and division using shift

- Shift left multiplies by 2, overflow can occur because the number will increase. C can capture the unsigned overflow and V can capture the signed overflow.
- Arithmetic shift right divides by 2. Sign bit is preserved. Overflow can not occur because division reduces the number.
- To multiply/divide by $2^N$ shift left/right N times
- Much faster than multiply and divide instructions
- Example:

  % 0011 0010 = 50
  Shift left: %0110 0100 = 100
  Shift left: %1100 1000 = 200
  Shift left: %1001 0000 = 220 not correct because there is overflow and C should be a part of the result so % 1 1001 0000 = 400 correct

  %0001 1000 = 24
  Shift right: %0000 1100 = 12
  Shift right: %0000 0110 = 6
Divide each element in an array by 2

org $800
  Len dc.b 3       ; length of array
  array dc.b 200, 10, 150

org $1000
  ldx #array       ; x is array pointer
  clra            ; A is used as a counter

Loop:
  ; array[i] = array[i]/2 and i = i + 1
  asr 1,x+        
  inca            
  cpa Len         
  blt Loop

if op = 1 then operand = operand / \(2^N\) else operand = operand \(\times\) \(2^N\)
overflow = 00 if there is no overflow, else it is FF

org $800
  N equ 2
  operand dc.b 20
  op dc.b 00
  overflow ds.b 1
org $1000
  movb #00,overflow
  ldaa #N     ; a = N
  ldab op     ; sets Z flag
  beq multiply
  div: asr operand ; no overflow
       dbne a,div
       bra done
  multiply: asl operand
            bcc no_overflow
            movb #FF,overflow
            bra done
  no_overflow: dbne a,multiply
  done:
Write a program to convert a 4-digit BCD number stored at memory locations $900$ and $901$ into its 16-bit binary equivalent stored at locations $902$ and $903$.

- 12 BCD should be converted to 0001100 in binary.
- If locations $900$ and $901$ contain 20 BCD and 48 BCD, the 4-digit BCD number is 2048 BCD. This number should be converted to $0800$. $08$ and $00$ are stored in $902$ and $903$, respectively.

\[
\text{Binary} = (((\text{BCD}_3 \times 10) + \text{BCD}_2) \times 10 + \text{BCD}_1) \times 10 + \text{BCD}_0
\]

- Notice: BCD$_i$ is multiplied by 10 $i$ times which coincides with decimal numbers.

- Each byte has two BCD numbers, how can we separate them?

<table>
<thead>
<tr>
<th>AND</th>
<th>BCD2</th>
<th>BCD1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>1111</td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{array}{c}
\text{Shift right 4 times} \\
\hline
0000 & \text{BCD2} \\
\hline
0000 & \text{BCD1}
\end{array}
\]

Mask using and.
org $800

; process thousands digits
Convw: ldaa $900 ; get the most significant two digits of BCD number
    Lsra ; move thousands position to lower nibble
    lsra
    lsra
    lsra
    ldab #10
mul ; thousands digit x 10 operation 1 in previous slide
    std $902 ; store the result in $902 and $903

; process hundreds digit
    ldab $900 ; reload the most significant two digits
    andb #$0F ; mask off the upper nibble
    clra ; unsigned 16 bits extension a = 0 and B = BCD2 so D = BCD2
    addd $902 ;operation 2 in previous slide
    ldy #10
    emul ;Y:D=DxY operation 3 in previous slide multiply total by 10
    std $902
; Process tens digit

ldab $901   ; b = least significant two digits of BCD number
lsrb
lsrb
lsrb
lsrb
clra
addd $902   ; operation 4, add tens digit to running total
ldy #10
emul        ; operation 5
std $902

; process ones digit

ldab $901   ; reload the least significant two digits
andb #$0F   ; mask off the upper nibble
clra        ; unsigned 16 bits extension a = 0 and B = BCD2 so D = BCD2
addd $902   ; operation 6
std $902
Outline

2.1 Assembly language program structure

2.2 Arithmetic instructions

2.3 Branch and loop instructions

2.4 Shift and rotate instructions

2.5 Boolean logic instructions

2.6 Bit test and manipulate instructions
- Changing a few bits are often done in I/O applications.
- Boolean logic operation can be used to change a few I/O port pins easily.
- 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></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(A) • (M) ⇒ A</td>
</tr>
<tr>
<td>andb &lt;opr&gt;</td>
<td>AND B with memory</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(B) • (M) ⇒ B</td>
</tr>
<tr>
<td>andcc &lt;opr&gt;</td>
<td>AND CCR with memory (clear CCR bits)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(CCR) • (M) ⇒ CCR</td>
</tr>
<tr>
<td>eora &lt;opr&gt;</td>
<td>Exclusive OR A with memory</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(A) ⊕ (M) ⇒ A</td>
</tr>
<tr>
<td>eorb &lt;opr&gt;</td>
<td>Exclusive OR B with memory</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(B) ⊕ (M) ⇒ B</td>
</tr>
<tr>
<td>oraa &lt;opr&gt;</td>
<td>OR A with memory</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(A) + (M) ⇒ A</td>
</tr>
<tr>
<td>orab &lt;opr&gt;</td>
<td>OR B with memory</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(B) + (M) ⇒ B</td>
</tr>
<tr>
<td>orcc &lt;opr&gt;</td>
<td>OR CCR with memory (set CCR bits)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(CCR) + (M) ⇒ 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

<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>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\[
\text{AND} \rightarrow \begin{array}{cccccccc}
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
\end{array}
\]

Thanks to: \[ \text{Bi AND 0} = 0 \]
\[ \text{Bi AND 1} = \text{Bi} \]

"OR" is used to set one or few bits

Ex. Set the first 4 bits in register B

<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>

\[
\text{OR} \rightarrow \begin{array}{cccccccc}
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
\end{array}
\]

Thanks to: \[ \text{Bi OR 0} = \text{Bi} \]
\[ \text{Bi OR 1} = 1 \]
“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

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

Thanks to:

Bi XOR 0 = Bi

Bi XOR 1 = Bi'

Bi' = the inversion of Bi

Exclusive or

1010 1101 (data)
0000 1111 (mask)
1010 0010 (result)

Bits preserved

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

Clear bits

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

Selected bits
ldaa $56
andra #$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 M
oraa #%00011000
staa M

Force bits 3,4 of M to be 1's

ldaa M
andra #%11100111
staa M

Force bits 3,4 of M to be 0's

ldaa M
andra #%01100000
cmpa #%01100000
beq bothones

Test if both bits 5,6 of M are 1's

1 0 1 0 1 0 1 0
AND 0 1 0 1 0 1 0 1
———
0 0 0 0 0 0 0 0

C unaffected
N = 0
V unaffected
Z = 1
Outline

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

2.6 Bit test and manipulate instructions
**Table 2.8: Bit Test and Manipulation Instructions**

<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) \cdot (\overline{mm}) \Rightarrow M)</td>
</tr>
<tr>
<td>bita &lt;opr&gt;¹</td>
<td>Bit test A</td>
<td>((A) \cdot (M))</td>
</tr>
<tr>
<td>bitb &lt;opr&gt;¹</td>
<td>Bit test B</td>
<td>((B) \cdot (M))</td>
</tr>
<tr>
<td>bset &lt;opr&gt;²,msk8</td>
<td>Set bits in memory</td>
<td>((M) + (\overline{mm}) \Rightarrow M)</td>
</tr>
</tbody>
</table>

<opr>¹: can be specified using all relative addressing modes for bita and bitb

<opr>²: can be specified using direct, extended, and indexed (exclude indirect) addressing modes

msk8: 8-bit value

- A mask value is used to test or change the value of individual bits in an accumulator or in a memory location
- bita and bitb are used to test bits **without changing the value of either operand**. They do AND operation and update flags but do not store the result.

**bclr 0,X,$81** ; clear the most significant and least significant bits of the memory location pointed by register X ($81=\%10000001)
bita #$44 ; Test bit 6 and bit two of register A and updates Z and N flags and V flag is cleared ($44=%01000100)

bitb #$22 ; Test bit five and bit 1 of register b and updates Z and N flags and V flag is cleared.

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

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

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

Test if either bit 5 or bit 6 of M is 1

lda M  
bita %#01100000  
bne eitherones  

Bita does M AND %01100000. This masks off (zeros) all bits except bits 5 and 6
- The execution time of an instruction is measured in E cycles.

- There are many applications that require the generation of time delays, e.g., to generate a square wave signal, output 1 to an output pin, then wait for some time, then output 0 and wait for some time, repeat.

- The creation of a time delay involves two steps:
  1. Select a sequence of instructions that takes a certain amount of time to execute.
  2. Repeat the selected instruction sequence for an appropriate number of times.

- The instruction sequence on the next slide takes 40 E cycles to execute. By repeating this instruction sequence certain number of times, any time delay can be created.

- Assume that E frequency is 8 MHz and hence its clock period is 125 ns. Therefore, 40 E cycles instruction sequence will take 40 x 125 ns = 5 µs to execute.
ldx #20000  ;2 E cycles
Loop:
  psha  ; 2 E cycles
  pula  ; 3 E cycles
  psha
  pula
  psha
  pula
  psha
  pula
  psha
  pula
  psha
  pula
  nop  ; 1 E cycle
  nop  ; 1 E cycle
  dbne  x,loop
  ; 3 E cycles whether the condition is satisfied or not

Total delay ≈ 5 µs x initial value of X

If we want a 100 ms delay, the inner loop should be repeated 100 ms/5µs times = 20,000 times.

Min. delay
If X = 0, the min. delay ≈ 5µs

Max. delay
If X = 65535, the max. delay ≈ 5µs x 65,535 = 327.675 ms

If a longer delay is needed, add an outer loop
- However, the time delays calculated in previous slide are not accurate because

1- Interrupt processing (if there is) adds delay.

2- We neglected the overhead to set up the loop count (ldx #20000) that is 2 E cycles and the overhead of dbne x,loop instruction 3 cycles.

- A more accurate number of E cycles required in previous slides is

\[
\text{Delay} = (2 \text{ (for ldx #20000)} + X \text{ (for inner loop)} + 3 \text{ (for dbne x,loop)}) * 125 \text{ ns} = (2 + 43 X) * 125 \text{ ns}
\]

- If \( X = 20,000 \) then more accurate delay = 107.5ms and approximated delay = 5μs * 20,000 = 100ms

- One way to reduce the overhead is to reduce the number of loops in \( X \)

- To get a more accurate delay: if the delay 100ms in previous equation, \( X = 18604.6 \). Since \( X \) cannot be fraction, the delay = 99.99675 ms and 100.002125 ms when \( X = 18604 \) and 18605 respectively.

- Software delay can be used in applications that require non-precise time delays. For applications that require accurate delays use timers - the topic of a coming chapter
Example:

- **Time the following instructions**
  
  ```
  ldx #1234 ; 2E cycles
  Loop:     psha ; 2E cycles
            pula ; 3E cycles
            psha ; 2E cycles
            pula ; 3E cycles
            dex ; 1E cycle
            bne Loop ; 3E cycles → branch taken
                        ; 1E cycle → branch not taken
  ```

- This requires a slightly different count! \( X = 1234 \)
  
  \[
  2 + ( (X-1)*14 + (1)*12) \text{ E Cycles} = 17,276 \text{ E Cycles}
  \]

You have to consider that “bne Loop” takes 3 cycles if the branch is taken and one when it is not taken
- Execution times of each instruction can be obtained from Instruction set file on the course web page (http://iweb.tntech.edu/mmahmoud/teaching_files/undergrad/ECE3120/InstructionSet.pdf)
- Number of letters in the column “Access Detail” of Appendix A indicates the number of E cycles that a specific instruction takes to execute that particular instruction.
- For example access detail column of PULA instruction contains three letters UFO which indicates that the PULA instruction takes 3 E cycles to complete.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Access Detail</th>
<th>E-Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>PULA</td>
<td>(M_{sp}) \Rightarrow A; (sp) + 1 \Rightarrow SP</td>
<td>UFO</td>
<td>32</td>
</tr>
<tr>
<td>PULB</td>
<td>(M_{sp}) \Rightarrow B; (sp) + 1 \Rightarrow SP</td>
<td>UFO</td>
<td>33</td>
</tr>
<tr>
<td>PULC</td>
<td>(M_{sp}) \Rightarrow CCR; (sp) + 1 \Rightarrow SP</td>
<td>UFO</td>
<td>38</td>
</tr>
<tr>
<td>PULD</td>
<td>(M_{sp}; M_{sp+1}) \Rightarrow AB; (sp) + 2 \Rightarrow SP</td>
<td>UFO</td>
<td>3A</td>
</tr>
<tr>
<td>PULX</td>
<td>(M_{sp}; M_{sp+1}) \Rightarrow XH; Xl; (sp) + 2 \Rightarrow SP</td>
<td>UFO</td>
<td>30</td>
</tr>
<tr>
<td>PULY</td>
<td>(M_{sp}; M_{sp+1}) \Rightarrow YH; Yl; (sp) + 2 \Rightarrow SP</td>
<td>UFO</td>
<td>31</td>
</tr>
</tbody>
</table>
Hand Assembly

- To see how the assembler translates assembly instructions into machine code, see the instruction set, e.g., Ldaa $10 → 96 10
- From the instruction set, you can know:
  - The number of bytes required for each instruction.
  - What addressing modes used in each instruction
  - What flags are affected by each instruction

Carefully check this file to be familiar to it. I will distribute it in the midterm and final exams.

![Table showing source form, operation, address mode, machine coding, and access detail for various instructions.]

- We can use direct mode since the address is < 256
- The machine code is $96 $10
- Each letter represents one clock cycle – this takes 3 clock cycles
Thank You!

Questions

Mohamed Mahmoud