One-pass Assembler Design for a Low-end Reconfigurable RISC Processor

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

Abstract— **Implementation of processor core on a programmable device such as Field Programmable Gate Array (FPGA) has been widely adopted by researchers due to its flexibility and hardware reconfigurability. However, with processor design is a tightly integrated development of hardware and software, changes in the processor's hardware architecture would require the same alterations being made on the software side. This paper presents a one-pass assembler design technique that adapts to modifications of the instruction set architecture (ISA) 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 perform the instruction set extension. A lexical analyzer and tokenization technique is adopted in the assembler development with several hash tables are setup to store all the tokens. The assembler would generate a coefficient file that contained all the translated instruction codes sourced from an assembly program. The coefficient file then is initiated in the memory module of the RISC processor core using Xilinx Spartan-3AN FPGA board. Based on the simulation results, the assemblers have been successfully developed with working coefficient file output format that matched to the ISA modifications.**

Keywords-assembler; RISC; reconfigurable processor

I. INTRODUCTION

The conventional system that utilized discrete microcontrollers has been well known for its great support and simple board construction. However, the advancement of reconfigurable platforms such as FPGA has facilitated the development of processor-based system to become more cost-efficient and easily customizable. On-the-fly programming and customizable features could easily be added to the system without major modification of the circuit design. Furthermore, design prototyping and circuit debugging could be implemented at much lower cost and flexibility.

A synthesizable processor called soft-core processor is required to embed a processor inside an FPGA. A soft-core processor essentially is a processor architecture that is described in hardware description language (HDL) such as Verilog or VHDL. The architecture could be modified by reprogrammed the HDL code and implement it to the FPGA fabric. Thus, the system designer has the freedom to modify any part of the processor's architecture such as memory expansion and instruction set extension in order to optimize the device utilization or to gain other performance advantages. Conversely, any modifications that are implemented in the processor's architecture would make the existing software tool chain, which is provided by the processor's vendor, to be rendered unusable.

There are a few researches that involved in developing a software tool chain for reconfigurable processor around several platforms. A customizable framework for assembler design and code simulator is proposed in [1] which is implemented on C++ programming language. In order to minimize the development cycle, architecture description language (ADL) is proposed by [2] to form a retargettable framework. The framework also includes generating development tools such as simulators, assemblers and debuggers through Eclipse environment for an embedded system design.

The reconfigurable procedures in processor's architecture will inevitably lead to an approach called application specific instruction set processor (ASIP). Part of the reconfiguration would include instruction set customization and also ISA modification. There are retargettable assembler/compilers were developed based on simplified Directed Acyclic Graph-based recursive mechanism [3] and also ADL construct [4] to address ASIP instruction set customization. For different sizes of ISA, employing ADL proved to generate low processing time and gradually consistent with the number of assembled instruction [4]. Assembler design in ASIP also involved optimization techniques in order to produce efficient code design. Apart from ADL, an XML-based hardware description language [5] is also introduced. The ISA and the microarchitecture of the processor have separate descriptions for the instruction set architecture (ISA) and it is compatible to be interface with the existing hardware description language. Optimization method such as peephole optimization [6] and combination of backend and gcc compiler [7] have been proposed with significant results on the assembler runtime and speed performance.

The commercially-of-the-shelf assembler/compiler are developed by the chip manufacturers and are targeted to a their own processor family [8, 9]. Although the tool chain is highly

978-1-4799-3704-2/14/$31.00 ©2014 IEEE

optimized for the particular processors, users could not customize the assembler configuration due to licensing and copyright issue. Therefore, a customizable assembler design is essential to generate the required object file for the processor programming once the processor's architecture is modified.

In this paper, the low-end reconfigurable RISC processor is identified as a processor core called UTeMRISC02 [10] is used as the target platform in the FPGA implementation phase. The internal architecture of the processor is shown in Fig. 1. UTeMRISC02 is a 16-bit soft-core processor and its architecture is described in Verilog HDL.

II. METHODOLOGY

A. *UTeMRISC02 Processor Soft-core*

The previous iteration of UTeMRISC02 processor [11] consisted of basic instruction sets that fulfilled it purposed as general-purpose 8-bit RISC processor. There are a total of 33 instruction sets that are available for the user to program the processor. In reality, not all of the instruction sets are utilized during the implementation of any specific program. Thus, there are opportunities to further improve the processor's architecture by optimizing the available instructions to include only the required operations. Furthermore, redundancy in instruction set execution could also be addressed by creating a new instruction set to combine the all the operations in a single instruction. This process, called instruction set extension is widely adopted as part of the ASIP methodology.

The instruction set extension process mainly involved establishing a new instruction set architecture (ISA) in the RISC processor core. Fundamentally, ISA distinguished several key points in operation between the hardware and software of a processor core. ISA determined the total number of instruction available, the type and location of instruction operand and the size of the instruction format. In the processor core module, the ISA is reflected in the instruction decoder module whereby the all the instruction fetched from the assembly program are decoded in the instruction decoder module which is setup based on the ISA configuration. The decoded instruction is then stored in the instruction register (IR) to be executed by the ALU accordingly.

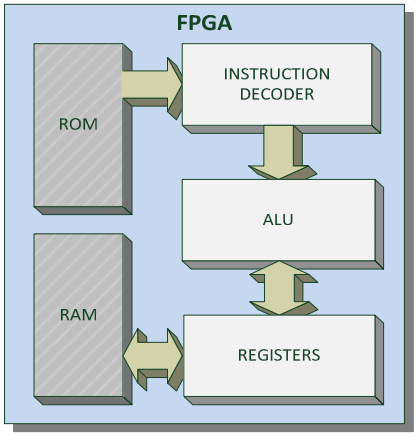

Fig. 1: Block Diagram of UTeMRISC02

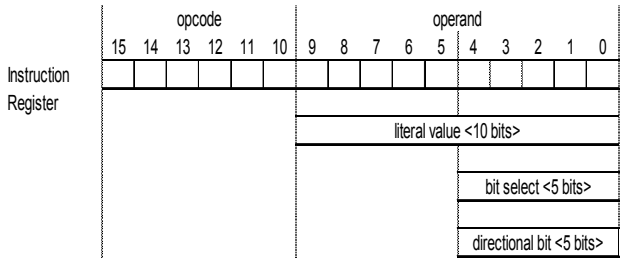

Fig. 2: The ISA Format

TABLE I
LIST OF ALL INSTRUCTION SETS

Mnemonic	Description	Modified-16 bit opcode			
ADDWF	Add W and f	0001	11ff	fffd	dddd
ANDWF	AND W with f	0001	01ff	fffd	dddd
CLRF	Clear f	0111	00ff	ffff	ffff
CLRW	Clear W	0000	0100	0000	0000
COMF	Complement f	0010	01ff	fffd	dddd
DECF	Decrement f	0000	11ff	fffd	dddd
DECFSZ	Decrement f, Skip if 0	0010	11df	fffd	dddd
INCF	Increment f	0010	10ff	fffd	dddd
INCFSZ	Increment f, Skip if 0	0011	11ff	fffd	dddd
IORWF	Inclusive OR W with f	0001	00df	fffd	dddd
MOVWF	Move W to f	0110	11ff	ffff	ffff
NOP	No Operation	0000	0000	0000	0000
RLF	Rotate left f through Carry	0011	01ff	fffd	dddd
RRF	Rotate right f through Carry	0011	00ff	fffd	dddd
SUBWF	Subtract W from f	0000	10ff	fffd	dddd
SWAPF	Swap f	0011	10ff	ffff	ffff
XORWF	Exclusive OR W with f	0001	10ff	fffd	dddd
BIT-ORIENTED FILE REGISTER OPERATIONS					
BCF	Bit Clear f	0100	00ff	fffb	bbbb
BSF	Bit Set f	0100	01ff	fffb	bbbb
BTFSC	Bit Test f, Skip if Clear	0100	10ff	fffb	bbbb
BTFSS	Bit Test f, Skip if Set	0100	11ff	fffb	bbbb
LITERAL AND CONTROL OPERATIONS					
ANDLW	AND literal with W	0110	01kk	kkkk	kkkk
CALL	Call subroutine	0101	01kk	kkkk	kkkk
CLRWDT	Clear Watchdog Timer	0111	1100	0000	0000
GOTO	Unconditional branch	0101	10kk	kkkk	kkkk
IORLW	Inclusive OR Literal with W	0110	00kk	kkkk	kkkk
MOVLW	Move Literal to W	0101	11kk	kkkk	kkkk
OPTION	Load OPTION register	1000	1000	0000	0000
RETLW	Return, place Literal in W	0101	00kk	kkkk	kkkk
SLEEP	Go into standby mode	1000	0000	0000	0000
TRIS	Load TRIS register	1000	0010	0000	0000
XORLW	Exclusive OR Literal to W	0110	10kk	kkkk	kkkk

The ISA modification is made by extending the instruction code length to 16 bits compared to 12 bits in the previous iteration of UTeMRISC01 processor. Therefore, the IR is composed of 6-bit opcode, 5-bit operand and 5-bit directional/bit select as shown in Fig. 2. Using the new configuration, there are possibility to employ up to 64 instruction sets and 32 general-purpose registers. As a soft-core processor, the ISA configuration could be easily extended further to suit the requirement for any specific application by reprogramming the instruction decoder module.

The overall instruction sets are categorized according to the ISA format. The instruction opcode would determine the instruction type and also the necessary operands before decoding process is executed. Table 1 shows all the available instruction sets with the modified ISA format according to the instruction's category.

B. One-pass Assembler Design Technique

The assembler design could be commenced once the ISA has been established. For UTeMRISC02, the assembler is developed using one-pass encoding techniques. In the one-pass assembler, the instruction set encoding is executed in the first pass itself. Each line of instruction is encoded immediately by referring to various tables of references that include symbol table and instruction opcode table [12, 13]. Only single pass of the program file is required to generate a complete coefficient file as shown in Fig. 3.

At the start of the assembly process, the instruction sets' opcode, mnemonics and operands' type are first loaded to a hash table called 'inst_code'. Then, the lexical analyzer [14] reads each line of instruction from the assembly program file and divides it into tokens. Each token is recognized as either the instruction mnemonics, or a data/operand or labels. All comments, which are preceded by a semicolon sign, are also treated as white spaces and are eliminated by the lexical analyzer.

Implementation of one-pass assembler will inevitably lead to the forward referencing issue. In order to overcome this problem, the one-pass assembler has a hash table called 'symbol_table' that stored the labels and its corresponding line count (LC) or literal value in for reference during code translation process. Once the token is recognized, the code translator will decode each token to their respective instruction code by referring to the 'inst_code' hash table. Instruction mnemonics, labels and operands refer to the symbol table and then are converted to their opcode values. In accordance to the ISA format, each line of assembly program will generate a 16-bit instruction code that consisted of the instruction opcode, operand (file register address or memory location) and control bit (directional bit or bit select).

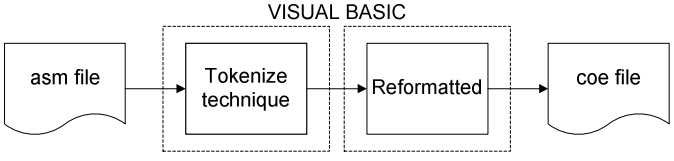

Fig. 3: The Assembler Design Flow

Under normal circumstances, the labels are defined prior to it being used in the assembly program. However, when a label is detected but yet to be defined in the symbol table, it indicates that the label symbol will be found in the later part of the assembly program. Instantly, the undefined label is stored in the symbol table with flag X that indicates that the label is yet to be defined. The assembler will keep track of the symbol table and constantly updating its entries once the undefined label is properly addressed in the subsequent instructions. When the end-of-file (EOF) is reached, the symbol table's entries are supposed to be cleared of X flags in its data field or else an error message would be generated implying that there are unrecognized labels that are yet to be declared.

All translated instruction codes are stored in a hash table called 'table_InstCode'. The hash table contains only a line count number and the instruction code for the respective line count. The coefficient file is generated by utilizing all the data available in 'table_instCode' and rearranged the data specific format to conform to the requirement during the initiation process of ROM module during the FPGA implementation of the processor core. The resulting elapsed time is also recorded as a performance indicator for the assembler's execution speed, which is measured from the start of the tokenization process until the coefficient file is successfully generated.

C. Initiation of Coefficient File in ROM

To verify the coefficient file that is generated by the assembler, the file must be initiated in a read-only memory (ROM) module of the UTeMRISC02 processor core. As all modules are instantiated in the FPGA platform, the generation of ROM module is done by CoreGen program provided in the Xilinx ISE suite. The ROM module specification includes 16 bits in width which matches the IR data width. The coefficient file content is loaded in the ROM module during the initialization process. Then, the other modules in the UTeMRISC02 processor core are instantiated and implemented in the FPGA. Each instruction code in the ROM module is fetched, decoded by the instruction decoder module and finally executed by the ALU module. The instruction pipelining process is repeated until all end-of-file is reached. A behavioral simulation process is conducted in order to verify the syntax and the functionality of the design. The waveform signals from the processor core are observed using the ISim simulator to verify the correct instruction code obtained from the processor core during the program execution.

III. RESULTS & DISCUSSION

In this project the Visual Basic is used as the platform to compile and convert the assembly language code directly to coefficient file. Fig. 4 shows the interface for the compiler that have been created in Visual Basic which consist of a file directory, a button to browse the file, a button to view the file in Notepad application and a button to assemble the file. The 'Browse' button is used to find the directory for the assembly code file, which will be compiled by the assembler. After the file is selected the file directory is shown in the text box. This file is viewed in Notepad application by pressing the button 'View file'. This file is converted from assembly code to coefficient file by pressing the 'Compile' button.

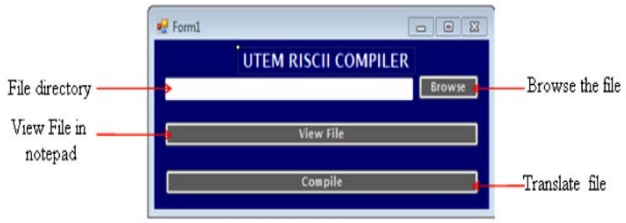

Fig. 4: User Interface of the Assembler

The assembly file and the coefficient file output are shown in Fig. 5 and Fig. 6 respectively. The average elapsed time from start to complete is recorded at 1.16ms, which is calculated over several loops of assembler execution.

The waveforms of internal signals are observed during the implementation of the UTeMRISC02 processor core. The signals are the corresponding internal connection between registers and modules inside the UTeMRISC02 processor core during its execution. 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 the instructions correctly and in a timely manner.

As shown in Fig. 7, the simulated result shows the correct sequences output. The sequences of instruction in the assembly program file are shown at waveform 'inst_string(63:0)' meanwhile the sequence of the coefficient file is shown at waveform 'inst(15:0)'. It means that the assembler has produced a correct sequence as defined in the coefficient file and successfully executed by the RISC processor.

The obvious advantage of one-pass assembler is the ability to decode the assembly program file in a single-read procedure. The forward referencing issue is solved using a linked list structure that housed in an array of the undefined symbols. However, the resource requirement on one-pass assembler is significantly higher due to allocation needed for all the hash tables in storing the reference variables and the instruction codes.

dummy3_2.asm

```
; start :
andlw FA ; move data
movlw 6A ; 1 upper byte
iorlw 5B ;
xorlw 7C ;
andlw 45 ;
movlw 78 ;
iorwf fsr,1 ;
subwf tmr0,0 ;

; start :
iorlw 67 ;
xorlw 1A ;
andlw BB ;
movlw AA ;
iorlw EE ;
xorlw 55 ;
bsf indf,6 ;
btfsc pcl,7 ;
; xorlw 7C,5 ;
```

Fig. 5: The Assembly Program File

dummy3_2.coe - Notepad

File Edit Format View Help

```
MEMORY_INITIALIZATION_RADIX=16;
MEMORY_INITIALIZATION_VECTOR=
64FA,5C6A,605B,687C,
6445,5C78,1081,820,
6067,681A,64BB,5CAA,
60EE,6855,4406,4847;
```

Fig. 6: The Coefficient File

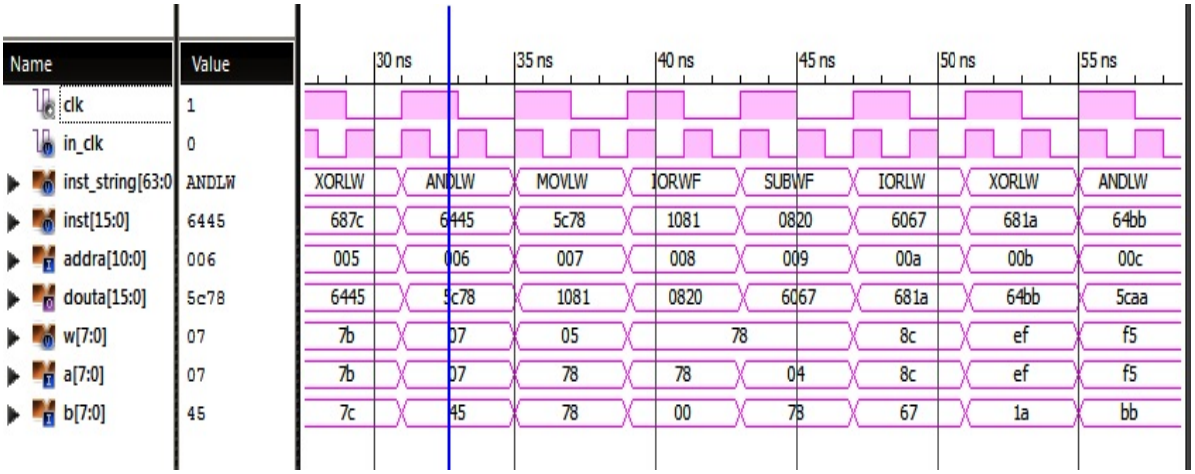

Fig. 7: UTeMRISC02 Simulation Result

IV. CONCLUSION

A reconfigurable processor offers a lot of customization opportunities for the any circuit designer to play around with the processor architecture and achieve extra performance gain, speed increase or optimum resource utilization. However, to implement the modified architecture would require an assembler that matched the new configuration of the processor. This paper has demonstrated the assembler design technique to accommodate instruction set extension of a reconfigurable soft-core processor. A one-pass assembler is developed to streamline the decoding procedures from the assembly program file to the generation of the coefficient file. The output file is verified during the processor core implementation in FPGA platform with successful implementation of all the instruction sets. The assembler design will be a part of the future work in developing a fully customizable software tool-chain for the UTeMRISC02 platform.

ACKNOWLEDGMENT

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.

REFERENCES

[1] A. Vasudeva, A. K. Sharma, and A. Kumar, "Saksham: Customizable x86 Based Multi-Core Microprocessor Simulator," in *International Conference on Computational Intelligence, Communication Systems and Networks*, 2009, pp. 220-225.

[2] J. C. Metrolho, C. A. Silva, C. Couto, and A. Tavares, "Retargetable frameworks for embedded systems exploration," in *IEEE International Conference on Industrial Technology*, 2006, pp. 2223-2227.

[3] H. Kultala, P. Jaaskelainen, and J. Takala, "Operation set customization in retargetable compilers," in *Signals, Systems and Computers (ASILOMAR), 2011 Conference Record of the Forty Fifth Asilomar Conference on*, 2011, pp. 761-765.

[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] S. P. Seng, K. V. Palem, R. M. Rabbah, W. F. Wong, W. Luk, and P. Y. K. Cheung, "PD-XML: Extensible Markup Language for Processor Description," *Field-Programmable Technology,* pp. 437-440, 2002.

[6] L. Songchao, T. Huailiang, and H. Lei, "An embedded cross-assembler optimization method," in *Computer Science and Information Processing (CSIP), 2012 International Conference on*, 2012, pp. 641-644.

[7] 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.

[8] Mikroelektronika. (2013, 30 September). *MikroC Pro for PIC*. Available: http://www.mikroe.com/mikroc/pic/

[9] C. Inc. (2013, 30 September). *Code Optimizing C Compiler for Microchip PIC® and dsPIC® DSCs*. Available: http://www.ccsinfo.com/content.php?page=compilers

[10] A. J. Salim, N. R. Samsudin, S. I. M. Salim, and S. Yewguan, "Modification of Instruction Set Architecture in a UTeMRISCII Processor," *International Journal of Computer Trends and Technology (IJCTT),* vol. 4, pp. 1196-1201, 2013.

[11] N. R. Samsudin, S. I. M. Salim, and A. J. Salim, "Designing UTeMRISCII Processor for Multiply-Accumulate Operation," in *3rd International Conference on Engineering and ICT (ICEI2012)*, 2012, pp. 88-91.

[12] D. Solomon, *Assemblers and Loaders*: Prentice Hall, 1993.

[13] K. Cooper and L. Torczon, *Engineering a Compiler*, 2nd ed.: Morgan Kaufmann, 2011.

[14] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, & Tools*, 2nd ed.: Addison Wesley, 2007.