CUSTOM FPGA-BASED MICRO-ARCHITECTURE FOR STREAMING COMPUTING

José Carlos Alves

FEUP/INESC-Porto
R. Dr. Roberto Frias, s/n
4200-465 Porto, Portugal

Pedro C. Diniz

INESC-ID (Tagus Park)
Av. Prof. Dr. Cavaco Silva
2780-990 Porto Salvo, Portugal

ABSTRACT

This paper describes a micro-architecture for a custom programmable FPGA-based processor, with direct support for streaming and vector computations relying on custom cache memory storage. The processor combines a custom data-path with several parallel data ports for accessing operands in streaming mode thus efficiently supporting nested looping constructs found in high-level languages while mitigating the impact on external memory bandwidth. The architecture leverages the strided access patterns of streaming data access using a microcoded sequencer with multi-dimensional nested looping capability. We present synthesis results for the main components of the architecture on a Xilinx's Virtex-4 FPGA device. The results reveal the architecture to be extremely flexible and consume few FPGA resources.

1. INTRODUCTION

FPGA devices have evolved from small regular arrays of fine-grain configurable logic blocks to complex user programmable devices, integrating in a single chip a rich variety of logic elements such as unconstrained logic blocks, memories and arithmetic multipliers. The increasing number of these elements in today's high-end FPGAs have allowed them to cater to the needs of more performance demanding scientific applications by directly supporting floating-point arithmetic processing [1].

As deeply pipelined implementation of these operators have become commonplace as FPGA soft-macros (*e.g.*[2]), they have enabled the development of multi-operation, vector-style custom data-paths using standard, or even non-standard, floa-ting-point formats. These data-paths with multiple arithmetic operators, possibly interconnected in a custom fashion, leverage the flexibility of FPGAs to build efficient, highly optimized computing cores that can support streaming computations directly from data ports and streaming-like vector operations over array data structures in memory. Although the proposed architecture does adequately support these two flavors of streaming operations, in the remainder of this paper we focus on the support for vector operations in memory as typically these cores act as *vector-operation* accelerators to a conventional processor.

The execution of multiple streaming/vector operators, however, exacerbates the processor-memory bandwidth bottleneck. To be effective, the hardware data-paths which include deeply pipelined functional units, require operands to be fetched in a stream-like fashion at the highest possible clock rate. This imposes tight throughput requirements for the memory system thus limiting the attained performance of current FPGA-based engines for scientific computing.

Unlike previous approaches to the implementation of custom vector units, we propose a micro-programmed controller and a parameterized local cache memory system. Although the approach is very general, the system proposed as a proof-of-concept is geared for present FPGA devices, exploring the feasibility of building independent and heterogeneous on-chip cache memory blocks to interface an external large memory or scratch memories for temporary data.

At synthesis time, the vectors used by the target application are assigned to individual data ports, allowing the designer (or in its place a compiler) to configure the best implementation for each data port, according to the requirements of the application (either a local buffer or a cache memory interface). Dedicated address generators associated with each memory block implement automatic vector iteration operations, simplifying the calculation of the memory effective address and reducing the length of the microinstruction field that specifies one data operand.

This paper makes the following contributions:

- It proposes a general micro-architecture for a vector processor, oriented for streaming applications.
- It presents design and implementation results for a microcoded sequencer capable of supporting the efficient address generation for nested `for(;;)` loops.
- It describes an address generation unit that implements iteration commands over vector indexes, based on simple post-increment/decrement commands

This work has been partially funded by the Portuguese Science and Technology Foundation (FCT - *Fundação para a Ciência e Tecnologia*) under grant number PTDC/EEA-ELC/71556/2006, and INESC-ID/INESC-Porto under it multiannual funding through the PIDDAC Program funds.

978-1-4244-8848-3/11/$26.00 ©2011 IEEE

- It describes the design of a parameterized cache memory unit using FPGA RAM blocks and presents implementation results for a Xilinx Virtex-4 FPGA device.

High-performance scientific and engineering computing is an area where memory structures such as the one described here will undoubtedly continue to play an important role given the continue need to data availability in key computational kernels. The flexibility of a micro-programmed system interface will, we believe, be a key enabling implementation technique to facilitate the mapping of high-level programing constructs to hardware.

The remainder of this paper is organized as follows. We briefly survey related work in the next section. In section 3 we describe our micro-architecture and its main components in the subsequent sub-sections. The microcode sequencer unit and the loop generation mechanism are described in section 4. Section 5 presents the novel micro-coded architecture of the vector address generator unit associated with each memory port. Section 6 details the organization of the multi-port memory system. We then conclude in section 7.

2. RELATED WORK

Given their regularity in terms of memory access patterns *vector* computations were recognized early in commercial super-computers machines [3] and in early DSP processors with support for vectorizing access modes.

Recently the notion of vectorization has been revisited in the context of processing-in-memory hardware architectures. The VIRAM architecture [4] is a scalable vector co-processor implementing a multi-lane processing core with a centralized vector register file. Its instruction set is geared for multimedia applications with a complete vector load-store unit defined as a co-processor extension for the 64-bit MIPS architecture. An evolution of this work [5] uses a clustered vector register file that distributes the vector registers defined in the ISA by different (physical) groups, thus reducing the data traffic among functional units and improving the performance of the VIRAM processor. However, as these architectures were designed as processor cores to be integrated in ASIC designs, they do not provide special support for instance-specific customization targeting FPGAs.

Various FPGA-based vector units have been proposed to extend the computing power of soft processors embedded in FPGA devices. Jason *et. al* [6] describe a parameterized vector co-processor for the Altera Stratix III FPGA family, which can be configured at design time, by selecting several parameters such as the number of vector lanes, the processor data width or the organization of the local memory. The scalar core is a NIOS-II compatible soft-processor and the vector instruction extension set is similar to the one used in VIRAM. Other authors (*e.g.* [7]) have developed custom, yet programmable address generation units for FPGA-based designs that exploit strided access modes in particular for stride-1 accesses as is common in vector operations.

Ang *et. al*[8] describe a flexible multi-ported caching architecture for FPGA-based computing systems where a single cache can feed a multi-ported data-path with asynchronous data memory accesses to the various ports of this cache. Internally it divides the caches into sub-cache memories and uses an arbitration logic for coordinating the accesses to the sub-caches.

Other hybrid architectures (*e.g.*, VESPA [9]), combine a MIPS-based scalar soft-core with a configurable vector co-processor, supporting the same instruction set defined for the VIRAM architecture. Although this was designed for FPGA implementation, it does not make specific assumptions on specific aspects of each FPGA family, as the block RAM size or its usable aspect ratio.

The work described in this paper differs from these efforts in several aspects. The work that has focused on internal storage structures for FPGA-based computing has leveraged the variety of internal memories such as BRAM blocks and offered a static (albeit very flexible) mapping of data to storage (see *e.g.* [10]). Although several recent approaches are very similar in spirit to the work presented here, namely the work by Ang *et. al* [8] our approach focuses on a micro-coded address generation unit that is more flexible than previous programmable implementations. Besides, offering the illusion of a cache of multiple distinct resources (possibly heterogeneous), rather than combining homogeneous storage resources in a single address space abstraction, offers increased flexibility for best utilizing a scarce FPGA resource that are the internal RAM blocks.

3. A PROGRAMMABLE MICRO-ARCHITECTURE FOR STREAMING COMPUTING

This paper proposes a micro-architecture for a programmable custom processor dedicated to streaming computations on FPGA-based platforms. The processor combines a custom data-path, synthesized from core computations identified in the application's basic blocks, with several parallel data ports for accessing operands from local memories and a microcode sequencer with streaming and nested looping capabilities. As mentioned previously we focus on the implementation details of effectively supporting vector-like operations over array variables mapped to external or internal memories. The micro-architecture and its implementation, nevertheless, also supports streaming execution directly from its data ports.

Each memory access port provide operands to the data-path inputs, coming from on-chip memories (either temporary buffers for local storage or heterogeneous cache blocks). Associated to a memory port there is a custom address generator that translates references to vector elements into its effective memory address, supporting strided addressing of

array data. As vector access patterns commonly observed in applications can be generated by simple iteration commands on its indexes (like the post increment/decrement operators in the C programming language—`i++`), this can be done efficiently by custom logic that translates the memory address calculation to additions and subtractions with constants precomputed at synthesis time.

Given a target streaming application a set of vectors can be identified as the sources and targets of data during the several stages of computation. Each computing core (commonly supporting the execution of one or more nested loops) will require specific data-paths which, typically, consume two or more operands and produce one (or more) result. If these operands are provided in parallel from independent data lanes, a pipelined data-path can only be clocked at the data read rate. This is usually the limiting factor of the execution speed, imposing more restrictive constraints than the pipelined data-path that executes the operations. Data reuse can be exploited by using conventional cache memories associated to each memory port or custom memory organizations specifically designed for each computing core (for example FIFO buffers or stacks). While the bufer/stack option generally leads to the best memory utilization, the various computing cores may require several different memory organizations, which may be impractical to simultaneously satisfy. Run-time reconfiguration to reuse the same physical memory is also typically not an option. On the other hand, while conventional memory caching does not exploit any particular knowledge of the application, it provides an efficient, albeit not optimal, way to exploit the data locality exposed in different cores.

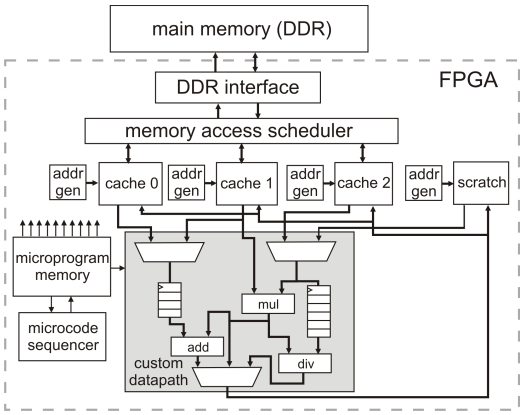

Fig. 1. Organization of the custom streaming processor.

We propose to assign to each memory port a set of conventional cache memories which may be independently configured (in terms of its size, aspect ratio and organization) according to the needs identified for each particular memory port. By associating to each memory port a set of data variables (in general vectors) the cache block that interfaces each port should be configured according to the access patterns identified for its set of variables and also the available amount of on-chip memory. In addition, the vectors providing data to each core should be assigned to different memory ports in order to maximize the memory read bandwidth. These decisions can be derived from an application's source code, possibly annotated with design constraints (for example the maximum number of memory ports allowed) and hints to guide the synthesis of the custom processor. Figure 1 illustrates the general organization of the streaming processor for a configuration with three cache memories and one scratch memory, whose components we describe next.

4. MICROCODED CONTROLLER

The microcoded controller runs a stored (encoded) microprogram thus allowing easy reconfiguration of the control section and facilitating the compilation process from high-level software specifications. The custom microcode includes three different sections, namely loop control, operand specification and data-path configuration.

4.1. Loop control

The loop control component of a design supports the execution of nested loops with fixed iteration limits for a number of loops in a program that is constrained only by the size of the memory used to store the loop parameters (number of iterations of each loop). The looping control is done with a single 2-bit field in the instruction word that specifies if an instruction is a non-loop instruction (code 00_2), start loop (code 01_2) or end loop (code 10_2). The additional 2-bit code 11_2 is used for additional looping instructions such the immediate transfer to the next or last iteration of the inner-most loop and thus akin to the `break` and `continue` statements of the C language. Figure 2 shows an example of the part of the microcode that controls this unit.

Looping with symbolically constant but compile-time unknown is also supported for a specific limit of maximum number of iterations. A register holds the value of the number of iterations at run-time and looping is controlled by testing for a non-zero condition the value of this register.

The use of the code 11_2 with the semantics of the `break` statement is used to support the execution of streaming operations. In this case the implementation uses a predefined streaming termination data-value, say the binary zero value or -1 or any other stream-specific value that will make a termination predicate evaluate to true. The code example in Figure 3 illustrates the translation of a streaming *add* example where the output stream is the addition element-wise of two input streams. The figure also depicts the micro-coded instructions of our controller where reading and writing of the data values is always done to the same address which is

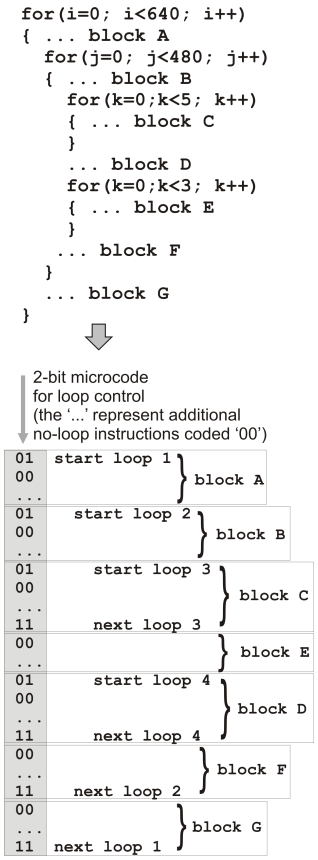

Fig. 2. Implementation example of the looping instructions in the microcode. An additional ROM table is necessary to store the constant loop limits.

I/O mapped rather than memory-mapped as is the case of the examples with memory-mapped vectors.

The implementation of this *forever* loop uses a *null* iteration count as termination is achieved by a `breakif` instruction. The `breakif` instruction receives a logical hardware signal from a comparator module assigned to the input data ports and terminates the loop if the condition (in this example when one of the two inputs is equal to zero) synthesized from the high-level source code is true. Line `0` of this microcode sets the address generators attached to three different memory ports to select pre-assigned vectors A, B and C assuming these are mapped into input and output external ports; line `1` starts an infinite loop and reads data in parallel from the two input ports; the `break` instruction terminates the infinite loop, when a boolean signal, coming from a comparator that evaluates the end of loop condition, is true; last instruction writes the data-path result to the output vector C and hence to the corresponding external output port.

Regarding the implementation of this micro-controller structure, the bit-field that specifies the looping is concate-

```
while(1){
  tmp_a = getData(port0);
  tmp_b = getData(port1);
  if((tmp_a == 0 ) || (tmp_b == 0))
    break;
  tmp_c = tmp_a + tmp_b;
  putData(port2,tmp_c);
}

// Streaming microcode
0:  Noloop;    SelectIn A;  SelectIn B;  SelectOut C;
1:  Loop 0;    Read A;      Read B;      Nop;
2:  Breakif;   Nop;         Nop;         Nop;
3:  Next;      Nop;         Nop          Write C
```

Fig. 3. Streaming example and corresponding microcode.

nated with the other application dependent fields of the microinstruction (operand read/write and address iteration). The micro instruction sequencer is implemented by a hardware template of a finite state machine that is customized at synthesis phase with the following information: (i) maximum number of loops found in the application, (ii) maximum number of nested loop levels and (iii) maximum number of iterations in each nested loop level. A table with the number of iterations for each loop is addressed by a *loop index* that is the order of appearance of each loop in the code. This table can be maintained in a local RAM loaded at run time (this enables loops with variable iteration limit) or can be built with a ROM defined at hardware synthesis time. The complexity of the hardware for this block does not depend significantly on the absolute number of loops found in a program but rather on the maximum number of nested loops.

Table 1 summarizes implementation results for a selected sample of configurations of the loop control unit on a Xilinx's Virtex4LX80-11 FPGA synthesized using XST with optimization goal 'speed-high'. All the configurations achieve a clock frequency above 200 MHz where the maximum number of nested loops supported is the parameter that most affects the maximum clock speed.

max iter.	max inst.	# loops	# nested loops	LUTS / FFs
255	512	7	3	194 / 48
255	512	15	7	329 / 83
255	1024	31	7	338 / 83
1023	512	7	3	258 / 54
1023	512	15	7	350 / 92
1023	1024	31	7	364 / 95

Table 1. Synthesis results for the loop control unit. (excluding table that holds the number of iterations of each loop).

5. ADDRESS GENERATION

Each microinstruction specifies a set of operands to be applied to the inputs of the data-path. Rather then including the effective memory address of each operand in a microinstruc-

tion the implementation accesses the operands using address generators closely coupled to each memory port. Each address generator builds the effective memory addresses for one or more vectors assigned to its memory port through short iteration commands, thus reducing the length of the microinstruction control field. Although several vectors may be assigned to the same address generator (and thus to the corresponding memory port), only one vector is active at a time. We can thus use as input to a custom data-path elements from different vectors that are read in parallel from distinct memory ports, thus increasing the data access bandwidth. Also, memory ports can even be implemented as scratch-pad memories holding temporary data from previous computations or as heterogeneous cache memories, with different organization, sizes and aspect ratios. The set of cache and scratch pad memories provide a set of independent read and write data lanes to a custom computation data-path and include application specific address generation units.

3-bit command	action
000	keep current index
001	post-increment index by 1
010	post-decrement index by 1
101	post-increment index by a constant stride
110	post-decrement index by a constant stride
011	set index to its minimum
100	set index to its maximum
111	reserved

Table 2. Iteration commands for each index of a n-dim. vector (constant stride is defined at synthesis time).

Once determining (at compile time) which application vector is assigned to each memory port, the base addresses, number of dimensions and sizes along each dimension are embedded in custom address increment/decrement operators. A simple 3-bit command for each vector index implements the operations listed in table 2 and an additional 2-bit command specifies the operation to execute with that vector element, as presented in table 3 (the "set current vector" command uses as argument the bits normally used for the index iteration command to specify the vector to make the current active vector.

The address generation units realize the abstraction of vectors over the arrays by iterating over the indexes of the arrays which can be configured for either row-major or column-major organization.

Memory address generators can also support a vector mapped to an external input/output port in a pure streaming mode to implement read and write operations, respectively.

2-bit command	action
00	set current vector
01	read data
10	write data
11	write after read

Table 3. Commands for accessing a vector element.

Once set as the active vector, this mechanism provides support to embed a true streaming computing core, reading data from external buses and writing results to external outputs. When the active vector of an address generator is set to an external stream, the iteration commands used to iterate the indexes of vectors (for memory-stored vectors) are ignored.

6. HETEROGENEOUS MEMORY SYSTEM

The proposed cache memory system is geared towards vector processing units that require data, in streaming or sequentially strided mode from an external (and possibly slow) dynamic memory. Internally, the cache system can be composed of various independent cache memories with different, and configurable aspect ratios (depth or number of lines) and width (number of bytes per line) sharing a common access to an off-chip DDR2 dynamic memory. In addition, the memory infrastructure can also be configured with various scratch-pad memories holding temporary data and thus avoiding communication with the main memory.

The design has been specialized for FPGA devices with internal block RAMs , in what respects to the size and depth of the cache memories and for the DDR2 data interface. The cache memories are direct-mapped and the default data-path width is 32 bit (single precision floating point).

6.1. Direct-mapped cache memories

The cache memories are configured from a parameterized HDL templates to code synchronous dual-port RAM memories, in order to instruct the Xilinx synthesis tool (XST) to map those memory structures into block RAMs (BRAM). For a specific FPGA device family, for example the Virtex4$^{\mathrm{TM}}$, and given that the size of the BRAM primitives is 18 Kbit, each cache block should be, ideally, specified with a size multiple of this value (or, if rounding up to the nearest power of two, 16 Kbit or 2 KByte).

As the read/write transfers with the DDR2 memory interface works with two consecutive blocks of 128 bits, the cache line size must be defined as a multiple of 128 bits (4 single-precision floating-point numeric values). This requires the utilization of a minimum of 4 BRAMs to support a 128 bit data bus and thus the optimal utilization of the BRAM resources is any combination of number of cache lines vs. line size that sums to multiples of 2 KByte.

The tag memory is built with distributed BRAM implemented with the FPGA's look-up tables. This enables asynchronous reads from this memory to determine the hit/miss condition, needed to allow a read on hit within a single clock cycle. While the size of the cache line has little direct impact in the logic complexity of the controller, the number of cache lines impacts almost linearly in the logic complexity of the system, due to the tags.

Regarding coherency issues between the various caches, it is up to the compiler (or in its absence the programmer) to enforce it by requiring for instance the data streams to always be disjoint or when one data-path input writes data is done so by the storing the data in the same cache before any other read on that data is performed.

6.1.1. Cache implementation

Table 4 presents synthesis results for a selected set of direct-mapped cache memories for 16 cache lines. As expected the number of BRAMs required scale linearly with the desired capacity of the memories. The percentage of the resources used is fairly small, but most importantly, the variation of the clock rate is minimal.

cache capacity			synthesis results		
num. lines	line size (bytes)	cache size (bytes)	BRAMs	LUTs	Freq. (MHz)
16	512	8192	4 (2%)	388 (0.5%)	94.0
16	1024	16,364	8 (4%)	388 (0.5%)	94.4
16	2048	32,768	16 (8%)	389 (0.5%)	94.5
16	4096	65,536	32(16%)	366 (0.5%)	92.4
16	8192	131,072	64(32%)	264 (0.5%)	89.9

Table 4. Synthesis results of a direct-mapped cache memory with 16 lines on a Virtex4LX80-10 FPGA.

6.1.2. N-way set associative cache

The parameterized cache memory model was extended to implement N-way set associative cache memories (currently limited to $N = 2$, 4 or 8). The policy implemented for the replacement of cache blocks is LRU (least recently used), based on a set of *age registers* associated to each tag. Because it is necessary to know which of the entries is older, for a given set, it is sufficient to use $log_2(N)$ bits to track the age of each entry (2 bits for a 4-way set-associative memory).

A cache hit on a tag entry that is not the last recently used, sets the corresponding register to the maximum value (all ones) and decrement by 1 all the other registers, (thus *aging* them and saturating at zero). Subsequent hits to the same cache line do not modify these registers, to avoid the rapid aging of the other cache entries. When a cache miss happens, the cache line to be replaced is the one with the minimum value (all zeros) in its history register.

7. CONCLUSIONS

We described the organization of a custom micro-coded processor synthesized from high-level vector and streaming application specification and presented implementation results for its core components. A micro-programmed controller supports the implementation of nested loops with compile time or symbolically constant bounds, using a 2 bit field microinstruction and a small ROM to store loops parameters. Implementation results on a contemporary FPGA device reveal that the proposed architecture to be very compact while attaining respectable clock rates.

8. REFERENCES

[1] K. Underwood and S. Hemmert, "Closing the Gap: CPU and FPGA Trends in Sustainable Floating-Point BLAS Performance," in *Proc. of the 12th Annual IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM'04)*, 2004, pp. 219–228.

[2] *Floating-Point Op. v3.0 - Product Spec. (DS335)*, Xilinx, Sep. 2006.

[3] R. Espasa, M. Valero, and J. Smith, "Vector architectures: Past, present and future," in *Proc. of the 2nd Intl. Conf. on Super Computing*, July 1998, pp. 425–432.

[4] C. Kozyrakis and D. Patterson, "Scalable, vector processors for embedded systems," *Micro, IEEE*, vol. 23, no. 6, pp. 36–45, Dec. 2003.

[5] ——, "Overcoming the limitations of conventional vector processors," in *Proc. 30th Annual Intl. Symp. on Computer Architecture (ISCA'99)*, 2003, pp. 399–409.

[6] J. Yu, G. Lemieux, and C. Eagleston, "Vector Processing as a soft-core CPU Accelerator," in *Proc. of the 16th Intl. Symp. on Field Programmable Gate Arrays (FPGA'08)*, 2008, pp. 222–232.

[7] J. Park and P. Diniz, "Synthesis of pipelined memory access controllers for streamed data applications on FPGA-based computing engines," in *Proc. of the 2001 Intl. Symp. on System Sysnthesis (ISSS'01)*, 2001, pp. 221–226.

[8] S.-S. Ang, G. Constantinides, P. Cheung, and W. Luk, "A Flexible Multi-port Caching Scheme for Reconfigurable Platforms," in *Proc. of the 2006 Intl. Symp. on Applied Reconfigurable Computing (ARC'06)*, 2006, pp. 205–216.

[9] P. Yiannacouras, G. Steffan, and J. Rose, "VESPA: portable, scalable, and flexible FPGA-based vector processors," in *Proc. of the 2008 Intl. Conf. on Compilers, Architectures and synthesis for embedded systems (CASES'08)*, 2008, pp. 61–70.

[10] N. Baradaran, J. Park, and P. Diniz, "Data Reuse in Configurable Architectures with RAM blocks: Extended Abstract," in *Proc. of the 2004 Intl. Conf. on Field Programmable Logic and Applications (FPL'04)*, 2004, pp. 1113–1115.