EXAMPLE 10

OBJECTIVE

This example has the following objectives:

- Introduce 8-bit binary numbers
- Introduce connection between 8-bit binary numbers and 2-digit hex numbers.
- Introduce 8-bit binary arithmetic
- Introduce the half carry (bit 3 carry) concept
- Introduce 8-bit binary logic
- Introduce masking

PROGRAM

Use Ex10.asm code to do the following:

i) Construct the numbers zero, one, two, three, four, five, six, seven, eight, nine, ten, eleven,
twelve, thirteen, fourteen, fifteen, sixteen, seventeen using the addition command ADDA. Represent these numbers in decimal, binary, and hex using the display options of accA.

ii) Perform $3 + $2 = $5 in binary and hex using LDAA, LDAB, ABA

iii) Perform $13 - $6 = $7 in binary and hex using LDAA, LDAB, SBA

iv) Perform $0f + $01 = $10 in binary and hex using LDAA, LDAB, ABA (half carry, or bit 3 carry)

v) Perform $10 - $01 = $0f in binary and hex using LDAA, SUBA

vi) Perform $ff + $01 = $00 in binary and hex using LDAA, ADDA (MSB carry)

vii) Perform $00 - $01 = $ff in binary and hex using LDAA, SUBA

viii) 1 AND 0 = 0 using LDAA, ANDA

ix) 1 OR 0 = 1 using LDAA, ORAA

x) 1 XOR 0 = 1 using LDAA, EORA

xi) 1 OR 1 = 1 using LDAA, ORAA

xii) 1 XOR 1 = 0 using LDAA, EORA

xiii) %11111111 NAND %00000000 = %11111111 using LDAA, ANDA, COMA

xiv) %00000000 NOR %11111111 = %00000000 using LDAA, ORAA, COMA

xv) %10101110 AND %00000010 = %00000010 using LDAA, ANDA (masking)

(Note: the NAND and NOR instructions do not exist in the MC6811 instruction set, but can be obtained by using the COM instruction after the AND and OR instructions.)
CODING AND EXECUTION
Open THRSim11. Maximize THRSim11 window. Close the Commands window. Open and assemble Ex10.asm. Tile the windows, set the breakpoint, and reset the registers. Reset accA and accB by typing zero into them. Set the display option of accA to binary. Your screen should look like this:
i) **SECTION: REPRESENT NUMBERS ONE, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, ELEVEN, TWELVE, THIRTEEN, FOURTEEN, FIFTEEN, SIXTEEN, SEVENTEEN IN DECIMAL, BINARY, AND HEX**

Use the Step button to step through this section of your program. After first step your screen looks like this:

![Image of a computer screen showing assembly code and status register]

During this operation, the number zero has been loaded into accA. Your Assembly instruction used the decimal 1, and the simulator converted it to binary, `%00000001`, per your display setting. The number one stored in accA is represented using all the 8 bits, though the seven highest bits (preceding bits) are zero, i.e., `%00000001`. This rules always applies when binary numbers are represented in this simulator.

Change the display option of accA to hex. Your screen will look like this:

![Image of a computer screen showing assembly code and status register]

Step through the rest of the operations, incrementing each time accA by one. At each step, observe the binary representation, then switch to hex, then to decimal, and back to binary. When incrementing, observe how the binary bits either increment from 0 to 1, or increment from 1 to 0 with a binary carry to the left. When you get to the equivalent of number ten, your binary screen should look like this:

![Image of a computer screen showing assembly code and status register]
Change the display option of accA to HEX, your screen looks like this:

Continue with the incrementation process until you reach fifteen. Note that during this process, only the lower four binary bits and the corresponding hex LSD are being affected. The higher four binary bits and the hex MSD are not being affected. When you reached fifteen, your binary screen should look like this:
Change the display option of accA to HEX, your screen looks like this:

Note that all the lower bits (bit 0, bit 1, bit 2, bit 3) have been filled with 1’s, while all the higher bits (bit 4, bit 5, bit 6, bit 7) are still filled with 0’s. You have reached the point where further incrementation will produce a carry from bit 3 into bit 4. When this happens, the lower bit will be returned to 0, while a 1 will appear in the least significant position of the upper bits. This even also corresponds to a hex carry, i.e. from $0f$ to $10$. Your binary screen looks like this:
Change the display option of accA to HEX, your screen looks like this:

![Image of THRSim11 screen showing the display option change]

The decimal equivalent of the number in accA at this point is 16. Further incrementation affects again only the lower bits, until a bit 3 carry takes again place.

At the end of this section of your program, the binary screen looks like this:

![Image of THRSim11 screen showing the end of the program]
Change the display option of accA to HEX, your screen looks like this:

![Screen capture showing the display option set to HEX]

**ii) SECTION: PERFORM 3 + 2 = 5 IN BINARY AND HEX USING LDAA, LDAB, ABA**

Place PC at the beginning of section ii) in the code. Turn display of accA and accB to binary. Step through this section of the code until the numbers three and two have been loaded into accA and accB, respectively. They will show as binary, as illustrated by the screen capture below:

![Screen capture showing the display set to binary]

Perform the last step in this section. The addition of accA and accB is performed, and the result is shown in accA. Observe that a carry from bit 1 into bit 2 has happened. The screen should look like this:
Repeat this section with the display of accA and accB turned to hex. Observe that, in hex, no carry is apparent. The corresponding screen captures are:
iii) **SECTION: PERFORM 13 – 6 =7 IN BINARY AND HEX USING LDAA, LDAB, SBA**

Place PC at the beginning of section iii) in the code. Turn display of accA and accB to binary. Step through this section of the code until the number six has been subtracted from number thirteen. After the two numbers have been loaded into accA and accB, respectively, the screen looks like this:

Perform the subtraction operation. Note the borrow that was enacted from bit 3 into bit 2 into bit 1 to permit the operation to take place. After subtraction, your screen looks like this:
Repeat this section with the display of accA and accB turned to hex. Observe that, in hex, no borrow is apparent. The screen corresponding captures are:
iv) **SECTION: PERFORM $0F + $01 = $10 IN BINARY AND HEX USING LDAA, INCA (HALF CARRY OR BIT 3 CARRY)**

Reset accA and accB. Place PC at the beginning of section iv) in the code. Turn display of accA to binary. Step through this section of the code until the number $0f$ has been loaded into accA. The screen looks like this:

Perform the incrementation. Note the carry that takes place from bit 3 into bit 4. Your screen looks like this:
Repeat this section with the display of accA turned to hex. Observe that, in hex, a carry is also apparent. The corresponding screen captures are:
v) **SECTION: PERFORM $10 -$01 = $0F IN BINARY AND HEX USING LDAA, SUBA**

Reset accA and accB. Place PC at the beginning of section v) in the code. Turn display of accA to binary. Step through this section of the code until the number $10 has been loaded into accA. The screen looks like this:

Perform the decrementation. Note the borrow that takes place from bit 4 into bit 3. Your screen looks like this:
Repeat this section with the display of accA turned to hex. Observe that, in hex, a borrow is also apparent. The corresponding screen captures are:
vi) **SECTION: PERFORM $FF + $01 = $00 IN BINARY AND HEX USING LDAA, ADDA (MSB CARRY)**

Reset accA and accB. Place PC at the beginning of section vi) in the code. Turn display of accA to binary. Step through this section of the code until the number $ff has been loaded into accA. The screen looks like this:

Perform the increase. Note the lost carry that takes place from bit 7. Your screen looks like this:

Repeat this section with the display of accA turned to hex. Observe that, in hex, a lost carry is also apparent. The corresponding screen captures are:
vii) **SECTION: PERFORM $00 - $01 = $FF IN BINARY AND HEX USING LDAA, SUBA**

Reset accA and accB. Place PC at the beginning of section vii) in the code. Turn display of accA to binary. Step through this section of the code until the number $00 has been loaded into accA. The screen looks like this:

Perform the decrementation. Note the free borrow that takes place from into bit 7. Your screen looks like this:
Repeat this section with the display of accA turned to hex. Observe that, in hex, a free borrow is also apparent. The corresponding screen captures are:
viii) **SECTION: 1 AND 0 = 0 using LDAA, ANDA**

Reset accA and accB. Place PC at the beginning of section viii) in the code. Turn display of accA to decimal. Step through this section of the code until the number 1 has been loaded into accA. The screen looks like this:

![Screen Image 1](image1.png)

Step again. The number 1 is AND-ed with 0. The result is 0. Your screen looks like this:

![Screen Image 2](image2.png)
ix) **SECTION: 1 OR 0 = 1 USING LDAA, ORAA**

Reset accA and accB. Place PC at the beginning of section ix) in the code. Leave display of accA to decimal. Step through this section of the code until the number 1 has been loaded into accA. The screen looks like this:

![Screen capture showing the code execution steps](image)

Step again. The number 1 is OR-ed with 0. The result is 1. Your screen looks like this:
x) **SECTION: 1 XOR 0 = 1 USING LDAA, EORA**

Place PC at the beginning of section x) in the code. Leave display of accA to decimal. Step through this section of the code until the number 1 has been loaded into accA. The screen looks like this:

Step again. The number 1 is XOR-ed with 0. The result is 1. Your screen looks like this:
xi) **SECTION: 1 OR 1 = 1 using LDAA, ORAA**

Place PC at the beginning of section xi) in the code. Leave display of accA to decimal. Step through this section of the code until the number 1 has been loaded into accA. The screen looks like this:

Step again. The number 1 is OR-ed with 1. The result is 1. Your screen looks like this:
xii) **SECTION: 1 XOR 1 = 0 USING LDAA, EORA**

Place PC at the beginning of section xii) in the code. Leave display of accA to decimal. Step through this section of the code until the number 1 has been loaded into accA. The screen looks like this:

![Image of a computer simulation interface showing the code execution]

Step again. The number 1 is OR-ed with 1. The result is 0. Your screen looks like this:

![Image of a computer simulation interface showing the result of OR operation]

- XOR 1 = 0
- LDAA, EORA
xiii) SECTION: %11111111 NAND %00000000 = %11111111 USING LDAA, ANDA, COMA

Place PC at the beginning of section xiii) in the code. Turn display of accA to binary. Step through this section of the code until the number %11111111 has been loaded into accA. The screen looks like this:

Step again. The number %11111111 is AND-ed with %00000000. The result is %00000000. Your screen looks like this:
Step again. We get the Boolean inversion of the result using the COMA (complement of accA) opcode. The result is now %11111111 (note that COMA inverses all the bits). Your screen looks like this:

xiv) SECTION: %00000000 NOR %11111111 = %00000000 USING LDAA, ORAA, COMA

Place PC at the beginning of section xiv) in the code. Turn display of accA to binary. Step through this section of the code until the number %00000000 has been loaded into accA. The screen looks like this:
Step again. The number %00000000 is OR-ed with %11111111. The result is %11111111. Your screen looks like this:

Step again. We get the Boolean inversion of the result using the COMA (complement of accA) opcode. The result is now %00000000 (note that COMA inverts all the bits). Your screen looks like this:
xv) SECTION: %10101110 AND %00000010 = %00000010 using LDAA, ANDA (MASKING)

Place PC at the beginning of section xv) in the code. Leave display of accA to binary. Step through this section of the code until the number %10101110 has been loaded into accA. The screen looks like this:

Step again. The number %10101110 is AND-ed with %00000010. The result is %00000010. Your screen looks like this:

Notice the effect of the mask. Out of all the bits that were set in accA, only that bit (bit 1) that coincides with the mask has remained. Mask is useful for checking the status of selected bits (e.g., flags).
WHAT YOU HAVE LEARNED
In this example, you have learned:

- The 8-bit binary numbers
- The connection between 8-bit binary numbers and 2-digit hex numbers.
- The 8-bit binary arithmetic
- The half carry (bit 3 carry) concept
- The 1-bit and 8-bit binary logic
- The fact that logic operations are performed bit-by-bit
- Boolean inversion using the complement opcode (e.g., COMA).
- Masking, i.e., the selection of bits using masks
- New words and notations: 8-bit binary arithmetic, 8-bit binary logic; complement opcode; masking, flags.