ADVANCES IN CYBER-PHYSICAL SYSTEMS

Vol. 1, No. 2, 2016

PROGRAM CODE PARALLELIZATION METHOD

Liubomyr Tsyhylyk

Lviv Polytechnic National University, 12, Bandera Str., Lviv, 79013, Ukraine.
Authors e-mail: l.tsyhylyk@gmail.com

Submitted on 04.12.2016

Abstract. **Method of parallelism extraction from sequential program is proposed. The definition of three-address code is given. The requirements to the sequential and parallel program are determined. The structure and design performance of the parallel program are given. The description of two stages of the parallelism extraction method is proposed: stage of preliminary field initialization and recursive stage of the parallel extraction. Evaluate efficient of the parallelism extraction method based on an example of FFT 64p.**

Index Terms **— parallelization, program parallelization, method of program parallelization, parallel code structure, tree-address program**

I. INTRODUCTION

More than a dozen companies are engaged in developing automated systems for design special devices and processors [1]. To achieve flexibility in the design, the minimum of effort for the design and performance of such devices, the company developed a technology C to HDL [1] [2] [3] [4]. The technology used for those tasks where time for execution on universal processors architecture is unacceptably high. These tasks include processing biological data, modeling of physical processes, financial forecasts and others. Embedded systems that require high-performance computing or data processing in real time also use this technology.

C to HDL compilers should be used in large projects or tasks which might need to change the algorithm in future. Development of large projects solely on the HDL code is a complex task that requires a lot of time to design a system. Abstraction level of C code for such projects significantly reduces design time, over-order design written in HDL code, modify heavier compared to the C language. If the engineer needs to add new functionality to the existing system, he/she needs to add/modify a few lines of C language code. For instance, to perform the same task in HDL needs to modify/add functional modules of the system, which require more time to develop.

A high-level synthesis system of specialized devices "Chameleon" [4] [6] was developed using technology C to HDL (hereinafter C2HDL). The aim of this system is the generation of HDL code that performs the algorithm presented in C with predetermined performance. The IP core synthesis process consists of several stages [7]. An intermediate stage is the generation of three-address code program, which aims at submitting flow graph algorithm for further processing.

For the maximum performance of any computer system, it is necessary to use the parallelism and the pipe [8] [9]. These approaches will significantly reduce the computation time. Given that the main purpose of the system configuration is C2HDL productivity performance computing, it is recommended to use instructions and data pipelining to goal achievement.

II. TASK

To develop a parallel three-address application code structure, create data pipelining stages and develop a consistent method of three-address code parallelization.

III. SEQUENSIAL PROGRAM CODE STRUCTURE

One of the main characteristics of the system design is reflected in C2HDL flow graph algorithm [10] from a sequensial program code (three-address code). It is a simple code structure to extract maximum of parallelization and tracking dependencies between the commands.

We assume that three-address source program is a sequence of commands describing algorithm where one line is represented by only one command that should be performed on two operands of data and the result of this operation is assigned an another operand for example (1).

$$add \quad R3 \quad R2 \quad R1 \tag{1}$$

where add – mnemonic mark of adding command, R1 and R2 – operands of data for computation execution, R3 – operand that stores result of command execution.

According to the main problems, facing the system C2HDL, three-address code must meet the following requirements:

- Contains only the algorithm description, reflects the algorithm's flow graph [10]. There is no index of the array calculations.
- Loops, used in the C language, is fully unwound and presented in a sequential way.
- Three-address code consists of a set of operands (registers, R1, R2, ...), their number is unlimited, as three-address code is an intermediate representation of the algorithm and is not based on a predefined computer architecture. One command may use the same name operands:

$$add \quad R1 \quad R1 \quad R1 \tag{2}$$

Table 1

List of the three-address commands

#	Category	Number of condition command	Number of condition command for which current command dependent	Condition of current command execution	Mnemonic mark	Operand of the result	Operand of the argument #1	Operand of the argument #2	Port number	Command description
1	2	3	4	5	6	7	8	9	10	11
1	Loading of input data	0	y	z	in	R1	–	–	*i*	Input data loading from the external port number #*i* (port number can be any natural number). Here "y" – is a number of conditional command on which the current command depends, "z" – the execution condition of the current command. The In-command will be executed when the result of the conditional command "y" is equal to "z".
2		0	y	z	ld	R1	*k*	–	–	Constant loading from internal memory. Where "*k*" is integer constant. Also, it can be written in float point format. Example: 234.76543 or -0.000345
3	Giving the results	0	y	z	out	R1	–	–	*i*	Giving output result using external port #*i*
4	Arithmetical commands	0	y	z	add	R3	R1	R2	–	Adding command execution between R1 and R2, the test result is written in R3
5		0	y	z	sub	R3	R1	R2	–	Subtracting command execution between R1 and R2, the test result is written in R3
6		0	y	z	mul	R3	R1	R2	–	Multiplication command execution between R1 and R2, the test result is written in R3
7		0	y	z	div	R3	R1	R2	–	Division command execution between R1 and R2, the test result is written in R3
8		0	y	z	adds	R3	R1	R2	–	Adding command execution between R1 and R2, one bit shift right performed, the test result is written in R3
9		0	y	z	subs	R3	R1	R2	–	Subtracting command execution between R1 and R2, one bit shift right performed, the test result is written in R3
10	Logical commands	0	y	z	sll	R3	R1	R2	–	Shift logic left R1 for R2 bits, the test result is written in R3
11		0	y	z	sal	R3	R1	R2	–	Shift arithmetic left R1 for R2 bits, the test result is written in R3
12		0	y	z	slr	R3	R1	R2	–	Shift logic right R1 for R2 bits, the test result is written in R3
13		0	y	z	sar	R3	R1	R2	–	Shift arithmetic right R1 for R2 bits, the test result is written in R3
14		0	y	z	and	R3	R1	R2	–	Logic 'and' execution between R1 and R2, the test result is written in R3
15		0	y	z	or	R3	R1	R2	–	Logic 'or' execution between R1 and R2, the test result is written in R3
16		0	y	z	xor	R3	R1	R2	–	Logic 'xor' execution between R1 and R2, the test result is written in R3
17		0	y	z	not	R2	R1	–	–	Logic 'not' execution with R1, the test result is written in R2
18	Condition commands	x	y	z	cmpeq	–	R1	R2	–	'Equal' operation execution between R1 and R2. Where "x" – sequential number of conditional command. The test result is automatically stored on HW predefined register.
19		x	y	z	cmpneq	–	R1	R2	–	'Not equal' operation execution between R1 and R2. The test result is automatically stored on HW predefined register.
20		x	y	z	cmpleq	–	R1	R2	–	'Less or equal' operation execution between R1 and R2. If R1 is less or equal to R2 that the result is 'true'. The test result is automatically stored on HW predefined register.
21		x	y	z	cmpl	–	R1	R2	–	'Less' operation execution between R1 and R2. If R1 is less than R2 that the result is 'true'. The test result is automatically stored on HW predefined register.

Table i continuation

1	2	3	4	5	6	7	8	9	10	11
22		x	y	z	cmpgreq	–	R1	R2	–	‘Greater or equal’ operation execution between R1 and R2. If R1 is greater or equal to R2 that the result is ‘true’. The test result is automatically stored on HW predefined register.
23		x	y	z	cmpgr	–	R1	R2	–	‘Greater’ operation execution between R1 and R2. If R1 is greater than R2 that the result is ‘true’. The test result is automatically stored on HW predefined register.
24	Jump	0	y	z	jmp	@R1	–	–	–	‘Jump’ operation. Command pointer jumps to the address stored in @R1.
25	Assign	0	y	z	asgn	R1	R2	–	–	Assign operation. The value of the register R2 is assign to the R1 register.

• Commands of loading input data and giving output results should be composed in the following way:

$$com \quad R1 \quad n \qquad (3)$$

where, com – command of loading/giving data (in/out), R1 – data register, n – port number ($n \in Z$).

• Conditional command supporting [10].

• Contains the following set of commands (Table 1).

Mathematical model of three-address code shows in next way:

$$P = \left\{ \left(\begin{array}{l} \langle x|0\rangle\langle y|0\rangle\langle z\rangle\langle comand\rangle\langle reg_{dest}\rangle \\ \langle reg_{\arg 1}|const|\otimes\rangle\langle reg_{\arg 2}|\otimes\rangle\langle port|\otimes\rangle \end{array} \right)_i \right\} \qquad (4)$$

where $x\in N$ – conditional command number; $y\in N$ – conditional command number that instant command dependents; $z\{0 \mid 1\}$ – condition of the instant command execution; command $\in$ COM – mnemonic mark of a command, COM – set of the commands; $reg_{dest}\in$ REG – the result register, REG – set of registers; $reg_{arg1}\in$ REG – the first argument register; $const\in R$ – constant (fix and float (IEEE 754) point supported); $reg_{arg2}\in$ REG – the second argument register; $port\in N$ – port number; $i\in N$ – sequential number of the command line.

This mathematical model *P* describes any combination of the three-address program.

IV. PARALLEL CODE REQUIREMENTS

C2HDL system uses configurable architecture of special processor [7] for high performance IP core generation. The main advantage of this architecture to compare with existing [8] [11] [12] [13] [14] [15] [16] is its scalability and “adaptation” to the input algorithm. Under “adaptation” we should understand configuration possibility of all units in that way to set desired system performance with minimal HW resource usage. The next parallel code requirements were elaborated during analyzing particularity of the configurable architecture:

• The parallelization level of the sequential code should depend definitely on the input algorithm characteristics.

• Pipeline usage between stages: input data loading, output data extracting and perform calculations.

V. PARALLEL CODE STRUCTURE AND CHARACTERISTIC

The next list refers to the main parallel code characteristic:

• Number of the parallel code branches for arithmetic and logic operations execution.

• Command-loading level of each branch (the value is determinates in percentage). This value has higher priority than number of the parallel code branches set by user. Thereafter if command-loading level is not reach at least for one branch, the parallel branch number decrease by one and repeat this procedure again.

• The input/output ports number.

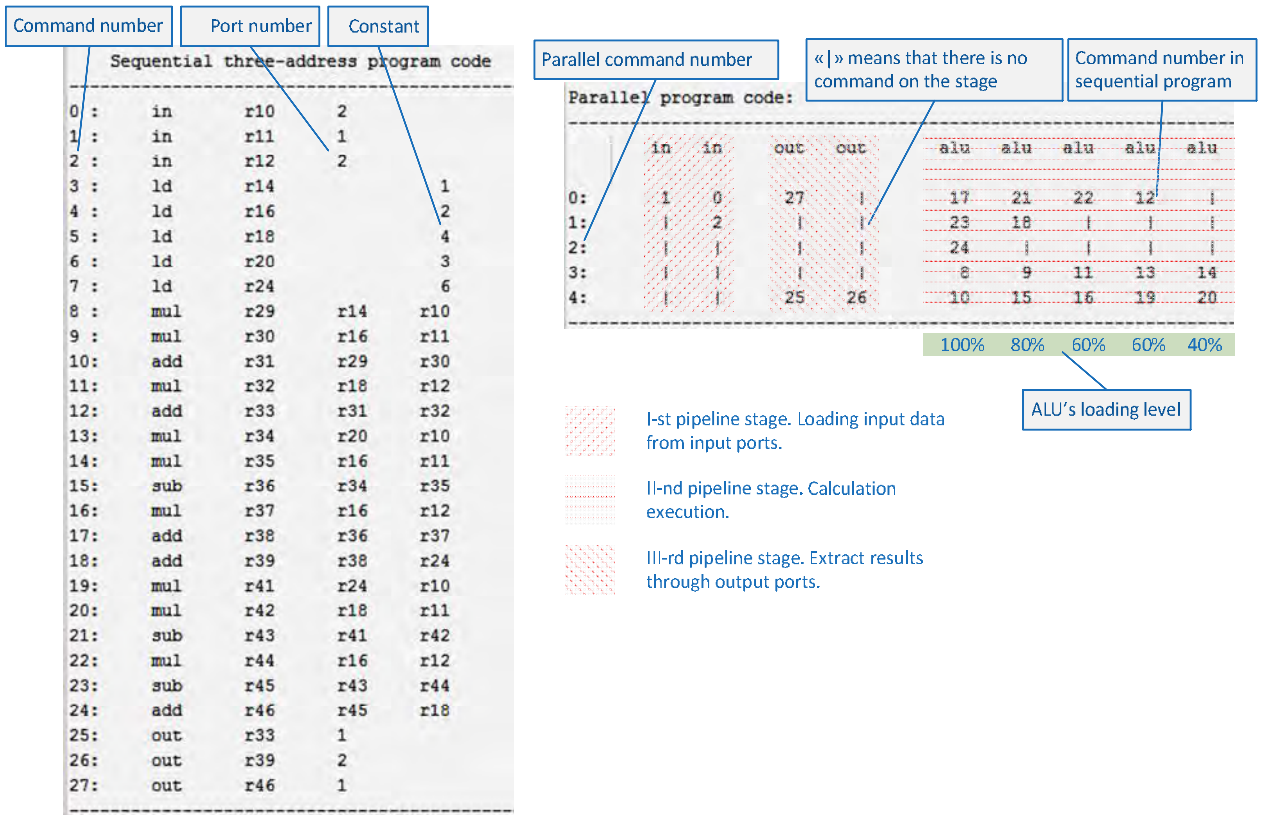
• Stages count (parallel lines) of the program.

Performance of any designed system is determined by the combination of these entire characteristic (Fig. 1), but the dominant role belongs to count of the parallel code branches. This value shows how many parallel arithmetical units (ALU) are used for program execution. The higher this number is, the higher isprogram parallelization and, accordingly, higher system performance can be achieved. Taking into account that each algorithm has its own parallel level, thereafter the code-loading level of each parallel branch can be different. One branch of the parallel code is additional ALU. The command-loading level is ratio of the commands that should be executed in this ALU, to the total amount of the parallel lines (stages) of code. Set the requirements to the command-loading level for each ALU we determine performance of the desired system.

Additional characteristic of system performance is pipeline usage. Based on three stages for computation execution [14]: loading input data, computation execution and output data extraction, thereafter these stages can be implemented using pipeline. The first stage is loading input data, three-address command is in (e.x. loading *i* “portion” of data). The second stage is computation execution (e.x. computation of the *i* “portion” of data and loading *i-1* “portion”). The third and the last one stage is extract output data, three-address command is out (e.x. extract output results of the *i* “portion” of data, computation *i-1* “portion” of data and loading *i-2* “portion”).

The advantage of using such pipeline is reducing number of parallel lines of code. All commands of loading and extracting data and also arithmetic and logical commands are executed in parallel way but with different "portions" of data.

C2HDL system executes synthesis all the HW modules that are necessary for computation execution. Accordingly, there is no needed to use commands to load constants into the RAM provisionally read this data from the ROM but expedient directly save this data in the RAM originally have initialized it during HDL code synthesis. This allows saving the ROM resource usage assigned to store parallel program that are required by the HDL system. Fig. 1 shows example of three-address RGB→YUV program (coefficients are random). Here are shown parallel program characteristics and determined pipeline stages of data computation. The parallel branches number (ALU = 5) was chosen based on the efficiency level. Conditional command fields are absent for convenience.


Command number
Port number
Constant
Parallel command number
«|» means that there is no command on the stage
Command number in sequential program
100% 80% 60% 60% 40%
ALU's loading level
I-st pipeline stage. Loading input data from input ports.
II-nd pipeline stage. Calculation execution.
III-rd pipeline stage. Extract results through output ports.

```
   Sequential three-address program code
---------------------------------------------
0 :    in     r10     2
1 :    in     r11     1
2 :    in     r12     2
3 :    ld     r14             1
4 :    ld     r16             2
5 :    ld     r18             4
6 :    ld     r20             3
7 :    ld     r24             6
8 :    mul    r29     r14     r10
9 :    mul    r30     r16     r11
10:    add    r31     r29     r30
11:    mul    r32     r18     r12
12:    add    r33     r31     r32
13:    mul    r34     r20     r10
14:    mul    r35     r16     r11
15:    sub    r36     r34     r35
16:    mul    r37     r16     r12
17:    add    r38     r36     r37
18:    add    r39     r38     r24
19:    mul    r41     r24     r10
20:    mul    r42     r18     r11
21:    sub    r43     r41     r42
22:    mul    r44     r16     r12
23:    sub    r45     r43     r44
24:    add    r46     r45     r18
25:    out    r33     1
26:    out    r39     2
27:    out    r46     1
---------------------------------------------
```

```
Parallel program code:
-----------------------------------------------------------------
      in   in     out   out     alu   alu   alu   alu   alu

0:     1    0      27     |      17    21    22    12     |
1:     |    2       |     |      23    18     |     |     |
2:     |    |       |     |      24     |     |     |     |
3:     |    |       |     |       8     9    11    13    14
4:     |    |      25    26      10    15    16    19    20
-----------------------------------------------------------------
```

Fig. 1. Three-address program code in parallel and sequential

The parallel code structure can be shown in next mathematical model (5).

$$PL = \left\{ \left((IN)_x (OUT)_y (ALU)_z \right)_j \right\} \quad (5)$$

where j∈ N – ordinal number of a stage, x∈ N – ordinal number of input port, y∈ N – ordinal number of output port, z∈ N – ordinal number of ALU.

$$IN = \left\{ \langle 0\rangle \langle y|0\rangle \langle z\rangle \left\langle \left\langle \langle in\rangle \langle reg_{dest}\rangle \right\rangle \middle| \otimes \right\rangle \right\} \quad (6)$$

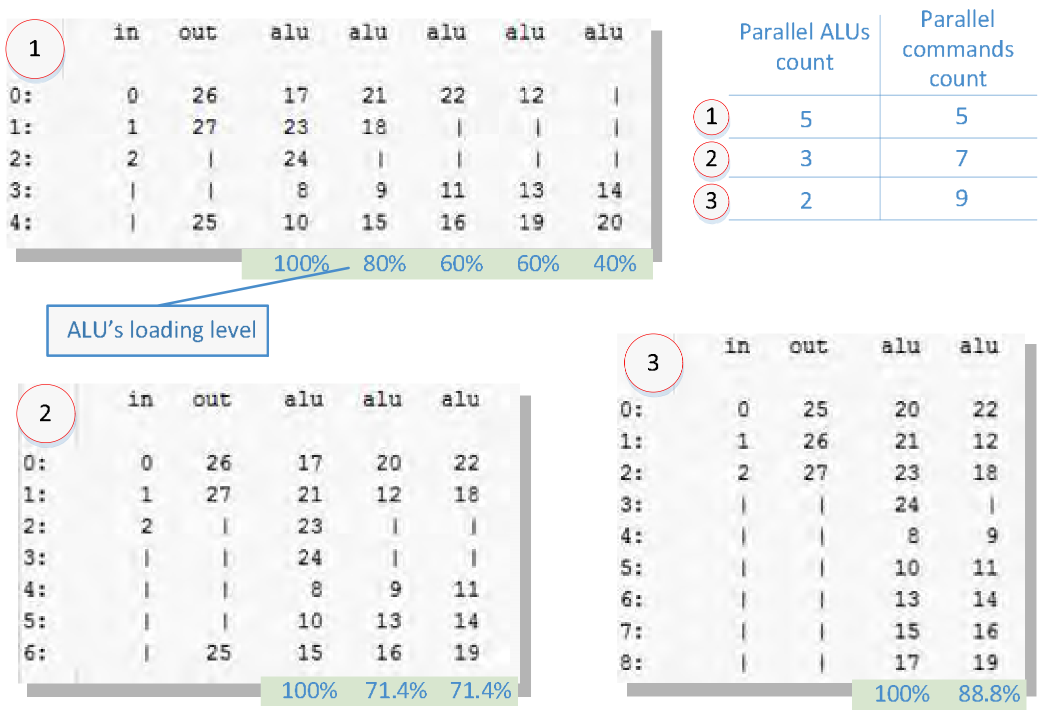
$$OUT = \left\{ \langle 0\rangle \langle y|0\rangle \langle z\rangle \left\langle \left\langle \langle out\rangle \langle reg_{dest}\rangle \right\rangle \middle| \otimes \right\rangle \right\} \quad (7)$$

$$ALU = \left\{ \begin{matrix} \langle x|0\rangle \langle y|0\rangle \langle z\rangle \\ \left\langle \left\langle \begin{matrix} \langle comand\rangle \langle reg_{dest}\rangle \langle reg_{arg1}\rangle \\ \langle reg_{arg2}|\otimes\rangle \end{matrix} \right\rangle \middle| \otimes \right\rangle \end{matrix} \right\} \quad (8)$$

Mathematical model (5) allows describing any parallel algorithm that executes as input task of C2HDL system. Input/output ports and ALUs numbers are configurable (Fig. 2) that allows select configuration to get user desired performance after analyzing. Fig. 2 shows example of previous (Fig. 1) three-address sequential code in three different parallel configurations. There is no needed to make parallel input/output ports thus the round of execution arithmetic operations, with maximum parallel branches, is longer than the round of input/output commands execution (see the first configuration on Fig. 2).

VI. SEQUENTIAL TO PARALLEL CODE TRANSFORMATION METHOD

Taking into account requirements described in section IV lets dig deeper and create additional requirements to sequential code. Take mathematical model *P* as essential and complement it with additional fields.



```
1     in   out   alu   alu   alu   alu   alu
0:     0    26    17    21    22    12     |
1:     1    27    23    18     |     |     |
2:     2     |    24     |     |     |     |
3:     |     |     8     9    11    13    14
4:     |    25    10    15    16    19    20
                100%   80%   60%   60%   40%
```

```
2     in   out   alu   alu   alu
0:     0    26    17    20    22
1:     1    27    21    12    18
2:     2     |    23     |     |
3:     |     |    24     |     |
4:     |     |     8     9    11
5:     |     |    10    13    14
6:     |    25    15    16    19
                100% 71.4% 71.4%
```

```
3     in   out   alu   alu
0:     0    25    20    22
1:     1    26    21    12
2:     2    27    23    18
3:     |     |    24     |
4:     |     |     8     9
5:     |     |    10    11
6:     |     |    13    14
7:     |     |    15    16
8:     |     |    17    19
                100% 88.8%
```

	Parallel ALUs count	Parallel commands count
1	5	5
2	3	7
3	2	9

Fig. 2. Parallel configurations of three-address program RGB→YUV

The aim of extending mathematical model is to enlarge information of sequential code with relationships between commands. Sequential code structure will be next – (9).

$$P\,\mathrm{mod} = \{P \cup I\} \qquad (9)$$

$$I = \left\{ \begin{pmatrix} executeInLine_{dest};\, source_{arg1};\\ source_{arg2};\, sourceOut_{dest} \end{pmatrix}_j \right\} \qquad (10)$$

where $j \in$ N – ordered line number of command in sequential code; $executeInLine_{dest} \in$ N – ordered stage number (parallel line number), in which, the instant command will be executed; $source_{arg1} \in$ N – ordered line number of command in sequential code, the first argument of instant command is initialized there; $source_{arg2} \in$ N – ordered line number of command in sequential code, the second argument of instant command is initialized there; $sourceOut_{dest} \in$ N – ordered line number of command in sequential code, the output argument of "*out*" command is initialized there.

Additional fields in (10) are prime numbers. It doesn't contain any arrays or data structures.

Method for program code parallelization consists of two stages:

1. Field initialization (Table 2): $source_{arg1}$, $source_{arg2}$ and $sourceOut_{dest}$. The aim of this stage is to show up all command relationships. Each command argument is analyzed to determine number of line where this argument was initialized. "*in*" and "*out*" commands placement is executed on this stage also. Each "*in*" or "*out*" command contains port number (5). These port numbers are used as vectors for in/out command execution. Arrangement of mutual placement of in/out commands is the same, for the same port names, as in sequential program code. Algorithm flow chart can be created based on the results of this stage. This information allows displaying the algorithm for visualization.

2. Recursive method of the code parallelization (Table 3). This is the main round of code parallelization. Field $executeInLine_{dest}$ is formed in this stage, this field is equal to j ($j \in PL$), all fields of PL (5) are determined also in this stage. This method is based on two-level recursion where the first level is command determination for initialization state analyzing its arguments, the target of the second level is command placement analyzing in parallel code that initializes input arguments of instant command. The idea of this method is to place those commands in parallel code the results of execution of which are required for farther computations.

VII. EFFICIENCY OF THE METHOD

Parallelization method efficiency can be determined according to the next criteria:

• PC time consuming required for parallelization process.

• PC memory usage required for parallelization process.

Parallel commands count.

• Parallelization level of sequential code (parallel branches count).

Personal computer with next performance characteristic used for investigation efficiency of current method:

Processor	Intel Core i7-3630QM 2.4 ГГц
RAM	DDR3 1600 МГц; 16 ГБ
Operation system	Windows 7 Professional 64-bit.

Table 2

Field initialization method

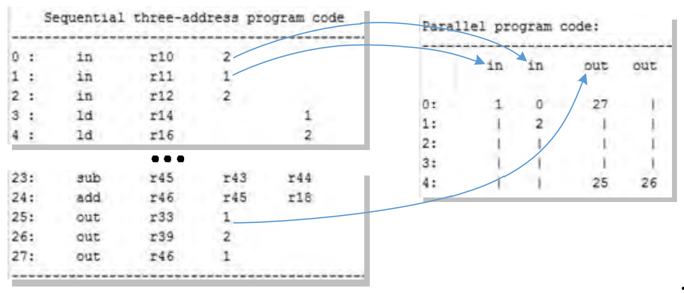
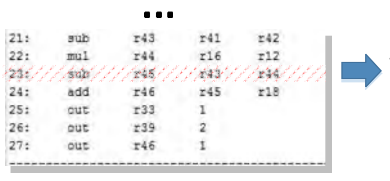
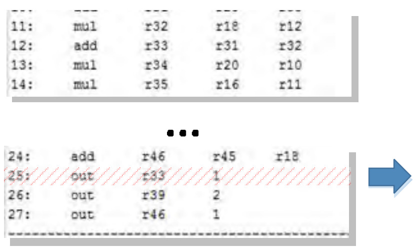
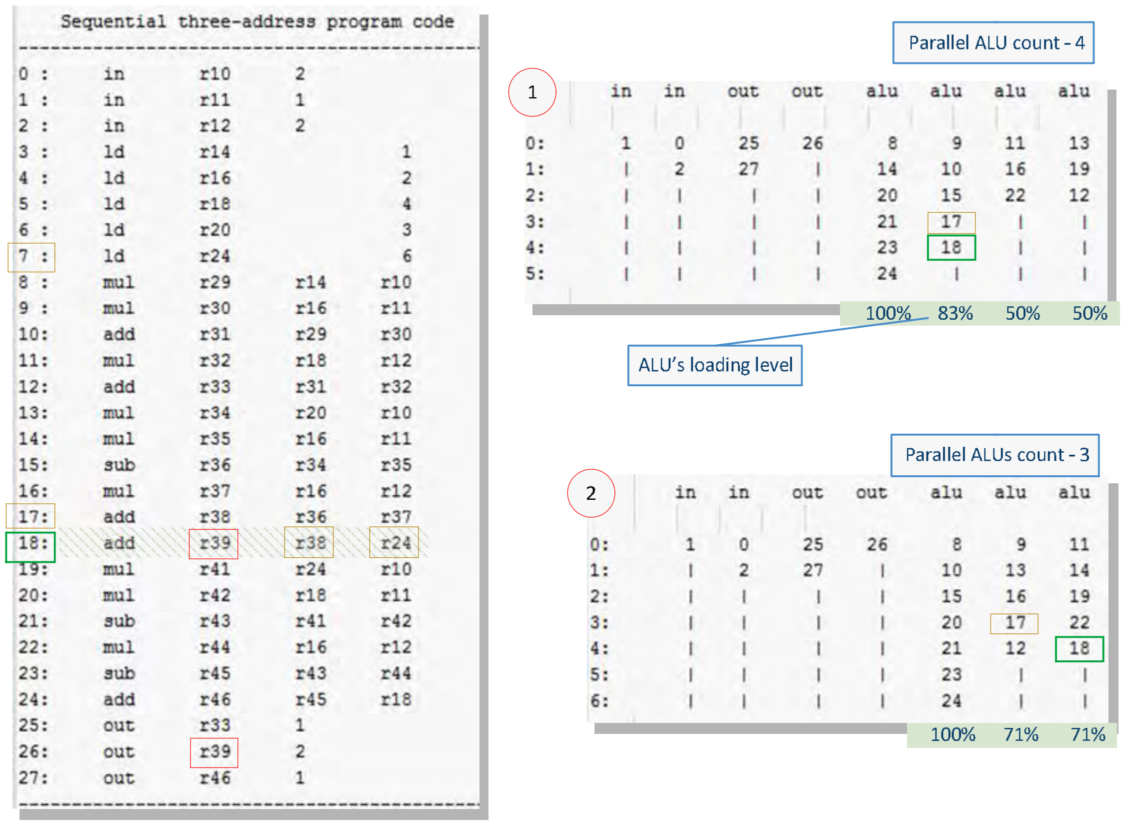
Step #	Discription	Example
1	All sequential commands analyzing. If instant command is *"in" or "out"* – this command should be assigned to correspond row ' *j* ' of parallel command and column according to the port number of this command. If there is any input/output command placed in ' *j* ' line and specific port of parallel code – thereafter this instant command should be placed in the next free parallel line.	*IN* (6) and *OUT* (7) of set *PL* (5) are determined in this stage.  Sequential three-address program code 0 : in r10 2 1 : in r11 1 2 : in r12 2 3 : ld r14 1 4 : ld r16 2 … 23: sub r45 r43 r44 24: add r46 r45 r18 25: out r33 1 26: out r39 2 27: out r46 1 Parallel program code: in in out out 0: 1 0 27 | 1: | 2 | | 2: | | | | 3: | | | | 4: | | 25 26 .
2	All sequential commands analyzing. If the instant command is not *"in" or "out"* – need to determine in which line of sequential code command arguments are initialized (*reg*$_{arg1}$ and *reg*$_{arg2}$, see (4)). The row numbers where these arguments are initialized assigned to *reg*$_{arg1}$ and *reg*$_{arg2}$ accordingly.	 … 21: sub r43 r41 r42 22: mul r44 r16 r12 23: sub r45 r43 r44 24: add r46 r45 r18 25: out r33 1 26: out r39 2 27: out r46 1 [illegible]
3	All sequential commands analyzing. If the instant command is *"out"* – need to determine in which line of sequential code its argument is initialized (*reg*$_{dest}$, see (4)). The row number where this argument is initialized assigned to *sourceOut*$_{dest}$.	 11: mul r32 r18 r12 12: add r33 r31 r32 13: mul r34 r20 r10 14: mul r35 r16 r11 … 24: add r46 r45 r18 25: out r33 1 26: out r39 2 27: out r46 1 [illegible]

Table 3

Recursive method of the code parallelization

Step #	Description	Explanation
1	Initialize parallel code characteristic: – parallel ALU count; – ALU loading level (percentage of commands). Initialized field *executeInLine*$_{dest}$=-1. This field is used by arithmetic, logic and conditional commands. Field *executeInLine*$_{dest}$ is also set for commands *"in"* and *"out"*. Set condition for command placement in parallel code *placeCommand = false*	Example: – ALU count = 4; – ALU loading level (*com_level*) at least 70 %. Command placement condition is used for recursive level identification. It is forbidden to place commands in parallel code on the first level of recursion. Commands are placed in parallel code only on the second level of recursion.
2	The first level of recursion executes. Select arithmetic, logic, condition or input/output command from the sequential code.	
3	If instant command is one of the arithmetic, logic or conditional commands – select *source*$_{arg1}$ and *source*$_{arg2}$ for farther analyzing. If this is '*out*' – select number of command from *sourceOut*$_{dest}$ field for analyzing.	Select commands that initialized input arguments of the instant command select *source*$_{arg1}$ and *source*$_{arg2}$, or *sourceOut*$_{dest}$.
4	According to selected command *executeInLine*$_{dest}$ field is analyzed. Go to step #5 if this field is equal to '-1'. If *executeInLine*$_{dest}$ is equal to instant parallel line (this checking is applicable only for arithmetic, logical or condition commands) – go to step #2, in other case go to step #6.	'-1' means that instant command is not placed in parallel code. Only those commands can be placed in parallel code which arguments are already initialized and placed in parallel code.
5	The second level of recursion executes. Analyze selected command. Set field *placeCommand = true*. Go to step #3.	
6	If *placeCommand* field is *true* – place instant command in parallel code, in other case – increase ALU number and go to the step #2.	



```
  Sequential three-address program code
------------------------------------------
0 :    in      r10     2
1 :    in      r11     1
2 :    in      r12     2
3 :    ld      r14             1
4 :    ld      r16             2
5 :    ld      r18             4
6 :    ld      r20             3
7 :    ld      r24             6
8 :    mul     r29     r14     r10
9 :    mul     r30     r16     r11
10:    add     r31     r29     r30
11:    mul     r32     r18     r12
12:    add     r33     r31     r32
13:    mul     r34     r20     r10
14:    mul     r35     r16     r11
15:    sub     r36     r34     r35
16:    mul     r37     r16     r12
17:    add     r38     r36     r37
18:    add     r39     r38     r24
19:    mul     r41     r24     r10
20:    mul     r42     r18     r11
21:    sub     r43     r41     r42
22:    mul     r44     r16     r12
23:    sub     r45     r43     r44
24:    add     r46     r45     r18
25:    out     r33     1
26:    out     r39     2
27:    out     r46     1
------------------------------------------
```

Parallel ALU count - 4

```
1      in   in   out  out   alu  alu  alu  alu

0:      1    0    25   26     8    9   11   13
1:      |    2    27   |     14   10   16   19
2:      |    |    |    |     20   15   22   12
3:      |    |    |    |     21   17    |    |
4:      |    |    |    |     23   18    |    |
5:      |    |    |    |     24    |    |    |
                           100%  83%  50%  50%
```

ALU's loading level

Parallel ALUs count - 3

```
2      in   in   out  out   alu  alu  alu

0:      1    0    25   26     8    9   11
1:      |    2    27   |     10   13   14
2:      |    |    |    |     15   16   19
3:      |    |    |    |     20   17   22
4:      |    |    |    |     21   12   18
5:      |    |    |    |     23    |    |
6:      |    |    |    |     24    |    |
                           100%  71%  71%
```

Example of placing command #18 in parallel code.

Execute step #2. Select command #26 (this is output command *"out"*). Step #3. ***sourceOut***$_{dest}$ field is analyzed of instant command. This field is equal to command #18. Step #4. ***executeInLine***$_{dest}$ field is analyzed of command #18, this field is equal to '-1'. Step #5. Set ***placeCommand*** = *true*. Step #3. ***source***$_{arg1}$ and ***source***$_{arg2}$ fields of command #18 are analyzed. ***source***$_{arg1}$ field is 17. Step #4. ***sourceInLine***$_{dest}$ field of command #17 is analyzed, this field is equal to '3'. The line '3' is not equal to current line – '4', then repeat step #4 for ***source***$_{arg2}$. ***source***$_{arg2}$ field is 7.
Command #7 is constant command, as a result this command is not used for analyzing since it will be stored in RAM during synthesis process. The arguments of command #18 are already placed in parallel code, thus command #18 can be placed in parallel code according to index of ALU. Step #6. ***placeCommand*** field is analyzed. The field value is ***true***. Place command #18 in parallel code.

Check ALU-loading level with commands for each parallel branch. The minimal command loading level for parallel code #1 is 50 %. According to the input requirements, this value should not be less than 70%. Therefore, reduce number of parallel ALUs from 4 to 3 and repeat parallel process again. The result of this process is parallel code #2 that has minimal ALU loading value 71 %. All requirements are met.

The most optimal example for method efficiency investigation is FFT algorithm [17] [18] [19]. This algorithm allows execute sequential commands parallelization to load ALUs by commands uniformly. If number of parallel commands is equal to FFT base – thus all parallel ALUs is 100 % loaded by commands. The FFT characteristic shows below.

Number of points	64
Base	2
Input ports	2
– Real part	1
– Imaginary part	1
– Output ports	2
– Real part	1
– Imaginary part	1
Constants	32
Number of sequential three-address commands	2208

Based on the parallel code characteristic 30 parallel configurations of FFT algorithm were generated. The maximum number of ALUs (for which cycle of commands execution is the least) is 30. There is no reason to parallel code for bigger number of ALUs since the cycle of loading input and extracting output data for FFT algorithm is 64 commands (based on FFT characteristic of sequential code: (2208-(2·64)-(2·64)-32)/64=30). Plot of the dependency between parallel commands number vs parallel ALUs is created (Fig. 3). This plot shows that the parallelization method is efficient for any number of ALUs < 30. Parallel commands number decrease with exponential dependency without any peaks this mean that all ALUs are 100 % loaded by the commands.

Analyzing information of this graph, we see that calculation cycle for one ALU (1920) is less than sequential commands count (2208). The root cause of this difference is that system uses pipe-line for calculation. The parallel process of input/output data and data processing is used.

Sequential code parallelization based on proposed method doesn't require exponential dependency of computer memory and does not use matrix representation for storing sequential command dependencies [20]. Based on (10) we just need four additional fields for each command to perform parallelization.

The result is linear dependency between number of sequential commands and RAM memory usage.

The main criteria of method evaluation are machine time consuming during algorithm execution. The method consider being efficient if dependency function of time of algorithm execution vs parallel ALUs number is linear and gain of linear function (coefficient α) is less than 1. Fig. 4 shows dependency graph between time required for parallelization and parallel ALUs number.

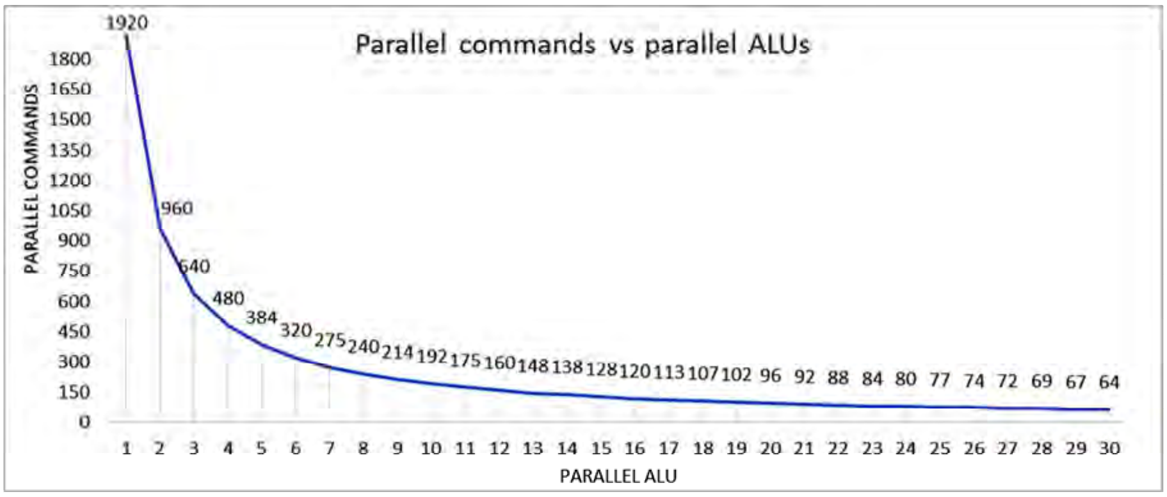


Fig. 3. Parallel commands number vs parallel ALU dependency graph

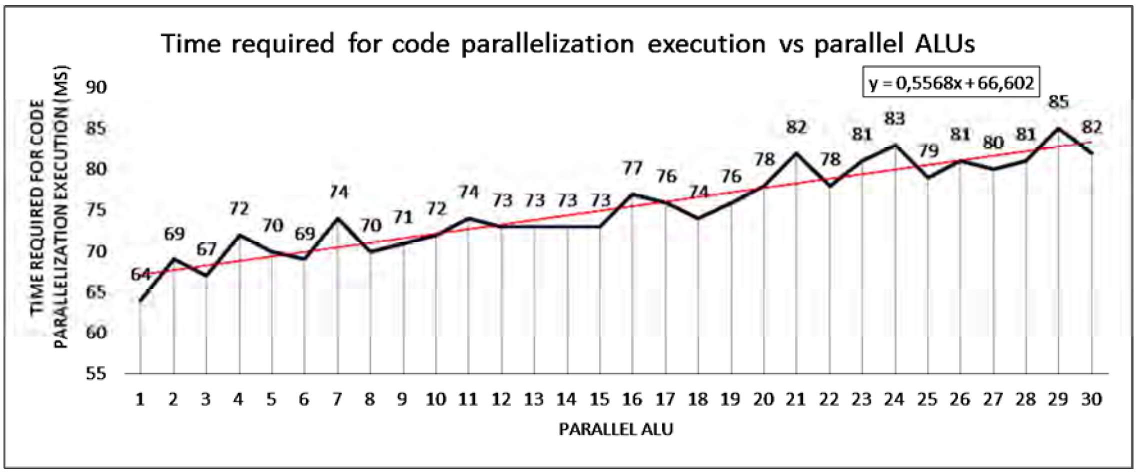


Fig. 4. Dependancy graph between time required for code parallelization vs parallel ALUs

Straight line on this graph is approximated function of the data. This function can be outlined using next equation (11):

$$y = \alpha \cdot x + b \qquad (11)$$

where y – time required for parallelization execution; x – number of ALUs; α – linear function gain; b – approximately time of algorithm execution using just one ALU.

This gain is equal to 0.5568 for current algorithm. This means that proposed method is efficient because dependence function between time required for parallelization and parallel ALUs number is linear (Fig. 4) and gain of this function α is less than 1. So in other words, speed of increasing parallelization time is in two times slower than speed of increasing parallel ALUs. Equation (11) can be used to forecast approximate time required for algorithm execution with specify ALUs number tentatively determine time required for algorithm execution for one ALU.

VIII. CONCLUSIONS

This article describes three-address program parallelization method. Three-address program definition has been determined and a list of all commands (Table 1) is shown. Structure and set of the main requirements for the sequential program code have been determined. These requirements allow determining output characteristics and parallel code structure. Mathematical model of parallel code was created. Purpose of all fields of the algorithm and their structure was described. Method of code parallelization was described according to its requirements. This method contains of two stages: field initialization and recursive method of code parallelization. The input data of code parallelization method is sequential three-address code and desired performance. Desired performance value is specified by minimal percentage of commands of ALU loading. The example of code parallelization is shown that emphasize on reaching of performance target. Efficiency of the method is shown based on FFT 64 algorithm. The graph of dependency between parallel commands and parallel ALUs is shown (Fig. 3). The efficiency of this method is proved based on graph analyzing result. The command loading level for each ALU is almost equal 100 % for any parallel configuration (ALUs number <= 30). The dependency graph between time required for code parallelization vs parallel ALUs is shown (Fig. 4). Based on the data of this graph, dependency between time required for code parallelization execution and parallel ALUs is linear function with gain 0.5568. Taking into account parallel code requirements and investigation results, we can make conclusion that this method is efficient. The method efficiency denominates consuming required for algorithm execution and computer resources usage in time.

REFERENCES

[1] “Compile Your C code into Verilog”, [Online]. Available: http://c-to-verilog.com/index.html.

[2] “C-to-FPGA Solutions”, [Online]. Available: http://www.impulseaccelerated.com/products_universal.htm.

[3] “Handel-C Synthesis Methodology”, [Online]. Available: http://www.mentor.com/products/fpga/handel-c/.

[4] I. Innovations, “CHAMELEON – the System-Level Design Solution”, [Online]. Available: http://intron-innovations.com/?p=sld_chame.

[5] A. Melnyk, A. Salo, V. Klymenko та L. Tsyhylyk, “Chameleon – system for specialized processors high-level synthesis”, Scientific-technical magazine of National Aerospace University “KhAI” No. 5, pp. 189-195, 2009.

[6] A. Melnyk, A. Salo, “Automatic generation of ASICs”, NASA-ISA Conference AHS-2007, pp. 96-101, 2007.

[7] D. Cordes, A. Heinig and P. Marwedel, “Automatic Extraction of Pipeline Parallelism for Embedded Software Using Linear Programming”, in IEEE 17th International Conference on Parallel and Distributed Systems, 2011.

[8] J. V. Dyken and J. O. Delgado-Frias, “A Medium-Grain Reconfigurable Processor Organization”, School of Electrical Engineering and Computer Science, Washington, 2011.

[9] A. Melnyk, “Design of SCS”, 1996.

[10] L. Tsyhylyk, “Transformation Method of conditional comands in parallel way”, Bulletin of National University “Ukraine”, pp. 156–159, 2010.

[11] D. Cordes, M. Engel, O. Neugebauer and P. Marwedel, “Automatic Extraction of Pipeline Parallelism for Embedded Heterogeneous Multi-Core Platforms”, Dortmund, Germany, 2013.

[12] A. Gontmakher, A. Mendelson, A. Schuster and G. Shklover, “Code Compilation for an Explicitly Parallel Register-Sharing Architecture”, in International Conference on Parallel Processing, 2007.

[13] C. Roth, S. Reder, H. Bucher, O. Sander and J. Becker, “Adaptive Algorithm and Tool Flow for Accelerating SystemC on Many-Core Architectures”, in 17th Euromicro Conference on Digital System Design, 2014.

[14] T. Bernard; K. Bousias; L. Guang; C. R. Jesshope; M. Lankamp; M. W. van Tol; L. Zhang, “A General Model of Concurrency and its Implementation as Many-core Dynamic RISC Processors”, Institute for Informatics, University of Amsterdam, Amsterdam, Netherlands, 2008.

[15] A. Melnyk and V. Melnyk, Personal Supercomputers: Architecture, Design, Application, Lviv: Lviv National Polytechnic University Publishing, 2013.

[16] L. Yan, B. Wu, Y. Wen, S. Zhang and T. Chen, “A reconfigurable processor architecture combining multi-core and reconfigurable processing unit”, in 10th IEEE International Conference on Computer and Information Technology (CIT 2010), 2010.

[17] A. Melnyk and B. Dunets, “FFT Processor IP Cores synthesis on the base of configurable pipeline architecture”, CADSM’2003, Lviv-Slasko, 2003.

[18] V. Chandrakanth; Tripathi Srijan, “Customized Architecture For Implementing Configurable FFT on FPGA”, 3rd IEEE International Advance Computing Conference (IACC), pp. 1280–1282, 2013.

[19] Y. Li, Z.-y. Wang, J. Ruan and K. Dai, “Research and Implement a Low-Power Configurable Embedded Processor for 1024-Point Fast Fourier Transform”, in School of Computer, National University of Defense Technology, Hunan Changsha, P. R. China, 2007.

[20] A. Melnyk, I. Yakovleva, V. Uschenko, “Design and Matrix representation of Data Flow Graph”, Bulletin of Vinnitsky Polytechnic Institute No. 3, pp. 93–99, 2009.

[21] “C to HDL”, 16 September 2014. [Online]. Available: http://en.wikipedia.org/wiki/C_to_HDL.

Liubomyr Tsygylyk (l.tsyhylyk@gmail.com) received his MS degree at Electronic Computational Machine Department of Lviv Polytechnic National University, Lviv, Ukraine in 2007. Since 2008, he has been working as assistant at Lviv Polytechnic National University. His research interests include high-performance computations, FPGA-based systems, C to HDL code compilation and configurable processors synthesis.

He is the author of eight articles and several theses. Current research includes C to HDL algorithm compilation with specialized processors synthesis using full set of C-language structures.