The C Compiler generating a Source file in VHDL for a Dynamic Dataflow Machine being executed direct into a Hardware

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

KELTON A.P. DA COSTA
University of Sao Paulo
School of Engineering of Sao Carlos
Av. Trabalhador Saocarlense, 400
BRAZIL
kelton@eesc.usp.br

VALENTIN OBAC RODA
University of Sao Paulo
School of Engineering of Sao Carlos
Av. Trabalhador Saocarlense, 400
BRAZIL
valentin@eesc.usp.br

Abstract: In order to convert High Level Language (HLL) into hardware, a Control Dataflow Graph (CDFG) is a fundamental element to be used. Related to this, Dataflow Architecture, can be obtained directly from the CDFG. The ChipCflow project is described as a system to convert HLL into a dynamic dataflow graph to be executed in dynamic reconfigurable hardware, exploring the dynamic reconfiguration. The ChipCflow consists of various parts: the compiler to convert the C program into a dataflow graph; the operators and its instances; the tagged-token; and the matching data. In this paper, a C compiler to convert C into a dataflow graph and the graph implementation in VHDL is described. Some results are presented in order to show a proof-of-concept for the project.

Key–Words: C Compiler; Dynamic Dataflow Architecture; Dynamic Reconfigurable Hardware; Tagged-token; Matching-Data.

1 Introduction

A Dataflow Architecture is an architecture where a natural parallelism is present. This kind of architecture was first researched in the 1970s and was discontinued in the 1990s (5; 10; 14). With the advance of technology of microelectronics, the Field Programable Gate Array (FPGA) has been used, mainly because of its flexibility, the facilities to implement complex systems and intrinsic parallelism. Thus, dataflow architecture is a topic which has come to light again (7; 13), especially because of the reconfigurable architecture, which is totally based on FPGAs. On the other hand, much work is being done to covert high level language as a C language into hardware, in order to help engineers to project their systems using a high level of abstraction as well as a digital logic level. In particular, the ChipCflow project is a system where a C program is initially converted into a Dynamic Dataflow graph, followed by its execution in Reconfigurable Hardware. Its flow diagram is shown in Figure 1. As can be clearly seen in Figure 1, the ChipCflow system begins in a host machine where a C program is edited, to be converted into a control dataflow graph (CDFG) generating a CDFG object program. The CDFG object program is converted into a VHDL where modules of CDFG are accessed from a data base of VHDL modules. After generating the complete VHDL program, an EDA tool to convert the VHDL program into a bitstream and to download it to a FPGA is used.

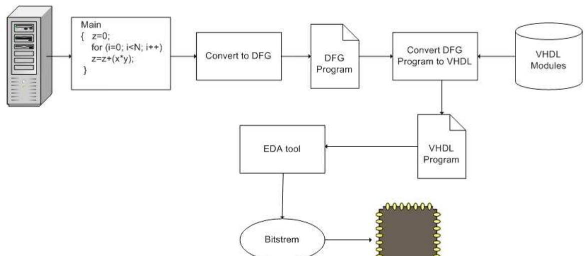


Figure 1: The Flow Diagram for ChipCflow tool.

2 The Operators and C statements implemented in Dataflow Graphs

The operators to be used in the ChipCflow project are: ”decider”, ”non deterministic merge”, ”deterministic merge”, ”branch”, ”copy” and ”operator”. They are described in Figure 2.

As can be clearly seen in Figure 2, the ”decider” operator will be used to generate a control signal

”TRUE” or ”FALSE” after to execute a boolean operation as ”<,>,=,=/,>=,<=”. The ”non deterministic merge” will be used to forward a item of data coming into the operator. Otherwise, in ”deterministic merge” the forward depends on the control signal. The ”branch” operator will be used to forward a item of data through the ”TRUE” or ”FALSE” arc. The ”copy” operator will duplicate a item of data. Finally an ”operator” will be used to generate a result after to execute arithmetic operations as ”$+,-,\star,/,\star\star$”.

The dataflow graph of the *While* statement was implemented using these operators and is described in Figure 3. In Figure 3 there are two branch operator; two deterministic merge; five copy; one decider with boolean operator ”>” and two operators with arithmetic operation ”+”.

In the next section the basic structure for the C compiler and some examples of graphs are presented.

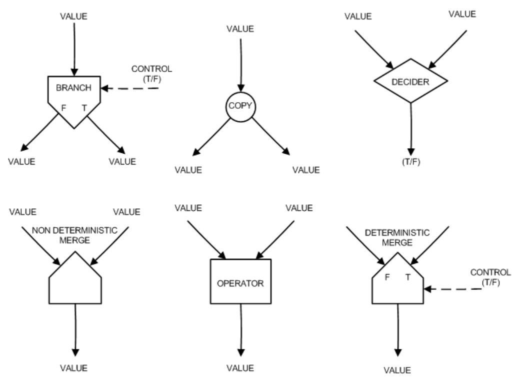


Figure 2: Operators of the Dynamic Dataflow Model.

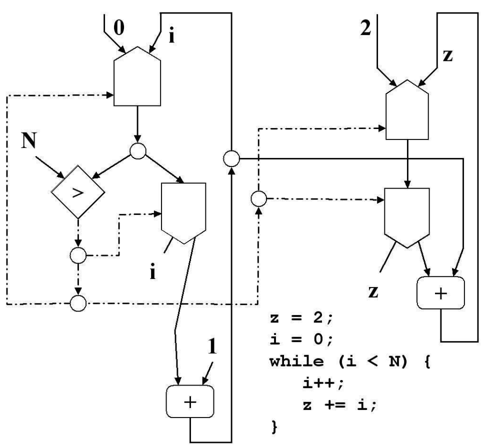


Figure 3: Graph extracted from a while command.

3 The C Compiler to generate dataflow graph

The compiler structure was implemented in C++ and it is made up of two main parts: lexical analysis and VHDL code generator. The lexical analysis part is performed over the original C code and for each letter, number and reserved word, a code scanning generate a token. After the lexical analysis, and code scanning, a file with all the tokens are generated to be converted into a VHDL file.

3.1 Generating a Binary Mapping File after Lexical Analysis

To generate a Binary Mapping file, a token was defined and its format is described in the Figure 4. As can be clearly seen in the Figure 4, the first 4-bits of the token has been used to identify the operator; the second, the thirty and the fourth 5-bits have been used to identify the three inputs *(a,b and c)* of the operator; finally the sixth and the seventh 5-bits to identify the outputs *(s and z)* of the operator. This is a generic template for the operator with three inputs and two outputs signal, however there are operators which less than three inputs signals and just one output signal.

operator	Input a	Input b	Input c	output s	output z

Figure 4: The format of token.

An example of the dataflow graph and its packets of bits for a *While* C command is describes in Figure 5. It is shown in the Figure 5 that each operator has a set of bits to identify its function, as well as each arc has a set of bits to identify its interconnections. In particular, in the left top of the figure has an operator with the code ”0001” and its arcs ”00000” *(value ”0”)*, ”00001” *(value ”i”)*, ”00010” *(a control signal)* and ”00011” *(the output signal)*, corresponding to three input signals and one output signal respectively. The packet of bits for this particular operator can be clearly seen in the first packet of bits in the Figure 6, that is accorded on the format described in Figure 4. The ”xxxxx” in the packet of bits represent an arc with no connection signal. Thus, a file with these packets of bits, is a binary representation for a dataflow graph extracted from a while C statement in the C pre-compiler.

The next step is to use the binary representation to

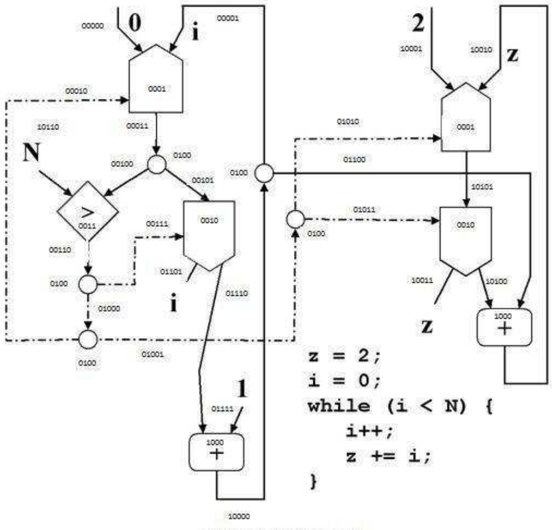


Figure 5: A While C command and its tags.

```
0001000000000010001000011XXXXX
000110001100100101010101XXXXX
00100010100111XXXXX0110101110
00101010101011XXXXX1001110100
00111011000100XXXXX00110XXXXX
10000111001111XXXXX10000XXXXX
10001010001100XXXXX10010XXXXX
010000011XXXXXXXXXX0010000101
010000110XXXXXXXXXX0100000111
010001000XXXXXXXXXX0001001001
010010000XXXXXXXXXX0000101100
010001001XXXXXXXXXX0101001011
```

Figure 6: The Binary Mapping File generated for While C command.

identify the VHDL operators, what components will be used and what instances and their interconnections will be generated for the execution of the VHDL program in the ISE Xilinx platform. This step is described in the nest item.

3.2 Generating a VHDL file

The operators described for the ChipCflow project were implemented and tested and are described in (16). In order to generate a VHDL program, the file with a C program is converted into a binary mapping representation and then converted into a VHDL code. Various examples of C program and their representations in VHDL, were implemented and are described below.

In Algorithm 1 is described an *IF* command.

Algorithm 1 The IF Command

if $x > 0$ **then**
  $z \Leftarrow a + b$
else
  $z \Leftarrow c - d$
end if

The correspondent dataflow graph for the *IF* command is described in Figure 7 that is just a graphic representation. The binary mapping file for the *IF* command is described in Figure 8. Using the binary mapping file it is relatively simple to generate a source

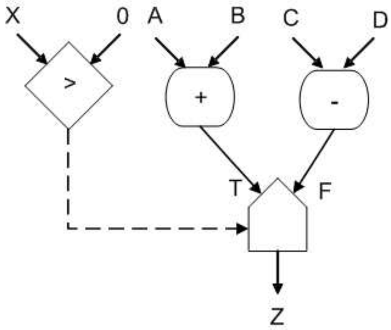


Figure 7: The Dataflow Graph for an *IF* command.

file in VHDL. The VHDL file for *IF* command is described in Listing 1. As can be clearly seen in the Listing 1, all the components and instances were generated as a source file in VHDL.

In Algorithm 2 is described an *WHILE* command; the correspondent dataflow graph for the *WHILE* command is described in Figure 9; the respective binary mapping file for the *WHILE* command is described in Figure 10 and finally the VHDL file for *WHILE* command is described in Listing 2.

In Algorithm 3 is described an *IF, FOR, and WHILE* command; the correspondent dataflow graph for the *IF, FOR, and WHILE* command is described in Figure 11; the respective binary mapping file for the *IF, FOR, and WHILE* command is described in Figure 12 and the VHDL file for *IF, FOR, and WHILE* command is described in Listing 3.

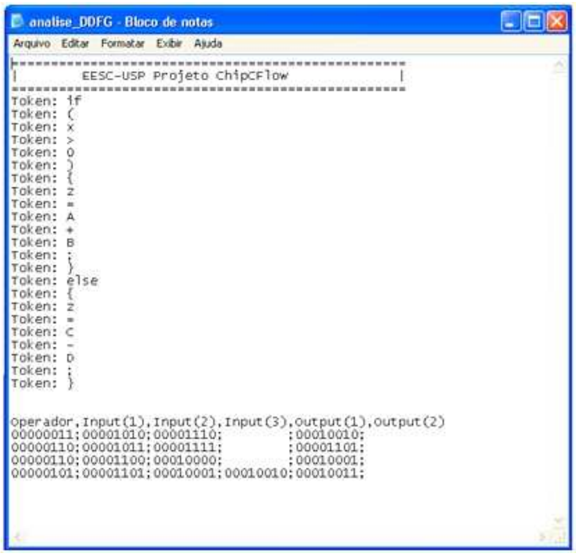


Figure 8: The Binary Mapping File for *IF* command.

Algorithm 2 The $WHILE$ Command

$x \Leftarrow 3$
while $x > 1$ **do**
  $x \Leftarrow x + 1$
  $y \Leftarrow y + x$
  $z \Leftarrow x + y$
end while

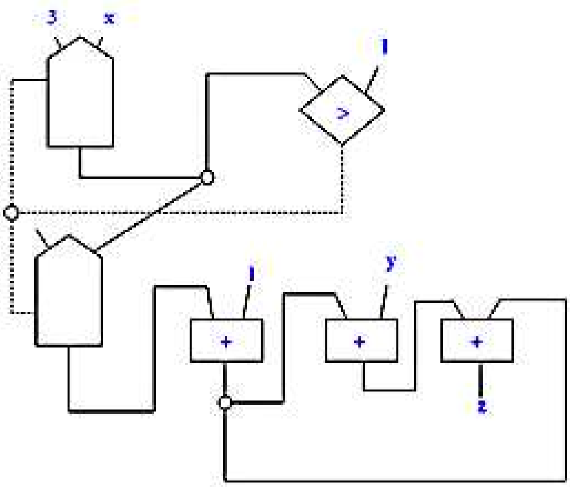


Figure 9: The Dataflow Graph for an *WHILE* command.

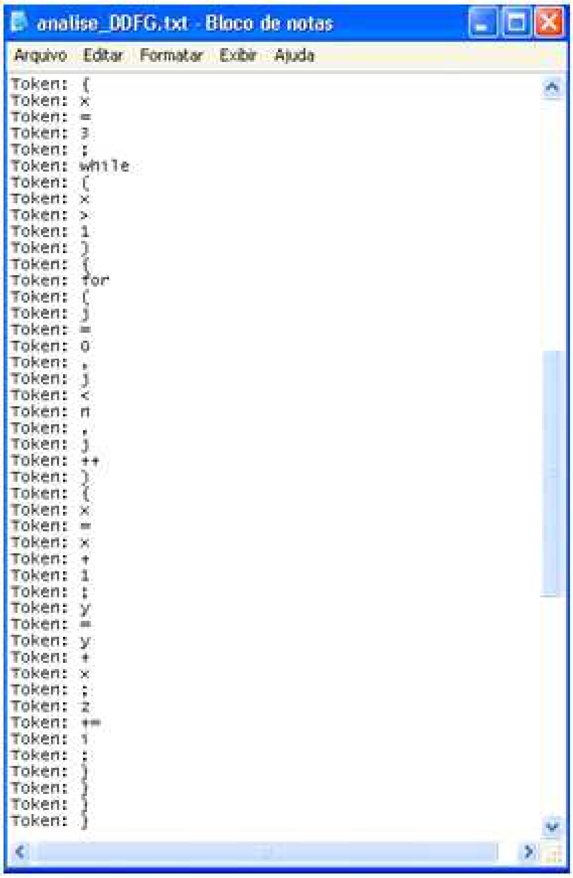


Figure 10: The Binary Mapping File for *WHILE* command.

Algorithm 3 The $IF, FOR, WHILE$ Command

$a \Leftarrow 3$
$b \Leftarrow 0$
$y \Leftarrow 1$
$z \Leftarrow 2$
for $I = 0$ to n **do**
  if $z > 1$ **then**
    $z \Leftarrow a + b$
  else
    $x \Leftarrow 3$
    while $x > 1$ **do**
      for $j = 0$ to n **do**
        $x \Leftarrow x + 1$
        $y \Leftarrow y + x$
        $z \Leftarrow z + i$
      end for
    end while
  end if
end for

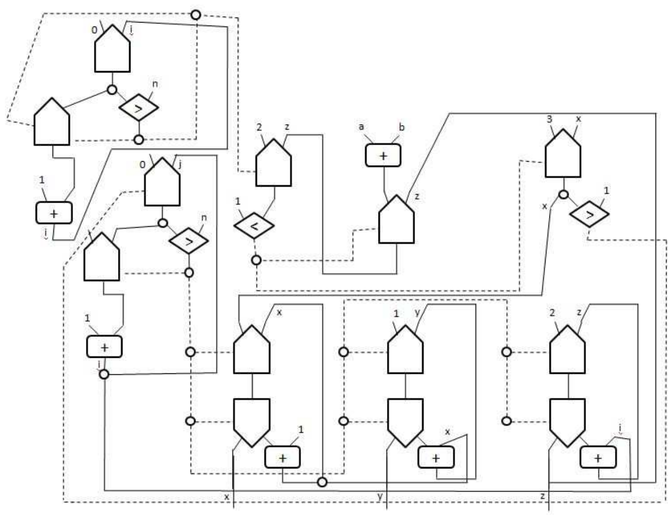
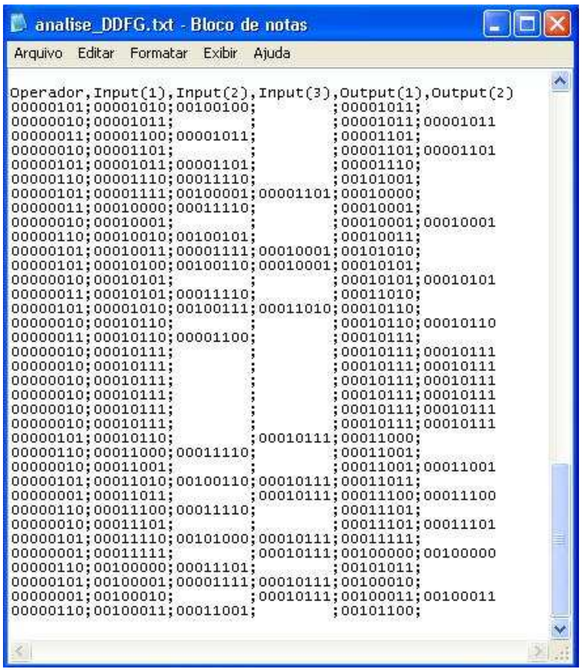


Figure 11: The Dataflow Graph for an *IF, FOR, WHILE* command.



```
analise_DDFG.txt - Bloco de notas
Arquivo Editar Formatar Exibir Ajuda

Operador,Input(1),Input(2),Input(3),Output(1),Output(2)
00000101;00001010;00100100;        ;00001011;
00000010;00001011;        ;        ;00001011;00001011
00000011;00001100;00001011;        ;00001101;
00000010;00001101;        ;        ;00001101;00001101
00000101;00001011;00001101;        ;00001110;
00000110;00001110;00011110;        ;00101001;
00000101;00001111;00100001;00001101;00010000;
00000011;00010000;00011110;        ;00010001;
00000010;00010001;        ;        ;00010001;00010001
00000110;00010010;00100101;        ;00010011;
00000101;00010011;00001111;00010001;00101010;
00000101;00010100;00100110;00010001;00010101;
00000010;00010101;        ;        ;00010101;00010101
00000011;00010101;00011110;        ;00011010;
00000101;00001010;00100111;00011010;00010110;
00000010;00010110;        ;        ;00010110;00010110
00000011;00010110;00001100;        ;00010111;
00000010;00010111;        ;        ;00010111;00010111
00000010;00010111;        ;        ;00010111;00010111
00000010;00010111;        ;        ;00010111;00010111
00000010;00010111;        ;        ;00010111;00010111
00000010;00010111;        ;        ;00010111;00010111
00000010;00010111;        ;        ;00010111;00010111
00000101;00010110;        ;00010111;00011000;
00000110;00011000;00011110;        ;00011001;
00000010;00011001;        ;        ;00011001;00011001
00000101;00011010;00100110;00010111;00011011;
00000001;00011011;        ;00010111;00011100;00011100
00000110;00011100;00011110;        ;00011101;
00000010;00011101;        ;        ;00011101;00011101
00000101;00011110;00101000;00010111;00011111;
00000001;00011111;        ;00010111;00100000;00100000
00000110;00100000;00011101;        ;00101011;
00000101;00100001;00001111;00010111;00100010;
00000001;00100010;        ;00010111;00100011;00100011
00000110;00100011;00011001;        ;00101100;
```

Figure 12: The Binary Mapping File for *IF, FOR, WHILE* command.

Listing 1: The *IF* Command in VHDL

```
--------------------------------------------------------------------------------
-- Company: Projeto ChipCflow
-- Engineer: kelton Augusto Pontara da Costa
--
-- Create Date:    17:41:27 03/26/2009
-- Design Name:
-- Module Name:    Circuito_IF - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

---- uncomment the following library declaration
     if instantiating
---- any xilinx primitives in this code
--library UNISIM
--use UNISIM.VComponents.all

ENTITY circuito_final IS
port( a : in std_logic_vector(31 downto 0);
      b : out std_logic_vector(31 downto 0);
END;

ARCHITECTURE behavior OF circuito_final IS

Component decider
port(
a: in std_logic_vector(31 downto 0);
b: in std_logic_vector(31 downto 0);
z: out std_logic_vector(31 downto 0)
);
END Component;
Component operator
port(
a: in std_logic_vector(31 downto 0);
b: in std_logic_vector(31 downto 0);
z: out std_logic_vector(31 downto 0)
);
END Component;

Component merge_deterministic
port(
a: in std_logic_vector(31 downto 0);
b: in std_logic_vector(31 downto 0);
c: in std_logic_vector(31 downto 0);
z: out std_logic_vector(31 downto 0)
);
END Component;

signal i10, i14, i11, i15, i12, i16, i18, i13,
i17, i19 : std_logic_vector(31 downto 0);

BEGIN
u1: decider port map (i10,i14,i18);
u2: operator port map (i11,i15,i13);
u3: operator port map (i12,i16,i17);
u4: merge_deterministic port map (i13,i17,i18,i19);
END behavior;
```

Listing 2: The *WHILE* Command in VHDL

```
--------------------------------------------------------------------------------
-- Company: Projeto ChipCflow
-- Engineer: kelton Augusto Pontara da Costa
--
-- Create Date:    12:45:28 04/02/2009
-- Design Name:
-- Module Name:    VHDL_WHILE - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
```

```
use ieee.std_logic_signed.all;

---- uncomment the following library declaration
      if instantiating
---- any xilinx primitives in this code
--library UNISIM
--use UNISIM.VComponents.all

ENTITY circuito_final IS
port( a : in std_logic_vector(31 downto 0);
      b : out std_logic_vector(31 downto 0);
END;

ARCHITECTURE behavior OF circuito_final IS

Component merge_deterministic
port(
a: in std_logic_vector(31 downto 0);
b: in std_logic_vector(31 downto 0);
z: out std_logic_vector(31 downto 0)
);
END Component;

Component copy
port(
a: in std_logic_vector(31 downto 0);
z: out std_logic_vector(31 downto 0);
y: out std_logic_vector(31 downto 0)
);
END Component;

Component decider
port(
a: in std_logic_vector(31 downto 0);
b: in std_logic_vector(31 downto 0);
z: out std_logic_vector(31 downto 0)
);
END Component;
Component operator
port(
a: in std_logic_vector(31 downto 0);
b: in std_logic_vector(31 downto 0);
z: out std_logic_vector(31 downto 0)
);
END Component;

signal i10, i15, i16, i17, i11, i12, i13,
i14, i18, i19, i111, i112, i121, i122,
i141, i142 : std_logic_vector(31 downto 0);

BEGIN
u1: merge_deterministic port map (i10,i15,i11);
u2: copy port map (i11,i111,i112);
u3: decider port map (i111,i16,i12);
u4: copy port map (i12,i121,i122);
u5: merge_deterministic port map (i112,i121,i13);
u6: operator port map (i13,i16,i14);
u7: copy port map (i14,i141,i142);
u8: operator port map (i141,i17,i18);
u9: operator port map (i142,i18,i19);
END behavior;
```

Listing 3: The *IF, FOR, WHILE* Command in VHDL

```
--------------------------------------------------------------------------------
-- Company:  Projeto ChipCflow
-- Engineer:  kelton Augusto Pontara da Costa
--
-- Create Date:    08:43:20 03/27/2009
-- Design Name:
-- Module Name:    Circuito_completo - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

---- uncomment the following library declaration
      if instantiating
---- any xilinx primitives in this code
--library UNISIM
--use UNISIM.VComponents.all

ENTITY circuito_final IS
port( a : in std_logic_vector(31 downto 0);
      b : out std_logic_vector(31 downto 0);
END;

ARCHITECTURE behavior OF circuito_final IS

Component merge_deterministic
port(
a: in std_logic_vector(31 downto 0);
b: in std_logic_vector(31 downto 0);
c: out std_logic_vector(31 downto 0);
z: out std_logic_vector(31 downto 0)
);
END Component;

Component copy
port(
a: in std_logic_vector(31 downto 0);
z: out std_logic_vector(31 downto 0);
y: out std_logic_vector(31 downto 0)
);
END Component;

Component decider
port(
a: in std_logic_vector(31 downto 0);
b: in std_logic_vector(31 downto 0);
z: out std_logic_vector(31 downto 0)
);
END Component;

Component operator
port(a: in std_logic_vector(31 downto 0);
b: in std_logic_vector(31 downto 0);
z: out std_logic_vector(31 downto 0)
);
END Component;

Component branch
port(
a: in std_logic_vector(31 downto 0);
c: in std_logic_vector(31 downto 0);
z: out std_logic_vector(31 downto 0);
y: out std_logic_vector(31 downto 0)
);
END Component;

signal i10, i36, i12, i30, i15, i33, i18, i37, i20,
i38, i39, i40, i11, i13, i14, i41, i16, i17, i19, i42,
i21, i26, i22, i23, i24, i25, i27, i28, i29, i31, i32,
i43, i34, i35, i44, i111, i112, i131, i132, i171, i172,
i211, i212, i221, i222, i231, i232, i231, i232,
i231, i232, i231, i232, i231, i232, i231, i232, i251,
i252, i291, i292 : std_logic_vector(31 downto 0);

BEGIN
u1: merge_deterministic port map (i10,i36,i11);
u2: copy port map (i11,i111,i112);
u3: decider port map (i12,i111,i13);
u4: copy port map (i13,i131,i132);
u5: merge_deterministic port map (i112,i131,i14);
u6: operator port map (i14,i30,i41);
u7: merge_deterministic port map (i15,i33,i132,i16);
u8: decider port map (i16,i30,i17);
u9: copy port map (i17,i171,i172);
u10: operator port map (i18,i37,i19);
u11: merge_deterministic port map (i19,i15,i171,i42);
u12: merge_deterministic port map (i20,i38,i172,i21);
u13: copy port map (i21,i211,i212);
u14: decider port map (i211,i30,i26);
u15: merge_deterministic port map (i10,i39,i26,i22);
u16: copy port map (i22,i221,i222);
u17: decider port map (i221,i12,i23);
u18: copy port map (i23,i231,i232);
u19: copy port map (i23,i231,i232);
u20: copy port map (i23,i231,i232);
u21: copy port map (i23,i231,i232);
u22: copy port map (i23,i231,i232);
u23: copy port map (i23,i231,i232);
u24: merge_deterministic port map (i222,i231,i24);
u25: operator port map (i24,i30,i25);
u26: copy port map (i25,i251,i252);
u27: merge_deterministic port map (i26,i38,i232,i27);
u28: branch port map (i27,i233,i28,i28);
u29: operator port map (i28,i30,i29);
u30: copy port map (i29,i291,i292);
u31: merge_deterministic port map (i30,i40,i234,i31);
u32: branch port map (i31,i235,i32,i32);
u33: operator port map (i32,i291,i43);
u34: merge_deterministic port map (i33,i15,i236,i34);
u35: branch port map (i34,i237,i35,i35);
u36: operator port map (i35,i251,i44);
END behavior;
```

4 The Implementation of a dataflow graph

An operator is a complex element in ChipCflow which consists of various different parts: the intercommunication system; the matching data; the instances generator; and the control to execute the instances. The schematic of the operators is described in Figure 13.

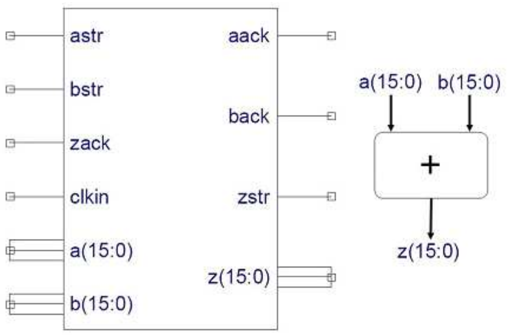


Figure 13: The Schematic of one Operator.

As can be clearly seen in Figure 13, the operator has six input signals *astr, bstr, clkin, a(15:0), b(15:0), and zack*. The *astr* and *bstr* are used to indicate the current operator that it has an item of data coming through the input *a(15:0)* or *b(15:0)* respectively. The *clkin* is an internal clock for the operator. Although the operator has an internal clock, the communication system between the operator characterizes the asynchronism for the dataflow system.

The operator has four output signals *aack, back, zstr, z(15:0)*. The *aack* and *back* are used to indicate the previous operators that the current operator has received an item of data through its input *a(15:0)* or *b(15:0)* respectively.

The *zstr* is used to indicate the next operator that the current operator has an item of data going through an input in the next operator.

The difference between the operators are in the function that the operator has to execute and the number of input and output signals that can be different for each operator. In Figure 14 an ASM Chart of the operator is described.

As can be clearly seen in Figure 14, the protocol of the operator for all input signals are verified and if all items of data are present in the operator, it is executed and the protocol concludes its execution with an acknowledge signal for each input signal. The strobe and result signal also are sent by the operator.

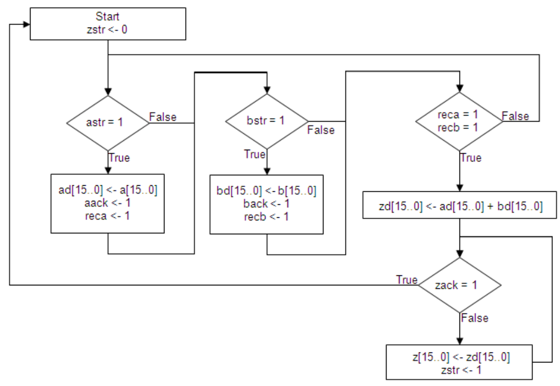


Figure 14: The ASM chart of the Operator.

The *IF* command is a simple interconnection of various operators and in Figure 15, a simulation of its operation is presented. In Table 1 a set of data used in the simulation is described, where the value for the items of input data are 2(a), 3(b), 4(c), 1(d), 2(x); the *z* is 5; and 3(a), 4(b), 9(c), 9(d), 0(x); the *z* is 0.

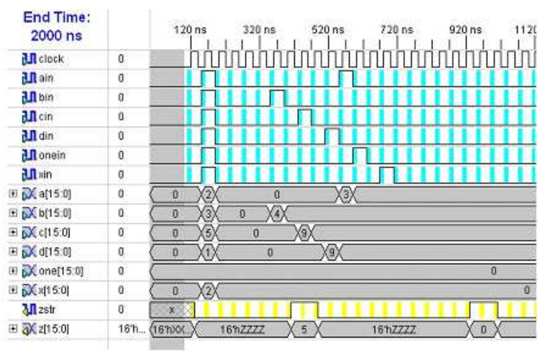


Figure 15: The Simulation Result for the IF Command.

Table 1: The Result of If Command

				Inputs	Output
a	b	c	d	x	z
2	3	4	1	2	5
3	4	9	9	0	0

The *WHILE* command is also a simple interconnection of various operators and in Figure 16 a simulation of its operation is presented. In Table 2, a set of data used in the simulation is described, where the

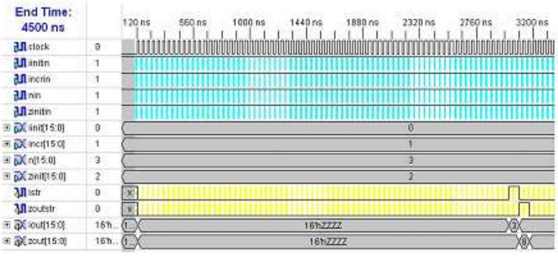


Figure 16: The Simulation Result for the WHILE Command.

value for the items of input data are 0(a), 1(b), 2(iinit), 1(incr), 3(n); the *z* is 8.

Table 2: The Result Data of WHILE command

		Inputs			Output
a	b	iinit	incr	n	while
0	1	2	1	3	8

The Fibonacci sequence is a more complex construction than a simple *IF command* or *WHILE command*. It is an iterative construction where each element consist of adding two previous elements, except for the two first elements that are respectively 0 and 1. In Figure 17 the dataflow graph for the Fibonacci

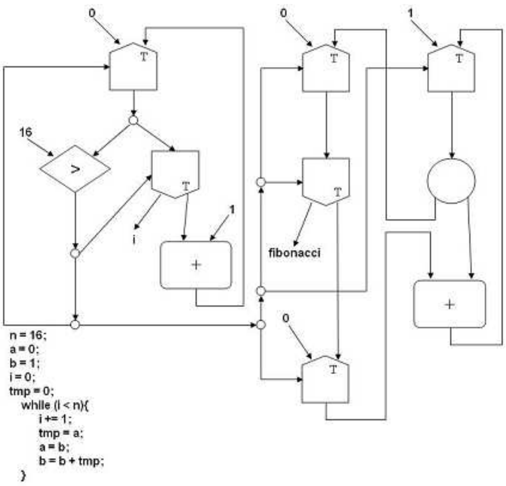


Figure 17: The Fibonacci Sequence.

sequence is described and in Figure 18 a simulation of its operation is presented.

As can be clearly seen in Figure 18, the result of the simulation for the Fibonacci sequence is 987, that corresponds to the 16th number of the sequence.

In Table 3 a set of data used in the simulation is described, where the value for the two first sets of data are 0(a) and 1(b); the *n* is 16; the value for *i* is 0 *iinit* and the increment value is 1 *incr*.

Table 3: The Result Data of Fibonacci sequence

		Inputs			Output
a	b	iinit	incr	n	fibonacci
0	1	0	1	16	987

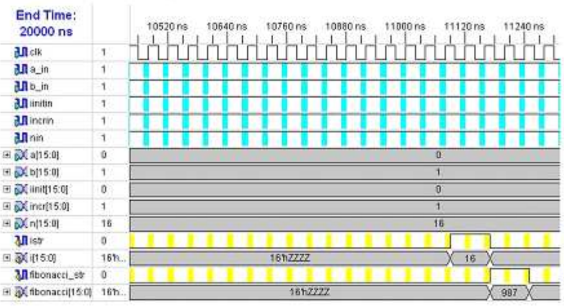


Figure 18: The simulation result for the Fibonacci sequence.

In Figure 19 the execution result for the Fibonacci sequence that was executed in C in a desktop computer is described and it proves the correctness of the Fibonacci sequence which was executed in the dataflow graph.

Figure 19: The Fibonacci sequence result in a desktop computer.

In Table 4, the delay and utilization summary for the Fibonacci sequence implemented in a platform Xilinx II Pro are presented.

Table 4: The Virtex II Pro - Delay and Utilization summary

C	Device	Clock	RAM	Minimum Period
IF	xc2vp2	224MHz	17%	4.451 ns
While	xc2vp2	224MHz	34%	4.451 ns
DoWhile	xc2vp2	224MHz	34%	4.451 ns
For	xc2vp2	224MHz	34%	4.451 ns
Fibonacci	xc2vp2	224MHz	41%	4.451 ns

5 Conclusion

Research to convert High Level Language (HLL) into hardware has put forward various possibilities mainly with the flexibility and capacity of the reconfigurable architectures. A Control Dataflow Graph (CDFG) is a fundamental element in this process. Otherwise, a Dataflow Architecture, which was the focus in the 1980s, can be obtained directly from the CDFG. In particular, dynamic dataflow architecture can be generated in order to produce a high level of parallelism. In this paper, the ChipCflow project was described as a system to convert HLL into a dynamic dataflow graph to be executed in dynamic reconfigurable hardware, exploring the dynamic reconfiguration. The operator, which is the main element in the dataflow graph, was implemented, and spent *4ns* to execute all the process. Examples of C program were converted into a VHDL language and some of them were implemented and simulated using the basic operator that was initially implemented and simulated. The next steps of the ChipCflow project are to implement the complete model of instances and generate an analysis with benchmarks to verify the impact of this approach.

References:

[1] Ali, F. M. and Das, A. S.Azme, Hardware-software co-synthesis of hard real-time systems with reconfigurable FPGAs, *ELSEVIER - Computer and Electrical Engineering*(2004), volume=30,pg 471-489

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

[3] Arnold, J. and D. Buell and E. Davis, SPLASH 2, *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures* (1992), 316-324

[4] Cardoso J. M. P, Compilação de Algoritmos em JAVA para Sistemas Computacionais Reconfiguráveis com Exploração de Paralelismo ao Nível das Operações, *PhD thesis, Universidade Técnica de Lisboa* (2000)

[5] Arvind, Dataflow: Passing the token, *ISCA Keynote* (2005)

[6] Celoxica Limited, DK Design Suite Datasheet, *see http://www.celoxica.com* (2007)

[7] Capelli, A, A Dataflow Control Unit for C-to-Configurable Pipelines Compilation Flow, *IEEE Sumposium on Field-Programmable Custom Computing Machines FCCM'04* (2004)

[8] Impulse Inc, *see http://www.impulsec.com* (2005)

[9] Joelmir Jose Lopes and Jorge Luiz e Silva, A Benchmark Approach for Compilers in Reconfigurable Hardware, *The 6th International Workshop System-on-Chip for Real-Time Applications IWSOC'06* (2006).

[10] Dennis, J. B., A preliminary architecture for a basic dataflow processor, *Proceedings of the 2nd Annual Symposium on Computer Architecture*, (1975)

[11] Silva, J.L., Executing Algorithms for Dynamic Dataflow Reconfigurable Hardware - A Purpose for Matching Data *The 6th IEEE International Workshop System-on-Chip for Real-Time Applications - IWSOC'06*,(2006)

[12] Pellerin, D. and Thibault, S, Practical FPGA Programming in C, *Prentice Hall PTR* (2005)

[13] Swanson, S., Wavescalar, *36th Annual International Symposium on Microarchitecture* (2003)

[14] Veen, A. H., Dataflow Machine Architecture, *ACM Computing Surveys*, n.4, (1986), v.18, pp 365-396

[15] Astolfi, V. F. A., Silva, J. L., Execution of algorithms using a Dynamic Dataflow Model for Reconfigurable Hardware - Commands in Dataflow Graph., *The 3th IEEE Southern Conference on Programmable Logic - SPL2007*, Mar del Plata, Argentina. pp 225-230. (2007)

[16] Silva, J. L., Correia, V. M., C commands Implemented direct into the hardware using the ChipCflow Machine. *The 9th IEEE International Reconfigurable Computing and Applications Conference - JCRA2009*, Alcalá, Espanha, (2009).

[17] Davis D., Beeravolu S and Jaganathan, R, Hardware/Software Codesign for Platform FPGAs - Xilinx *Xilinx*, (2005)