Two-pass Assembler Design for a Reconfigurable RISC Processor

Sani Irwan Md Salim, Hamzah Asyrani Sulaiman, Rahimah Jamaluddin, Lizawati Salahuddin, Muhammad Noorazlan Shah Zainudin, Ahmad Jamal Salim
Faculty of Electronics & Computer Engineering
Universiti Teknikal Malaysia Melaka
Melaka, Malaysia
sani@utem.edu.my, asyrani@utem.edu.my

Abstract— **Hardware software co-design plays a crucial part in the embedded processor development especially with the current advancement of reconfigurable platforms. The reconfigurability features offered by platforms such as Field Programmable Gate Array (FPGA) has permitted the modification of the internal processor architecture with lower cost and higher performance. While the hardware architecture could be changed through various methods, the modifications need to be complemented with a compatible assembler that suits the amended architecture. This paper presents a two-pass assembler design technique that adapts to any instruction set architecture (ISA) modifications being applied on a reconfigurable processor. A Reduced Instruction Set Computer (RISC) processor core, which is described in Verilog Hardware Description Language (HDL), is used as the testing platform whereby its ISA is expanded to include new instruction sets. The assembler is developed based on two-pass approach and the assembling process would generate a coefficient file that is used as initialization files during the FPGA implementation of the processor core. The assemblers have been successfully developed with correct output format and verified during the FPGA implementation using Xilinx Spartan-3AN board.**

Keywords-assembler; RISC; reconfigurable processor

I. INTRODUCTION

Commercial-of-the-shelf assemblers and compilers are specifically developed for mass-produced microcontrollers that are available in the market. Typically, the assemblers are integrated inside the compilers as part of its integrated design environment (IDE). These IDEs supported higher-level language such as C and BASIC although an assembler is required at the backend of the IDE to generate the object file required in microcontroller programming. For PIC Microcontroller, MPLAB IDE is presented as the official compiler/assembler which is developed by Microchip itself. There are also numerous alternatives that rivaled MPLAB IDE such as MikroC and Sourceboost. Although the level of code optimization is different from one compiler to another, nevertheless, the object file generated from the compilers is still functional.

In embedded processor design, the term soft-core processor reflects the capability of a designer to customize a processor architecture to suit any specific application [1]. The processor's internal configuration could be modified by reprogramming the HDL codes in order to achieve specific goal such as speed performance and area reduction. Moreover, the hardware designer has the capability to expand the memory capacity, ALU operations and to add extra peripherals to the existing architecture. All the changes made are only affecting the hardware design and not the software part. When developing a system that is based on a soft-core RISC processor, once the architecture is amended, inevitably a new assembler is needed in order to accommodate all the changes that are made at the architectural level.

A customizable assembler essentially offers added advantages for a system that is based on a reconfigurable soft-core RISC processor. Any ISA modification or the introduction of new instruction sets are reflected directly in the customizable assembler by the processor's instruction opcodes and formats. The object files of the assembler are generated in coefficient file format to match the format requirement by the soft-core RISC processor during the FPGA implementation.

RISC is a type of microprocessor that has a relatively limited number of instructions. It is designed to perform a smaller number of types of computer instructions so that it can operate at a higher speed. One advantage of RISC is that it can achieved fast instruction execution owing to its simple instruction sets that are simple and basic. Most instructions are completed in one machine cycle, which allows the processor to handle several instructions at the same time through its pipeline.

Essentially, the two-pass assembler is developed using the Visual Basic platform and tokenization technique is applied where each line of assembly code are break into tokens. The text and token type is passed to a parser in order to generate the respective hexadecimal instruction codes. The instruction codes are then formatted to coefficient file (.coe) that is compatible to the FPGA memory block initialization file.

Having a retargetable feature in the assembler could benefit the algorithm developers in evaluating the efficiency of application codes on different architectures. The assembler could be custom-configured to match the core architecture setup hence promoting rapid reconfiguration during the

The authors would like to thank Universiti Teknikal Malaysia Melaka and Ministry of Higher Education Malaysia for the financial support given through the research grant number PJP/2012/FKEKK(45C)/S01050.

algorithm performance assessment. It also offers more architecture exploration for a processor designer. For example, the core architecture could be optimized to execute efficiently for an algorithm in a particular domain in after a successful implementation of the algorithm in the assembling phase. Thus, various application can be applied this algorithm such as microwave and frequency application for extracting system, and others [2, 3].

II. Related Works

Assembler design has been developed as part of a larger tool chain called automatically retargetable code generation tool chain which consists of compiler backend, instruction-set simulator, pre-processor, assembler, linker and debugger [4]. Internal architecture modifications using Architecture Description Language (ADL) constructs have been applied in [4, 5] with the aim to derive architecture-dependent component of a compiler backend. GNU Compiler Collection (gcc) tool is also combined in the compiler development to take advantage of the high-level optimization that is readily available through the GNU tool chain [5, 6].

The reconfigurable architecture in compiler development has been applied to several different platforms. The reconfigurable architecture permitted the ISA modification and also enables users to integrated application-specific hardware module with the processor core. Basic 32/16 bits MIPS architecture in [6] has been used as the reconfigurable architecture while programmable micro-coded controller (PMC) architecture in [7] was chosen due to its high speed performance and covering wide instruction formats. The retargetable compiler development in [8] was focused on dynamic instruction set computer (DISC) processor. DISC processor is a partially reconfigurable FPGA-based processor that combines a programmable core and application-specific hardware modules in order to achieve high performance and architectural flexibility.

In this paper, a processor core called UTeMRISC is used as the target platform in the FPGA implementation phase. The internal architecture of the processor is shown in Fig. 1. UTeMRISC is a 16-bit soft-core processor and its architecture is described in Verilog HDL [9]. By utilizing RISC and Harvard architecture, UTeMRISC provides opportunity to extend the instruction set and also to modify its ISA to suit any specific application.

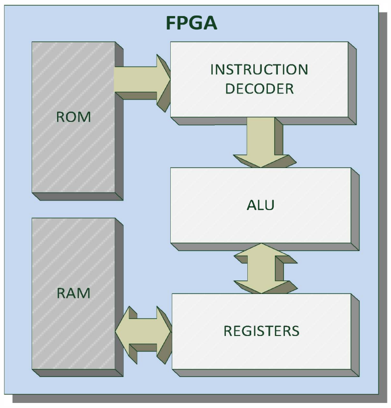


Figure 1. Block Diagram of the soft-core RISC Processor

III. Methodology

A. Custom Instruction Set Generation

Modification of the processor core architecture is implemented based on the application specific instruction set processor (ASIP) design methodology. In ASIP, the ISA of the processor core is configured to include new instructions that ultimately optimized the operation of a specific application. The custom instruction set generation can be categorized into two approaches; full and partial customization [10]. Full instruction set customization requires a complete overhaul on the instruction set architecture, instruction's opcode and the related instruction commands from ground-up. On the other hand, the partial customization approach only involved generating several new instruction sets on top of the existing instructions that are already established in the original RISC processor core. Therefore, only several new instructions, which are closely related to a specific application, are added to the processor core. Furthermore, modifications on the other existing instruction sets are also possible in order to optimize its functionality and to avoid any redundancy during the instruction execution. In this paper, partial modification technique has been applied to the ISA with three new instructions are added to the instruction set list.

B. Instruction Set Architecture

The ISA essentially is an instruction format that consists of instruction's opcode, data or register file address and control bit such as directional bit or selection bit. In general, the instruction sets are categorized into three formats as shown in Fig. 2. Each format consists of different configuration of ISA that is reflected by its bit positioning according to the designated instruction set category.

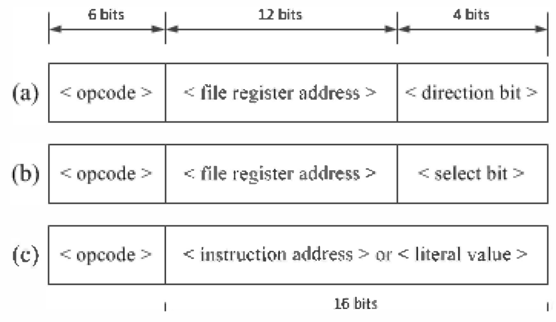


Figure 2. ISA Formats (a) Byte-oriented operation, (b) bit-oriented operation and (c) literal and control operation

The instruction's opcode size is fixed to 6-bit wide to ensure better organization during the tokenization process and lexical analysis. Each instruction sets are given its own unique opcode according to its instruction format. Indirectly, the opcodes' bit size indicated that the total instruction set for the processor is capped at 64 instruction sets, which is adequate considering the core is a RISC-type processor.

The target RISC processor platform is comprised of 16-bit data bus which directly enables the processor to handle a 16-bit wide data at any particular time. To accommodate the data width requirement, the ISA is designed to include all 16-bit data together with the instruction's opcode and any control bit associated with the particular instruction set.

Currently, a total of 35 instruction sets are listed in Table I and there are plenty of allocations available to add any new instruction sets that would directly improve the processor execution on a specific application.

C. Two-pass Assembler Design

The construction of the assembler is developed by using two-pass encoding. Two-pass assembler would generate an intermediate file during the instruction set encoding. In the first pass, the program file is analyzed thoroughly to identify several elements such as labels, comments and blanks. Then, an intermediate file called INT file is created that contains organized and uncluttered instruction line. In the second pass, the assembler will reread the INT file and directly encodes each line of instruction with the help of symbol tables that are already established during the first pass. By the end of second pass, the object file and the coefficient file are generated. As the goal of the assembler is to create the object file and the coefficient file, the execution time for the assembler to encode all the instruction sets is measured. The resulting elapsed time is recorded as performance indicator for the assembler's execution speed.

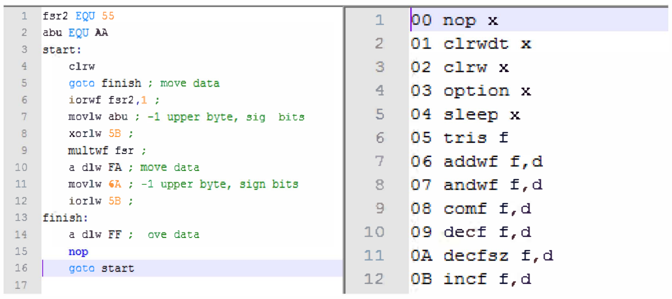
TABLE I. LIST OF INSTRUCTION SET

No.	Mnemonics	Instruction Code		Status
00	nop	000000	0000_0000_0000_0000	Existed
01	clrwdt	000001	0000_0000_0000_0000	Existed
02	clrw	000010	0000_0000_0000_0000	Existed
03	option	000011	0000_0000_0000_0000	Existed
04	sleep	000100	0000_0000_0000_0000	Existed
05	tris f	000101	0000_0000_0000_0000	Existed
06	addwf f,d	000110	ffff_ffff_ffff_fffd	Existed
07	andwf f,d	000111	ffff_ffff_ffff_fffd	Existed
08	comf f,d	001000	ffff_ffff_ffff_fffd	Existed
09	decf f,d	001001	ffff_ffff_ffff_fffd	Existed
0A	decfsz f,d	001010	ffff_ffff_ffff_fffd	Existed
0B	incf f,d	001011	ffff_ffff_ffff_fffd	Existed
0C	incfsz f,d	001100	ffff_ffff_ffff_fffd	Existed
0D	iorwf f,d	001101	ffff_ffff_ffff_fffd	Existed
0E	rlf f,d	001110	ffff_ffff_ffff_fffd	Existed
0F	rrf f,d	001111	ffff_ffff_ffff_fffd	Existed
10	subwf f,d	010000	ffff_ffff_ffff_fffd	Existed
11	xorwf f,d	010001	ffff_ffff_ffff_fffd	Existed
12	bcf f,b	010010	ffff_ffff_ffff_bbbb	Existed
13	bsf f,b	010011	ffff_ffff_ffff_bbbb	Existed
14	btfsc f,b	010100	ffff_ffff_ffff_bbbb	Existed
15	btfss f,b	010101	ffff_ffff_ffff_bbbb	Existed
16	andlw k	010110	kkkk_kkkk_kkkk_kkkk	Existed
17	call k	010111	kkkk_kkkk_kkkk_kkkk	Existed
18	goto k	011000	kkkk_kkkk_kkkk_kkkk	Existed
19	iorlw k	011001	kkkk_kkkk_kkkk_kkkk	Existed
1A	movlw k	011010	kkkk_kkkk_kkkk_kkkk	Existed
1B	retlw k	011011	kkkk_kkkk_kkkk_kkkk	Existed
1C	xorlw k	011100	kkkk_kkkk_kkkk_kkkk	Existed
1D	clrf f	011101	ffff_ffff_ffff_ffff	Existed
1E	movfw f	011110	ffff_ffff_ffff_ffff	Existed
1F	movwf f	011111	ffff_ffff_ffff_ffff	Existed
20	multw f	100000	ffff_ffff_ffff_ffff	New
21	swapn f	100001	ffff_ffff_ffff_ffff	New
22	swapfw f	100010	ffff_ffff_ffff_ffff	New

D. Object Code Assembly and Conversion

The two-pass assembler requires two input files to be included prior to the assembly process. Processor's opcode file (*.op) and the assembly program file (*.asm) are loaded to the assembler through file selection dialog. The opcode file contains a list of all the instruction sets together with its opcode and instruction format. The assembly program file contains the programming code written in assembly language. The Examples of both files are shown in Fig. 3. The assembly process is started once both files are valid. Fig. 4 shows the overall assembly process of the assembler design.

The instruction sets' opcode, mnemonics and formats form the opcode files are first loaded to hash tables at the start of the assembling process. Then, each line of code from the assembly program file is read by the lexical analyzer. The lexical analyzer will divide this string into tokens. Segregation between instruction's mnemonics, data/operand and comment section are identified by using special character, comma or whitespace. The white space, newline, tab, and other characters are used by the lexical analyzer to separate each line of assembly code into tokens. The lexical analyzer then reads every token and special character such as blanks and white spaces are ignored. Comments, which are preceded by a semicolon sign, are also treated as white spaces and are eliminated by the lexical analyzer.



```
fsr2 EQU 55
abu EQU AA
start:
    clrw
    goto finish ; move data
    iorwf fsr2,1 ;
    movlw abu ; -1 upper byte, sig  bits
    xorlw 5B ;
    multwf fsr ;
    a dlw FA ; move data
    movlw 6A ; -1 upper byte, sign bits
    iorlw 5B ;
finish:
    a dlw FF ;  ove data
    nop
    goto start

```

```
00 nop x
01 clrwdt x
02 clrw x
03 option x
04 sleep x
05 tris f
06 addwf f,d
07 andwf f,d
08 comf f,d
09 decf f,d
0A decfsz f,d
0B incf f,d
```

Figure 3. Example of the assembly program file (left) and the opcode file (right)

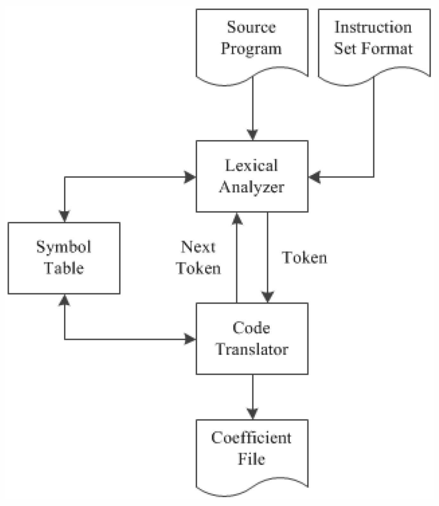


Figure 4. Overall Assembler Design Flow

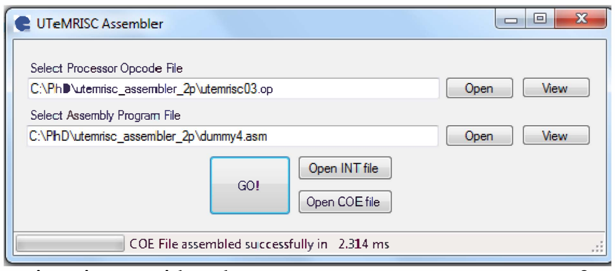
The main function of the code translator is to decode all the tokens to their respective instruction code according to the tokens' type. All the instruction sets' mnemonics and labels are referred to the symbol table and then are converted to their opcode values. Meanwhile, the operands token are decoded referring to the instruction set's format retrieved from the hash table. Therefore, each line of assembly code consisted of 22 bits instruction code which include the instructions opcode, operand (file register address or memory location) and control bit (directional bit or bit select). Additional program directive instructions such as equate (EQU) and end-of-file (END) could also be included in the assembly program file but it will never be considered as instruction set and thus will not be assembled. All completed instruction codes are stored in a hash table called 'table_InstCode'. Essentially, the table contains only line count number and the instruction code for the respective line count. For the object file, the assembler generates a coefficient which is created using the data in the 'table_InstCode'. The data are organized in a specific format to conform to the requirement during the initiation process of ROM module during the FPGA implementation of the processor core.

To gauge the performance of a retargetable assembler, the time taken to complete the entire process of tokenization, lexical analyzer and code translation are measured and recorded. The elapsed time is realized using a stopwatch function that provides accurate timing in order to determine the execution time of the assembler. Although the stopwatch function is considered as an accurate measurement for the execution times, the yielded results may vary from time to time due to the dependencies in on the computer's hardware and operating system (OS). Therefore, a loop mechanism is executed to invoke the processes in multiple times and subsequently obtained the average elapsed time for the process execution.

E. FPGA Implementation

The ROM module in the RISC processor core is instantiated by invoking the IP CORE Generator (COREgen) program which is a part of Xilinx software. The ROM module is configured as a block memory core with single port and the memory size is set to 22 bits on width and 2048 on depth. This indicated that the ROM module is capable in handling a total of 2048 instruction address. Ultimately, the ROM module size could be expanded further more. However, the ROM's size is bounded to the width of the program counter (PC) register. A larger ROM size would require an expansion on the PC register in order to accommodate all the instruction address. Each instruction address is consisted of 22-bit code which is in line with the modified ISA.

During the memory initialization, the coefficient file is loaded to the module. Once the coefficient file is initialized in the ROM module, the whole RISC processor core is implemented in the FPGA. A behavioral simulation process is conducted in order to verify the syntax and the functionality of the design without any timing information. The waveform signals from the processor core are observed using the ISim simulator to verify the correct instruction code obtained by the processor core during the program execution.



UTeMRISC Assembler

Select Processor Opcode File: C:\PhD\utemrisc_assembler_2p\utemrisc03.op

Select Assembly Program File: C:\PhD\utemrisc_assembler_2p\dummy4.asm

COE File assembled successfully in 2.314 ms

Figure 5. User Interface of the Two-pass Assembler

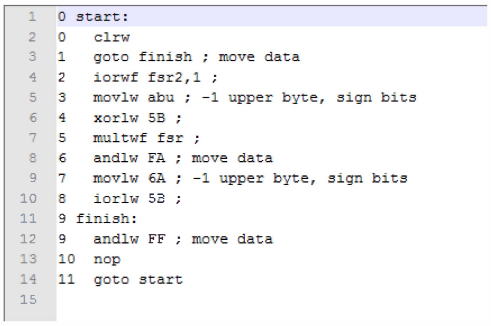
IV. RESULTS & DISCUSSION

A. Graphical User Interface (GUI)

As the assembly program file and the opcode file are text-based file, a lot of string manipulation techniques are adopted in order to perform the required functionalities during the assembling and conversion process. Fig. 5 shows the graphical user interface for the customizable assembler.

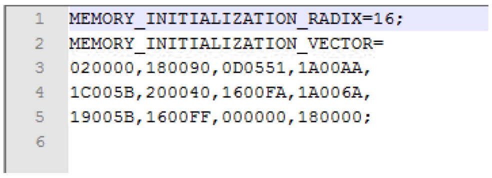
Firstly, user is required to select the processor opcode file, designated with file extension '.op'. Then, the assembly program file with extension '.asm' is loaded to the assembler. The instruction set used in the assembly program must adhere to the instruction set list as mentioned in Table 1. The 'View' button is available to examine the content of both processor opcode and assembly program files in Notepad application.

Button ‘Go’ is pressed to start the assembling process. Status bar indicator would indicate the progress during the execution of the processes. As for reference, the elapsed time for the assembler operation is displayed at the status bar. The elapsed time is measured from the start of the input files initiation until the generation of the coefficient file. To reflect the accuracy of the elapsed time, the assembling process is repeated several times and the average time is calculated. To observe the generated coefficient file, the ‘Open COE file’ button is pressed and the coefficient file will be opened through the Notepad application. User can compare and cross-check the coefficient file output with the listing file output to verify it accuracy. Fig. 6 and Fig. 7 show the INT file and the coefficient file output respectively. The average elapsed time from start to complete is recorded at 2.22 ms which make the average execution time for each instruction is 0.2 ms.



```
0 start:
0   clrw
1   goto finish ; move data
2   iorwf fsr2,1 ;
3   movlw abu ; -1 upper byte, sign bits
4   xorlw 5B ;
5   multwf fsr ;
6   andlw FA ; move data
7   movlw 6A ; -1 upper byte, sign bits
8   iorlw 5B ;
9 finish:
9   andlw FF ; move data
10  nop
11  goto start
```

Figure 6. INT File



```
MEMORY_INITIALIZATION_RADIX=16;
MEMORY_INITIALIZATION_VECTOR=
020000,180090,0D0551,1A00AA,
1C005B,200040,1600FA,1A006A,
19005B,1600FF,000000,180000;
```

Figure 7. The Coefficient File

B. *FPGA Implementation*

After successful synthesis process, the RISC processor architecture is translate, map and place-and-route before being programmed in the FPGA. Fig. 8 shows the waveforms of selected internal signals observed during the implementation of the RISC processor core. In this case, the test program’s instruction flow is monitored to verify whether the RISC processor is capable to fetch, decode and execute all of the instructions correctly and in timely manner.

As shown in Fig. 8, the test program is implemented by the RISC processor core in a correct sequence as defined in the coefficient file. The RISC processor has successfully read and decoded the instruction codes generated by the two-pass assembler and executed the instructions properly. For new instruction set such as ‘multwf’, the RISC processor core has able to decode the 22 bits instruction code and perform the multiplication process as instructed.

C. *Discussion*

Referring to the internal working operation of the RISC processor core, the instructions are fetched from ROM module pointed by the PC register. Then, the instruction codes are decoded in order to identify each operation and type of operand involved. The instruction decoding process is performed by the decoder module which depends on to the ISA setup during the RISC processor HDL design. Accordingly, both opcode file and instruction decoder module must have identical set of instruction list in order to successfully decode and assemble any test program during the FPGA implementation. The output results are observed as internal processor signals and could be sourced out to the input/output ports depending on the processor core design.

Based on the assembler design and the elapsed time, two-pass assembler offered more organized and simpler assembling process. Symbol identification and table of references are completed in the first pass and it makes the assembling process in the second pass ensued smoothly. Nonetheless, having an intermediate file required additional read/write processes to generate and it does abound to affect the elapsed time of the assembler execution.

V. CONCLUSION

A customizable assembler is an important component when modifying the ISA of a RISC processor core. The introduction of new instruction set that is tailored-made for a specific application would directly improve certain aspect of the processor’s performance. Thus, to implement the new instruction set, the RISC processor architecture would require a compatible assembler and also an instruction decoder module with identical data set of instruction opcodes. The two-pass, retargetable assembler design in this paper has the ability to adopt the changes made in the ISA through the instruction opcode file. The implementation of the RISC processor on the FPGA chip has produced correct program execution of the test program that is assembled by the assembler. The assembler design techniques would be a good platform in order to develop a full-scale compiler that supports higher level language for the ASIP processor design in the future.

ACKNOWLEDGMENT

The authors wish to thank all members of the FPGA Group and lab technicians at the Microelectronics Lab in FKEKK, UTeM whose suggestions and assistance have helped to improve the quality of this work.

REFERENCES

[1] A. J. Salim, S. I. M. Salim, N. R. Samsudin, and Y. Soo, "Conversion of an 8-bit to a 16-bit Soft-core RISC Processor," *International Journal of Electronics Communication and Computer Technology,* vol. 3, pp. 393-397, 2013.

[2] M. A. Othman, M. Z. A. A. Aziz, and M. Sinnappa, "The exposure of Radio Frequency (RF) radiation on pharmaceuticals medicine for RF application," 2012, pp. 40-44.

[3] M. M. Ismail, M. A. Othman, H. A. Sulaiman, M. H. Misran, R. H. Ramlee, A. F. Z. Abidin, *et al.*, "Firefly algorithm for path optimization in PCB holes drilling process," 2012, pp. 110-113.

[4] L. Taglietti, J. Filho, D. Casarotto, O. Furtado, and L. dos Santos, "Automatic ADL-Based Assembler Generation for ASIP Programming Support," in *Embedded Computer Systems: Architectures, Modeling, and Simulation.* vol. 3553, T. Hämäläinen, A. Pimentel, J. Takala, and S. Vassiliadis, Eds., ed: Springer Berlin / Heidelberg, 2005, pp. 341-350.

[5] F. Brandner, D. Ebner, and A. Krall, "Compiler generation from structural architecture descriptions," in *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2007, pp. 13-22.

[6] A. L. Rosa, L. Lavagno, and C. Passerone, "A software development tool chain for a reconfigurable processor," in *International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2001, pp. 93-98.

[7] R. Leupers and P. Marwedel, "Instruction-set modelling for ASIP code generation," in *VLSI Design, 1996. Proceedings., Ninth International Conference on*, 1996, pp. 77-80.

[8] D. A. Clark and B. L. Hutchings, "Supporting FPGA microprocessors through retargetable software tools," in *FPGAs for Custom Computing Machines, 1996. Proceedings. IEEE Symposium on*, 1996, pp. 195-203.

[9] A. J. Salim, N. R. Samsudin, S. I. M. Salim, and Y. Soo, "Multiply-Accumulate Instruction Set Extension in a Soft-core RISC Processor," in *IEEE International Conference on Semiconductor Electronics (ICSE 2012)*, 2012.

[10] C. Galuzzi and K. Bertels, "The Instruction-Set Extension Problem: A Survey," *ACM Trans Reconfigurable Technol. Syst.*, vol. 4, pp. 1-28, 2011.