2009 NASA/ESA Conference on Adaptive Hardware and Systems

MORA – An Architecture and Programming Model for a Resource Efficient Coarse Grained Reconfigurable Processor

Sai Rahul Chalamalasetti, Sohan Purohit, Martin Margala
Dept. of Electrical and Computer Engineering
University of Massachusetts Lowell, MA 01854
martin_margala@uml.edu

Wim Vanderbauwhede
Dept. of Computing Science
University of Glasgow, UK
wim@dcs.gla.ac.uk

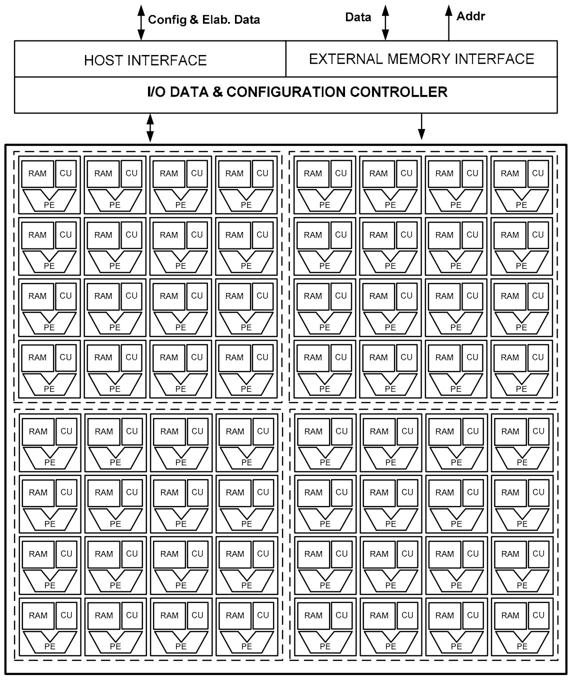
Abstract

This paper presents an architecture and implementation details for MORA, a novel coarse grained reconfigurable processor for accelerating media processing applications. The MORA architecture involves a 2-D array of several such processors, to deliver low cost, high throughput performance in media processing applications. A distinguishing feature of the MORA architecture is the co-design of hardware architecture and low-level programming language throughout the design cycle. The implementation details for the single MORA processor, and benchmark evaluation using a cycle accurate simulator are presented.

1. Introduction

Recent advances in reconfigurable computing have led to the development of increasingly sophisticated reconfigurable platforms for media processing and other applications. Due to the rapid advances in algorithms and applications in DSP, there has been an increasing demand for the computing platforms to be easily adaptable, low cost while consistently delivering maximum performance at all times. Coarse grained reconfigurable (CGRA) solutions cater to this demand to some extent, with their high level of flexibility and more efficient routing structures.

Several coarse grained solutions have been previously proposed. Based on their architectures CGRAs can be classified as either linear systems or mesh based systems. MORA[12] is a 2-D mesh-based CGRA system, consisting of an array of reconfigurable cells arranged in four 4×4 quadrants, shown in Fig.1. Within its category, MORA bears resemblance to MATRIX[1] and Silicon Hive[2] which both have in-memory processing. However MORA's processing element is much more powerful than MATRIX and less complex than Silicon Hive. In addition to this, a distinguishing feature of MORA is the massive reduction in the number of actual transistors per core. In fact with each MORA core requiring just over 60,000 transistors, the entire array of 64 cells requires only a fraction of the resources of architectures like Ambric[3] and DAPDNA[5]. MORA provides for most of the basic arithmetic and logic functions required by the target domain of media processing, while providing reasonably high throughput with minimum resource utilization. Another feature that sets MORA apart from existing CGRAs as well as FPGAs is the simple application mapping. Unlike most of the other architectures, MORA acts more as a general platform and does not optimize for a particular algorithm for maximum throughput implementation of an application. This ability allows MORA to remain a low-cost generalized platform for the media processing (and other application) domain.


Config & Elab. Data
Data
Addr
HOST INTERFACE
EXTERNAL MEMORY INTERFACE
I/O DATA & CONFIGURATION CONTROLLER
RAM
CU
PE
Figure 1: 2-D array of MORA processors

This paper presents the custom design and implementation of the MORA reconfigurable cell. Each RC consists of an 8-bit Processing Element, 256×8 dual port data memory and a central controller. An important feature of the work is that hardware and software were co-designed throughout the design cycle. The paper also provides an overview of the development of the MORA assembly language and evaluation for DCT, DWT and H.264 benchmarks, using a cycle accurate simulator for MORA.

The rest of the paper is organized as follows. Section 2 describes the implementation of the Processing Element. The Control unit and the Memory organization are detailed in Sections 3 and 4 respectively. Section 5 discusses in brief the development of the MORA assembly language. Performance results and comparisons with competing architectures are presented in Section 6.

2. Processing Element

The Processing Element (PE) is the main computational unit of the RC. Its structure is inherently based on the data path unit proposed in [6]. A significant improvement is the ability to implement a full range of signed and unsigned integer arithmetic, as well as provisions for logical, shifting and comparison operations.

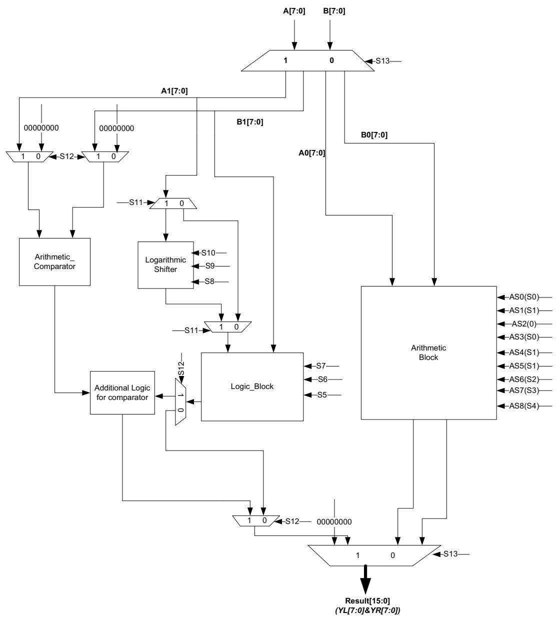


Figure 2: Top level organization of Processing Element

Fig. 2 shows the top level organization of the Processing Element. The PE is organized into a logic section and an arithmetic section. A combination of multiplexers and demultiplexers controlled by vectors *S[13:0]* efficiently route the operands through the two sections. The following sub-sections detail the organization and implementation of the logic and arithmetic sections of the PE.

2.1. Logic Unit: Implementation Strategies

The logic unit provides for bitwise AND/NAND, OR/NOR, XOR/XNOR, shifting and comparison operations on 8-bit operands A and B. The logical unit is comprised of a comparator, logarithmic shifter and an array of AND, OR, XOR gates. Multiplexers at each stage route the operands to perform shifting, comparison or simple bit-wise logic operations. At each stage, multiplexers also serve an additional purpose of reducing the total loading on the operands by reducing the total number of gates to be driven by each stage. Although this scheme trades off slightly the performance of the logic path, the delay is still within the total worst case delay of the data path.

Shifting uses an 8-bit logarithmic shifter built with 2:1 multiplexers. This implementation of the logarithmic shifter can only support one type of shifting operation at a time, i.e. Round Shift or Shift Out. Supporting both types of shifting therefore means including additional logic to convert from round shifting to shift out and vice versa. After considering the trade-offs in each approach, we implemented the round shifting topology, and use the logic block to additionally support shift-out operations. A sample shifting operation is demonstrated below.

Round Shift:
SHIFT 4 A (FFh) → Round Right shift Operation
00001011 → operand A(value to be shifted)
4 → number of bits to be shifted
10110000 → Intermediate result of shifter ...(1)
11111111 → operand B (Logical AND with (1))
10110000 → final result
Shift-Out:
SHIFT 4 A (0Fh) → Right shift Operation
10111111 → operand A(value to be shifted)
4 → number of bits to be shifted
11111011 → Intermediate result of shifter. ...(2)
00001111 → operand B (Logical AND with (2))
00001011 → final result.

Another useful operation supported by the MORA RC is comparison. This ability also extends the applications of MORA beyond media processing, to other domains like encryption, pattern matching, network intrusion detection, etc which involve heavy, dedicated comparison operations on incoming data sets. For this purpose the MORA PE uses a specially designed comparison block. Different structures of comparators were studied for area and power consumption. A comparison operation can essentially be considered as an extension of simple 2's complement subtraction. The subtraction operation A-B will result in a borrow Bout of 0 or 1 depending on whether A is less than equal to or greater than B. The result of the subtraction, if 0 implies the two numbers are equal. If not zero, the state of the Bout decides A<B or A>B. Using this approach allows us to use the results of the subtractor along with the gates from the logic block to clearly define three outputs of the comparator, i.e. A=B, A<B and A>B. Since the most important bit is the Bout, we eliminate the excess circuitry from the subtractor and build only the carry path instead. This approach allows significant savings in delay, power and area. A potential error in comparison for negative numbers is eliminated by using the sign bit of the operands, and inverting the decision logic for negative numbers. This implementation of comparison is chosen to eliminate the need for additional area for a dedicated comparator, and also encourage reuse and maximum utilization of the available hardware resources.

2.2. Arithmetic Unit

A bulk of operations that need to be supported in the domain of media processing involves repetitive arithmetic. The design of arithmetic data paths is therefore critical to the performance of the entire system. We suggest 8 bits bit as an ideal operand size for our category of reconfigurable structures. A detailed analysis and design of 8-bit data path has been presented in [6]. The data path used in the current architecture builds on the data path presented in [6]. The signed arithmetic data path in [6] is limited to being a 7 bit data path due to the loss of one bit position as a sign bit. As a result the range of numbers to be represented by this data path drops down from 0-255 to 0-127. In order to overcome this shortcoming, we propose a new adaptable data path structure to support both 8-bit signed and unsigned arithmetic operations. The data path is shown in Fig. 3. It consists of two 8×4 hybrid multipliers, a compressor stage and 16- bit carry linked adders. The hybrid multiplier structure based on [7] uses control signal *S8* and switches between signed and unsigned multiplication modes. During multiplication, the two multipliers provide the intermediate products *B[7:0]* × *A[7:4]* and *B[7:0]* × *A[3:0]*. The intermediate products are then compressed using the 3:2 compressor stage. The final 16- bit carry linked adders compute the result of the multiplication. The data path uses a control signal specified by the control unit and instruction word, to indicate signed or unsigned operations. This saves a bit position from being lost to the sign bit, limits the operand size to 8 bits, and allows flexible switching between signed and unsigned operations.

For addition and subtraction, the multipliers provide *A[7:4]* × *00000001* and *B[7:0]* × *0001*. The adders combine these partial products and produce the final result of the addition or subtraction operation. For subtraction, the operand A is negated and input carry Cin is set to 1, therefore performing 2's complement subtraction.

The results of the PE are available at the output of the registers. These can then be sent to memory of the same or different RC, as specified by the instruction. Media processing often requires that these arithmetic and logic operations be carried out in a repetitive manner. For this purpose the registers at the input and output of the RC are synchronized with each other, so as to allow accumulation operations. The registers also allow data from output register of one RC to be routed through to the input register of another RC directly, thus bypassing the memory. These operations are managed centrally by the control unit which synchronizes both the internal and external handshake mechanisms within the architecture, thereby making each RC behave as a small independent DSP style processor. The following section describes the design and implementation of the control unit.

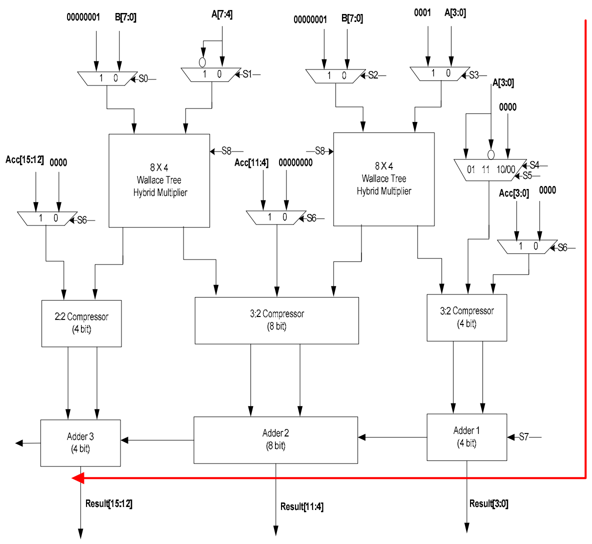


Figure 3: Design of Arithmetic Unit

3. Control Unit

The Control unit shown in Fig, 4 is the main decision making block of the reconfigurable cell, and ensures complete synchronization within each component of the RC. It consists of a small refreshable Instruction Memory, a finite state Instruction Machine, Instruction Decoder, and Address generator.

The instruction memory is a 16 word SRAM with each instruction word being 92 bits wide. The instruction word consists of fields for instruction code, base addresses of the operands A and B, base address for storing the output, address descriptors for traversing the data memory, and other fields to describe the number of times an operation is to be performed. The RC supports a total of 28 instructions, including all arithmetic, logic and memory instructions. Table 1 shows the list of supported instructions and a brief description of the action performed. Each arithmetic operation has an unsigned

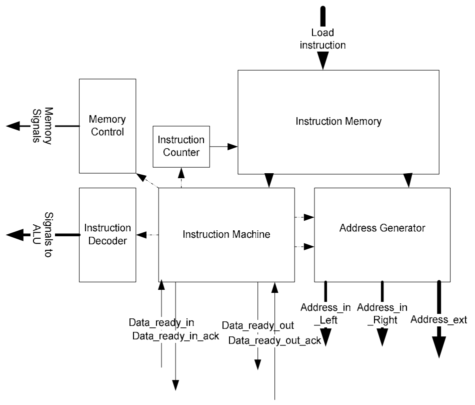


Figure 4: Control Unit Design

Table 1: Description of supported instructions

Instruction	**Description**	**Instruction**	**Description**
ADD,SUB,MUL	*Add, subtract, multiply*	**MOVE_B**	*Move (Bank to Bank)*
(ADD,SUB,MUL)ACC	*Add, sub, mult with Accumulate*	**MOVE_O**	*Move (RAM to output)*
NOT, AND, OR,NAND, NOR,XNOR	*Combinatorial operations*	**PASS_TROUGH**	*Simple Data pass*
CMP	*Compare*	**JUMP**	*Jump to particular Inst*
SHIFT	*Shift*	**JUMPIF**	*Cond. Jump to part. Inst.*

counterpart (suffix _U).

The arithmetic and logic instruction set is supplemented with the instructions to perform jumps (JMP) and conditional branches (JMPIF), as well as moving data within internal memory. These features of the instruction set, together, with the ability to perform multiple executions of a single instruction, makes the RC extremely versatile and adaptable and greatly simplifies the programming model. The flexibility comes at the cost of a longer instruction word, and a slightly larger control and synchronization circuit, but considering the benefits obtained, this is a small price to pay.

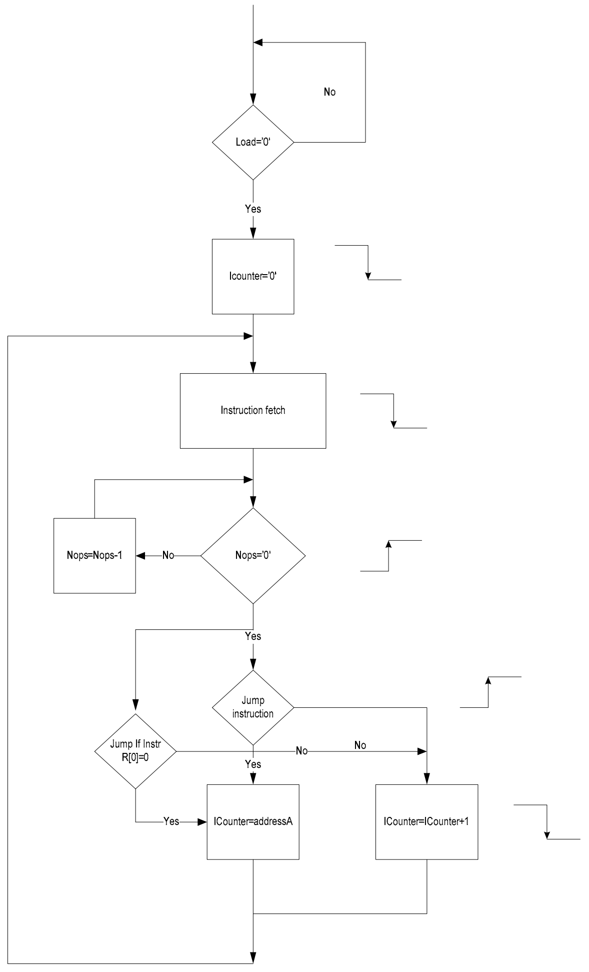


Figure 5: Instruction Machine sequential operation.

The Instruction machine is a small state machine that controls the instruction fetch operations, synchronizes the other integral components of control unit and control the external handshake signals to communicate with other RC's. The most important function of this unit is to coordinate access to the instruction memory and control the instruction queue. It also handles the external communication to other RCs. The basic operation of the Instruction memory is illustrated as a flow chart in Fig. 5. Depending upon the information in the instruction word, the Memory Communication block controls the flow of signals to and from the data memory for read, write and move operations. The opcode is extracted from the Instruction word, and converted into a set of 14 meaningful control vectors *S[13:0]* by the Instruction Decoder.

A special feature of the control unit is the address generator. Fig. 6 shows the block diagram of the address generator. The address generator accepts the base address or initial memory address to fetch operands. Depending upon the number of times the same operation is to be performed, and using the address descriptors *Step, Skip* and *Subset* it calculates the address of the memory location from where the next data sets are to be fetched, or next outputs are to be stored. The control unit utilizes two copies of the address generator, one each for the operands A and B and a third copy for the output C.

The RC performs *Read* and *Write* operations in opposite phases of the clock. As a result the output address generator and the operand address generators are also made to work out of phase with each other. The address descriptors *Step, Skip, Subset* allow the control unit to traverse the entire memory space, in either direction. This proves particularly important and useful, considering the large amount of matrix based operations involved in the target application domain.

Besides the internal synchronization signals, the control unit also handles asynchronous handshake signals to communicate with adjacent cells. These signals are used by the RCs to indicate to communicate to the neighboring RC about availability of data either in its data

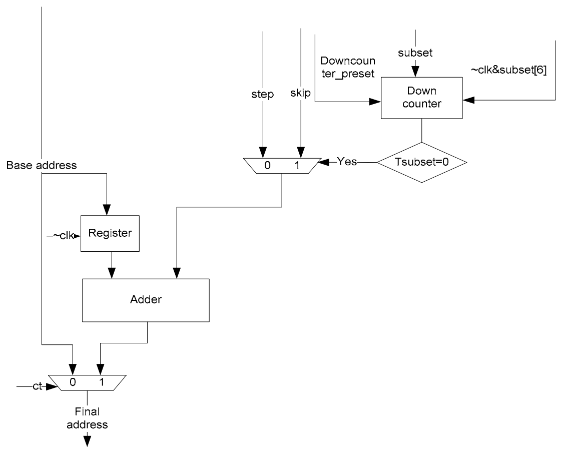


Figure 6: Address Generator.

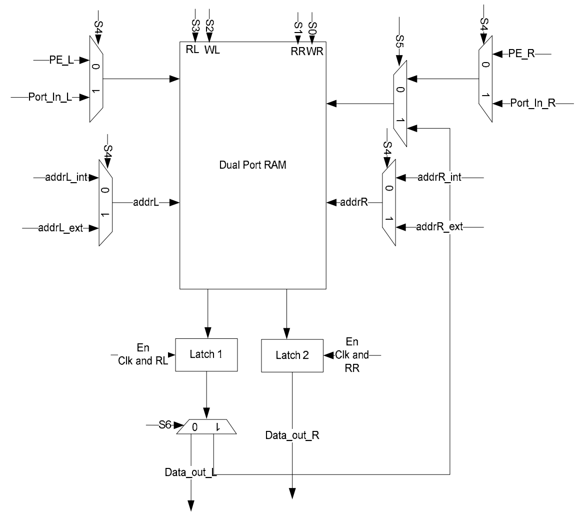


Figure 7: Organization of RC Memory

memory or the output registers. The neighboring RC then sends an acknowledge signal, once the data transfer is completed. These signals are asynchronous and allow each RC to continue with its own independent execution cycle. It can be seen from the above discussion, that the control unit provides a large amount of intelligence and flexibility to each RC, transforming it into a tiny DSP style processor that can work independently during a program execution. Another important feature of the RC that allows it to be so flexible is efficient memory organization and is presented in the subsequent section.

4. Memory Organization

A striking difference between MORA and several competing architectures in the same category is that MORA does not use a centralized RAM system. Instead each RC is provided with a 256×8 bits data memory bank. This allows each RC to work as a tiny Processor In Memory (PIM[11) [8], i.e. operations are be performed close to memory. The approach allows each RC to work independently of the others, and eliminates the possibility of contention for memory resources among 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. Considering that the RCs during program execution may require data from each other, a dual port memory topology is preferred, to allow easier memory access to the RCs. Fig. 7 shows the organization of data memory in each RC. The data memory is made up of 256×8 dual port SRAM array. Each port has individual address decoders and read, write control signals. Multiplexers controlled by signal S4 control data flow into the memory. Depending on the instruction word, these multiplexers write the result of the current PE operation or data from external RC into the specified memory address, through the specified port. Similarly the output multiplexer controlled through S6 controls the flow of data out of the data memory by directing it either into the PE of the same RC or into the PE or data memory of a neighboring RC. Since media processing often requires operations based on matrices and vectors, it is necessary for the RC to be able to move data within its data memory. This is implemented using an additional multiplexer controlled through S6, which allows the RC to read data from a memory location through the left port, and transfer it to another memory location, internally through the right. The read and write operations are performed during opposite phases of the clock signal. This allows the RC to perform the MOVE operation in a single clock cycle. The alternately phased read and write allows the RC to perform the read-execute-write sequence in a single clock cycle. As a result, each of the 28 instructions mentioned earlier take exactly one clock cycle to complete.

Bit-line loading is a common problem with building large memory arrays. Addressing this problem, requires a careful design of a strong driving circuit and stable sense amplifier. Several solutions have been proposed to counter this problem at the organizational level. [9]. Our approach was to divide the 256 locations into four banks of 64 words each. The two MSB bits, *A7* and *A6* of the address *A[7:0]* are used to select the appropriate bank. This bank select signal is further AND gated with the *Write_Enable* and *Data* to control the writing of data into the memory. The AND gates are sized to be strong, along-with an appropriately sized inverter at the output, to be robust enough to drive the bit-lines. This allows us to provide an efficient driving scheme for the memory, with inherent control signaling. It should be noted that to the outside circuitry, i.e. Processing Element and Control Unit, as well as to the programmer, it appears as a single array of 256 memory locations. This eliminates the need of any special precautions or programming styles to

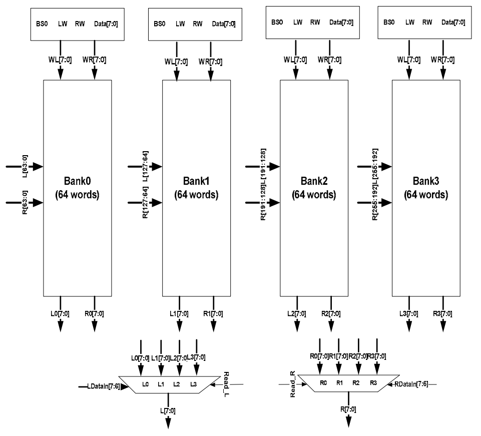


Figure 8: Internal structure of RC RAM banks

handle the memory banking. Fig. 8 shows the internal arrangement of the 4 RAM banks. The individual memory cells use 8T dual port SRAM topology. This topology is preferred primarily due to a perfect combination of size, speed, simplicity and robustness. The memory array provides a read access time of 0.8ns with an average power dissipation of 1.56 mW at 166 MHz.

5. Programming Model

5.1 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.

5.2 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.

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
```

The arguments represent the number of operations, the virtual destination register, the output address and the operand addresses. 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.

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^{τ})

Q: *N/2×N/2* matrix (QT: transposed matrix Q^{τ})

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.

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.

5.3 Generation Language

This component of the language is in itself an imperative mini-language with a simple and clean syntax inspired mainly by Ruby[1]. 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.

Example: 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.

[1] www.ruby-lang.org

5.4 Coordination Language

MORA's coordination language is a compositional, hierarchical netlist-based language inspired by hardware design languages such as Verilog and VHDL. 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 2x2 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 non-RTL Verilog. 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)
```

6. Performance Results

A full custom design of the entire RC was carried out in IBM's 0.13µm CMOS process. For ease of simulation and verification, a test instruction sequence was executed using an external ROM as the instruction memory. Each individual block was evaluated for speed and power performance at 166 MHz. The results are presented in Table 2. To emphasize the resource efficiency of the proposed cell, we have included the transistor count of each block.

For estimating the performance of the MORA architecture, a cycle accurate simulator was developed. Using the MORA assembly language, several benchmarks were coded into the compiler/simulator. Table 3 presents a performance analysis of MORA when performing popular benchmark applications.

Table 2: Block wise power and transistor count

Component	Transistors	Power (mW)
PE	11,662	0.425
Control Unit	24,140	1.185
DPRAM	27,010	1.558

Table 3: Performance analysis of MORA processor core for benchmark applications @ 166 MHz

Benchmark	**Total Delay (ns)**	**Total Latency (ns)**	**# Blocks processed in parallel**	**Throughput (MOPS)**	**# RCs utilized**
8×8 2-D DCT,min. delay	432	216	1	4.63	56
8×8 2-D DCT max. throughput	1536	1536	8	5.21	64
4×4 H.264 2-D IT min. delay	108	54	1	18.52	56
4×4 H.264 2-D IT max. throughput	192	192	8	41.67	64
32×32 DWT LeGall (5,3)	14,592	13,056	16	1.23	64

The MORA architecture is evaluated for 8×8 2-D DCT, 4×4 H.264 2-D integer transform and 32×32 8-point LeGall (5,3) DWT applications. The DCT and H.264 transforms are implemented as simple 8-bit integer matrix multiplications, 2-D DCT: $C.X.C^T$ and H.264: $(H.X.H^T)\otimes E$. The matrix multiplication is split into a number of parallel operations as explained in the previous section. Note that the direct product does not require extra time on MORA as it is integrated with the operations for combining the partial results. The table shows two different optimizations for DCT and H.264, minimal delay and maximal throughput. The first implementation (minimum delay) uses as many RCs as possible to process a single image block; the second implementation (maximum throughput) uses as few RCs as possible per image block in order to maximize the number of image blocks processed in parallel, leading to a higher throughput (more than double for H.264).

The 8-point wavelet transform is implemented using a pipeline of 4 RCs, each RC computes following equations using 6 single-cycle operations:

$y_i = x_i - (x_{i-1}+x_{i+1})/2$

$y_{i-1} = x_{i-1} + (y_i+y_{i-2})/4$

To maximize throughput, 16 blocks are processed in parallel.

The analysis presents the performance of the MORA processor in terms of total computational time and overall system throughput. Another important point to be noted is that all the algorithms are implemented for maximum resource utilization. The benchmarks are programmed to minimize the number of cores idle during execution time, thereby guaranteeing a high utilization factor.

7. Conclusion

This work presented the organization and VLSI implementation of the MORA processor core. The circuit was evaluated for speed and power performance; the architecture was evaluated for popular benchmark applications implemented in the MORA assembly language using a cycle accurate simulator. The proposed architecture builds on its predecessor proposed in [10] and extends the functionality to further suit the demands of media processing applications and algorithms. The architecture is extremely light in terms of actual silicon resources, while still maintaining high throughput efficiency.

Future work in hardware as well as software development aims to build on these encouraging preliminary results, to provide a scalable, low-cost, highly programmable reconfigurable platform for media processing. We also aim to explore the possible applications of MORA beyond the media processing domain.

8. References

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

[2] M. Cocco, J. Dielissen, M. Heijligers, A. Hekstra,J. Huisken, S. Hive, and N. Eindhoven, "A scalable architecture for LDPC decoding," in Proceedings of Design, Automation and Test in Europe Conference and Exhibition, vol.3,pp- 88-93, 2004.

[3] Mike Butts, "Synchronization through Communication in a Massively Parallel Processor Array," *IEEE Micro*, vol. 27, no. 5, pp. 32-40, September/October, 2007.

[4] 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, 2008.

[5] DAPDNA-2 Dynamically Reconfigurable Processor product brochure, IPFlex Inc., 13th March 2007. http://www.ipflex.com/en/E1-products/dd2Arch.html.

[6] 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, 2008.

[7] S. Krithivasan, M.J Schulte, "Multiplier architectures for media processing," Record of the 37th Asilomar Conference on Signals, Systems and Computers, pp. 2193-2197, 2003.

[8] M. Lanuzza, S. Perri, P. Corsonello, M. Margala, "A New Reconfigurable Coarse-Grain Architecture for Multimedia Applications," 2nd NASA/ESA Conference on Adaptive Hardware and Systems, pp-119-126, 2007.

[9] A. Karandikar, K.K Parhi, "Low power SRAM design using hierarchical divided bit-line approach ," Proceedings of International Conference on Computer Design, pp. 82-88, 1998.

[10] 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.