Instruction Set Design for Coarse-Grained Reconfigurable Architectures

Maria Dalila Vieira[1] Lucas Bragança[2] Michael Canesche[2] Ricardo S. Ferreira [2] José Augusto M. Nacif[1]
[1]Science and Technology Institute, Universidade Federal de Viçosa, Florestal, Minas Gerais, Brasil
[2]Informatics Departament, Universidade Federal de Viçosa, Viçosa, Minas Gerais, Brasil
{maria.d.vieira, lucas.braganca, michael.canesche, ricardo, jnacif}@ufv.br

***Abstract*—In the last ten years, the demand for performance improvements in computing systems has not fulfilled by CPU enhancements. A solution widely applied in different computing is the use of hardware accelerators. In the industrial scenario, accelerators such as Graphics Processing Unit (GPU) are more popular because they offer a well-defined and established programming interface. Besides having competitive performance, especially in terms of energy efficiency, Field Programmable Gate Array (FPGA) based accelerators are still challenging to develop. Thus, it is necessary to look for ways to make the FPGA more accessible and hiding hardware implementation details from the programmer. This work proposes a low-level language to describe configurations of a Coarse-Grained Reconfigurable Architecture, which operates as a virtual layer on a FPGA. This virtual layer is intended to hide hardware details and reduce reconfiguration time. Our results present a concise set of instructions and a competitive performance. We obtain more than 2.5 GOPS for benchmarks with a dozen operations.**

***Index Terms*—ISA, CGRA, FPGA, reconfigurable architectures, dataflow computing**

I. Introduction

Recent improvements in CPU performance do not supply the increasing demand of computing systems, thus hardware accelerators have become a widely adopted solution. In this direction, a large part of high-performance systems is usually composed of heterogeneous platforms connected to a general-purpose processor, such as a CPU, and a hardware accelerator [6]. Currently, the most common accelerators are Graphics Processing Units (GPUs), Field Programmable Gate Array (FPGAs), and Application Specific Integrated Circuits (ASICs). Each day, GPUs are becoming more accessible, because they are programmed by specific languages: OpenCL or C for Compute Unified Device Architecture (CUDA). Whereas FPGAs are still programmed in hardware description languages, such as Verilog or VHDL. Incidentally, even when programming with OpenCL, FPGAs still require knowledge of hardware details so that acceleration can be efficient [6].

One reason to prefer FPGAs over GPUs is the difference in energy cost, FPGAs are clearly less expensive in terms of energy costs [6]. The abstraction of hardware details that are not fundamental to general-propose acceleration is a well-known strategy to facilitate programming. Creating a Coarse-Grained Reconfigurable Architecture (CGRA) as a virtual layer over the FPGA is a method to dispense the gate-level reconfigurability and gain a lower reconfiguration time [1]. Several papers in literature propose the implementation of CGRAs with dataflow entries expressed in operations graphs.

In this paper, we propose a low-level language to describe the CGRA input. The objective of this language is standardizing the description of the dataflow graph. For the matter of simplicity, we choose a RISC-based ISA model, with fixed formats. One of the main advantages of our work is the concept of sources and destinations. In our architecture, they can represent more than internal registers.

This article presents an Instruction Set Architecture (ISA) to configure the CGRA specified in [1]. The remainder of this paper shows more details as follows: in Section II we show similarities of this work with other papers. Section III describes CGRA characteristics that were fundamental in ISA construction. Thereon, Section IV has ISA instructions description. Finally, in Section V, we show our results and in Section VI, we discuss the results and propose future work.

II. Related Work

This section presents sets of instructions that describe operations performed in dataflow processors by other works. Each of the following paragraphs describes one of these works.

WaveScalar ISA [2] is a set of instructions that describe operations performed over a dataflow architecture. An important difference between previous dataflow architectures and WaveSacalar is the traditional Von Newmann-style memory semantics support. The instructions and values that are operated are stored in a cache. The motivation for using this cache is the optimization of the physical placement of instructions which depend on each other. Through a good physical positioning of instructions, it is possible to explore the dataflow locality. In the instruction set, besides conventional operations, there are operations to manage the control flow. Each WaveScalar instruction contains an operation code and a list of targets for each of the outputs of the instruction.

TRIP [3], [5] owns an instruction set architecture with operand communication for data processing. The work goal is to achieve more performance and energy efficiency. Performance improvements are directed to the choice of a communication mechanism for each instruction during compilation. This choice is made through a heuristic that defines the communication mechanism. To encode this compiler decision the authors use the instruction set definitions. Also, there is micro-architectural support to treat hybrid communication mechanism.

MORA [4] is a Coarse-Grained Reconfigurable Processor, which is implemented in hardware. Similar to the works above mentioned, it has a low-level programming language to describe the performed operations. The objective of this work is the accelerating media process applications through a computing platform, which is easily adaptable, has a low cost and great performance.

III. Coarse-Grained Reconfigurable Architecture

Bragança *et al.* [1] define an architecture that has a processing elements array, each one with 2 inputs and 2 outputs. Its architecture particularity is the interconnection between inputs and outputs, which is given by a blocking interconnection network, more specifically an omega network.

The architecture is configured through high-level dataflow graphs. In these graphs, each vertex represents an operation, and the edges represent the data path. And each operation must be mapped to a processing element. Once we did the mapping, we must establish the routing in order for the input data from the algorithm flow through the processing elements. This happens so that every input can be connected to another element output. Routing is only possible if we can connect every input to an output.

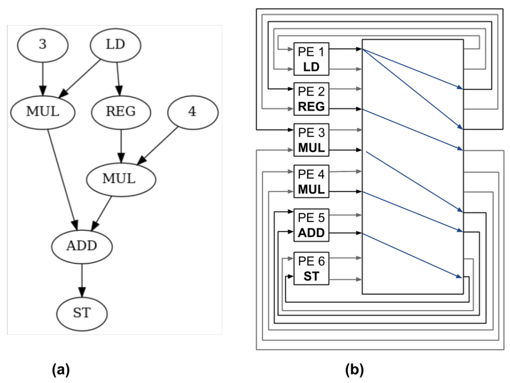
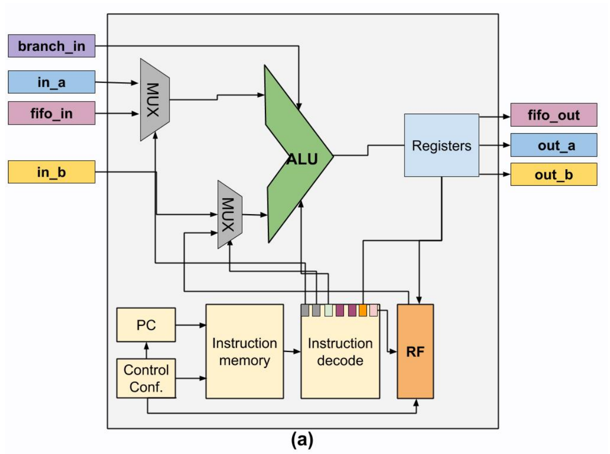


Fig. 1. Abstract example of mapping and routing using Omega Network.

In addition to the input data of the algorithm performed in the CGRA, we also need to send the configuration expressed in bits to the FPGA. A part of these bits is used to represent the configuration of each processing element, and other bits determine the mapping and routing of the network. Figure 1 shows a mapping and routing example. In Figure 1 (a) we show an operation graph, whose vertices are mapped for processing elements in Figure 1 (b). The routing interconnects the elements according to relations established by the graph edges, as Figure 1 (b) denotes.

Figure 2 (a) is a simplified diagram of processing element architecture. This diagram has only parts of the architecture that have a direct relationship with ISA. The processing elements implementation is not exactly as defined in Figure 2 (a).



(a)

OpCode	Control	Register Read	Register Write
4 bits	4 bits	4 bits	4 bits

(b)

Fig. 2. Processing Element Architecture.

The CGRA implements three types of processing elements, which belong to one of the following sets:

- Elements that can receive data from the input queue;
- Elements that can write data in the input queue;
- Elements that do not read or write in a FIFO, only perform operations;

Figure 2 (a) shows the Arithmetic Logic Unit with 2 inputs connected to multiplexers, each one with 2 input options. The input A of the unit can receive data from input FIFO (fifo_in) or standard input (in_a). And input B receives data from standard input or a register of RegisterFile. The processing elements outputs can be standard outputs A or B, or output FIFO. This input and output definition is the basis to instruction format, which is : *Operation dst, A, B*

Wherein dst represents the processing element output, and the inputs are defined by A and B. The instruction set architecture, which is explained in the next Section, turns each instruction into a set of bits. Thus, the bits are classified according to Table 2 (b). Opcode defines the operation identifier, Control is given according to the dst, A and B types, Register Read specifies a 4-bit number to identify a register to be read and Register Write indicates a register to write.

Table 2 (b) defines the number of bits required for each instruction field. As shown in this table, each bit set has 16 bits, and each field has 4 bits. Thus, the ISA represents the processing elements with a limitation of 16 registers, because they need to be represented with only 4 bits.

IV. Instruction Set Architecture

The Instruction Set Architecture (ISA) receives an instruction set and returns the configuration bits of the network. Therefore, the set of instructions is equivalent to the format of the operations established in the dataflow graph. And the format of the instructions follows the known RISC format, with exceptions regarding the specifics of processing elements.

All ISA instructions have a destination and one or two sources, as specified in Table I. Besides the format and fields, that Table also shows which operation is represented by each instruction. Table I defines the instructions fields according to the performed operation. As the table shows, there are three possible field types: destination (dst) and sources (A or B). The definition of these fields, as mentioned at the end of section III, is bound to the processing elements architecture.

TABLE I
INSTRUCTION SET ARCHITECTURE.

OpCode	Instruction	Assembly	Description
0	passa	dst, A	dst = A
1	passb	dst, B	dst = B
2	min	dst, A, B	dst = min(A, B)
3	max	dst, A, B	dst = max(A, B)
0	beq	dst*, A, B	dst = equal(A, B)
1	bne	dst*, A, B	dst = not(A, B)
2	slt	dst*, A, B	dst = A < B
3	sgt	dst*, A, B	dst = A > B
4	add	dst, A, B	dst = A + B
5	sub	dst, A, B	dst = A - B
6	mult	dst, A, B	dst = A * B
7	xor	dst, A, B	dst = xor(A, B)
8	and	dst, A, B	dst = and(A, B)
9	or	dst, A, B	dst = or(A, B)
10	not	dst, src	dst = not(src)
11	shl	dst, A, B	dst = A << B
12	shr	dst, A, B	dst = A >> B
13	merge	dst, A, B, PE	dst = PE ? A : B
14	abs	dst, src	dst = abs(A, B)

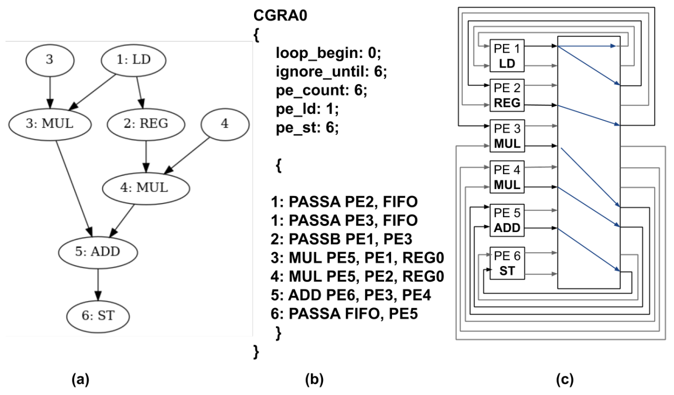
*dst indicates branch output.

In Table I, the instructions passa and passb copy an input value to an output. The following instructions, min and max, compare two inputs and return the bigger or the smaller, respectively. And beq, bne, slt, and sgt, return a bit through the branch output. It indicates if the result of the operation is true. Instructions with OpCodes 4-6 are arithmetic operations. And OpCodes 7-12 indicates logic operations, being shl the Shift Logical Left and shr the Shift Logical Right. The merge is a multiplexer, it receives two values and last a control. And finally, the operation absolute (abs) converts the input to the respective unsigned value.

Instruction set fields are defined so that dst defines the output of the processing element. That output can be directed to the input of another element, or an output FIFO, or can even be written in an element register. The first input is represented by A, which indicates the possibility of input via FIFO or output from another processing element. Due that a processing element cannot be simultaneously capable of reading and writing in the FIFO. The use of FIFO in A depends on whether this feature has not been employed in dst. And finally, the B indicates entrance of data from the output other processing elements or the RegisterFile of the element.

As stated in the previous section, the instruction set aims to describe an operation graph, wherein both represent dataflow operations. In Figure 3, we show an example of conversion from a graph to instructions. Furthermore, it shows the use of the instruction set and how it represents the operations graph.

In the example 3 (b), before each instruction statement, there is an integer. That value indicates the processing element that executes the instruction. The use of a graph of operations such as Figure 3(a) facilitates the correct formulation of instructions. In these graphs, the LD label indicates input FIFO, and the ST label denotes output FIFO. The other vertices represent the operations labeled on them. The edges that arrive at the corresponding operation vertices denote instructions source. Arrows that come out of these corresponding vertices indicate the instructions destinations.



```
CGRA0
{
  loop_begin: 0;
  ignore_until: 6;
  pe_count: 6;
  pe_ld: 1;
  pe_st: 6;

  {

  1: PASSA PE2, FIFO
  1: PASSA PE3, FIFO
  2: PASSB PE1, PE3
  3: MUL PE5, PE1, REG0
  4: MUL PE5, PE2, REG0
  5: ADD PE6, PE3, PE4
  6: PASSA FIFO, PE5
  }
}
```

Fig. 3. Operations graph and our ISA instructions.

In Figure 3(b), the instructions are grouped into a CGRA due to ISA can perform several CGRAs at once. However, we must establish some important CGRA configurations before. Those configuration values, respectively are:

- loop begin: it is the height of the input graph to which the loop should return;
- ignore until: represents an initial time for FIFO writings;
- pe count: represents the total of processing elements;
- pe ld: it is a list of processing elements with FIFO input;
- pe st: it is a list of processing elements with FIFO output;

For the FIFO reading to occur, the processing element responsible for it must belong to the list (pe ld). The same applies to the FIFO writing and the list (pe st). Moreover, a processing element cannot have an index greater than the value defined in the term (pe count).

V. RESULTS

We developed an Instruction Set Architecture to facilitate the description of programs executed in CGRA. For this reason, the instructions are simple, standardized and clearly describe the processing element's operation and the input data flow. We present the results analysis in two parts, the first compares the number of instructions that describes a code, and the second compares it in terms of efficiency.

A. Code analysis

This Section compares the x86-64 instruction set and our instruction set. In Figure 4, we show an algorithm implemented in C (a), and in x86-64 assembly (b). We choose compare our instruction set with the x86-64 assembly because we aim to compare the i5 with our architecture. And we choose implement this code because of simplicity and the employment of several resources of our architecture. That is because we decide to use a single example in the whole article. A good part of the i5 instructions is spent with the loop and indexing the vector. Precisely because our ISA describes dataflow operations, loops do not represent an increase in the number of instructions. Furthermore, for the C code, the compiler needs to do loop unrolling or pipeline software to achieve performance, which it is not necessary for our architecture. And the dataflow replaces the vectors. Thus, we can describe the same algorithms with fewer instructions. Figure 3 (b) shows our instruction set which represents the operations graph mapped in Figure 1. That instructions are equivalent to the x86-64 instructions, shown in Figure 4 (b). That is, the operations of the graph in 3 (b) correspond to the algorithm in 4 (a).

```c
#include <stdio.h>

int main(){
   int i, in[1000], out[1000];

   for(i=0; i<1000-1; i++){
      out[i] = 4*in[i] + 3+in[i+1];
   }

   return 0;
}
```

(a)

```
main:
push rbp
mov rbp, rsp
sub rsp, 7896
mov DWORD PTR [rbp-4], 0

.L3:
cmp DWORD PTR [rbp-4], 998
jg .L2
mov eax, DWORD PTR [rbp-4]
cdqe
mov eax, DWORD PTR [rbp-4016+rax*4]
sal eax, 2
lea edx, [rax+3]
mov eax, DWORD PTR [rbp-4]
add eax, 1
cdqe
mov eax, DWORD PTR
[rbp-4016+rax*4]
add edx, eax
mov eax, DWORD PTR
[rbp-4]
cdqe
mov DWORD PTR
[rbp-8016+rax*4], edx
add DWORD PTR [rbp-4], 1
jmp .L3

.L2:
mov eax, 0
leave
ret
```

(b)

Fig. 4. Algorithm in C, and x86-64 instructions correspondents.

B. Performance analysis

This section compares our performance results with the performance of other common processors. Table II presents the Giga Operations per Second (GOPS) for benchmarks. The CPU baseline is the common processor Intel Core i5-4210U 1.70GHz with 4 cores and 3072K of L3 cache. For the CPU performance measurement we use C++ codes parallelized with OpenMP equivalent to the graph of each benchmark. We used g++ 8.3.1 (Red Hat 8.3.1-2) to compile these codes. The FPGA synthesis is running at 200 MHz in Intel Arria10.

TABLE II
BENCHMARKS : PERFORMANCE RESULTS IN GOPS.

Benchmark	Relevant Operations	GOPS		Speedup
		Intel i5	**Arria10**	
FIR	125	28.04	25	1.12
K-means	93	12.65	18.6	0.68
Paeth	19	5.55	3.8	1.46
Sobel	37	2.88	7.4	0.38
Poly5	27	2.99	5.4	0.55
Poly6	44	4.14	8.8	0.47
Poly8	32	2.9	6.4	0.45
Mibench	13	2.59	2.6	0.99
Sgfilter	15	2.61	3	0.87
Qspline	26	2.68	5.2	0.52
Chebyshev	7	4.2	1.4	3.00

These values represent only the results of code execution, they disregard the times of CGRA network reconfiguration and data entry to the FPGA. For this reason, the actual performance of our system is slightly smaller than the Table II shows. Moreover, so that the time of configuration and transfer of data to the FPGA has less influence in total execution time, it is substantial to ensure that we use a great proportion of the processing elements.

VI. CONCLUSIONS AND FUTURE WORK

We have created an instruction set architecture that describes the operations in the CGRA [1]. Besides to describe operations, the ISA is also responsible for mapping each instruction to a processing element. This architecture also generates the network configuration for the interconnection of the processing elements. In other words, the ISA gives all the settings required by CGRA. Then, as we aim, the contact of the programmer with the FPGA is facilitated. The programmer does not have to worry about specific details of this platform, and there is still practically no loss of performance in the execution and improvement in reconfiguration time. Our results indicate that we got a brief set of instructions, which can simplify the use of FPGAs as a generic accelerator.

We plan to search for the most commonly used instructions in the context of work, which is hardware acceleration. Once we found expected answers, we intend to add these instructions to ISA. Therefore, we consider to include instructions with immediate fields as our first step. Another interesting future project about this work is to adjust the instructions to the RISC-V specifications.

ACKNOWLEDGMENT

The authors thank CAPES, FAPEMIG, and CNPQ for the financial support. We also thank Intel Altera, and Synopsys for the software licenses. Finally we thank Paderborn Center for Parallel Computing for providing access to the CPU-FPGA platform.

REFERENCES

[1] Lucas Bragança da Silva et al. ”READY: A Fine-Grained Multithreading Overlay Framework for Modern CPU-FPGA Dataflow Applications”. ACM Trans. Embed. Comput. Syst. (To appear).

[2] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. ”WaveScalar”. In Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36). IEEE Computer Society. 2003.

[3] Li, Dong, Behnam Robatmili, Sibi Govindan, Doug Burger and Steve Keckler. Hybrid Operand Communication for Dataflow Processors. 2009.

[4] S. R. Chalamalasetti, S. Purohit, M. Margala and W. Vanderbauwhede, ”MORA - An Architecture and Programming Model for a Resource Efficient Coarse Grained Reconfigurable Processor,” NASA/ESA Conference on Adaptive Hardware and Systems. 2009.

[5] A. Smith et al., ”Compiling for EDGE architectures,” International Symposium on Code Generation and Optimization (CGO’06). 2006.

[6] Lucas Bragança et al. ”Simplifying HW/SW integration to deploy multiple accelerators for CPU-FPGA heterogeneous platforms.” In Proceedings of the 18th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS ’18). 2018.