HW-SW Implementation of a Decoupled FPU for ARM-based Cortex-M1 SoCs in FPGAs

Jaume Joven*‡, Per Strid†, David Castells-Rufas‡, Akash Bagdia†, Giovanni De Micheli*, Jordi Carrabina‡
*LSI-EPFL, Station 14, 1015 Lausanne, Switzerland
Email: {jaime.jovenmurillo, giovanni.demicheli}@epfl.ch
†ARM Ltd, 110 Fulbourn Rd, Cambridge CB1 9NJ, United Kingdom
Email: {per.strid, akash.bagdia}@arm.com
‡CEPHIS-MISE, Campus UAB, Building Q (ETSE), 08193 Bellaterra, Spain
Email: {jaume.joven, david.castells, jordi.carrabina}@uab.es

***Abstract*—Nowadays industrial monoprocessor and multiprocessor systems make use of hardware floating-point units (FPUs) to provide software acceleration and better precision due to the necessity to compute complex software applications.**

This paper presents the design of an IEEE-754 compliant FPU, targeted to be used with ARM Cortex-M1 processor on FPGA SoCs. We face the design of an AMBA-based decoupled FPU in order to avoid changing of the Cortex-M1 ARMv6-M architecture and the ARM compiler, but as well to eventually share it among different processors in our Cortex-M1 MPSoC design. Our HW-SW implementation can be easily integrated to enable hardware-assisted floating-point operations transparently from the software application.

This work reports synthesis results of our Cortex-M1 SoC architecture, as well as our FPU in Altera and Xilinx FPGAs, which exhibit competitive numbers compared to the equivalent Xilinx FPU IP core. Additionally, single and double precision tests have been performed under different scenarios showing best case speedups between 8.8x and 53.2x depending on the FP operation when are compared to FP software emulation libraries.

I. Introduction

The *IEEE Standard for Binary Floating-Point Arithmetic (IEEE-754)* [1] is the most widely-used standard (since 1985) for floating-point computation, and it is used by many processors, compilers and custom hardware floating-point units (FPU) implementations. The standard defines formats for representing floating-point numbers in single and double precision (i.e. including zero and denormal numbers, infinities and NaNs) and special values together (such as $\pm\infty$ or ± 0).

The current trends towards multiprocessor systems [2][3] requires to deliver high system performance under very constrained power and area budgets, specially in the embedded domain. At the architecture level, NoCs (Networks-on-Chips) [4][5] and efficient multi-layer AMBA-based [6][7] fabrics have been proposed to communicate the IP blocks in order to create highly-scalable systems at reasonable hardware cost.

Industrial applications have to execute a great variety of kernels which often use occasionally or intensively floating-point arithmetic rather than non-accurate fixed-point arithmetic. Examples of these applications range from digital multimedia (e.g. audio and video) and signal processing (e.g. DCT/iDCT, FFT) tasks, 3D gaming for graphics processing, software-defined radio, wireless communication, to control or computation intensive applications on embedded automotive real-time systems. As a consequence of the application requirements, most of them integrate floating-point (FP) support in hardware to accelerate applications.

In 2007 ARM launched Cortex-M1[8], a streamlined three-stage 32-bit soft-core architecture designed to be used in generic FPGA-based SoCs. This processor is fully compatible with his predecessor, the Cortex-M3 [9] but replacing ARMv7 architecture with ARMv6-M, which use even smaller and compact ISA. Both processors execute floating point instructions by means of software emulation routines instead of using hardware-assisted floating units.

Thus, in this work, the first objective is to design of a IEEE-754 FPU which can be integrated effortlessly in any FPGA device together with the Cortex-M1 soft-core processor in order to accelerate FP operation whenever it is required. At the same time, the second target is to face a decoupled AMBA-based design of the FPU without changing the ARMv6-M architecture and the compiler. This particular design opens the possibility to share a single FPU among different processors in low cost Cortex-M1 MPSoCs in FPGAs. As a consequence, a part from the hardware design, in this paper we also focus on the exploration of different alternatives to integrate the decoupled Cortex-M1 FPU by taking into account the CPU-FPU communication protocol, as well as the software requirements. The main contributions of this paper are:

- The design of an experimental AMBA-based decoupled hardware FPU for Cortex-M1 FPGA systems.
- The design of 8-core Cortex-M1 MPSoC for FPGA platforms.
- The exploration and synchronization methods between the hardware and the software (i.e. CPU-FPU communication protocol).
- The study of the viability to share several FPUs among different processors by means of floating-point transactions on AMBA fabrics.
- FPU evaluation, in terms of area and performance, in a real Cortex-M1 FPGA system.

This paper is organized as follows. Section II presents the

related work on FPU design highlighting the relevant ones on the FPGA domain, where the ARM Cortex-M1 soft-core is targeted. Section III gives an overview of the hardware implementation and the architectural extensions to design our AMBA-based decoupled Cortex-M1 FPU. Section IV explains the CPU-FPU communication protocol. Section V describes the FPU hardware-dependent software communication library to enable a transparent hardware-assisted execution of FP operations. Section VI reports the synthesis results on different FPGA devices, and the performance results of running floating-point kernels on different Cortex-M1 based systems. Finally, Section VII concludes the paper.

II. Related Work

Nowadays, FPGA devices offers the possibility to use embedded DSP blocks (i.e. adders, multipliers, barrel shifter, etc.) to compute floating point operations, however these resources are limited in FPGAs.

In addition, main FPGA vendors such Altera or Xilinx – thanks to the use of soft-core processors and depending on the requirement of our system – offer the possibility to include customized hardware FPUs using this embedded DSP blocks together with their soft-core processors. Altera offers the FPU as a custom instruction inside the Nios II processor [10], and also the possibility to create standalone FP Altera Megafunction [11] (e.g. addition/subtraction, multiplication, division, logarithm, etc.) to speed up FP applications. Xilinx provides an equivalent approach, and therefore, the designer can optionally integrate a FPU on MicroBlaze or PowerPC [12] processors, or a custom FP operation macros [13] on the system. In Gaisler[14], LEON3-based systems also includes a GRFPU, an FPU compliant with SPARC V8 standard (IEEE-1754), which can be synthesized in both, ASICs or FPGA devices.

Xtensa processors also provide these capabilities extending the base ISA by using Tensilica Instruction Extension (TIE) [15][16] to add new features as for instance AES/DES encryption, FFT and FIR operations, double precision FP instructions. Thus, the processor can be tailored to a particular set of SoC applications delivering high-performance and energy efficiency at reasonable cost, becoming an ASIP-like processor.

All these FPU architectures have tightly-coupled coprocessors in each CPU to compute FP operations purely in hardware following the IEEE-754 standard. Thus, they have support for single and/or double precision addition, subtraction, multiplication, division, square root, comparison, etc, as well as built-in conversion instructions to transform from integer to FP types and vice versa.

In this work, the target is to provide an ARM *plug&play* FPU to be used together with the Cortex-M1 in FPGA applications. As in previous works, our FPU design is IEEE-754 compliant, and it has been optimized to be synthesized efficiently in FPGA devices. Additionally, we face a different design challenge to shed light on the possibility to share a single FPU among different processors on the system. Therefore, we believe that a hardware-software decoupled implementation of such a FPU design will be useful to speed up FP kernels in Cortex-M1 applications on FPGA systems. Furthermore, in Cortex-M1 MPSoC embedded architectures in FPGAs, where the number of area resources and specially DSP blocks are limited, a shared FPU will be crucial to design low-cost systems which occasionally they have to compute floating-point operations.

III. Design of a Decoupled Cortex-M1 Floating Point Unit

Main FPGA vendors, i.e. Altera and Xilinx, offer the possibility to include their Nios II and MicroBlaze soft-core processors, respectively, with or without a FPU in their FPGA devices. In this section, we describe our FPU "accelerator" design targeted to be effortlessly integrated with Cortex-M1 soft-core processor in different FPGA devices. To face the hardware design of the FPU, two consideration have been taken into account to avoid to modify the Cortex-M1 ARMv6-M architecture:

- The FPU design must be an independent extension of ARMv6-M architecture, and therefore no changes are allowed on the Cortex-M1 core.
- The FPU must be completely independent from the ARM compiler.

Both considerations imply that, the FPU rather than being tight-coupled inside the processor datapath as it is done Cortex-M4, it will be designed to be standalone and decoupled from the Cortex-M1 processor architecture. As a consequence, the external FPU interface will be designed to support AMBA 2.0 AHB[17] transactions, since it will attached directly to the interconnection fabric.

Our Cortex-M1 FPU is based on the Cortex-R4F FPU [18] internal architecture shown in Figure 1. However, a complete redesign has been done by decoupling the FPU microarchitecture from the original one, using external multipliers rather than using the integer unit, and finally by optimizing the design to be mapped in FPGA devices. Our design is IEEE-754 compliant and most important it is backward compatible with earlier ARM FPUs (VFP9/10/11).

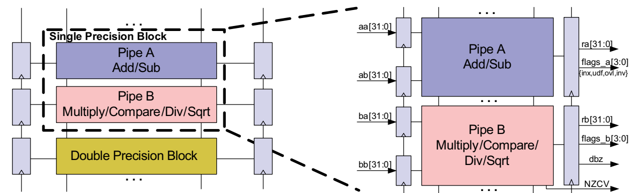

Fig. 1. Internal pipelined FPU architecture

The FPU architecture is divided in two fully-pipelined parts: *(i)* a dual-pipelined architecture for SP, and *(ii)* a unified pipeline for DP floating point operations. This architecture is optimized for single precision (SP) without sacrificing double precision (DP). The only two instructions that are not pipelined are the division and square-root, which use an iterative algorithm stalling the pipeline of the FPU. However, they are

calculated in a separate non-blocking execution unit, allowing all other operations to be performed in parallel without stalling the FPU pipeline. In all the pipelines, the results of SP and DP operations are stored in the WR stage in 32 and 64 bit registers, respectively. Moreover, the FPU also stores, in FPSCR register (see Figure 3), the corresponding value of FP exceptions and flags generated by the operation executed. This block has been designed and integrated as a part of the "back-end" of our Cortex-M1 FPU.

On the other hand, the "front-end" implementation of our Cortex-M1 FPU (see Figure 2) is an AMBA-based memory-mapped subsystem which includes the following blocks:

- AMBA 2.0 AHB front-end interface and control units, fully compatible with AMBA 2.0 AHB [17] or AHB ML standards [6].
- Memory-mapped FP register banks.
- FP micro-instruction decoders.
- The required external multipliers required.
- Hardware semaphores to lock/unlock exclusive access to the FPU.

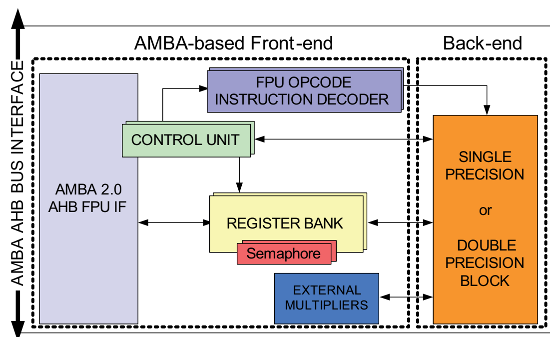

Fig. 2. AMBA-based Cortex-M1 Floating-Point Unit

The AMBA AHB interface module translates control and data AHB transactions from the CPU towards our FPU. Since the FPU will be attached upon AMBA-based systems, a FSM acts as a control unit of the FPU. This control unit supervises the CPU-FPU protocol presented in Section IV, by asserting/de-asserting properly the control lines to/from the other blocks in our decoupled FPU.

To support the same FP features as in traditional tight-coupled ARM FPUs, we build an equivalent register bank in our AMBA FPU. As shown in Figure 3, it consists of a multiple 32-bit registers memory-mapped on the address space of the system. The FPU register bank includes, an identification register (i.e FPSID), a control and status register (i.e. FPSCR), an execution register (i.e. FPEX), as well as a of set of registers to store the SP and DP operands, the opcode of the FP operation, and the final result of the FP operation. It is important to notice that, in SP only *Operand AA*, and *Operand AB* are used together with both results register depending if the requested operation is performed on the pipe A or B of the SP dual-pipeline architecture (see Figure 1). On the other hand, if DP is used, all pairs of 32-bit operands and the results are used.

FPSID (32-bit RO)
FPSCR (32-bit RW)
FPEX (32-bit RW)
FP Operand AA (32-bit WO)
FP Operand AB (32-bit WO)
FP Operand BA (32-bit WO)
FP Operand BB (32-bit WO)
FP Opcode (32-bit WO)
Result A (32-bit RO)
Result B (32-bit RO)

Fig. 3. Cortex-M1 FPU memory-mapped register bank

This front-end has been designed to be extremely slim (integrates only one register bank per core) and latency efficient. It only takes one cycle delay for AHB writes, and two cycles to handle AHB read operations on the register bank.

IV. HW-SW CPU-FPU Communication Protocol

The fact that our Cortex-M1 FPU is completely decoupled and memory-mapped on the system opens different alternatives when the CPU needs to issue a FP instruction. Figure 4 shows our proposed CPU-FPU protocol at transaction level:

(i) Check whether the FPU is busy or not computing the previous FP operation (Step 0).
(ii) FPU configuration (Steps 1-3), involving the transaction of the *operands* and the *opcode* of the FP operation.
(iii) FPU computes the operation, and afterwards the CPU collects it (Steps 4-6).

In *(i)*, the principle is to enable the ability to share the FPU among several processors in the system. At any time, any processor can issue a FP, and therefore, some serialization and synchronization must be enforced. Thus, the proposed CPU-FPU protocol must ensure a certain order during the execution of FP operation between the FPU and the CPU.

Often, *test&set* instructions and mechanism to lock the interconnection fabric are provided by the ISA. Unfortunately, the architecture ARMv6-M, and the Cortex-M1 do not support exclusive access transactions. As a consequence, a slave-side synchronization support has been provided to the FPU by means of a semaphore in the FPEX register to enable traditional lock/unlock support under concurrent access on the shared FPU.

In the simplest synchronization case, one or more processors competing for our shared FPU resource may poll the semaphore to gain access to the shared FPU. Once, a CPU gets access to the FPU, it automatically asserts the semaphore *FPU_BUSY* bit. This prevents that at the same time two FP operations are executed on the shared resource from the same CPU. When the result has been read by the requesting CPU, the FPU unlocks by de-asserting the semaphore bit. This simple protocol will ensure serialization of the FP instructions to share the FPU in a MPSoC system.

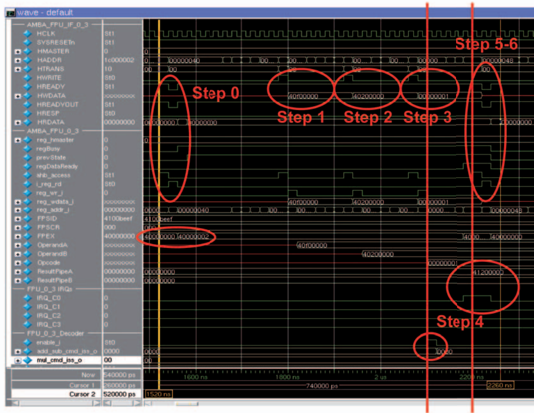

Fig. 4. CPU – FPU hardware-software protocol

In *(ii)*, multiple 32-bit AHB write transactions involving the copy of the operands (i.e. 32 and 64 bits operands for SP and DP, respectively) and the *opcode* of the FP operation on the corresponding memory-mapped registers. The transfer of the *opcode* triggers the starting of the FP operation.

Finally, in *(iii)*, different ways can be implemented to notify/wait and collect the result of the FP operation from the CPU. In this work, we explored and support the following notification schemes in our Cortex-M1 FPU:

- *No Operation (NOP)* – This mechanism includes NOPs operations on the software library wasting CPU cycles to synchronize exactly the latency required by the FPU to compute the FP instruction.
- *Polling* – This mechanism is done from the CPU by polling permanently the *DATA_READY* bit on FPEX register.
- *Interrupts (IRQ)* – A very common slave-side method used when a slave wants to notify the master an issue, in our case, when the FPU result is ready.

V. FPU Hardware-Dependent Software Library

This section presents a lightweight hardware-dependent communication library for our decoupled Cortex-M1 FPU according to the CPU-FPU protocol and notification mechanisms presented in Section IV. Since our FPU is a decoupled and memory-mapped on the system, each FP function access the FPU using an address pointer which indicates the base address of the FPU.

```
unsigned int * AMBA_FPU = (unsigned int *) (FPU_ADDRESS);
```

In Listing 1, we show as an example of the implementation of a single-precision `fsub` operation. Relative offsets to `AMBA_FPU` base address are used to access all FPU registers together with bit masks in order to program and execute FP instructions in our AMBA-based Cortex-M1 FPU.

Additionally, all the other FP functions (such as `fadd`, `fmul`, `fsqrt`, `ddiv`, `dadd`, `dmul`, etc), as well as `fsub` can potentially make use of the alternatives CPU-FPU notification protocols described in Figure IV by means of a pre-compiler directives (i.e. `-DNOPS_PROTOCOL`, `-DPOLLING_PROTOCOL` or `-DIRQS_PROTOCOL`).

It is important to remark that, all FP functions included in our FPU software library to work with the Cortex-M1 FPU, only generate few assembly instructions, which lead to a very efficient and fast CPU-FPU communication interface.

Listing 1. *FSUB* implementation using our decoupled Cortex-M1 FPU

```
unsigned int amba_fpu_fsub(unsigned int a, unsigned int b)
{
  // Getting access to the Cortex-M1 FPU (Step 0)
  volatile unsigned int busy = 1;
  do {
    busy = (AMBA_FPU[FPEX_OFFSET_REG] &
            FP_EXC_FPU_BUSY_MASK) >> 1;
  } while (busy);

  // Operands and opcode transfers (Step 1-3)
  AMBA_FPU[FP_OPAA_OFFSET_REG] = a;
  AMBA_FPU[FP_OPAB_OFFSET_REG] = b;
  AMBA_FPU[FP_OPCODE_OFFSET_REG] = FSUBS_OPCODE;

  // CPU waits the result from the FPU (Step 4)
  #ifdef NOPS_PROTOCOL
    __nop();
    __nop();
    __nop();
  #endif

  #ifdef POLLING_PROTOCOL
    while (!(AMBA_FPU[FPEX_OFFSET_REG] &
             FP_EXC_DATA_READY_MASK)){}
  #endif

  #ifdef IRQS_PROTOCOL
    while (irqHandler == 0){}
      irqHandler = 0;
  #endif

  // Return the result (Step 5-6)
  return AMBA_FPU[FP_RESULT_PIPEA_OFFSET_REG];
}
```

The presented FP function and all the others included in our software library will replace the existing `__aeabi_<fname>()` SW emulation FP library by means of defining the appropriate substitution `$Sub$$__aeabi_<fname>()` prototypes. In Listing 2, we show as an example, different prototypes of the EABI standard from ARM architecture.

Listing 2. Hardware-assisted FPU routines

```
// Hardware-assisted functions that will replace
// the SW emulation routines using our equivalent
// amba_fpu_<fname> routines
float $Sub$$__aeabi_fadd(float a, float b);
float $Sub$$__aeabi_fsub(float a, float b);
float $Sub$$__aeabi_fmul(float a, float b);
float $Sub$$_fsqrt(float a);
double $Sub$$__aeabi_dadd(double a, double b);
double $Sub$$__aeabi_dmul(double a, double b);
double $Sub$$__aeabi_ddiv(double a, double b);
double $Sub$$__aeabi_f2d (float a);
float $Sub$$__aeabi_d2f (double b);
...
```

These functions call directly our equivalent `amba_fpu_<fname>()` hardware FP implementation using our software library. Thus, whenever is required, at

compile time, we can effortlessly instrument the application code to enable hardware-assisted FP operations on our Cortex-M1 FPU by means of another pre-compiler directive, i.e. -DCM1_AMBA_FPU.

Despite this fact, the original microlib software library that only executes FP instructions purely in software can be also called using %Super%%__aeabi_<fname>() corresponding to each FP operation.

VI. Experimental Results

In this section, we report the results obtained to implement our Cortex-M1 FPU in real prototyping FPGA devices from Xilinx and Altera. Later, we integrate the FPU on a Cortex-M1 MPSoC cluster-on-chip architecture presented in Figure 5 in order to evaluate different alternatives CPU-FPU protocol to communicate the operation result and the overall performance, in terms of clock cycles and and speed up versus the pure EABI software emulation FP functions.

A. Hardware Synthesis

In this section we report the synthesis results, in terms of LUTs, circuit performance (f_{max}), embedded RAM and DSP blocks used of our Cortex-M1 FPU. To evaluate the trend, and since both, the ARM Cortex-M1 and our FPU design have been targeted to be used in FPGAs, we show synthesis results of our FPU in multiple FPGA from Xilinx and Altera.

	Part/Device	Speed (MHz)	LUTs	RAM	DSP blocks
Xilinx	xc2v1000bg575-6	84.0	3592	1 M512	4
	xc4vfx140ff1517-11	101.6	3655	1 M512	4
	xc5vlx85ff1153-3	134.2	2890	1 M512	4
Altera	EP1S80B956C6	76.9	3937	0	1
	EP2S180F1020C4	132.0	2697	0	1
	EP3SE110F1152C2	176.5	2768	0	1

TABLE I
CORTEX-M1 SP FPU SYNTHESIS ON XILINX AND ALTERA FPGA DEVICES

Focusing on the single precision implementation, we can affirm that our Cortex-M1 FPU accelerator (see Table I) is quite competitive in contrast with the one provided by Xilinx (see the results from Xilinx in [12]). Thus, in our implementation we use only 270 LUTs more (2790 of our Cortex-M1 SP FPU accelerator against the 2520 LUTs for Xilinx FPU). This overhead may be caused by the integration of additional blocks in our decoupled memory-mapped FPU, such as the register bank, the AMBA 2.0 AHB interface and its decoder unit, the semaphore, etc, which are presumably not counted in the Xilinx implementation. In terms of circuit f_{max}, our SP FPU is only 5% slower compared to the Xilinx FPU, 134.2 MHz vs. 140 MHz, respectivelly, on equivalent low-latency configuration in Virtex-5 FPGAs. However, depending on the FPGA technology and speed grade, these results can vary, and for instance, the circuit performance can scale up to 176.5 MHz on Altera Stratix III devices.

Finally, our SP FPU design on Xilinx FPGAs use 4 blocks DSP48 against the 3 used by PowerPC FPU, and it requires 1 M512 block of embedded RAM to map the FPU register bank. On Altera FPGAs, our FPU does not require embedded FPGA memory (the register bank is implemented using LUTs and registers) and it only needs one DSP block.

On the other hand, as shown in Table II, our DP FPU uses the same amount of DSP blocks and embedded RAM, but it is really competitive as compared to Xilinx FPU. First, it employs 9 DSP blocks less as compared to the Xilinx DP FPU, and it uses only 4312 LUTs, which means 12.8% less than the 4950 LUTs that takes the equivalent Xilinx FPU. In terms of circuit f_{max}, our DP FPU can reach 149.6 MHz in Virtex-5 FPGAs, which is on the same performance range of the Xilinx DP FPU.

	Part/Device	Speed (MHz)	LUTs	RAM	DSP blocks
Xilinx	xc2v1000bg575-6	80.3	5652	1 M512	4
	xc4vfx140ff1517-11	97.4	5694	1 M512	4
	xc5vlx85ff1153-3	149.6	4312	1 M512	4
Altera	EP1S80B956C6	70.9	5760	0	1
	EP2S180F1020C4	122.7	3970	0	1
	EP3SE110F1152C2	168.0	4007	0	1

TABLE II
CORTEX-M1 DP FPU SYNTHESIS ON XILINX AND ALTERA FPGA DEVICES

Usually, FPUs are relatively bigger than soft-core processors, i.e. ≈1.5-2x the area of Nios II [10], MicroBlaze [19], or ARM Cortex-M1, and ≈3x-4x bigger when double precision support is included. Thanks to the fact that our FPU is decoupled from the CPU, we can explore to share an arbitrary number of FPUs among different CPUs in order to design low-cost systems that requires occasionally FP support.

To validate this approach we designed a Cortex-M1 based MPSoC evaluation framework in a Xilinx FPGA.

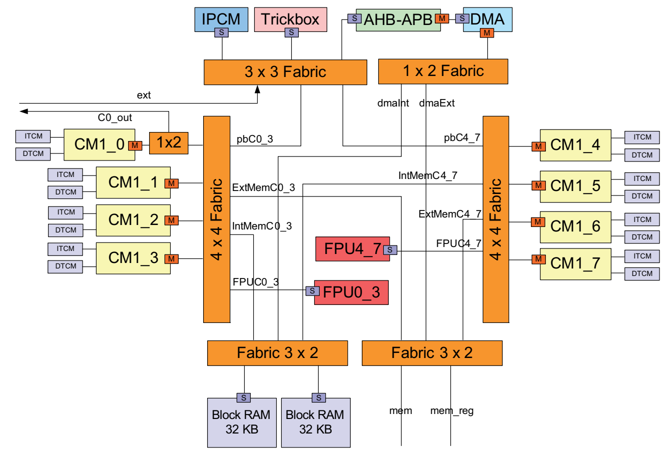

Fig. 5. ARM Cortex-M1 based MPSoC architecture

Figure 5 shows that that the system contains two groups of 4 Cortex-M1 and their associated ITCMs/DTCMs memories interconnected by an AMBA 4x4 AHB interconnection and several common peripherals and a memories. In this architecture, we integrated our decoupled AMBA 2.0 AHB compliant SP or DP FPU attached on the interconnection fabric, one in each side of the cluster.

In Table III and IV, we show synthesis results of the Cortex-M1 MPSoC using an AHB multi-layer as a communication backbone, with and without SP floating-point support. This architecture has been configured to use 8 KB for DTMC/ITCM, and 32 KB for scracthpads AHB memories.

Part/Device	Speed (MHz)	LUTs	rams	DSP blocks
xc4vfx140ff1517	70.0	50315	RAM16: 512 Block RAMs: 98	24 DSP48
xc5vlx85ff1153	117.6	38294	RAM128X1D: 512 Block RAMs: 98	24 DSP48
EP2S180F1020C4	90.9	34360	M4Ks: 400 M512s: 16 ESB: 1613824 bits	8 (78 nine-bit DSP)
EP3SE110F1152C2	122.3	34696	M9Ks: 226 ESB: 1613824 bits	4 (32 nine-bit DSP)

TABLE III
SYNTHESIS OF CORTEX-M1 MPSOC WITHOUT FPU ON XILINX AND ALTERA FPGAS

Part/Device	Speed (MHz)	LUTs	rams	DSP blocks
xc4vfx140ff1517	74.4	57346	RAM16: 512 Block RAMs: 98	32 DSP48
xc5vlx85ff1153	123.9	43790	RAM128X1D: 512 Block RAMs: 98	32 DSP48
EP2S180F1020C4	87.8	39979	M4Ks: 400 M512s: 18 ESB: 1614592 bits	10 (78 nine-bit DSP)
EP3SE110F1152C2	124.1	39936	M9Ks: 226 ESB: 1614592 bits	6 (46 nine-bit DSP)

TABLE IV
SYNTHESIS OF CORTEX-M1 MPSOC WITH FPU ON XILINX AND ALTERA FPGAS

On this system, the f_{max} is not strongly affected by the inclusion of the FPU, and surprisingly in some cases (possibly due to the synthesis heuristics) the resulting f_{max} is even better when the system includes FP support in hardware. This is due to critical path of the system is not in our SP FPU block. In terms of area, the overhead to include the proposed shared SP FPU among each bank of 4 Cortex-M1s represents about 13-14% of the whole system depending on the FPGA technology. In the same shared scheme the area to support DP increases up to approximately 18-20%. As a consequence, we can affirm that the impact, in terms of circuit performance and area, to integrate 2 FPUs to be shared for each bank of 4 Cortex-M1 processors even if is not negligible, can be tolerated.

Nevertheless, in the same MPSoC architecture where each processor has its own tight-coupled FPU, the overhead only to include SP will raise up to 35-39%. These numbers are still more rellevant whether a DP FPU is required. In this case, only the dedicated area on the system to support DP floating-point operations in hardware will raise up to 48-51% depending on the FPGA device. As a consequence, in low-cost systems when application only use FP ocassionally, a shared FPU can be a promising alternative.

B. FPU Evaluation in our Cortex-M1 Based Systems

In order to validate the designed HW and SW components, we run some μkernels in a mono-processor and multi-processor scenario to test the performance when the FPU is shared, and varying at the same time the different synchronization and notification CPU-FPU protocols.

1) Experiments with a Single Cortex-M1 Soft-Core.

First, to test the FPU performance, a single mono-processor architecture has been used using with only one Cortex-M1 processor connected to our FPU accelerator through an AHB ML. Figure 6 and Figure 7 report the latencies to execute a FP instruction in our Cortex-M1 FPU accelerator, measured in clock cycles, according to the different CPU-FPU notification protocols in our SP and DP precision Cortex-M1 AMBA-based FPU.

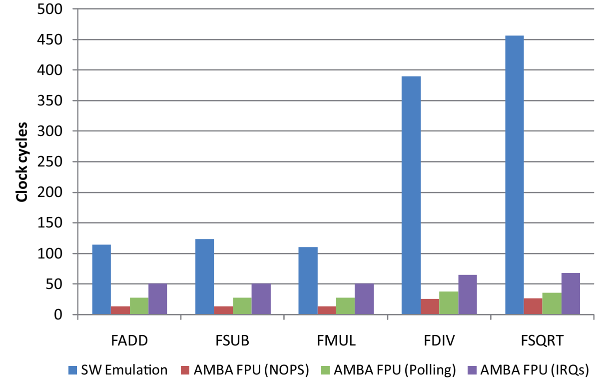

Fig. 6. Cortex-M1 SP FPU latencies of various FP operations using different CPU-FPU notification methods vs. SW emulation FP library

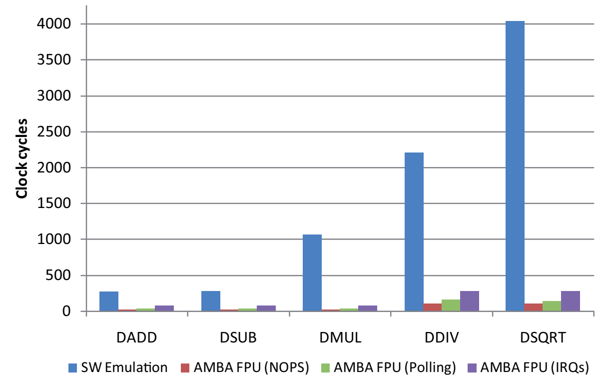

Fig. 7. Cortex-M1 DP FPU latencies of various FP operations using different CPU-FPU notification methods vs. SW emulation FP library

As shown in Figure 6, using our FPU, the latencies of single precision FP operations take tens of cycles, whereas using the ARM SW emulation FP library, the latencies increase up to hundreds of cycles to execute them. When DP is used, the trend is similar, but now the latencies in hardware takes hundreds of cycles, and purely in software, it raises up to thousands of cycles with the exception of DADD and DSUB.

According to each CPU-FPU notification protocol, when NOPs are used, we achieve the maximum performance, i.e. between 15-30 clock cycles to compute SP and DP operations.

This is because, the Cortex-M1 and the FPU on the bus are synchronized ideally including in the communication library exactly the number of NOPs according to the latency of our hardware design.

On the other hand, when we use polling, the latencies are a bit worse as compared with NOPs. This is because polling often penalizes, because a de-synchronization between the time the result is ready on the FPU and the instant when the CPU collects it. In addition, polling generates undesirable traffic on the system that can influence on the traffic of other peripherals on the system.

Finally, the last alternative is to use interrupts. However, this protocol, even if Cortex-M1 is optimized to perform fast IRQs, the notification protocol takes almost the double of clock cycles than NOPs or polling. We explain this effect by observing the context switching and the IRQ handler routines takes around 30 cycles. Despite this fact, in contrast to polling notification, no traffic is added on the communication backbone, resulting on a reduce bus utilization.

To validate the effectiveness of our decoupled design, we compare our FPU design against a purely SW FP emulation libraries. In Figure 8, we show the speedups of a subset of representative FP operations. Depending on the CPU-FPU and the FP operation, the best case (i.e when NOPs are used) speedups in our SP FPU range from 8.8x to 17.6x compared to a purely software FP library. The results are even better if DP operations are executed. Taking the best case, our FPU outperforms by 13.4x up to 53.2x the equivalent SW DP floating-point library.

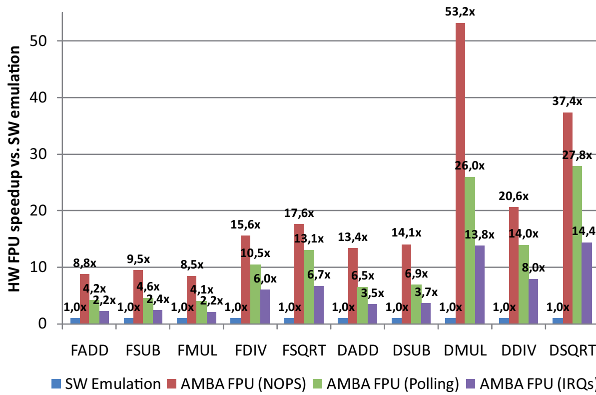

Fig. 8. Single and double precision speedup of different FP operations executed on our Cortex-M1 FPU vs. SW emulation FP library

It is important to remark that the results presented in Figure 6, 7, 8 show single operation with zero latency which are not using the effect of the pipeline. However, since our FPU is pipelined, better overall performance can be achieved under continuous execution of FP operations. Thus, the throughput of our FPU is determined by the capacity of injection of FP instructions from the Cortex-M1, which is 3 cycles for the SP and 5 cycles for DP, accordingly to each AHB transfer to move the two operands and the opcode from the CPU to the FPU.

2) Experiments in our Cortex-M1 MPSoC Architecture. Similarly to the previous experiments, we show now results of the execution time and speedups between our FPU design and the SW emulation FP library. However, in this case, we explore the scalability and the throughput according to the CPU-FPU protocol in the system when the number of sharers (i.e. Cortex-M1 processors) increase under a stress test. The test is an intensive FP benchmark (worst case test) using the pipelined (addition, multiplication, etc) and non-pipeline (division and square root) FP operations.

The plots in Figure 9 and Figure 10 shows that as the number of Cortex-M1 processors increase requesting multiple FP operation, the access congestion lead to delay the completion of the FP operation even if in all the cases both SP and DP FPU outperform the SW FP emulation library. However, the resulting execution times grow when increasingly more processors request FP operations, and therefore the speedups decrease smoothly depending on each CPU-FPU protocol.

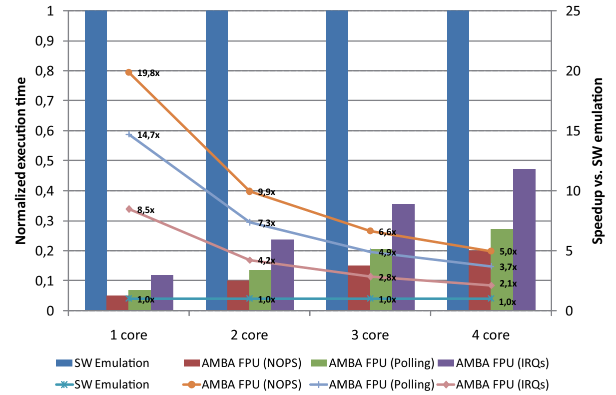

Fig. 9. Scalability tests of our SP FPU versus the SW FP library increasing the number of Cortex-M1 processors in our MPSoC

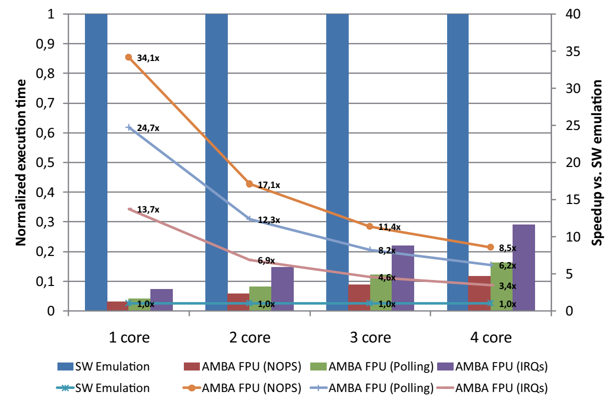

Fig. 10. Scalability tests of our DP FPU versus the SW FP library increasing the number of Cortex-M1 processors in our MPSoC

Additionally, the outcome results of this study show that, the scalability of our system has a bound in presence of concurrent tasks running constantly only FP instructions on the Cortex-M1s. As shown in Figure 9, the bound is almost reached when

IRQ are used to communicate the CPU and our SP FPU, obtaining only a 2.1x speedup versus the SW FP emulation library. Preliminary studies on the presented system reveal that under the same tests, our SP FPU will not outperform the SW FP emulation library whether a fifth Cortex-M1 shares our SP FPU on the system.

As a consequence, we can conclude that sharing one FPU every for 4 cores is an upper bound of the shareability of our decoupled FPU design under very high and adverse FP stress test in our Cortex-M1 system. Nevertheless, it is important to remark that this extreme scenario is not expected to occur often, because workloads and applications do not issue exclusively and continuously FP operations.

VII. Conclusion and Future Work

In this paper a complete design of decoupled SP and DP FPU have been presented to be used with for Cortex-M1 soft-core processor. FP support have been included without modifying ARMv6-M processor architecture and the ARM compiler by means of a decoupled AMBA 2.0 AHB architecture. Our experimental Cortex-M1 FPU have been synthesized and tested in different FPGAs devices to evaluate the performance in real prototyping FPGA systems. In addition, we compared our design with an equivalent Xilinx FPU core integrated with MicroBlaze and PowerPC, exhibiting promising results in terms of area and circuit performance.

Performance results show speedups from 8.8x up 17.6x for single precision and 13.4x to 53.2x for double precision as compared against a software FP emulation library, depending on the FP operation.

On the other hand, in this work we also presented the hardware-dependent CPU-FPU software communication library exploring different slave-side communication mechanisms to an easy integration of our FPU in a wide range of system requirements and architectural schemes. This communication software library can be effortlessly integrated at compilation time by means of a pre-compiler directive with the application code in order use hardware-assisted SP or DP floating-point acceleration. In addition, the outcome of our CPU-FPU protocol exploration, shows that NOPs and polling outperforms the IRQ CPU-FPU notification protocol because of the overhead added by the switching context, and the IRQ handler. However, on heavily traffic conditions, when other peripherals required to use the communication backbone, IRQs may perform better than polling, since it does not inject any traffic on the intercommunication architecture.

The exploration to share the a FPU among few Cortex-M1 processors has been demonstrated as a feasible alternative when a low-cost constraints are imposed on the system, and of course, when the application does not require very intensive and simultaneous FP operations. Thus, the scalability bound under extreme stress FP test shows a shareability ratio of 1:4, i.e. 1 FPU accelerator every 4 ARM Cortex-M1 processors.

For all these reasons, we believe that the presented approach can be used effortlessly in a great variety of Cortex-M1 industrial systems in FPGAs that require to process digital multimedia and signal processing applications. In other words, our decoupled FPU will help to speedup FP operations to achieve high-performance FP computing specially on low-cost ARM Cortex-M1 MPSoCs in FGPAs reconfigurable platforms reducing the amount of hardware resources and DSP blocks.

Finally, it is important to remark, that our experimental decoupled FPU has been targeted and optimized to be used with Cortex-M1 soft-core processor in FPGAs. Nevertheless, thanks to its fully AHB 2.0 compatible interface, it can be used by any other processors or systems that follows AMBA 2.0 AHB standard.

Acknowledgments

This work was partly supported by the the Catalan Government Grant Agency (Ref. 2006FI02007/2009BPA00122), the European Research Council (Ref. 2009-adG-246810) and a HiPEAC Industrial PhD Internship collaboration grant conducted in the R&D Department at ARM Ltd (Cambridge, United Kingdom).

References

[1] "IEEE 754: Standard for Binary Floating-Point Arithmetic," 1985, http://grouper.ieee.org/groups/754/.

[2] A. Jerraya and W. Wolf, *Multiprocessor Systems-on-Chips*. Morgan Kaufmann, Elsevier, 2005.

[3] W. Wolf, "The Future of Multiprocessor Systems-on-Chips," in *Proc. DAC*, June 2004, pp. 681–685.

[4] L. Benini and G. De Micheli, "Networks on Chips: A new SoC Paradigm," *IEEE Computer*, vol. 35, no. 1, pp. 70 – 78, January 2002.

[5] L. Benini and G. D. Micheli, *Networks on chips: Technology and Tools*. San Francisco, CA, USA: Morgan Kaufmann Publishers, 2006.

[6] *Multi-layer AHB Technical Overview*, ARM Ltd., 2001, http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dvi0045b/index.html.

[7] "AMBA 3 AXI overview," ARM Ltd., 2005, http://www.arm.com/products/system-ip/interconnect/axi/index.php.

[8] "ARM Cortex-M1 – ARM Processor," ARM Ltd., 2007, http://www.arm.com/products/processors/cortex-m/cortex-m1.php.

[9] "ARM Cortex-M3 – ARM Processor," ARM Ltd., 2004, http://www.arm.com/products/processors/cortex-m/cortex-m3.php.

[10] "Altera Nios II Embedded Processor," Altera, http://www.altera.com/literature/lit-nio2.jsp.

[11] "Altera Floating Point Megafunctions," Altera, http://www.altera.com/products/ip/dsp/arithmetic/m-alt-float-point.html.

[12] "Virtex-5 APU Floating-Point Unit v1.01a," Xilinx, http://www.xilinx.com/support/documentation/ip_documentation/apu_fpu_virtex5.pdf.

[13] "Floating-Point Operator v5.0," Xilinx, http://www.xilinx.com/support/documentation/ip_documentation/floating_point_ds335.pdf.

[14] "Aeroflex Gaisler," http://www.gaisler.com/.

[15] "Create TIE Processor Extensions," Tensilica Inc., 2006, http://www.tensilica.com/products/xtensa-customizable/xtensa-lx/floating-point-2.htm.

[16] G. Ezer, "Xtensa with User Defined DSP Coprocessor Microarchitectures," in *Computer Design, 2000. Proceedings. 2000 International Conference on*, 2000, pp. 335–342.

[17] "ARM AMBA 2.0 AHB-APB Overview," ARM Ltd., 2005, http://www.arm.com/products/system-ip/interconnect/amba-design-kit.php.

[18] "ARM Cortex-R4(F) – ARM Processor," ARM Ltd., 2006, http://www.arm.com/products/processors/cortex-r/cortex-r4.php.

[19] "Xilinx MicroBlaze Soft Processor core," Xilinx, http://www.xilinx.com/tools/microblaze.htm.