Design Automation Model for Application-Specific Processors on Reconfigurable Fabric

B. Kurumahmut, G. Kabukcu, R. Ghamari and A. Yurdakul
CASLAB, Computer Engineering Department, Boğaziçi University, İstanbul, Turkey
{roza.ghamari, yurdakul}@boun.edu.tr

***Abstract*—The process of embedded system design on reconfigurable architectures needs smart solutions to reduce development life-cycle and to use resources efficiently at run-time. Current solutions are insufficient to enable the embedded system designer to reflect the flexibility that a reconfigurable architecture can offer. Some of the basic problems are lack of flexible operator definitions, very detailed hardware abstraction procedures, a few or no constraints for tasks or loops at the high-level of design abstraction.**

In this paper, we propose a new model for automated design of application-specific processors in run-time reconfigurable architectures, solving the aforementioned inefficiency problems. Based on the proposal, a design language, a framework and a compiler have also been developed.

I. Introduction

In the embedded systems arena, there are hundreds of microcontrollers from different manufacturers. However, it is usually impossible to find a microcontroller that exactly meets all requirements of an embedded system. As a result customizable processors [8], [22] have started to appear. On the other hand, reconfigurable devices have a significant place in embedded design domain. Consequently, soft-processors [1], polymorphic processors [25], [23], [11] are proposed. These processors can be tailored to meet the exact requirements of an embedded system. Some of them extends the instruction set while preserving the basic instruction set [8], [22], [1] while newer ones modify the instruction set by removing unused instructions and/or introducing new instructions [25], [23] or design the complete application-specific instruction set [11]. Most of them support the customizations to be done prior to execution while the newer ones support run-time customization of the instruction set.

There are also design entry tools [8],[26],[21],[19]. The following discussion can be carried out for those tools 1) The design tools require detailed hardware definition for the new instructions. Some tools invoke them with function calls which makes embedded programming a confusing issue. ADRES [11] uses a 32-bit three-input/output functional template to define all operations since ADRES is a coarse grain array. As a result, the resources of ADRES are underutilized for simple operations. 2) They use programming languages (like C, C++) as they are or extend them. Usually retargetting of the compiler has to be done manually. In some tools, the designer designs the instructions and determines very low details like opcode, operand fields. DRESC provides a parametic template for instruction generation since the underlying architecture, ADRES, is fixed. However, in all of these tools the instruction set is fixed by the user. DRESC allows architecture exploration but instruction generation is not mentioned. 3) Retargettable compilers map the built-in operations of the language to the built-in operators of the underlying processor which is assumed to be static during runtime. For example, if the underlying processor is a scalar one, then "+" operation of C will be mapped to the ALU that makes scalar addition. For a vector addition, a new instruction has to be generated and this must be invoked via a function call. When the underlying processor is a runtime reconfigurable one, these tools cannot benefit from this utility of the device. For example, the designer cannot set "+" as a scalar adder in one loop and as a vector adder in another loop. 4) They do not enable the user to define timing and resource constraints. If the design does not meet a constraint, then the overall design process has to be revised. Today, design tools for programmable processors [7] support code optimization according to user constraints. However, the tools for reconfigurable ones do not have such an option.

Based on these observations we propose that a new model and a language must be used for embedded systems on reconfigurable systems. The issues that necessitates a new language are explained in the following section. Then we explain our model, RH(+), which is general enough to handle the run-time reconfigurable processors as well as other types of customizable soft processors and hard processors (Section III). Based on this model, we developed a new language, LRH(+) which allows the embedded system to be designed with a single language, a framework, FRH(+) which satisfies the basics set by our model, and a compiler which retargets itself automatically by using the templates generated by the instruction set genertation tool (Section IV). A design example is presented in Section V. In the last section, we explain the current status of our project and summarize our future work

II. Languages for Customizable Processors

Traditionally, programming languages, like C, C++, Java are selected for developing embedded applications. They are also extended to support hardware elements of a system. SpecC, SystemC, Handel-C and Impulse-C are some of the extended-C languages [20], [14]. An embedded design has to be specified to exploit the properties of the target architectures. Today, the most popular extended-programming-language for hardware design is SystemC [14]. There are

This work has been fully supported by the Turkish Foundation of Science and Technology, under Research Project Program (pr. no:104E038)

SystemC-based approaches use a protoyping tool that can estimate area, execution time and reconfiguration overhead [13]. Moreover reconfiguration can be automatically triggered. However, the user-defined system specification constraints like resource and timing constraints, in SystemC [10]. Besides, it does not allow processor customization [18]. Another C-variant language, which is includes design of both hardware and software portions, is SpecC [20]. However, like SystemC, SpecC does not have the features like mapping, estimation and system constraint definition. Impulse-C and Handel-C use C-to-gates tools for architecture mapping to FPGAs. Although they are capable of generating hardware using some units automatically, the software code needs optimizations after compilation [2]. Some low-level timing optimizations are available in Handel-C; however, there is no optimization feature in Impulse-C [2]. Moreover, the operations are predefined in Handel-C; therefore, the reconfigurable system designer encounters with limitations during application development. Extension of programming languages requires modifications in standard compilers inorder to be able to separate the hardware and the software.

The modeling languages have also been proposed for modeling embedded systems. SDL and UML [5], are some of the well-known examples. However, their cores are inappropriate to use them as solutions for automatic synthesis. Thus, the community has proposed extensions to adapt them to the applications on reconfigurable hardware xUML [17], xtUML [12]. GASPARD is a platform that inrtoduces synthesis and reconfiguration to a UML-based model MARTE [16] but it cannot be used to generate custom reconfigurable processors. Hence the modeling languages must only be used as representation languages to express the design visually.

In the literature, extending the standard tools is called as *retargetting*. To retarget existing tool suite, architecture description languages (ADL) are used . However, they need low-level details which are irrelevant for software development level. For example MIMOLA models target hardware in a similar manner to VHDL. Thus, it explicitly [6] requires definition of hardware modules with their behavioral algorithm and detailed interconnection scheme between modules. MIMOLA has problems with complex instructions because it is difficult to extract complex instructions from MIMOLA descriptions. As a result, the quality of the produced code is low [9]. nML is another example of formalism which generates description based on both structural and behavioral model, and is applicable for *retargetting* code generation [6]. The main disadvantage of nML is that it requires low-level details to define target architecture. These are behavior, assembly language mnemonic, and binary code of the instructions which are constructed by using predefined operators. LISA [26] is an enhanced ADL compared to nML. However, it has the disadvantage that it requires behavioral description of the extended instructions as nML. In both nML and LISA, new operators (instructions) can be created in two ways. Firstly, predefined operators (+, -, *, etc.) are used to create new operators. Secondly, these created operators can be used to create more complex operators. In an application, both types of operators for the same functionality can be created. Software designer can choose the one providing more efficient solution. However, if software designer uses nML or LISA, he will have a huge work overhead while switching between these two types of implementation. The basic reason is that nML and LISA force the user to give different operator names for each type of implementation. Thus, designer must scan all lines in software, find each references to the operator, and replace it with new name. DRESC uses a parametric template which can be used by the designer to decide on the instructions. Instruction level parallelism is automatically established by DRESC.

III. RH(+) Model

To solve the above-mentioned problems of current design languages, we propose a model, RH(+), for embedded system design on run-time reconfigurable architectures like FPGAs that support dynamic reconfiguration [24]. RH(+) stands for hardware/software co-design on reconfigurable hardware. Based on this model, we propose a new language, LRH(+) which utilizes run-time processor customization and application development in a single language.

A. Abstraction of Low-Level Details for Instruction Set

For embedded system on reconfigurable hardware, we believe that users should not deal with the details about instruction due to the flexibility of the target hardware. In RH(+), smallest instruction corresponds to an operator. For each operator, it must be sufficient for the designer to enter the number of inputs and outputs of the operator. For example, assume two instructions: A and B. Instruction A might have 2 inputs, and 1 output (e.g. scalar addition) and instruction B might have 8 inputs, and 4 outputs (e.g. matrix multiplication of two 4-by-4 matrices). Operators can be combined to generate complex instructions. Complex instruction generation can be done either by the user or left to the design automation tool that generates the optimum instruction set for the application and its imposed constraints.

B. Flexible Operator Definition

Reconfigurable hardware comes in with different on chip resources. This means that fixing built-in operators in a language for customizable soft processors will hinder the flexibility offered by reconfigurable hardware. Moreover, different applications require different operations. For example, in FIR filtering, there is vector multiplication (inner product). On the soft-processor, the hardware units for this application can be one of the following: 1) a multiplier and an ALU, 2) a multiply-accumulate unit and an ALU, 3) an inner product. Note that in all of them, there is no operation requirement like subtraction or shifting. Therefore the instruction set of the soft-processor will consist of only one of the following instruction set groups 1) MULTIPLY, ADD, INCREMENT, COMPARE, 2) MULTIPLY_AND_ACCUMULATE, INCREMENT, COMPARE, 3) DOT_PRODUCT.

We propose that the designer should determine how the built-in operators will be. Therefore, while developing the application, the user has the freedom to define the operators. The user picks one of the methods while defining an operator: 1)Select an operator from an already-designed-operators library or

invoke a module generating function from the related library. 2)Define the operator with an HDL that the user selects. RH(+) treats HDL-defined modules as black boxes. Therefore it can handle different types of HDLs simultaneously. It uses interface signals to connect the operator to the data path. 3) Write the behavior of the operator with the same language that is used for developing an application. LRH(+) is the language that we propose as a solution. In that case, the basic operations in the operator definition must be introduced to the system by using the methods described above.

The operators can be defined globally as well as locally for each loop or function. This flexibility will give the chance for run-time reconfiguration of the soft-processor, i.e. for each function and loop, the instruction-set and data-path will be reconfigured if it is desired by the user. Each operator must have a name specified by the user. However, the same name can be given to operators of similar nature so that it will make code development much easier. This can be explained with a simple example: Assume that in a function, there is an addition operation acting on scalar variables. Let its name be "+" . Assume that in another function there are two vectors of the same size and we want to realize a vector addition. We can also set its name as "+". It is the job of the compiler to map the same "+" to the related hardware unit. Actually, the front end compiler must attach an attribute to the intermediate format of the user code so that the back end compiler resolves two different "+"s in two different functions. An example is shown in Section V.

Note that our model leaves the decision about some parameters like the operator's opcode, behaviour of the operator, assembly syntax of the operator, register files at which operands are located, wires defining input and output nets, and physical addresses for memory locations to the compiler. Further details about operator definition are explained in Section IV-B.

C. Flexible Data Types

In software development, the application developers use variables that have different size. Usually they use keywords like char, int, long to identify the size. However, these keywords might correspond to different bitsize and format in each data type might change in each processor. Moreover for reconfigurable systems, having custom bit-lengths for each variable causes low utilization of the underlying hardware. For example, assume that we declare a counter which has to set a flag whenever it reaches 5. If we use 8-bit Freescale processor and use *unsigned char* data type for each variable, then we have to use either two 8-bit memory locations. Another solution might be usage of only one *unsigned char* variable and partition it so as to access flag and counter in the same variable. This shows that we will use either 8-bit or 16-bit memory locations to manipulate 4 bits (1 bit for flag, 3 bits for maximum value of the counter).

We claim that the user must have the freedom to enter the size of each variable separately, in bits. Moreover, designers dealing with processor customization must consider only the size and format of the variable types, because instruction generation must be carried out by the compiler, which either 1) maps operations of the same type to the same processing unit or 2) maps operations of the same type to different processing units or 3) sets the reconfiguration parameter so that the hardware will be reconfigured for different operation types or operations with different variable size (spatio-temporal mapping). Reconfiguration is a user-specified compiler directive and once it is set, the decision about reconfiguration instances is given by the design evaluation tool. The compiler also decides on the size of the datapath.

D. Configurability and Self-Retargettable Compiler

Ideally, a reconfigurable processor must have the chance of being reconfigured prior to execution of each instruction. However, reconfiguration is a slower process than computation. If the size of the target architecture allows, compile-time configuration might be sufficient. Otherwise, run-time reconfiguration must be used only when it is necessary. As a result, spatio-temporal scheduling tools must involve in for the optimized area-performance objectives of the design.

Since target architecture is fully configurable, initially there must be no instructions on the processor. The instuction selection tool must identify all the necessary instructions that will be used in the application, i.e. a partitioning must be realized on the design representation graph. Therefore, we claim that current instruction selection tools need to be improved because they only extract the critical instructions. Other nodes on the design representation graph are realized by general purpose instructions of the processor.

After instruction selection, a template for each instruction must be automatically generated so that the compiler will automatically retargets itself for the new instruction set. Hence, the designer will not spend time in retargetting the compiler.

E. Constraints Setting

An embedded design has to meet several constraints like area, time, energy, etc. Therefore any embedded application design tool must support entry of the global and local constraints. Since instrcuitons are generated specific to the application, behavioral synthesis tools can be used to satisfy both global and local constraints which appear at functions, loops, conditional branches and even basic blocks (i.e. remaining statement blocks)

IV. Implementation of RH(+)

Base technology used in the implementation is eXtensible Markup Language (XML) due to its capabilities that are noticed in many works in the literature [3], [4]. XML provides us with storing data structurally. We have implemented LRH(+) by using XML. Also, the templates, the operators, the board definition, the application, the graphs, the constraints, the IR of the application, and the map of the embedded software to the hardware are stored in XML format. Because, outputs of RH(+) IDE is in XML format, it can moved to another application easily via deserialization of them. Besides, thanks to eXtensible Stylesheet Language (XSL), data transformation from XML to the other representations can be achieved via a set of XSL translation rules.

A. FRH(+)

FRH(+) is the implementation of the RH(+) model. In the first level, the embedded system designer must define three

TABLE I
SYNTAX OF STATEMENTS AND CONDITIONS

Tag	Syntax 1	Syntax 2
Statement	c = a + b;	.=(c, .+(a, b));
	d = func1(c);	.=(d, .func1(c));
	e = d/c*4;	.=(e, .*(./(d, c), 4));
Condition	(a > b) && (c < d)	.&&(.>(a,b), .<(c,d))

application-specific environments: Templates, Board Support Package (BSP), and Operator Definitions (OPDEF). They include hardware abstraction required for RH(+) explained in Section III. Based on these environments, processor configuration data is generated automatically. In the second level, the application with LRH(+) language by taking configuration data into consideration. Third level meets "efficient mapping of the designed system to the reconfigurable architecture requirement" of RH(+) . At this level, the front end compiler generates an enhanced and optimized Control Data Flow Graph (CDFG) . The user-defined constraints and reconfigurability parameters are used to generate an optimal CDFG. The CDFG is in XML format and used during instruction selection and parse tree generation. In the fourth level, a fuzzy expert system is executed on the CDFG to select the instructions. Behavioral synthesis is applied on each selected intruction so as to generate control and data path for each instruction. Instruction set generator determines the instruction codeword size and the register file size required for each instruction. It also prepares the templates for the compiler so that the full assembly code is generated in the final level.

B. LRH(+)

LRH(+) is a language which facilitates sequential and concurrent programming in the same environment. It includes both traditional programming constructs and the constructs special to Custom Hardware (CH) which utilizes reconfigurable and nonreconfigurable parts at the same time on the same device. In addition, it includes structures proposed in FRH(+).

There are two data-types in LRH(+). These are *general* and *array*. The parameters to define a variable with *general* data type are variable name, size in terms of bits, location and initial value of the variable. Location information is retrieved from Templates or definitions at Application Development level of the FRH(+) since embedded system designer can also add location definitions at application level. *Array* data-type requires size of array and initial values of its elements whose sizes are also given in terms of bits. It should be noted that new data-types can be generated by using these data-types. Data types like char, int, long can easily be with *general* type.

In LRH(+), we propose a syntax which is different than traditional languages used in embedded systems programming. LRH(+) compiler also supports traditional syntax indicated as Syntax 1 in Table I. However, we advise the designer to use the new syntax shown as Syntax 2 in Table I, because this syntax has an advantage that each operator can have more than two operands. Since the underlying processor is customizable, it is not unlikely to have a operator with more than two inputs on the data-path.

In LRH(+), each statement in the code has a chance to operate concurrently. Data dependency between the statements dictates the sequential behavior between the statements. However, "?" operator can be used to impose a sequential behavior between two data-independent statements. For example, consider program segment in Figure 1(a). Assuming that each statement has identical execution times, the execution order of the segment is found as it is shown in Figure 1(a). In this figure, "s" keyword represents execution order. However, if a sequential behavior is desired by the user then "?" operator must be prepended to line l4, shown as modified code in Figure 1(b). As a result, execution becomes sequential

Program Segment	Execution order
l1: .=(c,.+(a,b));	s1: l1, l4
l2: .=(d,.*(c,e));	s2: l2, l5
l3: .=(f,.+(d,k));	s3: l3
l4: .=(g,.-(a,b));	
l5: .=(h,.+(g,i));	

(a)

Program Segment	Execution order
l1: .=(c,.+(a,b));	s1: l1
l2: .=(d,.*(c,e));	s2: l2
l3: .=(f,.+(d,k));	s3: l3
l4: ?.=(g,.-(a,b));	s4: l4
l5: .=(h,.+(g,i));	s5: l5

(c)

Fig. 1. Data dependency graph for delaying example

C. Constraint and Configurability Setting

In our implementation, configurability is a global option. Global and function level constraints can be processed.

D. Instruction Selection and Generation

Instruction selection is done by a fuzzy expert system based on FuzzyClips [15]. By using the CDFG, it extracts recurring subgraphs and sets of subgraphs that can execute concurrently. Then using behavioral synthesis as a subroutine, it evaluates the area and time costs of operations, subgraphs and concurrent candidates. This information is passed to the fuzzy expert system and the set of selected instructions are passed to the graph evaluator. The best set is selected based on the constraints and the adaptive threshold. The selected instructions are marked on the CDFG and the procedure repeats itself from the extraction of subgraphs phase until all nodes are marked.

After the instructions are selected, the CDFG is modified so that each recurrent subgraph or each set of concurrent subgraphs selected as an instruction is compressed to a node. The related hardware modules are generated and a node-based decision is taken for reconfiguration by using data dependencies and constraints.

A register file is generated based on the assumption that each instruction fetches the operands from a register file and writes the result to the register file. The basic reason for this assumption is that RAM blocks in FPGA support single read-write in each access. Register file can contain registers with different widths.

An instruction template is generated for each instruction. The template generator assumes that the operands have to be moved from the memory to the registers and the output can reside in register(s). For example, consider the statement .=(t4,.+(t4,t3)) in Figure 4. It is simply the addition of two numbers (t4 and t3) and the result is written back in t4. Assume that t4 and t3 are defined as 8 bit scalars using General data-type. Assume that a scalar adder and store instruction is

```
% scalar addition .+(t4,t3)
MOV RG8_0 op1
MOV RG8_1 op2
PLUSG8G8G8_0 RG8_0 RG8_0 RG8_1
% scalar store .=(t4,...)
MOV op2 RG8_0
```

Fig. 2. A sample template generated by the compiler for add

selected as an instruction for this operation. Then the related template is generated as shown in Figure 2.

MOV is a keyword that realizes the movement from memory to the register file. There are as many MOVs as the number of inputs. Register names starts with either RG or RA. RG registers stand for general register type. These are in the block RAMs of the FPGA. RA registers represent array type registers. These registers are distributed in the FPGA. A bulk move from distributed memory to distributed register file is realized when RA is the target register. The number following RG or RA represents the wordlength of the register. The numbers separated by "_" represent the instance of the register. For example, there are two RG8 registers in the sample template, namely RG8_0 and RG8_1. The operands of the instruction starts with op1, op2,... and they are as many as inputs of the instruction. Each instruction starts with its node name. It should be noted that the compressed nodes are given special names by the instruction generation module. If an operation does not contain alphabetical characters, special names are given in a way not to coincide with other instruction names. For example, PLUS is given for "+" operand. It should be noted that the compressed nodes are given special names by the instruction generation module. Type and size of the registers are appended to the node name prevent misinterpretation of the instruction.

Note that the register and operand instances, and MOV instructions are dummy, i.e. they are replaced with the correct instances by the compiler. Actually, the instances of all registers and operands start with 0 in a template of an instruction.

Based on the number of instructions and number of inputs, the machine code of instructions are automatically generated with a predefined format. In addition to the instructions covered in the CDFG, instructions like MOV, RECONFIGURE, NOP, STOP are always included to the set.

E. Compiler

The compiler firstly converts the modified CDFG to a parse tree. Then it calculates the minimum size of the register file by using the templates. This size can be increased by the compiler if the register size does not suffice during code generation. Note that reduction is always possible by tightening area constraints.

During the traversal of the parse tree two lists for registers are updated to monitor status of the registers. The first list keeps track of busy registers. A busy register means a register which is occupied at that time. The second list keeps track of the operand which makes the register busy. The compiler generates the assembly code for each node in the parse tree if its siblings have already been processed. Therefore the output of each sibling is searched in the second list so as to see whether it appears. If it is the case, then the related MOV instruction in the template is skipped and the busy flag in the first list is set. Otherwise, an empty register is searched in the first list so as to replace the related dummy register with the empty register. Then the status of the selected register is changed from empty to busy and the second list is updated with the output of the sibling. As soon as the related code is generated, the status of the registers using the outputs of the siblings are reset to free. There exists no case where all registers are busy because the size of register file is calculated prior to compilation.

After the assembly code is generated, then machine code is generated by using the instruction machine code explained in Section IV-D

TABLE II
OPERATION DEFINED FOR MOTION INTENSITY CALCULATOR EXAMPLE

OP Name	Input1 Type	Input2 Type	Output Type	Description
>=	General	General	General	Bigger or Equal
<	General	General	General	Smaller
+	General	General	General	Integer Addition
*	General	General	General	Integer Multiplication
/	General	General	General	Integer Division
root	General	-	General	Root
++	General	-	General	Increment
square	Array	-	Array	Square amount of each index in array
+	Array	Array	Array	Addition of each index in 2 arrays
[]	Array	General	General	Amount of the special index in a array

V. ILLUSTRATIVE EXAMPLE

In this section we consider a motion intensity calculator and implement it using RH(+)IDE. In BSP, we define two RAMs to store the general and array variables separately, and a serial bus to put the MPEG-7 representative of each block motion block to the array memory. For this application, we defined the operators shown in Table II in OPDEF. Six general-type (average, intensity, index, t3, t4, 12) and five array-type (motionVectorX, motionVectorY, T1, T2, T) variables are defined.

Figure 3 demonstrates the code we write in RH(+). This function gets two arrays, *motionVectorX* and *motionVectorY* with length of 12 and then calculates the square of each item in these arrays using "square" operation and the results are written to *T1* and *T2* arrays, respectively. Next, in a loop, each item in *T1* and *T2* are added, square root of the sum is calculated and the result is added to *t3*. After the loop, *average* is obtained by dividing *t3* to length of arrays, i.e. 12. At the same expression the two square arrays are added and saved in array *T*. Then the root of each value in *T* is assigned to *t3* and then inside an IF statement, it is checked whether the amount of *t3* is bigger than or equal to *average*, by using ">=" operator. If it is bigger then *t3* is added to *t4*. Afterwards, the intensity variable is calculated as the ratio of *t4* to *12* (length of arrays).

The produced .cdfg file of this design is about 2000 lines and it is inappropriate to demonstrate this file in this paper. However, we try to show a part of this file in Figure 4. This figure indicates the generated code for expressions *".+(t4,t3);"* and *".+(T1,T2);"*. The compiler also generates an assembly file which consists of 32 lines of code. We do not put this generated code here because of space restrictions.

```
Function;MotinIntensityCalculator:F1;F1;
Expressions;exp1;exp1;
    .=(T1,.square(motionVectorX));
    .=(T2,.square(motionVectorY));
ExpressionsEnd
Loop;L1;L1;
    .<(index,12);
    Expressions;exp2;exp2;
        .=(t3,.+(t3,.root(.+(.[](T1,index),.[](T2,index)))));
        .=(index,.++(index));
    ExpressionsEnd
LoopEnd
Expressions;exp3;exp3;
    .=(average,./(t3,12));
    .=(T,.+(T1,T2));
ExpressionsEnd
Loop;L2;L2;
    .<(index,12);
    Expressions;exp4;exp4;
        .=(t3,.root(.[](T,index)));
    ExpressionsEnd
    If;IF1;IF1;
        .>=(t3,average);
        Expressions;exp5;exp5;
            .=(t4,.+(t4,t3));          //selected statement
        ExpressionsEnd
    IfEnd
LoopEnd
Expressions;exp6;exp6;
    .=(intensity,./(t4,12));
ExpressionsEnd
FunctionEnd
```

Fig. 3. LRH(+) code of Motion Intensity Calculator

VI. Conclusions and Future Work

In this paper, we have proposed a model, RH(+), for embedded system development on run-time reconfigurable hardware. Based on this model, we have implemented a design environment which includes a framework FRH(+), a new design language LRH(+), an instruction selection and generation tool and a compiler. The LRH(+) language enables the user to develop the application with flexible data types and operators. It generates an extended CDFG which is an intermediate representation in XML format. The instruction selection/generation toolbox generates the the complete instruction set from the CDFG. The compiler automatically retargets itself for the new instruction set.

As the current work we try to implement a runtime reconfiguration of the processor, i.e. to activate RECONFIGURE instruction. Another issue is the automatic generation of the whole processor with its control unit and data path being connected to. Finally, we are dealing with the integration of these tools in the same development environment.

```
-<component.type>                    -<component.type>
  <!–specifications–>                  <!–specifications–>
  <Type>General</Type>                 <Type>Array</Type>
  - <GeneralTypeData>                  - <GeneralTypeData>
    <Length>8</Length>                   <Length>8</Length>
    <Value>0</Value>                     <Value>0</Value>
    <Location>MEM1</Location>            <Location>MEM1</Location>
    <refName>t4</refName>                <refName>T1</refName>
  </GeneralTypeData>                   </GeneralTypeData>
- </component.type>                   - </component.type>
**********************               **********************
- <component.type>                   - <component.type>
  <!–specifications–>                  <!–specifications–>
  <Type>General</Type>                 <Type>Array</Type>
  - <GeneralTypeData>                  - <GeneralTypeData>
    <Length>8</Length>                   <Length>8</Length>
    <Value>0</Value>                     <Value>0</Value>
    <Location>MEM1</Location>            <Location>MEM1</Location>
    <refName>t3</refName>                <refName>T2</refName>
  </GeneralTypeData>                   </GeneralTypeData>
- </component.type>                   - </component.type>
**********************               **********************
- <SubExpression>                    - <SubExpression>
  <Number>754</Number>                 <Number>745</Number>
  <Name>SE85</Name>                    <Name>SE660</Name>
  <!–specifications –>                 <!–specifications –>
  <OpName>+</OpName>                   <OpName>+</OpName>
  - <OperandType>                      - <OperandType>
    <Location>MEM1</Location>            <Location>MEM1</Location>
    <Type>General</Type>                 <Type>Array</Type>
  </OperandType>                       </OperandType>
  - <OperandNames>                     - <OperandNames>
    <string>+</string>                   <string>+</string>
    <string>t4</string>                  <string>T1</string>
    <string>t3</string>                  <string>T2</string>
  </OperandNames>                      </OperandNames>
</SubExpression>                     </SubExpression>
```

Fig. 4. Intermediate CDFG code generated for scalar and array adders

References

[1] Altera Corporation, "Nios embedded processor," 2007.

[2] J. A. Bower and et.al., "A java-based system for fpga programming," *FPGA World Conference*, 2008.

[3] W. O. Cesario and et.al., "Colif: A design representation for application-specific multiprocessor socs," *IEEE Des. Test*, vol. 18, pp. 8–20, 2001.

[4] F. P. Coyle and M. A. Thornton, "From UML to HDL: A model driven architectural approach to hardware-software co-design," *ISNG*, pp. 88–93, 2005.

[5] R. Damasevicius, "A subset-based comparison of main design languages," *Inf. Tech. Cont., Technologija*, vol. 1, pp. 49–56, 2004.

[6] A. Fauth, J. V. Praet, and M. Freericks, "Describing instruction set processors using nML," *EDTC'95*, pp. 503–507, 1995.

[7] Freescale Semiconductor Inc., "CodeWarrior development tools," 2007.

[8] R. E. Gonzalez, "Xtensa: A configurable and extensible processor," *IEEE Micro*, vol. 20, pp. 60–70, 2000.

[9] A. Halambi and et.al., "A customizable compiler framework for embedded systems," *SCOPES*, March 2001.

[10] T. Kambe, A. Yamada, and M. Yamaguchi, "Trend of system level design and an approach to c-based design," *Microelec. Jour.*, vol. 33, pp. 875–880, 2002.

[11] B. Mei and et.al, "Architecture exploration for a reconfigurable architecture template," *IEEE Des. Test*, vol. 22, pp. 90–101, 2005.

[12] S. J. Mellor, "Executable and translatable UML," 2003.

[13] K. M. Nikolaos S. Voros.

[14] Open SystemC Initiative, "www.systemc.org," 2009.

[15] R. Orchard, "Fuzzyclips version 6.04a Users' guide," 1998.

[16] I. R. Quadri, S. Meftali, and J.-L. Dekeyser, "High level modeling of dynamic reconfigurable fpgas," *Int. Jour. of Reconf. Comp.*, 2009.

[17] C. Raistrick, P. Francis, and J. Wright, *Model Driven Architecture with Executable UML(TM)*, New York, NY, USA, 2004.

[18] O. Schliebusch and et.al., "Architecture implementation using the machine description language lisa," *ASP-DAC'02*, pp. 239–244, 2002.

[19] Silicon Hive, "HiveCC software development kit v3.5," 2009.

[20] SpecC Technology Open Consortium, "Specc language reference manual, version 2.0," 2002.

[21] Synfora Inc., "www.synfora.com," 2009.

[22] Target Compiler Tech., "CHESS/CHECKERS:A retargetable tool-suite for embedded processors," Belgium, Tech. Rep., June 2003.

[23] S. Vassiliadis and et.al., "The MOLEN polymorphic processor," *IEEE Trans. Comp.*, vol. 53, pp. 1363–1375, 2004.

[24] Xilinx Inc., "www.xilinx.com," 2007.

[25] P. Yiannacouras, J. Rose, and J. G. Steffan, "The microarchitecture of fpga-based soft processors," *CASES'05'*, pp. 202–212, 2005.

[26] V. Zivojnovic, S. Pees, and H. Meyr, "LISA-machine description language and generic machine model for HW/SW co-design," *IEEE Workshop on VLSI Signal Processing IX*, pp. 127–136, 1996.