Original scientific paper

Informacije MIDEM
Journal of Microelectronics,
Electronic Components and Materials
Vol. 45, No. 4 (2015), 225 – 236

Instruction Decompressor Design for a VLIW Processor

Abdul Rehman Buzdar, Liguo Sun, Azhar Latif, Abdullah Buzdar

University of Science and Technology of China (USTC), Department of Electronic Engineering and Information Science, Hefei, Peoples Republic of China

Abstract: The FlexCore processor is a wide instruction word processor, which allows the control of datapath elements at a very precise level. The FlexCore scheme offers full control over the architecture and helps to improve the overall performance. As the memory is very expensive in embedded systems both in terms of power and area, to gain the full advantages of long instruction word of the FlexCore we need to use the memory footprint very efficiently. To remedy this the instructions in the FlexCore processor memory are stored as application-specific, compressed instruction format (AS-ISA) which is then converted on-the-fly to a native, decompressed instruction format (N-ISA) by an instruction decompressor. This paper deals with the implementation of the instruction decompressor and the analysis of compression and decompression schemes used in the FlexCore processor. The instruction decompressor is designed and implemented in VHDL and synthesized using Cadence RTL compiler into three different process technologies 130-nm, 90-nm, and 65-mn provided by the STMicroelectronics. The synthesis results show that the design and implementation of instruction decompressor greatly impacts the performance of FlexCore in terms of power, area and timing. We show the impact of different parameters of compression scheme used for the implementation of instruction decompressor in hardware which was previously shown in software. These parameters include the formation of lookup table (LUT) groups, the size of LUTs and the LUT-Load instruction Interval meaning how often the LUTs needs to be updated and how many LUTs are updated through a single LUT-Load instruction.

Keywords: FlexSoC; FlexCore; VLIW Processor; Instruction Decompressor; LUT; ASIC

Ukazni dekodirnik za VLIW procesor

Izvleček: Procesor FlexCore je procesor z zelo dolgo ukazno besedo, ki omogoča kontrolo poti elementov z visoko natančnostjo. Shema FlexCore pmogoča popolni nadzor nad arhitekturo in omogoča izboljšavo delovanja. Za doseganje vseh prednosti dolge ukazne besede in visoke cene pomnilnika je potrebno spomin učinkovito izrabiti. Ukazi so v spominu FlexCore procesorja shranjeni kot aplikacijsko specifični in stisnjeni v AS-ISA formatu. Dekodiranje v N-ISA format poteka v ukaznem dekodirniku. Ukazni dekodirnik, opisan v članku, je realiziran v treh tehnologijah (130 nm, 90 nm in 65 nm). Rezultati kažejo, da ima dizajn in implementacija velik vpliv na učinkovitost procesorja v luči moči, prostora in časa. Vplivi parametri so prikazani v strojni opremi. Ti parametri vključujejo tvorjenje skupin vpoglednih tabel (LUT), njihovo velikost in potreben interval njihovega osveževanja.

Ključne besede: FlexSoC; FlexCore; VLIW Procesor; Ukazni dekoder; LUT; ASIC

** Corresponding Author's e-mail: abdul.buzdar@alumni.chalmers.se*

1 Introduction

There is an ever increasing demand for the electronic gadgets to have a wide range of applications ranging from multimedia to video games and the list of demands is increasing day by day. To efficiently manage all these applications the electronic devices should have functionalities offered by general purpose processors and must also be efficient in terms of both power and area. This is a demanding task, to run the applications which are compute-intensive, one has to use specialized hardware accelerators or dedicated application-specific processing units which are controlled by microprocessors [1-4], such as an ARM core [5], placed on a single chip. The memory management is also very critical for embedded systems both in terms of cost and area. To accommodate these hardware accelerators the I/O activity and memory usage has to be kept down. The approach of adding hardware accelerators in this way does not cater the rapidly changing depends of users, so we need to have an architecture which offers the efficiency of an ASIC and flexibility of a programmable platform. The demand for the embedded systems to have higher performance and more functionality makes general purpose processors unsuitable for

them. The higher functionalities offered by the general purpose processor comes with a cost of higher power dissipation which will result in shorter battery life and increased weight in the form of cooling parts. To gain the required performance in the embedded systems with low power and small area, using heterogeneous system-on-chips is one of the options [6-9]. The heterogeneous SoCs uses some special purpose hardware blocks, which are controlled by one or more embedded microprocessors. One of the major drawbacks of heterogeneous SoCs is their high non-recurring engineering (NRE) costs.

Application-Specific Instruction-set Processors (ASIPs) [10-14] try to combine the flexibility of programmable processors and the efficiency offered by the customized integrated circuitry. The ASIPs are generally constructed by adding specialized hardware blocks to programmable processor cores. The instruction set of ASIPs consists of some general instructions to gain the advantage of general purpose processors and some application-specific instructions to gain the efficiency of specialized hardware. This scheme makes it easy to add specialized hardware blocks in the existing datapath and subsequently add application-specific instructions. By modifying the application software running on ASIPs, late design alterations can be accommodated easily, enabling flexible and high performance SoCs. This makes it possible to adopt a hardware-software co-design methodology, in which the conventional software design flow can be adopted. The major drawback of ASIPs is that as the addition of new instructions make them prone to binary incompatibility issues between various hardware implementations.

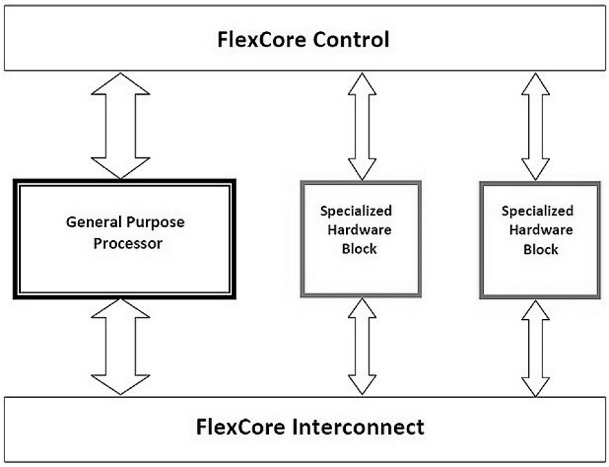

Figure 1: Overview of FlexCore processor

The FlexCore processor [15-20] which is based on the concept of the FlexSoC, is an attempt to integrate the efficiency of an ASIC (or special-purpose hardware) and the flexibility or programmability of general purpose processors. The FlexCore integrates all the functional units in a homogenous way to take the advantage of traditional general purpose processors, shown in Fig. 1. The specialized hardware blocks are added into the datapath of general purpose processor to gain the benefits of conventional five stage pipelined processors. The FlexCore processor does not have a standard instruction set architecture (ISA) like that offered by conventional general purpose processors, in which the ISA is used to control the pipeline stages of the processor at various clock cycles. The FlexCore is a wide control word processor which controls the datapath at a much finer grained level than conventional processors. The FlexCore processors wide control word takes a single cycle to control the whole datapath. The datapath units of a FlexCore processor consist of conventional five-stage processor components and some specialized hardware blocks. The wide control word of the FlexCore processor contains all the signals to every datapath unit and the interconnecting structure. The use of a wide control word gives full control of underlying hardware to the programmer/compiler, resulting in increased performance, which lacks in the conventional instruction set architecture (ISA) approach. The previous research on datapath [23-27] has shown to improve the efficiency due to increased controllability.

2 FlexCore processor Architecture

The Baseline FlexCore processor [15-20] without any hardware accelerators and datapath units connected in their minimum configuration, act as a single issue five-stage pipelined processor e.g. similar to the Hennessy-Patterson 32-bit DLX [21] and MIPS R2000 [22] . This feature of the FlexCore makes it possible to execute the application code of a general purpose processor as efficiently as a single issue five-stage processor. Unlike the conventional methods, the performance benefits in the FlexCore processor are gained through the use of hardware accelerators and the fine grained control of datapath units. Depending on the application requirements, the FlexCore processor can be easily extended with special-purpose hardware accelerators [29], [30]. The FlexCore processor has a native ISA (N-ISA), which is 91-bit wide, when no hardware accelerators are used. The N-ISA is capable of controlling the datapath units and interconnects at a very fine-grained level. The instructions in the memory of the FlexCore are stored as applications specific ISAs (AS-ISA), which are then converted on-the-fly to a native ISA (N-ISA) format, by a reconfigurable instruction decompressor.

The AS-ISA can be configured for a particular class of applications, those who have identical processing needs. The addition of new application needs only to

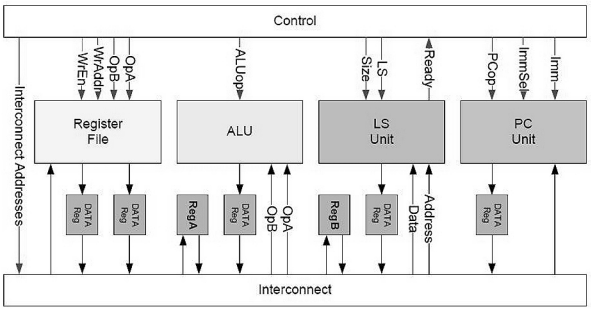

Figure 2: Baseline FlexCore processor

define a new ASISA, thus the N-ISA and the translation process would remain unchanged. This feature of defining a new AS-ISA offers a possibility for performance optimization for the compiler e.g. using the already available instruction sequence instead of expanding the N-ISA. Fig. 2 shows the datapath units used in a baseline FlexCore processor. It consist of a register file, arithmetic and logic unit (ALU), load/store unit and a program counter unit. All these datapath units are fully interconnected, meaning that the interconnect configuration can be changed for different application requirements during the design stage. The baseline FlexCore has many unused interconnect paths that may be removed later, which is one of the main reason for the FlexCore enhanced performance. The output of each datapath unit is connected to a data register, which acts as pipeline registers, so that the FlexCore can emulate the functionality of a general purpose processor. Since data can be routed to any place, different datapath pipeline schemes can be created. The flexCore processor can be extended with new hardware accelerators depending on an application requirements. The FlexCore processor was used to run fast Fourier transform (FFT) benchmark application. Since this algorithm makes extensive use of multiplication operations, the baseline FlexCore was extended with a 32-bit multiplier unit, shown in Fig. 3. The addition of a multiplier unit also affected the size of N-ISA with the addition of two 32-bit inputs, 64-bit output and an enable signal, became part of N-ISA. The N-ISA of multiplier extended Baseline FlexCore processor consists of 109-bit control signals.

The concept of the FlexCore N-ISA is very different from the conventional ISA approach, and in this way changes the abstraction level at which the compiler/programmer manages the datapath and interconnect. The conventional ISA of a general purpose processor contains instructions like ADD, SUB etc. and the results of these instructions are stored on the register file. In case of a statically scheduled processor if the input operands are not yet available, the processor needs to be stalled and wait for the input operands. However in a

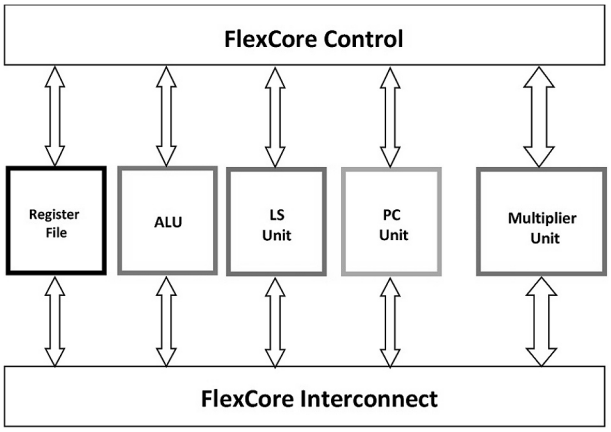

Figure 3: FlexCore Processor extended with Multiplier

dynamically scheduled processor the result of previous instructions, can be rerouted if it has been executed but not yet written on the register file. This technique makes the scheduling process simpler, but reduces the performance because of putting extra load on the register file. Instead of storing back every result unnecessarily on the register file, it can be routed directly to the instructions that needs it. The FlexCore compiler [33], [34] has complete control over the datapath units and interconnects for each clock cycle. For example while performing the multiplication operation the FlexCore compiler will set the control signals for the multiplier unit, when the input values for the multiplier are available at the right clock cycle and route the result of multiplication to the destination, where it is needed. This technique improves the overall performance of the system at the cost of complicating the scheduling process. In this way the compiler can freely route the data to any destination. This results in the minimum register file access as the data can be routed to the place where it is required, instead of storing it on a register file. Hence this technique saves power and improves performance.

3 Flexible datapath interconnect

The flexible interconnect of the baseline FlexCore processor [20] consists of a matrix switch, shown in Fig. 4. This means that there is a multiplexer connected to the inputs of each datapath unit, which can select any of the inputs coming from output ports of other datapath units. This maximum freedom of routing the data to any location, results in scheduling efficiency in contrast to a general purpose processor, where there are limited options for routing. This also helps the compiler to control the order of the pipeline stages and increase the efficiency of datapath units.

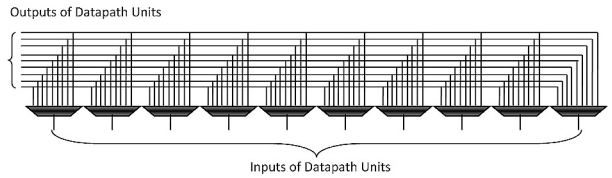

Figure 4: Illustration of FlexCore Datapath Interconnect

The FlexCore processor is statically scheduled, which means that the compiler knows in advance which interconnect paths will be used for a particular set of applications. This can help to save power and improve performance by removing the unused interconnect paths based on the application profiling at design time. To make sure that the FlexCore can emulate the functionality of a general purpose processor, those interconnect paths which are necessary for the FlexCore to act as a general purpose processor, are not removed. The research on the FlexCore flexible interconnect, shows that the performance improves when just a few paths are added beyond the GPP case and almost half of the interconnect paths are never used by a particular set of applications executed. So these unused paths are removed physically at design time, without any impact on the performance and the number of cycles needed to execute a set of applications.

4 The FlexSoC framework

A lot of work has been done on the FlexSoC framework, since this project has started. The FlexSoC framework [33], [34] consists of a compiler, simulator and a hardware generator, shown in Fig. 5.

4.1 Compiler

The input to the compiler is the MIPS assembly which is produced by a MIPS cross-compiler. The EEMBC [28] benchmarks have been used to produce MIPS assembly and then compile it using FlexSoC compiler. The output of the compiler is Register Transfer Notations (RTN) format instructions. These RTN format instructions are statically scheduled and are used to exploit the inherent parallelism of the FlexCore processor. These instructions later can be used to compare the performance of FlexCore with a general purpose processor.

4.2 Simulator

A cycle accurate simulator is implemented in Haskel and is capable of simulating both the FlexCore and MIPS assembly. This feature of simulator helps to trace bugs in the compiler and measure its performance. The simulator is capable of giving simulation cycle count,

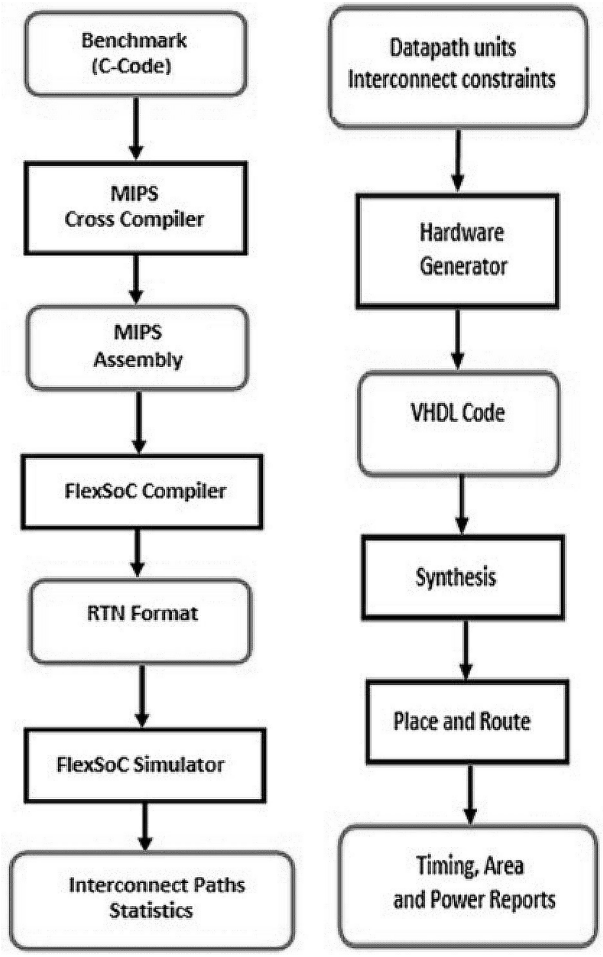

Figure 5: Illustration of FlexSoC Framework

profiling and simulation trace statistics with accuracy. As the FlexCore processor is flexible in terms of both its datapath units and their interconnections, this feature can be emulated in the simulator and the simulation of FlexCore processor can be done in different hardware configurations. The simulator can also be configured to a single issue five-stage processor to emulate a general purpose processor.

4.3 Hardware Generator

The FlexSoC hardware generator is capable of generating VHDL code for the FlexCore processor in different configurations, some of which have been implemented on FPGAs. The FlexSoC framework also has the capability of verifying the VHDL code generated and synthesis, place and route features have also been provided. It also gives information about area, timing and power usage.

5 Existing compression schemes

FlexCore is a wide instruction word processor, so to take the full advantage of the expressiveness found in its wide control word, the instructions are stored on the memory in compressed format. Let's take a brief look at the compression scheme used in the FlexCore processor. The main idea behind the encoding scheme [35] is the use of lookup tables (LUTs) to store the bit patterns, Shown in Fig. 6.

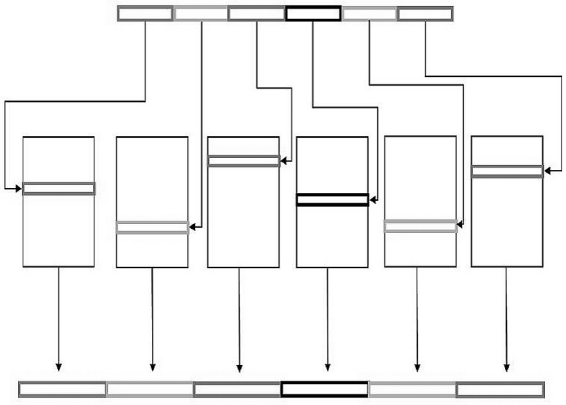

Figure 6: Illustration of Compression Scheme Implemented

The indexes of these LUTs are then combined to form the compressed instructions. The bit patterns are generated at compile-time based on the fact that some combination of bits in the control word of the FlexCore will not be used in some portions of the code being executed. The full advantage of the expressiveness found in the wide control word of the FlexCore processor is thus not utilized. This technique can be implemented in hardware with a simple logic and the sizes of the LUTs are also reasonably small. The contents of the LUTs can be changed using special instructions (LUT-Load instructions) and the bit patterns to be stored in the LUTs are sent through these Load instructions. The processor is stalled each time the contents of LUTs need to be changed, so the placement of the LUT-load instructions will affect the overall performance. The size of the LUTs will affect the compressed instruction size and the interval of LUT-Load instructions. The indices of the LUTs are combined to form the compressed instruction, and the size of the LUT decides the number of bits needed for each index. The main goal of this compression scheme is to utilize the expressiveness found in the wide control word of the FlexCore processor and to be able to store large programs, yet keeping the runtime costs low. The compression scheme [35] is also associated with a methodology for the partitioning of wide instruction stream that is, how many LUTs will be needed for a particular application and what should be the size of each LUT. The NISC [23-27] project also proposes the use of LUTs for compression and decompression of long instruction word. It uses only one or two LUTs to store the entire program, making the LUT size very large. Therefore it is more suitable for implementing on FPGA, rather than on an ASIC platform.

6 Implementation of compression scheme

The compression scheme [35] is implemented in VHDL to study the impact of this scheme on the performance of the FlexCore processor in terms of area, timing and power requirements. Let's take a look at the specification of the instruction decoder implemented. The 71-bit instruction stream is coming from the Cache of the FlexCore processor, as an input to the instruction decoder. The 71-bit instruction stream consists of 39 bits of instructions coming from I-Cache, and 32 bits of data coming from D-Cache. There are two types of instructions, shown in Fig. 7, one to load the LUTs with new content (Load instructions) and one used to send the already stored content of the LUTs to form the decompressed full 109-bit wide control word of the FlexCore (Normal instructions). The last bit of 71-bit wide compressed instruction is used to decide between the two types of instructions. One entry each of two LUTs can be loaded with a single LUT-Load instruction. The two instruction types consist of sub fields, shown below:

Load Instruction:
6-bit Index of LUTn, 6-bit Index of LUTm, Data of LUTn, Data of LUTm, Unused bits, 8 Ctrl bits, Load=1

Normal Instruction:
LUT1 address, LUT2 address, LUT3 address LUT8 address, 32-bit imm, Load=0

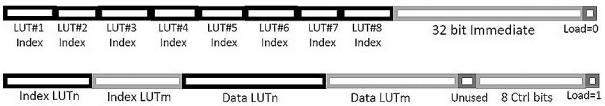

Figure 7: Illustration of Instruction Format used

The index of the LUT decides the depth of each LUT, with n-bit index the depth of the LUT would be 2^n. Fig. 8 illustrates the implementation scheme of the FlexCore processor with the instruction decoder. The 109-bit control word of the FlexCore is divided into eight groups and each group forms one LUT. These groups are formed using the FlexSize tools, which were developed for implementing the compression scheme [35].

Table 1: Specification of LUTs Implemented

LUT Name	LUT Index Bits	LUT-Entry Data Bits
ALU Group	6	13
RFA Group	5	6
RFB Group	5	6
RFW Group	6	10
LS Group	4	13
BUF Group	4	10
PC Group	4	9
Mult Group	4	10

Table I shows the specification of eight LUTs used in the implementation of the instruction decompressor. Here the LUT-Entry Data Bits indicate the width of each LUT and index bits are the minimum bits required to access all entries of each LUT. The sum of all Index bits of each LUT group, 32 immediate bits and one bit for indicating the instruction type equals 71 bits, the total length of compressed instruction which is the input to the instruction decompressor i.e. :

6+5+5+6+4+4+4+4+32+1 = 71 bits

The output of the instruction decompressor is 109-bit wide control word of the Baseline FlexCore processor, which is formed by concatenating all the data bits from one entry each of eight LUTs and 32 immediate bits i.e. :

13+6+6+10+13+10+9+10+32 = 109 bits

The above mentioned LUT groups are formed using a methodology which is used for the partitioning of the wide instruction word into smaller groups and is associated with the compression scheme [35]. The method consists of four steps, the first step is the identification of bits that are highly correlated and should be placed in the same group. Later the groups formed are evaluated using a user-defined cost function. In our case the LUT-access time, compression ratio and energy efficiency forms the cost function. Here the energy efficiency means that to reduce the power dissipated by the instruction decompressor during the LUT-Load and Normal instructions.

7 Instruction decompressor

Fig. 9 shows the block diagram of instruction decompressor, it consists of a main unit and eight LUT units, which act as simple memory units. As the input to the instruction decompressor is 71-bit compressed instruction stream, which is divided into different sub fields internally in the main unit to control the eight LUT units.

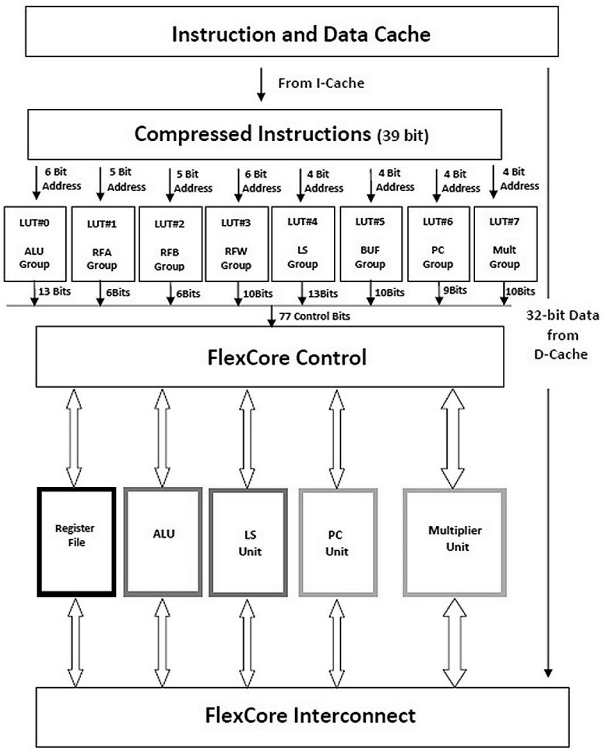

Figure 8: FlexSoC scheme with Instruction decompressor

The 8-bit ctrl field of CTN-ISA is used to indicate which LUT unit to load, and one bit each of 8-bit ctrl field is connected to the Load signal of LUT units. The address bits coming through the CTN-ISA, are connected to each LUT unit address signal which is used to decide which LUT entry to load or to send the stored data out in case of Normal instructions.

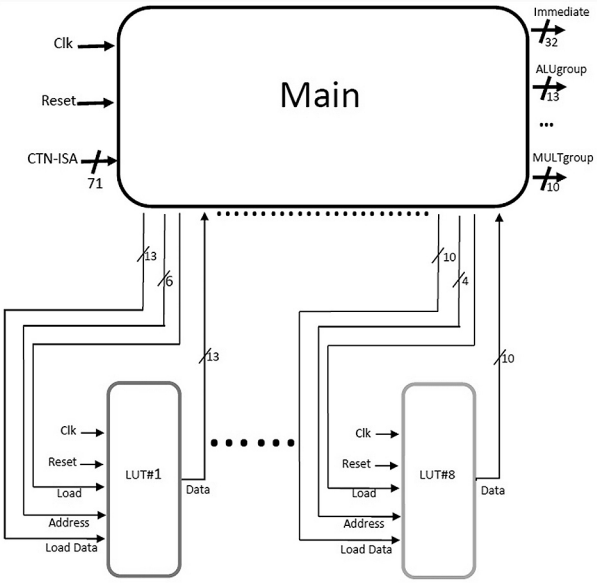

Figure 9: Block Diagram of Instruction decompressor

Similarly data bits coming through CTNISA are connected to the Load-Data signals of each LUT unit, which carries the data to be loaded in to the specified LUT address. Fig. 10 shows the block diagram of a LUT unit, it consists of a DeMux which is used to select the LUT-Load Data based on the LUT-Address, to send it to the specified LUT entry. In case of an n-bit wide LUT, n flip-flops for each LUT entry are used to store the LUT-Entry Data. A multiplexer is used to select which LUT-Entry Data is to be send to the output, based on the LUT-Address. Fig. 11 shows the input output pin configuration of the instruction decompressor. The pins on the left are the input pins and the pins on the right are the output pins. The detail of each pin is as follows:

- Clk /Reset
  As the name implies the Clk pin is used as an external clock to the instruction decompressor and the Reset pin is used to give global reset to the instruction decompressor.
- CTN_ISA
  This pin is used to get the 71-bit compressed instruction stream as an input into the instruction decompressor from the Cache of the FlexCore processor.
- Immediate
  This pin is used to output the 32-bit immediate data coming from the D-Cache of the FlexCore processor.
- ALUgroup
  This pin is used to output the 13-bit wide data from one of the entries of ALU LUT and contains the signals for ALU of the FlexCore processor.
- RFgroupA
  This pin is used to output the 6-bit wide data from one of the entries of RFA LUT and contains the signals for Register File of the FlexCore processor.
- RFgroupB
  This pin is used to output the 6-bit wide data from one of the entries of RFB LUT and contains the signals for Register File of the FlexCore processor.
- RFgroupW
  This pin is used to output the 10-bit wide data from one of the entries of RFW LUT and contains the signals for Register File of the FlexCore processor.
- LSgroup
  This pin is used to output the 13-bit wide data from one of the entries of LS LUT and contains the signals for Load Store Unit of the FlexCore processor.
- BUFgroup
  This pin is used to output the 10-bit wide data from one of the entries of BUF LUT and contains the signals for Interconnect and Buffer of the FlexCore processor.
- PCgroup
  This pin is used to output the 9-bit wide data from one of the entries of PC LUT and contains the signals for PC unit of the FlexCore processor.
- MULTgroup
  This pin is used to output the 10-bit wide data from one of the entries of Mult LUT and contains the signals for Multiplier unit of the FlexCore processor.

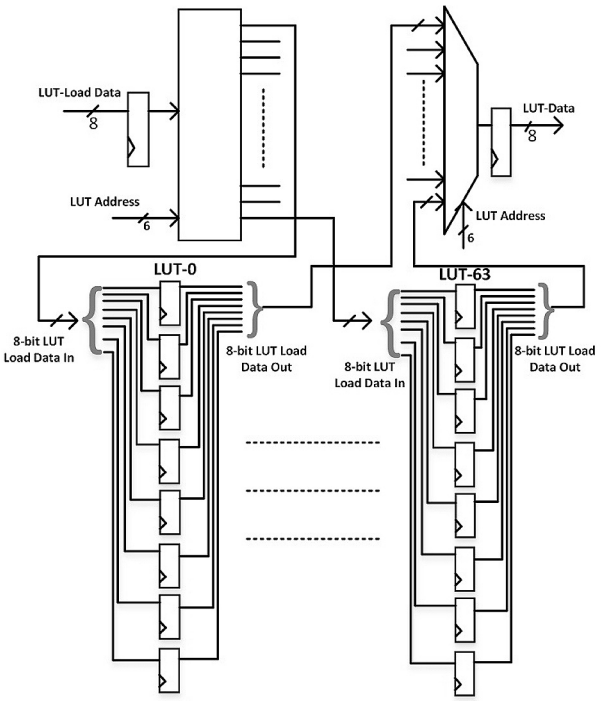

Figure 10: Block Diagram of a LUT Unit

The timing diagram in Fig. 12 shows what happens during a LUT-Load operation. When the load signal goes high, the data coming from the I-Cache is loaded into the specified entry of that LUT. One entry each of two LUTs can be loaded through one LUT-Load instruction. The LUT-Load instruction takes one cycle to load the data into the specified LUT entry.

Similarly, the timing diagram in Fig. 13 shows what happens during a Normal Instruction. When the load signal goes low the address of each LUT entry for eight LUTs is sent to corresponding LUTs and the data corresponding to each address is sent out on eight output pins. This operation takes a single cycle.

8 Implementation of instruction decompressor

After the VHDL implementation of the instruction decompressor, the next task was to synthesize the VHDL

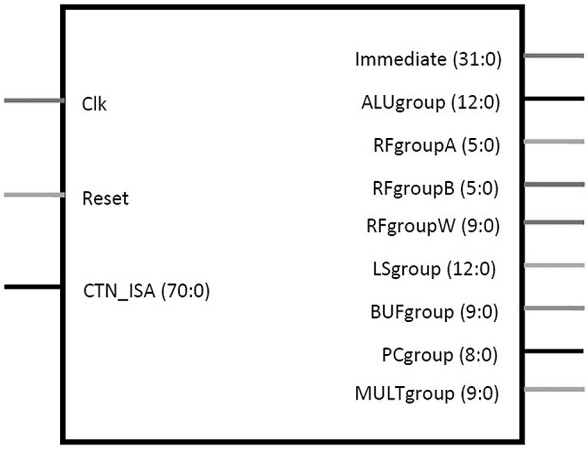

Figure 11: Instruction Decompressor Pinout

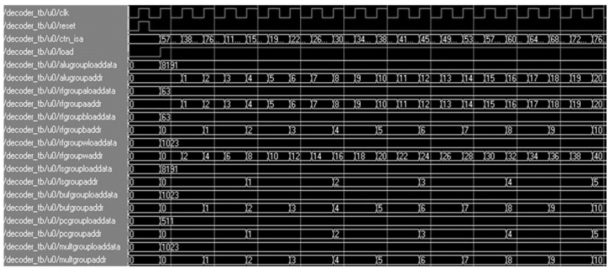

Figure 12: Timing Diagram of LUT-Load operation

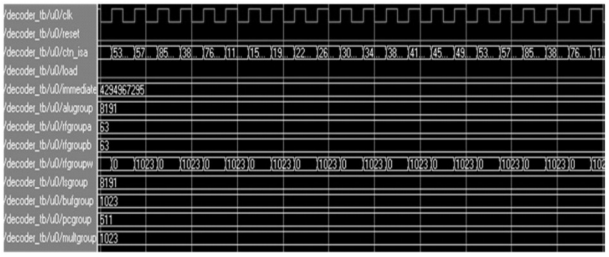

Figure 13: Timing Diagram of Normal Instructions

description to a certain process technology using Cadence RTL compiler [31]. Three different process technologies 130-nm, 90-nm, and 65-mn were used for synthesis, provided by the STMicroelectronics [32]. But we present here only the synthesis results for 65-nm technology. The aim of synthesizing the VHDL description of the instruction decompressor is to study the impact of inclusion of the instruction decompressor into the FlexCore processor in terms of timing, area and power requirements. The reason for this study is that the instruction decompressor will greatly affect the overall performance of the FlexCore processor, because its purpose is to efficiently manage the memory footprint. The focus of this section would be to study the impact of lookup tables (LUTs), in terms of power usage which are used in implementing the instruction decompressor. Also study the effect of LUT-Load instruction Interval, meaning how often the LUTs needs to be updated and how many LUTs are updated through a single LUT-Load instruction. After starting the RTL Compiler, some basic steps were performed such as setting up the library paths for 130-nm, 90-nm and 65-nm process technologies and linking the VHDL files required for synthesis. The RTL Compiler was instructed to assemble the VHDL files into an internal representation i.e. network of virtual gates using the elaboration command. The VHDL code of the instruction decompressor was found to be synthesizable with no errors. The next step in the synthesis process was to map the network of virtual gates to real hardware that is to the real standard cells provided by the STMicroelectronics. Initially no timing constraint was set. Also, a low computational effort was used to get some idea of the intrinsic timing behavior of the implementation, via Static Timing Analysis (STA). The worst-case delay and area of implementation were documented. The worst-case signal propagation path was found to be passing through RFgroupW LUT, because the size of this LUT is bigger than most of the other LUTs implemented for the instruction decompressor.

The clock frequency for the FlexCore processor was set to 400 MHz. The design was re-synthesized with the timing constraint of 2.5 ns and using medium effort. The worst-case delay and area of implementation were documented again for these specifications. This time the worst-case signal propagation path was found to be passing through ALUgroup LUT, since this LUT is width and length wise bigger than the other LUTs implemented for the instruction decompressor. Table II shows the timing and area results for the instruction decompressor. The worst-case delay value shows that the instruction decompressor can be synthesized with a more strict timing constraint.

Table 2: Timing and Area results

Timing Constraint (ps)	Synthesis Effort	Worst-case Delay (ps)	Estimated Area (μm2)
no	low	1053	44840
2500	medium	1240	44551

The power analysis of instruction decompressor was performed initially by assigning some switching probabilities on the primary data inputs using medium effort. Table III shows the power results with probability for high logic state on CTNISA=0.5, Reset=0.0 and toggling probability (ns) on CTNISA=0.02, Reset=0.0.

Table 3: Initial power results

Leakage Power(mW)	Dynamic Power(mW)	Total Power(mW)	Clk Net Power(mW)
1.694	11.086	12.781	1.511

Later different test vectors were generated, by setting different LUT-Load intervals and the number of LUTs loaded through a single Load instruction. Three different set of test vectors were generated setting 60, 100 and 300 as LUT-Load instruction intervals, each set having a total of 20000 test vectors. Two variants of these three set of test vectors were also generated, first by setting one entry of a single LUT is loaded through one LUT-Load instruction and the other one by setting that one entry each of two LUTs is loaded through a single Load instruction.

Table 4: Signal Statistics for test vectors from TCF files

Test Vectors Type	Toggle Rate (toggles/ns) CTN-ISA	Toggle Rate (toggles/ns) NISA
Less Random	0.0858	0.0358
More Random	0.1808	0.1632

Later another set of test vectors was also generated keeping the same specifications as mentioned earlier, but this time the LUT-Load data fields and immediate field were generated, where as in the previous version of test vectors the LUT-Load data and immediate fields do not have much variations among the test vectors. The reason for generating these two set of test vectors is to get a better idea of power consumption of the instruction decompressor. The first version of the test vectors will be referred to as test vectors having less randomness and the later one as test vectors having more randomness in this document. The Table IV shows the signal statistics for test vectors, obtained from the Toggle Count Format (TCF) files.

Table 5: Power results with test vectors having less randomness

No. of LUTs Loaded	LUT-Load Instruction Interval	Leakage Power (mW)	Dynamic Power (mW)	Total Power (mW)
two	60	1.816	2.720	4.536
one	60	1.804	2.740	4.544
two	100	1.815	2.717	4.533
one	100	1.804	2.734	4.539
two	300	1.809	2.703	4.512
one	300	1.814	2.711	4.525

Tables V and VI shows the power results using 20000 test vectors having less and more randomness respectively for the instruction decompressor.

Table 6: Power results with test vectors having more randomness

No. of LUTs Loaded	LUT-Load Instruction Interval	Leakage Power (mW)	Dynamic Power (mW)	Total Power (mW)
two	60	1.713	10.747	12.460
one	60	1.721	10.758	12.480
two	100	1.713	10.721	12.435
one	100	1.721	10.741	12.462
two	300	1.713	10.712	12.425
one	300	1.721	10.728	12.449

To compare the power dissipation of Normal and LUT Load instructions more precisely, two set of test vectors were generated each having 1000 test vectors, one set only contained Normal instructions while the other one only contained the LUT-Load instructions. Later power analysis was performed using these two set of test vectors. Table VII shows the power comparison of Normal and LUT-Load instructions using 1000 test vectors for the instruction decompressor.

Table 7: Power comparison of Normal and LUT-Load Instructions

Instruction Type	Leakage Power(mW)	Dynamic Power(mW)	Total Power(mW)
Normal	1.616	10.239	11.855
LUT-Load	1.702	10.435	12.138

Table VIII shows the synthesis results for the FlexCore processor with full interconnect configuration, synthesized with medium effort and timing constraint of 3 ns.

Table 8: Synthesis results of the FlexCore processor

Benchmark EEMBC-Telecom	No. of Instructions	Cycle Count	Total Power (mW)	Estimated Area (μm^2)
autcor	1399	16110	7.30	49527
fft	1730	136596	8.91	
viterb	1639	265291	7.80	
conven	1457	262039	7.45	

9 Discussion on synthesis results

The results of power analysis shows that the power consumption of the instruction decompressor slightly decreases with reducing the LUT-Load instruction interval, which is obvious because less switching would take

place. It means that, applications that will require less LUT reloads would consume less power, not by much. Another observation is that the power consumption with updating a single entry each of two LUTs is lower than with updating a single entry of one LUT through a single LUT-Load instruction. This is because more LUT-load instructions would be required for loading all the entries of eight LUTs, than with updating a single entry of one LUT through a single LUT-Load instruction. Fig. 14 shows the power results with 20000 test vectors having less randomness for the instruction decompressor, synthesized with medium effort and timing constraint of 2.5 ns using three different process technologies.

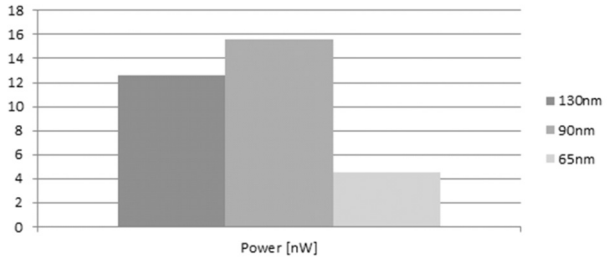

Figure 14: Power comparison of Instruction Decompressor

If we compare the power consumption of the instruction decompressor between the three different technologies, we can see that the power consumption is higher for 90nm than for 130nm technology, but the worst case delay and area is smaller for 90nm than for 130nm. As the timing constraint for both the technologies is same, the higher worst case delay and area for 130nm suggests that it should have higher power consumption than for 90nm technology, since it has to put more effort to meet this timing constraint which results in higher worst case delay and area. The technology files used for 90nm technology, can be a reason for these unexpected results. The LUT-Load instruction interval do not affect the power consumption of the instruction decompressor to a greater extent, which was shown previously in software [35] and this implementation confirms the idea in hardware. The major drawback of having more LUT-Load instructions is that the processor needs to be stalled each time the contents of a particular LUT is updated. So the LUT Load instruction interval must to be kept down for optimum performance. After observing the wide control word of the FlexCore processor, one can see that some combination of control bits e.g. (MULTA, MULTB, READ ADDR1 REG, READ ADDR2 REG) are most of the time zero and the compression scheme takes advantage of this fact. Also if we see the compressed instructions produced by the compression algorithm, most of the bits remain zero repeatedly, which can help to reduce power consumption because less switching would take place. If we look at the power consumption of individual LUT groups, more power is being consumed by the LUT groups having large size, which is obvious. It will be a good idea to reduce the sizes of larger LUT groups and see its effect on the power consumption of instruction decompressor. The synthesis results for the Instruction decompressor were obtained using a timing constraint of 2.5 ns, but synthesis results for the FlexCore processor are obtained using a timing constraint of 3 ns, which are presented here as reference and the difference of timing constraint between the two designs will have an impact on the area and power results.

10 Conclusion

The aim of this research was to design an instruction decompressor for a very long instruction word (VLIW) processor to save the memory footprint based on an optimal compression scheme. The instruction decompressor is designed and implemented in VHDL and synthesized using Cadence RTL Compiler into three different process technologies 130-nm, 90-nm, and 65-mn provided by the STMicroelectronics. We have shown that various parameters of instruction decompressor greatly impacts the overall performance of FlexCore in hardware in terms of power, area and timing. These parameters includes the formation of LUT groups, the size of LUTs and the LUT-Load instruction Interval meaning how often the LUTs needs to be updated and how many LUTs are updated through a single LUT-Load instruction. It will be interesting to compare the average toggle rate on NISA for the test vectors which are used to compute the power results for the instruction decompressor, to the average toggle rate on NISA for the benchmark applications which are used to compute the power results for the FlexCore. It can give us a better idea about the power consumption of instruction decompressor. The instruction decompressor implemented needs to be verified, for this we need to have real traces of compressed instructions produced by the compression algorithm using various benchmark applications. After getting these real traces of compressed instructions the accurate power analysis of the instruction decompressor would be possible. Later it would be interesting to see the integration of instruction decompressor into the FlexCore processor and verify the whole design using some benchmark applications.

11 Acknowledgment

This work is partially supported by the Chinese Academic of Sciences (CAS) and The World Academy of Sciences (TWAS).

12 References:

1. V. Sklyarov, I. Skliarova, A. Rjabov, A. Sudnitson, "Zynq-based System for Extracting Sorted Subsets from Large Data Sets", Informacije MIDEM-Journal of Microelectronics, Electronic Components and Materials Vol. 45, No. 2 (2015), 142 – 152.
2. Abdul Rehman Buzdar, Liguo Sun, Azhar Latif and Abdullah Buzdar, "Distance and Speed Measurements using FPGA and ASIC on a high data rate system" International Journal of Advanced Computer Science and Applications (IJACSA), 6(10), 2015, 273 – 282.
3. J. Noguera, R.M. Badia, "HW/SW codesign techniques for dynamically reconfigurable architectures" IEEE Trans. Very Large Scale Integration (VLSI) Systems, vol. 10, no. 4, pp. 399-415, Aug. 2002.
4. M. D. Edwards, et al., "Acceleration of software algorithms using hardware/software co-design techniques", J. Syst. Architecture, vol. 42, no. 9/10, pp.1997.
5. ARM Inc., [Online]. Available: http://www.arm.com/.
6. Yun Wu, J. Nunez-Yanez, R. Woods, D.S. Nikolopoulos, "Power modelling and capping for heterogeneous ARM/FPGA SoCs",IEEE International Conference on Field-Programmable Technology (FPT), Dec. 2014, pp 231-234
7. D. Gebhardt, Junbok You, K.S. Stevens, "Design of an Energy-Efficient Asynchronous NoC and Its Optimization Tools for Heterogeneous SoCs" IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems, vol. 30, no. 9, pp. 1387-1399, Sept. 2011.
8. M.D. Grammatikakis, A. Papagrigoriou, P. Petrakis, G. Kornaros, "Monitoring-Aware Virtual Platform Prototype of Heterogeneous NoC-Based Multicore SoCs",IEEE International Conference on Digital System Design (DSD), Sept. 2013, pp 497-504
9. B. Ristau, T. Limberg, G. Fettweis, "A Mapping Framework for Guided Design Space Exploration of Heterogeneous MP-SoCs",IEEE International Conference on Design, Automation and Test in Europe (DATE), March 2008, pp 780-783
10. Wu-An Kuo , TingTing Hwang, A.C.-H. Wu, "A power-driven multiplication instruction-set design method for ASIPs" IEEE Trans. Very Large Scale Integration (VLSI) Systems, vol. 14, no. 1, pp. 81-85, Jan. 2006.
11. Yosi Ben Asher, Irina Lipov, Vladislav Tartakovsky, Dror Tiv, "Using Multi-op Instructions as a Way to Generate ASIPs with Optimized Pipeline Structure",IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), May 2014, pp. 29.
12. Hong Chinh Doan, H. Javaid, S. Parameswaran, "Using Multi-op Instructions as a Way to Generate ASIPs with Optimized Pipeline Structure", IEEE International Conference on Design, Automation and Test in Europe (DATE), March 2014, pp. 1-6
13. M. Jacome and G. de Veciana "Lower bound on latency for VLiW ASIPs", Proc. of ACM/IEEE International Conference on Computer Aided Design (ICCAD), 1999
14. K. Keutzer, Malik S., and A. R. Newton, "From ASIC to ASIP: The next design discontinuity," in Proc. Int. Conf. on Computer Design, 2002, pp. 84–90.
15. M. Thuresson, M. Själander, M. Björk, L. Svensson, P. Larsson-Edefors, and P. Stenström, "FlexCore: Utilizing exposed datapath control for efficient computing," Springer J. of Signal Processing Systems, vol. 57, no. 1, pp. 5–19, Oct. 2009.
16. T. Schilling, M. Själander, and P. Larsson-Edefors, "Scheduling for an embedded architecture with a flexible datapath," in Proc. IEEE Computer Society Annual Symp. on VLSI, 2009, pp. 151–156.
17. J. Hughes, K. Jeppson, P. Larsson-Edefors, M. Sheeran, P. Stenström, and L.J. Svensson, "FlexSoC: Combining flexibility and efficiency in soc designs," in Proc. IEEE NorChip Conf., 2003.
18. M. Själander, M. Larsson-Edefors, and M. Björk, "A flexible datapath interconnect for embedded applications," in Proc IEEE Computer Society Annual Symp. on VLSI, 2007, pp. 15–20.
19. U. Jälmbrant and E. der Hagopian, "Improved configurability with FlexSoC," Msc. thesis, Chalmers University of Technology, Mar. 2009.
20. Tung Thanh Hoang, Ulf Jälmbrant, Erik der Hagopian, Kasyab P. Subramaniyan, Magnus Själander, and Per Larsson-Edefors, "Design Space Exploration for an Embedded Processor with Flexible Datapath Interconnect," in Proc. of IEEE Int. Conf. on Application-specific Systems, Architectures and Processors, 2010, pp. 55-62.
21. J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach, Elsevier Publisher Inc., 2007.
22. MIPS Technologies Inc., [Online]. Available: http://www.mips.com/.
23. Bita Gorjiara and Daniel Gajski, "Custom Processor Design Using NISC: a Case-Study on DCT Algorithm," in Workshop on Embedded Systems for Real-Time Multimedia, September 2005.
24. Bita Gorjiara, Mehrdad Reshadi, and Daniel Gajski, "Designing a Custom Architecture for DCT Using NISC Design Flow," in Asia and South Pacific Conference on Design Automation, 2006.
25. B. Gorjiara, D. Gajski, "FPGA-friendly code compression for horizontal microcoded custom IPs," Proceedings of the 2007 ACM/SIGDA 15th inter-

national symposium on Field programmable gate arrays (ISFPGA), ACM Press 2007.
26. M. Reshadi and D. Gajski, "A cycle-accurate compilation algorithm for custom pipelined datapaths," in Proc. 3rd IEEE/ACM/IFIP Int. Conf. on Hardware/ Software Codesign and System Synthesis, 2005, pp. 21–26.
27. B. Gorjiara, M. Reshadi, and D. Gajski, "Merged dictionary code compression for FPGA implementation of custom microcoded PEs," ACM Trans. Reconfigurable Technol. Syst., vol. 1, pp. 11:1–11:21, Jun. 2008.
28. Embedded Microprocessor Benchmark Consortium (EEMBC), [Online]. Available: http://www.eembc.org.
29. Muhammad Waqar Azhar, Tung Thanh Hoang, and Per Larsson-Edefors, "Cyclic Redundancy Checking (CRC) Accelerator for the FlexCore Processor," in Proc. of EUROMICRO Conf. on Digital System Design, 2010, pp. 675-680.
30. Muhammad Waqar Azhar, Magnus Själander, Hasan Ali, Akshay Vijayashekar, Tung Thanh Hoang, K. K. Ansari, and Per Larsson-Edefors, "Viterbi Accelerator for Embedded Processor Datapaths," in Proc. of IEEE Int. Conf. on Application-specific Systems, Architectures and Processors, 2012.
31. Cadence EDA Tools. [Online]. Available: www.cadence.com/en/default.aspx
32. STMicroelectronics. [Online]. Available: www.st.com/web/en/home.html
33. Kasyab P. Subramaniyan, Erik Ryman, Magnus Själander, Tung Thanh Hoang, Mafijul Md Islam, and Per Larsson-Edefors, "FlexDEF: A Toolchain Framework for Processor Development," in Proc. of IEEE Conf. on Ph.D. Research in Microelectronics and Electronics, 2011, pp. 37-40.
34. Erik Ryman, Kasyab P. Subramaniyan, Tung Thanh Hoang, Mafijul Md Islam, Magnus Själander, and Per Larsson-Edefors, "FlexTools: Design Space Exploration Tool Chain from C to Physical Implementation," in Proc. of the Fifth Annual Cadence User Conf., 2010.
35. M. Thuresson, M. Själander, L. Svensson, and P. Stenstrom, "A flexible code compression scheme using partitioned look-up tables," in Proc. Int. Conf. on High Performance Embedded Architectures and Compilers, 2009, pp. 95–109.

Arrived: 26. 05. 2015
Accepted: 24. 12. 2015