Design of parallel vector/scalar floating point co-processor for FPGAs

E. Manikandan and K. Immanuvel Arokiia James Veltech Multitech Dr. Rangarajan Dr. Sakunthala Engineering College.

ABSTRACT
Current FPGA soft processor systems use dedicated hardware modules or accelerators to speed up data-parallel applications. This work explores an alternative approach of using a soft vector processor as a general-purpose accelerator. Due to the complexity and expense of floating point hardware, these algorithms are usually converted to fixed point operations or implemented using floating-point emulation in software. As the technology advances, more and more homogeneous computational resources and fixed function embedded blocks are added to FPGAs and hence implementation of floating point hardware becomes a feasible option. In this research we have implemented a high performance, autonomous floating point vector co-processor (FPVC) that works independently within an embedded processor system. We have presented a unified approach to vector and scalar computation, using a single register file for both scalar operands and vector elements. The FPVC is completely autonomous from the embedded processor, exploiting parallelism and exhibiting greater speedup than alternative vector processors. The FPVC supports scalar computation so that loops can be executed independently of the main embedded processor.

Introduction
Reconfigurable hardware bridges a gap between ASICs (Application Specific Integrated Circuits) and microprocessor based systems. Recently, there has been an increased interest in using reconfigurable hardware for multimedia, signal processing and other computationally intensive embedded processing applications. These applications perform floating point arithmetic computation for high data accuracy and high performance. Reconfigurable hardware allows the designer to customize the computational units in order to best match application requirements and at the same time, optimize device resource utilization. Because of these advantages, extensive research has been done to efficiently implement floating point computations on the reconfigurable hardware. Floating point (FP) computations can be categorized in three classes:
1. Software library
2. General purpose floating point unit
3. Application specific floating point data path

Embedded RAMs in FPGAs provide large storage structures. While the capacity of a given block RAM is fixed, multiple block RAMs can be connected through the interconnection network to form larger capacity RAM storage. A key limitation of block RAMs is that they have only two access ports allowing just two simultaneous reads or writes. The multiply-accumulate blocks, also referred to as DSP blocks, have dedicated circuitry for performing multiply and accumulate operations. These DSP blocks can also perform addition, subtraction and barrel shifter functions.

The major FPGA companies provide embedded cores, both hard and soft, for use with their processors. Altera has the Nios II soft core and Xilinx offers the MicroBlaze soft and PowerPC hard cores on their FPGAs. All these large embedded logic blocks make more efficient use of on-chip FPGA resources. However, they can also waste on-chip resources if they are not being used. In this work, we will explore the utilization of these embedded blocks on Xilinx Virtex FPGAs in implementing floating-point operations and vector processing.

Related Work
The implementation of a floating point unit in general purpose computing is extremely common but it makes an interesting case study for an FPGA based reconfigurable computing system. Up to now there have been many research efforts applied to the implementation of an FPGA based Floating point unit. This research can be categorized based on the type of communication with main processor, precision support, number of computation units and level of autonomy.

One of the earliest works in this area is done by Fagin et al.[1] They have implemented IEEE-754 standard compliant floating point adder and multiplier function on the FPGA for design space exploration. They found that the floating point unit substantially improves performance, but technology limitations made it difficult to implement floating point units at that time.

Recently, Pittman et al.[2] have implemented a custom floating point unit (CFPU) which is compliant with the IEEE 754 standard and improves floating-point intensive computation performance. The CFPU is implemented on the Xilinx Virtex FPGA based eMIPS platform which is partitioned into fixed and reconfigurable regions. They demonstrated various trade-offs for area, power and speed with the help of software profiling and run time reconfiguration for the CFPU.

The Xilinx’s MicroBlaze processor supports a single precision floating point unit in hardware that is tightly coupled to the embedded processor pipeline. The FPU implements addition, subtraction, multiplication and comparison. Floating point division and square root are available as an extension, as well as conversions from integers to floating point and vice versa. If the FPU is not instantiated, floating point operations are emulated in software. The Xilinx’s PowerPC hard core processor [3] has a floating point unit available that supports...
IEEE-754 floating-point arithmetic operations in single or double precision. Floating point instructions supported include add, subtract, multiply, divide, square root and fused multiply-add instructions. The FPU is tightly coupled to the PowerPC processor core with the Auxiliary Processing Unit (APU) interface [4]. The Xilinx FPU includes 32 floating point registers.

**Vector Processing Overview**

Most current microprocessors have scalar instruction sets. A scalar instruction set is one which requires a separate opcode and related operand specifications for every operation to be performed. Vector processors provide vector instructions in addition to scalar instructions. This chapter reviews vector processing in general, defines terms used in the rest of the thesis and lists recent trends in vector execution architecture.

**Vector Operation**

The code in Figure 2.1 is called SAXPY/DAXPY loop which forms the inner loop of the Basic Linear Algebra Subprograms library [42]. For the above code, A and Y are vectors and x is a scalar value. Each iteration of the SAXPY/DAXPY loop, performs below six steps.

1. Load element from vector A
2. Load element from vector Y
3. Load scalar value x
4. Multiply A’s element with x
5. Add result of multiplication to element of Y
6. Store result back in vector Y

For a scalar processor, these operations will be performed in a loop. A Vector processor provides direct instruction set support for operations on whole vectors i.e., on multiple data elements rather than on a single scalar value.

This vector instruction specifies operand vectors and a vector length, and an operation to be applied element-wise to these vector operands. Assuming the vector length is equal to the number of elements in each vector register then the SAXPY/DAXPY operation can be performed with just six instructions. Thus, vector operations reduce the dynamic instruction bandwidth requirement.

**Vector Memory and Vector Register Architecture**

There are two main classes of vector architecture: Vector Memory Architecture and Vector Register Architecture. In vector memory architecture such as the CDC STAR 100, operands are read from memory and results of the operation performed on operands will be stored back in to memory.

In vector register architecture such as Cray series, operands are read from vector registers and results of the operation performed on operands are stored back in the vector registers. Vector memory architecture has higher memory bandwidth requirement than vector register architecture.

**Vector Length Control**

A vector processor has a natural length determined by the number of elements in each vector register, which is called the Maximum Vector Length (MVL). It is highly unlikely that a given program will have vector length that equals MVL. The size of all the vector operations for SAXPY/DAXPY depends on “n”, which may not be known until run time. The solution is to create a vector-length register (VLR). The VLR controls the length of any vector operation. However, the value of VLR cannot be greater than the natural vector length of the processor.

**Vector Lane**

The vector lanes of a vector unit are shown in detail in Figure 2.2 Each vector lane has a complete copy of the functional units, a partition of the vector register file and vector flag registers. All vector lanes receive the same control signals and operate independently without communication for most vector instructions. With more vector lanes, a fixed-length vector can be processed in fewer cycles, improving performance. Processor which supports vector operations through parallel lanes is generally known as Single Instruction Multiple Data (SIMD) processors. Many popular microprocessors have extended instruction set architecture (ISA) to support SIMD instructions such as Intel SSE, MMX and PowerPC.

**Design Of Fpvc**

This describes the design and implementation of the Floating Point Vector Co-processor (FPVC) targeting Xilinx FPGAs. Three key features distinguish our work in floating-point architecture: a unified approach to scalar and vector processing, support for different latency of each functional unit and simplicity of organization. The initial section of this chapter provides an overall architecture of the processor. Next section describes the Instruction Set Architecture (ISA) features whereas the unified vector core is described in following sections. Finally, FPGA specific novel inter-lane communication features, vector compression and expansion and configurable parameters are described.

**Floating Point Vector Co-processor Architecture**

The Floating Point Vector Co-processor (FPVC) is a configurable soft-core vector processor architecture developed specifically for FPGAs. It leverages the configurability of FPGAs to provide many parameters to configure the processor for a specific application for desired performance and area. The FPVC instruction set supports both scalar and vector instructions with unified register files and execution units. Instruction set features are heavily borrowed from the instruction set of VIRAM and RISC processors such as PowerPC and Microblaze. Currently the FPVC does not support virtual memory and certain bit manipulation instructions, but it adds new instructions to take advantage of DSP functionality and embedded memories.

**III FPVC – The Vector Co-processor**

Some of the key design features of Floating Point Vector Core are:

- Completely autonomous from the main processor
- Supports single precision and 32-bit integer arithmetic operations
- 4 stage RISC pipeline for integer arithmetic and memory access
- Variable length RISC pipeline for floating point arithmetic
- Unified vector scalar general purpose register file
- Supports modified Harvard style memory architecture where there are separate level 1 instruction and data RAM but unified level 2 memory

**Local Instruction and Data RAM**

The FPVC implements a modified Harvard style memory system architecture. The FPVC’s memory system is divided in two levels: main memory and local memory. Main memory is connected to FPVC through master port of system bus interface whereas local memory sits in between. Apart from our approach, there are many options exist for connecting the FPVC to the main memory, such as through unified cache memory, separate instruction and data cache, through direct connection to main
memory etc. For off-chip memory, caches are used to hide the memory latencies, but for streaming applications in the area of embedded and scientific applications this may not be true. This range of different memory system configurations could be interesting to explore in the future.

**System Bus Interface**

The FPVC has one slave (SLV) port for communicating with the main processor and one master (MST) port for main memory accesses. The system bus interface for master and slave ports is not restricted to a specific bus protocol. The slave port interface can be connected to any type of bus including point-to-point, shared bus or simple glue logic.

In the current design implementation, we have implemented Processor Local Bus (PLB) [3] as the system bus interface. The PLB can be configured for 32-bit, 64-bit or 128-bit interface. Data alignment is also done in the system bus interface. The master port of the system bus interface includes a Direct Memory Access (DMA) controller to provide software prefetch mechanism for the vector core. DMA transfers setup include below three steps.

1. Write source/destination address of main memory in global address register
2. Write destination/source address of local memory in local address register.
3. Configure DMA transfer parameters: includes direction of the DMA, size of the DMA, which local memory involves in the transfer, state of DMA transfer are provided written in configuration register.

**Vector Scalar ISA**

The goals of the Vector-Scalar ISA(Figure 3.1) design are to be flexible and scalable and to provide a simple architecture suitable for a wide range of floating point applications. Due to trade-offs between performance versus design complexity, we selected to design new ISA which is inspired by RISC instruction architectures such as Power ISA [3], Microblaze ISA [2] and VIRAM vector ISA.

**Vector-Scalar Pipeline**

The FPVC is based on the classic dynamic scheduling (In-order issue and our-of-order completion) RISC pipeline. The four stages of the pipeline are Instruction Fetch, Decode, Execution and Write Back. The pipeline is intentionally kept short so integer vector instructions can complete in a small number of cycles to eliminate the need for forwarding multiplexers and to reduce area. Due to the short pipeline, floating point instruction spends most of their time in the floating point unit which optimizes the overall execution latency. As both scalar and vector instructions are executed from the same instruction pipeline, both type of instructions are freely mixed in the program execution and stored in the same local instruction memory.

As shown Figure 3.2, the fetch and decode stages are common to all instructions. The instruction fetch stage (IF) is used to access the instruction. During IF stage, the next instruction address is determined and sent to local instruction RAM, which returns the instruction by the end of the cycle. We assume that all instructions fit in the local instruction RAM. Therefore the size put a limitation on the size of program. An alternative approach is an instruction cache which would support larger programs at the cost of increased complexity. The fetch unit of the pipeline always assumes that branches are not taken and fetches an instruction from the next instruction address. Hence, the branch penalty for the vector co-processor will be two cycles.

During the decode stage (ID), instructions pass through three steps and all steps are performed in a single clock cycle. In the first step, the instruction is decoded and data hazard checkers perform checks to find out whether the current instruction's source and/or destination registers are in flight. Due to dynamic scheduling, not only RAW, (Read after Write) hazards exist but the data hazard checker must also check WAW (Write after Write) and WAR (Write after Read) hazards. Until all hazards are cleared, the Instruction Decode unit stalls the next instruction fetch.

Once all hazards are cleared, the instruction enters the vector state machine. If the instruction is a scalar instruction then the instruction state machine issues the instruction to the execution unit and the new fetched instruction enters the decode stage in next clock cycle. If it is a vector instruction, based on the maximum vector length stored in the MVL register, the instruction will be repeatedly issued to the execution unit.

As the instruction encoding can address 32 short vector registers as source/destination registers, the vector counter together with source/destination register references the current instruction’s operand data. Finally, in the last step, when the decode stage issues an instruction to the execution unit, it updates the scoreboard to keep track of in-flight instructions for data hazard checks. Once the instruction reaches the end of the execution unit, the result is written back to registers.

**Experimental Setup**

We have tested the FPVC for correctness as well as for area usage and speed. The FPVC is implemented in VHDL and synthesized using Xilinx ISE 10.1 CAD tools targeting Virtex-5 FPGAs. We have compared various FPVC configurations against a hard processor (PowerPC 440 with the Xilinx FPU Figure 4.1 using various linear algebra kernels. We have also compared area and speed for each configuration of the FPVC.

**Conclusion**

The completely autonomous floating point vector/scalar co-processor exhibits speedup on important linear algebra kernels when compared to the implementation used by most practitioners: embedded processor FPU provided by Xilinx. The FPVC is easier to implement at the cost of a decrease in performance compared to a custom datapath. Hence the FPVC occupies a middle ground in the range of designs that make use of floating point. The FPVC is completely autonomous. Thus, the PowerPC can be doing independent work while the FPVC is computing floating point solutions. The FPVC is configurable at design time. The number of lanes, size of vectors and local memory sizes can be configured to fit the application.

**References**


