The ChipCflow Project to Accelerate Algorithms using a Dataflow Graph in a Reconfigurable System

ANTONIO CARLOS FERNANDES DA SILVA
Federal Technological University of Parana
Coordination of Informatics
Av. Alberto Carazzai, 1640
BRAZIL
antonio@utfpr.edu.br

BRUNO DE ABREU SILVA
University of Sao Paulo
Department of Computer Systems
Av. Trabalhador Saocarlense, 400
BRAZIL
brunao@icmc.usp.br

JOELMIR JOSE LOPES
University of Sao Paulo
Department of Computer Systems
Av. Trabalhador Saocarlense, 400
BRAZIL
joelmir@icmc.usp.br

JORGE LUIZ E SILVA
University of Sao Paulo
Department of Computer Systems
Av. Trabalhador Saocarlense, 400
BRAZIL
jsilva@icmc.usp.br

Abstract: In this paper, the ChipCflow Project to accelerate algorithms using a design of a field programmable gate array (FPGA) as a prototype of a static dataflow architecture is presented. The static dataflow architecture using operators interconnected by parallel buses was implemented. Accelerating algorithms using a dataflow graph in a reconfigurable system shows the potential for high computation rates. The results of benchmarks implemented using the static dataflow architecture are reported at the end of this paper.

Key–Words: Accelerating algorithms, Reconfigurable Computing, Static Dataflow Graph, Modules C to VHDL.

1 Introduction

With the advent of reconfigurable computing, basically using a Field Programmable Gate Array(FPGA), researchers are trying to explore the maximum capacities of these devices, which are: flexibility, parallelism, optimization for power, security and real time applications [7, 19].

Because of the complexity of the applications and the large possibilities to develop systems using FPGAs, many applications to convert algorithms into these devices associated with a General Purpose Processor (GPP) using high level language like C and Java is one of the challenges for researchers nowadays, especially for accelerating algorithms [15, 16].

The main aim of the ChipCflow Project is to accelerate the algorithms which convert parts of programs written in C language into a static dataflow model implemented in a FPGA.

This paper is organized as follows. Related work is described in section 2. The Dataflow Graph Model is discussed in section 3. In section 4 the Benchmarks implemented in the Dataflow graph are presented. Section 5 shows the results of the implementations. Section 6 concludes the paper and suggests future works.

2 Related Work

The dataflow graph model and its architecture was first researched in the 1970s and was discontinued in the 1990s [2, 6, 13, 14]. Nowadays, it is a topic of research once more, mainly because of the advance of technology, particulary with the advent of the FPGA [3, 13, 19].

Because the dataflow model has an implicit parallelism and the FPGA is composed by parallel circuits, the dataflow model applied to a FPGA has the perfect combination to execute applications which also have parallelism in their execution [13]. However, as applications become more complex, software development is only possible using high level language such as C or Java [4] although only parts of the program will be executed directly into the hardware. Thus several tools have been developed to convert C into hardware using VHDL language [8, 11, 12].

In order to analyze the data dependence, many of these systems generate an intermediate dataflow graph for pipeline instructions. The optimizations, using several techniques such as loop unrolling, are concluded and finally a reconfigurable hardware using the VHDL language is generated. The hardware generated using these tools consists of coarse grain

elements or assembler instructions for a customized processor as Picoblase or Nios from Xilinx and Altera respectively [21].

In our approach, a fine grain instruction using VHDL to implement a static dataflow architecture, consisting of various nodes of processing elements and arcs to connect those nodes in a graph, is used to accelerate algorithms.

3 The Dataflow Graph Model

In the Asynchronous Dataflow Graph project developed by Teifel et al. [19], the asynchronous system is a collection of concurrent hardware processes that communicate with each other through message-passing channels. These messages consist of atomic data items called tokens. Each process can send and receive tokens to and from its environment through communication ports. In the Teifel project, asynchronous pipelines are constructed by connecting these ports to each other using channels, where each channel is allowed only one sender and one receiver. Since there is no clock in an asynchronous design, processes use handshake protocols to send and receive tokens via channels.

In Fig. 1 Teifel describes an equation converted into a dataflow graph in three different situations: (a) a pure dataflow graph, (b) a token-based asynchronous dataflow pipeline and (c) a clocked dataflow pipeline.

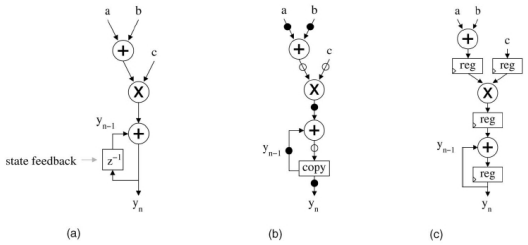

Figure 1: Computation of y_n=y_{n-1}+c(a+b):(a) pure dataflow graph, (b) token-based asynchronous dataflow pipeline (filled circles indicate tokens, empty circles indicate an absence of tokens), and (c) clocked dataflow pipeline [19].

In our project, a collection of concurrent hardware processes that communicate with each other, but using a parallel bus with bits for data and bits to control the communication in a synchronous system of communication as described in part (c) of the Fig. 1, is also used.

3.1 Dataflow Computations

In the dataflow graph to the ChipCflow Project, a traditional dataflow model described in the literature, where a node is a processing element and an arc is the connection between two elements, is used [2, 3, 6, 13, 14]. A data bus and a control bus to execute the communication between the operators were implemented. The static dataflow graph model, where only one item of data can be in an arch was developed.

In Fig. 2, a basic operator and its data buses and control buses for communication are described. The signal data *a*, *b* and *z* in Fig. 2 are 16-bit data traveling through the parallel buses. The signals *stra*, *strb*, *strz*, *acka*, *ackb* and *ackz* are 1-bit control data to control communication between operators.

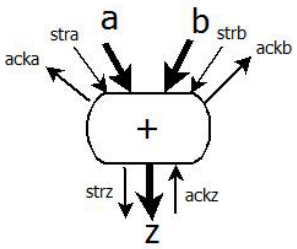

Figure 2: The basic operator with its data buses and control buses.

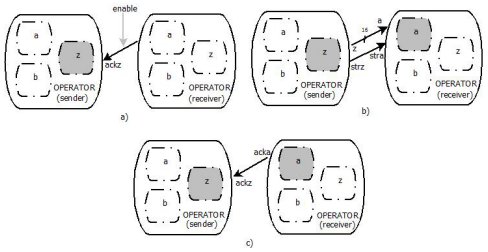

Figure 3: The communication: a) enabling the communication, b) sending an item of data, c) Acknowledging an item of data.

The communication between operators is described in Fig. 3. As can be clearly seen in the figure, a sender operator and a receiver operator have two input data buses *a* and *b*, one output data bus *z* and its respective control signals *stra*, *strb*, *strz*, *acka*, *ackb* and *ackz*. Each of the input data bus and output data bus is connected to a register to store a receiving item of data and to store a sending item of data, represented by rectangles with rounded edges denoted with the characters 1a, b and z in the figure.Each of the input data bus and output data bus is connected to a register to store a receiving item of data and to store a sending

item of data, represented by rectangles with rounded edges denoted with the characters *a*, *b* and *z* in the figure. The output data bus *z* from the sender operator is connected to input data bus *a* from the receiver operator, the output control signal *strz* from the sender operator is connected to the input control signal *stra* from the receiver operator and the input control signal *ackz* from the sender operator is connected to the output control signal *acka* from the receiver operator.

A "logic-0" in the signal *ackz* informs the sender operator that the receiver operator is ready to receive data. A "logic-1' in the signal *ackz* informs the sender operator that the receiver operator is busy. A "logic-1" in the signal *stra* informs the receiver operator that an item of data is ready to be sent to it from the sender operator. A "logic-0' in the signal *stra* informs the receiver operator that the sender have not an item of data to be sent to it.

To initiate the communication, an *enable* signal with a "logic-0" to the *ackz* connected to the sender, is set, Fig. 3a. When the receiver operator is ready to receive data, a "logic-1" in the *stra* strobes an item of data to the input data bus *a* in the receiver operator, Fig. 3b. Consequently, a "logic-0" in the *acka* acknowledges that the item of data *a* was received, Fig. 3c.

3.2 The Dataflow Operators

The dataflow operators were the traditional operators described by Veen in [14], which are: copy, non deterministic merge, deterministic merge, branch, conditional and primitive operators (add, sub, mul, div, and, or, not, etc.).

In order to execute the computation of an operator it is necessary that an item of data is presented in all its input buses of data. In Fig. 4, operators are described where filled circles indicate items of data and empty circles show an absence of items of data and the situation of the operator before computation and after computation [19].

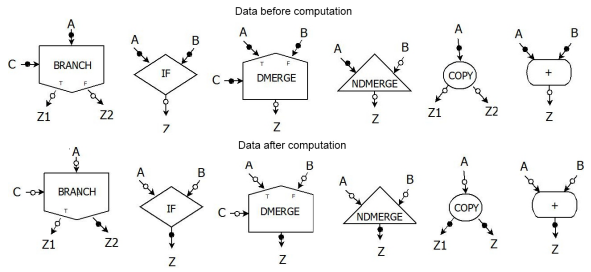

Figure 4: The operator. [19]

The functional execution of dataflow operators is described below:

1. *Copy*: This dataflow node duplicates an item of data to two receiver operators. It receives an item of data in its input data bus and copies the item of data to two output data buses.

2. *Primitive*: This dataflow node receives two item of data in its input data buses, computes the primitive operation with these two items of data and generates the result sending it to the output data bus. Operators such as add, sub, multiply, divide, and, or, not, if, etc., are implemented in the same way.

3. *Dmerge*: This dataflow node performs a two-way controlled data merge and allows an item of data to be conditionally read in input data buses. It receives a TRUE/FALSE item of data to decide what input data *a* or *b* respectively to send to the output data *z*

4. *NDmerg*: This dataflow node performs a two-way not controlled data merge and allows an item of data to be read on input data buses. The first data to arrive into the Ndmerge operator from input *a* or *b* is sent to the output data *z*.

5. *Branch*: This dataflow node performs a two-way controlled data branch and allows the item of data to be conditionally sent on to two different output buses. It receives a control TRUE/FALSE item of data to decide what output data *t* or *f* respectively to transfer the input data *a*.

3.2.1 The Basic Dataflow Operator Architecture

A register-transfer-level datapath (RTL) diagram for a sum (ADD) Operator is given in Fig. 5. In the figure, the 1-bit register *bita* and 1-bit register *bitb* are used to inform the ADD operator when the 16-bit register *dadoa* and/or 16-bit register *dadob* are filled with an item of data, respectively.

A "logic-1" in the *bita* or *bitb* informs the ADD operator that there is a item of data within *dadoa* or *dadob* respectively. A "logic-0" in *bita* or *bitb* informs the ADD operator that the *dadoa* or *dadob* is empty.

When both items of data are in the receiver operator, the ADD operator is executed and the result is filled within a 16-bit register *dadoz*. The 1-bit register *bitz* receives a "logic-1" to inform that there is a item of data to send to the next operator (the signal *strz* in Fig. 5).

The operation process of the ADD operator is described in the ASM chart in Fig. 6. In the figure, there are four described states *S0, S1, S2 and S3*. As can be

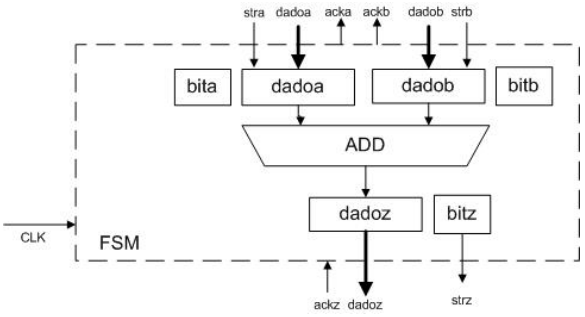

Figure 5: Datapath (RTL) Diagram of ADD Operator.

clearly seen in the figure, the initial state *S0* is used to initialize several signals of the operation process. In state *S1*, an item of data from the input data buses can be received within the operator and the correspondent bit of status can be set. Simultaneously the acknowledge signal is also set. After receiving all the items of data, the execution of the function within the operator is started, described in state *S2*. Finally, in state *S3*, several signals of the operation process are set to "logic-0" to continue the execution process of the operator.

In the process of the operator there is a Finite State Machine (FSM) that controls each step of the execution and the communication between operators.

Although there is a clock (signal CLK in Fig. 5), communication between operators is asynchronous because it is unpredictable when data will be sent to the next operator.

There are three different architectures of operators. One of them is already described in Fig. 5, with two input data buses and just one output data bus. That is the case of the primitive operators *ADD*, *SUB*, *MUL* and *DIV*; the relational operators *IFgt*, *IFge*, *IFlt*, *IFle*, *IFeq* and *IFdf*; the logic operators *AND*, *OR* and *NOT*; and the control operator *NDmerge*. Another one is the control operator *Dmerge* with three input data buses and just one output data bus. Finally the last one, the control operator *Branch* with two input data buses and two output data buses.

4 The Benchmarks Implemented in the Dataflow Model

The benchmarks implemented in the dataflow model were: Fibonacci, Max, Dot prod, Vector sum, Bubble sort, and Pop count [20]. To convert the benchmark algorithms into a VHDL, each benchmark was described as a dataflow graph, them an assembler language was used to convert the dataflow graph into a VHDL. A compiler have been developed to convert

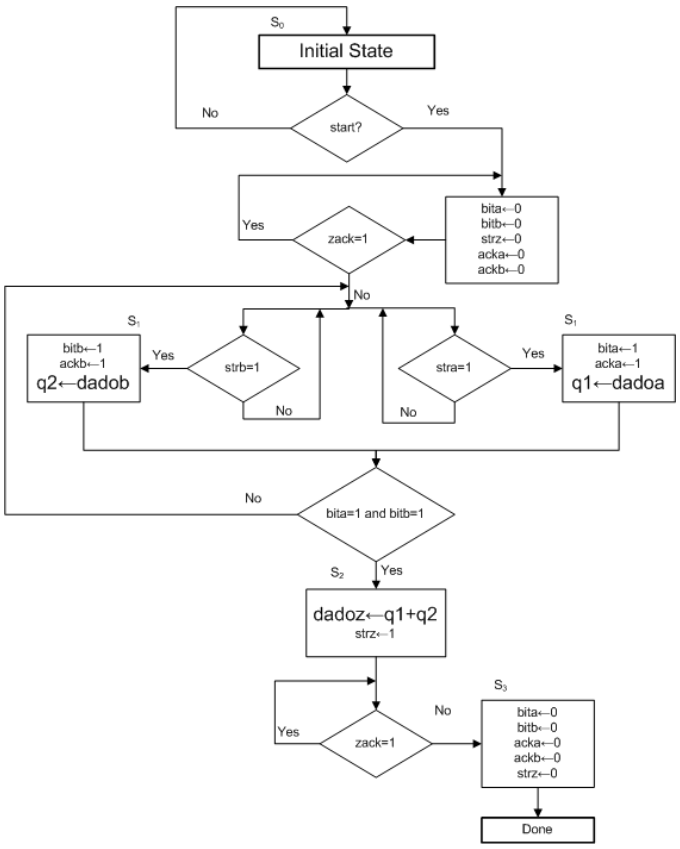

Figure 6: ASM Chart of ADD Operator.

C into a dataflow graph, but in this case, all the algorithms were converted by hand. The implementation of the benchmarks are described in the following items.

The Fibonacci Algorithm

The Fibonacci algorithm is described in Algorithm 1 and its dataflow graph is described in Fig. 7.

Algorithm 1 Calculate Fibonacci

$first \Leftarrow 0$
$second \Leftarrow 1$
$tmp \Leftarrow 0$
for $i = 0$ to n **do**
  $tmp \Leftarrow first + second$
  $first \Leftarrow second$
  $second \Leftarrow tmp$
end for

As can be clearly seen in Fig. 7, there are two parts in the dataflow graph: one of them is located on the left side of the figure and controls the loop with

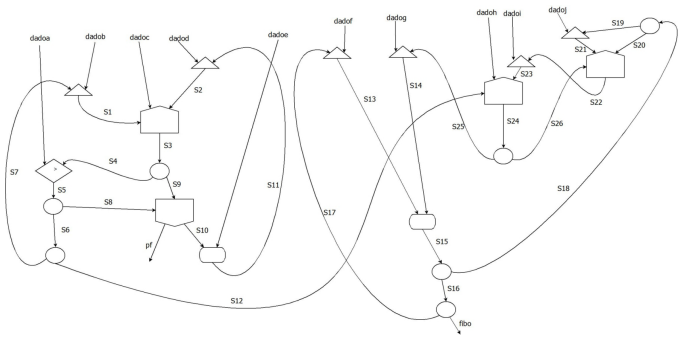

Figure 7: The Fibonacci algorithm described in Dataflow Graphics.

index *i*; on the right side of the figure the implementation of the Fibonacci sequence is described.

As the dataflow graph consist of nodes and arcs, each node represents an operator and each arc represents the communications between two operators. In Fig. 7, a label is attributed to each arc in the dataflow graph. As arcs represent the communication between two operators, the parallel data bus for items of data and the control data bus for control the communications are included in the label representations. The assembler language uses the name of the operator and its label arcs to convert the dataflow graph into a VHDL. The assembly language for Fibonacci dataflow graph is described in Listing 1.

As can be clearly seen in Listing 1, several node operators and their input and output arcs are listed. Labels used to connect nodes operators are described initializing with the *s* character followed by a number and the others are input or output data signals. The same organization is used for the others benchmarks implementation.

In the Listing 1 the labels *dadoa*, *dadob*, *dadoc*, *dadod*, *dadoe*, *dadof*, *dadog*, *dadoh*, *dadoi* and *dadoj* are input data signals used to initialize data for the Fibonacci dataflow graph and the label *fibo* is output data signal to inform the result of the Fibonacci sequence. Specifically for the Fibonacci sequence, *dadoa* receives and maintain the *n* Fibonacci argument; *dadob* and *dadoc* receive "logic-0" to initialize the value for "i" in the *for* command; *dadod* receives "logic-0" and *dadoe* receives and maintain "logic-1" to control the next value for "i"; *dadof* receives "logic-1' and *dadog*,*dadoh*,*dadoi* and *dadoj* receive "logic-0" to initialize the Fibonacci algorithm. Finally, the outputs *fibo* is the output data for the Fibonacci Algorithm.

Listing 1: *The Assembler Language for Fibonacci Dataflow Graph*

```
1.ndmerge s7,dadob,s1;
2.dmerge s2,dadoc,s1,s3;
3.ndmerge dadod,s11,s2;
4.gtdecider dadoa,s4,s5;
5.copy s3,s4,s9;
6.copy s5,s6,s8;
7.branch s9,s8,s10,pf;
8.copy s6,s7,s12;
9.add s10,dadoe,s11;
10.ndmerge s17,dadof,s13;
11.ndmerge dadog,s25,s14;
12.ndmerge dadoi,s22,s23;
13.ndmerge dadoj,s19,s21;
14.copy s18,s19,s20;
15.dmerge s23,dadoh,s12,s24;
16.dmerge s20,s21,s26,s22;
17.copy s24,s25,s26;
18.add s13,s14,s15;
19.copy s15,s16,s18;
20.copy s16,s17,fibo;
```

The Bubble Sort algorithm

The Bubble Sort algorithm is described in Algorithm 2 and its dataflow graph is described in Fig.8.

As can be clearly seen in Fig. 8, also there are two parts in the dataflow graph: one of them is located on the left side of the figure and controls the loop with index *i*; on the right side of the figure the implementation of the Bubble Sort sequence is described.

Algorithm 2 Calculate Bubble Sort

for $i = 0$ to n **do**
  for $j = 0$ to $n - 1$ **do**
    if $a[j] > a[j+1]$ **then**
      $aux \Leftarrow a[j]$
      $a[j] \Leftarrow a[j+1]$
      $a[j+1] \Leftarrow aux$
    end if
  end for
end for

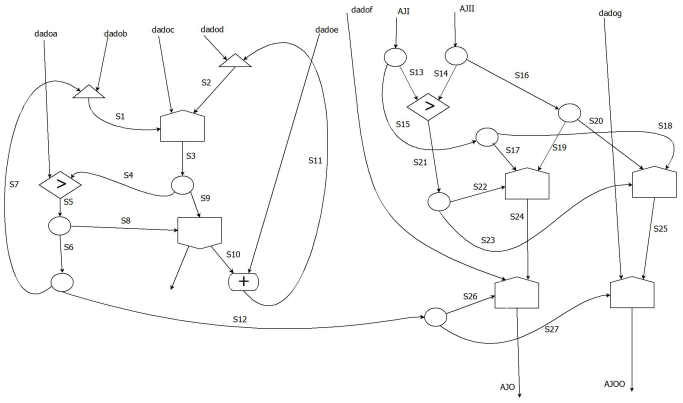

Figure 8: The Bubble Sort algorithm described in Dataflow Graphics.

The assembly language for Bubble Sort dataflow graph is described in Listing 2.

According on the Listing 2, the labels *dadoa*, *dadob*, *dadoc*, *dadod*, *dadoe*, *dadof*, *dadog*, *aji* and *ajii* are input data signals used to initialize data for the Bubble Sort dataflow graph and the labels *ajo* and *ajoo* are output data signal to inform the result of the Bubble Sort sequence. Specifically for the Bubble Sort sequence, *dadoa* receives and maintain the *n* Bubble Sort argument; *dadob* and *dadoc* receive "logic-0" to initialize the value for "i" in the *for* command; *dadod* receives "logic-0" and *dadoe* receives and maintain "logic-1" to control the next value for "i"; *dadof* and *dadog* receive "logic-0' to initialize the Bubble Sort algorithm. Finally, the inputs *aji* and *ajii* correspond to *a[j]* and *a[j+1]* respectively in the input data vector for Bubble Sort algorithm, and the outputs *ajo* and *ajoo* correspond to *a[j]* and *a[j+1]* respectively in the output data vector for Bubble Sort algorithm.

Listing 2: *The Assembler Language for Bubble Sorte Dataflow Graph*

```
1.ndmerge s7,dadob,s1;
2.ndmerge dadod,s11,s2;
3.dmerge dadoc,s2,s1,s3;
4.copy s3,s4,s9;
5.gtdecider dadoa,s4,s5;
6.copy s5,s6,s8;
7.copy s6,s7,s12;
8.branch s9,s8,pfi,s10;
9.add s10,dadoe,s11;
10.copy aji,s15,s13;
11.copy ajii,s14,s16;
12.gtdecider s13,s14,s21;
13.copy s16,s19,s20;
14.copy s15,s17,s18;
15.copy s21,s23,s22;
16.dmerge s19,s17,s22,s24;
17.dmerge s18,s20,s23,s25;
18.copy s12,s27,s26;
19.dmerge s24,dadof,s26,ajo;
20.dmerge s25,dadog,s27,ajoo;
```

The Dotprod algorithm

The Dotprod algorithm is described in Algorithm 3 and its dataflow graph is described in Fig.9.

As can be clearly seen in Fig. 9, also there are two parts in the dataflow graph: one of them is located on the left side of the figure and controls the loop with index *i*; on the right side of the figure the implementation of the Dotprod sequence is described.

Algorithm 3 Calculate Dotprod

$dotprod \Leftarrow 0$
for $i = 0$ to $n - 1$ **do**
  $dotprod \Leftarrow dotprod + x[i] * y[i]$
end for

The assembly language for Dotprod dataflow graph is described in Listing 3.

According on the Listing 3, the labels *dadoa*, *dadob*, *dadoc*, *dadod*, *dadoe*, *dadof*, *dadog*, *xi* and *yi*

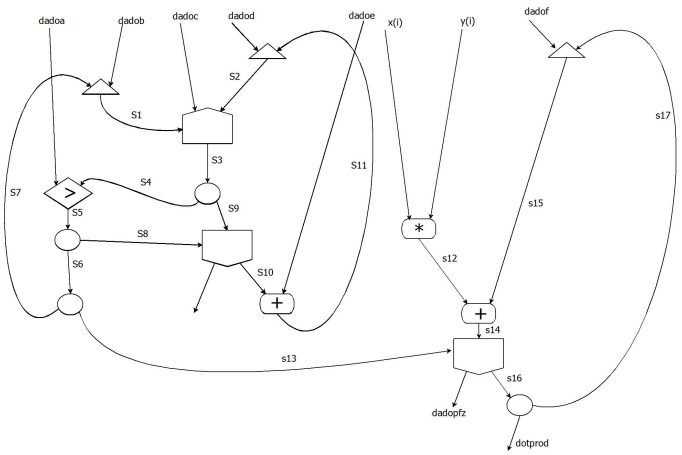

Figure 9: The Dotprod algorithm described in Dataflow Graphics.

are input data signals used to initialize data for the Dotprod dataflow graph and the label *dotprod* is output data signal to inform the result of the Dotprod sequence. Specifically for the Dotprod sequence, *dadoa* receives and maintain the *n* Dorprod argument; *dadob* and *dadoc* receive "logic-0" to initialize the value for "i" in the *for* command; *dadod* receives "logic-0" and *dadoe* receives and maintain "logic-1" to control the next value for "i"; *dadof* receives "logic-0' to initialize the Dotprod algorithm. Finally, the inputs *xi* and *yi* correspond to *x[i]* and *y[i]* respectively in the input data vector for Dotprod algorithm, and the output *dotprod* correspond to *dotprod* in the output data for Dotprod algorithm.

Listing 3: *The Assembler Language for Dotprod Dataflow Graph*

```
1.ndmerge s7,dadob,s1;
2.ndmerge dadod,s11,s2;
3.dmerge dadoc,s2,s1,s3;
4.copy s3,s4,s9;
5.gtdecider dadoa,s4,s5;
6.copy s5,s6,s8;
7.copy s6,s7,s13;
8.branch s9,s8,pfi,s10;
9.add s10,dadoe,s11;
10.ndmerge dadof,s17,s15;
11.mul xi,yi,s12;
12.add s12,s15,s14;
13.branch s14,s13,pfz,s16;
14.copy s16,dotprod,s17;
```

The Max algorithm

The Max algorithm is described in Algorithm 4 and its dataflow graph is described in Fig.10.

As can be clearly seen in Fig. 10, also there are two parts in the dataflow graph: one of them is located on the left side of the figure and controls the loop with index *i*; on the right side of the figure the implementation of the Max sequence is described.

Algorithm 4 Calculate Max

$maxval \Leftarrow 0$
for $i = 0$ to $n - 1$ **do**
  if $maxval < v[i]$ **then**
    $maxval \Leftarrow v[i]$
  end if
end for

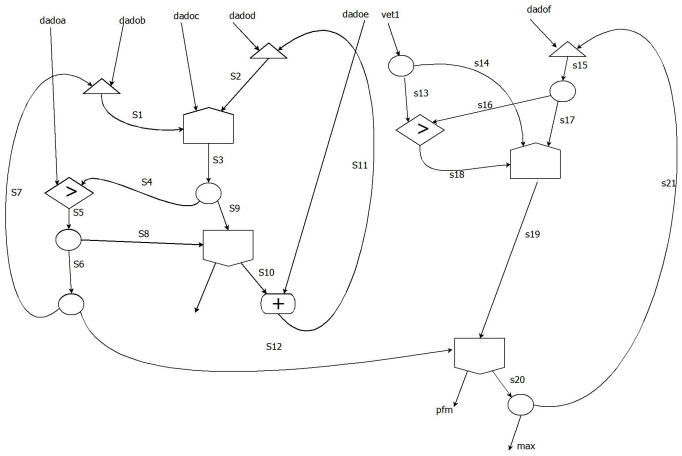

Figure 10: The Max algorithm described in Dataflow Graphics.

The assembly language for Max dataflow graph is described in Listing 4.

According on the Listing 4, the labels *dadoa*, *dadob*, *dadoc*, *dadod*, *dadoe*, *dadof* and *vet1* are input data signals used to initialize data for the Max dataflow graph and the label *max* is output data signal to inform the result of the Max sequence. Specifically for the Max sequence, *dadoa* receives and maintain the *n* Max argument; *dadob* and *dadoc* receive "logic-0" to initialize the value for "i" in the *for* command; *dadod* receives "logic-0" and *dadoe* receives and maintain "logic-1" to control the next value for "i"; *dadof* receives "logic-0' to initialize the Max algorithm. Finally, the input *vet1* correspond to *v[i]* in the input data vector for Max algorithm, and the output *max* correspond to *maxval* in the output data for Max algorithm.

Listing 4: *The Assembler Language for Max Dataflow Graph*

```
1.ndmerge s7,dadob,s1;
2.ndmerge dadod,s11,s2;
3.dmerge dadoc,s2,s1,s3;
4.copy s3,s4,s9;
5.gtdecider dadoa,s4,s5;
6.copy s5,s6,s8;
7.copy s6,s7,s12;
8.branch s9,s8,pfi,s10;
9.add s10,dadoe,s11;
10.ndmerge dadof,s21,s15;
11.copy vet1,s13,s14;
12.gtdecider s13,s16,s18;
13.copy s15,s16,s17;
14.dmerge s17,s14,s18,s19;
15.branch s19,s12,pfm,s20;
16.copy s20,max,s21;
```

The Pop Count algorithm

The Pop Count is described in Algorithm 5 and its dataflow graph is described in Fig.11.

As can be clearly seen in Fig. 11, also there are two parts in the dataflow graph: one of them is located on the left side of the figure and controls the loop with index *i*; on the right side of the figure the implementation of the Pop Count sequence is described.

Algorithm 5 Calculate Pop Count

for $i = 0$ to $n - 1$ **do**
  $input \Leftarrow a[i]$
  $sum \Leftarrow 0$
  for $j = 0$ to 15 **do**
    $sum \Leftarrow sum + (input)\&1$
    $input \Leftarrow input/2$
  end for
  $b[I] \Leftarrow sum$
end for

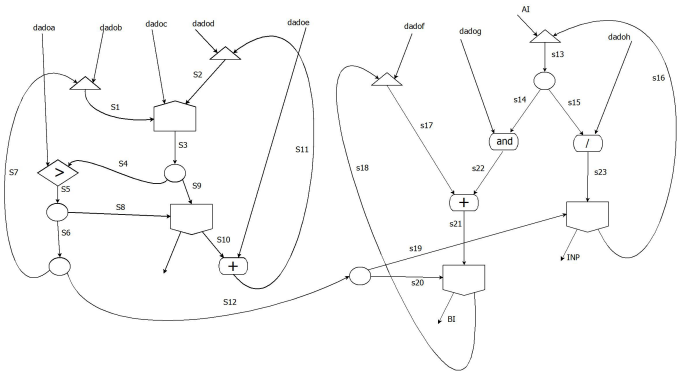

Figure 11: The Pop Count algorithm described in Dataflow Graphics.

The assembly language for Pop Count dataflow graph is described in Listing 5.

According on the Listing 5, the labels *dadoa*, *dadob*, *dadoc*, *dadod*, *dadoe*, *dadof*, *dadog*, *dadoh* and *ai* are input data signal used to initialize data for the Pop Count dataflow graph and the label *bi* is output data signal to inform the result of the Pop Count sequence. Specifically for the Pop Count sequence, *dadoa* receives and maintain the *n* Pop Count argument; *dadob* and *dadoc* receive "logic-0" to initialize the value for "i" in the *for* command; *dadod* receives "logic-0" and *dadoe* receives and maintain "logic-1"

to control the next value for "i"; *dadof* receives "logic-0', *dadog* receive "logic-1" and *dadoh* receive and mantain "logic-10" to initialize the Pop Count algorithm. Finally, the inputs *ai* correspond to *a[i]* in the input data vector for Pop Count algorithm, and the outputs *bi* correspond to *b[i]* in the output data vector for Pop Count algorithm.

Listing 5: *The Assembler Language for Pop Count Dataflow Graph*

```
1.ndmerge s7,dadob,s1;
2.ndmerge dadod,s11,s2;
3.dmerge dadoc,s2,s1,s3;
4.copy s3,s4,s9;
5.gtdecider dadoa,s4,s5;
6.copy s5,s6,s8;
7.copy s6,s7,s12;
8.branch s9,s8,pfi,s10;
9.add s10,dadoe,s11;
10.ndmerge s18,dadof,s17;
11.ndmerge ai,s16,s13;
12.copy s13,s14,s15;
13.andi dadog,s14,s22;
14.add s17,s22,s21;
15.div s15,dadoh,s23;
16.copy s12,s20,s19;
17.branch s21,s20,bi,s18;
18.branch s23,s19,inp,s16;
```

The Vector Sum algorithm

The Vector Sum algorithm is described in Algorithm 6 and its dataflow graph is described in Fig.12.

As can be clearly seen in Fig. 12, also there are two parts in the dataflow graph: one of them is located on the left side of the figure and controls the loop with index *i*; on the right side of the figure the implementation of the Vector Sum sequence is described.

Algorithm 6 Calculate Vector Sum

for $i = 0$ to $n - 1$ **do**
  $c[i] \Leftarrow a[i] + b[i]$
end for

The assembly language for Vector Sum dataflow graph is described in Listing 6.

According on the Listing 6, the labels *dadoa*, *dadob*, *dadoc*, *dadod*, *dadoe*, *vet1* and *vet2* are input data signals used to initialize data for the Vector sum dataflow graph and the label *vet3* is output data signal to inform the result of the Vector sum sequence. Specifically for the Vector sum sequence, *dadoa* receives and maintain the *n* Vector sum argument; *dadob* and *dadoc* receive "logic-0" to initialize the value for "i" in the *for* command; *dadod* receives "logic-0" and *dadoe* receives and maintain "logic-1" to control the next value for "i". Finally, the inputs *vet1* and *vet2* correspond to *a[i]* and *b[i]* respectively in the input data vector for Vector sum algorithm, and the outputs *vet3* correspond to *c[i]* in the output data vector for Vector sum algorithm.

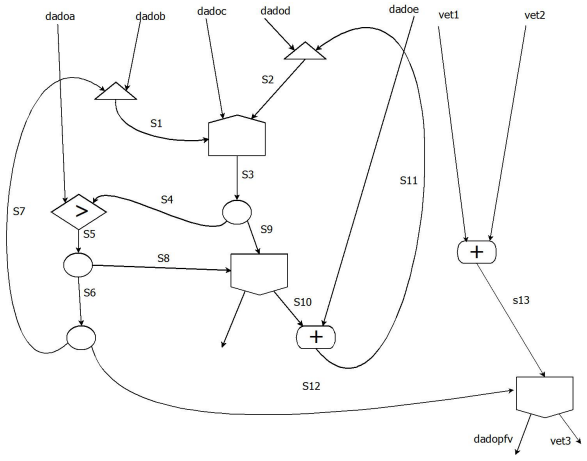

Figure 12: The Vector Sum algorithm described in Dataflow Graphics.

Listing 6: *The Assembler Language for Vector Sum Dataflow Graph*

```
1.ndmerge s7,dadob,s1;
2.ndmerge dadod,s11,s2;
3.dmerge dadoc,s2,s1,s3;
4.copy s3,s4,s9;
5.gtdecider dadoa,s4,s5;
6.copy s5,s6,s8;
7.copy s6,s7,s12;
8.branch s9,s8,pfi,s10;
9.add s10,dadoe,s11;
10.add vet1,vet2,s13;
11.branch s13,s12,pfv,vet3;
```

5 Experimental Results

The benchmarks were implemented using a (7v285tffg1157-3) Virtex FPGA from Xilinx and synthesized in ISE 13.1 and the results were compared with the same benchmarks implemented in C-to-Verilog and LALP described in [20] that were implemented using a (EP1S10F780C6) Stratix FPGA from Altera and synthesized in Quartus II 6.1.

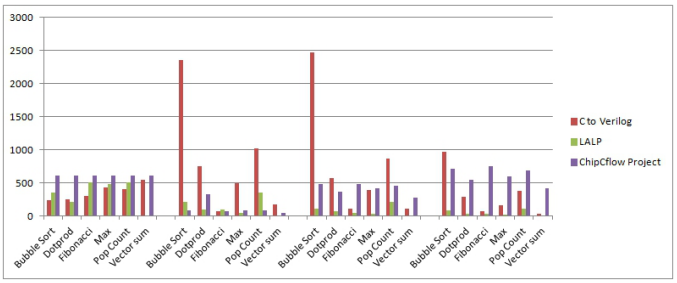

Figure 13: comparing the Benchmarks

In Table 1 the results of implementations for each benchmark in C-to-Verilog, LALP and Acceleration

Table 1: The results of implementation for Benchmarks

	Benchmarks	FF	LUT	Slices	Mas Freq.
C-to-Verilog	Buble Sort	2353	2471	971	239.45
	Dot prod	758	578	285	249.36
	Fibonacci	73	108	69	297.81
	Max vector	496	392	164	435.9
	Pop count	1023	872	384	411.22
	Vector sum	177	113	34	546.538
LALP	Buble Sort	219	105	79	353.16
	Dot prod	97	69	32	213.14
	Fibonacci	104	41	30	505.08
	Max vector	50	39	20	484.97
	Pop count	350	215	115	503.73
	Vector sum	——	——	——	——
ChipCflow Project	Buble Sort	85	485	712	613.685
	Dot prod	323	362	542	613.685
	Fibonacci	72	482	755	612.108
	Max vector	80	425	598	613.685
	Pop count	79	453	684	613.685
	Vector sum	52	284	419	613.685

Algorithms are described. In Fig. 13, a synthesis of the results is described.

As can be clearly seen in Fig. 13, the Acceleration Algorithms occupy less Flip Flops (FF) than the C-to-Verilog system, but more than the LALP system, for all the benchmarks. For LUT occupancy, the Acceleration Algorithms occupy less LUTs than the C-to-Verilog system, except for the Fibonacci, Max and Vector sum benchmarks, but more than the LALP system, also for all the benchmarks. In the Slices occupancy, the Acceleration Algorithms occupy more slices than the C-to-Verilog and the LALP system (except for the Bubble sort benchmark). Finally, for Maximum Frequency, the Acceleration Algorithms had more speed than the other two systems.

6 Conclusion and Future Work

The ChipCfow Project to accelerate Algorithms, by and large, occupy more space within the FPGA than the C-to-Verilog and the LALP system. However, the ChipCflow Project have more speed than the other two systems, although the main aim in this project was to validate the implementation model likely to convert algorithms into the dataflow graph and into a VHDL. Taking this into account, the ChipCflow Project become one more solution for parallelism in FPGA. The benchmarks used in this paper basically perform operations using vectors, but it is very important to explore the maximum parallelism of the dataflow graph using real parallel applications. Future work would be to develop a module to convert C directly into a VHDL, associated with the FPGA and to implement a dynamic dataflow model to obtain a better performance than the static model implemented in this paper.

References:

[1] Arnold, J (1993) The SPLASH 2 Software Evironment, *IEEE Workshop on FPGAs for Custom Computing Machines*, IEEE, USA, pp. 88–93.

[2] Arvind (2005) Dataflow: Passing the token, *The 32th Annual International Symposium on Computer Architecture (ISCA Keynote)*, ACM, Madison, USA, pp. 1-42.

[3] Cappelli, Andrea and Lodi, Andrea and Mucci, Claudio and Toma, Mario and Campi, Fabio. (2004) A Dataflow Control Unit for C-to-Configurable Pipelines Compilation Flow, *IEEE Sumposium on Field-Programmable Custom Computing Machines FCCM'04*, IEEE, USA, pp. 323–333.

[4] Cardoso, J., H. Neto (2003) Compilation for FPGA Based Reconfigurable Hardware, *IEEE Design Test of Computers*, IEEE, pp. 65–75.

[5] Choi, J. et al (2003) Compilation Approach for Coarse-Grained Reconfigurable Architecture, *IEEE Design Test of Computers*, IEEE, pp. 26–33.

[6] Dennis, Jack B. and Misunas, David P. (1974) A preliminary architecture for a basic dataflow processor, *Computer Architecture News - SIGARCH'74*, ACM, USA, pp. 126–132.

[7] Hauck, S. (2000) The Roles of FPGAs in Reprogrammable Systems, *Proceedings of the IEEE*, IEEE, pp. 615-638.

[8] ImpulseC (2005) *Impulse Accelerated Technologies, Inc-ImpulseC From C software to FPGA hardware* , ImpulseC.

[9] Mei, B. et al (2005) Architecture Exploration for a Reconfigurable Architecture Template, *IEEE Design Test of Computers*, IEEE, pp. 90-101.

[10] Resano, J. et al (2005) A Reconfiguration Manager for Dynamically Reconfigurable Hardware, *IEEE Design Test of Computers* , IEEE, pp. 452-460.

[11] Spark (2004) *User Manual for the SPARK Parallelizing High-Level Synthesis Framework Version 1.1*, Center for Embedded Computer Systems.

[12] Suif (2006) *The Stanford SUIF Compiler Group* , Suifcompiler system.

[13] Swanson, S. and Schwerin, A. and Mercaldi, M. and Petersen, A. and Putnam, A. and Michelson, K. and Oskin, M. and Eggers, S. J. (2007) The Wavescalar Architecture, *ACM Transactions on Computer Systems*, ACM, pp. 4:1-4:54.

[14] Veen, A. H. (1986) Dataflow Machine Architecture, *ACM Computing Surveys*, ACM, pp. 365–396.

[15] Chen, Z. and Pittman, R. N. and Forin, A. (2010) Combining multicore and reconfigurable instruction set extensions, *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays - FPGA'10*, ACM, USA, pp. 33–36.

[16] Hefenbrock, D. and Oberg, J. and Thanh, N. T. N. and Kastner, R. and Baden, S. B (2010) Accelerating Viola-Jones Face Detection to FPGA-Level Using GPUst , *18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, ACM, USA, pp. 11–18.

[17] Campi, F. (2003) A VLIW Processor With Reconfigurable Instruction Set for Embedded Applications, *IEEE Journal of Solid-State Circuits*, IEEE, pp. 1876–1886.

[18] Epalza, M. (2004) Adding Limited Reconfigurability to Superscalar Processors, *Parallel Architecture and Compilation Techniques (PACT2004)*, ACM, France, pp. 53–62.

[19] Teifel, J. Rajit, M.(2004) An asynchronous dataflow FPGA architecture, *IEEE Transactions on Computers*, IEEE, pp. 1376–1392.

[20] Menotti, R. and Cardoso, J. M. P. (2010) Agressive Loop Pipelining for Reconfigurable Architecture, *IEEE Internationl Conference of Field Programmable Logic*, IEEE, Italy, pp. 501–502.

[21] Bobda, C. (2007) *Introduction to Reconfigurable Computing* , Springer.

[22] Andrea Lodi, Mario Toma, Fabio Campi, Andrea Cappelli, Roberto Canegallo, and Roberto Guerrieri(2003) A VLIW Processor With Reconfigurable Instruction Set for Embedded Applications, *IEEE Journal of Solid-State Circuits*, IEEE, USA, pp. 1876–1886.

[23] Epalza, Marc and Ienne, Paolo and Mlynek, Daniel (2004) Adding Limited Reconfigurability to Superscalar Processors, *13th International Conference on Parallel Architectures and Compilation Techniques PACT04*, IEEE, USA, pp. 53–62.