A LOW COST RECONFIGURABLE SOFT PROCESSOR FOR MULTIMEDIA APPLICATIONS: DESIGN SYNTHESIS AND PROGRAMMING MODEL

Sai Rahul Chalamalasetti [1], Wim Vanderbauwhede [2], Sohan Purohit [1], Martin Margala [1]
[1]Electrical and Computer Engineering, University of Massachusetts Lowell, Lowell, MA, USA
[2]Department of Computer Science, University of Glasgow, UK
sairahul_chalamalasetti@student.uml.edu, {sohan_purohit, martin_margala}@uml.edu
wim@dcs.gla.ac.uk

ABSTRACT

This paper presents an FPGA implementation of a low cost 8bit reconfigurable processor core for media processing applications. The core is optimized to provide all basic arithmetic and logic functions required by the media processing and other domains, as well as to make it easily integrable into a 2D array. This paper presents an investigation of the feasibility of the core as a potential soft processing architecture for FPGA platforms. The core was synthesized on the entire Virtex FPGA family to evaluate its overall performance, scalability and portability. A special feature of the proposed architecture is its simple programming model which allows low level programming. Throughput results for popular benchmarks coded using the programming model and cycle accurate simulator are presented.

1. INTRODUCTION

Adaptable architectures capable of processing large amount of data in parallel are increasingly becoming popular as low-cost, flexible solutions for media processing and other applications. This has resulted in an ever increasing interest in low cost, high throughput reconfigurable architectures, in recent times. Architectures that provide reasonably high throughput at extremely low cost and low power are being seen as key players for media processing applications. Traditionally, FPGAs have been considered ideal contenders in this category due to their ability to deliver high throughput at relatively lower costs than dedicated DSP ASICs. However, FPGAs only offer bit level granularity, resulting in a large routing overhead, thus decreasing overall system throughput and silicon efficiency. Coarse Grained Reconfigurable Architectures (CGRAs) provide high speed parallel computations with lower routing and configuration overheads. As a result several CGRAs like MATRIX[1] MorphoSys[2], and the new AsAp[3] were proposed to provide extremely high throughput parallel processing performance. Although these solutions offer exceptional performance, they come at a high cost, employing millions of transistors, consuming large amounts of power and using complex programming models.

In spite of the throughput and efficiency lost out by FPGAs due to the routing overheads, the generic architecture of the FPGA makes it a low cost solution with reduced time to market. As an alternative to application specific custom IP which increase manufacturing cost and time to market, soft processors provide a more generic platform to implement various design algorithms. Recently, several soft processing cores have been introduced to map onto the FPGAs and function as complete 8/16/32 bit RISC processors. These soft cores allow for a more high level programming style for the devices rather than using the hardware description languages. This allows the programmer to program the various algorithms in his native programming language, rather than model them using a hardware description language (HDL). Soft processors like Pico Blaze, Micro Blaze, serve this category of applications.

However, these soft processing solutions employ a generalized RISC architecture, not entirely optimized for media processing needs. Media processing applications are continuously increasing in their complexity. The current available soft processing units are limited by a general lack of reconfigurability which renders them somewhat non-feasible for multimedia processing applications which require the architecture to employ a high level of parallelism. This inability to extract parallelism out of the algorithm could prove to be a huge bottleneck when implementing media processing algorithms on these platforms. A highly parallel, easily programmable soft core solution will therefore be viable for media processing tasks.

We recently proposed MORA[5,6], a coarse grained reconfigurable architecture for multimedia processing. The MORA architecture aims to introduce resource utilization and programming flexibility as equally important parts of the design philosophy for reconfigurable platforms. In this paper, we propose the architecture for a simple, yet efficient 8bit reconfigurable DSP style processor to be part of MORA, a coarse grained reconfigurable array for media processing applications. The reconfigurable cell (RC) was implemented in VHDL and synthesized on the entire Virtex family of FPGAs to demonstrate performance, cost and portability of the proposed architecture Using the MORA assembly language and cycle accurate simulator, the architecture was evaluated for popular benchmark

978-1-4244-3892-1/09/$25.00 ©2009 IEEE

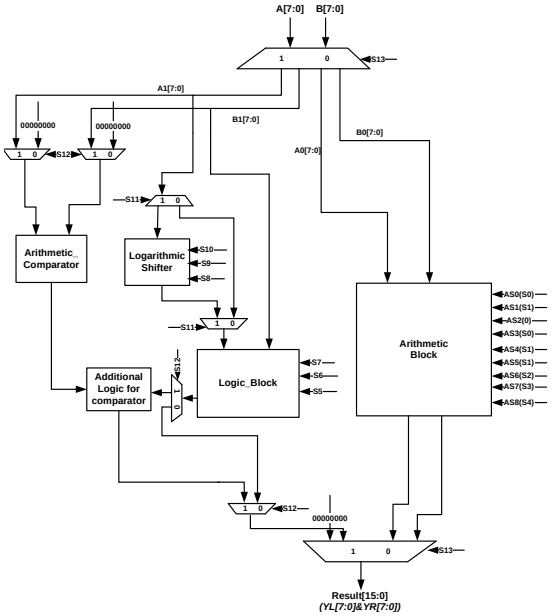

Fig 1. Processing Element Architecture

algorithms. This study also helps evaluate the feasibility of the proposed architecture as a soft processor.

2. MORA ARCHITECTURE

The MORA architecture consists of a 2-D array of identical Reconfigurable Cells (RC) arranged in 4X4 quadrants and connected through a hierarchical reconfigurable interconnection network. Storage for data is partitioned among RCs by providing each RC with internal data memory. Each individual RC is a tiny Processor-in-Memory (PIM)[8]. Every RC consists of an 8bit Processing Element, 256X8 Dual Port Data Memory, and a central controller for overall synchronization.

2.1. Processing Element

The Processing Element is the main computational unit of the RC. Prior work on the design of data paths, focused on optimizing the data path design and organization for efficient single cycle arithmetic operations [7]. Fig. 1 shows the organization of the PE. It includes the signed arithmetic data path [7] along with additional blocks for shifting and comparison operations. The PE uses a logarithmic shifter to implement bitwise shifting operations on the operands. The shifter working in conjunction with the logic block provides support for both round shifting and shift out operations. The arithmetic data path is organized to provide single-cycle addition, subtraction and multiplication operations. The PE also provides two sets of registers at the input and output to enable accumulation style operations, as often required for media processing applications.

2.2. Control Unit

The control unit provides the handshaking signals between memory and data path, and ensures that the two units work in perfect sync with each other. The unit consists of a 16-word instruction memory, three address generators, instruction decoders and instruction counters. The instruction word is 92 bits wide and encodes the operation, base addresses for an instruction operands and output data set and address offsets for traversing through memory, as well as the number of times a specific operation is to be performed. The address generator accepts four data fields: *Base address*, *Step*, *Skip* and *Subset*. The *Base address* is initially loaded into the address generator, and depending on the values of *Step*, *Skip* and *Subset*, the address of the next memory location to fetch the data is calculated. The three fields allow the controller to move anywhere throughout the available data memory. The address generator thus generates the range of addresses over which a given instruction is to be performed. The control unit provides support for a total of 28 arithmetic, logic and memory based instructions. Fig. 2 shows the basic organization of the control unit.

2.3. Memory Organization

Each RC is provided with a 256×8 bits data memory. This allows each RC to work as a tiny PIM [8], i.e. operations are performed close to memory. The approach allows each RC to work independent of the others, and eliminates the possibility of contention for memory resources between RCs, thus also bypassing the need for special contention resolving logic. The result is an optimized cell performance in terms of power, area, memory access time, and reduction in complexity of the interconnect switches. In addition to individual read and write operations on the ports, an additional MOVE instruction is provided for port-port data

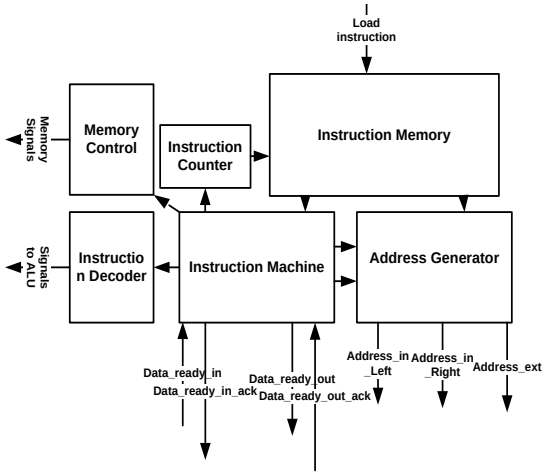

Fig 2. Control Unit Design

transfer during matrix and vector operations. The read and write operations are performed on opposite phases of the clock signal. This allows the RC to perform the MOVE operation in a single cycle. The alternately phased read and write allow the RC, to perform the read-execute-write sequence in a single internal clock cycle. As a result, each of the 28 instructions takes exactly one clock cycle to complete. This ability is particularly useful in that it allows the MORA processors to treat each algorithm, as a sequence of simple single cycle instructions. The architecture thus, prevents itself from imposing restrictions on programming style or favoring a particular algorithm, and allows for a simple programming model, with a great degree of freedom for the programmer.

3. PROGRAMMING MODEL

3.1. Processing model

The MORA RCs operate asynchronously, using a simple handshake mechanism to notify downstream RCs of availability of data. Synchronous operation of all RCs would require that at every moment every RC would execute the same number of clock cycles for an operation and that the clocks would need to be synchronized over a very large area. Having to fulfill both requirements would be extremely unpractical and inefficient.

The RCs can receive data via two input ports A, B and transfer processed data via two output ports YL, YR. Each of these output ports can be connected to up to two RCs. The RC has two states: waiting and processing; these states depend on the state of the RC's local memory and the states of the memories of the RCs connected to its output ports. The RC memory has two states: ready and not ready. The local memory is ready when it has received a "ready" signal from all RCs connected to its input ports (unconnected input ports are always ready). The RC will process data if its local memory is ready and all of the memories of the connected RCs are not ready (unconnected output ports are always not ready).

3.2. Co-design of RC and Assembly Language

To ensure that MORA can be programmed efficiently, the RC and the assembly language were co-designed from an early stage. The design of the assembly language informed in particular the choice of non-arithmetic instructions in the instruction set, the address generator design and the virtual register/ virtual memory bank system.

The MORA "assembly" language consists of three components: a coordination component which allows expressing the interconnection of the RCs in a hierarchical fashion, an expression component which corresponds to the conventional assembly languages for microprocessors and DSPs and a generation component which allows compile-time generation of coordination and expression instances.

3.3. Expression language

The MORA expression language is an imperative language with a very regular syntax similar to other assembly languages: every line contains an instruction which consists of an operator followed by list of operands. The main differences are:

- *Typed operators:* the type indicates the wordsize on which the operations is performed, e.g. bit, nybble, byte, short, long (resp. B, N, C, S, L).
- *Typed operands:* operands are actually tuples indicating not only the address space but also the data type, i.e. word, row, column, or matrix and the scan direction (forward or reverse)
- *Virtual registers and address banks:* MORA has no directly accessible registers. Operations take the RAM addresses as operands; however, "virtual" registers indicate where the result of an operation should be directed (RAM bank A/B, output L/R)
- *All arguments are optional:* the MORA assembler will infer defaults for non-specified arguments, considerably simplifying the most common instructions.

3.3.1. Instruction structure

An instruction is of the general form

```
instr ::= op nops? dest? opnd*
op ::= uop:(B|N|C|S|I|L)?
dest ::= virtreg? addrtup?
opnd ::= addrtup|const
virtreg ::= Y|YL|YR|YA|YB
addrtup ::= (ram_id:)?addr(:type)?
ram_id ::= A|B
addr ::= 0..(MEMSZ-1)
type ::= (W|C|R|M|MT|Q|QT)(R|F)?
const ::= C:num
num ::= -(MEMSZ/2-1)..(MEMSZ/2-1)
```

For example, the instruction for signed addition of two bytes would be:

```
ADD 1 Y A:0:W A:0:W B:0:W
```

However, because of the "reasonable defaults" strategy, this can simply be written as

```
ADD
```

Similarly, a multiply-accumulate of the first row of an N×N-matrix in bank A with the first column of a matrix in bank B would in full be

```
MULACC 8 Y A:0:W A:0:R B:0:C
```

but can simply be written as

```
MULACC R C
```

The MORA assembler will infer defaults for all implicit fields.

3.3.2. Address Types

As discussed in Sec. 3, the RC supports complex address scan patterns through the use of 4 fields in the instruction word: base_address, step, subset and skip. The MORA assembler supports a subset of all possible values of *Step*, *Subset* and *Skip* through its type system. The type component of the address tuple (W|C|R|M|MT) indicates the nature of the datastructure referenced by the base address *(ram_id*:*addr):*

W: word (single byte)

C: Column (*N×1*)

R: Row (*1×N*)

M: *N×N* matrix (MT: transposed matrix M^T)

Q: *N/2×N/2* matrix (QT: transposed matrix Q^T)

The type suffix (F|R) indicates a forward or reverse scan direction. Thus MORA's simple address type system supports the typical vector operations required for N×N matrix manipulation.

3.3.3. Operation Types

The operator of an instruction can be explicitly typed, indicating the length of the word on which the operation should be performed. This information is used to generate the step and the virtual output register. As the MORA RAM is byte-addressable, operation types B (bit) and N (nybble) have no effect on the address generation but result in single-byte output; operations on multiple bytes (types S and L, resp. 2 and 4 bytes) result in a step of the number of bytes; the assembler generates the individual byte-operations that make up the multi-byte operation.

3.4. Coordination Language

MORA's coordination language is a compositional, hierarchical netlist-based language. The language consists of primitives definitions, module definitions, module templates and instantiations. Primitives describe a MORA RC and are defined as prim_name { ... }, e.g. a primitive to compute a determinant of a matrix would be:

```
DET2x2 {
  MULT YB B:0 A:0 A:9
  MULT YB B:1 A:1 A:8
  ADD YR A:0 B:0 B:1
}
```

Instances are defined as (net_{out1} ,...) = name (net_{in1} ,...); unconnected ports are marked with a '_'.

Modules are groupings of instantiations, very similar to compositional coding in VHDL. As modules can have variable numbers of input and output ports (but no inout ports), the definition is module_name (inport1,inport2,...) { ... } (ouport1,outport2,...). For example, a module to compute 16-bit addition can be built out of 8-bit addition primitives (ADD8) as follows:

```
ADD16 (b1,b0,a1,a0) {
  (c0,z0) = ADD8 (b0,a0)
  (c1,s1) = ADD8 (b1,a1)
  (_,z1) = ADD8 (c0,s1)
} (c1,z1,z0)
```

3.5. Generation Language

This component of the language is in itself an imperative mini-language with a simple and clean syntax inspired mainly by Ruby [9]. The language acts similar to the macro mechanism in C, i.e. by string substitution, but is much more expressive.

The current MORA RC does not support registered memory access and hence addressing is completely static. While this is not an issue for run-time performance, it would make algorithm implementation repetitive and cumbersome. The generation language allows instructions to be generated in loops or using conditionals. As an example, consider matrix multiplication. Because of the parallelism in MORA, 8×8 matrix multiplication can be done very efficiently by splitting the matrices into 4×8 and perform 4 partial multiplications in parallel.
The C code for such a partial multiplication is:

```
for (int i=0;i<4;i++) {
  for (int j=0;j<4;j++) {
        m[i][j]=0;
        for (int k=0;k<8;k++) {
              m[i][j]+=a[i][k]*b[k][j];
        }
  }
}
```

In MORA assembly, this becomes:

```
for j in 0..24 step 8
   for i in 0..3
      out=i+j
      k=i*8
      MULACC A:out:W A:j:R B:k:R
   end
end
```

The MORA RC performs this computation in 128 clock cycles.

3.6. Implementation

The MORA assembly language was implemented in an assembler combined with a cycle-accurate interpreter. The assembler first generates the full assembly text by

Table 1. Performance of soft-MORA processor on Virtex 5 FPGA for benchmark applications

Benchmark	FPGA type	Delay (ns)	Latency (ns)	# Samples in parallel	Throughput (MOPS)	Utilisation (#RCs)
8×8 2-D DCT, (minimum delay)	V5	1,054	527	2	0.94	112
8×8 2-D DCT, (max. throughput)	V5	3,749	3,749	20	1.17	160
4×4 2-D IT H.264, (1 block)	V5	264	132	2	0.94	112
4×4 2-D IT H.264, (8 blocks)	V5	469	469	20	1.17	160
32×32 DWT LeGall (5,3)	V5	35,613	31,864	40	1.17	160

evaluating the generation language; it then compiles the instruction words from the expression language part of every primitive definition; the connectivity of the RCs is extracted from coordination language. The actual placement and routing of the RCs is not handled by the MORA assembler: the final output of the assembler is a VHDL netlist which is processed using the FPGA toolchain.

4. FPGA SYNTHESIS AND PERFORMANCE RESULTS

The RC was synthesized on the Virtex 5 XC5VL330 FPGA from Xilinx [4]. Since MORA is array based architecture, we targeted high end FPGA devices to allow implementation of extra cores on the FPGAs. The synthesis results show that the RC working at 68.29MHz occupies only a fraction of the total FPGA resources and it is possible to map an estimated 160 RCs, which is approximately equivalent to mapping 3standard MORA arrays (64 8bit RCs) on a single Virtex 5.

Using cycle accurate simulator, we evaluated the MORA architecture for throughput and other performance metrics while performing popular benchmark applications. Table 1 presents the benchmark performance evaluation of the MORA soft cells. In each case, the algorithm has been mapped onto the maximum number of RCs available for the FPGA device at hand. This guarantees maximum parallelization of the task at hand by maintaining a high percentage of resource utilization.

5. CONCLUSION

This paper has presented the FPGA synthesis results for MORA, a coarse grained soft processing core. The core was synthesized on the Virtex 5 XC5VL330 FPGA from Xilinx. Synthesis results show that the core occupies only a fraction of the FPGA's resources, thus making it possible to integrate up to 3 arrays of the proposed architecture on a single Virtex 5 device. Throughput and utilization results for benchmark applications show that the proposed architecture and compiler allow mapping of algorithms for maximum throughput while at the same time ensuring near 100% utilization of all available MORA processors on nearly all the FPGAs. This makes a strong case for the portability of the proposed architecture and the flexibility of its programming model. Based on these preliminary results, MORA appears to be extremely feasible as a reconfigurable multi-core soft processing architecture which allows high level programming of the entire applications on FPGA. Further research continues to build on the encouraging primary investigation and will involve further cost and performance optimization of MORA to make it a strong contender in the soft processing category.

6. REFERENCES

[1] E. Mirsky and A. DeHon, "MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources," in Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines, 1996., pp. 157–166, 1996.

[2] Singh H, Ming-Hau Lee, Guangming Lu, Kurdahi F.J, Bagherzadeh N, Chaves Filho E.M, "MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications,"IEEE Transactions on Computers, vol.49, no.5, pp.465-481, May 2000.

[3] Z. Yu, M. Meeuwsen, R. Apperson, O. Sattari, M. Lai, J. Webb, E. Work, D. Truong, T. Mohsenin, B. Baas, "AsAP: An Asynchronous Array of Simple Processors,"IEEE Journal of Solid-State Circuits (JSSC), vol. 43, no. 3, pp. 695-705, March 2008.

[4] Xilinx Processor Reference Guides www.xilinx.com

[5] M. Lanuzza, S. Perri, P. Corsonello, M. Margala, "A New Reconfigurable Coarse-Grain Architecture for Multimedia Applications", in Proceedings of Second NASA/ESA Conference on Adaptive Hardware and Systems (AHS), pp. 119-126, 2007

[6] M. Lanuzza, S. Perri, P. Corsonello," MORA- A New Coarse Grain Reconfigurable Array for High Throughput Multimedia Processing", Proceedings of International Symposium on Systems, Architecture, Modeling and Simulation,(SAMOS), pp-159-168, 2007.

[7] S. Purohit, S. Chalamalasetti, M. Margala, P. Corsonello, "Power-Efficient High Throughput Reconfigurable Datapath Design for Portable Multimedia Devices," in Proceedings of International Conference on Reconfigurable Computing and FPGAs, pp. 217-222, December 2008.

[8] Brandon J. Jasionowski, Michelle K. Lay, Martin Margala, "A Processor-In-Memory Architecture for Multimedia Compression," in Transactions on Very Large Scale Integrated(VLSI) Systems, pp. 478-483,2007.

[9] D. Flanagan, Y.Matsumoto, "The Ruby Programming Language",O'Reilly,2008.