Programming in machine code on the Amiga

A. Forness & N. A. Holten

ISSUE 3

Content

DATA SCHOOL
Postbox 62
North Engen 18
2980 Kokkedal

Phone 49 18 00 77
Postgiro 7 24 23 44

LOGIC OPERATIONS

In this chapter, we must look at binary algebra (primarily plus and minus) and so-called logic operations (AND, OR, NOT and XOR). This is something you will need very often. It may seem a bit complicated when you see it the first time - but it is not difficult.

When we take the decimal number system and have the number 9 and add 1 so we reset the position reserved for the one's and increase the position for the tens about 1.

To: 9 + 1 = 10

In the decimal number system we have 10 numbers (from 0 to 9) available when we count. In binary the number system we have only 2 numbers: 0 and 1 But it works in exactly the same way. Let us look at the following setup:

1 + 1 = 10

Using the decimal number system, it does not make sense at all, but it is absolutely correct in the binary number system. We have been through it in the past (see issue I), but now we must go a little more in depth.

The following calculation (binary) will be used as explanatory example in this section.

Example 1:

```
      %1000 1001      (CARRY = 0)
+     %1010 1010
      ----------
=     %0011 0011      (CARRY = 1)
      ==========
```

Now we review this calculation literally bit by bit. The bit called LSB (the far right) in the upper number (%1000 1001) is to be added to the LSB in number below (%1010 1010). So if we add the two bits 1 and 0 together. It will be 1.

It works exactly as in the decimal system: 1 + 0 = 1. So we take the next bit next to the left of the LSB in both numbers: 0 and 1 Again it works just in the decimal system: 0 +1 = 1

The next couple of bits are two zeros. Also serves as the decimal system: 0 +0 = 0

The next two bits are interesting. No the principles is applied as we already mentioned before, when you go from 9 to 10 in the decimal number system. We now have only two numbers that can be used when we count in binary – "1" and "0" so when two "1" are added, we get a "0" and keep one "1" in mind.

So it is appropriate that we began this section with: 1 + 1 = 10.

The “1” did pass into the next position to the left so that it becomes in the current position the digit becomes “0”. We keep the “1” in mind and use this so-called “overflow” with the next operation at the next position.

The next two bits are two “0”. As shown above, is 0 + 0 = 0 for all number systems. Now there is also a “1” in our mind from the previous operation - so the total sum at this position becomes:

0 + 0 + 1 = 1

The next bit pair is “0” and “1”. As before the result will be 1 (no thought).

In the next position, there are two zeros. Conclusion here is 0 (no thought)

The bit-pair far left the MSBs, says two ones. The result is thus: 1 + 1 = 10 Here we write a “0” in this position in the result and additionally set the CARRY-flag to 1 indicating that there is a “1” in mind – a so called carry (or overflow). We therefore would need 9 bits to give the correct answer.

Task 0301: Add the binary numbers %10100111 and%10001101 together.

Now you should be ready to try the subtraction of binary numbers. This is done in the same way as by subtraction of decimals numbers. The rules are here:

Binary	Rule 1:	0 - 0 = 0
	Rule 2:	1 - 0 = 1
	Rule 3:	1 - 1 = 0
	Rule 4:	0 - 1 =? (The figure is negative)

Rules 1, 2 and 3 should be obvious. It is just as in decimal, as you see. We can not imagine that you have problems with it. We will return to rule 4 in a subsequent section within a chapter that will deal with negative numbers.

AND - OR - NOT – XOR

Now we need to look at the DATA logic. Honorable mathematicians designed this logic before there were computers, but this logic was not fully appreciated before computers were invented. Now we can not imagine doing this logic.

This logic - that we start at a moment - are used for instance when we must manipulate a bit of a register without disturbing the other bits; or if we want to reset a bit in a register without changing the other bits. We must now explain this in a little more detail.

We begin with AND.

The word AND means AND. If you have two bytes, and you have to "and" them, a specific operation is performed where each bit in the two binary numbers are taken into account.

Example No 2:

```
        %1011 0101
AND     %1001 1110
        ----------
=       %1001 0100
        =========
```

We said that meant AND, and now you get to see how the whole works. Bits far left in the two binary numbers in the example 2 are two ones. The logic says that if you “AND” two “1” (1 AND 1) then the result will be “1” If you have something else – like “1” AND “0” or “0” AND “1”, then the answer is “0”. If you compare the bits of the two binary numbers, you can see that each time you “AND” two ones, then the answer is “1” - in all other cases the answer is zero.

You can set a table up against it. Such a table for logical operations is called a truth-table.

Truth-table for AND:

```
0 AND 1 = 0
0 AND 0 = 0
1 AND 0 = 0
1 AND 1 = 1
```

As you see, you get only 1 in reply if you “And” two bitS and both are set 1 AND 1. In all other cases as I said, you get zero. This feature is very good when you need to reset bits in a register in which different bits have a different meaning.

For example, if you want to reset bit number 3 in a register (the fourth bit from right - 3, 2, 1, 0), you “AND” the register with the following binary number: %1111 0111 – if the registers contains: %0110 1011 it becomes %0110 0011 – bit no. 3 will be reset. It becomes a little more manageable if we stack up the numbers over each other:

```
REGISTER contains:    %0110 1011
We “AND” with:        %1111 0111
                      ----------
Result:               %0110 0011
                      ==========
```

Here one sees clearly that the fourth bit from the right (it is the number 3) is reset. We do not know and care what value the other bits in the register have. It is enough that you know that bit 3 will be reset to “0” - all other bits stay untouched. The logical function AND will be used frequently in programming on Amiga.

Task 0302: Perform an AND at %10110110 and %10001111.

The next logical function is called OR. This works in that way that when two binary numbers are “OR”-ed and only one of the two bits has value “1” (at least one or both are true), then the answer is “1”.

Truth table for OR:

```
0 OR 0 = 0
1 OR 0 = 1
0 OR 1 = 1
1 OR 1 = 1
```

In the truth table for OR you can see that if one of the two bits - or both are “1”, then the result of the two bits will be: “1”

With this example we use the “OR” operation on two binary numbers. Study carefully and compare it with the truth table for AND.

```
      %1010 1010
OR    %1100 0101
      ----------
=     %1110 1111
      ==========
```

The “AND” and “OR” logical operations have opposite effects. With the “OR” the corresponding bits in the register are set to “1”. Let us take the same example as in the explanation of the “AND” operation. The register is assumed to contain %0110 1011. After applying the “AND” operation the result is the value %0110 0011. Now we will use the “OR” operation which sets bit 3 (the fourth bit from LSB) to the value “1”.

Register contains:	%0110 0011
We perform an OR with:	%0000 1000

Result:	%0110 1011
	=========

The fourth bit is again “1”, and we are back where we started. This is also a very useful operation which you will use many times.

Task 0303: Perform an OR of %01010101 and %10011001.

Now we come to the logical operator NOT. We appreciate the short truth table:

Truth table for NOT (NOT):

NOT 0 = 1
NOT 1 = 0

NOT “0” must of course be “1”. In the same way, NOT “1” is equal to zero. With This logical operation performs a so-called inversion of all bits. If one writes: NOT %1010 1010 the result is: %0101 0101. It cannot get easier! To the “NOT” operation we will come back in the context with readings of the keyboard on the Amiga.

And then there's the last logical operation XOR. It can be translated with EXCLUSIVE OR. We first look at truth table:

0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0

From the table we can see that the result is only “1” when one bit is “1” performing an XOR. We show here an example of an XOR of two bytes:

REGISTER contains:	%1010 1010
We perform an XOR with:	%1101 1011

Result:	%0111 0001
	=========

One more time: When two bits are “XOR”-ed, then the result will be “1” (bit is set) if only one bit is set - and only then. The result of an XOR is “0”, if both bits are “0” or both bits are “1”. This logical operation is seldom used, but we will need it in a later chapter.

MACHINE CODE PROGRAMMING

So we throw ourselves into the machine code again. We make a small program that loads the content of the first 16 memory addresses of the Amiga. Then we put them somewhere else in the machine memory in a buffer, which we have reserved ourselves.

We show two examples of the same program. The first is a bit clumsy but for clarity's sake (anyway correctly working), and second is slightly more advanced: shorter, faster and smarter.

Here is the first version:

```
1     move.l      #16, d0
2     move.l      #$00, a0
3     lea.l       buffer, a1
4     loop:
5     move.b      (a0), d1
6     add.l       #1, a0
7     move.b      d1, (a1)
8     add.l       #1, a1
9     sub.l       #1, d0
10    cmp.l       #0, d0
11    BNE         loop
12    RTS
13
14    buffer:
15    blk.b 16.0
```

Line 1: Move the decimal number 16 into D0. This register is used as a counter in this routine. We need that the loop should be executed a certain number of times (here: 16 times). The character "#" indicates that the following number is a constant and not an address. In line 6 for instance: add the following constant number (1); in line 9: subtract following constant number (1), and in line 10: compare the following constant number (0) with the register D0.

Line 2: Moves the constant number of $00 into the address register A0. Note that longword is used for this operation which will reset all bits (32bit) of the address register A0. This address register is used as a pointer (the pointer, points at address $000000): the address that we will read from.

Line 3: Loads the effective address of our buffer (see line 14) into A1. In other words: We defined a part of Amiga memory at line 15 to be uses as a buffer for our data. This part of memory starts at the address (after our program) where the label "buffer" is located. It therefore represents the beginning of our buffer). The address register A1 is used as a pointer where we want to store our data. Please note that we do not change anything - we only read. The data is simply copied into our buffer.

Line 4: The label “loop”.

Line 5: Copies the byte located at the address in address register A0 into the data register Dl. The round brackets around A0 mean that not the content of the register is to copied, but the byte the register points to. This way of address register use is called: REGISTER INDIRECT (indirect addressing) and is used very often in MC on the Amiga.

Line 6: Increasing the address in A0 by “1”. This leads to A0 pointing to the next byte, we will read.

Line 7: Moves the byte from D1, to the location the address register A1 points to (which is our buffer). View comments LINIE 5

Line 8: Increase the address in A1 by “1”. This leads to A1 pointing to the next free byte in our buffer we can write to.

Line 9: Subtract the constant number “1” from D0. This is our counter, which is reduced by one each time we copy our byte into our buffer - a total of 16 times.

Line 10: Check the content of D0 if it is “0” CMP means "compare" and is used here to see if we have already copied 16 bytes (this is the case when D0 is “0”).

Line 11: If D0 is not equal to “0”, the program jumps back to our label "loop" (line 4). If D0 is equal to “0”, the loop was already executed 16 times and the program will not jump back. Instead the program provides with the next instruction.

Line 12: RETURN FROM SUBROUTINE (or quit), and give control back to the calling instance (here: K-Seka).

Line 14: The label that identifies the beginning of our own buffer.

Line 15: Here we have declared 16 bytes for our buffer. We could also have written:

blk.l 4.0 or blk.w 8.0.

It would be the same result. BLK stands for "block" and 4.0 (and 8.0) shows the number of longwords (or number of words), we want to reserve as a buffer. The number "0" indicates the pattern the free memory is filled with so that we get an empty buffer (so old data from the past should be cleared). See also next section.

Let us look at some new instructions:

CMP (compare)
BNE, BEQ, BHL, BLO (different "branch" commands)
BLK, DC (BLOCK and DECLARE commands)

We will first review the status register before we go through these instructions.

STATUS REGISTER

	MSB				LSB
CONTENTS:	X	N	Z	V	C

This is part of the so-called STATUS REGISTER of our CPU (MC68000). We will not read or write directly into this register. You do not know what bit numbers the various “flags” have. Everything you need to know is the flags that are set what they used for. The term FLAG is just another word for a bit which indicates a certain condition or event.

The flags of the status register:

C - carry flag (indicator when an arithmetic carry or borrow has been generated)
V - overflow flag (warning, the figure was too large)
Z - zero flag (something has been zero)
N - negative flag (the number is negative)
X - extend flag (special purpose)

When flag is “1”, the condition is true. Imagine a long-jump competition. The referee will raise a red flag every time there is a violation - and white if the jump is valid.

The same principle applies to the flags in STATUS REGISTER. They tell you for example when a figure is negative (N-flag) when the result was zero (Z-flag) when a number was too large (C-flag and V-flag) and so on. We explain the use of each flag in more detail below.

In the next example, the C-flag is “1” because the sum of the two numbers (200 and 100) exceeds the number that can be represented by a BYTE (255).

```
move.b      #200, D0
add.b       #100, D0
RTS
```

As you know with 8 bit (a byte) one can only represent numbers from 0 to 255. In the above example we have 200 in D0, and add 100. It should therefore be 300 in our byte which is impossible. The CARRY flag is set (to “1”) to tell us that the number was too large.

```
          11001000     (200 decimal)
+         01100100     (100 decimal)
          ------------
= 1        00101100    (300 decimal)
          =======
```

As you see we have to take a ninth bit in order to obtain the correct result. This is the bit (far left in the result), which lands in the CARRY-flag (Carry-bit). If the result is 255 or less the C-flag is cleared instead.

The OVERFLOW-flag (V-flag) is very similar to the C-flag. The difference is that it is used as the "ninth bit" in 2-compliments-number or SIGNed number. More details on this in a later chapter.

The ZERO-flag (Z-flag) is used often and in many different contexts. Each time an operation results in a zero result is this flag is set. In the first example in this issue are in line 10: cmp.l #0, d0. This line could be removed because as soon as the line 9 leading to zero in the data register d0 the Z-flag is set automatically. For example, the instruction:

```
cmp.l   #100, D0
```

Take the following number (100) and compare it with what is in the data register d0. If D0 contains 100 the Z-flag is set.

The NEGATIVE-flag (N-flag) is set to show that the result is negative (minus). We will deal with this later.

The EXTEND-flag (X-flag) is being used in rotation (and shifts) or change of bits. The flag is also used in complex addition or multiplication-routines and functions as a CARRY flag. We come back to this flag later.

The CMP (compare) instruction to is used to compare numbers. In order to have any benefit it may be combined with a BRANCH. We take some examples:

```
        move.l      #100, D0
        cmp.l       #50, D0                 (next line is one of the following):
```

```
BEQ   number is equals (BRANCH if EQUAL)
BNE   number, not equal with (BRANCH if NOT EQUAL)
BHI   number is higher (BRANCH if HIGHER)
BLO   number is lower (BRANCH if LOWER)
```

BEQ-instruction (Branch if Equal) will not be performed because the register D0 (containing 100) does not equal 50.

BNE-instruction (Branch if Not Equal) performs a jump, because D0, is not equal to 50.

BHI-instruction (Branch if higher) performs a jump because D0 is "higher" than 50. Here is the number 50 used as a reference. You must be clear about the "awkward" instruction for it becomes true, since D0 is greater than 50.

BLO-instruction (Branch on lower) does not perform a jump, because D0 is not "lower" than 50.

It is very important that you understand this well. See also the example on line 11. We could have written in BHI instead of BNE.
Now we look at the BLK and DC instruction. BLK-instructions (Block) is used to reserve an area in memory in which we can store data. BLK is no instruction. Let us look how to use it.

```
1       blk.b   10,0
2       blk.b   $10,0
3       blk.l   100,0
4       blk.b   $2800,255
5       blk.w   $2800,255
```

Line 1: Reserves 10 bytes and resets them all.

Such as: $00, $00, $00, $00, $00, $00, $00, $00, $00, $00

Line 2: Reserves $10 ($10 = 16) bytes and resets them.
Such as: $00, $00, $00, $00, $00, $00, $00, $00, $00, $00, $00, $00, $00, $00, $00, $00

Line 3: Reserved 100 longwords (in total 400 bytes) and resets them all.

Such as: $00000000, $00000000, $00000000, $00000000... (100 times)

Line 4: Reserved $2800 bytes (decimal = 10,240) and set all bytes to 255 ($FF).

Such as: $FF, $FF, $FF, $FF, $FF, $FF, $FF, $FF, $FF, $FF, $FF, $FF... (10240 times)

Line 5: Reserved $2800 words and sets all words to 255 ($00FF).

Such as: $00FF, $00FF, $00FF, $00FF, $00FF, $00FF, $00FF... (10,240 times)

DC has the same function as BLK, but only for a byte, a word or longword.

```
1       dc.b    0
2       dc.b    255,0,10
3       dc.b    "Amiga"
4       dc.w    $100
5       dc.l    123456
```

Line 1: Reserves a byte and sets it to 0.

Such as: $00

Line 2: It is possible to have multiple numbers on the same line. So here are three bytes are reserved and set to 255 ($FF), 0 and 10 (= $0A).

Such as: $FF, $00, $0A

Line 3: This is a bit special. You probably know that a character can be represented using a byte. Once characters are stored in computers, they are stored as numbers. We have previously stated that letter “A” is stored as number 65 ($41). At the end of the Amiga manual is a table showing the numbers that represent the character. This table is called an ASCII table. The abbreviation stands for: American Standard Code for Information Interchange, translated. This data is standard to store characters in all computers today. The defined characters can be read as hexadecimal numbers.

Such as: $41, $4D, $49, $47, $41

Line 4: Reserves a word and allow set it to $100 (256).

Such as: $0100

Line 5: Reserves a longword and sets it to 123456 (= $1E240).
Such as: $ 0001E240

Did you understand all of it? If you have the slightest doubt, please read it again. It is extremely important that you understand it. It must be one of the cornerstones of your knowledge of programming.

Here comes the second version of our program:

```
1     move.l    #15, d0
2     move.l    #$00, a0
3     lea.l     buffer, a1
4     loop:
5     move.b    (a0)+, (al)+
6     dbra      d0, loop
7     RTS
8
9     buffer:
10    blk.b     16,0
```

Line 1: Moves the constant number of 15 into D0. Here, D0 is used again as a counter.

Line 2: Moves the constant number of $00 into A0 (A0 points to address $000000)
Line 3: loads the effective address of our buffer into A1.
Line 4: The label “loop”.
Line 5: Replacing lines 5,6,7 and 8 in the second example.

We explain this instructions in detail below.

Line 6: Replace lines 9, 10 and 11 in the second example, Again, this instruction will be explained below.

Line 7: Exits the program.

Line 9: The label to our buffer.

Line 10: Reserves 16 bytes and set them to “0”.

The kind of move-instructions you see in line 5 is used very often, so let us explain it first: You have learned that if you put brackets on the address register as in line 5 in the first example, that then the content is used the register points to. This new version works the same way but has an extra finesse. View in this example:

```
move.b      (A0)+, D0
```

It performs the same as:

```
move.b      (A0), D0
add.l       #l, A0
```

The number, located in A0 is used as the address. The instruction fetches the byte which is located at the address and copies it into the data register D0. Then it automatically adds “1” to A0 because there is a “plus sign” after parenthesis (post increment). Had there been "move.w" instead for "move.b" it would have added “2” to A0 (because a word consists of two bytes). Had there been "move.l" instead of "move.b" it would have added 4 to A0 (because a longword consists of four bytes).

The instruction `move.b (A0)+, (A1)+` will then:

1: Getting the byte A0 point to and stores it in the address A1 points to.
2: Increase the address in A0 about “1” (we work with the size bytes).
3: Increase the address in A1 about “1” (we work with the size bytes).

You will use this kind of way very often. Especially when you examine data tables, and when you move around data in Amiga's memory. It is therefore very important that you learn these things.

The next instruction we must have a look at is DBRA.:

```
        move.l      #9, D0
loop:
        dbra        D0, loop
        RTS
```

The example performs the same as:

```
        move.l      #10, d0
loop:
        sub.l       #l, d0
        cmp.l       #0, d0
        BNE         loop
        RTS
```

What does DBRA (Decrement BRanch Always) exactly do:

1: Subtract 1 from the register (in this case D0).
2: Compare with “-1”.
3: If it is larger “-1” jump back to loop.

(How can a negative number be represented by a byte, word or a longword? We’ll come back to this topic later. At the moment do not worry about this problem).

When D0 is “-1” the CPU will continue at the next instruction. Notice that DBRA always compares with “-1”, so we must use a number in the register (here D0), which is “1” less than the number of times we want the loop be executed.

COPPER AND BITPLANES

Now we throw ourselves on the great topic about bitplanes. We explain more about the copper and it will cause some necessary changes back and forth between the subjects.

It is worthwhile to first read through the whole section again to get an overview of the topic and then read it all again in detail.

We begin by explaining a little about the Amiga’s hardware registers. The register of current interest is named "DMACON" which is an abbreviation of "DMA CONTROL". In fact the DMACON is a special word (a 2 bytes address). Hardware registers have nothing to do with the registers of the MC68000, e.g., D0, A0 or others. The hardware registers lie in a completely different part of the machine, namely the custom chips Agnus, Paula and Denise.

The hardware registers have an address. The DMACON's address is $DFF096. All hardware register addresses starting with $dffxxx. We call $DFF000 the base-address for the hardware registers.

KEEP IN MIND! All hardware registers are 16 bits wide, representing a word.

It is important to note that the hardware registers are constructed so that either it can only be written to or only be read from. It can only be written to the color registers. If you try to read them, you will be returned either zero or some other arbitrary value.

The Joystick registers are from the second type, it can only be read from. Writing to them has no meaning. (You can not ask your Joystick smoking and travel! REMARK: of course you can but it makes no sense like this automatic google-translation from Danish to English ;-) Some registers have two addresses - an address to write to and an address to read from. That is the case with the register (DMACON) we must now examine. If you must write DMACON you use the address $DFF096, but if you must read from it, use the address $DFF002.

DMACON is configured as follows (Remember: 1 word = 16 bit):

$DFF096 WRITE - $DFF002 READ

BIT NO.	FUNCTION	ACCESS
15	set / clr	W
14	blitter busy	R
13	blitter zero	R
12	-	-
11	-	-
10	blitter nasty	R / W
9	DMA enable	R / W
8	bitplane DMA	R / W
7	copper DMA	R / W
6	blitter DMA	R / W
5	sprites DMA	R / W
4	disk DMA	R / W
3	audio channel 3 DMA	R / W
2	audio channel 2 DMA	R / W
1	audio channel 1 DMA	R / W
0	audio channel 0 DMA	R / W

As you see we have chosen use the English names. There are two reasons: First, you can easily refer to (and understand) specialized literature in English when you know the English expression. And second translations are often quite difficult because there sometimes exist similar words in Danish with a different meaning.

Notice that in the last column (ACCESS) indicates whether the bit can be read ("R" for READ) or the bit can be written ("W" for WRITE).

KEEP IN MIND! If a DMA must be OFF (or reset), set it to “0”. If a DMA rather be ON (or activated), set ot to “1”

bit 0-3: audio DMA from 0 to 3 on / off

bit 4: disk-DMA on / off

bit 5: sprite-DMA on / off

bit 6: blitter-DMA on / off

bit 7: copper-DMA on / off

bit 8: bitplane-DMA on / off

bit 9: This bit called DMA ENABLE and must be set (1) to be able to use DMA at all (comparable to a main switch). If you wish for any reason to turn off all DMA, then reset the bit. You should also be aware of disk- and sprite-DMA when you turn off the DMA, then only these two DMA channels will be activated when you turn DMA on again.

bit 10: Blitter nasty. More about this bit in issue 6 and 7

bit 11-12: not used.

bit 13: Blitter zero. More about this bit in issue 6 and 7

bit 14: Blitter busy. More about this bit in issue 6 and 7

bit 15: SET/CLEAR. This will be explained below.

The register DMACON is a bit special when it comes to writing. If you need to activate copper-DMA, then you must put these data into the register at address $dff096:

bit 7 and 15 must be "1", the rest must be "0".
Hexadecimal: $8080,
Binary: %1000 0000 1000 0000.
(If you have forgotten conversion between hexadecimal and binary, read the chapter again.)

When COPPER is to be turned off bit 7 must be "1" the rest must be "0".
Hexadecimal: $0080,
Binary: %0000 0000 1000 0000.

When you turn off for example the bitplane-DMA and sprite-DMA, bit 15 must be 0, bit 5 and bit 8 should be “1” and the rest be “0”
Hexadecimal: $0120
Binary % 0000 0001 0010 0000

As an instruction:

move.w #$0120, $DFF096

If you now want to switch on the same DMA channels again, you change just bit 15 to 1 So:

move.w #$8120, $DFF096 (or move.w #%1000000100100000, $DFF096)

We would like to explain it again. In fact, there are 4 registers (8 if you count both READ and WRITE register), which operates in this manner and it is therefore important that you completely understand this topic: If you put bit 15 to "1" and shows which bits are to be affected, then the Amiga itself ensure that the bit you selected are set to "1", also. If you set bit No. 15 to "0", it will enable you to choose the bit to be set to “0”. One more time - but in other words: all the bits you set to "1" get the same value as 15 bit. All other bits (which are "0") remain unchanged.

Simple and easy, right?

KEEP IN MIND!

If you need to turn on a DMA channel and turn off another DMA simultaneously, you must write to register twice – once to turn off, e.g., disk-DMA and once to turn on, e.g., blitter-DMA.

KEY TO COPPER-INSTRUCTIONS:

Study following configurations of the two words as a complete copper-instruction consists of:

First Word:

BIT NO:	MOVE	WAIT
15	A15	V7
14	A14	V6
13	A13	V5
12	A12	V4
11	A11	V3
10	A10	V2
9	A9	V1
8	A8	V0
7	A7	H8
6	A6	H7
5	A5	H6
4	A4	H5
3	A3	H4
2	A2	H3
1	A1	H2
0	0	1

and the Second Word:

BIT NO:	MOVE	WAIT
15	D15	BFD
14	D14	MV6
13	D13	MV5
12	D12	MV4
11	D11	MV3
10	D10	MV2
9	D9	MV1
8	D8	MV0
7	D7	MH8
6	D6	MH7
5	D5	MH6
4	D4	MH5
3	D3	MH4
2	D2	MH3
1	D1	MH2
0	D0	0

A Copper-instruction is always 4 bytes long (2 words). We tend to divide it into two parts, as shown here:

```
dc.w        $5001, $FFFE        (dc.w = declare/define word)
```

Let's go start with the WAIT instruction:

In our explanation we will use "W" to describe the vertical hold. With it the copper is told to wait until the electron beam has reached the specified screen-line before the copper retrieves the next instruction. "HH" is used to describe the horizontal hold of the copper. With it the copper is told to wait until the electron beam has reached a certain horizontal point on the screen-line until the copper retrieves the next instruction.

VVHH, $FFFE

```
VV = $ 00    to    $FF
HH = $01     to    DF    (only odd numbers can be used)
```

We will use the value $01 as a horizontal position in all WAIT instructions. (That is the point furthest to the left on the line.) Waiting for horizontal positions greater than $01 will be explained in issue 12.

When it comes to the vertical position of the following applies:

The upper white line you see in the example of your Workbench screen is no line 0 It is usually line $2C (44 decimal). This means that the electron beam actually begins 44 lines further up than the top line you see. It will again say that the bottom line on Workbench screen becomes number 256 + 44 = 300 (Workbench screen is 256 lines high). Under these 300 lines are 13 additional lines - which you partially cannot see - so the total size of a full-screen Amiga is 313 lines. The copper has a total waiting area from line 0 and ending at line 312

The second word of the wait -instruction is rarely needed to change. This part will be explained later. We come to use the same value all the time, namely: $FFFE

EXPLANATION OF MOVE instructions

MOVE-instruction can only write data to hardware registers in the range $DFF000 to $DFF200.

```
Example:    dc.w    $180, $0FFF
```

The first word of the move-instruction must contain the address. The second word of contains the data you want to copy to that address.

The address calculated as follows: $0180 + $DFF000 = $DFF180. This address must be an equal number - next address will be $DFF182 (a distance of a WORD). If you write this (an odd- address):

```
dc.w    $0181, $0FFF
```

The copper will interpret it as a wait-instruction instead. The copper uses bit No. 0 of the first word to distinguish between a wait- and a move-instruction.

The third copper-instruction is called SKIP. This instruction you will virtually never need, so we will not waste time on it now. It will be explained in a later issue.

Solutions to the example in the issue II (page 17):

Line 1: Turn off bitplane, copper- and sprite-DMA.

Line 2: Loads the effective address of the copper-list into Al.

Line 3: Moves the address of the copper list (which is in A1) into $DFF080. We need our own copper-list in this register so the copper knows where our copper list is located. As you see we use in this longword line when we copy the address to the register, although we have said that you can only use word when you write address to hardware registers. This is an exception. In essence, the copper-pointer has two addresses which are located side by side:

$DFF080 (HI) and
$DFF082 (LO)

The names HI and LO stands for the upper part and the lower part of the address. As you know a longword is the same as two words. Bit 31-16 are thus written to $DFF080 and represent the 16 highest (most "valuable") bits in the complete address and bit 15-0 are written to $DFF082 and represent the 16 lowest (less "valuable") bits in the complete address.

Line 5: Activation of the copper-DMA. This instruction starts our own copper-list.

Line 6: The label “wait”.

Line 7: This instruction, we have not reviewed yet. What it does is to check bit 6 (btst = BitTeST) in address $BFE001. If bit is “0” it will enable the Z-flag to “1” and if this bit is “1”, the Z-flag is set to “0”. When you press the left mouse button the bit No. 6 of $BFE001 is set to “0”.

Line 8: This branch (branch not equal) checks if the Z-lag is “0” and if it is, you have not pressed the left mouse button and the program jumps back to the label "wait". Program lines 7 and 8 are a method to wait until you press the mouse.

Line 10: Turn off COPPER-DMA

Line 11: Moves the contents of address $000004 into A6 (lines 11 to 13 will be explained in issue X. One can briefly say that they reactivate the Workbench and the old copper list).

Line 12: Moves the content A6+156 point to into A1.

Line 13: Moves the content A1+38 point to into the copper-pointer.

Line 14: Restores the old DMA settings again.

Line 15: Exits the program.

Line 17: Label to the copper-list.

Line 18: WAIT ($01, $90)

Line 19: MOVE $0F00 -> $DFF180 red

Line 20: WAIT ($01, $A0)

Line 21: MOVE $0FFF-> $DFF180 white

Line 22: WAIT ($01, $A4)

Line 23: MOVE $000F -> $DFF180 blue

Line 24: WAIT ($01, $AA)

Line 25: MOVE $0FFF -> $DFF180 white

Line 26: WAIT ($01, $AE)

Line 27: MOVE $0F00 -> $DFF180 red

Line 28: WAIT ($01, $BE)

Line 29: MOVE $0000 -> $DFF180 black

Much more and more detail - will be explained in Issue IV.

THE BITPLANE SYSTEM OF THE AMIGA

We begin by setting up the screen:

```
1       move.w      #$01a0, $dff096
2
3       move.w      #$1200, $dffl00
4       move.w      #0, $dff102
5       move.w      #0, $dff104
6       move.w      #0, $dff108
7       move.w      #0, $dff10a
8
9       move.w      #$2c81, $dff08e
10      move.w      #$f4c1, $dff090
11      move.w      #$38c1, $dff090
12
13      move.w      #$0038, $dff092
14      move.w      #$00d0, $dffD94
15
16      lea.l       copper, A1
17      move.l      A1, $dff080
18
19      move.w      #$8180, $dff096
20
21      wait:
22      btst        #6, $bfe001
23      BNE         wait
24
25      move.w      #$0080, $dff096
26
27      move.l      $4, A6
28      move.l      156(A6), A1
29      move.l      38(A1), $dff080
30
31      move.w      #$80a0, $dff096
32
33      RTS
34
35 copper:
36      dc.w        $2c01, $fffe
37      dc.w        $0100, $1200
38
39      dc.w        $00e0, $0000
40      dc.w        $00e2, $0000
41
42      dc.w        $0180, $0000
43      dc.w        $0182, $0ff0
44
45      dc.w        $ffdf, $fffe
46
47      dc.w        $2c01, $fffe
48
49      dc.w        $0100, $0200
```

```
50
51      dc.w            $ffff, $fffe
```

Here is a brief explanation of each instruction. In issue IV it will be explained in more detail. Take it not too serious, if you do not understand everything in the first attempt.

Line 1: Close bitplane-, copper- and sprite-DMA

Line 3: Sets lores (low resolution 320 * 256) and one bitplane that are 2 colors.

Line 4: Sets scroll value to “0”

Line 5: Sets bitplane-priority to “0”

Line 6: Sets the modulo for odd bitplanes to “0”

Line 7: Sets the modulo for even bitplanes to “0”

Line 9: Sets the upper left corner of the screen to position: Y (vertical) = $2C and X (horizontal) = $81

Line 10: Set the bottom right corner of the screen to the position: Y (vertical) = $F4, X (horizontal) = $C1.

Line 11: Adds $38 to Y-position, i.e. $fe + $38 = $12C. X-position is the same.

Line 13: Sets data-fetch-start to $0038

Line 14: Sets data-fetch-stop to $00d0.

Line 16: Loading copper list's address into Al.

Line 17: Moves the address of the copper-list in Al into the copper-pointer.

Line 19: Activates bitplane- and copper-DMA again (not sprite-DMA)

Line 21: The label “wait”.

Line 22: Check if the left mouse button pressed.

Line 23: If mouse button is not pressed, then branch back to the label "wait".

Line 25: Turns off copper-DMA

Line 27: Copies the longword that is located at address $000004 into register A6.

Line 28: Copies the longword A6+156 point to, into register A1.

Line 29: Copies the longword A1+38 points to, into the copper-pointer ($DFF080).

Line 31: Opens copper and sprite-DMA again.

Line 33: Exits the program and returns to the calling instance (e.g. K-Seka)

Line 35: Label for our copper-list.

Line 36: WAIT ($01, $2C)

Line 37: MOVE $1200 -> $dff100

Line 39: MOVE $0000 -> $dff0E0

Line 40: MOVE $0000 -> $dff0E2

Line 42: MOVE $0000 -> $dff180 (black background)

Line 43: MOVE $0FF0 -> $dff182 (yellow foreground)

Line 45: WAIT ($DF, $FF). This causes the machine to use PAL with 256 lines (in the U.S. another system (NTSC) is used where the Amiga has only 200 lines).

Line 47: WAIT ($01, $2C).

Line 49: MOVE $0200 -> $dff100

Line 51: The end of the copper-list. Restart the list at the next vertical blank.

SOLUTIONS FOR TASKS IN ISSUE II

Task 0201: 24 in total because there are 8 sprite-, 4 audio-, 4 blitter-, 1 disk-, 6 bitplane- and 1 copper-DMA.

Task 0202: A subroutine is a routine called during a program that performs a specific task.

Task 0203: A word has two bytes which equals 16 bit.

Task 0204: A longword has 4 bytes which equals 32 bit.

Task 0205: The MC68000 has 8 address registers which are called A0, A1, A2, A3, A4, A5, A6 and A7.

Task 0206: In MC68000 has 8 data registers which are called D0, D1, D2, D3, D4, D5, D6 and D7.

Task 0207: It moves the longword from address $10 points to into the address register A3.

Task 0208: MSB means “most significant bit” and indicates the highest value-bit in a bit group. LSB means “least significant bit”, and indicates the lowest value-bit in a bit group.

Task 0209: The copper has 3 instructions: MOVE, WAIT and SKIP.

Task 0210: The video beam draws the screen line by line – it is generated by the electronic gun in the monitor.

COMMENTS ON ISSUE III

This was a long issue! There is much to examine. It is not that easy, but with little patience you will come far. When one writes a course which covers such a broad topic as the Amiga, it happens that one has to jump between the topics and their explanations. Should there be something you did not understand or which you think is somewhat messy, please wait until you receive issue IV, it explains much of what is perhaps a little "foggy" at the moment. It is very important that you use K-Seka (or any other assembler) and practice. Write all examples and try to change them. If the program locks, then try again. It is incredible how much one can learn from one's mistakes. Whatever you find out - do not give up!

Sincerely,
DATA SCHOOL

Carsten Nordenhof