Atari SFP-004 by example

Atari SFP-004 is an FPU extension for the Mega ST. The integration of the optional FPU in the Mega STE is technically equivalent. There is very little documentation available on programming the FPU in the SFP-004, so the following shall serve as an example of programming. This example does not replace the documentation of the FPU. Please refer to the following literature, which is also used in this example: [1], [2].

DIFFERENCES TO 68020/68030

It is important to first consider the difference to the FPU integration in a 68020/68030-based system, such as the Atari TT. There, the programmer uses an FPU instruction in the code, a line-F opcode, where the first nibble is $F. The CPU and FPU then communicate with each other over the bus to coordinate the mathematical operation, any data transfer, error handling, etc. This all happens invisibly to the programmer.

However, the 68000 in a Mega ST/STE does not have a coprocessor interface. For this reason, the interface – integrated in hardware on the 68020/68030 – must be emulated in software, i.e., by the programmer.

THE INTERFACE OF THE SFP-004

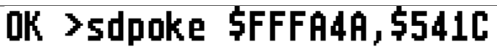
The bus interface of the FPU is mapped into the address space between $FFFA40 and $FFFA60 in the form of various registers. Details are described in [2]. The most important registers are:

$FFFA40: Response, 16 Bit:	The status of the FPU can be read here.
$FFFA4A: Command, 16 Bit:	The command to the FPU is written to this address.
$FFFA50: Operand, 32 Bit:	Transfer of operands (i.e. data) between CPU and FPU.

AN EXAMPLE

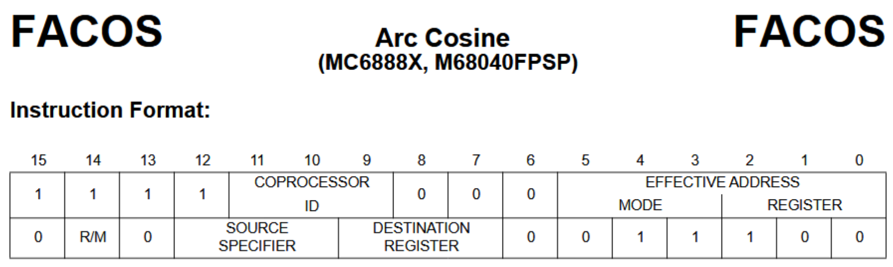
The programming is explained in the following using the – arbitrarily chosen – example of the implementation of the arc cosine function for the SFP-004 in [3]. The assembler source code is shown as well as an interactive execution of the individual steps in GFA Basic.

```
lea     0xfffa50,%a0
movew   #0x541c,%a0@(comm)
```



```
OK >sdpoke $FFFA4A,$541C
```

Loading the command into the command register. The FPU command is always encoded in the *second* word of the corresponding Line F opcode.[1] Therefore [4] can be used to decode the command:



FACOS — **Arc Cosine** **(MC6888X, M68040FPSP)** — **FACOS**

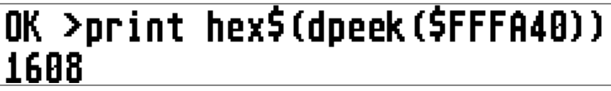
Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	COPROCESSOR ID			0	0	0	EFFECTIVE ADDRESS MODE			EFFECTIVE ADDRESS REGISTER		
0	R/M	0	SOURCE SPECIFIER			DESTINATION REGISTER			0	0	1	1	1	0	0

[1] The first word of the opcodes contains information that is important for the CPU (68020/68030), e.g. the source or destination of data transfers, i.e., the *effective address.*

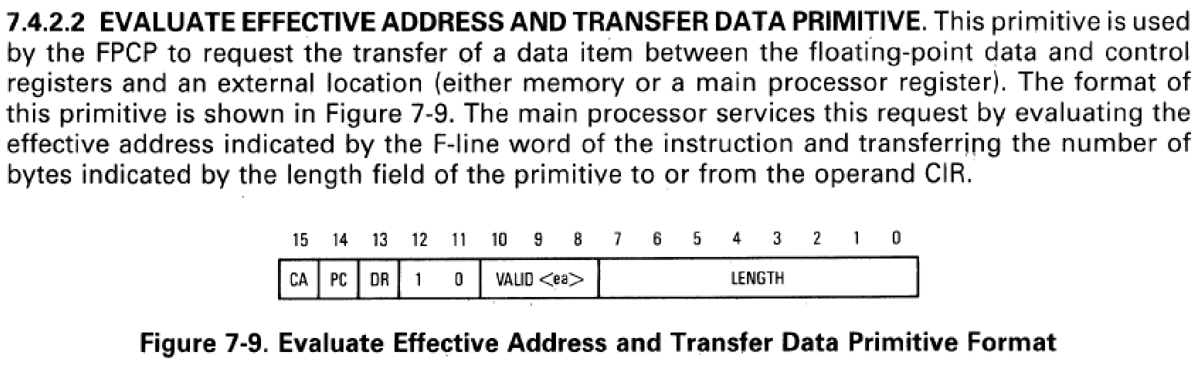
Specifically, the $541C command is thus the arc cosine, with the source being data from the CPU (R/M = 1) in double precision real format (Source Specifier = 101). The result is to be stored in FPU register FP0 (Destination Register = 000).

```
cmpiw   #0x8900,%a0@(resp)
```



```
OK >print hex$(dpeek($FFFA40))
1608
```

The FPU starts processing a command at the earliest after the response register has been read for the first time. Although the command in the example is a comparison (CMPI), the result of the comparison is not used. For example, a possible error message from the FPU would not be intercepted by this code. The code assumes to receive the expected response, in this case $1608, which according to [1] is the *Evaluate Effective Address and Transfer Data Primitive*:



7.4.2.2 EVALUATE EFFECTIVE ADDRESS AND TRANSFER DATA PRIMITIVE. This primitive is used by the FPCP to request the transfer of a data item between the floating-point data and control registers and an external location (either memory or a main processor register). The format of this primitive is shown in Figure 7-9. The main processor services this request by evaluating the effective address indicated by the F-line word of the instruction and transferring the number of bytes indicated by the length field of the primitive to or from the operand CIR.

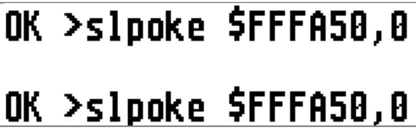
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	DR	1	0	VALID <ea>			LENGTH							

Figure 7-9. Evaluate Effective Address and Transfer Data Primitive Format

This would instruct a 68020/68030 to transfer the data specified in the FPU instruction, for example, from memory. Without explaining all the fields here, it is still apparent that the FPU expects a transfer from CPU to FPU (DR = 0) with a length of 8 bytes.

This is the operand *x* of the arc cosine function acos(*x*) to be calculated, which is 64 bits (8 bytes) long as a double precision floating point value.

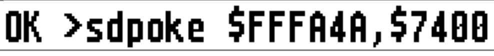
```
movel   %a7@(4),%a0@
movel   %a7@(8),%a0@
```



```
OK >slpoke $FFFA50,0

OK >slpoke $FFFA50,0
```

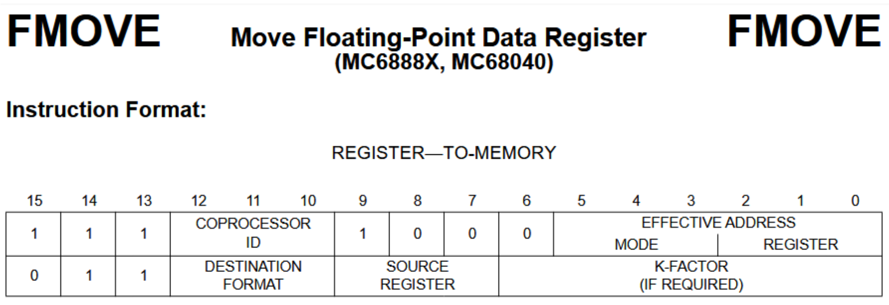
The argument is therefore transferred to the operand register by two long word transfers, first the upper 32 bits, then the lower 32 bits. In the example shown, the number +0.0 is transferred, where in floating-point representation all bits are 0.

```
movew   #0x7400,%a0@(comm)
```



```
OK >sdpoke $FFFA4A,$7400
```

The next command is transferred to the command register. Again, [4] explains its meaning:



FMOVE **Move Floating-Point Data Register** **FMOVE**
(MC6888X, MC68040)

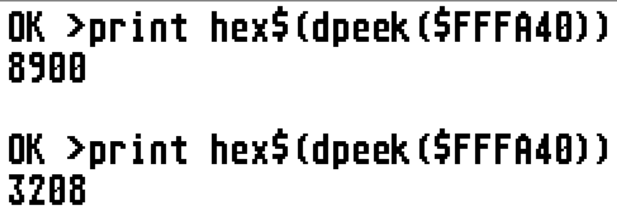
Instruction Format:

REGISTER—TO-MEMORY

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	COPROCESSOR ID			1	0	0	0	EFFECTIVE ADDRESS MODE			REGISTER		
0	1	1	DESTINATION FORMAT			SOURCE REGISTER			K-FACTOR (IF REQUIRED)						

Therefore, it is a FMOVE instruction from an FPU register to the CPU.[2] The data format is again double precision real (Destination Format = 101), the source register is FP0 (Source Register = 000), in which the result of the previous calculation was stored.

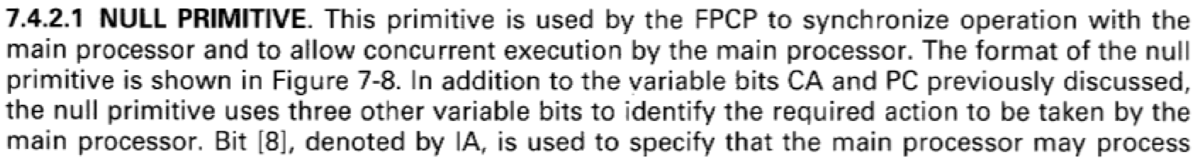
```
.long   0x0c688900,0xfff067f8
```
… disassembled:
```
wait:
cmpiw   #0x8900,%a0@(resp)
beq.s   wait
```



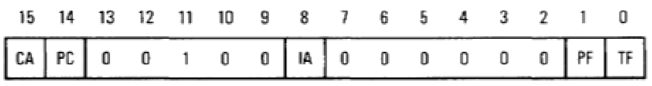
```
OK >print hex$(dpeek($FFFA40))
8900

OK >print hex$(dpeek($FFFA40))
3208
```

The response register is read out again. Again, no error handling takes place, only waiting until the value is not equal to $8900.

The example shows that the FPU actually returns $8900, a *Null Primitive*:



7.4.2.1 NULL PRIMITIVE. This primitive is used by the FPCP to synchronize operation with the main processor and to allow concurrent execution by the main processor. The format of the null primitive is shown in Figure 7-8. In addition to the variable bits CA and PC previously discussed, the null primitive uses three other variable bits to identify the required action to be taken by the main processor. Bit [8], denoted by IA, is used to specify that the main processor may process

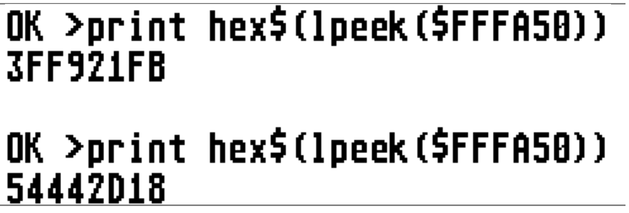


15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CA	PC	0	0	1	0	0	IA	0	0	0	0	0	0	PF	TF

Figure 7-8. Null Primitive Format

In particular, CA = 1 (*come again*) indicates that the register should be read again. The second read operation results in the already known *Evaluate Effective Address and Transfer Data Primitive*. However, this time $3208 indicates a transfer from the FPU (DR = 1) of 8 bytes in length. This is the data of register FP0 requested via FMOVE.

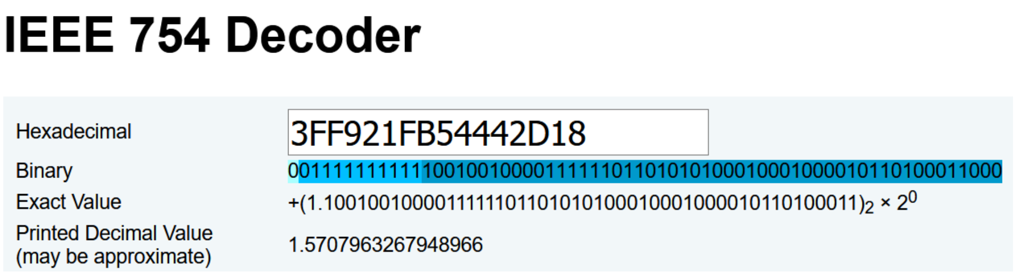
```
movel   %a0@,%d0
movel   %a0@,%d1
```



```
OK >print hex$(lpeek($FFFA50))
3FF921FB

OK >print hex$(lpeek($FFFA50))
54442D18
```

The data is read out – again as two long words – from the operand register. As requested, it is a double-precision floating-point number, first the upper 32 bits, then the lower 32 bits.

Since the 68881/68882-FPU follows the IEEE-754 number format, the number can be decoded:



IEEE 754 Decoder

Hexadecimal	3FF921FB54442D18
Binary	0011111111111001001000011111101101010100010001000010110100011000
Exact Value	$+(1.1001001000011111101101010100010001000010110100011)_2 \times 2^0$
Printed Decimal Value (may be approximate)	1.5707963267948966

It can be seen that this is π/2, which is the correct result of the calculated function acos(0).

[2] The description of the instruction as *register to memory* is to be understood from the view of the CPU. The FPU has no possibility to write data into the memory. It is the task of the CPU to decode the *effective address* in the first word of the opcode and to write data into the memory if necessary.

Mathematical operations with two operands, e.g. a multiplication, require that one operand is present in an FPU register. Therefore, for these operations, an operand is first loaded into the FPU using an *FMOVE <EA> to Register* instruction.

BIBLIOGRAPHY

[1] Motorola, Inc., MC68881/MC68882 Floating-Point Coprocessor User's Manual, Englewood Cliffs, NJ, USA: Prentice Hall, 1987.

[2] S. Sanders, „The 68881 Floating Point Coprocessor," in *The Atari Compendium*, Long Beach, CA, SDS Publishing, 1993.

[3] Multiple authors, „Portable Math Library (PML)," [Online]. Available: https://github.com/th-otto/pml/blob/01810cabc77c9540cc91d19b4c547232e8ba137e/pmlsrc/acos.c#L185-L196. [Accessed 2 August 2020].

[4] Motorola, Inc., M68000 Family Programmer's Reference Manual, 1992.