# 68080 FPU Core

## Overview

Since the **GOLD2.7 Core**, the Vampire boards embed a brand new FPU Core in the FPGA.

This new FPU Core is:

- mostly **hardware-implemented**,
- very close to an **FPU060**, technically speaking,
- **100% pipelined** (parallel CPU/FPU coding support),
- **fast** (around 35-40 MFLOPS in SysInfo),
- able to use a dedicated Floating Point Software Package (FPSP080).

The whole set of instructions from the MC68040 and MC68060 is available, as well as the whole MC68881 and MC68882 subset.

From an end-user or end-coder perspective, the 68080 FPU does not provide any new floating point instructions or any other big changes; the FPU instruction set is the legacy one.

The APOLLO-Team experimented with different options, based on the work initiated by Jari Eskelinen in **FEMU** (kudos to you, Jari). This work offered the Team an awesome test-bed to incrementally improve the FPU in hardware, and to figure out how to efficiently handle the emulated FPU instructions that are NOT implemented in hardware. During the investigations, it was clear that the usual TRAP'ing mechanism is costly, wasting precious cycles.

Since the 68080 is aiming at MC68040 compatibility, the Team decided to implement a full FPU040/FPU060 Core in hardware and to offer an efficient FPU interface for the instructions that are not implemented in hardware. As a consequence, all the legacy MC68881 and MC68882 instructions are handled with new optimized mechanics.

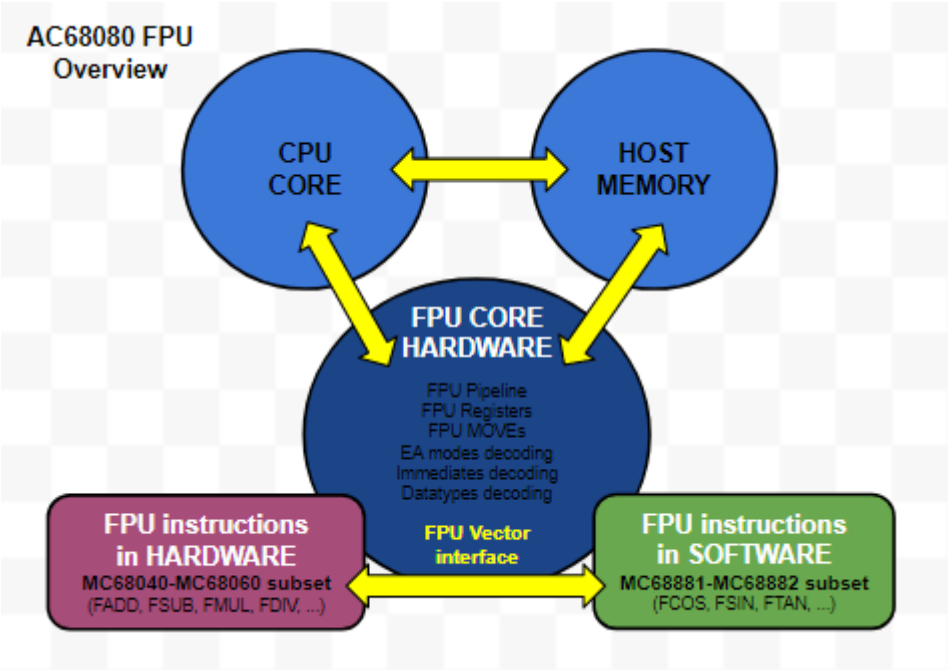
This new approach intends to neatly solve all the constraints the Team has to deal with, such as compatibility, speed, and room in FPGA.

The result of this work consists of an autonomous FPU, not relying on any third-party tool / library anymore (as opposed to what FEMU does, or, on other machines, to what 68040.library/68060.library does).

## Architecture

Below is a simplified diagram of the 68080 FPU architecture, featuring a hardware-implemented **FPU Core** offering all the materials to operate floating-point calculations at the hardware level, and **two**

**subsets** of instructions (040/060 and 881/882 subsets).




## Registers

- FP0 to FP7
- FPSR
- FPCR
- FPIAR
- FPU Vector

All FPU registers are implemented in hardware and are fully operational.

The FPU Vector is the entry point to the FPSP interface. It can be modified only in Supervisor mode.

In some extreme use cases, the end-coder can also access all 68080 64-bit registers (E00 to E23), in addition to the usual FPn registers.

## Data types

- (.b) BYTE
- (.w) WORD
- (.l) LONG
- (.s) SINGLE
- (.d) DOUBLE
- (.x) EXTENDED
- (#imm) IMMEDIATES

All INTEGER from/to FLOAT casts are handled in hardware.

All IMMEDIATES are handled in hardware, which was not the case on the MC68060.

Most of the work is done in hardware, before calling the emulation vector when dealing with 881/882 instructions.

PACKED data type (.p) is handled through the emulation vector.

# EA modes

All existing legacy FPU Effective Address (EA) modes are computed by the 68080 FPU Core, at the hardware level.

Motorola gave up on integrating all the EA modes in hardware in the MC68060 CPU. (See “IMMEDIATES” in the “Data types” section above.) The 68080 FPU Core brings them all back again.

Even for emulated instructions, the EA computation is done in hardware before calling the emulation vector.

# Hardware instructions

- FMOVE, FDMOVE, FSMOVE,
- FMOVEM, FMOVECR,
- FABS, FDABS, FSABS,
- FNEG, FDNEG, FSNEG,
- FADD, FDADD, FSADD,
- FSUB, FDSUB, FSSUB,
- FMUL, FDMUL, FSMUL, FSGLMUL,
- FDIV, FDDIV, FSDIV, FSGLDIV,
- FCMP, FBCC, FSCC, FTST, FINTRZ,
- FSAVE, FRESTORE,
- FNOP

Instructions embedded in hardware offer a large floating-point instruction set, very close to the 040/060 FPUs. Most of the time, a coder can easily avoid the non-implemented ones since all the necessary primitives are available. For example, Quake can run 100% on hardware instructions.

All those instructions run more or less in **1 cycle, so they are very fast**. It all depends on how well the ASM coder makes smart use of the superscalar, and on cache hits.

All those hardware instructions can be used in the emulated instructions.

They are largely used in the embedded FPSP to accelerate the emulation.

# Emulated instructions

- FSQRT, FDSQRT, FSSQRT,
- FACOS, FCOS, FCOSH,
- FASIN, FSIN, FSINH, FSINCOS,
- FATAN, FATANH, FTAN, FTANH,

- FETOX, FETOXM1, FTENTOX, FTWOTOX,
- FGETEXP, FGETMAN, FINT,
- FMOD, FREM, FSCALE,
- FLOG10, FLOG2, FLOGN, FLOGN1P

Those instructions are handled using some **optimized FPSP code**.

The FPSP code is instantiated by using a new dedicated FPU Vector.

It takes full advantage of the hardware FPU Core, such as the pre-computed EA modes, data type casting, and implemented primitives.

Depending on the FPGA size constraints, the APOLLO Core is able to propose different implementations.

Some instructions can be emulated in the FPSP or embedded into the FPU Core, such as the **SQRT** instruction.

# Precision

The original **APOLLO FPU Core** was designed to perform all calculations in **80-bit** “extended precision”, like the original 68k FPUs. It was a very, very large Core, much bigger than the whole TG68 Core itself!

When integrating the **APOLLO FPU Core** into the current Vampire generation, the precision was reduced to **64-bit** “double precision”, and the Core was renamed to **68080 FPU Core**. This decision was made after careful consideration of the following:

- “Extended precision” wastes too much space on the FPGA.
- In real-world scenarios, a precision higher than 64 bits is almost never needed.
- In rare situations where a higher precision is needed, it can be simulated using other techniques.
- CPU manufacturers no longer think that “extended precision” is a good idea. Modern CPUs such as PowerPC and ARM do not support “extended precision”. Even Motorola dropped support for it in their ColdFire processor, which is derived from the 68k architecture.
- Other than a couple of old fractal explorers, there is no Amiga software that is known to require “extended precision”.

The Vampire Standalone has an **Altera Cyclone V** FPGA, which has enough space to accommodate the full 64-bit FPU Core. However, Vampire accelerator boards connected to a classic Amiga have an **Altera Cyclone III** FPGA, which does not have enough space for the full 64-bit FPU Core. Therefore, on these boards, the FPU Core had to be reduced to **52-bit** precision, to make it fit. This precision should be enough to run most apps, games and demos requiring an FPU. There are very few programs that require the full 64 bits of precision and so do not work on the 52-bit FPU Core. The APOLLO-Team is ready to help the authors of those programs to adapt their software to work with 52-bit precision. If adapting the software is not possible, then you can try the following:

1. Turn off the 52-bit FPU Core using `VControl FPU`.
2. Check if the program you want to use detects the missing FPU and falls back to non-FPU routines. If the detection is not automatic:

    - You might need to manually switch to non-FPU routines in the program's settings.
    - You might need to load a separate program file that contains the non-FPU version.
    - You might need to reinstall the program so that it installs the non-FPU version.
3. If it turns out that the program absolutely requires an FPU, you can employ a full-precision FPU emulator in software.
    - If you are using a Macintosh emulator, you can emulate an extended-precision FPU using SoftwareFPU.

# Performance

Overall, the performance of the GOLD2.7 FPU is very acceptable:

- showing a nice 35-40 MFLOPS in SysInfo.
- showing incredible floating point results in AIBB.
- able to execute FPU instructions in parallel to CPU code.
- able to run most of the Amiga RTG+FPU demos/compos at full-speed.
- able to render scenes in 3D modeling software as fast as the fastest existing 060.
- able to run Quake at a decent frame rate (25 fps in Low Res, more than 15 fps in High Res).

The following scores are produced by a small program written specifically to measure the cycle count per FPU instruction, giving a useful overview of the actual speed.

**Raw FPU cycles (GOLD2.7):**

```
* FABS       :        1 cycles
* FACOS      :      182 cycles
* FADD       :        1 cycles
* FASIN      :      185 cycles
* FATAN      :      231 cycles
* FATANH     :      180 cycles
* FCMP       :        1 cycles
* FCOS       :      253 cycles
* FCOSH      :      333 cycles
* FDABS      :        1 cycles
* FDADD      :        1 cycles
* FDDIV      :        2 cycles
* FDIV       :        2 cycles
* FDMOVE     :        1 cycles
* FDMUL      :        1 cycles
* FDNEG      :        1 cycles
* FDSQRT     :      221 cycles
* FDSUB      :        1 cycles
* FETOX      :      296 cycles
* FETOXM1    :      289 cycles
* FGETEXP    :       91 cycles
* FGETMAN    :       91 cycles
* FINT       :      117 cycles
* FINTRZ     :        2 cycles
* FLOG10     :      281 cycles
* FLOG2      :      291 cycles
```

```
* FLOGN      :      266 cycles
* FLOGN1P    :      266 cycles
* FMOD       :      134 cycles
* FMOVERm    :        1 cycles (Read from memory)
* FMOVEWm    :        1 cycles (Write to memory)
* FMOVERi    :        1 cycles (Read from register)
* FMOVEWi    :        1 cycles (Write to register)
* FMOVECR    :        1 cycles (Read constants)
* FMOVECTRL  :        8 cycles (Fmovem fpsr/fpcr/fpiar,(An))
* FMOVEMR    :        8 cycles (Fmovem (An),fp0-fp7)
* FMOVEMW    :       19 cycles (Fmovem fp0-fp7,(An))
* FMUL       :        1 cycles
* FNEG       :        1 cycles
* FREM       :      155 cycles
* FSABS      :        1 cycles
* FSADD      :        1 cycles
* FSCALE     :      121 cycles
* FSDIV      :        2 cycles
* FSGLDIV    :        2 cycles
* FSGLMUL    :        1 cycles
* FSIN       :      266 cycles
* FSINCOS    :      336 cycles
* FSINH      :      354 cycles
* FSMOVE     :        1 cycles
* FSMUL      :        1 cycles
* FSNEG      :        1 cycles
* FSQRT      :      223 cycles
* FSSQRT     :      221 cycles
* FSSUB      :        1 cycles
* FSUB       :        1 cycles
* FTAN       :      244 cycles
* FTANH      :      343 cycles
* FTENTOX    :      289 cycles
* FTST       :        1 cycles
* FTWOTOX    :      340 cycles
* FBCC       :        1 cycles
* FSCC       :        1 cycles
* FNOP       :        1 cycles
```

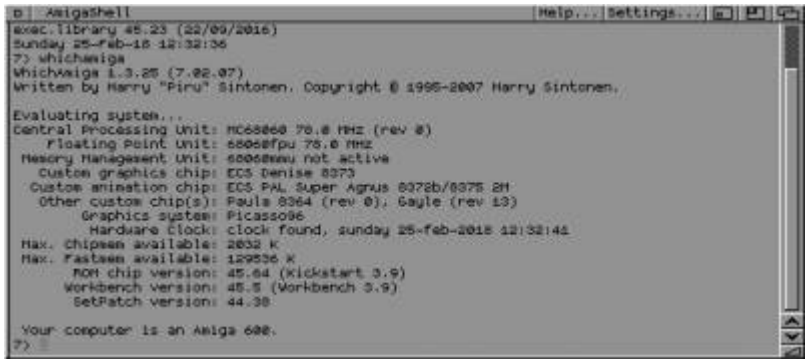
The cycles count clearly shows what is handled in hardware or through FPSP.

# Benchmarks

Below are some well-known Amiga benchmark measurements, from GOLD2.7 Core.

**SysInfo 4.0**


### WhichAmiga 1.3.25



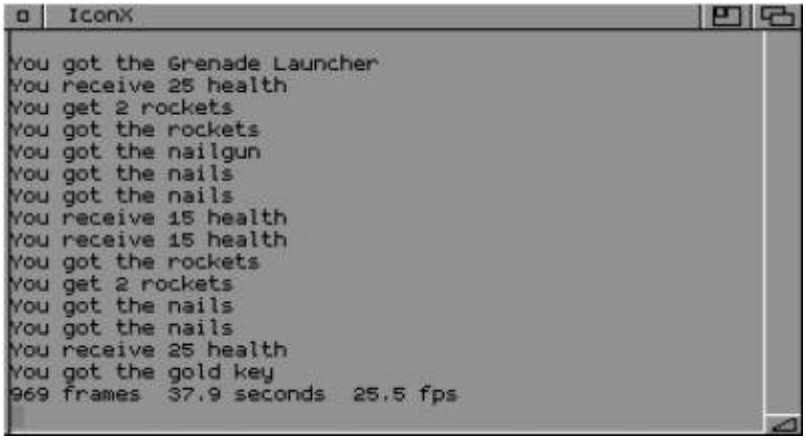

### **AIBB 6.5 BeachBall bench** (Rendered on PAL: High Res 4 Colors)


### **AIBB 6.5 BeachBall bench** (Rendered on PAL: High Res 16 Colors)

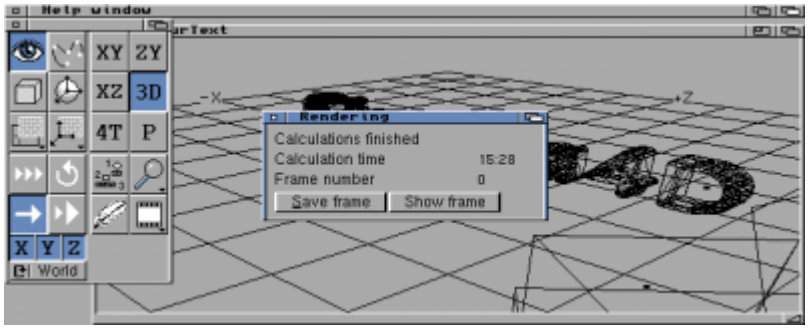

### AIBB 6.5 FMath bench


### AIBB 6.5 FMatrix bench

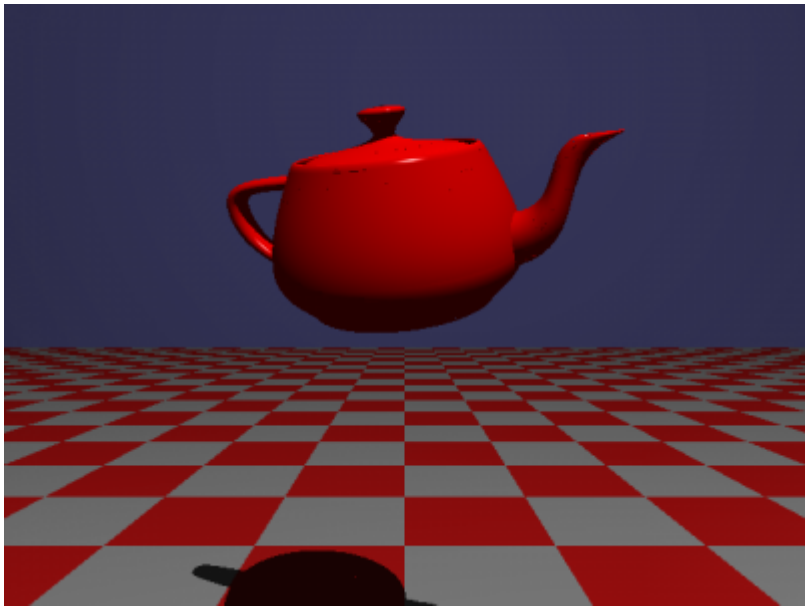
### Quake in 320×200


**Cinema 4D FPU** (Colourtext, C4D 080fpu@77, 640×480, AA 4×4: 15m 28s)




**POV-Ray 3.1** (Teapot rendering)



**LightWave 3D 3.5 (FP version)** (Benchmark scene, rendered on A600 ECS screen: 1h 2m 15s)


(...compared to Amiga 1200 ACA 1233/40MHz FPU 50MHz: 11h 22m 24s)

Video

# Videos

Below are some videos recorded by the beta-testers.

- FPU happy tests
- Imagine 4.0 FPU version
- CineMorph FPU version
- FPU RTG demo
- Mini Metal Slug FPU in an RTG Workbench window
- Quake FPU
- VistaPro Makepath FPU

# Deprecated: FEMU by Jari

The current FPU Core is NOT based on the FEMU program anymore. It is a full rewrite, aiming at a cleaner, faster, and safer emulation code. In fact, FEMU was calling the OS math libs to emulate the missing instructions, which is NOT the case anymore.

As a reference for comparison, here is a sample of FEMU scores on a GOLD2.5 Core:

```
* FATAN      :     1327 cycles
* FCOS       :     2522 cycles
* FLOG10     :     3030 cycles
* FLOG2      :     3561 cycles
* FLOGN      :     2909 cycles
* FSIN       :     1558 cycles
* FSINH      :     1219 cycles
* FSINCOS    :     4972 cycles
* FTAN       :     1531 cycles
* FTANH      :     2460 cycles
* FTENTOX    :     4258 cycles
* FTWOTOX    :     2565 cycles
```

---

You are here: start » apollo_core » fpu