Development of a tool for generating customized processors for specific applications

Steven Ávila Ardón
Computer Engineering Academic Area
Instituto Tecnológico de Costa Rica
Email: sfaa03@gmail.com

Jonathan Rojas González
Computer Engineering Academic Area
Instituto Tecnológico de Costa Rica
Email: jonnaro14@gmail.com

Andrés Stephen Cantillano
Computer Engineering Academic Area
Instituto Tecnológico de Costa Rica
Email: andstecan@gmail.com

***Abstract*—In this paper, it is presented the details of the tool developed which allows generating customized processor for specific C application. They were obtained several improvements regarding the use of resources in comparison with using the full base processor. For proving the resulting processors it was used the CHStone benchmark.**

I. INTRODUCTION

Nowadays, technology is changing and growing faster than we can imagine. Companies are looking for solutions because they need to accomplish goals in the best way and with the lower amount of resources. Application Specific Processor (ASP) was born with the main idea of developing a solution that can create a processor just with the unique and necessary characteristics in order to execute an input program written in C code.

Extensible processors allow the user to augment a base processor, typically an in-order RISC or VLIW, with application specific custom instruction set extensions (ISEs). ISEs may be realized in ASIC technology, along with the processor, or synthesized on a closely-coupled reconfigurable datapath. FPGA-based soft processors, e.g., Altera Nios II, are extensible as well [1].

The high level synthesis is a topic that has a lot of history but right now with embedded systems it has become stronger. Efficiency is a key point when designing new systems because in development and design every detail counts. The good use of methodologies leads to a better product with greater efficiency and power from the point of view of effectiveness. Sometimes high level synthesis is a natural module for distributed parallel computing. [2].

In [3], it is mentioned they provide a new degree of flexibility for soft processor design by customizing them to support a reduced, selected set of instructions out of the set of instructions in the Instruction Set Architecture (ISA). They implemented customized processors using C code based on the SPIM processor simulator of the MIPS ISA.

Today, the synthesis of memory is becoming the main bottleneck for the generation of efficient hardware accelerators [4]. However, in a hardware implementation, each use of a shared memory resource can be expensive. In [5] is mentioned the headache that is the implementation of memory in any system and obviously the implementation in the synthesis of high level is not the exception.There are some references to previous works that have done something similar to what was done in this project.

The rest of the paper is going to have a description of the system functions, explaining in general terms, how the main components do their job. After that, in the results section, there will be some examples of the system execution over some input programs, with a discussion of how much optimization has been done by system over the architecture and his impact and relevance in some applications. At the end, there will be some conclusions and recommendations for future research in this area.

II. DEVELOPED SYSTEM

In Fig. 1 it is presented a high level diagram of the system. As shown, the implemented system consists of the master and three other modules: Analyzer, Linker and Generator. Below, it is described the function of each of these modules.

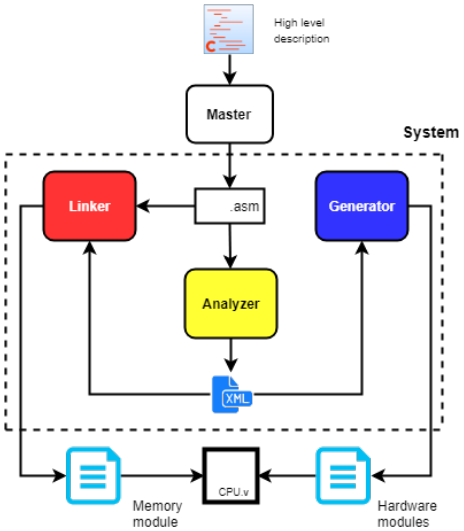

Fig. 1. System's high level diagram.

A. ANALYZER

This module is responsible for determining all the characteristics of the customized processor that will be implemented for the application of interest. These characteristics refer to the details of the different elements of the CPU such as the registers bank, the ALU, and the addressing levels the application require to be executed.

To do this, this module takes as input the assembly code of the application, which is obtained from the work done by the Linker module, and performs an analysis on it. The information obtained by this stage is required by the Linker and Generator modules, so the Analyzer gathers all this data in an XML file in order to serve as an interface between the other modules.

Once the entire file has been read the module goes through the entire code, instruction by instruction, and determines the number of registers that the program uses, and calculates the bits that the registers bank requires to be able to address them. Moreover, while the code is being scanned, the addressing levels used and the operations of the ALU are also identified. To determine the operations of the ALU that are necessary, the operations required by each program instruction are reviewed. It is also important to mention that RISC-V has several pseudo-instructions which are translated into one or more of the real architecture instructions [6]. For example, the *move* (*mv*) instruction actually consists in placing in a destination register the result of adding the value of another record with the immediate 0, that is, the *addi* instruction is used.

Finally, the program displays the data memory options that can be used in the range between 32KB and 128KB on the screen. The user must enter the option that he considers most appropriate according to the application he will execute.

B. LINKER

The Linker module has been designed in order to generate the instruction memory for customized architecture from the assembly code who was made before. In order to do that, the Linker module creates the object file from the assembler with the toolchain help. This file is important because with him, the system generates 2 files; the dump file and a text file with the machine code in hexadecimal. The dump file is used to get the initial address of the program counter. The file with the machine code is optimized to reduce the memory consumption, and later the system prepares the resulting file to be ready to be loaded into the memory.

As part of the process described before, the system do an analysis of the file with the machine code and the dump file, to define the size of the program. This information is used to obtain the parameters that determines the memory size. This data is stored in a file that is included in the Verilog description of the main module of the CPU. This file is shown in Listing.5. The parameters who defines the PC size, is stored in a XML with the same format of Fig.2. At end, reorganizes the temporal files.

```
1 `ifndef _ROMmemoryparam_vh
2 `define _ROMmemoryparam_vh
3 `define ROM_ADDR_BITS 11
4 `define ROM_ADDR_START_BITS 15
5 `define BEGIN_ADDR_ROM_PROGRAM 32'd32768
6 `define END_ADDR_ROM_PROGRAM 32'd34815
7 `define STACKPOINTER 32'd32768
8 `define PROGRAMSTARTADDRESSPC 32'd32926
9 `endif
```

Listing 1. Header file to memory ROM definiton.

C. GENERATOR

The research is looking for created a specific processor. Custom instruction set extensions (ISEs) are added to an extensible base processor to provide application-specific functionality at a low cost [1]. This generator can provided specific modules using an specific information from the xml file.

This module is responsible for generating each of the specific hardware modules. By means of a communication interface with the Analyzer the meta data with the necessary information is obtained. The generator has functions in the programming language Python that creates each specific module according to the XML file provided by the analyzer. In the Fig. 2, the communication between the Analyzer and the generator can be better observed.

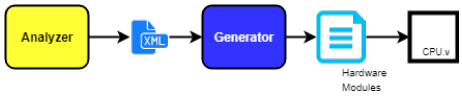

Fig. 2. Communication between Analyzer and Generator.

Specifically, the XML file comes the information shown in the code 2. Through this information the generator creates the following modules:

- Bank Registers
- Memory RAM
- Arithmetic Unit Logic (ALU)

```xml
<?xml version="1.0"?>
<CPU>
    <Registers>
        <Amount>7</Amount>
        <Bits>32</Bits>
    </Registers>

    <Data_Memory>
        <Option>16</Option>
    </Data_Memory>

    <Operations>
        <Operation>ADD</Operation>
        <Operation>SUB</Operation
        <Operation>AND</Operation>
    </Operations>

    <Addressing_levels>
        <Level>8</Level>
        <Level>32</Level>
    </Addressing_levels>
</CPU>
```

Listing 2. XML file

For example, with the information in code 2 the Generator creates a CPU with seven registers, a memory with a size of 64KB, and an ALU just with ADD, AND and SUB operations. The memory has the option of creating three memories with different sizes. The table I explains each memory size.

TABLE I
MEMORIES RAM SIZE.

Option	Normal size	Size (KB)
15	32 768	32
16	65 536	64
17	131 072	128

A class, developed in the programming language Python, creates the register bank. Based on a template the module with the necessary maximum registers is created in a dynamic way. Then, in a file the parameter for the memory RAM is specified. As we can see in the code 3 the file define the size for the memory RAM. In this case the size is 15 and the table I specified a size of 32KB.

```
`ifndef _RAMmemoryparam_vh_
`define _RAMmemoryparam_vh_
`define RAM_ADDR_BITS 15
`endif
```

Listing 3. Header file to memory RAM definiton

This module also has an interface that allows communication with the Linker. In another XML file the linker provides the necessary information to create the program counter module. With this information, the minimum size required for the program counter (PC) of the system is established. In the Fig. 3, the communication between the Analyzer and the generator can be better observed.

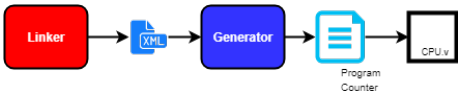

Fig. 3. Communication between Linker and Generator.

Finally, the last module is the Arithmetic Logic Unity (ALU). With the information from the XML file and the template from the processor the module is created. The template provides the base to customize the module. Commenting every operation that the processor does not need the ALU ignores those operations and only use the specific operations.

D. MASTER

The master module is responsible of controlling the order of execution of the other modules. Basically, it asks to the user for the input file, then compile it with the RISCV tool-chain in order to obtain the assembler code file. If there is not any error in the process, it runs the Analysis module over a copy of the file. After that, Master executes the Linker and the Generator, respectively. To finalize, reorder some temporally and output files.

III. RESULTS

The system was proved with three source codes in order to show his behavior and compare the resources consumption with the original processor. The first code (op.c), had some few basic operations. On the other hand, the second and third code (adpcm and blowfish tests) were much more demanding. This tests programs have been taken of CHStone benchmark. CHStone it's a suite of benchmark programs for C-based high-level synthesis and give a baseline to make future comparisons. More about it could be found at [7].

A. Op.c

As first input to the system, as mentioned earlier, it was selected the op.c file, whose code is show in Fig.4. The system made an analysis of this source code, and gives as result an architecture described and defined by files shown in Listing.4 and Listing.5 respectively. The RAM memory for this code was 32KB.

```
void main(void){
    int a = 2;
    int b = 3;
    int c = 0;
    c = a + b;
    c = a & b;
    c = a | b;
}
```

Listing 4. Op.c source code

The synthesis process, made on Quartus Prime of Altera, displays relevant information about the resources who have been used by the customized CPU. These resources are

compared with the resources used by the synthesis of the original CPU in table II.

TABLE II
COMPARISON IN THE USE OF RESOURCES BETWEEN THE ORIGINAL ARCHITECTURE AND THE CUSTOMIZED ARCHITECTURE BASED ON OP.C.

Resources	Original	Customized	Resource savings
Logic units	6 996	4 432	36.6%
Registers used	1 989	1 118	43.7%
Memory bits blocks	524 288	262 144	50%
Synthesis time	08:46	05:55	32.5%

In order to verify that the CPU behavior was not affected by the changes, the resulting system parts were put under a series of simulation with ModelSim of Altera, as is shown in: Fig.4, Fig.5 and Fig.6. All the customized architectures have their own verification process, in the next subsections this part is omitted to focus on the changes originated by the input code.

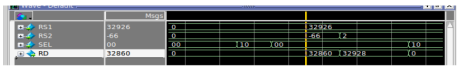

Fig. 4. Simulation of the customized ALU.

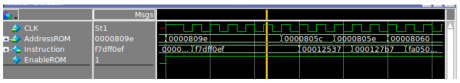

Fig. 5. Simulation of the customized ROM.

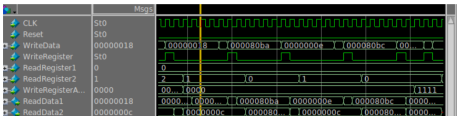

Fig. 6. Simulation of the customized register bank.

B. Adpcm

The second code who was been process by the system was adpcm.c of CHStone benchmark. This code is an adaptive differential pulse code modulation decoder and encoder that implements the CCITT G.722 ADPCM algorithm for voice compression [7]. The RAM memory for this code was 64KB. The synthesis result can be found in table III.

TABLE III
COMPARISON IN THE USE OF RESOURCES BETWEEN THE ORIGINAL ARCHITECTURE AND THE CUSTOMIZED ARCHITECTURE BASED ON ADPCM.C OF CHSTONE.

Resources	Original	Customized	Resource savings
Logic units	6 996	6 646	5%
64 Registers used	1 989	1 299	34.6%
Memory bits blocks	524 288	524 288	0%
Synthesis time	08:46	08:18	5.3%

C. Blowfish

The third and last code tested, was bf.c of Blowfish CHStone benchmark. Blowfish code is a data encryption standard that implements a symmetric block cipher. This program contains only the encryption function [7]. The RAM memory for this code was 128KB. The synthesis result can be found in table IV.

TABLE IV
COMPARISON IN THE USE OF RESOURCES BETWEEN THE ORIGINAL ARCHITECTURE AND THE CUSTOMIZED ARCHITECTURE BASED ON BF.C OF CHSTONE.

Resources	Original	Customized	Resource savings
Logic units	6 996	10 514	-50.3%
Registers used	1 989	1 399	29.7%
Memory bits blocks	524 288	1 048 576	-100%
Synthesis time	08:46	16:30	-88.2%

D. Architectures variations comparison

In this subsection there are graph comparison between the resource consumption of the 3 customized architectures generates for the codes: op.c, adpcm.c and bf.c respectively: Fig.7, Fig.8, Fig.9 and Fig.10.

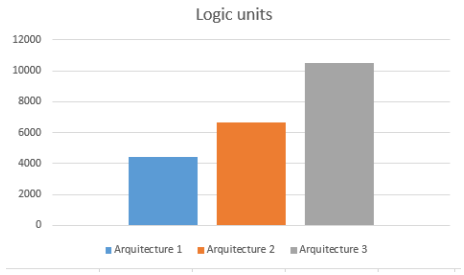

Fig. 7. Logics units comparison between architectures.

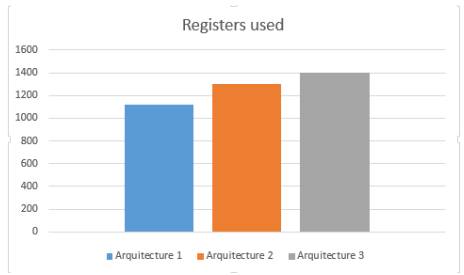

Fig. 8. Registers used comparison between architectures.

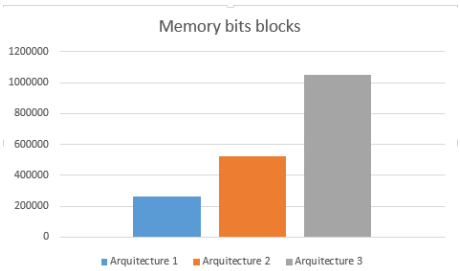

Fig. 9. Memory bits blocks comparison between architectures.

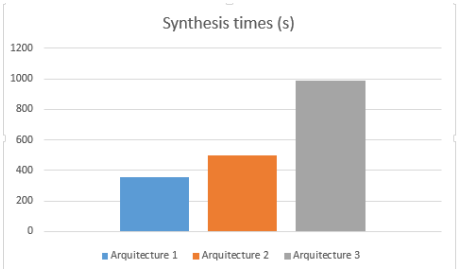

Fig. 10. Synthesis comparison between architectures.

In these images, it's very easy to see the analysis result with the variation originated by the specific hardware resources needs for the execution of each source code, or in other words, the changes of the architecture resulting of the system analysis in order to satisfy the software needs without wasting hardware resources.

IV. CONCLUSIONS

In this work we proposed different architectures for specif applications customizing the hardware modules. We implemented customized processors using C code, compiling it with the RISC-V tool-chain, and using the RISC-V RV32I Instruction Set.

The synthesization times are reduced when obtaining optimizations of the hardware modules. This allows the synthesis process to be done on a smaller amount of hardware. Sometimes the optimizations that are made on the modules are small, but each small optimization in the long run generates optimizations that are clearer and makes the performance better.

This kind of high level synthesis tools simplify in great way the generation of architectures with the specific hardware resources in order to run and specific software application written in C language.

REFERENCES

[1] G. Di Guglielmo, C. Pilato, and L. P. Carloni, "A design methodology for compositional high-level synthesis of communication-centric SoCs," pp. 1–6, ACM, 2014.

[2] S. A. Edwards, R. Townsend, and M. A. Kim, "Compositional dataflow circuits," pp. 175–184, ACM, 2017.

[3] S. Skalicky, T. Ananthanarayana, S. Lopez, and M. Lukowiak, "Designing customized isa processors using high level synthesis," in *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pp. 1–6, Dec 2015.

[4] C. Pilato, F. Ferrandi, and D. Sciuto, "A design methodology to implement memory accesses in high-level synthesis," in *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '11, (New York, NY, USA), pp. 49–58, ACM, 2011.

[5] X. Liu, Y. Chen, T. Nguyen, S. Gurumani, K. Rupnow, and D. Chen, "High level synthesis of complex applications: An h.264 video decoder," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, (New York, NY, USA), pp. 224–233, ACM, 2016.

[6] *The RISC-V Instruction Set Manual, Volume 1: User Level ISA Version 2.1*. Andrew Waterman and Krste Asanovic, 2 ed., 2017.

[7] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii, "Chstone: A benchmark program suite for practical c-based high-level synthesis," in *2008 IEEE International Symposium on Circuits and Systems*, pp. 1192–1195, May 2008.